Style guidelines (*not done*)

34.1 OVERVIEW

To facilitate the exchange of Eiffel software, it is preferable to follow a standardized programming style. This appendix describes a set of guidelines which help in this effort. Clearly, these rules are not part of the language definition.

Many of the rules are rather low-level, dealing with such mundane questions as how to phrase comments, where to put blanks adjacent to parentheses, and whether to use verbs or nouns for routine names. Modest as some of these concerns may seem, they are not to be neglected. Adherence to a uniform style for the more superficial aspects of software texts may indeed be of great benefit to both readers and writers of Eiffel software:

In the process of getting acquainted with previously written classes, you will feel more comfortable if they follow a commonly agreed style, enabling you to understand the details more accurately, and to move on without delay to the deeper aspects of the classes under review.

When you write new classes, or modify existing classes, the existence of simple, well-defined guidelines helps you avoid wasting your time hesitating on minor issues.

Far from stifling their creativity, then, the style discipline described in this chapter encourages software developers to apply it to the true challenges of quality software engineering: design of elegant, modular system architectures; selection of appropriate data structures; and use of the best possible algorithms.

34.2 LETTER CASE

Letter case is not significant for entity, feature and type names. The recommended style observes the following conventions.

Any identifier that may be used as a type, or part of a type, should be written all in upper case. This includes:

- Class names, in all possible uses.
- Formal generic parameters, such as *G* in *LIST* [*G*].
- Basic types (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, their sized variants and *POINTER*), *ARRAY* and *STRING*.

Constant attributes should be written with an initial capital letter, with the rest in lower case, as in

Area: REAL is 43_512/.g*57*;
Red, *Green*, *Blue*, *Yellow*: *INTEGER*

The same convention, initial upper-case letter, also applies to Void, the entity of type NONE representing void references, and to the two predefined entities *Current* and *Result*.

All other features (variable attributes and routines) and local variables should use all-lower-case names.

34.3 CHOICE OF NAMES

Names for features and entities should be clear and informative. Do not use abbreviations, except possibly for formal routine arguments, which are only used in a restricted context.

Complex names should use the underscore character to connect various components, as in

put_right

The use of internal upper-case letters for the same purpose, as in *putAtRight*, contradicts the standard conventions of English and most other languages and is not part of the recommended style.
.FS
Apart from proper names of the form *MacName* or *McName*, the only common use of internal upper-case letters seems to be for composite proper names of French origin as spelled in North American English. This convention, however, is unknown in actual French.

If two related names have some elements in common, make them differ at the beginning, rather than at the end; for example, use *x_position* and *y_position* rather than *position_x* and *position_y*. This will decrease the probability of confusion.

Clarity does not imply length. Although names may be as long as needed to avoid ambiguity, you should resist the temptation to overqualify. In particular, feature names should not include an identification of the enclosing class. For example, a feature for updating a customer's invoice in a class *INVOICE* should be called *update*, not *update_invoice* or *invoice_update*.

The design of the Basic Eiffel Libraries has gone even further in the direction of simplifying and standardizing feature names. This means in particular that the Data Structure Library makes little use of the specific terminology traditionally applied by the computer science literature to each individual kind of data structure. For example, you will **not** find features called

push, pop, top for stacks.

add, remove, oldest, latest for queues.

enter, entry for arrays.

insert, value, search for lists and hash tables.

Such names, widely used in textbooks about algorithms and data structures, highlight the differences between the various structures rather than their common properties.

In contrast, the Eiffel libraries are based on a taxonomy of data structures, grouped into well-structured families such as "dispensers", "chains" and "tables". The taxonomy is directly implemented into the library through multiple inheritance from separate hierarchies of deferred classes.

For an in-depth discussion of these issues and other aspects of library design, see "Eiffel: The Libraries" and the chapter entitled "Lessons from the Design of the Eiffel Libraries" in "An Eiffel Collection". The references are in the bibliography of appendix ====.

Feature names are in line with this approach; they reflect the deeper common properties rather than the superficial differences. Some of the most important universal names are: make (Basic initialization operation; should be creation procedure) *item* (Basic access operation. *p*) *count* (Number of significant items in a structure.) *put* (Basic operation to insert or replace an item.) *force* (Like *put*, but will always succeed when it can. For example, it may resize the structure if full.) *remove* (Basic operation for removing an item.) *wipe_out* (Basic operation for removing all items.) *empty* (Test for absence of significant items. Should return the same value

as *count = 0*.) *full* (Test for lack of space for more items.) *to_external* (Function providing a pointer to actual data structure, for example the sequence of values making up an array or string, useful for transmission to external routines. May have language-specific variants such as *to_c* or *to_fortran*. All such functions should have a result of type *NONE* to preclude any feature application on the Eiffel side.)

from_external
(Inverse of *to_external*: procedure to reinitialize a data structure such as a string from an external form. May have language-specific variants such as *from_c*.)

