Sergio Castrillón – s.castrillon
Mario Freire – mario.freire

INFORME

PRÁCTICA DE DISEÑO

1. Sistema de compra online

Principios de diseño

En el ejercicio 1 seguimos los principios de responsabilidad única, abierto-cerrado y inversión de la dependencia.

Principio de Responsabilidad Única

El principio de responsabilidad única establece que cada objeto debe tener una única responsabilidad y que la responsabilidad esté encapsulada dentro de la clase. Por tanto los servicios que provee el objeto están ligados con dicha responsabilidad. En nuestro caso se puede observar que Order delega la responsabilidad en la interfaz ProductState y son las clases que implementan la interfaz las que realizan el trabajo. Además las distintas clases que necesitan cambiar la cantidad de un producto delegan esa tarea a ProductCount que se encarga de la gestión de las cantidades de productos pedidas por el cliente mientras que esta clase a su vez delega a Product la gestión del stock (y la comprobación de no estar pidiendo más productos de los que hay en stock).

Principio Abierto-Cerrado

El principio abierto-cerrado indica que si es necesario añadir nuevas funcionalidades al programa no debería de ser necesario modificar las clases ya existentes para añadir dichas funcionalidades, es decir que se deben crear las clases de manera que sean compatibles con futuras extensiones. En este ejercicio debido al patrón seguido no es posible seguir este principio estrictamente, aún así la interfaz OrderState permite añadir nuevas funcionalidades fácilmente, modificando esta interfaz y creando nuevas clases. Aún así se puede necesitar hacer cambios mínimos a algunas clases ya existentes como Order.

Por ejemplo para permitir el seguimiento de la entrega de un pedido haría falta añadir un método en Order y en OrderState que permita el cambio de estado a entrega (o añadir alguna condición en algún método ya existente) e implementar el método en algún estado anterior para pasar a estado de entrega, pero el resto de clases no relacionadas no necesitan cambiar su código debido a que todos los métodos de la interfaz ProductState tienen (y deberían seguir

teniendo tras la extensión) una implementación por defecto. Esto además permite crear clases que solo implementan el método que se quiere modificar.

(Para añadir el seguimiento de la entrega al Log haría falta añadir un método a Log).

Principio de Inversión de la Dependencia

El Principio de la Inversión de la Dependencia establece que deberíamos depender de abstracciones en vez de concreciones en otros términos depender de interfaces o clases abstractas en vez de clases o funciones concretas. En este ejercicio se cumple perfectamente debido al patrón seguido, Order depende la interfaz OrderState (y de la interfaz que hereda de OrderState, AbstractOrderState), y así que el funcionamiento de los métodos de Order depende de las clases que implementan la interfaz OrderState, permitiendo que el comportamiento de Order cambie según las clases de OrderState.

Patrones de diseño

En el ejercicio 1 aplicamos el patrón Estado y el patrón Instancia Única.

