

Comparison of two object-oriented languages: Eiffel and Ruby

Quentin De Clerck

January 13, 2014

Contents

1	Foreword	1
2	General OO concepts	2
2.1	Everything is an object . . .	2
2.2	Access Control	2
2.3	Inheritance	3
2.4	Polymorphism	4
3	Language-specific Features	5
3.1	Eiffel	5
3.1.1	Design by Contract .	5
3.2	Void-safety	6
3.3	Ruby	7
3.3.1	Open Classes	7
3.3.2	Metaclass model . .	8
4	Conclusion	9
5	References	9
6	Code listings	10

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 [10].

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[9] 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-classes. 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, 7] 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[6]:

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.

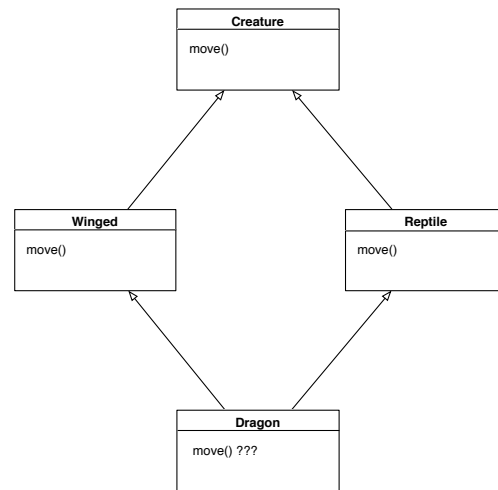


Figure 1: Example of the diamond problem

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* [6, 11] 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*[8]. 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 be 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 to 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 of 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: These 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* [5].

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 Metaclass model

A metaclass 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.

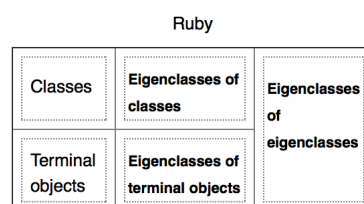


Figure 2: The Ruby metaclass model. Image from Wikipedia

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

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. Monkey patching— Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Monkey_patching, December 2013. Online; Accessed 13-January-2014.
- [6] n.d. Ruby Programming/Syntax/Classes— Wikibooks. http://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Classes, May 2013. Online; Accessed 13-January-2014.
- [7] n.d. Uniform access principle— Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Uniform_access_principle, December 2013. Online; Accessed 13-January-2014.
- [8] n.d. Duck typing— Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Duck_typing, January 2014. Online; Accessed 13-January-2014.
- [9] ruby doc.org. Basicobject class. <http://ruby-doc.org/core-2.0/BasicObject.html>. Online; Accessed 13-January-2014.
- [10] ruby lang.org. About ruby. <http://www.ruby-lang.org/en/about/>. Online; Accessed 13-January-2014.
- [11] David Thomas and Andrew Hunt. Programming ruby. *Dr. Dobb's Journal*, 5, 2001.

6 Code listings

Listing 1: Access Control in Ruby

```

1# CODE FOR ACCESS IDENTIFIERS
2
3class Lord
4
5  private
6
7  def plot
8    puts "I_plot_to_behead_king_Joffrey"
9  end
10
11  protected
12
13  def mistrust
14    puts "I_want_to_conspire,_but_hold_it_secret"
15  end
16
17
18  public
19
20  def toad
21    puts "You_are_such_a_magnificent_person,_my_grace"
22  end
23
24  def publicTalk
25    toad
26    self.toad
27  end
28
29  def protectedTalk
30    mistrust # works
31    self.mistrust #works
32  end
33
34
35  def privateTalk
36    plot #works
37    self.plot #does not work
38  end
39
40
41end
42
43l = Lord.new
44l.publicTalk
45l.protectedTalk
46l.privateTalk
47
48class Lord
49  public
50
51  def plot
52    puts "I_say_it_publicly:_I_want_to_behead_king_Joffrey!"
53  end
54end

```

```

55
56 l.privateTalk

```

Listing 2: Access Control in Eiffel

```

1  -----
2  -- CLASS LORD --
3  -----
4
5 note
6   description: "LORD_diplomacy_class."
7
8 class
9   LORD
10
11 create
12   make
13
14 feature {ANY} -- public
15
16   name: STRING assign set_name --
17     assigner command
18
19   set_name (n : STRING)
20     do
21       name := n
22     end
23 feature {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
33 feature {NONE} -- private, will not be
34   called outside this scope
35
36   plot
37     do
38       print("I plot to behead king_
39         Joffrey%N")
40     end
41
42 feature -- public, syntactic sugar for
43   feature {ANY}
44
45   toad
46     do
47       print("You are such a magnificent_
48         person, my grace%N")
49     end
50 feature {LORD} -- public for specified
51   classes and subclasses, same as
52   protected in C++ for example

