System design document för spelet Chans

Måns Josefsson, Felix Holmesten, William Andersson, Vilhelm Hedquist och Oscar Johansson

23/10/2020

1 Introduction

In this document we will give insight to how our application Chalmers Chans works. Two important points in the SDD are system architecture and system design.

The application is an altered adaptation of the famous board game Risk with the unique twist that you play as student divisions from Chalmers and on its different campuses. The game has support for up to 16 concurrent players in the same game and the gameplay is fully functional.

1.1 Definitions, acronyms, and abbreviations

Risk
A classic board game, which revolves around conquering the world and battle against opponents for each other's territories using dice. (For more information: https://en.wikipedia.org/wiki/Risk_(game))

Chalmers Chance
The name of the game.

Student division
A student division in the context of the game is a local organization of Chalmers student union, where each of the 16 student divisions are connected to a unique study programme.

Space
A button on the map which gets its name from the location it is placed at on the map. The owner's color and amount of units is displayed on the space.

Area
A collection of **spaces**. If a player owns all of the **spaces** of an area it will receive more units to deploy on their next turn.

2 System architecture

There are three important packages in the project that controls the application. Model, view and controller. Model holds the game loop and all the data in the project. The view contains two classes, MapView and AttackView. MapView creates the different scenes, colors, texts and everything GUI related. AttackView is fairly simple, it displays all the relevant information when an attack is made. The controller is the biggest package, it controls when the different scenes are supposed to change and what's supposed to happen when clicking the different buttons but mostly, the controller controls the view.

The normal flow of operation for the application is as follows: The first scene in the GUI is the start menu. There the user can choose to either start a local game, start an online game, or quit the game, the usual choice that the user takes is to either start a local game or an online game. Next the user either sets up the local game or connects to the server and sets up an online game. When the set up is done, the game can start. The next scene is the map and all the spaces. The game then goes on to the three phases that a player can do in one round. Usually, the first is "Deploy" where the user deploys units. The second one is "Attack" where the user attacks another space. The third one is "Movement" where the player gets to move units. That concludes one round for one player and then the turn goes over to the next player in line. The game is won when one player owns all spaces.

Except for the normal flow there are also some more scenes in the game. There is a pause menu where the players can quit the game which terminates the application, start a new game or quit to the start menu.

Furthermore, there exists a server package which is responsible for hosting the online version of the game.

The server package consists of a main class called ServerManager, which manages all the connected clients and their requests, a controller package and a model package. The model package consists of three classes: abstract Lobby, MenuLobby and GameLobby. The controller package consists of one class named LobbyController which uses two lists, one for GameLobby and one for MenuLobby.

The online mode flow of operation for the application is as follows: When a player presses the start online button it tries to connect to the server and if successful it receives all available lobbies. When a player has chosen a lobby, the server receives which lobby was chosen and then assigns that lobby to that client. When 2 or more players have chosen the same lobby, the server will notify the lobby leader(first player in lobby) that the game can be started and that person's start button lights up and can then start the game. Then the game works just like single player mode except each client sends over every single action taken to the server and the server notifies all the other clients so they can update to a correct state.

3 System design

The project is using an MVC structure as can be seen in Figure 1. However because all changes in the view are happening when the users click on buttons or spaces it was decided

to change it up a little. As of right now the controller depends on the view and the model, but the controller is also the one to initiate change in the view. Which means the view knows when to change and what changed, and can then easily call the model for the new information. The most important thing to note is that the model doesn't have any dependencies to the other packages. This means the model can be used as a stand alone package, with other controllers and views. The controller is the biggest package in the project, because of all different views and user input. The use of JavaFX and fxml-files also makes the controller classes bigger.

