HUNT THE WUMPUS

Tristin Doherty & Eugene Kong

Contents

1. Analysis

1.1 Overview

In this practical assignment, we were given the task of programming the text-based game, Hunt the Wumpus.

Our final submission achieves all parts of the specification such as generating a cave system, spawning entities, interaction between entities, movement of entities, and player controls such as moving/shooting towards these entities, and successfully implements the class components of the program to follow our decided game rules

1.2 Feasibility

With available resources like git hub we were able to collaborate independently, alongside the resources provided such as the joint labs in John Honey and Jack Cole to implement the code, therefore everything in the specifications were deemed feasible in terms of capacity to cooperate as a team effectively and finish the project.

1.3 Scope and boundaries

We decided that our scope would include:

- A fixed map with 20 rooms that were interconnected with each other three rooms each, following the structure of a dodecahedron
- Three hazards (pit, bats, and Wumpus) that would interact with the player.
- Player to entity interactions would include slaying the Wumpus, being transported, or dying to either bat or Wumpus.
- Player would be given the two options move or shoot, and then specify where.
- The player would be provided 5 arrows, and if all were expended, would lose.

And we decided that the definite boundaries within time constraints would be

- The game would be strictly text based
- The map would not be generated
- Quantity and behaviours of hazards would be fixed (bats within the range of 3-4)

2. Design

2.1 Design Decisions

We used a total of six classes for this assignment – a main class, a class for the rooms of the cave system, a player class, a Wumpus class, a bat class, and a pit class. To create a cave system that is logically similar to a collapsed dodecahedron, with every room being adjacent to three other room, we decided to use a HashMap, where each key (the room number, which is an int data type) will be mapped to its corresponding room object (the room's properties such as status and linked rooms), which we used as a wrapper class.

In our main class (HuntWumpus), we initialized our cave system using a method in the Room class, which generated a fixed HashMap that was populated with 20 key-value pairs, each representing a room on our cave system map. Following this, we then spawned in our Wumpus, pit, bats, and player, ensuring that the room in which we spawned each object was empty. HuntWumpus class also forms the basis for our actual game, which takes in the input that the user keys into the terminal. The user will continue keying in their movement/shooting and room number choices until one of the game over conditions are met.

There was a general idea that all classes should mainly interact with each other through the room class so that confusion between class interactions were minimised which minimised code disruption if an error was found.

In our Room class, apart from generating the HashMap to represent our cave system, we also included many boolean attributes to tell our program if the room currently houses the Wumpus, the pit, any bats, or is in the vicinity of any of those hazards. Our Room is thus able to determine if any game over conditions are met using these boolean values to see if the player has entered a room with a pit or Wumpus.

For our Player class, we decided to make it static since there will only be a single player on the map. We provided attributes for the player position and the number of arrows that the player has left. Apart from the standard accessor methods, we also created methods to spawn the player, move the player, and allow the player to shoot. We also used a separate method to verify the room number that the user entered, ensuring that it is adjacent to the player's current room.

For our Wumpus class, which was arguably the most complex hazard to program, we also decided to make it static since there would only be a single Wumpus in the entire cave system. Similar to the Player class, we included a spawn method and a move method for when the player shoots an arrow and misses. We also wrote a separate method to make the rooms surrounding the Wumpus be filled with the Wumpus' stench (i.e. we made it possible to detect the Wumpus if the player stood in a room that was adjacent to the Wumpus). We did this by toggling the boolean values that indicated the Wumpus' nearby presence in specific room objects to true.

Within our Bat class, we also had a spawn method and a move method. Using these methods, we could randomize the bats' initial positions and their new positions after they moved the player. Similar to the Wumpus class, we also included a method to set the bat presence values in adjacent rooms to true. A problem that might arise is

when two bats are in adjacent rooms, but we easily handled this by ensuring our code flow will disallow any unintended conflicts from occurring. In this case, the player will first be moved by the bat to a random room before printing out any new hazards detected.

Lastly, our Pit class was the simplest to implement as there was only a single unmoving pit in the cave system, meaning we just had to spawn it in and set the pit presence values in the adjacent rooms to true.

As an obvious design choice, we made sure to employ good coding practices into the implementation of code such as meaningful variable names, indentation, white space, and of course comments. Additionally, we made sure that the names of methods matched the purpose and scope of their contents.

