DOCUMENTACIÓN PARA LOS EJERCICIOS CONTENIDOS DENTRO DEL BOLETÍN 3 -> DS-34-01-PD

(GRUPO 34-01-> david.molares@udc. es y oscar.perez.sanchez@udc.es)

/EJERCICIO 1/

/EXPLICACIÓN DE LOS PRINCIPIOS SOLID USADOS EN EL CÓDIGO/:

-Principio de responsabilidad única -> Se aplica al cambio establecido para devolver las vueltas al comprar productos en la máquina, ya que si el tipo de cambio tiene que variar también debe hacerlo el método que lo devuelve. La variación en una de las clases acopladas a la interfaz Cambio no se verá extendida a las demás clases.

-Principio abierto-cerrado -> El código está abierto a modificaciones, ya que se pueden añadir nuevas clases que implementen distintos tipos de cambios para la máquina expendedora, así como introducir en la máquina nuevos tipos de productos (productos con distinto nombre, actualización en el precio de un producto, …). Por lo tanto, permite la extensión del código incorporando nuevas clases si así es necesario, pero no admite la modificación del anterior para que el programa siga funcionando con normalidad.

-Principio de sustitución de Liskov -> A las subclases se les puede pasar cualquier tipo de método de su superclase para implementar uno propio modificando el hecho en la superclase.

-Principio de inversión de la dependencia -> Al no utilizar clases concretas, el cambiar una implementación de la lista de monedas no tiene efecto sobre el código que utiliza los métodos del tipo List (no se depende de la clase implementadora). A pesar de esto, auto limita a no poder usar métodos de ArrayList que no se encuentran en el tipo List.

-Principio de segregación de interfaces -> El interfaz Cambio está implementado para proporcionar a una máquina dos tipos de cambios completamente diferentes (se fija en el uso y no en el desarrollo del código). No es variable, ya que si añadiésemos nuevos métodos a esta interfaz, daría errores el código de las clases de CambioSimple y CambioDeposito que heredan de la misma (ya que hay cosas que habría que modificar y el código no estaría completo).

/EXPLICACIÓN DEL DISEÑO/:

Hemos elegido el patrón de diseño estrategia debido a que se ajusta a los métodos dados del código que debemos crear para simular el comportamiento de una máquina expendedora que tiene una lista de productos en su interior y a la que se le pasa una lista de monedas para comprar uno de estos productos.

/EJERCICIO 2/

/EXPLICACIÓN DE LOS PRINCIPIOS SOLID USADOS EN EL CÓDIGO/:

-Principio de responsabilidad única -> Aplicado a los clientes concretos (simples y detallados) cuya finalidad es observar el mercado de acciones y obtener unos datos de ellas (dependiendo del tipo de cliente). Cada uno de ellos tiene una finalidad única, ya que si alguno de sus intereses por las acciones cambia, también debe cambiar el código del método que lo implementa.

-Principio abierto-cerrado -> El código está abierto a cambios, ya que pueden aparecer nuevos observadores con unas características concretas y se podrían añadir a los observadores actuales sin modificar a estos en ningún rasgo.

Principio de sustitución de Liskov -> Las subclases extienden la superclase y comparten sus métodos modificándolos para un uso concreto. Al mismo tiempo, la clase de Acciones implementa la interfaz Observable para poder añadir o eliminar a nuevos observadores, y notificarles en caso de que algún valor cambie.

-Principio de inversión de la dependencia -> Se implementa una lista de acciones con sus datos propios (diferenciadas por su símbolo), pudiendo cambiarse la implementación de la misma a otro tipo de lista pero con la limitación de que hay métodos que no pueden ser implementados por otros tipos. También se produce inyección por medio de los setters de cada valor de una acción.

-Principio de segregación de interfaces -> El interfaz Observer está implementado tal y como se usa y no como se implementa. Por lo tanto, ni es demasiado general, ni se fija sólo en cosas muy concisas relativas al desarrollo del código. Tampoco es variable, ya que si añadiésemos nuevos métodos a la interfaz, fallaría el código de las clases que lo implementan (ya que hay cosas que habría que modificar y el código no estaría completo).

/EXPLICACIÓN DEL DISEÑO/:

Hemos elegido el patrón de diseño observador debido a que se ajusta a los métodos dados del código que debemos crear para implementar la fluctuación en bolsa de un mercado de acciones y notificar a los observadores de este mercado (en este caso, clientes simples que solo buscan el precio de cierre, o detallados, a los que le interesan todos los valores de la acción) de que un valor o valores de una acción en concreto ha variado.

Los UML (Diagramas de Clases y Dinámicos) se encuentran en formato PDF en la carpeta doc.