Architecture

Developing architecture is essential for a game's success. Through architecture, we've crafted an environment that prioritises efficiency and adaptability. By visualising the structure and design principles, we gained insight into how individual components are intended to collaborate, manage player data throughout the system, and execute various key tasks, such as rendering and implementing game logic.

We chose to use draw.io to design our architecture diagrams, due to its simplicity and familiarity amongst the team. It has features specifically suited to the development of architecture diagrams such as built-in support for UML diagrams that we used to design the class structure of the program and support for hosting diagrams on google drive, reducing the friction of collaboration amongst team members. This also made it easy for team members implementing the game to refer back to the architecture plans as they went.

The preliminary plan for designing our game was centred around a simple framework which outlined the organisation and collaboration of our classes [see Figure 1 Heslingtonhustle-Initial draft.png]. This was built after some initial brainstorming and research on how LibGDX libraries worked together.

Using the LibGDX library meant that all game components are contained within a core folder. It is supplemented by a desktop folder that manages platform-specific requirements. Fortunately for us, our meeting with our customer determined that only desktop compatibility was necessary.

As we already proposed our requirements, we built our rough architect diagram with NFR_Delay in mind. This non functional requirement focuses on ensuring the map is loaded within reasonable time and the gameplay isn't affected significantly by lag. In order to do this we introduced a class which solely renders our game and refrains from all other tasks such as managing user input and game logic. By structuring our classes this way we avoid lag by having effective resource management and an optimised rendering pipeline.

Our two initial engine classes are responsible for managing all user inputs and also facilitating all actions between the student and interactable sprites- (computer science building, sports gym etc). This approach takes into consideration our NFR_Scalability, during our meeting with the customer, it became apparent that implementing multiple sprites would be essential for assessment 2. To ensure that others can easily incorporate this feature into our game, we've structured the engines accordingly. This approach significantly reduces the workload compared to having a single main game engine, making it more straightforward for others to

integrate additional sprites into the game, adhering to NFR_Scalability. Additionally, the engine classes handle interactable instances and the user , this gives us a presentable code which has a decluttered MainGameScreen class. A byproduct of this is that we streamline the communication exclusively between these two engines further optimising our game and fitting the criteria for NFR_Delay.

Based on our rough framework we began our first step in formalising and created collaboration cards for classes [see Figure 2 HeslingtonHustleCollabarationCards.png].These cards delegate and group responsibilities for our classes.They are useful at this stage as we can visualise development while keeping ambiguity.

To align with our fit criteria for NFR_Delay we need a strategy to reduce lag and delay between frame processing .The approach we selected was every interactable object instance and the user/player independently loads their sprites. They are then loaded into sprite batches which are integrated into their own belonging engine class,then loaded right into the MainGameScreen class.By using batches we significantly reduce overheads as we are rendering in groups instead of at one at a time, streamlining the process and smoothening our gameplay - NFR_Smooth_Gameplay.

Following our modularity ,the user requirement UR_student_Move, each Player class encapsulates its own position on the screen and possesses a function to facilitate movement. However, it's important to note that the actual movement execution is delegated to the PlayerEngine class.This separates responsibilities while reinforcing NFR_SCALABILITY ensuring the fit criteria is met.

Following our scalability optimisations each type of building (computer science lab, sports gym, etc) will inherit from the GameMapObject class. These classes will manage their own interaction timing and determine the statistics gained from each interaction, such as energy, study points and leisure points. This design choice allows for the seamless creation of new building types as needed, such as communal cafes or libraries. This again adheres to NFR_scalability and can be a good showing point for our presentation ahead of assessment 2. This approach also enables each building to deny interaction based on the user's current statistics. Our functional requirement FR_Invalid_student_interact requires certain interactions to be invalid , an example of where this would be deployed is where a student has maximum energy and is trying to sleep, therefore his action is invalid and he can't carry it out.