```

```

51
52 mistrust
53   do
54     print("I want to conspire, but hold
55       it secret%N")
56   end
57 feature {LIEGELORD} -- public for
58   specified classes and subclasses,
59   will only work in LIEGELORD class and
60   subclasses
61
62   allegiance (n : STRING)
63     do
64       print("I am your humble subject, my
65         lord.")
66       print(n)
67       print("%N")
68     end
69 end
70 -----
71 -- CLASS LIEGELORD --
72 -----
73 note
74   description: "LIEGELORD_diplomacy_class
75     ."
76
77 class
78   LIEGELORD
79
80 inherit
81   LORD
82
83 create
84   makeLiege
85
86 feature
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
96         the scope of LIEGELORD
97         subclasses
98       print("Yes you are, lord.")
99       print(subject.name)
100       print("%N")
101       man.mistrust -- works within the
102         scope of LORD subclasses
103       -- man.plot does not work
104     end
105 end

```

```

104
105-----
106-- ROOT CLASS --
107-----
108
109note
110  description : "Eiffel-project_
111              application_root_class"
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",
128                             lord)
129
130      lord.toad
131      liege.toad
132    end
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_
11       legs!!"
12 end
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 "Rooooooooooooh!"
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_
7              CREATURE."
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_
25              CREATURE."
26class
27  REPTILE
28
29inherit
30  CREATURE
31
32feature
33  move
34    do
35      print ("creature_crawls_on_the_
36            ground")
37      print ("%N")
38    end
39end
40
41-----
42-- CLASS WINGED --
43-----

```

```

44
45 note
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
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_
24      polymorphism."
25
26  class
27    SELLSWORD
28
29  inherit
30    WARRIOR
31
32  create
33    make
34
35  feature
36
37    name: STRING
38
39    make (n : STRING)
40      do
41        name := n
42      end
43
44    fight
45      do
46        print(name)
47        print("_asks: _How_much_are_you_
48          willing_to_pay??%N")
49      end
50
51  end
52
53  _____
54  — CLASS KNIGHT —
55  _____
56
57  note
58    description: "KNIGHT_subclass_for_
59      polymorphism."
60
61  class
62    KNIGHT
63
64  inherit
65    WARRIOR

```

```

61  WARRIOR
62
63  create
64    make
65
66  feature
67
68    name: STRING
69
70    make (n : STRING)
71      do
72        name := n
73      end
74
75    fight
76      do
77        print(name)
78        print("_shouts: _FOR_THE_RIGHTFUL_
79          QUEEN!!%N")
80      end
81
82  end
83
84  — ROOT CLASS —
85  _____
86
87  class
88    APPLICATION
89
90  create
91    make
92
93  feature {NONE} — Initialization
94
95    defendQueen (warrior: WARRIOR)
96      do
97        warrior.fight
98      end
99
100    make
101      — Run application.
102      local
103        bronn: SELLSWORD
104        barristan: KNIGHT
105
106      do
107        create bronn.make ("Bronn")
108        create barristan.make ("Barristan")
109        defendQueen(bronn)
110        defendQueen(barristan)
111      end
112  end

```

Listing 7: Design by Contract in Ruby

```

1  module Contract
2    attr_accessor :old
3    @invariants = []
4
5    # checks if attribute has to be stored
6    def eligible attribute

```

```

7   nonEligible = ["@old", "@invariants"]
8   nonEligible.collect do |var|
9     if var == attribute
10      return false
11    end
12  end
13  return true
14 end
15
16 # Returns an older value for the given
   attribute
17 def getOld oldAttr
18   return @old["#{oldAttr}"]
19 end
20
21 # Stores the values of the attributes
22 def setOld(bind)
23   @old = Hash.new
24   vars = self.instance_variables
25   vars.collect do |v|
26     if eligible("#{v}")
27       @old["#{v}"] = eval("#{v}", bind)
28     end
29   end
30 end
31
32 # Returns to previous state (old
   attribute state)
33 def callback
34   vars = self.instance_variables
35   vars.collect do |v|
36     if eligible("#{v}")
37       val = @old["#{v}"]
38       eval("#{v} = #{val}")
39     end
40   end
41 end
42
43 # check if the conditions are
   maintained
44 def checkCond(cond, bind)
45   cond.collect do |c|
46     if !eval(c, bind)
47       raise "precondition_violation"
48     end
49   end
50 end
51
52 # Require clauses
53 def requir(cond, bind)
54   setOld(bind)
55   checkCond(cond, bind)
56 end
57
58 # Ensure clauses
59 def ensur(cond, bind, retVal = nil)
60   begin
61     checkCond(cond, bind)
62   rescue
63     callback
64     raise "postcondition_violation"
65   end

