NTNU

Kunnskap for en bedre verden

Department of Computer Science

TDT4240 - Project

Architecture

Authors:

NTNU email	Last name	First name
@stud.ntnu.no	etternavn	fornavn
@stud.ntnu.no	etternavn	fornavn
@stud.ntnu.no	etternavn	fornavn
xxxx@stud.ntnu.no	etternavn	fornavn
eliaswhe@stud.ntnu.no	Heimdal	Elias Ward
@stud.ntnu.no	etternavn	Snorre Kværnø
@stud.ntnu.no	Åsetternavnheim	fornavn

Game name: xyz

Group number: 17

Chosen COTS: xyz

Primary quality attributes chosen: xyz

Secondary quality attributes(s) xyz

Spring 2025

Table of Contents

1 Introduction

2 Architectural Drivers / Architecturally Significant Requirements (ASRs)

2.1 Significant requirements

This multiplayer game features an "eat-to-grow" mechanic where players compete to become the largest by collecting bites and eliminating smaller players. The game supports real-time multiplayer matches through a lobby system, tracks achievements through persistent leaderboards, and maintains player profiles across sessions. The following sections detail the key requirements for implementing these core features.

2.1.1 Multiplayer

The game features real-time multiplayer functionality that requires networking architecture to synchronize game states across multiple clients over the internet. The system must handle concurrent player interactions.

2.1.2 Core gameplay mechanics

Player growth and interaction

Players can increase their size by collecting "bites" scattered throughout the game world. Regular bites increase player size incrementally, while special bites grant unique powerups and abilities. The core interaction involves larger players eliminating other ones through collision: a player is eliminated if their noodle head hits another players noodle body. Victory can be achieved either by reaching a specific size or being the last player standing.

Power-up system

The game implements a dynamic power-up system that provides temporary special abilities to players. These powerups significantly affect player interactions and strategies, requiring careful balance in their distribution and effectiveness to maintain engaging gameplay.

2.1.3 Session management

Lobby system

Players begin at a lobby interface where they can create new game lobbies with unique identifiers or join existing ones using lobby codes. The system manages lobby participants and allows lobby creators to configure game settings before launch.

Game session flow

Each game session follows a structured flow, beginning with initialization when the lobby creator starts the game. The active gameplay phase features real-time player interactions until victory conditions are met, at which point the session transitions to a game over state. Players can then return to the lobby or explore new game options.

2.1.4 Persistent features

Leaderboard system

The game maintains a persistent top-ten leaderboard that tracks highest historical scores across all game sessions. The system updates dynamically as new high scores are achieved, providing competitive goals for players while showcasing their rankings and achievements.

User profile system

Player authentication and profiles serve as the foundation for the game's persistent features. The system enables unique player identification across sessions, tracks individual statistics and achievements, and manages secure player authentication. This profile architecture is essential for supporting the leaderboard system, multiplayer functionality, and long-term player progression.

2.2 Drivers

The architectural drivers define the key non-functional factors that influence the structure and design of the system. These drivers are shaped by both technical requirements and external constraints, ensuring that the architecture supports the project's goals effectively.

2.2.1 Modifiability

As the primary quality attribute, modifiability is central to the game's architecture. The system must be designed to accommodate changes efficiently, allowing for:

- Dynamic game configuration, enabling adjustments to player limits, game speed, and power-up properties without code modifications.

- A modular architecture, where gameplay mechanics, networking, and UI components are loosely coupled to support independent updates.

- Extendability, allowing new power-ups, abilities, or gameplay modes to be introduced with minimal refactoring.

These considerations ensure that the game can be adapted and expanded without requiring fundamental architectural changes.

2.2.2 Usability

The game must provide an intuitive and engaging experience for players of all skill levels. Key usability concerns include:

- Simple and responsive controls, allowing for fluid gameplay.

- Clear UI feedback, ensuring players can easily understand their status and available actions.

- A structured onboarding process, helping new players quickly grasp the mechanics.

These factors directly influence how UI and input-handling components are structured within the architecture.

2.2.3 Performance

As a real-time multiplayer game, performance is a critical concern. The system must:

- Minimize latency, ensuring synchronized game states across players.

- Optimize collision detection, preventing performance bottlenecks as player count increases.

- Ensure stable frame rates, allowing smooth gameplay even under high player load.

The architecture should support scalable networking and efficient resource management to meet these performance requirements.

2.2.4 Testability

To maintain reliability and facilitate debugging, the game must be structured for easy testing. This includes:

- Automated unit tests for core mechanics like movement, collision handling, and power-ups.

- Multiplayer simulation tools, enabling testing of networking conditions and edge cases.

- Logging and monitoring, providing insight into game state changes and system performance.

A testable architecture ensures that modifications and feature additions do not introduce unintended issues.

3 Stakeholders and Concerns

- developers (the group)

The development team aims to create an architecture that is modular, maintainable, and easy to modify. Clear documentation of system components, patterns, and tactics is essential for both current and future contributors. The team also seeks to expand their knowledge of architectural principles, including design patterns, modularization strategies, and quality attribute trade-offs. Ensuring an efficient workflow with well-structured and understandable code will make debugging and future expansions easier.

- players of the game

End users expect a smooth and responsive gaming experience. The game must be intuitive, with well-designed controls and UI elements that allow players to engage without frustration. Multiplayer performance is critical—users demand low latency, consistent frame rates, and a fair competitive environment. Features such as leaderboards, profiles, and power-ups must be seamlessly integrated and responsive to provide an engaging experience.

- course evaluators

The teaching staff will assess whether the architectural design aligns with project requirements, particularly modifiability. The system should be well-documented, structured using appropriate patterns and tactics, and demonstrate a clear separation of concerns. Code quality, testability, and adherence to best practices are key concerns for evaluators. The team must also be able to justify design choices and reflect on how the architecture supports the selected quality attributes.

- future students

If the project is referenced in later courses, it must serve as an educational example of well-structured software architecture. The codebase should be easy to read and extend, with clear documentation and explanations of key architectural decisions. Future students should be able to analyze and build upon the system without extensive refactoring.

4 Architectural Viewpoints

4.1 Logical view

Purpose

The purpose of the architectural view is to focus on describing the functionality the end-user experiences in a software product, but is also used for developers to understand the functionality. This view is relevant when looking at modifiability, integrability and testability.

Often the logical view is described as a UML class diagram with elements and relations between the elements. We will focus on describing the logical view for developers.

When describing software architecture to end-users one might do so on a higher level by dividing functionality into modules and explain how these work together textually.

Target audience

The target audience of the logical view is as stated end-users and developers. In our case these are people which are going to play the game, as well as us (the project group). The course evaluators will also be interested in the logical view for determining how well the group has executed on the modifiability and testability quality attributes.

4.2 Process view

Purpose

The purpose of the process view is to model a software systems internal processes and communication within the system. This can be described as sequence diagrams or activity diagrams, with sequence diagrams describing focus on describing runtime processes while activity diagrams focus on the workflow of the software.

This view focuses on quality attributes like performance, availability, safety and energy efficiency.

Target audience

The formal target audience for the process view is integrators and developers. As in the logical view we the group are the developers. Integrators would be a specific developer of the group focusing on process management, performance optimization and system integration.

We would also argue that course staff is a target audience because they have to understand the process of the software to evaluate performance and availability.

4.3 Development view

Purpose

The development view should describe software with the purpose of easing development for developers. It should inform the developers of how the software architecture is structured in such a way that it aids development. This can be by describing packages of the software and which subsystems these packages utilize (in other words imports/exports). The packages and components within this package diagram is often divided into layers communicating which components are visible to other components and how the software is grouped into different partitions.

The development view can also be represented by documenting how the software is development, as in planing and execution of plans for development.

Target audience

For this project the target audience of the development view is us the developers. In the first phase we act more like managers interested in creating plans for development, while in the later phase of the project we will focus more on the actual development and something like a component diagram will become more relevant.

4.4 Physical view

Purpose

The physical view should map out physical hardware and which parts of the software runs on which machines, as well has how these communicate. In this view one will demonstrate quality attributes like deployability, availability and performance through a deployment diagram. This diagram describes the architecture as a network of nodes, where different parts of the software run on these nodes.

Target audience

The target audience are system engineers which have to understand which physical hardware is involved, how the hardware communicates and what the hardware does. This would be a specific developer in the group focusing on system engineering, but also course evaluators which will evaluate the availability and performance quality attributes.

5 Architectural tactics

In this section, we will describe the architectural tactics used to meet the quality attributes and explain how these tactics will be reflected in the architectural design.

5.1 Modifiability

Tactics for controlling modifiability aim to manage the complexity, time, and cost of making changes [1]. To enhance modifiability, we will focus on three key tactics: increasing cohesion, reducing coupling, and deferring binding time.

5.1.1 Increase cohesion

Cohesion measures how strongly the responsibilities within a module are related [1]. Maximizing cohesion ensures that closely related responsibilities are grouped within the same module. This will reduce the likelihood that modifications in one module will impact others. We will focus on two tactics to increase cohesion:

- **Split Module:** If a module contains responsibilities which aren't closely related, it should be broken down into smaller, more cohesive modules. This will reduce the cost of future modifications.
- **Redistribute responsibilities:** If several similar responsibilities are scattered across different modules, they should be moved together, either by creating a new module or by gathering them in an existing module.

5.1.2 Reduce coupling

Coupling refers to the probability that a modification to one module will propagate to the other [1]. By minimizing coupling, we can reduce the likelihood that modifications in one module will impact others. We will focus on three tactics to reduce coupling:

- **Encapsulate:** Encapsulation restricts direct access to certain details of an element and exposes only what is necessary through a defined interface. This improves dependency management, thus reducing the probability that a change to one element will propagate to other elements.

- **Use an Intermediary:** By introducing an intermediary between interacting components, we can ensure that components don't need direct knowledge of each other. This will help break direct dependencies, reducing coupling.
- **Abstract Common Services:** If multiple components provide similar service, it can be beneficial to abstract them behind a common interface or intermediary. This allows other elements in the system to interact with a single unified service rather than dealing with each specific component separately, thus reducing coupling.

5.1.3 Defer binding time

Deferring binding time is a tactic used to delay the decision on specific values or parameters until later in the system's life cycle [1]. By doing this, we can make the system more flexible and modifiable, as changes can be made without needing to modify the code. This reduces the cost and complexity of making changes after the system has been built. In simple terms, instead of hard-coding values into the system at the time of writing the code (like during compile time), we allow those values to be determined at a later time, such as during runtime or deployment.

5.2 Usability

Usability refers to how easy it is for a user to accomplish a task within the system and the level of support the system offers to assist them in doing so [1]. To accommodate usability, we will focus on two main tactics: supporting user initiative and supporting system initiative.

5.2.1 Support user initiative

The goal of supporting user initiative is to allow players to have control over their actions and make adjustments when needed. Our primary tactic to support user initiative would be the ability for players to **cancel** out of a game session at any time. The system would have to listen for this, the activity should be terminated, and any resources used by it should be freed. Implementing a cancel button in the UI or allowing players to press a dedicated key to exit the game will improve usability.

5.2.2 Support system initiative

The system should anticipate user needs and provide helpful feedback without requiring user input. This can be accomplished by implementing the tactic of **maintaining a system model**. For Noodle.io, this model could include a leaderboard that tracks and displays the biggest noodles. By continuously updating the leaderboard, the system can provide players with feedback on their progress, improving usability.

5.3 Testability

Tactics for testability are intended to promote easier, more efficient, and more capable testing [1]. We will focus on two categories of testability tactics. The first focuses on improving controllability and observability, ensuring that the system can be easily monitored and manipulated during testing. The second focuses on limiting the complexity in the design of the system, making it easier to isolate and understand individual components.

5.3.1 Control and observe system state

Controlling and observing system state is an essential part of improving testability, and there are several tactics to ensure controllability and observability. In our case, we can implement the **Localize State Storage** tactic by centralizing the system state into a dedicated GameState class. This will ensure that the most critical game variables are grouped together in one location. Having a centralized state allows us to reset the game state efficiently before each test, ensuring consistent and reproducible test conditions. Additionally, we reduce dependencies between different components, improving modularity and testability.

5.3.2 Limit complexity

Because complex software is much more difficult to test, we will focus on **limiting the structural complexity** of our system. We will achieve this by avoiding cyclic dependencies between components and ensuring that each component has well-defined responsibilities. By reducing dependencies between different parts of the game, we can improve modularity, making the system more testable.

Additionally, we will keep our inheritance hierarchy shallow, limiting the number of parent and child classes, thus avoiding unnecessary complexity. Limiting polymorphism and dynamic calls may also be beneficial. We will also control the *response* of classes, ensuring that each class interacts with only a limited number of methods from other classes. By keeping this metric low, we can reduce coupling and improve testability.

5.4 Performance

The goal of performance tactics is to generate a response to events arriving at the system under some time-based or resource-based constraint [1]. In simpler terms, we want to implement tactics to minimize latency within our system. Latency is typically caused by resource limitations. To address this, we will use one main category of tactics: controlling resource demand.

5.4.1 Control resource demand

Controlling resource demand involves reducing the demand for important resources. This will improve both the performance and energy efficiency of our system. There are a few tactics we can implement to reduce resource demand:

- **Manage work requests:** We want to reduce the number of requests coming into our system. By implementing a service level agreement, we can limit the number of incoming events our system is willing to support. For example, if the leaderboard system is updated too frequently, we can impose a refresh limit (e.g. update every 5 seconds) to avoid excessive requests. However, we need to ensure that this limitation does not come at the expense of usability.
- **Prioritize events:** Because not all events are equally important, we can assign priorities to ensure that the most important events are handled first. For example, in our game, collision detection should be prioritized over updating visual elements. This way, we optimize performance while maintaining a smooth gameplay experience. However, lower-priority tasks should still be handled in a way that does not negatively impact usability.
- **Reduce computational overhead:** Optimizing how events are processed can improve performance. For example, minimizing levels of abstraction can reduce latency, such as allowing direct communication between game components instead of routing through intermediaries. However, this may conflict with our modifiability goals. We therefore need to find a balance between improving performance and maintaining modifiability in our system.

6 Design and Architectural Patterns

In this section, we will list the design and architectural patterns used, and describe how these impact architecture and quality attributes.

6.1 Architectural patterns

6.1.1 Model-View-Controller

We will implement the Model-View-Controller (MVC) architecture to separate concerns, making our system more modular. The model will handle game logic, such as noodle movement, collision detection and score tracking. The view will be responsible for the visual elements, such as displaying noodles, ingredients and animations. Finally, the controller will process user inputs and update the model accordingly. This separation of concerns will improve modifiability, allowing us to update the UI or game logic independently without affecting other components. Additionally, it accommodates usability, as the view can be modified quickly without touching the model or controller.

6.2 Design patterns

6.2.1 Singleton

We will use the Singleton pattern for managing global game states and shared resources, such as a game state manager. By ensuring that only one instance of this class exists, we can prevent redundant memory usage and simplify access to shared data throughout the game.

6.2.2 State

We will use the state pattern to manage different game states, where each state will be encapsulated in its own class, such as PlayState and MenuState. This improves modifiability, as adding or modifying states can be done without altering unrelated code. It also improves testability, as each state can be tested independently to ensure that it behaves correctly under different conditions. From a usability perspective, clear state transitions help maintain a responsive and predictable user experience. These benefits makes the state pattern a strong choice for managing game state efficiently.

6.2.3 Observer

The observer pattern enables us to automatically update various elements of the system when relevant game events occur. For instance, if a noodle reaches the maximum size and the game ends, all relevant components are immediately notified and updated accordingly. The observer pattern improves modifiability, as it allows subjects to notify observers while keeping them loosely coupled. It also improves usability by providing instant feedback to players when important events occur, such as a game-ending condition or a new high score. Testability also benefits from this pattern, as observers can be tested independently to verify that they correctly respond to state changes.

7 Architectural views

7.1 Logical view

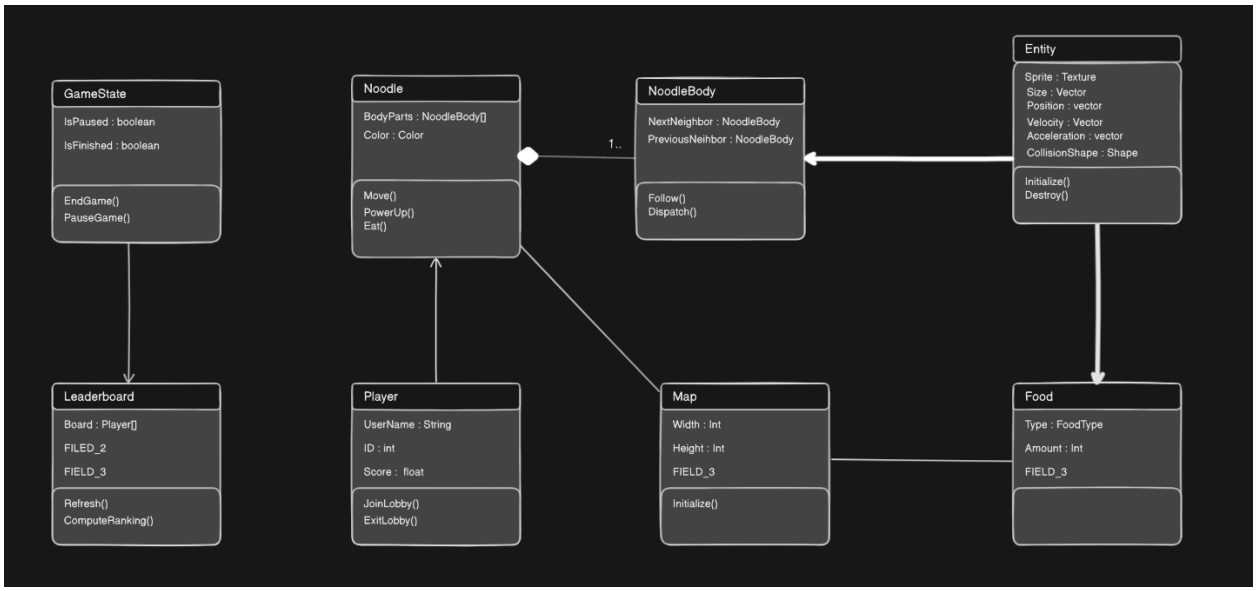

Figure 1: Class Diagram v1

7.2 Process view

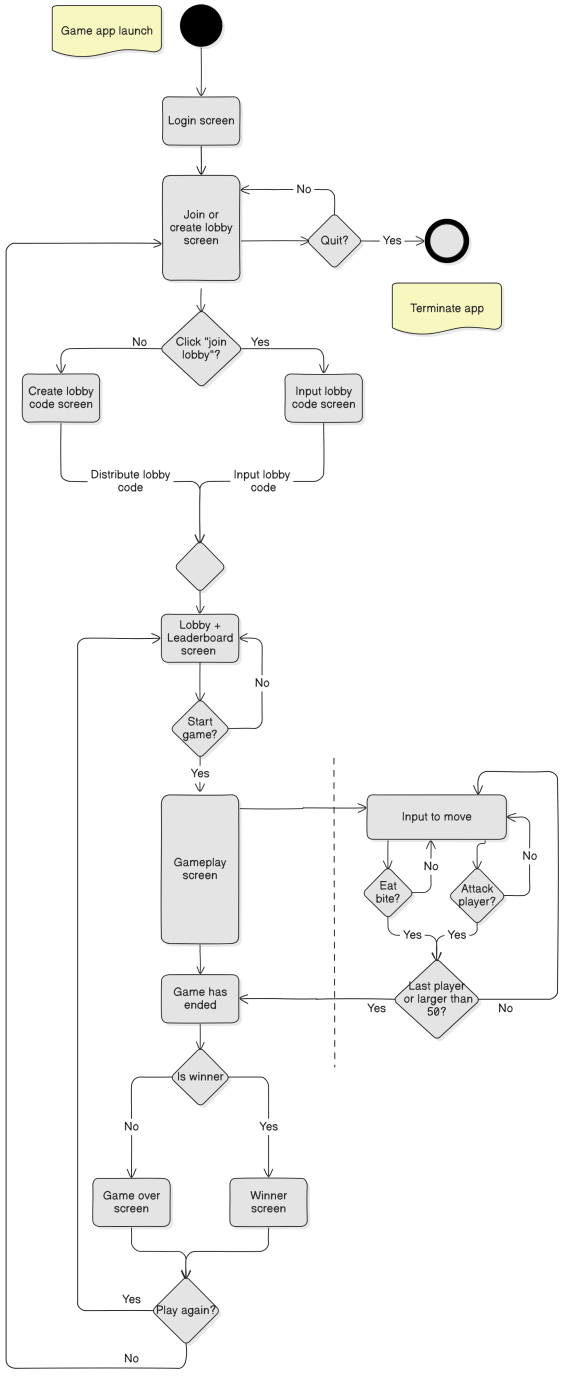

Figure 2: Activity diagram v3

7.3 Development view

The development diagram reflects which parts of the software can be developed in parallel, where all packages on the same level can be developed at the same time. The arrows pointing from one package to another indicates that the package is depending on the other package it is pointing at.

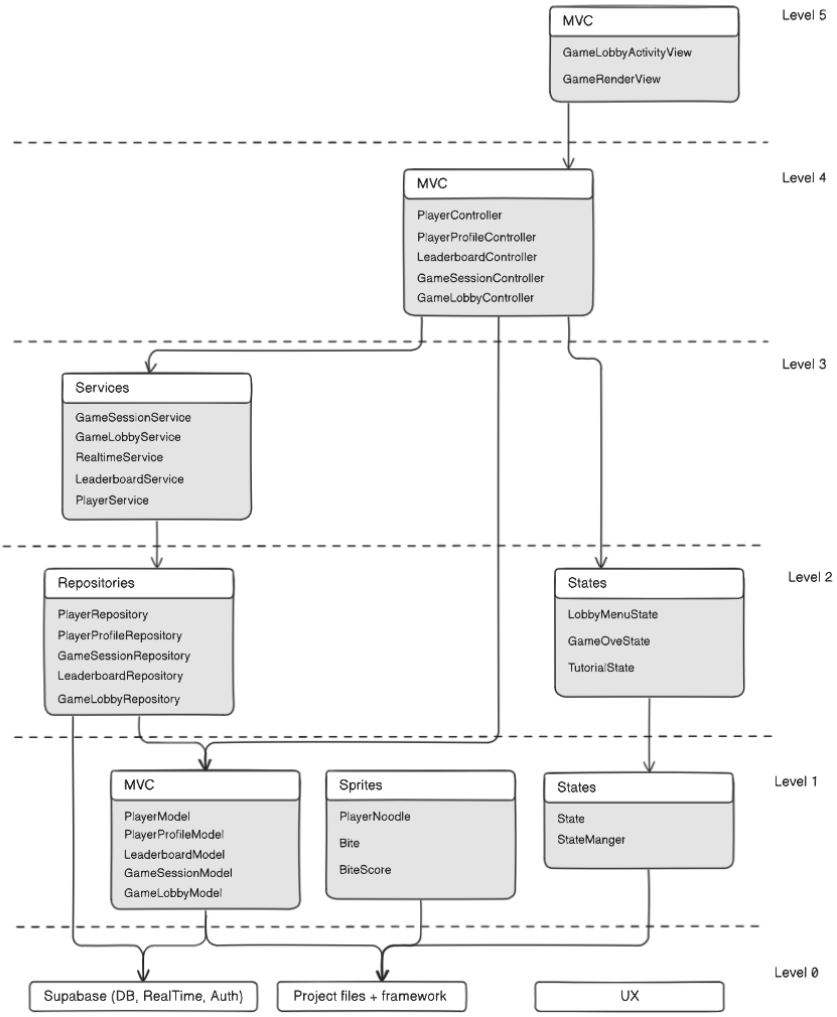

Figure 3: Developer diagram v4

7.4 Physical view

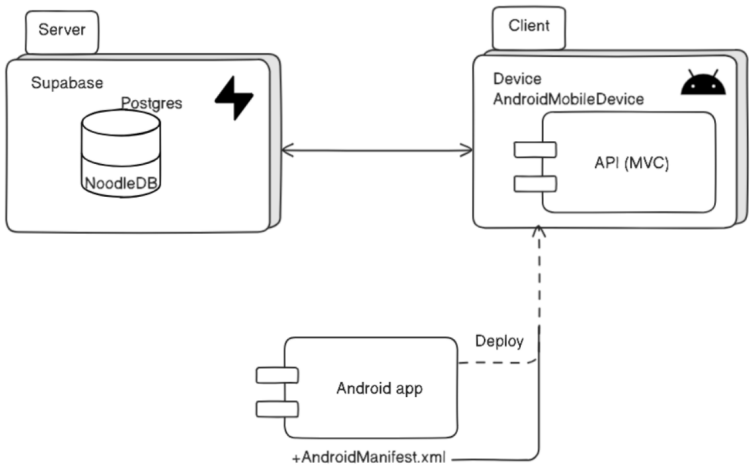

Figure 4: Deployment diagram v1

8 Consistency among architectural views

For this first deliverable we have not standardized naming of components, classes or entities and attributes in the view diagrams.

9 Architectural rationale

The architecture of Noodle.io was designed to ensure modifiability, usability, testability, and performance. Our decisions were guided by the need to balance these qualities without introducing unnecessary complexity.

The primary quality attribute of our project is modifiability, so we prioritized a highly modular design, focusing on increasing cohesion and reducing coupling across our system. This modular approach should be reflected in our core architectural pattern. At first, we considered implementing the entity-component-system (ECS) pattern, which is widely used in game development. However, ECS is most useful when managing large numbers of entities with overlapping traits, and our game primarily consists of distinct entities with relatively independent behavior. We therefore concluded that ECS would have introduced unnecessary complexity without significant benefits.

Instead, we opted for the model-view-controller (MVC) pattern, which provides a clear separation of concerns while keeping the implementation relatively simple. The model is responsible for handling the game state, including noodle movement, collision detection, and game rules. The view handles rendering of UI elements. The controller processes user inputs and updates the model accordingly. We think this structure will make the system more modular without introducing too much complexity. However, MVC still adds some complexity, which could introduce some overhead in performance. Despite this, we still consider MVC a strong choice for our architecture, due to the benefits of modularity.

To further improve modifiability, we decided on incorporating the observer pattern in our architecture due to its decoupled nature. This pattern allows different components to react to game events without being tightly coupled to one another. For instance, when a noodle is eliminated, multiple subsystems need to react. Instead of having direct dependencies between these components, the observer pattern enables them to subscribe to relevant events and update independently when

notified. This approach improves modifiability, as new features can be introduced without modifying existing components, and also improves testability, since individual systems can be tested in isolation.

Additionally, we chose the state pattern to manage different phases of the game, such as the menu, active gameplay, and game-over states. By encapsulating each state in a separate class, we avoid complex conditional logic and improve maintainability. This approach ensures that adding new game states requires minimal changes to other parts of the system, which improves modifiability.

The singleton pattern was chosen for specific cases, such as the game state manager, as it ensures a single, centralized instance that can be accessed globally without redundant memory allocation. However, we should be careful to limit our use of the singleton pattern to avoid excessive dependencies, which could hinder testability and modifiability.

To ensure good performance, we wanted to focus on implementing tactics to control resource demand. By managing work requests, we can prevent unnecessary processing overhead. For instance, limiting the update frequency of the leaderboard allows for providing feedback to players, while preventing excessive computations. Similarly, prioritizing critical events, such as collision detection, ensures that the most essential game mechanics work as intended, even under high system load. However, while these optimizations may improve performance, they must be balanced with usability. If updates are delayed too much, players may experience lag in visual feedback, negatively affecting the user experience. Likewise, reducing computational overhead by minimizing levels of abstraction can improve efficiency, but may come at the expense of modifiability. Therefore, we must evaluate each optimization to find a balance between performance, usability, and maintainability.

10 Issues

11 Changes

Bibliography

[1] Rick Kazman Len Bass Paul Clements. *Software Architecture in Practice, Fourth Edition.* Addison-Wesley, 2021. ISBN: 0136886094.

Appendix