Technical Report

Topic 8 FopValley

Tutorial Group : OCC 10

Group Name: ZEROBASEONE

Lecturer Name: Lim Chee Kau

Group Leader

DONG MING KANG (22121729)

Group Member

LONG SHUAI MING (22119548)

XIE JIA (23054220)

WANG YING QI (23050522)

1.0 Assigned Project

- 1.1 MapDesign
- 1.2 Creating Archetypes
  - 1.2.1 Archetypes.java
  - 1.2.2 Archer.java
  - 1.2.3 Mega.java
  - 1.2.4 Paladin.java
  - 1.2.5 Rogue.java
  - 1.2.6 Warrior.java
- 1.3 Design Monster
  - 1.3.1 Monster.java
  - 1.3.2 Goblin.java
  - 1.3.3 Harpy.java
  - 1.3.4 Orc.java
  - 1.3.5 Skeleton.java
- 1.4 Spells
  - 1.4.1 Ability.java
  - 1.4.2 SpellsService.java
  - 1.4.3 Using Spells.java
- 1.5 RoundBasedBattle.java
- 1.6 GameState.java
- 1.7 StoryLine.java
- 1.8 Game.java
- 1.9 app.java
- 1.10 BasicFunction.java
- 1.11 ReadSpellsUtil.java
- 1.12 Constants.java

2.0 Requirements of the project

- 2.1 Basic Part (8 marks)
  - 2.1.1 Map Design (0.5mark)
  - 2.1.2 Creating Archetypes (1 mark)
  - 2.1.3 Implement leveling-up System
  - 2.1.4 Design Monster (1 mark)
  - 2.1.5 Design Spells (2.5 marks)
  - 2.1.6 Round-Based Battle System(3 marks)
- 2.2 Extra Challenges(4 marks)
  - 2.2.1 Save Game and Load Game Function

3.0 Approach taken to solve the task

1.0 Assigned Project

This assignment requires you to design a text adventure game. The game should feature a 40x40 grid map with obstacles, five character classes with unique attributes, a leveling-up system, monsters and spells, and a round-based battle system. Additional features like game save functionality, handling abnormal inputs, colorful text display, database implementation, and ASCII art are also to be included. The focus is on using Object-Oriented Programming (OOP) principles, modularity, and collaboration tools like Git and GitHub.

1.1 MapDesign.java

This Java file outlines the `MapDesign` class for a text adventure game. It includes:

- Initialization of a 40x40 grid map with boundaries.
- Player placement at a random position on the map.
- Generation and placement of seven types of monsters at random locations.
- Player movement handling, allowing movement in four directions and updating the map accordingly.

- Interaction mechanics when the player encounters monsters.
- A function to check if the map has been cleared of monsters.

The class utilizes arrays to represent the game map and track occupied positions, ensuring dynamic interaction as the player navigates and encounters different elements within the game world.

1.2 Creating Archetypes

1.2.1 Archetypes.java

The `Archetypes` class in this Java file serves as a base class for different character types in a text adventure game. It includes attributes common to all characters, such as name, health points, mana points, defenses, and attacks. The class provides two constructors: one for initializing with specific values and a default constructor. It also includes a method `expCheck()` for experience management and `levelUp()` for character development. Additionally, there are getter and setter methods for each attribute, allowing for character customization and progression tracking. The `toString()` method provides a formatted string representation of the character's attributes.

1.2.2 Archer.java

The `Archer` class in this Java file extends the `Archetypes` class, primarily used for creating Archer characters in a text adventure game. The class has two constructors:

1. The default constructor reads the character's initial attributes (like health points, mana points, defenses, and attacks) from a file named `archetypes.txt`. It specifically targets the fifth line in the file, presumably where the Archer's attributes are defined.
2. The overloaded constructor allows for custom initialization of an Archer character with specified attributes.

Additionally, the class overrides the `levelUp` method from the parent class, providing a unique implementation for increasing the physical attack attribute of the Archer character upon leveling up. This reflects the character development aspect tailored to the Archer's archetype.

1.2.3 Mega.java

The `Mage` class, extending the `Archetypes` class, is designed for creating Mage characters in a text adventure game. It includes two constructors:

1. The default constructor reads Mage-specific attributes from a file, targeting the second line, where the Mage's attributes are presumably defined.
2. The overloaded constructor allows for initializing a Mage with specified attributes.

The class also overrides the `levelUp` method, providing a unique increase in magical attack and mana points for the Mage character upon leveling up, reflecting the Mage's focus on magical abilities.

1.2.4 Paladin.java

The `Paladin` class, derived from the `Archetypes` class, is tailored for creating Paladin characters in a text adventure game. It features two constructors:

1. The default constructor reads Paladin-specific attributes from a file, focusing on the fourth line, which likely contains the Paladin's initial attributes.
2. The overloaded constructor allows customization of a Paladin character with specific attributes.

The class overrides the `levelUp` method to uniquely increase both physical and magical attack stats for the Paladin upon leveling up, highlighting the Paladin's balanced approach to physical and magical combat.

1.2.5 Rogue.java

The `Rogue` class in Java, extending the `Archetypes` class, is designed for creating Rogue characters in a text adventure game. It includes two constructors:

1. The default constructor reads Rogue-specific attributes from a file, focusing on the third line, where the Rogue's attributes are presumably defined.
2. The overloaded constructor allows for the initialization of a Rogue with specified attributes.

This class also overrides the `levelUp` method, providing an increase in both physical attack and defense upon leveling up, aligning with the Rogue character's typical attributes in role-playing games.

1.2.6 Warrior.java

The `Warrior` class in Java, extending the `Archetypes` class, is designed for creating Warrior characters in a text adventure game. It has two constructors:

1. The default constructor reads Warrior-specific attributes from a file, targeting the first line, where the Warrior's attributes are presumably defined.
2. The overloaded constructor allows for custom initialization of a Warrior character with specified attributes.

This class overrides the `levelUp` method, providing an increase in both health points and defenses upon leveling up, highlighting the Warrior's role as a resilient and tough character in combat situations.

1.3 Design Monster

1.3.1 Monster.java

The `Monster` class in Java, part of the `testmonster` package, is designed for creating Monster characters in a text adventure game. This class includes:

- Attributes such as name, health points, mana points, physical and magical attacks, and defenses.
- A constructor for initializing these attributes.
- The `MonsterLvUp` method to increase the monster's attributes every time player's level up to 5 levels
- Also do some change on each monster's attributes(monsters.txt) for some values of monster are inappropriate.
- Getter and setter methods for each attribute, allowing for the modification and retrieval of the monster's characteristics.

This class provides a structure for defining the various monsters in the game, with attributes and behaviors that can be customized as needed.

1.3.2 Goblin.java

The `Goblin` class extends the `Monster` class, specifically representing a goblin creature in the game. It initializes the goblin with predefined attributes like health points, mana points, physical and magical attacks, and defenses. The class's constructor sets these attributes to reflect the typical characteristics of a goblin - relatively low health, modest physical attack, and weak defenses, embodying a creature that relies on numbers rather than individual strength.

1.3.3 Harpy.java

The Harpy class, derived from the Monster class, portrays a harpy creature in the game. This class defines the harpy with specific attributes, characterized by moderate health, a balance of physical and magical attacks, and average defenses. The constructor sets these attributes, depicting the harpy as a versatile and potentially challenging opponent, distinct from creatures like goblins that rely more on numbers than individual strength.

1.3.4 Orc.java

The Orc class, derived from the Monster class, represents an orc creature in the game. This class defines the orc with distinct attributes, typically characterized by higher health, stronger physical attacks, and robust defenses compared to creatures like goblins or harpies. The constructor of the Orc class sets these attributes, painting the orc as a formidable and physically imposing opponent, relying on individual strength and resilience in combat.

1.3.5 Witch.java

The Witch class, extending from the Monster class, represents a witch character in the game. This class defines the witch with unique attributes, typically highlighting a blend of magical capabilities and moderate physical traits. The constructor of the Witch class sets these attributes, illustrating the witch as a character who relies more on magical prowess and strategic abilities in combat, differentiating from physically dominant creatures like orcs.

1.3.6 Skeleton.java

The Skeleton class, derived from the Monster class, represents a skeleton creature in the game. This class defines the skeleton with specific attributes, typically characterized by lower health but potentially higher physical attack and defense, reflecting its nature as a reanimated creature. The constructor of the Skeleton class sets these attributes, portraying the skeleton as a creature that, while not as robust in health, may pose a threat through its physical resilience and attack capabilities.

1.4 Spells

1.4.1 Ability.java

The `Ability` class in the `testSpells` package is designed to represent a spell in a text adventure game. It includes attributes for the spell's name, level required to unlock it, mana cost, damage output, healing points, cooldown period, usage count, and a lock based on cooldown period. The class provides getters and setters for these attributes, allowing customization and tracking of each spell's properties. Additionally, it includes a `toString` method for a formatted representation of the spell's attributes, which is useful for debugging or displaying spell information to players.

1.4.2 SpellsService.java

The `SpellsService` class in the `testSpells` package provides functionality to manage spells in a game. Key methods include:

- `isAbility(Ability ability)`: Determines if a spell can be used based on its cooldown and total usage count.
- `unlockSpellsLevel(Ability ability, int level)`: Checks if a spell is unlocked at the player's current level.
- `isMpEnough(Ability ability, int MP)`: Verifies if the player has enough mana points to use a spell.
- `getRequiredSkill_MP`, `getDamage_HP`, `getSpellsHealthPoints`: Return the mana cost, damage, and healing points of a spell if it's unlocked at the player's level.

These methods collectively handle the spell's usability, cooldown, mana cost, and level requirements, crucial for gameplay dynamics involving spell casting.

1.4.3 UsingSpells.java

The `UsingSpells` class in the `testSpells` package manages the spell-casting mechanics in a text adventure game. It includes methods to display available spells and their statuses, and to handle the casting of spells during combat. The class utilizes a utility class `ReadSpellsUtil` to retrieve spell information and a `SpellsService` instance to check spell availability, cooldown status, and mana requirements. Methods are provided for different character classes (like Warrior, Mage, Rogue, Archer, and Paladin) to handle their unique spell lists and casting mechanics. The class ensures that spells are used according to game rules, such as level restrictions and mana costs.

