Sistema de compra online

Principios de diseño empleados

- **Encapsula lo que varía**: Cada estado está encapsulado en una clase distinta. (*Cancelled*, *CheckOut*, *Completed*, *Payment*, *ShoppingCart*)
- **Principio de responsabilidad única**: puesto que cada estado es responsable únicamente de las acciones (métodos) que se pueden realizar en el mismo (Por ejemplo, no podemos hacer un *cancellOrder* desde el estado *Completed*, pero sí podemos hacer un *addProduct* desde el estado *CheckOut*)

Explicación del patrón/es usados

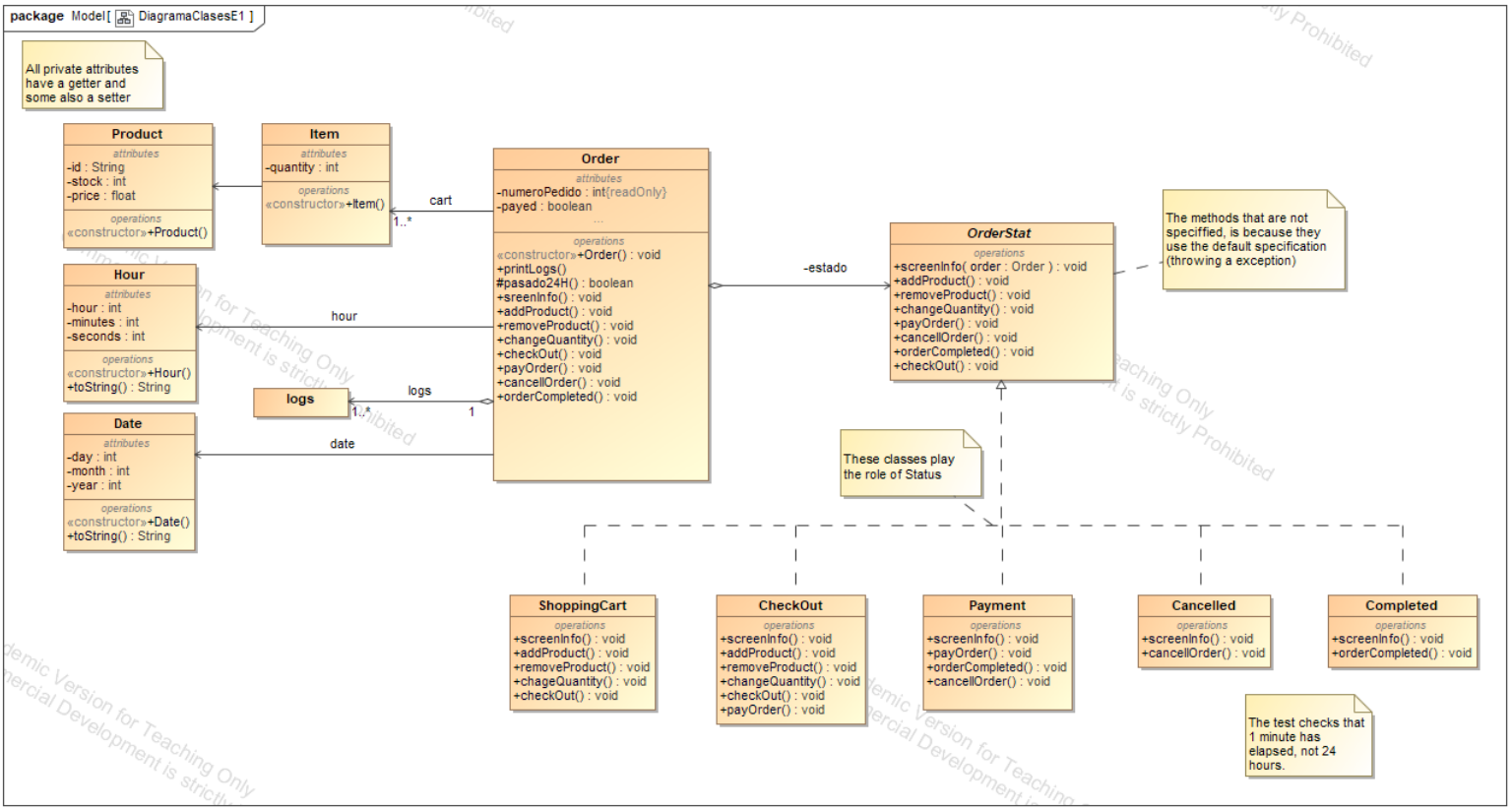
Para este ejercicio hemos empleado un **Patrón Estado**. Es un patrón de comportamiento que permite a un objeto, en nuestro caso “Order”, **modificar su conducta** al **cambiar** su **estado interno**.

