Architecture

Group 9

Lewis Ramsey
Toby Rochester
Henry Sanger
Remi Shaw
Ethan Spiteri
William Timms
Antonio Tiron

Introduction

We decided to use PlantUML for our project diagrams because it's simple and fits perfectly into our workflow. Since it's text-based, we could easily create and update diagrams using code, which also made it compatible with version control tools like Git. PlantUML supports a wide range of diagrams, including use case, sequence, class, and activity diagrams, so it was an easy choice for documenting all aspects of our system. Being open-source and free, it was also a budget-friendly choice compared to tools like Microsoft Visio or Lucidchart. We researched what the industry standards are for UML tools and found that PlantUML is widely used in development teams due to its integration with popular IDEs like IntelliJ IDEA and Visual Studio Code. *[1]* Many developers prefer PlantUML for its automation capabilities, ease of collaboration, and compatibility with Agile methodologies. After comparing it with other tools like Lucidchart and Draw.io, we found PlantUML to be the most developer-friendly, flexible, and efficient option for our needs.

Behavioural Diagram

Assessment 1

The first step we decided to do was to design a use-case diagram. We decided to do this because it provided us with a general overview of how the user will interact with the game, ensuring that once the game is implemented we have fulfilled the requirements and done so in a way which would be pleasing to the user. To create this diagram, we used a drawing application called Goodnotes 6. The reason we decided to use this application was due to the fact that it provided a simple, easy to use facility for designing a quick sketch, including the ability to draw shapes to contain the elements for each screen, as well as arrows to demonstrate the outcome of the user's interactions.

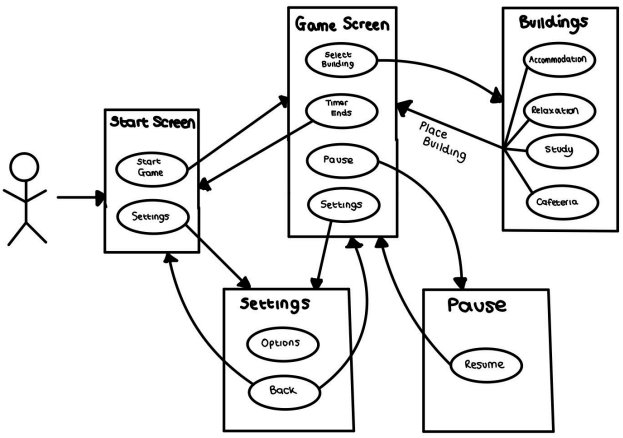

Figure 1: Initial Use Case Diagram

The next thing we wanted to do was to draw a sequence diagram. Now that we have a rough idea of what the game will look like, we wanted to see more specifically at which points the user will interact with the game and how. We decided to use a sequence diagram to

represent the flow of our game because it shows clearly how the user would interact with our system. This would then give us a good starting point to understand how to start coding the game from there, as it lets us keep the user in mind. We wanted to keep our design user focused and as close to the requirements as possible, so we decided that a use-case diagram would bring those out best.

