TDT4240 - Software Architecture

Architectural Description

Avalanche Rush

Group 09

Members:

Nico Hannoun

Kacper Multan

Alicja Jonczyk

Arianna Moser

Margaux Xu

Gagan Gupta

Nicolò Costa

Chosen COTS:

Android Device

Android Studio

LibGDX

Firebase Database

Google Play Games Services

Primary quality attribute: Modifiability

Secondary quality attribute: Usability

Spring 2024

1 Introduction

1.1 Description of the project and this phase

The project involves creating a multiplayer game or fun app for Android with the following features:

1. Multiplayer Functionality
2. Online Functionality
3. Server Component
4. Complexity

The primary goal is to create a modifiable application, allowing for easy modification and addition of features.
The app must exhibit at least one secondary quality attribute. The chosen quality attributes must be reflected in the quality requirements, in the architectural design (tactics and patterns), the implementation, and in the final tests of the application.

The first phase is about addressing the background for this project, and the architecturally significant functional requirements. The intention of this document is to help the development team to determine how the app will be structured at the highest level.

1.2 Description of the game concept

Avalanche rush is an endless runner game, where the player has to flee from an avalanche while avoiding obstacles. It is a game similar to the *Subway Surfer's* game which we have inspired from but with a multiplayer mode option. In multiplayer mode, the aim is to be the last one still in the race. Players can push each other to get the power-ups. The game ends when one of the players hits an obstacle.

2 Architectural Drivers / Architecturally Significant Requirements (ASRs)

2.1 Functional

- **Game State Management**: This component manages the overall state of the game, including starting a new game, pausing, resuming, and ending the game. It also tracks the score and determines the winner in the multiplayer mode.

- **Multiplayer Networking**: This handles the communication between players in multiplayer mode. It synchronizes the player movements, actions, and game state for the different players. This helps us ensure smooth and fair multiplayer experiences.

- **Player Controller**: Each player will have a controller responsible for handling input from the user. This controller will interpret touch gestures or button presses to make the player move around and interact with other players / power-ups.

- **Obstacle Generation**: This will be responsible for randomly generating obstacles in the player's path. It should ensure the obstacles are placed at appropriate distances to maintain the game's difficulty level.

- **Power-up System**: This will manage the spawning and effect of power-ups. In multiplayer mode, it allows players to push each other to grab power-ups.

2.2 Quality

2.2.1 Modifiability

This is one of the main quality attributes of our game. It should be easy to add and modify power-ups, characters and obstacles.

2.2.2 Usability

This is the other main quality attribute. The interface should be simple to use. The rules of the game are easy to understand. It should also be easy to invite an opponent to a game.

2.3 Business

The main goal will be to meet all the project requirements and implement a game with a good level of completeness within the time constraint. We will aim to deliver satisfactory work for each phase within the deadlines. Therefore, we will focus on the main functionality with a good level of modifiability, so the game can be easily improved.

3 Stakeholders and Concerns

In this part of the project, we take into consideration the stakeholders and their concerns. We have identified mainly 3 of them (the course staff and the ATAM evaluators are two different stakeholders but having the same concerns, we have therefore decided to consider them as one in this phase).

Stakeholders	Concerns
Project Team	The goal is to create a game that is easily usable by end users and that respects the standards given during the requirements collection phase. Furthermore, the game must be easily maintainable and scalable
Course Staff and ATAM evaluators	The code must be written so that it can be easily read and commented on, even for those who did not directly participate in the writing. Furthermore, the documentation must be well written to make the evaluation process easier
End Users	They are the true and final recipients of our project, for them, the game must be easily usable, free of bugs and above all interesting

4 Architectural Viewpoints

Viewpoint	Purpose	Stakeholder	Notation
Logical View	Show what functionality that the system provides to end-users	Developers, Course Staff, ATAM evaluators, End-user	UML diagram, class diagram
Process View	Describes how the system processes communicate with each other during the run time behavior	Developers, Course Staff, ATAM evaluators	UML diagram, state diagram, activity diagram
Development View	Show development process of system: tools and technologies which have been used. Maps software components to file packages	Developers, Course Staff, ATAM evaluators	UML diagram, component diagram

Physical View	Describes mapping of software components to physical components and connections between physical components	Developers, Course Staff, ATAM evaluators	Client-server image

5 Architectural Tactics

5.1 Modifiability

To reduce the cost of changes made to a system, we use modifiability tactics. In our project, these are the ones we decided are most important.

