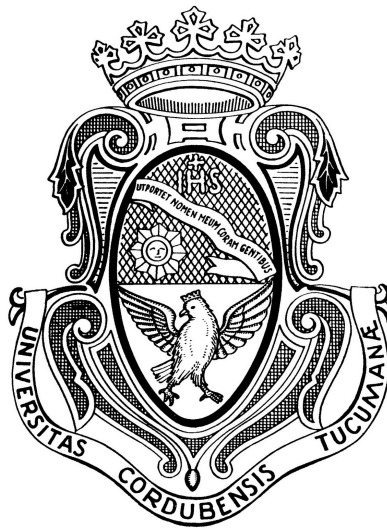


# UNIVERSIDAD NACIONAL DE CÓRDOBA

FACULTAD DE CIENCIAS EXACTAS  
FÍSICAS Y NATURALES



Paradigmas de Programación  
TRABAJO PRÁCTICO N°3: Pharo - Smalltalk



Alumno:

- Cazajous Miguel A. - 34980294

Córdoba - Argentina  
6 de diciembre de 2018

# Índice

<b>1. Introducción:</b>	<b>1</b>
<b>2. Enunciado:</b>	<b>1</b>
<b>3. Repaso:</b>	<b>2</b>
3.1. Clases y objetos: . . . . .	3
3.1.1. Cola: . . . . .	3
3.1.2. Buffer: . . . . .	5
3.1.3. BufferSalida: . . . . .	7
3.1.4. Paquete: . . . . .	10
3.1.5. BufferInterno: . . . . .	11
3.1.6. Red: . . . . .	13
3.1.7. Nodo: . . . . .	14
3.1.8. Vecino: . . . . .	16
3.1.9. Config: . . . . .	17
<b>4. Desarrollo:</b>	<b>18</b>
4.1. Modificaciones y nuevas implementaciones: . . . . .	18
4.2. Diagrama de Clases . . . . .	22
4.3. Pruebas: . . . . .	23
4.4. Guardar información en un archivo: . . . . .	29
4.5. Inicializar mediante un archivo: . . . . .	30
4.6. Compartir el código: . . . . .	36
4.7. Últimas modificaciones: . . . . .	37
<b>5. Anexo:</b>	<b>40</b>
5.1. Sintaxis básica: . . . . .	40
5.2. Patrón Singleton: . . . . .	42
5.3. Streams: . . . . .	43
5.3.1. Introducción: . . . . .	43
5.3.2. Stream para acceder a archivos: . . . . .	43
<b>Referencias</b>	<b>45</b>

## 1. Introducción:

Pharo es un lenguaje de tipado dinámico orientado puramente a objetos que utiliza como base el lenguaje de programación Smalltalk.

Pharo cuenta con un entorno de programación que contiene, en el, todas las herramientas para el desarrollo de aplicaciones.

Pharo utiliza una máquina virtual (similar a lo que ocurre con Java), una “imagen” que provee una “instantánea” del sistema Pharo y determina la versión de cada release.

Los cambios que se realicen en el entorno se guardan en “changes”, lo que, junto con la imagen del sistema proveen el entorno completo para cada usuario con clases y métodos que Pharo provee, como así también clases y métodos desarrollados por cada usuario.

## 2. Enunciado:

Tomando como base el archivo con lo hecho en clase 18/06, deber completar el funcionamiento del programa de simulación del tráfico.

Esto consiste en:

- Leer la configuración de los nodos, los enlaces y ancho de banda de cada enlace desde un archivo .txt.

ref: tutorial de pharo, cap de manejo de archivos (Streams).

- Definir las tablas de ruteo en cada nodo . En lugar de generarlas por el algoritmo de Dijkstra, definir las estáticas y leerlas desde el archivo de configuración (el del punto anterior).
- Implementar el buffer interno que contiene las páginas que tienen como destino ese nodo como una colección de diccionarios. Un diccionario por página.
- Crear páginas en cada nodo que tengan como destino otros nodos y enrutarlas
- Realizar el movimiento de paquetes, de a un nodo por vez, lo que constituye un ciclo, la cantidad de ciclos necesarios para que todos los paquetes lleguen a destino.
- Por cada ciclo de movimiento hacer un “dump” a un archivo .txt de los paquetes que hay en cada uno de los buffers de nodo.

### 3. Repaso:

Vamos a utilizar como base el código implementado en clase.

Para importar un archivo .st, dentro del entorno Pharo, buscar “File browser” en “tools”, abrir el archivo y presionar “install”.

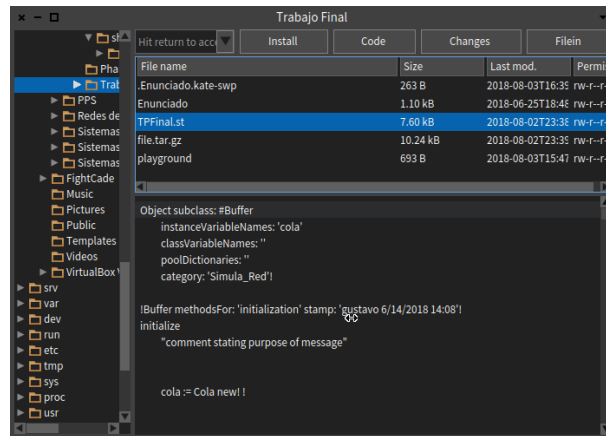


Figura 1: File Browser

En “System browser”, luego, podemos ver que tenemos un nuevo paquete. En el segundo cuadro desde la izquierda podemos observar las clases que se han creado hasta el momento.

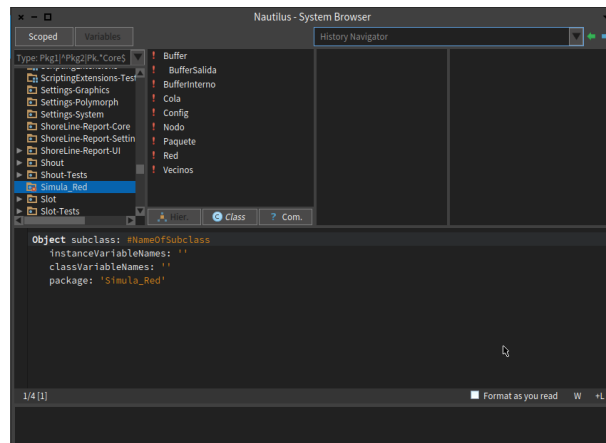


Figura 2: System browser

Para demostrar el funcionamiento de cada clase, se usará “Playground”, que permite la creación de los diferentes objetos, y “Transcript” que es una consola de salida, donde se permite ver mensajes.

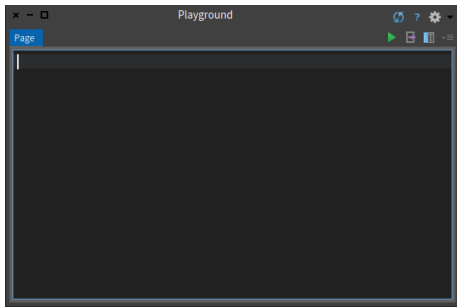


Figura 3: Playground

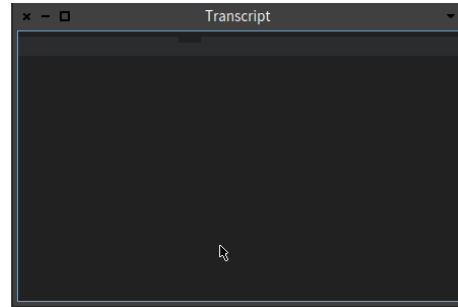


Figura 4: Transcript

### 3.1. Clases y objetos:

Se mostrará a continuación el comportamiento de los objetos para cada una de las clases.

#### 3.1.1. Cola:

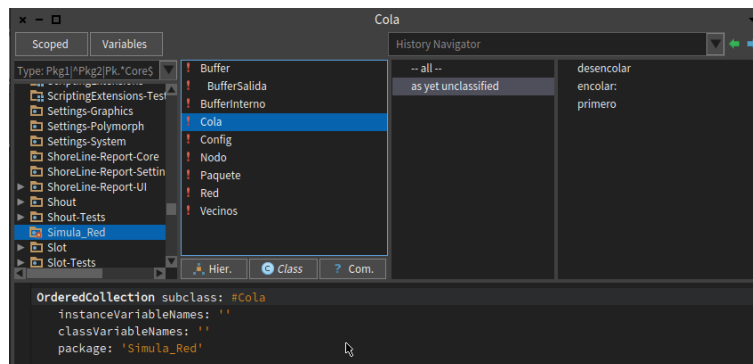


Figura 5: Clase Cola

#### Métodos (Mensajes):

- “primero”: Devuelve el primer valor que ingreso en la cola.
- “encolar”: Agrega al final de la cola un nuevo elemento.

- “desencolar”: Obtiene el primero valor, lo guarda en una variable temporal, remueve el valor de la cola y devuelve el valor almacenado en la variable temporal.

```
desencolar
"saca el primer elemento de la cola"

| temporary |
temporary := self primero.
self removeFirst.
^ temporary
```

Figura 6: desencolar

### Ejecuciones:

#### Secuencia de instrucciones:

1. cola := Cola new.
2. Transcript show: (cola encolar: 10);cr.
3. Transcript show: (cola encolar: 11);cr.
4. Transcript show: cola primero;cr.
5. Transcript show: cola desencolar;cr.
6. Transcript show: cola.

