The clusters themselves are specified in the last clause, ***cluster***, here listing only two clusters, each with a cluster name (*my_cluster1*, *her_cluster2*) and the associated directory in double quotes. The cluster name, an identifier, will be used to refer to the cluster elsewhere in the Ace, for example here in the reference to *my_cluster1* appearing in the ***root*** clause as seen above. A system may have as many clusters as desired.

The ***default*** clause serves to specify compilation options. The above Ace relies on the EiffelBase library, precompiled; hence the ***precompiled*** clause, which specifies the standard directory for storing this library, as explained in more detail in an earlier chapter ("PRECOMPILATION", 6.5, page 110). Because of the presence of EiffelBase, with its dozens of clusters, a system with such an Ace as above might actually be a significant system even with just two clusters of its own.

Precompilation is explained in "PRECOMPILATION", 6.5, page 110.

The other compilation option specified above is ***assertion*** (***ensure***) which states that the generated code must at run time monitor assertions up to the postcondition (***ensure***) level. The possible levels include ***require*** (preconditions only, the default), ***ensure*** and ***invariant***; each of them implies the previous ones.

You can also specify various levels of assertion monitoring separately for a cluster, or for a specific class. For details, see the full description of Lace: appendix D of *Eiffel*: *The Language*.

names to 8 characters (plus the extension) you may have to truncate that name, for example by storing a class *BINARY_SEARCH_TREE* into a file called *binstree.e*.

The Eiffel method also suggests grouping related classes — typically 5 to 40 classes — into collections called **clusters**. Since each class is stored in a file, it is natural to associate each cluster with a directory.

These observations yield the mechanism that you will use to specify a universe (item 1 in the above list): you specify a list of directories, which yields a list of clusters; the classes contained in the ***.e*** files of the directories constitute the universe.

The classes of a system will include its root class and all the classes that it **needs** directly or indirectly, where a class is said to need another if it is one of its heirs or clients.

To specify a system you will need to state, in addition to the list of directories, the name of the root class (which must be one of the classes of the universe) and of the root creation procedure (which must be one of the creation procedures of the root class).

Here is an example of such a system specification. This text is **not** written in Eiffel, although it definitely has an Eiffel-like flavor. It is called an Ace (for Assembly of Classes in Eiffel) and is written in a notation called Lace (Language for the Assembly of Classes in Eiffel). As elsewhere in this book, the vertical bar character | is a placeholder for the path separator used on your operating system; replace it by / on Unix, \ on Windows, or the equivalent convention for specifying path names on VMS and other platforms.

```
system
    example
root
    CALCULATOR (my_cluster1): "make"
default
    assertion (ensure);
    precompiled
        ("$EIFFEL3|precomp|spec|$PLATFORM|base")
cluster
    my_cluster1: "mydir|project1|subdir";
    her_cluster2: "herdir|project2|subdir1|subdir2";
end
```

You will need an Ace to compile any system, but except for advanced uses you do not have to learn the details of Lace, which are simple anyway: the installation contains plenty of example Aces that you will be able to copy and adapt for your own use. Look in particular in the ***examples*** subdirectory of the delivery directory.

The subset of Lace illustrated by the above Ace suffices for basic uses of Eiffel. The ***system*** clause gives a name to the system, here *example*; the environment will use this name for the executable file generated as the result of compilation. The ***root*** clause gives the name of the root class, here *CALCULATOR*, and its creation procedure, here *make*; the name of the cluster to which *CALCULATOR* belongs, appearing in parentheses, is optional if no other cluster contains a class of that name. This Ace, by the way, is adapted from a small digital calculator system, one of the examples included in the delivery — hence the name of the root class.

This example fully exploits polymorphism: depending on the outcome of "Some test", *v* will be treated as a car or a truck, and the appropriate registration algorithm will be applied. Also, "Some test" may depend on some event whose outcome is impossible to predict until run-time, for example the user clicking with the mouse to select one among several vehicle icons displayed on the screen.

Deferred classes are particularly useful at the **design** stage. The first version of a module may be a deferred class, which will later be refined into one or more effective (non-deferred) classes. Particularly important for this application is the possibility of associating a precondition and a postcondition with a routine even though it is a deferred routine (as with *register* above), and an invariant with a class even though it is a deferred class. This enables the designer to attach precise semantics to a module at the design stage, long before making any implementation choices.

These possibilities make Eiffel an attractive alternative to PDLs (Program Design Languages). The combination of deferred classes to capture partially understood concepts, assertions to express what is known about their semantics, and the language's other structuring facilities (information hiding, inheritance, genericity) to obtain clear, convincing architectures, yields a higher-level design method. A further benefit, of course, is that the notation is also a programming language, making the development cycle smoother by reducing the gap between design and implementation.

The role of Eiffel in this cycle is not limited to design and implementation but extends to the earliest stage of development, **analysis**. Deferred classes written at that stage describe not software objects, but objects from the external reality's model — documents, airplanes, investments. Here again the presence of assertions to express constraints, and the language's other structuring facilities, provide an attractive alternative to older methods and notations.

D.12 PUTTING A SYSTEM TOGETHER

This discussion has focused so far on individual classes. This is consistent with the Eiffel method, which emphasizes reusability and, as a consequence, the construction of autonomous modules.

To execute software, you will need to group classes into executable compounds. Such a compound is called a **system** — the Eiffel concept closest to the traditional notion of program — and is defined by the following elements:

1 • A set of classes, called a **universe**.

