Clients and exports

7.1 OVERVIEW

Along with inheritance, the client relation is one of the basic mechanisms for structuring software.

In broad terms, a class *C* is a client of a type *S* — which is then a *supplier* of *C* — when it can manipulate objects of type *S* and apply *S*'s features to them.

The simplest and most common way is for *C* to contain the declaration of an entity of type *S*.

This occurs for example when *C* includes a declaration of the form

```
x: S
```

To an entity such as *x*, *C* may apply the features that the designer of *S* has explicitly made **available** (has **exported**) to the clients of *S*. In other words, the client relation allows a class to rely on the facilities provided by another as part of its official interface.

Variants of the relation introduce similar dependencies through other mechanisms, in particular generic parameters.

Although the original definitions introduce "client" in its various forms as a relation between a class and a type, we'll immediately extend it, by considering *S*'s base class, to a relation between classes.

This chapter defines the client relation in its diverse forms; it studies how a class can export its features to its clients, and how these clients can use the exported features. The discussion ends with a solution, resulting from the export mechanism, to an important practical question: how to document a class.

7.2 ENTITIES

Classes become clients of one another by using typed components, **entities** and **expressions**, both denoting run-time values (references or objects). An *entity* of a class *C* is one of the following:

→ *Chapter 19 covers entities, with a full definition on page 504 . Chapter 28 covers expressions.*

- An attribute of *C*.
- A formal argument to a routine of *C*.
- A local variable of a routine of *C*, including (for a function) the predefined entity **Result** denoting the result.
- An Object-Test Local (in an Object_test).
- **Current**.

Any such entity has a type, defined in its declaration.

Expressions are obtained by combining entities and function calls through operators (which themselves denote calls). Any expression has a type, deduced from the type of its components.

It's those entities and expression types that generates the client relation, by making *C* a "simple client" of *T*, as defined below, as soon as it has an entity or expression of type *T*.

7.3 CONVENTIONS

We need a few conventions to simplify the discussion of the client and supplier relations.

It is useful to distinguish between several variants of the client relation: simple client, expanded client and generic client relations. Each is studied below. The more general notion of client is the union of these cases, according to the following definition.

Client relation between classes and types

A class *C* is a **client** of a type *S* if some ancestor of *C* is a simple client, an expanded client or a generic client of *S*.

Recall that the ancestors of *C* include *C* itself. The inclusion of *C*'s ancestors is necessary because the dependencies caused by inherited features are just as significant as those caused by the immediate features of *C*. Assume that an inherited routine *r* of *C* uses a local variable *x* of type *S*; this means that *C* depends on *S*, even if the text of *C* does not mention *S*.

← "RELATIONS INDUCED BY INHERITANCE", 6.5, page 171.

Next, we need to clarify a technical point: when does the discussion of clients and suppliers involve classes, and when is it about types? If, as above, you declare in class *C* the entity *x* as being of type *S*, *S* is a type. That type may be a class, but it may also be a less trivial type; for example, *S* may be the *generically derived* type

← *A similar problem arose for inheritance: syntactically, a* Parent *is a type, not a class, but the definitions in 6.3, page 168 and 6.5, page 171, made it possible to talk about parent* classes.

D [*U*]

where *D* is a generic class and *U*, itself a type, is the actual generic parameter for this particular generic derivation of *D*.

As this example indicates, the client relation in its most basic form holds between a class and a type, not necessarily between a class and another class. It generalizes immediately, however, to a relation between classes, since every type is derived from some class called its *base class*. In most cases, the base class of a type is obvious: for example, in a generic derivation such as *D* [*U*], the base class is *D*; and if a non-generic class is used as a type, it is its own base class. Hence a simple convention:

→ *The complete definition of "base class", for every possible category of type, appears in chapters 11 to 13.*

Client relation between classes

A class *C* is a **client of a class** *B* if and only if *C* is a client of a type whose base class is *B*.

The same convention applies to the simple client, expanded client and generic client relations.

As a result of these conventions, it suffices for the following sections to define what it means for a class *C* to be a client (in one of the three variants) of a *type S*.

The next convention applies to the indirect forms of the relations. If *C* is a client of *S* and *S* is a client of *B*, we will say that *C* is an indirect client of *B*. The full definition reads:

Indirect client

A class *A* is an **indirect client** of a type *S* of base class *B* if there is a sequence of classes $C_1 = A, C_2, \ldots, C_n = B$ such that $n > 2$ and every C_i is a client of C_{i+1} for $1 \leq i < n$.

The indirect forms of the simple client, expanded client and generic client relations are defined similarly.

Finally, we sometimes need to refer to the inverse relations:

Supplier

A type or class *S* is a **supplier** of a class *C* if *C* is a client of *S*, with corresponding variants: simple, expanded, generic, indirect.

7.4 SIMPLE CLIENTS

The most immediate case of the client relation is for a class *C* to be a **simple client** of a type *S*, which is then said to be a **simple supplier** of *C*. This happens in particular whenever *C* contains a declaration of the form

```
x: S
```

Assuming the class skeletons:

```
class A feature
        x: B
        y: C [D]
        …
end
```

```
class B feature
        z: E
        …
end
```

Then *A* is a simple client of *B* and *C*, and *B* a simple client of *E*. *B* and *C* are, conversely, simple suppliers of *A*, and *E* of *B*.

In this example a class becomes a simple client of certain types through the declarations of its entities. *C* will also be a simple client of *S* whenever it contains an expression of type *S*.

