Architecture

Cohort 1, Group 6 - M6

Members:

Mir Baydemir
Adam Fraulo
Azib Hj Abu Akmar
Agata Pittarello
Esther Scanlon
Jazz Stubbins
Sonia Szetela

This document serves to describe the product's different architectural structures, inner components, the relationships between them and the justification for choosing them. The document will focus on the completed architectural diagrams for the current version of the game, with evidence of the iterative history of the project lifecycle found on the supporting Design Iterations webpages. The diagrams in this document have first been stormed or sketched using the tools Miro, draw.io and starUML before being properly implemented via UML syntax using plantUML.

System Structure

Throughout the architectural design process the team followed the 5 steps of software development outlined by Sommerville (2016), iteratively identifying and updating components based on characteristics to meet requirements.

After reviewing the different architectural styles described by Sommerville (2016) it was clear that a monolithic closed-layered structure with an embedded ECS system was best suited for the project due to it being most applicable to small-scale applications where scalability and distribution isn't a priority, with maintainability being convenient due to layer isolation and game logic centralised in the business layer. However after development we realised we had simply followed the traditional OOP inheritance hierarchy with a helper system and a God Object antipattern caused by our Main.java. (Richards and Ford, 2020; University of York, 2025). The following elements outline the characteristics, design decisions and principles that informed this architecture structure.

Architectural Characteristics (non-functional requirements)

Following steps 1-3 of the Software Development Life Cycle, the team conceptualised initial components based on the most applicable non-functional requirements. These characteristics derived from use cases and team storming sessions were then grouped with corresponding user and constraint requirements into a table that would serve as a reference throughout the project lifecycle (see Characteristics Table page for original and https://keybordkat.github.io/websiteGroup6/Req2.pdf for part 2 requirements). After changes to the user requirements, the table was no longer completely accurate, but still provided a good reference for what we aimed to achieve.

Design Decisions

The team agreed on architectural design decisions based on user and system requirements whilst ensuring these designs were compatible with the chosen Java based game engine: libGDX (satisfying CR_ENGINE).

The team chose this OOP game engine due to the on-going library support, tile integration and compatibility with the chosen architecture style.

A traditional AABB and 'Trigger' based collision system were implemented for movement and event handling for simplicity and performance (UR_EVENTS, UR_GAMEPLAY). This helped us keep the game extensible, so more events could easily be added in the future by utilising this system.

We embedded event status tracking within entities to maintain inter package traceability for score and event logic. (UR_SCORE, UR_EVENTS).

Design Principles

The design favours high cohesion since each helper system handles one responsibility, though coupling is tight through static references to the Main.java controller.
Separation of concerns is partially achieved through system specialization: RenderingSystem handles presentation, InputSystem manages input, and CollisionSystem handles physics.
However, Main.java centralizes game logic, event handling, and system coordination as a controller class.
Relative modularity from the OOP inheritance hierarchy allows new entity types to extend base classes. Map-based events enable level designers to add content without code changes, though new entities require Main.java modifications for instantiation.
Information hiding is inconsistently applied: systems encapsulate internal logic, but entities directly access Main's static system references, creating coupling.

Together these principles support functional gameplay through clear system responsibilities and inheritance-based code reuse, while acknowledging architectural trade-offs in coupling and separation of concerns.

Figure 1: Monolithic closed-layered architecture diagram.

This diagram illustrates the overall architectural structure of the Escape from Uni game, composed of three groups of classes: entity hierarchy (), helper systems (), and main controller. The different systems are specialised, yet game logic is centralised by Main.java, giving the game strong cohesion. This style was selected for simplicity and maintainability (NFR_RESILIENCE), which is optimised for small-scale projects, fitting the team size and timeframe.

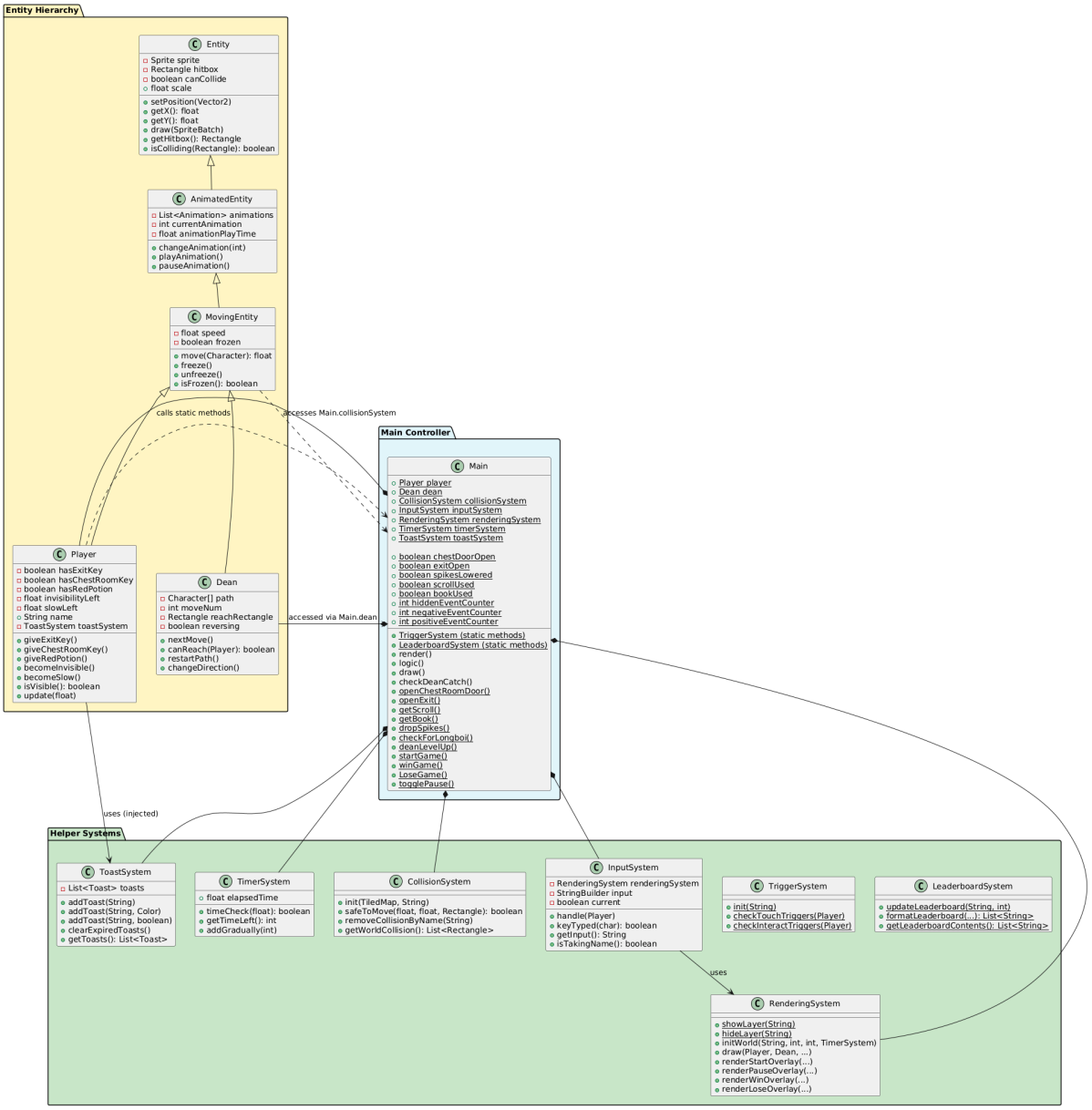
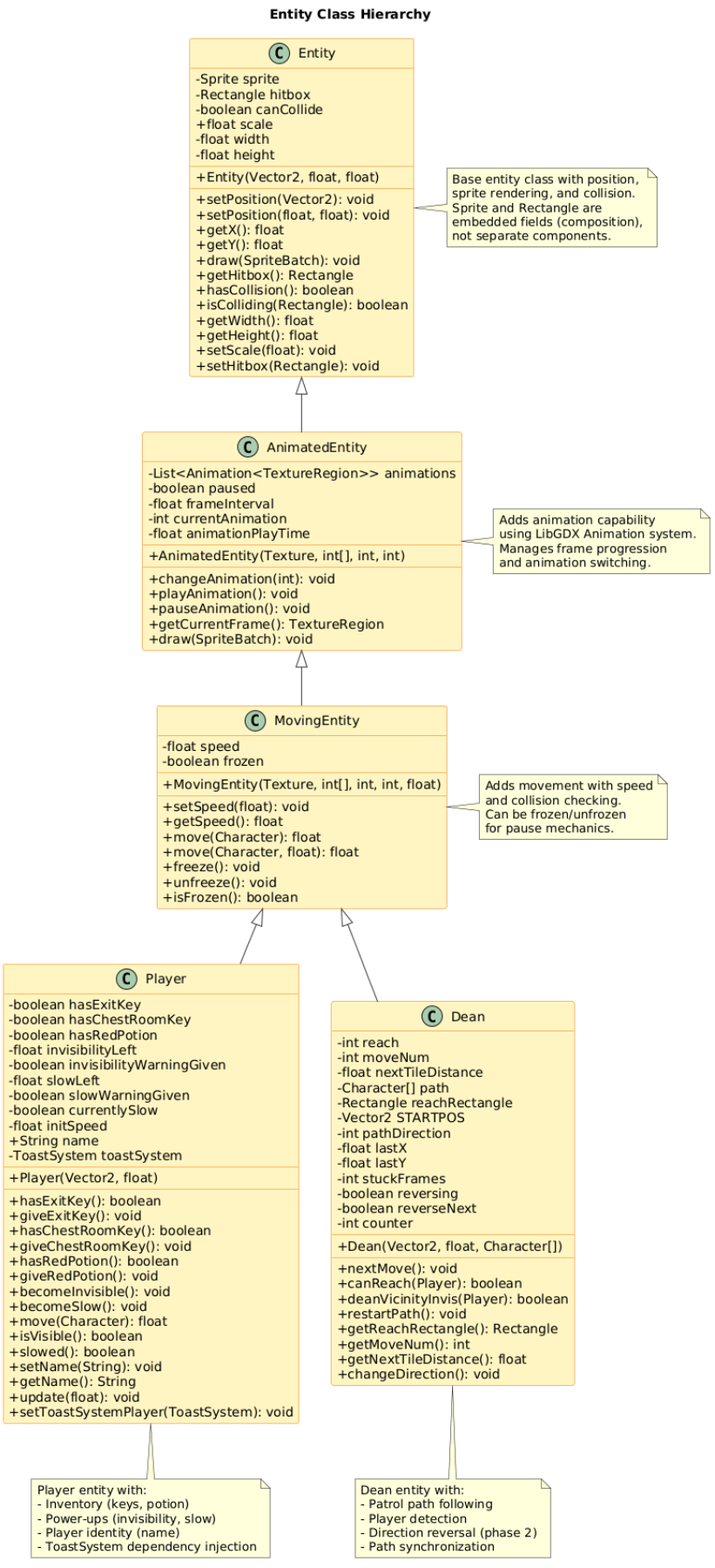
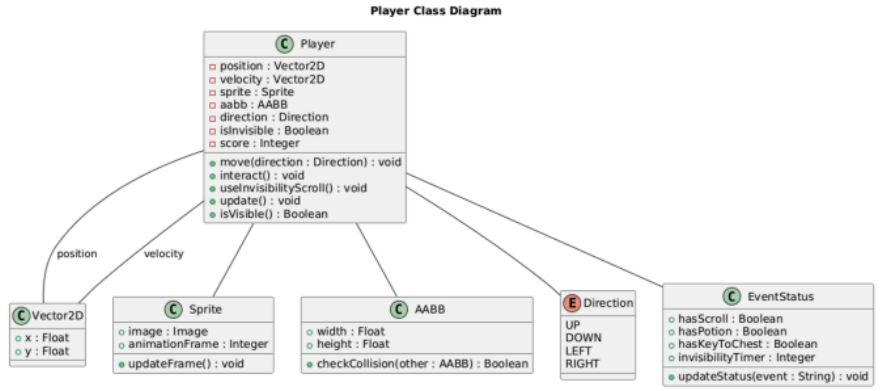


Figure 2: Entity-Component-System architecture diagram


Entity Class Hierarchy
Entity
-Sprite sprite
-Rectangle hitbox
-boolean canCollide
+float scale
-float width
-float height
+Entity(Vector2, float, float)
+setPosition(Vector2): void
+setPosition(float, float): void
+getX(): float
+getY(): float
+draw(SpriteBatch): void
+getHitbox(): Rectangle
+hasCollision(): boolean
+isColliding(Rectangle): boolean
+getWidth(): float
+getHeight(): float
+setScale(float): void
+setHitbox(Rectangle): void
Base entity class with position, sprite rendering, and collision. Sprite and Rectangle are embedded fields (composition), not separate components.
AnimatedEntity
-List<Animation<TextureRegion>> animations
-boolean paused
-float frameInterval
-int currentAnimation
-float animationPlayTime
+AnimatedEntity(Texture, int[], int, int)
+changeAnimation(int): void
+playAnimation(): void
+pauseAnimation(): void
+getCurrentFrame(): TextureRegion
+draw(SpriteBatch): void
Adds animation capability using LibGDX Animation system. Manages frame progression and animation switching.
MovingEntity
-float speed
-boolean frozen
+MovingEntity(Texture, int[], int, int, float)
+setSpeed(float): void
+getSpeed(): float
+move(Character): float
+move(Character, float): float
+freeze(): void
+unfreeze(): void
+isFrozen(): boolean
Adds movement with speed and collision checking. Can be frozen/unfrozen for pause mechanics.
Player
-boolean hasExitKey
-boolean hasChestRoomKey
-boolean hasRedPotion
-float invisibilityLeft
-boolean invisibilityWarningGiven
-float slowLeft
-boolean slowWarningGiven
-boolean currentlySlow
-float initSpeed
+String name
-ToastSystem toastSystem
+Player(Vector2, float)
+hasExitKey(): boolean
+giveExitKey(): void
+hasChestRoomKey(): boolean
+giveChestRoomKey(): void
+hasRedPotion(): boolean
+giveRedPotion(): void
+becomeInvisible(): void
+becomeSlow(): void
+move(Character): float
+isVisible(): boolean
+slowed(): boolean
+setName(String): void
+getName(): String
+update(float): void
+setToastSystemPlayer(ToastSystem): void
Player entity with:
- Inventory (keys, potion)
- Power-ups (invisibility, slow)
- Player identity (name)
- ToastSystem dependency injection
Dean
-int reach
-int moveNum
-float nextTileDistance
-Character[] path
-Rectangle reachRectangle
-Vector2 STARTPOS
-int pathDirection
-float lastX
-float lastY
-int stuckFrames
-boolean reversing
-boolean reverseNext
-int counter
+Dean(Vector2, float, Character[])
+nextMove(): void
+canReach(Player): boolean
+deanVicinityInvis(Player): boolean
+restartPath(): void
+getReachRectangle(): Rectangle
+getMoveNum(): int
+getNextTileDistance(): float
+changeDirection(): void
Dean entity with:
- Patrol path following
- Player detection
- Direction reversal (phase 2)
- Path synchronization

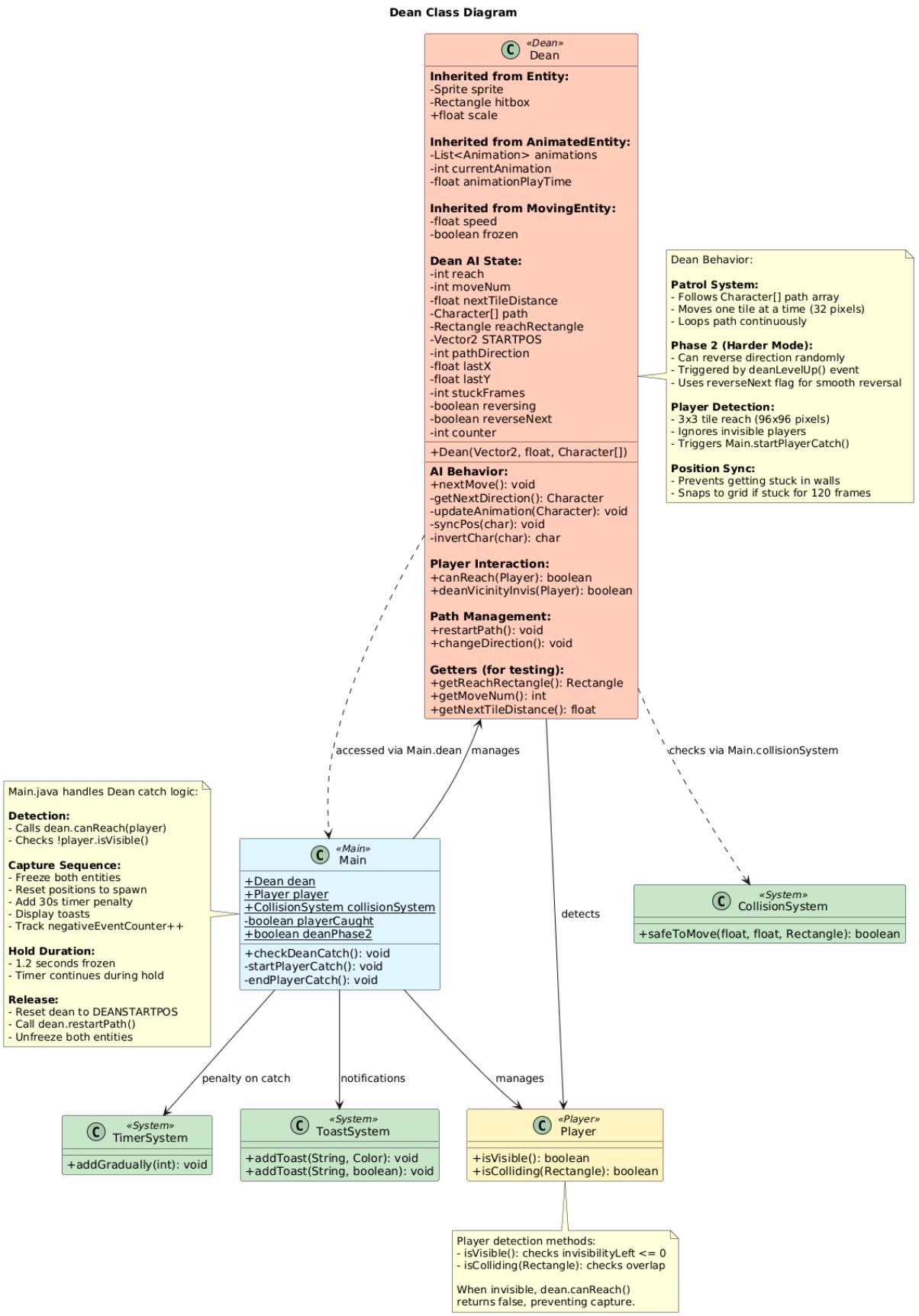
This diagram illustrates the OOP structures inter-package relationships via components to form modular and reusable game logic. The diagram shows the dependencies and association/ container relationships encompassing: Player and Dean entities that inherit from the Entity superclass and share components such as Sprite, AABB and Vector2D etc.Static booleans track triggered items, visibility and score in Main and pass that data to systems like Trigger, Collision and Event to manage activation consistently across entities. Systems like Collision and Rendering execute on shared components rather than directly on entities themselves, reducing unnecessary code duplication and improving modularity and reusability satisfying NFR_RESILIENCE and NFR_RELIABILITY. Longboi, Scroll and potion aren't true entities, but are managed through the rendering system, and therefore do not appear in this diagram.

Figure 3: Player class diagram



These diagrams focus on specific behaviour and dependencies of classes. The player class integrates Sprite, AABB, Vector2D and EventStatus components to manage position, rendering and invisibility (detectable) state. Collision and interaction triggers (TriggerA and TriggerB) enable movement, input handling and event activation while tracking score and visibility. This design satisfies FR_MOVEMENT and FR_MAP by ensuring consistent player rendering, input position response and collision feedback.

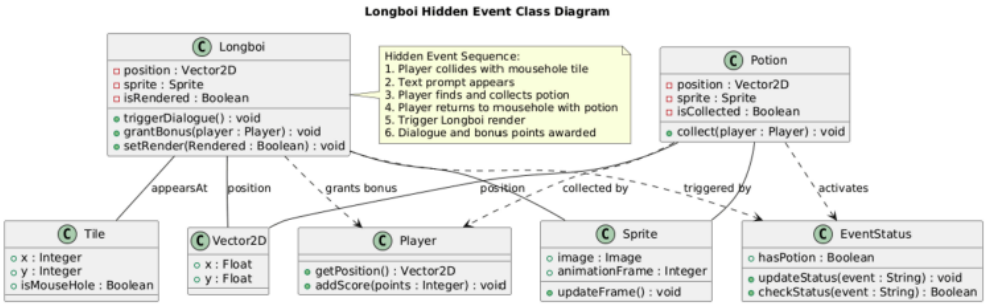
Figure 4: Dean class diagram (negative event trigger)



This diagram illustrates how the Dean class functions as a negative event while sharing components with other classes such as Vector2D, Sprite, AABB, and EventState. The Dean extends base Entity functionality with patrol, detection, capture and reset behaviour,

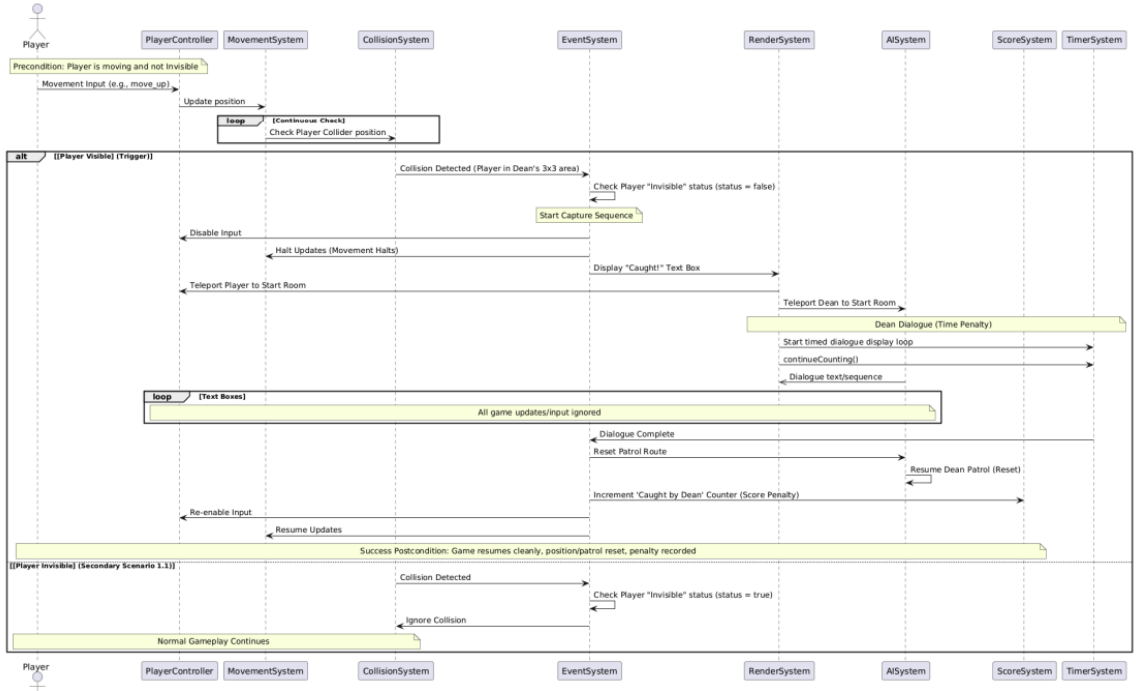
interacting with the Player and EventStatus classes to trigger its event. These methods form its patrol and capture logic: player invisibility checks, reset logic and score penalties upon capture, demonstrating class reuse and dependency management thus satisfying: FR_SCORE_CALC and UR_EVENTS.

Figure 5: Longboi class diagram (Hidden event)



This diagram illustrates how the hidden event links class associations between Player, Potion and Longboi. Collisions with Potion (TriggerA) activate EventStatus.hasPotion and returning to the MouseHole tile triggers the rendering of Longboi, awarding bonus points. This logic satisfies UR_EVENTS, FR_SCORE_CALC, FR_EVENT by using trigger systems and shared components to coordinate a multi-step event hidden from the player.

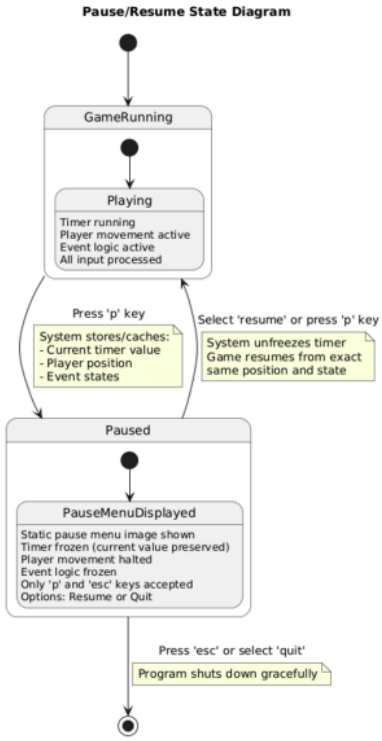
Figure 6: Negative event behavioural sequence diagram



The diagram above illustrates the full negative event process triggered when the player is detected and captured by the patrolling Dean. The sequence includes branching edge case parallel event logic handled without conflict. Initially adapted from earlier use cases this diagram evolved iteratively to update event logic following re-evaluation of requirements and testing feedback. The diagram demonstrates detection, input halting, score modifiers, resetting event logic and ensuring reliable failState handling smoothly, all within the OOP

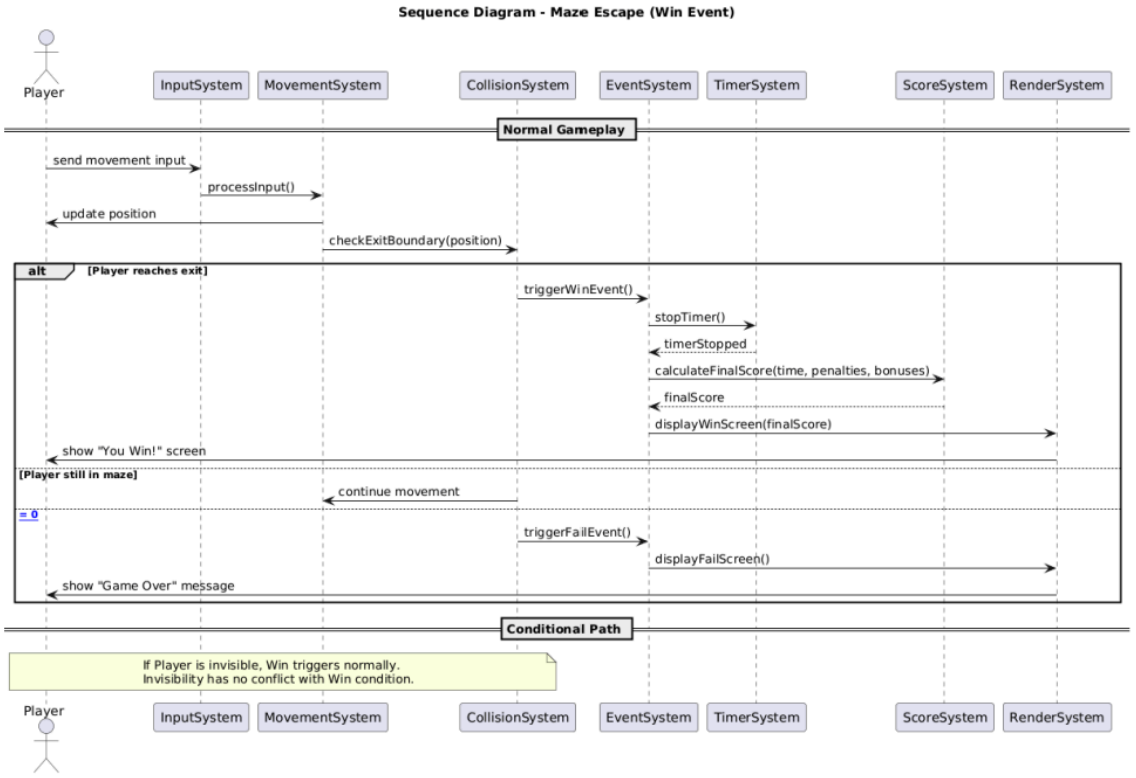
inheritance hierarchy, satisfying UR_EVENTS, FR_SCORE_CALC, FR_TIMER, NFR_RESILIENCE.

Figure 7: Pause behavioural state diagram



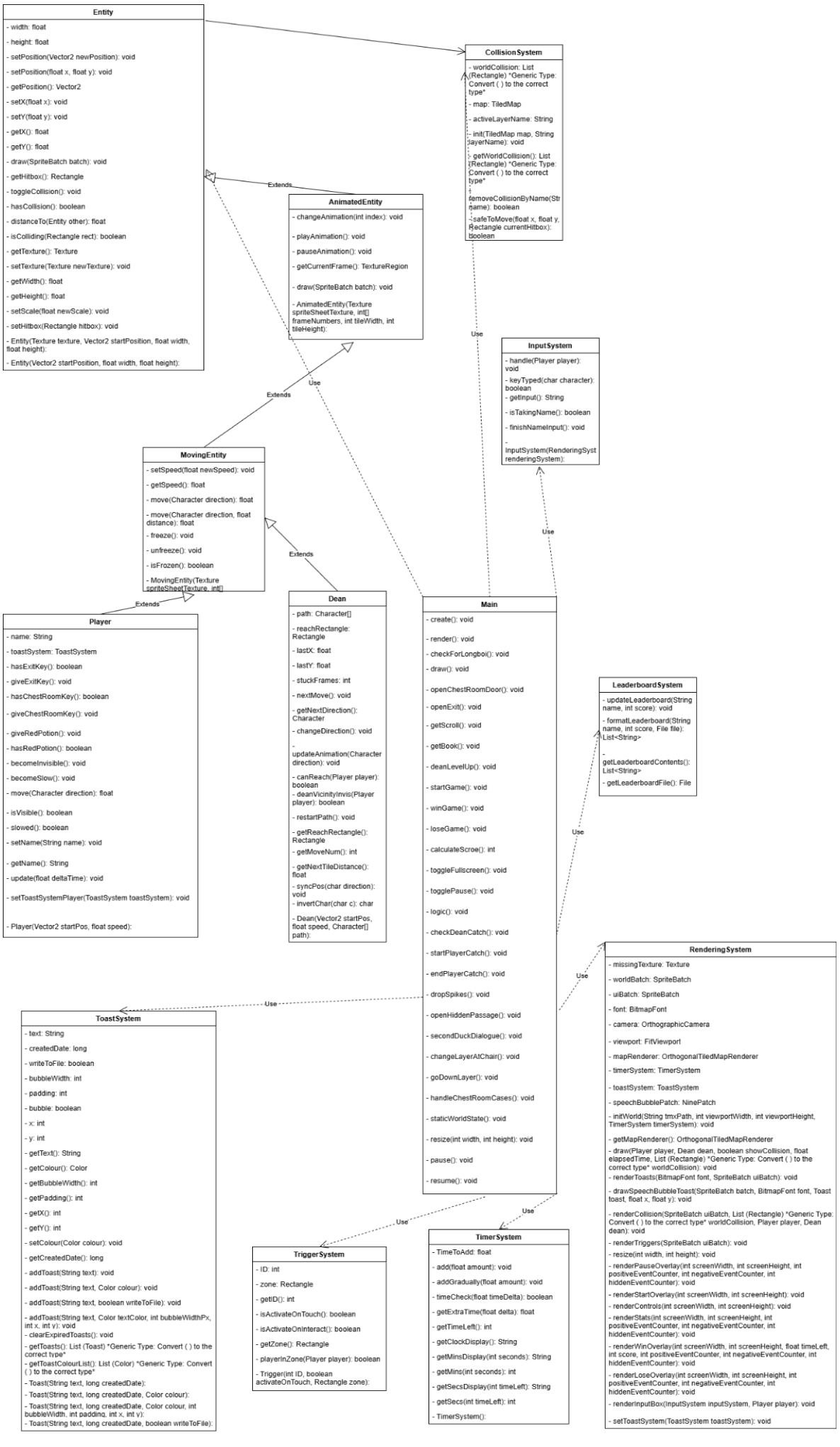
A state diagram was an intuitive choice for illustrating the pause game logic behaviour before and after the player inputs the pause key (P). In accordance with the user requirements, the game must include a pause feature that halts all game logic including the timer, ignoring all player input with exception to menu keys such as P (resume) and Esc (quit). The pause key triggers a static image screen representing a menu displaying the pause/resume and quit keys along with simple directional key instructions and a catalogue of the events you have interacted with so far. Upon resuming all game logic including events and timer is unfrozen successfully addressing: UR_PAUSE, FR_TIMER, UR_ACCESSIBILITY, NFR_USABILITY, NFR_UI and NFR_RELIABILITY (quit key).

Figure 8: Maze Escape sequence diagram



This diagram illustrates a complete successful maze escape run with event conflict handling. It was adapted from initial use cases and updated following programming team feedback and requirements re-evaluation. The sequence shows the player triggering the win event only when crossing the exit boundary while the timer is greater than zero. If the timer reaches zero the failEvent is triggered instead displaying the failure screen (see Design Iterations menu screens) skipping score calculation. Triggering the win event halts the timer and all other game logic, calculates the player score based on time remaining and any event bonuses or penalties thus satisfying: UR_VICTORY_CONDITION, UR_SCORE and FR_SCORE_CALC.

Figure 9: Complete class diagram for updated code
This diagram shows how the different classes interact and connect to each other after the code was expanded to include the achievements and leaderboard system were added, and highlights some other errors made regarding which items and events are considered entities. The use of the OOP inheritance hierarchy allows new entity types to extend base classes as shown by the arrows labelled “use” and “extend” in the diagram, and systems encapsulate internal logic, but entities directly access Main's static system references.


Entity
- width: float
- height: float
- setPosition(Vector2 newPosition): void
- setPosition(float x, float y): void
- getPosition(): Vector2
- setX(float x): void
- setY(float y): void
- getX(): float
- getY(): float
- draw(SpriteBatch batch): void
- getHitbox(): Rectangle
- toggleCollision(): void
- hasCollision(): boolean
- distanceTo(Entity other): float
- isColliding(Rectangle rect): boolean
- getTexture(): Texture
- setTexture(Texture newTexture): void
- getWidth(): float
- getHeight(): float
- setScale(float newScale): void
- setHitbox(Rectangle hitbox): void
- Entity(Texture texture, Vector2 startPosition, float width, float height):
- Entity(Vector2 startPosition, float width, float height):
CollisionSystem
- worldCollision: List (Rectangle) *Generic Type: Convert ( ) to the correct type*
- map: TiledMap
- activeLayerName: String
- init(TiledMap map, String layerName): void
- getWorldCollision(): List (Rectangle) *Generic Type: Convert ( ) to the correct type*
- removeCollisionByName(String name): boolean
- safeToMove(float x, float y, Rectangle currentHitbox): boolean
Extends
AnimatedEntity
- changeAnimation(int index): void
- playAnimation(): void
- pauseAnimation(): void
- getCurrentFrame(): TextureRegion
- draw(SpriteBatch batch): void
- AnimatedEntity(Texture spriteSheetTexture, int[] frameNumbers, int tileWidth, int tileHeight):
Use
Extends
InputSystem
- handle(Player player): void
- keyTyped(char character): boolean
- getInput(): String
- isTakingName(): boolean
- finishNameInput(): void
- InputSystem(RenderingSystem renderingSystem):
MovingEntity
- setSpeed(float newSpeed): void
- getSpeed(): float
- move(Character direction): float
- move(Character direction, float distance): float
- freeze(): void
- unfreeze(): void
- isFrozen(): boolean
- MovingEntity(Texture spriteSheetTexture, int[]
Extends
Extends
Dean
- path: Character[]
- reachRectangle: Rectangle
- lastX: float
- lastY: float
- stuckFrames: int
- nextMove(): void
- getNextDirection(): Character
- changeDirection(): void
- updateAnimation(Character direction): void
- canReach(Player player): boolean
- deanVicinityInvis(Player player): boolean
- restartPath(): void
- getReachRectangle(): Rectangle
- getMoveNum(): int
- getNextTileDistance(): float
- syncPos(char direction): void
- invertChar(char c): char
- Dean(Vector2 startPos, float speed, Character[] path):
Player
- name: String
- toastSystem: ToastSystem
- hasExitKey(): boolean
- giveExitKey(): void
- hasChestRoomKey(): boolean
- giveChestRoomKey(): void
- giveRedPotion(): void
- hasRedPotion(): boolean
- becomeInvisible(): void
- becomeSlow(): void
- move(Character direction): float
- isVisible(): boolean
- slowed(): boolean
- setName(String name): void
- getName(): String
- update(float deltaTime): void
- setToastSystemPlayer(ToastSystem toastSystem): void
- Player(Vector2 startPos, float speed):
Main
- create(): void
- render(): void
- checkForLongboi(): void
- draw(): void
- openChestRoomDoor(): void
- openExit(): void
- getScroll(): void
- getBook(): void
- deanLevelUp(): void
- startGame(): void
- winGame(): void
- loseGame(): void
- calculateScroe(): int
- toggleFullscreen(): void
- togglePause(): void
- logic(): void
- checkDeanCatch(): void
- startPlayerCatch(): void
- endPlayerCatch(): void
- dropSpikes(): void
- openHiddenPassage(): void
- secondDuckDialogue(): void
- changeLayerAtChair(): void
- goDownLayer(): void
- handleChestRoomCases(): void
- staticWorldState(): void