2 • The designation of one of these classes as the system's **root class**.

3 • The designation of one of the creation procedures of the root class as the **root creation procedure**.

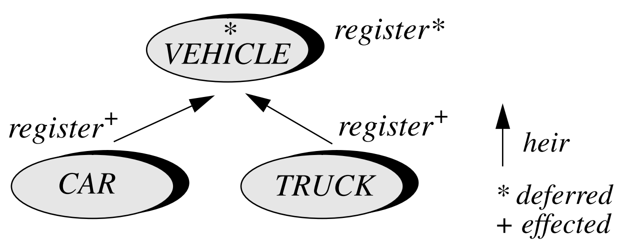
To execute such a system is to create one direct instance of the root class (the **root object** for the current execution), and to apply to it the root creation procedure — which will usually create other objects, call other routines and so on.

For simplicity and conformance to the method (which uses the class as the fundamental form of module) ISE Eiffel assumes that you store the text of each class in a separate file, called its class file. The default convention is to use file names of the form *name***.***e*. For clarity, *name* should normally be the class name, but because DOS and Windows limit file

```
deferred class VEHICLE feature
    dues_paid (year: INTEGER): BOOLEAN is
        do... end;
    valid_plate (year: INTEGER): BOOLEAN is
        do... end;
    register (year: INTEGER) is
            -- Register vehicle for year.
        require
            dues_paid (year)
        deferred
        ensure
            valid_plate (year)
        end;
    ... Other features, deferred or effective...
end -- class VEHICLE
```

This example assumes that no single registration algorithm applies to all kinds of vehicle; passenger cars, motorcycles, trucks etc. are all registered differently. But the same precondition and postcondition apply in all cases. The solution is to treat *register* as a deferred routine, making *VEHICLE* a deferred class. Descendants of class *VEHICLE*, such as *CAR* or *TRUCK*, **effect** this routine, that is to say, give effective versions. An effecting is similar to a redefinition; only here there is no effective definition in the original class, just a specification in the form of a deferred routine. The term **redeclaration** covers both redefinition and effecting.

Vehicle inheritance hierarchy



Deferred classes describe a group of implementations of an abstract data type rather than just a single implementation. A deferred class may not be instantiated: !! *v* is invalid if *v* is an entity declared of type *VEHICLE*. But such an entity may be assigned a reference to an instance of a non-deferred descendant of *VEHICLE*. For example, assuming *CAR* and *TRUCK* provide effective definitions for all deferred routines of *VEHICLE*, the following will be valid:

```
v: VEHICLE; c: CAR; t: TRUCK;
...
!! c ...; !! t ...;...
if "Some test" then v := c else v := t end;
v.register (1996)
```

The conformance requirement used here is the inheritance-based type compatibility rule; in simple cases, *V* conforms to *T* if and only if *V* is a descendant of *T*.

Structures such as *pl* may contain objects of different types, hence the name "polymorphic data structure". Such polymorphism is, again, made safe by the type rules: by choosing an actual generic parameter (*POLYGON* in the example) based higher or lower in the inheritance graph, you extend or restrict the permissible types of objects in *pl*. A fully general list would be declared as

LIST [*ANY*]

where *ANY*, a Kernel Library class, is automatically an ancestor of any class that you may write.

The other mechanism for combining genericity and inheritance is **constrained genericity**. By indicating a class name after a formal generic parameter, as in

VECTOR [*T* –> *ADDABLE*]

you express that only descendants of that class (here *ADDABLE*) may be used as the corresponding actual generic parameters. This makes it possible to use the corresponding operations. Here, for example, class *VECTOR* may define a routine ***infix*** "+" for adding vectors, based on the corresponding routine from *ADDABLE* for adding vector elements. Then by making *VECTOR* itself inherit from *ADDABLE*, you ensure that it satisfies its own generic constraint and enable the definition of types such as *VECTOR* [*VECTOR* [*T*]].

As you have perhaps guessed, unconstrained genericity, as in *LIST* [*G*], may be viewed as an abbreviation for genericity constrained by *ANY*, as in

LIST [*G* –> *ANY*].

D.11 DEFERRED CLASSES

The inheritance mechanism includes one more major component: deferred routines and classes.

Declaring a routine *r* as deferred in a class *C* expresses that there is no default implementation of *r* in *C*; such implementations will appear in eventual descendants of *C*. A class which has one or more deferred routines is itself said to be deferred. A non-deferred routine or class is said to be **effective**.

For example, a system used by a Department of Motor Vehicles to register vehicles could include a class of the form

can always define a new descendant of that class (with new features, redefined ones, or both) without any change to the original. This facility is of great importance in software development, an activity which — whether by design or by circumstance — is invariably incremental.

The power of polymorphism and dynamic binding demands adequate controls. First, feature redefinition, as seen above, is explicit. Second, because the language is typed, a compiler can check statically whether a feature application *a*•*f* is correct. In contrast, dynamically typed object-oriented languages defer checks until run-time and hope for the best: if an object "sends a message" to another (that is to say, calls one of its routines) one just expects that the corresponding class, or one of its ancestors, will happen to include an appropriate routine; if not, a run-time error will occur. Such errors will not happen during the execution of a type-checked Eiffel system.

In other words, the language reconciles dynamic *binding* with static *typing*. Dynamic binding guarantees that whenever more than one version of a routine is applicable the *right* version (the one most directly adapted to the target object) will be selected. Static typing means that the compiler makes sure there is *at least one* such version.

