Comparison of two object-oriented languages: Eiffel and Ruby

Quentin De Clerck

January 13, 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 [2]. Ruby is dynamically typed and has for goal to be simple and flexible [11].

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 some 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[10] and in Eiffel it is called ANY[2]. Besides the ANY class, Eiffel also has the NONE class, which is the class that inherits from every class in the language. Ruby pushes the object-orientation a little bit further with meta-objects. This subject will come back later on in section 3.3.2.

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* [2, 8] 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[7]:

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 "if the receiver is explicit" is 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*[2, 1] . 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 similar possibilities as multiple inheritance.

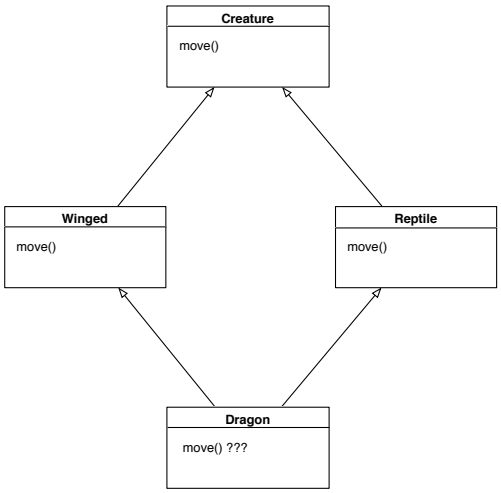

Figure 1: Example of the diamond problem

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:

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.

Ruby does not support multiple inheritance, but a *mixin* [7, 13] 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 behaviours. 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*[9]. 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 for the variable or parameter. However, this could be resolved with a keyword that instructs the compiler that a variable should be dynamically typed (like the "dynamic" keyword of C#). 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. The maintainability of the software would also be tricky, imagine that a method call changes of name in one of the classes, with duck typing it would not be clear that this method was used in a polymorphic call and by changing its name the programmer introduces errors that can be tedious to resolve. Thus it is important to be aware of the methods that are used in a polymorphic context when a language supports duck typing. Subtype polymorphism in a statically typed language is a much more reliable mechanism because the polymorphic calls are only possible for a restricted set of types and eliminate a lot of possible failures. It is also more easily maintained, because it is checked at compile time that every class implements the method. In Ruby's case, it is also logical that duck typing is used and not subtype polymorphism. First, subtype polymorphism is implicit with duck typing. Second, Ruby is dynamically typed, which means the types are known at run-time. This implies that there would be no secure way to ensure that the method call is applied on a particular class hierarchy. Perhaps it could be achieved with a kind of type cast for the desired superclass. But then it would still return a run-time error when the types are not correct, like it does when the method is not defined for an object. This is much more restrictive and goes also against the philosophy of Ruby which is simplicity and flexibility.

3 Language-specific Features

While the first section enumerates differences in general object-oriented concepts, this section focusses more on features that characterize the languages. The aim is not make an exhaustive list of features but more to pick some that really show the purpose and philosophy of the languages.

3.1 Eiffel

The goal of Eiffel is to provide rather a method that guides the programmer in software development than only a language for programming. It focusses on some the whole software development process and on the quality of the software.

3.1.1 Design by Contract

If there is one language specific feature that is essential in Eiffel, then this concept is *Design by Contract*[2, 3]. 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. They are called *preconditions* and are introduced by the *require* keyword. The benefits describe what the result should be if the precondition was met. Benefits are thus *postconditions* and can be specified using the *ensure* keyword. Every contract also include *class invariants*, which are conditions that have to be ensured during the lifetime of an object, including at its creation. Class invariants are specified after the *invariant* keyword. These are the three main categories of contracts and are implemented using assertions. Each assertion may be tagged, it is not mandatory and does not influence the contract but it is helpful for debugging and provides extra documentation. Since assertions are boolean expressions, it is possible to formulate them in function calls. This enables to express more complex conditions.

