Architecture

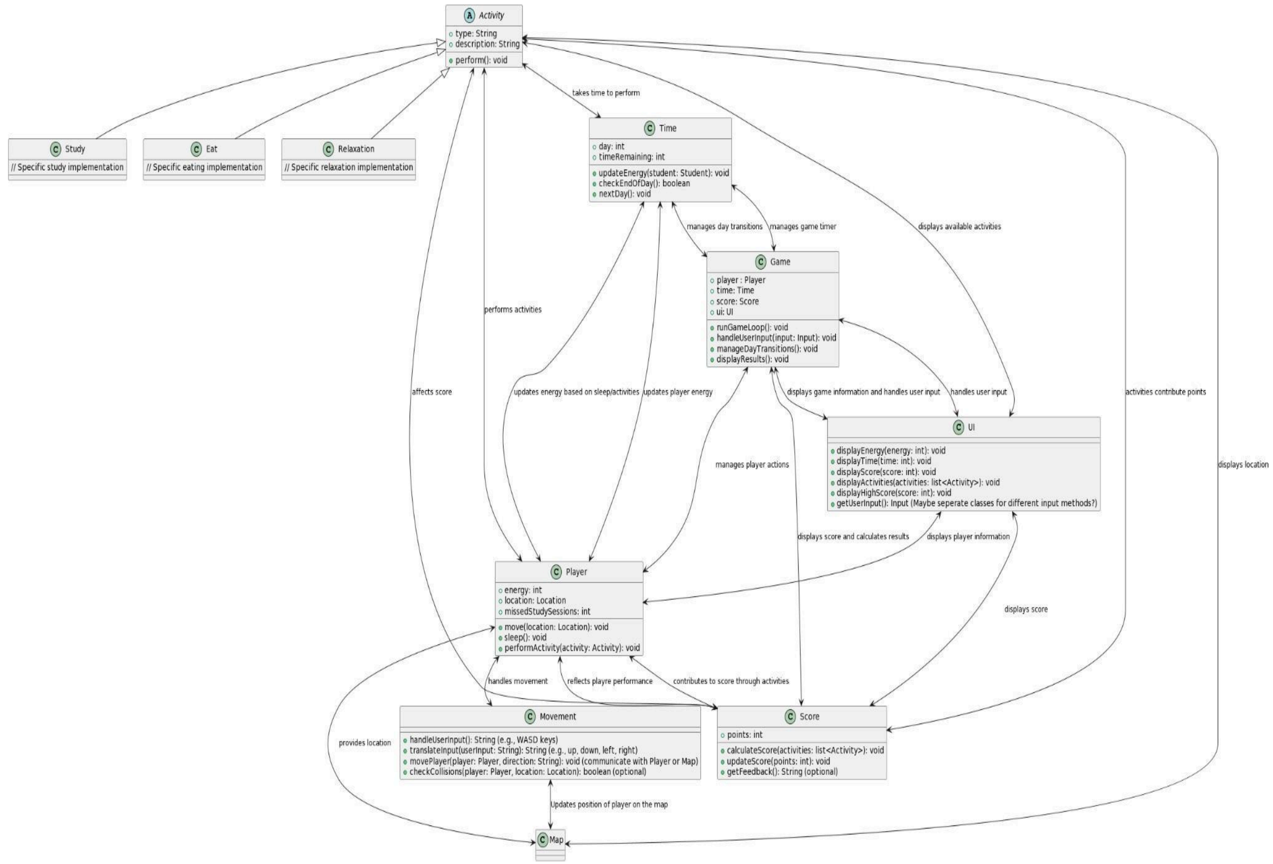
Cohort 1 Group 8

Roy Asiku, Tianqi Feng, Andrew Jenkins, Nicholas Lambert, Tom Byron

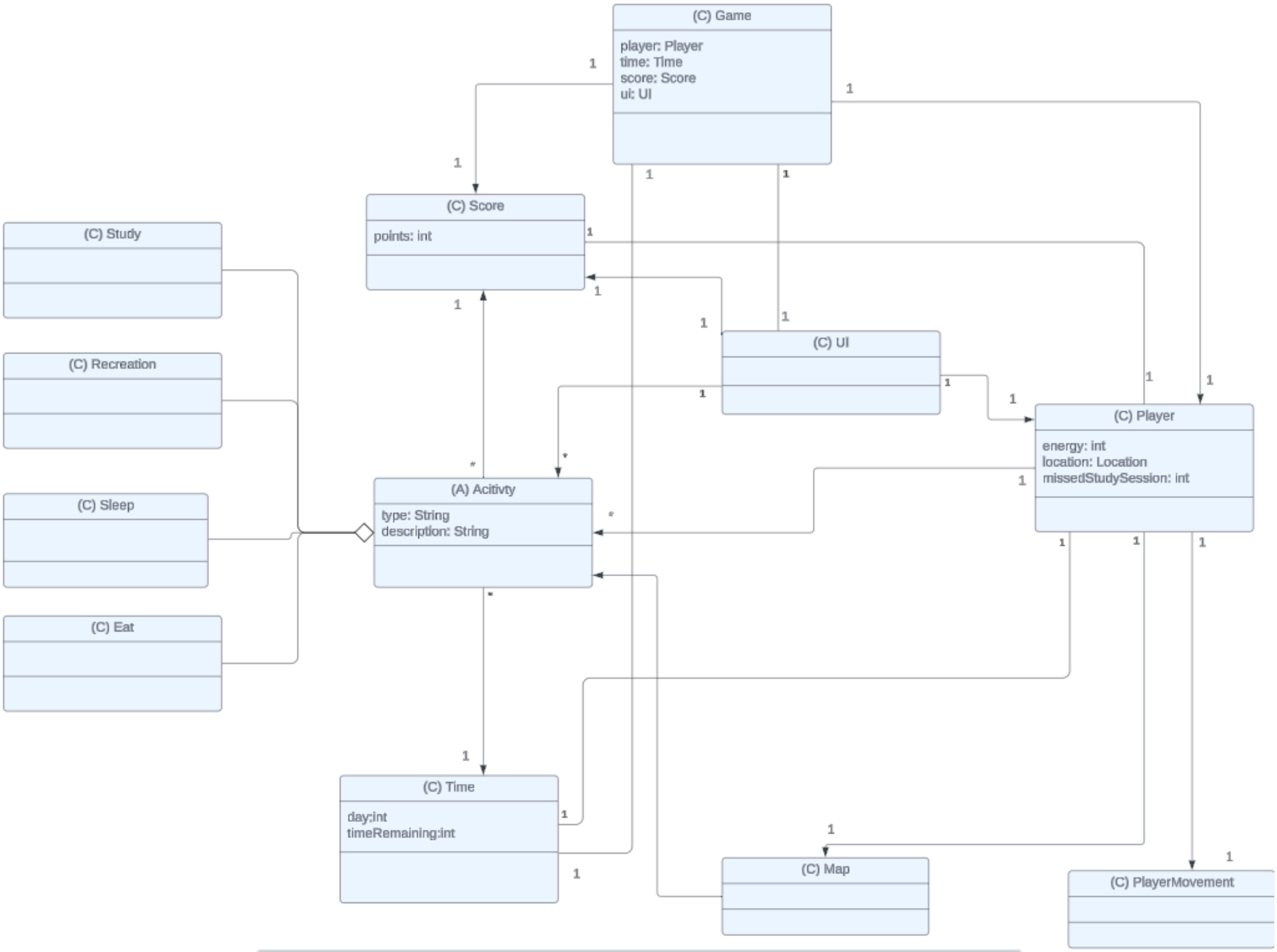
When creating our software architecture, we began by identifying the initial components. We adopted an ‘Event Storming’ approach to this, as we knew our game would consist of event-driven processes. In order to visualise how different components would interact, we chose to illustrate it using UML diagrams. UML would allow us to showcase the relationships between components and the sequence of events using a series of diagrams.

