CS1006

Practical 1: Hunt the Wumpus

This is a GROUP submission

Table of Contents

1 Requirements

This section aims to reduce the ambiguity of the original specification by explicitly defining our requirements for a successfully completed practical. The original specification was left intentionally vague in order for students to determine their own requirements and explore degrees of freedom. In line with this, these requirements have been considered in terms of the specification provided and our own interpretation. The main requirements that the *Practical 1: Hunt the Wumpus* project has completed to ensure a successfully completed practical are listed below.

1. **Create a basic deliverable that mimics the original text-based adventure game.** The original game included functions to move and shoot as well as hazards such as pits, bats and the wumpus which the user would try to avoid. Additionally, the original game also functioned in a command-line interface.

2. **Design the program in Java with Object-Oriented Programming (OOP) principles.** Using principles such as encapsulation, inheritance, polymorphism and abstraction wherever they are appropriate.

3. **Add additional beneficial functionality to improve the basic deliverable.** In this practical these additional enhancements to the program included but were not limited to randomising elements of the cave network and hazards, adding additional arrows, allowing the wumpus to move if the user misses and adding a Graphical User Interface (GUI).

4. **Test the program to ensure that the game is functional and to a user-ready standard.** Testing is done to ensure that the program can handle cases such as extreme, abnormal and normal inputs. In the testing section, this requirement is explained and evidenced in greater detail.

5. **Use of GitHub throughout the Practical to ensure peer contribution and version control.** As a project with more than one programmer, it is recommended we use a version control resource such as GitHub. This is important during an evolutionary development process where parts of a program are constantly being added and removed.

2 System Overview

2.1 Description of the Game System

This section aims to cover the notable differences between the original game and our implementation. In the original game, the wumpus cannot move[1] and hazards do not feature as much unpredictability as the version developed here. This has been implemented in this version to introduce a stochastic element to create uncertainty for the player. Additionally, the conditions for losing have been expanded. In the original version, the user loses if they are eaten, fall into a pit or if they run out of arrows. This version has been augmented so that when the user runs out of arrows they are still able to traverse the cave network to search for additional arrows. Furthermore, the Wumpus has a 75% chance of moving into a new room every time the user misses which is done to dissuade users from spam shooting arrows into every adjacent room without consequence. Despite the changes in the game, the main objective remains constant: the player's goal is to kill the Wumpus.

During development, we utilised principles of evolutionary prototyping to improve the efficiency and functionality of methods as development progressed as well as add features to improve the game's enjoyability. An important addition to the game which differs from the original would be the implementation of a Graphical User Interface (GUI) in the form of a map. The original game in BASIC works solely on a command-line interface. The addition of the GUI drastically aids the user in keeping track of their history of traversal. Lastly, a notable difference is the programming language as BASIC is generally considered to be procedura[2]l whereas Java is an Object-Oriented Programming (OOP) language. As a result, the development of the game was designed around the principles of classes, objects, inheritance, encapsulation, abstraction and to a degree polymorphism.

[1](Mann, 2024)
[2] (GeeksForGeeks, 2019)

3 Design

Initially, elements of the game such as wumpus, pit and bats were each a separate class. However, due to very few attributes and methods needed in each class, combining them under one class Hazards.java resulted in a more concise design.

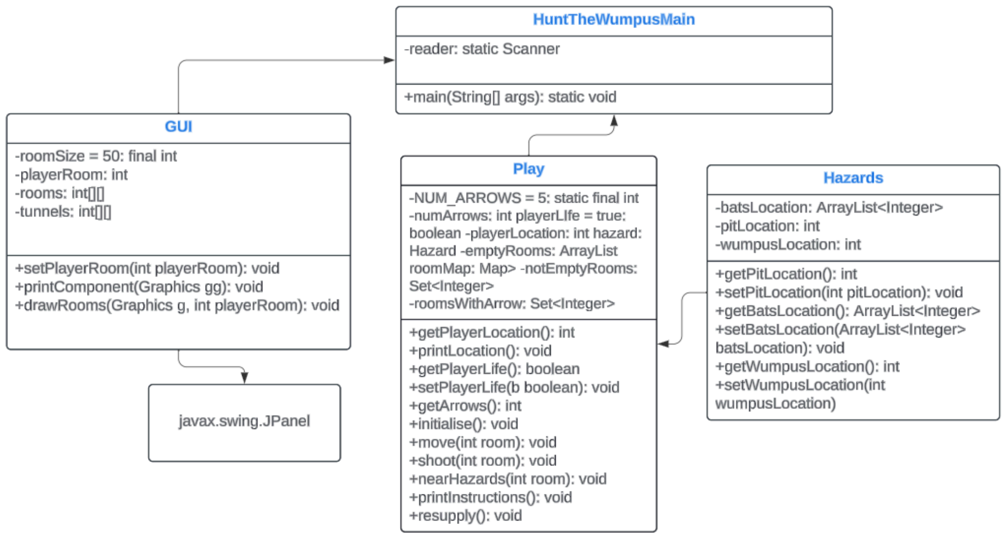
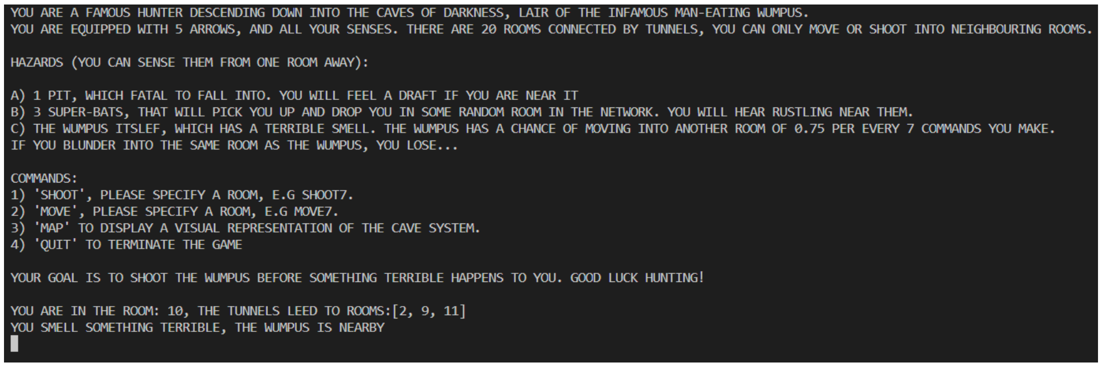