- resize(int width, int height): void
- pause(): void
- resume(): void
Use
Use
Use
Use
Use
Use
LeaderboardSystem
- updateLeaderboard(String name, int score): void
- formatLeaderboard(String name, int score, File file): List<String>
- getLeaderboardContents(): List<String>
- getLeaderboardFile(): File
RenderingSystem
- missingTexture: Texture
- worldBatch: SpriteBatch
- uiBatch: SpriteBatch
- font: BitmapFont
- camera: OrthographicCamera
- viewport: FitViewport
- mapRenderer: OrthogonalTiledMapRenderer
- timerSystem: TimerSystem
- toastSystem: ToastSystem
- speechBubblePatch: NinePatch
- initWorld(String tmxPath, int viewportWidth, int viewportHeight, TimerSystem timerSystem): void
- getMapRenderer(): OrthogonalTiledMapRenderer
- draw(Player player, Dean dean, boolean showCollision, float elapsedTime, List (Rectangle) *Generic Type: Convert ( ) to the correct type* worldCollision): void
- renderToasts(BitmapFont font, SpriteBatch uiBatch): void
- drawSpeechBubbleToast(SpriteBatch batch, BitmapFont font, Toast toast, float x, float y): void
- renderCollision(SpriteBatch uiBatch, List (Rectangle) *Generic Type: Convert ( ) to the correct type* worldCollision, Player player, Dean dean): void
- renderTriggers(SpriteBatch uiBatch): void
- resize(int width, int height): void
- renderPauseOverlay(int screenWidth, int screenHeight, int positiveEventCounter, int negativeEventCounter, int hiddenEventCounter): void
- renderStartOverlay(int screenWidth, int screenHeight): void
- renderControls(int screenWidth, int screenHeight): void
- renderStats(int screenWidth, int screenHeight, int positiveEventCounter, int negativeEventCounter, int hiddenEventCounter): void
- renderWinOverlay(int screenWidth, int screenHeight, float timeLeft, int score, int positiveEventCounter, int negativeEventCounter, int hiddenEventCounter): void
- renderLoseOverlay(int screenWidth, int screenHeight, int positiveEventCounter, int negativeEventCounter, int hiddenEventCounter): void
- renderInputBox(InputSystem inputSystem, Player player): void
- setToastSystem(ToastSystem toastSystem): void
ToastSystem
- text: String
- createdDate: long
- writeToFile: boolean
- bubbleWidth: int
- padding: int
- bubble: boolean
- x: int
- y: int
- getText(): String
- getColour(): Color
- getBubbleWidth(): int
- getPadding(): int
- getX(): int
- getY(): int
- setColour(Color colour): void
- getCreatedDate(): long
- addToast(String text): void
- addToast(String text, Color colour): void
- addToast(String text, boolean writeToFile): void
- addToast(String text, Color textColor, int bubbleWidthPx, int x, int y): void
- clearExpiredToasts(): void
- getToasts(): List (Toast) *Generic Type: Convert ( ) to the correct type*
- getToastColourList(): List (Color) *Generic Type: Convert ( ) to the correct type*
- Toast(String text, long createdDate):
- Toast(String text, long createdDate, Color colour):
- Toast(String text, long createdDate, Color colour, int bubbleWidth, int padding, int x, int y):
- Toast(String text, long createdDate, boolean writeToFile):
TriggerSystem
- ID: int
- zone: Rectangle
- getID(): int
- isActivateOnTouch(): boolean
- isActivateOnInteract(): boolean
- getZone(): Rectangle
- playerInZone(Player player): boolean
- Trigger(int ID, boolean activateOnTouch, Rectangle zone):
TimerSystem
- TimeToAdd: float
- add(float amount): void
- addGradually(float amount): void
- timeCheck(float timeDelta): boolean
- getExtraTime(float delta): float
- getTimeLeft(): int
- getClockDisplay(): String
- getMinsDisplay(int seconds): String
- getMins(int seconds): int
- getSecsDisplay(int timeLeft): String
- getSecs(int timeLeft): int
- TimerSystem():

This document, supported by the Design Iterations evidence, presents the complete architectural lifecycle of ‘Escape from Uni’ following the 5 stages of software development (Sommerville, 2016). The final design integrates a closed-layered architecture with an embedded OOP inheritance hierarchy with a helper system to achieve maintainability, modularity and scalability while meeting core user and system requirements. Each UML diagram was iteratively developed alongside the requirements and characteristics referencing systems, demonstrating traceability between system, behaviour and structure of the chosen architecture.

References

Sommerville, I. (2016) Software Engineering. 10th edn. Harlow: Pearson Education Limited.
Richards, M. and Ford, N. (2020) Fundamentals of Software Architecture. Sebastopol, CA: O'Reilly Media.
University of York (2025) ENG1: Software Architecture Lecture Slides. Department of Computer Science, University of York
PlantUML (2025) UML Diagramming Tool. Available at: https://plantuml.com (Accessed: 9 November 2025).
draw.io (2025) Diagramming and Visual Design Tool. Available at: https://www.draw.io (Accessed: 9 November 2025).