This policy also yields an important performance benefit: in contrast with the costly run-time searches that may be needed with dynamic typing (since a requested routine may not be defined in the class of the target object but inherited from a possibly remote ancestor), the EiffelBench implementation always finds the appropriate routine in constant-bounded time.

Assertions provide a further mechanism for controlling the power of redefinition. In the absence of specific precautions, redefinition may be dangerous: how can a client be sure that evaluation of *p*•*perimeter* will not in some cases return, say, the area? Preconditions and postconditions provide the answer by limiting the amount of freedom granted to eventual redefiners. The rule is that any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original. In other words, it must stay within the semantic boundaries set by the original assertions.

The rules on redefinition and assertions are part of the Design by Contract theory, where redefinition and dynamic binding introduce subcontracting. *POLYGON*, for example, subcontracts the implementation of perimeter to *RECTANGLE* when applied to any entity that is attached at run-time to a rectangle object. An honest subcontractor is bound to honor the contract accepted by the prime contractor. This means that it may not impose stronger requirements on the clients, but may accept more general requests, so that the precondition may be weaker; and that it must achieve at least as much as promised by the prime contractor, but may achieve more, so that the postcondition may be stronger.

For example of assertion adaptation in descendants, through the ***require else*** *and* ***ensure then*** *clauses, see "Reconstructing the whole feature", 9.4.2, page 171.*

D.10 COMBINING GENERICITY AND INHERITANCE

Genericity and inheritance, the two fundamental mechanisms for generalizing classes, may be combined in two fruitful ways.

The first technique yields **polymorphic data structures**. Assume that in the generic class *LIST* [*G*] the insertion procedure put has a formal argument of type G, representing the element to be inserted. Then with a declaration such as

pl: *LIST* [*POLYGON*]

the type rules imply that in a call *pl*•*put* ("...") the argument may be not just of type *POLYGON*, but also of type *RECTANGLE* (an heir of *POLYGON*) or any other type conforming to *POLYGON* through inheritance.

original by a descendant; for a routine it may also affect the implementation, replacing the original's routine body by a new one.

Assume for example a class *POLYGON*, describing polygons, whose features include an array of points representing the vertices and a function *perimeter* which computes a polygon's perimeter by summing the successive distances between adjacent vertices. An heir of *POLYGON* may begin as:

```
class RECTANGLE inherit
    POLYGON redefine perimeter end
feature -- Specific features of rectangles, such as:
    side1: REAL; side2: REAL;
    perimeter: REAL is
            -- Rectangle-specific version
        do Result := 2 * (side1 + side2) end;
    ... Other RECTANGLE features ...
```

Here it is appropriate to redefine *perimeter* for rectangles as there is a simpler and more efficient algorithm. Note the explicit ***redefine*** subclause (which would come after the ***rename*** if present).

Other descendants of *POLYGON* may also have their own redefinitions of *perimeter*. The version to use in any call is determined by the run-time form of the target. Consider the following class fragment:

```
p: POLYGON; r: RECTANGLE;
... !! p; !! r; ...
if c then
    p := r
end;
print (p.perimeter)
```

The polymorphic assignment *p* := *r* is valid because of the above rule. If condition c is false, *p* will be attached to an object of type *POLYGON* for the computation of *p*.*perimeter*, which will thus use the polygon algorithm. In the opposite case, however, *p* will be attached to a rectangle; then the computation will use the version redefined for *RECTANGLE*. This is known as **dynamic binding**.

Dynamic binding provides a high degree of flexibility. The advantage for clients is the ability to request an operation (such as perimeter computation) without explicitly selecting one of its variants; the choice only occurs at run-time. This is essential in large systems, where many variants may be available; each component must be protected against changes in other components.

This technique is particularly attractive when compared to its closest equivalent in traditional approaches. In Pascal or Ada, you would need records with variant components, and ***case*** instructions to discriminate between variants. This means that every client must know about every possible case, and that any extension may invalidate a large body of existing software.

These techniques support a development mode in which every module is open and incremental. When you want to reuse an existing class but need to adapt it to a new context, you

hierarchical features from *TREE* (a class in EiffelBase describing trees), and all graphical features from a class *RECTANGLE*.

Inheritance yields remarkable economies of effort — whether for analysis, design, implementation or evolution — and has a profound effect on the entire software development process.

The very power of inheritance demands adequate means to keep it under control. Multiple inheritance, in particular, raises the question of name conflicts between features inherited from different parents; this case will inevitably arise in practice, especially for classes contributed by independent developers. Such a name conflict may be removed through **renaming**, as in

```
class C inherit
    A rename x as x1, y as y1 end;
    B rename x as x2, y as y2 end
feature...
```

Here, if both *A* and *B* have features named *x* and *y*, class *C* would be invalid without the renaming.

Renaming also serves to provide more appropriate feature names in descendants. For example, class *WINDOW* may inherit a routine *insert_subtree* from *TREE*. For clients of *WINDOW*, however, such a routine name is no longer appropriate. An application using this class for window manipulation needs coherent window terminology, and should not be concerned with the inheritance structure that led to the implementation of the class. So you may wish to rename *insert_subtree* as *add_subwindow* in the inheritance clause of *WINDOW*.

As a further facility to protect against misusing the multiple inheritance mechanism, the invariants of all parent classes automatically apply to a newly defined class. So classes may not be combined if their invariants are incompatible.

