Changes from early versions

G.1 OVERVIEW

The previous appendix summarized the history of Eiffel versions and described the changes from Eiffel 3, as described in the first edition of this book, to Eiffel 5.

The present discussion recalls briefly what had changed from the very first incarnations of Eiffel, especially ISE Eiffel 2 — used in the first (1988) edition of the book *Object-Oriented Software Construction*, which was many people's original introduction to Eiffel — to Eiffel 3. It will provide current Eiffel users with a glimpse of the language's early evolution.

For more historical background see Appendix E, A brief history of Eiffel.

G.2 SCOPE OF THE CHANGES

Whereas changes from Eiffel 3 to Eiffel 5 essentially don't break any existing code, the changes from Eiffel 2 to Eiffel 3 did not guarantee backward compatibility, since it was felt appropriate to tune some of the original constructs. The translation, however, was simple and systematic, enabling ISE to provide a translator that automatically converted most of a system and left only a few items for manual programmer action, such as renaming any identifiers conflicting with new keywords.

The differences were of three kinds:

- Changes to the concrete syntax, improving the consistency of the language and the clarity of software texts.
- Adjustment or clarification of the semantics of a few constructs, taking care of cases which proved confusing, such as the combination of repeated inheritance and redeclaration.
- A few new constructs to increase the expressive power of the language.

G.3 OLDER POST-OOSC-1 EXTENSIONS

Prior to Eiffel 3, the following mechanisms were added in versions 2.1 (mid-1988), 2.2 (mid-1989) and 2.3 (mid-1990) of ISE Eiffel, after the original publication of the book *Object-Oriented Software Construction* (hereafter *OOSC-1*) in March of 1988:

- Constrained genericity, enabling a generic class to place certain requirements, expressed through inheritance, on possible actual generic parameters. (*OOSC-1* in fact mentioned this, but only in an exercise.)
- The Indexing clause (now Notes) for recording important information about a class, to be used by archival, browsing and query tools.
- The Assignment_attempt, with its **?=** symbol, for type-safe assignments going against the inheritance hierarchy, widely imitated in other languages.
- Infix and prefix operators, for more flexible call syntax.
- Expanded types, supporting composite objects and avoiding unnecessary dynamic allocation.
- The Obsolete clause (in classes and routines) for smooth library evolution.
- Unique values to define integer codes without having to choose values.
- The Multi_branch instruction for discriminating between a set of cases without using dynamic binding. (This was limited to character and integer values; the extension to intervals came with Eiffel 3, and to strings and type descriptors with Eiffel 5.)
- The boolean operator for implication (**implies**), which was previously expressed through the operator **or else**.
- Support for double-precision reals (type *DOUBLE*, later removed).
- Basic expanded classes from the Kernel Library, defining *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL* and (then) *DOUBLE*.
- The join mechanism for merging one or more inherited deferred routines with compatible signatures and specifications. (In 2.3 this required a now obsolete keyword, **define**, and an effecting of the resulting features.)
- More flexibility in the interface with other languages, in particular through the introduction of the *$* symbol (@ in 2.3).

The rest of this appendix covers changes from Eiffel 2.3 to Eiffel 3, first introduced in 1993.

G.4 SEMICOLONS

Eiffel originally used semicolons as separators. With Eiffel 3, semicolons were made optional in most cases. For a while, the style rules still recommended including them, until it was realized — partly from comments of students in programming classes — that instead of helping readability they obscured software texts, providing no benefit except in the rare case of multiple instructions on a single line. The style rules were revised to reflect this realization that most instruction-separating semicolons are just noise.

"OPTIONAL SEMICOLONS", 34.10, page 909.

G.5 FEATURE ADAPTATION

The syntax of the Feature_adaptation subclause, in the Parent clause of an Inheritance part, indicating changes in inherited features, was made more regular by the introduction of a required **end** terminator, consistent with the conventions used elsewhere in the language (routine declarations, control structures). Previously, there was no **end**; this meant that a mistakenly added extra semicolon, for example between a Rename and a Redefine subclauses, could make the construct ambiguous, resulting in minor but annoying syntactical errors. This is now harmless, and semicolons have, as noted, been made mostly irrelevant anyway.

Page 169.

G.6 SPECIFYING EXPORT STATUS

Eiffel 3 removed the **export** clause which was used, at the beginning of a class, to specify the export regime of every feature of the class. Instead, there may be more than one Feature_clause; each defines the export regime of the features it introduces. If a Feature_clause just begins with the **feature** keyword with no further qualification, all the features it introduces are publicly available.