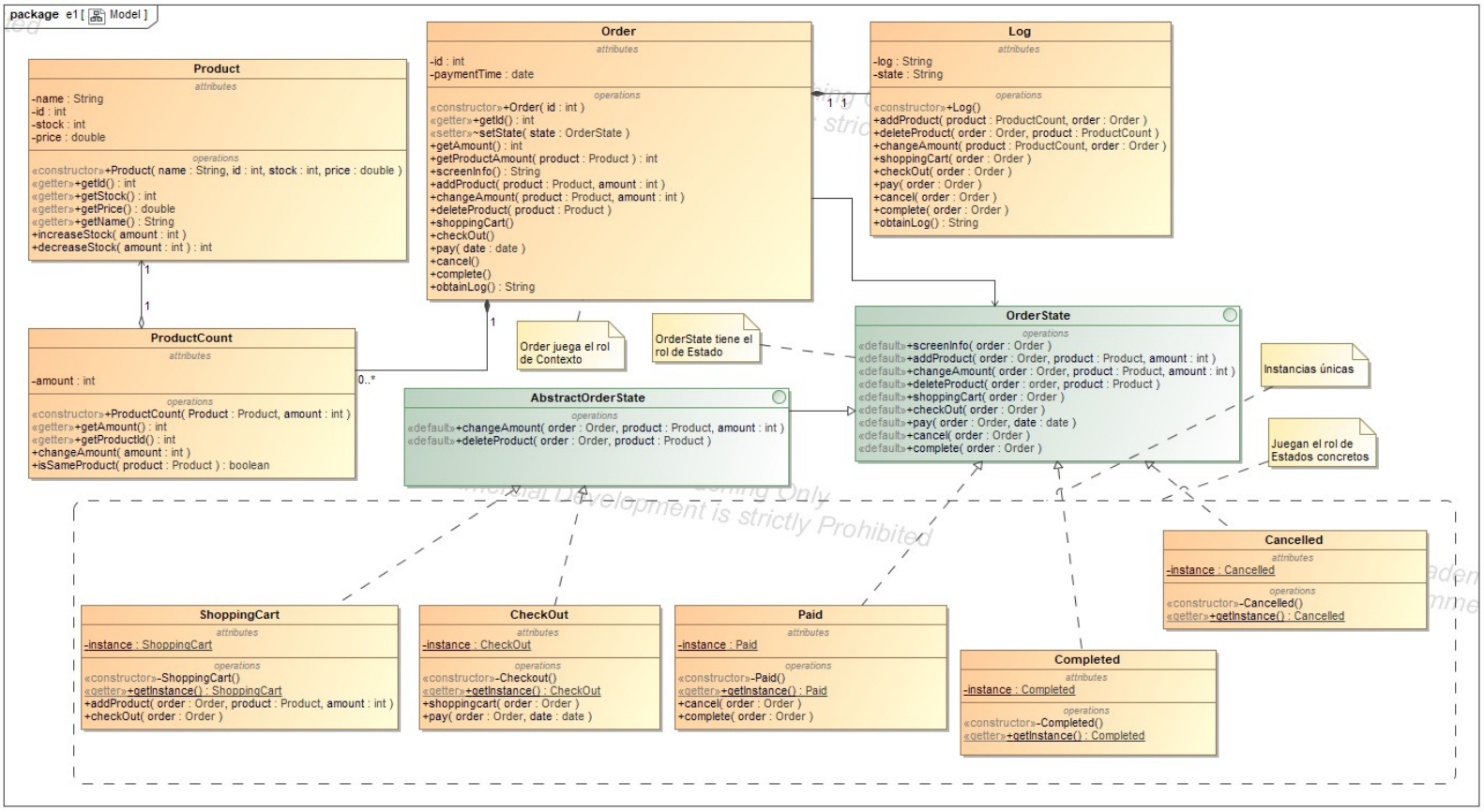
Patrón Estado

El patrón estado es un tipo de patrón que permite crear objetos que cambian su comportamiento al cambiar el estado interno del objeto. Se eligió este patrón porque Order debe cambiar el comportamiento de las operaciones que tiene (en este caso bloquear o permitir ciertas operaciones) dependiendo del estado interno en el que se encuentre la orden, por poner un ejemplo, no podemos permitimos cambiar productos de la orden una vez ya ha sido pagada.

