A

Eiffel: The Essentials

This appendix addresses people who are familiar with the object-oriented approach but do not know Eiffel very well. It introduces all the concepts needed to understand the core of this thesis.

However, it is not an exhaustive presentation of the Eiffel language. The reference manual of the current version of Eiffel is the book by Meyer *Eiffel: The Language*. The next version of the language is defined in the third edition of this book, which is currently available as a draft. [Meyer 1992]. [Meyer 200?b].

A.1 SETTING UP THE VOCABULARY

First, Eiffel uses vocabulary that sometimes differs from the one used in other object-oriented languages like Java or C#. This section sets up the vocabulary with references to the terminology used in these other languages you may be familiar with.

Structure of an Eiffel program

The basic unit of an Eiffel program is the **class**. There is no notion of module or assembly like in .NET, no notion of package like in Java (no **import**-like keyword).

Classes are grouped into **clusters**, which are often associated with a file directory. Indeed, an Eiffel class is stored in a file (with the extension .e); therefore it is natural to associate a cluster with a directory. But this is not compulsory. It is a logical separation, not necessary a physical one. Clusters may contain subclusters, like a file directory may contain subdirectories.

An Eiffel **system** is a set of classes (typically a set of clusters that contain classes) that can be assembled to produce an executable. It is close to what is usually called a “program”.

Eiffel also introduces a notion of **universe**. It is a superset of the system. It corresponds to all the classes present in the clusters defined in an Eiffel system, even if these classes are not needed for the program execution.

Eiffel uses a notion of **root class**, which is the class that is instantiated first using its creation procedure (the constructor) known as the **root creation procedure**. An Eiffel system corresponds to the classes needed by the root class directly or indirectly (the classes that are reachable from the root creation procedure). The universe contains all classes in all the clusters specified in the system.

The definition of what an Eiffel system contains is done in an **Ace file**, which is a configuration file written in a language called LACE (*Language for Assembly Classes in Eiffel*).

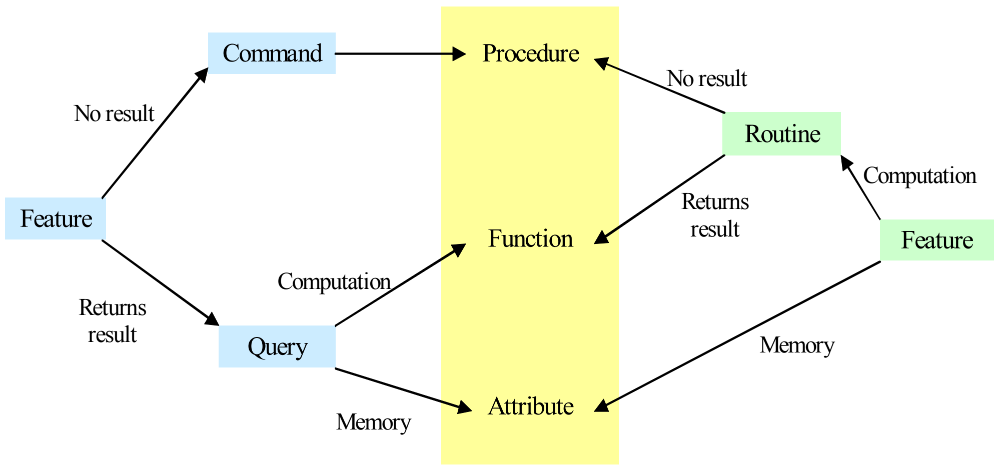
Classes

A class is a representation of an Abstract Data Type (ADT). Every **object** is an instance of a class. The object creation uses a so-called **creation procedure**, which is similar to the notion of "constructor" in languages such as Java or C#.

A class is characterized by a set of **features** (operations), which may be either **attributes** or **routines**. (In Java/C# terminology, features tend to be called "members", attributes are called "fields" and routines are called "methods".) Eiffel further distinguishes between routines that return a result (**functions**) and routines that do not return a result (**procedures**). This is a classification by implementation: routines vs. attributes, namely computation vs. memory.

There is another classification: by role. Features can be either **commands** (if they do not return a result) or **queries** (if they do return a result). Then, queries can be either functions if they involve some computation or attributes if the value is stored in memory.

The following picture shows the different feature categories:



Feature categories (by role and by implementation)

Design principles

Eiffel is not only a programming language but also an object-oriented method to build high-quality software. As a method, it brings some design principles:

- As mentioned above, Eiffel distinguishes between "commands" and "queries". Even if not enforced by any compiler, the Eiffel method strongly encourages following the **Command/Query Separation principle**: A feature should not both change the object's state and return a result about this object. In other words, a function should be side-effect-free. As Meyer likes to present it: "*Asking a question should not change the answer.*" [Meyer 1997], *p 751.*
- Another important principle, which is **Information Hiding**: The supplier of a module (typically a class) must select the subset of the module's properties that will be available officially to its client (the "public part"); the remaining properties build the "secret part". The Eiffel language provides the ability to enforce this principle by allowing to define fine-grained levels of availability of a class to its clients. [Meyer 1997], *p 51-53.*
- Another principle enforced by the Eiffel method and language is the principle of **Uniform Access**, which says that all features offered by a class should be available through a uniform notation, which does not betray whether features are implemented through storage (attributes) or through computation (routines). Indeed, in Eiffel, one cannot know when writing *x*.*f* whether *f* is a routine or an attribute; the syntax is the same. [Meyer 1997], *p 57.*

Types

As mentioned earlier, in Eiffel, every object is an instance of a class. There is no exception; even basic types like *INTEGER*s or *REAL*s are represented as a class.

Besides, Eiffel is strongly typed; every program entity is declared of a certain **type**. A type is based on a class. In case of non-generic classes, type and class are the same. In the case of generic classes, a class is the basis for many different types.

See "Genericity", page 387.

The majority of types are **reference types**, which means that values of a certain type are references to an object, not the object itself. There is a second category of types, called **expanded types**, where values are actual objects. It is the case of basic types in particular. For example, the value *5* of type *INTEGER* is really an object of type *INTEGER* with value *5*, not a pointer to an object of type *INTEGER* with a field containing the value *5*.

A.2 THE BASICS OF EIFFEL BY EXAMPLE

This section shows you what an Eiffel class looks like with an example.

Structure of a class

The basic structure of an Eiffel class is the following:

```
class

        CLASS_NAME

feature -- Comment

        ...

feature -- Comment

        ...

end
```

Basic structure of an Eiffel class

It starts with the keyword **class** and finishes with the keyword **end**, and in-between a set of features grouped by "feature clauses" introduced by the keyword **feature** and a comment. The comment is introduced by two consecutive dashes and is not compulsory (although recommended to improved readability and understandability).

