Ejercicios Patrones de Diseño

Ejercicio 1

La PUC se encuentra estudiando la puesta en marcha de un nuevo sistema computacional de asignación de becas que pueda conectarse con los registros de admisión y asignar becas a los estudiantes según distintos tipos de criterios. La base de datos de admisión define objetos del tipo *Student*, donde cada estudiante tiene los atributos Name (string), FamilyIncome (integer) y PsuAverage (float)

Para ello se ha pensado en crear una clase llamada ***ScholarshipAssigner*** que recibe en el constructor la lista de alumnos que sale de la base de datos de admisión *(student_list)* como un *array.* Esta clase define un método llamado ***assign_scholarships(number_of_scholarships)*** *que* imprime el nombre de un número de alumnos dado por el parámetro *number_of_scholarships* que corresponda a aquellos estudiantes del listado de estudiantes que más ameriten una beca según un determinado criterio o estrategia.

Admisión se encuentra interesado en considerar, para comenzar, los siguientes criterios para la asignación de becas:

- Mejor puntaje PSU: se le asignarán becas a los alumnos con mejor promedio PSU
- Mayor necesidad económica: se le asignarán becas a los alumnos con menor ingreso familiar
- Híbrido: se le asignarán becas a los alumnos con menor ingreso familiar siempre y cuando hayan obtenido igual o más de 700 puntos en la PSU. Si hay más becas que alumnos que cumplan la restricción, no se asignan las becas sobrantes.

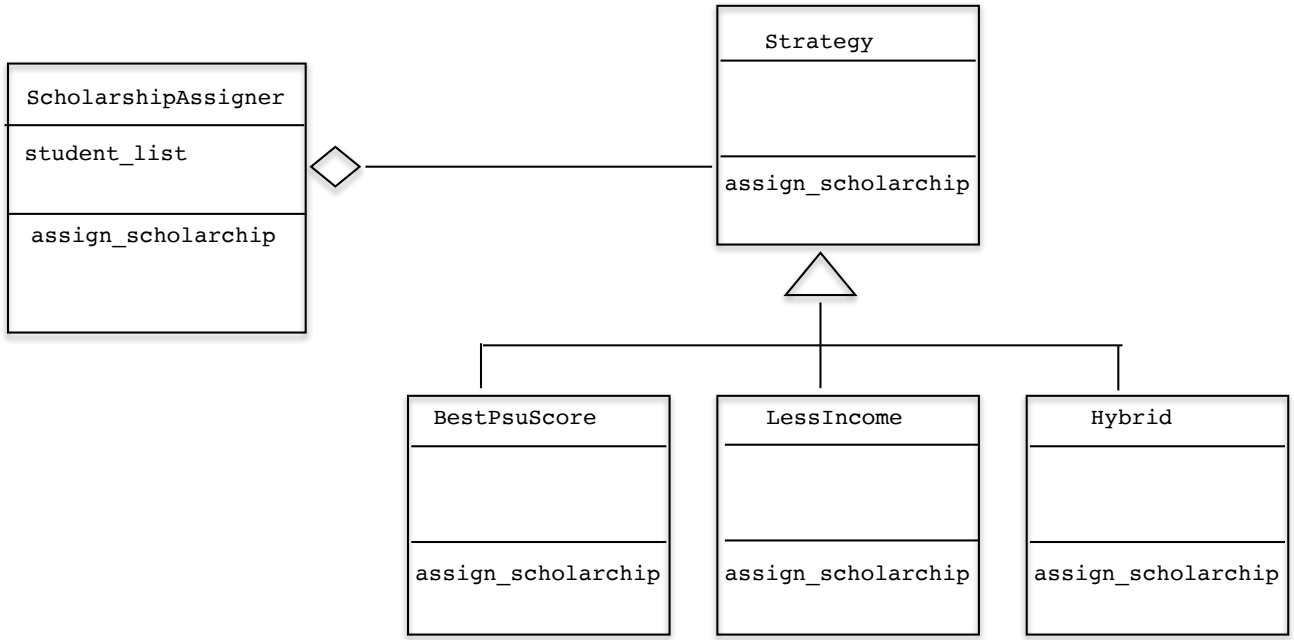
No se descarta que en el futuro aparezcan otros criterios de selección.

Implementar la clase ScholarshipAssigner junto con todas las demás clases que se estime necesarias para una buena solución basada en el patrón Strategy. Las 3 estrategias descritas deben estar implementadas en la solución.

Puede asumir que la clase Student viene dada por lo que puede instanciarla pero no puede implementarla.

Puede asumir también que number_of_scholarships < student_list.length.

