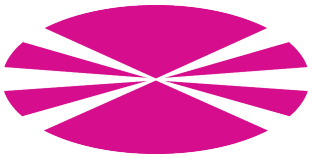

UNIVERSIDADE DA CORUÑA

Diseño Software

Práctica 2 (2023-2024)

INSTRUCCIONES:
Fecha límite de entrega: 17 de noviembre de 2023 (hasta las 23:59).

- **Estructura de los ejercicios**
  - Los ejercicios se entregarán usando *GitHub Classroom*. En concreto en el *assignment* `2324-P2` del *classroom* `GEI-DS-614G010152324-OP1`.
  - Se generará un repositorio con el nombre del *assignment* y el nombre del grupo de prácticas que será también el nombre del proyecto IntelliJ (por ejemplo `2324-P2-DS-12_jose.perez_francisco.garcia`).
  - Se creará un paquete por cada ejercicio del boletín usando los siguientes nombres: `e1`, `e2`, etc.
  - Es importante seguir las instrucciones del ejercicio, ya que persigue el objetivo de probar un aspecto determinado de Java y la orientación a objetos.
- **Tests JUnit y cobertura**
  - Cada ejercicio deberá llevar asociado uno o varios tests JUnit que permitan comprobar que su funcionamiento es correcto.
  - **Es responsabilidad vuestra desarrollar los tests y asegurarse de su calidad** (índice de cobertura alto, un mínimo de 70-80 %, probando los aspectos fundamentales del código, etc.).
  - **IMPORTANTE: La prueba es parte del ejercicio. Si no se hace o se hace de forma manifiestamente incorrecta significará que el ejercicio está incorrecto.**
- **Evaluación**
  - Este boletín corresponde a 1/3 de la nota final de prácticas.
  - Aparte de los criterios fundamentales ya enunciados en el primer boletín habrá criterios de corrección específicos que detallaremos en cada ejercicio.
  - Un criterio general en todos los ejercicios de este boletín es que **es necesario usar genericidad correctamente** en todos aquellos interfaces y clases que sean genéricos en el API. Revisad la ventana de *Problems* (`Alt+6`) de IntelliJ.
  - No seguir las normas aquí indicadas significará también una penalización.

1. **Clases abstractas**

   Nos piden implementar el sistema informático de un banco teniendo en cuenta los siguientes requisitos:

   Cuentas bancarias

   - Las cuentas bancarias deben identificarse con un código IBAN (International Bank Account Number) y deben almacenar un saldo en euros.
   - Las cuentas deberán tener métodos para consultar el saldo y para modificar el saldo ingresando o retirando dinero.
   - Existen, al menos inicialmente, dos tipos de cuentas bancarias, las cuentas corrientes y las cuentas a plazo (o de ahorro).
   - La cuenta corriente no tiene ninguna restricción en su funcionamiento, sin embargo la cuenta a plazo incluye las siguientes restricciones: penalizan cada retirada de dinero con una comisión de un 4 % y un mínimo de 3 euros, y sólo permiten introducir dinero en cantidades superiores a 1.000 euros.
   - Por norma general ninguna cuenta permite descubierto (retirar más dinero del que existe).

   Clientes

   - Los clientes se identifican básicamente por su DNI.
   - Por simplificar el problema podemos suponer que una cuenta pertenece únicamente a un solo cliente.
   - Existen tres tipos de cliente: los clientes normales o estándar, los clientes preferentes y los clientes VIP.
   - El cliente normal no tiene ningún tipo de privilegio.
   - El cliente preferente puede sacar dinero dejando las cuentas en saldos negativos, siempre y cuando la deuda no sea superior a 1.000 euros, el límite mínimo de ingreso en las cuentas a plazo se rebaja hasta 500 euros y la comisión por la retirada en las cuentas a plazo es de un 2 % con un mínimo de 1 euro.
   - Los clientes VIP pueden sacar dinero dejando las cuentas en saldos negativos, no tienen un límite mínimo de ingreso en las cuentas a plazo y tampoco se les cobra comisión de retirada en dichas cuentas.

   Criterios:

   - Encapsulación.
   - Creación de estructuras de herencia y abstracción.
   - Uso de polimorfismo y ligadura dinámica.

   De utilidad: Podéis suponer que las cantidades monetarias están en céntimos de euro para evitar trabajar con números en coma flotante. Usad de esta forma un tipo `long`. Es una forma sencilla de evitar usar clases más complejas como `BigDecimal`.

2. **Interfaces**

   Se pide implementar un módulo software para representar y gestionar personajes de un juego de ordenador con temática de combate uno contra uno. Cada personaje puede portar un conjunto de objetos de carácter defensivo (objetos de protección) y un conjunto de objetos de ataque. Se tendrán en cuenta los siguientes requisitos:

   Objetos

   - Existen cuatro tipos de objetos que los personajes pueden portar: Armor, Wand, Sword y FireBallSpell.
   - Un objeto puede tener una función defensiva (de protección), de ataque o ambas. **Armor** es un objeto de protección; **FireBallSpell** es un objeto de ataque; **Wand** y **Sword** son objetos que tienen características tanto de ataque como de defensa, y pueden utilizarse en un personaje para una u otra función.
   - **Objetos de ataque**. Tienen un valor numérico entero que representa el daño que pueden infligir (disminuir la vida del adversario en esa cantidad). En un ataque, el valor de daño se reducirá con el valor de protección del personaje que recibe el ataque. Tienen un valor entero que representa el mínimo valor de maná que debe tener un personaje para poder utilizarlo en un ataque. Además, tienen un valor numérico que indica el número de veces que el personaje puede usarlo para atacar (se decrementará en una unidad con cada ataque). Cada tipo de objeto de ataque tiene un número máximo de usos por defecto.
     - **Wand:** este objeto, cuando se usa para ataque multiplica por 2 su poder de ataque en el primer uso.
     - **FireBallSpell:** este objeto de ataque resta 1 a su poder de ataque cuando solo le queda un uso.
   - **Objetos de protección (defensa)**. Tienen asociados valores numéricos que indican la protección que proporcionan y la mínima fuerza que debe tener el personaje para poder utilizarlo en una acción defensiva.
   - Todos los objetos tienen un atributo *codename* (string) que los identifica.
   - Cada tipo de objeto podría tener más atributos propios en el futuro o diferentes comportamientos (ej. objetos de ataque con un número de usos ilimitado). En el futuro también podría interesar añadir nuevos tipos de objetos.

   Personajes

   - Los personajes tendrán un nombre identificativo, un valor de vida (entero entre 0 y 20), un valor de fuerza (entero entre 1 y 10) y un valor de maná (entero entre 0 y 10).
   - Actualmente existen las clases de personaje **Wizard** y **Warrior**, aunque podría considerarse añadir otras en el futuro. Por simplicidad podemos considerar que las distintas clases de personajes no tienen atributos propios, es decir, todos los personajes tienen los mismos atributos pero estos se pueden inicializar con valores distintos al crearlos.
   - Cada personaje puede equiparse con un conjunto de objetos defensivos (máximo 5), de modo que la protección que aporten al personaje será la suma de

la protección de todos ellos. Asimismo puede equiparse con una lista de objetos de ataque (máximo 5), utilizándose en cada ataque el primero de ellos y eliminándose cuando el número de usos restantes del objeto llegue a cero.

- Cuando a un personaje se asigne un objeto con funciones tanto defensivas como de ataque, deberá elegirse con cuál de ellas se le asigna al personaje.

Juego

- Se implementará un simulador básico del juego que proporcione dos opciones:
  - Simulación de un único ataque de un personaje a otro, de modo que el primero actúe como atacante y el segundo como defensor. El resultado será el daño efectivo del ataque, teniendo en cuenta el valor de ataque y protección de los personajes según el caso.
  - Simulación de una lucha por turnos entre dos personajes (secuencia de ataques alternativos entre ambos). La lucha por turnos finalizará cuando uno de ellos muera (su nivel de vida baje a cero) o cuando se alcance un número máximo de turnos indicado como parámetro. Al finalizar se devolverá el personaje ganador o *null* si se ha alcanzado el límite de turnos sin victoria.

Criterios:

- Resolución de herencia múltiple a través de interfaces.
- Uso de herencia y abstracción.

3. **Comparación y ordenación de artistas musicales**

   Para comparar elementos y ordenar colecciones en Java usaremos los interfaces `Comparable<T>` y `Comparator<T>` y métodos como `sort` de la clase `Collections`. A continuación incluimos una breve descripción de los mismos. La descripción completa está en la documentación del API (`https://docs.oracle.com/en/java/javase/21/docs/api/`):

   - `Comparable<T>` incluye un método `compareTo(T o)` que compara un objeto con otro usando el *orden natural* especificado en el propio objeto. Devuelve un número entero negativo, cero o un número entero positivo, si el objeto actual es menor, igual o mayor que el objeto pasado por parámetro.
   - `Comparator<T>` es similar al anterior pero incluye un método para comparar dos objetos cualesquiera `compare(T o1, T o2)`. En este caso el resultado será un número entero negativo, cero o un número entero positivo, si el el primer argumento es menor, igual o mayor que el segundo.
   - `Collections.sort` tiene dos versiones. En la primera le pasas una lista de elementos que hayan implementado el interfaz `Comparable<T>` y el método te ordena la lista por su *orden natural*. En la segunda le pasas una lista de objetos cualesquiera y un `Comparator<T>` para esos objetos y te ordena la lista usando el comparador.

   Este ejercicio continúa la temática del ejercicio 3 de la práctica 1. En primer lugar, ve a `https://musicbrainz.org/` y fíjate en las características que definen a un **artista**. Basándote en ello, crea la clase `Artist` e incluye algunos de esos atributos o atributos similares, aparte de otros atributos que se indicarán más abajo en el ejercicio.

   También deberás poder gestionar **listas de artistas**. Dichas listas se podrán **ordenar** y deberás implementar dos métodos para ello. En primer lugar, un método sin parámetros que utilice el *orden natural* de los artistas (que en este caso estará basado en un `String` llamado `id`). En segundo lugar, un método con un único parámetro de tipo `Comparator<T>` en el cual se delegará el criterio de ordenación.