Book example

An Eiffel class may contain other clauses than the basic ones just shown. For example, it may start with an "indexing clause" (introduced by the keyword **indexing**), which should gives general information about the class.

The following class *BOOK* is a typical example of what an Eiffel class looks like. (If you do not understand everything, don't panic; each new notion will be described after the class text.)

```
indexing

        description: "Representation of a book"

class

        BOOK

create

        make
```

Class representation of a book in a library

```
feature {NONE} -- Initialization

        make (a_title: like title; some_authors: like authors) is
                        -- Set title to a_title and authors to some_authors.
                require
                        a_title_not_void: a_title /= Void
                        a_title_not_empty: not a_title.is_empty
                do
                        title := a_title
                        authors := some_authors
                ensure
                        title_set: title = a_title
                        authors_set: authors = some_authors
                end

feature -- Access

        title: STRING
                        -- Title of the book

        authors: STRING
                        -- Authors of the book
                        -- (if several authors, of the form:
                        -- "first_author, second_author, ...")

feature -- Status report

        is_borrowed: BOOLEAN
                        -- Is book currently borrowed (i.e. not in the library)?

feature -- Basic operation

        borrow is
                        -- Borrow book.
                require
                        not_borrowed: not is_borrowed
                do
                        is_borrowed := True
                ensure
                        is_borrowed: is_borrowed
                end

        return is
                        -- Return book.
                require
                        is_borrowed: is_borrowed
                do
                        is_borrowed := False
                ensure
                        not_borrowed: not is_borrowed
                end

invariant

        title_not_void: title /= Void
        title_not_empty: not title.is_empty

end
```

Let's have a closer look at this class *BOOK*:

- The optional **indexing clause** was mentioned before. Each entry (there may be several) has two parts: a tag "description" and the text "Representation of a book". The tag name "definition" is not a keyword; you can use any name, although it is quite traditional to use "definition" to describe the general purpose of the class. Other commonly used tags include "author", "date", "note", etc. The indexing clause is interesting both for the programmers, the people who will use the class, and for tools which may use this indexed information to do different kinds of things with the class.
- After the **class** keyword and class name, *BOOK*, we find a clause introduced by the keyword **create**. It introduces the name of each creation procedure (constructor) of the class. Indeed, in Eiffel (contrary to Java or C#), creation procedures do not have a predefined name; they can have any name, although "make" is commonly used. Here, the class declares only one creation procedure called *make*. There may be several, in which case the names would be separated by commas. There may also be no **create** clause at all, which means that the class has the default creation procedure called *default_create*. (The feature *default_create* is defined in class *ANY* from which any Eiffel class inherits. This appendix will come back to this point later when mentioning inheritance.)
- The class name, *NONE*, in curly brackets between the keyword **feature** and the comment "-- Initialization" is an application of information hiding. It means that the features listed in this feature clause are exported to *NONE*. *NONE* is a virtual class that inherits from all classes (it is at the bottom of the class hierarchy). A feature exported to *NONE* means that no client class can access it. It can only be used within the class or one of its descendants. (It is close to "protected" in languages such as Java or C#.) It is possible to have any class name between these curly brackets, providing a fine-grained exportation mechanism.

  You may wonder why a creation procedure is exported to *NONE*. It does not mean that the class cannot be instantiated because clients cannot access the creation procedure. Not at all. In fact, in Eiffel, creation procedures are not special features. They are normal features that can be called as creation procedure in expressions of the form **create** *my_book*.*make* but also as "normal" features in instructions of the form *my_book*.*make* (where *my_book* must already be instantiated) to reinitialize the object for example. Exporting a creation routine to *NONE* means that it can only be used as a creation procedure; it cannot be called by clients later on as a normal procedure.

- The basic structure of an Eiffel routine is the following:

```
routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
                -- Comment
        do
                ... Implementation here (set of instructions)
        end
```

Basic structure of an Eiffel routine

  It may also have a **require** clause after the comment to introduce preconditions and an **ensure** clause before the **end** keyword to express postconditions. It is the case of the procedure *make*. This section will not say more about preconditions and postconditions for the moment. They will be covered in detail in the next section about Design by Contract™.

  See "Design by Contract™", page 378.

  A routine may also have a **local** clause, listing the local variables used in this routine; it is located before the **do** keyword (after the **require** clause if any).

If you compare the basic scheme of a routine shown above and the text of *make* on the previous page, you can deduce that the type of the first argument *a_title* is **like** *title* and the type of the argument *some_authors* is **like** *authors*. What does this mean? It is called anchored types in Eiffel. In **like** *title*, *title* is the anchor. It means that the argument *a_title* has the same type as the attribute *title* defined below in the class text. It avoids having to redefine several routines when the type of an attribute changes for example. It will become clearer when we talk about inheritance. But just as a glimpse: imagine *BOOK* has a descendant class called *DICTIONARY* and *DICTIONARY* redefines *title* to be of type *TITLE* instead of *STRING*, then *make* also needs to be redefined to take an argument of type *TITLE*. Anchored types avoid this "redefinition avalanche" by "anchoring" the type of a certain entity to the type of another query (function or attribute). The anchor can also be **Current** (a reference to the current object, like "this" in C# or Java).

The text of feature *make* also shows the syntax for assignment := (not = like in Java and C#; in Eiffel = is the reference equality, like == in Java and C#). You may also encounter syntax of the form ?= which corresponds to an assignment attempt. The semantics of an assignment attempt *a* ?= *b* is to check that the type *B* of *b* conforms to the type *A* of *a*; then, if *B* conforms to *A*, *b* is assigned to *a* like a normal assignment; if not, *a* is set to ***Void***. Therefore, the typical scheme of assignment attempts is as follows:

Typical use of assignment attempts

```
a: A
b: B

a ?= b
if a /= Void then
        ...
end
```

The typical use of assignment attempts is in conjunction with persistence mechanisms because one cannot be sure of the exact type of the persistent data being retrieved.

- The next two feature clauses "Access" and "Status report" introduce three attributes: *title* and *authors* of type *STRING*, and *is_borrowed* of type *BOOLEAN*. The general scheme for an attribute is the following:

Structure of an Eiffel attribute

```
attribute_name: ATTRIBUTE_TYPE
                -- Comment
```

In the current version of Eiffel, attributes cannot have preconditions or postconditions. It will be possible in the next version of the language.

See "Assertions on attributes", page 391.

- The routines *borrow* and *return* follow the same scheme as the feature *make* described before. It is worth mentioning though the two possible values for *BOOLEAN*s, namely **True** and **False**, which are both keywords.
- The last part of the class is the invariant, which has two clauses in this particular example. Contracts (preconditions, postconditions, class invariants) are explained in the next section about Design by Contract™.
- One last comment about the class *BOOK*: the use of ***Void***. ***Void*** is a feature of type *NONE* defined in class *ANY*. It is the equivalent of "null" in other languages like C# or Java. It corresponds to a reference attached to no object.