HuntTheWumpusMain
-reader: static Scanner
+main(String[] args): static void
GUI
-roomSize = 50: final int
-playerRoom: int
-rooms: int[][]
-tunnels: int[][]
+setPlayerRoom(int playerRoom): void
+printComponent(Graphics gg): void
+drawRooms(Graphics g, int playerRoom): void
javax.swing.JPanel
Play
-NUM_ARROWS = 5: static final int
-numArrows: int playerLIfe = true: boolean -playerLocation: int hazard: Hazard -emptyRooms: ArrayList roomMap: Map> -notEmptyRooms: Set<Integer>
-roomsWithArrow: Set<Integer>
+getPlayerLocation(): int
+printLocation(): void
+getPlayerLife(): boolean
+setPlayerLife(b boolean): void
+getArrows(): int
+initialise(): void
+move(int room): void
+shoot(int room): void
+nearHazards(int room): void
+printInstructions(): void
+resupply(): void
Hazards
-batsLocation: ArrayList<Integer>
-pitLocation: int
-wumpusLocation: int
+getPitLocation(): int
+setPitLocation(int pitLocation): void
+getBatsLocation(): ArrayList<Integer>
+setBatsLocation(ArrayList<Integer> batsLocation): void
+getWumpusLocation(): int
+setWumpusLocation(int wumpusLocation)

Figure 3: UML diagram

The class `Hazards.java` defines methods and attributes for each of the hazards that the methods in Play.java class use. `Play.java` includes the core methods of the game and attributes related to the player, which are discussed in more detail in the Implementation section of the report. Through creating an instance of the class `Play` in the main method in `HuntTheWumpusMain.java` and the use of a Scanner object, the user gets access to all the game functionality.

One of the most important aspects of the game design is the user experience, which the program tries to enhance by including very clear instructions as well as helpful text comments during the game including scenarios where the user's input is not appropriate. Additionally, a visual representation of the cave system (a flat dodecahedron with numbered nodes representing rooms) was included which is accessed by the user with the text command "MAP". The player is able to see the neighbouring rooms, their current location on the map (coloured in red) and all the rooms they have previously visited (coloured in orange) which could aid the user's understanding of the game as well as make the game more pleasant.



```
YOU ARE A FAMOUS HUNTER DESCENDING DOWN INTO THE CAVES OF DARKNESS, LAIR OF THE INFAMOUS MAN-EATING WUMPUS.
YOU ARE EQUIPPED WITH 5 ARROWS, AND ALL YOUR SENSES. THERE ARE 20 ROOMS CONNECTED BY TUNNELS, YOU CAN ONLY MOVE OR SHOOT INTO NEIGHBOURING ROOMS.

HAZARDS (YOU CAN SENSE THEM FROM ONE ROOM AWAY):

A) 1 PIT, WHICH FATAL TO FALL INTO. YOU WILL FEEL A DRAFT IF YOU ARE NEAR IT
B) 3 SUPER-BATS, THAT WILL PICK YOU UP AND DROP YOU IN SOME RANDOM ROOM IN THE NETWORK. YOU WILL HEAR RUSTLING NEAR THEM.
C) THE WUMPUS ITSLEF, WHICH HAS A TERRIBLE SMELL. THE WUMPUS HAS A CHANCE OF MOVING INTO ANOTHER ROOM OF 0.75 PER EVERY 7 COMMANDS YOU MAKE.
IF YOU BLUNDER INTO THE SAME ROOM AS THE WUMPUS, YOU LOSE...

COMMANDS:
1) 'SHOOT', PLEASE SPECIFY A ROOM, E.G SHOOT7.
2) 'MOVE', PLEASE SPECIFY A ROOM, E.G MOVE7.
3) 'MAP' TO DISPLAY A VISUAL REPRESENTATION OF THE CAVE SYSTEM.
4) 'QUIT' TO TERMINATE THE GAME

YOUR GOAL IS TO SHOOT THE WUMPUS BEFORE SOMETHING TERRIBLE HAPPENS TO YOU. GOOD LUCK HUNTING!

YOU ARE IN THE ROOM: 10, THE TUNNELS LEED TO ROOMS:[2, 9, 11]
YOU SMELL SOMETHING TERRIBLE, THE WUMPUS IS NEARBY
```

