

Architecture

ENG1 Team 9

1. Jacob Dicken
2. Bertie Cartwright
3. William Croft
4. James Dovener
5. Henry Chan

This document outlines the architecture phase of the development process of our game, showing the thought process behind the design decisions (that were based on our user and system requirements), and the relevant tools that we used to create these designs.

We have used the Unified Modeling Language (UML) to create a blueprint of our game. We have drawn class diagrams to represent the structure of the game by showing its classes, attributes, methods, and the relationships between the classes. We have drawn a state diagram to model the behaviour of the system by representing its states and the events that trigger transitions between those states. We are also including a sequence diagram to represent the behaviour of the system by showing how objects interact over time.

We have mainly used PlantUML (extension in VSCode and Google Docs) to create our diagrams. We have also used the [PlantUML Web Server](#) to create some of these diagrams.

The game makes use of a simple object-oriented architecture, with one top-level class, Main, that has responsibility for managing the whole system, but delegates jobs to other classes that have increasingly less responsibility. For example, MapScreen handles all the gameplay, but delegates building selection to the BuildingMenu class. This use of decomposition allows us to write sections of code that only handle one thing, which simplifies the implementation and should allow it to accomplish that thing well. We feel that this architectural style is suitable for a small low-risk project such as this game because it is a very simple architectural style with a low overhead, which is ideal for our small team and time frame. This way we can spend more time implementing features rather than developing internal layers or APIs.

Class Diagrams:

Grid Position

Addresses the following requirements: **UR_BUILDING_PLACEMENT**, **FR_BUILDING_PLACEMENT**

C	GridPosition
□	x : int
□	y : int
●	GridPosition(int x, int y)

In the beginning, we designed the GridPosition class to be a positional object that holds the x-coordinates and y-coordinates of a certain position.

Grid Area

Addresses the following requirements: **UR_BUILDING_PLACEMENT**, **FR_BUILDING_PLACEMENT**

C GridArea	
□ origin : GridPosition	
□ width : int	
□ height : int	
□ xRange : IntRange	
□ yRange : IntRange	
● GridArea(GridPosition origin, int width, int height)	
● GridArea(int x, int y, int width, int height)	
● getX() : int	
● getY() : int	
● contains(GridPosition other) : boolean	
● contains(GridArea other) : boolean	
● getXRange() : IntRange	
● getYRange() : IntRange	
● calculateXRange() : IntRange	
● calculateYRange() : IntRange	

The grid area class is designed to hold the information about the area of the map. The initial design is linked on the website (*GridArea1.png*). Getter methods are used to retrieve the origin coordinates of the object and the contains() methods check whether a specified position is within the GridArea map. To get the total range area of the map, four range methods are used.

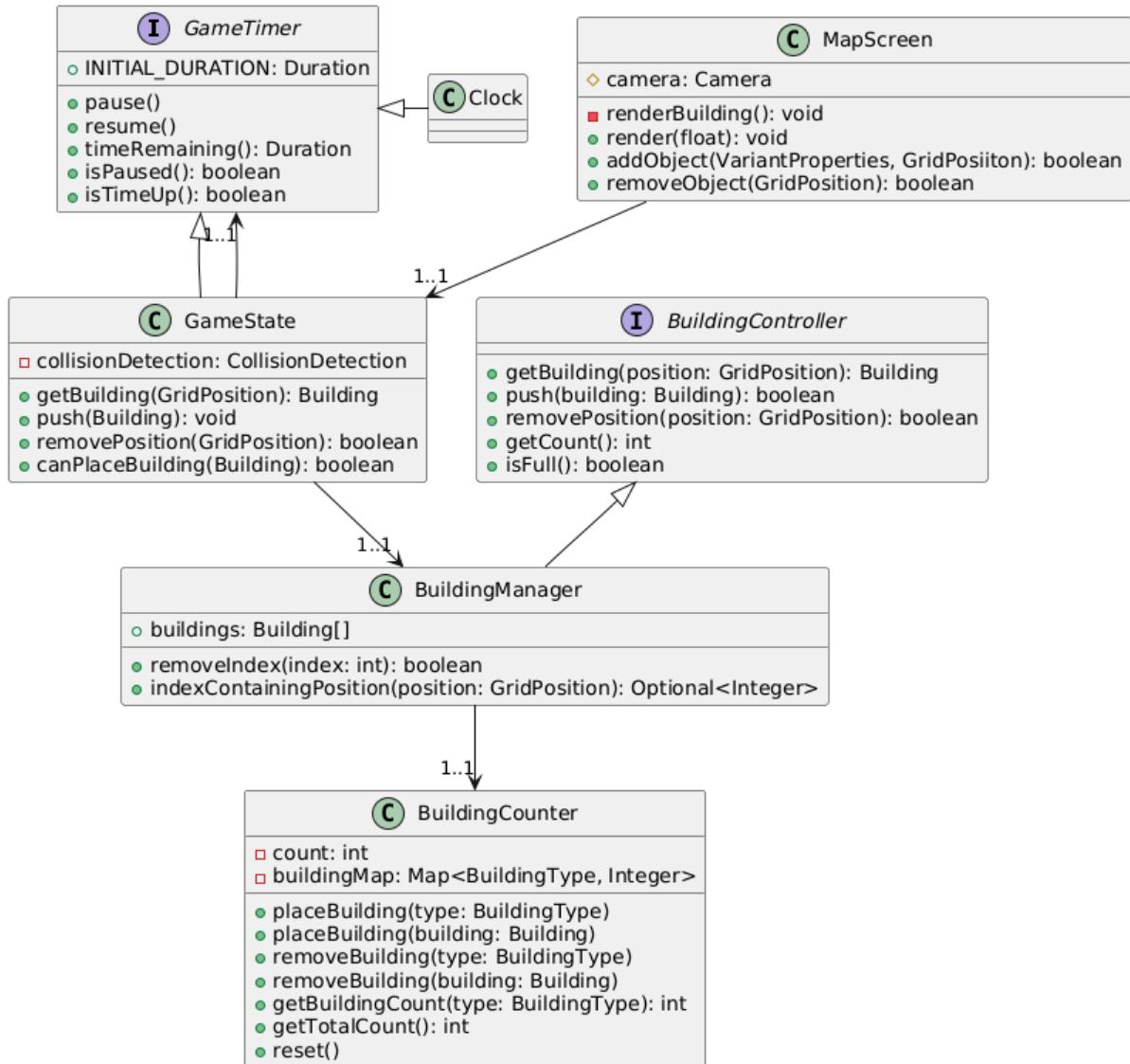
Collision Detection

Addresses the following requirements: **UR_BUILDING_PLACEMENT**, **FR_BUILDING_PLACEMENT**

The purpose of the collision detection object is to do the calculations needed to check if the player can place a building on the specified tile and the associated operations to place and remove a building.

C CollisionDetection	
□ buildingGrid : boolean	
● CollisionDetection(int xSize, int ySize)	
● canPlaceBuilding(int x, int y, int width, int height) : boolean	
● placeBuilding(int x, int y, int width, int height) : boolean	
● removeBuilding(int x, int y, int width, int height) : void	

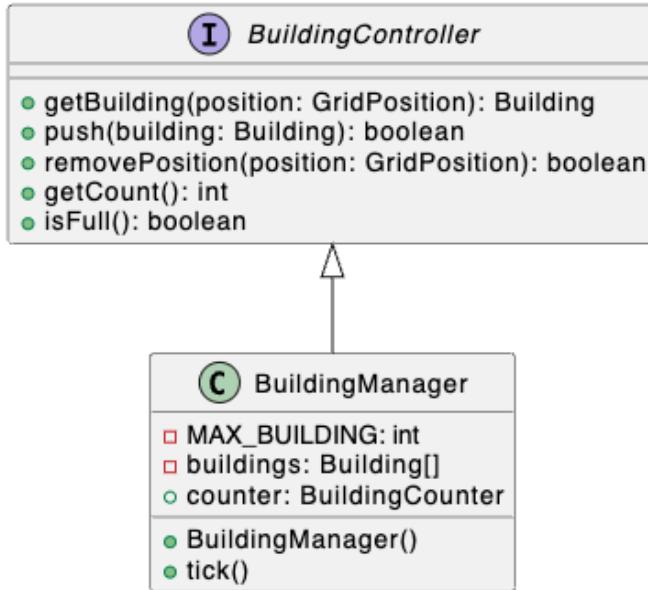
Structure Before New Requirements



The final architecture structure before the extra requirements is shown above. We included this in order to justify the quantity of changes that were necessary to make in order to meet the new requirements. For example, in the above structure, the `render(float)` method in `MapScreen` handles user interface, building placement as well as rendering the world, all the buildings and the UI, all without delegating work to other methods or classes. It was necessary to rework most of the user interface and gameplay code in order to make it more extensible.

