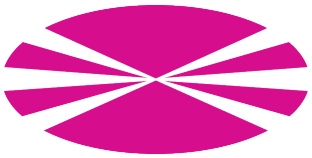

UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 2 (2022-2023)

INSTRUCCIONES:
Fecha límite de entrega: 11 de noviembre de 2022 (hasta las 23:50).

- **Estructura de los ejercicios**
  - Se entregarán en vuestro repositorio Git en un proyecto de IntelliJ IDEA cuyo nombre sea vuestro grupo con el sufijo `-B2`. Por ejemplo: `DS-11-01-B2`.
  - Se creará un paquete por cada ejercicio del boletín usando los siguientes 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 correctamente** en todos aquellos interfaces y clases que sean genéricos en el API.
  - Revisad la ventana de *Problems* (`Alt+6`) de IntelliJ en busca de avisos como "*Raw use of parametrized class...*" o "*Unchecked call to...*" que indican un mal uso de la genericidad.
  - No seguir las normas aquí indicadas significará también una penalización.

1. **Equipo humano de una producción cinematográfica**

   Dentro de una producción cinematográfica, el **equipo humano** que la compone se identifica con un nombre, apellido, DNI, teléfono y nacionalidad y suele estar dividido en las siguientes categorías:

   - **Equipo técnico**: en esta categoría se incluyen a los cargos más importantes de la producción de una película: **guionistas**, **músicos**, **productores** y **directores**.
   - **Equipo artístico**: dentro de esta categoría tenemos a **intérpretes**, **especialistas** y **dobladores**. De los intérpretes nos interesa conocer su rol dentro de la película: bien sea principal, secundario o extra.

   Cada miembro del equipo de una película recibe un salario, que se calcula, de forma simplificada, como el número de horas de trabajo por el importe/hora de su categoría. Los datos de cada categoría del equipo humano son los siguientes:

Categoría	Importe/hora (€)
Guionistas	70
Músicos	60
Productores	90
Directores	100
Intérpretes	200
Especialistas	40
Dobladores	20

   A los intérpretes se les añade un complemento salarial en función de si desempeñan el papel protagonista. En concreto, se les triplica el sueldo. Asimismo, a los especialistas se les agrega un extra de 1000€ si han rodado escenas muy peligrosas. Por parte del equipo técnico, los guionistas cobrarán un suplemento de 4000€ si el guion es original. En el caso de los directores, se tendrán en cuenta además los años de experiencia dentro de la industria cinematográfica, añadiendo al sueldo base un complemento de 1000€ por año.

   Por otra parte, los miembros del equipo técnico cobran además derechos de autor que se reparten de la siguiente forma: 5 % guionistas, 5 % directores, 4 % músicos y 2 % productores. Todo a través del registro de la propiedad intelectual. Los derechos de autor se calculan sobre la recaudación de la película en taquilla.

   Finalmente, existirá una clase **Película** que se identificará con un título y una recaudación en taquilla (en euros), y tendrá los métodos que se detallan a continuación:

   - Un método `printSalaries` que devolverá un `String` con un listado con el nombre, apellido, la categoría y el salario de cada miembro del equipo de la película y, al final, la suma total de los salarios. Por ejemplo, un resultado de llamar a `printSalaries` para la película "Alcarrás" sería:

```
Josep Abad (Stunt performer with extra for danger): 5000.0 euro
Ainet Jounou (Actor protagonist): 54000.0 euro
Xenia Roset (Actor secondary): 10000.0 euro
Cristina Puig (Dubber): 400.0 euro
Carla Simon (Director, 7 years of experience): 57000.0 euro
Maria Zamora (Producer): 9000.0 euro
Andrea Koch (Musician): 12000.0 euro
Arnau Vilaro (Screenwriter, original screenplay): 25000.0 euro
The total payroll for Alcarras is 172400.0 euro
```

Donde Josep Abad es un especialista que ha trabajado 100 horas en la película Alcarrás, incluyendo el rodaje de escenas muy peligrosas. Su sueldo base sería de 100 horas $\times$ 40€/hora = 4000€, a lo que se le suma un complemento adicional de 1000€ por peligrosidad, lo que deja su salario total en 5000€.

