Architecture

James Smith
Amanda Ling
Fran Medland
Hannah Vas
James Kellett
Malik Tremain
Mischa Zaynchkovsky

1.1 Initial Architecture

All the diagrams mentioned in this document can be found in the *'Architecture'* section of our website: https://publicmutiny.github.io/f1sh-webs1te/.

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.

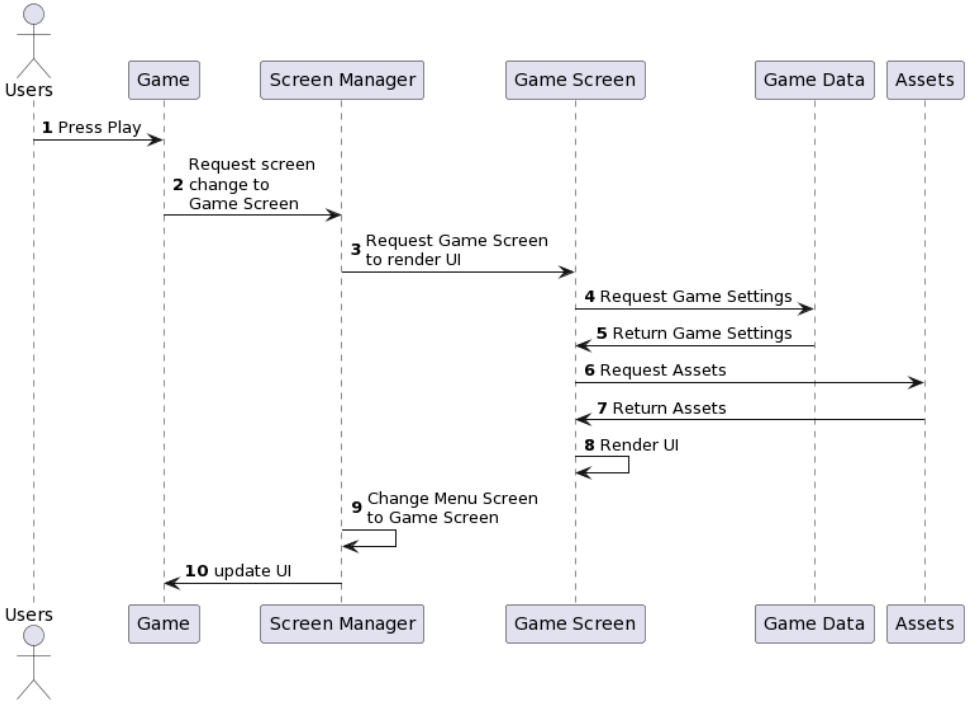
All the new classes were created one week after the CRC cards diagram was finished. We decided to reflect these changes in the initial class diagrams instead of further modifying the CRC cards diagram so we can focus on other parts of development. The *initial* class diagram was created using the CRC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

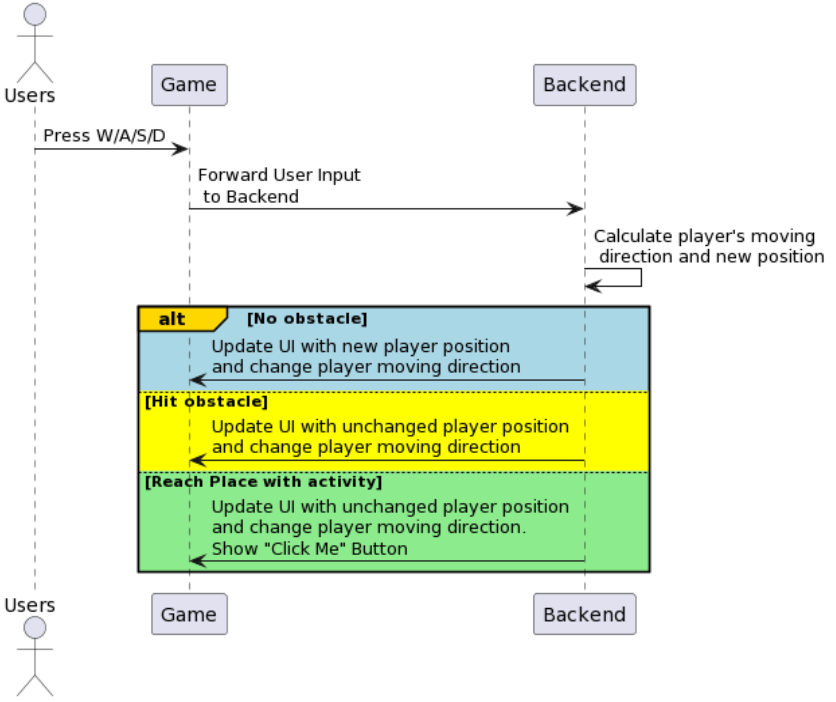
1.2 Justification of Tools

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 to edit 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.

