# Graduation Gauntlet

Architecture

Cohort C2 Group 8:

Ben Burniston

David Cavanagh-Lancaster

Hamza Saqib

James Owens

Michael Kapustin

Sanan Wazir

Shubham Gurung

# Architecture

## Languages, Tools and Assets

We used the programming language Java, and the open source community driven codebase LibGDX for game development. LibGDX provides comprehensive tools for rendering, input handling and asset management. LibGDX is the foundation for managing the game loop design pattern. We leveraged its included game life structure to efficiently manage the initialization, updating and rendering phases of the game, allowing us to focus on implementing game logic and features rather than low-level engine details.

Another tool employed was Tiled (https://www.mapeditor.org/), a tile map creator. **Our project is not dependent on this tool  and is not required to create a valid map layout but greatly simplifies the process of creating a map**. Tiled, when given a tileset, assigns integer values to each tile and provides a visual interface to create a tilemap. Then it exports it directly as a csv file in a grid (length,width) format. This csv file is added to the project's asset where it is findable by the project and is parsed into a playable tilemaps where the integer value directly associates to a particular tile type.

## Design Process

### Figure 1



| Map | Path | SOF1 Retake (Event) | Glasses On Bob(Event) | Staff Landyard (Event) | |
|---|---|---|---|---|---|
| A collection of tiles and walls which the player will be able to navigate through, in a grid-like structure | Any tile that the player can stand on top of and navigate through | Triggered event which requires the player to backtrack<br><br>Negative Event | An item needs to be collected from one area, and moved to another area for the player to proceed.<br>Negative Event | Due to collecting an item or a trigger of other means, the player can go undetected by the enemy or get powered up to help proceed to other parts of the maze<br><br>Positive Event | |
| Tiles | Wall | Tuition Fees (Event) | ChatGPT (Event) | Pause | Tutorial |
| Individual boxes that form the grid that makes up the map. Certain tiles may have specific attributes | Any tile that the player cannot cross | Due to a trigger of sorts, coins could be scattered around the map where the player needs to collect them before proceeding<br>Hidden Event | Due to a trigger, the player is able to have protection from a negative event<br><br>Positive Event | A method which would stop the game for however long the user requires | A menu or screen of sorts showing/telling the player what they need to do |
| Exit | Lucky Tile (Event) | Surprise Time Table Change (Event) | Enemy (Event) | Game Over | |
| A tile which allows the user to leave the area, and end the game to a victory screen | A trigger that can cause a random event to occur such as score change increase<br><br>Positive Event | Due to a trigger, the player would be randomly teleported across the map, hindering their progress<br>Hidden Event | An entity of sorts which would chase the player around the map.<br><br>Negative Event | A screen which tells the user they have lost, and gives them to option to restart | |
| Bob Scare (Event) | Gamble | Player | Victory | Playing | |
| Due to standing on a particular tile, Bob would stop the user from being able to proceed<br><br>Hidden Event | A list of events that can occur provided a trigger given | The character the user controls with set inputs to navigate the maze | A screen which tells the user they have won, gives them their score, and the option to restart the game | A state which allows the user to move around, entities to move and the game to continue working | |
| Check-In Code (Event) | Quizzzz (Event) | Controls | Score | Timer | |
| The user needs to enter a check-in code as an input in order to proceed<br><br>Negative Event | A simple problem the user has to solve before continuing<br><br>Negative Event | Method for the player to move around and interact with items | Directly related to the timer - its the time they had left over after completing the maze. Events can change the score | A count-down from a set time in which the player should try to escape the maze by - directly relates to the score | |

**Figure 1** shows the CRC cards we created during the initial design phase, after we elicited user requirements. These cards represent the distinct ideas and concepts for the game, helping us organize and visualise the systems potential classes, responsibilities and interactions. From this collection we selected a few events to add to implement into the first iteration of the game:

Red- Core game object, Green - Event, Orange - Implemented Event, Grey Event

## Structure Architecture

Building on these concepts, we decided an object-oriented programming approach was best suited to implement these ideas. We grouped the CRC cards into modules, creating a mapping between each module, the requirements they tackle and their responsibilities, and the relevant classes. By doing this, we provide a conceptual clarity and we ensure the system abides to all user requirements elicited. As discussed further in the main architecture section, our design evolved iteratively resulting in the initial CRC cards being refined and adjusted to reflect implementation decisions and emerging requirements.