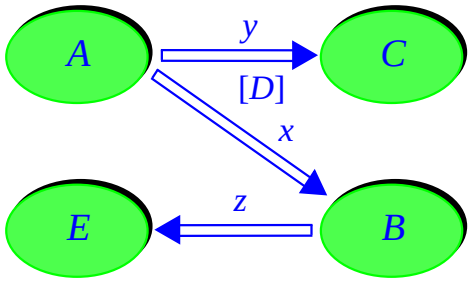
Here is the precise definition:

Simple client

A class *C* is a **simple client** of a type *S* if, in *C*, *S* is the type of some entity or expression or the Explicit_creation_type of a Creation_instruction, or is one of the Constraining_types of a formal generic parameter of *C*, or is involved in the Type of a Non_object_call or of a Manifest_type.

The constructs listed reflect the various ways in which a class may, by listing a type *S* in its text, enable itself to use features of *S* on targets of type *S*.

The suggested graphical representation, illustrated below, shows the simple client relation with a double arrow. The arrow may be labeled above by the name of the corresponding entity, and below by the names of the actual generic parameters in brackets, as with [*D*] for the relation between *A* and *C*.



Simple Clients and suppliers

No constraint restricts how the classes of a system may be simple clients of one another. In particular, cycles are permitted: a class may be its own simple client, both directly according to this definition and indirectly.

For example you might need a class *PERSON* introducing attributes

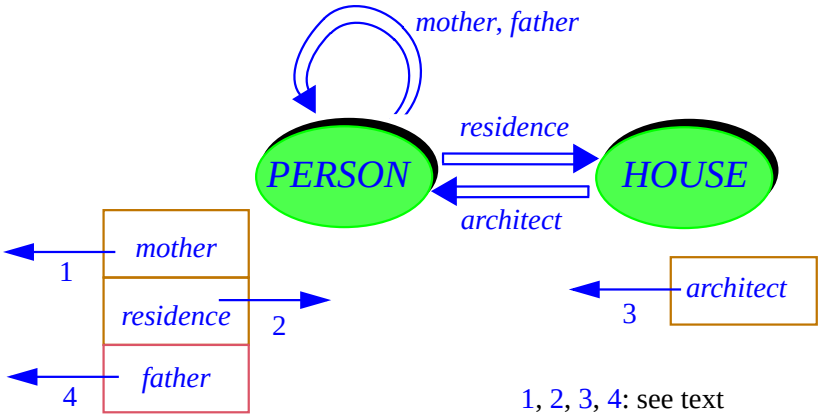
```
mother, father: PERSON
```

This is an example of a direct cycle of the simple client relation. Cycles may also be indirect; for example, a class *HOUSE* might introduce an attribute

```
architect: PERSON
```

with class *PERSON* having an attribute *residence* of type *HOUSE*.

Cycles in the simple client relation



As usual, the ellipses represent classes. The rectangles show typical instances of these classes, with their fields

1, 2, 3, 4: see text

This means is that every person has a mother, father and residence, and every house has an architect. There is nothing contradictory (no vicious circle) in these declarations; at the implementation level they create no difficulty either since it is possible to implement the corresponding attributes as references, as the lower half of the figure suggests by showing typical instances of the classes: references 1, 3 and 4 are to instances of *PERSON*, reference 2 to an instance of *HOUSE*.

Some of these references could also be void, but only if the attribute types are declared as detachable: **?** *PERSON*, **?** *HOUSE*.

→ *Chapter 24*

To avoid any confusion we must distinguish the client relation between *classes* (and types), which is the topic of the chapter, from any specific link that it induces between individual *objects* that are instances of these classes. In particular, a cycle between two classes does not imply a cycle between specific objects; in the situation of the above figure, links 2 and 3 will only connect the objects shown in a “Frank Lloyd Wright setup” (the case of an architect that lives in a house he has designed). Links 1 and 4 cannot be cyclic since no person is his own father or mother. This should in fact be an invariant of the class: *mother* /= *Current*.

7.5 EXPANDED CLIENTS

Expanded types introduce a special variant of the client relation, called "expanded client".

Expanded types describe objects that behave with **copy semantics** rather than reference semantics: an assignment or argument passing will copy the object, not just attach a reference to it. Non-expanded types, which use reference semantics, are called **reference types**.

→ *See chapter 11 for the details of expanded types, starting with 11.9, page 327.*

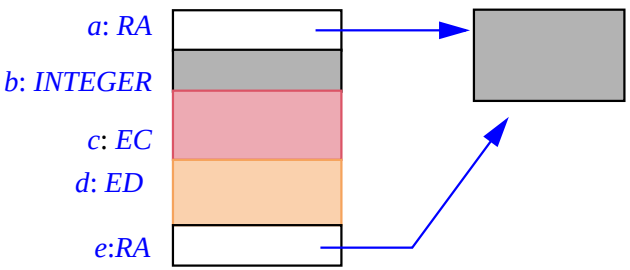
A type is expanded if and only if its base class is itself expanded; it must be declared as **expanded class** rather than just **class**.

An application of expanded classes and types is to describe **composite** objects, the name given to objects that contain **subobjects**. Consider a class declaration with the following attributes (routines omitted)

```
class B feature
        a: RA
        b: INTEGER
        c: EC
        d: ED
        e: RA
end
```

where *RA* is a reference type, *EB* and *ED* are expanded types; *INTEGER*, a basic type, is also expanded. Then instances of *C* can be viewed as composite objects. The figure below shows a typical one

Composite object



The figure shows a conceptual view of the objects and subobjects; it does not necessarily describe the actual representation, since it is always possible to represent expanded fields by references rather than subobjects. See below.

