NTNU – Trondheim
Norwegian University of
Science and Technology

TDT4240 Software Architecture

Group 4 - Architectural Design

Jens Martin Norheim Berget

Magnus Vesterøy Bryne

Sverre Nystad

Mattias Tofte

Tim Matras

Tobias Fremming

Chosen COTS: Android SDK, LibGDX and Firebase

Primary quality attribute: Modifiability

Secondary quality attributes: Usability, Performance and Availability

March 03 2024

Table of Content

1 Introduction

Our main goal is to learn how to design and use software architecture concepts in a real project while also creating a game that is fun to play. We will create the architecture from scratch using the theory we have learned in the course, and implement it using Java.

The architectural phase is a critical phase in our project's lifecycle as it shapes how the application and development process will evolve. The decisions we make about the architecture now will directly impact our options in the future, so it is critical that we create an architecture that leaves as many options available as possible. Together with the requirements document, this document will serve as the original foundation of our work.

In this architectural phase we will:

- Choose quality attributes
- Find architectural drivers
- Identify stakeholders and concerns
- Decide our founding architectural tactics
- Decide on what design and architectural patterns to use
- Define the architectural views
- Make sure that there is consistency between our architectural views

We expect that the outcome of this phase is an architecture that fulfills our requirements while also leaving as many options open for the future as possible.

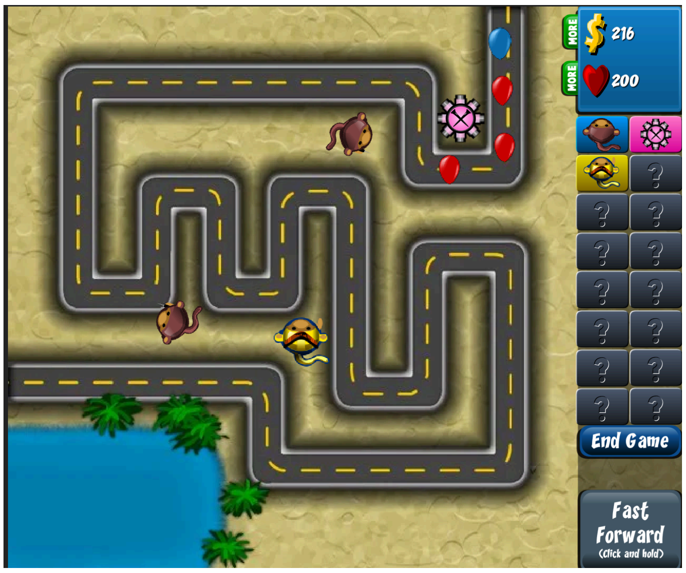
1.1 Game description

The name of our game is *Besieged!*. *Besieged!* is a real-time cooperative multiplayer tower defense game based on viking- and norse mythology. Players join shared instances where they have to cooperate in real-time as enemies invade. Together with your friend, you will need to buy and place cards to build towers and defend against waves of foul creatures from norse mythology to save your village. In order to build a tower, a player has to place a "tower-card" or an "element-card" on an empty grid/building-site-block on the map. The other player then has to place another card on the same grid to create a tower. The two cards will combine to create a unique tower with unique characteristics. The tower will then attack any enemies that come within its range. When an enemy is killed each player is rewarded with some money that they can use to buy more cards from the store.

Enemies will spawn in waves at a starting position and follow a predetermined path to your village. If they reach your village it will take damage. If the village loses all its health it will burn down and you will lose the game.

Besieged! builds upon ideas from other tower defense games, such as Bloons Tower Defence 4 (BTD4). BTD4 exemplifies some of the core concepts within the tower defense genre, and a snapshot is presented in Figure 1.1.1. The game uses balloons as enemies and monkeys as towers. *Besieged!* attempts to iterate on the core concept of static towers by creating dynamic combinations using multiple cards.

Figure 1.1.1 Snapshot of Bloons Tower Defense 4



2 Architectural Drivers / Architecturally Significant Requirements (ASRs)

2.1 Functional requirements

Real-time Online Multiplayer

The most technically challenging feature of the game is the real-time online multiplayer. For this to work, the current state of the game must be synced and accurately reflected across multiple devices. To make the implementation of this as simple as possible, we have selected the Client-Server pattern instead of a peer-to-peer approach. In this way, we clearly separate the actual updating of the logic in the *server* with what the players actually sees in the *client*. Such an overarching pattern will of course significantly affect the architecture of the program, as it dictates how the different components must communicate with each other.

Different Tower Cards and Element Cards

An essential aspect of the gameplay is the tower cards and element cards, which combine to form a tower. Given the large number of card-types, the architecture needs to facilitate both easy creation and modification of them. This will be done using factories that create card- and tower-entities and populate them with the necessary components.

Different Enemies

Having different types of enemies with different characteristics is an important part of avoiding repetitiveness in tower defense games. The most important architectural aspect of the implementation of enemies is that it is easy to add new types of enemies without having to change existing code, as well as being able to modify existing enemies without breaking the game.

2.2 Quality requirements

Modifiability

This is the main quality attribute of our game, and it will therefore affect the architecture considerably. It needs to be easy to add, remove or modify different types of enemies, towers and cards.

Usability

One of the most important qualities of any game is that it is easy to use. A game that lacks this quality will not attract very many players. Therefore, our game must have both intuitive menus and easy-to-use controls in-game that can be understood by new players without going through a lengthy tutorial. It is also important that the game doesn’t lack basic functions that players expect of similar games, like some sort of undo-functionality for when they do something they didn’t mean to. An example of this could be the ability to sell towers for some amount of coins.

Performance

Game performance is an integral part of the player’s gaming experience and directly impacts their enjoyment of the game. Our game logic is relatively simple compared to large-scale games, and a significant pause/delay to complete specific calculations should not be necessary. However, in later waves where the number of enemies increases significantly, some delay must be tolerated, simply due to the limited compute/hardware resources available in mobile devices. This is also the case for tower defense games made by professional game studios, such as Bloons TD 4, so we view this as a reasonable exception to performance requirements under normal operation.

To ensure good performance overall, our architecture must be as simple as possible, components should communicate as little as possible, and there should be little unnecessary overhead in our implementation. When components need to communicate, they should do so as efficiently as possible by only sending the necessary data and nothing else.

Availability

Players have come to expect a high degree of availability from the games and services they use. To ensure our game is available we have created a robust architecture with sufficient error handling to prevent crashes, combined with a highly reliable Firebase server. Firebase have listed an uptime of at least 99.95% in their Service Level Agreement[1], we aim for an uptime of at least 99%, as described in our quality requirements.

2.3 Business requirements

Educational and developmental objectives

The architecture must enable practical applications of architectural patterns and tactics, as well as design patterns learned in the course.

Game mechanics and features

The primary objective of the architecture is to fulfill all requirements given by the assignment. The game should offer intuitive touch-based controls for interaction, ensuring a seamless user experience on devices with different screen sizes. It should also offer an easy to use multiplayer system where a player can invite a friend to play with.We would also need an architecture that supports dynamic content delivery, making it easy to add new content like towers, cards etc. without significant downtime.

Use of COTS

The application must be compatible with the chosen COTS (Commercial off-the-shelf) components including the Android SDK, LibGDX and Firebase.

Resource optimization

Given the hardware constraints on devices like phones and tablets, we need to consider the application's resource usage. It needs to run smoothly even on devices with limited computing power available, which means the architecture has to be efficient and not waste hardware resources.

[1]Firebase, 2020

3 Stakeholders and Concerns

3.1 Concerns/interests

Stakeholder	**Architectural concerns/interests**
Lecturer and teaching assistant	● Appropriate design patterns and tactics should be used for the game. ● The architecture for the game should facilitate a good learning arena for concepts in software architecture. ● The game should contain mechanics that require exploration of different architectural theories and practices. ● The game should follow all the project requirements given by the course's staff. ● All the functional requirements and quality attributes should be met.
Us, the developers	● The software architecture should be easy to modify. ● The software architecture should facilitate a good arena to discuss different software architecture concepts to make us learn more about it. ● It should be easy to add new content to the game. ● We should not be too dependent on any COTS.
ATAM evaluators	● Does the software architecture meet educational objectives? ● Are the arguments for the design of the software architecture justified in a satisfactory way? ● Does the software architecture make sense for the game described?
End-User	● The game should be interesting and fun to play ● The game should be easy to use by having an intuitive UI ● The game should perform well, i.e. not be laggy

table 3.1.1: Concerns/interests

3.2 Architectural Viewpoints

View	**Purpose**	**Related stakeholders**	**Notation**
Logical	Understanding how the system meets its functional requirements and provides a basis for stakeholders to grasp the system's capabilities and design decisions	Lecturer and teaching assistant Us, the developers ATAM evaluators	UML Class Diagrams: Show the system's classes, their attributes, operations, and the relationships among objects. UML Package Diagrams: Organize the model of the system's classes, interfaces, and other elements into packages
Process	Understanding the system's behavior under different operational scenarios, including concurrency, distribution, and interaction patterns	Lecturer and teaching assistant Us, the developers ATAM evaluators	UML Sequence Diagrams: Illustrate how objects interact in a sequential order, which is useful for detailing game dynamics and interactions. UML State Diagrams: Describe the workflow of activities and actions, showcasing the game's processes from a high-level perspective.
Development	Making the system easier to understand, maintain, and extend	Lecturer and teaching assistant Us, the developers	UML Package Diagrams: Organize the model of the system's classes, interfaces, and other elements into packages

