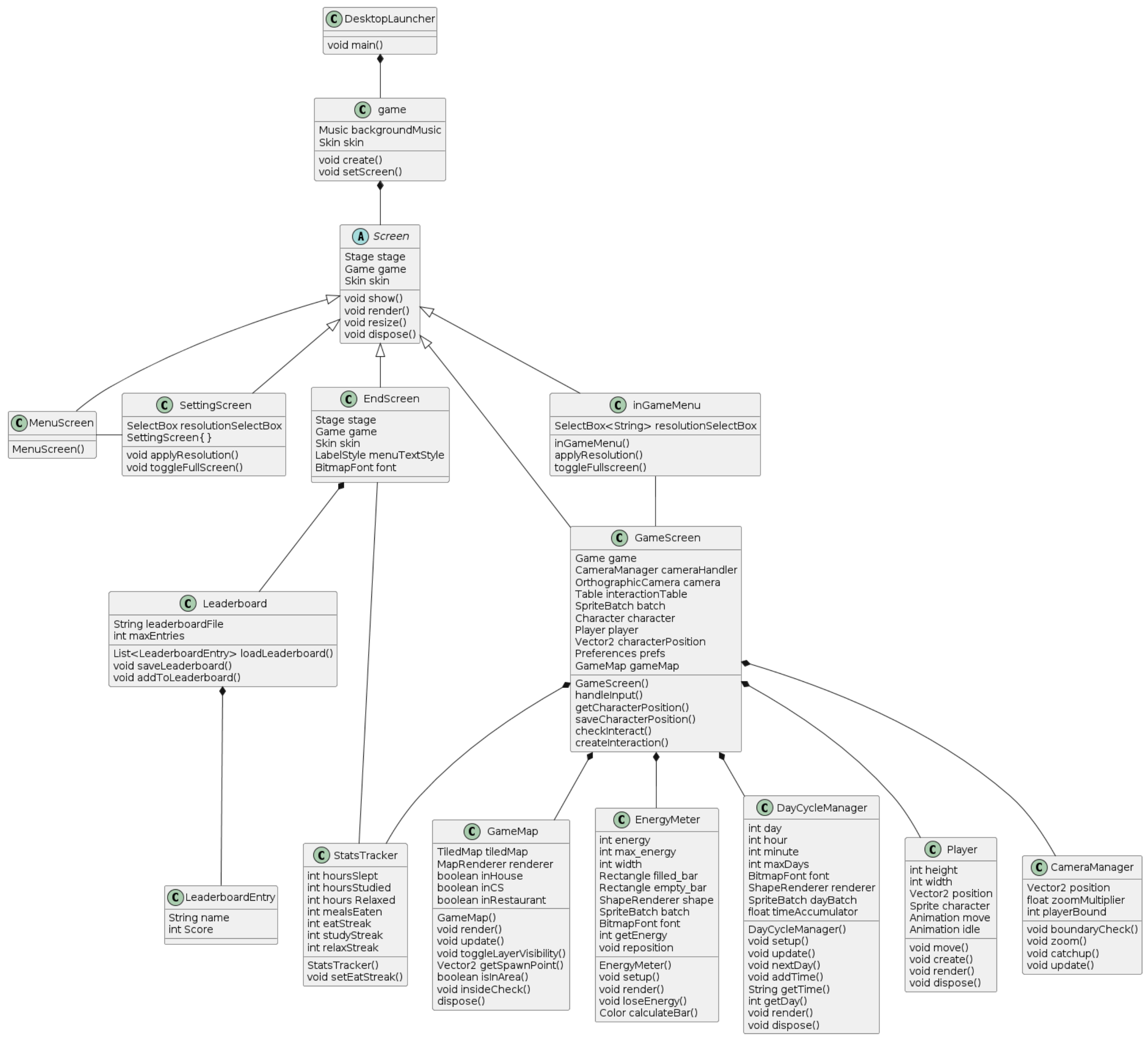
Architecture

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>

Our System Architecture



This class diagram was made using PlantUML and it shows the relations between all the classes in our architecture, as well as their attributes and methods. Please use the embedded link as it is too large to fit in here.

UML was used as it allows us to create structural diagrams to assist in visualising the system architecture, and provides a clean method of updating and expanding our diagrams as the scope of the project increases in size. PlantUML is an online tool used to create architectural diagrams. It can be accessed either directly from its website, or it can be used as a plugin in certain documentation software such as Google Docs. Using PlantUML provided an easy and simple way to generate these architectural diagrams and could be edited directly on our documentation.

