An Eiffel Tutorial

Interactive Software Engineering

Manual identification

Title: *An Eiffel Tutorial*, ISE Technical Report TR-EI-66/TU.

Publication history

First published July 2001. Corresponds to release 5.0 of the ISE Eiffel environment.

Author

Bertrand Meyer.

Software credits

See acknowledgments in book Eiffel: The Language.

Cover design

Rich Ayling.

Copyright notice and proprietary information

Special duplication permission for educational institutions

Degree-granting educational institutions using ISE Eiffel for teaching purposes as part of the Eiffel University Partnership Program may be permitted under certain conditions to copy specific parts of this book. Contact ISE for details.

An Eiffel tutorial

This document is available both locally, as part of the ISE Eiffel delivery, and on the eiffel.com Web site, in both HTML and PDF versions. See the list of introductory documents.
This is **not** an introduction to the EiffelStudio development environment. Follow the preceding link for a Guided Tour of EiffelStudio (HTML or PDF).

You will also find there a shorter introduction: “Invitation to Eiffel”.

1 OVERVIEW

Eiffel is a method and language for the efficient description and development of quality systems.

As a language, Eiffel is more than a programming language. It covers not just programming in the restricted sense of implementation but the whole spectrum of software development:

- *Analysis*, *modeling* and *specification*, where Eiffel can be used as a purely descriptive tool to analyze and document the structure and properties of complex systems (even non-software systems).
- *Design* and *architecture*, where Eiffel can be used to build solid, flexible system structures.
- *Implementation*, where Eiffel provides practical software solutions with an efficiency comparable to solutions based on such traditional approaches as C and Fortran.
- *Maintenance*, where Eiffel helps thanks to the architectural flexibility of the resulting systems.
- *Documentation*, where Eiffel permits automatic generation of documentation, textual and graphical, from the software itself, as a partial substitute for separately developed and maintained software documentation.

Although the language is the most visible part, Eiffel is best viewed as a **method**, which guides system analysts and developers through the process of software construction. The Eiffel method is focused on both productivity (the ability to produce systems on time and within budget) and quality, with particular emphasis on the following quality factors:

- *Reliability*: producing bug-free systems, which perform as expected.
- *Reusability*: making it possible to develop systems from prepackaged, high-quality components, and to transform software elements into such reusable components for future reuse.
- *Extendibility*: developing software that is truly *soft* — easy to adapt to the inevitable and frequent changes of requirements and other constraints.
- *Portability*: freeing developers from machine and operating system peculiarities, and enabling them to produce software that will run on many different platforms.
- *Maintainability*: yielding software that is clear, readable, well structured, and easy to continue enhancing and adapting.

2 GENERAL PROPERTIES

Here is an overview of the facilities supported by Eiffel:

- Completely *object-oriented* approach. Eiffel is a full-fledged application of object technology, not a "hybrid" of O-O and traditional concepts.
- *External interfaces*. Eiffel is a software composition tool and is easily interfaced with software written in such languages as C, C++, Java and C#.
- *Full lifecycle support*. Eiffel is applicable throughout the development process, including analysis, design, implementation and maintenance.
- *Classes* as the basic structuring tool. A class is the description of a set of run-time objects, specified through the applicable operations and abstract properties. An Eiffel system is made entirely of classes, serving as the only module mechanism.
- *Consistent type system*. Every type is based on a class, including basic types such as integer, boolean, real, character, string, array.
- *Design by Contract*. Every system component can be accompanied by a precise specification of its abstract properties, governing its internal operation and its interaction with other components.
- *Assertions*. The method and notation support writing the logical properties of object states, to express the terms of the contracts. These properties, known as assertions, can be monitored at run-time for testing and quality assurance. They also serve as documentation mechanism. Assertions include preconditions, postconditions, class invariants, loop invariants, and also appear in "check" instructions.

- *Exception handling*. You can set up your software to detect abnormal conditions, such as unexpected operating system signals and contract violations, correct them, and recover
- *Information hiding*. Each class author decides, for each feature, whether it is available to all client classes, to specific clients only, or just for internal purposes.
- *Self-documentation*. The notation is designed to enable environment tools to produce abstract views of classes and systems, textual or graphical, and suitable for reusers, maintainers and client authors.
- *Inheritance*. You can define a class as extension or specialization of others.
- *Redefinition*. An inherited feature (operation) can be given a different implementation or signature.
- *Explicit redefinition*. Any feature redefinition must be explicitly stated.
- *Subcontracting*. Redefinition rules require new assertions to be compatible with inherited ones.
- *Deferred features and classes*. It is possible for a feature, and the enclosing class, to be specified — including with assertions — but not implemented. Deferred classes are also known as abstract classes.
- *Polymorphism*. An entity (variable, argument etc.) can become attached to objects of many different types.
- *Dynamic binding*. Calling a feature on an object always triggers the version of the feature specifically adapted to that object, even in the presence of polymorphism and redefinition.
- *Static typing*. A compiler can check statically that all type combinations will be valid, so that no run-time situation will occur in which an attempt will be made to apply an inexistent feature to an object.
- *Assignment attempt* ("type narrowing"). It is possible to check at run time whether the type of an object conforms to a certain expectation, for example if the object comes from a database or a network.
- *Multiple inheritance*. A class can inherit from any number of others.
- *Feature renaming*. To remove name clashes under multiple inheritance, or to give locally better names, a class can give a new name to an inherited feature.
- *Repeated inheritance*: *sharing and replication*. If, as a result of multiple inheritance, a class inherits from another through two or more paths, the class author can specify, for each repeatedly inherited feature, that it yields either one feature (sharing) or two (replication).
- *No ambiguity under repeated inheritance*. Conflicting redefinitions under repeated inheritance are resolved through a "selection" mechanism.
- *Unconstrained genericity.* A class can be parameterized, or "generic", to describe containers of objects of an arbitrary type.

- *Constrained genericity.* A generic class can be declared with a generic constraint, to indicate that the corresponding types must satisfy some properties, such as the presence of a particular operation.
- *Garbage collection*. The dynamic model is designed so that memory reclamation, in a supporting environment, can be automatic rather than programmer-controlled.
- *No-leak modular structure*. All software is built out of classes, with only two inter-class relations, client and inheritance.
- *Once routines*. A feature can be declared as "once", so that it is executed only for its first call, subsequently returning always the same result (if required). This serves as a convenient initialization mechanism, and for shared objects.
- *Standardized library.* The Kernel Library, providing essential abstractions, is standardized across implementations.
- *Other libraries*. Eiffel development is largely based on high-quality libraries covering many common needs of software development, from general algorithms and data structures to networking and databases.

It is also useful, as in any design, to list some of what is **not** present in Eiffel. The approach is indeed based on a small number of coherent concepts so as to remain easy to master. Eiffel typically takes a few hours to a few days to learn, and users seldom need to return to the reference manual once they have understood the basic concepts. Part of this simplicity results from the explicit decision to exclude a number of possible facilities:

- *No global variables*, which would break the modularity of systems and hamper extendibility, reusability and reliability.
- *No union types* (or record type with variants), which force the explicit enumeration of all variants; in contrast, inheritance is an open mechanism which permits the addition of variants at any time without changing existing code.
- *No in-class overloading* which, by assigning the same name to different features within a single context, causes confusions, errors, and conflicts with object-oriented mechanisms such as dynamic binding. (Dynamic binding itself is a powerful form of inter-class overloading, without any of these dangers.)
- *No goto instructions* or similar control structures (break, exit, multiple-exit loops) which break the simplicity of the control flow and make it harder or impossible to reason about the software (in particular through loop invariants and variants).
- *No exceptions to the type rules*. To be credible, a type system must not allow unchecked "casts" converting from a type to another. (Safe cast-like operations are available through assignment attempt.)
- *No side-effect expression operators* confusing computation and modification.
- *No low-level pointers, no pointer arithmetic*, a well-known source of bugs. (There is however a type *POINTER*, used for interfacing Eiffel with C and other languages.)

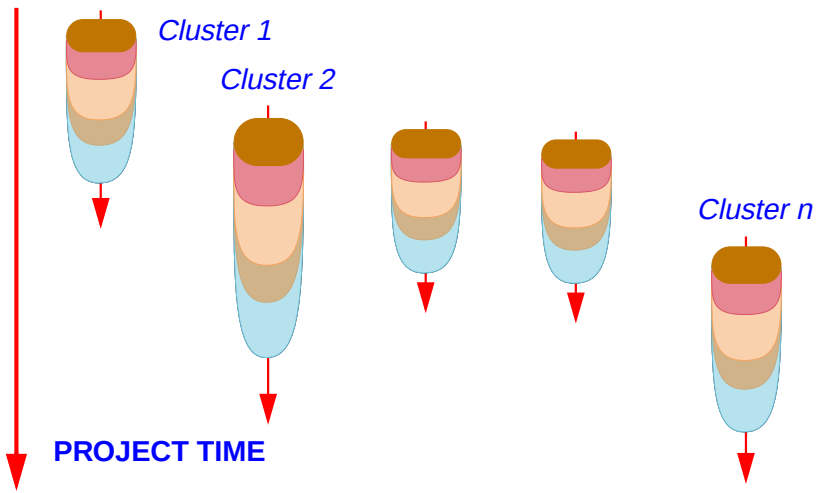
3 THE SOFTWARE PROCESS IN EIFFEL

Eiffel, as noted, supports the entire lifecycle. The underlying view of the system development lifecycle is radically different not only from the traditional "Waterfall" model (implying a sequence of discrete steps, such as analysis, global design, detailed design, implementation, separated by major changes of method and notation) but also from its more recent variants such as the spiral model or "rapid prototyping", which remain predicated on a synchronous, full-product process, and retain the gaps between successive steps.

Clearly, not everyone using Eiffel will follow to the letter the principles outlined below; in fact, some highly competent and successful Eiffel developers may disagree with some of them and use a different process model. In the author's mind, however, these principles fit best with the language and the rest of the method, even if practical developments may fall short of applying their ideal form.

Clusters and the cluster model

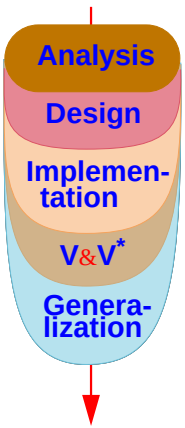
Unlike earlier approaches, the Eiffel model assumes that the system is divided into a number of subsystems or **clusters**. It keeps from the Waterfall a sequential approach to the development of each cluster (without the gaps), but promotes **concurrent engineering** for the overall process, as suggested by the following picture:.



The cluster model*: *sequential and concurrent engineering

The Eiffel techniques developed below, in particular information hiding and Design by Contract, make the concurrent engineering process possible by letting the clusters rely on each other through clearly defined interfaces, strictly limiting the amount of knowledge that one must acquire to use the cluster, and permitting separate testing. When the inevitable surprises of a project happen, the project leader can take advantage of the model's flexibility, advancing or delaying various clusters and steps through dynamic reallocation of resources.

Each of the individual cluster lifecycles is based on a continuous progression of activities, from the more abstract to the more implementation-oriented:



Individual cluster lifecycle

*V&V: Validation and Verification

You may view this picture as describing a process of accretion (as with a stalactite), where each steps *enriches* the results of the previous one. Unlike traditional views, which emphasize the multiplicity of software products — analysis document, global and detailed design documents, program, maintenance reports … —, the principle is here to treat the software as a **single product** which will be repeatedly refined, extended and improved. The Eiffel language supports this view by providing high-level notations that can be used throughout the lifecycle, from the most general and software-independent activities of system modeling to the most exacting details of implementation tuned for optimal run-time performance.

These properties make Eiffel span the scope of both "object-oriented methods", with their associated notations such as UML and supporting CASE tools (whereas most such solutions do not yield an executable result), and "programming languages" (whereas most such languages are not suitable for design and analysis).

Seamlessness and reversibility

The preceding ideas define the **seamless approach** embodied by Eiffel. With seamlessness goes **reversibility**: the ability to go back, even late in the process, to earlier stages. Because the developers work on a single product, they can take advantages of bouts of late wisdom — such as a great idea for adding a new function, discovered only at implementation time — and integrate them in the product. Traditional approaches tend to discourage reversibility because it is difficult to guarantee that the analysis and design will be updated with the late changes. With the single-product principle, this is much easier to achieve.

Seamlessness and reversibility enhance extendibility by providing a direct mapping from the structure of the solution to the structure of the problem description, making it easier to take care of customers' change requests quickly and efficiently. They promote reliability, by avoiding possible misunderstandings between customers' and developers' views. They are a boost to maintainability. More generally, they yield a smooth, consistent software process that helps both quality and productivity.

Generalization and reuse

The last step of the cluster lifecycles, Generalization, is unheard of in traditional models. Its task is to prepare the results of a cluster for reuse across projects by looking for elements of general applicability, and transform them for inclusion in libraries.

Recent object-oriented literature has used the term "refactoring" to describe a process of continuous improvement of released software. Generalization includes refactoring, but also pursues a more ambitious goal: helping turn *program elements* (software modules useful only as part of a certain program) into *software components* — reusable parts with a value of their own, ready to be used by diverse programs that can benefit from their capabilities.

Of course not all companies using the method will be ready to include a Generalization phase in their lifecycles. But those which do will see the reusability of their software greatly improved.

Constant availability

Complementing the preceding principles is the idea that, in the cluster lifecycle, the development team (under the responsibility of the project leader) should at all times maintain a *current working demo* which, although covering only a part of the final system, works well, and can be demonstrated or — starting at a suitable time — shipped as an early release. It is not a "prototype" in the sense of a mockup meant to be thrown away, but an initial iteration towards the final product; the successive iterations will progress continuously towards until they become that final product.

Compilation technology

The preceding goals benefit from the ability to check frequently that the current iteration is correct and robust. Eiffel supports efficient compilation mechanisms through such mechanisms as the **Melting Ice Technology** in ISE's EiffelStudio. The Melting Ice achieves immediate recompilation after a change, guaranteeing a recompilation time that's a function of the size of the *changes*, not of the system's overall size. Even for a system of several thousand classes and several hundred thousand lines, the time to get restarted after a change to a few classes is, on a typical modern computer, a few seconds.

Such a "melt" (recompilation) will immediately catch (along with any syntax errors) the type errors — often the symptoms of conceptual errors that, if left undetected, could cause grave damage later in the process or even during operation. Once the type errors have been corrected, the developers should start testing the new functionalities, relying on the power of **assertions** — explained in "DESIGN BY CONTRACT™, ASSERTIONS, EXCEPTIONS", page 38 — to kill the bugs while they are still larvae. Such extensive unit and system testing, constantly interleaved with development, plays an important part in making sure that the "current demo" is trustworthy and will eventually yield a correct and robust product.

Quality and functionality

Throughout the process, the method suggests maintaining a constant **quality** level: apply all the style rules, put in all the assertions, handle erroneous cases (rather than the all too common practice of thinking that one will "make the product robust" later on), enforce the proper architecture. This applies to all the quality factors except possibly reusability (since one may not know ahead of time how best to generalize a component, and trying to make everything fully general may conflict with solving the specific problem at hand quickly). All that varies is **functionality**: as the project progresses and clusters come into place, more and more of the final product's intended coverage becomes available. The project's most common question , "Can we ship something yet?", translates into "Do we cover enough?", not "Is it good enough?" (as in "Will it not crash?").

Of course not everyone using Eiffel can, any more than in another approach, guarantee that the ideal just presented will always hold. But it is the theoretical scheme to which the method tends. It explains Eiffel's emphasis on getting everything right: the grandiose and the mundane, the structure and the details. Regarding the details, the Eiffel books cited in the bibliography include many rules, some petty at first sight, about such low-level aspects as the choice of names for classes and features (including their grammatical categories), the indentation of software texts, the style for comments (including the presence or absence of a final period), the use of spaces. Applying these rules does not, of course, guarantee quality; but they are part of a quality-oriented process,

along with the more ambitious principles of design. In addition they are particularly important for the construction of quality libraries, one of the central goals of Eiffel.

Whenever they are compatible with the space constraints, the present chapter and the rest of this book apply these rules to their Eiffel examples.

4 HELLO WORLD

When discovering any approach to software construction, however ambitious its goals, it is reassuring to see first a small example of the big picture — a complete program to print the famous “Hello World” string. Here is how to perform this fascinating task in the Eiffel notation.

You write a class *HELLO* with a single procedure, say *make*, also serving as creation procedure. If you like short texts, here is a minimal version:

```
class HELLO create make feature
    make is
        do print ("Hello World%N") end
end
```

In practice, however, the Eiffel style rules suggest a better documented version:

```
indexing
    description: "Root for trivial system printing a message"
    author: "Elizabeth W. Brown"
class HELLO create
    make
feature
    make is
            -- Print a simple message.
        do
            io.put_string ("Hello World")
            io.put_new_line
        end
end -- class HELLO
```

The two versions perform identically; the following comments will cover the more complete second one.

Note the absence of semicolons and other syntactic clatter or clutter. You may in fact use semicolons to separate instructions and declarations. But the language’s syntax

is designed to make the semicolon optional (regardless of text layout) and it's best for readability to omit it, except in the special case of successive elements on a single line.

The **indexing** clause does not affect execution semantics; you may use it to associate documentation with the class, so that browsers and other indexing and retrieval tools can help users in search of reusable components satisfying certain properties. Here we see two indexing entries, labeled *description* and *author*.

The name of the class is *HELLO*. Any class may contain "features"; *HELLO* has just one, called *make*. The **create** clause indicates that *make* is a "creation procedure", that is to say an operation to be executed at class instantiation time. The class could have any number of creation procedures.

The definition of *make* appears in a **feature** clause. There may be any number of such clauses (to separate features into logical categories), and each may contain any number of feature declarations. Here we have only one.

The line starting with -- (two hyphen signs) is a comment; more precisely it is a "header comment", which style rules invite software developers to write for every such feature, just after the **is**. As will be seen in "The contract form of a class", page 44, the tools of EiffelStudio know about this convention and use it to include the header comment in the automatically generated class documentation.

The body of the feature is introduced by the **do** keyword and terminated by **end**. It consists of two output instructions. They both use *io*, a generally available reference to an object that provides access to standard input and output mechanisms; the notation *io.f*, for some feature *f* of the corresponding library class (*STD_FILES*), means "apply *f* to *io*". Here we use two such features:

- *put_string* outputs a string, passed as argument, here "*Hello World*".
- *put_new_line* terminates the line.

Rather than using a call to *put_new_line*, the first version of the class simply includes a new-line character, denoted as *%N*, at the end of the string. Either technique is acceptable.

To build the system and execute it:

- Start EiffelStudio
- When prompted, ask EiffelStudio to build a system for you; specify *HELLO* as the "root class" and *make* as the "root procedure".
- You can either use EiffelStudio to type in the above class text, or you may use any text editor and store the result into a file **hello.e** in the current directory.
- Click the "Compile" icon.
- Click the "Run" icon.

Execution starts and outputs *Hello World* on the appropriate medium: under Windows, a Console; under Unix or VMS, the windows from which you started EiffelStudio.

5 THE STATIC PICTURE: SYSTEM ORGANIZATION

We now look at the overall organization of Eiffel software.

References to ISE-originated libraries appearing in subsequent examples include: **EiffelBase**, the fundamental open-source library covering data structures and algorithms; the **kernel library**, a subset of EiffelBase covering the most basic notions such as arrays and strings; and **EiffelVision 2**, an advanced graphics and GUI library providing full compatibility across platforms (Unix, Windows, VMS) with native look-and-feel on each.

Systems

An Eiffel system is a collection of classes, one of which is designated as the root class. One of the features of the root class, which must be one of its creation procedures, is designated as the root procedure.

To execute such a system is to create an instance of the root class (an object created according to the class description) and to execute the root procedure. In anything more significant than “Hello World” systems, this will create new objects and apply features to them, in turn triggering further creations and feature calls.

For the system to make sense, it must contains all the classes on which the root **depends** directly or indirectly. A class *B* depends on a class *A* if it is either a **client** of *A*, that is to say uses objects of type *A*, or an **heir** of *A*, that is to say extends or specializes *A*. (These two relations, client and inheritance, are covered below.)

Classes

The notion of class is central to the Eiffel approach. A class is the description of a type of run-time data structures (*objects*), characterized by common operations (*features*) and properties. Examples of classes include:

- In a banking system, a class *ACCOUNT* may have features such as *deposit*, adding a certain amount to an account, *all_deposits*, yielding the list of deposits since the account’s opening, and *balance*, yielding the current balance, with properties stating that *deposit* must add an element to the *all_deposits* list and update *balance* by adding the sum deposited, and that the current value of *balance* must be consistent with the lists of deposits and withdrawals.
- A class *COMMAND* in an interactive system of any kind may have features such as *execute* and *undo*, as well as a feature *undoable* which indicates whether a command can be undone, with the property that *undo* is only applicable if *undoable* yields the value true.

- A class *LINKED_LIST* may have features such as *put*, which adds an element to a list, and *count*, yielding the number of elements in the list, with properties stating that *put* increases *count* by one and that *count* is always non-negative.

We may characterize the first of these examples as an analysis class, directly modeling objects from the application domain; the second one as a design class, describing a high-level solution; and the third as an implementation class, reused whenever possible from a library such as EiffelBase. In Eiffel, however, there is no strict distinction between these categories; it is part of the approach's seamlessness that the same notion of class, and the associated concepts, may be used at all levels of the software development process.

Class relations

Two relations may exist between classes:

- You can define a class *C* as a **client** of a class *A* to enable the features of *C* to rely on objects of type *A*.
- You may define a class *B* as an **heir** of a class *A* to provide *B* with all the features and properties of *A*, letting *B* add its own features and properties and modify some of the inherited features if appropriate.

