Bilkent University

Department of Computer Engineering

CS319 Term Project

RISK

Design Report

Group 2I

Ahmet Kaan Uğuralp – 21803844

Selcen Kaya – 21801731

Alperen Can – 21601740

Şükrü Can Erçoban – 21601437

Selin Kırmacı – 21802177

Instructor: Eray Tüzün

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

Table of Contents

1. Introduction

1.1 Purpose of the system

Risk is originally a board game that is implemented in a computer game. Game offers a competitive but friendly environment where the player that has the best strategy and luck wins. With its aesthetically pleasing interface and challenging rules, the player is guaranteed to have a good time regardless of the end result. Players all get territories at the beginning of the game and their purpose is to conquer every territory and become the overall ruler of the world. Gameoverall aims to have a smooth gameplay meaning a high performance and aesthetic user interface where users can also feel the atmosphere of the war.

1.2 Design goals

Our main design goals are to have an easy to understand user interface with fast gameplay to maximize the experience for the user. Following trade-offs show what we decided to sacrifice to achieve these goals.

1.2.1 End User Criteria

Usability:

Our main purpose for this game is for the user to have an easy and enjoyable experience while playing the game without the frustration of figuring out the controls and rules. Our simplistic design for the user interface is understandable for every user whether they are advanced gamers or not. How to play page has all the information of the game with divided categories so players won't get bored reading through it. Game contains not many functions to keep the design very minimal. This way the system offers access to any function of the game in any point of the game. With the sound settings the player can arrange the volume for their comfort.

Performance:

Since our game is a war game based on good strategies, performance of the game is very important. We want players to experience the war as much as possible and since a slow game would take away from that experience, we paid extra time for the performance to make the game breathtaking.

1.2.2 Maintenance Criteria

Extensibility:

We have extended the original game by adding some new features like new cards, new attack strategies and alliance options to make the game more interesting, fun and hard.

Modifiability:

Our game is a multiplayer game. The way it's structured allows us to modify all players features very easily. The game also is available for further modifications like playing against a computer or adding more features since the way it is implemented is not very complex.

Portability:

We implemented our game using Java for it to be able to be played on many devices so it encourages many players to play the game.

Readability:

Since our game does not involve complex functions and is implemented in Java it is very readable for outsiders. This allows other engineers to fix the bugs or add new features to the game very easily which is a good thing for the modifiability of the game.

1.2.3 Performance Criteria

Response time:

As mentioned before, to create the atmosphere of the war game needs to respond to the inputs very quickly to run any animation or play and sound effect to keep the flow of the game fast.

1.2.4 Trade-Offs

Space vs. Speed:

Since our game is a strategic war game with multiple players, its main purpose is to have fast gameplay so we preferred to increase the usage of the memory rather than decreasing the speed of the game.

Functionality vs. Usability:

Our game has basic rules and game methods which are not so complex meaning it offers a better interface and usability by putting the functionality behind usability.

Security vs Usability:

Since our game does not hold sensitive or private data, security is not a priority of the system meaning it is sacrificable for a better usability which is one of the main purposes of the game.

Understandability vs Functionality:

As mentioned in the functionality vs. usability our game does not involve complex functions but rather it pays more attention to the user's experience of the game. So the system makes sure that the user understands the game for the user to have a better time playing the game. We have how to play option in our main menu and we also offer the player the option to pause the game and go back to how to play page while playing the game.

Development Time vs Performance:

In order to reduce the development time we have decided to use Java as our language since all the group members were more comfortable with that language. This cost us some performance issues since C++ offers better solutions in some cases, but this decision helped us have a better coding experience with much less problems and faster process.

Delivery time vs. Quality:

Originally the user interface was planned to have custom artwork, music and sound effects but since we do not have that much time to finish the project we decided to reduce the number of original artwork and decided to use music and sound effects that are available to us.

2. Software Architecture

2.1 Overview

We started to detail the composition of the system from a higher level, then divide it into smaller subsystems for a better maintainability and understandability.

2.2 Subsystem Decomposition

The system is decomposed into multiple subsystems for ease with manageability and extendibility. We tried to achieve high coherence and low coupling with the subsystem design and decomposition. This will allow the complexity of the system to stay low and make changes easier for the developers or maintainers.

We decided to follow the Model-View-Controller design with our project and divided the system into three subsystems called Management, UserInterface and GameComponents, as shown in the figure below. This choice will help us in the implementation stage with efficiency and simplicity since the interactions between the subsystems is minimal.

