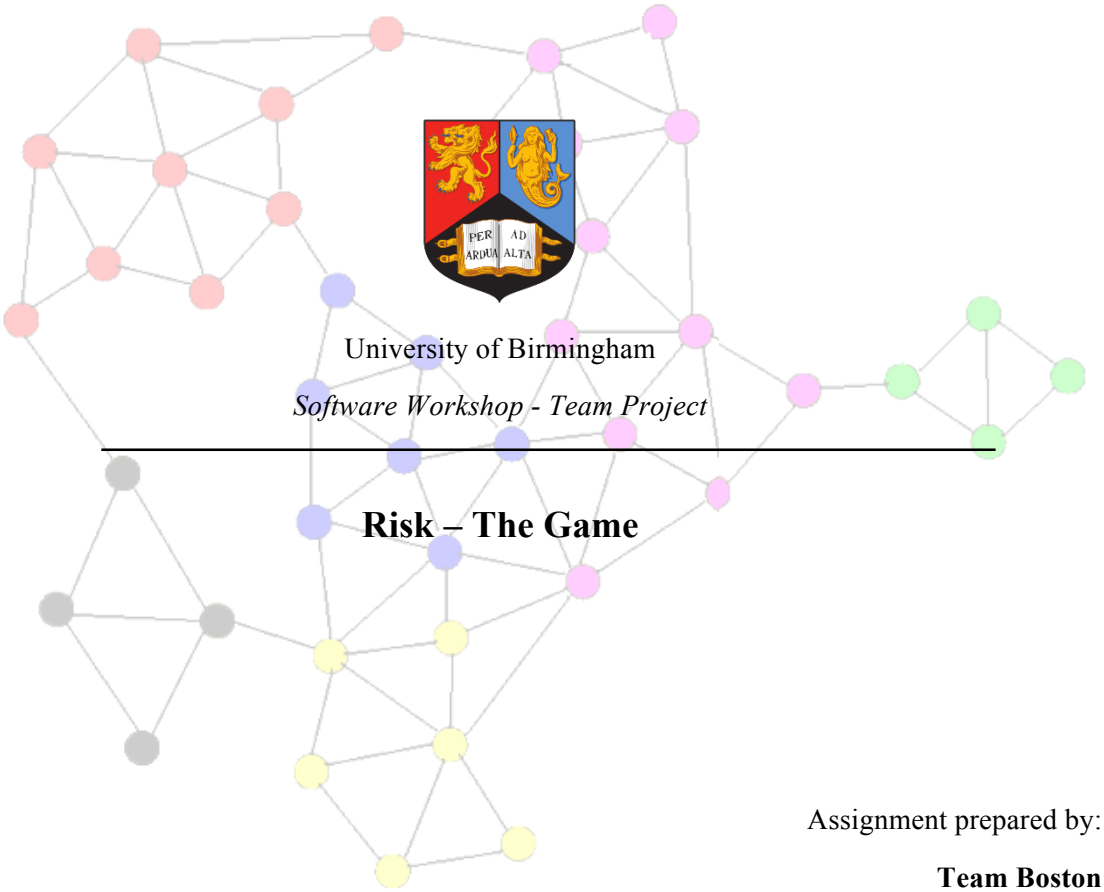


University of Birmingham

Software Workshop - Team Project

Risk – The Game

Assignment prepared by:

Team Boston

Mark Alston

Abdikhaliq Timer

Khalid Ramadan

Matt Byrne

Oliver Kamperis

TABLE OF CONTENTS

1 INTRODUCTION

This report has been written to show the technical side of how our system was created. It involves in depth analysis of how the different components of our game were designed, created and then tested. The game Risk, created by Team Boston, is one that challenges all aspects of creating an ambitious game. The team focused heavily upon the following sections; server, client, database, game logic and GUI.

Our project aims to implement the board game risk as a 2 player online game. Risk is a game where two players compete for 'world domination' by taking over their opponent's territories.

In our system, we have incorporated a database that is accessed through JDBC and stores the details of users as well as saved games, a multithreaded server, a client that communicates to the server through Object input/output streams, the game functionality and a GUI that displays the risk map and current state of the game. Our system also includes a GUI that displays the number of threads currently running on the server and a GUI that guides the user through the registration/login process. Once a user logs in they are then able to join/create lobbies and chat to other users before starting a game.

Each part of the system was discussed and the design outlined before being developed. As much testing was completed as time allowed and the test plan indicates further tests that we would conduct to ensure our system's robustness.

1.1 SPECIFICATION OF THE CLIENT

This report aims to cover all the requirements needed from our systems specification provided to us by the Uday Reddy (small team project assignment). From this document, we identified exactly what is needed within our system.

Our system should allow our user to:

1. Have a client and server running on different machines
2. The connection between the server and the client should implement the following:
    a. Sockets
    b. Threads
    c. Synchronization
    d. Avoid races/deadlocks
3. Implement a polish GUI
4. Access a database from Java
5. Use version control
6. XML/DTD/Schemas are recommended
7. Test plan

1.2 FUNCTIONAL REQUIREMENTS

1.2.1 *Database Functional Requirements*

1. Our system will have a database in the form of a Relational Database Management System which communicates with the server through JDBC.
2. The database will contain a 'User' table with attributes such as id (PrimaryKey), user_name, password, email and rating.
3. The database will contain a 'Game' table with attributes such as id, start_time, end_time, number_of_players, result_id(FK) and game_time (FK).
4. The database will contain a 'Move' table with attributes such as id, participant_id,(FK) move_type, game_id(FK), from_position and to_position.

5. The database will contain a 'Participant' table with attributes such as id, user_id (FK) and game_id (FK).
6. The database will contain a 'Piece' table with attributes such as id and type.
7. The database will contain a 'Result' table with attributes such as id, game_id (FK) and description.

1.2.2 *GUI Functional and non-functional Requirements*

1.2.2.1 Functional Requirements of the GUI - what the GUI should do:

1. User will be able to start a game with another user once he has logged into our system.
2. User will be able to identify whether it is his turn, or the other player's turn.
3. User should be able to identify who he is playing against.
4. The user should be able to start a new game, exit out of our game, minimize our game.
5. The user should be able to identify what turn phase of the game the user is in, hence whether they are in a draft, attack or fortify stage.
6. The user should be able to identify how many troops they can deploy, attack with, fortify.
7. The user should be able to see the territories that belong to them, whilst also being able to easily identify the territories that belong to the other user.
8. The user should be able to interact with the map, depending on the turn phase they are in.
9. The user should only be able to move to the next turn phase if they have finished their turn phase (e.g. running out of troops to deploy in the draft stage).
10. The system should not let show the user any available troops to move/attack/deploy with if it's not their turn.
11. The system should not let users deploy/attack/fortify with more troops than they are allowed to deploy/attack/fortify with. Hence, adhering to the rules always.
12. The system should adhere to the game rules always.
13. The system should send the correct game updates to the other player.
14. The system should play music as soon as the game is loaded up.
15. The system should have a button that can mute the music.

1.2.2.2 The non-functional requirements of the GUI are - how the system works:

1. The system's performance should be consistent for both players:
    a. The system should send constant updates between the two users so that both users can see the same updates.
    b. The system should respond to a user's request in adequate time.
2. The system should allow for scalability and capacity - allowing for two or more players to play a single game.
3. The game should be available to both users, if one disconnects, it should not automatically kick the other player, but it should update the other user.
4. The GUI should look the same on all computers.
5. The GUI should have an easy way to save a game and exit out of a game.

2 GAME ENGINE

2.1 RISK – THE GAME

The game is intended to be played between 2-6 players and initially starts off with allocating a fixed number of troops to all players, having each select a territory to own in turns & depositing a single troop unit on it until all territories are taken. Thereafter, any number of the troops that are left over can be dropped off (still in turns) until every troop unit has been allocated a territory.

Thereafter, the game truly commences, with each players' turn following a DRAFT-ATTACK-FORTIFY structure. Note that only the player whose turn it is can make moves and other players will thus have to wait until it is theirs.

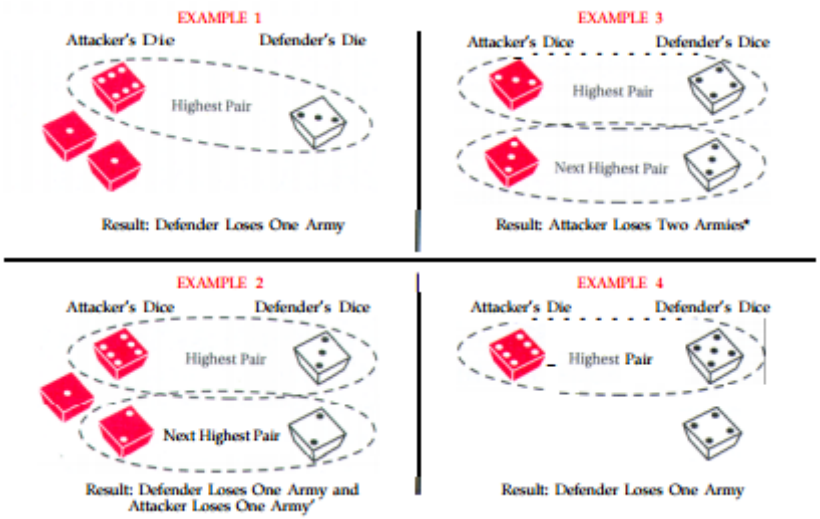
2.1.1 *DRAFT:*

At the start of every turn, the player is allocated an additional troop unit for every 3 territories owned with additional units awarded depending on whether all the territories in each continent is owned by the player (as specified by the map being used). The game ensures that, regardless of the number of territories owned, a player will have at least 3 troops to allocate when in the draft stage. Hence, for every turn, there will be troops that need to be allocated and all given troops have to be allocated to proceed to the attack stage.

2.1.2 *ATTACK:*

Once in the Attack stage, the player has the option to attack other territories and thus attempt to take them over. This involves selecting a territory from which to attack and an adjacent opponent-owned territory to attack. The attacking territory must have at least 2 troop units to attack as a successful take over requires at least one unit to be left behind and at least one to move into the new country. For every attack, the attacking player attacks with either 1, 2 or 3 troops whilst the defending player defends with 1 or 2 troops, with the numbers depending on available troops.

Dice rolls are used to decide who loses troops based on the number of troops used to attack/defend (the diagram shows the process, with the # of dice showing the # of troops allocated for the attack). If the attacker's rolled number is greater than the defender's number, then the defender loses a unit and vice versa in the opposite case (a tie is considered a win for the defender). The diagram below shows a few cases that could occur with their subsequent results with a table of all possible attacks and their results further below in the overview of the player class.



In the case that an attack leads to the defender losing all the troops on the defending territory, the attacker takes possession of the territory and moves in any number of troops into the newly-conquered country. This number is required to be at least as many troop units as were used in the attack up to whatever number would leave behind 1 unit in the territory from where the attack was initiated.

At the end of every attack, the player can decide to keep attacking the same territory, a different territory (either from the same territory or a different one), or to end the attack state of his turn. Given the constraints, it should be noted that it can be entirely possible that there is no valid attack opportunity available to the player. The player is also allowed to forego the opportunity to attack and proceed to the fortify stage without attacking, regardless if the opportunity to do so is present or not.

2.1.3 *FORTIFY:*

The player has the option to send any number of troops from one owned territory to another owned territory, if at least one unit is left behind. Note that only one fortify move is allowed and just like the attack state, the player has the option to not fortify. Regardless of his/her choice, the player's turn ends and the above repeats for subsequent players.

This continues until a player owns all territories.

Of particular note is that at any one time following the initial picking of territories, the owned territories of each player forms a partition of the map; no territory is shared between any two players and no territory is ever considered "free". In addition, any territory will have at least one unit present at any given time, regardless of any of the player moves.

There are various different versions of Risk which differ mainly in different maps being used, restrictions placed on when to fortify and additional objectives that can lead to a win for the player, but the above described is for the most part considered the standard version played by most players.

2.2 FUNCTIONAL REQUIREMENTS

Given the above description, the following functional and non-functional requirements have been designed to closely mimic the game as much as possible. Of note is the decision to automatically allocate the territories and troop numbers that are placed on these territories for each player and to only have two players. This should lead a fast-paced game and also prevents the problem caused by having only two players where player selections will effectively split the map in two.

2.2.1 *Start of Game*

- Both players should be allocated the same number of troops at the start of the game
- Both players should be allocated the same number of territories owned (if possible)

2.2.2 *Draft Stage*

- All stages below are only performed by the player with the current turn
- Number of additional troop units given to the player should reflect the total number of territories owned
- Players should be able to place any number of troops in any owned territories (as long as total number of placed troop units matches the given total)
- Draft Stage should not end until all allocated troops are placed

2.2.3 *Attack Stage*

- Player should be able to attack from any owned territory that contains more than one troop unit directly adjacent to an enemy-owned territory
- Player should be able to proceed to the Fortify without attacking if they wish to do so
- Player should be able to end their turn completely from the attack or fortify stage

2.2.4 *Fortify Stage*

- The player should be able to move any number of troops from one owned territory to another
- The number of units that are moved should be strictly less than the total number of units on the first territory
- This move should only be allowed once
- The player (if s/he wishes to do so) should have the option to not fortify
- Player's turn should end upon completion of this stage

2.2.5 *Other Requirements*

- At any time, the game map should be partitioned between the players with no territories being owned by multiple players
- The other player who's waiting for their turn should only be able to observe the results of the other player at the end of the draft stage, an attack or at the end of the fortify stage.

2.2.6 *Non-Functional Requirements*

- The game should be scalable in regards to allowing multiple players as opposed to two & using different maps
- The game should, at any time choice is taken from the player such as initial troop/territory allocation, ensure that game parity is maintained as much as possible
- The game should minimise superfluous moves/options as much as possible (skipping turnphases where moves aren't possible)

2.3 OVERVIEW OF CLASSES

2.3.1 *COUNTRY*

In deciding to emulate the board game, I felt that a good start would have been to create a Country class which simply contains the name and a reference to adjacent countries (depending naturally on the map that is used). The reasoning behind this is that the progress of a player is effectively measured by the number of countries s/he owns and a majority of the player's moves are restricted to countries adjacent to each other (such as attack). The adjacent countries are stored in a HashSet in order to help facilitate fast lookups for checks and validation of player moves. The class contains 2 constructors, with a standard taking in Strings of the country's name and names of adjacent countries whilst the other simply takes in the country's name. The second constructor is mainly used when there is no requirement for the set of adjacent countries (an equals method has been constructed to ensure that either are considered the same if the name matches). Another method

2.3.2 *MAP*

Given that Risk takes place on a map, I felt that a Map class should encapsulate all the countries for a given map, with all the countries of a Map object being stored in a TreeSet as the natural ordering is required for subsequent allocation of countries to players. A TreeMap is used to store the continents specific to that map which is to help with allocating troops for the draft stage which more closely replicates the game of Risk. A method called findCountryObject() has been created to help with the creation of countries by returning a country from the standard game map given the String of said country.

2.3.3 *DICE*

To help replicate the real-life counterpart, I've created a Dice object that each player can use for the attack stage. It uses SecureRandom to simulate dice rolls and also contains a method to simulate the comparisons done when an attack is initiated. The screenshot below shows the method (diceRolls() takes in an integer and creates an ArrayList of dice roll results in decreasing order). For example, to simulate an attack with 3 troops against 2 defending troops, diceComparison(3,2) would return either {2,0} (defending troops vanquished), {1,1} (a troop unit lost on either side) or {0,2} (a completely failed attack).

```
public int[] diceComparison(int attNumber, int defNumber){

    ArrayList<Integer> attRolls = diceRolls(attNumber), defRolls = diceRolls(defNumber);

    int faceOff = Math.min(attNumber, defNumber), attWins = 0, defWins = 0;

    for (int i = 0; i < faceOff; i++) {

        if (attRolls.get(i)>defRolls.get(i)) {

            attWins++;

        } else {

            defWins++;
        }
    }

    int[] comparisonResults = {attWins, defWins};

    return comparisonResults;

}
```

2.3.4 *TURNPHASE*

TurnPhase was created to help keep track of each individual stage that a player is in and to update the gui accordingly. It simply contains a String referencing one of the three phases a player could be in.

2.3.5 *VALIDITYCHECKS*

ValidityChecks has been created to contain all the possible checks that would be required to ensure all moves are valid and that the gui can show the various comboboxes and notices of the game state at the correct places/times.

2.3.6 *PLAYER*

A Player object is naturally intended to model a player of the game and thus contains quite a few variables to help keep track of the various stages the player could be in throughout a game in addition to methods that model the . The main variables involved are a TreeMap storing the countries the player owns with the current troop count and a few boolean fields indicating whether the player is the host, has won/lost, a Dice object to facilitate attack moves, a ValidityChecks object for checks regarding that particular player and a givenTroops count to decide the total number of allocated troops at the start of the player's turn. calculateTroops() (shown below) is what's used to calculate givenTroops, ensuring that an additional unit is given for every 3 territories owned plus an additional 5 for every continent owned. Note that a minimum of 3 is given regardless of territories owned.

```
public void calculateTroops() {

    Map map = new Map();

    int continentalTotal = 0;
    Set<Country> playersCountries = getOwnedTerritories().keySet();

    for (String continent : map.getContinents().keySet()) {

        if (playersCountries.containsAll(map.getContinents().get(continent))) continentalTotal += 5;

    }

    int allocatedTroopNumber = ((playersCountries.size() / 3) + continentalTotal);

    givenTroops = Math.max(allocatedTroopNumber, 3);

}
```

draft() is meant to facilitate the addition of troops to a specified country by simply increasing the troop count at the specified location and subsequently decreasing givenTroops by the same amount. Note that the gui restricts the troops that can be allocated to a given country in line with givenTroops, hence the lack of checks in the draft() (and subsequent methods). endDraft() simply changes the turnphase from draft to attack.

attack() is one of the more involved methods as there are a lot of restrictions imposed on the player in regards to what country can be targeted/attacked from and what number of troops can be used in an attack. In addition, as the success/failure of a given attack is entirely dependent on the number of troops used on both sides and the resulting number of dice rolls (entirely random), there are numerous outcomes, ranging from a completely failed attack leading to a loss of the attacking troops used to a takeover of the defending territory, potentially leading to the attacker winning the entire game. Given the rules governing the number of troops that can be used in attack/defense, the below table shows all the possible outcomes, where w indicates number of troops the defender loses and L indicates the number of troops lost by the attacker.

# of Attack Units	# of Def Units	Attack Winning State	Even States	Attack Losing State
3	2	W2 = {2,0}	W1, L1 = {1,1}	L2 = {0,2}
3	1	W1 = {1,0}		L1 = {0,1}
2	2	W2 = {2,0}	W1, L1 = {1,1}	L2 = {0,2}
2	1	W1 = {1,0}		L1 = {0,1}
1	2	W1 = {1,0}		L1 = {0,1}
1	1	W1 = {1,0}		L1 = {0,1}

The cells colored in green indicate cases where a takeover is possible if the defending units are the only troop units belonging to the defender on the given defending territory (the exception being the case that a win for the attacker still leaves a defending unit on the targeted territory).

With this in mind, the decision was made to automatically allocate maximum number of attacking troops possible. Whilst this limits the number of realistic moves that could be performed by players, it helps automate the process and consequently speeds up the game.

```
attNumber = Math.min(currentAttTroops - 1, 3)

defNumber = currentDefTroops > 1 ? 2 : 1;
```

The above snippet shows the aforementioned selection for the attacking number of troops whereas the defending number of troop units is simply either 2 or 1 depending on the troop number present for the defender.

```
int[] diceComparisons = playerDice.diceComparison(attNumber, defNumber)
```

Thereafter, the relevant number of dice are rolled for each player and an array is produced (shown above for each possible case). attack() thereafter uses the given array, updates the relevant territories and notifies the gui.

fortify() simply changes the number of troops in the selected countries according to the number of troops that need to be moved from one to another and notifies the gui.

2.3.7 *GAME*

The game class is what encapsulates an entire game in respect to what each player should see: 2 Player objects designated as me and other, a map specifying the map that the game is to be played on (with the standard being used as the default) and a Client and Data object to help communicate changes, allowing both game objects that each player owns to be synchronised. In addition, a troop number variable is included to decide the number of troops each unit starts off with.

A few constructors are present: the first (and main one) is intended to be used when creating a new game, the second for loading in games from the database and a third for testing purposes. The first is shown in the screenshot below.

```
public Game(String hostName, String guestName, boolean isHost, Data data, Client client) {

    this.client = client;
    this.data = client.getData();

    this.me = new Player(hostName, isHost, isHost, client);
    this.other = new Player(guestName, !isHost, !isHost, client);

    me.allocateTroops(troopNumber);
    other.allocateTroops(troopNumber);

    countryAllocation();

    Player[] players = { me, other };

    for (Player player : players) {

        troopDeployment(player);

    }

    me.calculateTroops();
    other.calculateTroops();
}
```

Note the presence of a boolean variable referred to as isHost in the constructor; this is used to determine who the host is and subsequently decides the who starts the game. Then, both players are given the troop number specified for the game, countryAllocation() decides which player owns which territory on the map and troopDeployment() spreads the troop units for each player around their owned territories respectively (the method is shown below, noting troopTotal should and always will be greater than the number of territories owned by the player).

```
private void troopDeployment(Player player) {

    int troopTotal = player.getGivenTroops();

    Set<Country> countrySet = player.getOwnedTerritories().keySet();

    Iterator<Country> it = countrySet.iterator();

    while (it.hasNext()) {

        player.updateTroopAndCountry(it.next(), 1);

        troopTotal--;
    }

    while (troopTotal != 0) {

        it = countrySet.iterator();

        while (it.hasNext() && troopTotal != 0) {

            Country country = it.next();
            int currentTroopCount = player.getOwnedTerritories().get(country);

            player.getOwnedTerritories().put(country, ++currentTroopCount);
            troopTotal--;
        }
    }
}
```

The game class also contains a few processAction methods that are called whenever the client receives an Action object from the other player (i.e. the opposite player did a move, changed turnphase etc.) which effectively mirrors the move for the player receiving the action object.

3

3 GRAPHICAL USER INTERFACE – GUI

3.1 PLANNING THE GUI DESIGN

Creating the GUI for Risk is a vital part of our system, as that is the main system. The Risk GUI would need to link the different clients to allow them to play a game of Risk. The selling point of our game is to allow a user to play a game of Risk, meaning that the game needs to work as efficiently and quickly as it can, whilst providing the user with a GUI that is visually appealing whilst also being well organized considering the user experience.

When creating the GUI, the following objectives were considered:

1. An innovative design, one which would allow a user to quickly identify the type of game Risk is.
2. Provides the right amount of information to the user. Making sure that the screen is not overloaded with information as this is bad practice of good UI design.
3. Gives users the ability to enjoy the UI by checking that we meet the requirements for Jakob Nielsen's heuristics. This will further improve our user's experience with our game.
4. The GUI would need to provide a way to combine the logic of the game with the actual interactions of the user with the system. Creating a software where multiple users can do multiple actions has meant that the GUI needed to consider all the possible actions of a user.

This section of the report will now detail both the design phase of the GUI, whilst also adding in a section for how some of the code has been written.

3.2 DESIGN PLANNING

3.2.1 *Design phase 1 - Design plan*

The first phase of the design phase was to create the famous Risk map. This section will detail how I designed the map that would be used to create our game. We felt that this was one of the most crucial aspects of the game, as this is the where the user will spend most of his time.

When creating a game such as Risk, creating it in java seemed impossible. Creating a game that is both complex and UI demanding in java meant that I had to spend a lot of time to find the best way to create a game that would make sure that all the user needs are met, whilst also making a game that looked compelling to a user.

I planned to create a design of the Risk map on the research from both the board type game of Risk and the online version.

3.2.2 *Design phase 2 – Considering the user*

After the initial design phase, I had to consider all the needs of the user. This meant creating UML diagrams that would detail and explain to us the process a user would take. This would allow me to design a map that would incorporate all the functionalities which a user would need.

Before creating the different UML diagrams, I identified the both the functional and non-functional requirements that my GUI needed: These can be found in an earlier section in this report.

3.2.3 *Design phase 3 – Designing the GUI*

The design of the GUI has had a lot of elements involved, a process that has had multiple iterations. This is mainly due to the constraints placed upon the designer, as creating a well-polished looking game is not easy in java.

The design of the GUI began by deciding that we only need one main page for the actual game, as this will make it easier for the user to understand what the user can and cannot do. The page however had to include the fundamental parts of our game:

1. The Game map itself
2. Buttons that let me deploy/attack/fortify on each of the territories
3. An area of the screen that identifies:
    a. Whose go it is
    b. What turn phase of the game we are in
    c. How many troops I have to deploy/fortify/attack with
4. A section that will send me updates from the other user
5. Chat button that will allow me to speak to the other user
6. Buttons to:
    a. Save a game
    b. Minimise the screen
    c. Exit out of a game

The list above would all have to be put onto a page that would let the user be able to identify each of those. This was the challenging part of the GUI design; how to design and create a GUI for a complex game as Risk.

The plan would be to create the GUI so that it is visually pleasing, whilst still having all the functional requirements it needs. Hence, the process of design began.

The research into how the Risk GUI could look began with looking at various other designs of the Risk game. As Risk is predominantly a board game, the first goal was to find inspirations of how our GUI could look. The following images indicated how others had designed the same game:

Board game versions of Risk:

(Link to appendix)

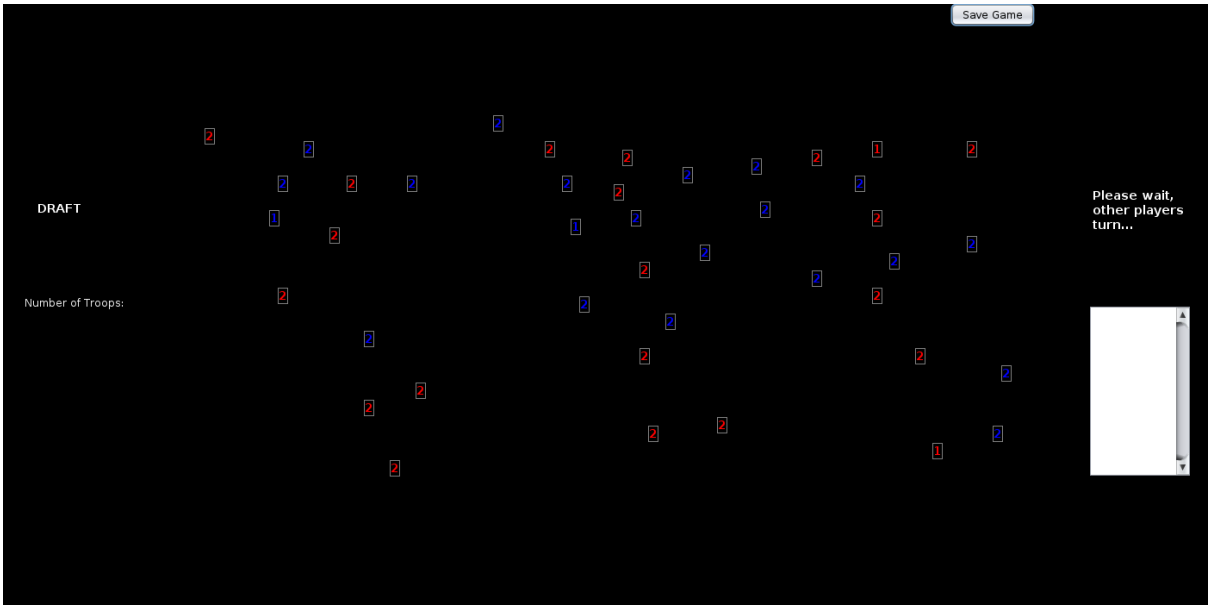
Online versions of Risk:

(Link to appendix section)

From the above you can see that the general design of the Risk map is one that follows the same convention. You have a map, and some way of identifying the troops that are located on the separate territories. This is the research that led me to better informing myself of how the Risk gui would look.

Hence, when designing the map, I had to aim towards creating a map that would stick out to a user, whilst also creating a map that was unique, yet true to its origin; the actual Risk game.

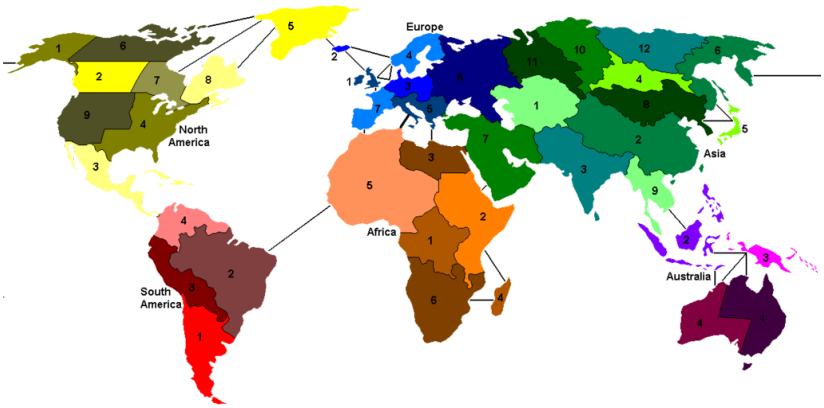
Initially, the design of the game began in eclipse, where the look and feel of the GUI was not satisfying enough, so a different route was needed to be taken.



The image above is taken from code which I created within eclipse, setting the JPanels background to a color, and adding in the specific troop numbers and other buttons which we needed. This design was one of the early implementation of our games to check if everything was working correctly, but you can see that this is not want we want from a GUI.

A different design was needed, hence the design process changed to incorporate photoshop.

As Khalid, the member of the team working on the logic side of the game, had decided that we should base the entire game on the official Risk concept, that can be found on wikipedia (https://en.wikipedia.org/wiki/Risk_(game)), I decided to use the official Risk territories map. The look of the official map is similar to the ones from the research; both the board games and the online versions.



This design led to a few iterations of designs;

The design above incorporateS everything that is needed from the functional requirements. However, from the design, you can see that more needs to be done to create a game map that looks more friendly.

