

## SOEN 350 – OOP development

### Assignment 2 (10%)

---

<b>Due Date:</b>	By 11:59 pm May 10th, 2025
<b>Evaluation:</b>	10% of final mark ( <b>Late submission not allowed</b> )
<b>Purpose:</b>	The purpose of this assignment to practice object-oriented programming (polymorphism, encapsulation and UML)

#### Key Notes:

---

**NOTE1:** You are not allowed to post any assignment and/or its solution anywhere on the World Wide Web (WWW) or the internet. Intellectual Property Rights are reserved. If any case is discovered via your account or IP address; your submission will NOT be considered and will be reported immediately to the appropriate authority.

#### General Guidelines When Writing Programs:

---

Include the following comments at the top of your source codes

```
//-----  
// Lab (include number)  
// Written by: (include your name and student id)  
// For SES350 Section (your section) – Spring 2025  
//-----
```

- Write comment, give a general explanation of what your program does.
- As the programming questions get more complex, the explanations will get lengthier.
- Include comments in your program describing the main steps in your program. Focus your comments more on the ‘why’ than the ‘how’.
- Display a welcome message.
- Display clear prompts for users when you are expecting the user to enter data from the keyboard.
- All output should be displayed with clear messages and in an easy-to-read format.
- End your program with a closing message so that the user knows that the program has terminated.

#### Environment:

---

Please make sure that you have Maven and Java installed on your computer.

- OS: macOS/Windows 10
- Java: version 1.8+
- [Git](#)

## Introduction:

---

This assignment practices object-oriented programming. Note, based on lecture on version control (April-8<sup>th</sup>) your goal is to use Git to keep track of code changes. You decide how you want to commit your changes. However, commit must have [atomic](#) changes. You are also expected to add code comment using @JavaDoc. You **do not need** to add CheckStyle for this assignment. However, you **do** need to **screenshot** of your text-based user interface (refer to the submission guideline). You must name your new project **Assignment2\_{FullName}**. For more information refer to the submission guide at the end of the document.

## Teamwork is allowed

---

You can choose to work **alone** or with only **one** other person. However, you must use version control to track your code changes. If you collaborate with another student, only one submission is required. However, you must **indicate** the name your partner in the submission. I will review the version history to confirm equal contribution from both partners. Note, if you are working with a partner, you need to host it on Github so your code can be shared with your partner.

## Assignment: Dungeon Role-Playing Game:

---

Implement a role-playing game in which a player must find a **path** through a dungeon from a starting chamber to a goal chamber. The description of this task is intentionally vague to give you freedom in exploring object-oriented design and implementation.

The player can choose a **Character** (e.g., a wizard or warrior); every character has experience in terms of **strength** and **craft**, as well as **health** points (the player loses when the character health becomes 0). Characters can carry **Items** (e.g., a **sword**, a **shield**, or a **magic** wand) in an inventory, and designate up to two items as being in use (one per hand).

A **Dungeon** consists of **Chambers**; each chamber has one or several **Doors** that connect to other chambers (and connect back to the current chamber, i.e., the dungeon forms an undirected graph). A door may be guarded by a **Monster** (e.g., a **Goblin** or a **Spider**), which are adversarial characters.

As characters, they have health and either strength or craft (but unlike the player's character not both, e.g., if craft is non-zero then strength is 0). When the player enters a chamber, a list of possible **Actions** is presented (e.g., **Fight** a monster, **Pick** an item, or **Move** through a door to an adjacent chamber, which is only possible if the guarding monster of the door is defeated in a fight).

A fight is directed by the monster's skill: if the monster has strength (i.e., craft is 0), the fight uses the strength values of the player and the monster. Otherwise, if the monster is skilled in craft (strength 0), the fight uses the craft values.

In a fight, the player and the monster each roll a die (random `int` between 1 and 6) and add it to the strength/craft value, plus the strength or craft value of any item that the player carries in hand.

The difference gets subtracted from the health of the monster or player.

When the health of the monster drops to 0, the monster is defeated (gets removed from the door).

