D3 Architecture

Primary: Hamza Ahmad, Leo Durnion
Secondary: Wasif Mustafa, Samuel Knight
Tertiary: Anexa Bijoy, Emma Fuller

D3a - Statement on Tools & Languages

This section outlines the architecture and design approach used to develop the university escape maze game *QUACK?*, a Java-based game built using the LibGDX framework. It summarises the project's architectural style, tools used, and design principles supporting development.
The architecture follows a layered, modular object-oriented structure, separating the game logic and user interaction. Each class represents a logical component, for example player or energy, encouraging high cohesion and low coupling.

Tools and languages used

Language and framework: UML was used to make the class and behavioural diagrams. For the class diagrams we used structural elements such as nodes that contained attributes and methods. For the behavioural diagram we used, lifelines, messages and guards.

PlantUML was used to create these diagrams due to its in-depth documentation and simplicity in its syntax which allowed for efficient development of these diagrams.

Justification

The architecture ensures strong separation of concerns, such as logic and interface. Separating the various classes within the product brief allows for a more organised and efficient approach to development. We planned the start of our implementation based on splitting up the classes between our implementers so the class diagrams were vital to our organisation. By clearly dividing responsibilities, each team member can focus on specific classes, ensuring that they are completed to a high standard. Additionally, allowing team members to choose the classes they feel most confident or experienced in ensures better quality work, increases motivation, and leverages individual strengths. It also allows us to work in parallel as we can use our time more efficiently.

The behavioural diagrams allowed us all to have a thorough understanding on how the game would flow and how all of the elements would interact with each other. This diagram allowed us to effectively work out which parts needed to be implemented first so that delays due to dependencies are limited. It also clarifies the way that the game will flow so that the chances of us having to do extra coding due to miscommunications in how the game will run is limited.

Overall, our architectural goals were to promote modularity and allow for scalability for future development. This would allow more efficient debugging and testing and allow each module to develop independently.