Design by Contract™

Design by Contract™ is a method of software construction, which suggests building software systems that will cooperate on the basis of precisely defined contracts.

The method

Design by Contract™ is a method to reason about software that accompanies the programmer at any step of the software development. Even if it is called "design" by contract, it does not only target the design stage of an application. It is useful as a method of analysis and design, but it also helps during implementation because the software specification will have been clearly stated using "assertions" (boolean expressions). Design by Contract™ is also useful to debug and test the software against this specification.

The idea of Design by Contract™ is to make the goal of a particular piece of software explicit. Indeed, when developers start a new project and build a new application, it is to satisfy the need of a client, match a certain specification. The Design by Contract™ method suggests writing this specification down to serve as a "contract" between the clients (the users) and the suppliers (the programmers).

This idea of **contract** defined by some obligations and benefits is an analogy with the notion of contract in business: the supplier has some obligations to his clients and the clients also have some obligations to their supplier. What is an obligation for the supplier is a benefit for the client, and conversely.

Different kinds of contracts

There are three main categories of contracts: preconditions, postconditions, and class invariants:

- **Preconditions** are conditions under which a routine will execute properly; they have to be satisfied by the client when calling the routine. They are an obligation for the clients and a benefit for the supplier (which can rely on them). A precondition violation is the manifestation of a bug in the client (which fails to satisfy the precondition).

  Precondition clauses are introduced by the keyword **require** in an Eiffel routine:

  ```
          routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
                  -- Comment
              require
                  tag_1: boolean_expression_1
                  tag_2: boolean_expression_2
              do
                  ... Implementation here (set of instructions)
              end
  ```

  Structure of an Eiffel routine with precondition

  Each precondition clause is of the form "tag: expression" where the tag can be any identifier and the expression is a boolean expression (the actual assertion). The tag is optional; but it is very useful for documentation and debugging purposes.

- **Postconditions** are properties that are satisfied at the end of the routine execution. They are benefits for the clients and obligations for the supplier. A postcondition violation is the manifestation of a bug in the supplier (which fails to satisfy what it guarantees to its clients).

Postcondition clauses are introduced by the keyword **ensure** in an Eiffel routine:

```
        routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
                        -- Comment
                do
                        ... Implementation here (set of instructions)
                ensure
                        tag_1: boolean_expression_1
                        tag_2: boolean_expression_2
                end
```

Structure of an Eiffel routine with postcondition

Of course, a routine may have both preconditions and postconditions; hence both a **require** and an **ensure** clause (like in the previous *BOOK* class).

- **Class invariants** capture global properties of the class. They are consistency constraints applicable to all instances of a class. They must be satisfied after the creation of a new instance and preserved by all the routines of the class. More precisely, it must be satisfied after the execution of any feature by any client. (This rule applies to **qualified calls** of the form *x*.*f* only, namely client calls. Implementation calls — **unqualified calls** — and calls to non-exported features do not have to preserve the class invariant.)

  Class invariants are introduced by the keyword **invariant** in an Eiffel class

```
class
        CLASS_NAME

feature -- Comment

        ...

invariant

        tag_1: boolean_expression_1
        tag_2: boolean_expression_2

end
```

Structure of an Eiffel class with class invariant

There are three other kinds of assertions:

- **Check instructions**: Expressions ensuring that a certain property is satisfied at a specific point of a method’s execution. They help document a piece of software and make it more readable for future implementers.

  In Eiffel, check instructions are introduced by the keyword **check** as follows:

```
        routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
                        -- Comment
                do
                        ... Implementation here (set of instructions)
                        check
                                tag_1: boolean_expression_1
                                tag_2: boolean_expression_2
                        end
                        ... Implementation here (set of instructions)
                end
```

Structure of an Eiffel routine with check instruction

- **Loop invariants**: Conditions, which have to be satisfied at each loop iteration and when exiting the loop. They help guarantee that a loop is correct.
- **Loop variants**: Integer expressions ensuring that a loop is finite. It decreases by one at each loop iteration and has to remain positive.

This appendix will show the syntax of loop variants and invariants later when introducing the syntax of loops.

See “Syntax of loops”, page 383.

Benefits

The benefits of Design by Contract™ are both technical and managerial. Among other benefits we find:

- *Software correctness*: Contracts help build software right in the first place (as opposed to the more common approach of trying to debug software into correctness). This first use of contracts is purely methodological: the Design by Contract™ method advises you to think about each routine's requirements and write them as part of your software text. This is only a method, some guidelines for software developers, but it is also perhaps the main benefit of contracts, because it helps you design and implement correct software right away.
- *Documentation*: Contracts serve as a basis for documentation: the documentation is automatically generated from the contracts, which means that it will always be up-to-date, correct and precise, exactly what the clients need to know about.
- *Debugging and testing*: Contracts make it much easier to detect "bugs" in a piece of software, since the program execution just stops at the mistaken points (faults will occur closer to the source of error). It becomes even more obvious with assertions tags (i.e. identifiers before the assertion text itself). Contracts are also of interest for testing because they can serve as a basis for black-box test case generation.
- *Management*: Contracts help understand the global purpose of a program without having to go into the code in depth, which is especially appreciable when you need to explain your work to less-technical persons. It provides a common vocabulary and facilitates communication. Besides, it provides a solid specification that facilitates reuse and component-based development, which is of interest for both managers and developers.

A.3 MORE ADVANCED EIFFEL MECHANISMS

Let's describe more advanced Eiffel mechanisms, typically the facilities on which the pattern library relies on.

See "Pattern Library", page 26.

Book library example

This section uses an example to introduce these mechanisms. Because we talked about books in the previous section, here is the example of a library where users can borrow and return books.

