UNIVERSITY OF YORK

DEPARTMENT OF COMPUTER SCIENCE

Architecture Design

Group 20

Formerly Group 16

Group Members:

Group 16

Charlotte MacDonald
Hollie Shackley
Luis Benito
Kaustav Das
Sam Hartley
Owen Gilmore

Group 20

Leuay Dahhane
Max Irvine
Sam Butler
Flynn Gadsden
Jacob Wharton
Billy Moore

Architecture Design Process

Our team used Responsibility-Driven Design (RDD) as the method to create the initial design of our system. It is specialised for object-oriented design which is how we have decided to implement the product. The aim of RDD is to maximise abstraction, distribute behaviour and provide flexibility [1]. The first step was to consider the product brief, interview with the client and requirements for a detailed description of the system. A designer story was developed to help us understand the key parts of the design. By underlining nouns in the product brief the main candidate objects were found based on the themes. With these we created CRC (Candidate, Responsibilities, Collaborators) cards, each with a small description of the concept and stereotypes. Next, from grouping the CRC cards it was clear some were unnecessary as they duplicated functionality so were removed. For instance, the Cell card was unnecessary as the player can move freely throughout the map so it doesn't need to be split into squares. Also, the GamePauser is not required as this can be done in GameScreen. Finally, individual responsibilities and collaborators were added to the cards. Collaborators are other cards that will need to be interacted with in order to meet responsibilities. These initial CRC cards with responsibilities and collaborators can be seen on the website [https://samh366.github.io/crc_cards.html].

Creating CRC cards is merely an initial estimate of what classes will be required to fulfil the product brief. When we were happy after looking through this a few times, we moved onto trying to map out these CRC cards to UML diagrams. At the outset we started with sketches drawn by hand as this allows for informal discussion where we don't have to focus on syntax and just lay out ideas. Then we moved to a tool called plantUML for formal UML diagrams from the sketches. A variety of diagrams were made to show the structure and behaviour of the system including class, sequence and state diagrams. Many iterations of each diagram were created throughout the project as new features and improvements were made.

