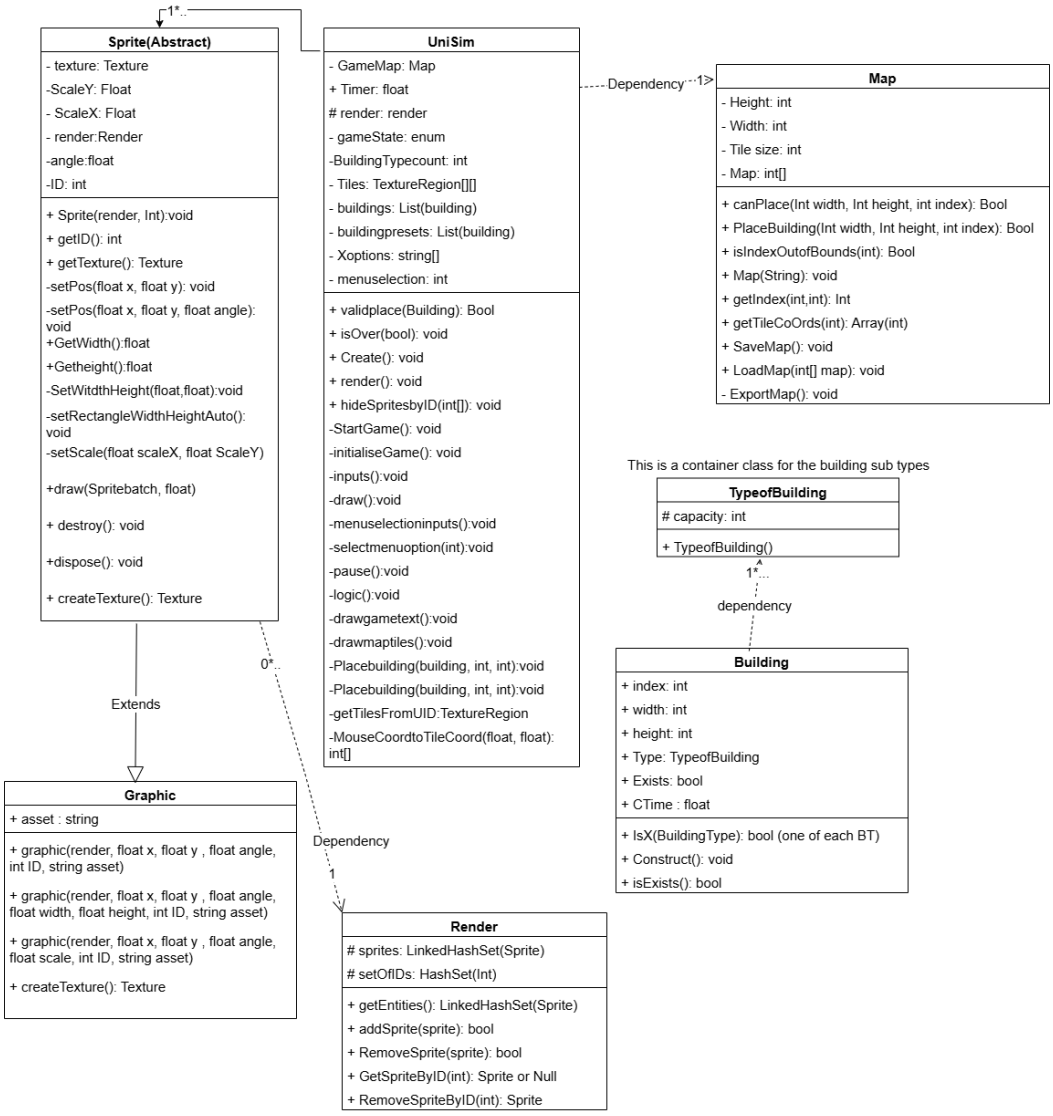
Architecture

Windows 007
Cohort 3 Team 7

Stephen Lavender
Sam Leach
Daron Lepejian
Ding Lim
Hasan Majid
Joshau Marshall-Law

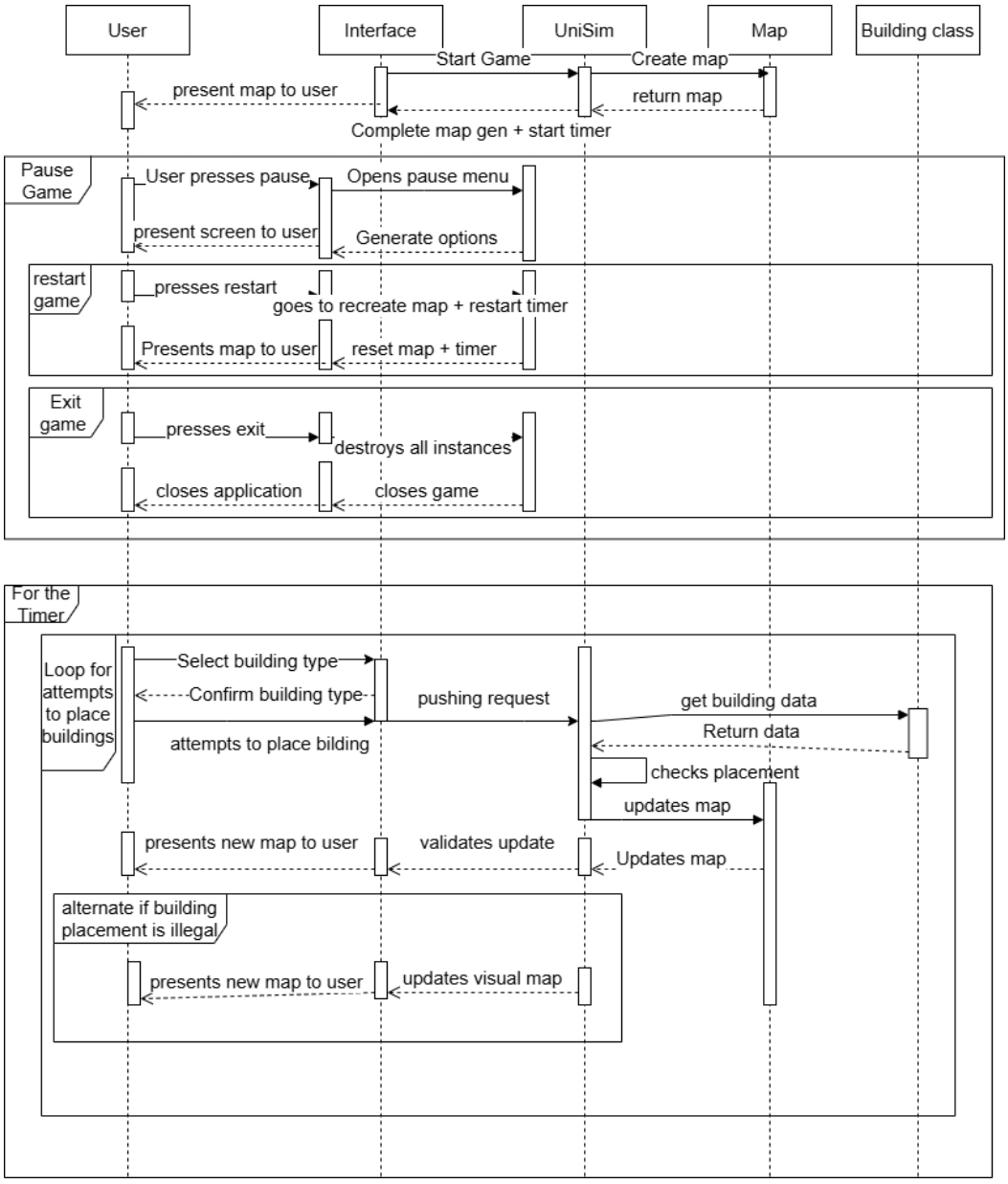
All of the below diagrams were created using draw.io and implemented using the UML language.

