Architecture

Group 12

Team 12

Matt Smith

Daniel Agar

Leyi Ye

Charles Thompson

James Leney

Vanessa Ndiangang

Previous versions of all architectural diagrams can be found on the website, under 'Interim Architecture Versions'

Structural Diagrams

Class Diagrams

To create the class diagrams we used the PlantUML extension and constructed the diagrams using UML code.

The original class diagram was how a simple game would be structured with little thought about the requirements in order to get an understanding of how the skeleton of the system should look. It contained many classes that later became redundant, as the libGDX library contained prebuilt versions of the classes, for example the Sprite class was unnecessary due to libGDX's Sprite class, instead creating the Player class which extended the Sprite class.

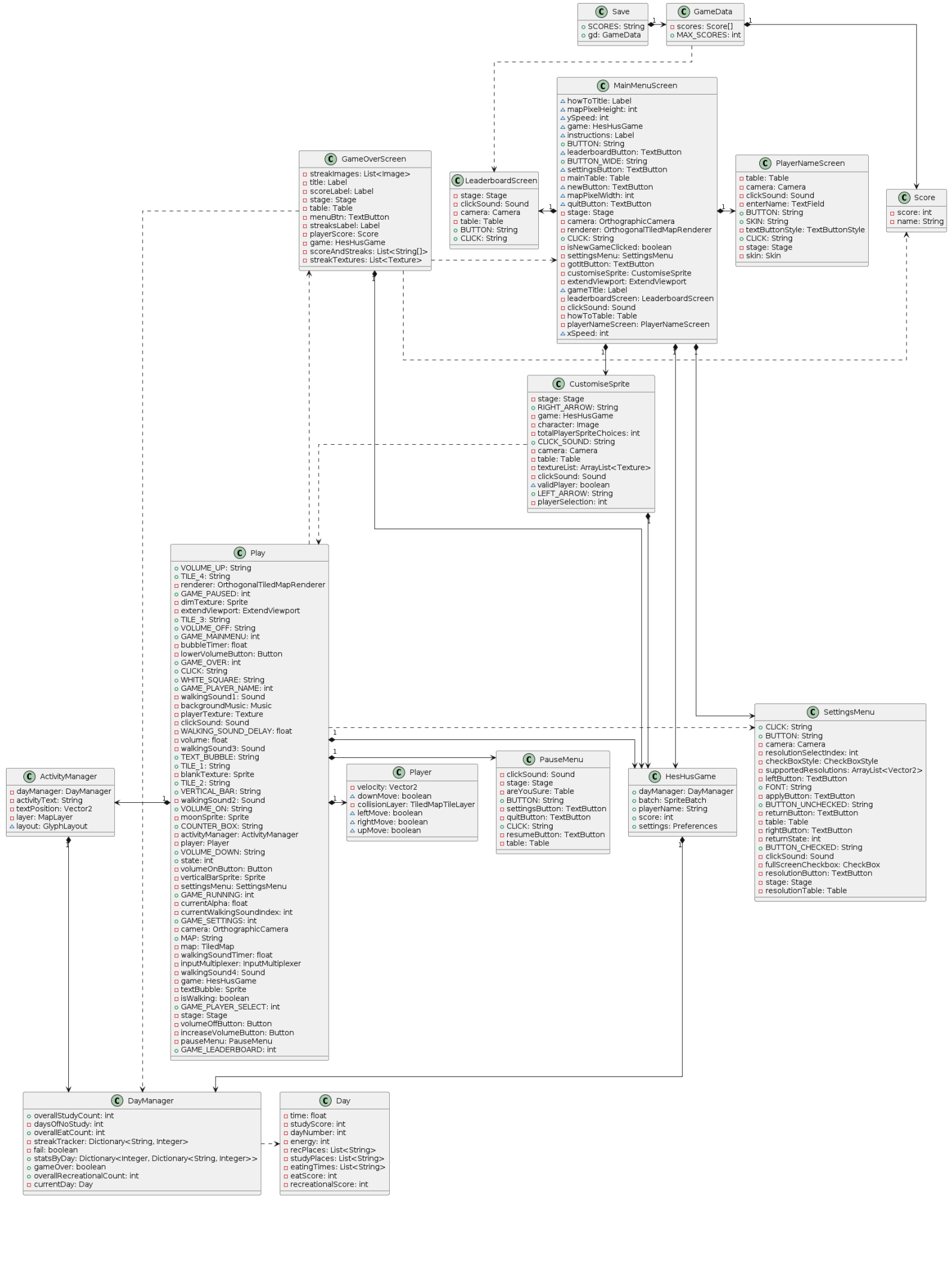
This led to the development of a new class diagram, primarily focused around game states, which were later replaced with screens (implementing the libGDX Screen class) due to simplicity. This class diagram also contained redundant classes such as the SoundManager, due to sound also being managed in libGDX.

The final class diagram shown below contains all the classes needed for the system to work, without making the system too complex with small unnecessary classes. Since the gameplay is not that complex, we limited the classes to controlling the flow of the game (all of which are functioning under the HesHusGame class) as well as decomposing the Play class into multiple related classes for easier management (DayManager, ActivityManager).

In relation to the requirements, the ActivityManager was created to control the different activities, the DayManager provides a place to track the different days and the Play/DayManager classes control the collection and displaying of the activity counters. FR_ENERGY_TIME_MANAGEMENT, FR_STATIC_TIME and FR_ACTIVITY_COUNTER are accounted for in the DayManager class, which implements a Day class allowing the game to progress through days, as well as play again if the player wants to. FR_FINISH, FR_PAUSE_MENU, FR_MAIN_MENU and UR_MAIN_MENU_NAVIGATION are all represented through different screens, controlled in classes extending the HesHusGame class. The ActivityManager uses the method checkActivity() in order to determine which activity is being interacted with, and therefore UR_CHOOSE_TASKS is complete.

The addition of LeaderboardScreen completes the new requirements UR_LEADERBOARD, FR_LEADERBOARD_DISPLAY and along with the new classes Score, GameData and Save to implement UR_SCORE, FR_LEADERBOARD_SCORE and FR_SCORE.

We also adapted previously created classes (evident when comparing to the previous diagrams) in order to account for the new requirements FR_ACTION_LOGS, FR_RECREATIONAL_ACTIVITY_TASK, UR_STREAKS, UR_STREAKS_ACHIEVABLE, FR_STREAKS, FR_STREAKS_DISPLAY and FR_ACTION_LOGS.


Save
SCORES: String
gd: GameData
GameData
scores: Score[]
MAX_SCORES: int
MainMenuScreen
howToTitle: Label
mapPixelHeight: int
ySpeed: int
game: HesHusGame
instructions: Label
BUTTON: String
leaderboardButton: TextButton
BUTTON_WIDE: String
settingsButton: TextButton
mainTable: Table
newButton: TextButton
mapPixelWidth: int
quitButton: TextButton
stage: Stage
camera: OrthographicCamera
renderer: OrthogonalTiledMapRenderer
CLICK: String
isNewGameClicked: boolean
settingsMenu: SettingsMenu
gotItButton: TextButton
customiseSprite: CustomiseSprite
extendViewport: ExtendViewport
gameTitle: Label
leaderboardScreen: LeaderboardScreen
clickSound: Sound
howToTable: Table
playerNameScreen: PlayerNameScreen
xSpeed: int
GameOverScreen
streakImages: List<Image>
title: Label
scoreLabel: Label
stage: Stage
table: Table
menuBtn: TextButton
streaksLabel: Label
playerScore: Score
game: HesHusGame
scoreAndStreaks: List<String[]>
streakTextures: List<Texture>
LeaderboardScreen
stage: Stage
clickSound: Sound
camera: Camera
table: Table
BUTTON: String
CLICK: String
PlayerNameScreen
table: Table
camera: Camera
clickSound: Sound
enterName: TextField
BUTTON: String
SKIN: String
textButtonStyle: TextButtonStyle
CLICK: String
stage: Stage
skin: Skin
Score
score: int
name: String
CustomiseSprite
stage: Stage
RIGHT_ARROW: String
game: HesHusGame
character: Image
totalPlayerSpriteChoices: int
CLICK_SOUND: String
camera: Camera
table: Table
textureList: ArrayList<Texture>
clickSound: Sound
validPlayer: boolean
LEFT_ARROW: String
playerSelection: int
Play
VOLUME_UP: String
TILE_4: String
renderer: OrthogonalTiledMapRenderer
GAME_PAUSED: int
dimTexture: Sprite
extendViewport: ExtendViewport
TILE_3: String
VOLUME_OFF: String
GAME_MAINMENU: int
bubbleTimer: float
lowerVolumeButton: Button
GAME_OVER: int
CLICK: String
WHITE_SQUARE: String
GAME_PLAYER_NAME: int
walkingSound1: Sound
backgroundMusic: Music
playerTexture: Texture
clickSound: Sound
WALKING_SOUND_DELAY: float
volume: float
walkingSound3: Sound
TEXT_BUBBLE: String
TILE_1: String
blankTexture: Sprite
TILE_2: String
VERTICAL_BAR: String
walkingSound2: Sound
VOLUME_ON: String
moonSprite: Sprite
COUNTER_BOX: String
activityManager: ActivityManager
player: Player
VOLUME_DOWN: String
state: int
volumeOnButton: Button
verticalBarSprite: Sprite
settingsMenu: SettingsMenu
GAME_RUNNING: int
currentAlpha: float
currentWalkingSoundIndex: int
GAME_SETTINGS: int
camera: OrthographicCamera
MAP: String
map: TiledMap
walkingSoundTimer: float
inputMultiplexer: InputMultiplexer
walkingSound4: Sound
game: HesHusGame
textBubble: Sprite
isWalking: boolean
GAME_PLAYER_SELECT: int
stage: Stage
volumeOffButton: Button
increaseVolumeButton: Button
pauseMenu: PauseMenu
GAME_LEADERBOARD: int
SettingsMenu
CLICK: String
BUTTON: String
camera: Camera
resolutionSelectIndex: int
checkBoxStyle: CheckBoxStyle
supportedResolutions: ArrayList<Vector2>
leftButton: TextButton
FONT: String
applyButton: TextButton
BUTTON_UNCHECKED: String
returnButton: TextButton
table: Table
rightButton: TextButton
returnState: int
BUTTON_CHECKED: String
clickSound: Sound
fullScreenCheckbox: CheckBox
resolutionButton: TextButton
stage: Stage
resolutionTable: Table
ActivityManager
dayManager: DayManager
activityText: String
textPosition: Vector2
layer: MapLayer
layout: GlyphLayout
Player
velocity: Vector2
downMove: boolean
collisionLayer: TiledMapTileLayer
leftMove: boolean
rightMove: boolean
upMove: boolean
PauseMenu
clickSound: Sound
stage: Stage
areYouSure: Table
settingsButton: TextButton
quitButton: TextButton
CLICK: String
resumeButton: TextButton
table: Table
HesHusGame
dayManager: DayManager
batch: SpriteBatch
playerName: String
score: int
settings: Preferences
DayManager
overallStudyCount: int
daysOfNoStudy: int
overallEatCount: int
streakTracker: Dictionary<String, Integer>
fail: boolean
statsByDay: Dictionary<Integer, Dictionary<String, Integer>>
gameOver: boolean
overallRecreationalCount: int
currentDay: Day
Day
time: float
studyScore: int
dayNumber: int
energy: int
recPlaces: List<String>
studyPlaces: List<String>
eatingTimes: List<String>
eatScore: int
recreationalScore: int

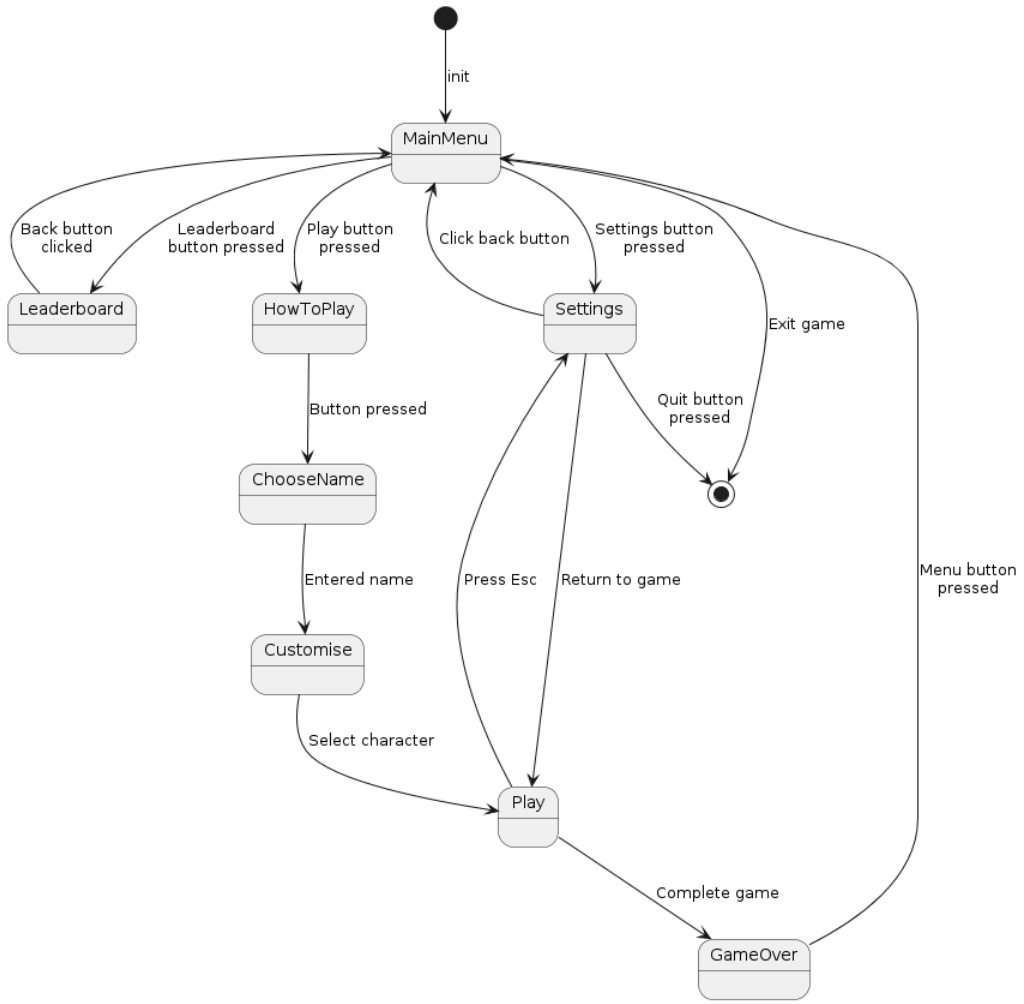
Behavioural Diagrams

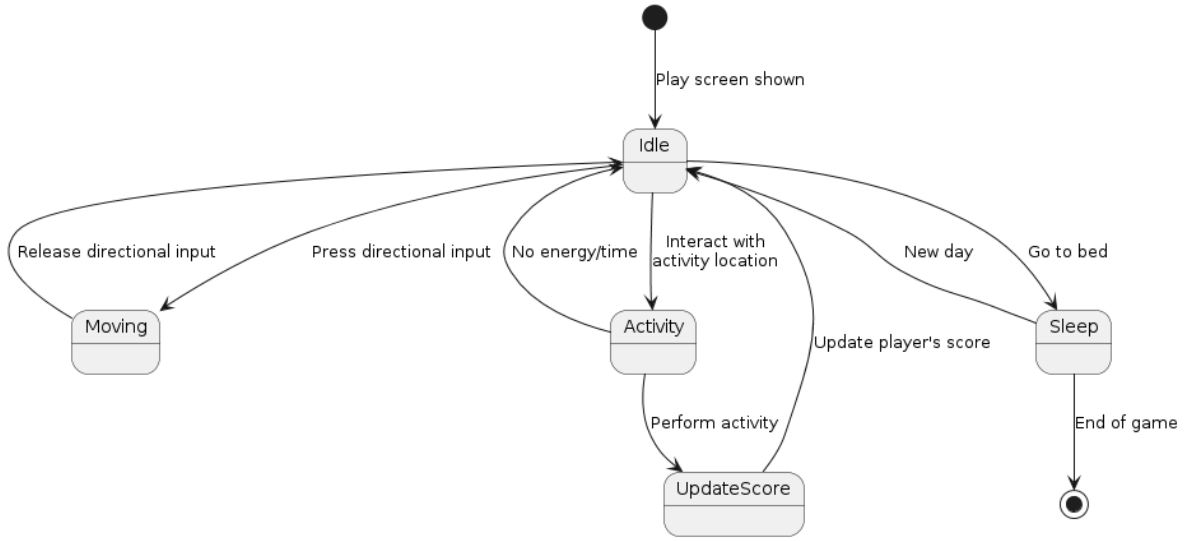
State Diagrams

The game being produced has 2 main flows to indicate how the game works:

- Flow of screens
- Flow of gameplay

These diagrams have been created using PlantUML and have mostly stayed the same throughout the different stages of development.
The only changes to the state diagrams are an additional GameOver branching off of play to enable the game to be replayed, in line with requirement UR_COMPLETE_GAME_SCREEN, as well as an intermediate state Customise to accommodate for UR_CUSTOMISE_AVATAR. The pause state allows the player to change their settings, even while the game is running (UR_PAUSE_MENU_VISIBILITY). The second state diagram describes what will happen during the play state of the original state diagram, which includes the description of a moving player (UR_CONTROL). The inclusion of the state NoMoreInteract falls in line with the requirement UR_SLEEP_FEATURE_TASK.
We decided to use multiple state diagrams because understanding the flow of the entire system was key to implementing the required classes; knowing what needed to happen at each stage of the game simplified this process.

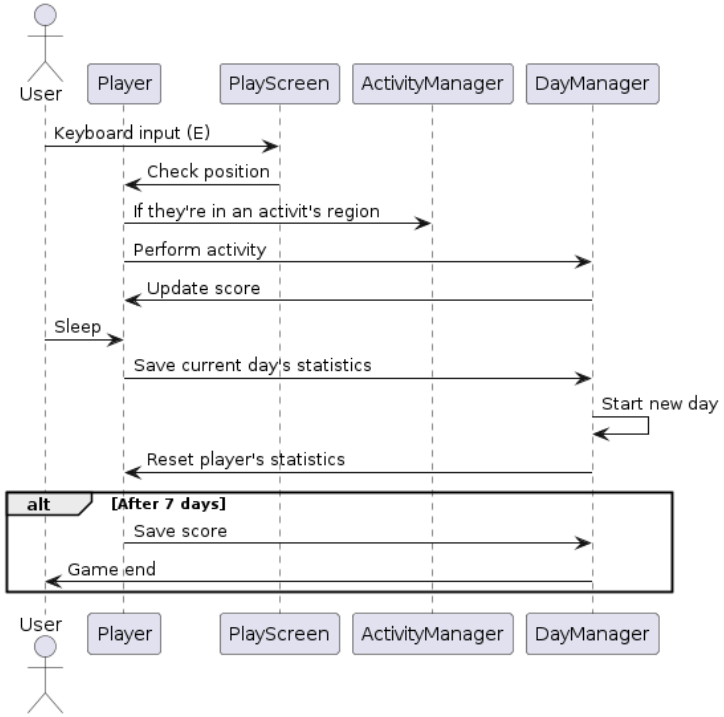




Sequence Diagrams

The sequence diagram for the main interaction logic is shown below. This is the most significant sequence of interactions between different classes. Our first iteration of this was based largely on the initial class diagrams, and requirements like UR_ENERGY_TIME_FEATURE and UR_CONTROL, which governed how this logic should work. UR_CHOOSE_TASKS governs a lot of the alternative cases, where the player runs out of energy or time left in the day.

As development continued and this logic was implemented, the structure of the interactions changed: some classes like ActivityManager were responsible for modifying their own internal variables, rather than passing this off as a method call to another Manager class like we initially planned. However, the core functionality and sequence of events in the diagram below stayed consistent throughout development.


User
Player
PlayScreen
ActivityManager
DayManager
Keyboard input (E)
Check position
If they're in an activit's region
Perform activity
Update score
Sleep
Save current day's statistics
Start new day
Reset player's statistics
alt
[After 7 days]
Save score
Game end
User
Player
PlayScreen
ActivityManager
DayManager