		ATAM evaluators	
Physical	Ensuring the system's performance and availability, addressing how the architecture supports scalability, load balancing, and fault tolerance	Lecturer and teaching assistant Us, the developers ATAM evaluators	UML Deployment Diagrams: Demonstrate the physical deployment of artifacts on nodes, showing how software and hardware interact in the system.
Scenarios	Validating the architecture against real-world requirements, ensuring that the system behaves as expected in different situations. Understanding the practical application of the system, its usability, and its performance under various conditions	Lecturer and teaching assistant Us, the developers ATAM evaluators End-users	Sequence Diagrams (for specific scenarios): Details the interactions between components and actors within specific scenarios, providing a step-by-step visualization of particular functions or gameplay elements.

table 3.2.1: Architectural viewpoints

4 Architectural Tactics

4.1 Modifiability

4.1.1 Increase cohesion

We are increasing the cohesion of our software architecture by making sure that each package (part of the architecture) is only responsible for one isolated part of the game. For example, all the core logic of the game is divided into three components: the gameServer, gameClient and ECS-system packages. This follows the Client-Server Architectural pattern. The gameServer is responsible for handling the mechanics of the game. In other words it will make sure that all the rules are followed, and process the players' actions. The gameClient is responsible for displaying the game's current state to the player(s) and collects the players' inputs like placing a card. The inputs are then sent back to the gameServer to be processed. Similarly, everything to do with for example networking is part of the networking package, everything to do with graphics and rendering is part of the graphics package and so on. The ECS-system is responsible for handling much of the game-logic happening in the background like the attacking of enemies, movement and rendering. This clear division of the architecture into separate parts, each with their own responsibility, makes it considerably easier to carry out changes later in the development process.

4.1.2 Reduce coupling

Our core logic will be in the gameServer and gameClient packages and should therefore not be dependent on the other packages. We have therefore decided that these two packages will define abstract classes/interfaces that other packages will need to implement, to make sure that the game can still work even if we for example change our networking implementation/solution. Any changes in the other packages would not need the gameServer and gameClient to change, and thus makes our game more modifiable. The rationale behind this is further defined in 8 Architectural Rationale. Throughout the development process, we have decided to move most of the interfaces to the ECS part of the game logic. Some interfaces are still in gameServer and gameClient as planned to reduce coupling.

4.1.3 Defer bindings

Since gameServer and gameClient define the abstract classes/interfaces that other packages will need to implement, they do not need to know exactly which implementation these other packages are using. This means that it's possible to use different implementations of the abstractions during runtime based on what's best for the current situation and the user's actions.

4.1.4 Single Responsibility Principle

The Single Responsibility Principle is described by Robert C. Martin as "A module should have one and only one reason to change," and "A module should be responsible for one, and only one, actor." Following this principle will result in that a module will only get one responsibility, and only addresses a single functionality aspect, and thus has only one reason to change. This can be seen in the Data Access Object pattern (DAO), which has only one responsibility, and only one reason to be changed.

4.1.5 Open-Closed Principle

The Open-Closed Principle (OCP) is one of the SOLID Principles, and tells us that a software artifact should not be open for modifications, but open for extension (Martin, 2019, p. 70). This means that a module should be made in a way that easily allows for adding new logic, without modifying existing code. This is something we try to use in our architecture. A great example of where this is useful, is within the Entity Component System (ECS). When a tower is added to the game through the tower factory, the ECS will get an entity with the wanted components, and their corresponding systems. If you decide that you want a tower with Area Of Effect damage, in order to make the tower attack multiple enemies, all you need to do is add an AreaOfEffect component and add the logic in the corresponding system, and then the tower will attack multiple enemies. By doing so, the ECS can be extended, without being modified.

4.1.6 Liskov Substitution Principle

The Liskov Substitution Principle is also one of the five principles known as the SOLID Principles (Martin, 2019, p. 78). It tells us that when extending a base class or implementing an interface, we do not change its behavior. You should be able to treat any instantiation of a class the same way you would treat the super class. We use this principle as a tactic in our architecture, and do not violate this principle.

4.1.7 Interface Segregation Principle

The Interface Segregation Principle is well known as a SOLID Principle. It aims to reduce the impact of changes, and avoid situations where change of an artifact doesn't affect modules that don't even use the changed artifact. This is done by segregating the operations into interfaces (Robert, 2019, p. 84). This can help us reduce coupling, and is a principle we use.

4.1.8 Dependency inversion principle

The Dependency Inversion Principle (DIP) is one of the SOLID Principles. It tells us that to make a system flexible, you should refer only to abstractions and not to concretions (Martin 2019, p. 87). This means only referring to source modules containing interfaces, abstract classes or other abstract declarations. In order to do so, we want to surround the high level modules in a layer of abstractions. As you may see below in the package diagram in Figure: 6.3.1, the three main modules; game_client, game_server, and ecs have a layer of abstraction around it (also clock, but this does not qualify as a part of the main logic). For instance ecs only refers to the interfaces of input, sound and graphics, as well as the other three main modules. This way we have also abstracted away libGDX, so the core logic will not be directly dependent on this dependency. By having the high level modules only referring to interfaces and other abstractions that they define, these modules will only refer to the implementations of these interfaces, and the core logic is unaware of the actual concretions of the interfaces. By doing so we also ensure that the directions of dependencies are only towards the core logic. If you study Figure: 6.3.1 closely, you may see a violation of this principle, as the launcher creates the game client. However one violation like this is, as stated by (Martin 2019, p. 91), A DIP violation of this nature cannot be completely removed, and most systems will contain at least one violation of this nature.

4.1.9 Stable Dependency Principle

The Stable Dependency Principle (SDP) is mentioned in the book *Clean Architecture*, which we have used actively in our architectural decisions. SDP states that dependencies should point towards stability. In practice, this means that if a package points towards (depends on) another package in a package/class diagram, then the arrow should point towards the package that is more stable. With stability we here mean how likely some piece of software, like a package or component, is to change. If something is more stable, then it is less likely to change. So by making an architectural boundary between the core logic of the game, and LibGDX and Firebase by creating abstractions, we separate the core logic of our game from LibGDX and Firebase, as this should be the most stable part of our system because it defines what are game should be, and this idea should be stable. We do not want LibGDX and or Firebase to define our core logic of the game, since we might at one point want to change to other COTS or implementations without it affecting how our core logic plays out.

4.2 Usability

We had originally planned to use both the support user initiative and the support system initiative tactics. Unfortunately, due to a lack of time, we did not implement it in our game before the deadline. However if we did have enough time, we would implement these tactics as written below.

4.2.1 Support user initiative

It will be possible for the player to cancel some of their actions. If they for example try to place a card, they will need to press a checkmark button to place the card and can cancel it if they so desire. The user will also be able to pause the game and resume it. In multiplayer, if one player pauses the game it will be paused for both players. To resume play, only one of the players has to hit the unpause-button. This is important for the user’s experience.

4.2.2 Support system initiative

The game will maintain a “task model” for the user by showing them the possible options of an action. For example: when the player wants to place a card somewhere, all the available areas will light up because the system anticipates that the user will place the card in one of these places and in this sense maintain a task model for the user. This will help the user understand how to play the game.

4.3 Performance

4.3.1 Manage resources

The host mobile device will run both the server and client for the game. To make sure that both of these processes get enough resources, they will run concurrently. In this way both players will hopefully have a smooth experience while playing the game, even though one of the players runs the server at the same time as playing the game. This structure of the multiplayer-mode works as a load balancer, as we do not have ever-increasing strain on a centralized server that every game uses, but rather make the players themselves be responsible for distributing the server-load. Another benefit of this choice is that the host player can continue playing even if the client player disconnects.

4.4 Availability

4.4.1 Heartbeat

We will detect if the server or the player has crashed by using a heartbeat. This is a type of communication packet sent between nodes with a regular interval in order to check their “health”. If a crash is detected the appropriate actions will be taken. We want to do this so that we can take appropriate actions if the server or one of the players were to crash or leave. Our implementation of heartbeat is done indirectly by the client by asking the server for a new game state and checking the timestamp. If the last update from the server is more than 10 seconds old, then the client will go to the Game Over screen and show that it lost connection.

4.4.2 Recovery

If we detect that the server has crashed, the host client will try to restart it. If the server detects that a client has crashed, then the game will pause until another player has joined, given that it’s in multiplayer mode. This will hopefully let us meet the requirement for high availability. Unfortunately this was not implemented before the deadline.

