Junior Honours Project

Risk - World Domination Game

Final Team Report - Group I

Team Members:

Bence Szabó

Patrick Opgenoorth

Alex Wilton

Ryo Yanagida

Supervisor:

Professor Stephen Linton

University of
St Andrews

2015-04-21

Abstract

This report describes the implementation details of our Risk World Domination Game. It describes the four core parts of the project: the game engine, the client-host communication, the web client and the artificial intelligence. It shows how we tested our implementation and how successful we have been bringing our program together. It gives an overview of the changes in implementation since the day we started. Lastly it assesses how successful we were and gives some ideas about possible future improvements.

Declaration

We declare that the material submitted for assessment is our own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 5,600 words long, including project specification and plan.
In submitting this project report to the University of St Andrews, we give permission for it to be made available for use in accordance with the regulations of the University Library. We also give permission for the report to be made available on the World Wide Web.
We give permission for this work to be used in research within the University of St Andrews, and for any software to be released on an open source basis.
We retain the copyright in this work, and ownership of any resulting intellectual property.

Contents

1 Introduction

Our task this year was to implement a "peer-to-peer world domination game". The class has decided very quickly in favour of the famous board game Risk. We (the class) have chosen a version of Risk that is focused on World Domination, therefore the objectives were completely removed from the game except for one: the player who manages to beat all other players is going to be the winner. We also decided that the P2P structure will rely on a known, trusted host implementation and that there will be weekly meetings of groups, where an elected "group representative" will always be present. At these meetings, the groups have defined a protocol for message passing over the network. See issues about the protocol below.

Our implementation aims to provide all the functionality specified in the protocol in Java and JavaScript, using an MVC (Model/View/Controller) pattern. The game engine is built in Java, that allowed for easy unit testing and object orientation, The game engine implements and checks the logic of the game while keeping everything in the network communication separate from here.

The back-end of the game relies on Java Sockets for network communication. Both the server and the client are multi-threaded and update the game engine in the background while still keeping a constant flow of network protocol messages. These messages are implemented via separate classes as well to increase decoupling of components in the system.

The front-end however appears and runs inside a web browser, in a scalable window, heavily relying on JavaScript and bootstrap. It queries the game state from time to time and sends messages to the game engine. This completes the circle of MVC architecture.

2 Run Instructions

Three jars are submitted. They are found in the *code* folder, where they are expected to be run from.

Risk.jar

This jar contains the main game. It will launch a browser window in the default browser and guide the user through hosting and/or playing a game.

```
$ java -jar Risk.jar
```

Evolver.jar

This jar contains the code to advance the evolution of the EvolvedCommandRaterAI (see 3.4).

```
$ java -jar Evolver.jar CommandRater.evo
```

AIAnalyser.jar

This jar analyses the AIs and will output a csv with the results of the analysis.

```
$ java -jar AIAnalyser.jar
```

3 Project Details

As a team we have managed to get to a stage where the game can be played against each other and some AIs can be chosen to be played as well. All of this can be done from the web interface after running Risk.jar. The following sections will discuss the achievements of the project as a whole with some highlights.

3.1 Project Overview

In this project we have achieved a number of milestones in the implementation of Risk - World Domination Game. Here we discuss some of the key achievements in terms of implementation details/decisions.

3.1.1 Planning

We have started by defining languages to be used, ways to build projects, and ways to share projects between each other. Java was selected as the primary language of the project and thus the game engine. We used Java to write the protocol handlers, i.e. the host and client implementations are written in this language as well. We decided to create a web interface for the game that will conduct creating a host or connecting to a remote host with the client.
The above means that we basically decided in favour of using MVC architecture (Thanks to our supervisor) and started implementing the program bearing this in mind.
Maven was used to build projects, which allowed us to use external tools to develop the software and run it more easily.

We used version control heavily. First we started up with a mercurial repository on one of the group members' storage, but then we decided to move to github into a private repository and work there. This was mainly because of the better support of git over the IDEs we used and the easier merging that this carried. At the end, everyone ended up using IntelliJ exactly because of this. We used branches where large changes were made to the project and attempted to keep the master branch clean.
To achieve consistency we used Travis for checking the code and to run tests every time a push was made on github. This integration allowed us to see that some code has broken some tests. Obviously, there were points when this was expected and ignored for a (short) time period.