If *C* is a client of *A*, *A* is a **supplier** of *C*. If *B* is an heir of *A*, *A* is a **parent** of *B*. A **descendant** of *A* is either *A* itself or, recursively, a descendant of an heir of *A*; in more informal terms a descendant is a direct or indirect heir, or the class itself. To exclude *A* itself we talk of **proper descendant**. In the reverse direction the terms are **ancestor** and **proper ancestor**.

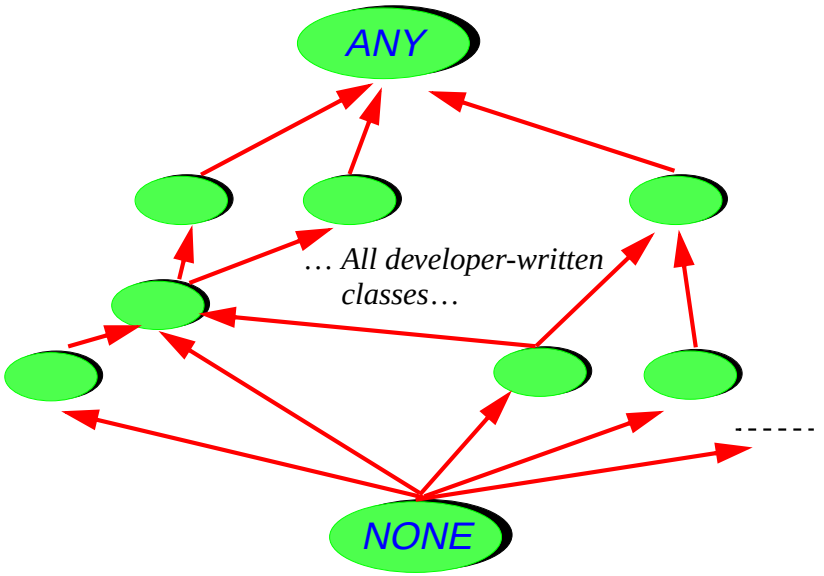
The client relation can be cyclic; an example involving a cycle would be classes *PERSON* and *HOUSE*, modeling the corresponding informal everyday "object" types and expressing the properties that every person has a home and every home has an architect. The inheritance (heir) relation may not include any cycle.

In modeling terms, client roughly represents the relation "has" and heir roughly represents "is". For example we may use Eiffel classes to model a certain system and express that every child *has* a birth date (client relation) and *is* a person (inheritance).

Distinctive of Eiffel is the rule that classes can only be connected through these two relations. This excludes the behind-the-scenes dependencies often found in other approaches, such as the use of global variables, which jeopardize the modularity of a system. Only through a strict policy of limited and explicit inter-class relations can we achieve the goals of reusability and extendibility.

The global inheritance structure

An Eiffel class that you write does not come into a vacuum but fits in a preordained structure, shown in the figure and involving two library classes: *ANY* and *NONE*.



Global inheritance structure

Any class that does not explicitly inherit from another is considered to inherit from *ANY*, so that every class is a descendant, direct or indirect, of *ANY*. *ANY* introduces a number of general-purpose features useful everywhere, such as copying, cloning and equality testing operations (page 28) and default input-output. The procedure *print* used in the first version of our "Hello World" (page 11) comes from *ANY*.

NONE inherits from any class that has no explicit heir. Since inheritance has no cycles, *NONE* cannot have proper descendants. This makes it useful, as we will see, to specify non-exported features, and to denote the type of void values. Unlike *ANY*, class *NONE* doesn't have an actual class text; instead, it's a convenient fiction.

Clusters

Classes are the only form of module in Eiffel. As will be explained in more detail, they also provide the basis for the only form of type. This module-type identification is at the heart of object technology and of the fundamental simplicity of the Eiffel method.

Above classes, you will find the concept of cluster. A cluster is a group of related classes. Clusters are a property of the method, enabling managers to organize the development into teams. As we have already seen (section 3) they also play a central role in the lifecycle model. Clusters are an organizational concept, not a form of module, and do not require an Eiffel language construct.

External software

The subsequent sections will show how to write Eiffel classes with their features. In an Eiffel system, however, not everything has to be written in Eiffel: some features may be **external**, coming from languages such as C, C++, Java, C# Fortran and others. For example a feature declaration may appear (in lieu of the forms seen later) as

```
file_status (filedesc: INTEGER): INTEGER is
        -- Status indicator for filedesc
    external
        "C" alias "_fstat"
    end
```

to indicate that it is actually an encapsulation of a C function whose original name is *_fstat*. The **alias** clause is optional, but here it is needed because the C name, starting with an underscore, is not valid as an Eiffel identifier.

Similar syntax exists to interface with C++ classes. ISE Eiffel includes a tool called *Legacy++* which will automatically produce, from a C++ class, an Eiffel class that encapsulates its facilities, making them available to the rest of the Eiffel software as *bona fide* Eiffel features.

These mechanisms illustrate one of the roles of Eiffel: as an system architecturing and software composition tool, used at the highest level to produce systems with robust, flexible structures ready for extendibility, reusability and maintainability. In these structures not everything must be written in the Eiffel language: existing software elements and library components can play their part, with the structuring capabilities of Eiffel (classes, information hiding, inheritance, clusters, contracts and other techniques seen in this presentation) serving as the overall wrapping mechanism.

6 THE DYNAMIC STRUCTURE: EXECUTION MODEL

A system with a certain static structure describes a set of possible executions. The run-time model governs the structure of the data (*objects*) created during such executions.

The properties of the run-time model are not just of interest to implementers; they also involve concepts directly relevant to the needs of system modelers and analysts at the most abstract levels.

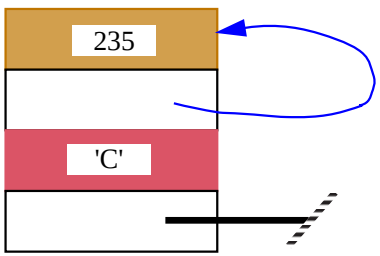
Objects, fields, values and references

A class was defined as the static description of a a type of run-time data structures. The data structures described by a class are called **instances** of the class, which in turn is called their **generating class** (or just "*generator*"). An instance of *ACCOUNT* is a data structure representing a bank account; an instance of *LINKED_LIST* is a data structure representing a linked list.

An **object**, as may be created during the execution of a system, is an instance of some class of the system.

Classes and objects belong to different worlds: a class is an element of the software text; an object is a data structure created during execution. Although is possible to define a class whose instances represent classes (as class *E_CLASS* in the ISE libraries, used to access properties of classes at run time), this does not eliminate the distinction between a static, compile-time notion, class, and a dynamic, run-time notion, object.

An object is either an atomic object (integer, real, boolean, double) or a composite object made of a number of **fields**, represented by adjacent rectangles on the conventional run-time diagrams:



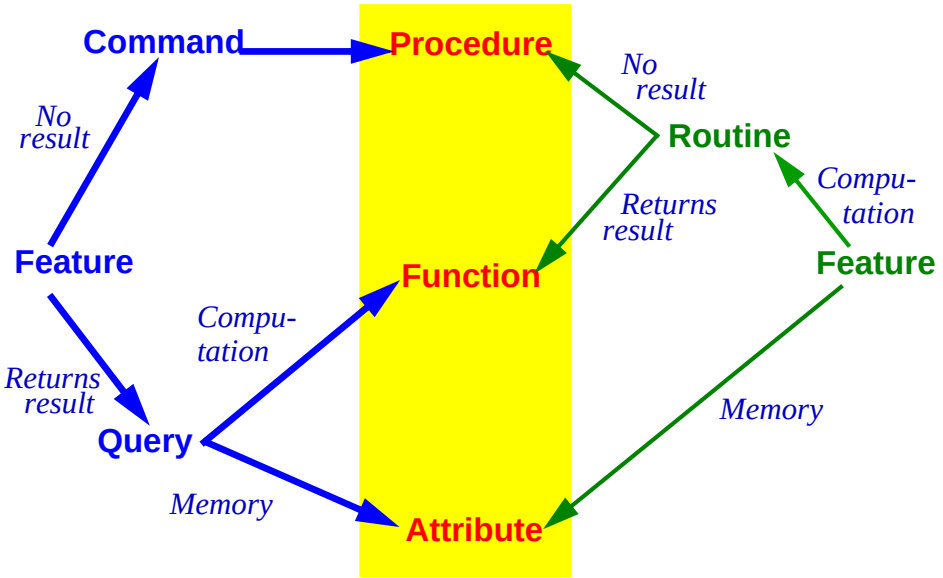
Composite object

(with 4 fields including self-reference and void reference)

Each field is a **value**. A value can be either an object or an object reference:

- When a field is an object, it will in most cases be an atomic object, as on the figure where the first field from the top is an integer and the third a character. But a field can also be a composite object, in which case it is called a **subobject**.
- A **reference** is either void or uniquely identifies an object, to which it is said to be **attached**. In the preceding figure the second field from the top is a reference — attached in this case, as represented by the arrow, to the enclosing object itself. The bottom field is a void reference.

Features



A feature, as noted, is an operation available on instances of a class. A feature can be either an **attribute** or a **routine**. This classification, which you can follow by starting from the *right* on the figure above, is based on implementation considerations:

- An attribute is a feature implemented through memory: it describes a field that will be found in all instances of the class. For example class *ACCOUNT* may have an attribute *balance*; then all instances of the class will have a corresponding field containing each account's current balance.
- A routine describes a computation applicable to all instances of the class. *ACCOUNT* may have a routine *withdraw*.
- Routines are further classified into **functions**, which will return a result, and **procedures**, which will not. Routine *withdraw* will be a procedure; an example of function may be *highest_deposit*, which returns the highest deposit made so far to the account.

If we instead take the viewpoint of the **clients** of a class (the classes relying on its feature), you can see the relevant classification by starting from the *left* on the figure:

- **Commands** have no result, and may modify an object. They may only be procedures.
- **Queries** have a result: they return information about an object. You may implement a query as either an attribute (by reserving space for the corresponding information in each instance of the class, a memory-based solution) or a function (a computation-based solution). An attribute is only possible for a query without

argument, such as *balance*; a query with arguments, such as *balance_on* (*d*), returning the balance at date *d*, can only be a function.

From the outside, there is no difference between a query implemented as an attribute and one implemented as a function: to obtain the balance of an account *a*, you will always write *a.balance*. In the implementation suggested above, *a* is an attribute, so that the notation denotes an access to the corresponding object field. But it is also possible to implement *a* as a function, whose algorithm will explore the lists of deposits and withdrawals and compute their accumulated value. To the clients of the class, and in the official class documentation as produced by the environment tools, the difference is not visible.

This principle of **Uniform Access** is central to Eiffel's goals of extendibility, reusability and maintainability: you can change the implementation without affecting clients; and you can reuse a class without having to know the details of its features' implementations. Most object-oriented languages force clients to use a different notation for a function call and an attribute access. This violates Uniform Access and is an impediment to software evolution, turning internal representation changes into interface changes that may disrupt large parts of a system.

A simple class

The following simple class text illustrates the preceding concepts

```
indexing
        description: "Simple bank accounts"
class
        ACCOUNT
feature -- Access
        balance: INTEGER
                        -- Current balance
        deposit_count: INTEGER is
                        -- Number of deposits made since opening
                do
                        if all_deposits /= Void then
                                Result := all_deposits.count
                        end
                end
```

```
feature -- Element change
        deposit (sum: INTEGER) is
                -- Add sum to account.
            do
                if all_deposits= Void then
                    create all_deposits
                end
                all_deposits.extend (sum)
                balance := balance + sum
            end

feature {NONE} -- Implementation
        all_deposits: DEPOSIT_LIST
                -- List of deposits since account's opening.
invariant
        consistent_balance:
        (all_deposits /= Void) implies (balance = all_deposits.total)
        zero_if_no_deposits:
            (all_deposits = Void) implies (balance = 0)
end -- class ACCOUNT
```

(The {*NONE*} qualifier and the **invariant** clause, used here to make the example closer to a real class, will be explained shortly. *DEPOSIT_LIST* refers to another class, which can be written separately using library classes.)

It's easy to deduce, from a feature's syntactic appearance, the category to which it belongs. Here:

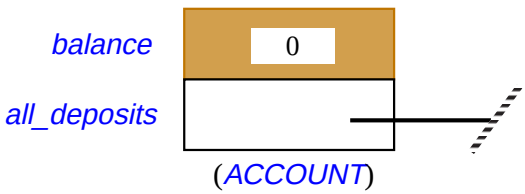
- Only *deposit* and *deposit_count*, which include a **do** … clause, are routines.
- *balance* and *all_deposits*, which are simply declared with a type, are attributes. Note that even for attributes it is recommended to have a header comment.
- Routine *deposit_count* is declared as returning a result (of type *INTEGER*); so it is a function. Routine *deposit* has no such result and hence is a procedure.

Creating and initializing objects

Classes, as noted, are a static notion. Objects appear at run time; they are created explicitly. Here is the basic instruction to create an object of type *ACCOUNT* and attach it to *x*:

```
create x
```

assuming that *x* has been declared of type *ACCOUNT*. Such an instruction must be in a routine of some class — the only place where instructions can appear — and its effect at run time will be threefold: create a new object of type *ACCOUNT*; initialize its fields to default values; and attach the value of *x* to it. Here the object will have two fields corresponding to the two attributes of the generating class: an integer for *balance*, which will be initialized to 0, and a reference for *all_deposits*, which will be initialized to a void reference:



Instance with fields initialized to defaults

The language specifies default initialization values for all possible types:

Type	Default value
INTEGER, *REAL*, *DOUBLE*	Zero
BOOLEAN	False
CHARACTER	Null
Reference types (such as *ACCOUNT* and *DEPOSIT_LIST*)	Void reference
Composite expanded types (see next)	Same rules, applied recursively to all fields

It is possible to override the initialization values by providing — as in the earlier example of class *HELLO* — one or more creation procedures. For example we might change *ACCOUNT* to make sure that every account is created with an initial deposit:

```
indexing
        description: "Simple bank accounts, initialized with a first deposit"
class
        ACCOUNT1
create
        make
feature -- Initialization
                make (sum: INTEGER) is
                        -- Initialize account with sum.
                do
                        deposit (sum)
                end
… The rest of the class as for ACCOUNT …
end -- class ACCOUNT1
```

A **create** clause may list zero or more (here just one) procedures of the class.

Note the use of the same keyword, **create**, for both a creation clause, as here, and creation instructions such as **creat** *x*.

In this case the original form of creation instruction, **create** *x*, is not valid any more for creating an instance of *ACCOUNT1*; you must use the form

```
create x.make (2000)
```

known as a creation call. Such a creation call will have the same effect as the original form — creation, initialization, attachment to *x* — followed by the effect of calling the selected creation procedure, which here will call *deposit* with the given argument.

Note that in this example all that *make* does is to call *deposit*. So an alternative to introducing a new procedure *make* would have been simply to introduce a creation clause of the form **create** *deposit*, elevating *deposit* to the status of creation procedure. Then a creation call would be of the form **create** *x*.*deposit* (*2000*).

Some variants of the basic creation instruction will be reviewed later: instruction with an explicit type; creation expressions. See "Creation variants", page 89.

Entities

The example assumed *x* declared of type *ACCOUNT* (or *ACCOUNT1*). Such an *x* is an example of **entity**, a notion generalizing the well-known concept of variable. An entity is a name that appears in a class text to represent possible run-time values (a value being, as defined earlier, an object or a reference). An entity is one of the following:

- An attribute of the enclosing class, such as *balance* and *all_deposits*.
- A formal argument of a routine, such as *sum* for *deposit* and *make*.

- A local entity declared for the internal needs of a routine.
- The special entity *Result* in a function.

The third case, local entities, arises when a routine needs some auxiliary values for its computation. Here is an example of the syntax:

```
deposit (sum: INTEGER) is
        -- Add sum to account.
    local
        new: AMOUNT
    do
        create new.make (sum)
        all_deposits.extend (new)
        balance := balance + sum
    end
```

This example is a variant of *deposit* for which we assume that the elements of a *DEPOSIT_LIST* such as *all_deposits* are no longer just integers, but objects, instances of a new class, *AMOUNT*. Such an object will contain an integer value, but possibly other information as well. So for the purpose of procedure *deposit* we create an instance of *AMOUNT* and insert it, using procedure *extend*, into the list *all_deposits*. The object is identified through the local entity *new*, which is only needed within each execution of the routine (as opposed to an attribute, which yields an object field that will remain in existence for as long as the object).

The last case of entity, *Result*, serves to denote, within the body of a function, the final result to be returned by that function. This was illustrated by the function *deposits_count*, which read

```
deposit_count: INTEGER is
        -- Number of deposits made since opening (provisional version)
        if all_deposits /= Void then
            Result := all_deposits.count
        end
```

The value returned by any call will be the value of the expression *all_deposits.count* (to be explained in detail shortly) for that call, unless *all_deposits* has value *Void*, denoting a void reference (*/=* is “not equal”).

The default initialization rules seen earlier for attributes (see the table on page 21) also serve to initialize local entities and *Result* on routine entry. So in the last example, if *all_deposits* is void (as in the case on initialization with the class as given so far), *Result* keeps its default value of 0, which will be returned as the result of the function.

Calls

Apart from object creation, the basic computational mechanism, in the object-oriented style of computation represented by Eiffel, is feature call. In its basic form, it appears as

```
target.feature (argument1, …)
```

where *target* is an entity or more generally an expression, *feature* is a feature name, and there may be zero or more *argument* expressions. In the absence of any *argument* the part in parentheses should be removed.

We have already seen such calls. If the *feature* denotes a procedure, the call is an instruction, as in

```
all_deposits.extend (new)
```

If *feature* denotes a query (function or attribute), the call is an expression, as in the right-hand side of

```
Result := all_deposits.count
```

Following the principle of Uniform Access (page 19), this form is the same for calls to attributes and to functions without arguments. In this example, feature *count* from class *DEPOSIT_LIST* may indeed be implemented in either of these two ways: we can keep a *count* field in each list, updating it for each insertion and removal; or we can compute *count*, whenever requested, by traversing the list and counting the number of items.

In the case of a routine with arguments — procedure or function — the routine will be declared, in its class, as

```
feature ( formal1: TYPE1; …) is
    do … end
```

meaning that, at the time of each call, the value of each formal will be set to the corresponding actual (*formal1* to *argument1* and so on).

In the routine body, it is not permitted to change the value of a formal argument, although it is possible to change the value of an attached object through a procedure call such as *formal1.some_procedure* (…).

Infix and prefix notation

Basic types such as *INTEGER* are, as noted, full-status citizens of Eiffel's type system, and so are declared as classes (part of the Kernel Library). *INTEGER*, for example, is characterized by the features describing integer operations: plus, minus, times, division, less than, and so on.

With the dot notation seen so far, this would imply that simple arithmetic operations would have to be written with a syntax such as *i.plus (j)* instead of the usual *i + j*. This would be awkward. Infix and prefix features solve the problem, reconciling the object-oriented view of computation with common notational practices of mathematics. The addition function is declared in class *INTEGER* as

```
infix "+" (other: INTEGER): INTEGER is
        do … end
```

Such a feature has all the properties and prerogatives of a normal "identifier" feature, except for the form of the calls, which is infix, as in *i + j*, rather than using dot notation. An infix feature must be a function, and take exactly one argument. Similarly, a function can be declared as **prefix** "–", with no argument, permitting calls of the form *–3* rather than *(3).negated*.

Predefined library classes covering basic types such as *INTEGER*, *CHARACTER*, *BOOLEAN*, *REAL*, *DOUBLE* are known to the Eiffel compiler, so that a call of the form *i + j*, although conceptually equivalent to a routine call, can be processed just as efficiently as the corresponding arithmetic expression in an ordinary programming language. This brings the best of both worlds: conceptual simplicity, enabling Eiffel developers, when they want to, to think of integers and the like as objects; and efficiency as good as in lower-level approaches.

Infix and prefix features are available to any class, not just the basic types' predefined classes. For example a graphics class could use the name **infix** "|–|" for a function computing the distance between two points, to be used in expressions such as *point1 |–| point2*.

Type declaration

Every entity appearing in an Eiffel text is declared as being of a certain type, using the syntax already encountered in the above examples:

```
entity_name: TYPE_NAME
```

This applies to attributes, formal arguments of routines and local entities. You will also declare the result type for a function, as in the earlier example

```
deposit_count: INTEGER is …
```

Specifying such a function result type also declares, implicitly, the type for *Result* as used in the function's body.

What is a type? With the elements seen so far, every type is a **class**. *INTEGER*, used in the declaration of *deposits_count*, is, as we have seen, a library class; and the declaration *all_deposits*: *DEPOSIT_LIST* assumes the existence of a class *DEPOSIT_LIST*.

Three mechanisms introduced below — expanded types (page 26), genericity (page 36) and anchored declarations (page 79)— will generalize the notion of type slightly. But they do not change the fundamental property that **every type is based on a class**, called the type's **base class**. In the examples seen so far, each type *is* a class, serving as its own base class.

An instance of a class *C* is also called "an object of type *C*".

Type categories

It was noted above that a value is either an object or a reference. This corresponds to two kinds of type: reference types and expanded types.

If a class is declared as just

```
class CLASS_NAME …
```

it defines a reference type. The entities declared of that type will denote references. So in the declaration

```
x: ACCOUNT
```

the possible run-time values for *x* are references, which will be either void or attached to instances of class *ACCOUNT*.

Instead of **class**, however, you may use the double keyword **expanded class**, as in the EiffelBase class definition

```
indexing
        description: "Integer values"
expanded class
        INTEGER
feature -- Basic operations
        infix "+" (other: INTEGER): INTEGER is
                do … end
        … Other feature declarations …
end -- class INTEGER
```

In this case the value of an entity declared as *n*: *INTEGER* is not a reference to an object, but the object itself — in this case an atomic object, an integer value.

