Invitation to Eiffel

Interactive Software Engineering

Manual identification

Title: *Invitation to Eiffel*, ISE Technical Report TR-EI-67/IV.

Publication history

First published 1987. Some revisions (in particular Web versions) have used the title "*Eiffel in a Nutshell*"
This version: July 2001. Introduces coverage of agents; several other improvements. 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.

Invitation to Eiffel

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 detailed Eiffel Tutorial.

1 WHAT MUST I KNOW FIRST?

This Invitation assumes that you have some experience of software development, but that's all. Previous exposure to object technology is not required. If you've had it, it will help; but if it has all been to notations like UML or programming languages like C++ and Java, you should not let it guide your study of this Invitation. Although Eiffel shares a number of properties with these other approaches, it takes a fresh path to object technology, based on a small number of simple, far-reaching concepts.

Once you are familiar with the basic ideas you may want to try them with EiffelStudio, which provides a direct implementation of the Eiffel concepts, available in a completely portable way across Windows, Linux, many versions of Unix and VMS.

2 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 direct 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 serve 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:

- 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.

3 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 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.

4 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.

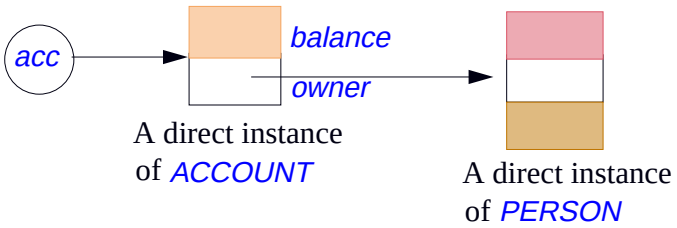
> "Object-Oriented" is a misnomer; "Class-Oriented Analysis, Design and Programming" would be a more accurate description of the method.

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

```
create 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.

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.

> A note on syntax: you may separate instructions by semicolons, and indeed you should when, as on the next-to-last line of the example, two or more instructions appear on a line. But the language's syntax has been designed so that the semicolon is almost always *optional*, regardless of the layout. Indeed the practice is to omit it between instructions or declarations on separate lines, as this results in lighter, clearer software texts.

In class *ACCOUNT*, is feature *balance* an attribute, or is it a function with no argument? The above extract of the client class *X* doesn't say, and this ambiguity is intentional. A client of *ACCOUNT* must not need to know how class *ACCOUNT* delivers an account's balance when requested: by looking up a field present in each account object, or by calling a function that computes the balance from other fields. 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. This is known as the **Uniform Access Principle**, stating that the choice between representing a property through memory (an attribute) or through an algorithm (function) shall not affect how clients use it.

So much for how client classes will typically use *ACCOUNT*. Below 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.

```
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
```

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: 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 seen below in a revised version of this example.

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. You may only use *Result* in functions.

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. A client invoking the operation specifies this object by writing the corresponding 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 class text above 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 objects 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.

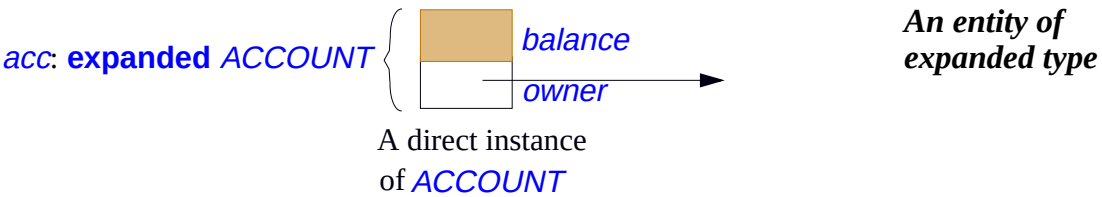
5 TYPES

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 (see the declaration of *acc* and the accompanying picture as given on page 6), denotes possible run-time references to objects of type *ACCOUNT*.

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 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 "<".

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.

6 DESIGN BY CONTRACT AND 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 yet appear in the preceding example.

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 class *ACCOUNT* becomes:

```
class ACCOUNT create
        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
```

```
        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** *expression* is only valid in a routine postcondition. It denotes the value the *expression* had on routine entry.