Chapter 7, Clients and exports, gives all the details of how to set the export status of features.

To obtain the effect of a secret feature, begin the Feature_clause with

```
feature {NONE}
```

To obtain the effect of a feature available selectively to specified classes, begin the Feature_clause with

```
feature {A, B, C}
```

This also removed the need for the **repeat** subclause (which was part of an **export** clause and served to repeat a parent's export specification). By default, inherited features keep the export status they had in the parent, unless they are redefined. The status of a redefined feature is determined by the qualification of the Feature_clause in which the redefinition occurs. To change the status of an inherited feature that is not redefined, use an **export** subclause in the Feature_adaptation clause at the point of inheritance, as in

```
class D inherit
        C
                export
                        {NONE} all
                        {A, B} remove, count
                        {ANY} put
                end
…
```

Here all features inherited from *C* are secret, except for *remove* and *count*, available to *A* and *B*, and *put*, available to all clients

G.7 ADAPTING PRECONDITIONS AND POSTCONDITIONS

Another important language improvement affects the rule on adaptation of preconditions and postconditions for redefined routines is now a language mechanism, rather than a purely methodological guideline. In pre-version 3, a Precondition or Postcondition always appeared in full, even for a redefined routine for which the assertions had not changed. If they did change, you were only supposed to replace an original precondition with a weaker one, or an original postcondition with a stronger one; but the language did not support these rules directly.

"REDECLARATION AND ASSERTIONS", 10.17, page 277.

It now does. In a redefined routine, an absent Precondition means "keep the original's precondition", and similarly for an absent postcondition. You may change these assertions using the forms

```
require else new_precondition_clause
ensure then new_postcondition_clause
```

which yields as new preconditions and postconditions the **or** and **and**, respectively, of the original versions with the added ones, automatically enforcing the rule on precondition weakening and postcondition strengthening.

G.8 REMOVING AMBIGUITIES IN REPEATED INHERITANCE

Separate paths of repeated inheritance may cause a feature to be redefined in different ways. The 2.3 language specification left it to the implementation to resolve the dynamic binding conflicts that may arise in such a case.

To solve this issue, Eiffel 3 introduced a Select clause, in the Inheritance part for a class. An example, assuming *B* and *D* both inherit a feature *f* from a common ancestor *A* and both redefine it, was:

```
class D inherit
        B
                rename f as bf select bf end
                        -- This select the B version for
                        -- dynamic binding from A.
        C
                rename f as cf end
…
```

A potential ambiguity arises only with calls of the form *a1*.*f* for *a* declared of type *A* but dynamically attached to an instance of *D*. The **select** resolves this ambiguity by prescribing the use of *bf*, the *B* version, in this case.

Eiffel 5 replaces this mechanism by the mechanism of non-conforming inheritance, slightly less flexible but simpler.

"THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT", 16.13, page 455.

G.9 RENAMING, REDEFINING, UNDEFINING AND JOINING

In pre-version 3, it was possible to duplicate an inherited feature by renaming it and keeping the old one under a different name; dynamic binding would then apply to entities of the parent type will trigger the redefined version. This mechanism was difficult to explain and was replaced by the Select clause just described (then by non-conforming inheritance). It was in fact unnecessary since repeated inheritance also achieves feature duplication in a more uniform way.

Complementing Redefine, a new clause, Undefine, was introduced to allow de-effecting a feature inherited in effective form, making it deferred. A related constraint was added to prohibit redefining an effective feature into a deferred one, since one may now use undefinition instead.

"UNDEFINING A FEATURE", 10.19, page 283.

In an extension and simplification of the language semantics, inheriting two or more deferred features under the same name will yield a single deferred feature. This is known as the join mechanism and is useful to merge abstractions. An essentially equivalent mechanism existed in pre-version 3 but required the inheriting class to effect the features and to mark them using the keyword **define** (not a reserved word any more). These restrictions do not apply any more.

By combining the previous two possibilities, you may merge a set of effective features inherited from parents, one of these features imposing its implementation on the others.

G.10 SYNONYMS

Eiffel 3 introduced the possibility of introducing two or more features with a single declaration, as in

"SYNONYMS AND MULTIPLE DECLARATION", 5.18, page 159

```
f1, f2 (…) is …
```

This is equivalent to duplicate declarations; the features declared together are not otherwise connected. Redefining or renaming one in a proper descendant has no effect on the others.

