Eiffel: An Introduction

Author: Bertrand Meyer.

Presented by: Philip Hucklesby.

Société des Outils du Logiciel
14 rue Jean Rey
75015 Paris
France

Abstract

The Eiffel language and environment apply the concepts of object-oriented design and programming to the construction of high quality software.

As a language, Eiffel provides a range of features for the construction of reusable and reliable software components: classes, multiple inheritance, polymorphism and dynamic binding, genericity, strict static type checking, a disciplined exception mechanism, systematic use of assertions, invariants and other constructs for ensuring program correctness.

Eiffel is implemented by compilation through C, ensuring wide portability. On option, a stand alone C package, movable to any machine supporting C, may be generated from the text of an Eiffel system. However the language itself is an original design and has no relation to C.

The environment ensures separate compilation of Eiffel classes; it also takes care of recompilation mangagement, automatically triggering re-compilation of modified classes without programmer intervention. The tools of the environment include facilities for automatic documentation (producing a class interface description from the claass texts), class browsers, an interactive debugger, a system for graphical display of class hierarchies, an optimising postprocessor and other facilities.

An important part of the environment is a set of libraries of reusable software components (classes). They include the Data Structure Library, which covers fundamental data structures and algorithms, Graphics library, supporting windowing and bit-mapped graphics, and the Parsing Library, for writing compilers ad other parser-based tools.

WHAT IS EIFFEL?

Eiffel addresses the software industry's need for a modern language and development environment supporting the analysis, design and implementation of high-quality software.

The language is based on advanced principles of object-oriented programming, with a special emphasis on the reliability or the resulting software components and systems.

The supporting **environment**, available on a number of hardware platforms, includes tools for such tasks as automatic compilation management, C package generation, class browsing, automatic documentation and debugging.

A complementary component of Eiffel is the set of **libraries** of pre-packaged reusable software components covering many of the common tasks of software development, from fundamental data structures and algorithms to graphics, parsing, window management and others.

Beyond the language, environment and libraries, Eiffel is also a **method** of software construction by combination of self-contained and flexible modules, and opens the perspective of a true *industry* of reusable software components.

The present article is a general introduction to Eiffel. More detailed information [1, 2, 5-7] is available. A recent book [3], explores the issues of object-oriented software engineering in depth, and explains the Eiffel method of software design and implementation.

DESIGN PRINCIPLES

Software quality is a combination of many factors. In the current state of the industry, some of these factors are in dire need of improvements. One is *reusability*, or the ability to produce components that may be used in many different applications. Another is *extendibility*: "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 the other goals (such as efficiency, ease of use or integrity).

The third fundamental factor is *reliability*. Techniques such as assertions, disciplined exception handling and static typing, enabling developers to produce software with dramatically fewer bugs, are part of the distinctive Eiffel approach to the engineering of quality software.

Other requirements were *portability* of the implementation, and *efficiency* of Eiffel-generated software in both time and space, a concern that could not be neglected in a tool aimed at practical, medium- to large-scale industrial developments.

OBJECT-ORIENTED DESIGN

To achieve reusability and extendibility, the principles of object-oriented design seem to provide the best known technical answer. An in-depth discussion of these principles would fall beyond the scope of this introduction (see [3]), but we need a definition. Object-oriented design is the **construction of software systems as structured collections of abstract data type implementations.** The following points are worth noting in this definition:

- The emphasis is on structuring a system around the classes of objects it manipulates rather than the functions it performs on them, and on reusing whole data structures, together with the associated operations, rather than 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, 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 important relationships between classes, particularly the **multiple inheritance** relation.

Eiffel results from a systematic effort to apply the full extent of object-oriented technology, without the compromises that have marred previous object-oriented language designs.

Eiffel as a language includes more than presented in this introduction, but not *much* more; it is a small language, comparable in size (by such a measure as the number of keywords) to 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 a number of full-scale projects, some in the hundreds of thousands of lines, in a number of companies.

CLASSES

A class, it was said above, represents an implementation of an abstract data type, that is to say a set of run-time objects characterized by the operations available on them (the same for all instances of a given class), and the properties of these operations. These objects are called the **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.

A simple example is a class *ACCOUNT* describing bank accounts. Before presenting the class itself, it is useful to illustrate how it may be used by other classes, called its **clients**.