UML Diagrams

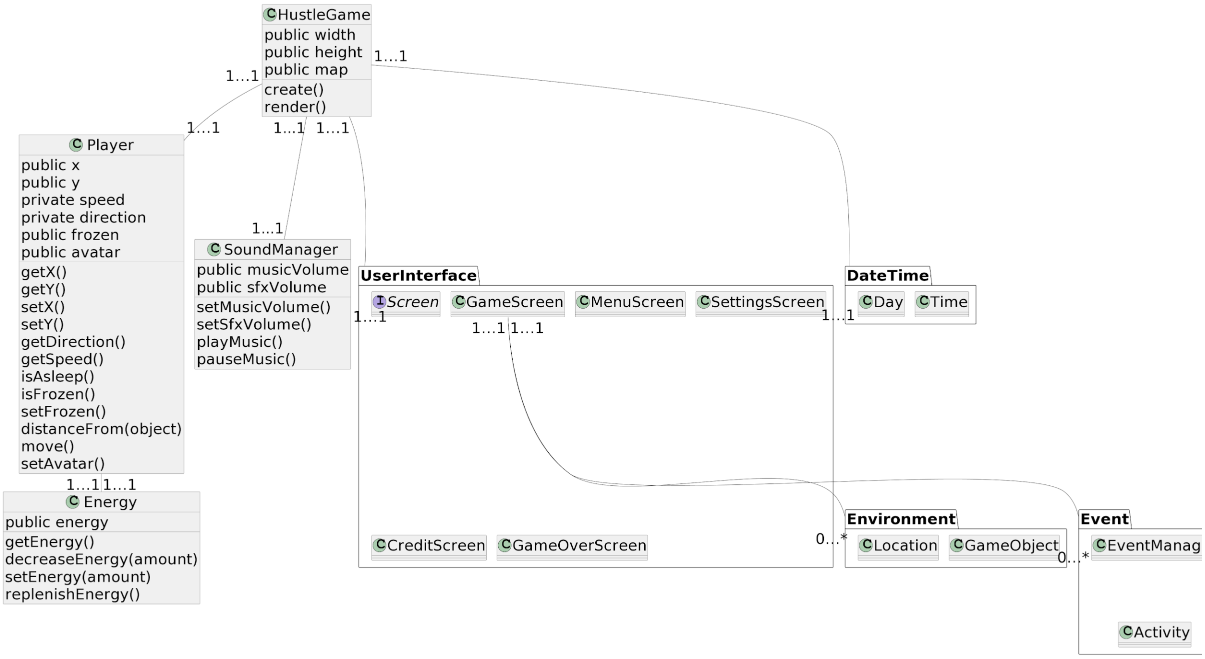
Tools used

To create the structural and behavioural diagrams needed to represent the system we used plantUML. One reason we selected it was because it can be used across multiple different types of platforms: in browser; embedded in a Google Document with the plantUML Gizmo extension and with IntelliJ IDEA's plugin by simply making a .puml file. As we are already using IntelliJ for the implementation it's an IDE the whole team should already have installed and is available on lab computers. The code is very human-readable and the documentation is well developed with lots of examples making it simple to learn and implement. One issue with PlantUML is that in the diagrams the arrows can go in sub-optimal routes which can overcomplicate them. The text was also often very small so to fix these issues we tried altering the arrow length and text size.

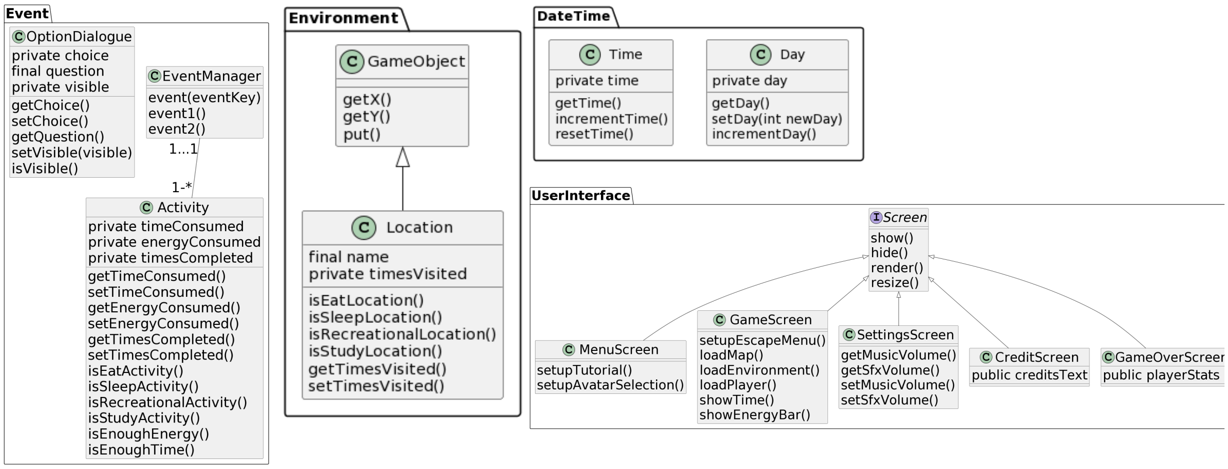
Structural Diagrams

Class diagram with packages for the whole system

When creating the initial class diagram [https://samh366.github.io/architecture.html] it was clear it would be very cluttered as there are many classes so we broke it down into packages where possible. The Screen package was for all screens used throughout the game (MenuScreen, GameScreen and SettingsScreen). The Event package was for coordinating and managing all the in-game events (EventManager, Activity and OptionDialogue). GameObject and Location are in the Environment package as they are to be placed throughout the map. DateTime is a package for the date and time as they are closely linked and rely on each other when it comes to incrementing the day. HustleGame, Player, Map and Energy didn't quite fit into packages so have been left alone.
For the next version [https://samh366.github.io/architecture.html], an interface called Screen was added as all screens had attributes/methods in common but no Screen instance will ever need to be created. This means all screen classes in this package will inherit from the Screen class. A SettingsScreen was also added as we realised a separate screen would be best for this rather than including it in the MenuScreen. The Screen package was changed to UserInterface so as not to confuse with the new interface also called Screen. The map class was removed as in the game it would be an asset rather than its own class. Relationships between classes/package classes were changed so Environment and Event now relate to GameScreen instead of HustleGame. This is because they are only needed and will be rendered/used on this screen.



