Architecture

Assessment 1

Architecture Design Process

After creating the requirements, we had to make some decisions about the style and theme of our game before we could progress with creating an initial architecture, because the customer gave us a lot of freedom on those aspects of the product. The theme we decided on was retro-realism. We felt this would satisfy the non-functional requirement NFR_THEME because it is a popular style for video games of a similar nature that are played by similar demographics, such as older versions of The Sims. We also decided on a tile-based art style, because tiles integrate perfectly with the retro part of the theme, and they also simplify the implementation of the game, which will help to achieve the non-functional requirement NFR_PERFORMANCE.

We chose to use Responsibility Driven Design (RDD) to create the initial architecture for our game. Due to its focus on behaviours, we felt that it would give us freedom to abstract the low-level details for as long as possible while developing the design. This is because RDD brings the benefit of increased encapsulation by leaving the implementation of responsibilities to be private to each RDD object. It is also a well suited design technique for developing object-oriented programs (ideal as we are working with Java) as the technique makes good use of the inheritance hierarchy.

We began by creating a designer story based on the requirements that we had established and the style and themes we had decided upon. The goal here was to write a few sentences which encapsulated the requirements which would help us to all be on the same page with what we were creating, and to decrease friction during the design process. We settled on the following:

“We are developing a single-player 2D sandbox game where the user has 5 minutes to create a university campus by placing buildings on a predetermined map. The game is to be run as a desktop application running locally. The map will be tile-based and buildings can be placed on the tiles by the user. There will be a 5-minute timer that counts down. There will also be a counter for how many of each building category the user has placed.”

From here we began creating our first themes and candidate objects, and iterating on them, following the Responsibility Driven Design process using CRC cards (photos of these can be found on our website). During this stage, we were leaning towards a delegated control style, as it would allow us to keep our discussion at a higher level and we wanted to delegate work to objects that are more specialised to keep the design easy to understand. Once we were happy with our set of candidate objects, we began to map our candidates to classes and constructed our initial UML class diagram (Diagram 1), representing the overall structure of the project as we envisioned it at that point.

As shown in the diagram, we decided to use a closed, layered structure for our architecture, which takes the form of an inclusion tree in our class diagram. This gives one top-level object (Game) responsibility for everything in our application, and it handles this by dividing the work into smaller sections, and delegating them to smaller objects in the layer beneath, which it holds references to.

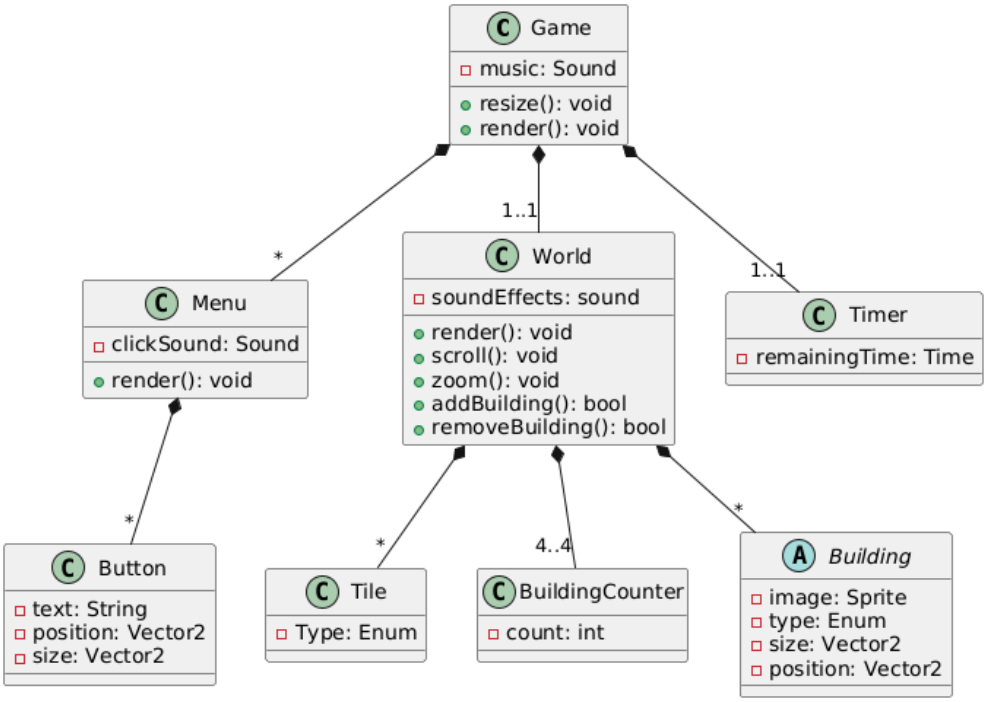

Diagram 1

We decided upon this structure because it encourages modular design, each object delegating its unrelated tasks to separate objects on a lower layer, and having no knowledge of other objects or systems not in the same subtree. By having objects not hold references to objects higher in the tree, we reduce bugs and keep the inclusions logical. Similar class structures are used in modern games and game engines, because the tree is easily scalable and it makes best use of the object-oriented paradigm.

Another advantage of this architecture is the ease of adding new features. Due to the nature of systems being separated into their own subtrees and not having access to the layers above, adding a new system can be done by creating the necessary classes, then inserting them into the right section of the tree. For example, if we were to add a player to the game, we would create a class to manage the data and behaviour of the player, which we would then create a reference to from the world class (because the player lives within the world). We would then just modify the render method for the world and we would be done. The main advantage of this is that no other parts of the game would be affected, increasing code modularity and readability. This quality is needed to satisfy the non-functional requirement NFR_CODE_MODULARITY.

