Práctica de Diseño Software

Daniel Sergio Vega Rodríguez, Carlos Torres Paz

Curso 2019-2020

1. Introducción

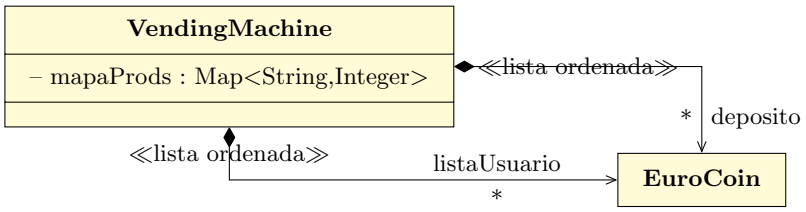
En este artículo se describen los diseños del software de una `máquina de vending` que usa dos tipos de devolución de cambio y de `acciones con una serie de clientes` interesados en ellas. El diseño es presentado a través de una serie de diagramas UML junto con una descripción de los principios y patrones de diseño aplicados para lograr una correcta implementación, susceptible de ser usada y extendida con facilidad, pragmatismo y eficiencia. Los diagramas **completos** se muestran al final del documento.

2. Diagrama de clases

2.1. Ejercicio 1

Las clases están organizadas alrededor de una clase principal que representa a la máquina de *vending*. **VendingMachine** tiene una relación de composición con **EuroCoin**, que es una clase *inmutable*[1] para evitar así una posible modificación externa de la misma. Además, semánticamente no tiene sentido que esta sea mutable. **EuroCoin** presenta una *multiplicidad* variable sin límites definidos respecto a **VendingMachine**, que corresponde al almacenamiento de estas instancias de moneda en la máquina. Concretamente, **VendingMachine** alberga dos estructuras que referencian monedas: `depósito` y `listausuario` ("buffer" de las monedas introducidas por el usuario).

Los `productos` son almacenados en un mapa en el que se busca por *nombre del producto* y se obtiene su *valor* eficientemente.

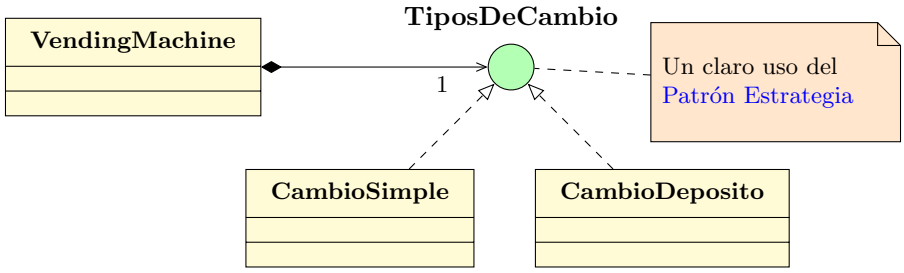


[1] Patrón inmutable

Las listas de **EuroCoin** que componen **VendingMachine** permanecen siempre ordenadas descendentemente para simplificar la implementación y complejidad de los métodos de cambio.

La clase principal también tiene una relación de composición con **Tipos de Cambio**, una interfaz que establece unas condiciones que deben cumplir sus clases *realizadoras*. Por tanto, las clases realizadoras **CambioSimple** y **CambioDeposito** poseen un marco común por el cual las clases que las usen las pueden tratar como una misma clase *indistintamente de su implementación*[2]

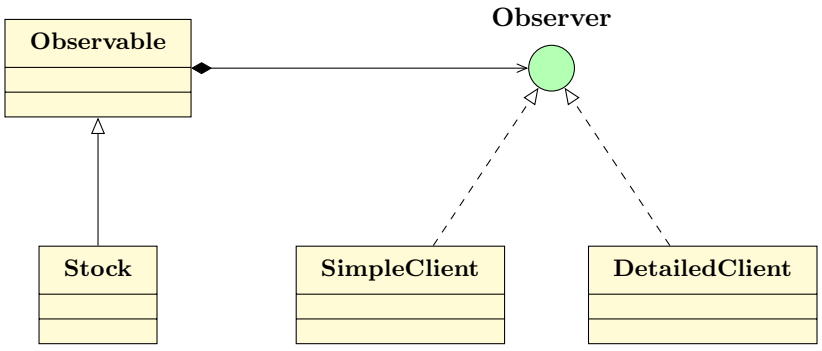
El diagrama de clases completo se muestra, junto con los de secuencia, al final del documento.



2.2. Ejercicio 2

En el diseño final se muestran 4 clases y una interfaz. El diseño en su totalidad es una clara realización del Patrón Observador. Se da como observable concreto a la clase **Stock**, que *hereda* de la clase *abstracta* **Observable** y por tanto posee las estructuras y métodos necesarios para relacionarse con una serie de observadores.

La interfaz **Observer** agrupa a todos los observadores dando una interfaz común para todos ellos y siendo esta la que genera dependencia en **Observable** y no sus clases implementadoras (véase de nuevo: Pr. Inv. Dependencia).



[2]Mientras cumplan con los contratos declarados por la interfaz. Prin. Inversión de la Dependencia

3. Principios de diseño utilizados

(a): Primer diseño (b): Segundo diseño

1. **Principio de inversión de la dependencia:**
   Las dependencias deben ser clausuradas en las interfaces y no en las clases implementadoras para que el código sea asequible para las susceptibles modificaciones futuras.

   a) En este diseño el principio es aplicado en la interfaz **TipoDeCambio** que es la que posee las relaciones que de otra manera se dirigirían a las clases **CambioSimple** y **CambioDeposito**.

   b) La interfaz **Observer** colapsa las dependencias de sus clases implementadoras en sí misma a la hora de invocar un objeto con el tipo Observer.

2. **Principio de Responsabilidad Única:**
   Toda clase e instancia respectiva deberían ofrecer únicamente servicios estrechamente relacionados entre sí, repartiendo las responsabilidades del todo en diferentes clases. El nivel de **cohesión** de estos diseños es demostrable a través de los diagramas de secuencia situados al final del documento.