Here is a possible implementation of an Eiffel class *LIBRARY*:

Class representation of a book library

```
indexing

	description: "Library where users can borrow books"

class

	LIBRARY

inherit
	ANY
		redefine
			default_create
		end
```

```
feature {NONE} -- Initialization

        default_create is
                        -- Create books.
                do
                        create books.make
                end

feature -- Access

        books: LINKED_LIST [BOOK]
                        -- Books available in the library

feature -- Element change

        extend (a_book: BOOK) is
                        -- Extend books with a_book.
                require
                        a_book_not_void: a_book /= Void
                        a_book_not_in_library: not books.has (a_book)
                do
                        books.extend (a_book)
                ensure
                        one_more: books.count = old book.count + 1
                        book_added: books.last = a_book
                end

        remove (a_book: BOOK) is
                        -- Remove a_book from books.
                require
                        a_book_not_void: a_book /= Void
                        book_in_library: books.has (a_book)
                do
                        books.start
                        books.search (a_book)
                        books.remove
                ensure
                        one_less: books.count = old books.count – 1
                        book_not_in_library: not books.has (a_book)
                end

feature -- Output

        display_books is
                        -- Display title of all books available in the library.
                do
                        if books.is_empty then
                                io.put_string ("No book available at the moment")
                        else
                                from books.start until books.after loop
                                        io.put_string (books.item.title)
                                        books.forth
                                end
                        end
                end

feature -- Basic operation
```

```
        borrow_all is
                    -- Borrow all books available in the library.
                do
                    from books.start until books.after loop
                        books.item.borrow
                        books.forth
                    end
                ensure
                    all_borrowed: books.for_all (agent {BOOK}.is_borrowed)
                end

invariant

        books_not_void: books /= Void
        no_void_book: not books.has (Void)

end
```

This example introduces two controls we had not encountered before: **conditional structures** (in feature *display_books*) and **loops** (in *display_books* and *borrow_all*).

- Here is the syntax scheme for conditional structures:

Syntax of conditional structures

```
        if some_condition_1 then
            do_something_1
        elseif some_condition_2 then
            do_something_2
        else
            do_something_else
        end
```

The **elseif** and **else** branches are optional.

- Here is the syntax scheme for loops (there is only one kind of loops in Eiffel):

Syntax of loops

```
        from
            initialization_instructions
        invariant
            loop_invariant
        variant
            loop_variant
        until
            exit_condition
        loop
            loop_instructions
        end
```

The **variant** and **invariant** clauses are optional. The **from** clause is compulsory but it may be empty.

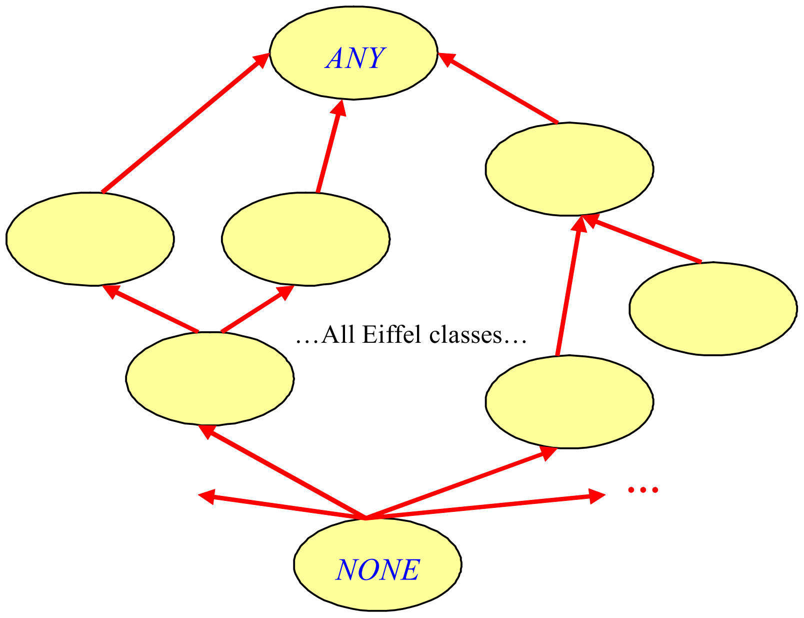
Let's now discover the other Eiffel techniques used in this example:

Inheritance

The class *LIBRARY* contains a clause we have not seen yet: an **inherit** clause. It introduces the classes from which class *LIBRARY* inherits. Here *LIBRARY* inherits from *ANY*.

ANY is the class from which any Eiffel class inherits. If you remember, we saw before that *NONE* is the class that inherits from any Eiffel class, which means we have a “closed” hierarchy here:

See “Business Object Notation (BON)”, page 394.



Global inheritance structure

In fact, one does not need to write that a class inherits from *ANY*; it is the default. Here, the class *LIBRARY* expresses the inheritance link explicitly to be able to “redefine” the feature *default_create* inherited from *ANY*. Redefinition allows changing the body of a routine (the **do** clause) and changing the routine signature if the new signature “conforms” to the parent one (which basically means that the base class of the new argument types inherits from the base class of the argument types in the parent, and same thing for the result type if it is a function).

Redefinition of return type and argument types follows the rules of covariance.

See “Non-conforming inheritance”, page 391.

The routine body can be changed, but it still has to follow the routine’s contract. The Design by Contract™ method specifies precise rules regarding contracts and inheritance: preconditions are “or-ed” and can only be weakened, postconditions are “and-ed” and can only be strengthened (not to give clients any bad surprise). Class invariants are also “and-ed” in descendant classes (subclasses).

See “Design by Contract™”, page 378.

As suggested by the inheritance figure on the previous page, a class may inherit from one or several other classes, in which case we talk about **multiple inheritance**. Contrary to languages like C# or Java, Eiffel supports multiple inheritance of classes. (It is restricted to “interfaces” in the Java/C# worlds.)

Allowing multiple inheritance means that a class may get features from two different parents (superclasses) with the same name. Eiffel provides a renaming mechanisms to handle name clashes. For example, if we have a class *C* that inherits from *A* and *B*, and *A* and *B* both define a feature *f*, it is possible to rename the feature *f* from *A* as *g* to solve the name clash. Here is the Eiffel syntax:

```
class
        C
inherit
        A
                rename
                        f as g
                end
        B
feature -- Comment
        ...
end
```

Renaming mechanism

As mentioned above, Eiffel also provides the ability to redefine features inherited from a parent using a **redefine** clause. In the redefined version, it is possible to call the version from the parent by using the **Precursor**. Here is an example:

Redefinition with call to Precursor

```
class
        C
inherit
        A
                redefine
                        f
                end
feature -- Comment
        f (args: SOME_TYPE) is
                        -- Comment
                do
                        Precursor {A} (args)
                        ...
                end
        ...
end
```

Eiffel also enables to "undefine" a routine, that is to say making it deferred (abstract). A deferred feature is a feature that is not implemented yet. It has no **do** clause but a **deferred** clause. Yet it can have routine preconditions and postconditions. Here is the general structure of a deferred routine:

Structure of a deferred routine

```
        routine_name (arg_1: TYPE_1; arg_2: TYPE_2): RETURN_TYPE is
                        -- Comment
                require
                        ... Some precondition clauses
                deferred
                ensure
                        ... Some postcondition clauses
                end
```

A class that has at least one deferred feature must be declared as deferred. In the current version of Eiffel, it is also true that a deferred class must have at least one deferred feature. In the next version of the language, it will be possible to declare a class deferred even if all its features are effective (implemented). Besides, contrary to other languages like Java or C#, a deferred class can declare attributes. It can also express a class invariant.

