NTNU
Kunnskap for en bedre verden

Department of computer science

TDT4240 - Software architecture

Implementation Document

Group 04 - Besieged

Authors:
Jens Martin Norheim Berget, Magnus Vesterøy Bryne, Sverre Nystad, Mattias Tofte, Tim Matras and Tobias Fremming.

Chosen COTS: Android SDK, LibGDX and Firebase
Primary Quality Attribute: Modifiability
Secondary Quality Attributes: Usability, Performance and Availability

21.04.2024

Table of Contents

List of Figures ii

List of Tables iii

1 Introduction 1

2 Design and Implementation Details 1

2.1 General overview of how the game runs when starting either a singleplayer or multiplayer-match . . . 1

2.2 A day in the life of the ECS-system . . . 2

2.3 Class Diagram . . . 3

2.4 Patterns . . . 4

2.4.1 Entity Component System (ECS) . . . 5

2.4.2 Singleton . . . 5

2.4.3 Factory . . . 5

2.4.4 Object Pooling . . . 5

2.4.5 State . . . 5

2.4.6 Observer . . . 5

2.4.7 DAO . . . 5

2.4.8 Game Loop . . . 5

2.4.9 Client-Server . . . 6

2.4.10 Facades . . . 6

2.5 Description of Core classes . . . 6

2.5.1 Clock . . . 6

2.5.2 ECS (Entity Component System) . . . 6

2.5.3 GameApp . . . 8

2.5.4 Game Client . . . 9

2.5.5 Game Server . . . 10

2.5.6 Graphics . . . 11

2.5.7 Input . . . 11

2.5.8 Launcher . . . 11

2.5.9 Math . . . 11

2.5.10 Networking . . . 11

2.5.11 Sound . . . 12

3 User Manual **12**

3.1 Installation . . . 12

3.1.1 Prerequisites . . . 12

3.2 Starting the application . . . 13

3.2.1 Starting on Android . . . 13

3.2.2 Starting on Desktop . . . 14

3.3 Screens . . . 14

3.3.1 Tutorial . . . 14

3.4 Towers . . . 18

3.5 Enemies . . . 19

4 Test Report **20**

4.1 Functional Requirement Tests . . . 20

4.2 Quality Requirement Tests . . . 25

4.3 Usability Tests . . . 25

4.4 Performance Tests . . . 26

4.5 Availability Tests . . . 27

4.6 Modifiability Tests . . . 28

5 Relationship with Architecture **30**

5.1 Changes made . . . 30

5.2 Performance Tactics . . . 31

5.2.1 Object Pooling . . . 31

6 Problems, Issues and Points Learned **31**

7 Conclusion **32**

8 Individual Contribution **34**

List of Figures

1 Android studio . . . 13

2 Main Menu . . . 14

3 Tutorial screen: When clicked on tutorial in main menu, you will be greeted with a tutoial on how to play the game. . . . 15

4 The Options Menu: Here, you can adjust the volume. . . . 15

5 Multiplayer Menu: Join an existing game, or host your own game, where you can play with a friend. . . . 15

6 Highscore Menu: Here you can see the leaderboard of the 5 best players. Are you good enough to beat the highscore? 16

7 Join Game Menu: Here you can view all games that are available to join 16

8 Map Choice Menu: Choose between two available maps 16

9 Gameplay: Buy cards, build towers and have fun! 17

10 Game Over: If you run out of health, you reach the Game Over screen. 17

11 Lost Connection: If the host disconnects mid-game, the client is sent to the Lost Connection screen. 17

12 Various bow towers: from left to right; bow and lightning, bow and fire, bow and bow, bow and magic 18

13 Various magic towers: from left to right; magic and lightning (Tor, the god of thunder), magic and technology, magic and magic (a witch weaving the reality from Norse mythology), magic and fire (the witch has experimented with fire magic, and became a fire demon). 18

14 Various tech towers: from left to right; technology and lightning, technology and technology, technology and magic, technology and fire. 19

15 Various enemies 19

16 Productivity by release. Messy code reduces the productivity 33

List of Tables

1 Individual Contribution 34

1 Introduction

In this phase of the project, we have implemented our software architecture by creating our game, Besieged. This process has taught us a lot about the process of planning, designing, implementing and testing a software architecture for a large application. Our architecture remained mostly unchanged throughout the development process, which illustrates how our architectural patterns were appropriately picked during the planning stage.

Besieged, inspired by Norse mythology, is a tower defense game similar to Bloons TD 4. The player has five cards of different types available to them, which they can purchase and place on the map. To create a tower, two cards must be placed on top of each other. The types of cards you combine dictate the type of tower you receive.

Enemies spawn in waves at one end of the path and move across the map to the other side. If they successfully make it through, they will damage your village. Your goal is to place towers to attack and kill these enemies before they get to your village and destroy it, and survive as long as possible.

Besieged has 15 unique hand-drawn towers and 8 enemies with fully-fledged animations. Currently, we just have two maps, but we can easily add more. The game also has a custom-designed Viking-themed user interface with menus, cards and map textures all made by our team.

The game also has an online real-time multiplayer mode where two players can cooperate on defending their village by placing cards and towers simultaneously. In this game mode both players share the same health and money pool, acting like one combined village.

2 Design and Implementation Details

This section will describe the control-flow of the application by explaining how two large parts of the application function in great detail.

2.1 General overview of how the game runs when starting either a singleplayer or multiplayer-match

1. The GameApp, which is the entry point of the application, creates the GameLauncher.
2. The GameLauncher creates all the different instances of the necessary interfaces that are needed to run the application like drawing, audio and input. It also creates the Data Access Objects (DAOs) for the client and server. It also creates the GameClient itself.
3. The GameClient tells the ScreenManager to initialize and enter the Menu state, i.e. the main menu.
4. The Menu-state instantiates the required ECS-systems like the InputSystem, RenderingSystem, AudioSystem.
5. The GameApp starts its update-method which updates the GameClient continuously, which in turn updates the ECS-system. Review the next subsection to get a better deeper understand of the update-cycle in the ECS.
6. After each iteration of the ECS update-cycle, the GameClient checks if it has joined a game or not.
7. User navigates through the multiplayer-menu to the "Choose map"-menu and chooses a map.
8. After choosing a map, the server is started on a separate thread. Depending on if the game is a singleplayer or a multiplayer-game, the Server and Client get either a LocalDAO or a FirebaseDAO, respectively.

9. The server continuously polls to check if there is a pending player waiting to join the game.
10. A client notifies the server that it wants to join the game. This happens in the "Join Game"-menu by pressing an available game. It then waits a short period before checking for a response from the server.
11. The server handles the join request and puts the player hosting the game into the game itself, i.e. the InGame-state. This also happens to the joining player.
12. The server then initializes the game itself by creating entities and adding them to the ECS-system, and updating all clients.
13. During each update-cycle in the Server while In-game, the following things happen: The ECS-systems are updated, progressing the game. It then checks if the game has ended or not. Then, it looks for pending actions received from players, and handles them if they are valid, like placing a card on an available tile if the player has enough money to buy it. At the end of each update-cycle the new updated GameState is applied and sent to all clients.
14. Repeat main update-cycle until the game is over, or the host terminates the application or loses connection.
15. Clients continuously pull updates to the GameState from the server and updates their local ECS-systems.
16. Clients send request for doing actions to the server.
17. The game ends in one of the two following ways: Case 1. The server abruptly loses connection. If a client does not receive new updates from the server within 10 seconds, it exits to the "lost connection"-screen. Case 2: The game ends with the players losing all their health, and the "Game Over"-screen is displayed. The server does a teardown-procedure which include updating the global highscore, and removes the game from the list of pollable games.

To clarify the Client and Server do not have direct contact with each other., but are actually communicating through a common database. When trying to join a game, a client posts a row with the game-ID and a prefix with the corresponding value being the clients player-ID. The server then looks for this row and adds the player-ID of the second player to the row in the database. The only piece of information that both players need to join a game is the Game-ID, which is publicly accessible in the database.

2.2 A day in the life of the ECS-system

Here's a general overview of how the Entity Component System (ECS) works.