3.1.2 Project structure

The following list gives a structural overview of the packages we are using in our project.

uk.ac.standrews.cs.cs3099.useri.risk. Root Package

- **clients** Package to represent different kinds of clients in the game and hold classes to support them.
  - **AI** This sub-package holds the different kinds of Artificial players we have created. There are sub-packages inside this package as well which we are not going to list here. Refer to section 3.4.

webClient The web-based client is stored here, consisting of JettyServer class and ParamHandler class and the webClient itself, that runs the front-end of our game.

game Package to represent the model of the game model itself.

action Package to represent actions that a player can take. These have been developed to mirror real, human actions.

gameModel Game model representation.

helpers Package to hold miscellaneous helper classes.

randomnumbers This is the package that holds the unified Random number Generator given by one of the class members, namely Carson Leonard. Any implementation specifics must be asked from him.

main Package to hold main executables used to run the different parts of the program.

protocol Package to represent and handle the network protocol.

command Command objects to represent all the actions conforming to the protocol.

exception Custom exceptions that the protocol handlers may choose to throw.

In the following section, the structure of the project will be discussed in further depth.

3.2 Architecture

In earlier stages of development, as mentioned previously, MVC model was decided to be used. Since Java was chosen as the language, the use of O-O Programming (Object-Oriented Programming) was natural and thus became significant in the designing phase. In order to organise the project with many classes representing the game, protocol, and all the other functionality needed, the package structure explained previously was used. A more detailed list of the relevant classes is to follow below.

3.2.1 Package:game

The game package provides the actual representations of the model along with mechanism to drive them. Within the game package, there are couple packages underneath, including action, gameModel packages.

Package:action The "action" package handles validation and execution of Actions required in the game. These classes provide mechanisms to control the model as part of controller. All of the subclasses provide methods for validating themselves against the state and for applying the state changes. Package action contains following classes:

package:action package that holds actions

- **Action** Abstract Class defining the Action objects, super-class of the Action objects
- **AttackAction** Class representing attack action (only after all the commands related with an attack are processed, is this object created). Validates the generated Attack action object with given parameters
- **AttackCaptureAction** This action represents directly the Attack Capture Command defined below.
- **DeployArmyAction** Action object representing army deployments.
- **FortifyAction** Class representing Fortify Action, where armies are relocated by moving across countries.
- **ObtainRiskCardAction** Class representing action where risk card is obtained by the player after winning a battle.
- **SetupAction** Action made when setting up the game. Updates the owner of specified country.
- **TradeAction** Action to trade risk cards, it checks the occupied countries and cards to decide whether this is valid or not and how many armies to be traded back to the player.

package:gameModel This package holds the game model-related classes

Continent Class representing continent. Each continent has a set of countries associated with it and a "value": The number of armies a player gets for owning all the countries in the continent.

ContinentSet Data structure defined to hold continents.

Country Object representing country. Name, ID, countries that are linked, number of troops and player object representing the owner of this country are stored.

CountrySet Data structure to hold multiple countries.

Map Object representing map. It contains continentSet, countrySet, and JSON object representing map data passed across the network.

Player Object representing the player, holds Client object (the client used over the network or an AI), name, id, set of countries occupied, cards they have.

RiskCard Object representing a risk card. Types of cards defined by RiskCardType Enum.

RiskCardType Enum representing the type of risk cards available (artillery, cavalry, infantry, wildcard).

GameEngine Object running the loop, the core of the model on the client side.

State Object representing the current state of the game. It ties up all of the information needed to model the game. Map, players, card deck, current player to make a move etc.

TurnStage Enum representing stages within each game turn (trading, deploying, battles, drawing a card, fortifying and end of turn).

3.2.2 Package:protocol/ and protocol/commands

`protocol/commands/` contains objects representing the messages passed across the network. There are multiple commands that represent an attack that have to be collected together by the protocol handlers to form an attack action, therefore some of these commands are not directly connected to the Action classes above.

Command object Command Object extends JSONObject as all of the messages defined in the protocol are written in JSON. This is the template class extended by specific command objects like AttackCommand object. Command object contains following methods to allow marshalling/unmarshalling:

parseCommand(String envelopeString) Returns appropriate type of Command object or null for an unparseable String.

toJSONString() calls toJSONString to inner contents and generates String representation of message in JSON format that can be sent over the network directly.

Directly underneath the protocol package, the following classes are responsible for handling the network transactions in the host and to implement protocol.

ServerSocketHandler This class handles the game initialisation until the card deck is shuffled. It makes sure that the clients are initialised and all have the same initial state.

RejectingThread Thread to monitor incoming connections after game starts. This thread is always rejecting. Whenever a new client attempts to connect, this thread will immediately send a `reject_join_game` command and close the connection without waiting for a reply.

MessageQueue Essentially self-explanatory: it handles sending incoming messages (and messages originating from the server) to all clients. For this it maintains a list of currently open sockets.

ListenerThread Class dedicated to initialisation of each individual client that is connected to the host. This is not doing any useful work after the cards are shuffled.

initState Enum that represents the game initialisation state in the host. It has a customised next method as well.

HostForwarder This is the class that does the job of processing each command that arrives from a given client (each client has an associated HostForwarder). There are duplicates between this class and the GameEngine class.

ClientSocketHandler This is the client-side handler for the protocol messages. It runs essentially the same algorithms as the server-side for processing data sent on the network.

Essentially, `protocol` package handles the core of the network stack of this program. It introduces lots of concurrent programming through multiple threads and shared variables.

3.2.3 Package:clients

`client` package provides the representation of the controller of the game. There are mainly three types of clients, local human client, local AI client, and network client. Local human client requires UI to control the game, but with AI, UI only displays the state. This is similar to the network client where UI can't control the game as 'network' client but will display the state of the game.

Client Abstract class defining base class and base constructor

NetworkClient Network Client object defining players connected through the network on different instances

CLIClient CLI client which runs as part of the same instance as local client.

AI/ package containing AI related classes. Refer to the section 3.4

webClient/ Package Containing front-end-related classes

JettyServer This is the standard Jetty server class.

ParamHandler This is the class that handles incoming requests from the web UI. It then calls the appropriate functions in the model.

WebClient Local client which uses web browser as a way to view and control the game

3.2.4 Package:helper

As the name suggests, helper package contains classes that assist the program.

randomnumbers/ package containing random number generator implementation from standards committee

HashMismatchException

OutOfEntropyException

RandomNumberGenerator Class that holds the pseudo random number generation implementation [1]

3.3 Modelling

One of the earliest packages of the program that we have completed and was reasonably untouched for the latter months is the model. Modelling of the program began with the class diagram which grew and expanded as we discussed further by looking into the rather complex rulebook of risk. During the process of filling in the class diagram, areas of responsibility have slowly developed. This also affected how we split the work as well to a certain extent. Since the idea of using Java has already been established by this stage, rigorous O-O (ObjectOriented) design was given.

The game model, as we call it, is responsible for keeping the game state up-to-date at all points in the game. It holds a list of players and the game map. All moves (as we call them, Actions) have to be validated before they are applied to the game state. At any point during the game, if the host/client side detects that an invalid move has been taken, they will print out an error message to the log file. Both the host and the clients hold a game state and modify it according to all the commands received, let them come from remote or local clients. This is usually done on the same thread as the processing of commands so that each command gets interpreted before the next command can be taken, as described below. This inherently leads to some slow reactions from both the client and the server.

3.4 AI

Part of the project was to design and implement an Artificial Intelligence to play a Risk game against other AIs or human players. Due to the complex nature of Risk, and the

[1] implemented by Carson Leonard

various choices the players have to make during the game, this is not an easy task. [1]

We have chosen to approach this problem from a few different angles, and had the resulting AI Players compete to see which is the most successful strategy.

3.4.1 AI Clients concepts

ChihuahuaAIClient - Random choices The first approach chosen was to generate a list of almost all possible commands a player can possibly (and legally) take at the time he is asked for a move, and choose one randomly.

BulldogAIClient - (Almost) boundless aggression Bulldog client is a modification of RandomAIClient. It will still take random moves, but it will not end its turn before it has attacked all weaker neighbour countries and either conquered them or lost all it's attacker armies. It will then move the attackers all to the captured countries.

Problems

- sets up, deploys and fortifies without strategy, randomly
- does not choose attack targets strategically
- no defence strategy

GreyhoundAIClient - Considering the defence Greyhound client is the next iteration of the Bulldog AI - it keeps its aggressive behaviour, but knows to stop attacking if its attacking country gets weaker than the defending country. It will also deploy and fortify defensively, trying to match opponent strength on all its borders.

Problems

- sets up without strategy, randomly
- does not choose attack targets strategically
- spreads its armies too thin
- may get into an arms race in which no one will ever attack

GreatDaneAIClient - Set up with a strategy GreatDane is aware of the value of having connected territories, and the value of owning Australia in the beginning for defence purposes and getting an army production started. When setting up, it will try to minimize enemy borders by acquiring less connected territories, and territories connected with currently owned one. When deploying, it spreads its armies into two groups, trying to match enemy strength at its borders.

Problems

- does not choose attack targets strategically, just chooses weaker country, does not consider it may have additional armies capable of attacking in adjacent territories.
- may get into an arms race in which no one will ever attack

CommandRaterAIClient - Rating the decisions This client uses a slightly different approach than the others. It has a strategy for every country, Hold or Capture. For each stage, the strategies all suggest a move that would benefit its goal. Then the moves are all rated using data about the troops (enemy and friend) in and around them, and multiplying them by definable weights. Choosing the right weights can result in a powerful, strategy- and board-aware player.

Problems

- Hard to choose correct weights
- choice of the considered variables limits *how* good the player can get, but the more weights are needed, the harder it gets to choose good ones.

EvolvedCommandRaterAIClient - Letting the weights choose themselves This is an extension of the CommandRaterAIClient. It uses the same concept, but the weights for rating the commands are determined by a genetic algorithm. To realise that, I reused the genetic algorithm framework I had used for the Starcraft practical in first year and adapted it.

A generation consists of 100 individuals, one of which is always the static CommandRaterAI to prevent degradation below this instance. Every individual represents a combination of weights for rating commands (Every command client needs 15 weights). Their "fitness" is evaluated by playing 6 player games against each other, for every individual 12 games are evaluated and they score player points. A game is capped to 6000 turns to speed up the process of evolution. If a game is ended before anyone wins, every player gets one point for every worse player plus one base point. If a game ends with somebody winning, the winner gets twice the amount of opponents as points, and everyone else get points like it was described previously. This is multiplied by an "environment" factor, which is the square root of the amount of player points the individual can score in a game against Chihuahua, Bulldog, GreatDane, Greyhound and CommandRaterAI.

After the fitness of all individuals in the generation is evaluated the next generation is constructed as follows:

- 1 individual is the static CommandRaterAI
- 9 completely random, new individuals are introduced (preventing local minima).
- the top 10 individuals are preserved for the next generation
- 15 Slightly mutated versions of the elite are added to find better solutions that lie close by.
- 55 crosses randomly made between the top 20 individuals are added
- 10 Slightly mutated crosses are added

Then the next iteration is started. The training can be stopped and continued, as every generation is saved to a file.

Problems

- Needs a long time to evolve
- Can get stuck at local minima
- choice of the considered variables limits *how* good the player can get, but the more weights are needed, the harder it gets to choose good ones.

3.4.2 Analysing AI Performance

To measure the performance of the AIs, we analysed their placements in 100 6 player games. These were the placements. The rating is calculated by giving 1 point for every 6th place, 2 for every 5th and so on, apart from 1st place, which gives 12 points (because winning is valued much higher than losing last).

Player	1st	2nd	3rd	4th	5th	6th	Rating
Bulldog	11	24	8	15	13	29	384
Greyhound	1	10	13	18	24	34	250
GreatDane	22	34	22	14	7	1	579
Chihuahua	1	18	22	18	27	14	312
CommandRater	43	3	19	17	10	8	686
CommandRaterEvolved	22	11	16	18	19	14	489

We can clearly see in 3.4.2, the CommandRater scored the best in the rating. This is as expected, as the CommandRaterEvolved had only evolved for 4 Generations at this point. Evolving it longer would have improved it more probably, but there was not enough time for that. It is surprising that Greyhound does worse than even Chihuahua. This can probably be attributed to the fact that it spreads out its armies far, quickly falling victim to the other players aggressive behaviour. Greyhound was designed with 1v1 play in mind. We can also see that although there are clear patterns, that randomness plays a huge role in determining winners, as even the random player managed to win a game.

It has to be noted that CommandRater could probably be improved by better weights.

3.5 User Interface Design

The intention behind the User Interface was to create a User Experience which would enable a human player to configure and effortlessly play and/or watch a game of risk. The User Interface Design consisted of the following key elements:

Jetty Server The Java-based Jetty Server was able to provide two key pieces of functionality. Firstly, it served up the static html, JavaScript, css and images needed for the Web Client. Secondly, it provided an API with the ability to make moves, fetch the latest copy of the game state and interact with the Game Model. The API functionality was defined in the ParamHandler class. The API allowed the web client to make AJAX requests and update the User Interface appropriately.

Kinetic.js JavaScript Canvas Framework The web client regularly updates itself with the latest game state by making AJAX requests to the Jetty Server. The risk board is then graphically rendered from the game state using Kinetic.js which is an HTML5 Canvas JavaScript framework, which provided useful high performance event detection.

The following code written by third parties must be acknowledged:

- From `http://dev.filkor.org/RiskMap`, the following code/data has either been directly used or adapted:
  - assetManager.js
  - coordinates.js
  - gameData.js
  - kinetic-v4.4.2.min.js
  - main.js
  - paths.js
  - risk.js
  - names.png
  - map_grey_new.jpg

- The following standard libraries have also been employed in the web client:
  - Twitter Bootstrap : Common HTML, CSS and JavaScript. (such as buttons and font)
  - jQuery: A library providing numerous JavaScript functions.

The key screens making up the web client are:

The Launcher Screen Provides the ability to configure game options, host a game and connect to another game. (index.html) See Figure 1.

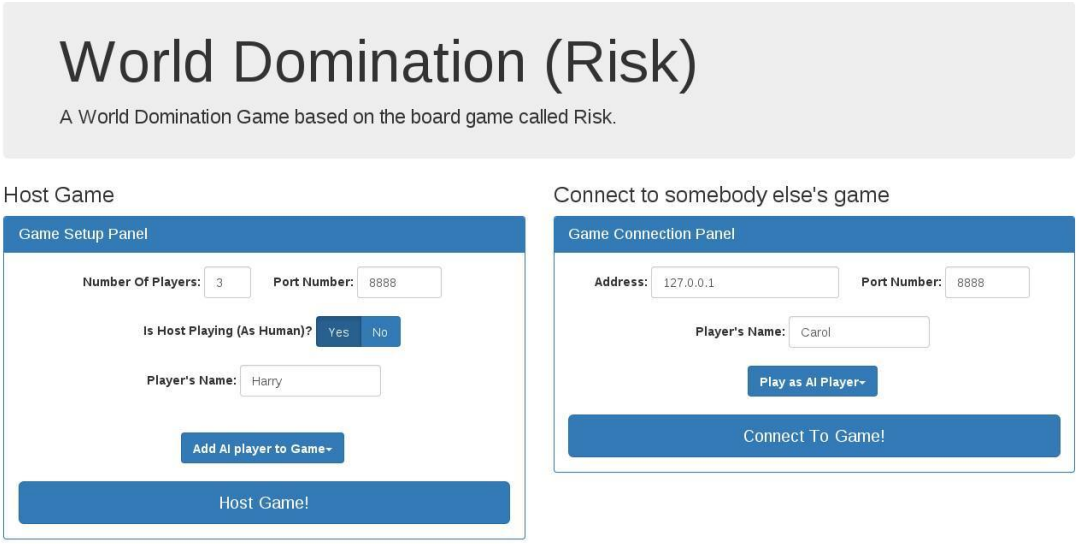

Figure 1: Screenshot of Launcher Screen (index.html served from the Web Client)

The Play Game Interface Screen Provides step by step instructions to the player informing them of the current state of the game as well as the necessary action(s) needed for the game from the player. See Figure 2.

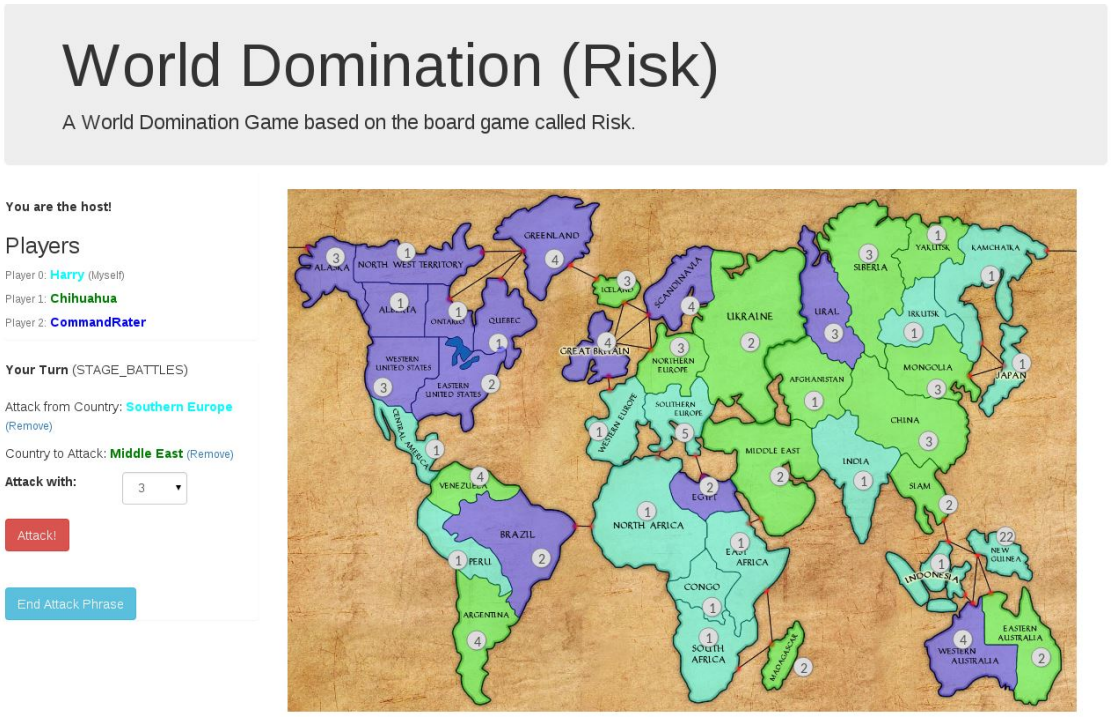

Figure 2: Screenshot of game play interface which allows a human player to make a move (play.html served from the Web Client)

The Game Watching Screen Allows Games to be watched without requiring any interaction with the user. Used for watching game where the host isn't playing or connecting to another game as an AI. See Figure 3.

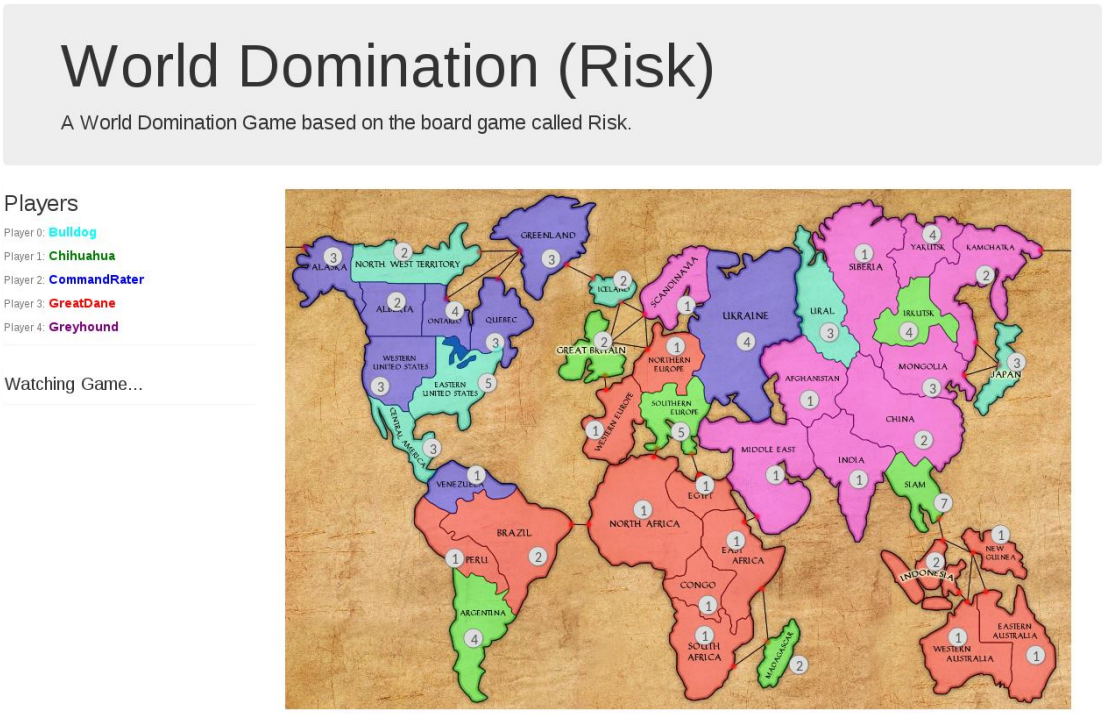

Figure 3: Screenshot of Game Watching Mode observing a game between multiple AIs (server.html served from the Web Client)

The Web Client has purposely been designed using responsive JavaScript frameworks (Bootstrap and Kinetic.js) in order that the Web Client should be usable by a full range of desktop, laptop, tablet and mobile devices with a modern web browsers. Figure 4 shows the game being played on a tablet.

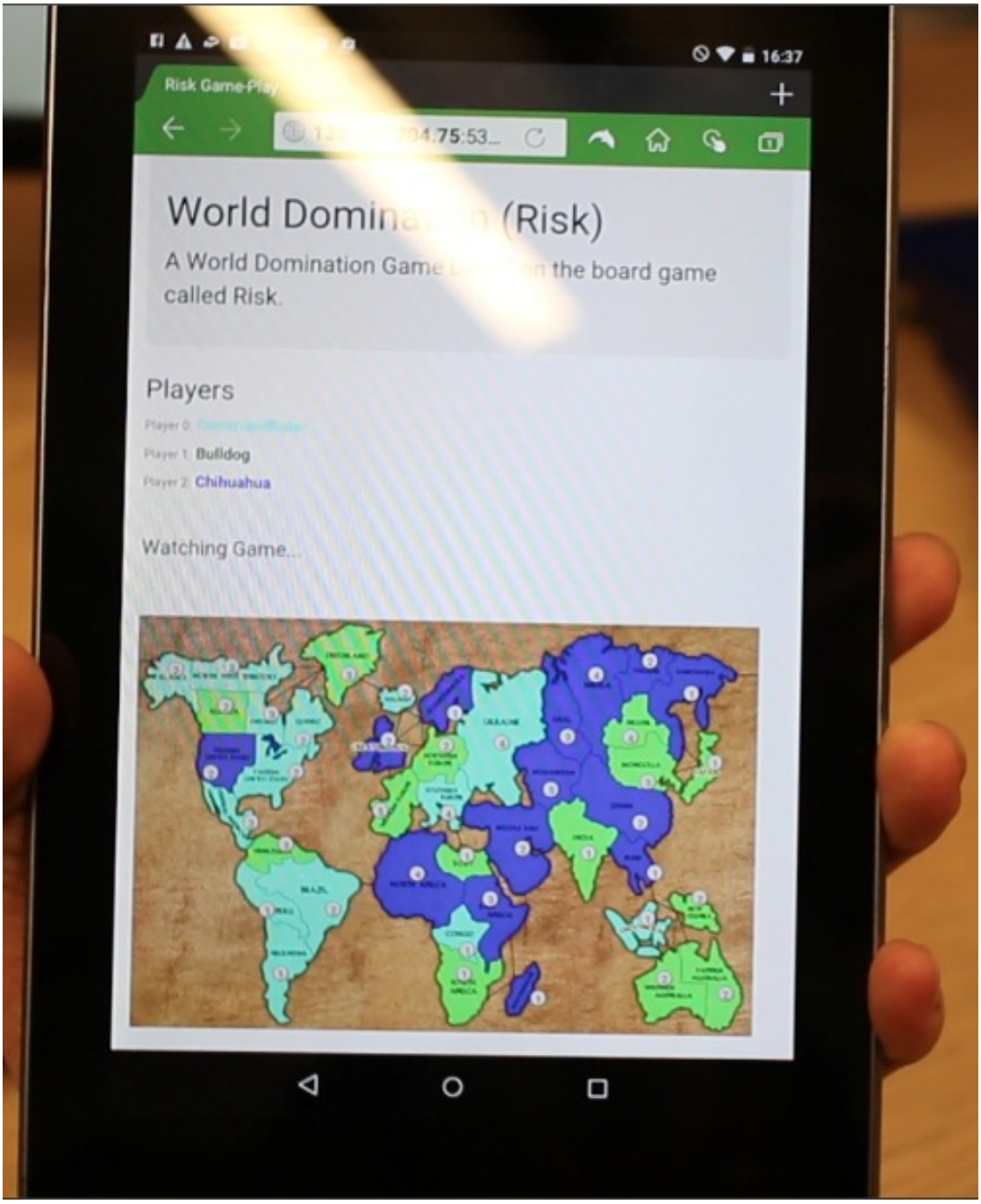

Figure 4: Photograph of Risk being played on a Nexus Tablet

3.6 Network communication

Networking was part of the core functionality of the project. The protocol established by the standards committee has been implemented to allow inter-group game play.

3.6.1 Host

The host ran on multiple threads by default. When the host (ServerSocketHandler) is started, it will created a new object for each client that connects, until the game was started. The game starts as soon as the specified number of players have connected. Any player that becomes disconnected in the game initialisation stage will be removed from the player list but they will be counted against the number of players that have connected. When the game has started, the objects that take care of each client will be started in a separate thread. From this stage, normal commands will be processed in HostForwarder.

HostForwarder, as stated above, only takes care of one client's actions. It will run the playgame method and process each incoming command according to the protocol. There are only two processing functions that have significantly different functionality:

processAttackCommand This method will process an incoming attack command. It will parse the attack request and wait until the player required to defend sends a defend command. At this point it will wait for all clients currently in the game to send their roll hashes and roll numbers, by invoking the getRolls method in the same class and on all other Threads (calling remote threads is done via the MessageQueue).

getRolls This method will wait for getting all `roll_hash` and `roll_number` commands from all clients that are currently connected to the game. When all are received, it will set getRolls variable to false and return. This ensures that by the point this method exits, all threads have finished processing all dice roll related commands. At the exit point of this method, `seed.finalize()` can be called safely. This will only happen in one thread at all points in the game, twice in `ServerSocketHandler`, and all the other times in `processAttackCommand`. Notice that all the HostForwarder threads share the same state object in the host so

only the incoming messages from each player and not from other players that get forwarded to this client.

3.6.2 Client

The client works in essentially the same way as the host in the sense that it processes the commands arriving from a peer and applies them to the state after checking validity. This means that there are a lot of things that are similar or identical in the way the two classes work. Even though this is the case, most of these processing functions use a lot of internal state and therefore cannot be extracted easily. This is thus a remaining issue with the program.

There are essentially two threads that run when a client is initiated within the game. One will only be pulling commands from the clients and one will be pushing them. This makes the client practically a lot easier to implement than the host where concurrency has to be maintained. The client will implement poll and push methods and will retain a queue of messages that have to be processed. This makes it slightly inefficient but a lot safer against a lot of messages being sent and received over the network.

