Architecture

Oliver Dixon
George Cranton
Praj Dethekar
Denys Sova
Shivan Ramharry
Albara Shoukri
Rafael Duarte

1.1 Initial Architecture

All the diagrams mentioned in this document can be found in the *'Architecture'* section of our website: https://eng1-group-23.github.io/A2-website

To ensure that our finished product met the requirements, we began the project by imagining, sketching and modelling a high-level architecture of the system's layout around the packages and utilities provided by LibGDX. We utilised LibGDX's features that enhance the game's interactive and visual elements like handling graphics, audio, and input processing. For instance, we integrated the 'InputProcessor' for sophisticated input management, 'SpriteBatch' for drawing sprites on the screen, 'Camera' to control what's visible on the screen and 'Vector2' for storing and manipulating points or directions in 2D space.

Using the 'Event Storming' approach, we initially identified key components for our architecture, including the 'Screen' interface from LibGDX for rendering game views, and the 'Entity' component for managing entities, starting with 'Player' and extendable to include non-player characters. The 'Screen' component retrieves necessary data and assets from other components to render the UI efficiently. Recognizing the utility of segregating game stages, we decided on implementing various screen classes. This approach not only enhances organisation by separating game phases but also optimises performance. By isolating resources and processing to the active screen, the game can efficiently manage memory and reduce loading times, ensuring smoother transitions and making the game more responsive.

From this simplified view of the system, we created the first architectural diagram as a set of Class-Responsibility-Collaborator (CRC) cards. We first identified all candidate objects and created a CRC card for each object. Each card contains the name of the class, the responsibility, or purpose of each candidate, and any other classes that are linked to it (collaborators). The collaboration between each object is organised using the delegated control style. We grouped similar candidates together and removed any duplicates. The CRC cards diagram can be found on our website (*Architecture, Fig.1*).

After reviewing the CRC cards, we identified some problems in our design, particularly with the MainGameScreen class handling game settings, character choices, energy levels, and audio. Modifying settings through the MainSettingsScreen class required an active MainGameScreen instance, which meant there was unnecessary resource usage and performance dips, as both UIs were rendered while in the settings screen.To solve this problem, GameData class was created to centrally manage game settings and data. This class is designed to be accessible by other classes. This ensured minimal performance impact and eliminated the need for MainGameScreen to be active during settings adjustments.

The sound and music classes (GameSound and GameMusic) were created to handle music and sound effects after only being called once - this reduces overhead and processing delays by efficiently handling sound resources. We introduced a ScreenHandler class to prevent performance issues by ensuring only one screen can be initiated at a time, reducing memory usage and improving game loading times. This class streamlines screen transitions, freeing up memory when switching screens, thereby enhancing user experience.