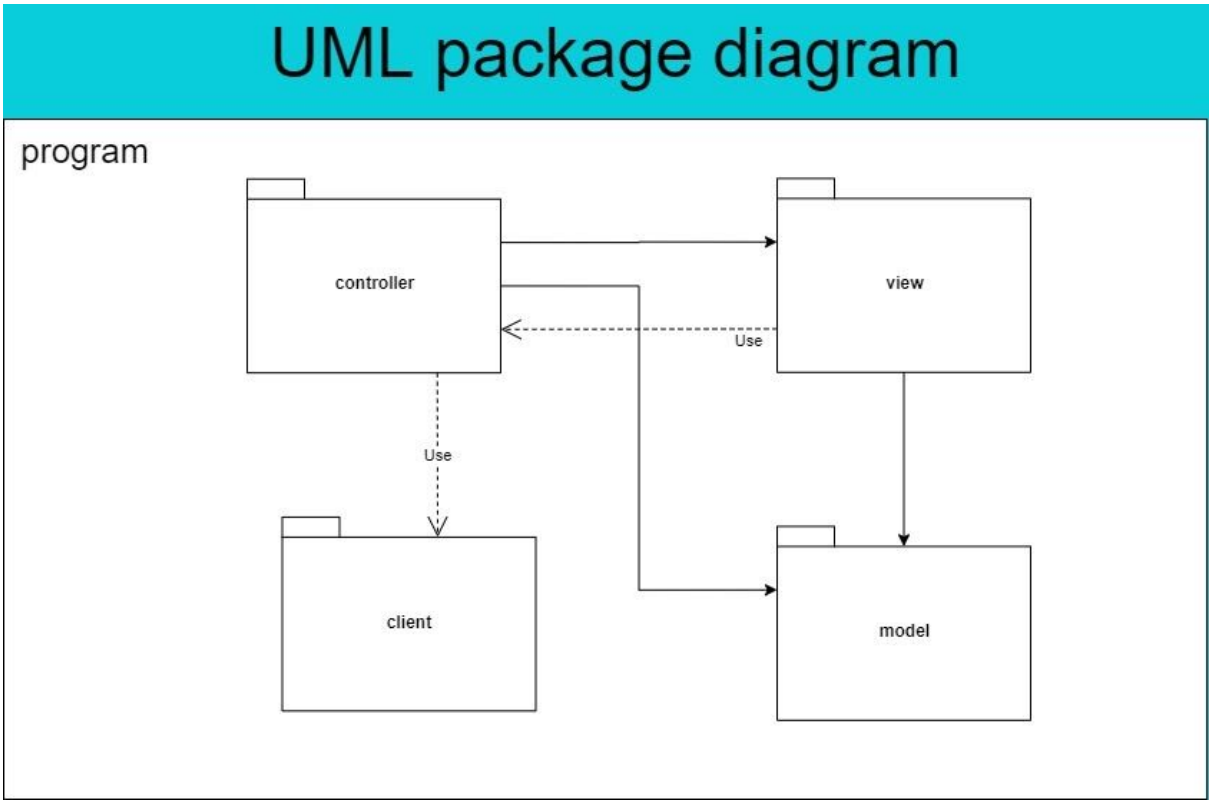

Figure 1: UML package diagram from the application

In Figure 2 an UML-diagram of the model package is seen and this is also the design model of the application. When the design model is compared to the domain model the clear relation between the two is that they consist of the same classes, with small modifications. The main idea between the two are the same, but the design model also contains implementations of design patterns and technical aspects of the code that are not necessary to understand the full picture of the application, but are vital for the implementation of the application. Some clear examples of this are the interfaces seen in the design model. The design model on the other hand lacks the multiplicity from the domain model, because multiplicity is more important for the understanding of what the application does, then how it is implemented.

UML-diagrams of all packages in the application can be seen in the Figures below.