**Your assignment is to document, implement and test the game.**

**1. Task 1: UML**

- Document your software design with UML class diagram. You could use **draw.io** (search on Google) to help you document the UML.

**2. Task 2: All task finished and are correct**

- Use object-oriented programming principles to **encapsulate** data in classes, inheritance to factor **commonalities** into base classes, and **polymorphism** to choose the **concrete** behavior of methods depending on the dynamic type of objects at runtime (e.g., how items are printed).
- Demonstrate your game with user tests, taken as **screenshots** from the text UI outputs as game play evolves. You should submit this screen shot as well.

**3. Task 3: Testing**

- You must write **unit test** to ensure that your code behaves correctly (considering both happy and unhappy path).

The code snippet below serves as a partial specification of the API of your classes and it provides the text-based user interface. Your implementation **should** work with this code.

## Game Setup

```
public class Game {
    public static void main(String[] args) {
        // initialize some chambers
        Chamber[] chambers = new Chamber[] {
            new Chamber(),
            new Chamber(new Axe()),
            new Chamber(new Shield()),
            new Chamber(),
            new Chamber()
        };

        // connect the chambers with doors
        Door.connect(chambers[0], chambers[1]);
        Door.connect(chambers[1], chambers[2], new Monster("Goblin", 1, 0, 3));
        Door.connect(chambers[2], chambers[3], new Monster("Spider", 3, 0, 5));
        Door.connect(chambers[3], chambers[4]);

        Character player = new Wizard("Gandalf");

        // initialize the dungeon with player, entry chamber, and goal chamber
        Dungeon d = new Dungeon(player, chambers[0], chambers[4]);

        TextUI ui = new TextUI();
        ui.play(d);
    }
}
```

## Text-Based User Interface

The code snippet below defines the text-based user interface. It prints the current chamber, then prints all the available actions, and asks the user to select one (by inputting a number). Then the UI executes the action and loops.

- **Task: Address the TODO in print and print the monsters that are guarding doors**

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;

public class TextUI {
    public void play(Dungeon d) {
        while (!d.isFinished()) {
            print(d);
            Action a = ask(d);
            a.execute();
        }
    }
    /** Print the current room of the dungeon. */
    private void print(Dungeon d) {
        Chamber r = d.getCurrentChamber();
        StringBuilder s = new StringBuilder();
        s.append("You are in a chamber with " + r.getDoors().size() + " doors\n");
        s.append("There are " + r.getItems().size() + " items in the chamber\n");
        // TODO: print for each door which monster is there, how strong it is, how skilled in  craft, and how healthy
        System.out.println(s.toString());
    }
    /** Asks the user for an action. */
    private Action ask(Dungeon d) {
        StringBuilder s = new StringBuilder();
        s.append("Here are your options:\n");
        List<Action> actions = d.getActions();
        for (int i = 0; i < actions.size(); i++) {
            Action a = actions.get(i);
            s.append("\t" + i + ": " + a.toString() + "\n");
        }
        System.out.println(s.toString());

        // ask for action
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
        try {
            int command = Integer.parseInt(reader.readLine());
            return actions.get(command);
        } catch (IOException e) {
            return new PrintError(d, e);
        }
    }
}
```

### Hint:

---

Below is a list of potential Java files you can use to develop your code.

- Action.java
- Axe.java
- Chamber.java
- Character.java
- Door.java
- Dungeon.java
- Fight.java
- Game.java
- Item.java
- Monster.java
- Move.java
- Pick.java
- PrintError.java
- Shield.java
- TextUI.java
- Warrior.java
- Wizard.java

### Submission Guide:

---

1. Your project must follow IntelliJ's default Maven folder structure (check other guideline at the end of this assignment):

