1 OVERVIEW

Eiffel is a *method* and *language* for the efficient description and development ofquality systems.

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

•*Analysis, modeling* and *specification*, where Eiffel can be used as a purelydescriptive tool to analyze and document the structure and properties of complex systems (even non-software systems).

• *Design* and *architecture*, where Eiffel can be used to build solid, flexible system structures.

•*Implementation*, where Eiffel provides practical software solutions with an efficiency comparable to solutions based on such traditional approaches as C and Fortran.

• *Maintenance*, where Eiffel helps thanks to the architectural flexibility of the resulting systems.

• *Documentation*, where Eiffel permits automatic generation of documentation, textual and graphical, from the software itself, as a partial substitute for separately developed and maintained software documentation.

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

• *Reliability*: producing bug-free systems, which perform as expected.

• *Reusability*: making it possible to develop systems from prepackaged, high-quality components, and to transform software elements into such reusable components for future reuse.

• *Extendibility*: developing software that is truly soft — easy to adapt to the inevitable and frequent changes of requirements and other constraints.

• *Portability*: freeing developers from machine and operating system peculiarities, and enabling them to produce software that will run on many different platforms.

• *Maintainability*: yielding software that is clear, readable, well structured, and easy to continue enhancing and adapting.

2 GENERAL PROPERTIES

Here is an overview of the facilities supported by Eiffel:

• Completely *object-oriented* approach. Eiffel is a full-fledged application of object technology, not a “hybrid” of O-O and traditional concepts.

• *External interfaces*. Eiffel is a software composition tool and is easily interfaced with software written in such languages as C, C++, Java and C#.

• *Full lifecycle support*. Eiffel is applicable throughout the development process, including analysis, design, implementation and maintenance.

• *Classes as the basic structuring tool*. A class is the description of a set of run-time objects, specified through the applicable operations and abstract properties. An Eiffel system is made entirely of classes, serving as the only module mechanism.

• *Consistent type system*. Every type is based on a class, including basic types such as integer, boolean, real, character, string, array.

• *Design by Contract*. Every system component can be accompanied by a precise specification of its abstract properties, governing its internal operation and its interaction with other components.

• *Assertions*. The method and notation support writing the logical properties of object states, to express the terms of the contracts. These properties, known as assertions, can be monitored at run-time for testing and quality assurance. They also serve as documentation mechanism. Assertions include preconditions, postconditions, class invariants, loop invariants, and also appear in “check” instructions.

• *Exception handling*. You can set up your software to detect abnormal conditions, such as unexpected operating system signals and contract violations, correct them, and recover

• *Information hiding*. Each class author decides, for each feature, whether it is available to all client classes, to specific clients only, or just for internal purposes.

• *Self-documentation*. The notation is designed to enable environment tools to produce abstract views of classes and systems, textual or graphical, and suitable for reusers, maintainers and client authors.

• *Inheritance*. You can define a class as extension or specialization of others.

• Redefinition. An inherited feature (operation) can be given a different implementation or signature.

• *Explicit redefinition.* Any feature redefinition must be explicitly stated.

• *Subcontracting.* Redefinition rules require new assertions to be compatible with inherited ones.

• *Deferred features and classes*. It is possible for a feature, and the enclosing class, to be specified — including with assertions — but not implemented. Deferred classes are also known as abstract classes.

• *Polymorphism.* An entity (variable, argument etc.) can become attached to objects of many different types.

• *Dynamic binding*. Calling a feature on an object always triggers the version of the feature specifically adapted to that object, even in the presence of polymorphism and redefinition.

• *Static typing.* A compiler can check statically that all type combinations will be valid, so that no run-time situation will occur in which an attempt will be made to apply an inexistent feature to an object.

• *Assignment attempt* ("type narrowing"). It is possible to check at run time whether the type of an object conforms to a certain expectation, for example if the object comes from a database or a network.

• *Multiple inheritance.* A class can inherit from any number of others.

• *Feature renaming.* To remove name clashes under multiple inheritance, or to give locally better names, a class can give a new name to an inherited feature.

• *Repeated inheritance:* sharing and replication. If, as a result of multiple inheritance, a class inherits from another through two or more paths, the class author can specify, for each repeatedly inherited feature, that it yields either one feature (sharing) or two (replication).

• *No ambiguity* under repeated inheritance. Conflicting redefinitions under repeated inheritance are resolved through a "selection" mechanism.

• *Unconstrained genericity*. A class can be parameterized, or “generic”, to describe containers of objects of an arbitrary type.

• *Constrained genericity*. A generic class can be declared with a generic constraint, to indicate that the corresponding types must satisfy some properties, such as the presence of a particular operation.

• *Garbage collection*. The dynamic model is designed so that memory reclamation, in a supporting environment, can be automatic rather than programmer-controlled.