It is also possible, for some non-expanded class C, to declare an entity as

```
x: expanded C
```

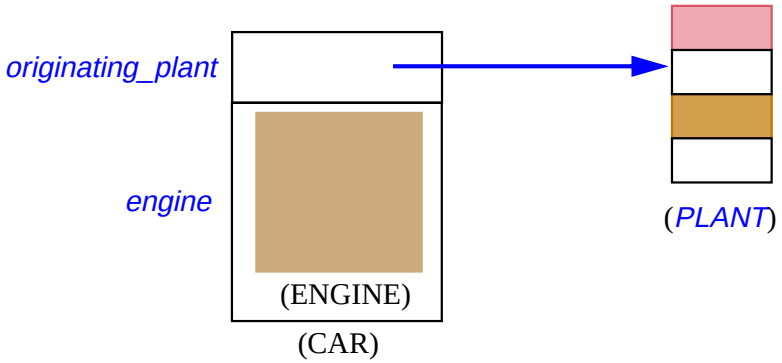
so that the values for *x* will be objects of type *C*, rather than references to such objects. This is our first example of a type — **expanded** *C* — that is not directly a class, although it is based on a class, *C*. The base type of such a type is *C*.

Note that the value of an entity of an expanded type can never be void; only a reference can. Extending the earlier terminology, an expanded entity is always **attached to** an object, atomic (as in the case of *n*: *INTEGER*) or composite (as in *x*: **expanded** *ACCOUNT*).

Expanded declarations make it possible to construct composite objects with subobjects, as in the following abbreviated class declaration (indexing clause and routines omitted):

```
class CAR feature
        engine: expanded ENGINE
        originating_plant: PLANT
end -- class CAR
```

Here is an illustration of the structure of a typical instance of *CAR*:



Composite object with reference and subobject

This example also illustrates that the distinction between expanded and reference types is important not just for system implementation purposes but for high-level system modeling as well. Consider the example of a class covering the notion of car. Many cars share the same *originating_plant*, but an *engine* belongs to just one car. References represent the modeling relation "knows about"; subobjects, as permitted by expanded types, represent the relation "has part", also known as aggregation. The key difference is that sharing is possible in the former case but not in the latter.

Basic operations

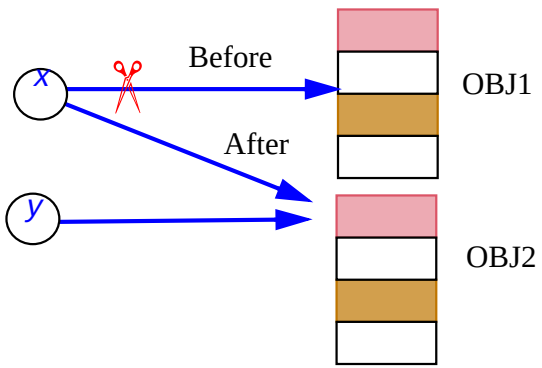
To assign, copy and compare values, you can rely on a number of mechanisms. Two of them, assignment and equality testing, are language constructs; the others are library features, coming from the top-level class *ANY* seen earlier (page 15).

Assignment uses the symbol :=. The assignment instruction

```
x := y
```

updates the value of *x* to be the same as that of *y*. This means that:

- For entities of reference types, the value of *x* will be a void reference if the value of *y* is void, and otherwise *x* will be attached to the same object OBJ2 as *y*:



Effect of reference reattachment
x := y

- For entities of expanded types, the values are objects; the object attached to *x* will be overwritten with the contents of the object attached to *y*. In the case of atomic objects, as in *n := 3* with the declaration *n*: *INTEGER*, this has the expected effect of assigning to *n* the integer value *3*; in the case of composite objects, this overwrites the fields for *x*, one by one, with the corresponding *y* fields.

To copy an object, use *x.copy (y)* which assumes that both *x* and *y* are non-void, and copies the contents of *y*'s attached object onto those of *x*'s. For expanded entities the effect is the same as that the of the assignment *x := y*.

A variant of the *copy* operation is *clone*. The expression *clone (y)* produces a newly created object, initialized with a copy of the object attached to *y*, or a void value if *y* itself is void. For a reference type (the only interesting case) the returned result for non-void *y* is a reference to the new object. This means we may view *clone* as a function that performs

```
create Result
Result.copy (y)
```

So in the assignment *x* := *clone (y)*, assuming both entities of reference types and *y* not void, will attach *x* to a **new object** identical to *y*'s attached object, as opposed to the assignment *x := y* which attaches *x* to the **same object** as *y*.

To determine whether two values are equal, use the expression *x = y*. For references, this comparison will yield true if the values are either both void or both attached to the same object; this is the case in the last figure in the state after the assignment, but not before. The symbol for *not* equal is */=*, as in *x /= y*.

As with assignment, there is also a form that works on objects rather than references: *x.is_equal (y)* will return true when *x* and *y* are both non-void and attached to field-by-field identical objects. This can be true even when *x = y* is not, for example, in the figure, *before* the assignment, if the two objects shown are field-by-field equal.

A more general variant of *is_equal* is used under the form *equal (x, y)*. This is always defined, even if *x* is void, returning true whenever *is_equal* would but also if *x* and *y* are both void. (In contrast, *x.is_equal (y)* is not defined for void *x* and would, if evaluated, yield an exception as explained in "Exception handling", page 46 below.)

Void denotes a void reference. So you can make *x* void through the assignment *x := Void*, and test whether it is void through **if** *x = Void* **then** …

Where assignment := and the equality operators = and */=* were language constructres, *copy*, *clone*, *is_equal*, *equal* and *Void* are **library features** coming from class *ANY*. The type of *Void*, as declared in *ANY*, is *NONE*, the "bottom" type.

Using the redefinition mechanisms to be seen in the discussion of inheritance, a class can redefine *copy* and *is_equal* to cover specific notions of copy and equality. The assertions will ensure that the two remain compatible: after *x.copy* (*y*), the property *x.is_equal* (*y*) must always be true. The effect of *clone* will automatically follow a redefinition of *copy*, and *equal* will follow *is_equal*.

To guarantee the original, non-redefined semantics you may use the variants *standard_copy*, *standard_clone*, *standard_equal*, all defined in *ANY* as "frozen", that is to say non-redefinable.

Deep operations and persistence

Feature *clone* only duplicates one object. If some of the object's fields are references to other objects, the references themselves will be copied, not those other objects.

It is useful, in some cases, to duplicate not just one object but an entire object structure. The expression *deep_clone* (*y*) achieves this goal: assuming non-void *y*, it will produce a duplicate not just of the object attached to *y* but of the entire object structure starting at that object. The mechanism respects all the possible details of that structure, such as cyclic reference chains. Like the preceding features, *deep_clone* comes from class *ANY*.

A related mechanism provides a powerful **persistence** facility. A call of the form

```
x.store (Some_file_or_network_connection)
```

will store a copy of the entire object structure starting at *x*, under a suitable representation. Like *deep_clone*, procedure *store* will follow all references to the end and maintain the properties of the structure. The function *retrieved* can then be used — in the same system, or another — to recreate the structure from the stored version.

As the name suggests, *Some_file_or_network_connection* can be an external medium of various possible kinds, not just a file but possibly a database or network. ISE's EiffelNet client-server library indeed uses the *store*-*retrieved* mechanism to exchange object structures over a network, between compatible or different machine architectures, for example a Windows client and a Unix server.

Memory management

Reference reattachments *x* := *y* of the form illustrated by the figure on page 28 can cause objects to become unreachable. This is the case for the object identified as OBJ2 on that figure (the object to which *x* was attached before the assignment) if no other reference was attached to it.

In all but toy systems, it is essential to reclaim the memory that has been allocated for such objects; otherwise memory usage could grow forever, as a result of creation instructions **create** *x* … and calls to *clone* and the like, leading to thrashing and eventually to catastrophic termination.

Tthe Eiffel method suggests that the task of detecting and reclaiming such unused object space should be handled by an automatic mechanism (part of the Eiffel run-time environment), not manually by developers (through calls to procedures such as Pascal's *dispose* and C/C++'s *free*). The arguments for this view are:

- **Simplicity**: handling memory reclamation manually can add enormous complication to the software, especially when — as is often the case in object-oriented development — the system manipulates complex run-time data structures with many links and cycles.
- **Reliability**: memory management errors, such as the incorrect reclamation of an object that is still referenced by a distant part of the structure, are a notorious source of dangerous and hard-to-correct bugs.

ISE Eiffel provides a sophisticated **garbage collector** which efficiently handles the automatic reclamation process, while causing no visible degradation of a system's performance and response time.

Information hiding and the call rule

The basic form of computation, it has been noted, is a call of the form *target*.*feature* (…). This is only meaningful if *feature* denotes a feature of the generating class of the object to which *target* (assumed to be non-void) is attached. The precise rule is the following:

Feature Call rule

A call of the form *target*.*feature* (…) appearing in a class *C* is only valid if *feature* is a feature of the base class of *target*'s type, and is available to *C*.

The first condition simply expresses that if *target* has been declared as *target*: *A* then *feature* must be the name of one of the features of *A*. The second condition reflects Eiffel's application of the principles of information hiding. A **feature** clause, introducing one or more feature declarations, may appear not only as

```
feature -- Comment identifying the feature category
    … Feature declaration …
    … Feature declaration …
    …
```

but may also include a list of classes in braces, **feature** {*A*, *B*, …}, as was illustrated for *ACCOUNT*:

```
feature {NONE} -- Implementation
    all_deposits: DEPOSIT_LIST
            -- List of deposits since account's opening.
```

This form indicates that the features appearing in that clause are only **available** — in the sense of available for calls, as used in the Feature Call rule — to the classes listed. In the example feature *all_deposits* is only available to *NONE*. Because of the global inheritance structure (page 15) this means it is in fact available to no useful client at all, and is equivalent in practice to **feature** { } with an empty class list, although the form listing *NONE* explicitly is more visible and hence preferred.

With this specification a class text including the declaration *acc*: *ACCOUNT* and a call of the form

```
acc.all_deposits
```

violates the Feature Call rule and will be rejected by the EiffelStudio compiler.

Besides fully exported features (introduced by **feature** … without further qualification) and fully secret ones (**feature** { } or **feature** {*NONE*}), it is possible to export features selectively to some specified classes, using the specification

```
feature {A, B, …}
```

for arbitrary classes *A*, *B*, … This enables a group of related classes to provide each other with privileged access, without requiring the introduction of a special module category above the class level (see "Clusters", page 15).

Exporting features selectively to a set of classes *A*, *B*, … also makes them available to the descendants of these classes. So a feature clause beginning with just **feature** is equivalent to one starting with **feature** {*ANY*}.

These rules enable successive feature clauses to specify exports to different clients. In addition, the recommended style, illustrated in the examples of this chapter, suggests writing separate feature clauses — regardless of their use for specifying export privileges — to group features into separate categories. The standard style rules define a number of fundamental categories and the order in which they should appear; they include: Initialization for creation procedures, Access for general queries, Status report for boolean-valued queries, Status setting, Element change, Implementation (for selectively exported or secret features. Every feature in the EiffelBase library classes belongs to one of the predefined categories.

The Feature Call rule is the first of the rules that make Eiffel a **statically typed** approach, where the applicability of operations to objects is verified at compile time rather than during execution. Static typing is one of the principal components of Eiffel's support for reliability in software development.

Execution scenario

The preceding elements make it possible to understand the overall scheme of an Eiffel system's execution.

At any time during the execution of a system, one object is the **current object** of the execution, and one of the routines of the system, the **current routine**, is being executed, with the current object as its target. (We will see below how the current object and current routine are determined.) The text of a class, in particular its routines, make constant implicit references to the current object. For example in the instruction

```
balance := balance + sum
```

appearing in the body of procedure *deposit* of class *ACCOUNT*, the name of the attribute *balance*, in both occurrences, denotes the *balance* field of the current object, assumed to be an instance of *ACCOUNT*. In the same way, the procedure body that we used for the creation procedure *make* in the *ACCOUNT1* variant

```
make (sum: INTEGER) is
            -- Initialize account with sum.
      do
            deposit (sum)
      end
```

contains a call to the procedure *deposit*. Contrary to earlier calls written in dot notation as *target*.*feature* (…), the call to *deposit* has no explicit target; this means its target is the current object, an instance of *ACCOUNT1*. Such a call is said to be **unqualified**; those using dot notations are **qualified** calls.

Although most uses of the current object are implicit, a class may need to name it explicitly. The predefined expression *Current* is available for that purpose. A typical use, in a routine *merge* (*other*: *ACCOUNT*) of class *ACCOUNT*, would be a test of the form

```
if other = Current then
      report_error ("Error: trying to merge an account with itself!")
else
      … Normal processing (merging two different accounts) …
end
```

With these notions it is not hard to define precisely the overall scenario of a system execution by defining which object and routine will, at each instant, be the current object and the current routine:

- Starting a system execution, as we have seen, consists in creating an instance of the root class, the root object, and executing a designated creation procedure, the root procedure, with the root object as its target. The root object is the initial current object, and the root procedure is the initial current procedure.

- From then on only two events can change the current object and current procedure: a qualified routine call; and the termination of a routine.

- In a call of the form *target*.*routine* (…), *target* denotes a certain object TC. (If not, that is to say, if the value of target is void, attempting to execute the call will trigger an exception, as studied below.) The generating class of TC must, as per the Feature Call rule, contain a routine of name *routine*. As the call starts, TC becomes the new current object and *routine* becomes the new current routine.

- When a routine execution terminates, the target object and routine of the most recent non-terminated call — which, just before just before the terminated call, were the current object and the current routine — assume again the role of current object and current routine.

- The only exception to the last rule is termination of the original root procedure call; in this case the entire execution terminates.

Abstraction

The description of assignments stated that in *x* := *y* the target *x* must be an entity. More precisely it must be a **writable** entity. This notion excludes formal routine arguments: as noted, a routine *r* (*arg*: *SOME_TYPE*) may assign to *arg* (reattaching it to a different object), although it can change the attached objects through calls of the form *arg*.*procedure* (…).

Restricting assignment targets to entities precludes assignments of the form *obj*.*some_attribute* := *some_value*, since the left-hand side *obj*.*some_attribute* is an expression (a feature call), not an entity: you may no more assign to *obj*.*some_attribute* than to, say, *a* + *b* — another expression that is also, formally, a feature call.

To obtain the intended effect of such an assignment you may use a procedure call of the form *obj*.*set_attribute* (*some_value*), where the base class of *obj*'s type has defined the procedure

```
set_attribute (v: VALUE_TYPE) is
        -- Set value of attribute to v.
    do
        attribute := v
    end
```

This rule is essential to enforcing the method. Permitting direct assignments to an object's fields — as in C++ and Java — would violate all the tenets of information hiding by letting clients circumvent the interface carefully crafted by the author of a supplier class. It is the responsibility of each class author to define the exact privileges that the class gives to each of its clients, in particular field modification rights. Building a class is like building a machine: you design the internals, to give yourself the appropriate mechanisms; and you design the control panel, letting users (clients) access the desired subset of these mechanisms, safely and conveniently.

The levels of privilege available to the class author include, for any field:

- Hide the field completely from clients, by exporting the corresponding attribute to *NONE*.
- Export it, but in read-only mode, by not exporting any procedure that modifies it.
- Export it for free read and write by any client, by also exporting a procedure of the *set_attribute* kind.
- Export it in **restricted-write** mode, by exporting a procedure such as *deposit* of class *ACCOUNT*, which adds a specified amount to the *balance* field, rather than directly setting the balance.

The last case is particularly interesting is that it allows the class designer to set the precise way in which clients will manipulate the class instances, respecting the properties of the class and its integrity. The exported routines may, through the Design by Contract mechanism reviewed later (8), place some further restrictions on the permitted modifications, for example by requiring the withdrawn amount to be positive.

These rules follow directly from the more general goals (reusability, extendibility, reliability) and principles (Uniform Access, information hiding) underlying Eiffel software design. They reflect a view that each class must denote a well-understood abstraction, defined by a set of exported features chosen by the class designer — the "control panel".

The class documentation (the *contract form*, see page 44) makes this view clear to client authors; no violation of that interface is permitted. This approach also paves the way for future **generalization** — the final step of the cluster lifecycle, seen earlier on page 9 — of the most promising components, and their inclusion into reusable libraries.

7 GENERICITY AND ARRAYS

Some of the classes that we will need, particularly in libraries, are **container** classes, describing data structures made of a number of objects of the same or similar types. Examples of containers include arrays, stacks and lists. The class *DEPOSIT_LIST* posited in earlier examples describes containers.

It is not hard, with the mechanisms seen so far, to write the class *DEPOSIT_LIST*, which would include such features as *count* (query returning the number of deposit objects in the list) and *put* (command to insert a new deposit object).

Most of the operations, however, would be the same for lists of objects other than deposits. To avoid undue replication of efforts and promote reuse, we need a way to describe **generic** container classes, which we can use to describe containers containing elements of many different types.

Making a class generic

The notation

```
class C [G] … The rest as for any other class declaration …
```

introduces a generic class. A name such as *G* appearing in brackets after the class name is known as a **formal generic parameter**; it represents an arbitrary type.

Within the class text, feature declarations can freely use *G* even though it is not known what type *G* stands for. Class *LIST* of EiffelBase, for example, includes features

```
first: G
        -- Value of first list item

extend (val: G) is
        -- Add a new item of value val at end of list
        …
```

The operations available on an entity such as *first* and *val*, whose type is a formal generic parameter, are the operations available on all types: use as source *y* of an assignment *x* := *y*, use as target *x* of such an assignment (although not for *val*, which as a formal routine argument is not writable), use in equality comparisons *x* = *y* or *x* /= *y*, and application of universal features from *ANY* such as *clone*, *equal* and *copy*.

To use a generic class such as list, a client will provide a type name as **actual generic parameter**. So instead of relying on a special purpose class *DEPOSIT_LIST*, the class *ACCOUNT* could include the declaration

```
all_deposits: LIST [DEPOSIT]
```

using *LIST* as a generic class and *DEPOSIT* as the actual generic parameter. Then all features declared in *LIST* as working on values of type *G* will work, when called on the target *all_deposits*, on values of type *DEPOSIT*. With the target

```
all_accounts: LIST [ACCOUNT]
```

these features would work on values of type *ACCOUNT*.

A note of terminology: to avoid confusion, Eiffel always uses the word **argument** for routine arguments, reserving **parameter** for the generic parameters of classes.

Genericity reconciles extendibility and reusability with the static type checking demanded by reliability. A typical error, such as confusing an account and a deposit, will be detected immediately at compile time, since the call *all_accounts.extend* (*dep*) is invalid for *dep* declared of type *DEPOSIT*. What is valid is something like *all_accounts.extend* (*acc*) for *acc* of type *ACCOUNT*. In other approaches, the same effect might require costly run-time checks (as in Java, C# or Smalltalk), with the risk of run-time errors.

This form of genericity is known as **unconstrained** because the formal generic parameter, *G* in the example, represents an arbitrary type. You may also want to use types that are guaranteed to have certai operations available. This is known as **constrained** genericity and will be studied with inheritance.

Arrays

An example of generic class from the Kernel Library is *ARRAY* [*G*], which describes direct-access arrays. Features include:

- *put* to replace an element's value, as in *my_array.put* (*val*, *25*) which replaces by *val* the value of the array entry at index 25.
- *item* to access an entry, as in *my_array.item* (*25*) yielding the entry at index 25. A synonym is **infix** "@", so that you may also write more tersely, for the same result, *my_array @ 25*.
- *lower*, *upper* and *count*: queries yielding the bounds and the number of entries.
- The creation procedure *make*, as in **create** *my_array.make* (*1, 50*) which creates an array with the given index bounds. It is also possible to resize an array through *resize*, retaining the old elements. In general, the Eiffel method abhors built-in limits, favoring instead structures that resize themselves when needed, either from explicit client request or automatically.

The comment made about *INTEGER* and other basic classes applies to *ARRAY* too: Eiffel compilers know about this class, and will be able to process expressions of the form *my_array.put* (*val*, *25*) and *my_array @ 25* in essentially the same way as a C

or Fortran array access — *my_array* [*25*] in C. But it is consistent and practical to let developers treat *ARRAY* as a class and arrays as objects; many library classes in EiffelBase, for example, inherit from *ARRAY*. Once again the idea is to get the best of both worlds: the convenience and uniformity of the object-oriented way of thinking; and the efficiency of traditional approaches.

A similar technique applies to another Kernel Library class, that one not generic: *STRING*, describing character strings with a rich set of string manipulation features.

Generic derivation

The introduction of genericity brings up a small difference between classes and types. A generic class *C* is not directly a type since you cannot declare an entity as being of type *C*: you must use some actual generic parameter *T* — itself a type. *C* [*T*] is indeed a type, but class *C* by itself is only a type template.

The process of obtaining a type *C* [*T*] from a general class *C* is known as a **generic derivation**; *C* [*T*] is a **generically derived type**. Type *T* itself is, recursively, either a non-generic class or again a generically derived type *D* [*U*] for some *D* and *U*, as in *LIST* [*ARRAY* [*INTEGER*]].)

It remains true, however, that every type is based on a class. The base class of a generically derived type *C* [*T*] is *C*.

8 DESIGN BY CONTRACT™, ASSERTIONS, EXCEPTIONS

Eiffel directly implements the ideas of Design by Contract™, which enhance software reliability and provide a sound basis for software specification, documentation and testing, as well as exception handling and the proper use of inheritance.

Design by Contract basics

A system — a software system in particular, but the ideas are more general — is made of a number of cooperating components. Design by Contract states that their cooperation should be based on precise specifications — *contracts* — describing each party's expectations and guarantees.

An Eiffel contract is similar to a real-life contract between two people or two companies, which it is convenient to express in the form of tables listing the expectations and guarantees. Here for example is how we could sketch the contract between a homeowner and the telephone company:

provide_service	**OBLIGATIONS**	**BENEFITS**
Client	(Satisfy precondition:) Pay bill	(From postcondition:) Get telephone service
Supplier	(Satisfy postcondition:) Provide telephone service	(From precondition:) No need to provide anything if bill not paid

Note how the obligation for each of the parties maps onto a benefit for the other. This will be a general pattern.

The client's obligation, which protects the supplier, is called a **precondition**. It states what the client must satisfy before requesting a certain service. The client's benefit, which describes what the supplier must do (assuming the precondition was satisfied), is called a **postcondition**.

In addition to preconditions and postconditions, contract clauses include **class invariants**, which apply to a class as a whole. More precisely a class invariant must be ensured by every creation procedure (or by the default initialization if there is no creation procedure), and maintained by every exported routine of the class.

Expressing assertions

Eiffel provides syntax for expressing preconditions (**require**), postconditions (**ensure**) and class invariants (**invariant**), as well as other assertion constructs studied later (see "Instructions", page 84): loop invariants and variants, check instructions.

Here is a partial update of class *ACCOUNT* with more assertions:

```
indexing
        description: "Simple bank accounts"
class
        ACCOUNT
feature -- Access
        balance: INTEGER
                        -- Current balance
        deposit_count: INTEGER is
                        -- Number of deposits made since opening
                do
                        … As before …
                end
```

```
feature -- Element change
    deposit (sum: INTEGER) is
            -- Add sum to account.
        require
            non_negative: sum >= 0
        do
            … As before …
        ensure
            one_more_deposit:
                deposit_count = old deposit_count + 1
            updated: balance = old balance + sum
        end

feature {NONE} -- Implementation
    all_deposits: DEPOSIT_LIST
            -- List of deposits since account's opening.

invariant
    consistent_balance: (all_deposits /= Void) implies
                                (balance = all_deposits.total)
    zero_if_no_deposits: (all_deposits = Void) implies
                                (balance = 0)
end -- class ACCOUNT
```

Each assertion is made of one or more subclauses, each of them a boolean expression (with the additional possibility of the **old** construct). The effect of including more than one subclause, as in the postcondition of *deposit* and in the invariant, is the same as connecting them through an **and**. Each clause may be preceded by a label, such as *consistent_balance* in the invariant, and a colon; the label is optional and does not affect the assertion's semantics, except for error reporting as explained in the next section, but including it systematically is part of the recommended style. The value of the boolean expression *a* **implies** *b* is true except if *a* is true and *b* false.

Because assertions benefit from the full power of boolean expressions, they may include function calls. This makes it possible to express sophisticated consistency conditions, such as "*the graph contains no cycle*", which would not be otherwise expressible through simple expressions, or even through first-order predicate calculus, but which are easy to implement as Eiffel functions returning boolean results.

The precondition of a routine expresses conditions that the routine is imposing on its clients. Here a call to *deposit* is correct if and only if the value of the argument is non-negative. The routine does not guarantee anything for a call that does not satisfy the precondition. It is in fact part of the Eiffel method that a routine body should **never**

test for the precondition, since it is the client's responsibility to ensure it. (An apparent paradox of Design by Contract, which is reflected in the bottom-right entries of the preceding and follwing contract tables, and should not be a paradox any more at the end of this discussion, is that one can get *more* reliable software by having *fewer* explicit checks in the software text.)

The postcondition of a routine expresses what the routine guaranteed to its clients for calls satisfying the precondition. The notation **old** *expression*, valid in postconditions (**ensure** clauses) only, denotes the value that *expression* had on entry to the routine.

The precondition and postcondition state the terms of the contract between the routine and its clients, similar to the earlier example of a human contract:

deposit	**OBLIGATIONS**	**BENEFITS**
Client	(Satisfy precondition:) Use a non-negative argument.	(From postcondition:) Get deposits list and balance updated.
Supplier	(Satisfy postcondition:) Update deposits list and balance.	(From precondition:) No need to handle negative arguments.

The class invariant, as noted, applies to all features. It must be satisfied on exit by any creation procedure, and is implicitly added to both the precondition and postcondition of every exported routine. In this respect it is both good news and bad news for the routine implementer: good news because it guarantees that the object will initially be in a stable state, averting the need in the example to check that the total of *all_deposits* is compatible with the *balance*; bad news because, in addition to its official contract as expressed by its specific postcondition, every routine must take care of restoring the invariant on exit.

A requirement on meaningful contracts is that they should be in good faith: satisfiable by an honest partner. This implies a consistency rule: if a routine is exported to a client (either generally or selectively), any feature appearing in its precondition must also be available to that client. Otherwise — for example if the precondition included **require** *n* > *0*, where *n* is a secret attribute — the supplier would be making demands that a good-faith client cannot possibly check for.

Note in this respect that *guaranteeing* a precondition does not necessarily mean, for the client, *testing* for it. Assuming *n* is exported, a call may test for the precondition

```
if x.n > 0 then x.r end
```

possibly with an **else** part. But if the context of the call, in the client's code, implies that *n* is positive — perhaps because some preceding call set it to the sum of two squares — then there is no need for an **if** or similar construct.

In such a case, a **check** instruction as introduced later ("Instructions", page 84) is recommended if the reason for omitting the test is non-trivial.

Using contracts for built-in reliability

What are contracts good for? Their first use is purely methodological. By applying a discipline of expressing, as precisely as possible, the logical assumptions behind software elements, you can write software whose reliability is built-in: software that is developed hand-in-hand with the rationale for its correctness.

This simple observation — usually not clear to people until they have practiced Design by Contract thoroughly on a large-scale project — brings as much change to software practices and quality as the rest of object technology.

Run-time assertion monitoring

Contracts in Eiffel are not just wishful thinking. They can be monitored at run time under the control of compilation options.

It should be clear from the preceding discussion that contracts are not a mechanism to test for special conditions, for example erroneous user input. For that purpose, the usual control structures (**if** *deposit_sum >= 0* **then** …) are available, complemented in applicable cases by the exception handling mechanism reviewed next. An assertion is instead a **correctness condition** governing the relationship between two software modules (not a software module and a human, or a software module and an external device). If *sum* is negative on entry to *deposit*, violating the precondition, the culprit is some other software element, whose author was not careful enough to observe the terms of the deal. Bluntly:

Assertion Violation rule

A run-time assertion violation is the manifestation of a bug.

To be more precise:

- A precondition violation signals a bug in the client, which did not observe its part of the deal.
- A postcondition (or invariant) violation signals a bug in the supplier — the routine — which did not do its job.

That violations indicate bugs explains why it is legitimate to enable or disable assertion monitoring through mere compilation options: for a correct system — one without bugs — assertions will always hold, so the compilation option makes no difference to the semantics of the system.

But of course for an incorrect system the best way to find out where the bug is — or just that there is a bug — is often to monitor the assertions during development and testing. Hence the presence of the compilation options, which ISE's EiffelStudio lets you set separately for each class, with defaults at the system and cluster levels:

- **no**: assertions have no run-time effect.
- **require**: monitor preconditions only, on routine entry.
- **ensure**: preconditions on entry, postconditions on exit.
- **invariant**: like **ensure**, plus class invariant on both entry and exit for qualified calls.
- **all**: like **invariant**, plus **check** instructions, loop invariants and loop variants ("Instructions", page 84).

An assertion violation, if detected at run time under one of these options other than the first, will cause an exception ("Exception handling", page 46). Unless the software has an explicit "retry" plan as explained in the discussion of exceptions, the violation will cause produce an exception trace and cause termination (or, in EiffelStudio, a return to the environment's browsing and debugging facilities at the point of failure). If present, the label of the violated subclause will be displayed, to help identify the problem.

The default is **require**. This is particularly interesting in connection with the Eiffel method's insistence on reuse: with libraries such as EiffelBase, richly equipped with preconditions expressing terms of use, an error in the **client software** will often lead, for example through an incorrect argument, to violating one of these preconditions. A somewhat paradoxical consequence is that even an application developer who does not apply the method too well (out of carelessness, haste, indifference or ignorance) will still benefit from the presence of contracts in *someone else*'s library code.

During development and testing, assertion monitoring should be turned on at the highest possible level. Combined with static typing and the immediate feedback of compilation techniques such as the Melting Ice Technology, this permits the development process mentioned in the section "Quality and functionality", page 10, where errors are exterminated at birth. No one who has not practiced the method in a real project can imagine how many mistakes are found in this way; surprisingly often, a violation will turn out to affect an assertion that was just included for goodness' sake, the developer being convinced that it could never "possibly" fail to be satisfied.

By providing a precise reference (the description of what the software is supposed to do) against which to assess the reality (what the software actually does), Design by Contract profoundly transforms the activities of debugging, testing and quality assurance.

When releasing the final version of a system, it is usually appropriate to turn off assertion monitoring, or bring it down to the **require** level. The exact policy depends on the circumstances; it is a tradeoff between efficiency considerations, the potential cost of mistakes, and how much the developers and quality assurance team trust the product. When developing the software, however, you should always assume — to avoid loosening your guard — that in the end monitoring will be turned off.

The contract form of a class

Another application of assertions governs documentation. Environment mechanisms, such as clicking the **Contract Form** icon in EifffelStudio, will produce, from a class text, an abstracted version which only includes the information relevant for client authors. Here is the contract form of class *ACCOUNT* in the latest version given:

```
indexing
        description: "Simple bank accounts"
class interface
        ACCOUNT
feature -- Access
        balance: INTEGER
                -- Current balance
        deposit_count: INTEGER
                -- Number of deposits made since opening
feature -- Element change
        deposit (sum: INTEGER)
                -- Add sum to account.
            require
                non_negative: sum >= 0
            ensure
                one_more_deposit: deposit_count = old
deposit_count + 1
                updated: balance = old balance + sum
invariant
        consistent_balance: balance = all_deposits.total
end -- class interface ACCOUNT
```

The words **class interface** are used instead of just **class** to avoid any confusion with actual Eiffel text, since this is documentation, not executable software. (It is in fact possible to generate a compilable variant of the Contract Form in the form of a deferred class, a notion defined later.)

Compared to the full text, the Contract Form of a class (also called its "*short form*") retains all its interface properties, relevant to client authors:

- Names and signatures (argument and result type information) for exported features.
- Header comments of these features, which carry informal descriptions of their purpose. (Hence the importance, mentioned in section 4, of always including such comments and writing them carefully.)
- Preconditions and postconditions of these features (at least the subclauses involving only exported features).
- Class invariant (same observation).

The following elements, however, are not in the Contract Form: any information about non-exported features; all the routine bodies (**do** clauses, or the **external** and **once** variants seen in "External software", page 16 above and "Once routines and shared objects", page 82 below); assertion subclauses involving non-exported features; and some keywords not useful in the documentation, such as **is** for a routine.

In accordance with the Uniform Access principle (page 19), the Contract Form does not distinguish between attributes and argument-less queries. In the above example, *balance* could be one or the other, as it makes no difference to clients, except possibly for performance.

The Contract Form is the fundamental tool for using supplier classes in the Eiffel method. It enables client authors to reuse software elements without having to read their source code. This is a crucial requirement in large-scale industrial developments.

The Contract Form satisfies two key requirements of good software documentation:

- It is truly abstract, free from the implementation details of what it describes and concentrating instead on its functionality.
- Rather than being developed separately — an unrealistic requirement, hard to impose on developers initially and becoming impossible in practice if we expect the documentation to remain up to date as the software evolves — the documentation is extracted from the software itself. It is not a separate product but a different view of the same product. This prolongs the **Single Product** principle that lies at the basis of Eiffel's seamless development model (section 3).

The Contract Form is only one of the relevant views. EiffelStudio, for example, generates graphical representations of system structures, to show classes and their relations — client, inheritance — according to the conventions of BON (the Business Object Notation). In accordance with the principles of seamlessness and reversibility, EiffelStudio lets you both work on the text, producing the graphics on the fly, or work on the graphics, updating the text on the fly; you can alternate as you wish between these two modes. The resulting process is quite different from more traditional approaches based on separate tools: an analysis and CASE workbench, often based on UML, to deal with an initial "bubble-and-arrow" description; and a separate programming environment, to deal with implementation aspects only. In Eiffel the environment provides consistent, seamless support from beginning to end.

The Contract Form — or its variant the Flat-Contract Form, which takes account of inheritance ("Flat and Flat-Contract Forms", page 72) are the standard form of library documentation, used extensively, for example, in the book *Reusable Software* (see bibliography). Assertions play a central role in such documentation by expressing the terms of the contract. As demonstrated *a contrario* by the widely publicized $500-million crash of the Ariane-5 rocket launcher in June of 1996, due to the incorrect reuse of a software module from the Ariane-4 project, **reuse without a contract documentation** is the path to disaster. Non-reuse would, in fact, be preferable.

Exception handling

Another application of Design by Contract governs the handling of unexpected cases. The vagueness of many discussions of this topic follows from the lack of a precise definition of terms such as "exception". With Design by Contract we are in a position to be specific:

- Any routine has a contract to achieve.
- Its body defines a strategy to achieve it — a sequence of operations, or some other control structure involving operations. Some of these operations are calls to routines, with their own contracts; but even an atomic operation, such as the computation of an arithmetic operation, has an implicit contract, stating that the result will be representable.
- Any one of these operations may **fail**, that is to say be unable to meet its contract; for example an arithmetic operation may produce an overflow (a non-representable result).
- The failure of an operation is an **exception** for the routine that needed the operation.
- As a result the routine may fail too — causing an exception in its own caller.

Note the precise definitions of the two key concepts, failure and exception. Although failure is the more basic one — since it is defined for atomic, non-routine operations — the definitions are mutually recursive, since an exception may cause a failure of the recipient routine, and a routine's failure causes an exception in its own caller.

Why state that an exception "may" cause a failure? It is indeed possible to "rescue" a routine from failure in the case of an exception, by equipping it with a clause labeled **rescue**, as in:

```
read_next_character (f: FILE) is
        -- Make next character available in last_character;
        -- if impossible, set failed to True.
    require
        readable: file.readable
    local
        impossible: BOOLEAN
    do
        if impossible then
            failed := True
        else
            last_character := low_level_read_function (f)
        end
    rescue
        impossible := True
        retry
    end
```

This example includes the only two constructs needed for exception handling: **rescue** and **retry**. A **retry** instruction is only permitted in a rescue clause; its effect is to start again the execution of the routine, without repeating the initialization of local entities (such as *impossible* in the example, which was initialized to *False* on first entry). Features *failed* and *last_character* are assumed to be attributes of the enclosing class.

This example is typical of the use of exceptions: as a last resort, for situations that should not occur. The routine has a precondition, *file.readable*, which ascertains that the file exists and is accessible for reading characters. So clients should check that everything is fine before calling the routine. Although this check is almost always a guarantee of success, a rare combination of circumstances could cause a change of file status (because a user or some other system is manipulating the file) between the check for *readable* and the call to *low_level_read_function*. If we assume this latter function will fail if the file is not readable, we must catch the exception.

A variant would be

```
local
    attempts: INTEGER
do
    if attempts < Max_attempts then
        last_character := low_level_read_function (f)
    else
        failed := True
    end
rescue
    attempts := attempts + 1
    retry
end
```

which would try again up to *Max_attempts* times before giving up.

The above routine, in either variant, never fails: it always fulfills its contract, which states that it should either read a character or set *failed* to record its inability to do so. In contrast, consider the new variant

```
local
    attempts: INTEGER
do
    last_character := low_level_read_function (f)
rescue
    attempts := attempts + 1
    if attempts < Max_attempts then
        retry
    end
end
```

with no more role for *failed*. In this case, after *Max_attempts* unsuccessful attempts, the routine will execute its **rescue** clause to the end, with no **retry** (the **if** having no **else** clause). This is how a routine **fails**. It will, as noted, pass on the exception to its caller.

Such a rescue clause should, before terminating, restore the invariant of the class so that the caller and possible subsequent **retry** attempts from higher up find the objects in a consistent state. As a result, the rule for an absent **rescue** clause — the case for the vast majority of routines in most systems — is that it is equivalent to

```
rescue
    default_rescue
```

where procedure *default_rescue* comes from *ANY*, where it is defined to do nothing; in a system built for robustness, classes subject to non-explicitly-**rescue**d exceptions should redefine *default_rescue* (perhaps using a creation procedure, which is bound by the same formal requirement) so that it will always restore the invariant.

Behind Eiffel's exception handling scheme lies the principle — at first an apparent platitude, but violated by many existing mechanisms — that a routine should **either succeed or fail**. This is in turn a consequence of Design by Contract principles: succeeding means being able to fulfill the contract, possibly after one or more **retry**; failure is the other case, which must always trigger an exception in the caller. Otherwise it would be possible for a routine to miss its contract and yet return to its caller in a seemingly normal state. That is the worst possible way to handle an exception.

Concretely, exceptions may result from the following events:

- A routine failure (**rescue** clause executed to the end with no **retry**), as just seen.
- Assertion violation, if for a system that runs with assertion monitoring on.
- Attempt to call a feature on a void reference: *x.f (…)*, the fundamental computational mechanism, can only work if *x* is attached to an object, and will cause an exception otherwise.
- Developer exception, as seen next.
- Operating system signal:arithmetic overfolow; no memory available for a requested creation or clone — even after garbage collection has rummaged everything to find some space. (But no C/C++-like "wrong pointer address", which cannot occur thanks to the statically typed nature of Eiffel.)

It is sometimes useful, when handling exceptions in **rescue** clauses, to ascertain the exact nature of the exception that got the execution there. For this it is suffices to inherit from the Kernel Library class *EXCEPTIONS*, which provides queries such as *exception*, giving the code for the last exception, and symbolic names ("Constant and unique attributes", page 83) for all such codes, such as *No_more_memory*. You can then process different exceptions differently by testing *exception* against various possibilities. The method strongly suggests, however, that exception handling code should remain simple; a complicated algorithm in a **rescue** clause is usually a sign that the mechanism is being misused.

Class *EXCEPTIONS* also provides various facilities for fine-tuning the exception facilities, such as a procedure *raise* that will explicitly trigger a "developer exception" with a code than can then be detected and processed.

Exception handling helps produce Eiffel software that is not just correct but robust, by planning for cases that should *not* normally arise, but might out of Murphy's law, and ensuring they do not affect the software's basic safety and simplicity.

Other applications of Design by Contract

The Design by Contract ideas pervade the Eiffel method. In addition to the applications just mentioned, they have two particularly important consequences:

- They make it possible to use Eiffel for analysis and design. At a high level of abstraction, it is necessary to be precise too. With the exception of BON, object-oriented analysis and design methods tend to favor abstraction over precision. Thanks to assertions, it is possible to express precise properties of a system (“*At what speed should the alarm start sounding?*”) without making any commitment to implementation. The discussion of deferred classes (“Applications of deferred classes”, page 60) will show how to write a purely descriptive, non-software model in Eiffel, using contracts to describe the essential properties of a system without any computer or software aspect.
- Assertions also serve to control the power of inheritance-related mechanisms — redeclaration, polymorphism, dynamic binding — and channel them to correct uses by assigning the proper semantic limits. See “Inheritance and contracts”, page 66.

9 INHERITANCE

Inheritance is a powerful and attractive technique. A look at either the practice or literature shows, however, that it is not always well applied. Eiffel has made a particular effort to tame inheritance for the benefit of modelers and software developers. Many of the techniques are original with Eiffel. Paul Dubois has written (*comp.lang.python* Usenet newsgroup, 23 March 1997): *there are two things that* [*Eiffel*] *got right that nobody else got right anywhere else*: *support for design by contract, and multiple inheritance*. *Everyone should understand these “correct answers” if only to understand how to work around the limitations in other languages*.

Basic inheritance structure

To make a class inherit from another, simply use an **inherit** clause:

```
indexing … class D creation … inherit
        A
        B
        …
feature
        …
```

This makes *D* an heir of *A*, *B* and any other class listed. Eiffel supports **multiple** inheritance: a class may have as many parents as it needs. Later sections ("Multiple inheritance and renaming", page 64 and "Repeated inheritance and selection", page 73) will explain how to handle possible conflicts between parent features.

> This discussion will rely on the terminology introduced on page 14: *descendants* of a class are the class itself, its heirs, the heirs of its heirs and so on. *Proper descendants* exclude the class itself. The reverse notions are *ancestors* and *proper ancestors*.

By default *D* will simply include all the original features of *A*, *B*, …, to which it may add its own through its **feature** clauses if any. But the inheritance mechanism is more flexible, allowing *D* to adapt the inherited features in many ways. Each parent name — *A*, *B*, … in the example — can be followed by a Feature Adaptation clause, with subclauses, all optional, introduced by keywords **rename**, **export**, **undefine**, **redefine** and **select**, enabling the author of *A* to make the best use of the inheritance mechanism by tuning the inherited features to the precise needs of *D*. This makes inheritance a principal tool in the Eiffel process, mentioned earlier, of carefully crafting each individual class, like a machine, for the benefit of its clients. The next sections review the various Feature Adaptation subclauses.

Redefinition

The first form of feature adaptation is the ability to change the implementation of an inherited feature.

Assume a class *SAVINGS_ACCOUNT* that specializes the notion of account. It is probably appropriate to define it as an heir to class *ACCOUNT*, to benefit from all the features of *ACCOUNT* still applicable to savings accounts, and to reflect the conceptual relationship between the two types: every savings account, apart from its own specific properties, also "is" an account. But we may need to produce a different effect for procedure *deposit* which, besides recording the deposit and updating the balance, may also need, for example, to update the interest.

