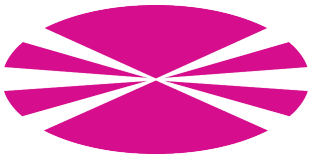

UNIVERSIDADE DA CORUÑA

Diseño Software

Práctica 2 (2024-2025)

INSTRUCCIONES:
Fecha límite de entrega: 15 de noviembre de 2024 (hasta las 23:59).

- **Estructura de los ejercicios**
  - Los ejercicios se entregarán usando *GitHub Classroom*. En concreto en el *assignment* `2425-P2` del *classroom* `GEI-DS-614G010152425-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 `2425-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 un 40 % 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. **Travian**

   Realizar varias clases en Java que permitan la implementación de un sencillo simulador de batallas basado en el juego de navegador online Travian. En el simulador existirán diversos aldeas clasificadas en tres categorías: **Romanos**, **Galos** y **Teutones**. En la parte de los Romanos podrán crearse varios tipos de tropas: **Legionarios**, **Pretorianos** y **Equites Imperatoris**, por parte de los Galos podrán crearse también varios tipos de tropas: **Druidas**, **Falanges** y **Rayos de Teutates**, mientras que por parte de los Teutones existirán **Guerreros de hacha**, **Gerreros de porra** y **Paladines**.

   El **objetivo del simulador** es la creación de dos aldeas con ejércitos, y hacer que se enfrenten entre ellos, uno como atacante y otro como defensor para saber qué aldea saldría vistoriosa de ese enfrentamiento. Para ello, cada aldea estará caracterizada por un nombre, un número de años (número entero) y un nivel de resistencia de su muralla (número entero). En cada simulación una aldea atacará y defenderá con las siguientes peculiaridades:

   - **Romanos**: Cuando ataca los puntos de ataque de sus tropas se ven aumentados en un 10 % por disponer de las mejores armas. Cuando defiende los puntos de defensa de sus tropas se automentan en 2 puntos por cada nivel de resistenca de su muralla de piedra.
   - **Galos**: Cuando ataca los puntos de ataque de sus tropas se ven aumentados en un 20 % por disponer de las pociones ideadas por Panoramix. Cuando defiende los puntos de defensa de sus tropas se automentan en 1.5 puntos por cada nivel de resistenca de su muralla de madera.
   - **Teutones**: Cuando ataca los puntos de ataque de sus tropas se ven reducidos en un 5 % ya que sus tropas siempre se presentan ebrias a la batalla. Cuando defiende los puntos de defensa de sus tropas se automentan en 1 punto por cada nivel de resistenca de su muralla de tierra.

   Una vez calculada su potencia ofensiva y defensiva, se calculará el ganador del enfrentamiento por ser el que más puntos tiene. Se utilizarán los puntos ofensivos por parte de la aldea atacante y los puntos defensivos por parte de la aldea defensora.

   Las tropas disponibles para el ejercito de cada aldea tienen cantidades de puntos de ataque y defensa diferentes y son las siguientes:

   Romanos:

   - **Legionario**:Dispone de 40 puntos de ataque y 35 de puntos defensa.
   - **Petroriano**: Dispone de 30 puntos de ataque y 65 de puntos defensa. Esta tropa puede equiparse una armadura, en el caso de tener la armadura equipada la defensa de esta unidad se ve incrementada en un 10 %.
   - **Equites Imperatoris**: Dispone de 120 puntos de ataque y 65 de puntos defensa.

   Galos:

   - **Druida**:Dispone de 45 puntos de ataque y 115 de puntos defensa.

- **Rayo de Teutates**: Dispone de 100 puntos de ataque y 25 de puntos defensa. Esta tropa puede equiparse una armadura pesada, de tenerla equipada la defensa de esta unidad se ve incrementada un 25 %, sin embargo al reducirse su movilidad, el ataque de esta unidad se ve decrementado en un 25 %.
- **Falange**: Dispone de 15 puntos de ataque y 40 de puntos defensa.

Teutones:

- **Guerrero de hacha**:Dispone de 60 puntos de ataque y 30 de puntos defensa.
- **Guerrero de maza**: Dispone de 40 puntos de ataque y 20 de puntos defensa. Esta tropa tiene la posibilidad de mejorar su maza a una de metal, en el caso de tener el arma mejorada el ataque de esta unidad se incrementa en un 25 %.
- **Paladín**: Dispone de 55 puntos de ataque y 100 de puntos defensa.

