Project: Dungeons

Date: Dec 08, 2021

Written By: **Kaiyuan Wei**

Introduction

This document is the 2nd version of project design for the Dungeons Java program for CS5010 course at Khoury college of Computer Science.

System Overview

The project is to build a GUI base JAVA program to navigate in an unknown dungeon, pick the treasure (or arrow) and avoid (or kill) the monster if it's nearby. The game begins at a random spot within the dungeon and will finish when player arrive the destination spot or eaten by a monster. The program use Java SWING library to achieve then function of visualize the game and provide the capability of using mouse to navigate or perform actions. The detail of the object will be listed in the next section of this document.

Design Considerations

Goals and Guidelines

The base of game is a matrix like unknow dungeon (size: M by N or M by M). The color of a location is black if it's not visit by the player. The color will change after visit by the player. Yellow location indicates it's a cave (with 1, 3 or 4 exits). Gray color means it's a tunnel (only two exits are available). The spot where the player current stay will border with red color. The player can click on the neighboring cells or click on the direction button on the right hand side to move forward (if the direction is available in current position). When arrive a new place, the

color of cell will turn into the appropriate color and information about the spot will be display. Player can take appropriate action such as pick up stuff, shooting an arrow to a specific direction. The smell indication the distance of a monster is far or close. The result of shooting an arrow will be display beneath the pick and shoot button. The game will finish when player arrive the destination or eaten by a monster.

Properties of the dungeon:

- The dungeon is a 2-Dimension grid which will be display in the middle of the GUI.
- Player can allow move to the neighboring spot (if the direction is available).
- There will be arrows distribute randomly in both tunnels and caves. But treasure only hide in a cave not tunnel.
- A monster will only hide in a cave not in tunnel. If there is a monster in 2 spots away, there smell of current cell will be less pungent. If the monster is closer or there are multiple monster nearby, the smell will became pungent.
- The property of a dungeon could be setup only when a game is initiated. If the player wants to try another setting for a game. The user needs to turn off current program and start a brand-new game with new setting.
- The player can check the information about the dungeon on the menu bar.
- The player can choose to start a new game or resume the same game or quit the program at the first menu bar.

Constraints and rules of the game:

- The player will start a game with 3 arrows and no treasure at a random location.
- There will be three types of treasure: diamonds, rubies, and sapphires distribute in caves within the dungeon.
- The updated player state and the information about current place will be displayed on the left hand side of the GUI.
- The available direction for current spot, and the buttons of taking action are displayed on the right hand side of the Dungeon panel.

- When player want to pick up stuff in current place, click on the “pick” button and then select type of stuff to pick up. The “arrow” option will only collect the arrow, and the “treasure” option will collect any gem that available.
- If the player wants to shoot an arrow, click the “shoot” button. On the following message box input the distance (from 1 to 5) first and then input the direction (“n/e/s/w”) to shoot. The program will not accept invalid input for each step.
- The result of shooting will be display beneath the shoot button.

  “You shoot an arrow into the darkness” means the arrow hit nothing.

  “You hear a great howl in the distance” indicates the arrow hit a monster.

- It takes 2 hits to kill an Otyugh. Players has a 50% chance of escaping if the Otyugh if they enter a cave of an injured Otyugh that has been hit by a single crooked arrow.
- Distances must be exact. For example, if you shoot an arrow a distance of 3 to the east and the Otyugh is at a distance of 2, you miss the Otyugh.

Assumptions and Dependencies

The assumptions for implementation are:

1. the player will pass in the size of dungeon (m and n), the degree of interconnectivity (degree) and the percentage of treasure (percentage) before the program start to execute. This means the four parameters will be needed to start the game, like (int m, int n, int degree, double percentage).
2. The start and the end will be randomly chosen, but their distance should be no less than 5.
3. The options to move in a location should be from 1 – 4 (e, w, s, n). If the option is 2, the location should be classified as a tunnel, otherwise should be considered as a cave.
4. The locations on the edge will be randomly designed as “wrap” or “not wrapped”.
5. While the construction of dungeon, appropriated algorithm must be chosen to meet the demand of interconnectivity constraint (listed as point 3 on constraints of Dungeon).

Development Methods and Strategies

The program will adopt MVC design pattern to develop.

Model object will control the progress of the program

Controller object is responsible for communication between all the three classes.

View object will display the updated state of the game to player. For this project, the Java SWING library is used to achieve the functionality.

