OOP Project

# Related image**Project Design Report: Iteration 2**

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

Project Group Members

* Doğa Oruç – 21602022
* Fazilet Simge Er - 21602358
* Kaan Ünlü - 21602622
* Ege Turan – 21502441
* Çağrı Orhan - 21503102

Contents

**[Project Design Report: Iteration 2](#_Toc7569264)** [1](#_Toc7569264)

[1.0 Introduction 4](#_Toc7569265)

[1.1 Purpose of the system 4](#_Toc7569266)

[1.2 Design goals 4](#_Toc7569267)

[2.0 High-level software architecture 5](#_Toc7569268)

[2.1 Subsystem decomposition 5](#_Toc7569269)

[2.1.1 The Player 6](#_Toc7569270)

[2.1.2 The Dice Panel 7](#_Toc7569271)

[2.2 Hardware/software mapping 8](#_Toc7569272)

[2.3 Persistent data management 8](#_Toc7569273)

[2.4 Access control and security 9](#_Toc7569274)

[2.5 Boundary conditions 9](#_Toc7569275)

[2.5.1 Initialization 9](#_Toc7569276)

[2.5.2 Termination 9](#_Toc7569277)

[2.5.3 Failure 10](#_Toc7569278)

[3.0 Subsystem services 10](#_Toc7569279)

[3.1 View 10](#_Toc7569280)

[3.1.1 The Game Board 10](#_Toc7569281)

[3.1.2 GUI View 10](#_Toc7569282)

[3.1.3 Card and Dice View 11](#_Toc7569283)

[3.2 Logic 11](#_Toc7569284)

[3.2.1 Turn Logic 11](#_Toc7569285)

[3.2.2 Province Logic 11](#_Toc7569286)

[3.2.3 Cards and Dice Logic 11](#_Toc7569287)

[3.2.4 AI Logic 11](#_Toc7569288)

[3.2.6 GUI Logic 13](#_Toc7569289)

[4.0 Low-level design 13](#_Toc7569290)

[4.1 Object design trade-offs 13](#_Toc7569291)

[4.2 Final object design 13](#_Toc7569292)

[4.3 Class Interfaces 14](#_Toc7569293)

[4.3.1 Package Game Logic 14](#_Toc7569294)

[4.3.2 Package Risk UI 19](#_Toc7569295)

[4.3.3 Package Exceptions 27](#_Toc7569296)

[4.3.4 Package Loaders & Globals 27](#_Toc7569297)

[4.3.5 Package Networking 29](#_Toc7569298)

[4.3.6 Classes without a package: 31](#_Toc7569299)

[5.0 Glossary & References 32](#_Toc7569300)

# 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
* 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 does not follow standard MVC conventions in that the model, the view, and the controller isn’t always separated into different classes. The main reason behind this is programming practicality. Most of the time, each object carries its entities (properties) and view (if present) in itself. Controllers whose effects are limited to the object itself are also inside the object, where controllers which affect multiple objects are governed in a game manager. This makes the access privileges of functions, properties and whatnot more intuitive, and makes the navigation of the code faster for us, since most things they’d need to look up while writing in a class are probably in that class.

Consider two random example code segments (cut):

### 2.1.1 The Player

//packages

//imports

public class Player extends DoaObject {

private static final long serialVersionUID = 1411773994871441922L;

public static final Map<String, Player> NAME\_PLAYER = new HashMap<>();

private static int number = 1;

private Color playerColor;

private String playerName;

protected boolean isInTurn;

private int id;

private boolean isLocalPlayer;

private Province source = null;

private Province destination = null;

public Player(String playerName, Color playerColor, boolean isLocalPlayer) {}

@Override

public void tick() {}

@Override

public void render(DoaGraphicsContext g) {}

public void turn() {}

public String getName() {}

public Color getColor() {}

public int getID() {}

public static int findStartingTroopCount(int numberOfPlayers) {}

public static int calculateReinforcementsForThisTurn(Player player) {}

public static List<Province> getPlayerProvinces(Player player) {}

public void endTurn() {}

public boolean isLocalPlayer() {}

}

The player is represented on the game board on the provinces they own based on their order and the colours they have chosen for themselves before entering the game: On the centre of each province is a ring that symbolises the continent that province is in. Inside the ring is the player’s symbol, determined by their loading order in the game selection menu. The province itself is also painted in the colour the player has chosen for themselves in the same game selection menu. As the player does not choose their symbol but their colour, symbol is not a player property, but the colour is, as seen in the above code segment.

