Feature adaptation

10.1 OVERVIEW

Chapter 6 introduced inheritance as a module enrichment technique. You inherit from a class out of sheer mercenary interest: you want its features. But that doesn't necessarily mean accepting all these features at face value.

A key attraction of the inheritance mechanism is that it lets you tune inherited features to the context of the new class. This is known as feature adaptation. The present discussion covers the principal mechanisms, leaving to a later one some important complements related to repeated inheritance.

→ *Chapter 16 presents repeated inheritance.*

This chapter is the longest of this book, which should not be a surprise since it explores in full detail some of the most fascinating aspects of object technology: how to play mix and match with software components, taking advantage of the best features of existing classes while refining, adapting or overriding what is not exactly suited to your new need. Only a few basic concepts are involved, but they interact in diverse and powerful ways.

So make sure you have a comfortable armchair and a big cup of coffee, and for the 50 pages of this chapter be prepared to question, implement, override, rename, merge or otherwise wring all those features that your ancestors, for better or worse, bequeathed to you.

There are actually 40 more pages of wringing in chapter 16.

10.2 TERMINOLOGY: REDECLARATION, REDEFINITION, EFFECTING

Our major focus will be the two **redeclaration** mechanisms that help adapt inherited features to the local context of a class:

- **Redefinition**, which may change an inherited feature's original implementation, signature or specification.
- **Effecting**, which provides an implementation (or **effective** version) for a feature that did not have one in the parent. The parent's version, deprived of any implementation, but with a signature and specification, is said to be **deferred**; deferred features play an important role in analysis and design, which this chapter will explain.

The term "effecting" sometimes surprises at first, but achieves consistent terminology: to effect a a feature is to make it effective.

The purpose of redefining a feature is often to extend (rather than discard) its original implementation. We will see how the **precursor** mechanism enables you, in a redefinition, to reuse and extend the original version.

Two closely related facilities, which the discussion will address in detail, are the possibility of of **undefining** an inherited feature, to forget its original implementation, and of merging abstractions by **joining** two or more features inherited from different parents.

Another adjacent concept is **repeated inheritance**, which enables a class to inherit twice or more from a given ancestor, letting the designer control what happens to the common feature heritage. This topic is important enough to deserve a chapter of its own, coming only later in this book, after the conformance chapter, since repeated inheritance rules rely extensively on those of type conformance.

→ *Chapter 16.*

Although with the present chapter the major language constructs involving inheritance will have been introduced, we are still missing an important part of the picture. To grasp the full extent and practicality of the techniques introduced below, you will need to understand *polymorphism* and *dynamic binding*, studied in subsequent chapters. Together, these notions are responsible for some of the most powerful characteristics of the object-oriented method.

→ *"POLYMORPHISM", 22.11, page 598*; *"DYNAMIC BINDING", 23.12, page 630.*

10.3 REDECLARING INHERITED FEATURES: WHY AND HOW

A class inheriting from another may add new features of its own. But what about the old ones? So far the presentation has assumed that an heir will be happy enough to obtain every inherited feature "as is" from a parent. To be sure, the heir may *rename* the feature, but this does not change it; the effect is simply to make it available to the client's dependents under a name that is better suited to the local context.

← *"RENAMING", 6.9, page 180.*

Inheritance offers more. When you inherit a set of features, you may want to adapt those whose original *specification* or *implementation* did not take advantage of the heir's specific properties.

Redefinition is the basic method for achieving such an adaptation. By redefining an inherited feature you may give it a new implementation, a new signature, or a new set of assertions, as long as you follow the applicable rules to ensure that the new version remains compatible with the old one as seen by clients. You may even redefine a function into an attribute, switching from an algorithmic representation to one that simply stores feature values. Every proper descendant of a class may provide its own alternative redefinition.

In some cases, the original form of a routine does not provide any default implementation at all; this is an explicit invitation for proper descendants to offer various implementations. Such unimplemented features, and the classes that introduce them, are said to be **deferred**; proper descendants may then **effect** those features (make them **effective**).

In the software construction process, classes and features may in fact remain deferred for a long time, providing a high-level notation for system analysis and design.

The basic terminology has already been previewed:

Redeclare, redeclaration

A class **redeclares** an inherited feature if it redefines or effects it.

A declaration for a feature *f* is a **redeclaration** of *f* if it is either a redefinition or an effecting of *f*.

This definition relies on two others, appearing below, for the two cases: *redefinition* and *effecting*.

Be sure to distinguish *redeclaration* from *redefinition*, the first of these cases. Redeclaration is the more general notion, redefinition one of its two cases; the other is *effecting*, which provides an implementation for a feature that was deferred in the parent. In both cases, a redeclaration does not introduce a new feature, but simply overrides the parent's version of an inherited feature.

redeclaration:

In the case of a redefiniti

Getting the full power of deferred features requires two more mechanisms:

- Sometimes a class will be able to merge two or more features that it inherits from separate parents; in so doing the class combines several abstractions into one. This is the join mechanism.
- In some cases, as you inherit an effective feature from a parent, you may want to discard the inherited implementation altogether, recanting all the sins of its earlier effective life. This is the process of undefinition, which turns an effective feature into a born-again deferred feature.

The following sections explore redefinition, deferred features, undefinition and join. The discussion will first explain these facilities and their role in software analysis, design and implementation. The second part of the chapter, which you may skip on first reading, gives the more formal set of corresponding syntactic rules and validity constraints, together with the resulting semantic definitions.

The formal part starts with 10.25, page 300.

10.4 FEATURE ADAPTATION CLAUSES

For a start, let us just refresh our memory as to the syntactical context of this discussion: the Inheritance clause of a class declaration, which may contain one or more Parent parts. Here is a simplified form of the the beginning of class *TWO_WAY_TREE* in EiffelBase:

← *Another descendant of TREE, class FIXED_TREE, served to illustrate inheritance basics in "AN INHERITANCE PART", 6.2, page 167*

```
note
        … (Notes clause omitted)…
class TWO_WAY_TREE [T] inherit
        TREE [T]
                redefine
                        higher, …
                end
        BI_LINKABLE [T]
                rename
                        … (Rename subclause omitted)…
                redefine
                        put_between
                end
        TWO_WAY_LIST [like Current]
                rename
                        … (Rename subclause omitted)…
                redefine
                        first_child, update_after_insertion,
                        duplicate, merge_right, merge_left
                end
feature
                … Rest of class omitted …
```

Each Parent part is relative to one of the class's parents and may include a Feature_adaptation subclause (optional, but present for all three parents above). Here again is the corresponding syntax:


```
Inheritance ≜ inherit Parent_list
Parent_list ≜ "{Parent ";" … }
Parent ≜ "Class_type
         [Feature_adaptation]
Feature_adaptation ≜ [Rename]
                     [New_exports]
                     [Undefine]
                     [Redefine]
                     [Select]
                     end
```

← *The original presentation of this syntax is on page 169.*

The Rename and New_exports clauses have been discussed in previous chapters. The next sections explain Redefine, Undefine and Select.

← *"RENAMING", 6.9, page 180; "Adapting the export status of inherited features", , page 200.*

10.5 WHY REDEFINE?

The first mechanism to study is feature redefinition, which allows you to to change some aspects of an inherited feature.

Assume you write a class *C* that describes a specific variant of the concepts covered by an existing class *B*. *C* will be an heir of *B*. You may find that, for this variant, the inherited version of a certain feature *f* is not appropriate any more. This sets the stage for redefining *f* in *C*.

Besides its name, a feature is characterized by three properties:

← *"FEATURE DECLARATIONS: SYNTAX", 5.10, page 140.*

- The feature has a **signature**, defined by the number and type of its arguments and result, if any.
- It either is deferred or has an **implementation**, including the choice between attribute or routine, external or not, and for a non-external routine the Routine_body, Local_declarations and Rescue.
- It has a **specification**, defining the feature's **contract**: Precondition and Postcondition (for routines only).

A routine may also have a Header_comment *and an* Obsolete *clause, which a redefinition may change.*

A feature redefinition may affect one or more of these three aspects. In general, a change of specification implies a change of implementation.

There are two possible reasons, *correctness* and *efficiency*, for redefining a feature:

- The original version may perform actions or compute results that are incorrect for the new class, for example because they do not update some of the new attributes.

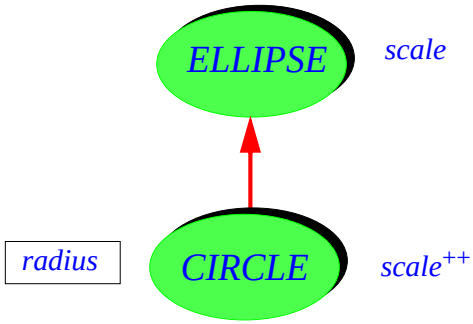
- If the original version is still appropriate, it may not be efficient enough, because it fails to take advantage of specific properties of the new class.

Signature redefinition falls in the correctness category: the types of arguments or results, as originally declared, are not appropriate for the new class. *Implementation* redefinition may be for correctness, efficiency or both. A change of *specification* involves correctness since it means the redefined version offers a new contract to its clients.

10.6 REDEFINITION EXAMPLES

To get a good feel for redefinition, let us look at a pair of simple examples illustrating each of the two purposes cited.

As a case of redefinition for correctness, assume a class *CIRCLE*, inheriting from *ELLIPSE* and adding an attribute *radius*:



Inheritance structure for circles and ellipses

As always, it is useful to check that we are not misusing inheritance. Here there is hardly any doubt that the structure is right: every circle may be viewed as an ellipse that happens to have only one focus.

Let *scale* be the procedure that scales a figure by a certain ratio. Since attribute *radius* is not present in *ELLIPSE*, the version of *scale* inherited from *ELLIPSE* does not update the value of that attribute. Class *CIRCLE* must redefine *scale* to make sure that it updates not just the attributes inherited from *ELLIPSE*, but also the specific *CIRCLE* attributes such as *radius*. (The problem would not arise if *radius* was a function, defined in terms of attributes inherited from *ELLIPSE*, rather than an attribute.)

As illustrated by the figure, the graphical convention for a redefined feature uses two plus signs after the feature's name, as in *scale*$^{++}$.

Now an example of redefinition for efficiency. Class *CIRCLE* may redefine as follows the function *contains* which determines whether a point is inside a closed figure:

```
contains (p: POINT): BOOLEAN
            -- Is p inside circle?
        require
            point_exists: p /= Void
        do
            Result :=(origin.distance (p) <= radius)
        ensure
            … Postcondition omitted …
        end
```

ELLIPSE has a version of *contains* too. Because an ellipse is a more general figure than a circle, the *ELLIPSE* version is more complex than the above; it would still be correct for circles, but less efficient since it does not take advantage of the special properties of circles. Redefinition solves the problem.

10.7 THE REDEFINITION CLAUSE

Whether for correctness or efficiency, the redefinition of a feature must be explicitly announced in a Redefine subclause of the Feature_adaptation for the corresponding parent, as in

```
class CIRCLE inherit
        ELLIPSE
            rename
                …
            redefine
                scale, contains,…
            end
… Rest of class omitted…
```

The names given in the Redefine subclause must be the final names of features inherited from the given parent. (In other words, these are the names *after* any renaming; this is easy to remember since the Rename clause always appears before **redefine** and other feature adaptation clauses.)

← *"Final name" was defined on page 183.*

Such a Redefine subclause allows — and requires — class *CIRCLE* to include (in a Feature_clause) new feature declarations, such as given above, for *scale*, *contains* and others listed after the keyword **redefine**. These declarations will override the ones inherited from the parent, here *ELLIPSE*. Without the Redefine subclause, such declarations would make *CIRCLE* invalid, since it would now have two features called *scale*, *contains* etc., a case of invalid **name clash**.

→ *"NAME CLASHES", 10.23, page 290*

To discuss redefinition it will be convenient to refer to the "precursor" of an inherited feature — its original form in the parent:

Precursor (initial definition)

If a class inherits a feature from a parent, either keeping the feature unchanged or redefining it, the parent's version of the feature is called the **precursor** of the feature.

With the mechanisms seen so far, every feature of a parent yields a feature in the heir; so every inherited feature has **one** precursor. Mechanisms explored later — joining of deferred features, and sharing under repeated inheritance — may result in the merging of two or more parent features into just one heir feature. This will require extending the definition to account for features having more than one precursor.

→ *"Precursor (joined features)", page 309. See also the more formal definition, page 465.*

10.8 REDEFINITION IN THE SOFTWARE PROCESS

(This section introduce no new language concept but broadens the discussion by presenting methodological aspects.)

