DIT212 - Final Report

Emmie Berger
Gabriel Hagström
Samuel Hammersberg
Björn Rosengren
Sebastian Selander

2021 October

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.

1.1 Definitions, acronyms, and abbreviations

- 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 model tells the server to serve the clients with the data.
- 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 keys and values.

2 Requirements

2.1 User story

Description

As a user I want to able to get a hand of cards from a deck so I can get resources and beat my competitors

Confirmation

Functional

- Each player will be handed seven cards at the start of each game
- The cards on each player's hand is only visible to them, not the other players
- When only one card remains in each player's hand, they will be discarded and the next age will begin
- All cards in the deck must have a type
- Cards can only be played during their specified age
- If a card has a cost, it must be shown to the player
- If a card gives resources, it must be shown to the player

2.2 User story

Description

As a user I want to be able to draw cards from my hand when it's my turn so I will be able to perform actions that I can get resources and perform actions

Confirmation

Functional

- A player can draw only one card each turn
- A player can't skip drawing a card
- After a player has chosen a card, they must choose exactly one of the three available actions
- If the card gives the player resources, they will be added to the player's pile of resources

2.3 User story

Description

As a user I want to have a monument so that I can play the game

Confirmation

Functional

- A player must be able to see a graphical representation of their own monument
- All players must have exactly one monument
- A player cannot change monument when the game has started
- All players must have a unique monument

2.4 User story

Description

As I user I want to be able to build stages on my Monument so I can get rewards and beat my competitors

Confirmation

Functional

- A player must be able to see what is required to build every stage of their monument
- All monuments have three possible stages to build
- When the game starts, the monument will not have any stages built yet
- When a player chooses to build the next stage of their monument, they must have sufficient resources to do so
- When a player builds the first stage of their monument, they will be given the first stage reward
- When a player builds the second stage of their monument, the player will be given the second stage reward
- When a player builds the third stage of their monument, they will be given the third stage reward
- When a player has built a stage of their monument, the card must be removed from the game
- A player must build the stages of their monument in order (first stage 1, then stage 2, then stage 3)
- A player doesn't have to build any stage of their monument to able to win the game
- A player can build multiple stages of their monument during an age

2.5 User story

Description

As a player I want to able to select a nickname

Confirmation

Functional

- A nickname must be set before the game starts
- A nickname must be at least three characters long
- A nickname must be unique
- A nickname cannot be changed during the game once it has started

2.6 User story

Description

As a user I want to be able to play in the three ages so I can go forward in the game

Confirmation

Functional

- The game starts in age 1
- The game changes to age 2 when all cards in age 1 has been drawn or thrown into the trash can
- The game changes to age 3 when all cards in age 2 has been drawn or thrown into the trash can
- At the end of age 3 the game should calculate all scores and decide a winner
- At the end of each age, war tokens will be handed out

2.7 User story

Description

As a user I want to be handed war tokens at the end of each age so I can win the game by beating my opponents by war

Confirmation

Functional

- If a player has higher war points than one of it's neighbors, they will be rewarded with a war token with a value depending on the current age:
  - At the end of age 1, the value will be 1
  - At the end of age 2, the value will be 3
  - At the end of age 3, the value will be 5
- If a player has lower war points than one of it's neighbors, they will be given a war token worth -1

2.8 User story

Description

As a user I want to be able to perform an action during each turn so that I can progress in the game

Confirmation

Functional

- After each player has chosen a card from their hand, they will have to choose exactly one of the following actions:
  - Build the structure on their chosen card
  - Build a stage of their monument
  - Discard the card to get three coins
- If the player chooses to build the structure of their chosen card, they must have sufficient resources
- If the player chooses to build a stage of their monument, they must have sufficient resources.
- If the player chooses to discard their card, they will be handed 3 coins. The card will then go into the trash can.
- When the player has performed one of the actions, they will give the remaining pile of cards to the player on their left