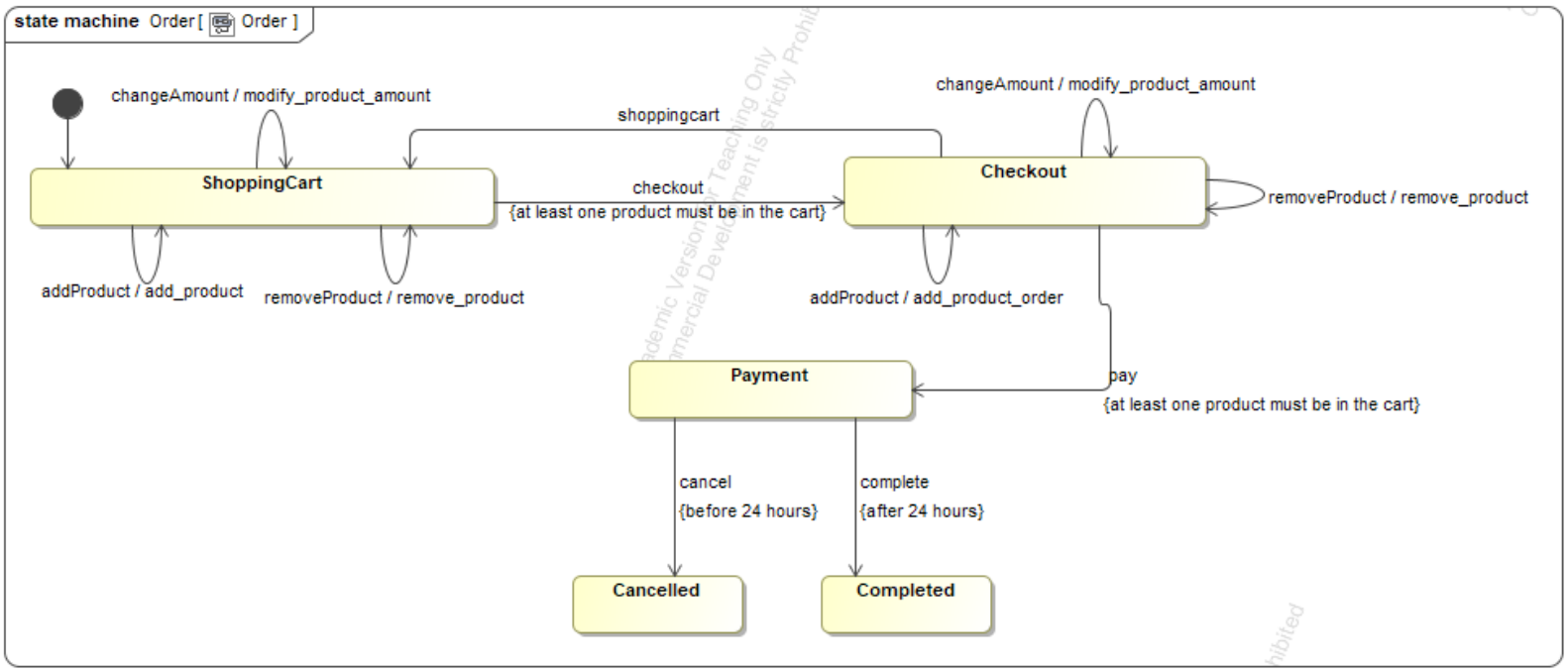
En el diagrama podemos observar que la clase Order juega el rol de *contexto*, OrderState el rol de e*stado* y todas las clases que implementan OrderState (o AbstractOrderState) juegan el rol de *estado concreto.*

