CS 319
Object-Oriented Software Engineering
Design Report

RISK
Group 1-G

Adeem Adil Khatri	21801174
Abdulmalak Albeik	21801277
Mannan Abdul	21801066
Maryam Shahid	21801344
Osama Tanveer	21801147
Yahya Mahmoud Ahmed Elnouby Mohamed	21801332

Contents

1. Introduction

1.1. Purpose of Risk

Risk is designed to bring the classic Risk board game to a digital platform. Although the gameplay of Risk is identical to its real world counterpart, digitalizing the game will give room to new and more interesting features.

Risk is an online multiplayer game which can be played by two to five people. The game is compatible on all computers and its only restriction is having a working internet connection. It will consist of several maps that the player will be allowed to choose from, including the original map of the Risk board game. The player who conquers all territories will be deemed the winner. The user will be allowed to save and load previous games.

1.2. Design Goals

1.2.1. Usability

The user interface will consist of four main screens: main menu, log-in, game settings and gameplay. The screens will consist of entertaining and appealing elements (such as background image etc.) to make the game more engaging. Other than the gameplay, the screens will have limited elements (maximum of 2-3 buttons per screen) to aid user input and create a user-friendly environment for non-tech users. All the screens will have clear buttons to navigate back and forth.

The game screen will inevitably consist of certain complex features to mimic the board game. However certain usability strategies will be used to overcome complexities. A status bar at the top will keep track of the current troops of each player. A score board will contain the number of territories occupied by each player. A tutorial will be given in case a first-time user accesses the game. Lastly, the components that are not required at the given stage of the game will remain hidden such as the number of troops to deploy will not be visible during a battle or while throwing dice, the number of troops of other territories will not

be visible during a battle between two. All this will add to the usability of the game.

1.2.2. Robustness

The program will not act upon invalid inputs such as trying to perform in another player's turn or trying to occupy territories they have not yet reached. In case the user tries to do so, an error message will be displayed and the user will be notified why the requested action is invalid.

Extensive and thorough testing of the program will be done before it is made available for users to verify that it has a solid stance.

1.2.3. Extensibility

As updates and new features are an integral part of a program, the development of the game will proceed in a way that will ease addition of new functionalities. Our source code will make use of strategy and decorator design patterns. This will ensure that in case we need to extend an existing class or function, we will not have to bring major changes to the code and the addition will be implemented independently. New changes can consist of more map options, an advanced algorithm to roll dice etc.

1.2.4. Performance

Since game performance is one of the most important elements in terms of a game's success (as game performance reflects on the user's experience), we focused on optimizing performance and efficiency of the game. We achieved this by minimizing response time of each action taken by the player to less than 1 second. The time taken for other players to see the given player's action is also limited to less than one second. Likewise, each player's action will be reflected in the status bar at the top of the screen which will be updated within 1 second of an action's occurrence.

Since the game is online, users will be allowed to play the game with a minimum internet speed of 5 Mbps. This will ensure constant connectivity throughout the game.