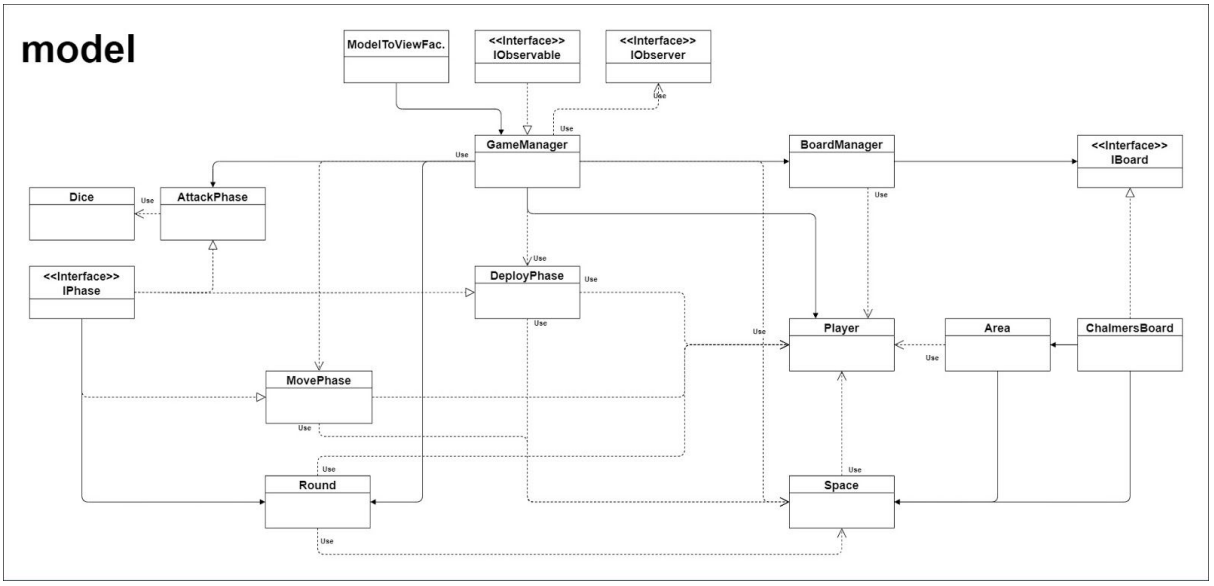

Figure 2: Dependencies in the model package.

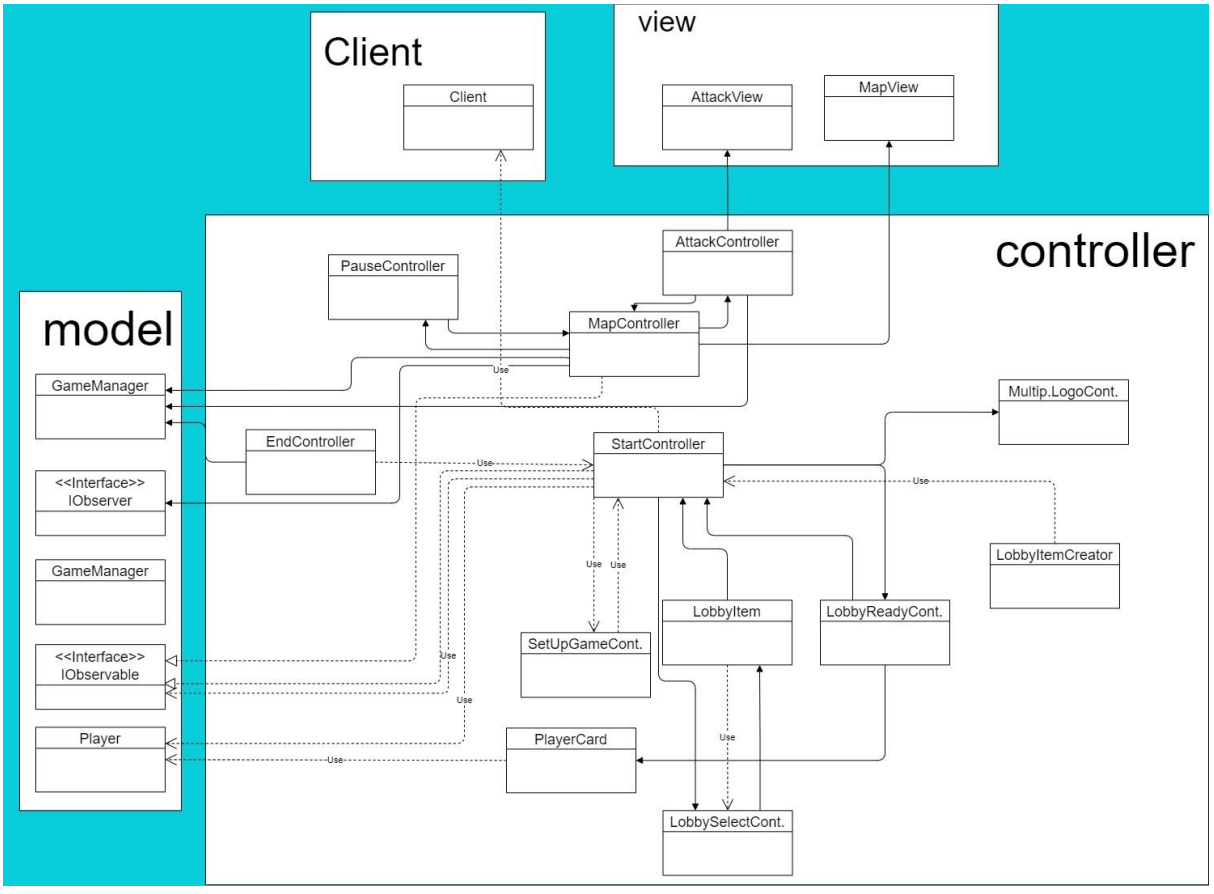

Figure 3: Dependencies in the controller package and dependencies to other packages.

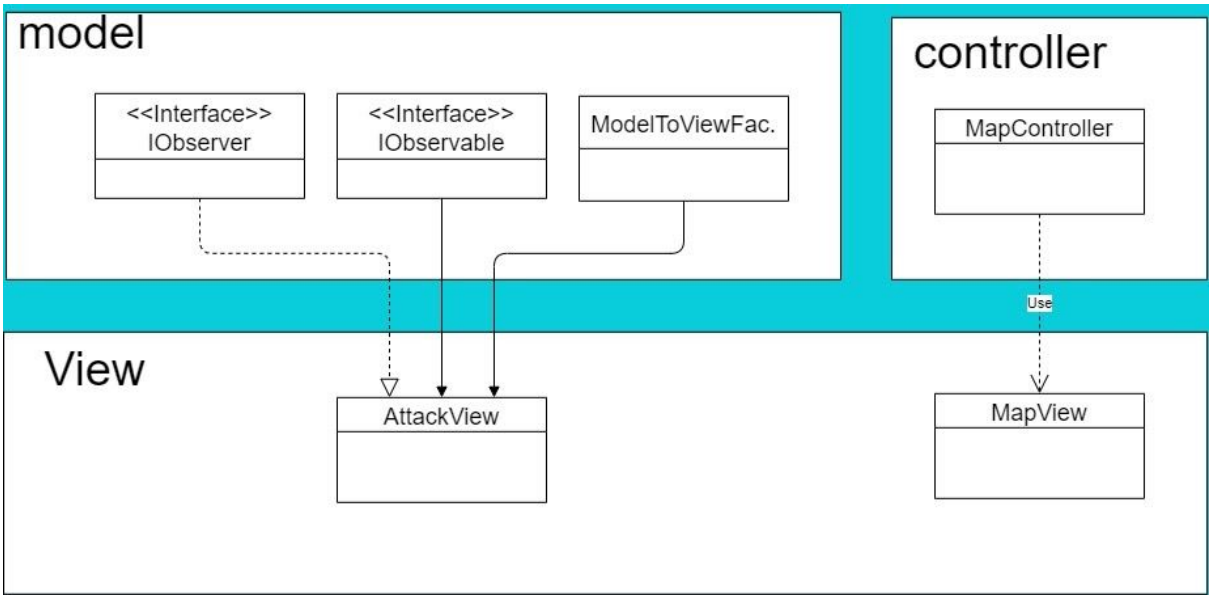

Figure 4: Dependencies in the view package and dependencies to other packages.