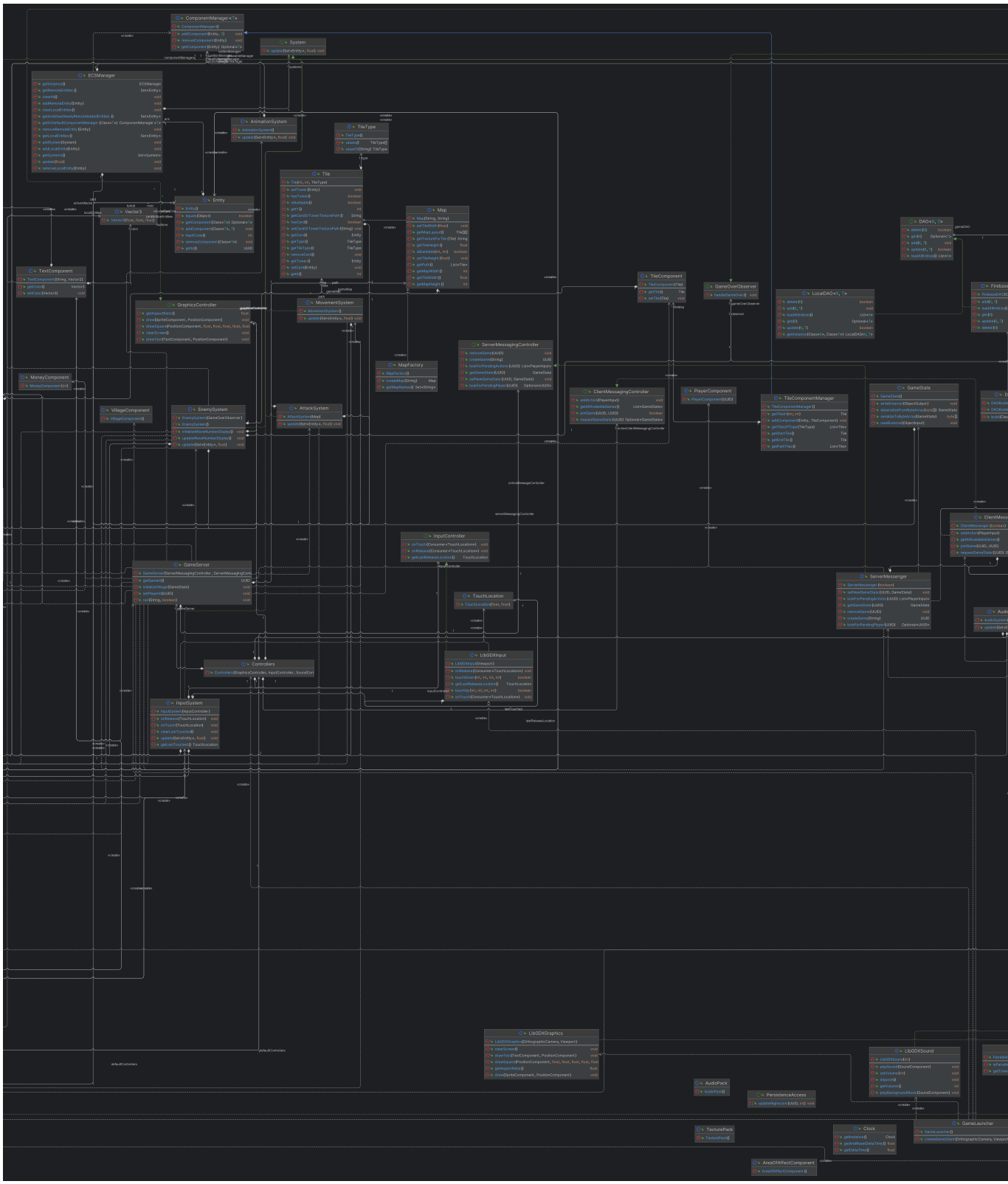
1. An entity is created somewhere.
2. Components for the entity are created and added to it. This is done indirectly through the entity, but the components live in `ComponentManager`s in the `ECSManager`.
3. The entity is added to the ECS.
4. Systems are added to the ECS.
5. The ECS' update function gets called with how much time has passed since the last update call as an argument, also known as deltatime.
   (a) If any entities from previous updates are to be added they are now added before the system runs through all the entities.
   (b) If any entities are to be removed from a previous update, they are now removed.
   (c) The Update-method, with deltatime as an argument, is called on all the systems in the ECS-system. The systems continuously loop through all the entities and mutate them.

6. At some point, when switching to a different State, the ECS' entities, component managers, and systems are cleared, and new ones are added.

We also have a list of *remote entities* in the `ECSManager` that contains entities and components that are not handled by the current thread/client. This was done to separate the *local entities* and their components from the entities and components given by the server.

2.3 Class Diagram

The following figure is the entire core module for our application. We removed other modules like the Android and Desktop as these are not the core logic of our application. Our project is so large that a full class-diagram would not fit and we did not a satisfying to showing the diagram split in two. We have tried to describe the structure of the application more detailed below.



2.4 Patterns

Patterns are good solutions to common problems. We have implemented the following ones in the architecture of Besieged:

2.4.1 Entity Component System (ECS)

The ECS pattern has been implemented in the `ecs` package (folder). It makes it easier for us to keep track of entities, components, and systems to enable a decoupled and flexible game or application architecture.

2.4.2 Singleton

In our application we use Singletons for the `ECSManager`, the `ScreenManger`, the `LocalDAO`, and the `Clock`. We needed to have a single source of truth, for how much time had gone, or what was in the local database and for the state of the game. The singletons gives us global access to these.

2.4.3 Factory

Our application has several factories like the `DAOFactory` making us change the database implementation based on the application configuration, as well as the `EnemyFactory`, `MapFactory`, `CardFactory`, `TowerFactory` and `ButtonFactory`. The factories make it very easy to expand the game by adding new content.

2.4.4 Object Pooling

To improve the performance of the game we used the Object Pooling pattern to reuse Entities. We use this in the EnemySystem for reusing dead enemies rather then having to allocate new memory to create new ones.

2.4.5 State

The State Pattern is central to Besieged as it is used to determine what page/menu is displayed on the screen. Each State determines what systems the ECS should have and how the different States change between each other. The ScreenManager holds the current menu.

2.4.6 Observer

Observers are used in various places of Besieged's architecture, like providing callbacks to buttons so that they lead to the necessary actions.

2.4.7 DAO

The Data Access Object makes the core logic of the application decoupled from the database system being used. This makes it easy to change the database implementation, making our game even more modifiable.

2.4.8 Game Loop

Since the frametime of the game is not necessarily constant, the Game Loop pattern decouples the game's state from time and processor speed Nystrom 2024. This makes the game feel more natural and makes the game progress at the same pace for everyone. We have used this in the `GameClient` and `GameServer` classes to check how much time has passed since the last update.

2.4.9 Client-Server

To make the game multi-player as well as flexible we have used the Client-Server architecture. The server also acts as a single source of truth in addition to being responsible for maintaining the rules of the game. The `game_client` package contains the client side of the game and the `game_server` package contains the server side.

2.4.10 Facades

We used the facade pattern to hide information about libraries and frameworks such as LibGDX and Firebase to make sure that we only use what we need. We also used it to abstract away the COTS, so that our code stays highly modifiable. Most of the facades can be found in graphics, inputs and sound packages. Abstractions of these facades are in the ecs, game_client, and game_server packages, in order to satisfy the Dependency Inversion Principle.

2.5 Description of Core classes

The following is a complete list of all the classes and their responsibilities and implementation details.

2.5.1 Clock

Clock.java - manages time tracking within the application, providing a centralized method to monitor and update various systems in the ECS.

2.5.2 ECS (Entity Component System)

The ECS module contains all Components, Entities, and helper classes needed for this Architectural Design Pattern. It also contains all the necessary controllers like Graphics, Audio and InputController.

Card.java - represents the abstract base for card types in the game, incorporating core components that dictate its placement, cost, and visual representation.

ComponentManager.java - manages components of a specific type for entities in the ECS. The ComponentManager allows for the addition, retrieval, and removal of components associated with entities, using the entity's UUID as the key.

