Architecture Deliverable

Team 17 - Rich Tea(m) 17

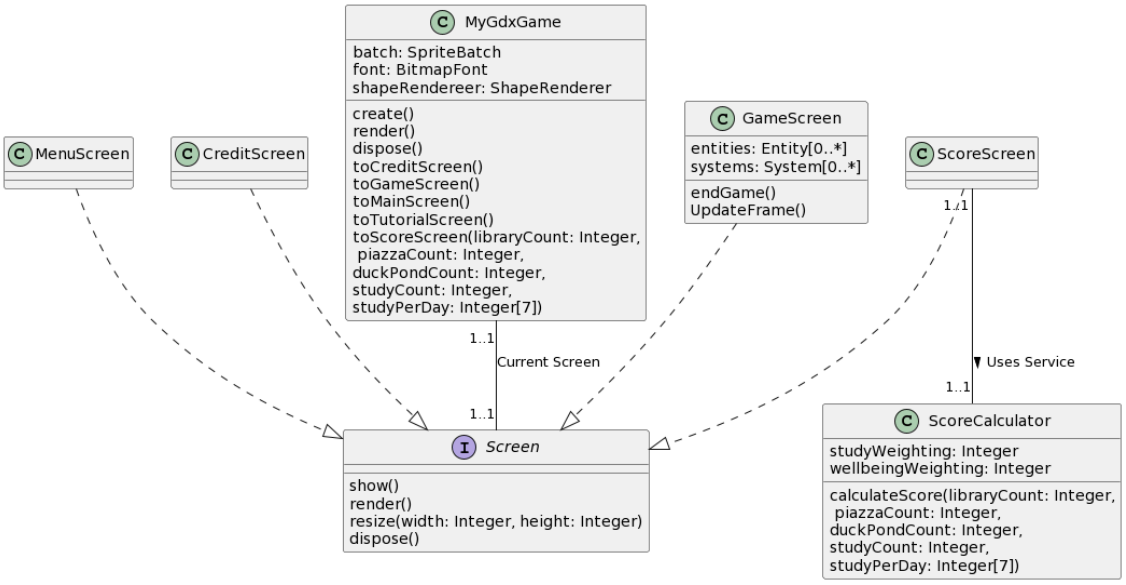
William Potts
Meg Tierney
Jamie Burgess
George Jopson
Alex Staicu
Seyi Towolawi
Ben Slater

Architecture

In this project we decided to use an Entity-Component-System (ECS) style architecture to implement the game aspect of the project, and we used a regular object-oriented architecture for the rest of the system.
An ECS approach works by having Entities which are instances of an “Entity” class that have no functionality but are composed of components. These components are simple stores of data. Then there are Systems which are called every frame, with Systems acting on any Entity that has the specified set of components. In this way, the components an entity holds specifies the entity's behaviour.

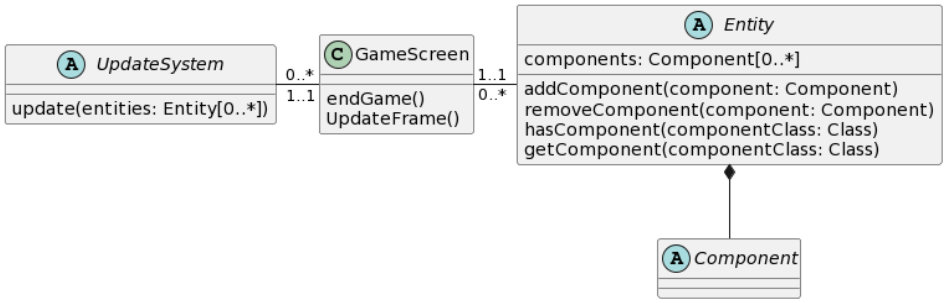
Diagrammatic Representation of the Architecture

Structural Diagram
Main System Class Diagram:



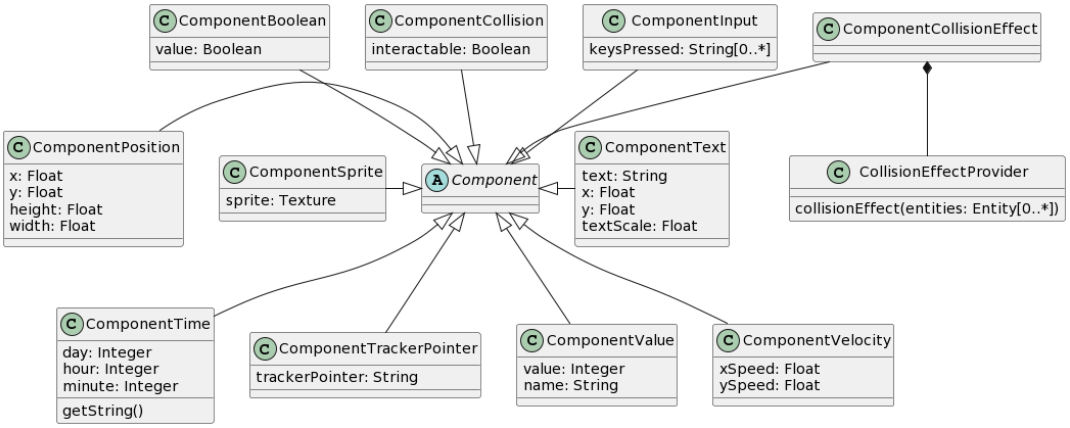
Justification and Link to Requirements:
In this architecture, MyGdxGame functions as the main loop of the system.
Through Responsibility-Driven Design we identified that the main components of the software were the separate screens which would then be displayed by the main game loop. The screens would be responsible for defining what should happen when they were created (the create method), rendered (the render method), and updated (the update method).
Some of these screens (such as the score screen) utilised service providers like ScoreCalculator. This meant that if we wanted to change how the score was calculated we simply had to update ScoreCalculator without changing the entire Score screen.
The screen objects are responsible for meeting the FR_MAIN_MENU, FR_CREDITS, FR_TUTORIAL as they implement the screens.
The game screen is then in charge of managing the ECS system to implement game related functional requirements like FR_MAP, FR_DURATION, and so on.

Game Class Diagram:



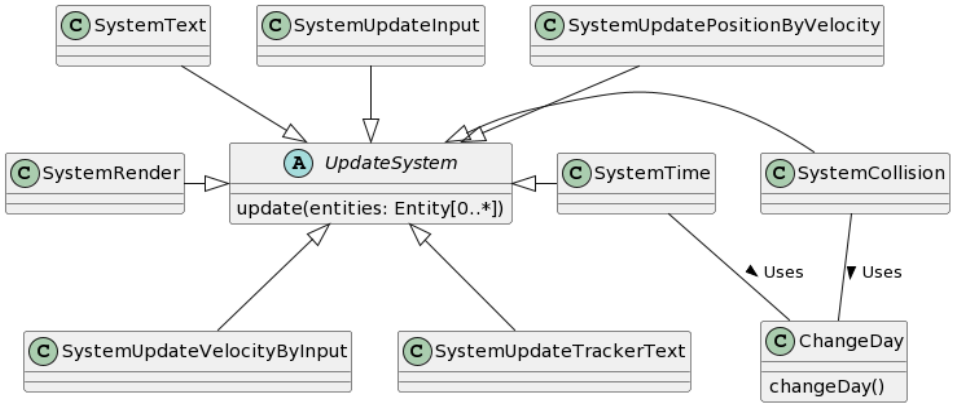
Justification and Link to Requirements: The "Game Screen" acted as the central hub of control from which systems could be called and entities stored. Then abstract classes for System, Entity, and Component were created to allow for the separate aspects of ECS to be created.