- Un método `printRoyalties` que devolverá un `String` con un listado con el nombre, apellido, la categoría y la cantidad que recibe cada miembro del equipo técnico por los derechos de autor de la película. Por ejemplo, un resultado de llamar a `printRoyalties` sería:

```
Carla Simon (Director): 15000.0 euro
Maria Zamora (Producer): 6000.0 euro
Andrea Koch (Musician): 12000.0 euro
Arnau Vilaro (Screenwriter): 15000.0 euro
```

Siendo de 300,000€ el dinero recaudado en taquilla por la película Alcarrás.

Criterios:

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

2. **Recorridos de listas e iteradores**

En un concurso de telerrealidad solo queda una plaza disponible para que entre el último concursante pero quedan muchos candidatos posibles (n) que han sacado la misma calificación en las pruebas clasificatorias.

Para desempatar los productores han decidido alinear a todos los concursantes, asignarles un número consecutivo del 1 a n y luego ir recorriendo la lista eliminando concursantes siguiendo el siguiente proceso:

- Se saca un número entero aleatorio k entre 1 y n.
- Se cuentan k posiciones empezando por la primera de la lista (que será la posición 1).
- Al llegar al elemento k este se borra de la lista quedando descalificado.
- Se repite el proceso comenzando por el siguiente elemento en orden después del elemento k, que será la primera posición de la nueva cuenta.
- El proceso finaliza cuando solo queda un concursante en la lista, que será el elegido.

Para evitar suspicacias y que los concursantes o la organización puedan calcular de antemano la posición de la lista ganadora existen dos posibles formas de recorrer la lista, que se explican a continuación y que se decidirá en el último momento:

- **Recorrido con rebote**: al llegar al final de la lista (contando hacia adelante, se volverá hacia el principio contando hacia atrás). Así el recorrido de una lista con cinco elementos sería `1-2-3-4-5-4-3-2-1-2-3`, etc.
- **Recorrido circular**: al llegar la final de la lista se empieza de nuevo por el principio. Así el recorrido de la lista sería ahora `1-2-3-4-5-1-2-3-4-5-1`, etc.

Veamos dos ejemplos de funcionamiento del algoritmo con ambos recorridos:

En un **recorrido con rebote**, con n=5 y k=3 sería:

`1-2-(3)-4-5`	Empezamos a contar en el 1 (inclusive) y llegamos hasta el 3 que borraremos.
`1-2-(4)-5`	Empezamos en 4 (siguiente al 3) contamos 3 posiciones (incluyendo al 4), al rebotar acabamos otra vez en el 4 que será el elemento a borrar.
`1-(2)-5`	Empezamos en el 2 (que es el siguiente al 4 yendo hacia la izquierda), de nuevo contamos tres posiciones y con el rebote acabamos otra vez en el 2, que borraremos.
`1-(5)`	Empezamos en el 5 (siguiente al dos hacia la derecha), rebotamos dos veces y volvemos al 5 que eliminaremos.
`1`	Solo queda un elemento en la lista, el 1, que será el ganador.

En un recorrido circular, con n=5 y k=3 sería:

`1-2-(3)-4-5`	Empezamos a contar en el 1 (inclusive) y llegamos hasta el 3 que borraremos.
`(1)-2-4-5`	Empezamos en 4 (siguiente al 3) contamos 3 posiciones (incluyendo al 4), al ser circular el siguiente al 5 es el 1, que será el tercer elemento y por tanto el que borraremos.
`2-4-(5)`	Empezamos en el 2 (que es el siguiente al 1, en este caso siempre iremos hacia la derecha), de nuevo contamos tres posiciones y acabamos en el 5, que borraremos.
`(2)-4`	Empezamos de nuevo en el 2 y contando 3 volveremos a caer en el 2 que eliminaremos.
`4`	Solo queda un elemento en la lista, el 4, que será el ganador.

Lo que se pide es:

- Crear una clase `TVRealityList` que mantenga una lista de `String` que será la lista con los nombres de los candidatos a entrar en el concurso.
- Esta clase debe implementar el interfaz `Iterable` y devolver, cuando se le solicite, un objeto de tipo `Iterator` que la recorra. Debe ser posible seleccionar de alguna forma que queremos que devuelva un iterador con rebote o un iterador circular.
- Crear dos clases que implementen `Iterator` que representen los dos recorridos, con rebote y circular.
- Crear en otra clase un método `selectCandidates` que dado una `TVRealityList` (a la que habremos indicado que recorrido hacer) y un salto `k` devuelva un `String` con el nombre del candidato ganador.