This example is typical of the form of reuse promoted by inheritance and crucial to effective reusability in software: the case of *reuse with adaptation*. Traditional forms of reuse are all-or-nothing: either you take a component exactly as it is, or you build your own. Inheritance will get us out of this "reuse or redo" dilemma by allowing us to reuse *and* redo. The mechanism is feature redefinition:

```
indexing
        description: "Savings accounts"
class
        SAVINGS_ACCOUNT
inherit
        ACCOUNT
                redefine deposit end
feature -- Element change
        deposit (sum: INTEGER) is
                        -- Add sum to account.
                do
                        … New implementation (see below) …
                end
        … Other features …
end -- class SAVINGS_ACCOUNT
```

Without the **redefine** subclause, the declaration of *deposit* would be invalid, yielding two features of the same name, the inherited one and the new one. The subclause makes this valid by specifying that the new declaration will override the old one.

In a redefinition, the original version — such as the *ACCOUNT* implementation of *deposit* in this example — is called the **precursor** of the new version. It is common for a redefinition to rely on the precursor's algorithm and add some other actions; the reserved word *Precursor* helps achieve this goal simply. Permitted only in a routine redefinition, it denotes the parent routine being redefined. So here he body of the new *deposit* (called "New implementation" above) could be of the form

```
Precursor (sum)  -- Apply ACCOUNT's version of deposit
… Instructions to update the interest …
```

Besides changing the implementation of a routine, a redefinition can turn an argument-less function into an attribute; for example a proper descendant of *ACCOUNT* could redefine *deposits_count*, originally a function, as an attribute. The Uniform Access Principle (page 19) guarantees that the redefinition makes no change for clients, which will continue to use the feature under the form *acc.deposits_count*.

Polymorphism

The inheritance mechanism is relevant to both roles of classes: module and type. Its application as a mechanism to reuse, adapt and extend features from one class to another, as just seen, covers its role as a **module extension** mechanism. But it's also a **subtyping** mechanism. To say that *D* is an heir of *A*, or more generally a descendant of *A*, is to expresses that instances of *D* can be viewed as instances of *A*.

Polymorphic assignment supports this second role. In an assignment *x* := *y*, the types of *x* and *y* do not have, with inheritance, to be identical; the rule is that the type of *y* must simply **conform** to the type of *x*. A class *D* conforms to a class *A* if and only if it is a descendant of *A* (which includes the case in which *A* and *D* are the same class); if these classes are generic, conformance of *D* [*U*] to *C* [*T*] requires in addition that type *U* conform to type *T* (through the recursive application of the same rules).

> In addition, it follows from the earlier discussion of tuples ("Tuple types", page 91), that *TUPLE* [*X*] conforms to *TUPLE*, *TUPLE* [*X, Y,*] to *TUPLE* [*X*] and so on.

So with the inheritance structure that we have seen, the declarations

```
acc: ACCOUNT; sav: SAVINGS_ACCOUNT
```

make it valid to write the assignment

```
acc := sav
```

which will assign to *acc* a reference attached (if not void) to a direct instance of type *SAVINGS_ACCOUNT*, not *ACCOUNT*.

Such an assignment, where the source and target types are different, is said to be polymorphic. An entity such as *acc*, which as a result of such assignments may become attached at run time to objects of types other than the one declared for it, is itself called a polymorphic entity.

For polymorphism to respect the reliability requirements of Eiffel, it must be controlled by the type system and enable static type checking. We certainly do not want an entity of type *ACCOUNT* to become attached to an object of type *DEPOSIT*. Hence the second typing rule:

Type Conformance rule

An assignment *x* := *y*, or the use of *y* as actual argument corresponding to the formal argument *x* in a routine call, is only valid if the type of *y* conforms to the the type of *x*.

The second case listed in the rule is a call such as *target*.*routine* (…, *y*, …) where the routine declaration is of the form *routine* (…, *x*: *SOME_TYPE*, …). The relationship between *y*, the actual argument in the call, and the corresponding formal argument *x*, is exactly the same as in an assignment *x* := *y*: not just the type rule, as expressed by Type Conformance (the type of *y* must conform to *SOME_TYPE*), but also the actual run-time effect which, as for assignments, will be either a reference attachment or, for expanded types, a copy.

The ability to accept the assignment *x* := *Void* for *x* of any reference type ("Basic operations", page 28) is a consequence of the Type Conformance rule, since *Void* is of type *NONE* which by construction ("The global inheritance structure", page 15) conforms to all types.

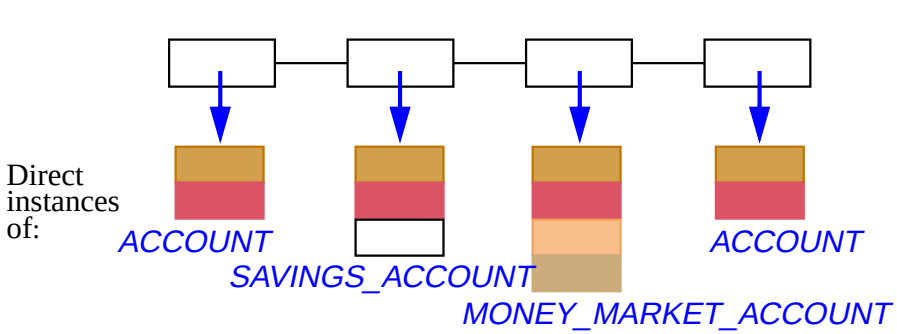
Polymorphism also yields a more precise definition of "instance". A **direct instance** of a type *A* is an object created from the exact pattern defined by the declaration of *A*'s base class, with one field for each of the class attributes; you will obtain it through a creation instruction of the form **create** *x*…, for *x* of type *A*, or by cloning an existing direct instance. An **instance** of *A* is a direct instance of any type conforming to *A*: *A* itself, but also any type based on descendant classes. So an instance of *SAVINGS_ACCOUNT* is also an instance, although not a direct instance, of *ACCOUNT*.

A consequence of polymorphism is the ability to define **polymorphic data structures**. With a declaration such as

accounts: *LIST* [*ACCOUNT*]

the procedure call *accounts*.*extend* (*acc*), because it uses a procedure *extend* which in this case expects an argument of any type conforming to *ACCOUNT*, will be valid not only if *acc* is of type *ACCOUNT* but also if it is of a descendant type such as *SAVINGS_ACCOUNT*. Successive calls of this kind make it possible to construct a data structure that, at run-time, might contain objects of several types, all conforming to *ACCOUNT*:

Polymorphic data structure



Such polymorphic data structures combine the flexibility and safety of genericity and inheritance. You can make them more or less general by choosing for the actual generic parameter, here *ACCOUNT*, a type higher or lower in the inheritance hierarchy. Static typing is again essential here, prohibiting for example a mistaken insertion of the form *accounts*.*extend* (*dep*) where *dep* is of type *DEPOSIT*, which does not conform to *ACCOUNT*.

At the higher (most abstract) end of the spectrum, you can produce an unrestrictedly polymorphic data structure *general_list*: *LIST* [*ANY*] which makes the call *general_list*.*extend* (*x*) valid for any *x*. The price to pay is that retrieving an element from such a structure will yield an object on which the only known applicable operations are the most general ones, valid for all types: assignment, copy, clone, equality comparison and others from *ANY*. Assignment attempt, studied below, will make it possible to apply more specific operations after checking dynamically that a retrieved object is of the appropriate type.

Dynamic binding

The complement of polymorphism is dynamic binding, the answer to the question "What version of a feature will be applied in a call whose target is polymorphic?".

Consider *acc* is of type *ACCOUNT*. Thanks to polymorphism, an object attached to *acc* may be a direct instance not just of *ACCOUNT* but also of *SAVINGS_ACCOUNT* or other descendants. Some of these descendants, indeed *SAVINGS_ACCOUNT* among them, redefine features such as *deposit*. Then we have to ask what the effect will be for a call of the form

```
acc.deposit (some_value)
```

Dynamic binding is the clearly correct answer: the call will execute the version of *deposit* from the generating class of the object attached to *acc* at run time. If *acc* is attached to a direct instance of *ACCOUNT*, execution will use the original *ACCOUNT* version; if *acc* is attached to a direct instance of *SAVINGS_ACCOUNT*, the call will execute the version redefined in that class.

This is a clear correctness requirement. A policy of *static binding* (as available for example in C++ or Delphi, for non-virtual functions) would take the declaration of *acc* as an *ACCOUNT* literally. But that declaration is only meant to ensure generality, to enable the use of a single entity *acc* in many different cases: what counts at execution time is the object that *acc* represents. Applying the *ACCOUNT* version to a *SAVINGS_ACCOUNT* object would be wrong, possibly leading in particular to objects that violate the invariant of their own generating class (since there is no reason a routine of *ACCOUNT* will preserve the specific invariant of a proper descendant such as *SAVINGS_ACCOUNT*, which it does not even know about).

In some cases, the choice between static and dynamic binding does not matter: this is the case for example if a call's target is not polymorphic, or if the feature of the call is redefined nowhere in the system. In such cases the use of static binding permits slightly faster calls (since the feature is known at compile time). This application of static binding should, however, be treated as a **compiler optimization**. The EiffelStudio compiler, under its "finalization" mode, which performs extensive optimization, will detect some of these cases and process them accordingly — unlike approaches that make developers responsible for specifying what should be static and what dynamic (a tedious and error-prone task, especially delicate because a minute change in the software can make a static call, in a far-away module of a large system, suddenly become dynamic). Eiffel programmers don't need to worry about such aspects; they can rely on the semantics of dynamic binding in all cases, with the knowledge that the compiler will apply static binding when safe and desirable.

Even in cases that require dynamic binding, the design of Eiffel, in particular the typing rules, enable compilers to make the penalty over the static-binding calls of traditional approaches very small and, most importantly, **constant-bounded**: it does not grow with the depth or complexity of the inheritance structure. The discovery in 1985 of a technique for constant-time dynamic binding calls, even in the presence of multiple and repeated inheritance, was the event that gave the green light to the development of Eiffel.

Dynamic binding is particularly interesting for polymorphic data structures. If you iterate over the list of accounts of various kinds, *accounts*: *LIST* [*ACCOUNT*], illustrated in the last figure, and at each step let *acc* represent the current list element, you can repeatedly apply

```
acc.deposit (…)
```

to have the appropriate variant of the *deposit* operation triggered for each element.

The benefit of such techniques appears clearly if we compare them with the traditional way to address such needs: using multi-branch discriminating instructions of the form **if** "*Account is a savings account*" **then** … **elseif** "*It is a money market account*" **then** … and so on, or the corresponding **case** … **of** …, **switch** or **inspect** instructions. Apart from their heaviness and complexity, such solutions cause many components of a software system to rely on the knowledge of the exact set of variants available for a certain notion, such as bank account. Then any addition, change or removal of variants can cause a ripple of changes throughout the architecture. This is

one of the majors obstacles to extendibility and reusability in traditional approaches. In contrast, using the combination of inheritance, redefinition, polymorphism and dynamic binding makes it possible to have a **point of single choice** — a unique location in the system which knows the exhaustive list of variants. Every client then manipulates entities of the most general type, *ACCOUNT*, through dynamically bound calls of the form *acc.some_account_feature* (…).

These observations make dynamic binding appear for what it is: not an implementation mechanism, but an **architectural technique** that plays a key role (along with information hiding, which it extends, and Design by Contract, to which it is linked through the assertion redefinition rules seen below) in providing the modular system architectures of Eiffel, the basis for the method's approach to reusability and extendibility. These properties apply as early as analysis and modeling, and continue to be useful throughout the subsequent steps.

Deferred features and classes

The examples of dynamic binding seen so far assumed that all classes were fully implemented, and dynamically bound features had a version in every relevant class, including the most general ones such as *ACCOUNT*.

It is also useful to define classes that leave the implementation of some of their features entirely to proper descendants. Such an abstract class is known as **deferred**; so are its unimplemented features. The reverse of deferred is **effective**, meaning fully implemented.

LIST is a typical example of deferred class. As it describes the general notion of list, it should not favor any particular implementation; that will be the task of its effective descendants, such as *LINKED_LIST* (linked implementation), *TWO_WAY_LIST* (linked both ways), *ARRAYED_LIST* (implementation by an array), all effective, and all indeed to be found in EiffelBase.

At the level of the deferred class *LIST*, some features such as *extend* (add an item at the end of the list) will have no implementation and hence will be declared as deferred. Here is the corresponding form, illustrating the syntax for both deferred classes and their deferred features:

```
indexing
        description: "Sequential finite lists, without a commitment%[
                        to a representation%]"
deferred class
        LIST [G]
feature -- Access
        count: INTEGER is
                        -- Number of items in list
                do
                        … See below; this feature can be effective …
                end

feature -- Element change
        extend (x: G) is
                        -- Add x at end of list.
                require
                        space_available: not full
                deferred
                ensure
                        one_more: count = old count + 1
                end

… Other feature declarations and invariant …
end -- class LIST
```

A deferred feature (considered to be a routine, although it can yield an attribute in a proper descendant) has the single keyword **deferred** in lieu of the **do** *Instructions* clause of an effective routine. A deferred class — defined as a class that has at least one deferred feature — must be introduced by **deferred class** instead of just **class**.

As the example of *extend* shows, a deferred feature, although it has no implementation, can be equipped with assertions. They will be binding on implementations in descendants, in a way to be explained below.

Deferred classes do not have to be *fully* deferred. They may contain some effective features along with their deferred ones. Here, for example, we may express *count* as a function:

```
count: INTEGER is
        -- Number of items in list
    do
        from start until after loop
            Result := Result + 1; forth
        end
    end
```

This implementation relies on the loop construct described below (**from** introduces the loop initialization) and on a set of deferred features of the class which allow traversal of a list based on moving a fictitious cursor: **start** to bring the cursor to the first element if any, **after** to find out whether all relevant elements have been seen, and **forth** (with precondition **not** *after*) to advance the cursor to the next element. Procedure *forth* itself appears as

```
forth is
        -- Advance cursor by one position
    require
        not_after: not after
    deferred
    ensure
        moved_right: index = old index + 1
    end
```

where *index* — another deferred feature — is the integer position of the cursor.

Although the above version of feature *count* is time-consuming — it implies a whole traversal just for the purpose of determining the number of elements — it has the advantage of being applicable to all variants, without any commitment to a choice of implementation, as would follow for example if we decided to treat *count* as an attribute. Proper descendants can always redefine *count* for more efficiency.

Function *count* illustrates one of the most important contributions of the method to reusability: the ability to define **behavior classes** that capture common behaviors (such as count) while leaving the details of the behaviors (such as *start*, *after*, *forth*) open to many variants. As noted earlier, traditional approaches to reusability provide closed reusable components. A component such as *LIST*, although equipped with directly usable behaviors such as count, is open to many variations, to be provided by proper descendants.

Some O-O languages support only the two extremes: fully effective classes, and fully deferred “interfaces”, but not classes with a mix of effective and deferred features. This is an unacceptable limitation, negating the object-oriented method’s support for a seamless, continuous spectrum from the most abstract to the most concrete.

A class *B* inheriting from a deferred class *A* may provide implementations — effective declarations — for the features inherited in deferred form. In this case there is no need for a **redefine** subclause; the effective versions simply replace the inherited versions. The class is said to **effect** the corresponding features. If after this process there remain any deferred features, B is still considered deferred, even if it introduces no deferred features of its own, and must be declared as **deferred class**.

In the example, classes such as *LINKED_LIST* and *ARRAYED_LIST* will effect all the deferred features they inherit from *LIST* — *extend*, *start* etc. — and hence will be effective.

Except in some applications restricted to pure system modeling — as discussed next — the main benefit of deferred classes and features comes from polymorphism and dynamic binding. Because *extend* has no implementation in class *LIST*, a call of the form *my_list.extend (…)* with my_list of type *LIST [T]* for some *T* can only be executed if *my_list* is attached to a direct instance of an effective proper descendant of *LIST*, such as *LINKED_LIST*; then it will use the corresponding version of *extend*. Static binding would not even make sense here.

Even an effective feature of *LIST* such as count may depend on deferred features (start and so on), so that a call of the form my_list.count can only be executed in the context of an effective descendant.

All this indicates that a deferred class must have **no direct instance**. (It will have instances, the direct instances of its effective descendants.) If it had any, we could call deferred features on them, leading to execution-time impossibility. The rule that achieves this goal is simple: if the base type of *x* is a deferred class, no creation instruction of target *x*, of the form **create** *x* …, is permitted.

Applications of deferred classes

Deferred classes cover abstract notions with many possible variants. They are widely used in Eiffel where they cover various needs:

- Capturing high-level classes, with common behaviors.
- Defining the higher levels of a general taxonomy, especially in the inheritance structure of a library.
- Defining the components of an architecture during system design, without commitment to a final implementation.
- Describing domain-specific concepts in analysis and modeling.

These applications make deferred classes a central tool of the Eiffel method's support for seamlessness and reversibility. The last one in particular uses deferred classes and features to model objects from an application domain, without any commitment to implementation, design, or even software (and computers). Deferred classes are the ideal tool here: they express the properties of the domain's abstractions, without any temptation of implementation bias, yet with the precision afforded by type declarations, inheritance structures (to record classifications of the domain concepts), and contracts to express the abstract properties of the objects being described.

Rather than using a separate method and notation for analysis and design, this apprroach integrates seamlessly with the subsequent phases (assuming the decision is indeed taken to develop a software system): it suffices to refine the deferred classes progressively by introducing effective elements, either by modifying the classes themselves, or by introducing design- and implementation-oriented descendants. In the resulting system, the classes that played an important role for analysis, and are the most meaningful for customers, will remain important; as we have seen ("Seamlessness and reversibility", page 9) this *direct mapping* property is a great help for extendibility.

The following sketch (from the book Object-Oriented Software Construction) illustrates these ideas on the example of scheduling the programs of a TV station. This is pure modeling of an application domain; no computers or software are involved yet. The class describes the notion of program segment.

Note the use of assertions to define semantic properties of the class, its instances and its features. Although often presented as high-level, most object-oriented analysis methods (with the exception of Waldén's and Nerson's Business Object Notation) have no support for the expression of such properties, limiting themselves instead to the description of broad structural relationships.

```
indexing
        description: "Individual fragments of a broadcasting schedule"
deferred class
        SEGMENT

feature -- Access
        schedule: SCHEDULE is deferred end
                -- Schedule to which segment belongs
        index: INTEGER is deferred end
                -- Position of segment in its schedule
        starting_time, ending_time: INTEGER is deferred end
                -- Beginning and end of scheduled air time
        next: SEGMENT is deferred end
                -- Segment to be played next, if any
```

```
        sponsor: COMPANY is deferred end
                -- Segment's principal sponsor
        rating: INTEGER is deferred end
                -- Segment's rating (for children's viewing etc.)
        Minimum_duration: INTEGER is 30
                -- Minimum length of segments, in seconds
        Maximum_interval: INTEGER is 2
                -- Maximum time (seconds) between successive segments

    feature -- Element change
        set_sponsor (s: SPONSOR) is
            require
                not_void: s /= Void
            deferred
            ensure
                sponsor_set: sponsor = s
            end

        … change_next, set_rating omitted …

    invariant
        in_list: (1 <= index) and (index <= schedule.segments.count)
        in_schedule: schedule.segments.item (index) = Current
        next_in_list: (next /= Void) implies
                            (schedule.segments.item (index + 1) = next)
        no_next_iff_last: (next = Void) =
                            (index = schedule.segments.count)
        non_negative_rating: rating >= 0
        positive times: (starting_time > 0) and (ending_time > 0)
        sufficient_duration: ending_time – starting_time >=
                            Minimum_duration
        decent_interval: (next.starting_time) – ending_time <=
                            Maximum_interval
    end
```

Structural property classes

Some deferred classes describe a structural property, useful to the description of many other classes. Typical examples are classes of the Kernel Library in EiffelBase:

- *NUMERIC* describes objects on which arithmetic operations +, –, ∗, / are available, with the properties of a ring (associativity, distributivity, zero elements etc.). Kernel Library classes such as *INTEGER* and *REAL* — but not, for example, *STRING* — are descendants of *NUMERIC*. An application that defines a class *MATRIX* may also make it a descendant of *NUMERIC*.

- *COMPARABLE* describes objects on which comparison operations <, <=, >, >= are available, with the properties of a total preorder (transitivity, irreflexivity). Kernel Library classes such as *CHARACTER*, *STRING* and *INTEGER* — but not out *MATRIX* example — are descendants of *NUMERIC*.

For such classes it is again essential to permit effective features in a deferred class, and to include assertions. For example class *COMPARABLE* declares **infix** "<" as deferred, and expresses >, >= and <= effectively in terms of it.

The type **like** *Current* will be explained in "Covariance and anchored declarations", page 79; you may understand it, in the following class, as equivalent to *COMPARABLE*.

```
indexing
        description: "Objects that can be compared according to a total
preorder relation"
deferred class
        COMPARABLE

feature -- Comparison
        infix "<" (other: like Current): BOOLEAN is
                        -- Is current object less than other?
                require
                        other_exists: other /= Void
                deferred
                ensure
                        asymmetric: Result implies not (other < Current)
                end

        infix "<=" (other: like Current): BOOLEAN is
                        -- Is current object less than or equal to other?
                require
                        other_exists: other /= Void
                do
                        Result := (Current < other) or is_equal (other)
                ensure
                        definition: Result = (Current < other) or
                                                is_equal (other)
                end

… Other features: infix ">", min, max, …
invariant
        irreflexive: not (Current < Current)
end -- class COMPARABLE
```

Multiple inheritance and renaming

It is often necessary to define a new class in terms of several existing ones. For example:

- The Kernel Library classes *INTEGER* and *REAL* must inherit from both *NUMERIC* and *COMPARABLE*.
- A class *TENNIS_PLAYER*, in a system for keeping track of player ranking, will inherit from *COMPARABLE*, as well as from other domain-specific classes.
- A class *COMPANY_PLANE* may inherit from both *PLANE* and *ASSET*.
- Class *ARRAYED_LIST*, describing an implementation of lists through arrays, may inherit from both *LIST* and *ARRAY*.

In all such cases multiple inheritance provides the answer.

