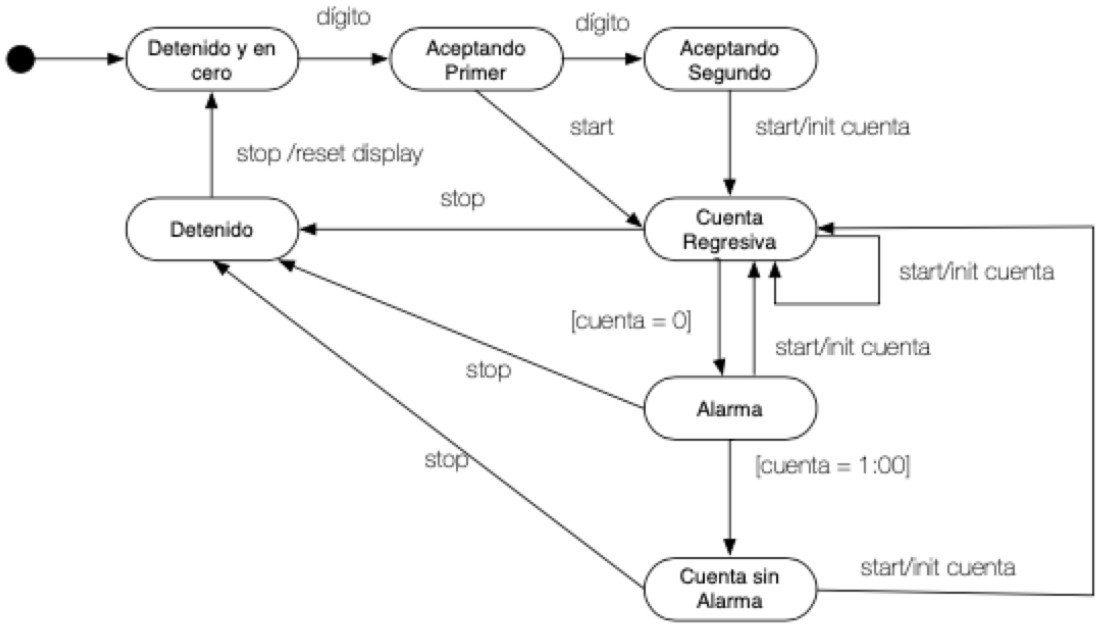
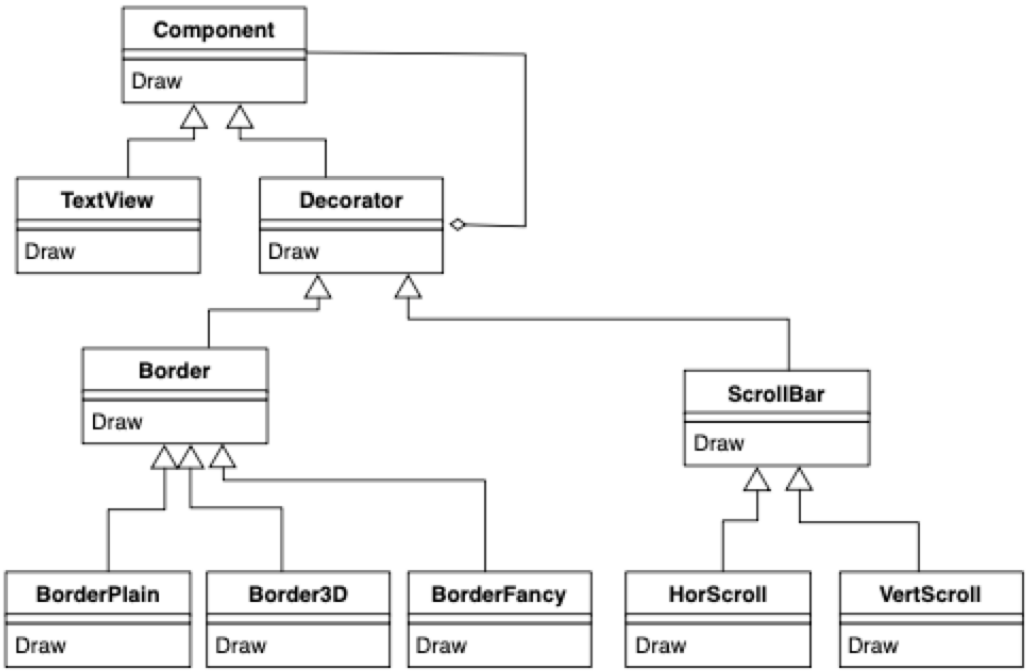
IIC 2143 Interrogación 2
Semestre 2021-1
Profesores M. Peralta y J. Navón

1. Las transiciones son ocasionadas por la acción de un botón salvo las de cuenta regresiva a alarma y de alarma a cuenta sin alarma que son espontáneas cuando se cumple la condición. Hay 5 transiciones que incluyen una acción. En cuatro casos para inicializar el contador al valor ingresado con los dígitos y en el otro para resetear el display



2.