Before proceeding with more technical aspects of redefinition, it is useful to reflect a little on the implications of this notion for object-oriented software engineering. Feature redefinition is part of the answer to a major software engineering issue: reconciling reusability with extendibility.

In software, it is seldom satisfactory to reuse an element exactly as it is; often, you must also adapt it to a specific context. With redefinition, as suggested by the simple examples above, you can keep those features that are still appropriate for the new context, while overriding the implementations of those which need to be adapted.

The ability to change the signature of an inherited routine, studied below, is also essential to the smooth functioning of Eiffel's type system.

It is useful to compare this technique with another of the mechanisms for adapting an inherited feature: renaming. The distinction to keep in mind is between a *feature* and a *feature name*:

← *"RENAMING", 6.9, page 180.*

- A feature of a class is a certain operation (routine or attribute) applicable to instances of the class. The feature is normally passed on to heirs, except for redeclaration, which allows an heir to substitute another feature.
- Every feature of a class has a **final name** relative to that class, called just its "feature name" if there is no ambiguity. This is the name used by the class, its clients and heirs to refer to the feature. The name is normally passed on to heirs, except for renaming, which allows an heir to substitute another name for the same feature.

Redefinition and renaming serve complementary purposes:

Redefinition and renaming

Redefinition changes the feature, but keeps its name.
Renaming keeps the feature, but changes its name.

You may want to apply both mechanisms to a given feature, to change both the feature and its name:

```
class B inherit
        A
                rename
                        f as new_f
                redefine
                        new_f
                end
feature
                … Rest of class omitted …
```

Remember that once you have renamed a feature the only name that makes sense for it in the rest of the class, past the Rename clause, is the new name, which becomes its final name in *C*, here *new_name*. In particular, the Redefine subclause — as well as Undefine — only refers to the new name. So in this example it would have been invalid to write

```
redefine f
```

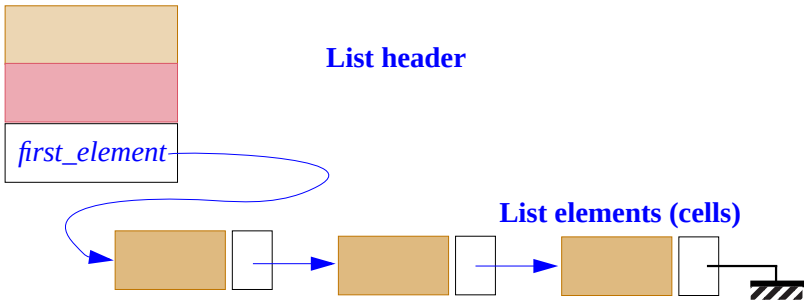
since *f* is not the name of a feature that *C* inherits from *B* (unless the Rename subclause separately renames another inherited feature to *f*).

10.9 CHANGING THE SIGNATURE

The preceding example redefinitions affected the implementation, for either correctness or efficiency reasons. Here now is an example where we need to change the signature of an inherited feature. That feature is an attribute, so its signature only includes the attribute's type.

Consider class *LINKED_LIST* [*T*] in EiffelBase, representing one-way linked lists of objects of type *T* (the formal generic parameter).

One-way linked list



One of the attributes of class *LINKED_LIST* is a reference to the first element of a list:

The actual class text uses LINKABLE [**like** *first*]. *This doesn't affect the discussion.*

```
first_element: LINKABLE [T]
```

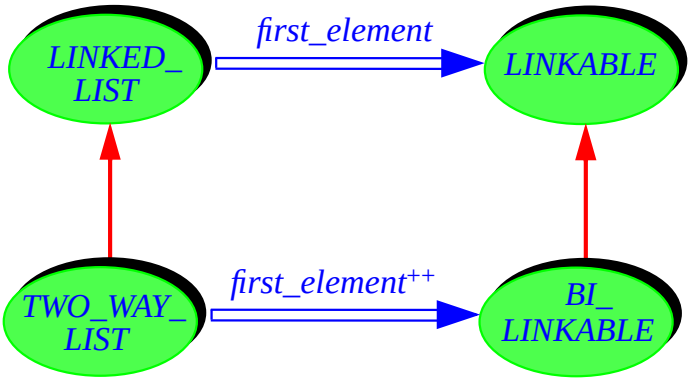
The type of the corresponding objects, *LINKABLE* [*T*], represents list cells, chained to their right neighbors:

Linkable list cell

Various proper descendants of *LINKED_LIST* support variants of the linked list data structure. An immediate heir is *TWO_WAY_LIST*, which, instead of linkables, uses "bi-linkables", chained not just to their successors but also to their predecessors:

Bi-linkable list cell

Class *BI_LINKABLE* is itself an heir from *LINKABLE*:



Signature redefinition

Clearly, the *first_element* of a *TWO_WAY_LIST* should not just be a linkable any more, but a bi-linkable. Hence the need to redefine that attribute, which will appear in *TWO_WAY_LIST* as

```
first_element: BI_LINKABLE [T]
```

The redefinition of *first_element* into a *BI_LINKABLE* in *TWO_WAY_LIST* follows the rule (given in detail below) requiring that any change of type in a redeclaration replace the original with a type that conforms to it (by being based on a descendant class, as *BI_LINKABLE* for *LINKABLE*).

→ *"REDECLARATION AND TYPING", 10.16, page 274*

In this example, the redefined feature is just an attribute. There is often a concomitant need to change the types of routine arguments. For example, the insertion routine *put_element* may be declared in *LINKED_LIST* as

```
put_element (lt: LINKABLE[T] ; i: INTEGER) is…
```

put_element is secret since clients of the list classes never explicitly manipulate linkables, only objects of type T.

Clearly, class *TWO_WAY_LIST* needs to adapt *put_element* to give it a first argument *lt* of type *BI_LINKABLE* [*T*]. A redefinition of *put_element* will achieve this.

10.10 THE NEED FOR ANCHORED DECLARATIONS

Cases such as the redeclaration of the argument *lt* of *put_element* are so frequent in inheritance hierarchies that they warrant a special mechanism, bypassing the need for explicit redefinition. Rather than the above, the signature of *put_element* as declared in *LINKED_LIST* is

```
put_element (lt: like first_element; i: INTEGER) is…
```

meaning that *lt* has the same type as *first_element*: type *LINKABLE* [*T*] in *LINKED_LIST* and, in any proper descendant of this class, the new type, if any, to which *first_element* has been redefined. This mechanism, known as **anchored declaration**, is discussed in detail in a subsequent chapter. It is a form of implicit signature redefinition.

→ "ANCHORED TYPES", 11.10, page 331

10.11 DEFERRED FEATURES

Feature redefinition, as just studied, lets you override the implementation, signature or specification of a feature that already had an implementation in a proper ancestor.

In some cases, the designer of that ancestor could not provide such a default implementation, or did not want to. It is possible to declare a feature without choosing an implementation by making it **deferred**. This transfers to proper descendants the responsibility for providing an implementation through a new declaration, called an **effecting** of the feature.

Although similar in many ways to redefinition, this case is more a "definition" (without the *re*) of the feature, since there was no original implementation in the parent. Accordingly, a class that effects a feature inherited as deferred will not list it in a Redefine clause.

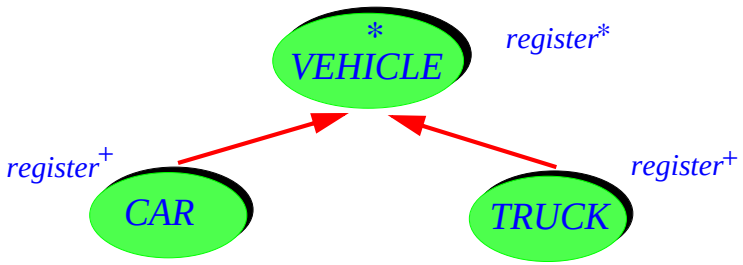
Some terminology:

- A feature that is not deferred — meaning it has an implementation, either as an attribute or as a non-deferred routine — is **effective**.
- The terms "deferred" and effective", originally defined for features, extend to classes: a class is deferred if it has at least one deferred feature; otherwise (if all its features are effective) the class is effective.

Although sufficient for the time being, these definitions will be made more rigorous below.

→ "Effective, deferred feature", page 303

In graphical representations of system structures, both deferred features and deferred classes will be marked by an asterisk *. Their effectings, as other forms of redeclaration, are marked with a plus sign +.



Deferred class, deferred feature, and effectings

As noted above, a class designer may decide to declare a feature as deferred because of either **inability** or **refusal** to provide an implementation. These two cases correspond to the two major uses of deferred routines and classes:

1 • You may want to write a class describing an abstract notion, covering several possible implementations. Then you cannot write an effective class, which would require that you provide full implementation information. Some of the features of such a class, and hence the class itself, will be deferred.

2 • In other cases, whether or not you have enough information to give the implementation, you prefer to concentrate on the abstract properties of a class and its features, postponing implementation concerns to later.

The next two sections explore these two applications.

10.12 DEFERRED CLASSES FOR DESCRIBING ABSTRACTIONS

The first application of deferred classes supports a central aspect of the Eiffel method, resulting from the use of inheritance as a classification mechanism. Often, classes appearing towards the top of inheritance hierarchies represent general categories, for which various proper descendants will provide specific implementations. The higher-level classes should usually be deferred.

The EiffelBase Library contains numerous such cases. A typical example is class *TREE*, describing the most general notion of tree, independent of any representation. Specific implementations are described by proper descendants of that class, such as *FIXED_TREE* and *TWO_WAY_TREE*, both sketched earlier. Class *TREE* contains a number of deferred features describing operations that cannot be made more precise without committing to a representation. Typical of these is the procedure

On FIXED_TREE see 6.2, page 167; on TWO_WAY_TREE, 10.4, page 257.

```
child_put (v: like item)is
        -- Put item v at active child position.
    require
        not_child_off: not child_off
    deferred
    ensure
        replaced: child.item = v
    end
```

which replaces by *v* the value stored in the “active child” (the child at current cursor position) of the current node.

The keyword **deferred**, indicating that the routine is deferred, comes in lieu of an Effective body introduced by **do**, **once** or **external**. As the example shows, the Precondition and Postcondition clauses may still be present; they characterize the semantics of the routine, which all descendant implementations must preserve (in a manner explained below).

Here are two further examples from other ISE Libraries.

See "Reusable Software" for details about these examples.

EiffelVision contains numerous classes representing various geometrical figures, some simple, some composite. They are all descendants of a deferred class *FIGURE*, usually through one of its heirs *OPEN_FIGURE* and *CLOSED_FIGURE*, still deferred themselves.

EiffelParse provides tools for analyzing programs or other structured texts. To build a parser for a particular language, you write classes describing the abstract structure of that language's constructs; for example, a parser for Eiffel will contain classes *EIFFEL_CLASS, ROUTINE, INSTRUCTION* etc. All such classes are descendants of the deferred class *CONSTRUCT*, through one of three heirs of *CONSTRUCT* describing three kinds of construct (the same as in the Eiffel syntax descriptions of this book):

← *"PRODUCTIONS", 2.5, page 88.*

- *AGGREGATE* describes constructs with a fixed number of parts. For example, in a parser for Eiffel, a class describing the syntax of a Loop (where the parts are an Initialization, an Invariant, a Variant and a Loop_body) would be written as an heir to *AGGREGATE*.

→ *The syntax for* Loop *is on page 487.*

- *CHOICE* describes constructs whose specimens are chosen from a number of possible constructs. For example an Eiffel Instruction is a Creation, or a Call, or an Assignment etc.; the corresponding class in a parser would be an heir of *CHOICE*.

← *The syntax for* Instruction *is on page 224.*

- *SEQUENCE* describes constructs with a variable number of components of the same kind, such as an Eiffel Feature_declaration_list, which may consist of zero or more specimens of Feature_declaration.

← *The syntax for* Feature_declaration *is on page 137.*

CONSTRUCT is almost fully deferred. The three heirs listed, although still deferred, are "less" deferred since they provide effective routines for parsing the corresponding types of constructs.

As you will remember, it is not possible to have a feature both deferred and frozen, since frozen features may never be redeclared, and deferred features are born for the very purpose of redeclaration.

← *"Feature Declaration rule", page 160*

10.13 DEFERRED CLASSES FOR SYSTEM DESIGN AND ANALYSIS

In the preceding examples, deferred classes were abstracted from effective ones, by removing implementation aspects. In other cases, deferred classes initially exist independently of any implementation. This is the second of the two major applications of deferred classes.

This situation — mentioned earlier as a case of the designer not *wanting* to consider any implementation — arises in particular out of the use of Eiffel as a tool for **system analysis and design**.

