

Architecture

Cohort 1, Group 5

Team Name:

JAZZ MoLeS

Group Members:

Sophia Taylor, Lucy Wood, Mitchell Gilbert

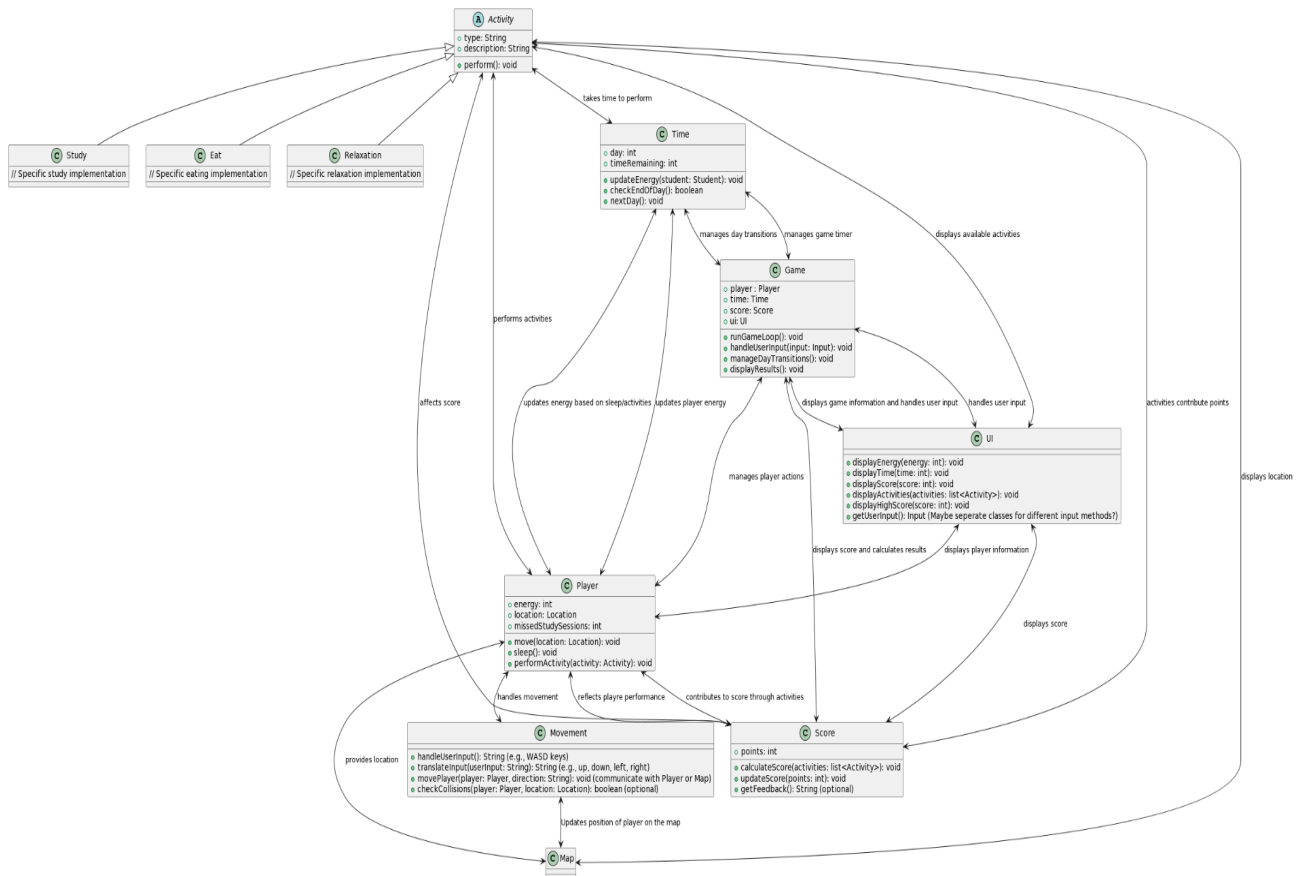
Jamie Creed, Archie Adams, Zayed Iqbal

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.

