University of York Department of Computer Science
Engineering 1

Architecture

Cohort 3 Team 5 - alltheeb5t

Aaron Heald
Alex Gu
Arun Hill
Jade Stokes
Maksim Soshchin
Meg Tierney
Will Hall

Use Case Diagram

After our meeting with our client, we created a use case diagram of how we wanted the user to interact with our game. A use-case diagram was chosen to represent the flow of our game as it can clearly and simply show how the user would interact with our system, enabling us to plan the structure of the game while keeping the user in mind.

This diagram is shown in Figure 1. Each circle represents a use-case (representing a screen in the game) and rectangles have been used to group use cases that are shown on the same screen for clarity. Arrows are also used to show the outcome of users' interactions.

In order to match the project requirements as accurately as possible, each use-case in the diagram was related to a corresponding requirement. For example, the 'Leaderboard' use-case links to FR_LEADERBOARD, and the 'Settings' use-case (which represents settings options such as volume adjustments) corresponds to FR_ACCESSIBILITY and FR_SOUND.

Each screen (and corresponding use-case) was also designed to break up functionality in a similar manner to other games on the market to help fulfil NFR_EASE_OF_USE, therefore avoiding clutter on each screen to help the user navigate between them more easily.

The use case diagram was created in the drawing application Goodnotes 6 due to the fact that it provided a simple, streamlined facility for designing a quick sketch that could be easily modified as the design process progressed.

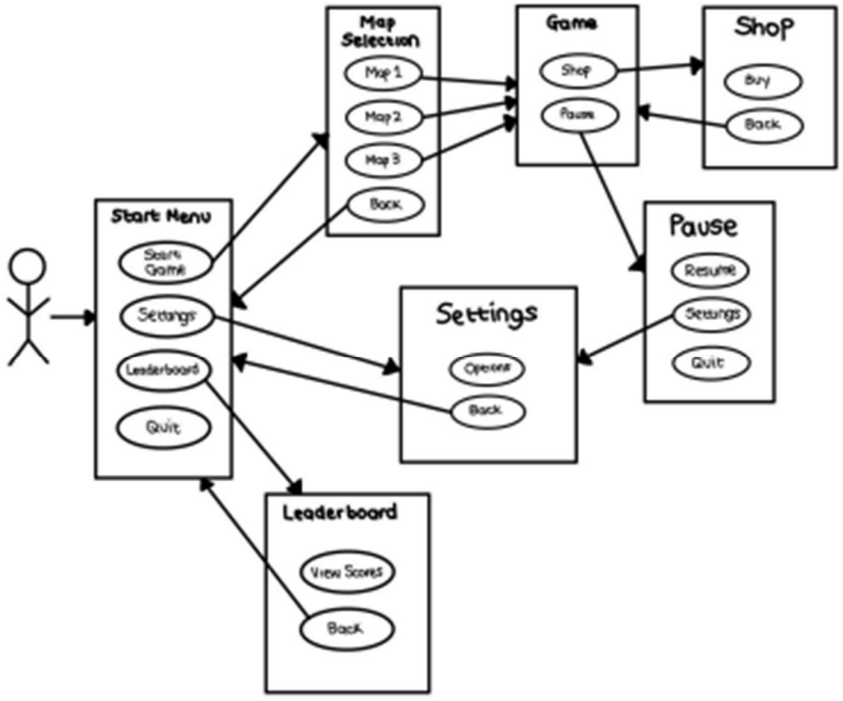

Figure 1: Use Case Diagram

Class Diagrams

Once the use case diagram had been created, it was used to develop UML class diagrams to represent and refine the structure and behaviour of the game.

These class diagrams were generated from text data input into the PlantUML tool. PlantUML was used due to its suitability for the project as it is both free and open source. Google Docs also supports the PlantUML Gizmo Extension, which enables us to conveniently insert diagrams directly into deliverables documents.

Initial Diagram

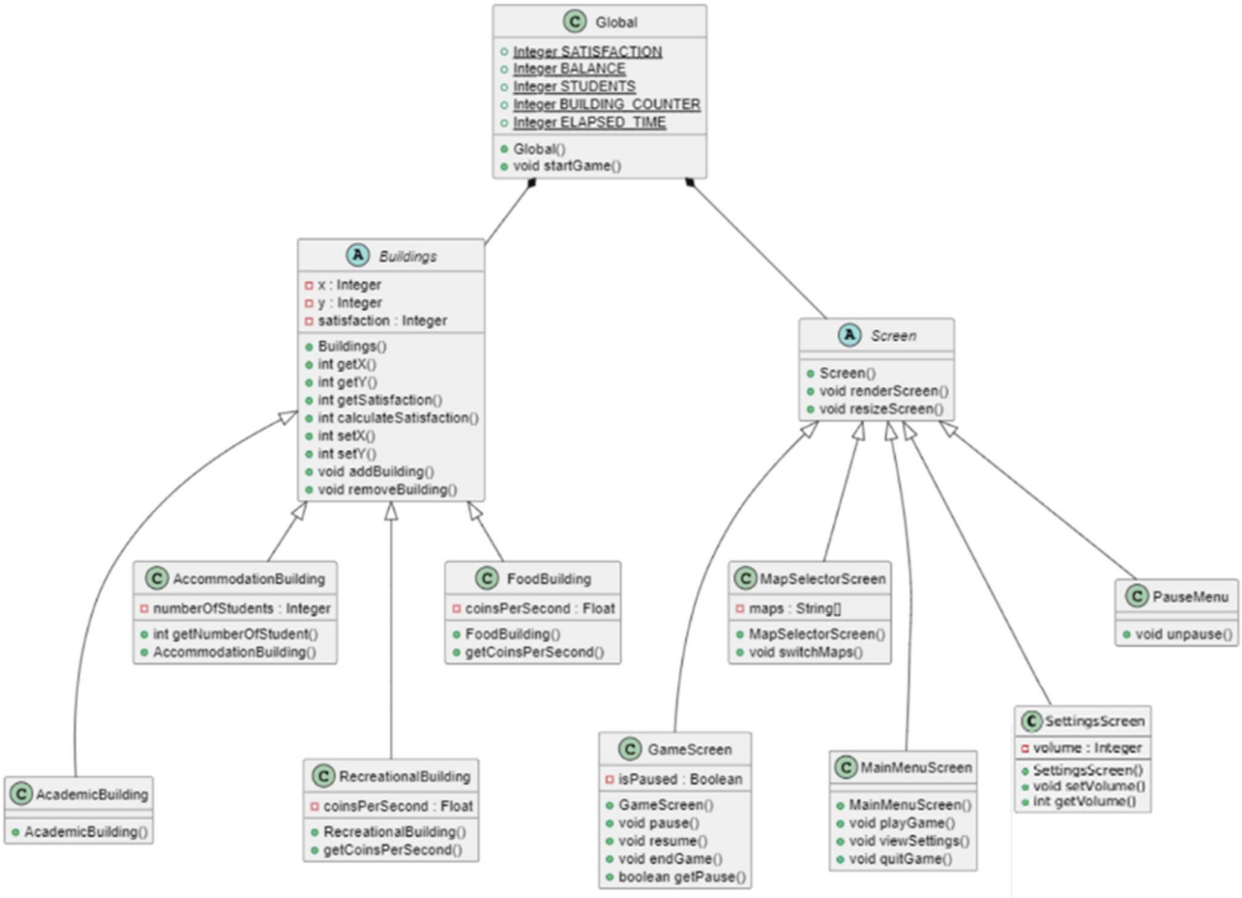

Figure 2: Initial Class Diagram

The class diagram above represents our initial design of the game. This was created by using both our requirements and use-case diagrams to ensure that all details of the product brief were met, and the details of how each class satisfies requirements are explained below.