Creation procedures

In its last version above, the class now includes a creation procedure, *make*. With the first version , clients used creation instructions such as **create** *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 **create** clause at the beginning of the class text, a class offers a way to override the default initializations. The effect of

```
create 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.

Note that the same keyword, **create**, serves both to introduce creation instructions and the creation clause listing creation procedures at the beginning of the class.

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)
```

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.

Design by Contract

Syntactically, assertions are boolean expressions, with a few extensions such as the **old** notation. Also, you may split an assertion into two or more clauses, as here with the precondition of *withdraw*; this is as if you had separated the clauses with an **and**, but makes the assertion clearer, especially if it includes many conditions.

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 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.

The Contract 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 **Contract Form** of a class, also called its "*short form*", 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 Contract Form of the above class:

```
class interface ACCOUNT create
        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
                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 the Uniform Access Principle (page 7), the output for *balance* would be the same if this feature were a function rather than an attribute.

You will find in ISE's EiffelStudio automatic tools to produce the Contract Form of a class. You can also get the **Flat Contract** form, based on the same ideas but including inherited features along with those introduced in the class itself. EiffelStudio can produce these forms, and other documentation views of a class, in a variety of output formats including HTML, so that collaborative projects can automatically post the latest versions of their class interfaces on the Internet or an Intranet.

Contracts for testing and debugging

Under EiffelStudio you may also set up compilation options, for the whole system or specific classes only, to evaluate assertions at run time, to uncover potential errors ("bugs"). EiffelStudio 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 that evaluates to false will trigger an exception, as described next; unless you have written an appropriate exception handler, the exception will cause an error message and termination with a precise message and a call trace.

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 monitoring, 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.

7 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 an assertion violation, if you've selected run-time assertion monitoring. Another is a signal triggered by the hardware or operating system to indicate an abnormal condition such as arithmetic overflow, or an attempt to create a new object when there's not enough memory available.

Unless a routine has made specific provision to handle exceptions, it will **fail** if an exception arises during its execution. This in turn provides one more source of exceptions: a routine that fails triggers an exception in its caller.

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 triggering 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**: it either fulfills its contract, or not; in the latter case it must notify its caller by triggering an exception.

Usually, only a few routines of a system will explicitly include a **rescue** clause. A routine that doesn't have an explicit **rescue** is considered to have an implicit one, which calls a routine *default_rescue* that by default does nothing, so that an exception will cause the routine to fail immediately, propagating the exception to the caller.

An example using the exception mechanism is a routine *attempt_transmission* that 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.

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.

8 EVENT-DRIVEN PROGRAMMING AND AGENTS

The division of roles in object technology is clear: of the two principal constituents of a system, *object types* and *operations*, the first dominates. Classes, representing object types, determines the structure of the software; every routine, representing an operations, belongs to a class.

In some circumstances it is useful to define an *object* that denotes an *operation*. This is especially useful if you want to build an object structure that refers to operations, so that you can later traverse the structure and execute the operations encountered. A typical application is **event-driven programming** for Graphical User Interfaces (GUI), including Web programming. In GUI programming you will want to record properties of the form

> "When the user clicks this OK button, the system must update the file"

each involves a **control** (here the OK button), an **event** (mouse click) and an **operation** (update the file). This can be programmed by having an "event loop", triggered for each event, which performs massive decision-making (**if** "The latest event was 'left mouse click on button 23" **then** "Appropriate instructions" **else if** … and so on with many branches); but this leads to bulky software architectures where introducing any new control or event requires updating a central part of the code. It's preferable to let any element of the system that encounters a new control-event-operation association

> [*control*, *event*, *operation*]

store it as a triple of objects into an object structure, such as an array or a list. Triples in that structure may come from different parts of the system; there is no central know-it-all structure. The only central element is a simple mechanism which can explore the object structure to execute each *operation* associated with a certain *control* and a certain *event*. The mechanism is not just simple; it's also independent of your application, since it doesn't need to know about any particular control, event or operation (it will find them in the object structure). So it can be programmed once and for all, as part of a library such as ISE's EiffelVision 2 for platform-independent graphics.

To build an object structure, we need objects. A *control*, an *event* are indeed objects. But an *operation* is not: it's program code — a routine of a certain class.

Agents address this issue. An agent is an *object* that represents a *routine*, which can then be kept in an object structure. The simplest form of agent is written **agent** *r*, where *r* is a routine. This denotes an object. If *your_agent* is such an agent object, the call