Components The ECS system in our game architecture includes multiple components that manage various gameplay and entity behaviors. These components store data related to game aspects such as health, position, and AI state. Each Entity's components are managed and modified by their respective systems.

- **AnimationComponent.java** - manages frame-based animations for entities, storing animation paths and controlling frame transitions based on time intervals.
- **AreaOfEffectComponent.java** - enables all tower-entities to inflict damage on all enemies within its attack range.
- **ButtonComponent.java** - manages interactive button functionalities within the game, defining button properties such as position (UV offset), size (UV dimensions), and layer (z-index). Each button also contains a Callback that gets activated when the button is clicked.

- **CardHolderComponent.java** - holds a list of Card-objects. It represent a collection of cards in the game, i.e. the players' deck
- **CostComponent.java** - represents the cost of an Entity in the game, i.e. a card. The component ensures that the cost is a non-negative value.
- **EnemyComponent.java** - represents an enemy entity in a game. Holds information about the enemy's damage, whether a reward has been claimed for defeating it, and whether it is dead.
- **HealthComponent.java** - keeps track of the amount of health an Entity has, either the Player or enemies.
- **MoneyComponent.java** - keeps track of the players' money as well as the reward the player should receive for killing an enemy of a specific type.
- **PathfindingComponent.java** - facilitates Entity navigation by managing a list of Tile objects that form a path to a target destination. This enables entities equipped with this component to follow predetermined routes within the game.
- **PlacedCardComponent.java** - represents a placed card in a game. Holds information about the type of card and a sound-related field.
- **PlayerComponent.java** - represents a player in a game. Holds a unique identifier (UUID) for the player, allowing for distinct identification.
- **PositionComponent.java** - represents the position of an Entity. Holds a 2D vector for the position in UV-coordinates and an integer for the z-index.
- **SoundComponent.java** - manages the playback of a sound. Holds information about the sound file path, whether the sound should loop and whether it is currently playing.
- **SpriteComponent.java** - contains the visual representation of an Entity on the screen. Holds the path to the texture file and the size of the texture to be displayed.
- **TextComponent.java** - represents a text Entity. Holds the text string, the font scale, and the font color.
- **TileComponent.java** - represents a single tile Entity which is what the map consists of (i.e. the squares that make up the map)
- **TowerComponent.java** - represents a tower Entity. Holds information about the tower's type, damage, range, attack cooldown, and the time since its last attack.
- **VelocityComponent.java** - represents the velocity of an Entity. Holds a float value for the current velocity and a base velocity.
- **VillageComponent.java** - marker component to track the player's base. Has no content.

ECSManager.java - central coordinator for the Entity Component System. It facilitates the handling of entities, components, and systems to enable a decoupled and flexible game architecture. Uses the Singleton Pattern to ensure that a single instance manages all entities.

Entity.java - represents individual entities in our Entity Component System. Each Entity has a unique ID as well as an arbitrary number of components.

GraphicsController.java - responsible for drawing sprites and text, clearing the screen, and other graphics utilities such as drawing shapes or querying screen properties like the width and height.

InputController.java - has callback-functions that get triggered when touch events are triggered so that touch locations are properly registered.

SoundController.java - triggers specific sounds linked to game events, manages background-music, and has methods for adjusting the volume.

System.java - establishes a framework for systems within the ECS architecture. It requires the different systems to implement an update-method that processes entities, focusing on specific game logic or rendering aspects. Systems operate on entities that have specific components, applying logic to them. The update method is designed to be invoked once per update cycle, enabling systems to manipulate entities and respond to changes based on the elapsed time since the last update.

Systems

- **AnimationSystem.java** - fetches all entities that have an AnimationComponent and updates their SpriteComponnet with the next frame in the animation once per update-cycle.
- **AttackSystem.java** - fetches all enemy and tower-entities, and makes sure that towers attack enemies if they are within range. Checks if towers have an AreaOfEffectComponent, and adjusts the attack-logic accordingly. If one is present, the tower attacks all enemies within its range simultaneously. If not, it only targets a single enemy at a time.
- **AudioSystem.java** - fetches all entities that have a SoundComponent and contains logic for playing the correct sounds depending on the type of entity like cards, towers and enemies. Also manages the playback of background-music.
- **EnemySystem.java** - responsible for dynamically spawning enemies, managing their removal when they die or reach the path's endpoint, and the updating of game states based on these interactions. This system also ensures that enemies that reach the end of the path inflict damage on the player, potentially leading to the player losing the game and the game over-screen being displayed.
- **GameOverSystem.java** - responsible for monitoring the health of the village entity and triggering a game over condition when its health drops to zero or below.
- **InputSystem.java** - captures touch inputs, determines if these touches correspond to button presses, and triggers the respective actions associated with those buttons.
- **MovementSystem.java** - processes entities that possess position, velocity, and pathfinding components, calculating and updating their positions based on their current velocities and the paths defined in their pathfinding components.
- **RenderingSystem.java** - renders visual components of entities within the game, such as sprites and text, based on their positional data and visibility settings.

TouchLocation.java - used by the InputSystem to keep track of where the player has touched the screen.

2.5.3 GameApp

GameApp.java - initializes a game client, camera, and viewport upon creation, setting up an environment with a defined screen size. It updates the game client in the render loop and handles screen resizing by maintaining a set aspect ratio.

2.5.4 Game Client

ClientMessagingController.java - facilitates communication between the client and server. It supports operations such as joining games, requesting game states, adding player actions, retrieving a list of available games, and fetching all high scores.

Controllers.java - aggregates controllers for graphics, input, sound, and server messaging, facilitating both online and local client communication. Also contains the link to the game server.

GameClient.java - is initialized with a set of controllers and a unique player ID, setting up the initial menu screen and integrating a ScreenManager to handle transitions between different game states. It also dynamically adjusts game states based on server feedback or loss of connection.

States

- **ButtonFactory.java** - implements the Factory Pattern. Creates buttons based on the enum given (i.e. the type of button), adds the correct texture to it, and adds an observer to it in the form of a callback-function.
- **ChooseMap.java** - is a screen that is part of the State Pattern. Screen where you can choose the map you want to play. Similar to all others screens it extends State and implements Observer.
- **GameOver.java** - is a screen that is part of the State Pattern. It creates the game over screen, and has a button to go back to the main menu.
- **GameOverObserver.java** - an Observer-interface that handles the event of game over.
- **HostLobby.java** - is a screen that is part of the State Pattern. It contains the "Host Lobby"-screen in the Multiplayer-menu.
- **InGame.java** - is a screen that is part of the State Pattern. Contains the game itself. Extends State and implements Observer and GameOverObserver, in order to handle events as a result of button-presses. It uses the rendering system, and provides the card menu on the bottom right of the screen.
- **JoinGameObserver.java** - is an interface for Observers. It has the method onJoinGame, that tells the connected observers to handle the event of when a player joins a game.
- **JoinLobby.java** - is a screen that is part of the State Pattern. Contains the multiplayer game lobby screen, where players can join available games.
- **Menu.java** - is a screen that is part of the State Pattern. Manages the main menu screen.
- **Multiplayer.java** - is a screen that is part of the State Pattern. Contains is the multiplayer menu screen.
- **Observer.java** - is an interface for Observers that can be attached to buttons. It has the method onAction, which handles button clicks.
- **Options.java** - is a screen that is part of the State Pattern. Contains the options-menu.
- **ScreenManager.java** - is the manager of the State pattern, responsible for transitioning between states (i.e. different screens). It has been defined as a Singleton to ensure that it only has one instance.
- **State.java** - is an Abstract class that all the states (screens) in the State pattern inherits from.
- **Tutorial.java** - is a screen that is part of the State Pattern. Manages the tutorial screen.

TexturePack.java - is a class that keeps a collection of the paths to all textures used in the game as public final variables.

2.5.5 Game Server

AudioPack.java - is a class that keeps a collection of paths to all audio files used in the game as public final variables.

CardFactory.java - utilizes the Factory pattern in order to create card entities by populating them with the necessary components.

EnemyFactory.java - utilizes the Factory pattern in order to create card entities based on an enum (i.e. the cards type). The EnemyFactory decides the attributes of an enemy, and populates their entities with the necessary components.

