HUNT THE WUMPUS

Tristin Doherty & Eugene Kong

(THIS IS A GROUP SUBMISSION)

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.

For additional enhancements we added room status effects and additional arrows to spawn into empty rooms, additional health for the Wumpus alongside the ability to heal when shot after enough turns, and moving the bat to an empty random room after it moves the player.

1.2 Feasibility

With available resources like GitHub 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.

Although we started before the specification was released due to mounting time pressure, there was no hindrance to the foundational structure of the program

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
- Four hazards (pit, bats, rooms with loss of senses effects, and the Wumpus) that would interact with the Player.
- Room(s) that would give the Player an additional arrow.
- Player to entity interactions, such as slaying the Wumpus, being transported by bats, losing their sense of smell or hearing by entering a room with an effect, or dying to either pit or Wumpus.
- A test-based gameplay, which involves the Player first being given the option to either move or shoot, then specifying the room to perform the action.
- The Player's win criteria, which involved the Player having to shoot the Wumpus twice within five moves, else the Wumpus' health will.

We decided that the definite boundaries within time constraints for the project would be:

- The game will be strictly text-based
- The map will not be generated (i.e. the map will be hard-coded)
- The quantity and the behaviours of hazards would be fixed (e.g. the number of bats, the number of rooms inflicted with the hearing and smell loss effect etc.)

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 initialised 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. Additionally, the Wumpus class also had a health field to regenerate the Wumpus if the Player does not shoot him twice within a set number of moves.

Within our Bat class, we also had a spawn method and a move method. Using these methods, we could randomise 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.

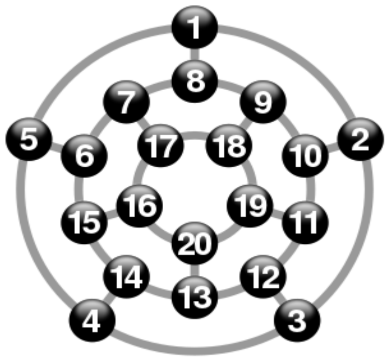
Lastly, our Pit class was the simplest to implement as there was only a single unmoving pit in the cave system, meaning we just had to spawn it in and set the pit presence values in the adjacent rooms to true.

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 each key-value pair being:
> key int roomNumber
> value Room room{roomNumber, adjacentRoom1, adjacentRoom2, adjacentRoom3}

This hashmap data structure allowed us to store properties about each room. For instance, in a given room, we could store and check for the presence of a specific hazard.

