ARCHITECTURE

Group 3

Liam Martin

Aaliya Williams

Lucy Crabtree

Kai Nichol

Sammy Hori

Tim Gorst

Zac Ribbins

Introduction to the Architecture

After extensively outlining the requirements of our game, our team shifted focus towards defining the architecture, ultimately settling on an object-oriented approach. To facilitate the development of a robust system structure, we opted for Unified Modeling Language (UML). This strategic decision played a crucial role in laying the groundwork for our game system by enabling us to:

1. Create detailed models and visual representations of individual system components, including state and sequence diagrams. These visualisations provided clarity and structure to our architecture, facilitating the understanding and communication of complex system relationships.
2. Establish a solid starting point for our system, providing a framework for iterative development and adaptation during implementation. This iterative approach ensures adaptability to evolving project requirements and facilitates efficient development.
3. Identify potential areas for code reuse and anticipate potential challenges or complexities within the system. By pinpointing these components early on, we could streamline development efforts and enhance code maintainability and scalability.

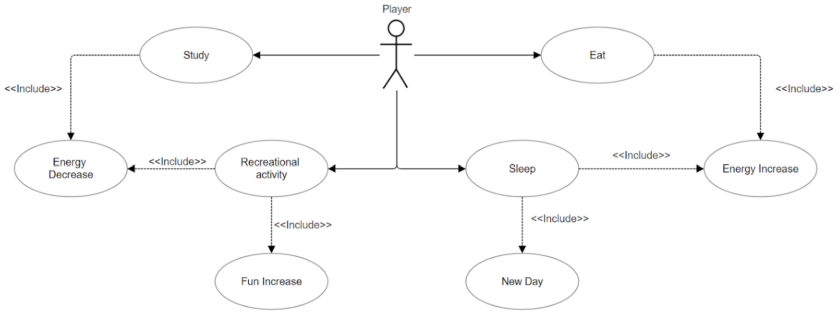
We used LucidChart [1] to create our UML diagrams due to its extensive library of pre-existing templates and libraries, which simplified the creation of comprehensive and visually engaging representations of our system architecture.

Behavioural Diagrams

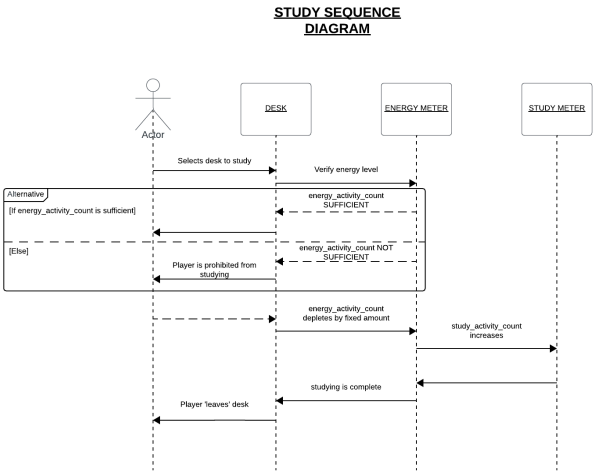
To gauge an idea of how our game will behave, we created a variety of behavioural diagrams, consisting of a ‘Use-Case Diagram’, ‘State Diagrams’ and ‘Sequence Diagrams’.

The state diagram, [State Diagram - Bedroom.png], demonstrates the state the system is in when the player enters their bedroom and is met with the choice of selecting a few options. It serves as a sequential representation of how the system proceeds after it receives each individual input from the user.

The use-case diagram, [Use Case Diagram.png], illustrates two particular paths the player can take once they've entered their student accommodation. The player has the option to 'Study' or 'Eat' in which both choices have a direct impact on the status of the player, both choices affecting the 'Energy' of the player in opposing ways.



A more in-depth illustration of this is displayed by the sequence diagram, [Sequence Diagram - Study.png], as it poses as a more explicit visual representation of the effects on the status of the player (actor). Each object has a direct impact on the other and this is shown by the 'Desk' object requesting verification on the 'Energy_Activity_Count' from the 'Energy Meter' object in order to determine the next course of action the system will take.



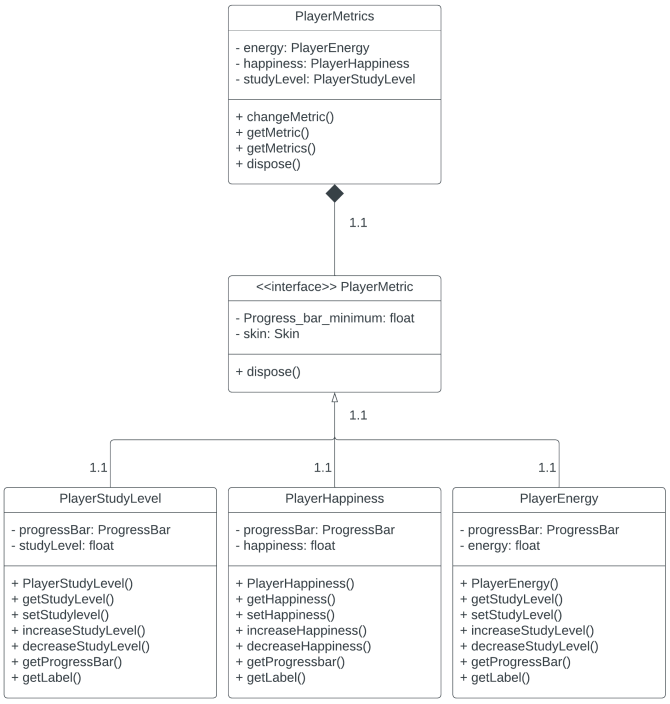
These behavioural diagrams served as a guiding framework for the subsequent implementation phase.