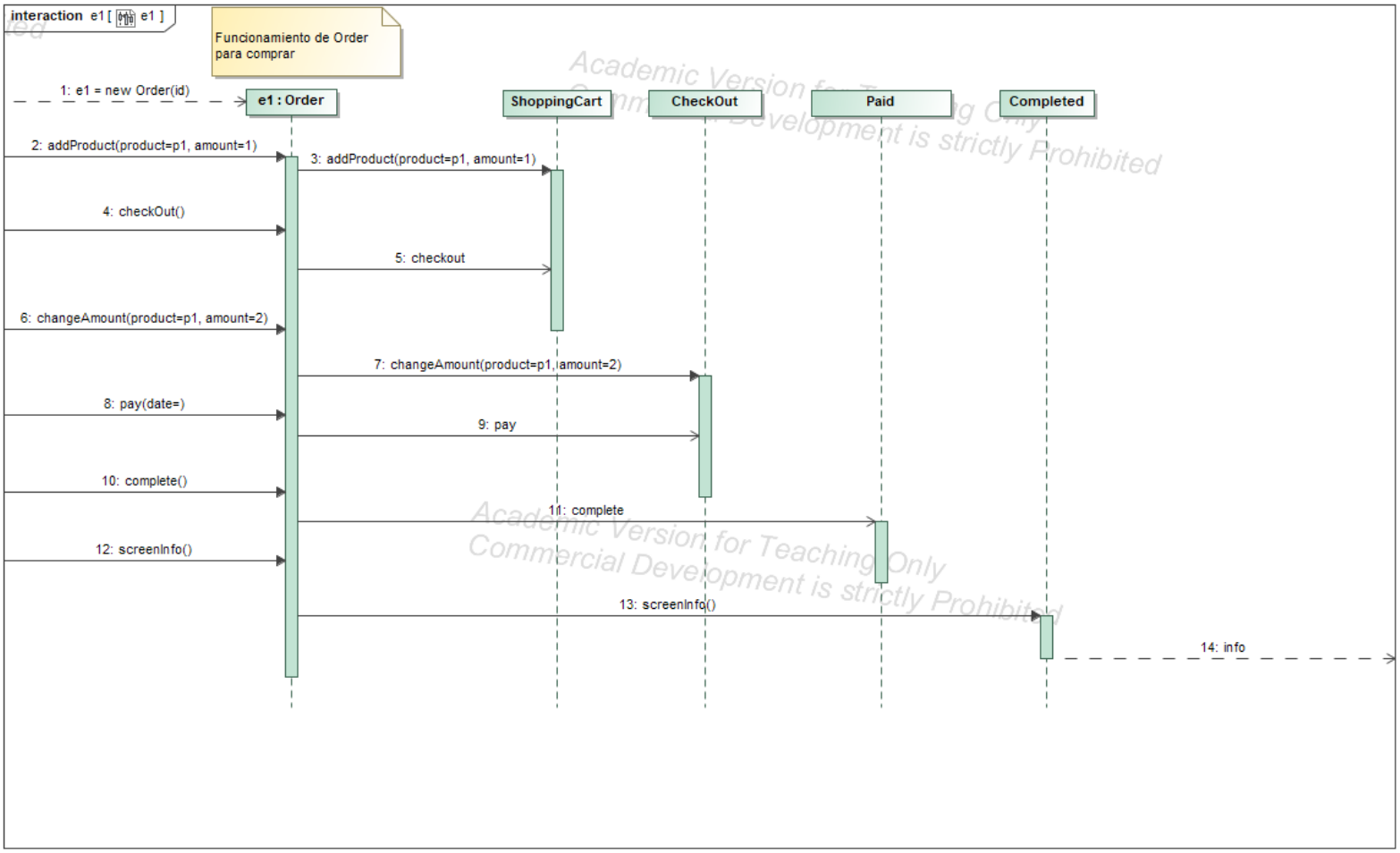
Patrón Instancia Única

El patrón instancia única se usa cuando queremos asegurarnos de solo tener una instancia de una clase manteniendo el acceso global a ella, con esto conseguimos reducir el uso de recursos al evitar crear instancias de clases que no son necesarias como es nuestro caso. Como las clases estado no contienen atributos (por lo que no pueden cambiar de una instancia a otra) y el comportamiento de sus métodos no varía entre instancias, podemos aplicar este patrón a todas las clases estado tal y como se observa en el diagrama.



Diagramas dinámicos





2. Gestión de alertas

Principios de diseño

En el ejercicio 2 seguimos los principios de responsabilidad única, inversión de la dependencia y abierto-cerrado.

Principio de Responsabilidad Única

El principio de responsabilidad única establece que cada objeto debe tener una única responsabilidad y que la responsabilidad esté encapsulada dentro de la clase. Por tanto los servicios que provee el objeto están ligados con dicha responsabilidad. En el ejercicio podemos comprobar que en general todas las clases cumplen el principio, por ejemplo Tank solo almacena sensores y los parámetros del Tanque, Sensor permite la lectura del valor que guarda y se encarga de alertar a sus observadores del cambio, Alert recibe el valor notificado y hace sus comprobaciones para avisar a sus observadores si es lo necesario...

Principio de Inversión de la Dependencia

El principio establece depender de abstracciones en vez de concreciones. El ejercicio como se basa en un patrón Observador, presenta dos interfaces Observador que le permiten a Sensor y a Alert no saber de que clase son sus observadores, permitiendo añadir nuevos Observadores sin cambiar el funcionamiento o implementación de alguno de ellos.

Principio Abierto-Cerrado

El principio abierto-cerrado indica que si es necesario añadir nuevas funcionalidades al programa no debería de ser necesario modificar las clases ya existentes para añadir dichas funcionalidades, es decir que se deben crear las clases de manera que sean compatibles con futuras extensiones. Bajo nuestra implementación, la creación de sensores que lean nuevos parámetros no requiere crear nuevas clases, solo hace falta establecer el parámetro typeSensor con un nombre adecuado y que las alertas asociadas al sensor se ajusten adecuadamente al nuevo parámetro. Por otra parte la creación de nuevos observadores también es sencilla debido a la interfaz Observer. En el resto de clases no es deseable que se permita una mayor abstracción debido a lo pedido por el enunciado, por lo que el esquema debe estar basado en Sensores y Alertas y no permitir nuevas clases que las puedan sustituír.

Patrones de diseño

En el ejercicio 2 aplicamos el patrón Observador.