To come back to inheritance and undefinition, here is the syntax that allows to make a routine deferred when inheriting it from a parent class:

Undefinition mechanism

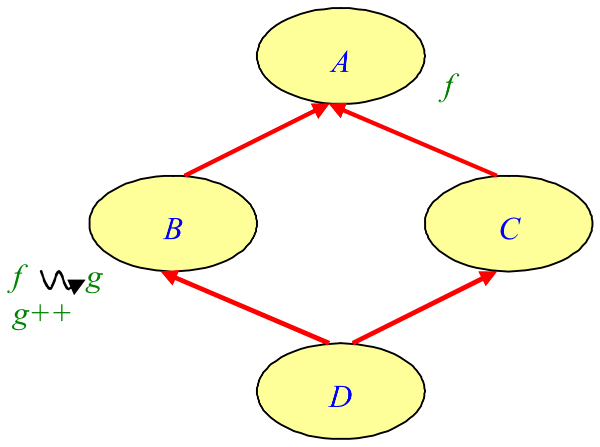
```
deferred class
        C
inherit
        A
                undefine
                        f
                end
feature -- Comment
        ...
end
```

This example supposes that class *A* declares a feature *f*, which is effective in *A*. The class *C* inherits *f* from *A* and makes it deferred by listing it in the clause **undefine**. Therefore, class *C* must be declared as deferred (it has at least one deferred feature).

Another problem that arises with multiple inheritance is the case of repeated inheritance (also known as the "diamond structure"). For example, we have a class *D* that inherits from *B* and *C*, which themselves inherit from *A*. Class *A* has a feature *f*. *B* renames it as *g* and redefines it. *C* leaves *f* unchanged. Here is the corresponding class diagram:

See "Business Object Notation (BON)", page 394.

Repeated inheritance ("diamond structure")



The class *D* inherits two different features from the same feature *f* in *A*. The problem occurs when talking about dynamic binding: which feature should be applied?

There is another inheritance adaptation clause called **select**, which provides the ability to say: "in case there is a conflict, use the version coming from this parent". For example, if we want to retain the feature *g* coming from *B*, we would write:

Selection mechanism

```
class
        D
inherit
        B
                select
                        g
                end
        C
feature -- Comment
        ...
end
```

The last inheritance adaptation clause is the **export** clause. It gives the possibility to restrict or enlarge the exportation status of an inherited routine. Here is the general scheme:

Export mechanism

```
class
        B
inherit
        A
                export
                        {NONE} f -- Makes f secret in B (it may have been exported in A)
                        {ANY} g -- Makes g exported in B (it may have been secret in A)
                        {D, E} x, z -- Makes x and z exported to only classes D and E
                end
feature -- Comment
        ...
end
```

Eiffel offers a keyword **all** to designate all the features of a class (defined in the class itself and inherited). It is often used to make all features inherited from a parent class secret (in case of implementation inheritance for example):

```
class
        B
inherit
        A
                export
                        {NONE} all
                end
feature -- Comment
        ...
end
```

Making all inherited features secret

The last point we need to see about inheritance is the order of adaptation clauses. Here it is:

```
class
        D
inherit
        B
                rename
                        e as k
                export
                        {NONE} all
                        {D} r
                undefine
                        m
                redefine
                        b
                select
                        g
                end
        C
feature -- Comment
        ...
end
```

Order of the inheritance adaptation clauses

Genericity

Let's come back to our class *LIBRARY*. The next mechanism we had not encountered before is genericity. Indeed, the class *LIBRARY* declares a list of *books*:

See "Class representation of a book library", page 381.

```
        books: LINKED_LIST [BOOK]
                        -- Books available in the library
```

Attribute of a generic type

The attribute *books* is of type *LINKED_LIST* [*BOOK*], which is derived from the generic class *LINKED_LIST* [*G*], where *G* is the **formal generic parameter**. A generic class describes a type "template". One must provide a type, called **actual generic parameter** (for example here *BOOK*), to derive a directly usable type like *LINKED_LIST* [*BOOK*]. Genericity is crucial for software reusability and extendibility. Besides, most componentized versions of design patterns rely on genericity.

The example of *LINKED_LIST* [*G*] is a case of **unconstrained genericity**. There are cases where it is needed to impose a constraint on the actual generic parameters. Let's take an example. Say we want to represent vectors as a generic class *VECTOR* [*G*], which has a feature *plus* to be able to add two vectors, and we also want to be able to have vectors of vectors like *VECTOR* [*VECTOR* [*INTEGER*]].

Let’s try to write the feature *plus* of class *VECTOR* [*G*]. In fact, it is unlikely to be called *plus*; it would rather be an **infix** feature “+”, which is a special notation to allow writing *vector_a* + *vector_b* instead of *vector_a*•*plus* (*vector_b*). There also exists a **prefix** notation to be able to write - *my_integer* for example.

To come back to class *VECTOR* [*G*], a first sketch may look as follows:

Addable vectors

```
class

        VECTOR [G]

create

        make

feature {NONE} -- Initialization

        make (max_index: INTEGER) is
                        -- Initialize vector as an array with indexes from 1 to max_index.
                require
                        ...
                do
                        ...
                end

feature -- Access

        count: INTEGER
                        -- Number of elements in vector

        item (i: INTEGER): G is
                        -- Vector element at index i
                require
                        ...
                do
                        ...
                end

        infix "+" (other: like Current): like Current is
                        -- Sum with other
                require
                        other_not_void: other /= Void
                        consistent: other•count = count
                local
                        i: INTEGER
                do
                        create Result•make (1, count)
                        from i := 1 until i > count loop
                                Result•put (item (i) + other•item (i), i)
                                        -- Requires an operation "+" on elements of type G.
                                i := i + 1
                        end
                ensure
                        sum_not_void: Result /= Void
                end
...
invariant
        ...
end
```

Our implementation of the "+" operation requires a "+" operation on elements of type *G*, which means that we cannot accept any kind of actual generic parameters. We need them to have such a feature "+". Here is when constrained genericity comes into play: it allows to say that actual generic parameters must conform to a certain type, for example here *NUMERIC*. (It basically means that the base class of the actual generic parameter must inherit from class *NUMERIC*.)

See "Non-conforming inheritance", page 391.

Here is the corresponding syntax:

```
class
    VECTOR [G –> NUMERIC]
```

It is not allowed to have multiple constraints, say **class** *C* [*G* –> {*A*, *B*}]. It may be allowed in the next version of the language.

[Meyer 200?b].