This made it clear to us that, for our implementation, a layered architecture was the most appropriate approach for our implementation allowing us to separate the concerns of the modules into clear defined layers. Layered architecture promotes high cohesion and low coupling, which helps maintain clarity, scalability, and ease of testing. Each module focuses on a well-defined responsibility.

**Figure 2**

| Module Name | User Requirements | Relevant Classes | Description |
|---|---|---|---|
| UI and presentation. | UR_TUTORIAL, UR_LOSS_SCREEN, UR_VICTORY_SCREEN, UR_HOME_SCREEN, UR_SETTINGS, UR_PAUSE_MENU, UR_FAMILY_FRIENDLY, UR_MAZE_THEME | Main.java | Manages user interface flow, and the visual presentation of what's on the screen. Rendering and interaction. |
| Core Gameplay Loop | UR_DIFFICULTY, UR_PLAYER_CHARACTER | Main.java | Manages player input, order of operations within the loop and responds to the state of the world. |
| World state / Maze Generation | UR_MAZE, UR_UNIVERSITY_THEME, UR_PLAYER_CHARACTER, UR_SCORE | World.java Tile.java | Handles game map creation and the state the game is in such as the time left and the current score. |
| TIle Actions | UR_POSITIVE_EVENTS, UR_NEGATIVE_EVENTS, UR_HIDDEN_EVENTS | TimedOverlay.java World.java Tile.java | TimedOverlay Manages all tile based interactions which require visual representation. The world class handles the logic of events. including positive, negative and hidden events |

**Figure 2**: A table with the modules, which requirements they meet, and which class primarily looks at them. Note: Not all requirements from the list of requirements are found here.

## Behavioural Architecture - Sequence of Events

The **Figure 3** below shows a table of the three events we have implemented alongside the respective type and description. Most of their implementation is contained within the "Tile Actions" Module. Classes TimedOverlay and World handle the actual logic and Tile holds the data. We have implemented 3 events in total. **Figure 4** shows the sequence diagram of how the three events are triggered. This diagram shows core functionality and is abstracted for ease of understanding.

**Figure 3**

| Event Type | Event Name | Description |
|------------|------------|-------------|
| Positive Event | Lucky Block | The Lucky Block is a type of tile found in the maze.<br>The Lucky Block will select from 2 random power-ups, them being: ADDSCORE and, REVEAL_MAP REVEAL_MAP uncovers the whole map and ADDSCORE will randomly add either 25,50 or 100 score to the players final score. |
| Hidden Event | Bob Block | The Bob Block is a type of tile found in the maze that cannot be seen by the player. Standing on the Bob Block tile causes a screen block, preventing the player from seeing where they're going for a set amount of time. |
| Negative Event | Enemy | The Enemy is an entity which, when triggered by standing on a certain tile, spawns and uses a tracking algorithm to chase the player down. Being caught by the enemy causes game loss. |