There are still three other types of assertions:

- Instruction check: checks if a certain condition is respected at a specific moment during the execution.
- Loop invariants: states that some conditions have to be ensured when exiting the loop.
- Loop variant: make sure the loop is finite by decreasing an integer expression at each loop iteration and check that the integer stays positive.

Even if Design by Contract is not mandatory to use when developing in Eiffel, it is strongly encouraged because it has many benefits. It is a method that helps the developers for designing and implementing correct software in first instance. They push the developers to think about specifications for the code to write. It has already been pointed out that using tags for assertions are useful for code documentation and debugging. Contracts serve also to generate automatically documentation in Eiffel, this means that the documentation is always up-to-date. Design by Contract is thus a methodology that encourage the programmer to think about the code, to write specifications down about the code and to design the code such that the specifications are fulfilled. This is thus a great feature for reliability and maintainability of the software.

There exist libraries in many programming languages that offers Design by Contract, even for Ruby. A basic implementation of Design by Contract in Ruby 7 has been implemented for this paper. It uses reflection in order to get information about the variables and metaprogramming for evaluating the expressions passed to the require, ensure and invariant clauses.

3.2 Void-safety

Void-safety[4] is a language feature that protects the software from run-time errors caused by method calls to void references. References are used for accessing objects in object-oriented programming languages. This can lead to problems when the reference is Void (or null in other languages).

Eiffel is statically typed and thus can ensure that a feature will be applied at

run-time to the correct object. But nothing ensures that the object will exist when the feature will be executed. With Void-safety the compiler can give the assurance that an object will be attached to the reference whenever the feature is executed. In other words, the compiler analyses the code statically and ensures that feature calls are valid only if the feature executes a call on an attached object and not to Void.

There are patterns that check if a variable is void-safe. The *Certified Attachment Pattern* (CAP) checks if a local variables or formal parameters is void. The *attached syntax* takes a step further. It is another sort of CAP that checks if the object is attached and provides a safe access to the objects that are attached as class attributes. Eiffel introduced two kind of types in order to assure the void-safeness of the software:

Attached Type: The compiler will prevent a variable of an attached type to be set to Void.

Detachable Type: Theses variables may be set to Void. Thus direct access to detachable typed variables is never void-safe.

It is also important to note that it is impossible to assign a detachable variable to an attached one, but the opposite is possible. The creation procedure is responsible for ensuring that all the attributes of an attached type are set after the creation.

This is a feature that improves the reliability of the produced software and shows again that Eiffel's primary concern is to enhance the software quality. In Ruby there is nothing like void-safety. Void safety is achieved at compile time and Ruby is an interpreted language, thus there is no way to make it void safe except by checking if the object is the nil object.

3.3 Ruby

Ruby is a programming language known for being very object-oriented and for its simplicity. It is also very flexible like duck-typing already attested in previous section.

3.3.1 Open Classes

In Ruby it is always possible to add new methods to an existing class, the class definitions can at any moment be opened for modifications. Even built-in classes are can be adapted. Adding or modifying content at run-time to an already existing class definition without altering the source code is known as *monkey patching* [6].

Monkey patching can be useful for different applications like extending or modifying the behaviour of an object at run-time from a third-party software without changing the source code. This enables for example to reuse existing code and adapt it in different files with different behaviours.

In first instance this feature seems really helpful because it gives the possibility to the developer to adapt code when needed. For example, as workaround to a bug by implementing a specific solution in this part of the code. But with great power comes great responsibility. Monkey patching can lead to bugs that break the code. If the source code changes, it is possible

that the patch behaves differently because it makes assumptions on the code that do not hold anymore. Different patches for the same method create conflicts and one patch will override the other. Thus it is important to be aware of the patches and save the implementation of a method if necessary by renaming it with an alias. It is also important to document the patches and method that are patched, this can prevent confusion if one forgets or does not know about the existing patches and does not understand why the code does not act like expected.

To summarize, open classes is thus a language feature that offers a lot of modularity and flexibility. It enable to extend and reuse third-party code in a simple way. This is certainly why this feature has been added in Ruby. But it is very important to note that open classes are in conflict with the principle of encapsulation. Every method can be accessed with open classes, even private methods and they can be changed into public ones. It is thus really important of not abusing of this feature.