The UML diagrams below illustrates the relationship between components:

This is the first UML diagram made representing a general outline of how the game will be implemented, the first initial draft of this is made in PlantUML and is very messy. It’s hard to follow as arrows are all over the place and it lists an unnecessary amount of methods that are not needed in a UML diagram as it’s meant to be more abstract.
[DIAGRAM]


A Activity
type: String
description: String
perform(): void
C Study
// Specific study implementation
C Eat
// Specific eating implementation
C Relaxation
// Specific relaxation implementation
takes time to perform
C Time
day: int
timeRemaining: int
updateEnergy(student: Student): void
checkEndOfDay(): boolean
nextDay(): void
manages day transitions
manages game timer
displays available activities
C Game
player : Player
time: Time
score: Score
ui: UI
runGameLoop(): void
handleUserInput(input: Input): void
manageDayTransitions(): void
displayResults(): void
performs activities
affects score
updates energy based on sleep/activities
updates player energy
displays game information and handles user input
handles user input
activities contribute points
displays location
C UI
displayEnergy(energy: int): void
displayTime(time: int): void
displayScore(score: int): void
displayActivities(activities: list<Activity>): void
displayHighScore(score: int): void
getUserInput(): Input (Maybe seperate classes for different input methods?)
manages player actions
displays score and calculates results
displays player information
displays score
C Player
energy: int
location: Location
missedStudySessions: int
move(location: Location): void
sleep(): void
performActivity(activity: Activity): void
handles movement
reflects playre performance
contributes to score through activities
provides location
C Movement
handleUserInput(): String (e.g., WASD keys)
translateInput(userInput: String): String (e.g., up, down, left, right)
movePlayer(player: Player, direction: String): void (communicate with Player or Map)
checkCollisions(player: Player, location: Location): boolean (optional)
C Score
points: int
calculateScore(activities: list<Activity>): void
updateScore(points: int): void
getFeedback(): String (optional)
Updates position of player on the map
C Map

[UML1]



The UML has been revised in the LucidChart because it allows more freedom and flexibility to place the diagrams and have it be more clear and concise. It's way easier to follow. The methods have been removed to make the UML more abstract. Municipalities have been added to further show the relationship between the two classes.

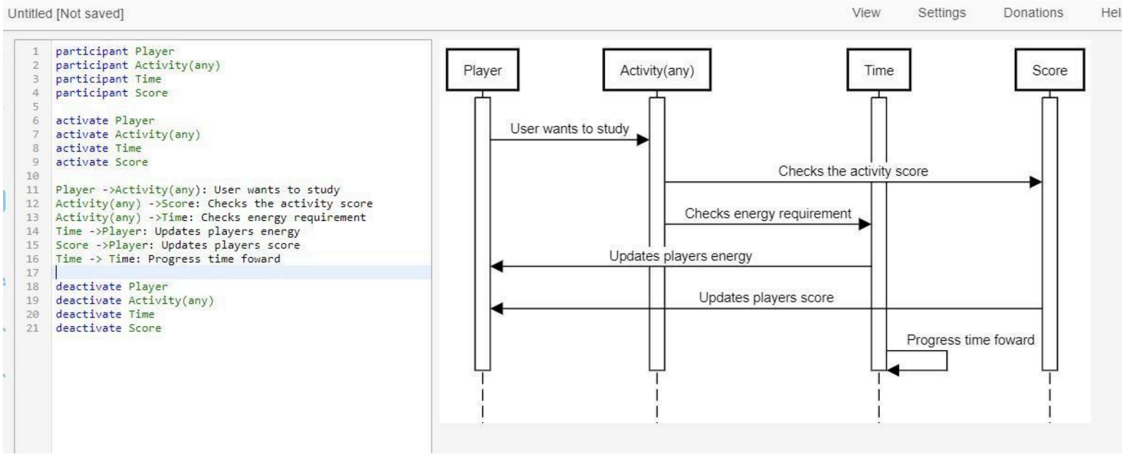
This UML diagram will serve as guidance for the developers to follow when creating the game. However, due to the complex nature of programming, this diagram may not be accurate post development. Developers may find more efficient solutions to implement certain features, or add/remove any additional or unnecessary features. This is why this UML diagram will be an iterative design, which will be updated upon every version of the game released. The UML diagram provides non-technical stakeholders with an insight into how the system will run and which entities have access to which resources.
This UML diagram successfully implements the following requirements:

- UR_ACTIVITIES
    - This user requirement is shown by the 'Study', 'Recreation', 'Sleep' and 'Eat' entities all inheriting from the 'Activity' entity. This is a clear implementation of the four types of activity the user can do.