Vemos la salida que se muestra en Transcript

```
a Cola(10)
a Cola(10 11)
10
10
a Cola(11)
```

Figura 7: Salida por Transcript

Los dos valores consecutivos iguales a “10” representan el primer valor y el valor obtenido luego de desencolar. Se pueden pasar cualquier tipo de objetos a la cola, eso dependerá de donde se utilice una instancia de la clase “Cola”.

### 3.1.2. Buffer:

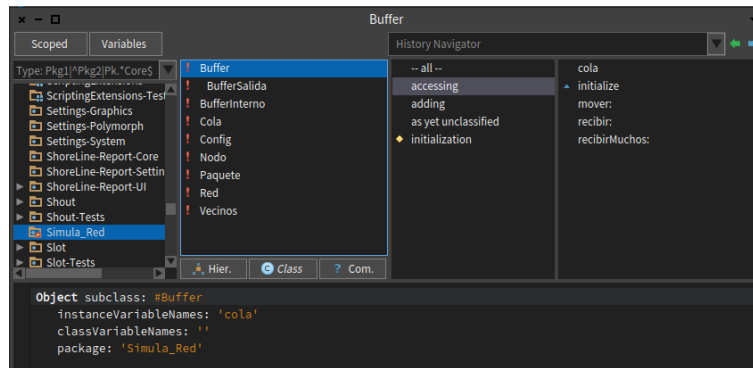


Figura 8: Clase Buffer

#### Métodos (Mensajes):

- “cola”: Devuelve una Cola (variable de instancia).
- “initialize”: Inicializa la variable de instancia “cola”.
- “recibir:”: Recibe un objeto y lo encola en “cola”.
- “mover:”: Recibe un “buffer”, el cual recibe “recibir” el valor desencolado de “cola”.

```
mover: aBuffer
    aBuffer recibir: (cola desencolar) .
```

Figura 9: primero

- “recibirMuchos:”: Comportamiento similar a “recibir” pero considerando como entrada una OrderedCollection en lugar de un simple valor.


```
recibirMuchos: aCollection
    "comment stating purpose of message"
    aCollection do: [ :anObj | self recibir: anObj ]
```

Figura 10: recibirMuchos

**Ejecuciones:**

1. a := Buffer new.
2. b := Buffer new.
3. a recibir: 1.
4. a recibirMuchos: #(2 3 4 5).
5. Transcript show: a cola;cr.
6. Transcript show: b cola;cr.
7. a mover: b.
8. Transcript show: a cola;cr.
9. Transcript show: b cola;cr.

La salida por Transcript



```
a Cola(1 2 3 4 5)
a Cola()
a Cola(2 3 4 5)
a Cola(1)
```

Figura 11: Salida Transcript

Creamos dos colas (a y b) y luego guardamos un simple valor (1) seguido de una colección de valores (2 3 4 5) usando “recibir:” y “recibirMuchos:” respectivamente.

Al usar “mover” vemos que se desencola el valor 1 de la cola asociada a buffer “a” y se guarda en la cola asociada al buffer “b”.



### 3.1.3. BufferSalida:

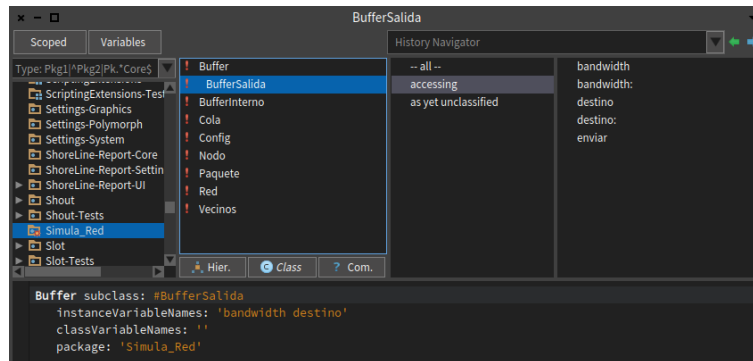


Figura 12: Clase BufferSalida

### Métodos (Mensajes):

- “bandwidth”: Retorna la variable de instancia “bandwidth”.
- “bandwidth:”: Setea la variable de instancia “bandwidth”.
- “destino”: Retorna la variable de instancia “destino”.
- “destino:”: Setea la variable de instancia “destino”.
- “enviar”:

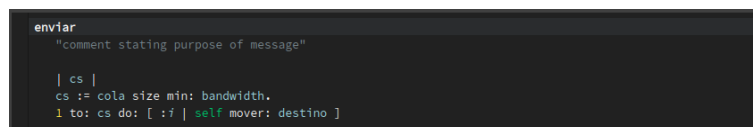


Figura 13: Clase BufferSalida

### Ejecuciones:

“BufferSalida” hereda de la clase “Buffer” todos sus métodos y la variable de instancia “cola”, y además implementa: “destino”, “destino:”, “bandwith”, “bandwith:” y “enviar”.

bandwidth=10

1. a := BufferSalida new.
2. b := Buffer new.
3. Transcript show: (a cola);cr.
4. a recibir: 1.
5. a recibirMuchos: #(2 2 8 48 3 4 9 11 22).
6. a bandwidth: 10.
7. Transcript show: (a bandwidth);cr.
8. a destino: b.
9. Transcript show: (a destino);cr.
10. a enviar.
11. Transcript show: (b cola);cr.
12. Transcript show: (a cola);cr.

bandwidth=5

1. a := BufferSalida new.
2. b := Buffer new.
3. Transcript show: (a cola);cr.
4. a recibir: 1.
5. a recibirMuchos: #(2 2 8 48 3 4 9 11 22).
6. a bandwidth: 5.
7. Transcript show: (a bandwidth);cr.
8. a destino: b.
9. Transcript show: (a destino);cr.
10. a enviar.
11. Transcript show: (b cola);cr.
12. Transcript show: (a cola);cr.

```
a Cola()
a Cola(1 2 2 8 48 3 4 9 11 22)
10
a Buffer
a Cola(1 2 2 8 48 3 4 9 11 22)
a Cola()

a Cola()
a Cola(1 2 2 8 48 3 4 9 11 22)
5
a Buffer
a Cola(1 2 2 8 48)
a Cola(3 4 9 11 22)
```

Figura 14: Salida por Transcript

Se muestran aquí las salidas por Transcript para valores diferentes de ancho de banda (bandwidth).

Primero creamos dos objetos, uno es una instancia de “BufferSalida” y el otro una instancia de “Buffer”.

Mostramos la cola de “a” antes y después de cargar todos los valores con “recibir” y “recibirMuchos”. Cargamos y mostramos el valor de ancho de banda (10 y luego 5) y el buffer de destino (a Buffer).

Finalmente utilizamos el método “enviar” de “a” y mostramos primero la cola en “b” y luego la cola en “a”.

El método enviar calcula el tamaño de la cola en “a”, compara ese valor con un ancho de banda y devuelve el menor. Al ser el tamaño de la cola igual a 9, primero devolverá 9, pues 9 es menor que 10 y luego devolverá 5. Luego itera desde 1 hasta el valor que devolvió anteriormente. De este modo primero envía todos los valores de la cola de “a” a la cola de “b” y después solo envía los 5 primeros valores, dejando los restantes en la cola de “a”.

### 3.1.4. Paquete:

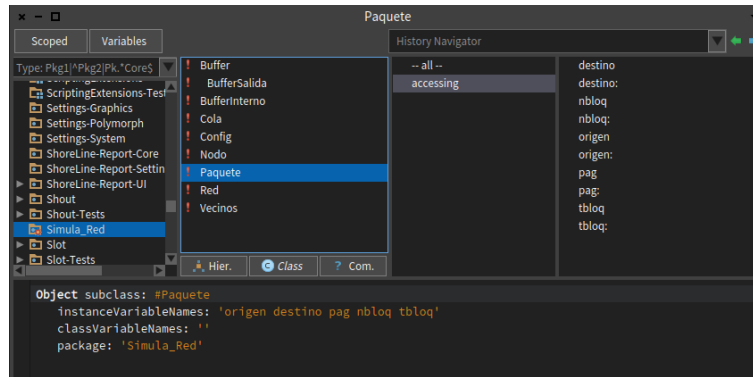


Figura 15: Clase Paquete

### Métodos (Mensajes):

Todos los métodos a continuación son métodos “get” y “set” para las variables de instancia “origen, destino, pag, nbloq, tbloq”.

- “destino”: get.
- “destino:”: set.
- “nbloq”: get.
- “nbloq:”: set.
- “origen”: get.
- “origen:”: set.
- “pag”: get.
- “pag:”: set.
- “tbloq”: get.
- “tbloq:”: set.

### Ejecuciones:

El comportamiento de una instancia no cambia en nada si se varían los tipos de objetos que se pasan a “set”, pero si influirán en el comportamiento global del programa.

### 3.1.5. BufferInterno:

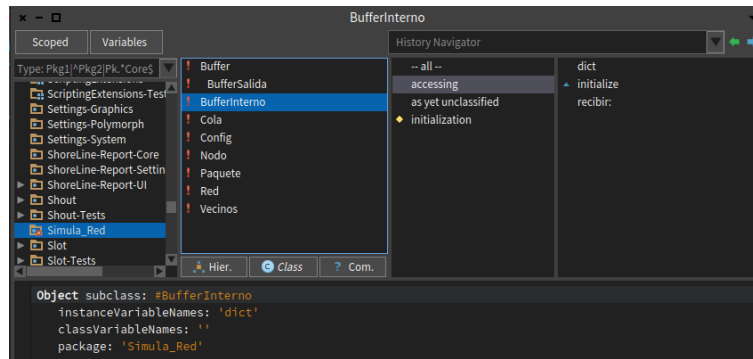


Figura 16: Clase BufferInterno

### Métodos (Mensajes):

- “dict”: Retorna “dict”.
- “initialize”: Inicialize la variable de instancia “dict”
- “recibir:”: Recibe un objeto “Paquete”.