GameServer.java - represents a game server that coordinates the lifecycle and state of a multiplayer game. This server manages game creation, state updates, player actions, and the main game loop, handling communication between clients and the server through message controllers.

GameState.java - is an object used to transfer the state of the game from the server to the client. When instantiating it the ECS will automatically overwrite it to incorporate the state of the game.

MapFactory.java - creates the maps based on a string that contains the layout of the map with the corresponding tile-types.

Map.java - is a map built of tiles (squares). Contains some essential logic for pathfinding and the correct rendering of textures based on map layout.

PairableCards.java - contains logic for deciding the type of tower you get from combining two cards.

PlayerInput.java - stores the actions of the player like placing a card and is used for communication between the GameServer and Clients.

ServerMessagingController.java - facilitates Server Client communication, which gives the server the possibility to create games, find players that are trying to join, get the player actions and update the state of the game for all clients.

Tile.java - represents a single tile on the map.

TileType.java - is an enum containing all possible types of Tiles like path tiles, buildable tiles (grass), and various non buildable tiles (rock, water, tree).

TowerFactory.java - is a factory that populates tower-entities with the necessary components based on the resulting type of tower from the PairableCards-class.

2.5.6 Graphics

- contains facades for handling graphics

LibGDXGraphics.java - is a facade that abstracts away the LibGDX COTS from the core logic, and handles graphics. It is an implementation of the GraphicsController-interface.

2.5.7 Input

-contains facades for handling inputs

LibGDXInput.java - is a facade that abstracts away the LibGDX COTS from the core logic, and handles inputs. It is an implementation of the InputController-interface.

2.5.8 Launcher

GameLauncher.java - configures the Controllers of the GameClient and GameServer by using dependency injection, and creates the UUID of the player.

2.5.9 Math

- contains useful math related classes used in the game.

Rectangle.java - is a tailor-made rectangle that is used to make boxes and contains width, height and a position, quite similar to the Rectangle class in LibGDX. We chose to make our own Rectangle-class in order to reduce coupling between the core logic and external dependencies.

Vector2.java - is a tailor made vector containing x and y-coordinates based on the Vector2 class in LibGDX. We chose to make our own in order to reduce coupling between the core logic and external dependencies.

Vector3.java - is a vector containing x, y, z. We use this to create colors.

2.5.10 Networking

- handles all communication between different devices.

ClientMessenger.java - is the implementation of ClientMessagingController.

ServerMessenger.java - is the implementation of ServerMessagingController.

Persistence - handles data that shall be accessible for several devices.

- **DAOFactory.java** - creates the DAO (Data Access Object) implementations based on the configurations of the game.

- **DAO.java** - standardizes access to various data storage systems without revealing their specifics. It uses the Data Access Object pattern. It supports generic CRUD operations—Create, Read, Update and Delete—using a primary key K to manage objects of type T .
- **FirebaseDAO.java** - is a DAO-implementation using Firebase's real-time database as the underlying technology.
- **LocalDAO.java** - is a DAO-implementation using a ConcurrentHashMap so that the game runs locally on the client without speaking to the Firebase-database. It is a Singleton, which means its possible to access everywhere in the client. The ConcurrentHashMap ensures that we do not run into any concurrency issues.

2.5.11 Sound

- contains facades for handling sound

LibGDXSound.java - abstracts away the LibGDX COTS from the core logic, and handles sound. It is an implementation of the soundController-interface.

3 User Manual

3.1 Installation

To be able to run the project make sure to have all the prerequisites and follow the the Starting the application section.

3.1.1 Prerequisites

- **Git**
  - Git is a distributed version control system essential for cloning the project from GitHub and collaborating with other developers.
  - Download and install Git from the official Git website.
  - After installation, verify the installation by running `git --version` in your command line or terminal.
- **Java JDK 17 (Download from Oracle's website)**
  - This project requires Java JDK 17 for compiling and running Java applications.
  - Download and install it from Oracle's Java JDK Download Page.
  - After installation, verify by running `java -version` and `javac -version` in your command line or terminal.
- **Gradle 6.7**
  - Gradle automates the process of building, testing, and deploying the application.
  - Download Gradle from the Gradle Download Page.
  - Alternatively, use a Gradle Wrapper script which manages the installation.
  - To confirm installation, run `gradlew -v` in your command line or terminal.
- **Android SDK (if you want to run the game on Android)**
  - Necessary for running the game on an Android emulator or device.

– Set up the Android SDK by creating a `local.properties` file in the project root:

```
echo sdk.dir=YOUR/ANDROID/SDK/PATH > local.properties
```

- **Firebase Secret Key (for multiplayer features)**
  - Obtain a Firebase secret key from the Firebase console to enable multiplayer features.
  - Find our more here: Firebase/API-KEY
  - Store the key in the ‘assets‘ folder with the filename `'FirebaseSecretKey.json.'`

3.2 Starting the application

The game is cross-platform and it is possible to play on Android as well as a desktop.

3.2.1 Starting on Android

Building APK file and installing on android To generate the executable APK file for Android run the following command at the root of the project:

- In Windows:

  ```
  gradlew android:assembleRelease
  ```

- In Linux based systems and Mac:

  ```
  gradle android:assembleRelease
  ```

Copy the APK file over to your Android device and install it. How you do this might be different from device to device. You can follow this guide on how to install the APK or look up your device and Android version on how to install the APK file.

Running in Android Studio Open the root folder as a project in Android Studio and let it configure itself the first time you open it. It should then automatically make an Android run configuration, so all you have to do is press the green run button on the top bar. Please take a look at figure 1.

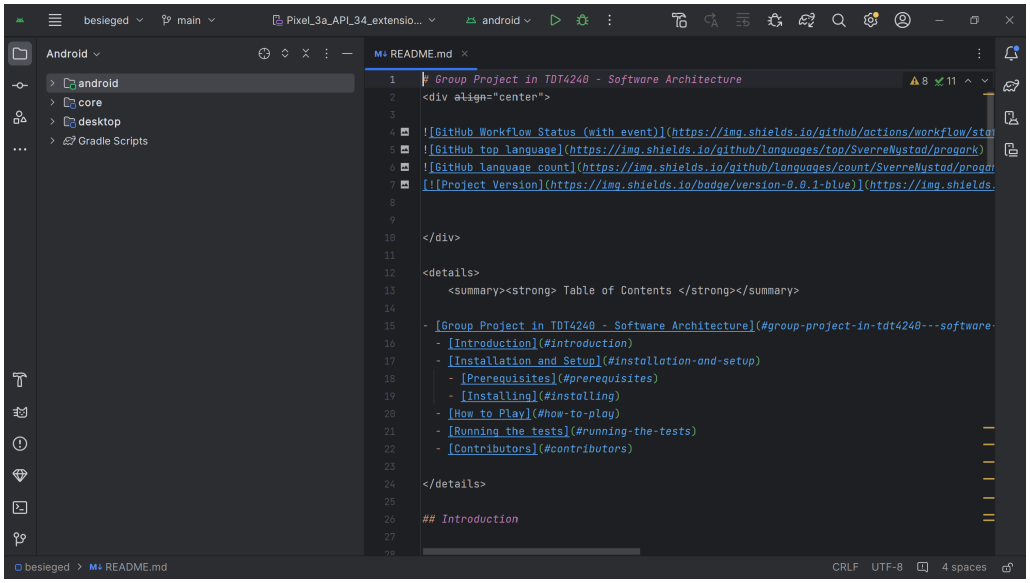


Figure 1: Android studio

3.2.2 Starting on Desktop

To run the project on a desktop run the following command at the root of the project:

- In Windows:

  ```
  gradlew desktop:run
  ```

- In Linux based systems and Mac:

  ```
  gradle desktop:run
  ```

3.3 Screens

The art style of the game is quite important to us, as we want to immerse the players in our magical world inspired by Norse mythology, a cultural inheritance for Scandinavians. Therefore, the background images are artwork, provided from OpenAI, depicting battles, monsters and scenery that fits perfectly for a game with this theme. In an effort to complete the Nordic theme, the buttons are textured as old Nordic runes edged into planks. Below you can see various menus and screens from the game.