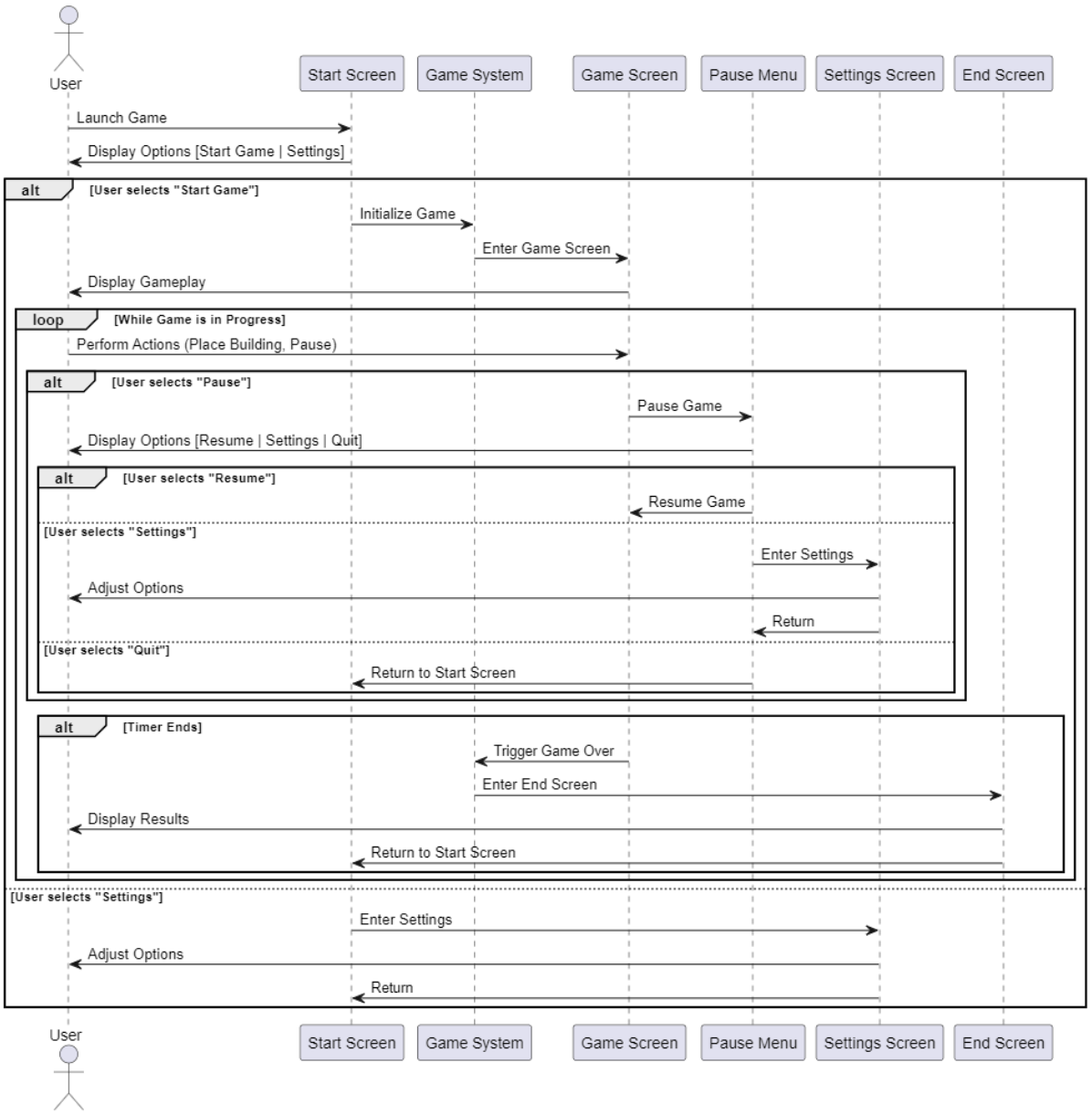

Figure 2: Initial Sequence Diagram

Writing a use sequence diagram was useful because we related each use-case to our requirements referencing. For example, including a leaderboard screen fulfils requirement FR_LEADERBOARD, and having these clear, easy to read buttons and navigation systems fulfils requirement NFR_OPERABILITY. Dividing the game up into multiple different screens makes it clear to the user where they should be going and what they should be doing. Therefore, having a settings menu will allow us to fulfil the requirement FR_MUTE, and including a pause menu will fulfil FR_PAUSE.

Assessment 2

In order to meet the deliverables for Assessment 2, we have updated the use case diagram and the sequence diagram to reflect the new requirements.

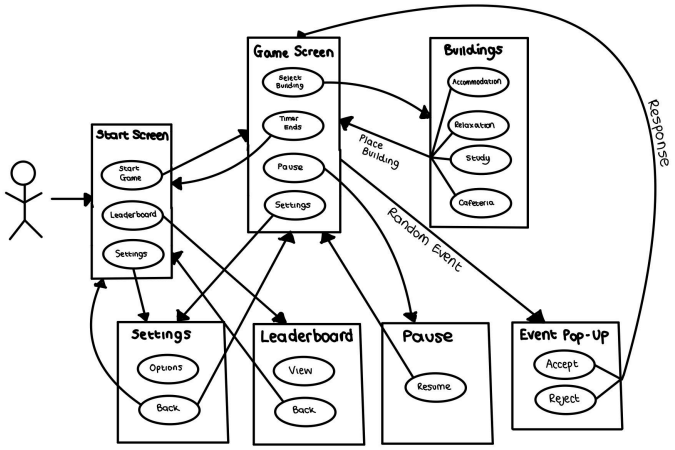

Figure 3: Final Use Case Diagram

The new requirements that we had to implement was a leaderboard, and an event system. To access the leaderboard, when the user first loads the game there will be a new option along with the “Start Game” and “Settings” buttons. This new option will be a button to view the leaderboard of previous game scores. Once the user has finished viewing the leaderboard, they will simply press a back button that will return them to the start screen where they can choose to start a new game and add scores to the leaderboard. This functionality will satisfy both the UR_LEADERBOARD and FR_LEADERBOARD requirements.