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

acc1: ACCOUNT

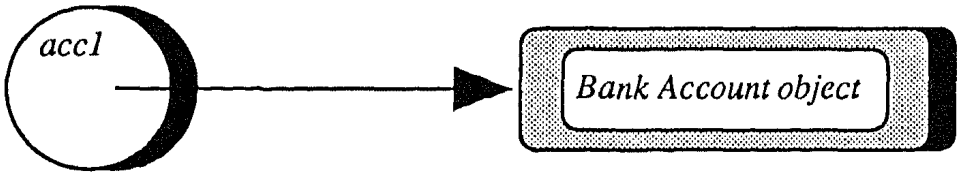

Figure 1: Entity and associated object

The term "entity" generalizes the more common notion of "variable". An entity declared of a class type, such as *acc1*, may at any time during execution become attached to an object; the type rules imply that this object must be an instance of *ACCOUNT* (or, as seen below, of a "descendant" of that class). An entity which is not attached to any object is said to be void. By default (at initialization) entities are void; an object is created by an instruction

acc1.Create

which attaches *acc1* to the newly created object. *Create* is a predefined "feature" of the language.

Once the client has associated *acc1* with an object, it may apply to it the features defined in class *ACCOUNT*. Examples are:

```
acc1.open ("Jill");
acc1.deposit (5000);

if acc1.may_withdraw (3000) then
   acc1.withdraw (3000)
end;
acc1.balance.print
```

Most feature applications use the dot notation: *entity_name.feature_name*. (Prefix and infix form, described below, are also available.) There are two kinds of features: **routines** (as *open*, *deposit*, *may_withdraw* or *withdraw*), representing operations applicable to instances of the class; and **attributes**, representing data items associated with these instances.

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

The above extract of class *X* does not show whether, in class *ACCOUNT*, *balance* is an attribute or an argumentless function. This ambiguity is intentional. A client of *ACCOUNT*, such as *X*, does not need to know how a balance is obtained: it could be stored as an 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.

Here now is a first sketch of how class *ACCOUNT* itself might look. Line segments beginning with -- are comments.

```
class ACCOUNT export
   open, deposit, may_withdraw,
   withdraw, balance, minimum_balance, owner
feature
   balance: INTEGER ;

   minimum_balance: INTEGER is 1000 ;

   owner: STRING ;

   open (who: STRING) is
         -- Assign the account to owner who
      do
         owner := who
      end; -- open

   add (sum: INTEGER) is
         -- Add sum to the balance
         -- (Secret procedure)
      do
         balance := balance + sum
      end; -- add

   deposit (sum: INTEGER) is
         -- Deposit sum into the account
      do
         add (sum)
      end; -- deposit
```

```
    withdraw (sum: INTEGER) is
        -- Withdraw sum from the account
      do
        add (-sum)
      end; -- withdraw
    may_withdraw (sum: INTEGER): BOOLEAN is
        -- Is there enough to withdraw sum?
      do
        Result :=
          (balance >= sum + minimum_balance)
      end -- may_withdraw
end -- class ACCOUNT
```

This class includes two clauses: **feature**, which describes the features of the class, and **export**, which lists the names of features available to clients of the class. Non-exported features are said to be secret. Here procedure *add* is secret, so that *acc1.add (-3000)* would be illegal in *X*. Attribute *minimum_balance* is also secret.

Let us examine the features in sequence. The **is ... do ...end** distinguishes routines from attributes. So here *balance* has been implemented as an attribute. The clause **is** *1000* introduces *minimum_balance* as a constant attribute, which will not occupy any physical space in objects of the class. Non-constant attributes such as *balance* do use space for each object of the class; they are similar to components of a record in Pascal.

The language definition guarantees automatic initialization, so that the initial *balance* of an account object will be zero after a *Create*. The initial values are zero for numeric attributes, false for booleans, null characters for characters, and void references for entities of class types.

The other five features are straightforward routines. The first four are procedures, the last one (*may_withdraw*) a function returning a boolean value. The special variable *Result* denotes the function result; it is initialized on function entry to the default value of the appropriate type, as defined above.

To understand the routines fully, you must remember that in Eiffel's object-oriented programming style any operation is relative to a certain object. In an external client invoking the operation, this object is specified by writing the corresponding entity on the left of the dot, as *acc1* in *acc1.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". The special variable *Current* may be used, if needed, to denote this object explicitly. For example the unqualified occurrences of *add* appearing in the above class are equivalent to *Current.add*.

In some cases, infix or prefix notation is more convenient. For example, most people will prefer calling the addition routine of a class *VECTOR* under the form *v + w* rather than *v.plus(w)*. This is indeed possible if you call the routine **infix** "+" rather than *plus*. Internally, the operation is still a 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.

TYPES

Eiffel is strongly typed. Every entity is declared of a certain type; a type may be either a class type or an expanded type.

A class type is simply defined by a class, such as *ACCOUNT*. As noted above, the run-time value of an entity declared of a class type is a reference to potential objects (instances of the class). Class types include such types as *ARRAY* and *STRING*, described by classes in the Basic Eiffel Library.

In contrast with class types, values of an expanded type are objects, not references to objects. Expanded types include the basic predefined 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 or infix operators. as a result, the Eiffel type system is fully regular and consistent: every type, including the basic types, is defined from a class, either as a class type or as an expanded type. (Of course, in the case of basic types, the compiler implements the standard arithmetic and boolean operations directly, not through routine calls; but this is only an optimization, which does not hamper the conceptual homogeneity of the type edifice.)

ASSERTIONS

Classes are defined as abstract data type implementations. What defines an abstract data type, however, is not just the available operations, but also the formal properties of these operations, which do not appear in the above example.

Eiffel enables and encourages programmers 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 withdrawal might only be permitted if it keeps the account's balance on 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.
- Class **invariants** must be satisfied by objects of the class at all times, or more precisely after object creation and after any call to a routine of the class. They are described in the **invariant** clause of the class and represent general consistency constraints that are imposed on all routines of the class.

With appropriate assertions, the *ACCOUNT* class becomes:

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

```
  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; -- withdraw
  may_withdraw ... -- as before
  Create (initial: INTEGER) is
    require
      initial >= minimum_balance
    do
      balance := initial
    end -- Create
invariant
  balance >= minimum_balance
end -- class ACCOUNT
```

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

This class now includes a specific *Create* procedure, as needed when the default initializations are not sufficient. Under the previous scheme, an account was created by, say, *acc1.Create*. Because of the initialization rules, *balance* is then zero and the invariant is violated. If a different initialization is required, possibly requiring (as here) client-supplied arguments, the class should include a procedure called *Create*. The effect of

```
acc1.Create (5500)
```

is to allocate the object (as in the default *Create*) and to call the procedure called *Create* in the class, with the given argument. This call is correct as it satisfies the precondition and ensures the invariant. (Procedure *Create*, when provided, is recognized as special; it is automatically exported and should not be included in the **export** clause.)

Syntactically, assertions are boolean expressions, with a few extensions (like the **old** notation). The semicolon (see the precondition to *withdraw*) is equivalent to an "and", but permits individual identification of the components, useful for producing informative error messages when assertions are checked at run-time.

Assertions play a central role in the Eiffel method for building reliable object-oriented software. They serve to make explicit the assumptions on which programmers rely when they write program fragments 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 *programming by contract* [3, 4] views software construction as based on contracts between clients (callers) and suppliers (routines), relying on mutual obligations and advantages made explicit by the assertions. As will be seen below, this theory also explains much of the meaning of inheritance, and lies at the basis of Eiffel's disciplined exception mechanism.

Assertions are also an indispensable tool for the documentation of reusable software components: as with hardware 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). The documentation tools of

the Eiffel environment, such as **short** (studied below), use assertions to produce information for client programmers, describing classes in terms of observable behavior, not implementation.

Assertions may be indeed be monitored at run-time; since such monitoring may penalize the performance, it is enabled on option, class by class. (For each class, two levels of monitoring are possible: preconditions only or all assertions.) This provides a powerful debugging tool, in particular because the classes of the Basic Eiffel Library, which are widely used in Eiffel programming, are protected by carefully written assertions. A violated assertion will trigger an exception, as described below; unless the programmer has written an appropriate exception handler, the exception will cause an error message and termination.

Run-time checking, however, is only one application of assertions, whose role as design and documentation aids exerts a strong influence on the Eiffel programming style.

EXCEPTIONS

Whenever there is a contract, the risk exists that someone will break it. This is where exceptions come in. (Other references [3, 4, 8] describe the Eiffel exception mechanism in more detail.)