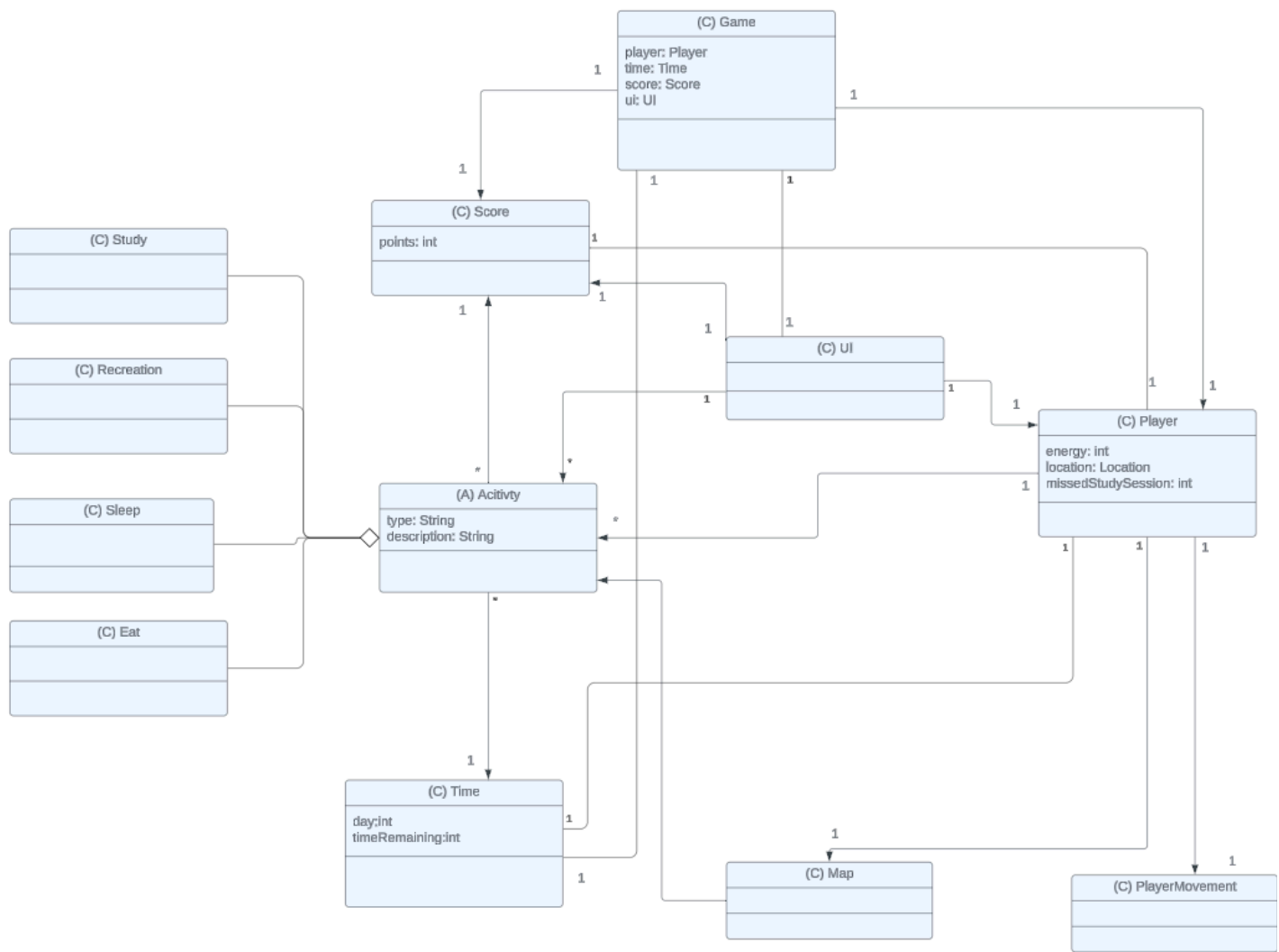
The UML diagrams below illustrates the relationship between components:

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.

[\[DIAGRAM\]](#)



[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.

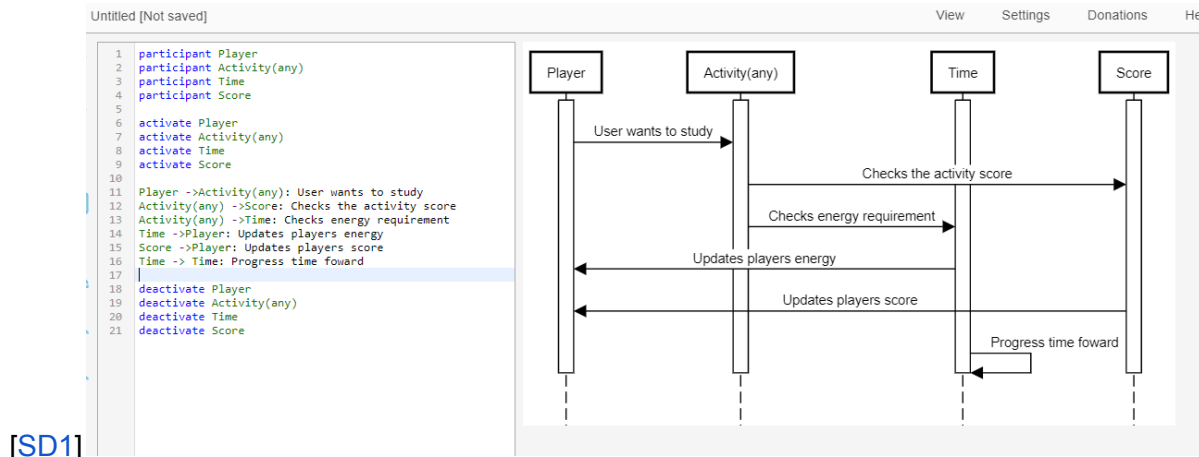
This UML diagram will serve as guidance for the developers to follow when creating the game. However, due to the complex nature of programming, this diagram may not be accurate post development. Developers may find more efficient solutions to implement certain features, or add/remove any additional or 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.