2.9 Definition of Done

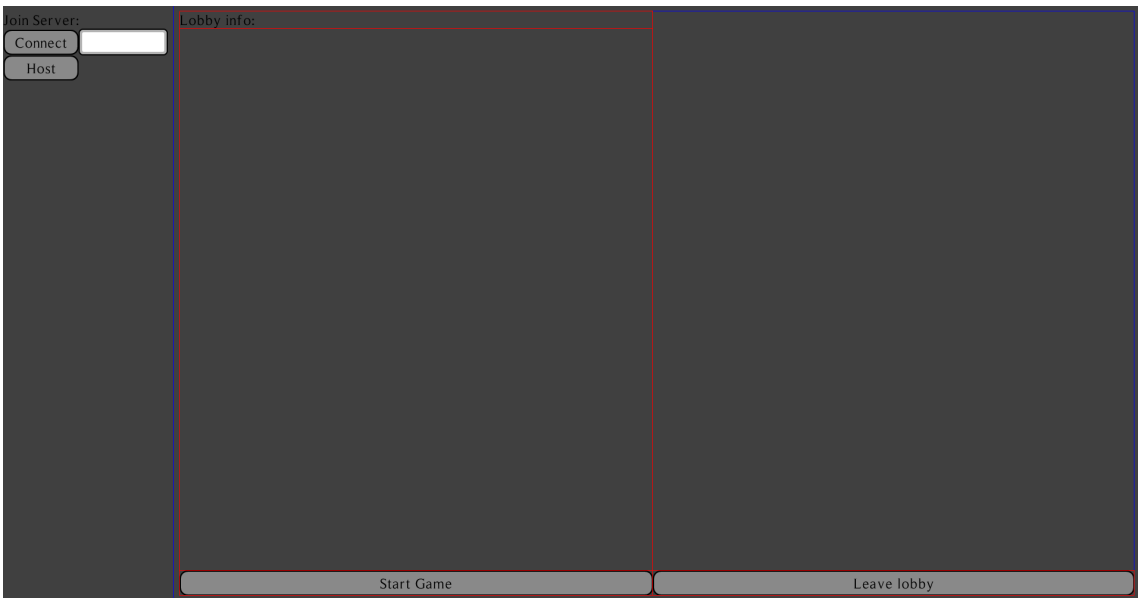
- Everything that can be tested must be tested ($\geq$ 90% line coverage)
- A task must be implemented both in back- and front-end to be considered done

2.10 User Interface

We have created a prototype for our game.

- Starting page: When the user starts the game, they will be presented with a starting screen with different options.

- If the user chooses to start the game, they will be able to choose from different servers to join, or choose to host their own.



- After the user has chosen a lobby and enough players have joined, the game will start. Each player will be given a monument and a set of cards at random. In this example, the user has been given *The great pyramid of Giza* and has 5 cards to choose from. Their left neighbor has the monument *The temple of Artemis at Ephesus*. The view will remain the same during the rest of the game, whereas the user continues to draw cards and upgrade their monument.

- Here is what it is supposed to look like after the player has drawn a card in the example of the previous picture.

- If the user chooses to go in to the settings menu, they will be able to change their username and some other options.

3 System Architecture

The application is structured into two major components, the server and the client. The server handles all the connections to all connected clients, while also acting as an interface to interact with the model, allowing the clients to send messages telling the server what they want to do with the model. For example; a client can send something like ”player A picks card 5”, and the server will then use a fitting method to interact with the model, sending back a message telling the client if the action succeeded or not.

The client acts like a renderer or the View/Controller in MVC terms, and all it does is simply send messages of what the player wants to do, and draws a fitting view of the data that is returned.

For the clients and server to be able to communicate over the internet we use JSON to represent the data sent from client to server and vice versa. This protocol was chosen for its simplicity, since everything sent over the internet is in the form of bytes, using JSON was an easy solution compared to creating our own parser, which would have been a major project on its own.

