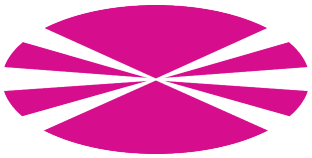

UNIVERSIDADE DA CORUÑA

Software Design

Practicum 2 (2023-2024)

INSTRUCTIONS:
Deadline: November 17th, 2023 (until 23:59).

- **Structure of the exercises**
  - Exercises will be submitted using *GitHub Classroom*. More specifically, the *assignment* `2324-P2` of the `GEI-DS-614G010152324-OP1` *classroom*.
  - A repository will be created with the name of the *assignment* and the name of the group, which will also be the name of the IntelliJ IDEA project (e.g., `2323-P2-DS-61_jose.perez_francisco.garcia`).
  - 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.
- **JUnit Tests 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 your own classes and 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. **Abstract classes**

   You need to implement an information system for a bank with the following requirements:

   Bank accounts

   - Bank accounts must be identified by an IBAN (International Bank Account Number) code and must store a balance in euros.
   - Accounts must have methods to consult the balance and modify the balance depositing or withdrawing money.
   - There exist, at least initially, two types of accounts. Namely, current accounts and savings accounts.
   - Current accounts have no restrictions in their functioning. On the other hand, savings accounts include the following restrictions: Each withdrawal is penalized with a commission of 4% and a minimum of 3 euros, and the only allow depositing money in amounts greater than 1000 euros.
   - As a general rule, no account allows overdrafts (i.e. withdrawing in excess of what is in a current account).

   Clients

   - Clients are basically identified by their DNI.
   - For simplicity, we can assume that an account belongs only to one client.
   - There are three kinds of clients: Normal/standard clients, preferred clients and VIP clients.
   - Normal clients do not have any special privileges.
   - Preferred clients may incur in overdrafts as long as the debt is not more than 1000 euros, the minimum amount for depositing in savings accounts is lowered to 500 euros and the commission for withdrawing from savings accounts is a 2% with a minimum of 1 euro.
   - VIP clients may incur in overdrafts, have no minimum amount for depositing in savings accounts, and there are no commissions for withdrawing from savings accounts.

   Criteria:

   - Encapsulation.
   - Creating inheritance structures and abstraction.
   - Using polymorphism and dynamic binding.

   May be of use: You can assume that amounts are un euro cents in order to avoid dealing with float numbers. That is, you can use a `long` type. It is a simple way of avoiding more complex classes such as `BigDecimal`.

2. **Interfaces**

   You must write code to represent and manage characters in a video game of one-to-one combat. Each character can carry a set of defensive objects (protection objects) and a set of attack objects. The following requirements will be considered:

   Objects

   - There are four types of objects the characters can carry: Armor, Wand, Sword and FireBallSpell.
   - An object may have a function which is defensive (protection), attack or both. **Armor** is a protection object; **FireBallSpell** is an attack object; **Wand** and **Sword** are objects that have both attack and defense characteristics, and can be used in a character for one function or the other.
   - **Attack objects**. They have an integer numerical value that represent the damage they can inflict (reducing the adversary's life by that amount). In an attack, the damage value will be reduced with the protection value of the character that receives the attack. They have an integer value that represents the minimum value of manna that a character must have in order to be able to use it in an attack. Moreover, they have a numerical value that indicates the number of times that a character can use it to attack (it will be decreased by one unit with each attack). Each type of attack object has a maximum number of uses by default.
     - **Wand:** this object, when used for attacking, multiplies by 2 its power of attack in the first use.
     - **FireBallSpell:** this attack object subtracts 1 to its power of attack when there is only one use left.
   - **Protection objects (defense)**. They have associated numerical values indicating the protection they grant and the minimum strength that the character must have to use it in a defensive action.
   - All the objects have a *codename* (string) attribute that identifies them.
   - Each type of object could have more attributes of its own in the future or in different behaviors (e.g. attack object with an unlimited number of uses). In the future it could also be interesting to add new types of objects.

   Characters

   - Characters have an identifiable name, a life value (integer between 0 and 20), a strength value (integer between 0 and 10) and a manna value (integer between 0 and 10).
   - Currently, there exist the character classes **Wizard** and **Warrior**, although we could also consider adding others in the future. For simplicity we can assume that the different classes of characters have no attributes of their own, that is, all characters have the same attributes but these may be initialized with different values when you create them.
   - Each character can be equipped with a set of defensive objects (with a maximum of 5), so the protection they give to the character will be the sum of the protection of all of them. Likewise, the character can be equipped with a set

of defensive objects (with a maximum of 5), using in each attack the first if them and removing it when the number of remaining uses becomes zero.

- When a a character is assigned an object with both defense and attack functions, it must be chosen which one of them is assigned to the character.

Game

- You will implement a basic simulator of the game that provides two options:
  - Simulation of a single attack from one character to another, so that the first acts as an attacker and the second as a defender. The result will be the effective damage of the attack, taking into account the value of the attack and protection of the characters, as it corresponds.
  - Simulation of a turn-based fight between two characters (sequence of alternating attacks between them). The turn-based fight ends when one of them dies (their life level becomes zero) or when we reach a maximum number of turns indicated as a parameter. At the end, we will return the winning character or *null* if the maximum number of turns has been reached with no victor.

Criteria:

- Solving multiple inheritance through interfaces.
- Use of inheritance and abstraction.

3. **Comparing and sorting musical artists**

   To compare elements and sort collections, Java uses the interfaces `Comparable<T>` and `Comparator<T>`, and also methods like `sort` from the class `Collections`. Below we include a brief description of them (the full description is in the Java API documentation: `https://docs.oracle.com/en/java/javase/21/docs/api/`):

   - `Comparable<T>` includes the method `compareTo(T o)`, which compares this object with the specified object according to their so-called "natural" order. It returns a negative integer, zero, or a positive integer if this object is, respectively, lesser than, equal to, or greater than the specified object.
   - `Comparator<T>` is similar to the above but includes a method to compare any pair of objects, namely, `compare(T o1, T o2)`. It returns a negative integer, zero, or a positive integer if the first argument is, respectively, lesser than, equal to, or greater than the second.
   - `Collections.sort` has two variants. In the first one, we pass a list of elements that implement the `Comparable` interface and the method sorts the list according to their "natural" order. The second variant adds a `Comparator` as an argument and sorts the list according to that comparator rather than the "natural" order.

   This exercise continues the topics of the third exercise of the first assignment. First of all, go to `https://musicbrainz.org/` and take a look at the characteristics that define an **artist**. Based on that, create the class `Artist` and include some of those attributes or similar attributes, in addition to other attributes that are indicated below in the exercise.

   You must also allow managing lists of artists. Those lists must allow **sorting** and you must implement two methods for that purpose. Firstly, a method without arguments that uses the *natural order* of the artists (which, in this case, it will be based on a `String` called `id`). Secondly, a method with just one argument, whose type must be `Comparator<T>`. Therefore the sorting criterion will be delegated to the specified comparator.

   You must implement two comparators for:

   - The average rating of an artist (an artist stores a collection of ratings).
   - An artist's eclecticism (an artist stores a collection of genres).

   In addition to those two comparators, add two other comparators based on other attributes. In general, try to use diverse data type and make sure that, among the **four comparators**, some are in ascending order and some are in descending order.

   N.B.: It isn't necessary to reuse any code from the first assignment.

   Criteria:

   - Sorting collections.
   - Using `Comparable<T>` and `Comparator<T>`.
   - Using collections of objects y generics.

4. **UML Design**

   Down Experience[1] is a pioneering project in Galicia promoted by Coruña[2]. Its objective is to create businesses that demonstrate that people with Down syndrome and other mental disabilities are capable of entrepreneurship, thus contributing to their social and labor inclusion. The pilot program is El Quiosco de Down Experience located in Plaza de Orense in A Coruña, a catering initiative in which people with and without disabilities work as baristas, serving coffees, calamari sandwiches and other gastronomic offers.

   The goal of the exercise is to implement the basic classes to be used in El Quiosco. Therefore, we will need to store information on all the people involved in the initiative. For each of them it is necessary to know: name, surname, DNI, contact phone number and email. Additional information is also needed depending on the category, as follows:

   - **Customers**: it is of interest to know their customer code and the number of purchases made. This last feature is necessary to offer loyalty discounts to regular customers.
   - **Servers**: it is necessary to know their social security number, salary and the assigned shift (morning, afternoon or alternate shift). To obtain the salary we will take into account that if they work alternate shifts they have an extra in the salary. We will also store their specialty (table, bar, kitchen staff, etc.).
   - **Support staff**: This job arises from the need for support at those times of the year with the heaviest workload. We must know their social security number, their salary and the shift to which they belong (morning or afternoon, there are no support staff with alternate shifts). It is necessary to store information about the organization they come from, since it usually favors collaboration with other entities in favor of the labor insertion of different groups.

   We have a class ElQuiosco that includes the following methods for managing clients and workers:

   - Methods for adding clients or workers to the quiosco, e.g. `addClient`/`addWorker`, `addClients`/`addWorkers` (passing a list as an argument in the last two cases).
   - A `salariesElQuiosco` method that returns the total sum of the salaries of all the workers in the quiosco.
   - A `personsElQuiosco` method that returns a list with all the people (clients and workers) involved in the quiosco.

   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.)

[1] `https://downexperience.com/`
[2] `https://www.downcoruna.org/`

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