Solución




```
class ScholarshipAssigner
  def initialize(student_list, strategy)
    @student_list = student_list
    @theStrat = strategy
  end

  def assign_scholarships(number_of_scholarships)
    @theStrat.assign_scholarships(@student_list, number_of_scholarships)
  end
end
```

```ruby
class BestPsuScoreStrategy
  def assign_scholarships(student_list, number_of_scholarships)
    list = student_list.sort { |a, b| b.psu - a.psu }
    (0...number_of_scholarships).each { |i| puts(list[i].name) }
  end
end

class LessIncomeStrategy
  def assign_scholarships(student_list, number_of_scholarships)
    list = student_list.sort { |a, b| a.income - b.income }
    (0...number_of_scholarships).each { |i| puts(list[i].name) }
  end
end

class HybridStrategy
  def assign_scholarships(student_list, number_of_scholarships)
    list = student_list.sort { |a, b| a.income - b.income }
    count = 0
    list.each do |student|
      if (count < number_of_scholarships) && (student.psu >= 700)
        puts(student.name)
        count += 1
      end
    end
  end
end
```

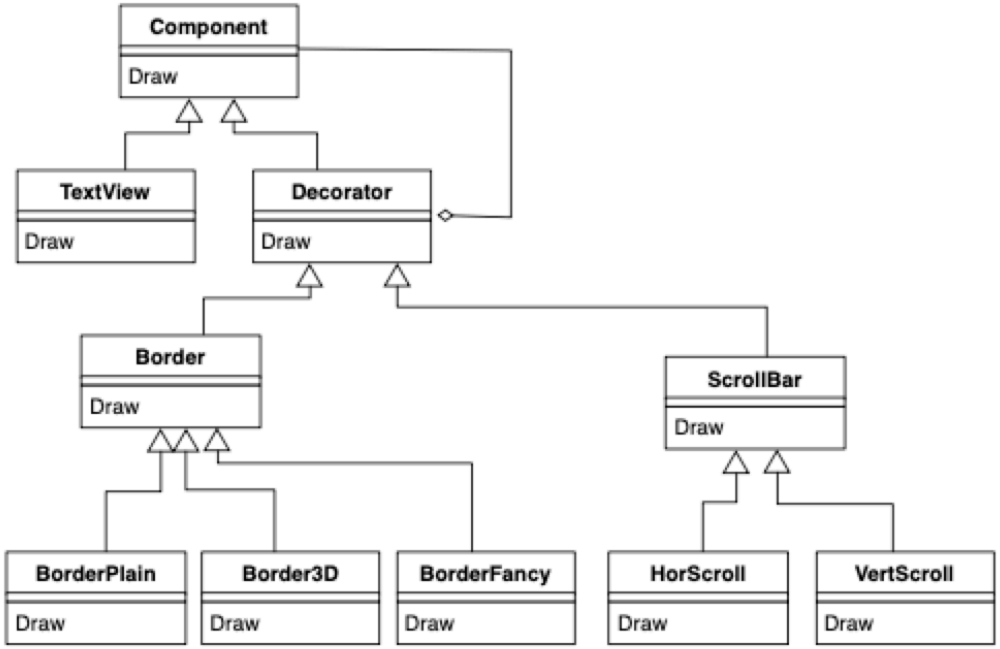
Ejercicio 2

Supon que se dispone de una componente gráfica *TextView* (clase) capaz de desplegar un texto en una ventana gráfica. El problema es que necesitamos ventanas con 3 distintos tipos de bordes (Plain, 3D, Fancy) y también poder agregar *scrollbars* tanto verticales como horizontales (una, otra, o ambas). Un alumno ha propuesto una solución que se basa en generar subclases con todas las especializaciones posibles de *TextView*. Así tendríamos por ejemplo:

```
TextView-Plain
TextView-Plain-Horizontal
TextView-Plain-Vertical
TextView-Plain-HorizontalVertical
TextView-Fancy
TextView-Fancy-Horizontal
TextView-Fancy-Vertical
TextView-Fancy-HorizontalVertical
```

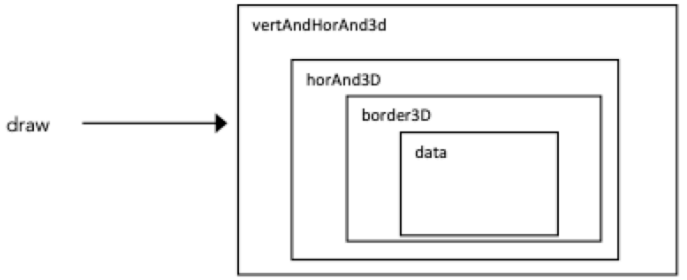
a) Propone un mejor diseño basado en el patrón decorador que estudiamos en clases. Dibuje el diagrama de clases UML que muestre la solución completa y explique por qué este diseño sería superior al propuesto

b) Escribe un pequeño programa Ruby de prueba que muestre como se usaría su solución para obtener una ventana 3D con scrollbars horizontal y vertical y otra ventana Fancy con scrollbar vertical solamente. No necesita escribir el código de la definición de las clases del diagrama mostrado en a)

