Bilkent University

Department of Computer Engineering

CS319 Term Project

Project short-name: Scrolls of Estatia

Group 3F

Design Report

Atakan Sağlam, Sarp Ulaş Kaya, Berk Kerem Berçin, Oğulcan Çetinkaya, Furkan Başkaya

Instructor: Eray Tüzün
Teaching Assistant(s): Barış Ardıç, Emre Sülün and Elgun Jabrayilzade

Iteration 1/Project Design Report
Dec 13, 2020

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS319.

Contents

Design Report

Project short-name: Scrolls of Estatia

1. Introduction

1.1 Purpose of the system

Scrolls of Estatia is a Monopoly-style game with various additions to make the gameplay more interesting, such as a class system with each class having unique abilities, and a new deck of cards called Scrolls, which the player can hold in their hand and use on one of their turns to unleash different effects. These Scrolls provide the player with more options to change the course of the game. The game will be an online multiplayer game with 2-8 players.

1.2 Design goals

End User Criteria

a. Usability

The game will be easy to play with well-defined user interfaces, and easy to understand gameplay rules. The user interfaces will have a focus on simplicity, and it will be easy for the player to be aware of the game's current state the whole time, i.e. which player owns which square etc… The game will feature a tutorial which explains how to play the game to the user, and a settings menu which will allow the user to configure various game settings.

b. Performance

The game will be lightweight, with simple graphics and minimal hardware load to be able to run on low-end

machines. The response time of the system and the communication with the database will be quick.

Developer Criteria

a. Modifiability

   The source code will be very readable and easily understandable, thus anyone looking to make modifications to the code can easily do so. Each subsystem will be independently modifiable without breaking the other subsystems.

b. Maintainability

   The source code will be well documented with comments that will allow anyone looking to maintain the program in the future to understand the system easily.

c. Reliability

   The program will keep track of the progress of disconnected players so that they can reconnect and continue playing. We will pay attention to the interaction between the subsystems to make sure the system doesn't crash on edge cases.

2. High-level software architecture

2.1 Subsystem decomposition

Our system will be a combination of Client-Server and Model-View-Controller architectural style, with the view being in the client side, and the model and controller being on the server side. In the client side's view, there will be a UIManager which is responsible for displaying the data from the model, as well as presenting options for the user to make inputs, the requests of which will be sent to the server side's controller with the socket of the Player class.

On the server side, there will be a controller with a Server class which handles the relationship between the inputs coming from the client side and the data on the server side's model. The model has all the data of the current state of the game, with the board and all its current specifications, and all the tokens on the board. The ServerSocket will be connected to the Socket of the client side's view, in order to display the current state of the game on the client side, to the user.