Clear and well documented code:

Having a thorough documentation for our code and committing even the smallest changes will help us avoid errors and review our code in case of adverse alterations. We have also decided to define coding conventions to increase the readability of our code and avoid multiple implementations. The conventions we have chosen are:

- *Camelcase* naming of functions and variables
- Variable names that describe the variable's functionality and are intuitive to understand
- Comments describing the functions task
- Functions, constants, and variables should be declared in the correct places in the code file, making it intuitive to understand for every programmer working with the code

Best fitted size of modules:

To improve the modifiability of our code, we will separate it into modules. The goal of this separation is to make adding adjustments and changes to our game as easy as possible. This will lead to reduction of modification costs.

Code cohesion:

This part will also have great influence on the modifiability of our project. We want the modules that concern similar functionality to be consistent with each other, but still independent enough to make implementing changes easy.

Reduce coupling:
To reduce coupling between modules, we have decided to use these tactics:

1. **Encapsulation**: this tactic enforces information hiding and limits the ways other modules can interact with the original module. By restricting the access, we lower the probability that a change made to one module propagates to other modules.
2. **Intermediates**: introducing an intermediary layer between modules helps to decouple dependencies by abstracting away the knowledge modules possess of each other. This intermediary layer establishes a standardized method of interaction for all modules that need to access the same service from each other.
3. **Refactoring of code**: by refactoring the modules, we can augment the grouping of similar modules and reduce their dependence on the system.

5.2 Usability

The usability tactics focus on making the game as user-friendly as possible and implementing a support system that will help to achieve this goal. .

Removing unnecessary elements:
We have decided to keep the game's design as simple as possible. The interface's design as well as the number of control buttons, which the user can use to play the game, should be kept to a minimum. We aim to make the game easy to learn and fun to play, so we choose to avoid adding unnecessary distractions.

Testing:
Usability testing is also a very important part of our project. We want our game to have an interface fitted for every user. To achieve this, we will produce prototypes, test them and apply the necessary changes to improve the user's experience.

6 Design patterns and Architectural Patterns

6.1 Singleton

Many times in application, we need only one instance of a class. Singleton pattern assures that there will be only a single object of a given class. It also allows us to use objects which are supposed to be global easily through the whole system.

6.2 Client-server

In our game, we have decided to implement a client-server architecture instead of a peer-to-peer setup. The decision was made due to the necessity of ensuring that obstacles are generated in identical positions for all players. Therefore, we require an intermediary server to facilitate this synchronization between users.

