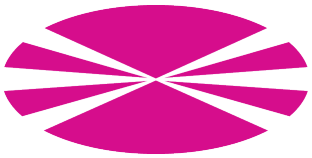

UNIVERSIDADE DA CORUÑA

Diseño Software

Boletín de Ejercicios 2 (2021-2022)

INSTRUCCIONES:
Fecha límite de entrega: 12 de noviembre de 2021 (hasta las 23:59).

- **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. **Integrantes del Colegio Hogwarts de Magia y Hechicería**

   En el Colegio Hogwarts de Magia y Hechicería existen **integrantes** que se dividen en dos categorías:

   - **Residentes**: en esta categoría se incluye tanto a **estudiantes** como a **fantasmas**. Cada residente deberá tener una casa asociada, que puede ser Gryffindor, Hufflepuff, Ravenclaw o Slytherin.
   - **Personal**: En Hogwarts, como en todas las demás escuelas, los cursos son impartidos por **docentes**. De los docentes nos interesa conocer la asignatura en la que imparten clase, que puede ser bien Historia, Transformaciones, Defensa, Herbología o Pociones. Cada asignatura es impartida por un único docente. Por simplicidad, los docentes se consideran neutrales y no pertenecientes a una casa. Además, forman parte del personal laboral del Colegio **conserjes** y **guardabosques**.

   Cada integrante de Hogwarts se identifica con un nombre, apellido, edad y recibe una recompensa anual por parte del Ministerio de Magia que debemos calcular, de forma simplificada, como el número de horrocruxes destruidos multiplicado por el importe en galeones de su categoría. Además, en el caso de que el residente pertenezca a la casa Slytherin, su recompensa se duplica. Asimismo, los docentes de la asignatura de Defensa verán reducida su recompensa un 25 %.

Categoría	Galeón/horrocrux
Docente	50
Conserje	65
Guardabosques	75
Fantasma	80
Estudiante	90

   Por otra parte, todo el personal laboral recibe un salario que se calcula como se indica a continuación dependiendo de su categoría:

   - **Guardabosques** y **Conserje**: el salario es la suma de un sueldo base, de 170 galeones para los guardabosques y 150 para los conserjes, más un complemento de 10 galeones por nocturnidad.
   - **Docente**: en este caso, el salario viene dado por la asignatura en la que imparte docencia, que será más elevado según su peligrosidad. No tendrán ningún tipo de complemento adicional.

Categoría	Sueldo (galeones)
Defensa	500
Transformaciones	400
Pociones	350
Herbología	250
Historia	200

Finalmente, existirá una clase **Colegio** que tenga los métodos que se detallan a continuación:

- Un método `imprimirRecompensas` que devolverá un `String` con un listado con el nombre, apellido, la categoría y la recompensa por la destrucción de horrocruxes de cada integrante y, al final, la recompensa total de Hogwarts. Por ejemplo, un resultado de llamar a `imprimirRecompensas` sería:

```
Hermione Granger(Estudiante de Gryffindor,3 horrocruxes): 270.0 galeones
Barón Sanguinario(Fantasma de Slytherin,1 horrocruxes): 160.0 galeones
Rubeus Hagrid(Guardabosques,2 horrocruxes): 150.0 galeones
Minerva McGonagall(Docente de Transformaciones,1 horrocruxes): 50.0 galeones
Severus Snape(Docente de Defensa,2 horrocruxes): 75.0 galeones
Argus Filch(Conserje,0 horrocruxes): 0.0 galeones
La recompensa total del Colegio Hogwarts es de 705.0 galeones
```

- Un método `imprimirSalarios` que devolverá un `String` con un listado con el nombre, apellido, la categoría y el salario del personal y, al final, el gasto total en personal de Hogwarts. Por ejemplo, un resultado de llamar a `imprimirSalarios` sería:

```
Rubeus Hagrid(Guardabosques): 180 galeones
Minerva McGonagall(Docente de Transformaciones): 400 galeones
Severus Snape(Docente de Defensa): 500 galeones
Argus Filch (Conserje): 160 galeones
El gasto de Hogwarts en personal es de 1240 galeones
```

  Donde Rubeus Hagrid es un guardabosques que ha destruido 2 horrocruxes, por lo que el Ministerio de Magia le dará una recompensa de $75 \times 2 = 150$ galeones. Su sueldo base como guardabosques es de 170 galeones, a los que se le suma un complemento adicional de 10 galeones por nocturnidad, lo que deja su salario total en 180 galeones. Severus Snape es el docente de la asignatura de Defensa, que ha destruido 2 horrocruxes, por lo que recibirá una recompensa de $50 \times 2 = 100$, a lo que habría que descontarle un 25 %, lo que deja su recompensa final en 75 galeones. Su salario es de 500 galeones.