2.2 Room Mapping Design

For our project, we decided to use a HashMap with the key value pairs being:

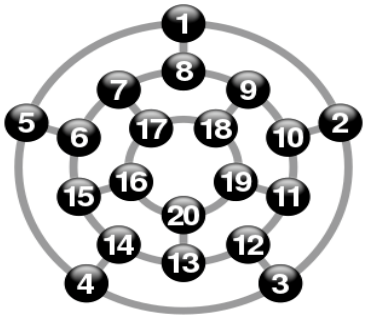
key int room_number and
value Room room{One,Two,Three} etc

As this allowed us to store properties about each room (key) such as the status of each room number (will the player sense presence, will the player occupy the same room as another entity, what are the adjacent rooms) and so forth.

The mappings went as follows

[1, roomOne] with Room roomOne = new Room(5, 8, 2)
[2, roomTwo] with Room roomTwo = new Room(1, 10, 3)
[3, roomThree] with Room roomThree = new Room(2, 12, 4)

and so forth.



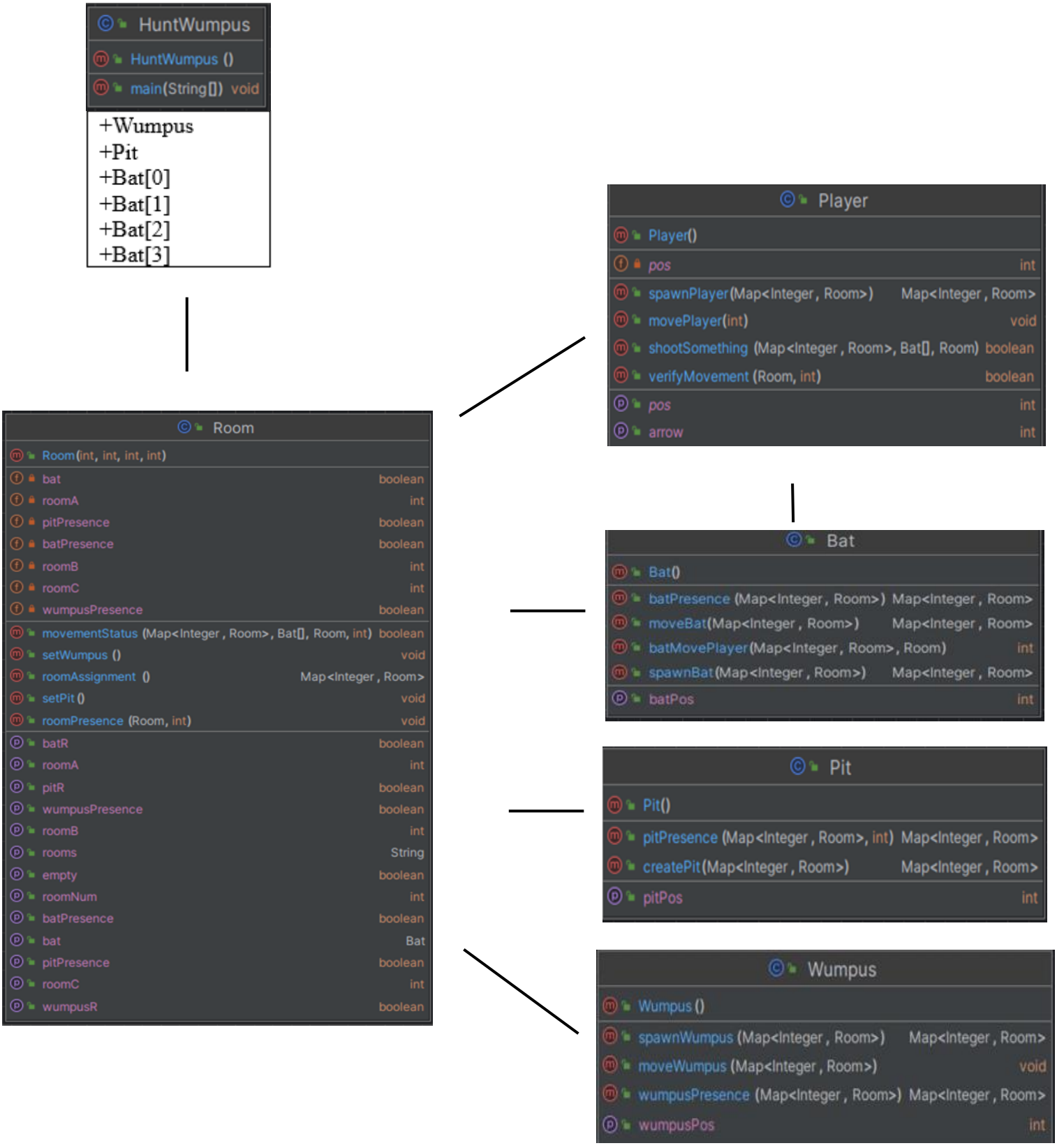
We chose this data structure since a 2D array cannot accommodate for two datatypes, which would be required as a marker (room number) and properties about that marker (room properties), alongside the fact we would have to write code to appropriately reference and index where data is stored in the array.