3.3.2 Metamodel

A metaclass [5, 12] is a special class that has classes as instances. It defines thus methods that every class and its instances should have. When asking to a class in Ruby from which class it is an instance, it will always return the class Class. This is logical since every class should inherit the methods defined in it. Thus the only metaclass in Ruby is thus the class Class.

But lets look the meta-system a little bit more in details. In Ruby there exist meta-objects called *eigenclasses*. Every object has its own eigenclass. Because classes are also objects in Ruby, they also have eigenclasses. And what is interesting is that eigenclasses also have eigenclasses. This is recursive and goes on forever. Thus there are three sorts of eigenclasses: eigenclasses for objects and for classes.

But what is the usefulness of theses eigenclasses? When an instance method is defined, it is stored in the class. When a class method is defined, it is stored in the class' eigenclass. Thus the eigenclass serves to store class variables and class methods. Eigenclasses of objects are useful if the developer wants that this particular object has a different behaviour than the other objects of the same class. Specialized method for this object will be stored in the eigenclass of the object. And finally, Eigenclasses of eigenclasses do not store any method or data, thus the reason of there existence is for conceptually correctness.

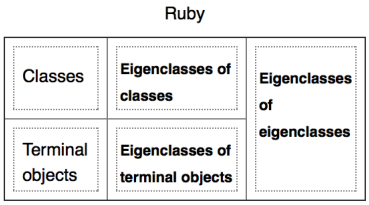

Figure 2: The Ruby metaclass model. Image from Wikipedia [5]

The method lookup mechanism works thus as follows:

1. A method is called on the object

2. The method is looked up in the methods of the object's eigenclass

3. If the method is not found, the lookup continues in the class of the object

4. If the method is still not found, it looks the whole inheritance structure up to end with the class Class.

The same lookup procedure is applied for class method calls, but then it is the eigenclass class that is looked up. Also it is interesting to note that the superclass of the eigenclass of the BasicObject class is the class Class, while the superclass of the BasicObject class is nil.

This metamodel description gives more explanation on how everything in Ruby are object. It also emphasizes another way of how Ruby can be flexible, every object can have its own specialized behaviour if the behaviour is added to its eigenclass object. Eiffel has no such metaclass model, even if everything is an object.

4 Conclusion

It is clear that even if Eiffel and Ruby are both object-oriented languages, they are very different in many aspects. Eiffel tries to achieve reliability through controlling everything. A good example is the selective export. It defines everything that may be exported and specifies a detailed list of clients. Ruby is more into the simplicity and flexibility. It gives the developer the tools for doing whatever they want and for complete different programming paradigms. The philosophies of the two languages are thus really different. Eiffel would probably be more used in the industry where people need a good framework for producing quality software. Ruby is a higher level programming language that can be used for a lot of different purposes, going from small to larger project. But in large project Ruby seems less adequate than Eiffel. Developers need to be aware of possibilities of Ruby's flexibility and may not abuse of it, otherwise the software will be difficult to maintain.

5 References

[1] Ilinca Ciupa and Bertrand Meyer. Lecture 3: Language constructs for modularity and information hiding. `http://se.inf.ethz.ch/old/teaching/ss2007/0050/slides/03_softarch_info_hiding_lang_constr_6up.pdf`, 2007. Online; Accessed 13-January-2014.

[2] Bertrand Meyer and Rich Ayling. An eiffel tutorial. 2001.

[3] n.d. Eiffel: The essentials. `http://se.inf.ethz.ch/courses/2013b_fall/eprog/additional_materials/eiffel_the_essentials.pdf`. Online; Accessed 13-January-2014.

[4] n.d. Void-safe programming in eiffel. `http://docs.eiffel.com/book/export/html/2200`. Online; Accessed 13-January-2014.

[5] n.d. Metaclass— Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Metaclass`,

December 2013. Online; Accessed 13-January-2014.