The “tick” function is one of the two staple functions in the composition of this game. Tick harbours the local controller of an object and loops constantly (at every tick, as the name implies). For example, each player’s tick function constantly checks whether it’s their turn in the game or not. If it is the player’s turn, tick loops waiting for the player to click, after which it checks whether the clicked area is a province or not. After this, tick checks what phase of the turn it is, and calls the related game manager controller functions to achieve the results expected in every phase.

Conversely, as troop counts are meaningful only in the context of each player and not between, their controller functions are inside the player class.

The “render” function is the other staple function in the composition of the game. Render harbours the view of an object and like tick, loops constantly rendering its content. As the player object has nothing to render by itself, it is empty.

### 2.1.2 The Dice Panel

//packages

//imports

public class DicePanel extends DoaUIContainer {

private static final long serialVersionUID = 8009744806803376915L;

private static final float ACCELERATION = 0.064f;

private static final DoaVectorF MIN = new DoaVectorF(-160f, 258f);

private static final DoaVectorF MAX = new DoaVectorF(0f, 823f);

private DoaImageButton one = DoaHandler.instantiateDoaObject(DoaImageButton.class, -111f, 367f, 54, 60, DoaSprites.get("dice1Idle"), DoaSprites.get("dice1Hover"));

private DoaImageButton two = DoaHandler.instantiateDoaObject(DoaImageButton.class, -138f, 441f, 109, 62, DoaSprites.get("dice2Idle"), DoaSprites.get("dice2Hover"));

private DoaImageButton three = DoaHandler.instantiateDoaObject(DoaImageButton.class, -139f, 524f, 109, 86, DoaSprites.get("dice3Idle"), DoaSprites.get("dice3Hover"));

private DoaImageButton blitz;

private boolean moving = false;

public DicePanel() {}

@Override

public void tick() {}

@Override

public void show() {}

@Override

public void hide() {}

@Override

public void render(DoaGraphicsContext g) {}

}

The dice panel, unlike the player, is a representation by itself, of the dice controllers on the game screen. The dice panel physically is a scroll that is automatically dragged open when the situation calls for rolling dice in game, and is dragged back in, when its usage expires for the player’s turn.

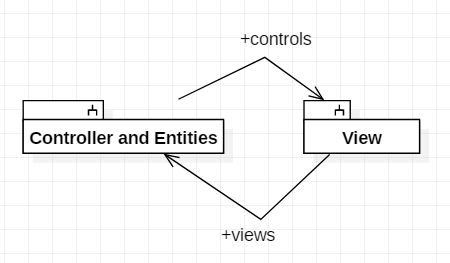
Due to that, the properties of this class are usually less entities and more either visual modifiers or UI element initializers. The dice buttons on the scroll for example, are prepared here by priming them with images.

The tick function, as with all objects, acts as the local controller of an object. As however the dice panel is primarily a view object, the local controller here only controls the movement of the scroll when called for.

Finally, to actually show the scroll itself, which is but an image, the render function renders the scroll at the coordinates determined by the tick function every logical frame.

Then, it can be seen that while the game doesn’t strictly follow MVC conventions, it does follow a loose architecture where some subsystems are more view oriented, such as the dice panel or any other GUI element, others are mostly a combination of entities and controllers, such as the player object and the game manager.

Thus, the subsystem decomposition can be summarized in a figure such as below:



While most objects are composed of entities and local controllers that build the game’s logic, most that have a view don’t have many other functions, if any at all. These view objects entities and controllers, where present, work only to help create the view itself.

Some other packages, such as Exceptions and Networking are hard to place on a diagram such as this however, as their properties do not classify as entities, they do not render anything, and do not add to the game logic.

## 2.2 Hardware/software mapping

Software-wise, as the game is implemented in Java, it can run in any environment with the JRE installed. 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 to 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. Due to time constraints.

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

## 2.3 Persistent data management

All data management is done 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 edited by the player like 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 as 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. If time permits, we are thinking of implementing an encryption algorithm for IP addresses, so that the server hosts can give their encrypted IP address, and clients’ game clients will decrypt the encrypted IP address.

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.

|  |  |  |
| --- | --- | --- |
|  | 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.