1.5 RoundBasedBattle.java

The Java file is a comprehensive script for a round-based battle system in a text adventure game. It includes methods for handling player choices, attacking, defending, and healing mechanics in battles against monsters. The class is designed to cater to different player archetypes like Warriors, Archers, Mages, Paladins, and Rogues, each with unique battle strategies. The script manages the flow of the battle, player and monster health points, and the consequences of different actions, emphasizing the interactive and dynamic nature of the game's combat system.

1.6 GameState.java

The `GameState` class in the `fop_valley` package, extending the `Archetypes` class and implementing `Serializable`, manages the game state in a text adventure game. It includes an `Archetypes` object to store the player's archetype and attributes such as level, experience points, health points, mana points, defenses, and attacks. The class provides methods to save and load the game state using serialization, allowing the game's current state to be written to or read from a file. This functionality is crucial for enabling players to save their progress and resume the game later.

1.7 StoryLine.java

The storyline class integrates three methods for printing the storyline. The prologue is printed when the player enters the game, the second chapter is printed when the player is level 15, and the final chapter is printed when the player is level 35.

1.8 Game.java

The Game class is the basic running logic of this CLI game. It consists of three main parts: setting player object properties by choosing whether load the game save or start a new game and set player's character and name, printing maps, and conducting battles. Key methods include:

- `Void begin()`:Print game title using ASCII art and group's name on beginning page, instantiate the basic function class then call formatted method printBeginningMenu to print menu on this page to ask user to choose load previous game or start a new game.
- `Void startGame()`: If user choose to load game in begin method then the Boolean value characterSet and nameSet will be set as true, so that the game will continue to

run based on the data saved during previous game. If user choose to start a new game, the player will enter the stage of setting the character and name, and enter the game after the setting is completed.

After the game starts, the map will be printed and monsters will be generated. Players can move and fight. If all the monsters on the map are defeated, a new map will be generated.

- `Public testmonster.Monster checkWhichMonster(char m)`: This method receives a character parameter, and then uses this character to determine what monster the player encounters and returns the type of the monster for following turn-based battles.

1.9 app.java

The main program that used to run the game.

1.10 BasicFunction.java

The BasicFunction class integrates some basic functions such as formatting printing, clearing the screen, and checking input.

- `Public static int readChoice(String prompt, int playerChoice)`: This method is used to check the user's selection in the menu, and when the user inputs abnormally, a prompt will pop up asking the user to re-enter until the data is within the input range.
- `Public static int readChoice(String prompt, String[] playerChoice)`: The string array parameter received is the option. This method is used to check the user's selection during the battle to choose use starter or spells, and when the user inputs abnormally, a prompt will pop up asking the user to re-enter until the data is within the input range.
- `Public static void clearScreen()`:Output twenty blank lines by using a for loop to clear current screen.
- `Public static void lineSeperator()`:Print "+---------------+" as a line separator.
- `Public static void continueGame()`: Let the user enter any character to continue the game.
- `Public static void printHeading()`: Print the title to distinguish it from other in-game text.
- `Public GameState printBeginningMenu()`:Print the menu of the game start interface. If the user chooses to [2]load the game, return an object GameState used to load the game.
- `Public static void printInGameMenu()`:Print the in-game menu, similar to the pause function, which allows players to [2]view the current status or [3]save the game and exit.

- `Public static void printASCII(String scenario)`: The string parameter scenario receives a corresponding scene parameter and prints the corresponding ascii picture.

1.11 ReadSpellsUtil.java

The primary function of ReadSpellsUtil class is to read and manage ability data from given text file:`spells.txt`, and provide some methods to get necessary data . Key methods and their functions:

- `Private static Map<String,List<Ability>> readSpells()`:This method uses the fastjson i/o stream framework in the lib folder to improve development efficiency. It reads the file line by line, connecting each line into a string. Then splits this string into separating parts using a delimiter (defined in Constants.semicolon). Each part is further processed to extract a key-value pair, where the key is a character name, and the value is a list of Ability objects, parsed from JSON format.
- `Public static List<Ability> getAbilitysByArchetypesName(String name)`: It accesses the `nameAbilities` map from the result of readSpells() and returns the list of abilities corresponding to the provided archetypes' name.
- `Public static void printAbilityList(String name)`: It fetches the list of abilities from the `nameAbilities` map and prints each `Ability` object to the console. Only used for character choose.
- `Public static Ability getAbilityBySpellsName(String name,String spellsName)`: This method returns a corresponding specific skill by receiving an archetype's name and spell's name, which can be used in the usage of specific skills. The parameter spellsName comes from the string array parameter returned by the getSpellsNameArr method.
- `Public static String[] getSpellsNameArr()`:It iterates through all abilities in `nameAbilities` and collects their spell names into an array.

1.12 Constants.java

Store constants such as pathnames and delimiters to prevent abnormalities in project operation due to different devices.

2.0 Requirements

2.1.1 Map Design(0.5 mark)

Programmer: Wang Ying Qi

1. Design a 40x40 map consisting of the character # as obstacles where players cannot pass through.
2. Enable players can move their characters using the W, A, S, and D keys. W means upward, S means downward, A means left, and D means right.

2.1.2 Creating archetypes (1 marks)

Programmer:Dong Ming Kang

1. Create five archetype classes based on the information provided in archetypes.txt. This file contains the names of archetypes along with their initial attributes, including healthPoints, manaPoints, physicalDefense, magicalDefense, physicalAttack, and magicalAttack. When a player creates a new character, the character's initial attributes should be determined by the data in archetypes.txt.

1. You should apply your knowledge of File I/O to handle the archetypes.txt instead of hardcoding it. In other words, you should read the data from the file rather than copying and pasting it. No marks will be given if you have not demonstrated your File I/O knowledge. However, it's acceptable to hard code it after you have showcased your File I/O knowledge.

2. Archetypes class should include a method called levelUp().

2.1.3 Implement leveling-up system

Programmer:Dong Ming Kang, Wang Ying Qi

As characters progress on their journey, they have the ability to level up. Here are the general rules for character progression:During the first 10 levels, characters will require fewer experience points to level up, allowing for rapid early advancement.Each time a character levels up, their healthPoints, manaPoints, physicalAttack, magicalAttack, physicalDefense, and magicalDefense will increase.After reaching level 10, characters will level up at a slower pace.
The maximum attainable level for characters is set at 35.
The rate of attribute increase upon leveling up will vary based on the character's archetype:

Warrior: Health Points and defenses will receive significant boosts.
Mage: Magical Attack and Mana Points will experience substantial improvements. Rogue: Both Physical Attack and Physical Defense will be enhanced.
Paladin: All forms of Attack will become more potent.
Archer: Physical Attack will be substantially strengthened.

You are required to create a custom leveling-up algorithm for each archetypes.

2.1.4 Design Monster (1 marks)

Programmer:Dong Ming Kang, Wang Ying Qi

1. Design a Monster class that possesses attributes such as healthPoints, manaPoints, physicalDefense, magicalDefense, physicalAttack, and magicalAttack.

2. Design **5 unique monsters** that extend the Monster class based on description in monsters.txt

3. The locations of the monsters on the map are generated randomly.

4. Each time a player enters the game, 7 monsters of different types will be generated.

2.1.5 Design spells (2.5 marks)

Programmer:Long Shuai Ming, Wang Ying Qi

The heroes usually have their own spells. The spells could have special effect, say Silence, makes the opponent unable to cast a skill on your character; Perhaps FireBall, that is, merely cause damage to the opponent.

1. Create **3 spells** for each archetype.
2. These spells will only become accessible once the character reaches a specific level. *For instance, let's*

   consider a character named the "Warrior." The first spell, named "Roaring," will unlock and become

   usable once the character reaches level 5.
3. The spells have a cooldown period, meaning that a spell can only be used after a certain amount of time. You should integrate the concept of cooldown into the spell design.
4. All of the spells for each archetype, including their descriptions and cooldowns, are documented in spells.txt.
5. The monsters have the capability to replenish their manaPoints.
6. You are not restricted to follow the following design. It only serves as a reference.

2.1.6 Round-based Battle System (3 marks)

Programmer:Wang Ying Qi, Xie Jia

Now that we have archetypes, monsters, and spells in place, the next step is to implement the battle system. In a text-based RPG game, players are allowed to engage in round-based battles through interfaces.

1. In each round, players can make moves, which may include using spells or launching attacks.
2. The battle will only end when the player wins, loses, or chooses to exit the game. It's important to note that both the player and the monster can cast spells on each other.
3. Players have the option to attack, defend, heal, or escape. They are also allowed to use spells.
4. Your interface should provide informative descriptions that reflect the current situation. For instance,

   You have HIT the skeleton warrior, causing 151 damage!

5. Your interface should display the HP (Health Points) bars and MP (Mana Points) bars of both parties.

6. The battle system should be able to make the monster automatically battle with the player. They will attack or cast spells only, without the intention to defend.

2.2 Extra Challenges(4 marks)

For this part , we choose 4 additional challenges .

2.2.1 Save game functionality (1 mark)

It would be a great pity if someone couldn't save their game progress. Without the ability to save, they'd have to replay the same content repeatedly, which can become monotonous.

1. To enhance the player experience, it's crucial to provide a way for them to record their current game status so they can pick up their progress next time.

2. The game could remind them to save their progress before exiting, in case they forget to do so. Players would greatly appreciate this feature, as it ensures they don't lose their hard-earned progress.

3. Remember to leave a place for them to save the game.

Consider using a database and the knowledge you have acquired in File I/O for implementing this functionality.

2.2.2 Abnormal input Handling (0.5 mark)

Programmer: Wang Ying Qi

Mistakes can occur, whether unintentionally or intentionally. Therefore, our game should be capable of handling these abnormal inputs wisely and, perhaps, provide valuable feedback to inform the player about what's wrong with the input. It could be a typo or an unrecognized command.

2.2.3 Colorful text (0.5 mark)

Programmer: Wang Ying Qi

It would be more interesting if certain keywords could be displayed in different colors. For example, it would provide a clearer reminder if the materials required for upgrading the weapon are marked in green when they are sufficient and in red when they are insufficient. You can implement this feature easily by searching on Google or using ChatGPT. You might found some useful resources here.