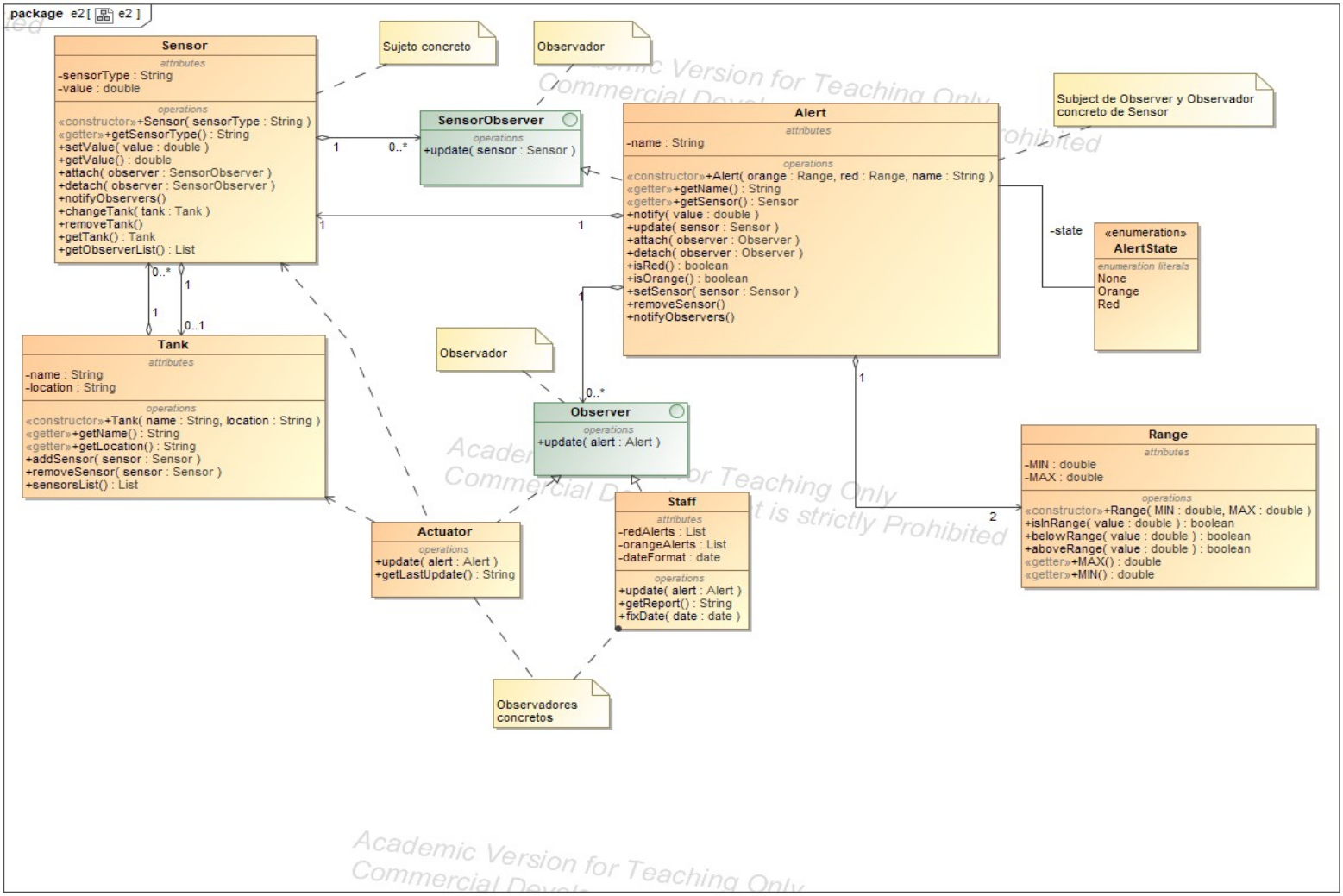
Patrón Observador

El patrón Observador permite crear una dependencia entre objetos de manera que cuando uno de ellos (el Sujeto) cambie de estado, los objetos dependientes (los Observadores) sean notificados y puedan actuar en consecuencia.