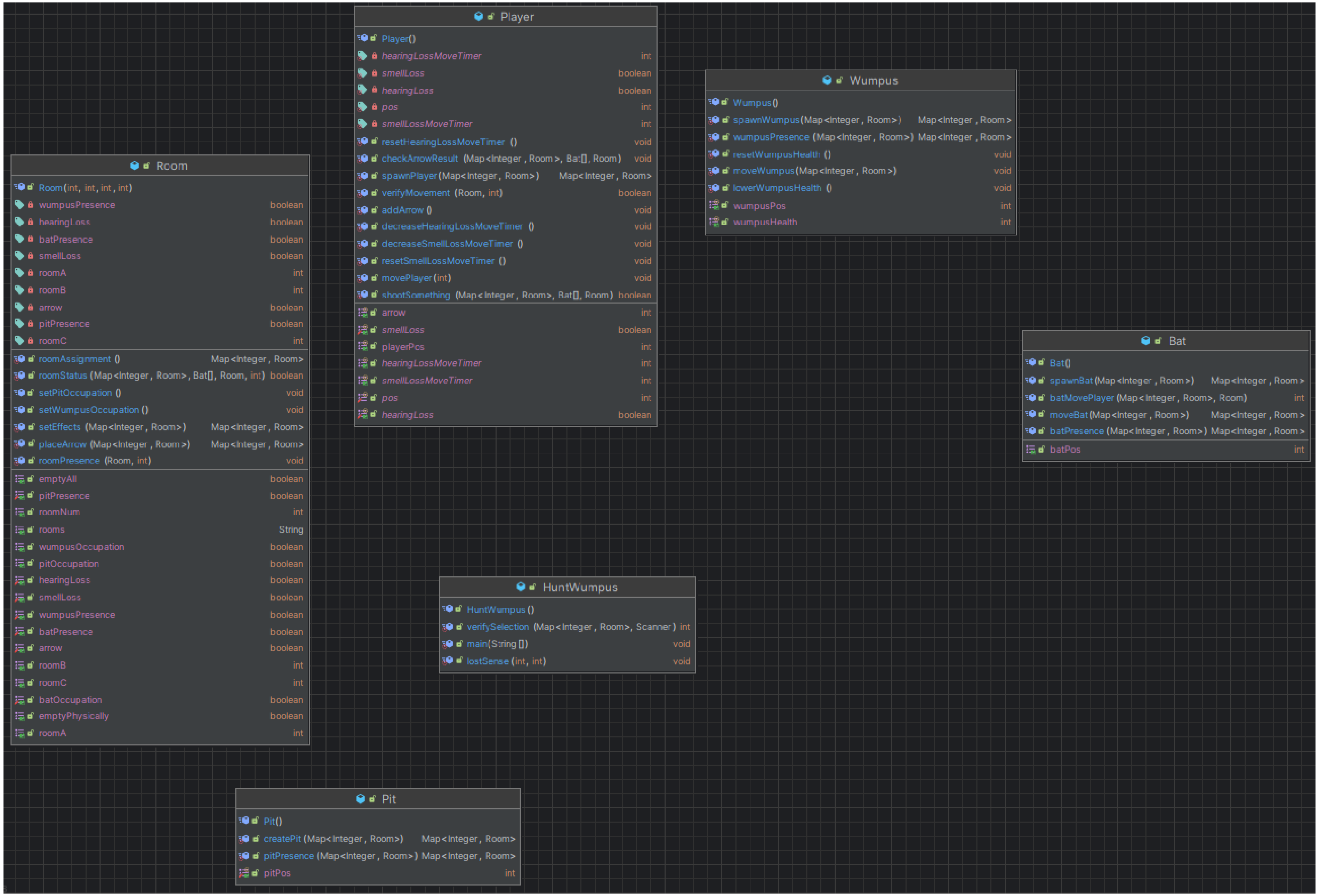


We chose this data structure as it is effective in storing two data types as a related key and value pair. We could also use a wrapper class (the Room class) to include other fields in our hashmap values. In comparison, an array cannot accommodate more than a single data type, which was necessary in the way we designed our program.

Other data structures we thought might be potentially useful were using two linked lists or sets. However, we decided that this would be more of a hassle as they are two separate data structures, while also adding the same complexity as before.

Therefore, after weighing the pros and cons of using each data structure, we finally decided on using a hashmap to represent our cave system.

2.3 Class Diagram



Player

Player()	
hearingLossMoveTimer	int
smellLoss	boolean
hearingLoss	boolean
pos	int
smellLossMoveTimer	int
resetHearingLossMoveTimer ()	void
checkArrowResult (Map<Integer , Room >, Bat[], Room)	void
spawnPlayer (Map< Integer , Room>)	Map< Integer , Room>
verifyMovement (Room, int)	boolean
addArrow ()	void
decreaseHearingLossMoveTimer ()	void
decreaseSmellLossMoveTimer ()	void
resetSmellLossMoveTimer ()	void
movePlayer (int)	void
shootSomething (Map< Integer , Room>, Bat[], Room)	boolean
arrow	int
smellLoss	boolean
playerPos	int
hearingLossMoveTimer	int
smellLossMoveTimer	int
pos	int
hearingLoss	boolean

Wumpus

Wumpus()	
spawnWumpus (Map<Integer , Room>)	Map <Integer , Room >
wumpusPresence (Map<Integer , Room>)	Map <Integer , Room >
resetWumpusHealth ()	void
moveWumpus (Map<Integer , Room >)	void
lowerWumpusHealth ()	void
wumpusPos	int
wumpusHealth	int