BuildingManager

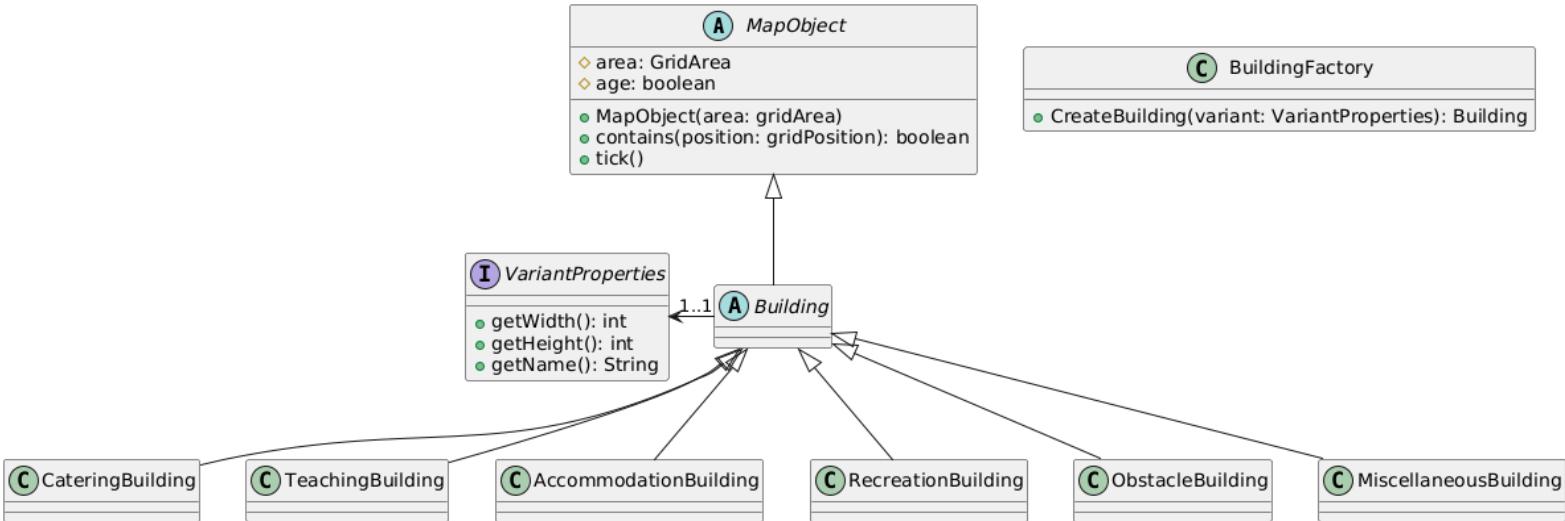
Requirements Addressed: FR_BUILDING_COUNTER, FR_BUILDING_PLACEMENT



BuildingManager holds all buildings placed in the world and handles adding/removing buildings. It also holds a reference to the BuildingCounter which tracks the number of each building placed and limits the number of buildings to a maximum. This system did not change significantly throughout development as the requirements for buildings did not change between both assessments.

Buildings

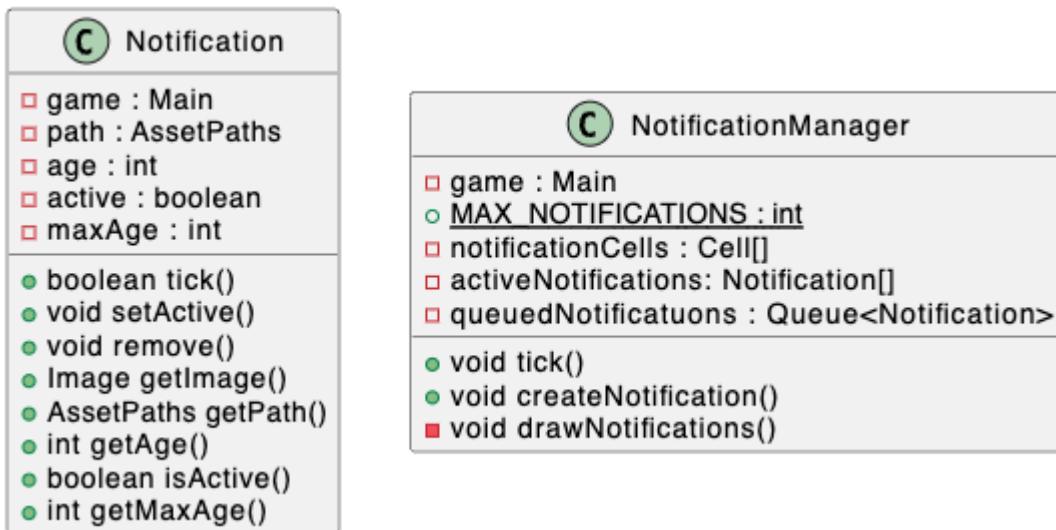
Requirements Addressed: FR_BUILDING_MANAGER, FR_BUILDING_PLACEMENT



