Se quiere construir una aplicación web sencilla que sirva para elaborar y tomar pruebas de selección múltiple en línea. Todas las preguntas son de alternativas e incluyen un enunciado, una lista de opciones y opcionalmente una figura. La prueba puede tener un número variable de preguntas. El profesor asigna además un tiempo máximo para responder la prueba.

Una vez que el profesor ha elaborado su prueba, éste la puede poner en línea indicando fecha y hora en que estará disponible. Un alumno puede acceder a la prueba a partir del momento indicado y desde que indica que ha comenzado dispone del tiempo que indicó el profesor. El alumno responde las preguntas marcando la opción correcta y pasa a la siguiente pregunta. En cualquier momento debería poder revisar o cambiar una respuesta ya dada. EL estudiante puede decidir dar por terminado su trabajo en cualquier momento aunque todas formas será forzado a hacerlo al completarse el tiempo (se le da un aviso 10 y 5 minutos antes).

a) Construya un modelo de dominio para este problema expresado en forma de diagrama de clases UML. No olvide incluir adornos especiales en asociaciones y también sus cardinalidades (1..*, etc). Explique brevemente su modelo

b) Construya un modelo de dominio alternativo para el mismo problema. Debe ser significativamente distinto al anterior y comparable en términos de su calidad. Explique brevemente su modelo

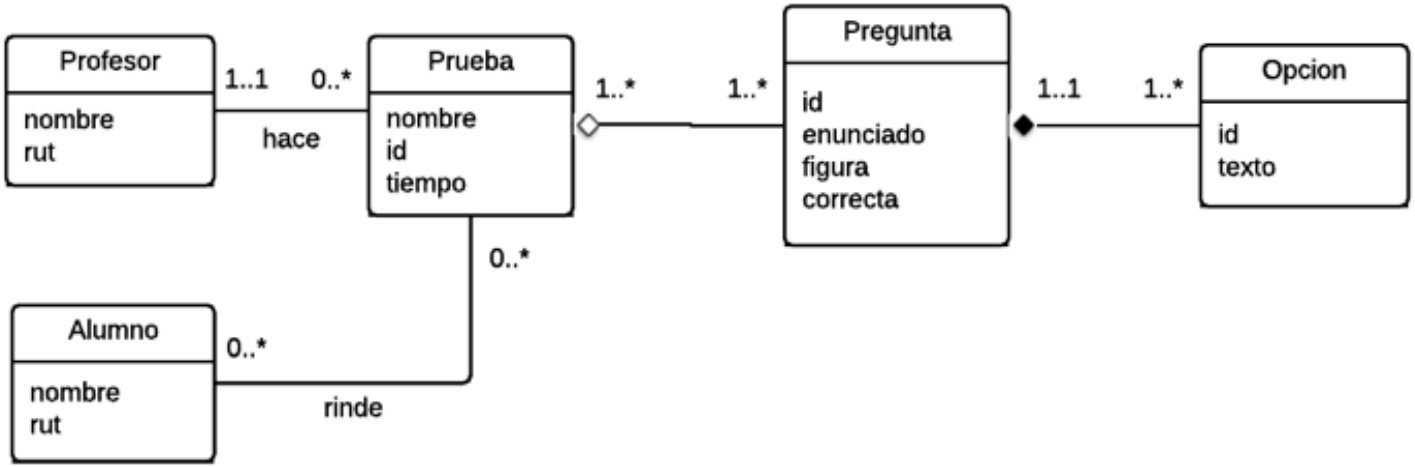
c) Discuta ventajas y desventajas entre los modelos a) y b)

Profesores hacen las pruebas y los alumnos las rinden.

Las pruebas tienen a lo menos una pregunta (en general mas de una) y ellas pueden ser reusadas en otras pruebas.

