# HW #3: Dungeons and Dragons

## Contents


**1 Overview** **2**

**2 Game Description** **3**

**3 Game flow** **3**

**4 Game units** **4**

4.1 Player . . . . . 4

4.1.1 Player classes . . . . . 4

4.2 Enemies . . . . . 6

4.2.1 Enemy types . . . . . 6

**5 Combat system** **7**

**6 The CLI (Command line interface)** **7**

6.1 Interacting with the CLI - Observer pattern . . . . . 8

**7 Forms of input** **8**

7.1 Testing your game . . . . . 9

7.1.1 Random numbers generation . . . . . 10

7.1.2 User input . . . . . 10

**8 Submission guidelines** **10**

# 1 Overview

In this assignment you will implement a single player multi level version of the *dungeons and dragons* board game [1].

You are trapped within a dungeon, full of enemies: monsters and traps. Your goal is to fight your way through them and get to the next level of the dungeon. Once you complete all levels, you win the game.

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

[1] In assignment 4, you will be required to extend the game to support multi-player mode

## 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 (monsters and traps), 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. Each file represent a game level.

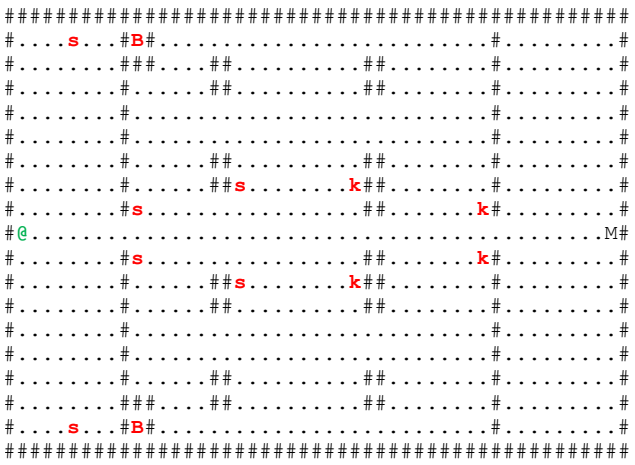

Figure 1: The game board

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 represent the desired levels. Here is the flow of the game:

- 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:
  - Player make a single action.
  - Each enemy make 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.

We will represent the game board 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.

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 Game units

Game units includes of both player classes and enemies. Each unit in our game will have the following properties:

- Name: String
- Health, defined by:
  - Health pool: Integer
  - Current health: Integer
- Attack points: Integer
- Defense points: Integer
- Position (x, y coordinates on a 2d board)

## 4.1 Player

In addition to the game unit properties, a player has the following properties:

- Experience: Integer, Initially 0. Increased by killing enemies.
- Level[2]: 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:
  - $experience \leftarrow experience - (50 \times level)$
  - $level \leftarrow level + 1$
  - $health\ pool \leftarrow health\ pool + (10 \times level)$
  - $current\ health \leftarrow health\ pool$
  - $attack \leftarrow attack + (5 \times level)$
  - $defense \leftarrow defense + (2 \times level)$
- Special ability: specific ability for each player class (See the section below).

### 4.1.1 Player classes

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

A user can *cast*[3] the player special ability to improve its situation at the cost of (limited) resources.

1. Warrior
   - Special ability: **Heal**, heal the warrior for amount equals to $(2 \times defense)$ (but will not exceed the total amount of health pool).
   - The warrior's ability has a cooldown, meaning it can only use it only every *cooldown* game ticks.

[2]Not confused by the game level
[3]Cast means use. This is the terminology used in this domain

- Fields:
  - *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*: Integer, initially 0. Represents the number of ticks remained until the warrior can re-cast its special ability.
- Upon leveling up, **in addition** to the normal updates of Player leveling:
  - $remaining \leftarrow 0$.
  - $health\ pool \leftarrow health\ pool + (5 \times level)$
  - $defense \leftarrow defense + (1 \times level)$
- On game tick:
  - $remaining \leftarrow remaining - 1$.
- On ability cast:

  **if** $remaining > 0$ **then**
  generate an appropriate error message.
  **else**
  $remaining \leftarrow cooldown$
  $current\ health \leftarrow current\ health + (2 \times defense)$

2. Mage

- Special ability: **Blizzard**, randomly hit enemies within its *range* for an amount equals to the mage's *spellpower*.
- Using mana as a resource.
- Fields:
  - *spell power*: Integer, ability scale factor. Initial value is received as a constructor argument.
  - *mana pool*: Integer, holds the maximal value of resources. Initial value is received as a constructor argument.
  - *current mana*: Integer, current amount of resources. Initially $\frac{mana\ pool}{4}$
  - *cost*: Integer, ability cost. Received as an argument.
  - *hit times*: Integer, maximal number of times the ability hit. Received as an argument.
  - *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:
  - $current\ mana \leftarrow min(mana\ pool, current\ mana + 1)$
- On ability cast:

  **if** $current\ mana < cost$ **then**
  generate an appropriate error message.
  **else**
  $current\ mana \leftarrow current\ mana - cost$
  $hits \leftarrow 0$
  **while** $hits < hit\ times \wedge \exists\ enemy\ s.t.\ EuclideanDistance(enemy, player) < range$ **do**
  Select random enemy within *range*.