**Figure 4**
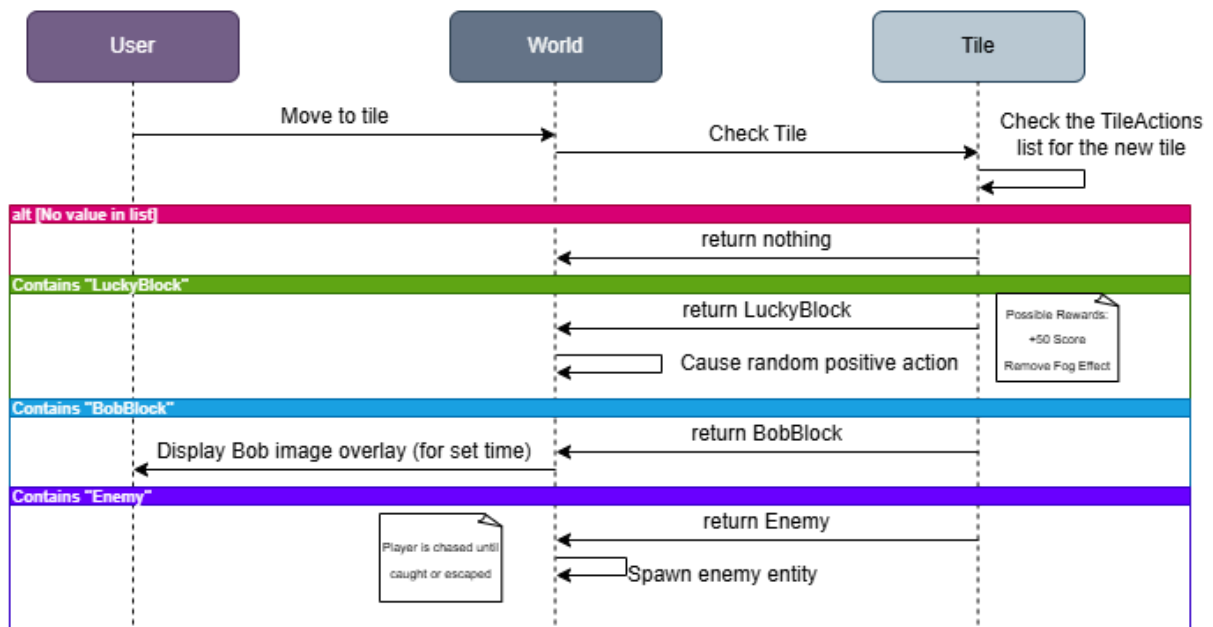


## Behavioural Architecture - States

**Figure 5** shows the distinct states the game can be in throughout the course of playing it. This follows our modular architecture structure, as within the gameplay loop, we further subdivide the individual high level states the player could be in, separating relayed code. Looking at **Figure 2** and the **Requirements**, the screens are controlled by the UI and presentation module, which is based around the Main.java class. Additionally, World state /

Maze Generation modules control which state the gameplay is depending on the Core Gameplay Loop module.

The game starts on the Home_screen in the Idle state which displays the settings. Pressing the spacebar changes the state from Idle to Playing, initiating the gameplay sequence and when 'R' key is pressed it ends the gameplay and returns the player to the Home_screen. From the Home_screen, players can also choose to exit the game by closing the game, providing clear control over whether they wish to continue playing or quit.

While in Playing, the spacebar can be pressed to pause the game, switching it to the Pause_menu state. Pressing the spacebar again resumes gameplay, returning to the Playing state. Pressing the 'R' (reset) key in either Playing or in the Pause_menu state resets the game and returns them to the Home_screen. The key also works on the Victory_screen and the Loss_screen states when the game is in the GameEnd state, allowing the user to go to the Home_screen anytime. In the Playing state, the game moves to GameEnd and the Loss_screen if the timer runs out, and the score is zero, or if the Enemy catches the player. Reaching the exit changes the game to GameEnd state and the Victory_screen.

This design highlights a degree of coupling between the input system and the Core Gameplay Loop, as specific keys like spacebar and 'R' directly trigger state changes. This coupling keeps transitions straightforward while ensuring that each input has a clear and predictable effect. There is also connascence between systems, since any change to a state name would require corresponding updates in the input handling and UI logic. This shared dependency helps maintain consistency across all game states and ensures that user actions always produce expected results.

**Figure 5**
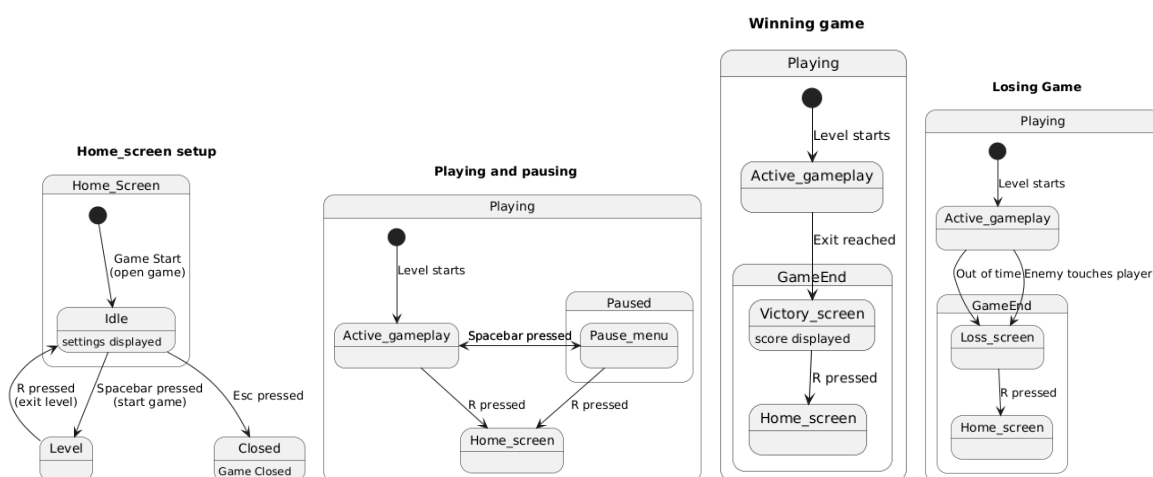


## Architecture Style and Evaluation

When we began our implementation of the project we started with a single monolithic OOP structure, contradicting our initial design thought process. This allowed us simplicity and speed in development, testing, and deployment, as well as easier debugging while we familiarized ourselves with development tools. This approach facilitated rapid initial