```
your_agent.call ([a, b])
```

where *a* and *b* are valid arguments for *r*, will have the same effect as a direct call to *r* with arguments *a* and *b*. Of course, if you know that you want to call *r* with those arguments, you don't need any agents; just use the direct call *r* (*a*, *b*). The benefit of using an agent is that you can store it into an object structure to be called **later**, for example when an event-driven mechanism finds the agent in the object structure, associated with a certain control and a certain event. For this reason agents are also called **delayed calls**.

The notation [*a*, *b*] denotes a sequence of elements, or **tuple**. The reason *call* needs a tuple as argument, whereas the direct call *r* (*a*, *b*) doesn't, is that *call* is a general routine (from the EiffelBase class *ROUTINE*, representing agents) applicable to any agent, whereas the direct call refers explicitly to *r* and hence requires arguments *a* and *b* of specific types. The agent mechanism, however, is statically typed like the rest of the language; when you call *call*, the type checking mechanism ensures that the tuple you pass as argument contains elements *a* and *b* of the appropriate types.

A typical use of agents with EiffelVision 2 is

```
ok_button.select_actions.extend (agent your_routine)
```

which says: "add *your_routine* to the list of operations to be performed whenever a *select* event (left click) happens on *ok_button*". *ok_button.select_actions* is the list of agents associated with the button and the event; in list classes, procedure *extend* adds an item at the end of a list. Here, the object to be added is the agent.

This enables the EiffelVision event-handling mechanism to find the appropriate agent when it processes an event, and call *call* on that agent to trigger the appropriate routine. EiffelVision doesn't know that it's *your_routine*; in fact, it doesn't know anything about your application. It simply finds an agent in the list, and calls *call* on it. For your part, as the author of a graphical application, you don't need to know how EiffelVision handles events; you simply associate the desired agents with the desired controls and events, and let EiffelVision 2 do the rest.

Agents extend to many areas beyond GUIs. In **numerical computation**, you may use an agent to pass to an "integrator" object a numerical function to be integrated over a certain interval. In yet another area, you can use agents (as in the iteration library of EiffelBase) to program **iterators**: mechanisms that repetitively apply an arbitrary operation — represented by an agent — to every element of a list, tree or other object structure. More generally, agent embody properties of the associated routines, opening the way to mechanism for **reflection**, also called "introspection": the ability, during software execution, to discover properties of the software itself.

9 GENERICITY

Building software components (classes) as implementations of abstract data types yields systems with a solid architecture but does not in itself 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 as unknown types, as in these examples from ISE's EiffelBase, an open-source library covering basic data structures and algorithms:

```
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* denotes this type.

A class such as these doesn't quite yet describe a type, but a type template, since *G* itself denotes an unknown type. To derive a directly usable list or array type, you must provide a type corresponding to *G*, called an **actual generic parameter**; this may be either an expanded type, including basic types 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.

It would not be possible, without genericity, to have 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 described after inheritance.

10 INHERITANCE

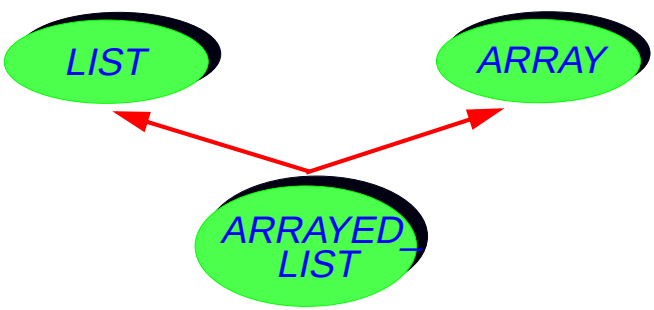
Inheritance, the other fundamental generalization mechanism, makes it possible to define a new class by combination and specialization of existing classes rather than from scratch.

