Architecture

Group 4 - THADJAM

Abdul Fofanah
Matthew Holleran
Toby Watchorn
Arwen Minton
Daneena Roydean
Jessica McKerill
Henry Bambrough

Introduction and Tools:

The system architecture for *Escape from University* was modelled using the Unified Modelling Language (UML). This language allowed us to provide a formal description of the structural and behavioural aspects of the game. UML is a standardised notation for representing classes, components and their interactions in object-oriented and entity-component-system designs. The architecture shows how the game's core modules work together to deliver gameplay and their alignment with the user and system requirements.

Structural diagrams were created using PlantUML. This is a text-based tool that uses source descriptions to automatically produce class and component diagrams. This allowed the team to iteratively improve architecture diagrams alongside implementation and ensure version control by using GitHub. During early discussions, we also used Lucidchart to make small component diagrams.

Behavioural aspects of the system were represented using UML sequence and state machine diagrams. These tools were used to model message flows between key objects during gameplay events. This includes player movement, collision handling and event triggering. These diagrams help to support traceability, and they show runtime behaviour by connecting interactions with requirements.

All diagrams and interim versions were produced following an Agile/Scrum workflow. This ensures that architectural models evolve alongside each sprint. Our approach was an iterative, tool-supported approach, which allowed the team to visualise dependencies and communicate design choices while ensuring that the implemented architecture matched the functional and non-functional requirements.

Structural Architecture Diagrams:

The architecture follows a modular object-oriented design built on the LibGDX framework. There is a clear separation between presentation, game logic and entity management. This structure emphasises composition over inheritance and also utilises LibGDX's screen-based architecture for managing game states.

Layered Architecture:

This structure separates any concerns between presentation, game logic and framework responsibilities, allowing changes to user interface elements without disrupting core gameplay mechanics.

Layer	Components	Responsibilities
Presentation layer	MenuScreen, GameScreen, WinScreen, GameOverScreen, TutorialScreen, LeaderboardScreen	Handles all user-facing elements such as screen transitions and menu navigation. Converts inputs into game actions
Game Logic layer	Player, Dean, NPC, Locker,	Contains the core game rules and entity behaviours.

	BusTicket, GameTimer, Achievement, Decrease_Time, Drown, Extra_Time, Freeze_Dean, Patrol_Dean, Save_Leaderboard, Slow_Down, Teleport	Manages player movement, enemy movement, item interactions and game timing
Framework layer	LibGDX Engine, SpriteBatch, OrthographicCamera, TiledMap	Provides the foundational services, such as rendering and screen lifecycle control, through LibGDX's framework

Class Relationships:

The core game entities follow a composition-based design where GameScreen acts as the central coordinator. Each entity encapsulates its own state and behaviour, whilst GameScreen manages their interactions and rendering order. The Player class provides character movement and direction, the Dean class implements pursuit AI, while the interactive objects like Locker and BusTicket manage their own state transitions.

OOP UML Component Diagram

Our final component diagram represents the implemented architecture where GameScreen serves as the central coordinator, directly managing all game entities and systems. The architecture employs composition-based relationships rather than deep inheritance hierarchies with each entity encapsulating its own state and behaviour, while GameScreen handles their interactions and rendering order. This design supports our core requirements: FR_MOVEMENT through direct input handling, FR_ANTAGONIST via the Dean's pursuit AI and the three event types (FR_POSITIVE_EVENT, FR_NEGATIVE_EVENT, FR_HIDDEN_EVENT) through specialised entity interactions. The final OOP component diagram for the game's architecture can be found on the website Architecture Appendix. | Escape from University!

Behavioural Architecture Diagrams:

The behaviour architecture models the dynamic interactions between core components at runtime. These sequence diagrams illustrate message flows in response to player actions and automated system events using three primary gameplay interactions.

Use Case 1 - Player movement:

This sequence models how the system processes player movement and validates moves within the maze environment. UML sequence diagram available on the website Architecture Appendix. | Escape from University!

Actors and Components:

- Player: entity receiving movement commands
- GameScreen: central coordinator processing input and game state
- TiledMap: provides collision data through tile properties

Process Summary:

- When the player inputs movement, GameScreen.handleInput() processes the command, checks collision via isCellBlocked() and updates the player position if their move was valid. The camera follows the player and the rendering system updates the display.

Use Case 2 - Dean's Pursuit:
This sequence describes the AI logic controlling the Dean's automatic movement through the maze. The UML sequence diagram is available on the website Architecture Appendix. | Escape from University!

Actors and Components:
- GameScreen: triggers entity updates each frame
- Dean: AI-controlled entity pursuing the player
- Player: target entity providing position data

Process Summary:
- For each frame, GameScreen updates the Dean, which calculates direction towards the player, checks for valid movement and updates the Dean's position accordingly. Collision detection occurs in GameScreen, triggering reset logic when the player is caught by the Dean.

Use Case 3 Player Searching Objects:
This sequence outlines system behaviour when the player interacts with game objects. UML sequence diagram available on the website Architecture Appendix. | Escape from University!

Actors and Components:
- Player: initiates interactions via the E key being pressed
- GameScreen: manages interaction flow and state changes
- Interactive Objects = BusTicket, Locker, NPC

Process Summary:
When the player presses E near interactive objects, GameScreen checks proximity and triggers appropriate responses: collecting items, activating boosts or displaying dialogue.

Justification and Design Rationale:

Why Object-Oriented Architecture with LibGDX Screens?
We selected a pure object-oriented architecture built on LibGDX's screen pattern because it provided the optimal balance between structure and framework integration. The screen-based architecture naturally fits the game's state transitions (menu -> game -> win/lose screens), while OOP design patterns made the codebase accessible to our team and maintainable throughout development.

Why UML and PlantUML?
UML diagrams helped standardise team communication and ensured shared understanding of system architecture. Using PlantUML allowed us to maintain diagrams in version-controlled text format, which allowed for easy updates and traceability throughout development. This workflow provided both clarity for design discussions as well as accurate documentation of the implemented system.

Support for Maintainability, Scalability and Testability
The screen-based OOP architecture improves maintainability by isolating different game states into separate classes. Each screen manages its own lifecycle and resources, preventing interference between game states. Scalability is supported through LibGDX's robust framework, which handles asset management and rendering efficiently. Testability is enhanced by well-defined class responsibilities with clear interfaces which allow for targeted unit testing of individual game entities.

Trade-offs
The main trade-off in our architecture is the central role of GameScreen as a coordinator as this creates some coupling between systems. However, this is offset by the clarity and simplicity of having a clear central point for game logic. By keeping entities well-encapsulated and using composition, we maintained flexibility while leveraging LibGDX's proven architectural patterns.

Evolution/Iteration Evidence:
Our architecture evolved significantly from the initial concepts to the final implementation. All interim architectural versions, including early ECS diagrams, OOP transition plans and CRC cards, are available on our project website at Architecture Appendix. | Escape from University!

Initial ECS Exploration -> Final OOP Implementation:
Initially, the team explored the idea of using an Entity-Component System (ECS) architecture (available on the website Architecture Appendix. | Escape from University!) and created component diagrams modelling entities as compositions of data components. However, after evaluating team expertise and project scope, we transitioned to the more familiar, traditional OOP approach using LibGDX's established patterns. This decision prioritised development efficiency and framework alignment over architectural innovation.

Screen Architecture Refinement:
Early designs used a monolithic game controller, but we evolved to LibGDX's screen-based architecture after recognising the fact that it better fit our game state management. This change improved the separation of concerns and simplified state transitions.

Entity Management Evolution:
Initially, we planned separate manager classes for collision, input and physics, but later decided to consolidate these responsibilities into GameScreen after finding out that LibGDX's integrated approach reduced complexity without sacrificing any clarity.
Final OOP Architecture:
Upon the extension of requirements, another package diagram was added and extended to include all the new classes. The final OOP component diagram for the game's architecture can be found on the website. Architecture Appendix. | Escape from University!

CRC Cards:

Class	**Responsibilities**	**Collaborators**

GameScreen	Process player input, update game entities, manage collisions, handle screen transitions, coordinate rendering	Player, Dean, BusTicket, Locker, GameTimer
Player	Handle movement direction, maintain position state, and manage sprite rendering	GameScreen, SpriteBatch
Dean	Pursue the player using simple AI, update position each frame, and reset when catching the player	GameScreen, Player
NPC	Display dialogue when the player interacts with it, manage dialogue visibility, render NPC sprite and message	GameScreen, Player, SpriteBatch
Locker	Provide speed boost when interacted with, manager boost duration timer, display interaction messengers, render locker sprite	GameScreen, Player, SpriteBatch
BusTicket	Manage collection state, render ticket in world when discoverable, render UI icon when collected, handle ticket discovery logic	GameScreen, SpriteBatch, OrthographicCamera
GameTimer	Count down game time from 300 seconds (5 minutes), format time display as mm:ss, update UI label display	GameScreen, uiStage, Skin
MenuScreen	Display main menu interface, handle menu navigation inputs, transition to tutorial/game scenes, and manage win screen graphics	MyGame, GameScreen, SpriteBatch
WinScreen	Display victory message and final score, show score breakdown with penalties, handle return to menu navigation, and render win screen graphics	MyGame, GameScreen, SpriteBatch
GameOverScreen	Display game over message, handle menu return navigation, render loss screen interface, and manage screen resources	MyGame, GameScreen, SpriteBatch
TutorialScreen	Display tutorial instructions image, handle progression to main game, manage tutorial resources, render tutorial information	MyGame, GameScreen, SpriteBatch
MyGame	Manage screen lifecycle and transitions, create initial application context, coordinate between different screens, and implement LibGDX game interface	All Screen classes, LibGDX framework
Achievement	Represents the achievements that the player can achieve. Each achievement provides a positive or negative impact.	MyGame
Decrease_Time	An interactable game object that acts as the negative game event by allowing the player to collide with a tree, giving a permanent time reduction.	MyGame, GameScreen, SpriteBatch
Drown	Check if the player has entered the water hazard and reset their position if so. Represents a water hazard that resets the player if they step into it. Uses the same rectangle overlap style as bus and ticket interactions.	MyGame, GameScreen, SpriteBatch

Extra_Time	Update attributes of NPC, and increment the timer on time increase and label timer, showing the label if the label timer is still active. An interactable game object that represents a positive event. When the player collides with this NPC, additional time is permanently added to the game timer.	MyGame, GameScreen, SpriteBatch
Freeze_Dean	A positive hidden event that allows the player to temporarily freeze all dean enemies for a fixed duration when interacting with the "Materials" area in the Tiled map.	MyGame, GameScreen, SpriteBatch
LeaderboardScreen	Implements a screen that displays the top scores stored in the local leaderboard file. The player can return to the main menu or quit the game from this screen.	MyGame, GameScreen, SpriteBatch
NameScreen	Implements a name entry screen using a simple on-screen keyboard. The player enters a first name (mandatory) and an optional last name, then starts the game.	MyGame, GameScreen, SpriteBatch
Patrol_Dean	Represents a Dean enemy that patrols vertically between two Y-axis bounds. The dean reverses direction when reaching its limits or when colliding with a blocked tile.	MyGame, GameScreen, SpriteBatch
Questionnaire	Render the quiz prompt, questionnaire questions, and the result text.	MyGame, GameScreen, SpriteBatch
Save_Leaderboard	Saves leaderboard scores to the local JSON file. Existing file contents are overwritten.	MyGame, GameScreen, SpriteBatch
Slow_Down	Update attributes of bush, and decrement the timer on speed decrease and label timer, showing the label if the label timer is still active.	MyGame, GameScreen, SpriteBatch
Teleport	Convenience method to be called by the game screen's render() method, to draw the teleportation device, and its label using a SpriteBatch at its coordinates.	MyGame, GameScreen, SpriteBatch

Traceability to Requirements:

Requirement ID	Requirement Description	Related Component(s)	Architectural Rationale
FR_MOVEMENT	The system shall allow keyboard inputs to control player movement	GameScreen.handleInput(), Player.setDirection(), Player.getPosition(), GameScreen.isCellBlocked( )	Input processed in GameScreen.handleInput(), which detects WASD keys, sets player direction via Player.setDirection(), validates movement through GameScreen.isCellBlocked() before updating player position
FR_ANTAGONIST	The system shall include an antagonist who follows the player	Dean.update(), Dean.getPosition(), Dean.resetPosition(),Game Screen.render(),	Dean.update() calculates pursuit direction each frame using player position, moves with speed = 0.7f, resets

		Player.getPosition(), GameScreen.isCellBlocked( )	player to start when caught via Dean.resetToStart()
FR_POSITIVE_EVENT _LOCKER	The system shall present a positive event in which the player finds a locker, opens it and gains a speed boost advantage	Locker.update(), Locker.isBoostActive(), GameScreen.handleInput() movement speed logic	Locker.update() detects player proximity and E key presses, activates temporary speed boost, GameScreen doubles movement speed when boost is active
FR_POSITIVE_EVENT _FREEZE_DEAN	The system shall allow the player to activate lab materials that freeze all deans (chase dean and patrol deans) for 30 seconds by pressing E key.	GameScreen.freezeAllDean s(), playerRect.overlaps()	If the playerRect overlaps with the materials and E is pressed, the game screen freezes all the deans.
FR_POSITIVE_EVENT _EXTRA_TIME	The system shall provide an NPC (non-player character) that adds 30 seconds to the game timer when the player collides with it. Auto-collects on contact and disappears after pickup.	Player.getPosition() timer.addTime()	If Player.getPosition() finds that it is overlapping with the NPC, then time is added
FR_NEGATIVE_EVENT _PATROL_DEAN	The system shall include 3 deans that patrol vertically in the bottom area at 0.7f speed. Collision with any patrol dean resets the player to the starting position with a -5 points penalty. Fourth patrol dean spawns if the questionnaire is failed.	gameScreen.isCellBlocked( )	If the deans that are patrolling get stuck in a cell found by isCellBlocked(), they can be reverted to get unstuck
FR_NEGATIVE_EVENT _SLOW_DOWN	The system shall include a bush obstacle that reduces player movement speed by 50% for 20 seconds when collided with.	player.GetPosition()	If .GetPosition() finds that it is colliding with the bush, debuffs can be applied
FR_NEGATIVE_EVENT _DROWN	The system shall include water tiles that trigger player respawn at the starting position with a -10 points penalty when collided with.	Player.getPosition() obj.getRectangle()	If .GetPosition() of both the player and rectangle are found to overlap, the player can be respawned.
FR_NEGATIVE_EVENT _DEAN_CHASE	The system shall present a negative event of the Dean chasing the player around the maze. When caught, the	Player.getPosition() gameScreen.isCellBlocked ForDean() tryMoveDiagonally()	GetPosition() can help guide the dean towards the player via diagonal movements.

	player is reset to the starting position with a -5 points penalty.		
FR_NEGATIVE_EVENT_DECREASE_TIME	The system shall include a tree obstacle that reduces remaining game time by 30 seconds when collided with.	Player.getPosition() timer.addTime()	If.GetPosition() of the player is colliding with tree then time can be added via .addTime()
FR_HIDDEN_EVENT_BUS_PASS	The system shall contain a hidden event of the player's bus pass being hidden in a bush, which, until triggered, cannot be seen on the map	BusTicket.isCollected(), NPC.showMessage, BusTicket.render() conditional rendering	Bus ticket initially hidden, revealed via BusTicket.discover() when the player gets close, the NPC provides a hint through a dialogue message
FR_HIDDEN_EVENT_TELEPORT	The system shall include a hidden science lab entrance that teleports the player to a shortcut location when triggered.	player.getPosition() getRandomSafePosition()	If the player triggers teleport, then getRandomSafePosition() means they can be teleported to a new set of coordinates
FR_HIDDEN_EVENT_QUESTIONNAIRE	The system shall include a hidden questionnaire that awards bonus points for correct answers. Failing the questionnaire spawns a fourth patrol dean.	player.getPosition() Keyboard Input gameScreen.spawnSecondDean()	If the player gets a question wrong via keyboard input, the deans get spawned via .spawnSecondDean()
FR_OFFLINE	The system and all its features shall not require any connection to a network	All asset loading via local files	All textures and maps are loaded from the local filesystem without network dependencies
FR_END_SCORE	The system shall display the user's end score when they complete or fail the game	WinScreen, GameOverScreen, GameScreen.calculateFinalScore(), GameTimer.getTimeLeft()	Final score calculated from remaining time minus penalties, displayed when the player wins the game
FR_USER_TIMER	The system shall display a timer to the user, which displays how long they have been playing for	GameTimer, timerLabel, uiStage	Countdown timer displays remaining time in mm:ss format through the Scene2D UI label
FR_ACHIEVEMENT_SYSTEM	The system shall track player achievements based on gameplay performance. Each achievement has a name, description, and bonusScore (positive or negative integer).	Achievements	Achievements get added through gameplay

FR_LEADERBOARD_SAVE	The system shall save the top 5 player scores persistently in a local JSON file (leaderboard.json)	loadScores() finalScores.add() finalScores.sort() saveScores() Integer.compare() finalScores.removeIndex ScoreFile.readString()	Scores can be loaded and added after the game is finished.
FR_LEADERBOARD_DISPLAY	The system shall display a leaderboard screen showing the top 5 scores with the format '[rank]-[fullName] - [score]'.	Render leaderboard leaderboard.getHighScores(); 	Loads scores from the file and ranks them.
FR_NAME_ENTRY	The system shall provide a name entry screen.	ScreenRender() CreateKeyboard()	CreateKeyboard() lets a player type their name in via the onscreen keyboard.