Definiciones

**Ingeniería de software:** es la aplicación de un proceso sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software.

**Modelo**: una representación o especificación desde un determinado punto de vista y con un objetivo en concreto.

**Diseño**: conjunto de planes y decisiones para definir un producto con los suficientes detalles como para permitir su realización física de acuerdo a unos requisitos.

**Patrón de diseño:** solución probada que se puede aplicar con éxito a un tipo de problemas que aparecen repetidamente en el desarrollo de software.

**Diferencias entre estilo arquitectónico y patrón de diseño:**

La principal diferencia entre ambos es que el patrón de diseño te ayuda a solucionar un problema con éxito en parte de un sistema, mientras que el estilo arquitectónico es la estructura que siguen los componentes en alto nivel. El estilo arquitectónico representa la estructura de alto nivel, mientras que el patrón de diseño es la especificación de una solución probada en un problema a resolver.

**BadSmell, refactoring:**

El BadSmell son fragmentos de código de tu proyecto que son feos de ver y están de algún modo mal. El proceso de refactor empieza con el badsmell, una vez has investigado, y has identificado el problema, rehaces la estructura del código. Una vez has terminado, está todo listo para volver a realizar refactor.

3 procesos de refactor: (ME HE QUEDADO SIN TIEMPO AQUI, CHUSCLALA)

Lo voy a acabar pero por estudiar

Extraer en una clase un método cuyo contenido sea muy extenso.

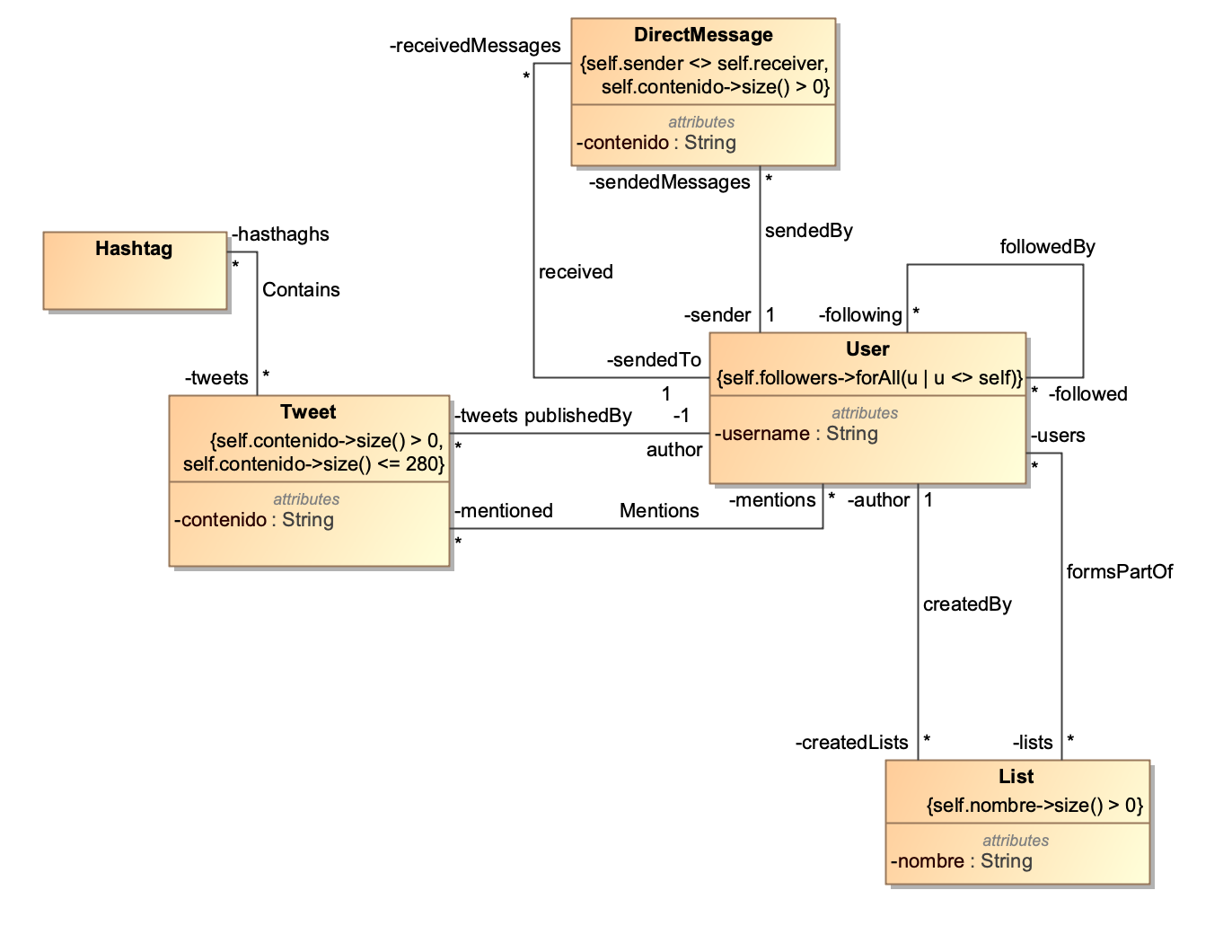
Extraer código duplicado y “hacer factor común”.