This example illustrates the expanded variant of the client relation:

Expanded client

A class *C* is an **expanded client** of a type *S* if *S* is an expanded type and some attribute of *C* is of type *S*.

Only attributes matter for this definition, since other expressions and queries do not cause subobjects.

The last example and its illustration appear to suggest that we should prohibit cycles in the expanded client relation, as in

```
expanded class EA feature
        c: EC
        …
end
```

```
expandedclass EC feature
        a: EA
        …
end
```

or the even more absurd-looking case of a direct cycle:

```
expandedclass EB feature
        b: EB
        …
end
```

It's indeed not possible physically for every instance of *EC* to contain an instance of *EA* if every instance of *EA* contains an instance of *EC*, or for every instance of *EB* to contain another of the same type.

But in fact such examples — useful or not — create no particular problem and we don't need to prohibit them. Remember that the figure showing expanded fields as subobjects is just an illustration; the only semantic property that matters is that instances of expanded types have *copy semantics*, meaning that:

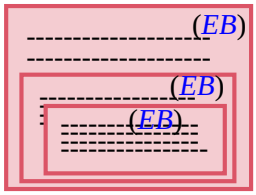
- An assignment or argument passing will copy the object, not just assign a reference.
- An equality operation will compare objects contents, not references.

To support these rules, expanded types have **lazy** initialization semantics: expanded objects need only be created when first accessed.

Any implementation of expanded attributes that supports these properties is acceptable. In particular, while the *subobject* representation is generally preferable when possible (that is to say, in the absence of cycles), it is always possible to use *references* instead, and create the associated objects on demand, as part of lazy initialization. Cycles are then not a problem.

This solution is available for attributes *a*, *c* and *b* in the last example.:

Conceptually, you may consider that if an object OB of the expanded type *EB* has a field of that same type (the same would apply to the indirect case involving instances of *EA* and *EC*), the field still represents a that subobject, just "**written smaller**" inside the first:



Embedded objects

Lazy semantics implies that all subobjects are evaluated only when needed, and an execution can only perform a finite number of such evaluations; so the process will stop and the level of object embedding remains finite.

Of course we can't really "write smaller" in the memory of a computer, so the most obvious implementation will use embedded sub-objects for expanded attributes at the first level only, and then references for the (rare) case of cycles in the expanded relation. But the subobject embedding picture remains applicable conceptually.

Earlier versions of Eiffel had an "Expanded Client rule" prohibiting cycles in the expanded client relation. The lazy semantics of expanded types now makes it unnecessary.

The graphical representation of the expanded client relation uses a double line, as with the simple client relation, but with a brace near the arrowtip:

Expanded client

See corresponding convention for expanded inheritance, page 180.

7.6 GENERIC CLIENTS

Assume that *B* is a generic class, and that class *C* contains a declaration of the form

```
x: B [S]
```

using *S* as actual generic parameter for the generic derivation of *B*.

As seen above, this declaration makes *C* a simple client of *B*. But it also introduces a dependency between *C* and *S*. This dependency is in fact similar to what happens if *C* has an entity or expression of type *S*; this variant of the client relation is called **generic client**.

Generic client, generic supplier

A class *C* is a **generic client** of a type *S* if for some generically derived type *T* of the form *B* [*…, S, …*] one of the following holds:

1 • *C* is a client of *T*.

2 • *T* is a parent type of an ancestor of *C*.

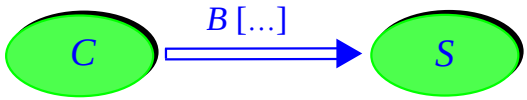
Case 1 captures for example the use in *C* of an entity of type *B* [*S*] (with *B* having just one generic parameter). Case 2 covers *C* inheriting directly or indirectly (remember that *C* is one of its own ancestors) from *B* [*S*].

As case 2 of the definition indicates, *C* may become a generic client of *S* by using *S* as actual generic parameter not just in the type of an entity or expression (as with *x* above), but also in a Parent part, as in

```
class C inherit
        B [..., S,...]
feature
        ...
```

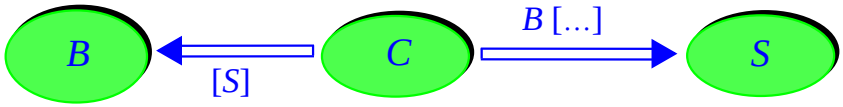
The Parent part may appear not just in *C* itself as here, but in any one of its ancestors: generic client status is passed on through inheritance.

The graphical convention for the generic variant of the client relation uses a double arrow from the generic client to its generic supplier, listing the base class with three dots in brackets […]. For a declaration *x*: *B* [*S*]:



Generic client

The full picture is in this case:



Generic client and simple client

Do not confuse the two forms of client relation arising here: *C* is a simple client of *B*, with *S* as a generic parameter, through the declaration of *x* (left part of the figure); but that declaration also makes *C* a generic client of *S*, assumed here for simplicity to be a non-generic class..

There is no restriction on how the classes of a system may become generic clients of each other.

7.7 EXPORT CONTROLS AND INFORMATION HIDING

The client relation determines how a class may call features of a certain type, on entities of that type. Such calls are subject to *export* controls, implementing a policy of "information hiding".

