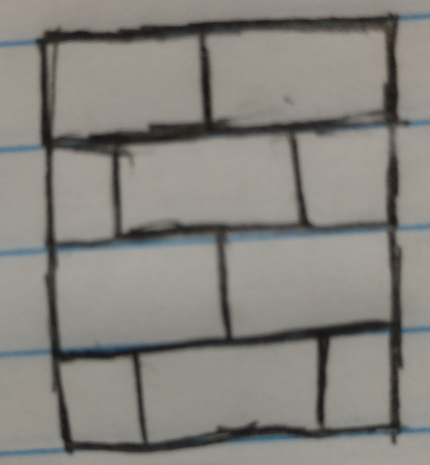
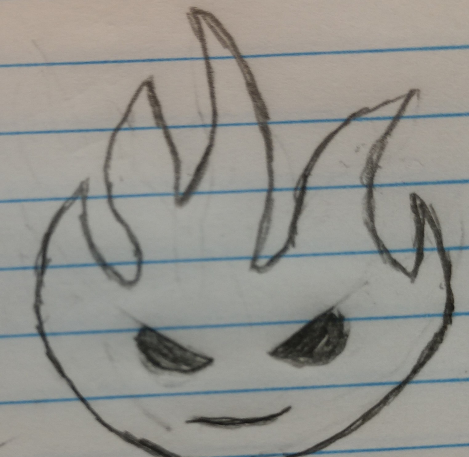
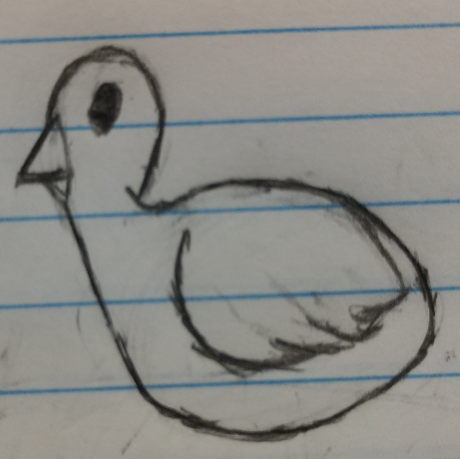
**CS 342 Project #5: Duck Maze   
 Design Document**



**Team 11 – 12:30 PM**

**Angela Cuevas**

**Aswin Manoj**

**Zeke Thornburgh**

**Overview:**

Our team project is a four-player networking game where players have to move a duck to the end of the maze safely. The maze contains wall and fires, and the duck is supposed to avoid the latter since touching it would cause the duck to go back to its starting location. Each player gets to see certain components of the maze/map (one player may see the fires, whereas someone else may see the location of the duck). With the unique clues that each client receives, the players are supposed to work together to decide where the duck should move. Its movement is decided by a voting system.

**The Goal (Section 1):**

This was a multi-purpose project. Firstly, software development is a team game, and this program allowed us the opportunity to practice our teamwork skills. Secondly, concepts like networking and threading are very important topics in programming, since many software applications use them. Thirdly, this project served to remind us of the importance of creating intuitive design patterns prior to coding on the text editor right away, which gave us a clear plan regarding what to do programmatically. Ultimately, the goal of this project was to submit a working program that simultaneously helps us grow in the three areas mentioned above.

Since this is a class project, this obviously does not serve society (or solve any of its problems) like software applications do in the real world. What it does, however, is that it helps us learn skills that shall benefit us in the near future. So, the submitters of this assignment are the ones who may go back to this project one day to recall the valuable insights discovered in this project.

**High-Level Entities (Section 2):**

The high level entities that were involved in this program were items like a duck object, a maze object, and the instances of the server and the clients. The Maze class contains an instance of a Duck, which gets placed on a particular position of the map. The maze object is in the game class, which is responsible for the logistical components of the program, such as distributing clues to each client. So, to describe the program in a high-level manner, the classes mentioned above have dependencies with each other. The maze needs the duck, and the game needs the maze, so in turn, the game needs the duck indirectly. The reasoning behind the entities here is that server and clients need to be instantiated in order for a network connection to arise, and the duck is technically a part of the maze, since it has to move through it. The maze is part of the game itself, so logically it makes sense to include it in the game class.

**The Low-Level Design for each Entity (Section 3):**

The duck object has a char variable that is assigned to a dollar symbol. This was made so that we could print out the maze on the console/terminal and examine the integrity of the map for debugging purposes (because it would be harder to debug on the GUI). We first tried printing the duck on the maze itself and tried moving it around in an infinite loop to see how it worked. The maze was then drawn accordingly on the clientFX using Java’s built-in GUI system. We also used a lot of encapsulation, especially by using getters and setters in the duck class for integrity purposes. Field members can never be public. Although it was tedious to make a getter and setter for so many field members, it was necessary.

