ENG1 - Assessment 2

Updated Architecture

Arch2.pdf

Group 4

Mikaella Loppnow	ml2708
Tom Daly	td1026
Ethan Buss	eb2225
Dillon Pandya	dp1195
Ereife Odusi	ed781
Harriet Kirby	hk1114

To Start the Architecture process, we followed a Responsibility-Driven-Design approach to determine the main themes and components of our game.

Designer Story

We are designing a university life simulator, to be run as a desktop application running locally.In the simulator, the player will control the avatar of a student and aim to pass their exams at the end of the 7 days. The avatar will be able to interact with buildings/people around the map, in order to perform certain tasks such as studying, sleeping, recreation, and eating.The primary objective of the player is to pass their exams at the end of the week and obtain the highest score they can by performing the various tasks with limited time and energy.The player wins if they pass their exams.

Main Themes

- Desktop interface for singleplayer play
- University Map with various activities that the avatar moves around on
- Revision phase of 7 days where you perform activities
- Exam phase at the end where you find out how well you did (score and win/lose)

Candidates

- Key Concepts
- Player
- Map
- Game screen
- Place to sleep
- Places to eat
- Recreational activities
- Places to study
- Locations of places
- NPCs
- Interactions
- Minigame screens
- Menu screen
- Highscore
- Energy bar
- Clock
- Score
- End screen
- Starting customisation screen
- Days
- Sound effects
- Controls
- Time
- Energy

Using these candidates, we created CRC cards which we then split into groups, which can all be found on the architecture page of our website.
We then decided upon a delegated control style, mainly using the different screens as states for the main objects that are split into “pools” of responsibility and can operate independently and a centralised Game object that delegates to the screens. We also decided that we would use the Entity-Component-System architectural style for classes on the main game screen. We then used PlantUML to consolidate these decisions and create all the diagrams seen in this document [1].

System Development Cycle

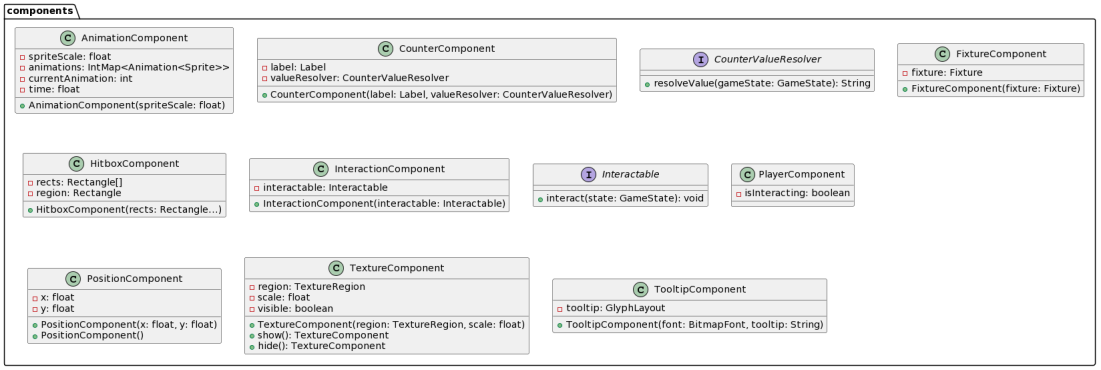
Following on from the responsibility driven design approach, there were several iterations of the system, designed alongside the implementation, fixing issues we came across and ensuring the system followed the requirements. As we progressed, we decided to cut out certain aspects of the game that were not essential to its functionality, so as to produce a

working game that met the requirements first and foremost. Amongst the things that were cut were the minigames. These were agreed to be a good idea for the game, as they would be fun and enticing to the target audience, which would benefit the games popularity. However, there was a lot of work required to implement them, so in the short timespan given for the project, we decided to remove them in favour of an animation that showed the user that they were in an interaction.
Another thing that was removed was the inclusion of a highscore on the end screen. We decided that this could be easily implemented at the end of the project if it was specifically wanted by the client, but wasn't required at this stage. This allowed us to focus on other areas of the project that were more time consuming and important to the architecture of the game.
The final thing that we decided to drop was the starting customisation screen. This was briefly mentioned by the client when asked about it as something that would be nice to have, but was in no way necessary. While it could be implemented with relative ease, due to the system architecture, it would have taken time to make it look good and would only provide additional content to the game rather than content that was needed.
Whilst several things from our original plan of the architecture were dropped, some things were added too. The main thing was the addition of the debug system. This proved to be an indispensable tool for debugging the game during implementation, especially during the collision implementation between the player avatar and the buildings. It allowed our developers to see the hitboxes of the entities and easily determine whether the physics of the game were working properly.
Another thing that was added, having been overlooked in the initial process, was a tooltip for the user when interacting with things on the map. This was a useful addition to the game as it made it more accessible and playable for the user and made it easier for those of us not developing the game to play test it without needing additional knowledge of the game.

