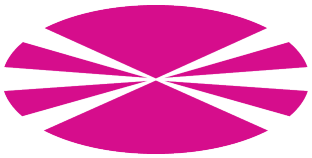

UNIVERSIDADE DA CORUÑA

Software Design

Assignment 2 (2022-2023)

INSTRUCTIONS:
Deadline: November 11, 2022 (until 23:50).

- **Structure of the exercises**
  - You must create a specific IntelliJ IDEA project for this assignment and push it to your repository. The name of the project will be the team's ID followed by the -A2 suffix. (e.g., `DS-61-01-A2`).
  - You will create a package for each exercise in the assignment, with the following names: `e1`, `e2`, and so on.
  - You must follow the instructions for all exercises closely, as they aim to test your knowledge of particular aspects of Java and object orientation.
- **Tests JUnit and coverage**
  - Each exercise must have one or more JUnit 5 tests associated with it to check that it is working correctly.
  - Unlike the first assignment, **it is your responsibility to create the tests and ensure their quality** (high coverage percentage, at least 70-80%, checking all the fundamental aspects of the code, etc.).
  - **IMPORTANT: The test is part of the exercise. If you neglect to do it or it is obviously incorrect, that will mean that the exercise is incorrect.**
- **Evaluation**
  - This assignment is 1/3 of the final practicum grade.
  - In addition to the general criteria already stated in the first assignment there will be specific correction criteria that we will detail in each exercise.
  - A global requirement for all the exercises of this assignment is **using generics correctly** in those interfaces and classes that are generic in the Java API.
  - Check the *Problems* window (`Alt+6`) of IntelliJ to find warnings such as "*Raw use of parametrized class...*" or "*Unchecked call to...*" that indicate an incorrect use of generics.
  - Failing to follow the rules we have set here will result in a penalization in the grade.

1. **Human crew of a film production**

   In a film, the **crew** consists of members that are identified by name, surname, id, telephone and nationality. It is usually subdivided into the following categories:

   - **Technician**: this category includes the most relevant people in the production part of the movie: **screenwriter**, **musician**, **producer** and **director**.
   - **Artist**: this category includes **actor**, **stunt performer** and **dubber**. For the actors, we are interested in knowing their role in the movie: protagonist, secondary or figurant.

   Each member of the crew receives a salary, which is calculated, in simple terms, as the number of hours of work multiplied by the salary/hours ratio of the category. The values for each category are as follows:

Category	Salary/hours (€)
Screenwriter	70
Musician	60
Producer	90
Director	100
Actor	200
Stunt performer	40
Dubber	20

   The actors receive an extra if they are a protagonist. More specifically, the salary is tripled. Similarly, the stunt performers receive a bonus of €1000 if they have participated in dangerous scenes. As for technicians, the screenwriters will receive an extra of €4000 if the script is an original screenplay. For directors we will consider their seniority in the industry, adding an extra of €1000 per year.

   Technicians also receive royalties, as follows: 5% screenwriters, 5% directors, 4% musicians and 2% producers. All of it through their Copyright Office. Royalties are calculated based on box office revenue.

   Finally, there will be a class called **Film** identified by title and box office revenue (in euros), with the following methods:

   - `printSalaries`, which returns a `String` listing the name, surname, category and salary of each member of the crew and, at the end, the total payroll of the whole crew. For example, a result of `printSalaries` for the movie "Alcarrás" would be:

```
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
```

...in which Josep Abad is a stunt performer who has worked 100 hours in the movie "Alcarrás", which include the filming of dangerous scenes. His base salary would be 100 hours × €40/hour = €4000, to which we add a bonus of €1000 for work hazard, which results in a total of €5000.

- `printRoyalties`, which returns a `String` listing name, surname, category and amount for each member regarding royalties. For example, a result of `printRoyalties` would be:

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

  ...assuming €300,000 is the box office revenue for the movie "Alcarrás".

Criteria:

- Encapsulation.
- Creation of inheritance- and abstraction-based hierarchies.
- Using polymorphism and dynamic binding.
- Using generics.

2. **Iterating through lists**

In a reality TV show, there is only one available slot for the last contestant, but there are many possible candidates (n) that have obtained the same score in the classification trials.

In order to break the tie, the show's producers have decided to gather all the candidates, assign them a consecutive number from 1 through n, and then iterate through the list discarding candidates using the procedure below:

- Take a random integer k between 1 and n.
- Count k positions starting from the first in the list (which will be position 1).
- When you reach the element k, you remove it from the list and consider it disqualified.
- Repeat the process starting from the next element in order after the element k, which will be the first position in the new count.
- The process ends when a single contestant remains in the list, who will be the winner.