Global Class

The Global class stores the current state of the game and is constantly read and updated while the game is running in order to satisfy several requirements (such as NFR_METRICS). For example, this class maintains the variable ELAPSED_TIME, in order to track the in-game timer (and ensure that the requirements FR_TIMER and UR_TIME are met), as well as the variable BUILDING_COUNTER to track the current number of existing buildings and therefore satisfy UR_COUNTER. Global also tracks both money and satisfaction metrics to meet both UR_SATISFACTION and UR_FINANCE.

Buildings Superclass and Subclasses

Several subclasses – such as AcademicBuilding and FoodBuilding – inherit from the abstract superclass Buildings in order to represent and implement each type of university building. Every instance of a Buildings subclass therefore contains that instance's position on the screen (stored as x and y coordinates), and an integer value representing its effect on student satisfaction. Some subclasses also store the income a building generates per unit time or the number of students that live there (for AccommodationBuilding instances). Due to this, the requirements UR_BUILDINGS, FR_BUILDING_TYPE, FR_ACCOMMODATION_BUILDING, FR_LEARNING_BUILDING, FR_EATING_BUILDING and FR_RECREATIONAL_BUILDING can be fulfilled.

Screen Superclass and Subclasses

The abstract superclass, Screen, is used to provide a general, basic representation for each individual screen within the game.

Subclasses that inherit from Screen include SettingsMenu and PauseMenu, which enable users to pause the game and timer and adjust settings such as the game music volume (therefore meeting requirements UR_SOUND, FR_SOUND, FR_MUTEABLE and FR_PAUSE).

MapSelectorScreen and MainMenuScreen also inherit from Screen and provide a landing screen (with a tutorial) for users when they first open the game, as well as a screen to enable them to select a map to play on. These subclasses help to satisfy requirements UR_MENU, FR_MAP, FR_TUTORIAL and NFR_EASE_OF_USE.

GameScreen inherits from Screen as well. This subclass is used to represent the actual game on the screen, showing the map and all existing buildings, as well as buttons representing actions for users to carry out (for example, buy an accommodation building, or pause the game). The subclass also handles displaying the end screen once the game timer has run for five minutes and the game is over. This class therefore enables the following requirements to be met:

- UR_BUILDINGS and UR_COUNTER
- UR_MAP and FR_MAP
- FR_ACCOMMODATION_BUILDING, FR_LEARNING_BUILDING, FR_EATING_BUILDING and FR_RECREATIONAL_BUILDING
- FR_SATISFACTION
- FR_PAUSE
- FR_END_SCREEN

Class Diagram Evolution

After the creation of the initial class diagram in Figure 2, several new diagrams were subsequently made to further refine the architecture of the game and ensure that the structure of all aspects of functionality had been planned and refined.

As the structure of the game was revised, several changes were made to existing classes and new ones were introduced with each new iteration of the system's structure.

Iteration 1

For example, the new class ScreenMultiplexer was created, alongside an enum Screens, in order to handle smooth switching between different screens and allow the user to navigate through the game's different menus (enabling UR_MENU to be met).

Two new classes, GameRenderer and BuildingRenderer, were also added to handle the rendering of the game screen and the graphical aspects of buildings in order to better manage the visual elements of playing the game and avoid control being too centralised.

The class diagram for this can be found on our website under the "Architecture" tab in the Assessment 1 section, marked at "Figure 2", at https://alltheeb5t.github.io/assessment-2.github.io/assessment1/architecture.html.

Iteration 2

Further changes made to the structure of the game include the introduction of new rendering classes such as UIRenderer to further delegate control for rendering tasks. GameRenderer now 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 therefore satisfy FR_SCALING and FR_ACCESSIBILITY, as they help to render elements dynamically according to screen size and user preferences.

The abstract class Screen was also renamed to SuperScreen and given the additional functionality of handling input, therefore enabling each of the screen subclasses (that inherit from this class) to do so.

Finally, the class Global was converted into the singleton class GameGlobals, therefore streamlining the tracking of 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 in the Assessment 1 section, marked at "Figure 3", at https://alltheeb5t.github.io/assessment-2.github.io/assessment1/architecture.html.