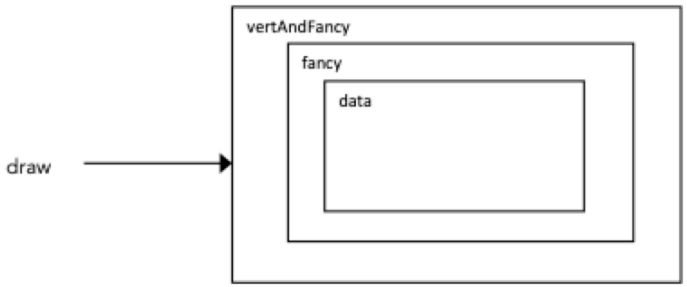


Método Draw de los decoradores simplemente invocan primero el draw de ls componente que contienen y luego dibujan lo especifico del decorador según sea un borde o una barra de scroll

```
data = TextView.new("Esta es una ventana 3D con dos scrollbars")
border3D = Border3D.new (data)
horAnd3D = HorScroll.new (border3D)
vertAndHorAnd3D = VertScroll.new (horAnd3D)
vertAndHorAnd3D.draw
```



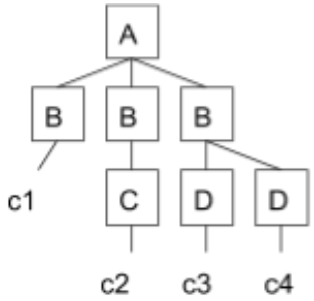

```
data = TextView.new("Esta es una ventana fancy con un scrollbar")
fancy = BorderFancy.new (data)
vertAndFancy = VertScroll.new (fancy)
vertAndFancy.draw
```



Ejercicio 3

Un documento xml tiene estructura de árbol. Un elemento xml (etiqueta) puede contener otros elementos xml o bien solo contenido. Por ejemplo, el árbol de la figura a la derecha corresponde al documento xml de la izquierda

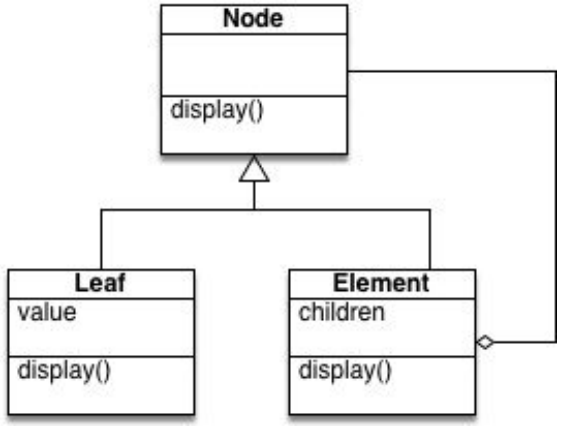
<A>
<B> c1 </B> <B>
<C> c2 </C> </B>
<B>
<D> c3 </D> <D> c4 </D>
</B> </A>



Utiliza el patrón Composite para implementar una clase XmlTree con un método display que genere la versión en texto del árbol. Haga un diagrama de clases con todo lo necesario (métodos y atributos) y escribe (Ruby) el método display.

No te preocupes por identación o espacios, puede mostrar el ábol anterior como:
<A> <B> c1 </B> <B> <C> c2 </C> </B> <B> <D> c3 </D> <D> c4 </D> </B> </A>

```
class Node
    def initialize()
    end
    def display()
    end
end
class Leaf < Node
    def initialize(value)
         @value = value
    end
    def display
         print " #{@value} "
    end
end
class Element < Node
    def initialize(label, children)
         @label = label
         @children = children #children es un array de Node
    end
    def display
         print "< #{label} >"
         @children.each {|node| node.display}
             print "</ #{label} >"
    end
```