- UR_SCORING

- This user requirement is shown by the directed association between the ‘Activity’ and ‘Score’ entities. This association is present as each activity has a completion score, which is used to calculate the ‘final exam’ result.

In order to visualise the order in which events occur, we created sequence diagram using sequencediagram.org



```
participant Player
participant Activity(any)
participant Time
participant Score

activate Player
activate Activity(any)
activate Time
activate Score

Player ->Activity(any): User wants to study
Activity(any) ->Score: Checks the activity score
Activity(any) ->Time: Checks energy requirement
Time ->Player: Updates players energy
Score ->Player: Updates players score
Time -> Time: Progress time foward

deactivate Player
deactivate Activity(any)
deactivate Time
deactivate Score
```

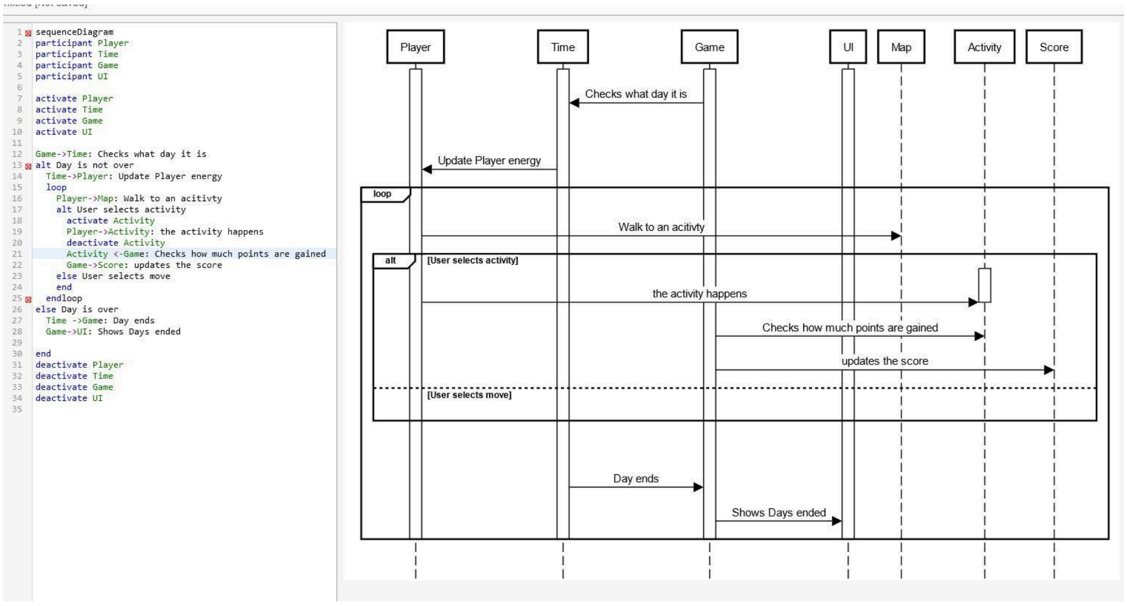
[SD1]

The first sequence diagram aims to showcase the interaction between the ‘Player’, ‘Activity’, ‘Time’ and ‘Score’ entities when the user attempts to complete an activity (e.g. studying). It concisely shows the order of interactions between the components listed above. It is a simplistic sequence diagram which will be built upon to create more holistic diagrams.

This sequence diagram successfully implements the following requirements:

- FR_POINT_ALLOCATION
  - This functional requirement is shown in the final interaction between the ‘Player’ and ‘Score’ entities where the ‘Score’ entity *Updates player’s score*.

