Architecture

Group 9

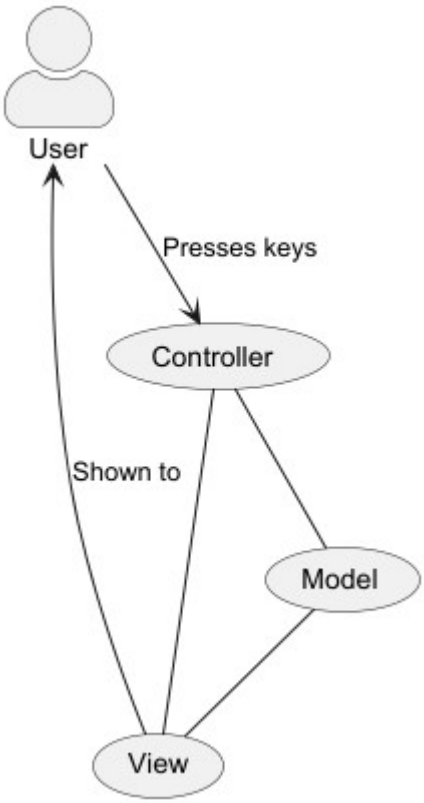
Group name: KOWLAAB

Aiden Sayer
Lurvïsh Polodoo
Ben Salkield
Oliver Rogers
Alex Sharman
William Roebuck
Kiyan Eiles

Architecture

Our primary concern in our architecture design was extensibility so that a different development team could easily pick up where we left off. To achieve this, we decided on a Model-View-Controller architecture. This meant that the core processing of the game (rendering and user input processing) was completely decoupled from the abstract model of the game, allowing it to be extended and improved independently. Furthermore, it provides a useful layer of abstraction in that the game model can be defined by how it should behave in the game world, without the need to consider interaction with the user. We also found that this architecture was well suited to an interactive game in LibGDX due to the way the library facilitates the use of a main process loop in which time-sensitive processing such as rendering and user input must be handled. The "Model" in this case was the game world (the rules concerning buildings, student satisfaction, etc). The "View" was clearly the renderer (or "Main" in LibGDX) where sequence was important to ensure sprites were drawn on the correct layer. Finally, the controller was input from the player, as this would dictate changes in the game world and could immediately affect the view (UI changes).

Figure 1 on https://eng1-cohort2-group9.github.io/Website/ (bottom of web page)