When we began our initial system architecture design, we looked to common software architectural styles to try and match our problem to. There was a bit of difficulty, however, as it seemed that our system was quite different from the ones which were implemented with these architectural styles - they tended to be for embedded systems or websites, which were required to be secure and memory efficient. These features would of course be ideal in our system also, but they were far from the main concern, as our game would be very simple, requiring little memory and not containing any sensitive data to protect.

Therefore, our architecture ended up being outside of any of these definitions of architectural styles, but it most closely resembled the plug-in style. Our game was designed to be packaged and installed on a single customer machine and can be split up into components which could be considered core components (graphics and player components) and plug-in components (interaction and score components). The game functions without the interaction and score components, and these could easily be modified without affecting the core game, however the opposite is not true.

Our architecture developed this way because it allowed us more flexibility in our development of the score and interaction components. By designing and implementing the core components first, we gained an understanding of how these components actually functioned when implemented and then used that to optimally design our "plug-in" components, rather than designing every component before-hand and then having to change our designs.

There were two main component changes between our initial and our final designs. The first change was concerning the user interface. Before we had done any of the actual programming, we thought that we would just have one component which controlled everything the player saw, and that this would be the user interface. Once we began programming, it quickly became clear that we would need to split this into several different components. First of all, we wanted the player to be able to move throughout the map, but not to be able to see the whole map at once, to provide more of a Stardew Valley/Pokemon experience as these were games which the stakeholders suggested we could take inspiration from. In order to achieve this, we needed a camera object to keep the player in frame. As well as this, we wanted to display several different screens - the start screen, game screen, pause screen. We realised that these screens were all very similar and had mostly the same attributes and methods, so we created the abstract class Screen which they could all extend. For all the other elements of the HUD, such as the day counter, energy metre etc., we decided to render these in their own classes.

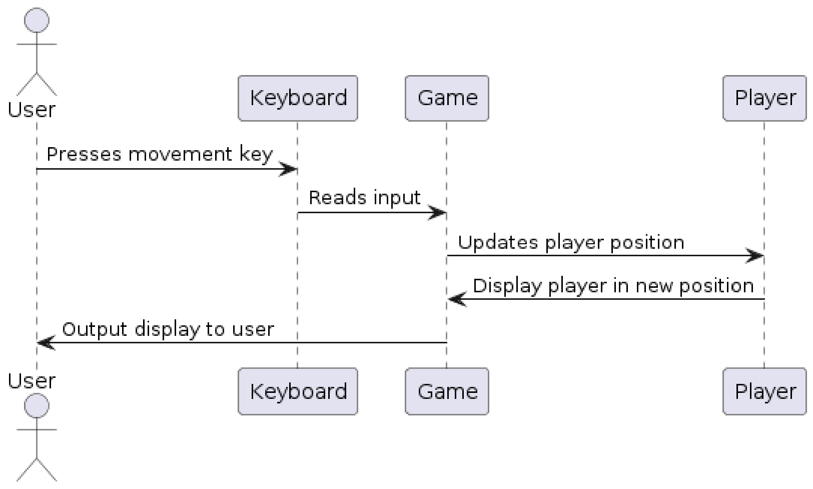
The second component change concerned the player interaction. In our initial design, we thought that we could have an abstract class Activity which they could all extend and this would have attributes such as activity duration, energy cost etc. However, our final implementation included the methods for player interaction in the GameScreen class, using methods from EnergyMeter and DayCycleManager.

Link to first version of architectural diagram.

How our Architecture relates to our Requirements

The GameMap class uses Tiled maps to manage the game's map, handling layer visibility, collision detection and providing spawn points for the player. GameMap contains toggleLayerVisibility() which is used to remove/replace the roofs of buildings as the player walks into them, allowing for a seamless entry with no need for a separate area or any loading screens to be created. This helps us fulfil the requirement NFR_ANIMATIONS. The isInArea() method is also part of the class and it is an important part of the collision detection, which is necessary to satisfy requirements FR_MAP_BOUNDARY and FR_MOVEMENT_RESTRICTIONS.

