Developer Manual

Risk 6
Written by: *Weng Phung*
Reviewed by: *Eliya Ameri*

Introduction

This manual provides an overview of the codebase. Understanding these details should assist in navigating the code and in implementing new features while preserving the existing structure and design.

Vital packages and classes

The software project comprises several key packages and classes:

- `logic`: This package contains the classes *MoveProcessor* and *GameController*, which handle game logic and flow.
  - `models`: Classes from the domain model can be found here.
  - `ai`: Contains four bots (*Easy*, *Medium*, *Hard*, *Tutorial*) which play the risk game to their specific abilities according to the structure given by the implementation of the game.
    - Package `montecarlo`: Used for the implementation of the HardBot. The HardBot calculates different branches of gameflow and chooses the best first move which had lead it to victory in previous simulations.
    - The win probabilities used in the *Probabilities* class have been sourced from `https://www.reddit.com/r/dataisbeautiful/comments/vknu9r/oc_the_probability_of_winning_a_battle_as_an/`
- `gui`: This package contains elements of the Graphical User Interface.
  - `controllers`: Includes controller classes, such as *LogInSceneController*, *GameSceneController*, *CreateAccountController*, for a variety of scenes.
  - `scene`: Includes various scenes, such as *CreateLobbyScene*, *GameScene*, *LogInScene*, that implement the visualization of many session states such as:
- `network`: Provides the *GameServerFrameHandler*, *GameClientHandler*, *Serializer*, and *Deserializer* classes which are responsible for server-side operations, data serialization, and deserialization.
- `database`: Classes such as *User*, *UserRepository*, *GameStatistic* and *GameStatisticRepository* play a role in the implementation of long term data storage of player profiles.
- `json`: Used for handling JSON documents related to e.g. country and continent configurations.

Code Structure

The team followed the java google style guide to increase the readability of our code and optimize a possible transfer of our project to another team in the future. The code is divided into major packages as has already been described in order to separate the different branches of development and simplify making changes. Furthermore, Getters and Setters are positioned at the end of each class for uniformity and ease of navigation.

Development of new features

If new features are to be implemented, such as for example an alternative custom board with custom rules, this is where most of the implementation would be involved in.

- **JSON Documents:** You can redefine/ add new countries in the JSON Documents and (re)define their adjacent countries. If a new additional map should be created a new Json document can be created and parsed.
- **Cards:** To add new special cards with different influences on the game state, changes in the Card class in the `logic` package are recommended. One should note that the implementation of new cards is accompanied by related changes in MoveProcessor and the logic.

- **New Move Type:** To add a new kind of move, it has to be defined as an extension of the abstract class *Move* in the `logic` package and be processed as a new case in the *GameServerFrameHandler*
- **GUI:** For new visual elements or interaction features, changes such as new classes would likely be required in the `gui` package.

Patterns and Principles

Several design patterns and architectural principles are used throughout the project:

- Observer Pattern: Used extensively for updating various components based on changes in the game state. The observers include *GameLobbyObserver*, *GameStateObserver*, *ChatObserver*, and *ServiceLobbyObserver*.
- Model-View-Controller (MVC) Pattern: Applied in various degrees across different components. A full MVC pattern is implemented for the Player (with *PlayerController* and *PlayerUI*), and a partial MVC pattern for the Deck (with *DeckController*) together with the Cards (with *CardUI*).
- Strategy Pattern: Utilized by AI. The *AiBot* interface is implemented by *EasyBot*, *MediumBot*, *HardBot*, *MonteCarloBot* and *TutorialBot*, meaning that the rest of the software can simply focus on calling the relevant methods required by the AiBot interface while being sure that each true bot instance performs their actions correctly.
- Builder Pattern: Used for creating Lobby, GameLobby, Cards, Players, Countries, and Continents. An example for a factory class is the *GameConfiguration* Class that initializes a GameState along with all of its attributes such as the different countries.
- Domain Model (Architectural Pattern): We've incorporated a layer with objects from the risk game domain to make interaction with the application as similar and naturaly to the real-world board game. Such classes are for example *Card*, *Player*, *Dice*, *Deck*, *Country* and *Continent*.

Java Version

For this project, we chose Java 17 as our primary programming language and version. The decision was mainly driven by the fact that Java 17 is a Long-Term Support (LTS) release, which means it will receive updates and support for an extended period of time. This ensures that our program remains secure and up-to-date, without requiring frequent rewrites.