0.1 Class Diagram



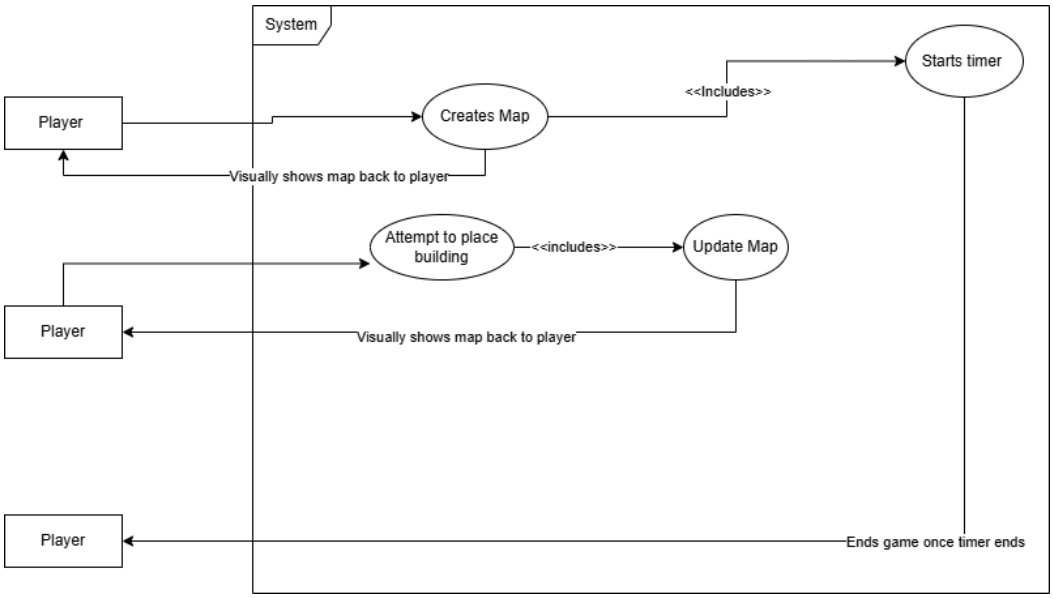
The structural diagram demonstrates the game architecture. The UniSim class is the central controller, coordinating with other classes to achieve object-oriented programs. The Map class manages the game grid and placement of buildings, while the Building class represents individual buildings that will be placed during the game. The Sprite class and Graphic class handle the visual elements with the Render class that manage the collections of sprites for real time updates to the game's UI.

0.2 User Sequence Diagram



This diagram served as the primary way of sequencing the game. The tags show a variety of different situations that should cover all use cases. The sequencing also allowed the implementation to go smoothly as it left a clear path as to the order in which classes should interact with one another.

0.3 Use Case Diagram



This is a basic version of a use case diagram dictating the general principle of game states and the actions the user can take. However, we decided that a user sequence diagram would be more suitable to our system due to the linear nature of the game, having a strict point by point chart allowed for easier implementation.

0.4 Justification of Architecture

When designing the architecture of our solution we first discussed the necessary features of the product. These were:

- Map Representation
- Building Class Structure

0.4.1 Map Representation

We decided on a 2D tile representation relatively quickly, this promted the representation of the map as a 2D array of images. Upon further examination it was clear that this would be very inefficent and memory intensive as images take up a lot of memory and comparing images is computationally intensive compared to comparing integers.
One idea was to replace the images with integers. This is done by assigning every tile a unique identifier (uid). This gives each tile (blank, grass, water, building) a unique represenation and it is very easy to draw a different image based on a tiles' uid.
However, when planning the algorithsm we realised that working with 2D arrays is rather clunky but one advantage of 2D arrays was that they work in a similar way to co-ordinates (`my2DArray[y][x]`). So we looked at how we could use a 1D array like a 2D array. We first need to be able to convert an 1D index into x and y co-ordinates. Through rough sketches and calculations we came up with this formula:

$$x = i \mod W$$
$$y = \lfloor i/W \rfloor$$

Where i is the index and W is the width of the map in tiles.
A similar thing can be done to go from x and y co-ordinates to a 1D index:

$$i = y * W + x$$

Where y, x is the x co-ordinate of the tile and W is the width of the map in tiles.
This provides us with the convenience of accessing a 2D array but the ease of altering a 1D array when adding buildings.