In order to avoid suspicions of cheating and calculating the winning position in advance, there must be two possible ways of iterating through the list, explained below. One of the two will be chosen at the last moment:

- **Rebound Iteration**: when you reach the end of the list moving forward, you go back to the beginning moving backwards. Therefore, iterating through a list with five elements will be like this: `1-2-3-4-5-4-3-2-1-2-3`, and so on.
- **Circular Iteration**: when you reach the end of the list you jump back to the beginning. Therefore, iterating through the list will be now: `1-2-3-4-5-1-2-3-4-5-1`, and so on.

Let us see two examples of the functioning of the algorithm with both iterations:

A **Rebound Iteration** with `n=5` and `k=3` would be as follows:

```
1-2-3-4-5 => The list of candidates. We start at 1.
1-2-4-5   => We count to 3 and remove candidate no. 3
1-2-5     => We start at 4, count to 3, and remove candidate no. 4
1-5       => We continue at 2, bounce back to 2 and remove no. 2.
1         => We continue at 5, bounce back twice and remove no. 5.
Candidate no. 1 is the winner.
```

A **Circular Iteration** with `n=5` and `k=3` would be as follows:

```
1-2-3-4-5 => The list of candidates. We start at 1.
1-2-4-5   => We count to 3 and remove candidate no. 3
2-4-5     => We start at 4, count to 3, and go back to 1 and remove no. 1.
2-4       => We continue at 2, arrive at 5 and remove no. 5.
4         => We continue at 2, then 4, start again at 2 and remove no. 2.
Candidate no. 4 is the winner.
```

Your task is as follows:

- Create a class `TVRealityList` that stores a list of `String`, which will be a list with the names of the candidates.
- This class must implement the `Iterable` interface and return, when it is requested, an object of the type `Iterator`, which iterates through it. There must be a way to choose which of the two possible iterators will be returned.
- Create two classes that implement the `Iterator` interface. Each class will implement one of the two possible iterators (i.e. Rebound and Circular).
- Create in another class a method called `selectCandidates`, which receives a `TVRealityList` (for which we will have specified one of the two iterations) and an increment `k`, and returns a `String` with the name of the winner.

Below we show a summary of the methods of the interfaces `Iterable` and `Iterator`. The full specification can be found at: `https://docs.oracle.com/en/java/javase/18/docs/api/index.html`.

The `Iterable<T>` **interface** only contains an abstract method:

- `Iterator<T> iterator()`: Returns an iterator over elements of type `T`.

The `Iterator<T>` **interface** has the following methods that must be implemented:

- `boolean hasNext()`: Returns `true` if the iteration has more elements.
- `E next()`: Returns the next element in the iteration. Throws `NoSuchElementException` if the iteration has no more elements
- `void remove()`: Removes from the underlying collection the last element returned by this iterator using `next()`. It can only be invoked once for each execution of `next()`. If `next()` has never been invoked or if `remove()` is called twice without invoking `next()`, the method will throw `IllegalStateException`.

Criteria:

- Iterating through collections independently of their implementation.
- Using the `Iterable` and `Iterator` iterfaces with generics.

3. **User authentication**

   You must write code for an MFA (Multi-Factor Authentication) system.

   First of all, you will need a class representing the **login screen**. A typical authentication algorithm on this screen is as follows:

   - Enter a user's **identifier** (typically an email, but this can be changed at any moment).
   - **Validate the identifier** (i.e. we check that the identifier has a valid *format*).
   - Enter the user's **password** (i.e. the *first authentication factor*).
   - **Authenticate the password** (i.e. we check that the password corresponds to the identifier).
   - If the password is correct, an **MFA code** is automatically generated (i.e. the *second authentication factor*).
   - In order to simplify the implementation, **the process stops here**, with the generation of the MFA code. The code is not actually sent o entered anywhere.

   This process can be repeated as many times as we want and for different **users**. That is, there will be several users, each one with identifiers, password, preferred MFA strategy, etc.

   In this exercise we will follow the Design Principle known as "Encapsulate what varies". The parts of the system that *vary* are:

   - Identifiers can be of several types — email, phone, etc.
   - MFA codes can be of several types — an OTP (One-Time Password) sent via SMS, a code generated by an authenticator app such as Microsoft Authenticator, etc.

   In order to implement this we will have a hierarchy based on interfaces (or abstract classes) similar to the ones below:

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

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

   The actual behavior of the strategies will depend on the context, as follows. For example, imagine we are at the login screen and we are going to enter the information. Suppose the current strategy (which is stored as an attribute of the login screen class) is to use the email as an identifier. Then, `validateId()` will check validity according to the usual criteria for emails (e.g. that it contains an "@" symbol, etc). Likewise, `authenticatePassword()` will try to check that password against that email.