Cadenas de mensajes muy largas.

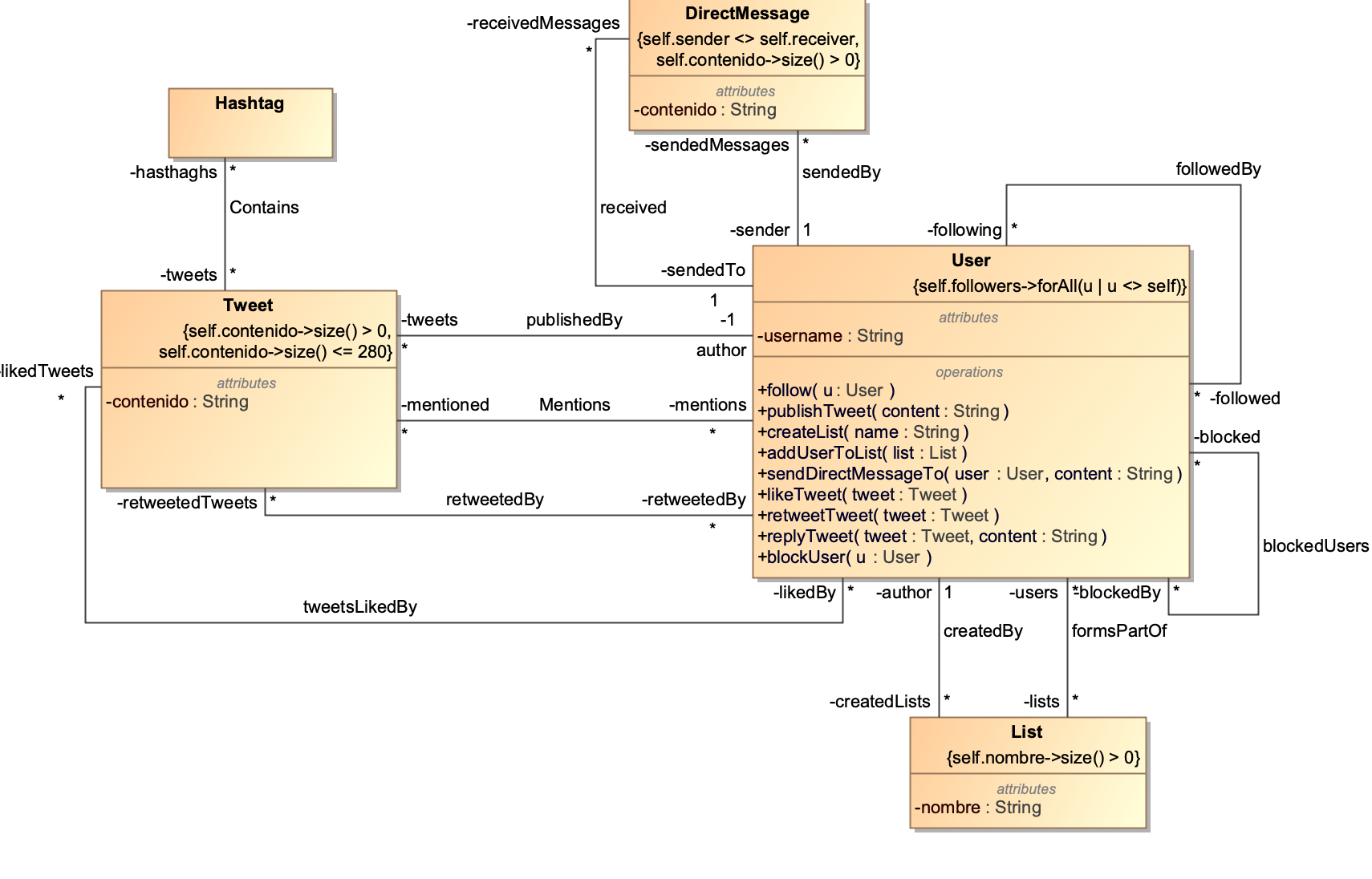
**Ejercicio 1:**

Metamodelo de twitter

1. Como en twitter puedes ser seguido y seguir a muchos usuarios, la cardinalidad de la relación reflexiva es \* en ambos roles. En un tweet pueden ser mencionados todos los usuarios (hasta que se acaben los 280 caracteres). Ocurre lo mismo con los hashtaghs, pueden haber tantos hashtaghs como permita la longitud de los caracteres, y los hashtaghs pueden estar presentes en muchos tweets. Un