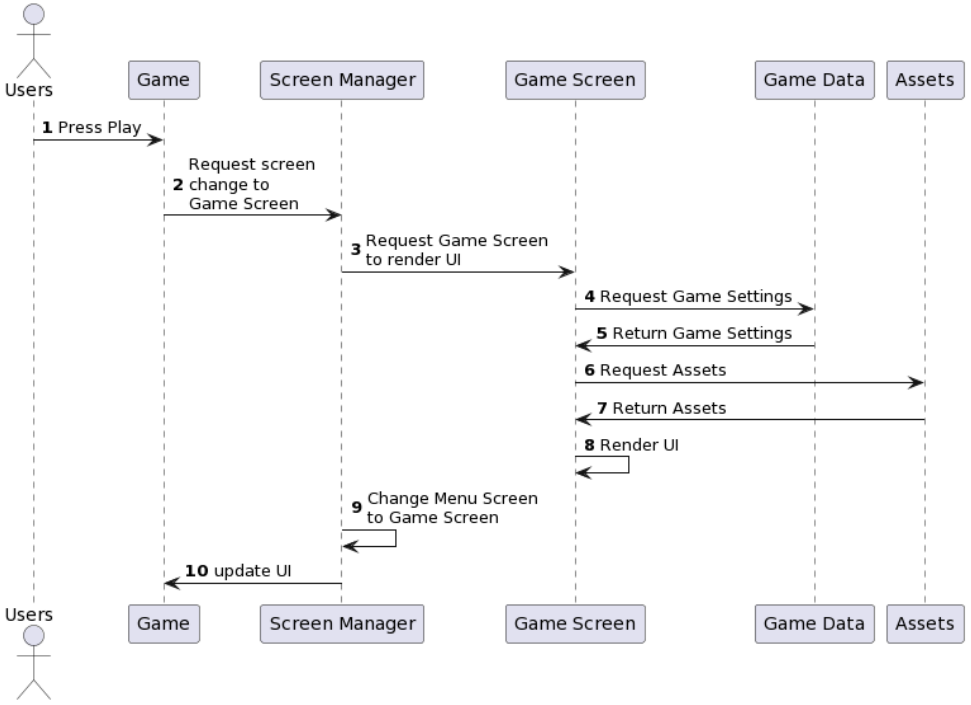
All the new classes were created one week after the CRC cards diagram was finished. We decided to reflect these changes in the initial class diagrams instead of further modifying the CRC cards diagram so we can focus on other parts of development. The *initial* class diagram was created using the CRC cards diagram as a reference. It can be found on our website (*Architecture, Fig.2*).

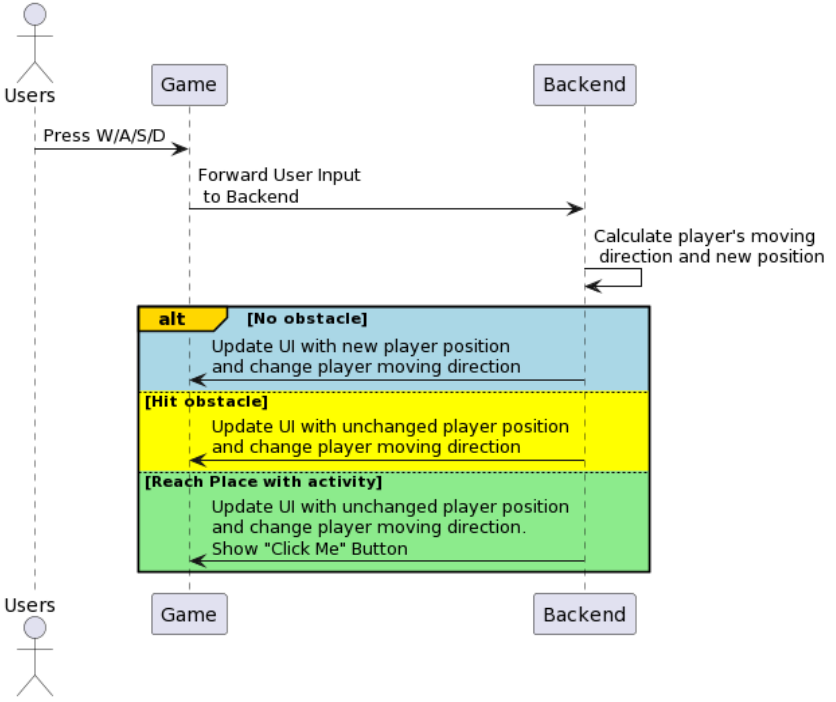
1.2 Justification of Tools

PlantUML is a simple and efficient way of creating diagrams that takes little time to learn, is easy to debug and integrates well into other software. We used it to create class and sequence diagrams, as well as plan our project with Gantt charts. This was easier than manually drawing diagrams because it is intuitive, and has autofill and suggestions that make the process quicker and easier than any other software. PlantUML was the right choice of tool for our team because it has good integration with Google Docs and IntelliJ IDEA, the IDE our team chose to use. The created diagrams can be easily exported as .png files from IntelliJ, and can be placed on our website.