   Deberás implementar sendos comparadores para:

   - La puntuación media del artista (un artista almacena una colección de puntuaciones).
   - El eclecticismo del artista (un artista almacena una colección de géneros).

   Aparte de esos dos comparadores, añade otros dos basados en otros atributos. En general, intenta que los tipos de datos sean diversos y que entre los **cuatro comparadores** haya algunos en orden creciente y otros en orden decreciente.

   Nota: En principio, no es necesario reutilizar nada del código de la práctica 1.

   Criterios:

   - Ordenación de una colección.
   - Uso de `Comparable<T>` y `Comparator<T>`.
   - Uso de colecciones de objetos y genericidad.

4. **Diseño UML**

   Down Experience[1] es un proyecto pionero en Galicia impulsado desde Down Coruña[2]. Su objetivo es la creación de empresas que demuestre que las personas con síndrome de Down y otras discapacidades mentales son capaces de emprender, contribuyendo de esta forma a su inclusión social y laboral. El programa piloto es El Quiosco de Down Experience situado en la Plaza de Orense de A Coruña, una iniciativa de restauración en la que personas con y sin discapacidad desempeñan la profesión de barista y trabajan ofreciendo cafés, bocatas de calamares y otras propuestas gastronómicas.

   El objetivo del ejercicio es implementar las clases básicas para utilizar en El Quiosco. Necesitaremos por tanto almacenar información de todas las personas implicadas en la iniciativa, para cada una de ellas es preciso conocer: nombre, apellidos, DNI, teléfono de contacto y correo electrónico. A mayores se necesita cierta información adicional en función de la categoría:

   - **Clientes**: interesa conocer su código de cliente y el número de compras realizadas. Esta última característica es necesaria para ofrecer descuentos de fidelización a los clientes habituales del local.
   - **Dependientes**: es necesario saber su número de la seguridad social, su salario y el turno asignado (mañana, tarde o alterno). Para obtener el salario se tendrá en cuenta que si trabaja en turno alterno tiene un extra en el salario. También almacenaremos su especialidad (personal de mesa, barra, cocina, etc.).
   - **Personal de apoyo**: Este puesto surge de la necesidad de refuerzo en aquellas épocas del año con mayor carga de trabajo. Debemos saber su número de la seguridad social, su salario y el turno al que pertenecen (mañana o tarde, no hay personal de apoyo con turno alterno). Es necesario almacenar información de la organización de la que proceden ya que habitualmente se favorece la colaboración con otras entidades a pro de la inserción laboral de diferentes colectivos.

   Se dispone de una clase ElQuiosco que incluye los siguientes métodos que nos permiten la gestión de clientes y trabajadores:

   - Métodos para agregar clientes o trabajadores al quiosco: `agregarCliente`/`agregarTrabajador` para agregar un cliente o un trabajador concreto, `agregarClientes`/`agregarTrabajadores` para agregar una lista de clientes o trabajadores que se pasen por parámetro.
   - Un método `salariosElQuiosco` que devuelve la suma total de los salarios de todos los trabajadores del quiosco.
   - Un método `personasElQuiosco` que devuelve una lista con todas las personas (clientes y trabajadores) involucrados en el quiosco.

   El objetivo de este ejercicio es desarrollar el modelo estático y el modelo dinámico en UML. En concreto habrá que desarrollar:

[1] `https://downexperience.com/`
[2] `https://www.downcoruna.org/`

- **Diagrama de clases UML** detallado en donde se muestren todas las clases con sus atributos, sus métodos y las relaciones presentes entre ellas. Prestad especial atención a poner correctamente los adornos de la relación de asociación (multiplicidades, navegabilidad, nombres de rol, etc.).
- **Diagrama dinámico UML**. En concreto un diagrama de secuencia que muestre el funcionamiento del método `salariosElQuiosco`.

Para entregar este ejercicio deberéis crear un paquete `e4` en el proyecto *IntelliJ* del segundo boletín y situar ahí (simplemente arrastrándolos) los diagramas correspondientes en un formato fácilmente legible (PDF, PNG, JPG, ...) con nombres fácilmente identificables.

Os recomendamos para UML usar la herramienta **MagicDraw** de la cual disponemos de una licencia de educación (en el Campus Virtual explicamos cómo conseguir la licencia). De todas formas, cualquier herramienta de dibujo que uséis es aceptable siempre que el diagrama sea legible y se entregue en el formato indicado.

Criterios:

- Creación de estructuras de herencia y abstracción.
- Uso de polimorfismo y ligadura dinámica.
- Los diagramas son completos: con todos los adornos adecuados.
- Los diagramas son correctos: siguen fielmente el estándar UML y no están a un nivel de abstracción demasiado bajo (especialmente los diagramas de secuencia).
- Los diagramas son legibles: tienen una buena organización, no están borrosos, no hay que hacer un zoom exagerado para poder leerlos, etc.