PRINCIPIOS DE DISEÑO

-Principio de responsabilidad única:

Reparte las responsabilidades entre varias clases, evitando que una clase “Dios” haga todo. Se reparten los trabajos entre las clases, lo que reduce el acoplamiento y aumenta la cohesión. Por ejemplo, las clases “Tanque”, “Actuador”, “Alerta” y “Sensor” trabajan conjuntamente. Seria posible tener una sola clase con la información del tanque y las funciones de alerta, actuador y sensor; pero no estaríamos repartiendo responsabilidades, sin embargo al tener estas clases separadas, trabajan conjuntamente llamándose entre ellas cuando lo necesitan. También la clase “Sujeto” se apoya en “Sensores”, al igual que “Personal” en esta y en “Alerta”, así al final acaban todas trabajando como una única clase “Dios”.

-Principio abierto – cerrado:

Para la misma clase, a veces buscamos comportamientos diferentes, el principio abierto-cerrado nos asegura poder modificar el comportamiento de una clase sin tener que modificar la clase en si, además permitiendo que clases nuevas extiendan a la principal aún sin tenerlas contempladas en un principio. Por ejemplo en nuestro código, la clase “Sensor” extiende a “Sujeto”, por lo que si queremos ampliar los tipos de sensores, uno nuevo tendrá las funciones de sujeto sin dar fallo y sin necesidad de cambiar la clase “Sujeto”, si lo que queremos es que el nuevo sensor tenga su propio comportamiento, podemos redefinir en la clase del nuevo sensor los métodos que creamos oportunos de la clase “Sujeto”. También “Personal”, que implementa “Observador” podría implementar nuevo personal sin necesidad de reescribir el código de la clase “Personal”, ya que hay una estructura de personal por defecto.

En el ejercicio 1, la clase “Product” también cumple este principio, ya que estamos cubiertos ante la llegada de nuevos tipos de productos con diferentes comportamientos gracias a su implementación por defecto.

-Principio de inversión de dependencias:

Lo que pretende este principio es abstraerse de la implementación lo máximo posible para hacer el código más eficiente. Además esto facilita futuros cambios en el código, reduciendo la cantidad de este que hace falta modificar. Es imposible abstraerse totalmente de la implementación, pero la clave está en un equilibrio entre la abstracción y el conocimiento.

En nuestro código implementamos este principio en la clase “Personal” por ejemplo, que tiene 2 listas que creamos de tipo list pero con el constructor de arraylist, para permitir una posible modificación futura a otro tipo de implementación de list.

También se cumple el principio mediante la inyección de dependencia en “Alerta”, “Sensor”, “Tanque”.

En el ejercicio 1, en la clase “Order” también implementamos dos listas con tipo list pero con constructor arraylist, para permitir un posible cambio a otro tipo de lista.

Al igual que “Product”, mediante la inyección de dependencia.

-Principio de segregación de interfaces:

Ayudándonos del principio de inversión de interfaces, nos damos cuenta de que algunas interfaces pueden no ser del todo eficientes o ser muy generales. A la hora de implementarlas debemos pensar en como serán usadas. Por eso mismo las interfaces que creamos deberían ser más concretas, por ejemplo, una empresa puede querer información sobre la interfaz trabajador, pero un trabajador puede hacer cosas que no le importan a la empresa, como relajarse por ejemplo. La empresa no debería tener por qué implementar relajarse para sus trabajadores ya que a ellos no les incumbe; así que, siguiendo el principio de segregación de interfaces, separamos la interfaz trabajador en trabajador y relajador. Los nuevos empleados de la empresa implementarán el método trabajador y no el relajador, ya que para la empresa ellos solo son trabajadores.

En nuestro código implementamos el principio de segregación de interfaces en la clase “Personal”, solo hay una interfaz, pero si nuestro personal fuera más extenso y se comportase como un humano además de como un trabajador, nuestro código estaría cubierto ante este principio. No hay segregación de interfaces en el ejercicio 1.

PATRÓN DE DISEÑO

-Ejercicio 1:

Elegimos el patrón estado para el ejercicio 1 ya que para implementar el funcionamiento de un carrito de la compra, este debe ir cambiando de estado para comportarse de una u otra forma. Esta solución es la mejor para un número no muy extenso de estados, ya que hay pocos métodos afectados por el estado y estos son estables. Aunque el patrón estado y el estrategia tienen un funcionamiento similar, el problema que tratan de resolver es distinto. Lo que queremos con el patrón estado es conseguir un funcionamiento diferente dependiendo de los estados, cosa que no conseguimos con el estrategia. Los estados se conocen entre ellos ya que establecen sus transiciones. Además el patrón estado permite añadir nuevos estados de ser necesario de forma sencilla solamente creando clases.

El patrón estado por todo esto es el mas indicado para resolver el problema.

-Ejercicio 2:

El patrón observador, el que elegimos para este ejercicio, desacopla al objeto de sus observadores, lo que permite añadir nuevos de estos sin modificar los sujetos observados, de esta forma; observadores y observados pueden variar de forma independiente. Implementamos una clase observable que juega el rol de sujeto y otra observer que será el observador.