2. High-level software architecture

2.1. Subsystem decomposition

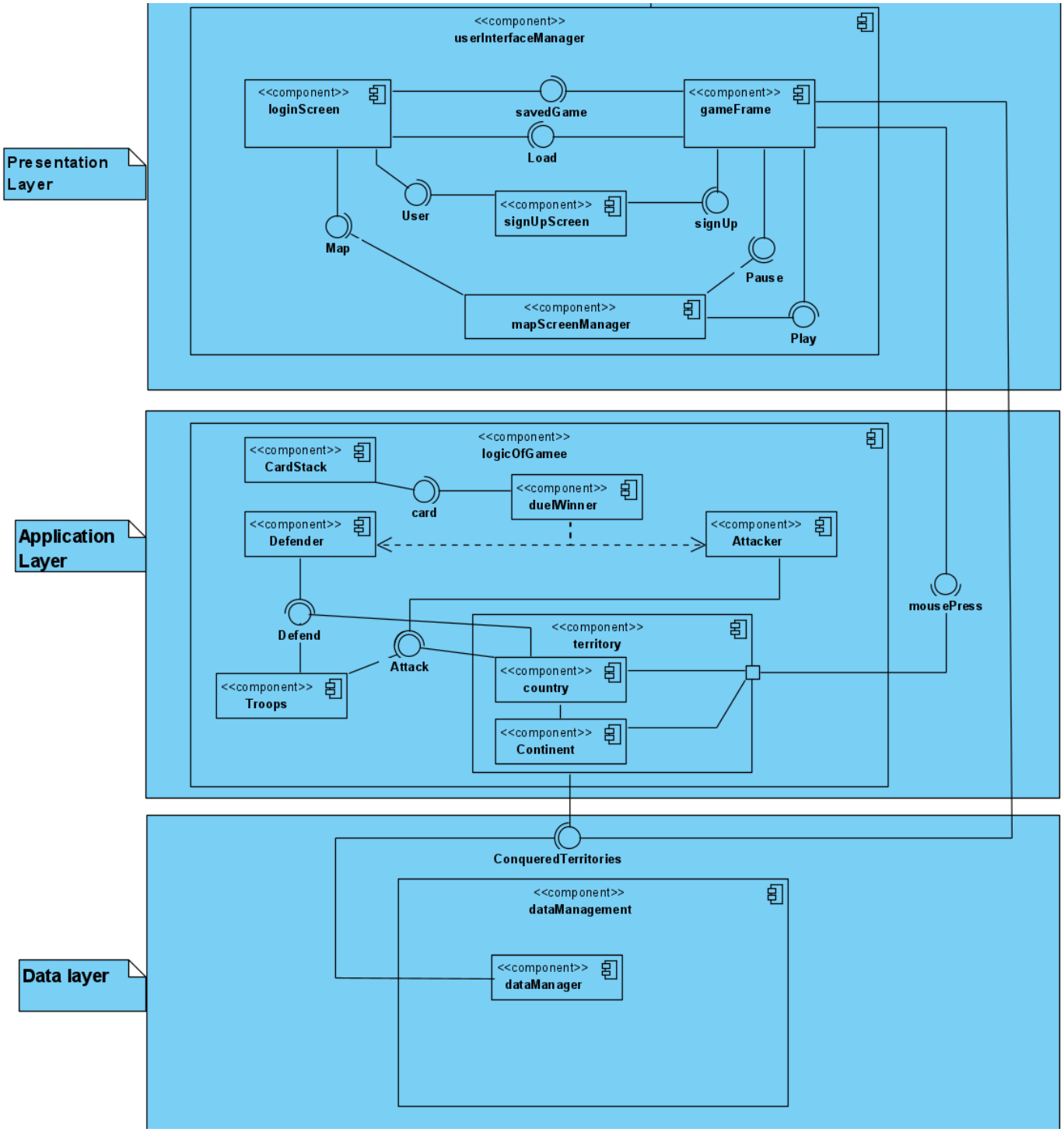

Figure 1 - Subsystem Decomposition Components UML Diagram

Our aim is to decompose our system into packages or subsystems, reducing dependencies between the subsystems as much as possible (high cohesion and less coupling). Focusing on removal of dependencies, to modify the subsystems makes the implementation much easier as there exists less coupling and high cohesion in the effort.

We have decided to use a 3-Layer Architectural style, which is a 3-tier architecture, for the design pattern of the web-based version of the board game Risk. As seen above in the figure(Components Diagram of the subsystems), our system has been divided by the rules of the 3-layer system design pattern.

This type of system design pattern is favorable to our system because:

I. The 3 layers are defined as:
    A. In our system, the *presentation layer* is the **userInterfaceManager** subsystem, and this contains the components for the player's UI.
    B. In our system, the *application layer* is the **logicOfGame** subsystem, where the components for game logic are used to manage the entity objects of the game.
    C. In our system, the *data layer* is the **dataManager** subsystem, where the components for managing data such as troops, territories (conquered/ unconquered) are stored.

II. The associations between subsystems and components can be represented easily via this system design pattern.

III. The 3-Layer architectural system design focuses on increasing the efficiency of the system/program via dividing up the system into hierarchy of 3 layers, that are independent and can be linked with each other either opaquely or transparently (depending upon the design goal - maintainability, flexibility, runtime efficiency). Such an architecture makes the whole procedure easier to commit to changes and quicker.

In our architectural style layout, there exists transparent layering, i.e open architecture, which defines that each layer can call operations from any layer

below it. Hence, this will allow us to increase performance, development speed and scalability of the application.

