EyAy

Ozan Özak - 87242
Kaya Yaylalı - 82899
Deniz Ünal - 87041
Demir İyigün - 87534
Ege Töman - 87169
Yiğit Tutaş - 83482

Table of Contents

1. Logical Architecture (UML Package)

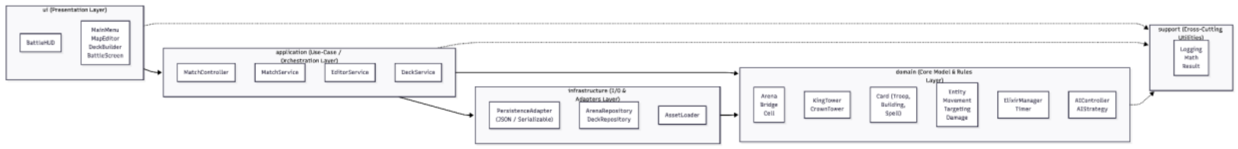
1.1 Purpose

The logical architecture divides the KU Royale system into clear, loosely-coupled layers that separate the user interface, application control logic, and domain rules.
This structure ensures maintainability, testability, and alignment with GRASP principles of *Low Coupling* and *High Cohesion*.

1.2 Package Diagram

Figure 1 below presents the overall logical architecture of the **KU Royale** system as a UML Package Diagram.
It illustrates the five main layers — *ui*, *application*, *domain*, *infrastructure*, and *support* — along with their dependencies and responsibilities.
The arrows indicate dependency directions, showing that control flows from the user interface down to the core model, while the *support* package provides cross-cutting utilities accessible to all layers.

Figure 1. Logical Architecture of KU Royale Project (UML Package Diagram)


ui (Presentation Layer)
BattleHUD
MainMenu MapEditor DeckBuilder BattleScreen
application (Use-Case / Orchestration Layer)
MatchController
MatchService
EditorService
DeckService
infrastructure (I/O & Adapters Layer)
PersistenceAdapter (JSON / Serializable)
ArenaRepository DeckRepository
AssetLoader
domain (Core Model & Rules Layer)
Arena Bridge Cell
KingTower CrownTower
Card (Troop, Building, Spell)
Entity Movement Targeting Damage
ElixirManager Timer
AIController AIStrategy
support (Cross-Cutting Utilities)
Logging Math Result

1.3 Dependency Rules

- Flow is one-directional: ui → application → domain.
- The UI never talks directly to domain objects.
- Persistence and asset handling are isolated in infrastructure.
- domain contains pure logic with no I/O or GUI dependencies.
- support offers optional utilities used across layers.

1.4 Package Responsibilities

Package	Responsibility
ui	Presentation layer: screens and HUD elements that interact with the player and display state updates. Implements the Observer pattern for real-time feedback.
application	Coordinates use cases (e.g., startNewMatch, deployCard). Contains controllers and services that validate input and invoke domain operations.
domain	Core business logic: arena layout, entities, combat, elixir, and AI strategies. Independent of UI and storage.
infrastructure	Persistence and external I/O (e.g., saving arena and deck data, loading assets). Implements adapters to decouple format changes.
support	Shared utilities (logging, math helpers, result wrappers). Cross-cutting concerns only.

1.5 Model–View Separation

- The UI observes changes via events from application.
- Domain objects never import UI classes.
- Persistence and rendering are intermediated by application services and adapters.

1.6 Use-Case Mapping (Examples)

- **Start New Match:** MatchController → MatchService.startNewMatch() → load arena & deck from repositories → initialize domain objects (elixir = 5, timer = 3:00).

- **Deploy Card:** MatchController → MatchService.deployCard() → domain validation → spawn entity → UI update.

- **Save/Load Arena or Deck:** EditorService / DeckService → ArenaRepository / DeckRepository → PersistenceAdapter.

1.7 Architectural Rationale

This layered architecture supports modularity, easy testing, and future expansion.

- **Low Coupling:** Each layer depends only on the one below it.

- **High Cohesion:** Each module has a single clear responsibility.

- **Testability:** Domain logic is unit-testable without graphics or I/O.

- **Extensibility:** New cards, arenas, and AI strategies can be added without affecting other layers.

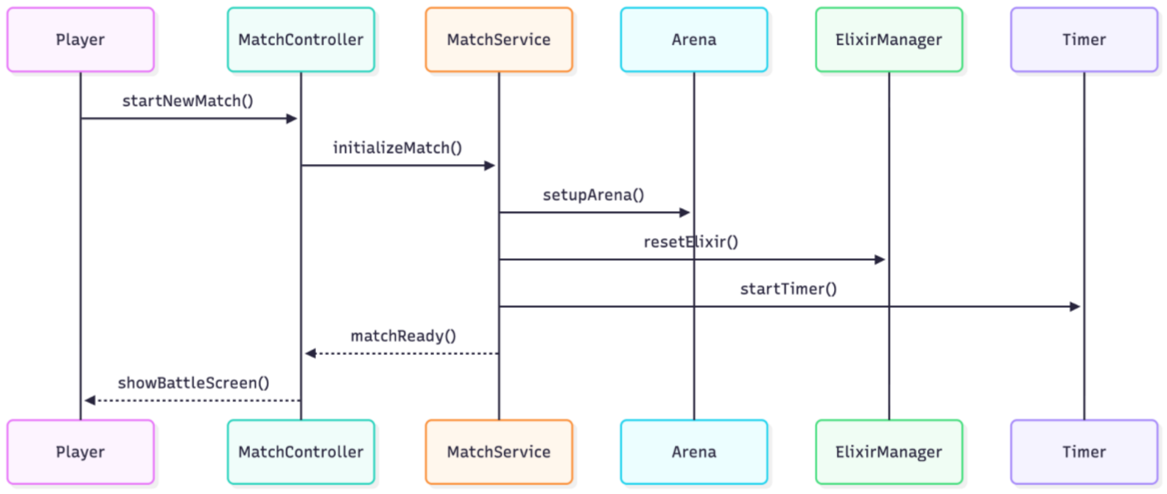
- **Preparation for Phase II:** The design naturally accommodates GoF and GRASP patterns such as Strategy (AI), State (Game Modes), Observer (UI updates), and Factory (Card creation).

2A – Core Gameplay Interaction Diagrams

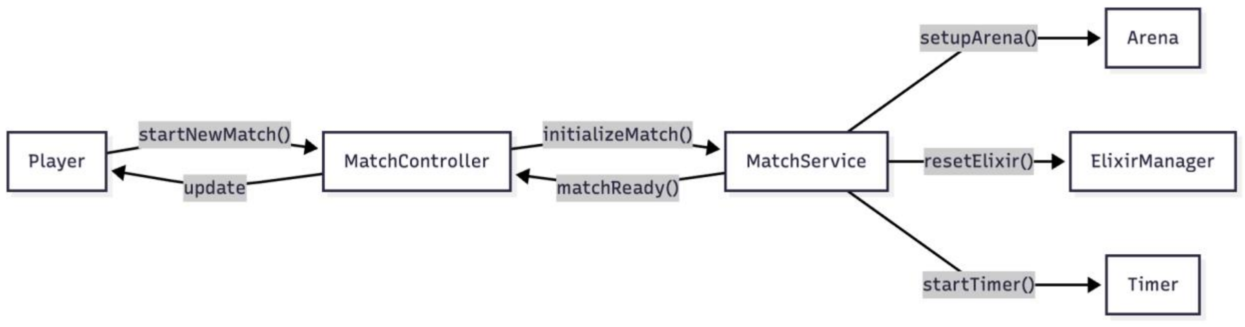
2.1 Start New Match

Purpose: Initializes the game arena, both players, timers, and elixir when a new match begins.

Sequence Diagram:



Communication Diagram:

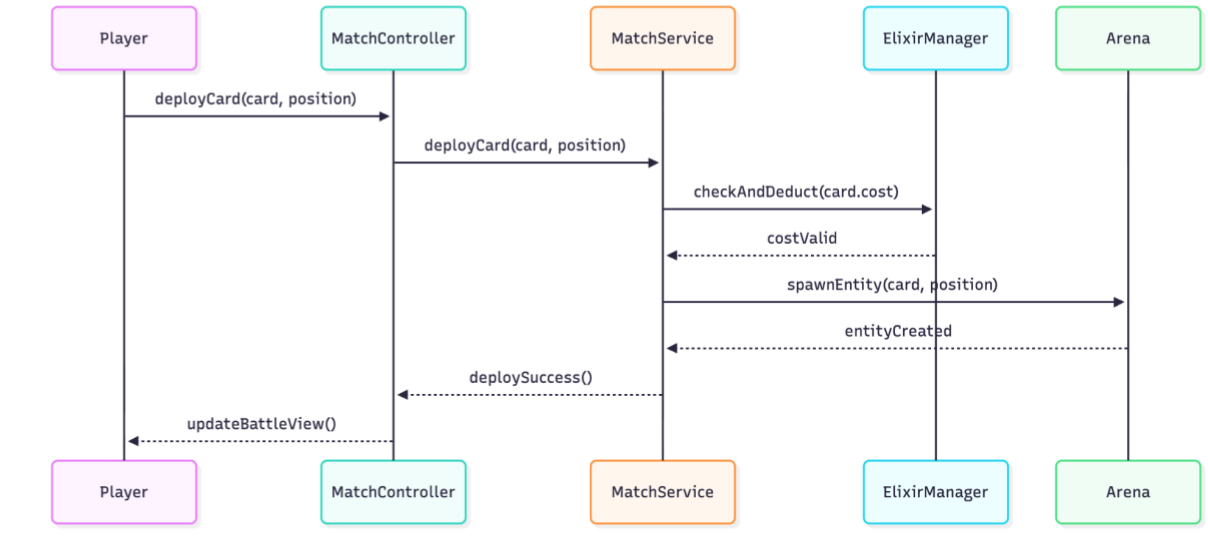


Explanation: The player starts a new match through the controller. The service initializes all core systems and returns control to the UI once the arena is ready. This sequence corresponds to the *Start Match* use case from R1M1.

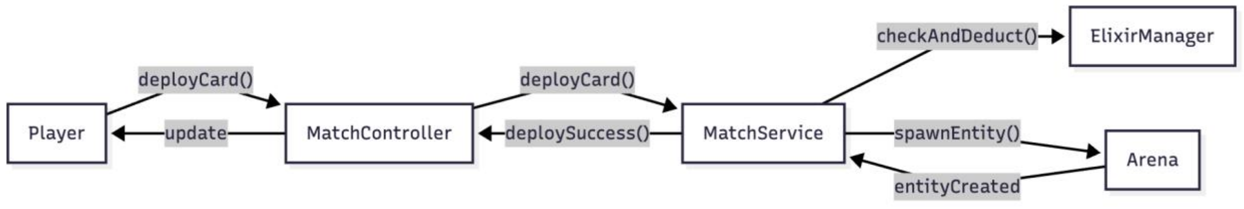
2.2 Deploy Card

Purpose: Handles card deployment validation, elixir deduction, and entity creation on the battlefield.

Sequence Diagram:



Communication Diagram:

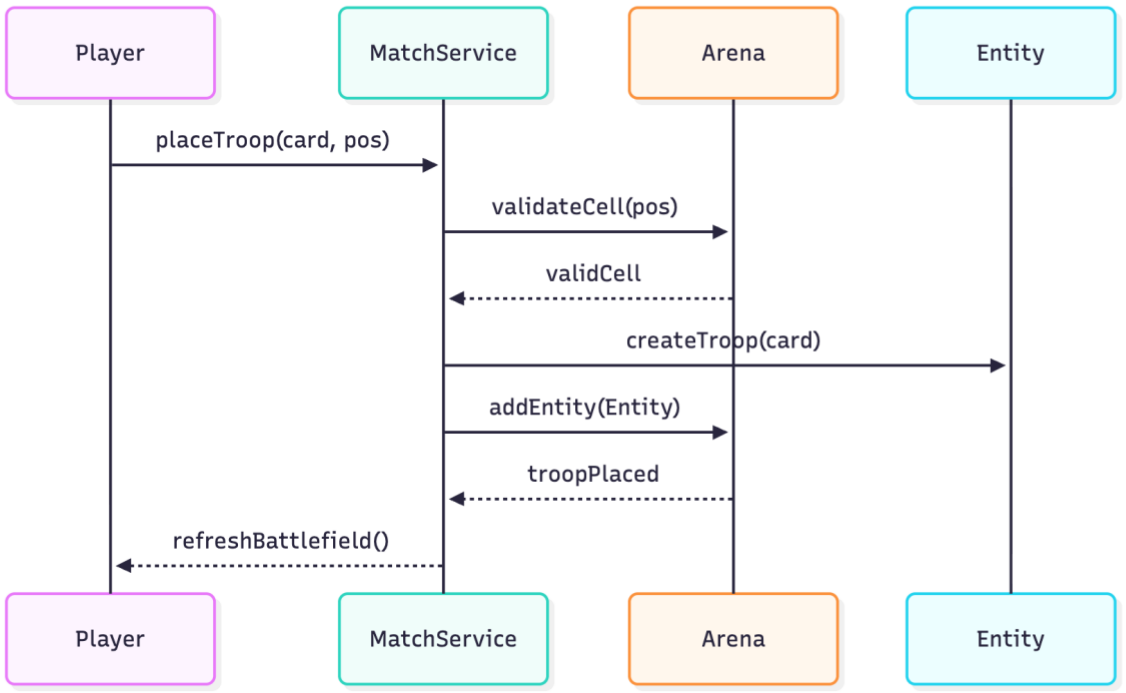


Explanation: The controller passes player input to the service layer. The service validates elixir cost, spawns the entity through the arena, and updates the UI. This represents the *Deploy Card* operation defined in R1M1.

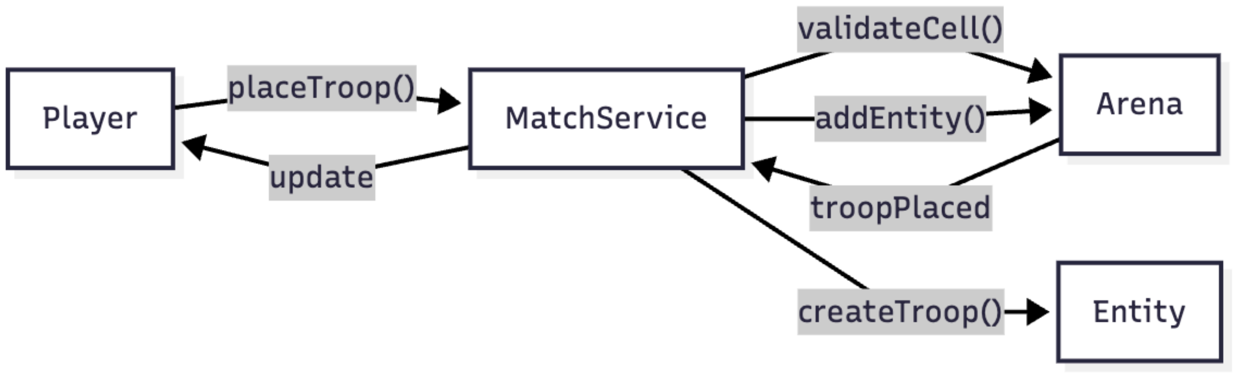
2.3 Place Troop

Purpose: Places a troop entity on a valid cell and starts its movement logic.

Sequence Diagram:



Communication Diagram:

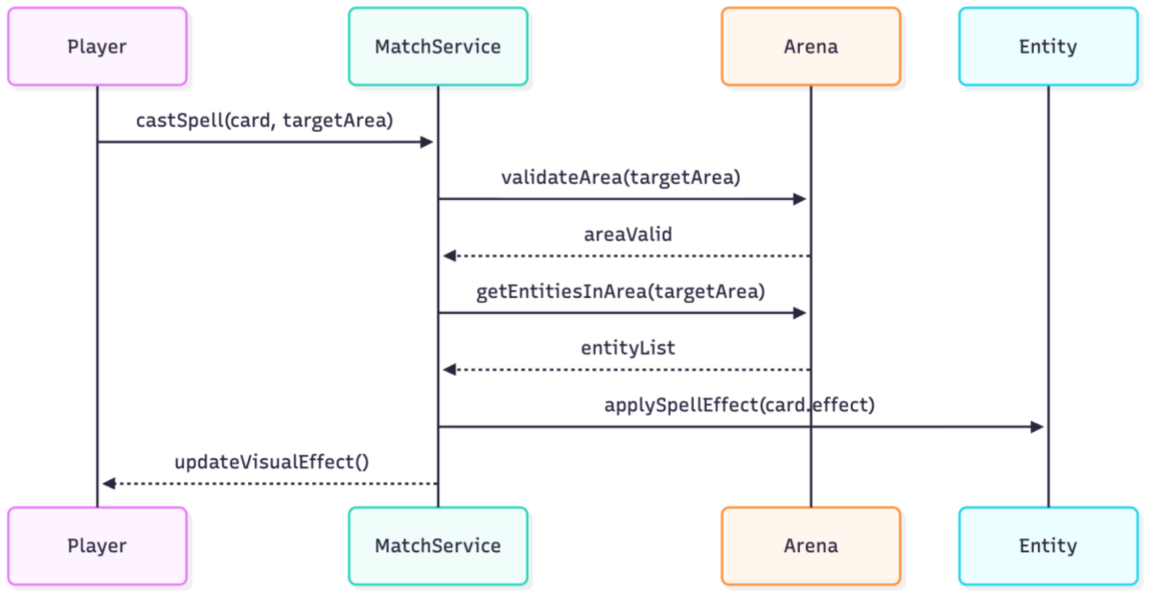


Explanation: After validation, the troop is created and added to the arena grid. Movement logic begins internally, and the UI is refreshed to reflect the new entity.

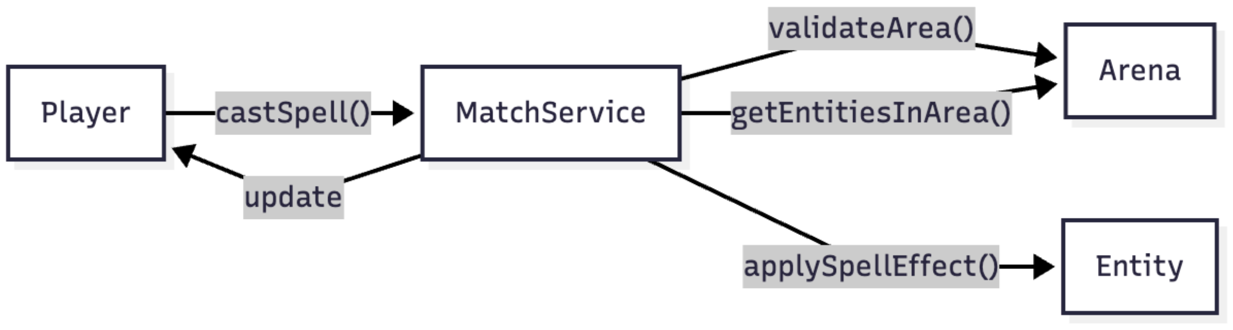
2.4 Cast Spell

Purpose: Casts an area spell, validates the target zone, and applies the effect to affected entities.

Sequence Diagram:



Communication Diagram:

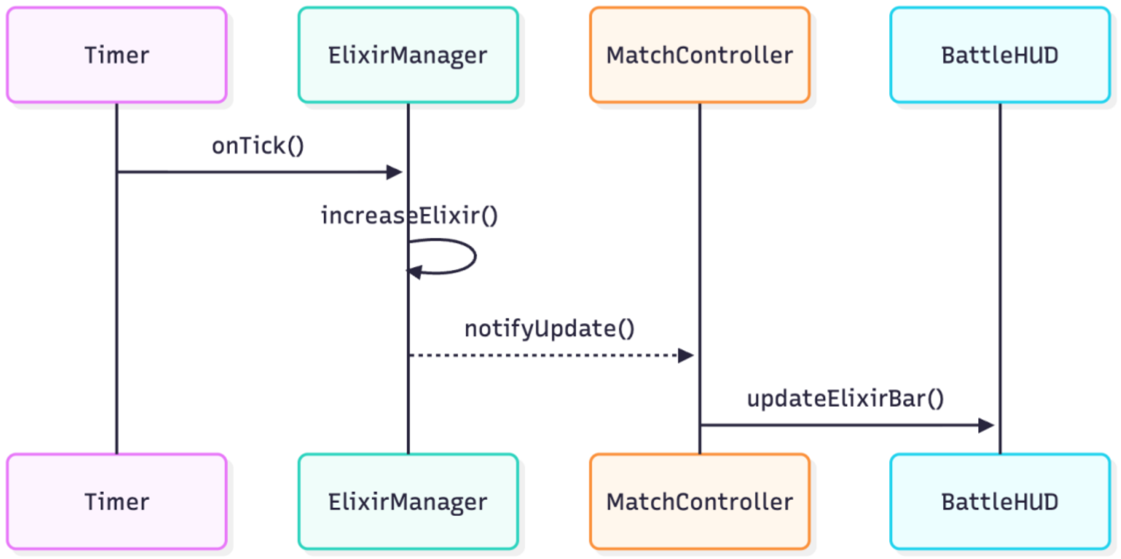


Explanation: The spell is validated for range and applied to all affected entities. The service coordinates with the arena and entity systems to process damage or buffs, then signals the UI for visual updates.

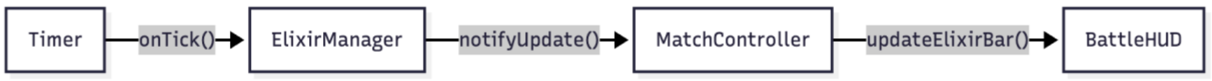
2.5 Elixir Tick & Update

Purpose: Automatically regenerates elixir over time and updates the player's elixir bar on the HUD.

Sequence Diagram:



Communication Diagram:



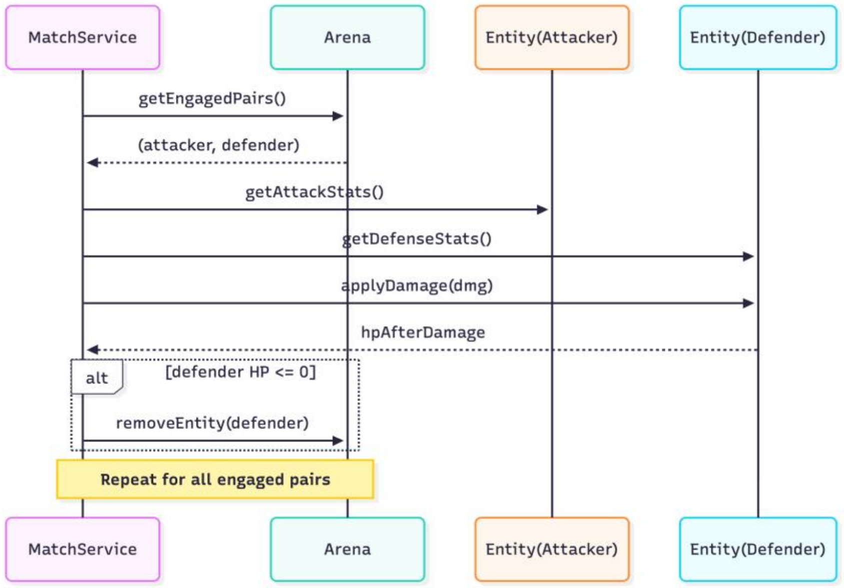
Explanation: Every timer tick triggers an elixir increase. The manager notifies the controller, which updates the HUD display. This demonstrates the Observer pattern that will be formalized in Phase II.

2B – Secondary / System-Level Interaction Diagrams

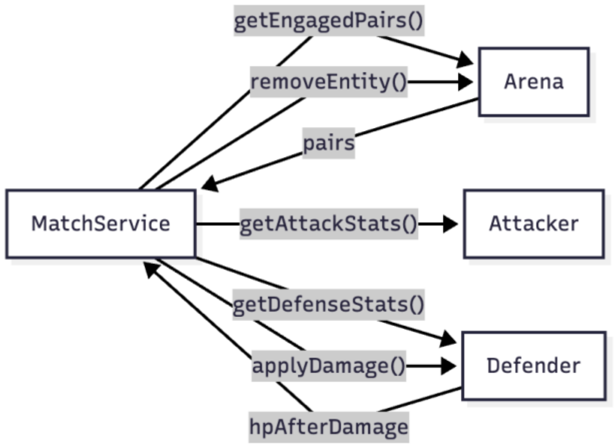
2.6 Resolve Combat

Purpose: Calculates damage exchanges between attacking and defending entities each tick.

Sequence Diagram:



Communication Diagram:

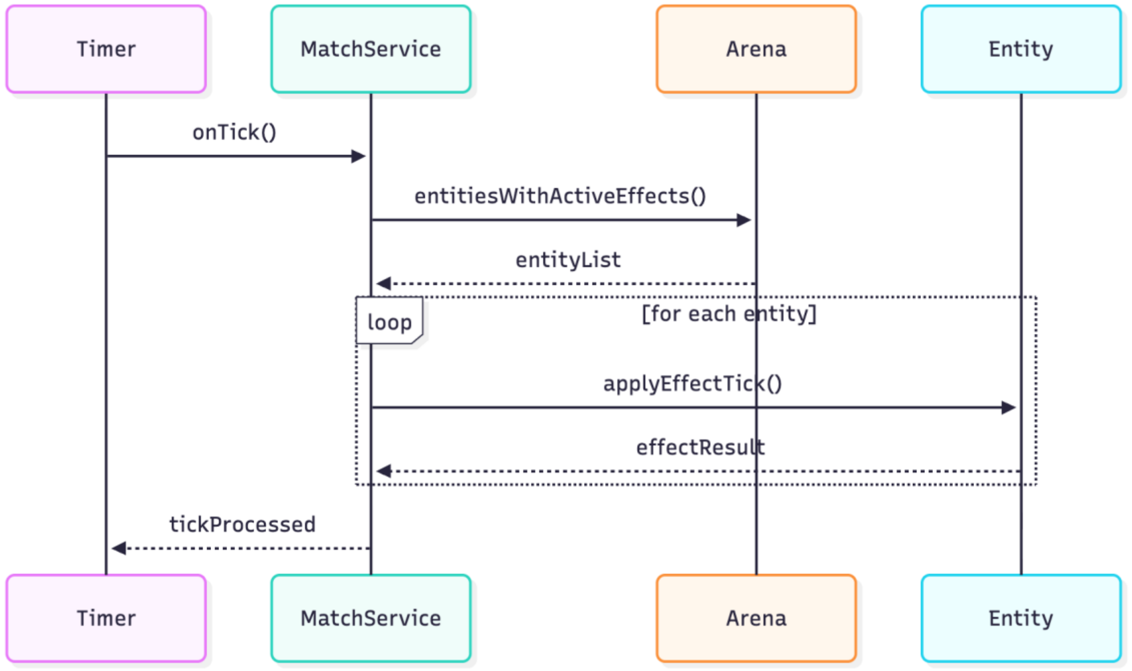


Explanation: When two entities engage, the service retrieves their stats from the domain model, applies attack–defense rules, and updates HP values. The Arena handles removals if HP drops to zero, ensuring consistent state synchronization.

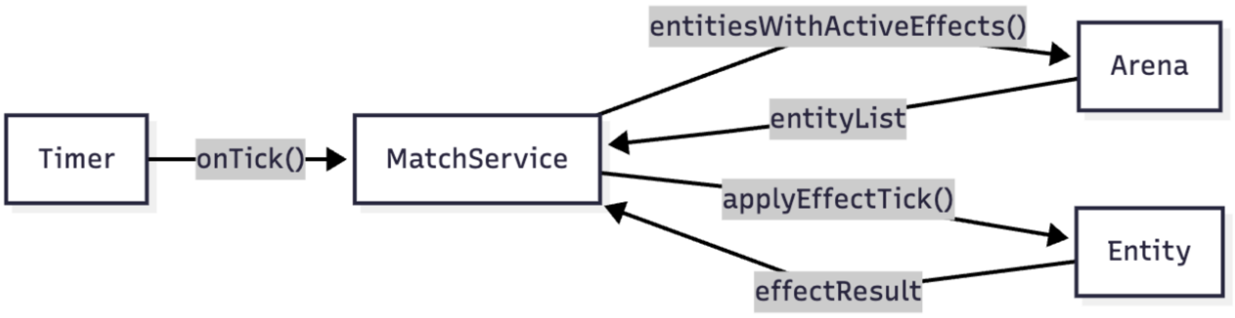
2.7 Apply Spell Effect

Purpose: Applies ongoing or delayed effects (e.g., poison, freeze) to entities within an affected area.

Sequence Diagram:



Communication Diagram:

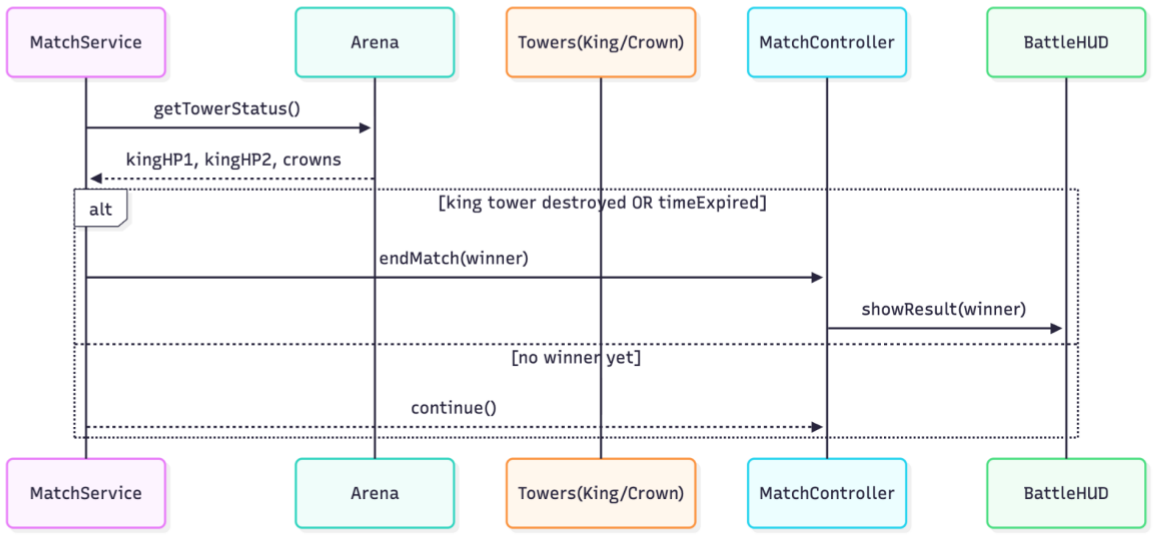


Explanation: The MatchService requests the Arena to locate entities within the spell radius, then calls their update functions to apply the spell's status effect. This captures the logic behind time-based or continuous effects, aligning with the *Apply Spell* use case from R1M1.

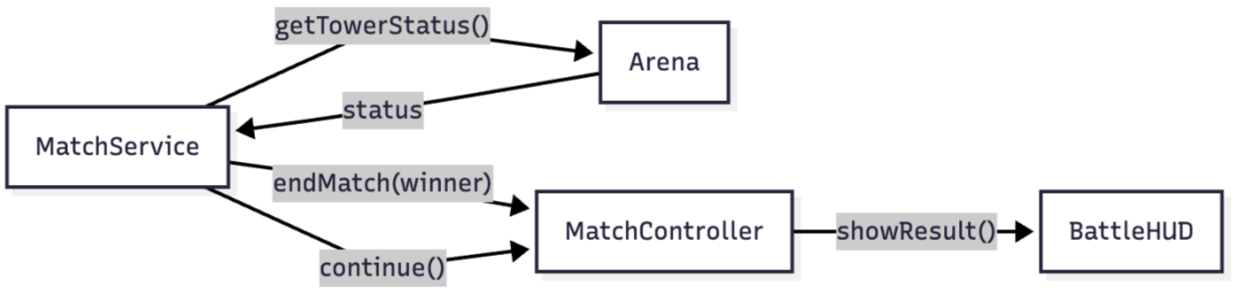
2.8 Check Win Conditions

Purpose: Evaluates whether a player has met victory criteria based on tower destruction or timer expiration.

Sequence Diagram:



Communication Diagram:

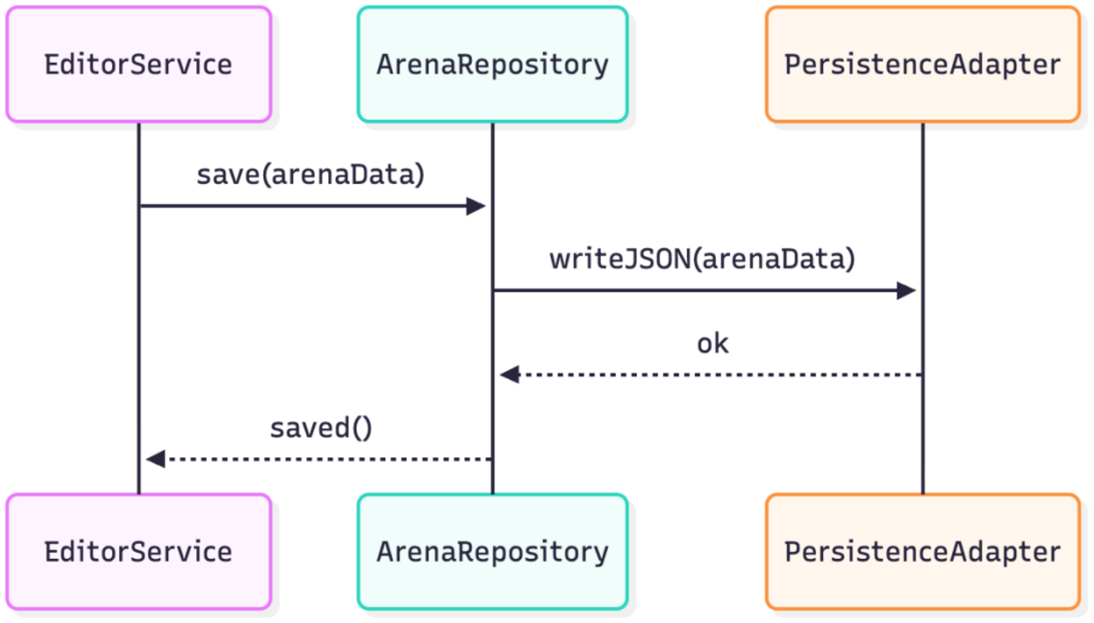


Explanation: After each tick or major event, the MatchService checks the remaining tower HP via the Arena. If one player's king tower is destroyed or time expires, it calls the UI to display the win screen and ends the match session.

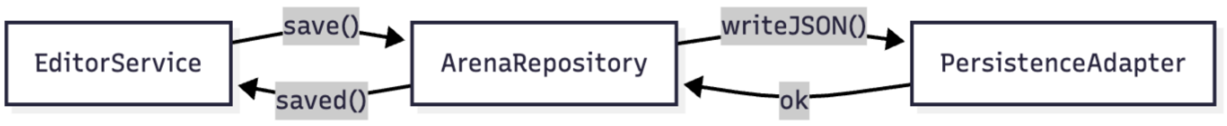
2.9 Save / Load Deck or Arena

Purpose: Saves or retrieves player decks or arena layouts through persistence adapters.

Sequence Diagram:



Communication Diagram:

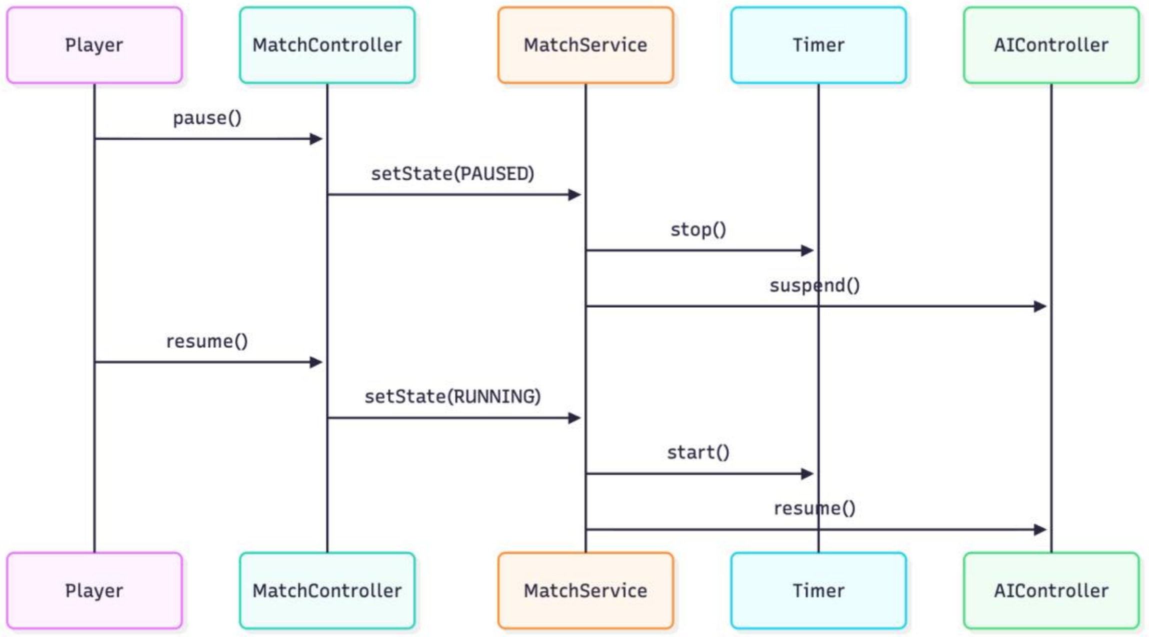


Explanation: The DeckService or EditorService interacts with the ArenaRepository or DeckRepository, which in turn uses the PersistenceAdapter to handle JSON serialization. This ensures game state can be saved and restored across sessions.

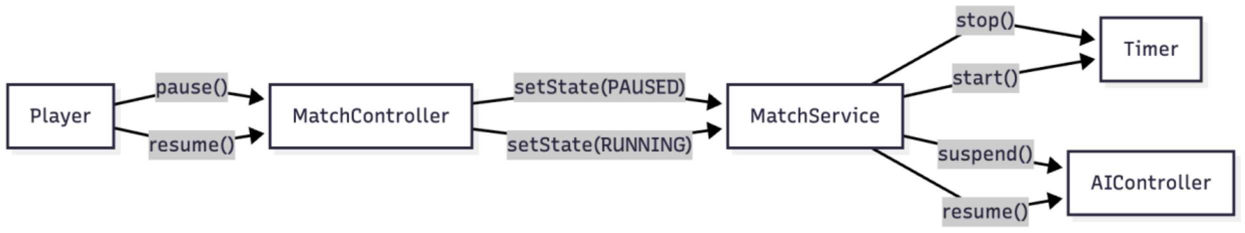
2.10 Pause / Resume Match

Purpose: Temporarily halts or resumes the active match loop, freezing updates and timers.

Sequence Diagram:



Communication Diagram:



Explanation: The player or system issues a pause command; the MatchController sets the game state to "paused," stopping timer and AI updates. Upon resume, all systems reactivate. This ensures consistent state handling and prevents logic errors during pauses.

3. UML Class Diagram

3.1 Purpose

Purpose: Present the main software classes, attributes, operations, and relationships that realize the KU Royale design.
This design follows the logical architecture (UI → Application → Domain → Infrastructure) and the GRASP principles of low coupling and high cohesion.

3.2 UML Class Diagram

UML Class Diagram:

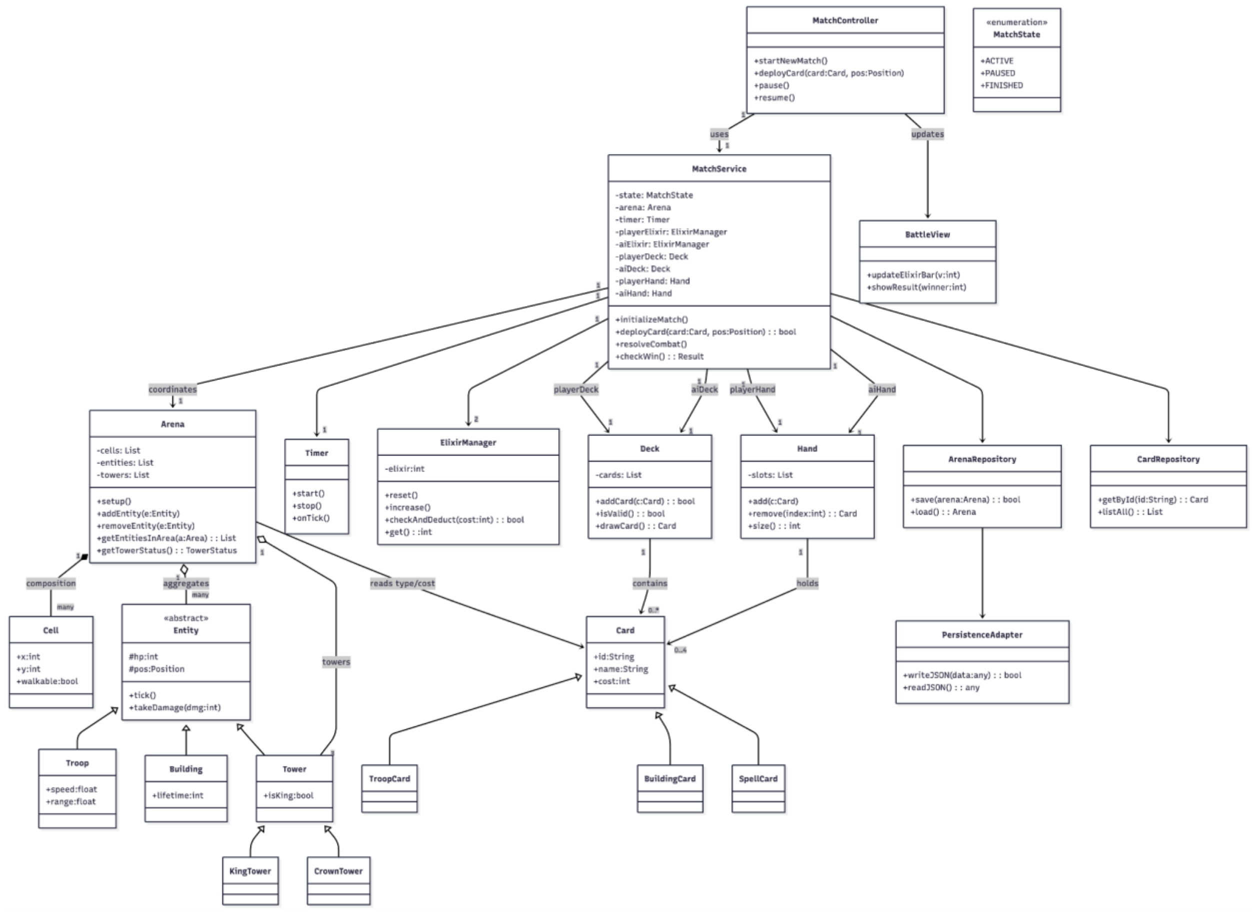


Figure 4.0 — UML Class Diagram for KU Royale (Design I)

3.3 Explanation

The diagram illustrates the layered structure of the KU Royale system:

- **UI Layer:** The BattleView class handles player-facing elements, including displaying match outcomes and elixir updates.
- **Application Layer:** The MatchController class manages player actions and delegates control to MatchService, which coordinates the match lifecycle.
- **Domain Layer:**
  - Arena composes Cell objects and aggregates Entity instances.
  - Entity is abstract and specialized into Troop, Building, and Tower (including KingTower and CrownTower).
  - ElixirManager manages resource regeneration, while Timer controls match timing.
  - Card represents deployable elements with subclasses TroopCard, BuildingCard, and SpellCard.
  - Deck and Hand organize cards for play.
- **Infrastructure Layer:** ArenaRepository and CardRepository handle data persistence through the PersistenceAdapter.

3.4 Consistency with R1M1

The operations within the diagram correspond directly to the R1M1 use cases and SSDs:

- startNewMatch, deployCard, resolveCombat, and checkWin are implemented in MatchService.
- Deck and Hand classes handle deck and card management for UC-02 and UC-04.
- ArenaRepository supports arena saving and loading for UC-01.

3.5 Notation

The diagram uses standard UML notation:

- **Inheritance:** Triangle arrow for generalization (e.g., Entity → Troop, Building, Tower).
- **Composition:** Solid diamond (e.g., Arena → Cell).
- **Aggregation:** Hollow diamond (e.g., Arena → Entity, Tower).
- **Multiplicity:**
    - Arena (1) contains many Cell objects.
    - Deck (1) includes multiple Card objects.
    - Hand (1) holds up to four Card objects.
    - MatchService (1) uses one Timer and two ElixirManager instances.

4. Design Alternatives Discussion

4.1 Alternative A — Monolithic Architecture

Description:
A single unified structure where all components — user interface, application logic, and domain rules — reside in one codebase or controller without separation into layers.

Pros:

- Simple and fast to prototype.
- Fewer classes and files to maintain during initial development.
- Minimal setup overhead for small teams.

Cons:

- Tight coupling between UI and logic makes future changes risky.
- Difficult to test and extend as the system grows.
- Poor scalability and maintainability in larger projects.

4.2 Alternative B — Layered MVC Architecture (Chosen)

Description:
Implements a Model–View–Controller separation aligned with the logical architecture (UI → Application → Domain → Infrastructure). Controllers delegate actions to services, which interact with the domain layer and repositories.

Pros:

- High cohesion and low coupling across modules.
- Clear separation of concerns enables easier debugging and testing.
- Supports GRASP and GoF design principles such as Strategy, Factory, and Observer.
- Simplifies collaboration by dividing responsibilities among layers.

Cons:

- Slightly higher initial development complexity.
- Requires consistent coordination between layers and service boundaries.

4.3 Alternative C — Event-Driven / Observer-Based Architecture

Description:
Relies on asynchronous event dispatchers and observer patterns to manage updates between systems. Each component listens for and responds to events instead of direct function calls.

Pros:

- Very high decoupling between modules.
- Enables smooth asynchronous behavior suitable for real-time game updates.
- Easier to add new components reacting to events without altering existing code.

Cons:

- Complex debugging due to distributed event handling.
- Higher implementation overhead compared to the layered approach.
- Can be excessive for Phase I scope of the project.

4.4 Comparison & Justification

Three alternative designs were evaluated: Monolithic, Layered MVC, and Event-Driven.
The monolithic design was considered too rigid and unscalable, while the event-driven model offered unnecessary complexity for this phase.
The **Layered MVC architecture** was ultimately selected for its balance of clarity, modularity, and extensibility.
It naturally supports Model–View separation and provides a strong foundation for Phase II improvements using GoF and GRASP patterns.

Among these alternatives, only the Layered MVC design was selected for implementation in Design I, while the others were analyzed conceptually to support the evaluation.