Although such names as *item*, *put* and *remove* for stacks (replacing the traditional *top*, *push* and *pop*) may be a shock to some users accustomed to the more traditional terminology, this unifying move was felt inevitable if client users are to master easily a large number of powerful reusable classes describing many data structures variants.

The inevitable differences in signatures and specifications should not be compounded by differences in names which (in a typed language where incorrect calls will be detected automatically) only stand in the way of understanding.

If you are familiar with these conventions, you will easily recognize the purpose of the major routines when you explore a class that follows them, and you will be able to find out quickly whether the class suits your needs.

Thanks to their systematic presence in the Basic Libraries, these names have acquired a status which is next in importance to that of the language keywords. Make sure you use them whenever they are applicable.

34.4 GRAMMATICAL CATEGORIES FOR FEATURE NAMES

Since procedures are commands to perform actions, their names should be drawn from **verbs** in the imperative mode: *put*, *write*, *remove* etc.

In contrast, functions and attributes (which are indistinguishable by clients, except through the presence or absence of arguments) describe access to information. A function or attribute of a type other than *BOOLEAN* should usually use a **noun**, possibly qualified by an adjective, as in *item* or *last_transaction*. Sometimes the noun may be implicit; then only the adjective or adjectives remain, as in *last_read*, which really stands for *last_item_read*.

The names of boolean functions or attributes should be of either of two forms:

The name may suggest a question, usually with the prefix *is_*, as in *is_leaf* for a boolean feature used to determine whether a node is a leaf.

Adjectives are also appropriate in some cases, as in *opened* for a boolean feature used to determine whether a file is open.

"opened" rather than "open" because the later might be confused with a verb, indicating a command to open the file.

Of the two possible names for a boolean feature, the one chosen should suggest the property which is **false** in the default case. This is because the default initialization rules will initialize a boolean attribute to false, so that there will be no need for the creation procedure to include a specific initialization. For example, if files are to remain closed until some call explicitly opens them, use an attribute *opened* rather than *closed*. Then the creation procedure of the class does not need to do anything special for this attribute.
.lA
Using a verb such as *get* for a function is usually inappropriate. Functions should not "do" something, but return information in a non-destructive way.

See "Object-Oriented Software Construction" about side effects in functions.

For example, a sequential read operation which advances the input cursor may be implemented as the combination of a procedure *get*, changing the state, and an attribute or function *last_item*, returning the last element read. A call to *get* updates the value of *last_item*, but calling *last_item* several times in a row repeatedly yields the same result.

The input routines from the Kernel Library class STANDARD_FILES, discussed in chapter ====, follow these rules: a procedure such as 'readint' will read an element, and an attribute such as 'lastint' will give access to the last value read.

34.5 GROUPING FEATURES

Classes introducing many features should group them into logical categories. The syntax encourages this by allowing a class to have more than one Feature_clause, each beginning with a Header_comment. (A Header_comment has the same form as a free-comment, but appears as an official although optional component of some construct in the syntax. The constructs which take header comments are Feature_clause, Creation_clause and Routine. The next section will examine header comments of routines.)

Syntax productions: Feature_clause, *page* ====; Creation_clause, *page* ====; Routine, *page* ====.

The presentation of features sketched a class text organized in this way: the Data Structure Library class *LINKED_LIST*, with feature groups introduced by

See ====, page ====.

```
    -- Number of elements
-- Special elements
-- Cursor movement
-- Chaining
-- Representation
```

Such header comments should be short, simple phrases characterizing a set of logically related features.

In some cases it may be more convenient to read a class text in alphabetical order of feature names. Part D of this book indeed used this convention for most of its presentations of library classes in flat-short form. In the class texts themselves, however, grouping by feature categories is usually better; then a good language processing tool (for example through options of the **short** and **flat**commands) will be able to produce alphabetical output from a class organized by category — the reverse being of course impossible.

See ====, page ====, about **short**, *and ====, page ====, about* **flat**.