D.9 POLYMORPHISM AND DYNAMIC BINDING

Inheritance is not just a module combination and enrichment mechanism. It also enables the definition of flexible entities that may become attached to objects of various forms at run time, a property known as polymorphism.

This remarkable facility must be reconciled with static typing. The language convention is simple: an assignment of the form *a* := *b* is permitted not only if *a* and *b* are of the same type, but more generally if *a* and *b* are of reference types *A* and *B*, based on classes *A* and *B* such that *B* is a descendant of *A*.

This corresponds to the intuitive idea that a value of a more specialized type may be assigned to an entity of a less specialized type — but not the reverse. (As an analogy, consider that if you request vegetables, getting green vegetables is fine, but if you ask for green vegetables, receiving a dish labeled just "vegetables" is not acceptable, as it could include, say, carrots.)

What makes this possibility particularly powerful is the complementary facility: **feature redefinition**. A class may redefine some or all of the features which it inherits from its parents. For an attribute or function, the redefinition may affect the type, replacing the

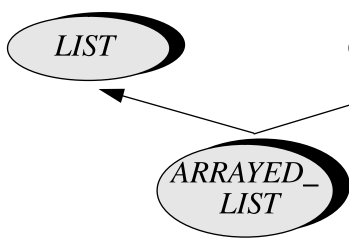
if the lists have a fixed number of elements uses an array. We may define the corresponding class by combination of *LIST* and *ARRAY*, as follows:

```
class ARRAYED_LIST [G] inherit
    LIST [G];
    ARRAY [G]
        export ... See below ...end
feature
    ... Specific features of fixed-size lists ...
end -- class ARRAYED_LIST
```

The ***inherit***... clause lists all the "parents" of the new class, which is said to be their "heir". (The "ancestors" of a class include the class itself, its parents, grandparents etc.; the reverse term is "descendant".) Declaring *ARRAYED_LIST* as shown ensures that all the features and properties of lists and arrays are applicable to fixed lists as well. Since the class has more than one parent, this is a case of *multiple* inheritance.

Standard graphical conventions serve to illustrate such inheritance structures:

An inheritance structure



An heir class such as *ARRAYED_LIST* needs the ability to define its own export policy. By default, inherited features keep their export status (publicly available, secret, available to selected classes only); but this may be changed in the heir. Here, for example, *ARRAYED_LIST* will export only the exported features of *LIST*, making those of *ARRAY* unavailable directly to *ARRAYED_LIST*'s clients. The syntax to achieve this is straightforward:

```
class ARRAYED_LIST [G] inherit
        LIST [G];
        ARRAY [G]
            export {NONE} all end
... The rest as above ...
```

Another example of multiple inheritance comes from a windowing system based on a class *WINDOW*, close to actual classes in EiffelVision. Windows have **graphical** features: a height, a width, a position, routines to scale windows, move them, and other graphical operations. The system permits windows to be nested, so that a window also has **hierarchical** features: access to subwindows and the parent window, adding a subwindow, deleting a subwindow, attaching to another parent and so on. Rather than writing a complex class that would contain specific implementations for all of these features, it is preferable to inherit all

This disciplined exception mechanism is essential for software developers, who need protection against unexpected events, but cannot be expected to sacrifice safety and simplicity to pay for this protection.

D.7 GENERICITY

Building software components (classes) as implementations of abstract data types yields systems with a solid architecture but does not in itself suffice to ensure reusability and extendibility. Two key techniques address the problem: genericity (unconstrained or constrained) and inheritance. Let us look first at the unconstrained form.

To make a class generic is to give it **formal generic parameters** representing arbitrary types, as in these examples from the Kernel and Data Structure Libraries of EiffelBase:

ARRAY [*G*]

LIST [*G*]

LINKED_LIST [*G*]

These classes describe data structures — arrays, lists without commitment to a specific representation, lists in linked representation — containing objects of a certain type. The formal generic parameter *G* represents this type.

Each of these classes describes a type template. To derive a directly usable type, you must provide a type corresponding to *G*, called an **actual generic parameter**; this may be either a basic expanded type (such as *INTEGER*) or a reference type. Here are some possible generic derivations:

il: *LIST* [*INTEGER*];

aa: *ARRAY* [*ACCOUNT*];

aal: *LIST* [*ARRAY* [*ACCOUNT*]]

As the last example indicates, an actual generic parameter may itself be generically derived.

Without genericity, it would be impossible to obtain static type checking in a realistic object-oriented language.

A variant of this mechanism, *constrained* genericity, will enable a class to place specific requirements on possible actual generic parameters. Constrained genericity will be introduced below, after inheritance.

D.8 INHERITANCE

Inheritance, the other fundamental generalization mechanism, makes it possible to define a new class by combination and specialization of existing classes rather than from scratch.

The following simple example, from the Data Structure Library in EiffelBase, is typical. *LIST*, as indicated, describes lists in any meaningful representation. One such representation

A routine may, however, handle an exception through a ***rescue*** clause. This optional clause attempts to "patch things up" by bringing the current object to a stable state (one satisfying the class invariant). Then it can terminate in either of two ways:

• The ***rescue*** clause may execute a ***retry*** instruction, which causes the routine to restart its execution from the beginning, attempting again to fulfil its contract, usually through another strategy. This assumes that the instructions of the ***rescue*** clause, before the ***retry***, have attempted to correct the cause of the exception.