Multiple inheritance can cause **name clashes**: two parents may include a feature with the same name. This would conflict with the ban on name overloading within a class — the rule that no two features of a class may have the same name. Eiffel provides a simple way to remove the name clash at the point of inheritance through the **rename** subclause, as in

```
indexing
        description: "Sequential finite lists implemented as arrays"
class
        ARRAYED_LIST [G]
inherit
        LIST [G]
        ARRAY [G]
                rename
                        count as capacity, item as array_item
                end
feature
        ...
end -- class ARRAYED_LIST
```

Here both *LIST* and *ARRAY* have features called *count* and *item*. To make the new class valid, we give new names to the features inherited from *ARRAY*, which will be known within *ARRAYED_LIST* as *capacity* and *array_item*. Of course we could have renamed the *LIST* versions instead, or renamed along both inheritance branches.

Every feature of a class has a **final name**: for a feature introduced in the class itself ("immediate" feature) it is the name appearing in the declaration; for an inherited feature that is not renamed, it is the feature's name in the parent; for a renamed feature, it is the name resulting from the renaming. This definition yields a precise statement of the rule against in-class overloading:

Final Name rule

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

It is interesting to compare renaming and redefinition. The principal distinction is between features and feature names. Renaming keeps a feature, but changes its name. Redefinition keeps the name, but changes the feature. In some cases, it is of course appropriate to do both.

Renaming is interesting even in the absence of name clashes. A class may inherit from a parent a feature which it finds useful for its purposes, but whose name, appropriate for the context of the parent, is not consistent with the context of the heir. This is the case with *ARRAY*'s feature *count* in the last example: the feature that defines the number of items in an array — the total number of available entries — becomes, for an arrayed list, the *maximum* number of list items; the truly interesting indication of the number of items is the count of how many items have been inserted in the list, as given by feature *count* from *LIST*. But even if we did not have a name clash because of the two inherited *count* features we should rename *ARRAY*'s *count* as *capacity* to maintain the consistency of the local feature terminology.

The **rename** subclause appears before all the other feature adaptation subclauses — **redefine** already seen, and the remaining ones **export**, **undefine** and **select** — since an inherited feature that has been renamed sheds its earlier identity once and for all: within the class, and to its own clients and descendants, it will be known solely through the new name. The original name has simply disappeared from the name space. This is essential to the view of classes presented earlier: self-contained, consistent abstractions prepared carefully for the greatest enjoyment of clients and descendants.

Inheritance and contracts

A proper understanding of inheritance requires looking at the mechanism in the framework of Design by Contract, where it will appear as a form of *subcontracting*.

The first rule is that invariants accumulate down an inheritance structure:

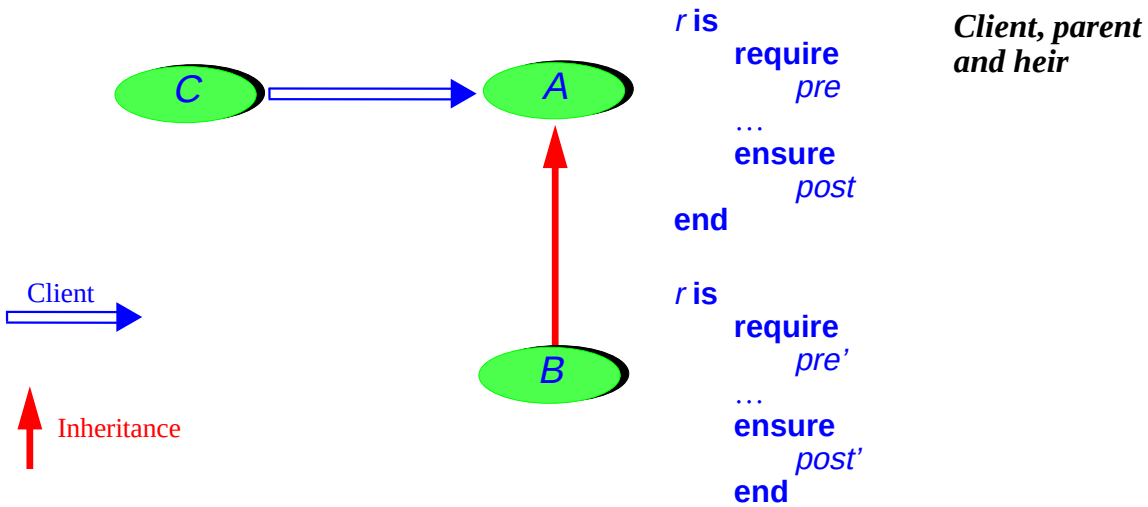
Invariant Accumulation rule

The invariants of all the parents of a class apply to the class itself.

The invariant of a class is automatically considered to include — in the sense of logical "and" — the invariants of all its parents. This is a consequence of the view of inheritance as an "is" relation: if we may consider every instance of *B* as an instance of *A*, then every consistency constraint on instances of *A* must also apply to instances of *B*.

Next we consider routine preconditions and postconditions. The rule here will follow from an examination of what contracts mean in the presence of polymorphism and dynamic binding.

Consider a parent *A* and a proper descendant *B* (a direct heir on the following figure), which redefines a routine *r* inherited from *A*.



As a result of dynamic binding, a call *a1*.*r* from a client *C* may be serviced not by *A*'s version of *r* but by *B*'s version if *a1*, although declared of type *A*, becomes at run time attached to an instance of *B*. This shows the combination of inheritance, redefinition, polymorphism and dynamic binding as providing a form of subcontracting; *A* subcontracts certain calls to *B*.

The problem is to keep subcontractors honest. Assuming preconditions and postconditions as shown on the last figure, a call in *C* of the form

```
if a1.pre then a1.r end
```

or just *a1.q*; *a1.r* where the postcondition of *q* implies the precondition *pre* of *r*, satisfies the terms of the contract and hence is entitled to being handled correctly — to terminate in a state satisfying *a1.post*. But if we let the subcontractor *B* redefine the assertions to arbitrary *pre'* and *post'*, this is not necessarily the case: *pre'* could be stronger than *pre*, enabling *B* not to process correctly certain calls that are correct from *A*'s perspective; and *post'* could be weaker than *post*, enabling *B* to do less of a job than advertized for *r* in the Contract Form of *A*, the only official reference for authors of client classes such as *C*. (An assertion *p* is stronger than or equal to an assertion *q* if *p* implies *q* in the sense of boolean implication.)

The rule, then, is that for the redefinition to be correct the new precondition *pre'* must be weaker than or equal to the original *pre*, and the new postcondition *post'* must be stronger than or equal to the original *post'*.

Because it is impossible to check simply that an assertion is weaker or stronger than another, the language rule relies on different forms of the assertion constructs, **require else** and **ensure then**, for redeclared routines. They rely on the mathematical property that, for any assertions *p* and *q*, *p* implies (*p* **or** *q*), and (*p* **and** *q*) implies *p*. For a precondition, using **require else** with a new assertion will perform an **or**, which can only weaken the original; for a postcondition, **ensure then** will perform an **and**, which can only strengthen the original. Hence the rule:

Assertion Redeclaration rule

In the redeclared version of a routine, it is not permitted to use a **require** or **ensure** clause. Instead you may:

- Introduce a new condition with **require else**, for or-ing with the original precondition.
- Introduce a new condition with **ensure then**, for and-ing with the original postcondition.

In the absence of such a clause, the original assertions are retained.

The last case — retaining the original — is frequent but by no means universal.

The Assertion Redeclaration rule applies to **redeclarations**. This terms covers not just redefinition but also effecting (the implementation, by a class, of a feature that it inherits deferred). The rules — not just for assertions but also, as reviewed below, for typing — are indeed the same in both cases. Without the Assertion Redeclaration rule, assertions on deferred features, such as those on *extend*, *count* and *forth* in "Deferred features and classes", page 57, would be almost useless — wishful thinking; the rule makes them binding on all effectings in descendants.

From the Assertion Redeclaration rule follows an interesting technique: **abstract preconditions**. What needs to be weakened for a precondition (or strengthened for a postcondition) is not the assertion's concrete semantics but its abstract specification as seen by the client. A descendant can change the *implementation* of that specification as it pleases, even to the effect of strengthening the concrete precondition, as long as the abstract form is kept or weakened. The precondition of procedure *extend* in the deferred class *LIST* provided an example. We wrote the routine (page 58) as

```
extend (x: G) is
        -- Add x at end of list.
    require
        space_available: not full
    deferred
    ensure
        one_more: count = old count + 1
    end
```

The precondition expresses that it is only possible to add an item to a list if the representation is not full. We may well consider — in line with the Eiffel principle that whenever possible structures should be of unbounded capacity — that *LIST* should by default make *full* always return false:

```
full: BOOLEAN is
        -- Is representation full?
        -- (Default: no)
    do
        Result := False
    end
```

Now a class *BOUNDED_LIST* that implements bounded-size lists (inheriting, like the earlier *ARRAYED_LIST*, from both *LIST* and *ARRAY*) may redefine *full*:

```
full: BOOLEAN is
        -- Is representation full?
        -- (Answer: if and only if number of items is capacity)
    do
        Result := (count = capacity)
    end
```

Procedure *extend* remains applicable as before; any client that used it properly with *LIST* can rely polymorphically on the *FIXED_LIST* implementation. The abstract precondition of *extend* has not changed, even though the concrete implementation of that precondition has in fact been strengthened.

Note that a class such as *BOUNDED_LIST*, the likes of which indeed appear in EiffelBase, is not a violation of the Eiffel advice to stay away from fixed-size structures. The corresponding structures are bounded, but the bounds are changeable. Although *extend* requires **not** *full*, another feature, called *force* in all applicable classes, will add an element at the appropriate position by resizing and reallocating the structure if necessary. Even arrays in Eiffel are not fixed-size, and have a procedure *force* with no precondition, accepting any index position.

The Assertion Redeclaration rule, together with the Invariant Accumulation rule, provides the right methodological perspective for understanding inheritance and the associated mechanisms. Defining a class as inheriting from another is a strong commitment; it means inheriting not only the features but the logical constraints. Redeclaring a routine is bound by a similar committment: to provide a new implementation (or, for an effecting, a first implementation) of a previously defined semantics, as expressed by the original contract. Usually you have a wide margin for choosing your implementation, since the contract only defines a range of possible behaviors (rather than just one behavior), but you **must** remain within that range. Otherwise you would be perverting the goals of redeclaration, using this mechanism as a sort of late-stage hacking to override bugs in ancestor classes.

Join and uneffecting

It is not an error to inherit two deferred features from different parents under the same name, provided they have the same signature (number and types of arguments and result). In that case a process of **feature join** takes place: the features are merged into just one — with their preconditions and postconditions, if any, respectively or-ed and and-ed.

More generally, it is permitted to have any number of deferred features and at most *one* effective feature that share the same name: the effective version, if present will effect all the others.

All this is not a violation of the Final Name rule (page 65), since the name clashes prohibited by the rule involve two *different* features having the same final name; here the result is just *one* feature, resulting from the join of all the inherited versions.

Sometimes we may want to join *effective* features inherited from different parents, assuming again the features have compatible signatures. One way is to redefine them all into a new version; then they again become one feature, with no name clash in the sense of the Final Name rule. But in other cases we may simply want one of the inherited implementations to take over the others. The solution is to revert to the preceding case by **uneffecting** the other features; uneffecting an inherited effective feature makes it deferred (this is the reverse of effecting, which turns an inherited deferred feature into an effective one). The syntax uses the **undefine** subclause:

```
class D inherit
    A
        rename
            g as f          -- g was effective in A
        undefine
            f
        end
    B
        undefine f end      -- f was effective in B
    C
        -- C also has an effective feature f, which will serve as
        -- implementation for the result of the join.
feature
    …
```

Again what counts, to determine if there is an invalid name clash, is the final name of the features. In this example to of the joined features were originally called *f*; the one from *A* was called *g*, but in *D* it is renamed as *f*, so without the undefinition it would cause an invalid name clash.

Feature joining is the most common application of uneffecting. In some non-joining cases, however, it may be useful to forget the original implementation of a feature and let it start a new life devoid of any burden from the past.

Changing the export status

Another Feature Adaptation subclause, **export**, makes it possible to change the export status of an inherited feature. By default — covering the behavior desired in the vast majority of practical cases — an inherited feature keeps its original export status (exported, secret, selectively exported). In some cases, however, this is not appropriate:

- A feature may have played a purely implementation-oriented role in the parent, but become interesting to clients of the heir. Its status will change from secret to exported.

- In implementation inheritance (for example *ARRAYED_LIST* inheriting from *ARRAY*) an exported feature of the parent may not be suitable for direct use by clients of the heir. The change of status in this case is from exported to secret.

You can achieve either of these goals by writing

```
class D inherit
    A
        export {X, Y, …} feature1, feature2, … end
    …
```

This gives a new export status to the features listed (under their final names since, as noted, **export** like all other subclauses comes after **rename** if present): they become exported to the classes listed. In most cases this list of classes, *X*, *Y*, …, consists of just *ANY*, to re-export a previously secret feature, or *NONE*, to hide a previously exported feature. It is also possible, in lieu of the feature list, to use the keyword **all** to apply the new status to all features inherited from the listed parent. Then there can be more than one class-feature list, as in

```
class ARRAYED_LIST [G] inherit
    ARRAY [G]
        rename
            count as capacity, item as array_item, put as array_put
        export
            {NONE} all
            {ANY} capacity
        end
    …
```

where any explicit listing of a feature, such as *capacity*, takes precedence over the export status specified for **all**. Here most features of *ARRAY* are secret in *ARRAYED_LIST*, because the clients should not permitted to manipulate array entries directly: they will manipulate them indirectly through list features such as *extend* and *item*, whose implementation relies on *array_item* and *array_put*. But *ARRAY*'s feature *count* remains useful, under the name *capacity*, to the clients of *ARRAYED_LIST*.

Flat and Flat-Contract Forms

Thanks to inheritance, a concise class text may achieve a lot, relying on all the features inherited from direct and indirect ancestors.

This is part of the power of the object-oriented form of reuse, but can create a comprehension and documentation problem when the inheritance structures become deep: how does one understand such a class, either as client author or as maintainer? For clients, the Contract Form, entirely deduced from the class text, does not tell the full story about available features; and maintainers must look to proper ancestors for much of the relevant information.

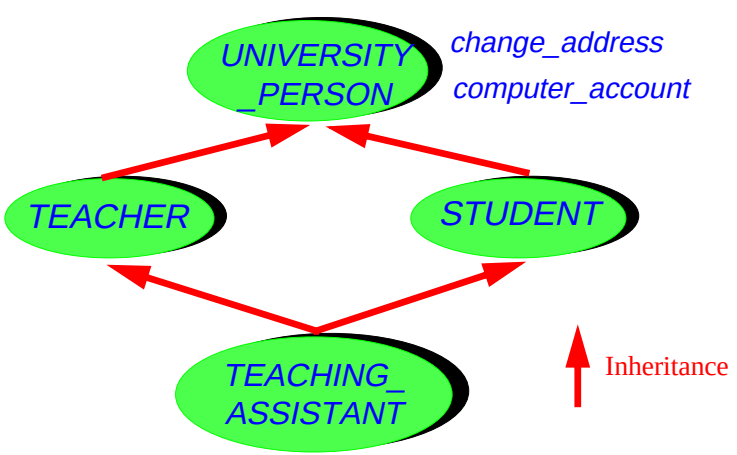
These observations suggest ways to produce, from a class text, a version that is equivalent feature-wise and assertion-wise, but has no inheritance dependency. This is called the **Flat Form** of the class. It is a class text that has no inheritance clause and includes all the features of the class, immediate (declared in the class itself) as well as inherited. For the inherited features, the flat form must of course take account of all the feature adaptation mechanisms: renaming (each feature must appear under its final name), redefinition, effecting, uneffecting and export status change. For redeclared features, **require else** clauses are or-ed with the precursors' preconditions, and **ensure then** clauses are and-ed with precursors' postconditions. For invariants, all the ancestors' clauses are concatenated. As a result, the flat form yields a view of the class, its features and its assertions that conforms exactly to the view offered to clients and (except for polymorphic uses) heirs.

As with the Contract Form ("The contract form of a class", page 44), producing the Flat Form is the responsibility of tools in the development environment. In EiffelStudio, you will just click the "Flat" icon.

The Contract Form of the Flat Form of a class is known as its F**lat-Contract Form**. It gives the complete interface specification, documenting all exported features and assertions — immediate or inherited — and hiding implementation aspects. It is the appropriate documentation for a class.

Repeated inheritance and selection

An inheritance mechanism, following from multiple inheritance, remains to be seen. Through multiple inheritance, a class can be a proper descendant of another through more than one path. This is called repeated inheritance and can be indirect, as in the following figure, or even direct, when a class *D* lists a class *A* twice in its **inherit** clause.



Indirect repeated inheritance

The figure's particular example is in fact often used by introductory presentations of *multiple* inheritance, which is a pedagogical mistake: simple multiple inheritance examples (such as *INTEGER* inheriting from *NUMERIC* and *COMPARABLE*, or *COMPANY_PLANE* from *ASSET* and *PLANE*) should involve the combination of **separate abstractions**. Repeated inheritance is an advanced technique; although invaluable, it does not arise in elementary uses and requires a little more care.

In fact there is only one non-trivial issue in repeated inheritance: what does a feature of the repeated ancestor, such as *change_address* and *computer_account*, mean for the repeated descendant, here *TEACHING_ASSISTANT*? (The example features chosen involve a routine and an attribute; the basic rules will be the same.)

There are two possibilities: sharing (the repeatedly inherited feature yields just one feature in the repeated descendant) and duplication (it yields two). Examination of various cases shows quickly that a fixed policy, or one that would apply to all the features of a class, would be inappropriate.

- Feature *change_address* calls for sharing: as a teaching assistant, you may be both teacher and student, but you are just one person, with just one official domicile.
- If there are separate accounts for students' course work and for faculty, you may need one of each kind, suggesting that *computer_account* calls for duplication.

The Eiffel rule enables, once again, the software developer to craft the resulting class so as to tune it to the exact requirements. Not surprisingly, it is based on names, in accordance with the Final Name rule (no in-class overloading):

Repeated Inheritance rule

- A feature inherited multiply under one name will be shared: it is considered to be just one feature in the repeated descendant.
- A feature inherited multiply under different names will be replicated, yielding as many variants as names.

So to tune the repeated descendant, feature by feature, for sharing and replication it suffices to use renaming.

- Doing nothing will cause sharing, which is indeedthe desired policy in most cases (especially those cases of *unintended* repeated inheritance: making *D* inherit from *A* even though it also inherits from *B*, which you forgot is already a descendant of *A*).

- If you use renaming somewhere along the way, so that the final names are different, you will obtain two separate features. It does not matter where the renaming occurs; all that counts is whether in the common descendant, *TEACHING_ASSISTANT* in the last figure, the names are the same or different. So you can use renaming at that last stage to cause replication; but if the features have been renamed higher you can also use last-minute renaming to *avoid* replication, by bringing them back to a single name.

The Repeated Inheritance rule gives the desired flexibility to disambiguate the meaning of repeatedly inherited features. There remains a problem in case of redeclaration and polymorphism. Assume that somewhere along the inheritance paths one or both of two replicated versions of a feature *f*, such as *computer_account* in the example, has been redeclared; we need to define the effect of a call *a*.*f* (*a*.*computer_account* in the example) if *a* is of the repeated ancestor type, here *UNIVERSITY_PERSON*, and has become attached as a result of polymorphism to an instance of the repeated descendant, here *TEACHING_ASSISTANT*. If one or more of the intermediate ancestors has redefined its version of the feature, the dynamically-bound call has two or more versions to choose from.

A **select** clause will resolve the ambiguity, as in

```
class TEACHING_ASSISTANT inherit
        TEACHER
                rename
                        computer_account as faculty_account
                select
                        faculty_account
                end
        STUDENT
                rename
                        computer_account as student_account
                end
…
```

We assume here that that no other renaming has occurred — *TEACHING_ASSISTANT* takes care of the renaming to ensure replication — but that one of the two parents has redefined *computer_account*, for example *TEACHER* to express the special privileges of faculty accounts. In such a case the rule is that one (and exactly one) of the two parent clauses in *TEACHING_ASSISTANT* **must** select the corresponding version. Note that no problem arises for an entity declared as

```
ta: TEACHING_ASSISTANT
```

since the valid calls are of the form *ta.faculty_account* and *ta.student_account*, neither of them ambiguous; the call *ta.computer_account* would be invalid, since after the renamings class *TEACHING_ASSISTANT* has no feature of that name. The **select** only applies to a call

```
up.computer_account
```

with *up* of type *UNIVERSITY_PERSON*, dynamically attached to an instance of *TEACHING_ASSISTANT*; then the **select** resolves the ambiguity by causing the call to use the version from *TEACHER*.

So if you traverse a list *computer_users*: *LIST* [*UNIVERSITY_PERSON*] to print some information about the computer account of each list element, the account used for a teaching assistant is the faculty account, not the student account.

You may, if desired, redefine *faculty_account* in class *TEACHING_ASSISTANT*, using *student_account* if necessary, to take into consideration the existence of another account. But in all cases we need a precise disambiguation of what *computer_account* means for a *TEACHING_ASSISTANT* object known only through a *UNIVERSITY_PERSON* entity.

The **select** is only needed in case of replication. If the Repeated Inheritance rule would imply sharing, as with change_address, and one or both of the shared versions has been redeclared, the Final Name rule makes the class invalid, since it now has **two different features** with the same name. (This is only a problem if both versions are effective; if one or both are deferred there is no conflict but a mere case of feature joining as explained in "Join and uneffecting", page 70.) The two possible solutions follow from the previous discussions:

- If you do want sharing, one of the two versions must take precedence over the other. It suffices to **undefine** the other, and everything gets back to order. Alternatively, you can redefine both into a new version, which takes precedence over both.
- If you want to keep both versions, switch from sharing to replication: rename one or both of the features so that they will have different names; then you must **select** one of them.