5 Design patterns and Architectural Patterns

The architectural patterns we have chosen are the Entity Component System (ECS) and Data Access Object (DAO). We also use the Client-Server architectural pattern.

The design patterns we have chosen are the Factory Method, the State Pattern and the Singleton pattern. Furthermore, we chose to add the design patterns Object Pooling, Observer, Game Loop and the Facade pattern.

The ECS and Factory Patterns both make it easier to modify our game. The ECS Pattern organizes game entities into individual components, and makes it easy to combine different types of components into specialized component-types (Unity, 2024). An example would be creating a special type of hybrid enemy that is a combination of two other enemy-types. The ECS makes this process easy as it ensures you don't have to create interfaces for every type of enemy. This saves a lot of time, as the number of interfaces would grow exponentially with the number of enemy-types. The Factory Pattern makes it easier to add new content and features by making it possible to introduce new types or variations of entities without impacting existing system components. Examples of this would be adding a new enemy that is a slightly stronger and faster version of an existing enemy or adding a new tower upgrade based on previously existing towers.

To make the game multiplayer and 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.

ECS can improve game performance by optimizing the processing of entities and components, which is especially important in a multiplayer setting where the efficient handling of numerous game objects is crucial for minimizing latency and maintaining sufficient FPS. Since the two players are supposed to work together while playing, they should be able to cooperate in real-time. The class managing the Entity Component System will be implemented using the Singleton design-pattern to make sure that only one can exist at any given time. This is to ensure that the sole instance has full control over the system, hereby reducing the number of possible bugs.

The DAO design pattern ensures simple data access and manipulation, contributing to overall system modifiability by optimizing database interactions, which can be critical for games that rely on frequent data retrieval and updates (Oracle, 2024). By creating an abstraction for accessing the data access layer, the DAO helps in ensuring that data-related operations are reliable and efficient. Proper implementation can also speed up changes to be made to the database or data sources, as one only needs to modify the DAO, not the program code itself.

By managing game states (e.g., menu and in-game) seamlessly, the State pattern contributes to a more intuitive user experience. Transitions between different parts of the game are smoother, making the game easier to learn and use, thereby improving usability. The State pattern significantly streamlines the management of game entities' behaviors, like attacking or moving, by recognizing that a program can exist in a finite number of states at any moment, as well as making it easier to show different "scenes" in the game, like the lobby or the game itself. This behavioral design pattern enables an object to modify its behavior when its internal state changes, effectively appearing as if the object has changed its class. Consequently, it facilitates easy modifications of entity behaviors through state changes, thereby boosting the game's modifiability. The pattern organizes the code and reduces the number of bugs, thereby improving usability. Moreover, it increases performance by eliminating the necessity for constant state checks, leading to smoother, more responsive gameplay.

Further on in the development process we decided to add a few design patterns. To improve our performance of the game we used the Object Pooling pattern to reuse Entities. Observers are used in a variety of Besieged, for example for providing callbacks to the buttons so that they gain the wanted behavior. 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 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.

In summary, the chosen design and architectural patterns not only align with the specific quality requirements of modifiability, usability, performance, and availability but also interconnect to support a robust, efficient, and user-friendly game architecture.

6 Architectural views

6.1 Logical views

The class diagram below titled figure 6.3.2 illustrates the system's implementation of Entity-Component-System (ECS). This pattern is commonly used in game development in order to manage game entities in an efficient and flexible way. This diagram provides an overview of the components, entities, and systems of the application, including how the different elements interact. Singleton is used on the manager.

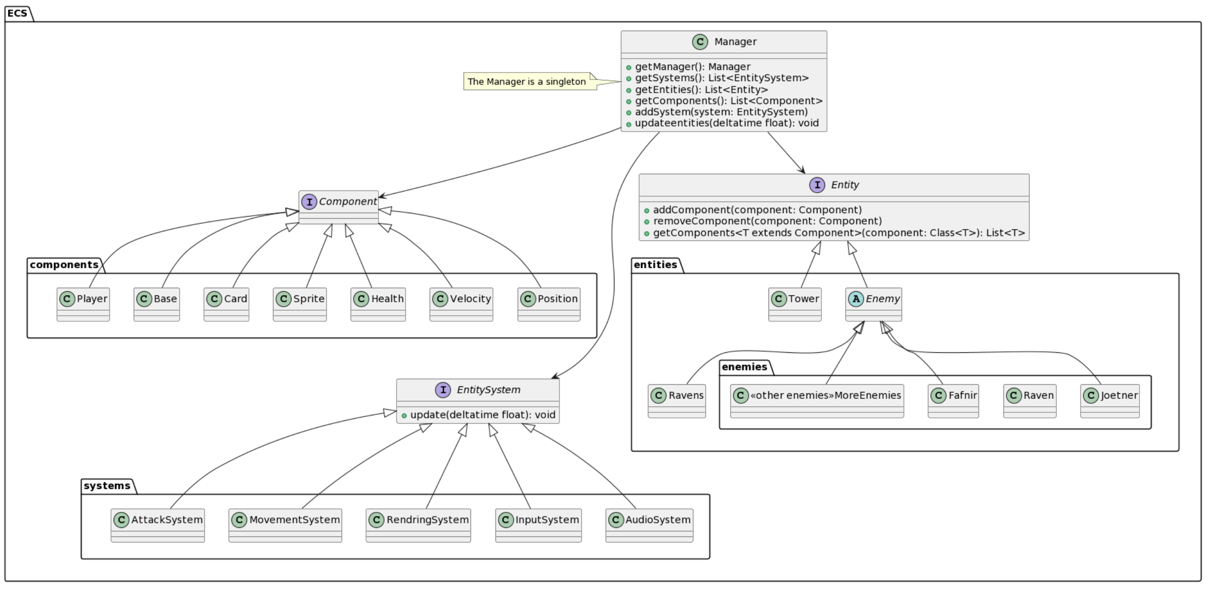

ECS
Manager
getManager(): Manager
getSystems(): List<EntitySystem>
getEntities(): List<Entity>
getComponents(): List<Component>
addSystem(system: EntitySystem)
updateentities(deltatime float): void
The Manager is a singleton
Component
components
Player
Base
Card
Sprite
Health
Velocity
Position
Entity
addComponent(component: Component)
removeComponent(component: Component)
getComponents<T extends Component>(component: Class<T>): List<T>
entities
Tower
Enemy
Ravens
enemies
«other enemies»MoreEnemies
Fafnir
Raven
Joetner
EntitySystem
update(deltatime float): void
systems
AttackSystem
MovementSystem
RendringSystem
InputSystem
AudioSystem

Figure 6.1.1: Class diagram: ECS pattern

The class diagram below describes our implementation of the Data Access Object pattern we intend to use. By doing so we abstracts the data source in a way that makes the system independent on what database is utilized, and enables us to be able to easily replace the database if needed. The DAOFactory class is used to build DAO objects, which are then used by the rest of the system.