We made use of an inheritance hierarchy for buildings which allowed us to reuse the logic code in MapObject and Building for all the building types, while maintaining the ability to add type-specific code to each building type if required. Most of the building-specific code lies in the BuildingVariant, which stores the buildings' size and AssetPath. Building factory manages all the constructors for each building type, hiding the complexity of the inheritance hierarchy by allowing us to create Building objects by simply supplying a building variant.

Notifications

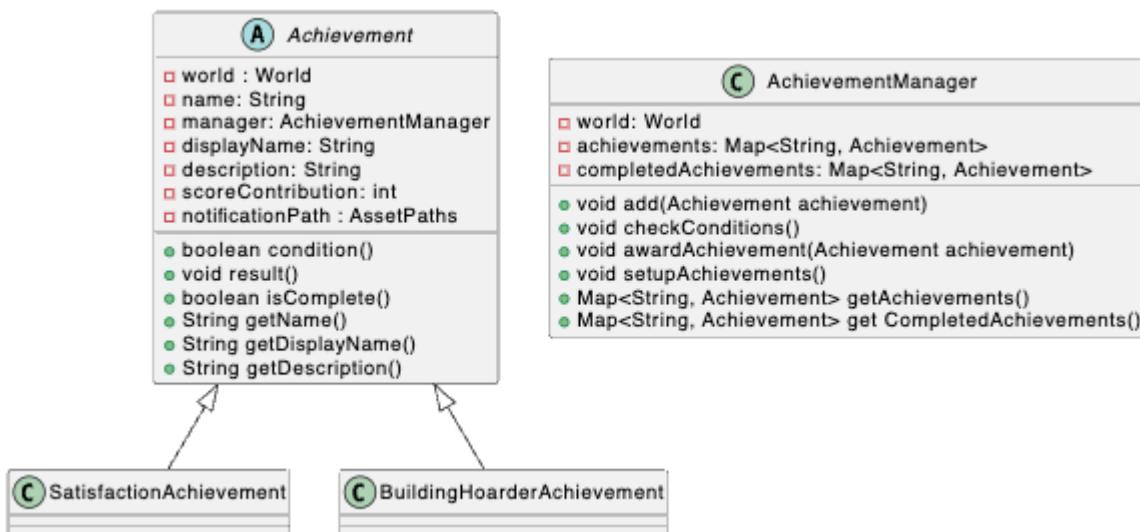
Requirements Addressed: FR_USER_INTERFACE, FR_INTERACTIVE_ELEMENTS



Notifications are only created directly within the NotificationManager, which handles displaying multiple notifications on the screen and queueing any that cannot fit. Systems such as Achievements and Events may create a notification with a call to the NotificationManager createNotification() method. Initially, notifications were created directly by these systems and drawn onto the screen, however the need to support multiple notifications from different sources meant the NotificationManager was needed.