G.11 FROZEN FEATURES

To preserve not just the specification of a feature (through its assertions) but also its exact implementation in descendants, you may, since Eiffel 3, declare it as **frozen**. This prevents any redefinition in descendants. Combined with the synonym mechanism, as in

"REDECLARATION RULES", 10.28, page 306.

```
frozen f1, f2 (…) is …
```

which prevents *f1* from being redefined, but does not so restrict *f2*, this makes it possible to provide both a fixed version and a redefinable one. This scheme can be used for a number of features of the universal class *ANY*, such as copy, close, is_equal, which have both a standard version and one adaptable to any class.

G.12 ANCHORING TO A FORMAL ARGUMENT

In an anchored declaration of the form **like** *anchor*, Eiffel 3 made it possible to use *anchor* not just *Current* or an attribute of the enclosing class, but also, in a routine text, a formal argument of that routine, as in

"ANCHOREDTYPES", 11.10, page 331.

```
equal (some: ANY; other: like some): BOOLEAN is …
clone (other: ANY): like other is …
```

In a call to *equal*, the type of the second actual argument must conform to that of the first. In *y* := *clone* (*x*), the type of *x* must conform to that of *y*.

G.13 CREATION SYNTAX

Eiffel 1 and 2 permitted a single creation mechanism per class, called under the form *x*.*Create*. Eiffel 3 introduced the notion of multiple creation procedures, and a syntax of the form

```
! ! x.make (arg1, …)        -- With creation procedure make
! ! x                       -- Without a creation procedure
```

or, if *D* is a descendant of the type declared for *x*:

```
! TYPE ! x.make (arg1, …)
! TYPE ! x
```

The idea was right but the syntax, with its reliance on a special symbol **!**, departed from the usual principles of clarity of Eiffel. It was replaced in Eiffel 5 by a keyword-based form, using the keyword **create**.

Chapter 20 discusses creation.

G.14 UNIFORM SEMANTICS FOR DOT NOTATION

The *x*.*Create* notation of Eiffel 1 and 2 was not the only case in which the dot in *x*.*f* had special semantics. For all "normal" *f*, the notation x dot f described the application of feature *f* to the object attached to *f*, and required *x* to be non-void, triggering an exception otherwise.

The convention was different, however, for a small set of special language-defined features: *Create*, *Clone*, *Forget*, *Void* and *Equal*. For these, the operation really applied to the reference value of *x*, and was legal even if *x* was void (not attached to any object).

These cases were removed in Eiffel 3 to ensure full consistency: dot notation always has the semantics of an operation applicable to an object, and requires *x* to be non-void. A void *x* will cause an exception.

Clone, *Forget*, *Void* and *Equal* are no longer reserved words of the language; instead, the operations use features of the universal class *ANY*, of which all Eiffel classes are descendants. These features' names (*clone*, *Void* and *equal*) are normal identifiers, and proper descendants of *ANY* may rename the features. The cloning instruction *y*.*Clone* (*x*) is now written as the assignment *y* := *clone* (*x*). The instruction *x*.*Forget* is written as the

assignment *x* := *Void*. Feature *Void* of class *ANY* returns a reference of type *NONE*, the class that has no instances. The test for a void reference, previously written *x*.*Void*, is now *x* = *Void*. The object equality test, instead of *x*.*Equal* (*y*), is now *equal* (*x*, *y*). Since the routines involved are normal features of *ANY*, descendants may redefine them while, as noted, always retaining their frozen synonyms.

G.15 MANIFEST ARRAYS

In the same way that a *STRING* object may be given in manifest form (such as "*some string value*"), rather than by successive calls to fill its character positions, Eiffel 3 introduced manifest arrays, such as

"MANIFESTARRAYS", 36.6, page 927.

```
<<val1, val2, …>>
```

which defines an array by its elements. Complemented in Eiffel 5 by tuples, this provides a simple way to achieve the effect of routines with a variable number of arguments.

G.16 DEFAULT RESCUE

It is often convenient to define a default exception response for those routines which do not have a specific Rescue clauses. In pre-version 3, this was done by having a Rescue clause at the class level. The rescue clause was not passed on to descendants because of potential conflicts in the case of multiple inheritance.

As simpler and more flexible convention was introduced by Eiffel 3. The universal class *ANY* has a procedure *default_rescue*, which does nothing. Any class may redefine this procedure to perform specific exception handling actions. Any routine with no Rescue clause is considered to have a Rescue clause that just calls *default_rescue*. This means that any exception occurring in such a routine will lead to the default exception handling mechanism defined at the level of its class.

"THE DEFAULT RESCUE", 26.5, page 686.