The client discards acknowledgements. This is for faster processing and to avoid deadlocks. Even though it discards them, the client adheres to the protocol and will give priority to any message that requires an acknowledgement in the process, so the acknowledgements are sent out as soon as the incoming message is read and it is decided that an ack command is needed to be sent.

4 Evaluation

JH project as a whole was not particularly an easy task. We as a team, worked together to get this large project done. There has been some positives and negatives on how we worked and how our end product turned out to be. In this section, the project is evaluated from different aspects.

4.1 Difficulties and Challenges

4.1.1 Collaboration

Collaboration was the key in this project as it required collaboration at multiple levels: collaboration between groups to produce a reliable and well-defined protocol and collaboration within the group to produce the final product. Inter-group collaboration was not trivial, there were numerous changes to the protocol right up till 1-2 weeks before the deadline.

Intra-group communication. however, was a lot better. We met regularly, posted updates and issues frequently to the facebook group that we set up at the beginning of the year and helped each other finding bugs. Without this approach, completion of the project would not have been possible, especially on the level that we presented above.

4.2 Remaining issues

4.2.1 Networking

The most important issue that would need further work was that the protocol specification's ambiguities had led to problems in the inter-group communications. Because the issue affected most groups, we did not have the chance to test with many groups. Those who we tested with confirmed that there might be issues when unspecified sections of the protocol (e.g. how the dice rolls are exactly calculated from the shared seeded random number generator's `nextInt` function's return value) were touched. We were very positive about this and we were satisfied with our own implementation successfully running on multiple machines and playing a game.

4.2.2 Possible improvements to the AIs

There is significant improvement that could be implemented in the AIs. Some of them are listed here:

- Train CommandRater for longer
- Add more weights to CommandRater (for example individual country importance)

- Add country strategies
- Add tree searches to look into future moves
- Design an AI to learn from games

5 Testing Summary

There were a number of JUnit cases made to test the program. These tests were created early when only the game model was working and some of them has been changed to implement slight changes to the protocol that were reflected in the game model. All of our unit tests pass. Most of the game logic is implemented in actions, therefore we concentrated on testing these classes. Actions have 77% class coverage, 77% method coverage and 62% line coverage with Junit Tests.

All other testing of the program has been done by playing complete games across our AI players or humans with the web client. This showed slight bugs as mentioned before which we managed to correct. Some bugs remain in the code, in particular with communication with other implementations. Because most of the groups (among us) only finished their product on the last day, we did not have a chance to correct irregularities across different implementations. We have not tested with all of the implementations though.

6 Conclusions

The project consisted of four core elements: Game Model, Networking, Web interface and AI. Each component has been successfully constructed and brought together to produce a fully functional final product.

Of significant note where the project has gone beyond its initial scope are two particular features. Firstly, the development of advanced AI using a genetic algorithm framework. Secondly, a flexible cross-platform game interface has been developed.

The largest difficult was inter-group communication and working with a dynamically evovling protocol.

A future direction for the project could be investigating how a more straightforward and less ambiguous protocol could be agreed and developed by multiple groups as well as looking at how it could be extended to facilitate additional functionality such as text and voice chat integration for the game.

7 Appendices

7.1 Changes from the initial plan

The initial plan document was written back in October. As the project progressed further, a number of things has changed. One of the changes we made was, although we were looking into C++ for AI, we've changed to stick with JAVA. However, JavaScript was used to run UI as suggested in the initial plan. Although we originally mentioned TDD (Test Driven Development), we ended up writing tests after codes were written. Even though we did this, as mentioned above, we have a good code coverage on the game engine and some of the early bugs we had were actually uncovered by the tests. Another change was the repository, we began by using Mercurial, ended up using git on GitHub private repository as git's merge function handled conflicts better. TravisCI was used instead of Jenkins for continuous integration, and we integrated this with github so we all got emails when tests or build failed. Other than that most of the original plan stayed as stated in the initial plan.

There has been number of architectural changes, one of which is the network model. In the initial plan, it was meant to be P2P architecture. However as the protocols developed, the architecture has changed into ad-hoc one-to-many architecture. This means that one computer will always act as a server and this computer will detect any malicious activity.

7.2 Differences from the protocol

As far as we could test, there were no differences between the protocol and our implementation of it. Even though this was the case, groups that claimed the same have failed to communicate with our implementation for long enough without error to finish a game. This might, however, mean that they have implemented the protocol ini a slightly dif-

ferent way than us. This is possible because the protocol is ambiguous at points. These points include rolling dice and giving IDs to risk cards (in particular, the wild cards do not have predefined IDs). This meant that a move did not validate on one end of the channel but validated on the other end. This obviously resulted in failures.

References

[1] Wolf, *An Intelligent Artificial Player for the Game of Risk*, Darmstadt University of Technology, 2005.