**Diagrams/Models:**

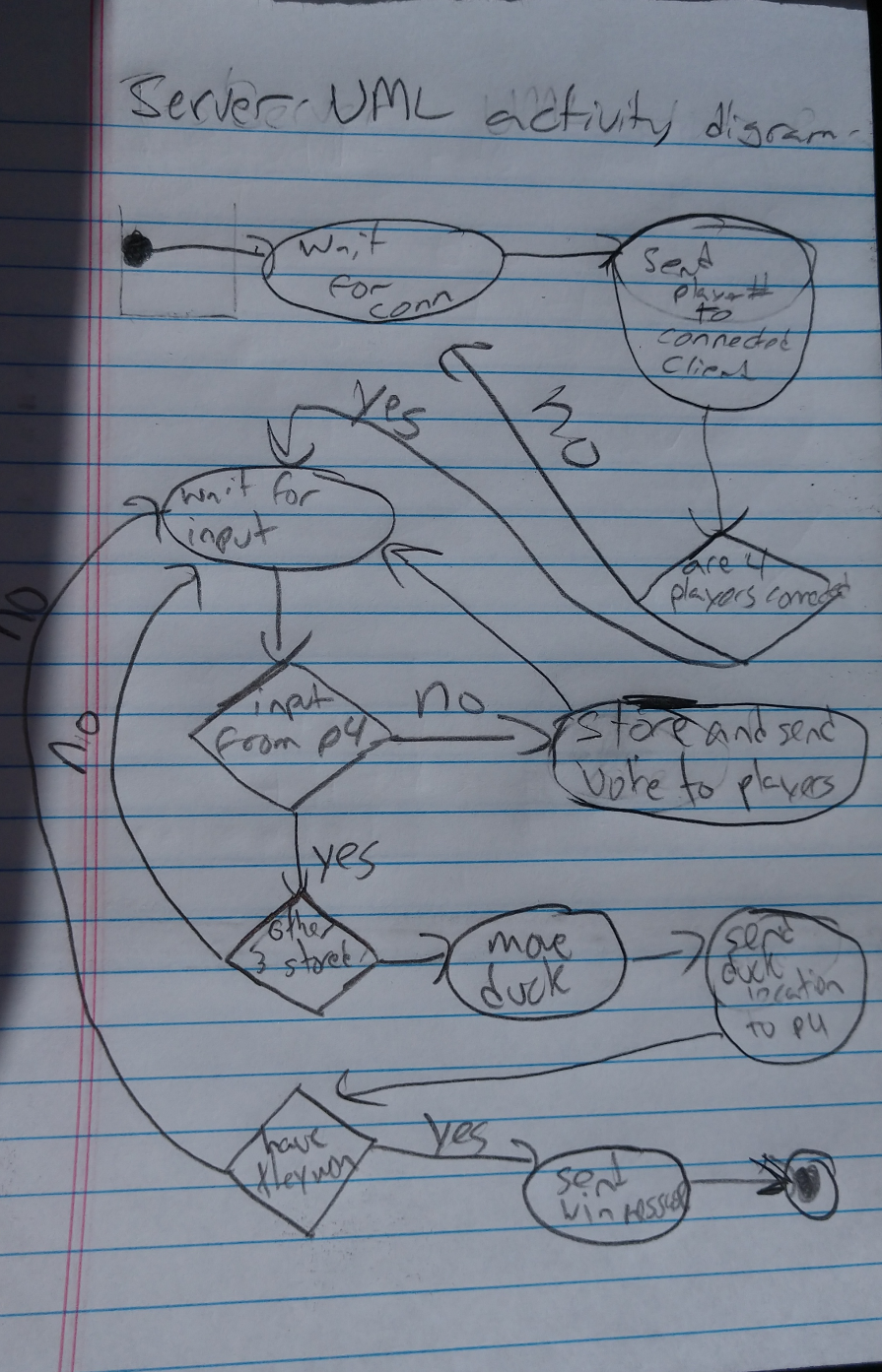


Figure 1: A UML activity diagram describing the server and its relationship with the clients