- FR_ACTION_ENERGY_ALLOCATION
  - This functional requirement is shown in the final interaction between the ‘Player’ and ‘Time’ entities where the ‘Time’ entity *Updates player’s energy*.



```
sequenceDiagram
participant Player
participant Time
participant Game
participant UI

activate Player
activate Time
activate Game
activate UI

Game->Time: Checks what day it is
alt Day is not over
  Time->Player: Update Player energy
  loop
    Player->Map: Walk to an acitivty
    alt User selects activity
      activate Activity
      Player->Activity: the activity happens
      deactivate Activity
      Activity <-Game: Checks how much points are gained
      Game->Score: updates the score
    else User selects move
    end
  endloop
else Day is over
  Time ->Game: Day ends
  Game->UI: Shows Days ended

end
deactivate Player
deactivate Time
deactivate Game
deactivate UI
```

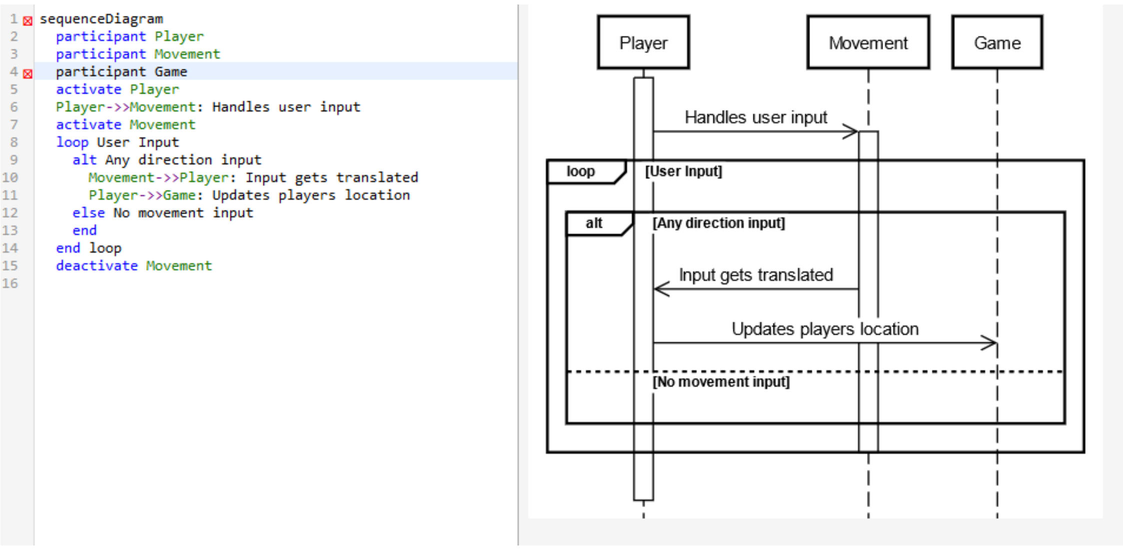
[SD2]

The second sequence diagram shows the sequence of interactions between the ‘Player’, ‘Time’, ‘Game’, ‘UI’, ‘Map’, ‘Activity’ and ‘Score’ entities with relation to the day cycle. It utilises a loop (*loop*) to portray the 7 days that happen in our game, and an embedded loop (*alt*) to show the multiple activities that could occur during one day.

This sequence diagram successfully implements the following requirements:

- FR_POINT_ALLOCATION
    - This functional requirement is shown within the *alt* loop - where after *the activity happens*, the ‘Game’ entity will *check how many points are gained* with the ‘Activity’ entity.

- UR_TIME
    - This user requirement is clearly shown within the loop (*loop*). Where after a set amount of time (indicated by the ‘Time’ entity) the day will automatically end without the user being able to do anything.



```
sequenceDiagram
  participant Player
  participant Movement
  participant Game
  activate Player
  Player->>Movement: Handles user input
  activate Movement
  loop User Input
    alt Any direction input
      Movement->>Player: Input gets translated
      Player->>Game: Updates players location
    else No movement input
    end
  end loop
  deactivate Movement
```