A continuación se muestra un resumen de los métodos de los interfaces `Iterable` e `Iterator`, la especificación completa la tenéis en el API de Java `https://docs.oracle.com/en/java/javase/18/docs/api/index.html`.

El **interfaz** `Iterable<T>` tiene un único método abstracto:

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

El **interfaz** `Iterator<T>` tiene los siguientes métodos que tendremos que implementar:

- `boolean hasNext()` devuelve `true` si la iteración tiene elementos que recorrer.
- `E next()` devuelve el siguiente elemento (`E`) a recorrer en la iteración o lanza la excepción `NoSuchElementException` si la iteración no tiene más elementos.
- `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`.

Criterios:

- Iteración de colecciones de forma independiente a su implementación.
- Uso de los interfaces `Iterable` e `Iterator` con genericidad.

3. **Autenticación de usuarios**

   Deberéis escribir código para un sistema de autenticación de usuarios que utilice Autenticación Multifactor (MFA).

   En primer lugar necesitaréis una clase para representar la **pantalla de login**. Un algoritmo de autenticación típico en esta pantalla sería como sigue:

   - Se introduce el **identificador** del usuario (habitualmente un email, pero eso se puede cambiar en cualquier momento).
   - Se **valida el identificador** (es decir, se comprueba que el identificador tiene un *formato* válido).
   - Se introduce la **contraseña** del usuario (es decir, el *primer factor* de autenticación).
   - Se **autentica la contraseña** (es decir, se comprueba que la contraseña corresponde al identificador).
   - Si la contraseña es correcta, se genera automáticamente un **código MFA** (es decir, el *segundo factor* de autenticación).
   - Para simplificar la implementación, **el proceso termina aquí**, con la generación del código MFA. Realmente no se enviará el código ni se introducirá en ningún sitio.

   Este proceso se puede repetir cuantas veces se quiera y para diferentes **usuarios**. Es decir, habrá varios usuarios, cada uno con sus identificadores, su contraseña, su estrategia MFA preferida, etc.

   En este ejercicio vamos a aplicar un Principio de Diseño conocido como "Encapsula lo que varía". Las partes que *varían* en nuestro sistema son:

   - Los identificadores de usuario pueden ser de diversos tipos: email, teléfono móvil, etc.
   - Los códigos MFA pueden ser de diversos tipos: una OTP (One-Time Password) enviada vía SMS, un código generado por una aplicación autenticadora como Microsoft Authenticator, etc.

   Para implementar esto tendremos una jerarquía basada en interfaces (o clases abstractas) similares a las siguientes:

```
public interface LoginStrategy {
    boolean validateId(String id);
    boolean authenticatePassword(String id, String password);
    // ...
}
```

```
public interface MfaStrategy {
    String generateCode();
    // ...
}
```

   El comportamiento de las estrategias dependerá del contexto, y es como sigue. Por ejemplo, imagina que estamos en la pantalla de login y vamos a introducir los datos. Supongamos que la estrategia de login actual (que se almacena como un atributo de la clase pantalla de login) es que el identificador sea el email. Entonces,

`validateId()` comprobará la validez del identificador según los criterios típicos de los emails (p.ej., que contenga una "@", etc.). Asimismo, `authenticatePassword()` buscará que esa contraseña corresponda a ese email.

La estrategia de login actual para la pantalla de login se puede cambiar en cualquier momento, pasándole a un **setter una instancia de la nueva estrategia** por parámeto. El setter sería similar a este:

```
public void setLoginStrategy(LoginStrategy loginStrategy) {
    this.loginStrategy = loginStrategy;
    // ...
}
```

Entonces, si por ejemplo decidimos cambiar la estrategia actual de identificación para pasar a usar el número de teléfono móvil como identificador, los métodos mencionados arriba se comportarán de manera distinta: `validateId()` comprobará que el id sea un número entero con una longitud determinada, etc. Y `authenticatePassword()` comprobará que esa contraseña corresponda a ese móvil.