3. **Principio Abierto-Cerrado:**
   Toda clase debe estar cerrada a modificaciones y abierta a extensiones.

   a) No se da el caso.

   b) Un ejemplo de la implementación de este principio se ve en la clase **Stock** que posee una relación de herencia con **Observable** por tanto **Stock** posee los servicios propios de su *superclase* además de de los servicios correspondientes al manejo de los datos de las acciones.

4. **Principio de mínimo conocimiento**[3]
   Las relaciones entre clases deben ser directas siempre que no se busque explícitamente lo contrario. La aplicación de este principio resulta en una disminución del acoplamiento y, por tanto, de la dependencia entre clases favoreciendo futuras modificaciones del software. Todas las relaciones en estos diseños se apegan a la filosofía de este principio.

[3] Ley de Démeter

4. Patrones de diseño utilizados

(a): Primer diseño (b): Segundo diseño

1. **Patrón Inmutable:**
   Un objeto no debería ser mutable salvo que sea específicamente necesario que lo sea, esto facilita el manejo de referencias y refuerza la integridad del estado de objetos más complejos que se compongan de estos.

   a) En este diseño el patrón se aplica en la clase **EuroCoin**.

   b) No se aplica (es semánticamente incongruente con el concepto de valor bursátil)

2. **Patrón Estrategia:**
   Hay muchos algoritmos que comparten una misma interfaz y comportamiento externo con otros, se podría así agrupar algoritmos en una "familia" en la cual *todos implementan a la misma interfaz* y todos pueden ser usados en sustitución de todos sin que las clases que los necesiten tengan que ser adaptadas. Esto **aumenta la extensibilidad a la hora de la implementar futuros algoritmos de esa familia.**

   a) En este diseño el patrón se aplica en la interfaz **TipoDeCambio** y sus realizadoras.