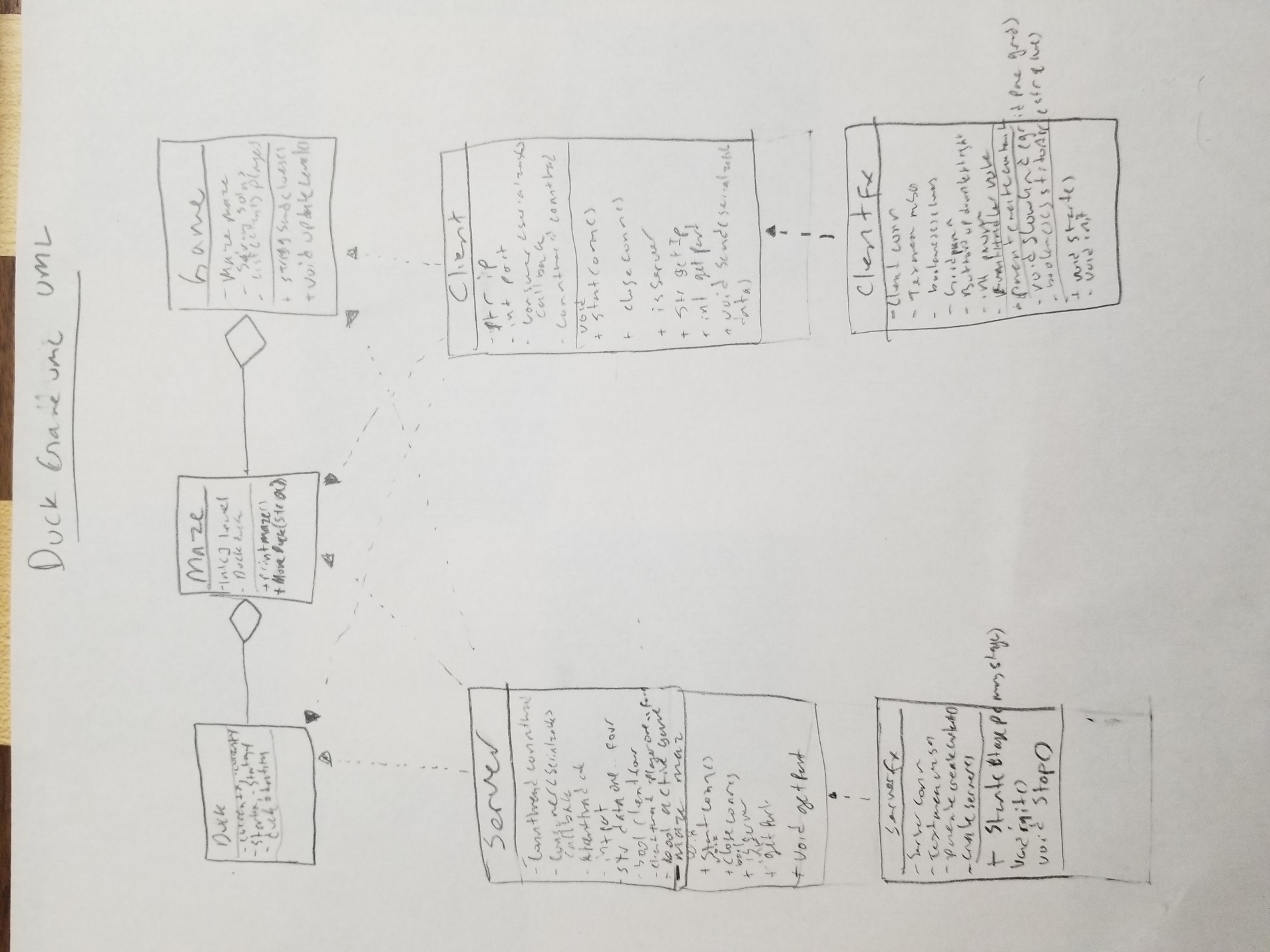


Figure 2: The UML diagram of the Game itself, highlighting the key classes involved in the program

**Benefits, Assumptions and Risks (Section 4):**

Top 5 benefits of our plan:

* Intuitive design that follows the principles of Java, such as encapsulation and inheritance
* The choice of JavaFX over Node.js (since we knew the former a lot more due to our previous experience with it)
* The dependencies between instances of classes. We could have easily written everything in one file, but that would not have been an easy-to-maintain software design choice. For example, the duck logic could have all been inside the maze, but it was more professional to make the duck its own class.
* Testing the maze on the console – This saved us a lot of time that we would have spent debugging on the front-end/GUI. Printing the whole maze and the duck out to the console was easier, since that did not require the intermediate steps of reflecting it on each client via the clientFX mechanisms.
* Dispersing unique clues to each player – Each player gets to decide which move is the best by seeing the clues.

**Assumptions:**

* We initially thought we were going to use Node.js for this.
* We thought about adding more maze levels, and we thought that was going to come to fruition

**Risks:**

* Node.js would have taken too much time, so we decided to stick to JavaFX
* The new maze levels would have taken more time, and due to the time constraints with finals and other classwork, this did not happen. We calculated the risks associated with the lack of time, so we omitted the two points mentioned above.
* A current risk is that we may not have programmed it too defensively. What if the player tries to vote several times in a row? Will there be any glitches? There may also be other loopholes that some users may be able to exploit.

**Conclusion:**

This project involved a lot of change of plans. During the first week of this project’s release, our group had not met up or accomplished anything due to how hectic the last few weeks of the semester were. Eventually, as we started to meet up to work on the program more, each person did the items that they were responsible for on the sprint-sheet. After a few hours of brainstorming and debugging, we eventually saw a game that displays the maze GUI for each player. To make the software design smoother, we chose to uphold the OOP principles of encapsulation and inheritance.