Architecture

Cohort 1 Group 10

Cai Hughes<cabh500@york.ac.uk>
Ben Slater<bs1463@york.ac.uk>
Adeola Adeniji<aa3098@york.ac.uk>
Mathew Riedy<mr1723@york.ac.uk>
Riad Kasmi<rmk526@york.ac.uk>
Simon Konieczny<sk2144@york.ac.uk>

Architecture

The implementation of the product will require the use of object-oriented design techniques using the Java language to create a desktop program **(Requirement: UR10_Platform_Compatibility)**. This entails setting up multiple classes for each component of the game which can be utilised by a main java file. The modularity of this architecture design will assist in making troubleshooting and collaboration easier, as well as making it simpler to add additional features if the customer requests them or if their requirements change.

To start, we needed to understand what classes we might need for the design, so we created Class Responsibility Collaborator (CRC) Cards to identify what classes we may need, the information they need to know, actions they must be able to complete and other classes they may need to interact with.

Initial Designs

Initially, our CRC cards and subsequent class diagram only considered a generic class, "**Event**" the types for all map locations that didn't have any way to specify what type of location it was and had no way to differentiate what type of actions should occur when interacting with different locations.

In an attempt to remedy this, we created a second design with a new abstract class, **MapInteractable**. We also changed the Event class to be an abstract class, and created 3 new classes, **Activity**, **StudyArea** and **RestArea(Requirements:FR6, UR5_Game_Objective)**. Activity and StudyArea are subclasses of Event and RestArea is a subclass of MapInteractable. This allows each of these subclasses to override their methods, allowing different interactions between each interactable map location.

This approach, although better, would force the scoring to be decided by the activity and not the player, which would not allow the points the player earned to change based on other factors such as the facilities they used, how many times they used each type of facility or the order in which they used each type of facility.

This led us to create another design, similar to the first one, where there is only a generic class "**Interactables**", however this time, we created an enum, "**Location**", containing constants describing the different types of locations the game contains. This allows us to assign a constant from the enum to each instance of the Interactable class, which allows differentiation between the different types of interactable locations and also allows the player to be in control of the result of these interactions allowing for better tracking and scoring variability.

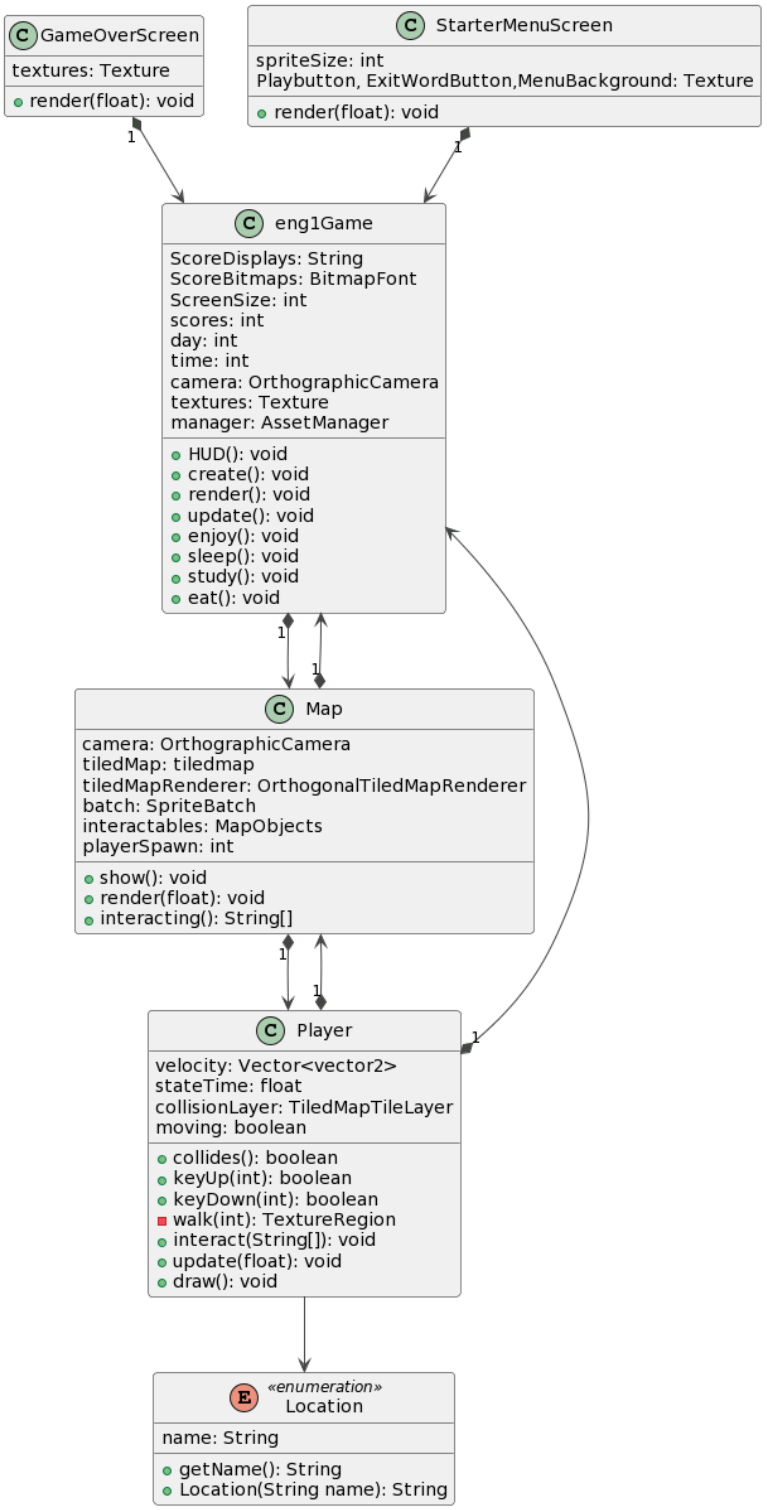
Below are links to the CRC cards in interim versions of the class diagram:

- CRC cards
- Initial class diagram
- Second class diagram

Class Diagram

After considering the classes we will use for our design and their responsibilities, we created a UML class diagram using PlantUML to show the classes that will be included in the implementation, the relationships between them and key attributes and methods that the classes require..

Some of the classes contain sprite and vector data types. These are implemented by LibGDX, the Java game development framework the project will be using.



Map

The Map class contains and displays everything that exists within the game world.

Attributes

- **SpriteBatch** Batch - For repeat textures, it is more efficient to collect the textures and send them as a group to the GPU to be drawn, this is what the SpriteBatch class does.
- **OrthographicCamera** camera - The camera allows the player to see the map and move through the game.
- **TiledMap** tileMap - Loads the current location and displays it to the screen.
- **OrthogonalTiledMapRenderer** tiledMapRenderer - Renders an orthogonal (top-down) tile map.
- **MapObjects** interactables - Loads the sections of the tilemap that the player can interact with, such as the study, eating, exercising areas.

Methods

- show() - When a screen gains focus this is called to display the specified screen..
- render() - Renders the current screen so that it is displayed to the user.

- interacting() - Returns the name of the object that is interacted with, allowing the user to gain points or sleep.

Location

The location enum stores values defining the names of the locations in the game, allowing the player to move through different areas of the game map.

Values

- **String** name - Stores the current location name.

Methods

- getName() - Returns the string of the location name.
- Location(String name) - sets the current location to the name variable.

Player

The player class represents who the user is playing as. It has limited energy per the user requirements that can be expended through events.

Attributes

- **Vector2 float** velocity - A 2D vector storing the velocity of the player.
- **float** stateTime - Used to calculate the speed for the player animation
- **TiledMapLayer** collisionLayer - Stores the areas where the player cannot go, constraints the player to stick to the map.
- **boolean** moving - stores whether the player is moving or not to determine what should be drawn to the screen

Methods

- collides() - Returns a boolean value if there is a collision on attempted movement, if false is returned the player can move, if true is returned the player cannot.
- keyDown() - Detects what key is pressed to determine which direction the player is trying to move. Uses WASD keys. **(Requirements UR2_GAME_CONTROLS, FR3**, the game supports WASD movement, which is a simple and well known control scheme.**)**
- keyUp() - When the player stops pressing a key it stops all player movement. **(Requirements UR2_GAME_CONTROLS, FR3**, the game supports WASD movement, which is a simple and well known control scheme.**)**
- interact(String) - Determines the type of activity the player is interacting with to properly allocate points, or changes the location based on the player's movement.

GameOverScreen

Displays the score the user gets at the end of the game, grading them either an A,B,C, or F. **(Requirement: UR7_Game_Scoring**, the game gives a score and grade, based on typical and widely understood school grading systems, incentivising the player to improve their score for a better grade.**)**

Attributes

- **Texture** textures - various textures to display the grade achieved.

Methods

- render() - draws the achieved grade to the screen.(**Requirements: UR8_Game_Difficulty, the game can be completed with both low and high scores, with different endings for better levels of play**)

StarterMenuScreen

First screen the player sees, gives them the ability to start the game.

Attributes

- **Int** spriteSize - values that define the size of the interactable buttons on the screen.
- **Texture** textures - textures for the different elements drawn to the screen

Methods

- render() - draws the start menu to the screen

eng1Game

This class runs the main game loop. It handles creating the camera, loading the map, displaying the screens and keeping track of the scores, days, and time remaining.