Las preguntas incluyen un enunciado (texto) una figura y un número entero que indica la opción correcta (1 - A, 2 - B, etc)

Las opciones tienen un texto y no pueden ser reusadas en distintas preguntas

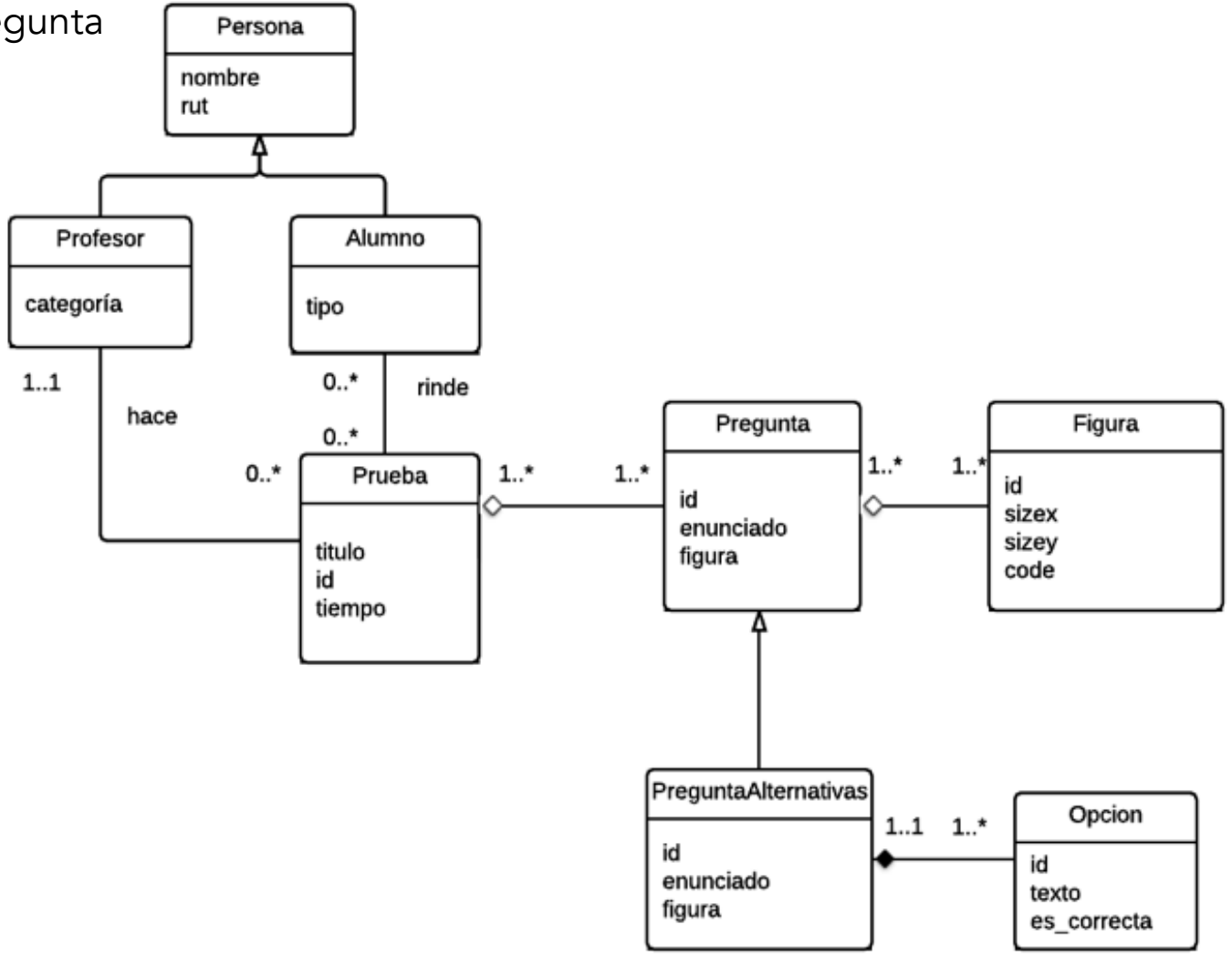


Ahora los profesores y alumnos son especializaciones de persona.

Las pruebas incluyen preguntas pero ellas pueden ser de distintos tipos, en particular de selección múltiple.

Las figuras ahora son objetos separados y pueden ser reutilizados en distintas preguntas.