Before deciding on this structure, we rejected a lot of other concepts. One we considered heavily was the use of an entity component system. We recognised that the use of this type of system can be very advantageous in games, both in terms of speed, code structure and readability. The main benefit of this comes when there are a lot of different objects that reuse subsets of a set of components. We decided that in the case of our game, which has only a few types of objects (buildings, tiles and menus), using an entity component system would not be advantageous. Because there are so few objects, and in order for them to be implemented efficiently, their implementations will be quite dissimilar, the advantages of an

entity component system would be very small, and not enough to outweigh the increased complexity introduced by such a complicated system in such a small game.

The above class diagram helps us to meet the following functional requirements:
FR_MAP - By the inclusion of the world class, which holds a collection of Tile objects, which make up the map.
FR_BUILDING - The building class will facilitate the placement of buildings.
FR_OBSTACLES - By the use of tiles, we can have certain tiles be non-buildable, which could then be used to make obstacles on the map.
FR_TIMER - The timer class will allow us to display a counting down timer to the user.
FR_BUILDING_COUNTER - Facilitated by the building counter class

Evolution of the Architecture

Implementation for the project lasted for about 3 weeks. For the first week or so, we followed the initial architecture class diagram quite closely. As we were working with the libGDX game engine, there were a few things that we changed, such as making use of the libGDX classes TiledMap and IsometricTiledMapRenderer. This eliminated the need for our own Tile class, and reduced the complexity of the world class, which improved the conciseness of the code, and harnessed the well-optimised code provided by libGDX. This helps us work towards requirements NFR_CODE_MODULARITY and NFR_PERFORMANCE respectively. To reduce confusion, we also renamed the top level Game class to Main, as the top level libGDX class that Main/Game needs to extend is already called Game.

When we came to implementing the menu interfaces for the game, it became clear that, in order to make optimal use of libGDX's menu framework (scene2d-ui), we would need to deviate slightly from our original application structure. This is because of the way scene2d provides multiple menus - the Screen interface. This would allow us to easily switch between different menus and the game screen without having to reload each menu before every switch. In order to take full advantage of this, it made sense for the World object to be held by a Screen Class, rather than the game class, and then make the Main class hold a reference to all the different screens.

Quite soon after we began to implement the idea above, we encountered a problem with this change to our architecture: it became necessary for the individual implementations of the Screen interface to be able to modify the current screen that is being displayed to the user. For example, if you are currently being shown the startup screen of the game, and you press play, the screen needs to detect that click, then somehow notify the Main class that it needs to switch the current screen to the game screen. This conflicts with our ideal tree structure in which each object has no knowledge of the objects that hold references to it.

At this point, it became necessary to research how more established libGDX games accomplish this. Taking inspiration from the game Mindustry, which has a class called Vars[1]that holds all the information about the current state of the game as static members, we

[1] "Anuken/Mindustry," *GitHub*, 2017. https://github.com/Anuken/Mindustry/blob/master/core/src/mindustry/Vars.java (accessed Nov. 07, 2024).

created a class called GameState which holds information such as the current screen and whether or not the game is paused as static member variables. This allows those fields to be accessed similarly to global variables by any code that needs to. It is important that we make only minimal use of this class, as overdependence on global variables can lead to code that is very difficult to trace.

With these new changes, we were able to continue smoothly to the end of the current development stage, implementing all the features required for this assessment. The tree structure is now split into individual screen classes and the state of the game is stored as static members of a class. This was necessary because we are working with the event-based libGDX in combination with our own layered architecture. This highlights how it is not necessary to conform perfectly to any specific architectural concept, but to take the best bits of all of them to suit our needs, while still leaning more towards one (layered). The final structure of the architecture is shown below. (Diagram 2)