•*No-leak modular structure*. All software is built out of classes, with only two inter-class relations, client and inheritance.

• *Once routines*. A feature can be declared as “once”, so that it is executed only for its first call, subsequently returning always the same result (if required). This serves as a convenient initialization mechanism, and for shared objects.

• *Standardized library*. The Kernel Library, providing essential abstractions, is standardized across implementations.

• *Other libraries*. Eiffel development is largely based on high-quality libraries covering many common needs of software development, from general algorithms and data structures to networking and databases.

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

• *No global variables*, which would break the modularity of systems and hamper extendibility, reusability and reliability.

• *No union types* (or record type with variants), which force the explicit enumeration of all variants; in contrast, inheritance is an open mechanism which permits the addition of variants at any time without changing existing code.

• *No in-class overloading* which, by assigning the same name to different features within a single context, causes confusions, errors, and conflicts with object-oriented mechanisms such as dynamic binding. (Dynamic binding itself is a powerful form of inter-class overloading, without any of these dangers.)

• *No goto instructions* or similar control structures (break, exit, multiple-exit loops) which break the simplicity of the control flow and make it harder or impossible to reason about the software (in particular through loop invariants and variants).

• *No exceptions to the type rules*. To be credible, a type system must not allow unchecked "casts" converting from a type to another. (Safe cast-like operations are available through assignment attempt.)

•*No side-effect* expression operators confusing computation and modification.

• *No low-level pointers*, no pointer arithmetic, a well-known source of bugs. (There is however a type POINTER, used for interfacing Eiffel with C and other languages.)

3 THE SOFTWARE PROCESS IN EIFFEL

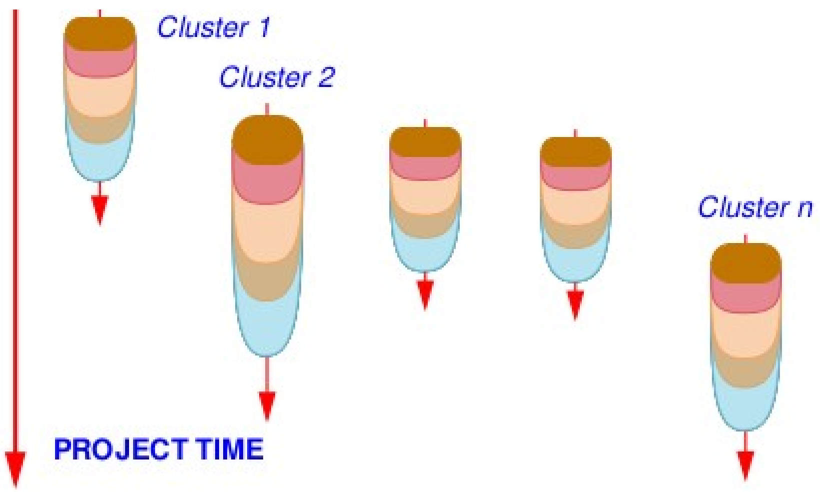
Eiffel, as noted, supports the entire lifecycle. The underlying view of the system development lifecycle is radically different not only from the traditional "Waterfall" model (implying a sequence of discrete steps, such as analysis, global design, detailed design, implementation, separated by major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

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

Clusters and the cluster model

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

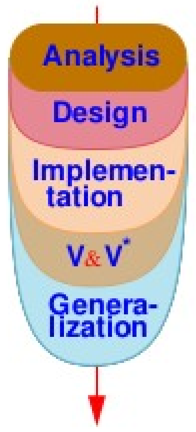
The cluster model: sequential and concurrent engineering



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

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

Individual cluster lifecycle



V&V: Validation and Verification

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

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

Seamlessness and reversibility.

The preceding ideas define the seamless approach embodied by Eiffel. Withseamlessness goes reversibility: the ability to go back, even late in the process, toearlier stages. Because the developers work on a single product, they can takeadvantages of bouts of late wisdom — such as a great idea for adding a new function,discovered only at implementation time — and integrate them in the product.Traditional approaches tend to discourage reversibility because it is difficult toguarantee that the analysis and design will be updated with the late changes. With thesingle-product principle, this is much easier to achieve. Seamlessness and reversibility enhance extendibility by providing a direct mapping from the structure of the solution to the structure of the problem description, making it easier to take care of customers’ change requests quickly and efficiently. They promote reliability, by avoiding possible misunderstandings between customers’ and developers’ views. They are a boost to maintainability. More generally, they yield a smooth, consistent software process that helps both quality and productivity.

Generalization and reuse.

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

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

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

Constant availability.

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

Compilation technology.