At *design* time, you are concerned with the architecture of a system, not its implementation; deferred classes provide an ideal way to express the abstract properties of an architecture, including contracts, without making decisions about representation or algorithms

At a stage even more remote from implementation concerns, deferred classes are an *analysis* tool: to model and analyze a certain category of real world objects, you may write fully deferred classes that capture the abstract properties of those objects. Not only are such classes independent of any implementation; they may in fact be independent of any computerization. It is indeed possible through deferred classes to describe in Eiffel many natural or artificial systems, whether or not they involve computers and software, as long as their structure and semantics are well understood.

"Fully deferred class" means that all the class's features are deferred. In general, a class is deferred as soon as it has one deferred feature, even if some of its other features are effective.

Object-oriented systems analysis may be defined as the discipline of describing systems of any kind through collections of fully deferred classes, connected by client and heir relations (capturing system structure) and characterized by preconditions, postconditions and invariants (capturing system semantics). Although a detailed presentation of these topics falls beyond the goal of this book, the following class sketch should enable you to form a general idea of O-O system analysis.

Extracted from the hypothetical description of a chemical plant, it illustrates the gist of the method, in particular its use of contracts to characterize the known abstract properties of a set of objects. As noted, such a specification is independent from any computer implementation — although it will of course serve as an ideal basis for the software design and implementation process if computerization does occur.

```
deferred class TANK feature
    fill
            -- Fill tank with liquid
        require
            in_valve.open
            out_valve.closed
        deferred
        ensure
            in_valve.closed
            out_valve.closed
            is_full
        end
```

```
        … Other deferred features, such as:
        empty, is_full, is_empty, in_valve, out_valve
        gauge, maximum, …
invariant
        is_full = ((0.97 * maximum <= gauge) and
                                (gauge <= 1.03 * maximum))
        … Other invariant clauses …
end
```

10.14 EFFECTING A DEFERRED FEATURE

Unless you are using Eiffel just as a modeling language, and do not plan to build software for the system that you first described using deferred classes, you will eventually give these classes proper descendants that **effect** (redeclare as effective) the features they inherit in deferred form.

Any class *C* that inherits a deferred feature from one of its parents may provide a declaration making the feature effective in *C*. (This is a possibility, not an obligation; if the designer of *C* elects to leave some or all of the inherited features deferred, *C* itself will still be a deferred class.)

Effecting a feature is similar to redefining an inherited feature. Here you will not list the feature in a Redefine clause since it was not "defined" in the first place.

→ *The "Redeclaration rule", page 307, states what exactly must appear in the* Redefine *clause.*

As an example of effecting, one of the many proper descendants of *TREE* that effect *child_put* above is *TWO_WAY_TREE*, where the redeclaration, describing the routine's implementation for this particular representation, looks like this:

← *The deferred version of child_put was on page 267.*

```
child_put (v: like item)
        -- Make v the value of the node at active child position;
        -- if current node is leaf, create active child with value v.
    require else
        is_leaf_or_not_off: (not is_leaf) implies (not child_off)
    local
        node: like parent
    do
        if is_leaf then
            create node.make (v)
            put_child (node)
            child_start
        else
            child.put (v)
        end
    ensure then
        set: child_item = v
    end
```

Note the new form of the precondition and postcondition clauses. The precondition of the effective version is the boolean "or" of the original (deferred) routine's precondition and of the assertion given in the **require else** clause; the new postcondition is the boolean "and" of the original postcondition and of the assertion given in the **ensure then** clause. This is part of the general Redeclaration rule, as given below.

→ *"REDECLARATION AND ASSERTIONS", 10.17, page 277; "Redeclaration rule", page 307.*

For an effecting, as with the redeclaration of *put_child* here, you will not list the feature in a Redefine clause.

10.15 PARTIALLY DEFERRED CLASSES AND PROGRAMMED ITERATION

As defined above, a class is deferred as soon as it has at least one deferred feature. But nothing requires it to be **all** deferred: it may contain a combination of deferred and effective features.

This yields one of the most powerful techniques of Eiffel development: producing partially deferred classes which capture what you know for sure about the behaviors and data structures characterizing a certain application area, while leaving open what you do not yet know and what is open to individual variation. You will describe the known aspects through effective features, the variable ones through deferred routines. In particular, an effective routine, covering a known general behavior, may call one or more deferred features, which stand for the variable components of that behavior.

A typical application of this technique appears in many user-interface building systems, where the application software is under the control of an outside loop, sometimes called an **event loop**, which controls the overall scheduling of individual operations: detecting input events, processing these events, updating the screen etc. The event loop is the same for all applications, but each application will define its own version of the individual operations. To implement this scheme elegantly, you may write a deferred class covering the properties of all applications of a certain type, with an effective routine that serves as event loop and calls deferred routines representing the individual operations. Each specific application will then effect these routines, according to its own needs, in a proper descendant of the deferred class.

This scheme is an attractive alternative to the “call-back” mechanisms present in lower-level programming languages.

→ *On call-back mechanisms see also 31.8, page 823, indicating how to enable an existing call-back mechanism, implemented in another language, to call Eiffel routines.*

Another important application of the same idea is illustrated by the **iteration** classes of EiffelBase. These classes provide various iteration mechanisms on arbitrary structures: linear iteration (forward only), two-way iteration, tree iteration (preorder, in order, postorder). For example, class *LINEAR* [*G*] has iteration procedures such as

```
until_do (action: PROCEDURE [ANY, G];
          test: PREDICATE [ANY, G])
          -- Starting at beginning of structure, apply action to
          -- every item up to but excluding first satisfying test.
     do
          from
               start
          until
               after or else test.item ([item])
          loop
               action.call ([item])
               forth
          end
     ensure
          found_if_not_after: not after implies test.item ([item])
     end
```

The actual implementation in the library class is slightly different as it takes advantage of other iteration procedures.

In this procedure, *action* and *test* are **agents**: objects representing operations to be applied. They both take an argument of type *G*, representing a list item; *action* is a procedure that processes such an item, *test* a boolean-valued function (predicate) that determines whether a certain property is true of the item. A typical call, using *your_integer_list* of type *LIST* [*INTEGER*] — where EiffelBase's *LIST* is indeed a descendant of *LINEAR* — is:

→ *Chapter 27 discusses in detail the notion of agent and its application to iteration.*

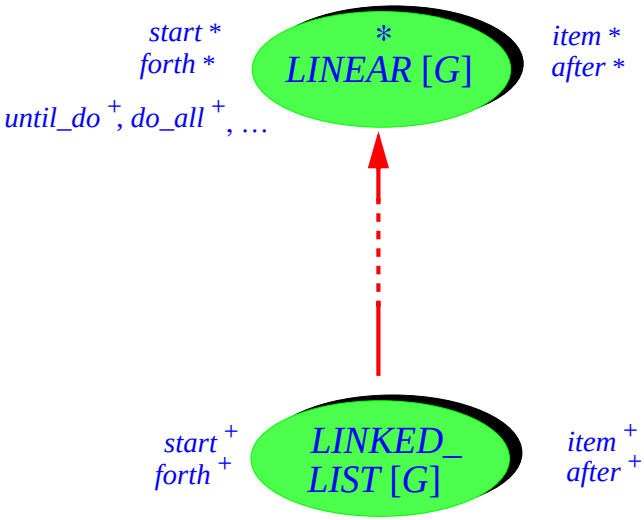
```
your_list.until_do (~ {INTEGER}.print, ~ is_positive)
```

using two agent arguments, one built from procedure *print* as applicable to class *INTEGER* and the other from a function *is_positive* assumed to be available in the current class to determine whether an integer is positive. This call will print the initial elements of the list, if any, up to and excluding the first positive one.

Along with *until_do*, traversal classes such as *LINEAR* and their descendants provide other iterators: *do_until*, *do_all*, *while_do*, *do_while*, *do_if*, *exists*, *for_all*.

LINEAR is a very general deferred class, requiring its effective descendants to provide features representing basic traversing steps: *start* to start traversal, *forth* to advance by one position, *item* to yield the item at cursor position, *after* to find out if the traversal has passed the last item. All the classes of EiffelBase and other libraries that describe traversable data structures such as chains, lists and many others are its descendants.

Effecting the details of iteration



Effective procedures such as *do_until* define traversal patterns. Deferred features such as *start* and *item* describe the ingredients to be used in any particular application of these patterns.

To provide an actual iteration mechanism over a certain concrete structure — such as *LINKED_LIST* or *CIRCULAR_LIST* — it suffices to inherit from *LINEAR* or another of the traveral classes, and to effect the deferred features to describe the specific machinery of iteration processing on the chosen structure: how to start an iteration, move on to the next element, access the current element, and determine end of traversal, based on the specific implementation retained.

10.16 REDECLARATION AND TYPING

The two redeclaration mechanisms studied so far in this chapter, redefinition and effecting, share many properties; both are ways to refine the original declaration of an inherited feature, and both are subject to the same constraints.

Two important properties apply in both cases:

- The type constraint, which we will now explore informally.
- The rule on semantics of updated assertions, studied in the next section.

The formal version of these combined properties is the Redeclaration rule, given in full later.

→ *"Redeclaration rule", page 307.*

First, the type constraints. Let *f* be a precursor (parent's version) of an inherited feature. Assume that the signature of *f* (in the parent) is

```
[A, B], [C]
```

Recall that the first part, here [*A*, *B*], lists the arguments types for a routine (it is empty for an attribute), and that the second part, here *C*, lists the result type for an attribute or a function (it is empty for a procedure).

← *"THE SIGNATURE OF A FEATURE", 5.13, page 148.*

Then the Redeclaration rule will state that if you redeclare *f* into a new feature, the new signature must conform to the precursor's signature.

Conformance, a key concept of the type system, is discussed in detail in a later chapter, but the basic idea is straightforward: a type conforms to another if its base class is a descendant of the other's; a signature conforms to another if it has the same number of arguments and results and every type in the first signature conforms to its counterpart in the other. For example, the signature.

→ *See chapter 14 on conformance, particularly "EXPRESSION AND SIGNATURE CONFORMANCE", 14.4, page 378.*

```
[X , Y], [Z]
```

will conform to the above if type *X* conforms to *A*, *Y* to *B* and *Z* to *C*.

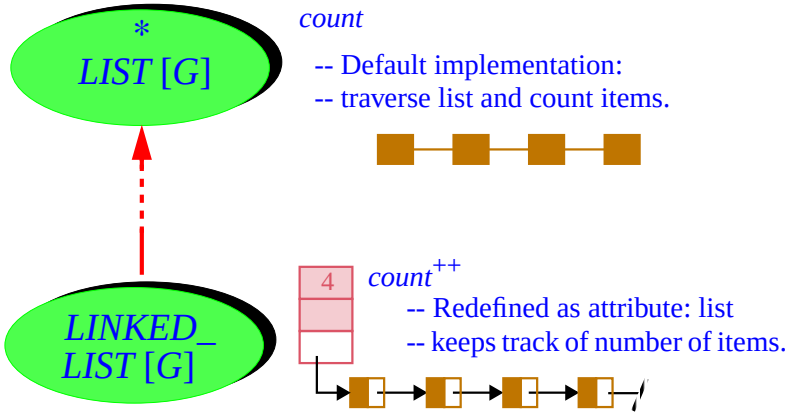
This rule means in particular that a redeclaration may not change the number of arguments and results, and may only replace types of arguments or results by conformant types. You can obtain the effect of changing the number of arguments and results by using tuples.

→ *Chapter 13.*

The Redeclaration rule also prohibits the redeclaration of an attribute into a function. It is permitted, however, to redeclare a function into an attribute; in this case the preceding constraint implies that the precursor function must have been without arguments (otherwise the new signature could not conform). The attribute used for the redeclaration may be variable or constant.

Redeclaring a function into an attribute is a useful and common practice. Here is a typical case. Feature *count*, present in most classes of EiffelBase, gives the number of elements of a structure. Classes high in the inheritance graph, such as *LIST*, the deferred class describing lists independently of any representation choice, declare *count* as a function, which traverses a structure to count its elements. The implementation of effective descendants such as *LINKED_LIST* keeps a record of a list's element count in the list header; these descendants accordingly redefine *count* into an attribute.

Redefining a function into an attribute



This is typical of why you may want to redefine a function into an attribute. A class *B* (*LIST* in this example) has a function *f* that computes some information about the corresponding objects (in the example, the number of items in a list). You devise a new implementation, represented by a descendant *C* of *B*, that keeps the information up to date in a field of the object, represented by an attribute of *C*. (In the example, *C* is *LINKED_LIST*, which keeps a record of the number of items in the list header.) In most object-oriented languages, you would have to define this attribute as a new feature of the class, and redefine *f* into a function that returns its value. But there is no need for two separate features, since they represent the same information: in Eiffel, *C* will simply redefine *f* into an attribute.