6.3 Factory pattern

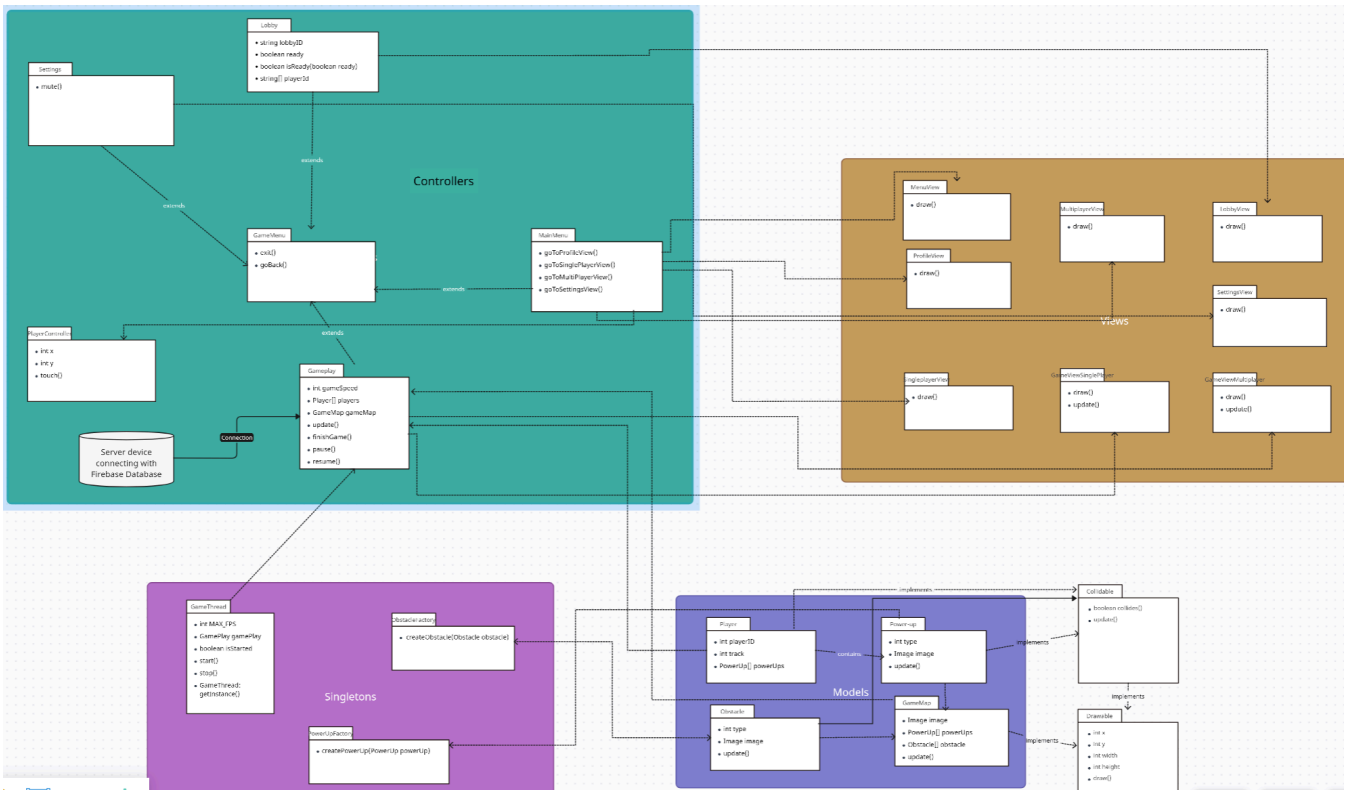
Factory pattern is used to create factories of similar objects which will be used for easier generation of objects during the runtime. For example, we have different obstacles that share some of the same components with each other and will be generated during the game.

6.4 Model-View-Controller

The game will utilize a Model class to manage components and game states, a Controller to handle game logic, and a View to manage appearance. Controller will work as an intermediary between Model and View classes.

7 Architectural views

7.1 Logical view



The logical view is split into Controllers, Models and Views according to the MVC pattern structure of the application. Singletons, due to their global accessibility throughout the application, are managed separately from the MVC components.

Controllers are mostly used to control game flow and navigation between different views.

The MainMenu serves as the initial point of entry into the application, allowing users to navigate to different views. The GameMenu superclass provides common functions like exit() and goBack() across multiple controllers.
PlayerController interacts directly with users during gameplay, handling input and updating player positions.
The Gameplay controller serves as a bridge between models and in-game views, managing game state and flow.

Our game consists of 8 distinct views. The MenuView serves as the gateway to the application, offering routes to SettingsView, ProfileView, and various game initiation views. We distinguish SinglePlayerView and MultiplayerView to accommodate the different setups for starting single and multiplayer games. LobbyView is used for inviting another player to a game and waiting for him. Also, GameViewSinglePlayer and GameViewMultiplayer are separated because in multiplayer mode there are additional tracks during the game.

There are 2 interfaces which are used for containing position and size of objects on screen. Drawable interface allows drawing an object on screen and Collidable interface is an extension of it which adds possibility of collision between objects which use this interface. Most models, except for GameMap, implement Collidable to reduce redundancy in code and increase adaptability. The GameMap model specifically renders the game environment, including obstacles and power-ups.