The preceding goals benefit from the ability to check frequently that the currentiteration is correct and robust. Eiffel supports efficient compilation mechanismsthrough such mechanisms as the **Melting Ice Technology** in ISE’s EiffelStudio. TheMelting Ice achieves immediate recompilation after a change, guaranteeing arecompilation time that’s a function of the size of the changes, not of the system’soverall size. Even for a system of several thousand classes and several hundredthousand lines, the time to get restarted after a change to a few classes is, on a typicalmodern computer, a few seconds.

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

Quality and functionality.

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

Of course not everyone using Eiffel can, any more than in another approach,guarantee that the ideal just presented will always hold. But it is the theoretical scheme towhich the method tends. It explains Eiffel’s emphasis on getting everything right: the grandiose and the mundane, the structure and the details.

Regarding the details, the Eiffel books cited in the bibliography include many rules, some petty at first sight, about such low-level aspects as the choice of names for classes and features (including their grammatical categories), the indentation of software texts, the style for comments(including the presence or absence of a final period), the use of spaces. Applying these rules does not, of course, guarantee quality; but they are part of a quality-oriented process, along with the more ambitious principles of design. In addition they are particularly important for the construction of quality libraries, one of the central goals of Eiffel.

4 HELLO WORLD

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

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

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

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

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

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

complete second one.

Note the absence of semicolons and other syntactic clatter or clutter. You may in fact use semicolons to separate instructions and declarations. But the language's syntax is designed to make the semicolon optional (regardless of text layout) and it's best for readability to omit it, except in the special case of successive elements on a single line.

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

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

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

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

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

• *put_string* outputs a string, passed as argument, here "Hello World".

• *put_new_line* terminates the line

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

To build the system and execute it:

• Start EiffelStudio

• When prompted, ask EiffelStudio to build a system for you; specify HELLO as the “root class” and make as the “root procedure”.

• You can either use EiffelStudio to type in the above class text, or you may use any text editor and store the result into a file hello.e in the current directory.

• Click the “Compile” icon.

• Click the “Run” icon.

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

5 THE STATIC PICTURE: SYSTEM ORGANIZATION

We now look at the overall organization of Eiffel software.

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

Systems

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

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

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

Classes

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

- In a banking system, a class *ACCOUNT* may have features such as *deposit* , adding a certain amount to an account, *all_deposits* , yielding the list of deposits since the account's opening, and *balance* , yielding the current balance, with properties stating that *deposit* must add an element to the *all_deposits* list and update balance by adding the sum deposited, and that the current value of *balance* must be consistent with the lists of deposits and withdrawals.

- A class *COMMAND* in an interactive system of any kind may have features such as *execute* and *undo* , as well as a feature *undoable* which indicates whether a command can be undone, with the property that *undo* is only applicable if *undoable* yields the value true.

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

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

Class relations

Two relations may exist between classes:

- You can define a class *C* as a **client** of a class *A* to enable the features of *C* to rely on objects of type *A* .

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

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

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

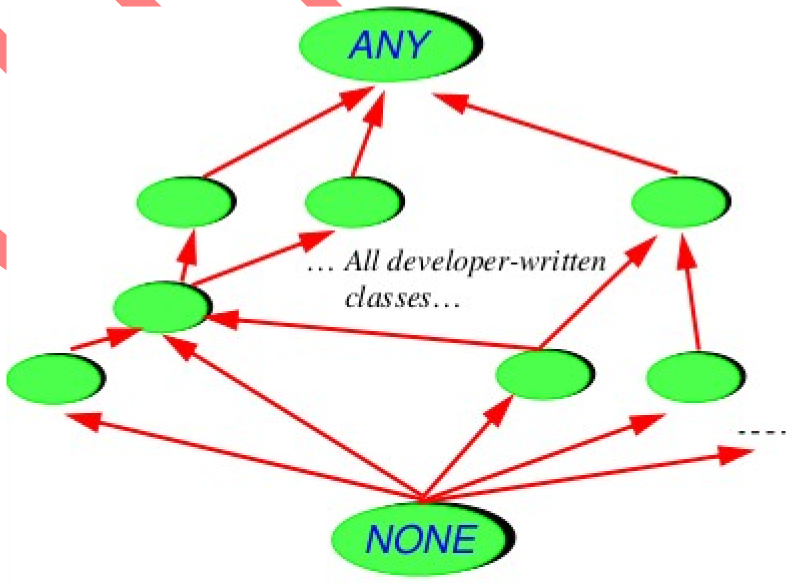
In modeling terms, client roughly represents the relation “has” and heir roughly represents “is”. For example we may use Eiffel classes to model a certain system and express that every child *has* a birth date (client relation) and *is* a person (inheritance).