Component Class Diagram



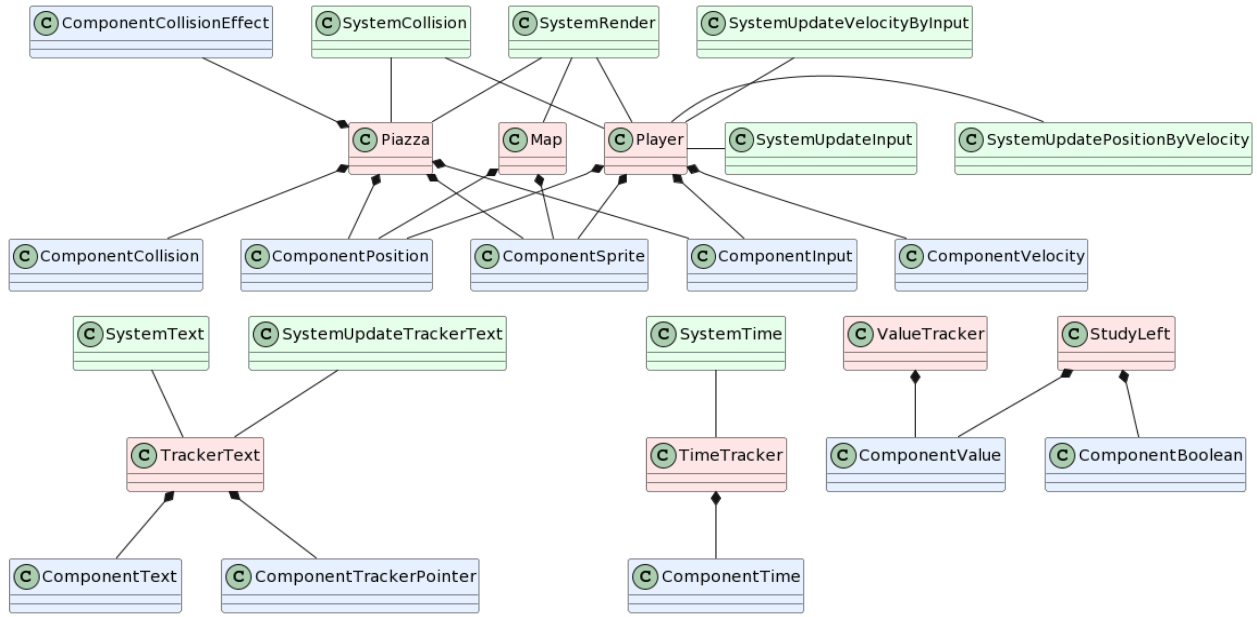
Justification and Link to Requirements: The components act as stores of information to allow entities to be able to fulfil their intended purpose.
For example the ComponentInput component was needed to fulfil FR_MOVE as the user's keyboard inputs needed to be stored, so that then the player character could be made to move in that direction.
ComponentCollisionEffect stores a service provider instance of CollisionEffectProvider, which is utilised to execute the effect of interacting with a Point of Interest (POI) such as increasing time. This was necessary to implement FR_LIBRARY/ FR_PIAZZA/ FR_ACCOMODATION/ FR_DUCKS.
The other components are also needed to keep track of information for other game related requirements.

System Class Diagram



Justification and Link to Requirements:
This diagram shows the different systems which were used to implement the required functionality. For example, SystemRender was needed to display Entities with sprites on the screen, which was necessary for FR_MAP to render the map.

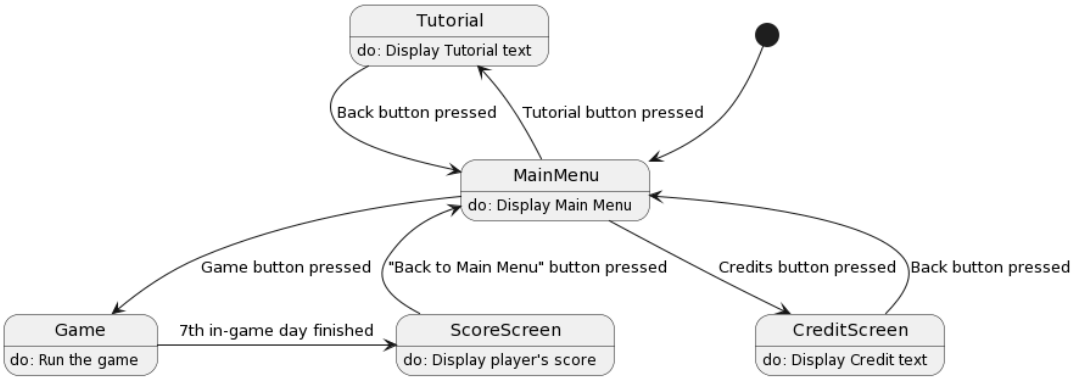
ECS Structural Diagram



Justification and Link to Requirements:
This diagram shows how Entities (red) are composed of Components (blue) and acted on by Systems (green). This system is used to implement all game related FRs, so for conciseness only FR_MOVE's implementation is explained below. The Player entity holds ComponentInput which stores the current keyboard input, with this being updated each frame by SystemsUpdateInput. Then this input is converted into a velocity (the direction of movement) by SystemUpdateVelocityByInput which is stored in ComponentVelocity. Finally, the position is updated to move the player in the direction of this velocity by SystemUpdatePositionByVelocity, with the position being stored in ComponentPosition. While multiple systems are required to do one task, it allows for high levels of flexibility and project extensibility. For example if we want to create an AI controlled enemy, we could use SystemUpdateVelocityByAI so the velocity was controlled by an AI decision rather than the keyboard input. You can also use the same components/systems in other aspects of the game such as ComponentInput being used for the POIs to check whether the "space bar" is being held down and so the player is interacting with them. Therefore, splitting the task of FR_MOVE into its component parts (each implemented by a different system) makes building the system easier (as each part has a small distinct purpose) and allows for the same systems/components to be utilised to solve new problems. This is vital in an agile environment where new features may be constantly added.