The event system will be internally controlled from the game screen. When a random event is triggered, the user will see a pop-up menu with two options for a given event scenario: Accept or Reject. Depending on the scenario, the user can accept or reject the event and will be rewarded or lose satisfaction based on their choice. Once they have selected their choice, the user will have their attention shifted back to the game screen where they can continue to expand their campus and manage incoming random events. This functionality will satisfy the FR_EVENTS requirement.

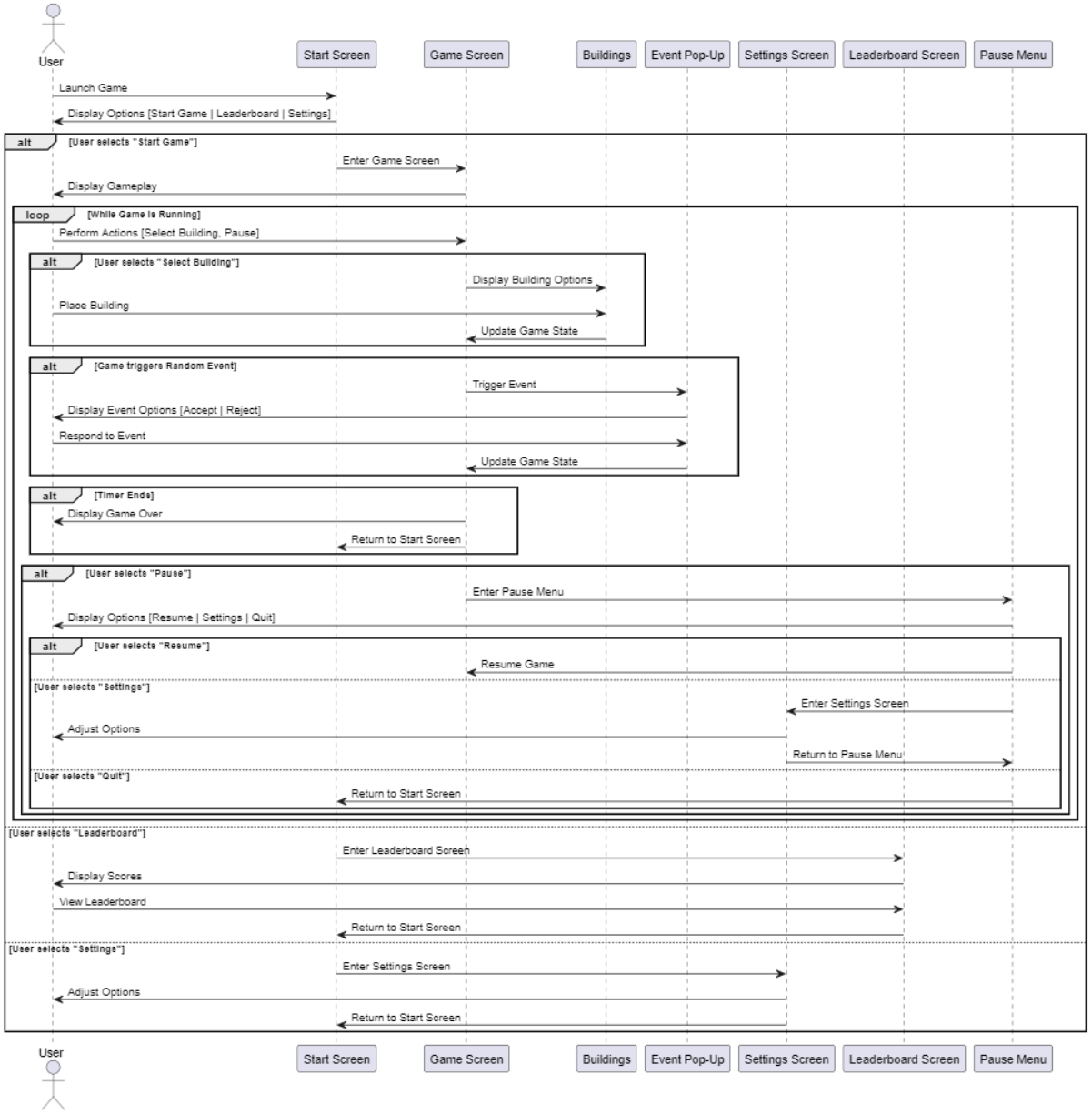

Figure 4: Final Sequence Diagram

Class Diagram

Assessment 1

Now that we have an initial overview of how to approach coding the game, we can develop this further by thinking about how to structure the code itself. The PlantUML extension was used in the Visual Studio Code interactive development environment to create the UML diagrams for the key parts of the architecture (for example map and buildings). We decided to use a class diagram to represent this because it allows us to balance visualising the concepts whilst still being technical. For example, we can visualise how classes relate to each other and the purpose of each class whilst also making sure we can see how the user will interact with the system as a whole.

Initial Diagram

The class diagram above represents our initial design of the game. In order to generate this, we used our requirements to understand the core functionalities required for the game to meet the criteria set by our customer. We started by thinking about how to code the buildings, and agreed that the best approach would be to use inheritance. So we made an initial sketch, as shown below.

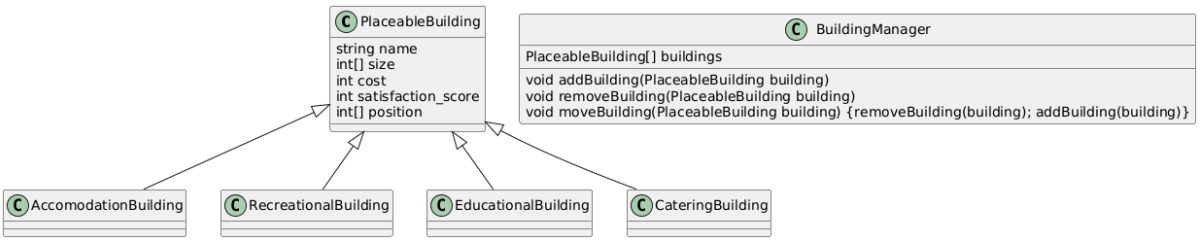

Figure 5: Initial Class Diagram