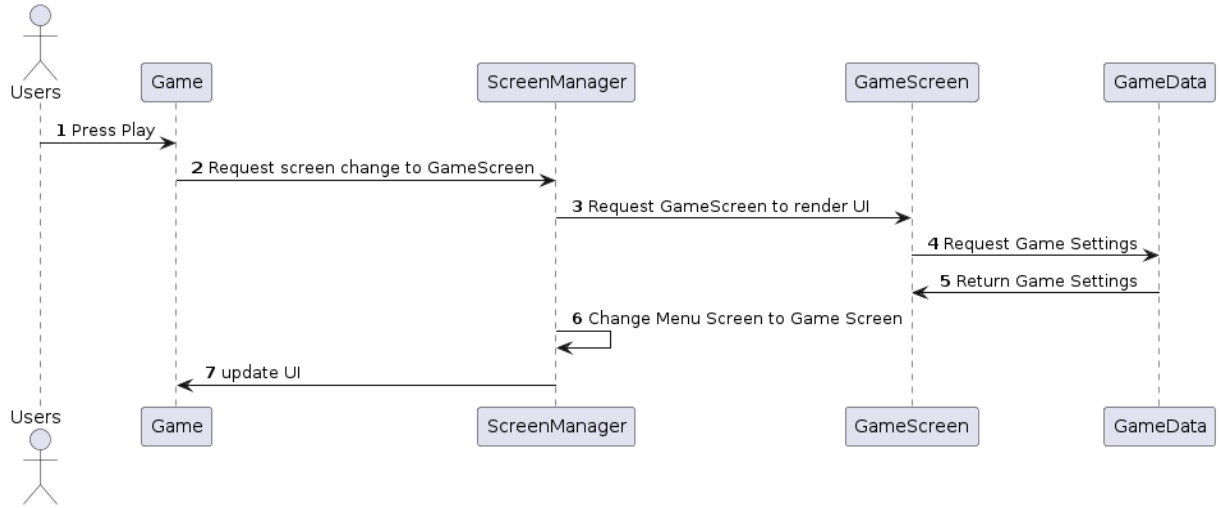
We used the @startgantt/@endgantt to make Gantt charts, and @startuml/@enduml to create class diagrams with 'packages'. For the sequence diagram, we used @startuml with 'autonumber' and 'actor' to define how the user interacts with the front end of the system, which then interacts with the backend.

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.



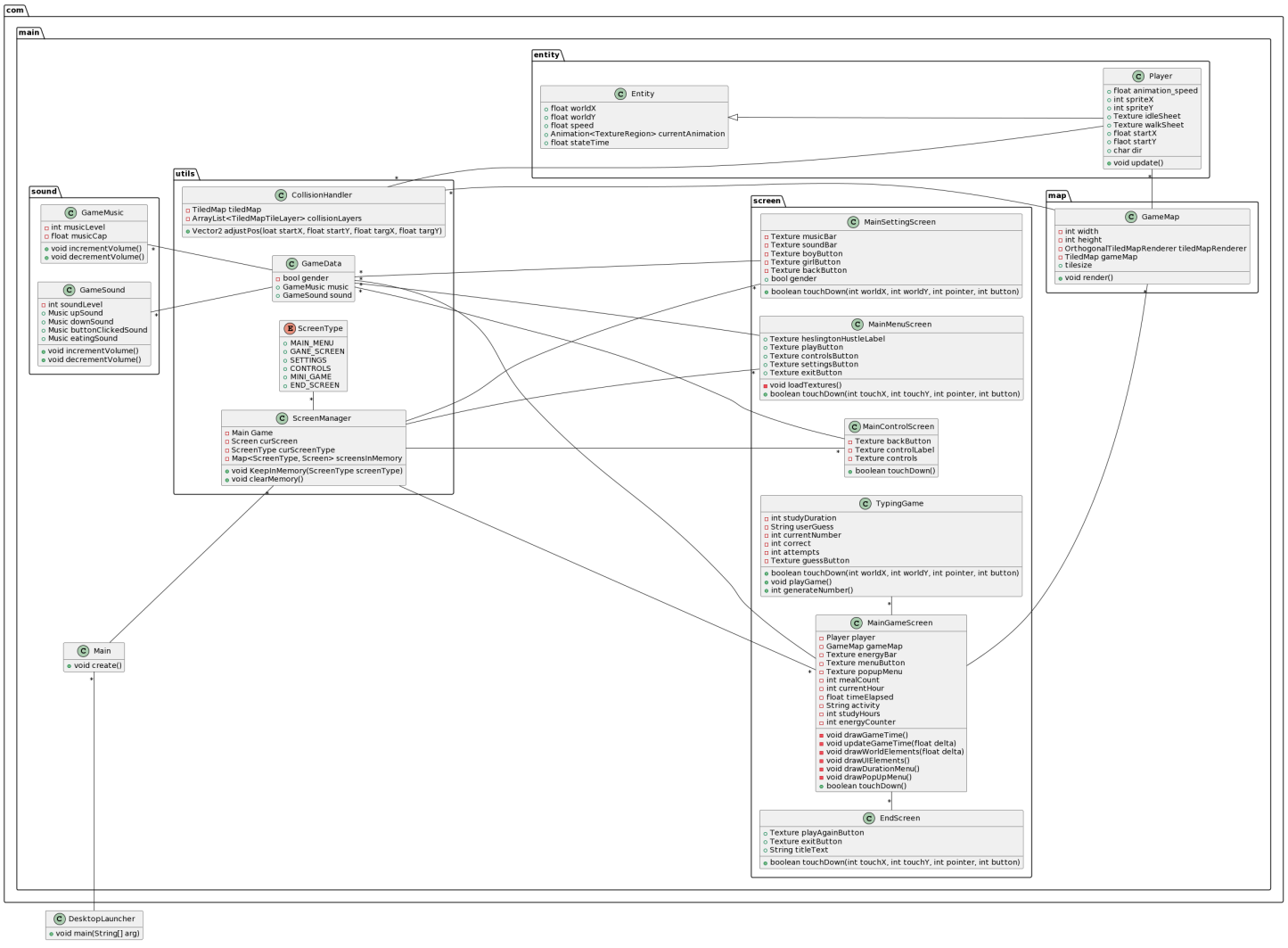


Below is an updated sequence diagram for (*Architecture, Fig.4*) which reflects how the current game in assessment 2 reacts to the users' input when they start the game (press play).