Another kind of constraint is to impose that actual generic parameters must have certain creation procedures. For example, the notation:

```
class
    MY_CLASS [G –> ANY create default_create end]
```

means that any actual generic parameter of *MY_CLASS* must conform to *ANY* and expose *default_create* in its list of creation procedures (introduced by the keyword **create** in an Eiffel class text).

Agents

There is still one mysterious point in the class *LIBRARY*, the postcondition of *borrow_all*:

See "Class representation of a book library", page 381.

Use of agents in contracts

```
    ensure
        all_borrowed: books.for_all (agent {BOOK}.is_borrowed)
```

What the postcondition of *borrow_all* does is to test for all items of type *BOOK* in the list *books* whether it *is_borrowed*. The postcondition will evaluate to **True** if all books are borrowed.

But what does this "agent" mean? An agent is an encapsulation of a routine ready to be called. (To make things simple, you may consider an agent as a typed function pointer.) It is used to handle event-driven development. For example, if you want to associate an action with the event "button is selected", you will write:

Events with agents

```
    my_button.select_actions.extend (agent my_routine)
```

where *my_routine* is a routine of the class where this line appears.

A typical agent expression is of the form:

Open and closed arguments

```
    agent my_function (?, a, b)
```

where *a* and *b* are **closed** arguments (they are set at the time of the agent's definition) whereas ? is an **open** argument (it will be set at the time of any call to the agent).

It is also possible to construct an agent with a routine that is not declared in the class itself. The syntax becomes:

Open and closed arguments

```
    agent some_object.some_routine (?, a, b)
```

where *some_object* is the **target** of the call. It is a **closed target**. The agent expression used in the postcondition of *borrow_all* had an open target of type *BOOK*:

Open target

```
    agent {BOOK}.is_borrowed
```

To call an agent, you simply have to write:

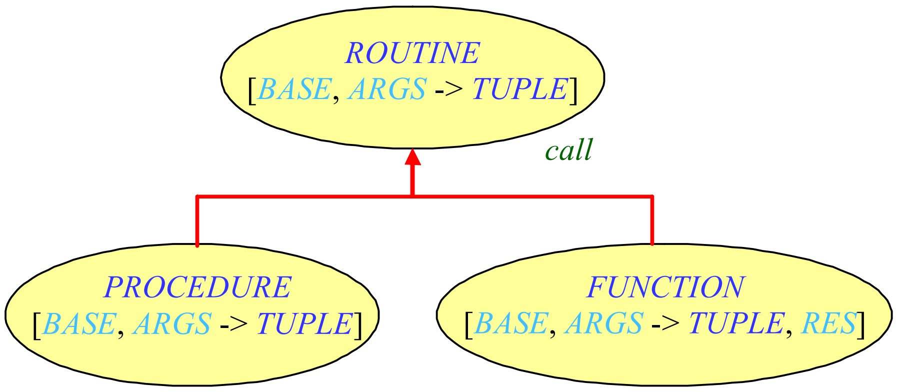
```
my_agent.call ([maybe_some_arguments])
```

Calling an agent

where *call* is a feature of class *ROUTINE*.

Here is the class diagram of the three agent types:

See "Business Object Notation (BON)", page 394.



Agent types

An important property of the Eiffel agent mechanism: it is completely type-safe.

Agents are a recent addition to the Eiffel language. They were introduced in 1999. Therefore, they are not described in the reference manual *Eiffel: The Language*. However, there is an entire chapter of the next revision of the book devoted to agents.

[Dubois 1999].

See chapter 25 of [Meyer 200?b].

Performance

Using agents has a performance overhead. To measure that overhead, I performed one million direct calls to a routine that does nothing and one million agent calls to the same routine. Without agents, the duration was two seconds (2µs per call); with agents, it was fourteen seconds (14µs per call), thus seven times as slow.

But in practice, one calls routines that do something. Therefore I added an implementation to the routine (a loop that executes *do_nothing*, twenty times) and did the same tests. The results were the following: 33s (33µs per call) without agents; 46s (46µs per call) with agents; thus 1.4 times as slow.

do_nothing is a procedure defined in ANY which does nothing.

In a real application, the number of agent calls in the whole code will be less significant. Typically, no more than 5% of the feature calls will be calls to agents. Therefore the execution of an application using agents will be about 0.07 times as slow, which is a acceptable performance overhead in most cases.

A.4 TOWARDS AN EIFFEL STANDARD

Eiffel is currently being standardized through the ECMA international organization. The working group in charge of the Eiffel standardization examines some issues of the Eiffel language and discusses possible extensions. I am an active member of the group; I am responsible for preparing the meetings' agenda and for writing the meeting minutes.

[ECMA-Web].

ECMA standardization

The process started in March 2002 when ECMA accepted the proposal to standardize Eiffel. It resulted in the creation of a new working group TG4, part of the Technical Committee 39 (originally "scripting languages", although this is just for historical reasons). The first group meeting took place at ETH Zurich in Switzerland in June 2002. Jan van den Beld, secretary general of ECMA, took part in the meeting and explained how the standardization work should proceed.

The goal is to have a first draft of the standard ready in 2004. To achieve this goal, the group meets at least four times a year and keeps interacting by email in between.

The group members are all Eiffel experts with several years experience and coming from both academia and industry. For the moment, members include ETH Zurich with Bertrand Meyer and me, LORIA in France with Dominique Colnet, Monash University, in Australia, represented by Christine Mingins. This is for the academia. Industry members are AXA Rosenberg with Mark Howard and Éric Bezault, Eiffel Software with Emmanuel Stapf and Bertrand Meyer, and Enea Data with Kim Waldén and Paul Cohen.

New mechanisms

This section presents some extensions to the Eiffel language that have been pre-approved by the committee. (To be finally approved, a mechanism needs to be implemented in at least one Eiffel compiler. For the four extensions presented here, there is no doubt about the final acceptation. Besides, three of these extensions are already implemented at least in part.)

Assertions on attributes

See "Structure of an Eiffel attribute", page 378.

See "Design principles", page 374.

As mentioned before, attributes cannot have contracts in the current version of Eiffel. Only routines can have contracts. This discrimination between routines and attributes goes against the *Uniform Access principle* presented at the beginning of this appendix. Indeed, clients should not have to know whether a feature is implemented by computation or by storage; they just need to know that the class offers this service, no matter what its implementation is.

Therefore the team agreed to introduce a new syntax for attributes, with a new keyword **attribute**, that allows putting preconditions and postcondition:

Assertions on attributes

```
attribute_name: ATTRIBUTE_TYPE is
                -- Comment
        require
                ... Some precondition clauses
        attribute
        ensure
                ... Some postcondition clauses
        end
```

The current notation:

```
attribute_name: ATTRIBUTE_TYPE
                -- Comment
```

becomes a shortcut for:

```
attribute_name: ATTRIBUTE_TYPE is
                -- Comment
        attribute
        end
```

Non-conforming inheritance

In the current version of Eiffel, inheritance always brings conformance. For example, if a class *B* inherits from a class *A*, then type *B* conforms to type *A*. As a consequence, an assignment like *a1* := *b1* where *b1* is of type *B* and *a1* is declared of type *A* is allowed. It is also possible to pass an instance of type *B* as argument of a feature expecting an *A* (for example, *f* (*b1*) with *f* declared as *f* (*arg*: *A*)).

However, sometimes, it may be useful to have inheritance without conformance, namely having subclassing without polymorphism. Non-conforming gives descendant classes access to the parent's features but forbids such assignments and arguments passing as those described above (between entities of the descendant type and entities of the parent type).

This facility is useful only in specific cases like "implementation inheritance". For example, we could have a class *TEXTBOOK* that inherits from *BOOK*. This inheritance relation should be conformant; we want both subclassing and polymorphism. But this class *TEXTBOOK* may need access to some helper features defined in a class *TEXTBOOK_SUPPORT*. One way to get access to these features is to declare an attribute of type *TEXTBOOK_SUPPORT* and have a client relationship. Another way to do it is to use what is usually called "implementation inheritance", that is to say inheriting from *TEXTBOOK_SUPPORT* just to be able to use these features. In that case, we just need subclassing (to get the features), not conformance (we don't want to assign an entity of type *TEXTBOOK* to an entity of type *TEXTBOOK_SUPPORT*). Hence the use of non-conforming inheritance.

Another example taken from EiffelBase is the class *ARRAYED_LIST*, which inherits from *LIST* and *ARRAY*. For the moment, both inheritance links are conformant (there is no other choice!). But what we really want is a class *ARRAYED_LIST* that inherits from *LIST* in a conformant way and a non-conformant link between *ARRAYED_LIST* and *ARRAY* (just to get the features of *ARRAY*, which are useful for the implementation of *ARRAYED_LIST*).

[EiffelBase-Web].

The syntax is not completely fixed yet. For the moment, the idea is to use the keyword **expanded** in front of the parent class name in the **inherit** clause to specify that the inheritance link is non-conformant (hence the name "expanded inheritance", which is sometimes used):

```
class
        B
inherit
        C
        expanded A
feature -- Comment
...
end
```

Possible syntax of non-conforming inheritance

What is the relation between non-conforming inheritance and expanded types? If a class *B* inherits from a class *A*, which is declared as expanded, then there is no conformance between *B* and *A*. Hence the use of the keyword **expanded**.

Non-conforming inheritance is currently being implemented into the SmartEiffel compiler.

Automatic type conversion

The third mechanism is automatic type conversion. The goal is to be able to add, for example a complex number and an integer, or a real and an integer. To do this, the group decided to include a two-side conversion mechanism into the Eiffel language. It is possible to convert *from* and to convert *to* a particular type thanks to a new clause **convert**.

Here is the syntax:

Type conversion mechanism

```
class
        MY_CLASS
create
        from_type_1
convert
        from_type_1 ({TYPE_1}),
        to_type_2: {TYPE_2}
feature -- Conversion
        from_type_1 (arg: TYPE_1) is
                        -- Convert from arg.
                do
                        ...
                end
        to_type_2: TYPE_2 is
                        -- New object of type TYPE_2
                do
                        ...
                end
end
```

If you have an instance of type *TYPE_1* and you want to add it to an instance of *MY_CLASS*, the instance of *TYPE_1* will be automatically converted to an instance of *MY_CLASS* by calling the conversion procedure *from_type_1*, which needs to be declared as a creation procedure.

On the other hand, if you have an instance of type *MY_CLASS* and you want to add it to an instance of type *TYPE_2*, your instance of *MY_CLASS* will be automatically converted to *TYPE_2* by calling the function *to_type_2*.

Here are typical cases that this new mechanism permits to write:

Examples of automatic type conversions

```
        my_attribute: MY_TYPE
        attribute_1: TYPE_1
        attribute_2: TYPE_2

        my_attribute + attribute_1
                -- Equivalent to:
                -- my_attribute + create {MY_TYPE}.from_type_1 (attribute_1)

        attribute_2 + my_attribute
                -- Equivalent to:
                -- attribute_2 + my_attribute.to_type_2
```

Thus, it becomes possible to add complex numbers and integers cleanly and completely transparently to the clients. Part of the automatic type conversion mechanism is already implemented in the ISE Eiffel compiler.

Frozen classes

Another mechanism pre-approved at ECMA is to allow frozen classes in Eiffel, meaning classes from which one cannot inherit. The syntax is simple: the header of a frozen class is extended with the keyword **frozen** as shown next:

Header of a frozen class

```
frozen class
        MY_CLASS
```

Section 18.3 describes the syntax and semantics of frozen classes in detail.

This extension is directly useful to the work presented in this thesis: it enables writing correct singletons in Eiffel, which is not possible with the current version of the language.

"Singleton pattern", 18.1, page 289.

Frozen classes are not supported by classic Eiffel compilers yet but they are already accepted by the ISE Eiffel for .NET compiler.

A.5 BUSINESS OBJECT NOTATION (BON)

This last section describes the BON method, with a focus on notation. It only introduces a small subset of BON — what you need to know to understand the class diagrams appearing in this thesis.

[Waldén-Web].

The method

The Business Object Notation (BON) is a method for analysis and design of object-oriented systems, which emphasizes seamlessness, reversibility and Design by Contract™. It was developed between 1989 and 1993 by Jean-Marc Nerson and Kim Waldén to provide the Eiffel programming language and method with a notation for analysis and design.

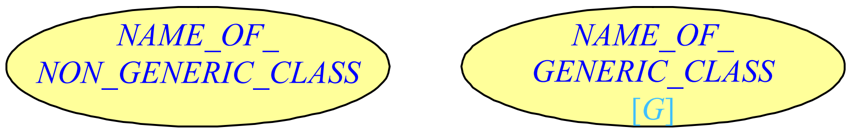
BON stresses simplicity and well-defined semantics. In that respect, it is almost at the opposite of widely-used design notations such as the Unified Modeling Language (UML) or the Rationale Uniform Process (RUP).

As mentioned above, one priority of BON is to bring seamlessness into software development, to narrow the gap between analysis, design, and implementation by using the same concepts and the same semantics for the notation on the tree stages. Therefore BON does not have state-charts or entity-relationship diagrams like in UML because they are not compatible with what is available at the implementation level and would prevent reversibility. Instead, BON relies on pure object-oriented concepts like classes, client and inheritance relationships.

