15

Multiple inheritance

Full application of inheritance requires an important extension to the framework defined in the preceding chapter. In studying the basics of the mechanism we have encountered the notion that a class may need more than one parent. Known as multiple inheritance (to distinguish it from the more restrictive case of *single* inheritance), this possibility is necessary to build robust object-oriented architectures by combining different abstractions.

Multiple inheritance, in its basic form, is a straightforward application of the principles of inheritance already seen; you just allow a class to include an arbitrary number of parents. More detailed probing brings up two interesting issues:

- The need for feature renaming, which in fact has useful applications in single inheritance too.
- The case of *repeated* inheritance, in which the ancestor relation links two classes in more than one way.

15.1 EXAMPLES OF MULTIPLE INHERITANCE

The first task is to form a good idea of when multiple inheritance is useful. Let us study a few typical examples from many different backgrounds; a few will be shown in some detail, others only sketched.

This review is all the more necessary that in spite of the elegance, necessity and fundamental simplicity of multiple inheritance, obvious to anyone who cares to study the concepts, this facility has sometimes been presented (often, as one later finds out, based solely on experience with languages or environments that cannot deal with it) as complex, mysterious, error-prone — as the object-oriented method's own "goto". Although it has no basis in either fact or theory, this view has been promoted widely enough to require that we take the time to review a host of cases in which multiple inheritance is indispensable.

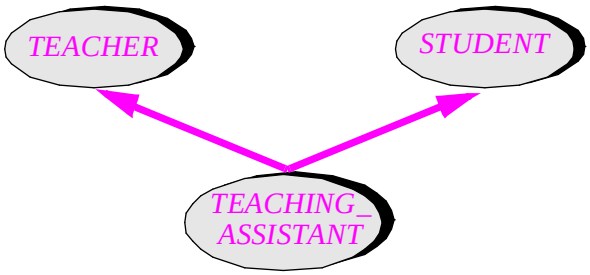
As it will turn out, the problem is not to think of valuable examples, but to stop the flow of examples that will start pouring in once we open the tap.

What not to use as an introductory example

To dispel a frequent confusion, we must first consider an example whose use (with some variants) by many introductory papers, books and lectures may account for some of the common mistrust of multiple inheritance. Not that there is anything fundamentally wrong with the example; it is simply inadequate for an introductory presentation, since it is not typical of simple, straightforward uses of multiple inheritance.

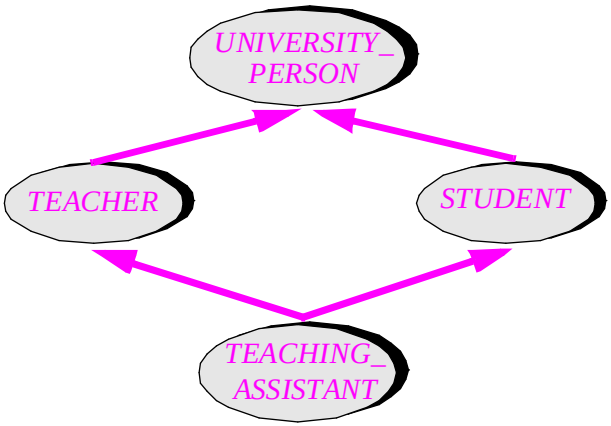
The standard form of this example involves classes *TEACHER* and *STUDENT*, part of the model for some university system; you will be invited to note that some students are also teachers, prompting a new class *TEACHING_ASSISTANT* that inherits from both *TEACHER* and *STUDENT*.

A case of multiple inheritance…



Is this example an improper use of inheritance? Not necessarily. But as an introduction to multiple inheritance it is about as bad as they can get. The problem is that *TEACHER* and *STUDENT* are not separate abstractions but variations on a common theme: person, or more accurately *UNIVERSITY_PERSON*. So if we draw the full picture we see a case of not just multiple but **repeated** inheritance — the scheme, studied later in this chapter, in which a class is a proper descendant of another through two paths or more:

… that is a case of repeated inheritance



Repeated inheritance is a special case; as will be noted when we get to it, using this facility requires good experience with the more elementary forms of inheritance, single and multiple. So it is not a matter for beginners, if only because it seems to create conflicts (what about a feature *name* or *subscribe_to_health_plan* which *TEACHING_ASSISTANT* inherits

For details see "REPEATED INHERITANCE", 15.4, page 543.

from both of its parents, even though they are really in each case a single feature coming from the common ancestor *UNIVERSITY_PERSON*?). With a well-reasoned approach we will be able to remove these conflicts simply. But it is a serious mistake to begin with such exceptional and seemingly tricky cases as if they were typical of multiple inheritance.

The truly common cases do not raise any such problem. Instead of dealing with variants of a single abstraction, they combine **distinct abstractions**. This is the form that you will need most often in building inheritance structures, and the one that introductory discussions should describe. The following examples belong to that pattern.

Can an airplane be an asset?

Our first proper example belongs to system modeling more than to software construction in the strict sense. But it is typical of situations that require multiple inheritance.

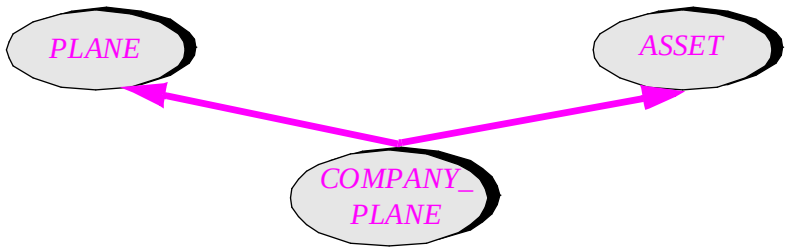
Assume a class *AIRPLANE* describing the abstraction suggested by its name. Queries may include *passenger_count*, *altitude*, *position*, *speed*; commands may include *take_off*, *land*, *set_speed*.

In a different domain, we may have a class *ASSET* describing the accounting notion of an asset — something which a company owns, although it may still be paying installments on it, and which it can depreciate or resell. Features may include *purchase_price*, *resale_value*, *depreciate*, *resell*, *pay_installment*.

You must have guessed where we are heading: companies may own company planes. For the pilot, a company plane is just a plane with its usual features: it takes off, lands, has a certain speed, flies somewhere. From the viewpoint of the accountant (the one who grumbles that the money would have been better kept in the bank or spent on more productive ventures) it is an asset, with a purchase value (too high), an estimated resale value (too low), and the need to pay interest on the loan each month.

To model the notion of company plane we can resort to multiple inheritance:

Company planes




```
class COMPANY_PLANE inherit
        PLANE
        ASSET
feature
        … Any feature that is specific to company planes
          (rather than applying to all planes or all assets) …
end
```

To specify multiple parents in the **inherit** clause, just list them one after the other. (As usual, you can use semicolons as optional separators.) The order in which you list parents is not significant.

Cases similar to *COMPANY_PLANE* abound in system modeling. Here are a few:

- Wristwatches (a special case of the notion of watch, itself specializing the general notion of clock — there are a few inheritance links here) provide commands such as setting the time, and queries such as the current time and date. Electronic calculators provide arithmetic features. There also exist some (quite handy) watch-calculators, elegantly modeled through multiple inheritance.
- Boats; trucks; *AMPHIBIOUS_VEHICLE*. A variant is: boats; planes; *HYDROPLANE*. (There is a hint of repeated inheritance here, as with *TEACHING_ASSISTANT*, since both parents may themselves be descendants of some *VEHICLE* class.)
- You eat in a restaurant; you travel in a train car. To make your trip more enjoyable, the railway company may let you eat in an instance of *EATING_CAR*. A variant of this example is *SLEEPING_CAR*.
- On an instance of *SOFA_BED* you may not only read but also sleep.
- A *MOBILE_HOME* is a *VEHICLE* and a *HOUSE*.

And so on. Multiple inheritance is the natural tool to help model the endless combinations that astute people never tire of concocting.

For a software engineer the preceding examples may at first appear academic, since we get paid not to model the world but to build systems. In many practical applications, however, you will encounter similar combinations of abstractions. A detailed example, from ISE's own graphical development environment appears later in this chapter.

Numeric and comparable values

The next example is much more directly useful to the daily practice of object-oriented software construction. It is essential to the buildup of the Kernel library.

Some of the Kernel library's classes — that is to say, classes describing abstractions of potential use to all applications — require arithmetic features: operations such as **infix** "+", **infix** "–", **infix** "*", **prefix** "–" as well as special values *zero* (identity element for "+") and *one* (identity element for "*"). Kernel library classes that use these features include *INTEGER*, *REAL* and *DOUBLE*; but many non-predefined classes may need them too, for example a class *MATRIX* describing matrices of some application-specific kind. It is appropriate to capture the corresponding abstraction through a deferred class *NUMERIC*, itself a part of the Kernel library:

```
deferred class NUMERIC feature
        … infix "+", infix "–", infix "*", prefix "–", zero, one …
end
```

Mathematically, *NUMERIC* has a precise specification: its instances represent members of a ring (a set equipped with two operations, both of which separately give it the structure of a group, one commutative, with distributivity between the two operations).

Some classes also need an order relation, with features for comparing arbitrary elements: **infix** "<", **infix** "<=", **infix** ">", **infix** ">=". Again this is useful not only to some Kernel library classes, such as *STRING* whose instances are comparable through lexical ordering, but also to many application classes; for example you may write a class *TENNIS_CHAMPION* which takes into account the ranking of professional tennis players, with a feature "<" such that *tc1 < tc2* tells us whether *tc2* is ranked ahead of *tc1*. So it is appropriate to capture the corresponding abstraction through a deferred class *COMPARABLE*, itself a part of the Kernel library:

```
deferred class COMPARABLE feature
    … infix "<", infix "<=", infix ">", infix ">=" …
end
```

Technically the exact model is that of a "preorder".

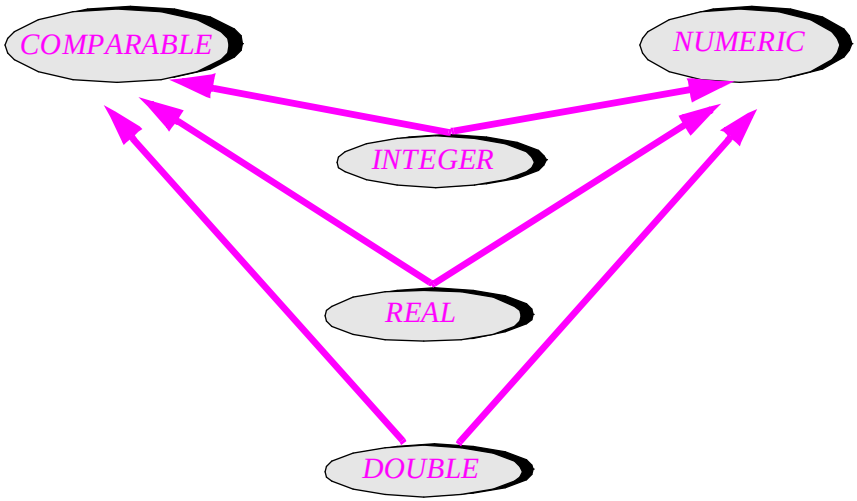
COMPARABLE has a precise mathematical model: its instances represent members of a set ordered by a total order relation.

Not all descendants of *COMPARABLE* should be descendants of *NUMERIC*: in class *STRING*, we need the order features for lexicographical ordering but not the arithmetic features. Conversely, not all descendants of *NUMERIC* should be descendants of *COMPARABLE*: the set of real matrices has addition, multiplication, zero and one, giving it a ring structure, but no total order relation. So it is appropriate that *COMPARABLE* and *NUMERIC*, representing completely different abstractions, should remain distinct classes, neither of them a descendant of the other.

Objects of certain types, however, are both comparable and numeric. (In mathematical terms. the structures modeled by their generating classes are totally ordered rings.) Example classes include *REAL* and *INTEGER*: integers and real numbers can be compared for "<=" as well as added and multiplied. These classes should be defined through multiple inheritance, as in (see the figure on the next page):

```
expanded class REAL inherit
    NUMERIC
    COMPARABLE
feature
    …
end
```

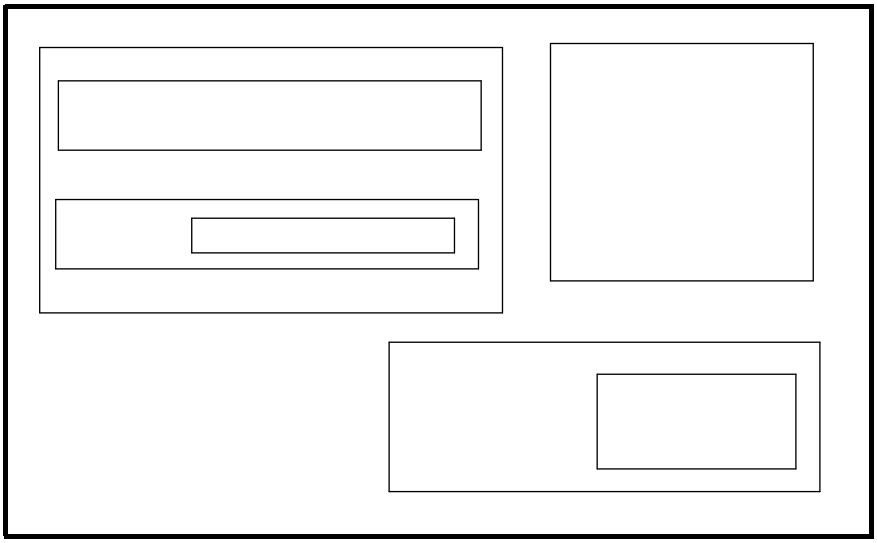
Types of objects that need to be both comparable and numeric are sufficiently common to suggest a class *COMPARABLE_NUMERIC*, still deferred, covering the merged abstraction by multiply inheriting from *COMPARABLE* and *NUMERIC*. So far this solution has not been adopted for the library because it does not bring any obvious advantage and seems to open the way to endless combinations: why not *COMPARABLE_HASHABLE*, *HASHABLE_ADDABLE_SUBTRACTABLE*? Basing such deferred classes on well-accepted mathematical abstractions, such as ring or totally ordered set, seems to yield the right level of granularity. Related issues in the methodology of inheritance are discussed in detail in chapter 16.



Multiple structure inheritance

Windows are trees and rectangles

Assume a window system that allows nesting windows to an arbitrary depth:



Windows and subwindows

In the corresponding class *WINDOW*, we will find features of two general kinds:

- Some deal with a window as a member of a hierarchical structure: list of subwindows, parent window, number of subwindows, add or remove a subwindow.
- Others cover its properties as a graphical object occupying a graphical area: height, width, *x* position, *y* position, display, hide, translate.

It is possible to write the class as a single piece, with all these features mixed together. But this would be bad design. To keep the structure manageable we should separate the two aspects, treating class *WINDOW* as the combination of two abstractions:

- Hierarchical structures, which should be covered by a class *TREE*.
- Rectangular screen objects, covered by a class *RECTANGLE*.

In practice we may need more specific class names (describing some particular category of trees, and a graphical rather than purely geometrical notion of rectangle), but the ones above will be convenient for this discussion. *WINDOW* will appear as:

```
class WINDOW inherit
        TREE [WINDOW]
        RECTANGLE
feature
        … Specific window features …
end
```

Note that class *TREE* will be generic, so we need to specify an actual generic parameter, here *WINDOW* itself. The recursive nature of this definition reflects the recursion in the situation modeled: a window is a tree of windows.

This example will, later on in the discussion, help us understand the need for a feature renaming mechanism associated with inheritance.

A further refinement might follow from the observation that some windows are purely text windows. Although we might represent this property by introducing a class *TEXT_WINDOW* as a client of *STRING* with an attribute

text: *STRING*

we may prefer to consider that each text window *is* also a string. In this case we will use multiple inheritance from *WINDOW* and *STRING*. (If all windows of interest are text windows, we might directly use triple inheritance from *TREE*, *RECTANGLE* and *STRING*, although even in that case it is probably better to work in two successive stages.)

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

The general question of how to choose between heir and client relations, as in the case of *TEXT_WINDOW*, is discussed in detail in the chapter on inheritance methodology.

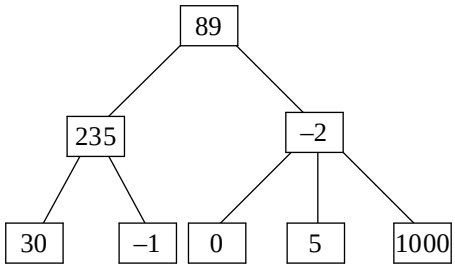
Trees are lists and list elements

Class *TREE* itself provides a striking example of multiple inheritance.

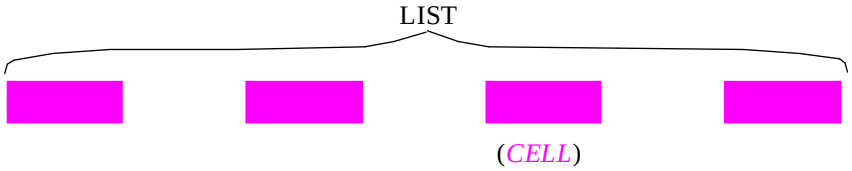
A tree is a hierarchical structure made of nodes, each containing some information. Common definitions tend to be of the form "A tree is either empty or contains an object called the root, together with (recursively) a list of trees, called the children of the root", complemented by a definition of *node*, such as "An empty tree has no nodes; the nodes of a non-empty tree comprise its root and (recursively) the nodes of its children". Although useful, and reflective of the recursiveness inherent in the notion of tree, these definitions fail to capture its essential simplicity.

To get a different perspective, observe that there is no significant distinction between the notion of tree and that of node, as we may identify a node with the subtree of which it is the root. This suggests aiming for a class *TREE* [*G*] that describes both trees and nodes. The formal generic parameter *G* represents the type of information attached to every node; the tree below, for example, is an instance of *TREE* [*INTEGER*].

A tree of integers



Now consider a notion of *LIST*, with a class that has been sketched in earlier chapters. A general implementation (linked, for example) will need an auxiliary class *CELL* to describe the individual elements of a list.



These notions suggest a simple definition of trees: a tree (or tree node) is a list, the list of its children; but it is also a potential list element, as it can be made into a subtree of another tree.

Definition: tree

A tree is a list that is also a list element.

Although this definition would need some refinement to achieve full mathematical rigor, it directly yields a class definition:

```
deferred class TREE [G] inherit
        LIST [G]
        CELL [G]
feature
        …
end
```

From *LIST* come the features to find out the number of children (*count*), add a child, remove a child and so on.

From *CELL* come the features having to do with a node's siblings and parents: next sibling, add a sibling, reattach to a different parent node.

This example is typical of the reusability benefits of multiple inheritance. Writing specific features for subtree insertion or removal would needlessly replicate the work done for lists. Writing specific features for sibling and parent operations would needlessly replicate the work done for list elements. Only a facelift is needed in each case.

In addition you will have to take care, in the **feature** clause, of the specific features of trees and of the little mutual compromises which, as in any marriage, are necessary to ensure that life together is harmonious and prolific. In a class *TREE* derived from these ideas, which has been used in many different applications (from graphics to structural editing), these specific features fit on little more than a page; for the most part, the class is simply engendered as the legitimate fruit of the union between lists and list elements.

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

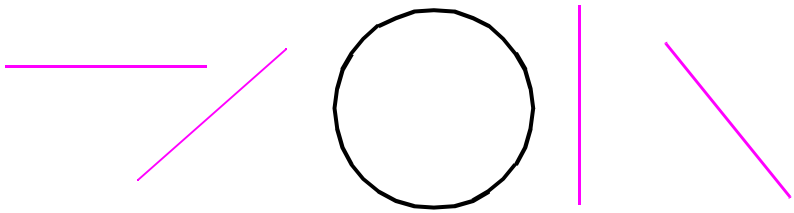
Composite figures

The following example is more than an example; it is a design pattern useful in many different contexts.

Consider an inheritance structure containing classes for various graphical figures, such as the one used in the preceding chapter to introduce some of the fundamental concepts of inheritance — *FIGURE*, *OPEN_FIGURE*, *POLYGON*, *RECTANGLE*, *ELLIPSE* and so on. So far, as you may have noted, that structure used single inheritance.

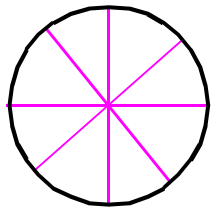
Assume that we have included in this hierarchy all the basic figure patterns that we need. That is not enough yet: many figures are not basic. Of course we could build any graphical illustration from elementary shapes, but that is not a convenient way to work; instead, we will want to build ourselves a library of figures, some basic, some constructed from the basic ones. For example, from basic segment and circle figures

Elementary figures



we may assemble a composite figure, representing a wheel

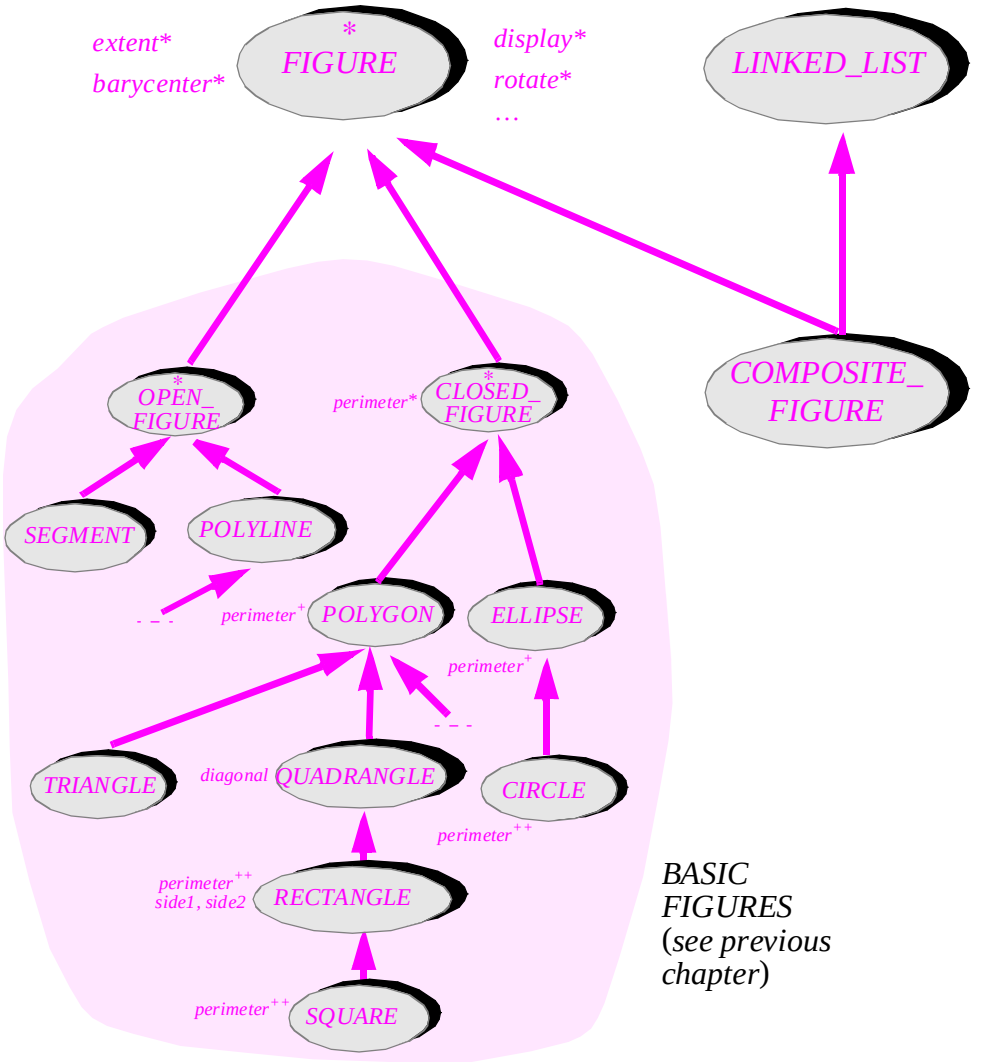
A composite figure



which someone may in turn use as a predefined pattern to draw, say, a bicycle; and so on.

We need a general mechanism for adding a new figure type which will be built from previously defined ones but, once defined, will be on a par with them. Computer drawing tools provide a Group command for this purpose.

Let us call the corresponding notion *COMPOSITE_FIGURE*. A composite figure is clearly a figure; so *COMPOSITE_FIGURE* should inherit from *FIGURE*, achieving the goal of treating composite figures "on a par" with basic ones. A composite figure is also a list of figures — its constituents; each of them may be basic or itself composite. Hence the use of multiple inheritance:



To get an effective class for *COMPOSITE_FIGURE* we choose an implementation of lists; *LINKED_LIST* is just one possibility. The class declaration will look like this:

```
class COMPOSITE_FIGURE inherit
        FIGURE
        LINKED_LIST [FIGURE]
feature
        …
end
```

The **feature** clause is particularly pleasant to write. An operation on a composite figure is, in many cases, an operation on all of its constituents taken in sequence. For example, procedure *display* will be effected as follows in *COMPOSITE_FIGURE*:

```
display is
                -- Display figure by displaying all its components in turn.
        do
                from
                        start
                until
                        after
                loop
                        item.display
                        forth
                end
        end
```

For the details see "ACTIVE DATA STRUCTURES", 23.4, page 774.

As in earlier discussions, we assume that our list classes offer traversal mechanisms based on the notion of cursor: *start* moves the cursor to the first element if any (otherwise *after* is immediately true), *after* indicates whether the cursor is past all elements, *item* gives the value of the element at cursor position, and *forth* advances the cursor by one position.

I find this scheme admirable and hope its beauty will strike you too. Almost everything is concentrated here: classes, multiple inheritance, polymorphic data structures (*LINKED_LIST* [*FIGURE*]), dynamic binding (the call *item.display* will apply the proper variant of *display* based on the type of each list element), recursion (note that any list element — any *item* — may itself be a composite figure, with no limit on the degree of nesting). To think that some people will live an entire life and never see this!

Exercise E15.4, page 567.

It is in fact possible to go further. Consider other *COMPOSITE_FIGURE* features such as *rotate* and *translate*; because they all must apply the corresponding operation to every member figure in turn, their body will look very much like *display*. For an object-oriented designer this is cause for alert: we do not like repetition; we transform it, through encapsulation, into reuse. (This could yield a good motto.) The technique to use here is to define a deferred "iterator" class, whose instances are little machines able to iterate over a *COMPOSITE_FIGURE*. Its effective descendants may include *DISPLAY_ITERATOR* and so on. This is a straightforward scheme and is left to the reader as an exercise.

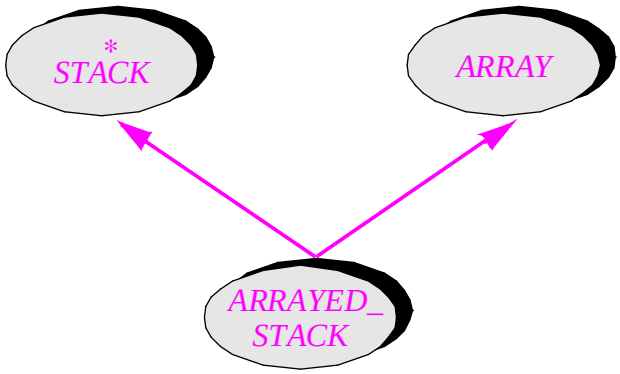
Exercises E15.8, page 568, and E21.6, page 716.

The technique describing composite structures through multiple inheritance, using a list or other container class as one of the parents, is a general **design pattern**, directly useful in widely different areas. Make sure to look at the exercise asking you to apply similar reasoning to the notion of *submenu* in a window system: a submenu is a menu, but it is also a menu entry. Another deals with *composite commands* in an interactive system.

The marriage of convenience

In the preceding examples the two parents played a symmetric role. This is not always the case; sometimes each parent brings a contribution of a different nature.

An important application of multiple inheritance is to provide an implementation of an abstraction defined by a deferred class, using facilities provided by effective class.



A marriage of convenience

Consider the implementation of stacks as arrays. Since classes are available to cover stacks as well as arrays (deferred for *STACK*, effective for *ARRAY*, both seen in earlier chapters), the best way to implement class *ARRAYED_STACK*, describing stacks implemented as arrays, is to define it as an heir to both *STACK* and *ARRAY*. This is conceptually right: an arrayed stack is a stack (as seen by clients) and is also an array (internally). The general form is:

The deferred STACK class appeared on page 501; class ARRAY was sketched on page 373.

```
indexing
        description: "Stacks implemented as arrays"
class ARRAYED_STACK [G] inherit
        STACK [G]
        ARRAY [G]
                … A rename subclause will be added here (see page 540) …
feature
        … Implementation of the deferred routines of STACK
          in terms of ARRAY operations (see below)…
end
```

ARRAYED_STACK offers the same functionality as *STACK*, effecting its deferred features such as *full*, *put*, *count* through implementations relying on array operations.

Here is an outline of some typical features: *full*, *count* and *put*. The condition under which a stack is full is given by

```
full: BOOLEAN is
        -- Is stack representation full?
    do
        Result := (count = capacity)
    end
```

Here *capacity*, inherited from class *ARRAY*, is the number of positions in the array. For *count* we need an attribute:

```
count: INTEGER
```

This is a case of effecting a deferred feature into an attribute. Here finally is *put*:

```
put (x: G) is
        -- Push x on top.
    require
        not full
    do
        count := count + 1
        array_put (x, count)
    end
```

Procedure *array_put*, inherited from *ARRAY*, assigns a new value to an array element given by its index.

See "Using a parent's creation procedure", page 539.

The array features *capacity* and *array_put* had different names in class *ARRAY*: *count* and *put*. The name change is explained later in this chapter.

ARRAYED_STACK is representative of a common kind of multiple inheritance, called the *marriage of convenience*. It is like a marriage uniting a rich family and a noble family. The bride, a deferred class, belongs to the aristocratic family of stacks: it brings prestigious functionality but no practical wealth — no implementation worth speaking of. (What good is an effective *change_top* with a deferred *put* and *remove*?) The groom comes from a well-to-do bourgeois family, arrays, but needs some luster to match the efficiency of its implementation. The two make a perfect match.

Besides providing effective implementations of routines deferred in *STACK*, class *ARRAYED_STACK* may also redefine some which were not deferred. In particular, with an array representation, *change_top* (*x*: *G*), implemented in *STACK* as *remove* followed by *put* (*x*), may be implemented more efficiently as

```
array_put (x, count)
```

To make this redefinition valid, do not forget to announce it in the inheritance clause:

```
class ARRAYED_STACK [G] inherit
    STACK [G]
        redefine change_top end
        … The rest as before …
```

The invariant of the class might read

```
invariant
    non_negative_count: count >= 0
    bounded: count <=capacity
```

The two parts of the assertion are of a different nature. The first expresses a property of the abstract data type. (It was in fact already present in the parent class *STACK*, and so is redundant; it is included here for pedagogical purposes, but should not appear in a final version of the class.) The second line involves *capacity*, that is to say the array representation: it is an **implementation invariant**.

"Implementation invariants", page 377.

You might take a minute to compare *ARRAYED_STACK*, as sketched here, with *STACK2* of an earlier discussion, and see how dramatically inheritance simplifies the class. This comparison will be pursued in the discussion of the methodology of inheritance, which will also address some of the criticisms occasionally heard against marriage-of-convenience inheritance and, more generally, against what is sometimes called *implementation inheritance*.

The methodological discussion is "It feels so good, but is it wrong?", page 844. STACK2 appeared on page 350.

Structure inheritance

Multiple inheritance is indispensable when you want to state explicitly that a certain class possesses some properties beyond the basic abstraction that it represents.

Consider for example a mechanism that makes object structures persistent (storable on long-term storage). You may have to request that the lead object in a storable structure be equipped with the corresponding store and retrieve operations: in addition to its other properties such an object is "storable". In the Kernel library, as we have seen, this property is captured by a class *STORABLE*, from which any other class can inherit. Clearly, such classes may have other parents as well, so this would not work without multiple inheritance. This form of inheritance, from a class that describes a general structural property — often with a name that ends with *-ABLE* — is similar to inheritance from classes *COMPARABLE* and *NUMERIC* seen earlier in this chapter. The discussion of inheritance methodology will define it as inheritance of the *structural* kind.

On STORABLE see "Deep storage: a first view of persistence", page 250.

For a more detailed discussion of this form of inheritance: "Structure inheritance", page 831.

Without multiple inheritance, there would be no way to specify that a certain abstraction must possess two structural properties — numeric and storable, comparable and hashable. Selecting one of them as *the* parent would be like having to choose between your father and your mother.

Facility inheritance

Here is another typical case. Many tools need "history" facilities, enabling their users to perform such operations as:

- Viewing the list of recent commands.
- Executing again a recent command.
- Executing a new command defined by editing a recent one and changing a few details.

- Undoing the effect of the last command not yet undone

Such a mechanism makes any interactive tool nicer to use. But it is a chore to write. As a result, only a few tools (such as certain "shells" under Unix and Windows) support it, often partially. Yet the general techniques are tool-independent. They can be encapsulated in a class, from which a session-control class for any tool can then inherit. (A solution based on the client relation may be possible, but is less attractive.) Once again, without multiple inheritance such an inheritance link would conflict with other possible parents.

A similar case is that of a class *TEST* encapsulating a number of mechanisms useful for testing a class: getting and storing user input, printing and storing output, comparing with expected values, recording all the results, comparing with earlier test runs (*regression testing*), managing the testing process. Although a client-based solution may be preferable in some cases, it is convenient to have the possibility, for testing a class *X*, of defining a class *X_TEST* that inherits from *X* and from *TEST*.

In later chapters we will encounter other cases of such *facility* inheritance, whereby a class *F* encapsulates a set of related facilities, such as constants or routines from a mathematical library, which any class can then obtain by inheriting from *F*.

See chapter 24.

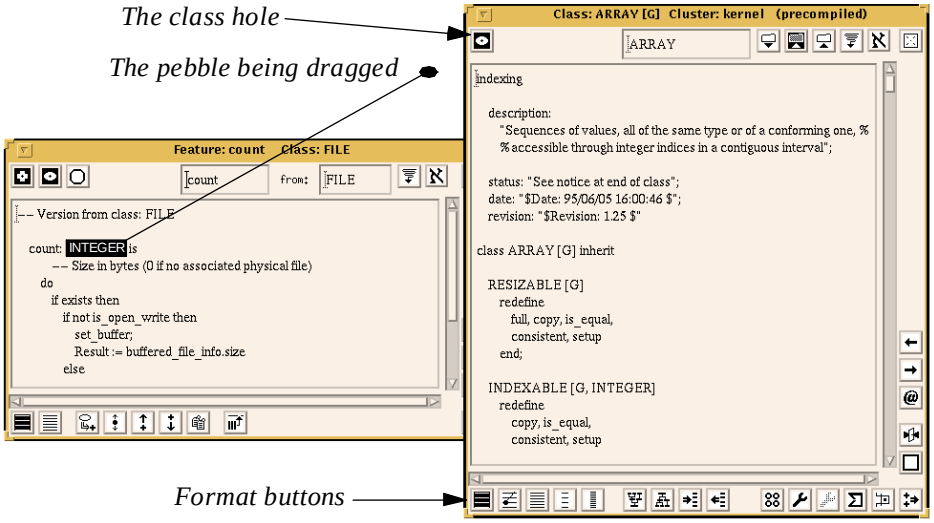
Although the use of inheritance in such cases is sometimes viewed with suspicion, it is in fact a perfectly legitimate application of the concept. It does differ in one respect from the other examples of multiple inheritance reviewed in this chapter: in the cases just reviewed, we could achieve our goals, albeit less conveniently, with a client rather than inheritance link.

Buttonholes

Here is a case in which, as in earlier ones, multiple inheritance is indispensable. It is similar in spirit to "company planes", "sleeping cars" and other examples of the combination-of-abstractions type encountered earlier. Rather than using concepts from some external model, however, this one deals with genuine software abstractions. The reason why it has been moved to the end of this review of multiple inheritance examples is that understanding it requires a little background preparation.

See chapter 36.

Like other graphical applications, many tools of the development environment presented in the last chapter offer "buttons", on which you can click to trigger certain operations. They also use a "pick and throw" mechanism (a variation on traditional "drag-and-drop"), through which you can select a visual object, causing the mouse cursor to change into a "pebble" that indicates the type of the object, and bring it to a **hole** of a matching shape. You can "throw" the pebble into the hole by right-clicking; this causes some operation to occur. For example, a Class Tool, which you use to explore the properties of a class in the development environment, has a "class hole" into which you can drag-and-drop a class pebble; this causes the tool to retarget itself to the selected class.



Pick-and-throw

In the figure, a user has picked somewhere — in a Feature Tool — the class *INTEGER*, by right-clicking on its name. He is moving it towards the class hole of the Class Tool currently targeted to (showing the text of) class *ARRAY*. Note the row of format buttons at the bottom; clicking on one of them will show other information for *ARRAY*; for example if you left-click on ☰ you will get the short form. The pick-and-throw (unless canceled by a left-click) will end when the user right-clicks on the class hole, whose shape, representing Class, matches that of the pebble. This will retarget the Class Tool on the right to the selected class — *INTEGER*.

In some cases it may be convenient to let a hole act as button too, so that you can not only throw an object into it but also, independently of any pick-and-throw, left-click on it to produce a certain effect. For example the class hole, in which the small dot suggests the presence of a current target (first *ARRAY*, then *INTEGER*) can serve as a button; left-clicking on it retargets the tool to its current target, which is useful if the display was overwritten. Such holes which double up as buttons are called buttonholes.

As you will have guessed, class *BUTTONHOLE* multiply inherits from *BUTTON* and from *HOLE*. The new class simply combines the features and properties of its parents, since a buttonhole reacts like a button to the operations on buttons, and like a hole to the operations on holes.

An assessment

See [M 1994a] *on library design.*

The examples accumulated so far are representative of the power and usefulness of multiple inheritance. Experience in building general-purpose libraries confirms that multiple inheritance is needed throughout.

Whenever you must combine two abstractions, not having multiple inheritance would mean that you choose one of them as the official parent, and duplicate all the other's features by copy-and-paste — making the new class, as it were, an illegitimate child. On the illegitimate side, you lose polymorphism, the Open-Closed principle, and all the reusability benefits of inheritance. This is not acceptable.

15.2 FEATURE RENAMING

Multiple inheritance raises an interesting technical problem: name clashes. The solution, feature renaming, turns out to have applications far beyond that original problem, and leads to a better understanding of the nature of classes.

Name clashes

A class has access to all the features of its parents. It can use them without having to indicate where they come from: past the **inherit** clause in **class** *C* **inherit** *A* …, a feature *f* of *C* is known just as *f*. The same is true of clients of *C*: for *x* of type *C* in some other class, a call to the feature is written just *x*.*f*, without any reference to the *A* origin of *f*. If the metaphors were not so incompatible, we could view inheritance as a form of adoption: *C* adopts all the features of *A*.

It adopts them under their assigned names: the set of feature names of a class includes all of its parents' feature name sets.

What then if two or more parents have used the same name for different features? We have relied on the rule of no intra-class overloading: within a class, a feature name denotes only one feature. This could now be violated because of the parents. Consider

```
class SANTA_BARBARA inherit
        LONDON
        NEW_YORK
feature
        …
end-- class SANTA_BARBARA
```

What can we do if both *LONDON* and *NEW_YORK* had a feature named the same, say *foo* (for some reason a favorite name in programming examples)?

> Do not attach too much importance to the names in this example, by the way. No useful abstraction is assumed behind the class names, especially none that would justify the inheritance structure. The names simply make the example easier to follow and remember than if we called our classes *A*, *B* and *C*.

Under no circumstances should we renounce the no-overloading rule, essential to keep classes simple and easy to understand. Within a class, a name should mean just one thing. So class *SANTA_BARBARA* as shown is invalid and the compiler must reject it.

This rule seems rather harsh. In an approach emphasizing construction-box-like combination of modules from several sources, we may expect attempts to combine separately developed classes that contain identically named features.

> As an example, we saw earlier a version of class *TREE* that inherits from *CELL* and *LIST*, both of which have a feature called *item*; for a cell, it returns the value stored in the cell, and for a list it returns the value at the current cursor position. Both also have a feature called *put*. These choices of name are all reasonable, and we would not like to have to change the original classes just because someone got a clever idea for defining trees by combining them.

What can be done? You should not have to go back to the parents. You may not have access to the source text of *LONDON* and *NEW_YORK*; you may have access to it, but not be permitted to change it; you may be permitted but unwilling, as *LONDON* comes from an external supplier and you know there will be new releases, which would force you to do the work all over again; and most importantly you know about the Open-Closed principle, which says one should not disturb modules when reusing them for new extensions, and you are rightly wary of changing the interface of classes (*LONDON* and *NEW_YORK*) which may already have numerous clients that rely on the old names.

It is a mistake to blame the parents for a name clash occurring in inheritance: the problem is in the would-be heir. There too should the solution be.

The language solution to name clashes follows from these observations. A class that inherits different but identically named features from different parents is invalid, but will become valid by including one or more **rename** subclauses in the inheritance clause. A **rename** subclause gives a new local name to one or more inherited features. For example:

```
class SANTA_BARBARA inherit
        LONDON
                rename foo as fog end
        NEW_YORK
feature
        …
end -- class SANTA_BARBARA
```

Both within *SANTA_BARBARA* and in its clients, the *foo* feature from *LONDON* will be referred to as *fog*, and the one from *NEW_YORK* as *foo*. Clients of *LONDON*, of course, will still know the feature as *foo*.

This is enough (assuming there is no other clash, and no other feature of *LONDON* or *NEW_YORK* is called *fog*) to remove the clash. Of course, we could have renamed the *NEW_YORK* feature instead; or we could have renamed both for symmetry:

```
class SANTA_BARBARA inherit
        LONDON
                rename foo as fog end
        NEW_YORK
                rename foo as zoo end
feature
        …
end -- class SANTA_BARBARA
```

The **rename** subclause follows the name of a parent and comes before the **redefine** subclause if any. It can of course rename several features, as in

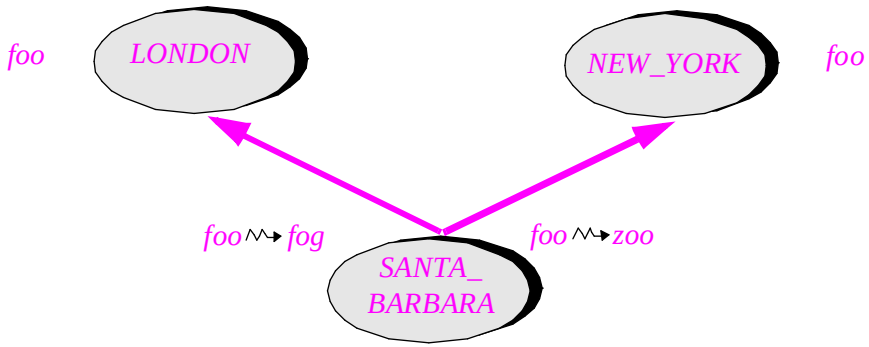
```
class TREE [G] inherit
        CELL [G]
                rename item as node_item, put as put_right end
```

which removes clashes between features of *CELL* and their namesakes in the other parent, *LIST*. The clause renames the *item* feature from *CELL* as *node_item*, since this feature denotes the item attached to the current node, and similarly renames *put* as *put_right*.

Effects of renaming

Let us make sure we fully understand the results of a renaming. Assume the last form of class *SANTA_BARBARA* (the one that renames both inherited versions of *foo*):

A name clash, removed



(Note the graphical symbol for renaming: ⇝.) Assume entities of the three types:

l: *LONDON*; *n*: *NEW_YORK*; *s*: *SANTA_BARBARA*

Then *l*.*foo* and *s*.*fog* are both valid; after a polymorphic assignment *l* := *s* they would have the same effect, since the feature names represent the same feature. Similarly *n*.*foo* and *s*.*zoo* are both valid, and after *n* := *s* they would have the same effect.

None of the following, however, is valid:

- *l*.*zoo*, *l*.*fog*, *n*.*zoo*, *n*.*fog* since neither *LONDON* nor *NEW_YORK* has a feature called *fog* or *zoo*.
- *s*.*foo* since as a result of the renaming *SANTA_BARBARA* has no feature called *foo*.

Artificial as the names are, this example also illustrates the nature of the name clash issue. Believe it or not, I have heard it presented as a “deep semantic problem”. It is neither semantic nor deep; rather, a simple syntactical problem. Had one of the class authors been led by the local context to choose the name *fog* in the first class or *zoo* in the second, no clash would have occurred; yet in each case the change is just one letter. The name clash is, as it were, a case of bad luck; it does not reveal any intrinsic problem with the classes or their ability to be combined. If you think of multiple inheritance as marriage, this is not a dramatic case, discovered at the last minute, of a rare blood incompatibility; it is more like realizing that the spouses’ mothers are both called Tatiana, making life a little more complicated for their grandchildren to come, but easy to solve through proper naming conventions.

Renaming and redeclaration

In the last chapter we studied another inheritance mechanism: redeclaration of an inherited feature. (Remember that redeclaration includes the redefinition of an already effective feature, and the effecting of a deferred one.) It is illuminating to compare the effect of renaming and redeclaring a feature:

- Redeclaration changes the feature, but keeps its name.
- Renaming changes the name but keeps the feature.

With redeclaration you can ensure that the *same* feature name refers to *different* actual features depending on the type of the object to which it is applied (that is to say, the dynamic type of the corresponding entity). This is a semantic mechanism.

Renaming is a syntactic mechanism, allowing you to refer to the *same* feature under *different* names in different classes.

In some cases you may want to do both:

```
class SANTA_BARBARA inherit
        LONDON
                rename
                        foo as fog
                redefine
                        fog
                end
        …
```

Then assuming *l*: *LONDON*; *s*: *SANTA_BARBARA* as before, and the polymorphic assignment *l* := *s*, the calls *l*.*foo* and *s*.*fog* will both trigger the redefined version (whose declaration must appear in a **feature** clause of the class).

You will have noted that the **redefine** subclause uses the new name. This is normal since that name is the only one under which the feature is known in the class. Accordingly, the **rename** clause appears before all other inheritance subclauses (**redefine**, and others yet to be studied: **export**, **undefine**, **select**). Past the **rename** clause, the feature — like an immigrant given a new identity at Ellis Island by a customs officer who found the old name too hard to pronounce — has shed its ancestral name and will be known under its new one to class, clients and descendants alike.

Local name adaptation

The ability to rename an inherited feature is interesting even in the absence of a name clash. It allows the designer of a class to define the appropriate name for every feature, whether immediate (declared in the class itself) or inherited.

The name under which a class inherits a facility from an ancestor is not necessarily the most telling one for its clients. The original name may have been well adapted to the ancestor's clients, but the new class has its own context, its own abstraction, which may

suggest its own naming conventions. To provide this abstraction it finds the ancestor's *features* useful, but not necessarily the feature names. Renaming, which enables us to distinguish features from feature names, provides the solution.

The construction of class *WINDOW* as an heir of *TREE* provides a good example. *TREE* describes the hierarchical structure, common to general trees and windows; but the tree names may not be desirable for the interface that *WINDOW* presents to its clients. Renaming provides the ability to put these names in tune with the local context:

```
class WINDOW inherit
        TREE [WINDOW]
                rename
                        child as subwindow, is_leaf as is_terminal, root as screen,
                        arity as child_count, …
                end
        RECTANGLE
feature
        … Specific window features …
end
```

Similarly, *TREE* inheriting from *CELL* may rename *right* as *right_sibling* and so on. Through renaming, a class may offer its clients a consistent set of names for the services it offers, regardless of how these services were built from facilities provided by ancestors.

The game of the name

The use of renaming for local name adaptation highlights the importance of naming — feature naming, but also class naming — in object-oriented software construction. A class is formally a mapping from feature names to features; the feature names determine how it will be known to the rest of the world.

See "Standard names", page 882.

In a later chapter we will see a number of systematic rules for choosing feature names. Interestingly, they promote a set of across-the-board names — *count*, *put*, *item*, *remove*, … — to emphasize commonalities between abstractions over the inevitable differences. This style, which increases the likelihood of name clashes under multiple inheritance, decreases the need for "vanity" renaming of the kind illustrated with *WINDOW*. But whatever general naming conventions we follow, we must have the flexibility to adapt the names to the local needs of each class.

Using a parent's creation procedure

Let us see one more example of renaming, illustrating a typical scheme where the renamed feature is a creation procedure. Remember *ARRAYED_STACK*, obtained by inheritance from *STACK* and *ARRAY*; the creation procedure of *ARRAY* allocates an array with given bounds:

```
make (minb, maxb: INTEGER) is
        -- Allocate array with bounds minb and maxb
        -- (empty if minb > maxb)
    do … end
```

To create a stack, we must allocate the array so that it will accommodate a given number of items. The implementation will rely on the creation procedure of *ARRAY*:

```
class ARRAYED_STACK [G] inherit
    STACK [G]
        redefine change_top end
    ARRAY [G]
        rename
            count as capacity, put as array_put, make as array_make
        end
creation
    make
feature -- Initialization
    make (n: INTEGER) is
            -- Allocate stack for at most n elements.
        require
            non_negative_size: n >= 0
        do
            array_make (1, n)
        ensure
            capacity_set: capacity = n
            empty: count = 0
        end
    … Other features (see "The marriage of convenience", page 530) …

invariant
        count >= 0; count <= capacity
end -- class ARRAYED_STACK
```

Note that here our naming conventions — the use of *make* as the standard name for basic creation procedures — would cause a name clash, which, however, does not occur thanks to renaming.

We also need to remove ambiguities for *count* and *put*, both used for features of *ARRAY* as well as *STACK*. Query *count*, by convention, denotes the number of items in a structure; for *ARRAYED_STACK*, the relevant count is the number of elements pushed, that is to say, *count* from *STACK*; the other *count*, from *ARRAY*, becomes the stack's capacity — the maximum number of pushable items — and so is renamed *capacity*. Similarly, *put* for stacks is the push operation; we keep the array *put* (the operation that replaces the element at a certain array position) under the new name *array_put*. It is used, as you will remember, in the effecting of the other *put*, the stack pushing procedure.

15.3 FLATTENING THE STRUCTURE

Renaming is only one of the tools that the inheritance craftsman can use to build rich classes satisfying the needs of his clients. Another is redefinition. Later in this chapter, and in the next one, we will see a few more mechanisms: undefinition, join, **select**, descendant hiding. The power of these combined mechanisms makes inheritance sometimes obtrusive, and suggests the need for a special, inheritance-free version of a class: the flat form.

The flat form

In the view that we see emerging, inheritance is a *supplier* technique more than a client technique. It is primarily an internal tool for constructing classes effectively. True, the client side will need to know about the inheritance structure if it is to use polymorphism and dynamic binding (with *a1*: *A*; *b1*: *B* you need to know that *B* is a descendant of *A* if you are to use the assignment *a1* := *b1*); apart from that case, however, the inheritance structure that led to a particular class is none of the clients' business.

> Like a good car mechanic, we are entirely led by the needs of our customers, but how we go about taking care of them in the back of the garage is our responsibility.

As a consequence, it should be possible to present a class in a self-contained manner, independent from any knowledge of its ancestry. This is particularly important in the case of using inheritance to separate various components of a composite abstraction, such as the tree and rectangle parts of the window concept.

The flat form of a class serves that purpose. It is not something you will ever write; instead, you will rely on a tool of the software development environment to produce it for you, through a command-line script (**flat** *class_name*) or when you click on a certain icon.

The flat form of a class *C* is a valid class text which has exactly the same semantics as *C* when viewed from a client, except for polymorphic uses, but includes no inheritance clause. It is what the class would have looked like had its author not been able to use inheritance. To produce a flat form means:

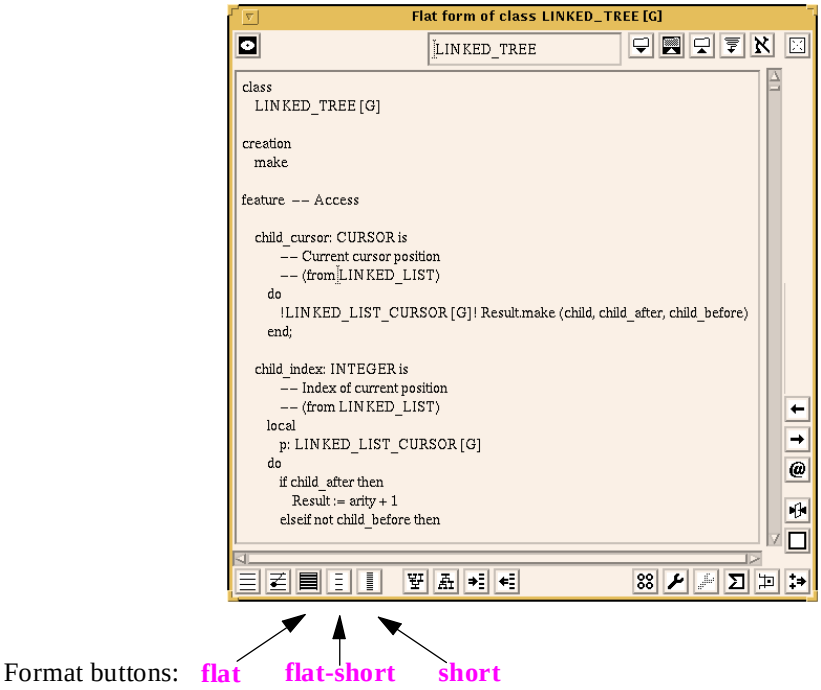
- Removing the entire **inherit** clause if any.
- Keeping all the feature declarations or redeclarations of *C*.
- Adding declarations for all inherited features, copied from the declarations in the applicable parents and taking into account all the inheritance transformations that were specified in the **inheritance** clause: renaming, redefinition, undefinition, **select**, feature join.
- Adding to each inherited feature a comment line of the form **from** *ANCESTOR* indicating the name of the proper ancestor from which the current version is derived: the closest one that declared or redeclared the feature (and, in the case of a feature join, described later in this chapter, the winning side).
- Reconstructing the full preconditions and postconditions of inherited routines (according to the rules on assertion inheritance explained in the next chapter).

- Reconstructing the full invariant, by **and**ing all the parents' invariants, after applying the proper transformations if they use any renamed or selected feature.

The resulting class text shows all the features of the class at the same level, not making any difference (except for the **from** *ANCESTOR* comments) between immediate and inherited features. If present, the labels of feature clauses — as in **feature** -- Access — are retained; clauses with identical labels, whether from parents or the class itself, are merged. Within each feature clause the features appear alphabetically.

An "immediate" feature is one introduced in the class itself.

The illustration below shows the beginning of the flat form of the Base library class *LINKED_TREE*, produced in a Class Tool of ISE's development environment (and scrolled past the **indexing** clause). To obtain this result, you target the Class Tool to the class, and click on the Flat format button.



Displaying a flat form

Uses of the flat form

The flat form is a precious tool for developers: it enables them to see the full set of properties of a class, all together in one place, ignoring how these features were derived in the inheritance games. A potential drawback of inheritance is that when reading a class text you may not immediately see what a feature name means, since the declaration can be in any ancestor. The flat form solves this problem by giving you the full picture.

The flat form may also be useful to deliver a stand-alone version of a class, not encumbered by the class history. That version will not be usable polymorphically.

The flat-short form

See "Using assertions for documentation: the short form of a class", page 390.

The flat form is a valid class text. So in its just mentioned role as documentation, it is of interest for the supplier side — for developers working on the class itself or a new descendant. The client side needs more abstraction.

In an earlier chapter we saw the tool that provides this abstraction: **short** (corresponding in the last figure to the second button to the right of **flat**.)

Combining the two notions yields the notion of flat-short form. Like the short form, the flat-short form of a class only includes public information, removing any non-exported feature and, for exported features, removing any implementation aspects, **do** clauses in particular. But like the flat form, it treats all features, immediate or inherited, as peers — whereas for a class with parents the non-flat short form only shows information about immediate features.

The flat-short form is the primary mechanism for documenting classes, in particular reusable library classes, for the benefits of their users (client authors). The book presenting the Base libraries [M 1994a] provides all the class specifications in that form.

15.4 REPEATED INHERITANCE

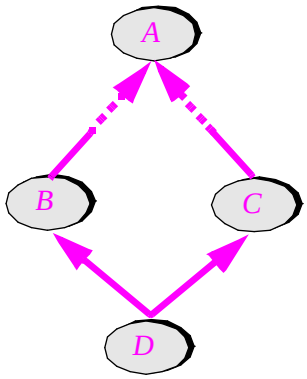
As noted at the beginning of this chapter, repeated inheritance arises whenever a class is a descendant of another in more than one way. This case causes some potential ambiguities, which we must resolve.

Repeated inheritance will only arise explicitly in advanced development; so if you are only surveying the key components of the method you may skip directly to the next chapter.

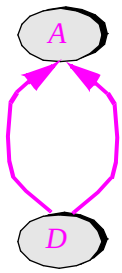
Sharing ancestors

As soon as multiple inheritance is allowed into a language, it becomes possible for a class *D* to inherit from two classes *B* and *C*, both of which are heirs, or more generally descendants, of the same class *A*. This situation is called repeated inheritance.

Repeated inheritance



(1) Indirect



(2) Direct

If *B* and *C* are heirs of proper descendants of *A* (case 1 in the figure), the repeated inheritance is said to be indirect. If *A*, *B* and *C* are all the same class (case 2), the repeated inheritance is direct; this is achieved by writing

```
class D inherit
        A
        A
        …
feature
        …
end
```

Intercontinental drivers

The following system modeling example will enable us to see under what circumstances repeated inheritance may occur and to study the problem that it raises. Assume a class *DRIVER* with attributes such as

```
age: INTEGER
address: STRING
violation_count: INTEGER        -- The number of recorded traffic violations
```

and routines such as

```
pass_birthday is do age := age + 1 end
pay_fee is
                -- Pay the yearly license fee.
        do … end
```

An heir of *DRIVER*, taking into account the specific characteristics of US tax rules, may be *US_DRIVER*. Another may be *FRENCH_DRIVER* (with reference to places where cars are driven, not citizenship).

Now we may want to consider people who drive in both France and the US, perhaps because they reside in each country for some part of the year. A simple way to express this situation is to use multiple inheritance: class *FRENCH_US_DRIVER* will be declared as heir to both *US_DRIVER* and *FRENCH_DRIVER*. As shown by the figure at the top of the facing page, this causes repeated inheritance.

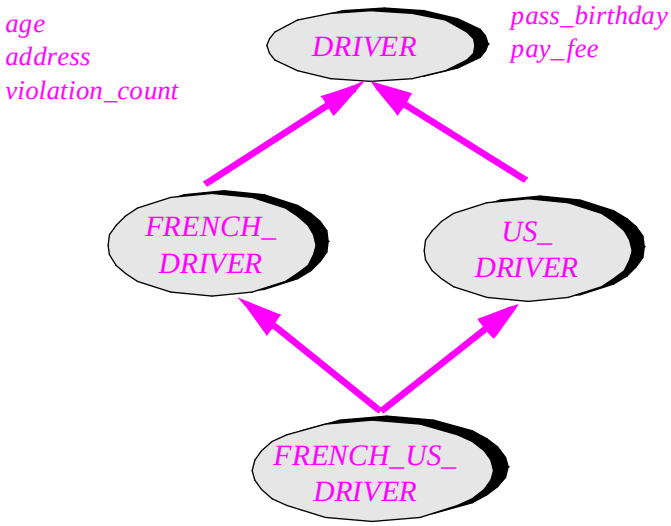
To make sure that the example is a proper use of inheritance we assume that *US_DRIVER* and *FRENCH_DRIVER* are not just distinguished by the value of some attribute representing the country of driving, but are indeed distinct abstraction variants, each with its specific features. Chapter 24 discusses in depth the methodology of using inheritance.

Sharing and replication

The first and principal problem of repeated inheritance appears clearly in the intercontinental driver example:

What is the meaning in the repeated descendant (*FRENCH_US_DRIVER in the example*) *of a feature inherited from the repeated ancestor* (*DRIVER*)?

Kinds of driver



Consider a feature such as *age*. It is inherited from *DRIVER* by both *US_DRIVER* and *FRENCH_DRIVER*; so at first sight the name clash rule seems to require renaming. But this would be too stringent: there is no real conflict since *age* from *US_DRIVER* and age from *FRENCH_DRIVER* are not really different features: they are one feature, from *DRIVER*. Unless you are trying to hide something from someone, you have the same age wherever you happen to be driving. The same applies to procedure *pass_birthday*.

If you read carefully the rule about name clashes, you will have noted that it does not preclude such cases. It stated:

Page 536.

> *A class that inherits different but identically named features from different parents is invalid.*

Here the versions of *age* and *pass_birthday* that *FRENCH_US_DRIVER* inherits from its two parents are not "*different*" features, but a single feature in each case. So there is no real name clash. (An ambiguity could still exist if one of the features was redeclared in an intermediate ancestor; we will see shortly how to resolve it. For the moment we assume that nothing is redeclared.)

In such cases, when a feature coming from a repeated ancestor is inherited under the same name from two or more parents, the clear rule is that it should give a single feature in the repeated descendant. This case will be called **sharing**.

Is sharing always appropriate? No. Consider *address*, *pay_fee*, *violation_count*: our dual drivers will most likely declare two different addresses to the respective Departments of Motor Vehicles; paying the yearly fee is a separate process for each country; and traffic violations are distinct. For each of these features inherited from *DRIVER*, class *FRENCH_US_DRIVER* needs not one but two different features. This case will be called **replication**.

What the example — and many others — also shows is that we could not get what we need with a policy that would either share all features of a repeated ancestor or replicate all of them. This is too coarse a level of granularity. We need the ability to tune the policy *separately for each repeatedly inherited feature*.

We have seen how to obtain sharing: just do nothing — inherit the original version from both parents under the same name. How do we obtain replication? By doing the reverse: inheriting it under two different names.

This idea is consistent with the general rule, simple and clear, that we apply to features and their names: within a class, a feature name denotes only one feature; two separate names denote two separate features. So to replicate a repeatedly inherited feature we simply make sure that some renaming occurs along the way.

Repeated Inheritance rule

In a repeated descendant, versions of a repeatedly inherited feature inherited under the same name represent a single feature. Versions inherited under different names represent separate features, each replicated from the original in the common ancestor.

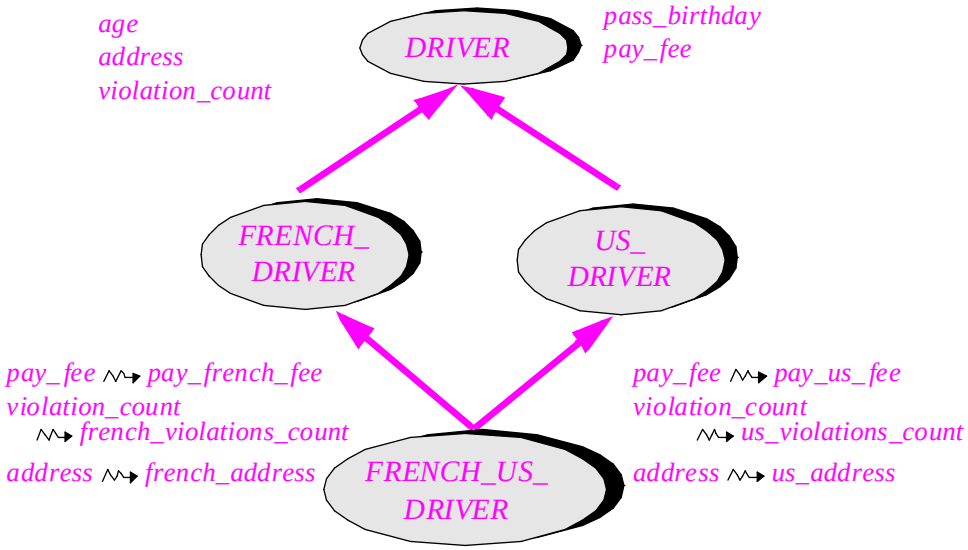
This rule applies to attributes as well as routines. It gives us a powerful replication mechanism: from one feature of a class, it is possible in a descendant to get two or more features. For an attribute, this means an extra field in all the instances; for a routine, it means a new routine, initially with the same algorithm.

Except in special cases involving redeclaration, the replication can be conceptual only: no code actually gets duplicated, but the repeated descendant has access to two features.

The rule gives us the desired flexibility for combining classes. For example the class *FRENCH_US_DRIVER* may look like this:

```
class FRENCH_US_DRIVER inherit
        FRENCH_DRIVER
                rename
                        address as french_address,
                        violation_count as french_violation_count,
                        pay_fee as pay_french_fee
                end
        US_DRIVER
                rename
                        address as us_address,
                        violation_count as us_violation_count,
                        pay_fee as pay_us_fee
                end
feature
        …
end -- class FRENCH_US_DRIVER
```

Sharing and replication

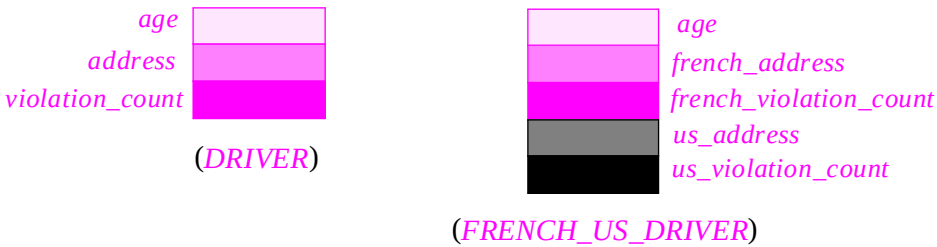


The renaming occurs here at the last stage — in the repeated descendant — but some or all of it could also have been done by intermediate ancestors *FRENCH_DRIVER* and *US_DRIVER*; all that counts is whether in the end a feature is repeatedly inherited under one name or more.