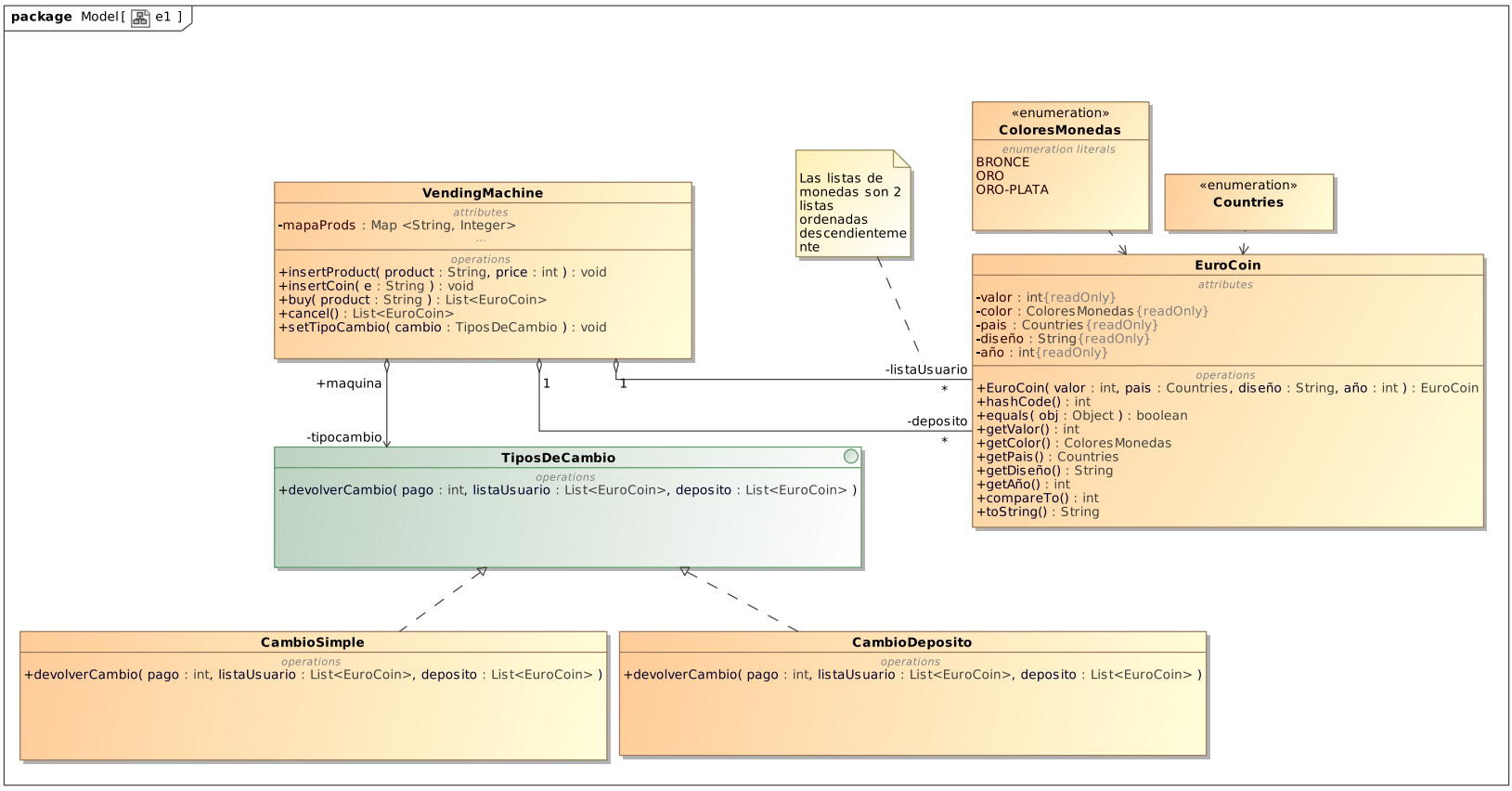
   b) No se aplica

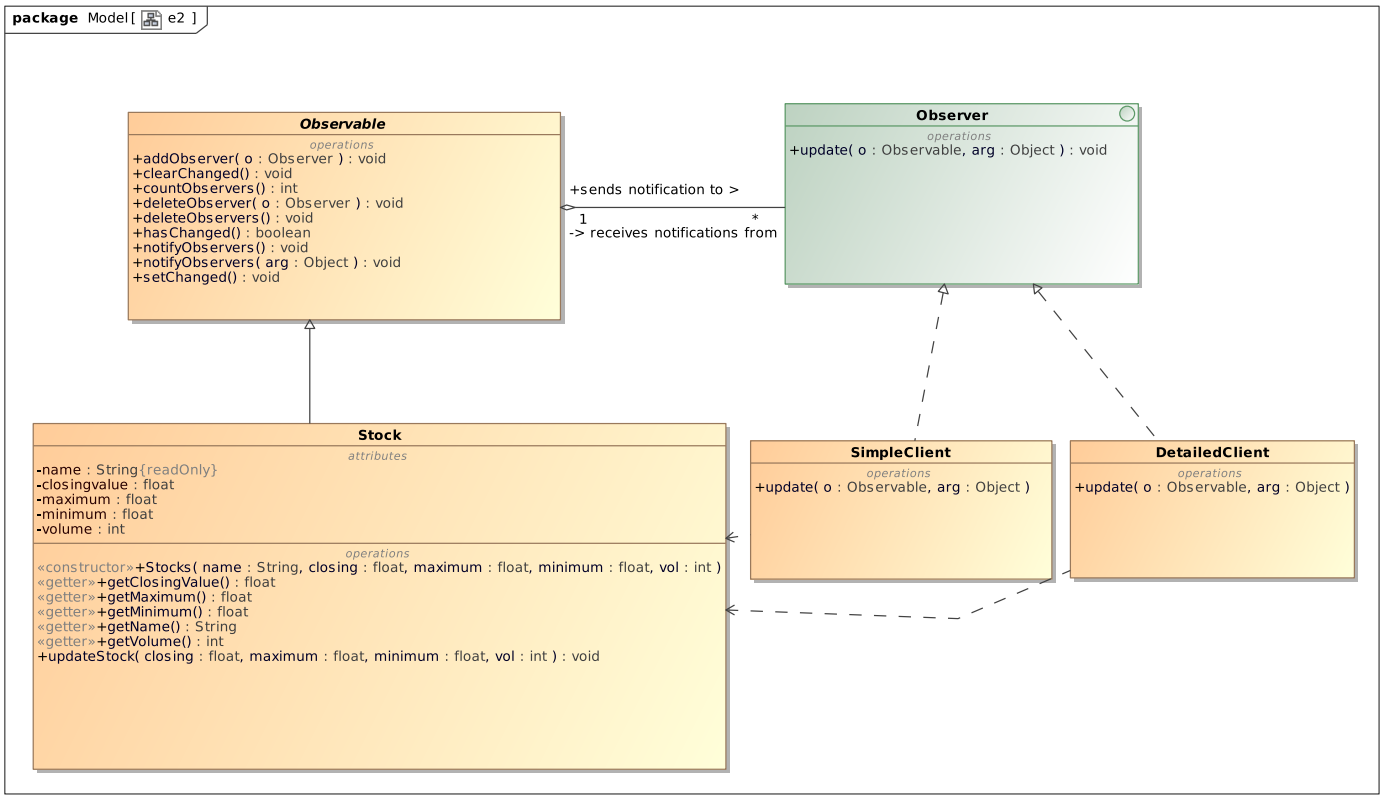
3. **Patrón Observador:**
   Una instancia que es observada[4] por otras no tiene por qué conocer la identidad de sus observadores. En su lugar la clase *observable* guarda una lista de observadores anónimos desde el punto de vista de la instancia, ya que estos se enmascaran tras una interfaz que todos implementan. De esta manera el *acoplamiento* del diseño se reduce.

   a) No se aplica

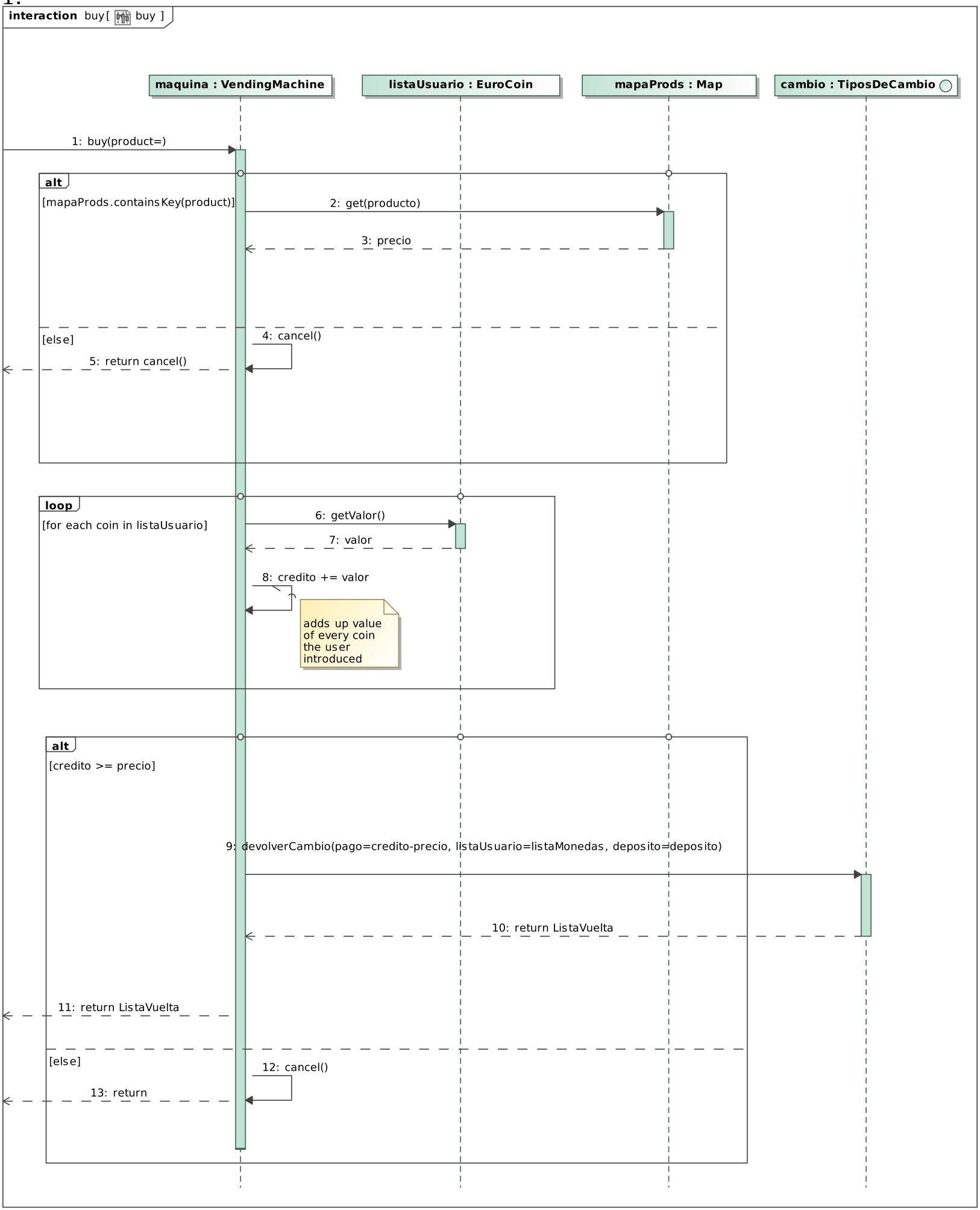
   b) El diseño en esencia es una realización de este patrón. En concreto su "versión" **PULL**, que da únicamente a una referencia de un objeto **Stock** a sus *observadores* en el momento de su notificación.