Moreover, each layer is explained in detailed as follows:

- In the presentation layer the **userInterfaceManager** conveys with the logicOfGame component, which detects the user's input when mouse key is pressed(using mouse action listeners) and the data (mouse press) is transferred to territory component where it will perform the desired action of the user(either defend or attack). Via the game frame, the userInterface manager transmits the conqueredTerritories , and to the dataManager component in the DataManagement subsystem where the player's conquered territories are updated. As the player every time saves or exits the game, the event of saving conquered territories and the data is first stored in the database then the game gets shut down.
- In the application layer the **logicOfGame** subsystem receives the mousePress event, subsequently this data is used to compute the movement made by the player such as attack or defend of troops on neighbouring territories. The Defender and Attacker require Troops from Defend and Attack interfaces respectively, also the duelWinner component distinguishes between whether the Defender or Attacker has won the duel/conflict event between 2 players. The CardStack component provides the card which is required by the duelWinner component to award it to the winner after the duel/conflict event.
- After the user selects to quit the game, the conqueredTerritories from the presentation layer are received by the **DataManagement** subsystem in the data layer. Moreover, the dataManager component serves to save the game session in the user profile to our database in the dataMangement subsystem so it does not correspond well with the userInterfaceManager or logicOfGame layers. Also, the 3-tier architecture system lines up with layers styling used in our system since the layers do not access any functions of the userInterfaceManager or logicOfGame layers.

- ❖ The external frameworks which are going to be used are Google FireBase and ReactJS for now, but it may be prompt to future change. As use of firebase realtime is also an option on the horizons.

2.2. Hardware/Software mapping

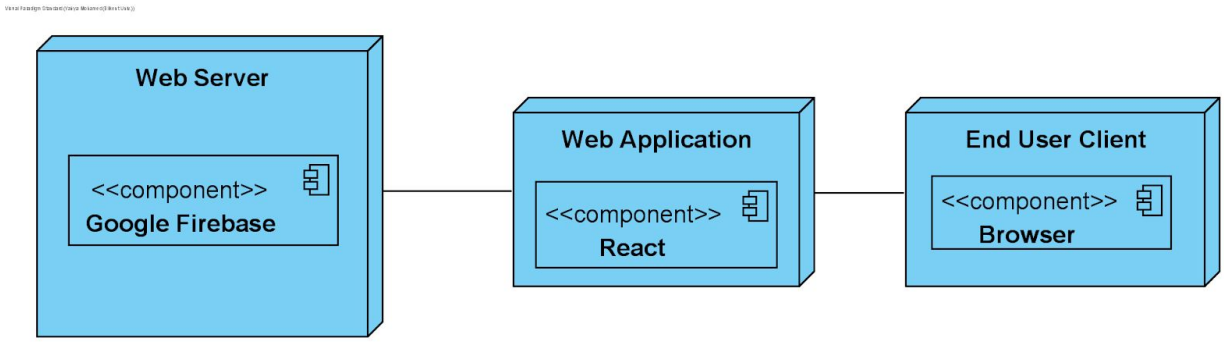

Figure 2 - Deployment Diagram

The version of our risk game will be an online version. The user interface will be developed in the frontend using react.js and the game logic will be implemented using javascript and controlling data management for saving the game and getting saved games, logging in and signing up will be maintained using Google Firebase. The game will be hosted on Amazon Web Services. Considering the software and hardware requirements, the game will be compatible with supported OSes as it's going to be an online game.

2.3. Persistent data management

The saved games for a user will be saved in the database. If, however, while saving the data a network failure or data corruption occurs, database rollback would be used to reset the data to the original state and the user would be notified that the game has not been saved. The game's data would be handled using interactions of JSON objects. It would be ensured that the object is not reassigned by using constants where possible. Since JavaScript is a very flexible language, it may not be possible to keep track of the properties of the objects. The JSON

objects will store the player's information, player's current state, countries' troops, player's colors etc..

2.4. Access control and security

The game can have anywhere from 3 – 5 players all of whom play on the same computer. Each user needs to sign up first before they can start playing and in case they already have an account, they just have to sign in. When players sign up for the first time, they are issued a unique ID using the UUIDv4 library. All save game data is stored at the back-end database that is hosted on Amazon Web Services (which is why internet connection is a must) to prevent player tampering. We also make API calls to the back-end using the player's unique ID to only get the needed data instead of all the data stored at the back-end to filter the data at the back-end instead of the front-end. Authentication will be done on each API call using either tokens or cookies depending on which is easy to implement and is secure, this will help prevent unnecessary security breaches. Since we have a React based front-end, only the data needed is rendered at the front-end. Since the players are supposed to play on the same computer, it is unlikely that data at the front-end can be manipulated using memory scanners i.e. cheat engines due to players monitoring each other which closely resembles a real-life risk board game. Man in the middle attacks are also possible when the data is requested from the back-end to the front-end and we use https when making API calls, which in turn uses an SSH line that sends the data but encrypted.

2.5. Boundary conditions

2.5.1 Initialization

We initialize the game by typing in the URL we have hosted the React project on, after which all the React UI components and the javascript file that has the game logic will be loaded to the browser on the local system. At first the browser

renders the MainMenu React component to display the main menu. The game is ready to play after the initial render is complete.

In order for each user to play, they have to sign up or sign in. They will then be redirected to a verify credentials page. For each of the players they will be asked to input their username and password which will then be verified at the back-end database which as mentioned before will be hosted on Amazon Web Services. After these player details are verified, the game can be started using the start game button which will then cause the Root React component to render and the game can start.

2.5.2 Termination

The user can exit the game and go back to the main menu using the “Exit” button provided both during and after the game. When the user does this after a game ends, the game will save and update their data on the back-end. If the user exits the game before it ends, the game will prompt the user notifying them that the saved game data will not be updated and if they still wish to continue.

2.5.3 Exceptions

Below, we list some possible failures that are most likely with our implementation of the game.

2.5.3.1 Loss of Network Connection and/or Server Issues

In case the user experiences a loss of a network or if the back-end hosting server goes down, the game will pause at this point until the server goes back up again or if the network is connected again. The user may also exit the game at this stage to return to the main menu. If a network error occurs once data has been modified in the database such that the user is not notified about those changes, then we perform a database roll back to restore the database to its previous state and notify the user that the action they requested was not completed.

2.5.3.2 User Authentication Failure

User authentication failures will occur when a user provides credentials that already exist while signing up or they provide inaccurate credentials when signing in to the game. These credentials will be checked against those stored at the back-end on AWS and in case they are wrong, the user will be notified about it and will not be signed in.

2.5.3.3 UI Elements Update Failure

A network error may also cause the game to lose connection with the server where UI elements such as images, sound files etc are stored, and hence they will fail to update if the components on the screen need to change. In this case, the game will pause and keep refreshing until the UI elements load.

2.5.3.4 Library Failure

In this scenario, React will fail to render or update a UI component, but the game logic might still be updated. This means that even though the player stats are updated, the UI does not represent such a change. This failure cannot be handled as this is a failure due to React.js.

2.5.3.5 IO Failure

The failure of mouse and keyboard can cause the user inputs to not be recorded or a corrupted input to be recorded. The best solution here is to validate each input before passing its value to the game logic. This way it can be ensured that no extraneous data is not passed to the game logic.

2.5.3.6 Process Termination

This can occur when the browser is closed from the Task Manager. This can result in the loss of game progress. The user can be prompted of browser closure

by using JavaScript's alert method. However, if the browser crashes, there is nothing that can be done in this scenario.

2.5.4.4 Memory Not Available

Since the UI components will be built by using latest React technologies, it is better to have sufficient system memory in order to run the game smoothly in the browser. If, however, the system does not provide sufficient memory to the browser, the game cannot be played smoothly. There is nothing that can be done to address this issue, except for making the major logic executions very efficient, which will be ensured.

3. Low-level design

3.1. Object design trade-offs

3.1.1 Security vs Performance

The game logic will be implemented in the frontend instead of the backend. Although this poses some security problems, i.e. the game is susceptible to cheating through memory scanners such as Cheat Engine, security is not our major concern. Since the players will be sitting together while playing the game, it is highly doubtful for 1 player to cheat using such tools if they are sitting with one another. Implementation of game logic in the backend would require API calls that, depending on the network connectivity of the main player, pose latency issues. The objective of the game is to remove any possible sources of delays. Storing the game state for a constant time and making API calls after this time has elapsed can be a viable solution, but this still poses other problems e.g. if the user loses connectivity seconds before the API call is made. Therefore, in this scenario it is plausible to compromise on security to achieve better game performance.

3.1.2 Portability vs Performance

The game's performance can be upgraded by multiple folds if it was being

written in C++. Instead, the game will be implemented in JavaScript so that it is easily accessible by anyone with a browser on any machine. One of the implementation's goals is for the game to be playable on any screen, including mobile phones. Therefore, even though implementation using C++ would provide us with better performance, it would not allow as much portability as running the game in the browser allows. The only requirement of our implementation would be a device that has a browser capable of running JavaScript. The performance of the application would be slightly compromised but it would not make much of a difference because it would be unnoticeable for the players.

3.1.3 Functionality vs Understandability

The game will have a limited number of features which will allow the players to easily navigate through it. The players would only use the mouse, the escape and enter key. The mouse would be used to perform major game functions e.g. deploy troops, choose attacking area, trade cards etc., the escape key to pause, the enter key only in case of form submission where an input from a player is required. This would increase the user interaction with the game and the players would develop an intuition with the game just after one game. No extra effort to memorize special functionalities is required so that the players do not have to know anything except for game logic. The extra features, e.g. login, saving games, are self-explanatory and the user would not have to do any extra effort to know what these functionalities accomplish.

3.1.4 Readability vs Performance

Even though it is better to implement the whole game as React components, the game would be implemented in Vanilla JavaScript and React would only be used to update the user interface. Implementing the game in React completely would mean that each component takes care of its rendering itself into the DOM. Since we will be separating the game logic from the UI, it would mean that the game logic would have to be updated separately from the

React part of the application. Although this might pose a short delay resulting in a performance hit, it would not affect the user experience since the delay would be very small. The code readability would be significantly improved because in the future a bug is discovered, it would be easy to navigate through the code to identify if the bug is in game logic or the UI. In addition to that, in the future if changes are to be made to the game logic, it would be very easy to simply make them in the Vanilla JavaScript section of the code.

3.2. Final object design

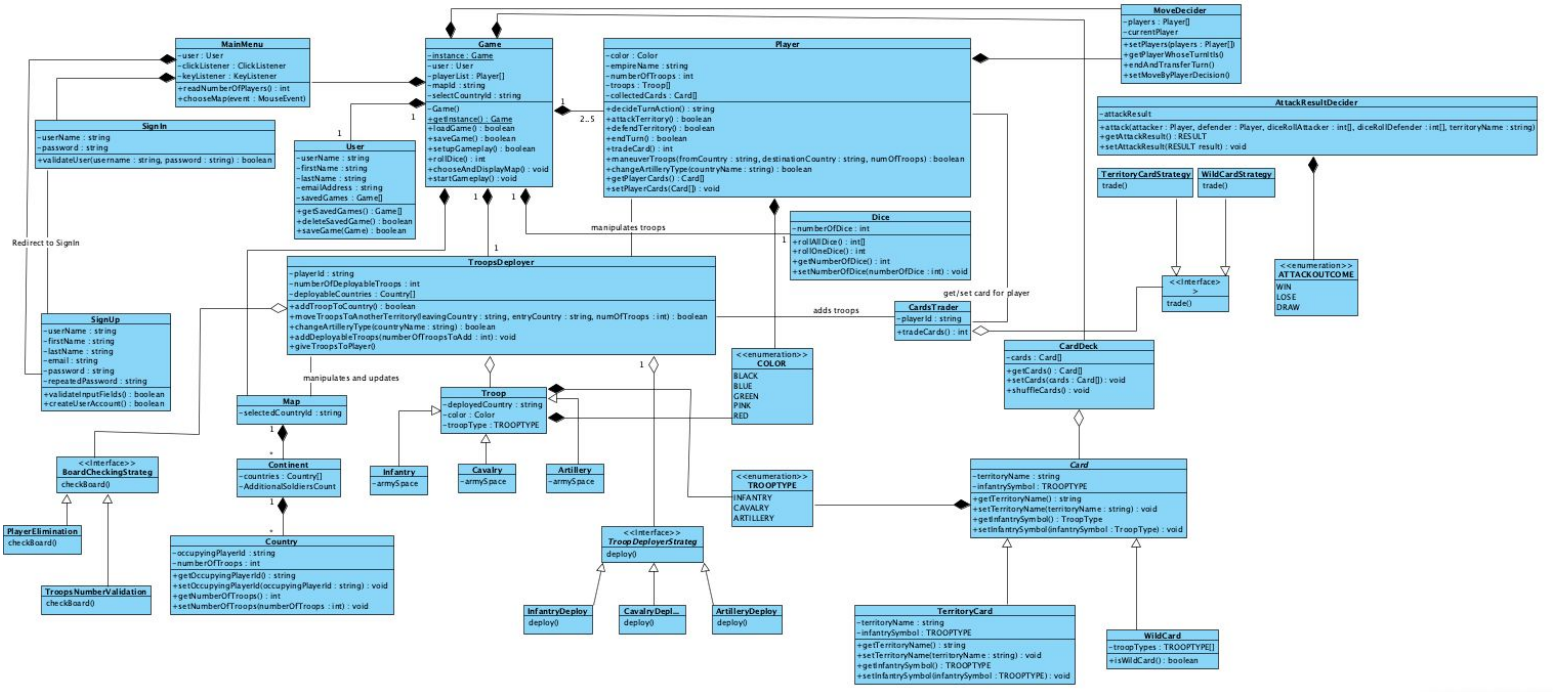

Figure 3 - Object Design Diagram

3.2.1. **Game Class:** This class is the main class controlling the program. It will get initialized when the game starts and it will let the player start a new game. It will be responsible for the gameplay rolling the dice, attacking a country, deploying troops in a country, saving the game, and loading a new game.

3.2.2. **TroopsDeployer Class:** This class is responsible for deploying n number of troops to a country for a player, and can also move troops from one country to another.

3.2.3. **Dice Class:** This class represents the dice that will be rolled to decide the outcome of a round. It can be rolled and it will generate a random dice value.

3.2.4. **Troops Class:** The generic troop object is a single fighting unit that will be deployed in a country to fight for a player.

3.2.5. **MainMenu Class:** This class is responsible for getting the initial user feedback on what kind of game they want to play. It also serves as the initial place for the user to signIn or signUp.

3.2.6. **SingIn Class:** This class is responsible for sending the sign in data to the server and authenticating the user.

3.2.7. **Signup Class:** This class is responsible for getting the sign up data from the user and validating it and then sending it to the server to be saved.

3.2.8. **Map Class:** This class houses all the countries and the continents of the game.

3.2.9. **Continent Class:** This class is the model for continents that have relations to countries.

3.2.10. **Country Class:** This class will store information related to the country and who is occupying it at that moment and how much troops are in there, also it is responsible for changing the number of troops in the country or the occupying player.

3.2.11. **User Class:** This is the main class responsible for user management. It will store the user information when signed in.

3.2.12. **Player class:** This class houses the game logic and the AI players' logic. It can trade cards, play turns, attack territories, defend its territories, and everything else that is related to the game logic.

3.2.13. **CardDeck Class:** This class stores all the cards and is responsible for shuffling the cards in the deck and getting a random card out of the deck.

3.2.14. **Card Abstract Class:** This class represents the generic Risk territory cards. It has a territoryName and an InfantrySymbol to identify the card. It is an abstract class and it has two possible subclasses - TerritoryCard and WildCard.

3.2.15. **TerritoryCard Class**: The class that corresponds to the simple risk territory card. It has a territoryName and an infantrySimple.

3.2.16. **WildCard Class**: The class that corresponds to the wild card in the risk game. It has an array of TROOPTYPES indicative of the troop types on its face.

3.2.17. **CardTrader Class:** This class is responsible for exchanging cards for troops in the game.

3.2.18. **MoveDecider Class:** The class that helps decide the player whose move it is in the game. It keeps track of the players using an array and the current player whose turn it is using the player's id.

3.2.19. **AttackResultDecider Class**: This class keeps track of an attack during each attack in the game. It stores the attackResult and has methods that perform the attacks. This class will have its attack method called from the player class in order to attack another territory.

3.2.20. **TradeCard Interface**: The interface that must be implemented by the CardsTrader class in order to set the deployment strategy.

3.2.21. **TerritoryCardStrategy Class**: The class corresponding to the algorithm when trading a territory card.

3.2.22. **WildCardStrategy Class**: The class corresponding to the algorithm when trading a wild card.

3.2.23. **ATTACKOUTCOME Enumeration**: The ways an attack can end. This is added to keep synchronization of attack result state throughout the game.

3.2.24. **TOOPTYPE Enumeration**: The types of troops in the game. This is added so that the troop types are consistent throughout the game.

3.2.25. **COLOR Enumeration**: The colors of the players during the game.

3.2.26. **BoardCheckingStrategy Interface**: This interface should be implemented by the TroopsDeployer class. The TroopsDeployer class will use PlayerElimination and/or TroopsNumberValidation to check the board throughout the game depending on the situation.

3.2.27. **PlayerElimination Class**: A class that checks if any player needs to be eliminated or not.