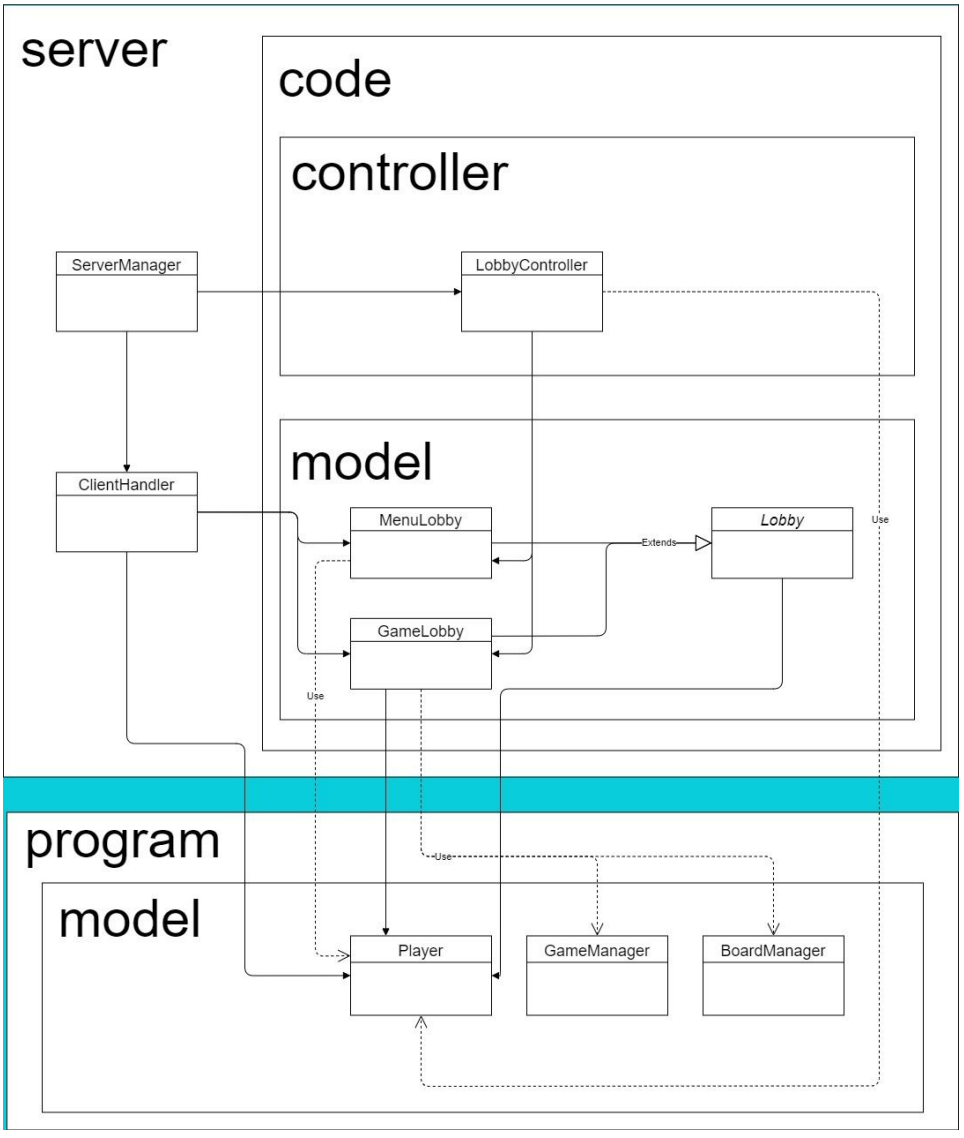

Figure 5: Dependencies in the server package and dependencies to other packages.

The main class of the application is Chans and is located outside of all packages. Chans initiates the application and creates the first stage, where it sets the StartController as the root of the stage.

The design patterns that primarily has been used in the project are as follows:

Observer pattern
The interface IObservable makes it possible to send information to an IObserver without making the IObservable dependent on the IObserver. The way this is done is by having the main model class implement IObservable and create a list of IObservers. It then notifies the observers when certain things happen. In this project it is only the Client that's listening to this information.

State pattern
The state pattern is used in coordination with IPhase. There are three classes that implement IPhase. DeployPhase, AttackPhase and MovePhase. These three classes all use the same methods from the interface but mainly the method startPhase. That method changes in behaviour depending on what state(phase) the game is in. The interface also has a method called nextPhase() which is used to go to the next phase. In setUpPhase() in GameManager the order of the phases are hard coded.

This pattern is used to encapsulate the varying behavior that an object can have based on its internal state. This is a clean way for an object to change its behaviour without having to depend on if statements. This would also improve the maintainability of the code.

Facade
The intent of this pattern is to hide implementation details with the help of a higher level interface that is easier to use than the subsystem. The class ModelToViewFacade is the start of this. ModelToViewFacade holds relevant methods for the view package and makes the dependency between the model and view clearer and therefore also weaker. This way the implementation can be modified easier without causing unsuspecting errors in the view. This pattern is however not implemented to its fullest intent in our application.

4 Persistent data management

Images
The only persistent data that is used in the application are images which are stored in a resource package within the source code.

5 Quality

The application is tested with JUnit-tests. All non-trivial public or package private methods should be tested with at least one test. The tests can be found in the test-package.

PMD was used to test the quality of the code and changes were made with the comments in mind.

List of all known issues:

- Analytical tool to show dependencies (eg. STAN or similar) hasn't been used yet. There were a lot of problems with getting Maven to work so there was little time to try and integrate STAN in the end.
- Sometimes when two players join the online lobby at the same time there is a bug where they both can end up picking the same team.
- There is a bug where the full screen sometimes toggles when going into the game.

With more time on this project the following features would've been added:

- Make the online lobbies better.
- Make a player able to choose to either host or join a lobby.
- Make it possible for all players to see who's ready in the lobby and who's not.
- Lower dependency in the project. Especially when it comes to the server, there was only enough time to make it work and not enough time to really care about all the dependencies.
- The server is incorporated in the code. However since the server was added to the project later there was no time to incorporate it into the domain model, design model or UML and such which would've been good.
- If a player leaves the game that player should be removed so that the game can keep going.
- A player should be able to surrender the game

6 References

JavaFX
Library used to create the GUI of the application. (See: https://openjfx.io/)

Scene Builder
Integrated with JavaFx and used to design the GUI with simple Drag and Drop features. (See: https://gluonhq.com/products/scene-builder/)

IntelliJ IDEA
IDE used for programming. (See: https://www.jetbrains.com/idea/)