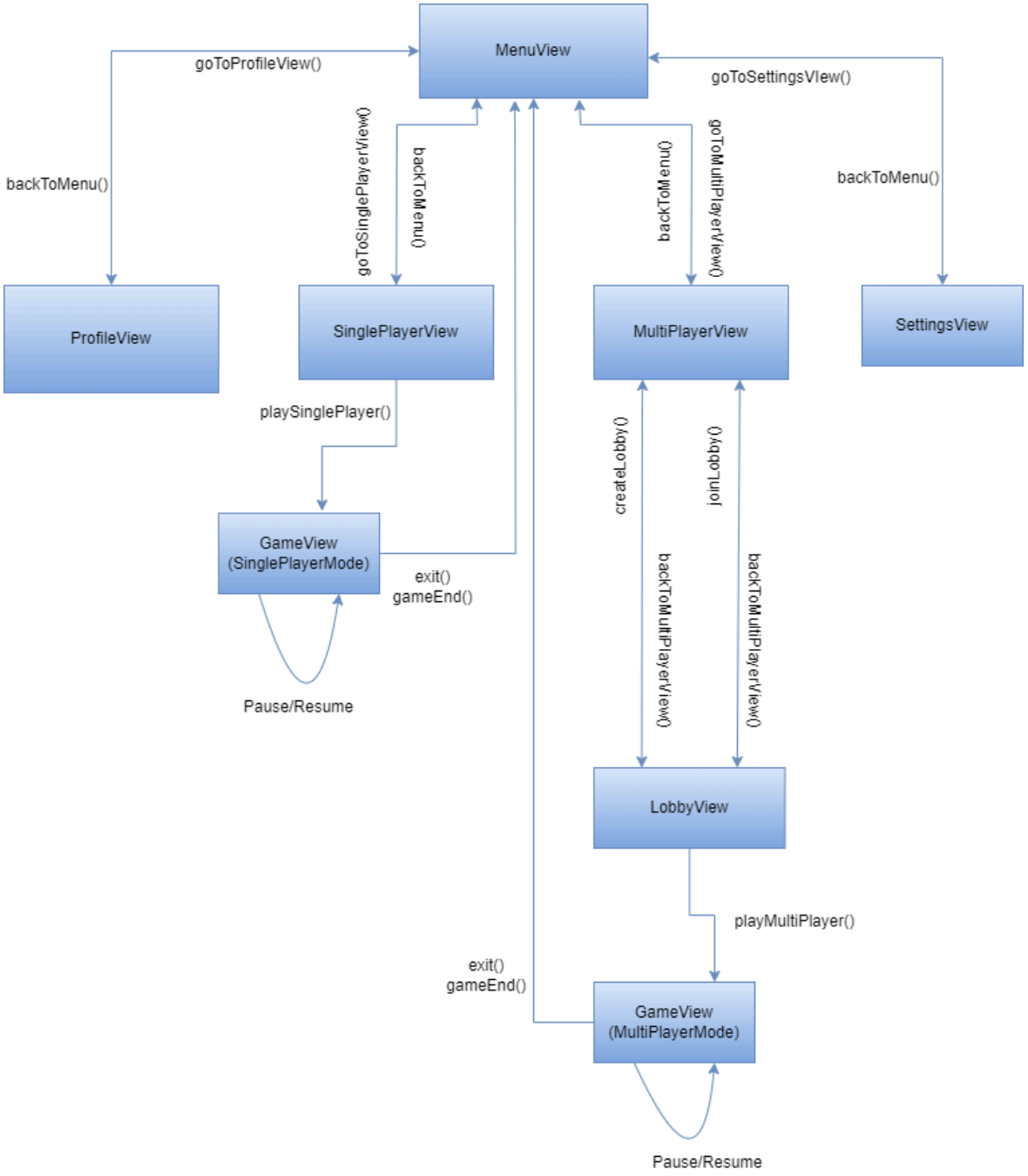
GameThread is a main class of the app, on which the whole game is running when the player is playing a game. It relies on the Gameplay class to initiate and terminate games, with a predefined maximum frame rate ensuring consistent performance across players.

Factory pattern is implemented through ObstacleFactory and PowerUpFactory classes, which are used to create new objects. Those factories increase performance by creating new objects during runtime. These classes operate as Singletons to maintain consistency in object creation, especially in multiplayer scenarios.

Client-Server architecture is realized by usage of a third device or web server that connects with Firebase Database for storing data and sustaining the same game state between players. This architecture ensures data persistence and synchronization, enabling seamless gameplay experiences between players.

7.2 Process view

7.2.1 View / Activity state machine



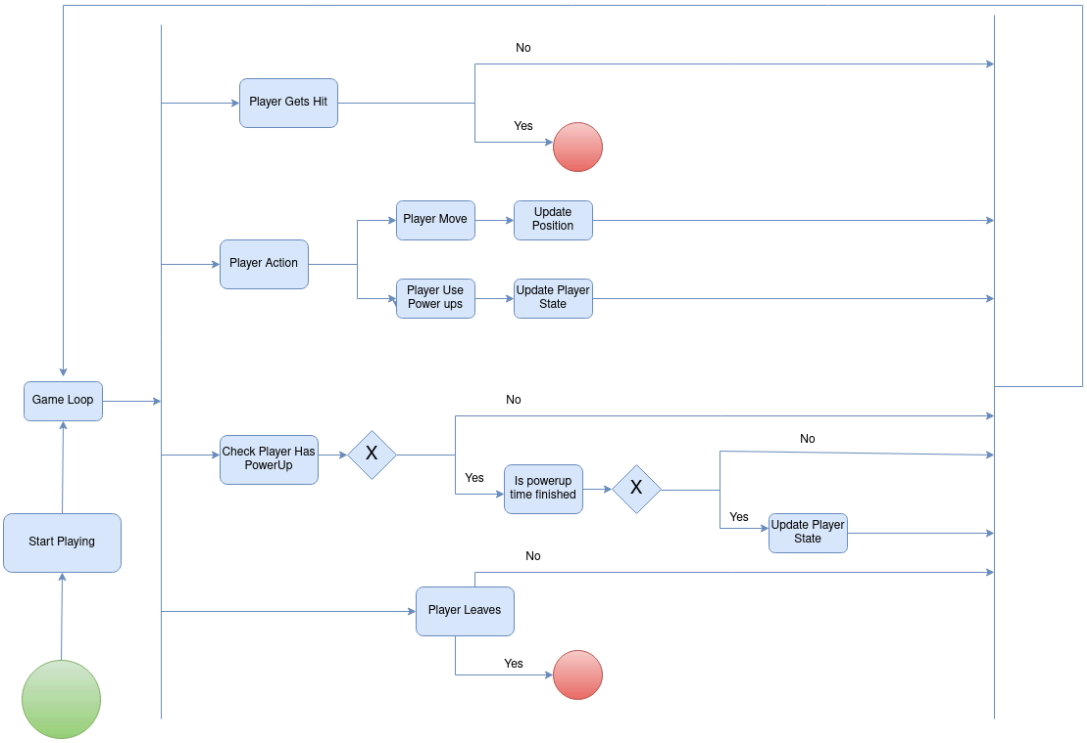
The state machine above describes the flow of the application in terms of processes run, giving an indication of how the application switches between its views.

Initially, once the application is started, the user finds himself in the MenuView. From here he can access the ProfileView (goToProfileView()) where he can check

his personal information including his user id, similarly he can access the SettingsView (goToSettingsView()).
Also, from here he can access the SinglePlayerView (goToSinglePlayerView()) or the MultiPlayerView (goToMultiPlayerView()).

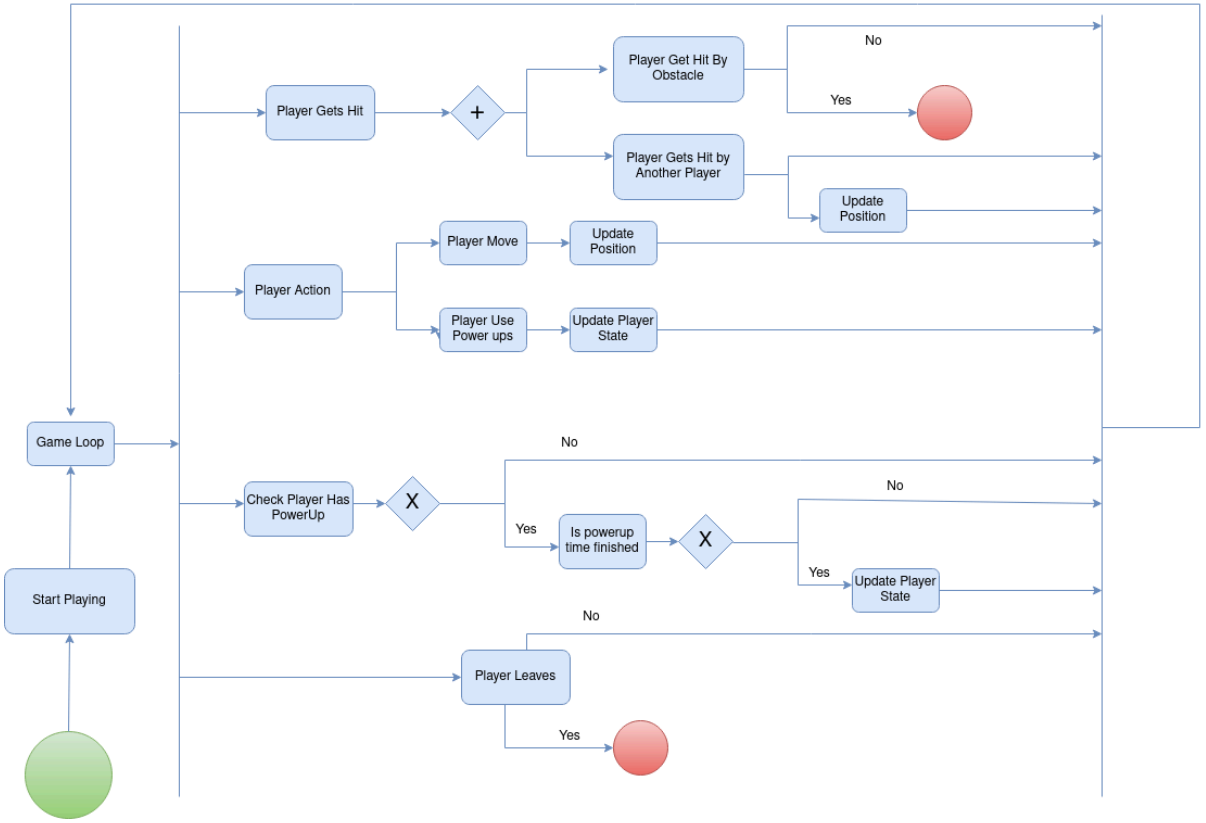
In the first case by pressing the 'Play' button (playSinglePlayer()) it is possible to start a game and thus move to GameView (SinglePlayerMode) where through the 'Pause' button it is possible to pause the game for an unlimited time.
In the second case, through the 'Create' (createLobby()) button, it is possible to create a lobby thus accessing the LobbyView or by pressing 'Join' (joinLobby()) it is possible to access the LobbyView but first inserting the ID of an already existing lobby, at this point pressing 'Play' (playMultiPlayer()) starts a game in multiplayer mode and you switch to GameView (Multiplayer Mode), here too it is possible to pause the game using the 'Pause' button but only for a limited time.
In both cases it is possible, once the game has been paused, to continue it using the 'Resume' button or to return to the MenuView by pressing the 'Quit' button (exit() and endGame()).

Finally, from the ProfileView, SinglePlayerView, MultiPlayerView and Settings it is possible to return to the MenuView (backToMenu()) and in the same way from the LobbyView to the MultiPlayerView (backToMultiPlayerView()).



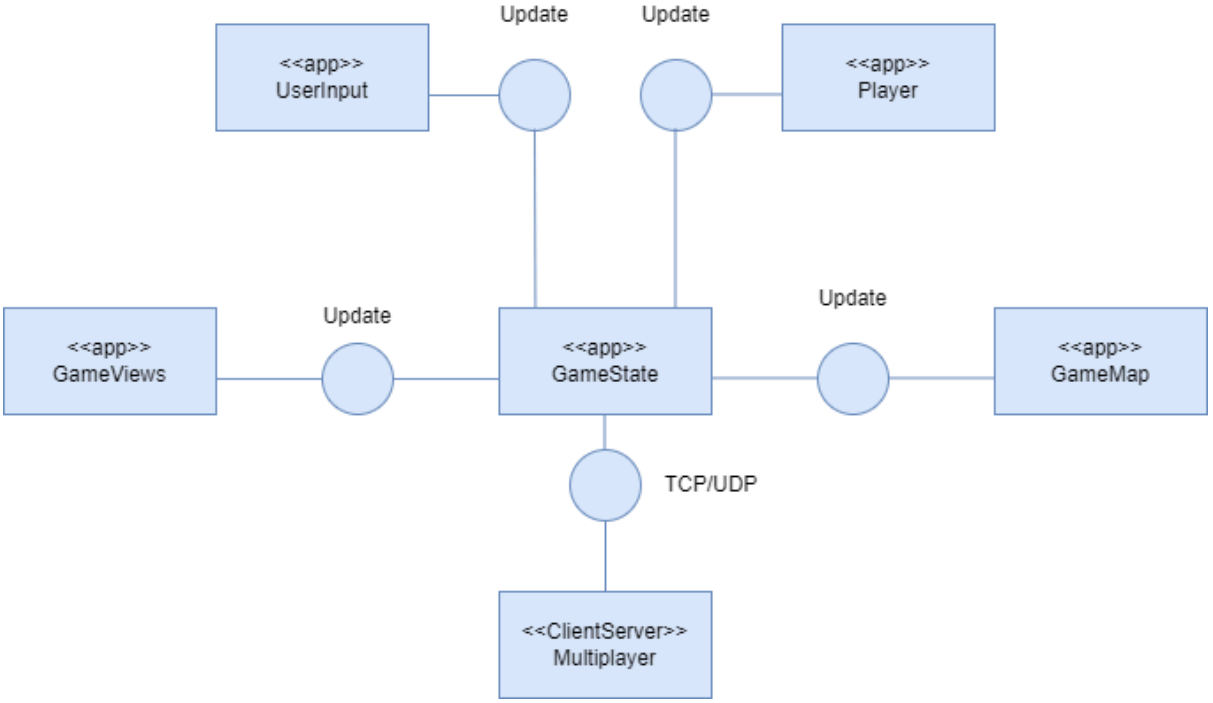
The diagram above describes a game loop and all the possibilities for the player in the game. This is for the single player mode.

After the player starts a game, we enter a game loop. The player can either move or use a power-up, it would then update the player's position or its state. Once the power-up time is finished, it updates the player state again. The player can also get hit by an obstacle, which would lead to a game over (end state).
The player can leave the game whenever they want to (end state).
All the states between the two bars lead back to the game loop until we reach one of the end states.



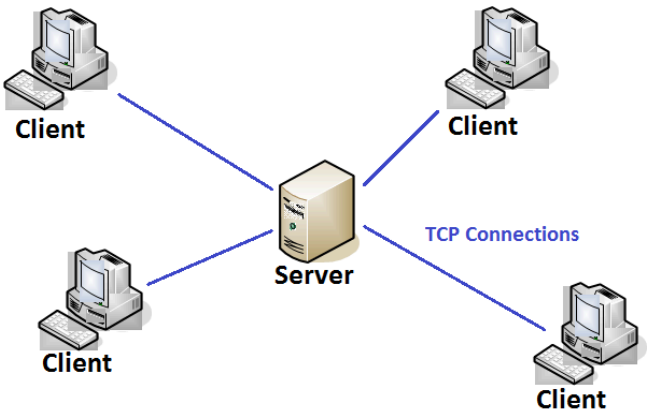
This represents the game loop for the multiplayer mode. It is almost the same as the one for single player mode, except that there are two different possibilities when one of the players gets hits.
They can either get hit by an obstacle, which would lead to the other player victory (end state) or they can get "pushed" by the other player, then it will update their position.
Everything else is the same as for the single player mode.

7.3 Development view



The component diagram shows the interaction between the components of the application, giving a clear overview for which parts should be developed independently.
We are establishing a client server that will be responsible for synchronizing the game states of the multiple players and generation of the obstacles and power ups. The game state will fetch the updates from the client server, which then will be reflected on the game views and game map. Also, game state will fetch the user inputs from the player's device and update the player and reflect those changes in the client server.

7.4 Physical view



source

Hardware structure consists of two player devices connected with each other through a client server which works as an intermediary server between them. All of the game activities of each player are going to be sent through the server to the second player. Also, our client server is responsible for the generation of the obstacles/power ups and synchronization of both players at real time.

Single player mode will only need one player device because no server communication is needed. In single player mode, the player's own device will be solely responsible for all the game logic and implementation of it.

8 Consistency among architectural views

In the Logical View we define a GameMenu package, but in the Process View we do not explicitly define a GameMenuView because we c9onsider it part of the GameView. Furthermore, the component diagram in the Development view can be redundant because it recalls exactly what is defined in the Logical View.
Finally, in the Physical View there are four clients in the image, when in reality the maximum number we can have is two.

9 Architectural Rationale

After days of planning and evaluating the various factors that affect the project's final layout, we chose the following architecture because we think we can fulfill the specific requirements effectively.

In order to guarantee that there won't be multiple instances, we chose to use Singleton Pattern. We are going to create some manager classes to handle global states or functionalities that need to be accessed from various parts of the game without creating multiple instances.

In our game, players compete against each other in real-time, requiring synchronization of game states between different devices. For this purpose, we are going to use the Client-Server pattern. For example, we need the obstacles to be generated in the same position for all the players. Thus, the server handles game logic, manages player interactions, and validates actions to maintain fairness across all clients.

The factory pattern is useful when we have to create multiple smaller objects that share the same properties. A factory function can easily return a custom object depending on the current environment, or user-specific configuration. For example, in our game we are going to generate different types of obstacles like rocks and trees.

Finally, to write a code easier to understand and maintain, we are going to create: a Model component to manage player score and behaviors, a View component to manage the visual effects of the game and a Controller component to manage player input and update the game state accordingly.

10 Issues

We had no issues that are suitable for this section.

11 Changes

This section lists change history for this document.

Date	Change history	Comments
February 26, 2024	First released version	None

12 References

- Subway Surfers: Play Subway Surfers Online for Free on PC & Mobile, [online] https://subway-surfers.org/, last accessed 20.02.2024.
- General knowledge about Norway, acquired in the first month being in Norway
- Example of final delivery group 11 (Blackboard project section)
- Example of a final project delivery (group 2) (Blackboard project section)
- Len Bass, Paul Clements, Rick Kazman, “Software Architecture in Practice – Third edition”, Addison Wesley, September 2012

13 Individual Contribution

For this first phase of the project, where it was necessary to collect everyone's ideas and opinions, it was not possible to specifically define everyone's contribution.

Content	Contributors
1 Introduction	Everyone
2 Architectural Drivers / ASRs	Gagan Gupta, Margaux Xu

3 Stakeholders and Concerns	Nicolò Costa
4 Architectural Viewpoints	Kacper Multan
5 Architectural Tactics	Alicja Jonczyk
6 Design and Architectural Patterns	Kacper Multan
7 Architectural Views	Everyone
8 Consistency among architectural views	Everyone
9 Architectural Rationale	Arianna Moser