Figure 2: Main Menu

3.3.1 Tutorial

Once you have launched the game, you will be greeted by the main menu screen. Here you can press the tutorial button for an introduction to the core game mechanics.

Figure 3: Tutorial screen: When clicked on tutorial in main menu, you will be greeted with a tutoial on how to play the game.

As you can see in the tutorial, by click on a card and then on a map tile, you can place a card. This costs gold, which is an in game currency you get from killing enemies. Then by clicking on another card and then on the already placed card, the two cards will combine into a tower, which deals damage to enemies. Then you will fight enemies until you fail to defeat a wave, and you will reach the game over screen.

Figure 4: The Options Menu: Here, you can adjust the volume.

Figure 5: Multiplayer Menu: Join an existing game, or host your own game, where you can play with a friend.

Figure 6: Highscore Menu: Here you can see the leaderboard of the 5 best players. Are you good enough to beat the highscore?

Figure 7: Join Game Menu: Here you can view all games that are available to join

Figure 8: Map Choice Menu: Choose between two available maps

Figure 9: Gameplay: Buy cards, build towers and have fun!

Figure 10: Game Over: If you run out of health, you reach the Game Over screen.

Figure 11: Lost Connection: If the host disconnects mid-game, the client is sent to the Lost Connection screen.

3.4 Towers

As you can see in the image of gameplay above, there are a total of five different cards. Each of these cards can combine into a unique tower. Each tower has its own gimmick, containing elements from both cards that make up a given tower. The cards have different ranges of how far they can attack an enemy, as well as different damage and attack cooldowns. Some towers also has the ability to attack all enemies in a given range, while others can only attack one enemy at a time. For instance, the inferno creature you make by combining the magic card with a fire card will emit a continuous heat wave, slightly damaging all enemies in a close vicinity. Here are some of the in total 15 towers that you can make (play the game to see the remaining towers and animations:

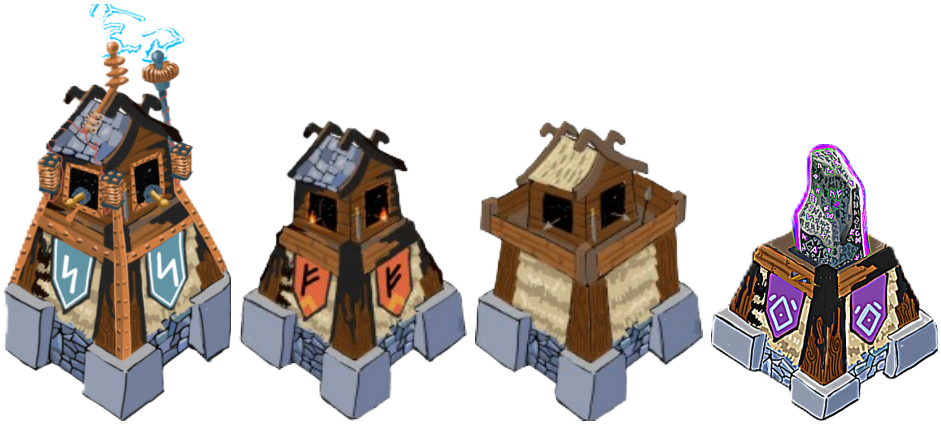


Figure 12: Various bow towers: from left to right; bow and lightning, bow and fire, bow and bow, bow and magic

Figure 13: Various magic towers: from left to right; magic and lightning (Tor, the god of thunder), magic and technology, magic and magic (a witch weaving the reality from Norse mythology), magic and fire (the witch has experimented with fire magic, and became a fire demon).

Figure 14: Various tech towers: from left to right; technology and lightning, technology and technology, technology and magic, technology and fire.

The towers are all hand drawn, having one to six animation frames. Towers that are made up of similar cards keep a theme from the cards. For instance, nearly every tower containing a technology card sits on a platform with gears, and containing some kind of technology. For instance, the combination of fire and technology is a giant furnace set on a bed of gears, shooting out flames when the lid on top of the furnace closes. Another tower, derived from technology and lightning, uses the same furnace, but converts the energy from the burning coals into electricity that emanates from a Tesla Coil. This way, if a player plays the game enough to really look at the textures, they will be rewarded with some kind of lore on how the towers connect to each other.

3.5 Enemies

The objective of the game is to defend your village from the foul creatures of Norse mythology. We have the children of Fenris, the giant wolf, that can run really fast. We also have various Vikings, all having different weapons and armor, giving them different velocity, damage, and hitpoints. There is also a huge ice giant, inspired by the giant Ymir from Norse Mythology. This giant is incredibly slow, but has a lot of health, making it a real threat to a player. With hand-made walking animations, we aim for immersive gameplay, where a player can marvel at the complexity and beauty of the textures.

Figure 15: Various enemies

4 Test Report

4.1 Functional Requirement Tests

FR1	The user should be able to interact with the game using touch
Executor:	Jens M
Date:	20.04.2024
Time Used:	5 seconds
Evaluation:	Success
Comment:	Touch works.

FR2	A player should be able to create a game lobby and invite friends to join their game
Executor:	Jens M and Magnus
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Success
Comment:	The multiplayer-menu is easy to navigate, it is easy for one player to host a game and for the other player to find the correct game.

FR3	If the game lobby is empty the player should not be able to start a multiplayer match
Executor:	Jens M
Date:	20.04.2024
Time Used:	30 seconds
Evaluation:	Success
Comment:	The hosts screen is completely black until a second player joins the game.

FR4	If the player(s) lose all their hitpoint the game over screen should appear
Executor:	Jens M
Date:	20.04.2024
Time Used:	2 minutes
Evaluation:	Success
Comment:	When the village's health reaches 0 the game-over screen appears and allows the player to return to the main menu.

FR5	The game should have an in-game menu with the necessary controls to play the game
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Success
Comment:	The player has an easy-to-use card-placing menu at the bottom of the screen, and has access to their health and money at the top right.

FR6	While the player uses the in-game menu in a singleplayer match, the game should pause for the player.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We have not implemented an in-game pause menu. The only menu available is the card menu.

FR7	While the player uses the in-game menu in a multiplayer match the game should pause for both players.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We have not implemented an in-game pause menu. The only menu available is the card menu.

FR8	The player should be able to forfeit the game
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We have chosen not to implement a forfeit feature.

FR9	The player should be able to adjust the game volume.
Executor:	Jens M
Date:	20.04.2024
Time Used:	30 seconds
Evaluation:	Success
Comment:	It is easy to adjust the volume of the game from the options-menu, which is easily accessible from the main menu.

FR10	The player should be able to place Cards on available tiles and cards.
Executor:	Jens M
Date:	20.04.2024
Time Used:	30 seconds
Evaluation:	Success
Comment:	A player is able to place cards and towers on any tile that is not obstructed.

FR11	When a card is placed on another card, the cards combine to create a tower
Executor:	Jens M
Date:	20.04.2024
Time Used:	30 seconds
Evaluation:	Success
Comment:	The combining of cards works well.

FR12	When a card is placed on a tower, the tower gains an upgrade given by the card.
Executor:	Jens M
Date:	20.04.2024
Time Used:	30 seconds
Evaluation:	Partial Success
Comment:	We chose not to implement this, because the gimmick of the game is combining cards to create a tower, and we wanted the users to create different towers instead of upgrading them, in order to distinguish our game from a typical tower defense game.

FR13	If a card is placed on an unavailable tile, the road or an enemy the card should not be placed and be returned to the player.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Success
Comment:	Players are not able to place cards on tiles that are blocked by water, rocks, trees or other foliage, and get a textual alert of this in the bottom left corner of the screen.

FR14	When a card is placed on a legal spot the cost of the card will drain the "wallet" of the player, given the player has enough currency.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Success
Comment:	This feature works well, the balance of the player correctly updates when placing a card.

FR15	Enemies shall appear at the entry point during a Wave.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Success
Comment:	This feature works as intended.

FR16	Enemies shall follow the path towards the Exit Point during a Wave
Executor:	Jens M
Date:	20.04.2024
Time Used:	30 seconds
Evaluation:	Success
Comment:	Enemies successfully follow the entire path from start to finish.