We used the @startgantt/@endgantt to make Gantt charts, and @startuml/@enduml to create class diagrams with ‘packages’. For the sequence diagram, we used @startuml with ‘autonumber’ and ‘actor’ to define how the user interacts with the front end of the system, which then interacts with the backend.

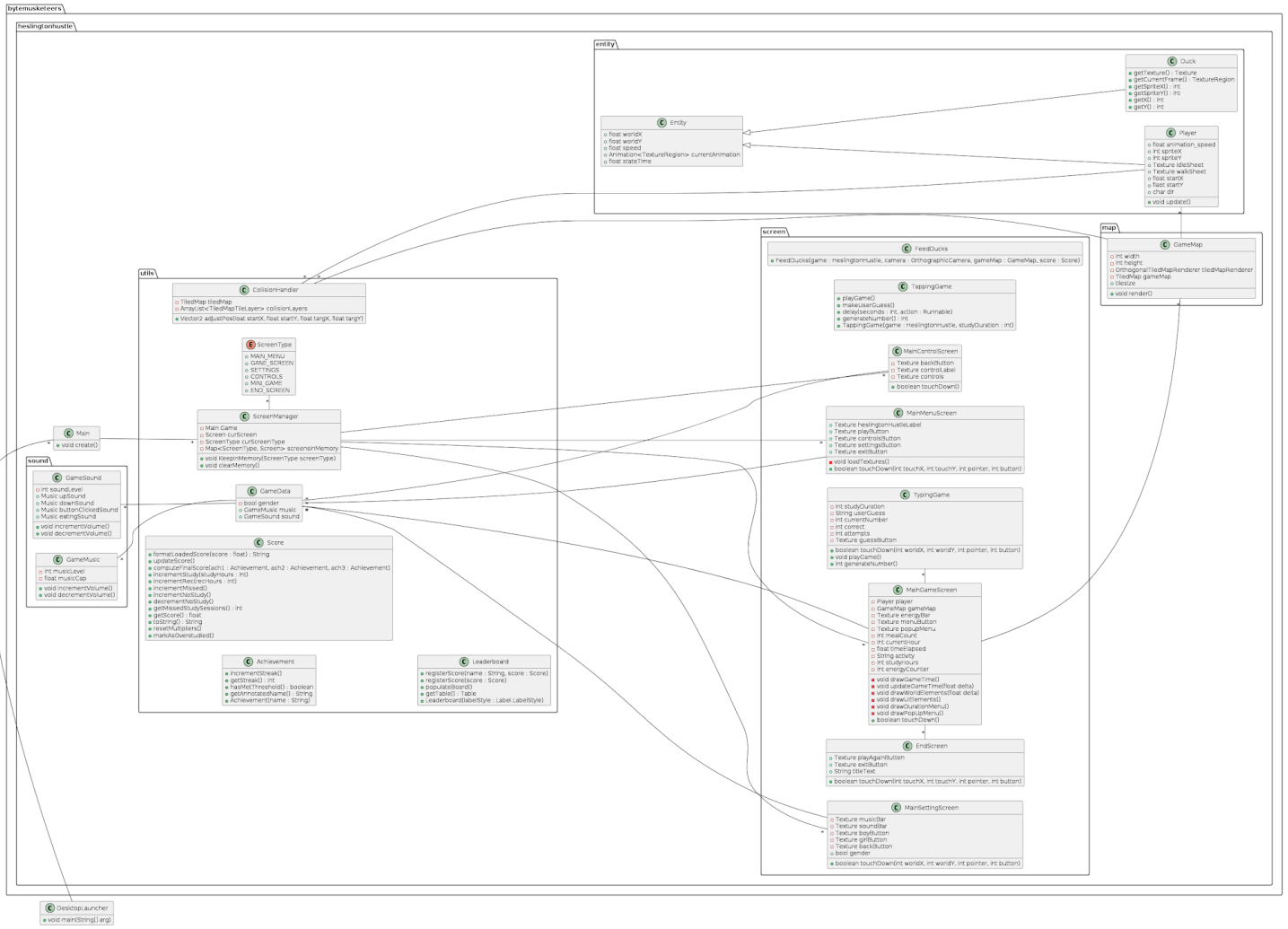
We created two sequence diagrams to show how the game reacts to users’ input in different situations. The sequence diagrams are shown below, and can be found on our website (*Architecture, Fig.4-5*). The second of these diagrams focuses specifically on what happens when the user moves the sprite.