Assessment One Final Class Diagram

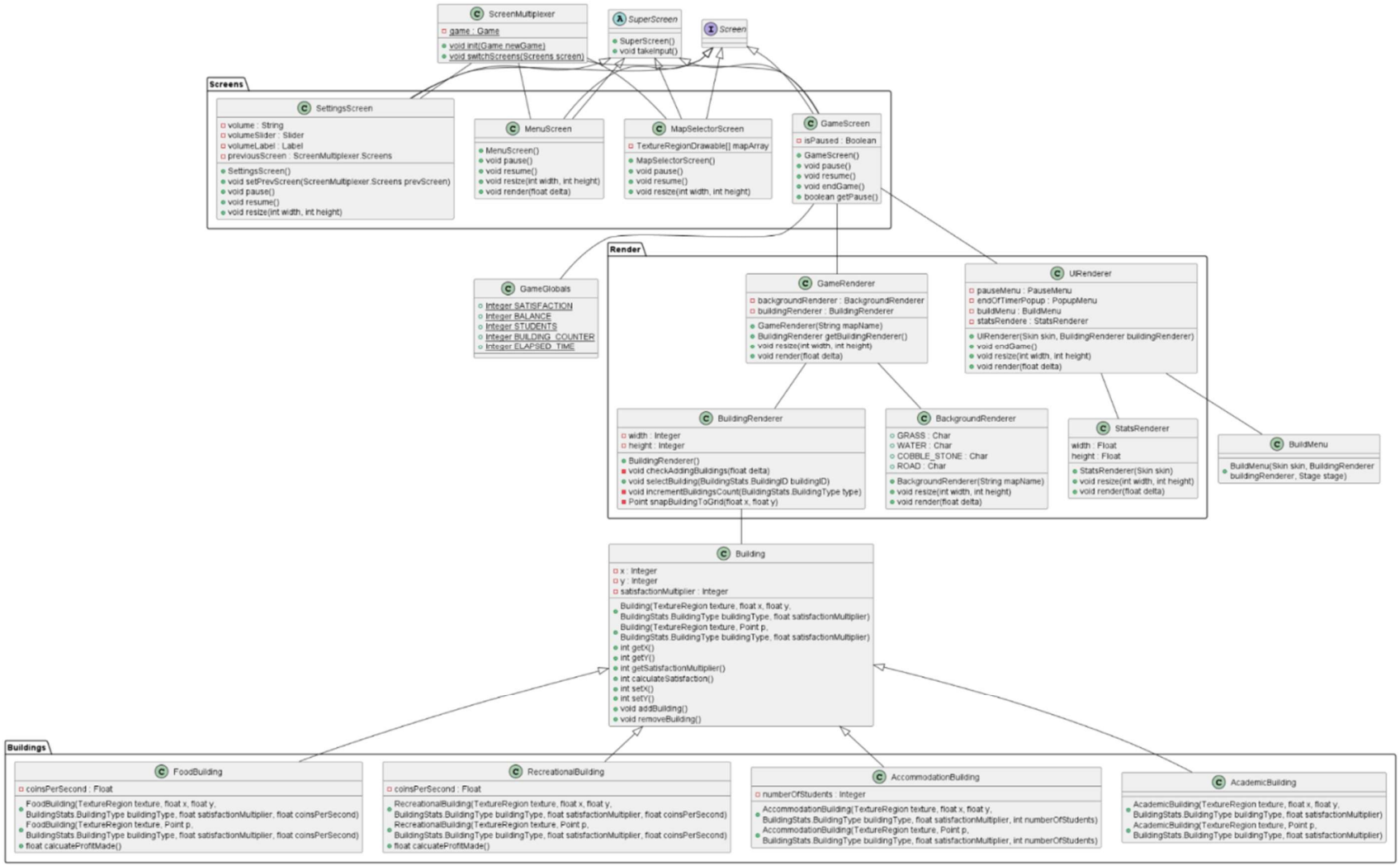

Figure 3: Assessment One final class diagram

The image above (Figure 3) represents the final class diagram for Assessment One. This diagram was the result of the agile approach of constantly updating and refining the game's architecture after each sprint and design iteration in order to maximise code maintainability.

GameGlobals Class

One of the major differences between this diagram and those of previous iterations is that GameGlobals has been linked to GameScreen in order to better control access to static key game metrics. This means that now, while playing the game, the main game screen will be managed by GameScreen, which will also handle input due to the functionality it inherits from SuperScreen.

In response to users' interactions, GameScreen therefore manipulates the static variables in GameGlobals and calls the relevant methods from the classes GameRenderer and UIRenderer to display the results of those changes on the screen. This enables the game to satisfy the following requirements:

- UR_BUILDINGS
    - NFR_FAST_PLACEMENT
    - FR_ACCOMMODATION_BUILDING, FR_LEARNING_BUILDING, FR_EATING_BUILDING and FR_RECREATIONAL_BUILDING
- UR_MAP
    - FR_MAP
- UR_SATISFACTION, UR_FINANCE and UR_TIME
    - NFR_METRICS
    - FR_TIMER
    - FR_SATISFACTION
- UR_COUNTER
    - FR_COUNT
- NFR_EASE_OF_USE
- FR_PAUSE

Renderer Classes

The rendering classes in Figure 3 are used to handle displaying the background map, UI, existing buildings and pop-up menus. The two main classes, which are associated with GameScreen, are GameRenderer and UIRenderer.

GameRenderer manages the rendering of the map and placing buildings. In order to perform this function (and meet requirements: UR_BUILDINGS, UR_COUNTER and UR_MAP), this class is associated with the two rendering classes BuildingRenderer and BackgroundRenderer.

BuildingRenderer handles the placement of new buildings on the map through its association with the Building class, while also ensuring that the building counter in GameGlobals is accurate. BackgroundRenderer handles the rendering of the maps that users choose to play on (and any rescaling in order to help satisfy FR_SCALING).

The other main rendering class, UIRenderer, handles displaying key game metrics to users - such as the game timer, student satisfaction and the building counter – through the use of the StatsRenderer class in order to satisfy UR_FINANCE, UR_SATISFACTION,

UR_COUNTER and FR_TIME. Furthermore, UIRenderer also uses the class BuildMenu to generate a pop-up menu enabling users to purchase new buildings for placement on the map. This therefore helps to ensure that the game meets UR_BUILDINGS.

Building Class

The Building class has also been modified so that instances store a satisfaction multiplier instead of an individual satisfaction score in order to meet UR_SATISFACTION. The overall satisfaction score, stored in the static variable SATISFACTION in GameGlobals is therefore now calculated based on the multipliers from all existing buildings, therefore resulting in a simpler, more maintainable approach to managing student satisfaction.

Assessment Two Class Diagrams

The diagrams below represent the final version of the game architecture for Assessment Two. These diagrams are the result of repeatedly revising and refining the system's structure over the course of the software development process throughout Assessment Two in order to meet all requirements set out in the product brief, and to ensure that the system is as simple and streamlined as possible while also being both easy and convenient to maintain and extend.