FR17	When Enemies arrive at the Exit Point, they shall disappear and the player(s) should lose hitpoints/health
Executor:	Jens M
Date:	20.04.2024
Time Used:	2 minutes
Evaluation:	Success
Comment:	The players' health is successfully decremented when an enemy makes it through the map.

FR18	When Enemies lose their entire health/hitpoints they should give their bounty to the treasury of the player(s) and disappear
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute 30 seconds
Evaluation:	Success
Comment:	The player is successfully awarded some money when killing an enemy and the enemy is returned to the pool to be respawned.

FR19	When Enemies are in the range of a Tower it should shoot a projectile at it lowering the health of the Enemy upon hitting it.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Partial Success
Comment:	We did not have time to implement projectiles, but towers successfully attack enemies within their range and make a sound when they do.

FR20	The player should be able to sell Tower(s) and a portion of the original cost of the tower should be returned to their "wallet"/treasury.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We did not have time to implement this feature.

FR21	The active wave will spawn enemies at different times throughout the duration of the wave.
Executor:	Jens M
Date:	20.04.2024
Time Used:	2 minutes
Evaluation:	Success
Comment:	Different types of enemies are spawned throughout the duration of a wave.

FR22	Once a tower has found an enemy to shoot, a projectile will be shot towards the enemy in a realistic curve that will be rendered. The enemy will be damaged.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We did not have time to implement this feature, nor to draw the projectiles by hand.

FR23	After a game has ended, the score should be appended to a public high score.
Executor:	Sverre Nystad
Date:	21.04.2024
Time Used:	4 minutes
Evaluation:	Success
Comment:	The Highscore menu shows the number of waves survived and the GameId of the multiplayer game after the game is over.

FR24	When a player selects a card the placeable areas shall be highlighted.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We did not see the need for this feature as almost all areas of the map are placeable and thus it is intuitive where cards can be placed.

FR25	When a player places a card the player needs to confirm the action for the card to be placed.
Executor:	Jens M
Date:	20.04.2024
Time Used:	1 minute
Evaluation:	Fail
Comment:	We decided to drop this feature to provide a more seamless gameplay experience.

FR26	It should be possible to rejoin a game if a player crashes
Executor:	Jens M
Date:	20.04.2024
Time Used:	3 minutes
Evaluation:	Partial Success
Comment:	When losing connection and regaining it in under 5 seconds, players are put back into the game.

FR27	If a player loses their connection, this player should be thrown out, while the other player can continue playing, and the remaining player receives the other player's resources
Executor:	Jens M
Date:	20.04.2024
Time Used:	3 minutes
Evaluation:	Partial Success
Comment:	If the host loses their connection, the other player is thrown out since we have not implemented host migration. If the client loses connection, the host can continue playing alone. Both players have the same resource pool, so the host is not awarded the clients' resources if they leave the game.

FR28	The entity hitpoints should be calibrated so the gameplay is not too easy nor too difficult.
Executor:	Magnus Vesterøy Bryne
Date:	20.04.2024
Time Used:	5 minutes
Evaluation:	Partial Success
Comment:	Health is varied among enemies, with trolls and ice giants having tons of health and ants having little health. When playing with most of the towers, the enemy hitpoints are sufficient for a fun game. Some towers, however, do too much damage.

FR29	There exists a tower for every single combination of cards.
Executor:	Magnus Vesterøy Bryne
Date:	20.04.2024
Time Used:	3 minutes
Evaluation:	Success
Comment:	Each combination of cards provides a unique tower, containing elements from both cards.

4.2 Quality Requirement Tests

4.3 Usability Tests

U1:	Inexperienced players should be able to view a tutorial for how to play the game.
Executor:	Magnus Vesterøy Bryne
Date:	20.04.2024
Time Used:	20 seconds
Environment:	Runtime, tutorial screen
Stimuli:	A use presses the tutorial button
Expected Response Measure:	Tutorial exists
Observed Response Measure:	Simple tutorial exists
Evaluation:	Success
Comment:	We have made a tutorial screen that briefly explains how the game works.

U2:	The player should be able to sell towers(undo).
Executor:	Tobias Fremming
Date:	20.04.2024
Time Used:	N/A
Environment:	Runtime
Stimuli:	A user regrets the placement of a tower and wants to delete it.
Expected Response Measure:	The tower should vanish and you should get the expected amount of money for selling the tower
Observed Response Measure:	You can not sell a tower.
Evaluation:	Fail
Comment:	This is not yet implemented.

U3:	The players should be able to pause/resume the game, given it is single player mode.
Executor:	Tobias Fremming
Date:	20.04.2024
Time Used:	N/A
Environment:	Runtime, singleplayer mode
Stimuli:	A user wants to pause the game, and then resume it.
Expected Response Measure:	everything should freeze up while paused.
Observed Response Measure:	You can not pause the game
Evaluation:	Fail
Comment:	This is not yet implemented, as the priority was quite low on this one, with the giving time constraints.

U4:	Inexperienced players should be able to easily start a new game within 60 seconds.
Executor:	Tobias Fremming
Date:	20.04.2024
Time Used:	20 seconds
Environment:	Runtime
Stimuli:	A user wants to start a new game
Expected Response Measure:	The game should start within 60 seconds of running the application
Observed Response Measure:	It takes less than 20 seconds to start a game.
Evaluation:	Success
Comment:	The expected response measure does not calculate for time to wait for your friend to join in multiplayer mode.

U5:	The gameplay should be fun and engaging.
Executor:	Magnus Vesterøy Bryne
Date:	20.04.2024
Time Used:	20 minutes
Environment:	Runtime, singleplayer mode
Stimuli:	A user wants to have fun and starts the game
Expected Response Measure:	The user finds playing the game to be fun.
Observed Response Measure:	The user enjoyed the game
Evaluation:	Success
Comment:	The game has enough content that it is fun to play.

U6:	No towers should feel overpowered or underwhelming, so that they all can be considered usable by players.
Executor:	Magnus Vesterøy Bryne
Date:	20.04.2024
Time Used:	20 minutes
Environment:	Runtime, singleplayer mode
Stimuli:	Tower attacks enemies.
Expected Response Measure:	Balance between tower's damage and attack cooldown, in addition to enemy velocity and hitpoints.
Observed Response Measure:	Tower entities are quite balanced, but a bug can make them attack too often in some cases.
Evaluation:	Fail
Comment:	The bug happened after moving the InGame logic into server, and hasn't been fixed yet.

4.4 Performance Tests

P1:	End-to-end delay should be less than 100 ms in singleplayer given the system being in normal operation.
Executor:	Sverre Nystad
Date:	20.04.2024
Time Used:	1 minute
Environment:	Runtime, singleplayer mode
Stimuli:	Normal operation with simulated user interactions
Expected Response Measure:	Delay < 100 ms
Observed Response Measure:	9 ms
Evaluation:	Success
Comment:	The system maintains a low latency under normal conditions, meeting the real-time interaction requirements for singleplayer mode.

P2:	End-to-end delay should be less than 100 ms in multiplayer given the system being in normal operation and internet connection being stable.
Executor:	Sverre Nystad
Date:	20.04.2024
Time Used:	1 minute
Environment:	Runtime, multiplayer mode
Stimuli:	Stable internet connection with simultaneous player interactions
Expected Response Measure:	Delay < 100 ms
Observed Response Measure:	183 ms
Evaluation:	Failure
Comment:	Multiplayer interactions are above the target latency. This is under controlled network conditions on PC. During user testing, users have not complained. The original target might have been unrealistic.

P3:	The game should be able to run at a frame rate that ensures nice movements of entities on the GUI, and smooth frame transitions.
Executor:	Sverre Nystad
Date:	20.04.2024
Time Used:	5 minutes
Environment:	Runtime, graphical interface
Stimuli:	High-density gameplay with many enemies and towers in singleplayer mode
Expected Response Measure:	Smooth frame transitions and movements
Observed Response Measure:	60 FPS consistently
Evaluation:	Success
Comment:	The game consistently maintains high frame rates, ensuring smooth visual performance and responsiveness.