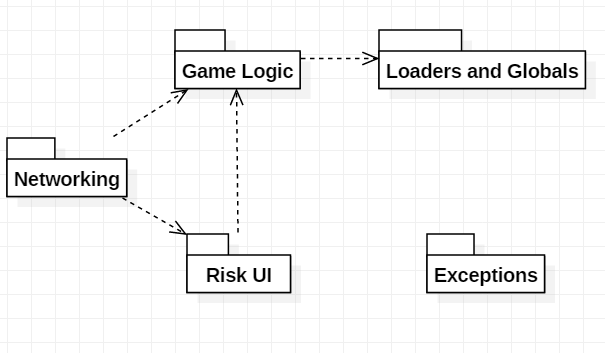
### 2.5.2 Termination

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 order to minimise data loss, we will implement an auto-save feature to our game. 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 the 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 are present in the actual board. The game board is connected with the logic of the game so that we can update the view of the board when a change occurs; for example, when a user attacks an enemy province and wins the battle which means ownership of the province had changed. Game board changes the 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 in 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. There are six types of AI and they will act as explained below:

#### 3.2.4.1 Passive AI

As the name implies, the passive AI does as less movements as possible. This AI was used for testing of the game and can be useful in the learning process of the players.

In the beginning of the game and at the draft phase, this AI places its soldiers randomly. It simply skips the attack and reinforcement phases without doing any actions.

#### 3.2.4.2 Easy AI

The easy AI is similar to the passive AI. It can be played by inexperienced player.

In the beginning of the game and at the draft phase, this AI places its soldiers randomly, just like the passive AI. At the attack phase, it attacks with the provinces it owns if those provinces have more than ten soldiers on them and a neighbouring province to that province has at least five less soldiers on. It skips the reinforcement phase.

#### 3.2.4.3 Medium AI

The medium AI is like an average player. It is recommended for most of the players.

In the beginning of the game, medium AI claims a random province, and continues to claim the neighbours of the provinces it owns. At the draft phase, it places its soldiers randomly to the provinces it owns which it doesn’t have all the neighbouring provinces. At the attack phase, it attacks with the provinces it owns if those provinces have more than ten soldiers on them and a neighbouring province to that province has at least five less soldiers on. It reinforces one of the border provinces from a province which it has all the neighbours of, and it has more than 4 soldiers on, if that kind of province is present.

#### 3.2.4.4 Hard AI

The hard AI is a lot more advanced than the other AI’s mentioned above. It is recommended for experienced players.

In the beginning of the game, hard AI claims one of the provinces which has the less adjacencies and continues to claim the neighbours of the provinces it owns. At the draft phase, it places its soldiers to the provinces it owns which it doesn’t have all the neighbouring provinces. At the attack phase, it attacks with the provinces it owns if those provinces have more than ten soldiers on them and a neighbouring province to that province has at least five less soldiers on. It priorities some provinces if by claiming it, it will claim a whole continent. It reinforces one of the border provinces from a province which it has all the neighbours of, and it has more than 4 soldiers on, if that kind of province is present.

#### 3.2.4.5 Insane AI

The insane AI is the hardest fair AI our game has. It is recommended for players who are looking for a challenge. In the beginning of the game, insane AI claims one of the provinces which has the less adjacencies and continues to claim the neighbours of the provinces it owns, from the continent which has the smaller number of provinces. At the draft phase, it places its soldiers to the provinces it owns which it doesn’t have all the neighbouring provinces. If the AI has almost every province in a continent, the provinces there will be prioritized. At the attack phase, it attacks with the provinces it owns if those provinces have more than ten soldiers on them and a neighbouring province to that province has at least five less soldiers on. It also calculates the percentage of claiming the province it is planning to attack, and acts accordingly by choosing not to attack it the percentage is lower than 70%. It priorities some provinces if by claiming it, it will claim a whole continent. It reinforces one of the border provinces from a province which it has all the neighbours of, and it has more than 4 soldiers on, if that kind of province is present.

#### 3.2.4.6 Cheater AI

The cheater AI is an unfair and unpredictable AI. It is only recommended for players who don’t get flustered easily.

The cheater AI cheats randomly during the whole game. It places double soldiers, attacks with more than three soldiers, does more than one reinforcement in one turn etc.

