Architecture

Group 2 - The Debug Thugs

Huxley Stevenson
Ben Leaver
Gergely Gal
Ali Tekin
Utkarsh Singh
Chihun Park
Faras Hbal

Architectural overview

Escape the Maze is a single-player, university-themed maze game built in Java 17 using the LibGDX framework. The player's goal is to escape the maze within 5 minutes while navigating through a mix of interactive events and obstacles.

The game follows a screen-based, layered, Model-View-Controller (MVC) architecture, the approach recommended by LibGDX, and was discussed in our software architecture lectures. The **Main** class manages the overall lifecycle and switches between the core screens: **MenuScreen**, **Firstscreen**, **SettingsScreen, LeaderBoardScreen**, and **WinScreen/LoseScreen**. Every screen handles its own logic and user interface, which keeps the code maintainable and easier to test.

During the early design phase, we used CRC cards to define class responsibilities and collaboration, following responsibility-driven design (RDD) principles. We then produced UML 2.0 diagrams using draw.io to document the architecture. This architecture was not static, and instead evolved iteratively when we adapted the project plan to delays. All the diagrams in this report link directly back to the requirements outlined in Req1.pdf, ensuring traceability and accuracy throughout the development.

Architectural Justification

As we are developing our game using LibGDX, we chose to base our architecture on a Screen Manager pattern, which works well with the game engine's framework. The pattern provides a high modularity, and a clear separation of concerns, which we believe justifies our architectural approach.

- **Modularity:** The 'Main' class acts as the central controller for the application, complying with the LibGDX lifecycle by managing the create() and dispose() methods. It is responsible for creating and switching between game states (e.g. FirstScreen, MenuScreen, WinScreen). 'Main' holds a reference to the active screen, but each screen is its own self-contained unit, as shown by our structural diagram. The Game State behavioural diagram also visualises this control flow. This design is very modular, making the code easy to manage, debug, and maintain. Therefore, the requirement NFR_CODE_MODULARITY is fulfilled.

- **Separation of Concerns:** Within the main structure, MVC-style separation is implemented via our primary gameplay screen (FirstScreen).
  - **Model:** 'FirstScreen' composes the core gameplay classes 'Player', 'Collision', and 'Events', which act as the 'Model' by managing game rules, state, and data. Our 'Player' State Machine diagram shows the separation in that complex logic (e.g. 'InteractingWithDoor') is entirely handled inside the model classes, not by the 'FirstScreen' itself.
  - **View/Controller:** 'FirstScreen' acts as both the 'View' and the 'Controller' by handling all rendering via its render() method, and also handling user inputs/directing the 'Model' components.

This separation heavily supported parallel development and traceability, allowing our team to work on core game logic (e.g. ‘Collision’ class) and UI (e.g. ‘WinScreen’) at the same time.

Evolution

The architecture of the project evolved during the development as design choices were changed for logistical reasons. To begin the design process we used CRC (Class Responsibility Collaborator) cards to identify what classes we needed and how they would interact with each other. Using a Responsibility Driven Design (RDD) approach, we identified the responsibilities of each of our main classes, such as Main, MenuScreen and FirstScreen, before defining the code structure. The use case created during requirements was also used to develop the sequence diagrams.