Achievements

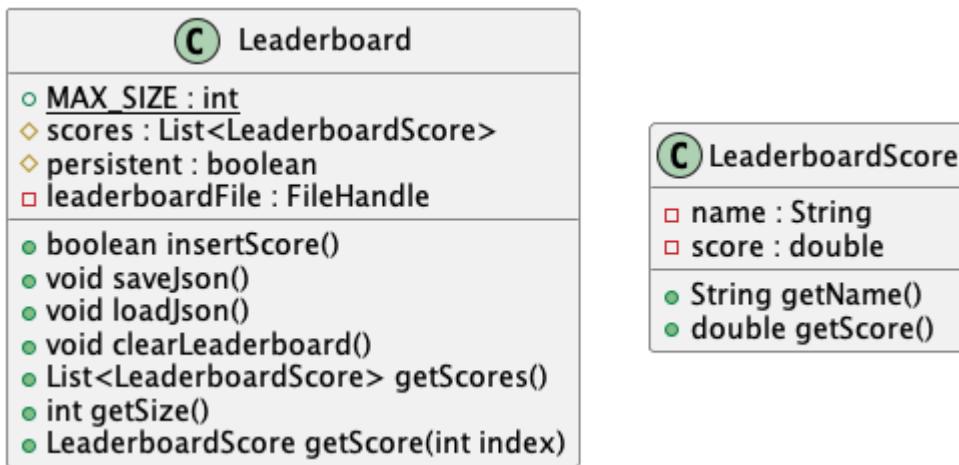
Requirements Addressed: FR_ACHIEVEMENT_EARN, FR_ACHIEVEMENT_CRITERIA, FR_ACHIEVEMENT_REWARD, FR_ACHIEVEMENT_NOTIFICATION



The Achievements system is designed to make adding more achievements very simple, minimising code reuse by placing as much functionality as possible in the Achievement and AchievementManager classes. All achievements extend the Achievement class which provides the base behaviour all achievements have in common, such as displaying a notification. The AchievementManager holds all the achievements to use in the game. When checkConditions() is called every game tick, the manager detects if any achievements have been earned, and applies the respective score bonus. Initially, the Achievements system was designed not to use an AchievementManager, however this led to significant code duplication in the render loop when checking for each condition.

Leaderboard

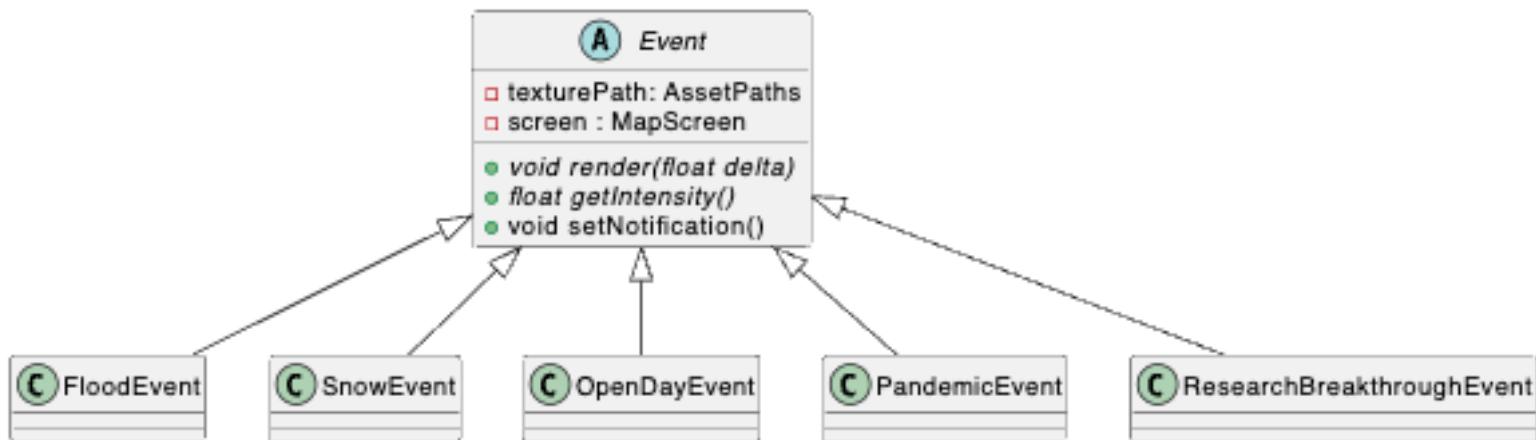
Requirements Addressed: FR_LEADERBOARD, FR_ADD_TO_BOARD



The Leaderboard records the top 5 recent scores and makes use of a JSON file to remain persistent between replays of the game. The Leaderboard has not changed significantly since its' design as the requirements around the Leaderboard system have remained the same.