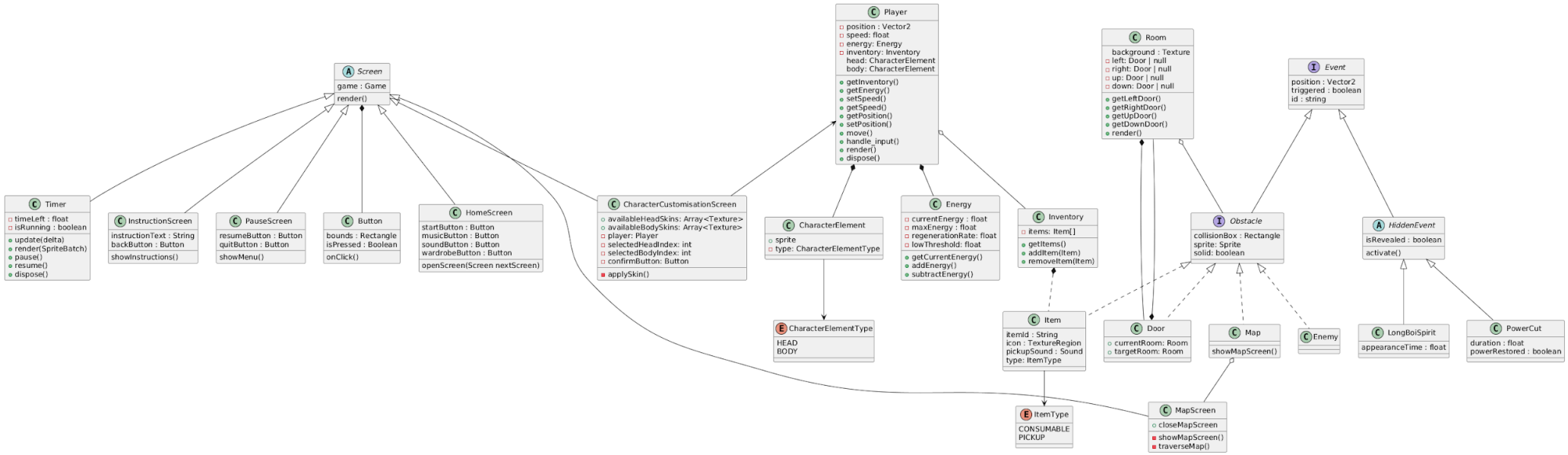
Development of architecture

The first iteration of the class diagram was based on the CRC cards that we completed collaboratively as a group. We then had a discussion about the ways in which we thought would be the most effective ways to organise our classes. With these we created a skeleton of the architecture using basic classes and relations. As we started implementing features we gained a better understanding of the game's direction and subsequently updated the diagrams to closely follow the game. We found that some of the classes were simply not needed as their features could be included more effectively in other classes. Also some classes were developed and expanded as our implementation progressed and our iterations of the class diagrams reflect those changes in class structure and mainly in the dependencies within our classes.

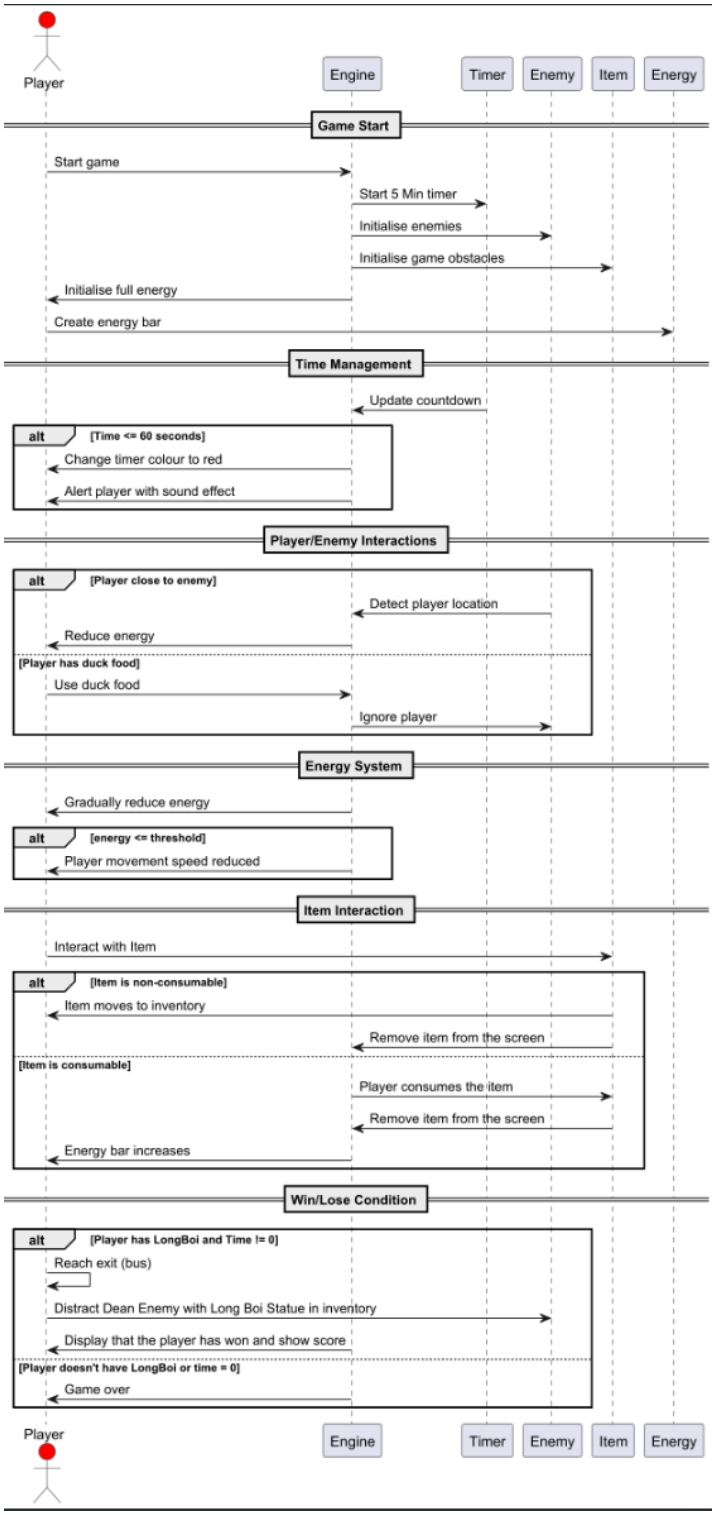
The first iteration of our behavioural diagram was made based on our group discussions about how we thought the game should run and how the different parts of the game would interact with each other. As implementation began, the behavioural diagrams became more complex and included more elements as we became clearer on some specific parts of the game. They were then split into smaller diagrams that work alongside the main diagram to provide even more detail that we needed at the later stages in implementation.