De manera análoga, un usuario tendrá una `MfaStrategy` preferida, que, en función del valor actual, provocará que se generen códigos del tipo correspondiente. Por ejemplo, la MFA de GitHub nos permite elegir entre: un SMS (que tendrá 6 dígitos), un código enviado a la app GitHub Mobile (que tendrá 2 dígitos), un código generado por una aplicación autenticadora (que tendrá 6–8 dígitos),... además de otros métodos con códigos alfanuméricos, etc. La idea es que existe una gran variedad de estrategias posibles que se comportan de manera muy diferente pero que siempre devuelven un String. Nuevamente, la `MfaStrategy` preferida actual se podrá cambiar mediante un setter.

Todo el proceso se ejecutará mediante setters y llamadas a métodos. No habrá interfaz gráfica ni se introducirán datos por teclado.

Se pedirán al menos tres clases de estrategias de login concretas y tres clases de estrategias de MFA concretas. En la práctica, los códigos MFA serán más o menos semi-aleatorios, pero intentad que las distintas formas de generarlos estén bien diferenciadas y sigan una lógica.

Adicionalmente, deberéis escribir las clases necesarias para gestionar el resto de los datos y las funcionalidades del sistema, decidiendo cuáles son y dónde va cada cosa: la información sobre los usuarios, las contraseñas, etc. Se recomienda utilizar estructuras tipo `Map` para establecer correspondencias entre identificadores y contraseñas.

Finalmente, deberéis incluir los habituales tests de unidad en los que se prueba cada clase por separado. Debido a que los códigos de MFA serán más o menos semi-aleatorios, no se harán asserts del valor del código MFA sino que se comprobarán cosas como que sea un número o no, que tenga la longitud esperada, que no se repita, etc. Adicionalmente a todo esto, y si queréis visualizar mejor la ejecución, se permitirá añadir un `main` que imprima el proceso por pantalla, pero el `main` no formará parte de los tests.

Criterios:

- Abstracción y uso de interfaces o clases abstractas.
- Polimorfismo y Ligadura dinámica.
- Manejo de estructuras tipo diccionario como `Map`.

4. **Diseño UML**

   Según un estudio, se estima que el mercado de las tecnologías de la información fue el responsable de entre el 3 y el 4 % de las emisiones de CO_2 del mundo en 2020. Ante esta realidad, los centros de investigación y las empresas empiezan a tomar medidas, con acciones dirigidas a reducir el impacto medioambiental de la digitalización y, en concreto, de la inteligencia artificial.

   Los **algoritmos** de inteligencia artificial se dividen, principalmente, en tres grandes categorías:

   - **Supervisados**: donde podemos encontrar tanto algoritmos de **clasificación** como de **regresión**.
   - **No supervisados**: en esta categoría se incluyen tanto algoritmos de **clustering** como de **asociación**.
   - **Por refuerzo**.

   Y nos interesa conocer su nombre, ubicación y la librería donde está implementado. Las ubicaciones posibles serán España, Portugal, Francia o Alemania. Además, de los algoritmos de clasificación debemos saber si son binarios o multiclase.

   Las emisiones de carbono emitidas por cada algoritmo, expresadas en kilogramos de CO_2, se calculan a través del producto de dos factores principales:

   - La **intensidad de emisión**, dada por la ubicación donde se está ejecutando el algoritmo.

Categoría	Intensidad de emisión (gCO_2/kWh)
España	190
Portugal	201
Francia	55
Alemania	301

   - La **potencia consumida** por el algoritmo (en kWh).

   En el caso de los algoritmos de aprendizaje por refuerzo, y debido a la fase de exploración que deben realizar con el entorno, entra en juego una tercera variable **interacción** en el cálculo de las emisiones, que multiplicará al producto anteriormente calculado.

   Además, se dispone de una clase **Proyecto**, definida por un nombre y el centro de investigación/empresa que lo desarrolla, y que incluye los siguientes métodos:

   - Un método para agregar los algoritmos de inteligencia artificial empleados en el proyecto: `addAlgorithm`.
   - Un método `printAlgorithms` que imprima el listado de los algoritmos del proyecto.
   - Un método `calculateC02Emissions`, que devuelva la suma total de las emisiones de CO_2 emitidas por los algoritmos del proyecto.

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 `calculateC02Emissions`.

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).

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.