Above is the final class diagram. A CreditScreen was added as this is now necessary, as well as setup screens as methods in MenuScreen for the tutorial and avatar selection which shouldn't need their own class. A GameOver screen was also added which implements the Screen interface. This displays final stats and has a button leading to the MenuScreen. Music and sound effects were not necessary but we had time to implement them and thought they would be a nice addition so a SoundManager class was created to control how sounds are used in the game. OptionDialogue was also renamed to DialogueBox as it was deemed a clearer name.



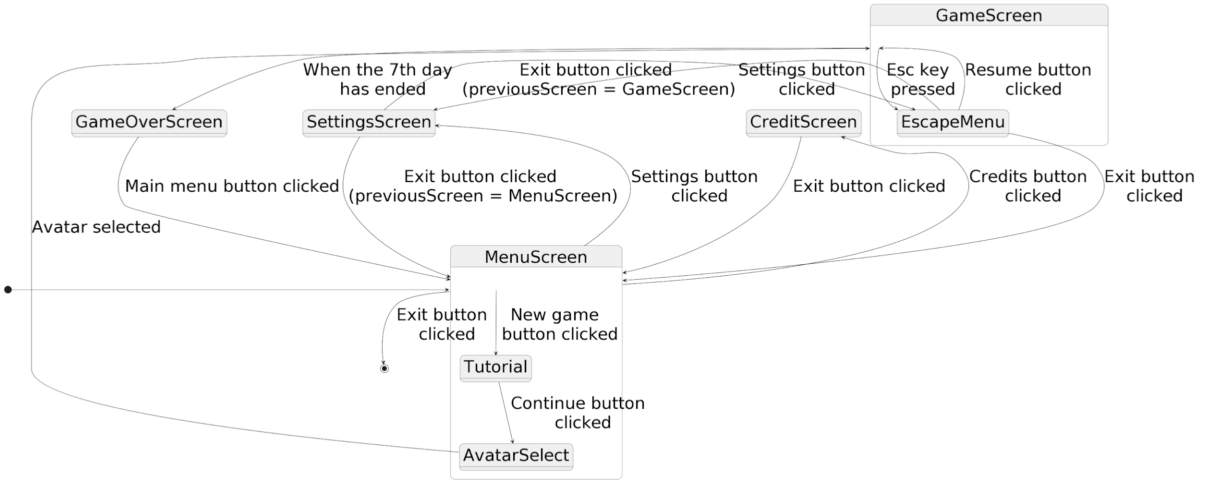
Above are the packages from the class diagram expanded. There is one event manager but only one instance. There can be many activities for the event manager to coordinate. In Environment, Location inherits from GameObject as it will use the same methods but needs more to track what type of location it is and how many times it has been visited. In UserInterface - MenuScreen GameScreen, SettingsScreen, CreditScreen and GameOverScreen all implement the Screen interface as this has methods all will use but will not be created.

Behavioural diagrams

State diagram for screens

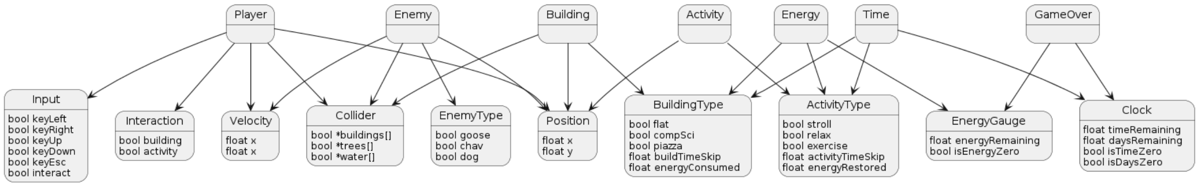
For the initial version of this state diagram [https://samh366.github.io/architecture.html], MenuScreen and GameScreen were the only screens. Within the MenuScreen it was necessary to have the ability to start a new game, access options and see credits. Two sub-screens had to be created to show the options and credits in a pop-up window. To get between these screens buttons were utilised. When on the GameScreen, by pressing the Esc key the Player can pause the game and a pop-up paused menu appears. From here the Player can resume or exit back to the menu. To completely exit the game there will be an “Exit” button on the MenuScreen.