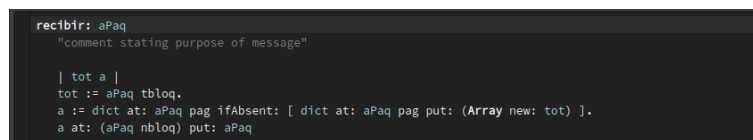


Figura 17: Clase BufferInterno

### Ejecuciones:

“BufferInterno” al contrario de lo que el nombre de clase pueda indicar, no hereda de Buffer.

Lo que hace el método “recibir” es: Obtener el tamaño del bloque total y guardarlo en una variable que servirá luego para crear un arreglo de ese tamaño.

Luego verifica si en la variable “dict” existe un valor en la llave que es igual a “aPaq pag”, es decir la página a la que pertenece el paquete. Si no existe crea en esa llave un arreglo del tamaño especificado por “aPaq tblog”. La última línea guarda el paquete en la llave correspondiente y en el lugar “aPaq nblog” del arreglo.

Veamos un ejemplo de ejecución:

1. paquete := Paquete new.
2. paquete destino: 1.
3. paquete nbloq: 2.
4. paquete origen: 3.
5. paquete pag: 4.
6. paquete tbloq: 5.
7. Transcript show: (tot := paquete tbloq);cr.
8. dict := Dictionary new.
9. Transcript show: (dict);cr.
10. a := dict at: paquete pag ifAbsent: [dict at: paquete pag put: (Array new: tot)].
11. Transcript show: (dict);cr.
12. a at: (paquete nbloq) put: paquete.
13. Transcript show: (dict);cr.

```
5
a Dictionary()
a Dictionary(4->#(nil nil nil nil nil) )
a Dictionary(4->an Array(nil a Paquete nil nil nil) )
```

Figura 18: Salida Transcript.

En las primeras líneas lo que hacemos es setear los campos de “paquete”. Luego las salidas por orden de aparición son: El tamaño total de bloque, la creación de un diccionario, la creación de la llave “paquete pag” y un arreglo de tamaño “paquete tbloq” como valor, y finalmente guardamos en esa llave en el lugar del arreglo correspondiente a “paquete nbloq” el paquete.

### 3.1.6. Red:

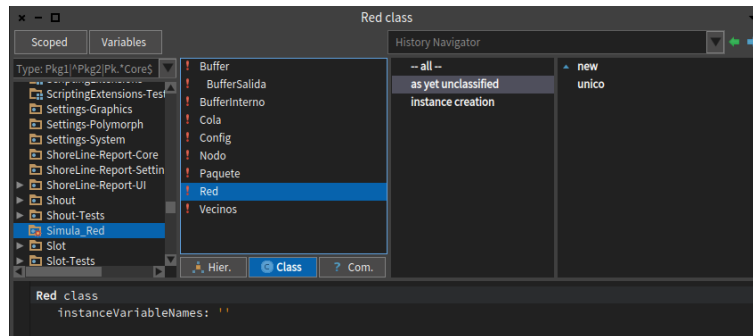


Figura 19: Clase Red

#### Métodos (Mensajes) de clase:

- “new”:



Figura 20: Clase Red

- “unico”:



Figura 21: Clase Red

Los métodos presentes en “Red” implementan el [patrón singleton](#)

## Ejecuciones:

Simplemente ejecutando:

Transcript show: (red := Red unico);cr.

Obtenemos.

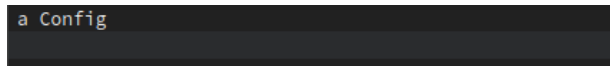


Figura 22: Salida Transcript

La salida anterior nos indica que se crea un objeto “Config”.

### 3.1.7. Nodo:

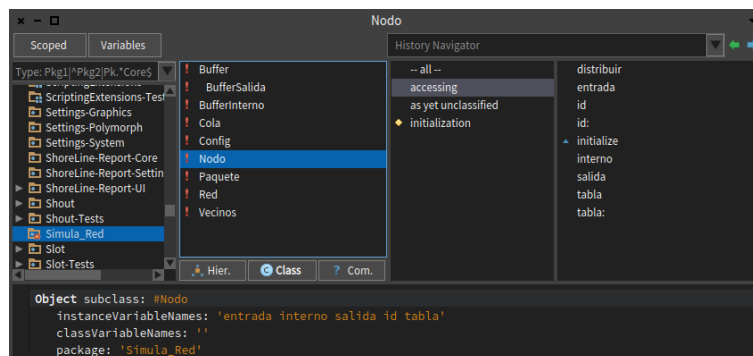


Figura 23: Clase Nodo

## Métodos (Mensajes):

- “distribuir”: La primera expresión entre corchetes seguido de “while-False” es un bucle que continua hasta que la cola esté vacía. Va descolando elemento a elemento de la cola de entrada, si el elemento descolado tiene un id igual al del nodo actual lo guarda en “interno”. De lo contrario guarda en una variable “v” el valor de “tabla” que corresponde a una llave igual a ese valor destino, y finalmente envía ese paquete a “v”.



```

distribuir
"identifica si el paq es para reenviar o queda local"

[ temp v |
  [ entrada cola isEmpty ]
  whileFalse: [ temp := entrada cola desencolar.
    temp destino = id
    ifTrue: [ interno recibir: temp ]
    ifFalse: [ Transcript show: temp destino.
      v := tabla at: temp destino.
      salida recibir: temp to: v ] ] ]

```

Figura 24: Clase Nodo

- “entrada”: Devuelve “entrada” de tipo Buffer.
- “id”: Devuelve “id”.
- “id.”: Setea la variable “id”
- “initialize”: Inicializa los campos.

```

initialize
"comment stating purpose of message"

entrada := Buffer new.
salida := Vecinos new.
tabla := Dictionary new.
interno := BufferInterno new.

```

Figura 25: Clase Nodo

- “interno”: Devuelve “interno” de tipo BufferInterno.
- “salida”: Devuelve “salida” de tipo Vecinos.
- “tabla”: Devuelve “tabla” de tipo Dictionary.
- “tabla.”: Setea tabla

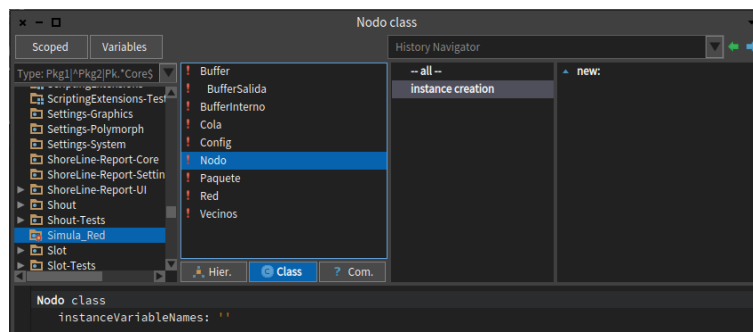


Figura 26: Clase Nodo

### Métodos (Mensajes) de clase:

- “new.”:

```
new: anId
"comment stating purpose of message"

| tem |
tem := self new.
tem id: anId .
^ tem.
```

Figura 27: Clase Nodo

Crea un nodo con un determinado “id”.

#### 3.1.8. Vecino:

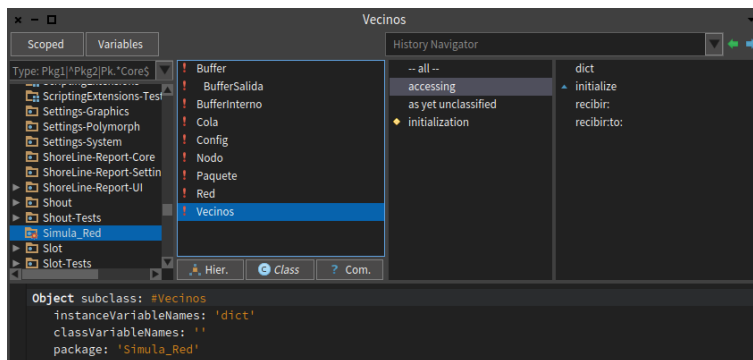


Figura 28: Clase Vecino

### Métodos (Mensajes):

- “dict”: Devuelve “dict”.
- “initialize”: Inicializa la variable “dict”.
- “recibir.”:

```
recibir: aPaq
"comment stating purpose of message"

| a |
a := dict at: aPaq pag ifAbsent: [ dict at: aPaq pag put: aPaq ].
a at: (aPaq nbloq) put: aPaq
```

Figura 29: Clase Vecino

- “recibir:to:”:

```
recibir: aPaq to:aVecino
"comment stating purpose of message"

(dict at: aVecino) recibir: aPaq .
```

Figura 30: Clase Vecino

Las clases “recibir:” y “recibir:to:” poseen errores que luego se verá como solucionarlos. La implementación puede estar incompleta.