Additionally two linked arrays would be more of a hassle as they are two separate data structures, while also adding the same complexity as before.

A set was not chosen because its data structure simply does not allow two elements, never mind two elements of a different datatype, or complexity as is in the case of the class instance of Room, and a Tree map was not chosen as there was no need to have the cave numbers sorted, since we directly searched for the room properties using the key.

Therefore as all these qualities were missing from other datatypes, we settled on a HashMap that met all these needs.

2.3 UML Class Diagram



2.4 Psuedo Code

Main Class (HuntWumpus)

```
Initialize game state
Repeat until game over
    Display current room information
    Get player choice (move or shoot)

    If choice is move
        Get player's target room
        Move player to the target room
        Check for game over conditions based on the new room

    Else If choice is shoot
        Get player's target room to shoot arrow
        Shoot arrow into the target room
        Check for game over conditions based on the arrow shot

EndRepeat
Close the game
EndProcedure
```

Room Class

```
// Attributes
  roomNumber: Integer
  roomA, roomB, roomC: Integer
  wumpus, pit, bat: Boolean
  wumpusPresence, pitPresence, batPresence: Boolean

// Constructor
  Room(roomNumber, roomA, roomB, roomC)

// Getter and setter methods for room properties

// Methods to handle game logic:
// - Check if the room contains hazards (Wumpus, Pit, Bat)
// - Handle player movement and update room presence
// - Assign initial rooms in the game
EndClass
```

3. Testing

3.1 Ongoing testing

We found an error where the displayed messages showed different rooms from where the player could actually go. We then found out that the issue was that we displaying the adjacent available rooms based on the players choice rather than the actual rooms surrounding the current player

There was another error where a player would enter a room which is adjacent to either a Wumpus or the pit where they would die. We thought that the pitPresence/pitR and wumpusPresence/wumpusR properties were being mishandled by the room class and main class in the form of assuming that status was equivalent to occupation, therefore leading to both presence and occupation being treated as fatal, however it was later found poor method referencing of setWumpus() and setWumpusPresenence() was the root cause instead.

There were general errors with static and non-static referencing of methods which happened quite commonly due to having several static classes and several non-static class instances of bat, which required us to reappropriate methods to handle instances rather than the class itself.

An error was found that if a player was in room adjacent to two bats, that once the player moved into a room with a bat, they would be randomly moved as expected, however, the presence in the room the player was in would be untoggled, despite one bat still being adjacent to it. This was solved by resetting the bat presence of a room every time the player moves. This therefore also accounts for cases where different types of presence might overlap, as different types of presence will notify the player at once, and if one of those presences are a bat, the ability to reset the cave and reapply the presence will ensure that all rooms have the appropriate status for presence

3.2 Actual testing

Prompt structure:

(A) You are in room (R)<room_number>. Do you want to |move| or |shoot| into the adjacent rooms (AR)<n1, n2, n3>. Do you want to move or shot into these adjacent rooms?
(B) Which room would you like to move into?
(C) Which room would you like to shoot an arrow into?

(D) Invalid input – please key in one of the rooms with appropriate numerical values as displayed!
(E) Invalid input – please choose one of the three adjacent rooms presented!
(F) You are in room (R)<room_number>. Which out of the adjacent rooms: (AR)<n1, n2, n3> do you want to move into?
(G) You are in room (R)<room_number>. Which out of the adjacent rooms: (AR)<n1, n2, n3> do you want to shoot into?
(H) Invalid input - Please enter either move or shoot

(I) The room is empty.
(J) You smell the fabled wretched smell of the Wumpus, it is rancid, and it is unbearable.
(K) A cold breeze envelops you.
(L) Shrieks envelop the cave.