Assume *C* is a simple or expanded client of *S*. *C* declares one or more entities or expressions of type *S*; let *x*: *S* be one of them. The benefit, for *C*, is to be able to call *S*'s features on entities and expressions such as *x*. The simplest form of call, occurring in *C*, is

→ *See chapter 23 about the various forms, uses and properties of calls.*

```
x.f (...)
```

where *r* is a feature of *S*. This form uses dot notation; forms using operators and assignment procedures are also possible.

Not all such calls, however, are permitted; in particular, not all the features of a class need be callable by all clients. The designer of a supplier class may want to keep some features private, or available to some clients only, because they are only of internal use and subject to change; letting any client access them directly would jeopardize further evolution, by requiring a change of the client classes every time these features change.

→ *Chapter 25 covers the conditions on call validity.*

This is especially true of features that reflect not the services directly offered by a class to its clients, but internal support for the implementation of these services, resulting from specific choices of representation and algorithms. By keeping such features private, the designer of the supplier class protects clients against the effects of later reversals of these choices. This policy is part of **information hiding**, a central principle of software development, which holds that the developer of a module must make a clear distinction between two categories of properties: those which are local to the module itself (its "secrets", or "private properties"); and those that are available to clients ("public properties").

Eiffel supports information hiding in a number of ways, including Design by Contract, the notion of contract view, and the principle of Uniform Access. One of the principal tools for information hiding is the ability for a class to define a specific export status for every one of its features. You can achieve this through two related mechanisms:

- For immediate features (those introduced in the class itself), you may specify export restrictions by listing clients in a Feature_clause. In the absence of such a restriction, features are available to all possible clients.
- For inherited features (those obtained from parents), you may change the export policy specified by each parent through the New_exports subclauses of the corresponding Parent parts. Inherited features not listed tthere retain the export status they had in the parent.

The following discussion explain these two mechanisms in detail.

Restricting exports

You define the export status of an immediate feature by specifying authorized clients in the Feature_clause where it is declared. A Feature_clause begins with the keyword **feature** followed by an optional Clients part; if present, this Clients part lists the classes to which the feature is available.

← Feature_clause *was introduced in* *5.7, page 137*.

If there is no Clients part, then every feature introduced in the Feature_clause is available to any client that cares to use it. So if class *PARAGRAPH* includes a Feature_clause of the form

The **feature** *keyword is not technically part of the* Feature_clause*, but introduces it.*

```
feature
        indent (n: INTEGER)
                -- Indent paragraph by n positions.
        … Procedure body omitted …
```

then any other class may declare an entity *p* of type *PARAGRAPH* and include a call such as

```
p.indent (5)
```

If, however, the Clients part of a Feature_clause is present, it consists of a list of classes in braces, and makes the features introduced by the clause available only to those classes and their descendants.

Here is such a Feature_clause, appearing in a class *LINKABLE* and listing three clients:

```
feature {LINKABLE, LINKED_LIST, TWO_WAY_TREE}
        right: like Current
        put_right (other: like Current)
                        --Make other right neighbor of this object.
                … Procedure body omitted…
```

This Feature_clause introduces two features, *right* and *put_right*, and makes them available to clients *LINKABLE* (the class itself, viewed as its own client), *LINKED_LIST* and *TWO_WAY_TREE*. This means that, for *l* of type *LINKABLE*, calls of the form

```
l.right
l.put_right (…)
```

are permitted if they appear in descendants of any of the classes *LINKABLE, LINKED_LIST* and *TWO_WAY_TREE*.

The next subsections explore two properties visible on this example:

- A class may need to make features available to itself.
- Making a feature available to a class also makes it available to all of its proper descendants.

Exporting to oneself

The above Clients part, appearing in class *LINKABLE*, listed *LINKABLE* itself among the classes to which *right* and *put_right* are available. This is required if the class contains a **qualified call** such as

```
l.put_right (…)
```

with *l* of a type based on *LINKABLE*. If the Feature_clause started with just **feature** {*LINKED_LIST, TWO_WAY_TREE*}, such a qualified call would be invalid outside of the two classes listed and their descendants; in particular, it would be invalid in *LINKABLE* itself.

The reason is clear: a qualified call *x.f* (…) always makes the enclosing class a client of *x*'s type; so the above call makes *LINKABLE* a client of itself, and if *LINKABLE* has made *put_right* selectively available to some clients only this will only be permitted if it has listed itself among them.

Although perhaps strange at first sight, this convention is consistent with the general rules on export. (Making exceptions for clients that happen to be the class itself, or one of its descendants, would lead to complicated rules.) Be sure to note, however, that all this only applies to qualified calls. There is no restriction, in the text of a class, on **unqualified calls** to features of the class itself, as with

→ *Chapter 23*.

```
put_right (…)
```

appearing in a routine of *LINKABLE*, with the semantics ofcalling *put_right* on the **current object**. This is always permitted regardless of the export status of *put_right* — that is, even if *put_right* appears in a Feature_clause whose Clients part does not include *LINKABLE*. Clearly, a secret or selectively available feature such as *put_right* would be useless if it couldn't be called in this way from within the class. Unlike qualified calls such as *l.put_right* (…), such an unqualified call is not considered to make the class a client of itself.

→ *"Target of a call", page 620*.

A general semantic property is that, except for invariant monitoring, an unqualified call *f* (…) will always have the same effect as the qualified call *Current.f* (…). But as a result of this discussion the validity constraints are slightly different: if *f* is not exported to the class itself, the first form may be valid and the second one not.