The final iteration I decided to build on top from looked like the following:

The above was the final stage of general theme look of our game. However, slight modification were needed to meet all the required functionalities, such as displaying whether it is my turn

or not and other needed requirements. The final design that was created looks like the following:

This image now had followed all the requirements needed, however, from further research and input from the team, I had identified that small adjustments needed to be made.

Furthermore, a better theme was chosen to go with; A pirate theme.

The image above shows the final design of the GUI. A design that incorporates the idea of Risk, whilst also providing the user with an elegant UI and a beautiful design.

A further idea came to me whilst creating the GUI, to have a night mode, the following was created:

This user would simply press a button that stated 'night mode', which would then show the user the same version of the game, but the background of the game is inverted. This cool feature is an additional feature that we hope can help the user who finds our design to be too bright.

3.2.4 *Design phase 4 – Considering the user*

(https://www.nngroup.com/articles/ten-usability-heuristics/)

After designing the GUI, I felt it was important to answer whether our system incorporates all the needs of the user. This could also be checked by checking it against the well known Jakob Nielsen's 10 general principles of interaction design.

3.2.4.1 Visibility of system status

Does the system keep the user informed about what is going on, through appropriate feedback within reasonable time?

The system that we currently have considers the above by implementing a text box at every stage of the entire system. This includes having an area on my screen, within the history section, that constantly updates the user with information from the other user. The system has also been designed in such a way that the user only has access to functions which they are allowed to have access to, for example, if it's not their turn, they won't be shown buttons that the other user will have access to.

3.2.4.2 Match between system and the real world

Does our system speak the user's language? Not system/object-orientated terms?

Yes, the GUI has been designed with this in mind. Hence, there are terms on the screen such as "Number of troops owned: " that will display the number of troops that you currently own.

3.2.4.3 User control and freedom

What if a user clicks something by mistake?

This was considered since the beginning of designing this map. The GUI works in such a way that it only shows the correct information to the user. This reduces the errors, as it does not allow the user to get to a point where an error can occur. This however will be heavily tested.

3.2.4.4 Consistency and Standards

User should not have to wonder whether different words/situations/actions mean the same thing.

The assumption which we make are that the user will be familiar with the game before the game starts, this will mean that the user should understand the terminology used in the game, these include words such as: draft, attack and fortify phases.

3.2.4.5 Error prevention

Is your GUI carefully designed to prevent problems from occurring in the first place?

One of the fundamental heuristic which we incorporated into our system from the initial design phase was the create a system that only shows the user possibilities that they should be able to do, for example, if it's not the users turn, they should not be shown the other players buttons.

3.2.4.6 Recognition rather than recall

The user should not have to remember a lot, it should be saved by the system as objects, instructions for the system should be easy to find

Our design will show the user only the necessary buttons, so for example if their turn has ended, a button will appear that they press to move onto the attack phase. Hence, we hold the user's hand throughout the game, making sure they follow the rules adequately.

3.2.4.7 Flexibility and efficiency of use

Do you tailor the game to the skill level of the player, for example someone who is a novice or an expert?

For this system, the game itself treats everyone to be at the same level. However, if you are an expert at Risk, the game will be quick to get into, whilst also providing means for a beginner to easily access the information which they need.

3.2.4.8 Aesthetic and minimalist design

There should not be any irrelevant information?

There are no components of this design that are unnecessary. The GUI also auto hides information which is not important to the user depending on the state of the game. For example, if the users turn has ended, it will hide all buttons on the map, displaying only the number of troops you have on each territory.

3.2.4.9 Help users recognise, diagnose and recover from errors

Error messages should be displayed in clear english?

The errors are represented in a logical way, however, the system design is one that centres its aim into not giving the user an option to make an error, hence all the handling is done before anything is shown to the user.

3.2.5 ***Design phase 5 – How the game map will work***

This section is directly linked to the rules of the game. However, this section will focus more on the interaction that the user will have with the game map. The game has three main phases which it cycles through until there is a clear winner. These stages are:

- Draft phase: This stage is where the user will have to deploy the troops which he has been given to the territories that are owned by him. The rules for this stage are explained further within this

report in the first part of this report. Furthermore, the user will only have an option to deploy troops if it is the users turn, and if it is not, the user will not have any option to deploy troops. This reduces the potential errors that can be caused by the user - one of the heuristics I have aimed to solve. The user whose turn it is will however have options to deploy troops to his own territories. As troops are deployed to the users owned territories, the user is shown an updated number of troops which he can deploy.

- Attack phase: within this phase, if it is the users turn, the user will be shown combo boxes on countries which they own, and they are then shown countries as strings within the combo box drop down list, where they can select a country to attack. The rules of this phase are explained in more depth within the Risk rules section of this report.
- Fortify phase: Similar to before, this stage of the game allows a users to move troops from one of their own territories, to another territory. However, there are rules which must be adhered to which can be found in earlier part of this report.

3.2.6 *Design phase 6 – How the logic of the game will be implemented within the GUI*

Creating the GUI for this complex game was a harder task than we initially thought. The challenge of combining the logic of the code to the GUI, then combine it to the client which is running of the server.

3.2.6.1 Combo Box code

The code had to be written in such a way that all the combo boxes and labels that indicate to the user their number of troops, had to be dynamically updated. This meant writing the code that could first be easily modified due to the complexity of this game, and having code that could easily be duplicated.

As we had created combo boxes, these had repeating lines of code which I determined could be written better within one method;

```
/**
 * this method sets up each of the combo boxes in the correct location.
 * It sets the combo box to have the same properties on the entirety of the map
 *
 * @param comboBoxName this is the name of the combo box that needs to bet modified
 * @param country The country object that the combo box belongs to
 * @param nameOfCountry this is the string name of the country
 * @param x This is the x position of the combo box
 * @param y This is the y position of the combo box
 * @param width this is the width of the combo box
 * @param height this is the height of the combo box
 *
 */
public void setupComboBox(JComboBox<String> comboBoxName, Country country, String nameOfCountry, int x, int y, int width, int height) {

    // Should this be accessible? - use turn phase
    // if its not my country, this will be blank
    comboBoxName.setVisible(myTurnButtonVisible(country));

    // Set colour of this box
    comboBoxName.setBackground(setComboBoxBackgroundColor(country));

    // number of rows in the drop down box
    comboBoxName.setMaximumRowCount(getMaximumRowCount(country));

    // what is written in each row of the drop down box
    if (me.turnphase.getTurnPhase().equals("draft")) {
        //set combo boxes to number of troops in that country
        comboBoxName.setModel(new DefaultComboBoxModel<>(setRowCountPopulation(country)));
    } else if (me.turnphase.getTurnPhase().equals("attack")) {
        //set combo boxes to the number of troops in adjacent countries that belong to me
        comboBoxName.setModel(new DefaultComboBoxModel<>(setRowCountPopulation(country)));
    } else if (me.turnphase.getTurnPhase().equals("fortify")) {
        //set combo boxes to the same as the drafting stage
        comboBoxName.setModel(new DefaultComboBoxModel<>(setRowCountPopulation(country)));
    }

    // see the name of the country when you hover over a combo boxes
    comboBoxName.setToolTipText(nameOfCountry);

    // set border for the combo boxes
    comboBoxName.setBorder(null);

    //Add it to our panel - at the correct position
    jLayeredPane3.add(comboBoxName, new AbsoluteConstraints(x, y, width, height));

}
```

The code above allows me to create combo boxes easily, that all share the following:

1. setVisibility(...); this checks whether this combobox should be shown to the user. This considers the turn of the player, and also the stage of the game which we are in.

2. setBackground(...); this checks who the country belongs to, and sets the correct colour of the combo box.
3. setMaximumRowCount(...) and setRowCountPopulation(...) sets the number of rows for the combo box, and set the content of the database.
4. There also are extra methods that shows the name of the country which the combo box belongs to when you hover over the combo box.

The combo boxes are generated in a way to allow for easy creation of all other combo boxes which I need. After, an Action Listeners is added to that specific combo box.

I also use absolute constraints: which helped with the design of our GUI as it allowed me to place components exactly where I wanted them to go, as it substitutes the "null" layout, where there are no constraints on an object.

3.2.6.2 Territory labels

These works in the same fashion as the combo boxes. I had created a method that would remove the duplication of code;

```
/**
 * This creates the labels showing the number of troops on each territory.
 *
 * @param labelName The name of the label; the country name
 * @param countryname The name of the country belonging to this label
 * @param x the x coordinate of this label
 * @param y the y coordinate of this label
 */
public void setupTerritoriesTroopLabel(JLabel labelName, Country countryname, int x, int y ) {
    //sets the font to be the same which I have used throughout this GUI.
    labelName.setFont(riskFont);


    //sets the colour of the text - if its my country or if its not my country
    //If I own the country
    if (me.checker.ownsCountry(countryname)) {
        //The colour of this label should be my color
        labelName.setForeground(colorMine);
    } else {
        //should be colour of other player
        labelName.setForeground(colorOther);
    }


    //set the troop number of that country
    //if its my country
    if (me.getOwnedTerritories().containsKey(countryname)) {
        //return number of troops that the country has
        labelName.setText(Integer.toString(me.getOwnedTerritories().get(countryname)));
    } else {
        labelName.setText(Integer.toString(other.getOwnedTerritories().get(countryname)));
    }

    //small border around the number
    Border border = LineBorder.createGrayLineBorder();
    labelName.setBorder(border);


    //sets the location of the text but + 39 due to the locations being the same as the combobox
    jLayeredPane3.add(labelName, new AbsoluteConstraints( (x+39), y, -1, -1));

}
```

Again, the methods here are self-explanatory. Where this method was created to make the code cleaner, and to allow for the easy creation of all the different labels.

3.2.6.3 Labels and other buttons that will update the user on the state of the game

These include:

1. Whose turn it is
2. What turn phase we are in
3. How many troops we can deploy

The above have all been implemented in the GUI, and tell the user exactly what they need to know.

3.2.7 *Design phase 7 – Extra functionality that the GUI provides*

This section will explain extra functions that that improve the overall user experience.

These include:

1. Music button
2. Night mode button
3. Draggable interface
4. Custom minimise and exit buttons
5. Buttons that only appear on the screen when the user gets to a certain stage

For example, another feature of the code written for the GUI are using the design of an image as buttons, and creating buttons that are invisible to the user, that act as means for the user to interact with the GUI. This created a design that is both innovative, and rather unique. Below are images showing three places where such code is used, that all have custom code to do the exact feature which we would like.

3.2.7.1 Draggable screen

The yellow box shows where on the screen you can hold, and which will allow you to drag the screen around. This was written by combining an invisible button above that region.

```
private void moveScreenJLabelMouseDragged(MouseEvent evt) {
    int xLocation = evt.getXOnScreen();
    int yLocation = evt.getYOnScreen();

    this.setLocation(xLocation-xMouse, yLocation-yMouse);

}

private void moveScreenJLabelMousePressed(MouseEvent evt) {
    xMouse = evt.getX();
    yMouse = evt.getY();
}
```

The above code shows how we are able to correctly move the screen around depending on the location of the mouse.

3.2.7.2 Minimise screen & exit game

From the red box, you can see that there are two unique buttons, these buttons perform actions that are easy to understand to the user; to minimise the screen or to exit the game.

```
private void closeJLabel1MouseClicked(MouseEvent evt) {
    System.exit(0);
    client.sendRequest(new LeaveLobby(""));
}
```

The above code was shows that when the close button has been clicked, it fires off an event which is handled within this Mouse Clicked method. It exits the system, whilst also notifying that the client has disconnected.

4 SERVER

The server implemented in this project is a multithreaded server capable of serving up to eighteen dedicated services, over a local network. The server can be simplified into five distinct sections listed below;

1) Acceptor Thread (*ServerMain.java*) – This thread runs constantly using a while loop. Its primary responsibility is to accept new connections and create a new service for each client. Threads are allocated using an *ExecutorServive.newFixedThreadPool()*. In the event more than eighteen clients request a connection the thread pool is large enough to accept all their requests, however when it comes to authentication if more that eighteen clients are connected the next client will not be successfully authenticated. This method of restricting clients allows clearer communication between the end user and the server. The worker thread (*ClientThread*) is an inner class of *ServerMain.java* which implements runnable. It is in this worker thread where authentication, registration, login and game play occurs.

2) Authentication – Within the *ClientThread* class authentication occurs. Authentication is only successful if the following two conditions are met; firstly it requires a shared key, in the form of a string, and the keys have to match. Secondly there must be less than eighteen logged in players. If the client successfully authenticates, an *Accept()* object is sent back on the object output stream otherwise a *Deny()* object is sent. Both contain a message which is displayed to the user. A sequence diagram of the authentication process can be seen below in Figure 1. Once authentication is successful the thread *ServerPing* is started on a different port number. This thread sends ping's to the client on a regular interval and expects a pong back. If this stops, the server will time the client out and terminate the thread after performing a clean-up.

3) Login / Register – Within the *ClientThread* class login and registration occurs. The login process is implemented in a while loop. It is only broken when the ping – pong thread between the client – server terminates or when the user successfully logs in. The server passes the login details to the database manager which checks if the details are correct. The *DatabaseManager* instance is created in the server acceptor thread and passed to each service. The *checkLogin()* method called is an instance method of the *DatabaseManager* instance created in the acceptor thread. The server and database manager expects as a precondition that the client, filters non alphanumeric inputs before sending. A sequence diagram of the registration / login process can be seen below in Figure 1.

4) Object Passing – Once logged in there are a number of paths the player may take. In order to take the correct course of action a single object is read in at a time on the object input stream. This is done in a while loop, which runs until either the user logs off or the ping – pong thread terminates. A sequence digram of the object passing can be seen in Figure 2. The while loop can be seen below;

```
pingPong:
while(pingPong.isAlive() && logout == false) {
	//ServerLogic class handles the inputs
	logout = ServerLogic.checkForInput(...);
}

public static boolean checkForInput(...) {
	Object readIn = in.readObject();
	if(readIn instanceof CreateLobby) { ...
	} else if(readIn instanceof JoinLobby) { ...
		...
	} else if(...) { ... }
}
```

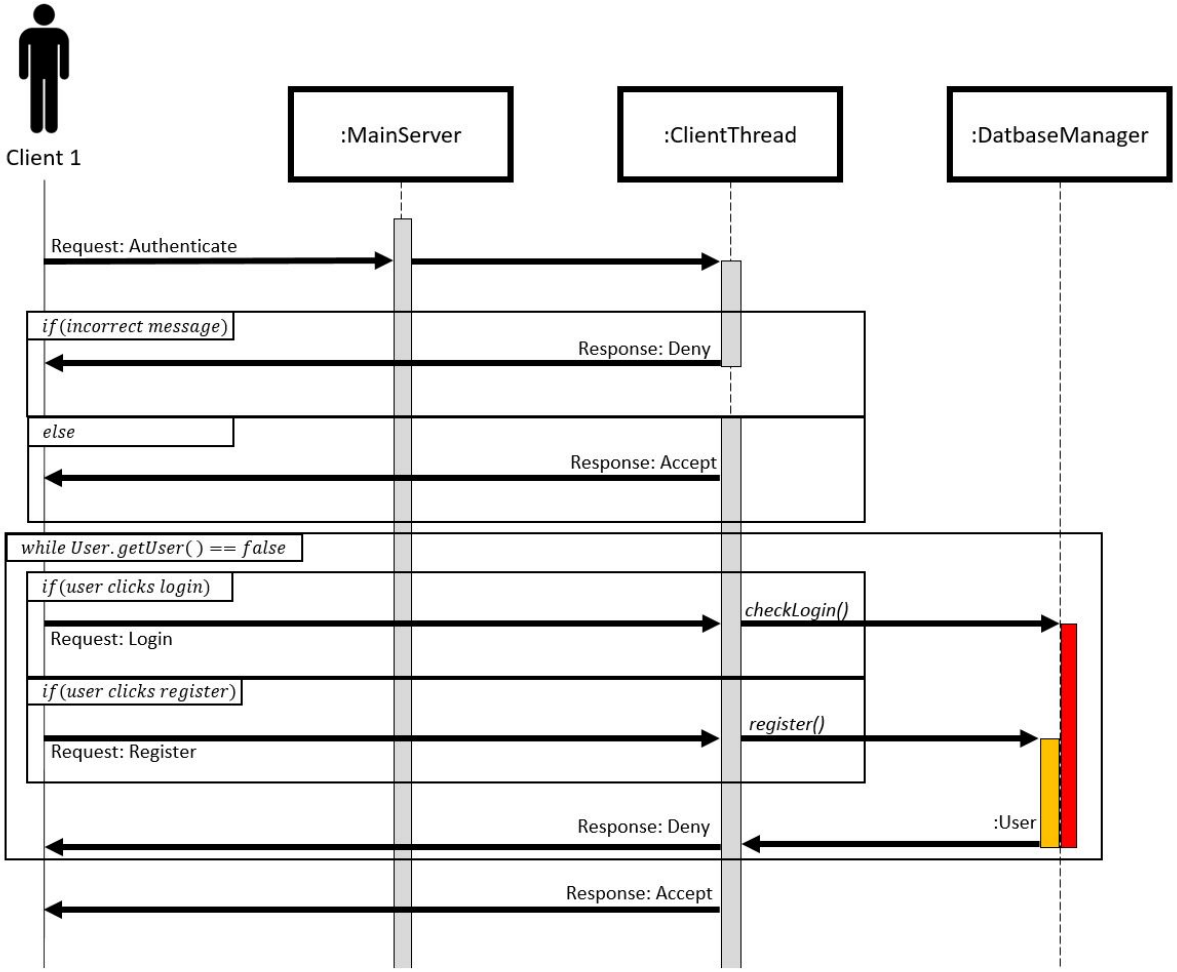
There are a total of thirteen different objects the server could receive from the client once login occurs. Each object received however falls under one of three protocols used between the client server. Table 1 below details each protocol and which objects they correspond to.

Table 1: Object transfer protocols used between server and client

Protocol Number	Corresponding objects	Description
Object Protocol 1	*DirectMessage*	This protocol is used for messages sent between clients. A message object is sent to the server and then redirected to the appropriate client(s). This protocol does not send a reply / recite to the issuing client thus minimising the network traffic. The decision was made not to send a recite for messaging due to the fact messages are not a critical function of the game application. The protocol sequence diagram can be seen below in Figure 3.
	LobbyMessage	
	GlobalMessage	
Object Protocol 2	*CreateLobby*	This protocol is the most widely used throughout the game. It is used for all objects extending request, and also do not have an effect other clients. The protocol sequence diagram can be seen below in Figure 4.
	NewGame	
	LoadGame	
	JoinLobby	
	Logout	
	SaveGame	
Object Protocol 3	*LeaveLobby*	This protocol is used when the object extends request and also has an effect of other clients. The protocol sequence diagram can be seen below in Figure 5.
	KickGuest	
	Action	
	StartGame	

5) Clean up – When a user logs out or closes their window, a clean up operation is called. This clean up operation is a instance method in the *ServerDataCollection* class called *removeUser()*.

This method removes any instance of the player on all clients GUI's as well as updating the server data GUI. This includes removing the players name, and lobbies and games they were part of and notifying any guests or hosts that the player may have been part of, that they have now left. A sequence diagram detailing when *removeUser()* is called can be seen in Figure 2.


Client 1
:MainServer
:ClientThread
:DatbaseManager
Request: Authenticate
if (incorrect message)
Response: Deny
else
Response: Accept
while User.getUser( ) == false
if (user clicks login)
checkLogin()
Request: Login
if (user clicks register)
register()
Request: Register
:User
Response: Deny
Response: Accept

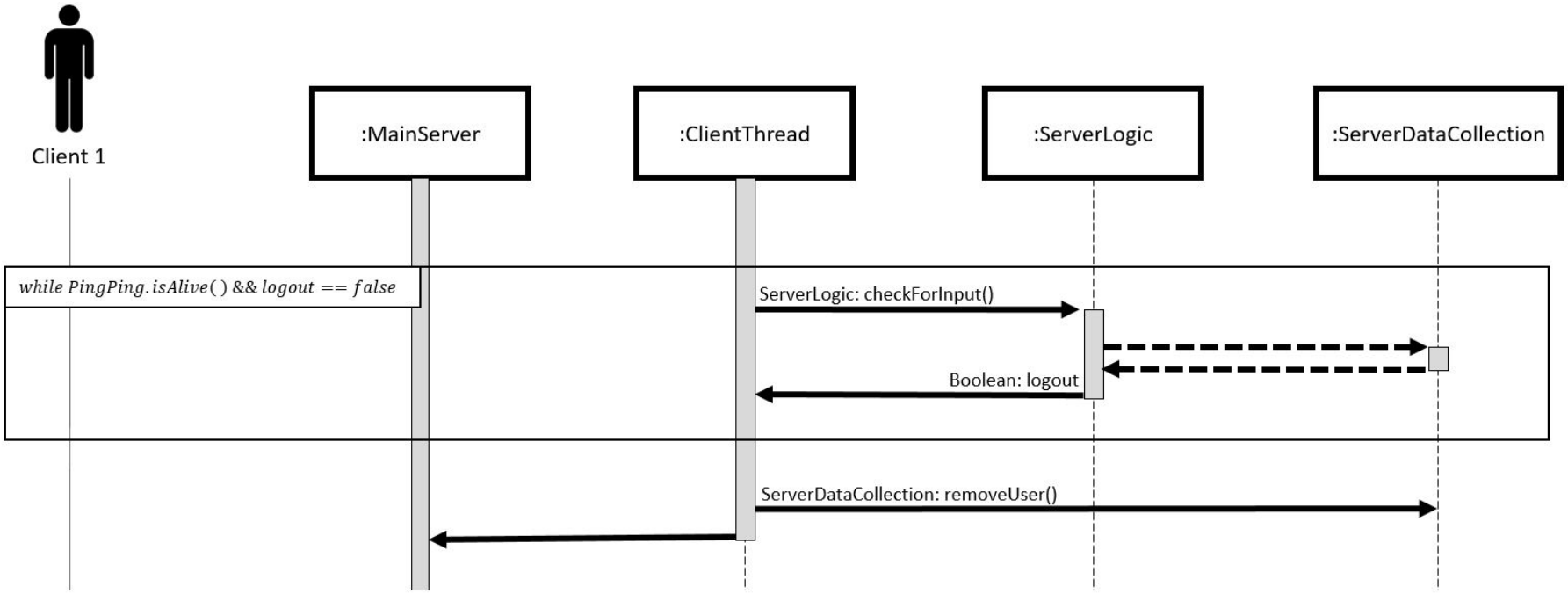


Figure 1: Sequence diagram of object passing and clean up. This occurs immediately after the sequence diagram in Figure 1.

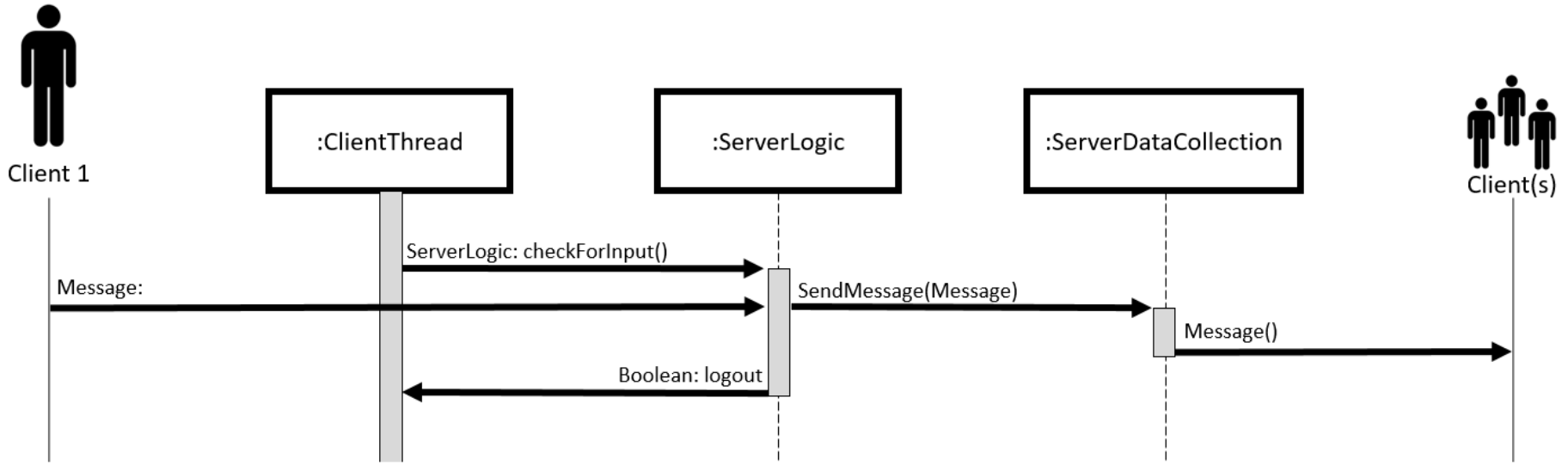


Figure 2: Protocol 1 - used for messaging between clients.

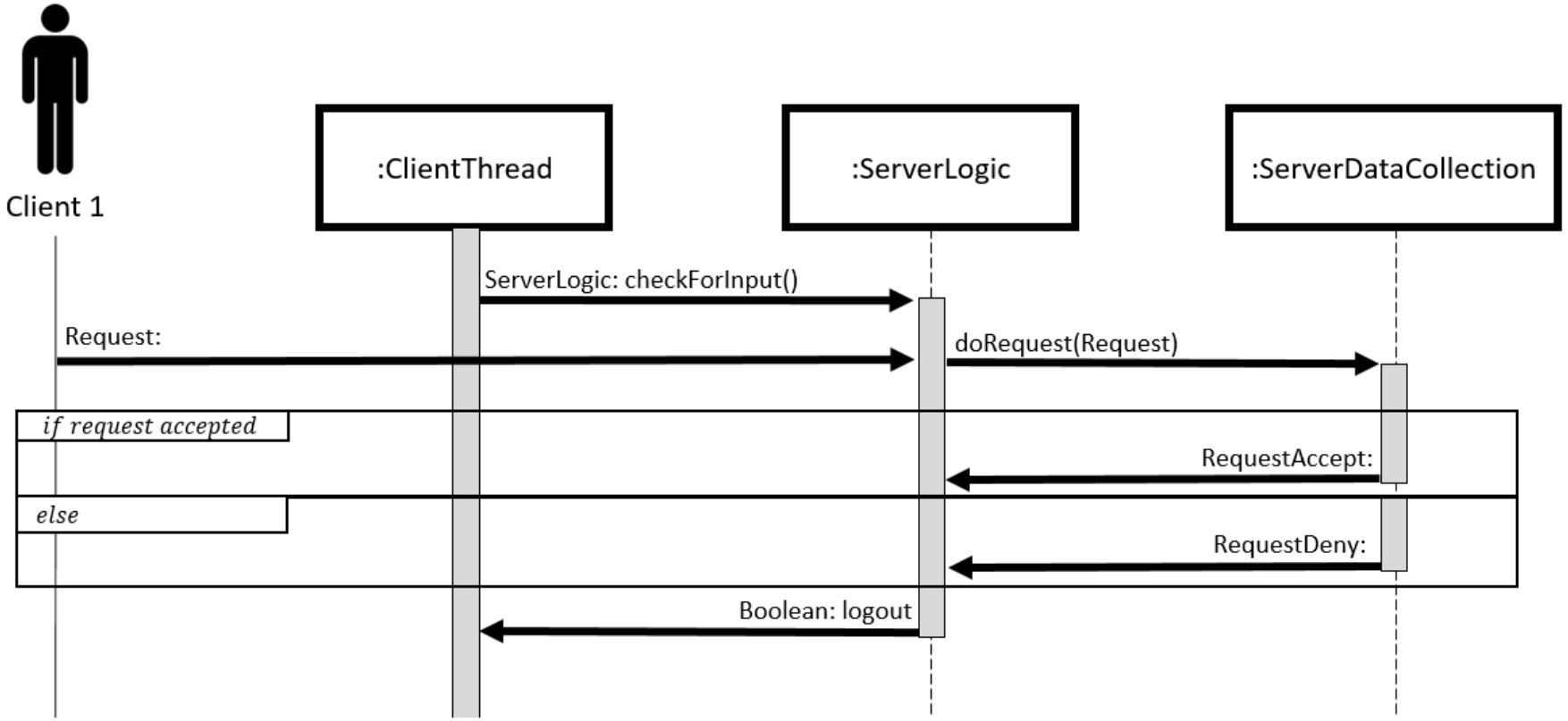


Figure 3: Protocol 2 - Used for requests that do not directly affect another client.

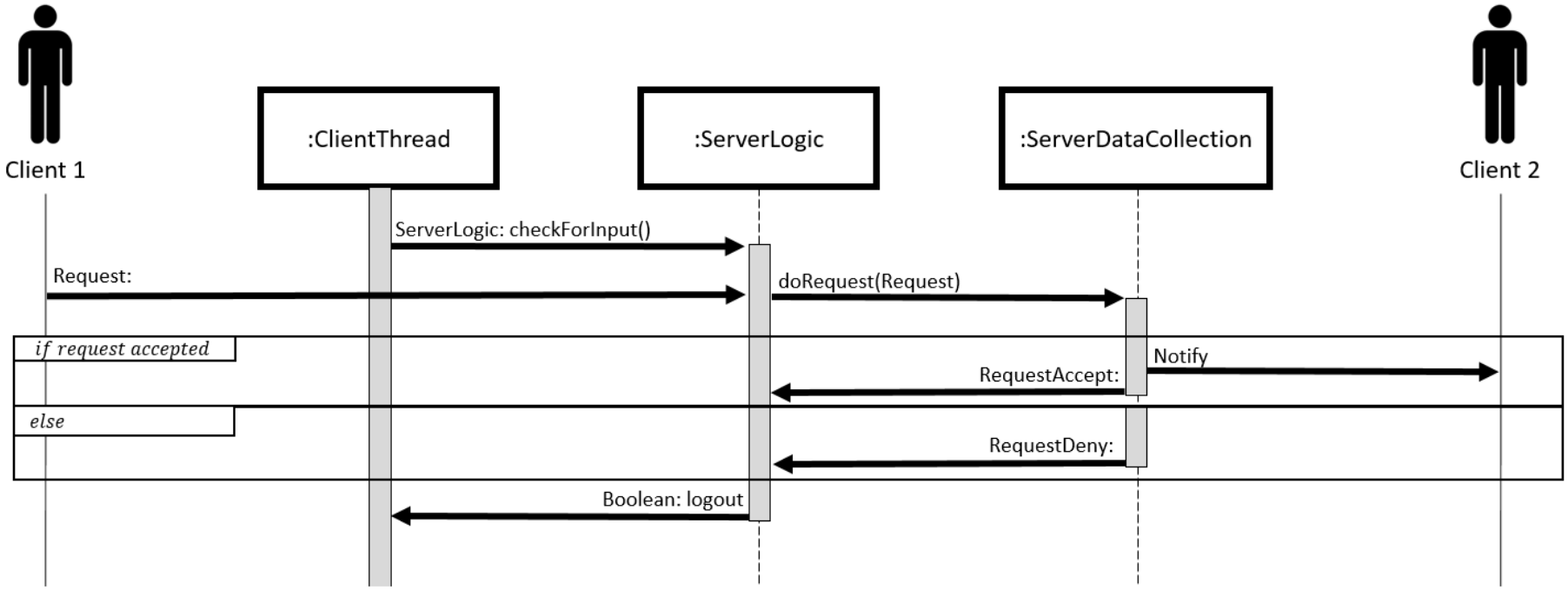


Figure 4: Protocol 3 - Used for requests that do affect other clients.