(M) You stumbled into the Wumpus! He munches you up and throws your remains in the trash. Game over.
(N) You stumbled down a pit! Your crushed bones reverberate around the caves. Game over.
(O) Despite the laws of physics, a bat is able to pick you up and place you in room (R)<room_number>

(P) You hear an arrow clink against the floor — Indicates an arrow entered an empty room
(Q) You do not hear an arrow clink against the floor — Indicates that an arrow entered a room with pit
(R) With just one arrow drawn, slay you a beast that has bested great hunters. You win!
(S) You missed! — Indicates Wumpus may have moved
(T) All arrows have been used up! Game over. — Indicates player's arrow count equals zero

For reference, for debugging purposes we have the locations of all bats, cave, and Wumpus, so that we could find entities and interact with them, with the code being as follows:

```
// Debugging lines
    System.out.println("Wumpus pos: " + Wumpus.getWumpusPos());
    System.out.println("Pit pos: " + Pit.getPitPos());
    for (int i = 0; i < bats.length; i++) System.out.println("Bat " + (i+1) + " pos: " + bats[i].getBatPos());
    System.out.println("Player pos: " + Player.getPos());
```

Our thought process through testing was that if we could achieve all of the prompts specified (A–T) without error, we would have comprehensively tested our program as all these prompts would be a result of class interaction and successful conditional outputs.

We later added (U) as we found there was no custom message for trying to shoot a bat, this was then tested for to ensure it had been implemented correctly
(U) "you hear the annoyed shrieks of a bat".

Response	Entry		Results		Results
Prompt	Type	Value	Expected	Actual	Notes
(I)(A)R16, AR 20,15,17	String	"5"	Prompt (H), (A)R16 AR 20,15,17	Mostly as expected, however a test note was left in the program and promptly removed	**Test Batch One** **Start of attempt** Room is empty **(I)** States current room and adjacent **(A)** (frequent)
(H)(A)R16, AR20,15,17	String	"move", "5"	Prompt (B) after "move", prompt (E)(F)R16 AR 20, 15, 17 after "5"	As expected	User enters a room that is not adjacent, error message **(E)** informs, **(F)** allows to retype room
(D)(F)R16 AR20,**15**,17	String	"**15**"	Prompt (I) & (A) R15, AR 14,16,6	As expected	Empty room **(I)**
(I)(A)R15 AR14,16,6	String	"move", "Fourteen"	Prompt (B) after "move", prompt (D)(F)R15 AR14,16,6 After "Fourteen"	As expected	User enters a room number as a word rather than an integer, error message **(D)** informs**, (F)** allows to retype room
(B)(D)(F)R15,	String	"**14**"	Prompt (K) & (A) R14, AR	As expected	Cave is adjacent **(K)**

