Architecture

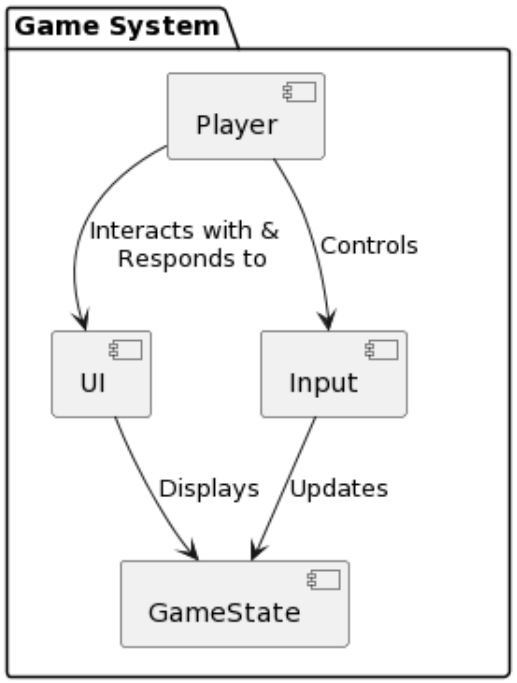
Group 9

Pluto Pioneers

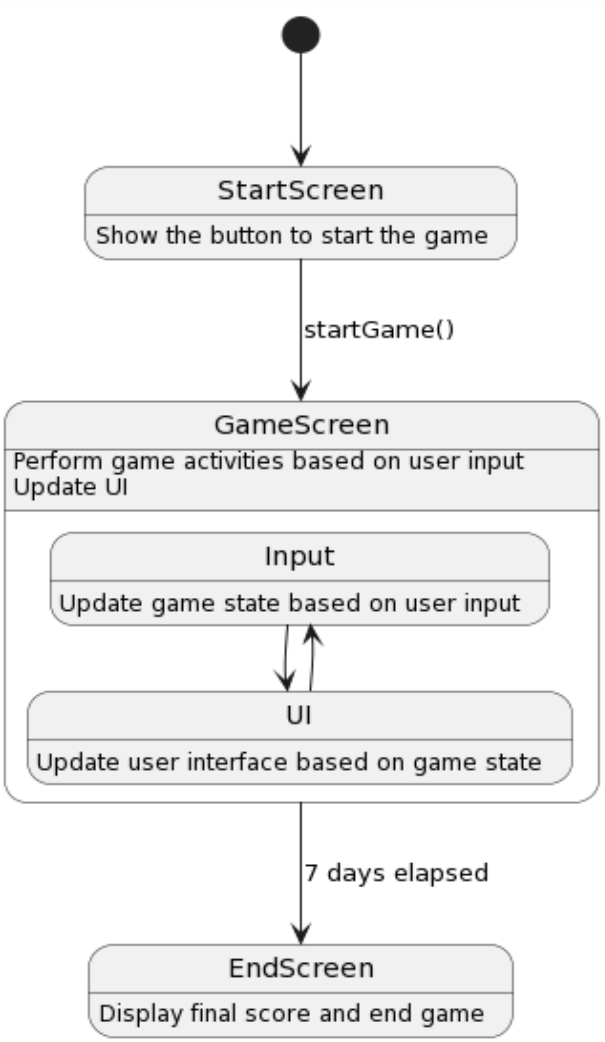
James Lawson, James Rooke, Dema Williams, Nicholas Barker,
Sam Slade, Joshku Gauntlett, Seiya Ikeda

The diagrams created provide useful information on how the program will work in certain areas which the programmers can refer back to throughout the process. It acts as a plan for what the project needs to do in a succinct useful format. The exact tool we used to make our UML diagrams was plantUML, we did this as it is a simple and efficient way to create the diagrams we needed. Using plantUML, instead of having to draw out all the objects and connections of our diagrams we could use concise textual descriptions to do so. The language for writing in plantUML was quick to learn and documentation for it was easy to find on the plantUML website. It also meant that making changes to the architecture diagrams would be quick and easy if we needed to change them down the line, this is because instead of having to open specific software for drawing and changing the shapes of the diagrams, we could just change a few lines in a text document to make the diagrams fit our new needs. plantUML is also useful because there was a plugin for IntelliJ, the IDE we are using which allowed us to convert UML code to diagrams very quickly and easily without needing any other software. As plantUML is text-based, it allows for easy version control.

