INFORME PRACTICA DE DESEÑO – MERCADO DE ACCIONES

1.Explicación de los principios de diseño (SOLID) usados

Comenzamos con el principio de responsabilidad única, cada clase está estrechamente relacionada con su responsabilidad, es decir tanto la clase Accion como ClienteSencillo y ClienteDetallado se centran única y exclusivamente en su responsabilidad.

Continuamos con el principio abierto-cerrado, nuestro código se puede extender muy fácilmente añadiendo nuevos clientes con distintas necesidades (que implementen Observer) y sin necesidad de modificar el código ya escrito.

El principio de sustitución de Liskov no es utilizado puesto que no pasamos ninguna subclase como parámetro, el principio de Inversión de la dependencia tampoco sería usado, las clases abstractas e interfaces usadas no tienen la función de invertir la dependencia. También mencionar al principio Kiss pues intentamos mantener un código lo mas simple y claro posible.

Por último, el principio de Segregación de interfaces es usado con Observer y Observable debido a que estamos segregando 2 capacidades que puede poseer o realizar una clase en una interfaz y una clase abstracta, esto también favorece al principio abierto-cerrado.

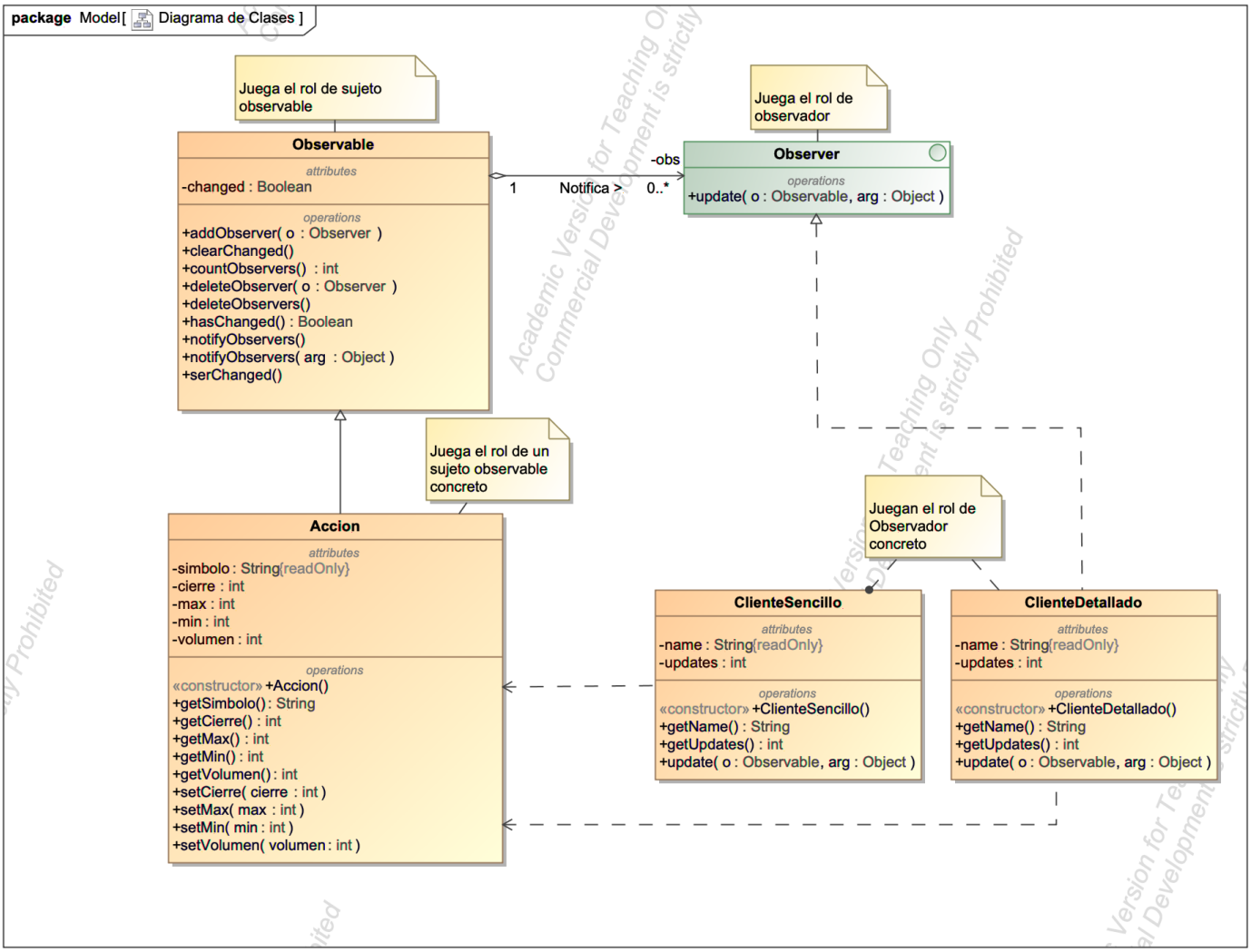
2.Explicación del patrón de diseño utilizado

La elección de este patrón fue muy simple, el ejercicio nos pedía que codificásemos una clase acción, la cual tiene distintos atributos y unos clientes, estos clientes no están interesados todos en la misma información de la acción. También nos dice que cuando se cambie cualquier dato de la acción, los clientes a los cuales les interese deben ser notificados.