• If the ***rescue*** clause does not end with ***retry***, then the routine fails: it returns to its caller, immediately signaling an exception. (The caller's ***rescue*** clause will be executed according to the same rules.)

The principle is that **a routine must either succeed or fail**: either it fulfils its contract, or it does not; in the latter case it must notify its caller by triggering an exception.

Usually, only a few routines of a system will include explicit ***rescue*** clauses. An exception occurring during the execution of a routine with no ***rescue*** clause will trigger a predefined rescue procedure, which does nothing, and so will cause the routine to fail immediately, propagating the exception to the routine's caller.

An example using the exception mechanism is a routine *attempt_transmission* which tries to transmit a message over a phone line. The actual transmission is performed by an external, low-level routine *transmit*; once started, however, *transmit* may abruptly fail, triggering an exception, if the line is disconnected. Routine *attempt_transmission* tries the transmission at most 50 times; before returning to its caller, it sets a boolean attribute *successful* to ***true*** or ***false*** depending on the outcome. Here is the text of the routine:

```
attempt_transmission (message: STRING) is
        -- Try to transmit message, at most 50 times.
        -- Set successful accordingly.
    local
        failures: INTEGER
    do
        if failures < 50 then
            transmit (message); successful := true
        else
            successful := false
        end
    rescue
        failures := failures + 1; retry
    end
```

Initialization rules ensure that *failures*, a local entity, is set to zero on entry.

This example illustrates the simplicity of the mechanism: the ***rescue*** clause never attempts to achieve the routine's original intent; this is the sole responsibility of the body (the ***do*** clause). The only role of the ***rescue*** clause is to clean up the objects involved, and then either to fail or to retry.

```
    withdraw (sum: INTEGER) is
            -- Withdraw sum from the account.
        require
            sum >= 0;
            sum <= balance – minimum_balance
        ensure
            balance = old balance – sum
    may_withdraw ...
end -- class ACCOUNT
```

This is not actual Eiffel, only documentation of Eiffel classes, hence the use of slightly different syntax to avoid any confusion (***class interface*** rather than ***class***). In accordance with observations made above, the output for *balance* would be the same if this feature were a function rather than an attribute.

In the ISE Eiffel environment, the short form is produced by automatic tools described in this book; it serves as the primary form of class documentation. A variant, the flat-short form, takes inheritance into account by including inherited features along with those introduced in the class itself.

See "Short, flat and flat-short forms", 8.3.3, page 149 on how to get the short form in EiffelBench by clicking on the Short format button. For the non-graphical equivalent see "The Class menu", 11.4.3, page 223.

It is also possible to evaluate assertions at run time, so as to uncover potential errors ("bugs"). The implementation provides several levels of assertion monitoring: preconditions only, postconditions etc. When monitoring is on, an assertion which evaluates to true has no further effect on the execution. An assertion which evaluates to false will trigger an exception, as described next; unless the software developer has written an appropriate exception handler, the exception will cause an error message and termination with a precise message and a call trace.

An example exception trace appears on page 201.

This ability to check assertions provides a powerful testing and debugging mechanism, in particular because the classes of the EiffelBase Libraries, widely used in Eiffel software development, are protected by carefully written assertions.

Run-time checking, however, is only one application of assertions, whose role as design and documentation aids, as part of the theory of Design by Contract, exerts a pervasive influence on the Eiffel style of software development.

D.6 EXCEPTIONS

Whenever there is a contract, the risk exists that someone will break it. This is where exceptions come in.

Exceptions — contract violations — may arise from several causes. One is assertion violations, if assertions are monitored. Another is the occurrence of a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow or lack of memory to create a new object.

Unless a routine has made specific provision to handle exceptions, it will **fail** if an exception arises during its execution. Failure of a routine is a third cause of exception: a routine that fails triggers an exception in its caller.

To make such a call valid, it would suffice to move the declaration of *make* to the first ***feature*** clause of class *ACCOUNT*, which carries no export restriction. Such a call does not create any new object, but simply resets the balance of a previously created account.

D.5.3 Design by contract

Syntactically, assertions are boolean expressions, with a few extensions such as the ***old*** notation. The semicolon (see the precondition to *withdraw*) is equivalent to an "and", but permits individual identification of the components.

Assertions play a central part in the Eiffel method for building reliable object-oriented software. They serve to make explicit the assumptions on which programmers rely when they write software elements that they believe are correct. Writing assertions, in particular preconditions and postconditions, amounts to spelling out the terms of the **contract** which governs the relationship between a routine and its callers. The precondition binds the callers; the postcondition binds the routine.

The underlying theory of *Design by Contract*, the centerpiece of the Eiffel method, views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and benefits made explicit by the assertions.

D.5.4 The short form

Assertions are also an indispensable tool for the documentation of reusable software components: one cannot expect large-scale reuse without a precise documentation of what every component expects (precondition), what it guarantees in return (postcondition) and what general conditions it maintains (invariant).