Attributes

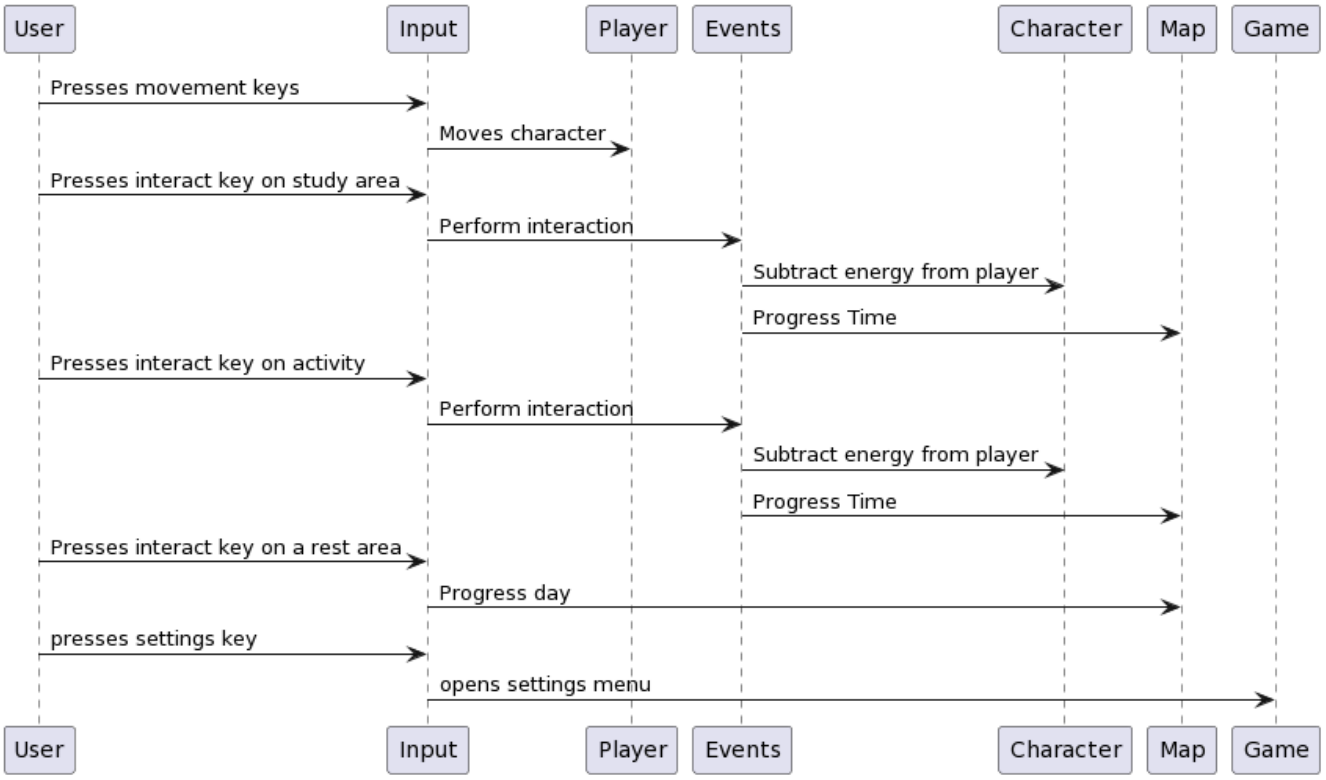
- **String** ScoreDisplays - A string representation of the players score. **(Requirement: UR9_Score_Tracking)**
- **BitMapFont** bitMapFonts - Writes the scoreDisplays to the screen.
- **Int** screenSize - sets the size of the screen
- **Int** scores, day, timer, energy - stores the key functionality for the game, updated as the player interacts with the game.(**Requirement:UR4_Game_Time, day; UR5_Game_Objective time, energy and timer**)

Methods

- HUD() - draws the current day and time to the screen as well as handles ending the game if the day counter crosses over the 7 day limit.
- create() - ensures the first screen loaded is the start menu, loads the assets, and creates the bitmap fonts for the scores.
- render() - renders the screens to the display and adds the scores to the screen, starts the music.
- eat() - increases the players eating score, decrements the timer, decreases energy.(**Requirement: UR9_Score_Tracking, FR6**)
- enjoy() - increases the players enjoyment score from doing recreational activities, decrements the timer, decrements energy.(**Requirement: UR5_Game_Objective, FR6**)
- study() - increases the players study score, decrements the timer, decreases the energy count.(**Requirement: UR5_Game_Objective, FR6**)
- sleep() - increases the day count, resets the time and energy totals. (**Requirement: UR5_Game_Objective, FR6**)

Sequence Diagram

A sequence diagram was additionally created to show some of the expected interactions between the objects in the system during runtime. It was created using PlantUML and shows our expected implementation of gameplay in order to adhere to our collected User Requirements.



User

The user has the ability to move around the map using the arrow keys, and can interact with areas in the environment. These interactions can be multiple different events each with their own effects on the user character.

Input

On input from the user, different methods are called which progress the game. This can be moving the player character, interacting with events, and opening the settings menu.

Events

The requirements for the game are to have multiple different types of events. There needs to be space to study, places to exercise/socialise, and one place to sleep. These events update both the character and map classes.