5 COMMUNICATION PROTOCOLS USING JAVA OBJECTS

Very early on in the project we decided to that the server and client would communicate between each other using object output/input streams. These are data streams that are provided by the Java API that allow for the transfer of "Serializable" objects across a network. Java will automatically deconstruct the object into a byte stream which it will then send to the target machine, the target machine will then automatically convert it back to the identical object on the other side. The Serializable interface is also provided by the Java API and is extremely easy to use. For a class to be serializable it must implement Serializable and declare a public static serial version unique identification code of type long. An objects of that class type can now be serialized and hence can be sent across a network.

This is an extremely powerful tool that the Java API provides, being able to send entire objects massively simplifies the communication between the server and client. It allows us to easily send a large variety of different requests/responses between the server and client without having to create complex protocols based on Strings. If we had used Strings we would have had to make bespoke methods for decoding the messages on either side of the client/server, this presents a significant problem, since there is a huge number of different protocol types that need to be sent between the server and client (see below). Different protocol types contain different bits/amounts of data, mostly Strings or primitives but some need to contain other objects encapsulated within them, using objects as protocols allows us to put any data we want as instance fields of that class, we can add/remove/modify these fields as and when we wish without having to rewrite much code, making the protocols extremely versatile.

There are some disadvantages to using object streams however. Sending objects over a network requires both machines to be running Java and that both have all the relevant up to date copies of the class files for every single class that can be sent. If the client or server has one of the classes missing ClassNotFoundExceptions will be thrown. If the class file is different on the server or client compared to the other we may get data inconsistantcies or InvalidClassExceptions, none of which we want. However these errors are easily avoidable through testing and checks to make sure everything is consistent between the server and client.

Using objects rather than Strings as protocols also allows us to determine the protocol type very easily, then once we have determined the protocol type we can cast the received object to the relevant type and then extract the required data from the object simply by calling the class' getter methods, hence there is no need to analyse long Strings. Java provides many useful tools that allow us to determine the type of an object, such as using the instanceof keyword, using someObject.getClass() == SomeClass.class, the isAssignableFrom(Class class) and isInstance(Object object) methods in the Class class which also allow for more dynamic assertion the object types.

5.1 PROTOCOL OBJECT TYPES:

There are four abstract protocol super types:

1. Request
2. Response
3. Message
4. Action

All objects sent between the client and server extend one of these classes. Request is sent by the client only, Response is sent by the server only and Message and Action can be sent by both. Responses are a little more complicated than the rest, since most responses can either be an accept or deny to whatever request the client sent. Hence there are multiple abstract subtypes to Response which themselves have subtypes that can contain different field variables depending of whether they were an accept of a deny. We could have used a boolean field variable to determine whether the Response was an accept of deny but we didn't want to have to send redundant data across the network if they weren't needed, for example, a AcceptLogin sends the User object that the user has requested to login as as a field variable,

but obviously if it is a DenyLogin the user isn't able to login as that user for whatever reason and hence the user object is not sent and there is no point in having the field variable there.

5.1.1 *Request subtypes:*

1. Authentification
2. ResendLast
3. Login
4. Logout
5. Register
6. ServerData
7. JoinLobby
8. CreateLobby
9. NewGame
10. LoadGame
11. LeaveLobby
12. KickGuest
13. StartGame
14. SaveGame
15. Resync

5.1.2 *Response subtypes:*

1. Accept
2. Deny
3. LoginResponse (abstract)
    a. AcceptLogin
    b. DenyLogin
4. RegisterResponse
5. ServerDataResponse
6. LogoutResponse
7. JoinLobbyResponse (abstract)
    a. JoinLobbyAccept
    b. JoinLobbyDeny
8. CreateLobbyResponse (abstract)
    a. CreateLobbyAccept
    b. CreateLobbyDeny
9. NewGameResponse
10. LoadGameResponse (abstract)
    a. LoadGameAccept
    b. LoadGameDeny
11. LeaveLobbyResponse
12. KickGuestResponse
13. StartGameResponse (abstract)
    a. StartGameAccept
    b. StartGameDeny
14. SaveGameResponse

5.1.3 *Action subtypes:*

1. Draft
2. Attack
3. SkipAttack
4. Fortify
5. EndTurn

5.1.4 *Message subtypes:*

1. GlobalMessage
2. LobbyMessage
3. DirectMessage

5.2 PROTOCOL OBJECT CLASS EXAMPLES:

These are some examples of classes that are used for communication between the server and client. This is just showing the classes used for logging into the server, these classes are used when the user clicks the login button on the client. Getter methods have been omitted.

```
public abstract class Request implements Serializable {
	private static final long serialVersionUID = 8779204053463182354L;
	private final String message;

	public Request(String message) {
		this.message = message;
	}
}

public class Login extends Request {
	private static final long serialVersionUID = -4488161542536300692L;
	private String userName;
	private String password;

	public Login(String message, String userName, String password) {
		super(message);
		this.userName = userName;
		this.password = password;
	}
}

public abstract class Response implements Serializable {
	private static final long serialVersionUID = -1840069425176786012L;
	private final String message;

	public Response(String message) {
		this.message = message;
	}
}

public abstract class LoginResponse extends Response {
	private static final long serialVersionUID = -251527400425334500L;

	public LoginResponse(String message) {
		super(message);
	}
}

public class AcceptLogin extends LoginResponse {
```

```
	private static final long serialVersionUID = -2785380680027777280L;
	private final User user;

	public AcceptLogin(String message, User user) {
		super(message);
		this.user = user;
	}
}

public class DenyLogin extends LoginResponse {
	private static final long serialVersionUID = 262789391782540923L;

	public DenyLogin(String message) {
		super(message);
	}
}
```

Both the Request and Response classes are abstract and both implement Serializable and hence all subclasses will also be serializable. You can see that all that above classes declare a auto-generated serialVersionUID which is required by the serializable interface. As mentioned above, the Request and Response classes are two of our super type protocols and their only field variables are a single String message, infact all four of the super type protocols contain this message field which means that all objects sent between the server client can contain a message, this field is only used for displaying information on the server/client console for means of debugging. Although the superclasses are identical (other than their names), we wanted to have separate classes for them so that their types were discernible.

The class Login extends Request and contains two additional instance field variables, one for the username and one for the password that the user of the client had entered before sending a login request. LoginResponse extends Response and is abstract, containing no addition variables, it simply exists to allow the client to determine what kind of response it is recieving. AcceptLogin and DenyLogin both extend LoginResponse, however AcceptLogin contains an additional instance field variable that contains a User object, this object contains all the data the user has stored on the database, the DenyLogin class in contrast contains no addition field variables, as explained before this is because the user was not able to login with the specified username/password combination for some reason and therefore no User object is required to be sent.

In contrast to the LoginResponse type the RegisterResponse type does not have any subclasses and is not abstract. This is because when the client receives a register response after having sent a register request it does not care if it was an accept or deny, since the client itself doesn't do anything after registering, whether it was successful or not, it just displays the text contained in the response's text field in the pane at the top of the client to inform the user the result of the register request.

6 CLIENT

6.1 THE CLIENT CLASS:

The client is what connects to the server over a network and allows user interaction with the server. It is a little confusing as the name of the main class is called Client, although this is just one of the many classes that make up the entire client. The main Client class encapsulates all other classes used by the client, including the Data class, the ClientGui class and the classes that deal with the game and the game

GUI and contains the main method for running the client. The methods in this class deal with sending and receiving Java objects to and from the server. This is the only class that can interact with the server, all other classes on the client communicate with the server through this class. Below are some of the field variables for the main Client class:

```
public class Client extends Thread {
        private static final String AUTHENTIFICATIONCODE =
                                        "IAMTEAMBOSTONRISKCLIENT";
        private final String server;
        private final int port;
        private final Socket socket;
        private final ObjectOutputStream objOut;
        private final ObjectInputStream objIn;
        private final Data data;
        private final ClientGui clientGui;
        private String waitingOn;
        public static final String LOGINRESPONSE = "LoginResponse";
        public static final String LOGOUTRESPONSE = "LogoutResponse";
        public static final String NOTHING = "Nothing";
        More public static final Strings here...
```

This is the method that sends requests to the server, this is the only method on any of the client side classes that can send requests. There are seperate methods for sending Messages and Actions, but those are far simpler methods.

```
public void sendRequest(Request request) {
        if (request instanceof ResendLast || waitingOn.equals(Client.NOTHING)) {
                switch (request.getClass().getSimpleName()) {
                        case "Login":
                                waitingOn = Client.LOGINRESPONSE;
                                break;
                        case "Logout":
                                waitingOn = Client.LOGOUTRESPONSE;
                                break;
                        case "Register":
                                waitingOn = Client.REGISTERRESPONSE;
                                break;
                        More case statements here...
                        default:
                                crash("Whilst attempting to send a request to the server
                                        the request type could not be determined"
                                        + "and therefore the waitingOn variable cannot
                                        be set which could lead to unexpected errors.");
                        }

                        objOut.writeObject(request);
        } else {
                data.setText("Cannot send another request at this time as the
                                client is currently waiting for a response of type: "
                                + waitingOn + ". Please wait a moment and try again.");
        }
}
```

When the client sends a request to the server it needs to listen for a response, but it also needs to block any further requests being sent to the server until it has received that response. When the client does receive a response from the server it must first check that the response is of the same type that it was waiting for.

The waitingOn field is critical to this function. It is a String that represents the simple name (i.e. the String that would be returned from someObject.getClass().getSimpleName()) of the Response type it is currently waiting for or simply "Nothing" if it is not expecting any responses. This ensures that the client will only execute the correct responses at the correct time, it also stops the user from sending a second request before having received a response for their first request. This line: if (request instanceof ResendLast || waitingOn.equals(Client.NOTHING)) ensures that requests can only be sent when the client is not waiting for a response, or if the request is a ResendLast (this is explained below).

Only the Login, Register, Logout and default cases are shown here, but all other cases are similar. The method automatically sets the waitingOn field when it sends requests by checking the simple name of the request against the case statements, or if it cannot determine the correct response type the method will call the client's crash(String stackTrace) please refer to our code for an explanation of that method. This ensures that the client can never send invalid requests. After successfully sending a request the waitingOn field will now be set correctly.

Here is the client's run() method which loops continuously whilst the client is running, to listen for incoming Responses, Actions and Messages. It determines which of these three possible super type protocols that it can receive from the server it is receiving and then sends it to the relevant method (try/catch exception handling statements have been omitted).

```
@Override
public void run() {
	while (true) {
		Object obj = null;
		obj = objIn.readObject();
		if (obj instanceof Response) {
			processResponse((Response)obj);
		} else if (obj instanceof Action) {
			processAction((Action)obj);
		} else if (obj instanceof Message) {
			processMessage((Message)obj);
		} else {
			sendRequest(new ResendLast("Recieved an object of type " +
				obj.getClass().getName() + " the only accepted types are,
				Response, Action and Message."));
		}
	}
}
```

This code blocks at the line obj = objIn.readObject(); until an object is received from the server. The variable obj is set to this received object and then a sequence of if statements are used to determine its type. The instanceof keyword will return true if the type of the object argument on the left is either the same as, or is a subclass of the class represented by the class name specified by the argument on the right. Hence in the first if statement, if the object is an instance of the Response class or any of its subclasses the statement will return true and the object will be cast to Response and sent as an argument to the processResponse(Response response) method which then does further processing to the object. Using this approach allows us to create a hierarchy of methods which the object is filtered down through, which, if it reaches the bottom of successfully, will execute the task that object represents.

Here is the client's processResponse(Response response) method, which is the next stage in the hierarchy of processing methods for incoming Responses (try/catch exception handling statements have been omitted):

```
public void processResponse(Response response) {
        if (response.getMessage() != null) {
                data.setText(response.getMessage());
        }
        if (!waitingOn.equals(Client.NOTHING) &&
                Class.forName("communicationObjects." + waitingOn)
                .isAssignableFrom(response.getClass())) {
                switch (waitingOn) {
                        case Client.LOGINRESPONSE:
                                processLogin((LoginResponse)response);
                                break;
                        case Client.LOGOUTRESPONSE:
                                logout();
                                break;
                        case Client.REGISTERRESPONSE:
                                waitingOn = Client.NOTHING;
                                Break;
                        More case statements here...
        } else if (response instanceof ServerDataResponse) {
                processServerData((ServerDataResponse)response);
        } else if (response instanceof KickGuestResponse) {
                kickedByHost();
        } else if (response instanceof StartGameResponse) {
                processStartGame((StartGameResponse)response);
        } else {
                sendRequest(new ResendLast("Got a response of type " +
                        response.getClass().getName() + " when was expecting a response of
                        type " + waitingOn + "."));
        }
}
```

This method processes responses if they are of the type the client is expecting or are of a special type. This line is particularly interesting;
Class.forName("communicationObjects."+waitingOn).isAssignableFrom(response.getClass()
)).
This statement returns a boolean, true if the class or interface represented by the Class object returned by the method Class.forName(String className) is either the same as, or is a superclass or superinterface of, the class or interface represented by response.getClass(). What this line does is dynamically determine whether the received response is a type that the client was expecting.

This ensures that the client will never execute invalid responses. For example, if I asked to create a new game whilst in the create lobby view and the server sends me a logout response by some error the client will not log itself out since that's not what it was expecting to happen.

The method then runs along a set of case statements, similar to the sendRequest method until it finds the correct one and executes the relevant code. Some cases cast the Response down and send it to another method, such as processLogin((LoginResponse)response), some call a method that does not require the Response object itself such as logout() and some simply return the waitingOn field to be "Nothing". The method always displays the message in the Response object to the panel at the top of the GUI if it is not null, some Responses are simply confirmations of reception of a request so the client just prints the message and returns the waitingOn to "Nothing" as with the RegisterResponse case.

You can see there are a few addition else if statements, these are needed because there are a few response types that can be received when they are not expected. These must therefore be handled specially, ServerDataResponse for example is sent by the server whenever data on the server changes such that it may be viewed on the client. But many clients can be logged in at the same time and all can be changing things on the server, other clients need to know what other clients are changing on the server but they don't know it's going to be changed, hence it can be recived at any time and needs to be handled differently to other Response types.

Here is the client's processLogin(Response response method), there are many methods that process different Response types, they can differ greatly in what they do.

```
public void processLogin(LoginResponse loginResponse) {
	if (loginResponse instanceof AcceptLogin) {
		data.setUser(((AcceptLogin)loginResponse).getUser());
		data.setDesiredViewState(ClientGui.MAINVIEW);
		clientGui.getLoginRegisterView().clearTextFields();
		waitingOn = Client.NOTHING;
		sendRequest(new ServerData(""));
	} else if (loginResponse instanceof DenyLogin) {
		waitingOn = Client.NOTHING;
	} else {
		sendRequest(new ResendLast("Recieved a " +
				loginResponse.getClass().getSimpleName() +
				" was expecting a AcceptLogin or DenyLogin."));
	}
}
```

This method is the final method in the hierarchy of methods for LoginResponse, it checks whether it is a AcceptLogin or a DenyLogin. If it is a AcceptLogin the method executes a few tasks, it sets the User object sorted in the Data class which contains information on the user so that it can be displayed on the client, changes the view state to the main view, requests new server data from the server and crucially sets the waitingOn field back to "Nothing" such that new requests may now be sent.

You can see that all these methods contain sendRequest(new ResendLast(... statements. A ResendLast is a special request type this was originally meant to be able to allow the server to dynamically determine exceptions over a network. The client sends them if it receives an unexpected or invalid response type. However it proved extremely difficult to practically impossible to implement so in the end the ResendLast object only contained a String message which was printed on the server whenever the server received a ResendLast. It proved extremely useful for debugging purposes.

6.2 THE DATA CLASS:

The Data class is used by the client to store all the data it needs to function. These are mostly things that are displayed on the GUI but also information used to run the game itself. It is an Observable class which the different GUI views observe, as explained below. Here is just the field variables of the class:

```
public class Data extends Observable {
        private String text;
        private String desiredViewState;
        private User user;
        private ArrayList<Lobby> games;
        private ArrayList<Lobby> lobbies;
        private ArrayList<User> players;
        private ArrayList<String> globalChat;
        private ArrayList<String> lobbyChat;
        private ArrayList<String> directChat;
        private Lobby myLobby;
        private Game game;
```

- String text: Is a single string of text that is displayed on the panel at the top of every client view to show information to the user, since as when you login successfully is displays "Password accepted. Login successful."
- String desiredViewState: This is a single string that is set to one of five public static final Strings that are declared in the ClientGui class. Changing this field to one of those will change the ClientGui's view state.
- User user: The user object for the logged in user on this client, null if not logged in.
- ArrayList games, lobbies and players: The Lobby and User objects that the server sends to the client such that they may be displayed on the GUI.
- ArrayList<String> globalChat, lobbyChat, directChat: The lists of chat messages that are displayed on the GUI, they are kept at maximum size of 10 messages any messages added after there are already 10 messages in the list cause the oldest message to be removed from the list.
- Lobby lobby: The lobby the user is in, null if they are not in one.
- Game game: The game the user is playing, null if they are not playing a game.

The class contains many methods that allow for manipulation of these fields. Such as the setServerData(Client client, ServerDataResponse serverData) method which processes data from the server such as the lists of game, lobbies and players and automatically updates them, it also removes "ghost lobbies" which is a situation where the client thinks the user is in a lobby that no longer exists on the server for some reason.

All the methods that change data on the class call the notifyGUIs() method in the class which simply calls setChanged() and notifyAll() such that all observers of the class have their update methods called.

6.3 THE CLIENTGUI CLASS AND VIEW CLASSES:

The client GUI displays information to the user and allows them to interact with it. The main ClientGui class governs all the other GUIs. ClientGui extends JFrame and has four possible view classes, the LoginRegisterView, MainView, CreateLobbyView and the LobbyCiew all of which extend JPanel should have they can be printed onto the ClientGui's frame. The game has its own GUI which the ClientGui can open but it is a separate frame and is not displayed upon the ClientGui's frame itself. All

the views and the ClientGui itself observe the Data class. The ClientGui and all four of its views have been made manually without the aid of GUI builder software.

This is part of the code for the ClientGui class, some of its field variables are shown including the public static final Strings that govern the view states, as I have mentioned. The update method is also included which shows how the ClientGui controls the views.

```
public class ClientGui extends JFrame implements Observer {
	private String currentViewState;
	public static final String LOGINREGISTERVIEW = "LoginRegisterView";
	public static final String MAINVIEW = "MainView";
	public static final String CREATELOBBYVIEW = "CreateLobbyView";
	public static final String LOBBYVIEW = "LobbyView";
	public static final String RISKGUI = "RiskGUI";
	private static final int WIDTH = 600;
	private static final int HEIGHT = 630;
	Other field variables here...

public ClientGui(Client client) {
	this.setSize(WIDTH, HEIGHT);
	this.setTitle("Risk client");
	this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	this.setResizable(false);
	Omitted code here...
}
```

This is the update method for the ClientGui class. It functions based on the currentViewState in this class and the desiredViewState in the Data clas. It checks to see if the desired view state is the same as the current view state and if it is not then it changes the view state accordingly. It also updates the observers of the Data class, added the new view to the list and removing all others, other than itself.

```
@Override
public void update(Observable o, Object arg) {
	if (data.getDesiredViewState().equals(ClientGui.LOGINREGISTERVIEW)) {
		if (!currentViewState.equals(ClientGui.LOGINREGISTERVIEW)) {
			getContentPane().removeAll();
			getContentPane().add(loginRegisterView);
			repaint();
			printAll(getGraphics());
			setVisible(true);
			currentViewState = ClientGui.LOGINREGISTERVIEW;
			data.deleteObservers();
			data.addObserver(this);
			data.addObserver(loginRegisterView);
			data.notifyGUIs();
		}
	} else if (data.getDesiredViewState().equals(ClientGui.MAINVIEW)) {
		if (!currentViewState.equals(ClientGui.MAINVIEW)) {
			Omitted code here...
		}
	} else if (data.getDesiredViewState().equals(ClientGui.CREATELOBBYVIEW)) {
```

```
                if (!currentViewState.equals(ClientGui.CREATELOBBYVIEW)) {
                        Omitted code here...
                }
        } else if (data.getDesiredViewState().equals(ClientGui.LOBBYVIEW)) {
                if (!currentViewState.equals(ClientGui.LOBBYVIEW)) {
                        Omitted code here...
                }
        } else if (data.getDesiredViewState().equals(ClientGui.RISKGUI)) {
                if (!currentViewState.equals(ClientGui.RISKGUI)) {
                        if (data.getGame() != null && gameGui != null) {
                                setVisible(false);
                                gameGui.setVisible(true);
                                currentViewState = ClientGui.RISKGUI;
                        } else {
                                client.crash("The client GUI attempted to start the game GUI
                                        when the game was null or the game GUI was null.");
                        }
                        data.deleteObservers();
                        data.addObserver(this);
                        data.notifyGUIs();
                }
        } else {
                client.crash("The client cannot determine the desired view state.");
        }
}
}
```

All four of the client GUI view classes also have an update method that contains different code. Each view references different parts of the Data object on the client to display what is relevant to that particular view. Below is the update method for the main view of the client GUI. The variables userNameField, gamesPlayedField, gamesWonField, gamesLostField and ratingField refer to JTextFields. And the variables playersListPane, lobbiesListPane, gamesListPane, globalMessagingPane, lobbyMessagingPane, directMessagingPane and displayPane refer to JTextPanes.

```
@Override
public void update(Observable o, Object arg) {
        displayPane.setText(data.getText());

        userNameField.setText("Username: " + data.getUser().getUsername());
        gamesPlayedField.setText("Played: " +
data.getUser().getNumberOfGamesPlayed());
        gamesWonField.setText("Won: " + data.getUser().getNumberOfWins());
        gamesLostField.setText("Lost: " + data.getUser().getNumberOfLosses());
        ratingField.setText("Rating: " + data.getUser().getRating());

        playersListPane.setListData(data.getPlayers().toArray(
                                        new User[data.getPlayers().size()]));
        lobbiesListPane.setListData(data.getLobbies().toArray(
                                        new Lobby[data.getLobbies().size()]));
        gamesListPane.setListData(data.getGames().toArray(
```

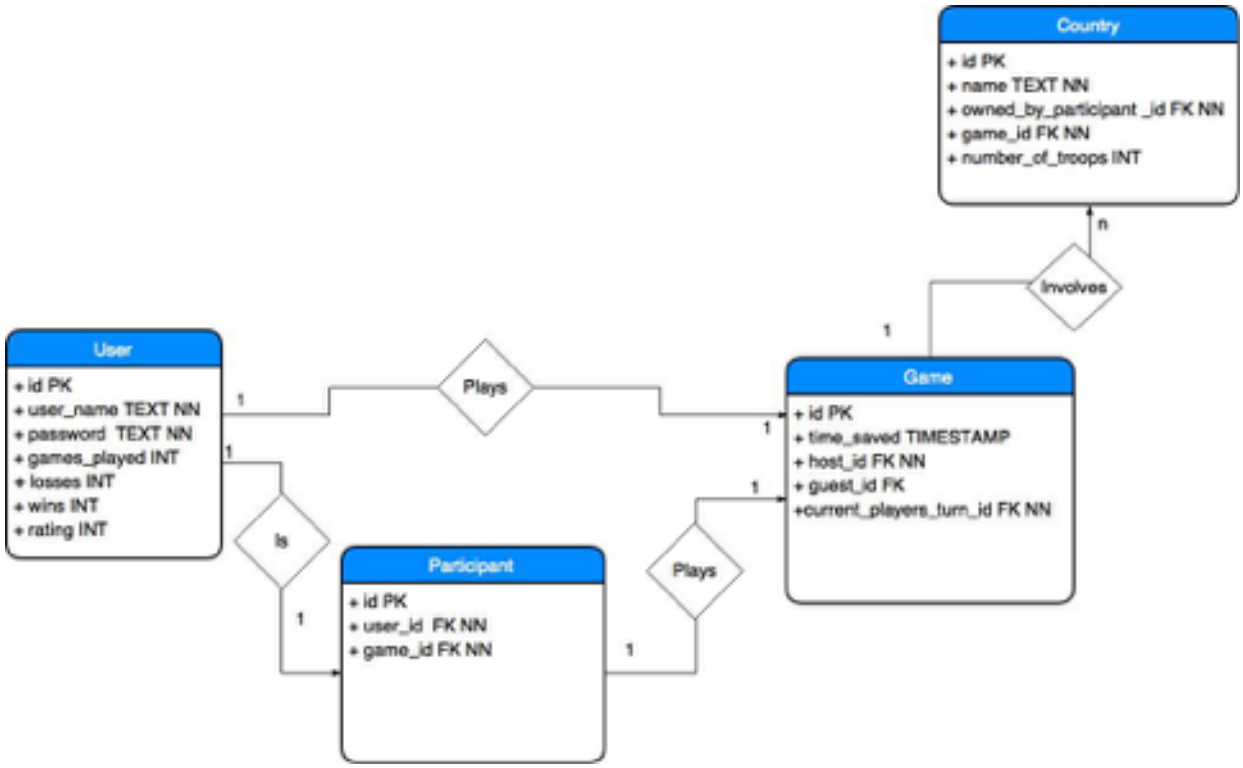
```
                                        new Lobby[data.getGames().size()]));

        globalMessagingPane.setListData(data.getGlobalChat().toArray(
                                        new String[data.getGlobalChat().size()]));
        lobbyMessagingPane.setListData(data.getLobbyChat().toArray(
                                        new String[data.getLobbyChat().size()]));
        directMessagingPane.setListData(data.getDirectChat().toArray(
                                        new String[data.getDirectChat().size()]));
        repaint();
}
```

These update methods simply update the correct field with the correct data that they need to display to the user, obviously different views need to display different things but it is all the same concept.

7 DATABASE

7.1 ENTITY-RELATIONSHIP DIAGRAM



The above image represents the structure of our database in the form of an Entity-relationship diagram. This diagram depicts the entities in our relational database and the relationships between them. The notation we have used for the diagram follows Peter Chen's model.

The four entities that we have identified are **User**, **Participant**, **Game** and **Country**. We consider a user to be a participant, a user and a participant to play one game and a game to involve many countries.

Each entity in this diagram represents a table in our database. Below I will discuss how the tables are formed:

7.2 USERS TABLE

The primary key in the user table is the user id attribute and this key will be used to uniquely identify any user added to the database. This primary key is automatically generated when a user is inserted into the database through declaring it type **serial.** When a user first registers, their username and password will be stored in the database and their other attributes such as ***games_played, losses***, ***wins*** and ***rating*** will be set to zero. The rating is calculated by the below method in the User class:

```
public void setRating(int numberOfWins, int numberOfGamesPlayed) {
        if(numberOfGamesPlayed == 0){
                this.rating = 0;
        }
        else {
                this.rating = (int) (numberOfWins*5/numberOfGamesPlayed);
        } }
```

7.3 PARTICIPANTS TABLE

A user becomes a **participant** when they join or start a game. The primary key for this table is the participant id and the two other attributes ***user_id*** and ***game_id*** are foreign keys referencing the User and Game tables. These attributed are constrained as being not null due to a participant not existing without a user_id and a game.

Currently our system only allows a user to play and save one game as a host but with a participants table incorporated in the database there is clear scalability available for our system to enable users to save more than one game. A guest may be part of more than one game and this is illustrated in the participants table.

Once a game is saved, the participants details our inserted into the database using the below SQL insert update:

```
String insertParticipant = "INSERT INTO PARTICIPANTS (user_id, game_id) " +
                                "VALUES ((SELECT id FROM USERS WHERE username = ?),
                                (SELECT id FROM GAMES WHERE name = ?));";
```

This SQL update is executed for both the host and the guest of the game using a prepared statement.

The parameterised values are set to the username of the guest/host and the game name.

7.4 GAMES TABLE

A game is uniquely identified by its primary key **id.** This is automatically generated when a game is saved onto the database. The name of the game is chosen by the host user when they start a new game and this is saved onto the database in the second column of the game table. The ***host_id, guest-id*** are foreign keys which reference the users table. They denote the host and guest of the game respectively. The ***host_id*** is constrained as being not null as there will never be a game without a host. The host will always remain as the same user but a new guest may join a host's loaded game to replace the previous guest.

For a game to be loaded from the database, the ***user_id*** of the current players turn is stored once a game is saved. This will be updated every time the game is saved via the execution of the SQL update shown below:

```
String updateGame = "UPDATE GAMES SET current_players_turn_id = ?, time_saved = CURRENT_TIMESTAMP WHERE id =?;";
```

The time that the game is saved is also stored on the database and updated accordingly. A game can only be saved at the start of either players turn. This is to ensure the operations on the database are atomic.

7.5 COUNTRIES TABLE

The countries table is used to store the countries owned by both players in a game. Similarly, to the other tables its primary key is an ***id*** number which is auto-incremented every time an insertion is made to the table. Columns ***owned_by_participant*** and ***game_id*** are foreign keys referencing the participants table and games table respectively. The last column represents the ***number of troops*** on a country.