[6] n.d. Monkey patching— Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Monkey_patching, December 2013. Online; Accessed 13-January-2014.

[7] n.d. Ruby Programming/Syntax/Classes— Wikibooks. http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Classes, May 2013. Online; Accessed 13-January-2014.

[8] n.d. Uniform access principle— Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Uniform_access_principle, December 2013. Online; Accessed 13-January-2014.

[9] n.d. Duck typing— Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Duck_typing, January 2014. Online; Accessed 13-January-2014.

[10] ruby doc.org. Basicobject class. http://ruby-doc.org/core-2.0/BasicObject.html. Online; Accessed 13-January-2014.

[11] ruby lang.org. About ruby. http://www.ruby-lang.org/en/about/. Online; Accessed 13-January-2014.

[12] A. Sundararaja. Metaclasses in ruby and smalltalk. https://blogs.oracle.com/sundararajan/entry/metaclasses_in_ruby_and_smalltalk, September 2006. Online; Accessed 13-January-2014.

[13] David Thomas and Andrew Hunt. Programming ruby. *Dr. Dobb's Journal*, 5, 2001.

6 Code listings

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

```
------------------
-- CLASS LORD --
------------------

note
  description: "LORD diplomacy class."

class
  LORD

create
  make

feature {ANY} -- public

  name: STRING assign set_name --
      assigner command

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

feature {NONE} --initialization

  make (name_lord: STRING)
    do
      name := name_lord
      print("I am lord ")
      print(name)
      print("%N")
    end

feature {NONE} -- private, will not be
    called outside this scope

  plot
    do
      print ("I plot to behead king
          Joffrey%N")
    end



feature -- public, syntactic sugar for
    feature {ANY}

  toad
    do
      print ("You are such a magnificent
          person, my grace%N")
    end


feature {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
57 feature {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
66 end
67
68 ----------------------
69 -- CLASS LIEGELORD --
70 ----------------------
71
72 note
73   description: "LIEGELORD diplomacy class
         ."
74
75 class
76   LIEGELORD
77
78 inherit
79   LORD
80
81 create
82
83   makeLiege
84
85 feature
86
87   subject : LORD
88
89 feature {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
103 end
```

```
104
105 ----------------
106 -- ROOT CLASS --
107 ----------------
108
109 note
110   description : "Eiffel-project
        application root class"
111
112 class
113   APPLICATION
114
115 create
116   make
117
118 feature {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
133 end
```

Listing 3: Inheritance in Ruby

```
1 # CODE FOR INHERITANCE
2
3 class 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
12 end
13
14 module Winged
15
16   def fly
17     puts "Flying creature"
18   end
19 end
20
21 module Reptile
22   def move
23     puts "Crawling creature"
24   end
25 end
26
```

```ruby
class Dragon < Creature
  include Winged

  include Reptile

  def breatheFire
    puts "Roooooooooh!"
  end
end

balerion = Dragon.new "Balerion"
balerion.fly
balerion.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
48class
49   WINGED
50
51
52inherit
53   CREATURE
54
55feature
56   move
57      do
58         print ("creature flies in the air")
59         print ("%N")
60      end
61
62end
63
64------------------------
65-- CLASS DRAGON --
66------------------------
67
68note
69   description: "DRAGON multiple
        inheritance from diamond problem
        example."
70
71class
72   DRAGON
73
74inherit
75   WINGED
76      rename
77         move as fly
78         select fly
79         end
80   REPTILE
81
82create
83      make
84
85feature
86
87   name: STRING
88
89feature -- Initialization
90
91      make (dragon_name: STRING)
92
93            do
94                 name := dragon_name
95                 print (name)
96                 print ("%N")
97            end
98
99end
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
5 note
6   description: "Superclass WARRIOR."
7
8 deferred class
9   WARRIOR
10
11 feature
12
13   fight
14     deferred
15     end
16 end
17
18----------------------
19-- CLASS SELLSWORD --
20----------------------
21
22 note
23   description: "SELLSWORD subclass for
         polymorphism."
24
25 class
26   SELLSWORD
27
28 inherit
29   WARRIOR
30
31 create
32   make
33
34 feature
35
36   name: STRING
37
38   make (n : STRING)
39     do
40       name := n
41     end
42
43   fight
44     do
45       print(name)
46       print(" asks: How much are you
             willing to pay??%N")
47     end
48
49 end
50
51----------------------
52-- CLASS KNIGHT --
53----------------------
54 note
55   description: "KNIGHT subclass for
         polymorphism."
56
57 class
58   KNIGHT
59
60 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
```