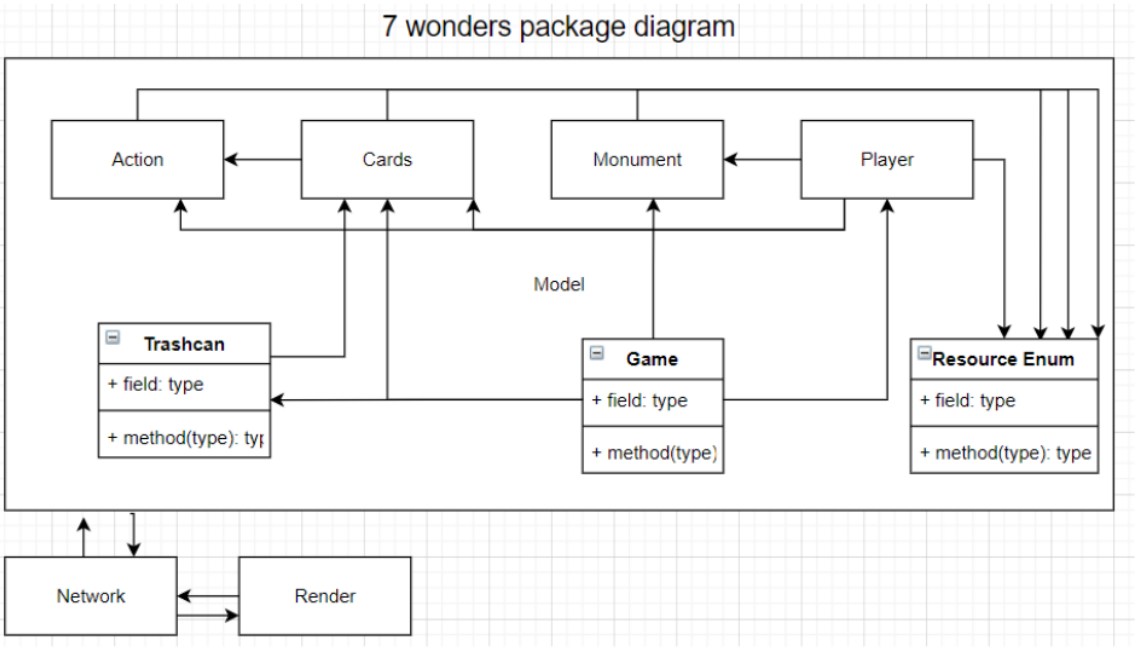
This server-client architecture was chosen due to reducing risk of cheating and giving too much control to the client. We can't let the user do any computation since the client could theoretically manipulate the data and gain an unfair advantage.

After startup the user is presented with the choice of either starting a server and joining it, or joining another users server. After that the user can start the game, sending each connected client to the game screen, allowing users to play the game.

When the game ends, everything should shut down, disconnecting all connected clients and removes the last played game. To be able to play a new game, you have to repeat the initial start process again.

4 System design

4.1 Relationships between packages



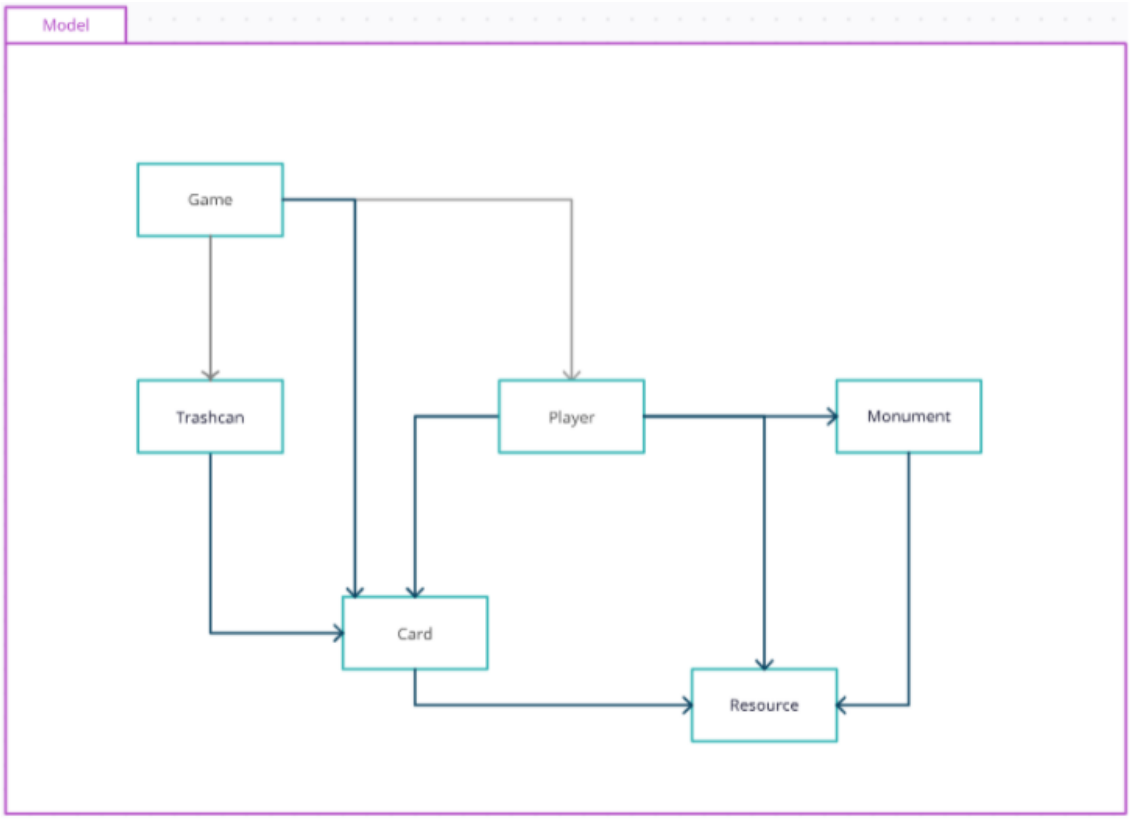
We have a package for our Model containing four packages. Each package represents a piece of the game.

