3

The architecture of Eiffel software

3.1 OVERVIEW

The constituents of Eiffel software are called **classes**. To keep your classes and your development organized, it is convenient to group classes into **clusters**. By combining classes from one or more clusters, you may build executable **systems**.

These three concepts provide the basis for structuring Eiffel software:

- A *class* is a modular unit.
- A *cluster* is a logical grouping of classes.
- A *system* results from the assembly of one or more classes to produce an executable unit.

Of these, only "class", describing the basic building blocks, corresponds directly to a construct of the language. To build clusters and systems out of classes, you will use not a language mechanism, but tools of the supporting environment.

Clusters provide an intermediate level between classes and systems, indispensable as soon as your systems grow beyond the trivial:

- At one extreme, a cluster may be a simple group of a few classes.
- At the other end, a system as a whole is simply a cluster that you have made executable (by selecting a *root class* and a *root procedure*).
- In-between, a cluster may be a library consisting of several subclusters, or an existing system that you wish to integrate as a subcluster into a larger system.

Clusters also serve to store and group classes using the facilities of the underlying operating system, such as files, folders and directories.

After the basic definitions, the language description will concentrate on classes, indeed the most important concept in the Eiffel method, which views software construction as an industrial production activity: combining components, not writing one-of-a-kind applications.

The present chapter introduces the overall structure of Eiffel software by discussing in turn the notions of class, system and cluster.

3.2 CLASSES

Classes are not just the modular units of software decomposition: they also serve as a basis for the types of Eiffel.

"Object-Oriented Software Construction" discusses the practical and theoretical roles of classes.

This dual view is essential to understanding the notion of class and, more generally, the principles of object-oriented software construction:

- As a decomposition unit, a class is a module, that is to say a group of related **services** packaged together into a named unit.
- As a type, a class is the description of similar run-time data elements, or *objects*, called the **instances** of the class.

→ Chapter 19 explains the precise nature of objects.

Although these two roles may at first seem rather different, it is in fact useful to support them through a single concept — the class — on the basis of an important observation (the starting point of the theory of *Abstract Data Types*): a good way of describing a set of similar objects without describing their implementation is to list the operations applicable to them. But then if the objects are all instances of the same class, we can define that class, viewed as a module, so that the services it offers are precisely the operations available on the instances of the class, viewed as a type.

This identification of services on modules with operations on instances is what makes it possible to merge the module and type views into the single concept of class. The **features** of a class are these services-operations.

For example, a document processing system could have classes such as *DOCUMENT*, *PARAGRAPH*, *FONT*, *TEXT_DISPLAY*. These are the modular units of the system; their texts can be processed by an Eiffel language processing tool, such as a compiler. They also describe possible run-time objects: documents, paragraphs, fonts, displayable views of text. Systems that include the given classes will be able to create such objects, modify them, and access their properties.

Each of these classes will contain features; for example, *PARAGRAPH* may include features *indent*, describing an operation that indents a paragraph, and *line_count*, to determine the number of lines of a paragraph.

To create an instance of a class, you may use a **creation instruction**; a typical form is

→ Chapter 20.

```
create x.cp (...)
```

where *x* is the name of the entity that will denote the newly created object, and *cp* is one of the features of the class, which must have been designated as a **creation procedure**. This creates an object, makes it accessible through the name *x*, and applies *cp* to initialize it. For example you might create an instance of class *DOCUMENT* through

```
create new_text.make ("Isabelle", 250)
```

assuming *DOCUMENT* has a creation procedure *make* with two arguments: a string for the author's name, an integer for the expected number of pages.

A bit of more precise terminology is useful here. An instance of a class *C* resulting from a creation instruction on a target of the corresponding type is called a **direct instance** of *C*; in the last example, *new_text* will be attached to a direct instance of *DOCUMENT*. The reason for this term is that with the introduction of inheritance we will consider direct instances of *proper descendants* of *C* also as instances (not direct) of *C*.