usuario puede formar parte en más de una lista. Los Mensajes directos son enviados por un único usuario a otro único usuario, por tanto, desarrollamos 2 relaciones entre MensajeDirecto para modelar quién los recibe y quién los envía. En cuanto a las restricciones, hemos añadido que un usuario no pueda seguirse a sí mismo, que los tweets y mensajes directos no sean vacíos y que los tweets no superen los 280 caracteres.



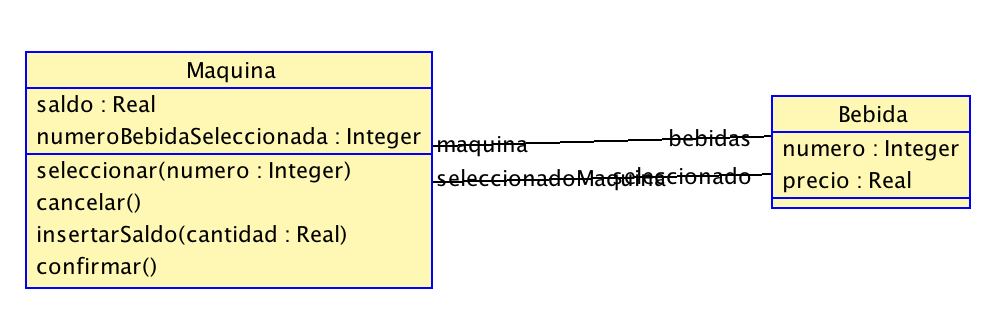
1. Operaciones para el metamodelo de twitter.



