Architecture

Group Number: Cohort 2 Team 7

Group Name: pickNmix

Group Member Names:

David Lun

Sameer Minhas

Harry Muir

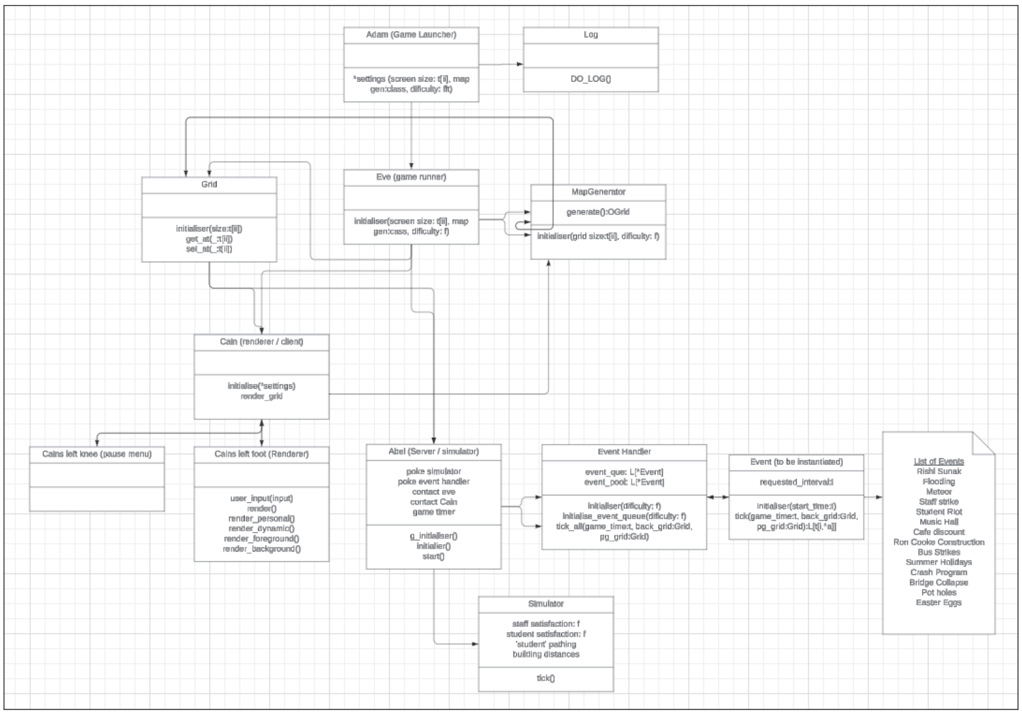
Phyo Lin

Alex McBride

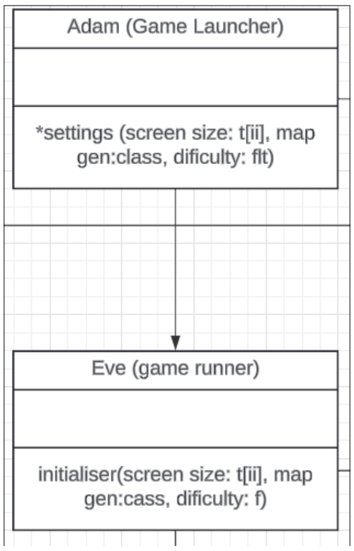
Architecture

The architecture design graphs make use of Lucidcharts to allow for the representation of the different classes and their respective methods and interactions in a visual way.

Architecture Design:



This design was created as a foundation for the minimum viable product. New additions and changes will be done later when holes in this architecture are found but this will be the start. Each of the classes were discussed and designed before we decided to use Lucidcharts to create a proper graphing of the architecture. Each class will have justification of its existence and functionality in accordance with the requirements we gathered and set, so to show where each applies the use of IDs after the reasoning will be shown like (**UR_MAP**).