prototyping of the game to validate ideas early which laid the groundwork for later modular refactoring.

As our project grew in complexity we quickly shifted our architecture into the intended layered model in order to separate the sections of the game's core concerns presentation, the core gameplay loop and the world state. This is because monolithic is much harder to scale or maintain for multiple contributors. By switching to layered, we are able to introduce modularity into the implementation of the code creating loose coupling between the subsystems of the game - making it easier for multiple contributors working independently on separate sections of the system and . This architecture allowed our project to be scaled more easily, being able to add additional core layers to the game, without much concern of the other layers and how they are implemented. Layers could now be tested in isolation and made debugging much easier and our project now represented our initial design.

Next we implemented an object-oriented state design pattern to manage high-level game flow reinforcing our modular design principle. We did this using a custom State enum to toggle between game states within the world class. Each gamemode/state behaves differently encapsulating its own behavior such as configuring allowed inputs, what's being rendered and controlling the main update flow of the game. We embedded this system within the main gameplay loop, simply changing the loop functionality based on the current state active. This helped us easily implement some of the core requirements we proposed in the design of the project such as pause functionally, and various end screens. Furthermore this pattern supports future feature expansion allowing for the easy addition of new states with unique behavior and reduces code duplication.

One problem we encountered with this architecture was enabling communication between loosely coupled layers such as the world class signaling graphical operations for the presentation layer. To implement this we employed an event (observer) pattern through the use of an interface I_UI. This allowed the world class to trigger UI-related events without having any direct dependency on the rendering/presentation system. This design maintained loose coupling while enabling interaction across layers further enhancing the scalability and especially maintainability of the project.

Initially we implemented the tiles in an inheritance model with a single master Tile class which the different types of tiles such as Path, Wall or even event tiles like the lucky block would inherit from. This was proven obsolete as tiles like the wall and path were functionally very indifferent and we did not see it fit to dedicate a whole new class for. As a result of this we switched to a composition model where the tiles would hold TileActions and a TileType within the fields of the instance of the tile. This approach made tile types and actions data-driven and much easier to extend simply by adding to the existing enums. As tiles grow more complex requiring more parameters as the game evolves we can adapt this model into a builder pattern improving readability of our code and maintainability of the system.

To implement the events from tiles within the game we use an action pattern design where actions/events functionally are encapsulated into a key defining its behavior, in this case a struct TileActions. Upon stepping on a tile, the contained actions are invoked and logically carried out within the world class which handles the behavior of the actions. This system allows flexible, extensible behaviours for the tiles. This pattern allows for easy design of tiles

separating the data from the tiles to the logic in the world class and allows dynamic allocation of actions to instances of tiles. This system is easily extendable if tiles actions grow in the future.

One problem we recognised towards finalisation of the project is that each object which required a Texture instantiated its own texture in memory regardless of if this texture was unique or not. We recognised this was very memory inefficient and dived a solution, a centralized singleton GameAssets class which wraps around the provided LibGDX AssetManager class allowing unified access to the assetManager for the classes which required them. This ensures each asset is loaded only once and any feature requiring assets can simply request the reference greatly optimizing the project increasing performance to a noticeable level, particularly becoming more noticeable as the project size is scaled..

Future expandability requirements were met through the modular design of the layered structure, as interfaces such as I_UI allowed components to be extended or replaced without impacting other parts of the system. This architectural approach ensured that all user requirements were satisfied while maintaining flexibility for future enhancements, such as the addition of new layers (for example, a persistence layer). Throughout development, we employed an iterative improvement process prototyping features, testing, refactoring, and refining our implementation continuously. This allowed us to respond to emerging requirements, optimize performance, and improve maintainability while ensuring that each core addition we defined in the design process integrated cleanly with the existing system.

**Figure 6 - UML Diagram of the project (Clearer image can be seen on the website)**