The final class diagram is shown below, and can be found on our website (*Architecture, Fig.3)*. We decided to use multiple packages to group classes:

- The 'screen' package contains all screen related classes except the screen manager.
- All the classes related to sound effect and music are placed in the 'sound' package.
- Classes related to map rendering are placed in the 'map' package.
- Classes that will be used by other packages are placed in the 'utils' package



(This diagram was modified by Team 23 to represent all architectural changes required for the purposes of Assessment 2.)

1.3 Evolution of the Architecture

During implementation of the code, we adapted the architecture to suit the specific needs of our program's requirements. One problem we encountered was slow transitions to the Main Game Screen and inefficiencies in saving the 'gameScreenState'.

Initially, our ScreenHandler disposed of screens upon switching, leading to increased loading times and overhead due to frequent instantiation and data saving to the GameData class, especially for the resource-intensive MainGameScreen. To enhance performance, we refined ScreenManager to selectively keep certain screens, like MainGameScreen, in memory while disposing others as needed. This approach minimises loading delays, reduces data saving overhead, and maintains MainGameScreen's state for better performance and memory efficiency, ensuring it remains the only screen in memory when active and improving overall system responsiveness.

Introducing the CollisionHandler class segregates all collision related functionality into a single class, making the game engine more efficient. This reduces overhead and processor delays, thus ensuring smoother gameplay and optimising resource use whenever collision functionality is invoked.

Two classes that should have been implemented to segregate functionality are one for choosing gender and the popup menu, which is mentioned in the docs string in the code above the relevant method. This is to avoid unnecessary calls within the Player and MainGameScreen class, this separation could have optimised the code more,

We introduced a popup Menu activated by the CollisionHandler's 'isTouching' method, featuring buttons for different activities next to the player. Selecting 'study' connects MainGameScreen to TypingGameScreen. Enhancements like a fade effect and eating sounds improve user experience, signalling completed activities and smoothing transitions, such as starting a new day. Our minigame, TypingGame, challenges players to memorise and type increasingly long sequences of numbers, engaging them during study periods. Lastly, we added an EndScreen class that appears after 7 in-game days, offering options to replay or exit the game.

Given the updated customer requirements, the Leaderboard, Score, and Achievement classes were added to facilitate their respective functionalities. All classes expose minimalistic and decoupled interfaces to their packages, such that private methods (such as those found in EndScreen) can be easily modified to implement the new requirements. No architectural changes of existing classes were required to satisfy the new requirements.

The player experience was further enhanced (thus realising the SSON and `UR_STUDY_GAME`) with the additional definitions of the `FeedDucks` (supported by the `Duck` entity) and `TappingGame` minigames. These were not strictly required by the updated customer brief, but it was believed that minigames with a greater quantity and variety would strongly support existing user and non-functional requirements.

1.4 Relating the Architecture to the Requirements

FR_INTERACTION_TRIGGER:
Class: CollisionHandler
Role: The CollisionHandler class plays a crucial role in detecting player Interactions with tiles.

isTouching Method: This method is designed to detect when an object is touching tiles of a certain layer for instance a door, building or tree
Justification: The isTouching Method in CollisionHandler directly contributes to the FR_INTERACTION_TRIGGER by providing the necessary logic to identify when an interaction-triggering condition occurs in the game
Class: MainGameScreen
Role: Acts as the main interface for player interaction within the game.
drawPopUpMenu Method: Once the CollisionHandler detects a trigger con-dition, the MainGameScreen class responds by generating a popup menu
Justification: The drawPopupMenu Method fulfils the FR_INTERACTION_TRIGGER by providing an interactive response to the detected player action.

