Architecture

Group Number: **Cohort 1, Group 11**
Group Name: **Y111 Studios**

Group Members:

1. James Hutchinson
2. Somto Igweonu
3. Robert Kisloski
4. Sam Knight
5. Kenneth Kok
6. Ashish Kumar

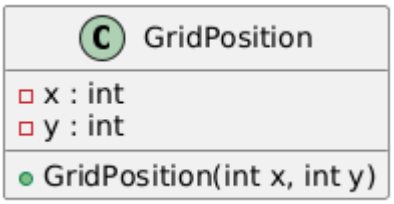
This document outlines the architecture phase of the development process of our game, showing the thought process behind the design decisions (that were based on our user and system requirements), and the relevant tools that we used to create these designs.

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), alongside the graph visualisation software Graphviz, to create our diagrams. We have also used the PlantUML Web Server and the graph drawing software application Draw.io to create some of these diagrams.

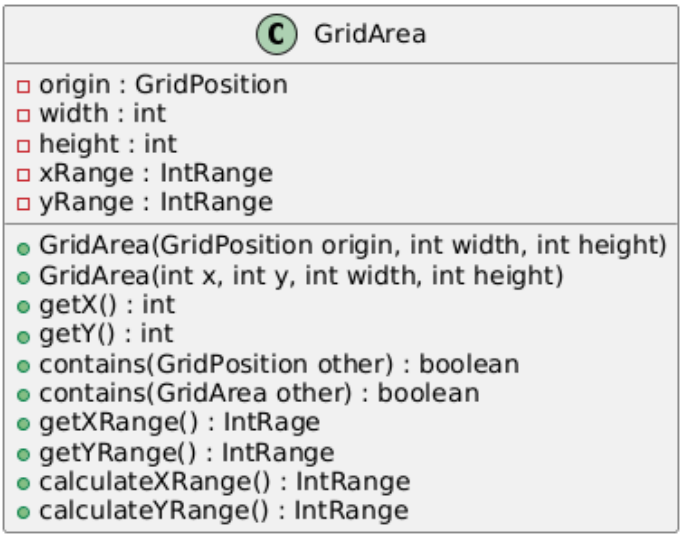
Class Diagrams:

Grid Position
Addresses the following requirements: **UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**



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.

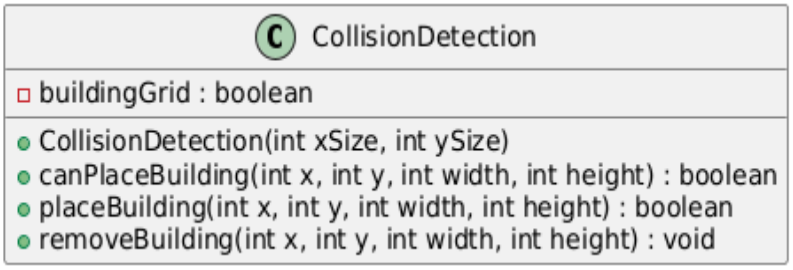
Grid Area
Addresses the following requirements: **UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**



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.

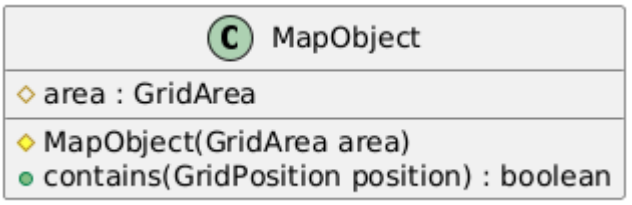
Collision Detection
Addresses the following requirements: **UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**

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.



Map Object
Addresses the following requirements: **UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**

The map object is an abstract class that is used to represent a map object within the game and is a superclass of all objects. There is also a check for whether a position is on a valid area of the map.