Overall Structure

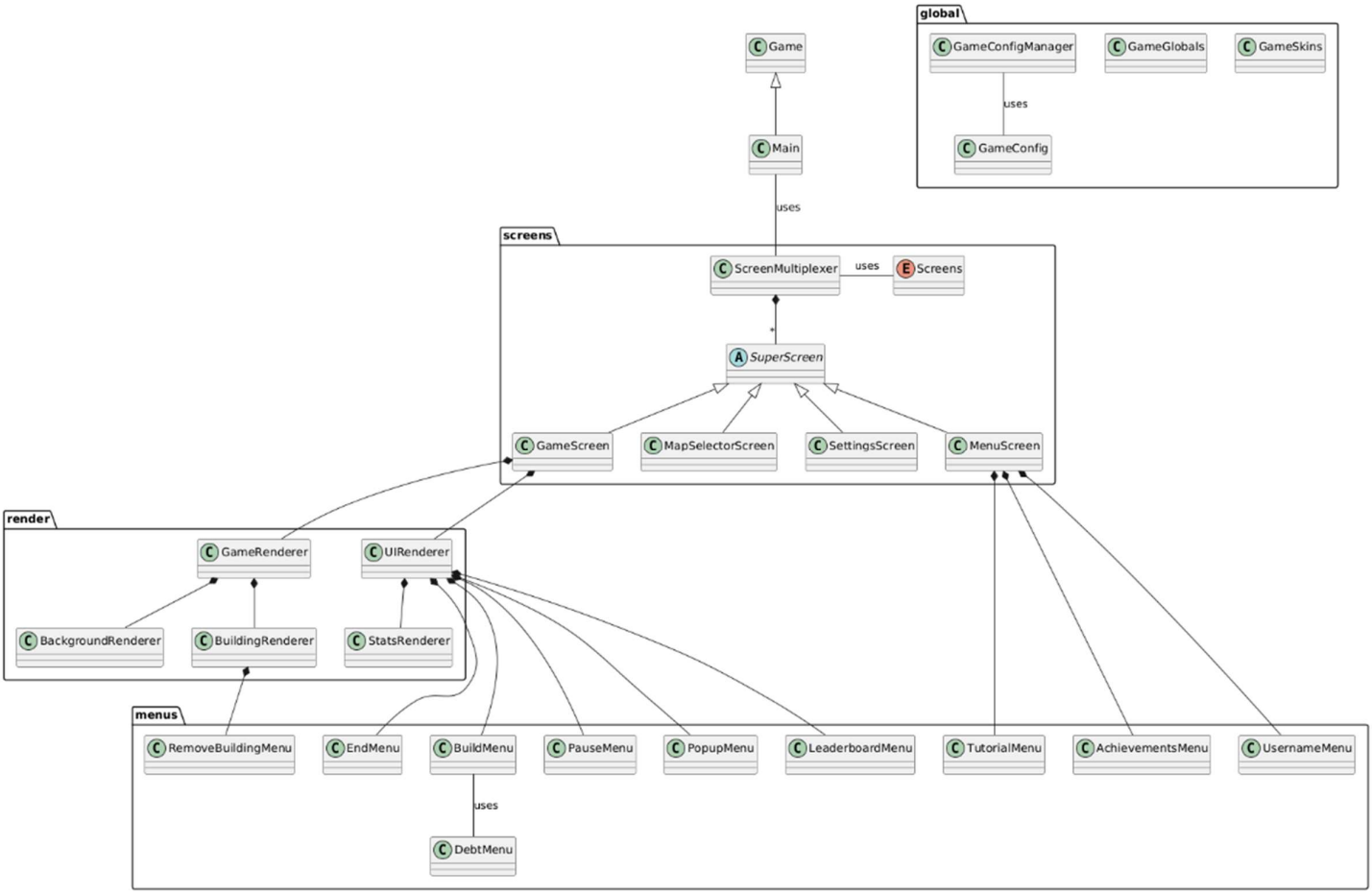

Figure 4: Assessment Two overall structure class diagram

The diagram above represents the overall final structure of the game application's architecture for Assessment Two after several revisions.

When the application is opened, Main initialises the class ScreenMultiplexer which handles the initialisation of instances of each type of screen and enables users to switch between them.

These screens then take and respond to user input, using various menu and rendering classes to display pop-up menus and render users' interactions and the current game state on the screen in order to help meet UR_MENU, UR_MAP and UR_EXPERIENCE.

The running of the actual game itself is handled by GameScreen, which again utilises both GameRenderer and UIRenderer to display the game's current state while also interacting with GameGlobals to track and manipulate key game metrics such as in-game time; currently existing buildings and their relative positions; and unlocked achievements.

The Global Package

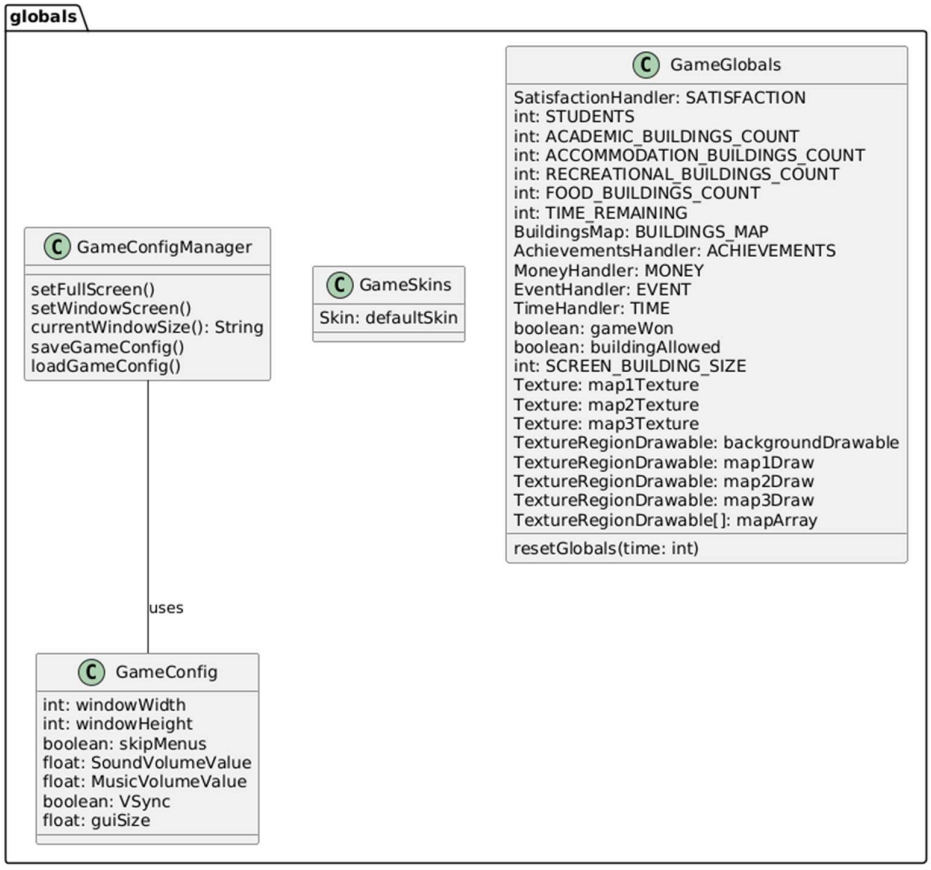

Figure 5: Class diagram of the global package

The classes within the package above can be accessed throughout the entire game and are used to save, store and manipulate the game's state.

The classes GameConfigManager and GameConfig work in tandem to save the current user's preferred configuration of the game application (for example, their volume settings and preferred window size), to a binary file that is loaded each time the application is opened to set the application to the current user's preferred configuration in order to help satisfy UR_EXPERIENCE.

The singleton class GameSkins is used to streamline the loading of Skins from asset files.

The main class within the global package is GameGlobals which, when a playthrough of the game is in progress stores the game's state within its static attributes as a mix of handler classes and raw values. These values can then be accessed and manipulated from any class within the system, therefore centralising the storage of key game data while enabling the delegation of control of that data to various classes throughout the application.