Room

Room (int, int, int , int)	
wumpusPresence	boolean
hearingLoss	boolean
batPresence	boolean
smellLoss	boolean
roomA	int
roomB	int
arrow	boolean
pitPresence	boolean
roomC	int
roomAssignment ()	Map <Integer , Room>
roomStatus (Map <Integer , Room> , Bat[] , Room, int)	boolean
setPitOccupation ()	void
setWumpusOccupation ()	void
setEffects (Map<Integer , Room>)	Map <Integer , Room>
placeArrow (Map <Integer , Room >)	Map <Integer , Room>
roomPresence (Room, int)	void
emptyAll	boolean
pitPresence	boolean
roomNum	int
rooms	String
wumpusOccupation	boolean
pitOccupation	boolean
hearingLoss	boolean
smellLoss	boolean
wumpusPresence	boolean
batPresence	boolean
arrow	boolean
roomB	int
roomC	int
batOccupation	boolean
emptyPhysically	boolean
roomA	int

Bat

Bat()	
spawnBat (Map <Integer , Room >)	Map <Integer , Room >
batMovePlayer (Map < Integer , Room>, Room)	int
moveBat (Map <Integer , Room >)	Map <Integer , Room >
batPresence (Map< Integer , Room>)	Map <Integer , Room >
batPos	int

HuntWumpus

HuntWumpus ()	
verifySelection (Map < Integer , Room>, Scanner)	int
main (String [])	void
lostSense (int, int)	void

Pit

Pit()	
createPit (Map< Integer , Room>)	Map < Integer , Room>
pitPresence (Map< Integer , Room>)	Map < Integer , Room>
pitPos	int