Building Variants
Addresses the following requirements: **UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**

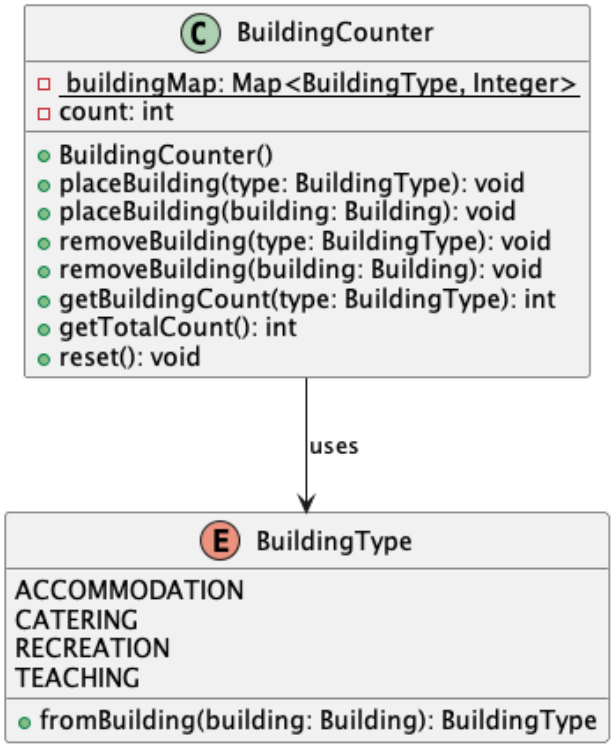
To implement the buildings and their respective variants, we utilised an interface that is then used to implement the four different types of buildings. Each building variant has getter methods for their width and height as well as the variant class for internal operations. (Diagram linked on website: *BuildingVariants.png*)

Buildings
Addresses the following requirements: **UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**

To implement the different types of building, we used an abstract class *Building*. By extending Building, each of the buildings store the grid positions and their respective width and height. (Diagram linked on website: *Buildings.png*)

Building Counter

Addresses the following requirements: **UR_BUILDING_COUNTER; UR_BUILDING_PLACEMENT; FR_PLACE_BUILDING**



Initially, we defined the BuildingCounter class with a nested enum *BuildingType*, which allowed us to track the counts of different types of buildings using a dictionary element (*buildingMap*). We track the total number of buildings by the *count* attribute. The primary functions included initialising the counter, placing and removing buildings, and retrieving the counts. *(BuildingCounter1.png on the website)*

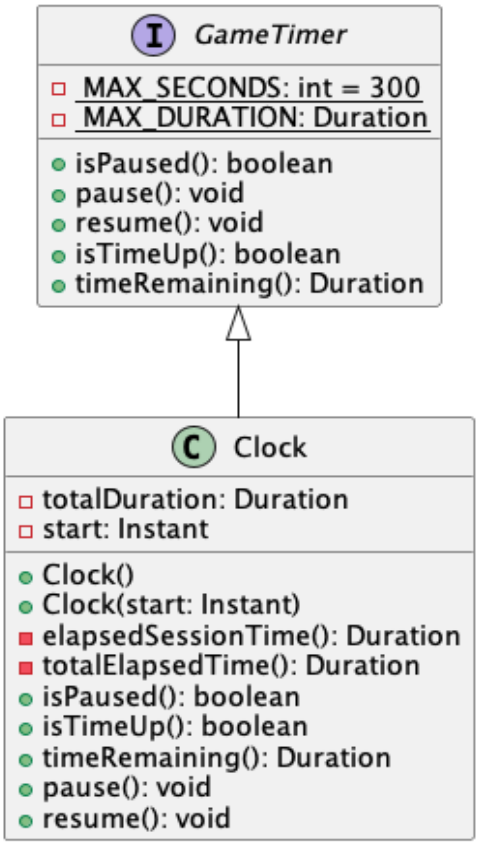
As we made progress, we decided to refactor the *BuildingType* enum outside the BuildingCounter class and introduce an additional method to determine a building's type based on its instance. The BuildingCounter class now has overloaded *placeBuilding* & *removeBuilding* methods that accept a Building object, which makes it easier to manage buildings.