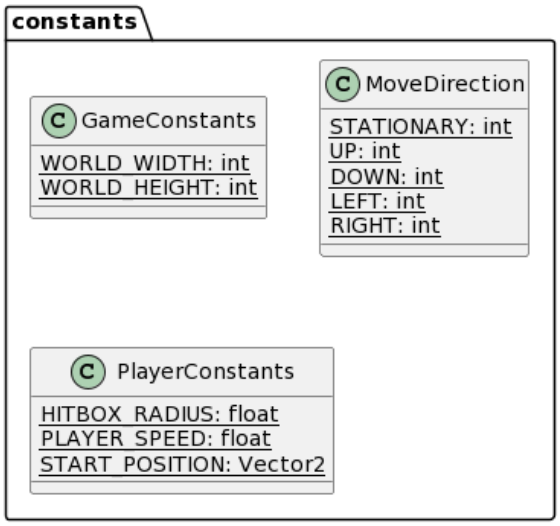
System Structure

All referenced diagrams can be found on the architecture page of the website.
As previously stated, our game uses an entity-component relationship as its structure, with a delegated control style. The main classes of the system are the screens, named Playing (originally named GameScreen) and MainMenu (based on the MenuScreen and StartScreen concepts). These two screens are in a package named screens and import code from most other packages, which can be seen in the *Third Implementation* section of the architecture webpage.All of the classes are split into relevant packages for the abstraction and readability of the system, which we will now go over.