Exporting to descendants

Making a feature available to a class also makes it available to the proper descendants of that class. This is because a class needs the same privileges that its parents had; for example, it could redefine an inherited routine, changing the original algorithm into a slightly different one, which still needs access to the same information from suppliers.

As a consequence, declaring features in a Feature_clause of the form

```
feature {ANY}
        … Feature declarations …
```

makes them available to all classes, since every developer-defined class is a descendant of *ANY*. Such a clause has the same effect as no Clients part at all, as in

← *"ANY", 6.6, page 172; see also chapter 35 for more details.*

```
feature
        … Feature declarations …
```

Making a feature secret

The export control mechanism as just described gives us, as a special case, the ability to make a feature *f* completely **secret** — available for call to no client. It suffices to declare the feature in a Feature_clause that starts\

```
feature {NONE}
        … Declaration for f and other secret features …
```

where *NONE* is the fictitious class at the bottom of the inheritance hierarchy. Because *NONE* has no usable instance, and no developer-written class can be a descendant of *NONE*, this makes it impossible for any class to use *f* as feature of a qualified call $x.f(\ldots)$.

← *"NONE", 6.7, page 175.*

The treatment of *ANY* and *NONE* for export controls is pleasantly symmetric: **feature** {*ANY*} introduces public features, available to all classes; **feature** {*NONE*} introduces private features, available to no class.

The conventions for shorthands are, however, different:

- **feature** with no Clients part is an abbreviation for **feature** {*ANY*}.
- **feature** {} with an empty Clients part is not permitted by the syntax: the production for Clients requires a Class_list, which cannot be empty.

→ *Page 204 below.*

feature { } could be accepted as a synonym for **feature** {*NONE*} (and actually was in earlier versions of Eiffel, although seldom used), but the language design has settled on a single convention, and chosen the more explicit one for clarity. It is not in the usual Eiffel style to use empty brace or parenthesis enclosures.

Adapting the export status of inherited features

The precedomg discussion has explained the export status of features introduced in a class (although the formal definitions have not yet been given). We also need to know what happens to *inherited* features.

→ *A class "redeclares" a feature if it provides a new declaration for it. This may be either a* redefinition *or an* effecting. *See chapter 10 for details.*

If a feature is redeclared, its new declaration will appear in a Feature_clause, whose Clients part, or absence thereof, will determine the export status as we have just seen. But what is the feature is not redeclared?

The rule is simple. By default, the feature will keep its export status. An heir can change that status, however, through a New_exports part, appearing as part of the Feature_adaptation subclause of a Parent part.

As an example, here is the beginning (Notes clause excluded) of a class of EiffelBase:

```
class FIXED_STACK [T] inherit
        STACK [T]
inherit {NONE}
        ARRAY [T]
                rename
                        put as array_put,
                        …Other renaming pairs omitted …
                export
                        {NONE} -- Implementation
                                all
                end
feature
        …
```

The New_exports part appears with the other possible subclauses of a Feature_adaptation: after Undefine, Redefine and Rename (only the last one present here) and before Select.

← *All these subclauses, and the* Feature_adaptation *as a whole, are optional. The syntax appeared on page 169.*

The New_exports subclause has the general form (shown here with the **export** keyword that introduces the subclause)

```
export
        {A, B, C} -- Feature category 1
                f1, f2, …
        {X, Y}-- Feature category 2
                g1, g2, …
        …
```

meaning: unless a redeclaration specifies a different status, make *f1*, *f2*, … available to clients *A*, *B*, *C* and any of their descendants; make *g1*, *g2*, … available to clients *X*, *Y* and any of their descendants; and so on.

It is good, as illustrated, to include after each Clients list a header comment, such as -- Implementation, indicating the new feature category. The notion of feature category, and the recommendation to list it through a Header_comment, are derived from the practice of labeling feature clauses in a similar way.

← *"FEATURES PART: EXAMPLE", 5.5, page 134* ; *syntax, page 137*.

If, instead of a feature list such as *f1*, *f2*, …, a Clients list is followed by the keyword **all**, then all non-redeclared inherited features are available to the given clients and their descendants, except for any features for which other parts of the subclause specify a different policy. For example, *FIXED_STACK* above hides all features inherited from *ARRAYED_LIST* from all clients, by exporting them to *NONE* only. This is a typical example of a class which inherits its interface from one parent (here *STACK*) and uses another parent (here *ARRAY*) for implementation purposes only.

If no part of the subclause mentions **all** in lieu of a feature list, any non-redeclared inherited feature that is not explicitly given a new export status keeps the exact export status that it had in the parent. Assuming class declarations of the form:

```
class B feature
        x: INTEGER
feature {A}
        f, g, h: INTEGER
feature {NONE}
        i, j, k: INTEGER
end

class C inherit
        B
                export
                        {D} -- Implementation
                                i, j
                        {ANY} -- Access
                                f
                end
end
```

the features of *C* have the following status:

- *x*, available to all clients, *h*, available to *A* and its descendants, and *k*, secret, do not appear in any of the New_exports subclauses: they both keep the status they had in *B*.
- *i* and *j*, regardless of their original status in *B*, are now available to *D* and its descendants.
- *f* is now available to all clients. (Re-exporting to *ANY* is how you make generally available a feature that was selectively available, or secret, in a parent.)

Expanding or restricting the export status

Elaborate changes of export status in inheritance, as in the last example, are uncommon. But two simpler cases causing the use of a New_exports clause do occur fairly often:

- *Extending*: you may want to re-export a feature which was used in the parent for implementation purposes only, but turns out to be of direct value for the clients of the new class, as with *f* in the last example.
- *Restricting*: in designing a new class, you may want to hide features that were exported by a parent.

