TP N°3: Diseño Avanzado en Ruby

La siguiente guía cubre los contenidos vistos en las clases teóricas **3 y 4**

Ejercicio 1

HTML es un lenguaje utilizado en el desarrollo de páginas web. Permite describir la estructura de un documento, así como darle formato. Para hacer esto, utiliza "etiquetas" que permiten indicar el formato a aplicar. A continuación se describen algunas de estas etiquetas:

- **b**: Permite definir un texto en **negrita**. Ejemplo: `<b>hola</b>`
- **i**: Permite definir un texto en **cursiva**. Ejemplo: `<i>hola</i>`
- **a**: Permite definir un **enlace**, agregando un atributo que indica la página a la cual se desea ir al hacer clic en el enlace. Ejemplo: `<a href="www.itba.edu.ar">Ir a la página del ITBA</a>`

Estas etiquetas pueden anidarse para combinar distintos formatos. Por ejemplo, el siguiente texto HTML muestra la cadena de texto "hola" en negrita y en cursiva: `<b><i>hola</i></b>`. El siguiente código hace lo mismo: `<i><b>hola</b></i>.`

Se cuenta con un módulo `HTMLText` que representa un texto HTML y provee un método para obtener el código fuente.

```ruby
module HTMLText
 def source
   raise 'Not implemented'
 end

 def to_s
   source
 end
end
```

Se cuenta además con una implementación para textos sin formato (en los cuales el código fuente coincide con el texto a mostrar, ya que no se aplica ninguna etiqueta).

```ruby
class PlainText
 include HTMLText

 attr_writer :content

 def initialize(content)
   @content = content
 end

 def source
   @content
 end
end
```

Se quiere ofrecer más funcionalidad para permitir representar textos en negrita, en cursiva y enlaces. **Implementar todo lo necesario y completar todos los ... para que, con el siguiente programa de prueba, se obtenga la salida indicada en los comentarios. Realizar el diagrama de clases correspondiente.**

```
text = PlainText.new 'Hola'
bold_text = ...
italic_text = ...
puts bold_text # <b>Hola</b>
puts italic_text # <i>Hola</i>
bold_italic_text = ...
puts bold_italic_text # <b><i>Hola</i></b>
text.content = 'ITBA'
puts bold_text # <b>ITBA</b>
puts italic_text # <i>ITBA</i>
puts bold_italic_text # <b><i>ITBA</i></b>
link_text = ...
link_bold_italic_text = ...
bold_link_text = ...
puts link_text # <a href:"www.itba.edu.ar">ITBA</a>
puts link_bold_italic_text # <a href:"www.itba.edu.ar"><b><i>ITBA</i></b></a>
puts bold_link_text # <b><a href:"www.itba.edu.ar">ITBA</a></b>
text.content = 'Ejemplo'
puts link_bold_italic_text # <a href:"www.itba.edu.ar"><b><i>Ejemplo</i></b></a>
puts bold_link_text # <b><a href:"www.itba.edu.ar">Ejemplo</a></b>
```

Ejercicio 2

Se cuenta con el módulo `Movable` que cuenta en su interfaz pública con cuatro métodos para mover a un objeto en las cuatro direcciones.

```ruby
module Movable
 def move_up(_delta)
   raise 'Not implemented'
 end

 def move_down(_delta)
   raise 'Not implemented'
 end

 def move_left(_delta)
   raise 'Not implemented'
 end

 def move_right(_delta)
   raise 'Not implemented'
 end
end
```

Implementar todo lo necesario para que las figuras geométricas del ***TP N°2*** puedan moverse en el espacio. Actualizar el diagrama de clases. Realice un programa de prueba que mueva a una elipse.

Ejercicio 5

Se cuenta con el módulo `Function` que ofrece un método `evaluate` con el fin de evaluar una funciones matemática de una variable real,

```
module Function
 def evaluate(_x)
   raise 'Not Implemented'
 end
end
```

De esta forma, se pueden tener implementaciones diversas que correspondan a distintas funciones matemáticas, como se muestra a continuación:

```
class LinearFunction
 include Function

 def initialize(a, b)
   @a = a
   @b = b
 end

 def evaluate(x)
   @a * x + @b
 end
end
```

```
class QuadraticFunction
 include Function

 def initialize(a, b, c)
   @a = a
   @b = b
   @c = c
 end

 def evaluate(x)
   @a * x**2 + @b * x + @c
 end
end
```

```
class SineFunction
 include Function

 def evaluate(x)
   Math.sin(x)
 end
end
```