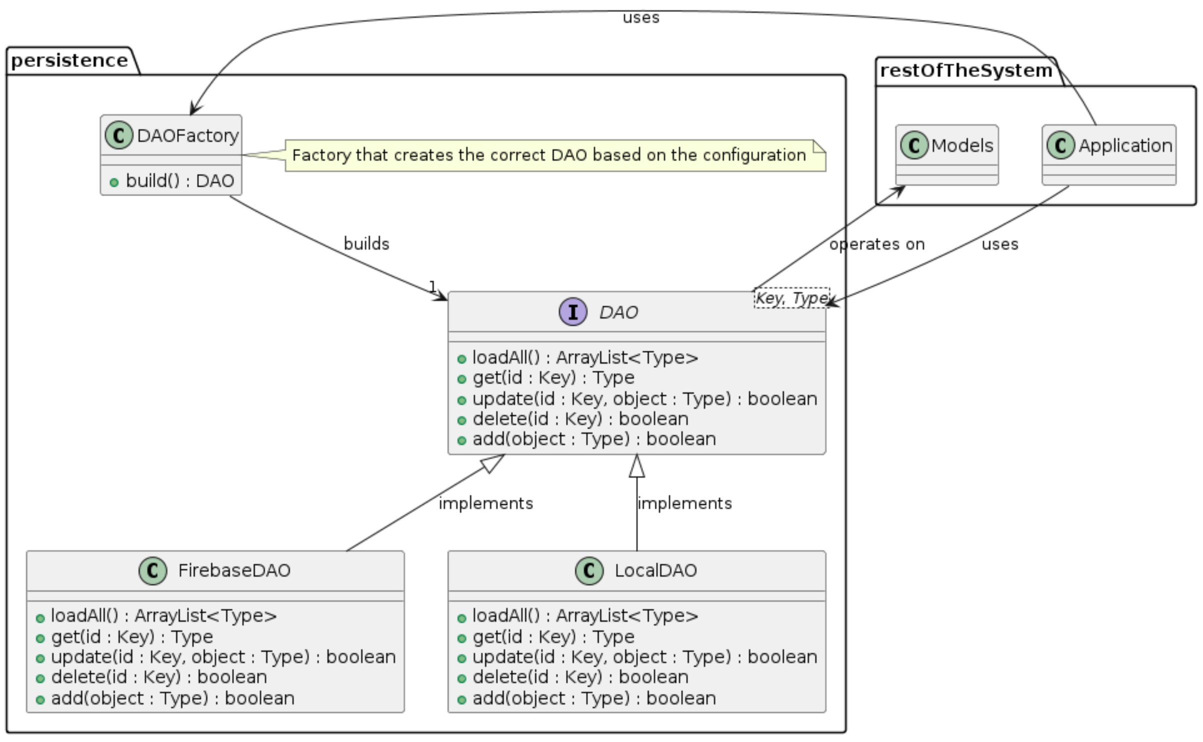

uses
persistence
DAOFactory
build() : DAO
Factory that creates the correct DAO based on the configuration
restOfTheSystem
Models
Application
builds
1
operates on
uses
DAO
Key, Type
loadAll() : ArrayList<Type>
get(id : Key) : Type
update(id : Key, object : Type) : boolean
delete(id : Key) : boolean
add(object : Type) : boolean
implements
implements
FirebaseDAO
loadAll() : ArrayList<Type>
get(id : Key) : Type
update(id : Key, object : Type) : boolean
delete(id : Key) : boolean
add(object : Type) : boolean
LocalDAO
loadAll() : ArrayList<Type>
get(id : Key) : Type
update(id : Key, object : Type) : boolean
delete(id : Key) : boolean
add(object : Type) : boolean

Figure 6.1.2: DAO pattern class diagram

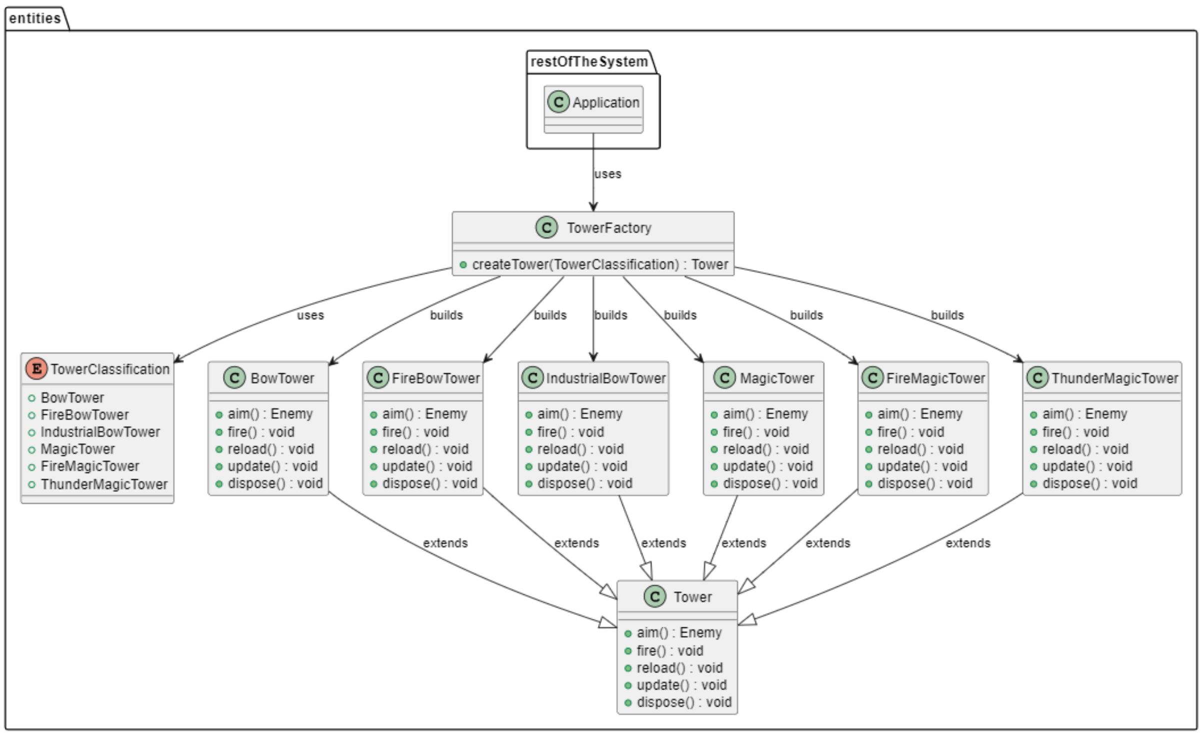


Figure 6.1.3: Factory pattern TowerFactory class diagram

The class diagram below details the structure and relationships of classes within the system, especially in the domain of networking. It provides a high-level view of the networking API's structure, illustrating messaging between client and server, and persistence within the application.

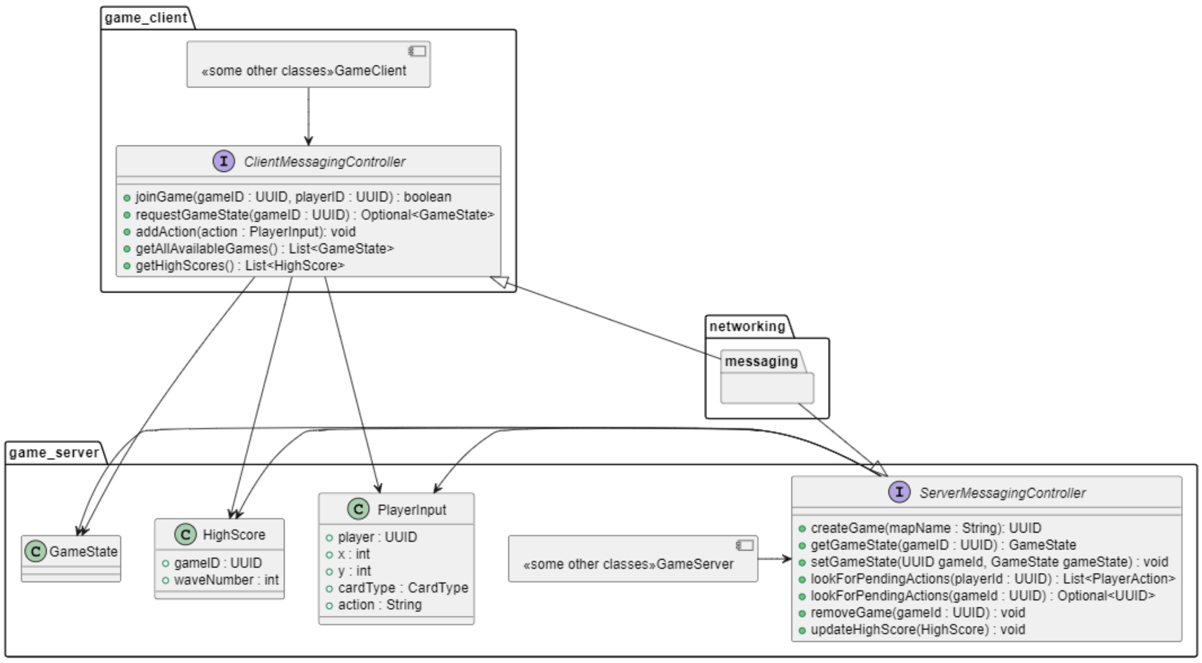

game_client
«some other classes»GameClient
ClientMessagingController
joinGame(gameID : UUID, playerID : UUID) : boolean
requestGameState(gameID : UUID) : Optional<GameState>
addAction(action : PlayerInput): void
getAllAvailableGames() : List<GameState>
getHighScores() : List<HighScore>
networking
messaging
game_server
GameState
HighScore
gameID : UUID
waveNumber : int
PlayerInput
player : UUID
x : int
y : int
cardType : CardType
action : String
«some other classes»GameServer
ServerMessagingController
createGame(mapName : String): UUID
getGameState(gameID : UUID) : GameState
setGameState(UUID gameId, GameState gameState) : void
lookForPendingActions(playerId : UUID) : List<PlayerAction>
lookForPendingActions(gameId : UUID) : Optional<UUID>
removeGame(gameId : UUID) : void
updateHighScore(HighScore) : void