Attempt to deal damage (reduce health value) 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 amount equal 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 $< 2$, attempt to deal damage (reduce health value) equals to the rogue's *attack* points (each enemy may try to $defend$ itself).

## 4.2 Enemies

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

- Experience value
- Tile (a char, used to represent the enemy character on the game board)

### 4.2.1 Enemy types

The enemies are divided into two types:

1. Monster

   - Additional constructor parameter:
     - *range*: Integer, represents the monster's vision range.
   - Each 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 range.
   - Movement rules described as follows:

     **if** $EuclideanDistance(monster, player) < range$ **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 move: left, right, up, down or stay in the same place.

2. Trap
   - Additional constructor parameters:
     - *range*: Integer, relocation range.
     - *respawn*: Integer, ticks until the trap relocate itself.
     - *visibility time*: Integer, ticks until the trap becomes invisible.
   - A trap can't move (unlike monster). It updates its state (re-spawn and visibility time) on each turn.
   - After *respawn* game ticks, the trap will relocate itself to a random free location within its *range*.
   - The trap will stay visible for the first *visibility time* ticks after each spawn (including the initial spawn).

# 5 Combat system

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

The attacker is always the unit that attempted to perform 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 and the enemy is removed from the game.
   - If the player is killed by an enemy, the player's location is marked with 'X' and the game ends.

# 6 The CLI (Command line interface)

The game start 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).
- Recent combat information.
- Level ups notifications.

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 |

### 6.1 Interacting with the CLI - Observer pattern

The business logic should not interact directly with the UI.

However, forms of interaction with the UI could be done using the returned values from functions or with the Observer pattern.

Think which classes should be marked as *Observable* and which should be marked as *Observer*.

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

## 7 Forms of input

The program takes a path of directory and an optional argument -D (explained later), all passed as command line arguments (not hard coded). The directory contains files represent the game boards. Each file is named "*level i*" where $i$ is the number of the level (See the "*level i.txt*" files attached to the assignment). We use the following tiles:

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 classes and enemies should be defined within your code:

- Players:

| Warriors | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | Health | Attack | Defense | Cooldown | | | | |
| Jon Snow | 300 | 30 | 4 | 6 | | | | |
| The Hound | 400 | 20 | 6 | 4 | | | | |
| Mages | | | | | | | | |
| Name | Health | Attack | Defense | Spell Power | Mana Pool | Mana Cost | Hit Times | Range |
| Melisandre | 160 | 10 | 1 | 40 | 300 | 30 | 5 | 6 |
| Thoros of Myr | 250 | 25 | 3 | 15 | 150 | 50 | 3 | 3 |
| Rogues | | | | | | | | |
| Name | Health | Attack | Defense | Cost | | | | |
| Arya Stark | 150 | 40 | 2 | 20 | | | | |
| Bronn | 250 | 35 | 3 | 60 | | | | |

- 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 | Range | Respawn | Visibility |
|---|---|---|---|---|---|---|---|---|
| Bonus "Trap" | 'B' | 1 | 1 | 1 | 250 | 5 | 6 | 2 |
| Queen's Trap | 'Q' | 250 | 50 | 10 | 100 | 4 | 10 | 4 |
| Death Trap | 'D' | 500 | 100 | 20 | 250 | 6 | 10 | 3 |

You may, however, add player classes and enemies of your own.

## 7.1 Testing your game

In our program, a random behavior are included in the monster movement and in the combat system. Such kind of randomness (non deterministic behavior) makes it difficult to test the program.

In addition, the interactivity of the game, in which the user moves the player, makes it difficult to perform the automatic tests.

In order to test your program, you should enable running a ***non interactive deterministic*** version that does not makes use of a *random number generator*.

It instead uses two files:

- *random_numbers.txt* - a file that includes of numbers that represents the random numbers.

- *user_actions.txt* - a file that includes all user actions.

Your program should run the non-deterministic version by default and the deterministic version if '**-D**' is given as a command line argument.

### 7.1.1 Random numbers generation

We will implement 2 variations of the following interface:

```
public interface RandomGenerator{
    int nextInt(int n);
}
```

- The default implementation holds a *java.util.Random* object. Calling $RandomGenerator.nextInt(n)$ will refer to the $Random.nextInt(n)$ method (which returns an integer between 0 to n).
- If '-D' is passed, the deterministic version is activated. This version uses a file full of numbers (see the attached file *random_numberts.txt*) and simply read the next integer.

### 7.1.2 User input

We'll use the same trick:

```
public interface ActionReader{
    String nextAction();
}
```

- The default implementation reads a line from the *System.in*.
- The deterministic version use a file which includes all user actions (represented as characters) in the current game (see the attached file *user_actions.txt*). Calling $nextAction()$ returns the next line from the file.

You should create **unit tests** as regular part of your work-flow. Make sure that you cover both basic and edge cases.

Use the *deterministic mode* (**-D**), so your tests will have an automatic & deterministic flow.

# 8 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 **source code** and **tests** in a jar file named 'hw3.jar'.

We will test your project by running:

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