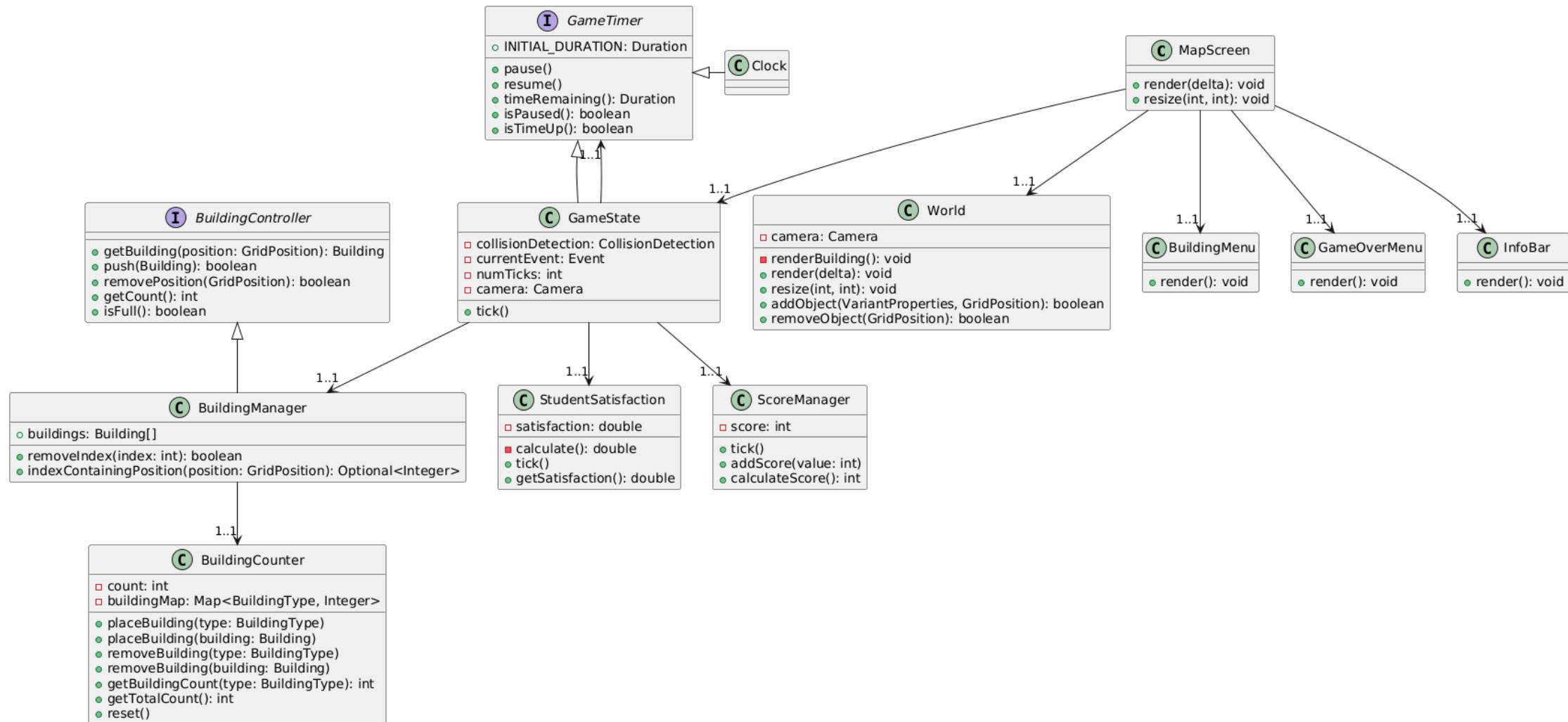
Events

Requirements Addressed: UR_EVENTS, FR_EVENT_TYPES, FR_EVENT_FREQUENCY



Each event is a subclass of the abstract Event class, providing custom behaviour by implementing the render method. The Event class handles common logic for all events such as providing a Notification.