2.2.4 ASCII Art (1 mark)

Programmer: Wang Ying Qi

Looking at text can be quite boring. Consider adding some ASCII art to make the game more attractive. You can refer to the following ASCII art. Your ASCII art should be stored in .txt file rather than hardcoding it into the code.

Feel free to incorporate any other features that you find interesting! Don't confine yourselves to the assignment question. Get as creative as you can, because there are no limits!

3.0 Approach Taken To Solve The Problem

3.1Map Design

1. Import Statements

```
import java.util.Random;
import java.util.Scanner;
import testmonster.*;
```

- The code imports the `Random` and `Scanner` classes for generating random numbers and obtaining user input.
- The import of the `testmonster` package suggests the existence of a package named `testmonster` containing classes related to monsters.

2. MapDesign Variables

```
public class MapDesign {
    private static final int MONSTER_TYPES = 7;
char[][] map = new char[40][40];
boolean[][] isOccupied = new boolean[40][40];
private int x, y, cur_x, cur_y;
private int monX, monY;
```

```
Random random = new Random();
Scanner scanner = new Scanner(System.in);
```

- `MONSTER_TYPES` defines the number of monster types.
- `map` is a 2D char array representing the map.
- `isOccupied` is a 2D boolean array indicating whether a position on the map is occupied.
- `x` and `y` are the current coordinates of the player.
- `cur_x` and `cur_y` are temporary coordinates after the player attempts to move.
- `monX` and `monY` are the coordinates of monsters.
- `Random` and `Scanner` are used for generating random numbers and taking user input, respectively.

3. Generate The Map

```
 public void generateMap() {
        for (int i = 0; i < 40; i++) {
            for (int j = 0; j < 40; j++) {
                if (i == 0 || i == 39 || j == 0 || j == 39)
                    map[i][j] = '#';
                else
                    map[i][j] = ' ';
            }
        }
    }
```

4. Bring Player

```
    public void bringPlayer() {
        x = random.nextInt(38) + 1;
        y = random.nextInt(38) + 1;
        map[x][y] = '@';
        isOccupied[x][y] = true;
    }
```

Places the player at a random position on the map, updating the map and occupied array.

5. Bring Monster And Generate Monster Method

```
    public void bringMonster() {
        while (true) {
            monX = random.nextInt(38) + 1;
            monY = random.nextInt(38) + 1;
            if (!isOccupied[monX][monY]) {
                isOccupied[monX][monY] = true;
                break;
            }
        }
```

```
    }

    public void generateMonster() {
        for (int i = 0; i < MONSTER_TYPES; i++) {
            int monsterType = random.nextInt(MONSTER_TYPES);
            switch (monsterType) {
                case 0:
                    bringMonster();
                    map[monX][monY] = 'W';
                    //generate a monster Witch
                    break;
                case 1:
                    bringMonster();
                    //generate a monster Goblin
                    map[monX][monY] = 'G';
                    break;
                case 2:
                    bringMonster();
                    //generate a monster Harpy
                    map[monX][monY] = 'H';
                    break;
                case 3:
                    bringMonster();
                    //generate a monster Orc
                    map[monX][monY] = 'O';
                    break;
                case 4:
                    bringMonster();
                    //generate a monster Skeleton
                    map[monX][monY] = 'S';
                    break;
                case 5:
                    bringMonster();
                    map[monX][monY] = 'W';
                    //generate a monster Witch
                    break;
                case 6:
                    bringMonster();
                    //generate a monster Goblin
                    map[monX][monY] = 'G';
                    break;
            }
        }
    }
```

- `bringMonster` places a monster at a random position on the map.
- `generateMonster` generates monsters of different types and places them on the map.

6. Print Map

```
    public void printMap() {
        for (char[] row : map) {
            for (char column : row)
                System.out.print(column + "   ");
            System.out.println();
        }
    }
```

7. Move Player System

```
    public char movePlayer() {
        cur_x = x;
        cur_y = y;
        boolean isMovementSuccessful = false;
        do{
            System.out.print("What's your next step?");
            System.out.println("-->");
            char dir = scanner.next().toUpperCase().charAt(0);
            switch (dir) {
                case 'W':
                    //actually the coordinate cannot be changed when
encounter the boundary
                    cur_x = Math.max((x - 1), 1);
                    isMovementSuccessful = true;
                    break;
                case 'S':
                    cur_x = Math.min(x + 1, 38);
                    isMovementSuccessful = true;
                    break;
                case 'A':
                    cur_y = Math.max(y - 1, 1);
                    isMovementSuccessful = true;
                    break;
                case 'D':
                    cur_y = Math.min(x + 1, 38);
                    isMovementSuccessful = true;
                    break;
                default:
                    System.out.println("Adventurer, you have gone too far.");
            }
        }while(!isMovementSuccessful);
        //check encounter
        if (map[cur_x][cur_y] != '#' && !isOccupied[cur_x][cur_y]) {
            //last position turn back to empty
            //update the map
            map[x][y] = ' ';
            isOccupied[x][y] = false;
            isOccupied[cur_x][cur_y] = true;
            //change the current coordinate
            x = cur_x;
```

```
        y = cur_y;
        //set the current position as '@'
        map[x][y] = '@';
        return ' ';
    }
    else if (map[cur_x][cur_y] != '#' && isOccupied[cur_x][cur_y]) {
        switch(map[cur_x][cur_y]) {
            //encounter witch
            case 'W':
                //last position turn back to empty
                //update the map
                map[x][y] = ' ';
                isOccupied[x][y] = false;
                isOccupied[cur_x][cur_y] = true;
                //change the current coordinate
                x = cur_x;
                y = cur_y;
                //set the current position as '@'
                map[x][y] = '@';
                return 'W';
            //encounter goblin
            case 'G':
                //last position turn back to empty
                //update the map
                map[x][y] = ' ';
                isOccupied[x][y] = false;
                isOccupied[cur_x][cur_y] = true;
                //change the current coordinate
                x = cur_x;
                y = cur_y;
                //set the current position as '@'
                map[x][y] = '@';
                return 'G';
            //encounter Orc
            case 'O':
                //last position turn back to empty
                //update the map
                map[x][y] = ' ';
                isOccupied[x][y] = false;
                isOccupied[cur_x][cur_y] = true;
                //change the current coordinate
                x = cur_x;
                y = cur_y;
                //set the current position as '@'
                map[x][y] = '@';
                return 'O';
            //encounter Skeleton
            case 'S':
                //last position turn back to empty
                //update the map
```

```
                map[x][y] = ' ';
                isOccupied[x][y] = false;
                isOccupied[cur_x][cur_y] = true;
                //change the current coordinate
                x = cur_x;
                y = cur_y;
                //set the current position as '@'
                map[x][y] = '@';
                return 'S';
            //encounter Harpy
            case 'H':
                //last position turn back to empty
                //update the map
                map[x][y] = ' ';
                isOccupied[x][y] = false;
                isOccupied[cur_x][cur_y] = true;
                //change the current coordinate
                x = cur_x;
                y = cur_y;
                //set the current position as '@'
                map[x][y] = '@';
                return 'H';
            default:
                //last position turn back to empty
                //update the map
                map[x][y] = ' ';
                isOccupied[x][y] = false;
                isOccupied[cur_x][cur_y] = true;
                //change the current coordinate
                x = cur_x;
                y = cur_y;
                //set the current position as '@'
                map[x][y] = '@';
                return ' ';
        }
    }
    else
        return ' ';
```

- **Player Movement:**
  - The method prompts the user for their next move and handles the input using a `do-while` loop.
  - The `cur_x` and `cur_y` variables store the tentative new coordinates based on the user's input.
- **Boundary Handling:**

- The code ensures that the player's coordinates (`cur_x` and `cur_y`) do not go beyond the boundaries of the map (1 to 38).

- **Encounter Checking:**
  - The method checks if the new position on the map involves encountering a wall (#) or an already occupied position.
- **Updating Map and Position:**
  - If the movement is successful, the player's position on the map is updated, and the previous position is cleared.
  - If there is an encounter with a monster or obstacle, it handles different cases based on the encountered entity.
- **Return Values:**
  - The method returns a character indicating the type of encounter ('W', 'G', 'O', 'S', 'H') or an empty character (' ') if there is no encounter.
- **Note:**
  - The `toUpperCase()` method is used to ensure case-insensitive input handling.

8. Set Occupied Method

```
        public void setOccupied(){
            isOccupied[x][y] = false;
        }
```

Marks the current player position as unoccupied.

9. curMapFinish Method

```
        public boolean curMapFinish() {
        int cnt = 0;
            for(int i = 1;i < 39;i++){
                for(int j =1;j < 39;j++){
                    if(isOccupied[i][j] == true)
                        cnt++;
                }
            }
            if(cnt == 1){
                return true;
            }
            else
                return false;
        }
    }
```

The purpose of the `curMapFinish()` method is to check if there is only one occupied position on the map, returning `true` if so, indicating that the map is considered finished, and `false` otherwise..

3.2 Creating 5 archetypes

1. Archetype

1. Attributes and Constructors

```
protected String name;
protected String playerName;
protected int healthPoints;
protected int showHP;
protected int manaPoints;
protected int showMP;
protected int physicalDefense;
protected int magicalDefense;
protected int physicalAttack;
protected int magicalAttack;
protected int level;
protected int xp;

public Archetypes(String name, String playerName, int healthPoints, int
manaPoints, int physicalDefense, int magicalDefense, int physicalAttack, int
magicalAttack) {
    // Parameterized constructor to initialize attributes
    this.name = name;
    this.playerName = playerName;
    this.healthPoints = healthPoints;
    this.showHP = healthPoints;
    this.manaPoints = manaPoints;
    this.showMP = manaPoints;
    this.physicalDefense = physicalDefense;
    this.magicalDefense = magicalDefense;
    this.physicalAttack = physicalAttack;
    this.magicalAttack = magicalAttack;
    this.level = 1;
    this.xp = 0;
}

public Archetypes() {
    // Default constructor
    this.showMP = manaPoints;
    this.showHP = healthPoints;
    this.level = 1;
    this.xp = 0;
}
```

- **Attributes:** The class contains attributes representing various characteristics of character archetypes.

- **Constructors:** Two constructors are provided - a parameterized constructor for initializing attributes with specific values and a default constructor that sets default values.