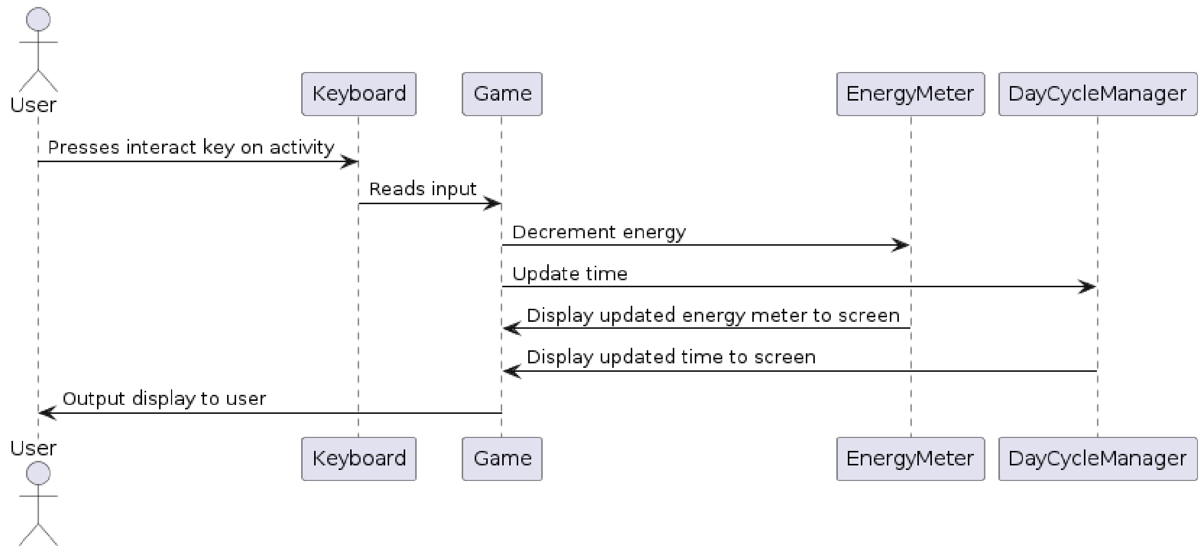
The Player class contains attributes used to render the character such as its texture, width and height, but its most important function is player movement. It does this with the help of isInArea() from gameMap, which it uses for collision detection. This helps us to fulfil the FR_MOVEMENT_CONTROLS requirement. Below is a sequence diagram displaying some of the expected stages in an interaction allowing the player to move.



The DayCycleManager class controls the in-game time and therefore plays a role in all of the activities that are completed. Whenever an activity is completed, addTime() is called to progress the in-game clock, satisfying the FR_TIME requirement. This class is also responsible for displaying the day count and in-game time on the player's HUD, which satisfies requirements FR_CLOCK, FR_DAY_NUMBER and NFR_PRECISE_TIME.

The EnergyMeter class is involved in any activity which affects the player's energy e.g. studying and sleeping. It contains attributes to keep track of the player's energy and methods to update and reset it, as well as to display the metre on screen. This gives the user a visual representation of their energy bar, satisfying requirement FR_ENERGY_BAR, and in conjunction with methods in the GameMap class it consumes player energy with each interaction and prevents an interaction if the player hasn't got enough energy left, satisfying requirements FR_INTERACTION_ENERGY and FR_DEPLETED_ENERGY.

Below is a sequence diagram depicting an interaction between the user and the game, where the user interacts with an activity and the game updates the energy meter and time of day using the previously mentioned EnergyMeter and DayCycleManager classes.



The Screen class is an abstract class which is extended by all of our GUI classes. It contains the attributes and methods which are found in all of the screen sub-classes, mainly for holding, showing and rendering elements in the GUI to create graphics.

The GameScreen class is the most important class which extends Screen. This is the main screen which the user plays the game on, and it is responsible for housing gameplay graphics as well as all player-map interaction functions. This class renders the map and creates interaction menus when the player interacts with any of the activities, as well as using the isInArea method from GameMap to display an “Interact” prompt when appropriate. This, alongside methods from EnergyMeter and DayCycleManager, helps us achieve the following requirements - FR_INTERACTION_CONTROLS (interact prompt), FR_SLEEP, FR_STUDY, FR_RECREATIONAL_ACTIVITY, FR_EAT, and FR_INTERACTION_TIME (advances time when activity is completed using addTime() from DayCycleManager).