Las preguntas de selección múltiple tienen asociado una serie de opciones las cuales incluyen un campo boolean que indica si es o no la opción correcta. Ello hace innecesario entonces el atributo que lo indica en la pregunta



A continuación se muestra código Ruby que implementa dos de los mejores algoritmos de ordenamiento conocidos: QuickSort y MergeSort.

```
def quicksort arr
 return[]if arr.length == 0
 x,*xs=*arr
 smaller,bigger = xs.partition {|other| other<x}
 quicksort(smaller) + [x] + quicksort(bigger)
end

def mergesort array
 if array.length<=1
   return array
 end
 middle = array.length/2
 left = array[0...middle]
 right = array[middle...array.length]
 left = mergesort left
 right = mergesort right
 return merge(left,right)
end

def merge left,right
 result = []
 while left.length>0 and right.length>0
   left.first<=right.first ? result<<left.shift:result<<right.shift
 end
 result.push *left if left.length>0
 result.push *right if right.length>0
 return result
end
```

a) Escribir (Ruby) una solución que sirva para ordenar con cualquiera de estos dos algoritmos sacando partido del patrón Strategy estudiado en clases. Al no especificar el método de ordenamiento debería asumir uno por defecto. Escriba unas líneas de código que muestre su solución en acción.

b) Escribir (Ruby) una solución que permita ordenar usando cualquiera de estos dos algoritmos si Ud. no conociera nada de patrones y es la primera vez que enfrenta el problema. Escriba unas líneas de código que muestre su solución en acción.

c) Discuta las ventajas de la solución basada en el patrón Strategy en comparación a la segunda

a) Patrón estrategia simple

```
class QuickSort
 def sort arr
   ...
 end
end

class MergeSort
 def sort array
   ...
 end

 def merge left,right
   ...
 end
end

class Sorter
 def self.sort arr,strategy
   theStrategy = strategy
   thestrategy.sort(arr)
 end
end
```

Para incluir una estrategia default hay que modificar ligeramente la clase Sorter ...

```
class Sorter
 @@default_strategy=QuickSort.new
 def self.sort arr,strategy=nil
   strategy ||= @@default_strategy
   strategy.sort(arr)
 end
end
```

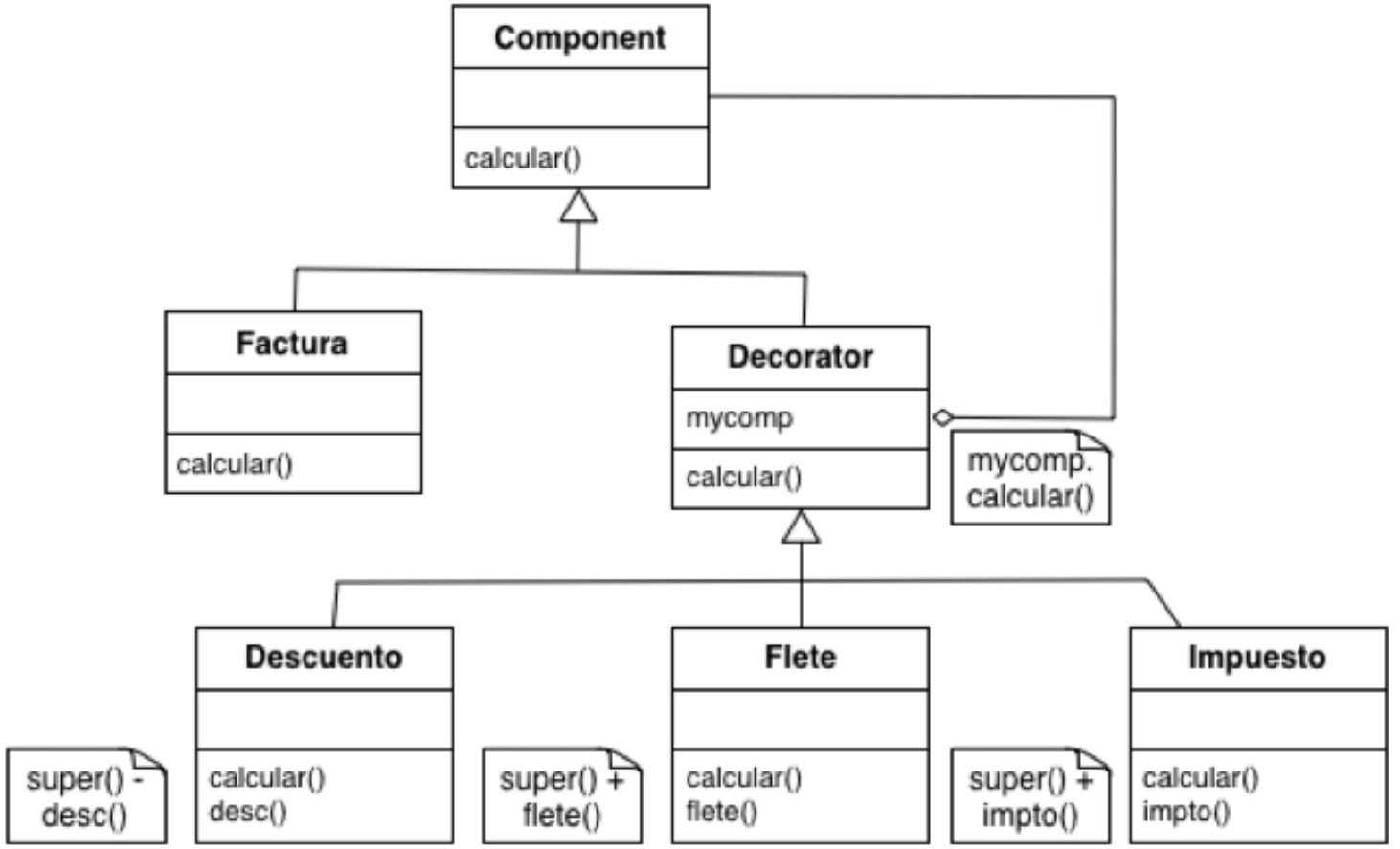
Ahora podemos hacer :

```
theArray = [3 5 2 6 8 9 3 5 8 2 ]
theSort = Sorter.new(theArray, MergeSort.new)
theSort = Sorter.new(theArray, QuickSort.new)
```

Se le ha pedido diseñar el código que genera la factura de una tienda que vende libros online. El cálculo de la factura tiene una pequeña complicación ya que una vez totalizado el monto correspondiente a los libros puede incluir una combinación cualquiera de las siguientes 3 cosas dependiendo del cliente, el país y el monto de la compra: un descuento, un flete y un impuesto. Estos elementos además pueden operar en cualquier orden.

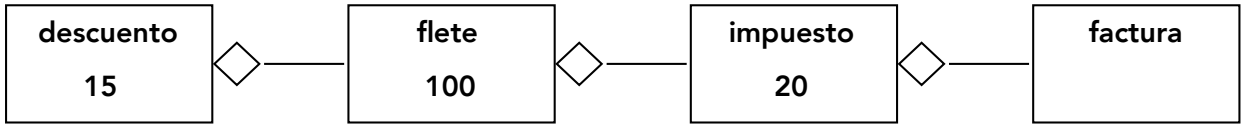
Ejemplo: $100 + 20% impuesto + 100 flete -15% descuento
$200 - 20% descuento + 10% impuesto + 50 flete

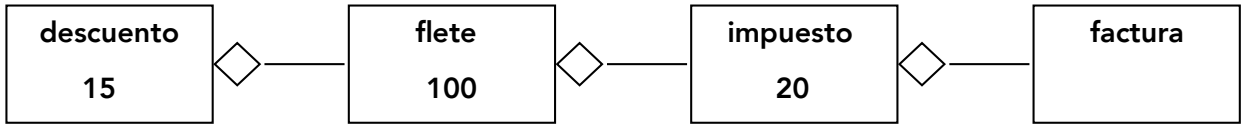
Suponga que dispone de una clase factura con un método calcular() que entrega el valor bruto de los libros. Use un patrón de diseño estudiado en clases de modo de tener esa flexibilidad para el cálculo del total dependiendo de la situación. Dibuje un diagrama de clases que ilustre la solución y luego dibuje un diagrama de secuencia que ilustre el funcionamiento para el primero de los dos ejemplos.