Buildings and the Game Map

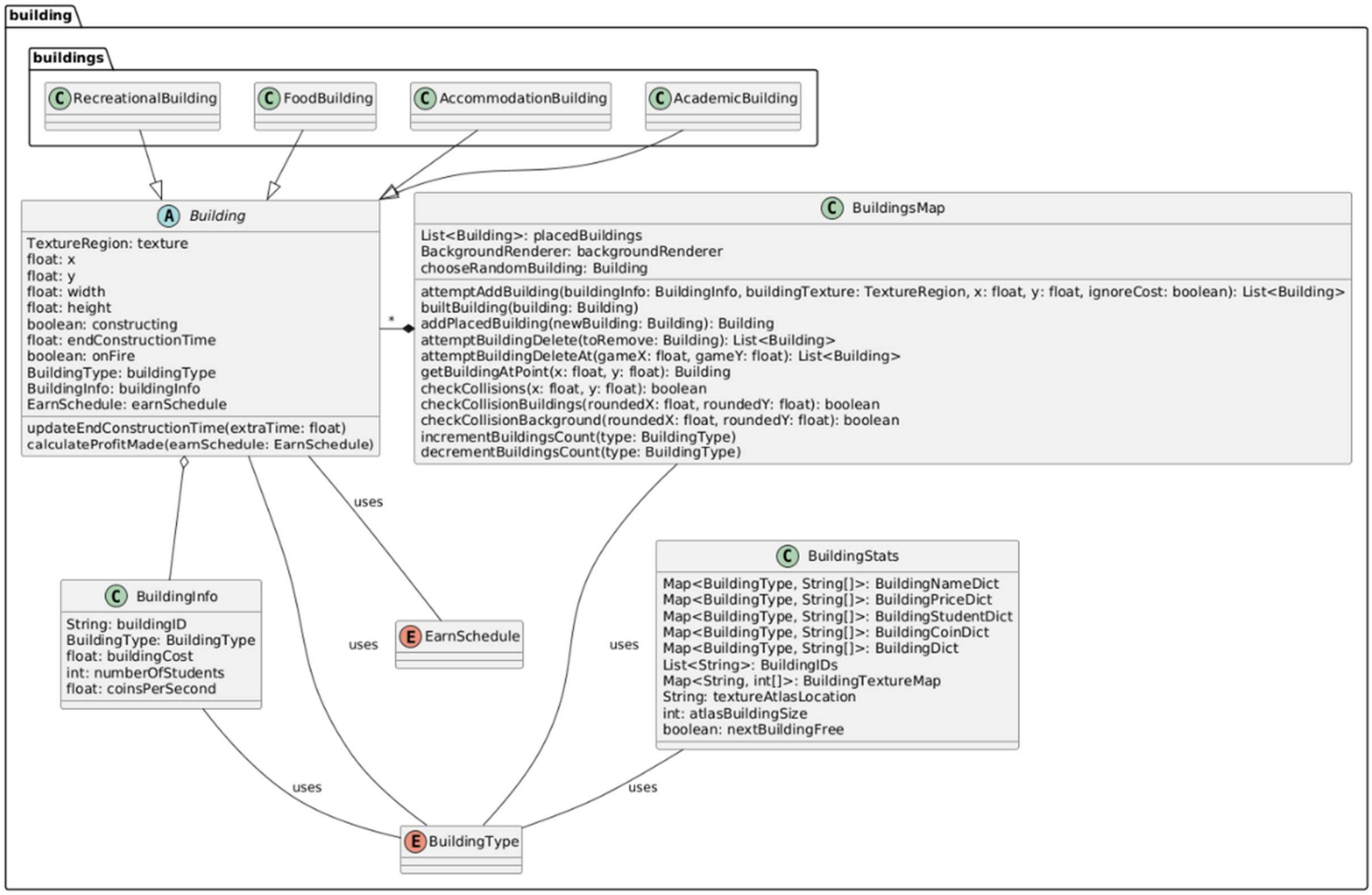

Figure 6: Class diagram showing the building package

The class diagram in Figure 6 represents the structure of how buildings are represented and stored within the application in order to meet requirements: UR_BUILDINGS, UR_MAP, FR_BUILDING_TYPE, FR_MAP, FR_ACCOMMODATION_BUILDING, FR_LEARNING_BUILDING, FR_EATING_BUILDING, FR_RECREATIONAL_BUILDING, FR_BUILD_TIME and FR_NO_OVERLAP.

Each type of building is represented by a subclass of the abstract superclass Building, which has attributes to store the position, construction time and current state of its instances. The type of building represented is specified using the BuildingType enum and – if it earns

money – the details of income generation (daily or semesterly) are specified using the EarnSchedule enum.

Instances of Building utilise and associate with corresponding instances of BuildingInfo in order to be able to provide key details to other classes (such as a building's unique ID or cost to buy), wrapped together as a single object for convenience.

These Building instances are managed by the class BuildingsMap, which tracks existing buildings and provides the functionality to add and remove new ones (all while ensuring that buildings are constructed within an appropriate amount of time and do not collide with each other or other obstacles around the map). BuildingsMap is stored as a static attribute within the GameGlobals class so that it can be accessed by other classes for rendering purposes.

The final class within the building package, BuildingStats, stores the details of every type of building within static dictionaries which can be used for reference by other classes throughout the application.

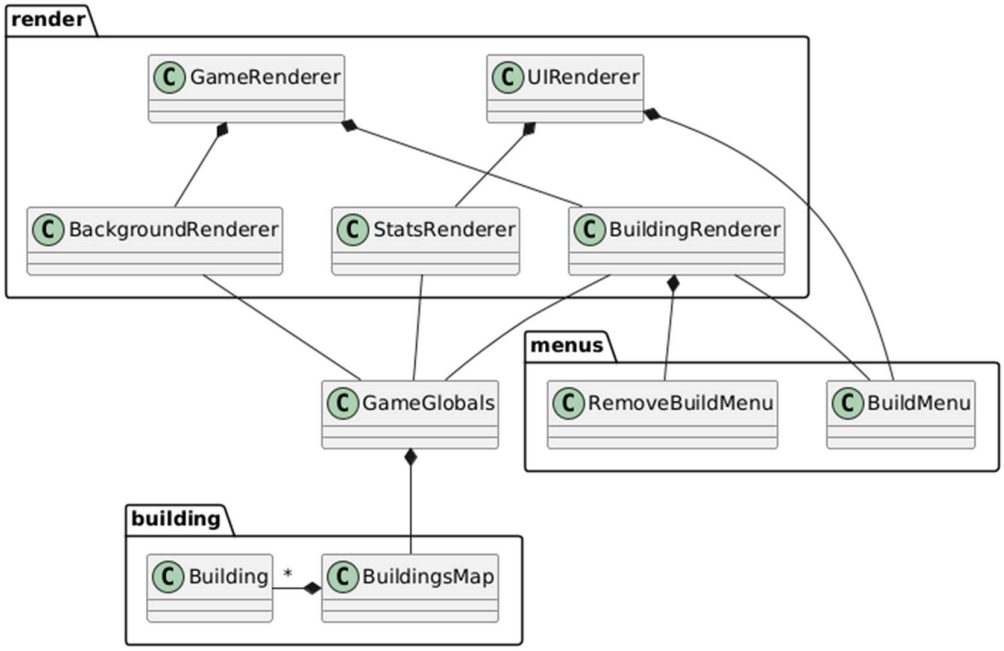

Figure 7: Class diagram showing how buildings are rendered and manipulated

The class diagram above represents how the Building instances stored in BuildingsMap and referenced in GameGlobals are used by the classes GameRenderer and UIRenderer to enable users to buy new buildings and interact with existing ones.

The UIRenderer class utilises StatsRenderer to access and display a counter for the number of each type of existing building, and also triggers the rendering of an instance of BuildMenu when users attempt to view the buildings available to buy.

BuildMenu displays these available buildings and their details on the screen, calling methods in the BuildingRenderer class when the user purchases a new building.

The primary function of the BuildingRenderer class is to interact with BuildingsMap - in GameGlobals - to display each of the existing Building instances at their correct relative positions on the game map, which is rendered by BackgroundRenderer. This class also works with both BuildingsMap and BuildMenu to enable users to place new buildings on the map, and also utilises the RemoveBuildMenu class to give users the option to demolish buildings by right clicking on them.

Achievements

Figure 8: Class diagram showing the structure of achievements and how they are managed

The diagram in Figure 8 shows how achievements are represented within the game application.

