# Architecture 1

## Group 2 - Vikingz

Damian Boruch

Tommy Burholt

Elliott Bryce

James Butterfield

Ren Herring

Zhenggao Zhang

Sharlotte Koren

# Architecture

## Use Case Diagram

After our meeting with our client, we brainstormed and came up with how we were going to go about implementing our requirements. Firstly, we started off by creating a use case diagram of how we wanted the user to interact with our game. We knew from the start that we wanted the user to start on a main menu, with buttons which would lead them to different screens, such as a settings screen and a main game screen. Each circle of the diagram represents a use case, and we decided to put rectangles over circles that are part of the same screen for clarity.

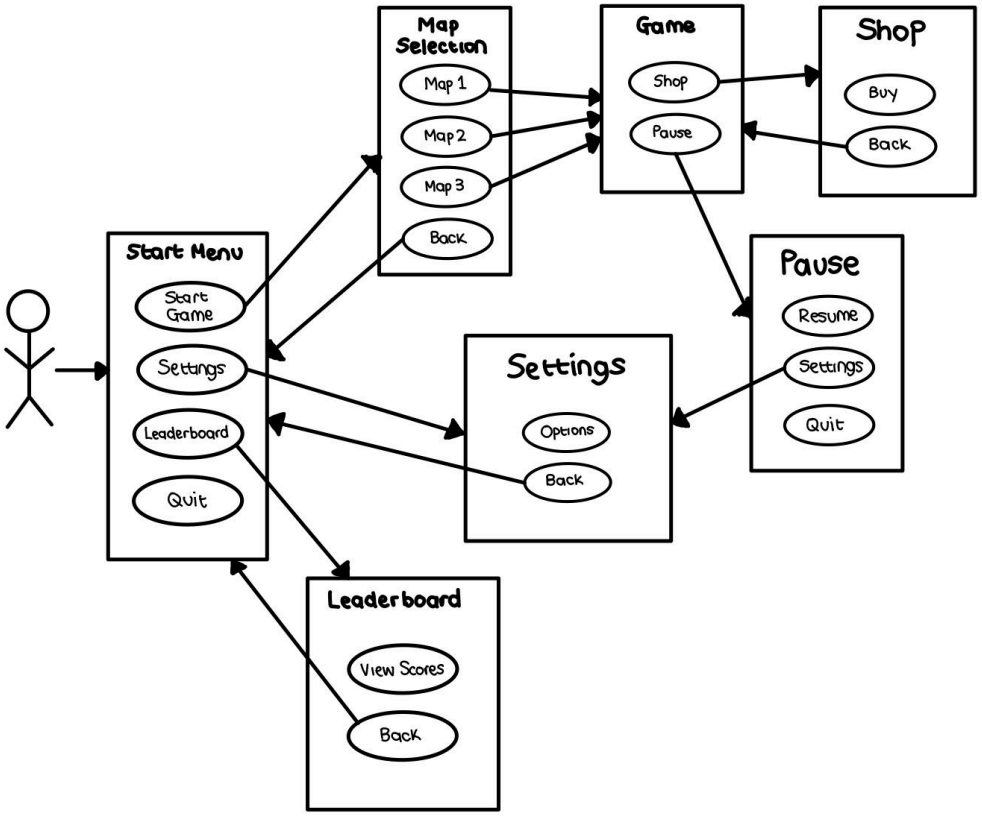


**Figure 1:** Use Case Diagram

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.

We decided to use a use-case 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.

This was useful because we related each use-case to our requirements referencing. For example, having the settings options which includes adjusting the volume will fulfil requirements FR_ACCESSIBILITY and FR_SOUND. Furthermore, the leaderboard represents FR_LEADERBOARD. 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. Also, we designed these screens in a way which is similar to other games that are already on the market. This would fulfil the requirement NFR_EASE_OF_USE as breaking down the game and avoiding clutter would help the user navigate the game seamlessly.