Realizar otra implementación que incluya al módulo `Function` que permita representar la **composición de dos funciones**. La clase a implementar debe llamarse **`CompositeFunction`** y debe recibir en el constructor dos funciones a componer, como se muestra en el siguiente programa de prueba:

```
f1 = LinearFunction.new(2, 0) # y = 2x
f2 = QuadraticFunction.new(1, 0, 0) # y = x^2
f3 = CompositeFunction.new(f1, f2) # y = (2x)^2
puts f3.evaluate(1) # 4
puts f3.evaluate(2) # 16
f4 = SineFunction.new # y = sin(x)
f5 = CompositeFunction.new(f1, f4) # y = sin(2x)
f6 = CompositeFunction.new(f5, f1) # y = 2 sin(2x)
puts f6.evaluate(0) # 0.0
puts f6.evaluate(Math::PI / 4.0) # 2.0
```

Analizar desde el punto de vista del paradigma de la programación orientada a objetos las distintas implementaciones que incluyen el módulo `Function`. ¿Se podría plantear alguna jerarquía de clases distinta para poder reutilizar código?

Ejercicio 4

Se cuenta con el módulo `Expression` que representa a una expresión lógica que puede ser evaluada en `true` o `false`:

```ruby
module Expression
 def evaluate
   raise 'Not implemented'
 end
end
```

Se cuenta con la clase `SimpleExpression` que modela la expresión de un operando:

```ruby
class SimpleExpression
 include Expression

 attr_writer :value

 def initialize(value)
   @value = value
 end

 def evaluate
   @value
 end
end
```

Se desea agregar la posibilidad de construir expresiones que realicen **operaciones lógicas (*and*, *or* y *not*)**. Implementar todo lo necesario para que el siguiente programa de prueba imprima en salida estándar lo que se indica en los comentarios.

```
exp1 = SimpleExpression.new(true)
exp2 = SimpleExpression.new(false)
exp3 = exp1.not
exp4 = exp1.or(exp2)
exp5 = exp3.and(exp4)
p exp1.evaluate # true
```

```
p exp3.evaluate # false
p exp4.evaluate # true
p exp5.evaluate # false
exp1.value = false
p exp3.evaluate # true
p exp4.evaluate # false
p exp5.evaluate # false
exp2.value = true
p exp5.evaluate # true
```

Ejercicio 5

Un **iterador cíclico** es un iterador que, a partir de una colección, permite recorrerla infinitamente. Al igual que un iterador convencional permite recorrer todos los elementos de la colección, pero la diferencia es que, una vez consumido todos los elementos de la colección, vuelve a recorrer los mismos elementos.

Por ejemplo, usando el iterador que provee Ruby, la siguiente invocación

```
p [1, 2, 3].take(7)
```

produce la siguiente salida

```
[1, 2, 3]
```

Implementar la clase `CyclicIterator` que recibe en su constructor la colección a iterar y cuenta con un método `each` para poder acceder al iterador cíclico. Por ejemplo, la siguiente invocación

```
cyclic_iterator = CyclicIterator.new([1, 2, 3])
p cyclic_iterator.each.take(7)
```

produce la siguiente salida

```
[1, 2, 3, 1, 2, 3, 1]
```

Ejercicio 6

Se debe diseñar un conjunto de clases que modelan un Video Club.

Las categorías de películas que ofrece el Video Club son las siguientes:

- **`Recent`:** Son estrenos, se cobra $3 por cada día de alquiler.
- **`Children`:** Películas infantiles, se cobra $3 por 3 días de alquiler, y cada día de atraso se cobra $1,5.
- **`Standard`:** El resto de las películas, se cobra $2 por 2 días de alquiler, y cada día de atraso se cobra $1,5

La forma de calcular los puntos de un cliente del Video Club es la siguiente:

- Cada película alquilada suma un punto.
- Si la película alquilada es un Estreno suma un punto extra por cada día extra alquilada.