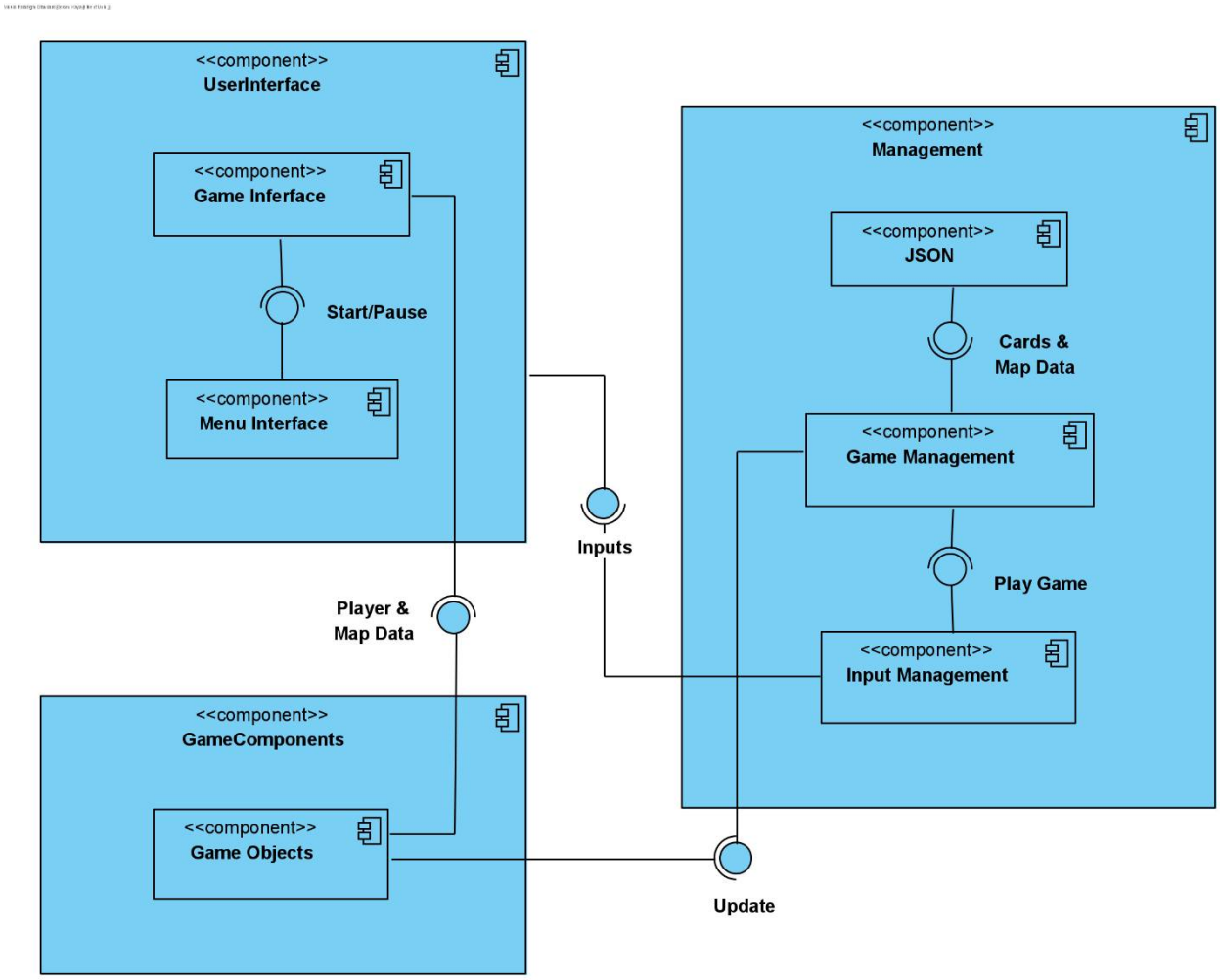

Figure 1: Subsystem Component Diagram

The UserInterface subsystem corresponds to the View subsystem of the MVC design. It consists of all user interface components and provides GUI to the user throughout the time the application is running. The Management subsystem corresponds to the Controller subsystem of

MVC and it handles all operations of the game as well as sequence of interactions with user and objects. The GameComponents subsystem corresponds to the Model subsystem of MVC. It includes all game objects and their data needed to play the game. These subsystems and their services are further explained in the section 3.0 Subsystem Services.

The architecture style of the project is nonhierarchical because of our choice with Model-View-Controller design. Each subsystems requires something from another subsystem and provides something to the other subsystem. None of them can work on their own which makes the architectural style of this system triangular.

2.3 Hardware/Software Mapping

Risk does not require special hardware to run on computers. Since our game is very minimal in the matter of hardware only hardware requirements are pc which will run the game and mouse which will make the player able to put their input to the game.

For the storage of the game, our game does not keep track of high scores or any of the players' data. Only data stored is the game code to initialize the game.

For software, our game is written in Java meaning it will be able to run on a computer that has Java Runtime Environment. Since all Linux, Mac and Windows have needed environment, our game will be available for those computers.

2.4 Persistent Data Management

As mentioned before our game will not hold any data of the previous games or any personal data of the players. Only data that will be stored are the UI artwork, music sound effects, avatars, game code which will be kept in json files. In the game players will be able to choose their territory color and their avatar but they will be stored just for that game meaning they are temporarily stored in json files. For artwork and avatars we will use .png format, and for music and sound effects we will use .wav format (json files).

2.5 Access Control and Security

Our game does not require internet connection. Since it can only be played with one computer and it does not include any private information, there is no access control. Therefore, it does not pose any security issues.

2.6 Boundary Conditions

2.6.1 Initialization

Risk game does not require any installation or software other than Java. Our game is an executable .jar file, therefore it can be run directly.

2.6.2 Termination

Players may exit the game by clicking *Quit Game* button through main menu or pause menu. Before the quit, a prompt will be displayed in order to ensure that the player is sure to exit game.

2.6.3 Failure

Our game includes sounds as well as images. If the path of these source files is changed or if any corruption occurred in files, an error prompt will be displayed during game to inform players about the source file errors. It will not result in a compile error or a run-time error.

We tried to minimize failure rate by disabling irrelevant buttons during some parts of the game. However, a failure may occur if there is a problem with the code, which may cause a run-time error.

3. Subsystem Services

This section provides detailed descriptions of services of the subsystems of our project which were introduced in the Subsystem Decomposition part above. Each subsystem will be separately described from the component diagram in figure 1.

3.1 UserInterface Subsystem

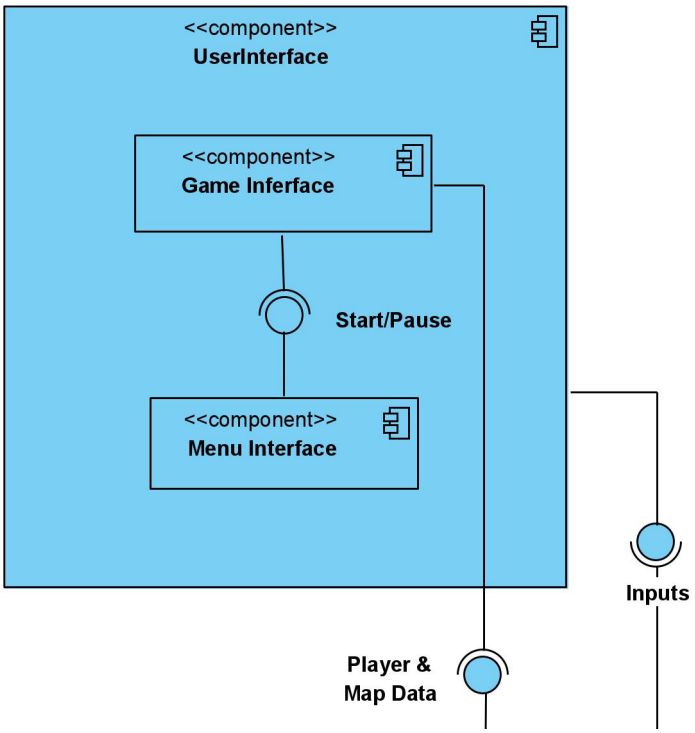

Figure 2: The UserInterface Subsystem diagram (cut)

This subsystem provides all user interfaces of the application, including the gameplay interfaces and all menu interfaces. These two are given as components in the figure above because of their importance. The UserInterface subsystem provides the user inputs that are obtained via the interface to the Management subsystem, such as the buttons clicked on during the game play or in the menu.

The Menu Interface component consists of all menus, which include the pause, customization, settings, credits and how to play menus as well as the menu displayed at the booting of the application. This component includes the GUI class in the frontend package. It does not need any updates or data from other subsystems since it should display the same things for all games. This component provides the necessary menus that can be accessed from the Game Interface such as the pause menu.

The Game Interface is the component that shows the map and players throughout the game. It requires data of those objects and obtains them from the GameComponents subsystem, the models themselves. This interface is first shown when the user starts the game and is

continuously shown afterwards until the end of the game with the exception of pause menu. The class associated with this component is the GUI class in frontend package.

3.2 Management Subsystem

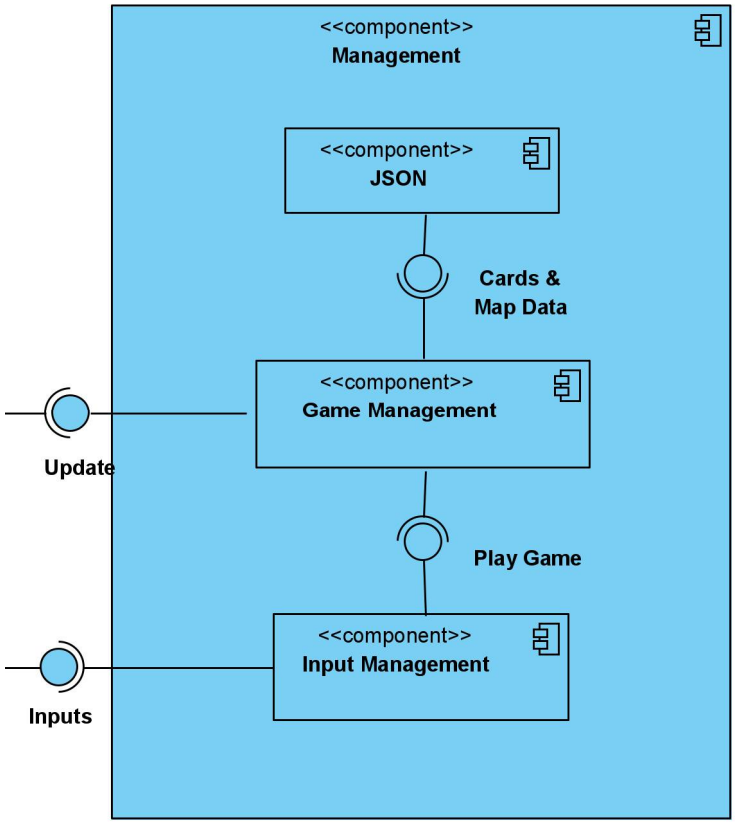

Figure 3: The Management Subsystem diagram (cut)

The Management subsystem corresponds to the Controller subsystem of MVC. It handles each interaction between users and objects, objects and objects as well as controlling the game itself. This subsystem consists of three components called JSON, Game Management and Input Management.

The JSON component consists of the JSON files map and cards as well as JSONParser class in the backend package. These provide the data of all cards, continents and territories. The Game Management component requires these data for the initialization of the game and JSON component provides it.

The Game Management component handles all operations within the game itself such as playing a turn, rolling dice, attacking and more. It requires player inputs from the Input Management component in order to act accordingly and carries on the game. It also updates the

GameComponents subsystem's objects based on the events of the game. Some of the classes associated with this component includes the Game, GameManager and CombatManager classes located in the backend package.

The Input Management component handles the interaction between user inputs and the game. It obtains the user input from the UserInterface subsystem and identifies the required action and provides the input to the Game Management component. This component includes the GUIController class in the backend package.

3.3 GameComponents Subsystem

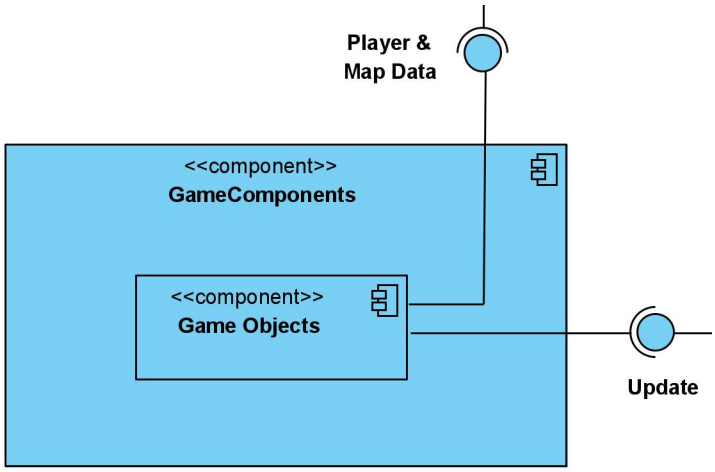

Figure 4: The GameComponents Subsystem diagram (cut)

The GameComponents represents the Model subsystem of MVC. It includes the Game Objects component which is all classes as well as instances of objects needed to run the game. All the different types of objects, such as troops, players and territories, are encapsulated by this component for simplicity.

The Game Objects component, as mentioned, includes all models of the game. The objects get updated by the Management subsystem each time there is a difference. This component provides the data needed for views to the UserInterface subsystem which include the player data and map data which include the owned territories, troop amount on each territory, cards and more. There are many classes associated with this subsystem in the backend package and some examples include Artillery, Calvary, Card, CurseCard and Continent classes.

4. Low Level Design

4.1 Object Design Trade-offs

Space vs. Speed:

The success of the game depends on the preferability compared to other games. To increase the preferability, user experience plays a crucial role. Thus, we cannot compromise the speed of our game because users can get a bad impression if our game works slowly, so this means the main focus for this aspect is speed instead of space. Moreover, since our game is a strategic war game with multiple players, its main purpose is to have fast gameplay so we preferred to increase the usage of the memory rather than decreasing the speed of the game.

Development Time vs Performance:

In order to reduce the development time we have decided to use Java as our language since all the group members were more comfortable with that language. This cost us some performance issues since C++ offers better solutions in some cases, but this decision helped us have a better coding experience with much less problems and faster process. Another important feature of java is the garbage collector, this means if we decide to use C++, we have to deal with memory leaks, but java can deal with this by itself. Therefore, choosing java decreases the development time as well.

Delivery time vs. Quality:

Originally the user interface was planned to have custom artwork, music and sound effects but since we do not have that much time to finish the project we decided to reduce the number of original artwork and decided to use music and sound effects that are available to us. Since, the quality comes from work time and effort, but we have to prefer delivery time for the sake of the project. Therefore, we need to compromise quality which are mainly user interface part for finishing the project before the deadline.

4.2 Final Object Design

Abstract object design is given below for clarity.

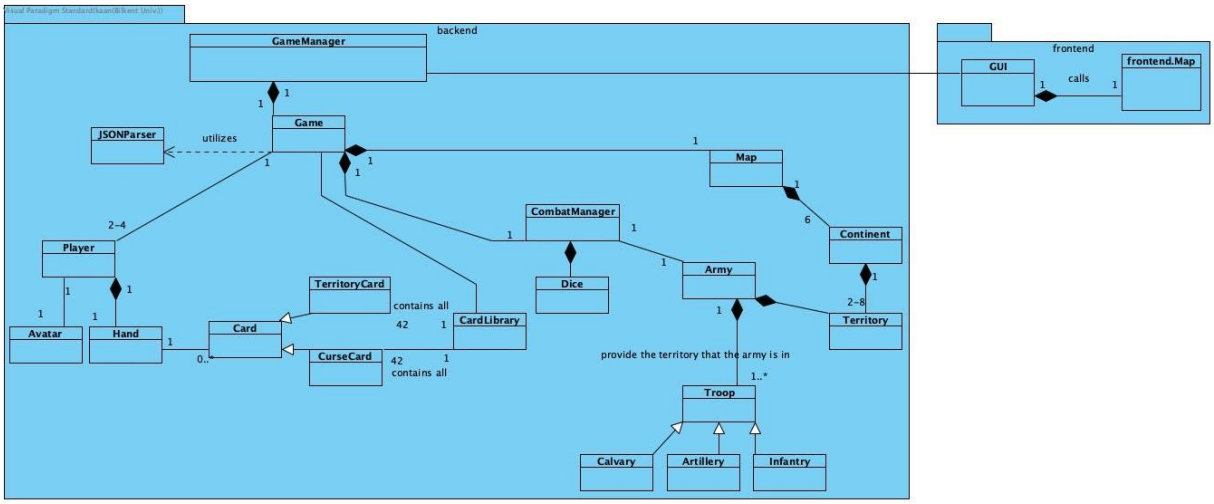

Figure 5.

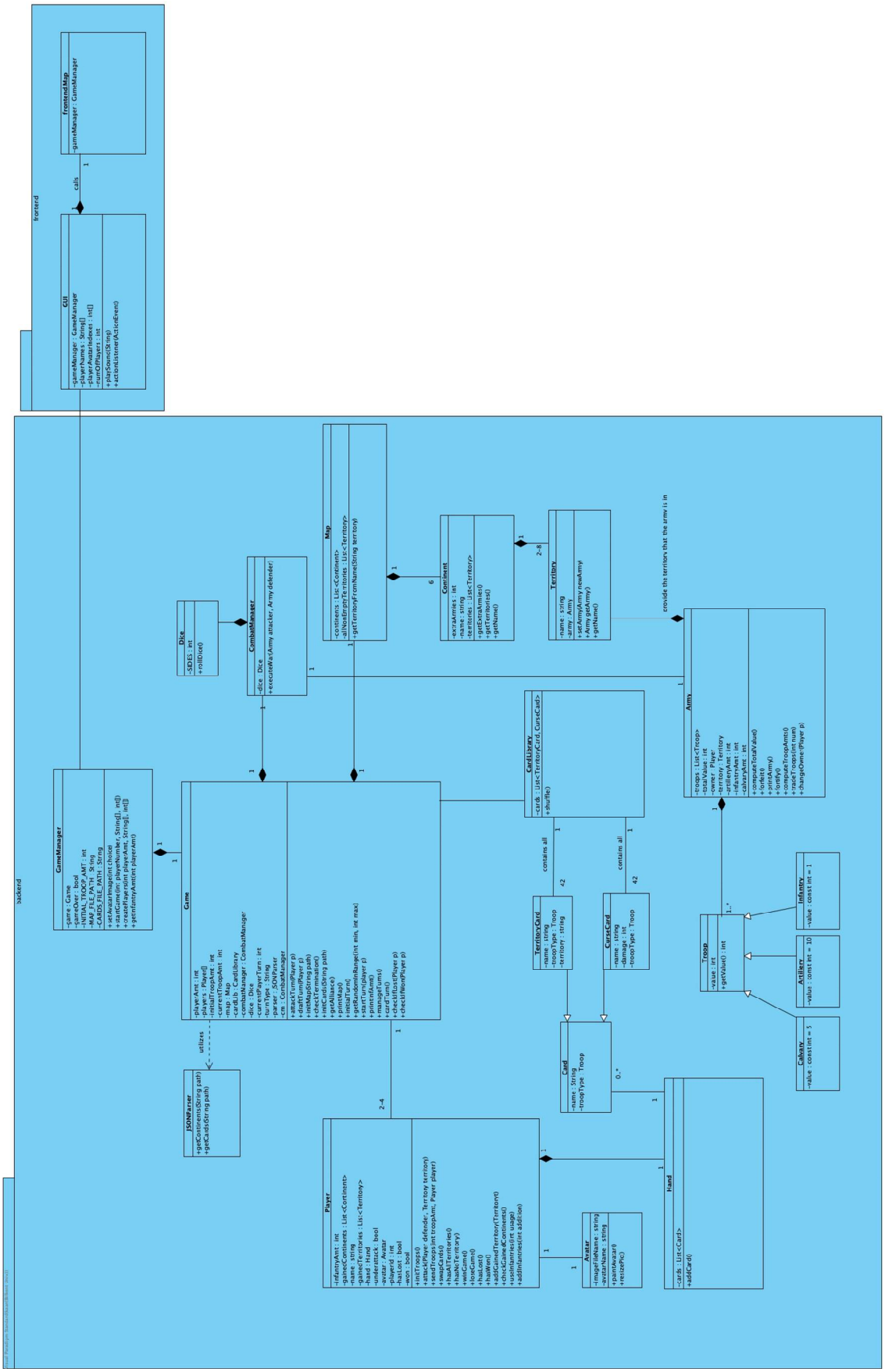
frontend
GUI
FrontendMap
calls
backend
GameManager
Game
Player
Map
Continent
Territory
Army
Troop
Hand
Card
TerritoryCard
CurseCard
CardLibrary
Dice
CombatManager
JSONParser
Avatar
Infantry
Artillery
Calvary
utilizes
contains all
provide the territory that the army is in

Figure 6

4.3 Packages

4.3.1 Internal Subsystem Packages

The internal subsystem packages are the packages that we defined to help ease the management of the code and help with collaboration and implementation.

4.3.1.1 Frontend Package

It includes the classes responsible for all user interfaces and inputs. This package is for classes in the UserInterface subsystem as described above.

4.3.1.2 Backend Package

This package includes all the controllers, managers, components and objects of the game. It consists of the classes in the Management and GameComponents subsystems.

4.3.2 External Subsystem Packages

There are some external packages for our implementation to help our game to better. These packages are crucial and necessary for our implementation and these are mainly JSON, java.lang, java.io, java.util, so mainly we will use java standard libraries.

4.3.2.1 Java.util

This package provides ArrayLists, scanner and random numbers to use in implementation.

4.3.2.2 Java.lang.reflect.Array

This package provides static methods for arrays and access to java arrays.

4.3.2.3 org.json.simple.JSONObject

This package provides reading and writing JSON data and full compliance.

4.3.2.4. org.json.simple

This package provides encoding and decoding JSON data.

4.4 Class Interfaces

In this section, we have introduced our design in more detail for clarity.

Backend Package:

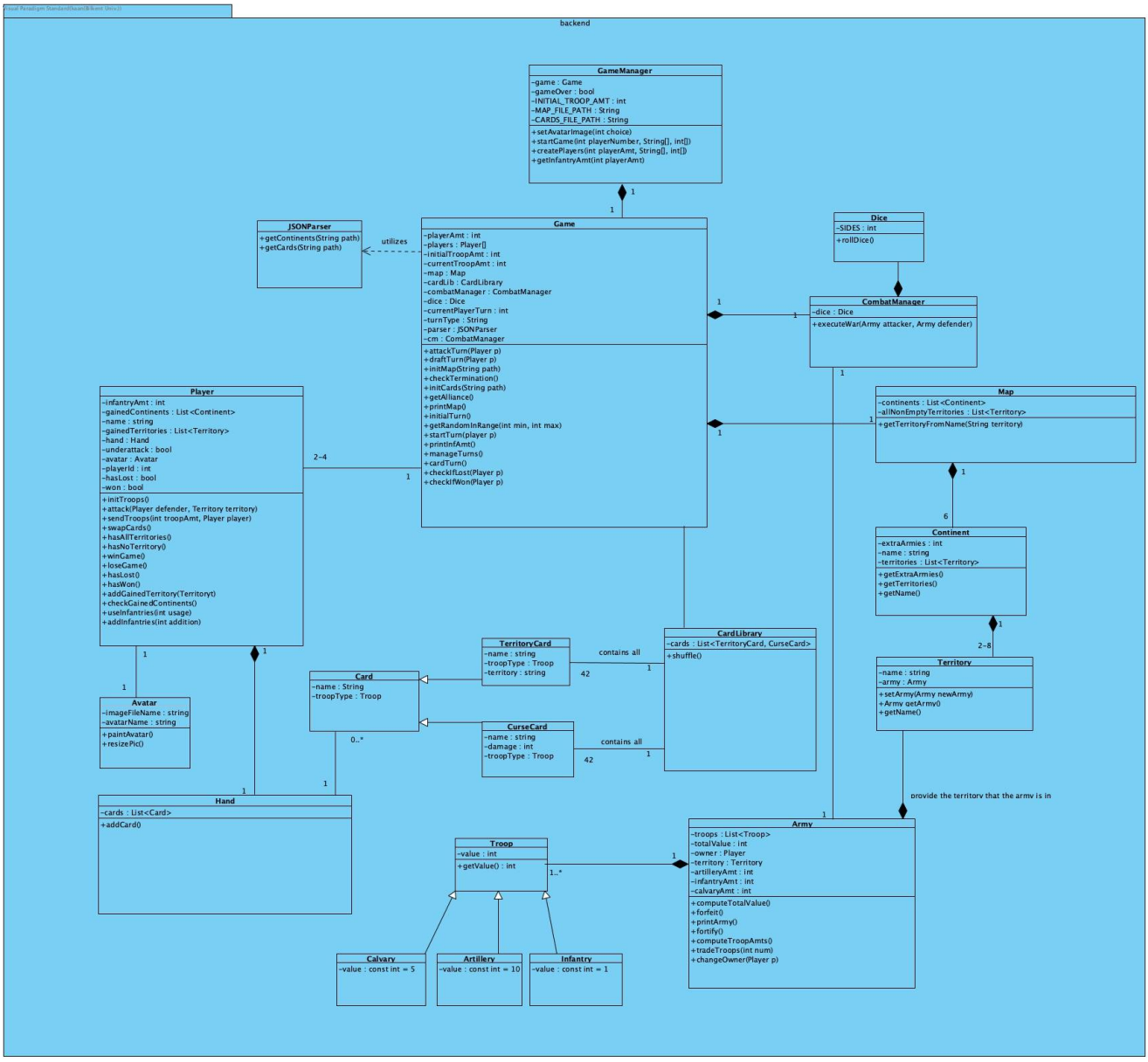

Figure 7

4.4.1 Game class

Attributes:

- private final int playerAmt: The amount of players
- private Player[] players: Contains player of the game

- private final Dice dice: the rolled dice
- private int currentTroopAmt: The current amount of troops of a player
- private CardLibrary cardLib: The collection of the cards
- private Map map: the map object of the game
- private final JSONParser parser:
- private final CombatManager cm: The object of CombatManager class to determine the winning side of a war
- private int initialTroopAmt: The initial number of troops of a player
- private final int CONTINENT_AMT: The number of continents

Methods:

- private void initMap(String path): Create map by parsing the data from a JSON file
- public void printMap(): prints the game map
- private void initCards(String path): Create cards by parsing the data from a JSON file
- public void initialTurn(): When the game starts, each player gets their continents and army
- private static int getRandomNumberInRange(int min, int max): return random number between min-max both inclusive
- private void startTurn(Player p): At the beginning of each turn, the player gets their infantries
- private void printInfAmt(): It prints number of infantries of a player
- private void manageTurns(): Manages the each turn to continuity with each features
- private void draftTurn(Player p): Replacing the player' troops inside his territories
- private void attackTurn(Player p): Determines which will win the declared war.
- private void cardTurn(): Checks the conditions where a player receives a card and the player determines whether he will use card or not.
- private void checkIfLosed(Player p): Checks the losing condition for a player.
- private void checkIfWon(Player p) : Checks the winning condition for a player.
- private void checkIfWon(Player p) : Checks the ending condition of the game.

4.4.2 Player Class

Attributes:

- private int infantryAmt: The number of infantry for a player
- private final String name: The name of the player
- private final Avatar avatar: The avatar object which is chosen by player
- private Hand hand: Contains the cards of the player
- private boolean underAttack: boolean type to check whether the player is under attack or not
- private ArrayList<Continent> gainedContinents: The list of player's gained continents
- private ArrayList<Territory> gainedTerritories: The list of player's gained territories
- private final int playerId: The Id of the player
- private boolean hasLost: Checks whether the player is lost or not
- private boolean won: Checks whether the player has won or not

Methods:

- public void winGame() : Controls whether the player win the game or not
- public void loseGame() : Controls whether the player lose the game or not
- public ArrayList<Continent> getGainedContinents() : Return the gained continents list
- public int getInfantryAmt() : returns the number of infantries of a player
- public ArrayList<Territory> getGainedTerritories(): Return the gained continents list
- public void addGainedTerritory(Territory t): Adds the gained territorry from the war to the gained territory list
- public void checkGainedContinents() : Checks the continent whether it is gained earlier or not
- public boolean hasNoTerritory(): Sets the number of territories to zero for a player
- public boolean hasAllTerritories(): Sets the number of territories to six which is the maximum for a player
- public void useInfantries(int usage): Decrements the number of infantries by the usage number
- public void addInfantries(int addition) : Increments the number of infantries by the addition number
- void sendTroops(int troopAmt, Player player): Sends the number of troops to opponent player for a war
- void attack(Player defender, Territory territory) : To declare war to another player for specific territory

4.4.3 JSONParser class

Methods:

- public ArrayList<Continent> getContinents(String path): Takes the JSON file's path as the parameter. Creates and returns a continents list by parsing the data from a JSON file.
- public ArrayList<Card> getCards(String path) : Takes the JSON file's path as the parameter. Creates and returns a cards list by parsing the data from a JSON file named path

4.4.4 Map class

Attributes:

- private final ArrayList<Continent> continents: The list of the continents in the map

Methods:

- public ArrayList<Territory> getAllNonEmptyTerritories() : Get all territories which has an army in it
- public Territory getTerritoryFromName(String territory) : Returns the territory with the corresponding name, returns null if not found

4.4.5 CombatManager class

Attributes:

- private final Dice dice: The dice object to be rolled

Methods:

- public void executeWar(Army attacker, Army defender): Execute war between two armies and update the armies and territory owner according to winner

4.4.6 Army class

Attributes:

- private ArrayList<Troop> troops: The list of troops which the army has
- private int totalValue: Total value of all of the troops in this army

- private Player owner: Owner of the army
- private Territory territory: Territory where the army is in
- private int infantryAmt: The number of infantries the army has
- private int calvaryAmt: The number of calvary the army has
- private int artilleryAmt: The number of artillery the army has

Methods:

- private void computeTotalValue(ArrayList<Troop> troops): Return total value of the army in terms of infantries
- public ArrayList<Troop> getTroops(): Return the troops from the list
- public Player getOwner(): Gets the owner of the army
- public int getTotalValue(): Gets the total troops in the army
- public void forfeit(): Reduces one infantry from the army
- public void fortify(ArrayList<Troop> newTroops): Increase the amount of soldiers
- private void computeTroopAmts(ArrayList<Troop> troops): Computes the total number of troops in an army
- public int getInfantryAmt(): Gets the number of infantries in a army
- public int getArtilleryAmt(): Gets the number of artillery in a army
- public int getCalvaryAmt(): Gets the number of calvary in a army
- public void tradeTroops(int num): Player can trade 5 infantries for one calvary and 10 infantries for one artillery

4.4.7 Dice class

Attributes:

- private static final int SIDES: It represents the number sides of the dice

Methods:

- public int rollDice(): Rolls a dice randomly

4.4.8 Troop class

Attributes:

- protected int value: The value of the troop

Methods:

- public int getValue(): Gets the value of the troop

4.4.9 Calvary class

Attributes:

- constant int value = 5: The value of calvary

4.4.10 Artillery class

Attributes:

- constant int value = 10: The value of Artillery

4.4.11 Infantry class

Attributes:

- constant int value = 1: The value of Infantry

4.4.12 Avatar class

Attributes:

- private final String imageFileName: Contains the avatar image path.

Methods:

- public void paintAvatar(): Prints and draws the avatar
- public void resizePic(): Resize the avatar image

4.4.13 Continent class

Attributes:

- private final int extraArmies: The armies are provided by continent for each turn
- private final String name: The continent name

- private final ArrayList<Territory> territories: The list of territories which are belongs to the continent
-

Methods:

- public int getExtraArmies(): Gets the extra army number
- public ArrayList<Territory> getTerritories(): Return the list of territories which continent has
- public String getName(): Return the name of the continent

4.4.14 Territory class

Attributes:

- private final String name: The name of the territory
- private Army army: The collection of armies which are placed in the territory

Methods:

- public void setArmy(Army newArmy): Sets the army with the new after the war to update the condition
- public Army getArmy(): Gets the army in the territory
- public String getName(): Gets the name of the territory

4.4.15 Hand class

Attributes:

- private ArrayList<Card> cards: The list of cards which a player has

Methods:

- public void addCard(Card card) : Add card to the list which a player has

4.4.16 Card class

Attributes:

- private final String name: The card name

- private final Troop troopType: To determine which type of troops can use the card

4.4.17 CurseCard class

Attributes:

- private final int damage: To use in wars, determine the amount of damage to opponent

4.4.18 TerritoryCard class

Attributes:

- private final String territory: To understand the card which territory it belongs

4.4.19 CardLibrary class

Attributes:

- private ArrayList<Card> cards: The list of card in the library

Methods:

- private void shuffle(): At the beginning of the game, mix the deck

4.4.20 GameManager class

Attributes:

- private Game game: Creating a game object
- private boolean gameOver: boolean type to check the game is finished
- private final int INITIAL_TROOP_AMT: At the beginning, destribute each player equal number of troops
- private final String MAP_FILE_PATH: To get map from the file
- private final String CARDS_FILE_PATH: To get cards from the file

Methods:

- public void startGame(int playerNumber,String[] playernames, int[]playerAvatars): Starts the game creating each player and give the initial statements

- private Player[] createPlayers(int playerAmt,String[] playernames, int[]playerAvatars) : Get the names and avatar pics of the users and create the players accordingly
- private int getInfantryAmt(int playerAmt) :return the number of infantries for each player based on the number of players
- private String setAvatarImage(int choice): returns the path of the avatar image with the given choice

Frontend Package:

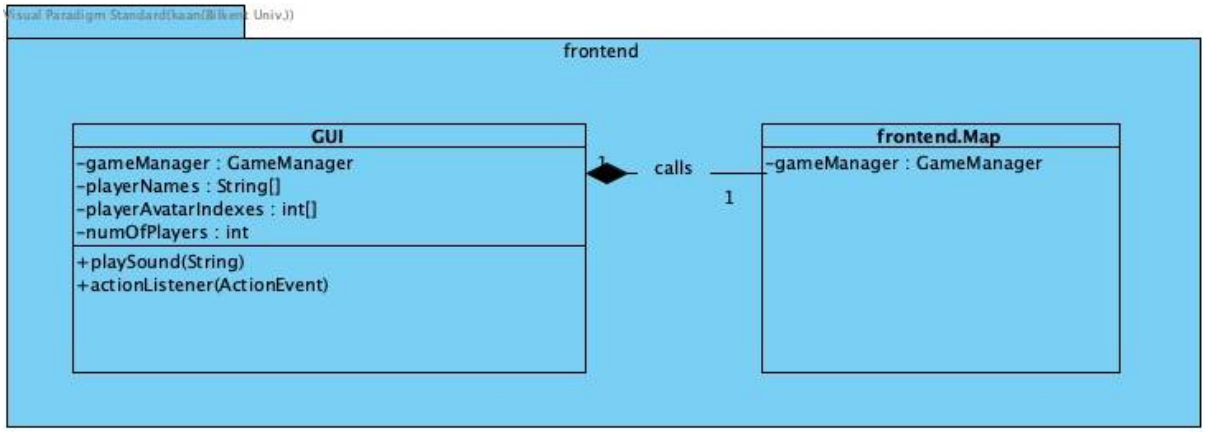

Figure 8.

4.4.21 GUI class

This class contains the main menu and game start panel which contains the fields for player names, colors and avatars.

Attributes:

- private GameManager gameManager: Creates a gameManager instance to set up the Game by taking in the player properties.
- private String[] playerNames
- private int[] playerAvatarIndexes
- int numOfPlayers

4.4.22 Map class

This class contains the UI elements for game map and interactions between players and the map. Map class gets created from the GUI class after the players fill their required fields and click start game button.

Attribute:

- private GameManager gameManager: GUI class passes this attribute to the Map class.

5. Conclusion

In this report, we explained the design of our Risk game. Our design goals are explained in the introduction section. Additionally, the design goals which depend on the implementation of the game are explained in criteria sections. Trade-offs are also mentioned at the end of the introduction. Then we divide our system into subsystems for a better maintainability and understandability. We covered the mapping between the hardware and the software as well as boundary conditions of our game. The access control and security is not explained in detail since the game does not pose any security issues. Ultimately, final object design of our project, packages and the class interfaces are explained in detail in the low level design section.

6. References and Glossary

[1] “Risk (Game).” *Wikipedia*, Wikimedia Foundation, 29 Nov. 2020, en.wikipedia.org/wiki/Risk_(game).