The architecture for buildings has made use of inheritance and classes to simplify the processes involved with buildings in the game. For example this design choice makes the placement and destruction of all types of buildings (user requirements: UR_PLACE_BUILD_SDY, UR_PLACE_BUILD_EAT, UR_PLACE_BUILD_HME, UR_PLACE_BUILD_RLX, UR_DESTROY_BUILD) much easier, because the main functionality of placing a generic building can be defined, then it can be called with specific parameters needed for the current building to be placed. In addition, the specific buildings inheriting from a general building class not only makes their design modular, but it also allows for editing/updates and maintenance to be carried out in a better way, due to changes to these building classes not affecting anything else as they are on the lowest level of the inheritance hierarchy.

Final Diagram

Buildings

As the code has been developed over the course of the project, the architecture has changed in all aspects; including the buildings, where currently the four types of buildings

now inherit from a class Building, which has been done this way to reduce code duplication. In addition the class building manager manages the placement of buildings, including functions such as addBuilding(), removeBuilding() and moveBuilding() and it manages the building function. The class BuildingCounter counts the class Building therefore counting every building placed or removed (incrementing and decrementing respectively) which relates to the functional system requirement FR_BUILDING_COUNTER. These cover the user requirements similar to the initial design which are UR_PLACE_BUILD_SDY, UR_PLACE_BUILD_EAT, UR_PLACE_BUILD_HME, UR_PLACE_BUILD_RLX and UR_DESTROY_BUILD. These changes to the architecture of buildings were made to make the functionalities of placing buildings more efficient and to be able to implement the process of counting buildings, relating to scoring the game.