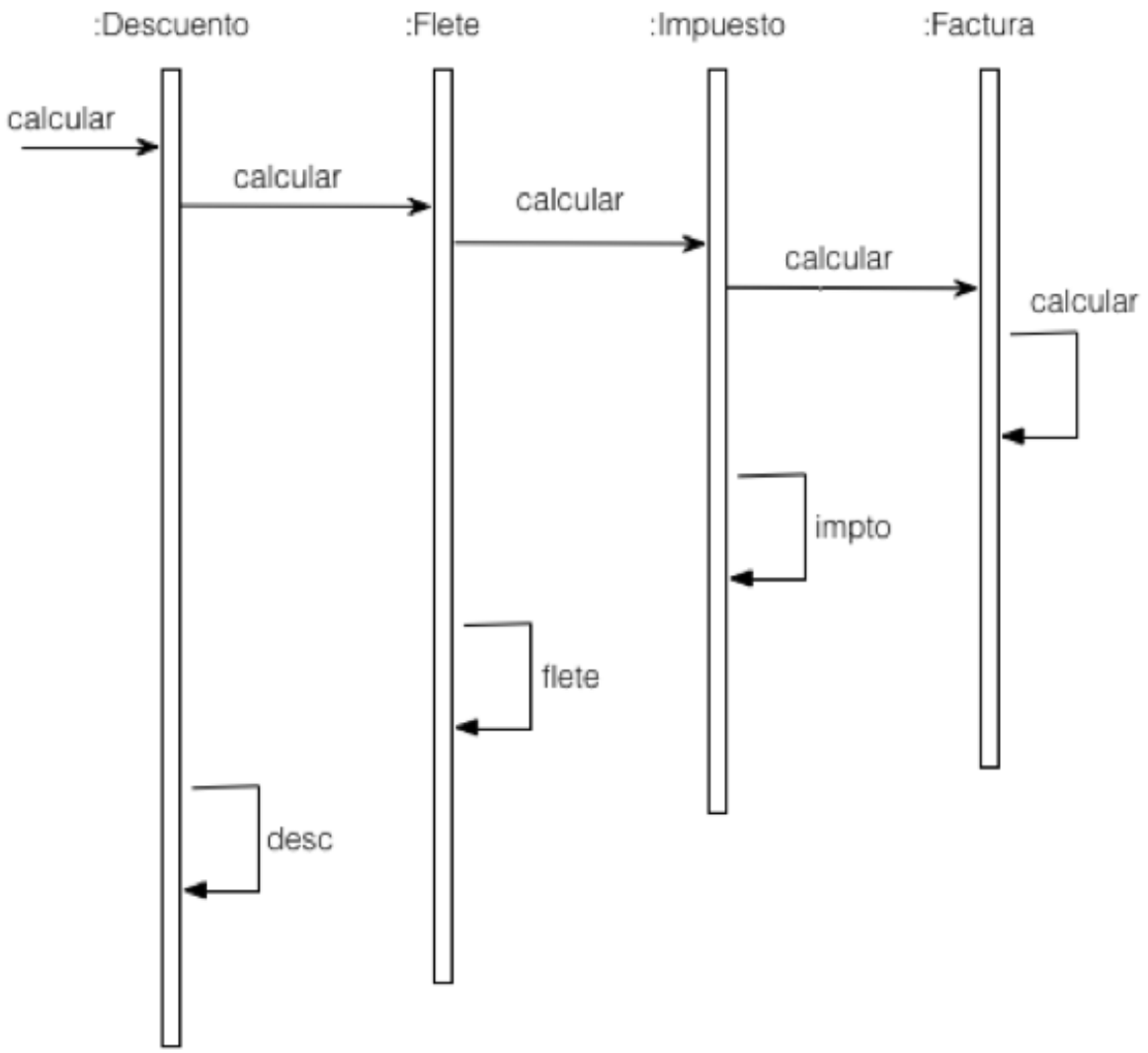


Para el ejemplo 1 el objeto a recibir el mensaje calcular debe ser construido como un descuento que contiene un flete que contiene un impuesto que contiene finalmente la factura. De este modo el mensaje calcular ejecuta primero el calcular del objeto interno hasta llegar a la factura. El diagrama de secuencia lo muestra mejor ...

x = Descuento.new(15, (Flete.new(100, Impuesto.new(20, factura)))))




descuento
15
flete
100
impuesto
20
factura


:Descuento
:Flete
:Impuesto
:Factura
calcular
calcular
calcular
calcular
calcular
impto
flete
desc

Se pide modelar mediante un diagrama de estados el comportamiento de una cuenta vista bancaria típica.