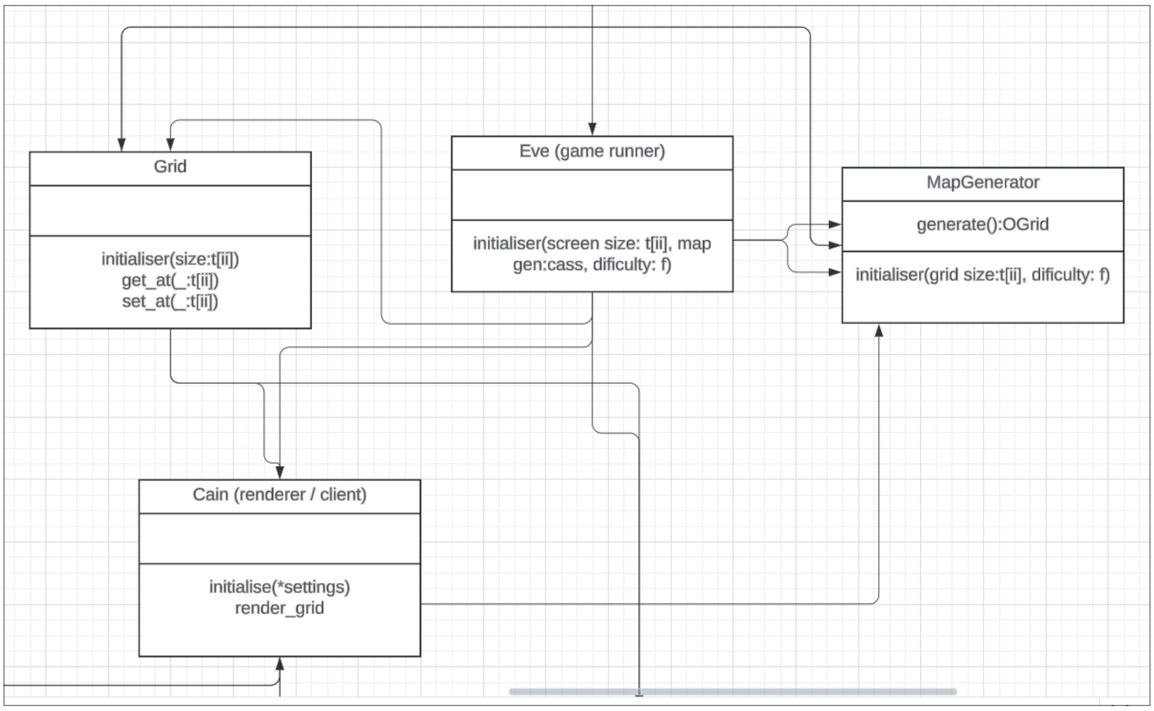

Adam (Game Launcher)
*settings (screen size: t[ii], map gen:class, dificulty: flt)
Eve (game runner)
initialiser(screen size: t[ii], map gen:cass, dificulty: f)

Let's start with the first two classes we decided on, the launcher and then the runner which will start the other processing aspects of the game.

Firstly Adam (the game launcher) will handle all the initial calculations needed and send the data to Eve (the game runner). Adam will handle such information like screen size (**NFR_FULLSCREEN**), map (**UR_MAP**), difficulty (**UR_DIFFICULTY**) as well as other data needed for the game to operate through ‘settings’. We wanted one class to handle processing of required information that's needed at game boot, so processing won’t be done during play time to improve performance (**NFR_PERFORMANCE**) and improving stability due to less processing needed during play time (**NFR_STABILITY**).

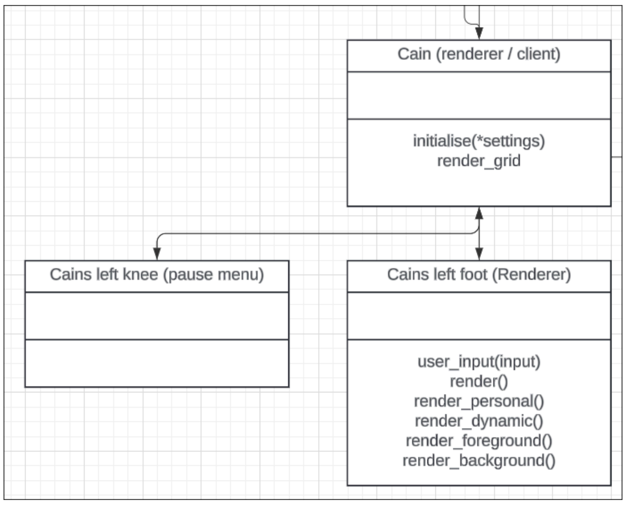
Screen size and map are both needed to allow for a clear way to generate such things like the map grid (**FR_MAP_INFO**) and how much to scale it by to fit the screen size (**FR_DISPLAY**). Difficulty is something while not specified in the requirements brief, was stated by the customer that they would like an option in the future (although they do say it's not necessary for the minimum viable product) (**UR_DIFFICULTY**). To allow this implementation in the future we decided that a way to handle difficulty is to scale it off of a float value which would be used to adjust such things like event frequency as well as other values which will cause the difficulty to increase.