Gracias al patrón observador, si desacoplamos los observadores de los observables, los segundos no necesitan conocer la clase concreta de los primeros, ya que solo notifican de cambios; una vez hecho esto, será responsabilidad de los observadores actuar en consecuencia.

Las desventajas se encuentran en el momento en que varios observadores coexisten, ya que si diversos de estos observan al mismo observable, este puede sufrir actualizaciones simultáneas, debido a que los observadores no se relacionan entre sí.

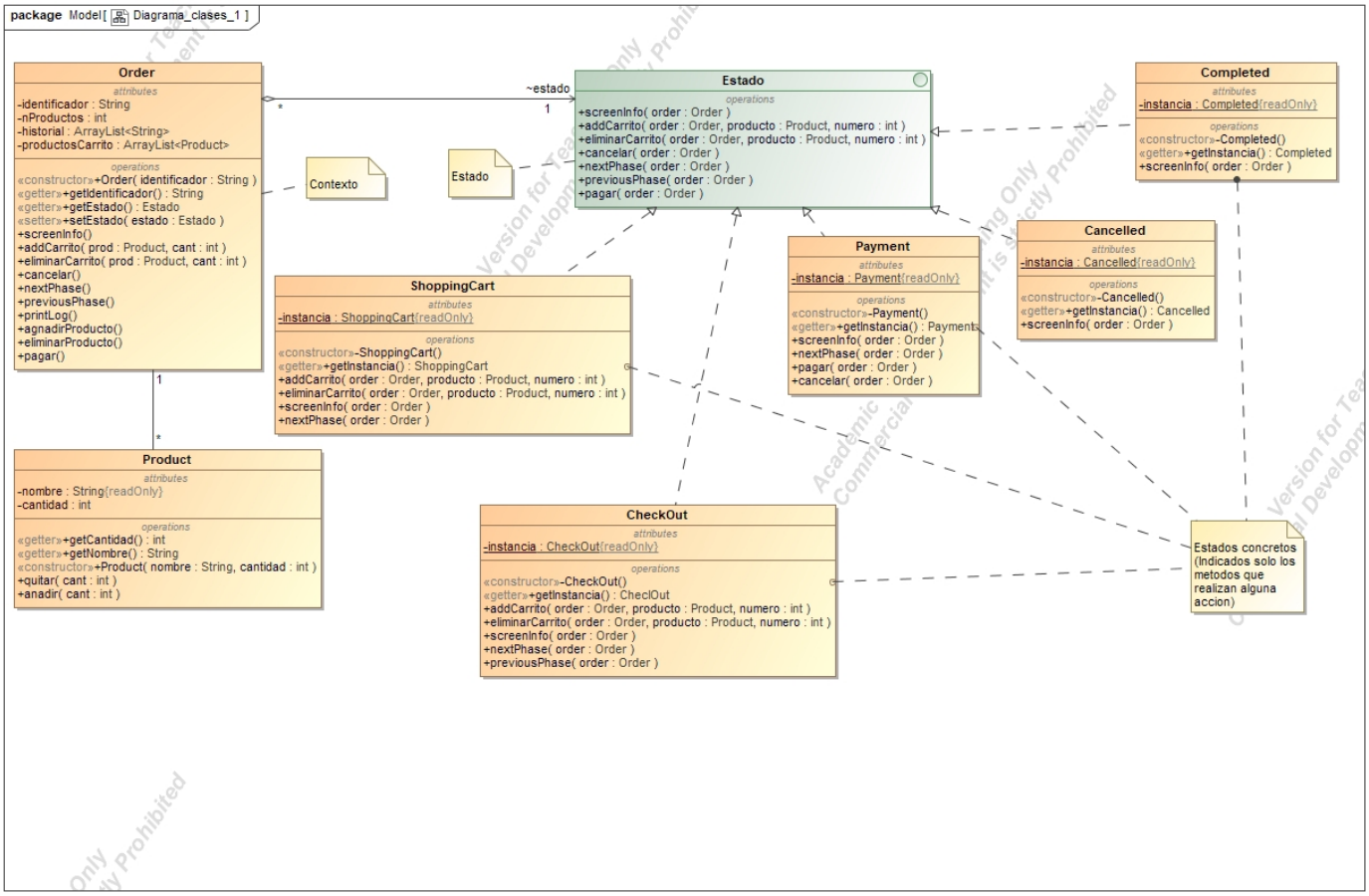
Además, debido a la simpleza de la relación observador-observado, es competencia del observador conocer que ha cambiado en el observado. Hay dos modelos para esto, el modelo pull(el implementado en nuestro ejercicio) y el push.

Las relaciones observado-observador del problema serían las existentes entre sensor-alerta y entre alerta-actuador/personal.