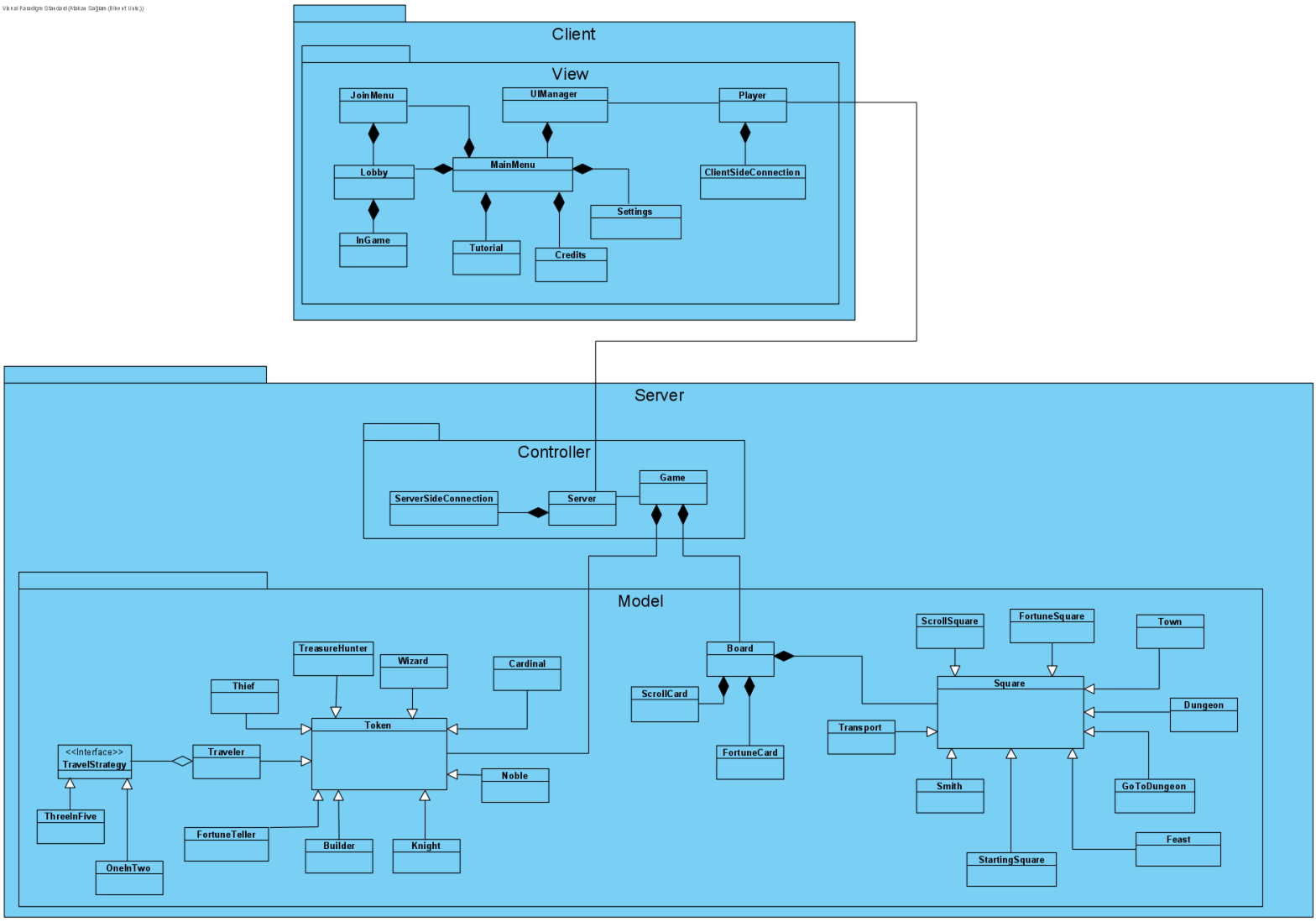

Figure 1 Subsystem Decomposition

2.2 Hardware/software mapping

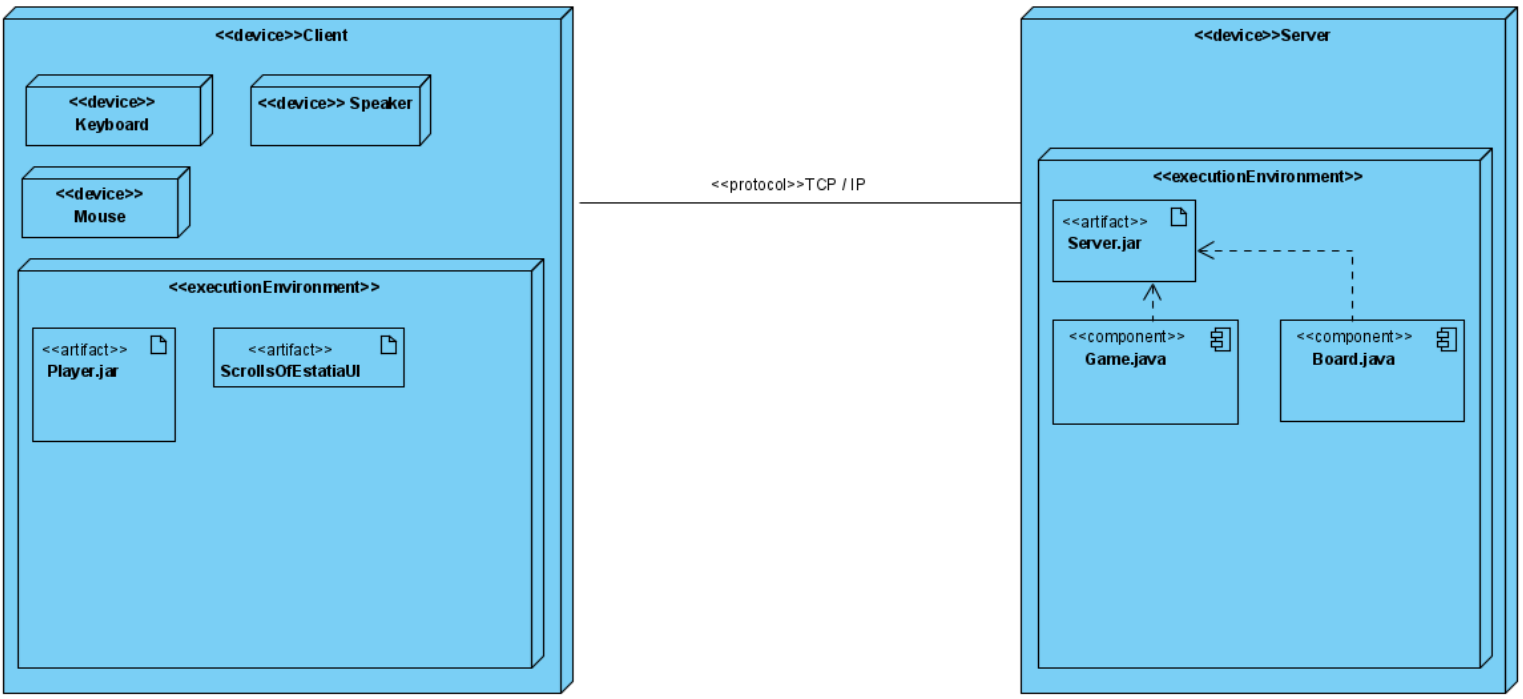

Figure 2 Deployment Diagram

The game will be implemented in Java 8, so it will require Java Runtime Environment 8 or higher to run. Since this will be an online multiplayer game, it will require an internet connection. Local files and user setting preferences will be stored locally on the user's computer, and the game's instance and its current state will be stored on the server. These will be accessed by the clients with requests and received with responses from the server.

2.3 Persistent data management

Since this will be a multiplayer game running on a server, the user's setting preferences will be stored on a JSON file on the user's computer. The content of the game such as the board image, the token images, the music, the sounds, the UI elements will be stored on the user's machine as files. During gameplay, the current state of the game will be sent from the server, which the client's view will display to the player using the locally stored files.

2.4 Access control and security

Our game will not have a log-in system, and anyone who has the room ID can join a game. However, if a game has already started, no new players will be able to join that particular game even if they have the room ID. The one exception to this is if they were already present in the lobby when the game had started, and for some reason they got disconnected, then they will be able to rejoin the game. During gameplay, the players will only be able to do actions that affect the game only during their own turns, such as rolling dice, buying property etc.

2.5 Boundary conditions

Initialization