3.3 CLASS TEXTS AND CLASS NAMES

Every class has a class name, such as *DOCUMENT* or *PARAGRAPH*, and a class text describing the features of the class and its other properties.

→ Chapter 4 presents the structure of class texts.

As you know, letter case is not significant in identifiers, so that you can write a class name as *doCumEnt* if you really want to. But this is strongly discouraged. The standard style is to write all class names using their upper names, such as *DOCUMENT*.

→ Appendix 34 presents style rules.

← The upper name is the name all in upper case. See "TEXTUAL CONVENTIONS", 2.13, page 101.

When you want to display a class in EiffelStudio, you may type its name in any mix of lower and upper case (lower case is usually more convenient); the tools will display the upper name.

The classes of a system must all, as discussed below, have different names.

→ "Class Name rule", page 110.

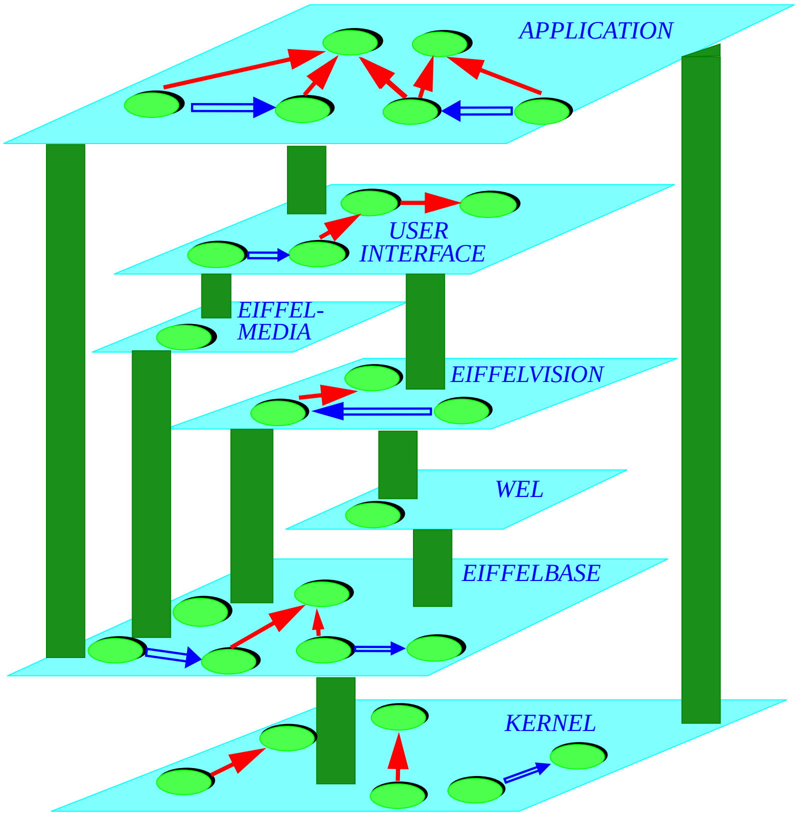
3.4 CLUSTERS

As the number of classes in your systems grows, you will need to arrange these classes into groups, called clusters.

Clusters correspond to the major divisions of a system. For example, a compiling system may include a lexical cluster, a parsing cluster, a semantic analysis cluster, an optimization cluster, a generation cluster. A cluster may encompass a library, such as EiffelBase or EiffelVision; or it may be an application cluster, encompassing a logically significant subset of a system's specific classes.

The figure on the next page illustrates a typical system structure as a set of layers, each representing a cluster. Every cluster of this example except *KERNEL* relies on others through pillars, representing the dependency relations, client and inheritance, between the clusters' classes. The lower clusters, which normally should be built first, provide the basic capabilities; the higher clusters are more specialized, including *APPLICATION* which is assumed to cover the application-specific facilities of the system. In practice, of course, a system may include several application clusters.