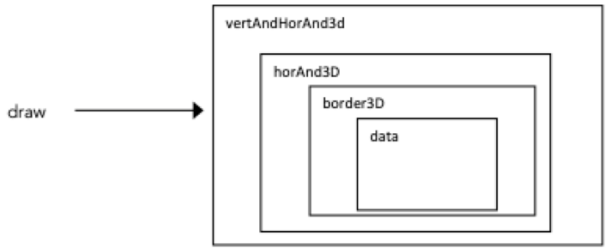
a) La clave del patrón decorador es tener una clase (decorator) que es una componente (especialización) y contiene una componente. Esto permite construir un objeto con capas y capas de decoración alrededor de él. En este caso tenemos además una jerarquía de decoradores que se agrupan en decoradores de borde y decoradores de scroll bars.



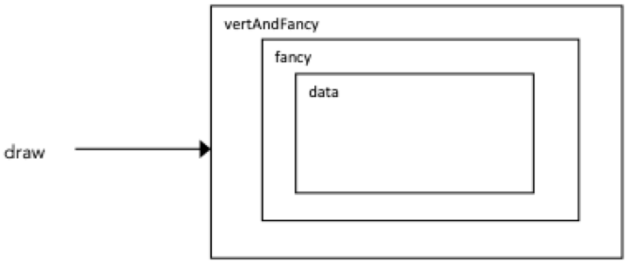
Método Draw de los decoradores simplemente invocan primero el draw de ls componente que contienen y luego dibujan lo específico del decorador según sea un borde o una barra de scroll

Esta solución es mejor porque agregar un nuevo tipo de borde simplemente implica escribir una clase. En la solución propuesta tendría que escribir 3 (una para cada tipo de barras de scroll). En un caso más complejo la explosión de clases podría ser mucho más extrema.

b) La figura ilustra la estrategia a seguir. La clave es generar objetos que incluyan los decoradores deseados




```
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
```

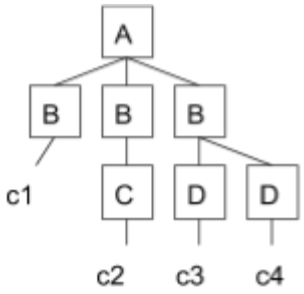
3.

a) Estructura de la solución usando el patrón Composite como un diagrama de clases UML

```
class Node
   def initialize()
   end
   def display()
   end
end


class Leaf < Node
   def initialize(value)
      @value = value
   end
   def display
      print " #{@value} "
      end
   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
end
```



b) 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
```

4.

a) El objetivo de esta pregunta es que el alumno modele un problema, reconociendo las relaciones que existen entre las clases y las restricciones de negocio. Las relaciones se traducen en la asociaciones en Rails, y las restricciones en las validaciones. Es requisito que el alumno haya ocupado las clases descritas para solucionar el problema. Si agregó más clases no hay ningún problema.

Distribución de Puntajes:

Clases y migraciones implementadas:
Por cada clase implementada (sin considerar asociaciones ni validaciones) se suma 0.5 puntos, y por cada migración se suma otros 0.5 puntos. Entonces tenemos:

- (1 punto) Client
- (1 punto) DeliveryWorker
- (1 punto) Order
- (1 punto) Pizza
- (1 punto) PizzaPart
- (1 punto) Ingredient
- (1 punto) Solución a la relación entre PizzaPart e Ingredient (Una clase PizzaPartIngredient por ejemplo)

Asociaciones implementadas:

- (1 punto) Un cliente puede pedir muchas órdenes, y una orden pertenece a un cliente.
- (1 punto) Un repartidor puede entregar muchas órdenes, y una orden es entregada por un repartidor.
- (1 punto) Una orden puede tener muchas pizzas, y una pizza pertenece a una orden.
- (1 punto) Una pizza puede tener muchas partes, y cada parte sólo pertenece a una pizza.
- (2 punto) Una parte puede tener muchos ingredientes, y un ingrediente puede pertenecer a muchas partes de pizza.

Las validaciones mínimas necesarias:

- (1 punto) Identificar el cliente y dirección: Un rut o nombre + dirección.
- (1 punto) Identificar al repartidor: Un rut o nombre + tipo de vehículo.
- (1 punto) Una orden debe tener al menos una pizza.
- (1 punto) Una pizza debe estar compuesta por 1, 2, 4 o 8 partes.
- (1 punto) La pizza debe tener tipo de pizza o tamaño, tipo de masa y tipo de queso.
- (1 punto) Una parte de pizza (PizzaPart) debe tener al menos un ingrediente.
- (1 punto) Identificar el ingrediente por un nombre.

Puntaje total parte a) (20 puntos)

b) El objetivo de esta pregunta es que el alumno pueda obtener información de valor para un cliente con la solución propuesta en la parte a). Si por alguna razón no completó la parte a), se puede corregir asumiendo que existen las clases mencionadas en el enunciado y con los supuestos que el alumno haya dejado explícitamente.

i) (3 puntos) ¿Qué ingrediente es el más pedido?
ii) (2 puntos) ¿Cuantos pedidos llevan bebidas o aperitivos?
iii) (2 puntos) ¿Cuánto dinero ha ganado con pedidos que sólo fueron repartidos por repartidores en motos?

Puntaje total parte b) (7 puntos)

c) El objetivo de esta pregunta es que el alumno reconozca cuál es la parte crítica de su código e implemente uno o varios tests que prueben ese código más vulnerable.

i) (1 punto) Es código crítico
ii) (1 punto) El test está correctamente implementado
iii) (1 punto) El test prueba todos los posibles casos para ese método, callback o validación

Puntaje total parte c) (3 puntos)