# Comparison of two object-oriented languages: Eiffel and Ruby

Quentin De Clerck

January 11, 2014

## Contents



# 1 Foreword

This paper is written in an evaluation context of the course Principles of Object-Oriented Programming Languages and has for aim to compare the features of two object-oriented languages. The chosen languages are Eiffel and Ruby. The reasons we decided to choose these two languages is that both have a different type system and philosophies. Eiffel is a statically typed language that aims to produce reusable, extensible and reliable code **REFERENCE** . Ruby is dynamically typed and has for goals simplicity and productivity. .

Ruby version: ruby 2.0.0p247 (2013-06-27 revision 41674) [universal.x86_64-darwin13]. Eiffel version: EiffelStudio 13 (13.11.9.3542 GPL Edition - macosx-x86-64).

The structure of the paper is divided into two sections. The first section compares for both languages the principal features that are present in most object-oriented programming languages. Since both languages are different in many ways, the second section will focus more on the features that are the most specific to the languages and those that reflects the best the philosophies proper to Eiffel and Ruby.

# 2 General OO concepts

The aim of this section is to discuss the design choices of the developers of the languages for main concepts of object-oriented languages and to compare the different approaches.

## 2.1 Everything is an object

Every value in both Eiffel and Ruby are object, even types that are in many languages called primitive types (for example: integers,

booleans). Eiffel and Ruby have a similar structure. There is in both languages a class at the top of the hierarchy, this means a class from which every other class in the language inherits its methods. In Ruby this class is called BasicObject and in Eiffel it is called ANY. Besides the ANY class, Eiffel also has the NONE class, which is the class that inherits from every class in the language.

## 2.2 Access Control

In Eiffel there is no possibility to directly perform an assignment on the value of an attribute. The reason for this inability to assign attributes from the outside is because in Eiffel it is impossible to know from the outside of the object if the feature called is a stored or computed value. If there are changes in the implementation, they do not affect the client class by forcing it to change its interface. This concept is called the uniform-access principle and is central in Eiffel. Because it is impossible to know if the expression is an attribute or a function, the only way to change the state of an object is thus to make a procedure than internally modifies the state: a "setter". But there exist a facility to make it look like assignment is directly possible. This mechanism is called assigner command and consists of specifying in the declaration of the attribute which is the related assignment procedure. The assignment of the attribute will be transformed at compile time in the assignment procedure specified in the declaration. They implemented this facility because developers are used to direct access in other programming languages.

An instance variable in Ruby cannot be read/written without calling a method. Thus there is a need for a "getter" and "setter". The keyword attr_read, attr_write and attr_accessor are syntactic sugar for creating theses methods. Like in Eiffel, the client is unaware if the method is a stored value or a computed one. Thus Ruby also implements the uniform-access principle.

Now there are other mechanisms we have not discussed yet about access controls, namely how to control access to methods to clients outside the scope of the class.

There are three kinds of access controllers in Ruby:

**public** accessible without restrictions.

**protected** *only* accessible within the class and subclasses.

**private** inaccessible *if receiver is explicit* within class and subclasses.

What is meant with ïf the receiver is explicitïs that if the method is called for a specific object, like self or a parameter, then the call will result in an error.By default, every method is public in Ruby and it is possible to change the visibility of the methods at run-time due to the dynamic nature of the language. This dynamic nature will be discussed in a later section.

Eiffel has another approach called Selective Export. It specifies a list of clients to export, enabling them to get access with the features they were listed for. This approach enables to be very precise about the scope of the features. The different possibilities are:

- Making a set of features private to the class by specifying that the feature set should not be exported: {NONE}.
- Making a set of features public to every possible client by specifying nothing or by specifying: {ANY}.

- Making a set of features public to a set of clients by specifying the clients, for example {Class_A, Class_B, Class_C}. It is possible to specify the current class as client, then every subclass will inherit the feature.

This export technology allows to be very specific in the choice of accessible features. Export violations are statically checked by the compiler and thus are detected at compile-time and not at run-time.

### 2.3 Inheritance

Both languages support single class inheritance but only Eiffel supports multiple inheritance. However Ruby supports mixins which offers the same possibilities as multiple inheritance.

