System design document for Group 28

Table of Contents

Version: 1.0
Date 22/5 - 2013
Author: Christoffer Medin, Anton Myrholm, Pontus Malm, Viktor Karlsson, Pontus Doverstav

This version overrides all previous versions.

1 Introduction

This is the system design document for group 28s project for the course TDA367. The project aims to create an online multiplayer version of the card game Dominion.

1.1 Design goals

Our goal is to create an fully functioning game that provides a fun experience for the user. We aim to create an application that is easy to use, visually appealing and provide the user with the full entertainment of a game of dominion without concerning the user with the boring parts such as shuffling set-up and clean-up.

We aim to only implement the basic version of Dominion, but build the application so that further expansion can be implemented with relative ease.

1.2 Definitions, acronyms and abbreviations

- MVC, Model-View-Controller, is a design pattern that is supposed to split an application with a GUI into three distinct parts. Where model, view and controller are the diffrent parts.
- GUI, graphical user interface, the graphical representation of a program.

2 System design

2.1 Overview

The application was developed to give a simple and enjoyable experience of a game of Dominion. It was designed to make it easy for a user to play a game of Dominion with friends and enemies. The application also aims to reduce the time the user spends setting up and cleaning up as well as remove annoying elements such as shuffling mid turn from the concern of the player.

The project will be developed using a MVC model, with the view and controller located at the users computer and fully separated from the server where the model is located.

2.1.1 Model functionality

The model is exposed in the *game* package. Since this includes some data structures with somewhat customized behaviour, the funtonality is divided between various classes representing this. The cards, which are an integral part of the model, are in a separate package and are further mentioned below. Other than various data structures, there are classes taking care of turns, gaining card, playing card etc.

2.1.2 Rules

The game itself doesn't have many rules. The only thing you have to follow are the various phases of a turn, and this is enforces via a TurnHandler class. The rest of the rules are directly tied to the cards, i.e. what happens when a card is played. These rules are represented in the application by classes representing each cards, which are played through a class called CardRulesHandler, which ensures that every card is played properly and that every player is affected as it should be.

2.1.3 Commons

Since the server and the client needs to share certain parts of the program, such as messages that are sent over the network and listeners, we introduced the *commons* package. The goal of this was to decrease the coupling and dependency between the two and create a clearer line between the two, to really create a standalone server from the client.

2.1.4 Cards

Cards are an integral part to the program as it is a digital version of a card game. Since each card has a unique funtionality, every card has been implemented as a class describing its funtion. For more information, see *Rules* above.

2.2 Software decomposition

2.2.1 General

See figure below for the different modules, and appendix for dependencies of individual packages.

- **Client: The part of the program that every user will be running, which works partly like a view, representing the model, and partly as a controller, modifying the model**
- Main: Starts a new instance of the client
- Controller: Controls the whole game by having listeners for the whole network
- Model: Contains all logic needed in the client, such as joining rooms and sending data to the server. Also contains clientwide settings such as playername and chosen options like resolution.
- Network: Contains the connection which links the client and the server
- View: Contains everything graphical about the client
- **Commons: Contains everything that needs to be shared between the client and the server**
- Messages: Contains all the messages that are to be sent between the clients and the server
- Network: Kryo needs to register every class it's going to use, this is done in this class
- Listener: Contains listeners and events for gamewide eventhandling
- **Server: The server which handles connections to players and the model**
- Cards: Contains classes for all the cards and defines how they interact with and affect the game
- Main: The main class for the server, creates a new instance of the server and starts it
- Game: Contains the model for the whole game, which controlls and manipulates everything, including ut not limited to turns, players, gamerooms, card rules
- Network: Provides fresh instances of the server and connections to players (clients)
- View: Serves as an input and output for the server

2.2.2 Decomposition into subsystems

There are two subsystems to this application: the server and the client.

- The client is what the end user will receive when he/she acquires the application and contains the view and the controller.
- The server is what will be running on a server provided by us and will contain the model.

