Tarea 2

(EN GRUPO DE 2 PERSONAS)

Enunciado:
El trabajo consiste en modificar el código del expendedor de bebidas que ha hecho en PA3P. En particular, además de bebidas, se desea vender dulces. El proyecto sigue con una entrada para monedas, un selector numérico para elegir el tipo de producto y un retorno de producto. Cada vez que se le ingresa una moneda y un número, retorne siempre un producto del tipo solicitado, si queda alguno en el depósito interno correspondiente. Debe haber un comprador que compre un producto, recupere todo el vuelto, lo consume, y responda su tipo y cuanta plata le devolvió expendedor

- Si el producto es más barato que el valor de la Moneda debe devolver la diferencia en monedas de $100 como vuelto.
- Si no hay productos, devuelve la misma Moneda como vuelto. Debe utilizar una excepción personalizada (NoHayProductoException)
- Si se quiere comprar un producto por un valor inferior al precio, solo devuelve la misma Moneda como vuelto. Debe utilizar una excepción personalizada (PagoInsuficienteException)
- Si se quiere comprar Bebidas sin dinero (moneda null) no retorna vuelto, ni producto. Debe utilizar una excepción personalizada (PagoIncorrectoException)

Fecha de entrega en Canvas: <u>20 de octubre hasta las 23:59.</u>

Entrega de un enlace de repositorio (<u>nuevo repositorio publico</u>) en GithHub:

- El readme debe contener los nombres completos de los/las estudiantes
- Diagrama UML (foto o imagen) de la aplicación en el origen del proyecto con clases, métodos, propriedades y relaciones (no es necesario especificar las cardinalidades, poner ArrayList, el Main o los getter/setter). Se puede usar software (como https://online.visual-paradigm.com/diagrams/features/uml-tool/) o a mano
- Código del programa y archivos del proyecto

Este documento contiene un recordatorio de cómo funciona el software (actualizado), una lista de cambios a realizar, y la rúbrica de evaluación.

Resumen del funcionamiento del software (similar a PA3P)

Imagina que main es "alguien" que le da a Comprador una sola moneda de algún valor y lo manda a comprar un producto de algún tipo y que luego le consume. Después ese "alguien" le pregunta a Comprador lo que consumió y cuanta plata le dieron de vuelto. Por supuesto puede ir a comprar productos que el expendedor puede tener o no, puede tratar de comprar con monedas que no alcanzan, o bien con monedas que sobrepasan el precio. Después de la compra:

- si la moneda no alcanza o no hay producto, no le dan producto y el vuelto es la misma moneda
- si trata de comprar sin moneda (null), no le devuelven nada
- o le alcanza y le dan vuelto, pero en este caso el expendedor devuelve sólo en monedas de 100.

Debes ponerte en el lugar del comprador y del expendedor para resolver el problema.

El Comprador al momento de su creación, en el constructor, debe:

- Recibir la moneda con la que comprará, un número que identifique el tipo de producto y la referencia al Expendedor en el que comprará (no necesita que sea almacenada permanente como propiedad)
- Comprar en el expendedor entregando una moneda y el entero con el número de depósito (imaginar que lo puede ver)
- Si consiguió un producto deberá consumirlo y registrar su sabor en una propiedad
- Recuperar las monedas del vuelto una a una, hasta que se acaben y sumar la cantidad en la propiedad que almacena la cantidad total
- NO almacenar las monedas como propiedad ni los productos como propiedad
- Las propiedades son un String y un entero, nada más. Los valores se devuelven cuando se lo pidan posteriormente

El Expendedor debe:

- Al momento de su creación, en su constructor recibir la cantidad única con la que debe llenar "mágicamente" todos depósitos de productos con la misma cantidad, y el precio único de todos los tipos de bebidas (múltiplo de $100, precio único por bebidas) y dulces (múltiplo de $100, precio único por dulces)
- Manejar monedas de $1000, $500 y $100 (para recibir como pago)
- Devolver un producto si se le entrega una moneda y el número del depósito en el que está el tipo de bebida deseada, si la moneda es de un valor mayor o igual al precio
- Si se le trata de comprar con una moneda null, se lanza una excepción PagoIncorrectoException
- Si el número de depósito es erróneo, no hay producto o no alcanza. Se lanza una excepción NoHayProductoException y no entrega bebida y deja la misma moneda que recibió en el depósito de monedas de vuelto.
- Si la compra es exitosa, devolver el vuelto (sólo monedas de $100), en el depósito de monedas de vuelto. El vuelto se crea "mágicamente" dejándolo en el depósito de vuelto: una a más monedas de 100
- Las monedas que recibe como pago no se almacenan en ningún deposito (se extinguen mágicamente)
- Las Monedas son polimórficas: ´Moneda1500, Moneda1000, Moneda500, Moneda100, …, múltiplos de 100.
- La(s) Moneda(s) de vuelto se rescatan una a una por cada llamada al método al método correspondiente
- Las monedas se diferencian por la dirección de memoria (la dirección que tiene this y equivale a su número se serie por lo tanto no hay que manejarlo) en la que se encuentran (puntero) y su valor. Deben entregar su valor y número de serie, al ser requerido (usando toString)
- Las monedas no tienen propiedades, no las necesitan, el valor puede ser retornando directamente el número
- Las bebidas deben ser de al menos tres tipos: CocaCola, Sprite, Fanta
- Los dulces deben ser de al menos dos tipos: Snickers, Super8
- Se debe entregar con un método main que incluya código con las pruebas que demuestren el funcionamiento: creas un Expendedor, Monedas, y Comprador e interrogas al comprador con cada producto, como pruebas las excepciones deben ser levantada y capturada también (en este caso se muestra un mensaje)

