Architecture

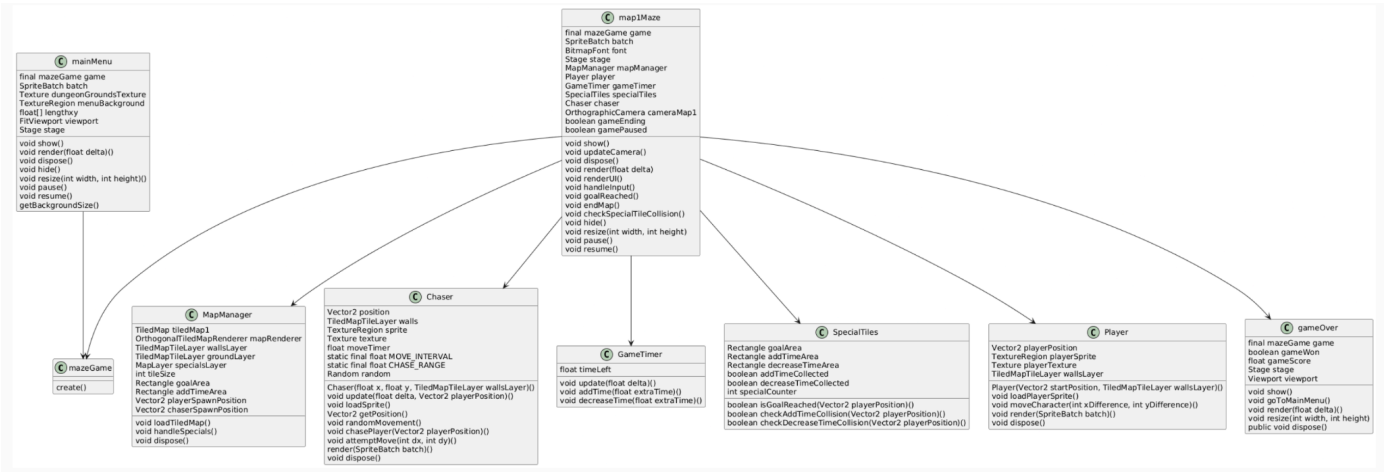
Group Name: Team 10
Group Number: Cohort 1 Group 10

Names:
Haaris Altaf
Casper Co
Will Garavelli
Kamso Osuji
Krishna Rajamannar
Diyar Savda
Sam Storrs

Structural and Behaviour Diagrams of Architecture

Below, we've attached images of the structural and behavioral diagrams that outline the overall architecture of the project. These diagrams were written using UML (Unified Modeling Language), and we used PlantUML as our tool to create the diagrams.

UML Structural Class Diagram



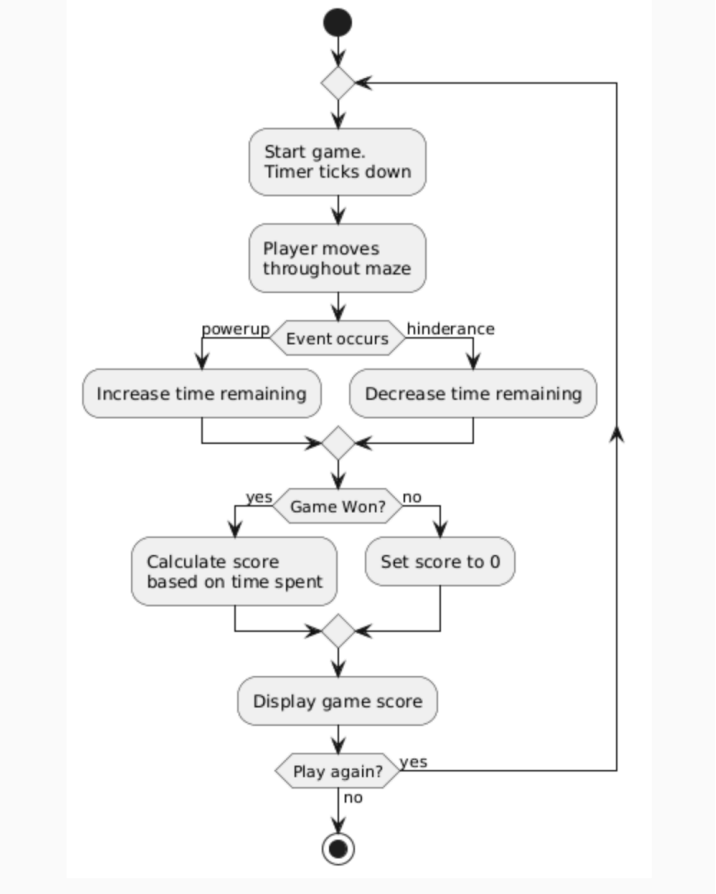
In this UML class diagram, we have eight classes; mainMenu, mazeGame, MapManager, Chaser, map1Maze, GameTimer, SpecialTiles and Player.

The class mainMenu represents the initial screen that users will see once they run the product. It implements the screen interface from the Java game engine LibGDX. This class uses the mazeGame class which extends from the Game class (also provided by LibGDX). The mazeGame class just essentially assigns mainMenu as a screen and ensures that it is the first screen to be displayed once the application runs.

The class map1Maze also uses mazeGame. This class is used in map1Maze to display the main menu once the game has ended. The class map1Maze also uses other classes which are smaller and represent different components of functionality. For example, the Chaser class represents the enemy which is constantly following the player and this class is used by the map1Maze class. Also, the gameOver class is used to display the game over screen when the timer has finished or when the chaser has caught the player.

MapManager represents the logic for rendering the tiled maps and it is also used by map1Maze to render the map and handle the special tiles. The SpecialTiles class is used to check whether a player has landed on a specific tile which could trigger an event. The Player class represents player movement and the initialization of the player sprite.

UML Behavioural Activity Diagram



In this UML activity diagram representing the behaviour of the game, the architecture of the behaviour within the game is sequential, such that the user is encouraged to continue playing again after they have completed the maze or their time has run out. This is to achieve the client's goal of getting the most players to play as much as possible for potential monetisation of the game. From the start, the game will be ruled by the timer; when it runs out, the game ends. In addition, when events occur, they can impact the time remaining in a negative or positive manner. The behaviour of the game is managed by the code in such a way that the timer has its own file called GameTimer.java that is accessed by the logic to detect when the game should end, while being kept separate to avoid being in the way and make the code easier to read when other people or teams of people are working.

Architecture Design Process and Justification

In terms of the overall architecture, we decided to implement a layered architecture, consisting of 3 layers: Presentation Layer (UI), Business Layer (Game logic), and an Infrastructure Layer (rather than a Persistence Layer or a Data Layer). We decided to use this architecture as it is easier to both understand and implement, thus promoting a separation of concerns. We also felt that it would be easier to test the game with this architecture compared to other architectures as we could test and also debug each layer independently from one another. Furthermore, we felt as if this architecture is quite scalable as well.

In the first iteration of our architecture design (which can be viewed on the website under the Architecture section), we initially had 3 classes: mainMenu, map1Maze and mazeGame. We decided to create a new class for each map that was implemented as we could easily just reuse the same classes that were already implemented.
As per the image displayed on the website, our first iteration had a lot of variables and methods within the map1Maze class, such as playerTexture or loadPlayerSprite(). We realised that the map1Maze class was becoming too large and was incredibly difficult to read and understand. To improve this, we decided to split the class into several smaller subclasses instead based on the variables and methods that can be grouped. An example of this is moving the methods loadPlayerSprite() and moveCharacter() into a separate Player class or moving the method CheckSpecialTileCollision() into a SpecialTile class. We did this for the sake of improving code modularity and our ability to test each component.
This iteration of our design ended up becoming the final iteration for the product, which can be seen above in this document as a UML class diagram.

Justification via reference to Requirements

For each of the requirements, the following explains how they are completed with the architecture of the code.

Functional Requirements

Game Engine - The game engine is accessed by the main code to pass in moves and receive back the visual display. It is used throughout the code, so it is accessed in all of the main files. This is noted by the importing com.badlogic, with extensions for the specific uses. The implementation of the game engine this way allows for many of the user requirements to be met. Hence, the Pausing_ user requirement is met by managing a user input of ‘ESCAPE’ on the keyboard within the code and then interpreting this with the game engine’s key-pressed function before changing the logic to state that the game is paused. The architecture of this integration is paramount to the functionality of the game as a whole to meet the requirements laid out by the client.

Powerups and Debuffs - By structuring the architecture of the system to manage events individually, when a user input should cause a powerup or debuff, this will be managed such that game logic will decide upon the necessary effect and display this information in the visual display using the game engine. The powerups and debuffs involve adding and subtracting time from the player. This is detected in the file SpecialTiles.java within the functions checkAddTimeCollision() and checkDecreaseTimeCollision(). The time is then added or subtracted using the file map1Maze.java using gameTimer.addTime() and gameTimer.decreaseTime().

Score - The score is calculated after the game has ended as a function call in the gameOver.java. It is first determined whether the game has been won with the gameWon boolean, and if it is decided that the game has been won, then the gameScore will be calculated by dividing a constant by the time spent, such that the less time taken, the higher the score.

Score Display - The score is displayed on the end screen managed using the file gameOver.java. After it is calculated as mentioned above, it will be displayed between ending the game and starting a new one; this is managed in the show() function.

Play Again - Within gameOver.java, the function goToMainMenu() uses the class attribute of mainMenu to start a new game without carrying any of the data from the old game across, as this could disrupt the new round rather than being completely independent as is intended. The overall structure of the code is modular, such that each game is run separately from one another and is implemented with a central file map1Maze.java. The functional components are then accessed from other files, such as the chasing dean being managed in Chaser.java and the game timer being managed in GameTimer.java

Non-functional requirements

Map - The movement within the map is managed in map1Maze.java and the map itself is made using an application called Tiled. It allows one to create maps using tile sets that one can design; this is then stored as an asset to be accessed by the logical code.

Specifications - LibGDX can be used on all mainstream operating systems, such that the game can be played by the most players possible as dictated by the client. There are no parts of the system architecture that restrict players from enjoying the game in whatever way they have access to.

Storyline - The building of the updateCamera() function in map1Maze.java allows the game to follow a progression that follows the player through university life as intended. The assets also hold a design of the map, which is modelled after the university to follow as closely as is appropriate.

Difficulty - The difficulty of the game is controllable because of the architecture of the code, the map being centralised means that for the game to have different difficulties as the client indicated may be desired in the future, this could be managed within the map assets without affecting the functionality of the game as a whole, rather than just changing the map as you wish. Having the map stored separately also means that a new difficulty (map) could be tested by simply designing a new asset and running the existing code to handle the logic of the game.

Controls - The handleInput() function manages the controls, which are ‘W’, ‘A’, ‘S’, and ‘D’ on a keyboard, because that is what most students are familiar with. This will allow the most users to play the game without instruction or consuming time learning.