Component Diagram



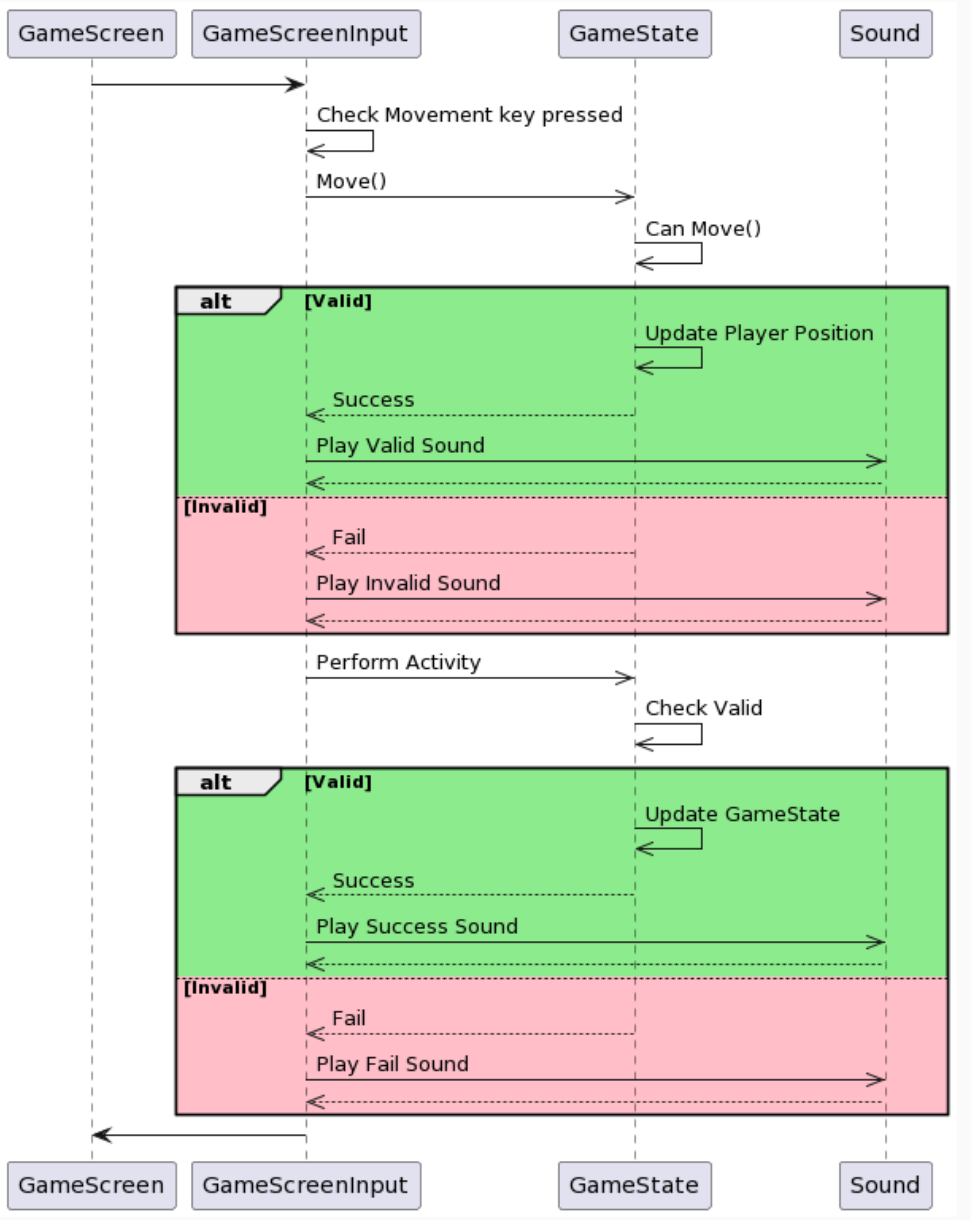
State Diagram



The component diagram shows how the player will interact with the game using the User Interface, which the user would be able to navigate, this user interface would display the current GameState, and the diagram shows how the Input would update the current GameState.

