Learning Object-Oriented Programming, Design and TDD with Pharo

Stéphane Ducasse

March 11, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Variations around Isograms

This chapter propose some little challenges around words and sentences as a way to explore Pharo collections.

1.1 A dictionary-based solution

To do blbl

```
String >> isIsogramDictionaryImplementation
  "'ALTRUISME' isIsogramDictionaryImplementation"
  | letters |
  letters := Dictionary new.
  self do: [ :aChar |
      letters at: aChar
        ifPresent: [^ false]
        ifAbsent: [ letters at: aChar put: 1].
      ].
  ^ true
```

1.2 A Bag-based solution

To do blbl

```
String >> isIsogramUsingBagImplementation
  "'ALTRUISME' isIsogramUsingBagImplementation"

  | bag |
  bag := Bag new.
  self do: [ :each |
      bag add: each.
      ].
  ^ bag sortedCounts first key = 1
```

1.3 A solution using the String API

To do blbl

```
String >> isIsogramFatestImplementation
  "'ALTRUISME' isIsogramFatestImplementation"
  1 to: self size -1 do: [ :ix |
  (self
    findString: (self at: ix) asString
    startingAt: ix +1
    caseSensitive: false) ~~ 0 ifTrue: [ ^ false ]
  ].
  ^ true
```

1.4 A recursive approach

Up until now we only used iterations to go from one letter to the others. In Pharo you can also define the computation in recursive manner.

Before looking at a solution, imagine how you would do it. For example we could define a method that iterates over the string by removing the first element of the string and pass it as argument to an auxialliary method whose goal is to check whether the receiver includes or not the element and invoke the first method when it is necessary to continue the iteration.

For example, the method `isIsogramRecursiveImplementation` can invoke the method `isIsogramContainCharacter:` on a subtring and the method `isIsogramContainCharacter:` can check whether its receiver contains the element and else it invokes `isIsogramRecursiveImplementation` to continue the exploration.

```
String >> isIsogramRecursiveImplementation
  ...

String >> isIsogramContainCharacter: aCharacter
  ...
```

Here is a possible implementation.

```
String >> isIsogramRecursiveImplementation
  "'ALTRUISME' isIsogramRecursiveImplementation"
  "'ALTRUISMEA' isIsogramRecursiveImplementation"
  ^ self isEmpty
    ifTrue: [ true ]
    ifFalse: [ self allButFirst isIsogramContainCharacter: self
    first ]
```

```
String >> isIsogramContainCharacter: aCharacter

  ^ (self includes: aCharacter)
      ifTrue: [ false ]
      ifFalse: [ self isIsogramRecursiveImplementation ]
```

Such solution is a solution that comes naturally when using functional languages such as Camel or Scheme. In Pharo it would be more adapted to data structures such as linked lists because it is cheap to get a structure representing the elements except the first one (basically on linked list is just the next element since it is also a linked list.). In the case of strings or arrays, `allButFirst` has to build a new string or array and this is costly.

1.5 A low-level solution

```
isIsogramBitImplementation
   | i |
   i := 0.
   self asLowercase do: [ :char |
         | val |
         val := (char asInteger - 96).
         (val between: 1 and: 26) ifFalse: [ ^ false ].
         (i bitAt: val ) == 1 ifTrue: [ ^ false ].
         i := (i bitAt: val put: 1).
         ].
     ^ true
```

An interesting observation here is that if #asLowercase is moved to each character instead “val := (char asLowercase asInteger - 96).” Then it leads to a 4-5x performance loss .

1.6 Are all solutions finding the same?

```
  | lines bits dicts  bags strings sets |
  lines := (ZnDefaultCharacterEncoder
    value: ZnCharacterEncoder latin1
    during: [
      ZnClient new
        get:
    'http://www.pallier.org/ressources/dicofr/liste.de.mots.francais.frgut.txt'
    ]) lines.

  bits := lines select: #isIsogramBitImplementation.
  dicts := lines select: #isIsogramDictionaryImplementation.
  bags := lines select: #isIsogramBagImplementation.
```

```
strings := lines select: #isIsogramStringImplementation.
sets := lines select: #isIsogramSetImplementation.
recurs := lines select: #isIsogramRecursiveImplementation.
{ bits . dicts .  bags . strings . sets} collect: #size. "#(23118
  47674 47674 47668 47674)"
```

1.7 Comparing solutions

Pharo proposes tools to measure execution speed (for example the message `timeToRun` and `millisecondsToRun:` shown below). When an operation is too fast you cannot measure well (Check the chapter on profiling of Deep Into Pharo), so you should execute multiple times the same expressions.

Here measuring an expression alone does not really help as you can see.

```
['PHRAO' isIsogramSetImplementation ] timeToRun.
>>> 0
Time millisecondsToRun: ['PHRAO' isIsogramSetImplementation ]
>>> 0
```

Pharo libraries also offers the bench method that gives the number of execution possible per second.

[[[[’ALTRUISME’ isIsogramSetImplementation] bench ’334,371 per second’

[’ALTRUISME’ isIsogramStringImplementation] bench ’546,823 per second’

[’ALTRUISME’ isIsogramUsingBagImplementation] bench ’142,432 per second’

[’ALTRUISME’ isIsogramDictionaryImplementation] bench ’147,276 per second’

[’ALTRUISME’ isIsogrambitImplementation] bench ’1,137,976 per second’

[’ALTRUISME’ isIsogramRecursiveImplementation] bench ”’487,492 per second”’]]]

Some observations and comments

- Size of code does not mean anything. Smaller code can be slower because they use more.

1.8 Handling french and the lesson behind

à, â, é, è, ê, ë, î, ï, ô, ù, û, ü, ÿ, ç, æ et œ

```
'ALTRUISME' isIsogramSetImplementation

'ALTRUISME' isIsogramStringImplementation

'ALTRUISME' isIsogramUsingBagImplementation

'ALTRUISME' isIsogramDictionaryImplementation

'ALTRUISME' isIsogramBit
```

1.9 So Watch out

You could find that it is worth to systematically use low-level messages. It is not a good strategy. Why? They are several reasons:

- You may spend a lot of time to find a strong optimised solution.
- You may spend a lot of time optimizing a part of the system that does not need to be optimized.
- Low-levels solutions are often more difficult to read and understand so if you need something slightly different they may break. For example what is we want to compute isogram in language where the letters are not contiguous.

Important A good engineering practice is: Make it work, make it right, make it fast and not the inverse

Variations

```
isIsogramBit
  "['ALTRUISME' isIsogramBit] bench '337,427 per second'"

   | i |
   i := 0.
   self do: [ :char |
         | val |
         val := (char asLowercase asInteger - 96).
         (val between: 1 and: 26) ifFalse: [ ^ false ].
         (i bitAt: val ) == 1 ifTrue: [ ^ false ].
         i bitAt: val put: 1
         ].
     ^ true
```

```
['ALTRUISME' isIsogramBit] bench '337,427 per second'
```

What you should see is that the execution engine may also support adaptatives optimisations, i.e., depending on the program it executes it can change it on the fly to gain speed. You may wonder why it is not always the case.

Why the compiler and execution engine (virtual machines) do not always optimise at the maximum, because optimisation takes time and space and that there is a tradeoff to reach. So virtual machines optimise aggressively code that is executed a lot because they see that it is worth doing it.

So now we will look at other kind of words or sentences.

Bibliography