Práctica de Diseño - Diseño Software

EJERCICIO 1

En el primer ejercicio del boletín, se nos pedía hacer un sistema de ventas online con una serie de fases (ShoppingCart, Checkout, Payment, Cancelled y Completed), con posibilidad de añadir fases posteriores en un futuro.

Explicación principios de diseño empleados en todas las fases:

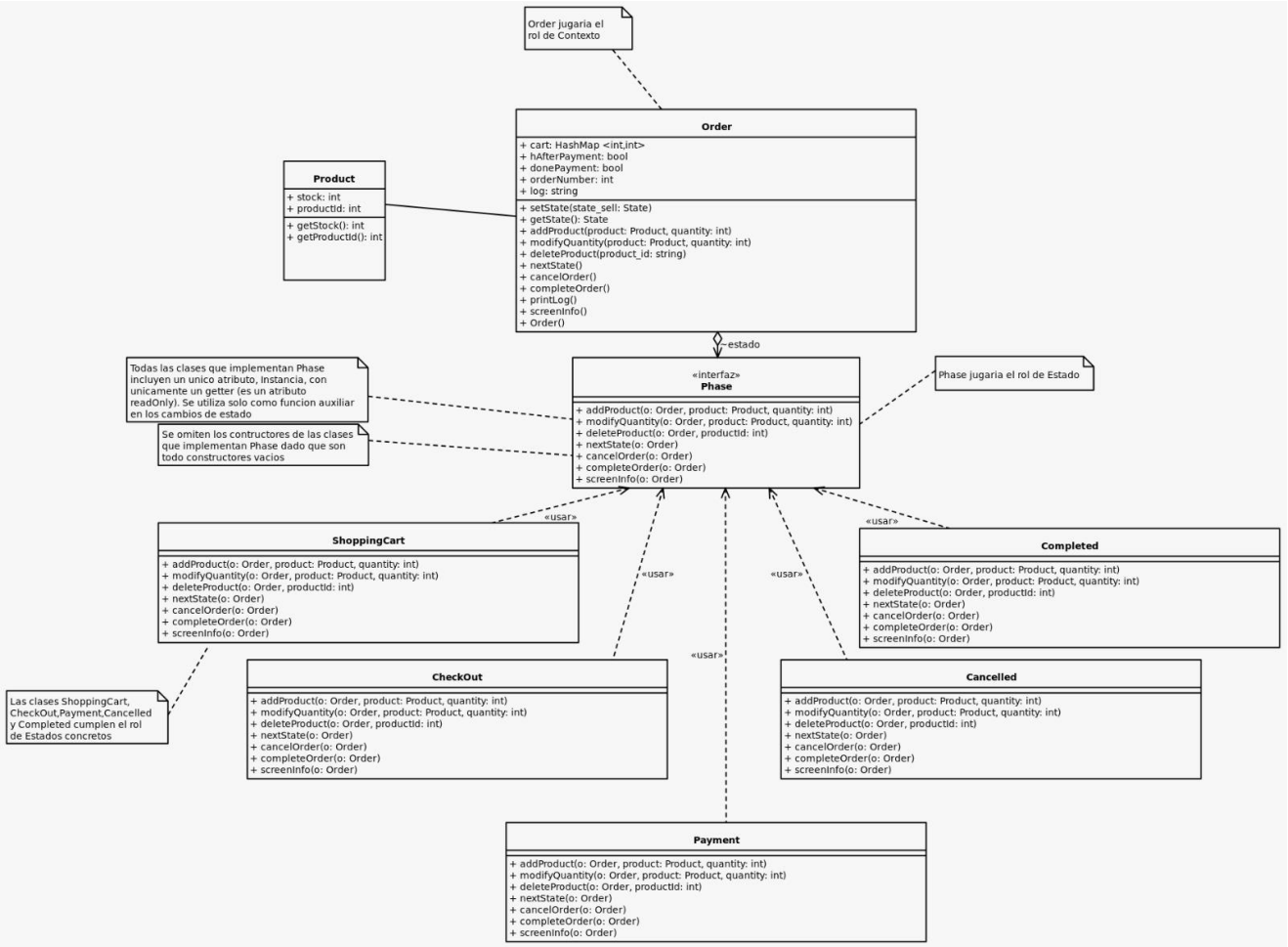
- Hemos empleado el principio SOLID abierto-cerrado, que dice que un módulo de software debería estar abierto a extensión y cerrado a su modificación. En nuestro código lo hemos empleado de la siguiente forma: Tenemos un interfaz llamado Phase con todos los métodos que se emplean en las fases de la venta online. Cada fase implementa los métodos de Phase de la forma determinada en la que los usen. Por ejemplo: el método addProducts en la fase ShoppingCart sirve para añadir nuevos productos al carrito, mientras que en la fase Completed no tiene ningún uso y devuelve una excepción porque el pedido ya ha sido realizado y no se puede modificar. Por lo tanto, usamos dicho principio en: Phase, ShoppingCart, Checkout, Payment, Cancelled y Completed.
- Usamos también el Principio de Responsabilidad Única, porque cada estado concreto tiene una responsabilidad y todas las otras clases están relacionadas y dependen de ella, además de tener sus propias responsabilidades con ellas mismas. Un ejemplo de esta responsabilidad entre fases sería por ejemplo en Payment, donde existe un valor boolean `hAfterPayment` que permite hacer la cancelación del producto. Por lo tanto, el acceso a la clase Cancelled depende del boolean para comprobar si han pasado 24h desde el pago del pedido.
- Principio Inversión de la Dependencia, porque todos los métodos de los estados dependen de los métodos de la interfaz y no de concreciones, un ejemplo sería cuando creamos el Hashmap para almacenar la información de los productos del pedido en el carrito.