2.4 Pseudocode

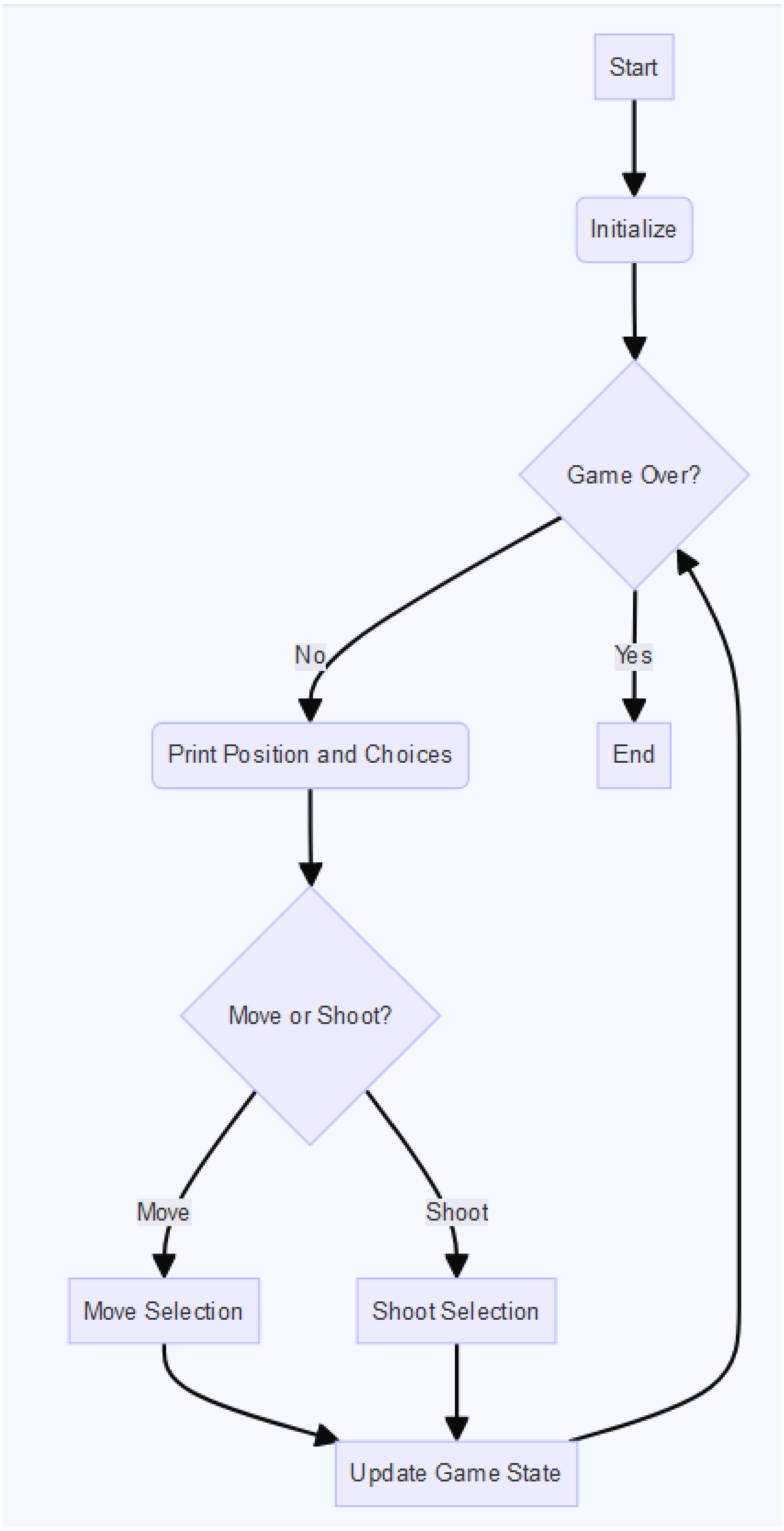
Note the pseudo code was updated to match the actual implementation after the specification had been released as it would not be realistic to maintain the pre-specification version.

```
Main Class
      Setup the cave system:
            Map rooms = roomAssignment()
            createPit()
            spawnWumpus()
            spawnBats()
            setEntityPresence()
            setRoomEffects()
            spawnPlayer()

      Display initial room status
            Loop until game over:
                  Print current position and available choices
                  Read player's choice

                  If choice is to move:
                              Read the room to move into
                              Verify the input and/or selected room choice
                              Update game status based on the selected room
                              Check for player's lost senses
                  Else If choice is to shoot:
                              Read the room to shoot into
                              Verify the input and/or selected room choice
                              Update game status based on the selected room
                              Shoot an arrow into the room
                              Check if the player is out of arrows
                  End If

                  If the Wumpus's health has been lowered:
                         Check if enough turns have passed for Wumpus to heal
                  End If

      End
```

```
Room Class Procedures:
       Method getAdjacentRooms():
       Return the adjacent room numbers of the current room.

       Method applyHearingLossEffect():
         Apply the hearing loss effect to the current room by updating properties.

       Method applySmellLossEffect():
         Apply the smell loss effect to the current room by updating properties.

       Method spawnArrowInRandomRoom():
              Randomly choose an empty room to spawn an arrow.
              Mark the chosen room as containing an arrow.
              Return the updated rooms map.

       Method EmptyRoomCheck():
              Check if the current room is physically empty (no hazards present).
              Return True if the room is physically empty, False otherwise.

       Method RoomClearCheck():
              Check if the current room is completely empty (no hazards/presences).
              Return True if the room is completely empty, False otherwise.

       Methods:
              setWumpusOccupation/presence(): // Update Wumpus presence/location
              setPitOccupation/presence(): // etc
              setBatOccupation/presence(): // etc

       Method setEffects():
              Randomly select rooms to apply hearing loss and smell loss effects.
              Check if rooms are not occupied; if so, save the room chosen.

       Method evaluateRoomStatus(rooms, bats, room, choice):
              Check if the Player encounters any hazards upon entering the room.
              If the room has hazard/presence/effect(s), notify accordingly.
              Return True if the game should end, False otherwise.
```

2.5 Flowchart

General Gameflow:



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 displayed the adjacent available rooms based on the user's choice rather than the actual rooms surrounding the Player.

There was another error where the Player would enter a room which is adjacent to either a Wumpus or the pit where they would die. We thought that the pitPresence/pitOcc and wumpusPresence/wumpusOcc 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 setWumpusOccupation() and setWumpusPresence() 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.

A bug was found in the event that the Player was in a room adjacent to two bats. Once the Player moved into one of the rooms with a bat, they would be randomly moved as expected. However, the presence in the room the Player was in would be untoggled (i.e. the room that the Player moved out from will have its batPresence attribute toggled to false). This should not be the case, since one bat is adjacent to it. This was solved by resetting the bat presence of a room every time the Player moves. This 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

We have tested our program rigorously to verify that it generates the desired outputs, and deals with problematic user inputs properly without breaking the code. Additionally, we also made sure to check that all of our game rules worked properly through thorough debugging.