This is all in line with the Uniform Reference principle, which states that attributes and functions without arguments should be indistinguishable from the outside, as they are just two alternative ways to provide a query, differing in implementation technique, not relevance to clients.

In implementing such a scheme, *C* must ensure that the value of the query will always be up to date when clients access it; this means that any procedure whose execution may have an effect on the query's value must be redefined in *C* to update the attribute. (In our example, *LINKED_LIST* must redefine all the procedures that insert or remove items, to make sure they increment or decrement *count*.) To make sure that you don't forget any

such redefinition, take a look at procedure postconditions: in well-written classes, the postcondition of any procedure should indicate whether the procedure has any effect on any particular query. For example the postcondition of *remove*, which deletes an item from a list, will have a clause of the form *count* = **old** *count* – *1*. This signals that together with any redefinition of *count* into an attribute there must be a redefinition of *remove* to include the instruction *count* := *count* + *1* or equivalent.

Sometimes the *B* version of *f* is deferred; this is the case in the above example if instead of *LIST* we consider its ancestor *SEQUENTIAL*, where *count* is deferred. (Deferred features are syntactically treated as routines, although if they have no arguments they are just features for which we have refused to choose yet between attribute and routine implementations.)

Why then (in spite of the Uniform Reference principle) does the type constraint prohibit the reverse form of redefinition – changing an attribute into a function? One of the reasons is that we would be unable, were this permitted, to make sense of certain routines inherited from parents. Assume class *B* with features

a: *INTEGER*; *set_a* **is do** *a* := *0* **end**	-- *a* is an attribute -- *set_a* assigns to *a*

Then if *C*, an heir of *B*, were allowed to redefine *a* into a function, but did not redefine *set_a*, there would be no way to execute *set_a* applied to instances of *C*, since one may not assign to a function. For the same reason, it is not permitted to redefine a variable attribute into a constant attribute.

10.17 REDECLARATION AND ASSERTIONS

The other fundamental property of redeclaration governs the Precondition and Postcondition clauses of a redeclared routine. Such assertions, if present, may not be of the basic forms using just **require** and **ensure**; instead they must use **require else** and **ensure then**. Consider a routine redeclaration. If it contains new assertion clauses, they must be of the form

← *See chapter 9 about* Precondition *and* Post-condition *clauses and their semantics in the absence of redeclaration.*

```
require else alternative_precondition
ensure then extra_postcondition
```

expressing the new assertions as a variation on the precursors' assertions.

What kind of variation? Consider a routine redeclaration and let pre_1, … pre_n be the precursors' preconditions and $post_1$, $post_n$ be the precursors' postconditions. (Remember that in most practical cases there is only one precursor, so that n is 1; only with a join of deferred features may there be two or more precursors.) Assume that new assertion clauses are present, of the above form. Then the redeclared routine will be considered to have the precondition and postcondition.

With sharing in repeated inheritance, there may also be two or more precursors, but this is not a case of redeclaration. See the definition of "inherited features" on page 462.

alternative_precondition **or else** pre_1 **or else** … **or else** pre_n
extra_postcondition **and then** $post_1$ **and then** … **and then** $post_n$

In other words, the precondition is or-ed with the original preconditions, and the postcondition is and-ed with the original postconditions. For the precondition, the use of operator **or else** rather than plain **or** guarantees that the assertion is defined, with value true, whenever one of the operands has value true, even if a subsequent one is not defined; similarly, **and then** for postconditions guarantees that any false operand makes the whole assertion false even if a subsequent one is not defined.

→ **or else** *and* **and then** *are the "semi-strict" versions of plain* **or** *and* **and**. *See "SEMISTRICT BOOLEAN OPERATORS", 28.6, page 765.*

If the assertion clauses are missing in a redeclaration, the convention is that the redeclared routine is considered to have *False* as *alternative_precondition* for an absent Precondition part and *True* as *extra_postcondition* for an absent Postcondition. Because of the rules of boolean algebra, this means keeping the corresponding precursor assertions. (Or-ing a boolean value with **false**, or and-ing it with **true**, does not change the condition.)

The use of **require else** and **ensure then** in a redeclared routine reflects an important part of the Design by Contract method underlying Eiffel. Redeclaring a routine means subcontracting to a descendant the job which clients originally entrusted to the precursor. A good subcontractor will do as well as better for clients as agreed in the original contract (involving the precursor). This means:

See "Object-Oriented Software Construction" and "Design by Contract" (references in appendix H) and the notion of subspecification in , page 232.

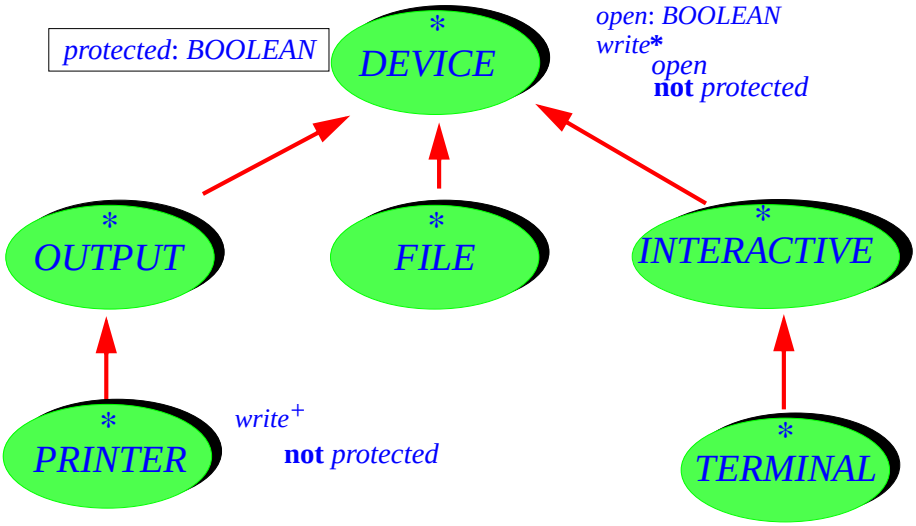
- Keeping or weakening the precondition, so as not to impose any new requirements on the original clients.
- Keeping or strengthening the postcondition, so as to return a result that is as good as what was originally promised to the clients.

The or-ing and and-ing automatically guarantee these rules, since p **or else** q is always weaker than or equal to p, and p **and then** q is always stronger than or equal to p.

A condition is stronger than or equal to another if it implies it, in the sense of boolean implication. "Weaker than or equal" is the inverse relation.

Examples of strengthening the postcondition routine are very common. In fact, almost any redefinition of a routine's implementation, or effecting of a deferred routine, will do something more — such as updating new attributes —, leading to a postcondition stronger than the original. The added properties should appear in the **ensure then** clause.

As an example of weakening the precondition, assume the inheritance hierarchy illustrated below. Procedure *write*, in *DEVICE*, has two clauses in its Precondition: the device must be open, and it must not be protected. Examples of devices are output devices, interactive devices and files.



Precondition weakening

Assume that printers, a kind of device, may not be write-protected. (The invariant of class *PRINTER* should include the clause **not** *protected*.) The precondition of *write* for *PRINTER* may then be weakened to just *open*.

To achieve this, just include in the redefined version of *write* in *PRINTER* the Precondition

require else *open*

The above semantic rule gives, as actual precondition:

open **or else** (**not** *protected* **and then** *open*)

which has the same value as just *open*.

SEMANTICS

If a declaration introduces an immediate feature — in other words, it's not a redeclaration — the **require else** and **ensure then** forms are still permitted, having the same effect as just **require** and **ensure**.

← *"SYNONYMS AND MULTIPLE DECLARATION", 5.18, page 158*

Since the longer forms are normally intended for redeclarations, you might expect a validity constraint which makes them invalid for an immediate feature. But there is no such constraint, among other reasons because this tolerance makes it easy to declare *synonym features* of which one is immediate and the other inherited. A declaration may be of the form

```
inherited, immediate
        require else
                pre
        do
                …
        ensure then
                post
        end
```

where *inherited* is a feature inherited from a parent, for which this declaration will be a redefinition or effecting, but *immediate* is a new feature. The **require else** and **ensure then** form are compulsory because of *inherited*. But they also work for *immediate*, being understood as **require** and **ensure**.

Remember that there is no tolerance in the reverse direction: for a redeclaration, only the **require else** and **ensure then** forms are permitted.

=---- UPDATE --- Assertion declaration, as we have now studied it, complements another property involving the combination of assertions and inheritance: the definition of "invariant of a class" as containing not only the local Invariant clause, but also any others inherited from parents. Together with the rules just seen on assertions of redeclared routines, this ensures that inheritance and redeclaration maintain the fundamental semantic properties of a class and its features, as expressed by the assertions.

We need to consider one more case in the combination of redeclaration and assertions. What happens, when you redefine a function without arguments into an attribute, to the function's assertions if any? Since an attribute has no precondition, we may consider that the precondition is changed to *True*; this is consistent with the preceding discussion since *True* is weaker than any other assertion. For a postcondition, the situation is different: the only way to express that the attribute's possible values will

satisfy the corresponding condition (with the attribute's name substituted for *Result*) is to make it part of the invariant of the class. The definition of class invariants took care of this by stating that the redefinition of a function into an attribute automatically adds the adapted postcondition to the invariant of the redefining class, replacing any occurrence of *Current* by the attribute name. So if a function was of the form

```
last_value: INTEGER
        do
                …
        ensure
                Result >= 0
        end
```

and a descendant *C* of its class of origin redefines *last_value* into an attribute, the invariant of *C* will automatically include the clause

```
last_value = 0
```

--- ADD DISCUSSION OF EFFECT OF REDECLARATION ON "ONLY" POSTCONDITION CLAUSES

10.18 RULES ON INHERITED ASSERTIONS

Unfolded form of an assertion

The **unfolded form** of an assertion *a* of local unfolded form *ua* in a class *C* is the following Boolean_expression:

1 • If *a* is the invariant of *C* and *C* has *n* parents for some $n \geq 1$: up_1 **and** … **and** up_n **and then** *ua*, where up_1, … up_n are (recursively) the unfolded forms of the invariants of these parents, after application of any feature renaming specified by *C*'s corresponding Parent clauses.

2 • If *a* is the precondition of a redeclared feature *f*: the combined precondition for *a*.

3 • If *a* is the postcondition of a redeclared feature *f*: the combined postcondition for *a*.

4 • In all other cases: *ua*.

The unfolded form of an assertion is the form that will define its semantics. It takes into account not only the assertion as written in the class, but also any applicable property inherited from the parent. The "local unfolded form" is the expression deduced from the assertion in the class itself; for an invariant we "and then" it with the "and" of the parents, and for preconditions and postconditions we use "combined forms", defined next, to integrate the effect of **require else** and **ensure then** clauses, to ensure that things will still work as expected in the context of polymorphism and dynamic binding.

The earlier definitions enable us to talk about the "precondition of" and "postcondition "of" a feature and the "invariant of" even in the absence of explicit clauses, by using **True** in such cases. This explains in particular why case 1 can mention "the invariants of" the parents of *C*.

Assertion extensions

The Assertion of a Precondition starting with **require else** is a **precondition extension**.

The Assertion of a Postcondition starting with **ensure then** is a **postcondition extension**.

These are the forms that routines can use to override inherited specifications while remaining compatible with the original contracts for polymorphism and dynamic binding. **require else** makes it possible to weaken a precondition, **ensure then** to strengthen a postcondition, under the exact interpretation explained next.

Covariance-aware form of an assertion extension

The **covariance-aware form** of an assertion extension *a* is:

1 • If the enclosing routine has one or more arguments x_1, … x_n redefined covariantly to types U_1, … U_n: the assertion

($\{x_1: U_1\}\ y_1$ **and** … **and** $\{x_n: U_n\}\ y_n$) **implies** *a'*

where y_1, … y_n are fresh names and *a'* is the result of substituting y_i for each corresponding x_i in *a*.

2 • Otherwise: *a*.

A covariant redefinition may make some of the new clauses inapplicable to actual arguments of the old type (leading to "catcalls"). The covariance-aware form avoids this by ignoring the clauses that are not applicable. The rule on covariant redefinition avoid any bad consequences.

Combined precondition, postcondition

Consider a feature *f* redeclared in a class *C*. Let f_1, … f_n ($n \geq 1$) be its versions in parents, pre_1, … pre_n the preconditions of these versions, and $post_1$, … $post_n$ their postconditions.