FR_COMPLETE_ACTION:
Class: MainGameScreen:
Role: Main interface for player interaction within the game.
Method: touchDown
Justification: This method directly fulfils FR_COMPLETE_ACTION by providing the functionality for the player to select and complete various activities from the interaction menu. Depending on the player's touch input, it triggers different actions such as studying, exercising, sleeping, or eating. Additionally, it handles interaction triggers by displaying a popup menu, allowing the player to choose actions like studying, eating, or exercising upon touching specific doors.

FR_START_GAME
Class: MainMenuScreen
Role: Representing the main menu screen and handling interactions within it.
Method: touchDown()
Justification: This method processes touch events on the main menu screen. In the context of FR_START_GAME, it detects when the player touches the "START GAME" button and initiates the game accordingly.

FR_FULLSCREEN
Class: MainGameScreen
Role: Represents the main gameplay interface where all game elements are rendered, including the player character, map, UI elements, and pop-up menus.
Method: resize(int width, int height)
Justification: The resize method ensures that all elements on the screen, including the player character, map, UI elements, and pop-up menus, are properly adjusted and scaled to fit the new window size, thereby fulfilling the requirement for a full-screen display on any window size.
Class: ScreenManager
Role: Manages the game screens, including creation, switching, and memory management of screens.
Justification: ScreenManager class plays a vital role in ensuring that all screens, including MainGameScreen, are resized appropriately to maintain full-screen display. Its resize() method iterates through all screens, including the current screen (MainGameScreen), and adjusts their dimensions to fit the new window size, thereby fulfilling the requirement for a full-screen display on any window size.

FR_LEADERBOARD
Class: Leaderboard
Role: Represents and manages an internal (persistent) model of each players' score, the 10 highest-scoring of which are selected for inclusion on the LibGDX table visible on the EndScreen.
Method: all exposed methods

Justification: registerScore allows the game-ending handler to update the score file appropriately, and getTable allows the game-ending UI (such as EndScreen) to retrieve a fully formed Table visualising the top-10-scoring players over all game invocations.

FR_SCORE
Class: Score
Role: Maintains and provides an interface to update and retrieve a numerical score of the user, subject to certain constraints regarding the attainment of achievements and (as described in the *Requirements* deliverable).
Method: all exposed methods
Justification: Each of the exposed methods exists to provide an interface to modify the parameters used to compute the final score, such as incrementRec (increment the number of recreational activities performed), and markAsOverstudied (weigh the disadvantages of overstudying with the other activities undertaken). Additionally, the getScore and toString override methods provide a consistent and predictable interface through which the MainGameScreen and EndScreen can retrieve and format the computed final score.

FR_ACHIEVEMENTS
Class: Achievement
Role: Provides an interface to registering and reading patterns of behaviour that are deemed, by the updated customer requirements, to merit special recognition in the form of an “achievement” or “streak”. These achievements are also staged, such that each achievement type can be enhanced, should the player continue the achievement-meriting behaviour.
Method: getStreak, hasMetThreshold, and getAnnotatedName
Justification: These methods are used by the EndScreen to display the achievements in a human-readable manner at the end of the seven-day cycle. In particular, given the discretely staged nature of each achievement, a human-readable prefix (“Beginner”, “Intermediate”, or “Master”) can be prepended to each achievement name to yield an annotated label to be used by the EndScreen.

NFR_SCALABILITY
The Architecture of the ScreenManager and GameData class supports further development by another team:
The ScreenManager class streamlines game screen management, offering an intuitive interface for screen creation, switching, and memory handling. Its use of a Map for storing screens enhances efficiency and scalability. The clearMemory() method aids in optimal memory use by removing unneeded screens, while setScreen() and createScreen() methods simplify adding and creating new screens.
The GameData class centralises game settings, such as gender selection and audio levels, facilitating easy access and modification. Its methods for setting preferences ensure a straightforward interaction with game data, promoting modular development and future extensibility.