AR **14**,16,6			13, 4, 15		
(K)(A) R14, AR **13**,4,15	String	“shoot”, “**13**”	Prompt (C) after “shoot” prompt (Q) after “13” (A)R14, AR13,4,15	As expected	Shooting into pit and receiving output that the arrow does not hit the floor **(Q)**
(A) R14, AR **13**,4,15	String	“move”, “**13**”	Prompt (B) after “move”, prompt (N) after “13”	As expected	Gameover moving into pit **(N)** **End of attempt.**
(I)(A) R14 AR**13**,4,15	String	“shoot” then “**13**” 5 times	Prompt (P), then (A) 5 times in a row, then Prompt (T)	As expected	**Test Batch Two** **Start and end of Attempt** Player runs out of arrows **(T)**
(I)(A) R9 AR8,18,**10**	String	“move”, “**10**”	Prompt (L)(A)R10 AR9,2,11	As expected	**Test Batch Three** **Start of Attempt** **(L)** Bats are nearby
(L)(A)R10 AR9,**2**,11	String	“move”, “**2**”	Prompt(K)(J)(A)R2 AR1,10,3	As expected	Both pit **(K)** and Wumpus **(J)** are nearby
(K)(J)(A)R2 AR**1**,10,3	String	“shoot”, “**1**”	Prompt(S)(Q)(K)(A)R2 AR1,10,3	As expected	Missed the Wumpus **(S)** Wumpus has moved as presence **(J)** is gone The room we hit holds a cave as arrow doesn’t clink **(Q)**
(S)(Q)(K) (A)R2 AR1,10,**3**	String	“move”, “**3**”	Prompt(J)(A)R3 AR2,12,4	As expected	Wumpus presence returns **(J)**
(J)(A)R3 AR,2,**12**,4	String	“shoot”, “**12**”	Prompt(R)	As expected	Wumpus is shot **(R)** **End of attempt**
(J)(L)(A)R18 AR17,9,**19**	String	“move”, “**19**”	Prompt(O)R4, (I)(A)R4, AR3,14,5	As expected	**Test Batch Four** **New Attempt** Wumpus and bat are nearby, move into room with bat and moved **(O)**
(I)(A)R4 AR3,**14**,5	String	“move”, ”**14**”	Prompt (L)(A)	As expected	Move into room with nearby presence of nearby bat
(L)(A)R14 AR13,4,**15**	String	“move”, ”**15**”	Prompt(O)R10, (L)(A)R10, AR9,2,11	As expected	Move into room with bat and moved into room with presence of nearby bat
(L)(A)R10 AR9,2,**11**	String	“shoot”, “**11**”	Prompt(A)R10, AR9,2,11	The program gives no existence of a bat in this room	Will be fixed with custom prompt, **(U)**
(A)R10 AR9,2,**11**	String	“move”, “**11**”	Prompt(O)R5, (L)(A)R5, AR4,6,1	As expected	Debugging logs show that the rooms with a bat that the player interacts with no longer have that bat in them, and that that bat has moved room. Additionally, during all the

					movement, no hazards moved into the same room
(L)(A)R1 AR5,**8**,2	String	“move”, “**8”**	Prompt (I)(A)	As expected	**Test batch Five New Attempt**
(I)(A)R8 AR7,1,**9**	String	“move”, “**9”**	Prompt(J)(A)	As expected	Move into room with nearby presence of Wumpus
(J)(A)R9 AR8,**18**,10	String	“move”, “**18”**	Prompt(M)	As expected	Munched up by the Wumpus (M) **End of Attempt**
(I)(A)R10, AR9,2,11	String	“Move”	Prompt(H)(A)R10, AR9,2,11	As expected, there is case sensitivity.	**Test Batch Six Start Attempt**
(H)(A)R10, AR**9**,2,11	String	“move”, “**9**”	Prompt(L)(A)R9, AR8,18,10	As expected	Bat presence nearby (L)
(L)(A)R9, AR**8,**18,10	String	“shoot”, “**8**”	Prompt(U)(A)R9, AR8,18,10	As expected	The custom message for shooting into a room with a bat (U) has been successfully implemented. **End Attempt**

Through effective ongoing testing, not many errors were detected through formalised testing. Due to expected responses appearing consistently, it is safe to say the program is robust to reasonably unexpected output (outside of extensive data entry which would cause memory issues), provides the intended functionality, and is fit for purpose.

The testing evaluation log to back up this claim is seen in the proceeding page.

Through test batch one (cave system mappings, error handling, arrow and movement functionality, ability to identify status of room with nearby hazards, the ability of the cave to kill the player):

- the mappings of each room match their expected rooms, as seen by prompt (A) and the mapping referenced in design earlier (this is shown throughout all test batches)
- that the program is able to handle the user entering a room that is not adjacent (E)(F) and prompt the user to re-enter
- that the program is able to handle text when it expects integers and prompt the user to re-enter (D)(F)
- the programs ability to implement presence as a notification to the player that an entity is in a nearby room (K)
- the arrow functionality was implemented and able to identify the status of a room as containing a pit hence prompt (Q)
- the ability of the player to perish was implemented by being able to identify the status of a room as containing a pit hence prompt (N)
- the ability to distinguish an outcome of interacting with a room by moving or arrow by the program

Through test batch two (running out of arrows):

- through consecutively shooting arrows, we have shown that the arrow count works properly by the appearance of prompt (T)

Through test batch three (multiple enemies adjacent, Wumpus functionality):

