Architecture 1

Cohort 1, Group 6

Group Members:

Hussain Alhabib

Ellen Matthews

Minnie Poon

Jason Ruan

Daniel Smith

Owen Smith

Introduction and Overview of Tools Used

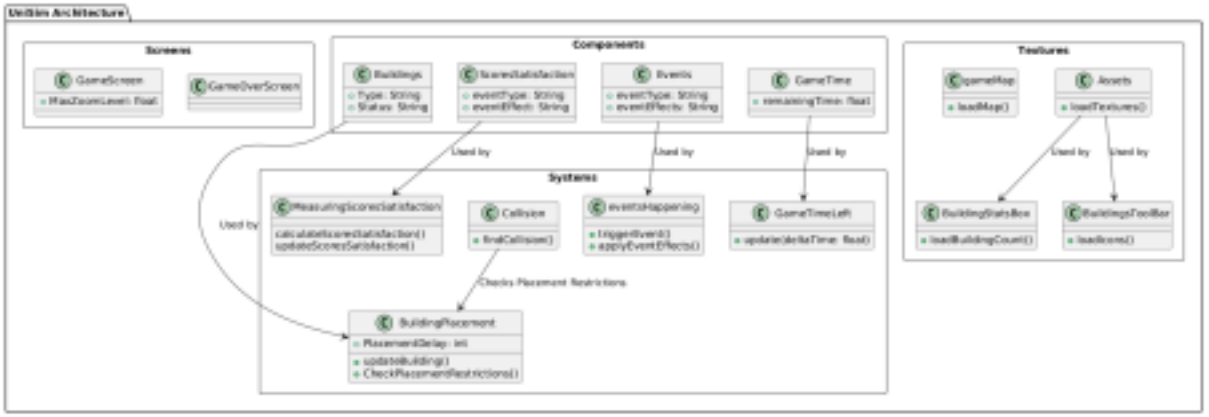
This report outlines the architecture and design of our team's program, UniSim. Our architecture prioritised modularity, scalability, and maintainability, to ensure that our design supports the Agile framework. This document provides an overview of the final architecture, as well as a summary of its evolution since the initial designs, giving stakeholders insight into key architectural changes and improvements over time.

To achieve UniSim's requirements, we selected Java and LibGDX as our core technologies due to their strong compatibility with cross-platform game development (further justification details in Plan1). For the illustration of our architecture, we use PlantUML to generate our UML diagrams. PlantUML's text-based format offers two primary benefits: ease of updates to accommodate iterative design changes and version control integration, which allows our team to track architectural modifications alongside changes to our code. These features are especially important given our evolving project requirements, enabling traceability back to specific requirements.

We did consider alternative tools, such as LucidChart, which offers an intuitive, drag-and-drop interface and advanced collaboration options. However, we chose PlantUML due to its enhanced cost-effective features, and compatibility with our version control needs, which ensures an efficient approach to documenting our architecture as it develops.

PlantUML also supports the effective organisation of project requirements through text based linking. By linking User Requirements with System and Non-Functional Requirements directly in the code, PlantUML enables us to maintain a clear and traceable representation of how each architectural component fulfils user requirements and follows our implementation.

Our Architectural Diagrams and Structure



Textures Module

The Textures module manages all visual assets required for gameplay, such as maps, building icons, and other UI elements. It includes:

- gameMap: which is responsible for loading and managing the game's map through the loadMap() function.
- BuildingsToolBar and BuildingStatsBox: UI elements which allow users to view and manage buildings, and also see the number of buildings placed on the map.
- ScoreBar: which informs the user of their current score to help them measure their current success in the game.

Components Module

The Components module is a core component for game handling. It includes central gameplay elements like buildings, timing, and scoring. Key classes include:

- ScoresSatisfaction: manages the scoring system and measures the satisfaction levels during gameplay, which provides data for the ScoreBar to show players their current progress. ● Events: facilitates the random events that can occur during gameplay, which helps enhance the player's experience.
- GameTime: tracks the in-game time, working with the GameTimeLeft class to inform the player of their remaining gameplay time.