Constrained genericity

Eiffel's inheritance mechanism has an important application to extending the flexibility of the **genericity** mechanism. In a class *SOME_CONTAINER* [*G*], as noted (section 7), the only operations available on entities of type *G*, the formal generic parameter, are those applicable to entities of all types. A generic class may, however, need to assume more about the generic parameter, as with a class *SORTABLE_ARRAY* [*G*…] which will have a procedure *sort* that needs, at some stage, to perform tests of the form

```
if item (i) < item (j) then …
```

where *item* (*i*) and *item* (*j*) are of type *G*. But this requires the availability of a feature **infix** "<" in all types that may serve as actual generic parameters corresponding to *G*. Using the type *SORTABLE_ARRAY* [*INTEGER*] should be permitted, because *INTEGER* has such a feature; but not *SORTABLE_ARRAY* [*COMPLEX*] if there is no total order relation on *COMPLEX*.

To cover such cases, declare the class as

```
class SORTABLE_ARRAY [G –> COMPARABLE]
```

making it **constrained generic**. The symbol –> recalls the arrow of inheritance diagrams; what follows it is a type, known as the generic constraint. Such a declaration means that:

- Within the class, you may apply the features of the generic constraint — here the features of *COMPARABLE*: **infix** "<", **infix** "<=" etc. — to expressions of type *G*.

- A generic derivation is only valid if the chosen actual generic parameter conforms to the constraint. Here you can use *SORTABLE_ARRAY* [*INTEGER*] since *INTEGER* inherits from *COMPARABLE*, but not *SORTABLE_ARRAY* [*COMPLEX*] if *COMPLEX* is not a descendant of *COMPARABLE*.

A class can have a mix of constrained and unconstrained generic parameters, as in the EiffelBase class *HASH_TABLE* [*G*, *H* –> *HASHABLE*] whose first parameter represents the types of objects stored in a hash table, the second representing the types of the keys used to store them, which must be *HASHABLE*. As these examples suggest, structural property classes such as *COMPARABLE*, *NUMERIC* and *HASHABLE* are the most common choice for generic constraints.

Unconstrained genericity, as in *C* [*G*], is defined as equivalent to *C* [*G* –> *ANY*].

Assignment attempt

The Type Conformance rule ("Polymorphism", page 53) ensures type safety by requiring all assignments to be from a more specific source to a more general target.

Sometimes you can't be sure of the source object's type. This happens for example when the object comes from the outside — a file, a database, a network. The persistence storage mechanism("Deep operations and persistence", page 30) includes, along with the procedure *store* seen there, the reverse operation, a function *retrieved* which yields an object structure retrieved from a file or network, to which it was sent using *store*. But *retrieved* as declared in the corresponding class *STORABLE* of EiffelBase can only return the most general type, *ANY*; it is not possible to know its exact type until execution time, since the corresponding objects are not under the control of the retrieving system, and might even have been corrupted by some external agent.

In such cases you cannot trust the declared type but must check it against the type of an actual run-time object. Eiffel introduces for this purpose the **assignment attempt** operation, written

```
x ?= y
```

with the following effect (only applicable if *x* is a writable entity of reference type):

- If *y* is attached, at the time of the instruction's execution to an object whose type conforms to the type of *x*, perform a normal reference assignment.
- Otherwise (if *y* is void, or attached to a non-conforming object), make *x* void.

Using this mechanism, a typical object structure retrieval will be of the form

```
x ?= retrieved
if x = Void then
        "We did not get what we expected"
else
        "Proceed with normal computation, which will typically involve
         calls of the form x.some_feature"
end
```

As another application, assume we have a *LIST* [*ACCOUNT*] and class *SAVINGS_ACCOUNT*, a descendant of *ACCOUNT*, has a feature *interest_rate* which was not in *ACCOUNT*. We want to find the maximum interest rate for savings accounts in the list. Assignment attempt easily solves the problem:

```
local
        s: SAVINGS_ACCOUNT
do
        from account_list.start until account_list.after loop
                s ?= acc_list.item
                        -- item from LIST yields the element at
                        -- cursor position
            if s /= Void and then s.interest_rate > Result then
                        -- Using and then (rather than and) guarantees
                        -- that s.interest_rate is not evaluated
                        -- if s = Void is true.
                    Result := s.interest_rate
            end
            account_list.forth
        end
end
```

Note that if there is no savings account at all in the list the assignment attempt will always yield void, so that the result of the function will be 0, the default initialization.

Assignment attempt is useful in the cases cited — access to external objects beyond the software's own control, and access to specific properties in a polymorphic data structure. The form of the instruction precisely serves these purposes; not being a general type comparison, but only a verification of a specific expected type, it does not carry the risk of encouraging developers to revert to multi-branch instruction structures, for which Eiffel provides the far preferable alternative of polymorphic, dynamically-bound feature calls.

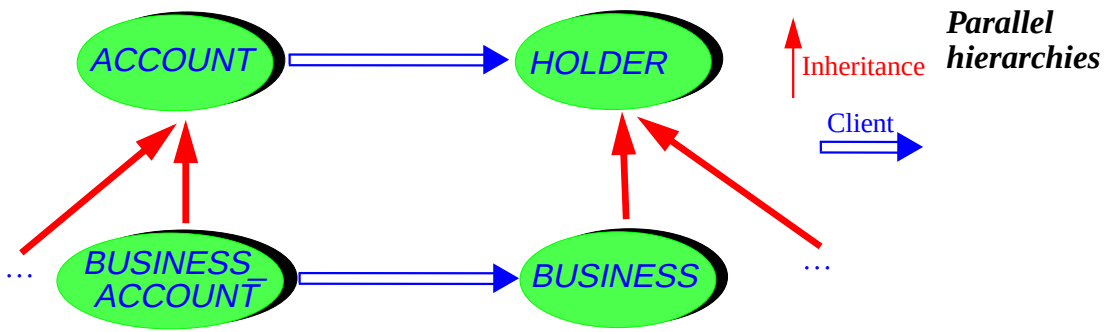
Covariance and anchored declarations

The final property of Eiffel inheritance involves the rules for adapting not only the implementation of inherited features (through redeclaration of either kind, redeclaration and redefinition, as seen so far) and their contracts (through the Assertion Redeclaration rule), but also their types. More general than type is the notion of a feature's **signature**, defined by the number of its arguments, their types, the indication of whether it has a result (that is to say, is a function or attribute rather than a procedure) and, if so, the type of the result.

In many cases the signature of a redeclared feature remains the same as the original's. But in some cases you may want to adapt it to the new class. Assume for example that class *ACCOUNT* has features

```
owner: HOLDER
set_owner (h: HOLDER) is
        -- Make h the account owner.
    require
        not_void: h /= Void
    do
        owner := h
    end
```

We introduce an heir *BUSINESS_ACCOUNT* of *ACCOUNT* to represent special business accounts, corresponding to class *BUSINESS* inheriting from *HOLDER:*



Clearly, we must redefine *owner* in class *BUSINESS_ACCOUNT* to yield a result of type *BUSINESS*; the same signature redefinition must be applied to the argument of *set_owner*. This case is typical of the general scheme of signature redefinition: in a descendant, you may need to redefine both results and arguments to types conforming to the originals. This is reflected by a language rule:

Covariance rule

In a feature redeclaration, both the result type if the feature is a query (attribute or function) and the type of any argument if it is a routine (procedure or function) must conform to the original type as declared in the precursor version.

The term "covariance" reflects the property that all types — those of arguments and those of results — vary together in the same direction as the inheritance structure.

If a feature such as *set_owner* has to be redefined for more than its signature — to update its implementation or assertions — the signature redefinition will be explicit. For example *set_owner* could do more for business owners than it does for ordinary owners. Then the redefinition will be of the form

```
set_owner (b: BUSINESS) is
        -- Make b the account owner.
    … New routine body …
    end
```

In other cases, however, the body will be exactly the same as in the precursor. Then explicit redefinition would be tedious, implying much text duplication. The mechanism of **anchored redeclaration** solves this problem. The original declaration of *set_owner* in *ACCOUNT* should be of the form

```
set_owner (h: like Current) is
        -- Make h the account owner.
        -- The rest as before:
    require
        not_void: h /= Void
    do
        owner := h
    end
```

A **like** *anchor* type, known as an anchored type, may appear in any context in which *anchor* has a well-defined type; *anchor* can be an attribute or function of the enclosing class, or an argument of the enclosing routine. Then, assuming *T* is the type of *anchor*, the type **like** *anchor* means the following:

- In the class in which it appears, **like** *anchor* means the same as *T*. Ffor example, in *set_owner* above, the declaration of *h* has the same effect as if *h* had been declared of type *HOLDER*, the type of the anchor *owner* in class *ACCOUNT*.
- The difference comes in proper descendants: if a type redefinition changes the type of *anchor*, any entity declared **like** *anchor* will be considered to have been redefined too.

This means that anchored declaration are a form of of implicit covariant redeclaration.

In the example, class *BUSINESS_ACCOUNT* only needs to redefine the type of *owner* (to *BUSINESS*). It doesn't have to redefine *set_owner* except if it needs to change its implementation or assertions.

It is possible to use *Current* as anchor; the declaration **like** *Current* denotes a type based on the current class (with the same generic parameters if any). This is in fact a common case; we saw in "Structural property classes", page 62, that it applies in class *COMPARABLE* to features such as

```
infix "<" (other: like Current): BOOLEAN is …
```

since we only want to compare two comparable elements of compatible types — but not, for example, integer and strings, even if both types conform to *COMPARABLE*. (A "balancing rule" makes it possible, however, to mix the various arithmetic types, consistently with mathematical traditions, in arithmetic expressions such as *3 + 45.82* or boolean expressions such as *3 < 45.82*.)

Similarly, class *ANY* declares procedure *copy* as

```
copy (other: like Current) is …
```

with the argument anchored to the current object. Function *clone*, for its part, has signature *clone* (*other*: *ANY*): **like** *other*, with both argument and result anchored to the argument, so that for any *x* the type of *clone* (*x*) is the same as the type of *x*.

A final, more application-oriented example of anchoring to *Current* is the feature *merge* posited in an earlier example (page 33) with the signature *merge* (*other*: *ACCOUNT*). By using instead *merge* (*other*: **like** *Current*) we can ensure that in any descendant class — *BUSINESS_ACCOUNT*, *SAVINGS_ACCOUNT*, *MINOR_ACCOUNT*… — an account will only be mergeable with another of a compatible type.

Covariance makes static type checking more delicate; mechanisms of "system validity" and "catcalls" address the problem, discussed in detail in the book Object-Oriented Software Construction (see the bibliography).

.

10 OTHER MECHANISMS

We now examine a few important mechanisms that complement the preceding picture: shared objects; constants; instructions; and lexical conventions.

Once routines and shared objects

The Eiffel's method obsession with extendibility, reusability and maintainability yields, as has been seen, modular and decentralized architectures, where inter-module coupling is limited to the strictly necessary, interfaces are clearly delimited, and all the temptations to introduce obscure dependencies, in particular global variables, have been removed. There is a need, however, to let various components of a system access common objects, without requiring their routines to pass these objects around as arguments (which would only be slightly better than global variables). For example various classes may need to perform output to a common "console window", represented by a shared object.

Eiffel addresses this need through an original mechanism that also takes care of another important issue, poorly addressed by many design and programming approaches: initialization. The idea is simple: if instead of **do** the implementation of an effective routine starts with the keyword **once**, it will only be executed the first time the routine is called during a system execution (or, in a multithreaded environment, the first time in each thread), regardless of what the caller was. Subsequent calls from the same caller or others will have no effect; if the routine is a function, it will always return the result computed by the first call — object if an expanded type, reference otherwise.

In the case of procedures, this provides a convenient initialization mechanism. A delicate problem in the absence of a **once** mechanism is how to provide the users of a library with a set of routines which they can call in any order, but which all need, to function properly, the guarantee that some context had been properly set up. Asking the library clients to precede the first call with a call to an initialization procedure *setup* is not only user-unfriendly but silly: in a well-engineered system we will want to check proper set-up in every of the routines, and report an error if necessary; but then if we were able to detect improper set-up we might as well shut up and set up ourselves (by calling *setup*). This is not easy, however, since the object on which we call *setup* must itself be properly initialized, so we are only pushing the problem further. Making *setup* a **once** procedure solves it: we can simply include a call

```
setup
```

at the beginning of each affected routine; the first one to come in will perform the needed initializations; subsequent calls will have, as desired, no effect.

Once functions will give us shared objects. A common scheme is

```
console: WINDOW is
        -- Shared console window
    once
        create Result.make (…)
    end
```

Whatever client first calls this function will create the appropriate window and return a reference to it. Subsequent calls, from anywhere in the system, will return that same reference. The simplest way to make this function available to a set of classes is to include it in a class *SHARED_STRUCTURES* which the classes needing a set of related shared objects will simply inherit.

For the classes using it, *console*, although a function, looks very much as if it were an attribute — only one referring to a shared object.

The "Hello World" system at the beginning of this discussion (section 4) used an output instruction of the form *io.put_string* ("*Some string*"). This is another example of the general scheme illustrated by *console*. Feature *io*, declared in *ANY* and hence usable by all classes, is a once function that returns an object of type *STANDARD_FILES* (another Kernel Library class) providing access to basic input and output features, one of which is procedure *put_string*. Because basic input and output must all work on the same files, *io* should clearly be a once function, shared by all classes that need these mechanisms.

Constant and unique attributes

The attributes studied earlier were variable: each represents a field present in each instance of the class and changeable by its routines.

It is also possible to declare constant attributes, as in

```
Solar_system_planet_count: INTEGER is 9
```

These will have the same value for every instance and hence do not need to occupy any space in objects at execution time. (In other approaches similar needs would be addressed by symbolic constants, as in Pascal or Ada, or macros, as in C.)

What comes after the **is** is a manifest constant: a self-denoting value of the appropriate type. Manifest constants are available for integers, reals (also used for doubles), booleans (*True* and *False*), characters (in single quotes, as *'A'*, with special characters expressed using a percent sign as in *'%N'* for new line, *'%B'* for backspace and *'%U'* for null).

For integer constants, it is also possible to avoid specifying the values. A declaration of the form

```
a, b, c, … n: INTEGER is unique
```

introduces *a*, *b*, *c*, … *n* as constant integer attributes, whose value are assigned by the Eiffel compiler rather than explicitly by the programmer. The values are different for all **unique** attributes in a system; they are all positive, and, in a single declaration such as the above, guaranteed to be consecutive (so that you may use an invariant property of the form *code* >= *a* **and** *code* <= *n* to express that *code* should be one of the values). This mechanism replaces the "enumerated types" found in many anguages, without suffering from the same problems. (Enumerated types have an ill-defined place in the type system; and it is not clear what operations are permitted.)

You may use Unique values in conjunction with the **inspect** multi-branch instruction studied in the next section. They are only appropriate for codes that can take on a fixed number of well-defined values — not as a way to program operations with many variants, a need better addressed by the object-oriented technique studied earlier and relying on inheritance, polymorphism, redeclaration and dynamic binding.

Manifest constants are also available for strings, using double quotes as in

```
User_friendly_error_message: INTEGER is "Go get a life!"
```

with special characters again using the % codes. It is also possible to declare manifest arrays using double angle brackets:

```
<<1, 2, 3, 5, 7, 11, 13, 17, 19>>
```

which is an expression of type *ARRAY* [*INTEGER*]. Manifest arrays and strings are not atomic, but denote instances of the Kernel Library classes *STRING* and *ARRAY*, as can be produced by once functions.

Instructions

Eiffel has a remarkably small set of instructions. The basic computational instructions have been seen: creation, assignment, assignment attempt, procedure call, **retry**. They are complemented by control structures: conditional, multi-branch, loop, as well as **debug** and **check**.

A conditional instruction has the form **if** … **then** … **elseif** … **then** … **else** … **end**. The **elseif** … **then** … part (of which there may be more than one) and the **else** … part are optional. After **if** and **elseif** comes a boolean expression; after **then**, **elseif** and **else** come zero or more instructions.

A multi-branch instruction has the form

```
inspect
        exp
when v1 then
        inst1
when v2 then
        inst2
…
else
        inst0
end
```

where the **else** $inst_0$ part is optional, *exp* is a character or integer expression, v_1, v_2, … are constant values of the same type as *exp*, all different, and $inst_0$, $inst_1$, $inst_2$, … are sequences of zero or more instructions. In the integer case, it is often convenient to use **unique** values ("Constant and unique attributes", page 83) for the v_i.

The effect of such a multi-branch instruction, if the value of *exp* is one of the v_i, is to execute the corresponding $inst_i$. If none of the v_i matches, the instruction executes $inst_0$, unless there is no **else** part, in which case it triggers an exception.

Raising an exception is the proper behavior, since the absence of an **else** indicates that the author asserts that one of the values will match. If you want an instruction that does nothing in this case, rather than cause an exception, use an **else** part with an empty $inst_0$. In contrast, **if** *c* **then** *inst* **end** with no **else** part does nothing in the absence of an **else** part, since in this case there is no implied claim that *c* must hold.)

The loop construct has the form

```
from
        initialization
until
        exit
invariant
        inv
variant
        var
loop
        body
end
```

where the **invariant** *inv* and **variant** *var* parts are optional, the others required. *initialization* and *body* are sequences of zero or more instructions; *exit* and *inv* are boolean expressions (more precisely, *inv* is an assertion); *var* is an integer expression.

The effect is to execute *initialization*, then, zero or more times until *exit* is satisfied, to execute *body*. (If after *initialization* the value of *exit* is already true, *body* will not be executed at all.) Note that the syntax of loops always includes an initialization, as most loops require some preparation. If not, just leave *initialization* empty, while including the **from** since it is a required component.

The assertion *inv*, if present, expresses a **loop invariant** (not to be confused with class invariants). For the loop to be correct, *initialization* must ensure *inv*, and then every iteration of *body* executed when *exit* is false must preserve the invariant; so the effect of the loop is to yield a state in which both *inv* and *exit* are true. The loop must terminate after a finite number of iterations, of course; this can be guaranteed by using a **loop variant** *var*. It must be an integer expression whose value is non-negative after execution of *initialization*, and decreased by at least one, while remain non-negative, by any execution of *body* when *exit* is false; since a non-negative integer cannot be decreased forever, this ensures termination. The assertion monitoring mode, if turned on at the highest level, will check these properties of the invariant and variant after initialization and after each loop iteration, triggering an exception if the invariant does not hold or the variant is negative or does not decrease.

An occasionally useful instruction is **debug** (*Debug_key, …*) *instructions* **end** where *instructions* is a sequence of zero or more instructions and the part in parentheses is optional, containing if present one or more strings, called debug keys. The EiffelStudio compiler lets you specify the corresponding **debug** compilation option: **yes**, **no**, or an explicit debug key. The *instructions* will be executed if and only if the corresponding option is on. The obvious use is for instructions that should be part of the system but executed only in some circumstances, for example to provide extra debugging information.

The final instruction is connected with Design by Contract. The instruction **check** *Assertions* **end**, where *Assertions* is a sequence of zero or more assertions, will have no effect unless assertion monitoring is turned on at the **Check** level or higher. If so it will evaluate all the assertions listed, having no further effect if they are all satisfied; if any one of them does not hold, the instruction will trigger an exception.

This instruction serves to state properties that are expected to be satisfied at some stages of the computation — other than the specific stages, such as routine entry and exit, already covered by the other assertion mechanisms such as preconditions, postconditions and invariants. A recommended use of **check** involves calling a routine with a precondition, where the call, for good reason, does not explicitly test for the precondition. Consider a routine of the form

```
r (ref: SOME_REFERENCE_TYPE) is
        require
                not_void: r /= Void
        do
                r.some_feature
                …
        end
```

Because of the call to *some_feature*, the routine will only work if its precondition is satisfied on entry. To guarantee this precondition, the caller may protect it by the corresponding test, as in

```
if x /= Void then a.r (x) end
```

but this is not the only possible scheme; for example if an **create** *x* appears shortly before the call we know *x* is not void and do not need the protection. It is a good idea in such cases to use a **check** instruction to document this property, if only to make sure that a reader of the code will realize that the omission of an explicit test (justified or not) was not a mistake. This is particularly appropriate if the justification for not testing the precondition is less obvious. For example *x* could have been obtained, somewhere else in the algorithm, as *clone* (*y*) for some *y* that you know is not void. You should document this knowledge by writing the call as

```
        check
                x_not_void: x /= Void end
                        -- Because x was obtained as a clone of y,
                        -- and y is not void because [etc.]
        end
a.r (x)
```

Note the recommended convention: extra indentation of the **check** part to separate it from the algorithm proper; and inclusion of a comment listing the rationale behind the developer's decision not to check explicitly for the precondition.

In production mode with assertion monitoring turned off, this instruction will have no effect. But it will be precious for a maintainer of the software who is trying to figure out what it does, and in the process to reconstruct the original developer's reasoning. (The maintainer might of course be the same person as the developer, six months later.) And if the rationale is wrong somewhere, turning assertion checking on will immediately uncover the bug.

Obsolete features and classes

One of the conditions for producing truly great reusable software is to recognize that although you should try to get everything right the first time around you won't always succeed. But if "good enough" may be good enough for application software, it's not good enough, in the long term, for reusable software. The aim is to get ever closer to the asymptote of perfection. If you find a better way, you must implement it. The activity of *generalization*, discussed as part of the lifecycle, doesn't stop at the first release of a reusable library.

This raises the issue of backward compability: how to move forward with a better design, without compromising existing applications that used the previous version?