3.2.28. **TroopsNumberValidation Class**: A class that checks if the number of troops as deployed by user are consistent with the troops on the board.

3.3. Packages:

We will be having three main packages:

1. Game UI Package: responsible for the user interface and all the interactions between the user and the game.
2. Game Logic Package: responsible for the game logic and controls what the game UI should display.
3. Game Server Package: responsible for storage of data that will be used by the game logic and game UI packages.

Having these packages will allow us to have Unit Tests for each package individually, it will also allow us to use the View-Controller-Model design pattern.

3.4. Design Patterns:

3.4.1 Strategy Pattern

There are some classes that do the same task, but use different ways to accomplish this task. There are 2 types of Risk Cards that can be traded – wild cards and territory cards. Both tradings involve accepting some cards from users and giving them troops, but different combinations of cards can be accepted for both, i.e. the algorithms are different. To allow code reuse and to prevent duplication, the Strategy Pattern is used. The strategy pattern would allow the change of trading card strategy at runtime for the CardsTrader. Moreover, since the algorithms are separate, in the future if a new way to trade risk cards comes up, for instance trading mission cards, it would involve changing very few areas of code.

The second area strategy pattern is used in the TroopsDeployer. The deployment of troops accomplishes the same results. For example, the deployment of infantry, cavalry, and artillery accomplish the same thing, i.e. deploy troops to a country. However, the algorithm for deploying troops is different because they have different army spaces. Strategy pattern is used here to prevent code duplication and allow reusability. Again, the troops deploying strategy can be changed at runtime. In the future, if a new type of troop is added, its strategy can simply be added by creating a class using the TroopsDeployerStrategy interface.

The strategy pattern would also be used to check the board. The 2 cases involving the board checking are – board validation and eliminating a player. Board validation is needed to check throughout the game that the troops deployed correspond to the troops that the user has actually deployed and there are no errors. The elimination of a player check involves checking the board to see if a player has all his troops eliminated. Both are similar but the algorithm is different.

Therefore, a strategy pattern is used so that a board checking strategy can be set at runtime.

3.4.2 Singleton Pattern

The singleton pattern is used in the game object. At any given time, we can never have more than 1 game object. It does not make sense to have multiple game objects each handling their own games. In the future, this behavior cannot be expected in the game as well. One screen should allow only 1 game to be played at any given time. This should be handled by 1 game object. The singleton pattern allows the accomplishment of this behavior. It allows the creation of only 1 instance of the Game class. The use of the singleton pattern ensures that developer errors are prevented and every time the same instance of the Game class is used.

3.4.3 Model-View-Controller Pattern

The application is built using the MVC pattern. The model is the data that the game is operating on. This is the game's state data for example the players information, the troops that the countries hold. The model also contains the saved games, the user's details. In other words, the model is the data the game is supposed to have. The view is the React part of the application which takes care of the game that is displayed to the players. This will include the user interface and it has nothing to do with the game logic except the updates from the model. The controller is the logic of the game's implementation. This is part of the code that will manipulate the model using the game logic algorithms.

This design pattern was used to separate concerns. The data part of the application is separated and placed in the model, the game logic in the controller, and the user interface in the view. In the future, this will help find bugs easily without having to go through a monolithic codebase. It is only through the interactions of these 3 parts of the codebase that the game operates. The controller (game logic) manipulates the model (game data), the model updates the view of the application. The players would be interacting with the model, which in turn instantiates the sequence of events as mentioned above.

4. Improvement Summary

There were some inconsistencies in the previous version of the report, for instance the storing of game data. These inconsistencies have been addressed in this report. Moreover, some details have been added to the class diagrams. Singleton and strategy design patterns have been added to improve the codebase structure. Previously, there was only one design pattern – the model-view-controller. More design patterns were added as permitted by the natural flow of the game. These design patterns help separate common functionality. Enumerations, such as COLOR, TROOPTYPES, ATTACKOUTCOME, have been added so that they are consistent throughout the codebase and errors resulting from inconsistencies can be prevented. Moreover, Card class has been made an abstract class so that multiple cards can extend this class for common functionality. Originally the Game class was supposed to implement the logic to decide moves and decide the outcomes of an attack. This functionality was separated into objects of their own named MoveDecider and AttackResultDecider respectively. The goal was to have the Game object as a container for all other objects. This provided an opportunity to implement a codebase that has decoupled functionality and with each object having its own purpose.