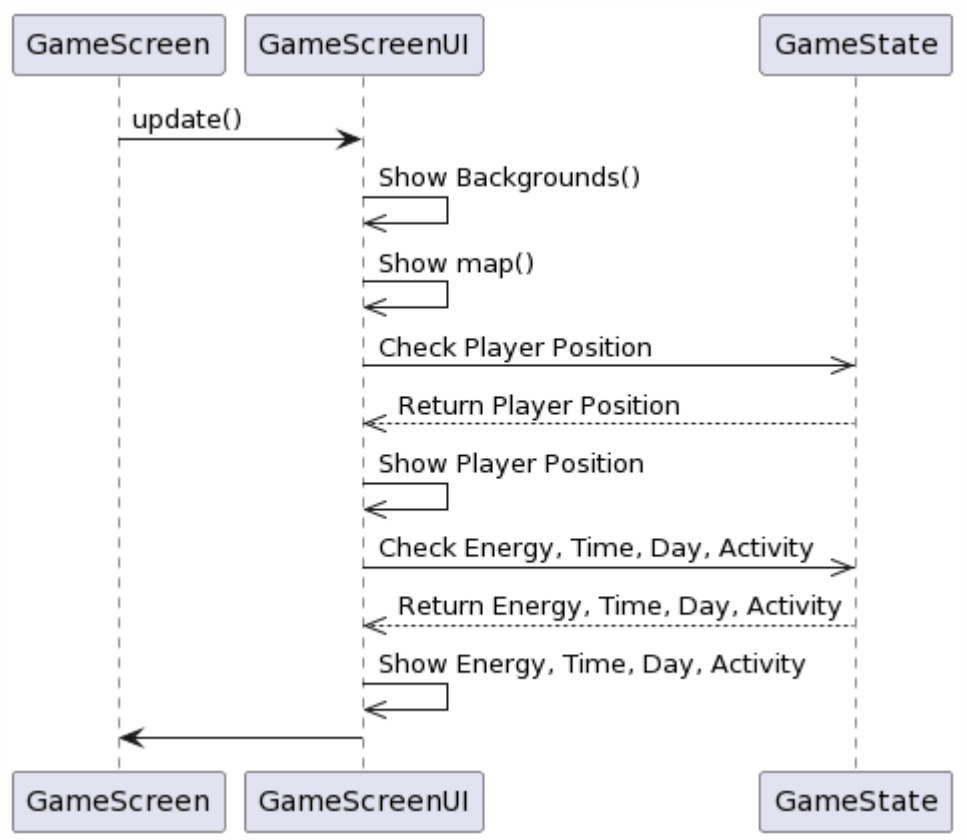
The State Diagram shows what state the game is in at different points of it running, and how it transfers from one state to another. The game opens in the StartScreen and when you actually begin the game, it transfers to the GameScreen state. Whilst in the GameScreen state, a constant feedback loop happens with the user's inputs updating the user Interface until they've played the game for the entire 7 days, at which point it transfers to the EndScreen state.

Sequence Diagram 1



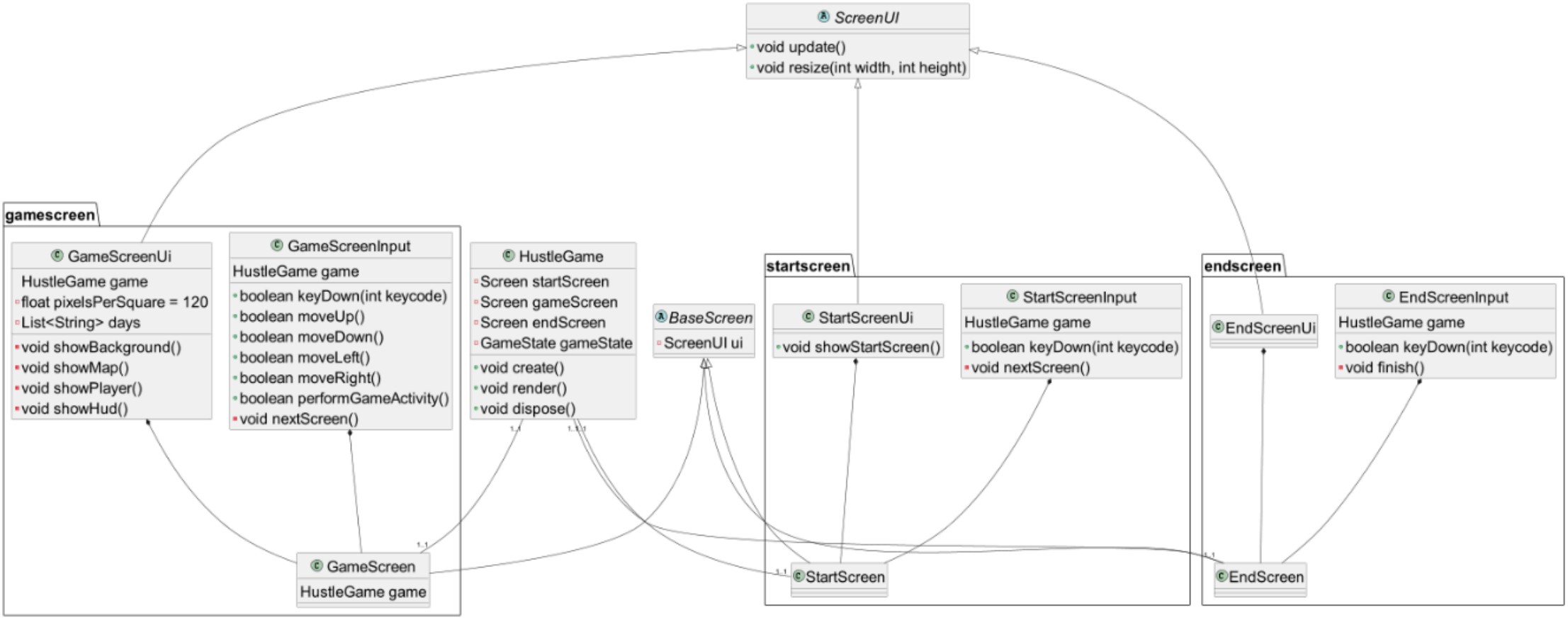
This diagram shows how the game handles player movement and the player performing activities with controls. Within Update(), the game continuously checks when the player tries to move or tries to perform an activity, whether or not it is possible to do so. Then, based on the result from this, it gives a valid outcome where it is allowed and an invalid outcome where it isn't, which have corresponding sound effects and results to them.