Figure 6.1.4 Class diagram of networking API

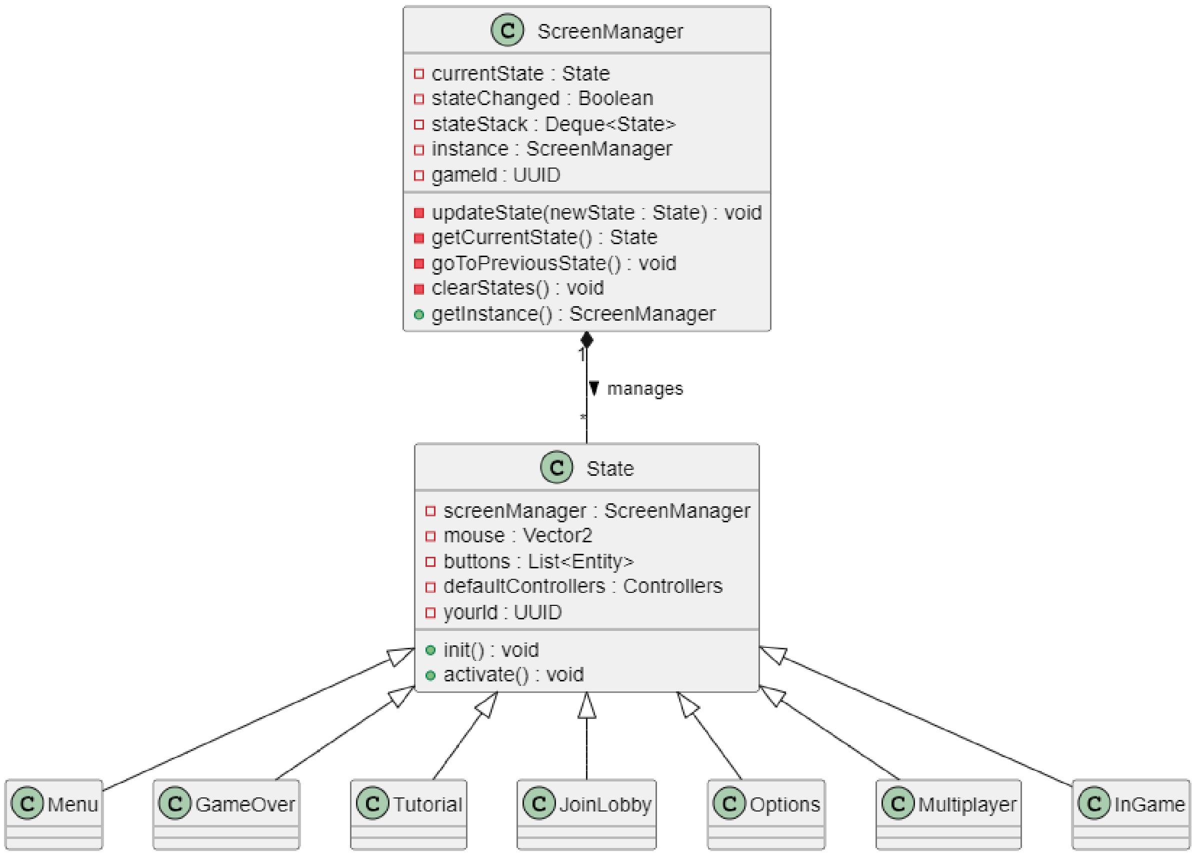


Figure 6.1.5 Class diagram of State pattern.

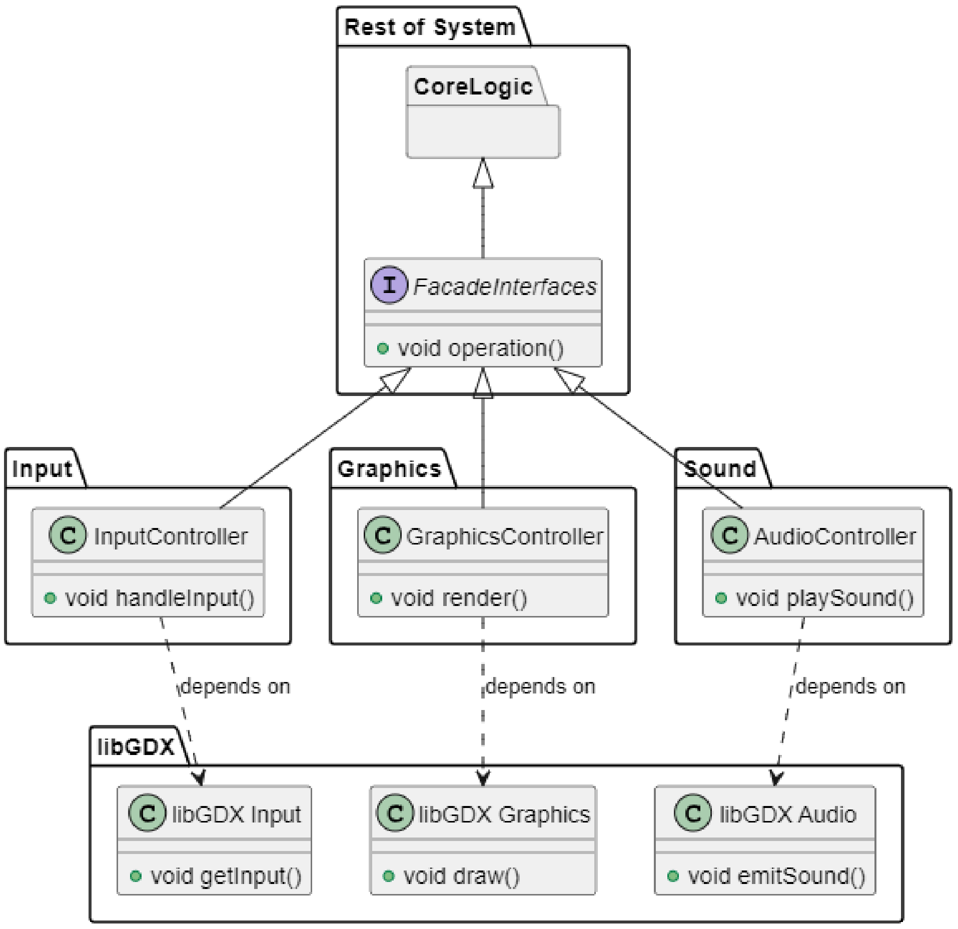


Figure 6.1.6 Class diagram of Facade pattern.

6.2 Process views

The state diagram shows all the possible game-menus, and maps the user journey. Logical progression between different states and player choices is depicted.

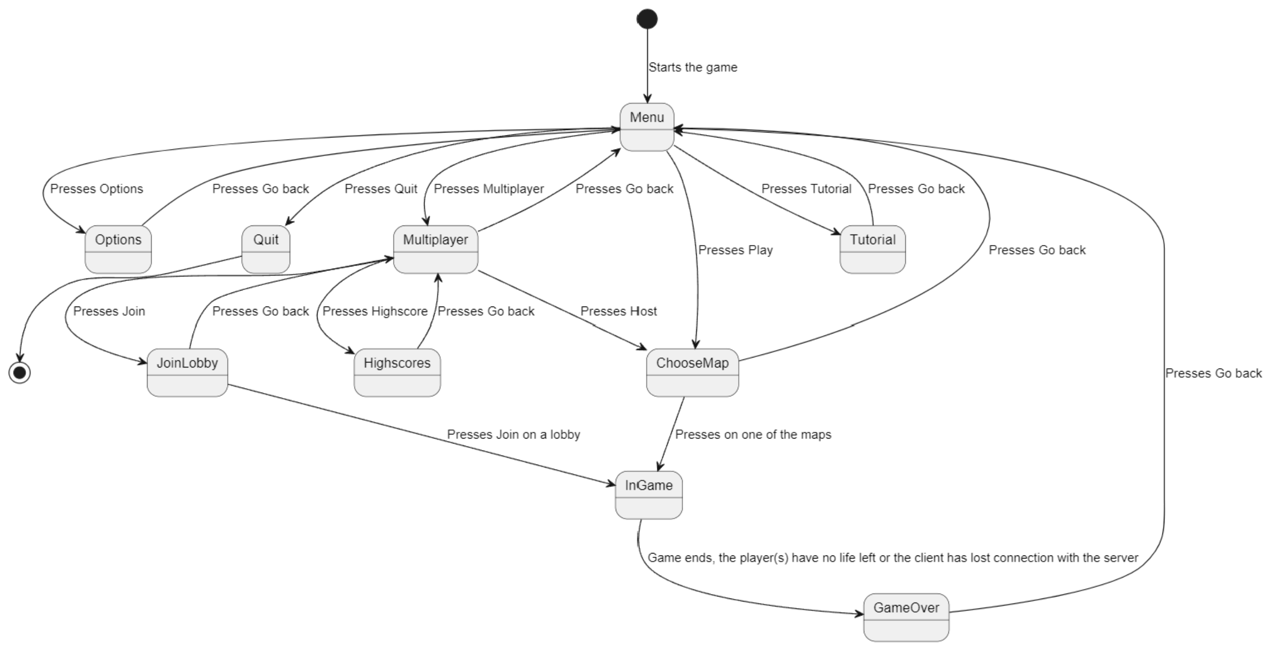


Figure 6.2.1 State diagram

The sequence diagram shows how the networking API works for players doing an action, and captures the interactions between client and server.