The final class diagram for assessment 1 is shown below, and can be found on our website (*Architecture, Fig.3)*. We decided to use multiple packages to group classes:

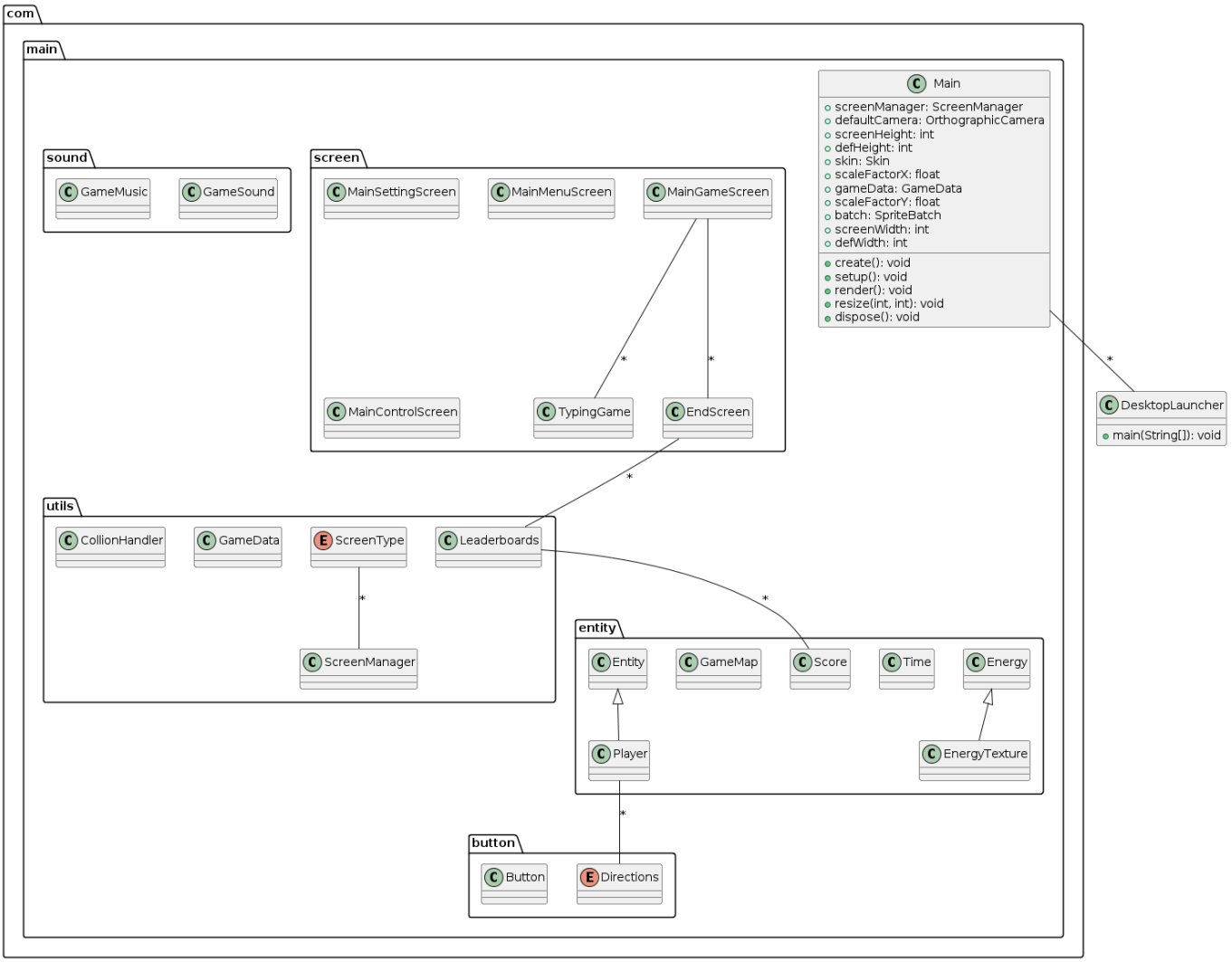
- The 'screen' package contains all screen related classes except the screen manager.
- All the classes related to sound effect and music are placed in the 'sound' package.
- Classes related to map rendering are placed in the 'map' package.
- Classes that will be used by other packages are placed in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