Sequence Diagram 2

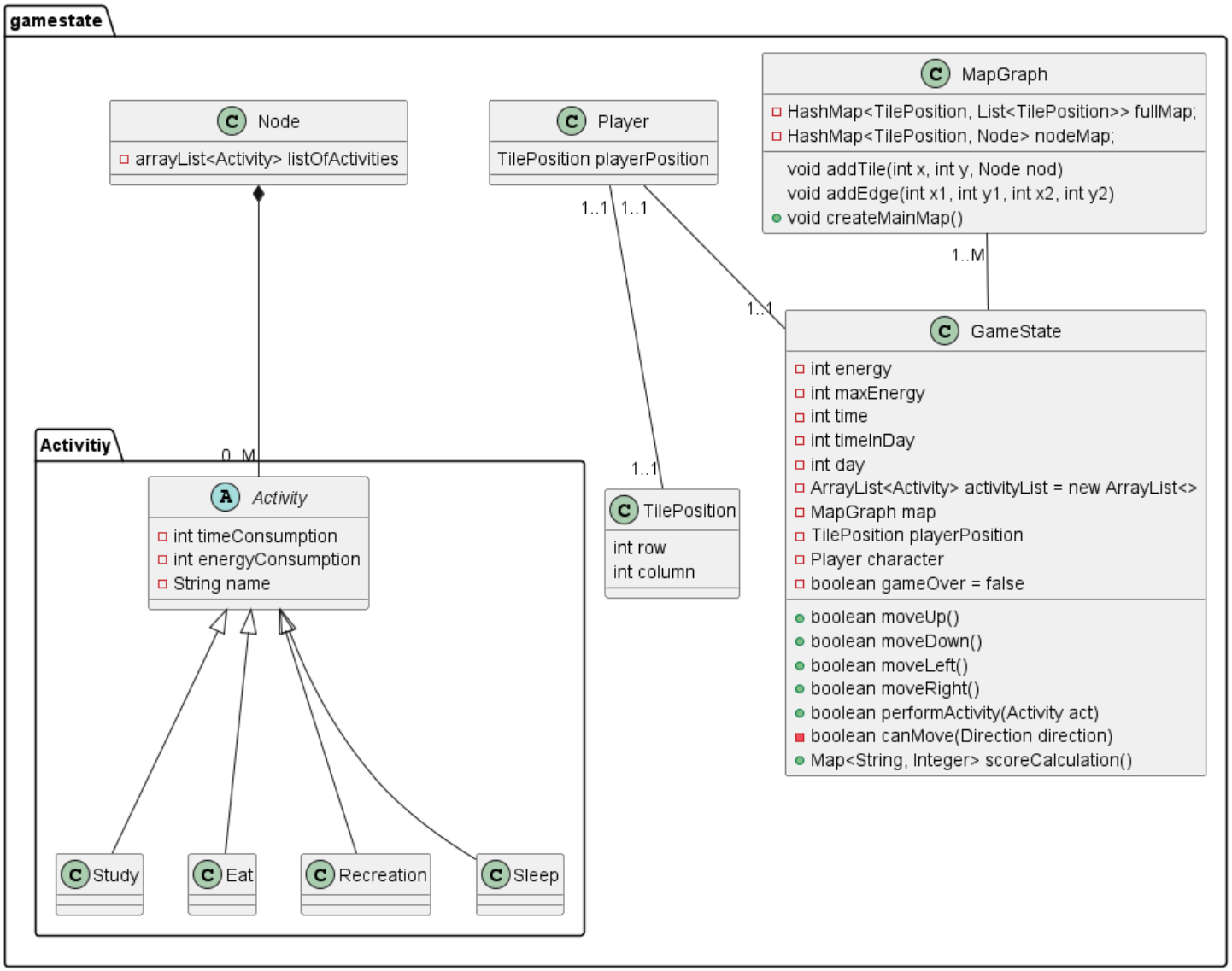


The first sequence diagram shows the update loop that will be running whilst the user plays the game. This involves displaying the correct aspects of the game like the background and map and making sure that all of the factors which need to be constantly updated are done so. For example, the player position, the Player Energy, the Time, The Day and the Activity.

Class Diagram 1



This UML class diagram shows the class structure for the front end of the game. Each of the startscreen, gamescreen and endscreen packages contain the specific UI and input classes, which controls what the user sees and how they interact with the map. The GameScreenInput class communicates with the backend of the program, allowing the user to move the player character around the map and complete activities.



This UML class diagram demonstrates the class structure of the backend of the game. Within the package of gamestate, the GameState class controls the logic of the game and how the player moves around the map. In the Activity package, the abstract class Activity is extended by the specific tasks that the user will perform at the various points on the map.

Architecture Approach and Justification

We decided to use a layered architecture approach. This was because it is a good choice for “small, simple applications” [1, p 139]. Therefore, this architectural approach is an appropriate choice as our project will be relatively small and does not require several major key components. Additionally, the author mentions how the layered architecture promotes “ease of development” [1,p139] if the project is relatively small in scale. As a result, this approach to architecture benefitted us during the implementation as it kept key areas of our game simple, allowing us to quickly implement a working game up to the standard outlined by our client. We had a two-layer approach - one for handling the game state and logic, and another for the GUI and input. This separation of concerns is crucial for ensuring the maintainability of the project, as the way the game is presented to the user can be modified without needing to consider how the game state is stored.