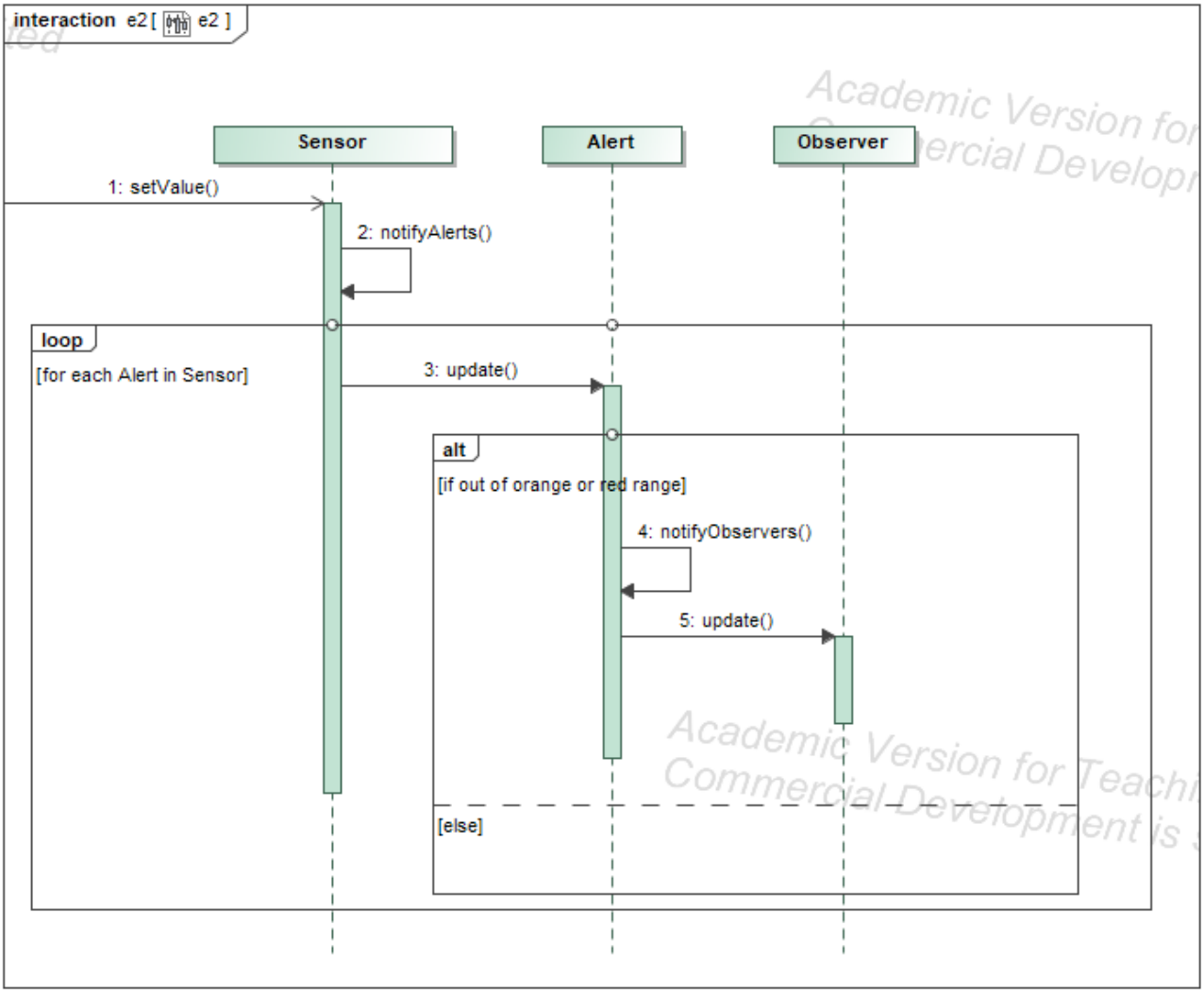
Este patrón cuadra perfectamente con el ejercicio porque Alert debe ser notificado de los cambios de Sensor y a su vez Staff y Actuator debe recibir algunos cambios en las alertas, pero no todos (no se debe avisar si la alerta se desactiva).