2. toString() Method

```
public String toString() {
    // Returns a string representation of the object
    return "Character Name: " + name + "\n" +
            "Player Name: " + playerName + "\n" +
            "HP: " + healthPoints + "\n" +
            "Magical Defense: " + magicalDefense + "\n" +
            "Physical Defense: " + physicalDefense + "\n" +
            "ManaPoints: " + manaPoints + "\n" +
            "Physical Attack: " + physicalAttack + "\n" +
            "Magical Attack: " + magicalAttack + "\n";
}
```

`toString()` Method: Overrides the `toString` method to provide a formatted string representation of the object.

3. Getter and Setter Methods

```
    public String getName() {

        return name;
    }

    public int getHealthPoints() {

        return healthPoints;
    }

    public int getManaPoints() {
        return manaPoints;
    }

    public int getShowHP() {
        return showHP;
    }

    public int getShowMP() {
        return showMP;
    }

    public int getPhysicalDefense() {
        return physicalDefense;
    }

    public int getMagicalDefense() {
        return magicalDefense;
```

```
}

public int getPhysicalAttack() {
    return physicalAttack;
}

public int getMagicalAttack() {
    return magicalAttack;
}

public int getLevel() {
    return level;
}

public int getXp() {
    return xp;
}

public void setName(String name) {
    this.name = name;
}

public void setHealthPoints(int healthPoints) {
    this.healthPoints = healthPoints;
}

public void setShowHP(int showHP) {
    this.showHP = showHP;
}

public void setManaPoints(int manaPoints) {
    this.manaPoints = manaPoints;
}

public void setShowMP(int showMP) {
    this.showMP = showMP;
}

public void setPhysicalDefense(int physicalDefense) {
    this.physicalDefense = physicalDefense;
}

public void setMagicalDefense(int magicalDefense) {
    this.magicalDefense = magicalDefense;
}

public void setPhysicalAttack(int physicalAttack) {
    this.physicalAttack = physicalAttack;
}
```

```
    public void setMagicalAttack(int magicalAttack) {
        this.magicalAttack = magicalAttack;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public void setXp(int xp) {
        this.xp = xp;
    }

    public String getPlayerName() {
        return playerName;
    }

    public void setPlayerName(String playerName) {
        this.playerName = playerName;
    }
}
```

4. Getter Methods

1. `public String getName()`: Returns the value of the `name` attribute (character's name).
2. `public int getHealthPoints()`: Returns the value of the `healthPoints` attribute.
3. `public int getManaPoints()`: Returns the value of the `manaPoints` attribute.
4. `public int getShowHP()`: Returns the value of the `showHP` attribute.
5. `public int getShowMP()`: Returns the value of the `showMP` attribute.
6. `public int getPhysicalDefense()`: Returns the value of the `physicalDefense` attribute.
7. `public int getMagicalDefense()`: Returns the value of the `magicalDefense` attribute.
8. `public int getPhysicalAttack()`: Returns the value of the `physicalAttack` attribute.
9. `public int getMagicalAttack()`: Returns the value of the `magicalAttack` attribute.
10. `public int getLevel()`: Returns the value of the `level` attribute.
11. `public int getXp()`: Returns the value of the `xp` attribute.

12. `public String getPlayerName()`: Returns the value of the `playerName` attribute.

5. Setter Methods

1. `public void setName(String name)`: Sets the value of the `name` attribute.
2. `public void setHealthPoints(int healthPoints)`: Sets the value of the `healthPoints` attribute.
3. `public void setShowHP(int showHP)`: Sets the value of the `showHP` attribute.
4. `public void setManaPoints(int manaPoints)`: Sets the value of the `manaPoints` attribute.
5. `public void setShowMP(int showMP)`: Sets the value of the `showMP` attribute.
6. `public void setPhysicalDefense(int physicalDefense)`: Sets the value of the `physicalDefense` attribute.
7. `public void setMagicalDefense(int magicalDefense)`: Sets the value of the `magicalDefense` attribute.
8. `public void setPhysicalAttack(int physicalAttack)`: Sets the value of the `physicalAttack` attribute.
9. `public void setMagicalAttack(int magicalAttack)`: Sets the value of the `magicalAttack` attribute.
10. `public void setLevel(int level)`: Sets the value of the `level` attribute.
11. `public void setXp(int xp)`: Sets the value of the `xp` attribute.
12. `public void setPlayerName(String playerName)`: Sets the value of the `playerName` attribute.

2. Archer

1. Constructor1

```
public Archer() {
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 5) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
```

```
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor reads data from the "archetypes.txt" file and initializes the attributes of the `Archer` object based on the content of the file.
- **File Reading:** It reads the file line by line and splits each line into parts using commas. If the line has exactly 7 parts and it is the 5th line (`L == 5`), it initializes the attributes accordingly.

2. Constructor2

```
public Archer(String name, String playerName, int healthPoints, int
manaPoints, int physicalDefense, int magicalDefense, int physicalAttack, int
magicalAttack) {
    super(name, playerName, healthPoints, manaPoints, physicalDefense,
magicalDefense, physicalAttack, magicalAttack);

    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 5) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
```

```
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor is similar to the first one, but it allows specifying initial values for the attributes when creating an `Archer` object.
- **File Reading:** It reads the file similarly to the first constructor and initializes the attributes based on the file content. The superclass constructor (`super(...)`) is used to initialize attributes common to all archetypes.

3. levelUpMethod

```
@Override
public void levelUp() {
    super.levelUp();
    physicalAttack += 10;
}
```

Purpose: Overrides the `levelUp` method from the superclass to customize the rate of attribute increase when an archer levels up. In this case, it increases the `physicalAttack` attribute by an additional 10 points.

3. Mage

1. Constructor 1

```
public Mage() {
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 2) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
```

```
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor reads data from the "archetypes.txt" file and initializes the attributes of the `Mage` object based on the content of the file.
- **File Reading:** It reads the file line by line and splits each line into parts using commas. If the line has exactly 7 parts and it is the 2nd line (`L == 2`), it initializes the attributes accordingly.

2. Constructor 2

```
public Mage(String name, String playerName, int healthPoints, int manaPoints,
int physicalDefense, int magicalDefense,
            int physicalAttack, int magicalAttack) {
    super(name, playerName, healthPoints, manaPoints, physicalDefense,
magicalDefense, physicalAttack, magicalAttack);
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 2) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor is similar to the first one, but it allows specifying initial values for the attributes when creating a `Mage` object.
- **File Reading:** It reads the file similarly to the first constructor and initializes the attributes based on the file content. The superclass constructor (`super(...)`) is used to initialize attributes common to all archetypes

3.levelUp Method

```
@Override
public void levelUp() {
    super.levelUp();
    magicalAttack += 10;
    showMP += 30;
    manaPoints += 30;
}
```

Purpose: Overrides the `levelUp` method from the superclass to provide specific behavior for mage characters during level-up. In this case, it increases the `magicalAttack` attribute by an additional 10 points and increases both `showMP` and `manaPoints` by 30 points.

4. Paladin

1. Constructor 1

```
public Paladin() {
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 4) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
```

```
    }
}
```

- **Purpose:** This constructor reads data from the "archetypes.txt" file and initializes the attributes of the `Paladin` object based on the content of the file.
- **File Reading:** It reads the file line by line and splits each line into parts using commas. If the line has exactly 7 parts and it is the 4th line (`L == 4`), it initializes the attributes accordingly.

2. Constructor 2

```
public Paladin(String name, String playerName, int healthPoints, int
manaPoints, int physicalDefense, int magicalDefense,
               int physicalAttack, int magicalAttack) {
    super(name, playerName, healthPoints, manaPoints, physicalDefense,
magicalDefense, physicalAttack, magicalAttack);
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 4) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor is similar to the first one, but it allows specifying initial values for the attributes when creating a `Paladin` object.
- **File Reading:** It reads the file similarly to the first constructor and initializes the attributes based on the file content. The superclass constructor (`super(...)`) is used to initialize attributes common to all archetypes.

3. levelUp Method

```
@Override
public void levelUp() {
    super.levelUp();
    physicalAttack += 10;
    magicalAttack += 10;
}
```

Purpose: Overrides the `levelUp` method from the superclass to provide specific behavior for paladin characters during level-up. In this case, it increases both `physicalAttack` and `magicalAttack` attributes by an additional 10 points.

5. Rogue

1. Constructor 1

```
public Rogue() {
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 3) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor reads data from the "archetypes.txt" file and initializes the attributes of the `Rogue` object based on the content of the file.
- **File Reading:** It reads the file line by line and splits each line into parts using commas. If the line has exactly 7 parts and it is the 3rd line (`L == 3`), it initializes the attributes accordingly.

2. Constructor 2

```
public Rogue(String name, String playerName, int healthPoints, int
manaPoints, int physicalDefense, int magicalDefense,
             int physicalAttack, int magicalAttack) {
    super(name, playerName, healthPoints, manaPoints, physicalDefense,
magicalDefense, physicalAttack, magicalAttack);
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 3) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor is similar to the first one, but it allows specifying initial values for the attributes when creating a `Rogue` object.
- **File Reading:** It reads the file similarly to the first constructor and initializes the attributes based on the file content. The superclass constructor (`super(...)`) is used to initialize attributes common to all archetypes.

3. `levelUp` Method

```
@Override
public void levelUp() {
    super.levelUp();
    physicalAttack += 10;
    physicalDefense += 10;
}
```

Purpose: Overrides the `levelUp` method from the superclass to provide specific behavior for rogue characters during level-up. In this case, it increases both `physicalAttack` and `physicalDefense` attributes by an additional 10 points.

6. Warrior

1. Constructor 1

```
public Warrior() {
    try (BufferedReader reader = new BufferedReader(new
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 1) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor reads data from the "archetypes.txt" file and initializes the attributes of the `Warrior` object based on the content of the file.
- **File Reading:** It reads the file line by line and splits each line into parts using commas. If the line has exactly 7 parts and it is the 1st line (`L == 1`), it initializes the attributes accordingly.