The architecture was designed through a series of group meetings. In these group meetings, we initially brainstormed ideas of how the game should be set out, in order to fully understand how we wanted to create the architecture we chose as mentioned above. This was initially done through the creation of Candidate-Responsibility-Collaborator cards [2]. Here we set out possible candidates within the system and then gave each candidate a stereotype and collaborator. This allowed us to gauge what each section would do which then helped us when creating the architecture.

After this, we then set out to create our initial structural and behavioural diagrams on a whiteboard. The initial structural diagrams we made were component [3] and class [4] diagrams. The component diagrams would help us gain a high-level view of the project. Whereas the class diagrams broke up our project into classes and components for the entities our game required. This diagram allowed us to gain a clear understanding of how the entire game would fit together and how we would structure the game when it came to finally implementing it. The initial behavioural diagrams we created were the state [5] and sequence diagrams [6]. The state diagram allowed us to gain an idea of what sections our game would be broken into. This then allowed us to identify certain actions each state needed to perform within the initial sequence diagrams we created. We then converted these into UML diagrams, as seen above, and continuously evaluated our diagrams when changes were made so that the UML diagram accurately represented our final ideas.

This architecture helps us to complete the requirements. For example, the architecture allowed us to achieve FR_character_control as this is handled by the Input component which then modifies the GameState entity.
Also, requirement FR_energy_allocation is achieved because the energy and time components will decrease after each activity. Then the GameState will not be able to perform any more activities if there is not enough remaining energy or time.
Additionally, FR_energy_costs is achieved because when creating each activity you can specify how much energy/time is consumed independently.

[1] Fundamentals of Software Architecture : An Engineering Approach Mark Richards, and Neal Ford
[2]https://docs.google.com/presentation/d/1lePP-5l3nA6M7EPCqWfCe3Edno_TcgMMu8U3qWkjfKs/edit?usp=sharing
[3]https://eng1-group-9-2024.github.io/Heslington-Hustle-Info/COMPONENT-DIAGRAM-WB.html
[4]https://eng1-group-9-2024.github.io/Heslington-Hustle-Info/CLASS-DIAGRAM-WB.html
[5]https://eng1-group-9-2024.github.io/Heslington-Hustle-Info/STATE-DIAGRAM-WB.html
[6]https://eng1-group-9-2024.github.io/Heslington-Hustle-Info/SEQUENCE-DIAGRAM-WB.html