HW #3: Dungeons and Dragons

Contents

1 Overview

In this assignment you will implement a single-player multi-level version of a *dungeons and dragons* board game.

You are trapped in a dungeon, surrounded by enemies. Your goal is to fight your way through them and get to the next level of the dungeon. Once you complete all the levels, you win the game.

The program will be checked for correctness, complying to OOP principles and coding conventions.

2 Game Description

The game is played on a board similar to the board in Figure 1. The game board consists of a player, enemies of different types, walls and empty areas that both players and enemies can walk through.

In this board, the symbol @ in green represents the player while the later red symbols B, s, k and M represent the monsters that the player should fight. In addition, there are dots scattered along the paths, representing the free areas and # symbols that represent the walls. The game takes a path to a directory that containing indexed files via the command line argument (explained later). Each file represents a game level.

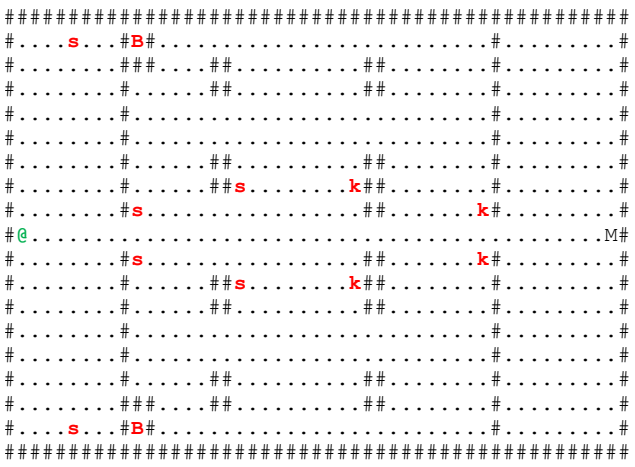

Figure 1: The game board

In this version of the game, there are three types of players which differ in the abilities they have (detailed in the next sections) and two types of enemies: monster and traps. The user controls the player using the keyboard and the computer controls the monsters.

3 Game flow

The game starts with a collection of boards which represent the desired game-levels, running as follows:

- The user chooses a player character.
- The game starts with the first level. Each level consists of several rounds. A round, also called **Game Tick**, is defined as follows:
  - The player performs a single action.
  - Each enemy performs a single action.
- The level ends once the enemies are all dead. In this case, the next level will be loaded up.
- The game ends once the player finished all levels, or if the player dies.

3.1 Game Board

Represented as a 2-dimensional array of chars for both input and output, with each char representing a wall, a game character or a free cell.

4 Game tiles

Each cell is represented by an object called **tile**. There are 3 major types of tiles:

- Empty: player and enemies can walk through.
- Wall: blocks the movement of player and enemies.
- Unit: described below.

Tile properties are:

- Tile: character
- Position: x,y coordinates in a 2d board

4.1 Range

A **Range** between 2 points on our board is defined by their **Euclidean Distance**:

$$range(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

4.2 Units

Game units include of both player and enemy classes. Each unit has the following properties:

- Name: String
- Health, defined by:
  - Health pool: Integer
  - Health amount: Integer
- Attack points: Integer
- Defense points: Integer

4.3 Player

This class represents the player's character. In addition to the game unit properties, each player has the following properties:

- Experience: Integer, Initially 0. Increased by killing enemies.
- Player Level[1]: Integer, Initially 1. Increased by gaining experience.
- After gaining $(50 \times level)$ experience points, the player will level up.
- Upon leveling up the player gains:
  - $experience \leftarrow experience - (50 \times level)$

[1] Not confused by the game level

- $level \leftarrow level + 1$
- $health\ pool \leftarrow health\ pool + (10 \times level)$
- $current\ health \leftarrow health\ pool$
- $attack \leftarrow attack + (4 \times level)$
- $defense \leftarrow defense + (1 \times level)$

4.3.1 Player types

There are three classes that extends *Player*. Each player class has a ***special ability***.

A user can *cast*[2] the special ability to improve its situation at the cost of resources.