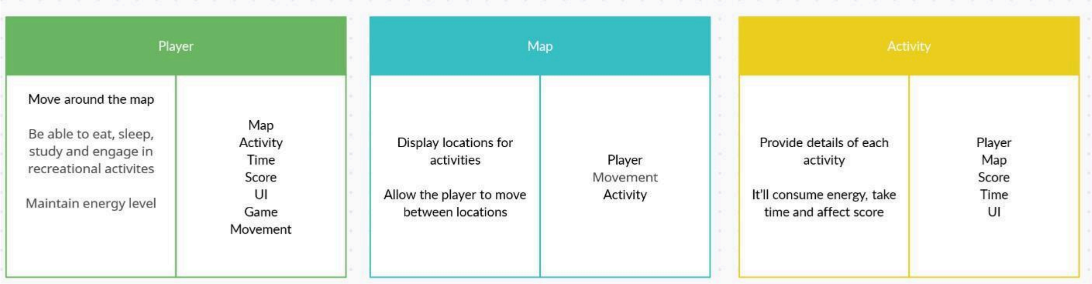
[SD3]

The third sequence diagram shows the sequence of interaction between the ‘Player’, ‘Movement’ and ‘Game’ entities. It utilises a loop (*loop*) to recognise any user input, and an embedded loop (*alt*) to handle any directional input.

This sequence diagram successfully implements the following requirements:

- FR_USER_MOVEMENT
  - This functional requirement is clearly shown within the *alt* loop - where if the user inputs *any direction input*, the ‘Movement’ entity translates the input and passes that to the ‘Player’ entity, who then *updates player’s location* and passes that to the ‘Game’ entity

Lastly, we needed to create diagrams to showcase class structure within Lucid Chart. We chose to create Class-responsibility-collaboration (CRC) cards as they can aid in visualising the responsibilities of each class within the system. CRC cards are simple, which means they are understandable from both technical (developers) and non-technical (stakeholders) perspectives. Below are the CRC cards we created to support the development of our system.

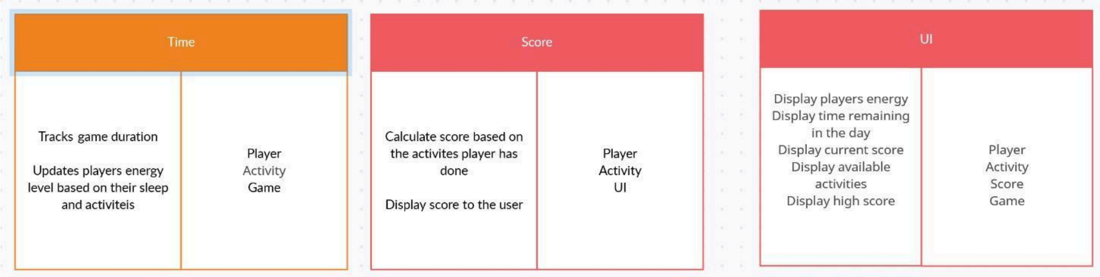


Player	
Move around the map Be able to eat, sleep, study and engage in recreational activites Maintain energy level	Map Activity Time Score UI Game Movement

Map	
Display locations for activities Allow the player to move between locations	Player Movement Activity

Activity	
Provide details of each activity It'll consume energy, take time and affect score	Player Map Score Time UI

[CRC1]

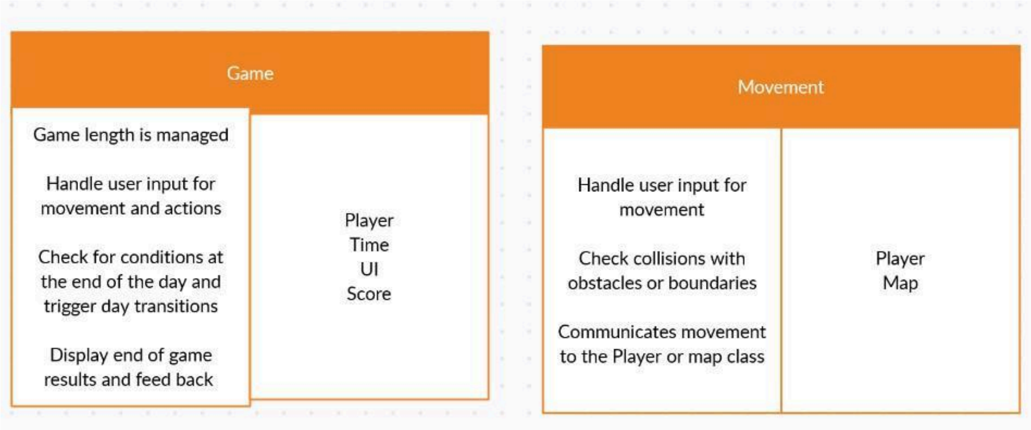


Time	
Tracks game duration Updates players energy level based on their sleep and activiteis	Player Activity Game