## Class Diagram

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. We created class diagrams to help us envision this using PlantUML. 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

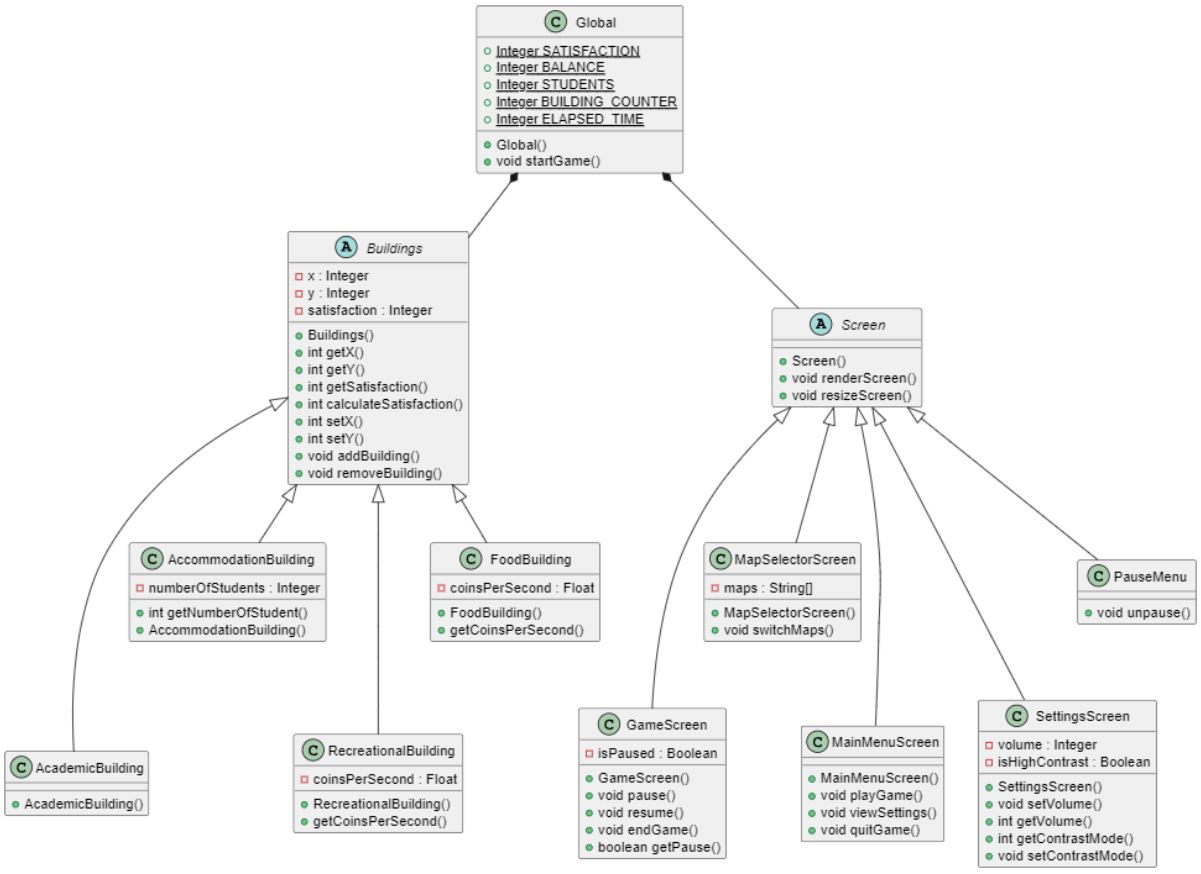


**Figure 2:** Initial Class Diagram

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. To begin with, we decided that our program should have a Global class that will act as the instance of the game. This class contains the variables that will be constantly read and updated throughout the program, such as calculating satisfaction, and updating the balance when a building is bought and placed onto the map. As a result of this, these variables have been assigned a public access modifier. One of the requirements that this class satisfies is FR_TIMER, which is achieved with the ELAPSED_TIME variable. This was important as our customer stated that the game should only be able to last 5 minutes. Another one of the requirements that this class satisfies is UR_COUNTER, which is achieved with the BUILDING_COUNTER variable. This is another important aspect as our customer stated that the user should be able to view the number of buildings that they currently have placed.

