**Software Implementation and Testing Document**

**For**

**Group 2**

Version 2.0

**Authors**:

Melanie B

Harrison G

Julio S

Alora C

Jacob P

# Programming Languages (5 points)

We are using C# within the Unity Engine to create our game, the Labyrinth. With the objects in our rooms, such as lamps and couches, we attach scripts to some of them to allow the player to interact with them. This allows them to learn more information about the room or try to solve puzzles leading to the next area. For example, there is an envelope on a small table next to the front door of the living room. Clicking on it allows the player to read the letter in it, giving them a clue to unlocking another area of the game. The clock on the wall is interactable, and adjusting the hands to the correct time is how to unlock the puzzle. We use our C# scripts in order to perform these tasks and allow the player to further interact with the environments.

# Platforms, APIs, Databases, and other technologies used (5 points)

We are using the Unity platform and the UnityHub application with built-in GitHub support to collaborate on our project. We use the GitHub desktop app to push and pull any updates to our game, allowing everyone to have access to the most updated version of the game. Our scripts rely on the UnityEngine library. Harrison used the iPad Pro app Procreate to draw assets. Melanie used the desktop application FireAlpaca and the mobile app ibis Paint X to draw sprites. Coding for scripts was done in Visual Studio and Visual Studio Code.

# Execution-based Functional Testing (10 points)

We adapted a test-driven approach where after every implemented feature we exhaustively tested the scene and all items, making sure the behavior was identical to expected behavior. After implementing a puzzle, the team would pull the new version and check that everything was working fine. For example, Melanie, Julio, and Jacob were in a meeting creating the clock puzzle with Melanie sharing her screen. We had issues getting the code to return the correct time, so Julio and Jacob kept giving suggestions as to what to change about the equation. Eventually, Jacob found the correct solution and changed the equation. Once Melanie opened the game, she confirmed it was working properly.

For the fridge magnet puzzle in the kitchen, Melanie worked with Harrison and Julio to properly connect all the magnets and the interface to the code. Once it was finished, Melanie’s Unity application froze in the middle of testing, so Julio tested it instead while live streaming it to the group. He ran the game, clicked on the fridge, and tried several combinations of letters for the anagram. Once he put in the correct word, the game successfully closed out of the puzzle and displayed the corrected sprite.

# Execution-based Non-Functional Testing (10 points)

We used Unity’s built-in play mode to test the application during development. When a feature was implemented, we deployed a Windows build and tested on a Windows 10 machine. To make sure the performance of the game was up to par, we used Unity’s profiler which analyzes execution time frame-by-frame.

One of the most important non-functional requirements is the ability to move in between rooms and not have the room reset to its original state. Harrison tested this function by completing puzzles in one room, exiting to another, and going back into the original room to see if the puzzles were still completed. He did this multiple times to ensure it was not a fluke or one-time success.

# Non-Execution-based Testing (10 points)

We had regular meetings on Discord where we would implement features and one person would share their screen while the team provided feedback and reviewed the code. We also reviewed code asynchronously by posting code snippets on Discord and checking changes using GitHub’s diff viewer feature.