When Adam finishes the ‘settings’ function and has the data needed it gets passed to Eve where it can start the ‘initialiser’ which takes in the returned data from Adam.



Eve initialises multiple different classes and their respective initialisation functions. We first discussed the ‘MapGenerator’ and ‘Grid’ classes as well as Cain (the renderer / client) and Abel (not on the above screenshot but it handles the simulation / server). The Cain and Abel classes are more complex so we brainstormed their functionality later.

The MapGenerator class takes in the grid size and difficulty values for the initialiser. This class generates the Grid class using the values it takes in from Eve. The Grid class just takes in a size parameter from the MapGenerator and fills it out with different tiles to create the map that also includes lakes or hills which restricts building placement (**FR_CONSTRAINTS**). The game is going to have a tiled based map so getting information about a certain position on the map is needed so we gave it a getter and setter function for finding what is at a certain position on the grid, this information will be used for the placing and removing of buildings (**UR_BUILDING**).

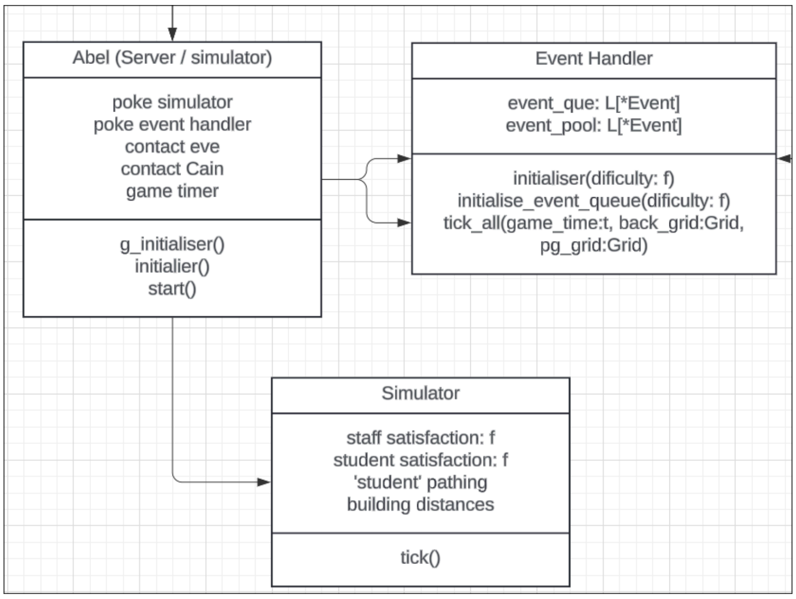


Cain takes in the initialised grid class and settings from Eve. This will create a render_grid class which will be handled by ‘Cains left foot’ class. This renders via the different render functions each of the different grid layers which each contain different tiles and data used in the game.

The respective grids that will be rendered are:

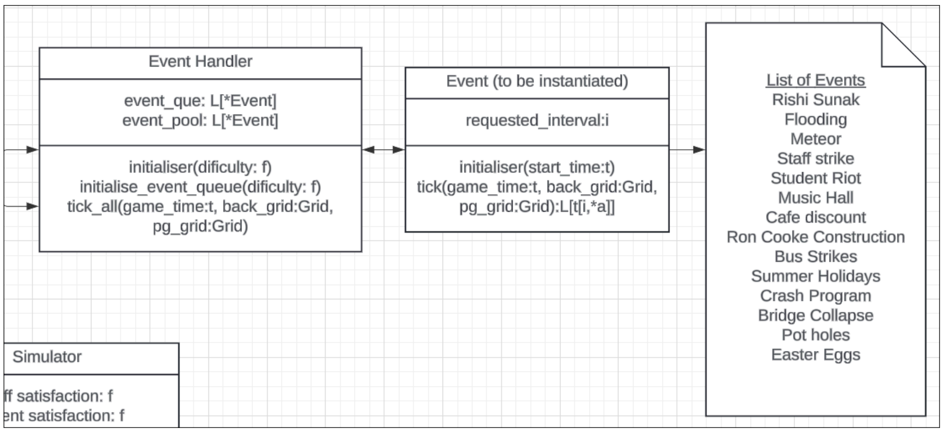
- The background contains the basic ground and river / lake tiles which will be checked on user input to see if a building is allowed to be built on the tile (**FR_CONSTRAINTS**).
- The foreground will contain the road tiles.
- The dynamic layer renders the placed buildings and other things like small student models moving on the map (at a later point of development and not for the MVP) (**UR_BUILDING**).
- Personal layer will render things like mini effects (**UR_UX**) and context menu for details on a certain building tile (**UR_UI**).