Documentation tools in EiffelBench use assertions to produce information for client programmers, describing classes in terms of observable behavior, not implementation. In particular the **short form** of a class, which serves as its interface documentation, is obtained from the full text by removing all non-exported features and all implementation information such as ***do*** clauses of routines, but keeping interface information and in particular assertions. Here is the short form of the above class:

```
class interface ACCOUNT creation
    make
feature
        balance: INTEGER;
        ...
    deposit (sum: INTEGER) is
            -- Deposit sum into the account.
        require
            sum >= 0
        ensure
            balance = old balance + sum
```

```
    withdraw (sum: INTEGER) is
            -- Withdraw sum from the account.
        require
            sum >= 0;
            sum <= balance – minimum_balance
        do
            add (–sum)
        ensure
            balance = old balance – sum
        end;
    may_withdraw ...  -- As before
feature {NONE}
    add ... -- As before;
    make (initial: INTEGER) is
            -- Initialize account with balance initial.
        require
            initial >= minimum_balance
        do
            balance := initial
        end
invariant
    balance >= minimum_balance
end -- class ACCOUNT
```

The notation ***old*** *attribute_name* may only be used in a routine postcondition. It denotes the value the attribute had on routine entry.

D.5.2 Creation procedures

In its last version above, the class now includes a creation procedure, *make*. With the first version of *ACCOUNT*, clients used creation instructions such as !! *acc1* to create accounts; but then the default initialization, setting balance to zero, violated the invariant. By having one or more creation procedures, listed in the ***creation*** clause at the beginning of the class text, a class offers a way to override the default initializations. The effect of

!! *acc1*•*make* (*5_500*)

is to allocate the object (as with the default creation) and to call procedure *make* on this object, with the argument given. This call is correct since it satisfies the precondition; it will ensure the invariant. (The underscore _ in the integer constant *5_500* has no semantic effect. The general rule is that you can group digits by sets of three from the right to improve the readability of integer constants.)

A procedure listed in the ***creation*** clause, such as *make*, otherwise enjoys the same properties as other routines, especially for calls. Here the procedure *make* is secret since it appears in a clause starting with ***feature*** {*NONE*}; so it would be invalid for a client to include a call such as

acc•*make* (*8_000*)

As a result of these conventions, the type system is uniform and consistent: all types, including the basic types, are defined from classes, either as reference types or as expanded types.

In the case of basic types, for obvious reasons of efficiency, the ISE Eiffel compilation mechanism implements the standard arithmetic and boolean operations directly through the corresponding machine operations, not through routine calls. But this is only a compiler optimization, which does not hamper the conceptual homogeneity of the type edifice.

D.5 ASSERTIONS

If classes are to deserve their definition as abstract data type implementations, they must be known not just by the available operations, but also by the formal properties of these operations, which did not appear in the above example.

D.5.1 The role of assertions

Eiffel encourages software developers to express formal properties of classes by writing **assertions**, which may in particular appear in the following roles:

- Routine **preconditions** express the requirements that clients must satisfy whenever they call a routine. For example the designer of *ACCOUNT* may wish to permit a withdrawal operation only if it keeps the account's balance at or above the minimum. Preconditions are introduced by the keyword ***require***.
- Routine **postconditions**, introduced by the keyword ***ensure***, express conditions that the routine (the supplier) guarantees on return, if the precondition was satisfied on entry.
- A class **invariant** must be satisfied by every instance of the class whenever the instance is externally accessible: after creation, and after any call to an exported routine of the class. The invariant appears in a clause introduced by the keyword ***invariant***, and represents a general consistency constraint imposed on all routines of the class.

With appropriate assertions, the *ACCOUNT* class becomes:

```
class ACCOUNT creation
    make
feature
    ... Attributes as before: balance, minimum_balance, owner, open ...
    deposit (sum: INTEGER) is
            -- Deposit sum into the account.
        require
            sum >= 0
        do
            add (sum)
        ensure
            balance = old balance + sum
        end;
```

entity on the left of the dot, as *acc* in *acc*•*open* (*"Jill"*). Within the class, however, the "current" instance to which operations apply usually remains implicit, so that unqualified feature names, such as *owner* in procedure *open* or *add* in *deposit*, mean "the *owner* attribute or *add* routine relative to the current instance".

If you need to denote the current object explicitly, you may use the special entity *Current*. For example the unqualified occurrences of *add* appearing in the above class are equivalent to *Current*•*add*.

In some cases, infix or prefix notation will be more convenient than dot notation. For example, if a class *VECTOR* offers an addition routine, most people will feel more comfortable with calls of the form *v* + *w* than with the dot-notation call *v*•*plus* (*w*). To make this possible it suffices to give the routine a name of the form ***infix*** "+" rather than *plus*; internally, however, the operation is still a normal routine call. Prefix operators are similarly available.

The above simple example has shown the basic structuring mechanism of the language: the class. A class describes a data structure, accessible to clients through an official interface comprising some of the class features. Features are implemented as attributes or routines; the implementation of exported features may rely on other, secret ones.

D.4 TYPES

Eiffel is strongly typed for readability and reliability. Every entity is declared of a certain type, which may be either a reference type or an expanded type.

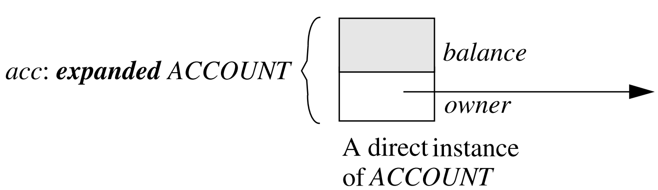
Any type *T* is based on a class, which defines the operations that will be applicable to instances of *T*. The difference between the two categories of type affects the semantics of an entity *x* declared of type *T*: for a reference type, the most common case, possible values for *x* are references to objects; for an expanded type, the values are objects. In both cases, the type rules guarantee that the objects will be instances of *T*.