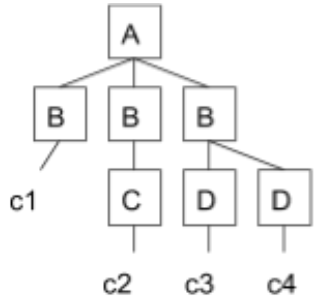


Podemos construir el árbol de abajo hacia arriba

n1 = Element.new('C', [Leaf.new('c2')])
n2 = Element.new('D', [Leaf.new('c3')])
n3 = Element.new('D', [Leaf.new('c4')])

n4 = Element.new('B', [Leaf.new('c1')])

n5 = Element.new('B', [n1])
n6 = Element.new('B', [n2, n3])
n7 = Element.new('A', [n4, n5, n6])



n7.display produce el documento xml asociado
< A > < B > c1 </B> <B> <C> c2 </C> </B> <B> <D> c3 </D> <D> c4 </D> </B> </A>

Ejercicio 4

Se quiere simular el funcionamiento de varios posibles negocios de venta de comida: una pizzería, una hamburguesería y una ensaladería. Para simplificar supongamos que en cada caso solo se fabrican dos productos:

- Pizzería: pizza de pepperoni y pizza vegetariana
- Hamburguesería: regular y not_meat
- Ensaladería: cesar y mediterranea

Queremos sacar partido del patrón Abstract Factory y para ello se pide

a) Definir tres fábricas abstractas: una para las pizzas una para las hamburguesas y una para las ensaladas. Además de escribir el código de las fábricas escriba el código de los productos (lo más simple posible)

b) Definir una clase Negocio con un método llamado simular que recibe una fábrica y el número de productos de cada tipo y procede a hacer la simulación como muestra el ejemplo. Su código debe funcionar exactamente de la forma que se muestra a continuación

```
Negocio.new(Hamburgueseria.new,3,1)).simular
hamburguesa regular 1 saliendo
hamburguesa regular 2 saliendo
hamburguesa regular 3 saliendo
hamburguesa not_meat 1 saliendo

Negocio.new(Pizzeria.new,2,3)).simular
pizza peperoni 1 saliendo
pizza peperoni 2 saliendo
pizza vegetariana 1 saliendo
pizza vegetariana 2 saliendo
pizza vegetariana 3 saliendo

Negocio.new(Ensaladeria.new,4,1)).simular
ensalada cesar 1 saliendo
ensalada cesar 2 saliendo
ensalada cesar 3 saliendo
ensalada cesar 4 saliendo
ensalada mediterranea 1 saliendo
```

Las fábricas son las clases Pizzeria, Hamburguesería y Ensaladería
Los productos concretos son las clases Pepperoni, Vegetarian, Regular, Not_meat, Cesar y Mediterranean.

```
class Pizzeria
    def new_p1(number)
        Pepperoni.new(number)
    end
    def new_p2(number)
        Vegetarian.new(number)
end end

class Hamburgueseria
    def new_p1(number)
        Regular.new(number)
    end
    def new_p2(number)
        Not_meat.new(number)

class Ensaladeria
    def new_p1(number)
        Cesar.new(number)
    end
    def new_p2(number)
Mediterranean.new(number) end

end
```

```
class Pepperoni
    def initialize (number)
        @name = 'pizza peperoni ' + number
    end
    def reveal
        return @name + ' saliendo'
end end
class Vegetarian
    def initialize (number)
      @name = 'pizza vegetariana ' + number
    end
    def reveal
        return @name + ' saliendo'
end end
class Regular
    def initialize (number)
      @name = 'hamburguesa regular ' + number
    end
    def reveal
        return @name + ' saliendo'
    end
end
class Not_meat
    def initialize (number)
         @name = 'hamburguesa not_meat ' + number
     end

     def reveal
        return @name + ' saliendo'
     end
end
class Cesar
    def initialize (number)
        @name = 'ensalada cesar ' + number
    end
    def reveal
        return @name + ' saliendo'
     end
end
class Mediterranean
    def initialize (number)
         @name = 'ensalada mediterranea ' + number
    end
    def reveal
        return @name + ' saliendo'
     end
end
```

```ruby
class Negocio
    def initialize(meal_factory, number_product_1, number_product_2)
        @the_factory = meal_factory
        @p1s = []
        number_product_1.times do |i|
            p1 = @the_factory.new_p1("#{i+1}")
            @p1s << p1
        end
        @p2s = [] number_product_2.times do |i|
            p2 = @the_factory.new_p2("#{i+1}")
            @p2s << p2
        end
    end
    def simular
        @p1s.each {|p1| puts(p1.reveal)}
        @p2s.each {|p2| puts(p2.reveal)}
    end
end

(Negocio.new(Hamburgueseria.new,3,1)).simular

(Negocio.new(Pizzeria.new,2,3)).simular

(Negocio.new(Ensaladeria.new,4,1)).simular
```