The analogy with a physical construction works only to a point; the author and publisher decline any responsibility should you build your house with the architecture shown.



A possible cluster structure

Here some of the clusters are Eiffel Software Libraries.

In each cluster, some classes are shown, with possible inheritance (single arrow) and client (double arrow) links.

You may nest clusters; a cluster included in another is called a **subcluster**. So we may represent a structure of classes and clusters as a tree, as shown at the top of the facing page. With this structure, a system as a whole is a cluster; a library is a cluster; and if you want to embed an existing system (itself having such a nested structure) as a subsystem in a larger system, you'll make it one of its subclusters. Such arbitrary nesting is part of the Eiffel method's support for software reuse and composition:

It is useful to define these notions precisely:

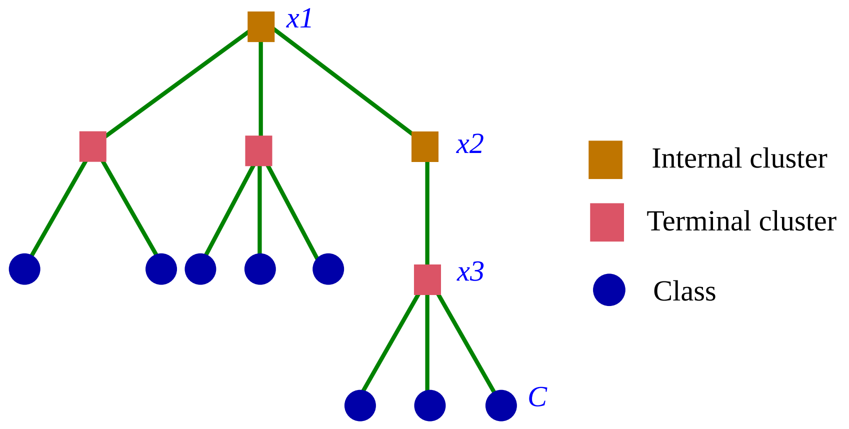
DEFINITION

Cluster, subcluster, contains directly, contains

A **cluster** is a collection of classes, (recursively) other clusters called its **subclusters**, or both. The cluster is said to **contain directly** these classes and subclusters.

A cluster **contains** a class *C* if it contains directly either *C* or a cluster that (recursively) contains *C*.

In the presence of subclusters, several clusters may contain a class, but exactly one contains it directly.



Clusters, subclusters and classes

In the figure clusters *x1*, *x2* and *x3* all contain class *C*; the one that contains *C* directly is *x3*. These observations lead us to define two kinds of cluster:

Terminal cluster, internal cluster

A cluster is **terminal** if it contains directly at least one class.

A cluster is **internal** if it contains at least one subcluster.

From these definitions, it is possible for a cluster to be both terminal and internal.

This is not the recommended style, however; the *methodological advice* is to keep the two cases separate, so that terminal clusters will contain only classes and internal clusters will contain directly only subclusters. This is the case in the example of the last figure.

Beyond this advice, there is no absolute rule on how to group classes into clusters. It is usually wise to observe the following informal criteria:

- The classes in a cluster should be conceptually related.
- In most cases, the number of classes in a terminal cluster should not exceed 20. You should consider splitting a terminal cluster into subclusters if it reaches that size, unless you feel that the classes are strongly connected and the cluster has a "flat" structure with no obvious criterion for splitting it.
- Cycles in the client relation should, in general, only involve classes that all belong to the same terminal cluster, avoiding cases in which *A* is a client of *B* and *B* a client of *A* with *A* and *B* in different clusters.
- For any terminal cluster, there should be at least one person who understands the cluster in its entirety.

→ "Client" is a relation between classes, studied in chapter 7. Cycles in the relation are explicitly permitted; see "SIMPLE CLIENTS", 7.4, page 189.

