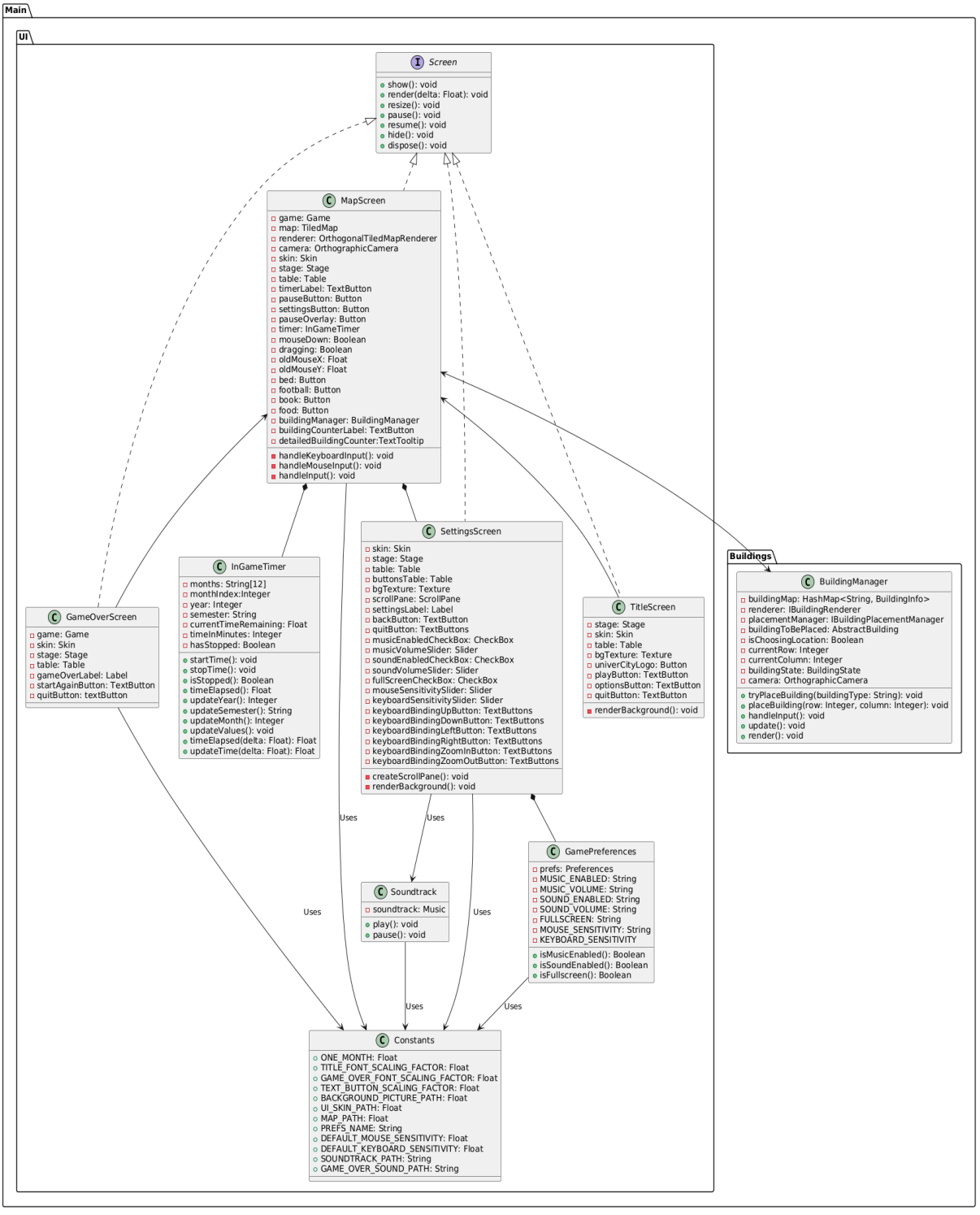

Figure 1: UML class diagram created using plantUML [1] depicting a high-level overview of different entities within the game from the view of the UI package with getter and setter methods omitted as per convention[1].

[1] Note packages are used for diagrammatic use only to aid visualisation of class entities.