Figure 4: Instructions before each game

4 Implementation

4.1 Hazards.java

The attributes represent the location of each hazard `pitLocation` and `wumpusLocation` are of type integer since they hold only one value and `batsLocation` is an ArrayList since there are 3 bats in total in the cave network. There is a *get* method and a *set* method for each of the attributes in the class which are necessary to access and update the appropriate fields from the class Play.java.

4.2 Play.java

Attributes in this class mainly hold information about the player (e.g boolean playerLife), the arrows and the cave system. All the rooms are represented by a `HashMap<Integer,List<Integer>> roomMap`, where the keys are the room numbers (1 to 20), corresponding to a list of their neighbouring rooms (values). The cave network could have been also represented by a 2D array, but the choice of a map was made due to more efficient access to the keys and their values and the availability of helpful methods such as `contains()` and `get()`. Other examples of the program's implementation of the Collections class include an ArrayList `emptyRooms` and a Set `roomsWithArrow` that were useful in the random allocation of the game's elements.

`initialise()`

The method is executed every time a new game occurs.

- Inside the method the `roomMap` is populated by first manually creating a List of Integer Lists that represent the neighbouring rooms and then associating them with the rooms (keys) in the map.
- All the hazards (wumpus, bats, pit) are randomly assigned to rooms by utilising the `java.util` class Random and keeping track of the empty rooms.
- Finally a `resupply()` method is called which allocates 3 arrows in 3 random rooms throughout the network for the user to find while playing.

`nearHazards(int room)`

The method checks if there is a hazard (pit, wumpus or a bat) in any of the rooms next to the player and if so prints an appropriate message. It takes an integer parameter which is the number of the room the player is currently in to then get its corresponding value from the roomMap (the list of the neighbouring rooms) which will be stored in `neighbourRooms`.

- The command `neighbourRooms.contains(hazard.getWumpusLocation())` checks if the list of the neighbouring rooms to the one that the user is in contains the room that the wumpus is located in.
- To check if any of the rooms the bats are in the `neighbourRooms`, `Collections.disjoint()` is used which returns false if there is at least one common element in two lists.

`move(int room)`

This is the method to move the player from one room to another and it takes in the number of the room that the user inputted in the terminal.

- *Nested if-statements* are used that go through all the possible scenarios (the player walks into the room with the wumpus/bats/pit/arrow or an empty room) and update the player's attributes (`playerLife` and `playerLocation`) accordingly. For each case, a message is displayed to the user explaining what happened.

`shoot(int room)`

This method takes in the user input which is the number of a room and goes through every possible scenario with *if-statements* similar to `move()`.

- The number of arrows available to the player is reduced by 1.
- The player wins if the room is equal to the wumpus' location. If the player misses, the wumpus has a 75% chance of "hearing it" and moving to a new random room, which is an additional feature that was implemented to make the game more interesting.
- Another addition is the ability for the player to pick up arrows scattered around the network. Therefore, the player only loses if they are out of arrows and there are none left in the rooms.

`printPlayerLocation()`

Prints out a message to the user that specifies the room that the user is in and the list of neighbouring rooms that the tunnels lead to.

4.3 GUI.java

The class extends `JPanel` which is a part of the Java Swing package[3].

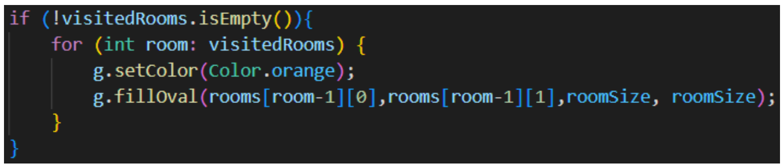
- The attributes include `roomSize` set to 50, `playerRoom` which is the player's current location and `visitedRooms` which is a set of integers representing the number of rooms that the player previously visited. The choice of a set was made to avoid iterating through the same elements when colouring the rooms (for instance if the player returned to the room they had previously visited).

