Forbidden Maze is a game designed and implemented by Tawheed Sarker Aakash, Josh Le, Jay Bator, and Rahul Naterwala. The game consists of the player traversing through a randomly generated map in order to collect gems. The player must avoid traps that will lower their score, and enemies that will kill them. Once the player collects all the gems the exit appears and the player can escape. https://www.youtube.com/watch?v=yKNMLWtUBlg provides a demo of the core feature of the game and how elements interact with each other. A tutorial is found at the beginning in the how to play screen, showing the objective of the game alongside controls and win/lose conditions

Phase 1 and Initial Planning:
During the initial planning phase, we envisioned a maze design with a matrix of containers as cells. The containers are allowed to hold an object or an entity in our case.This design allowed for the implementation of all entities using polymorphism and inheritance and stood out as the core design concept for the maze and game in general. While the initial planning and UML diagram had to be scrapped, this core design concept stuck with us throughout the game's development.

One of the core concepts of the maze design that we envisioned was the idea of a matrix of containers for the cells. Each cell will have the capability to hold an object or an entity in our case. This allowed us to implement all the entities using polymorphism and inheritance. To move these entities we would swap the entities between the cells each time we moved an entity. Although a lot of the initial planning along with the UML diagram had to be scraped but this model along with some lessons that we learnt while designing the game initially stuck with us throughout the game's development.

Maze generation:
As mentioned above the idea of containers and holding entities in them was the main idea of how the maze would interact with the game. To generate the maze we used Prim's maze generation algorithm which uses pathfinder logic to make a proper grid, allowing access to all the branches of the maze along with the start and end point of the maze. Using a static factory method allowed us to have different levels seamlessly.

Entities:
All entities had similar features while serving different purposes. To allow for ease in swapping, we used polymorphism to form the various entities in the game.This design decision allowed moving both players and enemies easier with less code.Polymorphism also reduced the number of checks required when checking for collisions, as we only needed to check the type of entity and cell before allowing movement and other aspects of collision handling.

User Interface:
Originally we wanted to build the user interface by making a bunch of different types of UI objects that inherit from each other, but we found it was easier to simply have one class for rendering UI then checking in another class so see if click hit the area where a button would be. The UI is created by looking at an 2d array of enums specifying what is where and rendering

based off of this. This was a powerful design decision because it allowed the UI to be robust to change in the game's logic by separating it from the game logic.

Event Handler:
In our design phase we did not plan to have an Event Handler class, but near the end of our phase 2, it became apparent that we were going to need a bridge class to make it easier to interface the multiple classes that we had built throughout the project. Event handler holds both game and UI instances that themselves are connected to the rest of the classes we built. Then when a game event happens like a tick or button press, the event handler will tell both the game and the UI to update along with passing the event that happened.

Assembly Phase:

Assembling was a lot easier due to certain design decisions, some which are mentioned above. In our state package, we had the game class handle everything that would require logic. This included methods for moving enemies and players, handling collisions and the class served as a hub between all the entities and the maze and allowed interaction between them.

Some of the most important lessons we learned throughout the design/implementation/testing phases of this project were:

- Creating an application outside of CLI
    - This was the first time any of us had designed and implemented a game outside of the command line. Creating a GUI is itself inherently a unique experience that we didn't have before this project, alongside having to connect a frontend/backend. In previous assignments print statements would be placed directly within the logic of the game, so connecting and transferring inputs and updates between the GUI and the logic of the game proved to be both difficult and a great learning experience.
- Testing an application using a framework
    - We used the JUnit5 testing framework for the testing phase of this project. This was vastly different from previous projects. Before learning about testing frameworks and about things like line coverage, method coverage, and branch coverage, we would simply test the program by running it over and over again with different input parameters. If the program would not produce the results we expected, we would litter the program with various print statements throughout to see where the program isn't acting like it intended. Now, we know how to write unit tests to cover all possible situations and ways our methods can be called in.
- Working and collaborating in a team
    - This was the first big group project that we had to do at SFU, and the first time working with others in relation to coding and development. Working in a group is very different in comparison to working by yourself. Some of the biggest issues we ran into when working as a team was how schedules did not line up and how each person's coding style is different. It was quite an adjustment as before this

we were only used to working with our own code, but for this project you had to adapt and adjust your implementation to match what is required by other people's code.

- Using version control
    - Git was a nightmare to use at the beginning. We had no idea how to use version control so it took us by surprise. We initially decided to each create a branch to work on so main wasn't constantly being updated and changed. This proved successful and was a smart decision to implement. One thing that really got us though was merging. We were so scared at the beginning of overwriting the master branch or messing it up that we were manually copying and pasting code from our branch that we opened up in a browser to the master branch and then pushing that. This was incredibly slow, painful, and unproductive but as the semester continued, so did our confidence with using git.