Criterios:

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

2. **Alquiler de apartamentos. Interfaces `Comparable<T>` y `Comparator<T>`**

   En este ejercicio deberéis hacer uso de los interfaces `Comparable<T>` y `Comparator<T>` del API de Java para implementar código de una aplicación de alquiler de apartamentos. Un anuncio de un apartamento se caracteriza por diferentes atributos, de los cuales será siempre obligatorio indicar el **número de referencia** del anuncio. En el precio, deberéis separar, por un lado, el **precio base** de la vivienda en sí y, por otro, los **precios de cada plaza o plazas de garaje** que pueda haber asociadas, que se consideran algo opcional. Incluid también atributos para especificar características típicas de los apartamentos relacionadas con el tamaño, el código postal y otras cosas que veáis necesarias. En el `equals` y el `hashCode` deberéis ignorar el atributo del número de referencia, ya que en la práctica es habitual mostrar varios anuncios para lo que en realidad es el mismo piso (por ejemplo, anuncios de diferentes agencias). El número de referencia constituirá también el llamado **orden natural** de comparación, es decir, el criterio de comparación "por omisión" cuando no se especifica otro. Su propósito se explica debajo.

   La clase que gestiona los apartamentos en alquiler deberá almacenar dos atributos: una **lista de apartamentos** y un **criterio de comparación** actual para los apartamentos o, dicho de otra manera, una *instancia* de lo que se conoce como un **comparador** (el concepto de comparador en Java se explica al final del enunciado). El valor inicial de este comparador deberá ser `null`. La aplicación deberá proporcionar un método para asignar un nuevo valor al comparador y otro para reordenar los apartamentos en base al valor actual del mismo. Más concretamente, este método para ordenar deberá primero comprobar si en el criterio de comparación hay almacenado algún valor en ese momento, es decir, si no es `null`. En caso de ser `null`, deberá simplemente ordenar por el orden natural, es decir, el número de referencia. En el caso de sí haya un comparador almacenado, deberá ordenar los apartamentos según dicho comparador.

   En cuanto a criterios de comparación personalizados, deberéis codificar varias **clases comparador**. Una de ellas deberá representar el precio excluyendo las plazas de garaje, es decir, simplemente el precio base. Otra será el precio total, es decir, el precio base más la suma de los precios de todas las plazas de garaje asociadas. Añadid a esas dos clases al menos otros dos comparadores adicionales basados en otros atributos que elijáis.

   En resumen, un uso típico de vuestra aplicación sería ir cambiando el criterio de ordenación y seguidamente mandar reordenar la lista de apartamentos, repitiendo este proceso tantas veces como se quiera.

   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 (`https://docs.oracle.com/en/java/javase/16/docs/api/`). A la hora de analizar un interfaz prestad especial atención a los métodos abstractos que mencionamos, 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.

Criterios:

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

3. **Red de intereses de usuarios**

   Este ejercicio consiste en implementar una red para que los usuarios puedan indicar aquellos temas en los que están interesados. La intención es poder utilizar esta información para enviar comunicaciones personalizadas, como promociones o avisos de determinados eventos. Para ello debéis crear la clase `Network`, que permite realizar las siguientes acciones:

   - Dar de alta y de baja a los usuarios de la red.
   - Modificar los temas que interesan a un determinado usuario.
   - Obtener los temas que le interesan a un determinado usuario.
   - Encontrar a todos los usuarios interesados en un tema específico.
   - Buscar los temas de interés que tienen en común dos usuarios.
   - Obtener la lista de todos los temas que interesan a los usuarios de la red.
   - Ver la información de todos los usuarios de la red, es decir, sus nombres de usuario junto con su correspondiente lista de temas de interés.