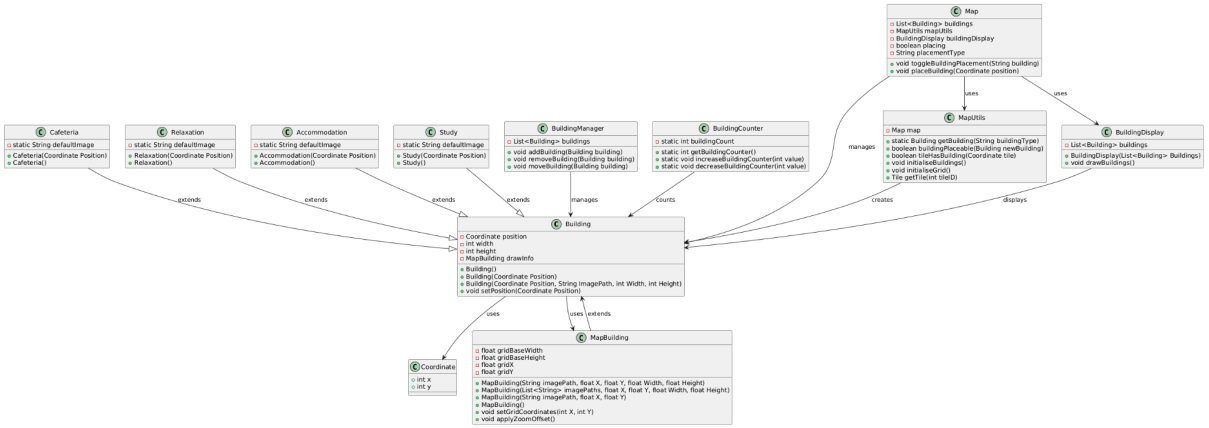

Figure 6: Final Building Class Diagram

UI

In regards to the user interface and how that interacts with the user, more functionalities have been added to the window class that work outside of the users domain, such as updating the resolution when the size of the window is changed. In addition to this the functionality of buttons has been added to the system, where this change allows for more intuitive interaction between the user and the system, relating to the user requirement UR_UX. The button class extends the component class which is used by the window and draw class, and the component class also uses the Sprite and Spritesheet class. This change in architecture to the usability of the system drastically improves the users experience when playing the game, as more intuitive interaction means less complex thought processes and less problems to come across for the user. This relates to the nonfunctional system requirement NFR_OPERABILITY, in regards to making the game accessible to users with a range of computer knowledge.

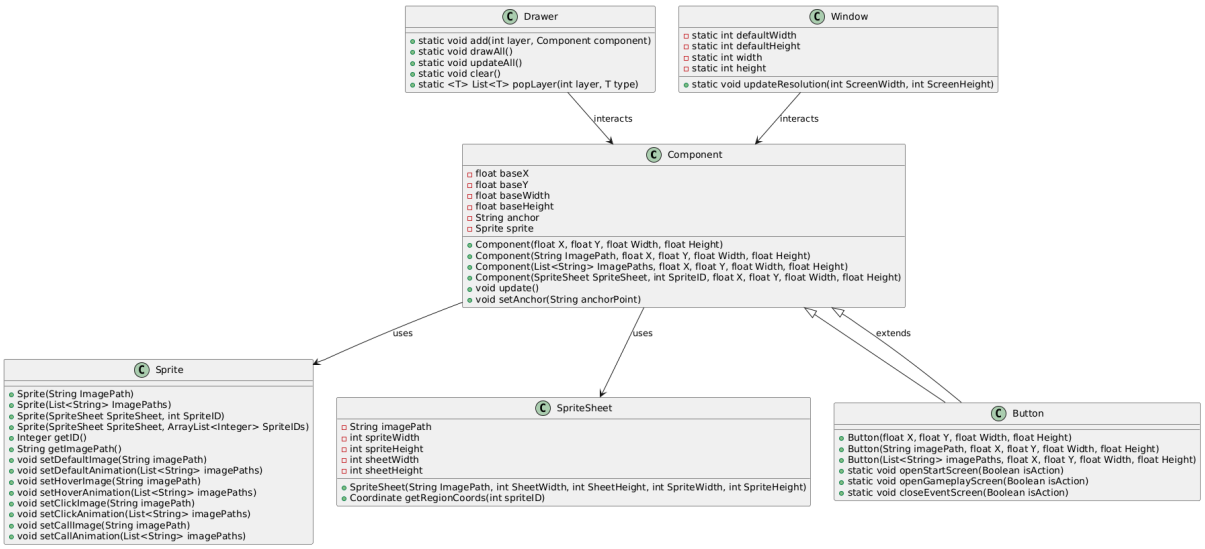

Figure 7: Final UI Class Diagram

Map Classes

In the case of the design for the map, this has been designed in a similar way to the buildings in terms of using inheritance in its implementation. This was done for easy updating by making the tile's classes modular. The functionalities for the map have been designed in classes and grouped based on their similarities and dependencies, which aids in understanding of code and allows for easier implementation of functionalities such as changing the size of the display, making the display automatically fit the window etc (user requirements: UR_SIZE_CHANGE).