Cada unidad debe redefinir el método `toString` utilizando sobrescritura de refinamiento. Se debe utilizar genericidad para almacenar las unidades en las aldeas.

En cada simulación se producirán todos los cálculos pero no se modificarán los estados de las aldeas para poder seguir haciendo simulaciones posteriores con el mismo estado.

La clase que representa al simulador deberá tener un método `batalla` que ponga a luchar a dos aldeas con con sus respectivos ejércitos y cuyo resultado sea una lista de `String` con el resultado de la partida. A continuación se presenta el contenido que debe tener dicha lista de `String`:

```
### Starts the battle! --> villageA Attacks villageB! ###
villageA have the following soldiery:
Gauls Soldiery - Druidrider
Gauls Soldiery - Theutates Thunder
Gauls Soldiery - Theutates Thunder with heavy armor
Gauls Soldiery - Phalanx
Total villageA attack power: 282

villageB have the following soldiery:
Teutons Soldiery - Axeman
Teutons Soldiery - Maceman
Teutons Soldiery - Maceman with iron mace
Teutons Soldiery - Paladin
Total villageB defense power: 180

villageA with an age of 500 years WINS!
```

Se debe escribir el código para que en un futuro se puedan añadir nuevas tropas y tipos de aldea sin necesidad de rehacer todo el código.

Criterios:

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

2. **Colección de monedas e interfaces `Comparable<T>` y `Comparator<T>`**

   Para comparar elementos y ordenar colecciones Java usa 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 [1]. A la hora de analizar un interfaz prestad especial atención a los métodos abstractos ya que son los que, obligatoriamente, hay que implementar en el interfaz.

   - `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.

   Dada la colección de monedas creada en el boletín anterior (cuyas clases deberéis copiar a este boletín), usaremos los interfaces y métodos antes citados para conseguir el siguiente comportamiento en nuestra colección de monedas:

   - El *orden natural* de las monedas de Euro consiste en ordenarlas primero por valor (las de más valor primero), luego por países (por orden alfabético incremental de su nombre), y finalmente por diseño (por orden alfabético incremental de su descripción).
   - La colección de monedas permitirá ordenar las monedas por su *orden natural*.
   - La lista permitirá ordenar las monedas usando cualquier comparador de monedas que se le pase por parámetro. En este ejercicio realizaremos el siguiente comparador:
     - Un comparador que organiza las monedas por países (por orden alfabético incremental de su nombre), luego por valor (monedas más grandes primero) y finalmente por año (años anteriores primero).

   En la clase `String` tenemos el método `int compareTo(String str)` y el método `int compareToIgnoreCase(String str)` que nos facilitarán realizar comparaciones alfabéticas de *strings*.

   Criterios:

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

[1] `https://docs.oracle.com/en/java/javase/21/docs/api/index.html`

- Uso de colecciones de objetos y genericidad.
- No deberá usarse código funcional (p.ej. expresiones lambda, *streams*, etc.) para enfatizar el desarrollo orientado a objetos.

3. **Colección de monedas e interfaces `Iterable<T>` e `Iterator<T>`**

   Para iterar sobre colecciones de objetos Java define los interfaces `Iterable<T>` e `Iterator<T>`. `Iterable<T>` lo implementa una colección para indicar que es posible obtener un iterador sobre la misma y tiene los siguientes métodos que implementar:

   - `Iterator<T>iterator()` devuelve un iterador sobre elementos de tipo `T`.

   El interfaz `Iterator<T>` tiene los siguientes métodos:

   - `boolean hasNext()` devuelve `true` si la iteración tiene elementos que recorrer.
   - `E next()` devuelve el siguiente elemento (`E`) a recorrer en la iteración. Si no hay más elementos que recorrer lanza `NoSuchElementException`.
   - `void remove()` elimina de la colección el último elemento devuelto usando `next()`. Solo puede llamarse una vez por cada ejecución de `next()`. En caso de que nunca se haya llamado a `next()` o de que se intente llamar dos veces a `remove()` sin haber llamado a `next()` el método lanzará la excepción `IllegalStateException`.

