CSE 1325: Object-Oriented Programming
University of Texas at Arlington
Summer 2023
Alex Dillhoff

Project Phase II

Description

In the second phase of the project, you will expand the features of the TableTop RPG Combat Simulator to include combat against Monsters. This phase includes the following Java concepts: Inheritance, Abstract Classes and Methods, and Exception Handling. The main features of this phase are:

- PVE Combat against monsters
- Ranged weapons and ammunition
- Better error handling with exceptions

Class Changes

`Item` class

Phase 2 will support ranged weapons. To implement this, create a new class `Item` which will serve as a superclass for two subclasses: `Weapon` and `Ammo`. `Item`s should have the following **instance fields**:

- Name
- Quantity

`Ammo` class

The `Ammo` class only needs a single additional instance field: `damage`, which will be the amount of additional damage each shot adds to the weapon's base damage.

`Weapon` class

The `Weapon` class will **add** the following **instance fields**:

- Range (integer)
- Ranged (boolean)

`Creature` class

Create a new class `Creature` which will serve as a superclass for two subclasses. This class should include the following **instance fields**:

- Name
- Creation Date
- Hit Points (HP)
- Armor Class (AC)
- Strength (STR)
- Dexterity (DEX)
- Constitution (CON)
- Inventory (List of `Item` objects)

This class will declare the `abstract` method `attack(Creature)` which will be defined in each of the following subclasses.

Additionally, this class will define the `takeDamage(int)` method which lowers the HP of the Creature based on the input value. **A creature's HP must not be allowed to drop below 0.**

`Player` class

Modify the `Player` class so that it `extends` the `Creature` class. It will include a `Weapon` as one of its instance fields. As with the other instance fields, this should be `private` with corresponding getter and setter methods.

Define the `abstract attack(Creature)` method. For the `Player` class, this will be almost identical to the implementation in Phase I. **Additionally**, if the `Weapon` is a ranged weapon, deduct one `Ammo` from the `Player`'s inventory. You can assume that the main game controller has already verified that the `Player` has enough `Ammo` to fire the weapon.

`Monster` class

Modify the `Monster` class so that it `extends` the `Creature` class. This class will include an instance field of type `enum MonsterType` specifying the type of monster (see below). Define the `abstract attack(Creature)` method. For the `Monster` class, the rules are simplified. First, roll to see if the monster hits the target (`d20 + DEX modifier`). A hit is successful if the resulting value is **greater than or equal to** the target's AC value. On a successful hit, roll damage (`d6 + STR modifier`). The target will take that damage with a call to `takeDamage(int)`.

`MonsterType enum`

Create an `enum` class file name `MonsterType`. This will be used as an instance field for the `Monster` class. This `enum` will have the following values:

- Humanoid
- Fiend
- Dragon

Features

Load/Save Characters

Included in the main menu should be the option to both save and load characters. When selecting the **save character** option, a list of the current characters in memory should be displayed to the user. The user should then be able to select with character is saved. When saving a character, write the character instance fields to CSV format followed by their Weapon stats. Saving a character should create a file based on the character's name (e.g. `"Grog.csv"`) and store it in the relative directory `"saved/players/Grog.csv"`.

Main Menu Example

```
1. Start Game
2. Create Character
3. Load Character
4. Save Character
5. Quit
```

Loading a character should search the relative directory `"saved/players"` for any CSV files and populate a list of files to load. If the user selects one, attempt to load the data and populate an object reflecting that character. If a character with that name is already loaded, warn the user and return to the main menu.

All file operations must catch any exceptions due to non-existing files or other I/O issues.

Name Restrictions for Players

When creating a new character, validate the name based on the following criteria:

- Cannot contain numbers.
- Cannot contain special characters.
- Must start with a capital letter.

Combat Against Monsters

When starting a new game, the user should have the option to play against monsters. If they choose this option, the user will then select the number of monsters to battle. Your code should randomly create the monster objects based on a database of monsters loaded from a CSV file (provided via Canvas).

Start Game Example

```
1. Play with Random Monsters
2. Play with Players Only (PvP)
3. Back
```

The monsters will not be controlled by the user. Instead, they will attempt to attack a player if a player is in range. If there are no players in range, then their turn is skipped. If there are multiple players in range, attack one of them at random. Monsters should be depicted on the 2D map as `M`.

BONUS (10 Points): Implement basic AI movement by having the monsters move as close as they can (5 spaces per turn) towards a player IF they are not already by a player.

Rolling Initiative

With multiple players and monsters on the field, rolling for initiative will become more complicated. Implement a method `rollInitiative(ArrayList<Creature>)` which will sort the list of all creatures based on their individual initiative rolls. If multiple creatures roll the same value, their order will be determined by their position in the sorted array. In other words, you don't need to do anything special for duplicates.

Combat Changes

As with Phase I, only `Player` characters can be controlled by the user. Any `Monster`s on the field must behave as defined above. The battle ends when either all `Player` characters reach 0 HP or all `Monster` characters reach 0 HP.

Ranged Weapons: If a `Player` has a ranged weapon equipped, they can attack a monster that is up to 5 spaces away. The attack will be resolved as normal. Implementing this feature will be done in the main game controller as well as the `Player` class.

When a player chooses to attack with their ranged weapon, the game controller should verify that the target is within range and has the requisite ammunition attack. From the `Player` class, you can assume that the player has a ranged weapon equipped with enough ammunition.

Grading Requirements

This section describes the requirements, based on the features above, that will be considered when grading projects.

Item Class

- Includes instance fields based on the descriptions above.
- Must have appropriate accessors and mutators for fields needed by other classes.

Ammo Class

- Includes instance fields based on the descriptions above.
- Must have appropriate accessors and mutators for fields needed by other classes.

Weapon Class

- Includes instance fields based on the descriptions above.
- Must have appropriate accessors and mutators for fields needed by other classes.

Creature Class

- Includes instance fields based on the descriptions above.
- Must have appropriate accessors and mutators for fields needed by other classes.

Player Class

- All previous instance fields from Phase I should no longer be here (except `Weapon`).
- Includes an instance field for the `Weapon class`.
- Implement `attack(Creature)` from `Creature`.

Monster Class

- Includes `MonsterType enum` as an instance field.
- Implement `attack(Creature)` from `Creature`.
- Show each `Monster` on the map as `M`.

Combat System

- Implement initialization for all Creatures.
- Add ranged attacks to the combat system.

Application

- Implement menus and program control flow based on the descriptions above.
- Should be implemented as a separate file.
- All Players and Monsters should be contained in an instance field of type `ArrayList<Creature>`

Code

- Code should be formatted consistently.
- The project should compile without warnings or errors.

UML Diagrams

- Each class in your project should have a corresponding class diagram.
- The main diagram should show the relationship between all classes in your project.

Submitting

Submit all class files, UML diagrams, and other code files on Canvas as a compressed zip file named `LASTNAME_ID_P2.zip`, where `LASTNAME` is your last name and `ID` is your student ID starting with 1xxx.