Tarea 2

(EN GRUPO DE 2 O 3 PERSONAS)

Enunciado:
El trabajo consiste en crear un expendedor de bebidas, que cuenta con una entrada para monedas, un selector numérico para elegir el tipo bebida y un retorno de bebida. Cada vez que se le ingresa una moneda y un número, retorne siempre una bebida del tipo solicitado, si queda alguna en el depósito interno correspondiente, si no, la devuelve como vuelto moneda.
Debe haber un comprador que compre una bebida, recupere todo el vuelto, beba, y responda si la bebió y cuanta plata le devolvió expendedor

- Si la Bebida es más barata que el valor de la Moneda debe devolver la diferencia en monedas de $100 como vuelto.
- Si no hay Bebidas, devuelve la misma Moneda como vuelto. Debe utilizar una excepción personalizada (NoHayBebidaException)
- Si se quiere comprar una Bebida 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 bebida. Debe utilizar una excepción personalizada (PagoIncorrectoException)

Fecha de entrega en Canvas: **27 de octubre hasta las 23:59.**

Entrega de un enlace de repositorio (nuevo repositorio) 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). Se puede usar software (como https://online.visual-paradigm.com/diagrams/features/uml-tool/) o a mano
- Código del programa y archivos del proyecto NetBeans

Detalles

Imagina que **main** es "alguien" que le da a Comprador una sola moneda de algún valor y lo manda a comprar una bebida de algún tipo y que luego la beba. Después ese "alguien" le pregunta a Comprador qué bebió y cuanta plata le dieron de vuelto. Por supuesto puede ir a comprar bebidas 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 bebida, no le dan la bebida 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 bebida 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 una bebida, deberá beberla 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 las bebidas 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 bebidas con la misma cantidad, y el precio único de todos los tipos de bebidas (múltiplo de $100)
- Manejar monedas de $1000, $500 y $100 (para recibir como pago)
- Devolver una bebida 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 bebida o no alcanza. Se lanza una excepción NoHayBebidaException 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
- 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 bebida, como pruebas las excepciones debe 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.

Lo siguiente son prototipos de clases y métodos que podemos identificar a partir de este texto (es posible utilizar más)

- prototipos:

```
  public Comprador(Moneda m, int cualBebida, Expendedor exp);
        public int cuantoVuelto();   //de vuelto
        public String queBebiste();  //el sonido de la Bebida: cocacola, sprite
  public Expendedor(int numBebidas, int precioBebidas);
        public Bebida comprarBebida(Moneda m, int cual) throws NoHayBebidaException,
PagoInsuficienteException, PagoIncorrectoException;
        public Moneda getVuelto(); //retorna moneda, null si deposito está vacío
  public Bebida(int numSerie);
        public int getSerie();
        public abstract String beber();
  public Moneda();
        public String getSerie ();  //significa que retorna su dirección en RAM como número de
serie
        public abstract int getValor(); //retorna la cantidad de $que vale la moneda
```

-subclases de Bebida: CocaCola, Sprite, Fanta
-subclases de Moneda: Moneda1000, Moneda500, Moneda100

-depósitos:

- se espera que reutilices lo que hiciste antes en PA3P
- puedes copiar el Código de la clase Deposito con otro nombre para crear una clase Deposito de vuelto, solo necesitas cambiarle el nombre y poner <Moneda> en el ArrayList

Rúbrica de evaluación 2

Criterios	Calificaciones				Pts
Adecuación del código al modelo UML	**6 para >4.0 pts** **Excelente** El código se corresponde perfectamente con el modelo UML o se justifican las diferencias existentes	**4 para >2.0 pts** **Bien** Una gran parte del código corresponde al modelo UML	**2 para >0.0 pts** **Por mejorar** Sólo algunos elementos corresponden al modelo UML	**0 pts** **Falta** El código no coincide con el modelo UML	6 pts
Ejecución del código	**6 para >4.0 pts** **Excelente** El código se ejecuta sin problemas y el main prueba todo el código como se requiere	**4 para >2.0 pts** **Bien** El código se ejecuta sin problemas pero falta algunas pruebas en el main o hay errores menores	**2 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	6 pts
Completitud y calidad de la aplicación	**6 para >4.0 pts** **Excelente** La aplicación contiene todo el código necesario y la implementación es correcta	**4 para >2.0 pts** **Bien** Faltan algunas partes menores del código o hay problemas menores de implementación	**2 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	6 pts
Trabajo en grupo	**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	**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)	**6 para >4.0 pts** **Excelente** Entrega el trabajo en la fecha establecida.	**4 para >2.0 pts** **Bien** Entrega el trabajo fuera de plazo, con justificación válida y oportuna.	**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	6 pts
Modelo UML	**6 para >4.0 pts** **Excelente** El modelo UML es correcto (contiene todas las clases, métodos, propiedades o relaciones para resolver el problema)	**4 para >2.0 pts** **Bien** Una gran parte del modelo UML está presente	**2 para >0.0 pts** **Por mejorar** Sólo algunos elementos son presente en el modelo UML	**0 pts** **Falta** El modelo UML está ausente o muy incompleto	6 pts
					Puntos totales: 42