In order to understand the systems functionality we formalised our brief and created a use case diagram to aid system design and provide a general guidance that's easy to understand.[see Figure 3 HeslingtonHustleUseCaseDiagram.png] presents us with the different interactable objects and how we intend the students to interact with

them. For example a student will go to the library when they need to increase their study score to reach the quota.Since assessment one requires us to strictly implement one of each building type we have curated a “piazzeria”,”sports centre”,”ron shook”, “computer science building”, and a student house.In order to avoid risks relating to copyright and privacy compliance issues we renamed these buildings while keeping them close to their original so that they are relevant to our brief. In our diagram the student interacts with each building to gain stats which contribute to beating the game. These are : paizzeria-increases study score, Student’s home-increases energy, Sports centre-increases leisure points, Ron shook-increases leisure and a computer science building-increases study score. If the student's energy is fully depleted, they fail to meet the pass quota, or if they run out of leisure points, they fail the game. This adherence to the UR_Lose_Game requirement ensures a challenging and balanced gameplay experience.

A behavioural diagram shows how a system functions and communicates with other systems and people. They facilitate a better knowledge of the system's operation by assisting stakeholders and developers in understanding how events and data flows through the system. Sequence diagrams are helpful for representing user interaction and scenarios in addition to being a visual aid. They assist in defining user needs and validating system performance by demonstrating how users use the system to accomplish particular tasks or objectives.[Figure 4 HeslingtonHustleSequenceDiagram.png] does exactly that. In the diagram we can see how the game handles all data and the general flow of the system.

Through the construction of these diagrams we can now create a complete UML diagram . Creating the UML class diagram is a key part of architectural design as this ensures the developers and any other people that join the coding process have a common language to discuss and understand the games architecture. This ensures clarity and consistency in the design process.They provide a visual representation of the game's architecture making it easier to understand the relationships between different components, such as map, player and events classes. For example, a player can choose multiple avatars, so would be represented in a (1….*) relationship. Also showing the structure of the architecture like this helps identify potential design flaws and could potentially optimise the game's performance by providing a clear overview of how different elements interact with each other.With our methodology each class is now responsible for a single aspect of functionality. This ensures the code's readability, maintainability, and flexibility, as it makes the code more modular and easier to understand. Smaller, more manageable classes also promote reusability as it creates building blocks that can be combined and extended to meet different requirements including future projects or future progress along the project. [See Figure 5 HeslingtonHustleClassdiagram1.png] contains modular classes.

When we completed our implementation we began testing fit criterias. Unfortunately, one crucial user requirement (UR_Scoring_System) was completely overlooked within the development process.To ensure clear communication to our developer team we created an updated revision of our UML class diagram [See Figure 6 HeslingtonHustleClassdiagram2.png].Once this was created it was clear where and how this would have to be implemented .Our implementation team used the diagrams visualisation to create this class with ease, Without this formalised representation, I anticipate that the process would have been considerably more time-consuming.
Once we had refined, revised, and validated our UML diagrams, we transitioned into the implementation phase of our software development process. As we developed the code we ensured our UML continued to stay up to date and were open to correct other omissions and errors should they have arisen.

We maintained an emphasis on successful version control, building on the existing code and further refinement on the project will be a streamlined process, completely necessary in our case with other individuals continuing on from our development

Version control is vital in a game project like this as they enable team members to visualise and understand the system's architecture at different stages of development. By comparing different versions, team members can also understand how the architecture has evolved, what design decisions were made, and why certain changes were implemented. This documentation is invaluable for maintaining a detailed record of the project's development history. Different versions of class diagrams could also result in insights into implementation strategies, helping ensure that all stakeholders have a shared understanding of the system's structure and behaviour.

To conclude, our architectural approach allowed us to build a robust project, we were able to visualise, collaborate and control our project; leading us to develop a high quality enjoyable game for our customer. We will continue to improve upon our approach going forward.

Figures

Follow the links to view the diagrams.

Figure 1

Figure 1 Heslingtonhustle-Initial draft.png

Figure 2

Figure 2 HeslingtonHustleCollabarationCards.png

Figure 3

Figure 3 HeslingtonHustleUseCaseDiagram.png

Figure 4

Figure 4 HeslingtonHustleSequenceDiagram.png

Figure 5

Figure 5 HeslingtonHustleClassdiagram1.png

Figure 6

Figure 6 HeslingtonHustleClassdiagram2.png