34.6 HEADER COMMENTS

Every routine should begin with a Header_comment. Here is an example:

```
    distance_to_origin: REAL
-- Distance to point (0, 0)
local
origin: POINT
do
create  origin/.gset_to_origin;
Result := distance (origin)
end
```

Header comments should be informative, clear, and concise. In general, brevity is one of the essential qualities of comments in programs; over-long comments tend to obscure the program text rather than help the reader. The following principles should help achieving brevity.

Avoid repeating information which is obvious from the immediately adjacent program text. For example, the header comment for a routine beginning with

```
    tangent_to (c: CIRCLE; p: POINT): LINE
```

should not be

```
-- Tangent to circle c through point p
```

but just

```
-- Tangent to c through p
```

as it is clear from the function header that c is a circle and p is a point.

For the same reason, the header comment should not usually include restrictions on using the routine (such as “Call only on non-void argument”) since such restrictions are the business of the Precondition clause, which will give a more complete and more precise view.

Avoid noise words and phrases. An example is “Return the...” in explaining the purpose of functions. In the above cases, writing “Return the distance to point (0, 0)” or “Return the tangent to...” does not bring any useful information as the reader knows a function must return something. Another example of a noise phrase is “This routine computes...”, or “This routine performs...”. Instead of

```
-- This routine updates the display according to the user’s last input
```

write

```
-- Update display according to last user input.
```

Every header comment should begin with an upper-case letter.

Do not use abbreviations in header comments. The purpose of a comment is to explain; a reader may not know the meaning of an abbreviation.

Header comments should have the following syntactical form, which parallel rules given above for routine names:

The header comment for a procedure should be a sentence in the imperative, as in the last example. The sentence should end with a period.

The header comment for a non-boolean function should be a nominal phrase, such as “Tangent...” above. A final period is not necessary in this case, unless the comment contains more than one sentence.

The header comment for a boolean function should be a question, ending with a question mark, as in “Is current node a leaf?”.

Header comments should be consistent. If a function of a class has the comment “Length of string”, a routine of the same class should not say “Update width of string” if it acts on the same attribute.

In general, comments should be of a level of abstraction higher than the code that they document. In the case of header comments, the comment should concentrate on the what" of the routine rather than the "how" of the algorithm used.

Finally, remember that much of the important semantic information about the effect of a routine may be captured more precisely and concisely through the Precondition and Postcondition clause than through natural language explanations.

34.7 OTHER COMMENTS

Although this does not appear in the syntax, a class should also begin with a comment. The class comment should be brief and come before the beginning of the class text proper. For a class describing a set of objects, the comment should characterize these objects in the plural, as in

```
-- Binary search trees, pointer representation
```

Other parts of class texts may also include free-comments, used to explain potentially unclear components. They should be indented to the right of the normal text so as not to interfere with the understanding of the software text proper.

Classes and routines should have ending comments repeating their names. These comments are in fact optional parts of the syntax.

See ====, page ==== for comments ending classes and ====, page ==== for comments ending routines.

It is not necessary to label the end of a control structure by a closing comment (such as in "**end** -- *if*"). The nesting depth of control structures should remain small in well-written Eiffel texts, not requiring any supplementary help for matching the beginning and end of each structure.

34.8 EIFFEL NAMES IN COMMENTS

In the examples above in the rest of this book, the name of a feature or other entity appearing in a comment is shown in italics to avoid any confusion with common words. For example a header comment could be of the form:

```
-- Record element under key
```

where *element* and *key* are formal arguments of the enclosing routine.

The corresponding convention in actual software texts (where font variation is not a possibility) is to enclose such an Eiffel name in single quotes (one opening quote ‘, one closing quote ’). So the actual form of the above comment in its class text should be:

```
-- Record element under key
```

Language processing tools which produce typesettable forms of classes should recognize this convention and use italics for Eiffel names quoted in comments. (As seen below, italics is the recommended convention in typeset output.)

34.9 LAYOUT

The recommended layout of Eiffel software texts results from the general form of the syntax, which is essentially an "operator grammar", meaning that any text is a succession of alternating "operators" and "operands". An operator is a fixed language symbol, such as a keyword (**do** etc.) or a separator (semicolon, comma etc.); an operand is a user-chosen symbol (identifier or constant).

As a consequence, the text should follow a "comb-like" structure where every syntactical component either fits on a line together with a preceding operator, or is indented just by itself on one or more lines, as in a comb whose branches normally begin and end with operators:
.pF ""Comb-like layout"

For an example, depending on the size of its components *a*, *b* and *c*, the same Conditional may be written, among other possibilities, as

```
if c then a else b end
```

or

```
if
c
then
a
else
b
end
```

or

```
if c then
a
else b end
```

For indentation, you should always use tabs, not spaces. (The only exception is if you are using a text editor that doesn't handle tabs well. Most modern tools, however, have no such problem.)