A non-expanded class such as *ACCOUNT* yields a reference type. As a result, an entity of type *ACCOUNT*, such as *acc* in the earlier client example, denotes possible run-time references to objects of type *ACCOUNT*.

See the declaration of 'acc' and the accompanying picture on page 267.

In contrast, the value of an entity *acc* declared of type ***expanded*** *ACCOUNT* is an object such as the one shown on the figure below, with no reference. The only difference with the earlier figure is that the value of *acc* is now an *ACCOUNT* object, not a reference to such an object. No creation instruction is needed in this case. (The figure does not show the *PERSON* object to which the *owner* field of the *ACCOUNT* object — itself a reference — is attached.)

An entity of expanded type



An important group of expanded types, based on library classes, includes the basic types *INTEGER, REAL, DOUBLE, CHARACTER* and *BOOLEAN*. Clearly, the value of an entity declared of type *INTEGER* should be an integer, not a reference to an object containing an integer value. Operations on these types are defined by prefix and infix operators such as "+" and "<".

```
class ACCOUNT feature
    balance: INTEGER;
    owner: PERSON;
    minimum_balance: INTEGER is 1000;
    open (who: PERSON) is
            -- Assign the account to owner who.
        do
            owner := who
        end;
    deposit (sum: INTEGER) is
            --  Deposit sum into the account.
        do
            add (sum)
        end;
    withdraw (sum: INTEGER) is
            -- Withdraw sum from the account.
        do
            add (–sum)
        end;
    may_withdraw (sum: INTEGER): BOOLEAN is
            -- Is there enough money to withdraw sum?
        do
            Result := (balance >= sum + minimum_balance)
        end
feature {NONE}
    add (sum: INTEGER) is
            -- Add sum to the balance.
        do
            balance := balance + sum
        end
end -- class ACCOUNT
```

The other public features, *open, deposit, withdraw* and *may_withdraw* are straightforward routines. The special entity *Result*, used in *may_withdraw*, denotes the function result; it is initialized on function entry to the default value of the function's result type.

The secret procedure *add* serves for the implementation of the public procedures *deposit* and *withdraw*; the designer of *ACCOUNT* judged it too general to be exported by itself. The clause ***is** 1000* introduces *minimum_balance* as a constant attribute, which will not occupy any space in instances of the class; in contrast, every instance has a field for every non-constant attribute such as *balance*.

In Eiffel's object-oriented programming style any operation is relative to a certain object. In a client invoking the operation, this object is specified by writing the corresponding

Once the client has attached *acc* to an object, it may call on this object the features defined in class *ACCOUNT*. Here is an extract with some feature calls using *acc* as their target:

```
acc.open ("Jill");
acc.deposit (5000);
if acc.may_withdraw (3000) then
    acc.withdraw (3000); print (acc.balance)
end
```

These feature calls use dot notation, of the form *target.feature_name*, possibly followed by a list of arguments in parentheses. Features are of two kinds:

- **Routines** (such as *open*, *deposit*, *may_withdraw*, *withdraw*) represent computations applicable to instances of the class.
- **Attributes** represent data items associated with these instances.

Routines are further divided into **procedures** (commands, which do not return a value) and **functions** (queries, returning a value). Here *may_withdraw* is a function returning a boolean; the other three routines called are procedures.

The above extract of class *X* does not show whether, in class *ACCOUNT*, *balance* is an attribute or a function without argument. This ambiguity is intentional. A client of *ACCOUNT*, such as *X*, does not need to know how a balance is obtained: the balance could be stored as attribute of every account object, or computed by a function from other attributes. Choosing between these techniques is the business of class *ACCOUNT*, not anybody else's. Because such implementation choices are often changed over the lifetime of a project, it is essential to protect clients against their effects.

So much for how client classes will typically use *ACCOUNT*. On the top of the following page is a first sketch of how class *ACCOUNT* itself might look. Line segments beginning with -- are comments. The class includes two ***feature*** clauses, introducing its features. The first begins with just the keyword ***feature***, without further qualification; this means that the features declared in this clause are available (or "exported") to all clients of the class. The second clause is introduced by ***feature*** {*NONE*} to indicate that the feature that follows, called *add*, is available to no client. What appears between the braces is a list of client classes to which the corresponding features are available; *NONE* is a special class of the Kernel Library, which has no instances, so that *add* is in effect a secret feature, available only locally to the other routines of class *ACCOUNT*. So in a client class such as *X*, the call *acc.add* (*–3000*) would be invalid.

Let us examine the features in sequence. The ***is*** ... ***do*** ...***end*** distinguishes routines from attributes. So here the class has implemented *balance* as an attribute, although, as noted, a function would also have been acceptable. Feature *owner* is also an attribute.

The language definition guarantees automatic initialization, so that the initial balance of an account object will be zero after a creation instruction. Each type has a default initial value, used for initialization: zero for *INTEGER* and *REAL*, false for *BOOLEAN*, null character for *CHARACTER*, and a void reference for reference types. The class designer may also provide clients with different initialization options, as will be shown below in a revised version of this example.

D.3 CLASSES

A class, it was said above, is an implementation of an abstract data type. This means that it describes a set of run-time objects, characterized by the **features** (operations) applicable to them, and by the formal properties of these features.

Such objects are called the **direct instances** of the class. Classes and objects should not be confused: "class" is a compile-time notion, whereas objects only exist at run time. This is similar to the difference that exists in classical programming between a program and one execution of that program, or between a type and a run-time value of that type.