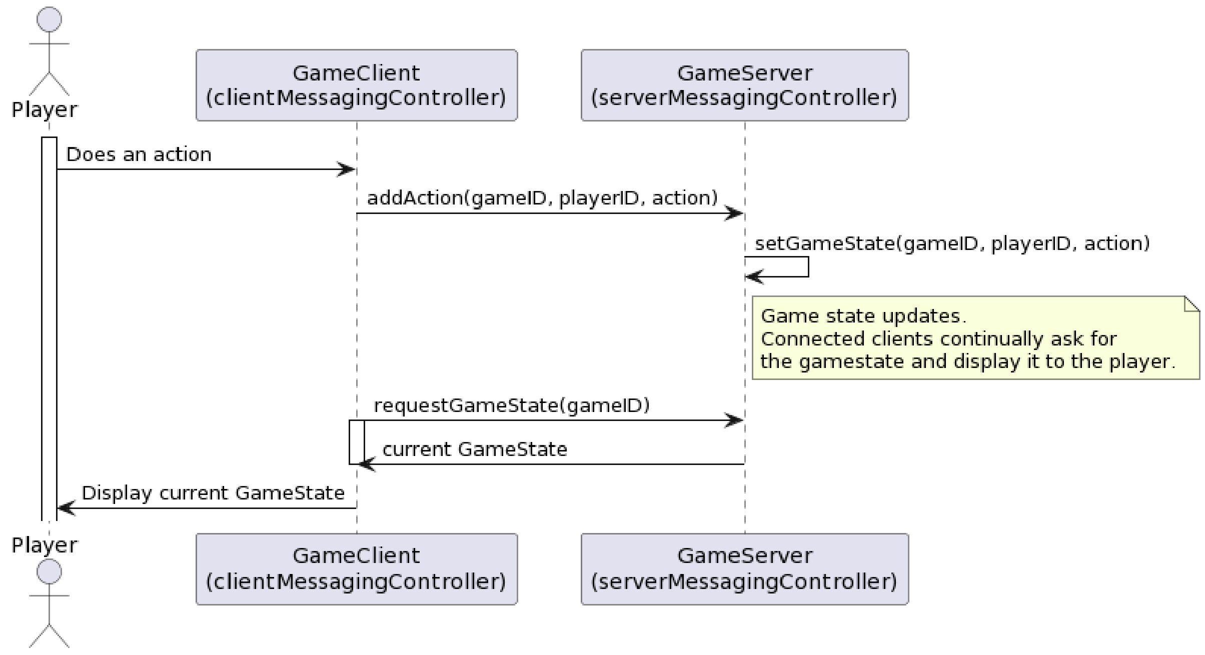

Player
GameClient (clientMessagingController)
GameServer (serverMessagingController)
Does an action
addAction(gameID, playerID, action)
setGameState(gameID, playerID, action)
Game state updates.
Connected clients continually ask for the gamestate and display it to the player.
requestGameState(gameID)
current GameState
Display current GameState
Player
GameClient (clientMessagingController)
GameServer (serverMessagingController)

Figure 6.2.2 Sequence diagram

The diagram below illustrates the abstract idea of how heartbeat is implemented in our system. However, this heartbeat is done indirectly by the client by asking the server for a new game state and checking the timestamp. If the last update from the server is more than 10 seconds old, then the client will go to the Game Over screen and show that it lost connection.

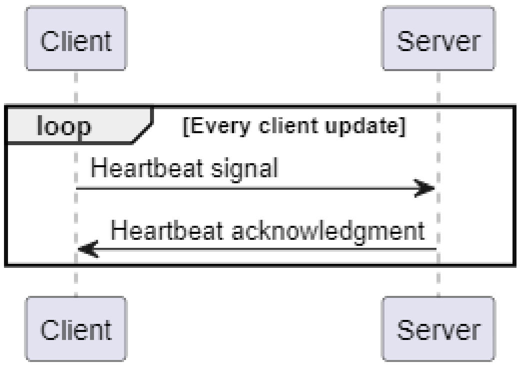


Figure 6.2.3 Sequence diagram

6.3 Development views

The view on Figure: 6.1.1 Package diagram of application describes the overall architecture of the application using a package diagram, illustrating the interconnectivity and relationships between the larger software components of the system, in a structured way that allows good division of labor for the developers. It is therefore applicable for a development view.

This package diagram describes the overall architecture of the application, illustrating the interconnectivity and relationships between the larger software components of the system. The launcher serves as the entry point for the application, and is responsible for initial loading of four key components it is dependent on: "input," "networking," "sound," and "graphics." These modules are utilized by the launcher to facilitate their respective core functionalities, and implements gameClient.

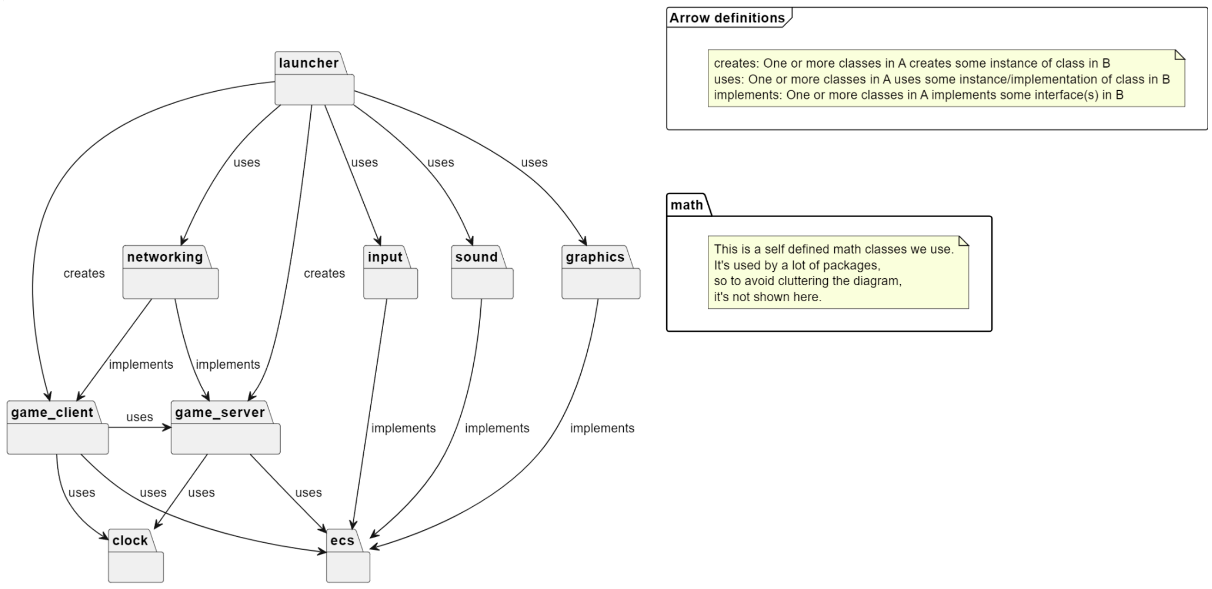

Arrow definitions
creates: One or more classes in A creates some instance of class in B
uses: One or more classes in A uses some instance/implementation of class in B
implements: One or more classes in A implements some interface(s) in B
math
This is a self defined math classes we use.
It's used by a lot of packages,
so to avoid cluttering the diagram,
it's not shown here.
launcher
networking
input
sound
graphics
game_client
game_server
clock
ecs
creates
uses
implements

Figure: 6.3.1 Package diagram of application

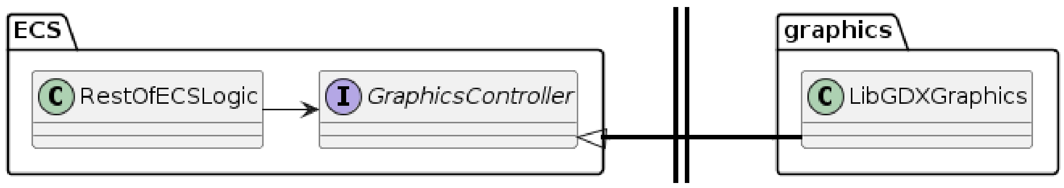


Figure 6.3.2 how the other packages implements ECS required interfaces

The layer diagram for the logic view describes the overall architecture, represented as a structure of three distinct layers: Presentation, Business, and Data access. The business layer encapsulates the core logic, rules, and operations of the application, processing and coordinating the application's functionality between the user interface and the data management component. The presentation layer provides the user interface, while the data access layer contains the networking and data management of the system.