G.17 EXPANDED CLASSES

As a notational facility, Eiffel 3 made it possible to declare a class as **expanded class** *E*, implying that any type based on *E* will be expanded. Previously, you could use the type **expanded** *T* based on an existing type *T*, but you couldn't specify that a class gives expanded type by default.

"CLASS HEADER", 4.9, page 124.

G.18 SEMANTICS OF EXPANDED TYPES

In what was probably the only non-trivial modification of an existing semantic property, the effect of an assignment

```
ref := exp
```

where the type of *ref* is a reference type and the type of *exp* is expanded, is specified as creating a new object identical to the value of *exp* (a clone) and attaching *ref* to it.

Table titled "The semantics of conformance reattachment", page 590.

Previously, no cloning occurred; *ref* would just become attached to the value of *exp*, a sub-object or some other object. This introduced a possibility for objects to contain references to sub-objects of other objects. This possibility, of dubious benefit, appears to have been used rarely if ever; it did, however, considerably complicate the run-time model and the implementation, in particular the garbage collector.

G.19 FREE INFIX AND PREFIX OPERATORS

Infix and prefix operators, restricted in Eiffel 2 to predefined symbols — arithmetic such as +, relational such as <, boolean such as **and** — now enjoy full syntactic status: you may give an infix or prefix alias to any function with the appropriate signature (no argument for prefix, one argument for infix), and define your own "free operators", whose symbols must start in Eiffel 3 with one of the four characters @ # | &. Eiffel 5 further generalized this to almost arbitrary names.

"OPERATOR FEATURES", 5.15, page 154.

For compatibility with tradition, boolean operators still use alphabetic keywords (such as **and** and **or else**). They are the only ones, however; integer operators use non-alphabetic symbols: **// and ** replaced the **div** and **mod** of Eiffel 2.

G.20 OBSOLETE CLAUSE

For consistency, the Obsolete clause of an obsolete routine now appears after the **is** keyword rather than before.

"OBSOLETE FEATURES", 5.21, page 163.

A class may also have an obsolete clause, indicating that usage of the class as a whole is discouraged — because you have written a better version that is not fully compatible, or just prefer a different class name. The Obsolete clause in this case comes just before the Class_header (that is to say, before **class**, **deferred class** or **expanded class**, but after the Notes clause if any).

G.21 RESERVED WORDS

Appendix K lists reserved words. See also, in the appendix before the present one, "LEXICAL AND SYNTACTIC CHANGES", F.6, page 1060.

The following ten names could be used as identifiers in pre-3 Eiffel. They became reserved words with Eiffel 3:

alias, **all**, **creation**, **elseif**, **frozen**, *NONE*, *POINTER*, **select**, **separate**, **strip**

The following eleven, decreasing the overall count, ceased to be reserved:

Clone, *Create*, **define**, **div**, **elsif**, *Equal*, *Forget*, **mod**, **name**, *Nochange*, **repeat**

(Other than not being needed any more, **name** may have been the worst choice of keyword in the history of programming languages, as every Eiffel beginner was bound to use it as identifier in a simple application, then wonder why the compiler was complaining.) The change from **elsif** to **elseif** reflected the general rule that no Eiffel reserved word should use an abbreviation, although in the absence of a proper English word for the associated concept **elseif** remains, to this day, the only reserved word in the language that does not consist of a single English word.

Of the new Eiffel 3 reserved words listed above, several have lost their reserved status in Eiffel 5: **creation** (now merged with **create**, coming back from Eiffel 1 and 2 with a new font style), **select** and **strip**. Don't use them yet as identifiers, however, since compilers such as ISE Eiffel may still for a while support the Eiffel 3 constructs in the sake of compatibility.

The role of **creation** in Eiffel 3 was to introduce the construct Creators that lists the creation procedures of a class. It's simpler and clearer to use the same keyword **create** as in creation instructions.

Constructs Creators, *page 539, and* Creation_instruction, *page 543*.

G.22 OTHER LEXICAL CHANGES

To improve readability of manifest number constants (integers, reals), Eiffel 3 introduced the possibility of using underscores to delimit groups of three digits in both the integral and (for a real constant) decimal parts. The commas do not affect the value. For example, 62_525_300.751_6 denotes the same value as 62525300.7516.

"INTEGER CONSTANTS", 29.5, page 782.; *"REAL CONSTANTS", 29.6, page 782*

The representation of special characters uses the percent sign **%** rather than the backslash ****.

"CHARACTER CONSTANTS", 29.7, page 783.