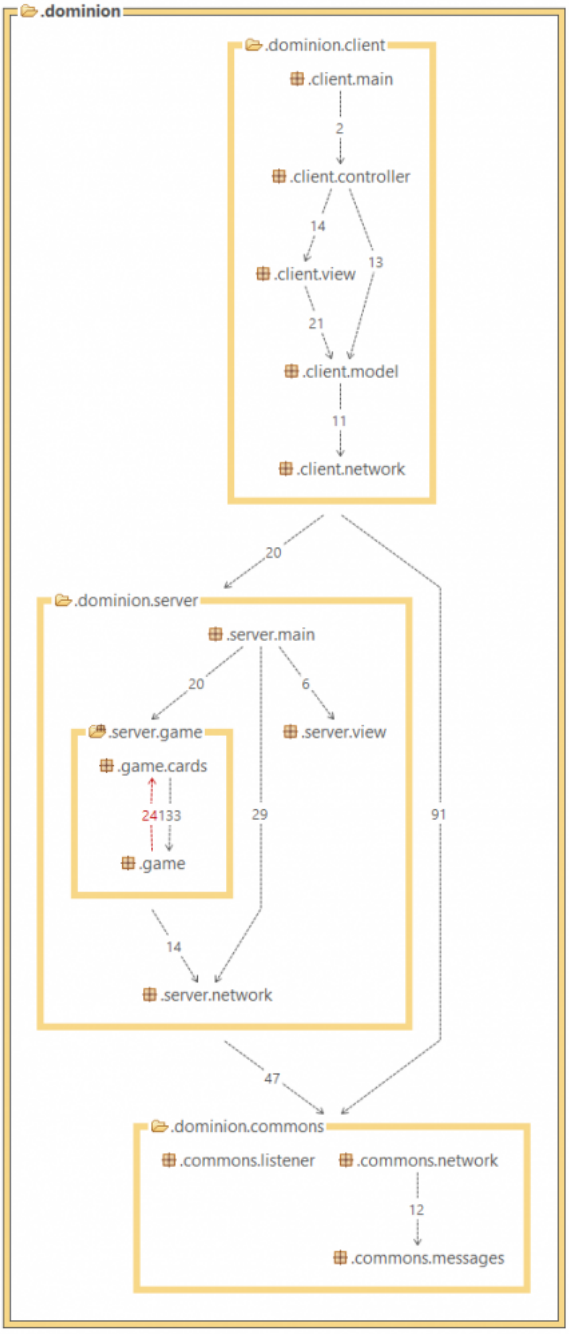
2.2.3 Layering

Note that the server and client are standalone applications, meaning that they essentially are in the same level of the layer. Commons is filled with classes and code that is shared between server and client, and thus is on a lower layer. Otherwise, see figure below.

2.2.4 Dependency analysis

See figure below

Dependency of packages and layering
For more in-depth analysis, see appendix



2.3 Concurrency issues

The only concurrency issues we might have are clients trying to send messages to the server when they are not supposed to. This is solved by the server simply not listening to a players connection when it isn't their turn.

The application should not be able to reach a deadlock other than a user not proceeding with his/her actions.

2.4 Persistent data management

The persistent data that exist is:

- Card info
- Card images
- Settings

Card info includes most of the data concerning the cards such as cost, type, name etc. These are stored in an txt file on the server.

The images are required to draw the graphical parts of the application and are stored in the loss less format PNG.

The settings include things like resolution, full screen etc. and are saved in an txt file. The settings are saved so that the user wont have to set up the program every time he/she wants to use it.

2.5 Access control and security

We have separated the client and the server in order to reduce the risk of malicious users messing with the application. We have done this by keeping the model on the server and letting the user have access to the view only.