To make testing convenient, we also made sure to print out the positions of the Player, the room with the spare arrow, and all the hazards in the map. This enabled us to deliberately move the Player into rooms that are adjacent to hazards, or into rooms with hazards to get the program to generate messages for testing purposes.

```
// Debugging lines
System.out.println();
System.out.println("Player position: " + Player.getPlayerPos());
System.out.println("Wumpus position: " + Wumpus.getWumpusPos());
System.out.println("Pit position: " + Pit.getPitPos());
for (int i = 1; i <= bats.length; i++) {System.out.println("Bat " + i + " position: " + bats[i-1].getBatPos());}
for (Map.Entry<Integer, Room> entry : rooms.entrySet()) {
  if (entry.getValue().getHearingLoss()) System.out.println("Hearing loss room: " + entry.getKey());
  if (entry.getValue().getSmellLoss()) System.out.println("Smell loss room: " + entry.getKey());
  if (entry.getValue().getArrow()) System.out.println("Arrow room: " + entry.getKey());
}
System.out.println();
```

To reduce the repetitive messages, we will also use the table below to abbreviate the messages.

Message Type	Message Code	In-Game Message
Input prompt	I1(X, Y, A, B, C)	You are in room X. You have Y arrows. Do you want to |move| or |shoot| into the adjacent rooms A, B, or C?
	I2	Which room would you like to move into?
	I3	Which room would you like to shoot into?
	I4(X, A, B, C)	You are in room X. Which rooms out of the adjacent rooms: A, B, or C do you want to move/shoot into?
Error outputs	E1	Invalid input – Please enter either "move" or "shoot".
	E2	Make sure your entry is valid by choosing one of the three rooms presented!
	E3	Invalid input – Please key in the rooms with appropriate numerical values as displayed!
Presence messages	M1	The room is empty.
	M2	Shrieks envelope the cave.
	M3	A cold breeze envelops you.
	M4	You smell the fabled wretched smell of the Wumpus, it is rancid, and it is unbearable.
Hazard messages	H1	You stumbled into the Wumpus! He munches you up and throws your remains into the trash. Game over.

	H2	You stumbled down a pit! Your crushed bones reverberate around the caves. Game over.
	H3(X)	Despite the laws of physics, a bat is able to pick you up and place you in room X.
	H4	Oh no! You have been affected by the HEARING LOSS effect! You can no longer hear bats!
	H5	Oh no! You have been affected by the SMELL LOSS effect! You can no longer smell the Wumpus!
	H6	You can hear again!
	H7	You can smell again!
	H8(X)	X turn(s) remain until the Wumpus heals his wounds.
	H9	Through unworldly means the Wumpus has regained his health.
Arrow messages	A1	You found an arrow!
	A2	All arrows have been used up! Game over!
Shooting messages	S1	As the arrow pierces the Wumpus and leaves him close to death, he roars and scurries away to a nearby room.
	S2	With just two arrows drawn, slay you a beast that has bested numerous great hunters. You win!
	S3	You missed!
	S4	You hear the annoyed shrieks of a bat.
	S5	You don’t hear the arrow clink against the floor.
	S6	You hear the arrow clink against the floor.

The table below shows our output, which checks if the message generated by the program after the user inputs an option matches our expected output.

Message Generated	User Input	Game Output	Expected? (Y/N)	Remarks
		Play #1		
M2, I1(20, 5, 19, 13, 16)	hello	E1	Y	Entering any Strings apart from “move” and “shoot” outputs E1
I1(20, 5, 19, 13, 16)	move	-	-	-
I2	1	E2	Y	Entering an invalid room number will generate E2
I4(20, 19, 13, 16)	test	E3	Y	Entering a non-integer String will generate E3
I4(20, 19, 13, 16)	19	M1	Y	-
I1(19, 5, 18, 11, 20)	move	-	-	-

I2	20	M2	Y	Being adjacent to a bat will generate M2
I1(20, 5, 19, 13, 16)	move	-	-	-
I2	16	H3(19), M2	Y	The bat picks up and moves the Player to a random empty room (note that the bat also moved into a random empty room)
I1(19, 5, 18, 11, 20)	move	-	-	-
I2	11	M2	Y	-
I1(11, 5, 10, 19, 12)	move	M1	Y	-
I1	10	M1	Y	-
I1(10, 5, 9, 2, 11)	move	-	-	-
I2	9	M4, M2	Y	Being adjacent to the Wumpus generates the appropriate output
I1(9, 5, 8, 18, 10)	move	-	-	
I2	8	H1	Y	Game stops after the Player runs into the Wumpus, with the correct death message being generated
Play #2				
M4, I4(15, 5, 14, 16, 6)	move	-	-	-
I2	16	H4, M1	Y	The Player gets the message that they have entered the hearing loss room
I4(16, 5, 20, 15, 17)	move	-	-	-
I2	17	M3, M1	N	The Player should receive M3 since the pit is adjacent, but should not be receiving M1
I1(17, 5, 16, 7, 18)	shoot	-	-	-
I3	7	S5	Y	The Player should receive S5 if they have the hearing loss effect active, unless they shoot the Wumpus

I1(17, 4, 16, 7, 18)	shoot	-	-	Note that the arrow count has decreased by one
I3	18	S5	Y	-
I1(17, 3, 16, 7, 18)	move	-	-	-
I2	7	H3(9), M3, M1, H6	N	M1 should not be there, since M2, M3 or M4 generating means that M1 should not be printed (however, H6 works as intended, since we programmed our hearing loss effect to vanish after two moves)
I1(9, 3, 8, 18, 10)	move	-	-	-
I2	8	M1	Y	-
I1(8, 3, 7, 1, 9)	move	-	-	-
I2	1	M2	Y	Bat was present in room 5
I1(1, 3, 5, 8, 2)	move	-	-	-
I2	2	M2	Y	Bat was present in room 10
I1(2, 3, 1, 10, 3)	move	-	-	-
I2	3	H5, M1	Y	H5 is correctly printed when the Player walks into the smell loss room
I1(3, 3, 2, 12, 4)	move	-	-	-
I2	4	H4, M1	Y	Hearing loss and smell loss effects can be stacked
I1(4, 3, 3, 14, 5)	move	-	-	-
I2	5	H3(17), M3, H7	Y	Smell loss effect vanishes after two moves
I1(17, 3, 16, 7, 18)	move	-	-	-
I2	16	H5, M1, H6	Y	Note that the smell and hearing loss rooms will each be randomly moved after the Player enters it
I1(16, 3, 20, 15, 17)	move	-	-	-
I2	15	M1	Y	The Wumpus is in room 6 (adjacent to 17) but the Player cannot smell it with the smell loss effect active

I1(15, 3, 14, 16, 6)	shoot	-	-	-
I3	14	S3, S6, M1	Y	If the Player shoots while they are adjacent to the Wumpus, S3 will be printed (also note that the Wumpus may move after the Player shoots while adjacent to him)
I1(15, 2, 14, 16, 6)	shoot	-	-	-
I3	6	S6	Y	-
I1(15, 1, 14, 16, 6)	shoot	-	-	-
I3	14	S6, A2	Y	Once the arrow count hits zero, the game ends
Play #3				
M2, I1(19, 5, 18, 11, 20)	shoot	-	-	-
I3	20	S4	Y	S4 is correctly generated when the Player shoots into a room with a bat
I1(19, 4, 18, 11, 20)	move	-	-	-
I2	18	M1, A1	Y	The Player walks into the room with the spare arrow
I1(18, 5, 17, 9, 19)	move	-	-	Note that the arrow count has increased by one following the previous move
I2	9	M1	Y	-
I1(9, 5, 8, 18, 10)	move	-	-	-
I2	10	M4	Y	-
I1(10, 5, 9, 2, 11)	shoot	-	-	-
I3	2	S1, M1, H8(5)	Y	S1 and H8 are correctly generated after the Player shoots the Wumpus for the first time
I1(10, 4, 9, 2, 11)	shoot	-	-	-
I3	2	S6, H8(4)	N	S6 and H8 are correctly generated, but M1 should also have been printed
I1(10, 3, 9, 2, 11)	move	-	-	-

I2	2	M4, H8(3)	Y	Notice how the Wumpus health decreases with each turn
I1(2, 3, 1, 10, 3)	shoot	-	-	-
I3	3	S2	Y	Wumpus dies if the Player shoots him twice within five moves, and the game ends
Play #4				
M2, I1(19, 5, 18, 11, 20)	move	-	-	-
I2	18	M4, M2	Y	-
I1(18, 5, 17, 9, 19)	shoot	-	-	-
I3	9	S1, M2, H8(5)	Y	-
I1(18, 4, 17, 9, 19)	shoot	-	-	-
I3	9	S6, M2, H8(4)	Y	Fixed error in Play #3
I1(18, 3, 17, 9, 19)	move	-	-	-
I2	19	M2, H8(3)	Y	-
I1(19, 3, 18, 11, 20)	move	-	-	-
I2	11	M1, H8(2)	Y	-
I1(11, 3, 10, 19, 12)	move	-	-	-
I2	12	M3, M2, H8(1)	Y	-
I1(12, 3, 11, 3, 13)	shoot	-	-	-
I3	13	S5, M3, M2, H9	Y	The Wumpus will regain its health after five moves
I1(12, 2, 11, 3, 13)	move	-	-	-
I2	13	H2	Y	The Player dies after falling into the pit, and it is game over

4. Individual Contributions

4.1 Eugene Kong

Apart from programming methods in the classes we used in our submission, I also drafted the report, which we implemented when planning and writing out our final version of our report.

I felt that I contributed most significantly during the debugging phase of our project. I constantly worked on the project, so I was relatively familiar with the methods in our program and how they were related to one another. I was therefore able to quickly figure out where certain bugs occurred and fix them accordingly.

Some of the bugs that I found were extremely hard to find, as they only occurred in highly specific cases of the generation of the different kinds of hazards. One such bug would be the aforementioned situation where there is more than one room adjacent to the Player (let's call the room that the Player is currently in room X) that has a bat present. If one of the bats moves to another room after teleporting the Player to another room, then room X should still have its batPresence boolean equal to true. This was not always the case, as earlier versions of our algorithm would have resulted in room X having its batPresence boolean toggled to false. This is just one of the few inconsistencies that happened with our program, and there were numerous other bugs that I had to painstakingly search for and fix.

For the report, I also helped by testing the code and contributed to creating the testing table, which ensured that everything in our program was working the way it should. Additionally, I also helped with formatting to ensure that everything looks presentable.

In conclusion, I am satisfied with our final product, and I am happy that Tristin and I managed to coordinate well with one another. We managed to finish the project together without too many hiccups, and I have learned more about pair programming as a whole.

4.2 Tristin Doherty

At the beginning of the project I formed the general design philosophy of every class mainly interacting through the Room class instances stored inside the rooms mapping. Once I had thoroughly discussed the (then available powerpoint) specifications I got to programming this main foundation of the programs with updating occupation and presence properties through our initial pair programming where we sat together to make sure we had a core foundation we could both work with independently.

Afterwards we then worked separately with each other updating the entities and hazard classes and myself adding additional functionality to the Wumpus.

Additionally I then worked through the code to comprehensively notate our program while also removing redundancy through methods such as validating room selection, unnecessary referencing of variables (as methods were updating them by reference), excess properties and variables which could be reduced, better variable/method names and generally making the program less redundant and more maintainable for future updates.

For the report I provided our project a strong foundation as I emphasised a comprehensive report that would clearly detail the whole project implementation, so I was responsible for forming the majority of each section and subsection from Analysis to Conclusion, however my partner did update my actual testing (3.2) logs after the specification had been released as the logs had to be completely rewritten, and they did provide a good description of our design decisions, on top of my description of our choice of data structure.
progress, alongside our joint written conclusion.

Overall I believe we were able to cooperate very well, there was enthusiasm to complete the project early on, and we communicated regularly on updates on the program and what was needed, and we helped each other out when one of us needed help, requested it, or noticed a need for it.

5. 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 the Player class, 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.

However, in future for the testing section, it would be more efficient to implement junit testing in order to fall under test driven development, rather than relying on ongoing testing throughout coding and actual testing which mostly has the aim to try prove our main functionality works, which can miss edge cases if we are unlucky with experimenting with the program and not finding undiscovered bugs.

6. 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.

7. 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/

[3] *Mermaid Syntax (flowchart diagram)* https://www.mermaidchart.com/app/dashboard