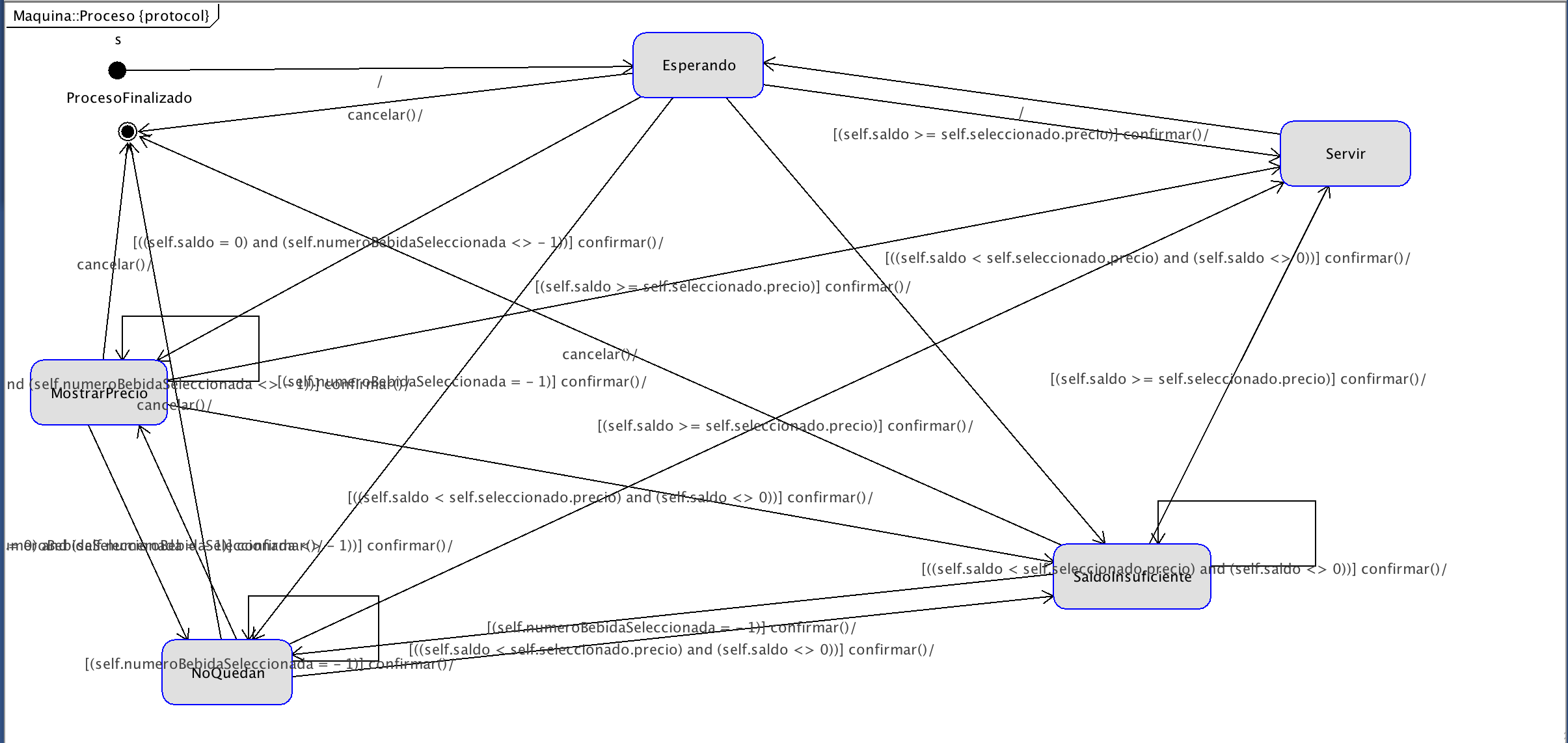
**Ejercicio 2:**

1. Para este ejercicio vamos a desarrollar operaciones en la clase Maquina para poder cambiar su estado y poder modelar su comportamiento. Para ello haremos uso de una operación llamada *confirmar()* que simula el botón de confirmar y así transitar entre estados dependiendo de los valores de los atributos actuales. Para poder simplificar las transiciones y las guardas, definiremos una relación entre Máquina y Bebida que represente la bebida seleccionada en un momento dado. Además, existirá otra relación que simula las bebidas que hay en un momento dado en la Máquina. Las bebidas tendrán un número y un precio. La Máquina guardará en todo momento el número de bebida seleccionado y el saldo que se encuentra en ese momento en la máquina. Para finalizar lo anterior, nuestra clase Máquina tendrá las siguientes operaciones, seleccionar una bebida en concreto con su número, cancelar la operación, e insertarSaldo, que nos permitirá aumentar el saldo de la máquina. Para poder definir correctamente la máquina de estados necesitaremos 5 estados. El estado inicial, en el que la máquina está esperando, el estado en el que nos muestra el precio cuando se selecciona una bebida y no hay saldo (MostrarPrecio), otro estado que representa que no hay saldo suficiente para esa bebida (SaldoInsuficiente), un estado para comunicar que no quedan bebidas del tipo seleccionado (NoQuedan), un estado que representa haber completado todo el proceso y el servicio de la bebida llamado Servir y para finalizar, un estado ProcesoFinalizado que representa haber terminado con todo el proceso.