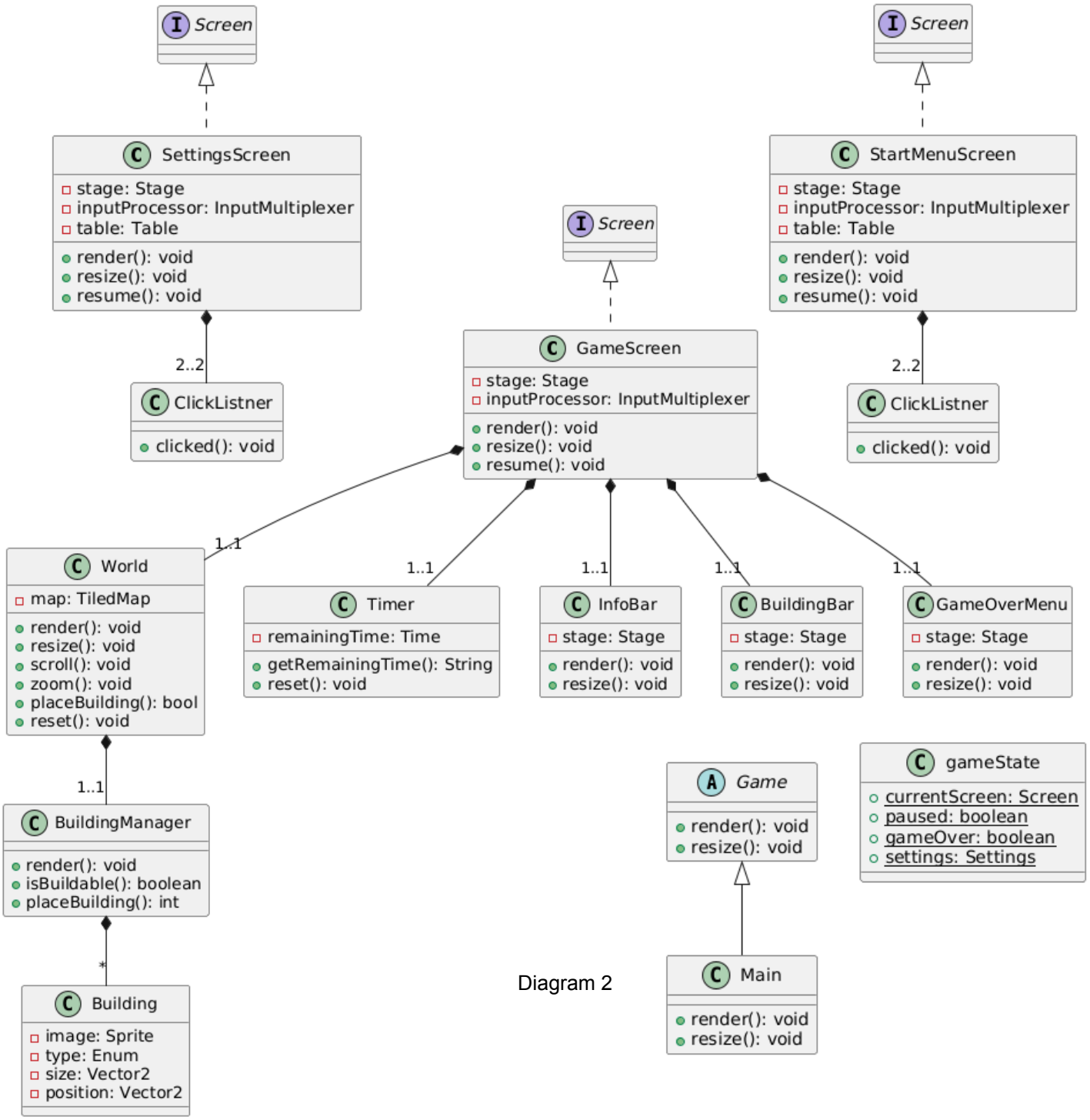

Diagram 2

Behavioural Diagrams

We created a state diagram to show the different game states that the program can be in throughout the gameplay. The idea of this diagram is to show all the different menu screens at a high level. (Diagram 3)

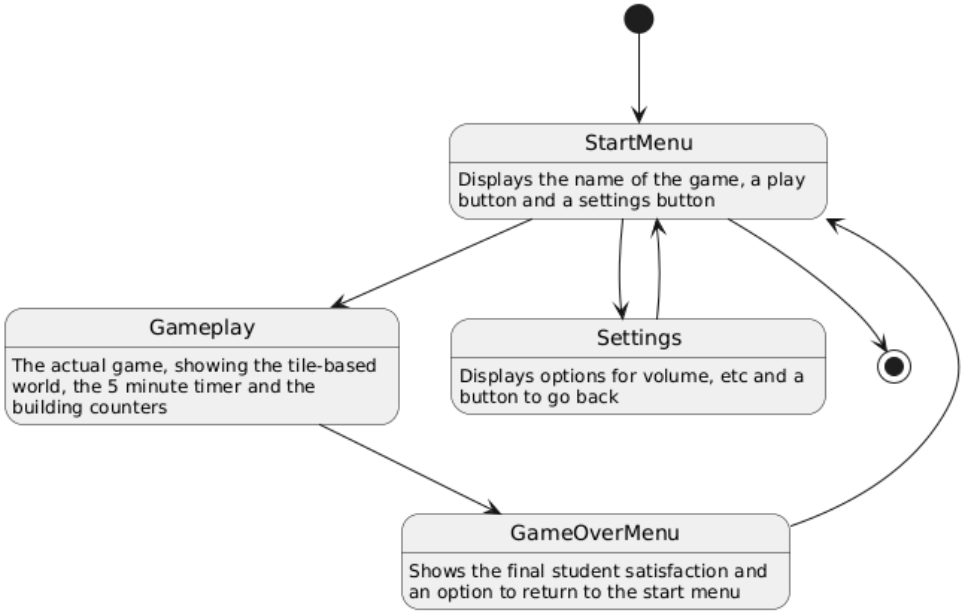

Diagram 3

We also created sequence diagrams to show the main actions that can happen during gameplay and relate these to our architecture. The most important (complex) behaviours that can be invoked by the payer are placing a building and traversing the map. The UML sequence diagrams for each can be seen in Diagrams 4 and 5 respectively. An interim version of the sequence diagram for placing a building can be found on the website.