Clock

Addresses the following requirements: **UR_GAME_CLOCK; UR_PAUSE_GAME; FR_GAME_PAUSE; FR_GAME_END**

Initially, we introduced the Clock class with attributes to track the start time, pause state, and the total elapsed duration. It included methods for checking the elapsed time and the time remaining, and functionalities to pause and resume the clock. This version established a basic structure for tracking time with a maximum duration of 5 minutes. *(Clock1.png on the website)*

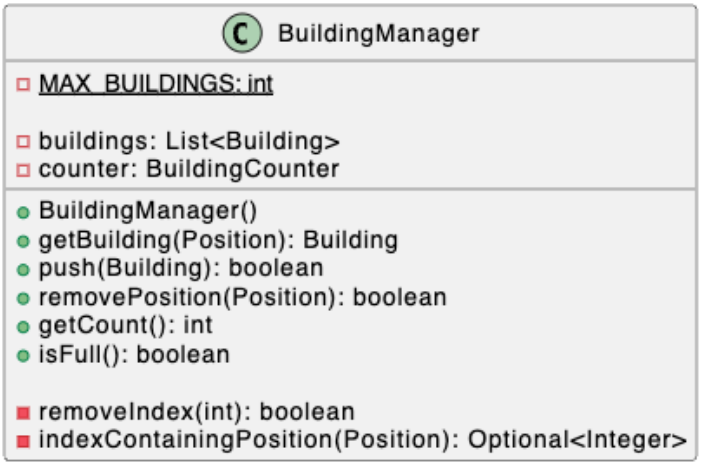
As we made progress, we decided to refactor the Clock class to implement an interface (*GameTimer*) that allows for a standardised approach to timer operations, thus facilitating integration with other components.



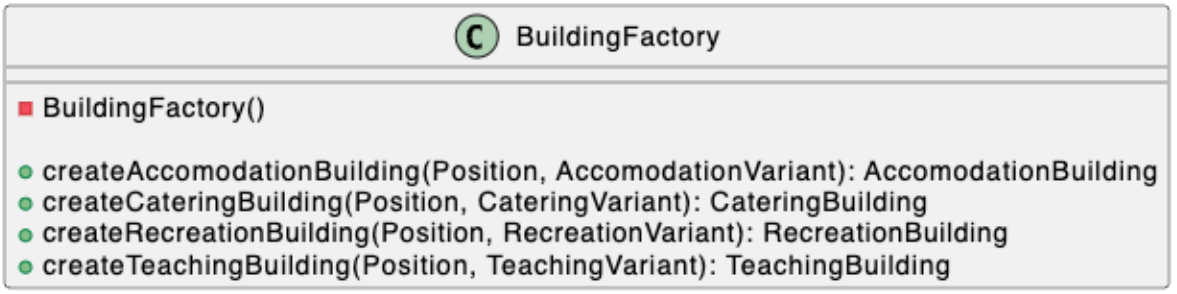
Building Manager
Addresses the following requirements: **UR_BUILDING_MANAGER; UR_BUILDING_PLACEMENT; FR_BUILDING_MANAGER**

Initially, the building manager had a public interface that allowed for a maximum number of buildings to be added to the game. This includes validation to ensure a valid state is maintained.

As progress was made the publicly available methods were extracted into an interface for ease of use. *(BuildingManager2.png on the website)*



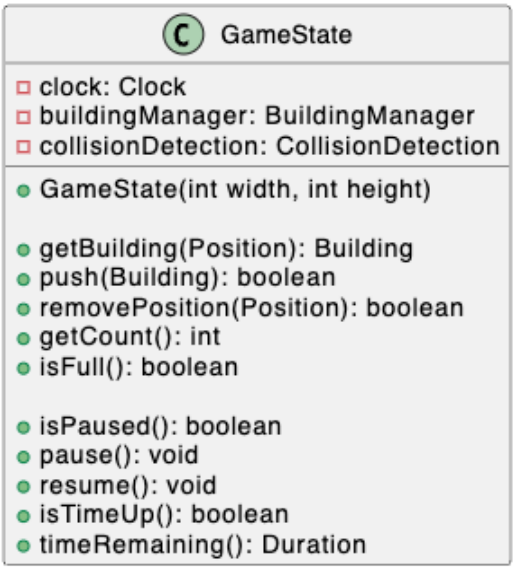
Building Factory
Addresses the following requirements: **UR_BUILDING_PLACEMENT**



The BuildingFactory is a static class that is used to facilitate the construction of Buildings within a fixed way using the attached fields of each building type variant. This provides a simple interface that can be easily presented to the user. This was refactored later to use method overloading as can be seen in (*BuildingFactory2.png* on the website).