When a game is saved the columns of this table are populated and when a game is loaded by a user the data in this table is used to import the loaded game information.

The below select query returns all the countries for a saved game and this is executed in the load game method in the Database Manager class:

```
String selectCountries = "SELECT * FROM COUNTRIES WHERE  game_id = ? ;";
```

7.6 DATABASE EXPANSION

There's scope to add extra tables to our database given the time to improve our system. These tables would include a **results** table. Once a game had been finished it would be updated with a ***result_id*** and a ***type.*** As far as our game is concerned the result would either be the host or the guest winning and a win would increase the users rating.

Another possible consideration would be to include a table which was updated after every move of the game. Now this would be computationally expensive as every move would need to be stored in memory. However, it would provide the database with more up to date information with regards to the current state of the game. If a table containing moves was to exist another one would be needed to uniquely identify each troop and again this would be very expensive in terms of storage.

7.7 DATABASE INTERACTION WITH THE SERVER

There are four main stages at which the database interacts with the server:

1. **Login –** After the server has received a request from the client to login the server calls the method *checkLogin(String username, String password)* located in the Database Manager class. This method checks whether the username and password entered match the details on the database and then returns a User object. This object contains a message parameter which indicates to the server whether the login has been successful.
2. **Registration –** When a user registers the server calls the method *register(String username, String password)* located in the Database Manager class. This method checks that both the username and password entered are in alphanumeric format and that there are no other matches for the username inputted contained on the database. If both these conditions hold then the username and password are inserted into the Users table. A user object is sent back to the server with the appropriate message detailing whether the registration has been accepted.
3. **Save Game –** When the client sends a save game request to the server the server calls the *saveGameDatabse(ArrayList<String>)* method which is again located in the Database Manager class. The Array List of Strings has been converted using the *saveGame()* method in the client

class from a game object to a list containing all of the games details including the game name, hosts name, guests name, countries owned, troops located in a country and whose turn it is. This method then checks whether a previous game has been inserted under this name and either inserts a new entry into the database or updates the previous entry.

4. **Load Game –** On receiving a load game request from the client the server calls the *loadGameFromDatabase(String hostName)* method in the DatabaseManager class. This method checks whether a game has been saved in the database under the hosts username and if so returns this game in the form of an *ArrayList<String>* to the server. This array list is then converted back to a game object in the client class. If there's no game attributed to this host, then null is returned.

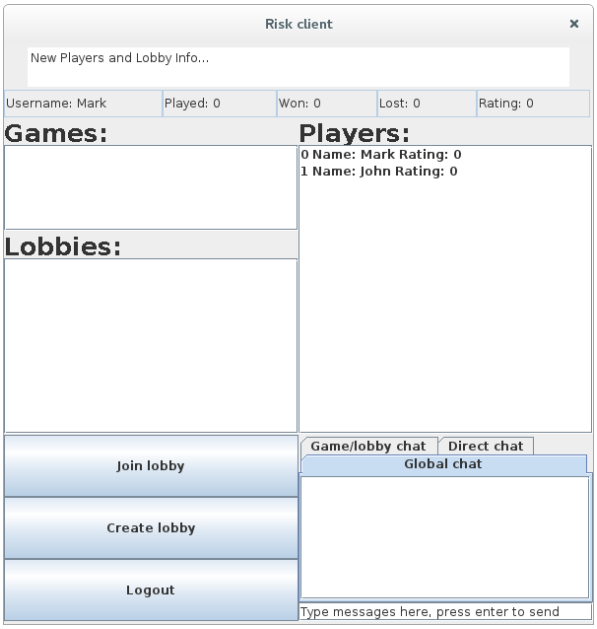
8 TEST PLAN

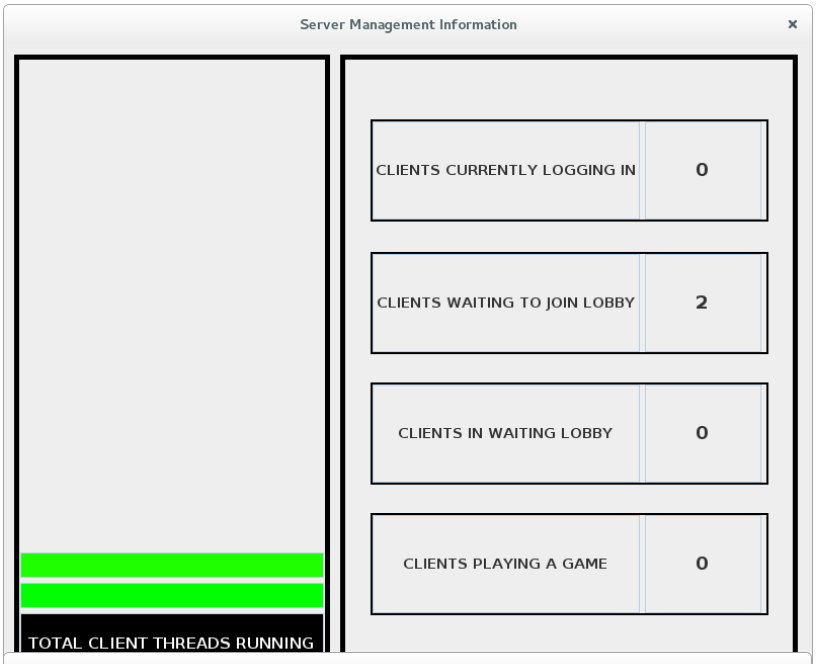
8.1 CLIENT SERVER TESTING

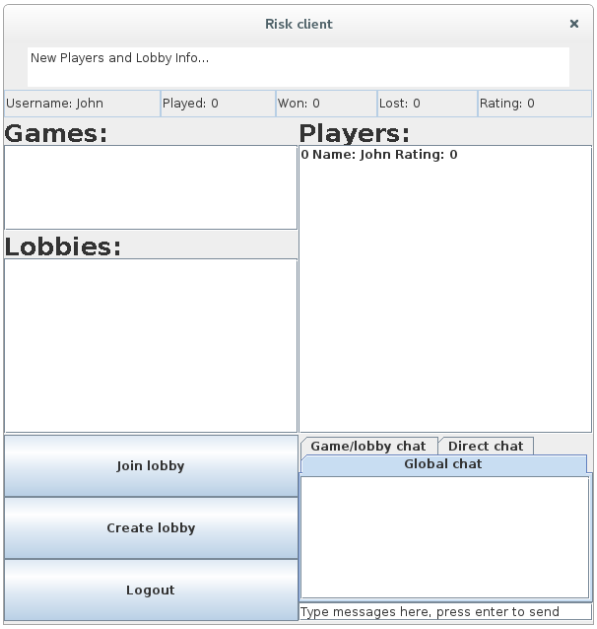
The server – client were tested using various scenarios. The effect was documented and screen shots were taken. The scenario tests are detailed below;

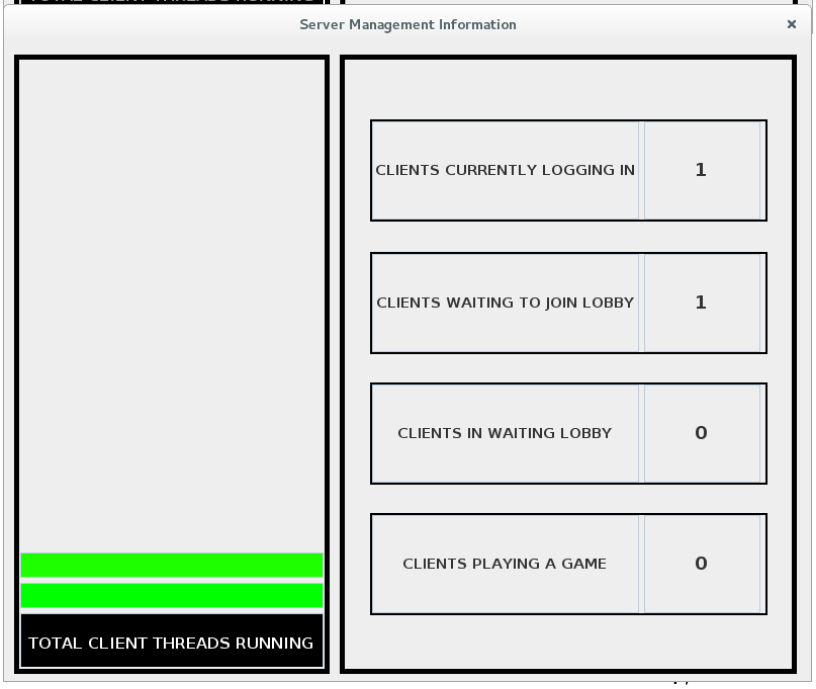
8.1.1 *Test 1*

Scenario: Logging in two clients and one logging out.
Testing: Testing to check the remaining client has it's view updated and that the server data GUI also updates with the changes.
Outcome: Successful







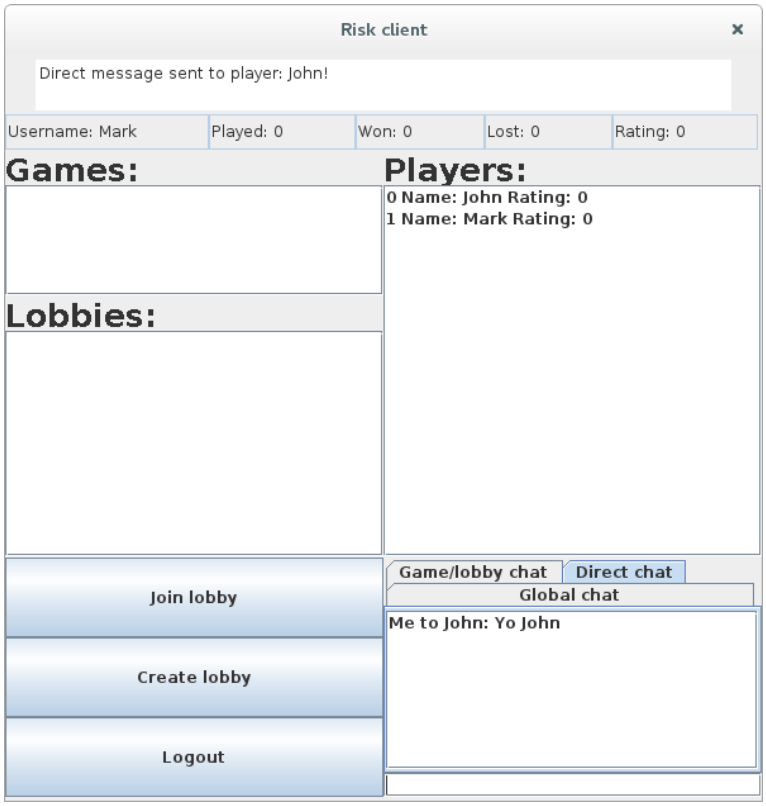


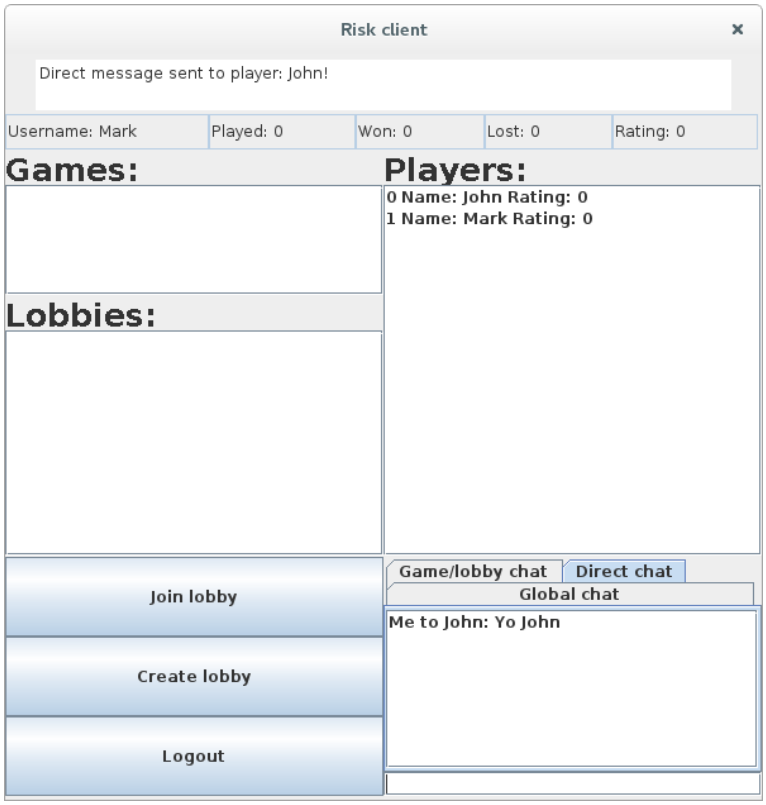
8.1.2 ***Test 2***

Scenario: Sending an individual a private message

Testing: Checking both individuals receive and display the private message

Outcome: Successful



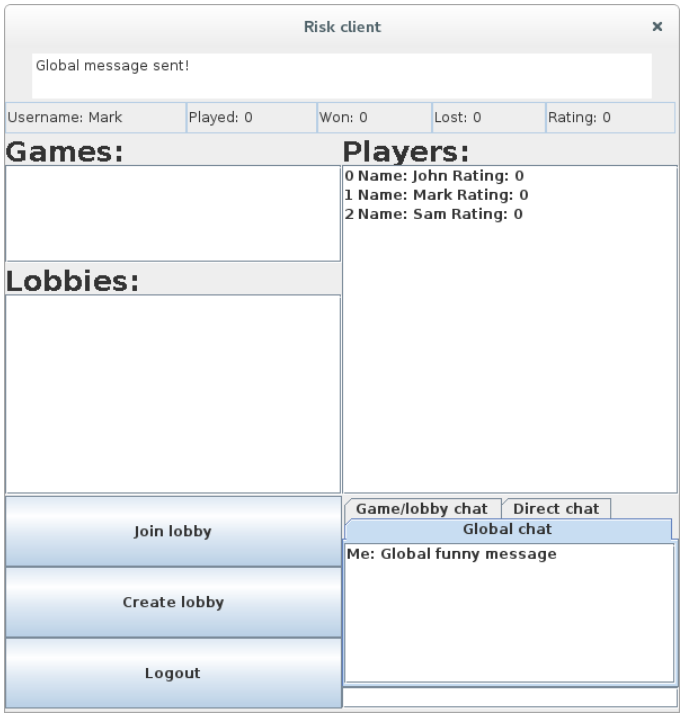


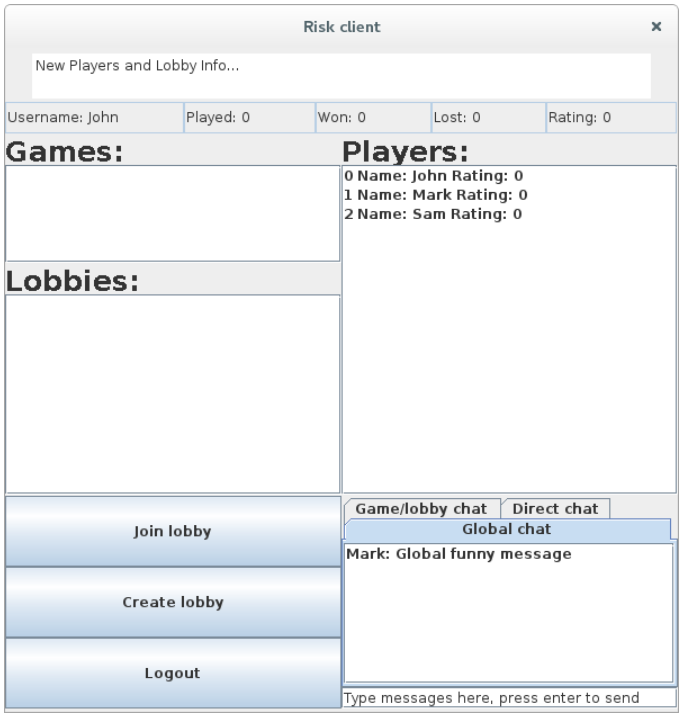
8.1.3 ***Test 3***

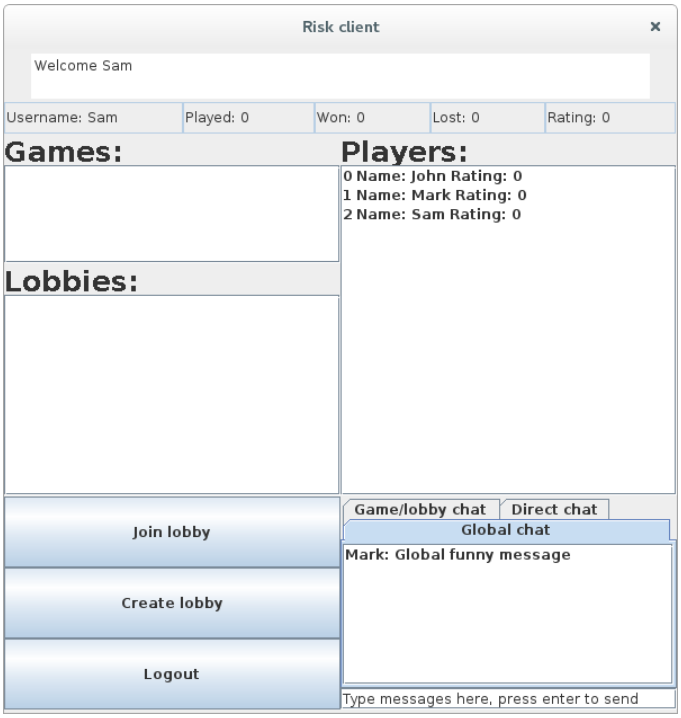
Scenario: Sending a global message

Testing: Checking all clients receive and display the message

Outcome: Successful





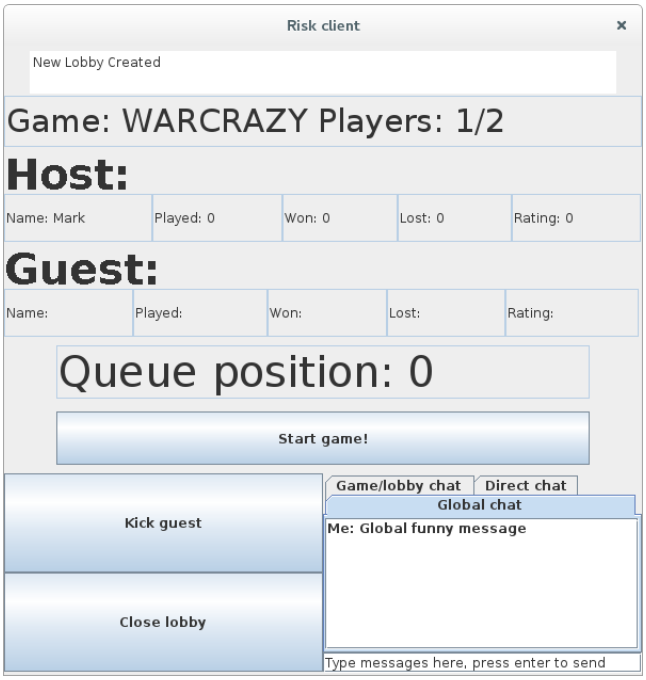


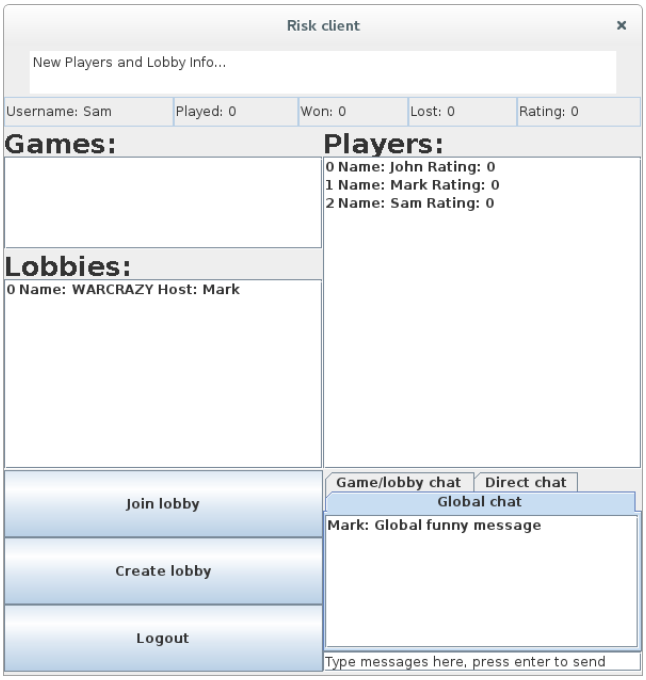
8.1.4 ***Test 4***

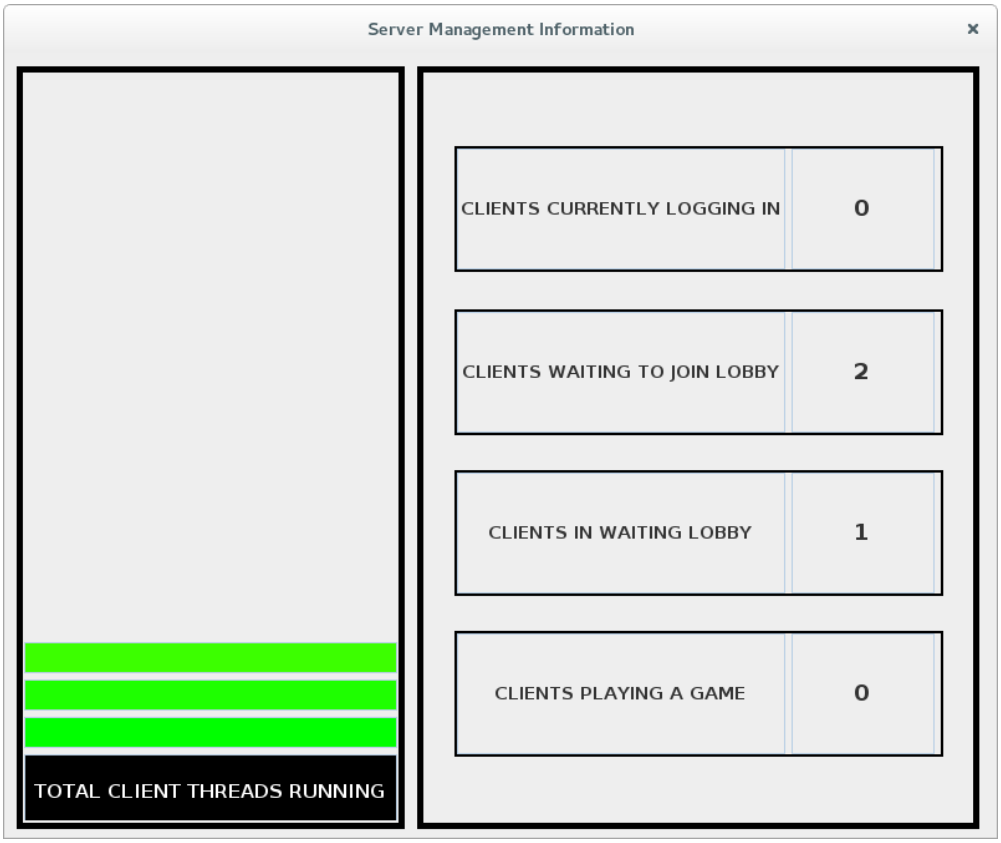
Scenario: Create a lobby

Testing: Checking the lobby is successfully created and that all clients views are updated.

Outcome: Successful





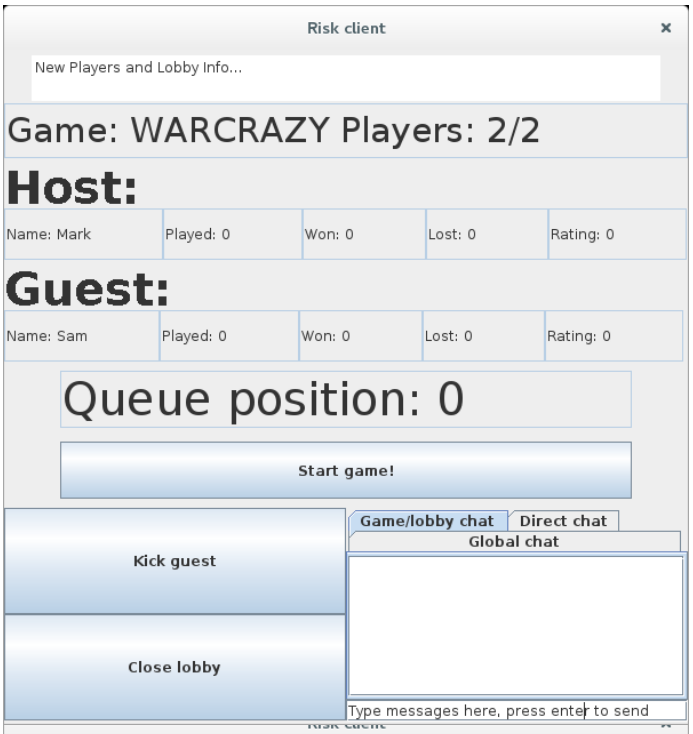


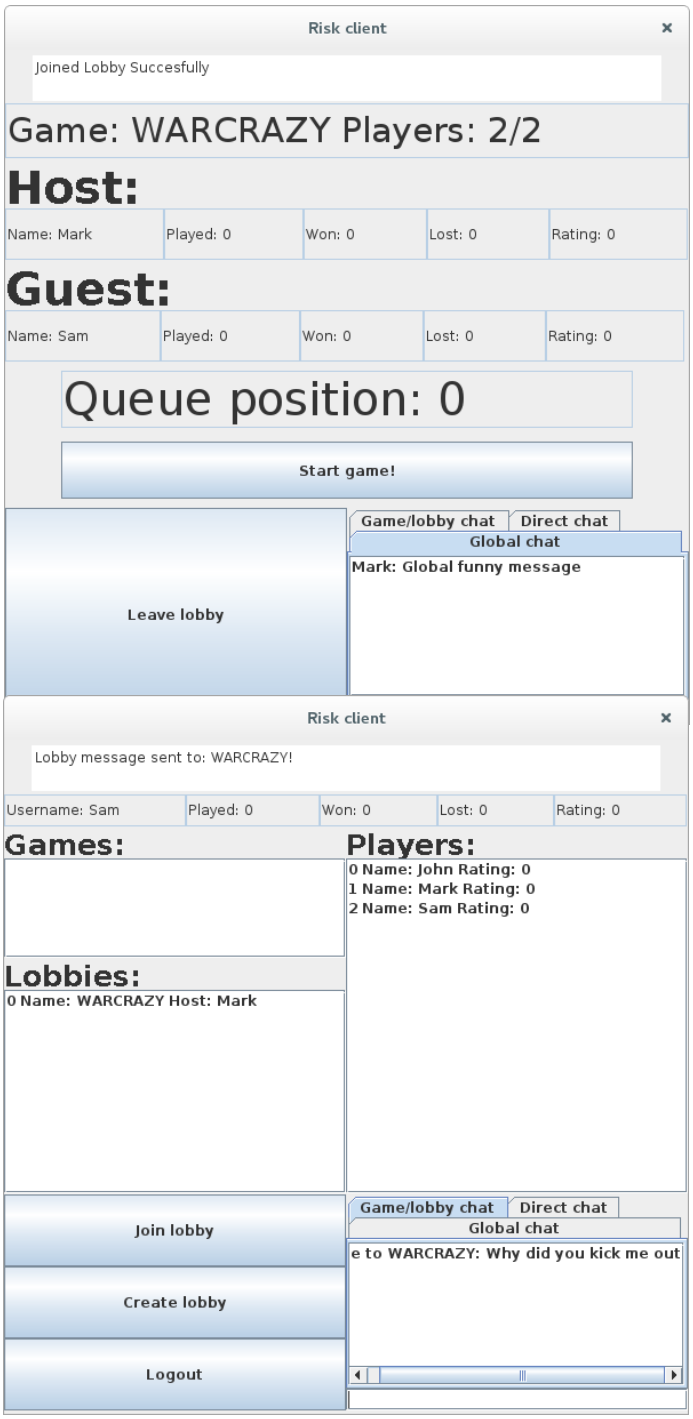
8.1.5 ***Test 5 & 6***

Scenario: Kicking a guest out of the lobby & messaging a lobby

Testing: Checking the clients view updates correctly.

Outcome: Successful



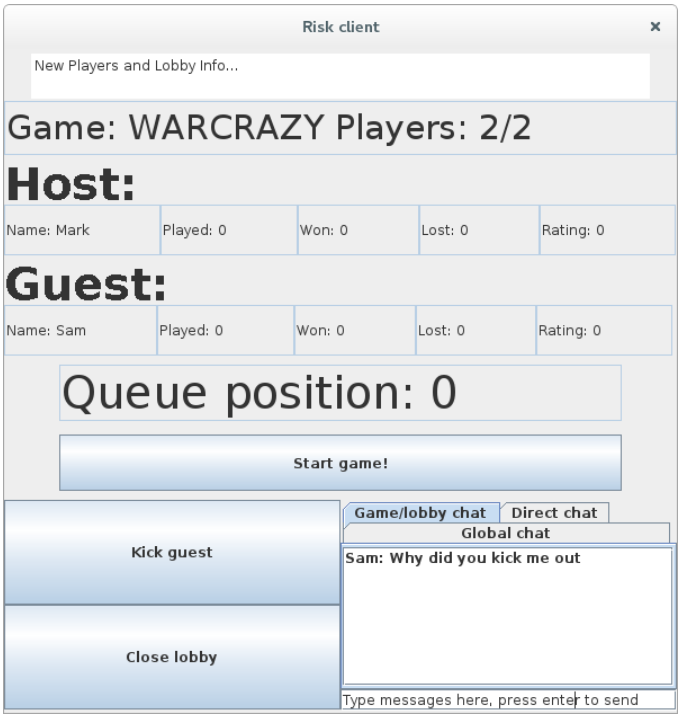


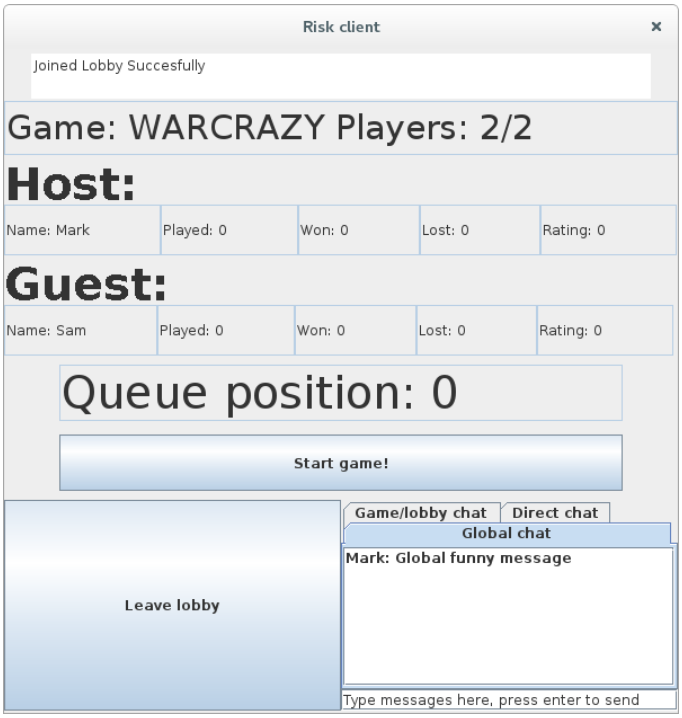
8.1.6 ***Test 7***

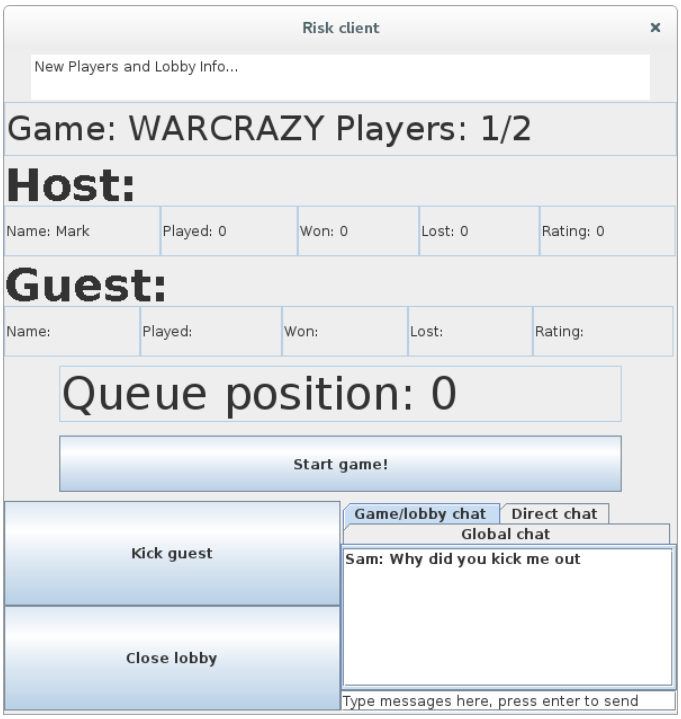
Scenario: Leave a lobby

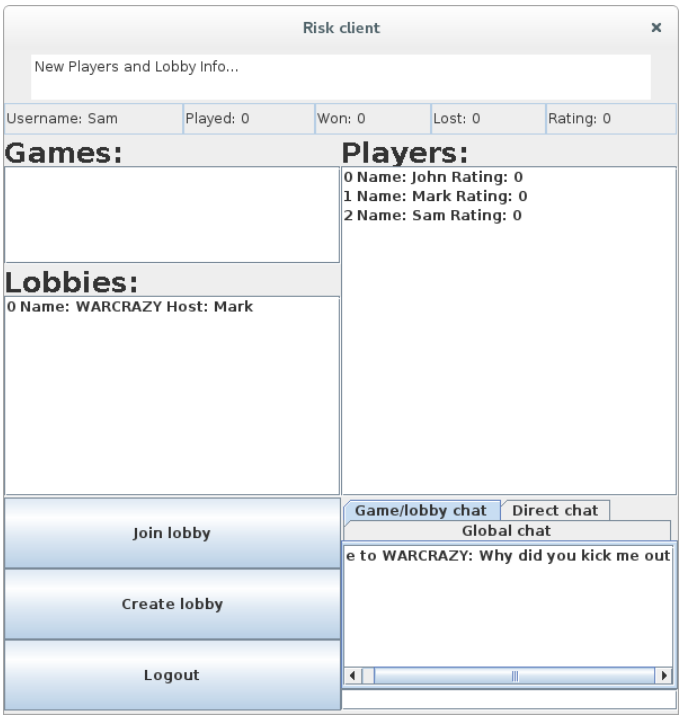
Testing: Checking the clients view updates correctly.

Outcome: Successful







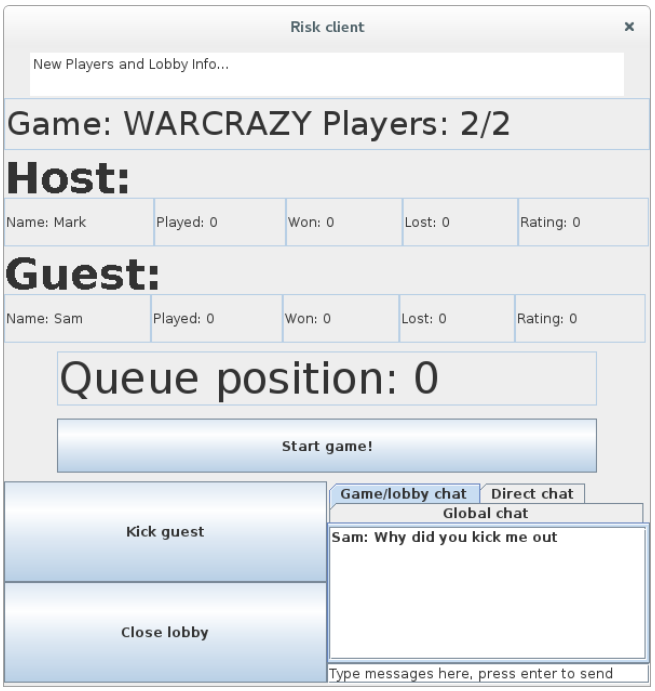


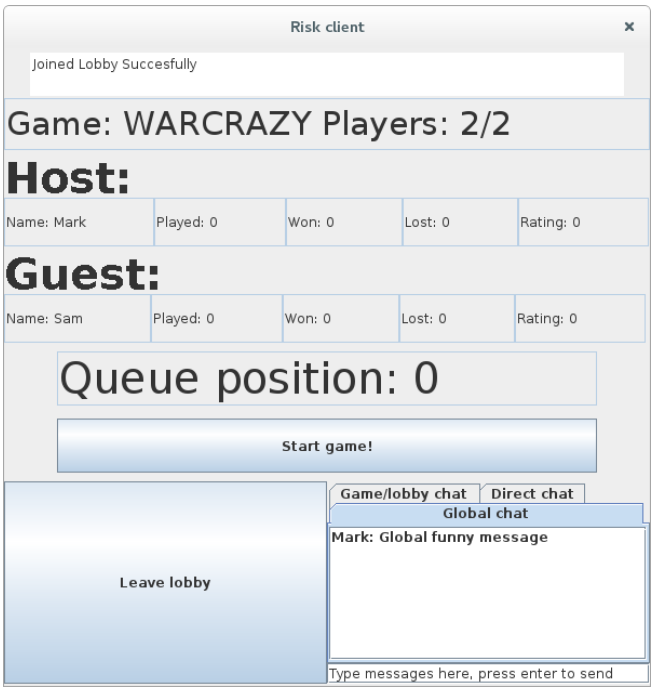
8.1.7 ***Test 8***

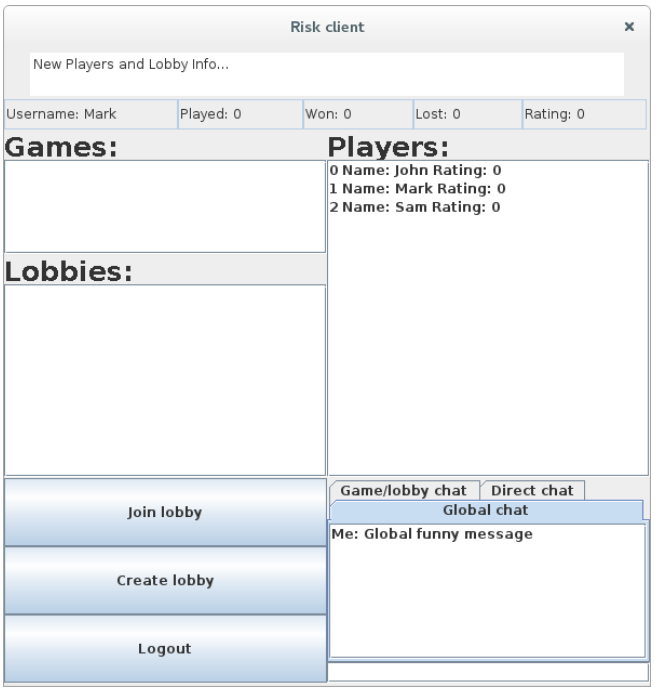
Scenario: Close a lobby

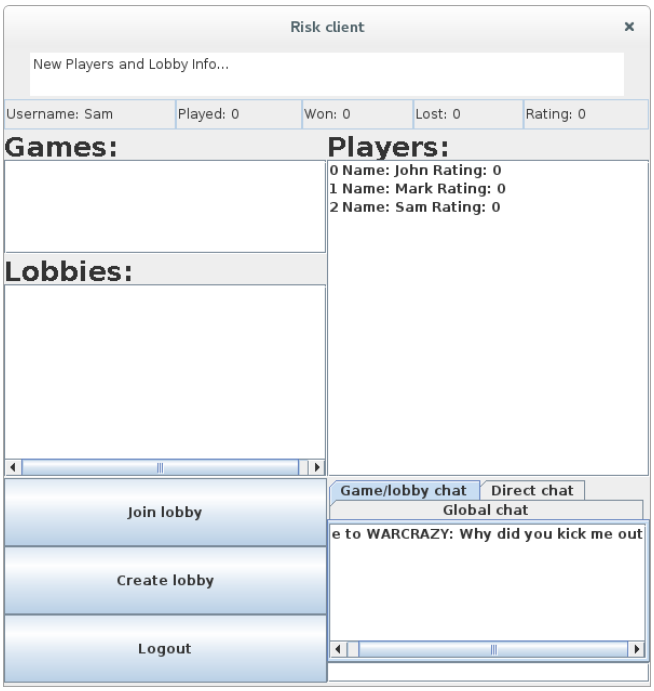
Testing: Checking the clients view updates correctly.

Outcome: Successful







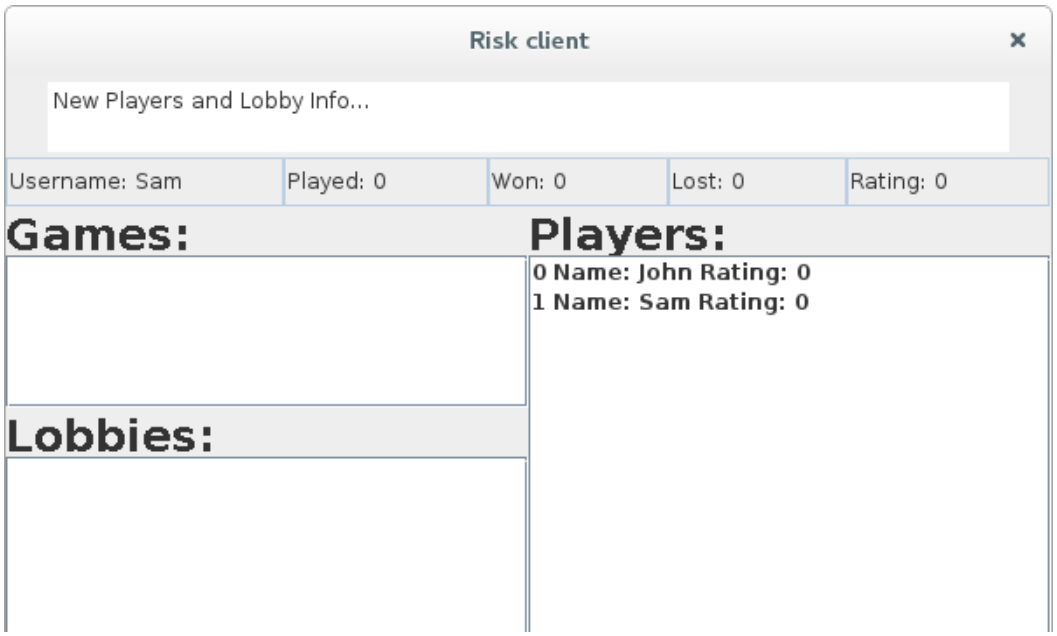


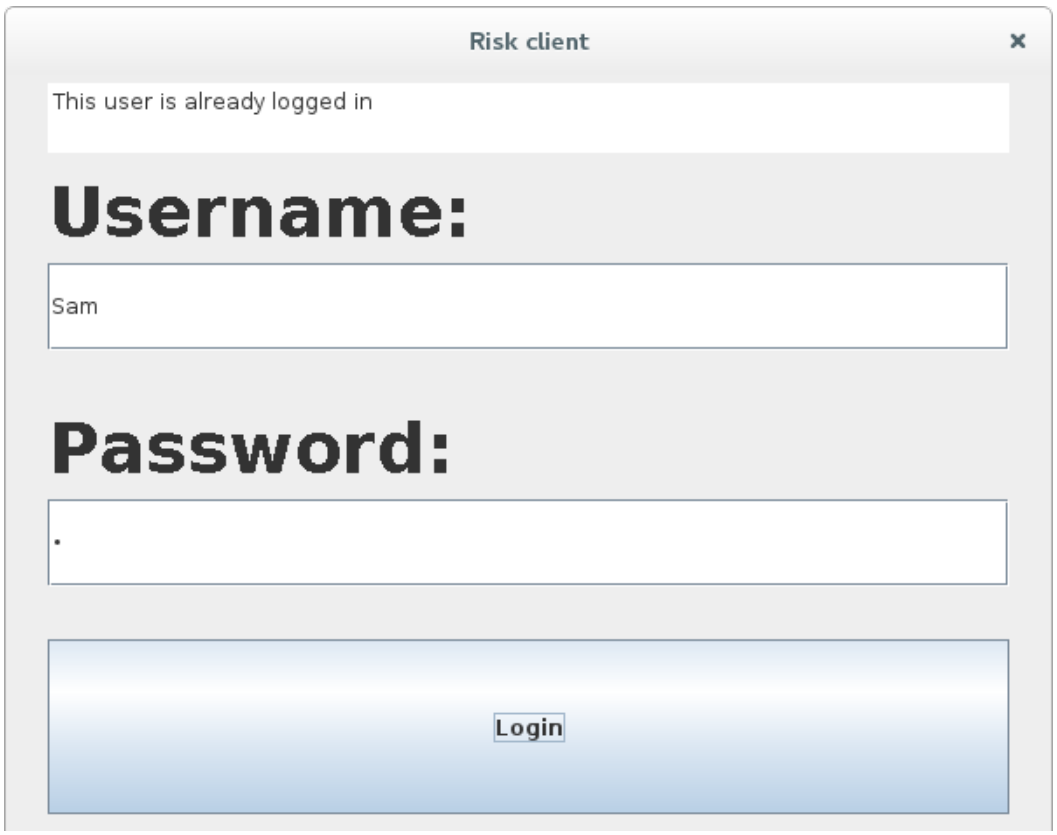
8.1.8 ***Test 9***

Scenario: Logging in as someone who is already logged in

Testing: Check to see that the player is denied access to login

Outcome: Successful



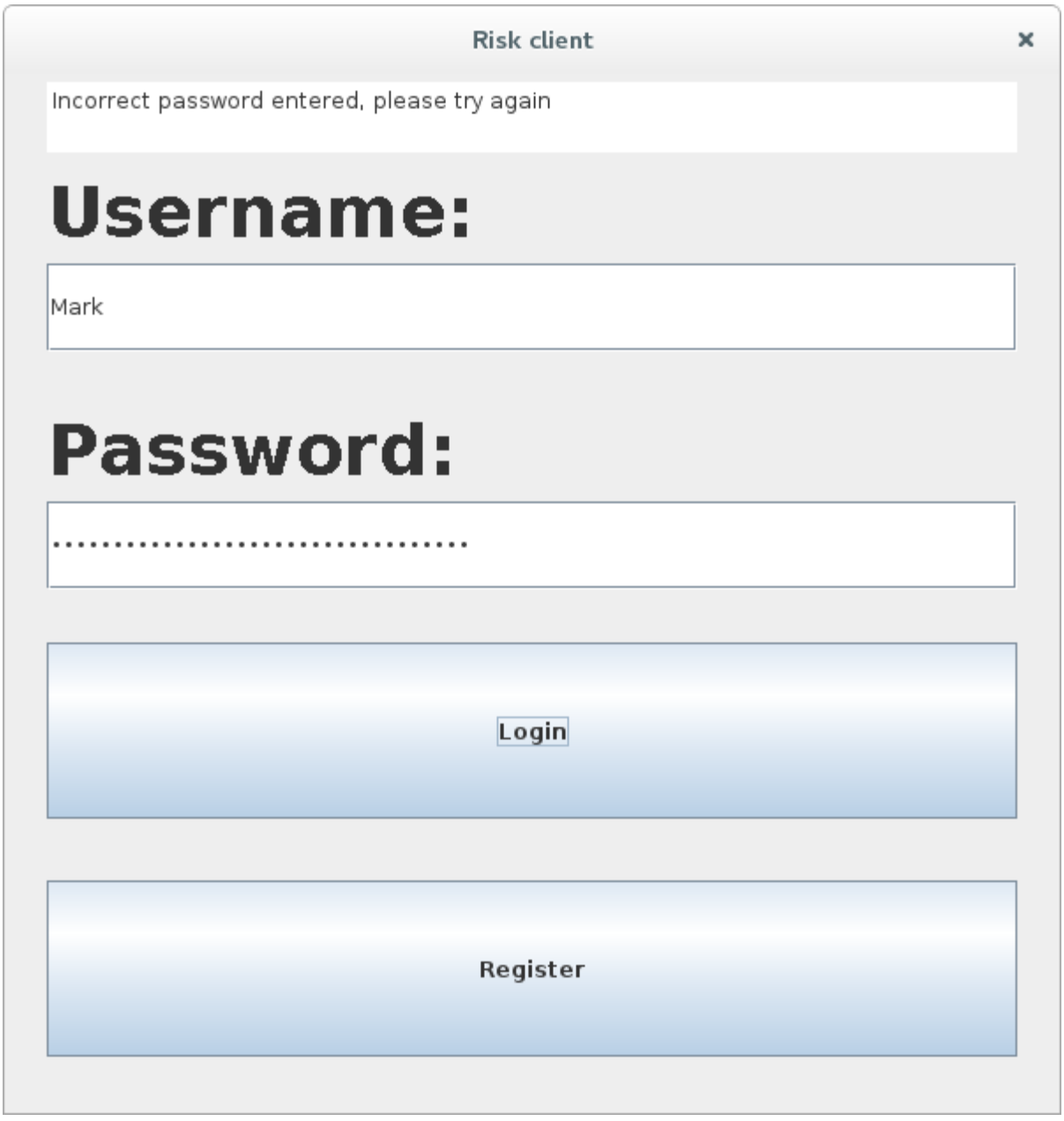


8.1.9 ***Test 10***

Scenario: Logging in with the incorrect login details

Testing: Check to see that the player is denied access to login

Outcome: Successful

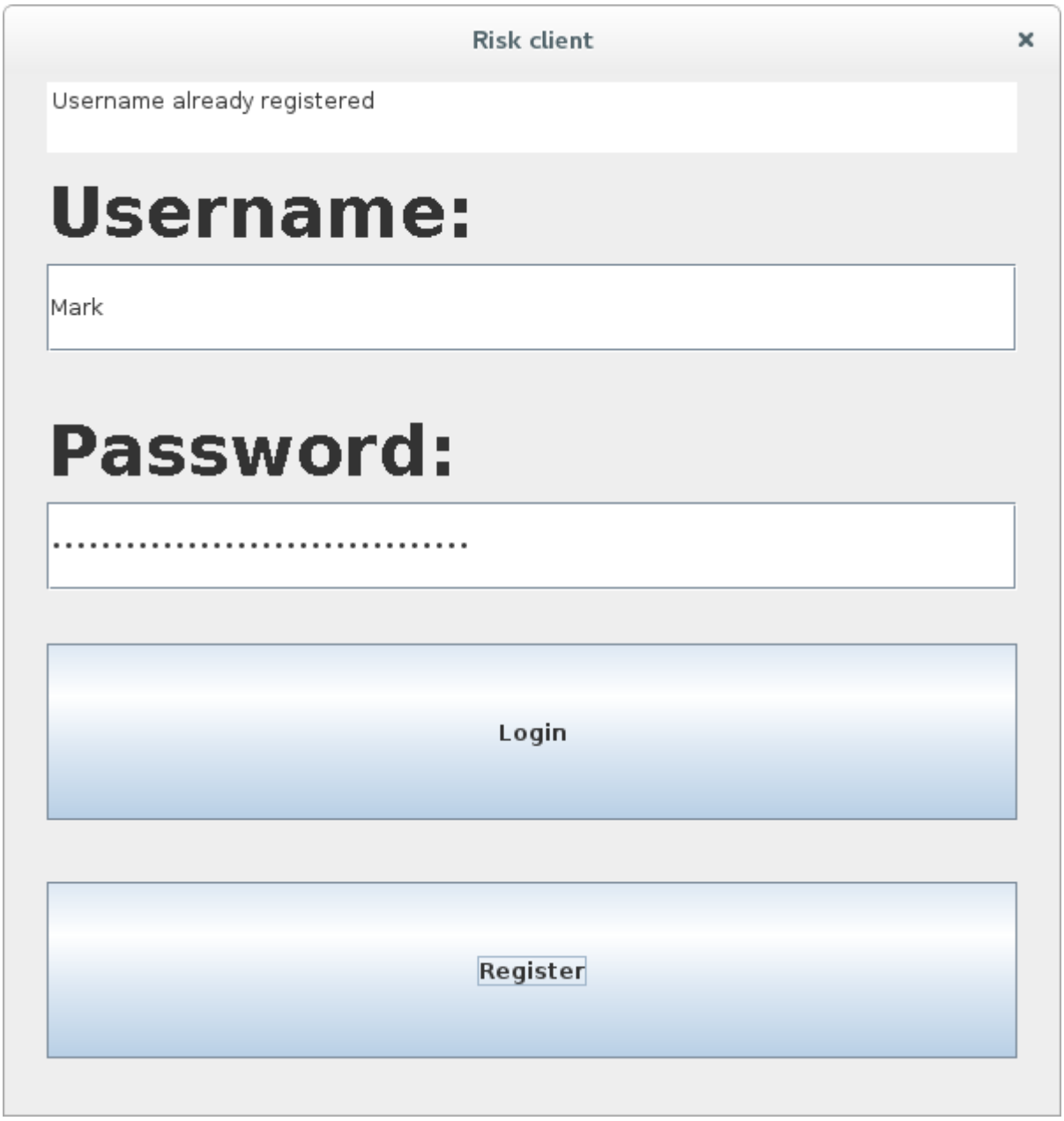


8.1.10 ***Test 11***

Scenario: Register a with a username that already exists

Testing: Check to see that the player is not allowed to register with an existing username

Outcome: Successful

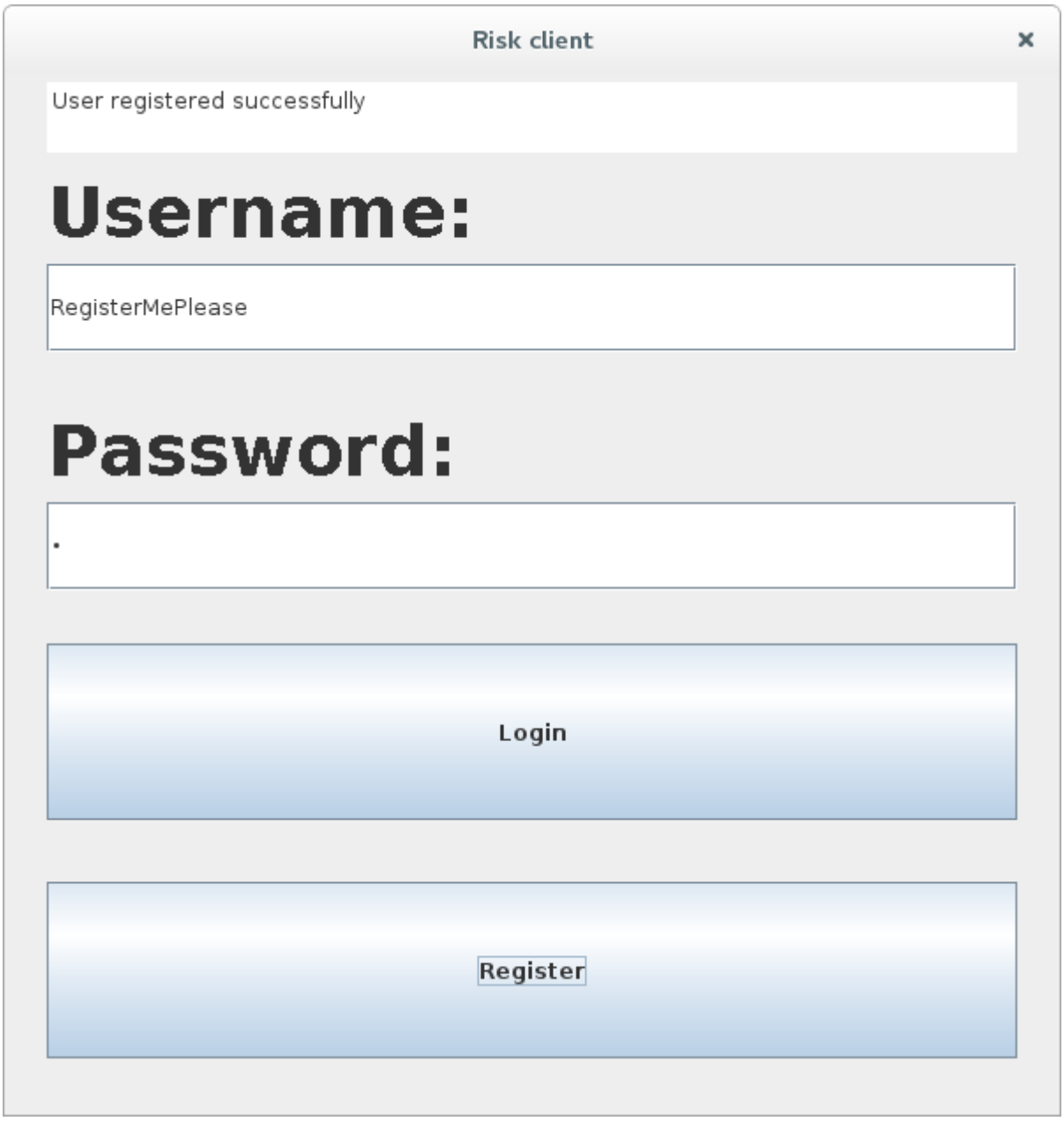


8.1.11 ***Test 12***

Scenario: Register a with legitimate username and password

Testing: Check to see that the player is allowed to register with an existing username

Outcome: Successful

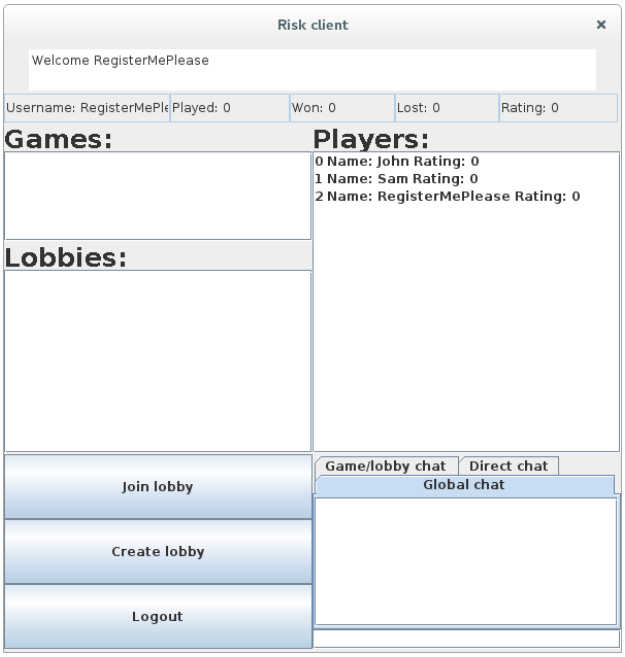


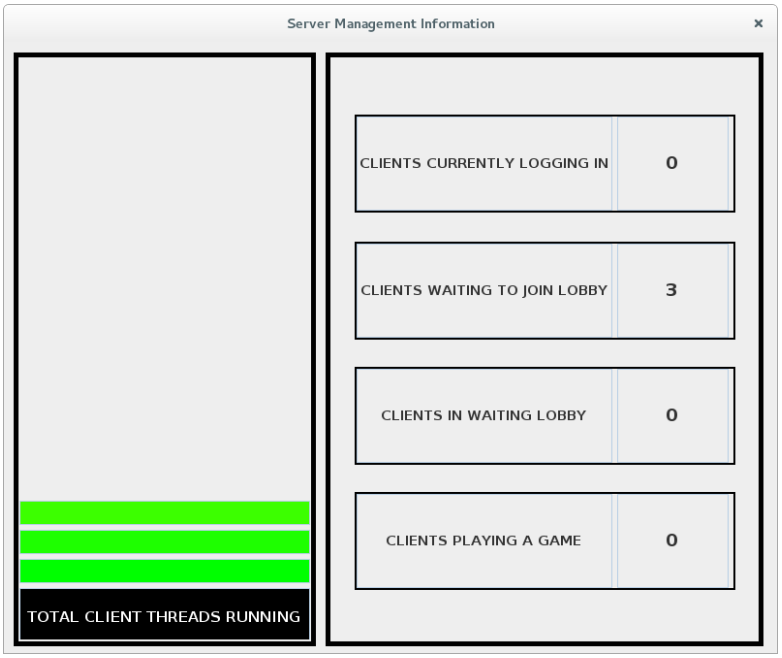
8.1.12 *Test 13*

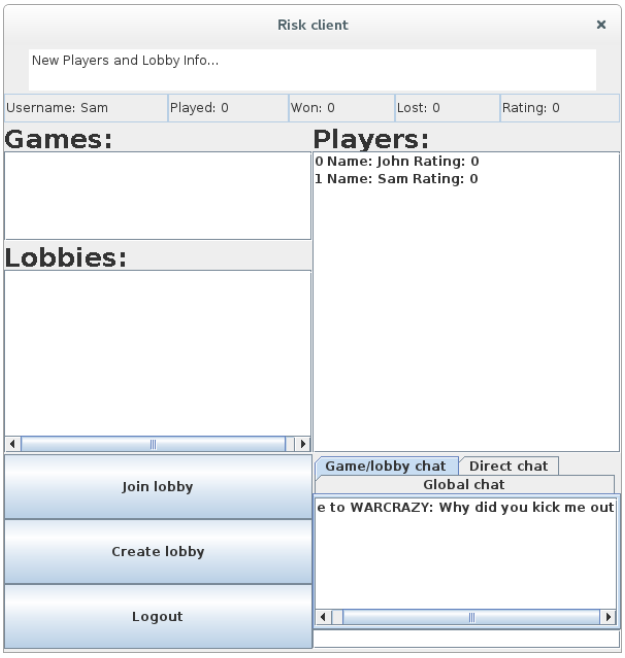
Scenario: Once logged, instead of clicking ‘Logout’ just select close.

Testing: Check to see that the player is removed from all clients view and the server data GUI is updated

Outcome: Successful







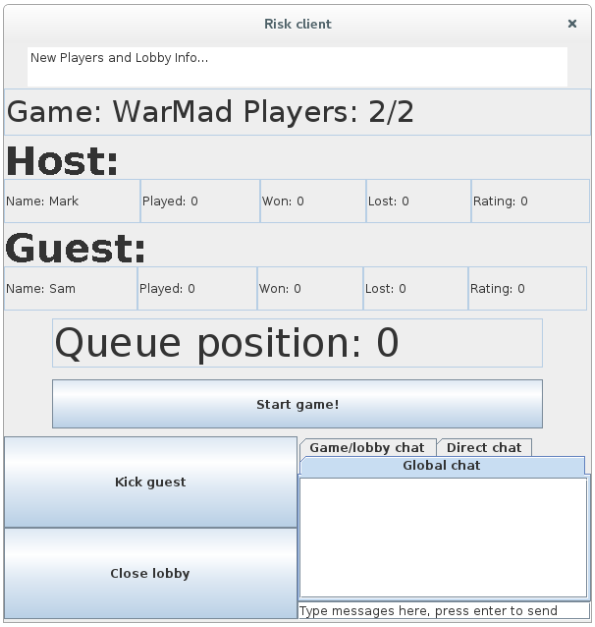
8.1.13 ***Test 14***

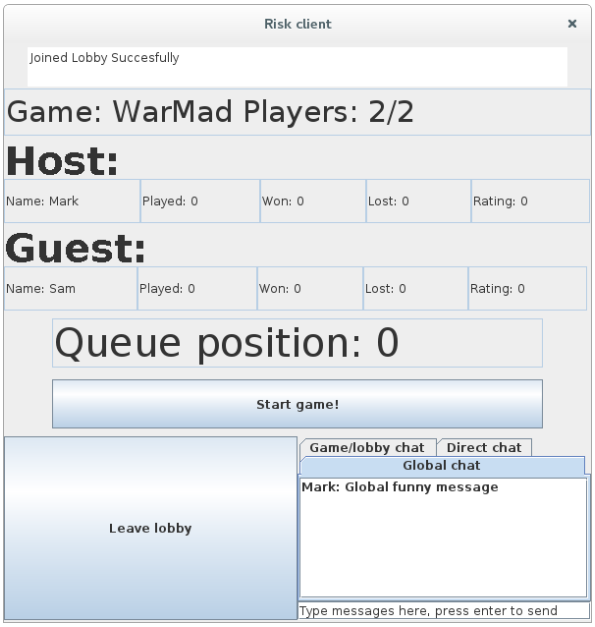
Scenario: Close the host window whilst in a lobby with a guest

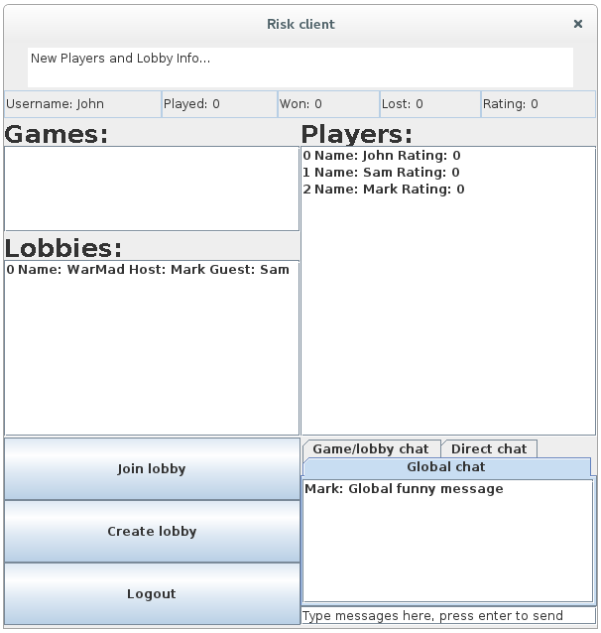
Testing: Check to see that the other player's view is updated accordingly

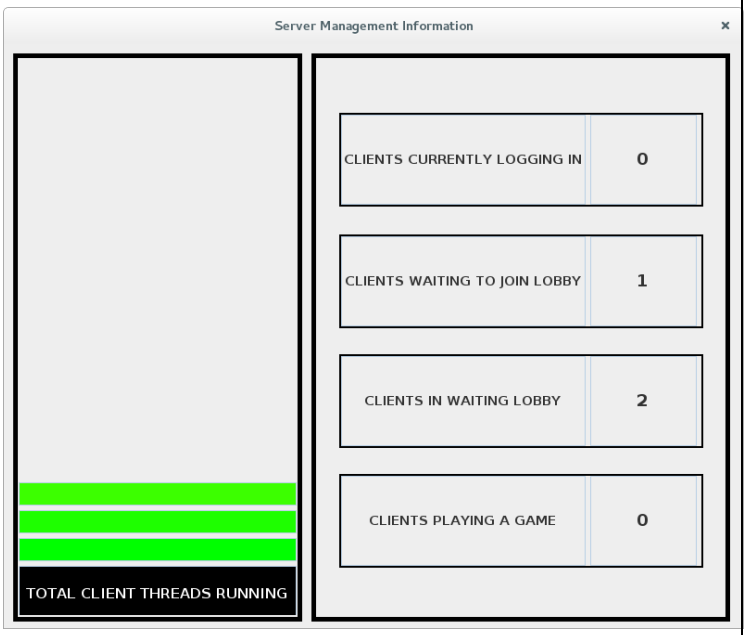
Outcome: Successful

Before closing:

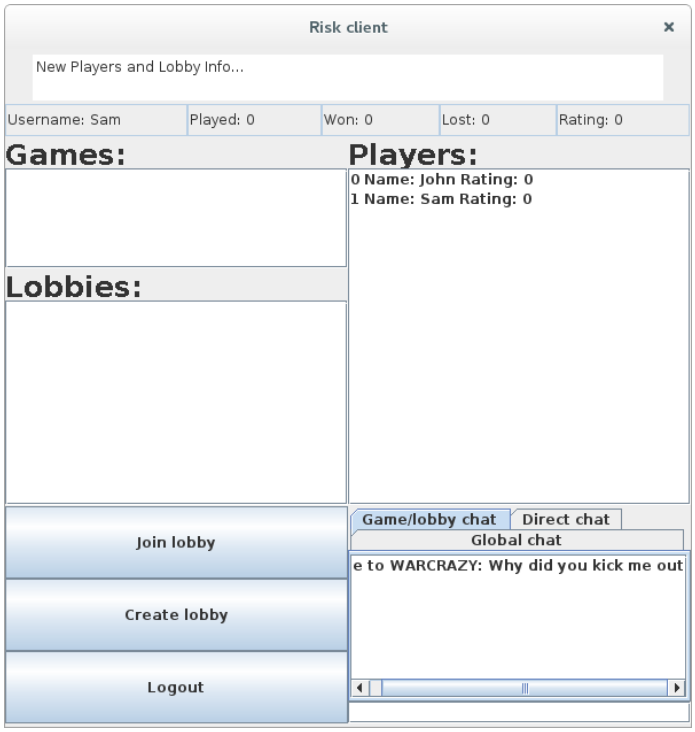








After closing:

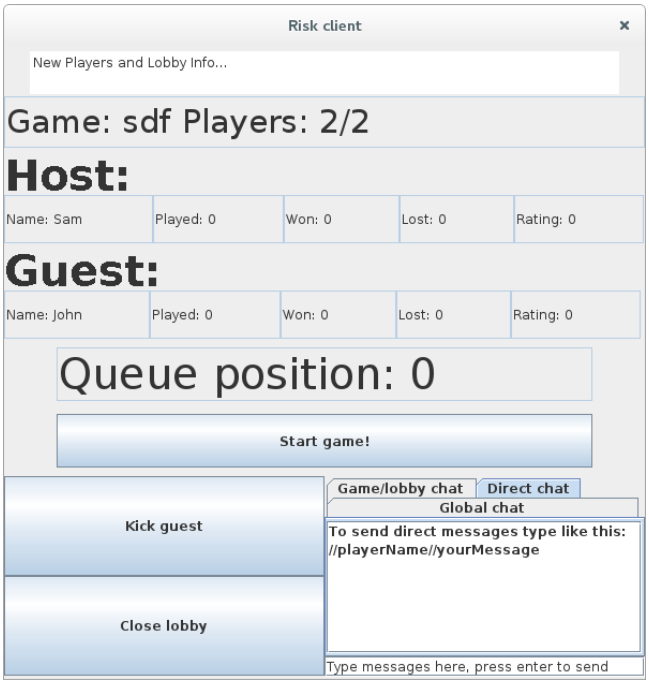


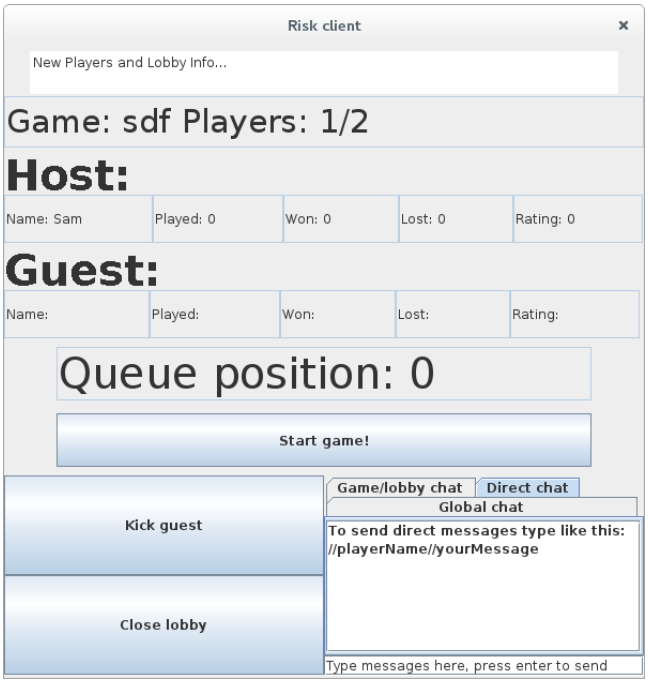
8.1.14 ***Test 15***

Scenario: Close the guest's window whilst in a lobby

Testing: Check that the hosts lobby view is updated to show only 1/2 players.

Outcome: Successful



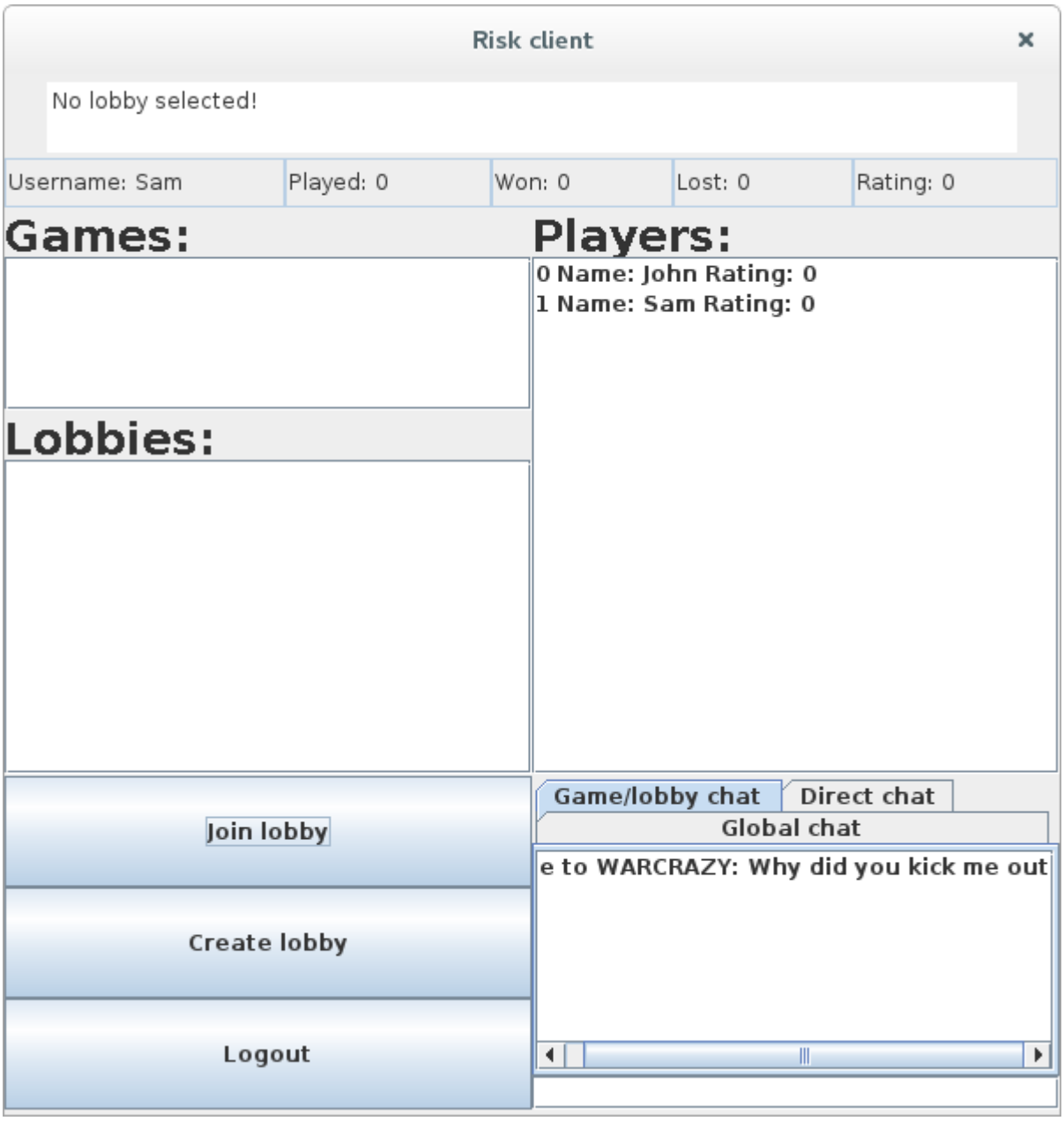


8.1.15 ***Test 16***

Scenario: Join a lobby when there are no lobbies to join

Testing: Check that the user is unable to join a non-existent lobby

Outcome: Successful

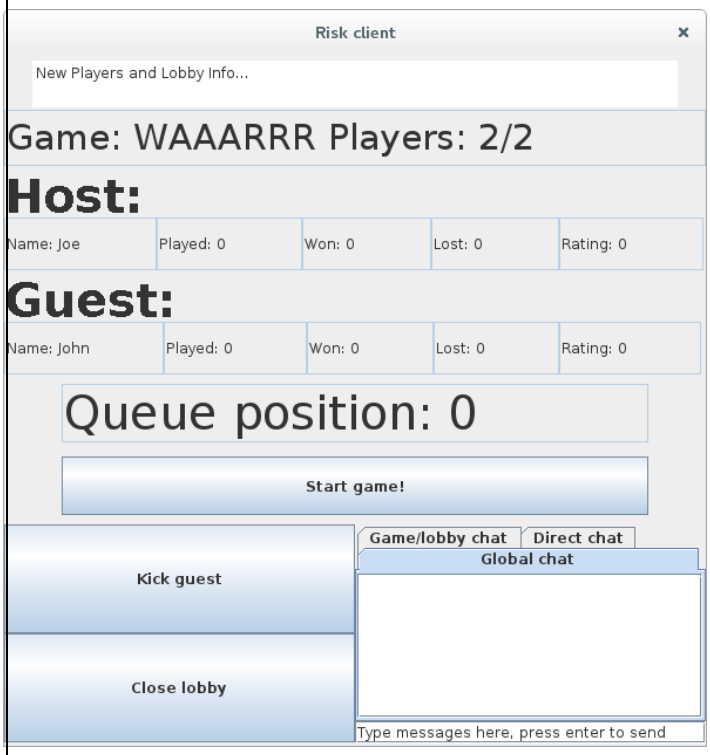


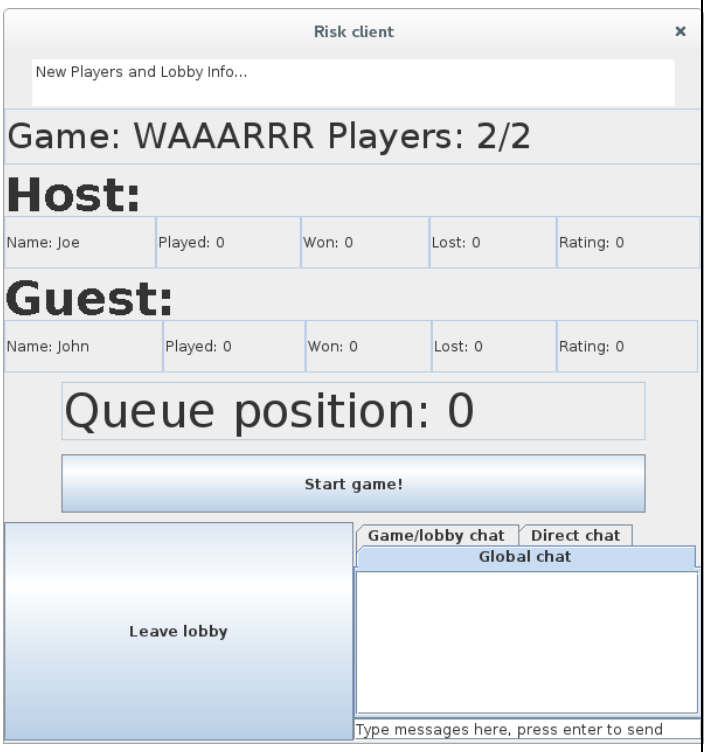
8.1.16 ***Test 17***

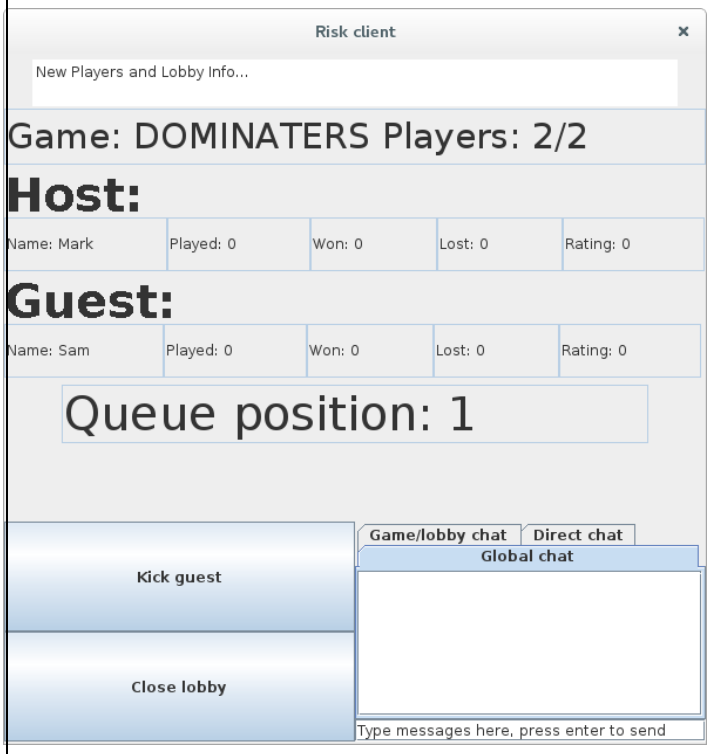
Scenario: When more than one lobby is created

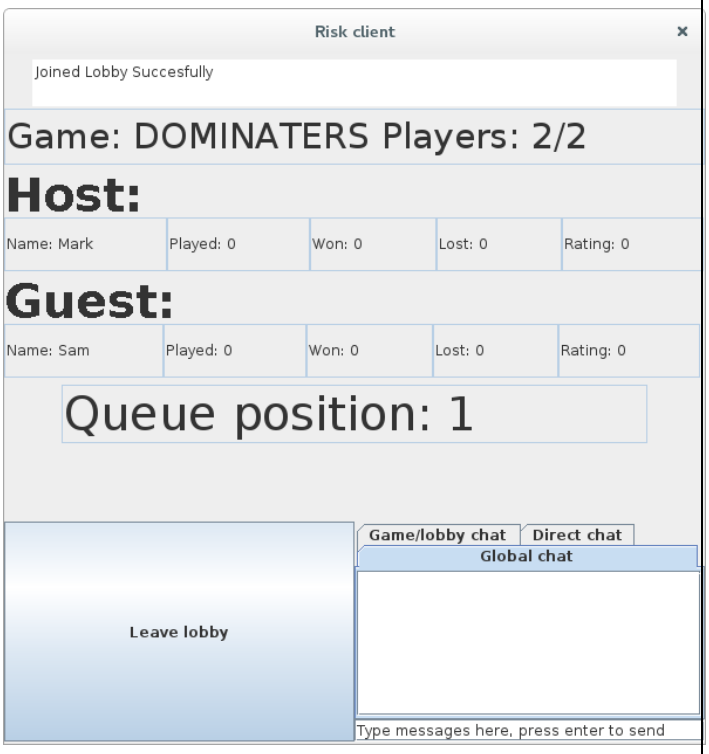
Testing: Checking to see that the lobby queue positions are indexed correctly

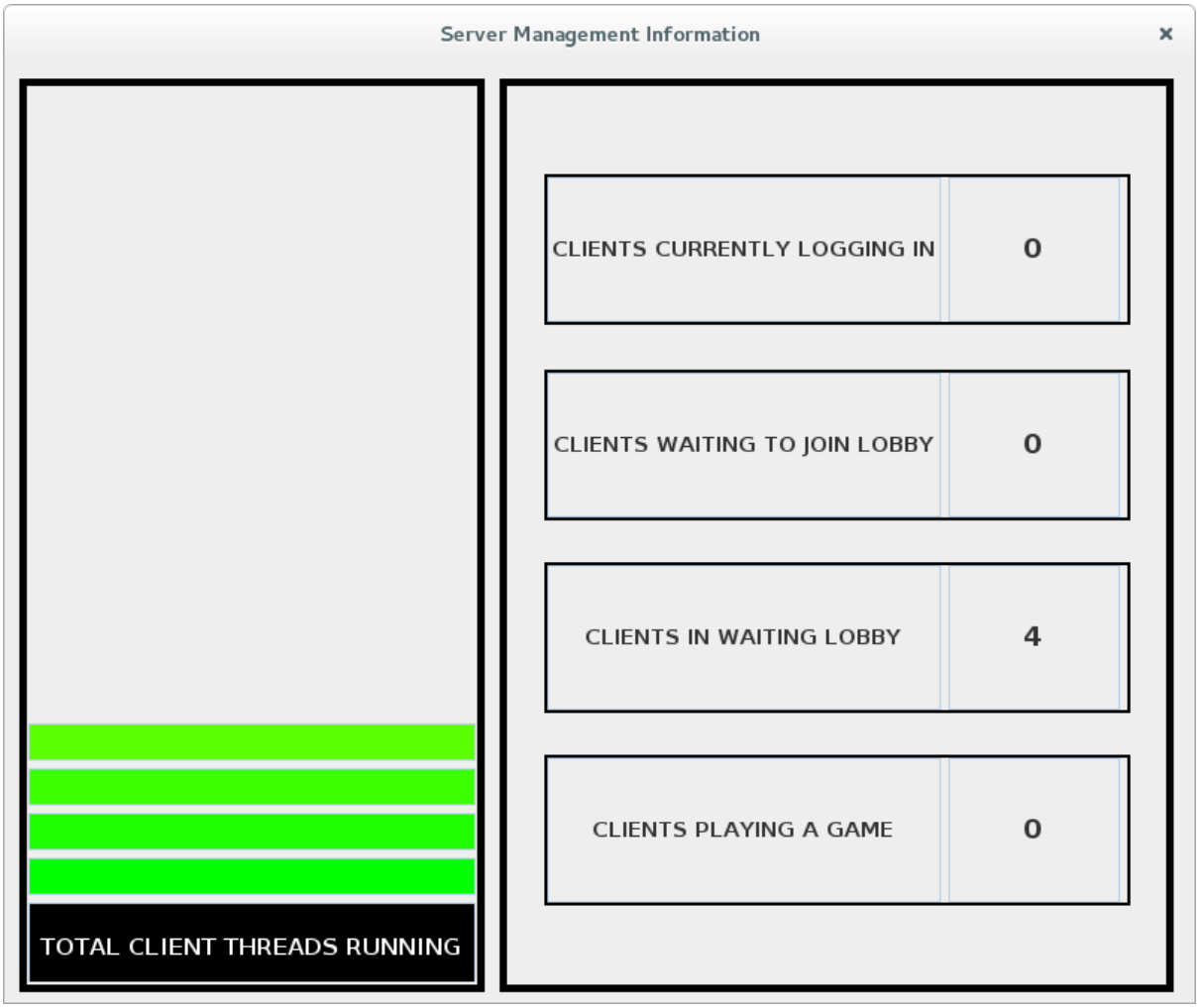
Outcome: Successful










Server Management Information
TOTAL CLIENT THREADS RUNNING
CLIENTS CURRENTLY LOGGING IN
0
CLIENTS WAITING TO JOIN LOBBY
0
CLIENTS IN WAITING LOBBY
4
CLIENTS PLAYING A GAME
0

8.2 DATABASE TESTING

The database has been thoroughly tested using Junit tests to test the read methods and a main method to write data onto the database for the write methods.

An example of a Junit test conducted is shown below:

```
public void test5(){


        User user = d1.register("bob$123", "CS124");


        String expectedMessage =
    "Username or password does not match the required format, please enter a username
    and password using only alphanumeric characters";
        String actualMessage = user.getMessage();


        assertEquals(expectedMessage, actualMessage);


    }
```

The above test checks that the registration method sends the correct message to the server as part of the user object when the username entered is not in alphanumeric format.

8.2.1 *Users Table*

MBYrne > mxb680 > users — Connected.

id	username	password	games_played	losses	wins	rating
1	sam	jones	8	5	4	2
2	matthew	byrne	8	5	4	2
3	Mark	1	20	10	10	2
4	David	computerscience	5	5	0	5
5	Alex45	password	5	5	0	5
6	dan464	marley	0	0	0	0
7	John	computerscience	5	5	0	5
8	dan	marley	0	0	0	0
9	qwerty	asdfg	0	0	0	0
10	asdfg	qwerty	0	0	0	0

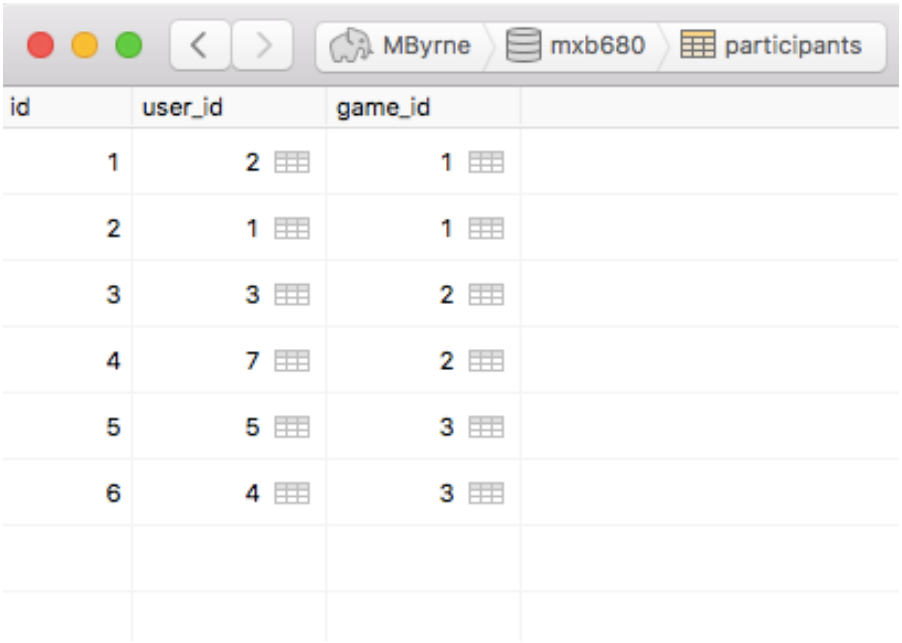
1. The above table was populated both through users registering and logging into the system as well as through insertion SQL statements in the terminal.
2. We added users with several games played and a win/loss record to ensure that the ratings formula used in the User class worked successfully.