An exception may arise from one of several causes. When assertions are monitored, an assertion violation will raise an exception. Another cause is the occurrence of a hardware-triggered abnormal signal, arising for example from arithmetic overflow or a failure to find the memory needed for allocating an 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.

A routine may, however, handle an exception through a **rescue** clause. This optional clause attempts to "patch things up" by bringing the current instance 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. This will cause the routine to restart its execution from the beginning, attempting again to fulfil the routine's 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.)

Note that a routine with no rescue clause is considered to have an empty rescue clause, so that any exception occurring during the execution of the routine will cause the routine to fail immediately.

The underlying 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.

This should be contrasted with the Ada exception mechanism, which does not rely on any notion of contract. This encourages writing routines that will fail to achieve their purpose but do not notify the caller because the exception is handled locally. Such examples, which may even be found in Ada textbooks (see example quoted in [4]), violate the above principle by involving routines that neither succeed (they didn't fulfil their job) nor fail (the caller is not notified and continues its execution on the false assumption that the call was normally completed). This is the reason why the Ada mechanism is so dangerous. Also note that the **retry** instruction must often be implemented in Ada using intricate control structures: **goto** instructions and multi-level loop exits [4].

An example of the Eiffel exception mechanism is a routine *attempt_transmission* that transmits 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 5 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
    -- Attempt transmission of message,
    -- at most 5 times.
    -- Set successful accordingly.
  local
    failures: INTEGER
  do
    if failures < 5 then
      transmit (message);
      successful := true
    else
      successful := false
    end
  rescue
    failures := failures + 1;
    retry
  end; -- attempt_transmission
```

The integer local variable *failures* is initialized to zero on entry.

This example shows one of the key reasons for the simplicity of the mechanism: the rescue clause never attempts to achieve the original intent of the routine; this is the sole responsibility of the normal body (the **do** clause). The only role of the rescue clause is to "patch things up" and either fail or retry.

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

MULTIPLE INHERITANCE

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 Eiffel techniques address the problem: inheritance, studied in this section, and genericity, studied below.

Multiple inheritance is a key technique for reusability. The basic idea is simple: when defining a new class, it is often fruitful to introduce it by combination and specialization of existing classes rather than as a new entity defined from scratch.

The following simple example, from the Basic Library, is typical. *LIST*, as indicated, describes lists of any representation. One possible representation for lists with a fixed number of elements uses an array. Such a class will be defined by combination of *LIST* and *ARRAY*, as follows:

```
class FIXED_LIST [T] export ...
inherit
  LIST [T];
  ARRAY [T]
feature
  ... Specific features of fixed-size lists ...
end -- class FIXED_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 *FIXED_LIST* as shown ensures that all the features and properties of lists and arrays are applicable to fixed lists as well.

Another example is extracted from a windowing system based on a class *WINDOW*. Windows have **graphical** features: a height, a width, a position etc., with associated routines to scale windows, move them and so on. 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 much preferable to inherit all hierarchical features from *TREE* (one of a number of classes in the Basic Eiffel Library describing tree implementations), and all graphical features from a class *RECTANGLE*.

Multiple inheritance yields remarkable economies of programming effort and has a profound effect on the software development process.

The very power of the mechanism demands adequate means to keep it under control. In Eiffel, no name conflict is permitted between inherited features. Since name conflicts inevitably arise in practice, especially for classes contributed by independent developers, the language provides a technique to remove them: **renaming**, as in

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

Here the **inherit** clause would be illegal without renaming, since both *A* and *B* have features named *x* and *y*.

Renaming also serves to provide more appropriate feature names in descendants. For example, class *WINDOW*, as mentioned, inherits routines such as *insert_subtree* from *TREE*. For clients of *WINDOW*, however, such routine names are not 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 it is appropriate to rename *insert_subtree* as *add_subwindow* in the inheritance clause of *WINDOW*.

As further incentive not to misuse 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.

POLYMORPHISM AND DYNAMIC BINDING

Inheritance is not just a module combination and enrichment mechanism. It also enables the definition of flexible program entities that may become attached to objects of various forms at run-time (hence the term "polymorphic").