The same principle applies to classes and routines. The following extract from the ARRAY class of the Kernel Library illustrates the standard indentation conventions for the different clauses. Ellipses (...) indicate omitted features.

.t1
-- One-dimensional arrays

.t1
note

.t1
names: *array*;
access: *index*;
representation: *array*;
size: *fixed, resizable*

.t1
class *ARRAY* [*T*] **creation**

.t1
make

.t1

.t1

inherit

.t1
INDEXABLE [*T, INTEGER*];

.t1
INDIRECT [*T*];

.t1
BASIC_ROUT

.t1

.t1

feature

.t1
make (*minindex*, *maxindex*: *INTEGER*)
-- If *minindex* <= *maxindex*, allocate array with bounds
-- *lower* and *upper*; otherwise create empty array.
do
upper := *—1*;
-- *lower* initialized to 0 by default, so invariant holds
if *minindex* <= *maxindex* **then**
lower := *minindex*; *upper* := *maxindex*;
actual_lower := *lower*; *actual_upper* := *upper*;
allocate (*maxindex — minindex + 1*)
end
ensure
empty_if_impossible: *minindex > maxindex* **implies** *count = 0*;
consistent_size: *minindex* <= *maxindex* **implies**
(*lower = minindex* **and** *upper = maxindex* **and**
count = upper — lower + 1)
end;

.t1

.t1

lower: *INTEGER*;
-- Minimum current legal index

.t1
upper: *INTEGER*;
-- Maximum current legal index

.t2

```
    item (i: INTEGER): T
-- Entry of index i, if within bounds
require
index_large_enough: lower <= i;
index_small_enough: i <= upper
do
Result := ext_item (area, i — lower )
end;
```

.t2

```
    force (v: T; i: INTEGER)
-- Replace i-th entry by v.
-- Always applicable: resize if i not in current bounds.
local
extra_block_size: INTEGER
do
extra_block_size :=
max (Block_threshold, Extra_percentage star count %eidiv% Hundred);
if i < actual_lower then
resize (i — extra_block_size, upper);
lower := i
elseif i > actual_upper then
resize (lower, i + extra_block_size);
upper := i
else
lower := min (i, lower);
upper := max (i, upper)
end;
put (v, i);
ensure
 inserted: item (i) = v;
higher_count: count >= old count
end;
```

```
    ...
```

.t1

```
feature {NONE} -- Representation details
```

.t1

```
    actual_upper: INTEGER;
    -- Actual upper bound

    .t1
    ...


    .t2

        feature -- Obsolete features

    .t1
    enter_force (i: INTEGER, v: T)
    obsolete "Use ‘force (value, index)’"
    do
    force (v, i)
    ensure
     inserted: item (i) = v;
    end
    ...

        invariant
    consistent_size: count = upper — lower + 1;
    non_negative_size: count >= 0
    end
```

Note in particular the indentation used for routine header comments.

The indentation step is the “tab” character. Blank characters should never be used for indentation.

34.10 OPTIONAL SEMICOLONS

Closely related to layout is the question of optional semicolons. For most repetition constructs which use the semicolon as separator, semicolons are optional (except between two adjacent specimens if the second one begins with an opening parenthesis).

The recommended style is to **omit the semicolons** between items appearing on successive lines. Extensive experimentation has shown that the semicolons impair readability since they add no information and detract from the meaningful components of the software text.

If successive items appear on the same line — as is sometimes useful for short multiple declarations or instructions — the semicolon should of course be included, since mere spaces are not visible enough to delimit the successive elements clearly.

Semicolons are optional between feature clauses (page ====), parent clauses (page ====), declarations of local variables (page ====), assertion clauses (page ====) and instructions (page ====).

34.11 LEXICAL CONVENTIONS

Lexical conventions follow the practice of ordinary text, both in language components and in comments. In particular:

There should be a blank before an opening parenthesis, and after a closing parenthesis, but none after an opening parenthesis or before a closing one. The same rule applies to square brackets.

A comma should always be followed by a blank, never preceded by one.

In a comment, a period should be followed by a blank, never preceded by one.

As an exception to non-software textual practice, the dot (period, full stop) used for qualified feature calls is neither preceded nor followed by a blank, as in *this_window/.gdisplay*. As this example indicates, it is preferable, for typeset texts, to use a very small bullet (appearing in this book and in the output of the **short** command when it is meant for typesetting), more visible than a dot.