Screens Module

The Screens module manages the player's interaction and visual settings for the gameplay. Key features include:

- GameScreen: supports in-game functionality like zooming, allowing the player to adjust their gameplay screen view with the MaxZoomLevel setting.
- GameOverScreen: informs the player that their current gameplay session has ended.

Systems Module

The Systems module provides the logic and processes behind integral gameplay functions. Key functionalities include:

- BuildingPlacement and Collision: manages the placement of buildings in-game, ensuring they are placed correctly and that objects interact appropriately.
- eventsHappening: executes in-game events randomly as defined in the Events module.

The structure and modularity of this architecture easily allows for gameplay mechanics to be extended or modified to support any new/additional requirements to be added.

Activity Diagram

To further illustrate our system's behavioural architecture, we designed a UML Activity Diagram showing the typical flow of gameplay from the player's perspective. The diagram maps out the series of actions a player might be able to take within the game, and how those actions interact with various system components, such as scoring, building placement, and event triggers.

The Activity Diagram shows the following key actions:

- **Start Game:** The initial game setup, loading in of game assets, and starting of 5-minute countdown.
- **Building:** A player interaction with the UI to place buildings on the map.
- **Score Update:** The ScoresSatisfaction class continuously updates the player's score, reflecting the actions the player's take.
- **Game Over:** An event triggered by either running out of time, failing to meet the necessary score threshold, or an interrupt in the running of the game.

This visualisation of the gameplay process helps clarify how the system's components interact from a player's perspective, aligning the architectural structure with the actual user experience.

Evolution of Our Architecture

In the previous section, you saw the final version of our Architecture. This section details the iterative development of UniSim's architecture, showing the design changes and refinements made during the project's development.

All architecture images can be seen under their respective headings on the Website [https://uoy-team-six.github.io] under the "Architecture" tab.

Iteration 1: Setup of Textures Module

In the initial iteration of our architecture, we prioritised implementing essential in-game features, such as loading the game map. As such, we created a “Textures” module, guided by the importance of a scalable and maintainable structure. The design of this module ensures that future iterations of our game could easily incorporate additional textures and assets.

The “Textures” module was constructed to manage the implementation related to textures e.g. maps. To achieve this, we introduced a “GameMap” class within the module, which is responsible for loading the game’s map efficiently. The class contains a “loadMap()” function which handles the loading process.

Iteration 2: Introduction of Screens Module

For the second iteration of our architecture, we created a new module called “Screens”. Designed in a similar way to our first module, it is easy to add new features to this module as new requirements are proposed and implemented.

As previously discussed, the “Screens” module was designed with a purpose to provide flexibility in managing the game’s display setting to enhance the player experience such as zooming in and out and minimising or maximising the game screen. To do this, a “GameScreen” class was added to the module which is used to specify the maximum zoom level with the MaxZoomLevel method.

Iteration 3: Adding Core Mechanics

In the third iteration of our architecture, we introduced the “Components” and “Systems” modules, which are fundamental to implementing the game’s core mechanics. Designed with modularity and scalability in mind, new components and systems can be easily added as the implementation progresses.

In this iteration, we created the “GameTime” class as part of the “Components” module, which represents the game’s time tracking feature. Paired with this is the “GameTimeLeft” class in the “Systems” module, which is responsible for managing and updating the game’s time in real time.

Iteration 4: Building Placement and Collision Features

The fourth iteration of our architecture created core components to our modules. This included adding the “Buildings” class to our “Components” module, and the “Collision” and “BuildingPlacement” classes to our “Systems” module. This integration was efficient and straightforward, showing the flexibility of our architecture and the efficient nature of our collaborative development process.

These new classes work in tandem to manage building placement on the map and ensure there is a suitable collision detection method within our implementation. “Collision” was made a standalone class due to the significant number of potential objects that require collision handling.