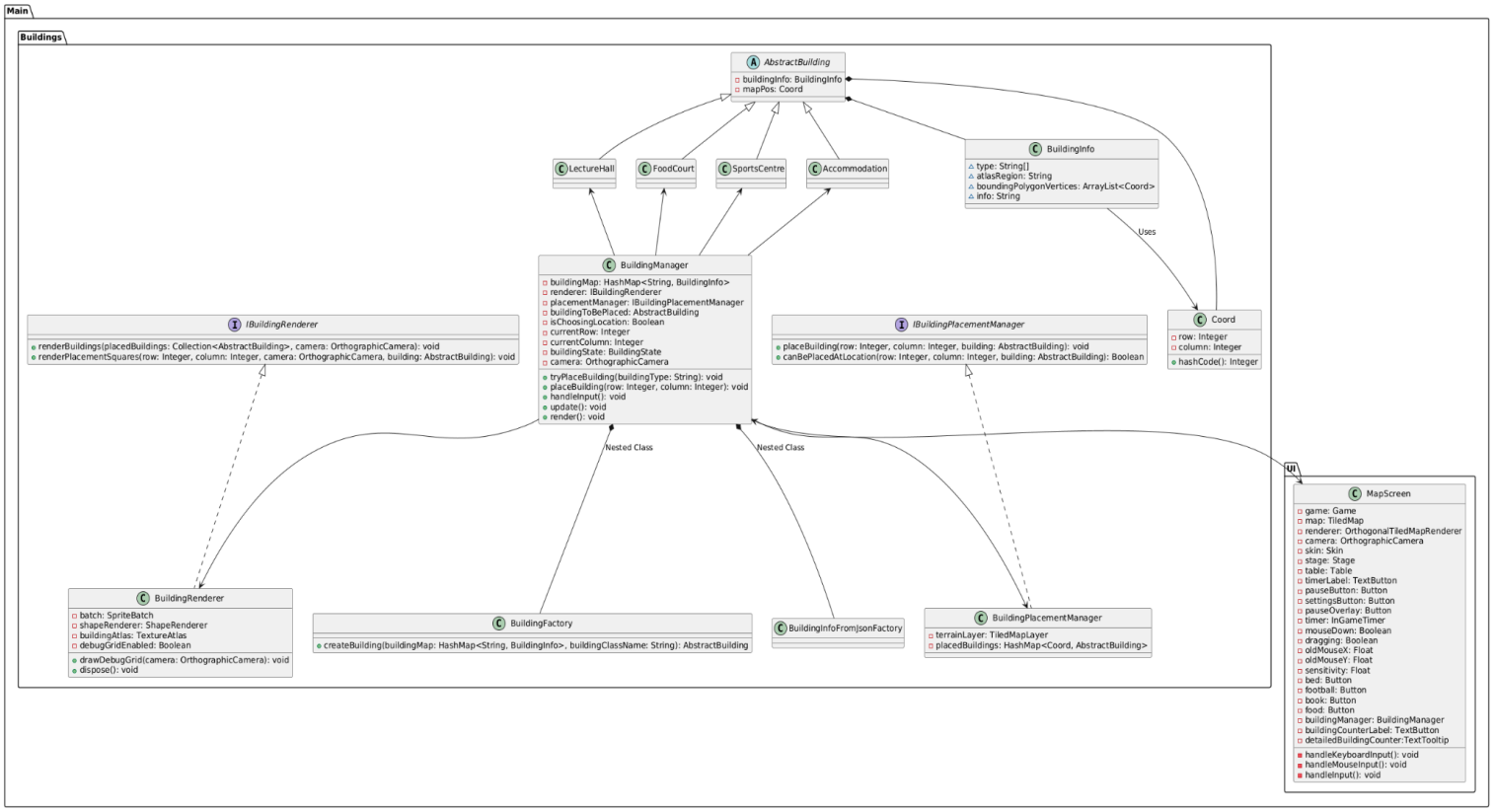

Figure 2: UML Class diagram depicting a high-level overview of the game entities from the perspective of the Building package.

Our final class diagram displays all relations between classes within our game. As a design choice, we opted to utilise interfaces and abstract classes heavily to reduce code duplication and codebase complexity whilst providing improved traceability throughout the different classes. This can be seen in figure 1, with ‘MapScreen’, ‘SettingsScreen’, ‘GameOverScreen’ and ‘TitleScreen’ classes all realising the libGDX ‘Screen’ interface [2], which ensures all necessary methods are implemented across these classes. A similar design choice can be observed in figure 2, with our four building types all inheriting from an ‘AbstractBuilding’ class with ‘BuildingRenderer’ and ‘BuildingPlacementManager’ inheriting from interfaces that we implemented ourselves.

