ENG1 Assessment 1: Architecture

Greenfield Development

Group 5

Members

Mark Faizi

Ben Faulkner

Christian Foister

Lewis Hanson

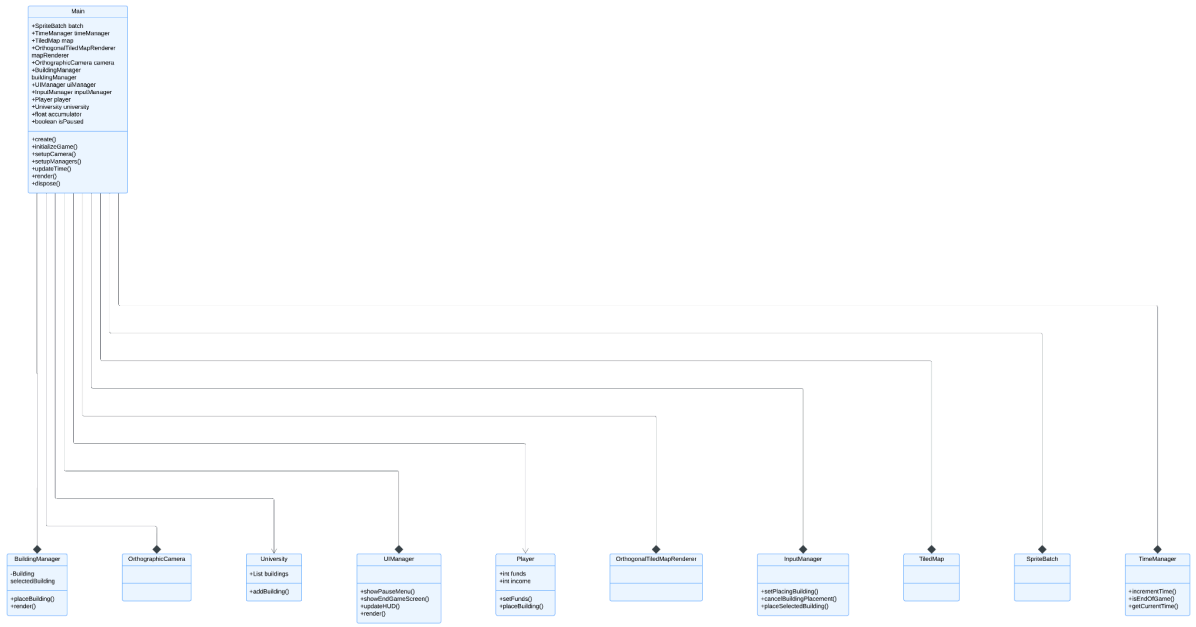
Ziyad Rashad Hassan

James Haworth

UML:

Tool Used:

The tool used for the UML was Lucidchart



Introduction

As detailed in our requirements the product that we are producing is a 2D top down University simulator game where the player controls a university by placing buildings and handling random events that will impact the university in some way. The game is to be made within Java17 and we have chosen to use LibGDX based on the vast amount of readily available learning material for this library. The key metrics that will be used to measure the success of the player is student satisfaction as well as financial health. Financial health encompasses the funds available to the player as well as the income of the university. These do not directly affect the score of the player however it does affect the buildings that can be placed.

Satisfaction may be directly impacted by both events and buildings that are placed by the player, the decisions that the player makes will have a direct impact on the metrics detailed above. They will not have full control however as events in some way will introduce randomness into the university simulator, and based on the reaction of the player there will be a greater or lower effect on the game metrics.

We have used a modular design, this separates classes that we have used in the game such as the player, buildings and events. This implementation allows us to easily

maintain and scale the code while keeping it flexible, giving us the ability to change specific components of each class while maintaining functionality in the overall game. Examples of these classes include the player class, this will hold the main game actions that a player can take such as input handling as well as some key game metrics such as satisfaction and funds. Another class, university, contains the buildings that can and will be placed in the game by the player.

Using a modular approach means that by maintaining individual, independent functionality of our components will maintain the overall functionality of the game. This keeps unit tests straightforward and intuitive and allows for simple delegation of tasks in the team based on modules they are responsible for.