The user input which affects the rendered grids will be sent back to the superclass Cain where it sends the MapGenerator class if needed. We also created an empty subclass for Cain to be our pause menu for when we need to implement it (**FR_UI_PAUSE_MENU**), for the MVP it isn’t needed so for later on.



Finally, Eve 'pokes' the Abel class (the server / simulator), starting the simulator and the event handler for the game. We wanted this class to be run on a thread for the duration of the game dealing with the aforementioned simulator and event handler on a thread, with the other thread handled by the Libgdx function and the renderer.

The Simulator subclass will constantly be ticking for every second of the games duration (5 minutes) (**FR_GAME_DURATION**), calculating the student satisfaction (**UR_STUDENT_SATISFACTION**) and on the requirement of the customer staff satisfaction. Student pathing building distances will be used for the gameplay loop of having the player try to have the highest possible student satisfaction which will require the tracking of the placement of the different buildings they have placed on the grid map. The calculated student satisfaction will be shown to the player and used as a score metric of sorts at the end of the game's duration (**UR_STUDENT_SATISFACTION**).



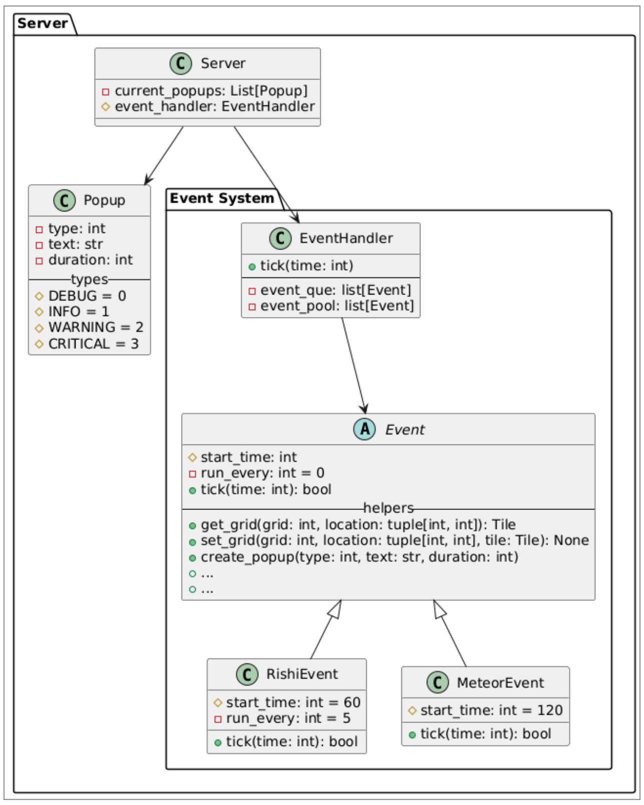
The 'Event Handler' subclass will take in a pool of events and their respective order queue calculated by the Adam loader class via the 'initialise_event_queue' class (**UR_EVENTS**). We decided how events were going to be implemented was to have it have a start time and a duration, when the start time of the first event in the queue is reached on the in-game timer it will be initialised as a new event class (**FR_EVENT_TYPES**).

To account for difficulty since we implemented it as a float value we can use it to adjust the time gap between events to allow time for the player to prepare for the next event whenever it may be (**FR_EVENT_REACTION)**.

Events may affect the map so the event handler takes in specifically the background grid and the foreground grid. Changes to the map or any other values to do with the gameplay loop will require Abel to 'contact_eve' or 'contact_cain' in order to account for the changes.

Architecture Changes and Additions:

There have been new changes and additions to the architecture during the implementation.



Firstly, the server class has been set up to handle popups, which are stored on a list of class 'Popup'. Each popup will have a type like 'INFO' or 'DEBUG'. This will be used as a means for the server to communicate information to the end user, like if an event has happened or an error occurred in the server's processing.

The EventHandler will work similar to how it has before, however the Event class itself has been overhauled. The 'requested_interval' attribute has been changed to 'run_every' which by default is 0 as most events only need to run once over a certain duration. There is an explicit 'start_time' for the 'tick' function to return a boolean value when the tick parameter from the EventHandler is within.

The event handler has helpers functions which allows the events to affect things like the map grid or create a popup for the server to display.