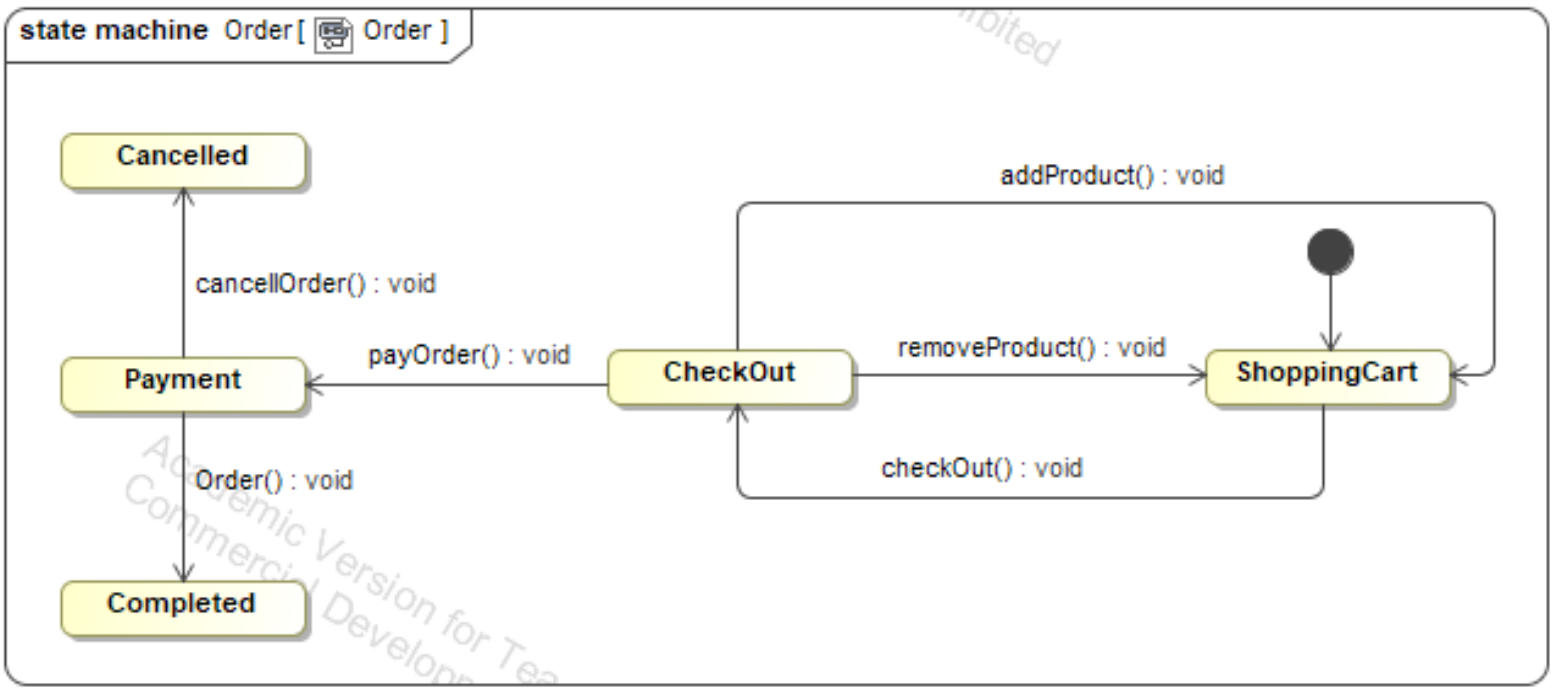
Esto se refleja en las clases *ShoppingCart*, *CheckOut*, *Payment*, *Cancelled* y *Completed*. Dichas clases heredan de la clase *OrderStat*, estado del pedido, e implementan las operaciones de la misma.

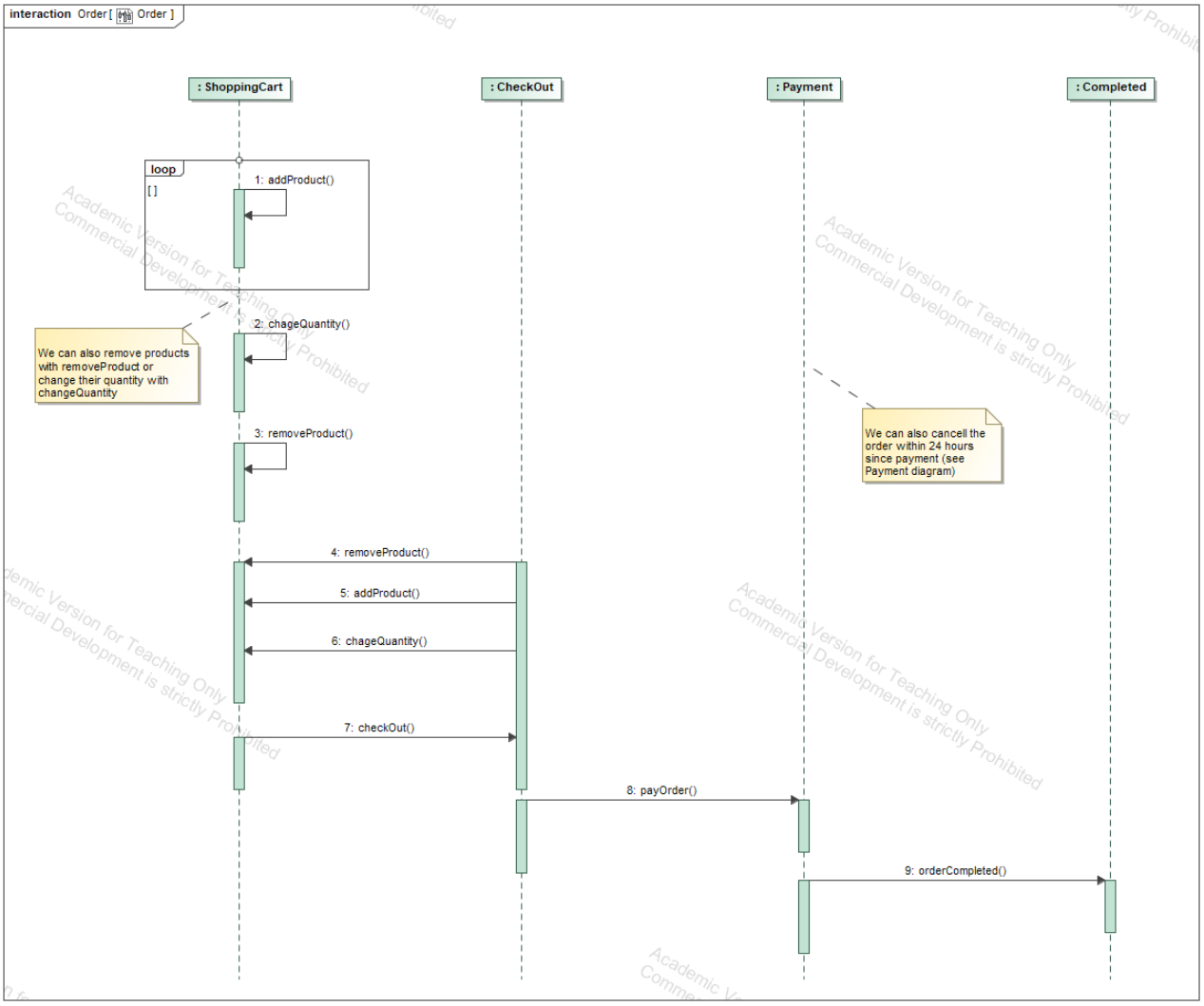
La clase *OrderStat* es una **interfaz** y no una clase abstracta, porque dispone de **implementaciones predeterminadas** de sus operaciones.

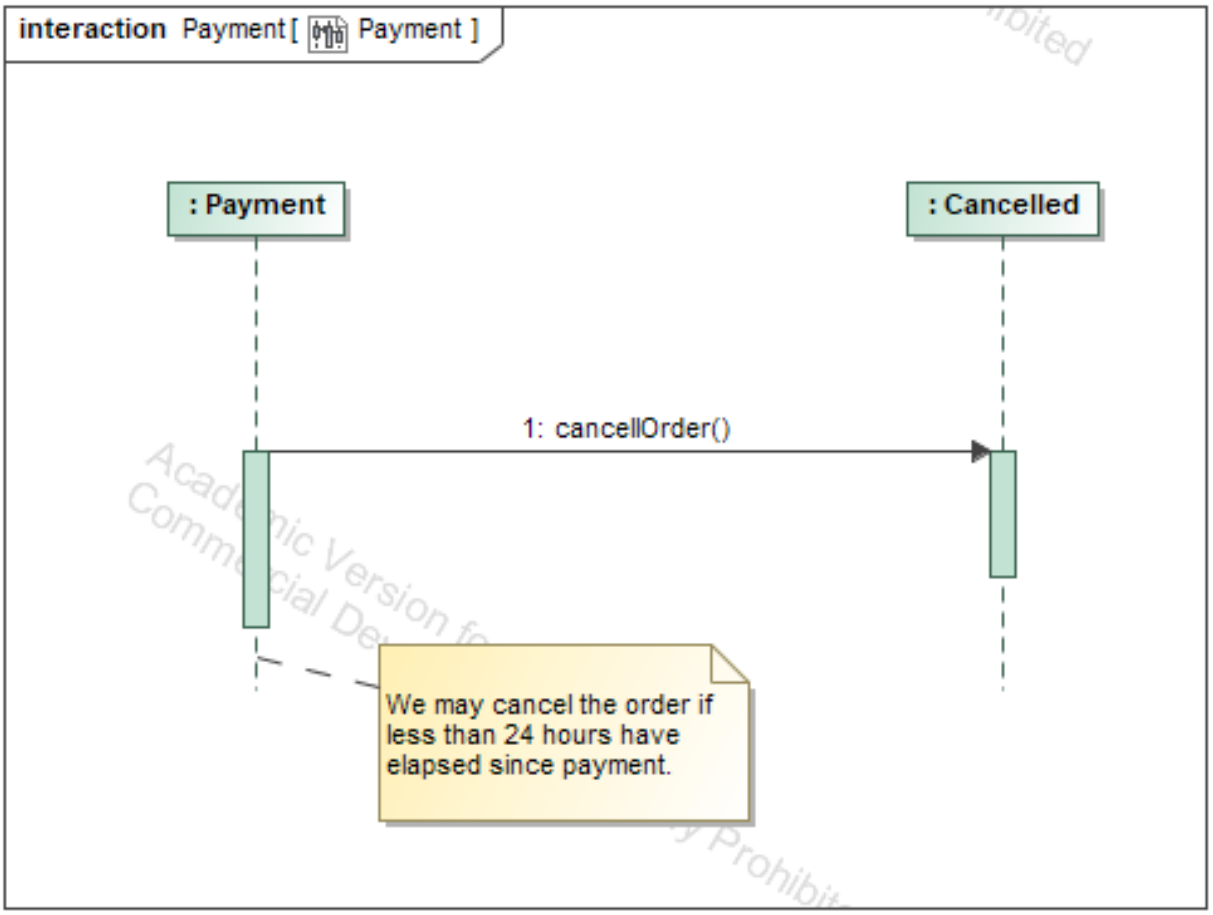
Este patrón nos permite encapsular el comportamiento específico de cada estado. Además, nos facilita añadir nuevos estados de forma sencilla, simplemente añadiendo nuevas clases. Las transiciones entre estados y el comportamiento de los mismos es m´as claro.
Por último, los objetos que representan a los estados pueden compartirse, siempre y cuando no incluyan información en variables de instancia.

El principal inconveniente es que el código es menos compacto que la solución basada en sentencias condicionales


package Model [DiagramaClasesE1]
All private attributes have a getter and some also a setter
Product
attributes
-id : String
-stock : int
-price : float
operations
«constructor»+Product()
Item
attributes
-quantity : int
operations
«constructor»+Item()
cart
1..*
Order
attributes
-numeroPedido : int{readOnly}
-payed : boolean
operations
«constructor»+Order() : void
+printLogs()
#pasado24H() : boolean
+sreenInfo() : void
+addProduct() : void
+removeProduct() : void
+changeQuantity() : void
+checkOut() : void
+payOrder() : void
+cancellOrder() : void
+orderCompleted() : void
-estado
OrderStat
operations
+screenInfo(order : Order) : void
+addProduct() : void
+removeProduct() : void
+changeQuantity() : void
+payOrder() : void
+cancellOrder() : void
+orderCompleted() : void
+checkOut() : void
The methods that are not specifified, is because they use the default specification (throwing a exception)
Hour
attributes
-hour : int
-minutes : int
-seconds : int
operations
«constructor»+Hour()
+toString() : String
hour
logs
1..*
1
Date
attributes
-day : int
-month : int
-year : int
operations
«constructor»+Date()
+toString() : String
date
These classes play the role of Status
ShoppingCart
operations
+screenInfo() : void
+addProduct() : void
+removeProduct() : void
+chageQuantity() : void
+checkOut() : void
CheckOut
operations
+screenInfo() : void
+addProduct() : void
+removeProduct() : void
+changeQuantity() : void
+checkOut() : void
+payOrder() : void
Payment
operations
+screenInfo() : void
+payOrder() : void
+orderCompleted() : void
+cancellOrder() : void
Cancelled
operations
+screenInfo() : void
+cancellOrder() : void
Completed
operations
+screenInfo() : void
+orderCompleted() : void
The test checks that 1 minute has elapsed, not 24 hours.


state machine Order [Order]
Cancelled
addProduct() : void
cancellOrder() : void
payOrder() : void
removeProduct() : void
Payment
CheckOut
ShoppingCart
Order() : void
checkOut() : void
Completed


interaction Order [Order]
: ShoppingCart
: CheckOut
: Payment
: Completed
loop
[ ]
1: addProduct()
2: chageQuantity()
We can also remove products with removeProduct or change their quantity with changeQuantity
3: removeProduct()
We can also cancell the order within 24 hours since payment (see Payment diagram)
4: removeProduct()
5: addProduct()
6: chageQuantity()
7: checkOut()
8: payOrder()
9: orderCompleted()


interaction Payment [Payment]
: Payment
: Cancelled
1: cancellOrder()
We may cancel the order if less than 24 hours have elapsed since payment.

Gestión de alertas de tanques de acuario

Clases Observadas: todas las instancias de *Tank*, *Warning* y *Sensor*.
Clases Observadoras: todas las instancias de *Sensor*, *Warning* y *Personal*.

Principios de diseño empleados

- **Responsabilidad Única**: Puesto que el objeto observado únicamente es responsable de notificar su cambio de estado a sus observadores, y son éstos últimos los que deben tomar decisiones en consecuencia.
- **Inversión de la dependencia**: Las clases *Subject* y *Device* son clases abstractas. Todos los objetos que interpreten el papel de Sujeto deben heredar de esta. Y todas los objetos que interpreten el papel Device (a su vez un Observador) deben heredar de dicha clase.
- **Abierto-Cerrado**:

Explicación del patrón/es usados

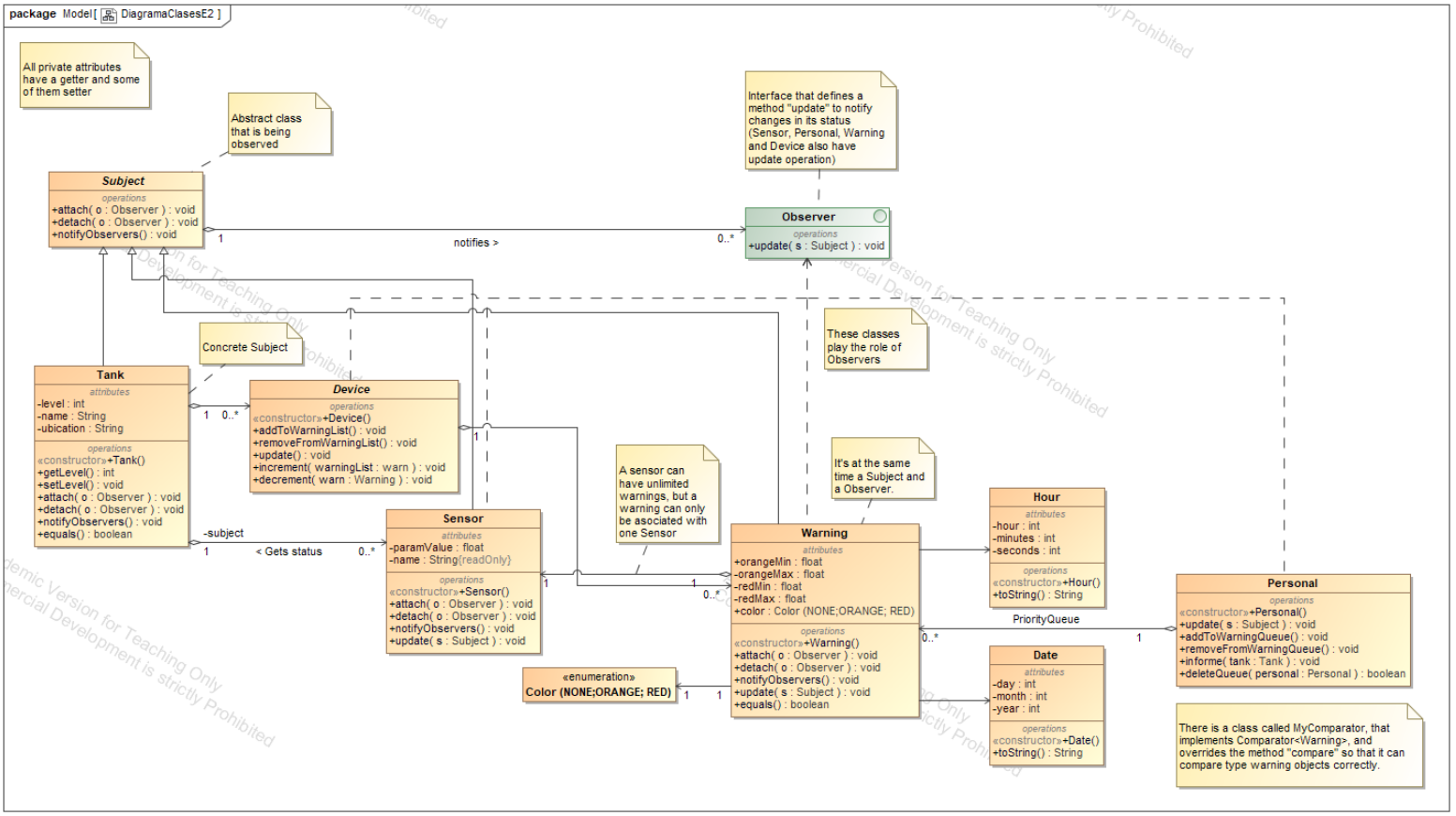
En este caso nos decantamos por un **Patrón Observador**, puesto que en este ejercicio existen muchas dependencias “uno a muchos” entre objetos. Esto se puede comprobar claramente con las clases *Sensor* y *Warning*, puesto que un sensor puede tener muchas alertas asociadas, pero una alerta tiene un único sensor asociado.

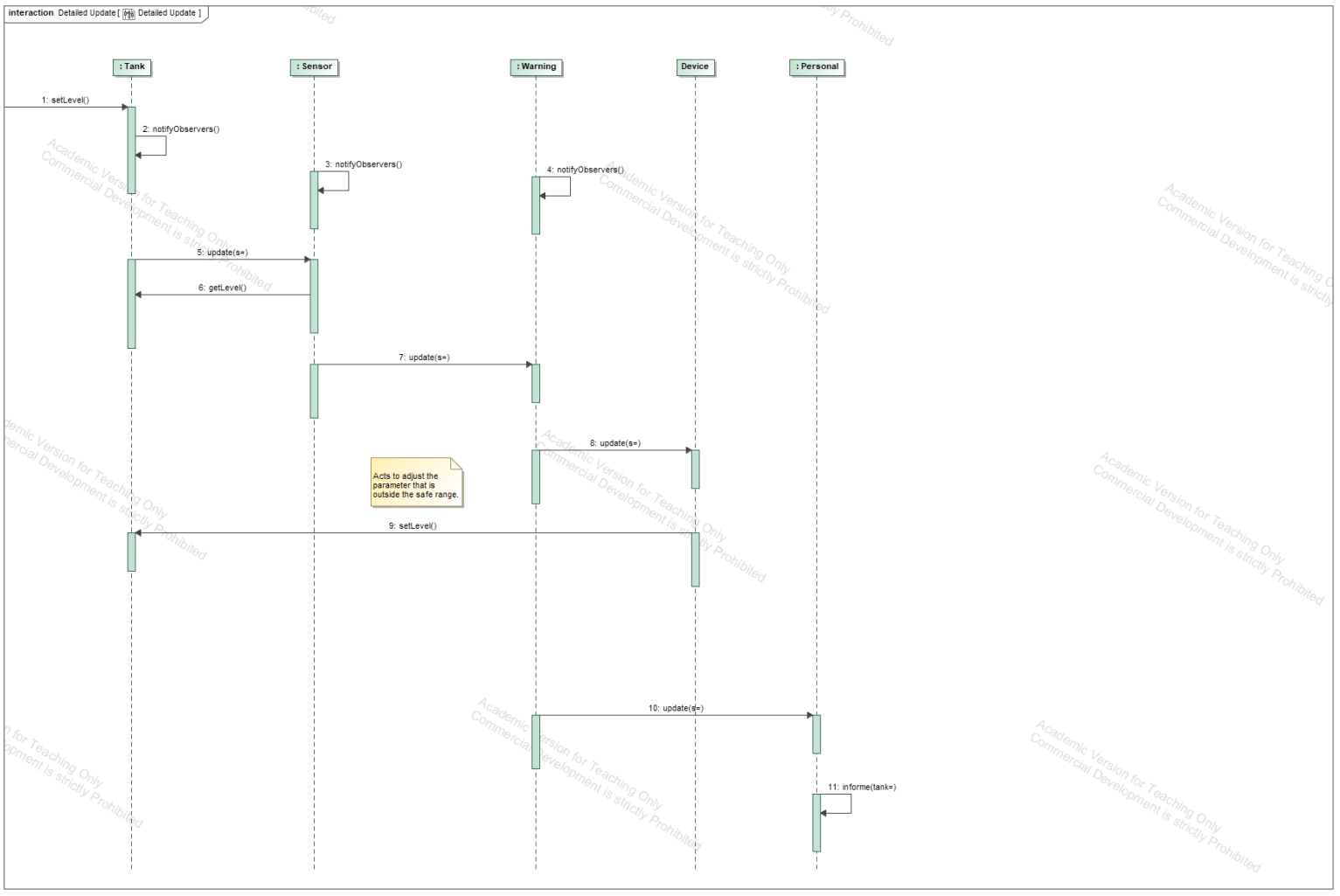
Además, una alerta observa a un sensor concreto, pero también es observada por los dispositivos y el personal. Así, cuando un sensor detecta un cambio en el valor que mide, notifica a las alertas y cada alerta, que es responsable de decidir qué acción tomar en consecuencia, debe notificar a sus observadores, que en el caso de las alarmas es el personal, que a su vez, actuará en consecuencia.

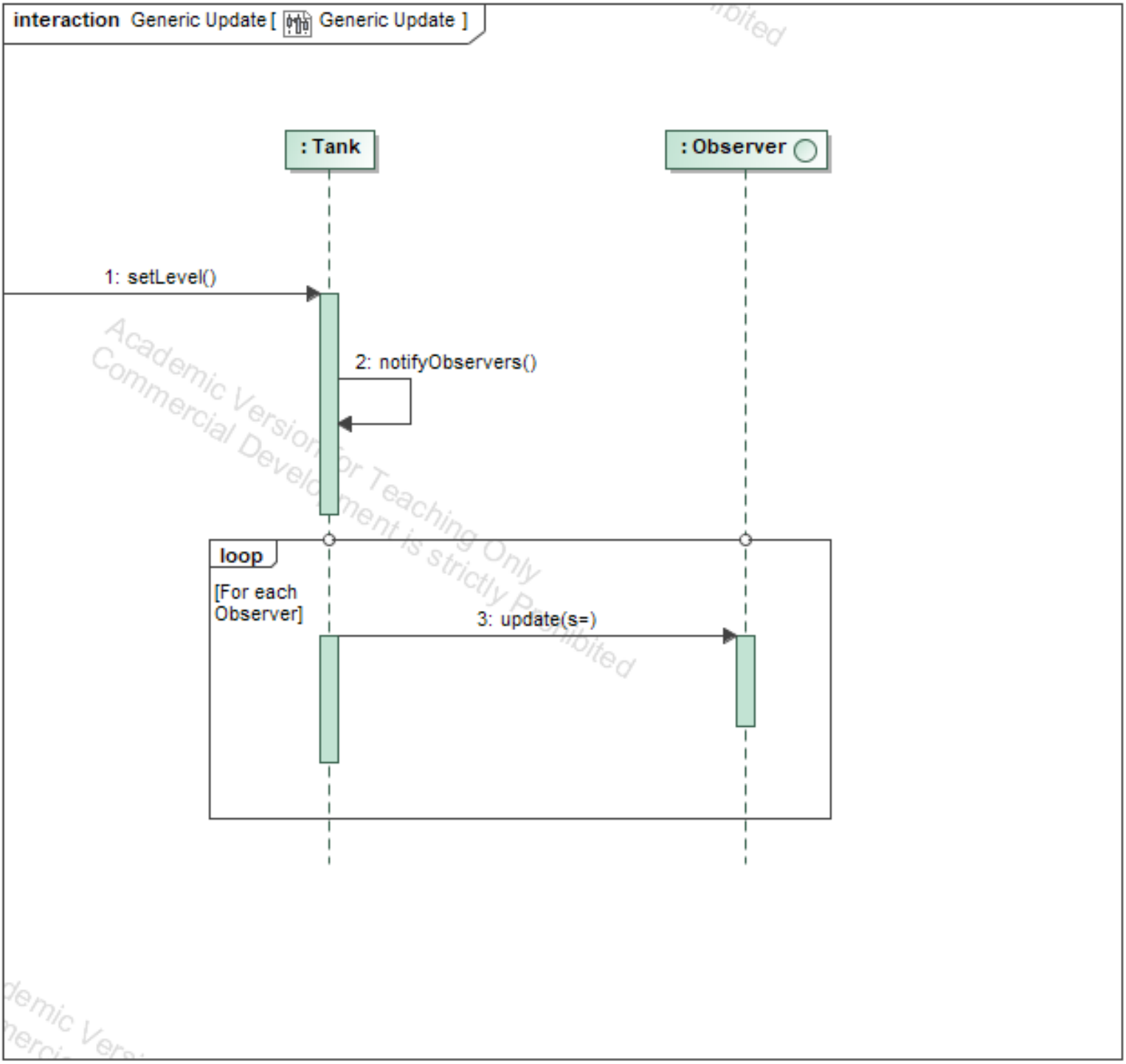
Es evidente que cuando determinados objetos cambian de estado, es necesario que todos los dependientes sean notificados y que actúen en consecuencia.

Las ventajas más destacables de este patrón son la capacidad de añadir nuevos observadores sin necesidad de modificar el sujeto observado, que la clase observable, no tiene por qué conocer la clase concreta a la que pertenecen sus observadores, únicamente se limita a notificar los cambios. Y que la responsabilidad del objeto observado se acaba con la notificación, son los observadores los que deciden qué acción llevar a cabo después de la notificación.

En nuestro caso, empleamos el modelo pull de este patrón, puesto que las clases sujeto, envían una notificación mínima a sus observadores, únicamente para notificar que su estado interno ha cambiado. Es responsabilidad de los observadores descubrir qué ha cambiado y actuar en consecuencia.
La principal ventaja de esto es que el observado en general las necesidades de los observadores. En contrapartida, puede ser ineficiente, ya que las clases observadoras tienen la que de descubrir qué ha cambiado sin la ayuda del sujeto.


package Model [DiagramaClasesE2]
All private attributes have a getter and some of them setter
Abstract class that is being observed
Interface that defines a method "update" to notify changes in its status (Sensor, Personal, Warning and Device also have update operation)
Subject
operations
+attach(o : Observer) : void
+detach(o : Observer) : void
+notifyObservers() : void
1
notifies >
0..*
Observer
operations
+update(s : Subject) : void
Concrete Subject
These classes play the role of Observers
Tank
attributes
-level : int
-name : String
-ubication : String
operations
«constructor»+Tank()
+getLevel() : int
+setLevel() : void
+attach(o : Observer) : void
+detach(o : Observer) : void
+notifyObservers() : void
+equals() : boolean
1
0..*
Device
operations
«constructor»+Device()
+addToWarningList() : void
+removeFromWarningList() : void
+update() : void
+increment(warningList : warn) : void
+decrement(warn : Warning) : void
A sensor can have unlimited warnings, but a warning can only be associated with one Sensor
It's at the same time a Subject and a Observer.
-subject
1
< Gets status
0..*
Sensor
attributes
-paramValue : float
-name : String{readOnly}
operations
«constructor»+Sensor()
+attach(o : Observer) : void
+detach(o : Observer) : void
+notifyObservers() : void
+update(s : Subject) : void
Warning
attributes
+orangeMin : float
-orangeMax : float
-redMin : float
-redMax : float
+color : Color (NONE;ORANGE; RED)
operations
«constructor»+Warning()
+attach(o : Observer) : void
+detach(o : Observer) : void
+notifyObservers() : void
+update(s : Subject) : void
+equals() : boolean
Hour
attributes
-hour : int
-minutes : int
-seconds : int
operations
«constructor»+Hour()
+toString() : String
PriorityQueue
0..*
1
Personal
operations
«constructor»+Personal()
+update(s : Subject) : void
+addToWarningQueue() : void
+removeFromWarningQueue() : void
+informe(tank : Tank) : void
+deleteQueue(personal : Personal) : boolean
«enumeration»
Color (NONE;ORANGE; RED)
Date
attributes
-day : int
-month : int
-year : int
operations
«constructor»+Date()
+toString() : String
There is a class called MyComparator, that implements Comparator<Warning>, and overrides the method "compare" so that it can compare type warning objects correctly.


interaction Detailed Update [Detailed Update]
: Tank
: Sensor
: Warning
Device
: Personal
1: setLevel()
2: notifyObservers()
3: notifyObservers()
4: notifyObservers()
5: update(s=)
6: getLevel()
7: update(s=)
8: update(s=)
Acts to adjust the parameter that is outside the safe range.
9: setLevel()
10: update(s=)
11: informe(tank=)


interaction Generic Update [Generic Update]
: Tank
: Observer
1: setLevel()
2: notifyObservers()
loop
[For each Observer]
3: update(s=)