System Design

For this project, below interface and classes will be built during the implementation:

Package	Interface	Class	
dungeon	Location	Cave, Tunnel	
	Dungeon	DungeonImpl	
	Player	ThePlayer	
	Monster	Otyguh	
controller	Controller	GameController	
	CommandController	Move, Pick, Shoot, Start	
model	Model	TheModel	
	ViewOnlyModel	TheViewOnlyModel	TheViewOnlyModel is for testing only
view	View	TheView	
		Main	
		Driver	
helper		Kruskal, Gfg, Grid, Vertex (most for building the backbone of Dungeon)	

(UML is on the last page of the document)

Test Design

Below is part of design testing to make sure the method function ideally. (more tests might needed during implementation)

Test of **Controller** class	Input	Expected Value
test playTheGame() method	c.playTheGame(m); assertEquals(true, m.isGameOn());	pass

test player has 3 arrow at the beginner of a game	c.playTheGame(m);
	c.handleShoot(1,"n");
	c.handleShoot(1,"s");
	c.handleShoot(1,"e");
	
	assertTrue(c.handleShoot(1,"w").toString().toLowerCase().contains("out"));
		pass

test illegalPickup	@Test (expected = IllegalStateException.class)
	public void illegalPickup() {
	c.handlePickup("A Test");
	}
		pass

Test pickup an illegal item	@Test (expected = IllegalArgumentException.class)
	public void legalPickup() {
	c.handlePickup("arrow");
	}
		pass

test controller's start a new game	c.startNewGame();
	assertEquals(true, m.isGameOn());
	assertEquals(3, m.getArrowQty());
		pass

Test the normal Model class to achieve all sorts of functionality

Testing **Model** class	Input	Expected Value

test making a move	int before;
	int after;
	before = m.currentPosition().getIndex();
	try {
	m.move("s");
	}
		pass

Testing **Model** class	Input	Expected Value

	catch (Exception ex) {
	System.out.println("move failed");
	}
	after = m.currentPosition().getIndex();
	assertTrue(before < after);
	
	before = m.currentPosition().getIndex();
	try {
	m.move("n");
	}
	catch (Exception ex) {
	System.out.println("move failed");
	}
	after = m.currentPosition().getIndex();
	assertTrue(before > after);	

test arrive the end at the beginning	assertFalse(m.checkArriveEnd());	pass

test function of start a game	m.terminateCurGame();
	assertFalse(m.isGameOn());
	
	m.startAGame();
	assertTrue(m.isGameOn());
		pass

test restart a new game	try {
	m.move("s");
	}
	catch (Exception ex) {
	System.out.println("move failed");
	}
	try {
	m.move("n");
	}
	catch (Exception ex) {
	System.out.println("move failed");
	}
	//assert history greater than 1 in current game
	
	assertTrue(m.getPlayerHistory().size() > 1);
	
	m.freshNewGame();
	//assert the history is just 1 in the new game
		pass

Testing **Model** class	Input	Expected Value
	assertFalse(m.getPlayerHistory().size() > 1);	
Test the function of close a game	m.terminateCurGame(); assertFalse(m.isGameOn());	pass

Test the ViewOnlyModel, the model can read but can't change the state of a game

```
m = new TheModel(game);

m.startAGame();

vom = (ViewOnlyModel) m;
```

Testing **ViewOnlyModel** class	Input	Expected Value
test the model can read arrow quantity	assertEquals(3, vom.getArrowQty());	pass
test the model can read player position	assertEquals(Arrays.toString(Arrays.stream(m.getPlayerPos()).toArray()), Arrays.toString(Arrays.stream(vom.getPlayerPos()).toArray()));	pass
Test get the type of current position	assertEquals(m.inCaveNow(), vom.inCaveNow());	pass
Test get the correct Dungeon Size of a game.	assertEquals(m.getDungeonSize(), vom.getDungeonSize());	pass
Test get the correct InterConnectivity of a game.	assertEquals(m.getInterConnectivity(), vom.getInterConnectivity());	pass
Test get the correct Treasure Percentage of a game.	assertTrue(m.getTreasurePercentage() == vom.getTreasurePercentage());	pass
Test get the correct Monster	assertEquals(m.getMonsterQuantity(), vom.getMonsterQuantity());	pass

Testing **ViewOnlyModel** class	Input	Expected Value
Quantity of a game.		
Test get the wrapping setting of a game.	assertEquals(m.getWrappingState(), vom.getWrappingState());	pass
Test get the same history of a player.	assertEquals(m.getInterConnectivity(), vom.getInterConnectivity());	pass
Test get the same info of a player.	assertEquals(m.getThePlayerState(), vom.getThePlayerState()	pass
Test get the same endPoint of a game.	assertEquals(m.getGameEndPoint(), vom.getGameEndPoint()	pass

GUI Advanture Project UML

Kaiyuan Wei