"Object-Oriented" is a misnomer; "Class-Oriented Analysis, Design and Programming" would be a more accurate description of the method.

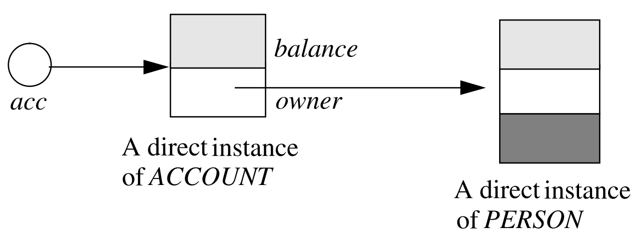
To see what a class looks like, let us look at a simple example, *ACCOUNT*, which describes bank accounts. But before exploring the class itself it is useful to study how it may be used by other classes, called its **clients**.

A class *X* may become a client of *ACCOUNT* by declaring one or more **entities** of type *ACCOUNT*. Such a declaration is of the form:

acc: *ACCOUNT*

The term "entity" generalizes the more common notion of "variable". An entity declared of a reference type, such as *acc*, may at any time during execution become "attached to" an object; the type rules imply that this object must be a direct instance of *ACCOUNT* — or, as seen below, of a "descendant" of that class.

An entity and the attached object



An entity is said to be void if it is not attached to any object. By default, entities are void at initialization. To obtain objects at run-time, a routine *r* appearing in the client class *X* may use a **creation instruction** of the form

!! *acc*

which creates a new direct instance of *ACCOUNT*, attaches *acc* to that instance, and initializes all its fields to default values. A variant of this notation, studied below, makes it possible to override the default initializations.

- The language enables implementors to produce high **efficiency** compilers, so that systems developed with Professional Eiffel may run under speed and space conditions similar to those of programs written in lower-level languages.
- Ensuring **openness**, so that Eiffel software may cooperate with programs written in other languages.
- Guaranteeing **portability** by a platform-independent language definition, so that the same semantics may be supported on many different platforms.

D.2 OBJECT-ORIENTED DESIGN

To achieve reusability, extendibility and reliability, the principles of object-oriented design provide the best known technical answer.

An in-depth discussion of these principles falls beyond the scope of this introduction but here is a short definition:

> Object-oriented design is the construction of software systems as structured collections of abstract data type implementations, or "classes".

The following points are worth noting in this definition:

- The emphasis is on structuring a system around the types of objects it manipulates (not the functions it performs on them) and on reusing whole data structures together with the associated operations (not isolated routines).
- Objects are described as instances of abstract data types — that is to say, data structures known from an official interface rather than through their representation.
- The basic modular unit, called the class, describes one implementation of an abstract data type (or, in the case of "deferred" classes, as studied below, a set of possible implementations of the same abstract data type).
- The word *collection* reflects how classes should be designed: as units which are interesting and useful on their own, independently of the systems to which they belong, and may be reused by many different systems. Software construction is viewed as the assembly of existing classes, not as a top-down process starting from scratch.
- Finally, the word *structured* reflects the existence of two important relations between classes: the client and inheritance relations.

Eiffel makes these techniques available to software developers in a simple and practical way.

As a language, Eiffel includes more than presented in this introduction, but not *much* more; it is a small language, not much bigger (by such a measure as the number of keywords) than Pascal. It was meant to be a member of the class of languages which programmers can master entirely — as opposed to languages of which most programmers know only a subset. Yet it is appropriate for the development of industrial software systems, as has by now been shown by many full-scale projects, some in the thousands of classes and hundreds of thousands of lines, in companies around the world.

D

Eiffel: An overview of the language and method

This text is extracted from "ISE Eiffel: The Environment", the manual for ISE's EiffelBench graphical development environment. It is made available for downloading by FTP for the benefit of people interested in learning the essentials of Eiffel. Released for personal use only; further publication requires the authorization of ISE. Low-cost personal licenses are available on several platforms. Please contact ISE at <info@eiffel.com>, telephone 805-685-1006, fax 805-685-6869, http://www.eiffel.com.

The Eiffel language and the underlying method lie at the basis of the ISE Eiffel environment. The language is described in detail in the book *Eiffel*: *The Language* and the method in *Object-Oriented Software Construction*. Here is an overview, adapted from the introduction to the first of these books.

D.1 DESIGN PRINCIPLES

The aim of Eiffel is to help specify, design, implement and modify quality software. This goal of quality in software is a combination of many factors; the language design concentrated on the three factors which, in the current state of the industry, are in direst need of improvements: reusability, extendibility and reliability. Also important were other factors such as efficiency, openness and portability.

Reusability is the ability to produce components that may be used in many different applications. Central to the Eiffel approach is the presence of predefined libraries such as EiffelBase, and the language's support for the production of new libraries.

Extendibility is the ability to produce easily modifiable software. "Soft" as software is supposed to be, it is notoriously hard to modify software systems, especially large ones.

Among quality factors, reusability and extendibility play a special role: satisfying them means having *less* software to write — and hence more time to devote to other important goals such as efficiency, ease of use or integrity.

The third fundamental factor is **reliability,** the ability to produce software that is correct and robust — that is to say, bug-free. Eiffel techniques such as static typing, assertions, disciplined exception handling and automatic garbage collection are essential here.

Three other factors are also part of Eiffel's principal goals: