Assessment 1: Greenfield Development

Architecture

Cohort 3 Group 9

Oliver Barden

Connor Burns

Sam Goff

Milap Keshwala

Lewis Mitchell

Jeevan Singh Kang

Harry Thomas

Titas Vaidila

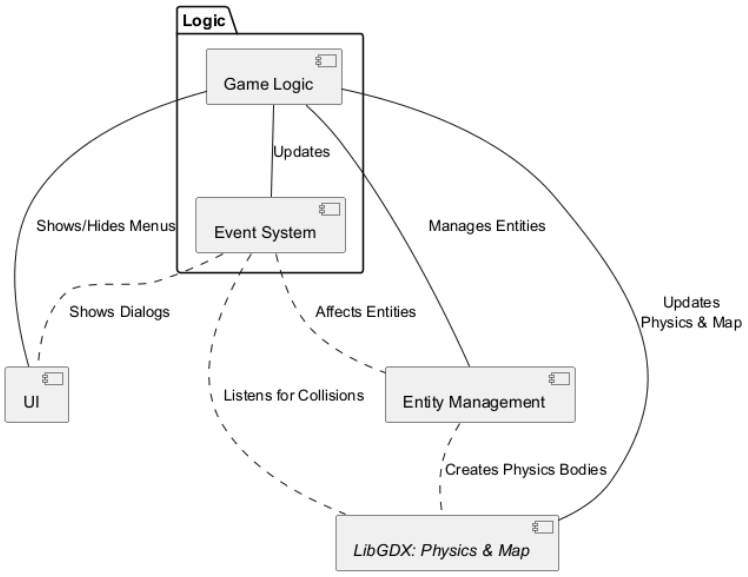
Introduction

Escape from Uni is a top-down maze-puzzle game in which the player aims to 'escape' a day at the University of York. The game provides a family-friendly and broadly accessible experience within a low-pressure environment. In this document, we will detail the software architecture for Escape from Uni, establishing structural and behavioural designs and justifying our decisions in line with our project requirements.

Any diagrams not directly included in this document can be found in the architecture gallery.

Architectural Design

A primary requirement for the project is future upgradability and scalability, as defined by the requirement **NFR_UPGRADE**. In light of this, we selected a **component-based architecture** for Escape from Uni as it provides significant flexibility within the design process by decoupling data and behaviours into reusable *Component* classes that are aggregated by *Entity* classes.

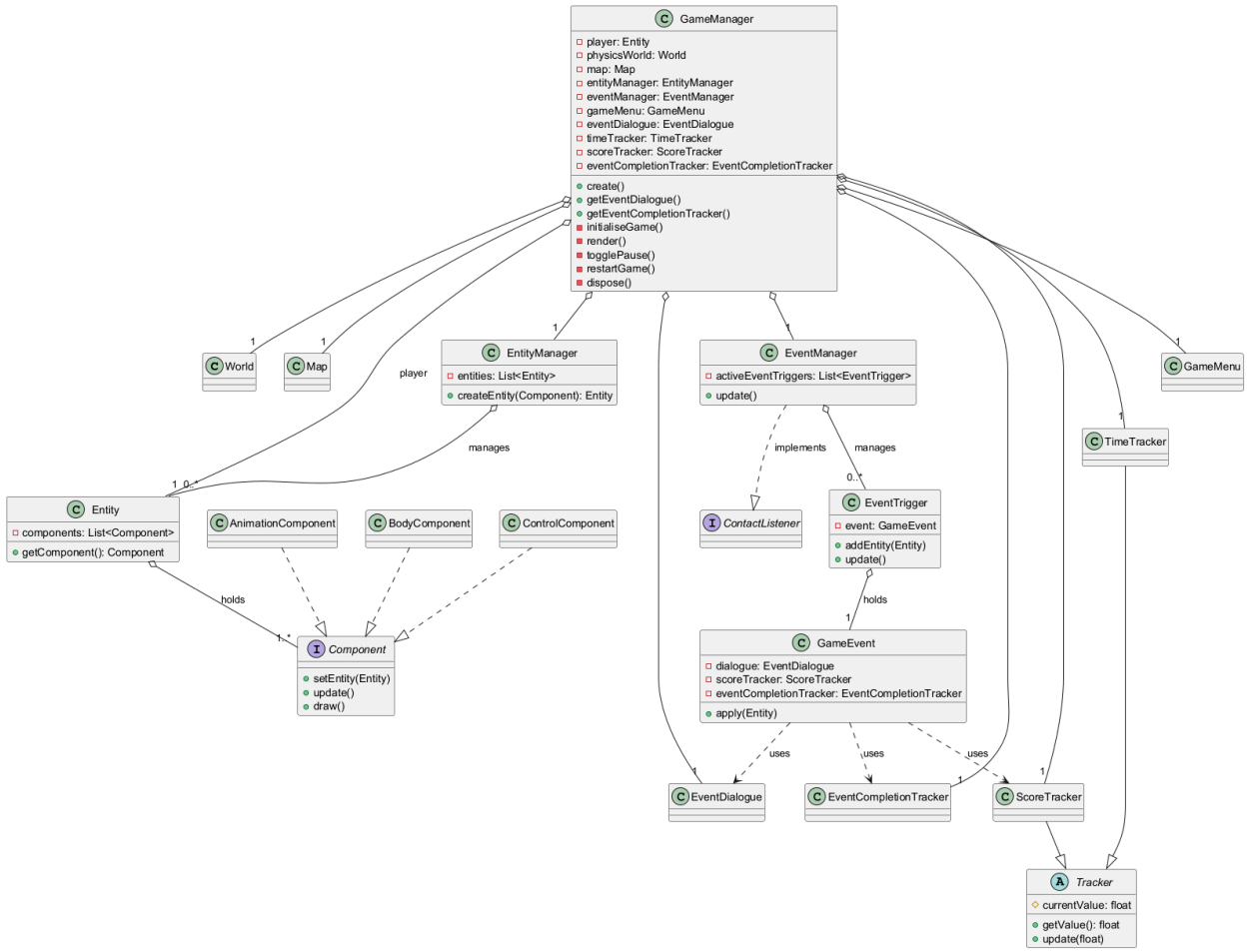


Structure

The class diagram below shows the central relationships that make up the architecture of this project. The *GameManager* class operates at the centre, creating and holding references to various relevant objects (*World, Map,* various managers and trackers).

Tracker classes inherit from a superclass that provides scalability options (**NFR_UPGRADE**) and reduces redundancy across the codebase. Subclasses *TimeTracker* and *ScoreTracker* extend from it, providing common interfaces for methods.

Entity management is controlled through the *EntityManager* which is responsible for assembling a list of entities. These hold components that are assigned on creation. *GameManager* creates a 'player' entity with various relevant components to allow for user control and physical interactions.



Behaviour

The behavioural architecture is modelled using UML sequence and state diagrams to describe the game's runtime interactions.

High-Level Game States

Here we outline the high level flow through the game. This is handled by a state machine *GameState* within the *GameManager* class. We move between multiple game states from when the game is initially launched to the game running and finishing. Animation states between the splash screen and main game are also included.

The initial SPLASH state renders the splash screen, fulfilling requirements for an introductory main screen for the game (**FR_Intro_Screen**). This is a permanent state until a user input is made and the main section of the game is launched. When a user input is received, the transition states FADING_OUT and FADING_IN improve separation between the splash screen and main game (**FR_UI**) and allow the game to be initialised.

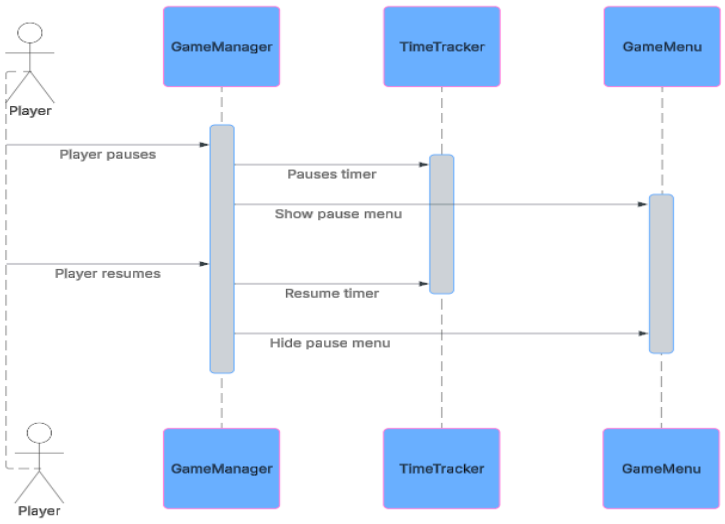
During the main game loop, the user is able to play the game (**UR_Play_Game**). The *EntityManager* and *PhysicsWorld* classes handle operations until the game timer (**FR_Countdown**) reaches zero. We then transition to state GAME_OVER until the program is exited.

A diagram for this behaviour can be seen here.

Event Triggers

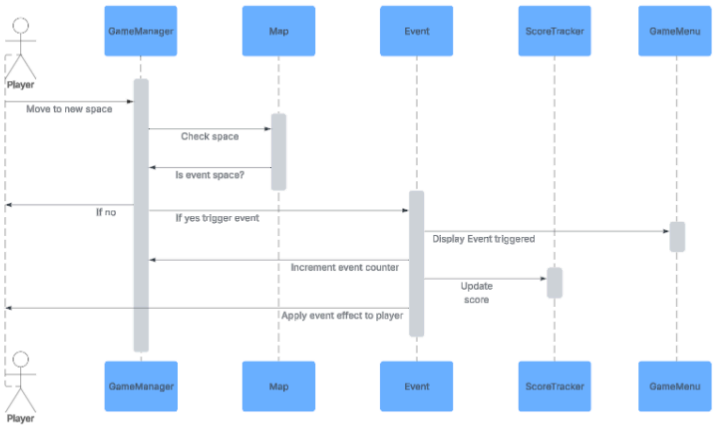
The diagram here describes the runtime behavior when a player encounters an event, implementing the requirements **UR_Events** and **UR_Score**. It begins with the player moving via user input. Then the *GameManager* asks the *Map* to determine if the new space will trigger an event. If it does, the *GameManager* triggers it. *Event* then informs the *GameMenu* to display an event message, increments the event counter, updates the score through *ScoreTracker* as needed and applies the corresponding effects to the player.

Each class carries out a specific role. For example, the *GameManager* takes care of the logic flow, the *Map* controls positional data and *Event* encapsulates its own behaviour. This structure makes maintainability and testing much easier. Also, the diagram demonstrates the *GameMenu* displaying a clear message to the user, which supports usability and immersion.



Game Pausing

Here we represent the runtime behaviour when a player pauses and resumes play, implementing the requirements **FR_Pause** and **FR_Countdown** linking to **UR_Play_Game**. The player pauses the game, which causes the *GameManager* to tell the *TimeTracker* to stop the 5 minute timer. It then has the *GameMenu* display the paused menu in the game. When the player resumes play, everything reverses. The timer is resumed via TimeTracker and the pause menu is removed from display.



The *TimeTracker* encapsulates its own behaviour, which allows easier and more accurate testing. A pause feature aligns with the target casual audience as a lot of users wouldn't always want to or be able to finish a run of the game all at once. Finally, this same model can be used to help implement other parts of the game such as the game over and victory screens, which supports scalability.

Languages and Tools

Our behavioural architecture was modelled using UML diagrams created in Lucidchart, which was chosen for easy collaboration and its auto-alignment features. This allowed multiple members of the team to construct initial designs, while later models could be refined by a single assigned team member. Structural architecture was modelled in PlantUML, which offers more sophisticated tools for managing variables and properties across larger diagrams. Additionally, as the software is text-based we had flexibility over version control, which Lucidchart only supports through its own proprietary software.

Architecture Evolution

Initial Design

The first architectural outline for this project was completed through CRC cards. This provided an opportunity to assess all components and separate them based on responsibility type to ensure proper modularity would be maintained across the future codebase. By modelling in this way, relationships between components were found and formalised.

Player

Responsibilities	**Collaborators**
Provides information of player: timetable, Inventory, Move inputs, position etc	EnergyComponent ScoreTracker Timetable Item PositionComponent

We established that our components could be separated into three main layers (diagram) on top of those provided inherently by our game engine LibGDX:

1. UI (drawing assets, handling animations and showing events)
2. Logic (managing events and player controls)
3. Data (handling map storage and player/entity positions)
4. *Map/Physics (handled through LibGDX)*

This initial design effectively maintained modularity, but was limited in scope. Not all opportunities for full abstraction were explored in this first pass. An initial class diagram can be seen on the website architecture gallery.

Refinement

A crucial consideration at this stage was the projects' future scalability. The initial design contained procedural elements in the construction of the *GameManager class* that were identified as a bottleneck. Refactorings were made to eliminate these issues:

1. Removal of event detection from *GameManager.* This functionality was shifted to the *events* package.
2. *EventManager* converted to be notified of collision events by the *World.* This was much more efficient than repeatedly querying the map to detect collisions.

These changes moved the project architecture to be fully event-driven and decentralised, which allows for simple scaling and upgrades moving forward.

To further aid modularity (**NFR_UPGRADE**), superclasses were introduced for the *Player* and various event classes. This allows future developers to construct additional features with similar functionality (e.g. hostile entities, additional events) without major refactoring required.

Requirements Traceability

Requirement ID	**Description**	**Implementions**
UR_Play_Game	Users can play through the game.	GameManager, EntityManager
UR_Events	Users will encounter events.	EventManager, EventTrigger, GameEvent
UR_Play_Tutorial	Users can play a tutorial.	GameManager (in SPLASH state), FR_Intro_Screen
UR_Player_Control	Users can move their character.	ControlComponent
UR_Score	The system will display the score.	ScoreTracker, GameManager (displays scoreLabel)
UR_UI	The UI should be clear and easy to navigate.	GameMenu, EventDialogue, Skin classes
FR_Countdown	The game will have a 5-minute timer.	TimeTracker (instantiated with 300f)
FR_Pause	Users can pause the game.	GameManager, GameMenu
FR_[P/N/H]_Events	Positive, Negative, and Hidden events.	GameEvent (constructor takes points and eventType)
FR_Event_Count	Keep track of encountered events.	EventCompletionTracker
FR_Score_Update	The system will update the score.	ScoreTracker, GameEvent
FR_Movement	Allow movement via WASD.	ControlComponent (logic read by GameManager)
NFR_UPGRADE	Designed for future scalability.	Component-Based Design (Entity, Component) and Events handling