Comparison of two object-oriented languages: Eiffel and Ruby

Quentin De Clerck

January 12, 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 similar possibilities as multiple inheritance.

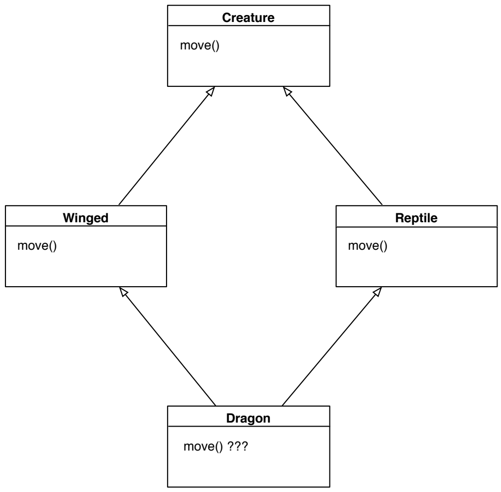

Figure 1: Example of the diamond problem

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* 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*. 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. 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.

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*. 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 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 known for being very object-oriented and for its simplicity. It is a programming language that is 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.

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. If there are different patches for the same method it is possible that they enter in conflict and one will override the other. It is possible to save the implementation of a method 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 patch and does not understand why the code does not act like expected.

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 to not abuse of this feature. Monkey-patching should only be used on third-party software if other concepts like inheritance fail.

3.3.2 Meta-class model

4 Conclusion

5 References

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

  mistrust
    do
      print("I want to conspire, but hold it secret%N")
    end

feature {LIEGELORD} -- public for specified classes and subclasses, will only work in LIEGELORD class and subclasses

  allegiance (n : STRING)
    do
      print("I am your humble subject, my lord ")
      print(n)
      print("%N")
    end

end

------------------------------
-- CLASS LIEGELORD --
------------------------------

note
  description: "LIEGELORD diplomacy class."

class
  LIEGELORD

inherit
  LORD

create

  makeLiege

feature

  subject : LORD

feature {NONE} -- Initialization

  makeLiege (n: STRING man : LORD)
    do
      name := n
      subject := man
      man.allegiance (n) -- works within the scope of LIEGELORD subclasses
      print("Yes you are, lord ")
      print(subject.name)
      print("%N")
      man.mistrust -- works within the scope of LORD subclasses
      -- man.plot does not work
    end

end
```

```
-------------------
-- ROOT CLASS --
-------------------

note
  description : "Eiffel-project application root class"

class
  APPLICATION

create
  make

feature {NONE} -- Initialization

  make
      -- Run application.
    local
      lord: LORD
      liege: LIEGELORD
    do
      create lord.make ("Karstark")
      create liege.makeliege ("Stark", lord)

      lord.toad
      liege.toad
    end

end
```

Listing 3: Inheritance in Ruby

```
# CODE FOR INHERITANCE

class Creature
  def initialize name
    @name = name
    puts "Creature #{name}"
  end

  def move
    puts "AAArg!! cannot move without legs!!"
  end
end

module Winged

  def fly
    puts "Flying creature"
  end
end

module Reptile
  def move
    puts "Crawling creature"
  end
end

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

```

-----------------------
-- CLASS CREATURE --
-----------------------

note
  description: "A class modeling a mythic CREATURE."

deferred class
  CREATURE

feature

  move
    deferred

    end
end

----------------------
-- CLASS REPTILE --
----------------------

note
  description: "REPTILE inheriting from CREATURE."

class
  REPTILE

inherit
  CREATURE

feature
  move
    do
      print ("creature crawls on the ground")
        print ("%N")
    end

end

---------------------
-- CLASS WINGED --
---------------------
```

```

note
  description: "WINGED inheriting from CREATURE."

class
  WINGED


inherit
  CREATURE

feature
  move
    do
      print ("creature flies in the air")
      print ("%N")
    end

end

--------------------
-- CLASS DRAGON --
--------------------

note
  description: "DRAGON multiple inheritance from diamond problem example."

class
  DRAGON

inherit
  WINGED
    rename
      move as fly
      select fly
    end
  REPTILE

create
    make

feature

  name: STRING

feature -- Initialization

    make (dragon_name: STRING)

        do
            name := dragon_name
            print (name)
            print ("%N")
        end

end

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

note
  description : "Eiffel-project application root class"

class
  APPLICATION

create
  make

feature {NONE} -- Initialization

  make
      -- Run application.
    local
      dragon: DRAGON
    do
      create dragon.make ("Balerion")
      dragon.fly
      dragon.move
    end
end
```

Listing 5: Polymorphism in Ruby

```ruby
class Knight
  def initialize name
    @name = "ser " + name
  end

  def fight
    puts "#{@name} shouts: FOR THE RIGHTFUL QUEEN!!"
  end
end

class Sellsword
  def initialize name
    @name = name
  end

  def fight
    puts "#{@name} asks: How much are you willing to pay??"
  end
end

def defendQueen knight
  knight.fight
end

barristan = Knight.new "Barristan"
bronn = Sellsword.new "Bronn"
defendQueen barristan
defendQueen bronn
```

Listing 6: Polymorphism in Eiffel

```
--------------------
```

```
-- CLASS WARRIOR --
------------------------

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
```

Listing 7: Design by Contract in Ruby

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

  # checks if attribute has to be stored
  def eligible attribute
```

```ruby
    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
      entered: number_of_starks = old number_of_starks + amount
    end

invariant -- class invariant
  always_a_stark: number_of_starks >= min_starks --with tag
  number_of_starks <= max_starks -- whitout tag
end


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

class
  HELLO

create
  make

feature {NONE} -- Initialization

  make
      -- Run application.
    local
      winterfell: WINTERFELL

    do
      create winterfell.build
      winterfell.starks_leaving_winterfell (3)
      --winterfell.starks_leaving_winterfell (4) raises contract violation
      winterfell.starks_entering_winterfell (2)
    end

end
```

Listing 9: Open classes in Ruby

```
class Dragon

  def breatheFire
    puts "It is still a baby dragon, it cannot breathe fire"
  end

end

dragon = Dragon.new
dragon.breatheFire

#Add a method fly
class Dragon

  def fly
    puts "It is flying high into the sky"
  end
```

```
end

dragon.fly
dragon.breatheFire

#Modify the breatheFire method
class Dragon

  alias babyFire breatheFire # renames previous method
  def breatheFire
    puts "Fire is everywhere"
  end
end

dragon.fly
dragon.babyFire
dragon.breatheFire
```