The choice of libGDX as our game engine enabled us to utilise a wide range of libraries provided by the framework. This massively reduced the size of the codebase as libGDX provides libraries for intricate functionalities, including input handling and graphics rendering with the assistance from the OpenGL API [3]. This allowed us to meet the majority of visual and interactive requirements outlined by our client. One feature requested by the client that is not implemented is the desire for the game to start in a paused state. Their rationale behind this request is that the user will have to consider their building placement in response to certain events, however with no events present in the current state, we deemed it unnecessary to start the game paused as the user can place buildings arbitrarily. Furthermore, implementing this feature would break convention as the vast majority of

games begin in a playable state and the system that requires a contrary implementation is not part of the game's current state.

Our initial class design was formed from a textual use case surrounding the building placement functionality of the game, as we identified that this interaction will expose the user to the majority of the game's functionality. This use case was then turned into five simple **CRC cards** which provided an abstraction of how classes would interact as well as a foundation from which to build the implementation of the project.

Earlier iterations of our class designs and relationships can be found **on our website.** Each iteration illustrates some of the ways the constructs in our game changed as functional iterations were observed and reviewed, a selection of which are considered below:

- **Alteration from a minutes:seconds timer system to a semester:month:year timer system:** Our initial timer featured a five minute counter to denote elapse of the game, however this was changed to a semester-based system upon consideration of our client needs. It was their request that the game represent the timetable of a university within a five minute total period. To achieve this we created a simple threshold named ONE_MONTH in the constants class that represents 8.33 seconds of real-time, or one month in-game. This allowed us to record the time elapsed since the last month and update the timer state accordingly. This core change helped satisfy the FR_TIMER_DISPLAY outlined in our **user requirements document.**
- **Introduction of a Constants class:** This class was implemented following the production of our 'MapScreen' and 'InGameTimerClasses'. In initial iterations of our game, a selection of constants were hard coded as primitive values with their purposes outlined by code comments. We felt that a new class should be created to house these constants which would allow descriptive variable names to be applied to them. This change provides more consistent and coherent traceability within our code and should contribute to the maintainability of the game in subsequent iterations.
- **Soundtrack and GamePreferences classes:** As seen from our user requirements, particularly our UR_AUDIO elicitation, our stakeholder proposed the idea of a game soundtrack to supplement the user experience. Upon implementing and testing this class, we deemed it necessary to provide the user with an ability to customise their game preferences, such as adjusting the volume of this soundtrack. By giving our stakeholder more control over and providing accessibility controls, we aim to enhance the desired user experience originally outlined in our requirements meeting.
- **Use of a texture atlas:** The use of a texture atlas was initially introduced in our earliest iteration of the game, featuring only the map screen and timer. Within this earlier iteration, the overheads of rendering were of trivial importance, however, to ensure that our game ran on a variety of basic hardware, as outlined in our NFR_HARDWARE_COMPATABILITY and as requested by our client, we aimed to maintain this usage throughout the later stages of development. We deemed this an appropriate choice, as using a texture atlas reduces the memory allocation for the images used in-game and offers efficient rendering capabilities. This design decision

extends across most classes and ensures our product is accessible for every user's hardware.

- **Colour blind Accessibility:** This issue was of great importance to our project after it was raised by our client in our requirements meeting, with our implementation combining the FR_LOCATION_PLACEABILITY and NFR_COLOURBLIND_ACCESSIBILITY elicitations in our requirements document. This functionality is contained in the BuildingRenderer class, specifically in the renderPlacementSquares() method. When a user selects a valid location to place a building, this is reflected by a green tick in the grid they have selected. In addition, buildings placed by the user have distinct shapes based on their category. These visual cues were implemented in later versions of our product on reflection of our clients excellent suggestions during our meeting.

It can be seen from our final iteration that game logic components are distributed among classes, particularly logic concerned with building placement. Intermediate class designs show a shift from a monolithic BuildingManager class that is responsible for all building logic to a collection of dependent classes that share information with the BuildingManager to produce more modular code, rather than containing vast amounts of functionality in a stand-alone class. This decision was made on a debugging and readability basis as modular code with descriptive method and attribute names is far easier to maintain and modify than a monolithic alternative.

The constructs within our codebase underwent a variety of name changes at different stages of development, for example the generic handleInput() method that features in our **second class diagram** has its functionality split across handleKeyboardInput() and handleMouseInput() in our final release. These changes were implemented following code reviews using GitHub's comment system. By getting objective feedback from each other using this system, we were able to suggest improvements to give our codebase more consistency and traceability.