2.Constructor 2

```
public Warrior(String name, String playerName, int healthPoints, int
manaPoints, int physicalDefense, int magicalDefense,
               int physicalAttack, int magicalAttack) {
    super(name, playerName, healthPoints, manaPoints, physicalDefense,
magicalDefense, physicalAttack, magicalAttack);
    try (BufferedReader reader = new BufferedReader(new
```

```
FileReader("src/test/archetypes.txt"))) {
        String line;
        int L = 0;

        while ((line = reader.readLine()) != null) {
            L++;
            String[] parts = line.split(",");
            if (parts.length == 7 && L == 1) {
                name = parts[0];
                healthPoints = Integer.parseInt(parts[1]);
                manaPoints = Integer.parseInt(parts[2]);
                this.showMP = manaPoints;
                this.showHP = healthPoints;
                physicalDefense = Integer.parseInt(parts[3]);
                magicalDefense = Integer.parseInt(parts[4]);
                physicalAttack = Integer.parseInt(parts[5]);
                magicalAttack = Integer.parseInt(parts[6]);
            }
        }
    } catch (FileNotFoundException e) {
        System.out.println("Archetypes.txt was not found");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Error reading from Archetypes.txt");
    }
}
```

- **Purpose:** This constructor is similar to the first one but allows specifying initial values for the attributes when creating a `Warrior` object.
- **File Reading:** It reads the file similarly to the first constructor and initializes the attributes based on the file content. The superclass constructor (`super(...)`) is used to initialize attributes common to all archetypes.

3.`levelUp` Method

```
@Override
public void levelUp() {
    super.levelUp();
    healthPoints += 10;
    showHP += 10;
    magicalDefense += 10;
    physicalDefense += 10;
}
```

Purpose: Overrides the `levelUp` method from the superclass to provide specific behavior for warrior characters during level-up. In this case, it increases `healthPoints`, `showHP`, `magicalDefense`, and `physicalDefense` attributes by an additional 10 points each.

3.3 Design Monster

1. Monster

1.Constructor:

```
public Monster(String name, int healthPoints, int manaPoints, int
physicalAttack, int magicalAttack, int physicalDefense, int magicalDefense) {
    this.name = name;
    this.healthPoints = healthPoints;
    this.showHP = healthPoints;
    this.showMP = manaPoints;
    this.manaPoints = manaPoints;
    this.physicalAttack = physicalAttack;
    this.magicalAttack = magicalAttack;
    this.physicalDefense = physicalDefense;
    this.magicalDefense = magicalDefense;
}
```

Purpose: The constructor initializes a `Monster` object with specified attribute values.

2. `MonsterLvUp` Method

```
public void MonsterLvUp() {
    healthPoints += 70;
    showHP += 70;
    manaPoints += 50;
    showMP += 50;
    physicalDefense += 10;
    magicalDefense += 10;
    physicalAttack += 15;
    magicalAttack += 15;
}
```

- **Purpose:** This method represents the level-up process for the monster. It increases various attributes by fixed amounts.
- Attributes Increase:
  - `healthPoints` and `showHP`: Increased by 70.
  - `manaPoints` and `showMP`: Increased by 50.
  - `physicalDefense` and `magicalDefense`: Increased by 10.
  - `physicalAttack` and `magicalAttack`: Increased by 15.

2. 5 types Monster

1. Goblin

Goblins are small, pesky creatures that rely on sheer numbers to overpower their foes. They have no special abilities.

```
package testmonster;

public class Goblin extends Monster{
    public Goblin(){
        super("Goblin",40,0,10,0,5,2 );
        this.showHP = healthPoints;
        this.showMP = manaPoints;
    }
}
```

2. Harpy

Harpies are agile and cunning. They possess two abilities: "Talon Strike," a physical attack, and "Wind Gust," a magical attack that can disrupt opponents.

```
package testmonster;

public  class Harpy extends Monster {
    public Harpy(){
        super("Harpy",60,20,14,10,8,8 );
        this.showHP = healthPoints;
        this.showMP = manaPoints;
    }
}
```

3. Orc

Orcs are hulking, brutish creatures known for their raw strength. They have no unique abilities.

```
package testmonster;

public  class Orc extends Monster {
    public Orc(){
        super("Orc",70,0,15,0,9,4);
        this.showHP = healthPoints;
        this.showMP = manaPoints;
    }
}
```

4. Skeleton

Skeletons are reanimated undead, driven by malevolent intent. They have no unique abilities.

```
package testmonster;

public class Skeleton extends Monster{
    public Skeleton(){
        super("Skeleton",60,0,12,0,7,3);
        this.showHP = healthPoints;
```

```
        this.showMP = manaPoints;
    }
}
```

5. Witch

Witches possess magical abilities, including the power to cast spells. Their main ability is to cast "Fireball," a potent fire-based magical attack.

```
package testmonster;

public class Witch extends Monster {
    public Witch(){
        super("Witch",50,80,8,20,6,12);
        this.showHP = healthPoints;
        this.showMP = manaPoints;
    }
}
```

3.4 Design Spells

Three spells are provided for each archetype except Archer, including two attack skills and a skill to avoid attacks in the next round. And adjusted the values in the spells.txt.

3.5 Basic Function

This class contains some functions that are widely used in games. Key methods include:

Two readChoice methods with abnormal input handling:

```
public static int readChoice(String prompt,int playerChoice){
    int input;
    do{
        System.out.print(prompt);
        try{
            input = Integer.parseInt(scanner.next());
        }catch (Exception e){
            //abnormal input handling
            input = -1;
            System.out.println(ColorText.colorText("Please Enter a Number
Within the Given Range as Choice and Nothing Else.",ColorText.RED));
        }
    }while(input > playerChoice || input < 1);
    return input;
}
public static String readChoice(String prompt,String[] playerChoice){
    String input;
    boolean rightChoice = false;
    do{
        System.out.print(prompt);
```

```
        try{
            input = scanner.next();
            for(String choice : playerChoice){
                if(input.equalsIgnoreCase(choice))
                    rightChoice = true;
            }
        }catch (Exception e){
            //abnormal input handling
            input = null;
            //"Please Enter Your Choice Within the Given Range and Nothing
Else."
            System.out.println(ColorText.colorText("Please Enter Your
Choice Within the Given Range and Nothing Else.",ColorText.RED));
        }
    }while(!rightChoice);
    return input;
}
```

ClearScreen, lineSeperator, continueGame and printHeading to make this game look good.

```
public static void clearScreen(){
    for(int i = 0; i < 20; i++){
        System.out.println();
    }
}

public static void lineSeperator(){
    System.out.print("+");
    for(int i = 0; i < 30; i++){
        System.out.print(ColorText.colorText("-",ColorText.BLUE));
    }
    System.out.print("+\n");
}

public static void continueGame(){
    System.out.println(ColorText.colorText("Press Any Key to
Continue...",ColorText.BLUE));
    scanner.next();
}

public static void printHeading(String str){
    lineSeperator();
    System.out.println(ColorText.colorText(str,ColorText.CYAN));
    lineSeperator();
}
```

Print Menu on beginning page and in-game page, print ASCII art during game process.

```
public GameState printBeginningMenu(){
    printHeading("MENU");
    System.out.println("[1]Start a new Journey");
    System.out.println("[2]Load Game");
    System.out.println("[3]Exit");
    switch(readChoice("-->",3)){
        case 1:
            Game game = new Game();
            game.startGame();
            return null;
        case 2 :
            GameState gamestate = new GameState();
            gamestate.loadGame(gamestate);
            return gamestate;
        case 3 :
            System.exit(0);
        default:
            return null;
    }
}
public static void printInGameMenu(Archetypes player){
    printHeading("MENU");
    System.out.println("Choose an action:");
    System.out.println("[1] Continue Journey");
    System.out.println("[2] Current Status Of Your Character");
    System.out.println("[3] Save Data And Exit Game");
    switch (readChoice("-->",3)){
        case 1:
            return;
        case 2:
            player.expCheck();
            System.out.println(player);
            continueGame();
            return;
       case 3:
            GameState saveGameState = new GameState();
            saveGameState.saveData(player);
            saveGameState.saveGame(saveGameState);
            System.exit(0);
    }

}

public static void printASCII(String scenario){
    List<String> Arts = ASCIIart.addASCIIArt();
    if(scenario.equals("intro")){
```

```
System.out.println(ColorText.colorText(Arts.get(0),ColorText.BLUE));
    }
    else if(scenario.equals("Harpy")){

System.out.println(ColorText.colorText(Arts.get(1),ColorText.BLUE));
    }
    else if(scenario.equals("title")){

System.out.println(ColorText.colorText(Arts.get(2),ColorText.CYAN));
    }
    else if(scenario.equals("Goblin")){

System.out.println(ColorText.colorText(Arts.get(3),ColorText.BLUE));
    }
    else if(scenario.equals("Orc")){

System.out.println(ColorText.colorText(Arts.get(4),ColorText.BLUE));
    }
    else if(scenario.equals("Skeleton")){

System.out.println(ColorText.colorText(Arts.get(5),ColorText.BLUE));
    }
    else if(scenario.equals("Final")){
        System.out.println(Arts.get(6));
    }

}
```

3.6 Round-Based Battle System

3.6.1. readBattleChoice

```
public String readBattleChoice(){
        String[] battleChoice = {"S1","S2","S3","A1","A2","A3"};
        String choice = BasicFunctions.readChoice("-->",battleChoice);
        return choice.toUpperCase();
    }
```

This code corresponds player's choice to given options by calling the readChoice method of the BasicFunctions class. Then use toUpperCase() to unify the format, which is convenient to read the player's choice later.

3.6.2. StarterHealing

```
public boolean starterHealing(Archetypes player){
        int curMp = player.getManaPoints();
        if(curMp > 20){
            player.setManaPoints(curMp - 20);
            int curHp = player.getHealthPoints();
```

```
            if(curHp + 20 >= player.getShowHP()){
                player.setHealthPoints(player.getShowHP());
                System.out.println("Your Health Points Is Full.");
            }
            else {
                player.setHealthPoints(curHp + 20);
            }
            return true;
        }
        else
            return false;
    }
```

When the player chooses to heal, judge whether the player can use this skill through if conditional sentence. If he successfully launches, return true, and reduce manaPoints by 20, increase healthPoints attribute by an additional 20 points(can no longer be increased after being full

3.6.3 useStarter

```
public boolean useStarter(Archetypes player,Monster monster,String choice){
        switch (choice){
            case "S1" :
                PlayerAttack(player, monster);
                return true;
            case "S2":
                return true;
            case "S3":
                return starterHealing(player);
            default:
                //player chooses to use spells
                return false;
        }
    }
```

Make actions based on player choices through switch, and returns a boolean value to determine whether the player has ended the turn.