Do not look, however, for a cluster construct in Eiffel. The highest-level construct is the class; clusters, although essential for organizing Eiffel software and managing its development, do not require language support. This is because such support would in most cases be redundant with the facilities provided by operating systems. If, as may be expected, classes are kept in files, then clusters will use the operating system mechanisms available to support the grouping of related files: folders (the Windows/ Macintosh term) and directories (Unix/Linux). If, as suggested above, you make a clear separation between terminal and internal clusters, then some cluster folders will only contain class files, and the others will only contain subfolders. In the figure on the preceding page, squares then represent folders and the circles represent class files.

Eiffel tools, unlike the Eiffel language, should support clusters. The notion of cluster is also prominent in the Lace control language.

→ Lace is a control language for assembling, compiling and executing Eiffel systems, covered by appendix B.

3.5 SYSTEMS

By themselves, classes are only building blocks. To obtain an executable software element, you must assemble one or more classes into a **system** and designate one of them as the "root". Here are the precise definitions.

We start with a "universe" of classes:

Universe

A **universe** is a set of classes.

The universe provides a reference from which to draw classes of interest for a particular system. Any Eiffel environment will provide a way to specify a universe.

For example, in the EiffelStudio environment, you may define a universe by specifying (through the control language Lace, or through the graphical interface) a set of *directories* (folders), each defining a **cluster**. The cluster is a set of classes; by default, any file in that directory with a name ending with **.e** — for example, *your_class***.e** —, called a **class file**, is expected to contain an Eiffel class. The class texts contained in the class files of the specified clusters then make up the universe.

A strong requirement constrains the names of classes in a universe:

Class Name rule *VSCN*

It is valid for a universe to include a class if and only if no other class of the universe has the same upper name.

Eiffel expressly does not include a notion of "namespace" as present in some other languages. Experience with these mechanisms shows that they suffer from two limitations:

- They only push forward the problem of class name clashes, turning it into a problem of namespace clashes.
- Even more seriously, they tie a class to a particular context, making it impossible to reorganize ("*refactor*") the software later without breaking existing code, and hence defeating some of the principal benefits of object technology and modern software engineering.

Name clashes, in the current Eiffel view, should be handled by *tools* of the development environment, enabling application writers to combine classes from many different sources, some possibly with clashing names, and resolving these clashes automatically (with the possibility of registering user preferences and remembering them from one release of an acquired external set of classes to the next) while maintaining clarity, reusability and extendibility.

The preceding validity rule leads to the rule giving the *meaning* of a class name:

Class name semantics

A Class_name *C* appearing in the text of a class *D* denotes the class called *C* in the enclosing universe.

As usual, the semantic rule only makes sense if the validity rule holds.

A system will be drawn from a universe; to do this we need to designate a particular type and a particular procedure as "roots":

System, root type name, root procedure name

A **system** is defined by the combination of:

1 • A universe.

2 • A type name, called the **root type name**.

3 • A feature name, called the **root procedure name**.

The names that you choose for root type and root procedure should correspond to suitable types and procedures in the system. To state this rule we need a notion of dependency between types:

Type dependency

A type *T* **depends** on a type *R* if any of the following holds:

1 • *R* is a parent of the base class *C* of *T*.

2 • *T* is a client of *R*.

3 • (Recursively) there is a type *S* such that *T* depends on *S* and *S* depends on *R*.

This states that *C* depends on *A* if it is connected to *A* directly or indirectly through some combination of the basic relations between types and classes — inheritance and client — studied later. Case 1 relies on the property that every type derives from a class, called its "base class"; for example a generically derived type such as *LIST [INTEGER]* has base class *LIST*. Case 3 gives us indirect forms of dependency, derived from the other cases.

→ Inheritance: chapter 6; client: chapter 7 (general definition on page 188); base class: chapter 11.

This makes it possible to define what's a proper choice of root type:

Root Type rule *VSRT*

It is valid to designate a type *TN* as root type of a system of universe *U* if and only if it satisfies the following conditions:

1 • *TN* is the name of a stand-alone type *T*.

2 • *T* only involves classes in *U*.

3 • *T*'s base class is not deferred.

4 • The base class of any type on which *T* depends is in *U*.

These conditions make it possible to create the root object:

- A type is "*stand-alone*" if it only involves class names; this excludes "anchored" types (**like** *some_entity*) and formal generic parameters, which only mean something in the context of a particular class text. Clearly, if we want to use a type as root for a system, it must have an absolute meaning, independent of any specific context. "Stand-alone type" is defined at the end of the discussion of types.
- A deferred class is not fully implemented, and so cannot have any direct instances. It wouldn't work as base class here, since the very purpose of a root type is to be instantiated, as the first event of system execution.

→ "DEFERRED FEATURES", 10.11, page 266

- To be able to assemble the system, we must ensure that any class to which the root refers directly or indirectly is also part of the universe.

In condition 2, a type *TN* "involves" a class *C* if it is defined in terms of *C*, meaning that *C* is the base class of *TN* or of any of its generic parameters: *U [V, X [Y, Z]]* involves *U*, *V*, *X*, *Y* and *Z*. A non-generic class *T* used as a type "involves" only itself.

→ "Types and classes involved in a type", page 335.

To complement the conditions on the root type we need one on the root procedure (the procedure that will start the system's execution):

Root Procedure rule *VSRP*

It is valid to specify a name *pn* as root procedure name for a system *S* if and only if it satisfies the following conditions:

1 • *pn* is the name of a creation procedure *p* of *S*'s root type.

2 • *p* has no formal argument.

3 • *p* is precondition-free.

A routine is *precondition-free* (condition 3) if it has no precondition, or a precondition that evaluates to true. A routine can impose preconditions on its callers if these callers are other routines; but it makes no sense to impose a precondition on the external agent (person, hardware device, other program...) that triggers an entire system execution, since there is no way to ascertain that such an agent, beyond the system's control, will observe the precondition. Hence the last condition of the rule.

Regarding condition 1, note that a non-deferred class that doesn't explicitly list any creation procedures is understood to have a single one, procedure *default_create*, which does nothing by default but may be redefined in any class to carry out specific initializations.

→ *"OMITTING THE CREATION PROCEDURE", 20.4, page 519.*

Another condition on the root procedure is that it must be effective (non-deferred): a deferred procedure has no implementation, and hence cannot be used to start system execution. But we don't need such a condition in the Root Procedure rule, because it follows from the Root Class rule: if the root class contained a deferred procedure, it would itself have to be declared as deferred (as a result of a rule to be seen in a later chapter), and we have already precluded that through condition 3 of the Root Class rule.

→ *"Class Header rule", page 126.*

Thanks to the Root Type and Root Procedure rules we no longer have to talk about type and procedure *names*, but can directly refer to the root type, the root procedure and the root class of a system:

Root type, root procedure, root class

In a system *S* of root type name *TN* and root procedure name *pn*, the **root type** is the type of name *TN*, the **root class** is the base class of that root type, and the **root procedure** is the procedure of name *pn* in that class.

Any language processing tool used to assemble and execute systems must enable you to perform the following tasks:

1 • Selecting a root class.

2 • If the root class has two or more creation procedures, selecting one of them — the **root procedure** — for the system's execution.

Techniques to perform these selections fall beyond the scope of Eiffel proper, relying instead on tools of the environment. One possibility is to use **Lace** (Language for Assembling Classes in Eiffel), a simple Eiffel-like notation for specifying how to build and process a system. You may find a detailed description of Lace in an appendix.

→ *Appendix B.*

The root type and root procedure are needed to *execute* the system:

System execution

To **execute** a system on a machine means to cause the machine to apply a creation instruction to the system's root type.

If a routine is a creation procedure of a type used as root of a system, its execution will usually create other objects and call other features on these objects. In other words, the execution of any system is a chain of explosions — creations and calls — each one firing off the next, and the root procedure is the spark that detonates the first step.