NTNU

Kunnskap for en bedre verden

Department of Computer Science

TDT4240 - SOFTWARE ARCHITECTURE

Fish Miner - Implementation

Quality Attributes

Primary: Modifiability

Secondary: Usability, 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

Contents

1 Introduction

1.1 Project description

The following report is for the game development project conducted as part of the course TDT4240 at NTNU. This document describes the design and implementation phase of our Android game Fish Miner. It also contains a user manual, test reports and different challenges we faced while making the game.

1.2 Game Concept Overview

The game we have made is heavily inspired by the game Gold Miner. Gold Miner is a 2d, asynchronous multiplayer game where the goal is to collect enough gold within 60 seconds to proceed to the next level. In the game, a hook swings back and forth and the player has to click the screen to fire the hook towards gold while dodging other game objects. In between levels a shop is presented where gold can be spent to obtain different upgrades.

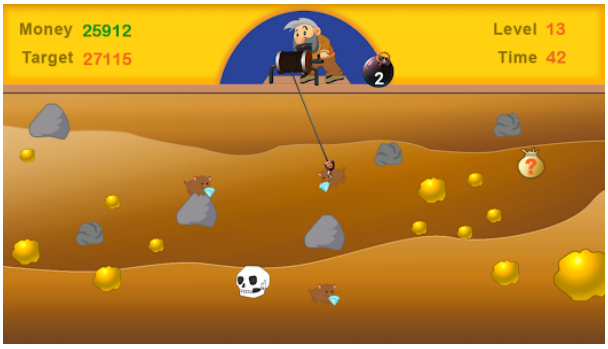
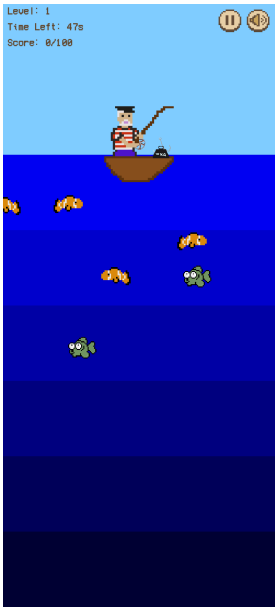
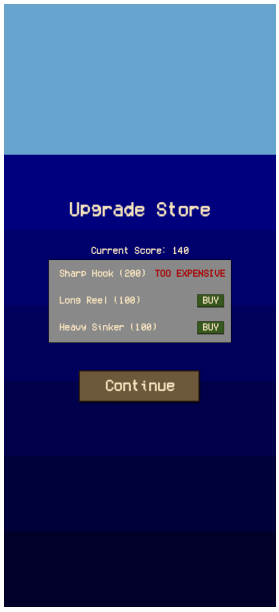


Figure 1: Screenshot from the game Gold Miner

Apart from the ui, our game Fish Miner is very similar to Gold Miner. The key features like the shop, leaderboard and hook functionality are the same. The main differences are the effects of the upgrades and that the fish move, meaning the player has to account for movement when aiming. The game is also in portrait mode.



(a) Fish Miner game



(b) Fish Miner store

Figure 2: Screenshots from Fish Miner

2 Architectural & Design Pattern Implementation

2.1 Architectural Patterns

When developing this project, we chose the design patterns we believed would help us achieve our quality attributes and meet the requirements we defined. In the process of designing the architecture and creating the development plan, we had a clear vision of how these patterns would be applied. However, during implementation, limiting the extent and commitment to each pattern became a challenge, as aspects of some patterns conflicted with others. In this section, we describe how the various patterns were implemented and the compromises we made so that the patterns would support—not hinder—us.

2.1.1 Layered Architecture

Our project has three modules, where the Core module is, as the name suggests, the core of the project. We decided to implement a layered architecture, separating the three main responsibilities into three layers:

1. **UI Layer**: Presentation and user interactions
2. **Domain Layer**: Business logic and state
3. **Data Layer**: Data access and communication with external modules

Additionally, we have a public infrastructure "layer" where global dependencies reside. To handle communication between layers, we borrow the concept of ports from hexagonal architecture to make the direction of communication explicit. An **outgoing port** declares the contract that this layer needs another layer or module to fulfill in order to "drive" it; an **incoming port** exposes the methods the layer offers to its drivers.

In the initial planning phase, we thought the Model–View–Controller pattern would be suitable; however, we found it conflicting and hard to grasp in combination with the ECS pattern. Instead, we kept some MVC terminology and incorporated it into our layered architecture. The model is `GameContext`, which implements the outbound port `IGameContext` (defined in the UI layer). When we create each screen (view), we inject only the `IGameContext` interface, so the UI sees *exactly* the methods it needs—nothing more.

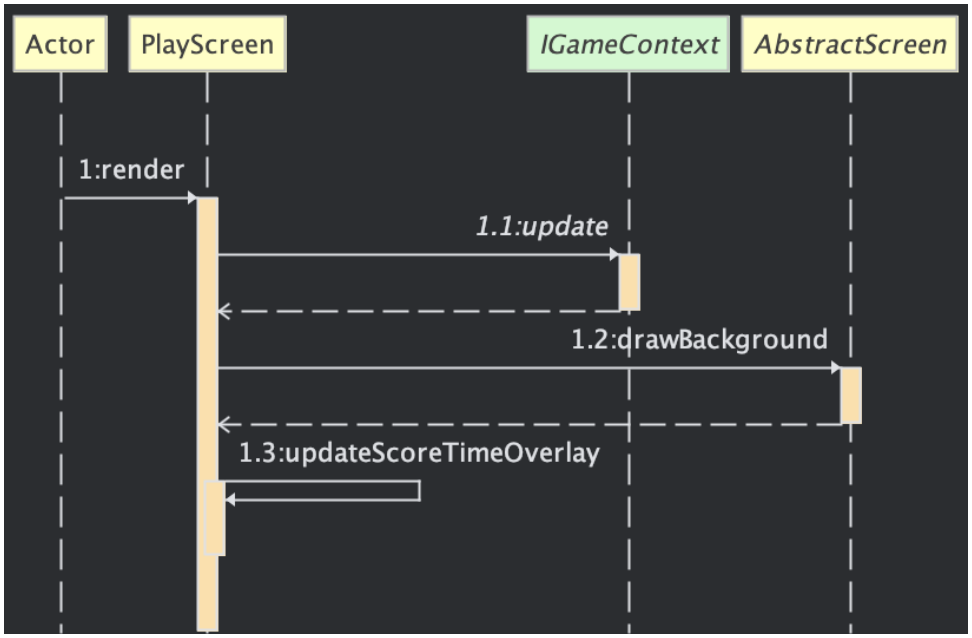


Figure 3: Shows the PlayScreen updating the IGameContext with no knowledge of the GameContext

This implementation allows us to abstract away each layer's dependencies, resulting in loose coupling and easily replaceable layers.

Another benefit of implementing the layered architecture using ports and adapters is how it supports other architectural patterns. In particular, each port/adapter pair forms a little client–server relationship. For example, the Android and Desktop modules implement the outbound port defined by the data-access layer—acting as the "server" that fulfills exactly the data-fetching operations the Core (the "client") requires.

2.1.2 Client–Server

As mentioned above, every port boundary represents a tiny client–server relationship. The caller of an interface is the **client**, and the implementer of that interface is the **server**. The primary implementation of this pattern lies in the communication between the Core and the Android module, and in how data is posted to and fetched from our database. We chose Firebase as our database, since our game does not require complex session handling or asynchronous actions—only immediate updates to the global leaderboard.
In our implementation, the Core (client) defines the access points via the data layer's outgoing ports. The Android module (server) implements those ports and fulfills the requests by talking to Firebase. To illustrate:

- `ILeaderBoardService` is defined in the Core's outbound port (`data.ports.out`).
- The Android module's `AndroidLeaderboardService` implements `ILeaderBoardService`.
- The `leaderboardFetcher` is injected with an instance of `ILeaderBoardService`.

At runtime, `AndroidLeaderboardService` itself becomes an HTTP client to the Firebase server, but from the Core's point of view, the Android module is the "server" that satisfies exactly the data-fetching contract it declared.

2.1.3 Event-Driven Architecture

Even though we used many interfaces for ports, our primary communication mechanism was an event-driven architecture—publish/subscribe via an event bus.

Our event bus is named `GameEventBus`, as the initial idea was that it would only handle domain-specific events. However, as time went on, we saw the benefit of having the UI layer send various events to the domain, further decoupling the layers. We moved `GameEventBus` to the public infrastructure package and had all events and listeners implement the ports the event bus required.
This initially made the boundaries of its usage unclear and added some complexity in confirming which events were triggered when. However, we introduced a logger (explained in depth later) to solve this. The event bus then proved to be a viable solution: to introduce a new class, we simply register it as a listener or have it dispatch events.

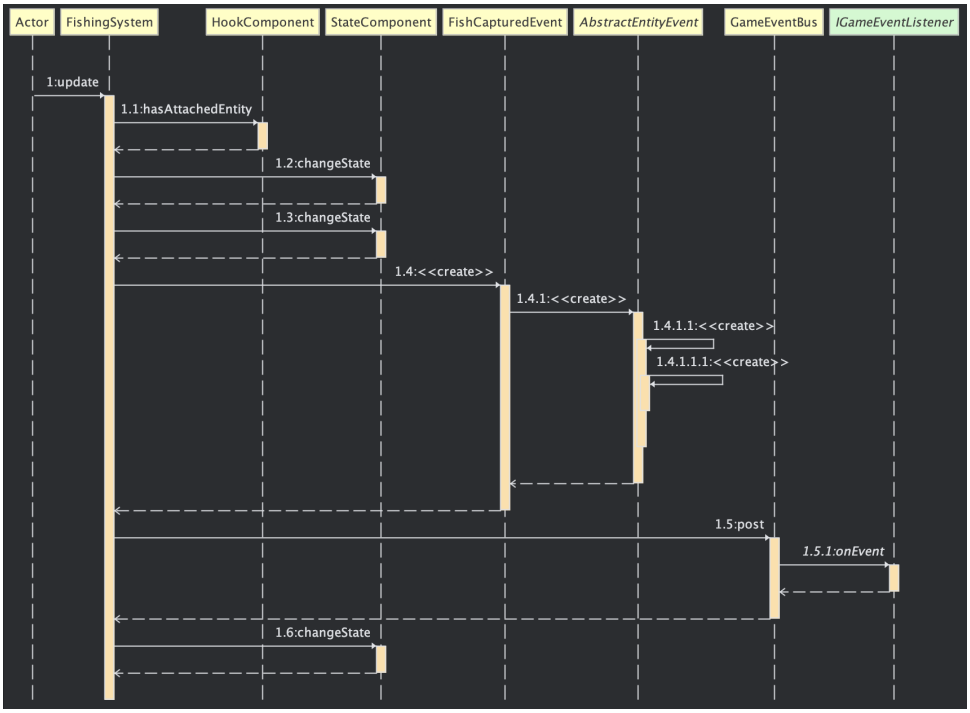


Figure 4: Shows a ECS system sending an event out from the domain to another layer

Integrating event-driven architecture with layered architecture worked seamlessly, as we simply implemented each event in the ports. This allowed us to keep track of which layer was the publisher and which was the subscriber. Although perhaps unorthodox, this

approach clearly communicated to the team which events a layer would send and which events its driver needed to handle.

The downside is that a centralized event bus can eventually affect game responsiveness. However, in the next section we outline how we would address this if it became a problem.

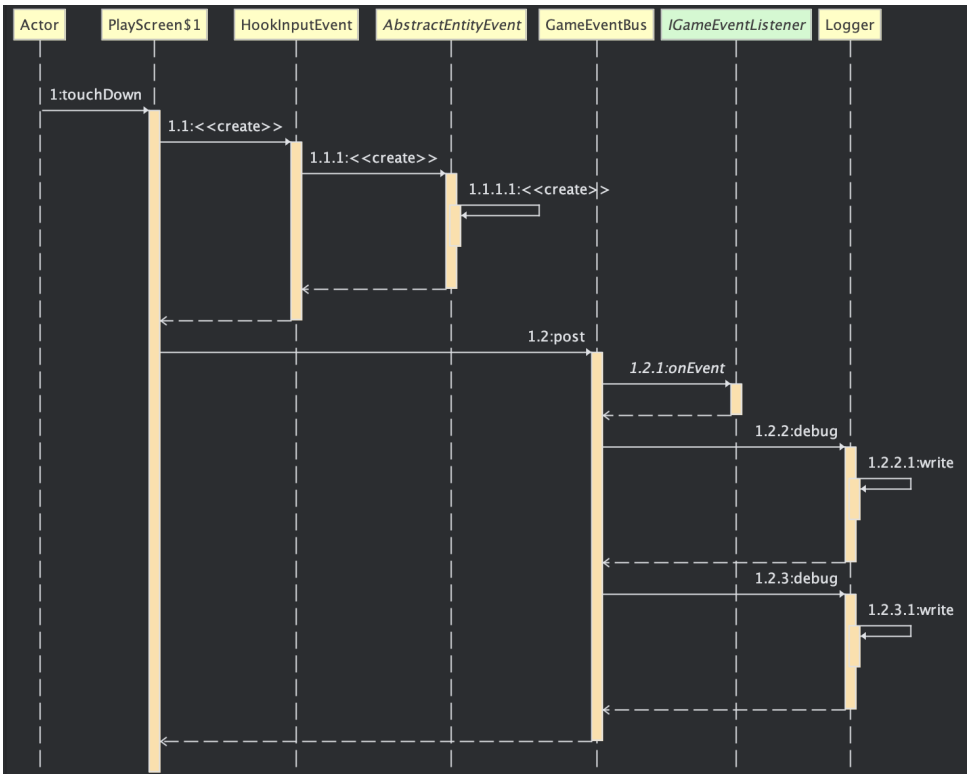


Figure 5: The series of events when the a touch input is triggered to the GameEventBus and illustrates how the Logger logs the occurance.

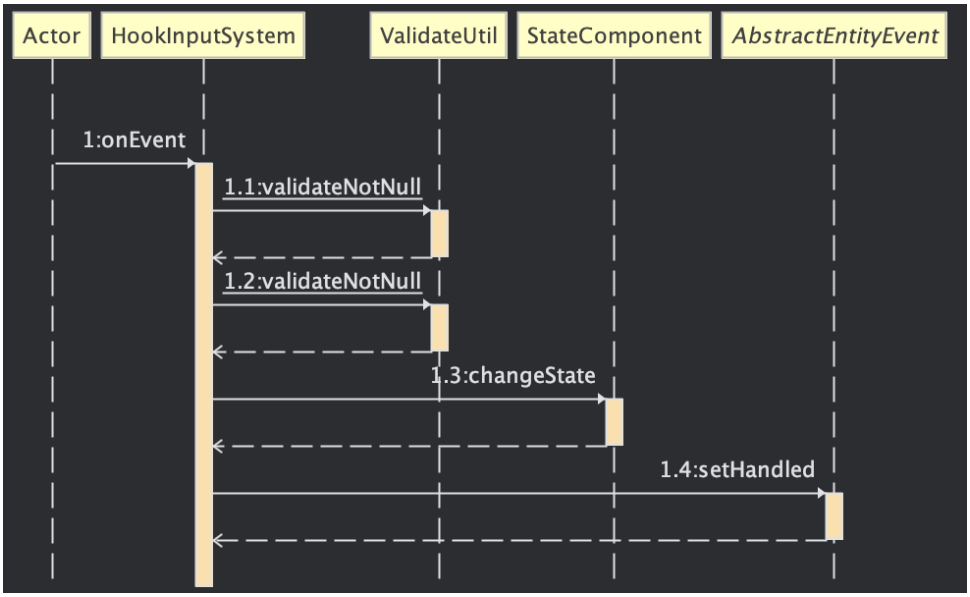


Figure 6: Illustrates the series events after the IGameEventListener recieves the IGameEvent from the GameEventBus

2.1.4 Entity-Component System

The entity-component system is the core of our game. We implemented it using Ashley's ECS framework for LibGDX. All game objects—sharks, fish, and trash—are entities. We also combined multiple entities into a single "super-entity" for the player. This allowed us to introduce new upgrades (hooks, sinkers) easily, and even unconventional upgrades like a hook with an attached fish that collects surrounding fish.

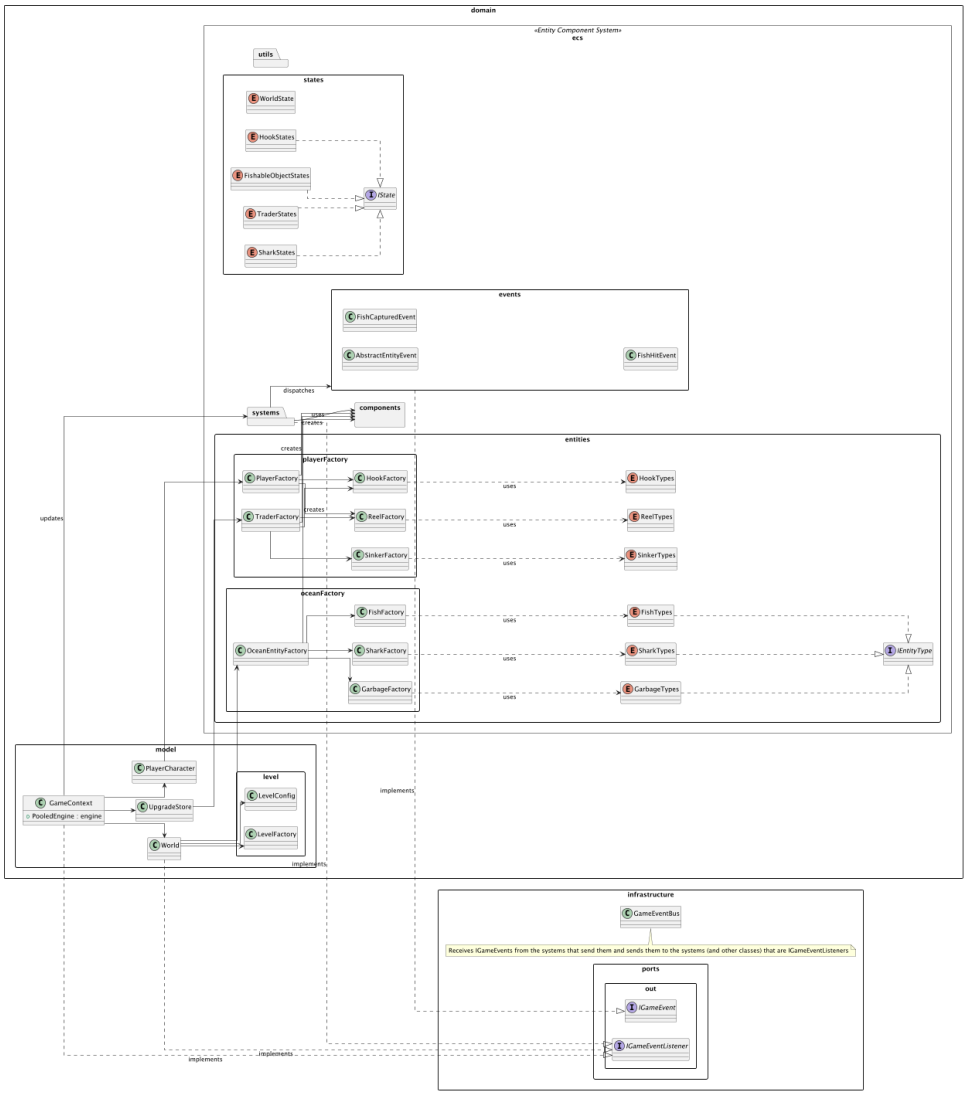


Figure 7: Shows the implementation of ECS and it relationship with the factories and the event bus

Initialization in `GameContext` All ECS setup happens in the `GameContext` model class, which serves as a facade between our domain logic and the LibGDX framework. In its constructor, `GameContext`:

1. Creates the engine (`PooledEngine`).
2. Initializes factories to build entities and attach appropriate components.
3. Adds all systems to the engine.
4. Instantiates the `PlayerCharacter`, `World`, and `UpgradeStore`, each of which hides any LibGDX dependencies behind a port interface (`IGameContext`, `IPlayer`, etc.).

Because the UI layer only ever sees the `IGameContext` (and related port interfaces), it remains completely decoupled from Ashley and other LibGDX internals.

Our implementation was only partially successful, as our initial knowledge of ECS was limited. We often used events via `GameEventBus` instead of having systems communicate state through built-in notification methods. This led to a heavier reliance on the event bus for both domain-specific and layer-to-layer communication. It also blurred the line between when to use systems/components versus regular classes. However, once the foundation was laid, the benefits increased exponentially. Even so, the groundwork required before seeing benefits might not have been worth it given the project's size and timeframe. Nevertheless, we achieved our primary quality attribute—performance—and the project is now highly modifiable, making further development more efficient.

The reliance on `GameEventBus` is something we will address in future iterations. Now that we understand the framework's internal notification capabilities better, we could use them to reduce latency and avoid performance bottlenecks from an overloaded event bus. Since reducing the cost of an algorithm in one system benefits all entities that system concerns, we gain performance leeway. This further illustrates the flexibility and benefits of our ECS implementation.

2.2 Design Patterns

2.2.1 Singleton

In our project, these singletons serve purely infrastructural roles and are initialized early in the application lifecycle. We mitigate most downsides, such as hidden dependencies, by keeping their scope narrow and purpose-specific, and by avoiding business logic inside them.

There is one exception, however, which is the `ScreenManager`. Early on we found this beneficial because the UI layer had direct access to it. Since we refactored the project so that the UI requested screen changes via the event bus, the global access to this class is unnecessary and will be addressed in future iterations.

We do, however, find the `Logger`, `Assets`, `Configuration`, and `GameEventBus` to be sensible singletons. By making `Logger` globally accessible, we ensure centralized logging and simplified debugging—any class can log events. Combined with the event-driven architecture, this allows us to analyze logs for unregistered listeners, event timings, and more. Although singletons can hinder testability, our implementation trades that cost for easier debugging, which in turn helped us better utilize patterns that enhanced modifiability.

2.2.2 Factory Pattern

We use the Factory pattern to centralize and encapsulate the complex logic of creating game-specific objects. We earlier mentioned how factories were used for ECS entity creation; we also apply a factory for level configuration and buttons for the ui.

For ECS, we use "super factories" (`PlayerFactory`, `OceanEntityFactory`, `TraderFactory`) that delegate to protected "sub factories." This separation allowed different team members to work in parallel—one on fish entities, another on hooks—without stepping on each other's toes. Each factory consumes enums from our types directory, centralizing

entity parameters (speed, value, etc.) for easy balancing and minimal redundancy.

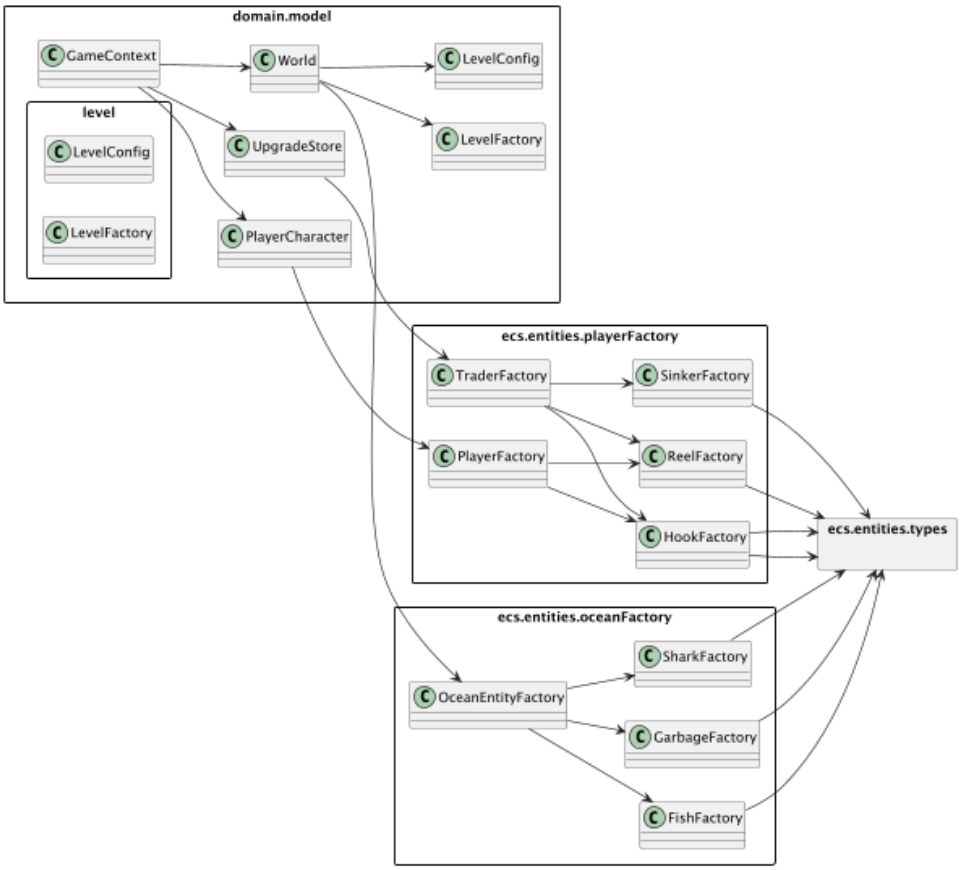


Figure 8: Illustrates our implementation of the factory pattern

A more interesting use of this pattern is the `LevelConfigFactory`, which determines spawn configurations per level and dynamically adjusts target scores based on past performance. By extracting all level-setup logic into a factory, we keep ECS systems and rendering code clean, and make level generation easy to test and modify.

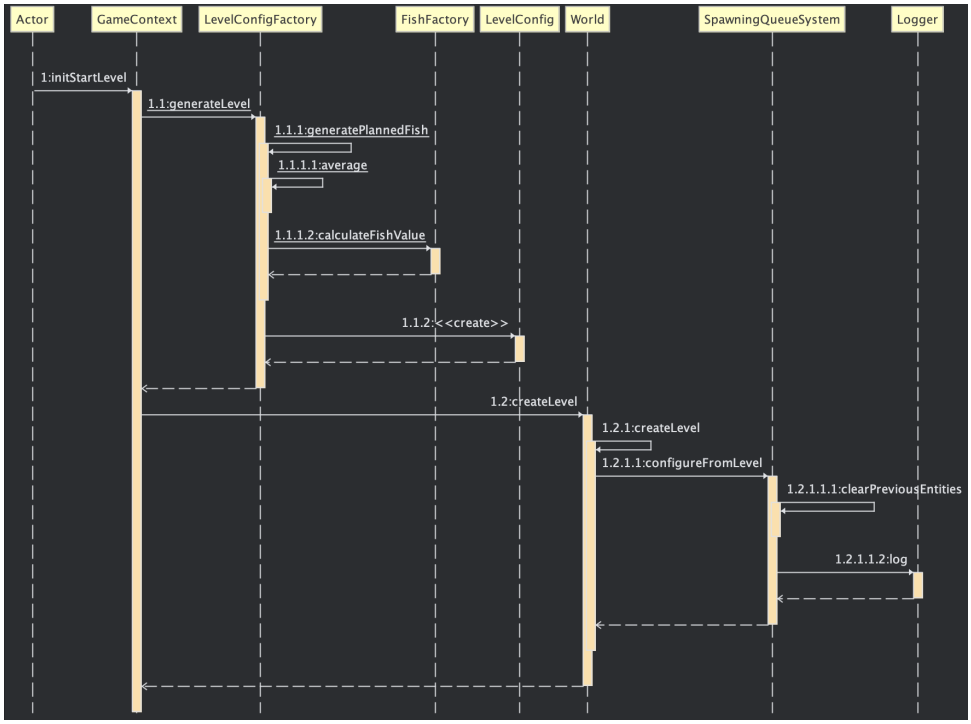


Figure 9: Sequence Diagram showing the creation of a new level

2.3 Reflections on Design Choices

In retrospect, our combination of layered architecture, event-driven design, and ECS provided a powerful yet complex foundation for game development. While some patterns introduced overhead, especially early on, they ultimately enabled a modular, scalable system aligned with our goals of modifiability and performance.

Trade-offs we encountered:

- **Increased complexity:** Combining multiple architectural patterns made onboarding and collaboration harder, especially for new contributors unfamiliar with ECS or event-driven models.
- **Overuse of event bus:** Centralizing all communication through the event bus decoupled logic but led to debugging challenges and potential performance bottlenecks.
- **Factory depth and duplication:** While factories enabled reuse, they also introduced nesting and made tracing logic harder—especially when factories delegate to sub-factories.
- **Limited short-term benefit from ECS:** ECS provided long-term modifiability but required upfront effort to understand, set up, and integrate correctly, which slowed early development.

Future iterations would involve reducing reliance on global state, simplifying factory hierarchies, and refining event flow to further enhance responsiveness and maintainability.

3 User Manual

This section offers a setup guide for the game and a thorough description of the gameplay.

3.1 Installation

3.1.1 Compiled Version (Android/Desktop)

1. **Download the APK from github releases**
2. **Run the Game**
    - **On Android:** Open the downloaded APK file.
    - **On Desktop:**
        - Open Android Studio.
        - Click *Profile or Debug APK*.
        - Select the APK file.
        - **Requirements:** Android Studio, latest Android SDK, and a selected device/emulator to run on.

3.1.2 Entire Codebase (Desktop)

1. **Download and Setup**
    - Clone the repository from GitHub or download the source files.
    - Ensure that Java (version 8 or above) and Android Studio are installed.
    - Open Android Studio, select *Open Project*, and navigate to the `Fish Miner` folder.
2. **Dependencies**
    - The project uses the LibGDX framework and external libraries such as Ashley ECS and Scene2D.
    - Dependencies are managed automatically by Gradle.
    - Missing dependencies can be resolved by syncing Gradle in Android Studio.
3. **Running the Game**
    - Open Android Studio and click the *Run* button to build and deploy the game.
    - Choose a target device (emulator or connected Android device), then click *Run*.
    - To run on desktop, select the `Lwjgl3Launcher.java` file as the main class and run it.

3.2 Navigating the game

3.2.1 Starting the game

When you start the game, you will be greeted by the **Menu Screen**.

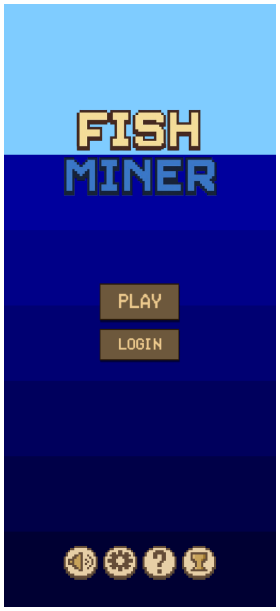


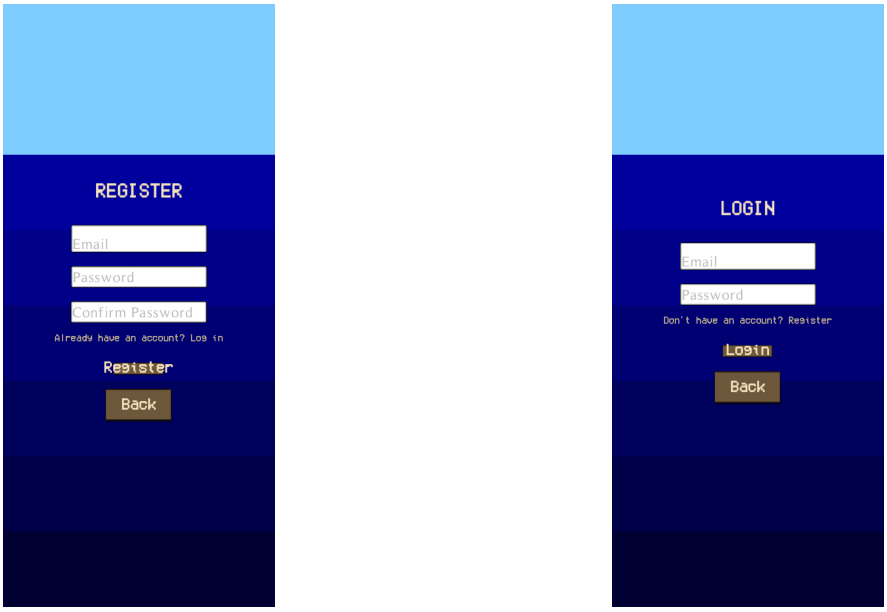
Figure 10: Menu Screen

On the menu screen you can press the following buttons:

- **Play:** Starts a new game.
- **Login:** Navigates to login sceen
- **The trophy:** Navigates to the online leaderboard which desplays the top 10 scores.
- **The questionmark:** Navigates to the tutorial
- **Mute/Unmute:** Toggles the Music playing
- **Settings:** Navigates to the settings screen where the music volume can be adjusted

3.2.2 Register to compete on the leaderboard

To save your score and compete with others you need to be logged in. This can be done in the **Login screen**.



(a) Login (b) Register

Figure 11: Login screen

3.2.3 Settings

In the settings you can adjust your music volume and go back to the menu.

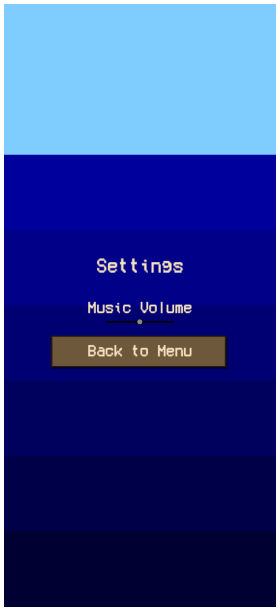


Figure 12: Settings screen

3.2.4 Leaderboard

The online leaderboard displays the top 10 scores among users. To see your own high score, you need to reach top 10 and be displayed. Remember to login first!

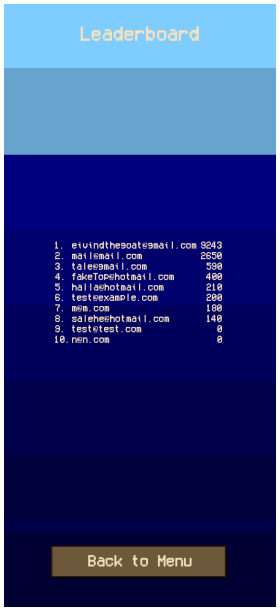


Figure 13: Leaderboard screen

3.2.5 Tutorial

If you are ever unsure of how to play. Press the question mark on the menu sqreen. This opens the tutorial, where you can swipe trhough images that explain the game.

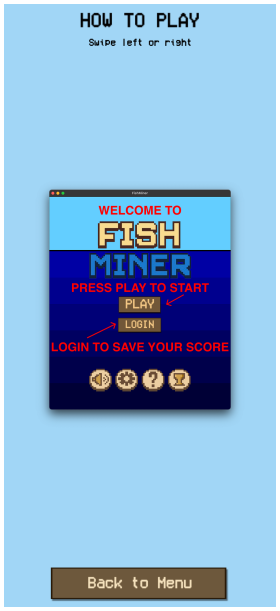


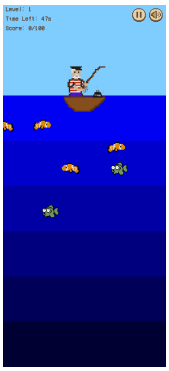
Figure 14: Leaderboard screen

3.3 How to play

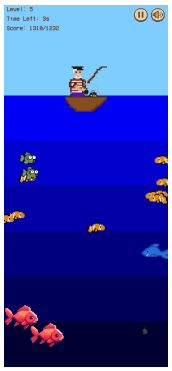
3.3.1 Objective:

- Your goal when playing Fish Miner is to catch enough fish to reach a target score and advance to the next level.
- As you progress, you unlock additional levels with increasing difficulty.

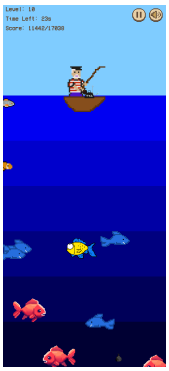
- Fish that are further down are worth more, but require more time and longer reels to catch.



(a) Level 1



(b) Level 5

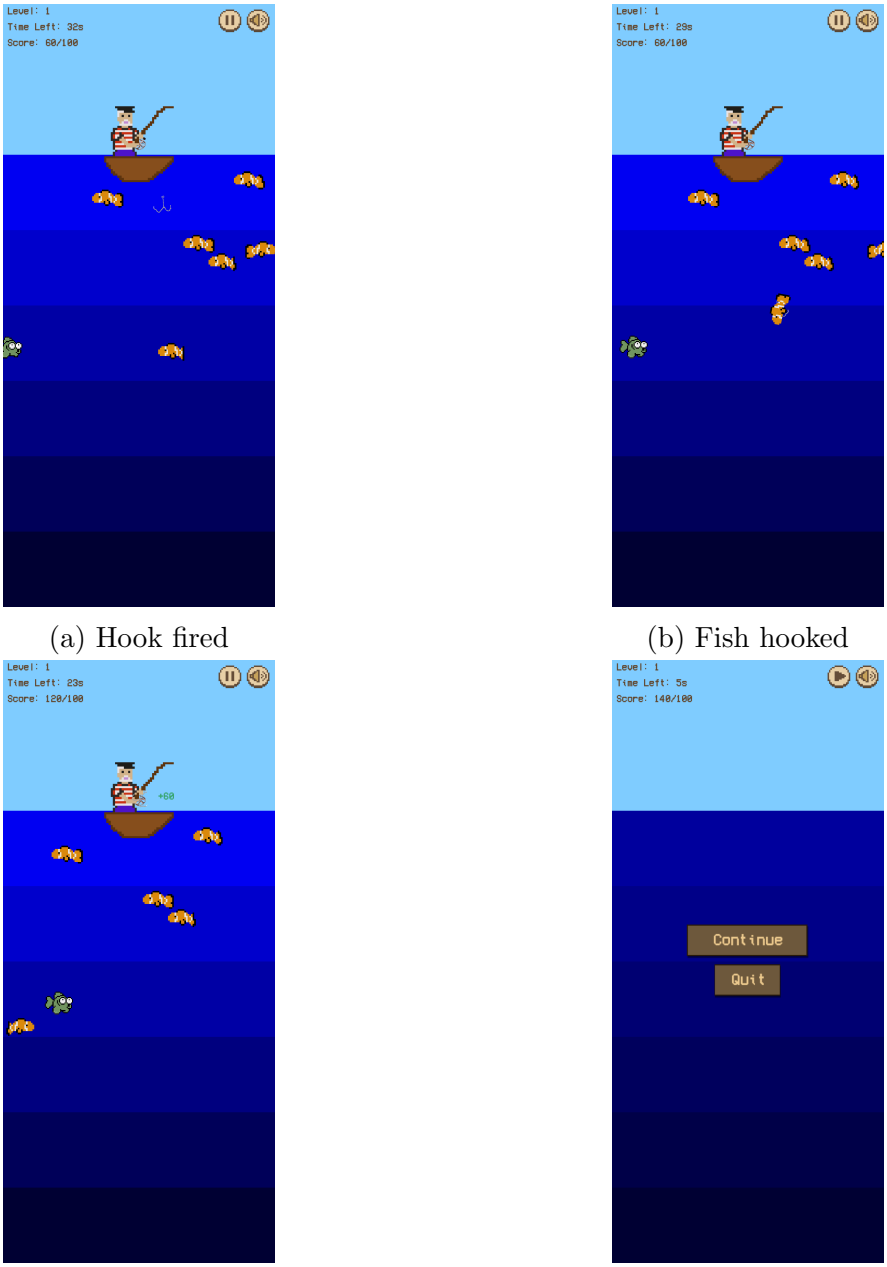


(c) Level 10

Figure 15: Fish Miner levels

3.3.2 Gameplay Controls

- The hook automatically swings back and forth. **Tap the screen** anywhere to release the hook in the direction it is pointing. Aim for the fish!
- **Toggle the music** by pressing the mute/unmute button.
- **Pause** at anytime by pressing the button during gameplay.
- **Quit the Game:** If you want to exit the game and go to the menu, tap the **Quit** button in the pause menu. This resets the game.



(a) Hook fired

(b) Fish hooked

(c) Points gathered

(d) Game paused

Figure 16: Fish Miner gameplay

3.3.3 Store and Upgrades

After each win, the **Upgrade Store** will appear. Here you can buy upgrades which last until you lose. You can upgrade your hook, reel or sinker once between each level. When you buy upgrades, the cost is deducted from your score, so use your points wisely!

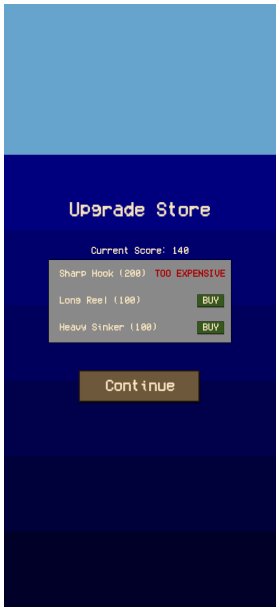


Figure 17: Upgrades screen

These are all the upgrades in the order they appear:

- **Sharp Hook:** Increases the hitbox of the hook, making it easier to catch fish.
- **Long Reel:** Increases the maximum depth the hook can reach.
- **Fast reel:** Increases the speed at which you reel in fish.
- **Legendary reel:** Increases both reel-speed and reach.
- **Heavy Weight:** Increases the speed of the hook's trajectory.
- **Heavier Weight:** Increases the speed of the hook's trajectory further

Once you have purchased the desired upgrades, you can continue the game and proceed to the next level.

3.3.4 Game Over

If you do not reach the target score within 60 seconds you will lose the game. The score you achieved will then appear. If you are logged in, your score is automatically submitted to the leaderboard. From here you can either go to the menu or see if you made top 10 by going to the leaderbaord.

Figure 18: Game over

4 Test Report

This section contains test reports for both functional- and quality requirements. These follow the same id's as the requirements in the requirement document.

4.1 Functional Requirement testing

FR1: The hook automatically swings back and forth and is "fired" when the user clicks.	
Executor:	
Date:	22.04.25
Time used:	5 seconds
Evaluation:	Success
Comment:	It took 5 seconds from the start point of the hook, to the end point, and then fire the hook.

FR2: The hook must catch a fish when it comes into contact with one.	
Executor:	
Date:	22.04.25
Time used:	5 seconds
Evaluation:	Success
Comment:	Did this in the same test as FR1, which took 5 seconds.

FR3: The system must track how quickly the player reaches the goal.	
Executor:	
Date:	22.04.25
Time used:	60 seconds

Evaluation:	Success
Comment:	The goal of the game is to get enough points to get to the next level before the time runs out. When the time runs out, the game has tracked the amount of points you have managed to attain.

FR4: The game must have a countdown timer for each level.	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success
Comment:	You can see that the game has a countdown timer on one minute before you get to the next level.

FR5: The game must notify the user when time is running out.	
Executor:	
Date:	22.04.25
Time used:	50 seconds
Evaluation:	Failure
Comment:	This function is not implemented in the game.

FR6: If the player does not reach the goal before the timer ends, the level should end.	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success
Comment:	After 60 seconds the game ended when the player did not have enough points to get to the next level.

FR7: The game must allow progression through levels when the goal is reached.	
Executor:	
Date:	22.04.25
Time used:	188 seconds
Evaluation:	Success
Comment:	After managing to reach the point requirements to reach the next levels, you will be able to buy upgrades for your character, and therefore be able to fish deeper down and faster reel.

FR8: Each new level should increase difficulty by adjusting speed, obstacles, or fish distribution.	
Executor:	
Date:	22.04.25

Time used:	60 seconds
Evaluation:	Success
Comment:	After 60 seconds the game ended when the player did not have enough points to get to the next level.

FR9: The game must include different types of fish with associated values based on size or rarity.	
Executor:	
Date:	22.04.25
Time used:	195 seconds
Evaluation:	Partially success
Comment:	After a few seconds you realize that the different fish gives you different scores. In the beginning, that is depended on the distance from the fisherman to the fish. After a few levels, bigger fish spawn deeper that is worth more.

FR10: Small fish swim near the surface, bigger fish swim in the deep end	
Executor:	
Date:	22.04.25
Time used:	186 seconds
Evaluation:	Success
Comment:	After three levels, bigger fish appears deeper down.

FR11: There should be obstacles that waste time when hit with the hook.	
Executor:	
Date:	22.04.25
Time used:	20 seconds
Evaluation:	Failure
Comment:	The fishing hook does not get affected when hitting obstacles regarding time waste.

FR12: The user can earn money/points based on the fish they catch	
Executor:	
Date:	22.04.25
Time used:	30 seconds
Evaluation:	Success
Comment:	The user gets bigger score depending on the size of the fish and how deep it is.

FR13: The user must be able to access the trader menu.	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success

Comment:	The user will be able to buy upgrades after the time has run out and the player has enough points.

FR14: The user must be able to purchase upgrades with acquired currency	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success
Comment:	The user will be able to buy upgrades after the time has run out and the player has enough points.

FR15: The user should see the leaderboard.	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success
Comment:	The user will be able to see the leaderboard when they lose after the timer has run out.

FR16: The leaderboard should be updated once a score arrives.	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success
Comment:	If you have registered a user and is logged in, your score will appear on the leaderboard after you lose.

FR17: The user should be able to find themselves on the leaderboard	
Executor:	
Date:	22.04.25
Time used:	60 seconds
Evaluation:	Success
Comment:	If you have registered a user and is logged in, your score will appear on the leaderboard after you lose, requiring that you have a high enough score.

FR18: The user should be greeted with a menu when the app loads.	
Executor:	
Date:	22.04.25
Time used:	3 seconds
Evaluation:	Success
Comment:	The user gets a menu, where it is possible to choose to play, login or go to settings.

FR19: Clicking "Start Game" should start a new game.	
Executor:	
Date:	22.04.25
Time used:	8 seconds
Evaluation:	Success
Comment:	The user can press a play button, and then a new game starts.

FR20: The user should be able to mute the game.	
Executor:	
Date:	22.04.25
Time used:	10 seconds
Evaluation:	Success
Comment:	The user can press a mute button in the menu page when the app opens, or in-game.

FR21: The user should be able to pause/resume the game.	
Executor:	
Date:	22.04.25
Time used:	12 seconds
Evaluation:	Success
Comment:	The user can press a pause button in the upper right corner, and can resume the game after.

FR22: There should be a tutorial available for the user.	
Executor:	
Date:	22.04.25
Time used:	12 seconds
Evaluation:	Success
Comment:	In the menu page, you can press a button, where you can find a how to play tutorial.

FR23: The game must support both portrait and landscape orientations	
Executor:	
Date:	22.04.25
Time used:	12 seconds
Evaluation:	Failure
Comment:	The game is only supporting portrait mode.

FR24: The game must save player progress, including level completion and earned currency.

Executor:	
Date:	22.04.25
Time used:	80 seconds
Evaluation:	Success
Comment:	When you earn enough points and the time runs out, you can see that your score has been saved and your upgrades is brought with you to the next level.

FR25: The game must allow reloading progress after restarting the app	
Executor:	
Date:	22.04.25
Time used:	70 seconds
Evaluation:	Failure
Comment:	The game does not give you back your progress if you restart the app mid-game. However, it does save the leaderboard and your former high-score.

4.2 Quality Requirement testing

M1: Add a new entity or upgrade	
Executor:	
Date:	22.04.25
Environment:	Runtime, Design time
Stimuli:	Wants to add a new entity or upgrade
Expected response measure:	Should be added in under 1 hour, with no side effects
Observed response measure:	Approximately 30 minutes
Evaluation:	Success
Comment:	Adding a new entity or upgrade is fairly easy to implement.

M2: Change entity or upgrade	
Executor:	
Date:	22.04.25
Environment:	Runtime
Stimuli:	Wants to change existing entity or upgrade
Expected response measure:	Approximately 20 minutes
Observed response measure:	Approximately 15 minutes
Evaluation:	Success
Comment:	It is fairly easy now to change already existing entities or upgrades

M3: Change game configurations	
Executor:	
Date:	25.04.25
Environment:	Design time
Stimuli:	Wants to change game configurations (e.g. screen size, entity speed, score goal)
Expected response measure:	Make change in a single location and the change is applied immediately across affected system, without additional manual updates. Under 10 minutes.
Observed response measure:	Changing entity speeds screen size, depth etc. was easy to modify, but somewhat confusing. 30 minutes.
Evaluation:	Success
Comment:	The getBaseMovement() in the Configuration file impacted the hook- and reel-speed as well, causing a little confusion.

U1: Helps developers improve gameplay	
Executor:	End user
Date:	22.04.25
Environment:	Runtime
Stimuli:	Wants to improve gameplay by providing development team with directive to: - Modify or add features/upgrades, or - Fix bugs or game balance
Expected response measure:	User understands the feedback system with no more than one error. Approximately 2/3 of users should be able to fill in the form within 2 minutes.
Observed response measure:	There were no feedback system for users implemented.
Evaluation:	Failure
Comment:	

U2: Learning to play game	
Executor:	User/Player
Date:	22.04.25
Environment:	Runtime
Stimuli:	Wants to learn how to play the game
Expected response measure:	Finishes the tutorial within 1 minute and is able to play the game with no more than 3 errors in the first game session
Observed response measure:	Takes 8 seconds to find the tutorial and 1 minute to finish the tutorial
Evaluation:	Success
Comment:	Tutorial is optional by button. Could maybe have been automatically started when opening for the first time to improve user experience.

U3: User can send feedback	
Executor:	User/Player
Date:	22.04.25
Environment:	Runtime
Stimuli:	Wants to help developers solve bugs and improve the game
Expected response measure:	Approximately 1-5 minutes from when the game starts, and the player can send feedback
Observed response measure:	There is no feedback button
Evaluation:	Failure
Comment:	No feedback button was implemented.

U4: Player is able to mute audio	
Executor:	User/Player
Date:	22.04.25
Environment:	Runtime
Stimuli:	Wants to customize the game experience based on personal preferences
Expected response measure:	7 seconds
Observed response measure:	4 seconds
Evaluation:	Success
Comment:	It is both a mute button in the menu screen and in-game.

P1: Game should run smoothly	
Executor:	End user
Date:	22.04.25
Environment:	Runtime
Stimuli:	User is playing the game during a standard gameplay session
Expected response measure:	The game continues to run smoothly
Observed response measure:	180 seconds.
Evaluation:	Success
Comment:	Played 3 rounds and the game continued to run smoothly.

P2: Finds the leaderboard	
Executor:	Logged-in user
Date:	22.04.25
Environment:	Runtime with internet connection
Stimuli:	User opens the leaderboard to view current rankings.
Expected response measure:	8 seconds in the menu screen

Observed response measure:	6 seconds
Evaluation:	Success
Comment:	The trophy logo on the button made the user quickly realize that was the leaderboard-button.

P3: Player can purchase upgrades using earned points	
Executor:	Logged-in user
Date:	22.04.25
Environment:	Runtime
Stimuli:	Player purchases an upgrade using earned currency
Expected response measure:	65 seconds for the player to finish the level, buy and then equip the upgrade
Observed response measure:	63 seconds
Evaluation:	Success
Comment:	The store checks if the player has enough cash to buy the upgrade, and for the player to equip it.

5 Relationship with Architecture

In this section, we compare our original architectural intentions with the actual implementation and highlight the key compromises and deviations from our plan as defined in our development view.

5.1 Planned vs. Actual

- **Model–View–Controller Pattern:** The biggest difference between our planned and actual implementation was our intended use of MVC.

  After completing the initial steps, we discovered that MVC easily compatible with the Entity Component System. Since a ECS typically has some variation of an input system and rendering system that would belong along the other systems, so forcing MVC became a hindrance rather than a help; It became more about "how do we force our project into this pattern?" instead of "how can this pattern work in our favor?".

  We therefore discarded the strict MVC pattern, though we retained some MVC terminology to clearly communicate the roles of the `model` and `view` packages.

  While it is possible to combine MVC with layered and event-driven architectures, doing so here would have been overkill.

- **State Pattern:** We originally planned a more explicit State pattern, but ECS-patterns, typically, inherently incorporates state via its components. Committed to an event-driven approach, we ended up mixing both patterns suboptimally.

  In practice, we used a state component in ECS systems and let `GameContext` track the `World` state. Although our plan was sound, we should have relied more on ECS's

built-in notifications rather than our own event-driven mechanism for domain state changes.

- **Layered Architecture:** Initially, we underestimated how strictly a layered architecture must be enforced and how layers should be represented (packages versus modules).

  We chose the simpler package-based approach. In hindsight, module-based separation would have provided stronger build-time enforcement. Nevertheless, starting with packages allowed incremental adoption and can evolve into modules when appropriate.

- **Client–Server Relationships:** The Client–Server pattern was implemented as planned, including the "mini" client–server relationships at each port boundary. We consider this aspect faithful to our original design.

- **Event-Driven vs. ECS Messaging:** We opted for a single `GameEventBus` for domain, UI-to-domain, and ECS events. Over time, this blurred the lines between cross-layer events and internal ECS notifications.

  Although convenient, a clearer split—using ECS messaging for system-to-system communication and reserving the event bus for layer-to-layer events—would improve traceability and performance.

- **Singleton Usage:** Singletons for `Logger`, `Assets`, `Configuration`, and `GameEventBus` worked as intended. However, refactoring `ScreenManager` out of a global singleton proved difficult once it was widely adopted. In future iterations, we would invert its instantiation or remove global access entirely.

- **Factory Pattern:** We planned to use the Abstract Factory pattern but initially implemented simple factories that evolved into a quasi-abstract approach. Early adoption of the standard pattern would have reduced custom complexity. Our `LevelConfigFactory`, however, matched the plan and remains a clear, testable implementation.

- **Entity Component System:** Our implementation of the Entity Component System went as planned. Our only problem here was that we were unaware of the full potential of the framework we used. Along the way we experienced that there were better solutions to the ones we had used, primarily regarding the usage of events. The usage of events did however allow us to toggle wents on and off in order to increase performance by disabling costly systems when they are not needed. Additionally, it was hard to find the balance between a clean approach using getters and setters, and input validation and following what we interpreted as the conventional way, which is to have very minimal components. We opted for a middleground depending on the importance of validation and generally used getters and setters.

  For the systems on the otherhand, we had trouble with some systems becoming overly complex. This is simply a matter of experience and does not reflect poorly on or decicions in regards to the architecture.

5.2 Lessons learned

- Layers as packages do not communicate explicitly enough the importance of the boundaries between the layers, but is easy-to-implement solution early on. It does however require stricter access modifier for classes and methods.
- Event bus convenience can degrade traceability—consider refactoring ECS systems to native messaging.
- Singletons simple access is a double-edged sword. Only classes that truly justify the usage of this pattern should inherit it. If it is accessible everywhere; it will be used everywhere. It is easier to introduce a Singleton pattern that it is to revert one.
- Factory logic centralization aids maintainability, but gaining full understanding of the Abstract Factory Pattern early on would have been beneficial.
- Complex patterns/frameworks such as ECS, requires that every member of the team learns how this works and the capabilities of the framework. Such problems could be solved by frequent use of pair programming or other team-coordinating techiniques.
- Planning for too many patterns can be overwhelming for inexperienced teams like ours. Had we started from the beginning, we suspect that a more incremental approach would be beneficial. By doing so, we could update our development view more frequently and focus on implementing some pattern that solves that problem at hand in respect to our quality attributes. This would coordinate the team, while also helping us focus on what is relevant at the moment, in favor of trying to predict the future.

6 Challenges, Issues, and Lessons Learned

This section describes the main challenges we faced during the implementation of our game, what we learned from the process, and what we would do differently if we started over.

6.1 Teamwork and communcation

Several issues we had during development could be attributed to poor communication within the team.

Firstly, reaching a common understanding of the game, how it would work and what to prioritize was a challenge. Confusion around this lasted throughout the project. For example, while working on the shop system, three different group members implemented parts of it based on their own understanding. This led to several unused classes (i.e `UpgradeSystem`, upgradeItem and `InventoryComponent`) and unfinished work like the animated trader.

We also struggled to maintain the standards we agreed upon in the Architecture document. In the beginning we had a Kanban board, branches for each task, and did code reviews. As the deadline approached, however, we stopped following these processes. People started solving problems as they appeared without updating the board or opening issues. This led to abandoned pull requests and branches, like dev-old, and wasted time due to duplicate work and managing merge conflicts.

Lastly, frequent absences within the group, due to vacations or sick leave, made coordination challenging. Sometimes big changes were made while others were absent, making it difficult for them to catch up later. In a way, the challenges mentioned earlier made absence more of an issue, but absence likely also made those challenges worse, creating a cycle that reinforced the overall difficulty of working together.

6.2 Too many patterns at once

Because this was a software architecture and design project, we were excited to try many patterns and techniques. But using so many patterns at once while learning them for the first time was arguably too ambitious.

One reason was that there was not enough time to discover and learn all the benefits of each pattern. This made it difficult to know when to use what pattern. For example, as previously mentioned, we could have utilized more of the features within the Ashley framework rather than putting extra load on the GameEventBus.

This was also an issue because of the way the work was divided. In the beginning, two people worked on the UI, two on Firebase, and two on the game logic. While this seemed efficient, it created a problem later: only a couple of people really understood how the core game logic worked. When their tasks were done, the others found it difficult to contribute. Not only because they were unfamiliar with game logic code, but also had to understand how the different architectural patterns were implemented and how to build upon them.

The project also became excessively large for a game so small. With almost 150 different classes the project became difficult to navigate, especially given the knowledge gaps in the group. Although the tests show we successfully made the game easily extendable with new upgrades and levels, one could argue extent to which it was modular made it less modifiable. Debugging also became harder as issues could stem from a lot of places. Tests would have helped this, but we had no time to develop tests.

6.3 What we would do differently

This project has been a valuable learning experience – not only in software architecture and design patterns, but also in teamwork. Although we are happy with our efforts, we would have liked having time to polish and refactor the game. In addition to the lessons mentioned earlier, here are some strategies we would employ to better the outcome if we were to do it again:

6.3.1 Start small and together

Although we did some pair programming, we would benefit from rotating pairs and tasks more often. By combining this with making the core gameplay first and using fewer patterns, all members would be more on board. We would start by making the hook, fish and player using the ECS. Adding more patterns later, like separating the layers, and knowing when to use which pattern would then be easier as everyone would be familiar with the existing ones. This would probably seem less effective at first, but could help distribute the workload more evenly later and help reduce misuse of patterns.

6.3.2 Spend more time on the documents

Due to poor time management, we had to rush the requirement and architecture documents. As a result, we did not use them much during development other than slavishly trying to implement the architecture we came up with. This proved hard.

Spending more time working on the requirements and aligning our ideas of the game would have saved time and confusion later. Furthermore we should have used them more when designing the architecture. Rather than using architectural patterns that fit the game, we tried to make the game fit the architecture we had planned. By spending more time on and updating our architecture document along the way, we would likely improve our understanding of why we used patterns and made our project less complicated. It would also have reduced time spent altering the document before delivery.

7 Individual Contributions

Member	Contributions
August	Worked on Firebase implementation and integration, in addition to implementing Firestore leaderboard logic and API.
Eivind	Worked on Firebase implementation and integration, in addition to implementing Firestore leaderboard logic and API. Wrote the Test Report.
Emil	Wrote sections: Introduction, Setup guide and Challenges. Designed and worked on UI. Worked on upgrades, Assets-class and general performance issues.
Tale	Main focus was the entity component system, implementing the level logic and spawning system. Worked on the game play process.
Salehe	Worked on the UI screens (Login, Settings), Firebase integration, and leaderboard logic for both Desktop and Android. Contributed also to the score system and adding upgrades and the second and third section.
Jesper	Worked mostly on the ECS with focus on everything related to the player and trader entities, Event-Driven architecture and overall enforcement of the architecture and direction of dependencies. Was responsible for 2 (Design Implementation) and 5 (Relationship Architecture)

Table 36: Individual Contributions