Let *pre'* be the covariance-aware form of the precondition extension of *f* if any, otherwise **False**, and *post '* the covariance-aware form of the postcondition extension of *f* if any, otherwise **True**.

The **combined precondition** of *f* is the Assertion

(pre_1 **or**… **or** pre_n) **or else** *pre'*

The **combined postcondition** of *f* is the Assertion

(**old** pre_1 **implies** $post_1$)
and … **and**
(**old** pre_n **implies** $post_n$)
and then *post'*

The informal rule is "perform an *or* of the preconditions and an *and* of the postconditions". This indeed the definition for "combined precondition". For "combined postconditions" the informal rule is sufficient in most cases, but occasionally it may be too strong because it requires the old postconditions even in cases that do *not* satisfy the old preconditions, and hence only need the new postcondition. The combined postcondition as defined reflects this property.

10.19 UNDEFINING A FEATURE

You may redefine an inherited feature; you may also, if it was effective, *undefine* it.

As the Redeclaration rule will express precisely, you may not use redeclaration to turn an effective feature into a deferred one, discarding its inherited implementation. In other words, redeclaration cannot decrease the "effectiveness level" of a feature: it can take the status of an inherited feature from deferred to deferred (redefinition), effective to effective (redefinition), or deferred to effective (effecting), but never from effective to deferred.

→ *Clause 5 of the Redeclaration rule, page 307.*

In some cases, however, this is desirable; when inheriting a feature, you may wish to give it back its virginity, by pretending you inherited it as deferred, even though its precursor (the parent's version) is in fact effective.

Undefinition serves this goal. To undefine one or more effective features inherited from a parent, just list them in the Undefine subclause of the corresponding Parent part, as in

```
class C inherit
        B
                rename
                        …
                undefine
                        f, g, h
                redefine
                        …
                … Other subclauses of Feature_adaptation …
                end
… Other parents and rest of class…
```

In the optional subclauses of a Feature_adaptation, Undefine comes after Rename and New_exports, and before Redefine.

To remember this order, note that all subclauses except Rename *refer to features by their final names, so* Rename *should come first. Since, as seen next, an undefined feature may then be redefined,* Undefine *must come before* Redefine.

Here *f*, *g*, *h* must be features that are effective in *B*. The effect of the above Undefine subclause is that *C* obtains these features from *B* as if they had been deferred rather than effective in that class; the process does not change the features' signature and specification.

It is possible to apply both undefinition and redefinition to the same inherited feature; this is useful if you want to make an inherited feature deferred and also change its signature or specification, as in

```
class E inherit
        B
                undefine
                        f
                redefine
                        f
        end
feature
        f (x: U) is deferred end
end
```

where the *B* version of *f* had an argument of type *T* rather than *U*, assuming (as required by the Redeclaration rule) that *U* conforms to *T*.

This leads to a precise definition of the inherited status of a feature:

Inherited as effective, inherited as deferred

An inherited feature is **inherited as effective** if it has at least one effective precursor and the corresponding Parent part does not undefine it.

Otherwise the feature is **inherited as deferred**.

10.20 REDEFINITION AND EFFECTING

We can now define precisely the two variants of redeclaration:

Effect, effecting

A class **effects** an inherited feature *f* if an only if it inherits *f* as deferred and contains a declaration for *f*.

Such a declaration is then known as an **effecting** of *f*

Effecting a feature (making it *effective*, hence the terminology) consists of providing an implementation for a feature that was inherited as deferred. No particular clause (such as **redefine**) will appear in the Inheritance part: the new implementation will without ado subsume the deferred form inherited from the parent.

Redefine, redefinition

A class **redefines** an inherited feature *f* if and only if it contains a declaration for *f* that is not an effecting of *f*.

Such a declaration is then known as a **redefinition** of *f*

Redefining a feature consists of providing a new implementation, specification or both. The applicable Parent clause or clauses must specify **redefine** *f* (with *f*'s original name if the new class renames *f*.)

Redefinition must keep the inherited status, deferred or effective, of *f*:

- It cannot turn a deferred feature into an effective one, as this would fall be an effecting.
- It may not turn an effective feature into a deferred one, as there is another mechanism specifically for this purpose, *un*definition. The Redeclaration rule enforces this property.

As defined earlier, the two cases, effecting and redefinition, are together called *redeclaration*.

← *Two feature names are "the same" if they are identical or differ only by letter case. See "Same feature name, same operator, same alias", page 153.*

10.21 THE JOIN MECHANISM

The notion of deferred feature yields a useful technique: *feature join*, allowing a class to merge several inherited features into just one.

The join mechanism supports an important aspect of object-oriented architecture design: the fusion of abstractions. The abstractions that need to be combined will come from different hierarchies of deferred classes.

The EiffelBase library, based on combinations of three such hierarchies, provides typical opportunities for such fusion. The hierarchies correspond to complementary classification criteria for general-purpose "container" data structures:

A container data structure, such as a queue or a hash table, serves to store and retrieve objects. Some of the most important kinds of container data structure are covered by the classes of EiffelBase. See "Reusable Software" for details.

- **Storage**, characterizing the representation properties of a container structure (fixed size, variable size but bounded, unbounded but finite, potentially infinite).
- **Access**, characterizing the methods through which clients store and retrieve elements (in last-in-first-out for stacks, through a key for hash tables etc.).
- **Traversal**, characterizing ways of exploring the container exhaustively (forward, backward, postorder, preorder and others.).

You can obtain a particular type of effective container by multiple inheritance from classes of these three categories. For example, a "fixed-size list" has fixed-size storage, access by index and other techniques, and forward traversal.

In this process of combining abstractions, it will often be useful to merge inherited deferred routines if they correspond to the same notion in the descendant. For example, the deferred EiffelBase class *CHAIN* (describing sequential structures such as lists) inherits from two deferred classes that both have an *item* function returning the item at cursor position:

- *ACTIVE*, from the Access hierarchy, describe structures with a client-controlled "cursor" position. Procedures are available to move the cursor to various elements. In this class, *item* denotes the value of the element at cursor position.
- *BIDIRECTIONAL*, from the Traversal hierarchy, describe structures that are sequentially traversable both forward and backward. In this class, *item* denotes the value of the current element at each step of a traversal operation.

Class *CHAIN* combines these two concepts and inherits both *item* functions. Normally, this would be considered a name clash, which we would have to remove through renaming. But here the clash is harmless, in fact desired, since for a *CHAIN* the two concepts are compatible. If the features were effective, we would have to choose between conflicting implementations; but they are both deferred, so we have no such problem. We can simply merge — "join" — them into one.

→*"NAME CLASHES", 10.23, page 290.*

It is valid, then, to write *CHAIN* as heir to both *BIDIRECTIONAL* and *ACTIVE* even without renaming the deferred *item* routines, which will yield a single deferred routine in *CHAIN*:

```
deferred class CHAIN [T] inherit
        BIDIRECTIONAL [T]
                -- BIDIRECTIONAL has a deferred routine item
            …
        ACTIVE [T]
                -- ACTIVE has a deferred routine item
            …
… Other parents and rest of class text omitted …
```

Here is another interesting application. Occasionally you will need to effect an inherited procedure to do nothing at all. For example a descendant of a general-purpose iteration class, as studied earlier in this chapter, might not need a particular initialization operation, provided in the ancestor by a procedure *prepare*. You can manually effect *prepare* into a procedure that does nothing. But it is simpler to use a join with the procedure *do_nothing* from class *ANY*, whose implementation faithfully respects its name:

```
class SIMPLE_ITERATOR inherit
        GENERAL_ITERATOR
            rename prepare as do_nothing end
… Other parents and rest of class text omitted …
```

That's all you have to do: renaming *prepare* causes a join with *do_nothing* and the associated effecting.

To be joined, inherited features must have the same final name in the class that performs the join. In the above case both precursors were called *item* in the parents, so no particular action was required from the designer of class *CHAIN* with respect to their names.In other cases you might want to join two deferred features that have different names, say *f* and *g*, in the respective parents. You should then use renaming to make sure that the features are inherited under the same final name:

```
-- C may be deferred or not (see below)
… class C inherit
        A
                rename
                        f as new_name
                …
                end
        B
                rename
                        g as new_name
                …
                end
```

If *C* inherits and joins two or more deferred features, the net result for *C* is as if it had inherited a single deferred feature. In the absence of further action from *C*, that feature remains deferred. *C* may of course provide an effective declaration, killing several abstract birds with one concrete stone by using a single redeclaration to effect several features inherited as deferred.

← *"Inherited as deferred" was defined (page 285) to mean: either coming from deferred precursors, or explicitly undefined.*

More generally, *C* may treat the result of the join as it would any other inherited deferred feature. *C* may in particular redefine the feature to change its signature while leaving it deferred. In that case *C* must list all the inherited features in the Redefine subclauses of their respective Parent parts.

The join mechanism imposes easily justifiable conditions on features to be joined in this way: they must be deferred (after possible undefinition, as detailed in the next section), inherited under the same name (after possible renaming), and equipped with the same signature (after possible redeclaration). The formal rule expressing these requirements is the Join rule, described later in this chapter.

→ *"Join rule", page 309.*

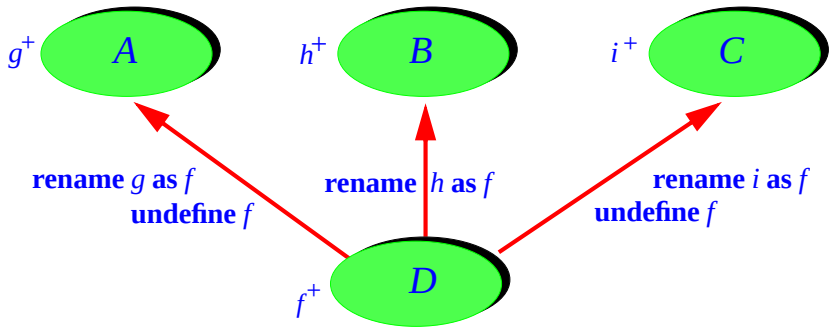
10.22 MERGING EFFECTIVE FEATURES

As introduced so far, the join mechanism applies only to deferred routines. The reason is obvious: an attempt by a class *D* to join two effective features inherited from parents of *D* may yield an ambiguous result in the absence of a clear universal criterion for choosing one of the two inherited implementations over the other.

What happens, however, if when you design *D* you *do* know which of the versions you want to override the other in *D*? Then the merging should not raise any particular problem.

The undefinition mechanism makes this possible. Here is an illustration of the scheme, used in this case to join three features:

← *"UNDEFINING A FEATURE", 10.19, page 283.*



Merging and overriding

We want to merge the three inherited features by renaming all of them into a single name, *f*. But the originals were all effective, yielding three implementations of which we may retain only one in *D*. To discard the *A* and *C* implementations, *D* undefines them, leaving the *B* version as the undisputed victor.

In the simplest case, there are only two competing features in parents *B* and *C*, and they already had the same name *f* in these parents. If you want the *B* version to take over in *D* all you need is to undefine the *C* version:

```
class D inherit
        B
                -- B has an effective feature f
        C
                undefine f end
feature
        …
end
```

Although *f*'s precursor in *B* was effective, the undefinition causes *f* to be "inherited as deferred" from *C*. The *B* version provides an effecting.

An application of this technique will appear in repeated inheritance when a class inherits conflicting versions of the same feature, and the class designer wants to retain only one of these versions.

→ *See the beginning of 16.5, page 434.*

The general rule is the natural one (although we must wait until a full definition of repeated inheritance to express it rigorously): inheriting two or more features under the same name may only be invalid — a case of **name clash** — if more than one is inherited as effective. If, after possible undefinition, they are all deferred, or all deferred except for one effective version, then we have a valid case of join, since there is no conflict of implementations: we have either no implementation or one. In the latter case the effective version will serve as common implementation for all the features inherited as deferred.

→ *The next section discusses name clashes.*

The examples have illustrated one way to reconcile conflicting effective versions from parents: undefine all but one of them. This is like a competition where one of the rivals win. There is another way — as in business or in war — to resolve a competition: a new entrant overcomes everyone else. The technique here will be to use **redefinition** rather than undefinition: redefine all the conflicting inherited versions into a new one. The last example becomes:

```
class D inherit
        B
                redefine f end
        C
                redefine f end
feature
        f
                do
                        … "Redefined algorithm" …
                end
        …
end
```

As you may have noted, it actually doesn't make any difference here if we replace either or even both of the **redefine** keywords by **undefine**. If we undefine one of the features, the other takes over, but gets redefined. If we undefine both, they are inherited as deferred, and hence joined; but then the declaration of *f* effects both.