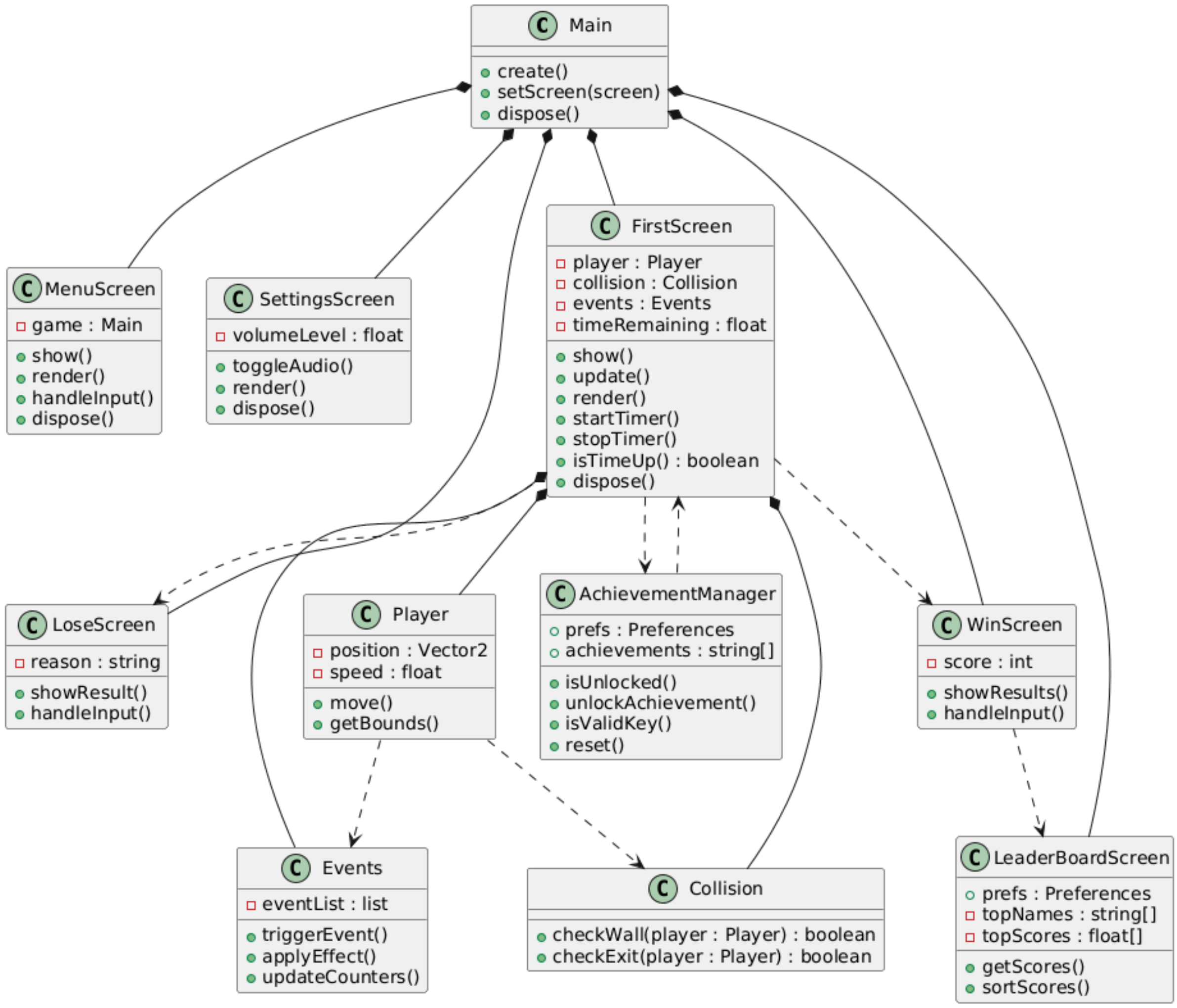
Initially, our plan was a rigid Waterfall cycle approach where our architecture design would be completed before implementation. However, due to delays in deciding our Requirements we shifted to a more agile plan. Our design and implementation were developed simultaneously. This meant our UML diagrams developed with the project and were changed iteratively instead of being fixed blueprints. Some changes we made from the initial design are:

- **Timer class** - We had a separate timer class in our initial architecture, this was merged into the FirstScreen class as the separate timer class created unnecessary complexity by adding more data passing between classes. This meant that time tracking became more maintainable and manageable by being incorporated into the screen.
- **EndScreen** - Initially, we had one EndScreen screen to display the end of the game (if the player won or lost). This was split into two separate, specific screens WinScreen and LoseScreen. This was done so messages could be more easily tailored to the player if they win or lose.
- **SettingsScreen** - A SettingsScreen was added after our initial designs as the project matured. The settings screen was an essential screen so users could interact with features such as toggling the audio of the game. This additional screen showed the change from a simpler prototype architecture to a more fleshed out design with more features.
- **Events** - The single events class has been decomposed into lots of different event classes, each one composing of its own self-contained event that link to FirstScreen to become interactive. This design choice made it much easier to create new events as each one was independent, making debugging and future testing much easier. These individual events have not been added to the UML diagrams as they all relate to FirstScreen and the player in the same way.