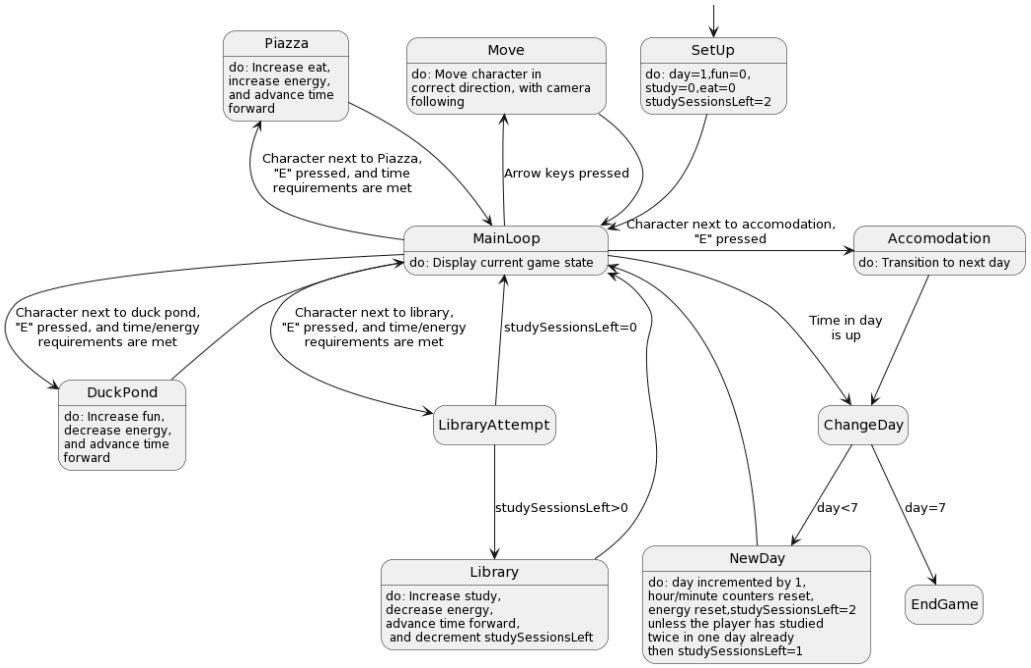
Behavioural Diagrams

State Diagram of Main System



Justification and Link back to Requirements: This diagram shows how the FR_MAIN_MENU, FR_SETTINGS, FR_CREDITS, and FR_TUTORIAL are implemented by showing how the system moves through these different screens.

State Diagram of Game



Justification and Link back to Requirements:

This shows the different stages of the game state. It shows how FR_DUCKS, FR_ACCOMODATION, and FR_PIAZZA should be implemented and what effect those activities should have. The logic for FR_LIBRARY is slightly more complicated as the user is only allowed to study once a day. However “studySessionsLeft” is initially set to 2, as once a week the user can study twice per day to catch up, as per FR_CATCH_UP. The “NewDay” state shows how the FR_NEXT_DAY requirement will be implemented.

Design Process and Evolution of Architecture over Time

Responsibility Driven Design

The first step in our design process was to follow Responsibility-Driven Design (RDD) to generate candidate objects. We then could use these to understand the pieces of software that would have to be created to create the game.

To complete RDD we generated Candidate-Responsibility-Collaborator (CRC) cards in the following method inspired by the method outlined by Wirfs-Brock [1]:

1) Generate a "designer story" by analysing the gathered requirements. The designer story describes the high-level process of playing the game. You can find the designer story we wrote here: **https://megant2004.github.io/ENG1-T17-Website/designStory.html**. This gave us a description of what the system should do that we could then follow.
2) Generate a set of candidate objects by analysing the designer story. This was initially done by identifying the nouns and themes in the story, which formed the key aspects of the user's experience. We also considered the software specific functionality (such as a candidate object needed to interact with the keyboard).
3) For each candidate object, we created a CRC card. We wrote a small description of the "purpose" of the candidate object, and identified the 2-3 role stereotypes that the candidate object fulfilled. If a candidate object had more than 3 role stereotypes then **Wirfs-Brock principle** of "Don't make any one role too big." would start to be violated. Therefore, to "objects [stayed] understandable" [1] we broke the candidate down into multiple smaller candidate objects.
4) Group related CRC cards(which could be the packages of our software) and then try to find any overlapping CRC cards.
5) Identify CRC card responsibilities and collaborators. We extended this to adding the functional requirements the candidate object will be responsible for fulfilling, to allow for greater traceability.
6) We repeated steps 3 to 5 several times, till we developed a model which covered all aspects of functionality.