10.23 NAME CLASHES

Now that we have seen the join mechanism we are in a position to define precisely the notion of **name clash** of features under multiple inheritance, and see what kinds of name clashes are permitted. From the previous section we know the rough form of the rule: a name clash in a class *D* between two or more inherited features will be OK, leading to a join of all of them, if they all have compatible signatures (so that we may indeed join them into a common version) and, taking any undefinitions into account:

- Either the resulting features, except possibly one, are all deferred.
- Or if this is not the case, meaning that two or more versions remain effective in *D*, then *D* redefines all of them into a common version, as in the example class text above.

The discussion of repeated inheritance will also add a permissible case: the "false alarm" resulting from features that come from different parents but are really the *same feature* inherited from a common ancestor. In the absence of conflicting redefinitions this can cause no trouble.

Let's see the precise form of the rule. The general guideline is the **no overloading principle**, dictated (although it may at first sound like an advertisement for a mutual fund) by criteria of clarity and simplicity.

> Overloading — the possibility for a single name to denote several features within the context of a given class — defeats the principles of object technology, running into conflict with the more powerful forms of *dynamic* overloading provided by polymorphism and dynamic binding. Introducing in-class overloading is probably the biggest mistake that one can make in the design of an O-O language.

In the absence of inheritance, the no-overloading principle is easy to enforce: all the features declared in a class must have different names. With single inheritance, we add the rule that no inherited feature may have the same final name as a feature of the class; renaming provides an easy way to correct any such potential conflict. With multiple inheritance, this last rule must still apply between the class and each of its parents, but in addition we have to take into account the case of conflicts between names of features in the parents themselves. This is what we call a name clash:

Name clash

A class has a **name clash** if it inherits two or more features from different parents under the same final name.

Since final names include the identifier part only, aliases if any play no role in this definition.

Name clashes would usually render the class invalid. Only three cases may — as detailed by the validity rules — make a name clash permissible:

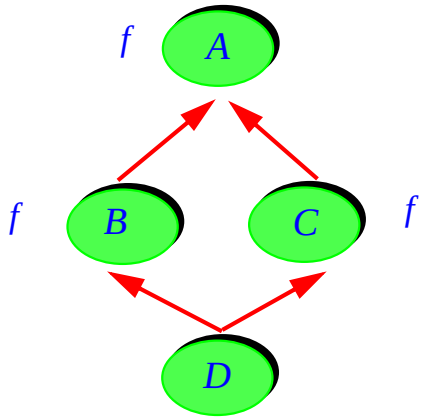
- At most one of the clashing features is effective.
- The class redefines all the clashing features into a common version.
- The clashing features are really the same feature, inherited without redeclaration from a common ancestor.

  This property is not expressed as a separate validity constraint since it follows from the Join rule given at the end of this chapter, and the complementary mechanisms discussed in the repeated inheritance chapter.

In the first permissible case, the clash involves only one implementation, or none; if the signatures are compatible, we may join all the features into a single one, with no particular difficulty (and without departing from the no-overloading principle). The second case is similar: joining through redefinition.

In the third case, we don't have a real clash at all, only the appearance of one, as if being scared in an empty house by a moving figure that turns out to be our own reflection in the mirror. This case arises out of repeated inheritance (as studied in a later chapter) in the situation represented on the figure:

→ *Chapter 16 explores repeated inheritance.*



A name clash that isn't really one.

D seems to inherit two features *f* from both its parents *B* and *C*, but they are not really different features, simply the same feature inherited from a common ancestor *A*, and not redeclared anywhere in the process. As we may expect, and the rules of repeated inheritance will state precisely, *D* inherits a single feature *f*, so this case causes no difficulty.Outside of these three cases, however, a name clash is always prohibited. In the typical situation

```
class C inherit
        A
        B
… Rest of class omitted …
```

where both *A* and *B* have a feature with the same name *fname*, class *C* will be invalid. It's quite easy to get rid of the name clash:

- Often you will want the features to remain distinct in *C*, because they indeed correspond to different operations; their sharing of a common name is just an unfortunate coincidence, a kind of pun. Then you will simply rename one, or both.

- Sometimes, however, it's not just a pun: in *C* you really want the clashing features to be merged into one. Then, if the signatures are compatible, you can rely on the join mechanism by undefining either one; the other's implementation will take over. You may also undefine both, leaving *C*or one of its own proper descendants in charge of effecting the joined result.

10.24 ADDING TO INHERITED BEHAVIOR: PRECURSOR

The last mechanism of this chapter, Precursor, simplifies writing a routine's redefinition when the new implementation relies on the original one.

The need for a precursor mechanism

You may redefine features of all kinds, but this section only applies to routines.

In studying redefinition we have seen that you can override a routine's inherited implementation (as well as its signature and contract). The new implementation may be completely different from the original one; but fairly often it just extends it, performing the same actions as the original plus some others, with a redefinition of the form

```
your_routine
        do
                "Something else"
                "Whatever the original version did"
                "Yet something else"
        end
```

If you need new actions only after the original processing there is no "Something else"; if only before, there is no "Yet something else".

A typical example would be the redefinition, in a proper descendant *TAXABLE_INVESTMENT* of a class *INVESTMENT*, of the procedure *sell*, where the new version performs what the original version did but must also compute the tax penalty associated with the sale of a stock.

The Precursor mechanism provides you, when writing redefinitions of this kind, with a simple way to include "Whatever the original version did" in the actions of the new routine body. Without Precursor, you would have two ways of achieving the intended effect:

- You could simply repeat the original algorithm in the body of the new version, in lieu of the line that reads "Whatever the original version did" above. This works but has the usual effects of code duplication: making the software bigger and less readable (since the reader doesn't immediately realize that a certain element is not original but the verbatim replication of something else); tediousness for the developer; and, most damaging, the need to remember, if the original changes, that you must update all duplicates as well, with the risk of forgetting some.

  As you will have noted, one of the recurring fetures of the Eiffel method is its phobia of unnecessary replication. Genericity, inheritance and other reuse mechanism are all intended to make sure that what needs to be said is said well, and said once.

- You can also use the replication mechanisms of repeated inheritance, studied in the relevant chapter, to keep a duplicate of the original feature, along with the redefined version. This approach avoids the drawbacks of the preceding technique; it was indeed the recommended method in early versions of Eiffel, and remains appropriate in some cases. For most common applications, however, it is overkill, and Precursor provides a simpler solution.

→ *"KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE", 16.8, page 443*

Precursor basics and examples

That solution is in fact disarmingly easy: in a routine redefinition, Precursor stands for what was written above as "Whatever the original version did". The form of the construct is simply the reserved word *Precursor*, followed by a list of actual arguments if any. So you can write the first example sketch above, for a routine with no arguments, as just

```
your_routine
        do
                "Something else"
                Precursor
                "Yet something else"
        end
```

For a routine with arguments, the redefinition might look like

```
sell (share_count: REAL; selling_price: PRICE)
                -- Record sale of share_count shares at selling_price.
        do
                Precursor (share_count, selling_price)
                compute_tax
        end
```

These examples illustrate how to use the *Precursor* reserved word: exactly like you would use the feature name (*new_version* and *sell* in these examples) for a new call to the corresponding routine, such as the calls *new_version* and *sell* (*share_count*, *selling_price*). Appearing in the body of the routine, these calls would be recursive — leading in fact, as written, to infinite recursion —, but instead of the feature name we use *Precursor* which yields the desired effect, calling the original version.

In the *sell* example the arguments to *Precursor* are the same as the formal arguments to the procedure, *share_count* and *selling_price*, meaning that you call the precursor with the same arguments that were passed to you. This is not a general requirement; in other circumstances you may pass to *Precursor* any arguments of the appropriate types.

These two examples use procedures. The mechanism works just as well with functions:

```
profit (share_count: REAL; selling_price: PRICE): AMOUNT
            -- Profit from sale of share_count shares at selling_price.
        do
            Result := Precursor (share_count, selling_price)
                        – tax_penalty (share_count, selling_price)
        end
```

The result of the *Precursor* call is the result returned by a call to the original version of the routine, with the given arguments.

The Precursor contruct is valid only in the case illustrated by these examples: the body of the redefinition of a routine, in which it denotes the original implementation in the parent.

Outside of this case a class may not refer to ancestor versions of its features (as provided by the "super" variables of some object-oriented languages, notably Smalltalk) because this would impair the consistency of the notion of class. A class is entirely defined by its features; how these features were arrived at through inheritance is internal information, and the original versions are not part of the information associated with the class. (They will often violate the contracts associated with the class, in particular by failing to maintain its invariant, typically stronger than the ancestors' invariants.) The only justification for accessing a parent version is that we may need the old implementation to define a new one, through the Precursor construct.

The precursor of a redefined feature is *not* a feature of the current class. If you do want to keep both the original and the redefinition as features of the class, you can, but you have to use a different mechanism: repeated inheritance, as explained in a later chapter.

→ *"KEEPING THE ORIGINAL VERSION OF A REDEFINED FEATURE", 16.8, page 443*

Choosing between multiple precursors

In ordinary cases, as illustrated by the examples, a redefined routine has only one effective precursor. In studying the join of routines, however, we have seen that it is possible for a routine declaration to be the redefinition of two or more parent versions (precursors). If you use a Precursor construct in such a case you will need to specify which precursor you want, by listing its name. Instead of just *Precursor* (*arguments*) the syntax in that case will be *Precursor* {*PARENT*} (*arguments*), where *PARENT* is the name of one of the parent classes from which we are redefining the feature.

The earlier join example illustrates the case of multiple precursors: ← *Page 290.*

```
class D inherit
        B
                redefine f end
        C
                redefine f end
feature
        f
                do
                        … "Redefined algorithm" …
                end
        …
end
```

In the "Redefined algorithm" a precursor call of the form *Precursor* (*arguments*) is invalid, because it leaves open the obvious question "*Which* precursor do you mean: the version from *A*, or from *B*?".

The qualified form removes the ambiguity: you should write either one of

```
Precursor {B} (arguments)
Precursor {C} (arguments)
```

You may, in fact, include both of these in the redefinition's body if you need to reuse both parents' original implementations to define the new one.

The form with explicit qualification, *Precursor* {*PARENT_NAME*}, is valid even in the absence of ambiguity. It is usually preferable to use this form in all cases since it clarifies the context and helps identify errors if you change parents. This is part of the style guidelines.

With these observations we have enough to introduce the formal properties of the Precursor construct. (They will mark the beginning of the formal part of this chapter; since it will introduce no new construct or technique, but only provide precise definitions of the concepts seen informally so far, you may on first reading skip to the next chapter.)

Precursor specification

The syntax of the Precursor construct covers the variants seen in the preceding examples:

← *The definition of* Parent_qualification, *repeated here for clarity, originally appeared with* Clients *on page 204.*

Precursor

Precursor ≜ **Precursor** [Parent_qualification] [Actuals]

Parent_qualification ≜ "**{**" Class_name "**}**"

For the validity and semantics, we avoid introducing special rules — which would repeat many of the properties of calls — by relying on our usual *unfolding* language definition technique: we just pretend that we were clever enough, in the parent class, to keep a duplicate of the original feature, by relying on a synonym feature:

Relative unfolded form of a Precursor

In a class *C*, consider a Precursor specimen *p* appearing in the redefinition of a routine *r* inherited from a parent class *B*. Its **unfolded form relative to** *B* is an Unqualified_call of the form *r'* if *p* has no Actuals, or *r'* (*args*) if *p* has actual arguments *args*, where *r'* is a fictitious feature name added, with a **frozen** mark, as synonym for *r* in *B*.

In other words, we will talk about the Precursor call as if the declaration of *r* in *B*, instead of just

r (a: T; …) … **do** … Body …**end**

had been written with a frozen synonym

r, **frozen** *r'* *(a: T; …)* … **do** … Body … **end**

The rule on multiple declarations implies that this is equivalent to having declared independent features with an identical Body. Because *r'* is frozen, it retains the original semantics of *r*, in the context of the new class *C*; this is exactly what we want to describe the validity and semantics of Precursor.

← *"Unfolded form of a possibly multiple declaration", page 158.*

Here indeed is the validity:

Precursor rule *VDPR*

A Precursor is valid if and only if it satisfies the following conditions:

1 • It appears in the Feature_body of a Feature_declaration of a routine *r*.

2 • If the Parent_qualification part is present, its Class_name is the name of a parent class *P* of *C*.

3 • Among the routines of *C*'s parents, limited to routines of *P* if condition 2 applies, exactly one is an effective routine redefined by *C* into *r*. (The class to which this routine belongs is called the **applicable parent** of the Precursor.)

4 • The unfolded form relative to the applicable parent is, as an Unqualified_call, argument-valid.