For the second version [https://samh366.github.io/architecture.html], a separate SettingsScreen now replaces the Options pop-up in the MenuScreen as it needs to be accessible from both the MenuScreen and GameScreen. The previous screen will be kept so when exiting settings the Player will go back to the screen they came from.



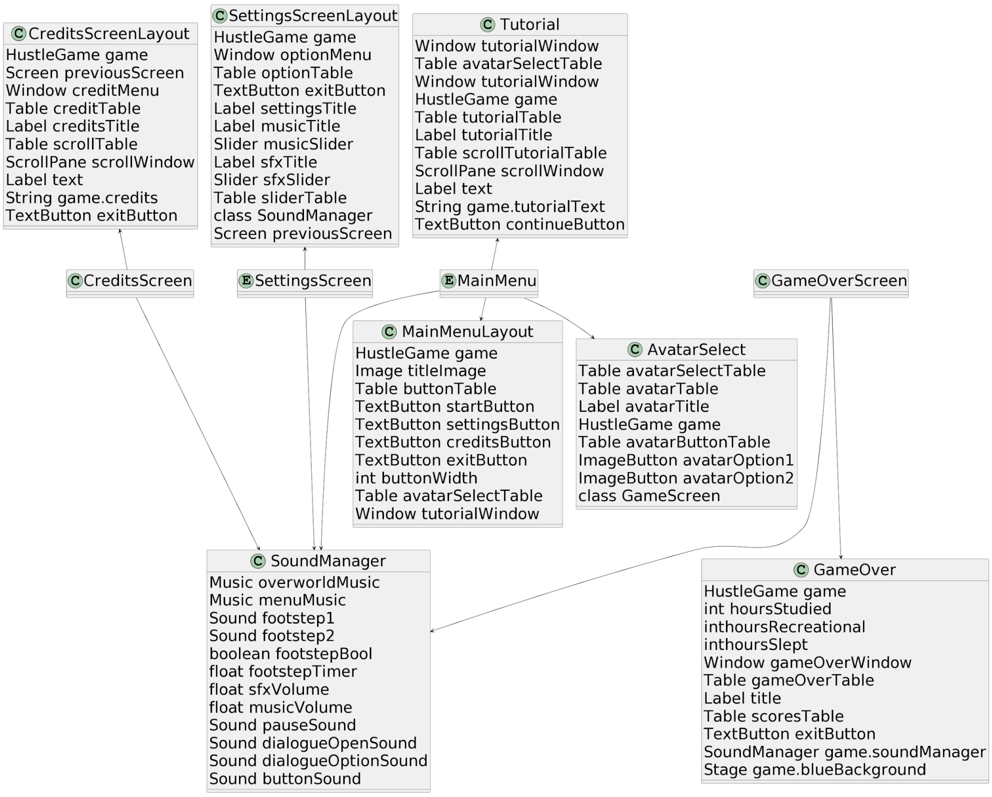
The above diagram is the final screen state diagram. A separate CreditScreen was added so each button on MenuScreen led to a new screen. However, when clicking “New game” you will be shown a short tutorial on how to play before selecting an avatar. Only after these two sub-screens will you go to the GameScreen. A GameOver screen is also added when the final day is up to display stats. Then it will take you back to the MenuScreen.

Component-Entity-System Diagram

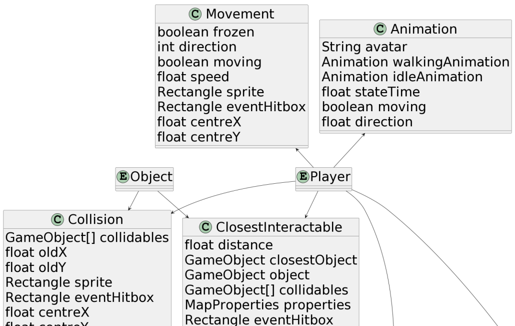


Above is the initial CES diagram created based on the product brief before the client. This was a very simplified approach to the game with only basic functionality. There are buildings which have activities which can only be completed if there is enough energy and time. The Player is able to move around the map based on Input and can collide with Buildings. The game is over when time is up. An Enemy was included in the initial diagram to provide more difficulty for the game.