Score	
Calculate score based on the activites player has done Display score to the user	Player Activity UI

UI	
Display players energy Display time remaining in the day Display current score Display available activities Display high score	Player Activity Score Game

[CRC2]



Game	
Game length is managed Handle user input for movement and actions Check for conditions at the end of the day and trigger day transitions Display end of game results and feed back	Player Time UI Score

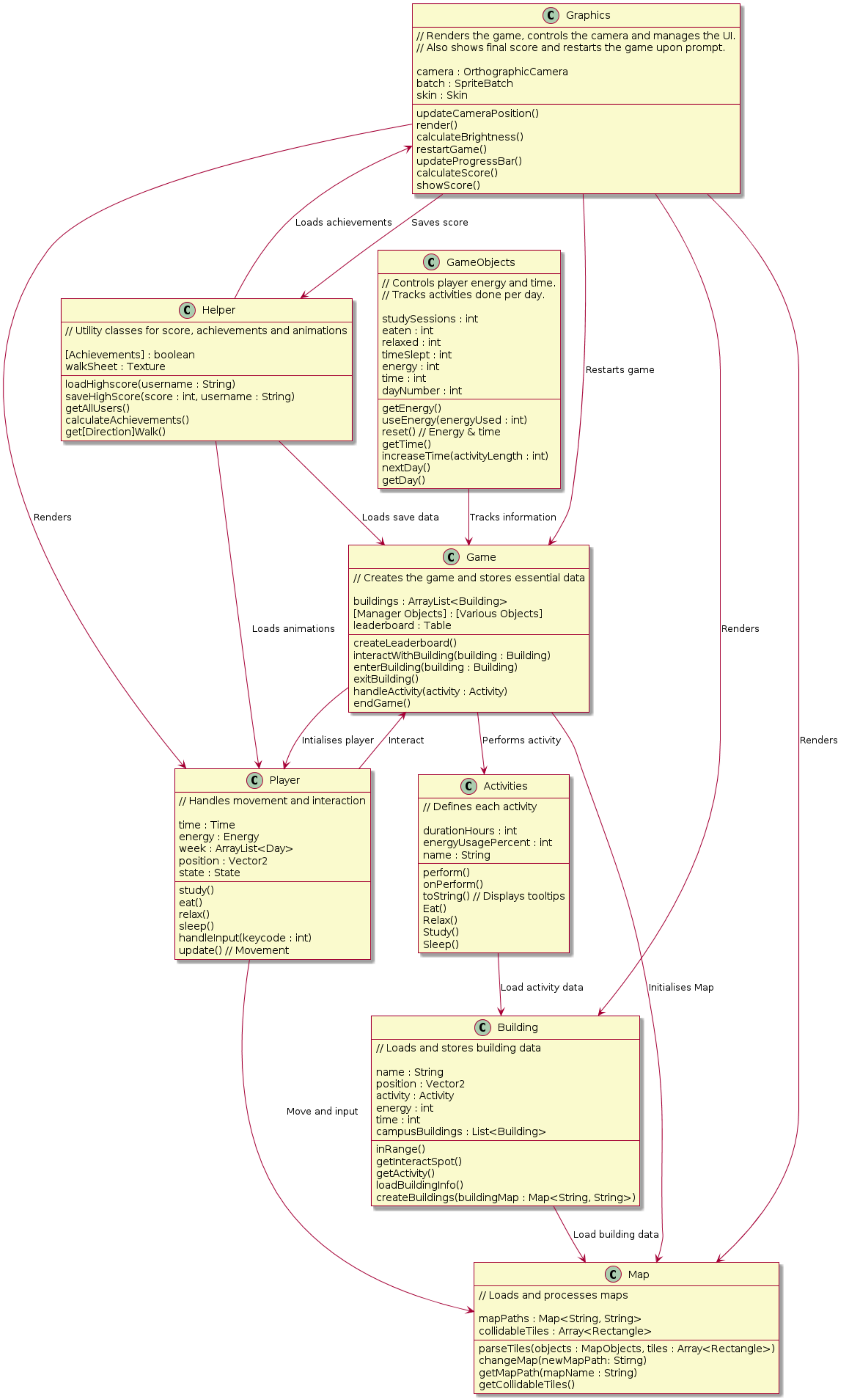
Movement	
Handle user input for movement Check collisions with obstacles or boundaries Communicates movement to the Player or map class	Player Map