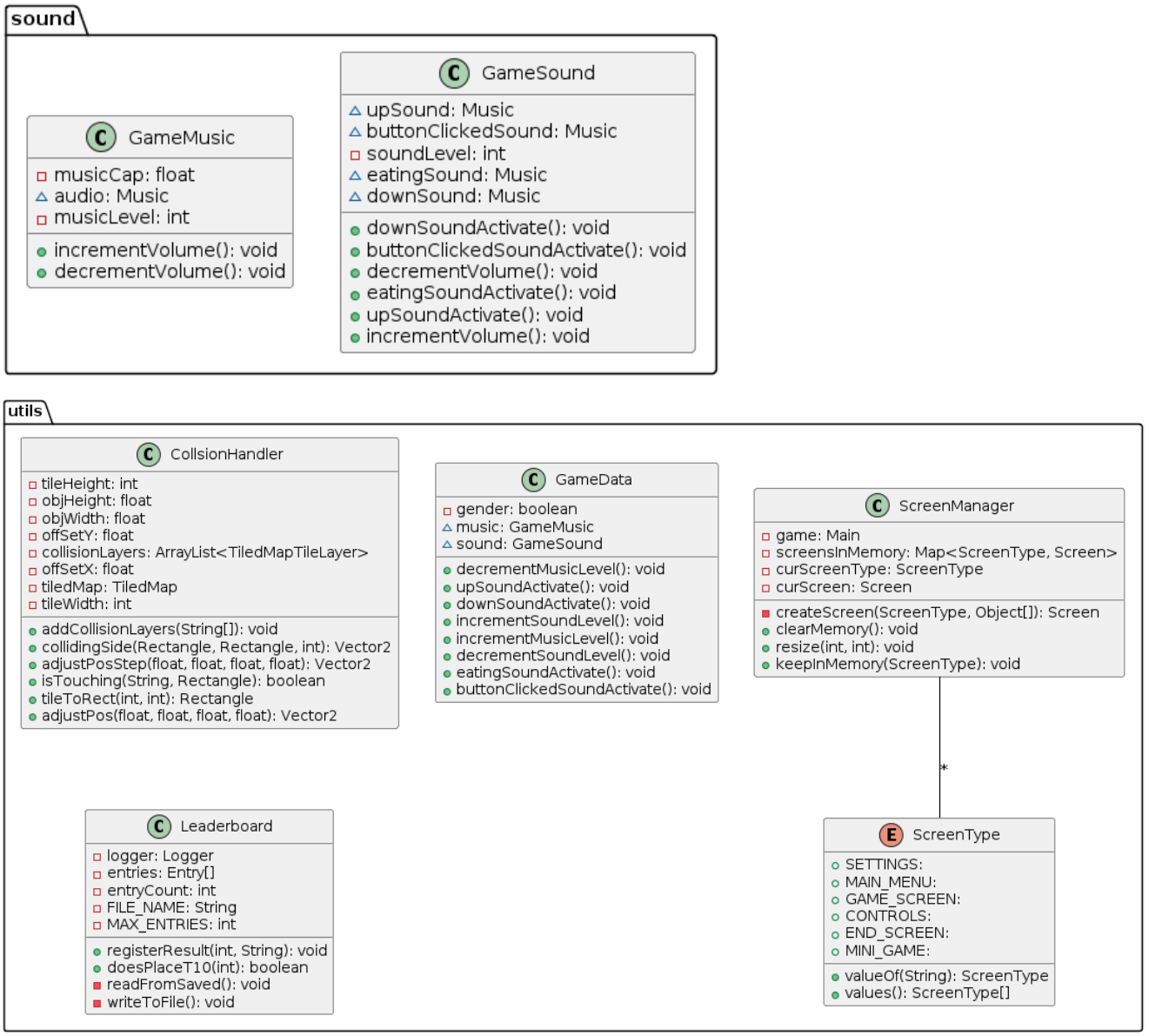


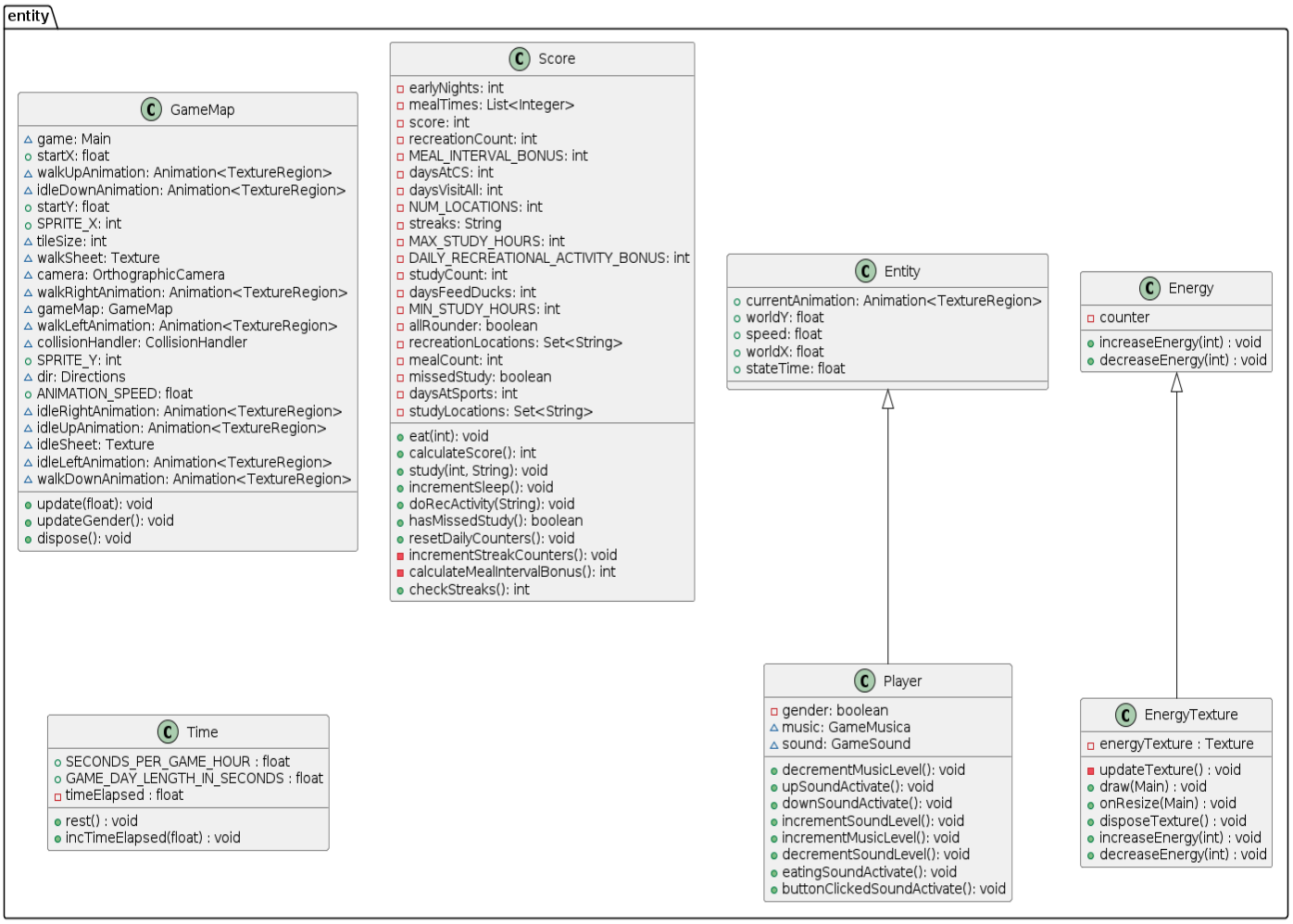
The final class diagrams for assessment 2 are shown below. Due to too many attributes and methods in certain classes a “ScriptError: Limit Exceeded” occurred which meant that a single, readable PlantUML diagram could not be generated from Google Docs. To overcome this, only necessary attributes and methods were kept in classes (getters/setters removed). Below is a skeleton class diagram and each individual package diagram to allow full readablity. In the skeleton class diagram the previous links remain the same with the addition of new links for the new implemented classes. The links between classes from different packages aren’t displayed due to readability issues as the generated lines crossover into a single line. The skeleton class diagram can be seen on our website.