### 3.1.9. Config:

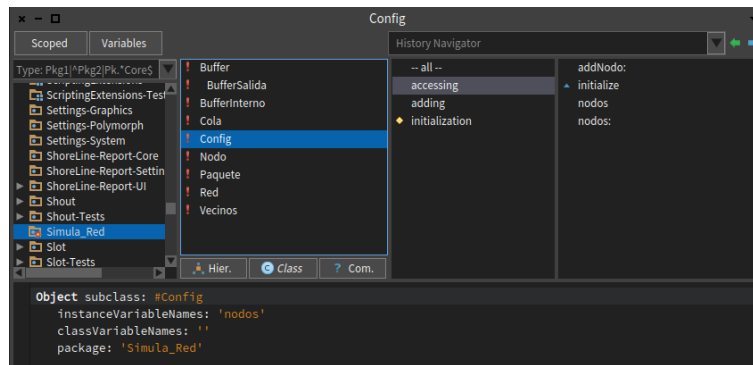


Figura 31: Clase Config

### Métodos (Mensajes):

- “addNodo:”: Agrega un “nodo” a “nodos”.
- “initialize”: Inicializa “nodos” de tipo “OrderedCollection”.
- “nodos”: Devuelve “nodos”.
- “nodos:”: Setea “nodos”.

## 4. Desarrollo:

### 4.1. Modificaciones y nuevas implementaciones:

A continuación se muestran algunos cambios que se han realizado en el programa para completar el comportamiento de una transmisión de paquetes en una red.

El primer cambio fue el agregado de una nueva clase llamada “Pagina” que contiene como variable de instancia una `OrderedCollection` de “Paquetes” para simplificar el llenado del buffer de entrada de cada nodo.

#### Clase Pagina:

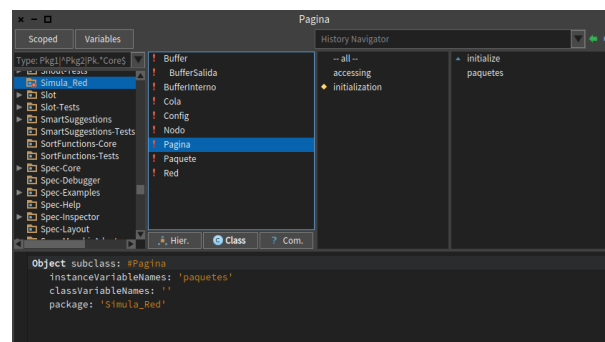


Figura 32: Clase Pagina

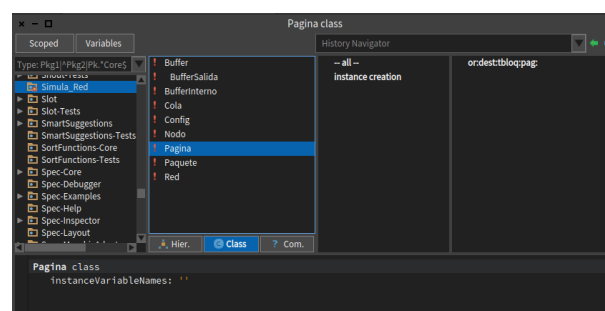


Figura 33: Clase Pagina

#### Métodos (Mensajes):

- “initialize”: Inicializa la variable de instancia “paquetes”.
- “paquetes”: Devuelve la variable de instancia.

- “or:dest:tbloq:pag.”: Es una sobreescritura al método “new” donde pasamos como argumentos los campos de un objeto “Paquete” para construir una OrderedCollection de estos. nbloq no es necesario, pues puede deducirse a partir de tbloq.

```
or: or dest: dest tbloq: tb pag: pag
"comment stating purpose of message"

| temp pags |
temp := self new.
pags := temp paquetes.
1 to: tb do: [ :each |
  pags add: Paquete new.
  (pags at: each) origen: or.
  (pags at: each) tbloq: tb.
  (pags at: each) nbloq: each.
  (pags at: each) destino: dest.
  (pags at: each) pag: pag ].
^ temp
```

Figura 34: Nuevo método

### Clase Nodo:

Se realizaron algunos cambios en algunos métodos de esta clase.

### Métodos (mensajes):

- “id:tam.”: Se modificó el método “new.” permitiéndole recibir otro argumento que es el tamaño de “salidas”, es decir la cantidad de buffers de salida.

```
id: anId tam: tam
"comment stating purpose of message"

| temp sals |
temp := self new.
temp id: anId.
sals := temp salidas.
1 to: tam do: [ :each | sals add: BufferSalida new ].
^ temp
```

Figura 35: Método modificado

- “distribuir:” Se modificó el método de la siguiente manera.

```
distribuir
  "identifica si el paq es para reenviar o queda local"
  | temp next_hop |
  next_hop := nil.
  [ entrada cola isEmpty ]
  whileFalse: [ temp := entrada cola desencolar.
    temp destino = id
    ifTrue: [ interno recibir: temp ]
    ifFalse: [ tabla
      at: temp destino
      ifPresent: [ next_hop := tabla at: temp destino.
        1 to: salidas size do: [ :each |
          (salidas at: each) prox_salto = next_hop
          ifTrue: [ (salidas at: each) recibir: temp ] ].
        tabla at: temp destino ifAbsent: [ Transcript show: 'No se puede llegar al destino' ] ] ]
    ]
  ]
```

Figura 36: Método modificado

En este método se realiza más o menos lo siguiente: Mientras la cola no esté vacía se verifica que el destino del paquete sea o no el nodo actual. Si lo es, se guarda en “interno”, de lo contrario verifica si es posible llegar a ese destino verificando la tabla de ruteo. De poder llegar comprueba el valor del diccionario “tabla” para identificar cual es el siguiente salto. El siguiente salto tiene un buffer de salida específico, al cual manda el paquete. Si no puede llegar lanza un mensaje de error por Transcript.

- “enviarTodo”: Al tener un objeto de la clase “BufferSalida” el campo “bandwidth” puede necesitarse más de un envío para completarse la transferencia, de este modo este método simplemente verifica que se vacíe la cola mediante un bucle.

```
enviarTodo: nodos
  "comment stating purpose of message"
  1 to: salidas size do:
    [ :each | [ (salidas at: each) cola isEmpty ] whileFalse: [ (salidas at: each) enviar: nodos ] ]
```

Figura 37: Nuevo método

### Clase BufferSalida:

Se modificó la variable de instancia “destino” a “prox\_salto” para más claridad.

- “prox\_salto:” Fue renombrado de “destino:” (setter)
- “prox\_salto” Fue renombrado de “destino” (getter)

- “enviar:” Ahora el método acepta un argumento que son el total de nodos guardados en una `OrderedCollection`, la cual se obtiene haciendo uso de “Config”.

```
enviar: nodos
"comment stating purpose of message"

| cs buffer nodo |
1 to: nodos size do: [ :each |
  nodo := nodos at: each.
  nodo id = prox_salto
    ifTrue: [ buffer := nodo entrada ] ].
cs := cola size min: bandwidth.
1 to: cs do: [ :i | self mover: buffer ]
```

Figura 38: Método modificado.

### Clase vecino:

La clase fue eliminada, la aproximación al comportamiento que esta clase indica fue reemplazada enviando la `OrderedCollection` de la totalidad de los nodos a cada nodo cuando se desea usar el método “enviarTodo”.

## 4.2. Diagrama de Clases

Habiendo hecho un repaso de las clases y estudio del comportamiento del programa hasta ahora, podemos representar todo en conjunto mediante un diagrama de clases, también considerando el agregado de nuevos métodos y clases. Vamos a especificar mejor las situaciones donde se mencione “anObject” por un tipo concreto para dar más coherencia al programa.

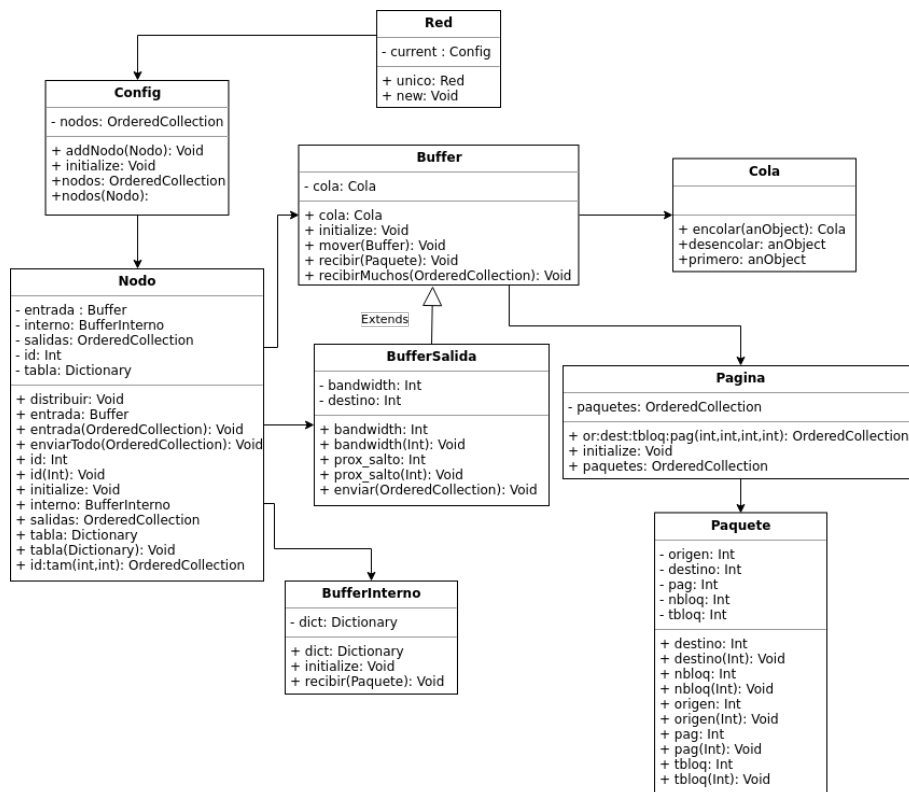


Figura 39: Diagrama de Clases



### 4.3. Pruebas:

Vamos a ver unas ejecuciones para ver como se comporta nuestro programa.

