Architecture

Cohort 4 Group 6

Javengers

Braithwaite, Max
Faruque, Amber
Fu, Zhuoran
Kocaman, Melike
McDermott, John
Rissen, James
Scott, Charlotte

D3a - Statement on Tools & Languages

(All Blue Highlighted Text is Changed for Assessment 2)

This section outlines the architecture and design approach used to develop the university escape maze game *QUACK?*, a Java-based game built using the LibGDX framework. It summarises the project's architectural style, tools used, and design principles supporting development.
The architecture follows a layered, modular object-oriented structure, separating the game logic, UI and technical systems. Each class represents a logical component, for example player or energy, encouraging high cohesion and low coupling.

Practical application of this architecture:

Presentation Layer - easily maps to LibGDX's Screen/stage system

Domain Layer - Aligns with the entity/component logic implemented

Infrastructure Layer - Corresponds to AssetManager, AudioManager and other utilities/classes

Abstract Architectural tools & notation:

UML was used to produce the class and behavioural diagrams. This allowed us to create a layout for the architecture of the system as well as any layer interactions and separations.

- Class diagrams: Emphasize the structural relationships between our classes (inheritance, composition) to help clarify architectural organization and dependencies.
- Sequence diagrams: Utilize lifelines and messages to document the interactions between architectural layers.

PlantUML was used to create these diagrams due to its in-depth documentation and simplicity in its syntax which allowed for efficient development of these diagrams. It also provided good implementation with the version control of GitHub being able to verify and review changes for diagrams

Architectural Style & Justification

The abstract architectural style of a layered design ensures strong separation of concerns, such as logic, technical systems and interface. Separating the various classes within the product brief allows for a more organised and efficient approach to development. We planned the start of our implementation based on splitting up the classes between our implementers so the class diagrams were vital to our organisation. By clearly dividing responsibilities, each team member can focus on specific classes, ensuring that they are completed to a high standard. Additionally, allowing team members to choose the classes they feel most confident or experienced in ensures better quality work, increases motivation, and leverages individual strengths. It also allows us to work in parallel as we can use our time more efficiently. This also helped reduce merge conflicts since most layers are relatively

isolated within themselves, barring a few method calls or initializing variables. Along with this, we also decided to use some OOP concepts and paradigms such as inheritance and polymorphism to create easier patterns and extensibility for key game features that may need to be extended in the future (e.g. Events, animation, etc.).

For our concrete architecture we decided it would be best to use libGDX and take advantage of the libGDX Screen framework due to the easier formatting of code for screen deposing and implementing. Although this also gave the constraints of the libGDX screen framework and slightly altering the layered format, it was a trade-off for the proven working screen lifecycle management and memory safety benefits.

The behavioural diagrams allowed us all to have a thorough understanding on how the game would flow and how all of the elements would interact with each other. This diagram allowed us to effectively work out which parts needed to be implemented first so that delays due to dependencies are limited. It also clarifies the way that the game will flow so that the chances of us having to do extra coding due to miscommunications in how the game will run is limited.

Overall, our architectural goals were to promote modularity and allow for scalability for future development. This would allow more efficient debugging and testing and allow each module to develop independently.

The architecture follows a layered, modular object-oriented structure:

Presentation layer ● All screen classes ● AchievementNotifcation ● Scene2D UI Components
Domain Layer ● Player, Energy, Timer ● Event hierarchy ● WorldMap + Room system
Infrastructure Layer ● AudioManager ● AchievementManager ● LeaderboardManager ● HeadlessTestPlatform ● JSON data structures

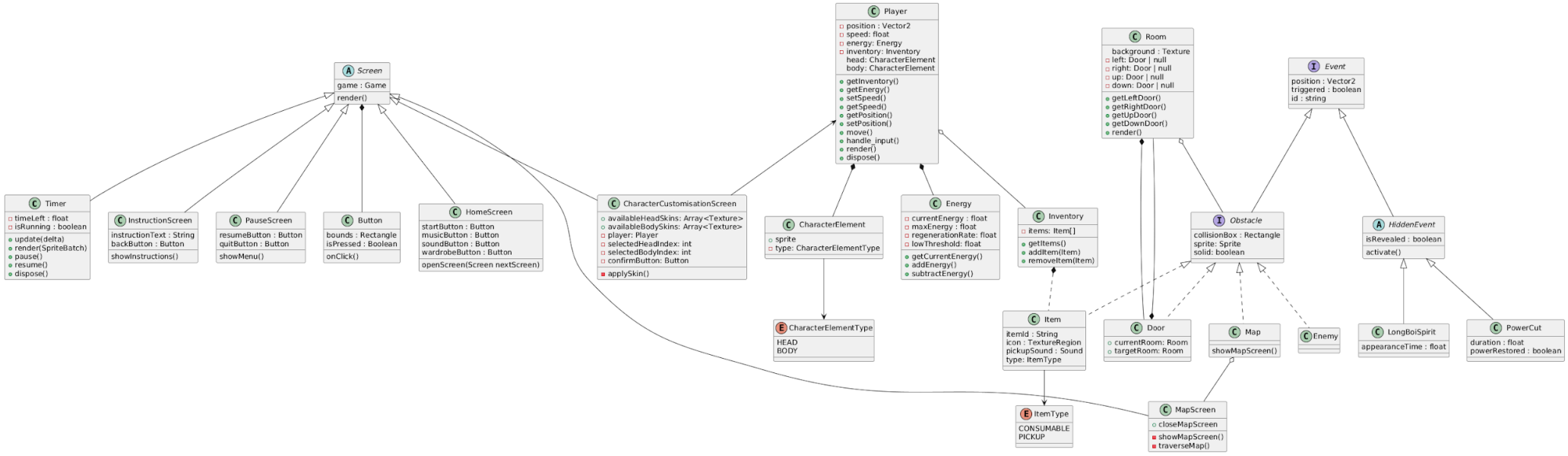
Development of architecture

The first iteration of the class diagram was based on the CRC cards that we completed collaboratively as a group. We then had a discussion about the ways in which we thought would be the most effective ways to organise our classes. With these we created a skeleton of the architecture using basic classes and relations. As we started implementing features we gained a better understanding of the game's direction and subsequently updated the diagrams to closely follow the game. We found that some of the classes were simply not needed as their features could be included more effectively in other classes. Also some classes were developed and expanded as our implementation progressed and our iterations of the class diagrams reflect those changes in class structure and mainly in the dependencies within our classes.

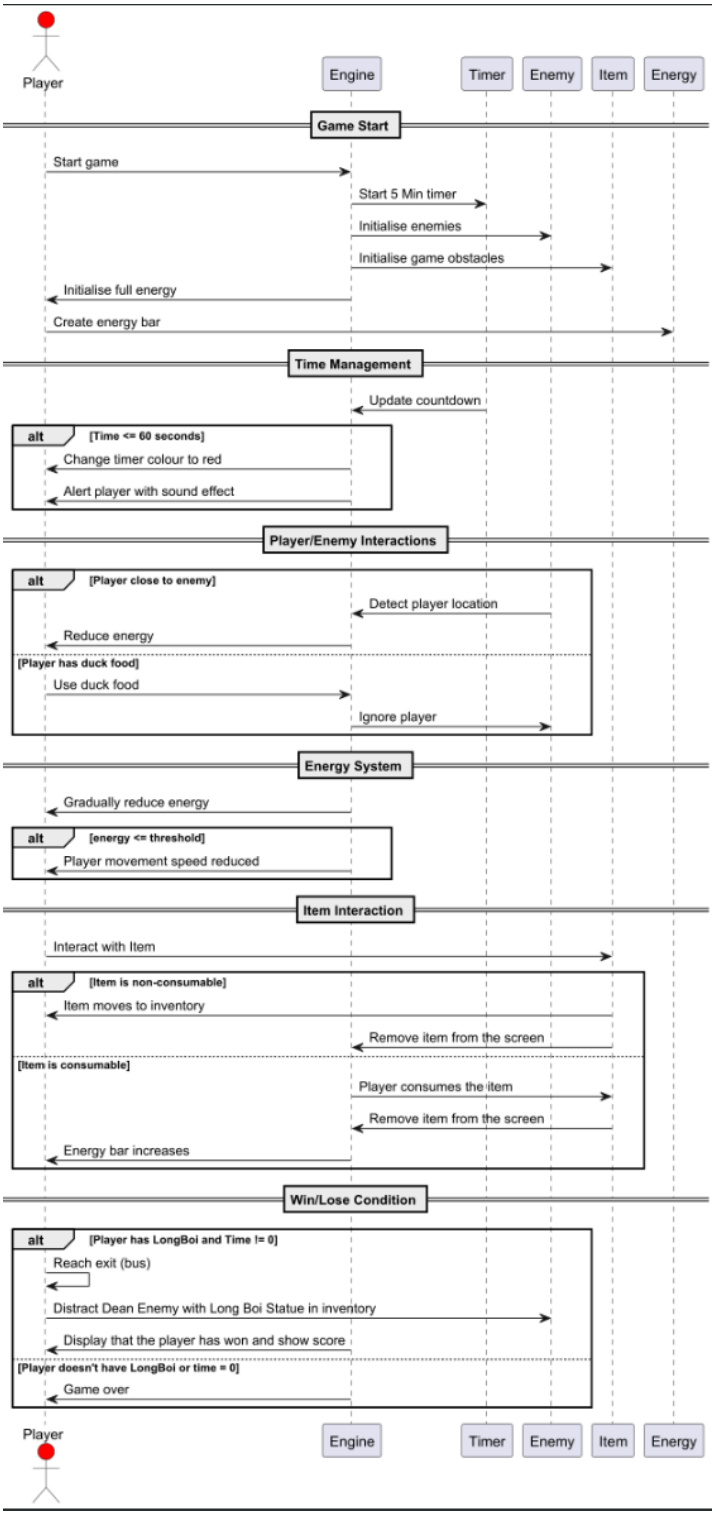
The first iteration of our behavioural diagram was made based on our group discussions about how we thought the game should run and how the different parts of the game would interact with each other. As implementation began, the behavioural diagrams became more complex and included more elements as we became clearer on some specific parts of the game. They were then split into smaller diagrams that work alongside the main diagram to provide even more detail that we needed at the later stages in implementation.

D3b - Structural UML Diagram (Link: whales-game.netlify.app/interim-uml-diagrams/)

High resolution copy available here

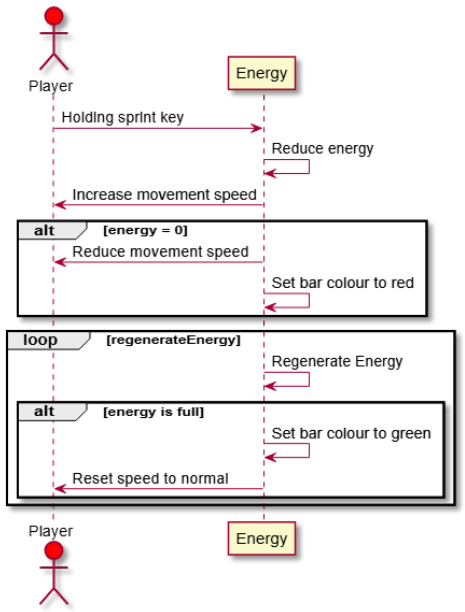


D3c - Behavioural UML Diagrams (Link: whales-game.netlify.app/interim-uml-diagrams/)



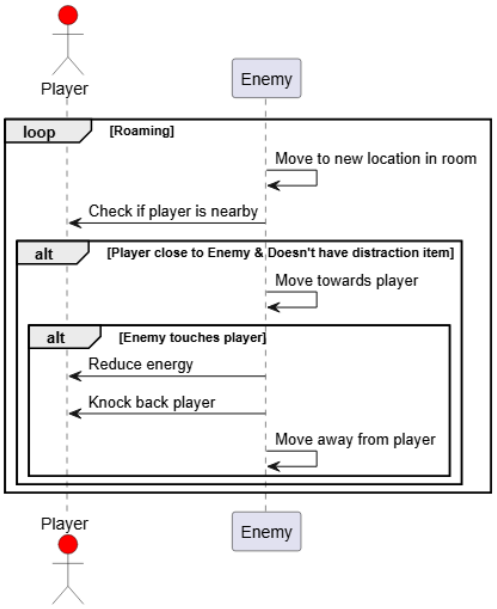
Energy

UR_ENERGY Requirement



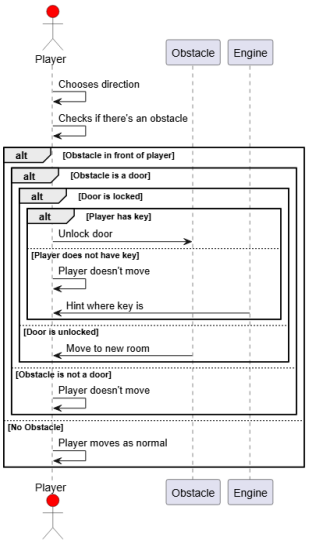
Enemies

Added as part of our events, for the UR_EVENTS Requirement. They move around randomly as part of fulfilling UR_EVENTS_LOCATION. Attacking players fulfils FR_NEGATIVE_EVENTS.



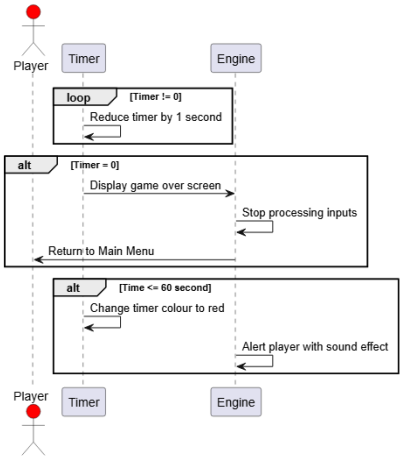
Movement and Obstacles

UR_MAP Requirement, with stationary screens to fulfil UR_DISPLAY. Movement fulfils UR_CONTROLS by keyboard controls.



Timer logic

FR_TIMER and FR_TIMER_END requirements.



D3d - Extension of WHALES Architecture for part 2:

PlantUML Diagram Iterations & Justification of Evolution:

Iteration 1 > 2: Inherited/Intermediate architectural structure

(figure 1a, appendix found below)

Infrastructure layer:

1. AudioManager Class created - Centralized previously scattered audio code into a single infrastructure layer service
2. HeadlessTestPlatform Added - Enable testing without any UI dependencies and supports continuous integration/Unit testing.

Benefits:

- Testability: Enables complete unit testing for previously untestable game logic
- Productivity: Allows easier parallel development with clear boundaries between responsibility
- Design: Follows a single responsibility principle, increasing extensibility and reducing buggy code due to overlap

Trade-offs:

- Added abstraction layer complexity
- Substantial setup time required
- Learning curve for unit testing practices

Architectural impact:

Technical concerns such as the audio and testing are properly moved from the domain layer (game logic) to the infrastructure layer. This strengthens the layer separation and amplifies benefits gained from this architectural design (productivity, reduced bugs, easier allocation of responsibilities/resources).

Domain Layer:

1. Collision method split - monolithic “collisions()” parted into specific triggers: checkCoffeeTrigger(), checkDuckTrigger(), checkVendingMachineTrigger(), checkPowerCutTrigger(), checkItemTrigger()
2. Duck logic encapsulation - Moved duck related behavior from general game class to a dedicated duck class.
3. WindEvent class added - New event type extending the existing event interface
4. Animator class hierarchy - Animation system with player (or other sprites) extending animator

Benefits:

- Higher cohesion: Each method now has one single, clear purpose
- Maintainability: Easier to read, test and debug code
- Team efficiency: Clearer work assignment for trigger types
- Extensibility: New trigger types can be added without altering the existing code (creates a pattern for new additions)
- Design: Stays consistent with the WindEvent class, following established event patterns
- Reusability: Animator system can be used for other sprites
- Polymorphism: Using OOP paradigms allows for expansion without breaking existing code

Trade-offs:

- Increased method calls instead of one
- Marginal performance overhead
- Increased number of methods used in main game class
- Increased inheritance complexity
- More classes to maintain (although simpler)

Architectural Impact:

Improved cohesion through segregation of methods/responsibilities - replaced the monolithic handler with specialized methods (also improves parallel development opportunities). Domain layer expansion also creates more reusable patterns while maintaining consistency to allow for more easily extensible features.

<u>Presentation Layer:</u>

1. Scene2D UI Framework Integration - Added professional UI components (Stage, Table, Label)
2. Enhanced Input System - Mouse support alongside existing keyboard controls

3. Improved Resource Management - Proper stage disposal and viewport optimization

Benefits:

- User experience: Professional, industry-standard interface
- Accessibility: Multiple input methods (Keyboard + Mouse)
- Cross-platform: Better compatibility with different systems
- Memory Safety: Preventing leaks during long sessions

Trade-offs:

- Learning curve for new framework integration
- Increased memory usage
- More complex event handling architecture

Architectural Impact:

Presentation layer swapped from simple LibGDX rendering to a more professional Scene2D framework integration. This provided a scalable UI foundation that still maintains our layer separation while also enabling for more complex user interaction and keeping the resource lifecycle manageable.

Trade-off Analysis:

Created/refined a clean layered architecture, extracted services, established testing infrastructure and implemented basic patterns/managers.

Negative impact	Benefits	Net result
More abstraction	Better testability	Positive
More classes/methods	Higher cohesion	Positive + increased maintainability
Framework dependant	Industry level UI	Positive + improved user experience
Increased inheritance	Reusable code structure	Neutral (depends on additions)

Iteration 2 -> 3: Intermediate/Final development

(figure 1b, appendix found below)

Infrastructure layer:

1. Achievement Added - Centralized achievement tracking with serialization
2. LeaderboardManager Improved - Extended with time-based scoring and player names
3. AudioManager Improved - Added frog/gulp/swipe sounds and volume control methods
4. JSON Data Structure Formalized - JsonRoom/JsonObstacle classes created for more structured map loading (less repeating loads)

Benefits:

- Achievements: Helps visual testing of function/events being completed and keeps player progress visible
- Scoring Changes: Time-based bonuses encourage replayability and speedrunning
- Audio Completeness: Full sound coverage for all major game events
- Service Interface Consistency: All managers follow a similar pattern of options (creating, updating, deleting the managed assets)
- Structured JSON Data: Clean separation of between code and configuration (keeping with the separation of concerns theme with the layers)

Trade-offs:

- Serialization: Creates small overhead due to file I/O operations and adds complexity
- Configuration Management: JSON files will require separate maintenance
- Audio assets: More sounds can increase download size (negligible)
- Interface Complexity: Multiple manager classes can increase initial complexity (scales better)

Architectural Impact:

Infrastructure continues to adapt more “service” related elements such as scoring, achievement tracking, and other configuration management for loading JSON files. This further establishes the boundary between layers and reinforces our proven/validated architecture, allowing for easier management allocation for tasks when it comes to managing configuration separately from game logic, screens, etc.

Presentation layer:

1. Achievement Screen Integration - AchievementScreen and AchievementNotification added for player feedback (also helps testing)

2. Active Score Updates - Dynamic scoring with bonuses from events + time multiplier at the end
3. Additional Screens - NameInputScreen, GameIntroScreen, WinScreen
4. Notification System - AchievementNotification provides immediate feedback in-game.

Benefits:

- Complete User Experience: Complete and tracks user progress with proper visible completion metrics
- User Incentive: Encourages exploration and replayability
- Improved User Feedback: Notifications reinforce player actions via visual feedback and can be reused for other feedback testing
- Polished System Flow: More screens create a more professional system flow and options for the user without information being overwhelming on a single screen

Trade-offs:

- Screen Management Complexity: Increased screen states to handle
- Memory Increase: Additional screens + notifications being loaded/rendered
- UI State Complexity: More conditional rendering logic
- Performance Overhead: Additional rendering for notifications + screens
- Framework Complexity: Increased integration of Scene2D reduces other UI flexibility

Architectural Impact:

Presentation layer changes from basic rendering layer to full UI management system. This demonstrates the scalability of Scene2D framework for more complex screen flows.

<u>Domain layer:</u>

1. Complex Event Additions - Frog which chases the player, security with patrol area and DuckStatue puzzles
2. Inheritance of Animation System - Frog and security extend animator (mentioned in first iteration) with unique behaviours
3. Event Pattern Consistency - HiddenEvent class (abstract) is formalized with
4. Room Transition Logic - WorldMap management improved

Benefits:

- Gameplay Depth: Complex event additions create more engaging challenges

- Template Method Pattern: HiddenEvent abstracts from normal events and provides a pattern for future events of the same type
- Interface Segregation: Only events that need HiddenEvent features will implement these features/patterns

Trade-offs:

- Increased Inheritance Hierarchy: Large animator inheritance can reduce flexibility
- Bug-Prone: Multiple states for different entities can increase risk of bugs (difficult to find)
- Temporal Coupling: Room transitions must be executed in a certain order with certain state changes

Architectural Impact:

The domain layer applies the same animation infrastructure but implements unique behaviours in each entity's update() methods, which demonstrates the polymorphic design applicable in our architecture.

Trade-off Analysis:

Applied architectural patterns (e.g. service layer within infrastructure) at scale, added more professional features via frameworks while maintaining layer boundaries and validated architectural scalability while increasing testability.

Negative impact	Benefits	Net result
Increased state complexity	More formalized screen pattern	Positive (more professional UI flow)
Persistence Coupling with JSON saves	Data-driven architecture (enables configuration changes without recompilation of code)	Positive (maintainable content/data pipeline)
Memory management increase	More complete UI system	Positive (better UX)
Interface management increase (increase manager interfaces)	Service layer pattern (most managers follow similar patterns of power over their features)	Positive (easier foundation for testing with managers)

Sequence Diagram Analysis:

Movement sequence (figure 2a):

This diagram demonstrates player input processing with WindEvent integration and Scene2D input handling. It shows how keyboard/mouse input captured by the Presentation layer, triggers velocity calculation (Domain) with wind effect modification, while AudioManager (Infrastructure) provides feedback. This sequence validates our layered separation with clear cross-layer interactions across Presentation (input), Domain (movement logic. + WindEvent) and Infrastructure (delta time), while also demonstrating the polymorphism implemented in the animation class.

Energy sequence (figure 2b):

Illustrates our updated energy management with tick-based regeneration and accessibility states. The sequence shows Energy.tick() processing delta time for automatic regeneration, with energy reduction during sprinting through Player.handle_input(). This demonstrates proper encapsulation of the energy class managing its own state and validates our single responsibility principle since the energy logic is centralized rather than scattered. This improves testability and maintainability.

Timer sequence (figure 2c):

Documents the timer implementation transitioning from simplistic second-counting to LibGDX-integrated, frame-rate independent timing. It demonstrates clean game state management with the timer self managing its GAME OVER transition and UI updates, while the presentation layer only initiates the rendering. This represents the architectural improvement from assessment 1's basic timing to a system with more memory management and user friendly formatting.

Coffee Audio sequence (figure 2d):

Demonstrates our centralised audio architecture through collision-triggered sound queues. When the specific "checkCoffeeTrigger()" detects a collision, the sequence shows AudioManager playing volume-controlled sound through "playCoffee()" without any scattered libGDX dependencies. This specifically evidences/validates one of our key architectural changes involving refactoring the scattered audio code

into a single infrastructure layer service. It also validates the reduction of coupling from multiple classes as well as enabling better test mocking and consistent volume control.

Project Modularity Assessment:

Architectural characteristics:

- Maintainability: improved - Method splitting and AudioManager extraction make changes easier and less bug-prone
- Testability: Greatly improved - HeadlessTestPlatform enables proper unit testing without needing the game UI
- Performance: Maintained - Delta-time calculations and optimized viewport keep game running smoothly even after adding more events
- Extensibility: Improved - Clear patterns (Event interface, animations, etc.) make adding new features straightforward

Cohesion Analysis:

- High Cohesion: AchievementManager (only handles achievements), Timer (self-contained timing), Specialized triggers/methods
- Medium Cohesion: WorldScreen (coordinates multiple subsystems/screens and events but follows the screen pattern), Player (handles input/movement and animation but follows the entity pattern)

Coupling Analysis:

- Reduced Coupling: Service interfaces decouple game logic from data configuration files and external systems
- Controlled Coupling: Screen transitions follow state pattern with clear entry/exit points
- Pattern Coupling: Pattern of animator inheritance creates manageable coupling for behaviour variety in different events/entities.

Layer Boundaries:

- Presentation: Complete screen system with clear state transitions and maintains the focus on UI/UX
- Domain: Entity/logic system with behavioural patterns and minimal infrastructure dependencies
- Infrastructure: Full service layer with configuration, managers and external frameworks

CRC Cards & Traceability Matrix: (link: QUACK? | Planning)

This traceability matrix demonstrates complete coverage of all functional requirements (FR) through our class implementations.

Requirement ID	**Requirement Description**	**CRC Cards Involved**	**Implementation Details**
UR_MAP	Player moves through various areas on a hidden map	WorldMap, JsonRoom, JsonObstacle, WorldScreen	Multi-room navigation with JSON configuration, room transitions
UR_EVENTS	Events triggered at visible/hidden checkpoints	Event, HiddenEvent, Frog, Security, SlipOnCoffee, DuckStatue, WindEvent, PowerCut, RamblingProfessorEvent	Complete event system with interface hierarchy, multiple event types
UR_EVENTS_LOCATION	Hidden/visible events in random positions each play	WorldMap, JsonRoom, JsonObstacle	JSON configuration enables procedural room/obstacle placement
UR_ENERGY	Energy/health levels changed by events	Energy, Player	Tick-based regeneration, sprint energy consumption,

			accessibility states
UR_MUSIC	Background music with volume control	AudioManager	Centralized audio service with play/stop/volume/mute controls
UR_SOUND_EFFECTS	Sound effects triggered by special events	AudioManager, WorldScreen, Event classes	Event-triggered sounds managed through AudioManager
UR_INSTRUCTION	One-page instruction screen	GameIntroScreen	Dedicated introduction screen with game controls
UR_TIMER	Timer counting down while playing	Timer, WorldScreen	Delta-time calculation, visible countdown, professional formatting
UR_TIME_LIMIT	Game completed in under 5 minutes	Timer, Main	5-minute countdown with game over logic
UR_DISPLAYS	Suitable for large monitor display	All Screen classes, AchievementNotificationScene	2D framework, professional UI, multiple screen sizes

UR_CONTROLS	Mouse or keyboard character movement	Player, WorldScreen	Keyboard (WASD/Arrows) + mouse input, Scene2D integration
UR_MENU	Menu system for game navigation	HomeScreen, Main, All Screen classes	Complete menu system with screen transitions
UR_UI	Pleasing, easy-to-navigate UI	All Screen classes, AchievementNotification	Professional UI with consistent navigation patterns
FR_POSITIVE_EVENTS	Positive effects benefiting character (x3)	Item, DuckStatue, AchievementManager	Collectible items, statue completion, achievement rewards
FR_DRINK_POSITIVE	Vending machine temporary buff	Item (CONSUMABLE type)	Consumable item system with temporary effects
FR_SCORE_BONUS	Map items increase score	Score, AchievementManager	Achievement-based scoring, time bonuses
FR_NEGATIVE_EVENTS	Negative effects hindering	Frog, Security, SlipOnCoffee, WindEvent, PowerCut	Multiple negative events with varied penalties

	gameplay (x5)		
FR_LIGHTS_NEGATIVE	Vision loss from lights failing	PowerCut	Darkness effect during power cuts
FR_SECURITY_NEGATIVE	Apprehended by campus security	Security, DuckStatue	Security patrols, statue protection mechanics
FR_COFFEE_NEGATIVE	Slip in coffee slows player	SlipOnCoffee	Movement penalty with slip effect
FR_FROG_NEGATIVE	Frog interaction slows player	Frog	Chasing AI with player slowdown
FR_HIDDEN_EVENTS	Surprise events on collision (x3)	HiddenEvent, WindEvent, RamblingProfessorEvent, PowerCut	Abstract hidden event system with multiple implementations
FR_WIND_HIDDEN	Windy effect on main path	WindEvent	Wind effects modifying player movement speed
FR_PROFFESSOR_HIDDEN	Professor providing no assistance	RamblingProfessorEvent	Professor interaction system

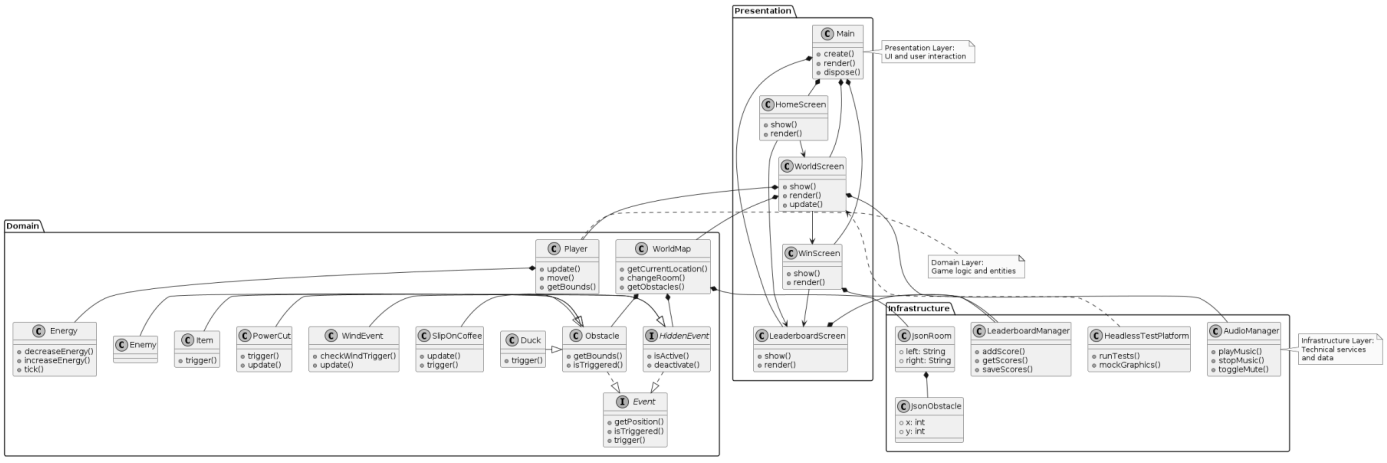
FR_ENERGY	Energy system for sprinting	Energy, Player	Sprint system with energy consumption/regeneration
FR_MUSIC	Background music during play	AudioManager	Continuous background music with looping
FR_MUSIC_VOLUME	Adjustable/mutable music volume	AudioManager	Volume slider and mute toggle functionality
FR_SOUND_EFFECTS	Event-triggered sound effects	AudioManager, Event classes	Centralized sound effect management
FR_INSTRUCTION	Read game controls from screen	GameIntroScreen	Clear instruction text and control diagram
FR_BUTTON	Clickable buttons for navigation	All Screen classes	Scene2D button system throughout all screens
FR_MAIN_MENU	Central navigation menu	HomeScreen, Main	Main menu with game start, leaderboard, achievements

FR_PAUSE	Pause button stopping timer	WorldScreen, Timer	Pause functionality with timer suspension
FR_LEADERBOARD	Accessible leaderboard menu	LeaderboardScreen, LeaderboardManager	Complete leaderboard with score saving/loading
FR_TIMER	On-screen countdown during play	Timer, WorldScreen	Visible timer display during gameplay
FR_TIMER_END	Game over on timer completion	Timer, Main	Game over logic triggered at timer zero
FR_CAMERA	Stationary camera per room	WorldScreen, WorldMap	Fixed camera with room transition system
NFR_LOADING	Game loads quickly (<5 seconds)	Main, All Managers	Efficient asset loading, asynchronous initialization
NFR_TIME_LIMIT	Game finishes within 5 minutes	Timer, Main	Enforced 5-minute gameplay limit
NFR_SHAPES	Shapes used alongside	All visual classes	Shape-based UI elements,

	colors for accessibility		colorblind-friendly design
CR_JAVA	Implemented in Java 17	All Code	Java 17 language features used throughout
CR_OPERATING_SYSTEMS	Runs on Windows, Mac, Linux	LibGDX Framework	Cross-platform LibGDX framework implementation

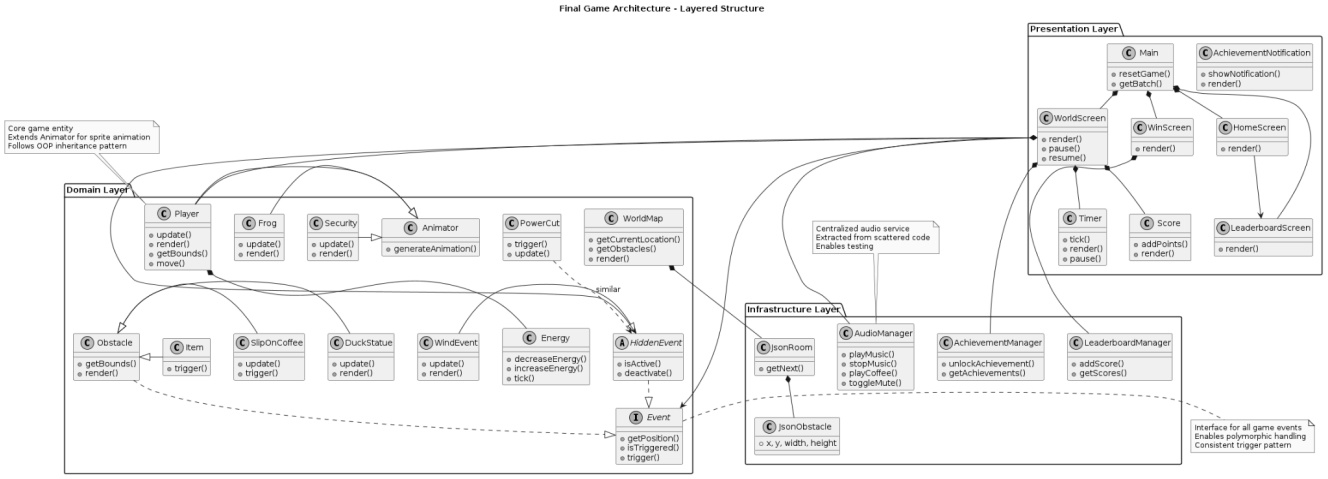
D3e - Continued Appendix:

Figure 1A:



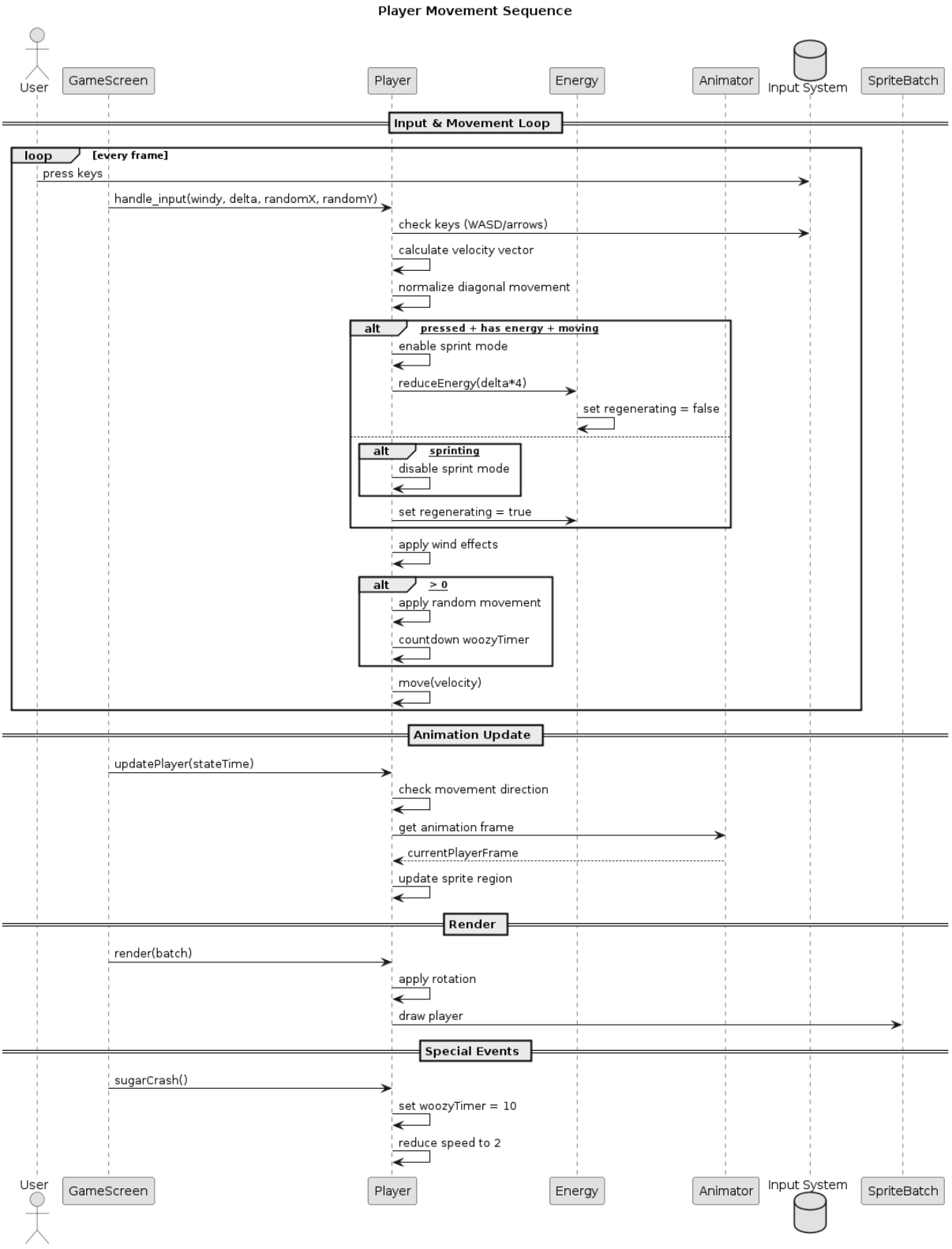
Link to view on website: [QUACK? | Planning]

Figure 1B:



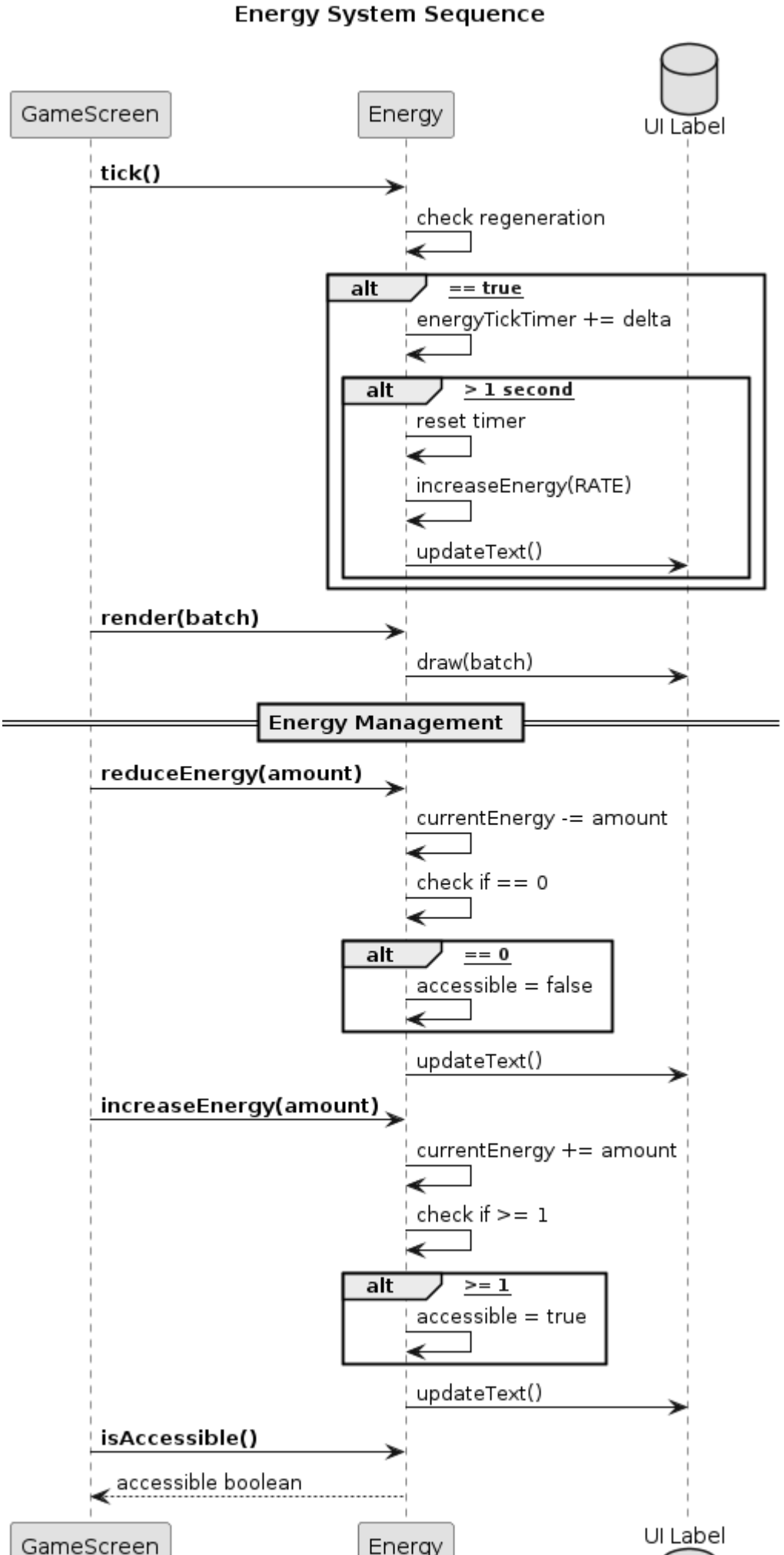
Link to view on website: [QUACK? | Planning]