Each individual achievement is represented in a class that inherits from the abstract Achievement superclass. Due to this, achievements all have a unique name and description, and each subclass overrides the method isCompleted() to perform its own checks as to whether that achievement has been unlocked.

A list of all the achievements is stored in the AchievementHandler class, which manages and controls the achievement classes' behaviour and is stored itself within GameGlobals. AchievementHandler also associates achievements with the player's username and saves them to a text file so they can be retrieved at a later time. This class maintains a queue of achievements to display as well, so that users can be notified mid-game when achievements are unlocked.

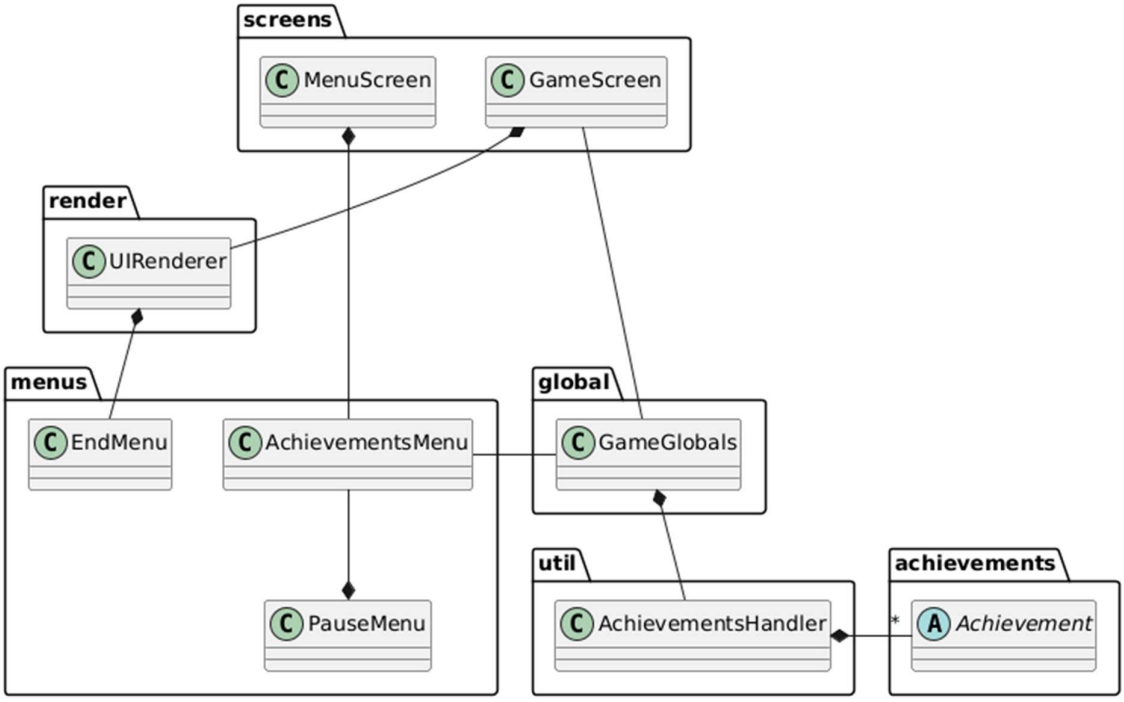

Figure 9: Class diagram showing how achievements are displayed

Figure 9 shows how achievements are both tracked and displayed when the game application is running.

While a playthrough of the game is active, the class GameScreen repeatedly calls the method checkAllAchievements() within the AchievementHandler instance (stored in GameGlobals) to ensure that the game registers when achievements are unlocked and can react accordingly. This therefore ensures that the game meets the following requirements:

- UR_ACHIEVEMENTS
- FR_MFA_UNLOCK
- FR_BM_UNLOCK
- FR_PRIORITIES_UNLOCK
- FR_ITAU_UNLOCK
- FR_UNLUCKY_UNLOCK
- FR_LUCKY_UNLOCK
- FR_INDECISIVE_UNLOCK
- FR_CS_UNLOCK
- FR_SAVIOUR_UNLOCK
- FR_BUSY_UNLOCK
- FR_CHANGE_UNLOCK
- FR_REALISTIC_UNLOCK

Achievements are displayed to users at several different points while the game application is open, therefore satisfying FR_ACHIEVEMENT_MENU.

They are displayed on the initial landing screen of the game (represented by the class MenuScreen), where users can trigger the display of a popup menu showing their unlocked

achievements. This is a menu is generated by AchievementsMenu, which accesses unlocked achievements through GameGlobals and formats them for viewing.

The same menu stored in the AchievementsMenu class can also be displayed when users choose to view their achievements after pausing the game (which is facilitated by the association between the PauseMenu class and AchievementsMenu).

Achievements are also displayed when a playthrough of the game ends. When the game time (stored in GameGlobals) reaches five minutes, GameScreen calls methods in UIRenderer to trigger the display of an instance of EndMenu (the popup menu that displays users' statistics and performance at the end of each game). This instance of EndMenu is passed a list of achievements that have been unlocked in the last playthrough, which can then be displayed on the screen.