The architectural style of our game is primarily Object-Oriented, with some similarities to an Entity-Component-System. One similarity from ECS that is present in our game is the use of composition to form functional classes such AbstractBuilding, which is composed of Coord and BuildingInfo. These are not truly ECS conforming however, as these classes do provide methods that operate on their stored data. Our decision to continue development using OOP was a matter of simplicity, as we all had experience with this paradigm. The potential overhead of learning a new paradigm with contrary fundamentals to OOP, composition over inheritance, could have caused problems, especially given the duration of the project and the Agile Methodology adopted. To ensure we were able to provide a product that fully encapsulated the clients needs in a short amount of time, we chose to adopt a paradigm that we felt most comfortable working with.

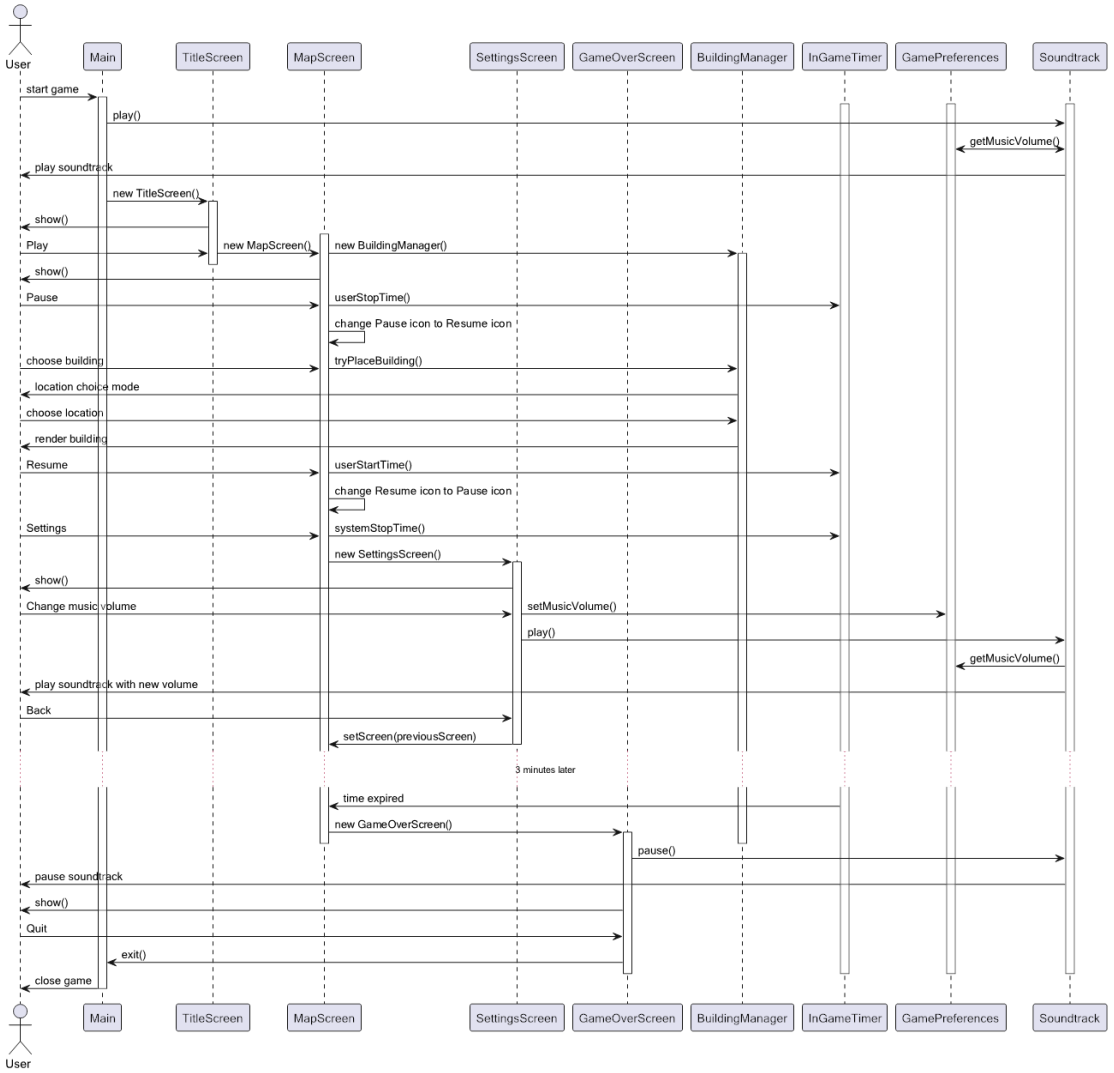

Figure 3: A plantUML sequence diagram displaying interactions between the user and the system.