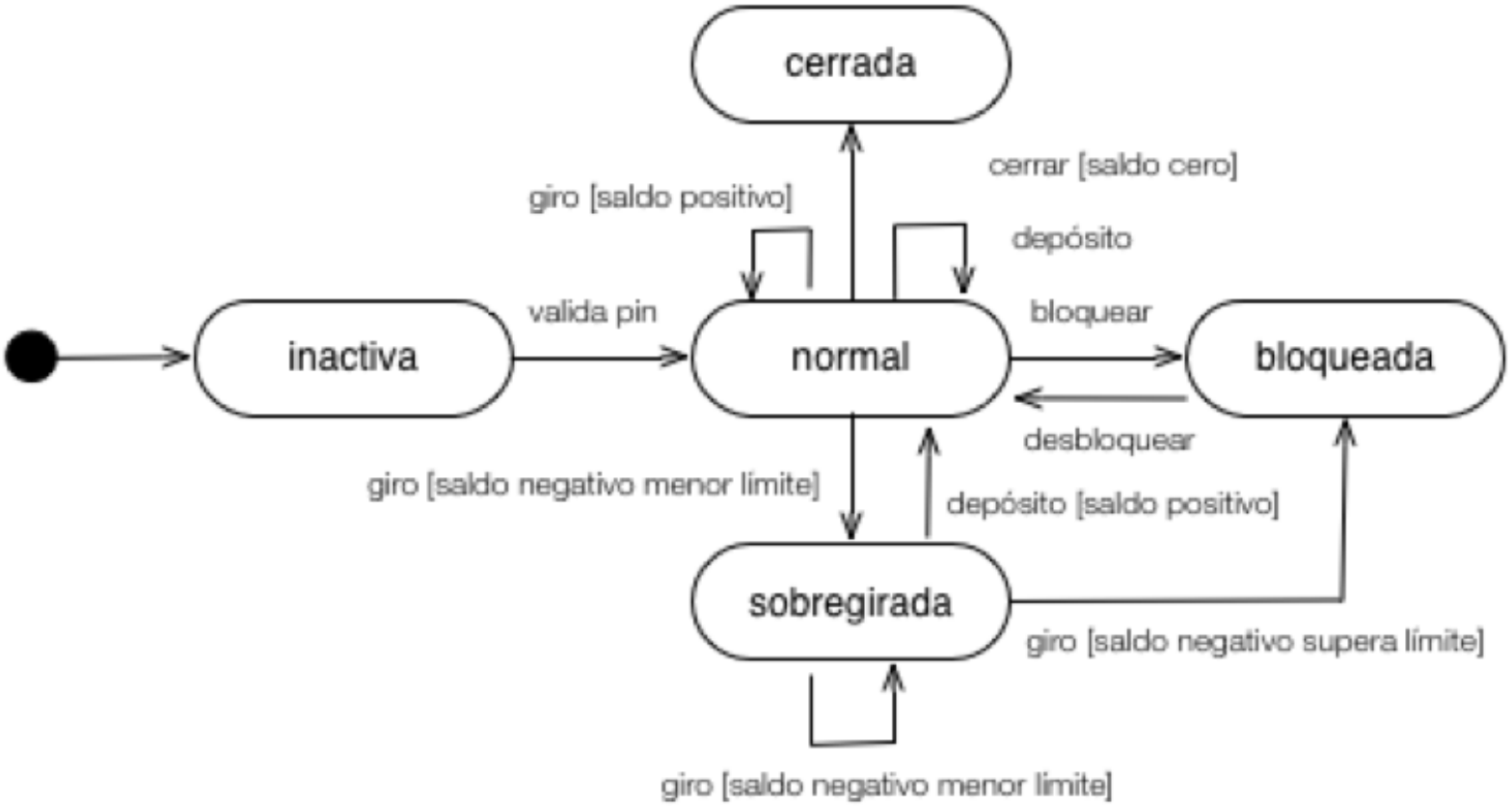
Una de estas cuentas primeramente debe ser creada con los datos del cliente y permanece así hasta que el cliente valida su PIN (en un cajero o vía web) después de lo cual pasa a estar activa.

Una cuenta activa admite depósitos y retiros. Si un retiro es mayor que el saldo pero menor que lo disponible en la línea de crédito la cuenta queda sobregirada pero se puede seguir usando.

Al llegar al tope de la línea de crédito la cuenta se bloquea y no permite nuevos retiros hasta que se hagan nuevos depósitos.

En cualquier momento la cuenta puede ser bloqueada por razones de seguridad (robo de la tarjeta) o porque el banco lo decide unilateralmente.