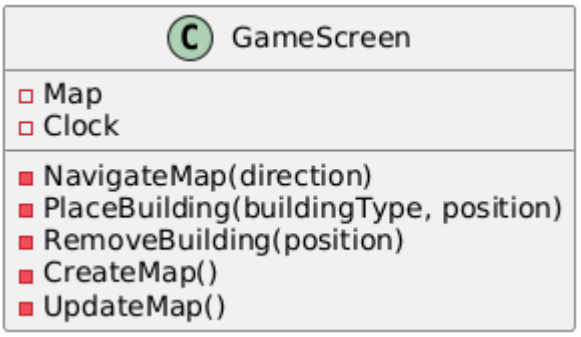
Game State
Addresses the following requirements: **UR_BUILDING_PLACEMENT; UR_GAME_CLOCK; UR_PAUSE_GAME; FR_PLACE_BUILDINGS; FR_GAME_PAUSE; FR_GAME_END**

The game state object ties all the previously named components together and manages the sum game state and allows for modification of it in one simplified interface. As progress was made, this object was modified to implement the interfaces added to all of its internal fields, ensuring the correct fields were available. (*GameState2.png* on the website)



GameScreen

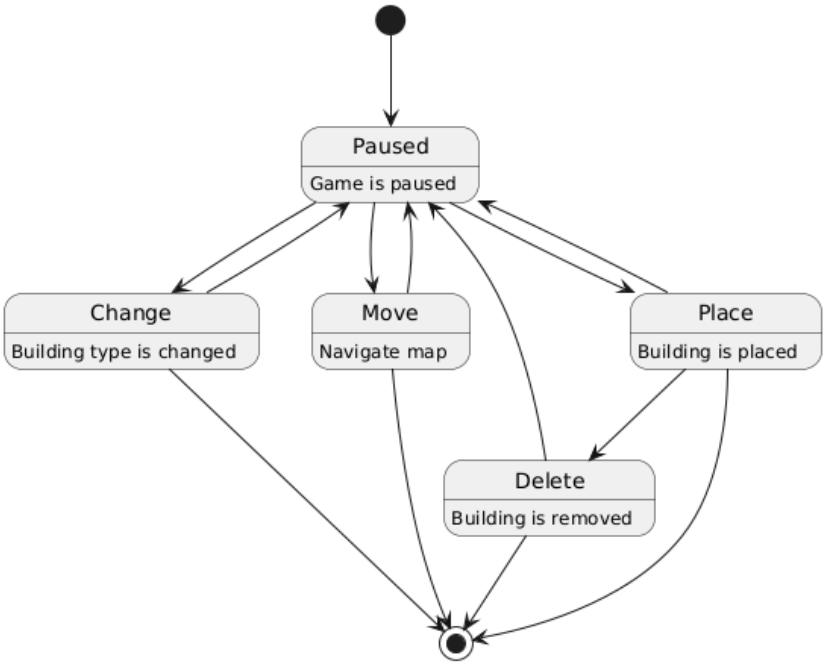
Addresses the following requirements: **UR_STATIC_MAP; UR_GAME_PERSPECTIVE; UR_BUILDING_PLACEMENT; FR_GAME_CONTROL**



Initially, we started by creating the idea of an idle state (*StartScreen*) however this was deemed an unnecessary addition as it was not a requirement. We decided to just have a game state (*GameScreen*) which would be displayed when the game is opened. The GameScreen would need access to an instance of the map to enact the user's inputs on and some clock object to display the time. We then added methods to allow the user to manipulate the game to meet the requirements such as moving around the map, placing buildings, and removing buildings. Finally, we needed to create the map and then be able to update it to show the effects of the user's inputs.