The following simple example, from the Data Structure Library in EiffelBase, is typical. *LIST*, as noted, describes lists in any representation. One such representation 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 arrayed lists as well. Since the class has more than one parent, this is a case of *multiple* inheritance.

Standard graphical conventions — drawn from the Business Object Notation or BON, a graphical object-oriented notation based on concepts close to those of Eiffel, and directly supported by EiffelStudio — 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 2. 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 hierarchical features from *TREE* (a class in EiffelBase describing trees), and all graphical features from a class *RECTANGLE*.

Inheritance complements the "client" relation by providing another form of reuse that yields remarkable economies of effort — for analysis, design, implementation, 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. You may remove such a name conflict 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 that uses this class needs coherent window terminology, and should have to concern itself with the inheritance structure that led to the class. So you may wish to rename *insert_subtree* as *add_subwindow* in the inheritance clause of *WINDOW*.

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

11 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 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
        ... create p; create 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; dynamic binding protects each component against changes in other components.

This technique is particularly attractive when compared to its closest equivalent in traditional approaches, where you would need records with variant components, or union types (C), together with **case** (switch) 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.

The combination of inheritance, feature redefinition, polymorphism and dynamic binding supports 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 can 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 that — by design or circumstance — is invariably incremental.

The power of these techniques 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. This means that 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: weaker precondition; and that it must achieve at least as much as promised by the prime contractor, but may achieve more: stronger postcondition.

12 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 ("p")* the permitted types for the argument *p* include not just *POLYGON*, but also *RECTANGLE* (an heir of *POLYGON*) or any other type conforming to *POLYGON* through inheritance.

The basic conformance requirement used here is the inheritance-based type compatibility rule: *V* conforms to *T* 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 –> NUMERIC]
```

you express that only descendants of that class (here *NUMERIC*) 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 *NUMERIC* for adding vector elements. Then by making *VECTOR* itself inherit from *NUMERIC*, 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].
```

Something else you may have guessed: if *ANY*, introduced in this session, is the top of the inheritance structure — providing all classes with universal features such as *equal* to compare arbitrary objects and *clone* to duplicate objects — then *NONE*, seen earlier in the notation **feature** {*NONE*}, is its bottom. *NONE* indeed conceptually inherits from all other classes. *NONE* is, among other things, the type of *Void*, the void reference.

13 DEFERRED CLASSES AND SEAMLESS DEVELOPMENT

The inheritance mechanism includes one more major notion: deferred features and classes.

Declaring a feature *f* as deferred in a class *C* expresses that there is no default implementation of *f* in *C*; such implementations will appear in eventual descendants of *C*. A class that has one or more deferred routines is itself said to be deferred. A non-deferred routine or class — like all those seen until now — is said to be **effective**.

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

```
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.

Whereas an effective class described an implementation of an abstract data types, a deferred class describes a *set* of possible implementations. You may not instantiate a deferred class: **create** *v* is invalid if *v* is declared of type *VEHICLE*. But you may assign to *v* a reference to an instance of an effective 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
...
create c ...; create t ...;...
if "Some test" then v := c else v := t end
v.register (2003)
```

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 classes. Eiffel's Design by Contract mechanisms are essential here: you may 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 you, as a designer, to attach precise semantics to a module at the design stage long before you will make any implementation choices.

Beyond design and implementation, these techniques extend to the earliest stage of development, **analysis**. Deferred classes written at that stage describe not software objects, but objects from the external world being modelled — documents, airplanes, investments. Here again the presence of contracts to express constraints, and the language's other structuring facilities, provide an attractive combination.

Eiffel appears here in its full role of a lifecycle approach, covering areas traditionally considered separate: program implementation, the traditional province of development environments; system modeling and architecture, the traditional province of CASE tools based on UML or similar notations disconnected from the rest of the lifecycle. Eiffel instead emphasizes the fundamental unity of the software process and the usefulness of a single set of notations, concepts and tools applicable throughout. Such a *seamless* approach is indispensable to support the inevitable *reversals* that occur during the process of building software, such as detecting at implementation time a problem that leads to a change in the system's functionality, set at analysis time. The use of separate tools and notations, such as UML on one side and a programming language on the other, makes such round-trips difficult at best and often leads to monolithic, hard-to-change software. Eiffel lets you focus on the issues, without interposing artificial barriers between different software development activities. You'll use the fundamental problem-solving techniques — data abstraction through classes, precise specification through contracts, modularity through information hiding, rational organization through inheritance, decentralized architecture through dynamic binding, parameterization of the solution through genericity, reusability through all these techniques — all along; only the level of abstraction changes.