While often paper CRC cards are used, we decided that using the online tool draw.io would be more appropriate, as it would allow for work to continue even when the team was spread out.

You can see our CRC cards here: **https://megant2004.github.io/ENG1-T17-Website/CRC.html**

UML Mapping

When generating the UML diagrams, the project was split in two.
For the non-game section, we followed an object-oriented approach. This allowed us to map the CRC cards to classes in a mostly one-to-one fashion.
However, it wasn't always a one-to-one conversion. For example the Button candidate object was split into the Button and ButtonAction classes, each Button having a ButtonAction instance which executes when the button is processed.

For the game section, we followed an ECS approach. This meant:

- Each candidate object became an entity (or multiple entities if necessary)
- What the candidate object should know became components
- What the candidate object should do became systems

This meant that each candidate object that shared functionality could have an instance of the same component (i.e. the DuckPond and Player having a position and therefore a Position Component).

Evolution of Architecture

Due to our agile approach, the system architecture changed as we continued development and understood the problem more. The original architecture can be seen here:
https://megant2004.github.io/ENG1-T17-Website/originalArchitecture.html

While many minor changes occurred to the architecture the major changes were things like:

- In the original architecture, the "camera" was owned by the Player, and then the CentreCamera system was responsible for centering the camera on the player. However in

researching LibGdx we found it was better for the SystemRender to be responsible for pointing the camera at the player's location. This meant all the details about displaying the sprites were all in one location (SystemRender).

- The architecture was simplified due to the scale of the project. For example, we decided not to create a whole "Engine" class to manage the entities/components/systems as the GameScreen could easily do this job. We also merged the ProcessCollisions and UpdateCollision systems into SystemCollision. By not spreading out functionality excessively it meant we could avoid an overly dispersed architecture which would lead to confusion.
- We changed entity to be an abstract class, using inheritance to create specific entity classes such as "Player" or "POI". For example to make a "Player" entity, previously we were creating an instance of Entity and adding all needed components. However, now you simply create an instance of "Player" which inherits from Entity and adds all needed components automatically.

Languages and Tools used

We used UML to specify the structural (with class and object diagrams) and behavioural diagrams (with state diagrams). We used an arrow with a straight line for inheritance, and an arrow with a dotted line for implementing an interface.
To generate the UML diagrams we used plantUML, which allowed us to specify the diagrams via text. We decided to use plantUML because it is under the GNU General Public License so it allows us "the freedom to use software for any purpose" [2]. This is important as it would allow us to still use the software, even if later the project became commercial and we could still use plantUML for free, even for a commercial purpose. A further benefit was the "Automated Diagram Generation" [3] which allowed us to quickly create new diagrams, perfect for an agile environment when architecture may change rapidly through the different sprints.
The only addition we made to the standard UML notation was that in diagrams showing the relationship between the entities, components and systems we used different colours to distinguish these separate groups. This added clarity to the diagrams.

Architecture Justification

We decided to use an ECS architecture over an object-oriented architecture for the game for the following reasons:

- **High agility:** [4] notes that "OOP principles such as encapsulation, message sending and inheritance can make game engine maintenance and scalability difficult" whereas an ECS approach allows for more flexibility. Entity's behaviour can be altered simply by composing them of different components, so new features can be easily added. This is vital in an agile project such as this one.
- **Code Reusability:** As systems and components are re-used for different entities it allows for high levels of re-usability (which is vital for a project with a short time frame such as this one where we do not want to be writing excessive code).
- **Performance:** NFR_CONTROL_RESPONSE specifies that the character must respond in under 0.05 seconds to player inputs. To meet this level of responsiveness, an ECS approach can run systems in parallel which can lead to a high level of performance.

However, for the non-game aspects of the system we decided to use an object-oriented approach. Implementing an ECS approach to build the main menu for example, would be excessive and lead to longer development times.

References

[1] R. Wirfs-Brock and A. Mckean, “A Brief Tour of Responsibility Driven Design” Accessed: Mar. 12, 2024. [Online]. Available: https://wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf

[2] “A Quick Guide to GPLv3 - GNU Project - Free Software Foundation,” Gnu.org, 2010. [Online] Available: https://www.gnu.org/licenses/quick-guide-gplv3.html

[3] “Frequently Asked Questions,” PlantUML.com. [Online] Available: https://plantuml.com/faq

[4] M. Muratet and D. Garbarini, “Accessibility and Serious Games: What About Entity-Component-System Software Architecture?,” Springer eBooks, pp. 3–12, Jan. 2020, doi: https://doi.org/10.1007/978-3-030-63464-3_1.