3.6.4. Battle System

```
public void battle(Warrior player,Monster monster){
    int roundCnt = 0;
    player.expCheck();
    int restLv = monster.MonsterLvUp(player.getLvMonsterLvUp());
    player.setLvMonsterLvUp(restLv);
    boolean isPlayerDodging = false;
    int leftDodgeRound = 0;
    do {
        roundCnt++;
        //player's turn
```

```
        showPlayerStatus(player);
        battleMenu();
        UsingSpells usp = new UsingSpells();
        usp.WarriorSpells(player);
        boolean isPlayerTurnEnd;
        boolean isPlayerDefending = false;
        do {
            String choice = readBattleChoice();
            isPlayerTurnEnd = useStarter(player, monster, choice);
            if(choice.equalsIgnoreCase("S2"))
                isPlayerDefending = true;

            if (!isPlayerTurnEnd) {
                switch (choice) {
                    case "A1":
                        isPlayerTurnEnd = usp.WarriorS1(player,
monster,roundCnt);
                        break;
                    case "A2":
                        if(usp.canUseDodgingSPells(player,
ReadSpellsUtil.getAbilitysBySpellsName(player.getName(),
UsingSpells.spellsName[1]),roundCnt)){
                            isPlayerDodging = true;
                            leftDodgeRound = 3;
                            isPlayerTurnEnd = true;
                            break;
                        }
                        else
                            break;
                    case "A3":
                        isPlayerTurnEnd =
usp.WarriorS3(player,monster,roundCnt);
                        break;

                }
            }
        }while(!isPlayerTurnEnd);
```

Before each round begins, the round counter is incremented by one. Then enter the battle, player character will make corresponding operations based on the input and determine whether the player operation is available. If not, the Boolean value isPlayerTurnEnd to determine whether the player's turn is over is still false, then re-enter the operation loop.

After the player makes action, determine whether the monster is dead and shows the monster's status. In the round when the monster attacks, if the player uses an escape spell, the monster cannot attack. And game over if the player's healthPoint drops below zero, otherwise the player can gain 20 experience points after defeating monsters.

```
 if(monster.getHealthPoints() <= 0){
            System.out.println(ColorText.colorText("\nYou Defeated the " +
monster.getName() + " ,Congratulations.",ColorText.CYAN));
            int curXP = player.getXp();
            player.setXp(curXP + 20);
            break;
        }
        //monster's turn
        showMonsterStatus(monster);
        if(!isPlayerDodging && !isPlayerDefending){
            MonsterAttack(player,monster);
        }
        else if(isPlayerDodging){
            System.out.println(ColorText.colorText("You dodged an
attack",ColorText.YELLOW));
            leftDodgeRound--;
            if(leftDodgeRound <= 0){
                isPlayerDodging = false;
            }
        }
        else if(isPlayerDefending){
            defendStatus(player,monster);
        }
        if(player.getHealthPoints() <= 0){
            System.out.println(ColorText.colorText("You DIED and GAME
OVER",ColorText.RED));
            System.out.println(ColorText.colorText("Press Any Button to
Quit.",ColorText.RED));
            BasicFunctions.continueGame();
            System.exit(0);
        }
    } while (player.getHealthPoints() > 0 && monster.getHealthPoints() >
0);
}
```

3.6.5. PlayerAttack

```
public void PlayerAttack(Archetypes player, Monster monster){
        int playerAtk = player.getPhysicalAttack() +
player.getMagicalAttack();
        int monsterDfs = monster.getMagicalDefense() +
monster.getPhysicalDefense();
        double monsterDmg = playerAtk -0.3 * monsterDfs;
        System.out.println("You have HIT the " + monster.getName() +
```

```
" ,causing " + (int)monsterDmg + " damage!");
        monster.setHealthPoints(monster.getHealthPoints() -
(int)monsterDmg);
}
```

Player attack and reduce the monster's health value by monster.setHealthPoints(), and provide informative descriptions that reflect the current situation.

3.6.6. MonsterAttack

```
public void MonsterAttack(Archetypes player, Monster monster){
    int monsterAtk = monster.getPhysicalAttack() +
monster.getMagicalAttack();
    int playerDfs = player.getMagicalDefense() +
player.getPhysicalDefense();
    double playerDmg = monsterAtk -  0.3 * playerDfs;
    if(playerDmg <= 0){
        player.setHealthPoints(player.getHealthPoints());
        System.out.println("You are strong enough, and the current
monsters can no longer harm you.");
    }
    else {
        System.out.println("> CRITICAL! " + monster.getName() + " has
SLASHED you for " + (int)playerDmg + " damage!");
        player.setHealthPoints(player.getHealthPoints() - (int)
playerDmg);
    }
}
```

Monster attack and reduce the player's health value by player.setHealthPoints(), and use System.out.println() to print the current situation.

3.6.7. Show Player Status

```
public void showPlayerStatus(Archetypes player) {
    System.out.println(player.getName());
    System.out.print("-->" + "HP: [");

    int HPratio = (int)(Math.ceil(player.getHealthPoints())/30);
    for (int i = 0; i < (int) Math.floor(player.getShowHP() / 30); i++) {
        System.out.print((i <= HPratio ? ":" : " "));
    }
    System.out.print("] " + "( " + player.getHealthPoints() + " / " +
player.getShowHP() + " )\n");

    System.out.print("-->" + "MP: [");
    int MPratio = (int)(Math.ceil(player.getManaPoints())/10);
    if (player.getManaPoints() != 0) {
        for (int i = 0; i < (int) Math.floor(player.getShowMP() / 10);
```

```
i++) {
            System.out.print((i <= MPratio ? "/" : " "));
        }

        System.out.print("] " + "( " + player.getManaPoints() + " / " +
player.getShowMP() + " )\n");
    }
    else {
        System.out.print("      ]\n");
    }
    BasicFunctions.lineSeperator();
}
```

Calculate the player's current HP and MP then display it.

3.6.8. Show Monster Status

```
public void showMonsterStatus(Monster monster) {
    System.out.println(monster.getName());
    System.out.print("-->" + "HP: [");

    int HPratio = (int)(Math.ceil(monster.getHealthPoints())/30);;
    for (int i = 0; i <(int)Math.floor(monster.getShowHP() / 30);i++){
        System.out.print((i <= HPratio ? ":" : " "));
    }
    System.out.print("] " + "( " + monster.getHealthPoints() + " / " +
monster.getShowHP() + " )\n");

    System.out.print("-->" + "MP: [");
    if(monster.getManaPoints() != 0){
        int MPratio = (int)(Math.ceil(monster.getManaPoints())/10);;
        for (int i = 0; i <(int)Math.floor(monster.getShowMP() / 10);i++){
            System.out.print((i <= MPratio ? "/" : " "));
        }
        System.out.print("] " + "( " + monster.getManaPoints() + " / " +
monster.getShowMP() + " )\n");
    }
    else {
        System.out.print("      ]\n");
    }
    BasicFunctions.lineSeperator();
}
```

Showing the damage suffered from the monster after using defence.

3.6.10. Battle Menu

```
public void battleMenu(){
        System.out.println(">> Starter");
        System.out.println("[S1] Attack");
        System.out.println("[S2] Defend");
```

```
        System.out.println("[S3] Heal");
    }
```

Prompt the player of the corresponding letters to the instruction.

3.7 Game Plot Sets

```
public static void printIntro(){
    BasicFunctions.printHeading("The beginning of the story");
    System.out.println("One day, you wake up to find your peaceful village
has been turned into ruins,");
    System.out.println("and the culprit behind all this is the sudden
appearance of monsters from another world.");
    System.out.println("Houses lie in rubble, and the villagers have
scattered, fleeing from the ominous presence of these otherworldly
creatures.");
    //TODO ASCII Art of Fire Here and Clear the console.
    System.out.println("After asking around, you learned that the
monsters' stronghold is located in a valley named FOP.");
    System.out.println("Amidst the chaos, you decide to face the challenge
head-on, embarking on a journey to explore this dangerous and mysterious
place.");
}
```

Print corresponding storylines at different levels of the player to enhance the fun of the game

3.8 Save / Load Game

3.8.1.Class Definition and Attributes

```
package fop_valley;

import test.*;
import java.io.*;

public class GameState extends Archetypes implements Serializable {
    protected int level;
    protected int xp;
    protected int healthPoints;
    protected int manaPoints;
    protected int physicalDefense;
    protected int magicalDefense;
    protected int physicalAttack;
    protected int magicalAttack;
    private Archetypes playerArchetype;
}
```

- The class `GameState` is declared in the package `fop_valley` and extends the `Archetypes` class while implementing the `Serializable` interface.

- It has several protected attributes representing different aspects of the game state (e.g., level, experience points, health points, etc.).
- There's a private attribute `playerArchetype` of type `Archetypes` representing the player's character archetype

3.8.2.Constructors

```
public GameState(Archetypes player) {
    super();
    this.playerArchetype = player;
}

public GameState() {
    // Default constructor, no explicit initialization.
}
```

- The first constructor is parameterized, taking an instance of `Archetypes` (`player`) and initializing the `playerArchetype` attribute.
- The second constructor is the default constructor with no parameters.

3.8.3. Getters and Setters

```
public Archetypes getPlayerArchetype() {
    return playerArchetype;
}

public void setPlayerArchetype(Archetypes playerArchetype) {
    this.playerArchetype = playerArchetype;
}
```

- Getter (`getPlayerArchetype()`) returns the player's archetype.
- Setter (`setPlayerArchetype()`) sets the player's archetype.

3.8.4. toString() Method

```
@Override
public String toString(){
    return "" + this.level +
            "\n" + this.xp +
            "\n" + this.healthPoints +
            "\n" + this.manaPoints +
            "\n" + this.physicalDefense +
            "\n" + this.magicalDefense +
            "\n" + this.physicalAttack +
            "\n" + this.magicalAttack;
}
```

Overrides the `toString()` method to provide a string representation of the `GameState` object.