```
YourProjectFolder/  
├── src/  
│   ├── main/java/ {Your main source code here}  
│   └── test/java/  {Your test code here}  
└── target/         {Optional CheckStyle reports go here}
```

2. Your code must **compile**, test must **execute** and **pass** and must have screen shots. You will receive immediate **zero** if code **does not compile**.
3. You must make atomic commits.
4. You must add code comment in the form of JavaDoc. Refer to this [link](#).
5. Adding CheckStyle linter to your project is **optional**.
6. Your final commit should be called “***Ready for submission***”.

## Guideline for adding CheckStyle:

To add [CheckStyle \(linter\)](#) you need to first understand [Maven](#). Once you understand what Maven is, go to this link to understand what [pom.xml](#) is.

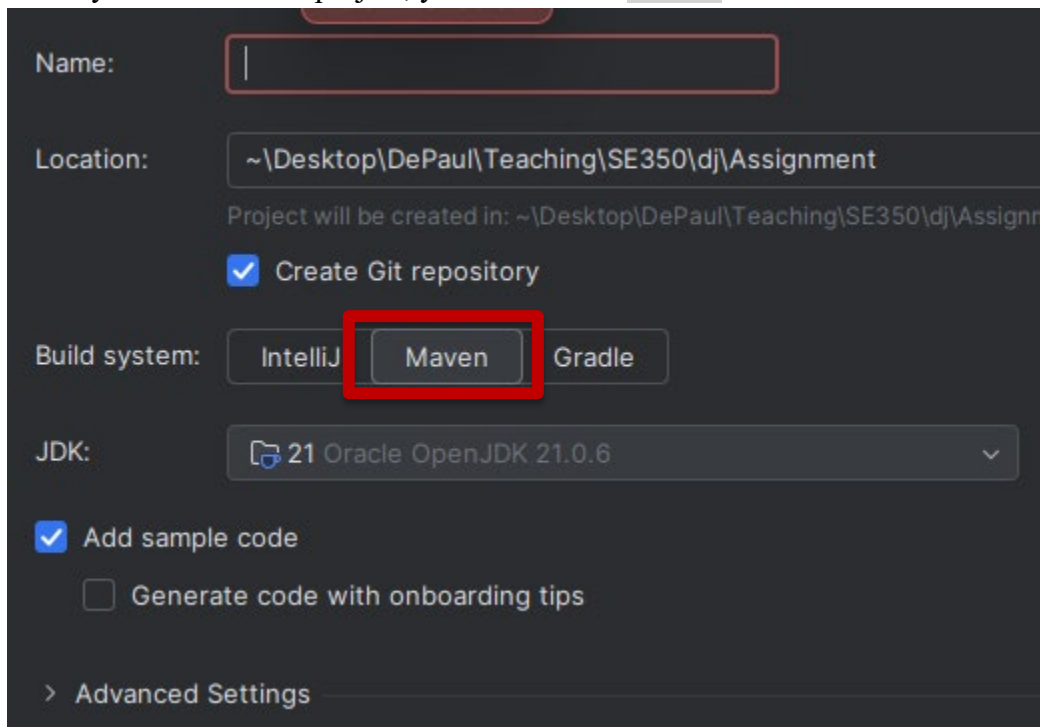
When incorporating [CheckStyle](#), you need to modify the build system (pom.xml) to output report when you run test cases. Refer to [link](#) to how to add CheckStyle in pom.xml

## Evaluation

Submission	Points
<b>Task completion:</b> all requirement implemented	<b>60%</b>
<b>Code correctness:</b> code is correct, demonstrated with unit tests	<b>15%</b>
<b>UML completion</b>	<b>10%</b>
<b>Style and Documentation:</b> Code is well-organized, follows conventions	<b>10%</b>
- <b>JavaDoc</b> for code comment	- 5%
- <b>Added screen shots</b>	- 5%
<b>Version Control:</b> Code is well version controlled	<b>5%</b>
<b>Total</b>	<b>100%</b>

Other guidelines:

When you create a new project, you must select **Maven**.



The screenshot shows the 'New Project' dialog in IntelliJ IDEA. The 'Build system' section has three buttons: 'IntelliJ', 'Maven', and 'Gradle'. The 'Maven' button is highlighted with a red rectangular box. Other visible fields include 'Name' (empty), 'Location' (set to a desktop path), 'Create Git repository' (checked), 'JDK' (set to '21 Oracle OpenJDK 21.0.6'), and 'Add sample code' (checked). There is also an unchecked option for 'Generate code with onboarding tips' and an 'Advanced Settings' link at the bottom.