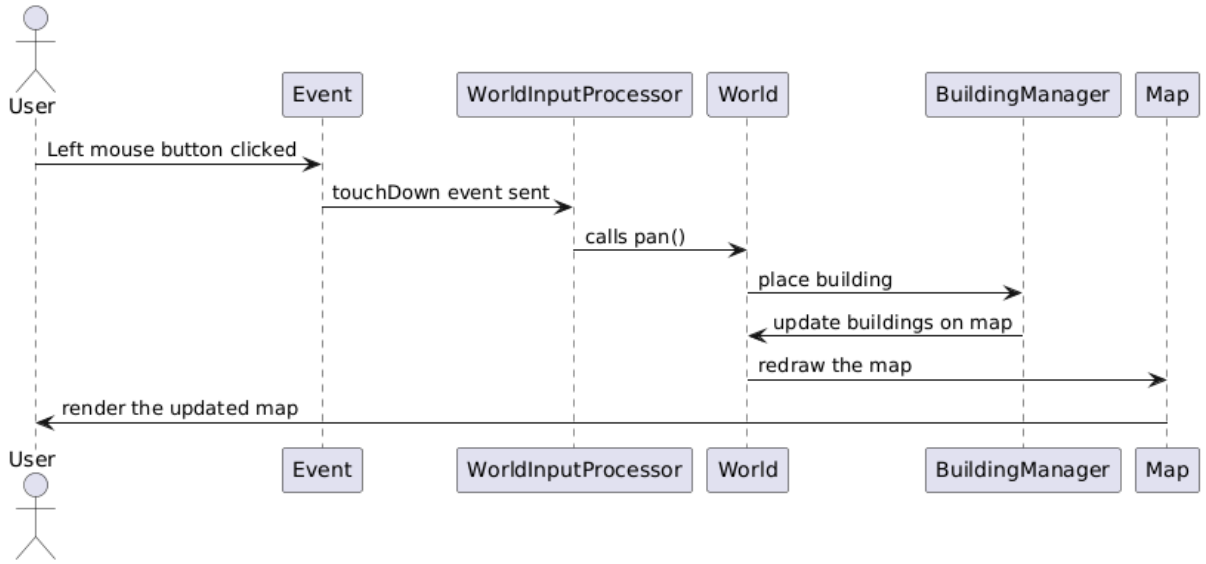

Diagram 4

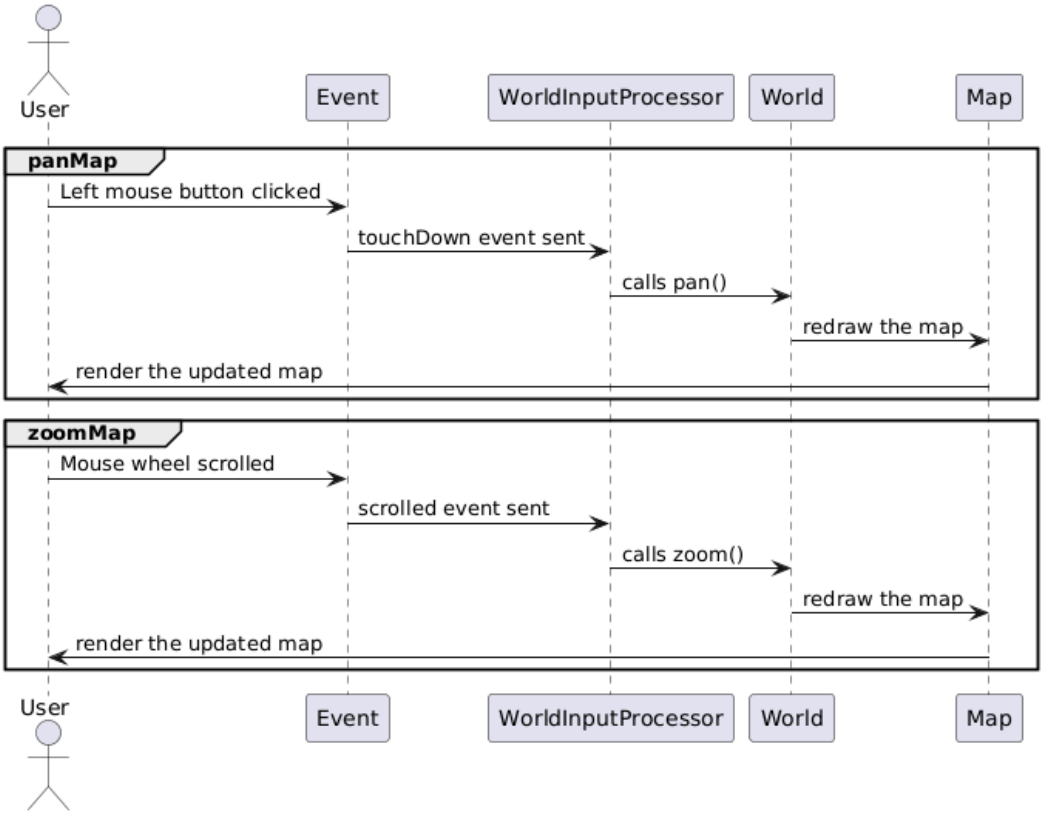

Diagram 5

Diagram Creation

We chose to create our architectural diagrams using the class diagram, sequence diagram and state diagram sections of PlantUML[2]. This is beneficial because the diagrams can be specified using text, then rendered dynamically using an interpreter. This suited our needs very well as the text-based sources for our diagrams integrated very well with Git, which was our chosen version control solution. This was important due to our use of an agile software development methodology, which necessitated evolving the structure of the project, and therefore the diagrams over time as we moved through development. The diagrams were rendered for this document using the online server on PlantUML