3.8.5.Save Game

```
public void saveGame(GameState gameState){
    try (ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("src/fop_valley/saveGame.dat"))) {
        out.writeObject(gameState);
    } catch (IOException e) {
        System.out.println("Problems Occurred When Saving Current Game.");
    }
}
```

- The `saveGame` method is responsible for saving the game state to a file using object serialization.
- `ObjectOutputStream` is used to serialize the `GameState` object and write it to a file.
- The file path is specified as `"src/fop_valley/saveGame.dat"`. This path is relative to the "src" directory.
- The `try-with-resources` statement is used to automatically close the `ObjectOutputStream` after the try block.
- If an `IOException` occurs during the process, a message is printed indicating that problems occurred when saving the current game.

3.8.6. Load Game

```
public static GameState loadGame(){
    try (ObjectInputStream in = new ObjectInputStream(new
FileInputStream("src/fop_valley/saveGame.dat"))) {
        return (GameState) in.readObject();
    } catch (IOException e) {
        System.out.println("Problems Occurred When Reading Data.");
    } catch (ClassNotFoundException e) {
        System.out.println("You Dont Have Any Previous Save Yet.");
    }
    return null;
}
```

- The `loadGame` method is responsible for loading the game state from a file using object deserialization.
- `ObjectInputStream` is used to read the serialized `GameState` object from the file.
- The file path is specified as `"src/fop_valley/saveGame.dat"` (relative to the "src" directory).
- The `try-with-resources` statement is used to automatically close the `ObjectInputStream` after the try block.
- If an `IOException` occurs during the process, a message is printed indicating that problems occurred when reading data.

- If a `ClassNotFoundException` occurs, it means that the serialized class (`GameState` in this case) is not found. In this case, a message is printed indicating that there is no previous save yet.
- If everything is successful, the method returns the loaded `GameState` object.

3.9 Abnormal input Handling

```
public static int readChoice(String prompt,int playerChoice){
    int input;
    do{
        System.out.print(prompt);
        try{
            input = Integer.parseInt(scanner.next());
        }catch (Exception e){
            //abnormal input handling
            input = -1;
            System.out.println(ColorText.colorText("Please Enter a Number
Within the Given Range as Choice and Nothing Else.",ColorText.RED));
        }
    }while(input > playerChoice || input < 1);
    return input;
}
```

After reading the user input selection, determine whether it is an available selection, and if available, return the option. Check the input value is within the option range to determine whether the selection is completed; If not available, which means the input value is not within the option, the user will be required to re-enter the selection.

```
public static String readChoice(String prompt,String[] playerChoice){
    String input;
    boolean rightChoice = false;
    do{
        System.out.print(prompt);
        try{
            input = scanner.next();
            for(String choice : playerChoice){
                if(input.equalsIgnoreCase(choice))
                    rightChoice = true;
            }
        }catch (Exception e){
            //abnormal input handling
            input = null;
            //"Please Enter Your Choice Within the Given Range and Nothing
Else."
            System.out.println(ColorText.colorText("Please Enter Your
Choice Within the Given Range and Nothing Else.",ColorText.RED));
        }
```

```
    }while(!rightChoice);
    return input;
}
```

After reading the user input selection, determine whether it is an available selection, and if available, return the option; the Boolean value to determine whether the selection is completed is set to true. If not available, the Boolean value used to determine whether the selection is complete remains false and the user is required to re-enter the selection.

3.10 Colorful text

```
public class ColorText {
    //use ANSI escape codes for colored text
    //Each ANSI escape sequence starts with ESC(\u001B) means the
following characters are a control sequence
    //m means the end of control code and the number is color code
    public static final String RESET = "\u001B[0m";
    public static final String RED = "\u001B[31m";
    public static final String GREEN = "\u001B[32m";
    public static final String YELLOW = "\u001B[33m";
    public static final String BLUE = "\u001B[34m";
    public static final String PURPLE = "\u001B[35m";
    public static final String CYAN = "\u001B[96m";

    public static String colorText(String text, String colorCode){
        return colorCode + text + RESET;
    }
}
```

The class defines several String constants, each representing a different color code. Also define a String constant named RESET (\u001B[0m) means resets the color formatting to the default terminal color. And Each constant sets the text color as indicated by their names.

`public static String colorText(String text, String colorCode)`: It connects the given `colorCode` with the `text` and then appends the RESET code. This way the text can be printed in the specified color.

3.11 ASCII Art

```
public class ASCIIart {
    public static List<String> addASCIIArt(){
        List<String> asciiArts = new ArrayList<>();
        StringBuilder sb = new StringBuilder();
        String lineSeperator = System.lineSeparator();
        try(
                InputStream ips = new
```

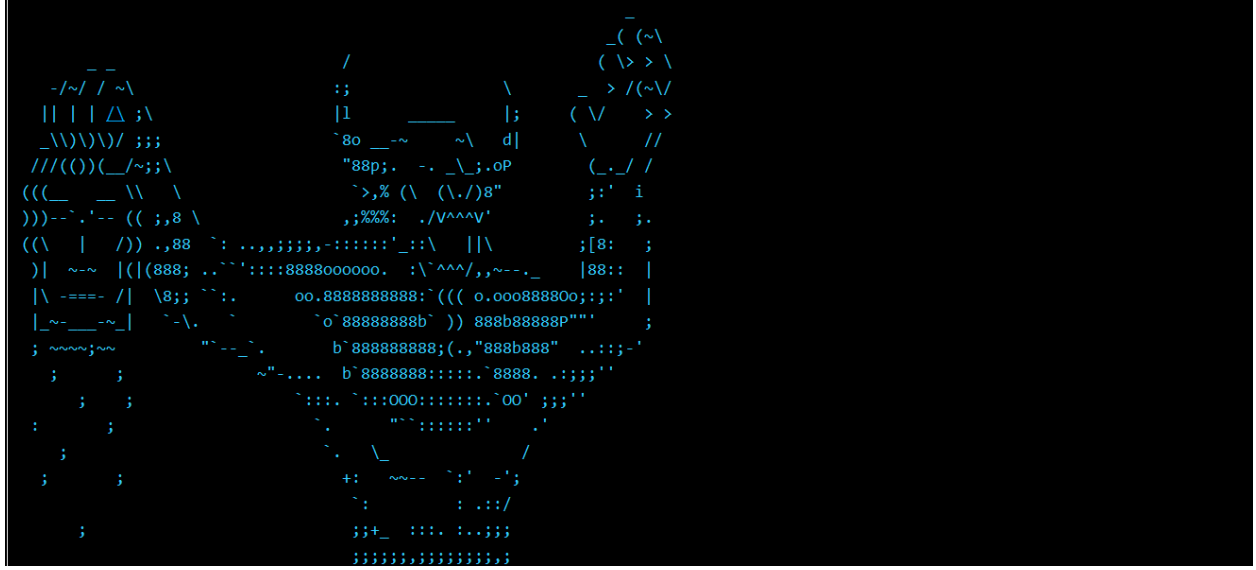
```
FileInputStream(Constants.ASCIIArtsPath);
                Reader isw = new InputStreamReader(ips);
                BufferedReader br = new BufferedReader(isw);
                ){
            String line;
            while ((line = br.readLine()) != null){
                if(line.equals(Constants.lineSeparatorBetweenArt)){
                    asciiArts.add(sb.toString());
                    sb = new StringBuilder();
                }
                else{
                    sb.append(line).append(lineSeperator);
                }
            }
            if(sb.length() > 0){//the last one hasnt been added
                asciiArts.add(sb.toString());
            }
        }catch (Exception e){
            System.out.println("Cannot Read ASCIIArts File.");
            e.printStackTrace();
        }
        return asciiArts;
    }
}
```

The ASCIIart class is to read ASCII art data from a file line by line and store it in a list.

4.0 Sample

Beginning Interface

```
The beginning of the story
+-----------------------------+
One day, you wake up to find your peaceful village has been turned into ruins,
and the culprit behind all this is the sudden appearance of monsters from another world.
Houses lie in rubble, and the villagers have scattered, fleeing from the ominous presence of these otherworldly creatures.
After asking around, you learned that the monsters' stronghold is located in a valley named FOP.
Amidst the chaos, you decide to face the challenge head-on, embarking on a journey to explore this dangerous and mysterious place.
```



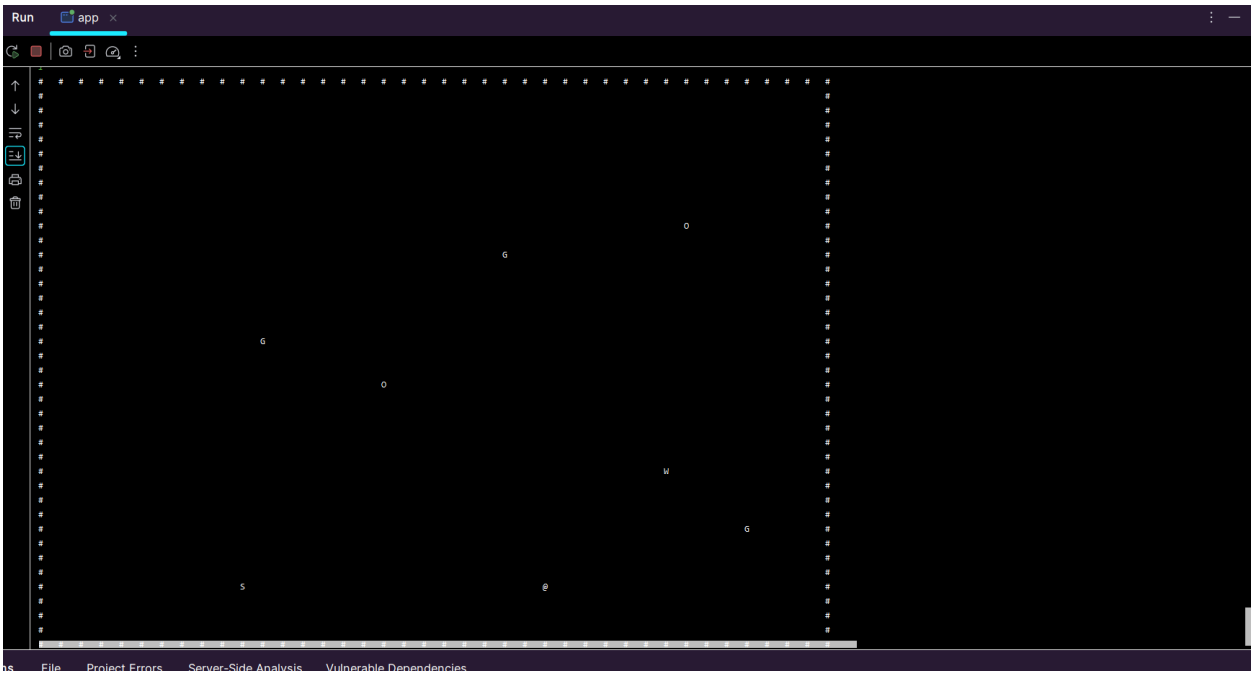
Print Intro

```
Spells Name: Divine Shield
Unlock Spells Level: 30
Required Skill MP: 20
Spells Damage: 0
Spells Healing HP: 0
CD: 6
Level Locked: false
Spells Description: Creates a protective barrier around the Paladin, rendering them immune to damage for 2 rounds


Spells Name: Dragon Slash
Unlock Spells Level: 30
Required Skill MP: 30
Spells Damage: 70
Spells Healing HP: 0
CD: 8
Level Locked: false
Spells Description: Use the Dragon Slash to slash enemies and deal heavy damage to them


+-----------------------------+
[5] Archer
Character Name: Archer
Player Name: null
Lv: 1
HP: 240
Magical Defense: 50
```