Listing 7: Design by Contract in Ruby

```
module Contract
  attr_accessor :old
  @invariants = []

  # checks if attribute has to be stored
  def eligible attribute
    nonEligible = ["@old", "@invariants"]
    nonEligible.collect do |var|
      if var == attribute
        return false
      end
    end
    return true
end

# Returns an older value for the given
    attribute
def getOld oldAttr
  return @old["#{oldAttr}"]
end

# Stores the values of the attributes
def setOld(bind)
  @old = Hash.new
  vars = self.instance_variables
  vars.collect do |v|
    if eligible("#{v}")
      @old["#{v}"] = eval("#{v}",bind)
    end
  end
end

# Returns to previous state (old
    attribute state)
def callback
  vars = self.instance_variables
  vars.collect do |v|
    if eligible("#{v}")
      val = @old["#{v}"]
      eval("#{v}␣=␣#{val}")
    end
  end
end

# check if the conditions are
    maintained
def checkCond(cond, bind)
  cond.collect do |c|
    if !eval(c, bind)
      raise "precondition␣violation"
    end
  end
end

# Require clauses
def requir(cond, bind)
  setOld(bind)
  checkCond(cond, bind)
end

# Ensure clauses
def ensur(cond, bind, retVal = nil)
  begin
    checkCond(cond, bind)
  rescue
    callback
    raise "postcondition␣violation"
  end
```

```
    checkInvariants bind
    return retVal
  end

  # Class invariant clauses
  def classInvariant(inv, bind)
    setOld(bind)
    @invariants = inv
    checkInvariants bind
  end

  #check the class invariant clauses
  def checkInvariants bind
    begin
      checkCond(@invariants, bind)
    rescue
      callback
      raise "class invariance violation"
    end
  end
end

# Example how it could be used in a class
class Winterfell

  attr_accessor :maxStarks
  attr_accessor :minStarks
  attr_accessor :starks

  include Contract

  def initialize
    @maxStarks = 7
    @minStarks = 1
    @starks = 7

    #define the class invariants
    classInvariant(['@starks <= @maxStarks',
      '@starks >= @minStarks'], binding)

  end

  def starksLeave amount
    requir(['amount > 0', 'amount < 5'], binding) # require clause

    @starks = @starks - amount

    ensur(['@starks == (getOld(:@starks) - amount)'], binding, @starks) # ensure clause
  end

end


w = Winterfell.new
w.starksLeave 1
w.starksLeave 7 # invariance violation
w.starksLeave 0 # precondition violation
w.starksLeave 5 # precondition violation
```