In addition:

5 • It is valid as an Instruction if and only if *r* is a procedure, and as an Expression if and only if *r* is a function.

This constraint also serves, in condition 3, as a definition of the "applicable parent": the parent from which we reuse the implementation. Condition 4 relies on this notion.

Condition 1 states that the Precursor construct is only valid in a routine redefinition. In general the language definition treats functions and attributes equally (*Uniform Access* principle), but here an attribute would not be permissible, even with an Attribute body.

Because of our interpretation of a multiple declaration as a set of separate declarations, this means that if Precursor appears in the body of a multiple declaration it applies separately to every feature being redeclared. This is an unlikely case, and this rule makes it unlikely to be valid.

← *"SYNONYMS AND MULTIPLE DECLARATION", 5.18, page 158*.

Condition 2 states that if you include a class name, as in *Precursor* {*B*}, then *B* must be the name of one of the parents of the current class. The following condition makes this qualified form compulsory in case of potential ambiguity, but even in the absence of ambiguity you may use it to state the parent explicitly if you think this improves readability.

Condition 3 specifies when this explicit parent qualification is required. This is whenever an ambiguity could arise because the redefinition applies to more than one effective parent version. The phrasing takes care of all the cases in which this could happen, for example as a result of a join.

Here is a more verbose form of clause 3, obtained from a mathematical specification. Let *PAR* be the set of classes defined as follows: if the Parent_qualification part is present, *PAR* is the single-element set containing the class whose name is listed in that Parent_qualification; otherwise *PAR* is the set of all parents of *C*. Let *REDEF* be the set of all the effective routines, from classes belonging to *PAR*, of which *r* is a redefinition. Then *REDEF* has exactly one element.

Condition 4 simply expresses that we understand the Precursor specimen as a call to a frozen version of the original routine; we must make sure that such a call would be valid, more precisely "argument-valid", the requirement applicable to such an Unqualified_call.

A Precursor will be used as either an Instruction or an Expression, in the same way as a call to (respectively) a procedure or a function; indeed Precursor appears as one of the syntax variants for both of these constructs. So in addition to being valid on its own, it must be valid in the appropriate role. Condition 5 takes care of this.

Pages 224 and 753.

This property really belongs to the validity of instructions and expressions, but having a single clause here saves two full-fledged validity rules in the respective chapters: "It is valid to use a Precursor as an Instruction if and only if its unfolded form is a call to a procedure", and "It is valid to use a Precursor as an Expression if and only if its unfolded form is a call to a function".

The definition of the "relative" unfolded form didn't necessarily yield a valid call; in fact it serves, in clause 4, to determine validity. If as a result we know we have a valid Precursor, we can define an unfolded form that is not relative any more:

Unfolded form of a Precursor

The **unfolded form** (absolute) of a valid Precursor is its unfolded form relative to its applicable parent.

The semantics follows immediately:

Precursor semantics

The effect of a Precursor is the effect of its unfolded form.

As usual, semantics is only defined for valid specimens, so it may legitimately use the "absolute" unfolded form.

10.25 REDEFINITION AND UNDEFINITION RULES

The agenda for the remainder of this chapter is to provide the precise rules for syntax, validity and semantics of the mechanisms seen so far — all feature adaptation mechanisms except for those involving repeated inheritance. As already noted, this will introduce no new techniques, so you may prefer on first reading to skip the rest of this chapter.

Let us begin with the straightforward syntax and validity of Undefine and Redefine subclauses. It will do no harm to repeat here (again) the general structure of Inheritance clauses:

Inheritance parts

Inheritance ≜ "**inherit** Parent_list

Parent_list ≜ "{Parent ";" … }

Parent ≜ "Class_type [Feature_adaptation]

Feature_adaptation ≜ [Rename]
[New_exports]
[Undefine]
[Redefine]
end

This syntax appeared first on page 169.

The clauses involved in the present discussion are Undefine and Redefine. Here is the syntax of Redefine:

← *See page 180 for* Rename *and 205 for* New_exports.

Redefinition

Redefine ≜ **redefine** Feature_list

The following constraint applies to Redefine subclauses:

Redefine Subclause rule *VDRS*

A Redefine subclause appearing in a Parent part for a class *B* in a class *C* is valid if and only if every Feature_name *fname* that it lists (in its Feature_list) satisfies the following conditions:

1 • *fname* is the final name of a feature *f* of *B*.

2 • *f* was not frozen in *B*, and was not a constant attribute.

3 • *fname* appears only once in the Feature_list.

4 • The Features part of *C* contains one Feature_declaration that is a redeclaration but not an effecting of *f*.

5 • If that redeclaration specifies a deferred feature, *C* inherits *f* as deferred.

In this definition:

- The final name of an inherited feature (clause 1) is its name as it results from possible renaming (the Feature_name part only, not including any Alias).

  ← *"FEATURES AND THEIR NAMES", 6.10, page 182.*

- A feature is "frozen" (clause 2) if it has been declared with the keyword **frozen** in its class of origin. The purpose of such a declaration is precisely to forbid any redefinition of the feature in descendants, guaranteeing that the exact original implementation remains in place.

  ← *"FEATURE DECLARATIONS: SYNTAX", 5.10, page 140.*

- A feature is a constant attribute (clause 2) if it is declared with a clause of the form **is** *v*, where *v* is Manifest_constant.

  ← *"HOW TO RECOGNIZE FEATURES", 5.12, page 145.*

- The condition for a redeclaration to be valid (clause 4) appears later in this chapter; in particular, the new signature must conform to the original's, and you may not redeclare an attribute into a function.

  → *"REDECLARATION RULES", 10.28, page 306.*

- If *C* provides an effective version of a feature that it inherits as deferred, this is a case of effecting, and hence of redeclaration, but not of redefinition; as a consequence, clause 4 indicates that the feature must not appear in the Redefine subclause.

  → *Effecting is defined precisely in the next section.*

As to the semantics:

Redefinition semantics

The effect in a class *C* of redefining a feature *f* in a Parent part for *A* is that the version of *f* in *C* is, rather than its version in *A*, the feature described by the applicable declaration in *C*.

This new version will serve for any use of the feature in the class, its clients, its proper descendants (barring further redeclarations), and even ancestors and their clients under dynamic binding.

The syntax of an Undefine clause is similar to that of a Redefine:

Undefine clauses

Undefine ≜ **undefine** Feature_list

The constraint is also similar:

Undefine Subclause rule *VDUS*

An Undefine subclause appearing in a Parent part for a class *B* in a class *C* is valid if and only if every Feature_name *fname* that it lists (in its Feature_list) satisfies the following conditions:

1 • *fname* is the final name of a feature *f* of *B*.

2 • *f* was not frozen in *B*, and was not an attribute.

3 • *f* was effective in *B*.

4 • *fname* appears only once in the Feature_list.

5 • Any redeclaration of *f* in *C* specifies a deferred feature.

--- EXPLAIN LAST CLAUSE ---

and the semantics:

Undefinition semantics

The effect in a class *C* of undefining a feature *f* in an Inheritance part for *A* is to cause *C* to inherit from *A*, rather than the version of *f* in *A*, a deferred form of that version.

→ *This also applies to clients of proper ancestors, under dynamic binding. "DYNAMICBINDING", 23.12, page 630*

10.26 DEFERRED AND EFFECTIVE FEATURES AND CLASSES

The discussion has already referred informally to features being "deferred" or "effective" in a class. We can now make these notions precise, and use the opportunity to define what it means to "effect" a feature

Effective, deferred feature

A feature *f* of a class *C* is an **effective feature** of *C* if and only if it satisfies either of the following conditions:

1 • *C* contains a declaration for *f*, specifying it as either as an attribute or as a routine whose Routine_body is of the Effective form (not the keyword **deferred** but beginning with **do**, **once** or **external**).

2 • *f* is an inherited feature, coming from a parent *B* of *C* where it is (recursively) effective, and *C* does not undefine it.

f is **deferred** if and only if it is not effective.

As a result of this definition, a feature is deferred in *C* not only if it is introduced or redefined in *C* as deferred, but also if its precursor was deferred and *C* does not redeclare it effectively. In the latter case, the feature is "inherited as deferred".

← *"Inherited as effective, inherited as deferred", page 285*.

The definition captures the semantics of deferred features and of their effecting. In case 1 it's clear that the feature is effective, since *C* itself declares it as either an attribute of a non-deferred routine. In case 2 the feature is inherited; it was already effective in the parent, and *C* doesn't change that status.

← *"UNDEFINING A FEATURE", 10.19, page 283*.

In case 1 the declaration may be for a new (*immediate*) feature, or it may be a redeclaration of an inherited feature, deferred in the parent but made effective in *C*. This is known as an *effecting*:

Effecting

A redeclaration into an effective feature of a feature inherited as deferred is said to **effect** that feature.

Some validity constraints, seen below, apply to this case: the effective feature must satisfy the Redeclaration rule, and if there are two or more deferred features among the lot, this is a *join*, governed by the Join rule.

It is possible under this definition for a redeclaration to effecting *several* inherited features. The only other case in which we permit inheriting several features with the same name without renaming is sharing under repeated inheritance. Here too we don't have a real name clash, as long as at most one of the features is effective and they satisfy the two applicable rules (Redeclaration and Join).

→ "Repeated Inheritance Consistency constraint", page 458.

Effecting may follow one three schemes:

1 • You may write *C* as heir to a class *B* where *f* is deferred, and provide an effecting of *f* in the form of a Feature_declaration in the Features part of *C*. This is the most common use of deferred features and effecting.

← As illustrated in "EFFECTING A DEFERRED FEATURE", 10.14, page 270.

2 • You may want to inherit a specification from one parent *A* and the corresponding implementation from another *B*. In this case *A* will provide a deferred feature and *B* an effective feature with compatible signature; if they have the same final name in *C*, the *B* version will serve as effecting of the *A* version. In this case there is no new feature declaration in *C*.

← As illustrated by the figure "Merging and overriding", page 289.

3 • *C* may also undefine a parent's effective feature, and use an effective feature (inherited from a parent, or introduced or redefined in *C* itself) to provide an implementation. This is less common, but provides the mechanism for merging effective features, with one of the implementations overriding the others, as in one of the earlier examples.

← Class D, page 289.

The above defines the meaning of "deferred" and "effective" for features. These qualifiers carry over to the classes that contain these features:

Deferred class property

A class that has at least one deferred feature must have a Class_header starting with the keyword **deferred**. The class is then said to be **deferred**.

This includes a validity requirement and a definition, both of which follow from the the original discussion of classes:

- The requirement to declare the class as **deferred** as soon as it has deferred feature is not a new validity constraint, but just repeats what the Class Header rule said — except that now, as a result of the definitions in this chapter, we have a precise definition of "deferred feature" (introduced as deferred, or inherited as deferred and not effected).

← "Class Header rule", page 126.

- As to the definition, it follows from the Class Header rule combined with the original definition of "deferred class", which stated that a class is deferred if its Class_header starts with **deferred**. That was a purely syntactic criterion; now we have a more meaningful one, reminding us that a class is deferred whenever it has a deferred feature.

← "Deferred class, effective class", page 127.

The reverse — that a class is effective if all its features are effective — is usually true, but not always since you have the option of declaring it explicitly as **deferred**, to specify that it remains abstract and not directly instantiatable. Hence the precise phrasing of the complementary property:

Effective class property

A class whose features, if any, are all effective, is effective unless its Class_header starts with the keyword **deferred**.

As a summary, remember that you **must** declare a class as

deferred class *C* …

This is not necessarily the beginning of the class text itself since there may be a Notes *clause first.*

as soon as it has a deferred feature *f* — not only if *f* is introduced in *C* as deferred, but also if *C* inherits it as deferred and does not effect it.

For an effective class, you will just use one of

class *C*…
expanded class *C*…
reference class *C*…

10.27 ORIGIN AND SEED

Two useful definitions follow from the discussion of redeclaration. Chapter 6 defined the **origin** of a feature introduced in class *C* as *C* itself.

← *This is a refinement of the initial definition of "origin" on page 133, which only covered case 1 of the present definition.*

We can now generalize this to arbitrary features, inherited as well as immediate. The associated notion is a feature's **seed**, its original version. These notions, which will be especially useful in the discussion of repeated inheritance, are defined as follows.

Origin, seed

Every feature *f* of a class *C* has one or more features known as its **seeds** and one or more classes known as its **origins**, as follows:

1 • If *f* is immediate in *C*: *f* itself as seed; *C* as a origin.

2 • If *f* is inherited: (recursively) all the seeds and origins of its precursors.