[3] (Oracle, 2019)

- The cave system is an undirected graph with 20 nodes (rooms) connected by edges (each node is connected to 3 other nodes) resulting in a flat dodecahedron. All the rooms are represented by a 2D array, where each room holds specific x and y coordinates that are necessary for a valid visualisation of the dodecahedron. The tunnels to the neighbouring rooms are also represented by a 2D array, where each row is a room that holds an array of its neighbouring rooms. For instance, `tunnels[1]={2,5,8}`.
- The constructor takes in `playerRoom` and `visitedRooms`.

`drawRooms (Graphics g, int playerRoom, HashSet<Integer> visitedRooms)`

- The line connections between the rooms are drawn using methods of the class `Graphics` such as `setColor(),setStroke(),drawLine()` and a *nested for-loop* iterates through `tunnels[][]`and sets the appropriate coordinates for the links.
- The nodes (rooms) are created and coloured in green by iterating through `rooms[][]` and the command `g.fillOval(r[0],r[1],roomSize,roomSize()`. To make the nodes more vivid, a grey outer layer is created for them.
- The rooms were numbered, using a similar nested for-loop to the one above and the method `drawString().`
- Colouring the rooms the player has already visited was achieved by the command below:



```
if (!visitedRooms.isEmpty()){
    for (int room: visitedRooms) {
        g.setColor(Color.orange);
        g.fillOval(rooms[room-1][0],rooms[room-1][1],roomSize, roomSize);
    }
}
```

Figure 5: Colouring visited rooms as orange

- Finally, the room that the player is located in is coloured in red.

`@Overrride`
`paintComponent(Graphics g1)`

For the GUI to be properly implemented the method paintComponent in the class JPanel has to be overridden.

- Inside the `Graphics2D g` object is created and passed to the method `drawRooms()` alongside the parameters about the player's location and the visited rooms.
- To make the edges of the nodes smoother the command `g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON)` was used.

4.4 HuntTheWumpusMain.java

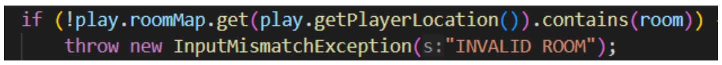
The main method consists of 3 *while loops*: the first one is the infinite game loop (for each game), the second one is for each player's move and the third one when the player decides if they want to play again after their game has ended.

Outer while loop

- A new Play object `play` is created, instructions are printed out and the method `play.initialise()` is executed. A GUI object `gui` is declared and the list of visited rooms (`visitedRooms`) is initialised.
- These commands will be executed every time the user starts a new game.

Inner while loops

- Before each player's move their location in the map is printed out along with the list of rooms they are allowed to go in. The `gui` object is updated by passing the current player's location and the rooms that they have visited.
- The user's input `line` is read from the command line using a Scanner object. Due to the fact that some user commands are only text such as "MAP" and others have the room number next to them ("SHOOT3"), the input line is split into a String component `command` and an integer component `room`. This was achieved by a command `replaceAll()` and regular expressions.
- Utilising a *switch statement*, enabled execution of specific game functionality and appropriate messages to the user based on the input command. There are 5 cases in total: "MAP", "AMMO" (to get the number of arrows the player has left), "MOVE", "SHOOT" and "QUIT".
- To handle invalid input from the user without the program breaking, a *try-catch statement* is implemented inside the loop. The program attempts to cover the most realistic scenarios of inappropriate input and consequently print out a message to the console that would guide the user. The program handles the input of an invalid text command, the absence of a room number next to "MOVE" and "SHOOT" and the input of an invalid room number. For instance, if the user tries to move or shoot not into a neighbouring room, the command below will be executed which has a matching catch statement at the end of the class.



```
if (!play.roomMap.get(play.getPlayerLocation()).contains(room))
    throw new InputMismatchException(s:"INVALID ROOM");
```

Figure 6: Invalid Room Condition

- To enhance the user's experience of the game, after the player loses or wins, they are asked if they want to play again. If yes, the program only breaks out of the inner loop to restart the game but if not, the program is terminated with the command `System.exit(0)` which will break out of the outer loop. The program is also terminated if at any point the player types "QUIT" into the command line.

5 Additional Functionality

5.1 Map GUI

The original game is a text-based adventure game, so as a consequence, it can become very difficult for the user to keep track of where they are within the cave system, especially due to the complexity of the dodecahedron graph system. To resolve this, the `GUI` class is called when the user chooses to view the map. The user can see their current room in red as well as previously visited rooms in yellow illustrated in Figure 7. The GUI is not meant to replace the text-based traversal system, but rather supplement the command-line experience. In turn, this quality of life game change may improve the user experience and enjoyment factor. By allowing the dodecahedron structure to be abstracted as a model, the overall complexity is reduced and code to be reused and referenced when referring to the cave system.