Listing 8: Design by Contract in Eiffel

```
-----------------------
-- CLASS WINTERFELL --
-----------------------

note
  description: "CLASS representing the
      castle of Winterfell. There should
      always be a Stark in Winterfell"

class
  WINTERFELL

create

  build

feature

  number_of_starks : INTEGER
  min_starks : INTEGER
  max_starks : INTEGER

  build
    do
      number_of_starks := 7
      min_starks := 1
      max_starks := 7
      starks_present
    end

  starks_present
    do
      print(number_of_starks)
      print(" Starks are present in
          Winterfell%N")
    end

  starks_leaving_winterfell (amount:
      INTEGER)
    require  -- precondition
      non_negative: amount > 0
    do
      number_of_starks :=
          number_of_starks - amount
      starks_present
    ensure  -- postcondition
      leaved: number_of_starks = old
          number_of_starks - amount
    end

  starks_entering_winterfell (amount:
      INTEGER)
    require
      non_negative: amount > 0
    do
      number_of_starks :=
          number_of_starks + amount
      starks_present
    ensure
```

```
52        entered: number_of_starks = old
              number_of_starks + amount
53      end
54
55 invariant -- class invariant
56   always_a_stark: number_of_starks >=
         min_starks --with tag
57   number_of_starks <= max_starks --
         whitout tag
58 end
59
60
61 ------------------
62 -- ROOT CLASS --
63 ------------------
64
65 class
66   HELLO
67
68 create
69   make
70
71 feature {NONE} -- Initialization
72
73   make
74       -- Run application.
75     local
76       winterfell: WINTERFELL
77
78     do
79       create winterfell.build
80       winterfell.
             starks_leaving_winterfell (3)
81       --winterfell.
             starks_leaving_winterfell (4)
             raises contract violation
82       winterfell.
             starks_entering_winterfell (2)
83     end
84
85 end
```

Listing 9: Open classes in Ruby

```
1  class Dragon
2
3    def breatheFire
4      puts "It is still a baby dragon, it
           cannot breathe fire"
5    end
6
7  end
8
9  dragon = Dragon.new
10 dragon.breatheFire
11
12 #Add a method fly
13 class Dragon
14
15   def fly
16     puts "It is flying high into the sky"
17   end
18 end
19
20 dragon.fly
21 dragon.breatheFire
22
23 #Modify the breatheFire method
24 class Dragon
25
26   alias babyFire breatheFire # renames
         previous method
27   def breatheFire
28     puts "Fire is everywhere"
29   end
30 end
31
32 dragon.fly
33 dragon.babyFire
34 dragon.breatheFire
```

Listing 10: Metamodel in Ruby

```
1  class GameOfThrones
2
3    @@numberOfPlayers = 0 # class variable
4
5    def initialize
6      @@numberOfPlayers = @@numberOfPlayers
           + 1
7    end
8
9    def numberOfPlayers
10     return @@numberOfPlayers
11   end
12
13   class <<self    # self refers here to
         the class itself
14
15     def whatAreYou
16       "I am the Game Of Thrones"
17     end
18   end
19
20   def whatAreYou
21     "I am a player"
22   end
23 end
24
25 #init the objects
26 tyrion = GameOfThrones.new
27 tyrion.numberOfPlayers # 1
28 cersei = GameOfThrones.new
29 cersei.numberOfPlayers # 2
30 cersei.numberOfPlayers # 2
31
32
33 #look at the classes
34 tyrion.class # class GameOfThrones
35 cersei.class # class GameOfThrones
36 GameOfThrones.class # class of a class is
       the Class class
37
38 #look at the eigenclasses
```

```
39 tyrion.singleton_class
40 cersei.singleton_class
41 GameOfThrones.singleton_class
42 # the eigenclass of tyrion and the seven
       are different
43 # each object has thus his eigenclass
44
45
46 # Now lets take a look at the
       eigenmethods
47 tyrion.singleton_methods
48 cersei.singleton_methods
49 GameOfThrones.singleton_methods
50 # we clearly see that only GameOfThrones
       has a singleton_method, the class
       method defined above
51
52 #look at the behaviour of the calls
53 GameOfThrones.whatAreYou
54 cersei.whatAreYou
55 tyrion.whatAreYou
56
57 #modify the behavior of the call for
       tyrion in his eigenclass
58 class << tyrion
59   def whatAreYou
60     "I am a dwarf, but my cunning makes
           me the better player"
61   end
62 end
63
64 #look at the behaviours
65 GameOfThrones.whatAreYou
66 cersei.whatAreYou
67 tyrion.whatAreYou
68 # the behaviour of the tyrion object
       changed, but not the behaviour of
       cersei
69
70 # And if we look again at the
       singleton_methods, we can see that
       tyrion has a specialized eigenmethod
       now
71 tyrion.singleton_methods
72 cersei.singleton_methods
73 GameOfThrones.singleton_methods
```