MTAT.03.295 - Agile Software Development

Regular Exam - 13 December 2022

Part 2. Practice

General notes:

- You are **allowed** to access any resource on the web.
- You are **NOT allowed** to communicate with anyone during the exam in any way (except with the lecturer).
- You **MUST create** a **PRIVATE repository** at https://gitlab.cs.ut.ee, and add the teaching staff there as collaborators (with admin rights).
- You should submit a TXT or PDF file with the link to your repository with the solution to the Moodle submission. Please leave your solution in the “main” branch of your repository, and do not make any commits to “main” after the end of the exam.
    - Username: orleny85, email: orlenys.lopez.pintado@ut.ee
    - Username: chapelad, email: david.chapela@ut.ee
- IMPORTANT: You MUST **commit to the repository each time that you complete a task** or requirement in the exam. You are welcome to add other commits, for example, if you fix a task completed before. For each commit missing, you will get a penalization of 20% of the points received corresponding to the task. Note that commits whose messages do not correspond to the task will get a penalization of 20% too. For example, if your commit message says, "Task 2, Completed", be sure that the code submitted corresponds to task 2; however, other parts of the implementation may be updated too. Finally, we will discard all the commits submitted after 13:30 (EEST time). These are the minimum required commits and messages (NOT necessarily in the same order). Note that each message must correspond to a different commit:
    - Initial Commit
    - Task 1 Completed
    - Task 2 Completed
    - Task 3 Completed
    - Task 4 Completed
    - Task 5 Completed
    - Task 6 Completed
    - Task 7 Completed

- IMPORTANT: We strongly suggest you **not to share your code before the end of the exam**, to avoid misunderstandings. In the case of projects in which it is evident that the students committed fraud, all the involved students will get 0 points in the exam, meaning that they all fail it. Also, if we observe suspicious behavior either in the implementation or in the commits, the students involved will be called for an online interview with the teaching staff. Examples of suspicious behavior can be, all the commits happening at the end of the exam or with a non-realistic timeline, too many similarities between projects, etc. During the interview, the students will be asked about the code they submitted. Note that, during the interview, we will downgrade the mark of the student for each question not answered correctly, meaning that the student may fail the exam in the meeting.

Road Fighter

You have been hired to work in what is going to be the best fighting simulator of all time: **Road Fighter.** For the first release of the simulator, we ask you to implement a Phoenix application with a basic set of features to handle fighters and a fighting system.

DIO	
Attack	150
Defense	200
HP	1,000

Jotaro	
Attack	700
Defense	100
HP	500

The application must satisfy the following requirements:

1. A fighter is defined with a **name** (which will be their ID), **two non-negative integers** representing their attack and defense power, respectively, and **one positive integer** representing their HP (i.e. their remaining life). None of these values can be empty.

   The system must allow a user to create a new fighter by introducing their name and characteristics, which should be stored in the database (if valid). The system must guarantee (not only in the front end) that the name of a fighter is unique in the database.

2. The system must allow a user to **organize a fight between two existing fighters**. With this end, the user can **specify the name of two fighters** (non-empty) and, if they exist in the database, the fight takes place.

   When two fighters fight in a fight, they both attack at the same time, **subtracting from the HP of the opponent**, a number of points equal to the **difference between their attack and the defense of the opponent**. For example, if Dio and Jotaro fight (example above), they both attack at the same time, resulting in subtracting 500 from Dio's HP (700 - 200), and subtracting 50 from Jotaro's HP (150 - 100). Meaning that their HP values in the database will be updated based on that subtraction.

   When the HP of a fighter decreases to 0 or lower, they are removed from the database.

The **output** of a fight is **a message** containing **the name of both fighters**, the **result of the fight**, and **if any of them was removed from the database**. The possible scenarios are:

a. **Victory by KO**: the winner survives and reduces the HP of the opponent to 0 (the opponent is removed from the database).
b. **Victory by points**: both fighters survive, but one (the winner) subtracted more points from the opponent than the opponent from them.
c. **Tie by double KO**: both fighters get their HP reduced to 0 (both are removed from the database).
d. **Tie by points**: both fighters survive, and both subtract the same number of points from the opponent.

Tasks to perform:

- **Task 1 (4 points)**: Specification of a Gherkin user story describing requirement number 2 (organizing a fight). As a minimum, your feature must include the clauses Given, And, When, and Then. Your user story must specify that some fighters are already in the database (providing a table with their data). Followed by the information required to organize a fight (passed from the Gherkin description) and the action which triggers such creation. Finally, it should check that the corresponding message is displayed to the user, including the involved fighters' names, the result of the fight, and the name of the players removed from the database (if any). You must consider only the scenario in which the fight ends up in a tie by points.
- **Task 2 (4 points)**: Implementation of the *white_bread* steps described by the user story in Task 1.
- **Task 3 (2 points)**: Setup of the application routes to support the creation of a fighter and the organization of a fight.
- **Task 4 (6 points)**: Setup & implementation of models (via migrations) and seeding the database with some initial data based on your models. The seeding of the database must include multiple fighters.
- **Task 5 (12 points)**: Implementation of controllers. You must implement the operations to render the templates to provide the input data for creating fighters and organizing fights. You must also implement the operations regarding the validation/creation of new fighters, and the validation of fights, including the messages to notify the success/failure (as described in requirements 1 and 2).
- **Task 6 (4 points)**: Implementation of the view(s) and template(s) to introduce the input data for creating fighters, and for organizing fights. You do not need to create/render a new template to display the messages. For that, you can just write in the *flash*, as we did during the course practical sessions.
- **Task 7 (8 points)**: Unit/Integration test checking, at least, the organization of a fight ending up in a "Victory by KO" (a single test may include several assertions). Remember that the test should check that the values in the database were updated as expected:
    - The winner is still in the database.
    - The winner's HP has been modified accordingly, and should be still a positive number (otherwise he/she would have lost or tied).
    - The loser's HP was, before the fight, lower or equal to the HP that she/he lost (otherwise, it wouldn't be a KO).
    - The loser is not in the database.