Multiple inheritance is a object-oriented feature where a class inherits from more than one parent class. This can lead to problems like the diamond problem depicted in figure 1. The diamond problem arises when two or more parent classes inherit from the same superclass. This will provoke nameclashes in the subclass inheriting from the multiple parents. Eiffel provides a flexible approach to multiple inheritance. It introduces different keywords that enable to adapt the features inherited from the parent classes. The keywords that are provided for feature adaptation are:

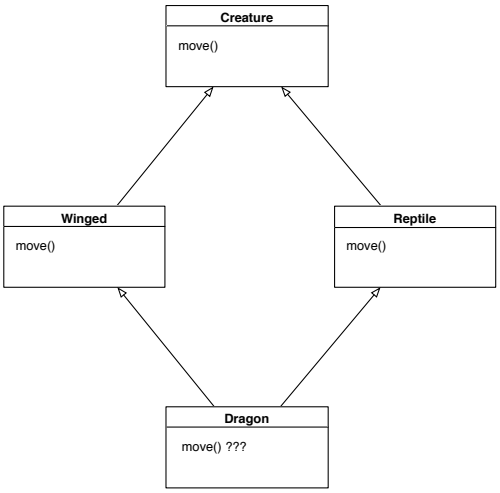


Figure 1: Example of the diamond problem

**rename** Renames a inherited feature.

**export** Changes the export list of the inherited features.

**undefine** Removes one of the inherited feature definitions.

**redefine** Redefines one of the inherited feature definitions.

**select** Selects the feature to use when there are homonyms.

Thus the diamond problem can be easily solved in Eiffel thanks to the provided tools. A simple example for solving the depicted problem in figure 1 is to rename the move feature from the Winged class into fly and select the fly feature. Another approach could be to undefine one of the features and selecting the other. Inheritance has also an impact in Design by Contract, a language-specific feature of Eiffel that will be discussed later on in the paper.

Ruby does not support multiple inheritance, but a *mixin* can be an equivalent feature. Before explaining what a mixin is it is important to explain modules in Ruby. A module is a sort of namespace grouping variables and functions together for obtaining a whole that provides functionalities. Modules cannot be instantiated, their purpose is to add functionality to a class. A mixin allows to include a module as a sort of superclass to the desired class, to mix the module in the class. It is possible to mix more than one module in a class, thus it looks very similar to multiple inheritance. In the example from figure 1,

the superclasses could be two modules that implement the different move behaviors. But even if Ruby uses mixins instead of multiple inheritance, the nameclash problem persists. Ruby resolves it automatically by overriding the previous definition, thus it is important for the programmer to be aware of this and give another name to one of the definition if the two method definitions are needed.

### 2.4 Polymorphism

Polymorphism is a key feature in object-oriented programming languages. Both Ruby and Eiffel support this feature but in a very different conceptual way. Eiffel supports subtype polymorphism, it means that Eiffel allows polymorphism only for the types that have a superclass in common. Thus inheritance is primordial for subtype polymorphism.

In Ruby it is also possible to achieve it using inheritance, but this is more a consequence of the mechanism that permits Ruby to support polymorphism. Ruby is dynamically typed and supports a special style of typing namely, *duck typing*. Duck typing focusses on the methods of an object instead of its type. If the method is supported by the object it will be called whatever the type or output is. If the method call is not supported, then a run-time error is returned. Duck typing is the concept used for polymorphism in Ruby allows thus polymorphism without inheriting from a superclass. It is trivial why polymorphism also works with inheritance in Ruby, classes that inherit from a superclass inherit its methods and duck typing focusses on the presence of methods.

So even if both languages support polymorphism, their approach is completely different. Polymorphic calls are dependent of the type of objects in Eiffel and in Ruby they are dependent of the presence of the method. From a software engineering point of view it is logical that Eiffel focusses on the subtype polymorphism. First it is statically typed, thus there should be a specific type declared with the variable or parameter. However, this could be resolved with a keyword that instructs the compiler that a variable should be dynamically typed. Second and most importantly, this sort of solution is not in the philosophy of the Eiffel language. One of the goals of Eiffel is to produce software that is reliable and maintainable. If duck typing should be adapted in Eiffel then it would be only reliable if the programmers know exactly which type of objects will passed to the methods and the maintainability would also be tricky. Imagine that the call changes of name in one of the classes, with duck typing it would not be clear that a method is used in a polymorphic call and by changing its name the programmer introduces errors that are difficult to resolve. Thus it is important to be aware of the methods that are used in a polymorphic context. Subtype polymorphism in a statically typed language is a much more reliable mechanism because the polymorphic calls are only possible for a predefined restricted set of types. It is also more easily maintained, because it is checked at compile time that every class implements the method.