Events

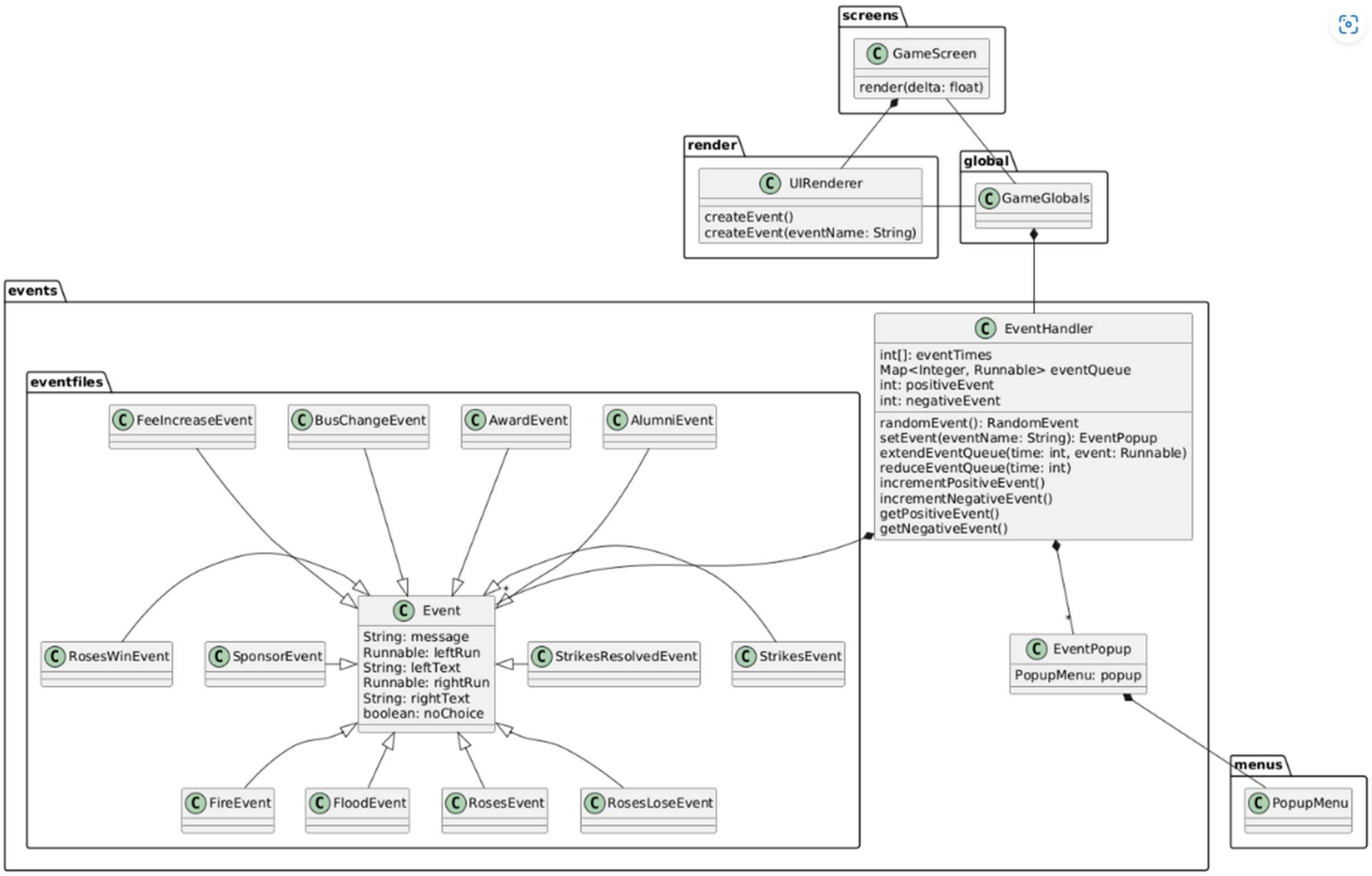

Figure 10: Class diagram showing how in-game events are structured

The diagram in Figure 10 represents the structure and behaviour of events in the game application, and how the requirements below are met:

- UR_EVENTS
- FR_EVENT_GENERATOR
- FR_EVENT_RESULT
- FR_EVENT_DISPLAY
- FR_EVENT_CHOICE
- FR_STRIKE_EVENT
- FR_STRIKE_CHOICE
- FR_BUS_CHANGE_EVENT
- FR_FEE_INCREASE_EVENT
- FR_SPONSOR_EVENT
- FR_ALUMNI_EVENT
- FR_FLOOD_EVENT
- FR_FIRE_EVENT
- FR_AWARD_EVENT
- FR_ROSES_EVENT
- FR_ROSES_RESULT_CALCULATION
- FR_ROSES_FINAL

In order to satisfy these requirements, an EventHandler controls the creation of at least three random events at random times throughout each playthrough of the game, and also enacts their effects.

When it is time for an event to happen, the EventHandler instantiates a random Event subclass and a corresponding EventPopup. The EventPopup has an attribute which is an instance of PopupMenu that displays the details of the current event on the screen to user, while also (for specific events) allowing them to select from one of two options to respond to the event – which then has an impact on key game metrics such as satisfaction.

The class GameScreen enables this to happen by constantly checking whether the in-game time matches one of the random event times stored in EventHandler, which it accesses through GameGlobals. If it is time for an event, GameScreen calls the relevant methods in UIRenderer, which trigger the creation of an event by EventHandler

Leaderboard

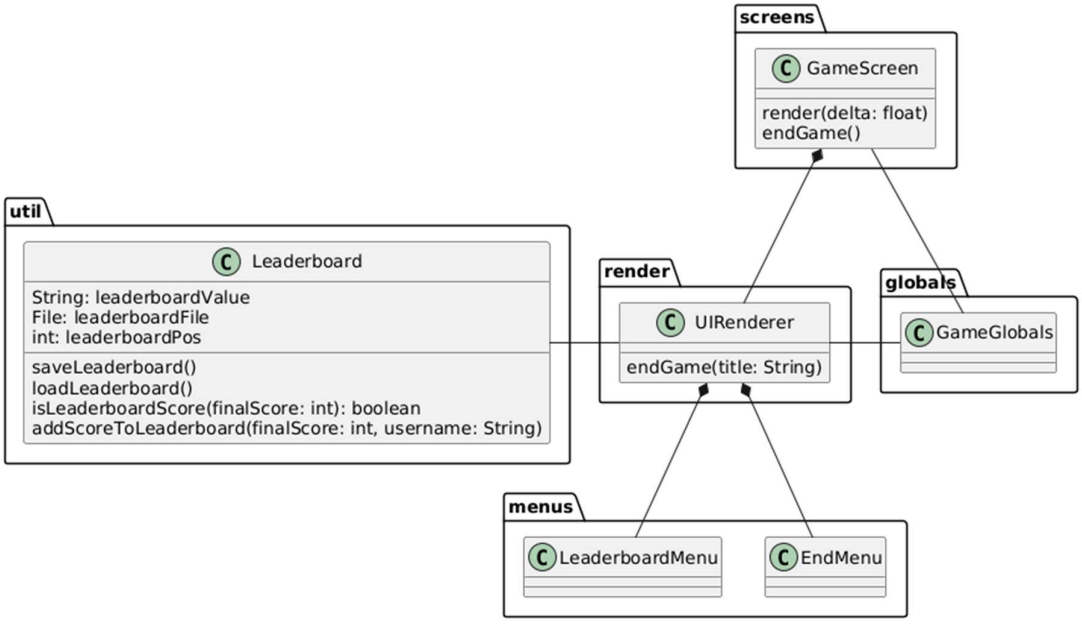

Figure 11: Class diagram showing how the leaderboard is maintained and displayed

In order to meet requirements FR_LEADERBOARD, UR_LEADERBOARD and FR_SAVES, the class diagram above has been implemented in the game application.

In this diagram, methods from the static class Leaderboard are called by the UIRenderer to maintain and save a leaderboard of users to a text file. Then, when GameScreen detects that the current playthrough of the game has ended (by checking the in-game time in GameGlobals), it calls methods in UIRenderer. These methods trigger the assembly and formatting of a LeaderboardMenu instance by using the methods in the Leaderboard class to retrieve an up-to-date version of the current leaderboard. This is then displayed on the screen, alongside the game end menu (EndMenu), for users to view.

The EndMenu instance itself is also assembled within the same method of UIRenderer through using game metrics from GameGlobals. If student satisfaction is above 50% and the current player is not in debt, the end screen shows them that they have won, therefore meeting requirements FR_END_SCREEN and FR_WIN.

Key Metrics

The structure and behaviour of the game architecture tracking and managing key game metrics, such as in-game time, student satisfaction and money, is explained below.