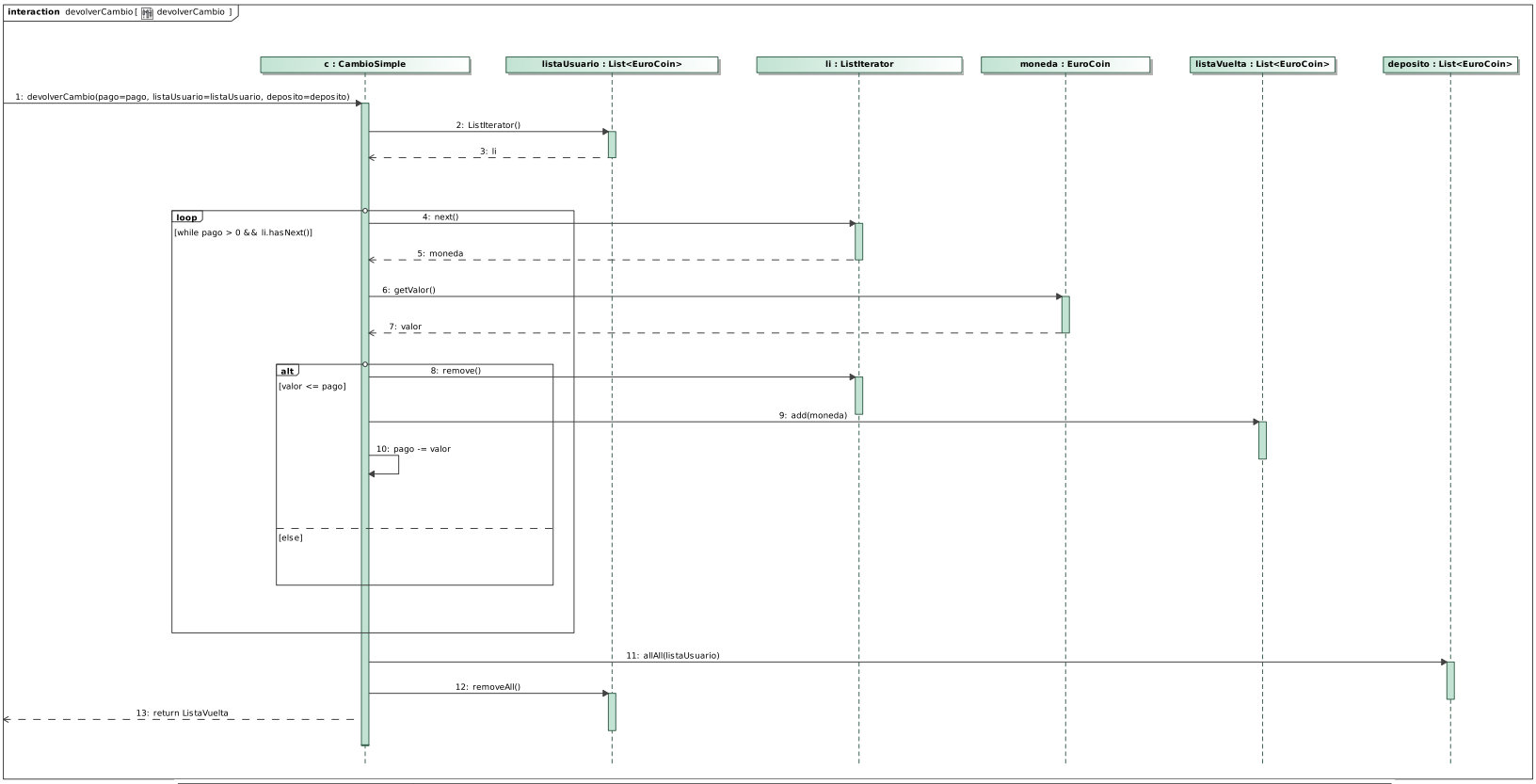
[4] Hay objetos que dependen del estado de esta instancia