4.5 Availability Tests

A1:	The database should be available 99 percent of the time.
Executor:	Jens M
Date:	20.04.2024
Time Used:	15 minutes
Environment:	Runtime, multiplayer mode
Stimuli:	Wishes to play the multiplayer mode
Expected Response Measure:	Multiplayer functions properly
Observed Response Measure:	Multiplayer functions without large amounts of delay
Evaluation:	Success
Comment:	The game is a bit choppy in terms of frames per second, but functions properly and updates quite fast.

A2:	The game should be available to players long after we are done with this subject.
Executor:	Jens M
Date:	20.04.2024
Time Used:	N/A
Environment:	Runtime
Stimuli:	Wishes to play the game
Expected Response Measure:	Game functions properly
Observed Response Measure:	Not possible
Evaluation:	Success
Comment:	Even if Firebase were to stop functioning, players could still play the singleplayer-mode without us having to do anything. This is because we don't have any hardware to maintain.

A3:	In case of unplanned outages, automatic recovery procedures should be in place to restore services in 15 minutes.
Executor:	Jens M
Date:	20.04.2024
Time Used:	5 minutes
Environment:	Runtime, multiplayer mode
Stimuli:	Wishes to play the game
Expected Response Measure:	Ongoing game should be restored to its previous state before outage
Observed Response Measure:	This feature has not been implemented.
Evaluation:	Fail
Comment:	We have no direct control over Firebase's database, but given that their system is up and running our service will automatically restore service

4.6 Modifiability Tests

It is important that the parts of the software that are likely to change are easy to change. It should be easy for a developer to add new cards, towers, maps, and enemy entities, with little to no refactoring of already existing code.

M1:	It should be easy for a developer to add new enemy entities, with little to none refactoring of already existing code.
Executor:	Tobias Fremming
Date:	16.04.2024
Time Used:	20 minutes
Environment:	Design Time
Stimuli:	Wishes to add a new Enemy to the system
Expected Response Measure:	With no more than 3 hours of work and without introducing any side effects.
Observed Response Measure:	2 minutes of work and without any side effects.
Evaluation:	Success
Comment:	The response measure assumes that we already have the necessary sprites drawn, as hand drawing the textures takes a lot of time if you want it to look nice.

M2:	It should be easy for a developer to add new cards, with little to none refactoring of already existing code.
Executor:	Tobias Fremming
Date:	18.04.2024
Time Used:	20 minutes
Environment:	Design Time
Stimuli:	Wishes to add a new Card to the system
Expected Response Measure:	With no more than 3 hours of work and without introducing any side effects.
Observed Response Measure:	2 minutes of work and without any side effects.
Evaluation:	Success
Comment:	Adding a new tower was easy as you just have to add another switch-case in the CardFactory. This totaled just 7 lines of code.

M3:	It should be easy for a developer to add new maps, with little to none refactoring of already existing code.
Executor:	Tobias Fremming
Date:	20.04.2024
Time Used:	1 hours
Environment:	Design Time
Stimuli:	Wishes to add a new map to the system
Expected Response Measure:	With no more than 3 hours of work and without introducing any side effects.
Observed Response Measure:	1 hours of work and without any side effects.
Evaluation:	Success
Comment:	The UI has to be refactored, and the map needs to be put together. Here you only need to make a string telling what tiles you want in order.

M4:	It should be easy for a developer to add new towers, with little to none refactoring of already existing code.
Executor:	Tobias Fremming
Date:	18.04.2024
Time Used:	20 minutes
Environment:	Design Time
Stimuli:	Wishes to add a new tower to the system
Expected Response Measure:	With no more than 3 hours of work and without introducing any side effects.
Observed Response Measure:	2 minutes of work and without any side effects.
Evaluation:	Success
Comment:	Adding a new tower was easy as you just have to add another switch-case in the TowerFactory. This totaled just 7 lines of code.

M5:	The development team should easily be able to change the database if the need arises.
Executor:	Tim Matras and Sverre Nystad
Date:	19.04.2024
Time Used:	3 hours
Environment:	Design Time
Stimuli:	The need to change the database arises in order to create local database for local play.
Expected Response Measure:	With no more than a work day (8 hours) of work and without introducing any side effects.
Observed Response Measure:	3 hours of work and without any side effects.
Evaluation:	Success
Comment:	This could take longer time for more complicated databases, but the database is invisible to the core logic past the factory producing database access object.

M6:	The system should not be too dependent on libGDX, and in case of failure of libGDX, we should be able to switch to other COTS that can provide similar services.
Executor:	Magnus Vesterøy Bryne
Date:	20.04.2024
Time Used:	N/A
Environment:	Core game logic
Stimuli:	The need to change the database arises in order to create local database for local play.
Expected Response Measure:	No libGDX in core logic.
Observed Response Measure:	No libGDX in core logic.
Evaluation:	Success
Comment:	We have abstracted libGDX, meaning we are not overly dependent on the libGDX library. We could switch over to JavaFX if we wanted too. Time used is N/A because we did not actually try switching out libGDX as it would still take a days worth of work.

M7:	A developer should easily be able to add new states to the game in order to expand it. .
Executor:	Sverre Nystad
Date:	20.04.2024
Time Used:	14 minutes
Environment:	Design Time
Stimuli:	A new state has to be added.
Expected Response Measure:	It should take less than 3 hours to make a new screen/state in the game
Observed Response Measure:	14 minutes of work and without any side effects.
Evaluation:	Success
Comment:	This could take longer time for more complicated states with a lot of logic, for instance a minigame, but here it is the logic that takes up time, as the states are quite easy to add.

5 Relationship with Architecture

Few changes have been made to the architecture from the initial planning phase to the finished product. This could be attributed to the fact that the architecture was quite high-level and didn't contain many low-level implementation details, which gave us the freedom to make changes along the way when necessary. The team had quite a bit of experience with both game development, LibGDX and the Entity Component System (ECS) pattern from earlier projects, which made the planning process a bit smoother. We did however make some changes along the way due to inexperience with making a real-time multiplayer game-mode, as this was quite a bit more complex than we imagined. (We worked agile throughout the project and were therefore able to adjust our implementation when we ran into problems.) kan kanskje ta bort den siste her, se an

5.1 Changes made

We have made changes to the ClientMessagingController and ServerMessagingController-interfaces by adding quite a few new methods from the ones we defined in the Class Diagram of our networking API in the first edition of our Architectural Description report. This is largely due to our inexperience with developing a real-time multiplayer mode. This meant we simply had not thought of all the methods necessary for developing such a mode, and therefore discovered these extra methods during development. Adjusting the interfaces were however not a complex process, and therefore did not hinder the development in any meaningful way.

We also made updates to how the Game Client and Game Server use the Entity Component System (ECS). In the original architectural description we thought of the ECS as being part of and baked into the Game Client and Game Server, as shown in our old package and layer-diagrams. We have now updated the ECS to be its own separate module that the Client and Server both use. This is because the ECS exists sort of on its own. It has entities with components that both the Client and Server use, and systems that run in the background and constantly update the entities in some way.

In addition to this we also made some smaller changes here and there to make things a bit cleaner and easier to use. We feel like these changes are not relevant to mention as they did not change the architecture or how the game works in any significant way.

5.2 Performance Tactics

5.2.1 Object Pooling

During the ATAM Evaluation session with another group, we received questions about our lack of performance tactics. We realized that we could easily implement some tactics, including Object Pooling of enemies. This means that instead of creating new enemy-entities each time a new enemy spawns, we reuse the existing entities of dead enemies. When enemies die, we move their sprite to outside of the map near the start of the path, and then simply refill their health and send them on their way through the map once more. This saves a lot of memory as we do not need to constantly invoke LibGDX-methods to create new Textures and use the ECS to initialize entities and populate all their components with the same content as was already there.

Nye ting i arkitektur: Extracta ECS fra Game Client og Game Server i System View. Har lagt det som en egen separat greie som både Client og Server bruker. ClientMessagingController og ServerMessagingController trengte flere metoder enn i Class Diagram of networking API Lite domenekunnskap om multiplayer førte til at det var enkelte ting vi ikke hadde tenkt på

6 Problems, Issues and Points Learned

Due to the complex nature of the ECS-system, and group members having almost no prior experience with it, we initially found it difficult to understand the control flow of it. This made the initial development slow, as it was hard to write clean and sustainable code without properly understanding the intricacies of the ECS.