During initialization, the game reads the user's setting preferences from the saved file and applies the settings. The user will interface will show the main menu to the player when the game is launched. When the user attempts to host or join a game, the system checks the internet connection.

Termination

The player can exit the game via the "Exit" button on the main menu, or use an operating system's methods for stopping a process such as clicking the "X" button, Alt+F4, or forcing it to stop through the task manager. A client can terminate independently from the server. When a player chooses to exit the game, the client notifies the server, and that player is removed from the game.

Failure

The server will check the validity of the requests coming from the clients such as whether the action is performed on the player's turn, whether the player has sufficient resources or whether the player's input is valid etc. If a player loses connection, the server will

keep their player ID and not remove them from the game, so that they can rejoin the same game using the room ID.

3. Low-level design

3.1 Object design trade-offs

Efficiency v. Portability

We are going to develop our project on Java 8 and use sockets to handle the server-client communications, so any device capable of running Java 8 will be able to run our program. We are going to use Serialization to send data between the server and client. This makes our game portable, and we believe this doesn't make a trade-off in efficiency.

Security v. Usability

In our game, the only way to join a game is to enter the unique room ID. There will not be a server browser, or matchmaking, and the players will have to get the room ID from the friends on some other platform, such as Discord [1]. This decreases usability, however, since this is the only way to join a room, only the people who possess the room ID can join a game, and this increases security.