For diagrammatic representations of our architecture we have used flowcharts and UML class diagrams. We have used these to demonstrate the static relationships between components by showing the scope each component has as well as how individual methods may depend on each other. Also flow charts allow us to illustrate the dynamic relationships by showing how decisions are made within the game logic. The tools that we have used are Lucidchart and draw.io both of these have the ability to collaborate with the team on diagrams as well as draw.io has the ability to use directly within our IDE meaning that we can have the architectural diagrams on hand while we develop our University simulator.

Requirements mapping

Functional requirements

R ID	Requirement Description	Rationale
R1	Campus map with geographical features for building placement.	Ensures buildings are positioned within the constraints of geographical features.
R2	Placeable buildings for classes, accommodations, food, and recreation.	Manages building types and allows them to be placed on the map according to player choice.
R3	Minimum of three events requiring player intervention during gameplay.	Adds interactive gameplay with varied responses impacting metrics, ensuring dynamic gameplay.

R4	Track and display student satisfaction, influenced by building placement and event responses.	Provides real-time feedback on player actions and helps measure player success.
R5	Top-down, 2D view for campus visualisation and interaction.	Supports the game's visual and interactive layout in line with gameplay requirements.
R6	Game duration of 5 minutes, representing a multi-year simulation, with pause functionality.	Ensures timed simulation of campus lifecycle while allowing user control over gameplay flow.
R7	Building proximity effects on student satisfaction.	Encourages strategic placement, influencing gameplay dynamics and outcomes.
R9	Game starts paused, with the ability to pause at any time.	Enhances player control, ensuring flexibility during gameplay.
R10	Event frequency balanced for engagement without overwhelming players.	Adjusts gameplay difficulty and event intervals for optimal player engagement.
R12	Scoring system based on campus efficiency and student satisfaction.	Evaluates player performance, giving clear objectives and feedback.

Non functional requirements

R ID	Requirement Description	Architectural Component(s)	Rationale
R8	Visual representation of student activities and satisfaction (e.g., icons, animations).	UIManager, VisualEffects Manager	Provides intuitive and engaging visual feedback, enhancing user experience.
R11	Maintain a light-hearted tone, with no violent or dark themes.	EventManage r, ContentFilter	Ensures all events and visual elements align with a light-hearted game tone.
R13	Accessibility support, avoiding sole reliance on colour; include sound and visual cues.	AccessibilityM anager, UIManager	Improves game inclusivity, meeting accessibility standards.
R14	Optional difficulty settings for event frequency and scoring harshness adjustments.	DifficultyAdju ster, EventSchedul er	Enhances user experience by allowing customization of game difficulty.
R15	Optional building upgrades to improve capacity or quality, affecting student satisfaction.	BuildingUpgra der,	Allows extended gameplay depth and progression for users seeking enhanced challenge.

		StudentSatisfaction	

Structural diagrams

As can be seen from our structural diagrams we have gone for a modular approach where we have some main classes such as player, building and event as well as some manager classes which handle different aspects of the game which are more abstract than what can be seen on the screen such as a timeManager and a buildingManager. These each contain their own individual methods that may interact with other classes to allow for functionality throughout the game.

Examples of this modular approach can be seen by our separation of the player and the university logic. The player is a separate entity to the university as the player holds information about their score and their funds available to them. These are the things that the player has direct access to and control over. Whereas the University class holds its own logic to add buildings as it manages the Building class.

The university class has a one to many relationship with the Building class as the for each university (this is abstractly speaking, there will only be one university per game) will manage multiple buildings. However, a time when we will use one to one relationships is the main class will only hold a single instance of a player or university for example as for each game we will only have one player and one university.

Having such an approach allows us to be flexible and scale the game throughout the development process. This will allow us to start with the very basics of creating the necessary classes required for the game such as player and building before building on top of it the necessary logic for a player to interface with the underlying foundations. It also allows us to easily increase the number of buildings that we want in the game and give them their own personal attributes and effects that they will have on the overall game. This allows us to increase the interest and complexity of the game for the user without increasing the complexity of code for the developers.

There is a point however, where modularity can become overused, in this game there will be a lot of mathematical functions that will be used across the entire game. For example mathematical functions will need to be used to work out where a building is to be placed or how much to increase the satisfaction by. We have chosen not to give our mathematical functions their own classes as there is a minimal likelihood that these functions will need to be reused in different modules of the game, a function for the zoom of the map will not apply elsewhere. Therefore these functions will remain in their parent packages.

Also an evolution is evident from the diagrams, throughout our iterations of the diagrams that we have been using to aid our implementation we have made sure to support a flexible and maintainable design. Our iterations improve our ability to build on top of what we already have and improve the ability of our classes to interact with one another while maintaining their own identities. We have also streamlined our code and therefore diagrams so that we have the highest clarity and efficiency for our architecture.

Behavioural diagrams

Our game, via our requirements, will be a real time game. Therefore we need some form of loop which will underpin the entire game so that we can have a constant flow and increments of time. This will allow the game to change states regularly without any form of player input, this will make the game more immersive and will maintain the users interest. Also this means that we can increment the stats such as funds available to the player to keep them interested and forcing them to make on the fly decisions. Also this allows for the inclusion of events in the future giving each event a probability and then on each increment of time using random chance to decide whether an event will occur or not. This does not need to be done for the first assessment however highlights that our structural and behavioural models are based around continual improvement and scaling of the game.

Related to the use of the real time aspect of the game are the different circumstances in which time will be paused. The ability to pause the game is necessary in real time games as a player may be unable to continue the game but may want to maintain their progress. The use of a timeManager class allows us to handle the methods in a way that will allow us to implement a pause functionality to stop the time incrementing and the game state from updating. As well as this there must be an end game function, this is easy to implement again as it goes alongside the real time functions, simply when the time hits 0 the game ends and the game in basic terms, pauses, and displays a statistics screen at the end.

Another important behaviour that the game has is the funds and satisfaction “scores” that are relevant to the way that the player will play the game. These statistics affect what the player can do and what the player should do respectively. These behaviours are controlled by the player class as the player is who these are directly relevant to. These aspects of the game will be checked at multiple stages such as when the player wants to place a building and then based on this check will determine what will happen to these statistics. This is indicative of a feedback loop for the player. This will in turn give the player things to do throughout the entirety of the game and maintains the requirement driven approach to the design.

A lot of other potentially more complex behaviours are controlled by the libGDX library. An example of these behaviours are handling a player's input. By using a reputable library that has a lot of learning material for it we are able to use libGDX effectively to reuse components that are already created. This keeps the development simpler and much easier to understand as the majority of the code can be centred around the functionality within the game and the game logic that surrounds it, rather than any of the lower levels controls for the game such as rendering the map or handling inputs can be left to our game library.

In terms of the map there is very little behaviour that it is involved with. However using tiled we will be able to create objects to determine where buildings can be placed and where they cannot be. A simple check using libGDX's polygon detection will allow for the game to determine whether an input is inside or outside of the available areas. We have kept this accessible for the user by creating the map in such a way that it is clear where a building can be placed and where it cannot be placed. This keeps the gameplay intuitive for the player as we aim to be user driven and requirement driven in our development.

Conclusion

In conclusion our requirements gave us the opportunity to create a 2d real time top down university simulator game. Throughout our architecture we have endeavoured to do so while also giving the player the best experience possible while being driven by the requirements that we have been given whether they are functional or non-functional. This has led us to create a structure which will align with the vision of the customer while maximising the engagement of the player.

Throughout the development process we have ensured that we have the cleanest and most effective architecture for the game while maintaining the ability to scale, maintain and extend the functionality of our game. This has been achieved by using a modular approach having classes that interact with each other in ways that separate their functionalities into intuitive and easily understandable partitions. This has evolved throughout our development and continual testing of the design and structure we have refined the architecture of the game and has resulted in a successful final design. Therefore, we have ended up with a blend of a structured process and also an interactive and dynamic architecture to align with our requirements.