   The current login strategy for the login screen can be changed at any moment by **passing an instance of the new strategy to a setter** as an argument. The setter would be something like this:

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

Then, if for example we decide to change the current login strategy to use a phone number as an identifier, the above mentioned methods will behave differently — `validateId()` will check that the id is an integer number with a specific length, etc. And `authenticatePassword()` will try to check that password against that phone number.

Analogously, a user will have a preferred `MfaStrategy`, which, depending on the current value, will cause different types of codes to be generated. For example, GitHub's MFA lets us choose between: an SMS (with 6 digits), a code sent to the GitHub Mobile app (with 2 digits), a code generated by an Authenticator (6–8 digits), ...in addition to other methods that return alphanumeric values, etc. The idea is that there is a great diversity of possible strategies that differ significantly from one another but always return a String. Again, the current preferred `MfaStrategy` can be changed with a setter.

The entire process is executed via setters and invoking methods. There is no graphical user interface and the information is not entered via keyboard.

You must write at least three concrete login strategy classes and three concrete MFA strategy classes. In practice, the MFA codes will be more or less semi-random, but try to make distinct and differentiated generation strategies that follow some kind of logic.

Additionally, you must create all the necessary classes for managing the rest of the data and the functionalities of the system, deciding what they are and where they go, i.e. the information on the users, the passwords, etc. We recommend using structures such as `Map` for establishing correspondences between identifiers and passwords.

Finally, you must include the usual unit tests for each separate class. Since the MFA codes are more or less semi-random, the asserts will not check the actual value but things like whether it is a number or not, if it has the expected length, if it does not repeat, etc. In addition to all of this, and if you want to visualize the execution better, you can add a `main` that prints the process on the console, but the `main` will not be a part of the tests.

Criteria:

- Abstraction and use of interfaces or abstract classes.
- Polymorphism and Dynamic binding.
- Managing dictionary-like structures such as `Map`.

4. **UML Design**

   According to a study, the IT sector was responsible for 3–4% of worldwide CO_2 emissions in 2020. In light of this fact, research centers and enterprises have begun to apply measures, with actions that aim to reduce the environmental footprint of digitization and Artificial Intelligence in particular.

   AI **algorithms** can be subdivided into three big categories:

   - **Supervised**, which includes **classification** and **regression** algorithms.
   - **Unsupervised**, which includes **clustering** and **association**.
   - **Reinforcement**.

   We are interested in knowing their name, locations and the libraries they are implemented in. The possible locations are Spain, Portugal, France and Germany. Additionally, we want to know if classification algorithms are binary or multiclass.

   The emissions of an algorithm are expressed in CO_2 kilograms and calculated as the product of two main factors:

   - The **intensity of emission**, resulting from the location where the algorithm is running.

Category	Intensity of emission (gCO_2/kWh)
Spain	190
Portugal	201
France	55
Germany	301

   - The **power consumed** by the algorithm (in kWh).

   Also, since reinforcement learning includes a phase of exploration of the environment, a third variable called **interaction** comes into play, which is multiplied by the product mentioned above.

   In addition to this, we will have a class called **Project**, defined by its name and the research center or enterprise that carries it out, which includes the following methods:

   - `addAlgorithm`, for adding algorithms to the project.
   - `printAlgorithms`, for printing the list of algorithms of the project.
   - `calculateC02Emissions`, which returns the total amount of CO_2 emissions of the algorithms of the project.

   The purpose of this exercise is to develop the static and dynamic models in UML. Specifically, your task is to draw the following:

   - A detailed **UML class diagram** in which all classes are shown, including their attributes, methods and relations. Pay special attention to adornments (multiplicity, navigability, roles, etc.)

- A **UML dynamic diagram**. In particular, a sequence diagram showing the functioning of the `calculateCO2Emissions` method.

In order to submit this exercise you must create an `e4` package in your *IntelliJ* project and simply put there the corresponding diagrams in a readable format (PDF, PNG, JPG...) with easily identifiable names.

We recommend using **MagicDraw** for drawing the diagrams. This university offers an education license (we have left you instructions in Moodle).

Criteria:

- Inheritance- and abstraction-based hierarchies.
- Polymorphism and dynamic binding.
- Diagrams are complete: with all the appropriate adornments.
- Diagrams are correct: they follow the UML standard faithfully and they are not very low-level (specially in sequence diagrams).
- Diagrams are legible: they have a good organization, they are not blurry, the elements are not so scattered that zoom is constantly needed; etc.