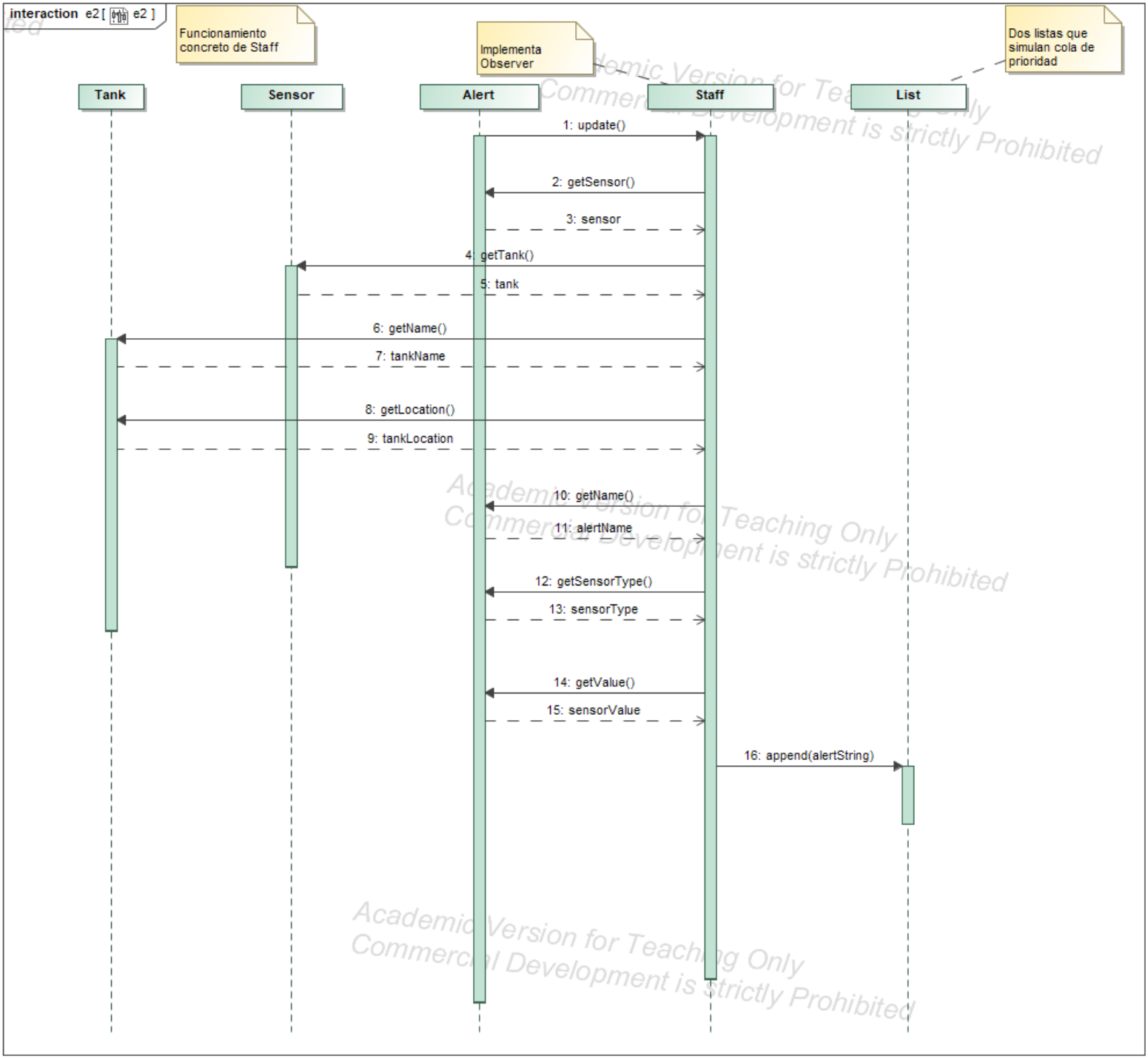
El patrón es aplicado dos veces, ambas siguiendo el esquema pull. De esta manera, SensorObserver es un observador de Sensor, los sensores son sujetos concretos, las alertas sus observadores concretos y a su vez las alertas actúan como sujeto concreto de la interfaz Observer que actúa como observador y las clases que la implementan (Actuator y Staff) como observador concreto. Por otra parte, para los requisitos del ejercicio modificamos ligeramente el patrón, por ejemplo eliminando las clases abstractas Sujeto, debido a que el ejercicio se basa en sensores y por tanto no esperamos añadir nuevos elementos a observar, además como