The second case does not arise in the last example; it does appear in the previous one, for the inheritance of *FIXED_STACK* from *ARRAY*, which hides **all** inherited features. It is not by accident that the Inherit_clause in that case started with

```
inherit {NONE}
```

meaning, as we have seen, non-conforming inheritance. Restricting the export availability of a class is, indeed, applicable only to non-conforming inheritance, as it could cause type problems in the conformance case.

← *"NON-CONFORMING INHERITANCE", 6.8, page 178.*

→ *"NOTES ON THE TYPE POLICY", 25.7, page 665.*

Extending the export status of an inherited feature is always possible, whether in conforming or non-conforming inheritance.

The rule that defines this policy is not a validity constraint but instead a part of the semantics. The "client set" of a feature *f* of a class *C* — the set of classes that have access to *f* for qualified calls — is defined below as the *union* of all applicable Clients lists: the list governing its declaration or redeclaration in *C*, the New_exports if applicable, and the applicable lists from *conforming* parents. So with non-conforming inheritance you can override the original status as you please; but with a conforming parent, even though it is not invalid to write **export** {*NONE*}, this will have no effect since the Clients list {*NONE*} will be combined with the feature's status in the parent, which will then remain applicable.

→ *"Client set of a Clients part", page 203.*

The export status of features

The previous discussion allows us to give a precise definition of the **export status** of any feature, which will determine to what classes the feature is **available** for qualified calls. This notion determines the validity of

```
x.f (...)
```

or the equivalent using operator expressions or assignment procedure calls, appearing in a class *C* which declares *x* of type *S*: the feature of final name *f* in *S* must be available to *B*.

→ *See chapter 25 on call validity. The precise requirement is condition 2 of export validity, page 624.*

We first need a notion of "client set", applying to Clients parts such as {*A*, *B*, *C*} which, as we have seen, may appear both at the beginning of a Feature_clause and in a New_exports subclause:

Client set of a Clients part

The **client set** of a Clients part is the set of descendants of every class of the universe whose name it lists.

By convention, the client set of an absent Clients part includes all classes of the system.

The descendants of a class include the class itself. The "convention" of this definition simplifies the following definitions in the case of no Clients part, which should be treated as if there were a Clients part listing just *ANY*, ancestor of all classes.

No validity rule prevents listing in a Clients part a name *n* that does not denote a class of the universe. In this case — explicitly permitted by the phrasing of the definition — *n* does not denote any class and hence has no descendants; it does not contribute to the client set.

This important convention is in line with the reuse focus of Eiffel and its application to component-based development. You may develop a class *C* in a certain system, where it lists some class *S* in a Clients part, to give *S* access to some of its features; then you reuse *C* in another system that does not include *S*. You should not have to change *C* since no bad consequence can result from listing a class not present in the system, as long as *C* does not itself use *S* as its supplier or ancestor.

Even in a single system, this policy means that you can remove *S* — if you find it is no longer needed — without causing compilation errors in the classes that list it in their Clients parts. With a stricter rule, you would have to remove *S* from every such Clients part. But then if you later change your mind — as part of the normal hesitations of an incremental design process — you would have to put it back in each of these places. This process is tedious, and it wouldn't take many iterations until programmers start making many features public just in case — hardly an improvement for information hiding, the purpose of all this.

Rules on setting the export status

(This section introduces no new concepts but gives a more formal presentation of ideas introduced above. You may skip it on first reading.)

The two constructs that determine the export status of a feature are Clients and New_exports. To conclude this discussion on export controls and information hiding, we need to express their precise syntax, constraints and semantics.

→ *Next section*: "DOCUMENTING THE CLIENT INTERFACE OF A CLASS", 7.8, page 207

Here is the syntax of the Clients part:

Clients

Clients ≜ "**{**" Class_list "**}**"

Class_list ≜ {Class_name "**,**" …}$^+$

This construct may appear in two positions. One is in a New_exports, as seen next; the other is as an optional component of a Feature_clause, as in

← Feature_clause *was specified on page 137.*

feature {*A*,*B*,*C*}
… Feature declarations …

There is **no validity constraint** on Clients part. In particular, it is valid for a Clients part both:

- To list a class that does not belong to the universe.
- To list a class twice.

These properties may at first seem at odds with the language's emphasis on including validity constraints that permit detection of errors and inconsistencies at compile time. But in fact there is no adverse effect:

- As noted, permitting a Clients part in a class *C* to listing a non-exist class *S* gives us useful flexibility. Of course you may misspell a class name in a Clients part and, in the absence of any constraint, not get a validity error. But this is not really cause for concern: if you mean to export *f* to *A* in *C* and mistakenly start the Feature_clause with **feature** {*B*} instead of **feature** {*A*}, then for any call *c1*.*f* (…) with *c1* of type *C* in *A* you will get a validity error. So the absence of a constraint on the class names listed in a Clients part introduces no risk of accidentally violating information hiding requirements.

  This policy contrasts with the Class Type rule, which addresses the only other possible use of a Class_name in the language: as part of a Class_type. There we will need, of course, to require that any class used as part of a type be part of the surrounding universe.

  → *"Class Type rule", page 325.*

- Similar reasoning explains why it is not invalid for a class to appear twice in a Clients part, as in {*A*, *A*}. Export privileges extend to descendants; so if we disallowed **feature** {*A*, *A*} we should also prohibit **feature** {*A*, *B*} if *B* is a proper descendant of *A*, since exporting to *A* also exports to *B*. Such a rule is too complicated for the benefits it brings.