1. **Warrior**
   - Special ability: **Avenger's Shield**, randomly hits one enemy withing $range < 3$ for an amount equals to 10% of the warrior's max health and heals the warrior for amount equals to $(10 \times defense)$ (but will not exceed the total amount of health pool).
   - The warrior's ability has a **cooldown**, meaning it can only be used once every *ability cooldown* game ticks.
   - Fields:
     - *ability cooldown*: Integer, received as a constructor argument. Represents the number of game ticks required to pass before the warrior can cast the ability again.
     - *remaining cooldown*: Integer, initially 0. Represents the number of ticks remained until the warrior can cast its special ability.
   - Upon leveling up (**in addition** to the normal updates of Player leveling):
     - $remaining\ cooldown \leftarrow 0$.
     - $health\ pool \leftarrow health\ pool + (5 \times level)$
     - $attack \leftarrow attack + (2 \times level)$
     - $defense \leftarrow defense + (1 \times level)$
   - On game tick:
     - $remaining\ cooldown \leftarrow remaining\ cooldown - 1$.
   - On ability cast:

     if $remaining\ cooldown > 0$ **then**
     generate an appropriate error message.
     else
     - $remaining\ cooldown \leftarrow ability\ cooldown$
     - $current\ health \leftarrow min\left(current\ health + (10 \times defense), health\ pool\right)$
     - Randomly hits one enemy within $range < 3$ for an amount equals to 10% of the warrior's *health pool*

2. **Mage**
   - Special ability: **Blizzard**, randomly hit enemies within *range* for an amount equals to the mage's *spell power* at the cost of **mana**.
   - Fields:

[2]Cast means use. This is the terminology used in this domain.

- *mana pool*: Integer, holds the maximal value of mana. Initial value is received as a constructor argument.
- *current mana*: Integer, current amount of mana. Initially $\frac{mana\ pool}{4}$
- *mana cost*: Integer, ability cost. Received as an argument.
- *spell power*: Integer, ability scale factor. Initial value is received as a constructor argument.
- *hits count*: Integer, maximal number of times a single cast of the ability can hit. Received as an argument.
- *ability range*: Integer, ability range. Received as an argument.

- Upon leveling up (**in addition** to the normal updates of Player leveling):
  - $mana\ pool \leftarrow mana\ pool + (25 \times level)$
  - $current\ mana \leftarrow min\left(current\ mana + \frac{mana\ pool}{4}, mana\ pool\right)$
  - $spell\ power \leftarrow spell\ power + (10 \times level)$
- On game tick:
  - $current\ mana \leftarrow min(mana\ pool, current\ mana + 1 \times level)$
- On ability cast:

  if $current\ mana < cost$ **then**
  generate an appropriate error message.
  else
  $current\ mana \leftarrow current\ mana - cost$
  $hits \leftarrow 0$
  while $(hits < hits\ count) \wedge (\exists\ living\ enemy\ s.t.\ range(enemy, player) < ability\ range)$ **do**
  - Select random enemy within *ability range*.
  - Deal damage (reduce health value) to the chosen enemy for an amount equal to *spell power* (each enemy may try to $defend$ itself).
  - $hits \leftarrow hits + 1$

3. **Rogue**

- Special ability: **Fan of Knives**, hits everyone around the rogue for an amount equals to the rogue's *attack* points at the cost of *energy*.
- Using energy as resource. Starting energy equals to the rogue's maximum energy which is 100.
- Fields:
  - *cost*: Integer, special ability cost. Received as a constructor argument.
  - *current energy*: Integer, initially 100 (maximal value).
- Upon leveling up (**in addition** to the normal updates of Player leveling):
  - $current\ energy \leftarrow 100$
  - $attack \leftarrow attack + (3 \times level)$
- On game tick:
  - $current\ energy \leftarrow min\,(current\ energy + 10, 100)$
- On ability cast:

  if $current\ energy < cost$ **then**
  generate an appropriate error message.
  else
  - $current\ energy \leftarrow current\ energy - cost$
  - For each enemy within range < 2, deal damage (reduce health value) equals to the rogue's *attack* points (each enemy will try to $defend$ itself).

4.4 Enemies

The player may encounter enemies while traveling around the world. Each enemy has the following property:

- Experience value - Integer. The amount of experience gained by defeating this enemy.

4.4.1 Enemy types

The enemies are divided into two types:

1. Monster
   - Fields:
     - *vision range*: Integer, represents the monster's vision range, a constructor argument.
   - On enemy turn:
     - The monster will attempt to traverse around the board.
     - Monsters can move 1 step in the following directions: Up/Down/Left/Right, and may chase the player if the player is within its *vision range*.
     - Movement rules described as follows:

       if $range(monster, player) < vision\ range$ **then**
         $dx \leftarrow enemyX - playerX$
         $dy \leftarrow enemyY - playerY$
         if $|dx| > |dy|$ **then**
           if $dx > 0$ **then**
             Move left
           else
             Move right
         else
           if $dy > 0$ **then**
             Move up
           else
             Move down
       else
         Perform a random movement action: left, right, up, down or stay at the same place.