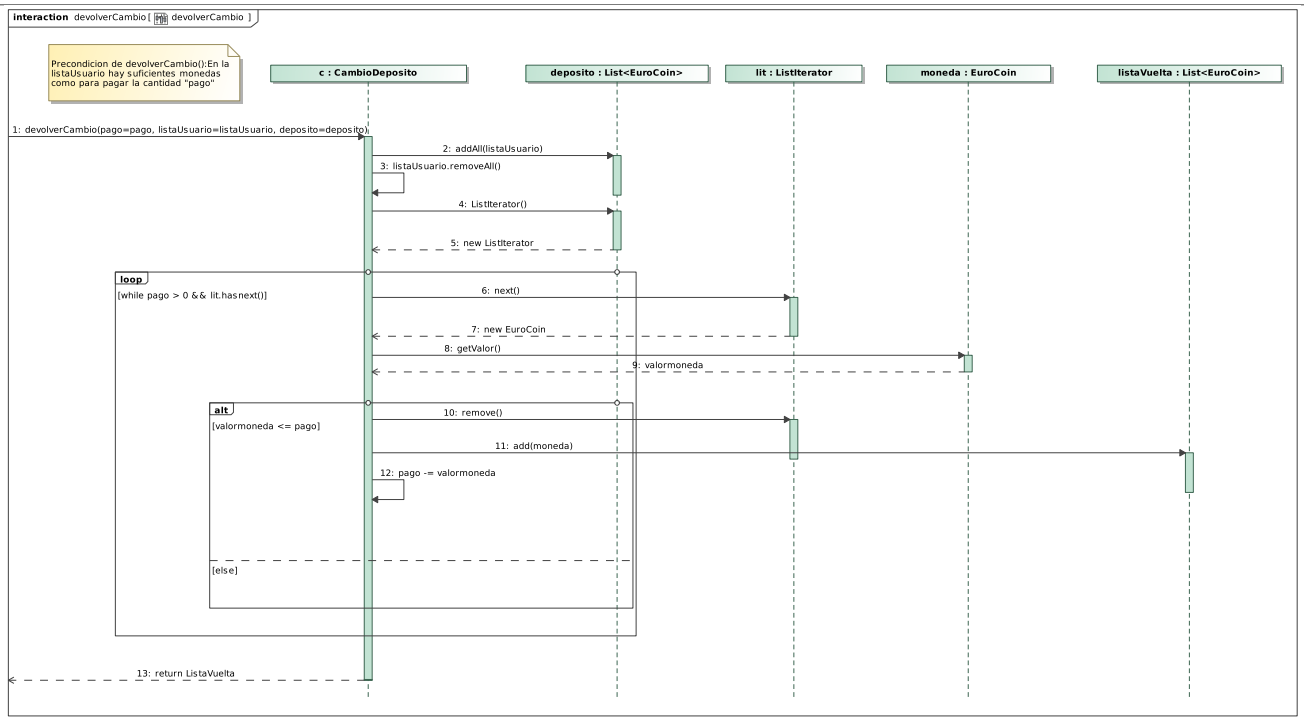

package Model [e1]
«enumeration»
ColoresMonedas
enumeration literals
BRONCE
ORO
ORO-PLATA
«enumeration»
Countries
Las listas de monedas son 2 listas ordenadas descendientemente
VendingMachine
attributes
-mapaProds : Map <String, Integer>
...
operations
+insertProduct(product : String, price : int) : void
+insertCoin(e : String) : void
+buy(product : String) : List<EuroCoin>
+cancel() : List<EuroCoin>
+setTipoCambio(cambio : TiposDeCambio) : void
EuroCoin
attributes
-valor : int{readOnly}
-color : ColoresMonedas{readOnly}
-pais : Countries{readOnly}
-diseño : String{readOnly}
-año : int{readOnly}
operations
+EuroCoin(valor : int, pais : Countries, diseño : String, año : int) : EuroCoin
+hashCode() : int
+equals(obj : Object) : boolean
+getValor() : int
+getColor() : ColoresMonedas
+getPais() : Countries
+getDiseño() : String
+getAño() : int
+compareTo() : int
+toString() : String
+maquina
1
1
-listaUsuario
*
-deposito
*
-tipocambio
TiposDeCambio
operations
+devolverCambio(pago : int, listaUsuario : List<EuroCoin>, deposito : List<EuroCoin>)
CambioSimple
operations
+devolverCambio(pago : int, listaUsuario : List<EuroCoin>, deposito : List<EuroCoin>)
CambioDeposito
operations
+devolverCambio(pago : int, listaUsuario : List<EuroCoin>, deposito : List<EuroCoin>)