14 PUTTING A SYSTEM TOGETHER

We have now studied the constituents of Eiffel software. It remains to see how you can combine these elements into executable **systems** — the Eiffel concept closest to the traditional notion of program — and libraries.

How do you get an executable system? All you need is to

- Provide a set of classes, called a **universe**.
- Designate of these classes as the **root class**.
- Designate one of its creation procedures as the **root procedure**.

This defines what it means to execute the system: create one direct instance of the root class (the execution's **root object**); and call the root procedure on it. That's all.

In any practical case, the root procedure will create other objects, call other routines on them, leading to further creations and calls.

For the system to be valid, it must include all the classes which the root **needs** directly or indirectly; a class "needs" another if it is one of its heirs or clients.

We can generalize these notions to encompass a library rather than an executable system, by accepting *NONE* as root class. Since *NONE* inherits from all other classes, it needs all the classes in the universe; compiling with *NONE* as root will compile all classes. In this case you don't specify a root procedure, and the result is not executable.

The Eiffel method suggests grouping related classes — typically 5 to 40 classes — into collections called **clusters**. A universe is then a set of clusters. For example the EiffelBase library is divided into clusters corresponding each to a major category of data structure: *lists*, *tables*, *iteration* and so on. You can nest clusters, using for example EiffelBase, with its own subclusters as listed, as a cluster of your system.

How will you specify a universe? Any Eiffel implementation can use its own conventions. EiffelStudio applies a simple policy:

- Store each class in a single file, called its class file, with a name of the form *name*`.e`. For clarity, *name* should be the lower-case version of the class name, although this is a style rule, not a requirement.
- Put all the class files of a cluster into a single directory (folder on Windows), called its cluster directory.

  It is desirable for clarity, as a style rule, to separate clusters that directly contain classes ("terminal clusters") from those that have subclusters. Cluster directories will then contain class files or cluster subdirectories, but not both.
- To specify a system, it suffices to provide a list of cluster directories, along with the name of the root class and root procedure. The universe consists of the classes contained in all the class files in the listed cluster directories.

Here is an example of such a system specification. It's *not* written in Eiffel, although it definitely has an Eiffel-like flavor. It is called an **Ace** (Assembly of Classes in Eiffel) and written in the Eiffel-like **Lace** notation (Language for the Assembly of Classes in Eiffel). The backward slash **** is the Windows path separator; Unix uses a forward slash **/** for the same purpose (EiffelStudio will accept both on Windows).

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

With EiffelStudio you don't have to write the Ace yourself; just specify the information interactively through the Preferences dialog and EiffelStudio will generate the Ace. You can also reuse and adapt one of the many example Aces in the delivery.

So you don't need to learn the syntax of Lace, although as the example shows it is straightforward. The Ace first gives the system a name, *example*, which will also serve as the name of the generated executable. It then specifies the root class, its cluster (optional), and the root procedure. Next, in the **default** clause, come compilation options; you can specify assertion monitoring, with choices that include **none**, **require** (preconditions only, the default), **ensure** (preconditions and postconditions, as here) and **invariant** (the previous two plus class invariants). You can also specify various levels of assertion monitoring separately for a cluster, or for a specific class. The **precompiled** option specifies use of a precompiled library, EiffelBase, at the path given. The Ace ends with a list of clusters, other than those of EiffelBase, specifying for a cluster name, such as *my_cluster1*, and the directory where it resides.

The path names in this example use the Windows path separator, a backward slash ****. Unix uses a forward slash **/** (also acceptable on Windows) for the same purpose.

This Ace is from an example in the delivery. If you compile and execute it, it will create an instance of class *CALCULATOR* and call its *make* procedure. This starts a small interactive calculator, illustrating some of the simple mechanisms of EiffelBase.