System design document for 7 Wonders game.

1. Introduction

The 7 Wonders board game has been a staple board game for a while, and due to the current pandemic, restrictions are limiting our capabilities of meeting people, and of course you still want to be able to play your favorite game.

In this document we will describe how we have implemented 7 Wonders in Java and how it is structured.

1.1 Definitions, acronyms, and abbreviations

-- Definitions etc. probably same as in RAD --

- Client - The part of the app that the user interacts with. Can be seen as a view/controller for the underlying model.
- Server - The link between the model and the clients. The server tells the model about new input, and gets a copy of the current state.
- Client-server model - Sits between the view-controller and the model. Acts like an adapter pattern to convert java-representation to be network compatible.
- Model - An entity that holds all of the logic and information about the game.
- View - The graphical interface that represents the game.
- Controller - An entity that lets the player give input to the game.
- JSON - A hashmap-like data structure with key and values.

2. System architecture

The application is structured into two major components, the server and the client. The server is an abstract overview of the model. The server acts like an entity that handles connections while the server controls the models in the background to be able to communicate with the clients. The networking module is acting as a messenger system to be able to communicate between the client and the server.

The underlying model to control the logic is the model which uses the server to send data to the clients. The client acts like a renderer of data which keeps dependencies to a minimum. This allows us to make customizations during runtime since the view is only dependent on what data is sent or received.

This architecture was chosen due to reducing risk of cheating and giving too much control to the client. We can't let the user do any computation since the client can manipulate the data and gain an unfair advantage.

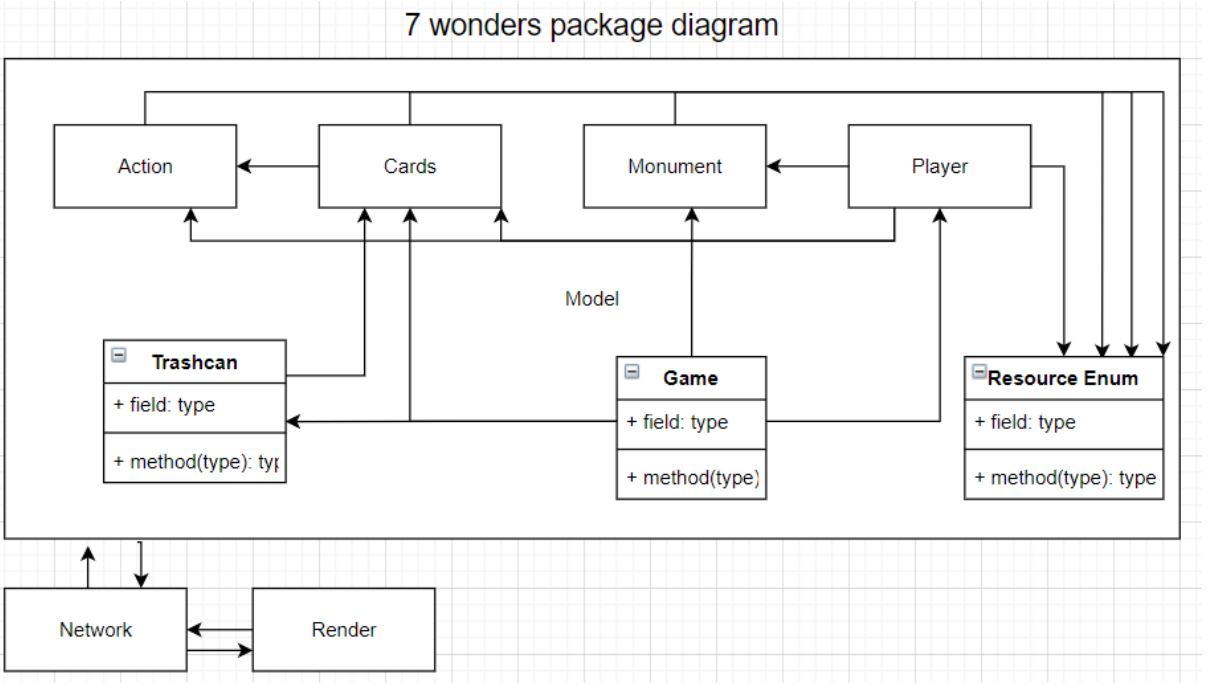
The entrypoint of the program is either starting the server as a single entity just running, waiting on clients, and the other option is to create the server in the client which then creates an instance of the model and connects the host client.

To be able for the client-server to be able to communicate over the internet we use JSON-objects to represent the data for the view to render, and this method was chosen for its simplicity. On the internet, all data is bytes. Using JSON is an easy solution rather than building a customized parser that may save some computation and bytes of data sent.

When the game ends, everything should shut down. To be able to play a new game, you have to repeat the initial start process again. This inevitably disconnects everyone and removes the last game entirely.

3. System design

Relationships between packages



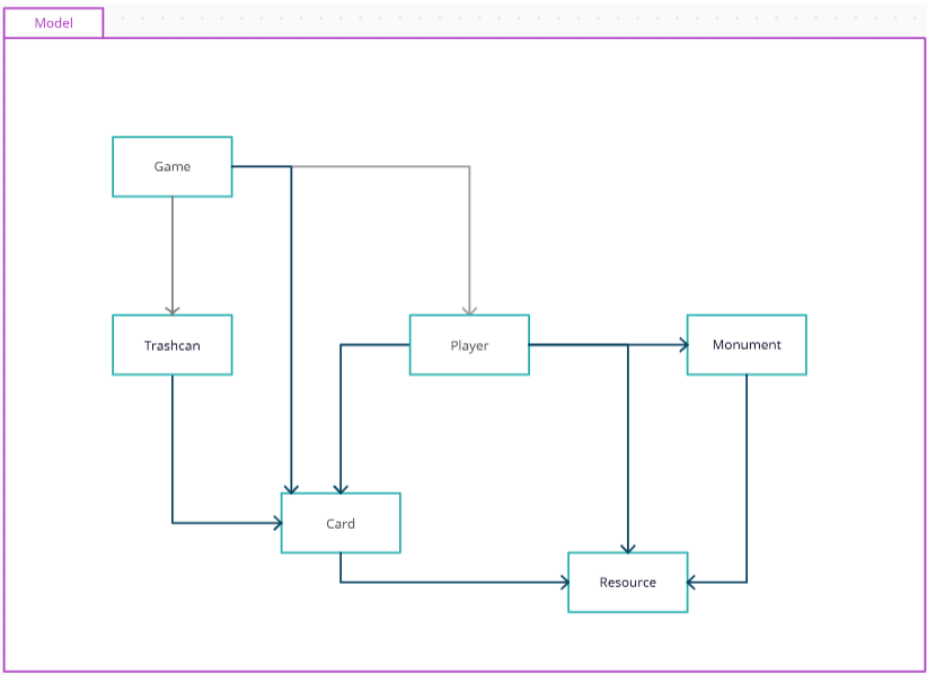
We have a package for our Model containing four packages. Each package represents a piece of the game.

The Player package represents a player in the game and contains data connected to each player: how many resources they have, what cards they have on their hand, each player's monument etc. Each player can do different kinds of actions each turn, which is represented by the Actions package.

The Monument package represents each monument in the game. When the game is initialized, a Monument factory creates a monument of each kind and assigns one monument to each player.

The Card package represents cards in the game. Cards can have different kinds of actions, represented by the Actions package. Each player has cards on their hands (represented as a list of Cards).

Domain model vs design model



Player, Monument and Card have been implemented as packages while we found the other entities (Game, Trashcan, Resource) to be best represented by a single class. There are more classes in the Model package in our design model but those we show here are the most important. The relationships in the design model are the same as in the domain model, except we have added a dependency from Game to Card and introduced a new package called Action. This is because Monument and Card have a lot of similarities in what kind of resources they can give the player (which we call an “Action”), so it’s an attempt to get more abstraction.

Usage of MVC

In our program we are using MVC. Our model is independent, but so is our view as well. The model will communicate with clients through a server-client protocol, which means it is a server that holds the model, and the clients hold the view.

Design patterns

Design patterns:

- Factory pattern
- Template pattern
- Observer pattern
- Server pattern
- Adapter pattern
- Decorator pattern
- Command pattern

4. Persistent data management

The client uses persistent data. We store it under an asset-folder which contains everything for the view to render. Settings are stored in a config file which the client generates.

5. Quality

The main method of testing is junit testing. This is the crucial part during development to be able to ensure that your refactoring or addition works.

PMD-report: See attached files
Dependencies: Not yet tested

6. References

Travis. Automatically testing after git push.
Maven. Be able to fetch packages/modules.
Libgdx. For the client to be able to render graphics.
Org.json. To package data into a json-object which the server and client uses for communication.