The final CES diagram was too large so it was broken down into the stages of the game. Below is the diagram of the Menu stage, Option stage, Credits stage and Game Over stage.

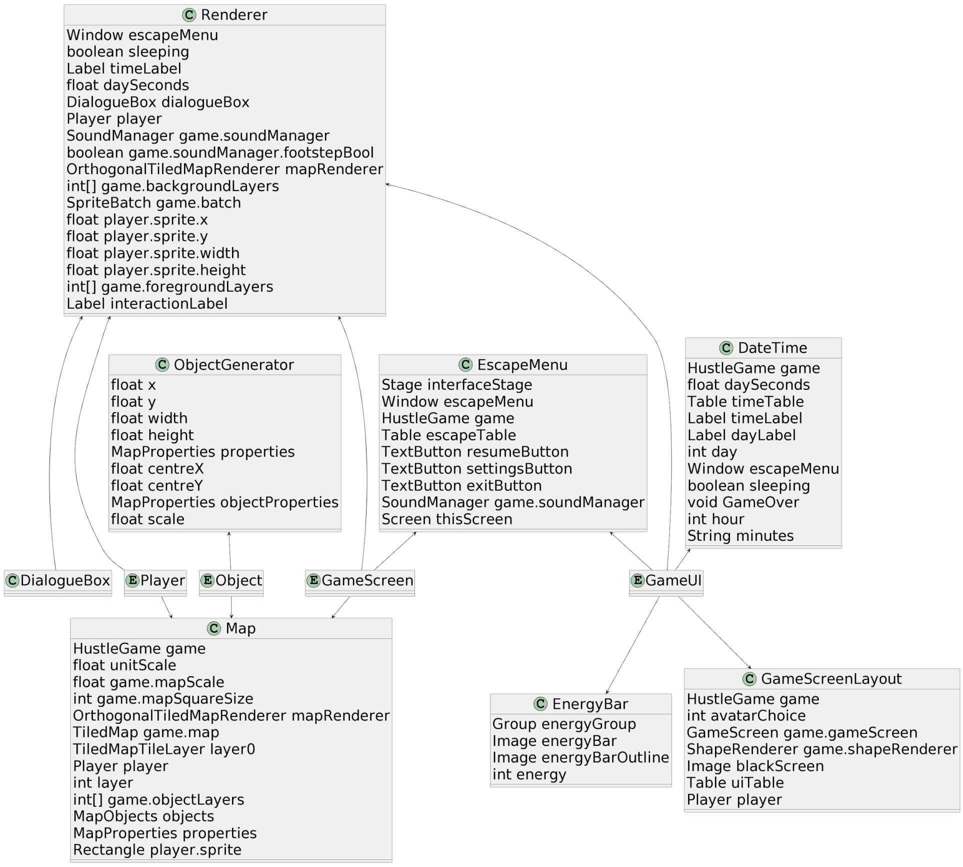


No screens were included in the initial CES diagram so they had to be added. These are all the non-game screens that allow the player to start the game, change music and sfx volume, see credits and see their final score. All use a layout and rely on the SoundManager. MainMenu uses AvatarSelect as there is a pop-up screen for the Player to select an Avatar and this selection must be stored.



Here is the diagram about Player-Object interaction. After the interview, the client specified no enemies were necessary at this stage so they were removed. The Player is able to move and each Avatar has an Animation. The Player uses SoundManager when it steps. InputAdapter allows the Player to react to arrow key presses (for moving the player) and other key presses for interactions. Both the Object and Player are able to Collide with each other making the game more natural.

Below is the sub-diagram for rendering the GameScreen.