The player package represents a player in the game and contains data that can not change during the game, except PlayerState. The PlayerState holds all data that can change during the game, such as: how many resources they have, what cards they have on their hand. Each player can via picking cards or upgrading their monument do different actions. This is represented via the Actions package.

The Monument package represents the monuments in the game. When the game is initialized, a Monument factory creates a monument of each kind. We then assign a monument to each player in the Player module.

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). When the game starts and/ or a new age begins, a card factory will generate a hand of cards for each player.

4.2 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.'

4.3 Class responsibilities

- Game: Is responsible for the overall logic during the game when it's played. Initializes all objects of Player, Monument, Trashcan and Card.
- Trashcan: Keeps cards that have been thrown away during the game and needs to be saved for later.
- Player: A module that represents a user and it's data. Player has a class PlayerState that keeps track of everything a Player has that can change during the game.
- Card: Represents a card in the game that a user can draw. A card has a color and a cost.
- Monument: A module that represents all monuments in the game. Stores data about what is required to upgrade each monument and how every upgrade will reward the player.
- Resource: An Enum that represents each possible resource in the game (wood, ore, stone, clay etc). Also includes coins, victory points and war tokens.
- Action: Represents an action on a card or a monument. It is an attempt at resembling an ability in most other games. E.g League of Legends

4.4 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.

4.5 Design patterns

These are the design patterns we are using in our project

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

5 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.

6 Quality

The main method of testing is JUnit testing. This is the crucial part during development, to be able to ensure that when adding features or refactoring our code still works.

7 Additional thoughts

We severely underestimated how large this project would end up being. The complexity of the project skyrocketed, after we started translating the game into code, worked on getting the server working, and all of this while following design principles and generally writing good code. In the beginning we were worried this project would end up being too small, but around 7000 lines of code later and we have a lot left to implement if we want to have all functionality the game has to offer.

The method pickCard in Game is very smelly. We're basically using instanceof and casting the Card type to the correct subtype. The reason for this is because depending on which type of card it is, different calculations should be done. As mentioned in the previous paragraph, we had to make the decision to stay with this and keep working to make deadline. If we would have had more time, this would have been a high priority to fix.

8 References

- Travis. Automatic testing after git commits. (In the beginning of the project)
- Github Actions. Automatic testing after git commits. (At the end of the project)
- 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.

Peer review of ESSBG by Grapefruit

The project uses a consistent coding style. Apart from some unnecessary blank lines, we have no other remarks on the code style.

The code uses several design patterns which makes the project object oriented. For example the project uses Factory pattern, for creating *Monuments* and different amounts of *Cards*. The Observer pattern is used when *Render* listens to the *Network*. Also, Template pattern, Decorator pattern and Adapter pattern is used.