Since there is no restriction on the classes listed in the Class_list, one of them may be the enclosing class or one of its ancestors, allowing the class, as noted earlier, to make a feature selectively available to the current class.

Now for New_exports. It is an optional element of Feature_adaptation in a Parent part, as illustrated by *FIXED_STACK* above, and has the following form:

← *Syntax on page 169.*

Export adaptation

New_exports ≜ **export** New_export_list

New_export_list ≜ {New_export_item ";" …}+

New_export_item ≜ Clients [Header_comment] Feature_set

Feature_set ≜ Feature_list | **all**

Feature_list ≜ {Feature_name "," …}+

← *The optional* Header_comment *indicates a feature category: see* page 200.

A constraint applies to any New_exports clause:

Export List rule *VLEL*

A New_exports clause appearing in class *C* in a Parent part for a parent *B*, of the form

export
{*class_list*$_1$} *feature_set*$_1$
…
{*class_list*$_n$} *feature_set*$_n$

is valid if and only if for every *feature_set*$_i$ (for *i* in the interval *1..n*) that is a Feature_list (rather than **all**):

1 • Every elements of the list is the final name of a feature of *C* inherited from *B*.

2 • No feature name appears more than once in any such list.

To obtain the export status of a feature, we need to look at the Feature_clause that introduces it if it is immediate, at the applicable New_exports clause, if any, if it is inherited, and at the Feature_clause containing its redeclaration if it is inherited and redeclared. In a New_exports, the keyword **all** means that the chosen status will apply to all the features inherited from the given parent.

The following definitions and rules express these properties. They start by extending the notion of "client set" from entire Clients parts to individual features.

Client set of a feature

The **client set** of a feature *f* of a class *C*, of final name *fname*, includes the following classes (for all cases that match):

1 • If *f* is introduced or redeclared in *C*: the client set of the Feature_clause of the declaration for *f* in *C*.

2 • If *f* is inherited: the union of the client sets (recursively) of all its precursors from conforming parents.

3 • If the Feature_set of one or more New_exports clauses of *C* includes *fname* or **all**, the union of the client sets of their Clients parts.

This definition is the principal rule for determining the export status of a feature. It has two important properties:

- The different cases are cumulative rather than exclusive. For example a "redeclared" feature (case 1) is also "inherited" (case 2) and the applicable Parent part may have a New_exports (case 3).
- As a result of case 2, **the client set can never diminish under conforming inheritance**: features can win new clients, but never lose one. This is necessary under polymorphism and dynamic binding to avoid certain type of "catcalls" leading to run-time crashes.

This is what "available", used informally up to now, exactly means:

DEFINITION

Available for call, available

A feature *f* is **available for call**, or just **available** for short, to a class *C* or to a type based on *C*, if and only if *C* belongs to the client set of *f*.

In line with others in the present discussion, the definition of "available for call" introduces a notion about *classes* and immediately generalizes it to *types* based on those classes.

The key validity constraint on calls, export validity, will express that a call $a.f(\ldots)$ can only be valid if *f* is available to the type of *a*.

There is also a notion of "available for creation", governing whether a Creation_call **create** $a.f(\ldots)$ is valid. "Available" without further qualification means "available for call".

→ *"RESTRICTING CREATION AVAILABILITY", 20.7, page 531*.

There are three degrees of availability, as given by the following definition.

DEFINITION

Exported, selectively available, secret

The export status of a feature of a class is one of the following:

1 • The feature may be available to all classes. It is said to be **exported**, or **generally available**.

2 • The feature may be available to specific classes (other than *NONE* and *ANY*) only. In that case it is also available to the descendants of all these classes. Such a feature is said to be **selectively available** to the given classes and their descendants.

3 • Otherwise the feature is available only to *NONE*. It is then said to be **secret**.

This is the fundamental terminology for information hiding, which determines when it is possible to call a feature through a *qualified call* $x.f$. As special cases:

- A feature introduced by **feature** {*NONE*} (case 3) is available to no useful classes.
- A feature introduced by **feature** {*ANY*}, or just **feature**, is available to all classes and so will be considered to fall under case 1.
- A feature introduced by **feature** {*A, B, C*}, where none of {*A, B, C*} is *ANY*, falls under case 2.

Or even **feature** {*ANY, A, B, ...*}; *adding classes after ANY brings nothing.*

A feature available to a class is also available to all the proper descendants of that class.

7.8 -DOCUMENTING THE CLIENT INTERFACE OF A CLASS

Now that we have seen the details of the client and export mechanisms, we can obtain an answer to a central issue of software development, especially relevant to the component-based, reuse-oriented software culture promoted by Eiffel: how can the author of a class provide authors of client and descendant classes, and maintainers of the class itself, with a clear description of the facilities offered?

Selecting features

A class will be documented through its features (as well as other properties such as the class invariant and the list of its parents). The first question is: which features do we show? Not necessarily all of them: for example, a client class needs only the features available to it, while a descendant has access to all features. Also, we might consider inherited features, or not. These observation suggest two orthogonal distinctions:

- Between views relevant to authors of: the class itself; all clients; a specific client (taking into account selective exports as discussed earlier in this chapter); proper descendants.
- Betwen views that take into account: *immediate* features and invariant clauses (those from the class itself) only; *inherited* ones as well.