El cliente puede decidir cerrar la cuenta solo si ella no está bloqueada y tiene el saldo en cero.


cerrada
cerrar [saldo cero]
giro [saldo positivo]
depósito
valida pin
bloquear
inactiva
normal
bloqueada
desbloquear
giro [saldo negativo menor límite]
depósito [saldo positivo]
sobregirada
giro [saldo negativo supera límite]
giro [saldo negativo menor límite]

Ud. debe escribir las especificaciones para el software de un ascensor. Este está destinado a ser instalado en edificios en que habrá un solo ascensor (para simplificar el problema). A continuación, la explicación que le han dado sobre el comportamiento del ascensor.

- En un momento dado el ascensor puede estar esperando, subiendo o bajando
- Cuando el ascensor va hacia arriba no cambia de dirección hasta agotar todas las solicitudes pendientes a pisos en esa dirección. Lo contrario ocurre cuando el ascensor se mueve hacia abajo.
- El ascensor es capaz de detectar cuando llega a un nuevo piso
- Cuando no hay ya solicitudes en la dirección de movimiento el ascensor cambia de dirección y comienza a atender las pendientes en la otra dirección. Si no hay, se detiene.
- Las solicitudes se generan cuando alguien presiona un botón en el ascensor o cuando alguien "llama" al ascensor desde un piso dado.
- La llamada puede ser de dos formas: para subir y para bajar. Dependiendo de ello el ascensor los agregará a sus solicitudes pendientes.

Tiempo	Piso	Estado	Evento
1	0	detenido	botón 4
2	0	detenido	botón 6
3	0	inicia subida	
4	1	subiendo	
5	2	subiendo	
6	3	subiendo	llamada para subir desde el 7
7	4	detenido	botón 8
8	4	reanuda subida	
9	5	subiendo	llamada para bajar desde el 4
10	6	detenido	
11	6	reanuda subida	
12	7	detenido	
13	7	reanuda subida	
14	8	detenido	
15	8	inicia bajada	
16	7	bajando	
17	6	bajando	
18	5	bajando	
19	4	detenido	botón 3
20	4	reanuda bajada	
21	3	detenido (ascensor queda en espera)	

Estados

MU moviendose hacia arriba MD moviendose hacia abajo
DE detenido en espera
DU detenido hacia arriba DD detenido hacia abajo

Eventos

boton(n) - alguien marcó el piso n en el ascensor
llamada (d, n) - una llamada desde el piso n para subir (d=1) o para bajar (d=0)
piso(n) - llegada al piso n

Vamos a suponer que el ascensor maneja dos listas: una para las detenciones pendientes hacia arriba (U) y una con detenciones pendientes hacia abajo (D)

Examinemos que sucede cuando esos eventos ocurren en los estados anteriores

MU

De este estado solo se puede salir con un evento de piso
botón (n) - Si n > piso actual insertar n en la lista U, en caso contrario en la lista D
llamada(d, n) - si d=1 insertar n en la lista U en caso contrario insertar n en lista D
piso(n) - si n está en U, eliminar n de la lista y pasar a DU

MD

De este estado solo se puede salir con un evento de piso
botón (n) - Si n < piso actual insertar n en la lista D, en caso contrario en la lista U
llamada(d, n) - si d=1 insertar n en la lista U en caso contrario insertar n en lista D
piso(n) - si n está en D eliminar n de la lista y pasar a detenido hacia abajo

