NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4240 - SOFTWARE ARCHITECTURE

Fish Miner - Architecture

Quality Attributes

Primary: Modifiability

Secondary: Usability and Performance

Chosen COTS

LibGDX
Scene2d
Ashley ECS

Group 9

First Name	Last Name	Email
August Damm	Lindbæk	august.d.lindbak@stud.ntnu.no
Eivind Biti	Holmen	eivinbho@stud.ntnu.no
Emil	Lunde-Bakke	emillu@stud.ntnu.no
Jesper	Seip	jespese@stud.ntnu.no
Salehe	Mortazavi	salehem@stud.ntnu.no
Tale Marie Wille	Stormark	tale.m.w.stormark@stud.ntnu.no

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the second part of the project documentation, which is the software architectural requirements and tactics of our fishing game. The game will be an extension of the classic arcade game Gold Miner into a fishing variant, which is explained in a greater depth in our requirements documentation. The main architectural quality attributes for this project will be modifiability, usability and performance.

2 Architectural Drivers / Architecturally Significant Requirements (ASRs)

Architectural drivers are essential for building the software architecture of our Android game. To define an architecture for a 2D fishing game that supports modular entity creation, scalable level progression, and maintainable game logic, we identified a set of architectural drivers. These drivers include functional requirements, quality attributes, and business goals that shape our design decisions.

2.1 Functional requirements

The functional requirements specify the features and behavior of the game. Essentially, they indicate what the system must accomplish to satisfy user needs and provide a clear framework for development (Len Bass 2022). While the requirements document provides a comprehensive list of functional requirements, this section outlines the primary ones that influence the architectural design.

- **Core Fishing Mechanic**
  Players must be able to control a hook to catch different types of fish with varying behaviors. This requires a real-time input system, collision detection, and entity interactions based on state transitions.
- **Online Leaderboard Integration**
  Players must be able to submit their scores and view rankings in a global leaderboard. This requires a client-server communication model and integration with a real-time backend.
- **Persistent Scoring System**
  The game must track the player's score based on gameplay actions and retain the highest score across sessions. This requires structured score tracking and a mechanism for persistent storage.
- **Upgradeable Items**
  Players should be able to upgrade equipment such as hooks, sinkers, and reels. This requires an extensible system for managing item variations and player progress.
- **Multi-Level Progression**
  The game should feature levels of increasing difficulty with configurable goals and spawn conditions. This requires a flexible level configuration system and centralized level state management.
- **Screen Navigation and UI Transitions**
  Players should be able to navigate between different screens (e.g., Menu, Play, Leaderboard, Level Lost). This requires a centralized mechanism for screen lifecycle management and UI flow coordination.
- **Pause and Resume Functionality**
  Players must be able to pause and resume the game at any point during gameplay. This requires runtime state control and the ability to suspend and resume game logic cleanly.

2.2 Quality attributes

The architecture of the fishing game is designed to satisfy specific quality attributes that are critical to achieving our project goals. These attributes influence both architectural decisions and implementation strategies. According to Bass et al., quality attributes are non-functional requirements that describe system properties and are essential for evaluating the suitability of an architecture (Len Bass 2022).

2.2.1 Modifiability

Modifiability is a primary quality attribute for our game. The architecture should facilitate easy updates, such as adding new fish types, maps, hooks, and power-ups. It should also support adjustments to game parameters like fishing speed, difficulty levels, and rewards. Bass et al. emphasize that modifiability allows a system to accommodate changes with minimal impact on existing components, which is achieved through techniques like encapsulation, abstraction, and the use of well-defined interfaces (Len Bass 2022).

2.2.2 Usability

Usability ensures that the game is intuitive and user-friendly. This involves designing a clear and responsive user interface (UI) with simple controls. Incorporating a visual tutorial and providing immediate visual feedback for player actions are strategies to enhance the user experience. According to Bass et al., usability encompasses aspects like learnability, operability, and user error protection, all of which contribute to a positive user experience (Len Bass 2022).

2.2.3 Performance

Performance refers to a system's ability to meet timing requirements such as responsiveness and efficient resource usage (Len Bass 2022). In real-time applications like games, this means ensuring the system reacts quickly to user input and maintains a consistent frame rate. According to Bass et al., performance can be supported through architectural tactics like concurrency, resource management, and limiting expensive operations during runtime (Len Bass 2022).

In the context of our fishing game, performance is critical due to its interactive and time-sensitive gameplay. The game should respond smoothly to player actions. Features such as score updates and game interactions should occur without noticeable delay to preserve a fluid experience. To support performance, the architecture should include strategies that reduce runtime overhead and ensure efficient use of resources. This may involve object pooling to avoid frequent memory allocation, asynchronous data handling to prevent blocking the game loop, and a rendering system optimized for real-time graphics. Planning for performance at the architectural level will help ensure that the final game delivers a responsive and enjoyable experience.

2.3 Business goals/requirements

In Software Architecture in Practice, Bass et al. emphasize that architecture is deeply shaped by an organization's business goals. These goals serve as the foundation for many architectural decisions and determine which quality attributes are prioritized in a system's design (Len Bass 2022). For our project, several business goals guided our architectural thinking:

- **Time to Market**: The project is developed within a single semester, making fast delivery essential. The architecture must therefore support parallel development, modular design, and reuse, allowing the team to work efficiently and independently across different game features.

- **Cost and Budget Constraints**: With only six students working on this as one of four parallel courses, development time and resources are limited. The architecture must favor simplicity, maintainability, and technologies that minimize setup and operational complexity.
- **Product Line and Reuse**: The architecture should support future updates and variations, such as new fish types, levels, or mechanics, without significant rework. This aligns with the project description, which emphasizes modifiability, and encourages reusable structures like entity factories and configurable level generation.
- **Market Differentiation**: The game aims to offer engaging gameplay and competitive features such as an online leaderboard. These goals inform design decisions that prioritize responsiveness, game feel, and support for dynamic, real-time feedback.
- **Operational Efficiency**: The game must run smoothly on both Android and desktop platforms. This requires architectural strategies that optimize rendering performance, resource usage, and runtime efficiency across devices.
- **Adaptability and Growth**: Long-term adaptability is important to support future content, updates, and game extensions. Modifiability is key here, enabling the addition of new mechanics and features without requiring architectural changes.

The architecture must be designed to enable these goals, not just accommodate them (Len Bass 2022). For example, meeting the time-to-market goal involves supporting modular development, while goals related to reuse and adaptability influence the use of flexible creation patterns and separation of concerns.

3 Stakeholders and Concerns

The following table, Table 1, is an overview of the relevant stakeholders and their concerns regarding our software. Each concern is mapped to a quality attribute or relevant actions to ensure that architectural decisions align with the stakeholders needs. By linking concerns to specific quality attributes and architectural views, we can prioritize design choices that enhance the most critical aspects of the game. According to the ISO/IEC/IEEE 42010 standard, an architecture must address the concerns of its stakeholders by organizing relevant information into views that respond to those concerns (ISO/IEC/IEEE 2022).

Stakeholder	Concerns
Player (End-user)	**Modifiability:** New content such as upgrades, fish, and levels should be added regularly without affecting existing gameplay. **Usability:** Clear and visually appealing interface with easy-to-learn controls. A short tutorial should be available if needed. The leaderboard should be fair and competitive, and gameplay should support multiple viable strategies. **Performance:** Gameplay should appear smooth, with responsive controls and minimal delays between actions and feedback.
Development team	**Modifiability:** The architecture should support easy addition of new features through modularity and reusable components, while enabling parallel development. **Usability:** Code structure should be consistent and readable, with clear interfaces and debug-friendly systems to ease implementation and testing. **Performance:** Systems should be efficient and easy to profile, with low runtime overhead and minimal memory churn.
Course supervisors	**Modifiability:** The architecture should demonstrate support for future changes using patterns and tactics that enable extension and reuse. **Usability:** Code and documentation should be understandable and well-organized to support assessment and maintainability. **Performance:** The system should reflect thoughtful trade-offs and justify performance-related design decisions.
ATAM evaluators	**Modifiability:** Architectural decisions should support growth and change with minimal impact on unrelated components. **Usability:** Architecture views and documentation should clearly express design intent and rationale. **Performance:** Performance scenarios should be defined, and the architecture should demonstrate readiness to meet timing and responsiveness goals.

Table 1: Stakeholder Concerns

4 Architectural Viewpoints

This section presents the architectural views of the FishMiner game based on the 4+1 View Model by Kruchten (Kruchten 1995). Each view addresses specific concerns of different stakeholders and supports reasoning about modifiability, usability, and performance.

View	Purpose in the Project	Stakeholders	Notation
Logical	Defines the main components, classes, and interactions such as the 'GameContext', 'World', and ECS systems. Helps the team understand structure and responsibilities, and supports modifiability.	Developers, course supervisors, end users, ATAM evaluators	UML Class Diagram
Process	Captures the dynamic behavior of the game, including communication between UI, game logic, and systems (event dispatching via 'GameEventBus'). Aids in understanding game flow, lifecycle, and event-based interactions.	Developers, course supervisors, ATAM evaluators	UML Activity Diagram and Sequence Diagram
Development	Provides a high-level overview of how the codebase is structured into modules (e.g., domain, UI, data, factories). Supports maintainability and enables effective team collaboration by clarifying boundaries.	Developers, course supervisors, ATAM evaluators	Development Diagram
Physical	Describes how the game is deployed, e.g., on Android devices using LibGDX. Useful for understanding platform constraints and ensuring performance.	Developers, course supervisors, ATAM evaluators	UML Deployment Diagram

Table 2: Architectural Views and Their Importance for the FishMiner Project

The fifth view in Kruchten's 4+1 model is the Use Case View, or Scenarios View as it is referred to in Software Architecture in Practice (Len Bass 2022), which captures typical interactions between users and the system to validate the architecture. These use cases (or scenarios) illustrate how the system meets its functional and quality requirements (Kruchten 1995). In this project, the purpose of the Use Case View is to demonstrate how the architecture supports key gameplay flows, developer workflows, and runtime expectations such as performance and responsiveness.

5 Architectural Tactics

The following are the architectural tactics we will employ to meet our main quality requirements: modifiability, usability and performance.

5.1 Modifiability

The goal of modifiability tactics is to optimize module size, cohesion, coupling, and binding time to make a system more adaptable to change (Bass et al. 2021). To accommodate changes like new upgrades / power-ups and new levels, our main focus will be on increasing cohesion and reducing coupling.

5.1.1 Increasing Cohesion

Cohesion measures how strongly related and focused the responsibilities of a module are. High cohesion simplifies maintenance, reduces redundancy, and enhances readability (Len Bass 2022). To increase cohesion, we apply the following tactics:

- **Split Module:** We reduce future modification costs by splitting large, complex systems into smaller, focused ones. For example, gameplay logic is separated into multiple systems such as `HookSystem`, `FishingSystem`, and `ScoreSystem` to isolate behavior and responsibilities.
- **Redistribute Responsibilities:** Each module should have a single, well-defined responsibility. For example, fish-catching logic is handled primarily in `FishingSystem`, while score updates are centralized in `ScoreSystem`, rather than being duplicated across other systems or screens.
- **Refactor regularly:** As features evolve, restructuring code to improve cohesion ensures the system remains manageable.

5.1.2 Reducing Coupling

Reducing coupling ensures that modules can be modified with minimal impact on the rest of the system (Len Bass 2022). We will achieve this through the following tactics:

- **Encapsulate:** Limit direct access to internal module details, exposing only what is necessary through predefined interfaces. An example is `RequestManager`, which exposes simple public methods to post leaderboard and auth requests while hiding the details of Firebase interaction.
- **Abstract Common Services:** Extract shared functionalities into independent modules or services to promote reusability and minimize dependencies. Common responsibilities—such as logging, configuration access, and Firebase communication—are encapsulated in dedicated modules (`Logger`, `Configuration`, `RequestManager`). These services are implemented as singletons to ensure reuse and avoid redundant instantiations.
- **Restrict Dependencies:** We avoid unnecessary dependencies between unrelated modules by enforcing separation between layers (UI, domain, and infrastructure). Systems interact only through well-defined events and components, rather than accessing other systems directly.
- **Use an Intermediary:** The `GameEventBus` acts as a mediator between publishers and subscribers, allowing systems to communicate through events rather than direct method calls. This decouples systems like `CollisionSystem` and `ScoreSystem`, improving modularity and extensibility.

5.1.3 Documentation and Coding Conventions

The design phase is ineffective if group members struggle to recall the patterns and tactics agreed upon. Ideally, everyone should understand the inputs and outputs of all modules. Therefore, clear documentation and well-defined standards are essential to the architecture, especially for ensuring modifiability (Bass et al. 2021).

Standards:

- Every module should have documented inputs and outputs.
- Functions longer than three lines must include comments explaining their purpose.

- Use `camelCase` for variable names.
- Boolean variables should be prefixed with `is` or `has` (e.g., `isPaused`, `hasDied`).

Git:

- Use the Kanban board actively
- Imagine "This commit will..." before writing a commit message.
- Follow trunk-based development with a branch per issue.

5.2 Usability

The system should behave in a way the user expects. To accomplish this, we will employ the following tactics:

- **Pause/Resume:** The player can pause the game and resume it later without losing progress. This improves control over gameplay sessions and aligns with user expectations for mobile games.
- **Aggregate:** We will display the user score in a summarized format between levels and at the end of each session. This helps players understand their achievements and learn from feedback.
- **Maintain User Model:** The system assumes the player is unfamiliar and includes a short visual tutorial to guide onboarding if needed. In addition we will use familiar icons and symbols to make the system more intuitive.
- **Maintain Task Model:** The game provides clear feedback for each user action, ensuring players always understand the outcome of their actions. For example, the value/score for each fish is visualized on capture.
- **Maintain System Model:** We aim for consistency in UI behavior across screens, helping users build mental models of how the game responds. As well as using interaction patterns from other systems.

5.3 Performance

Performance is a critical quality attribute for FishMiner, as the game must remain smooth and responsive across a range of devices, particularly on mobile platforms. To support this attribute, we apply several tactics from the "Control Resource Demand" and "Manage Resources" categories (Len Bass 2022):

5.3.1 Control Resource Demand

- **Manage Work Requests:** Rather than spawning fish continuously during gameplay, the system spawns a predefined set of fish at the beginning of each level. This controls the number of active entities and ensures a predictable load on systems like spawning, collision detection, and rendering.
- **Reduce Computational Overhead:** We use Ashley's `PooledEngine` to reuse entities and components instead of constantly allocating and garbage-collecting them. This reduces CPU load and memory churn, especially during gameplay where fish and hook entities are frequently created and removed.

- **Increase Efficiency:** Fish movement and collisions are computed using lightweight math operations and bounding boxes rather than expensive pixel-perfect checks. Systems are structured to only operate on relevant entities (via Ashley's Family filtering), avoiding unnecessary computation.

5.3.2 Manage Resources

- **Introduce Concurrency:** Although the game loop itself is single-threaded (as typical for LibGDX), non-critical operations like leaderboard score submissions are handled asynchronously via Firebase callbacks. This prevents network delays from blocking gameplay responsiveness.
- **Schedule Resources:** Critical assets such as textures and animations are loaded during initialization before gameplay begins, ensuring they are available when needed and not loaded on demand mid-session. Less critical assets load on demand (eg. the tutorial), reducing the memory overhead.

6 Design and Architectural Patterns & Architectural Rationale

6.1 Architectural Patterns

We will use four different architecture patterns in our fishing game:

- **Layered Architecture** - Separates UI, game logic, and data.
- **Entity Component System (ECS)** - Decouples game objects from logic, improving flexibility.
- **Event-driven Architecture** - Components communicate and coordinate by producing and responding to events.
- **Client-Server Architecture** - Enables the game to communicate with Firebase services for authentication and leaderboard storage and retrieval.

6.1.1 Layered Pattern

To ensure a modifiable and reusable code base, the software are structured in a way that allows modules to be developed and evolved independently. This separation of concerns is achieved through the layered architecture pattern, which organizes the software into distinct layers. Each layer groups related modules that together provide a cohesive set of services (Len Bass 2022). In Fish Miner, the software is structured into three distinct layers. The presentation layer manages the user interface and visual elements. The business logic layer contains the core gameplay mechanics and rules. Finally, the data layer, positioned at the base, is responsible for storing and retrieving data such as high scores and player statistics.

6.1.2 Entity-Component-System

Entity-Component-System (ECS) is a software architecture pattern that emphasizes composition over inheritance, aiming to provide a flexible and scalable approach for managing entities in large-scale, real-time applications. The core idea is to decouple objects into separate data and logic layers by organizing them into three distinct parts: entities, components, and systems (Härkönen 2019). This data-driven structure enhances modularity by enabling new game mechanics or interaction modalities to be integrated with minimal impact on existing code (Muratet and Garbarini 2020).

In our fishing game, each game object, such as the fish, hook, sinker, or boat, is represented as an `Entity`, a lightweight container with no behavior. Behavior and data are instead separated into `Components`, such as `AttachmentComponent`, `VelocityComponent` or `WeightComponent`, which store only the relevant data for each entity. The actual game logic is implemented in `Systems`, like `MovementSystem`, `CollisionSystem`, or `AnimationSystem`, which operate on entities that possess specific sets of components. This separation of data and behavior makes it easy to extend and modify game functionality. For example, new entity types or behaviors can be introduced simply by defining new components and systems, with minimal modification of existing code. This flexible structure improves the modifiability and scalability of our game. A detailed overview of the entities, components, and systems can be found in Appendix **??**.

6.1.3 Event-driven Architecture

FishMiner uses an event-driven architecture implemented via a publish-subscribe pattern through the custom `GameEventBus`. Event-driven architecture (EDA) is a software design paradigm in which components communicate and coordinate by producing and responding to events, allowing for highly decoupled, scalable, and reactive systems (Microsoft Learn 2023a). This promotes loose coupling and modularity between gameplay logic, UI, and system interactions. We discuss this further in Section 6.2.4.

6.1.4 Client-Server Architecture

At last, we also wanted to use a Client-Server architecture, which is a distributed application structure that partitions tasks between service requesters and providers of a resource (GeeksforGeeks 2025). This is used to divide the application into two main components: the client (the game on the player's devices) and the server (storing highscores and other shared data). An example from FishMiner could be that the client sends highscore data to the server and receives updates, and the server stores highscores and sends leaderboard information back to the client. We deliberately chose a lightweight implementation using Firebase instead of building a custom backend, as this allowed us to reduce complexity and focus our efforts on developing the core gameplay, while still retaining the key advantages of the client-server model.

6.2 Design Patterns

The design patterns that we want to use are:

- **Factory Method** – Handles flexible creation of game entities.
- **Singleton** - Ensures single instances for shared resources and game management.
- **State** - Manages different states of the game and its entities efficiently.
- **Publish-Subscribe** - Allows event-driven interactions.

6.2.1 The Factory Method

In the FishMiner game, we apply the Factory Method pattern to encapsulate the creation logic of complex game entities such as hooks, fish, and player characters. According to lecture material, the Factory Method is suitable in situations where "a class cannot anticipate the class of objects it must create" or "a class wants its subclasses to specify the objects it creates" (Sørensen 2025). This applies directly to our game, where systems can request entities without needing to know their specific configuration. For example, `HookFactory.createEntity(...)` constructs a hook entity using a `HookTypes` enum, and `PlayerFactory` coordinates the creation of an entire player with all relevant attachments. This approach enhances modifiability, as new fish, hooks, or reels can be

introduced by updating enum values and factory logic, without impacting the systems that consume them. It also supports performance by using Ashley's `PooledEngine` to reuse components and avoid costly object allocation during runtime. Additionally, it improves usability from a development standpoint, as developers interact with clean, centralized methods for entity creation rather than managing components manually. These benefits align with software engineering recommendations, including Microsoft's guidance on using factory methods to promote testability and modular design (Microsoft Learn 2023b).

6.2.2 Singleton Pattern

A second creational design pattern is the Singleton, which ensures that a class has only one instance and provides a global access point to it. According to the course material, this pattern is appropriate when "a class represents a concept that requires a single instance," and must ensure that it "is accessible from a well-known access point" while preventing other instances from being created (Sørensen 2025). The solution, as described, is to make the class itself responsible for tracking its sole instance through a static method such as `getInstance()`. In FishMiner, we apply the Singleton pattern to several key infrastructure classes: `Logger`, `Configuration`, `Assets`, `ScreenManager`, `MusicManager`, and `GameEventBus`. Each of these classes encapsulates global or shared behavior that must remain consistent throughout the game's lifecycle. For example, `Logger` manages a single log file to avoid conflicting writes; `Configuration` maintains display and audio settings; and `ScreenManager` controls screen transitions and state caching.

The Singleton pattern supports the quality attribute of modifiability by centralizing configuration, logging, and asset access, thereby reducing the need for boilerplate or duplicated state management. It also contributes to usability from a development perspective, since developers do not need to pass these instances around manually, they are always accessible via their static accessors. While Singleton is not inherently a performance optimization, it can contribute to performance when applied to components that are costly to initialize, such as audio managers, asset loaders, or persistent loggers. However, as noted in both the class material and external sources, care must be taken to avoid misuse, as Singletons can introduce hidden dependencies and complicate unit testing if state is not carefully managed (Microsoft Learn 2023c).

6.2.3 State Pattern

The State design pattern is a behavioral pattern that allows an object to alter its behavior when its internal state changes, appearing to change its class. This pattern is ideal when "an object must change its behavior at runtime depending on its state" and helps avoid large conditional statements or duplicated logic (RefactoringGuru n.d.). It promotes the encapsulation of state-specific behavior into separate classes or enums, enabling cleaner and more modular code. In FishMiner, the State pattern is used extensively to manage dynamic entity behavior across gameplay systems. For example, the `World` class maintains a `WorldState` enum to control global game flow (e.g., `RUNNING`, `PAUSED`, `WON`, `LOST`). Similarly, entities like hooks and fish implement the `IState` interface and transition between concrete states such as `SWINGING`, `REELING`, `CAPTURED`, or `ATTACKING`. These transitions trigger visual updates and gameplay behavior via systems like `FishingSystem` and `HookSystem`, which use `StateComponent<T>` to determine an entity's current state and act accordingly.

The use of the State pattern enhances modifiability, as it allows for new behaviors or transitions to be added simply by introducing new states. It also improves usability for developers, as each state's behavior is encapsulated and avoids tangled conditional logic. Additionally, this pattern contributes to readability and maintainability by expressing game logic in terms of domain-specific states, making the code-base easier to reason about and extend (Gamma et al. 1995).

6.2.4 Publish-Subscribe Pattern

The Publish-Subscribe pattern is a messaging architecture used to facilitate event-driven interactions between components in a system (Saafan 2024). In this pattern, components communicate asynchronously through events or topics, enabling a clear separation between event producers (publishers) and event consumers (subscribers). Publishers emit messages without knowledge of which components will receive them, while subscribers simply register interest in specific event types. The event bus acts as an intermediary, managing subscriptions and delivering messages to all interested subscribers at runtime. This implicit invocation mechanism results in loose coupling, as publishers and subscribers operate independently. A key benefit of this pattern is its flexibility and adaptability—new behavior can be introduced by adding subscribers without altering the publisher logic (Len Bass 2022).

In our game, this pattern is realized through a custom `GameEventBus`. For example, when a collision is detected between a hook and a fish in the `CollisionSystem`, a `FishHitEvent` is published. The `FishingSystem`, which subscribes to this event, receives it asynchronously and handles the logic for attaching the fish, updating its state to `HOOKED`, and eventually transitioning it to `CAPTURED` when reeling is complete. This decouples the collision detection logic from the state management and game progression logic, improving modifiability and testability. By centralizing communication through `GameEventBus`, the architecture allows new systems to react to events without requiring changes to the original sender. While not a low-level optimization, the Publish-Subscribe pattern contributes to performance by promoting decoupled, targeted interactions between systems. This reduces redundant computation and keeps each system focused on its core logic.

7 Architectural Views

This section presents the software architecture of FishMiner using the 4+1 View Model, as introduced by Philippe Kruchten (Kruchten 1995). The 4+1 model provides a framework for describing the system from multiple perspectives, each addressing different stakeholder concerns and quality attributes. These views include the Logical View, Process View, Development View, Physical View, and Use Case View (Scenarios).

7.1 Logical View

The Logical View describes the system's static structure, focusing on how functionality is organized across modules and how responsibilities are divided among classes and components. This view is especially relevant to developers and evaluators who want to understand how the system is built and how it supports key quality attributes like modifiability, performance, and usability (Kruchten 1995). Following the guidance of Kruchten's 4+1 View Model, the Logical View in FishMiner reflects the main architectural styles applied in the system:

- **Layered Architecture**: Separating concerns between the UI layer (`PlayScreen`, `MenuScreen`), domain layer (`FishingSystem`, `GameContext`), and data layer (Firebase services). This structure supports the Encapsulate and Restrict Dependencies tactics by clearly defining boundaries and limiting what each layer can access. A overview over the architecture is visualized in Figure 2 and a more detailed diagram in Figure 7.
- **Entity-Component-System (ECS)**: Using Ashley to model game objects as entities composed of components (`HookComponent`, `StateComponent`) that are processed by systems (`HookSystem`, `CollisionSystem`). A simplified overview is shown in Figure 1, a detailed diagram for entities in Figure 4, for components in Figure 5, and for systems in Figure 6. Split Module and Redistribute Responsibilities tactics, allowing each system to focus on a single aspect of gameplay while maintaining modularity and cohesion.
- **Event-Driven Architecture**: Implemented via a custom `GameEventBus`, enabling decoupled communication between systems through events like `FishHitEvent` and `ScoreEvent`.

As shown in the process view, Section 7.2, the `GameEventBus` supports the use of the "Use an Intermediary" tactic by decoupling event producers and consumers. This supports modifiability and scalability, especially in gameplay interactions.

- **Client-ServerArchitecture**: The game client connects to Firebase services for backend functionality. The database schema is represented in Figure 3. Firebase Authentication handles user registration and login, while the Firestore database consists of a single LeaderBoard collection. Each document within this collection represents a user and stores the following fields: score (integer) representing the user's highest score in the game, and a username/email (string). The authenticated user's email is used as a unique identifier (document ID) for each document in the LeaderBoard collection. When a user submits a new score, the client checks the existing score in the database. If the new score is higher, the document is updated. The client also queries the top 10 scores using a descending sort on the score field to render the leaderboard.

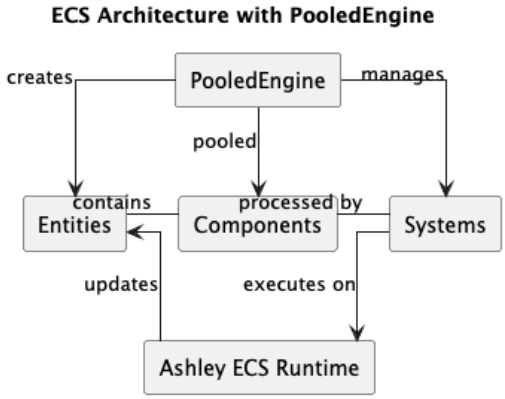

Figure 1: Flow diagram of the Entity-Component-System (ECS) architecture.

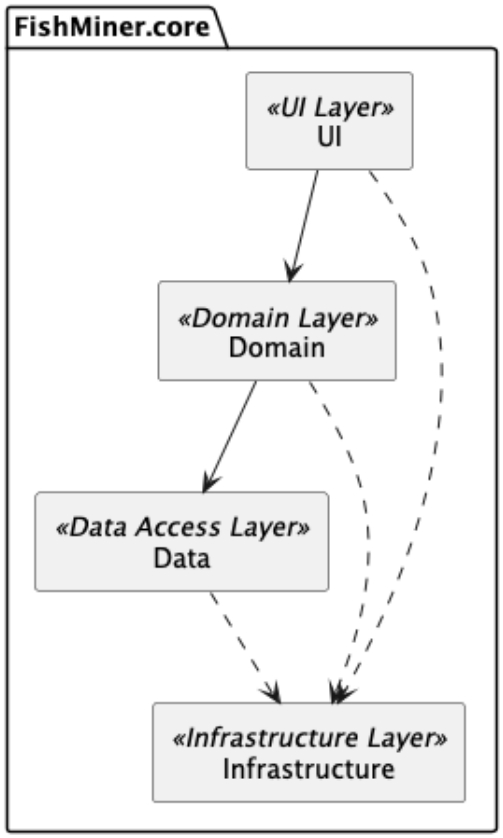

Figure 2: Simplified overview of the Layered Architecture.

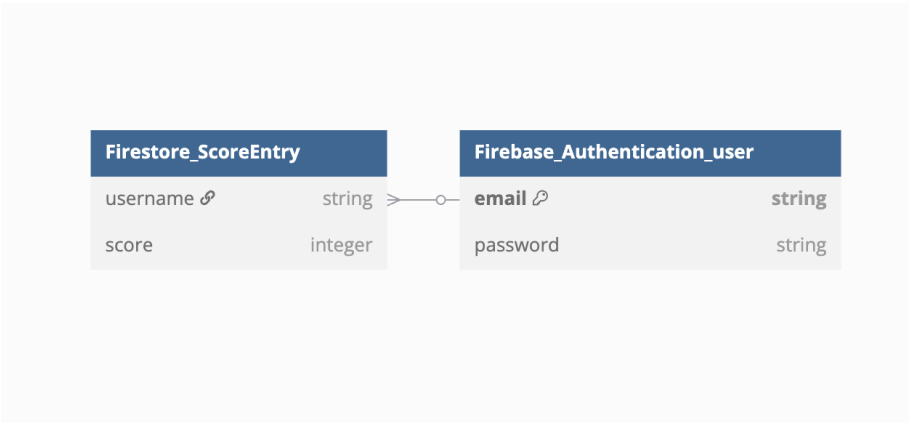

Figure 3: Server side database schema.

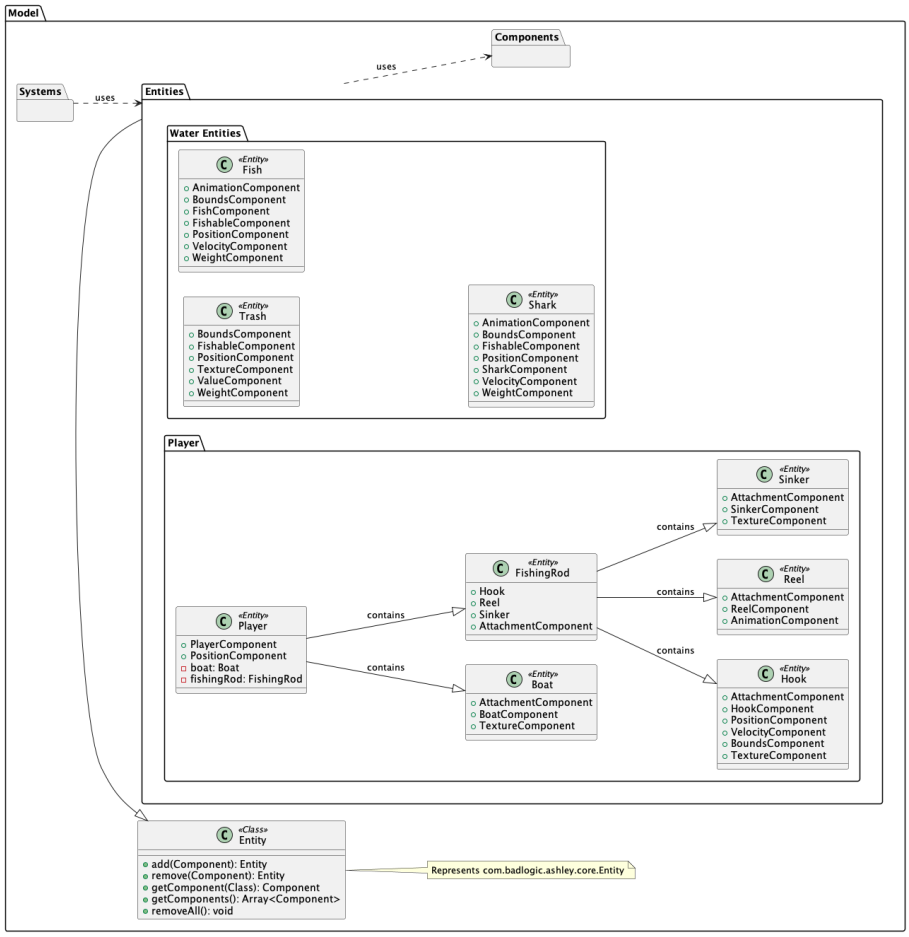

Figure 4: Detailed diagram visualizing the entites and their relations.

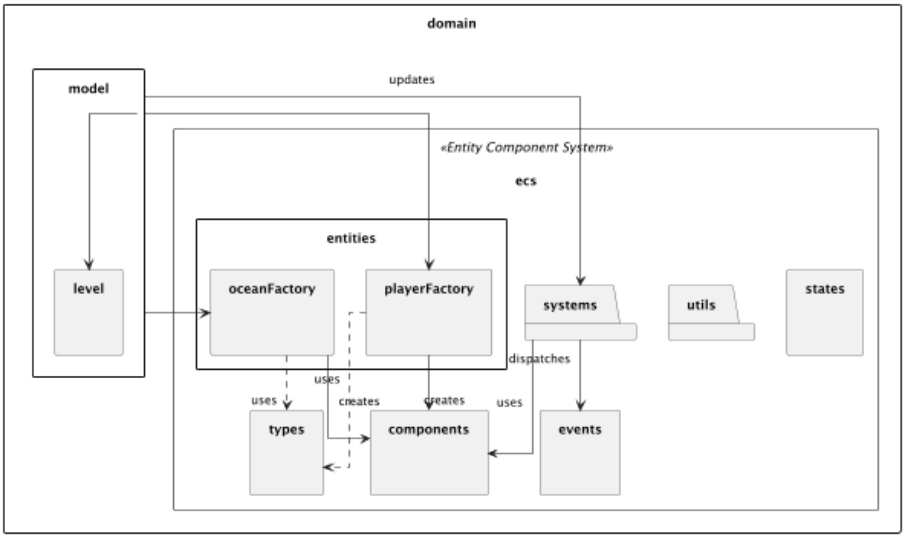

Figure 5: Detailed diagram visualizing the ECS's systems components and their relations internally and with the model package.

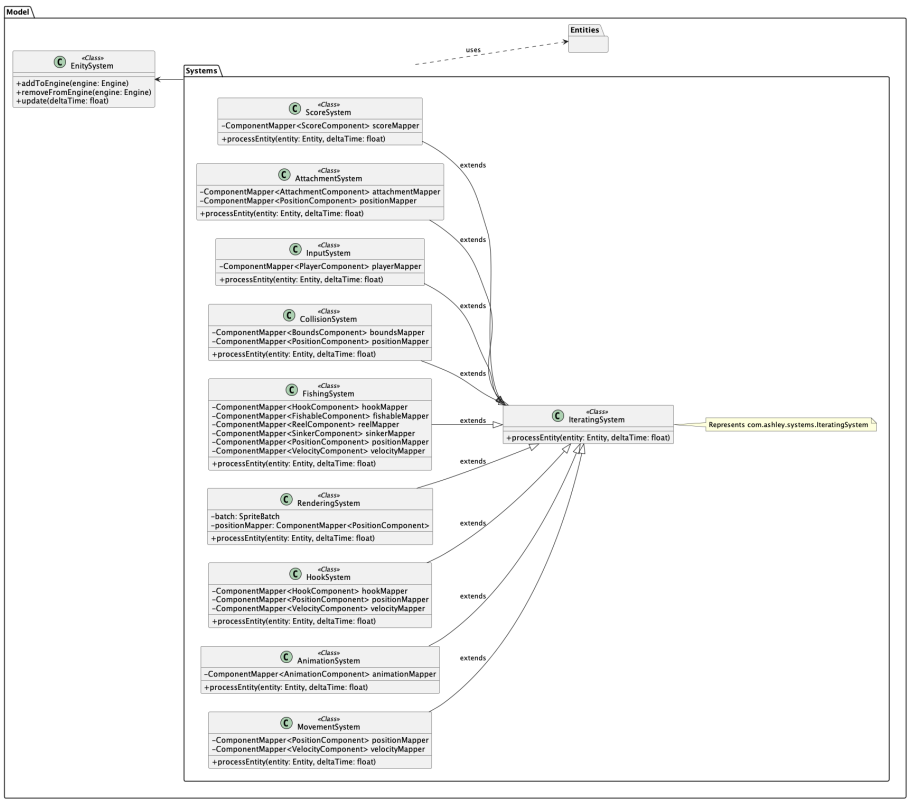

Figure 6: Detailed diagram visualizing the systems and their relations.

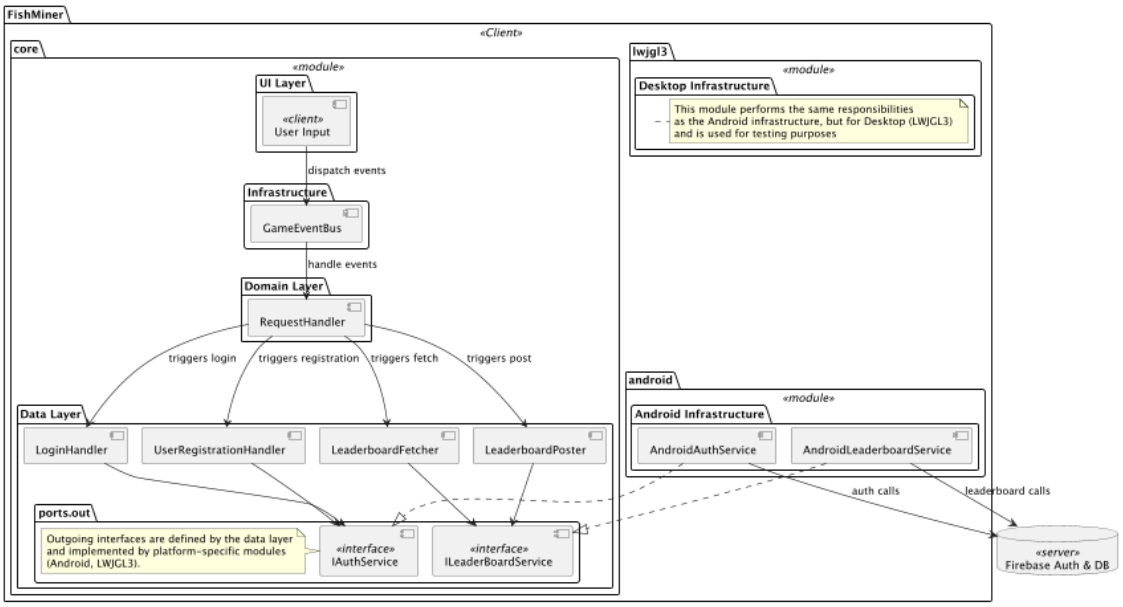

Figure 7: Detailed diagram visualizing the Layered Architecture.

7.2 Process View

The process view describes how the components of a system interact at runtime. It captures the concurrency, synchronization, and user-driven navigation through the game's interface and logic components, following the principles outlined by Kruchten in the 4+1 View Model (Kruchten 1995).
The activity diagram presented in Figure 8 illustrates a high-level overview of the FishMiner gameplay flow, starting from the Menu screen. From here, players can access Play, Tutorial, Settings, Leaderboard, or Login/Register screen. This structure introduces dual-menu states, one for general users and one for logged-in users. A decision point determines whether the player logs in successfully. If so, they are routed to the Logged-in Menu, which includes additional option such as Log Out.

During GamePlay, players can choose to pause the game. Pause View offers options to either **Continue**, resuming gameplay, or **Quit**, which returns the player to the appropriate menu depending on the status of the login. Upon completing the game, the system checks whether the level was won or lost. A win leads to the Upgrade Store, where users can enhance their abilities before moving on to the next level. A loss brings up the Game Over view, where players may either view the **Leaderboard** or return to the **Menu**.

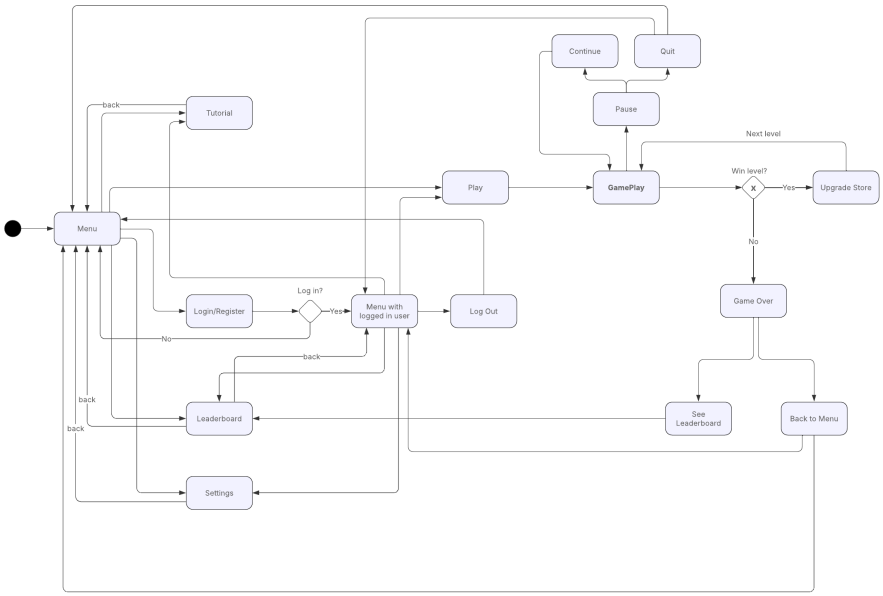

Figure 8: Acitivity diagram of game process.

The three process views below illustrate the interactions between different interfaces and key components in the game, and they show the runtime behavior of screen management, login process and catching a fish during gameplay.

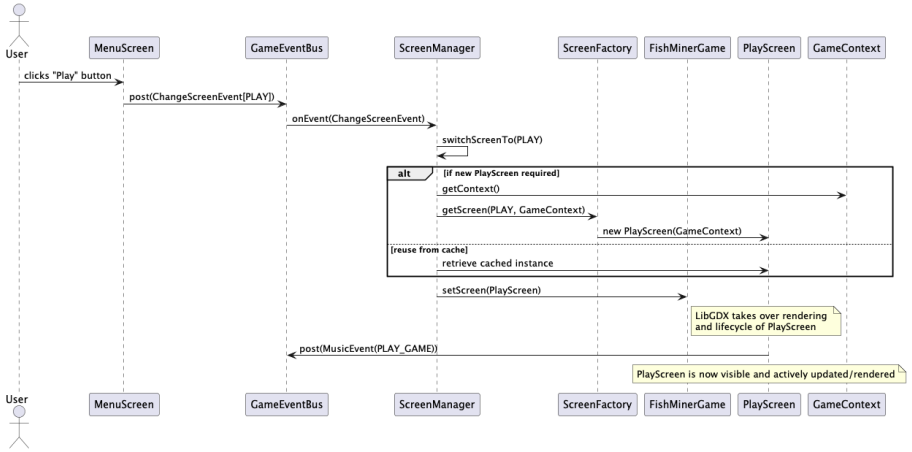

Figure 9: Sequence diagram for screen management.

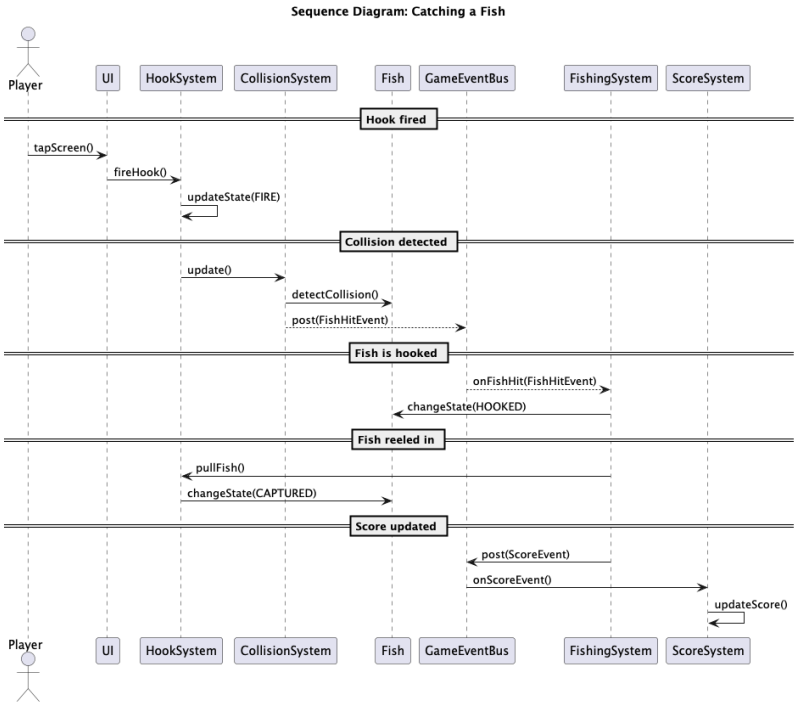

Figure 10: Sequence Diagram for catching a Fish

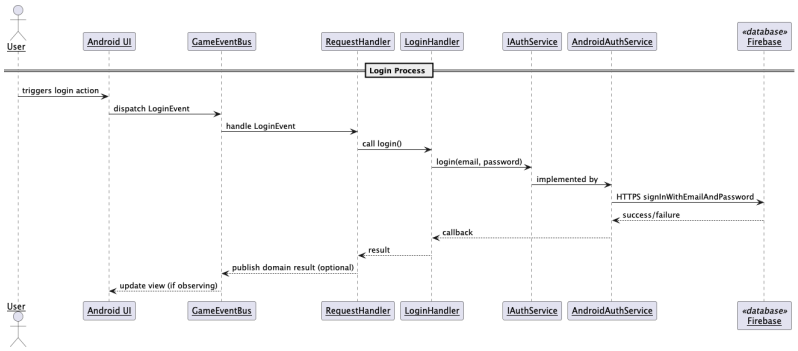

Figure 11: Sequence Diagram of the login process.

7.3 Physical View

The physical structure of the system is represented using a deployment diagram, illustrated in the figure below. This diagram outlines how the application runs on an Android device and details its communication with the server, following the principles of the Client-Server architecture.

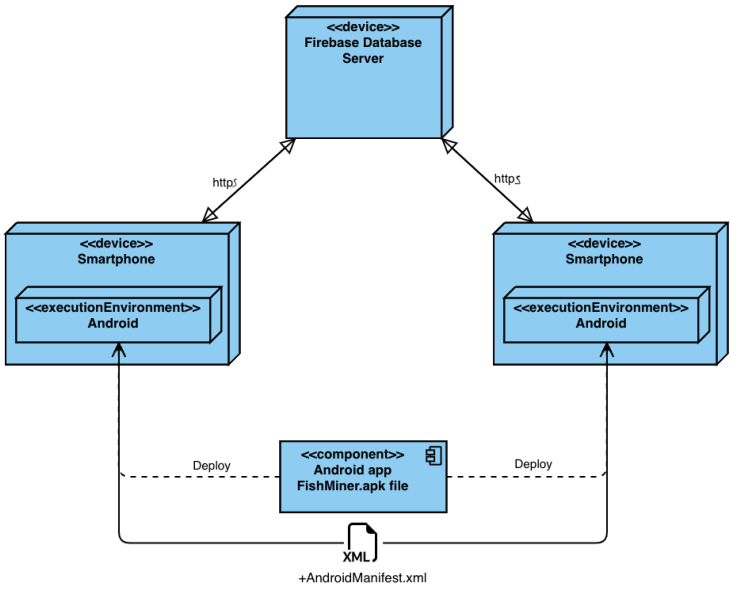

Figure 12: UML Deployment Diagram

The physical architecture consists of the Android game application that runs on user devices and communicates with Firebase over the public internet. All client-server interactions occur via HTTPS (GoogleForDevelopers 2025). Firebase Authentication and Firestore are hosted on Google Cloud as managed services. Firestore acts as the backend database engine and stores high scores in a document-based structure. The app connects via Wi-Fi or mobile data, depending on device configuration.

7.4 Development View

The Development View focuses on how the software is organized in the source code and addresses concerns related to maintainability, modularity, and team collaboration. According to Kruchten's 4+1 View Model (Kruchten 1995), this view describes the system's static software modules and their dependencies, as seen by programmers. In FishMiner this is represented through a package-based structure that separates concerns into logical modules such as domain, ui, data, factories, and firebase. This is visualized in the package diagram in Figure 7. The goal is to divide the work into parts which can be performed in parallel. The diagram in Figure 13 is a simplified view of the different packages and classes of the game organized.

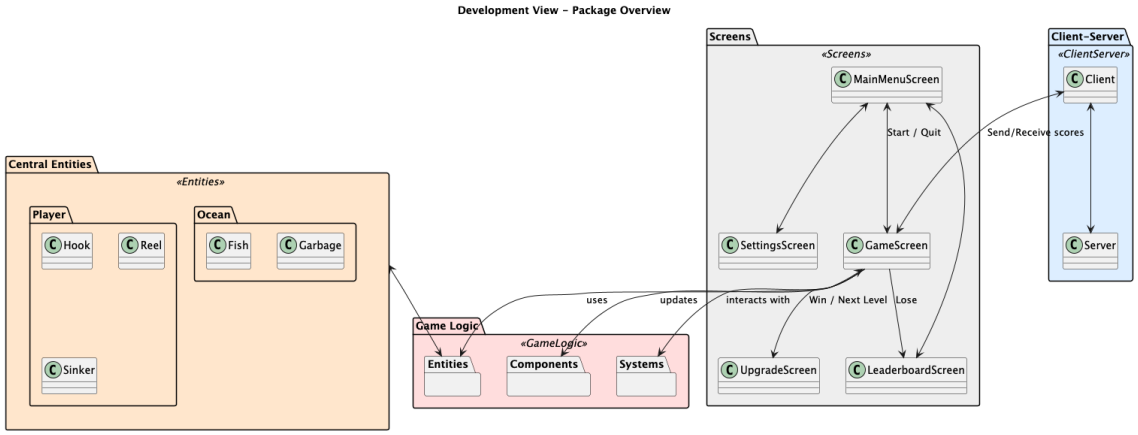

Figure 13: UML package diagram

This package structure supports modular development by grouping related functionality. To support modularity and platform abstraction, we want to define clear interface boundaries. For example, we aim to implement seperate interfaces for the login and leaderboard services, which allows different team members to implement independent logic and backend integration in parallel. The UI components interact only with domain-layer interfaces, enabling testable and decoupled design. The ECS model further supports separation between game logic and rendering. This organization aligns with the logical view and facilitates smooth team collaboration with minimal merge conflicts.

8 Consistency among Architectural Views

We ensured architectural consistency by enforcing clear layer boundaries, using interfaces for all cross-layer communication, and organizing code to reflect the layered structure.

9 Architectural Rationale

Our architectural decisions were primarily guided by the goals of modifiability, usability, and performance. In Section We adopted a layered architecture to ensure a clean separation of concerns between the user interface, domain logic, data access, and platform-specific implementations. This structure allows for easier testing, extension, and maintenance of individual parts of the application, while also supporting parallel development by clearly defining module boundaries.

The client-server architecture underpins communication between the Android game client and Firebase services, which handle authentication and leaderboard storage. This approach reduces the complexity of managing backend infrastructure and provides a scalable and highly available platform for real-time data operations.

Internally, the game follows an event-driven architecture. Gameplay systems are decoupled through the use of a central event bus, allowing them to react to events independently. This promotes modularity and makes it easy to introduce new behavior without modifying existing systems.

Additionally, the game is built on an entity-component-system (ECS) model, which emphasizes data-driven design. Entities are constructed from reusable components and the behavior is encapsulated in systems that operate independently. This supports a high degree of flexibility and reusability in how game mechanics are composed and evolved over time.

The purpose of using patterns is to achieve our requirement goals, thus not over-commiting to all the specifics of each pattern is essential to make them fit our purpose. Some times we have done

this by borrowing partial patterns from other architectures such as Hexagonal Architecture and Model View Controller. However, this partial usage of patterns is not worth mentioning explicitly as they are only used to support our main architectural patterns.

To reinforce these main architectural patterns, we employed tactics such as encapsulation through interface boundaries, indirection via abstractions for platform-dependent logic, and concurrency isolation to ensure that asynchronous Firebase interactions are safely wrapped on the main UI thread. Together, these design choices form a maintainable foundation that aligns with the project's quality goals and collaborative workflow.

10 Issues

One of the biggest challenges for our group has been defining and maintaining a clear scope for the game. With many creative ideas and potential features, it was important to continuously evaluate what to include in order to stay within time and resource constraints.

Another challenge was deciding how strictly to adhere to architectural patterns. We aimed to follow best practices and maintain consistency with the intended architecture, while also avoiding over-engineering. Striking the right balance between architectural purity and practical implementation was an ongoing consideration throughout the project.

11 Changes

ID	Date	Change history	Comment/Reason
1	2025-03-04	Added "performance" as a quality attribute to the architectural documentation.	Performance became increasingly relevant when we released the scope of the game.
2	2025-03-05	Added performance-related tactics to the architecture documentation.	Follows the introduction of "Performance" as a quality attribute, ensuring alignment between design goals and implementation strategies.
3	2025-03-04	Refined the architectural drivers	Refined after feedback from the professor
4	2025-03-10	Replaced MVC with a combination of Layered Architecture and Event-Driven Architecture	The use of ECS made it difficult to apply MVC as a whole. We realized that Layered Architecture was a better fit, especially since we expose domain functionality to the UI through interfaces like `IGameContext`. While we borrowed some structuring ideas from MVC to group related responsibilities, the final design aligns more clearly with established layering principles.
5	2025-03-20	Replaced use of Abstract Factory pattern with Factory Method pattern for entity creation.	The Factory Method pattern better matches our architecture, simplifies implementation, and improves modifiability without requiring multiple factory implementations.
6	2025-04-07	Added a database schema in the Logical View for the Firebase Firestore database and authentication.	To better illustrate the logic on the server-side.
7	2025-04-20	Added diagrams illustrating the ECS components in the Logical View.	To better illustrate the logic of the entity component system.
8	2025-04-20	Created the Development View package diagram.	After feedback on a confusing diagram, we recreated it.
9	2025-04-23	Renamed architecture tactics in the report to match the terminology from the course literature and lectures ("Split Module", "Redistribute Responsibilities").	Ensures consistency with the course syllabus and improves clarity for readers and evaluators.
10	2025-04-24	Added sequence diagrams in the Process View for screen management and login process.	To better illustrate the different processes of the game.
11	2025-04-24	Added a paragraph under the Physical View to explain the network and protocols used between the client and firebase server.	To clarify the different aspects of the Physical View of the Client-Server Architecture.
12	2025-04-24	Added more text explaining the Architecture rationale.	Illustrating the thoughts and motivation behind the different architecture implementations.

Table 3: Change History Table

12 Individual Contributions

Member	Contributions
August	Introduction, Logical View and Physical View. Also contributed on/wrote Process View and Architecture Rationale.
Eivind	Worked on Section 6 (Design and Architectural Patterns), Section 9 (Architectural Rationale), Logical view
Emil	Worked on Section 3(Stakeholders and Concerns), Section 5 (Architectural Tactics), Section 7.1 (Logical overview diagram), Section 7.3 (Development View) and Section 8 (Consistency among architecture views)
Tale	Refined the functional requirements, and wrote about the Architectural and Design Patterns. Reviewed and refined the doocument
Salehe	Worked on section 2 (Architectural Drivers), section 4 (Architectural Viewpoints), and implemented and described the activity diagram in section 7.2.
Jesper	Worked on Section 7 (Logical View), 5 (architectural viewpoints) and Functional Requirements as well as various diagrams

Table 4: Individual Contributions

Bibliography

Bass, Len, Paul Clements and Rick Kazman (2021). *Software Architecture in Practice*. 4th. Addison-Wesley, pp. 160, 143.

Gamma, Erich et al. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GeeksforGeeks (2025). *Client-Server Model*. Accessed: 2025-04-23. URL: https://www.geeksforgeeks.org/client-server-model/.

GoogleForDevelopers (2025). *Privacy and Security in Firebase*. Accessed: 2025-04-24. URL: https://firebase.google.com/support/privacy.

Härkönen, Toni (2019). 'Advantages and Implementation of Entity-Component-Systems'. In: *Tampere University, Tampere, Finland*, p. 6.

ISO/IEC/IEEE (2022). *ISO/IEC/IEEE 42010: Architecture Descriptions*. Accessed: 2025-04-22. URL: http://www.iso-architecture.org/42010/ads/.

Kruchten, Phillippe (1995). 'Architectural Blueprints - The "4+1" View Model of Software Architecture'. In: *IEEE Software*.

Len Bass Paul Clements, Rick Kazman (2022). *Software Architecture in Practice*. Accessed: 24 Feb. 2025. Addison-Wesley.

Microsoft Learn (2023a). *Event-driven architecture style*. Accessed: 2025-04-21. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven.

— (2023b). *Factory Method Design Pattern*. Accessed 2025-04-22. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/factory-method.

— (2023c). *Singleton Design Pattern*. Accessed 2025-04-22. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/singleton.

Muratet, Mathieu and Délia Garbarini (2020). 'Accessibility and serious games: What about Entity-Component-System software architecture?' In: *Games and Learning Alliance: 9th International Conference, GALA 2020, Laval, France, December 9–10, 2020, Proceedings 9*. Springer, pp. 3–12.

RefactoringGuru (n.d.). *State*. Accessed: 24 Feb. 2025. URL: https://refactoring.guru/design-patterns/state.

Saafan, Amr (2024). *Design Pattern: Publisher-Subscriber — Nile Bits*. Accessed: 24 Feb. 2025. Nile Bits. URL: https://www.nilebits.com/blog/2024/07/design-pattern-publisher-subscriber/.

Sørensen, TDT4240 Software Architecture - Carl-Fredrik (2025). *Factory Method Design Pattern Lecture Slide*. TDT4240 - Design Patterns Lecture, accessed 22 April 2025.