→ *"SHARING AND REPLICATION", 16.4, page 428.*

The origin, a class, is "where the feature comes from", and the seed is the version of the feature from that origin. In the vast majority of cases this is all there is to know. With repeated inheritance and "join", a feature may result from the merging of two or more features, and hence may have more than one seed and more than one origin. That's what case 2 is about.

----------If this is your first reading, do not let yourself be troubled by case 2, which refers to repeated inheritance. As soon as you have read the first three sections of the repeated inheritance chapter, the context in which case 2 occurs should be quite clear.

→ *Chapter 16*.

The origin of a feature is the most remote ancestor from which the feature comes, and its seed is its original form in that ancestor.

None of the reincarnations that the feature may have gone through along the inheritance part as a result of redefinition, effecting or renaming may affect its seed and its origin.

10.28 REDECLARATION RULES

(The rest of this chapter gives the formal rules applying to feature redeclaration. The essential concepts have already been seen, so you may safely skip to the next chapter on first reading.)

---- REMOVE ALL THIS!!! According to the earlier definitions, case ----- is an effecting. Case ------ is an effecting for deferred *f* and effective *g*, a redefinition if they are both deferred or both effective. Clause 5 of the constraint below will preclude the other apparent possibility: *f* effective, *g* deferred.

← *"Effective, deferred feature", page 303* and *"Effecting", page 303*.

In case -------, the text of *C* does not contain any declaration for *f*, but some other inherited feature *g* (which must come from a different parent) effects *f*. It is convenient to treat this implicit and somewhat special case as a redeclaration, along with the explicit and more common case -------.

The above definition says nothing about validity: case ---- simply states that if a declaration uses the name of an inherited feature, we must treat it as a redeclaration (valid or not) of that feature, not as the declaration of a new, or *immediate*, feature. Here is the rule that determines when a redeclaration (explicit or implicit) is valid:

Redeclaration rule *VDRD*

Let *C* be a class and *g* a feature of *C*. It is valid for *g* to be a redeclaration of a feature *f* inherited from a parent *B* of *C* if and only if the following conditions are satisfied.

1 • No effective feature of *C* other than *f* and *g* has the same final name.

2 • The signature of *g* conforms to the signature of *f*.

3 • The Precondition of *g*, if any, begins with **require else** (not just **require**), and its Postcondition, if any, begins with **ensure then** (not just **ensure**).

4 • If the redeclaration is a redefinition (rather than an effecting) the Redefine subclause of the Parent part for *B* lists in its Feature_list the final name of *f* in *B*.

5 • If *f* is inherited as effective, then *g* is also effective.

6 • If *f* is an attribute, *g* is an attribute, *f* and *g* are both variable, and their types are either both expanded or both non-expanded.

7 • *f* and *g* have either both no alias or the same alias.

8 • If both features are queries with associated assigner commands *fp* and *gp*, then *gp* is the version of *fp* in *C*.

Condition 1 prohibits name clashes between effective features. For *g* to be a redeclaration of *f*, both features must have the same final name; but no other feature of the class may share that name. This is the fundamental rule of **no overloading**.

No invalidity results, however, if *f* is deferred. Then if *g* is also deferred, the redeclaration is simply a redefinition of a deferred feature by another (to change the signature or specification). If *g* is effective, the redeclaration is an effecting of *f*. If *g* plays this role for more than one inherited *f*, it both joins and effects these features: this is the case in which *C* kills several deferred birds with one effective stone.

← *The bird-shooting was on page 288.*

Condition 2 is the fundamental type compatibility rule: signature conformance. In the case of a join, *g* may be the redeclaration of more than one *f*; then *g*'s signature must conform to all of the precursors' signatures.

→ *See details below: "RULES ON JOINING FEATURES", 10.29, page 309.*

Signature conformance permits *covariant* redefinition of both query results and routine arguments, but for arguments you must make the new type detachable — **?***U* rather than just *U* — to prevent "catcalls".

Condition 3 requires adapting the assertions of a redeclared feature, as governed by rules given earlier.

← *"REDECLARATION AND ASSERTIONS", 10.17, page 277.*

Condition 4 requires listing *f* in the appropriate Redefine subclause, but only for a redefinition, not for an effecting. (We have a redefinition only if *g* and the inherited form of *f* are both deferred or both effective.) If two or more features inherited as deferred are joined and then redefined together, **every one of them** must appear in the Redefine subclause for the corresponding parent.

← *"Redefine, redefinition", page 285.*

Condition 5 bars the use of redeclaration for turning an effective feature into a deferred one. This is because a specific mechanism is available for that purpose: undefinition. It is possible to apply both undefinition and redefinition to the same feature to make it deferred and at the same time change its signature.

← *As noted: see class E, page 284.*

Condition 6 prohibits redeclaring a constant attribute, or redeclaring a variable attribute into a function or constant attribute. It also precludes redeclaring a (variable) attribute of an expanded type into one of reference type or conversely. You may, however, redeclare a function into an attribute — variable or constant.

Condition 7 requires the features, if they have aliases, to have the same ones. If you want to introduce an alias for an inherited feature, change an inherited alias, or remove it, redeclaration is not the appropriate technique: you must rename the feature. Of course you can still redeclare it as well.

Condition 8 applies to assigner commands. It is valid for a redeclaration to include an assigner command if the precursor did not include one, or conversely; but if both versions of the query have assigner commands, they must, for obvious reasons of consistency, be the same procedure in *C*.

In earlier versions of the language, there was an extra condition, prohibiting a redeclaration from changing an External feature into an Internal one or conversely. Although initially justified by the original conventions on external features, this had become just an implementation constraint with no remaining conceptual justification.

→ *On* External *routines, see chapter 31, especially "BASICS OF EXTERNAL ROUTINES", 31.5, page 818.*

Note, however, that redefining an external routine into a non-external one will usually cause a small performance penalty for the *original* (non-redefined) version, as the Eiffel compiler will probably have to call the external routine through an Eiffel wrapper.

10.29 RULES ON JOINING FEATURES

The last constraint that we need to examine governs the validity and semantics of the join mechanism, used to merge two or more features, of which at most one is effective, by inheriting them under the same name.

It is useful first to extend the notion of precursor:

Precursor (joined features)

A **precursor** of an inherited feature is a version of the feature in the parent from which it is inherited.

← *The definition for in the non-join case was on page 262. A final, more formal definition covering both cases will appear on page 465 at the end of the repeated inheritance chapter.*

Without the join mechanism there was just one precursor; but a feature resulting from the join of two or more deferred features will have all of them as precursors.

Here now is the validity constraint for joining features:

Join rule *VDJR*

It is valid for a class *C* to inherit two different features under the same final name under and only under the following conditions:

1 • After possible redeclaration in *C*, their signatures are identical.

2 • They either have both no aliases or have the same alias.

3 • If they both have assigner commands, the associated procedures have the same final name in *C*.

4 • If both are inherited as effective, *C* redefines both into a common version.

The Join rule indicates that joined features must have exactly the same signature — argument and result types.

→ *"Repeated Inheritance rule", page 430*; *"Repeated Inheritance Conformance rule"*
← *"Redeclaration rule", page 307*.

What matters is the signature after possible redefinition or effecting. So in practice you may join precursor features with different signatures: it suffices to redeclare them using a feature which (as required by point 2 of the Redeclaration rule) must have a signature conforming to all of the precursors' signatures.

→ *A signature conforms to another if every type in it conforms to the corresponding type in the other. See "EXPRESSION AND SIGNA-*
← *Join-cum-effecting was described on page 288.*

If the redeclaration describes an effective feature, this is the case of both joining and effecting a set of inherited features. If the redeclaration describes a feature that is still deferred, it is a redefinition, used to adapt the signature and possibly the specification. In this case, point 4 of the Redeclaration rule requires every one of the precursors to appear in the Redefine subclause for the corresponding parent.

In either case, nothing requires the precursors' signatures to conform to each other, as long as the signature of the redeclared version conforms to all of them. This means you may write a class inheriting two deferred features of the form

```
f (p: P): T …
f (t: Q): U …
```

and redeclare them with

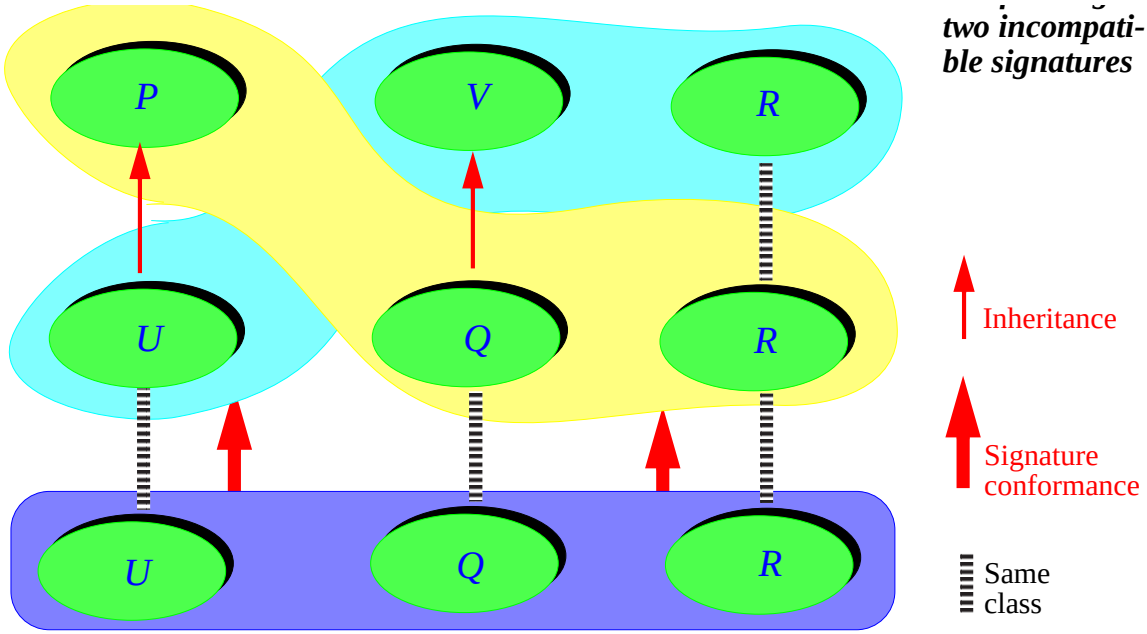
```
f (x: ? R): V …
```

provided *R* conforms to both *P* and *Q* and *V* to both *T* and *U*. No conformance is required between the types appearing in the precursors' signatures (*P* and *Q*, *T* and *U*).

The assumption that the features are "different" is important: they could in fact be the same feature, appearing in two parents of *C* that have inherited it from a common ancestor, without any intervening redeclaration. This would be a valid case of repeated inheritance; here the rule that determines validity is the Repeated Inheritance Consistency constraint. The semantic specification (sharing under the Repeated Inheritance rule) indicates that *C* will have just one version of the feature.

Conditions 2 and 3 of the Join rule are consistency requirements on aliases and on assigner commands. The condition on aliases is consistent with condition 7 of the Redeclaration rule, which requires a redeclaration to keep the alias if any; it was noted in the comment to that rule that redeclaration is not the appropriate way to add, change or remove an alias (you should use renaming for that purpose); neither is join. The condition on assigner commands ensures that any Assigner_call has the expected effect, even under dynamic binding on a target declared of a parent type.

The following figure illustrates a valid case, in which all types involved are non-generic classes (so that conformance is just inheritance). *U* is an heir of *P*, but for the second argument the relation is in the other direction: *Q* is an heir of *V*. Then a redeclaration into a feature of signature [*U*, *Q*], [*R*] will be valid.



This takes care of the validity of the join mechanism. The last rule gives the precise properties of the resulting feature:

Join Semantics rule

Joining two or more inherited features with the same final name, under the terms of the Join rule, yields the feature resulting from their redeclaration if any, and otherwise defined as follows:

1 • Its name is the final name of all its precursors.

2 • Its signature is the precursors' signature, which the Join rule requires to be the same for all precursors after possible redeclaration.

3 • Its precondition is the **or** of all the precursors' combined preconditions.

4 • Its postcondition is the **and** of all the precursors' combined postconditions.

5 • Its Header_comment is the concatenation of those of all precursors.

6 • Its body is deferred if all the precursors are inherited as deferred, otherwise is the body of the single effective precursor.

7 • It is not obsolete (even if some of the precursors are obsolete).

Point 5 leaves the concatenation order unspecified.

In point 7 (corresponding to a rare case) language processing tools should produce an obsolescence message for the class performing the join, but the resulting feature is not itself obsolete.

← *"OBSOLETE FEATURES", 5.21, page 163.*