Planteamos una topología ejemplo como la siguiente:

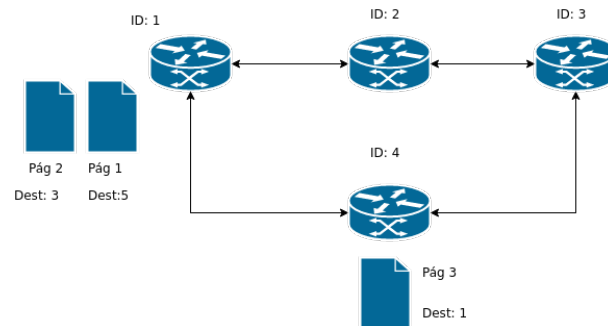


Figura 40: Topología de red.

Tenemos 4 nodos con IDs 1,2,3 y 4. El nodo 1 tiene dos páginas en su buffer de entrada, es decir una OrderedCollection de Paquetes, uno con destino inalcanzable (5) y el otro con destino al nodo 3. El nodo 4 posee un conjunto de paquetes con destino al nodo 1.

Podemos ver que cada nodo tiene dos interfaces. La cantidad de interfaces está relacionada con la cantidad de buffer de salidas, no así con el buffer de entrada, el cual siempre es único de acuerdo a esta implementación. Vamos a considerar también el campo “bandwidth” del buffer de salida, modificando “tbloq” para alguna de las páginas.

A continuación veamos como podemos ejecutar nuestra topología ejemplo.

### Secuencia de ejecución:

- Inicializamos los nodos con su respectivo id y con 2 buffers de salida cada uno, creamos una red y guardamos los nodos en la OrderedCollection (notar el llamado a “unico” debido al uso del patrón Singleton). Finalmente inicializamos las 5 páginas que habíamos mencionado, con origen, destino, tamaño de bloque y id de página.

```
nodo1 := Nodo id: 1 tam: 2.  
nodo2 := Nodo id: 2 tam: 2.  
nodo3 := Nodo id: 3 tam: 2.  
nodo4 := Nodo id: 4 tam: 2.  
  
red := Red unico.  
red addNodo: nodo1.  
red addNodo: nodo2.  
red addNodo: nodo3.  
red addNodo: nodo4.  
red nodos removeAll.  
Transcript show: (red nodos).  
  
pag1 := Pagina or: 1 dest: 5 tbloq: 5 pag:1.  
pag2 := Pagina or: 1 dest: 3 tbloq: 5 pag:2.  
pag3 := Pagina or: 4 dest: 1 tbloq: 10 pag:3.
```

Figura 41: Inicialización

Transcript nos muestra que se han agregado los 4 nodos creados.

```
an OrderedCollection(a Nodo a Nodo a Nodo a Nodo)
```

Figura 42: Salida Transcript

- En el primer bloque de código estamos asignando a cada una de los buffers de salida el próximo salto, es decir a que otro nodo se conectan directamente.

En segundo lugar para los mismos buffers de salida anteriores seteamos un ancho de banda al enlace con ese próximo nodo.

Finalmente a cada nodo le creamos una tabla de ruteo en donde se guardan como llave el destino, es decir todos los nodos restantes de la topología y como valor el próximo salto para llegar a ese destino.

```
(nodo1 salidas at: 1) prox_salto: 2.  
(nodo1 salidas at: 2) prox_salto: 4.  
(nodo2 salidas at: 1) prox_salto: 1.  
(nodo2 salidas at: 2) prox_salto: 3.  
(nodo3 salidas at: 1) prox_salto: 2.  
(nodo3 salidas at: 2) prox_salto: 4.  
(nodo4 salidas at: 1) prox_salto: 3.  
(nodo4 salidas at: 2) prox_salto: 1.  
  
(nodo1 salidas at: 1) bandwidth: 5.  
(nodo1 salidas at: 2) bandwidth: 5.  
(nodo2 salidas at: 1) bandwidth: 5.  
(nodo2 salidas at: 2) bandwidth: 5.  
(nodo3 salidas at: 1) bandwidth: 5.  
(nodo3 salidas at: 2) bandwidth: 5.  
(nodo4 salidas at: 1) bandwidth: 5.  
(nodo4 salidas at: 2) bandwidth: 5.  
  
nodo1 tabla at: 2 put: 2.  
nodo1 tabla at: 3 put: 2.  
nodo1 tabla at: 4 put: 4.  
nodo2 tabla at: 1 put: 1.  
nodo2 tabla at: 3 put: 3.  
nodo2 tabla at: 4 put: 3.  
nodo3 tabla at: 1 put: 2.  
nodo3 tabla at: 2 put: 2.  
nodo3 tabla at: 4 put: 4.  
nodo4 tabla at: 1 put: 1.  
nodo4 tabla at: 2 put: 3.  
nodo4 tabla at: 3 put: 3.
```

Figura 43: Inicialización

- Guardamos ahora la página 1 en el buffer de entrada del nodo 1 y “distribuimos”. Como vemos al tener como destino un nodo inexistente los paquetes nunca llegan a destino.

```
Transcript show: (nodo1 entrada cola);cr.  
nodo1 entrada recibirMuchos: (pag1 paquetes).  
Transcript show: (nodo1 entrada cola);cr.  
nodo1 distribuir.
```

Figura 44: Intentamos enviar la página 1

```
a Cola()  
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete)  
No se puede llegar al destino  
No se puede llegar al destino  
No se puede llegar al destino  
No se puede llegar al destino  
No se puede llegar al destino
```

Figura 45: Salida Transcript

- Haciendo los mismos pasos pero para la segunda página el resultado es diferente. Vemos como luego de distribuir, los paquetes llegan al primer buffer de salida del nodo 1.

```
Transcript show: (nodo1 entrada cola);cr.  
nodo1 entrada recibirMuchos: (pag2 paquetes).  
Transcript show: (nodo1 entrada cola);cr.  
Transcript show: ((nodo1 salidas at: 1) cola);cr.  
Transcript show: ((nodo1 salidas at: 2) cola);cr.  
nodo1 distribuir.  
Transcript show: ((nodo1 salidas at: 1) cola);cr.  
Transcript show: ((nodo1 salidas at: 2) cola);cr.
```

Figura 46: Intentamos enviar la página 2

```
a Cola()  
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete)  
a Cola()  
a Cola()  
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete)  
a Cola()
```

Figura 47: Salida Transcript

- Ahora enviamos todo desde el nodo 1 y vemos que se vacía el buffer que contenía los paquetes y el buffer de entrada del nodo 2 no está vacío.

```
nodo1 enviarTodo: (red nodos).  
Transcript show: ((nodo1 salidas at: 1) cola);cr.  
Transcript show: (nodo2 entrada cola).
```

Figura 48: Enviamos todo desde el nodo 1

```
a Cola()  
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete)
```

Figura 49: Salida Transcript

- Distribuimos ahora desde el nodo 2 y vemos que los paquetes se mudaron al buffer de salida que conecta con el nodo 3 (el destino de los paquetes que salieron del nodo 1)

```
Transcript show: ((nodo2 salidas at: 1) cola);cr.  
Transcript show: ((nodo2 salidas at: 2) cola);cr.  
nodo2 distribuir.  
Transcript show: ((nodo2 salidas at: 1) cola);cr.  
Transcript show: ((nodo2 salidas at: 2) cola);cr.
```

Figura 50: Distribuimos desde el nodo 2

```
a Cola()  
a Cola()  
a Cola()  
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete)
```

Figura 51: Salida Transcript

- Enviamos todo desde el nodo 2 e imprimimos en Transcript el buffer de entrada del nodo 3.

```
Transcript show: (nodo3 entrada cola);cr.  
nodo2 enviarTodo: (red nodos).  
Transcript show: (nodo3 entrada cola);cr.
```

Figura 52: Enviamos todo desde el nodo 2

```
a Cola()  
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete)
```

Figura 53: Salida Transcript

- Ahora distribuimos desde el nodo 3 y vemos su buffer interno. Se puede ver que la llave (2) coincide con el id de página y que contiene un array con 5 paquetes.

```
Transcript show: (nodo3 interno dict);cr.  
nodo3 distribuir.  
Transcript show: (nodo3 interno dict);cr.
```

Figura 54: Distribuimos desde el nodo 3

```
a Dictionary()  
a Dictionary(2->an Array(a Paquete a Paquete a Paquete a Paquete a Paquete) )
```

Figura 55: Salida Transcript

- Por último en una única captura ejecutamos el envío de paquetes del nodo 4 al nodo 1 y mostramos como se van llenando y vaciando los diferentes buffers de ambos nodos.

```
Transcript show: (nodo4 entrada cola);cr.  
nodo4 entrada recibirMuchos: (pag3 paquetes).  
Transcript show: (nodo4 entrada cola);cr.  
Transcript show: ((nodo4 salidas at: 1) cola);cr.  
Transcript show: ((nodo4 salidas at: 2) cola);cr.  
nodo4 distribuir.  
Transcript show: ((nodo4 salidas at: 1) cola);cr.  
Transcript show: ((nodo4 salidas at: 2) cola);cr.  
Transcript show: (nodo1 entrada cola);cr.  
nodo4 enviarTodo: (red nodos).  
Transcript show: (nodo1 entrada cola);cr.  
Transcript show: (nodo1 interno dict);cr.  
nodo1 distribuir.  
Transcript show: (nodo1 interno dict);cr.
```