The next class that we have designed is the Building class. As the diagram shows, this class has a composition relationship with the Global class. In other words, what we are trying to show here is that a building cannot exist without an instance of the game. The building class contains the necessary variables to represent a building within the game, including an x and y coordinate position, as well as a satisfaction variable to measure the satisfaction a particular building is generating. This class also contains all the necessary methods, such as getters and setters for the coordinates, functions to add and remove buildings from the game, and a function to calculate and update the satisfaction a building has generated. Furthermore, inheriting from the building class we have 4 other classes: AcademicBuilding, AccommodationBuilding, FoodBuilding, and RecreationalBuilding. We decided that inheritance would be the most suitable relationship here as each type of building has the same core functionality, such as generating satisfaction and having an x and y coordinate, but some have more specialised methods specific to that type of building. For example, the recreational and food buildings have the ability to generate coins, whereas the accommodation and academic buildings cannot generate coins. Inheritance was useful here as it ensured that we had minimal code duplication, making our code easier to understand and maintain. The requirements that these classes satisfy are UR_BUILDINGS and FR_BuildingType. These requirements state that the game should have the ability to place a variety of buildings, where each building is its own specific type (FR_ACCOMMODATION_BUILDING, FR_LEARNING_BUILDING, FR_EATING_BUILDING, and FR_RECREATIONAL_BUILDING).

The final class that we have designed is the Screen class. This class is responsible for displaying the contents of the game to the user's screen. Within this class contain the essential methods needed, such as renderScreen() to render the screen to the display, and resizeScreen() to allow the user to resize the screen to their specified window dimensions. This satisfies the FR_SCALING requirement, allowing the user to scale the game screen to any size depending on their device screen size. Inheriting from the Screen class are 5 other classes: GameScreen, MapSelectionScreen, MainMenuScreen, SettingsScreen, and PauseMenu. We decided here that inheritance would be the best relationship to use, as within each of these subclasses the user will also need to be able to resize the screen, and each of these screens also need to be rendered. The GameScreen class contains methods to pause and unpause the game, in situations where the user may need to update game settings without affecting their game progress. This is represented with the private boolean

isPaused and controlled by the PauseMenu class. The MapSelectionScreen provides the functionality for selecting a map before the game has started. This is made possible with the private maps array, in which we store the predefined maps the user can select. To select a map, the public function switchMaps() is used. The MainMenuScreen is the first screen the user is presented with upon loading the game. From here, they have 3 options: start the game with the startGame() method, view the settings screen with the viewSettings() method, and quit the game with the quitGame() method. When the viewSettings() method is evoked, it takes the user to the SettingsScreen which contains two private variables volume and isHighContrast. These variables relate to two of our requirements. FR_MUTABLE is achieved with the volume variable and the method setVolume(), allowing the user to set the volume of the music and sound effects to a level which is comfortable for them. The requirement FR_ACCESSIBILITY is achieved with the isHighContrast variable and the setContrastMode(), allowing the user to toggle the contrast of the screen in the event they are visually impaired. This was important to ensure we had designed within our class diagram, to ensure that the game can be played by anybody no matter their disabilities.

### Class Diagram Evolution

After implementing the classes in the diagram, we realised that there are elements of the game that we did not think about implementing, for example, how we allow the user to switch between screens. This led us to create a screen multiplexer class, which will switch between screens when requested by the user as they navigate through the game. We decided to make Screens an enum, and make it an inner class to ScreenMultiplexer as this way the ScreenMultiplexer can manage all available screens, simplifying screen transitions. This adjustment was made after realising that our initial design did not allow for seamlessly switching screens based on the user actions. In this version, we also introduced a rendering class BuildingRenderer to handle the graphical aspects of buildings separately, which would eventually be integrated into a GameRenderer. This separation of concerns became important as we understood LibGDX better, whilst also wanting to manage the visual elements efficiently without cluttering a single class. The class diagram for this can be found on our website under the “Architecture” tab, marked at “Iteration 1” here.