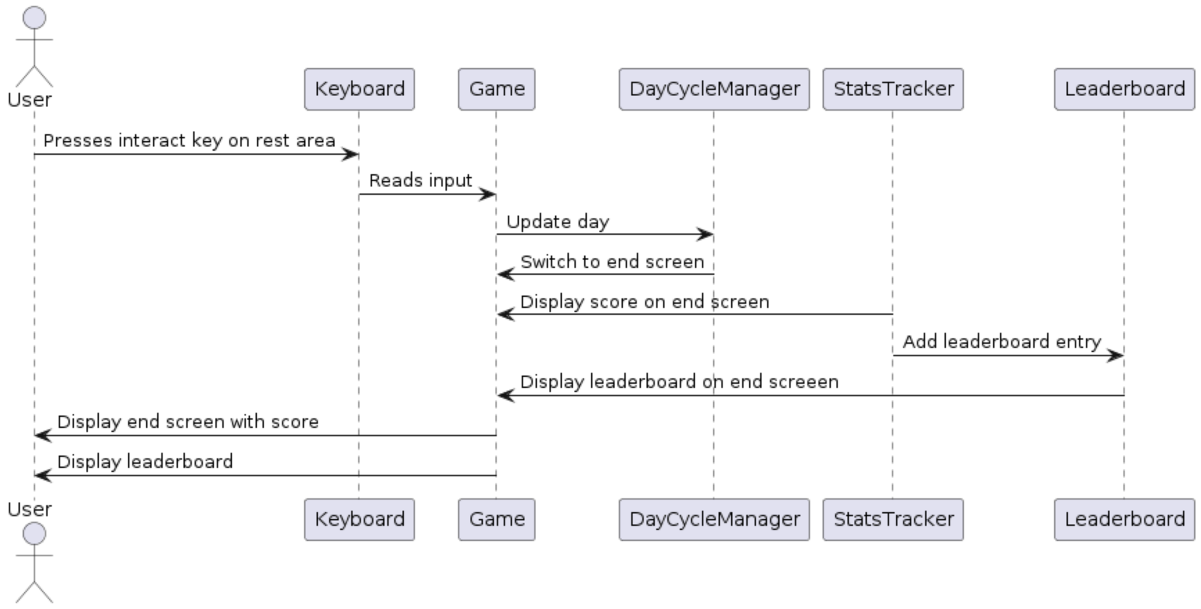
The MenuScreen, SettingScreen, EndScreen and inGameMenu all extend Screen and are very similar. They allow for the resolution of the game to be changed at any time, helping us to fulfil the requirement NFR_GRAPHICS. The MenuScreen provides a starting page, allowing the player to start a new game, satisfying requirements FR_START_PAGE and FR_NEW_GAME. The inGameMenu allows the player to pause the game, fulfilling the requirement FR_PAUSE. EndScreen displays a message to let the player know that they have completed the game and will also display a score in future implementations in order to satisfy requirement FR_END_PAGE. It gives the player the option to return to the MenuScreen to restart the game or to quit.

The CameraManager class contains methods to make sure that the Field of View is always centred on the player. While this does not achieve any of our requirements directly, it does help us achieve NFR_GRAPHICS and NFR_ANIMATIONS.

The game class is simply used to instantiate a game object. It contains a method that changes the type of screen that is being displayed (e.g. switching between displaying MenuScreen and GameScreen).

The StatsTracker class contains attributes which store the amount of in-game time the player has spent doing each activity, as well as methods to update and retrieve these values. These values are then used to calculate the player’s score at the end of the game. The class also contains methods for displaying these values to the character throughout their playthrough, satisfying the requirement FR_GRADE_TRACKER. The class also stores additional attributes and methods to provide a way to track the number of consecutive days the player has spent doing specific activities, so that achievements and point bonuses can be awarded to the player at the end of the game based on the type of streaks they kept. These satisfy requirements, FR_ACHIEVEMENTS ,UR_HIDDEN_ACHIEVEMENTS and FR_STREAK_BONUS.

The Leaderboard class is used to save and load player scores from different attempts of the game so that the highest user scores can be displayed at the end of the game. The Leaderboard class contains a method that returns a list of LeaderboardEntry objects. The LeaderboardEntry class was created to contain all of the attributes that would be required for each entry on the leaderboard. The EndScreen class can then use the Leaderboard and LeaderboardEntry classes to display the top scores at the end of the game, satisfying FR_LEADERBOARD and UR_VIEW_LEADERBOARD. Below, a sequence diagram was constructed to demonstrate an outline of the interaction with the Leaderboard class.


User
Keyboard
Game
DayCycleManager
StatsTracker
Leaderboard
Presses interact key on rest area
Reads input
Update day
Switch to end screen
Display score on end screen
Add leaderboard entry
Display leaderboard on end screeen
Display end screen with score
Display leaderboard
User
Keyboard
Game
DayCycleManager
StatsTracker
Leaderboard