Architecture

1. Architectural Style

The architecture will adopt a hybrid approach combining **MVC** and **ECS**, allowing components to handle user input and game mechanics while maintaining modular, event-driven interactions.

MVC Pattern

- **Model**: Represents the game's data (campus map, building placement, events, student satisfaction, time).
- **View**: Renders the game's grid, buildings, events, and satisfaction metre.
- **Controller**: Interprets user input (keyboard and mouse) and updates the model or view as required.

Entity-Component-System (ECS)

- **Entity**: Represents game objects like buildings, roads, or events.
- **Component**: Defines individual properties for each entity, such as building type, location, or impact on satisfaction.
- **System**: Manages interactions and logic between components, such as placing buildings on the grid, generating events, and adjusting student satisfaction based on game state.

2. Main Components

1. Game Engine Layer

The game engine is responsible for handling core mechanics like game state, event management, rendering, and user input handling. It will leverage an existing framework, such as **libGDX** or **jMonkeyEngine**, to speed up development and focus on game logic rather than reinventing basic functionality.

- **Grid-based Map**: The map is a two-dimensional grid where users can place buildings and roads (`FR_MAP`). Each grid cell contains either a building or a road, and players can modify building locations (`FR_PLACE_BUILDING`, `FR_MOVE_BUILDING`).
- **Game Timer**: A countdown timer that starts at 5 minutes, pausing when needed (`FR_TIMER`, `FR_COMPLETE_PAUSE`). The time is displayed on the user interface to keep the player informed.
- **Physics/Rendering**: Manages the real-time rendering of objects on the grid and simulates player actions, ensuring a smooth gameplay experience.

2. Map Module

The **Map Module** will represent the campus layout as a grid-based environment. It handles building placement, roads, and environmental constraints like lakes and trees that limit where structures can be placed (UR_PLACE_BUILDING).

- **Placement System**: Handles rules for placing buildings and moving them across the map. Buildings should be placed logically, following the game rules (FR_PLACE_BUILDING).
- **Road System**: Allows players to create roads and paths that connect buildings. The proximity and accessibility of buildings impact student satisfaction (FR_ROAD).

3. Student Satisfaction Module

This module is responsible for calculating and updating student satisfaction based on building placement, event handling, and proximity of facilities (FR_CALCULATE_SATISFACTION).

- **Satisfaction Algorithm**: Uses factors like building proximity (e.g., accommodation near lecture halls, food areas close to student zones) and event outcomes to update student satisfaction (UR_INFLUENCE_SATISFACTION).

4. Event System

The game will include both random and pre-planned events, which will influence the player's strategy in maximising satisfaction. Events may have positive, negative, or neutral effects, and players must react accordingly (FR_EVENTS).

- **Random Event Generator**: Triggers events based on certain intervals or conditions during gameplay.
- **Fixed Events**: These are predictable and happen in all playthroughs, adding consistency to the game.

5. User Interface (UI) System

The user interface provides visual feedback to the player and facilitates interaction with the game. The UI should be intuitive and responsive, allowing users to control the game through both keyboard and mouse inputs (FR_MOUSE_INPUT, FR_KEY_INPUT).

- **Grid Display**: The grid-based map is rendered clearly, with distinguishable buildings and road placements (NFR_CLEAR_GRAPHICS).
- **Hover Information**: Players can hover over buildings to view detailed information, including how it affects student satisfaction (FR_HOVER_INFO).
- **Settings Menu**: Provides options for muting, volume control, and adjusting gameplay experience (FR_MUTE, FR_ALTER_VOLUME, FR_SETTINGS).

6. Input Management

- **Flexible Input**: The system supports both keyboard and mouse controls. It must handle user inputs, whether simultaneous or separate, without conflicts (`UR_FLEXIBLE_UI`, `UR_SHORTCUTS`).
- **Keyboard Shortcuts**: Allow faster navigation and control of the game, enhancing the user experience (`UR_SHORTCUTS`).

3. Additional Features

Data Persistence

The system will store game data using JSON or similar formats to track progress and events. This allows easy storage of game states and retrieval across multiple platforms (`FR_EVENTS`, `FR_OPERATING_SYSTEMS`).

Modularity

Each game component is designed with high cohesion and low coupling, ensuring that individual features (like the map, satisfaction module, and event system) can be easily maintained or updated independently.

Scalability and Accessibility

The game will be accessible across different devices and screen sizes, from laptops to desktop monitors (`NFR_SCALABLE`). The interface adapts dynamically, ensuring a clear and consistent user experience.