Sobre la recuperación del vuelto:

- el comprador debe llamar varias veces a getVuelto obteniendo una moneda cada vez, hasta que le retorne null. Imagina que estás sacando las monedas desde el depósito donde caen, en uno real.

Se espera que reutilices lo que hiciste antes en PA3P, pero con algunas diferencias que son el trabajo para esta tarea:

Estructura del proyecto:

- Cada clase debe tener su propio archivo (Comprador en Comprador.java, Bebida en Bebida.java…). ¡no cree un archivo principal con todo el código ¡Le recomiendo que empiece por esto!
- Debe utilizar GIT tal y como se ha visto durante el curso
- Haz commits poco a poco para las modificaciones, y distribuye el trabajo. Por ejemplo, una persona puede trabajar en los productos, mientras otra se encarga de hacer excepciones.

Producto, Dulce, Bebida:

- Necesitas cambiar tu jerarquía de clases usando herencia. Para no tener más sólo bebidas, pero también una clase **Producto** que puede utilizarse polimórficamente para **Dulce** y Bebida (y las subclases de eso).
- También tendrá que actualizar sus otras clases para poder utilizar los productos
- Todos los productos deben poder tener un precio diferente mediante una enumeración (ver más abajo)

Excepciones:

- Deben creer y usar **PagoIncorrectoException, NoHayProductoException y PagoInsuficienteException.**
- Los throws se hacen desde el método comprarProducto().
- Los try y catch se hacen en el Main (entonces el Comprador puede hacer Throws también).
- No tienen que aparecer en el UML.

Depósitos genéricos**:**

- A diferencia de en PA3P, los depósitos deben usar un tipo genérico <T> para determinar si van a almacenar un Producto o Moneda

Monedas:

- Sin cambios en los tipos de monedas, pero deben implementar la interfaz Comparable y entonces el método compareTo (ver el curso sobre interfaz).

Documentación:

- Tienen que documentar el código usando el formato Javadoc que vimos durante el curso.
- No olvide que su editor de código (IntelliJ) puede ayudarle a generar la documentación

Pruebas:

- Deben usar Junit en su proyecto debe tener la configuración correcta para la ejecución de pruebas (recomiendo usar Maven).
- Sus pruebas deben cubrir casos de uso normales (por ejemplo, la compra de varios productos disponibles con una cantidad suficiente) y verificar su correcto funcionamiento (producto recibido, retorno de vuelto correcto...)
- También debe probar los casos extremos, los casos que lanzan excepciones o potencialmente con datos incorrectos o faltantes cuando sea relevante.
- Te recomiendo que escribas tantas clases de prueba como sea posible (no sólo una), siempre que haya lógica que probar (métodos que no sean simples getters o setters o constructores) entonces probablemente sea apropiado hacer una clase de prueba dedicada (Comprador, Expendedor, Depositos y Monedas me parece un mínimo).
- Cada método de prueba debe ser breve y probar un escenario específico, no debe hacer afirmaciones sobre casos totalmente diferentes (por ejemplo, no pruebe todas las excepciones en un método de prueba).
- Sus pruebas deben ser coherentes con sus métodos y clases, no pruebe escenarios que no tengan nada que ver con el software.

Enumeraciones:

- Debe sustituir las constantes que utiliza para seleccionar e indicar el precio del producto por una enumeración.
- Esta enumeración tendrá diferentes valores constantes, pero también variables y métodos para recuperar las propiedades asociadas a estos valores.
- Con eso debería poder obtener fácilmente un precio que puede ser diferente para cada producto.

Resultados de aprendizaje evaluados:

1. Aplicar el paradigma de la orientación a objetos al desarrollo de aplicaciones software de mediana envergadura, en un entorno de trabajo en equipos.

3. Aplicar principios de diseño para el desarrollo de sistemas informáticos de calidad.

5. Aplicar técnicas de testing unitario para pruebas de verificación de la funcionalidad que implementen.

Rúbrica de evaluación Tarea 2					
Criterios	**Calificaciones**				**Pts**
Ejecución del código	**12 para >8.0 pts** **Excelente** El código se ejecuta sin problemas y el main es suficiente para dar una idea de cómo funciona el software	**8 para >4.0 pts** **Bien** El código se ejecuta sin problemas pero falta algunas partes en el main o hay errores menores	**4 para >0.0 pts** **Por mejorar** El código tiene problemas durante la ejecución o la implementación de main está incompleta	**0 pts** **Falta** El código no se compila o no se ejecuta	12 pts
Completitud y calidad de la aplicación	**15 para >10.0 pts** **Excelente** La aplicación contiene todo el código necesario y la implementación es correcta	**10 para >5.0 pts** **Bien** Faltan algunas partes menores del código o hay problemas menores de implementación	**5 para >0.0 pts** **Por mejorar** Faltan algunas partes importantes del código o hay problemas importantes con la implementación	**0 pts** **Falta** Falta gran parte del código	15 pts
Trabajo en grupo Si no se respeta el trabajo en grupo, puedo aplicar una penalización de puntos o NCR	**6 para >4.0 pts** **Excelente** El trabajo está bien repartido entre los dos miembros del equipo	**4 para >2.0 pts** **Bien** El trabajo podría estar mejor distribuido o falta colaboración en las mismas clases	**2 para >0.0 pts** **Por mejorar** El trabajo está desequilibrado o no hay colaboración	**0 pts** **Falta** El trabajo está totalmente desequilibrado	6 pts
Uso de GIT Si no se respeta el uso de git, puedo aplicar una penalización de puntos o NCR	**6 para >4.0 pts** **Excelente** GIT se ha utilizado correctamente (los tamaños, las frecuencias y los mensajes de los commits son lógicos)	**4 para >2.0 pts** **Bien** GIT fue bien utilizado (tamaño, la frecuencia y los mensajes de los commits podrían mejorarse)	**2 para >0.0 pts** **Por mejorar** El tamaño, la frecuencia y los mensajes de los commits podrían mejorarse mucho	**0 pts** **Falta** GIT ha sido poco o nada explotado	6 pts
Puntualidad (dependiendo del retraso, la tarea puede no ser evaluada)	**3 para >2.0 pts** **Excelente** Entrega el trabajo en la fecha establecida.	**2 para >0.0 pts** **Por mejorar** Entrega el trabajo fuera de plazo, pero con justificación inoportuna.	**0 pts** **Falta** Entrega el trabajo fuera del plazo y sin justificación		3 pts
Modelo UML	**12 para >8.0 pts** **Excelente** El modelo UML es correcto (contiene todas las clases, métodos, propiedades o relaciones para resolver el problema) y el código corresponde	**8 para >4.0 pts** **Bien** Falta informaciones menores en el modelo UML y/o hay errores menores	**4 para >0.0 pts** **Por mejorar** El modelo UML es bastante incompleto o no respeta el estándar UML	**0 pts** **Falta** Sin modelo UML o realmente incompleto el código no coincide con el modelo	12 pts
Documentación	**6 para >4.0 pts** **Excelente** Hay javadoc en el formato correcto para las clases y métodos principales	**4 para >0.0 pts** **Por mejorar** Falta el javadoc para las clases principales o el formato no es totalmente correcto	**0 pts** **Falta** No hay Javadoc o esta incorrecta		6 pts
Pruebas con JUnit Si no se usa JUnit, puedo aplicar una penalización de puntos o NCR	**12 para >8.0 pts** **Excelente** Los casos de prueba cubren todos los casos de uso, clases y métodos importantes del software de forma organizada y coherente. El estándar JUnit y las etiquetas se utilizan adecuadamente.	**8 para >4.0 pts** **Bien** Faltan algunas pruebas unitarias o algunas pruebas tienen problemas con el código o existen algunos problemas al utilizar Junit o etiquetas.	**4 para >0.0 pts** **Por mejorar** Faltan a pruebas importantes o son incorrectas o JUnit y las etiquetas no se utilizan correctamente.	**0 pts** **Falta** No se utiliza JUnit o las pruebas están vacías o las pruebas no se ejecutan.	12 pts
Uso de Enum Si no se usa Enum, puedo aplicar una penalización de puntos o NCR	**3 para >2.0 pts** **Excelente** El modelo UML es correcto (contiene todas las clases, métodos, propiedades o relaciones para resolver el problema) y el código corresponde	**2 para >1.0 pts** **Bien** Falta informaciones menores en el modelo UML y/o hay errores menores	**1 para >0.0 pts** **Por mejorar** El modelo UML es bastante incompleto o no respeta el estándar UML	**0 pts** **Falta** Sin modelo UML o realmente incompleto el código no coincide con el modelo	3 pts
					Puntos totales: 75