UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 2 (2020-2021)

INSTRUCCIONES:

Fecha límite de entrega: 27 de noviembre de 2020 (hasta las 23:59).

- **Estructura de los ejercicios**
  - Se deberá subir al repositorio un único proyecto para boletín cuyo nombre será el nombre del grupo de prácticas más el sufijo -B2 (p.ej. `DS-11-01-B2`).
  - Se creará un paquete por cada ejercicio con los nombres: `e1`, `e2`, etc.
  - Es importante que sigáis detalladamente 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.
  - Al contrario que en el primer boletín **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** en todos aquellos interfaces y clases que sean genéricos en el API. Por lo tanto obtener el *warning* `unchecked` significará que no se está usando correctamente la genericidad y supondrá una penalización en la nota.
  - No seguir las normas aquí indicadas significará también una penalización en la nota.

1. **El Señor de los Anillos**

   Realizar varias clases en Java que permitan la implementación de un sencillo juego basado en la historia fantástica del libro del Señor de los Anillos. En el juego existirán diversos personajes clasificados en dos categorías: **Héroes** y **Bestias**. En la parte de los héroes podrán crearse varios tipos de personajes: **Elfos**, **Hobbits** y **Humanos** mientras que en la parte de las bestias existirán **Orcos** y **Trasgos**.

   El **objetivo del juego** es la creación de dos ejércitos de personajes, uno de héroes y otro de bestias, que se enfrentarán entre ellos, mediante un sistema de turnos, hasta que uno de ellos logre la victoria. Para ello, cada personaje estará caracterizado por un nombre, unos puntos de vida (número entero) y un nivel de resistencia de su armadura (número entero). En cada turno un personaje podrá atacar a un adversario con las siguientes peculiaridades:

   - **Héroes**: su ataque estará basado en la tirada de dos dados con valores entre `0` y `100` de los cuales se elegirá el valor mayor. Este valor determinará la potencia ofensiva de su ataque en el turno actual.
   - **Bestias**: la potencia de su ataque en cada turno estará basado en la tirada de un único dado con valores entre 0 y 90.

   Para hacer el **funcionamiento de los dados** más dinámico será el juego el encargado de pasarle a los héroes y a las bestias el dado concreto que tendrán que usar en la partida. De esa forma podemos pasarles dados con distintos valores máximos y buscar un desarrollo del juego más equilibrado. También deberemos ser capaces de usar una versión especial de los dados, que serán los dados trucados, que nos permitan pasarle una *semilla* al algoritmo de pseudoaleatoriedad de tal forma que la secuencia obtenida por el dado es siempre la misma. Eso facilita, por ejemplo, el desarrollo de pruebas al eliminar el elemento aleatorio de las mismas.

   Una vez calculada su potencia ofensiva, se **calculará el daño infligido al adversario** en función de su nivel de armadura. Sólo se producirá un daño al adversario si la potencia ofensiva del atacante es superior al nivel de armadura del defensor. En ese caso, el daño producido será la diferencia entre la potencia de ataque y el nivel de armadura del oponente. Además, en cada ataque habrá que tener en cuenta los siguientes supuestos particulares de cada tipo de personaje:

   - **Elfos**: estos personajes odian especialmente a los Orcos por lo que tendrán un nivel de rabia superior en su ataque únicamente contra este tipo de bestias, lo que incrementará su potencia ofensiva, calculada en la tirada de dados, en 10 unidades.
   - **Hobbits**: estos personajes tienen un miedo especial a los Trasgos por lo que siempre que se enfrenten a uno de ellos perderán 5 unidades en su potencia ofensiva.
   - **Orcos**: estos personajes poseen una fuerza desmesurada por lo que siempre que realicen un ataque el nivel de armadura que posee su oponente se verá reducido en un 10 % (el nivel de armadura no se reduce de forma permanente sino sólo para el turno actual).

Una vez descrito el proceso de lucha individual entre personajes, a continuación se describirá **el proceso de batalla entre ambos ejércitos**. Para ello, se seguirá un sistema basado en turnos en el que en cada turno un personaje de un ejército atacará a un único adversario del ejército oponente. Para simplificar el sistema de batalla, se enfrentarán siempre los personajes situados en la misma posición de cada ejército. Si alguno de los ejércitos dispone de más efectivos que el contrario los personajes sobrantes no participarán en ese turno de batalla.

En cada turno se producirán todos los enfrentamientos y se disminuirá la vida de cada personaje siguiendo las instrucciones mencionadas anteriormente. En el momento en que un personaje llegue a un nivel de vida igual o inferior a cero se producirá su muerte por lo que se eliminará de su posición y se desplazarán todos sus compañeros en posiciones posteriores para cubrir la baja. De esa forma alguno de los personajes inactivos podrá participar en la batalla en los siguientes turnos.