Notation

Here is a catalog of the notations used throughout this thesis:

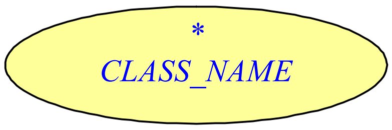
In BON, classes are represented as ellipses, sometimes referred to as "bubbles":



Notation for a class

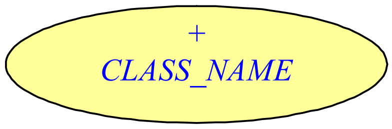
The ellipse may contain information about the class properties, for example:

- *Deferred class*: Class that is declared as **deferred**.



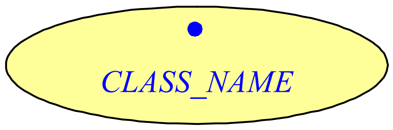
Notation for a deferred class

- *Effective class*: Class that is not **declared** as deferred but has at least one deferred parent, or redefines at least one feature.



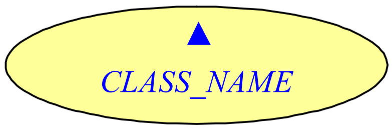
Notation for an effective class

- *Persistent class*: Class that inherits from *STORABLE* or has an indexing tag "persistent".



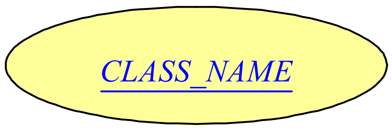
Notation for a persistent class

- *Interfaced class*: Class that interfaces with other programming languages (for example C) through "external features".



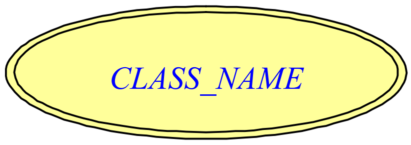
Notation for an interfaced class

- *Reused class*: Class that comes from a precompiled library. (ISE Eiffel compiler provides the ability to compile a library once and for all and then reuse it in so-called "precompiled" form.)



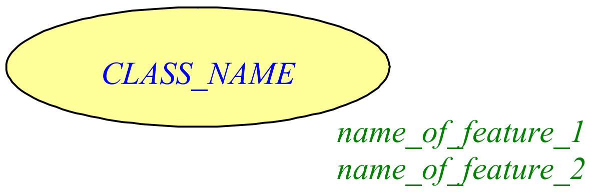
Notation for a reused class

- *Root class*: Class that is the program's entry point.



Notation for a root class

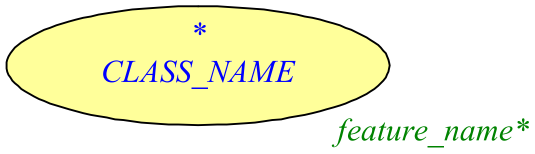
It is also possible to specify the features of a class by writing the feature names next to the ellipse representing the class:



Notation for features

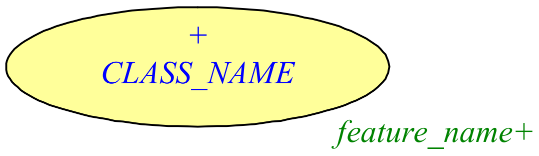
BON also permits to express the different categories of features:

- *Deferred feature*: Non-implemented feature.



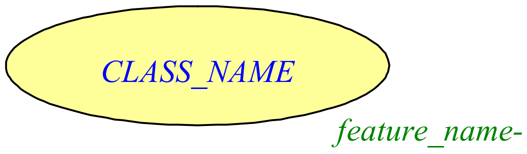
Notation for a deferred feature

- *Effective feature*: Feature that was deferred in the parent and is implemented in the current class.



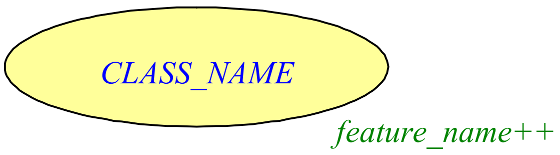
Notation for an effective feature

- *Undefined feature*: Feature that was effective in the parent and is made deferred in the current class through the undefinition mechanism.



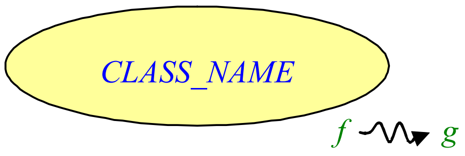
Notation for an undefined feature

- *Redefined feature*: Feature that was effective in the parent and whose signature and/or implementation is changed in the current class through the redefinition mechanism.



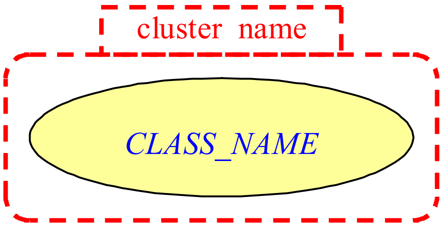
Notation for a redefined feature

BON also provides a notation for feature renaming:



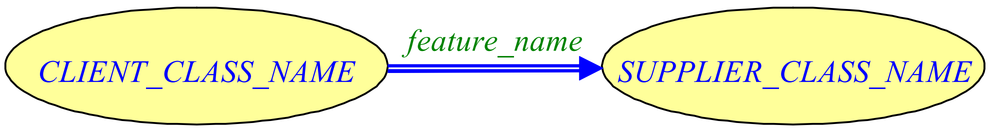
Notation for feature renaming

Classes may be grouped into clusters, which are represented as red stippled-rounded rectangles:



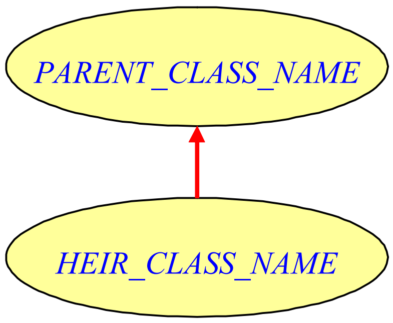
Notation for a cluster

Client relationship between two classes is represented as a blue double-arrow:



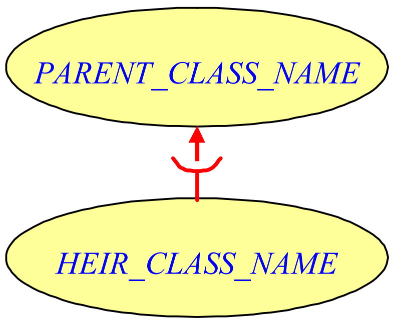
Notation for client/supplier relationship

Inheritance relationship is represented as a red single-arrow:



Notation for (conforming) inheritance

Here is the notation for non-conforming inheritance:



Notation for (non-conforming) inheritance