0.4.2 Building Class Structure

Our initial approach was to have a base class for a building that defined basic methods required for all buildings such as storing its index and width and height and then using inheritance to create a class for each of the building types.
At first this seems to be a very natural approach to take as inheritance and polymorphism are core concepts of object orientated programming however the difficulty arises when upgrading buildings and making them composite. Suddenly, it becomes very difficult to combine 2 different building types.
This led us to our second design where we used one of OOP's core principles - "favour composition over inheritance". By creating different classes for each type of building and assigning them as an attribute to a building class it is very easy to create composite buildings. This allows buildings to change types but also become one or more type of building when it is upgraded. This scales very

nicely when more building types are added as all you need to do is add a new attribute in the main building class for that type for building.

```
public class Building {
    private int index;

    private int width;
    private int height;

    private AccommodationBuilding accommodationBuilding;
    private CafeteriaBuilding cafeteriaBuilding;
    private CourseBuilding courseBuilding;
    private RecreationalBuilding recreationalBuilding;
}
```

0.5 User Requirements

- UR_PLACE_BUILDING: `UniSim.placeBuilding(Building building, int x, int y)` places a preset building `building` at the tile co-ordinates `x, y`.
- UR_TIME_LIMIT: `UniSim.timeElapsed` keeps track of the total time passed in a game and is updated via each frame by `timeElapsed += Gdx.graphics.getDeltaTime();` Once the five minutes is up the game is over and the user is presented with the game over screen.
- UR_GRAPHICS: Each building type is given a different tile, when these tiles are placed together they form detailed objects that describe their purpose to the user.
- UR_EMERGENCY_STOP: When the user presses the "P" key the game pauses and the timer stops through `UniSim.pauseGame()`. From here the user can resume at any point or exit the game and restart.
- UR_STABLE_PERFORMANCE: By using an object orientated approach it is easy to isolate and therefore fix any bugs providing a satisfying experience.
- UR_SCALABILITY: The game has a toggleable fullscreen mode (`UniSim.toggleFullscreen()`) which allow for versatile use, even on large displays.

0.6 System Requirements

- FR_POSITION_BUILDING: We use the co-ordinates of the mouse to determine the location of the building in the `UniSim.placeBuilding(Building building, int x, int y)` method which gives the user control over where to place the building.
- FR_BUILDING_TYPE: By separating the differnt building types into their own class and then using them as fields in the main building class it is very easy to create buildings of different types.
- FR_FRAME_RATE The frame rate has been kept high enough to provide a smooth experience by testing it after a new feature was added to make sure that it didn't have any unwanted side effects.

0.7 Non-Functional Requirements

- NFR_AVAILABILITY: The game is avaible at all times once the user has downloaded it from our website as our game does not connect to a server.
- NFR_SYSTEM_REQUIREMENTS: We have tested a program on a range of laptops and the performance is more than satisfactory.
- NFR_OPTIMISATION: Through thorough testing and awareness of our algorithms we were able to test for common boundaries and plan for them to reduce bugs.
- NFR_UI: The user interacts with the program primarily through the mouse which is a device that most users are comfortable working with. It is obvious the user when they have selected something and requires another input (left click) to execute the action reducing the likelihood of a mistake.
- NFR_PAUSE: The user is able to pause the game at any time by pressing the "P" key.
- NFR_FAST_RESPONSE: All actions the user takes are exectued within the next frame which at 60 frames per second is 17ms.