3.2 Final object design

Figure 3 Final Object Design

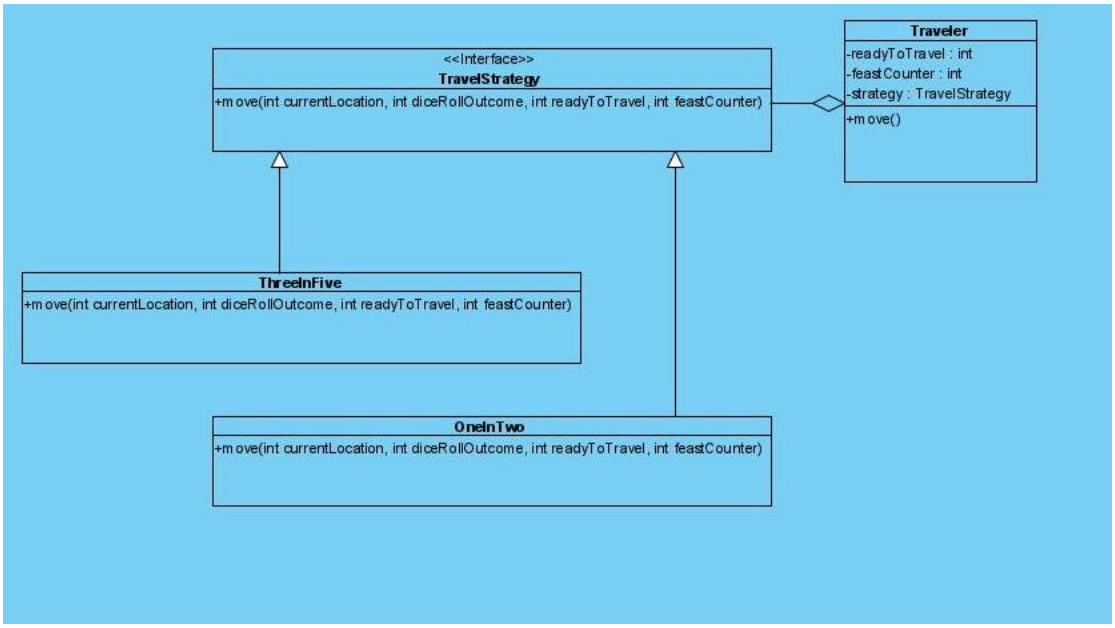

Figure 4 Strategy Pattern

For the move strategy of the Traveler Class, we decided to use a Strategy Pattern to change how the Traveler Class uses its special ability. This allows the player to select between two strategies: One that is more suitable for quick, short-term rewards, and another that rewards patience and is more suitable for long-term strategies.

Figure 5 Singleton Pattern

We used a Singleton Pattern for the Game class, and instantiated a Game inside it, to be certain that there is one game instance, and that every class accesses the very same instance of Game. For example, multiple Token's calling a method that make changes on the Board can access the same Board by `Game.instance.board`.

3.3 Packages

javafx.event.ActionEvent : This package will be used for handling actions. Such as button pressing and mouse tracking.

javafx.fxml.FXML, javafx.fxml.FXMLLoader : These packages serve to a similar purpose as HTML. We will use them to set up UI and organize layering of the UI components.

javafx.scene.layout : We will use them for making layout configurations.

javafx.scene.control : We will use this for declaring the controllers for our FXML files.

javafx.scene.Scene : We will use this for changing or updating our GUI according to user actions.

javafx.scene.control.Button : Buttons will be the main operating components of our games and we will use this class to assign functionalities to these buttons.

javafx.application.Application : Will be used for component lifecycle management[4].

java.io.Reader; java.io.BufferedReader, java.io.InputStreamReader, java.io.OutputStreamReader : These will be used for I/O operations that will occur between server and client

java.net.Socket, java.net.ServerSocket : These will be used handling data transformations between server and client

3.4 Class Interfaces

Runnable: This interface will be implemented by any class whose instances are intended to be executed by a thread [2].

Serializable: Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object [3]. We are going to use serialization when sending and receiving game state data between the server and clients.

4. Improvement Summary

We added a Deployment Diagram. We added two new design patterns: Strategy Pattern and Singleton Pattern. Strategy pattern will be used for making Traveler class’s move algorithm interchangeable for better user experience. Also, this approach encapsulates them. Singleton design pattern is used for coordinating actions across the system. We added an instance

object to our Game class that can be shared and used in different classes.

5. Glossary & references

[1] Discord. https://discord.com/. [Accessed 29 Nov. 2020].

[2] Runnable. https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html . [Accessed 29 Nov. 2020].

[3] Serialization. https://www.tutorialspoint.com/java/java_serialization.htm. [Accessed 29 Nov. 2020].

[4] JavaFX. http://tutorials.jenkov.com/javafx/index.html. [Accessed 29 Nov 2020].