La clase que representa al juego deberá tener un método `batalla` que ponga a luchar a dos ejércitos con los dados que decida el propio juego y cuyo resultado sea una lista de `String` con el desarrollo de la partida. A continuación se presenta un posible contenido de dicha lista de `String`:

```
Turn 1:
  Fight between Legolas (Energy=150) and Lurtz (Energy=190)
  Fight between Gandalf (Energy=50) and Mauhur (Energy=290)
Turn 2:
  Fight between Legolas (Energy=115) and Lurtz (Energy=140)
  Fight between Gandalf (Energy=50) and Mauhur (Energy=272)
Turn 3:
  Fight between Legolas (Energy=115) and Lurtz (Energy=140)
  Fight between Gandalf (Energy=50) and Mauhur (Energy=215)
Turn 4:
  Fight between Legolas (Energy=64) and Lurtz (Energy=109)
  Fight between Gandalf (Energy=50) and Mauhur (Energy=171)
Turn 5:
  Fight between Legolas (Energy=26) and Lurtz (Energy=63)
  Elf Legolas dies!
  Fight between Frodo (Energy=20) and Mauhur (Energy=171)
  Hobbit Frodo dies!
Turn 6:
  Fight between Gandalf (Energy=50) and Lurtz (Energy=46)
Turn 7:
  Fight between Gandalf (Energy=50) and Lurtz (Energy=8)
  Orc Lurtz dies!
Turn 8:
  Fight between Gandalf (Energy=50) and Mauhur (Energy=150)
Turn 9:
  Fight between Gandalf (Energy=42) and Mauhur (Energy=103)
Turn 10:
  Fight between Gandalf (Energy=42) and Mauhur (Energy=43)
  Goblin Mauhur dies!
HEROES WIN!!
```

Criterios:

- Encapsulación
- Creación de estructuras de herencia y abstracción.
- Uso de polimorfismo y ligadura dinámica.
- Uso de genericidad.
- Uso de la clase `Random`.

2. **Suma de Matrices**

   Crea una clase `Matrix` que representa el funcionamiento básico de una matriz de enteros. La matriz podrá crearse usando los siguientes constructores:

   - Indicando el número de filas y de columnas (el contenido de las celdas de la matriz se inicializará a cero)
   - Copiando los valores de un array bidimensional que se le pase por parámetro (debe ser un array rectangular, no *ragged*, sino lanzará una excepción avisando del error).

   Una vez creado el objeto matriz se podrán llevar a cabo las siguientes operaciones:

   - Leer el número de filas y columnas de la matriz.
   - Leer y modificar los valores concretos de una celda de la matriz dada una fila o una columna (lanzando excepciones si el valor pasado como fila o columna no es válido).
   - Devolver una copia de la matriz como un array bidimensional.
   - Devolver una copia de una fila o una columna de la matriz como un array unidimensional (lanzando excepciones si el valor pasado como fila o columna no es válido).
   - Devolver una versión en String de la matriz, representada de la siguiente forma:

     ```
     [1, 2, 3, 4]
     [5, 6, 7, 8]
     ```

   Para poder recorrer la matriz vamos a implementar dos iteradores con las siguientes características:

   - El primer iterador recorre la matriz por filas y luego por columnas y el segundo recorre por columnas y luego por filas.
   - Estos iteradores se crearán en clases aparte (pueden ser internas si sabéis hacerlas) que deben implementar el interfaz `Iterator` (no es necesario implementar el método `remove` y se debe lanzar una excepción si se intenta usar).
   - La matriz debe crear dos métodos para realizar la iteración (`rowColumnIterator()` e `ColumnRowIterator()`) que devuelvan los iteradores que hemos creado para que sean accesibles a otras clases.
   - La matriz debe implementar el interfaz `Iterable` y se podrá configurar (por ejemplo a través de un atributo *booleano* o un enumerado que almacene la matriz) cuál de los dos iteradores se devolverá al llamar al método `iterator()`.

   Finalmente, crea una clase `MatrixAddition` que tenga un método que, dadas dos matrices, las suma recorriéndolas usando los iteradores antes mencionados, y devuelva un nuevo objeto `Matrix` que es la suma de las dos pasadas por parámetro (si las matrices no tienen las mismas dimensiones debe lanzar un `ArithmeticException`).

   Criterios:

   - Usar la clase `Arrays` y sus métodos para simplificar las operaciones a realizar.
   - Ver las firmas de los métodos abstractos de `Iterator` e `Iterable` para implementarlos correctamente.

3. **El Juego del Pistolero**.

   El juego del pistolero es un juego infantil, del estilo del *piedra-papel-tijera* pero más sofisticado. Consiste en dos jugadores que se enfrentan entre sí encarnando a un pistolero. Los jugadores deletrean las sílabas `Pis-to-le-ro` y acto seguido realizan una acción marcándola con las manos. Las posibles acciones del pistolero son:

   - **Disparar (*shoot*)**: Dispara al rival. Sólo se puede disparar si se tienen balas en el revólver. El gesto es poner las dos manos en paralelo con el símbolo de la pistola (pulgar hacia arriba e índice hacia adelante).
   - **Recargar (*reload*)**: Recarga el revólver con una bala. Al inicio el revólver está vacío. El gesto es poner las dos manos con los pulgares hacia atrás.
   - **Proteger (*protect*)**: Nos protegemos del disparo. Si nos disparan y estamos protegidos nada nos ocurre. El gesto es cruzar las manos sobre el pecho.
   - **Ametrallar (*machine-gun*)**: Si conseguimos cargar nuestro revólver con 5 balas entonces podremos ametrallar al contrario. Ametrallando al contrario venceremos incluso aunque éste se proteja o nos dispare a la vez. El gesto es poner las dos manos con el símbolo de la pistola una delante de la otra.

   El objetivo del juego es ganar la partida disparando al contrario cuando está recargando o ametrallándolo. La partida puede terminar en tablas si dos jugadores se disparan a la vez o se ametrallan a la vez. También se pueden considerar tablas si después de un número determinado de rondas (por ejemplo, 50) ninguno de los jugadores ha conseguido ganar.

   El objetivo es programar el duelo entre pistoleros con las siguientes clases:

   - `Gunfight`: Clase que lleva a cabo el tiroteo entre los pistoleros y que representa el funcionamiento del juego. Incluirá un método `duel(Gunslinger g1, Gunslinger g2)` que ejecutará el duelo entre pistoleros de forma no interactiva (mostrando su discurrir por consola) y determinará quién es el ganador del mismo o si éste terminó en tablas.
   - `Gunslinger`: Representa a un pistolero que tendrá los siguientes métodos:
     - `public GunslingerAction action()`. Llamado por `Gunfight`. El pistolero decide qué acción realizar y se la comunica a `Gunfight`.
     - `public int getLoads()`. Devuelve el número de cargas del pistolero.
     - `public void rivalAction(GunslingerAction action)`. Usado por la clase `Gunfight` para comunicar al pistolero la última acción de su rival para registrarla y tenerla en cuenta en su estrategia si lo considera necesario.
     - `public List<GunslingerAction> getRivalActions()`. Devuelve la lista de acciones del rival que ha registrado el pistolero. Las últimas acciones estarán al final de la lista. Permite tratar de adivinar la estrategia del rival.
     - `public int getRivalLoads()`. Devuelve las cargas del rival.
     - `public void setBehavior(Behavior behavior)`. Establece cómo será el comportamiento del pistolero.
   - `GunslingerAction`. Un enumerado con las posibles acciones que puede realizar un pistolero: `SHOOT`, `RELOAD`, `PROTECT` y `MACHINE_GUN`.

- `Behavior`. Interfaz que define el comportamiento de un pistolero. Tendrá un único método `public GunslingerAction action(Gunslinger g)` y dada la información que obtenga del pistolero dado determinará qué acción debe realizar. La idea es que el pistolero (`Gunslinger`) delega en un objeto que implementa `Behavior` la decisión de qué acción tomar en un momento dado.

Un discurrir típico de una partida sería el siguiente:

```
Round 1-------------------------
Gunslinger 1: RELOAD
Gunslinger 2: RELOAD
The duel continues...
Round 2-------------------------
Gunslinger 1: SHOOT
Gunslinger 2: PROTECT
The duel continues...
Round 3-------------------------
Gunslinger 1: RELOAD
Gunslinger 2: RELOAD
The duel continues...
Round 4-------------------------
Gunslinger 1: SHOOT
Gunslinger 2: PROTECT
The duel continues...
Round 5-------------------------
Gunslinger 1: RELOAD
Gunslinger 2: SHOOT

The duel has ended

Winner: GUNSLINGER2
```

Finalmente, un objetivo adicional de la práctica (y que supondrá una calificación extra en el boletín) consistirá en conseguir implementar el mejor comportamiento de un pistolero, tanto dentro de vuestro clase de prácticas como de la asignatura en general. Para ello deberéis crear una clase que implemente `Behavior` que tenga como nombre el nombre de vuestro grupo (y que esté situada en el paquete `e3.behaviors`). Posteriormente haremos un enfrentamiento entre todos los pistoleros y los que obtengan mejor puntuación se llevarán el premio. Explicaremos los detalles concretos en un documento aparte.

Criterios:

- Herencia, Polimorfismo y Ligadura dinámica.
- Uso de interfaces y clases implementadoras.
- Para no complicar demasiado el ejercicio no es necesario que probéis el método `duel` de `Gunfight`. Su prueba consistirá simplemente en ejecutar el duelo con dos pistoleros que hayáis creado vosotros y mostrar el discurrir del mismo por la consola. Sí habrá que probar el resto de clases.

4. **Diseño UML**

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

   - **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 `batalla`.

   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 Moodle explicamos cómo conseguir la licencia).

   Criterios:

   - **Los diagramas son completos**: con todos los adornos adecuados.
   - **Los diagramas son correctos**: se corresponden con el código desarrollado pero no están a un nivel 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.