NFR_EFFICIENCY
The MainGameScreen class architecture reduces CPU and resource usage. Its Efficient Rendering method updates game elements as needed, clears the screen, and draws world and UI elements separately to cut down on needless rendering and enhance performance. The dispose() method disposes of unused resources, preventing memory leaks and optimising resource management.

Requirement ID	Related Architecture
UR_MOVEMENT	The player is able to move the avatar around the 2D map with the use of the Arrow keys. The movement of the Avatar is implemented in the Player class.The Player class represents the character in the game, handling movement, collision, and animations. The player class uses the setPos method in order to set the player's position to the specified coordinates adjusting the worldX and worldY variables
UR_CONTROLS	The game's controls are intuitive and are clearly presented to the player on the controls screen which is accessed via the main menu. Visually explaining the controls of the game is implemented through the MainControlScreen which is associated with the ScreenManager in order for it to be displayed. The ScreenManager class manages the game screens, including creation, switching, and memory management of screens.
UR_ACCESSIBILITY	The game is for new players, so it is easy to understand and play with no prior experience. This is reflected in the games easy to understand User Interface and the ability to access the main menu whenever.The MainMenuScreen class represents the main menu screen for the game. It handles the display and interaction with the main menu, including navigating to different parts of the game such as starting the gameplay, viewing controls, adjusting settings, or exiting the game.
UR_TIME_SCALE	In the game, one real-time minute equals one in-game day. The MainGameScreen manages time, updating the timeElapsed and currentHour variables to track in-game time, ensuring days align with real-time minutes. Three methods handle time display: updateGameTime() cycles active hours (8 AM to 12 AM), resetDay, and drawGameTime(). At 12 AM, the game resets to 8 AM for a new day.
UR_RECREATION	There is a building that the avatar can interact with to recreate. The recreation activity that has been used in the game is exercise in the gym. When exercise is selected the user is offered a choice of hours they would like to spend exercising from 1 to 4. When a time is selected a time skip occurs and the recreation count is incremented. For the MainGameScreen the code for recreation is within the touchdown() method.
UR_STUDYING	There is a building that the avatar can interact with to study. There are two buildings where the studying action can be started from. When selected the user will be prompted to select the amount of hours and then afterwards the studying minigame will start. Within the MainGameScreen class the method touchdown() is in charge of commanding the study activity. The method uses ScreenManager in order to switch to the minigame.
UR_SLEEPING	There is a building that the avatar can interact with to sleep. Sleeping can only be started after 8pm and is automatically completed when every day is over. After sleeping has finished, your character's position is placed outside the sleeping building. In the class MainGameScreen the method touchDown() is where the sleeping function is selected. A fade out is activated and the energy bar is reset.
UR_EATING	There is a building that the avatar can interact with to eat. In the class MainGameScreen the method touchDown(), along with the use of a switch statement is used to determine when to proceed with the sleeping function. When eating is selected the game data class is needed to be called in order to activate the associated sound. When eating is commenced the energyCounter is increased by 3 and the mealCount is incremented.

UR_LEADERBOARD	The `Leaderboard` class persists and provides an interface to manage all players' scores over each invocation of the game, and provides exposed helper methods to external classes (e.g. the `EndScreen`) to visualise the stored player-score mappings consisting of the highest scores.
UR_ACHIEVEMENTS	The `Achievement` class was added to satisfy the UR_ACHIEVEMENTS requirement, and dually the FR_ACHIEVEMENTS functional requirement. Throughout the gameplay, the behaviour of the character is monitored to determine any merit-worthy patterns of behaviour, such as studying, sleeping, and recreating consistently.
UR_PAUSE	No publicly visible architectural design choices were necessary to satisfy the UR_PAUSE requirement, as the existing main menu (implemented with `MenuMenuScreen`) could be used as a pause screen. Hence only implementation changes were necessary to toggle the display of the `MainMenuScreen` on a wider range of inputs (i.e. the Escape key being pressed).