As the implementation of our game progressed, new rendering classes have been introduced to handle the tasks within the game, such as GameRenderer and UIRenderer. The GameRenderer manages gameplay elements, while the UIRenderer handles the user interface, which ensures a clear separation of concerns and makes it easier to refine each part independently. These classes are essential for the FR_SCALING and FR_ACCESSIBILITY requirements, as they help in rendering elements dynamically according to screen size and user preferences. Moreover, we added the abstract class SuperScreen, which provides a common structure and shared methods for managing screens, like rendering, updating, and handling input. Each specific screen then inherits from SuperScreen. This inheritance not only improves code reuse and reduces duplication but also makes it easier to add or modify screens independently of one another. Finally, global variables like satisfaction, building count, and elapsed time were now placed in a GameGlobals class, making these values more accessible across different parts of the game. These changes streamlined our way of tracking key game metrics, making it easier to integrate with other components and improve maintainability. The class diagram for this can be found on our website under the “Architecture” tab, marked at “Iteration 2” here.

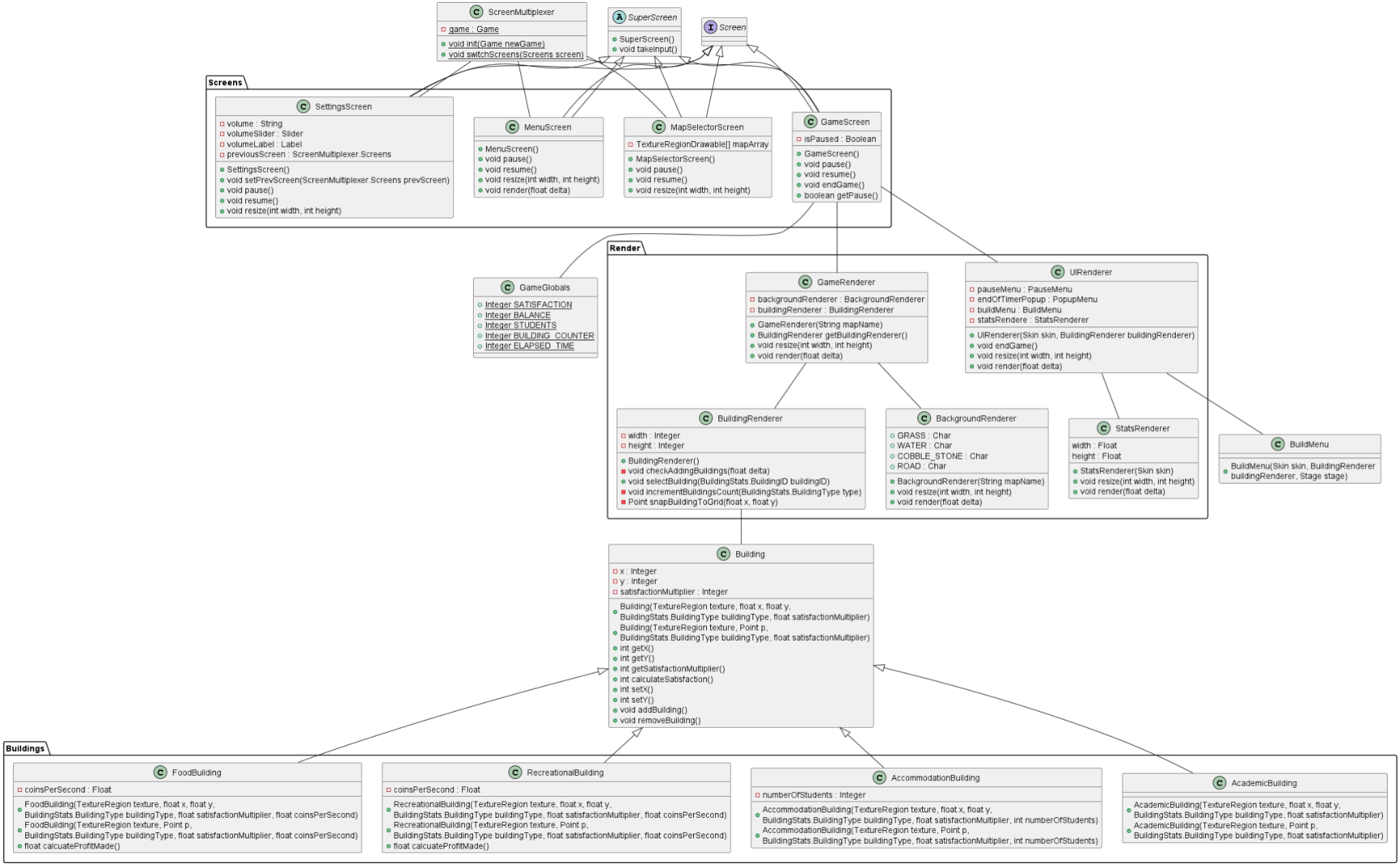


**Figure 3:** Final Class Diagram

## Final Diagram

The image above (Figure 3) represents our final class diagram. This diagram shows the evolution of our classes and how they interact from the initial design. The evolution of the class diagram was made possible through implementing our current class diagram, and then refining the next class diagram with new classes that were still needed that we did not previously think about, or maybe even removing classes that we realised were not necessary once we observed how it impacted the implementation. This agile approach to constantly updating our architecture is important as it ensures we are constantly allowing ourselves to improve the structure of our program, which in turn makes our code more maintainable allowing us to focus on the experience of the user.