The features *age* and *pass_birthday*, which have not been renamed along any of the inheritance paths, will remained shared, as desired.

A replicated attribute such as *address* will, as noted, yield a new field in each of the instances of the repeated descendant. So assuming there are no other features than the ones listed, here is how instances of the classes will look:

Attribute replication



(Instances of *FRENCH_DRIVER* and *US_DRIVER* have the same composition as those of *DRIVER* as shown.)

This is the conceptual picture, but with a good implementation it must be the concrete representation too. Particularly important is the ability not to replicate the fields for shared attributes such as *age* in *FRENCH_US_DRIVER*. A naïve implementation would replicate all fields anyway; some fields, such as the duplicate *age* field, would simply never be used. Such waste of space is not acceptable, since it would accumulate as we go down inheritance

hierarchies, and lead to catastrophic space inefficiency. (As a general rule, one must be very careful with attributes, as every attribute field will be present at run time in each one of the potentially many instances of a class and its descendants.)

The compiling mechanism of the development environment described at the end of this book indeed makes sure that no attribute space is lost: conceptually shared attributes are shared physically too. This is one of the most difficult parts of implementing inheritance and the calling machinery of dynamic binding, especially under the additional requirement that repeated inheritance must not affect the performance achievements described in earlier chapters:

- Zero cost for genericity.
- Small, constant-bounded cost for dynamic binding (that cost must be the same whether or not a system includes repeated inheritance).

The implementation meets these goals, making repeated inheritance a technique that any system can use at no extra cost.

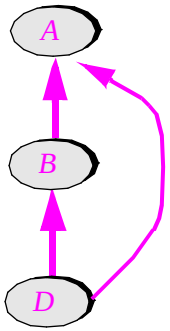
> Repeated inheritance in C++ follows a different pattern. The level of granularity for deciding to share or duplicate is the class. So if you need to duplicate one field from the repeated ancestor, you will need to duplicate all. For that reason, C++ users tend to stay away from this mechanism altogether. Java has eliminated the problem — by eliminating multiple inheritance.

Unobtrusive repeated inheritance

Cases of repeated inheritance similar to the "transcontinental drivers", with duplicated features as well as shared ones, do occur in practice, but not frequently. They are not for beginners; only after you have reached a good level of sophistication and practice in object technology should you encounter any need for them.

If you are writing a straightforward application and end up using repeated inheritance, you are probably making things more complicated than you need to.

Redundant inheritance



The figure shows a typical beginner's (or absent-minded developer's) mistake: *D* is made an heir of *B*, and also needs facilities from *A*; but *B* itself inherits from *A*. Forgetting that inheritance is transitive, the developer wrote

```
class D... inherit
        B
        A
...
```

This case causes repeated inheritance, but what it really shows is *redundant* inheritance. One of the pleasant consequences of the conventions discussed so far, and of the corresponding implementation, is that they will yield the expected behavior in such a case: in the absence of renaming, all features will be shared; no new features will be introduced, and there will be no performance overhead. Even if *B* renames some attributes, the only consequence will be some waste of space.

The only exception is the case in which *B* has redefined a feature of *A*, which causes an ambiguity in *D*. But then, as explained below, you will get an error message from the compiler, inviting you to select one of the two versions for use in *D*.

A case of redundant but harmless inheritance may occur when *A* is a class implementing general-purpose facilities like input or output (such as the class *STD_FILES* from the Kernel library), needed by *D* as well as *B*. It is enough for *D* to inherit from *B*: this makes *D* a descendant of *A*, giving it access to all the needed features. Inheriting redundantly will not, however, have any harmful consequences — in fact, it will have no consequences at all.

See "THE GLOBAL INHERITANCE STRUCTURE", 16.2, page 580.

Such involuntary and innocuous cases of repeated inheritance may also occur as a result of inheritance from universal classes *ANY* and *GENERAL*, studied in the next chapter.

The renaming rule

(This section introduces no new concept but gives a more precise formulation of the rules seen so far, and an explanatory example.)

We can now give a precise working of the rule prohibiting name clashes:

Definition: final name

The final name of a feature in a class is:

- For an immediate feature (that is to say, a feature declared in the class itself), the name under which it is declared.
- For an inherited feature that is not renamed, its final name (recursively) in the parent from which it is inherited.
- For a renamed feature, the name resulting from the renaming.

Single Name rule

Two different effective features of a class may not have the same final name.

A name clash occurs if two different features, both effective, still have the same name even after renaming subclauses have been taken into account. Such a name clash makes the class invalid, but is easy to correct by adding the proper renaming subclause.

The key word is *different* features. If a feature from a repeated ancestor is inherited from both parents under the same name, the sharing rule applies: only **one feature** is being inherited, so there is no name clash.

The prohibition of name clashes only applies to effective features. If one or more homonymous features are deferred, you can actually *merge* them since there is no incompatibility between implementations; the details will be seen shortly.

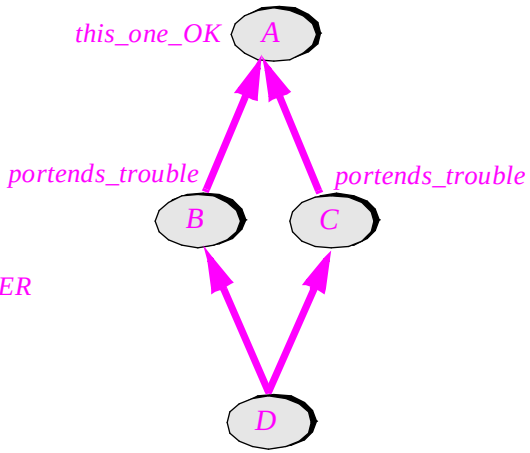
The rules are simple, intuitive and straightforward. To check our understanding one final time, let us build a simple example showing a legitimate case and an invalid case:

```
class A feature
        this_one_OK: INTEGER
end

class B inherit A feature
        portends_trouble: REAL
end

class C inherit A feature
        portends_trouble: CHARACTER
end

class D inherit
                -- This class is invalid!
        B
        C
end
```



That class *D* inherits *this_one_OK* twice — once from *B*, once from *C* — does **not** cause a name clash, since the feature will be shared; it is indeed the same feature, coming from *A*, in each case.

The two features called *portends_trouble*, however, deserve their name: they are different features, and so they cause a name clash, making class *D* invalid. (They have different types, but giving them the same type would not affect this discussion.)

It is easy to make class *D* valid through renaming; for example:

```
class D inherit
                -- This class is now quite valid.
        B
                rename portends_trouble as does_not_portend_trouble_any_more end
        C
end
```

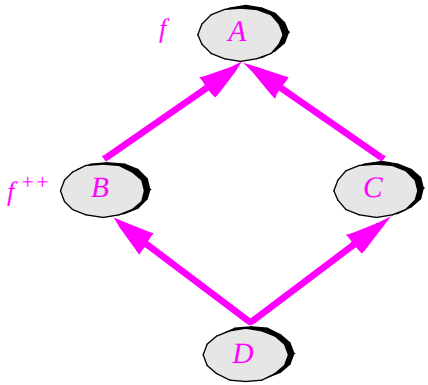
Conflicting redefinitions

In the cases seen so far only names could change along the various inheritance paths. What if some intermediate ancestor, such as *B* or *C* on the last figure, redeclares a feature that is then repeatedly inherited? Under dynamic binding there may be an ambiguity in *D*.

Two simple mechanisms, undefinition and selection, will solve the issue. As usual you will be invited to participate in the development of these mechanisms and will see that once a problem is stated clearly the language solution follows immediately.

Assume that somewhere along the way a repeatedly inherited feature gets redefined:

Redefinition causing potential ambiguity



Class *B* redefines feature *C* (this is the conventional meaning of the ++ symbol, as you will recall). So now you have two variants of *f* available in *D*: the redefined version from *B*, and the version from *C*, which here is the original version from *A*. (We might assume that *C* also redefines *f* in its own way, but this would bring nothing to the discussion except more symmetry.) This is different from all the previous cases, in which there was only one version of the feature, possibly inherited under different names.

What are the consequences? The answer depends on whether *D* inherits the two versions of *f* under the same name or different names, that is to say whether the repeated inheritance rule implies sharing or replication. Let us review the two cases in turn.

Conflicts under sharing: undefinition and join

Assume first that the two versions are inherited under the same name. This is the sharing: case: with just one feature name, there must be exactly one feature. Three possibilities:

S1 • If one of the two versions is deferred and the other effective, there is no difficulty: the effective version will serve to effect the other. Note that in the Single Name rule this case was explicitly permitted: the rule only prohibited name clashes between two effective features.

S2 • If both versions are effective, but each of them appears in a **redefine** subclause, there is no problem either: both inherited versions are merged into a new version, whose redefinition appears in the class.

S3 • But if the versions are both effective and not both redefined, we have a true name clash: class *D* will be rejected as violating the Single Name rule.

Often S3 will indeed reflect an error: you have created an ambiguity for a certain feature name, and you must resolve it. The usual resolution is to **rename** one of the two variants; then instead of sharing you get replication — two different features. This is the other main case, replication, studied next.

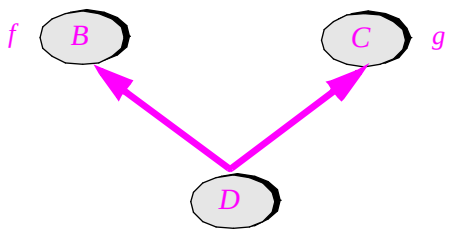
In some situations, however, you may want a more sophisticated resolution of the S3 conflict: letting one of the two variants, say the one from *B*, take over. Then the obvious solution is to transform this case into S1 by making one of the two variants deferred.

The rules on redefinition allow us to redefine an effective *f* into a deferred version; but they would force us to introduce an intermediate class, say *C'*, an heir of *C* whose only role is to redefine *f* into a deferred version; then we would make *D* inherit from *C'* rather than *C*. This is heavy and inelegant. Instead, we need a simple language mechanism: **undefine**. It will yield a new subclause in the inheritance part:

```
class D inherit
        B
        C
                undefine f end
feature
        …
end
```

If more than one subclause is present, **undefine** naturally comes after **rename** (since any undefinition should apply to the final name of a feature) but before **redefine** (since we should take care of any undefinition before we redefine anything).

A sign that a proposed language mechanism is desirable is, almost always, that it should solve several problems rather than just one. (Conversely, *bad* language mechanisms tend to cause as many problems, through their interactions with other language traits, as they purport to solve.) The undefinition mechanism satisfies this property: it gives us the ability to **join** features under multiple — not necessarily repeated — inheritance. Assume that we wish to combine two abstractions into one:



Two parents with features to be merged

We want *D* to treat the two features *f* and *g* as a single feature; this clearly requires that they have compatible signatures (number and types of arguments and result if any), and compatible semantics. Assuming that they have different names, and that we want to keep the *f* name, we can achieve the desired result by combining renaming with undefinition:

```
class D inherit
        B
        C
                rename
                        g as f
                undefine
                        f
                end
feature
        …
end
```

Here the victory of *B* is total: it imposes both the feature and the feature name. All other combinations are possible: we may get the feature from one of the parents and the name from the other; or we may rename both features to an entirely new name for *D*.

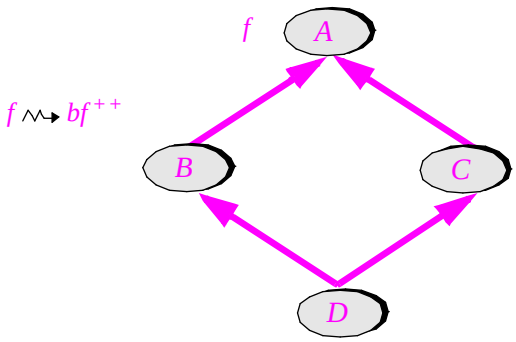
Another way to join features is more symmetric: replace both inherited versions by a new one. To achieve this, simply make sure that the features have the same final name, adding a **rename** subclause if necessary, and list them both in **redefine** subclauses, with a new declaration in the class. Then there is no illicit name clash (this is case S2 above), and both features are joined into the new version.

Note the versatility of the renaming mechanism (showing that it satisfies the just introduced criterion for good language traits): originally introduced as a technique for removing name clashes, it now enables us to *introduce* name clashes — name clashes of a desirable kind, resolved by undefining one of the inherited versions to let the other take over.

Conflicts under replication: selection

There remains to consider the case of conflicting redefinitions under replication, that is to say when the repeated descendant inherits the separately redefined features with different names, and they are both effective.