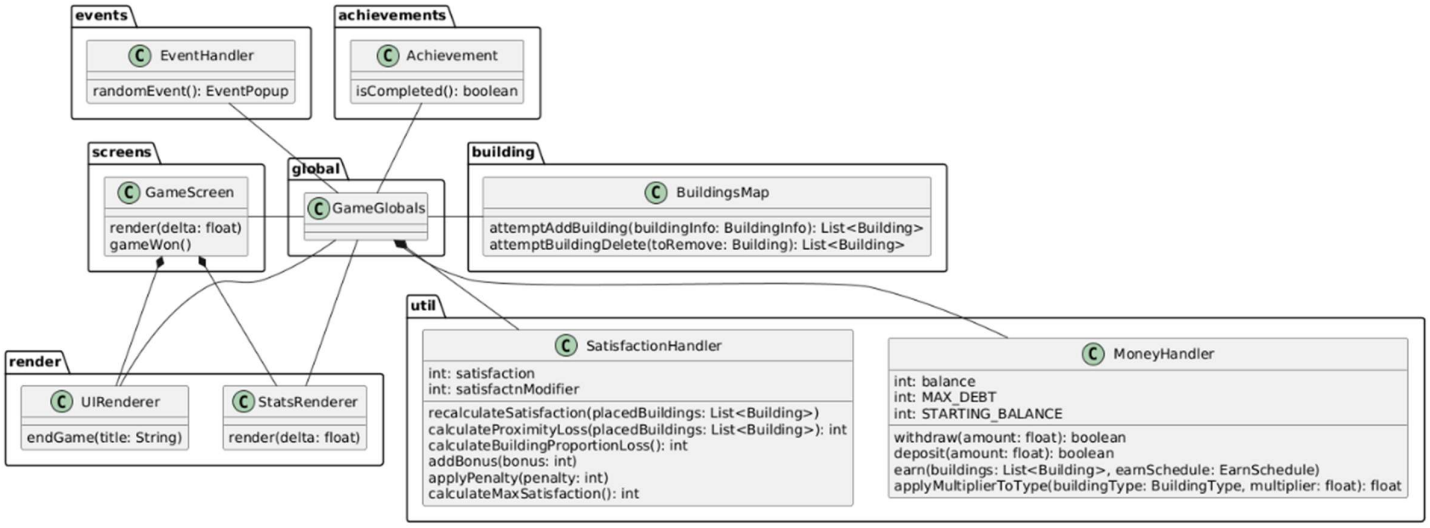

Figure 12: Class diagram showing how money and satisfaction are manipulated

Student Satisfaction

Student satisfaction is stored as an integer within the class SatisfactionHandler (that is stored in GameGlobals) in order to meet UR_SATISFACTION.

This SatisfactionHandler class is used to control the modification of satisfaction, calculating it by using the satisfaction multipliers of each existing building (accessed from BuildingsMap within GameGlobals).

Student satisfaction can also be accessed by both the EventHandler and Achievement classes in order to check whether achievements can be unlocked and to modify satisfaction as a result of an event occurring. This therefore helps to satisfy both UR_EVENTS and UR_ACHIEVEMENTS.

Satisfaction is also accessed by several rendering classes throughout each playthrough of the game. StatsRenderer constantly retrieves the metric from GameGlobals to display on the screen, while both GameScreen and UIRenderer use it to modify the end game screen and determine whether the player has won or lost.

Money

In order to satisfy UR_FINANCE, money is stored in the class MoneyHandler - in a similar way to satisfaction - and can be manipulated by calling the relevant methods in this class. For example, the method earn() is called periodically to calculate the amount of money earned by all existing buildings in the current playthrough.

Money can be accessed and modified by the EventHandler, Achievement and StatsRenderer classes in the same manner as student satisfaction. It is used by GameScreen in the same way as well – to determine whether the player has won or lost.

Time

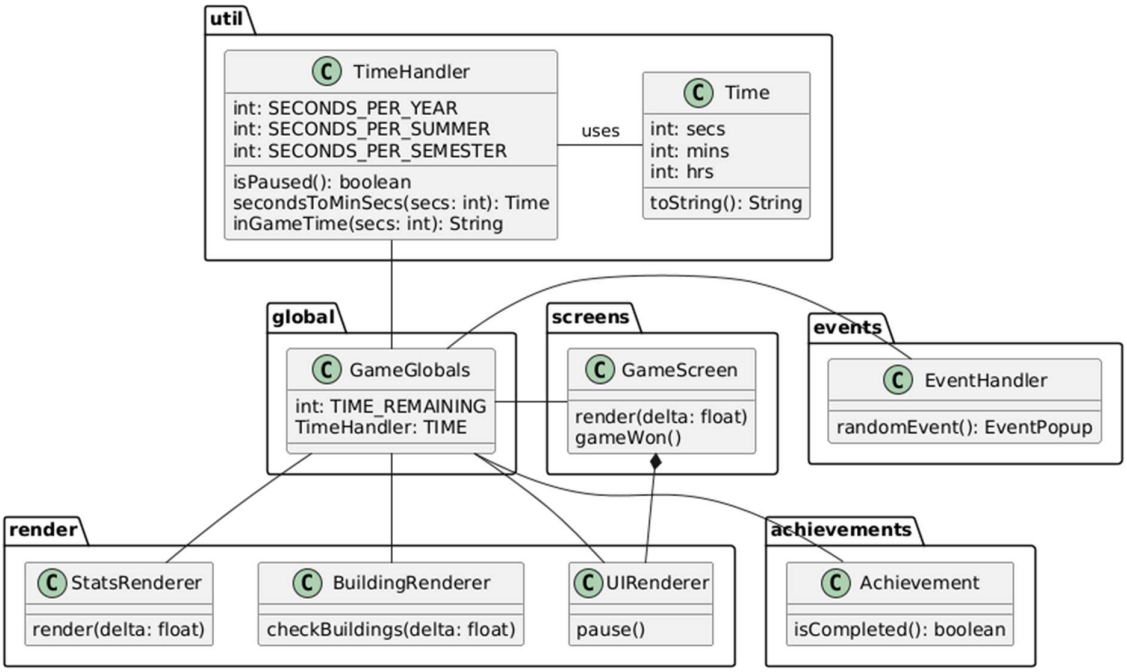

Figure 13: Class diagram showing how time is tracked and updated

In order to meet both UR_TIME and FR_TIMER, the remaining time for the current playthrough is stored as the integer TIME_REMAINING in GameGlobals, alongside an instance of the TimeHandler class - which is used to modify it.

The TimeHandler class uses its methods and the static class Time to perform several operations on the in-game time (TIME_REMAINING) for various classes throughout the game application. These are explained below.

The in-game time is constantly accessed through GameGlobals by both GameScreen and StatsRenderer. StatsRenderer uses the in-game time and TimeHandler to display a live timer (in both real-world and in-game time formats).

GameScreen uses TIME_REMAINING when the render() method is called in order to check whether the game is over (to meet UR_TIME), or if an event must be triggered (to meet UR_EVENTS). If the pause button is pressed, GameScreen calls the relevant methods in TimeHandler and UIRenderer to pause the timer and display the pause menu - therefore satisfying FR_PAUSE - as well.

Also, the class BuildingRenderer checks the in-game in order to monitor the construction of newly placed buildings and ensure that they are ‘built’ in an appropriate amount of time to meet requirement FR_BUILD_TIME.

Finally, the in-game timer is accessed by several Achievement subclasses to check whether those achievements have been unlocked and to help ensure that the UR_ACHIEVEMENTS requirement is met.