4. Technology Stack

- **Language**: The game will use **Java 17**, compatible with frameworks like libGDX, ensuring cross-platform functionality.
- **Rendering Engine**: **libGDX** will handle the 2D rendering for the grid, buildings, roads, and events.
- **Input Handling**: Both keyboard and mouse input will be managed using standard input libraries from the game engine.
- **Data Management**: **JSON** or similar data storage format for events, building states, and player interactions.

5. Architecture Trade-offs

Drawing on concepts from the software architecture lecture, we must consider trade-offs to ensure the architecture supports long-term maintainability and performance:

- **Modularity vs. Performance**: Modularisation increases maintainability but may introduce slight overheads due to component communication.

- **Event-based Communication**: An event-driven approach ensures flexibility but adds complexity in managing event processing and sequencing.

This architecture provides a robust foundation for UniSim, supporting its grid-based design and gameplay mechanics. By combining the MVC and ECS patterns, we ensure a clear separation of concerns and high modularity, making it easier to expand and maintain the game over time. The architecture also addresses key requirements like accessibility, flexibility in input handling, and a user-friendly interface, all while ensuring that the core game experience—building a thriving university campus—is intuitive and engaging.

6. Algorithms

1. Satisfaction Calculation Algorithm

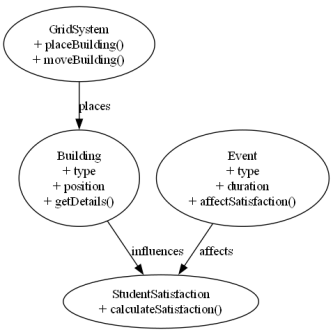
- **Input**: A list of all placed buildings and their proximity to each other.
- **Process**:
    1. For each building, calculate its impact on satisfaction based on its type and location.
    2. Sum the impact of all buildings.
    3. Adjust the satisfaction based on the outcome of any active events.
- **Output**: Updated satisfaction score.

2. Event Trigger Algorithm

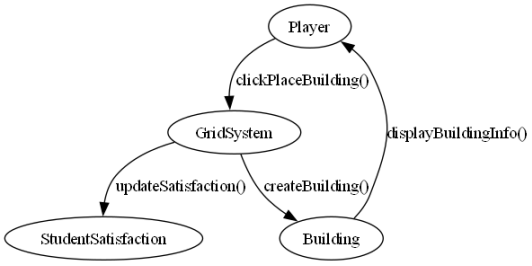
- **Input**: Current game time and random event timer.
- **Process**:
    1. If the timer reaches zero, trigger a random event from the `events` list.
    2. If it's time for a fixed event, trigger the specific event based on the game time.
    3. Resolve the event's impact on satisfaction and display the event to the player.
- **Output**: Updated game state and event log.

6.2 Diagrams

Here are some of the diagrams that we came up with using a Python module named Graphwiz based off the algorithms:



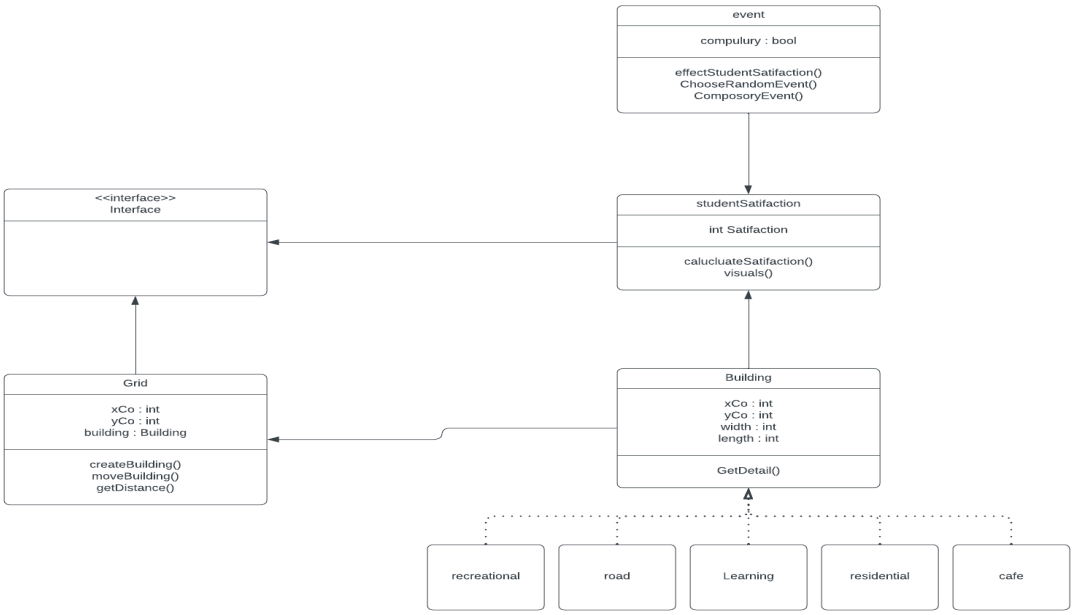
The first diagram is a class diagram showing the relationship between the grid system which represents the map where we can place and move buildings, Buildings which have a type and a position, Events which have a type and duration and can affect the satisfaction and finally the student satisfaction which has one method being calculating satisfaction. The chequered arrow represents the child class of the building class where the other



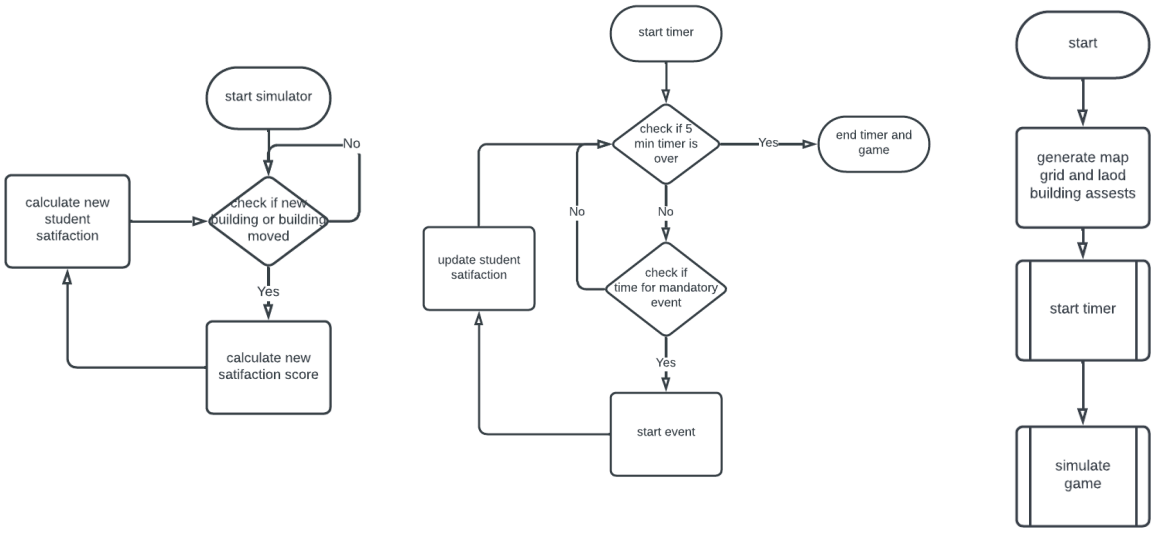
The second diagram shows a sequence of events of how the game should function. It starts with the player who places the game who places a building onto the grid system

which then updates the student satisfaction and then the details of the building are returned to the player.

We then updated the first diagram using Lucid shown earlier and made it into a proper OOP class diagram which then became an ECS where the black arrows represent how each class interacts with each other while the chequered lines represent the parent class of building to its children subclass. We also included a UI entity component These child subclasses all represent the potential objects that can be placed in the game as according to the product brief. This diagram then fulfils the requirements of `UR_INFLUENCE_SATISFACTION, FR_PLACE_BUILDING, FR_MAP, NFR_PLAYABLE.`



These are flow charts that represent sequences of the game. There are three flow charts, one which determines how the timer affects the events during the game and the satisfaction; another for the satisfaction is calculated based on the building proximity and finally one for the main simulation itself.



7. Data Structures

4.1. Grid Representation

- The **Grid System** is represented as a 2D array `(Cell[][] grid) of Cell` objects. Each cell can hold either a building, road, or be empty.
- **JSON Integration**: The grid and its contents such as the buildings and roads can be serialised into JSON format for data persistence.

4.2. Event Management

- Events are stored in a `List`, each event having attributes like `ev`.
- **JSON Integration**: Events are saved as JSON objects, allowing their state to be persisted and loaded during gameplay.