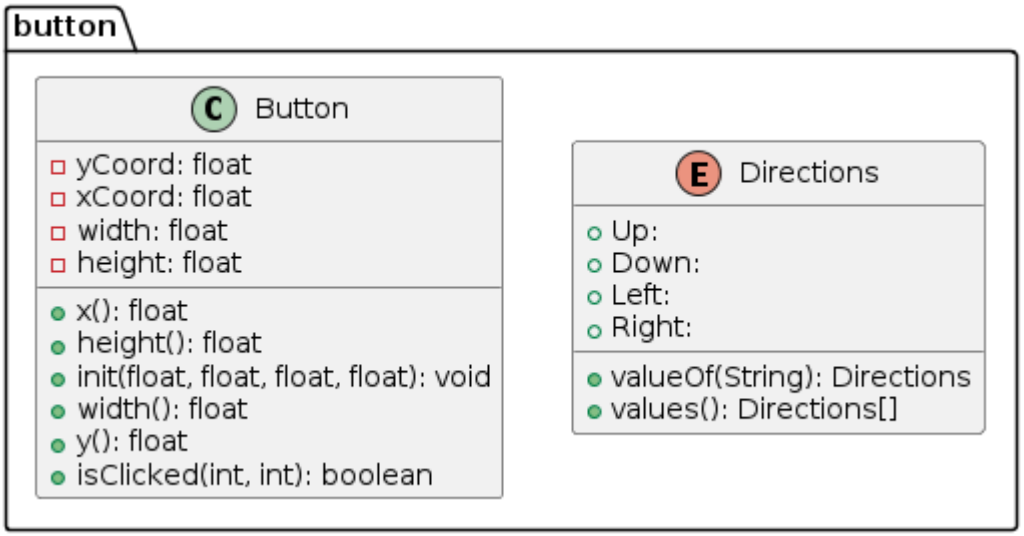
We decided to use a similar package design as previously done in assessment 1 but altered a few things. For instance:

- In the screen package, each class has its own package for readability (explained earlier).
- The map package was removed as only one class, GameMap, was in it. The GameMap class was moved to the entity package instead.
- New Leaderboards class was placed into the utils package as it manages the backend of file handling for scoring.
- New Score, Time, Energy and EnergyTexture were placed into the entity class as their class functions used to be managed in a single class but are now in individual classes for efficiency.
- New Button class has its own button package.