In Eiffel, this remarkable facility is reconciled with static typing. The underlying 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 class types 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 the fact 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 feature of a class may be redefined in any descendant class; the type of the redefined feature (if an attribute or a function) may be redefined as a descendant type of the original feature, and, in the case of a routine, its body may also be replaced 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 be:

```
class RECTANGLE export ... inherit
  POLYGON redefine perimeter
feature
    -- Specific features of rectangles, such as:
  side1: REAL; side2: REAL;

  perimeter: REAL is
      -- Rectangle-specific version
    do
      Result := 2 star (side1 + side2)
    end; -- perimeter
  ... 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.Create; r.Create; ...
if c then p := r end;
print (p.perimeter)
```

The 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**. We shall see below that it is implemented in Eiffel without negative effects on run-time performance.

Dynamic binding provides a high degree of flexibility. The key 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 every client must know about all possible cases, and that any extension may invalidate a large body of existing software. The Ada facilities for overloading and genericity do not bring any improvement in this respect, since they do not support a programming style in which a client module may issue a request meaning: "compute the perimeter of *p*, using the algorithm appropriate for whatever form *p* happens to have when the request is executed".

In contrast, dynamic binding and inheritance support a development mode in which every module is *open* and incremental: an existing class may always be given a new descendant (with new and/or redefined features) 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.

Eiffel handles polymorphism and dynamic binding in a disciplined way. First, feature redefinition, as seen above, is explicit. Second, because the language is typed, the compiler can check statically whether a feature application *a.f* is correct. In contrast, languages such as Smalltalk and its descendants 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 "method" (routine); if not, a run-time error will occur. Such errors may not happen during the execution of a correctly compiled Eiffel system. In other words, Eiffel 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) is automatically selected. Static typing means that the compiler makes sure there is *at least one* such version.

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

Eiffel's 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? One way to maintain the semantic consistency of routines throughout their redefinitions is to use preconditions and postconditions, which are binding on redefinitions. More precisely, any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original. Thus, by making the semantic constraints explicit, routine writers may limit the amount of freedom granted to eventual redefiners.

These rules should be understood in light of the contract theory. 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 by the contract accepted by the prime contractor: it may not impose stronger requirements on the clients (but may accept more general requests, which is why the precondition may be weaker); and it must achieve at least as much as promised by the original contractor (but may achieve more, which is why the postcondition may be stronger).

DEFERRED CLASSES

The inheritance mechanism includes one more major component: A deferred class is a class which contains at least one deferred routine; a routine is declared as deferred to express that implementations of the routine will only be provided in descendants. For example, a system used by the Department of Motor Vehicles to register vehicles could include a class of the form

```
deferred class VEHICLE export
  dues_paid, valid_plate, register, ...
feature
  dues_paid (year: INTEGER): BOOLEAN is
    ...
    end; -- dues_paid
  valid_plate (year: INTEGER): BOOLEAN is
    ...
    end; -- valid_plate
```

```
    register (year: INTEGER) is
        -- Register vehicle for year
      require
        dues_paid (year)
      deferred
      ensure
        valid_plate (year)
      end; -- register
  ... Other features ...
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. Effective versions of the routine are given in of class *VEHICLE*, such as *CAR* or *TRUCK*. They are similar to redefined versions of a routine; only here there is no effective definition in the original class, only a specification in the form of a deferred routine.

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.Create* is illegal 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 correct:

```
v: VEHICLE; c: CAR; t: TRUCK;
...
c. Create (...); t.Create (...);
...
if "some test" then v := c else v := t end;
v.register (1988)
```

The mechanisms of polymorphism and dynamic binding are fully exploited here: 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. Note that "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 for the application of Eiffel at the analysis and design stages. 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 to associate a precondition and a postcondition to a routine even though it is a deferred routine (as with *register* above), and an invariant to a class even though it is a deferred class. This enables the designer to attach a precise semantics to a module at the analysis or design stage, long before making any implementation choices.

The combination of deferred classes and assertions makes Eiffel a more appropriate tool for high-level design than existing "PDLs" (Program Design Languages). Common PDLs offer no facility comparable to Eiffel assertions for describing the semantics of routines independently of their implementations. The further benefit with Eiffel, of course, is that it is not just a design language, but can be efficiently executed as well, so that that no conceptual gap is introduced between design and programming.

GENERIC CLASSES

Together with inheritance, Eiffel's genericity is essential for writing flexible, parameterized classes. Genericity allows classes to have generic parameters, representing types. The following examples come from the Basic Library:

ARRAY [*T*]
LIST [*T*]
LINKED_LIST [*T*]

They respectively describe one-dimensional arrays, general lists (without commitment to a specific representation) and lists in linked representation. Each has a formal generic parameter *T* representing an arbitrary type. To use these classes, you provide actual generic parameters, which may be either simple or class types, as in the following declarations:

il: *LIST* [*INTEGER*];
aa: *ARRAY* [*ACCOUNT*];
aal: *LIST* [*ARRAY* [*ACCOUNT*]] -- etc.

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

Genericity may be **constrained**: 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 make it satisfy its own generic constraint, enabling the definition of types such as *VECTOR* [*VECTOR* [*T*]].

An earlier article [2] discussed the role of genericity in comparison to inheritance and explained their combination in Eiffel, especially in the context of strict type checking. An important component of the solution, required to guarantee type consistency, is the notion of "declaration by association" [2, 3, 6].

THE IMPLEMENTATION

The original Eiffel implementation ran on various versions of Unix (System V, 4.3BSD, Xenix) and has been ported to more than thirty different machine architectures. Versions of Eiffel are under way for VAX-VMS, OS/2 and other architectures.

The compiler uses C as intermediate language, making Eiffel potentially portable to any environment supporting C. An important consequence of this technique is the ability to use the Eiffel compiler as a cross-development tool, generating a self-contained C package as end product; this is explained in the next section. Another advantage is the ease of interfacing Eiffel with existing software written in C or other languages, as discussed below.

Although the compiler relies on C for practical purposes, Eiffel is in no way a C extension; as the above discussion should suffice to show, C had no influence on the language design. The use of C as implementation vehicle presents a number of advantages, but is only one possible technique.

Great care has been taken to provide efficient compilation and execution, so that the environment would support the development of serious software. The following points are particularly worth noting.

- Redefinition and dynamic binding imply that a qualified routine reference, say *p.perimeter*, may have many different interpretations depending on the value of *p* at run-time. As noted above, a run-time search for the appropriate routine, as implemented in many systems, would carry a heavy performance penalty. The maximum search length grows with the depth of the inheritance graph, putting reusability (which tends to increase this depth) and extendibility (which promotes redefinition) at odds with efficiency. With multiple inheritance, run-time search becomes hopeless: a complete graph of ancestor classes, not just a linear list, would have to be searched at run-time.

In contrast, the Eiffel implementation always finds the proper routine in constant time, with only a small penalty over the cost of procedure call. This result was difficult to achieve, but essential in light of the previous discussion. Regardless of the amount of redefinition and polymorphism in your system, *a.f* always takes the same (small) time.

• There is almost never any code duplication. Again this was difficult to achieve with multiple inheritance and genericity. For example, most Ada implementations duplicate code for every instance of a generic module.

• The run-time system handles object creation and memory de-allocation. It includes an incremental garbage collector, implemented as a coroutine which steals only negligible time from application programs. Automatic garbage collection is essential to the application of object-oriented techniques to real developments (as opposed to toy experiments). Object-oriented applications, which typically create many objects, should not be polluted with complex, error-prone memory management code.

Garbage collection may be turned off (for example during initialization); the collector coroutine may also be explicitly activated for a certain time at points where the programmer knows some CPU time is available (for example while awaiting user input in an interactive application).

• Compilation is performed on a class-by-class basis, so that large systems can be changed and extended incrementally. The Eiffel to C translation time is usually about half of the time for the next step, C to machine code.

Also important in practice is the openness of the environment. Eiffel classes are meant to be interfaced with code written in other languages. An environment promoting reusability should enable reuse of software built with other approaches, in particular if it was developed prior to its introduction.

Openness is supported by both call-out and call-in mechanisms. Call-out is achieved through the optional **external** clause of routine declarations, which lists needed external subprograms. For example, a square root routine might rely on an external function:

```
sqrt (x: REAL, eps: REAL): REAL is
    -- Square root of x with precision eps
  require
    x >= 0 ; eps > 0
  external
    csqrt (x: REAL, eps: REAL): REAL
  do
    Result := csqrt (x, eps)
  ensure
    abs (Result ^ 2 – x) <= eps
  end -- sqrt
```

It is possible with this mechanism to communicate with other languages without impacting the conceptual consistency of the Eiffel classes. Note in particular how the C function *sqrt* is granted a more dignified status as Eiffel routine with the addition of a precondition and a postcondition.

Thanks to the call-out facilities, Eiffel can serve as an integrating mechanism for components written in other languages. Using Eiffel classes and all the associated structuring facilities (multiple inheritance, genericity, export controls, assertions), you may be package elements written in other languages, such as numerical routines, graphics primitives, database facilities etc., into clean and convenient modules.

The external interface also supports call-in: software elements written in other languages may create Eiffel objects, call routines on these objects, and access their attributes.

THE ENVIRONMENT

The construction of systems in Eiffel is supported by a set of development tools.

Most important are the facilities for automatic compilation management integrated in the compilation command es (Eiffel System). When compiling a class *C*, es automatically looks for all classes on which *C* depends directly or indirectly (as client or heir), and recompiles those whose compiled versions are obsolete. Unix programmers will recognize this facility as giving the power of Make, but there is a fundamental difference: instead of having to describe manually the dependencies between modules, a tedious and error-prone process, the Eiffel programmer lets es take care of analyzing these dependencies automatically.

This problem is far from trivial because dependency relations are complex (a class may be, say, a client of one of its descendants) and, in the case of the client relation, may involve cycles. The Eiffel solution totally frees programmers from having to keep track of changes to maintain the consistency of their systems. The algorithm avoids many unneeded recompilations by detecting modifications of a class that do not affect its interface, so that clients need not be recompiled. This has proved very important in practice, preventing a chain reaction of recompilations in a large system when a feature implementation is changed in a low-level class.

The possibilities offered by Eiffel techniques for producing high-quality reusable software components mean that users need computerized support for finding out about available classes and their properties. A set of **browsing** facilities is offered for that purpose. They include a full-screen browser, **eb** (Eiffel browser), and a graphical browser, **GOOD** (Graphics for Object-Oriented Design), which shows the information in graphical, "bubbles and arrows" form. Both make it possible to query the environment on available classes on the basis of their features and other properties. Both **eb** and **GOOD** are human interfaces to a set of facilities which are directly available to any Eiffel application as the exported features of a library class, *E_CLASS*.

GOOD is actually not just a browser but also a design aid which allows developers to enter new system structures and class relations (client, multiple inheritance) using a graphical, mouse-driven interface, and will automatically generate the skeletons of the corresponding class texts.

The environment also offers **debugging** tools: run-time checking of assertions; a tracer; a full-screen source-level debugger, which lets programmers do their debugging in terms of object and classes (in contrast with to a traditional, function-oriented debugger), and enables them to explore the object structure at run-time.

A documentation tool, **short**, produces a summary version of a class showing the interface as available to clients: the exported features only and, in the case of routines, only the header, precondition and postcondition. The manual for the Basic Eiffel Library [5] is an example of Eiffel documentation produced almost entirely from output generated by **short**. Such documentation is essentially obtained "for free" and, even more importantly, is guaranteed to be consistent with the documented software, as it is extracted from it. This should be contrasted with classical approaches, which view software and documentation as separate products.

A related tool is **flat**, which produces inheritance-free versions of classes. When applied to a class, **flat** yields an equivalent class text where all inherited features have been expanded, taking into account renaming and redefinition. This makes it possible to distribute a self-contained version of the class and, in combination with **short**, to produce more complete documentation.

The **postprocessor**, part of **es**, performs optional optimizations: removal of unneeded routines; application of static binding to routines for which dynamic binding is not necessary; in-line expansion of routine calls satisfying appropriate criteria. All these optimizations are performed safely without any programmer intervention. They are essential to allow programmers to enjoy the elegant and highly modular programming style encouraged by Eiffel without paying an unacceptable performance overhead.

The postprocessor doubles as **cross-development** tool. On option, it will generate from an Eiffel system a stand-alone **C package**, complete with an automatically generated Make file and a copy of the run-time system (garbage collector, etc.), in C form. The result may be ported to any