3. The *saveGameDatabase* method was tested in the *DatabaseManagerMain* class. Three new games were saved and the below tables show the state of the participant, game and country tables afterwards.

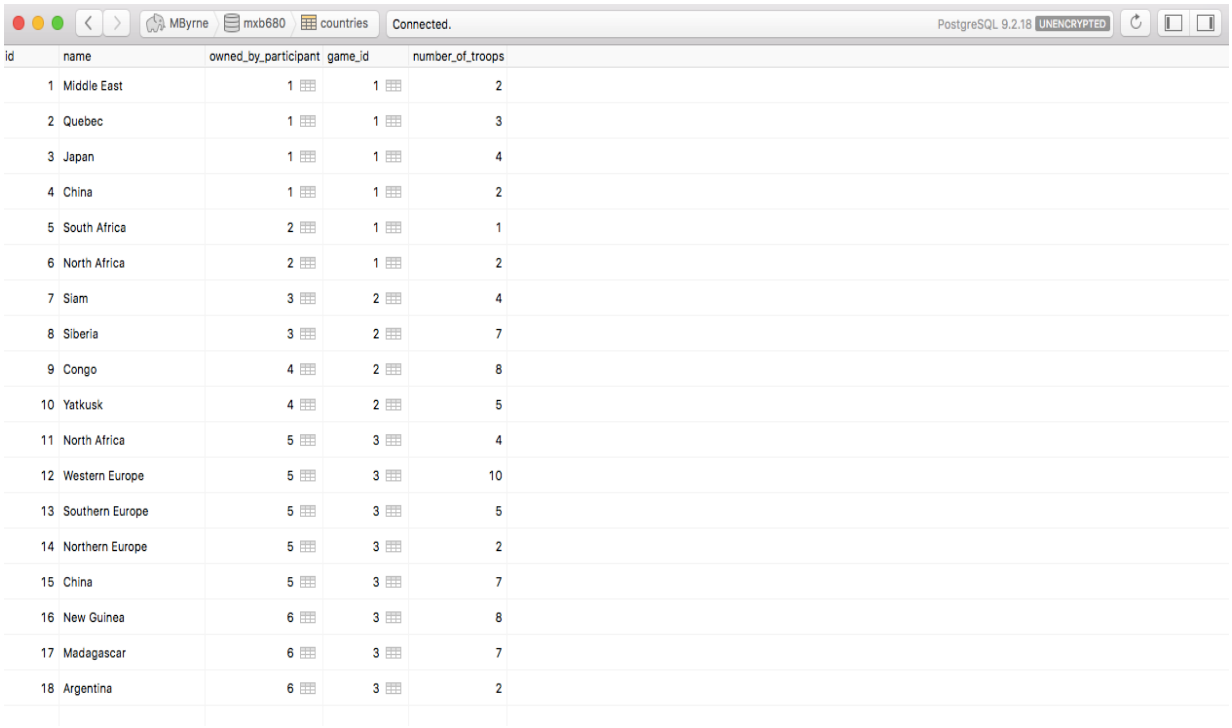
Saved Games

Name	Host	hostTurn	Countries and Troops	Guest	guestTurn	Countries and Troops
riskGame	matthew	true	Middle East – 2 Quebec – 3 Japan – 4 China – 2	sam	false	South Africa – 1 North Africa – 2
riskgame2	Mark	true	Siam – 4 Siberia - 7	John	false	Congo – 8 Yatkusk – 5
riskGame3	Alex45	false	North Africa – 4 Western Europe - 10 Southern Europe –5 Northern Europe –2 China - 7	David	true	New Guinea – 8 Madagascar – 7 Argentina - 2

8.2.2 ***Participants Table***

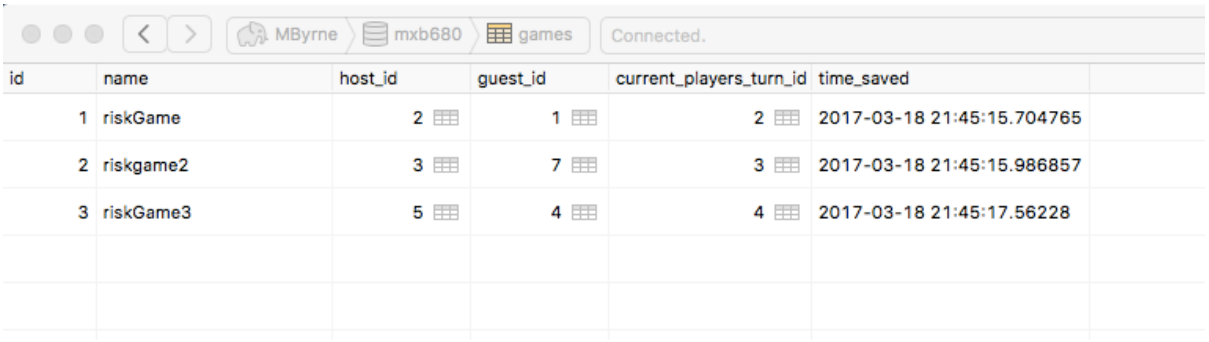


8.2.3 *Countries Table*



id	name	owned_by_participant	game_id	number_of_troops
1	Middle East	1	1	2
2	Quebec	1	1	3
3	Japan	1	1	4
4	China	1	1	2
5	South Africa	2	1	1
6	North Africa	2	1	2
7	Siam	3	2	4
8	Siberia	3	2	7
9	Congo	4	2	8
10	Yatkusk	4	2	5
11	North Africa	5	3	4
12	Western Europe	5	3	10
13	Southern Europe	5	3	5
14	Northern Europe	5	3	2
15	China	5	3	7
16	New Guinea	6	3	8
17	Madagascar	6	3	7
18	Argentina	6	3	2

8.2.4 *Games Table*



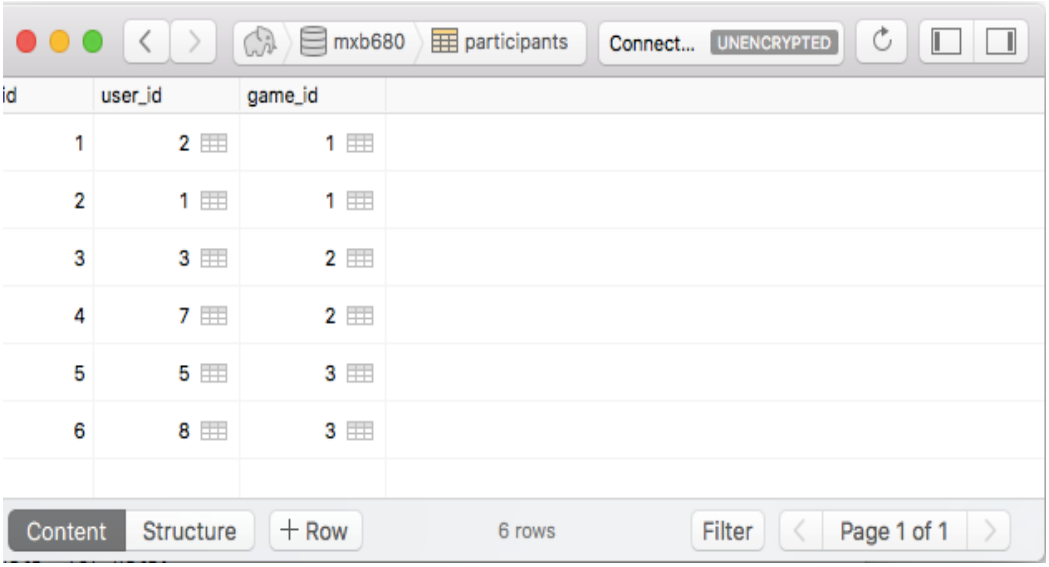
id	name	host_id	guest_id	current_players_turn_id	time_saved
1	riskGame	2	1	2	2017-03-18 21:45:15.704765
2	riskgame2	3	7	3	2017-03-18 21:45:15.986857
3	riskGame3	5	4	4	2017-03-18 21:45:17.56228

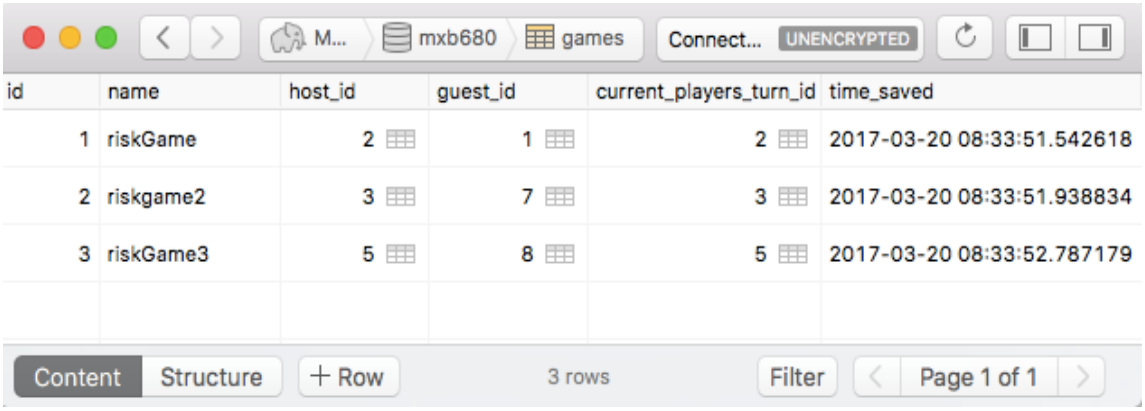
4. The last test we completed on the database was to save a game with a new guest which had already been previously saved.

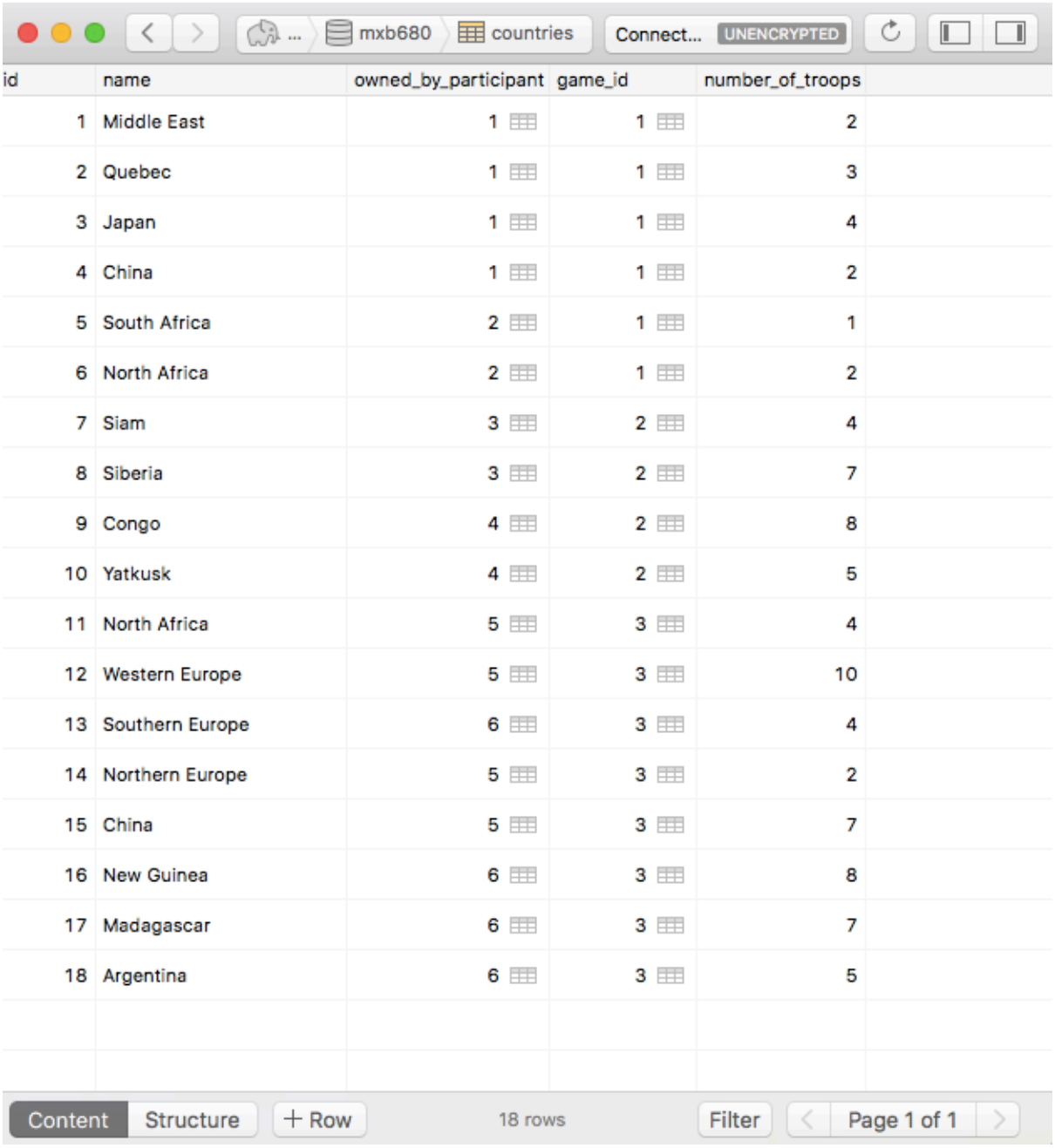
Amended Game

Name	Host	hostTurn	Countries & Troops	guest	guestTurn	Countries & Troops
riskGame3	Alex45	true	North Africa - 4 Western Europe – 10 Northern Europe – 2 China - 7	dan	false	New Guinea – 8, Madagascar – 7 Southern Europe - 4

Amended Tables







id	name	owned_by_participant	game_id	number_of_troops
1	Middle East	1	1	2
2	Quebec	1	1	3
3	Japan	1	1	4
4	China	1	1	2
5	South Africa	2	1	1
6	North Africa	2	1	2
7	Siam	3	2	4
8	Siberia	3	2	7
9	Congo	4	2	8
10	Yatkusk	4	2	5
11	North Africa	5	3	4
12	Western Europe	5	3	10
13	Southern Europe	6	3	4
14	Northern Europe	5	3	2
15	China	5	3	7
16	New Guinea	6	3	8
17	Madagascar	6	3	7
18	Argentina	6	3	5

8.1 TESTING THE GUI

The plan for testing the gui involved predominantly acting as the user, and running through the GUI. As the code is provided alongside this report, to test the GUI you are able to run the game the exact same way, and check to see if the game runs correctly.

However, due to time constraints, we were not able to test the GUI with other methods, however have added in a test plan that shows exactly how you would test the GUI. This includes;

8.1.1 *Implementing an AI*

This could be done by creating an artificial intelligence that would attempt to beat another user, the tester, whilst having four parameters given to it; the initial state of our game, the goal state where the AI owns all the countries, the location of the combo boxes that the AI can press to deploy/attack/fortify on.

As the game is heavily dependant upon probability, the game can be won by an AI by using a variety of different method. One method, presented by Garrett Robinson (http://web.mit.edu/sp.268/www/risk.pdf) . Furthermore, a Monte Carlo simulation could be implemented to simulate each roll, and calculate the outcome of the roll. Garret shows how his research into the game Risk leads to the following heatmap that is showing a monte carlo simulation, implemented in MATLAB, simulating all the attacks and defence between troops.

8.1.2 *Mouse position capture using Robot class*

To test the GUI, we would run further test that creates a robot that has been given certain location to click on the screen. These clicks would then transfer into selections on the GUI. This test is similar to a person clicking on buttons on the actual screen. Here is a snippet of the code:

```
try {
        Robot robot = new Robot();
        robot.setAutoDelay(5);
        int mask = InputEvent.BUTTON1_MASK;


        robot.delay(100);
        robot.mouseMove(350,50);
        robot.mousePress(mask);
        robot.mouseRelease(mask);
        robot.keyPress(KeyEvent.VK_END);
        robot.keyRelease(KeyEvent.VK_END);


        for (int i = 0; i < "Please enter URL".length(); i++){
           robot.keyPress(KeyEvent.VK_BACK_SPACE);
           robot.keyRelease(KeyEvent.VK_BACK_SPACE);
        }
```

The above uses the Robot class, which is used to generate native system input events for the purpose of our test. Hence, this would replicate what a user would do.

8.2 RISK ENGINE TESTING

The tests shown below are a subset of all the tests simply showing that the two game objects are effectively the same except mirrored as required in line with the respective views each player should have. G1 represents the game object that is sent to the host where g1me is the host & g1other is the guest. g2 represents the game object sent to the guest, meaning that g2me is the guest and g2other is the host.

Test 1: Testing that the countries allocated match up in both objects (i.e. the host has their respective country allocations in both objects as identical and same for the guest)

Outcome: Success

```
@Test
public void matchingCountryAllocation() {

    Set<Country> g1meCountries = g1me.getOwnedTerritories().keySet();
    Set<Country> g1otherCountries = g1other.getOwnedTerritories().keySet();

    assertTrue(g1meCountries.equals(g2other.getOwnedTerritories().keySet()));
    assertTrue(g1otherCountries.equals(g2me.getOwnedTerritories().keySet()));

}
```

Test 2: Checking that the troop number is the same between each game and all Player objects

Outcome: Success

```
@Test
public void correctTroopNumberTest() {

    int g1metroopCount = g1me.getNumberOfTroops();
    int g1othertroopCount = g1other.getNumberOfTroops();

    int g2metroopCount = g2me.getNumberOfTroops();
    int g2othertroopCount = g2other.getNumberOfTroops();

    assertEquals(g1.getTroopNumber(), g2.getTroopNumber());

    assertEquals(g1.getTroopNumber(), g1metroopCount);
    assertEquals(g1.getTroopNumber(), g1othertroopCount);

    assertEquals(g2.getTroopNumber(), g2metroopCount);
    assertEquals(g2.getTroopNumber(), g2othertroopCount);
}
```

Test 3: Checking that the individual troop count for each territory owned by any player is the same across both objects

Outcome: Success

```
@Test
public void matchingTroopAllocation() {

    //Checking whether the host has their troops synchronized across both games

    Iterator<Country> g1meCountries = g1me.getOwnedTerritories().keySet().iterator();

    ArrayList<Integer> g1meTroopCounts = new ArrayList<>();
    ArrayList<Integer> g2otherTroopCounts = new ArrayList<>();

    while (g1meCountries.hasNext()) {
        Country country = g1meCountries.next();

        int g1meCountryTroopCount = g1me.getOwnedTerritories().get(country);
        int g2otherCountryTroopCount = g2other.getOwnedTerritories().get(country);

        g1meTroopCounts.add(g1meCountryTroopCount);
        g2otherTroopCounts.add(g2otherCountryTroopCount);

    }

    assertEquals(g1meTroopCounts, g2otherTroopCounts);


//Checking whether the guest has their troops synchronized across both games

Iterator<Country> g1otherCountries = g1other.getOwnedTerritories().keySet().iterator();

ArrayList<Integer> g1otherTroopCounts = new ArrayList<>();
ArrayList<Integer> g2meTroopCounts = new ArrayList<>();

while (g1otherCountries.hasNext()) {

    Country country = g1otherCountries.next();

    int g1otherCountryTroopCount = g1other.getOwnedTerritories().get(country);
    int g2meCountryTroopCount = g2me.getOwnedTerritories().get(country);

    g1otherTroopCounts.add(g1otherCountryTroopCount);
    g2meTroopCounts.add(g2meCountryTroopCount);

}

assertEquals(g1otherTroopCounts, g2meTroopCounts);
```

9 TEAM ORGANIZATION REPORT

The team project was split into the following sections:

Name of system sub-section worked on	Name of person(s) who worked on sub-section
Server	Mark Alston
Client	Oliver Kamperis
Database	Matt Byrne
Game Logic	Khalid Ramadan
GUI	Abdikhaliq Timer

10 PROJECT DIARY AND MINUTES

10.1 MEETING MINUTES

Date: 14th February 2017
Time: 13:00
Attendance: All Members
Discussion: All team members read through the project specifications, after having a brief introduction given in the software workshop tutorial. Possible ideas were then discussed as a group. These ideas included battleships, RISK, scrabble, rate your landlord and snakes and ladders. It was decided to meet again to discuss the project further on the 16th February. In this time members were asked to think about other ideas for the project and to read through the project specification again.

Date: 16th February 2017
Time: 12:00 (noon)
Attendance: All members
Discussion: Further ideas were discussed as a group, but it was decided unanimously that RISK or battleships would be the most interesting option to pursue. It was recognised that RISK would be more complex than battleships so to determine the feasibility of the project a rough flow diagram was drawn to get an idea of what the game would involve. It was decided to go for the more complex option of RISK. Different sections of the project were then delegated to individual team members to research. And functional requirements of each part were to be produced by the researching member for the following meeting. The delegation was made as such;

Server: Mark
Client: Ollie
Database: Matt
GUI: Timer
Game model: Khalid

Date: 21st February 2017
Time: 13:00
Attendance: All members
Discussion: The group watched Alex's example of how to write a server – client, as well as how to connect using Java to the schools database. Next, we discussed project goal deadlines for the following week. It was decided that a feasible goal would be to have an operational GUI to handle login / register requests using the database. Ollie and Mark decided to meet up separately to discuss the client server protocols. It was decided between Matt and Mark that a single Boolean instance method named '*checkLogin()'* would be used to check the legitimacy of the users login details. Abdi and Khalid discussed in how our version of Risk would work, deciding upon the classes needed for the game.

Date: 23rd February2017
Time: 13:00
Attendance: Ollie, Mark, Matt
Discussion: Communication protocols were discussed, and it was decided to use serializable objects over an object input / output stream to communicate between the client and server. It was decided that all objects should either extend '*request'* or '*response'*. One is sent by the client and the other is sent by the server respectively. Furthermore, for debugging reasons, and for the purpose of informing the user, both super classes would include a field '*message'* which could be displayed on both sides to keep an eye on what was being sent. It was decided that there would be a generic *ServerDataCollection* object which was sent every time the number of players, lobbies or games changed.

Date: 25rd February2017
Time: 13:00
Attendance: Khalid and Abdi
Discussion: Discussion on how to combine the GUI with the logic of the game, the discussion involved on deciding the exact screens which the user would be shown at the various stages of the game. Including finalising the design for the GUI, and the general theme of the GUI.

Date: 28th February 2017
Time: 13:30
Attendance: All members
Discussion: Discussed the communication protocols with Alex. And showed the login and register screen. However due to a questionable connection to the database through the VPN the GUI did not work correctly. Another goal was set to be able to start a basic game in the CS downstairs lab the following week. Furthermore, it was decided that a database ER diagram would be available for the following week and all tables would have been already created.

Date: 1st March 2017
Time: 09:00
Attendance: Ollie, Mark

Discussion: Further communication objects were discussed as well as each' individual protocols. The new objects would allow users to create, join, load and leave lobbies, as well as kick guests from lobbies.

Date: 2nd March 2017
Time: 13:00
Attendance: All members
Discussion: Discussed including a user rating based on number of wins/losses which would be kept in the database within the user table. Set a goal that a simple game GUI as well as the background game functionality should be ready for Tuesday.

Date: 5th March 2017
Time: 13:00
Attendance: Abdi and Khalid
Discussion: The GUI had been completely designed at this stage, however, the client side needed to show information to the user, hence more labels were introduced to the GUI.

Date: 7th March 2017
Time: 14:00
Attendance: Matt, Khalid, Timer, Ollie
Discussion: Tried to show Alex the pre-game GUI (lobbies and game functionality) however due to a slight error in the server client, a connection across two lab computers was not possible. Timer also showed Alex the game GUI, but it was not yet connected to Ollie's client. A goal was set to implement the server client messaging system between players and lobbies, as well as being able to start a game for next time we see Alex. Discussed the save and load game methods for the database. It was decided an ArrayList would be sent between the client, server and database containing all the game information.

Date: 9th March 2017
Time: 13:30
Attendance: Ollie, Mark
Discussion: Discussed the messaging protocols and the object names in which to request messages to be sent. It was decided that all messages would extend *message* and that no reply would be required when messages were sent by a client.

Date: 13th March 2017
Attendance: Matt, Ollie, Mark
Discussion: All functionality of the server client and database was tested and debugged. Several issues were picked up and resolved, such as logged in players once logged out could not re-login. Some miscommunication between Mark and Ollie meant that the server was not capable of a user outside of a lobby messaging a lobby. This functionality was added and debugged. Some issues remained and all attendees decided to meet up again on Wednesday.

Date: 14th March 2017
Time: 13:30
Attendance: All members

Discussion: Showed Alex the almost operational pre game GUI. All worked apart from the fact if a host closed their screen whilst in a lobby the guests view was not sent back to the general view. The game GUI was shown to Alex, however it could not be started from the pre-game GUI yet. The server Client was then debugged for the rest of the known faults.

Date: 16th March 2017
Time: 14:00
Attendance: All members
Discussion: Implemented the connection between the game GUI and the client. By the end of the meeting the game could be started, however the game view was not consistently being generated for both of the clients. Debugging of the database method to save a game was done by Matt, this was done as the method was not operating correctly if the game save was an update rather than a completely new game being saved.

Date: 17th March 2017
Time: 10:00
Attendance: Timer, Khalid, Matt, Mark
Discussion: Added a server GUI manager to see all server data visually. Made screen shots to show the testing of the pre-game GUI that took place. Timer and Khalid continued to debug the game and Matt continued to debug the save game method.

Date: 19th March 2017
Time: 13:30
Attendance: All members
Discussion: Timer and Khalid continued to debug the game GUI. Matt, Mark, and Ollie, began collating individuals report sections together into the final report.

10.2 FALL-BACK REPORT

The following was how we split the tasks up, including a fall back section which the person would also study/learn in order to be able to step into at any moment. Hence, there was close communication between the following:

Server/Client/Communication; Ollie, Mark and Matt all focused upon their individual sections, however were consistently updating one another and explaining to each other their code, hence, if someone were to have dropped out of the assignment unexpectedly, the other team members would have them covered.

Game logic and GUI; Abdi and Khalid did the same thing as noted above, so that if one of them unexpectedly dropped out, the other would be able to carry on from that position.

11 EVALUATION

The final software which was created can be found within the SVN at –

https://codex.cs.bham.ac.uk/svn/modules/2016/msc-sw/boston/Team%20Boston/

The above link contains the work done by all members of this group. When we started this project, we knew we would have to spend a lot of time learning things which we did not know in order to create our ambitious game. Our report has shown that we have all spent weeks working on creating a game that can be played, and most importantly works.

A big factor which we had to consider for this project was Time. We were given a limited time schedule to complete a task of our choosing. As the report indicates, we spent a lot of time on our game, however, we were heavily constrained due to time. We have included some of the things which we would have done differently/improved upon if we were to start this project over, or if we had more time:

1. The game could have had the same GUI look for the user experience.
2. A change password function could have been implemented.
3. The GUI could have had the following improvements:
    a. The individual territories could have become buttons
    b. JavaFX could have been used to implement more animations
4. Another implementation the team really wanted to implement, but due to time constraints were not able to, was to have more than 2 users on a single map. The server and game logic both allow for scalability, hence it will be relatively easy to implement and build upon from the current code.
5. In terms of the database, given more time we would implement more tables such as results and moves. We would also try to ensure that the tables adhered to third normal form.

12 REFERENCES

Andrew Liszewski. 2017. Risk Is Getting a Complete Makeover in 2016. [ONLINE] Available at: https://gizmodo.com/risk-is-getting-a-complete-makeover-in-2016-1735053374. [Accessed 19 March 2017].

flaregames. 2017. Alliance Wars Map >> Risk Board Game - Suggestions & Improvements - flaregames. [ONLINE] Available at: http://forums.flaregames.com/topic/5550-alliance-wars-map-risk-board-game/. [Accessed 19 March 2017].

Flickr. 2017. Flickr. [ONLINE] Available at: https://www.flickr.com/photos/leftyimages/3286412705. [Accessed 19 March 2017].

Play Free Flash & HTML5 Games with FunkyPotato.com!. 2017. RISK: WORLD CONQUEST. [ONLINE] Available at: http://funkypotato.com/risk-world-conquest/. [Accessed 19 March 2017].

Risk: Adventures of a Board Game during a Radio Show | Tom and Alex | triple j. 2017. Risk: Adventures of a Board Game during a Radio Show | Tom and Alex | triple j. [ONLINE] Available at: http://www.abc.net.au/triplej/tomandalex/blog/s2791341.htm. [Accessed 19 March 2017].