We also had to continually adapt the ECS when introducing new features, especially the multiplayer-mode. For example, we had a bug which led to concurrency-issues when adding new entities to the ECS while the update-cycle was running. This occured when trying to add card-entities to the map. We struggled quite a bit with solving this issue, but managed to find an easy fix in the ECSManager after some time. This could probably have been solved faster if we had a better understanding of ECS from the start.

If we had chosen a COTS to take care of the same job as the ECS-system, we could probably have saved some development time by avoiding these kinds of problems. However, this is risky as we would have an external dependency in our core logic, which would greatly increase the coupling of the game. By building the ECS from the ground up ourselves, we got a tailor-made system that perfectly met our specific needs. We also think we became better programmers, as we had to face a lot of struggles we otherwise wouldn't if we used a COTS.

We found that the development speed greatly increased after developing the essential systems required by the ECS. From here on out, we felt that the initial upfront cost was worth it, as the ECS-system made it very easy to add new business rules to the core logic. As an example, late in the development cycle we decided to give some towers an AreaOfEffect-component, which gave them the ability to damage all enemies within their range. This was extremely easy, as we only needed to add a new component and slightly modify the AttackSystem. This serves as an example of our core logic following the Open-Closed Principle, which states that the code should be locked for modification, but open for extension Martin 2017, p. 69.

We also experienced some issues with conversion between XY-coordinates and UV-coordinates. An example are the Tiles on the map which have coordinates given in an XY-coordinates system, while the position-vectors in the PositionComponent in the ECS-system have their position given in UV-coordinates. This means we had to create quite a few conversion methods in various places, which is not ideal. Due to the ECSManager being a Singleton, we were however able to fetch the screen-size of tiles from it to use for conversion. Looking back on this, we should maybe have been more consistent in our usage of different coordinate systems to avoid these conversion-problems.

While the Singleton pattern used for the different kinds of managers (ECSManager, ScreenManager

etc.) made it easier for us to develop our game, it also led to some unforeseen issues when developing the multiplayer mode. This is because the Singleton pattern leads to increasingly complex connections over time GitLab 2024. This happens because every component can interact with the Singleton instance. This accessibility often results in quick, makeshift solutions that accumulate technical and architectural debt. Consequently, architectural boundaries become blurred as virtually anything can communicate with the Singleton. While this isn't a major issue in our current project, it has led to minor bugs. If this project were to extend over several years, the problem would likely become worse.

Due to the groups inexperience with developing a real-time multiplayer game, we had different mental models of how the multiplayer aspect affected where the core logic of the game was to be placed. Since the singleplayer mode served as a testing ground for the core business logic, and the fact that the multiplayer took a long time to develop, a large portion of the game-logic was written on the Client side of the application. When the multiplayer mode was finished, almost all of this logic needed to be moved into the Game Server, and some of it also needed to be changed to function with the Client-Server Architecture.

As we did not have a complete overview of needed features and methods for graphics, audio, input-detection and other input/outputs, we started using LibGDX directly in our code, with the intention of finding out what we needed and then abstract it away later. We did this to avoid making wrong abstractions early on, and to avoid accumulating technical debt Fowler 2019. When reaching the point where we knew the necessary abstractions, we had to pay our technical debt in the form of refactoring our already written code. Despite the large amount of time we used doing this, the interfaces between the components responsible for graphics, audio and input-detection become easier to work with after it. This decision protected our application and reduced its coupling to external dependencies. For example: We do not explicitly define how the sprite-components of the entities created on the client-side should be drawn, this is all done automatically by the LibGDXGraphics-class which implements our GraphicsController-interface. This is the only place LibGDX-specific methods are invoked, keeping the coupling to our application low.

It is crucial to protect core logic from external dependencies while also balancing the time spent on this task. In retrospect, abstracting away LibGDX may not have been necessary, as forking a functioning version could prevent sudden unavailability, unlike platforms such as Firebase.

Due to time constraints we did not do test-driven development. Therefore we did not have automated tests that always checked the health of our system, which made refactoring a bit riskier as it was hard to immediately discover bugs that may have been introduced.

Due to the asynchronous nature of the AudioSystem in the ECS, the logic for playing sounds on demand was a bit unintuitive. We had to add some boolean flags to the components of different entities like the EnemyComponent and TowerComponent, that had to be manually set whenever we wanted to play a sound, and then checked in the update cycle of the ECS.

When moving the logic from the Client to the Server as mentioned above, we ran into some unexpected issues with the way we were managing sounds. We discovered that sounds are not easily played on demand in an asynchronous environment like an ECS system. However, by improvising a bit we managed to get it working. This is a small downside of the ECS, but we think the advantages of it in other areas make up for this.

7 Conclusion

We had a lot of fun creating this game. The final product is an advanced cross-platform real-time multiplayer game, which we are proud of. We are very satisfied with the outcome. Both friends and family really enjoy the game and find it engaging and fascinating. The most important lesson learned during this project is the incredible value of creating a detailed and structured architecture that is easy to build upon before blindly starting development.

Towards the end of the development cycle we experienced that the development speed rapidly

increased. In previous projects, we have found that development progresses quickly in the beginning, but slows down later on due to overly intricate code with low modifiability. This results in efforts being diverted away from creating new features, in order to manage messy code. This is a phenomenon described in the book "Clean Architecture" by Robert C. Martin on page 8. Martin 2017, depicted by the following graph.

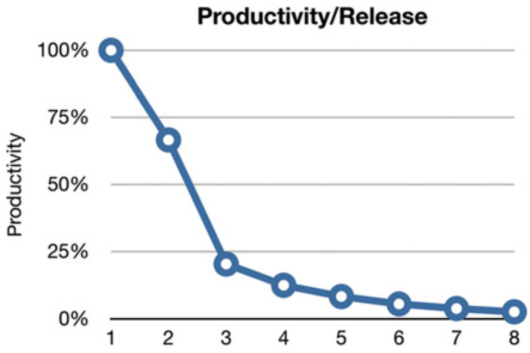


Figure 16: Productivity by release. Messy code reduces the productivity

We think that the productivity increase towards the end was due to us creating a solid architecture, and also our increased understanding of it over time. Designing structured sets of architectural elements also made implementing new features easy, even as the game grew in size and complexity, showcasing the benefits in modifiability from our Entity Component System.

Having a detailed and structured architecture has allowed us to have both modifiable and performant code. As mentioned earlier, the fact that productivity increased towards the end has given us an invaluable lesson in understanding how creating a good architecture can help us become better programmers. This is especially true when considering the contrast from earlier projects where we have been struggling with technical debt late in development from not creating a detailed architecture in the start. As stated by Bass, Clements and Kazman, functionality is achieved by assigning responsibilities to architectural elements, even though functionality is independent of any specific architecture Rick Kazman 2020, p. 40. This is a lesson we will take with us, especially later in life when we enter the industry and create applications that need to last for many years instead of just a semester.

article makecell float

8 Individual Contribution

Name	Nature of work	Hours
Jens Martin Norheim Berget	• Problems, Issues and Points Learned (3h) • Relationship with Architecture (2h) • Test Report (2h) ' • Design and Implementation Details (1h) • Conclusion (1h) • Reading and fixing language (4h)	13h
Magnus Vesterøy Bryne	• Problems, Issues and Points Learned (1h) • Testing and writing (3 h) • Writing User Manual (2 h) • Writing on classes (2 h) • Writing conclusion (1 h) • Proofreading (2 h)	11h
Mattias Tofte	• Writing on the Design And Implementation (2 h) • Writing on Problems, Issues and Points Learned (3 h) • Proofreading (1 h)	6h
Sverre Nystad 15h heightTim Matras	• Setting up document (1h) • Creating diagrams (6h) • Writing Design and Implementation Details (6h)	13h
Tobias Fremming	• Testing and writing (4 h) • Proofreading testing compared to requirements (1 h) • Writing User Manual (2 h) • Identifying patterns implemented (1 h) • Writing on User Manual (1 h) • Writing on classes (2 h) • Writing conclusion (1 h)	12h

Table 1: Individual Contribution