Este es el diagrama UML del sistema



Y este el diagrama de estados con sus transiciones y sus guardas:



Definición de la máquina de estados:

psm Proceso

states

s: initial

Esperando

MostrarPrecio

NoQuedan

SaldoInsuficiente

Servir

ProcesoFinalizado: final

transitions

s -> Esperando

Esperando -> MostrarPrecio {[self.saldo = 0 and self.numeroBebidaSeleccionada <> -1] confirmar()}

Esperando -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

Esperando -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

Esperando -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

MostrarPrecio -> MostrarPrecio {[self.saldo = 0 and self.numeroBebidaSeleccionada <> -1] confirmar()}

MostrarPrecio -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

MostrarPrecio -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

MostrarPrecio -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

MostrarPrecio -> ProcesoFinalizado {cancelar()}

NoQuedan -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

NoQuedan -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

NoQuedan -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

NoQuedan -> ProcesoFinalizado {cancelar()}

NoQuedan -> MostrarPrecio {[self.saldo = 0 and self.numeroBebidaSeleccionada <> -1] confirmar()}

SaldoInsuficiente -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

SaldoInsuficiente -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

SaldoInsuficiente -> ProcesoFinalizado {cancelar()}

SaldoInsuficiente -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

Servir -> Esperando

Esperando -> ProcesoFinalizado {cancelar(}

end

**fichero USE completo:**

model MaquinaBebidas

class Bebida

attributes

numero: Integer

precio: Real

end

class Maquina

attributes

saldo: Real init: 0

numeroBebidaSeleccionada: Integer init: -1

operations

seleccionar(numero: Integer)

begin

if self.bebidas->select(b | b.numero = numero)->size() > 0 then

if self.seleccionado->size() > 0 and self.bebidas->select(b | b.numero = numero)->size() > 0 then

delete (self, self.seleccionado) from Seleccionado;

self.numeroBebidaSeleccionada := numero;

insert(self, self.bebidas->select(b | b.numero = numero)->asSequence()->first()) into Seleccionado;

else

self.numeroBebidaSeleccionada := numero;

insert(self, self.bebidas->select(b | b.numero = numero)->asSequence()->first()) into Seleccionado;

end

else

self.numeroBebidaSeleccionada := -1

end

end

cancelar()

begin

self.saldo := 0;

delete (self, self.seleccionado) from Seleccionado;

self.numeroBebidaSeleccionada := -1;

end

insertarSaldo(cantidad: Real)

begin

self.saldo := self.saldo + cantidad;

end

confirmar()

begin

if self.numeroBebidaSeleccionada <> -1 and self.saldo >= self.seleccionado.precio then

delete (self, self.seleccionado) from contiene;

delete (self, self.seleccionado) from Seleccionado;

self.saldo := 0;

else

self.saldo := self.saldo

end

end

statemachines

psm Proceso

states

s: initial

Esperando

MostrarPrecio

NoQuedan

SaldoInsuficiente

Servir

transitions

s -> Esperando

Esperando -> MostrarPrecio {[self.saldo = 0 and self.numeroBebidaSeleccionada <> -1] confirmar()}

Esperando -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

Esperando -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

Esperando -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

MostrarPrecio -> MostrarPrecio {[self.saldo = 0 and self.numeroBebidaSeleccionada <> -1] confirmar()}

MostrarPrecio -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

MostrarPrecio -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

MostrarPrecio -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

MostrarPrecio -> Esperando {cancelar()}

NoQuedan -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

NoQuedan -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

NoQuedan -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

NoQuedan -> Esperando {cancelar()}

NoQuedan -> MostrarPrecio {[self.saldo = 0 and self.numeroBebidaSeleccionada <> -1] confirmar()}