environment supporting C. This facility makes it possible to design and implement software in Eiffel, and deliver it in C form. Eiffel need not be available on the target environments.

LIBRARIES

A key part of the Eiffel approach is the ability to rely on predefined libraries of reusable classes, collectively known as the Basic Eiffel Library.

The Data Structure Library is a repertoire of carefully designed classes covering many of the most important data structures and algorithms of everyday programming. Use of the library is one of the elements that give Eiffel programming its distinctive flavor, enabling programmers to reason and write in terms of lists, trees, stacks, hash tables etc. rather than arrays, pointers, flags and the like.

The Graphics Library enables users to manipulate windows, menus, geometrical figures and other graphical objects. The Graphics Library is internally based on the MIT X-Windows system, but hides the lower-level X concepts from programmers, who only need to think of abstract graphical concepts.

The Parsing Library offers tools for building compilers, interpreters and other parser-based tools with a clear object-oriented structure deduced naturally from grammar structure.

The Windowing Library supports the development of window-based, non-graphical applications.

A number of new libraries are under development. The Eiffel Software Shelf, a repertory of user-supplied libraries, allows for the quick dissemination of reusable components developed by Eiffel programmers the world over.

CONCLUSION

We believe that Eiffel is the first language to combine the most advanced ideas of object-oriented languages with the modern concepts of software engineering, and to make the result available to practicing software developers. Since its introduction in 1986, many companies have used Eiffel to produce systems in highly diverse fields. This experience consistently shows tremendous improvements in quality and productivity. Academic users have also found in Eiffel a powerful tool for teaching courses throughout the software curriculum, from introductory programming to algorithm design, advanced data structures and software engineering.

Technically, Eiffel brought in a number of important innovations. Among the new contributions are, from the language standpoint, the safe treatment of multiple inheritance through renaming, the combination between genericity and inheritance, disciplined polymorphism by explicit redefinition, the assertion mechanism and its combination with inheritance, a clean interface with external routines, the introduction of full static typing into an object-oriented language, the disciplined approach to exception handling based on the notion of contract, the clean dynamic model distinguishing between class types and expanded types, the systematic approach (not described in this presentation) to indexing library classes and phasing out obsolete library features. From the implementation and environment viewpoint, a number of our solutions are also original: constant-time feature access, separate compilation with automatic recompilation, coroutine garbage collection, object-oriented debugging and documentation, cross-development of portable C software packages, property-based browsing, graphical representations of class structures.

Perhaps even more important than the innovations is the coherence and completeness of the entire language and environment design. Each facility is essential to the integrity of the system, and was conceived in relation to the others: genericity with multiple inheritance; dynamic binding with static typing; polymorphism with the constant-time routine access mechanism and the automatic application of static binding by the postprocessor whenever appropriate; assertions with

exceptions; redefinition of routines with the redefinition of assertions; assertions (again) with the short and flat automatic documentation tools; and so on. Powerful as we may hope each of these facilities to be on its own right, the whole is more than the sum of its parts.

References

Documents [3] to [7] are available from Interactive Software Engineering. [3] may also be found in technical bookstores. A report entitled "An Eiffel Collection", available from Interactive, contains the major journal articles on Eiffel, including [1], [2] and [8].

References

1. Bertrand Meyer, "Reusability: the Case for Object-Oriented Design," *IEEE Software*, vol. 4, no. 2 , pp. 50-64, March 1987.
2. Bertrand Meyer, "Genericity, static type checking, and inheritance," *The Journal of Pascal, Ada and Modula-2*, 1988. (Original version in OOPSLA 86 proceedings, SIGPLAN Notices, Sept. 1986, pp. 391-405.)
3. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, 1988.
4. Bertrand Meyer, "Programming as Contracting," Technical Report TR-EI-12/CO, Interactive Software Engineering, Santa Barbara (Calif.), 1988.
5. Bertrand Meyer, "Eiffel: The Libraries," Technical Report TR-EI-7/LI, Interactive Software Engineering Inc., Santa Barbara (Calif.), October 1986 (version 2.2, August 1989). (To be published by Prentice-Hall in 1990.)
6. Bertrand Meyer, "Eiffel: The Language," Technical Report TR-EI-17/RM, Interactive Software Engineering Inc., Santa Barbara (Calif.), 1989. (To be published by Prentice-Hall in 1990.)
7. Bertrand Meyer, "Eiffel: The Environment," Technical Report TR-EI-5/UM, Interactive Software Engineering Inc., Santa Barbara (Calif.), 1989. (To be published by Prentice-Hall in 1990.)
8. Bertrand Meyer, "From Structured Programming to Object-Oriented Design: The Road to Eiffel," *Structured Programming*, vol. 10, no. 1, pp. 19-39, 1989.