Lo que necesitamos es el patrón observador. La acción será la observada y los distintos clientes los observadores, cuando se produzca un cambio en la clase acción por medio de los seters, estos llamaran a un método que notificará a todos los observadores (están completamente desacoplados de la clase accion). El API de java ya incluye una clase abstracta Observer y Observable y esto nos facilitará mucho el trabajo a la hora de programar.

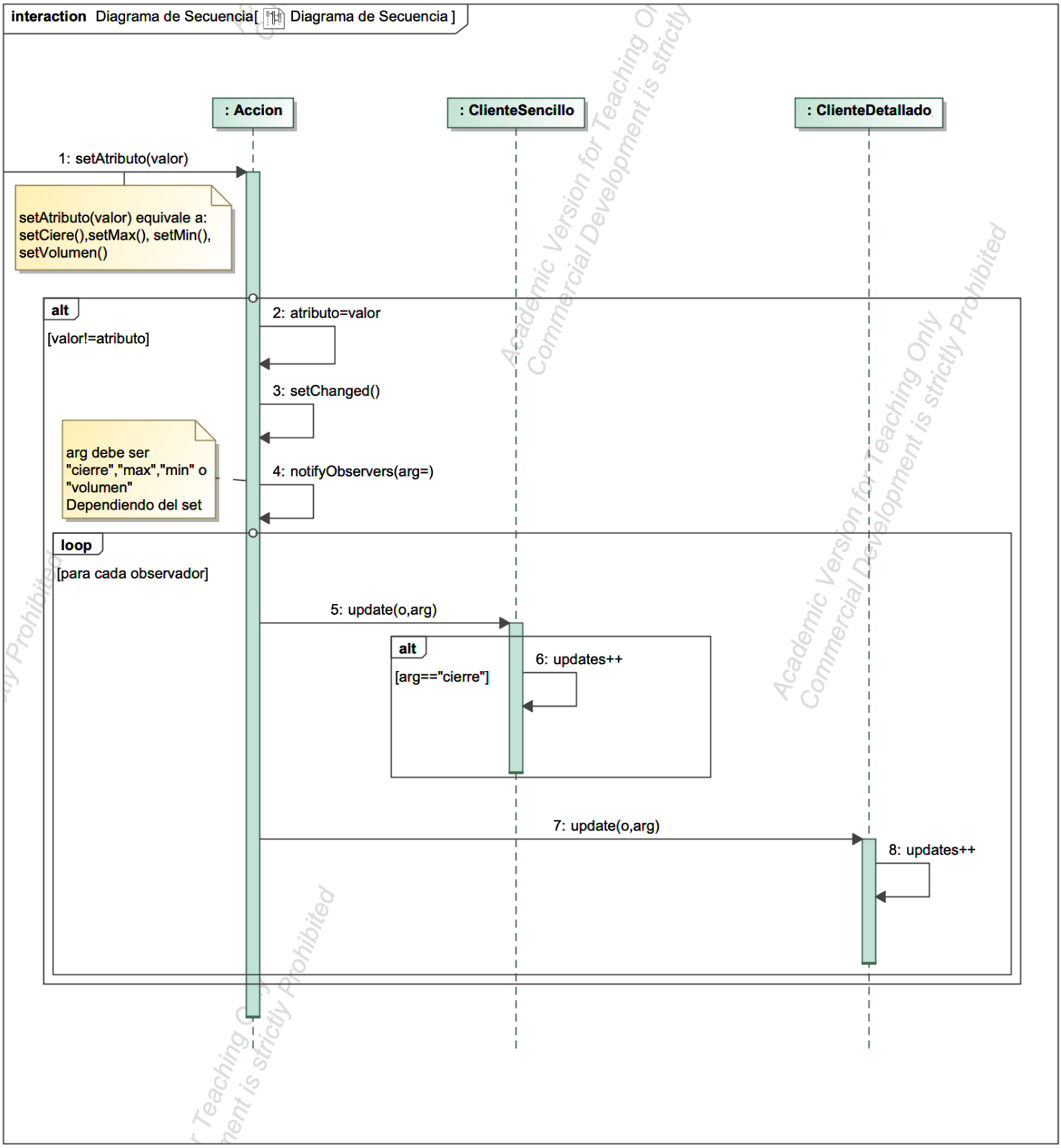
En el momento de tener que realizar la notificación podemos usar dos técnicas (PULL y PUSH). Usaremos push ya que necesitamos saber cuál fue el dato que cambió (pasamos un string que nos indica que atributo cambió) y saberlo sin tener esa ayuda haría que tuviésemos un código mucho mas complejo y difícil de entender.

3.Diagrama de clases



4.Diagramas dinámico

Hemos realizado un diagrama de secuencia ya que la principal característica de este patrón es poder notificar a los observadores, y esta característica podrá ser reflejada de mejor forma en el diagrama de secuencia (representa objetos intercambiando mensajes) .


interaction Diagrama de Secuencia[Diagrama de Secuencia]
: Accion
: ClienteSencillo
: ClienteDetallado
1: setAtributo(valor)
setAtributo(valor) equivale a:
setCiere(),setMax(), setMin(),
setVolumen()
alt
[valor!=atributo]
2: atributo=valor
3: setChanged()
arg debe ser
"cierre","max","min" o
"volumen"
Dependiendo del set
4: notifyObservers(arg=)
loop
[para cada observador]
5: update(o,arg)
alt
[arg=="cierre"]
6: updates++
7: update(o,arg)
8: updates++