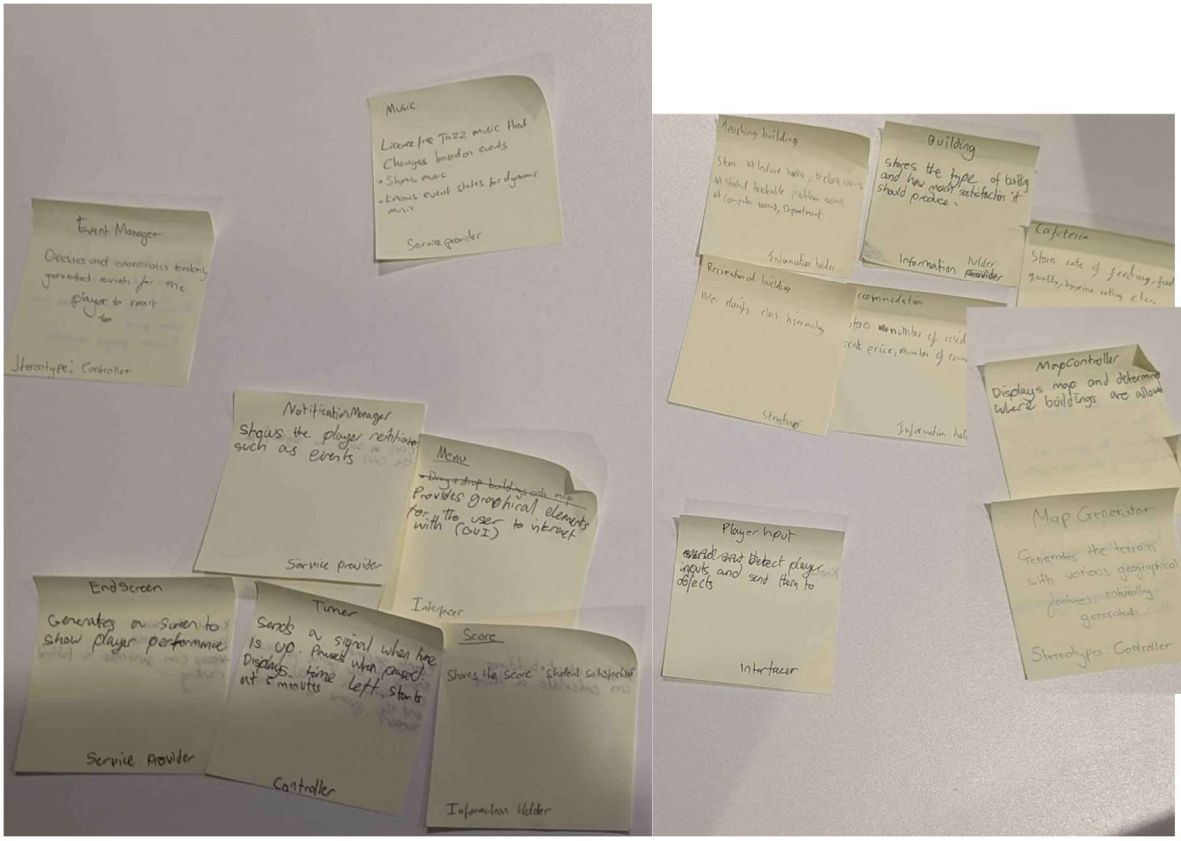
With this in mind, we set about clarifying this model using Responsibility Driven Design. The team chose this methodology as it provided a structured process to transition from high level requirements to a more detailed breakdown of potential classes.

Having already completed step one of RDD during our requirements gathering, we began step 2, creating a designer story:

“Our game will be a fast-paced, enjoyable university simulator, where young adults can build and manage their own university campus, and react to the different events throughout the game.
The player will seek to maximise their student satisfaction, and they can do this by placing different types of buildings. The location of the building will affect its effectiveness and buildings will take time to be constructed and demolished.
Events can change the rules of the game, for example making it so that buildings now produce more satisfaction when near to trees instead of water. The player will be incentivised to react to these events by constructing new buildings or relocating existing ones.
Buildings can be placed on a randomly generated map. This map will include obstacles such as hills, water, or trees. Some of these can block construction in that location.”

Then, as a group, we completed steps 3-7 using post-it notes. The initial set of classes we came up with are shown below. (Image has been edited to fit)

Figures 2, 3 and 4 on https://eng1-cohort2-group9.github.io/Website/



These initial ideas were generated wholly from the requirements.

- **EventManager** - Decides and co-ordinates randomly generated events for the player to react to.

    - From UR_EVENTS
- **Music** - CC0 music that changes based on events
    - From FR_MUSIC
- **NotificationManager** - Shows the player notifications such as events
    - From UR_EVENTS (the player should be shown the events)
- **EndScreen** - Generates a screen to show player performance
    - From UR_YEARLY_REPORTS and UR_END_OF_GAME_REPORT
- **Timer** - Sends a signal when time is up. Pauses when paused. Starts at 5 minutes
    - From UR_TIMING
- **Menu** - Provides graphical elements for the user to interact with
    - From UR_PLAYABILITY
- **Score** - Stores the score
    - From UR_SCORE
- **MapController** - Stores the current map, the current buildings, and controls how the map can be changed
    - From UR_MAP, FR_CONSTRUCTION
- **MapGenerator** - Generates a set of tiles representing the map
    - From FR_MAP_FEATURES
- **PlayerInput** - Receives inputs from the player and sends signals to different parts of the program
    - From UR_PLAYABILITY
- **Building** - Stores the type of building and how much satisfaction it should produce
    - From UR_BUILDINGS
- **Teaching building, recreational building, accommodation, and cafeteria**
    - From UR_BUILDING_VARIANTS