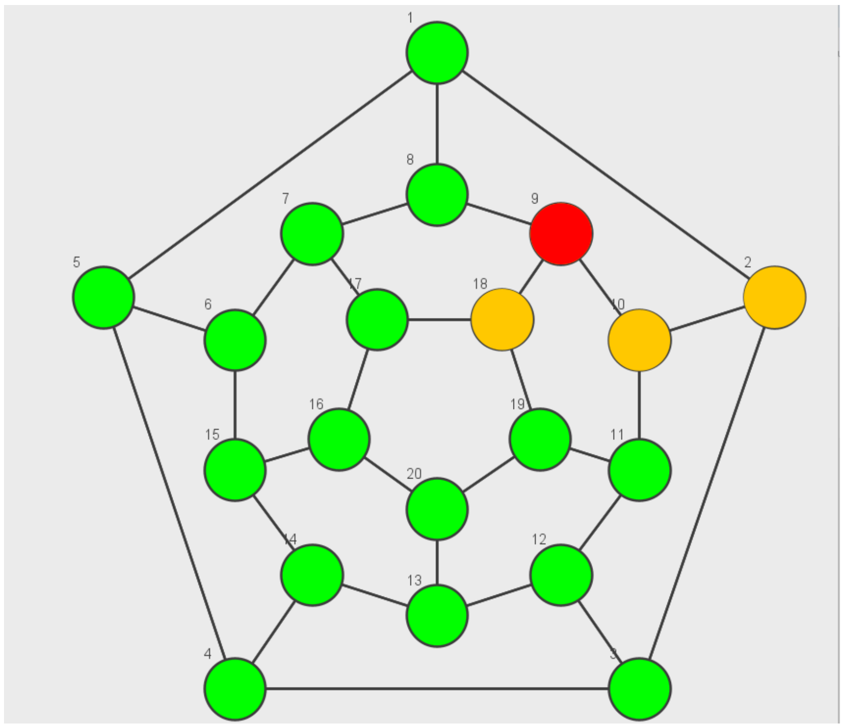


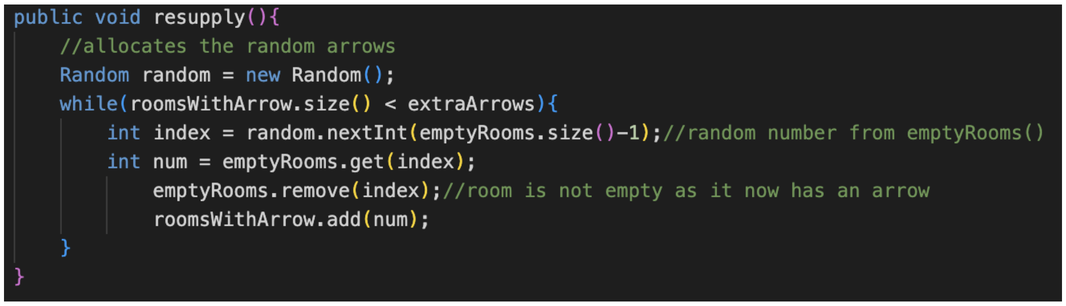
Figure 7: Dodecahedron representation of the cave system (the player is in room 9 and has previously visited rooms 2,10 and 18)

The map could have been further improved by making the numbers more readable which could be achieved by placing them in the middle of the nodes.

Even though the program provides a visual representation of the cave network, the game is text-based for the most part which could limit the player's experience. If we had more time, we would have tried to develop a fully functioning GUI, where instead of typing up the commands the user would navigate through the rooms using arrows on the screen.

5.2 Additional Arrow Functionality

To make the game more interesting for the player, Figure 8 demonstrates a function, `resupply()`, to randomly populate the set with non-duplicate rooms that contain arrows for the user to pick up. The randomisation works as it generates a value of the size of the `emptyRooms` set, representing an empty room, which is then added to a set of `roomsWithArrow` which contains only the number of arrows allowed to be scattered through the cave system. When a player enters a room the integer value is checked through the set. If the values match, the global variable `numArrows` increments and reduces every time the arrow is fired. One of the new losing conditions for the game is when `extraArrows` and `numArrows` are both zero and the wumpus is still alive. As a result, the game no longer ends if the user runs out of arrows in their quiver but ends if there are no more arrows in the cave system which is tracked through the global variable `numArrows`. This adds an unpredictable and more exciting nature to the game when arrows are allocated randomly across rooms within the cave system.