   Además, queremos separar las funcionalidades de la clase `Network` de la implementación propiamente dicha de la red. De esta forma, podremos analizar el funcionamiento de diferentes implementaciones y podremos comprobar, por ejemplo, cuál es la más eficiente. Para ello será necesario crear la interfaz `NetworkManager`, en la que se define la gestión básica de la red. Esta interfaz deberá incluir los siguientes métodos:

   - `void addUser(String user, List<TopicOfInterest> topicsOfInterest)`: añade un usuario a la red.
   - `void removeUser(String user)`: elimina a un usuario de la red.
   - `void addInterest(String user, TopicOfInterest topicOfInterest)`: añade un tema de interés a un usuario de la red.
   - `void removeInterest(String user, TopicOfInterest topicOfInterest)`: elimina un tema de interés a un usuario de la red.
   - `List<String> getUsers()`: devuelve una lista con todos los usuarios registrados.
   - `List<TopicOfInterest> getInterests()`: devuelve una lista con todos los temas que interesan a los usuarios de la red.
   - `List<TopicOfInterest> getInterestsUser(String user)`: devuelve los temas de interés para un usuario determinado.

   Vamos a crear ahora las siguientes dos implementaciones del interfaz `NetworkManager`:

   - La primera debe estar basada en tablas, para lo que se recomienda modelar la red haciendo uso de una matriz y utilizar listas de usuarios e intereses a modo de índices.
   - Para la segunda se hará uso del interfaz `Map` del API de Java, que permite representar una estructura de datos almacenando pares clave/valor (`https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Map.html`), haciendo uso de alguna de sus clases implementadoras, como por ejemplo `HashMap` o `LinkedHashMap`.

Es importante destacar que la clase `Network` tiene que delegar la gestión de la red de intereses a `NetworkManager`, sin ser consciente de si se está usando una u otra implementación a la hora de llevar a cabo sus funcionalidades. La implementación escogida será indicada en el constructor de la propia clase `Network`, que recibirá como parámetro un objeto de cualquiera de las clases implementadoras de la interfaz `NetworkManager`.

Criterios:

- Abstracción y uso de interfaces.
- Polimorfismo y Ligadura dinámica.
- Manejo de matrices y listas.
- Manejo de estructuras tipo diccionario como `Map`.

4. **Diseño UML**

   La UDC y la FIC están planificando nuevas actividades que forman parte del Plan de Igualdade para disminuir la Brecha Digital de Género y conseguir una igualdad real y efectiva entre hombre y mujeres. Estas actividades, una editatona y un taller introductorio, tienen el objetivo de familiarizar a los participantes con la enciclopedia libre Wikipedia. La editatona o maratón de edición con perspectiva de género es una actividad en la cual varias personas se juntan con el objetivo de crear contenido nuevo o mejorar el existente en Wikipedia.

   - La Editatona de Creación de Mujeres STEM será una actividad dirigida a alumnado y profesorado de la UDC.
   - El Taller de Introducción a la Wikipedia para Júniors será una actividad dirigida a estudiantes de 4º, 5º y 6º de Primaria y será preliminar a la campaña 11F (Día Internacional de la Niña y la Ciencia, `https://11defebrero.org/`).

   Para cualquier persona implicada en la iniciativa será necesario almacenar: nombre, apellidos, DNI y correo electrónico. A mayores se necesita cierta información adicional en función de la categoría:

   - Participantes: sexo y categoría.
   - Profesorado: departamento al que pertenece, número de horas de trabajo asignadas para la preparación y celebración del evento y, bonificación.
   - Alumnado: número de horas de trabajo asignadas para la preparación y celebración del evento y, bonificación.
     Para promover la participación tanto del profesorado como del alumnado en este tipo de iniciativas la institución les recompensa con una pequeña bonificación (en horas). En el caso de los profesores se gratificará con el 20 % de las horas dedicadas en horas libres, siempre y cuando la dedicación sea superior a 30 horas. El alumnado se beneficiará de un reconocimiento de créditos como actividades de aprendizaje servicio, se le podrá reconocer 1,5 créditos por cada 15 horas con un máximo de 3 créditos. Además, cada estudiante tendrá asignado un profesor para facilitar la organización de las tareas pertinentes.

   Se dispone de una clase `Evento` que incluye los siguientes métodos que nos permiten la gestión de participantes así como obtener cierta información de interés sobre el evento:

   - Métodos para agregar voluntarios/participantes al evento: `agregarVoluntario` y `agregarParticipante` para agregar un voluntario o un participante concreto y, `agregarVoluntarios` y `agregarParticipantes` para agregar una lista de voluntarios participantes que se pasen por parámetro.
   - Un método `listaParticipantes` que devuelva una lista con las personas participantes independientemente de la actividad a la que se han inscrito.
   - Un método `desgloseParticipantes` que devuelva el número de participantes hombres o mujeres inscritos en las actividades en función del valor que se le pasa por parámetro.

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 `desgloseParticipantes`.

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.