The interactions between game entities and the user are shown above. It can be seen from the diagram that the user interface consists of the different game screens the user can interact with. The primary user interface for our game is the MapScreen class, which is aggregated from the SettingsScreen and InGameTimer classes as well as providing methods to handle user input. This class also forms a dependency between the BuildingManager to retrieve and display building information and can be thought of as the coordinator for the game, linking all logical components together.

The sequence diagram also demonstrates a feature desired by our client whereby the game is paused, however the user can still place buildings. The reasoning from our client is that it encourages more thoughtful decision making when the ‘student satisfaction’ system is developed in future iterations. This breaks common convention of a pause invoking some secondary screen or menu that prevents the user from continuing until the game is resumed, however we felt the suggestion from the client, and its rationale, warranted its addition in this stage of development, fulfilling the FR_STOP_TIME elicitation in our requirements document.

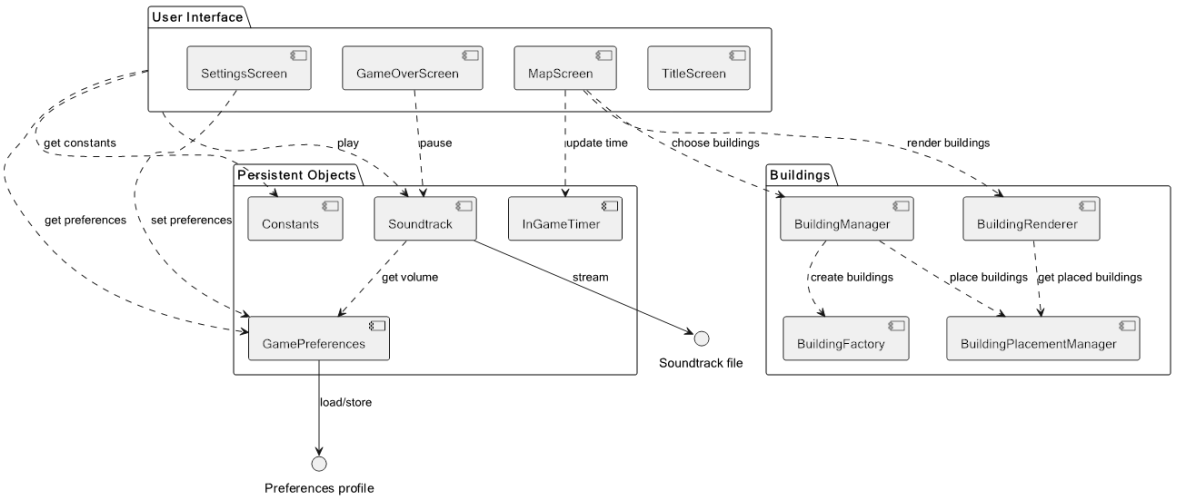

Figure 4: A plantUML component diagram for the game using the recommended style guide proposed by plantUML [4].

Our component diagram adopts the plantUML style guide which uses arrows instead of lollipop notation. We found conflicting examples of component diagrams that conformed to both standards, but to maintain cohesion across our UML modelling system of choice, we opted to follow the guide outlined by plantUML.

An abstracted view of the interactions between components provides further clarity and understanding of the roles and relationships between classes in the game, as well as some of the external dependencies that reside outside of the game's core functionality, such as the saving and loading of game preferences.

Figure 5: A plantUML state diagram displaying a high-level depiction of the varying game states and the events that occur to transition between them.

References

[1] *“PlantUML at a glance”*, plantuml.com, https://plantuml.com/ (accessed 9th November 2024)

[2] *“Extending the simple game”*, libGDX.com, https://libgdx.com/wiki/start/simple-game-extended , (accessed 9th November 2024)

[3] *“OpenGL (ES) Support”*, libGDX.com, https://libgdx.com/wiki/graphics/opengl-es-support , (accessed 9th November 2024)

[4] *“Component Diagram”*, plantuml.com, https://plantuml.com/component-diagram (accessed 9th November 2024)