The architecture of the map has also had a large redesign, where the Map class now uses two other classes, MapUtils and BuildingDisplay, where MapUtils manages class Building and initialises (creates) the buildings to be placed, and BuildingDisplay displays the buildings.In terms of the buildings being placed on specific locations on the map, relating to user requirements UR_PLACE_BUILD_SDY, UR_PLACE_BUILD_EAT, UR_PLACE_BUILD_HME and UR_PLACE_BUILD_RLX; the class Building uses class Coordinate to place buildings on specific locations on the map, also using the class MapBuilding to further this functionality. These changes were made to implement more precise placement of buildings in a more modular way, to improve code quality.

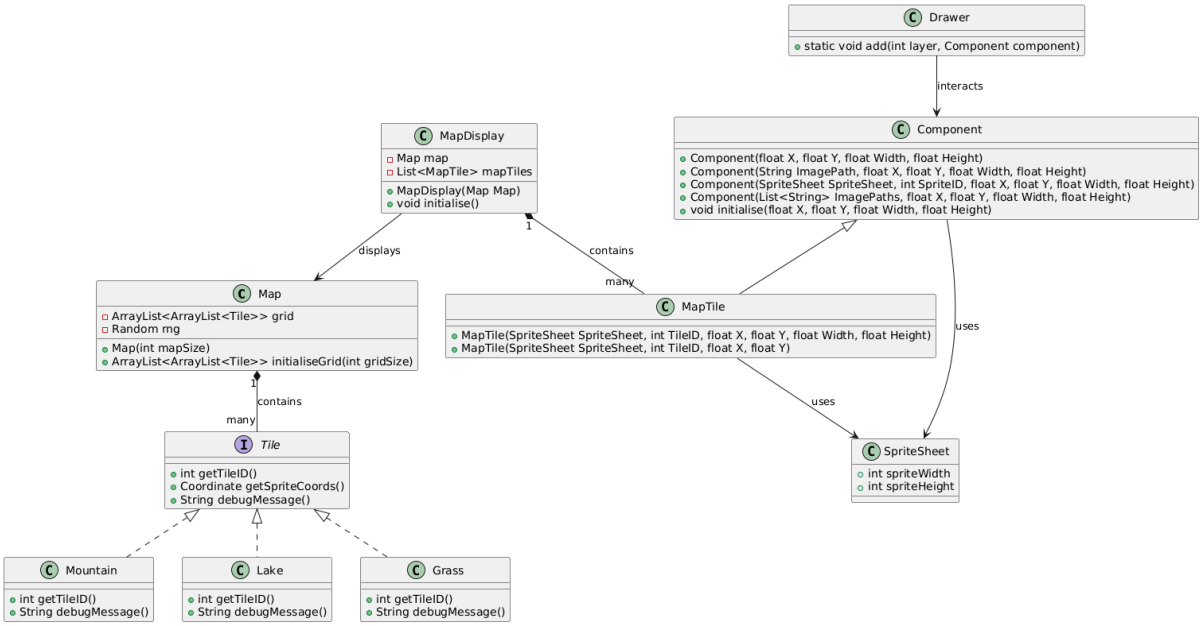

Figure 8: Final Map Class Diagram

Assessment 2

We spent some time getting familiar with the structure of the code and how the other team tried to implement the game. After careful consideration, we decided that the way their classes were structured wasn't very easy to read, and that they did not include all of the classes. For example, we liked how they wanted to separate the buildings from the UI and the maps of the game, but we felt that some classes belong in other categories. This can be seen with the Buildings package, where we took out the Map related classes like Map and MapUtils and placed them into different packages. This separates out the actual building classes and classes that control and manipulate them, making it more intuitive to read and understand. This directly supports requirements like placing buildings (UR_PLACE_BUILD), moving them (UR_MOVE_BUILD), and ensuring proper collision detection and placement limits (FR_BUILD_COLLISIONS, FR_MAP_COLLISIONS, UR_BUILD_LIMITS). We also added more subfolders for further clarity, for example Tiles to group all of the different types of tiles together.