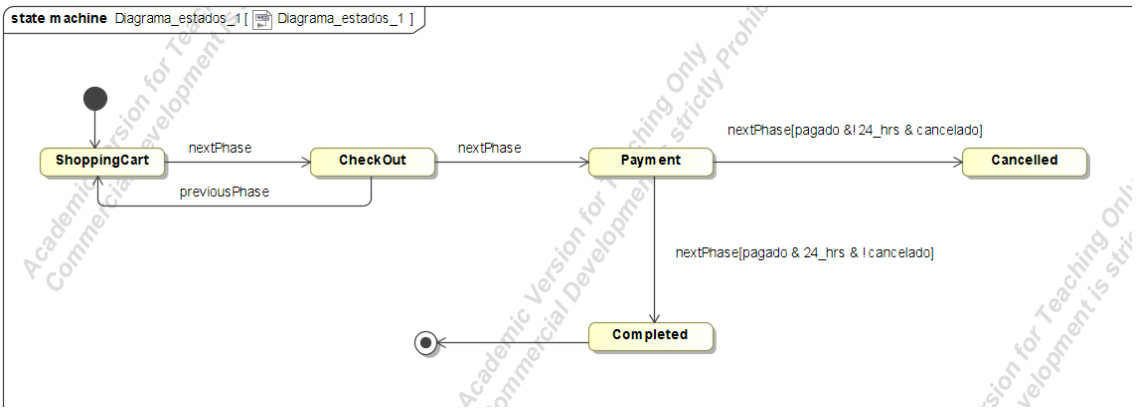
DIAGRAMAS UML

-Ejercicio 1:

-Diagrama de clases:

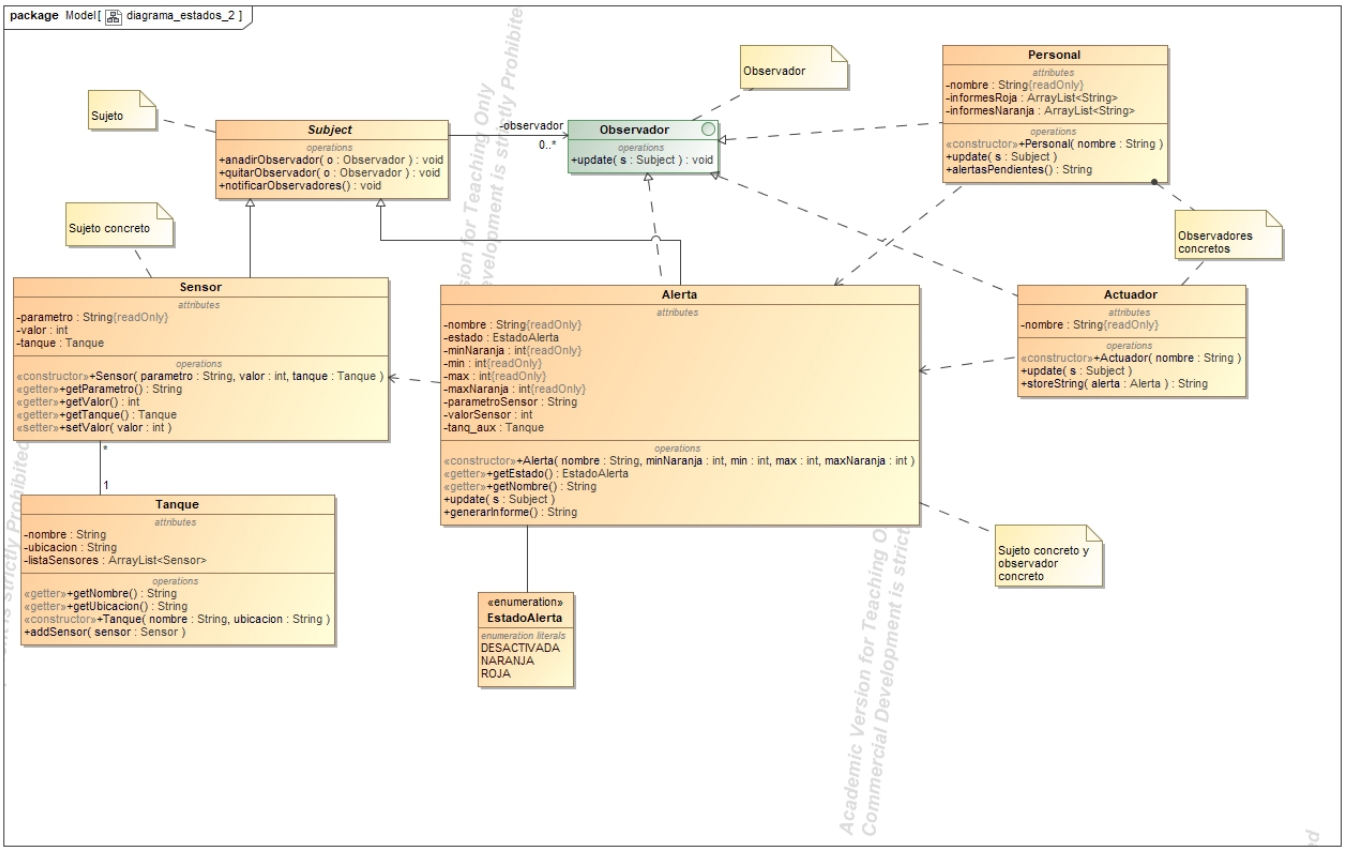


-Diagrama de estados:



-Ejercicio 2:

-Diagrama de clases:



-Diagrama de secuencia: