# Architecture

Cohort 3 Group 6 - Carbon Goose

Members: Bailey Horsley, Owen Jones, Rory Ingram, Ken Jacob, Abishek Kingslin Raj, Louis Polwarth, Adam Johnson

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

1) Input: The game world object.
2) Process:
   a) Generate a graph with all buildings as nodes, and the distance between them along pathways as edges.
   b) For each node which is an accommodation building:

i)   Initialise its satisfaction score to zero.

ii)   Check through each non-accommodation building:

    (1) Compare distance between nodes to the "satisfactory distance" and the "ignore distance" of the building category and calculate a distance satisfaction score using exponentiation (Figure 0).

$$d < d_i: \ min(1, \ e^{d_s(d_s-x)/d_i}), \ otherwise: \ 0$$

    (2) Calculate a "quality satisfaction" using the building cost divided by the maximum building cost in that category.

    (3) Combine the distance satisfaction and quality satisfaction by multiplying them, and add this to the "category satisfaction" score for the building's category (food, recreation, etc.), ensuring that it does not exceed 1.

iii)   Set the base satisfaction level for the accommodation to the average of the three category satisfaction scores.

iv)   Calculate and multiply by an "accommodation quality multiplier" which is the accommodation's build cost divided by the average of all possible accommodation building costs, and then normalised to be closer to 1:

$$\frac{2}{\pi} arctan(\frac{p}{p_a}) + 0.5$$

v)   Add any specific modifiers from the game state due to events, globally and local to the accommodation, again ensuring that it does not exceed 1 (100%).

    c)   Calculate the average satisfaction of all accommodation buildings.

   3)   Output: Updated satisfaction score.

Figure 0: The function used to calculate partial satisfaction based on distance, d_s and d_i being satisfactory distance and ignore distance respectively, x is the input path distance between the two buildings.
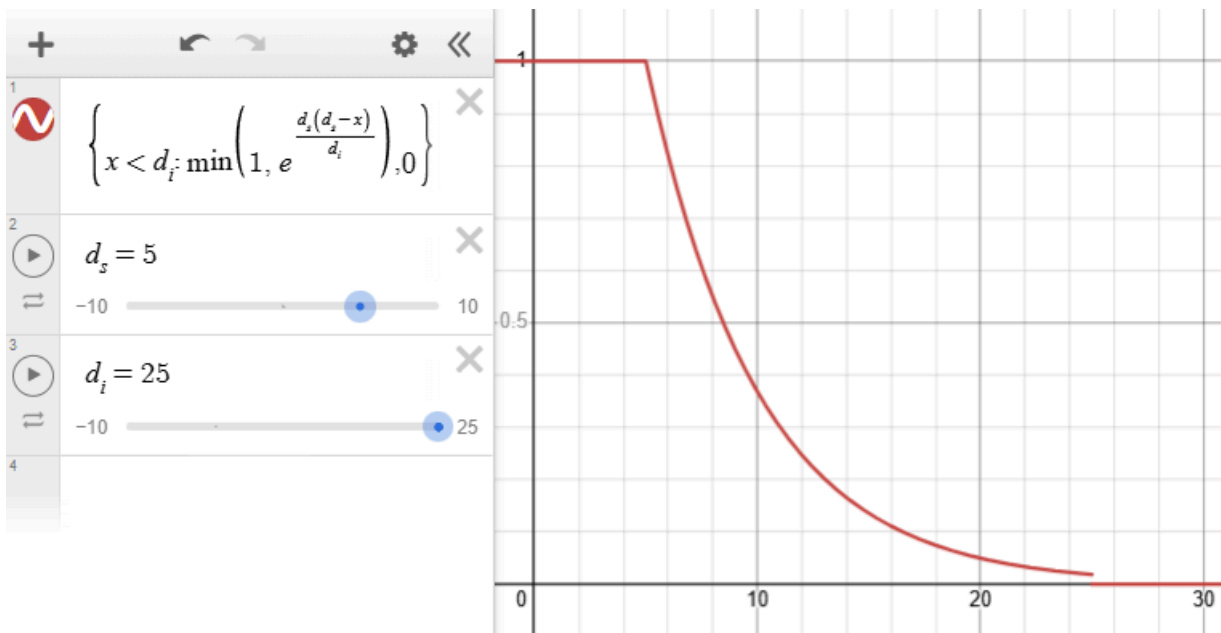
2. Event Trigger Algorithm

- 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.

6.1 Diagrams

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

The first diagram, Figure 1, 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 satisfaction and finally the student satisfaction which has one method being calculating satisfaction. The chequered arrow represents the child class of the building class whereas the second diagram, Figure 2,

Figure 1
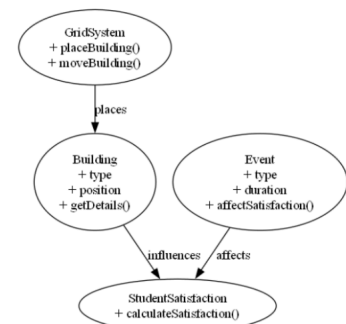
Figure 2
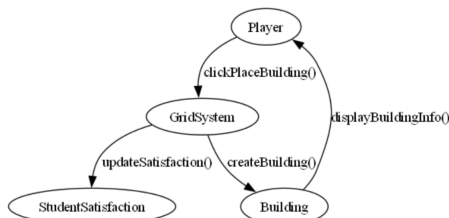
shows a sequence of events of how the game should function. It starts with the player who plays the game, which then places a building onto the grid system, updates the student satisfaction and then the details of the building are returned to the player.

We then updated Figure 1 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 according to the product brief, known as Figure 3. This new diagram then fulfils the requirements of `UR_INFLUENCE_SATISFACTION, FR_PLACE_BUILDING, FR_MAP, NFR_PLAYABLE.`

Figure 3

GameScreen

- mainGame
-UiManager

+ show()
- initialiseEvents()

MainTimer

- timer : timer

+ getTimer()

Tile

- tileType : string
- tileTexture : png
- isBuildable : boolean
- buildingRef : Building

+ getType()
+ isBuildable()
+ setBuildable()
+ getBuildingRef()
+ setBuildingRef()

buildings

- buildingType : string
- position : int
- size : int
- cost : int
- texture : png

+ getSize()
+ getPosition()
+ getBuildingType
+ getCost()
+ getTexture()

Figures 4, 5 and 6 are flow charts that represent sequences of the game. Figure 4 accounts for the satisfaction and how it is calculated based on the building proximity; Figure 5 determines how the timer affects the events during the game and satisfaction; Figure 6 walks through the main simulation itself.

Figure 4

start simulator

No

calculate new
student
satifaction

check if new
building or building
moved

Yes

calculate new
satifaction score

Figure 5

start timer

check if 5
min timer is
over

Yes → end timer and
game

No

No

update student
satifaction

check if
time for mandatory
event

Yes

start event

Figure 6

start

generate map
grid and laod
building assests

start timer

simulate
game
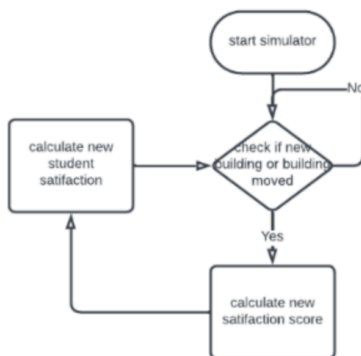
7. Data Structures

7.1. 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.

7.2. 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.

Structural Diagrams

Class Diagrams

Audio Management:
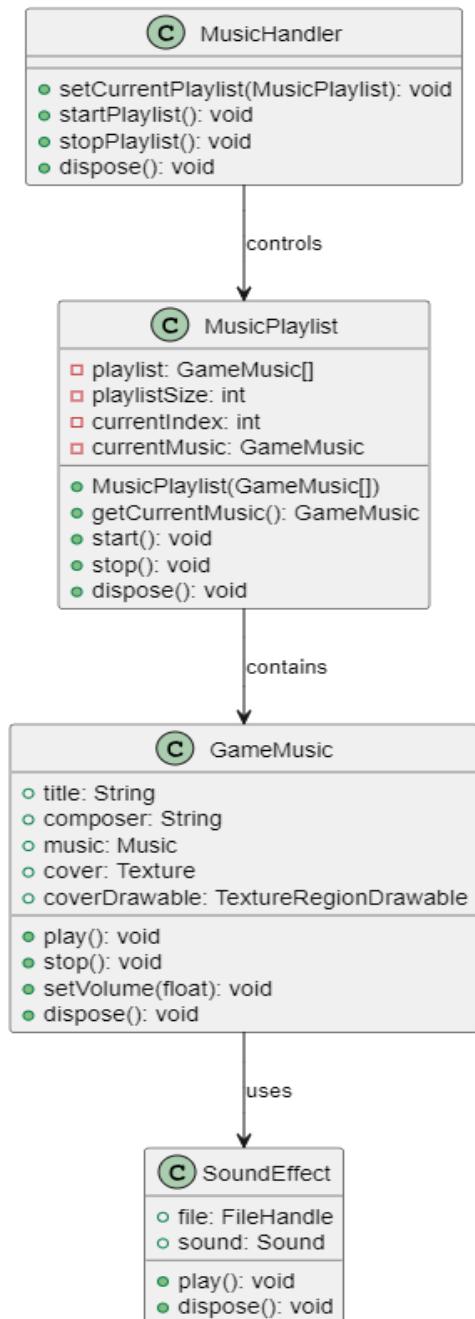
Building and World Management:
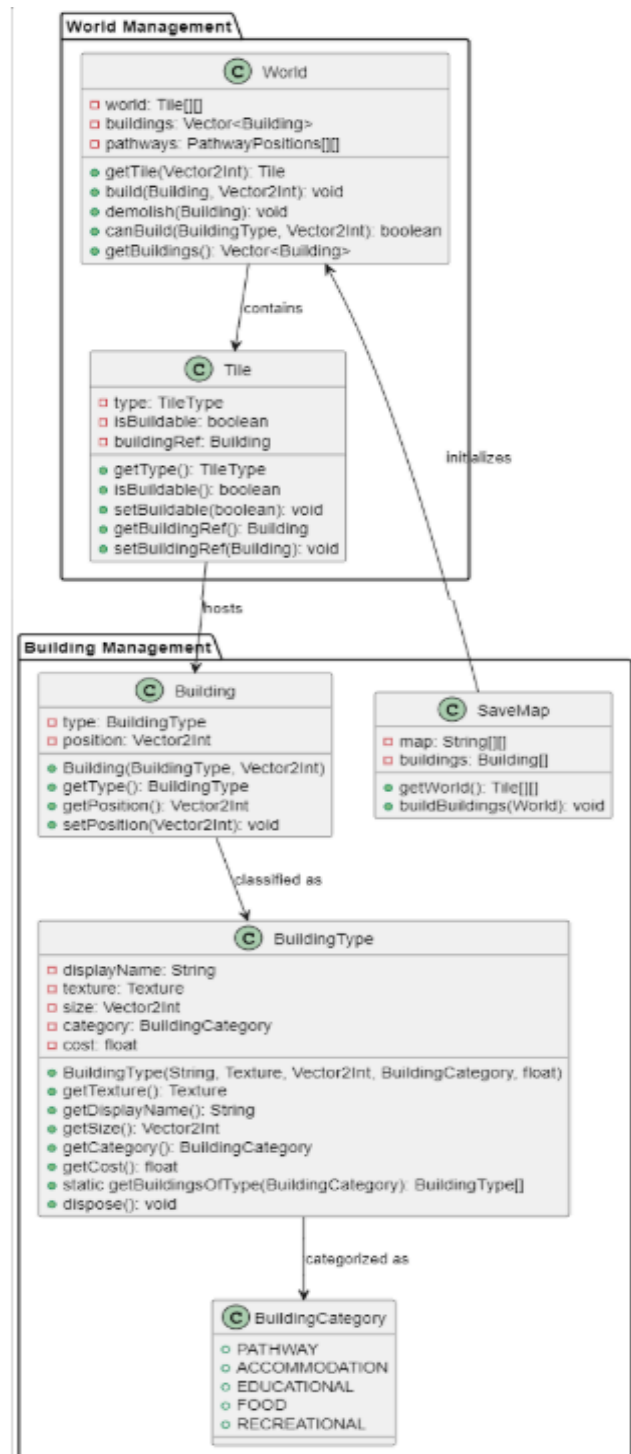
Core Architecture and State Management:

Main

○ fullscreen: boolean
○ prevAppWidth: int
○ prevAppHeight: int

● switchScreen(ScreenType): void
● create(): void
● dispose(): void

manages ↓

GameScreen

● show(): void
● render(delta: float): void
● resize(width: int, height: int): void
● dispose(): void

GameUtils

● getMouseOnGrid(World): Vector2Int
● calculateScaling(): void
● updateSatisfactionScore(World): float
● checkAccomSatisfaction(HashMap<Building, AccomSatisfactionDetail>): void
● updateVisibleSatisfactionScore(): void
● generateGraph(World): GraphNode[]
● getMoneyGenerated(Building): float
● getMoneyGenerated(BuildingType): float

accesses ↘   *utilities* ↙

GameState

○ scaleFactor: float
○ uiScaleFactor: float
○ money: float
○ targetSatisfaction: float
○ totalScore: int

● getBuildingCount(BuildingType): int
● reset(): void
● changeBuildingCount(BuildingType, int): void
● getState(): GameState

uses ↓   *utilities* ↓

Constants

○ GAME_DURATION: long
○ TILE_SIZE: int
○ UI_SCALING_MAP: TreeMap<Integer, Float>
○ sellCostRecovery: float
○ moveCostRecovery: float
○ dontRemoveSelection: BuildingCategory[]

Event and Achievement Management:

GameEventManager

- □ random: Random
- □ currentGameEvent: GameEventType
- □ lastGameEventTime: long
- □ gameEventChance: int
- □ gameEventBag: ArrayList<GameEventType>

- ● getGameEventManager(): GameEventManager
- ● getCurrentGameEvent(): GameEventType
- ● tryForGameEvent(): boolean
- ● rollGameEvent(): GameEventType

AchievementUI

- ● AchievementUI(Viewport, Table, Skin)
- ● render(): void
- ● constructTable(AchievementType, Skin): Table
- ● placeTable(): void

triggers

manages

displays

GameEvent

- □ type: GameEventType

- ● GameEvent(GameEventType)
- ● getType(): GameEventType

AchievementType

- ○ title: String
- ○ description: String
- ○ scoreChange: int
- ○ static FIRSTBUILDING
- ○ static TENBUILDING
- ○ static FIFTYBUILDING
- ○ static FULLMAP
- ○ static FIVEGREGGS
- ○ static THREEHUNDREDROAD
- ○ static ONEOFEACH
- ○ static FIVEOFEACH
- ○ static MILLIONAIRE
- ○ static PERFECTION

- ● AchievementType(String, String, int)
- ● isAchieved(): boolean

uses

GameEventType

- □ displayName: String
- □ scoreEffect: float
- □ eventMessage: String

- ● GameEventType(String, float, String)
- ● getDisplayName(): String
- ● getScoreEffect(): float
- ● getEventMessage(): String

Input and UI Management:

MenuScreen

□ game: Main
□ audio: AudioController
□ viewport: Viewport
□ stage: Stage
□ skin: Skin

● MenuScreen(Main)
● show(): void
● render(delta: float): void
● resize(width: int, height: int): void
● dispose(): void

UIBuildingCounter

□ counterLabels: Label[]

● UIBuildingCounter(Viewport, Table, Skin)
● render(): void
● placeTable(): void

UIBuildMenu

□ skin: Skin
□ tooltip: MoneyGenTooltip

● UIBuildMenu(Viewport, Table, Skin)
● openMenu(BuildingCategory): void
● closeMenu(): void
● render(): void
● placeTable(): void

counts → *opens* → **BuildingCategory**

utilities → **GameUtils**

accesses ↓   *displays* ↓

MenuState

○ leaderboard: boolean
○ achievement: boolean
○ inMenu: boolean

LeaderboardUI

□ nameLabel: Label
□ scoreLabel: Label

● LeaderboardUI(Viewport, Table, Skin)
● render(): void
● placeTable(): void

accesses →

UIOverview

● UIOverview(Viewport, Table, Skin)
● render(): void

LeaderboardUtils

□ json: Json
□ preferences: Preferences

● loadScore(): LeaderboardElement[]
● addScore(LeaderboardElement): void

uses   *manages* →

displays ↓

GameState

LeaderboardElement

□ name: String
□ score: int

● LeaderboardElement(String, int)
● getName(): String
● getScore(): int
● setName(String): void
● setScore(int): void

Utility and Support Classes:

RenderUtils

- drawWorld(SpriteBatch, World, Color): void
- drawBuildings(SpriteBatch, World, Color): void
- drawWorldGridlines(ShapeRenderer, World, Color): void
- drawPlacingBuilding(SpriteBatch, World, BuildingType, Color, Color): void
- outlineBuilding(ShapeRenderer, Building, Color, int): void
- drawParticles(SpriteBatch, ArrayList<ParticleSpawner>): ArrayList<ParticleSpawner>

uses

GameUtils

- getMouseOnGrid(World): Vector2Int
- calculateScaling(): void
- updateSatisfactionScore(World): float
- checkAccomSatisfaction(HashMap<Building, AccomSatisfactionDetail>): void
- updateVisibleSatisfactionScore(): void
- generateGraph(World): GraphNode[]
- getMoneyGenerated(Building): float
- getMoneyGenerated(BuildingType): float

utilities

GameState

Timer

- finishTime: long
- onPauseTime: long
- paused: boolean
- eventCalled: boolean
- event: Runnable

- setEvent(Runnable): void
- setTimer(long): void
- pauseTimer(): void
- resumeTimer(): void
- getTimeLeft(): long
- isPaused(): boolean
- poll(): boolean

InterfaceUtils

- enableActor(Actor): void
- enableChildren(Actor): void
- enableAllActors(Stage): void
- disableAllActors(Stage): void
- getMousePositionInViewport(Viewport): Vector2

LeaderboardUtils

- loadScore(): LeaderboardElement[]
- addScore(LeaderboardElement): void
- scoreSave(LeaderboardElement[]): void

executes

manages

manages

Runnable

Actor

LeaderboardElement

Component Diagram

Longboi Life

Utility & Support
- Vector2Int
- Timer
- EventHandler
- InterfaceUtils
- Constants
- CameraManager

Rendering Engine
- RenderUtils

Game Logic
- GameUtils
- AchievementManager
- GameState
- GameEventManager

Input Handling
- InputManager

Audio System
- SoundEffect
- MusicHandler

User Interface
- UIBuildMenu
- MenuScreen
- Leaderboard UI
- UIManager
- SettingsUI

Persistence
- LeaderboardUtils
- SaveMap

World Management
- Building
- Tile
- BuildingType
- World

Support Functions

UI Helper Functions

Displays UI

Provides Utilities

Updates UI

Processes Input

User Actions

Triggers Sounds

Saves Data

Updates State

Draws World

Behavioural Diagrams

State Diagram