[2] “PlantUML” *PlantUML.com*. https://www.plantuml.com/plantuml/uml/

Assessment 2

Original

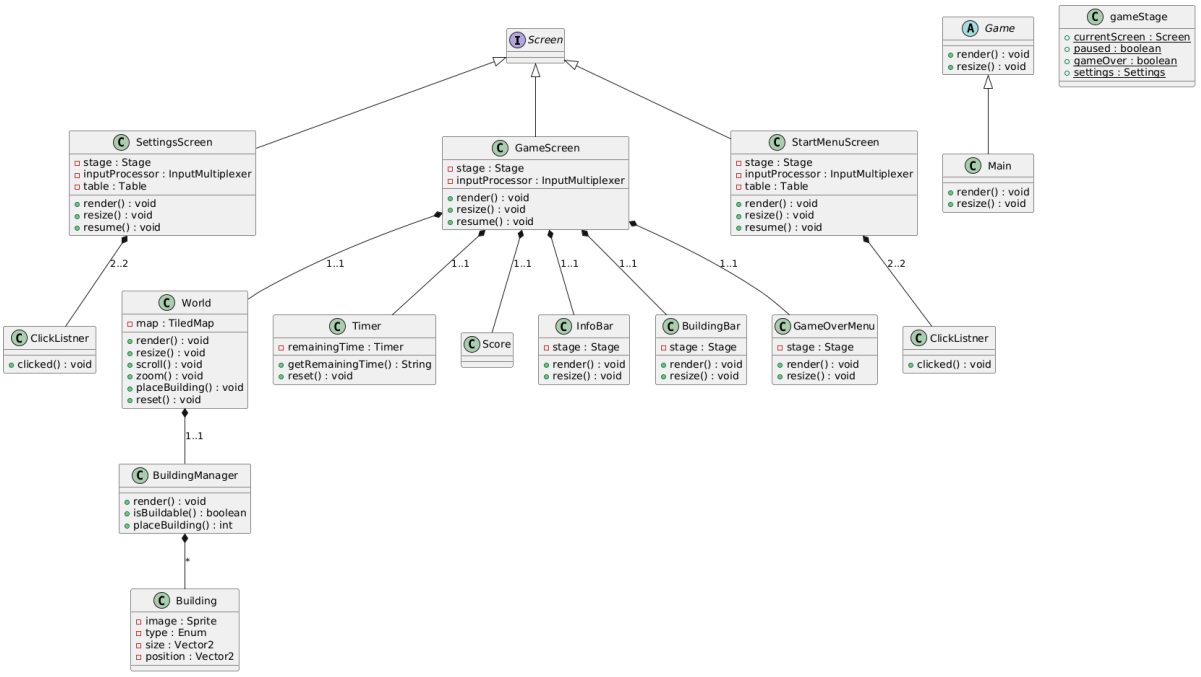

Diagram 6

Iteration 1

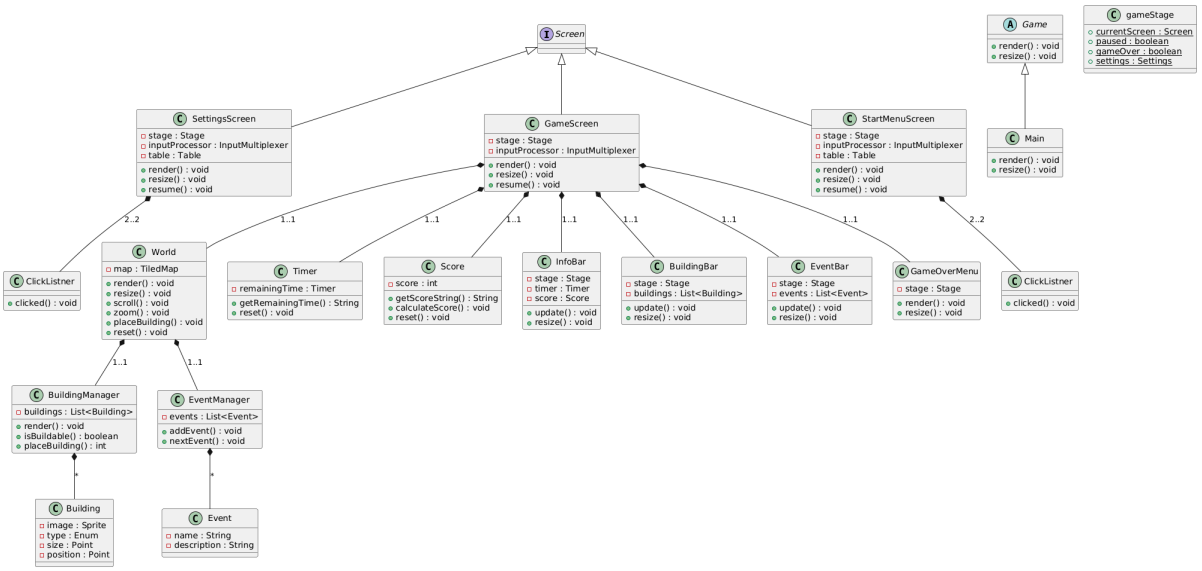

Diagram 7

In the first iteration of our new architecture we added the eventManager branch. This was needed so we could implement the requirements: UR_EVENTS and FR_EVENT_TYPES into the game. To be able to create and manage events eventManager would need some components from world and so we added it as a branch off world instead of as a new tree. We also added score and eventBar to the gameScreen as these would now be needed to display information about events to the player and to implement the requirement FR_LEADERBOARD.