Distinctive of Eiffel is the rule that classes can only be connected through these two relations. This excludes the behind-the-scenes dependencies often found in other approaches, such as the use of global variables, which jeopardize the modularity of a system. Only through a strict policy of limited and explicit inter-class relations can we achieve the goals of reusability and extendibility.15

The global inheritance structure

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

Global inheritance structure



Any class that does not explicitly inherit from another is considered to inherit from

ANY , so that every class is a descendant, direct or indirect, of *ANY* . *ANY* introduces a number of general-purpose features useful everywhere, such as copying, cloning and equality testing operations and default input-output. The procedure *print* used in the first version of our “Hello World” comes from *ANY* .

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

Clusters

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

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

External software

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

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

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

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

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

6 THE DYNAMIC STRUCTURE: EXECUTION MODEL

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

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

Objects, fields, values and references

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

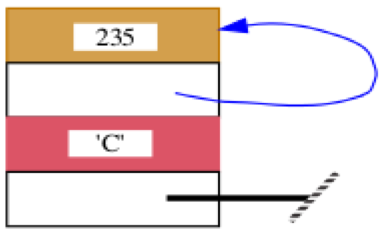
An **object**, as may be created during the execution of a system, is an instance of some class of the system.

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

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

'

Composite object

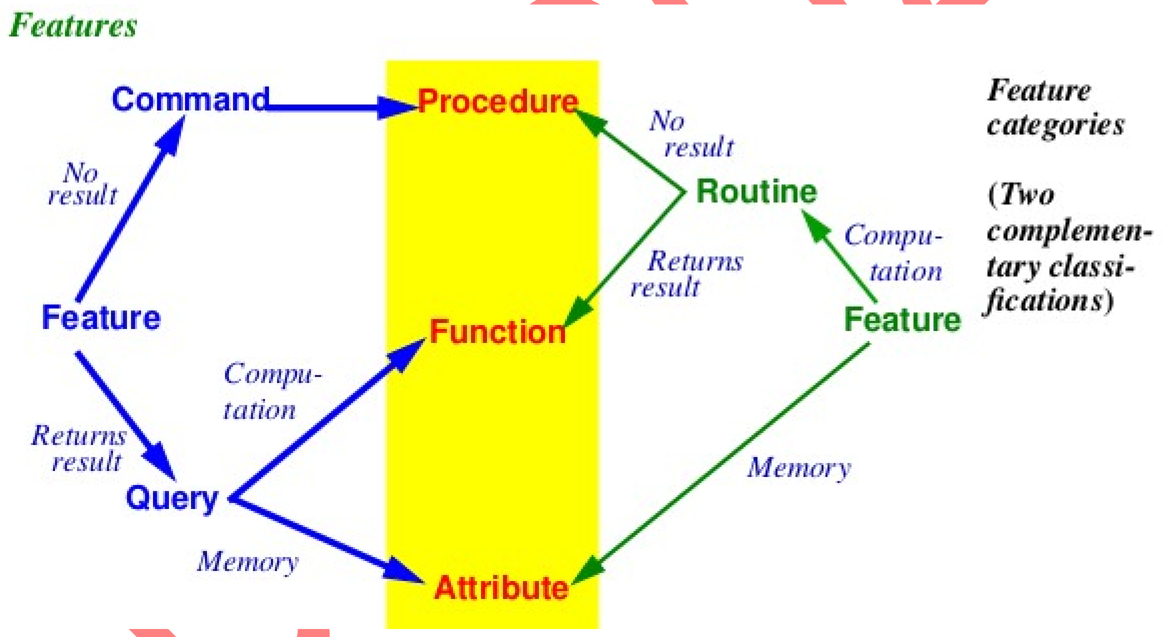


(with 4 fields including self-
reference and void reference)

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

• When a field is an object, it will in most cases be an atomic object, as on the figure where the first field from the top is an integer and the third a character. But a field can also be a composite object, in which case it is called a **subobject**.

• A **reference** is either void or uniquely identifies an object, to which it is said to be **attached**. In the preceding figure the second field from the top is a reference — attached in this case, as represented by the arrow, to the enclosing object itself. The bottom field is a void reference.



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

• An attribute is a feature implemented through memory: it describes a field that will be found in all instances of the class. For example class *ACCOUNT* may have an attribute *balance* ; then all instances of the class will have a corresponding field containing each account's current balance.

• A routine describes a computation applicable to all instances of the class. *ACCOUNT* may have a routine withdraw .

• Routines are further classified into functions, which will return a result, and procedures, which will not. Routine *withdraw* will be a procedure; an example of function may be *highest_deposit* , which returns the highest deposit made so far to the account.

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

• **Commands** have no result, and may modify an object. They may only be procedures.

• **Queries** have a result: they return information about an object. You may implement a query as either an attribute (by reserving space for the corresponding information in each instance of the class, a memory-based solution) or a function (a computation-based solution). An attribute is only possible for a query without argument, such as *balance* ; a query with arguments, such as *balance_on (d)*, returning the balance at date *d* , can only be a function.

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

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