DD

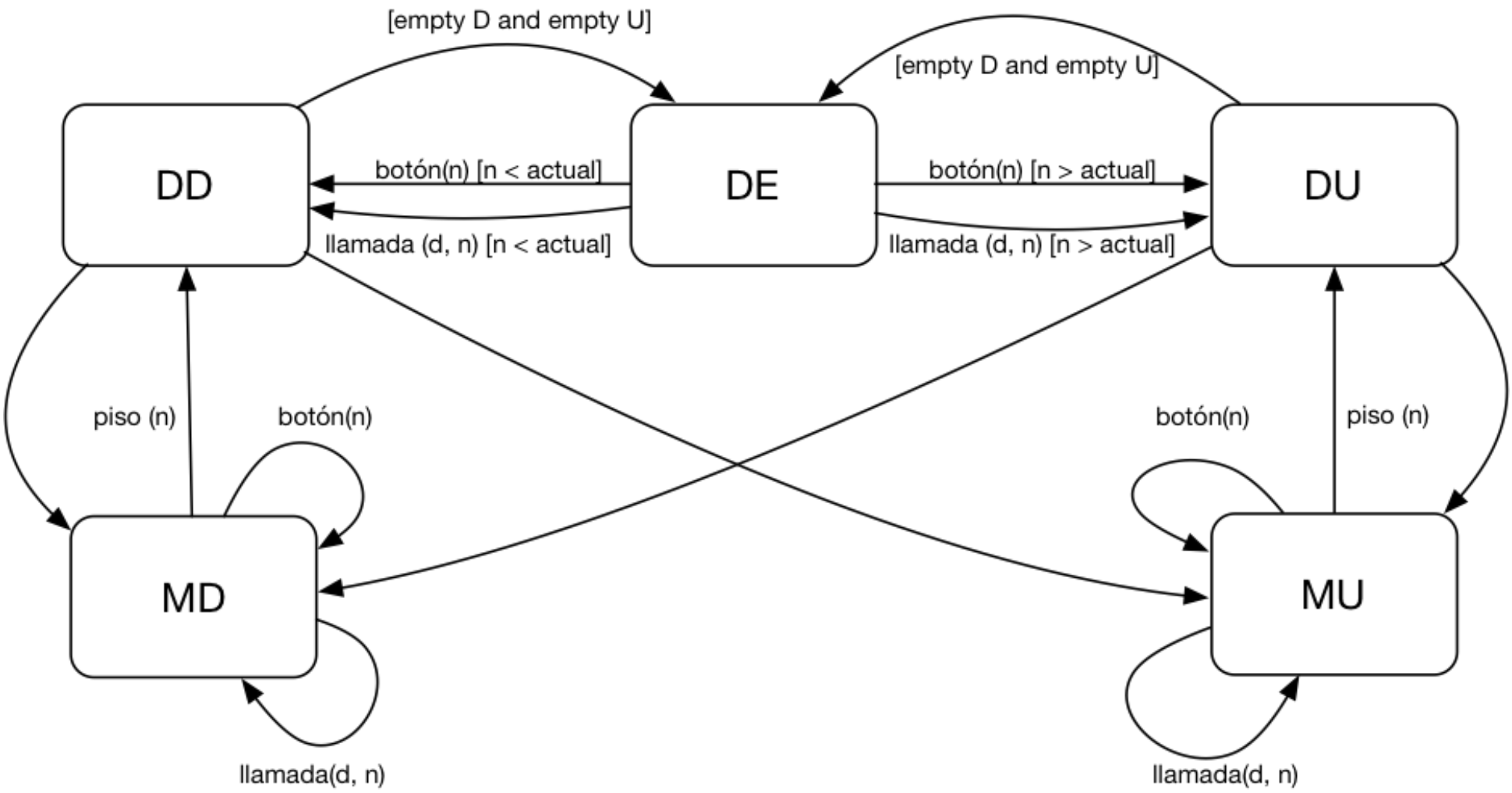
Hay una transición espontánea hacia MD, hacia MU o hacia DE dependiendo de la condición
Si D y U estan vacías pasar a detenido en espera. Si hay en D algún número menor que piso actual pasar a moviendose hacia abajo. En caso contrario si U no está vacía pasar a moviendose hacia arriba.

DU

Hay una transición espontánea hacia MD, hacia MU o hacia DE dependiendo de la condición
Si D y U estan vacías pasar a detenido en espera. Si hay en D algún número mayor que piso actual pasar a moviendose hacia arriba. En caso contrario si D no está vacía pasar a moviendose hacia abajo

DE

Puede pasar a DD o a DU ya sea con evento de botón o evento de llamada según la condición
botón (n) - Si n > piso actual insertar n en la lista U, y pasar a detenido hacia arriba, en caso contrario insertar en la lista D y pasar a detenido hacia abajo
llamada(d, n) - si d =1 insertar n en la lista U, en caso contrario en la lista D. Si n>actual pasar a detenido hacia arriba, en caso contrario detenido hacia abajo


[empty D and empty U]
[empty D and empty U]
DD
botón(n) [n < actual]
DE
botón(n) [n > actual]
DU
llamada (d, n) [n < actual]
llamada (d, n) [n > actual]
piso (n)
botón(n)
botón(n)
piso (n)
MD
MU
llamada(d, n)
llamada(d, n)