Iteration 5: UI Enhancements

In the fifth iteration of our architecture, we focused on developing UniSim’s User Interface. We used

our existing “Assets” class within the “Textures” module to load in new UI features: the “BuildingStatsBox” class and the “BuildingsToolBar” class.

The “BuildingStatsBox” class represents the number of buildings placed for each type by using the function “loadBuildingCount()”. In parallel, the “BuildingsToolBar” class allows players to click on certain icons to place corresponding buildings by using the “loadIcons()” function.

Iteration 6: Scoring Functionalities

For the sixth iteration of our architecture, we added more core game features. In particular, we added a scoring system to track player performance. To do this, we created a new “ScoresSatisfaction” class in the “Components” module, which is designed to work with a dedicated measurement system that calculates and updates the score as the game progresses. Additionally, we created a “ScoreBar” class in the “Textures” module to provide players with a visual representation of their current score, making it easier to track progress and improving the gameplay experience for players.

Iteration 7 (Assessment 1 Final Iteration)

In our final architecture iteration, we created one more core functionality - the events system. This involved adding the “Events” class to the “Components” module, and the “eventsHappening” class to the “Systems” module. These additions allow different types of events to occur throughout the game, each with different effects that affect the gameplay in different ways. We also implemented a gameover screen by adding the “GameOverScreen” class to the “Screens” module, which is shown when players fail the game.

Additionally, we revisited and refined our “BuildingStatsBox” class to ensure that it can accurately track and display the type and quantity of buildings placed within the game.

By adhering to the principles of modularity, scalability, and traceability throughout our design and implementation, we have developed an architecture that meets our specified requirements effectively. Our approach also allows for future functionalities to be added efficiently while maintaining a clear and structured system.

Note this iteration refers to the same diagrams listed under “Our Architectural Diagrams and Structure” at the start of this document.

Iteration 8 (ASSESSMENT 2)

When picking up the project it was recognised immediately that changes would be required to some features to ensure that they were properly implemented, and others would need to be added entirely. In this new build we began by making a prototype achievements system as well as a leaderboard which displays the top 5 saved scores alongside a 3 character name for each score (a design choice inspired by old arcade games). The main focus of this iteration was functionality; bugs and other issues were noted but not immediately

addressed until we finished adding new functionality.

Iteration 9 (Final Iteration of Assessment 2)

This iteration is what we considered as a group to be the most complete form of the project. All bugs were addressed and promptly fixed, the game was refined to make the controls more intuitive to new players and we created new textures for the new additions to replace the placeholders we were using before. All code was documented sufficiently and consistently in this iteration and the project was able to be compiled as a jar file.

Justification of Our Architecture

As previously mentioned, UniSim's architecture was designed with a focus on modularity, ensuring each component has a well-defined role. This modular approach simplifies both the development and maintenance of the architecture, as each module can be updated independently without impacting the rest of the system. This structure supports easy integration of future features and changes, as evidenced by the efficient evolution of our architecture through several iterations of development.
We also ensured direct traceability from the architecture to the project requirements - aligning each design decision with specific client needs. For example, the UR_TIME_TRACKING requirement is implemented through the GameTime class, which manages the game's five-minute timer (as defined in SR_TIME). Similarly, UR_BUILDING_LIMITS is enforced by the BuildingPlacement class, which prevents overlapping building placements, aligning with SR_BUILDING_RESTRICTIONS.

By organising the game's architecture into distinct modules: Textures, Components, Systems, and Screens - we were able to separate functionalities and implementation effectively. Each module has a clear purpose, as defined in the "Our Architectural Diagrams and Structure" section.

The modularity also enhances both maintainability and scalability, allowing for easy updates to assets, gameplay features, or UI elements. For example, the BuildingsToolBar and ScoreBar UI elements are contained within the "Textures" module which makes it easy to modify them as new building types or scoring systems are added to the game.