Figura 56: Enviamos desde el nodo 4 al nodo 1

```

a Cola()
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete)
a Cola()
a Cola()
a Cola()
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete)
a Cola()
a Cola(a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete)
a Dictionary()
a Dictionary(3->an Array(a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete a Paquete) )

```

Figura 57: Salida Transcript

#### 4.4. Guardar información en un archivo:

##### Clase Nodo:

Se agrego en la clase Nodo un nuevo método llamado “guardar”. Como el proceso se realiza página por página, nunca va a existir en el buffer interno más de una página, de modo que guardando todas las llaves en un arreglo y tomando la primera (la única) podemos saber que id de página es. Luego toma un paquete cualquiera de esa página para obtener los datos de origen, destino y tamaño de bloque, el cual compara con la cantidad de paquetes recibidos.

```

guardar
  "comment stating purpose of message"

  | name key array dict |
  dict := interno dict.
  key := dict keys at: 1.
  array := interno dict at: key.
  name := 'nodo' , id asString , '-pagina' , key asString.
  FileStream
    forceNewFileNamed: name
    do: [ :stream |
      stream
        nextPutAll: 'Origen: ' , (array at: 1) origen asString;
        cr;
        nextPutAll: 'Destino: ' , (array at: 1) destino asString;
        cr;
        nextPutAll: 'Tamaño de pág inicial: ' , (array at: 1) tbloq asString;
        cr;
        nextPutAll: 'Total recibido: ' , array size asString;
        cr;
        close ].
  interno initialize

```

Figura 58: Nuevo método

Al ejecutarlo desde el nodo 1 obtenemos el siguiente archivo. Vemos como el nombre del archivo corresponde al nodo 1 y a la página 3. El archivo nos indica que la página fue enviada desde el nodo 4 al nodo 1 y que está fragmentada en 10 bloques.

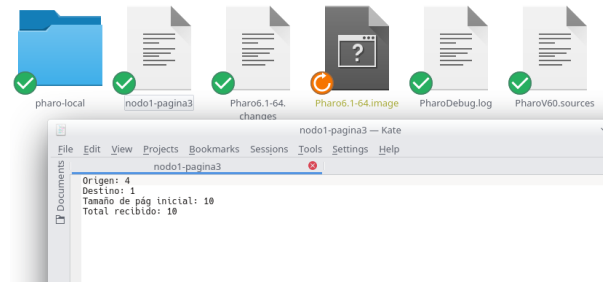


Figura 59: Archivo de salida.

#### 4.5. Inicializar mediante un archivo:

Ahora vamos a crear un archivo que contenga los datos de la topología antes estudiada y el programa se encargue de inicializar correctamente los valores. Se notó que era necesario modificar la clase Config para agregar una OrderedCollection que contenga la totalidad de las páginas de la misma manera que contiene la totalidad de los nodos.

##### Clase Config:

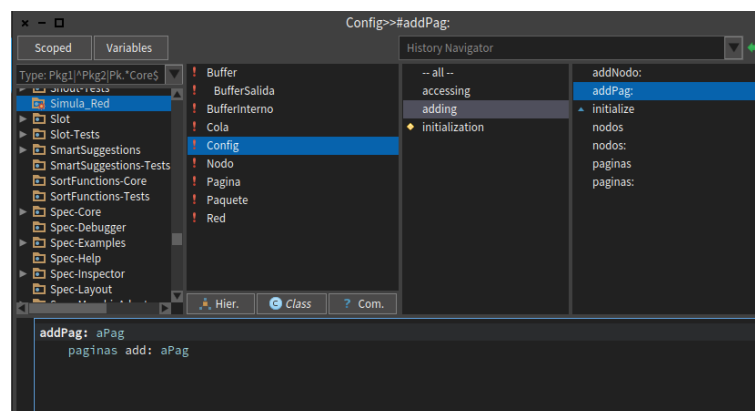


Figura 60: Modificaciones.



## Clase Red:

Fue necesario agregar un método reset para borrar la instancia, de lo contrario las modificaciones anteriores no tenían efecto sobre un objeto ya creado. El método simplemente verifica si current no es nulo, entonces lo hace nulo.

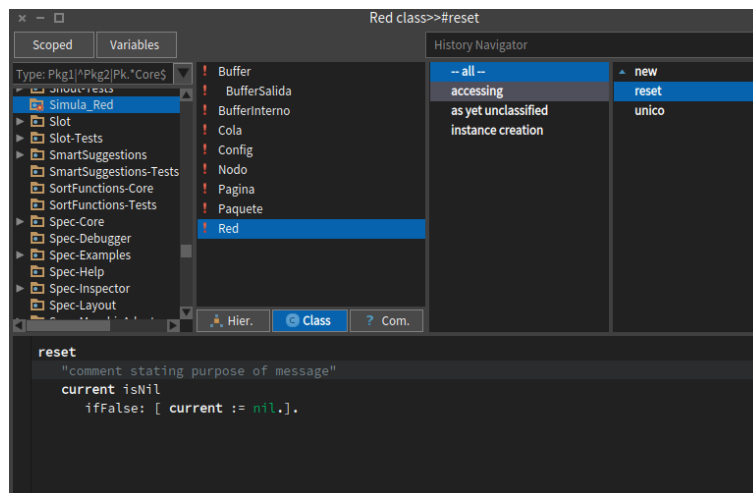


Figura 61: Nuevo método reset

## Clase Start:

Finalmente, ésta es la clase que inicializa desde el archivo.

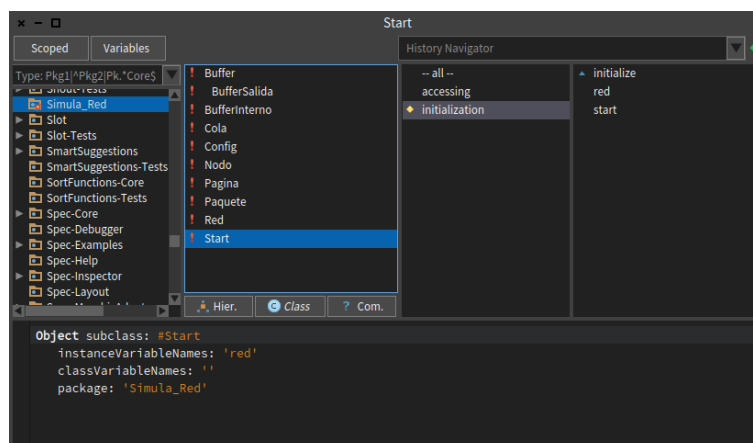


Figura 62: Clase Start.

### Métodos (Mensajes):

- “initialize”: Inicializa la variable red, primero resetea en caso de haber una instancia creada anteriormente.

```
initialize
"comment stating purpose of message"
Red reset.
red := Red unico.
```

Figura 63: Método initialize.

- “red”: Simplemente devuelve la variable “red”.
- “start”: El método comienza abriendo un archivo llamado “inicialización” y comienza leyendo línea por línea guardándolas en la variable “line”.

Luego utiliza una variable “var” para guardar una expresión booleana que luego va a ser evaluada por el bloque “ifTrue”.

La expresión booleana se obtiene haciendo uso del método “matches-Regex”.

El bloque “ifTrue” utiliza la variable “re” donde se guarda una expresión regular que luego se evalúa con cada línea obtenida del archivo, y mediante “subexpression” se obtienen subcadenas. Las subcadenas son del tipo “ByteString” por lo que necesitan ser convertidas a “SmallInteger” antes de ser utilizadas.

Como se ve en la imagen el procedimiento es muy similar para cada bloque “ifTrue”.

```
start
"comment stating purpose of message"

[file var line a b c d re ]
file := FileStream fileName: 'inicializacion'.
[ file atEnd ]
  whileFalse: [
    line := file nextLine.
    var := line matchesRegex: 'nodo\..+'.
    var
      ifTrue: [
        re := '(nodo\.)\(\d+)\.\(\d+)\.' asRegex.
        re matchesPrefix: line.
        a := (re subexpression: 3) asInteger.
        b := (re subexpression: 4) asInteger.
        red addNodo: (Nodo id: a tam: b)
      ].
    var := line matchesRegex: 'pagina\..+'.
    var
      ifTrue: [
        re := '(pagina\.)\(\d+)\.\(\d+)\.\(\d+)\.\(\d+)\.' asRegex.
        re matchesPrefix: line.
        a := (re subexpression: 3) asInteger.
        b := (re subexpression: 4) asInteger.
        c := (re subexpression: 5) asInteger.
        d := (re subexpression: 6) asInteger.
        red addPag: (Pagina or: a dest: b tbloq: c pag: d)
      ].
  ]
]
```

Figura 64: Método start.