These changes were routinely built on throughout development and are shown through the updating of our UML structured and behavioral diagrams. These diagrams were updated to reflect architectural changes and to keep the documentation aligned with the code of the game. Full evidence of CRC cards and intermediate diagrams are available on the teams website: https://thedebugthugs.github.io/#architecture.

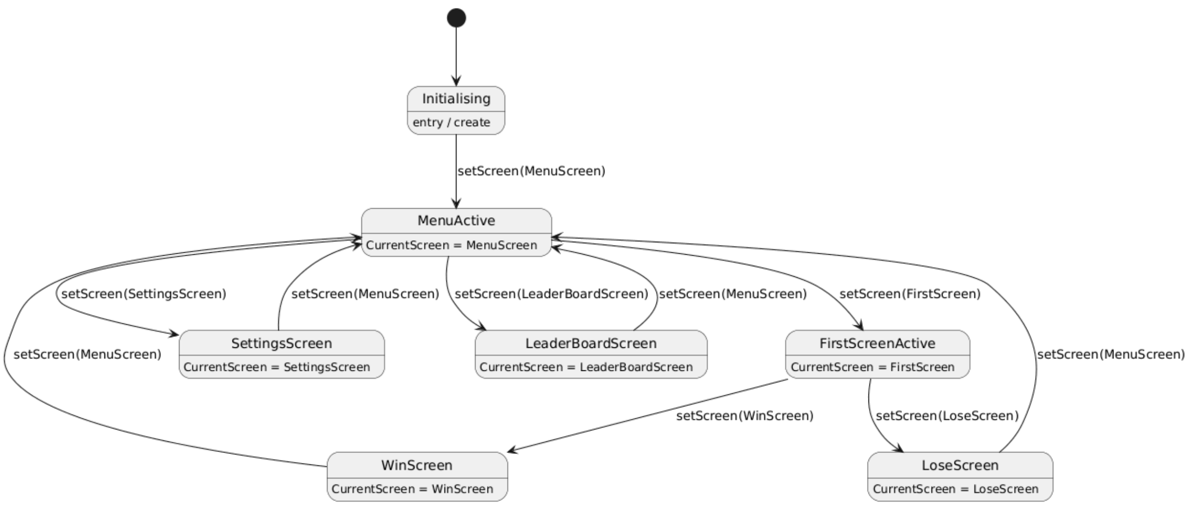
Architectural Diagrams

UML 3.0 class diagram showing the Structural View:

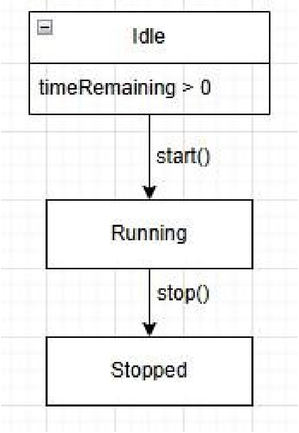


Behavioural Diagrams

3.1 Game state machine:

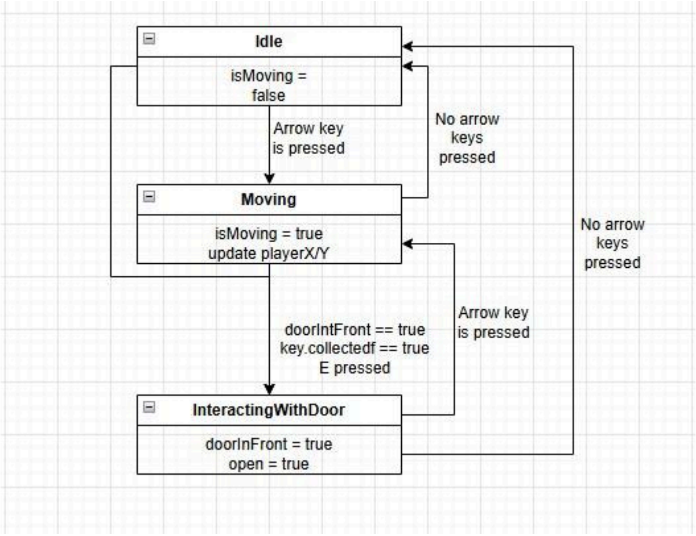


3.2 Timer state machine:

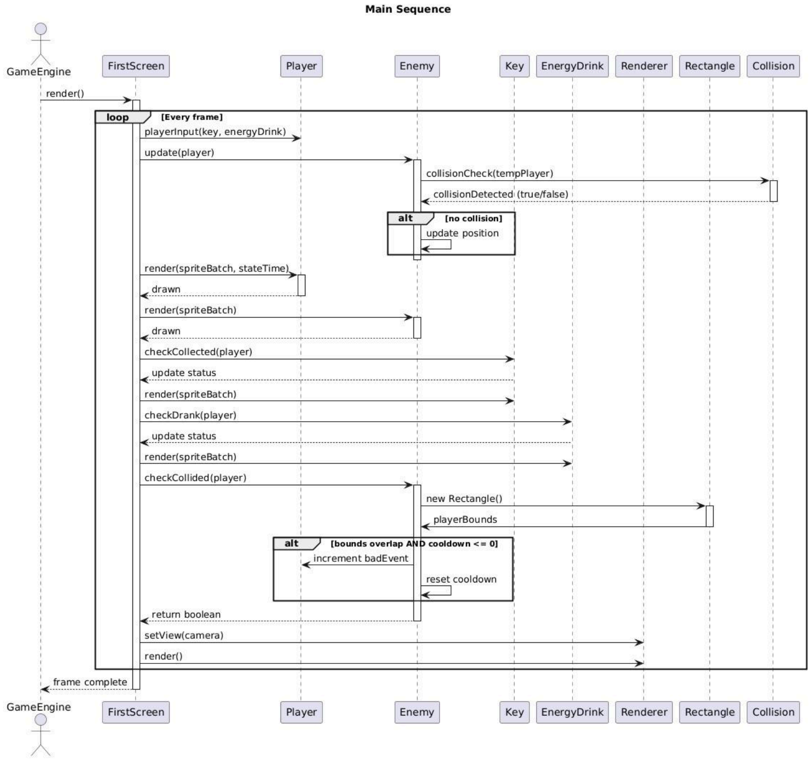


(Ended up being part of the FirstScreen class)

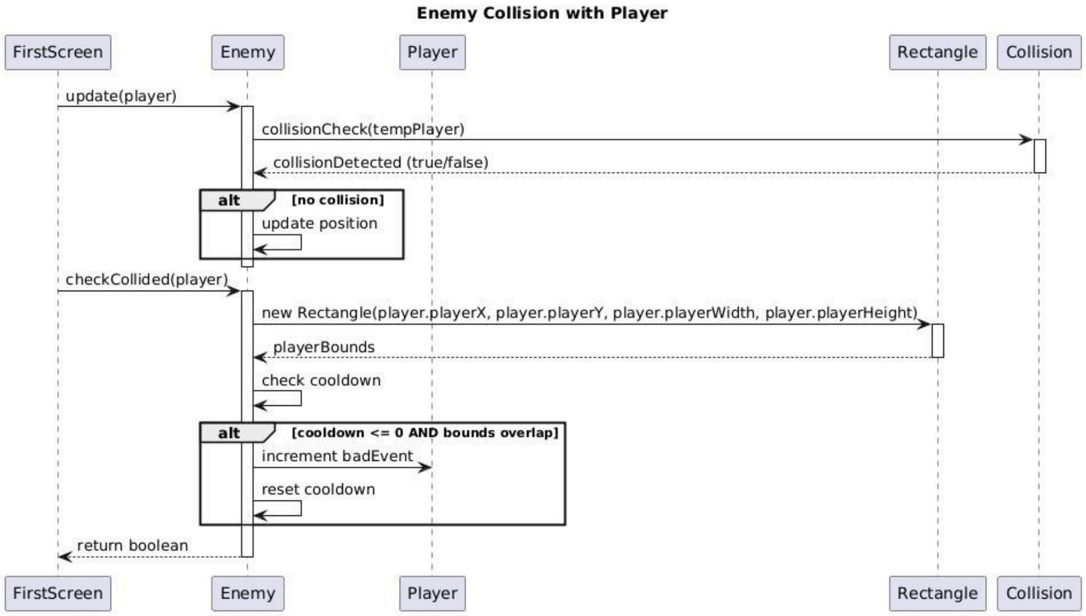
3.3 Player state machine:



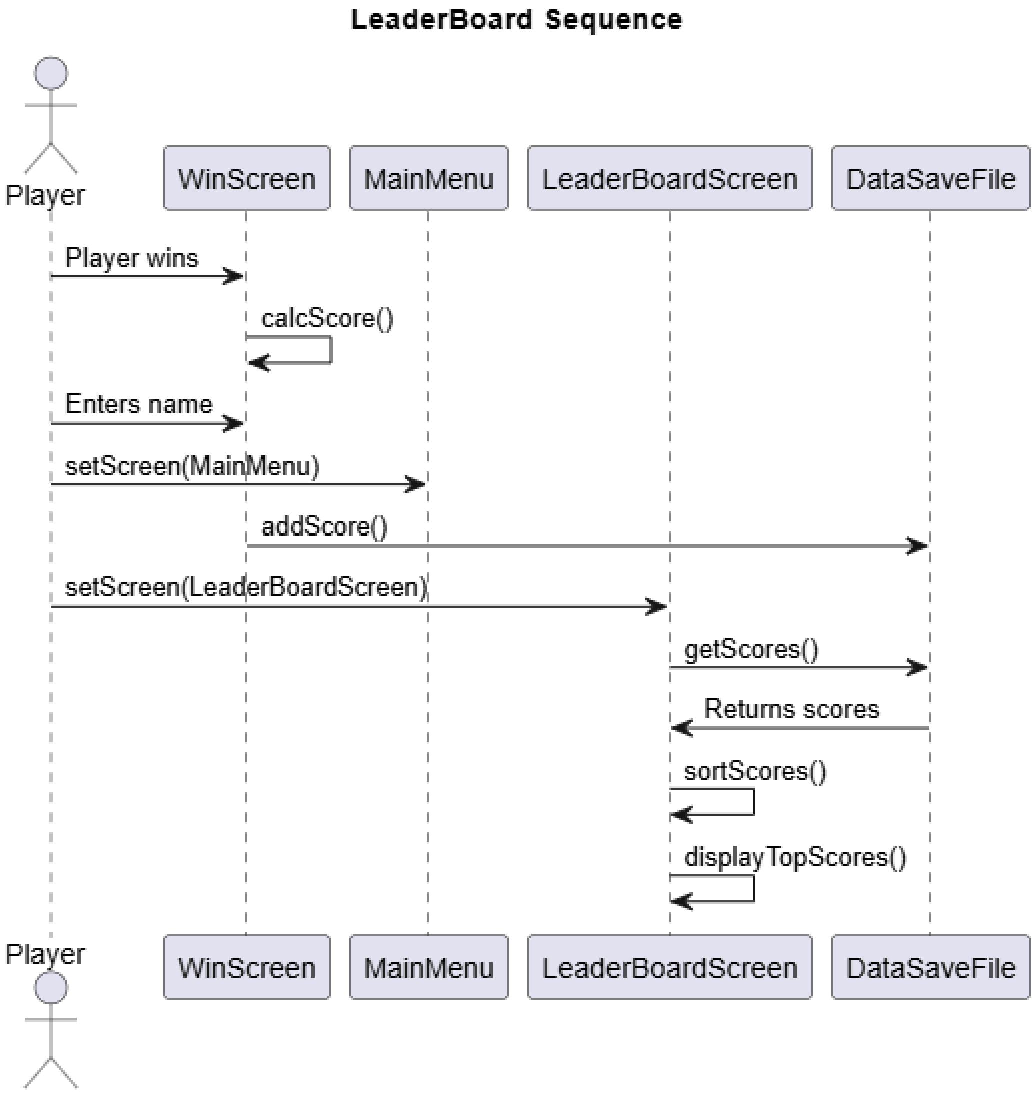
3.4 Main Sequence:



3.5 Event sequence:



3.6 Leaderboard Sequence



Requirements Traceability

The following table lays out our main architectural classes and their responsibilities, and traces them back to the requirements they satisfy.

Class	Purpose	Responsibilities	Collaborators	Requirement ID
Main	initialize LibGDX, and screen transitions	Initialize libgdx, switch between screens	**All screens**	-UR_PROJECT, -NFR_CODE_MODULARITY, -NFR_JAVA_VERSION
MenuScreen	Start menu for launching or exiting the game	Display UI, handle input, Start game, open settings	**Main, FirstScreen, SettingsScreen**	-FR_START_MENU, -UR_PAUSE, -UR_AUDIENCE
FirstScreen	Main gameplay loop	Manage timer. Renders maze and player, coordinates collision, handles events, detects win/loss	**Player, Collision, Events, Main, WinScreen, LoseScreen,**	-UR_MAZE, -UR_TIME, -UR_EVENTS, -FR_TIMER_RUN, -FR_TIMER_LOSE, -FR_EVENT_POS, -FR_EVENT_EG, -FR_EVENT_HID -FR_EVENT_CONNECT
WinScreen	Show win result and restart or exit, also passes player data to a persistent file for the leaderboard	Display outcome, return to menu, saves player data	**Main LeaderBoardScreen**	-FR_STATE_WIN_UI -UR_AUDIENCE -FR_LEADERBOARD_SCORE
LoseScreen	Show lose result and restart or exit	Display outcome, return to menu	**Main**	-FR_STATE_LOSE_UI, -UR_AUDIENCE
SettingsScreen	Ui for audio	Adjust sound	**Main**	-FR_SETTINGS_SOUND, -FR_SETTING_SUBTITLES, -UR_ACCESSIBILITY

Player	Represents player character	Handle movement input, expose bounds	**FirstScreen**, **Collision**, **Events**	-FR_MAP_ PROTAGONIST, -FR_MAP_ LIMITS, -UR_MAZE
Collision	Collision and exit queries	Detect wall collision, detect exit reached	**FirstScreen**, **Player**	-FR_MAP_ LIMITS, -FR_MAP_EXIT
Events	Event system	Manage events, Detect triggers, Apply effects, Update counters,	**FirstScreen**, **Player**	-UR_EVENTS, FR_EVENT_ CONNECT, -FR_EVENT_ COUNT_POS, -FR_EVENT_ COUNT_NEG, -FR_EVENT_ COUNT_HID,
LeaderBoardScreen	Handles the logic for the leaderboard data handling also includes rendering of top scores	Handles data, displays data	**Main** **WinScreen**	-FR_LEADERBOARD_SCORE -FR_LEADERBOARD_MENU
AchievementManager	Acts as a controller for all logic regarding achievements	Contains list of obtainable achievements, functions to correctly manipulate achievements	**FirstScreen** **Events**	-FR_ACHIEVEMENTS

References

Bramipis, k. (2025) Introduction to Software Architecture. Department of Computer Science, University of York.

Kolovos, D. (2025). Object-Oriented Modeling with UML. Department of Computer Science, University of York.

IEEE (2018) ISO/IEC/IEEE International Standard - System and software engineering life cycle processes - requirement engineering
https://standards.ieee.org/ieee/29148/6937/

LibGDX (2025) LibGDX Game Development Framework Documentation.
https://libgdx.com/wiki/

The Debug Thugs (2025) Escape the Maze - ENG1 Assessment 1 Project Website
https://thedebugthugs.github.io/