Requirements Traceability

UniSim's architecture was designed with a heightened focus on requirements traceability, to ensure that every core functionality aligns with the specified user and system requirements. Below are examples of how the architecture addresses our key requirements:

1. UR_BASIC_BUILDINGS and SR_PLACE_BUILDINGS; these requirements were fulfilled through the Buildings class and its related features, including BuildingPlacement and Collision (which also meets the requirements of UR_BUILDING_LIMITS and SR_BUILDING_RESTRICTIONS).
2. UR_TIME_TRACKING; we implemented this feature using the GameTime component paired with the GameTimeLeft feature which allows for game time to be tracked and updated,

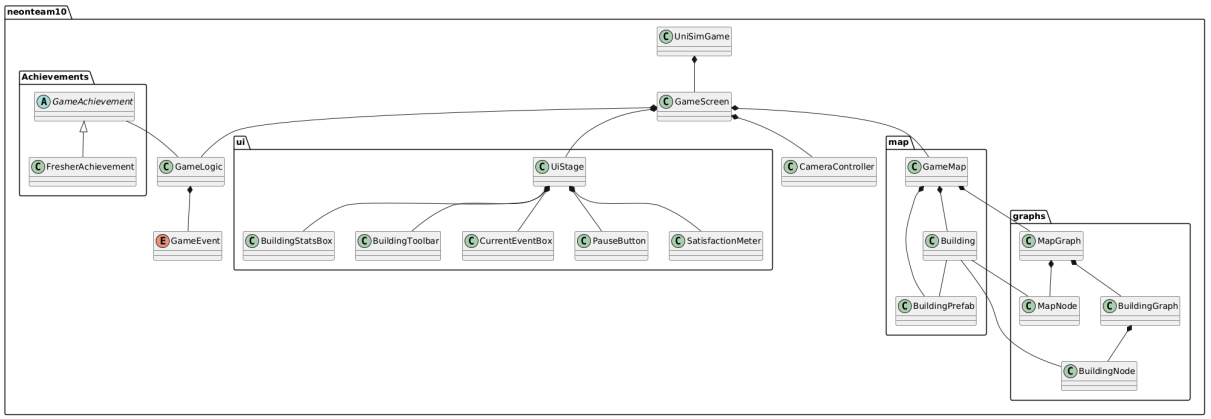
fulfilling the requirement of SR_TIME.

3. UR_BUILDING_COUNTER and SR_BUILDING_COUNTER; these are met by the BuildingsStatBox class which is located in the Textures module which tracks the number of placed buildings, and the Buildings module which when connected to the class can efficiently update the game's building counter.
4. UR_SCORE and SR_METRICS; the ScoreSatisfaction and ScoreBar classes help display a visual score tracker allowing players to easily monitor their performance.
5. UR_EVENTS; we developed an Events component to define the varying event types and their effects on gameplay - also fulfilling SR_EVENTS. The eventsHappening class is also linked to this by triggering events randomly during gameplay.

Overall, as you can see our chosen architectural design allows for easy updates and changes to accommodate new requirements or features, ensuring UniSim continues to meet both its user and system requirements.

Assessment 2:

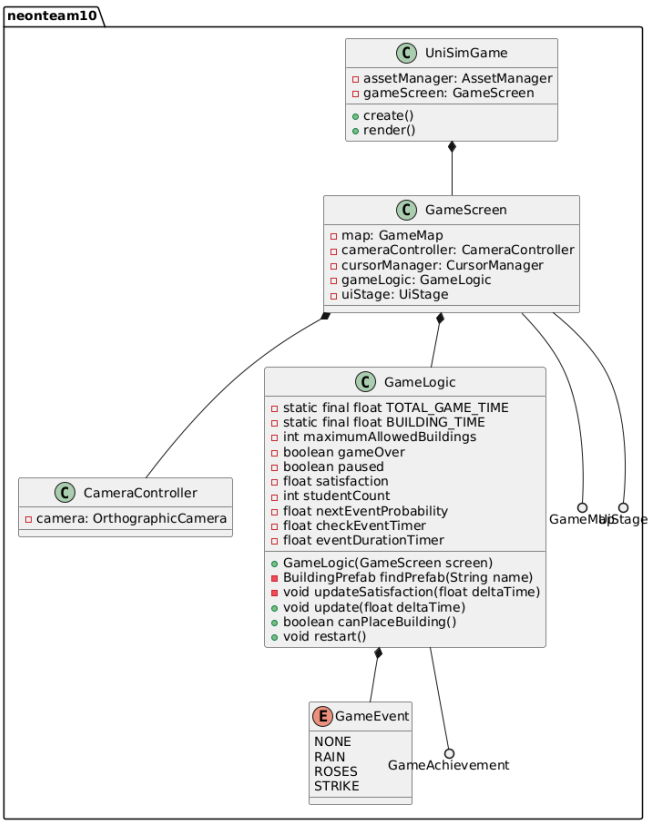
Architectural Diagrams:



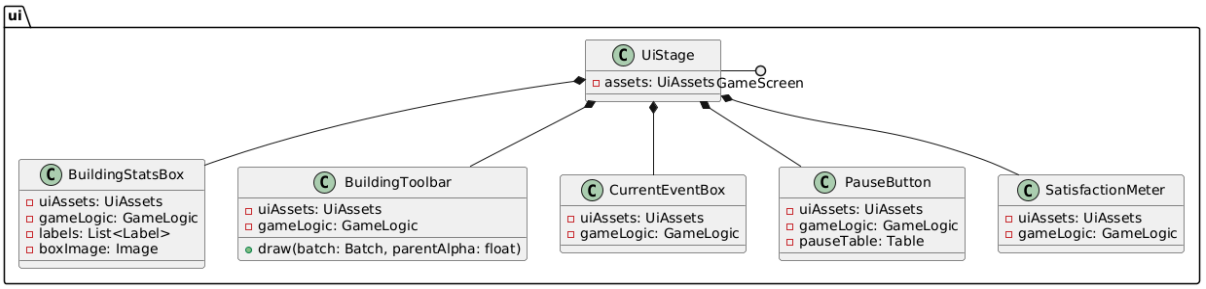
This is an abstracted diagram with all attributes and methods removed, to clearly show how the different packages and classes relate. The following diagrams will go more in depth on each individual package and describe a bit more about how the classes handle each task

Generic/ unpackaged:

This diagram details the unpackaged classes in our project: UniSimGame, GameScreen, GameLogic, CameraController, and the Enum Game Event. All of these are mostly for handling how the game is rendered for the user, with the exception of GameLogic, which handles most of the backend calls to other methods.



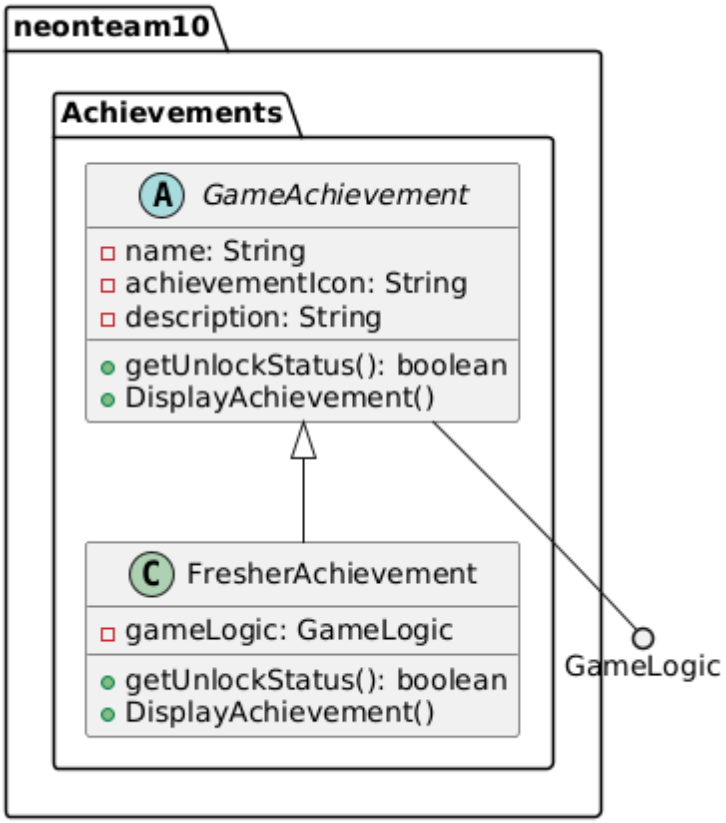
UI Package:



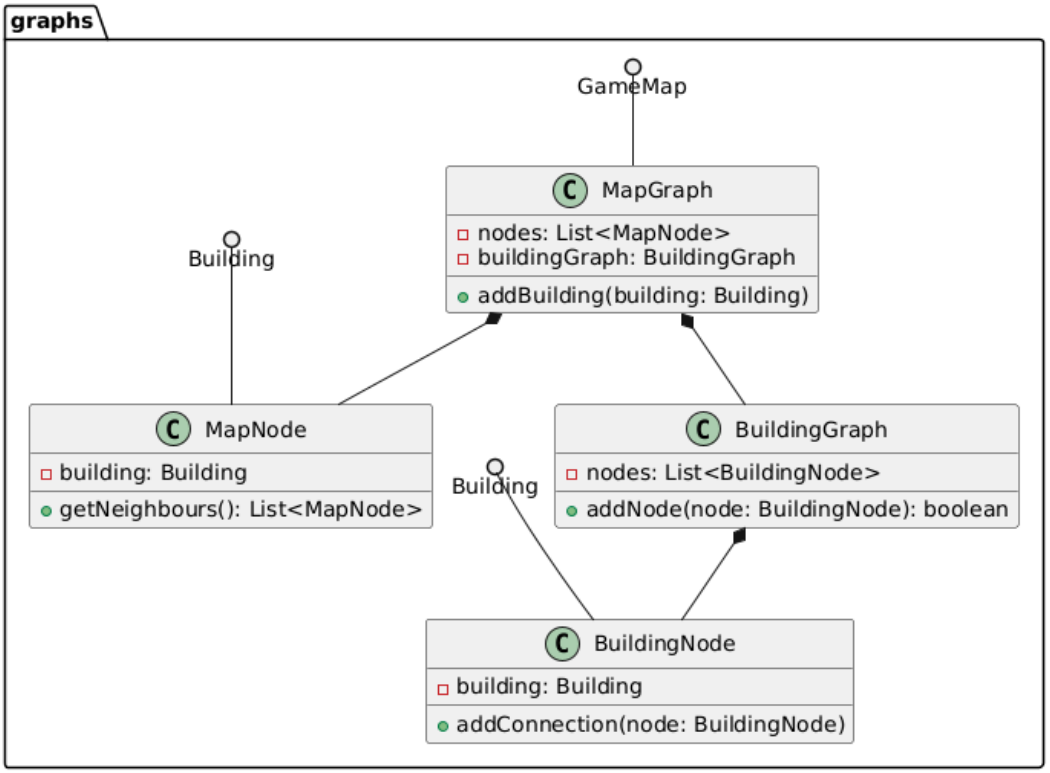
This package handles Updating and drawing all the UI elements on the screen.

Achievements Package

This package handles the Achievements system, containing the Abstract GameAchievementClass, And Several child classes to represent each individual achievement.



Graphs Package:



This package handles all the Graphs used in the game. MapGraph is the way that the game's map is stored in the backend, where BuildingGraph is a subclass that maps onto MapGraph, representing the player's building placements. MapNode and BuildingNode both represent the data types for each node on their respective graphs.

Map Package:

This package contains the Building Class, one of the most granular classes in our project, as well as the GameMap class, which uses the MapGraph class described previously to store the current state of the game. The BuildingPrefab class stores all the types of buildings in the game, and is used when constructing a building instance.