The need for selection



On the *B* branch in the figure, feature *f* is renamed *bf* and is also redefined. Favoring again simplicity over symmetry we assume no change in the *C* branch; renaming or redefining *f* in *C* would not affect the discussion. Also, note that the result would be the same if *B* redefined the feature without renaming it, the renaming then occurring at the *D* level. Let us assume this is not a case of join (which would arise if we redefined both features, under S2 above, or undefined one of them).

Because the features are inherited under different names *bf* and *f*, replication applies: *D* gets two separate features from the feature *f* of *A*. In contrast with previous cases of replication, these are not duplicates of the same feature, but different features.

Here, unlike in the sharing case, there is no name clash. But as the careful reader will have noted, a different problem arises (the last issue of repeated inheritance), due to dynamic binding. Assume that a polymorphic entity *a1* of type *A*, the common ancestor, becomes attached at run time to an instance of *D*, the common descendant. What then should the call *a1*.*f* do?

The rule of dynamic binding states that the version of *f* to apply is the one deduced from the type of the target object, here *D*. But now for the first time that rule is ambiguous: *D* has two versions — known locally as *bf* and *f* — of the original *f* of *A*.

The observation made in the case of name clashes, which led to the renaming mechanism, applies here too: we cannot, in an approach favoring clarity and reliability, let the compiler make the choice behind the scenes through some default rule. The author of the software must be in control.

This shows the need for a simple language mechanism to resolve the ambiguity:

```
class D inherit
        B
        C
                select f end
feature
        …
end
```

to trigger *C*'s version under dynamic binding for an entity of type *A*, and

```
class D inherit
        B
                select bf end
        C
feature
        …
end
```

to select *B*'s version instead. The **select** clause will naturally appear after **rename**, **undefine** and **redefine** if present (you select variants once everything has been named and defined). Here is the rule governing its usage:

The case in which both are redefined corresponds to S2, page 552.

Select rule

A class that inherits two or more different effective versions of a feature from a repeated ancestor, and does not redefine them both, must include exactly one of them in a **select** clause.

The **select** resolves the ambiguity once and for all: proper descendants of the class do not need to repeat it (and should not).

Selecting everything

Every redefinition conflict must be resolved through **select**. When combining two classes that cause several such conflicts, you may want one of the classes to win all or most of these conflicts. This happens in particular with inheritance of the "marriage of convenience" form, as illustrated by *ARRAYED_STACK* inheriting from *STACK* and *ARRAY*, if the parents have a common ancestor. (In the Base libraries, both classes cited are indeed distant descendants of a general *CONTAINER* class.) In such a case, since one of the parents — what has been called the noble parent, here *STACK* — provides the specification, you will probably want to resolve all conflicts, or most of them, in its favor.

The following important notational facility simplifies your task in such cases, by avoiding the need to list all conflicting features individually. At most one of the parent listings in the **inherit** clause may be of the form

```
SOME_PARENT
        select all end
```

The effect is simply, as suggested by the keyword **all**, to resolve in favor of *SOME_PARENT* all redefinition conflicts — more precisely all the conflicts that might remain after the application of other **select** subclauses. This last qualification means that you can still request some other parent's version for certain features.

Keeping the original version of a redefined feature

(This section describes a more specialized technique and may be skipped on first reading.)

"Using the original version in a redefinition", page 493.

In the introduction to inheritance we saw a simple construct allowing a redefined feature to call the original version: *Precursor*. The repeated inheritance mechanism, through its support for feature duplication, provides a more general (but also heavier) solution in those rare cases for which the basic mechanism does not suffice.

Consider again the earlier example: *BUTTON* inheriting from *WINDOW* and redefining *display* as

```
display is
            -- Display button on the screen.
        do
            window_display
            special_button_actions
        end
```

where *window_display* takes care of displaying the button as if it were a normal window, and *special_button_actions* adds button-specific elements such as displaying the button's border. Feature *window_display* is exactly the same as the *WINDOW* version of *display*.

We have seen how to write *window_display* simply as *Precursor*. (If there is any ambiguity, that is to say if two or more parents redefine their *display* routine into the new one, the selected parent will appear in double braces, as in {{*WINDOW*}} *Precursor*.) We can achieve the same goal, although less simply, through repeated inheritance:

```
indexing
        WARNING: "This is a first attempt — this version is invalid!"
class BUTTON inherit
        WINDOW
                redefine display end
        WINDOW
                rename display as window_display end
feature
        …
end -- class BUTTON
```

Because one of the branches renames *display*, the repeated inheritance rule indicates that *BUTTON* will have two versions of that feature, one redefined and keeping the original name, the other not redefined but having the name *window_display*.

As indicated, this is almost valid but not quite: we need a **select**. If (as will usually be the case) we want to select the redefined version, this will give:

```
indexing
        note: "This the (valid!) repeated inheritance scheme for continuing to use %
                %%the original version of a redefined feature"
class BUTTON inherit
        WINDOW
                redefine
                        display
                select
                        display
                end
        WINDOW
                rename
                        display as window_display
                export
                        {NONE} window_display
                end
feature
        …
end -- class BUTTON
```

The selection

If several features need this scheme, you can list them together (in other words, you do not need to inherit more than twice from the parent). Often you will want to resolve all conflicts in favor of the redefined versions; in that case, use **select all**.

> The **export** clause (studied only in the next chapter, although there is little more to it than shown here) changes the export status of an inherited feature: *WINDOW* probably exported the original *display*, now known as *window_display*, but *BUTTON* makes it secret. Although *window_display* is a full-fledged feature of the class, which needs it for its internal purposes, clients have no use for it. As discussed in earlier examples, exporting the original version of an inherited feature might make the class formally incorrect if that version does not satisfy the new class invariant.
>
> To apply hiding to all features inherited along a certain branch you can, here too, use the keyword **all**, as in **export** {*NONE*} **all**.

This pattern of exporting only the redefined version, making the original secret under a new name, is the most common. It is not universal; the heir class sometimes needs to export both versions (assuming the original does not violate the invariant), or to hide both.

How useful is this technique using repeated inheritance to keep the original version of a redefined feature? Usually you do not need it: the *Precursor* construct suffices. You should use repeated inheritance when you do not just require the old version for implementing the redefined one, but want to keep it, along with the redefined version, as one of the features of the new class.

Remember that if both are exported they must both make sense for the corresponding abstraction; in particular, they must preserve the invariant.

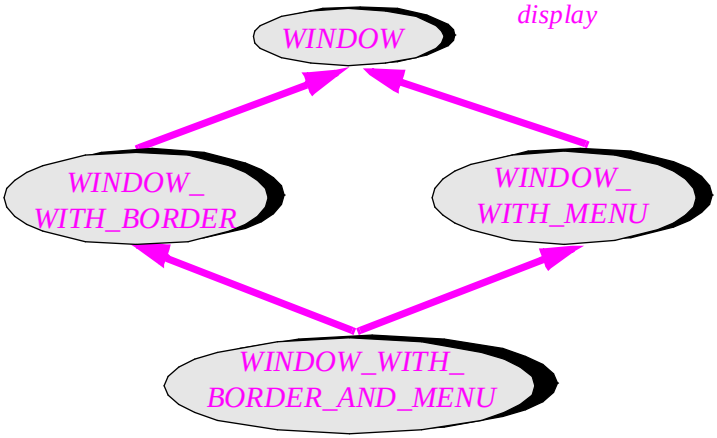
An advanced example

Here is an extensive example showing various aspects of repeated inheritance at work.

The problem, similar in spirit to the last example, comes from an interesting discussion in the basic book on C++ [Stroustrup 1991].

Consider a class *WINDOW* with its *display* procedure and two heirs, *WINDOW_WITH_BORDER* and *WINDOW_WITH_MENU* representing the abstractions suggested by their names. Each redefines *display* so that it will first perform the standard window display, and then display the border in the first case, and the menu cells in the second.

We may want to describe windows that have both a border and a menu; hence the use of repeated inheritance for class *WINDOW_WITH_BORDER_AND_MENU*.



Window variants

In class *WINDOW_WITH_BORDER_AND_MENU* we will again redefine *display*; here the redefined version should apply the standard window display, then display the border, then display the menu.

The original *WINDOW* class has the following form:

```
class WINDOW feature
        display is
                -- Display window (general algorithm)
            do
                ...
            end
        ... Other features ...
end
```

For an heir such as *WINDOW_WITH_BORDER* we need to apply the original *display* and add border display. We do not need repeated inheritance here, but can simply rely on the *Precursor* construct:

```
class WINDOW_WITH_BORDER inherit
        WINDOW
            redefine display end
feature -- Output
        display is
                -- Draw window and its border.
            do
                Precursor
                draw_border
            end
feature {NONE} -- Implementation
        draw_border is do ... end
        ...
end
```

Note the addition of a procedure *draw_border* which displays the border. It has been hidden from clients (exported to *NONE*), since from the outside it makes no sense to display the border only. Class *WINDOW_WITH_MENU* is exactly symmetrical:

```
class WINDOW_WITH_MENU inherit
        WINDOW
                redefine display end
feature -- Output
        display is
                        -- Draw window and its menu.
                do
                        Precursor
                        draw_menu
                end
feature {NONE} -- Implementation
        draw_menu is do … end
        …
end
```

It remains to write the common heir *WINDOW_WITH_BORDER_AND_MENU* of these two classes, a repeated descendant of *WINDOW*. Here is a first attempt:

```
indexing
        WARNING: "This is a first attempt — this version will not work properly!"
class WINDOW_WITH_BORDER_AND_MENU inherit
        WINDOW_WITH_BORDER
                redefine display end
        WINDOW_WITH_MENU
                redefine display end
feature
        display is
                        -- Draw window and its border.
                do
                        {{WINDOW_WITH_BORDER}} Precursor
                        {{WINDOW_WITH_MENU}} Precursor
                end
        …
end
```

Note the need to name the parent in each use of *Precursor*: each parent has a *display* feature, each redefined into the same new *display* (otherwise we would have an invalid name clash, of course), so in each case we must say which one we want.

But, as Stroustrup notes (for a different solution), this is not correct: both parent versions call the original *WINDOW* version, which will end up being called twice, possibly producing garbled output. To get a correct form, we may among other solutions let the new class inherit directly from *WINDOW*, making it a triple descendant of that class:

```
indexing
        note: "This is a correct version"
class WINDOW_WITH_BORDER_AND_MENU inherit
        WINDOW_WITH_BORDER
                redefine
                        display
                export {NONE}
                        draw_border
                end
        WINDOW_WITH_MENU
                redefine
                        display
                export {NONE}
                        draw_menu
                end
        WINDOW
                redefine display end
feature
        display is
                        -- Draw window and its border.
                do
                        {{WINDOW}} Precursor
                        draw_border
                        draw_menu
                end
        …
end
```

Note that for good measure we have made features *draw_border* and *draw_menu* hidden in the new class, as there does not seem to be any reason for clients of *WINDOW_WITH_BORDER_AND_MENU* to call them directly.

In spite of its lavish use of repeated inheritance, this class does not need any **select** since it redefines all inherited versions of *display* into one. This is the benefit of using *Precursor* rather than feature replication.

A good way to test your understanding of repeated inheritance is to rewrite this example without making use of the *Precursor* construct, that is to say by using repeated inheritance to obtain feature replication at the level of the two intermediate classes. You will, of course, need **select** subclauses.

Exercise E15.10, page 568.

In the version obtained above, there is sharing only, no replication. Let us extend Stroustrup's example by assuming that *WINDOW* also has a query *id* (perhaps an integer) used to identify each window. If each window is identified at most once, then *id* will be shared and we do not need to change anything. But if we want to keep track separately of instances of each window type, an instance of *WINDOW_WITH_BORDER_AND_MENU* will have three separate identifiers. The new class combines sharing with replication:

The only changes are the additions marked with an arrow.

```
indexing
        note: "More complete version with separate identifiers"
class WINDOW_WITH_BORDER_AND_MENU inherit
        WINDOW_WITH_BORDER
                rename
                        id as border_id                    <--
                redefine
                        display
                export {NONE}
                        draw_border
                end
        WINDOW_WITH_MENU
                rename
                        id as menu_id                      <--
                redefine
                        display
                export {NONE}
                        draw_menu
                end
        WINDOW
                rename
                        id as window_id                    <--
                redefine
                        display
                select                                     <--
                        window_id
                end
feature
        …. The rest as before …
end
```

Note the need for **select**ing one of the versions of *id*.

Repeated inheritance and genericity

To finish this review of repeated inheritance, we must consider a specific case which could cause trouble if left unchecked. It arises for features involving formal generic parameters. Consider the following scheme (which could also arise with indirect repeated inheritance):

```
class A [G] feature
    f: G; …
end

class B inherit
    A [INTEGER]
    A [REAL]
end
```

In class *B*, the repeated inheritance rule would imply that *f* is shared. But this leaves an ambiguity on its type: does it return an integer or a real? The same problem would occur if *f* were a routine with an argument of type *G*.

Such an ambiguity is not acceptable. Hence the rule:

Genericity in Repeated Inheritance rule

The type of any feature that is shared under the repeated inheritance rule, and the type of any of its arguments if it is a routine, may not be a generic parameter of the class from which the feature is repeatedly inherited.

You can remove the ambiguity by renaming the offending feature at the point of inheritance, to get duplication rather than renaming.