### 2.5 Reflection

## 3 Language-specific Features

### 3.1 Eiffel

#### 3.1.1 Deferred Classes

#### 3.1.2 Frozen Classes

#### 3.1.3 Generic Classes

#### 3.1.4 Design by Contract

If there is one language specific feature that is essential in Eiffel, then this concept is *Design by Contract*. The idea is that every system has interacting components and that their cooperation should follow some strict specifications (the contract) that settle the obligations and benefits for both client and supplier. The obligations have to be satisfied before feature calls and are called *preconditions*. The benefits describe what the result should be if the precondition was met. These benefits are called *post-conditions*. Every contract also include class invariants, which are conditions that have to be ensured during the lifetime of an object, including at its creation.

### 3.2 Void-safety

### 3.3 Ruby

#### 3.3.1 Open Objects

#### 3.3.2 Meta-classes

## 4 Conclusion

## 5 References

Listing 1: Access Control in Ruby

```ruby
# CODE FOR ACCESS IDENTIFIERS

class Lord

  private

  def plot
    puts "I plot to behead king Joffrey"
  end

  protected

  def mistrust
    puts "I want to conspire, but hold it secret"
  end


  public

  def toad
    puts "You are such a magnificent person, my grace"
  end

  def publicTalk
    toad
    self.toad
  end

  def protectedTalk
    mistrust # works
    self.mistrust #works

  end

  def privateTalk
    plot #works
    self.plot #does not work
  end



end

l = Lord.new
l.publicTalk
l.protectedTalk
l.privateTalk

class Lord
  public

    def plot
      puts "I say it publicly: I want to behead king Joffrey!"
    end
end

l.privateTalk
```

Listing 2: Access Control in Eiffel

```
1----------------
2-- CLASS LORD --
3----------------
4
5note
6  description: "LORD diplomacy class."
7
8class
9  LORD
10
11create
12  make
13
14feature {ANY} -- public
15
16  name: STRING assign set_name --
        assigner command
17
18  set_name (n : STRING)
19    do
20      name := n
21    end
22
23feature {NONE} --initialization
24
25  make (name_lord: STRING)
26    do
27      name := name_lord
28      print("I am lord ")
29      print(name)
30      print("%N")
31    end
32
33feature {NONE} -- private, will not be
      called outside this scope
34
35  plot
36    do
37      print ("I plot to behead king
          Joffrey%N")
38    end
39
40
41
42feature -- public, syntactic sugar for
      feature {ANY}
43
44  toad
45    do
46      print ("You are such a magnificent
          person, my grace%N")
47    end
48
49
50feature {LORD} -- public for specified
      classes and subclasses, same as
      protected in C++ for example
51
52  mistrust
53    do
54      print("I want to conspire, but hold
          it secret%N")
55    end
56
57feature {LIEGELORD} -- public for
      specified classes and subclasses,
      will only work in LIEGELORD class and
       subclasses
58
59  allegiance (n : STRING)
60    do
61      print("I am your humble subject, my
          lord ")
62      print(n)
63      print("%N")
64    end
65
66end
67
68----------------------
69-- CLASS LIEGELORD --
70----------------------
71
72note
73  description: "LIEGELORD diplomacy class
      ."
74
75class
76  LIEGELORD
77
78inherit
79  LORD
80
81create
82
83  makeLiege
84
85feature
86
87  subject : LORD
88
89feature {NONE} -- Initialization
90
91  makeLiege (n: STRING man : LORD)
92    do
93      name := n
94      subject := man
95      man.allegiance (n) -- works within
          the scope of LIEGELORD
          subclasses
96      print("Yes you are, lord ")
97      print(subject.name)
98      print("%N")
99      man.mistrust -- works within the
          scope of LORD subclasses
100     -- man.plot does not work
101   end
102
103end
104
105----------------
106-- ROOT CLASS --
```

```
107----------------
108
109note
110   description : "Eiffel-project
          application root class"
111
112class
113   APPLICATION
114
115create
116   make
117
118feature {NONE} -- Initialization
119
120   make
121         -- Run application.
122      local
123         lord: LORD
124         liege: LIEGELORD
125      do
126         create lord.make ("Karstark")
127         create liege.makeliege ("Stark",
              lord)
128
129         lord.toad
130         liege.toad
131      end
132
133end
```

Listing 3: Inheritance in Ruby

```
1# CODE FOR INHERITANCE
2
3class Creature
4   def initialize name
5      @name = name
6      puts "Creature #{name}"
7   end
8
9   def move
10     puts "AAArg!! cannot move without
           legs!!"
11  end
12end
13
14module Winged
15
16  def fly
17     puts "Flying creature"
18  end
19end
20
21module Reptile
22  def move
23     puts "Crawling creature"
24  end
25end
26
27class Dragon < Creature
28  include Winged
29
30  include Reptile
31
32  def breatheFire
33     puts "Rooooooooooh!"
34  end
35end
36
37balerion = Dragon.new "Balerion"
38balerion.fly
39balerion.move
```