We've updated all the diagrams to make sure they meet the brief for Assessment 2 and properly reflect the key features outlined in the requirements. For instance, we added classes for leaderboards and events. These changes directly address FR_EVENTS, which requires at least three events that affect gameplay and let the user respond, and FR_LEADERBOARD, which ensures the system includes a leaderboard showing the top five scores. These updates not only cover the functionality but also clarify how these features tie into other parts of the system, like scoring (UR_SATISFACTION) and achievements (FR_ACHIEVEMENTS).

Overall, the diagram now give a clear and complete picture of how the system works while fulfilling the requirements.

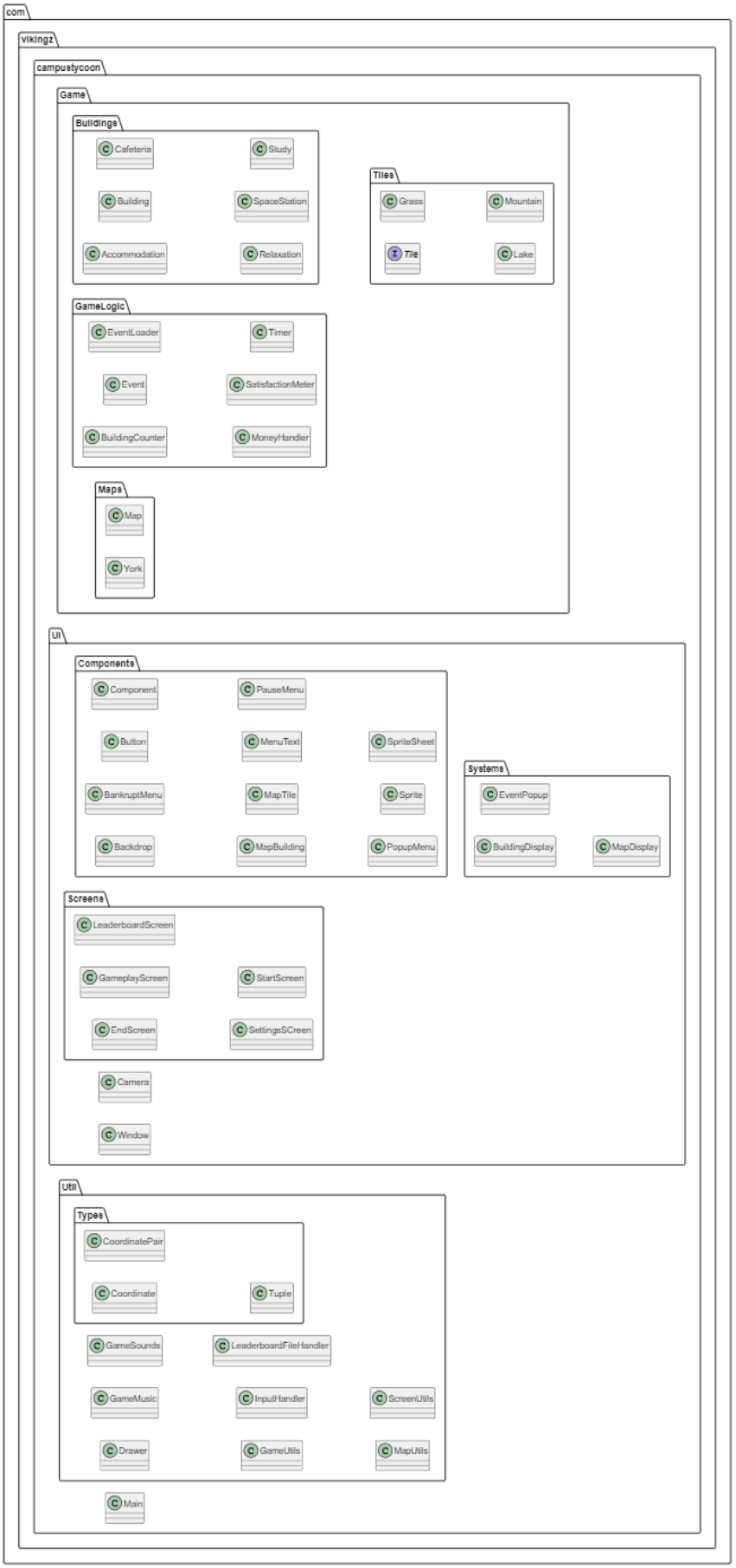

Figure 9: Final Class Diagram

References

[1] https://swimm.io/learn/code-documentation/why-you-need-a-coding-diagram-and-tools-to-get-you-started