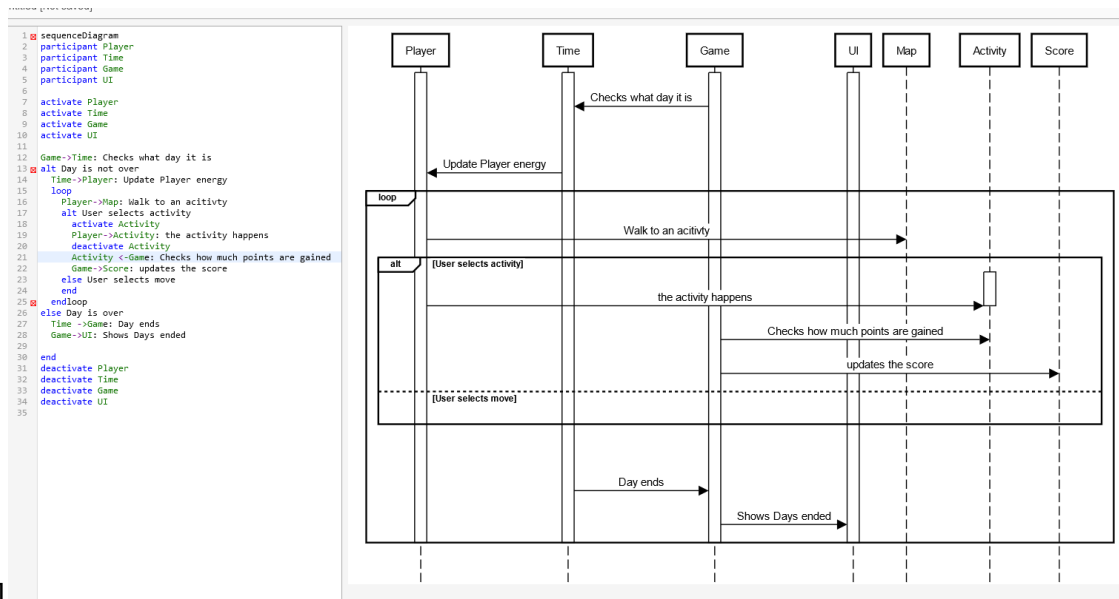
In order to visualise the order in which events occur, we created sequence diagram using sequencediagram.org



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*.

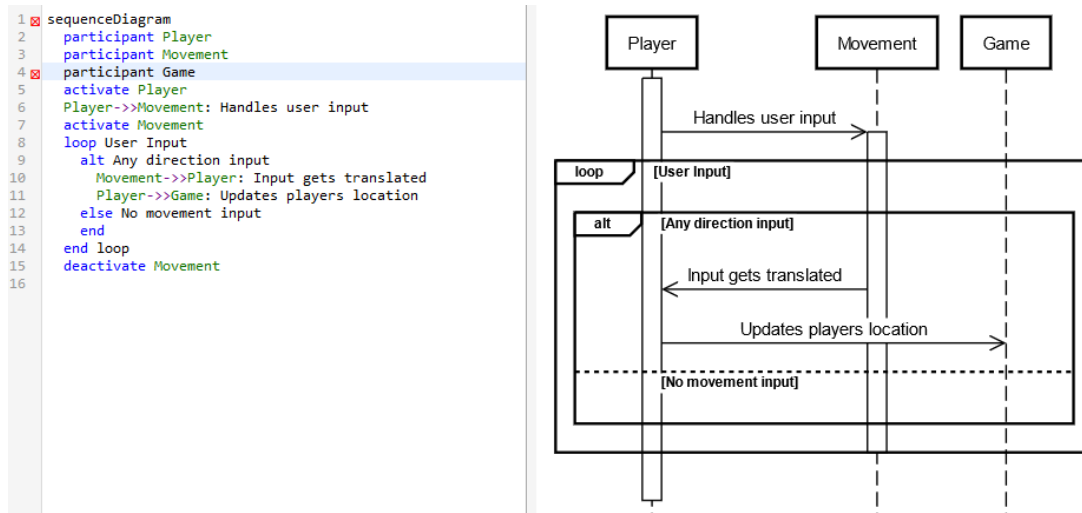


[SD2]

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.



[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.



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 don't 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.