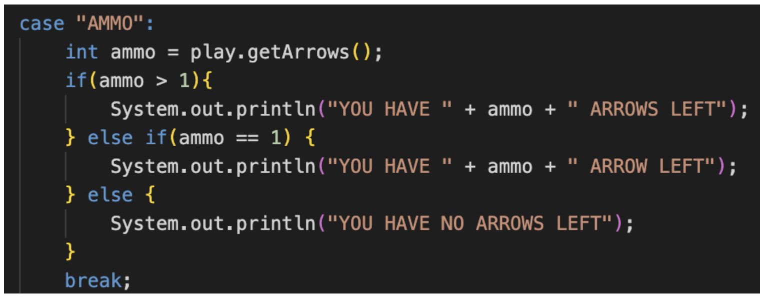


```java
public void resupply(){
    //allocates the random arrows
    Random random = new Random();
    while(roomsWithArrow.size() < extraArrows){
        int index = random.nextInt(emptyRooms.size()-1);//random number from emptyRooms()
        int num = emptyRooms.get(index);
            emptyRooms.remove(index);//room is not empty as it now has an arrow
            roomsWithArrow.add(num);
    }
}
```

Figure 8: Function to populate empty rooms with arrows

5.3 Display Arrows

The function to check the number of arrows in the quiver is a usability improvement that improves the fluidity and makes the game more user-friendly. During the initial playtests, it became cumbersome for the player to keep track of the number of arrows. This compounded when users could now pick up arrows, making it harder for users to keep track of their arrows due to the further dynamic nature of arrow availability. Unlike how hazards are intentionally difficult to keep track of, this was an unwanted challenge which introduced additional complexity. So although a relatively simple addition, it is one that drastically improves the user experience and gameplay enjoyability.

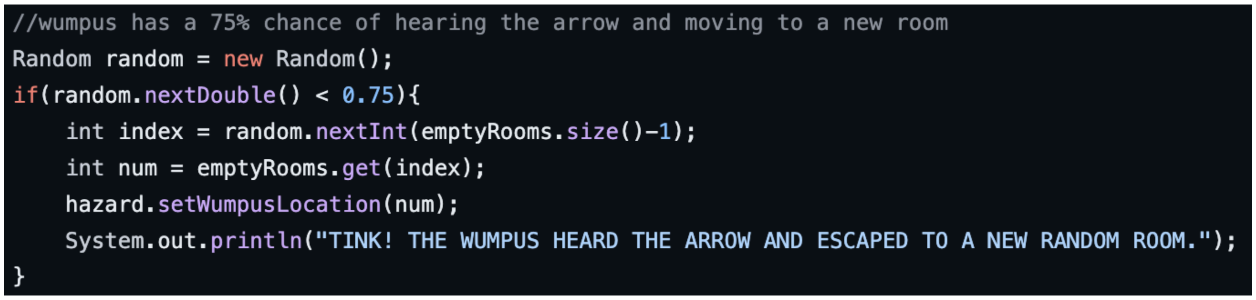


```java
case "AMMO":
    int ammo = play.getArrows();
    if(ammo > 1){
        System.out.println("YOU HAVE " + ammo + " ARROWS LEFT");
    } else if(ammo == 1) {
        System.out.println("YOU HAVE " + ammo + " ARROW LEFT");
    } else {
        System.out.println("YOU HAVE NO ARROWS LEFT");
    }
    break;
```

Figure 8: Switch statements for ammo

5.4 Randomisation for Wumpus

The base game that was created had no movement for the Wumpus. As a result, the game did not perhaps feel as dynamic and interactive as it possibly could. Additionally, with the extra arrows, the user could afford to waste more on adjacent rooms. To combat this, the Wumpus has a 75% chance of moving if the user misses, which introduces a stochastic element into the game similar to the allocation of random arrows. This not only makes the game more unpredictable but is also meant to dissuade users from shooting into adjacent rooms without properly strategising. This in turn increases the skill ceiling for the game making it more replayable.



```
//wumpus has a 75% chance of hearing the arrow and moving to a new room
Random random = new Random();
if(random.nextDouble() < 0.75){
    int index = random.nextInt(emptyRooms.size()-1);
    int num = emptyRooms.get(index);
    hazard.setWumpusLocation(num);
    System.out.println("TINK! THE WUMPUS HEARD THE ARROW AND ESCAPED TO A NEW RANDOM ROOM.");
}
```

Figure 9: Randomisation functionality for the wumpus

6 Testing

6.1 Manual Testing

In manual testing, the expected outputs and inputs of the game are recorded. Manual testing has limited scalability and test coverage but, it is relatively simple to test and offers the chance of exploratory testing[4]. This is when bugs not considered in development are dynamically uncovered as the program is developed. The majority of these tests were conducted after the program's development was finalised resulting in all passes. The results of the tests can be found in the appendix.

7 Critical Evaluation

7.1 Code Quality

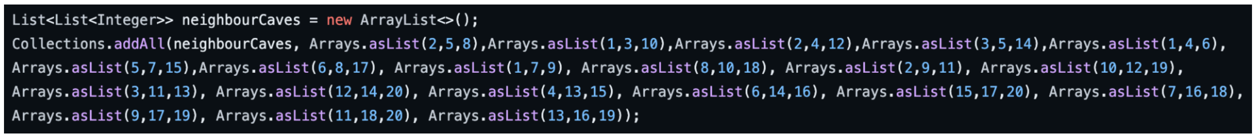
[4](Wikipedia, 2023)

Overall, the code is readable as class names, variables, methods and attributes are appropriately named. Additionally, comments were used throughout the design process and provided clarification wherever needed. An improvement, however, could be renaming the function `resupply()` to `addArrows()` for additional clarity. There are also instances where variable names such as `num` or `tempList` which could be more descriptive.

There are occurrences of redundancies where code is repeated when referring to the repeated instantiation of `random` objects in different methods. A solution would be to create a single instance which would allow for code reusability and less repetition. Another instance of repeated code is in the repeated logic of removing rooms in the `resupply()` and `initialise()` method. Here common functionality could be extracted into separate methods to reduce code repetition.

Regarding the use of data structures, each one was chosen due to its strengths and appropriateness. The use of sets such as `notEmptyRooms` and `roomsWithArrow` were chosen as opposed to arrays or lists. The primary reason is to ensure no overlap or repeated values when assigning hazards to rooms or assigning arrows to rooms. Additionally, the order of elements is unimportant as it does not matter what order the `roomsWithArrow` are in. In this section of the program, a set was the most optimal data structure.

Adding onto the choice of data structures, the representation of the dodecahedron was accomplished through a list of arrays `neighbourCaves`, as shown in Figure 10. It offers a very compact and flexible way of representing the connections of nodes in the cave system. We decided to choose this method over an adjacency matrix primarily because of space efficiency. As seen in Figure 10, each room is connected to only 3 other rooms. An adjacency matrix would have mostly no connections between nodes resulting in wasted space. However, an adjacency matrix may help visualise the connections between rooms better which may have streamlined other areas of development. Additionally, the simplicity and efficiency of traversal in a 2D array may have allowed for more additional features to be implemented.



```java
List<List<Integer>> neighbourCaves = new ArrayList<>();
Collections.addAll(neighbourCaves, Arrays.asList(2,5,8),Arrays.asList(1,3,10),Arrays.asList(2,4,12),Arrays.asList(3,5,14),Arrays.asList(1,4,6),
Arrays.asList(5,7,15),Arrays.asList(6,8,17), Arrays.asList(1,7,9), Arrays.asList(8,10,18), Arrays.asList(2,9,11), Arrays.asList(10,12,19),
Arrays.asList(3,11,13), Arrays.asList(12,14,20), Arrays.asList(4,13,15), Arrays.asList(6,14,16), Arrays.asList(15,17,20), Arrays.asList(7,16,18),
Arrays.asList(9,17,19), Arrays.asList(11,18,20), Arrays.asList(13,16,19));
```

Figure 10: List of Arrays to represent cave structure

Lastly, switch statements and *try-catch* blocks are useful constructs particularly when dealing with user input. They are better than a list of if statements and make logic easier to maintain as it is centralised in one place. *Try-catch* blocks gracefully handle the user input and deal with unchecked exceptions during runtime. However, a potential issue is the overuse when *try-catch* blocks carry a large scope in the program leading to debugging and diagnosing issues becoming harder to resolve as the program does not crash due to exceptions.

7.2 Testing

Manual testing is beneficial as developers can experience the user experience. This is helpful when designing elements such as ease of use, enjoyability and intuitiveness which are all aspects to consider when designing a game. However, manual testing has limited coverage and is prone to human error. JUnit tests are also automated and provide consistent and replicable results. On balance, both should be used to ensure a not only robust but enjoyable user experience. This could be a main area of improvement for the next practical.

8 Individual Feedback

8.1 Matriculation number: 230015013

The collaboration on the project was successful. Utilising GitHub and version control enabled us to work separately on different features and then merge the changes. On top of maintaining communication over text, we met up twice a week to exchange ideas and track the progress. The split was mostly equal. I worked on the class design, the choice of data structures, the core functionality in the Play.java and implemented the visual representation of the cave network (GUI). My partner focused on the randomasation of all the relevant elements in the game as well as all the additional functionality we included in the game. The main class was constructed together. Since I contributed slightly more to the code, my partner emphasised the report, the workload for which was divided approximately 60/40.

Overall, the practical was a positive experience which enhanced my skills in Java programming as well as taught valuable collaboration skills.

9 Appendix

Manual Testing Results

Test Number	Test Case	Expected Input	Expected Output	Result	Significance
1 Shooting the wumpus into an adjacent room when the wumpus is in that room.	Extreme: Valid Input	SHOOT1	“CONGRATULATIONS! YOU KILLED THE WUMPUS! YOU WIN.”	“CONGRATULATIONS! YOU KILLED THE WUMPUS! YOU WIN.”	Winning condition for the game.