The code is pretty well documented. JavaDocs need some improvements, describe the parameters and return parts. Some files are well documented and some are not, so it's a bit uneven. You could add @author at several classes.

We would say that the project has proper names. Good and describing names.

After an overview of the code we couldn't find any unnecessary or circle dependencies.

We could see that you tested a lot of the model package, but you could test a bit more to cover more lines of code.

Since we don't have much experience of working with servers, we struggle a bit with understanding how it all works. But looking at individual classes and files we understand their functionality.

The code in regards to MVC is separated into individual directories which gives a clear overview of MVC. Also, the Model is independent which means that we can easily add more functionality to the Model without messing up the view. The view can also be updated without messing up the Model.

After looking at the View, we found that the view seems to have the responsibilities that a view should have (only graphical methods). Though we noticed that the "show" methods in *GameScene* and *LobbyScene* contain a lot of code which could be separated into smaller ones. On the other hand, we think you did a good job splitting up methods in the class *DrawableBoard*.

The code for *Card* and *Monument* is reusable and extendable due to their abstract classes and use of enums and factories.

There are a lot of classes which depend on the org.json library. This could be improved by implementing some kind of jsonHandler that contains the json functionality throughout the program.

Regarding performance issues, we noticed that Mac users are not able to run the project via intellij. Moreover, we couldn't figure out how to actually start the game.

There are some classes with a lot of instance variables and sometimes there is a purpose for that.

Overall, it is a great start to your project. Both SDD and RAD are adequate and include good descriptions of the project. More JavaDocs and tests are some things to consider. Keep up the good work!

Peer review of cOOla-klubben B)

Introduction

At a first glance the project and documents look good, however there are some typos and incorrect grammar in the SDD and RAD. The back end of the project is pretty small, so there shouldn't exist too many problems there. The front end is written in React so we won't comment on it.

Code

- The project uses an inconsistent coding style. Lines are indented inconsistently. If you are using IntelliJ there is an auto formatting tool that you should probably use.
- The Calculator class and test class look an awful lot like the assignment from the first imperative programming course. If it is copied from a previous course we recommend checking with your supervisor if you're allowed to use the code.
- Most classes consist mainly of getters and setters for data. This gives consistency but also makes the classes thin. We however think it's reasonable to have classes mainly containing data, for example Product, and others containing logic. The classes are easy to digest and follow single responsibility principle mostly.
- We have no experience in Java spring or React so we can't really comment whether the project is following MVC. We also question why React is used to implement the GUI as that adds unnecessary dependencies on non java-native libraries.
- We see no usage of design patterns, but we're not sure if they are needed either.
- The Calculator class is lacking a bit in modularity. We understand why it is written the way it is, but if extracting the tokenize part of it, it would be easier for multiple people to work on. A side note on this class aswell is it seems like containsOperator is implemented incorrectly. The name suggets it checks if a string contain an operator, but this is not what it does.
- Saving the password as an unhashed string in LoginHandler could definitely be a security issue.
- Some classes are missing tests for some methods but that's understandable. The tests that are implemented are easy to understand and test what they're supposed to.

- The way you're checking if a string is an email is questionable. Currently "@@@" would be considered a viable email address.
- A lot of methods are package private (no private/public/protected), and while this works due to the project not being dividied into modules, it should be fixed as it is a codesmell.
- MockObjects should not be in the backend module since it seems to be a test file.

Documentation

The documentation in the code is lacking. The already existing documentation is commenting trivial lines and doesn't help much. We recommend briefly commenting on what each method does using javadoc instead of commenting in-line in methods. The names of existing methods and attributes are good however, they're easy to understand and the name mostly tell us what they are representing.

UML

The UML specifies a usage between User and Member when there should be an association between them. User holds an arraylist of members. The UML seems to be outdated slightly as the arraylists have been replaced by hashmaps.

Summarization

We suggest you divide the different parts of the model into packages, so it's easier to understand at a glance what each part does. It would also make it more obvious how you have implemented MVC, which now requires us to go through every class. Comment the code using javadoc, and make sure to not comment lines that are obvious. Consider extracting the tokenize part of calculator. Fix the security issue with saving password as an unhashed string. Rewrite the way you check if a string is an email.