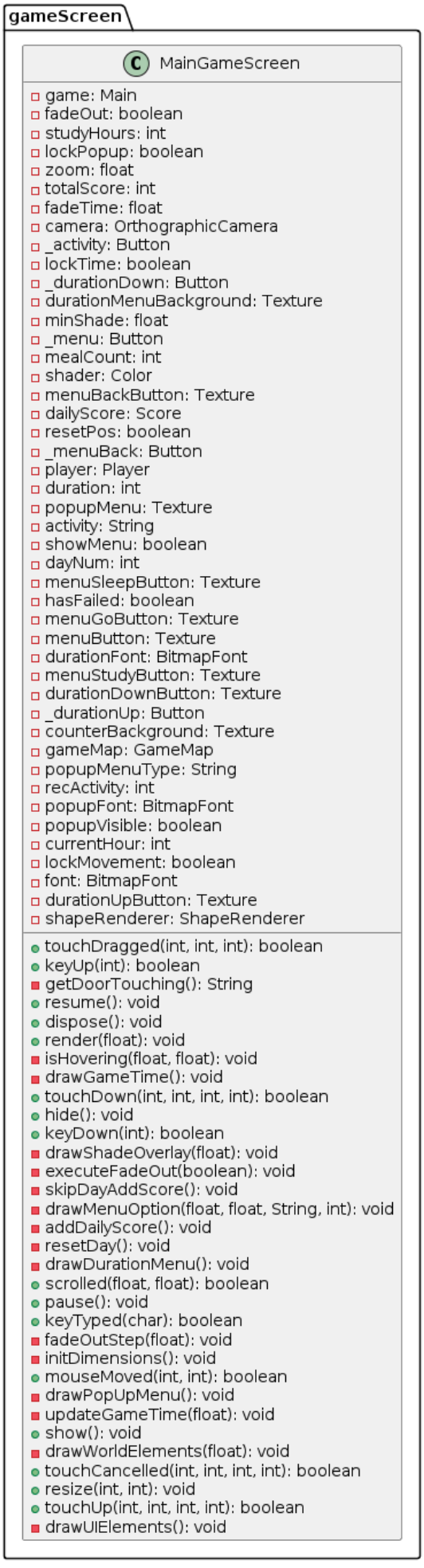

sound
GameMusic
musicCap: float
audio: Music
musicLevel: int
incrementVolume(): void
decrementVolume(): void
GameSound
upSound: Music
buttonClickedSound: Music
soundLevel: int
eatingSound: Music
downSound: Music
downSoundActivate(): void
buttonClickedSoundActivate(): void
decrementVolume(): void
eatingSoundActivate(): void
upSoundActivate(): void
incrementVolume(): void
utils
CollsionHandler
tileHeight: int
objHeight: float
objWidth: float
offSetY: float
collisionLayers: ArrayList<TiledMapTileLayer>
offSetX: float
tiledMap: TiledMap
tileWidth: int
addCollisionLayers(String[]): void
collidingSide(Rectangle, Rectangle, int): Vector2
adjustPosStep(float, float, float, float): Vector2
isTouching(String, Rectangle): boolean
tileToRect(int, int): Rectangle
adjustPos(float, float, float, float): Vector2
GameData
gender: boolean
music: GameMusic
sound: GameSound
decrementMusicLevel(): void
upSoundActivate(): void
downSoundActivate(): void
incrementSoundLevel(): void
incrementMusicLevel(): void
decrementSoundLevel(): void
eatingSoundActivate(): void
buttonClickedSoundActivate(): void
ScreenManager
game: Main
screensInMemory: Map<ScreenType, Screen>
curScreenType: ScreenType
curScreen: Screen
createScreen(ScreenType, Object[]): Screen
clearMemory(): void
resize(int, int): void
keepInMemory(ScreenType): void
*
Leaderboard
logger: Logger
entries: Entry[]
entryCount: int
FILE_NAME: String
MAX_ENTRIES: int
registerResult(int, String): void
doesPlaceT10(int): boolean
readFromSaved(): void
writeToFile(): void
ScreenType
SETTINGS:
MAIN_MENU:
GAME_SCREEN:
CONTROLS:
END_SCREEN:
MINI_GAME:
valueOf(String): ScreenType
values(): ScreenType[]


entity
GameMap
game: Main
startX: float
walkUpAnimation: Animation<TextureRegion>
idleDownAnimation: Animation<TextureRegion>
startY: float
SPRITE_X: int
tileSize: int
walkSheet: Texture
camera: OrthographicCamera
walkRightAnimation: Animation<TextureRegion>
gameMap: GameMap
walkLeftAnimation: Animation<TextureRegion>
collisionHandler: CollisionHandler
SPRITE_Y: int
dir: Directions
ANIMATION_SPEED: float
idleRightAnimation: Animation<TextureRegion>
idleUpAnimation: Animation<TextureRegion>
idleSheet: Texture
idleLeftAnimation: Animation<TextureRegion>
walkDownAnimation: Animation<TextureRegion>
update(float): void
updateGender(): void
dispose(): void
Score
earlyNights: int
mealTimes: List<Integer>
score: int
recreationCount: int
MEAL_INTERVAL_BONUS: int
daysAtCS: int
daysVisitAll: int
NUM_LOCATIONS: int
streaks: String
MAX_STUDY_HOURS: int
DAILY_RECREATIONAL_ACTIVITY_BONUS: int
studyCount: int
daysFeedDucks: int
MIN_STUDY_HOURS: int
allRounder: boolean
recreationLocations: Set<String>
mealCount: int
missedStudy: boolean
daysAtSports: int
studyLocations: Set<String>
eat(int): void
calculateScore(): int
study(int, String): void
incrementSleep(): void
doRecActivity(String): void
hasMissedStudy(): boolean
resetDailyCounters(): void
incrementStreakCounters(): void
calculateMealIntervalBonus(): int
checkStreaks(): int
Entity
currentAnimation: Animation<TextureRegion>
worldY: float
speed: float
worldX: float
stateTime: float
Energy
counter
increaseEnergy(int) : void
decreaseEnergy(int) : void
Time
SECONDS_PER_GAME_HOUR : float
GAME_DAY_LENGTH_IN_SECONDS : float
timeElapsed : float
rest() : void
incTimeElapsed(float) : void
Player
gender: boolean
music: GameMusica
sound: GameSound
decrementMusicLevel(): void
upSoundActivate(): void
downSoundActivate(): void
incrementSoundLevel(): void
incrementMusicLevel(): void
decrementSoundLevel(): void
eatingSoundActivate(): void
buttonClickedSoundActivate(): void
EnergyTexture
energyTexture : Texture
updateTexture() : void
draw(Main) : void
onResize(Main) : void
disposeTexture() : void
increaseEnergy(int) : void
decreaseEnergy(int) : void


button
Button
yCoord: float
xCoord: float
width: float
height: float
x(): float
height(): float
init(float, float, float, float): void
width(): float
y(): float
isClicked(int, int): boolean
Directions
Up:
Down:
Left:
Right:
valueOf(String): Directions
values(): Directions[]


settingsScreen
MainSettingScreen
musicLabel: Texture
boyButton: Texture
soundLabel: Texture
game: Main
soundBar: Texture
settingsLabel: Texture
musicBar: Texture
gender: boolean
musicUpButton: Texture
girlButton: Texture
musicDownButton: Texture
backButton: Texture
soundDownButton: Texture
soundUpButton: Texture
touchDown(int, int, int, int): boolean
keyTyped(char): boolean
hide(): void
keyDown(int): boolean
mouseMoved(int, int): boolean
touchUp(int, int, int, int): boolean
show(): void
resize(int, int): void
dispose(): void
scrolled(float, float): boolean
touchCancelled(int, int, int, int): boolean
resume(): void
pause(): void
touchDragged(int, int, int): boolean
keyUp(int): boolean
render(float): void
menuScreen
MainMenuScreen
settingsButton: Texture
exitButton: Texture
controlsButton: Texture
game: Main
playButton: Texture
exitFlag: boolean
heslingtonHustleLabel: Texture
keyTyped(char): boolean
touchCancelled(int, int, int, int): boolean
keyDown(int): boolean
scrolled(float, float): boolean
render(float): void
mouseMoved(int, int): boolean
touchUp(int, int, int, int): boolean
show(): void
resize(int, int): void
touchDragged(int, int, int): boolean
dispose(): void
pause(): void
touchDown(int, int, int, int): boolean
keyUp(int): boolean
loadTextures(): void
hide(): void
resume(): void
controlScreen
MainControlScreen
objective: String
controlLabel: Texture
backButton: Texture
controls: Texture
game: Main
font: BitmapFont
touchCancelled(int, int, int, int): boolean
resize(int, int): void
resume(): void
hide(): void
dispose(): void
keyUp(int): boolean
touchDown(int, int, int, int): boolean
touchDragged(int, int, int): boolean
mouseMoved(int, int): boolean
render(float): void
show(): void
keyTyped(char): boolean
keyDown(int): boolean
touchUp(int, int, int, int): boolean
scrolled(float, float): boolean
pause(): void