Listing 4: Inheritance in Eiffel

```
1
2----------------------
3-- CLASS CREATURE --
4----------------------
5note
6   description: "A class modeling a mythic
        CREATURE."
7
8deferred class
9   CREATURE
10
11feature
12
13   move
14      deferred
15
16      end
17end
18
19----------------------
20-- CLASS REPTILE --
21----------------------
22
23note
24   description: "REPTILE inheriting from
        CREATURE."
25
26class
27   REPTILE
28
29inherit
30   CREATURE
31
32feature
33   move
34      do
35         print ("creature crawls on the
              ground")
36            print ("%N")
37      end
38
39end
40
41----------------------
42-- CLASS WINGED --
43----------------------
44
45note
```

```
46    description: "WINGED inheriting from
          CREATURE."
47
48 class
49    WINGED
50
51
52 inherit
53    CREATURE
54
55 feature
56    move
57       do
58          print ("creature flies in the air")
59          print ("%N")
60       end
61
62 end
63
64 ----------------
65 -- CLASS DRAGON --
66 ----------------
67
68 note
69    description: "DRAGON multiple
          inheritance from diamond problem
          example."
70
71 class
72    DRAGON
73
74 inherit
75    WINGED
76       rename
77          move as fly
78          select fly
79          end
80    REPTILE
81
82 create
83       make
84
85 feature
86
87    name: STRING
88
89 feature -- Initialization
90
91       make (dragon_name: STRING)
92
93          do
94              name := dragon_name
95              print (name)
96              print ("%N")
97          end
98
99 end
100
101 ----------------
102 -- ROOT CLASS --
103 ----------------
104
```

```
105 note
106    description : "Eiffel-project
           application root class"
107
108 class
109    APPLICATION
110
111 create
112    make
113
114 feature {NONE} -- Initialization
115
116    make
117          -- Run application.
118       local
119          dragon: DRAGON
120       do
121          create dragon.make ("Balerion")
122          dragon.fly
123          dragon.move
124       end
125 end
```

Listing 5: Polymorphism in Ruby

```
1  class Knight
2    def initialize(name)
3      @name = "ser " + name
4    end
5
6    def fight
7      puts "#{@name} shouts: FOR THE
           RIGHTFUL QUEEN!!"
8    end
9  end
10
11 class Sellsword
12   def initialize(name)
13     @name = name
14   end
15
16   def fight
17     puts "#{@name} asks: How much are you
           willing to pay??"
18   end
19 end
20
21 def defendQueen(knight)
22   knight.fight
23 end
24
25 Barristan = Knight.new("Barristan")
26 Bronn = Sellsword.new("Bronn")
27 defendQueen(Barristan)
28 defendQueen(Bronn)
```

Listing 6: Polymorphism in Eiffel

```
1 ----------------
2 -- CLASS WARRIOR --
3 ----------------
4
```

```
note
  description: "Superclass WARRIOR."

deferred class
  WARRIOR

feature

  fight
    deferred
    end
end

--------------------
-- CLASS SELLSWORD --
--------------------

note
  description: "SELLSWORD subclass for
      polymorphism."

class
  SELLSWORD

inherit
  WARRIOR

create
  make

feature

  name: STRING

  make (n : STRING)
    do
      name := n
    end

  fight
    do
      print(name)
      print(" asks: How much are you
          willing to pay??%N")
    end

end

------------------
-- CLASS KNIGHT --
------------------

note
  description: "KNIGHT subclass for
      polymorphism."

class
  KNIGHT

inherit
  WARRIOR

create
  make

feature

  name: STRING

  make (n : STRING)
    do
      name := n
    end

  fight
    do
      print(name)
      print(" shouts: FOR THE RIGHTFUL
          QUEEN!!%N")
    end
end

----------------
-- ROOT CLASS --
----------------

class
  APPLICATION

create
  make

feature {NONE} -- Initialization

  defendQueen (warrior: WARRIOR)
    do
      warrior.fight
    end

  make
      -- Run application.
    local
      bronn: SELLSWORD
      barristan: KNIGHT

    do
      create bronn.make ("Bronn")
      create barristan.make ("Barristan")
      defendQueen(bronn)
      defendQueen(barristan)
    end

end
```