Three further conventions govern the use of blanks:

An Assignment or Assignment_attempt symbol (:= or ?=) should be preceded and followed by one blank.

These are single symbols: it would be invalid to insert a blank before the = character.

Arithmetic operators should also have a blank to the left and one to the right. When typesetting an Eiffel text with asterisks in expressions, make sure they appear properly as star; by default, typesetting systems will often print an asterisk as * (appearing too high above the line).

As mentioned on page ====, however, a form with intervening blanks would be valid.

34.12 FONTS

When Eiffel texts are typeset, as in this book, the following font conventions should be observed.

Keywords should appear in bold italics:

class

Type, class, feature and entity names (including predefined types and entities) should appear in italics:

INTEGER
ARRAY
put
x
Result

Comments, especially header comments of routines, should appear in roman font, except for identifiers from the software — denoting classes, features, arguments — which should appear in italics:

-- Change lending rate to highest of *rate1* and *rate2*.

As mentioned above, these names should appear in single quotes in the corresponding source texts.

34.13 GUIDELINES FOR ANNOTATING CLASSES

The Notes clause which optionally begins a class text may be used to record information about the class, for use by class browsing and retrieval tools. Such tools are important in the Eiffel approach to software construction, based on the reuse of industrial quality software modules.

The very idea of a Notes clause assumes a degree of standardization of annotation conventions. This section introduces some important guidelines.

It is important first to put the overall purpose of the Notes clause in perspective. The general principle of documenting Eiffel software is that as much of the documentation as possible should be within the class texts themselves. Documentation and browsing tools should use these texts as their primary source of information.

See “Eiffel: The Environment” about documentation and browsing tools.

Some properties of class designs, however, are of a higher level of abstraction than what is usually expressed in the class text proper. They include annotation categories, descriptions of design and implementation decisions, references to algorithms or data structures as published in the literature etc. The Notes clause is meant for such information. It should record it in a standardized format for use by documentation, archival and retrieval tools. Such tools should enable users to retrieve archived classes using query languages that express queries based on *<index, value>* pairs.

The following guidelines were used in the Basic Eiffel Libraries and are recommended for other software as well.

- Keep the Notes clauses short (2 to 10 entries is typical).
- Avoid repeating information which is in the rest of the class text.

- Use a set of standardized indices for properties that apply to many structures (such as choice of representation).

Such standardized indices are suggested below.

For values, define a set of standardized possibilities for the common cases.

Include positive information only. For example, a *representation* index is used to describe the choice of representation (linked, array, ...). A deferred class does not have a representation. For such a class the clause should not contain the entry *representation*: *none* but simply no entry with the index *representation*. A reasonable query language will make it possible to use a query pair of the form *<representation, NONE>*.

Here are a few of the standard index terms and typical values.

An entry of index *names* records alternative names for a structure. Although a class has only one official name, the abstraction it implements may be commonly known under other names. For example, a “list” is also called a “sequence”.

An entry of index *access* records the mode of access of the data structures. The standard values include the following; more than one value may be listed.

- *fixed* (only one element is accessible at any given time, as in a stack or queue).
- *fifo* (first-in-first-out policy).
- *lifo* (last-in-first-out).
- *index* (access by an integer index).
- *key* (access by a non-integer key)
- *cursor* (access through a client-controlled cursor, as with the list classes).
- *membership* (availability of a membership test).
- *min*, *max* (availability of operations to access the minimum or the maximum).

An entry of index *size* indicates a size limitation. Among common values:

- *fixed* means the size of the structure is fixed at creation time and cannot be changed later (there are few such cases in the library).
- *resizable* means that an initial size is chosen but the structure may be resized (possibly at some cost) if it outgrows that size. For extendible structures without size restrictions this entry should not be present.
- An entry of index *representation* indicates a choice of representation. Value *array* indicates representation by contiguous, direct-access memory areas. Value *linked* indicates a linked structure.

- An entry of index *contents* is appropriate for container data structures, used to keep objects. It indicates the nature of the contents. Possible values include *generic* (for generic classes), *integer_c, real_c, boolean_c, character_c* (for classes representing containers of objects of basic types).

The notion of container data structure was presented in ====, page ====, and ====, page ====.

For example, the *ARRAY_LIST* class describes lists implemented by one or more arrays, chained to each other. The clause in this case is:

note
names: *block_list*;
representation: *array, linked*; -- In this case it is both!
access: *fixed, cursor*;
size: *resizable*;
contents: *generic*