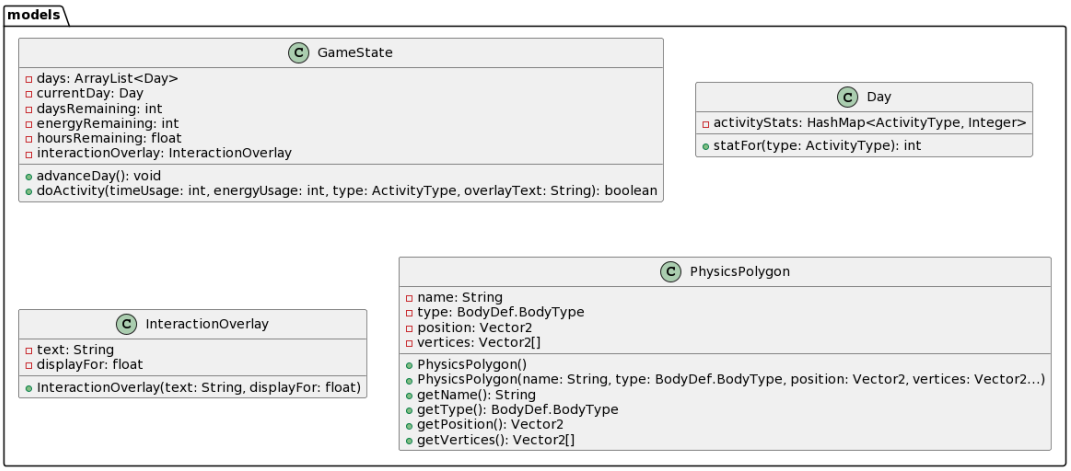
Components Package



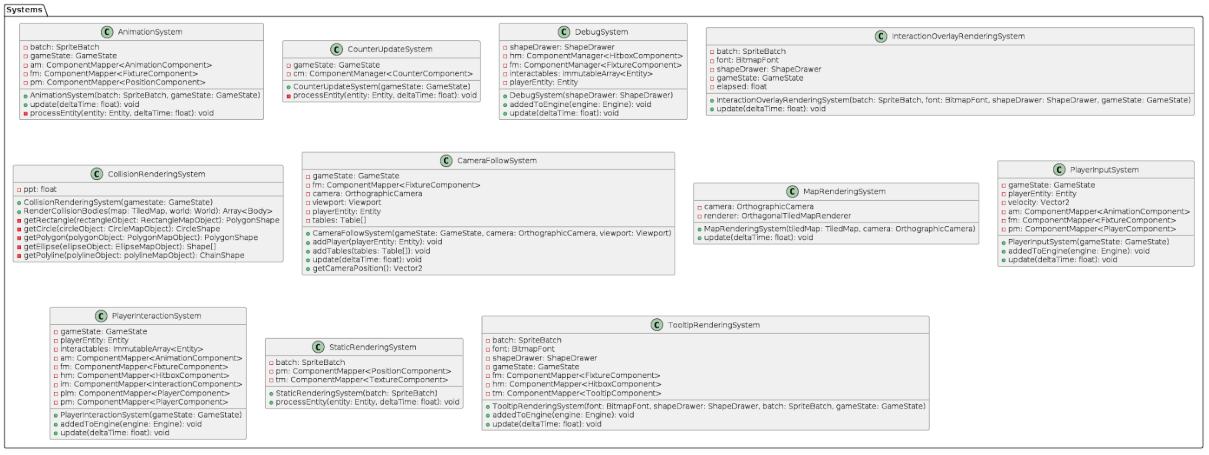
Constants Package



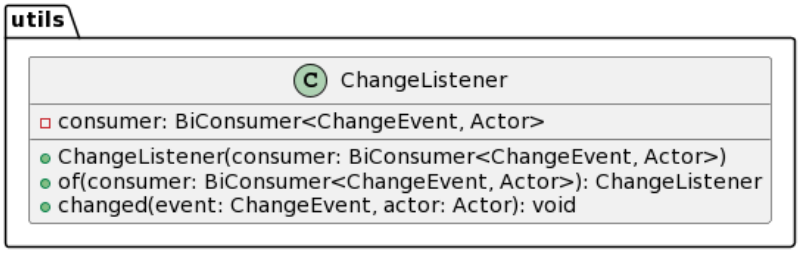
Models Package



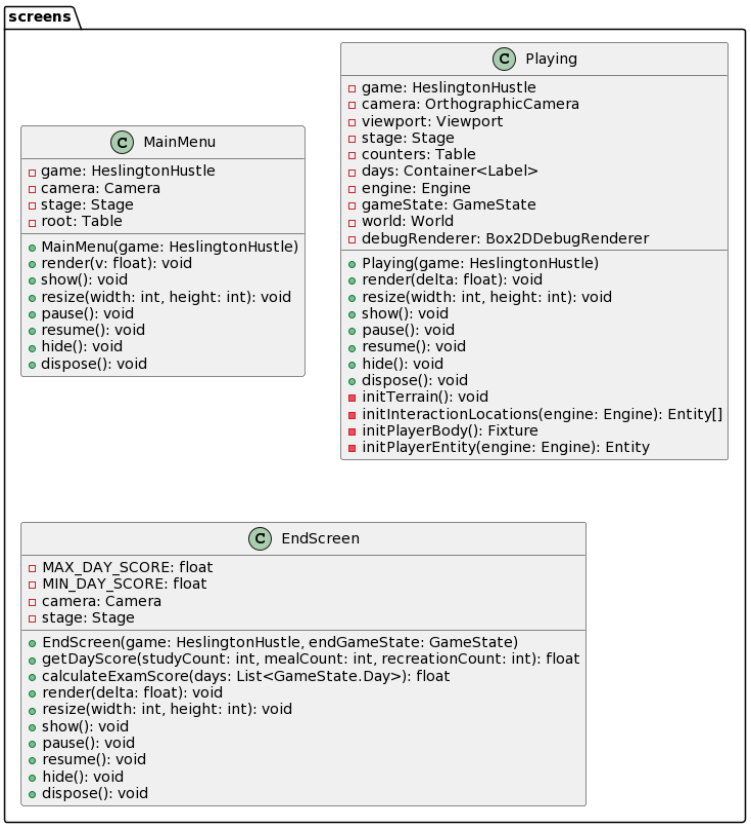