Show Archetype's Basic Attributes When Choosing

```
+------------------------------+
Is your character Paladin ?
+------------------------------+
[1] Yes!
[2] No I wanna change my character.
---> 1
+------------------------------+
Tell me your name:
+------------------------------+
沈泉锐
+------------------------------+
Is your name 沈泉锐 ?
+------------------------------+
[1] Yes!
[2] No I wanna change my name.
---> 2
+------------------------------+
Tell me your name:
+------------------------------+
ZhangHao
+------------------------------+
Is your name ZhangHao ?
+------------------------------+
[1] Yes!
[2] No I wanna change my name.
---> 1
Press Any Key to Continue...
```

Set Name



40 * 40 Map

```
Press " M " to open in game menu
m
+------------------------------+
MENU
+------------------------------+
Choose an action:
[1] Continue Journey
[2] Current Status Of Your Character
[3] Save Data And Exit Game
-->2
Character Name: Paladin
Player Name: ZhangHao
Lv: 15
HP: 560
Magical Defense: 200
Physical Defense: 210
ManaPoints: 170
Physical Attack: 360
Magical Attack: 340

Press Any Key to Continue...
```

Print In-Game Menu

```
#                                   S   @
#
#
#
#   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #   #
What's your next step?
-->a
Brace yourself,the skeleton has you in its sight...
       @@@@@@@@@@@@@@@@@@
     @@@@@@@@@@@@@@@@@@@@@@
   @@@@@@@@@@@@@@@@@@@@@@@@@@
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@
 @@@@@@@@@@@@@@@/      \@@@/   @
@@@@@@@@@@@@@@@\      @@  @___@
@@@@@@@@@@@@@ @@@@@@@@@@  | \@@@@@
@@@@@@@@@@@@@ @@@@@@@@@\__@_/@@@@@
 @@@@@@@@@@@@@@@/,/,/./'/_|.\'\,\
   @@@@@@@@@@@@@|  | | | | | | | |
                \_|_|_|_|_|_|_|_|

As you get stronger, the monsters get stronger too...
Paladin
-->HP: [::::::::::::::::::] ( 560 / 560 )
-->MP: [//////////////////] ( 170 / 170 )
+------------------------------+
>> Starter
[S1] Attack
[S2] Defend
[S3] Heal
```

Encounter Monster

```
What's your next step?
-->d
You startled the Witch!
As you get stronger, the monsters get stronger too...
Paladin
-->HP: [::::::::::::::::::] ( 560 / 560 )
-->MP: [//////////////////] ( 170 / 170 )
+------------------------------+
>> Starter
[S1] Attack
[S2] Defend
[S3] Heal
>> Spells
[A1] Holy Smite < -10MP, 45HP,  Smashes the target with divine light, dealing damage and healing the Paladin for a portion of the damage deal

[A2] <Locked - 30 >
[A3] <Locked - 30 >
-->d
-->d
-->d
-->a1
1/3 CD
-->a1
1/3 CD
-->a1
1/3 CD
-->
```

Monster Level Up And Show Spells Level Locked And Cool Down Count

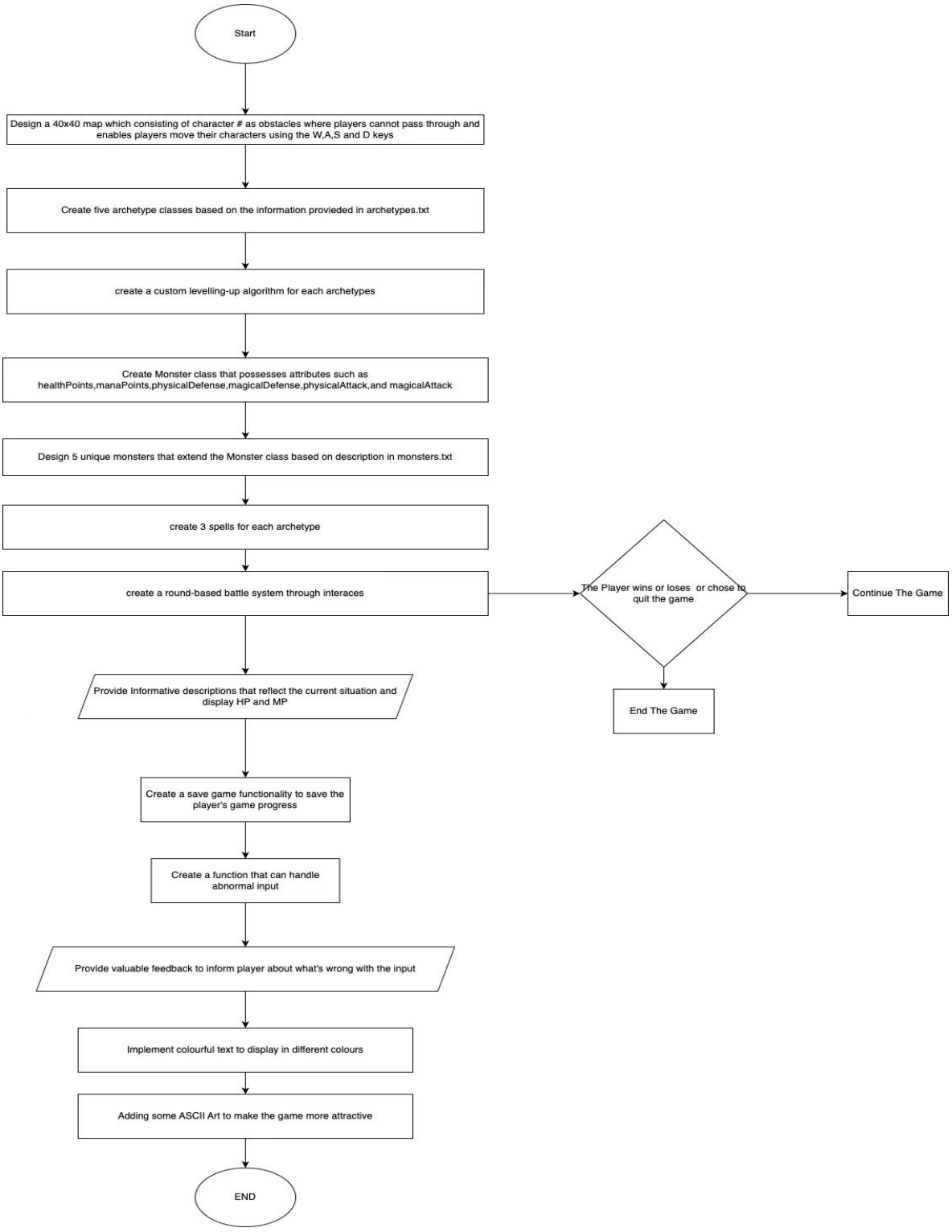
```
Spells Name: Storm of Blades
Unlock Spells Level: 33
Required Skill MP: 30
Spells Damage: 70
Spells Healing HP: 0
CD: 9
Level Locked: false
Spells Description: Unleash a storm of blades on your enemies, dealing heavy damage to them


Your character is --->1
+------------------------------+
Is your character Warrior ?
+------------------------------+
[1] Yes!
[2] No I wanna change my character.
---> aa
Please Enter a Number Within the Given Range as Choice and Nothing Else.
---> aa
Please Enter a Number Within the Given Range as Choice and Nothing Else.
--->
```

In-Game Abnormal Input Handling

```
+------------------------------+
Adventurer,welcome to
+------------------------------+
POP VALLEY

Made by Group ZEROBASEONE
Press Any Key to Continue...
l
+------------------------------+
MENU
+------------------------------+
[1]Start a new Journey
[2]Load Game
[3]Exit
-->2
Welcome back!
Press Any Key to Continue...
l
You've entered a whole new area...
Press " M " to open in game menu
m
+------------------------------+
MENU
+------------------------------+
Choose an action:
[1] Continue Journey
[2] Current Status Of Your Character
[3] Save Data And Exit Game
-->2
Character Name: Mage
Player Name: w
Lv: 1
HP: 200
Magical Defense: 100
Physical Defense: 30
ManaPoints: 150
Physical Attack: 20
Magical Attack: 30

Press Any Key to Continue...
```

Load Game And Save Game

5.0 Flow Chart



6.0 Modulus

The assignment for designing a text adventure game involves several key modules:

*6.1 **Serialization***

An interface we used for saving and loading game states.

6.2 File I/O

For reading and writing game data, like character attributes and game state.

6.3 Menu Navigation

Provides the player an interface for saving ,loading or quitting the game.

6.4 OOP

Creating 5 archetypes and different monsters make them inherited from the same class(Archetype and Monster) to get some different attributes.

6.5 NetBeans IDE

Used for developing, debugging, and managing Java projects.

6.6 Using Git and Github for Cooperation

During the whole cooperating period we chose to use git to push codes and update our codes on Github which makes we more convenient to manage the assignment