Event manager	
Knows what events there are Knows the events that have taken place Can trigger an event	Event

Menu	
Can display the time Can display the score Can display buttons that the user can press	Timer Score

Timer	
Knows how much time has elapsed Can be paused Can send a signal to end the game	

Music	
Knows what music can be loaded Can determine the relevant track to play based on parameters	EventManager

MapController	
Knows what the map looks like Knows what buildings have been placed Can determine whether a given construction location is valid Can begin map generation Can calculate each building's satisfaction	MapGenerator

Score	
Knows the score Knows what events have taken place Can calculate the current score	EventManager MapController

NotificationManager	
Can convert an event into something to send to the UI	EventManager Menu

MapGenerator	
Knows what kinds of terrain are possible Can randomly generate a map	

Building	
Knows what kind of building it is Knows information about that building (size, time to build, etc)	

EndScreen	
Knows the score Knows the time Can generate a fictional ranking	Timer ScoreCalculator

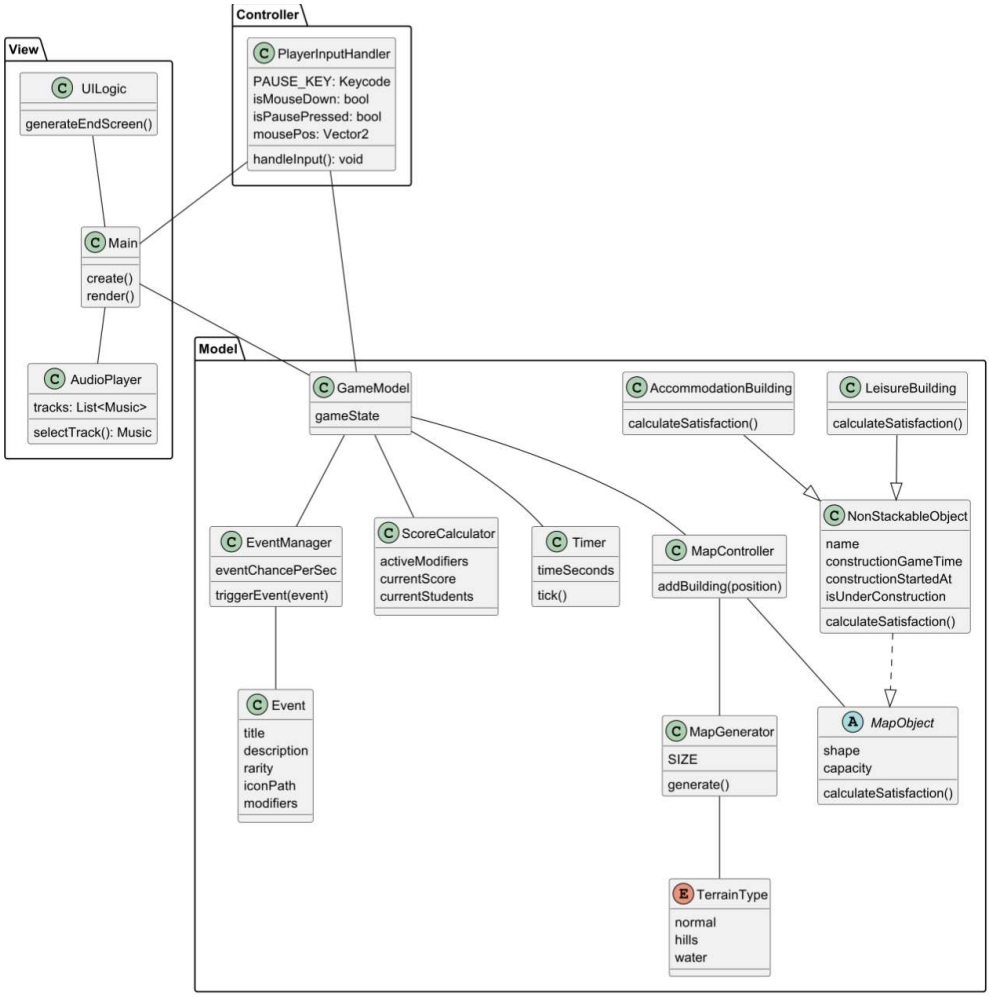
PlayerInput	
Can detect when a player presses a key Can send signals to other parts of the program based on keypresses	MapController

