Architecture

Team Name: Team Ⅳ

Cohort 1, Team 3

Ari Kikezos
Arya Enkhnasan
Ben Green
Calum Wright
Lilac Graham
Skylar Garrett

Structural Diagrams

We utilised structural diagrams within the project in order to effectively design how the components within the game will function and be structured. Using diagrams allowed everyone working on the project to understand the different components within the game without having to look through lengthy code. Furthermore, these diagrams were developed with the Unified modelling language (UML) in order to keep their notation standardised and easy to understand.

Class Diagrams

Why did we use Class Diagrams?

Class diagrams were a very important diagram when developing the architecture of the game. They allowed us to think about and design the different classes structure, attributes and methods that would be required in order to make our game function. Furthermore, these

diagrams allowed us to represent key programming concepts such as inheritance and relationships which determines how different classes work together. Class diagrams supported communication between everyone on the team as they were simple and easy to understand, making them a great tool for a game development project where tasks were split up between multiple people in different areas of the project. By considering classes and their attributes and methods, code generation was more efficient and allowed more consistency within naming conventions and functionality.

Class Diagram Development Tools

We experimented with numerous tools and methods of creating class diagrams. We started with a whiteboard and pen as it was the most effective at generating ideas and allowing communication within the group during team meetings. To formalise these ideas, we started experimenting with tools such as draw.io and StarUML which were good tools for developing interim versions of our class diagrams however we found that PlantUML was the most effective in creating clear representations of the classes and relationships within our game. It also made editing classes or inserting new ones much easier due to it being text based creation rather than drawing based creation.

Development of Class Diagrams

Throughout the project, the class diagrams changed and evolved as requirements were made clear with the customer and as we gathered a better understanding of the game engine LibGDX which we used to develop our game.

Using the initial brief, as a team we brainstormed potential superclasses and child classes and how they would interact with each other as seen in the first sketch and the second sketch. These ideas were generated by picking out specific requirements from the brief such as events and subsequently good, bad and hidden events. Other key ideas we picked out were a map/maze, a player, and some sort of timer or dean that the player has the escape from.

The meeting with the customer allowed us to clarify the brief and develop user, functional and non-functional requirements as seen in the Requirements document. As seen in the second draft, we introduced different types of screens such as the super class Screen and the subsequent child classes of PlayScreen, StartScreen, PauseScreen and EndScreen. These requirements were discussed during the meeting so it was important to include them within our class diagram and eventually our game. We also started to consider what our BadEvent could be since we discovered that it had to be something that would negatively affect the players ability to go through the maze, therefore we decided the player would have to collect a key in order to unlock a door to reach the exit of the maze.

The next change in our class diagram came after studying and learning how libGDX works. This allowed us to create classes and relationships between classes that would definitely be prevalent within the final game. During the early stages of the third draft, no methods or attributes were included however they were discussed during meetings, specifically with the implementation team which allowed us to create the final class diagram.

Justification of Architecture

Entity Class - The entity class included the logic for any object within the game such as the player, the walls of the maze, collectable items and all other events. This class implements the functional requirements of FR_MOVEMENT AND FR_INPUT which allows the player to move through the maze and also FR_COLLISION which was an important requirement as it made sure that the player couldn't pass through the walls of the maze and could interact with different events.

Event Class - The event class inherits from the entity class as it functions like all other entities which have logic such as FR_COLLISION. However, the event class allows additional functionality to be defined which comes under UR_EVENTS and FR_POSITIVE_EVENT, FR_NEGATIVE_EVENT and FR_HIDDEN_EVENT. This allows the user to collide with the different events and have something different happen depending which one the player collides with.

Timer Class - The timer is another requirement within user and functional requirements (UR_TIME and FR_TIME). This creates the element of time within the game which forces the player to have to complete the game by reaching the exit (FR_EXIT) within 5 minutes otherwise they fail and the end screen comes up (FR_GAME_OVER).

Utilities Class - The utility class is dependent on the Entity and LayoutPos classes as it uses them in order to load the map (FR_MAPGEN and UR_MAP) as well as centre on cell and wrap text such as UR_SCORE, UR_TIME, and pause and end screens and display using the **LayoutPos class**.

Main Class - The main class is dependent on the utilities class to do things such as load the map and display text. The main class has composition relationships with the Entity, Event and Timer class as those classes can't exist outside of the Main class. This is due to the main class initializing all the different entities and events that exist within the game and handles the different game logic.

Behaviour Diagrams

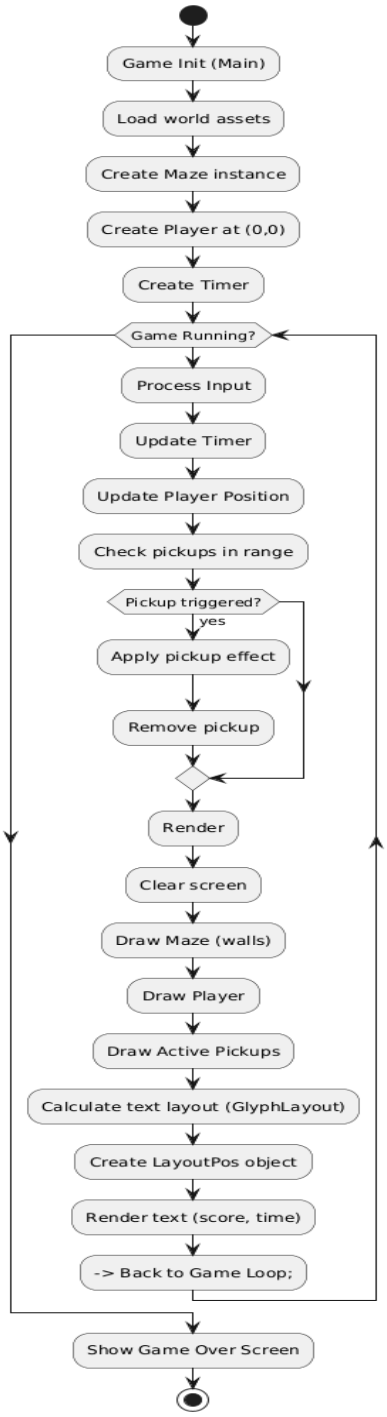
Why did we use Behaviour Diagrams?