Systems Package



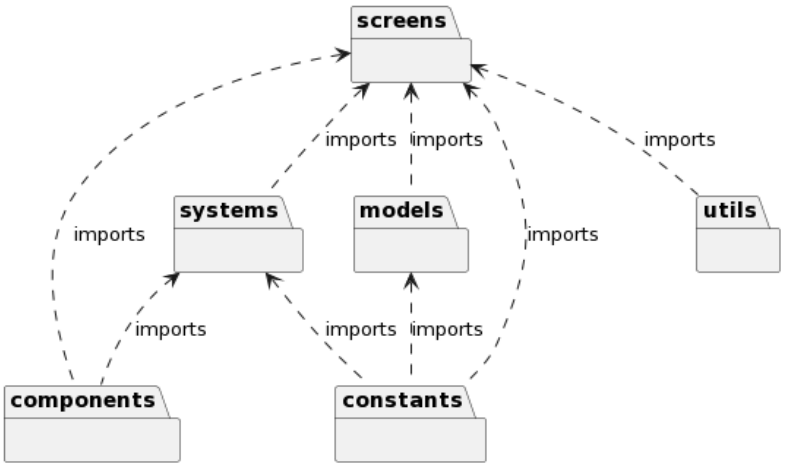
Utils Package

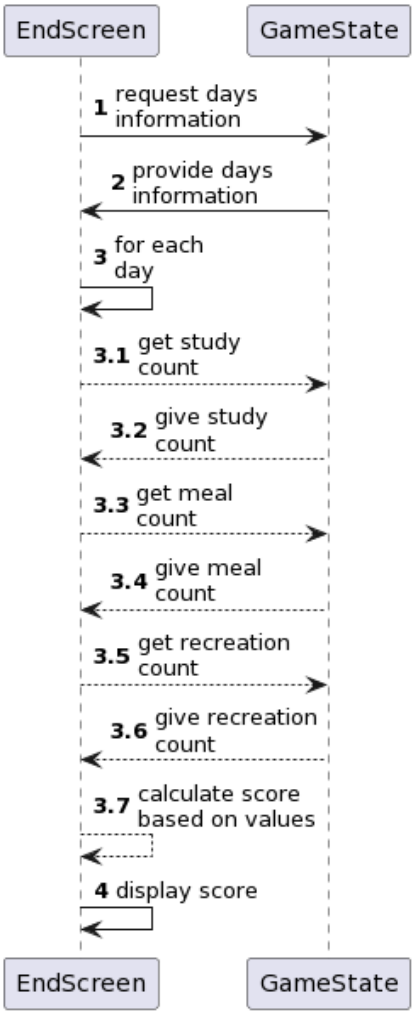


Screens Package



The packages are all linked to each other through importing to each other as represented in this diagram:

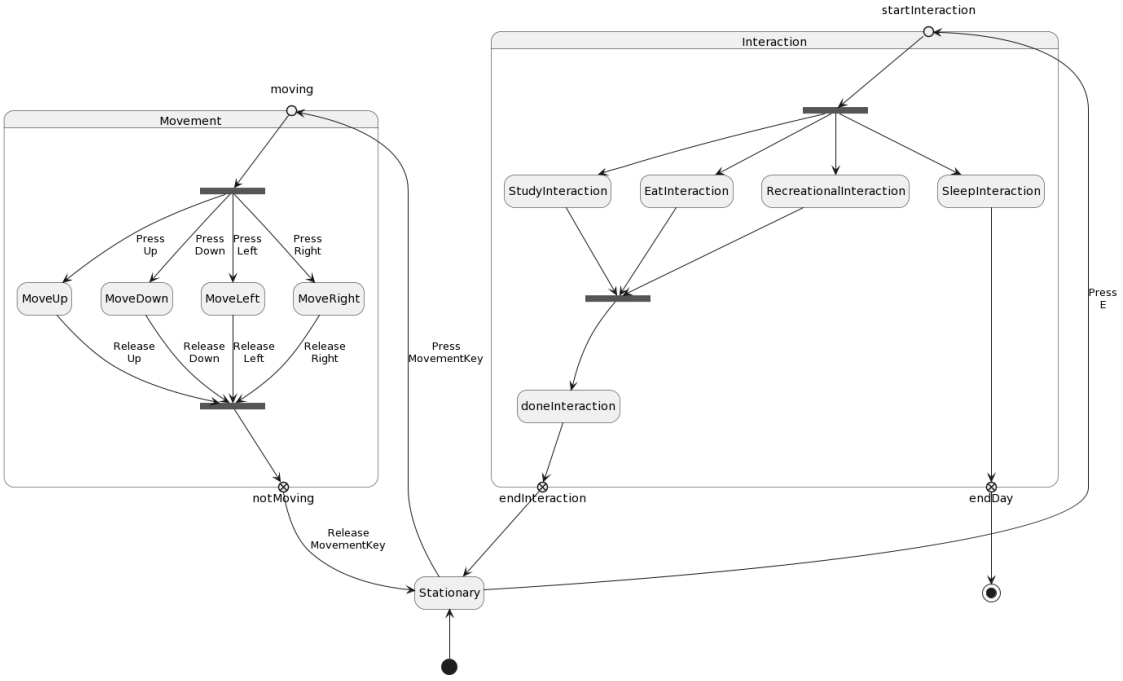




Score Calculation

The score is a vital component of the game as the aim of the game, as described by the product brief, is to pass your exams with the highest score possible. We decided to calculate the score at the end of the game with a deterministic calculation so that the same actions will result in the same outcome, as requested by the client. In the EndScreen class, there are two functions ‘calculateExamScore’ and ‘getDayScore’ that operate in the sequence shown on the left. We have all the important information about a given day stored in the GameState object so it is easily accessible by any part of the system. The EndScreen class imports this information, and using the methods in EndScreen, calculates the score. The score is then displayed on the screen as the client requested.

User Experience



In the game the user progresses through seven days, at the end of which they ‘take their exam’ and the game ends. The above state diagram shows what the player can do in a day in the game.

Architecture Characteristics

In designing this system, we have kept our architecture characteristics as close to those set out by the requirements and our client as possible. In terms of availability, our product never has downtimes, and we have new releases of the game when we are happy with the code. The code is easily deployable to desktop and laptop users by downloading one jar file. The code is easily extensible due to the way the code is split up and written. It allows for easy addition to a package and easy integration to whichever part of the system the new code is needed in. In terms of performance, the game is very low maintenance and doesn’t require very much processing power to run. From our testing the game is also reliable, without any obvious bugs.

Architecture Decisions

The architecture was directly designed to provide features that would satisfy the needed system and user requirements for the first implementation. Viewing the third implementation of the class diagram shown below, we can see the systems package which provides multiple classes that facilitate the satisfaction of key requirements. These rendering systems designated to different classes are responsible for features such as rendering interactions, the player, player movement, player animation and rendering the map. This links to satisfying user requirements such as *UR_MAP*, *UR_INTERACTION*, *UR_SLEEP*, *UR_AVATAR* and *UR_PC* including their associated system requirements both functional or non-functional. More importantly, it creates a base for the system to be able to create other interfaces and classes that make up the foundation needed to satisfy the remainder of the requirements.

The DebugSystem and HitboxComponent are both responsible for preventing the player from clipping into the environment of the map. This is not a needed requirement for assessment 1 however it ultimately helps satisfy the user requirement *UR_FAILURE* which should be prevalent in further iterations.

In the components package there are many classes that link to the avatar and the textures of the system. These classes allow the system to generate and manipulate in-game textures in addition to the systems package which are needed to satisfy *UR_MAP*, *UR_AVATAR* and *UR_ASSETS*

In the models package there is a GameState class that showcases the state of counters that must be kept track of such as currentDay and energyRemaining. In the systems package there is a class that is responsible for updating the counters which is used to satisfy *UR_ENERGY*, *UR_TIME_FINISH* and *UR_INTERACTION* since the system tracks how many of each interaction was taken by the player.

Amount of interactions is stored in a game state variable and updated using the CounterUpdateSystem function. The score is calculated inside of the screens package as part of the EndScreen class. These complete the needed user requirements for assessment 1 by fulfilling *UR_SCORE* and *UR_EXAM_END*.

References

[1] M.R & N.F. Mark Richards and Neal Ford. *Fundamentals of Software Architecture : An Engineering Approach*. O'Reilly Media, Incorporated. 2020-01-28.

[2] PlantUML. (February 14, 2024). *PlantUML at a Glance - Generating UML Diagrams* [Online]. Available: Open-source tool that uses simple textual descriptions to draw beautiful UML diagrams. (plantuml.com)