We then developed these ideas further on the reverse of the notes, expanding upon what the objects must "know" and do. These were then formalised as CRC cards digitally.

Figure 5 on https://eng1-cohort2-group9.github.io/Website/

From there, we began to categorise each class as either part of the model, the view, or the controller, with most classes ending up as part of the model. Some names were changed to be consistent with the RDD process, ensuring that every class that was not an information holder (excepting the predefined "Main") was named as a "doer" e.g. **Music** → **AudioPlayer**, and **Score** → **ScoreCalculator**. From this, we could create a more detailed architectural diagram using plantUML. UML was used as it is a well defined, standard format, and thus can be understood by future developers without the need for additional explanation. PlantUML was the perfect tool for this as it provides a way to define diagrams using text which is useful for storage and version control.
Our initial architecture diagram is shown below.

Figure 6 on https://eng1-cohort2-group9.github.io/Website/

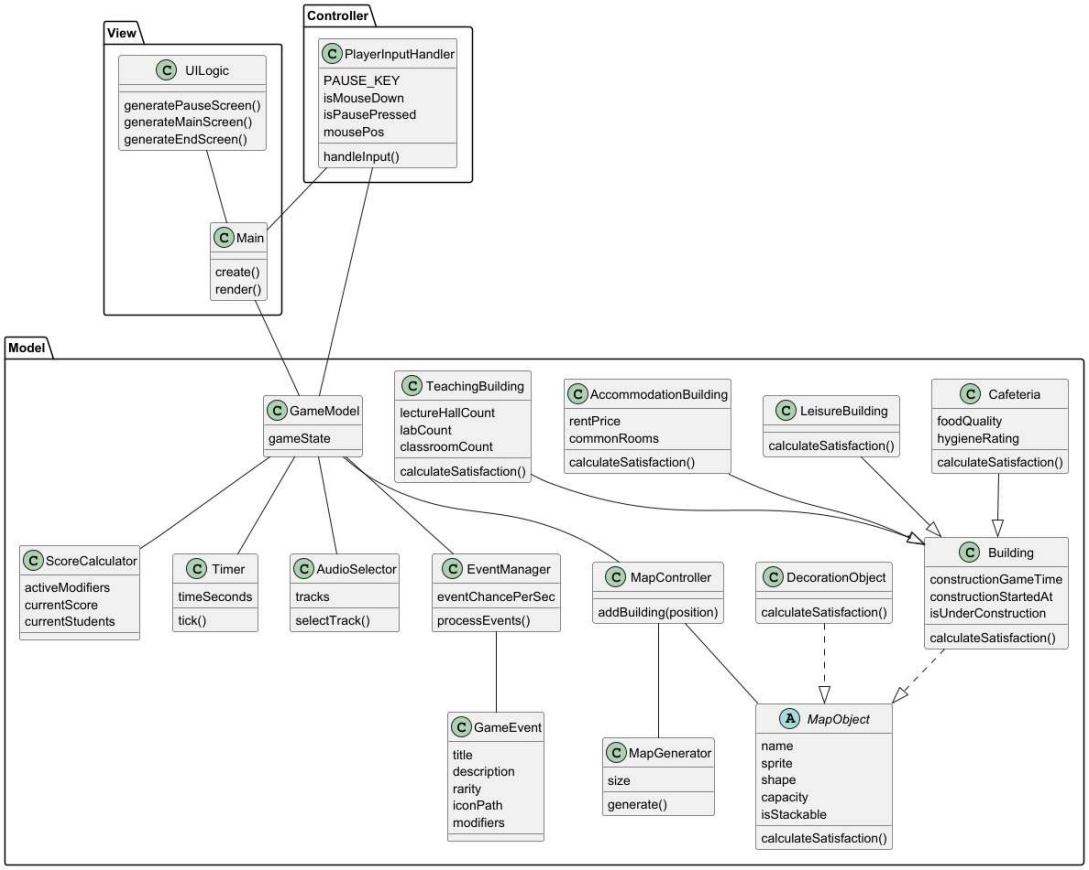


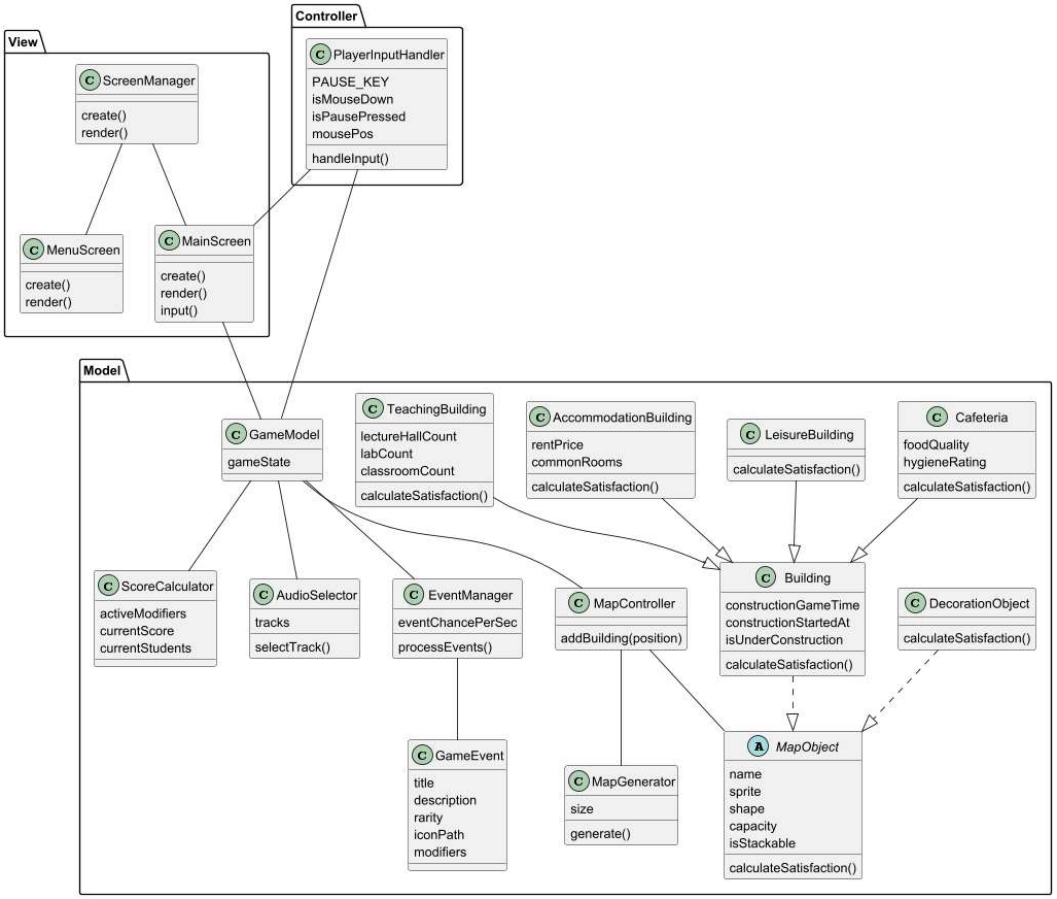
NB: This early version of our architecture diagram contains some errors which were later corrected. Some functions are missing, and low level details such as data types and an Enum are erroneously shown.