Iteration 2

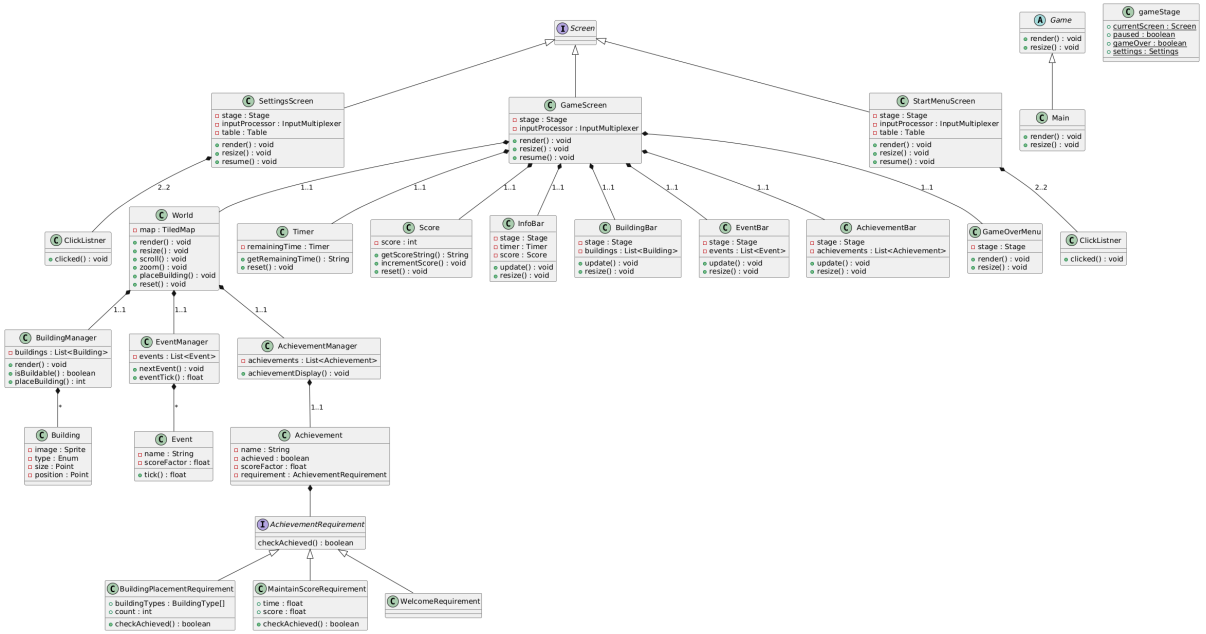

Diagram 8

For our second iteration we added achievementManager as another branch of world so it could use the components needed to allow management of achievements. This was so we could start implementing the requirement UR_ACHIEVEMENTS. The achievementManagement then has a branch to achievement and then to achievementRequirement interface. This was so we could easily create new achievements with different requirements and remove duplication.

We also updated our UI architecture by adding achievementBar to the game screen so we could send achievement notifications to players.

We then updated our previous classes to reflect the logic of our code. We mainly focused on the logic of our scoring system. Scores can be changed by three actions, gaining an achievement, events and placing a building. We made sure our architecture would reflect this. For example events can decrease the score every tick or just by a single set value at the start of the event. This is reflected by our methods and attributes in our EventManager and Event class. We then planned how achievements would change the score. We decided they would increase the score by a set value and so we added this as an attribute called scoreFactor to our Achievement class. Finally when placing a building the score would change depending on the variety of buildings that had been placed and how densely surrounded each building is.