typingGame
TypingGame
correct: int
displayCorrect: Boolean
attempts: int
title: Texture
acceptInput: Boolean
studyDuration: int
displayText: BitmapFont
guessButton: Texture
userGuess: String
currentNumber: int
game: Main
displayWrong: Boolean
gameObjective: String
touchUp(int, int, int, int): boolean
mouseMoved(int, int): boolean
keyTyped(char): boolean
delay(int, Runnable): void
hide(): void
touchDown(int, int, int, int): boolean
makeUserGuess(): void
generateNumber(): int
touchCancelled(int, int, int, int): boolean
resize(int, int): void
render(float): void
scrolled(float, float): boolean
playGame(): void
show(): void
dispose(): void
touchDragged(int, int, int): boolean
resume(): void
keyUp(int): boolean
pause(): void
keyDown(int): boolean
endScreen
EndScreen
exitFlag: boolean
username: String
playAgainButton: Texture
titleText: String
leaderboards: Leaderboards
blinkTimer: float
streaks: String
userScore: int
layout: GlyphLayout
playAgain: Button
font: BitmapFont
game: Main
usernameEntry: boolean
showUnderscore: boolean
touchCancelled(int, int, int, int): boolean
pause(): void
hide(): void
displayEntryBox(float, float): void
touchUp(int, int, int, int): boolean
resize(int, int): void
scrolled(float, float): boolean
initDimensions(): void
touchDown(int, int, int, int): boolean
render(float): void
displayLeaderboard(float): void
show(): void
dispose(): void
touchDragged(int, int, int): boolean
resume(): void
keyDown(int): boolean
keyUp(int): boolean
mouseMoved(int, int): boolean
keyTyped(char): boolean


gameScreen
MainGameScreen
game: Main
fadeOut: boolean
studyHours: int
lockPopup: boolean
zoom: float
totalScore: int
fadeTime: float
camera: OrthographicCamera
_activity: Button
lockTime: boolean
_durationDown: Button
durationMenuBackground: Texture
minShade: float
_menu: Button
mealCount: int
shader: Color
menuBackButton: Texture
dailyScore: Score
resetPos: boolean
_menuBack: Button
player: Player
duration: int
popupMenu: Texture
activity: String
showMenu: boolean
dayNum: int
menuSleepButton: Texture
hasFailed: boolean
menuGoButton: Texture
menuButton: Texture
durationFont: BitmapFont
menuStudyButton: Texture
durationDownButton: Texture
_durationUp: Button
counterBackground: Texture
gameMap: GameMap
popupMenuType: String
recActivity: int
popupFont: BitmapFont
popupVisible: boolean
currentHour: int
lockMovement: boolean
font: BitmapFont
durationUpButton: Texture
shapeRenderer: ShapeRenderer
touchDragged(int, int, int): boolean
keyUp(int): boolean
getDoorTouching(): String
resume(): void
dispose(): void
render(float): void
isHovering(float, float): void
drawGameTime(): void
touchDown(int, int, int, int): boolean
hide(): void
keyDown(int): boolean
drawShadeOverlay(float): void
executeFadeOut(boolean): void
skipDayAddScore(): void
drawMenuOption(float, float, String, int): void
addDailyScore(): void
resetDay(): void
drawDurationMenu(): void
scrolled(float, float): boolean
pause(): void
keyTyped(char): boolean
fadeOutStep(float): void
initDimensions(): void
mouseMoved(int, int): boolean
drawPopUpMenu(): void
updateGameTime(float): void
show(): void
drawWorldElements(float): void
touchCancelled(int, int, int, int): boolean
resize(int, int): void
touchUp(int, int, int, int): boolean
drawUIElements(): void

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, displaying a leaderboard and offering options to replay or exit the game.

With the addition of a new feature, leaderboard, the architecture was updated to include the class Leaderboards which handles the file writing for score management along with retrieving the score data for displaying. Moreover, the class Score was introduced to make calculating the score more efficient as there previously was no separate class for handling scoring as well as for another new feature, daily activity streaks.

Similarly to the Score class, classes Time, Energy and EnergyTexture were implemented to handle their class functions separately rather than in a single class; Time class manages the time conversions from real time to in-game time; Energy class manages the increase and decrease of the player's energy according to the type of activities done and EnergyTexture manages the UI rendering of the player's energy bar. Lastly, the class Button was also implemented to increase extensibility and reduce code duplication due to the oversighted use of the method 'touchdown' in MainGameScreen causing issues in implementation.

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/ FR_START_SCREEN
Class: MainMenuScreen
Role: Representing the main menu screen and handling interactions within it.
Method: touchDown()
Objects: playButton : Texture, controlButton : Texture, settingsButton : Texture, exitButton : Texture
Justification: This method processes touch events on the main menu screen with the objects above displayed as options. 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_SETTINGS_SCREEN
Class: MainSettingsScreen

Role: Represents the settings screen handling interactions to change sound level and gender of the avatar.
Method: touchDown()
Objects: musicUpButton : Texture, musicDownButton : Texture, musicBar : Texture, girlButton : Texture, boyButton : Texture
Justification: This method processes touch events on the settings screen with the objects above displayed. When the up/down button is touched the volume for music/sound increases/decreases with visible changes being shown through the musicBar level. When the 'Girl' button is touched, the avatar will change to female gender and similarly for 'Boy the avatar will change to the male gender once the game is started.

FR_CONTROLS_SCREEN
Class: MainControlScreen
Role: Displayed on the screen are instructions of how to play the game, keyboard inputs, and the objective of the game.
Method: render(float delta)
Objects/Attribute: instructionX : float, instructionY : float, controlLabel : Texture, controls : Texture, objective : string
Justification: The method render(float) is responsible for drawing the screen's content, the game objective, images of the buttons for control along with their description.

FR_MOVEMENT_KEYS/ FR_MOVEMENT_ARROWS
Class: Directions (enum)
Role: Player can move using arrow and WASD keys.
Method: Update()
Objects/Attributes: ANIMATION_SPEED: float, SPRITE_X:int, SPRITE_Y:int, game:Main, gameMap:GameMap, camera:OrthographicCamera, collisionHandler:CollisionHandler, tilesize:int, StartX:float, StartY:float, idleSheet, walksheet: Texture
Justification: The Method updates the player's position, animations, and handles collision.

FR_INTERACTION_MENU
Class: MainGameScreen
Role: A pop-up appears to indicate an activity the player can complete.
Method: drawPopUpMenu()
Objects/Attributes: popupMenu : Texture, popupVisible : boolean
Justification: Displays the popupMenu when popupVisible is set to true. The pop-up icon appears once the avatar is within the building door vicinity. Depending on the type of building, different activity icon(s) will appear.

FR_DISPLAY_ENERGY
Class: MainGameScreen
Role: Energy bar of the player is displayed on the screen.
Method: drawUIElements()
Objects/Attribute: energyBar : Texture, energyBarX, energyBarY, energyBarWidth, energyBarHeight
Justification: Renders the energy bar on the main game screen.
Class: EnergyTexture
Method: draw()
Objects/Attributes: energyBarY, energyBarX, energyBarWidth, energyBarHeight : float, energyTexture : Texture
Justification: updates the energy bar so that the user can see the current amount of energy remaining

FR_DISPLAY_TIME
Class: MainGameScreen
Role: Displayed on the screen will be the date and time.
Method: drawGameTime()

Attributes: dayNum : int, currentHour : int
Justification: Draws the current game time in the format of day X, HH:MM where the game clock is updated through the method updateGameTime()

FR_TIME
Class: Time
Role: Manages how time runs in the game, i.e how fast the hours days go.
Method: incTimeElapsed(float)
Attributes: GAME_DAY_LENGTH_IN_HOURS : float, SECONDS_PER_GAME_HOUR : float
Justification: Increments the elapsed time by a specified number of seconds.

FR_SCORING
Class: Score
Role: Calculates the score of the player's activity choices, including tracking streaks for certain activities done each consecutive day.
Method: calculateScore()
Attributes: score, studyCount, missedStudy, MIN_STUDY_HOURS, MAX_STUDY_HOURS, studyLocations, recreationLocations, recreationCount
Justification: Calculates and updates the player's score based on their actions and sets the score variable.

FR_END
Class: EndScreen
Role: Represents the final interface once the user has either completed the game successfully or has failed it.
Method: render(float)
Object/Attributes: streaks : String, leaderboards : Leaderboards
Justification: Once 7 days have passed, the end screen is displayed. The end screen shows a leaderboard of the top 10 scoring players; the user can enter their name and see their ranking if they score high enough. The daily streak of the player is also displayed in the top left corner of the screen.

FR_LOSE_CONDITION
Class: EndScreen
Role: Displays 'Game Over' as the title for the end screen if the player has failed the game, i.e the player has not scored enough points.
Method: render(float)
Attribute: titleText : String
Justification: Draws the end screen title as 'Game Over' for failure.

FR_WIN_CONDITION
Class: EndScreen
Role: Displays 'Game Over' and the users score if its large enough for the player to win along with the streaks achieved, and the leaderboard of past scores
Method: render(float)
Attribute: score : int, titleText: String
Justification: Draws the end screen title as 'Game Over' along with the score and any streaks, if the user achieved any, also displays the leaderboard of past scores

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.

NFR_PERFORMANCE
Relates to **FR_INTERACTION_TRIGGER** explained earlier above.

Requirement ID (from shall priority)	**Related Architecture**
UR_MOVEMENT	The player is able to move the avatar around the 2D map with the use of the WASD and 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 UR_INTERFACE	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 activities that have been used in the game are exercise in the gym, feeding the ducks and going into town. When the recreation building is selected the user is offered a choice of hours they would like to spend relaxing 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 three buildings where the studying action can be started from, the Computer Science Department, the Piazza and the Ron Cooke Hub. 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. The two buildings for eating are the Piazza and the Ron Cooke Hub. 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_INTERACTION	The player is able to interact with the buildings and complete an activity (study, eat, recreation) by clicking on the pop-up which shows up when the boolean attribute popupVisible in the MainGameScreen class is true.
UR_LEADERBOARD	The game keeps track of the top 10 scorers who complete the game through the class Leaderboards which writes the calculated score of a game to an excel file using the method writeToFile(). The class EndScreen displays the leaderboard and in the Leaderboards class the method readFromSaved() is used to retrieve the scores from the excel file.
UR_STREAKS	The player is able to get extra points if they do a specific activity, multiple times throughout the week. In the Score class, the method incrementStreakCounters() tracks the player's streaks.