SaldoInsuficiente -> Servir {[self.saldo >= self.seleccionado.precio] confirmar()}

SaldoInsuficiente -> NoQuedan {[self.numeroBebidaSeleccionada = -1] confirmar()}

SaldoInsuficiente -> Esperando {cancelar()}

SaldoInsuficiente -> SaldoInsuficiente {[self.saldo < self.seleccionado.precio and self.saldo <> 0] confirmar()}

Servir -> Esperando

end

end

association Seleccionado between

Maquina [0..1] role seleccionadoMaquina

Bebida [0..1] role seleccionado

end

association contiene between

Maquina [1] role maquina

Bebida [\*] role bebidas

end

1. Para este cambio de comportamiento, simplemente nos basta con modificar la operación confirmar y las transiciones de la siguiente manera, en la operación confirmar restar el saldo de la bebida seleccionada y volver al estado de Esperando. Con esta pequeña modificación, nuestro sistema es capaz de volver a empezar con el proceso con el saldo sobrante de la anterior operación. Si quisiera terminar el proceso, basta con darle al botón de cancelar().

**Ejercicio 3:**

1. Para evitar la creación de clases RedHopfield, etc, vamos a utilizar el patrón de método factoría, pero primero, necesitaremos crear un Enum con todos los tipos de neurona que hay en nuestro sistema. Nuestra factoría de neuronas tendrá un método llamado getNeurona dónde recibirá por parámetro el enum con el tipo de neurona a crear. Luego, nuestra clase Red, en su constructor, creará un objeto de tipo NeuronaFactory, y dependiendo de los parámetros que reciba, creará una red de un tipo de neuronas o de otro.

Clases en Java:

*package* examen2018V3;

*import* java.util.ArrayList;

*import* java.util.List;

*public class* Red {

*private* List<Neurona> neuronaList;

*public* Red(TipoNeuronas tipoNeuronas, *int* cantidadNeuronas) {

NeuronaFactory neuronaFactory = *new* NeuronaFactory();

neuronaList = *new* ArrayList<>();

*for*(*int* i = 0; i < cantidadNeuronas; i++) {

neuronaList.add(neuronaFactory.getNeurona(tipoNeuronas));

}

}

}

*package* examen2018V3;

*import* java.util.List;

*public abstract class* Neurona {

*private* List<Neurona> neuronas;

*public abstract void* aprender();

}

1. Para poder hacer la relación entre ambas Neuronas, creamos una clase Peso que guarde una referencia a la siguiente neurona y a la neurona de antes, y en Neurona ponemos el peso que le entra y el peso que le sale, como si fuera una Lista Enlazada pero con neuronas. Si en un momento dado el peso tiene una referencia a null de la neurona de antes, implica que es la primera neurona, de igual modo pero con la siguiente neurona, implica que es la última neurona. Se comprueba que el peso de la neurona esté entre 0 y 1, si no, se lanza una excepción.

*package* examen2018V3;

*public class* Peso {

*private float* peso;

*private* Neurona prevNeurona;

*private* Neurona nextNeurona;

*public* Peso(*float* peso, Neurona prevNeurona, Neurona nextNeurona) {

*if* (peso < 0 || peso > 1){

*throw new* RuntimeException("Peso inválido");

}

*this*.peso = peso;

*this*.prevNeurona = prevNeurona;

*this*.nextNeurona = nextNeurona;

*if*(*this*.prevNeurona != *null*)

*this*.prevNeurona.setNextPeso(*this*);

*if*(*this*.nextNeurona != *null*)

*this*.nextNeurona.setPrevPeso(*this*);

}

}

*package* examen2018V3;

*public abstract class* Neurona {

*private* Peso prevPeso;

*private* Peso nextPeso;

*public abstract void* aprender();

*void* setNextPeso(Peso peso) {

*this*.nextPeso = peso;

}

*public void* setPrevPeso(Peso peso) {

*this*.prevPeso = peso;

}

}