   En este ejercicios vamos hacer que la colección de monedas sea una colección iterable pero con una peculiaridad. A la colección de monedas le indicaremos cuál es el país que vamos a usar para iterar, y de esa forma la colección nos permitirá obtener un iterador que devuelva solo las monedas que pertenezcan a dicho país. Así, si en nuestra colección tenemos las siguientes monedas: {`EURO2-ES, EURO1-ES, CENT20-FR, EURO1-ES, EURO1-IT, CENT50-IT`}, la iteración sobre las monedas españolas devolverá: {`EURO2-ES, EURO1-ES, EURO1-ES`}

   Los **aspectos a tener en cuenta** al desarrollar la iteración serán:

   - Deberéis hacer uso de las clases `Iterable<T>` e `Iterator<T>`.
   - Si el país que se le indica para hacer la iteración es `null` el iterador recorrerá todas las monedas de la lista.
   - La iteración será *fail-fast*, cualquier modificación realizada sobre los elementos de la lista durante la iteración (y desde fuera del iterador) hará que el iterador lance la excepción `ConcurrentModificationException`.
   - La iteración permitirá, sin embargo, realizar la operación `remove()` que eliminará la última moneda devuelta por `next()` de la colección.
   - Es posible apoyarse en otros iteradores definidos en el API de Java, siempre y cuando la implementación de la estrategia *fail-fast* y el lanzamiento de excepciones (como son `NoSuchElementException`, `IllegalStateException` o `ConcurrentModificationException`) se implementen en vuestro iterador.
   - La iteración tiene que hacerse sobre los elementos de la colección sin modificarlos. No es adecuado reordenar la colección para hacer la iteracción, ni copiarla sobre otro tipo de colección para poder recorrerla.

Criterios:

- Iteración de colecciones de forma independiente a su implementación.
- Uso de los interfaces `Iterable<T>` e `Iterator<T>` cumpliendo las especificaciones.
- Uso de genericidad.
- No deberá usarse código funcional (p.ej. expresiones lambda, *streams*, etc.) para enfatizar el desarrollo orientado a objetos.

4. **Diseño UML** La carrera ENKI es una prueba lúdica, no competitiva, apta para todas las edades y condiciones que, a través del juego, promueve la inclusión y la visibilización de colectivos y personas con diversidad funcional, así como el respeto y la educación en la diversidad. La gran cantidad de participantes que congrega la iniciativa hace que su organización se complique cada año, necesitando de personas anónimas y también profesionales que ceden su tiempo para colaborar.

   El objetivo del ejercicio es implementar las clases básicas para nuestra carrera inclusiva. Necesitaremos por 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:

   - Participantes: interesa almacenar su género (enumerado), su código de inscripción (String), categoría (junior, senior) y la modalidad de la misma (carrera, juegos, actividades libres).
   - Personal de asistencia: personas anónimas que a título individual se ofrecen a colaborar para que la actividad se desenvuelva con normalidad. Es necesario conocer su turno (mañana, tarde, día completo) y el puesto en el que van a desempeñar su función (recepción, logística, etc.)
   - Profesionales: Por las necesidades específicas de algunas personas inscritas es necesario contar con profesionales de diferentes tipos (terapeutas, médicos, etc.). Debemos conocer su turno (mañana, tarde, día completo) y también la organización/institución profesional de procedencia.

   Todo el equipo de voluntariado recibirá una bonificación que pueden canjear por diferentes servicios ofertados en las empresas patrocinadoras del evento. La bonificación será diferente en función de si se trata de personal de asistencia o de un profesional.

   De este modo, se dispone de una clase `EnkiRace` que incluye los siguientes métodos que nos permiten la gestión de la misma:

   - Métodos para realizar inscripciones de participantes y altas en el equipo de voluntariado:
     - `addParticipant`/`addVolunteer` para incluir una persona concreta como participante o como personal voluntario,
     - `addParticipantList`/`addVolunteerList` para incluir una lista de personas que se pasen por parámetro al conjunto de participantes o al equipo de voluntariado.
   - Un método `volunteerRace` que devuelve una lista con todo del personal voluntario que participe en la iniciativa.
   - Un método `genderParticipants` que devuelva el número de participantes acorde al género inscritos en las diferentes actividades.

   El objetivo de este ejercicio es desarrollar el modelo estático y el modelo dinámico en UML. 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 `genderParticipants`.

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.