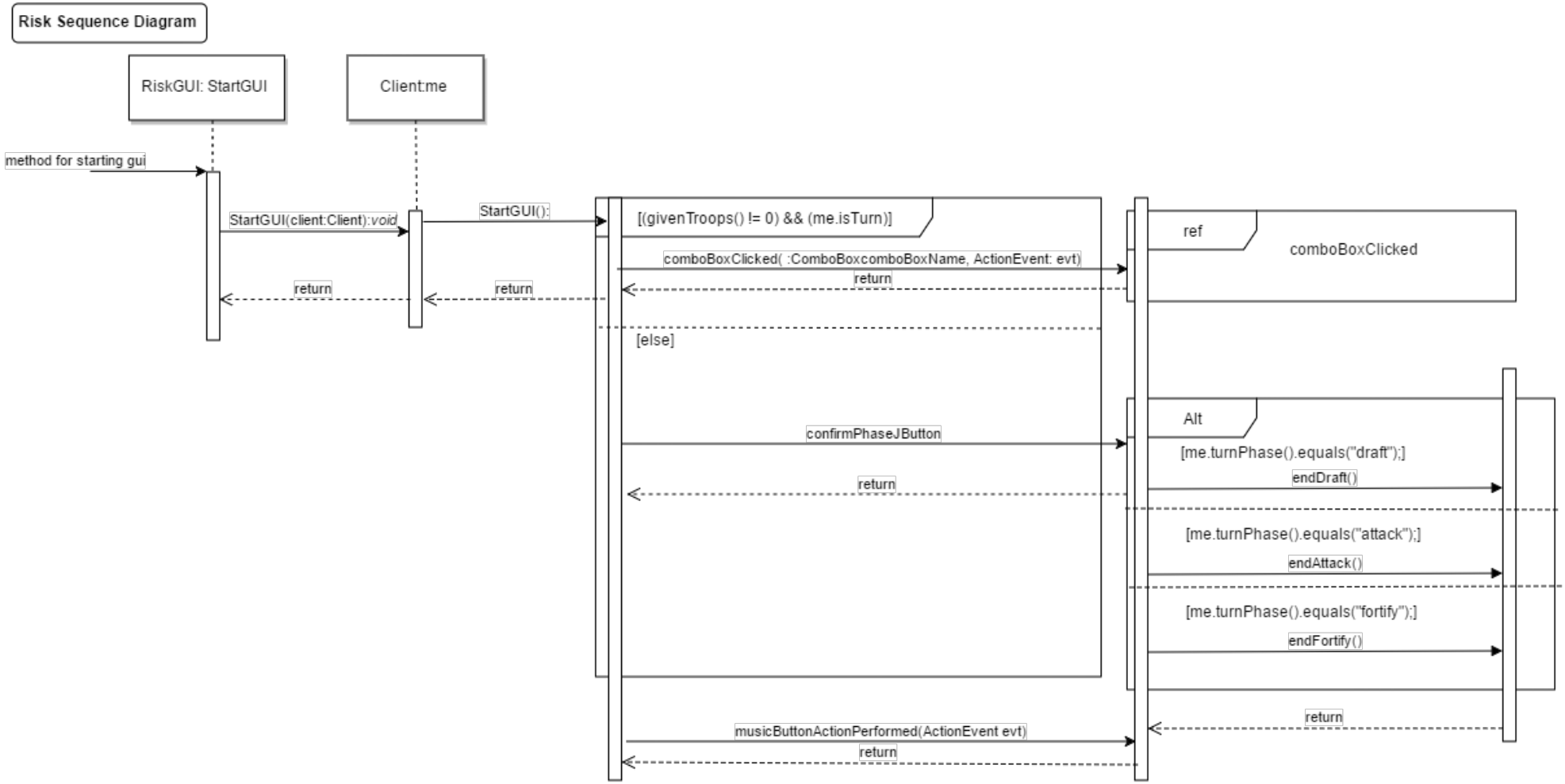
Total Diplomacy > Risk Strategy Guides on Online Risk Games . 2017. Total Diplomacy > Risk Strategy Guides on Online Risk Games . [ONLINE] Available at: http://www.totaldiplomacy.com/Home/tabid/67/articleType/CategoryView/categoryId/51/Online-Risk-Games.aspx. [Accessed 19 March 2017].

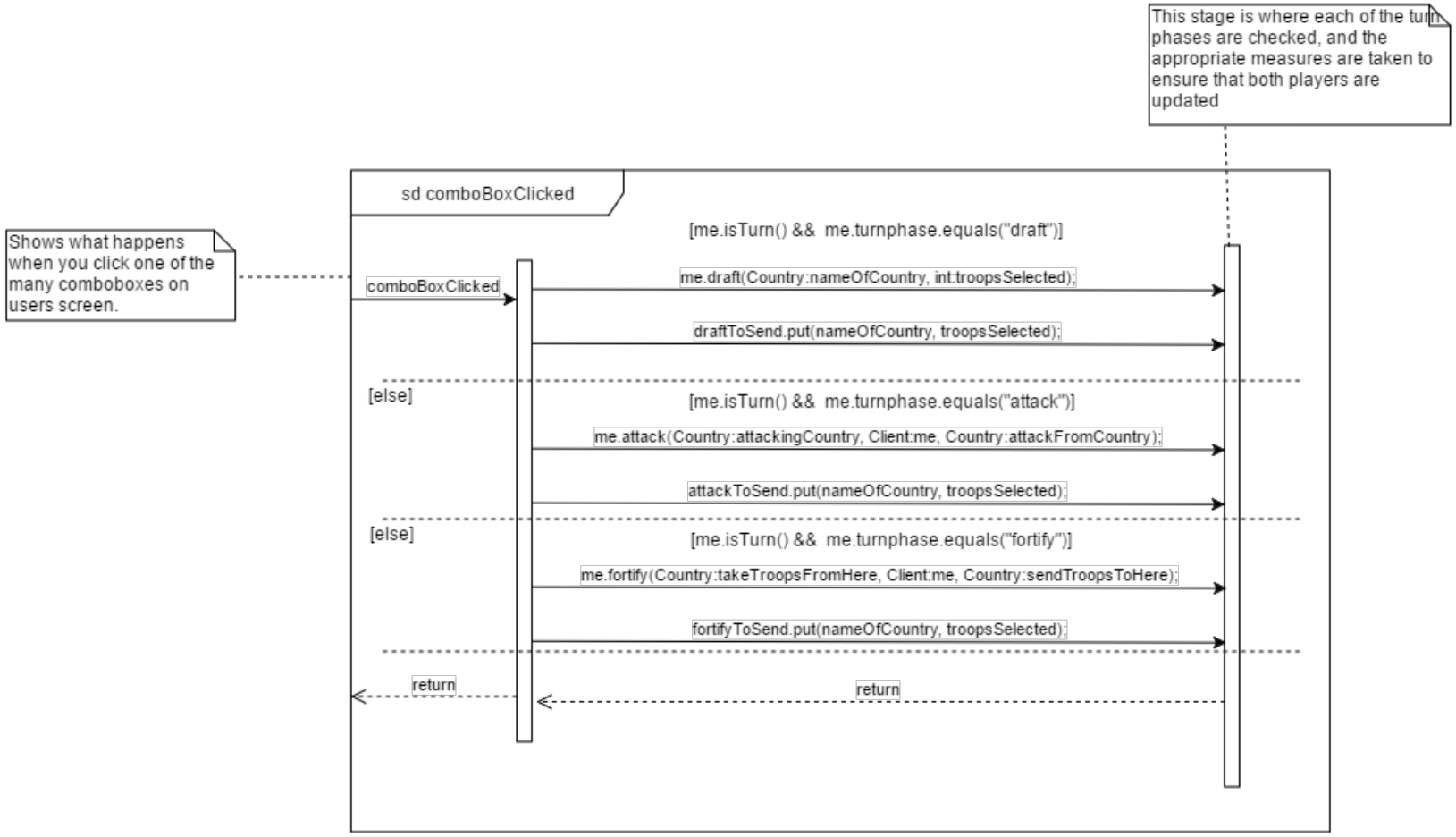
13 APPENDIX

13.1 UML DIAGRAMS

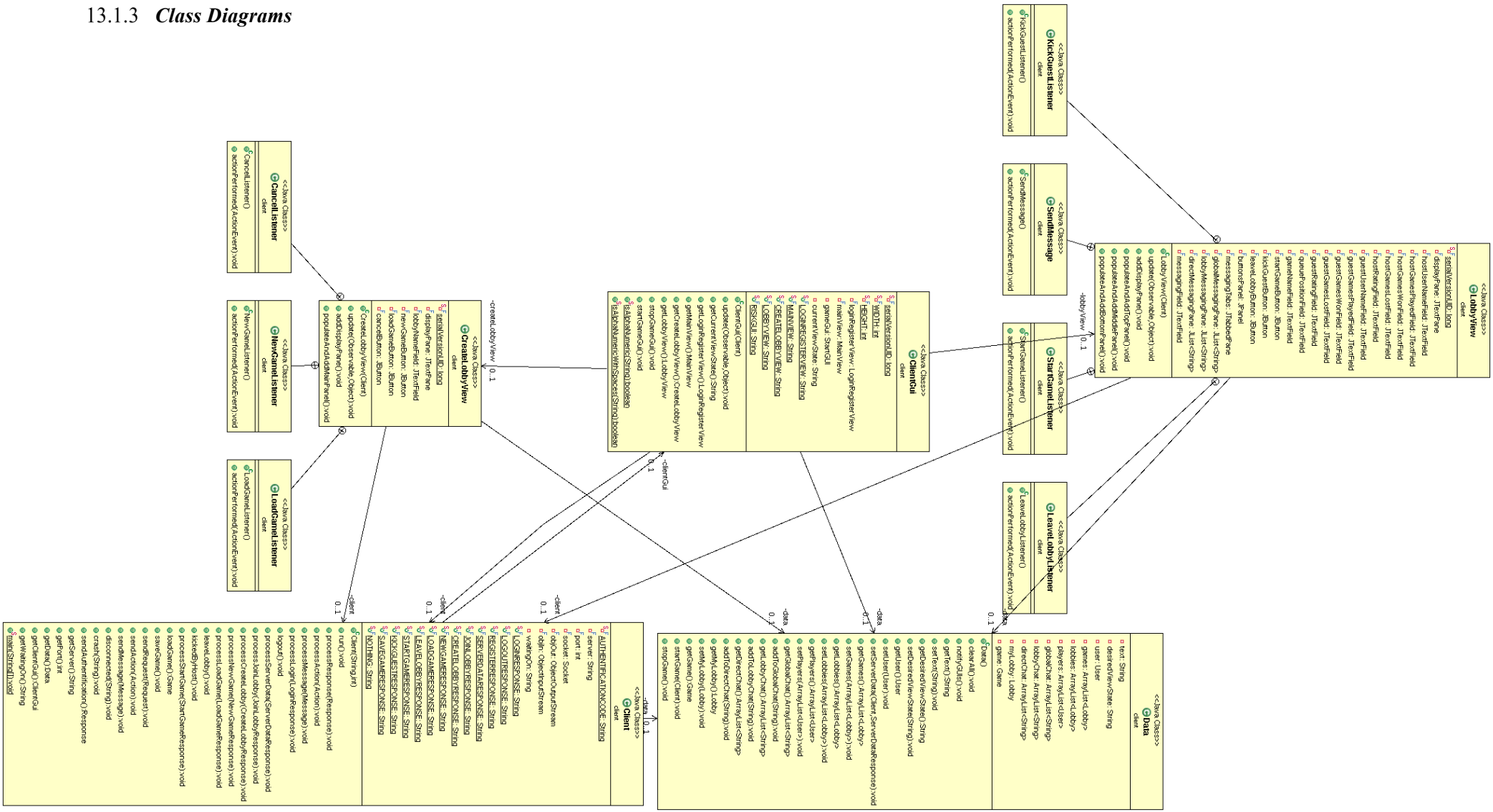
13.1.1 *Use case diagram*

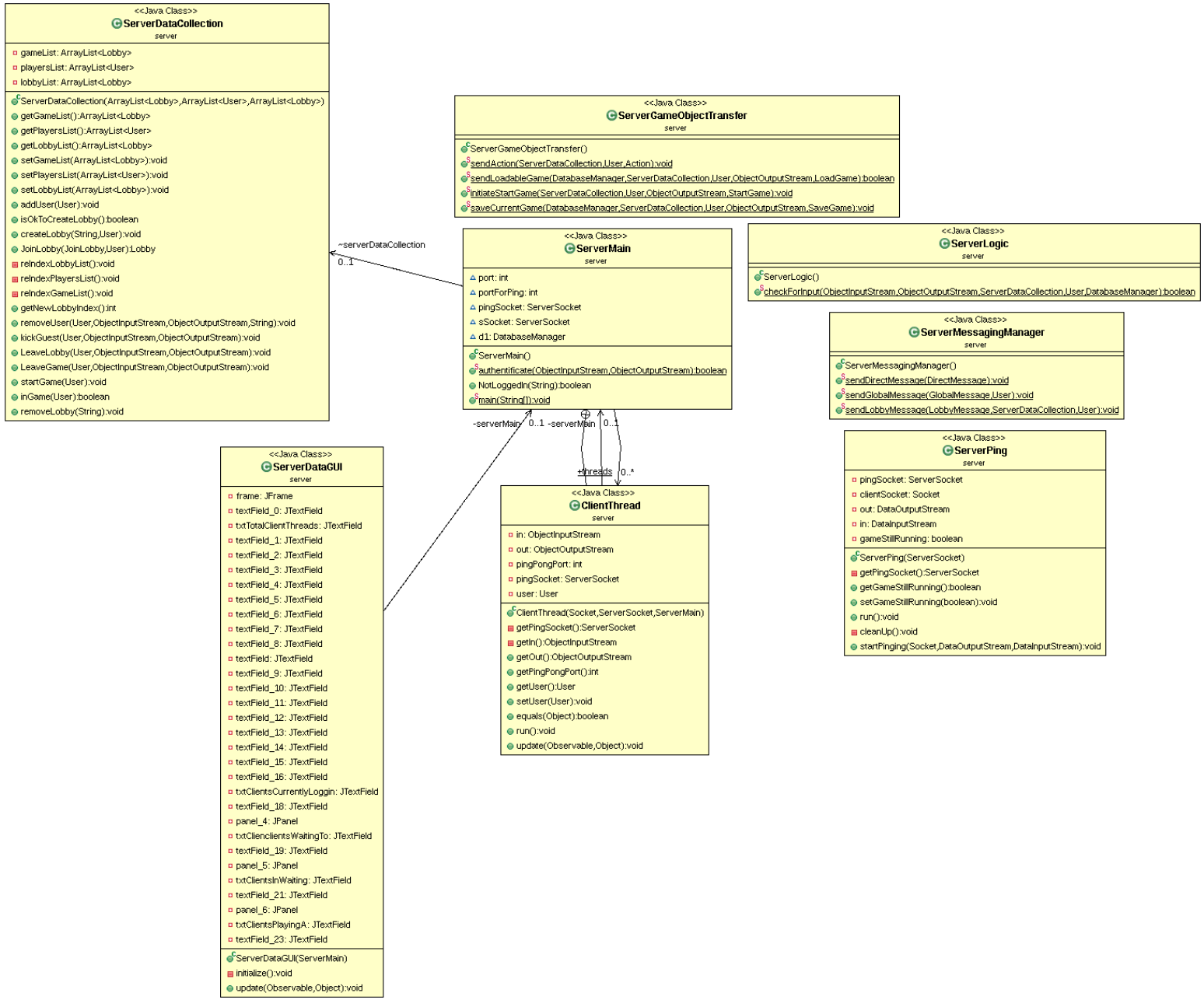
13.1.2 *Sequence diagram*

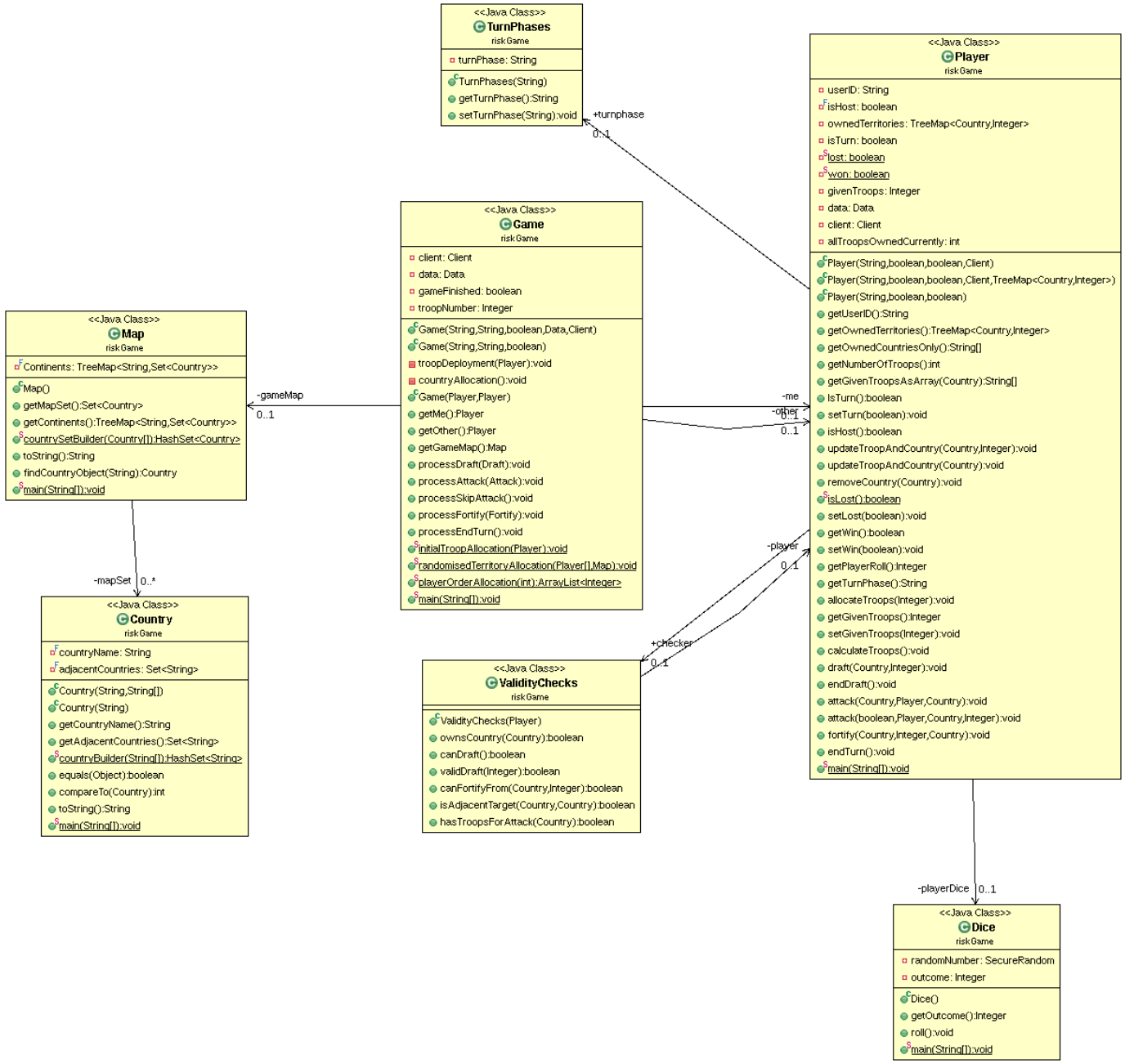



This stage is where each of the turn phases are checked, and the appropriate measures are taken to ensure that both players are updated
sd comboBoxClicked
Shows what happens when you click one of the many comboboxes on users screen.
[me.isTurn() && me.turnphase.equals("draft")]
comboBoxClicked
me.draft(Country:nameOfCountry, int:troopsSelected);
draftToSend.put(nameOfCountry, troopsSelected);
[else]
[me.isTurn() && me.turnphase.equals("attack")]
me.attack(Country:attackingCountry, Client:me, Country:attackFromCountry);
attackToSend.put(nameOfCountry, troopsSelected);
[else]
[me.isTurn() && me.turnphase.equals("fortify")]
me.fortify(Country:takeTroopsFromHere, Client:me, Country:sendTroopsToHere);
fortifyToSend.put(nameOfCountry, troopsSelected);
return
return

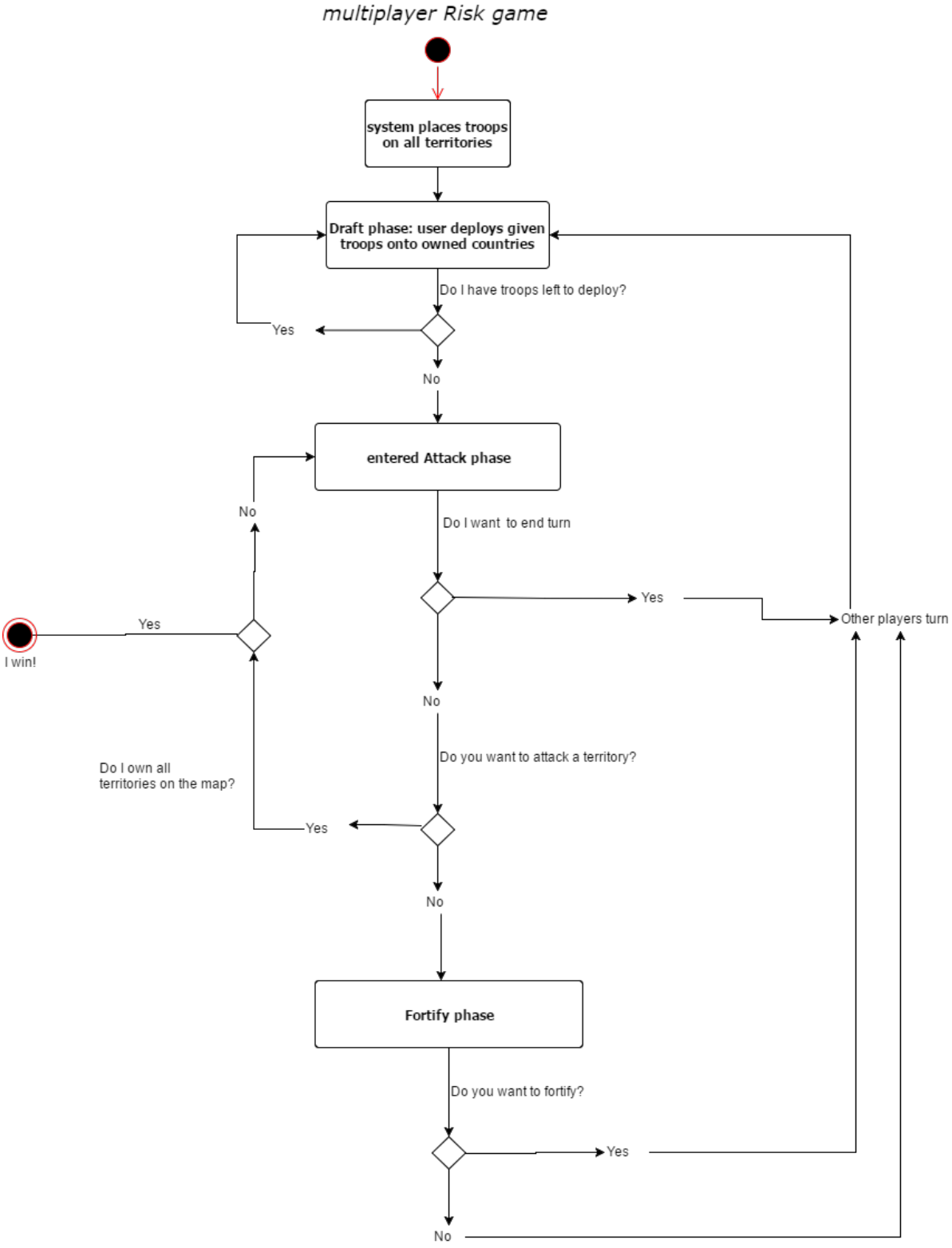
13.1.3 *Class Diagrams*




<<Java Class>>
ServerDataCollection
server
gameList: ArrayList<Lobby>
playersList: ArrayList<User>
lobbyList: ArrayList<Lobby>
ServerDataCollection(ArrayList<Lobby>,ArrayList<User>,ArrayList<Lobby>)
getGameList():ArrayList<Lobby>
getPlayersList():ArrayList<User>
getLobbyList():ArrayList<Lobby>
setGameList(ArrayList<Lobby>):void
setPlayersList(ArrayList<User>):void
setLobbyList(ArrayList<Lobby>):void
addUser(User):void
isOkToCreateLobby():boolean
createLobby(String,User):void
JoinLobby(JoinLobby,User):Lobby
reIndexLobbyList():void
reIndexPlayersList():void
reIndexGameList():void
getNewLobbyIndex():int
removeUser(User,ObjectInputStream,ObjectOutputStream,String):void
kickGuest(User,ObjectInputStream,ObjectOutputStream):void
LeaveLobby(User,ObjectInputStream,ObjectOutputStream):void
LeaveGame(User,ObjectInputStream,ObjectOutputStream):void
startGame(User):void
inGame(User):boolean
removeLobby(String):void
<<Java Class>>
ServerGameObjectTransfer
server
ServerGameObjectTransfer()
sendAction(ServerDataCollection,User,Action):void
sendLoadableGame(DatabaseManager,ServerDataCollection,User,ObjectOutputStream,LoadGame):boolean
initiateStartGame(ServerDataCollection,User,ObjectOutputStream,StartGame):void
saveCurrentGame(DatabaseManager,ServerDataCollection,User,ObjectOutputStream,SaveGame):void
-serverDataCollection
0..1
<<Java Class>>
ServerMain
server
port: int
portForPing: int
pingSocket: ServerSocket
sSocket: ServerSocket
d1: DatabaseManager
ServerMain()
authentificate(ObjectInputStream,ObjectOutputStream):boolean
NotLoggedIn(String):boolean
main(String[]):void
<<Java Class>>
ServerLogic
server
ServerLogic()
checkForInput(ObjectInputStream,ObjectOutputStream,ServerDataCollection,User,DatabaseManager):boolean
<<Java Class>>
ServerMessagingManager
server
ServerMessagingManager()
sendDirectMessage(DirectMessage):void
sendGlobalMessage(GlobalMessage,User):void
sendLobbyMessage(LobbyMessage,ServerDataCollection,User):void
-serverMain
0..1
-serverMain
0..1
+threads
0..*
<<Java Class>>
ServerDataGUI
server
frame: JFrame
textField_0: JTextField
txtTotalClientThreads: JTextField
textField_1: JTextField
textField_2: JTextField
textField_3: JTextField
textField_4: JTextField
textField_5: JTextField
textField_6: JTextField
textField_7: JTextField
textField_8: JTextField
textField: JTextField
textField_9: JTextField
textField_10: JTextField
textField_11: JTextField
textField_12: JTextField
textField_13: JTextField
textField_14: JTextField
textField_15: JTextField
textField_16: JTextField
txtClientsCurrentlyLoggin: JTextField
textField_18: JTextField
panel_4: JPanel
txtClienclientsWaitingTo: JTextField
textField_19: JTextField
panel_5: JPanel
txtClientsInWaiting: JTextField
textField_21: JTextField
panel_6: JPanel
txtClientsPlayingA: JTextField
textField_23: JTextField
ServerDataGUI(ServerMain)
initialize():void
update(Observable,Object):void
<<Java Class>>
ClientThread
server
in: ObjectInputStream
out: ObjectOutputStream
pingPongPort: int
pingSocket: ServerSocket
user: User
ClientThread(Socket,ServerSocket,ServerMain)
getPingSocket():ServerSocket
getIn():ObjectInputStream
getOut():ObjectOutputStream
getPingPongPort():int
getUser():User
setUser(User):void
equals(Object):boolean
run():void
update(Observable,Object):void
<<Java Class>>
ServerPing
server
pingSocket: ServerSocket
clientSocket: Socket
out: DataOutputStream
in: DataInputStream
gameStillRunning: boolean
ServerPing(ServerSocket)
getPingSocket():ServerSocket
getGameStillRunning():boolean
setGameStillRunning(boolean):void
run():void
cleanUp():void
startPinging(Socket,DataOutputStream,DataInputStream):void


<<Java Class>>
TurnPhases
riskGame
turnPhase: String
TurnPhases(String)
getTurnPhase():String
setTurnPhase(String):void
+turnphase
0..1
<<Java Class>>
Player
riskGame
userID: String
isHost: boolean
ownedTerritories: TreeMap<Country,Integer>
isTurn: boolean
lost: boolean
won: boolean
givenTroops: Integer
data: Data
client: Client
allTroopsOwnedCurrently: int
Player(String,boolean,boolean,Client)
Player(String,boolean,boolean,Client,TreeMap<Country,Integer>)
Player(String,boolean,boolean)
getUserID():String
getOwnedTerritories():TreeMap<Country,Integer>
getOwnedCountriesOnly():String[]
getNumberOfTroops():int
getGivenTroopsAsArray(Country):String[]
isTurn():boolean
setTurn(boolean):void
isHost():boolean
updateTroopAndCountry(Country,Integer):void
updateTroopAndCountry(Country):void
removeCountry(Country):void
isLost():boolean
setLost(boolean):void
getWin():boolean
setWin(boolean):void
getPlayerRoll():Integer
getTurnPhase():String
allocateTroops(Integer):void
getGivenTroops():Integer
setGivenTroops(Integer):void
calculateTroops():void
draft(Country,Integer):void
endDraft():void
attack(Country,Player,Country):void
attack(boolean,Player,Country,Integer):void
fortify(Country,Integer,Country):void
endTurn():void
main(String[]):void
<<Java Class>>
Game
riskGame
client: Client
data: Data
gameFinished: boolean
troopNumber: Integer
Game(String,String,boolean,Data,Client)
Game(String,String,boolean)
troopDeployment(Player):void
countryAllocation():void
Game(Player,Player)
getMe():Player
getOther():Player
getGameMap():Map
processDraft(Draft):void
processAttack(Attack):void
processSkipAttack():void
processFortify(Fortify):void
processEndTurn():void
initialTroopAllocation(Player):void
randomisedTerritoryAllocation(Player[],Map):void
playerOrderAllocation(int):ArrayList<Integer>
main(String[]):void
<<Java Class>>
Map
riskGame
Continents: TreeMap<String,Set<Country>>
Map()
getMapSet():Set<Country>
getContinents():TreeMap<String,Set<Country>>
countrySetBuilder(Country[]):HashSet<Country>
toString():String
findCountryObject(String):Country
main(String[]):void
-gameMap
0..1
-me
0..1
-other
0..1
-mapSet
0..*
<<Java Class>>
Country
riskGame
countryName: String
adjacentCountries: Set<String>
Country(String,String[])
Country(String)
getCountryName():String
getAdjacentCountries():Set<String>
countryBuilder(String[]):HashSet<String>
equals(Object):boolean
compareTo(Country):int
toString():String
main(String[]):void
-player
0..1
+checker
0..1
<<Java Class>>
ValidityChecks
riskGame
ValidityChecks(Player)
ownsCountry(Country):boolean
canDraft():boolean
validDraft(Integer):boolean
canFortifyFrom(Country,Integer):boolean
isAdjacentTarget(Country,Country):boolean
hasTroopsForAttack(Country):boolean
-playerDice
0..1
<<Java Class>>
Dice
riskGame
randomNumber: SecureRandom
outcome: Integer
Dice()
getOutcome():Integer
roll():void
main(String[]):void

13.1.4 ***Activity Diagram***



13.2 GUI APPENDIX

(Liszewski, 2017)

(Flickr, 2017)

(Tom and Alex, 2017)

(Total Diplomacy, 2017)

(FunkyPotato, 2017)

(flaregames, 2017)