D3b - Structural UML Diagram

High resolution copy available [here]

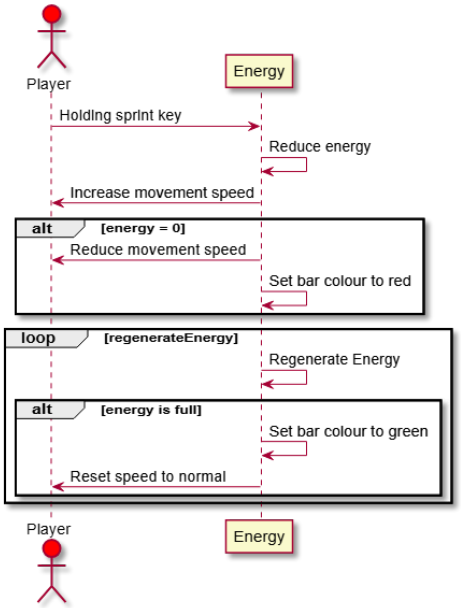


D3c - Behavioural UML Diagrams



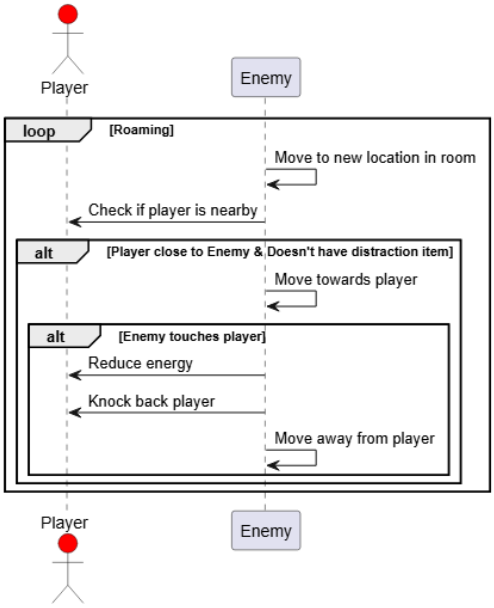
Energy

UR_ENERGY Requirement



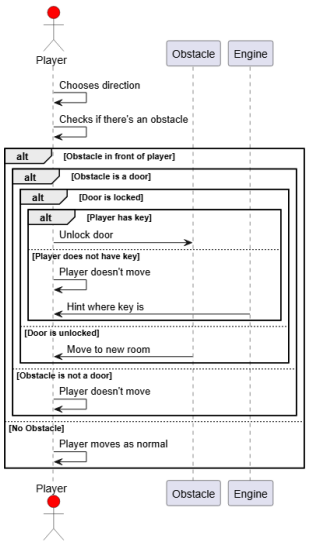
Enemies

Added as part of our events, for the UR_EVENTS Requirement. They move around randomly as part of fulfilling UR_EVENTS_LOCATION. Attacking players fulfils FR_NEGATIVE_EVENTS.



Movement and Obstacles

UR_MAP Requirement, with stationary screens to fulfil UR_DISPLAY. Movement fulfils UR_CONTROLS by keyboard controls.



Timer logic

FR_TIMER and FR_TIMER_END requirements.