Explicación del patrón de diseño usado:

- Para este ejercicio hemos empleado el patrón de estado debido a que necesitamos un diseño adaptable a los cambios y con el cual los objetos puedan modificar su conducta al cambiar su estado interno.
- Cada producto que comprar pasa por un determinado número de fases en las que su comportamiento es diferente: en ShoppingCart elegimos los productos que queremos y sus cantidades; en Checkout comprobamos que lo que hemos escogido se corresponde con lo que queremos, y si no es el caso, nos da la opción de volver al anterior paso y modificar lo que haga falta; en Payment tenemos la opción de cancelar la compra (dentro de las 24 horas de cancelación posteriores al pago) o no. De esta forma los objetos pasan por 4 estados hasta llegar a nuestras manos.

- En cada fase del pedido hemos usado el Patrón Instancia Única o Singleton, para asegurarnos de que cada clase tiene solamente una instancia, a la vez que proporciona un punto de acceso global a dicha instancia. En las cinco clases que representan las fases de la venta tenemos una declaración de instancia como esta: private static final N*ombrelase instanciaNombreclase* = *new Nombreclase*();
- Además de un método público estático getInstance y un constructor vacío para que nadie pueda crear un nuevo constructor y modificar la instancia.

- Diagrama de clases mediante el Patrón de Estados:

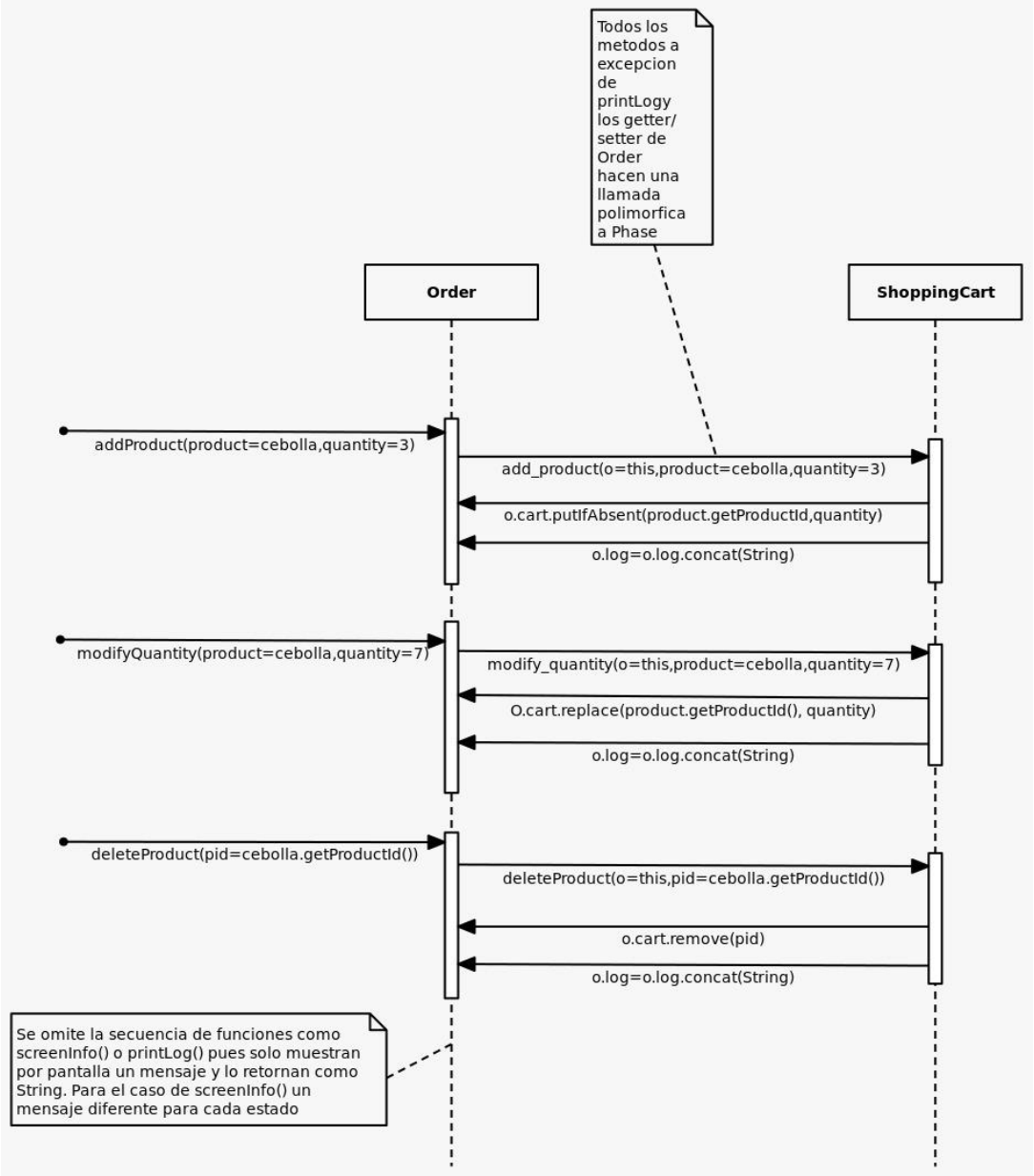


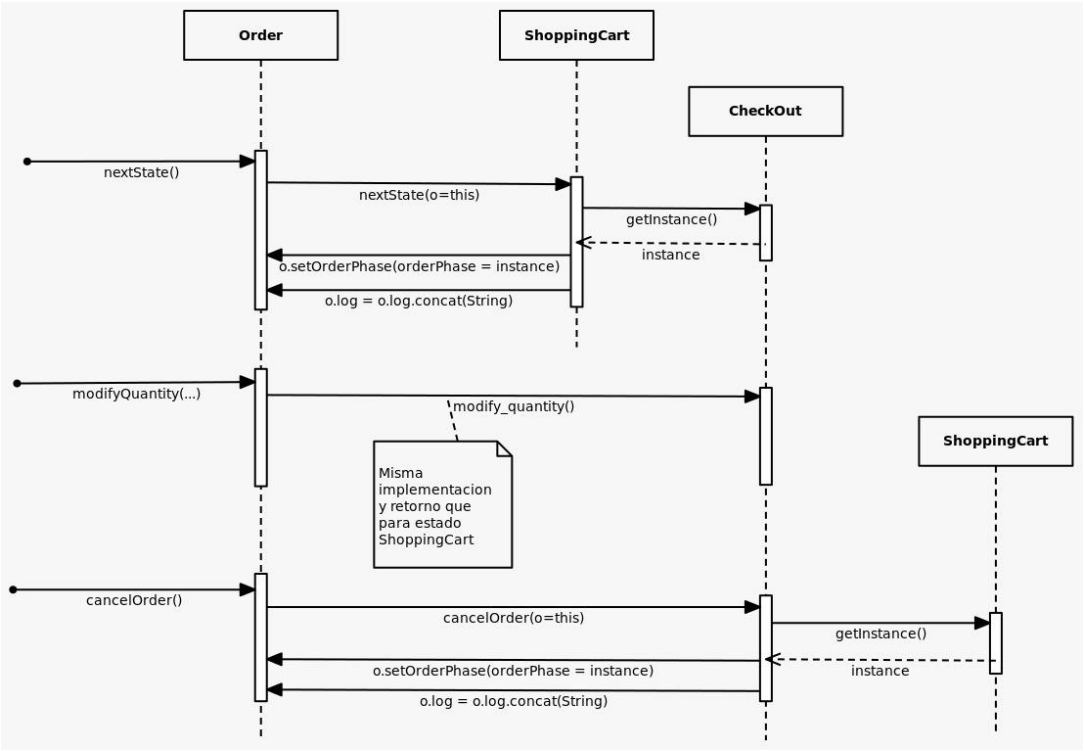
En este patrón de estados los roles serían los siguientes:

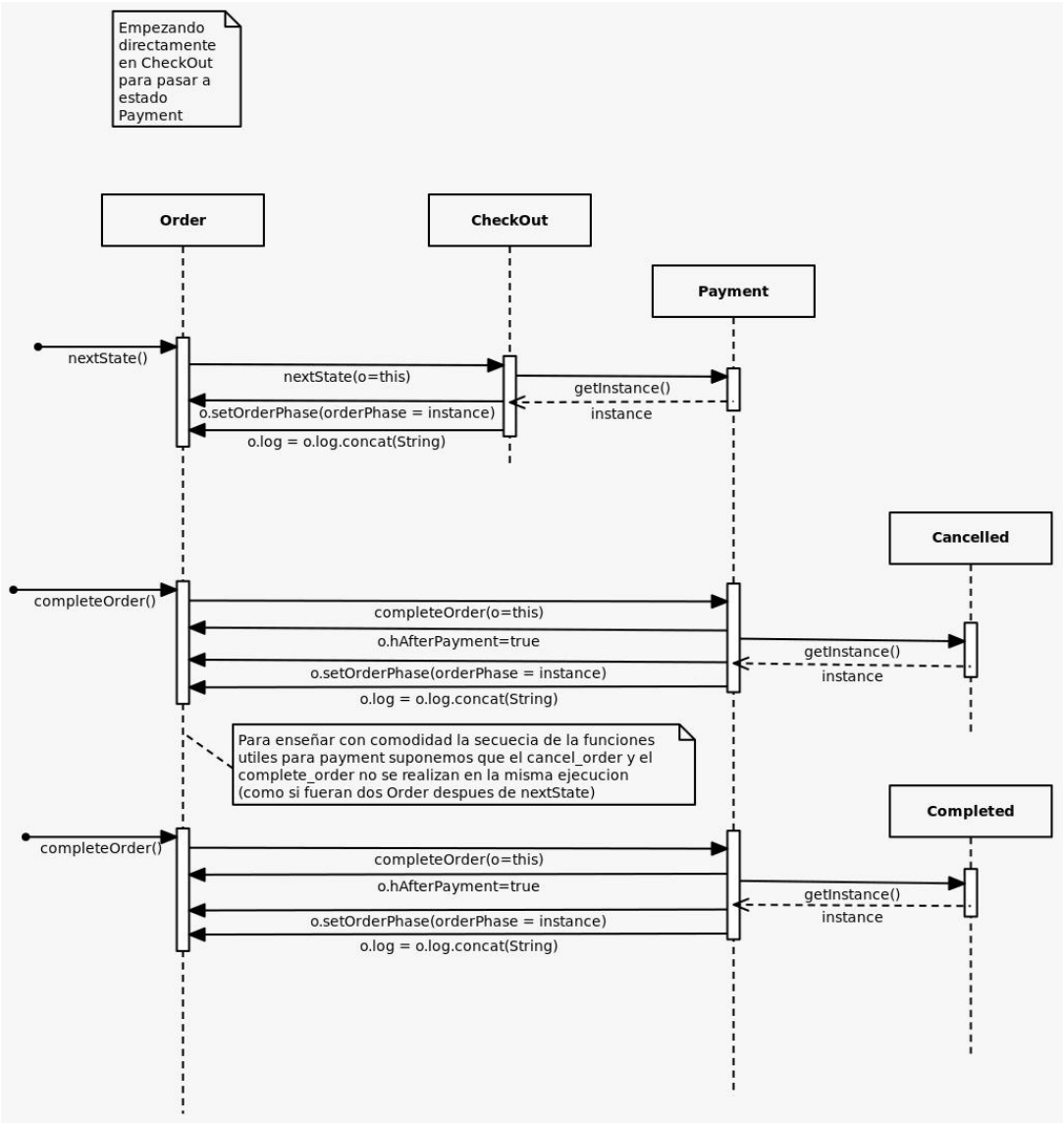
- Order juega el rol de Contexto.
- Phase juega el rol de Estado.
- ShoppingCart, Checkout, Payment, Cancelled y Completed juegan el rol de Estados concretos.

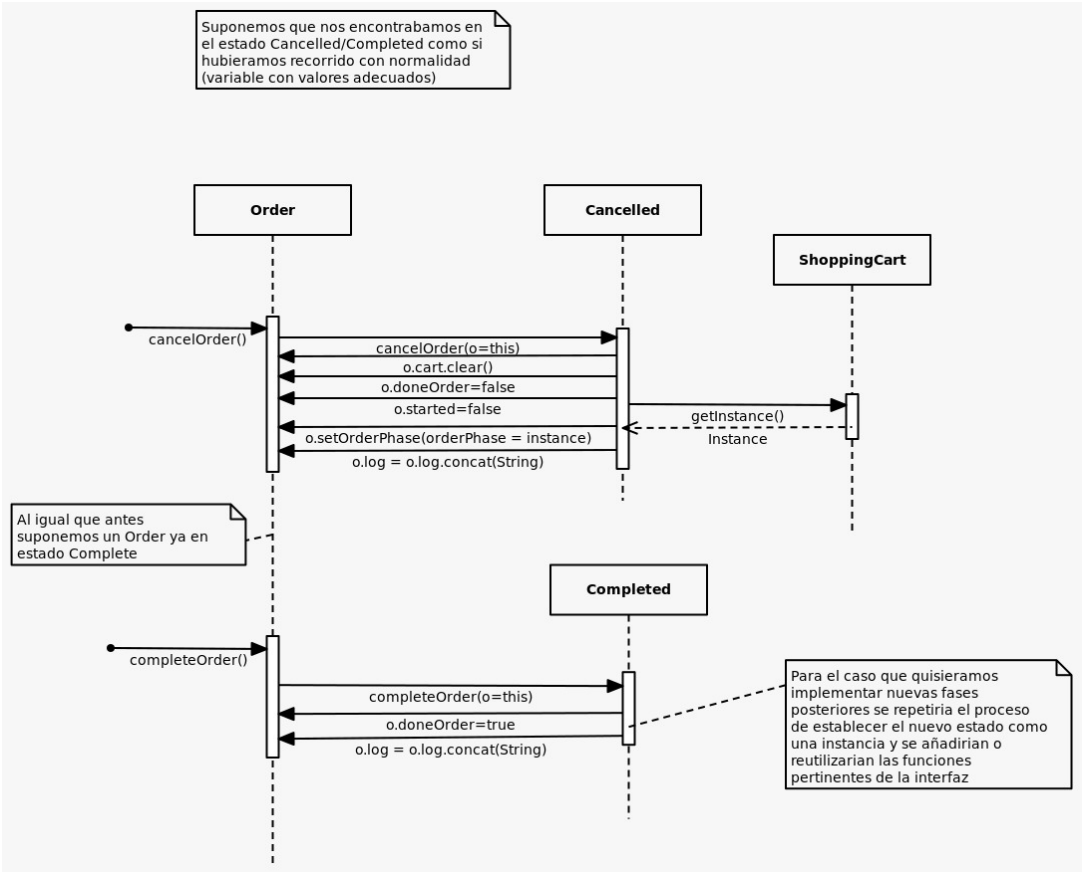
Cada estado concreto implementa los métodos del interfaz Phase y la clase Order utiliza los productos que se definen en la clase Product.

- Diagrama dinámico de secuencia:


Todos los metodos a excepcion de printLogy los getter/ setter de Order hacen una llamada polimorfica a Phase
Order
ShoppingCart
addProduct(product=cebolla,quantity=3)
add_product(o=this,product=cebolla,quantity=3)
o.cart.putIfAbsent(product.getProductId,quantity)
o.log=o.log.concat(String)
modifyQuantity(product=cebolla,quantity=7)
modify_quantity(o=this,product=cebolla,quantity=7)
O.cart.replace(product.getProductId(), quantity)
o.log=o.log.concat(String)
deleteProduct(pid=cebolla.getProductId())
deleteProduct(o=this,pid=cebolla.getProductId())
o.cart.remove(pid)
o.log=o.log.concat(String)
Se omite la secuencia de funciones como screenInfo() o printLog() pues solo muestran por pantalla un mensaje y lo retornan como String. Para el caso de screenInfo() un mensaje diferente para cada estado

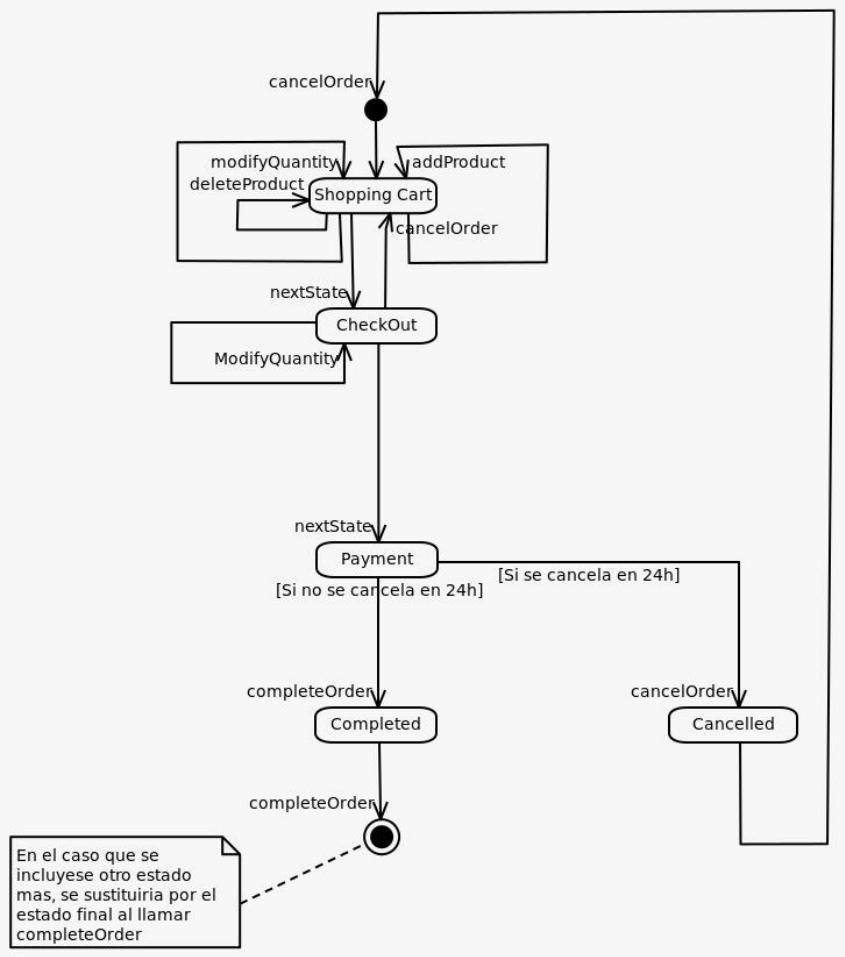

Order
ShoppingCart
CheckOut
nextState()
nextState(o=this)
getInstance()
instance
o.setOrderPhase(orderPhase = instance)
o.log = o.log.concat(String)
modifyQuantity(...)
modify_quantity()
ShoppingCart
Misma
implementacion
y retorno que
para estado
ShoppingCart
cancelOrder()
cancelOrder(o=this)
getInstance()
instance
o.setOrderPhase(orderPhase = instance)
o.log = o.log.concat(String)


Empezando directamente en CheckOut para pasar a estado Payment
Order
CheckOut
Payment
nextState()
nextState(o=this)
getInstance()
instance
o.setOrderPhase(orderPhase = instance)
o.log = o.log.concat(String)
Cancelled
completeOrder()
completeOrder(o=this)
o.hAfterPayment=true
getInstance()
instance
o.setOrderPhase(orderPhase = instance)
o.log = o.log.concat(String)
Para enseñar con comodidad la secuecia de la funciones utiles para payment suponemos que el cancel_order y el complete_order no se realizan en la misma ejecucion (como si fueran dos Order despues de nextState)
Completed
completeOrder()
completeOrder(o=this)
o.hAfterPayment=true
getInstance()
instance
o.setOrderPhase(orderPhase = instance)
o.log = o.log.concat(String)



- Hemos decidido separar el diagrama dinámico en varios para que se puedan ver mejor las interacciones entre las fases.

Diagrama dinámico de estados:



En respuesta a la parte final del ejercicio sobre añadir más fases en un futuro, de la forma en la que lo tenemos implementado sería muy sencillo y no modificaría prácticamente el resto de las clases. La nextState() de Completed sería la nueva fase Entrega, y mediante el valor de un boolean podemos saber si el pedido ha sido recogido o no. Basándonos en el tiempo transcurrido podemos comprobar si el paquete se ha perdido, funcionaría de la siguiente forma: si pasa un determinado número de días y el estado del boolean entregado es false, entonces el pedido se ha perdido.

En esta nueva fase podemos reutilizar el método de completeOrder para mostrar el estado de la entrega del pedido y el método cancelOrder para la devolución del pedido en el hipotético caso de que al comprador no le guste y quisiera devolverlo.

EJERCICIO 2

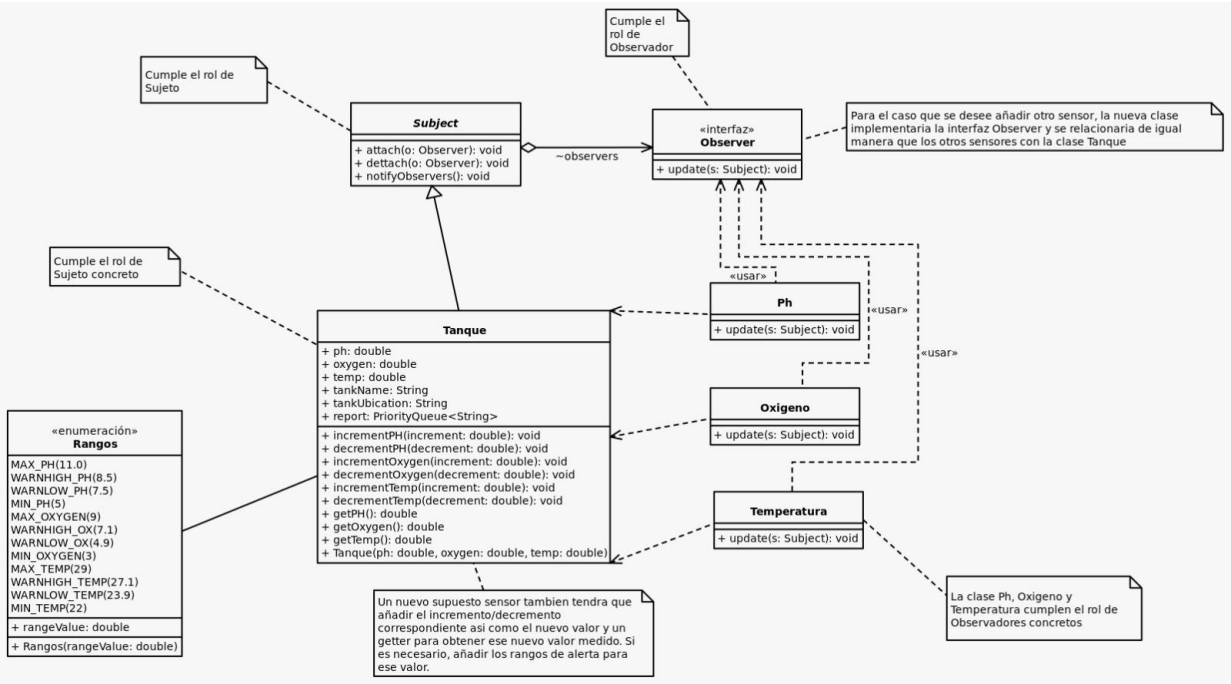
En el segundo ejercicio de la práctica de diseño se nos pide la gestión de un sistema de alertas para el control de los tanques del Aquarium Finisterrae de A Coruña. Tenemos tres sensores que comprueban que los niveles de pH, oxígeno y temperatura estén contenidos dentro de un rango de valores; en caso de que los valores no sean correctos se activarían 2 alertas dependiendo de dichos valores (alerta naranja y alerta roja).

Explicación de los principios SOLID empleados:

- Como en el anterior ejercicio, usamos el principio SOLID abierto-cerrado, ya que las clases que representan los sensores (Ph, Temp y Oxigeno) implementan los métodos creados en la interfaz Observer, siendo cada uno distinto a los demás. En el caso de que se añadan más sensores en un futuro, también implementaría los métodos de la interfaz Observer.
- Principio de Responsabilidad Única porque cada clase sensor depende únicamente de ella misma, por ejemplo: la clase Temp no depende de lo que suceda en las demás clases de sensores (Ph y Oxigeno), haciendo que no tengan responsabilidades entre ellas, pero sí con lo que suceda en el tanque (que suba la temperatura por encima del rango de valores correcto, por ejemplo).
- Principio Inversión de la Dependencia, usamos este principio porque dependen de interfaces y no de clases o funciones concretas, podemos ver este principio en el momento de crear la cola de prioridad.

Explicación del patrón de diseño usado:

- Para este ejercicio hemos usado el patrón Observador porque creemos que es el que mejor se ajusta a lo que se pide en el ejercicio, ya que nos piden que los sensores del acuario notifiquen a los Observadores en cuanto se produce un cambio en los tanques que puede hacer activar una alerta en ellos, y que desembocaría en su posterior arreglo poniendo el valor en un rango normal. Por ejemplo, si en la Piscina de Focas exterior se produce un cambio repentino en la temperatura que hace que se active una alerta naranja, se notificaría de inmediato a los dispositivos de control para restablecer la normalidad de los valores alterados.
- El uso de este patrón se ve de la mejor manera en el hecho de que el buen estado del tanque depende de tres factores: pH, temperatura y nivel de oxígeno en el agua; en el momento en el que uno de estos factores se salga del rango de valores adecuado, los dispositivos de control serán avisados y los valores serán actualizados.
- El patrón desconoce a la clase a la que pertenecen los observadores, solo sabe que implementan el interfaz Observer, y esto también es útil a la hora de añadir nuevos sensores, ya que no es necesario modificar el sujeto observado, además de permitir que los observadores y observados varíen de forma independiente.
- Diagrama de clases mediante el Patrón Observador:



En este patrón observador los roles serían los siguientes:

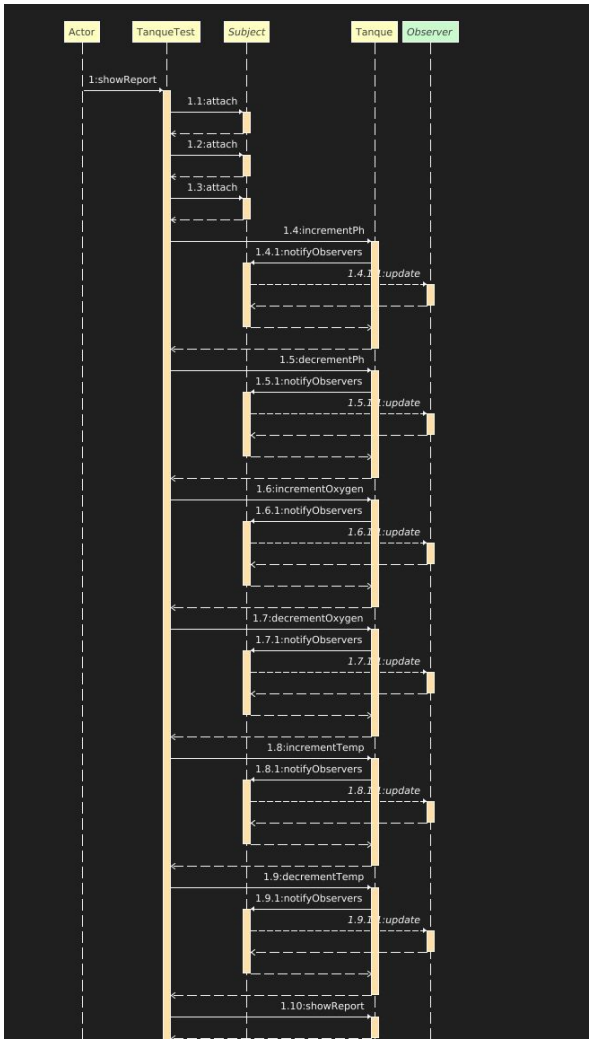
- Subject juega el rol de Sujeto.
- Observer juega el rol de Observador.
- Ph, Temperatura y Oxigeno juegan el rol de Observadores concretos.
- Tanque juega el rol de Sujeto concreto.

Cada observador concreto implementa el método update de la interfaz Observer de su forma correspondiente, añadiendo el informe a la cola de prioridad en función de que sea una alerta roja o naranja, y cambiando el valor anómalo del sensor por un valor aleatorio dentro del rango de valores adecuado.

Basándonos en información obtenida de distintas páginas web relacionadas con el mantenimiento de piscinas y acuarios marinos, hemos ideado la siguiente tabla de valores en la que se representan los rangos numéricos (el primer rango de valores normales y el segundo cuyos extremos sean los de las alertas naranjas):

- Nivel de pH:
    - Nivel normal: 7,5 – 8,5.
    - Alerta naranja: 5 – 7,4 y 8,6 – 11.
    - Alerta roja: 0 – 4,9 y 11,1 y 15.
- Nivel de oxígeno en agua:
    - Nivel normal: 5 – 7 mg/l.
    - Alerta naranja: 3 – 4,9 y 7,1 – 9 mg/l.
    - Alerta roja: 0 – 2,9 y más de 9 mg/l.
- Nivel de temperatura del agua:
    - Nivel normal: 24 – 27 grados.
    - Alerta naranja: 22 – 23,9 y 27,1 – 29 grados.
    - Alerta roja: menos de 22 grados o más de 29 grados.

Diagrama dinámico de secuencia:



En respuesta a lo que se solicita al final del ejercicio, la forma en la que solucionamos el problema es la óptima para añadir nuevos sensores en un futuro. Para añadir un sensor nuevo, por ejemplo, el nivel del agua, sólo tendríamos que crear una nueva clase Agua que implemente el método update de la interfaz Observer de forma que actualice los valores de la forma correspondiente. También sería necesario establecer un getter para obtener el nivel de agua del tanque en cuestión, además de un método incrementAgua, un método decrementAgua y su constructor. Para comprobar si se sale del rango de valores normales deberemos actualizar el enum Rangos con los valores de los dos rangos numéricos que representan los niveles de agua.

De esta forma se podrían añadir multitud de sensores en un futuro en los que se controlen sus valores y se modifiquen en caso de que estos sean anómalos.

Eloy Calvo Gens (grupo 1.1) eloy.calvo.gens@udc.es

Diego González González (grupo 2.3) diego.gonzalez7@udc.es