The notion of obsolete class and feature helps address this issue. By declaring a feature as **obsolete**, using the syntax

```
enter (i: INTEGER; x: G) is
        obsolete
                "Use 'put (x, i)' instead"
        require
                ...
        do
                put (x, i)
        end
```

you state that you are now advising against using it, and suggest a replacement through the message that follows the keyword **obsolete**, a mere string. The obsolete feature is still there, however; using it will cause no other harm than a warning message when someone compiles a system that includes a call to it. Indeed, you don't want to hold a gun to your client authors' forehead ("*Upgrade now or die*!'); but you do want to let them know that there is a new version and that they should upgrade at their leisure.

Besides routines, you may also mark classes as obsolete.

The example above is a historical one, involving an early change of interface for the EiffelBase library class *ARRAY*; the change affected both the feature's name, with a new name ensuring better consistency with other classes, and the order of arguments, again for consistency. It shows the recommended style for using **obsolete**:

- In the message following the keyword, explain the recommended replacement. This message will be part of the warning produced by the compiler for a system that includes the obsolete element.
- In the body of the routine, it is usually appropriate, as here, to replace the original implementation by a call to the new version. This may imply a small performance overhead, but simplifies maintenance and avoids errors.

It is good discipline not to let obsolete elements linger around for too long. The next major new release, after a suitable grace period, should remove them.

The design flexibility afforded by the **obsolete** keyword is critical to ensure the harmonious long-term development of ambitious reusable software.

Creation variants

The basic forms of creation instruction, and the one most commonly used, are the two illustrated earlier ("Creating and initializing objects", page 20):

```
create x.make (2000)
create x
```

the first one if the corresponding class has a **create** clause, the second one if not. In either form you may include a type name in braces, as in

```
create {SAVINGS_ACCOUNT} x.make (2000)
```

which is valid only if the type listed, here *SAVINGS_ACCOUNT*, conforms to the type of *x*, assumed here to be *ACCOUNT*. This avoids introducing a local entity, as in

```
local
    sx: SAVINGS_ACCOUNT
do
    create xs.make (2000)
    x := xs
        …
```

and has exactly the same effect. Another variant is the **creation expression**, which always lists the type, but returns a value instead of being an instruction. It is useful in the followingcontext:

```
some_routine (create {ACCOUNT}.make (2000))
```

which you may again view as an abbreviation for a more verbose form that would need a local entity, using a creation instruction:

```
local
        x: ACCOUNT
do
        create x.make (2000)
        some_routine (x)
                ...
```

Unlike creation instructions, creation expressions must always list the type explicitly, {*ACCOUNT*} in the example. They are useful in the case shown: creating an object that only serves as an argument to be passed to a routine. If you need to retain access to the object through an entity, the instruction **create** *x*... is the appropriate construct.

The creation mechanism gets an extra degree of flexibility through the notion of *default_create*. The simplest form of creation instruction, **create** *x* without an explicit creation procedure, is actually an abbreviation for **create** *x*.*default_create*, where *default_create* is a procedure defined in class *ANY* to do nothing. By redefining *default_create* in one of your classes, you can ensure that **create** *x* will take care of non-default initialization (and ensure the invariant if needed). When a class has no **create** clause, it's considered to have one that lists only *default_create*. If you want to allow **create** *x* as well as the use of some explicit creation procedures, simply list *default_create* along with these procedures in the **create** clause. To disallow creation altogether, include an empty **create** clause, although this technique is seldom needed since most non-creatable classes are deferred, and one can't instantiate a deferred class.

One final twistis the mechanism for creating instances of formal generic parameters. For *x* of type *G* in a class C [*G*], it wouldn't be safe to allow **create** *x*, since *G* stands for many possible types, all of which may have their own creation procedures. To allow such creation instructions, we rely on constrained genericity. You may declare a class as

```
[G -> T create cp end]
```

to make *G* constrained by *T*, as we learned before, and specify that any actual generic parameter must have *cp* among its creation procedures. Then it's permitted to use **create** *x*.*cp*, with arguments if required by *cp*, since it is guaranteed to be safe. The mechanism is very general since you may use *ANY* for *T* and *default_create* for *cp*. The only requirement on *cp* is that it must be a *procedure* of *T*, not necessarily a *creation* procedure; this permits using the mechanism even if *T* is deferred, a common occurrence. It's only descendants of *T* that must make *cp* a creation procedure, by listing it in the **create** clause, if they want to serve as actual generic parameters for *C*.

Tuple types

The study of genericity described arrays. Another common kind of container objects bears some resemblance to arrays: sequences, or "tuples", of elements of specified types. The difference is that all elements of an array were of the same type, or a conforming one, whereas for tuples you will specify the types we want for each relevant element. A typical tuple type is of the form

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

denoting a tuple of least three elements, such that the type of the first conforms to *X*, the second to *Y*, and the third to *Z*.

You may list any number of types in brackets, including none at all: *TUPLE*, with no types in brackets, denotes tuples of arbitrary length.

> The syntax, with brackets, is intentionally reminiscent of generic classes, but *TUPLE* is a reserved word, not the name of a class; making it a class would not work since a generic class has a fixed number of generic parameters. You may indeed use *TUPLE* to obtain the effect of a generic class with a variable number of parameters.

To write the tuples themselves — the sequences of elements, instances of a tuple type — you will also use square brackets; for example

```
[x1, y1, z1]
```

with *x1* of type *X* and so on is a tuple of type *TUPLE* [*X*, *Y*, *Z*].

The definition of tuple types states that *TUPLE* [*X1*, … , *Xn*] denotes sequences of *at least n* elements, of which the first *n* have types respectively conforming to *X1*, … , *Xn*. Such a sequence may have more than *n* elements.

Features available on tuple types include *count*: *INTEGER*, yielding the number of elements in a tuple, *item* (*i: INTEGER*): *ANY* which returns the *i*-th element, and *put* which replaces an element.

Tuples are appropriate when these are the only operations you need, that is to say, you are using sequences with no further structure or properties. Tuples give you "anonymous classes" with predefined features *count*, *item* and *put*. A typical example is a general-purpose output procedure that takes an arbitrary sequence of values, of arbitrary types, and prints them. It may simply take an argument of type *TUPLE*, so that clients can call it under the form

```
write ([your_integer, your_real, your_account])
```

As soon as you need a type with more specific features, you should define a class.

11 AGENTS

Our last mechanism, agents, adds one final level of expressive power to the framework describe so far. Agents apply object-oriented concepts to the modeling of *operations*.

Objects for operations

Operations are not objects; in fact, object technology starts from the decision to separate these two aspects, and to choose object types, rather than the operations, as the basis for modular organization of a system, attaching each operation to the resulting modules — the classes.

In a number of applications, however, we may need objects that *represent* operations, so that we can include them in object structures that some other piece of the software will later traverse to uncover the operations and, usually, execute them. Such "operation wrapper" objects, called agents, are useful in a number of application areas such as:

- GUI (Graphical User Interface) programming, where we may associate an agent with a certain event of the interface, such as a mouse click at a certain place on the screen, to prescribe that if the event occurs — a user clicks there — it must cause execution of the agent's associated operation.
- Iteration on data structures, where we may define a general-purpose routine that can apply an arbitrary operation to all the elements of a structure such as a list; to specify a particular operation to iterate, we will pass to the iteration mechanism an agent representing that operation.
- Numerical computation, where we may define a routine that computes the integral of any applicable function on any applicable interval; to represent that function and pass its representation to the integration routine, we will use an agent.

Operations in Eiffel are expressed as routines, and indeed every agent will have an associated routine. Remember, however, that the fundamental distinction between objects and operations remains: an agent is an object, and it is not a routine; it *represents* a routine. As further evidence that this is a proper data abstraction, note that the procedure *call*, available on all agents to call the associated routine, is only one of the features of agents. Other features may denote properties such as the class to which the routine belongs, its precondition and postcondition, the result of the last call for a function, the number of arguments.

Building an agent

In the simplest form, also one of the most common, you obtain an agent just by writing

```
agent r
```

where *r* is the name of a routine of the enclosing class. This is an expression, which you may assign to a writable entity, or pass as argument to a routine. Here for example is how you will specify event handling in the style of the EiffelVision 2 GUI library:

```
your_icon.click_actions.extend (agent your_routine)
```

This adds to the end of *my_icon.click_actions* — the list of agents associated with the "click" event for *my_icon*, denoting an icon in the application's user interface — an agent representing *your_routine*. Then when a user clicks on the associated icon at execution, the EiffelVision 2 mechanisms will call the procedure *call* on every agent of the list, which for this agent will execute *your_routine*. This is a simple way to associate elements of your application, more precisely its "business model" (the processing that you have defined, directly connected to the application's business domain), with elements of its GUI.

Similarly although in a completely different area, you may request the integration of a function *your_function* over the interval `0 ..1`through a call such as

```
your_integrator.integral (agent your_function, 0, 1)
```

In the third example area cited above, you may call an iterator of EiffelBase through

```
your_list.do_all (agent your_proc)
```

with *your_list* of a type such as *LIST* [*YOUR_TYPE*]. This will apply *your_proc* to every element of the list in turn.

The agent mechanism is type-checked like the rest of Eiffel; so the last example is valid if and only if *your_proc* is a procedure with one argument of type *YOUR_TYPE*.

Operations on agents

An agent **agent** *r* built from a procedure *r* is of type *PROCEDURE* [*T, ARGS*] where *T* represents the class to which *r* belongs and *ARGS* the type of its arguments. If *r* is a function of result type *RES*, the type is *FUNCTION* [*T, ARGS, RES*]. Classes *PROCEDURE* and *FUNCTION* are from the Kernel Library of EiffelBase, both inheriting from *ROUTINE* [*T, ARGS*].

Among the features of *ROUTINE* and its descendants the most important are *call*, already noted, which calls the associated routine, and *item*, appearing only in *FUNCTION* and yielding the result of the associated function, which it obtains by calling *call*.

As an example of using these mechanisms, here is how the function *integral* could look like in our *INTEGRATOR* example class. The details of the integration algorithm (straightforward, and making no claims to numerical sophistication) do not matter, but you see, in the highlighted line, the place were we evaluate the mathematical function associated with *f*, by calling *item* on *f*:

```
integral
        (f: FUNCTION [ANY, TUPLE [REAL], REAL];
        low, high: REAL): REAL is
                -- Integral of f over the interval [low, high]
        require
                meaningful_interval: low <= high
        local
                x: REAL
        do
                from
                        x := low
                invariant
                        x >= low ; x <= high + step
                        -- Result approximates the integral over
                        -- the interval [low, low.max (x – step)]
                until x > high loop
                        Result := Result + step * f.item ([x])
                        x := x + step
                end
        end
```

Function *integral* takes three arguments: the agent *f* representing the function to be integrated, and the two interval bounds. When we need to evaluate that function for the value *x*, in the line

```
Result := Result + step * f.item ([x])
```

we don't directly pass *x* to *item*; instead, we pass a one-element tuple [*x*], using the syntax for manifest tuples introduced in "Tuple types", page 91. You will always use tuples for the argument to *call* and *item*, because these features must be applicable to *any* routine, and so cannot rely on a fixed number of arguments. Instead they take a single tuple intended to contain all the arguments. This property is reflected in the type of the second actual generic parameter to *f*, corresponding to *ARGS* (the formal generic parameter of *FUNCTION*): here it's *TUPLE* [*REAL*] to require an argument such as [*x*], where *x* is of type *REAL*.

Similarly, consider the agent that the call seen above:

```
your_icon.click_actions.extend (agent your_routine)
```

added to an EiffelVision list. When the EiffelVision mechanism detects a mouse click event, it will apply to each element *item* of the list of agents, *your_icon.click_actions*, an instruction such as

```
item.call ([x, y])
```

where *x* and *y* are the coordinates of the mouse clicking position. If *item* denotes the list element **agent** *your_routine*, inserted by the above call to *extend*, the effect will be the same as that of calling

```
your_routine (x, y)
```

assuming that *your_routine* indeed takes arguments of the appropriate type, here *INTEGER* representing a coordinate in pixels. (Otherwise type checking would have rejected the call to *extend*.)

Open and closed arguments

In the examples so far, execution of the agent's associated routine, through *item* or *call*, passed exactly the arguments that a direct call to the routine would expect. You can have more flexibility. In particular, you may build an agent from a routine with more arguments than expected in the final call, and you may set the values of some arguments at the time you define the agent.

Assume for example that a cartographical application lets a user record the location of a city by clicking on the corresponding position on the map. The application may do this through a procedure

```
record_city (cn: STRING; x, y: INTEGER; pop: INTEGER)
        -- Record that the city of name name is at coordinates
        -- x and y with population pop.
```

Then you can associate it with the GUI through a call such as

```
map.click_actions.extend (agent record_city (name, population, ?, ?))
```

assuming that the information on the *name* and the *population* has already been determined. What the agent denotes is the same as **agent** *your_routine* as given before, where *your_routine* would be a fictitious two-argument routine obtained from *record_city* — a four-argument routine — by setting the first two arguments once and for all to the values given, *name* and *population*.

In the agent **agent** *record_city* (*name*, *population*, **?**, **?**), we say that these first two arguments, with their set values, are **closed**; the last two are **open**. The question mark syntax introduced by this example may only appear in agent expressions; it denotes open arguments. This means, by the way, that you may view the basic form used in the preceding examples, **agent** *your_routine*, as an abbreviation — assuming *your_routine* has two arguments — for **agent** *your_routine* (**?**, **?**). It is indeed permitted, to define an agent with all arguments open, to omit the argument list altogether; no ambiguity may result.

For type checking, **agent** *record_city* (*name*, *population*, **?**, **?**) and **agent** *your_routine* are acceptable in exactly the same situations, since both represent routines with two arguments. The type of both is

```
PROCEDURE [ANY, TUPLE [INTEGER, INTEGER]]
```

where the tuple type specifies the open operands.

A completely closed agent, such as **agent** *your_routine* (*25*, *32*) or **agent** *record_city* (*name*, *population*, *25*, *32*), has the type *TUPLE*, with no parameters; you will call it with *call* ([ ]), using an empty tuple as argument.

The freedom to start from a routine with an arbitrary number of arguments, and choose which ones you want to close and which ones to leave open, provides a good part of the attraction of the agent mechanism. It means in particular that in GUI applications you can limit to the strict minimum the "glue" code (sometimes called the *controller* in the so-called MVC, Model-View Controller, scheme of GUI design) between the user interface and "business model" parts of a system. A routine such as *record_city* is a typical example of an element of the business model, uninfluenced — as it should be — by considerations of user interface design. Yet by passing it in the form of an agent with partially open and partially closed arguments, you may be able to use it *directly* in the GUI, as shown above, without any "controller" code.

As another example of the mechanism's versatility, we saw above an integral function that could integrate a function of one variable over an interval, as in

```
your_integrator.integral (agent your_function, 0, 1)
```

Now assume that *function3* takes three arguments. To integrate *function3* with two arguments fixed, you don't need a new *integral* function; just use the same *integral* as before, judiciously selecting what to close and what to leave open:

```
your_integrator.integral (agent function3 (3.5, ?, 6.0), 0, 1)
```

Open targets

All the agent examples seen so far were based on routines of the enclosing class. This is not required. Feature calls, as you remember, were either unqualified, as in *f* (*x*, *y*), or qualified, as in *a*.*g* (*x*, *y*). Agents, too, have a qualified variant as in

```
agent a.g
```

which is closed on its target *a* and open on the arguments. Variants such as **agent** *a*.*g* (*x*, *y*), all closed, and **agent** *a*.*g* (**?**, *y*), open on one argument, are all valid.

You may also want to make the *target* open. The question mark syntax could not work here, since it wouldn't tell us the class to which feature *g* belongs, known in the preceding examples from the type of *a*. As in creation expressions, we must list the type explicitly; the convention is the same: write the types in braces, as in

```
agent {SOME_TYPE}.g
agent {SOME_TYPE}.g (?, ?)
agent {SOME_TYPE}.g (?, y)
```

The first two of these examples are open on the target and both operands; they mean the same. The third is closed on one argument, open on the other and on the target.

These possibilities give even more flexibility to the mechanism because they mean that an operation that needs agents with certain arguments open doesn't care whether they come from an argument or an operand of the original routine. This is particularly useful for iterators and means that if you have two lists

```
your_account_list: LIST [ACCOUNT]
your_integer_list: LIST [INTEGER]
```

you may write both

```
your_acccount_list.do_all (agent deposit_one_grand)
your_integer_list.do_all  (agent add_to_n)
```

even though the two procedures used in the agents have quite different forms. We are assuming here that the first one, in class *ACCOUNT*, is something like

```
deposit_one_grand is
        -- Add one thousand dollars to balance of account.
    do balance := balance + 1000 end
```

so that it doesn't take an argument: it is normally called on its target, as in *my_account.deposit_one_grand*. In contrast, the other routine has an argument:

```
add_to_n (x: INTEGER) is
        -- Add x to the value of total.
    do total := total + x end
```

where *total* is an integer attribute of the enclosing class. Without the versatility of playing with open and closed arguments for both the original arguments and target, you would have to write separate iteration mechanisms for these two cases. Here you can use a single iteration routine of *LIST* and similar classes of EiffelBase, *do_all*, for both purposes:

- Depositing money on every account in a list of accounts.
- Adding all the integers in a list of integers.

Agents provide a welcome complement to the other mechanisms of Eiffel. They do not conflict with them but, when appropriate — as in the examples sketched in this section — provide clear and expressive programming schemes, superior to the alternatives.

12 LEXICAL CONVENTIONS AND STYLE RULES

Eiffel software texts are free-format: distribution into lines is not semantically significant, and any number of successive space and line-return characters is equivalent to just one space. The style rules suggest indenting software texts as illustrated by the examples in this chapter.

Successive declarations or instructions may be separated by semicolons. Eiffel's syntax has been so designed, however, that (except in rare cases) **the semicolon is optional**. Omitting semicolons for elements appearing on separate lines lightens text and is the recommended practice since semicolons, as used by most programming languages, just obscure the text by distracting attention from the actual contents. *Do* use semicolons if you occasionally include successive elements on a single line.

63 names — all unabbreviated single English words, except for **elseif** which is made of two words — are reserved, meaning that you cannot use them to declare new entities. Here is the list:

Since this tutorial has covered all the essential mechanisms, you may ignore the keywords not encountered; they are reserved for future use.

agent	**alias**	**all**	**and**	**as**	**assign**	**check**
class	**convert**	**create**	*Current*	**debug**	**deferred**	**do**
else	**elseif**	**end**	**ensure**	**expanded**	**export**	**external**
False	**feature**	**from**	**frozen**	**if**	**implies**	**indexing**
infix	**inherit**	**inspect**	**invariant**	**is**	**like**	**local**
loop	**not**	**obsolete**	**old**	**once**	**or**	**prefix**
Precursor	**pure**	**redefine**	**reference**	**rename**	**require**	**rescue**
Result	**retry**	**separate**	**then**	*True*	*TUPLE*	**undefine**

Most of the reserved words are keywords, serving only as syntactic markers, and written in boldface in typeset texts such as the present one: **class**, **feature**, **inherit**. The others, such as *Current*, directly carry a semantic denotation; they start with an upper-case letter and are typeset in boldface.

These conventions about letter case are only style rules. Eiffel is case-insensitive, since it is foolish to assume that two identifiers denote two different things just on the basis of a letter written in lower or upper case. The obvious exception is manifest character constants (appearing in single quotes, such as '*A*') and manifest character strings (appearing in double quotes, such as "*lower and UPPER*").

The style rules, however, are precise, and any serious Eiffel project will enforce them; the tools of EiffelStudio also observe them in the texts they output (although they will not mess up with your source text unless you ask them to reformat it). Here are the conventions, illustrated by the examples of this tutorial:

- Class names in upper case, as *ACCOUNT*.
- Non-constant feature names and keywords in lower case, as *balance* and **class**.
- Constant features and predefined entities and expressions with an initial upper case, as *Avogadro* and *Result*.

In typeset documents including Eiffel texts, the standard for font styles is also precise. You should use **boldface** for keywords and *italics* for all other Eiffel elements. Comments, however, are typeset in roman. This lets a software element, such as an identifier, stand out clearly in what is otherwise a comment text expressed in English or another human langage, as in the earlier example

```
-- Add sum to account.
```

which makes clear that *sum* is a software element, not the English word.

There is also an Eiffel style to the choice of identifiers. For features, stay away from abbreviations and use full words. In multi-word identifiers, separate the constituents by underscores, as in *LINKED_LIST* and *set_owner*. The competing style of no separation but mid-identifier upper-case, as in linkedList or setOwner, is less readable and not in line with standard Eiffel practices.

Features of reusable classes should use consistent names. A set of standard names — *put* for the basic command to add or replace an element, *count* for the query that returns the number of element in a structure, *item* to access an element — is part of the style rules, and used systematically in EiffelBase. Use them in your classes too.

For local entities and formal arguments of routines, , it is all right to use abbreviated names, since these identifiers only have a local scope, and choosing a loud name would give them too much pretense, leading to potential conflicts with features.

The complete set of style rules applied by ISE is available on the web in both HTML and PDF forms. These rules are an integral part of the Eiffel method; in quality software, there is no such thing as a detail. Applying them systematically promotes consistency between projects in the Eiffel world, enhances reusability, and facilitates everyone's work.

13 TO LEARN MORE

Beyond this introduction, you will find the following two books essential to a masteryof the method and language:

- *Object-Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 2nd edition 1997. (Make sure to get the second edition.) About object technology in general; presents the method behind Eiffel.

- *Eiffel: The Language*, Bertrand Meyer, Prentice Hall, 1992. Language manual and reference.

Numerous other books are available on Eiffel and Eiffel-related topics. See an extensive list at http://www.eiffel.com/doc/documentation.html, from which you can order most of the titles listed. They include university textbooks, general introductions, presentations of Eiffel projects, descriptions of libraries and other applications, books on BON and object-oriented methodology.