2 Shooting the wumpus into an adjacent room when the Wumpus is not in that room.	Normal: Valid Input	SHOOT4	“YOU MISSED!”	“YOU MISSED!”	Sometimes, the result will include a message informing the user that the Wumpus heard the arrow and has moved.
3 Shooting the wumpus into a non-adjacent room.	Abnormal: Valid Input	SHOOT18	“INVALID ROOM”	“INVALID ROOM”	Important to indicate to the user that they are shooting into an invalid room.
4 Shooting an invalid number.	Extreme: Invalid Input	SHOOT21	“INVALID ROOM”	“INVALID ROOM”	Testing the boundaries of the program to ensure the edge cases are handled correctly.
5 Moving into an adjacent room.	Normal: Valid Input	MOVE2	“YOU ARE IN THE ROOM: 2, THE TUNNELS LEED TO ROOMS:[1, 3, 10]”	“YOU ARE IN THE ROOM: 2, THE TUNNELS LEED TO ROOMS:[1, 3, 10]”	Moving is a basic and essential function that needs to work.
6 Moving into your current room.	Abnormal: Invalid Input	MOVE1	“INVALID ROOM”	“INVALID ROOM”	The user should not be allowed to keep staying in their room.
7 Moving into a non-adjacent room.	Abnormal: Invalid Input	MOVE9	“INVALID ROOM”	“INVALID ROOM”	The user should only be allowed to move to connected rooms.
8 Uncapitalised user input for	Abnormal: Invalid Input	move2	“INVALID COMMAND, PLEASE TRY	“INVALID COMMAND, PLEASE TRY	Besides aesthetics, there is no

moving.			AGAIN”	AGAIN”	other reason for the user to need to type in capitals. In the future, this could be amended to allow non-capitalised characters.
9 Checking how many arrows are left when a user has 4.	Normal: Valid Input	AMMO	“YOU HAVE 4 ARROWS LEFT”	“YOU HAVE 4 ARROWS LEFT”	It is handy for the user to see how much ammo they have left. Plurality is correct.
10 Checking how many arrows are left when a user has 1.	Normal: Valid Input	AMMO	“YOU HAVE 1 ARROW LEFT”	“YOU HAVE 1 ARROW LEFT”	Plurality is also correct here.
11 Shooting a non-adjacent room.	Abnormal: Invalid Input	SHOOT3	“INVALID ROOM”	“INVALID ROOM”	Testing to see if a user can shoot into any room.
12 Checking what happens when the user runs out of ammo but there are still arrows in the cave system.	Normal: Valid Input	SHOOT1	“YOU HAVE RUN OUT OF ARROWS BUT CAN PICK UP ARROWS LEFT BEHIND BY FALLEN HUNTERS. THERE ARE 3 IN THE CAVE SYSTEM FOR YOU TO FIND.”	“YOU HAVE RUN OUT OF ARROWS BUT CAN PICK UP ARROWS LEFT BEHIND BY FALLEN HUNTERS. THERE ARE 3 IN THE CAVE SYSTEM FOR YOU TO FIND.”	Although this is mentioned in the rules, a helpful reminder is useful to inform the user of this information.
13 Shooting the last arrow	Normal: Valid Input	SHOOT1	“OH NO! YOU RAN OUT OF	“OH NO! YOU RAN OUT OF	There is no way to kill the wumpus after

when there are no more arrows left in a cave.			ARROWS! YOU LOSE”	ARROWS! YOU LOSE”	this point.
14 Walking into the same room as the wumpus.	Normal: Valid Input	MOVE3	“YOU LOST! THE WUMPUS ATE YOU! BETTER LUCK NEXT TIME”	“YOU LOST! THE WUMPUS ATE YOU! BETTER LUCK NEXT TIME”	This is a losing condition which must work.
15 Walking into a room with a bat.	Normal: Valid Input	MOVE15	“OH NO! YOU GOT PICKED UP BY BATS AND DROPPED INTO A NEW ROOM”	“OH NO! YOU GOT PICKED UP BY BATS AND DROPPED INTO A NEW ROOM”	Bats drop the user into a new empty room.
16 Walking into a room with a pit.	Normal: Valid Input	MOVE8	“AAHHHHHH... YOU FELL INTO A PIT! BETTER LUCK NEXT TIME HEHE”	“AAHHHHHH... YOU FELL INTO A PIT! BETTER LUCK NEXT TIME HEHE”	The user dies if they fall into a pit and lose the game.
17 Finding an arrow	Normal: Valid Input	MOVE12	“YOU FOUND AN ARROW”	“YOU FOUND AN ARROW”	If the user finds an arrow they are informed.

10 Bibliography

Mann, D.P. (2024). CS1006 Module Introduction & Practical 1. [online] studres.cs.st-andrews.ac.uk. Available at: https://studres.cs.st-andrews.ac.uk/CS1006/Lectures/CS1006_L1_Wumpus.pdf [Accessed 13 Feb. 2024].

GeeksForGeeks (2019). *Differences between Procedural and Object Oriented Programming - GeeksforGeeks*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/.

Oracle (2019). Trail: Creating a GUI With JFC/Swing (The JavaTM Tutorials). [online] Oracle.com. Available at: https://docs.oracle.com/javase/tutorial/uiswing/.

Wikipedia. (2023). *Exploratory testing*. [online] Available at: https://en.wikipedia.org/wiki/Exploratory_testing [Accessed 13 Feb. 2024].