- Showing the consistency that all three entities (pit, bat, and Wumpus) can communicate their presence to the player in the adjacent rooms, alongside the fact that multiple entities can communicate their presence to the player at once, and additionally that once an entity moves away, they do not overwrite the presence of another adjacent entity, as seen by the Wumpus moving away after a missed arrow, but the presence of the cave remaining (K)(J)
- The functionality of missing the Wumpus making him move to an adjacent location (S) and lack of prompt (J)
- The functionality of shooting the Wumpus ending the game (R)

Through test batch four (Testing the ability of bats consecutively for presence, moving the player, and moving themselves)

- Again proving the functionality of both the Wumpus and Bats ability to be adjacent to the player at once as seen by prompts (J)(L)
- The ability of the bat to move the player into a new random empty room (O) alongside moving their own positions as seen in the debugging logs used. This process was repeated three times with the bat moving us into empty rooms, or a room adjacent to a bat, while moving themselves in unique locations. Showing the consistency of this process
- Finding the omission of a notification prompt when you shoot into a room with a bat (U)

Through test batch five (Final piece of Wumpus functionality):

- That the Wumpus functionality to kill the player if they enter the same room is successfully functioning (M)

Through test batch six (testing case sensitivity and previously omitted notification prompt):

- Showing that the program does not have case sensitivity (it is presumed that the user will be able to recognise the emphasis on case in the error message)
- That the previously omitted notification prompt of shooting into a room with a bat was successfully implemented (U)

4. Evaluation

Through effective design there was a distinct separation between classes, as there was a clear model that the subclasses with superclass entities would provide the ability to spawn, update Rooms with presence and actual room occupation, and movement based on player interaction, and these changes would be actualised through all of their general interactions with the major class Room in order to make them present in the game.

There was a general consensus of what the program structure would be between us as we were coding it, as we had made sure to identify the requirements and the practical solutions to those requirements beforehand, however, it might have helped to be more exactly precise in terms of forming diagrams earlier (such as UML Class Diagrams and dataflow) as models to reference class interaction, as there were points where we were unsure of the overall impact of editing lines of code

Clear comments were made throughout the project which minimised confusion, code was moved and separated to be in the most appropriate classes (for example, there were two movePlayer() methods in player, however since one method was decisively used by the Bat class to move the player randomly, we moved it inside of Bat instead), and the scope of Methods were maintained and did not exceed their purpose and scope. All of this combined really helped to maintain overall program cohesion, making it easier to understand, edit, and adjust.

We used try and catch statements to handle inappropriate user input and allowed them to re-input their response through validation do while loops, therefore keeping our program robust and less prone to runtime errors.

In terms of team collaboration, we believe our decision to code the fundamental framework of our program together so that we were certain on the implementation details was the correct decision as this then allowed us to independently add components to the program separately and effectively.

Additionally, we both formed parts of the report ourselves and then met up to combine our reports into an overall cohesive program specification which has allowed us a more comprehensive report through both joint and independent perspectives.

5. Conclusion

In conclusion, our program successfully implements a working version of the Hunt the Wumpus game, which works in accordance with our desired rule set. The code has been clearly documented throughout our report, through our README.md file including our rule set, and through our actual code which has been clearly documented.

This project provided an insightful test of our programming skills. It served as a good summary of what we have learnt in CS1002 and in the first few lectures of CS1003. For instance, we also experimented with using Maps, which were a new concept that we only recently covered in CS1003. However, we ultimately managed to successfully implement a HashMap within our final project submission.

If we were given more time to work on our project, as mentioned before in the Analysis section we would like to implement additional features like a procedurally generated map that would take different shapes than a dodecahedron, rather than hard coding the mapping of one shape, alongside difficulty modes or sliders that indicate the frequency of bats and pits. There were further ideas like having additional behaviours for entities like the Wumpus who could move on their own and travel around the map regardless of shooting an arrow near it. All of these features could provide additional change and interest for the player.

One last feature we considered would be a GUI such as drawing out a map and marking out the locations that the player has been to before.

6. References

[1] *Hunt The Wumpus*. (1976). www.atariarchives.org. https://www.atariarchives.org/bcc1/showpage.php?page=247

[2] *Hunt the Wumpus.* (n.d.). https://osric.com/wumpus/