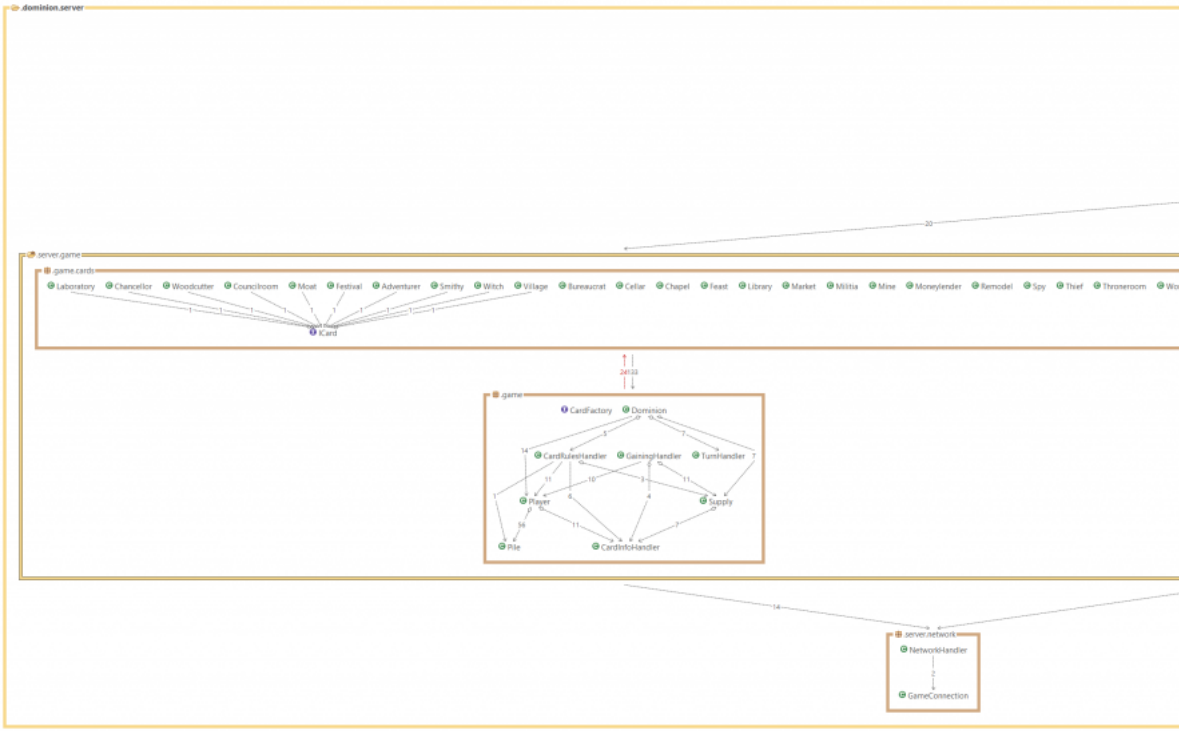
2.6 Boundary conditions

We do not aim to create all the expansions to the game, nor adding support for any cards requiring too much change to the original game.

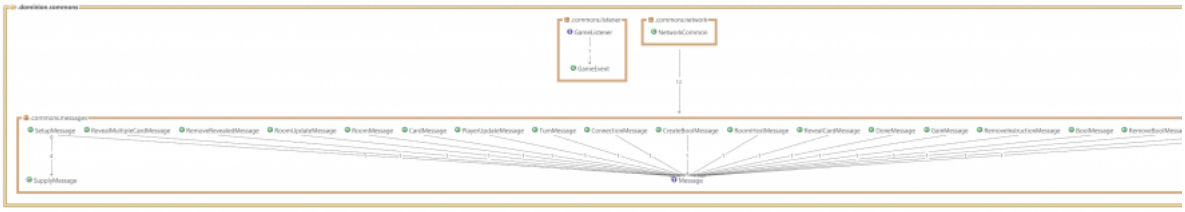
We do not plan on implementing any ability to use house rules anyone might wish to use.

Appendix

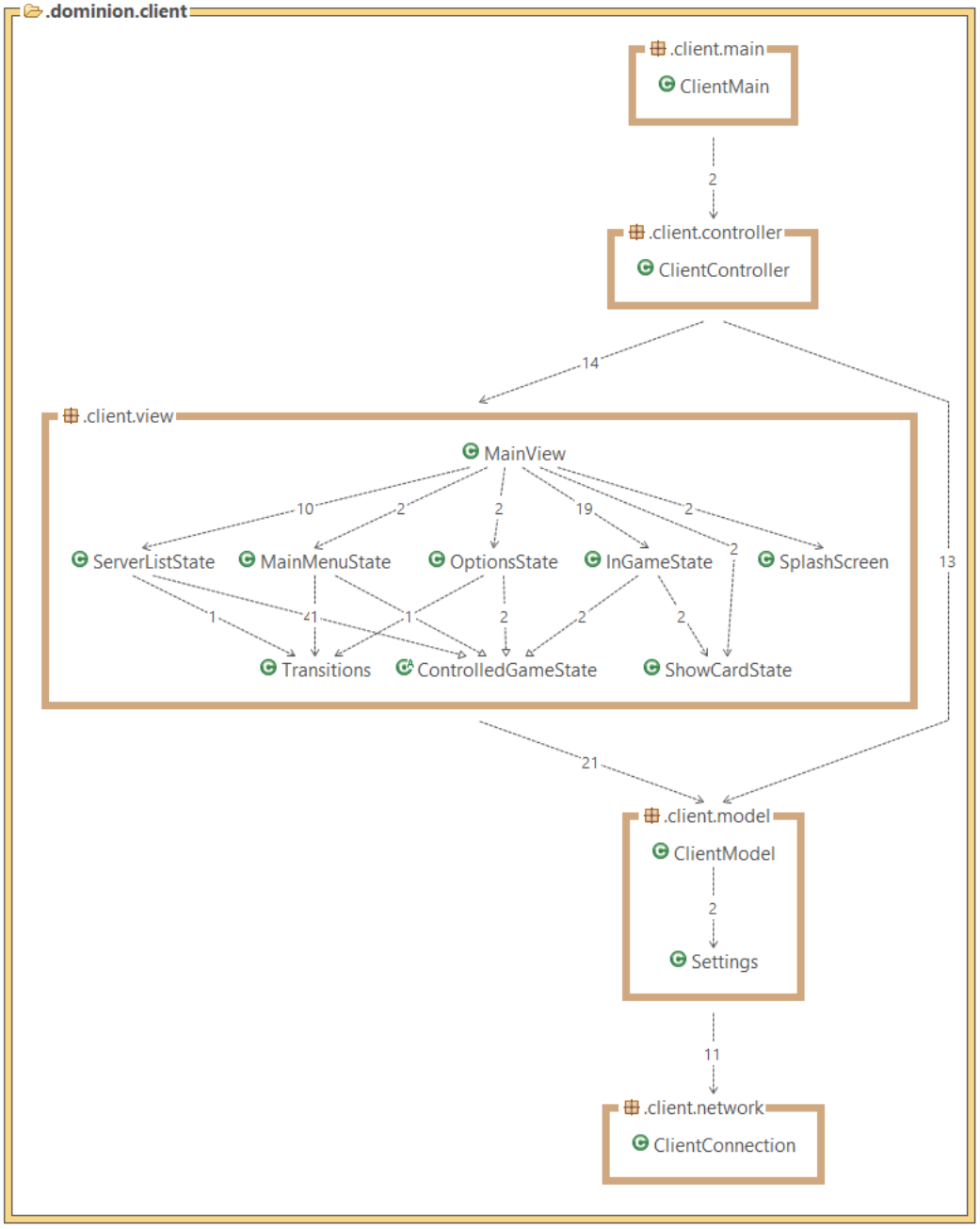
Dependency analysis of server



Dependency analysis of commons



Dependency analysis of client


.dominion.client
.client.main
ClientMain
2
.client.controller
ClientController
14
13
.client.view
MainView
10
2
2
19
2
2
ServerListState
MainMenuState
OptionsState
InGameState
SplashScreen
1
41
1
2
2
2
Transitions
ControlledGameState
ShowCardState
21
.client.model
ClientModel
2
Settings
11
.client.network
ClientConnection