*Nota: Para que el método funcione correctamente el archivo “inicialización” debe estar en el path: pharo/shared/.*

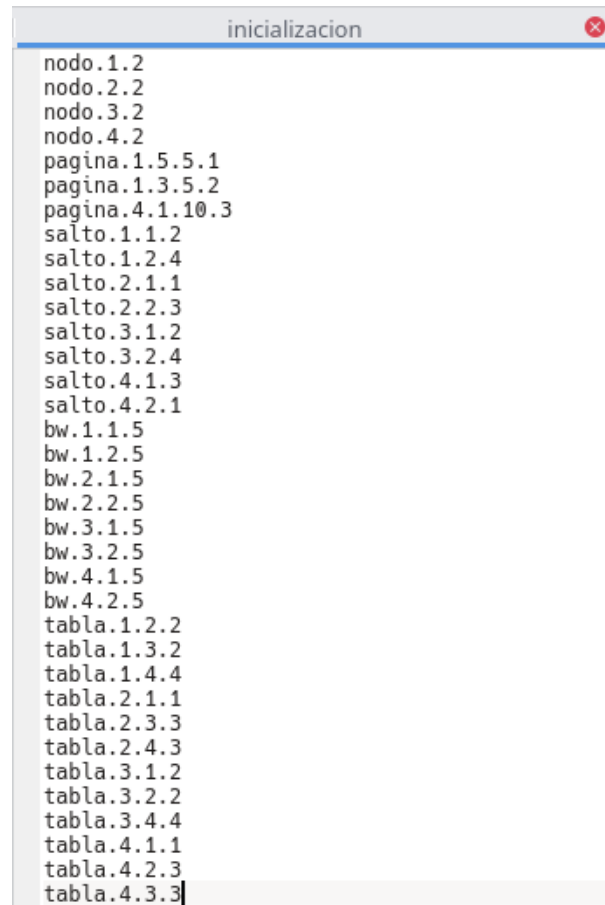
```

var := line matchesRegex: 'salto\..+'.
var
  ifTrue: [
    re := '(salto\.)\(\d+)\.\(\d+)\.\(\d+)' asRegex.
    re matchesPrefix: line.
    a := (re subexpression: 3) asInteger.
    b := (re subexpression: 4) asInteger.
    c := (re subexpression: 5) asInteger.
    ((red nodos at: a) salidas at: b) prox_salto: c
  ].
var := line matchesRegex: 'bw\..+'.
var
  ifTrue: [
    re := '(bw\.)\(\d+)\.\(\d+)\.\(\d+)' asRegex.
    re matchesPrefix: line.
    a := (re subexpression: 3) asInteger.
    b := (re subexpression: 4) asInteger.
    c := (re subexpression: 5) asInteger.
    ((red nodos at: a) salidas at: b) bandwidth: c
  ].
var := line matchesRegex: 'tabla\..+'.
var
  ifTrue: [
    re := '(tabla\.)\(\d+)\.\(\d+)\.\(\d+)' asRegex.
    re matchesPrefix: line.
    a := (re subexpression: 3) asInteger.
    b := (re subexpression: 4) asInteger.
    c := (re subexpression: 5) asInteger.
    ((red nodos at: a) tabla at: b) put: c
  ].

```

Figura 65: Método start continuación.

El archivo “inicializacion” tiene el siguiente formato.

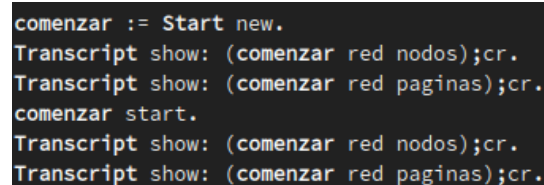


```
nodo.1.2
nodo.2.2
nodo.3.2
nodo.4.2
pagina.1.5.5.1
pagina.1.3.5.2
pagina.4.1.10.3
salto.1.1.2
salto.1.2.4
salto.2.1.1
salto.2.2.3
salto.3.1.2
salto.3.2.4
salto.4.1.3
salto.4.2.1
bw.1.1.5
bw.1.2.5
bw.2.1.5
bw.2.2.5
bw.3.1.5
bw.3.2.5
bw.4.1.5
bw.4.2.5
tabla.1.2.2
tabla.1.3.2
tabla.1.4.4
tabla.2.1.1
tabla.2.3.3
tabla.2.4.3
tabla.3.1.2
tabla.3.2.2
tabla.3.4.4
tabla.4.1.1
tabla.4.2.3
tabla.4.3.3
```

Figura 66: Archivo inicializacion

### Ejecución:

Una simple ejecución de la inicialización del programa se muestra a continuación.



```
comenzar := Start new.
Transcript show: (comenzar red nodos);cr.
Transcript show: (comenzar red paginas);cr.
comenzar start.
Transcript show: (comenzar red nodos);cr.
Transcript show: (comenzar red paginas);cr.
```

Figura 67: Inicialización

Vemos como primero las OrderedCollection de nodos y de páginas están vacías hasta que se ejecuta el método “start”.

```
an OrderedCollection()
an OrderedCollection()
an OrderedCollection(a Nodo a Nodo a Nodo a Nodo)
an OrderedCollection(a Pagina a Pagina a Pagina)
```

Figura 68: Salida Transcript

Ya estando la red inicializada podemos empezar a correr los métodos de los nodos para que los paquetes comiencen a circular hasta llegar a su destino.

```
(comenzar red nodos at: 1) entrada recibirMuchos: (comenzar red paginas at: 2) paquetes.
(comenzar red nodos at: 1) distribuir.
(comenzar red nodos at: 1) enviarTodo: (comenzar red nodos).
(comenzar red nodos at: 2) distribuir.
(comenzar red nodos at: 2) enviarTodo: (comenzar red nodos).
(comenzar red nodos at: 3) distribuir.
(comenzar red nodos at: 3) guardar.

(comenzar red nodos at: 4) entrada recibirMuchos: (comenzar red paginas at: 3) paquetes.
(comenzar red nodos at: 4) distribuir.
(comenzar red nodos at: 4) enviarTodo: (comenzar red nodos).
(comenzar red nodos at: 1) distribuir.
(comenzar red nodos at: 1) guardar.
```

Figura 69: Ejecución

Al finalizar esas ejecuciones vemos que ya se han guardado dos archivos. Uno que indica que el nodo 1 recibió la página 3 y otro que el nodo 3 recibió la página 2. Abriéndolos podemos ver su contenido.

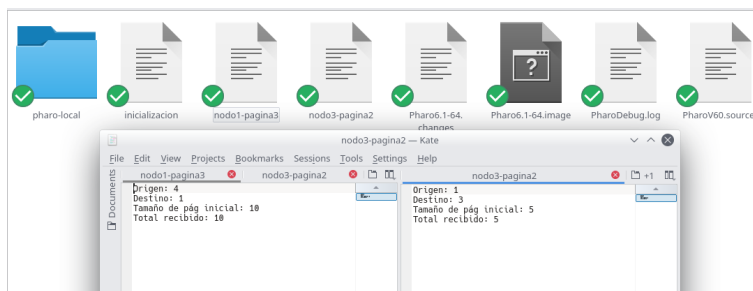


Figura 70: Guardado en archivos.

#### 4.6. Compartir el código:

Ahora podemos guardar nuestro proyecto con sus modificaciones para importarlo en otro entorno.

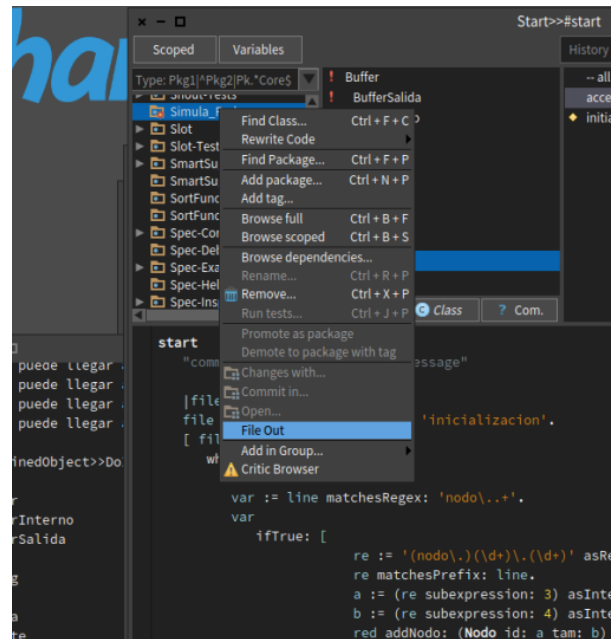


Figura 71: Guardando proyecto.

Tenemos ahora nuestro archivo .st que podemos importarlo como hicimos anteriormente.

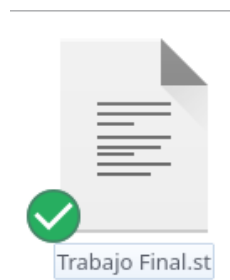


Figura 72: Archivo .st

## 4.7. Últimas modificaciones:

Si bien el código cumple con su propósito, hay ciertas prácticas que deben respetarse al usarse diferentes paradigmas de programación. En este proyecto varias colecciones (OrderedCollection) fueron recorridas usando una sintaxis muy similar a la que se usa en lenguajes imperativos como C.

Por ejemplo: en varias secciones del código se pueden encontrar sentencias como la que se muestra a continuación.

```
1 to: (comenzar red nodos) size  
do:  
  [ :i | Transcript show: (comenzar red nodos at: i) id;cr ].
```

Figura 73: Bucle usado en lenguajes imperativos

Lo más correcto para iterar sobre objetos en colecciones en un paradigma orientado a objetos es hacerlo mediante el uso de foreach, el cual se muestra a continuación.

```
comenzar red nodos do: [ :i | Transcript show: i id;cr ].
```

Figura 74: Bucle foreach usado en programación orientada a objetos.