Structural Diagrams

Given our OOP approach for game implementation, we opted to develop visual representations using UML to serve as the foundation for our class structures. We used Class Responsibility Collaborator (CRC) Cards [CRC cards] and Class Diagrams.

With the aid of the Visual-Paradigm Tutorial [2] on UML class graphical notation, a Class Diagram [Class Diagram], was created highlighting each individual class, their attributes and methods, as well as their relationships with other classes/objects. From the diagram, it is clear that inheritance would play a vital role in the subsequent implementation of the game as well as composite relationships between a multitude of classes, given many cannot exist without the other.

When considering the structure for the status effects of the player, we opted to use the single interface PlayerMetric which is held within the parent class, PlayerMetrics. The PlayerStudyLevel, PlayerHappiness and PlayerEnergy subclasses implement methods from the interface for clean code and to ensure all the metrics can be used as expected.



Systematic Justification of the Architecture

When deciding on the architectural approach for our system, the inherent nature of the game guided us towards adopting Object-Oriented Programming (OOP) as the most suitable methodology. Several key factors influenced this decision:

- The structure of our game prominently features elements conducive to inheritance. For example, various status meter bars exhibit similar behaviours, differing primarily in the rate of increase/decrease. Adopting OOP facilitated the creation of a ‘PlayerMetric’ parent class, from which specific meters such as ‘PlayerEnergy’ could inherit, thereby reducing redundancy and promoting code reusability.
- The potential for encapsulation played a pivotal role in the choice of programming style. OOP provided a framework for encapsulating an object’s game state, simplifying state management and facilitating modifications. Encapsulating the game state within an activity tracker object would streamline the manager of player actions within the specific time frame, enhancing gameplay dynamics and user experience.

Evolution of the project

Initially, the game design allowed players to freely choose activities without considering their in-game needs. However, as we aimed to mirror real-life student experiences, we re-evaluated this approach. We identified time constraints and streamlined the activities into essential categories: EAT, ENTERTAIN, STUDY, SLEEP, NAP [CLASS]. To align with this structure, we implemented checks to ensure players' needs were met before allowing certain activities [CLASS]. Instead of automatically advancing to the next day at midnight, players could only sleep after 16 hours of activities [CLASS], highlighting time management. This evolution simplified the gameplay, ensuring it remained engaging while accommodating our project's time limitations.

Requirements → Architecture

The table below relates the architecture to our requirements:

Classes	**Requirement → Architecture Link**
Player	UR_MOVEMENT, UR_INTERACTION, FR_INPUT_DETECTION, NFR_RESPONSIVE -> Accurately detects and responds to user input for movement and interaction.
PlayerMetric	FR_ENERGY_ACTIVITY_COUNT, FR_HAPPINESS_ACTIVITY_COUNT, FR_STUDY_ACTIVITY_COUNT -> Adjusts the need level based on completed activities.
GameScreen	UR_ACTIVITY_CHOICE -> Users can select from multiple activities through interaction with buildings and items. FR_GAME_DURATION -> Controls the game duration, ensuring it runs for roughly 5 to 10 minutes. FR_SLEEP -> Manages the automatic simulation of sleep between 12am and 8am after 16 hours of activities. NFR_GAME_LENGTH -> Ensures the game is played within a reasonable time frame, with the average duration not exceeding 10 minutes. FR_UI, UR_UX -> Ensures that the user interface meets expectations in terms of clarity and usability.
EndScreen	UR_GAME_END -> The game concludes after seven in-game days, with win conditions communicated to the user through the game interface.
PreferencesScreen	FR_UI, UR_UX -> Ensures that the user interface meets expectations in terms of clarity and usability.
MainMenuScreen	FR_UI, UR_UX -> Ensures that the user interface meets expectations in terms of clarity and usability.

BackgroundMusic	NFR_PLAYABLE -> Ensure smooth operation without technical issues or crashes, contributing to a stable and reliable gaming experience.
ButtonClickSound	NFR_PLAYABLE -> Ensure smooth operation without technical issues or crashes, contributing to a stable and reliable gaming experience.

References:

[1] Lucidchart
[2] Visual-Paradigm Tutorial

Bibliography:

Sequence Diagram Guide Used As Reference