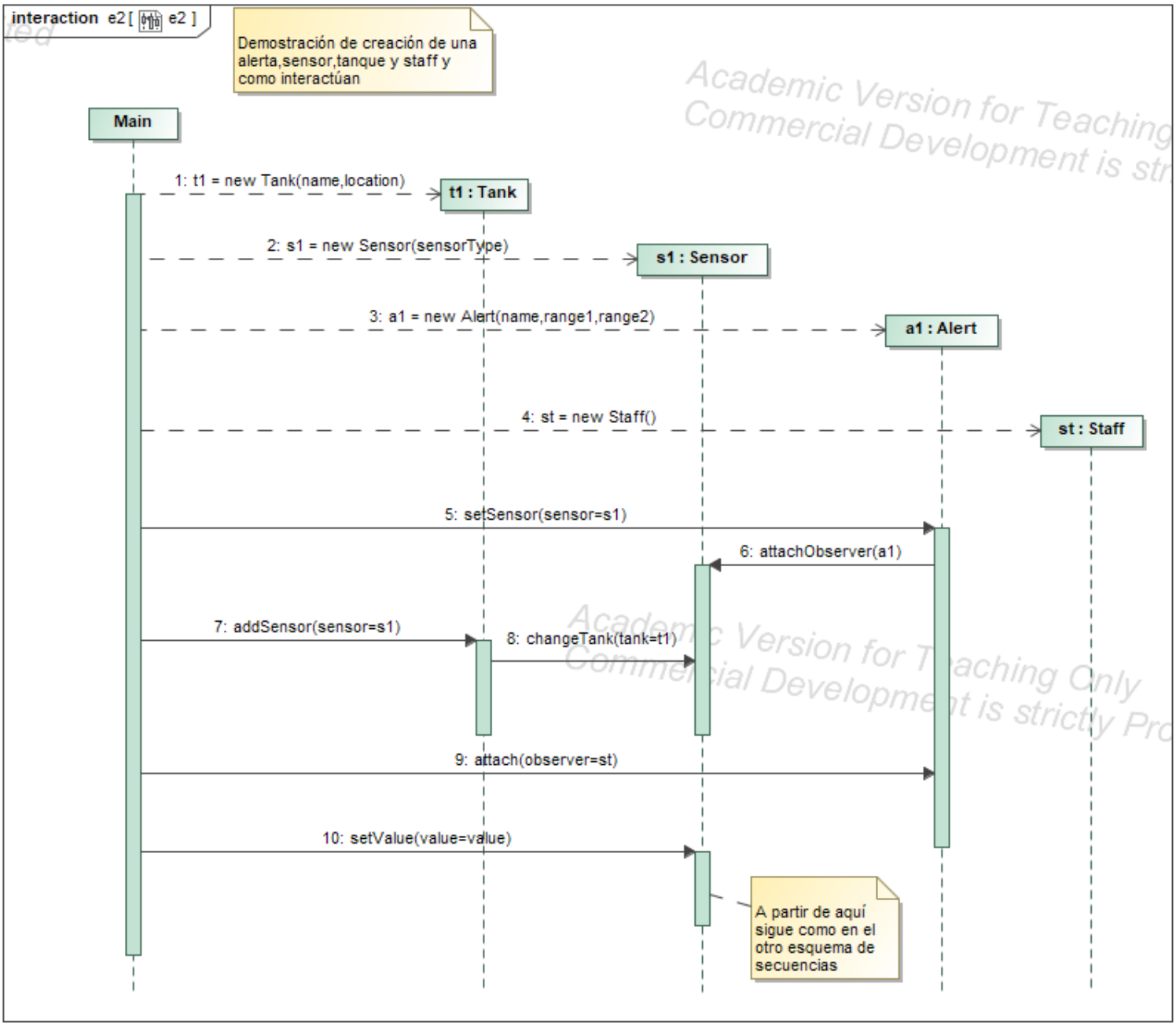
tenemos dos interfaces Observador, no podemos crear una clase abstracta sujeto de la que hereden Sensor y Alerta (ya que reciben interfaces Observer diferentes).



Diagramas dinámicos






interaction e2 [e2]
Demostración de creación de una alerta,sensor,tanque y staff y como interactúan
Academic Version for Teaching
Commercial Development is str
Main
1: t1 = new Tank(name,location)
t1 : Tank
2: s1 = new Sensor(sensorType)
s1 : Sensor
3: a1 = new Alert(name,range1,range2)
a1 : Alert
4: st = new Staff()
st : Staff
5: setSensor(sensor=s1)
6: attachObserver(a1)
7: addSensor(sensor=s1)
8: changeTank(tank=t1)
Academic Version for Teaching Only
Commercial Development is strictly Pro
9: attach(observer=st)
10: setValue(value=value)
A partir de aquí sigue como en el otro esquema de secuencias