Behaviour diagrams were another important diagram when developing the architecture of the game. They allowed us to model how different elements of the game would interact with each other and with the player. If class diagrams were a tool for visualising the building blocks of the project, behaviour diagrams were for seeing how those blocks would fit together. This includes general game flow, event logic, and player input, as outlined below. We used activity diagrams to outline logic for the larger sections of the game or the entire game whereas we used state diagrams to capture specific logic in classes such as the timer class.

Behaviour Diagram Development Tools

Like Class diagrams, we decided the best approach to designing behaviour diagrams was to use PlantUML. This allowed us to make clear and well designed diagrams that could be understood by everyone involved within the project including the customer. PlantUML also allowed easy editing which allowed us to develop on and change interim versions of the diagrams.

Full Game Logic



The final diagram can be split into four pieces: initialisation, game loop, events, and game over.

In the initialisation section, the game is initialised and the maze and player generated. This helps satisfy UR_TIME (the timer that decides if you win or lose begins), UR_SCORE (the score starts changing), UR_MAP (a map is generated), and UR_UX (the interface appears). The timer logic is given in more detail later in this document.

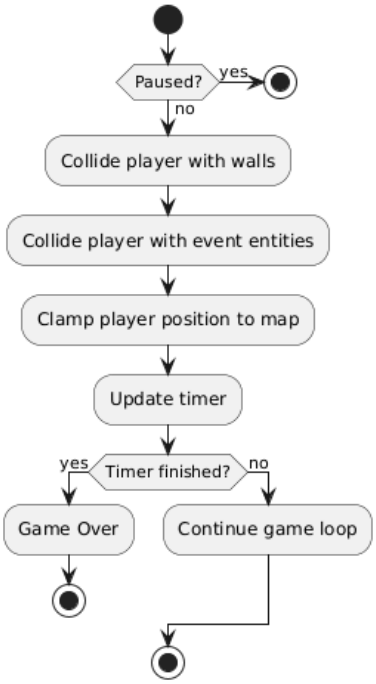
Now the game can begin. In the game loop section, user input is processed for movement, the timer is incremented, the score updates, and (optionally, as described later) an event occurs. These user-obvious processes are supplemented by a series of rendering functions. This phase helps satisfy UR_MOVEMENT (the user can control the character) and UR_AUDIO (the music plays if the game is running) user requirements the most but also adds to some aforementioned ones too. The user movement logic is given in more detail later in this document.

Optionally, during the game loop, an event will be triggered resulting in a change to the score or the map or the player. This most notably satisfies UR_EVENTS (events occur), UR_STORY (events are part of a story), and UR_AUDIO (sound effects play on pickup).

The game ends when the timer runs out or the end is reached. The timer running out is considered a failure state and the exit being reached and the player escaping is a success state. This satisfies UR_MAP (the end point is at a place on the map), UR_SCORE and UR_TIME (you lose if the timer runs out), and UR_ESCAPE (you win if you escape to the end).

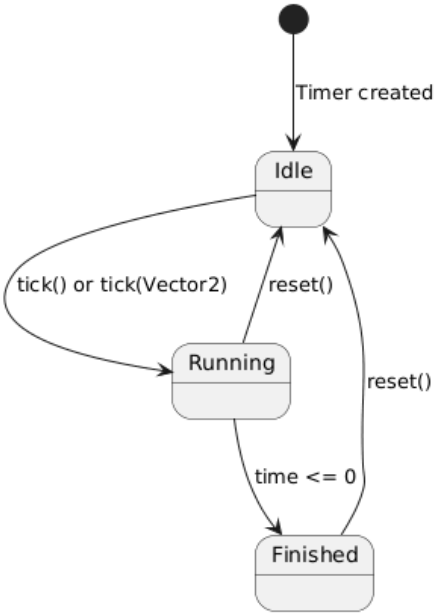
This diagram went through several changes in its development. As the diagram evolved from a napkin sketch to a draft to its final form that you see above, the most notable improvement was that it got less complex. As the intention of writing diagrams like this in a standardised manner is so that team members, clients, and other third parties, making the diagram as simple to understand as possible while keeping all of its necessary and descriptive parts is a priority.

Activity Logic



This very simple diagram outlining whether a player is colliding with anything and whether the game has finished or not had one sketch before the final render. One other form of this diagram which was not entirely implemented was the player state diagram, a draft for a more complex player system which got cut due to time constraints but helped us realise how the system should work.

Timer State



This very simple timer implementation would allow the game to have a start and an end with the ability to change the time per second and to apply time bonuses or penalties. A version of this structure is evident in the final implementation.