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

Architecture Style

The architectural style of the application follows two different styles for the game itself and the GUI.

For implementing the GUI, the team used the Object Oriented Programming (OOP) architecture style in order to break down the implementation of the GUI into several classes. This decision was made in order to keep the interactions between different aspects of the GUI both simple and controlled through information hiding and the encapsulation of data within classes.

The team used the Entity Component System (ECS) architectural style to represent the data and logic of the game itself. In this architecture, entity classes store information about key concepts within the game using components (which are classes that only contain attributes). The functionality of these entities is then implemented through system classes, which use the components within entities to implement their behaviour.

ECS architecture was used for this system as it helped split the game's behaviour (and the associated data) into clear and independent sections by linking specific systems and entities. This therefore would increase code readability and the maintainability of each aspect of the game's functionality.

Furthermore, due to the nature of ECS, the game's functionality can be easily modified or extended in response to changing requirements as each entity is made up of reusable components. This means that entities could be modified, destroyed or created without impacting the systems that interact with them as those interactions are based solely on which components an entity contains. As a result of this, the implementation of the game is much more flexible.

Design Process

The design of this system was first developed using the responsibility driven design (RDD) process in order to create an initial model for the system which could then be mapped to UML class diagrams and implemented after being further refined to correspond with ECS architecture.

The team chose to follow the RDD process for creating an initial model of the system as it isolated design from implementation, focusing on "what actions must get accomplished", not "How each action is accomplished" [1, p.2]. This therefore enabled objects' responsibilities and their interactions to be modelled in detail at a very high level, therefore maximising abstraction and preserving flexibility as is described in [2]. The RDD process is also iterative by design, so "offers techniques that fit with and complement agile practices" [3, p.373] and therefore works well with the team's chosen development methodology.

Responsibility Driven Design (RDD) process

The RDD process that was undertaken is described below. It is based on the steps described in "A Brief Tour of RDD" by Wirfs-Brock [2].

First, the team interviewed the stakeholder as part of eliciting requirements for the system, and then used the data collected from that interview to create a list of user, functional and non-functional requirements.

Next, the team used these requirements, together with the extra information gained from the stakeholder interview, to write a brief designer story. This designer story consisted of a brief, high-level summary of the game the system implements and its functionality.

The designer story can be found on the team's website at Designer Story: https://alltheeb5t.github.io/

Once it had been written, the designer story was then analysed to derive a set of themes linked to the game, from which the team could create candidate objects. Candidate objects describe important concepts from the designer story, have unique responsibilities and can be categorised into role stereotypes which describe their function.

In order to perform this process, the team used Candidate, Responsibilities, Collaborators (CRC) cards to describe the purpose and list the role stereotypes of each object. These cards were then organised into groups of related objects, which were reviewed to identify and eliminate redundant ones. Potential relationships and interactions between different objects were also then identified so that collaborators could be added to the cards.

The CRC cards created by the group can be found on the team's website at CRC Cards: https://alltheeb5t.github.io/

Finally, the process of creating and refining candidate objects was then repeated several times, until the team was satisfied with the outcome, in order to create as streamlined and comprehensive a model as possible that could accommodate all requirements.

Design evolution with UML class diagrams

The model created during the RDD process was then mapped to UML class diagrams to create initial models of the relationships between different objects within the system, which were then revised several times to further simplify.

After this, the model representing the game logic was converted into a format conforming to the rules of the ECS architecture style. Classes were split into entities and systems, where entities stored the information held by those classes (using new component classes to group and represent them), and systems held the functionality of those classes.

This ECS architecture model, as well as the OOP architecture model for the UI design and libGDX functionality, were both then revised several times over the course of the development process due to the iterative nature of the Agile methodology used by the team.

The class diagrams for the initial model from the RDD process; the initial ECS and OOP models; and their revisions can be found on the team's website at Design Evolution: https://alltheeb5t.github.io/

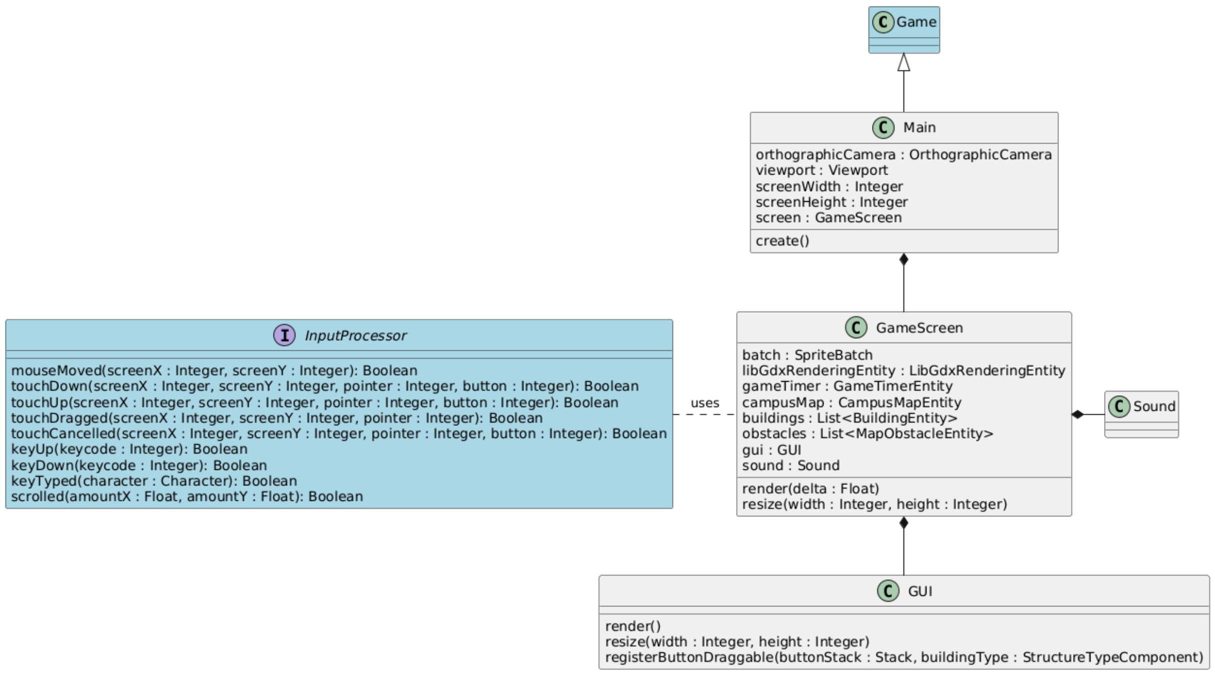
Diagrams and links to requirements

Languages and tools used

To create diagrammatic representations of the system's structure and behaviour, the team chose to use UML class and state diagrams. These were created using PlantUML, a free, open-source tool that enables the generation of UML diagrams from textual input. The PlantUML Gizmo Extension on Google Docs also enabled diagrams to be generated and inserted into documents conveniently.

In the diagrams below, classes representing entities have been coloured in grey, and classes and interfaces from the libGDX library have been coloured in blue to increase clarity.

<u>Overall Game Class diagram</u>

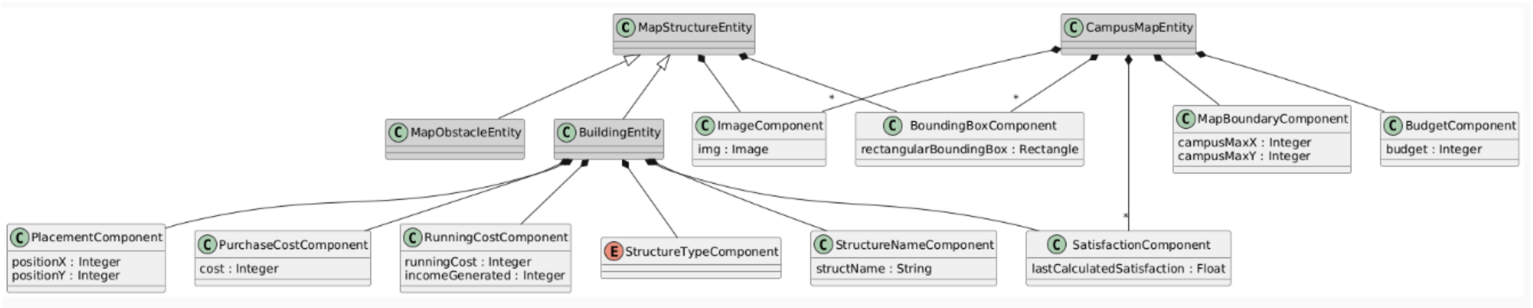

Game
Main
orthographicCamera : OrthographicCamera
viewport : Viewport
screenWidth : Integer
screenHeight : Integer
screen : GameScreen
create()
InputProcessor
mouseMoved(screenX : Integer, screenY : Integer): Boolean
touchDown(screenX : Integer, screenY : Integer, pointer : Integer, button : Integer): Boolean
touchUp(screenX : Integer, screenY : Integer, pointer : Integer, button : Integer): Boolean
touchDragged(screenX : Integer, screenY : Integer, pointer : Integer): Boolean
touchCancelled(screenX : Integer, screenY : Integer, pointer : Integer, button : Integer): Boolean
keyUp(keycode : Integer): Boolean
keyDown(keycode : Integer): Boolean
keyTyped(character : Character): Boolean
scrolled(amountX : Float, amountY : Float): Boolean
uses
GameScreen
batch : SpriteBatch
libGdxRenderingEntity : LibGdxRenderingEntity
gameTimer : GameTimerEntity
campusMap : CampusMapEntity
buildings : List<BuildingEntity>
obstacles : List<MapObstacleEntity>
gui : GUI
sound : Sound
render(delta : Float)
resize(width : Integer, height : Integer)
Sound
GUI
render()
resize(width : Integer, height : Integer)
registerButtonDraggable(buttonStack : Stack, buildingType : StructureTypeComponent)

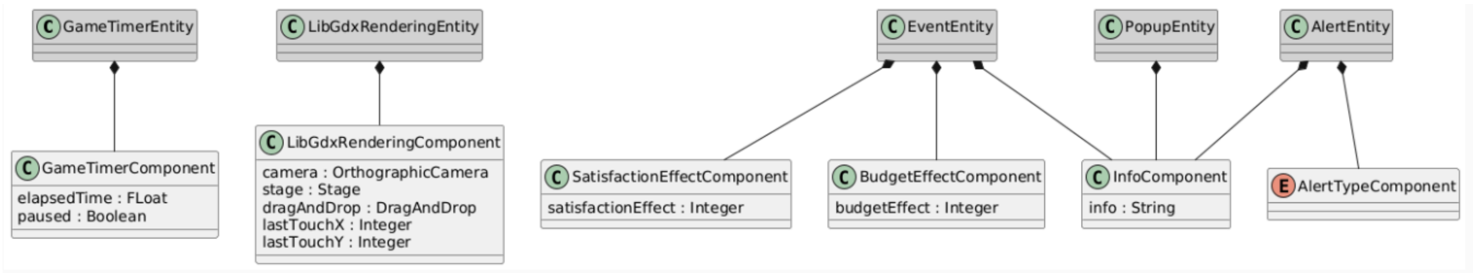
The diagram above represents the overall structure of the game, the classes and interfaces highlighted in blue are from the libGDX library. Once the Main class has started the game, GameScreen and GUI control the screen and respond to users' interactions. This enables the system to meet requirements UR_Audio, UR_Map, UR_Building, UR_Time, UR_Money, UR_Satisfaction, NFR_Scale and NFR_Timer.

The GameScreen class stores all existing entities and uses the methods overridden from the InputProcessor interface to detect user input and call the relevant static methods from system classes to update the affected entities in response.

The GUI class renders the options menu on the screen, which users can interact with to create new buildings and check the in-game time, overall satisfaction level and their current budget.

<u>ECS Class Diagrams</u>



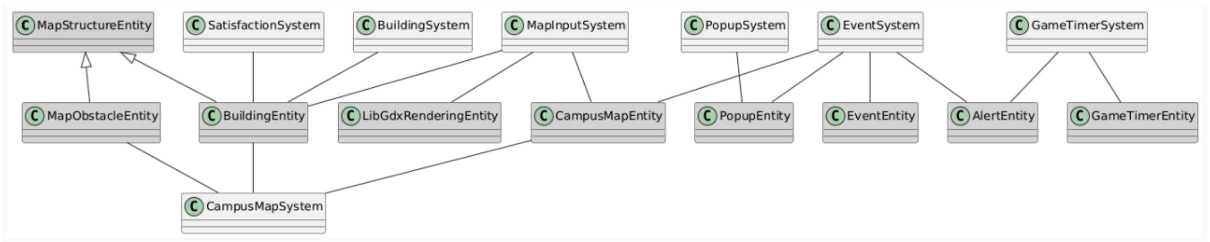


The diagrams above represent the relationships between entity and component classes within the game. Entity classes are coloured in dark grey for clarity.

The MapStructureEntity, BuildingEntity and MapObstacleEntity classes represent university buildings and fixed obstacles - such as rivers, roads or trees - on the map in the game.

The map itself is represented by CampusMapEntity, which tracks the space taken up by structures on the map (by storing the BoundingBoxComponents of each existing structure on the map), enabling the game to meet UR_Map and UR_Buildings. CampusMapEntity also stores the individual satisfaction scores of each existing building (by keeping track of their corresponding SatisfactionComponents), and also keeps track of the current budget as well as the boundaries of the map.

The GameTimerEntity class tracks the current in-game time and also whether the timer has been paused or not, helping to meet UR_Time. This class is used in tandem with AlertEntity and EventEntity to trigger events throughout the game, enabling the system to meet UR_Time, FR_Time and UR_Events, which are then displayed on screen by using PopupEntity.



The diagram above shows the links between entity classes (again highlighted in grey for clarity), and system classes, which manipulate components within the entities to implement the game's functionality in response to user input.

The overall map is controlled by CampusMapSystem, which deals with collision prevention and tracks the budget. Collision prevention is managed by using the BoundingBoxComponent of existing BuildingEntity instances when buildings are placed or moved, helping to meet FR_Place_Building, FR_Move_Building, FR_Remove_Building and FR_Building_Shape. This process is also aided by the MapInputSystem, which handles the drag and drop functionality of buildings, and BuildingSystem, which ensures the images of the buildings on the screen match the position of their BoundingBox.

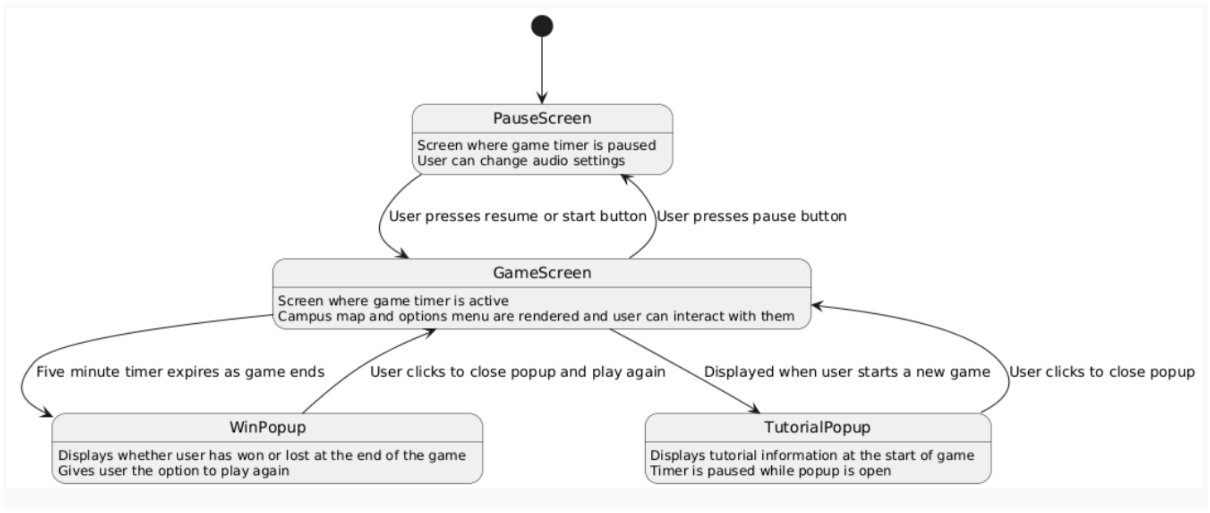
In order to keep track of the user's overall satisfaction score and determine whether they win at the end of the game (implementing requirements UR_Satisfaction, FR_Satisfaction_Score, FR_Building_Interaction and FR_End_Screen), SatisfactionSystem calculates the satisfaction score of each individual building and then aggregates these figures into a single score that is displayed on the screen.

To manage the creation of events during the game to meet requirements UR_Events, FR_Events_Popup, FR_Positive_Events, FR_Neutral_Events and FR_Negative_Events, the GameTimerSystem updates and tracks the GameTimerComponent within GameTimerEntity, generating instances of AlertEntity in order to initiate events.

Once an AlertEntity instance has been initialised, it is detected by EventSystem, which generates an EventEntity instance to represent an event and manages the enactment of its effects on budget and satisfaction scores (using its BudgetEffectComponent and SatisfactionEffectComponent).

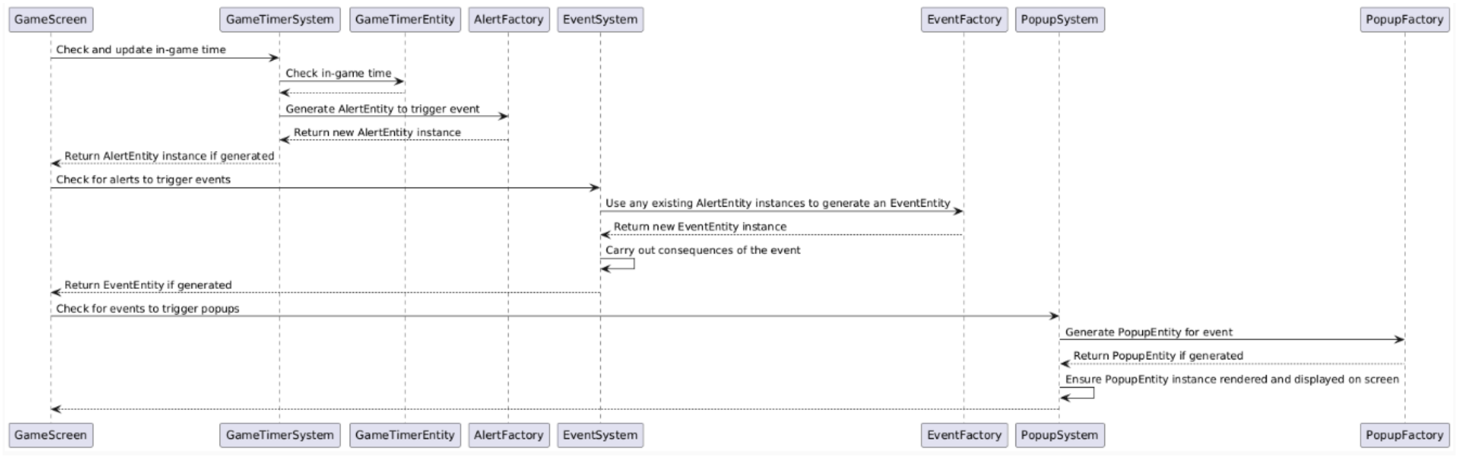
The generation of an EventEntity also causes PopupSystem to generate an instance of PopupEntity, which is then displayed on the screen in order to inform the user about the occurrence of an event.

<u>Behavioural diagrams</u>

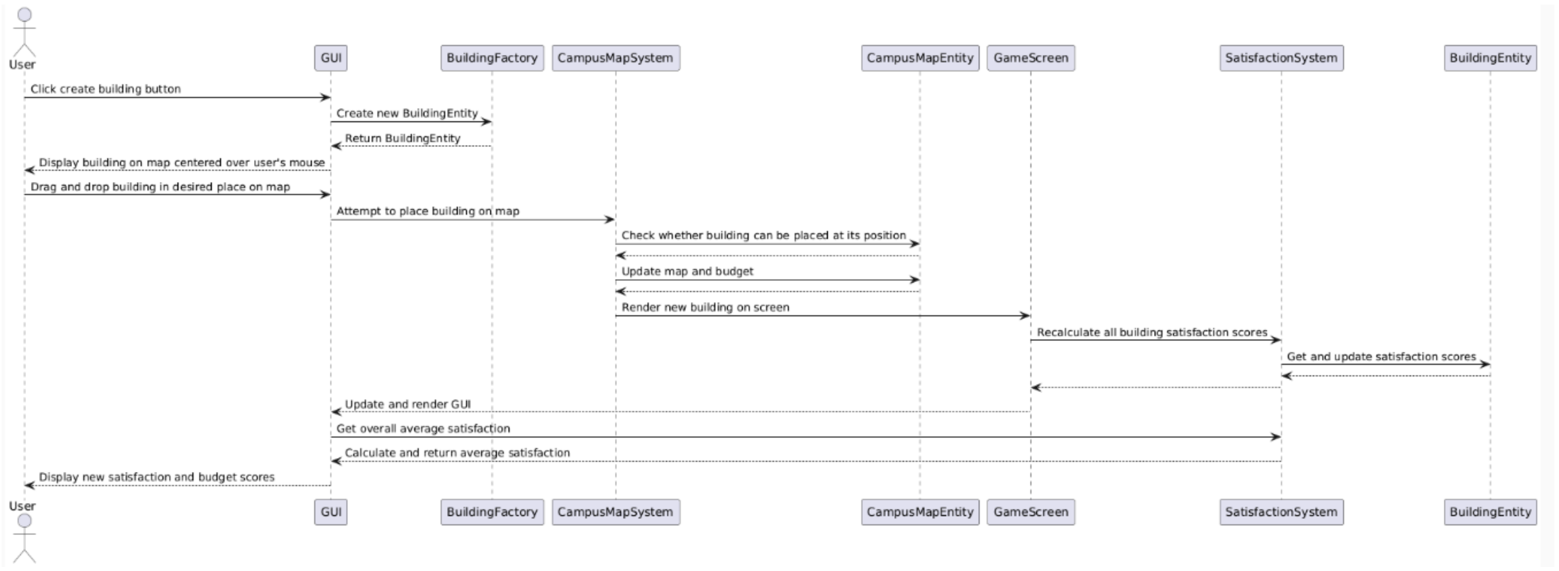

PauseScreen
Screen where game timer is paused
User can change audio settings
User presses resume or start button
User presses pause button
GameScreen
Screen where game timer is active
Campus map and options menu are rendered and user can interact with them
Five minute timer expires as game ends
User clicks to close popup and play again
Displayed when user starts a new game
User clicks to close popup
WinPopup
Displays whether user has won or lost at the end of the game
Gives user the option to play again
TutorialPopup
Displays tutorial information at the start of game
Timer is paused while popup is open

The state diagram above shows the behaviour of the game's UI and how the user can interact with it. This enables the game to meet requirements UR_Map, UR_Audio, UR_Map, UR_Time, UR_Satisfaction, FR_Pause, FR_Pause_Menu, FR_Settings, FR_End_Screen, NFR_Pause_Menu, NFR_End_Screen and NFR_Timer.

The two sequence diagrams below are examples of two important processes carried out during the game. The first represents the process of the in-game time being monitored and updated throughout the game (as well as the game's behaviour when an event is triggered), and the second shows the user creating and placing a building on the map.


GameScreen
GameTimerSystem
GameTimerEntity
AlertFactory
EventSystem
EventFactory
PopupSystem
PopupFactory
Check and update in-game time
Check in-game time
Generate AlertEntity to trigger event
Return new AlertEntity instance
Return AlertEntity instance if generated
Check for alerts to trigger events
Use any existing AlertEntity instances to generate an EventEntity
Return new EventEntity instance
Carry out consequences of the event
Return EventEntity if generated
Check for events to trigger popups
Generate PopupEntity for event
Return PopupEntity if generated
Ensure PopupEntity instance rendered and displayed on screen

The sequence diagram above demonstrates how the game meets requirements UR_Time, FR_TIME, FR_Semesters, UR_Events, FR_Positive_Events, FR_Neutral_Events, FR_Negative_Events and FR_Events_Popup.


User
GUI
BuildingFactory
CampusMapSystem
CampusMapEntity
GameScreen
SatisfactionSystem
BuildingEntity
Click create building button
Create new BuildingEntity
Return BuildingEntity
Display building on map centered over user's mouse
Drag and drop building in desired place on map
Attempt to place building on map
Check whether building can be placed at its position
Update map and budget
Render new building on screen
Recalculate all building satisfaction scores
Get and update satisfaction scores
Update and render GUI
Get overall average satisfaction
Calculate and return average satisfaction
Display new satisfaction and budget scores

The sequence diagram above shows how the game meets requirements FR_Entertainment_Building, FR_Study_Building, FR_Catering_Building, FR_Building_Interaction, FR_Satisfaction_Score, FR_Money_Tracker, FR_Place_Building, NFR_Buildings and NFR_Metrics..

<u>References</u>

[1] R. Wirfs-Brock, “Responsibility-Driven Design.” https://wirfs-brock.com/PDFs/Responsibility-Driven.pdf (accessed Oct. 22, 2024).

[2] Rebecca Wirfs-Brock and Alan McKean, “What is Responsibility-Driven design?,” Wirfs-Brock Associates, Jul. 2003. Accessed: Oct. 22, 2024. [Online]. Available: https://wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf

[3] R. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley Professional, 2002. Accessed: Oct. 22, 2024. [Online]. Available: https://ptgmedia.pearsoncmg.com/imprint_downloads/informit/op/9780201379433_objectdesign.pdf