Realizar el **diagrama de clases completo**, e **implementar todo lo necesario para que** el siguiente programa de prueba:

```
class MovieCategory
 STANDARD = 1
 RECENT = 2
 CHILDREN = 3
end

video_club = VideoClub.new
video_club.add_movie('Dumbo', MovieCategory::CHILDREN)
video_club.add_movie('ET', MovieCategory::STANDARD)
video_club.add_movie('ZZZ', MovieCategory::RECENT)
video_club.add_customer('Juan')
video_club.add_customer('Ana')
video_club.rent('Dumbo', 'Ana', 5)
video_club.rent('ET', 'Ana', 2)
video_club.rent('ET', 'Juan', 3)
puts video_club.resume('Ana')
puts video_club.resume('Juan')
begin
 video_club.rent('ET', 'Pedro', 3)
rescue RuntimeError => e
 puts e.message
end
```

genere la siguiente salida

```
Resume points: 2, charge: 8.0
Resume points: 1, charge: 3.5
Customer Pedro not found
```

~~*Ejercicio 7*~~

Diseñar la clase `Polynomial`, que representa un polinomio de cierto grado determinado por el usuario. Para esta implementación **se deben hacer todas las validaciones necesarias** realizando el manejo de errores con excepciones propias. Implementar todo lo necesario para que el siguiente programa de prueba:

```
fourth_grade_pol = Polynomial.new(4)
fourth_grade_pol.set(2, 3.1)
fourth_grade_pol.set(3, 2)
puts fourth_grade_pol.eval(2) # 28.4

puts Polynomial.new(3).eval(5) # 0

begin
  Polynomial.new(4.5)
rescue InvalidGradeError => e
  puts e.message
end

begin
  fourth_grade_pol.eval('Hola')
rescue InvalidValueError => e
```

```
  puts e.message
end

fourth_grade_pol.set(7, 1.5)
```

produzca la siguiente salida

```
28.4
0
Grado Inválido
Valor Inválido
/ej2.rb:28:...: Índice Inválido
```

~~*Ejercicio 8*~~

Se cuenta con la clase `CellPhoneBill` que permite registrar las llamadas realizadas por un teléfono celular. Dicha clase cuenta con el método `processBill()` que calcula el monto de la factura en base a las llamadas registradas. Cada llamada está modelada mediante la clase `Call`, que calcula el precio de una llamada conociendo el costo de la misma por segundo.

```
class Call
 COST_PER_SECOND = 0.01

 def initialize(from, to, duration)
   @from = from
   @to = to
   @duration = duration
 end

 def cost
   @duration * COST_PER_SECOND
 end
end
```

```
class CellPhoneBill
 def initialize(number)
   @number = number
   @calls = []
 end

 def register_call(to_number, duration)
   @calls.push(Call.new(@number, to_number, duration))
 end

 def process_bill
   @calls.map{ |c| c.cost }.reduce(:+)
 end
end
```

Se quiere agregar la promoción de "**números amigos**", mediante la cual el usuario registra cierta cantidad de amigos, con los cuales sus llamadas tendrán un **precio especial** que corresponde a un porcentaje del valor real.

La promoción contempla que se agreguen y eliminen números amigos en cualquier momento (en cada instante no puede haber más del límite definido al momento de crear la promoción), y también que se modifique el porcentaje a cobrar.

Los valores a cobrar se consideran en base al estado en el momento de calcular el monto total.

Realizar el diagrama de clases completo, implementar todo lo necesario y diseñar un programa de prueba donde se crea una promoción de números amigos de hasta tres participantes, se realizan llamadas a destinatarios dentro y fuera del grupo y se obtiene un mensaje de error al intentar agregar un cuatro miembro al grupo.

Ejercicio 9

Se desea implementar la clase `MonthYear` que representa un par mes y año. La misma debe poder ser usada en **rangos**, como los de la clase `Range` vistos en la teórica.

Investigar la documentación de la clase `Range` para consultar qué métodos son necesarios implementar (Sección *Custom Objects in Ranges*).

Implementar todo lo necesario para que el siguiente programa de prueba

```
my_range = MonthYear.new(11, 2017)..MonthYear.new(3, 2018)
p my_range
p my_range.to_a
puts my_range === MonthYear.new(1, 2018)
puts my_range === MonthYear.new(2, 2016)
```

produzca la siguiente salida

```
11/2017..3/2018
[11/2017, 12/2017, 1/2018, 2/2018, 3/2018]
true
false
```

Ejercicio 10

Implementar la clase `WordCount` que cuenta la cantidad de apariciones de cada una de las palabras de un archivo de texto plano. Implementar además la clase `WordCountTest` que constituye un test unitario que realiza la lectura de un archivo de prueba y verifica una por una la cantidad de apariciones de las palabras con los valores esperados. Verificar además en el test de unidad el resultado que se obtiene al leer un archivo vacío.