```

```

66   checkInvariants bind
67   return retVal
68 end
69
70 # Class invariant clauses
71 def classInvariant(inv, bind)
72   setOld(bind)
73   @invariants = inv
74   checkInvariants bind
75 end
76
77 #check the class invariant clauses
78 def checkInvariants bind
79   begin
80     checkCond(@invariants, bind)
81   rescue
82     callback
83     raise "class_invariance_violation"
84   end
85 end
86 end
87
88 # Example how it could be used in a class
89 class Winterfell
90
91   attr_accessor :maxStarks
92   attr_accessor :minStarks
93   attr_accessor :starks
94
95   include Contract
96
97   def initialize
98     @maxStarks = 7
99     @minStarks = 1
100    @starks = 7
101
102    #define the class invariants
103    classInvariant(['@starks <=
      @maxStarks',
104                  '@starks >= @minStarks'], binding)
105  end
106
107  def starksLeave amount
108    requir(['amount > 0', 'amount < 5'],
109          binding) # require clause
110
111    @starks = @starks - amount
112
113    ensur(['@starks == (getOld(:@starks) -
      amount)'], binding, @starks) #
      ensure clause
114  end
115
116 end
117
118
119 w = Winterfell.new
120 w.starksLeave 1
121 w.starksLeave 7 # invariance violation
122 w.starksLeave 0 # precondition violation
123 w.starksLeave 5 # precondition violation

```

Listing 8: Design by Contract in Eiffel

```

1  -----
2  -- CLASS WINTERFELL --
3  -----
4
5  note
6    description: "CLASS representing the
7                  castle of Winterfell. There should
8                  always be a Stark in Winterfell"
9
10
11  class
12    WINTERFELL
13
14  create
15
16    build
17
18  feature
19
20    number_of_starks : INTEGER
21    min_starks : INTEGER
22    max_starks : INTEGER
23
24    build
25      do
26        number_of_starks := 7
27        min_starks := 1
28        max_starks := 7
29        starks_present
30      end
31
32    starks_present
33      do
34        print(number_of_starks)
35        print("Starks are present in
36              Winterfell\n")
37      end
38
39    starks_leaving_winterfell (amount:
40      INTEGER)
41      require -- precondition
42        non_negative: amount > 0
43      do
44        number_of_starks :=
45          number_of_starks - amount
46        starks_present
47      ensure -- postcondition
48        leaved: number_of_starks = old
49          number_of_starks - amount
50      end
51
52    starks_entering_winterfell (amount:
53      INTEGER)
54      require
55        non_negative: amount > 0
56      do
57        number_of_starks :=
58          number_of_starks + amount
59        starks_present
60      ensure

```

```

52      entered: number_of_starks = old
53        number_of_starks + amount
54      end
55
56  invariant -- class invariant
57    always_a_stark: number_of_starks >=
58      min_starks --with tag
59    number_of_starks <= max_starks --
60      without tag
61  end
62
63  -----
64  -- ROOT CLASS --
65  -----
66
67  class
68    HELLO
69
70  create
71    make
72
73  feature {NONE} -- Initialization
74
75    make
76      -- Run application.
77      local
78        winterfell: WINTERFELL
79      do
80        create winterfell.build
81        winterfell.
82          starks_leaving_winterfell (3)
83          --winterfell.
84          starks_leaving_winterfell (4)
85          raises contract violation
86        winterfell.
87          starks_entering_winterfell (2)
88      end
89
90  end
91
92  end

```

Listing 9: Open classes in Ruby

```

1  class Dragon
2
3    def breatheFire
4      puts "It is still a baby dragon, it
5          cannot breathe fire"
6    end
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

```



```

18end
19
20dragon.fly
21dragon.breatheFire
22
23#Modify the breatheFire method
24class Dragon
25
26  alias babyFire breatheFire # renames
    previous method
27  def breatheFire
28    puts "Fire_is_everywhere"
29  end
30end
31
32dragon.fly
33dragon.babyFire
34dragon.breatheFire

```

Listing 10: Metamodel in Ruby

```

1class 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
23end
24
25#init the objects
26tyrion = GameOfThrones.new
27tyrion.numberOfPlayers # 1
28cersei = GameOfThrones.new
29cersei.numberOfPlayers # 2
30cersei.numberOfPlayers # 2
31
32
33#look at the classes
34tyrion.class # class GameOfThrones
35cersei.class # class GameOfThrones
36GameOfThrones.class # class of a class is
      the Class class
37
38#look at the eigenclasses

```

```

39tyrion.singleton_class
40cersei.singleton_class
41GameOfThrones.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
47tyrion.singleton_methods
48cersei.singleton_methods
49GameOfThrones.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
53GameOfThrones.whatAreYou
54cersei.whatAreYou
55tyrion.whatAreYou
56
57#modify the behavior of the call for
    tyrion in his eigenclass
58class << tyrion
59  def whatAreYou
60    "I_am_a_dwarf,_but_my_cunning_makes_
        me_the_better_player"
61  end
62end
63
64#look at the behaviours
65GameOfThrones.whatAreYou
66cersei.whatAreYou
67tyrion.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
71tyrion.singleton_methods
72cersei.singleton_methods
73GameOfThrones.singleton_methods

```