2. Trap
   - Fields:
     - *visibility time*: Integer, amount of ticks that the trap remains visible, a constructor argument.
     - *invisibility time*: Integer, amount of ticks that the trap remains invisible, a constructor argument.
     - *ticks count*: Integer, counts the number of ticks since last visibility state change. Initially 0.
     - *visible*: Boolean, indicates whether the trap is currently visible. Initially *true*.
   - On enemy turn:
     - A trap can't move (unlike monsters), but updates its state (visibility) on each turn.
     - After *visibility time* game ticks, the trap will turn invisible.
     - The trap becomes visible *invisibility time* game ticks afterwards.
     - The trap may attack the player if the $range\,(trap, player) < 2$.
     - The trap's state will be updated on each turn as follows:

$visible \leftarrow ticks\ count < visibility\ time$
if $ticks\ count == (visibility\ time + invisibility\ time)$ **then**
  $ticks\ count \leftarrow 0$
else
  $ticks\ count \leftarrow ticks\ count + 1$
if $range(trap, player) < 2$ **then**
  $attack(player)$

4.5 Combat system

When the player attempts to step on a location that has an enemy, or when an enemy attempts to step on the player's location, they engage in melee combat.

The attacker is always the unit that performed the step. The other unit will attempt to defend itself. The combat goes as follows:

1. The attacker rolls an amount between 0 and its *attack damage*.
2. The defender rolls an amount between 0 and its *defense points*.
3. If $(attack\ roll - defense\ roll) > 0$, the defender will be damaged, losing health equal to that amount.
4. The defender may die as a result of this attack if his health goes to or below 0.
   - If an enemy is killed by the player:
     - The player gains the experience value of the enemy.
     - The enemy is removed from the game.
     - The player takes the position of the enemy.
   - If the player is killed by an enemy, the player's location is marked with 'X' and the game ends.

4.6 Units implementation tips

- Use visitor pattern to handle the different interaction cases between **Unit** and:
  - Empty tile (free movement)
  - Wall (blocks movement)
  - Player (Enter combat if $Unit$ is an enemy, else nothing)
  - Enemy (Enter combat if $Unit$ is a player, else nothing)
- Avoid using **instanceof** and/or **casting**.
- Health and ability-resource mechanics can be implemented with special setters so it won't exceed boundaries.
- We recommend implementing 3 methods that return a String with the information of each unit:
  - $String\ Tile :: toString()$ - Returns the *tile* character. Use it to print the board.
  - $String\ Unit :: getName()$ - Returns the name of the unit. Use it to print the names upon combat engagement on ability cast.
  - $String\ Unit :: describe()$ - String, returns full information of the current unit (don't forget to **override** this method in each subclass). Use it to print the information of each unit during combat / on player's turn.
    - You can override the $Trap :: toString()$ method so it returns different characters depending on it's visibility state.

5 The CLI (Command line interface)

The game starts by asking the user to select the player character from a list of pre-defined characters.

The CLI should display the game state after each round. That is:

- Whole board.
- Player's stats (name, health, attack damage, defense points, level, experience, and class-specific properties).
- Combat information (Whole stats for both units, attack roll, defense roll, damage taken, death and experience gaining).
- Level up notifications (New level, stats gained).

A user can use the following actions:

Character	ASCII value	Action
'w'	119	Move up
's'	115	Move down
'a'	97	Move left
'd'	100	Move right
'e'	101	Cast special ability
'q'	113	Do nothing

5.1 Interacting with the CLI - Callbacks / Observer pattern

The business layer should not interact with the UI directly.

Forms of interaction with the UI could be done using callbacks or Observer pattern.

You are also allowed to use returned values from functions as long as its **reasonable**.

Do not use *System.out.println* directly from any non-UI class.

6 Forms of input

The program takes a path of directory as **command-line argument**. The directory contains files represent the game boards. Each file is named "level<i>.txt" where <i> is the number of the level (For example: "level1.txt", "level2.txt" etc. Also, see the *level* files attached to the assignment specifications).

We will use the following tiles:

Character	ASCII value	Description
'.'	46	Free, characters can step over
'#'	35	Wall, blocked, no characters may step over
'@'	61	Player's position
'X'	88	Dead player

Any other character may serve as an enemy tile.

The following player and enemy instances are required and should be defined within your code:

- Players:

Warriors								
Name	Health	Attack	Defense	Cooldown				
Jon Snow	300	30	4	3				
The Hound	400	20	6	5				
Mages								
Name	Health	Attack	Defense	Mana Pool	Mana Cost	Spell Power	Hit Count	Range
Melisandre	100	5	1	300	30	15	5	6
Thoros of Myr	250	25	4	150	20	20	3	4
Rogues								
Name	Health	Attack	Defense	Cost				
Arya Stark	150	40	2	20				
Bronn	250	35	3	50				

- Enemies
  - Monsters:

Name	Tile	Health	Attack	Defense	Vision Range	Experience Value
Lannister Solider	's'	80	8	3	3	25
Lannister Knight	'k'	200	14	8	4	50
Queen's Guard	'q'	400	20	15	5	100
Wright	'z'	600	30	15	3	100
Bear-Wright	'b'	1000	75	30	4	250
Giant-Wright	'g'	1500	100	40	5	500
White Walker	'w'	2000	150	50	6	1000
The Mountain	'M'	1000	60	25	6	500
Queen Cersei	'C'	100	10	10	1	1000
Night's King	'K'	5000	300	150	8	5000

  - Traps:

Name	Tile	Health	Attack	Defense	Experience Value	Visibility Time	Invisibility Time
Bonus Trap	'B'	1	1	1	250	1	5
Queen's Trap	'Q'	250	50	10	100	3	7
Death Trap	'D'	500	100	20	250	1	10

You may add pre-defined players and enemies of your own.

7 Testing your game

You should create **unit tests** as regular part of your work-flow.

Make sure that you cover both basic and edge cases.

8 Bonus (+5 points each)

8.1 Add new player class

In this task you will implement the **Hunter** class.

- Special ability: **Shoot**, hits the closest enemy for an amount equals to the hunter's *attack* points at the cost of an *arrow*.
- Using *arrows* as resource. Starting amount of arrows in quiver is equals to $(10 \times level)$.
- Fields:
  - *range*: Integer, shooting range, received as a constructor argument.
  - *arrows count*: Integer, current amount of arrows.
  - *ticks count*: Integer, initially 0.
- Upon leveling up (**in addition** to the normal updates of Player leveling):
  - $arrows\ count \leftarrow arrows\ count + (10 \times level)$
  - $attack \leftarrow attack + (2 \times level)$
  - $defense \leftarrow defense + (1 \times level)$
- On game tick:

  if $ticks\ count == 10$ **then**
  $arrows\ count \leftarrow arrows\ count + level$
  $ticks\ count \leftarrow 0$
  else
  $ticks\ count \leftarrow ticks\ count + 1$
- On ability cast:

  if $arrows\ count == 0$ **then**
  generate an appropriate error message.
  else if $\nexists\ enemy\ s.t.\ range(enemy, player) \leq range$ **then**
  generate an appropriate error message.
  else
  - $arrows\ count \leftarrow arrows\ count - 1$
  - Deal damage equals to *attack* points to the **closest** enemy within *range* (The enemy will try to $defend$ itself).

You are required to add the following to the pre-defined list of players:

Hunter				
Name	Health	Attack	Defense	Range
Ygritte	220	30	2	6

8.2 Add new enemy class

- Create an interface called **HeroicUnit** with a *castAbility* method (parameters are for your choice). Player classes should implement the HeroicUnit interface.
- Add a new enemy class called **Boss** which implement the HeroicUnit interface. Bosses behave mostly like monsters, except they can also use special abilities:
  - Fields:
    - *vision range*: Integer, represents the monster's vision range, constructor argument.
    - *ability frequency*: Integer, how often the boss will cast the ability during combat, constructor argument.

    * *combat ticks*: Integer, how long the boss remains in combat, initially 0.
  - On enemy turn:
    * The boss will attempt to traverse around the board.
    * Boss can move 1 step in the following directions: Up/Down/Left/Right, and may chase the player if the player is within its *vision range*.
    * Movement rules are mostly similar to the monster's rules, except:

if $range(monster, player) < vision\ range$ **then**
  if $combat\ ticks == ability\ frequency$ **then**
    - $combat\ ticks \leftarrow 0$
    - The boss cast the ability: shooting at the player for an amount equals to the boss *attack* points if the player is within *vision range* (the player will try to $defend$ itself).
  else
    $combat\ ticks \leftarrow combat\ ticks + 1$
    $dx \leftarrow enemyX - playerX$
    $dy \leftarrow enemyY - playerY$
    if $|dx| > |dy|$ **then**
      if $dx > 0$ **then**
        Move left
      else
        Move right
    else
      if $dy > 0$ **then**
        Move up
      else
        Move down
else
  $combat\ ticks \leftarrow 0$
  Perform a random movement action: left, right, up, down or stay in the same place.

- Change the following pre-defined enemies so that they implement the **HeroicUnit** interface:
  - The Mountain
  - Queen Cersei
  - Night's King

9 Submission guidelines

You are required to submit a zip file named 'hw3.zip' with the following:

- UML class diagram describing an overview of your work in a file named 'hw3.pdf'.
- Your **compiled files**, **source code** and **tests** in a jar file named 'hw3.jar'.

Your submissions will be check on **Windows 10** platform with **Java 8**.

You can verify that your submission is able to run on such platform by running the following command on such platform:

java -jar hw3.jar <Path to directory of files>