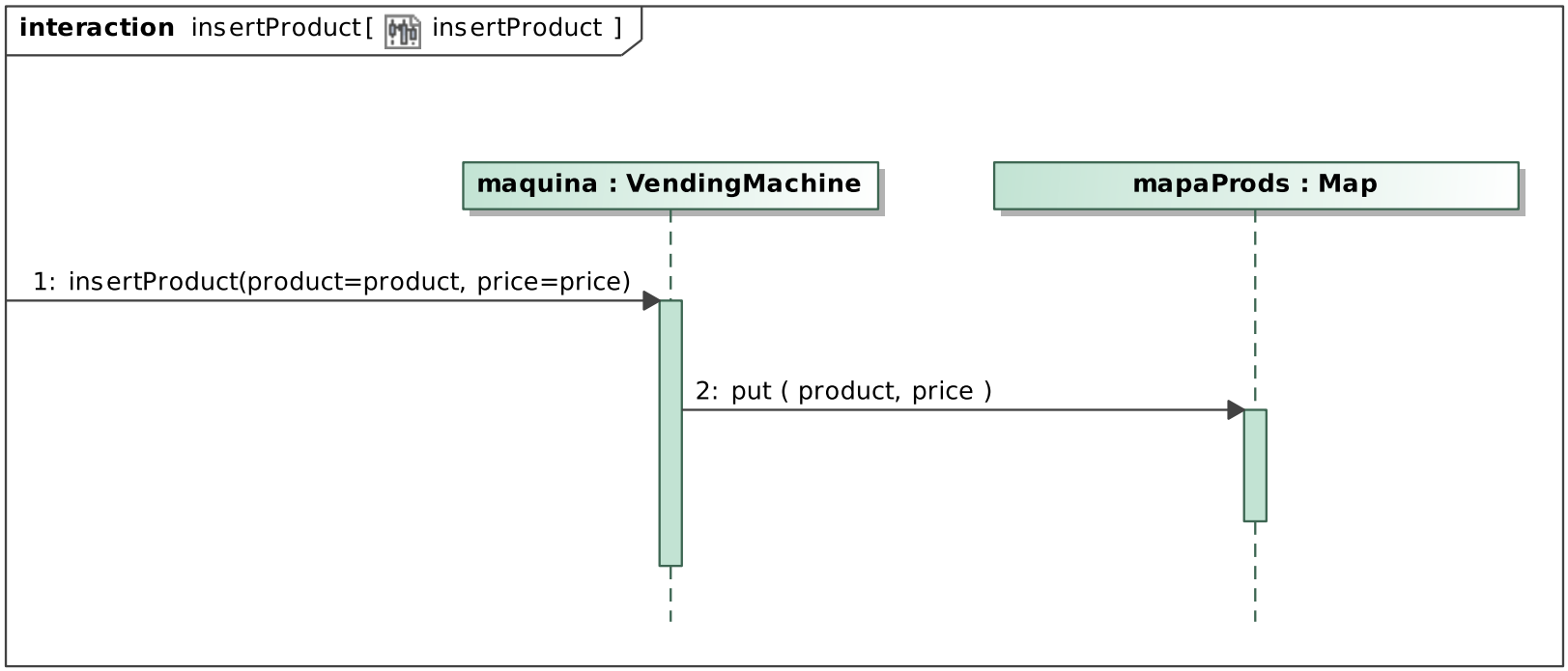

package Model [e2]
Observable
operations
+addObserver(o : Observer) : void
+clearChanged() : void
+countObservers() : int
+deleteObserver(o : Observer) : void
+deleteObservers() : void
+hasChanged() : boolean
+notifyObservers() : void
+notifyObservers(arg : Object) : void
+setChanged() : void
+sends notification to >
1
-> receives notifications from
*
Observer
operations
+update(o : Observable, arg : Object) : void
Stock
attributes
-name : String{readOnly}
-closingvalue : float
-maximum : float
-minimum : float
-volume : int
operations
«constructor»+Stocks(name : String, closing : float, maximum : float, minimum : float, vol : int)
«getter»+getClosingValue() : float
«getter»+getMaximum() : float
«getter»+getMinimum() : float
«getter»+getName() : String
«getter»+getVolume() : int
+updateStock(closing : float, maximum : float, minimum : float, vol : int) : void
SimpleClient
operations
+update(o : Observable, arg : Object)
DetailedClient
operations
+update(o : Observable, arg : Object)

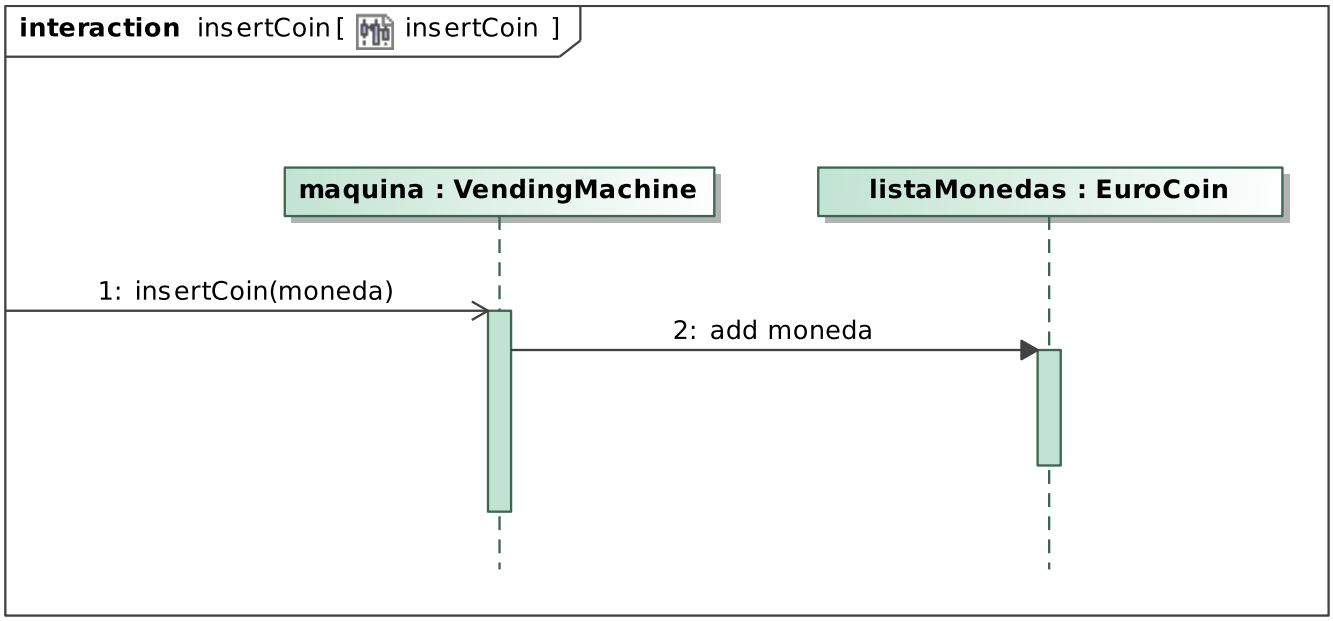
1:

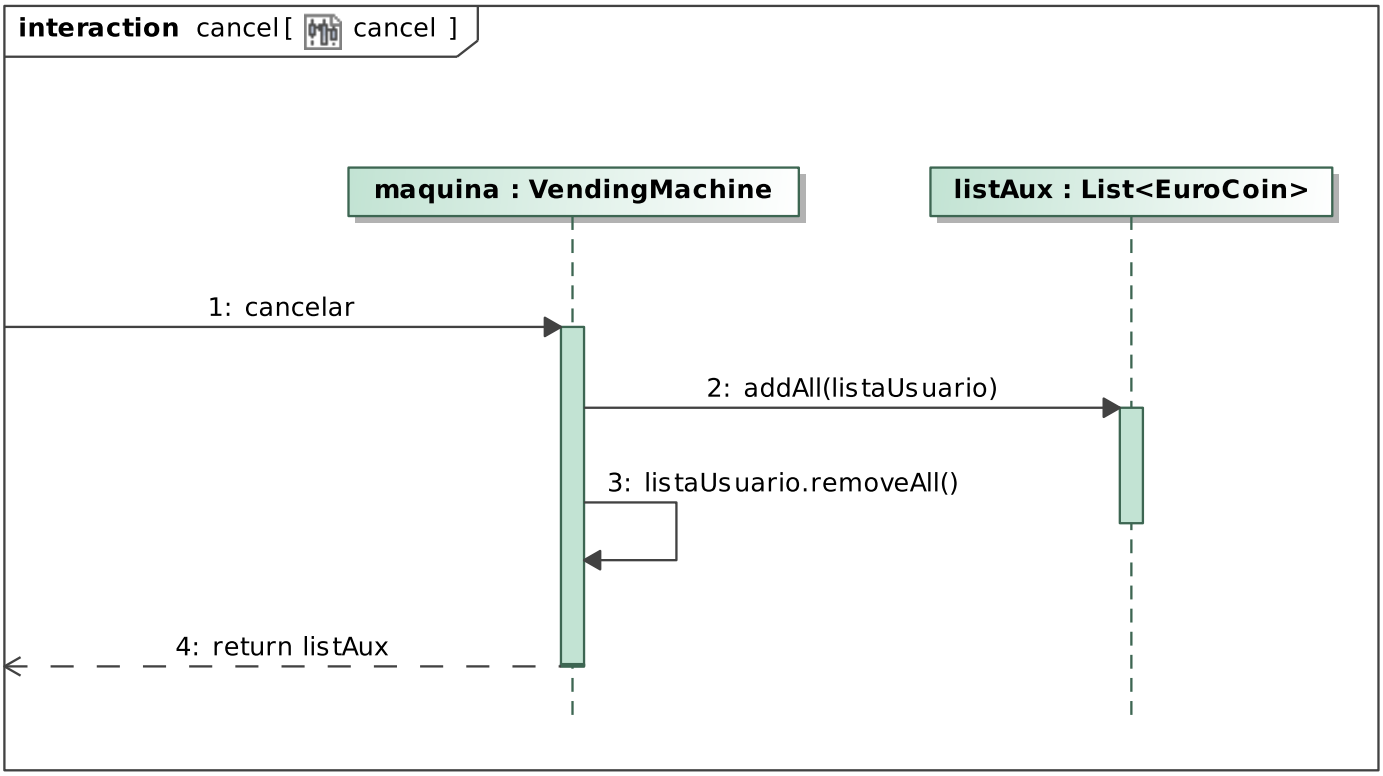

interaction buy[buy]
maquina : VendingMachine
listaUsuario : EuroCoin
mapaProds : Map
cambio : TiposDeCambio
1: buy(product=)
alt
[mapaProds.containsKey(product)]
2: get(producto)
3: precio
[else]
4: cancel()
5: return cancel()
loop
[for each coin in listaUsuario]
6: getValor()
7: valor
8: credito += valor
adds up value of every coin the user introduced
alt
[credito >= precio]
9: devolverCambio(pago=credito-precio, listaUsuario=listaMonedas, deposito=deposito)
10: return ListaVuelta
11: return ListaVuelta
[else]
12: cancel()
13: return


interaction devolverCambio [devolverCambio]
c : CambioSimple
listaUsuario : List<EuroCoin>
li : ListIterator
moneda : EuroCoin
listaVuelta : List<EuroCoin>
deposito : List<EuroCoin>
1: devolverCambio(pago=pago, listaUsuario=listaUsuario, deposito=deposito)
2: ListIterator()
3: li
loop
[while pago > 0 && li.hasNext()]
4: next()
5: moneda
6: getValor()
7: valor
alt
[valor <= pago]
8: remove()
9: add(moneda)
10: pago -= valor
[else]
11: allAll(listaUsuario)
12: removeAll()
13: return ListaVuelta

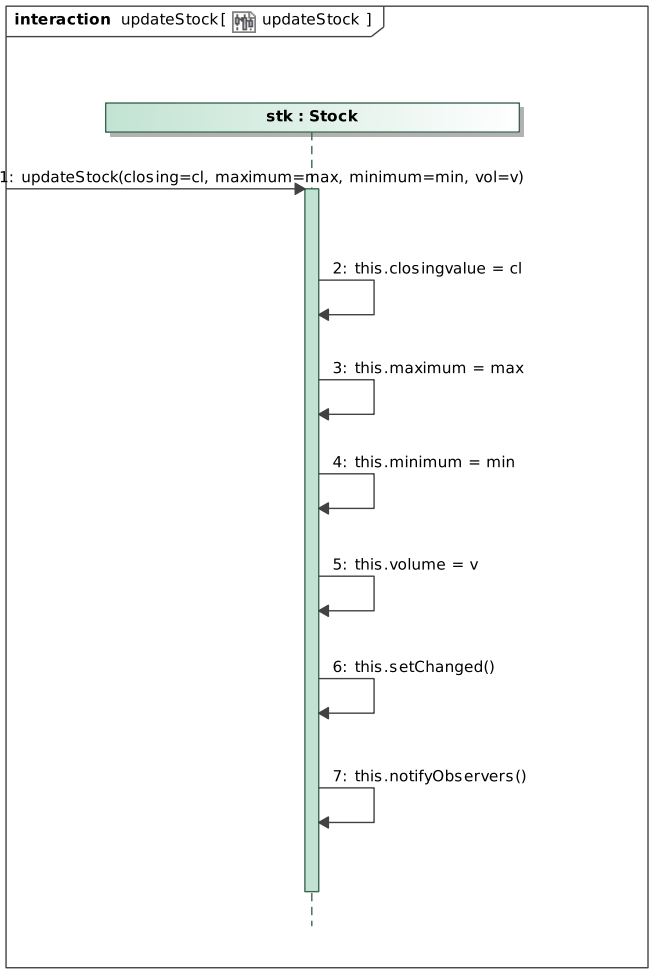

interaction devolverCambio [devolverCambio]
Precondicion de devolverCambio():En la listaUsuario hay suficientes monedas como para pagar la cantidad "pago"
c : CambioDeposito
deposito : List<EuroCoin>
lit : ListIterator
moneda : EuroCoin
listaVuelta : List<EuroCoin>
1: devolverCambio(pago=pago, listaUsuario=listaUsuario, deposito=deposito)
2: addAll(listaUsuario)
3: listaUsuario.removeAll()
4: ListIterator()
5: new ListIterator
loop
[while pago > 0 && lit.hasNext()]
6: next()
7: new EuroCoin
8: getValor()
9: valormoneda
alt
[valormoneda <= pago]
10: remove()
11: add(moneda)
12: pago -= valormoneda
[else]
13: return ListaVuelta

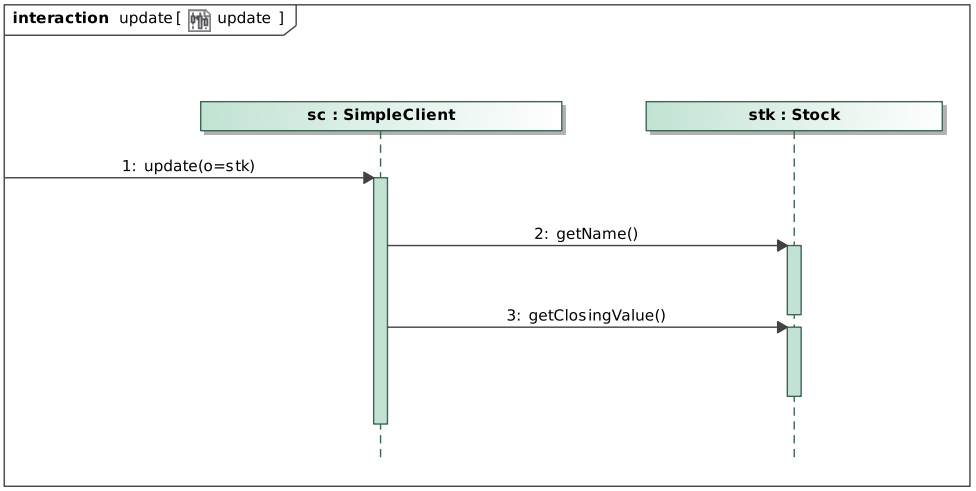

interaction insertProduct [insertProduct]
maquina : VendingMachine
mapaProds : Map
1: insertProduct(product=product, price=price)
2: put (product, price)

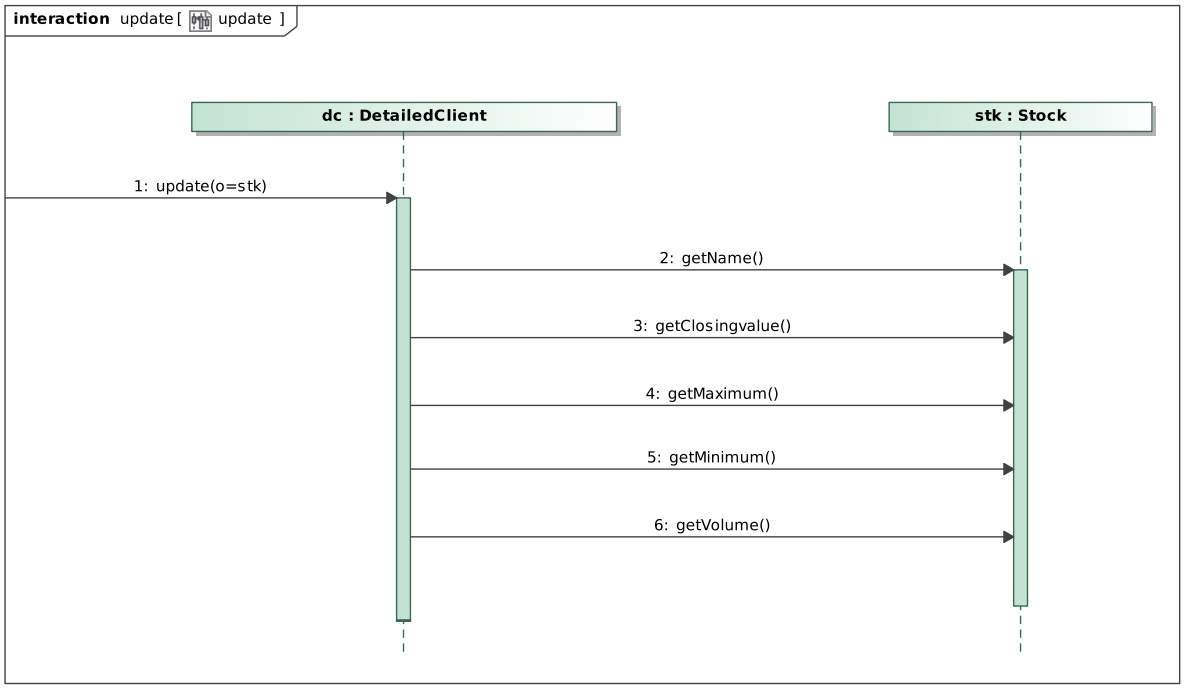

interaction insertCoin [insertCoin]
maquina : VendingMachine
listaMonedas : EuroCoin
1: insertCoin(moneda)
2: add moneda


interaction cancel [cancel]
maquina : VendingMachine
listAux : List<EuroCoin>
1: cancelar
2: addAll(listaUsuario)
3: listaUsuario.removeAll()
4: return listAux

2:


interaction updateStock [updateStock]
stk : Stock
1: updateStock(closing=cl, maximum=max, minimum=min, vol=v)
2: this.closingvalue = cl
3: this.maximum = max
4: this.minimum = min
5: this.volume = v
6: this.setChanged()
7: this.notifyObservers()


interaction update [update]
sc : SimpleClient
stk : Stock
1: update(o=stk)
2: getName()
3: getClosingValue()


interaction update [update]
dc : DetailedClient
stk : Stock
1: update(o=stk)
2: getName()
3: getClosingvalue()
4: getMaximum()
5: getMinimum()
6: getVolume()