Además de respetar el paradigmas se encuentra que el código queda mucho mejor representado de esta forma.

Vamos a mostrar como quedan los siguientes métodos luego de implementar los cambios propuestos.

- Método “enviar:” de la clase “BufferSalida”.

```
enviar: nodos
"comment stating purpose of message"

| cs buffer |
nodos
do: [ :i |
    i id = prox_salto
    ifTrue: [ buffer := i entrada ] ].
cs := cola size min: bandwidth.
1 to: cs do: [ :i | self mover: buffer ]
```

Figura 75: Método modificado

- Método “enviarTodo:” de la clase “Nodo”.

```
enviarTodo: nodos
"comment stating purpose of message"

salidas do: [ :i | [ i cola isEmpty ] whileFalse: [ i enviar: nodos ] ]
```

Figura 76: Método modificado



- Método “distribuir” de la clase “Nodo”.

```

distribuir
"identifica si el paq es para reenviar o queda local"

| temp next_hop |
next_hop := nil.
[ entrada cola isEmpty ]
  whileFalse: [ temp := entrada cola desencolar.
    temp destino = id
      ifTrue: [ interno recibir: temp ]
      ifFalse: [ tabla
        at: temp destino
          ifPresent: [ next_hop := tabla at: temp destino.
            salidas
              do: [ :i |
                i prox_salto = next_hop
                  ifTrue: [ i recibir: temp ] ] ].
        tabla
          at: temp destino
            ifAbsent: [ Transcript
              show: 'No se puede llegar al destino';
              cr ] ] ]

```

Figura 77: Método modificado

En los casos donde se notaba fácilmente que se estaba usando un bucle “no políticamente correcto” de acuerdo al paradigma, es cuando se obtenía primero el tamaño de la `OrderedCollection` con “size” y luego se accedía a un objeto en particular con “at:”, siendo mucho más largo y no utilizando la facilidad del `foreach` propio de la programación orientado a objetos.

## 5. Anexo:

### 5.1. Sintaxis básica:

#### Palabras reservadas:

- true, false: Objetos booleanos.
- nil: Objeto indefinido.
- thisContext: Método o bloque actual.
- self: Receptor del mensaje actual.
- super: Para acceder a una super clase.

#### Constructores de objetos y sintácticos:

- “Comentario”
- 'cadena de caracteres (string)'
- Caracter: \$a.
- Arreglos:
  - #(asdf 23)
  - #[12 23 3]
  - sdf. 2+3.
- Declaración de variables temporales: | a b |
- Asignación: a:=2.
- Separador (.): expresion1. expresion2
- Cascada: ; (semicolon)
- Return: ^
- Bloque: [ :a | 5 + a] value: 7. (resultado = 12).

#### Mensajes:

- Unario:
  - Array new.
  - #(12 1 3) size.

- Binario:

1 + 3.

“hello ”, “ world”.

- Keyword:

[ :a | 5 + a] value: 7. (resultado = 12).

“Pharo” allButFirst: 2. (resultado “aro”).

2 to: 10 by: 2.

## Condicionales

```
condition
  ifTrue: [action]
  ifFalse: [another action]
```

Ejemplo:

```
2 = 2
  ifTrue: [Transcript show: “hello”].
```

## Bucles:

```
0 to: 100 by: 25
  do: [:i | Transcript show: i; cr ].
```

*Resultado:*

0  
25  
50  
75  
100

## Iteradores:

```
b := #(A B C).
b do: [:each | Transcript show: each, ' '].
```

*Resultado:*

A,B,C,

## Colecciones:

```
#(3 1 2) copy at: 3 put: 1
```

*Resultado:*

#(3 1 1)

## 5.2. Patrón Singleton:

### Contexto:

En ocasiones puede ser deseable que cuando se crea una clase, solo se cree un objeto de ella. No es una buena práctica dejar tales comportamientos en manos del usuario, sino que el programa debe ser capaz de instanciar una sola vez la clase.

La solución es hacer que la clase sea en si misma responsable de crear un solo objeto.

Primero creamos una variable de clase que contendrá el “singleton” y un método “current” (actual) que responderá con la única instancia creada, además el método “new” se anula sobreescribiéndolo de manera que cuando se invoque lance una excepción.

*La clase “Red” de nuestro problema es la encargada de implementar el patrón de diseño Singleton.*

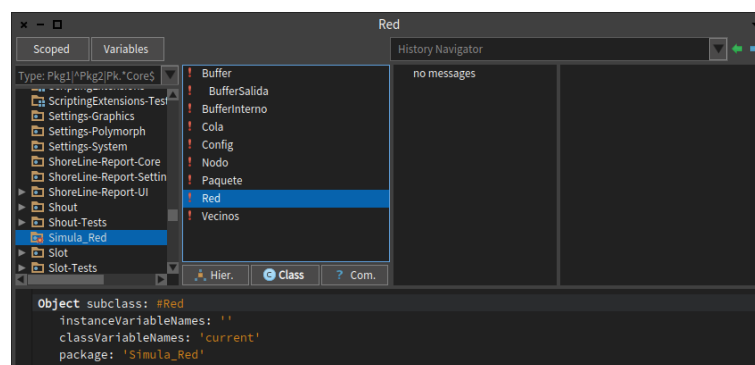


Figura 78: Clase Red. Se puede ver la variable de clase “current”

En [20] vemos que el método “new” lanza una excepción si se invoca, lo que evita que se creen nuevas instancias.

El método “unico” crea una instancia si la variable de clase encargada de almacenar el singleton está vacía, o devuelve el único objeto creado en caso contrario, como se puede ver en [21]

## 5.3. Streams:

### 5.3.1. Introducción:

Los stream se usan para iterar sobre secuencias en colecciones o en archivos. Pueden ser de tipo escritura, lectura o una combinación de ambos. Los stream pueden ser convertidos en colecciones y viceversa.

### 5.3.2. Stream para acceder a archivos:

Para crear un stream de archivos existen varias formas.

Debemos primero crear una instancia de la clase “FileStream” Luego:

1. **fileNamed**: Abre un archivo con el nombre dado para lectura y escritura. Si el archivo existe, su contenido previo puede ser reemplazado. El archivo será creado en el directorio por defecto sino se especifica ruta.
2. **newFileNamed**: Crea un nuevo archivo con el nombre dado, y responde con un stream abierto para escritura sobre ese archivo. Si el archivo existe, pregunta al usuario que hacer.
3. **forceNewFileNamed**: Crea un nuevo archivo con el nombre dado, y responde con un stream abierto para escritura en ese archivo. Si el archivo existe, lo borra sin preguntar antes de crear uno nuevo.
4. **oldFileNamed**: Abre un archivo existente con el nombre dado para lectura y escritura. Si el archivo existe, su contenido previo puede ser reemplazado. Si el archivo no tiene una ruta definida, se crea en el directorio por defecto.
5. **readOnlyFileNamed**: Abre un archivo existente con el nombre dado para lectura.

El stream debe cerrarse una vez utilizado. Esto se logra con un mensaje “close” a stream.

### Ejemplos:

```
FileStream
forceNewFileNamed: 'ejemplo'
do: [ :stream |
    stream
    nextPutAll: 'Paradigmas de programación' ].
```

Figura 79: Guarda un texto en un archivo.

En la captura anterior observamos como haciendo uso de bloques, guardamos una cadena en un archivo de nombre “ejemplo”.

```
string := FileStream
  readOnlyFileNamed: 'ejemplo'
  do: [ :stream | stream contents ].
string.
```

Figura 80: Guarda en una variable el contenido del archivo.

Para leer, se crea una variable y nuevamente usando bloques almacenamos el contenido.

Nótese que primero se usó “forceNewFileNamed” y luego “readOnlyFileNamed”

```
string do: [:each | Transcript show: each;cr].
```

Figura 81: Itera sobre la variable.

Mediante “DO-IT (ctrl-d)” se puede ver en *Transcript* la salida en columna del mensaje.

## Referencias

- [1] Pharo.org. Pharo Cheat Sheet. URL: <http://files.pharo.org/media/pharoCheatSheet.pdf>.
- [2] Pharo.org. Pharo By Example. URL: <https://ci.inria.fr/pharo-contribution/job/UpdatedPharoByExample/lastStableBuild/artifact/book-result/>.
- [3] Squeak.org. Pharo: Classes References. URL: <http://wiki.squeak.org/squeak/uploads/SqueakClassesRef.html>.
- [4] San Bernardino California State University. Smalltalk Methods. URL: <http://www.csci.csusb.edu/dick/samples/smalltalk.methods.html>.
- [5] Informatics University of Warsaw Faculty of Mathematics y Mechanics. Singleton Pattern. URL: [https://www.mimuw.edu.pl/~sl/teaching/00\\_01/Delfin\\_EC/Patterns/Singleton.htm](https://www.mimuw.edu.pl/~sl/teaching/00_01/Delfin_EC/Patterns/Singleton.htm).
- [6] JGraph Ltd. Draw.io. URL: <https://about.draw.io>.
- [7] Wiki.Squeak. MultiByteFileStream. URL: <https://wiki.squeak.org/squeak/6332>.
- [8] pharo.org. Regular Expressions in Pharo. URL: <https://ci.inria.fr/pharo-contribution/job/UpdatedPharoByExample/lastSuccessfulBuild/artifact/book-result/Regex/Regex.html>.