Iteration 3

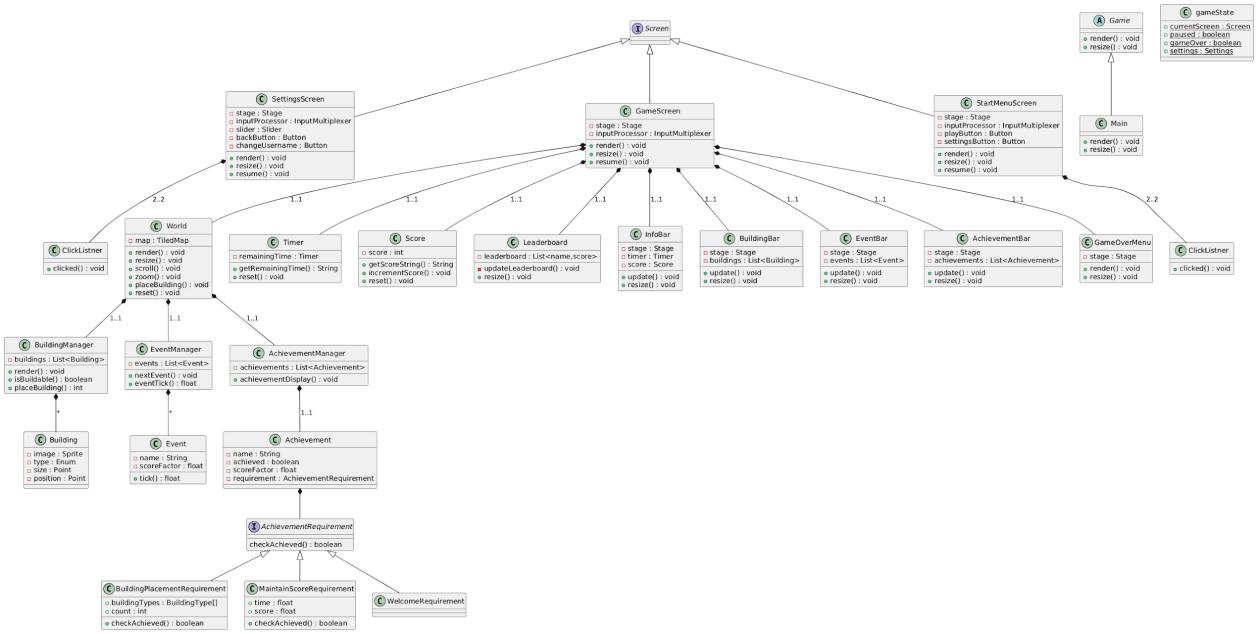

Diagram 9

For our third iteration we looked at one of our final requirements, UR_LEADERBOARD, FR_LEADBOARD. We knew this would be displayed to our user so we added the leaderboard class as a new branch to GameScreen. This contains the methods and attributes we thought would be necessary for its implementation.

We also made a couple changes to our UI. We added the ability for the user to set their username as this would be needed to identify scores set by different individuals. We added a back button to our SettingsScreen class as this screen layout had changed. We added these as attributes to our SettingsScreen class. We updated our StartMenuScreen class to also better reflect the buttons and attributes it contained.

Finally we made a few updates to the methods and attributes in our other classes to make our architecture align with our code. These were changes that occurred during the implementation of our last iteration.

Iteration 4

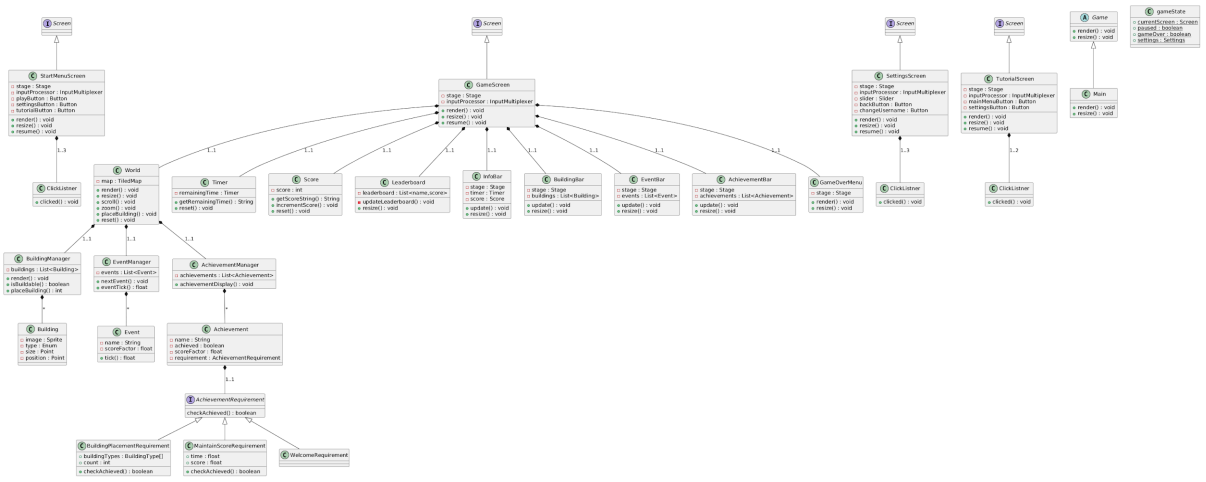

Diagram 10

For our fourth and final iteration we wanted our architecture to show a clearer representation of our screens than before. To make it easier to read we separated each screen into its own tree, but they all inherit from the same interface called Screen. We then updated the contents of each screen to reflect our code and what would be needed for our final implementation. For example we planned to add a tutorial screen. We created it as a new tree and gave it the buttons it would need for navigation. We also made sure screens were accessible from different points by adding the necessary buttons to each individual screen.

-

Behavioural/State Diagrams

Game State Diagram

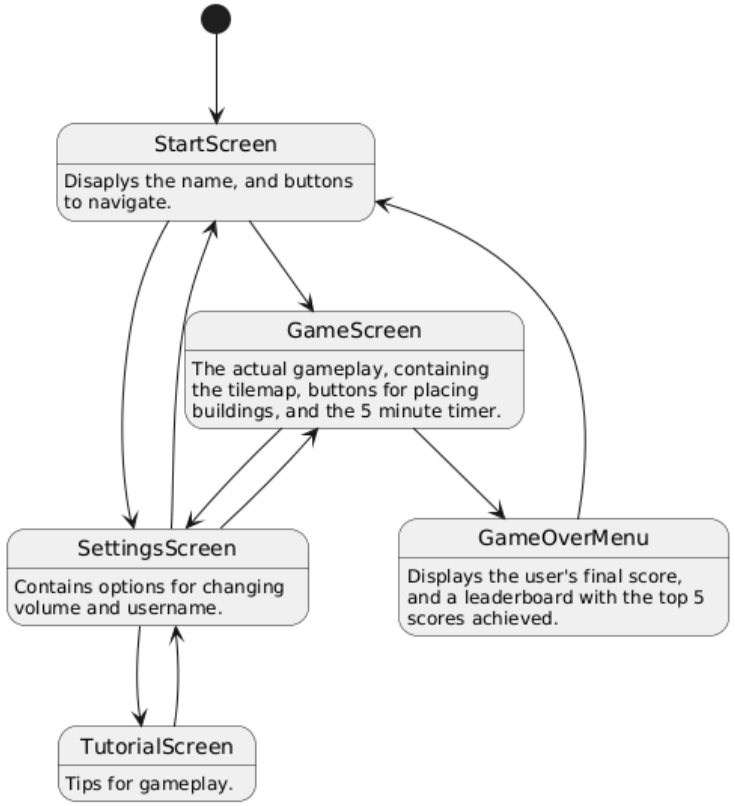

Diagram 11

To help with the creation of our final iteration we created the above state diagram to make sure all screens were accessible and the game would have a logical flow.

Events

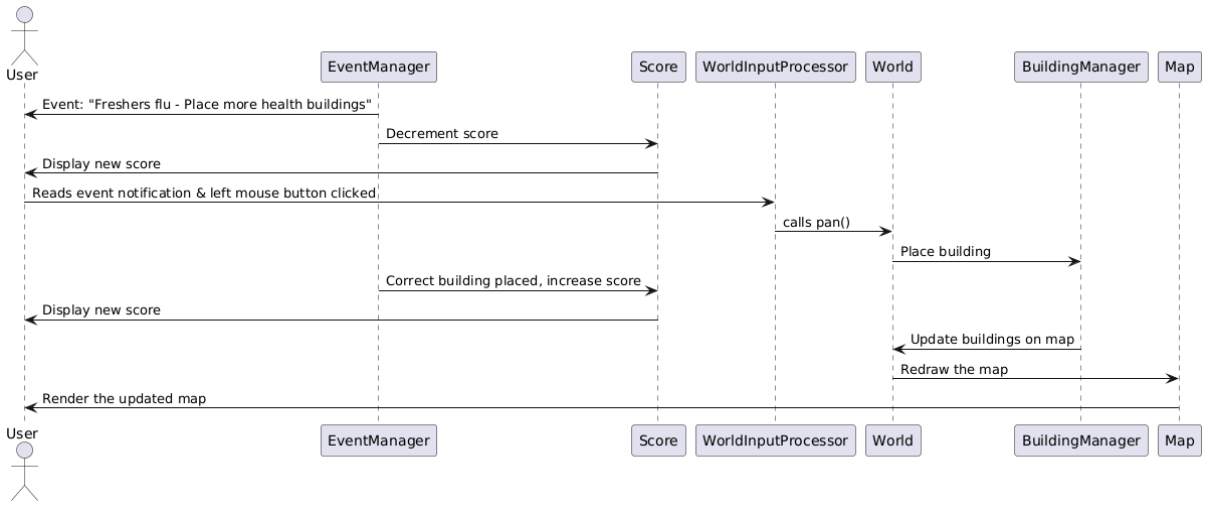

Diagram 12

Achievements

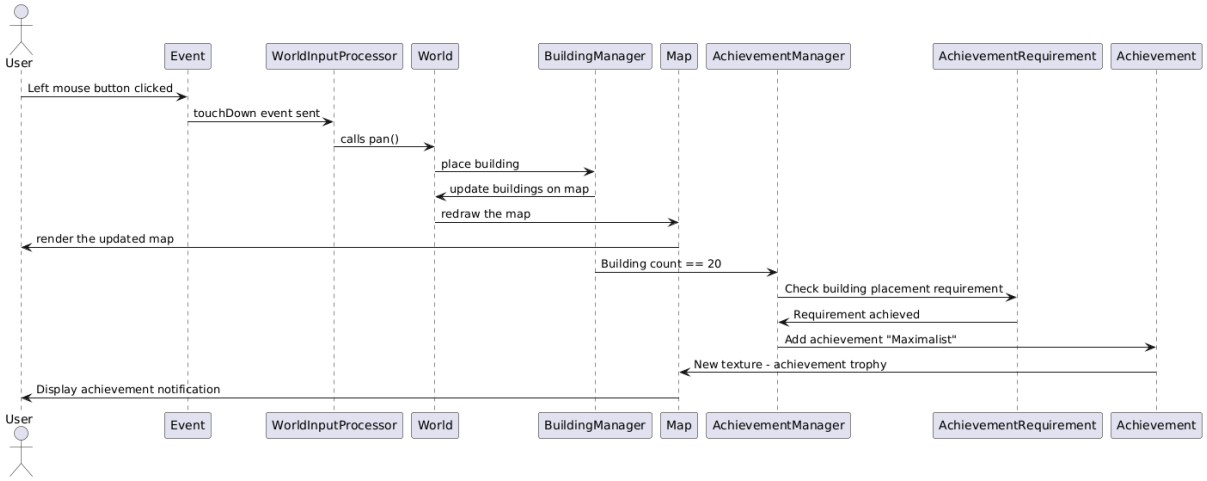

Diagram 13

We created two behavioural diagrams for events and achievements. This was so we could plan how user actions would affect gameplay and how this would relate to our architecture. We created diagram 12 to help us with our first architecture iteration and diagram 13 to help us with our second iteration. We first thought about one type of event and then worked through what classes would interact with each other as well as when the user would receive new information. We then did the same with achievements. For example when a user carries out an action we would check if it matches with one of our achievement requirements, if it does then the user would be awarded and notified of the achievement.