The Renderer is used by all entities as it is responsible for making assets appear on the screen. The GameScreen uses the Map to make the background for the game.
ObjectGenerator is used by Object to ensure all relevant objects appear on the map. The EscapeMenu is used by GameScreen as a pop-up that appears when the Player presses the Esc key. This will allow them to pause the game, see settings and quit.
EnergyBar and DateTime are used by the GameUI to display the Player's energy level and the current day and time on the screen. GameScreenLayout, like with the other screens above, is used by GameScreen to format the screen.
There is also another CES diagram to expand on events and event management which can be seen on the website [https://samh366.github.io/architecture.html].

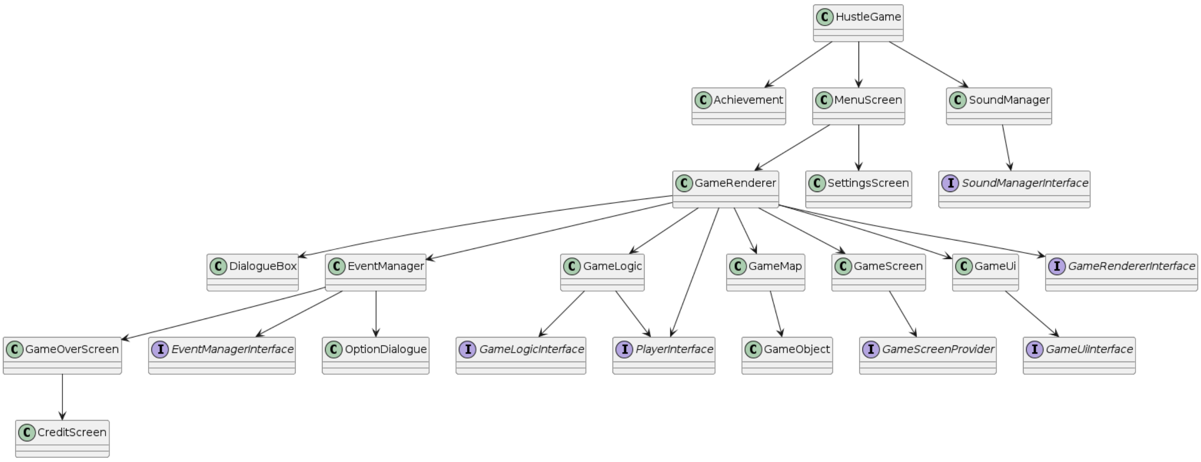
Brownfield project- Team 20

When taking up this project, one of our primary concerns was the architecture. From reviewing the code we could see that some classes were overloaded and took on a range of responsibilities. Let’s take the gamescreen class for example, this class handles rendering , pausing the game , manages game states, transitions and various ui components like the date time and energy - this means that this one class is simultaneously handling rendering, game state management and even UI interactions. We can see a more extreme example in the Hustlegame class which not only solely handles the coordination of many key items(player , map and energy) but also manages the entire game’s logic. This violates a very important principle called the Single Responsibility Principle which is concerned with mapping one responsibility to one class. This is extremely important as one small update can affect many aspects of the code , making it weak and fragile.

As a group we understood sorting out our classes and breaking them down was key before we began scaling the project. The code being in this cluttered state meant that reading , maintaining, scaling and finally testing the code would prove a much bigger challenge than it should be if we didn't correct this first. Classes are larger with jumbled responsibilities meaning tests are less focused and have to be broader since we can't effectively isolate the functionalities we are testing.

Our steps to fixing the code began by doing a deep analysis, we identified which responsibIlities we could group - based on similar concerts and logic. Once we had our groups sorted , we created classes which focused on handling one responsibility each. The transition process caused us some difficulty , however the modularity and testability of the code was well worth the challenge. We implemented tests now that we had greater modularity and the ability to isolate key features to check the integrity of our code and continued our architecture optimisation.

Now that we were satisfied with our class responsibilities, we wanted to decrease our dependencies. We achieved this by using interfaces. Interfaces allowed us to define methods and properties without implementing them. This enabled us to achieve polymorphism and decoupling which are both extremely useful for brownfield projects since we needed to focus on interchangeable components and modularity in order to carry out and maintain strong testing and scalability.