Rules on names

(This section only formalizes previously seen rules, and may be skipped on first reading.)

We have seen that name clashes are prohibited when they could cause ambiguity, but that some cases are valid. To finish off this presentation of multiple and repeated inheritance without leaving any ambiguity, it is useful to summarize the constraints on name clashes with a single rule:

Name clashes: definition and rule

In a class obtained through multiple inheritance, a *name clash* occurs when two features inherited from different parents have the same final name.

A name clash makes the class invalid *except* in any of the following cases:

N1 • The two features are inherited from a common ancestor, and none has been redeclared from the version in that ancestor.

N2 • Both features have compatible signatures, and at least one of them is inherited in deferred form.

N3 • Both features have compatible signatures, and they are both redefined in the class.

Case N1 is the sharing case under repeated inheritance.

In case N2, a feature is "inherited in deferred form" if it was deferred in the parent, or if it was effective but the class **undefine**s it.

Cases N2 and N3 have been separated but can be merged into a single case, the **join** case. Considering *n* features (*n* >= 2) rather than just two, these cases arise when the class gets *n* features with the same name, and compatible signatures, from its various parents. The name clash is valid if we can let the inheritance join all of these features into one, without any ambiguity. This means that:

- You can have any number of deferred features among the lot since they will not cause any conflicting definitions. (As noted, a deferred feature is either one that was already deferred, or one that the class undefines.)
- If exactly one of the features is effective, it imposes its implementation to the others.
- If two or more features are effective, the class must provide a common redefinition for all of them. (An example was the joining in *WINDOW_WITH_BORDER_AND_MENU* of the *display* procedures of the three parents.) The redefinition will also, of course, serve as effecting for any deferred feature participating in the clash.

Here then is the precise rule on the *Precursor* (…) construct. If a redefinition uses a precursor version, case N3 is the only one causing ambiguity as to whose version is intended. Then you must resolve the ambiguity by writing the precursor call as {{*PARENT*}} *Precursor* (…) where *PARENT* is the name of the desired class. In all other cases (simple inheritance, or multiple outside of N3) naming the parent is optional.

15.5 DISCUSSION

Let us probe further the consequences of some of the decisions made in this chapter.

Renaming

Any language that has multiple inheritance must deal with the problem of name clashes. Since we cannot and should not require developers to change the original classes, only two conventions are possible besides the solution described in this chapter:

- Require clients to remove any ambiguity.
- Choose a default interpretation.

With the first convention, a class *C* inheriting two features called *f*, one from *A* and one from *B*, would be accepted by the compiler, possibly with a warning message. Nothing bad would happen unless a client of *C* contained something like

```
x: C
… x.f …
```

which would be invalid. The client would have to qualify the reference to *f*, with a notation such as *x* . *f* | *A* or *x* . *f* | *B*, to specify one of the variants.

This solution, however, runs contrary to one of the principles emphasized in this chapter: that the inheritance structure leading to a class is a private affair between the class and its ancestors, not relevant for clients except through its influence on polymorphic uses. When I use service *f* from *C*, I should not need to know whether *C* introduced it itself or got it from *A* or *B*.

With the second convention, *x*.*f* is valid; the underlying language mechanisms select one of the variants, based on some criterion such as the order in which *C* lists its parents; a notation may be available for requesting another variant explicitly.

This approach has been implemented in several Lisp-based languages supporting multiple inheritance. But it is dangerous to let some underlying system choose a default semantics. The solution is also incompatible with static typing: there is no reason why two features with the same name in different parents should be typewise compatible.

The renaming mechanism solves these problems; it brings other benefits, such as the ability to rename inherited features with names that are meaningful to clients.

O-O development and overloading

This chapter's discussion of the role of names brings the final perspective on the question of in-class name overloading, complementing the preliminary observations made in earlier chapters.

Recall that in languages such as Ada (83 and 95) you can give the same name to different features within the same syntactical unit, as in

```
infix "+" (a, b: VECTOR) is …
infix "+" (a, b: MATRIX) is …
```

which could both appear in the same Ada package. C++ and Java have made the same possibility available within a single class.

An earlier presentation called this facility **syntactic** overloading. It is a static mechanism: to disambiguate a given call, such as *x* + *y*, it suffices to look at the types of the arguments *x* and *y*, which are apparent from the program text.

"Syntactic overloading", page 93.

Object technology introduces a more powerful of overloading: **semantic** (or *dynamic*) overloading. If classes *VECTOR* and *MATRIX* both inherit a feature

```
infix "+" (a: T) is …
```

from a common ancestor *NUMERIC*, and each redeclares it in the appropriate way, then a call *x* + *y* will have a different effect depending on the dynamic type of *x*. (Infix features are just a notational convenience: with a non-infix feature the call *x* + *y* would be written something like *x*.*plus* (*y*).) Only at run time will the ambiguity be resolved. As we know, this property is key to the flexibility of O-O development.

Semantic overloading is the truly interesting mechanism. It allows us to use the same name, in different classes, for variants of what is essentially **the same operation** — such as addition from *NUMERIC*. The next chapter's rules on assertions will make it even more clear that a feature redeclaration must keep the same fundamental semantics.

Does this leave a role for syntactic overloading in object technology? It is hard to find any. One can understand why Ada 83, which does not have classes, should use syntactic overloading. But in an object-oriented language, to let developers choose the same name for **two different operations** is to create the possibility of confusion.

The problem is that the syntactic form of overloading clashes with the semantic form provided by polymorphism and dynamic binding. Consider a call $x.f\ (a)$. If it follows the possibly polymorphic assignments $x := y$ and $a := b$, the result is exactly the same, in the absence of renaming, as that of $y.f\ (b)$, even if y and b have other types than x and a. But with overloading this property is not true any more! f may be the overloaded name of two features, one for the type of a and one for the type of b. Which rule takes precedence, syntactic overloading or the O-O concept of dynamic binding? (Probably the former, but not until it has fooled a few developers, novice or not.) To make things worse, the base class of y's type may redefine either or both of the overloaded features. The combinations are endless; so are the sources of confusion and error.

What we are witnessing here is the unpleasant consequences of the interaction between two separate language traits. (A language addition, as noted earlier in this chapter on another topic, should whenever possible *solve* new problems beyond its original purpose — not create new problems through its interaction with other mechanisms.) A prudent language designer, having toyed with a possible new facility, and encountering such incompatibilities with more important properties of the design, quickly retreats.

What, against these risks, is the potential benefit of syntactic overloading? On careful examination it seems dubious to start with. A simple principle of readability holds that within the same module a reader should have absolutely no hesitation making the connection between a name and the meaning of that name; with in-class overloading, this property collapses.

A typical example — sometimes mentioned in favor of overloading — is that of features of a *STRING* class. To append another string or a single character you will, in the absence of overloading, use different feature names, as in *s1.add_string* (*s2*) and *s1.add_character* ('A'), or perhaps, using infix operators, *s* := *s1* ++ *s2* and *s* := *s1* + 'A'. With overloading, you can use a single name for both operations. But is this really desirable? Objects of types *CHARACTER* and *STRING* have quite different properties; for example appending a character will always increase the length by 1; appending a string may leave the length unchanged (if the appended string was empty) or increase it by any amount. It seems not only reasonable but desirable to use different names — especially since the confusions cited above are definitely possible (assume that *CHARACTER* inherits from *STRING* and that another descendant redefines *add_string* but not *add_character*.)

"Multiple creation and overloading", page 239.

Finally, we have already encountered the observation that even if we wanted overloading we would in general need a different disambiguating criterion. Syntactic overloading distinguishes competing routines by looking at their signatures (numbers and types of arguments); but this is often not significant. The typical example was the creation procedures for points, or complex numbers: *make_cartesian* and *make_polar* both take two arguments of type *REAL* — to mean completely different things. You cannot use

overloading here! The routines' signatures are irrelevant. To express that two features are different, we should use the obvious technique, the same that we apply in everyday life to express that two things or concepts are different: give them different names.

> For creation operations ("constructors") such as *make_cartesian* and *make_polar* the Java and C++ solution is particularly ironic: you **may not** give them different names but are forced to rely on overloading, using the class name. I have been unable to find a good solution to this problem other than adding an artificial third argument.

In summary: syntactic (in-class) overloading appears in an object-oriented context to create many problems for no visible benefit. (Some methodological advice to users of languages such as C++, Java and Ada 95: do not use this facility at all, except for cases such as multiple constructor functions in which the language leaves no other choice.) In a consistent and productive application of object technology we should stick to the rule — simple, easy to teach, easy to apply and easy to remember — that, within a class, every feature has a name and every feature name denotes one feature.

15.6 KEY CONCEPTS INTRODUCED IN THIS CHAPTER

- The construction-box approach to software construction favored by object technology requires the ability to combine several abstractions into one. This is achieved by multiple inheritance.
- In the simplest and most common cases of multiple inheritance, the two parents represent disjoint abstractions.
- Multiple inheritance is frequently needed, both for system modeling and for everyday software development, in particular the construction of reusable libraries.
- Name clashes under multiple inheritance should be removed through renaming.
- Renaming also serves to provide classes with locally adapted terminology for inherited features.
- Features should be distinguished from feature names. The same feature can be known under different names in different classes. A class defines a mapping from feature names to features.
- Repeated inheritance, an advanced technique, arises as a result of multiple inheritance when a class is a descendant of another through two or more paths.
- Under repeated inheritance, a feature from the common ancestor yields a single feature if it is inherited under a single name, separate features otherwise.
- Competing versions from a common ancestor must be disambiguated, for dynamic binding, through a **select** subclause.
- The replication mechanism of repeated inheritance should not replicate any feature involving generic parameters.
- In an object-oriented framework, the semantic form of overloading provided by dynamic binding is more useful than syntactic overloading.

15.7 BIBLIOGRAPHICAL NOTES

The renaming mechanism and the repeated inheritance rules originated with the notation of this book. The undefinition mechanism is an invention of Michael Schweitzer, and the selection mechanism an invention of John Potter, both in unpublished correspondence.

Exercise E15.8, page 568.

The walking menu example comes from [M 1988c].

EXERCISES

E15.1 Windows as trees

Class *WINDOW* inherits from *TREE* [*WINDOW*]. Explain the generic parameter. Show that it yields an interesting clause in the class invariant.

E15.2 Is a window a string?

A window has an associated text, described by an attribute *text* of type *STRING*. Rather than having this attribute, should *WINDOW* be declared as an heir to *STRING*?

E15.3 Doing windows fully

Complete the design of the *WINDOW* class, showing exactly what is needed from the underlying terminal handling mechanism.

E15.4 Figure iterators

See also "Iterators", page 848.

The presentation of class *COMPOSITE_FIGURE* mentioned the possibility of using iterator classes for all operations that perform a certain operation on a composite figure. Develop the corresponding iterator classes. (**Hint**: [M 1994a] presents library iterator classes which provide the basic pattern.)

E15.5 Linked stacks

Write the class *LINKED_STACK* which describes a linked list implementation of stacks, as an heir to both *STACK* and *LINKED_LIST*.

E15.6 Circular lists and chains

Explain why the *LIST* class may not be used for circular lists. (**Hint**: a look at the assertions, benefiting from the discussion at the beginning of the next chapter, may help.). Define a class *CHAIN* that can be used as parent both to *LIST* and to a new class *CIRCULAR* describing circular lists. Update *LIST* and if necessary its descendants accordingly. Complete the class structure to provide for various implementations of circular lists.

E15.7 Trees

One way to look at a tree is to see it as a recursive structure: a list of trees. Instead of the technique described in this chapter, where *TREE* is defined as heir to both *LINKED_LIST* and *LINKABLE*, it seems possible to define

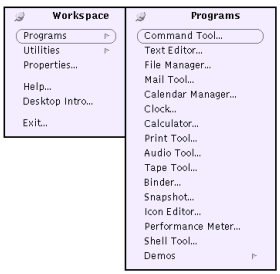
```
class TREE [G] inherit
        LIST [TREE [G]]
feature ... end
```

Can you expand this definition into a usable class? Compare it with the method used in the discussion of this chapter.

E15.8 Walking menus

Window systems offer a notion of menu, which we can cover through a class *MENU*, with a query giving the list of entries and commands to display the menu, move to the next entry etc. Since menus are made of entries we also need a class *MENU_ENTRY* with queries such as *parent_menu* and *operation* (the operation to execute when a user selects the entry), and commands such as *execute* (which executes *operation*).

Many systems offer cascading menus, also called "walking menus", where selecting an entry causes the display of a submenu. The figure illustrates a walking menu under Sun's Open Windows manager, where selecting the entry **Programs** brings up a submenu:



Walking menus

(*The last entry of the submenu*, **Demos**, *denotes in turn a submenu*.)

Show how to define the class *SUBMENU*. (**Hint**: a submenu is a menu and a menu entry, whose *operation* must display the submenu.)

Could this notion be described elegantly in a language with no multiple inheritance?

E15.9 The flat precursor

What should the flat form of a class show for an instruction using the *Precursor* construct?

E15.10 Repeated inheritance for replication

Write the *WINDOW_WITH_BORDER_AND_MENU* class without recourse to the *Precursor* construct, using replication under repeated inheritance to gain access to the parent version of a redefined feature. Make sure to use the proper **select** subclauses and to give each feature its proper export status.