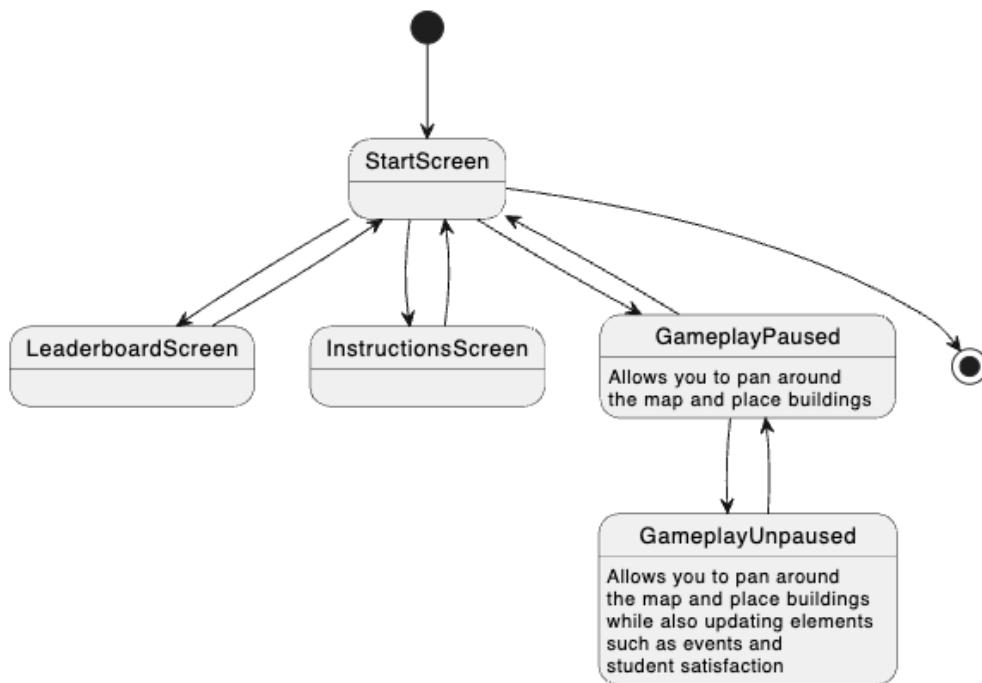
Structure After New Requirements



As shown in the diagram above, the architecture from before the new requirements has been divided up into smaller classes, which was essential to be able to extend the project. This allowed us to easily add the StudentSatisfaction and ScoreManager classes to meet the new requirements.

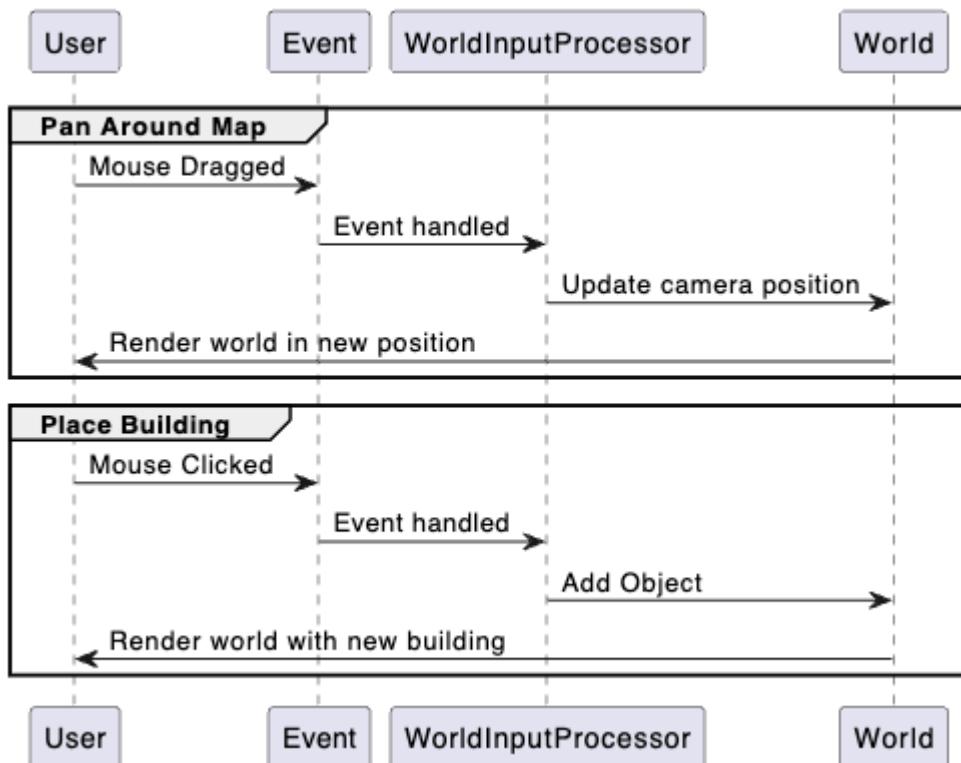
Behavioural Diagrams:

State Diagram

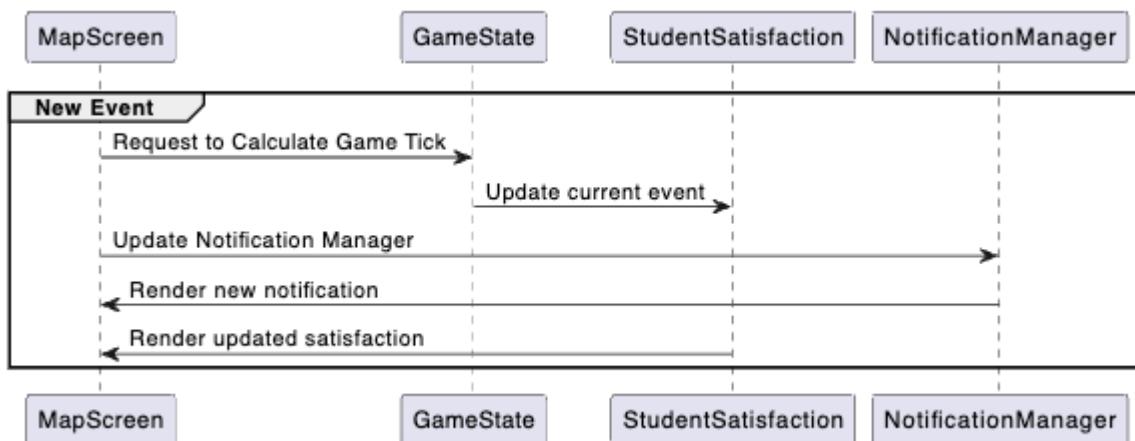


The game starts initially on the StartScreen, with options to view the Leaderboard, Instructions, play the game or quit. When playing the game, the user may pause which stops the in game time from progressing. When the timer runs out, the game is forcefully paused and the user has the option to return to the StartScreen.

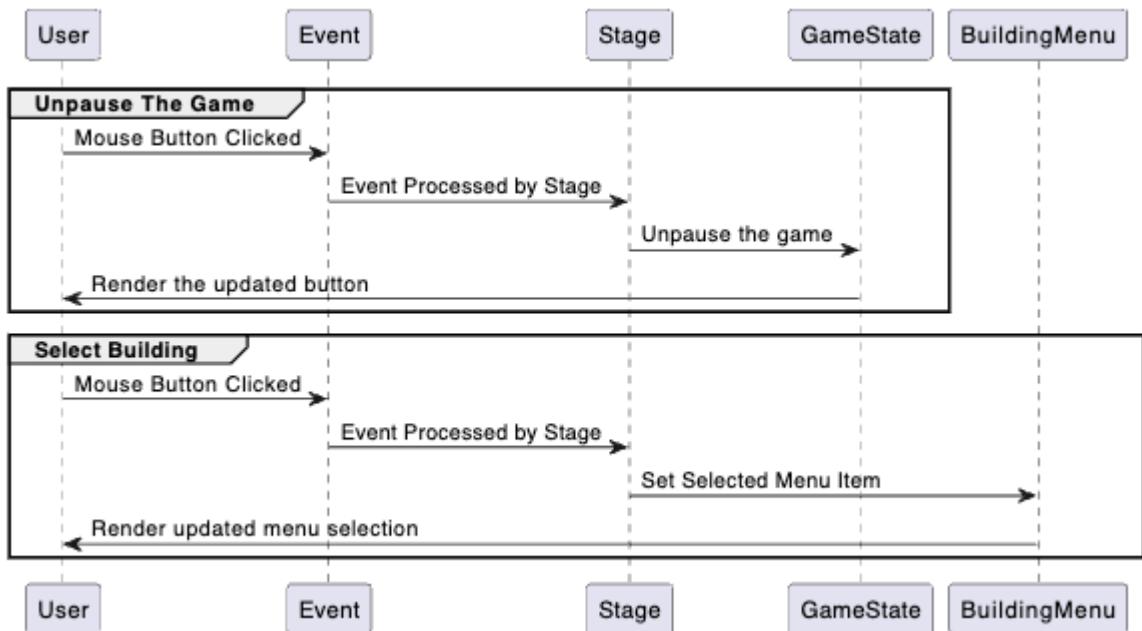
Sequence Diagrams



This diagram shows the two main ways the user interacts with the game - by moving the camera or placing a building. In both cases, events are triggered and handled by the WorldInputProcessor.



This diagram shows the process of a new event starting. A notification is delivered and the student satisfaction modifier applied.



This diagram shows the main interactions the user makes with the UI. In each case, the interaction creates an event which is handled by the Stage the UI element is attached to.