One of the major differences between the initial class diagram and the final class diagram is how we represent the top level class for the game. Instead of having the “Global” class, we have 3 classes: ScreenMultiplexer, SuperScreen, and Screen. Once we became more familiar with LibGDX, we realised that the majority of the classes depended on these screen classes to display the game to the user, and allow them to interact with the game. Our game consists of 4 screens, the screen shown to the user when the game is first loaded (MenuScreen), a settings screen (SettingsScreen), a screen to select a different map (MapSelectionScreen) and the screen that displays the actual running game (GameScreen). All these screens inherit the Screen interface provided by LibGDX, which contains the necessary functionality to control each screen. The ScreenMultiplexer class provides the functionality of switching between the different screens as a result of the user’s interaction. Furthermore, now, instead of storing the global variables (satisfaction, building count, elapsed time, etc…) in the Global class which no longer exists, we created a new class GameGlobals to store these. These variables are all of type public static, meaning that they can easily be accessed from any class that should require them, such as the Building class when it needs to calculate satisfaction.

Another major difference between the initial class diagram and the final class diagram is the use of rendering classes. Our game uses two main renderers, GameRenderer which has an association relationship with BuildingRenderer and BackgroundRenderer, and UIRenderer which has an association relationship with StatsRenderer and BuildMenu. In other words, the GameRender takes care of rendering content related to the gameplay such as placing buildings and viewing the background map, and the UIRenderer takes care of displaying the statistics such as the number of buildings placed and the time counter, as well as pop-up menus such as the pause menu and the building purchase menu. The reason that we decided to change our class diagram to include these renderers is mostly due to how LibGDX displays content to the screen, something we were not too familiar with before we started any implementation. Each of these renderers are associated with the GameScreen class, as this is where most of the interaction happens, and therefore the screen is constantly needed to be rendered with updates as a result of the user’s input.

A final, small difference between the initial class diagram and the final class diagram is within the Building class. Instead of storing the overall satisfaction of a building, we represent each building with a satisfaction multiplier, which changes the rate at which satisfaction is increased depending on the building type which is much simpler to maintain.