^Final Uml class diagram with interface additions

- EventManagerInterface & SoundManagerInterface: Define the methods needed for managing in-game events. This is very useful for using different management strategies .

- GameLogicInterface: this Interface specifies the methods for game logic. This separates game logic from other components, making the code easier to extend and maintain.

- PlayerInterface:Focuses on player related functionality.

- GameScreenProvider:This facilitates dynamic screen management and swapping without altering the core logic.

- GameUiInterface: concerned with ui management.

- GameRenderInterface: This interface is concerned with rendering logic, It lets us use different strategies without tampering with our game logic.

As previously mentioned, since these are isolated responsibilities testing has become significantly easier than it would have been if we stuck with the greenfield architecture.

Refactoring overloaded classes and incorporating interfaces significantly improved our game's architecture, making it more modular, maintainable, and testable. By doing this we have set up a cleaner code environment to work with, setting us up to begin implementation of our new requirements.

Relating Architecture to Requirements

User Requirements

ID	Architecture
UR-MENU	There is a MainMenu class with “New Game”, “Settings”, “Credits” and “Exit” buttons that navigate to different features. This is further shown in the screen state diagram above. [*]
UR-CUSTOMISE	There is an avatar pop-up menu after the game tutorial (within the MainMenu class) that will allow you to select between 2 different avatars.
UR-WORLD	The GameScreen renders the map, locations and GameObjects onto the screen.
UR-INTERACT	When the Player approaches a GameObject, interaction options appear as a DialogueBox.
UR-TIMED	The Time and Day classes keep a track of the time and day respectively. When the time gets to 24 hours the day in the Day class is incremented. When it reaches 7 the game ends.
UR-INFO	The Energy class stores the energy level of the Player and it is represented as a bar on the GameScreen.
UR-SOUND	The SoundManager class manages when sounds are made. It also controls the music volume and sfx volume separately.
UR-SETTINGS	The SettingsScreen class allows the user to change the music volume and sfx volume.
UR-SLEEP	When the Player does an Activity where isSleepActivity() returns true, energy levels are replenished back to full by calling replenishEnergy().

Functional System Requirements

ID	Architecture
FR-VIEW	The game uses topdown graphics and 3rd person sprites with arrow keys that allows the user to move North, East, South and West according to WASD and Arrow keys
FR-START	requires the player to be able to select between avatars which is fulfilled by the Avatar pop-up screen in the MenuScreen class.
FR-INTERACT1	Interaction initiates a pop-up screen inside the GameScreen which freezes the character movement until exited through choices or by pressing E
FR-INTERACT2	When a player starts to interact with a building, there shall be a pop-up with text and choices
FR-MENU1	In the MenuScreen class, TextButton(s) such as, “startButton”, “settingsButton”, “creditsButton” and “exitButton” allows for the creation of buttons that lead to their respective Screens once clicked.
FR-MENU2	No class for saving the game. This was an intentional choice.
FR-MENU3	While in GameScreen, Window escapeMenu allows the player to escape to MenuScreen by pressing Esc key followed by the exit button
FR-NAVIGATE	State diagram of player moving [https://samh366.github.io/architecture.html]
FR-SLEEP1	EventManager checks time of day before allowing activity. If 16 hours have passed all activities except sleeping are locked.
FR-SLEEP2	EventManager checks energy class to measure energy level. Disallows every other activity aside from sleep if energy level drops to 0.
FR-ENERGY1	Energy class and event
FR-ENERGY2	EventManager checks energy class for energy value

FR-WEEK	Day class, when on 7th day and time in Time class gets to 24 hours game will stop
FR-TIME	Activity class has amount of time it uses up which increases time in time class Dialogue allows
FR-GAME-PLAY1-4	to make decisions at location. Location has isSleepLocation() etc. to determine which is which.
FR-MENU4	MenuScreen has buttons allowing the player to select between multiple options
FR-COUNTER	each Location counts how many times visited, each Activity counts how many times completed

References

[1] R. Wirfs-Brock. (2006, Jul.). A Brief Tour of Responsibility-Driven Design [Online]. Available: https://wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf