OOP Project

# Related image**Project Design Report: Iteration 1**

CS 319 Object Oriented Software Engineering  
Eray TÜZÜN  
Group 1-I: Private Moon Inc.

Project Group Members

• Doğa Oruç – 21602022

• Çağrı Orhan - 21503102

• Fazilet Simge Er - 21602358

• Kaan Ünlü - 21602622

• Ege Turan – 21502441

Contents

[**Project Design Report: Iteration 1** 1](#_Toc3932545)

[1.0 Introduction 4](#_Toc3932546)

[1.1 Purpose of the system 4](#_Toc3932547)

[1.2 Design goals 4](#_Toc3932548)

[2.0 High-level software architecture 5](#_Toc3932549)

[2.1 Subsystem decomposition 5](#_Toc3932550)

[2.2 Hardware/software mapping 6](#_Toc3932551)

[2.3 Persistent data management 6](#_Toc3932552)

[2.4 Access control and security 7](#_Toc3932553)

[2.5 Boundary conditions 7](#_Toc3932554)

[2.5.1 Initialization 7](#_Toc3932555)

[2.5.2 Termination 7](#_Toc3932556)

[2.5.3 Failure 7](#_Toc3932557)

[3.0 Subsystem services 7](#_Toc3932558)

[3.1 View 8](#_Toc3932559)

[3.1.1 The Game Board 8](#_Toc3932560)

[3.1.2 GUI View 8](#_Toc3932561)

[3.1.3 Card and Dice View 8](#_Toc3932562)

[3.2 Logic 8](#_Toc3932563)

[3.2.1 Turn Logic 8](#_Toc3932564)

[3.2.2 Province Logic 9](#_Toc3932565)

[3.2.3 Cards and Dice Logic 9](#_Toc3932566)

[3.2.4 AI Logic 9](#_Toc3932567)

[3.2.6 GUI Logic 9](#_Toc3932568)

[4.0 Low-level design 9](#_Toc3932569)

[4.1 Object design trade-offs 9](#_Toc3932570)

[4.2 Final object design 10](#_Toc3932571)

[4.3 Class Interfaces 10](#_Toc3932572)

[4.3.1 Class: Main 10](#_Toc3932573)

[4.3.2 Class: GameManager 10](#_Toc3932574)

[4.4.3 Class: AssetLoader 11](#_Toc3932575)

[4.3.4 Class: Globals 11](#_Toc3932576)

[4.3.5 Class: Camera 11](#_Toc3932577)

[4.3.6 Class: Vertex2D 11](#_Toc3932578)

[4.3.7 Class: DebugPabel 11](#_Toc3932579)

[4.3.8 Class: MapLoader 12](#_Toc3932580)

[4.3.9 Class: GameBoard 12](#_Toc3932581)

[4.3.10 Class: Dice 12](#_Toc3932582)

[4.3.11 Connection Classes 12](#_Toc3932583)

[4.4.12 Exception Classes 16](#_Toc3932584)

[5.0 Glossary & references 18](#_Toc3932585)

# 1.0 Introduction

Private Moon Inc.’s Project Risk is a digital revival of the 1957 table-top classic Risk [1]. The original game is a 2 to 6 player game that rewards strategizing and a silver tongue.

The game consists of a game board, 6 sided dice, cards and pawns. The game board is usually a map of the world with named provinces. The provinces are grouped inside continents of different colours. There are two types of dice: Red and White. Also, there are 2 different card types: Province Cards and Secret Mission Cards. Finally, there are 6 different colours of pawns meant to be used by up to 6 players. These pawns can be found in three different shapes, usually dubbed infantry, cavalry and artillery, symbolising 1, 5 and 10 armies respectively.

The objective of the game for a player is either to conquer the entire world or if the players decide to play with secret missions, finish four secret mission before the others do. For this, each player takes a turn one after another and act to conquer new lands. Each player turn has three phases:

* As their turn begins, the player gets a number of new armies equal to their total number of provinces divided by 3 rounded down, plus the value of their continent if they completely have conquered one, plus the next number on the army table to the bottom of the game board if they have turned in 3 province cards of the same marking, or 3 province cards each of a different marking. Also if any of the cards the player has turned in that turn has a province they already have pictured on it, they receive an extra 2 armies specifically for that province. The rest of the armies can be deployed wherever the player wants.
* Then, if wished so, the player can attack a province neighbour to one of their own provinces where the attacking province must have at least 2 armies in it. The player can attack as many times as they want on the same or on different provinces. On any singular attack, the attacking side can choose to roll up to 3 red dice (where DiceNumber < armiesOnAttackingProvinceNumber) and in response the defending side can choose to roll up to 2 white dice (where DiceNumber <= armiesOnDefendingProvinceNumber). All dice are rolled, and the maximum white dice number of attacker’s dice rolls are compared against the results of the white dice. The red dice have to score higher than white dice, or for every die that doesn’t, an army on the attacking side perishes. The opposite of this happens for every higher red dice value: An army on the defending side perishes.
* Finally, to fortify their position, the player can move as many armies as they want from one and only one province to one and only one other neighbouring province [2]. The fortification can only happen between connected provinces.

## Purpose of the system

Risk is a strategy game that incentivizes the players into thinking ahead. As however, the original Risk only consists of a single map with a relatively small province connection graph, strategies can start getting stale very quick, with some moves getting more and more familiar from previous playthroughs. This digital adaptation of Risk therefore aims to include more maps and the opportunity to play against AI or other players to keep the experience novel for longer, and allow for a more engaging gameplay that gives space to players to strategize more freely.

## 1.2 Design goals

Planning the design of the game works on a must have-should have-could have-won’t have basis. Design goals are distributed between these categories according to their priority and how essential they would be at improving the player experience and make this game more than its traditional predecessor. The MoSCoW analysis below shows the details of these goals:

The Game must have:

* A Map
  + Whose provinces can be selected viewed accurately
  + On whose provinces armies can be deployed, viewed and sent to other provinces in transportation or attack
* Logic and visualization to show the number of armies received at the start of a turn
* Dice and card shuffling logic
* Province and secret objective cards
* Offline playing capability with friends
* Online playing capability over direct IP
* Offline playing capability over AI
* Limited multiple language support
* Limited colour blindness support

Should have:

* Mod Support
  + Filter based colour blindness support
  + Multiple resolution support
  + Custom map addition support
* More languages
* Better dice and card visualization

Could have:

* Music and sounds
* Map zoom
* Better province name visualization
* Server based online multiplayer capability
* AI, with flavour
* Game saving in offline games
* Diplomacy mechanics

Won’t have:

* Micro-transactions, loot boxes
* Accounts
* Game saving in online games

# 2.0 High-level software architecture

## 2.1 Subsystem decomposition

The system is composed of logic and view, respectively called “Tickable” and “Renderable”. Separating the logic and view makes the system easier to maintain, and easier to expand on.

The view is used to show the user a representation of the game logic, and allow for interaction with that logic. It consists of:

* The Game Board
* GUI View
* Card and Dice View

The logic on the other hand, works in the background to manage the mechanics of the game. Where for example the logic shows the province cards to the user and allows them to interact with them, the logic manages which cards to send to the viewer, which to receive from the user, shuffle the cards and et cetera. It consists of:

* The Game Manager, managing:
  + Turn Logic
  + Province Logic
  + Card and Dice Logic
  + AI Logic
* GUI Logic(GUI updates)

More details on the subject can be found in part 3.

## 2.2 Hardware/software mapping

Software-wise, as the game is implemented in java, it can run in any environment with the JRE setup in it. However, if the user wishes to play the game online, their computer will need to be connected to the internet.

Hardware-wise, as nothing is rendered in 3D, the computer used to run the game doesn’t need spectacular specifications. IO-wise, a mouse is needed to navigate in the game board and interact with the provinces. A keyboard is also needed for map control shortcuts and menu access. Also, while not much, hard drive space(<100MB) will be required to save game data in offline playing modes and store game files.

When playing the game online, one person hosts the game, and the rest of the players connect to the game by using the IP address of the host. Saving the game while playing online is not implemented as loading such a save would mean re-establishing lost connections from the previous game, which isn’t done here.

As there are no accounts in game, nothing is stored online, so databases aren’t necessary.

## 2.3 Persistent data management

Most data management is done locally. The sole case it isn’t is during a multiplayer game, as objects are sent and received via sockets.

Locally, there are two kinds of save data. One holds user settings, the other holds game saves.

User settings include options such as screen resolution, game language, sound levels and colour blindness options and can be mutated using the options menu which is accessed from the settings button in the main menu. User settings data will be saved to the underlying operating system’s hierarchical nodes. Every time new options are applied, they are saved onto the existing node.

Game saves include information about game state, such as the number of armies in each province, to which player they belong to, what turn it is, what cards each player has and et cetera. This save data cannot be mutated by the player as they could the user settings; rather, they are saved using the save button in the game screen during the game, and can be loaded from the main menu or during a game from the game screen. Unlike in user settings, whenever a game is saved, it is saved on a separate non-human readable file to the “/saves” directory.

## 2.4 Access control and security

Risk will not require any personal information to play the game, be it an e-mail, or even a name. The only information that might be counted as sensitive will be the IP of the computer (which belongs to the player who is hosting the game), and that will be requested from the host right before starting a multiplayer game.

The host and the clients mentioned above are players as well. Therefore, the only actor in our system is the player. The access matrix of the system can be seen below. Keep in mind, the matrix is heavily simplified in order to maintain this report’s length.

|  |  |  |
| --- | --- | --- |
|  | DoaMouse | DoaKeyboard |
| Player | \*MOUSE\_BUTTONS\* | \*KEYBOARD\_KEYS\* |

## 2.5 Boundary conditions

### 2.5.1 Initialization

Risk will be released both as a JAR (.jar) file and as an EXE file (.exe). While the JAR file can be used by everyone on any platform, the EXE file will only be used by users whose computers operate on Microsoft Windows. We made the decision to release Risk in both formats because even though JAR files can run on any system, it also requires the Java Runtime Environment to be installed on the machine. The freedom given by JAR is taken away by JRE in that case. In order to make Windows Users’ lives easier, we decided to release the game also in EXE format. This will allow Windows Users without an installed JRE to play our game without going out of their way to install other software. When the user gets a hold of either one of the Risk executables, what s/he needs to do is run the executable, this will result in a splash screen being shown until the game fully loads itself, then the actual game’s main menu being shown. From there, the user can press one of the “Play” buttons to start playing Risk.

### 2.5.2 Termination

The game can be terminated either by clicking the Exit button on the Main Menu or Esc button on the keyboard while the game is ongoing. Either one of these actions will be followed by a pop-up message which says “Are you sure?” on the screen, and can be answered by either “Yes” or “No”. If the player chooses “No”, the player will be redirected to the last screen s/he was in. Otherwise, the game will terminate.

### 2.5.3 Failure

Although we will try our best to get rid of anything that might cause any kind of inconvenience for the player, we might encounter some undesired situation, which can be a failure. In case of failure, game will display an error message in the form of a pop-up message. A stack trace will also be provided in the message. This trace can be sent to us, if the player wants, to notify us that such a situation arised.

# 3.0 Subsystem services

As mentioned in Section 2, the system is composed of logic and view. Although these two different systems run separately, they need to be combined in order to form the actual game. Designing as this way, we can achieve modularity to make things better.

## 3.1 View

All objects that the user can see and use on the screen are about view. View is allowing us to establish the bridge between player and actual code. We use modularity again to supply a representation of the game with users. View are as follows:

### 3.1.1 The Game Board

The game board is our main component of our game. As Risk is a board game and the game board is representing the physical board of the Risk to the users. The Game Board consists of the fundamental elements of Risk game such as continents, provinces, troops and all the elements that present in the actual board. The game board is connected with logic of the game so that we can update the view of the board when a change occurs. For example when a user had attacked an enemy province and won the battle which means ownership of the province changed. Game board changes the border colour of the captured province with the up-to-date version. Interaction with users also is provided by the game board. Users can select a province to attack, view the number of troops by the help of the game board.

### 3.1.2 GUI View

GUI view is our main tool to draw images according to data that comes from the logic part. Implementing a well-designed Risk game requires a strong GUI view to satisfy the expectations of the users. The HUD in the game screen is shown to the player using the GUI view. Menu screen, options screen, and setting menu are also related with GUI view. GUI provides menu screen’s views to the user. All in all, GUI view lets the player see what they are interacting with on the user interface of the game on screen.

### 3.1.3 Card and Dice View

Dice are only available when a player tries to either attack or defend. Cards are available either at the very start of the game (secret objective cards), when conquering a province (getting province cards) or at the beginning of each turn (turning in province cards). This separation allows us to follow modularity conventions in order to make things better. For example, if a player wants to capture a province, the player has to roll a dice. Process is as follows:

* The player selects the number of dice to roll on a field supplied by the GUI view.
* The logic of the game generates a random number from one to six per dice and sends the data to a dice view.
* Dice view shows the result on the screen.

So values of the dices are generated by the logic while visual of dices are generated by dice view, they work separately.

## 3.2 Logic

Another subsystem of the Risk game is the logic. Logic works in the background and cannot be directly accessed by the player. But what the player does on the view is controlled and supported by the behaviour and calculations done in the logic. Most of this logic is managed in the game manager and items that extend it. The rest is controlled inside GUI logic separately.

### 3.2.1 Turn Logic

Turn logic manages all processes that may occur a single turn. Examples to its functions are as follows:

* Determine whether the player’s movement attempts are valid or not
* Check turn phase and ask player for meaningful input
* Manage movement of armies

All the mechanics above are handled by turn logic. Turn logic allows players to make a valid move only if it is the player’s turn.

### 3.2.2 Province Logic

Province logic manages the properties of provinces such as:

* Name and location of provinces
* Manage Province ownerships
* Managing paths from one province to another

### 3.2.3 Cards and Dice Logic

Both dice and cards depend on RNG based systems to a certain extent, as cards are shuffled and dice is rolled throughout the game. This RNG is controlled by their logic. Dice logic doesn’t include much else. Cards logic on the other hand, includes behaviour such as sending data to the game manager about special end-game conditions assigned to players via secret mission cards, or the extra number of armies that will be given to the player to deploy when they turn in a set of province cards.

### 3.2.4 AI Logic

AI’s logic is based on that of an actual player. The AI player interacts with the logic of the game directly unlike a human player, as it needs no visual interface, but is limited to things that can be done by a human player.

#### 3.2.4.1 Placing the Soldiers

When the AI is choosing places for the new soldiers, there are two situations that exist. The first one occurs at the start of the game. In this situation, AI first considers the continents to gain more soldiers in incoming turns and looks at the borders of the continents. AI starts putting its soldiers to decided provinces and tries to sustain them to gain control of continents later in the game.

Secondly, when each turn AI takes new soldiers as other players. AI prefers to put soldiers to the passing point to protect its continent and territorial integrity.

#### 3.2.4.2 Defensive and Aggressive Decisions

When AI is deciding in its movements, it considers the neighbour provinces and its owners. It first counts the number of soldiers of the possible defenders. Then it counts its own soldiers count and also determines the possible threats for its defensive situation. It counts the number of soldiers of the neighbour factions and combining these it decides to attack or fortify its protection points.

### 3.2.6 GUI Logic

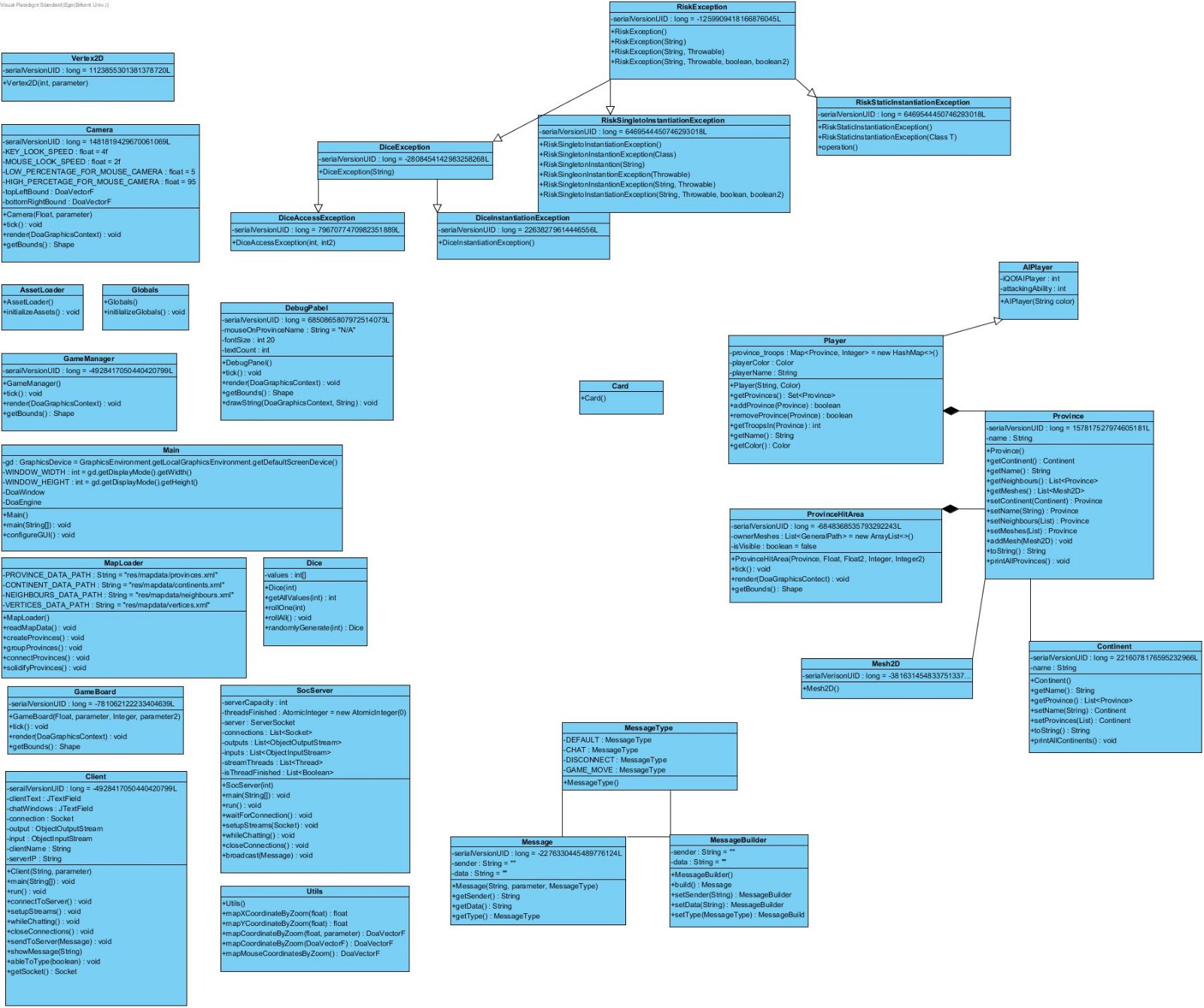
GUI logic determines what GUI view is going to draw by supplying data. It processes data that is needed for GUI view like where to place a button, where that button sends the user to when pressed and et cetera. It sends the data to the GUI view so that a GUI view will be able to draw images and locate the images into the proper location with proper properties. Generally, GUI logic determines the way of working of the GUI view.

# 4.0 Low-level design

## 4.1 Object design trade-offs

* **Delivery Time vs. Functionality:** We will implement the main ideas first, and we will add additional features in our remaining time. In case we run out of time, some features we planned to implement earlier might not be included in the game. Refer to the MoSCoW analysis earlier in 1.2.
* **Efficiency vs. Portability:** While it is advantageous to use Java in programming due to its cross-platform support, because platforms such as Android do not support AWT (which is used in this software), the portability is limited in some aspects.
* **Cost vs. Robustness:** As is with nearly all software, bugs are a large part of the workload that comes with development. Due to that, the process of bug-fixing costs a lot of precious time that instead can be used to develop more features. Colleagues will test the game at various stages to hopefully circumvent this cost as much as possible.

## 4.2 Final object design



## 4.3 Class Interfaces

### 4.3.1 Class: Main

* Operations
  + public main(String[]) : This method arranges DoaEngine, DoaMouse, Globals, DoaCamera and starts the game.
  + private static void configureGUI() : This method arranges the size, title, location properties of the GUI and makes it visible to the user.

### 4.3.2 Class: GameManager

This class extends DoaObject.

* Operations
  + public void tick() : This method updates the inner state of the object and also updates the attribute values and makes fit for the game situation.
  + public void render(DoaGraphicsContext) : This method does nothing, exists because of abstraction.
  + public Shape getBounds() : Returns null, exists because of abstraction.

### 4.4.3 Class: AssetLoader

* Operations
  + public static void initializeAssets() : This method reads the maps from png files to create sprites.

### 4.3.4 Class: Globals

* Operations
  + public static void initilalizeGlobals() : This method reads map data from MapLoader and also initializes the read assets from AssestLoader.

### 4.3.5 Class: Camera

This class extends DoaObject.

* Operations
  + public Camera(Float x, Float y): It takes float x, float y and arranges the bounds of the camera and creates that object with these values.
  + public void tick() : This method updates the inner state of the object and also updates the attribute values and makes fit for the game situation.
  + public void render(DoaGraphicsContext): This method does nothing, exists because of abstraction.
  + public Shape getBounds() : Returns null, exists because of abstraction.

### 4.3.6 Class: Vertex2D

This class extends DoaVectorI.

* Operations
  + public Vertex2D(int x, int y): Represents a point in a Mesh2D.

### 4.3.7 Class: DebugPabel

This class extends DoaObject.

* Operations
  + public DebugPanel(): This constructor creates the debug panel object of the game and initializes its properties.
  + public void tick() : This method updates the inner state of the object and also updates the attribute values and makes fit for the game situation.
  + public void render(DoaGraphicsContext g) : This method renders the Debug Panel and with using the DoaGraphicsContext object, it creates the visualized version of the Debug Panel. Debug Panel has various text that will help us debug the game.
  + public Shape getBounds() : Returns null, exists because of abstraction.
  + private void drawString(DoaGraphicsContext g , String s): This method uses the DoaGraphics object and takes the desired string and draws that gives a string to the desired place on the screen. Used for automated string rendering.

### 4.3.8 Class: MapLoader

* Operations
  + private MapLoader(): Static class, private constructor.
  + public static void readMapData() : This method reads the map data and creates the properties of the map.
  + private static void createProvinces() throws JDOMException, IOException : This method creates provinces using XML files.
  + private static void groupProvinces() throws JDOMException, IOException: This method groups provinces into continents using XML files.
  + private static void connectProvinces() throws JDOMException, IOException: This method connects provinces (puts them to a graph) using XML files.
  + private static void solidifyProvinces() throws JDOMException, IOException: This method reads the mesh data of provinces.

### 4.3.9 Class: GameBoard

This class extends DoaObject.

* Operations
  + public GameBoard(Float x, Float y, Integer width, Integer height): This constructor takes the x, y coordinates of the Game Board and it’s width and height.
  + public void tick(): This method does nothing, exists because of abstraction.
  + public void render(DoaGraphicsContext g): This method uses DoaGraphicsContext object and draws the GameBoard with given x, y, width and height properties. World map is drawn here.
  + public Shape getBounds(): Returns null, exists because of abstraction.

### 4.3.10 Class: Dice

* Operations
  + public Dice(int numberOfDice): This constructor creates a Dice collection with the desired number of dice.
  + public int[] getAllValues(): This method returns the values of the dice in the collection.
  + public int getValueAt(int index): This method returns the die value at a particular index.
  + public void roolOne(int diceIndex): This method rolls only one die. Takes index of the die to be rolled.
  + public rollAll(): This method rolls all of the dices in the collection.
  + public static Dice randomlyGenerate(int numberOfDice): This method takes the desired number of dices and creates a dice collection, rolls all of these dice. Finally, it returns the collection itself.

### 4.3.11 Connection Classes

#### 4.3.11.1 Class: SocServer

This class implements Runnable.

* Operations
* public SocServer(int serverCapacity): This constructor takes the number of participants of the server and creates the server object for creating threads, connections, outputs, inputs, streamThreads.
* public static void main(String[] args): this method creates SocServer and threads.
* public void run() : This method makes port forwarding task and then starts waitForConnection, setupStreams, whileChatting, closeConnections methods. And sustains the workflow of these methods. Briefly, it controls all the server instructions and functions and it closes the connections when desired.
* private void waitForConnection() throws IOException: This method accepts connection calls and then, initializes the connections. After it initialized the connections it adds these connections to the connections list.
* private void setupStreams(Socket socket) throws IOException: This method creates the ObjectInput and ObjectOutput streams. Then adds these streams to the list. Briefly, It creates the input and output streams for the accepted connections.
* private void whileChatting() : This method control the messaging and the data transfer among the participants of the server. It examines all of the steams and determines the type of the input with using MessageType. According to the MessageType, it determines is the input for chat or for game data. Then sends this input to the other clients.
* private void closeConnections() : This method works after all the participants of the server left the server. It closes connections and streams. Briefly, it sustains closing operations of the Server.
* private void broadcast(Message message): This method is used to send messages coming from the ObjectInputStream, if Server decides that the MessageType is a Chat Message then, it broadcasts the message to the other Clients.

#### 4.3.11.2 Class: Client

This class extends JFrame and implements Runnable.

* Operations
  + public Client(String clientName, serverIP): This constructor creates Client object with the client name(username) and the serverIP. In addition, it creates the chatting box for the user interaction with the server and the other users.
  + public static void main(String[] args): This is the main method for client class, it reads the client name and the server IP, then creates the Client object to establish connections.
  + public void run() : This method is the controller method for the Client. This method sustains the workflow of the connectToServer, setupStreams, and the whileChatting.
  + Private void setupStreams() throws IOException: Every client needs one output stream and one input stream to sustain the data flow between itself and the Server. Therefore, this method establishes these object and send the host name to the Chat Box.
  + private void whileChatting() : This method control the send button then read the messaging box in the chat Box, when the send is clicked, it uses the sendToServer method to send the message to the server via Output stream.
  + private void closeConnections() : This method works when the client leave the connection, it closes the input and output streams.
  + private void sendToServer(Message message) : This method sends the message object to the server with using the output stream.
  + private void showMessage(final String text): This method updates the Chat windows and show the messages to the user.
  + private void ableToType(boolean condition): This method controls the typing situation of the chat box, if the connection is not established yet, then it doesn’t give permission for typing.
  + public Socket getSocket(): This method returns the connection which is type of the Socket.

#### 4.3.11.3 Class: Utils

* Operations
  + public static float mapXCoordinateByZoom(final float x) : This methods works for the zooming operation. It maps an x coordinate by zoom.
  + public static float mapYCoordinateByZoom(final float y) : This methods works for the zooming operation. It maps a y coordinate by zoom.
  + public static DoaVectorF mapCoordinateByZoom(final float x, final float y) : Maps an x and a y coordinate by zoom. 2X zoom means x and y coordinates are now equal to 2\*x and 2\*y.
  + public static DoaVectorF mapCoordinateByZoom(final DoaVectorF coordinateToBeMapped) : Ditto, but by using a vector.
  + public static DoaVectorF mapMouseCoordinatesByZoom() : Maps the current mouse coordinate by zoom. As explained above.

#### 4.3.11.4 Class: Message

This class implements Serializable.

* Operations
  + Message(String, sender, String data, MessageType type): This constructor takes sender, data, and message type then, creates the message object.
  + public String getSender() : This method returns the sender of the Message object.
  + public String getData(): This method returns the data of the Message object.
  + public MessageType getType() : This method returns the MessageType of the Message object.

#### 4.3.11.5 Class: MessageType

* Operations
  + public MessageType(): This constructor specifies the MessageTypes and creates the MessageType object.

#### 4.3.11.6 Class: MessageBuilder

* Operations
  + public MessageBuilder(): This constructor determine the sender, data and the type of the message object.
  + public Message build() : This method builds a Message object and returns it.
  + public MessageBuilder setSender(String sender) : This method takes the sender of the message and assigns this sender to the message object, then return the Builder.
  + public MessageBuilder setData(String data) : This method takes the data of the message and assigns this data to the message object’s data, then return the Builder.
  + public MessageBuilder setType(MessageType type) : This method takes the type of the message and assigns this type to the message object’s type, then return the Builder.

#### 4.3.11.7 Class: Mesh2D

This class extends ArrayList<Vertex2D>.

* Operations
  + public Mesh2D(): Represents a Mesh, used in detecting mouse events on provinces.

#### 4.3.11.8 Class: Continent

This class implements Serializable.

* Operations
  + public Continent(): This constructor creates the Continents for the game back-end.
  + public String getName() : This method return the name of the continent.
  + public List<Province> getProvince() : This method returns the provinces in the selected continent.
  + public Continent setName(String name) : This method takes the name of the continent and assigns that name to the continent.
  + public Continet setProvinces(List<Provinces> provinces) : This method takes the provinces list and assigns this provinces list to selected continent.
  + public String toString() : This method prints the continents name and then, prints its provinces.
  + public static void printAllContinents() : This method prints all the continents to the console.

#### 4.3.11.9 Class: Province

* Operations
  + public Continent getContinent() : This method returns the continent that desired province is
  + public String getName() : This method returns the name of the province.
  + public List<Province> getNeighbours() : This method returns the neighbours of the province as a List.
  + public List<Mesh2D> getMeshes() : This method returns the Mesh list of the province.
  + public Province setContinent(Continent continent) : This method sets the continent for the province and returns the province.
  + public Province setName(String name : This method sets name to the desired province and returns the province.
  + public Province setNeighbours(List<Province> neighbours) : This method takes province list and assigns these provinces as the neighbor of the desired Province, then returns the province.
  + public Province setMeshes(List<Mesh2D> meshes) : This method takes a mesh list and assigns those meshes to the province.
  + public void addMesh(Mesh2D mesh) : This method takes the mesh and adds that mesh to the province’s mesh list.
  + public Stirng toString() : This method returns the continent of the province and the name of the province as a String.
  + public static void printAllProvinces() : This method prints all the provinces to the console.

#### 4.3.11.10 Class: AIPlayer

This class extends Player class.

* Operations
  + AIPlayer(String playerName, Color playerColor): This constructor takes player name and the player color and with these values it creates the one player as a AIPlayer.

#### 4.3.11.11 Class: Player

* Operations
  + public Player(String playerName, Color playerColor): This constructor assigns the player name and the player color then creates a new player object. In this process name and color must be unique thus, it controls that as well.
  + public Set<Province> getProvinces() : This method returns the province set of the selected player.
  + public boolean addProvince(Province p) : This method takes the province and adds that province to the player and return boolean value according to the operation result.
  + public Boolean removeProvince(Province p): This method takes the province and removes that province from the player and return boolean value according to the operation result.
  + public int getTroopsIn(Province p) : This method takes province and then, return the number of troops inside of the province p.
  + public String getName() : This method returns the name of the player.
  + public Color getColor() : This method returns the Color of the asked player as a Color object.

#### 4.3.11.12 Class: Card

* Operations
  + public Card(): This constructor creates a card object.

#### 4.3.11.13 Class : ProvinceHitArea

* Operations
  + public ProvinceHitArea(Province owner, Float x , Float y, Integer width, Integer height): This constructor takes the x, y ,width and the height values to determine the hit area of the province. Then using mesh it applies these boundaries to the hit area.
  + public void tick(): This method controls the mouse movements and determines that the mouse is on the hit area.
  + public void render(DoaGraphicsContext g): This method draws determined province hit area to the window. Hit area is the border of a province.

### 4.4.12 Exception Classes

#### 4.4.12.1 Class: DiceException

* Operations
  + public DiceException(String message): This constructor creates Dice exception with the entered message.

#### 4.4.12.2 Class: DiceAccessException

* Operations
  + public DiceAccessException(int size , int capacity): This constructor creates Dice Access Exception with the entered size and the capacity. Thrown instead of ArrayIndexOutOfBoundsException.

#### 4.4.12.3 Class: DiceInstantiationException

* Operations
  + public DiceInstantiationException(): This constructor creates DiceInstantiationException. Since Risk has 2 groups of dice (attack & defend) more than 2 dice is disallowed.

#### 4.4.12.4 Class: Risk Exception

This class extends RuntimeException

* Operations
  + public RiskException(): This constructor is the default constructor and directly creates the RiskException.
  + public RiskException(String s): This constructor creates RiskException with the message string s. Thrown instead of generalized RuntimeException.

#### 4.4.12.5 Class: RiskSingletoInstantiationException

This class extends RiskException

* Operations
  + public RiskSingletoInstantiationException():This constructor creates default RiskSingletoInstantiationException.
  + public RiskSingletoInstantiationException(String s): This constructor creates RiskSingletoInstantiationException with the message string s. Thrown when a singleton designed object is instantiated more than once.

#### 4.4.12.6 RiskStaticInstantiationException

This class extends RiskException.

* Operations
  + public RiskStaticInstantiationException():This constructor creates default RiskSingletoInstantiationException.
  + public RiskStaticInstantiationException(String s): This constructor creates RiskSingletoInstantiationException with the message string s. Thrown when a static class is tried to be instantiated.

# 5.0 Glossary & references

|  |  |
| --- | --- |
| [1] | D. Gordon, «TotalDiplomacy,» 4 October 2013. [Çevrimiçi]. Available: https://www.cardboardrepublic.com/classics/risk-vs-diplomacy. |
| [2] | UltraBoardGames, «Risk Game Rules,» [Çevrimiçi]. Available: https://www.ultraboardgames.com/risk/game-rules.php. [Accessed: 10 March 2019]. |