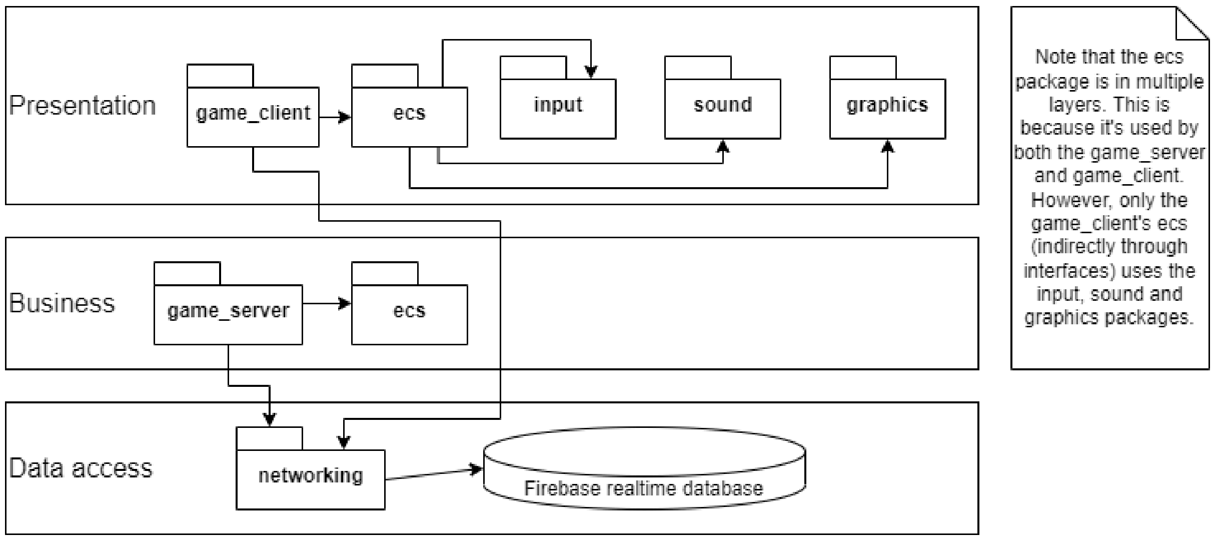

Presentation
game_client
ecs
input
sound
graphics
Business
game_server
ecs
Data access
networking
Firebase realtime database
Note that the ecs package is in multiple layers. This is because it's used by both the game_server and game_client. However, only the game_client's ecs (indirectly through interfaces) uses the input, sound and graphics packages.

Figure: 6.3.2 Layer structure

6.4 Physical views

The deployment diagram below serves as the physical view, and illustrates the physical deployment of artifacts. The diagram depicts how the application utilizes a client-server architecture, and how it is deployed on a mobile device, and how it, through the internet, interacts with the Firebase database.

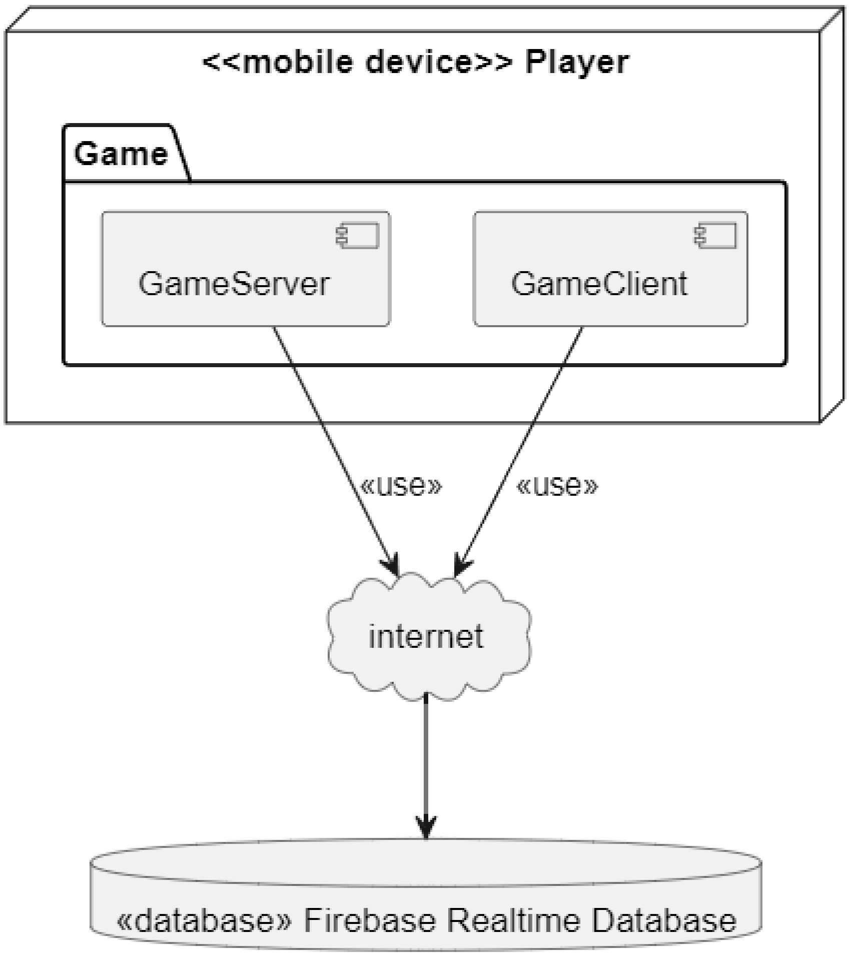


Figure 6.4.1: Deployment diagram

6.5 Scenarios

Below is a sequence diagram of a scenario depicting lobby activity. One person creates a lobby, shares the passcode with a friend, and then the friend joins the game using the passcode.

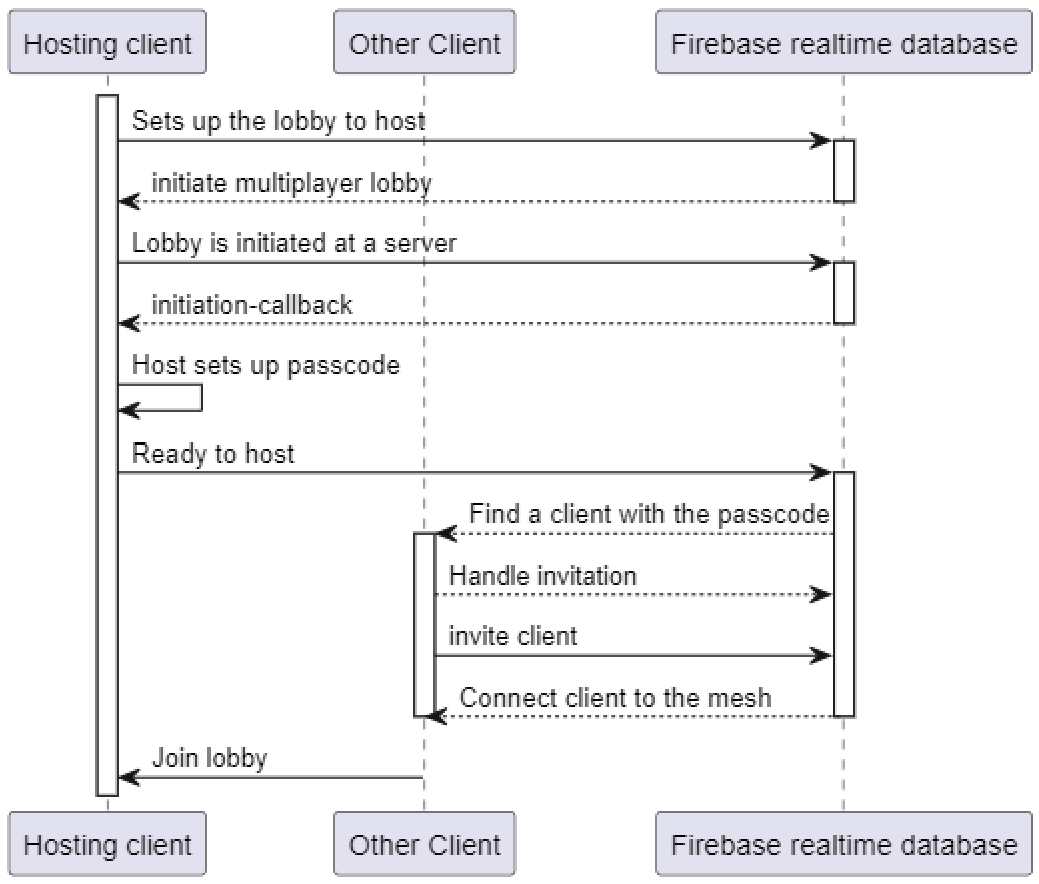

Hosting client
Other Client
Firebase realtime database
Sets up the lobby to host
initiate multiplayer lobby
Lobby is initiated at a server
initiation-callback
Host sets up passcode
Ready to host
Find a client with the passcode
Handle invitation
invite client
Connect client to the mesh
Join lobby
Hosting client
Other Client
Firebase realtime database

Figure 6.5.1: Sequence diagram: set up lobby

7 Consistency among architectural views

There are some simplifications in all of the diagrams to make them more understandable. This can be seen for example in Figure 6.3.2: “DAO pattern class diagram” which illustrates the development views, and has the abstraction “RestOfTheSystem,” which is described as a package in the diagram. This is not a package calledRestOfTheSystem, but an abstraction of the rest of the system in order to make the diagram more readable and intuitive. This is also the case in "Figure 6.1.4 “Class diagram of networking API”, describing the logical view. This figure utilizes the phrase “some other classes” in the game client package and game server package in order to avoid unnecessary details. In figure 6.1.6 we also use some simplification, as there isn’t an interface named FacadeInterfaces. These are interfaces with corresponding names to the packages of the respective Facade. To make the figure more intuitive we chose to show it as just one interface with a name to clarify it is actually multiple interfaces.

Another thing to take in consideration is the similarities between *Figure 6.5.1: Sequence diagram: set up lobby* in the Scenario, and *Figure 6.2.2 Sequence diagram* in the Process view. They are quite similar, but the fact that *Figure 6.5.1: Sequence diagram: set up lobby* is a direct abstraction of a functional requirement, it belongs better as a Scenario view than in the Process view.

8 Architectural Rationale

In the design of the architecture for our game, *Besieged!*, we first and foremost focus on modifiability as our primary quality attribute. This is due to software being ever evolving. To accommodate new needs, our architecture must be flexible, allowing for easy modifications, updates, and expansions without extensive rewriting of code. Therefore it is extremely important to make the pieces of the software that are likely to change easy to change.

To make our game architecture as modifiable as possible we have decided that we are going to abstract LibGDX and Firebase from the core logic of our game. We made this choice because Robert C. Martin's book *Clean Architecture* mentions how frameworks, database-solutions and other similar technologies are implementation details that should be used as tools for applications instead of being used to design the architecture of an application[2]. This way of creating and maintaining the architecture ensures that the game's core logic can evolve independently of any specific implementation details.

We recognize that LibGDX and Firebase are essential parts of our game, but by treating them as implementation details and not depending on them directly by abstracting them, we can use them as interchangeable tools instead of them being essential parts of our game's architecture. This makes our game more modifiable as we can use different COTS-components or frameworks as long as they implement the abstracts we have defined. This means that if we want to change from LibGDX to another framework we are free to do so, without changing the core logic of the game. If LibGDX was an integral part of the core logic of the game, changing from LibGDX to another framework would require us to rewrite the entire codebase to conform with the new framework. This ensures that our game is very flexible and modifiable even if we were to for example change frameworks or database-solutions in the future.

We recognize that this project is just a prototype, but we want to create an architecture that would be applicable for a real project that could be fully finished. With this in mind, it is possible that we would need to change databases or frameworks in the future and this makes them more likely to change, which is why they are abstracted away. This is why most of the packages, like input, networking and

[2] Martin, 2018, p. 202

sound implement abstractions, i.e. interfaces or abstract classes, which are defined in the gameClient and gameServer packages, as they contain the core logic of the game and thus they need to define what they need from the other packages. Thus the more volatile packages containing the implementation details depend on the more stable ones.

Besieged! uses the state pattern for the different game menu screens seen on Figure 6.2.1. This makes it easy to add new game screens if it is required, as one only needs to create a new state and the required behavior for the new state. There were discussions on rather using MVC for this aspect but this architectural pattern would introduce more complexity to the application than it might require to solve the problem.

The main gameplay has by its nature a lot of intricacies and complexity, therefore it needs some scaffolding to keep the complexity at bay. The Entity Component System (ECS) architectural pattern helps us improve our modifiability and performance. This pattern is used in other well established games in the game development industry as Game engines like Unity uses it. This pattern facilitates easy creation of new game tokens like enemies and towers by reusing components and entities and combining their attributes and methods.

The Data Access Object (DAO) design pattern abstracts away the data storage mechanism. This creates an architectural boundary between the database and the core logic of the application. The use of the DAO pattern reduces coupling in the application making the choice of database a small implementation detail as one can change from one database to another without changing other logic in the program. This makes it possible to be "independent of database"[3] and limit the coupling from COTS-software like Firebase.

The game also uses a client server architecture to address its availability concerns. All the devices that are part of the game have a GameClient running on them. The player that hosts the game will also have the server on their device. The common convention is often to have an external server running on better hardware than a personal computer. However, to guarantee that there is always a server running, one of the players will host the server themselves. This improves the availability at the cost of slightly lower performance for the player that hosts the server, but we believe this is a trade-off that can be tolerated when there are only two players per game.

[3] Martin, 2018, p. 202

9 Changes

Table 10.1: Changelog

Date	Change History	Comments
March 03, 2024	First released version	None
April 20, 2024	Switch logical and development view diagrams	As per the feedback, we switched some of the diagrams in the logical and development views.
April 20, 2024	Updated package diagram	Did this so that it better reflects the changes that were made under development to let the architecture fit our needs better. Like the fact that most of the game logic will be in the ECS manager, and since it's responsible for rendering things, it should define graphics interfaces.
April 20, 2024	Updated layer diagram	Did this so that it better reflects how the packages are using each other. For example both the game_client and game_server use the ecs package.
April 20, 2024	Updated class diagram of networking API	Added more methods to the ClientMessagingController and ServerMessagingController
April 21, 2024	Updated Increase Cohesion-section under Modifiability-tactics.	Added reference to the Client-Server Pattern. Updated description of where logic is

		placed by including ECS along with GameClient and GameServer. This was done because we got feedback about it.
April 21, 2024	Moved Stable Dependency Principle up to Modifiability-tactics section	Moved from Architecture Rationale-section. This was done because we got feedback about it.
April 21, 2024	Write about Dependency Inversion Principle	We did this because of feedback.
April 21, 2024	Reformulated game description	The game description needed to be clearer about how multiplayer works in our game
April 21, 2024	Add facade pattern class diagram	We did this because of feedback.
April 21, 2024	Added text about missing architectural and design patterns	None
April 21, 2024	Write about the Open-Closed Principle as a modifiability tactic	We did this because of feedback.
April 21, 2024	Write about the Single Responsibility Principle as a modifiability tactic.	We did this because of feedback.
April 21, 2024	Write about the Liskov Substitution Principle as a modifiability tactic.	We did this because of feedback.
April 21, 2024	Write about the Interface Segregation Principle as a modifiability tactic.	We did this because of feedback.
April 21, 2024	Fixed even line spacing	None
April 21, 2024	Updated state diagram	Updated the state diagram to show the current possible states in the game. We realized that the states we had previously

		did not make fully sense, so we updated it.
April 21, 2024	Updated the heartbeat diagram	We updated the heartbeat diagram because we did not have time to implement heartbeat from server to client and the other way around. So we were only able to make the client check if it still had connection to the server, if not it says that it lost connection and the player can go to the main menu.

10 References

Rick Kazman, Len Bass & Paul Clements. *Software Architecture in Practice* - Fourth Edition. Addison-Wesley, 2022.

Robert C. Martin. (2019). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Pearson.

Firebase, (2020, April 9) *Service Level Agreement for Hosting and Realtime Database*. Firebase. https://firebase.google.com/terms/service-level-agreement

Unity, (2024) *ECS Concepts*. Retrieved from: https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_core.html

Oracle, (2024). *Data Access Object*. Retrieved from: https://www.oracle.com/java/technologies/data-access-object.html

Nystrom, R. (2024). Game loop: Sequencing patterns in game programming patterns. Retrieved April 20, 2024, from https://gameprogrammingpatterns.com/game-loop.html

11 Individual Contribution

Table 12.1: Individual contribution

Name	**Nature of work**	**Hours**

Jens Martin Norheim Berget	● 2.1 Functional Requirements- ASRs ● 2.2 Quality Requirements - ASRs ● 2.3 Business Requirements ● General cleaning up of text and improvement of sentence formulation	9 hours
Magnus Vesterøy Bryne	● 3.2 Architectural viewpoints ● 5 Design patterns and Architectural Patterns ● Fixed viewpoints based on feedback ● Proofreading ● Writing further about patterns	12 hours
Mattias Tofte	● Architectural Drivers / Architecturally Significant Requirements (ASRs) ● Design patterns and Architectural Patterns ● 2.1 Functional Requirements - ASRs ● Proofreading	8 hours
Sverre Nystad	● Writing on Architectural Design document (6 hours) ● DAO pattern logic view (1 hour) ● Working on networking views (2 hour)	9 hours
Tim Matras	● Writing on Architectural Design document ○ Introduction (1.5h) ○ Stakeholders and concerns (2h) ○ Working on the different views (diagrams in git) (3.5h) ○ Working on Architectural tactics (3.25h) ○ Working on the different views (9.5h) ○ Working on Architectural Rationale (2.5h) ○ Updating diagrams (1.5h) ● Reading through the document and noting where we need to make changes after feedback (0.5h) ● Updating diagrams (2.5h)	26.75 hours
Tobias Fremming	● DAO pattern logic view (1 hours) ● Development view: factory (1 hours)	12 hours

	● 2.3 Business Requirements ● Write on Architectural Design document and discuss architecture (1.5 hours) ● write (5.5 hours) ● Write (3 hours)	