Python OOP Exam Preparation – 02 April 2020

Overview

Players and Monsters is a battle game. It's all about battles between players with their cards. Each player has health and deck of cards. Each card gives bonus damage and bonus health. The players fight on the battlefield with their cards. You will be provided with **a skeleton** with the project structure.

Setup

- Use the provided **skeleton**
- Build the project and submit it in the **judge system**
- **Do not modify folder structure**
- Name your **methods/attributes exactly as shown**

Judge Upload

For the **first 2 problems**, create a **`zip`** file with the name **`project`** and upload it to the judge system

For the **last problem**, create a **`zip`** file with the name **`tests`** and upload it to the judge system

Problem 1: Structure and Functionality

You need to create **2 abstract classes (`Player, Card`)**. Each of them will have **2 children** (as described below).

You will also need to create **3 more classes**: **`Battlefield`**, **`PlayerRepository`** and **`CardRepository`**

Player

`Player` is a **base class** for any **type of player,** and it **should not be able to be instantiated**.

Constructor

- **`username`**: **`str`** (If the username **is empty string,** raise a **ValueError** with message "**`Player's username cannot be an empty string.`** ")
- **`health: int`** – the health of a player (if the health is below **0,** raise a **ValueError** with message "**`Player's health bonus cannot be less than zero.`** ")

Attributes

- **`card_repository: CardRepository`** - new **card repository** upon initialization.
- **`is_dead: bool –`** calculated property which returns **bool**. (health is 0 or less)

Behavior

`take_damage (damage_points:int)`

The take_damage method decreases players' health with the damage points.

- If the **`damage_points`** are **below 0** raise a **ValueError** with message "**`Damage points cannot be less than zero.`**"

Child Classes

There are several concrete types of **players**:

Beginner

Has 50 initial health points.

Constructor should take the following values upon initialization:

`username`

Advanced

Has 250 initial health points.

Constructor should take the following values upon initialization:

`username`

Card

The **`Card`** is a **base class** for any type of card, and it **should not be able to be instantiated**.

Constructor

- **`name`**: **`str`** (If the card name **is empty string** raise a **ValueError** with message "**`Card's name cannot be an empty string.`**")
- **`damage_points`**: **`int`** (If the damage points are **below zero,** raise a **ValueError** with message "**`Card's damage points cannot be less than zero.`**")
- **`health_points`**: **`int`** (If the health points are **below zero,** raise a **ValueError** with message "**`Card's HP cannot be less than zero.`**")

Child Classes

There are several concrete types of **cards**:

MagicCard

Has 5 damage points and 80 health points.

Constructor should take the following values upon initialization:

`name`

TrapCard

Has 120 damage points and 5 health points.

Constructor should take the following values upon initialization:

`name`

PlayerRepository

A **PlayerRepository** holds information about the players in it.

Attributes

- **`count: int`** – the count of players – **0 upon initialization**
- **`players: list`** – **collection** of players – **empty upon initialization**

Behavior

`add(player: Player)`

Adds a player in the collection.

- If a player exists with a name equal to the name of the given player, raise a **ValueError** with message "**`Player {username} already exists!`**".
- Otherwise, **add the player** and increase the count.

`remove(player: str)`

Removes a player from the collection.

- If the player **is an empty string**, raise a **ValueError** with message "**`Player cannot be an empty string!`**".
- Otherwise, **remove the player** and **decrease** the count of players

`find(username: str)`

Returns a player with that username.

CardRepository

A **CardRepository** holds information for the cards in it.

Attributes

- **`Count: int`** – the count of cards – **0 upon initialization**
- **`Cards: list`** – **collection** of cards – **empty upon initialization**

Behavior

`add(card: Card)`

Adds a card in the collection.

- If a card exists with a name equal to the name of the given card, raise a **ValueError** with message "**`Card {name} already exists!`**".
- Otherwise, **add the card** and increase the **count**

`remove(card: str)`

Removes a card from the collection

- If the **card is an empty string**, raise a ValueError with message "**`Card cannot be an empty string!`**".
- Otherwise, **remove the card** and **decrease the count**

`find(name: str)`

Returns a card with that name.

BattleField

The battlefield is the place where the fight happens.

Behavior

`fight(attacker: Player, enemy: Player)`

That's the most interesting method.

- If one of the users **is_dead**, raise new **ValueError** with message "**`Player is dead!`**"
- If a player is a **beginner,** increase his **health** with **40** points and **increase** the **damage points** of each **card** in the players' deck with **30**.
- Before the fight, both players get bonus health points from their deck. (sum of all health points of his cards)

- Attacker attacks **first** and after that the enemy attacks (deal **damage points to opponent for each card**). If **one of the players** is dead, you should **stop** the fight.

The Controller Class

The controller class should contain methods for **manipulating** the **cards** and the **players**

Commands

There are several commands (**methods**), which control the business logic of the application. They are stated below.

Attributes

- `player_repository`: `PlayerRepository` – **new** repository upon initialization
- `card_redpository: CardRepository` - **new** repository upon initialization

add_player(type: str, username: str)

Functionality

Creates a **player** with the provided **type** and **name**. The method should **return** the following **message**:

`"Successfully added player of type {type} with username: {username}"`

add_card(type: str, name: str)

Functionality

Creates a **card** with the provided **type -> "Magic" or "Trap"** and **name**. The method should **return** the following message:

`"Successfully added card of type {type}Card with name: {name}"`

add_player_card(username: str, card_name: str)

Functionality

Adds the given **card** to the **user** card **repository**. The method should **return** the following message:

`"Successfully added card: {card_name} to user: {username}"`

fight(attack_name: str, enemy_name: str)

Functionality

The **attacker** and the **enemy start a fight** in a **battlefield**. The method should return the following message:

`"Attack user health {attacker_health_left} - Enemy user health {enemy_health_left}"`

report()

Functionality

Returns a report message in format:

```
Username: {username1} - Health: {health1} - Cards {cards_count1}
### Card: {name1} - Damage: {card_damage1}
```

…

Problem 2: Unit Tests

In **the tests folder** there are separate **files** in which you must implement **tests for each class**. The tests should cover all the **functionality and structure** of each class