2a: Movement sequence



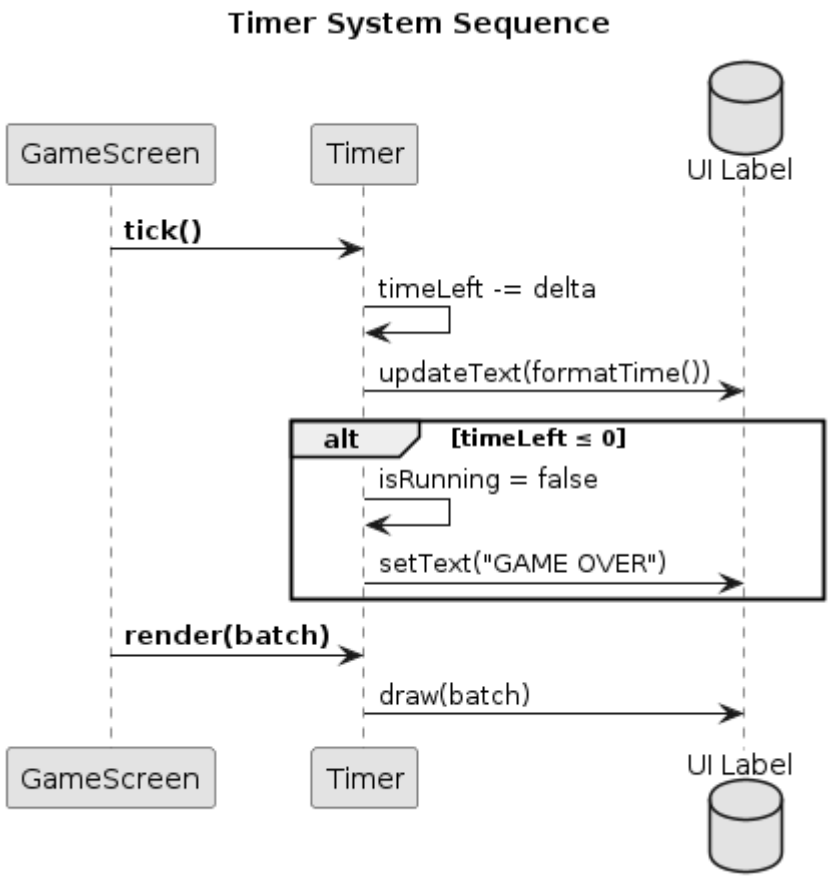
Link to view on website: [QUACK? | Planning]

2b: Energy Sequence


Energy System Sequence
GameScreen
Energy
UI Label
tick()
check regeneration
alt
== true
energyTickTimer += delta
alt
> 1 second
reset timer
increaseEnergy(RATE)
updateText()
render(batch)
draw(batch)
Energy Management
reduceEnergy(amount)
currentEnergy -= amount
check if == 0
alt
== 0
accessible = false
updateText()
increaseEnergy(amount)
currentEnergy += amount
check if >= 1
alt
>= 1
accessible = true
updateText()
isAccessible()
accessible boolean
GameScreen
Energy
UI Label

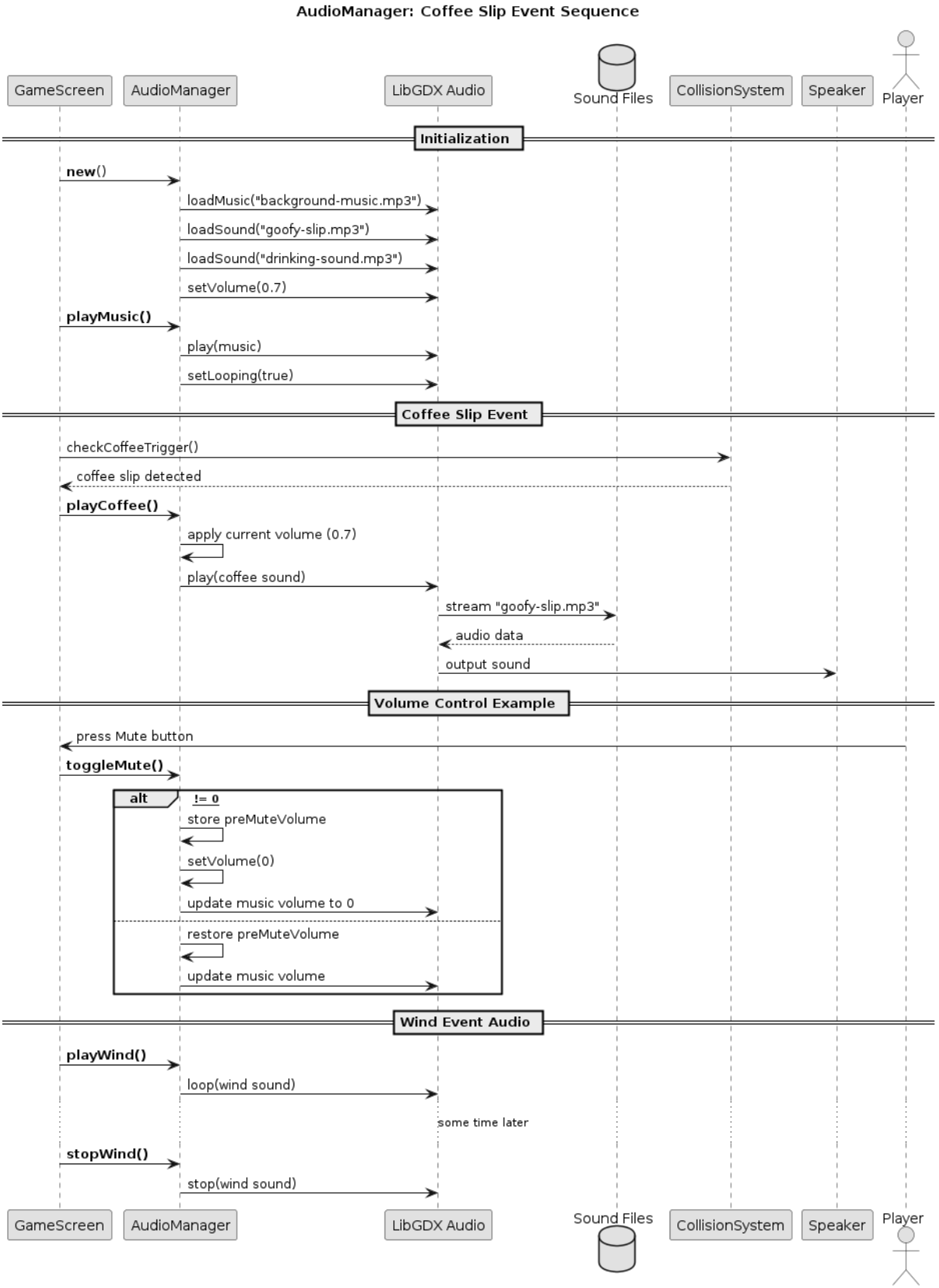
Link to view on website: [QUACK? | Planning]

2c: Timer sequence



Link to view on website: [QUACK? | Planning]

2d: CoffeeAudio Sequence



Link to view on website: [QUACK? | CRC Cards]

Figure 3a: Original CRC cards:

Goose / Enemy

- Chases the player and reduces their energy.
- Blocks the player's path. Can be distracted using duck food or a statue.
- Knows the player's energy level and reduces energy if available or causes the player to stop while energy regenerates.
- Knows what item the player currently has.
- Stereotypes: Controller, Coordinator

Inventory

- Shows icons of items (duck food, long boi statue) in the bottom-left corner to allow the player to see what they can use to get past enemies.
- Knows what items the player currently has access to.
- Can remove items once they have been used to distract/give to an enemy.
- Stereotypes: Information Holder, Service Provider

Character Element

- A segment of the character sprite which can be customised.
- Knows which parts of character are customisable.
- Allows browsing through items using arrows.
- Knows the items it can wear/ equip.
- Stereotypes: Information Holder, Service Provider

Player

- Moves around the screen and between rooms.
- Can interact with things in various ways.
- Energy (influences its speed).
- Knows its location in the room and overall in the map.
- Detects the key input / mouse (movement, sprint, pause, mute).
- Stereotypes: Controller, Coordinator, Interfacer

Door

- Provides a method to move between rooms. May be open or closed.
- Knows if there needs to be a key to open it and where it is on the map.
- Responsible for being an obstacle for the player and unlocking new parts of the map.
- Stereotypes: Structurer, Interfacer

Room

- Place where the player can explore and interact with game events.
- Knows its location and layout.
- It knows the events that can occur within.
- The room is a maze user input allows movement trough room keys.

- Stereotypes: Coordinator, Interfacer, Information Holder

Timer

- Counts down from 5 minutes.
- Warning after every minute/close to end?
- Records time and has a leaderboard.
- Ends game at zero.
- Knows: how long is left in the game. Where on the screen it should render. Whether it should render right now. Whether it should be kicking down.
- Responsible for: Ending the game. Ticking down.
- Stereotypes: Service Provider, Information Holder

Title / Home Screen

- Allows player to start game, view leaderboard, or change settings.
- Easy to navigate.
- Directs player during loading up/using.
- Knows: Whether an option has been pressed, where to take you
- Responsible for: Allowing the player to navigate through options like play, settings etc.

Music / Sound Controller

- Allows muting and volume adjustment.
- Controls background music and sound effects.
- Tracks mute state.
- Stereotype: Controller

Button

- Performs some action when clicked.
- Knows its screen position and state.
- Displays correct content.
- Stereotypes: Service Provider, Interfacer

Link to view on website: [QUACK? | CRC Cards]

Figure 3b: Final CRC Cards:

Class: Main

Responsibility	**Collaboration**
• Start the game	HomeScreen, WorldScreen, WinScreen

• Set and change screens • Control the overall game flow	

Class: Player

Responsibility	**Collaboration**
• Store player information • Update player state during the game • Allow events to change player values	WorldScreen, Energy, Animator

Class: Energy

Responsibility	**Collaboration**
• Store the player's energy value • Increase or decrease energy	Player

Class: Animator

Responsibility	**Collaboration**
• Load animations • Update animation frames	Player, Frog, Security

Class: WorldMap

Responsibility	**Collaboration**
• Store the game world layout • Track the player's current room • Control movement between rooms	JsonRoom, JsonObstacle, OrthogonalDirection, Obstacle

Class: JsonRoom

Responsibility	**Collaboration**
• Store room data loaded from JSON • Hold room description and exits	JsonObstacle, OrthogonalDirection

Class: JsonObstacle

Responsibility	**Collaboration**
Store obstacle data from JSON	JsonRoom, WorldMap

Interface: Event

Responsibility	**Collaboration**

• Base structure for all events • Provide shared behavior for displaying text, applying effects, etc.	Player, WorldScreen

Class: HomeScreen

Responsibility	**Collaboration**
• Display the main menu UI • Handle menu navigation	NameInputScreen, LeaderboardScreen, AchievementsScreen

Class: Event

Responsibility	**Collaboration**
• Base structure for all events • Provide shared behavior for displaying text, applying effects, etc.	Player, WorldScreen

Class: Item

Responsibility	**Collaboration**
• Represent a usable or collectable item	ItemType, Obstacle

Class: Frog

Responsibility	**Collaboration**
• Control frog behaviour and interaction for negative event of getting eaten by frog	Animator, Player, Main, WorldScreen

Class: Security

Responsibility	**Collaboration**
• Control the campus security character • Apply penalties to the player	WorldMap, WorldScreen, DuckStatue

Class: SlipOnCoffee

Responsibility	**Collaboration**
• Handle the coffee slip event • Apply effects to the player	Player, WorldScreen

Class: DuckStatue

Responsibility	Collaboration
● Control the duck statue event ● Check progress using statue pieces	DuckStatuePiece, WorldMap, Main

Class:DuckStatuePiece

Responsibility	Collaboration
● Represent a statue piece ● Support the statue event	WorldMap, Main

Class:HiddenEvent

Responsibility	Collaboration
● Act as a base for background events	WorldScreen, Event

Class: PowerCut

Responsibility	Collaboration
● Apply a power cut effect randomly if in Piazza	HiddenEvent, Main

Class:RamblingProfessorEvent

Responsibility	Collaboration
● Control the professor event	HiddenEvent, Main

Class: WindEvent

Responsibility	Collaboration
● Apply wind effects during gameplay	HiddenEvent, Main

Class: HomeScreen

Responsibility	Collaboration
● Show the main menu ● Handle menu input	Main, NameInputScreen, AchievementScreen, LeaderBoardScreen

Class: NameInputScreen

Responsibility	Collaboration
Get the player’s name	Main, HomeScreen, GameIntroScreen

Class: GameIntroScreen

Responsibility	Collaboration

• Show the game introduction • Start the game world	Main, WorldScreen, WorldMap, HomeScreen

Class: WorldScreen

Responsibility	**Collaboration**
• Main gameplay screen • Display the current Room and handle player movement • Trigger and run events • Manage game progression and win condition	Main, Player, WorldMap, Obstacle, Event, HiddenEvent, AudioManager, Timer, WinScreen, Item, ItemType

Class: WinScreen

Responsibility	**Collaboration**
• Show the win screen • Handle input after winning	HomeScreen, Main,LeaderboardManager, AchievementManager

Class: AudioManager

Responsibility	**Collaboration**
• Load game sounds and music • Play and stop sound effects and background music	WorldScreen

Class: Timer

Responsibility	**Collaboration**
• Track elapsed time or countdown time for gameplay • Provide timing values to screens/events	WorldScreen

Class: Score

Responsibility	**Collaboration**
• Store and manage the player's score	Main, LeaderboardManager

Class: LeaderboardManager

Responsibility	**Collaboration**
• Save and load scores • Sort leaderboard entries	LeaderboardData, ScoreEntry

Class: LeaderboardData

Responsibility	Collaboration
• Store leaderboard information	ScoreEntry

Class: ScoreEntry

Responsibility	Collaboration
• Store one player score	LeaderboardData

Class: AchievementManager

Responsibility	Collaboration
• Track achievement progress • Unlock achievements	Achievement, AchievementData

Class: Achievement

Responsibility	Collaboration
• Store one achievement	AchievementManager

Class: AchievementData

Responsibility	Collaboration
• Store saved achievement data	Achievement

Class: AchievementNotification

Responsibility	Collaboration
• Show achievement pop-ups	Achievement

Class: AchievementScreen

Responsibility	Collaboration
• Display achievements	AchievementManager Achievement