As development progressed, we updated and made clearer some parts of the diagram to better fit our program. Some notable changes include:

- Two classes implementing MapObject were developed, NonStackableObject and StackableObject. For clarity, these were renamed to Building and DecorationObject respectively. These were included to satisfy requirements FR_MAP_FEATURES and FR_CONSTRUCTION, imposing restrictions on where buildings can be placed
- Following some investigation into how music might be implemented in LibGDX, AudioPlayer was redefined as AudioSelecter, now inside the game model, providing tracks based on events occurring in the game. The audio playing is now handled inside Main. This implementation continues to satisfy FR_MUSIC
- We discovered a LibGDX pattern called “Screens” and changed our “Main” class to be a “ScreenManager”, with the MainScreen interacting with the Model and Controller.

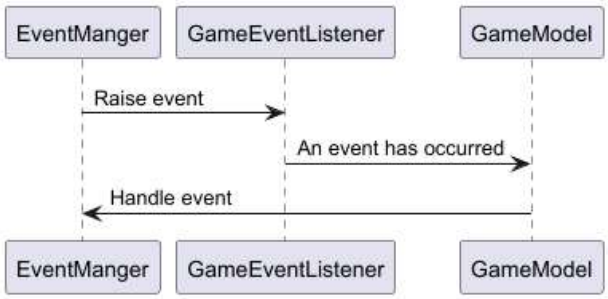
Figure 7 and 8 on https://eng1-cohort2-group9.github.io/Website/

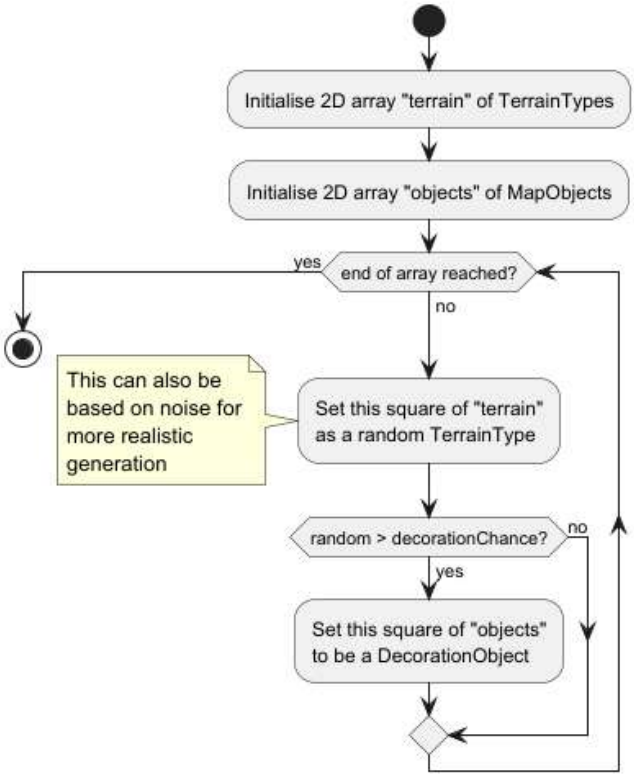




To further clarify our ideas, behavioural diagrams were created for some of the more advanced behaviours of the game, such as event handling and map generation, shown below

Figures 9 and 10 on https://eng1-cohort2-group9.github.io/Website/


EventManger
GameEventListener
GameModel
Raise event
An event has occurred
Handle event
EventManger
GameEventListener
GameModel


Initialise 2D array "terrain" of TerrainTypes
Initialise 2D array "objects" of MapObjects
yes
end of array reached?
no
This can also be based on noise for more realistic generation
Set this square of "terrain" as a random TerrainType
random > decorationChance?
no
yes
Set this square of "objects" to be a DecorationObject