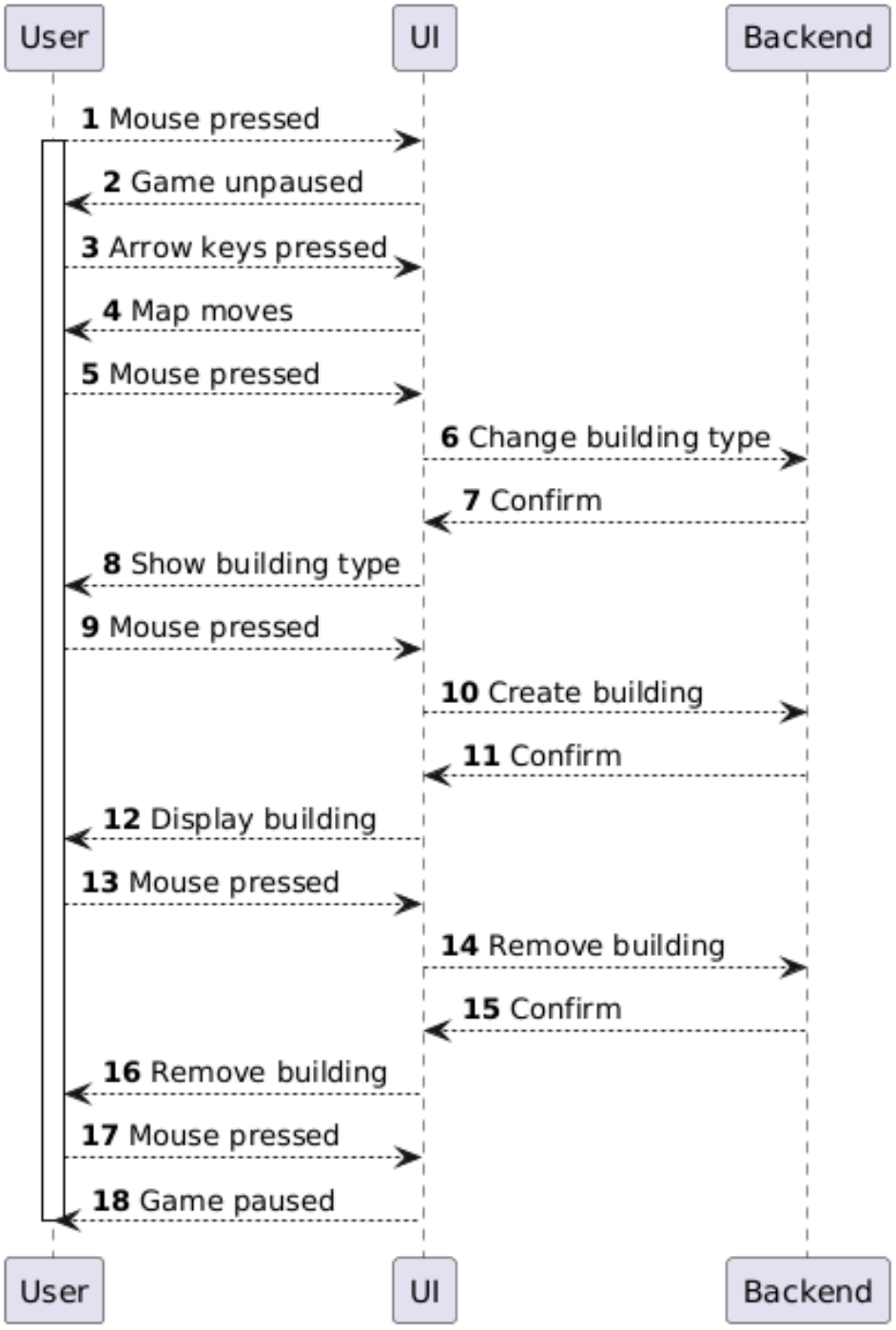
Behavioural Diagrams:

State Diagram



Here we describe how the game starts paused and then once unpaused, the user can navigate the map, change the type of building being placed, place a building, and, once a building has been placed, can remove a building. The game can be paused from any state.

Sequence Diagram



This diagram outlines a typical user interaction with the game. Each use of ‘mouse pressed’ will be on a different location on the screen hence why the result is different however the location hasn’t been specified here due to this being a high-level overview.