[CRC3]

Each of the above cards represents one of the entities within the class UML diagram. Using CRC cards alone isn’t great for understanding the functionality of a system, as they dont show how each class interacts - but when used in conjunction with class UML diagrams and UML sequence diagrams, they become incredibly useful as they better show the functionality of each class.

Updates to Architecture in Part 2

Upon picking up the project and comparing its architecture with its implementation, we noticed that there were some aspects that were unclear about how specific classes interact with each other and the game's structure as a whole. It is also unclear as to whether or not each entity is a separate class, as some of these entries are individual classes whereas others seem to be referring to whole packages. To correct this, we decided to revise the project's UML class diagram to better reflect its architecture. Since the game is split up into a large amount of different classes, many of which overlap in their functionality, we decided that it would be more beneficial to split the project into packages on our UML diagram. So each class on the diagram is actually a package in the code of the project. Additionally, we included the extra requirements introduced for part 2 of the project, FR_ACHIEVEMENTS_SHOW and FR_LEADERBOARD_SHOW; as seen in the Helper package's variables, functions and connections.


C Graphics
// Renders the game, controls the camera and manages the UI.
// Also shows final score and restarts the game upon prompt.
camera : OrthographicCamera
batch : SpriteBatch
skin : Skin
updateCameraPosition()
render()
calculateBrightness()
restartGame()
updateProgressBar()
calculateScore()
showScore()
Loads achievements
Saves score
C GameObjects
// Controls player energy and time.
// Tracks activities done per day.
studySessions : int
eaten : int
relaxed : int
timeSlept : int
energy : int
time : int
dayNumber : int
getEnergy()
useEnergy(energyUsed : int)
reset() // Energy & time
getTime()
increaseTime(activityLength : int)
nextDay()
getDay()
C Helper
// Utility classes for score, achievements and animations
[Achievements] : boolean
walkSheet : Texture
loadHighscore(username : String)
saveHighScore(score : int, username : String)
getAllUsers()
calculateAchievements()
get[Direction]Walk()
Restarts game
Renders
Loads save data
Tracks information
C Game
// Creates the game and stores essential data
buildings : ArrayList<Building>
[Manager Objects] : [Various Objects]
leaderboard : Table
createLeaderboard()
interactWithBuilding(building : Building)
enterBuilding(building : Building)
exitBuilding()
handleActivity(activity : Activity)
endGame()
Loads animations
Renders
Intialises player
Interact
Performs activity
Renders
C Player
// Handles movement and interaction
time : Time
energy : Energy
week : ArrayList<Day>
position : Vector2
state : State
study()
eat()
relax()
sleep()
handleInput(keycode : int)
update() // Movement
C Activities
// Defines each activity
durationHours : int
energyUsagePercent : int
name : String
perform()
onPerform()
toString() // Displays tooltips
Eat()
Relax()
Study()
Sleep()
Load activity data
Initialises Map
C Building
// Loads and stores building data
name : String
position : Vector2
activity : Activity
energy : int
time : int
campusBuildings : List<Building>
inRange()
getInteractSpot()
getActivity()
loadBuildingInfo()
createBuildings(buildingMap : Map<String, String>)
Move and input
Load building data
C Map
// Loads and processes maps
mapPaths : Map<String, String>
collidableTiles : Array<Rectangle>
parseTiles(objects : MapObjects, tiles : Array<Rectangle>)
changeMap(newMapPath: Stirng)
getMapPath(mapName : String)
getCollidableTiles()