Retaining the useful combinations gives the following ways of selecting features to document:

	Available to all clients	Available to client *X*	Available to descendants (all features)
Immediate + redeclared	Incremental view	*X*-client incremental view	
All including inherited features (“flat views”)	Client view	*X*-client view	Descendant view

In the first row, we select not only immediate features but also those inherited from a parent and *redeclared*. This is for two reasons:

- The combination of immediate and redeclared features gives us a good idea of the "added value" of the class: what it adds to its parents' features. The term *incremental view* expresses this notion.
- More prosaically, such a view is easy to produce: a simple parsing tool, working on the basis of just one class without having to access its proper ancestors, can process all feature declarations — not having to differentiate between new declarations and redeclarations — and, on the basis of Clients parts, retain the public ones (incremental view) or those available to a specific client (*X*-client view).

It would be possible to spot the redefinitions (by analyzing the Redefine *clauses) but not the effectings.*

In the second row, we include all inherited features.

Contract views

Once we have selected the features to document, what information do we retain for them? The most obvious answer would be to give the source code. But this is usually not appropriate: for a "client programmer" (author of a client class), the class text usually includes implementation details along with interface properties. The principle of information hiding requires that we include only the latter. For the view to be offered to a client programmer, the interface properties include:

- The name of a feature.
- Its signature: types of arguments and result if any.
- The **contracts**: precondition, postcondition.
- Some properties of the class other than its features, in particular the class invariant.

Introducing this new dimension into our classification gives the following variant of the previous table, again retaining useful combinations only:

	Available to all clients, contracts only	Available to client *X*, contracts only	Available to descendants (all features), source text
Immediate + redeclared	Incremental contract view	*X*-client incremental contract view	
All including inherited features ("flat views")	Contract view	*X*-client contract view	Descendant view

The leftmost column yields the most intereresting form of documentation for Eiffel classes: the **contract view**, incremental or not.

Here are the precise definitions leading to this notion. The basic ideas are in the preceding discussion, but the defintions need to take all cases and details into account.

Secret, public

A property of a class text is **secret** if and only if it involves any of the following, describing information on which client classes cannot rely to establish their correctness:

1 • Any feature that is not available to the given client, unless this is overridden by the next case.

2 • Any feature that is not available for creation to the given client, unless this is overridden by the previous case.

3 • The body and rescue clause of any feature, except for the information that the feature is external or Once and, in the last case, its once keys if any.

4 • For a query without formal arguments, whether it is implemented as an attribute or a function, except for the information that it is a constant attribute.

5 • Any Assertion_clause that (recursively) includes secret information.

6 • Any parent part for a non-conforming parent (and as a consequence the very presence of that parent).

7 • The information that a feature is frozen.

Any property of a class text that is not secret is **public**.

Software developers must be able to use a class as supplier on the basis of public information only.

A feature may be available for call, or for creation, or both (cases 1 and 2). If either of these properties applies, the affected clients must know about the feature, even if they can use it in only one of these two ways.

Whether a feature is external (case 3) or constant (case 4) determines whether it is possible to use it in a Non_object_call and hence is public information.

These notions yield the definition of the incremental contract view:

Incremental contract view, short form

The **incremental contract view** of a class, also called its **short form**, is a text with the same structure as the class but retaining only public properties.

Eiffel environments usually provide tools that automatically produce the incremental contract view of a class from the class text. This provides the principal form of software documentation: abstract yet precise, and extracted from the program text rather than written and maintained separately.

The definition specifies the information that the incremental contract view must retain, but not its exact display format, which typically will be close to Eiffel syntax.

Below is an extract — beginning, middle and end — from the incremental contract view of the *HASH_TABLE* class of EiffelBase, displayed at the click of a button by Eiffel Software's EiffelStudio (5.6) running on Windows.

An incremental contract view (extracts)

As you will note from this example, the view relies on syntactic conventions slightly different from those of Eiffel; for example, it uses **class interface** instead of the Eiffel keyword **class**. This avoids any confusion with actual Eiffel, since a short form is not a class *text* but class *documentation*. The above definition indeed leaves environments such freedom as to the exact appearance of such views; it only specifies which information to retain and which to discard.

The next definition introduces the non-incremental variant:

Contract view, flat-short form

The **contract view** of a class, also called its **flat-short form**, is a text following the same conventions as the incremental contract view form but extended to include information about inherited as well as immediate features, the resulting combined preconditions and postconditions and the unfolded form of the class invariant including inherited clauses.

The contract view is the full interface information about a class, including everything that clients need to know (but no more) to use it properly. The "combined forms" of preconditions and postconditions take into account parents' versions as possibly modified by **require else** and **ensure then** clauses, and hence describing features' contracts as they must appear to the clients. The "unfolded form" of the class invariant includes clauses from parents. In all these, of course, we still eliminate any clause that includes secret information, as with the incremental contract view.

The contract view is the principal means of documenting Eiffel software, in particular libraries of reusable components. It provides the right mix of abstraction, clarity and precision, and excludes implementation-dependent properties. Being produced automatically by software tools from the actual text, it does not require extra effort on the part of software developers, and stands a much better chance to remain accurate when the software changes.

The contract views, incremental and full, are a fundamental tool for many aspects of software construction. By providing the right level of abstraction to talk about classes, they constitute the Eiffel method's technique of choice for discussing class designs and documenting reusable components. The resulting documentation is free — no need to hire a technical writer, since environment tools take care of producing the document — and, being extracted from the class text, is not subject to the major risk of software documentation, the *reverse Dorian Gray phenomenon*: ceasing to be truthful as the software evolves.