### 3.2.6 GUI Logic

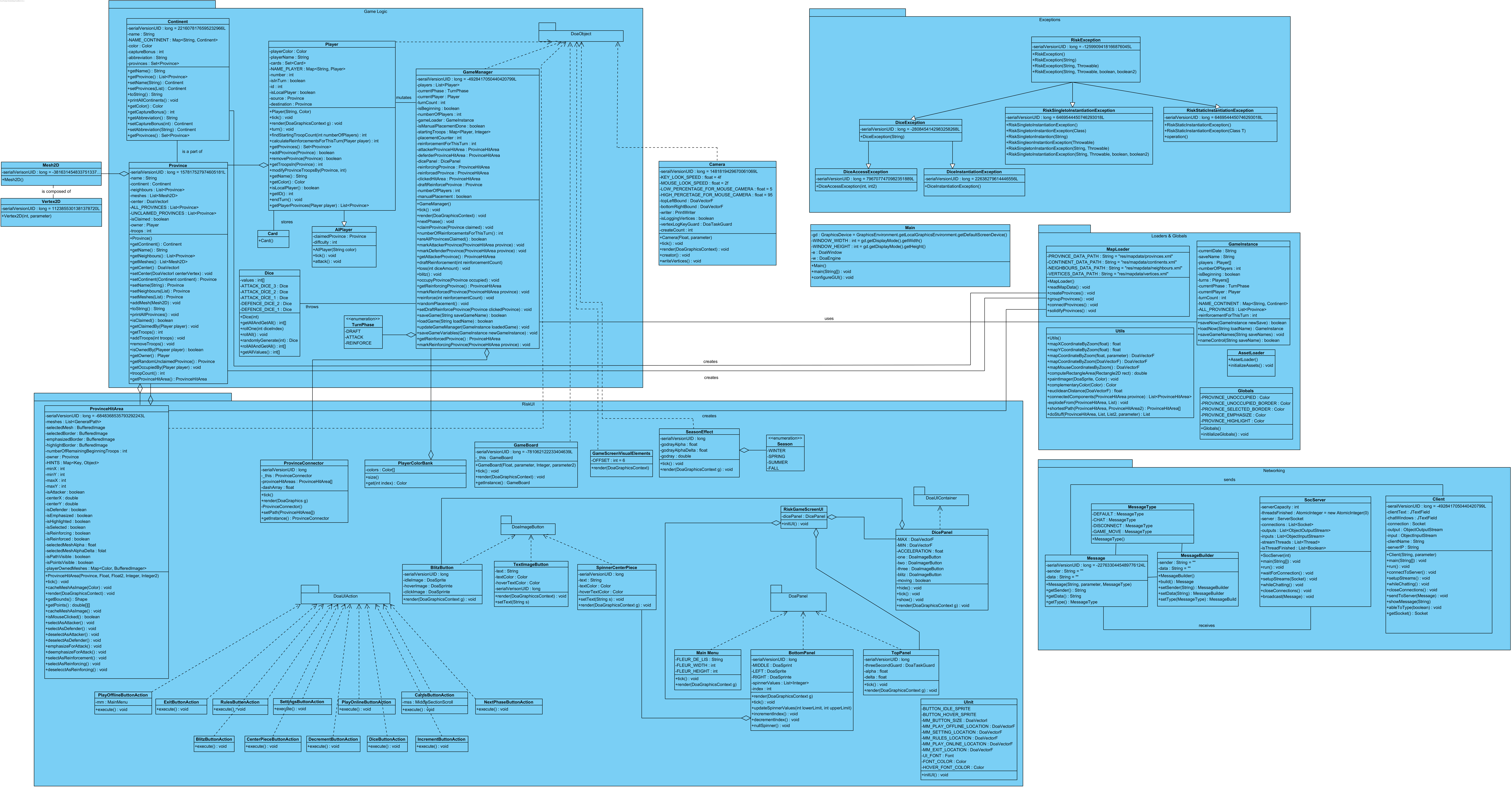
GUI logic determines what GUI view is going to draw by supplying data. It processes data that is needed for GUI view like where to place a button, where that button sends the user to when pressed and etc. It sends the data to the GUI view so that a GUI view will be able to draw images and locate the 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 Package Game Logic

#### Class GameManager

* + - Attributes
      * players: List<Player>
        + The list of players that are currently in the game.
      * currentPhase: TurnPhase
        + Represents the current phase of the turn of the playing player.
      * currentPlayer: Player
        + Represents the player that is currently playing.
      * turnCount: int
        + How many turns have passed since beginning.
      * numberOfPlayers: int
        + The amount of players that are current in the game.
      * gameLoader: GameInstance
        + A game save/load object.
      * isManualPlacementDone: boolean
        + Whether or not the players have completed the manual placement of their troops.
      * startingTroops: Map<Player, Integer>
        + Map that maps a player to their starting troop count. When all players reach 0, the game can begin.
      * placementCounter: int
        + A simple counter to keep track of which player will place their troops next on the placement phase of the game.
      * reinforcementForThisTurn: int
        + The amount of reinforcement the current player will gain when it’s their turn.
      * attackerProvinceHitArea: ProvinceHitArea
        + The province that will attack somewhere.
      * defenderProvinceHitArea: ProvinceHitArea
        + The province attacked by attackerProvinceHitArea
      * dicePanel: DicePanel
        + A UI panel for throwing dice.
      * reinforcingProvince: ProvinceHitArea
        + The province that will be the source of reinforcement in reinforce phase.
      * reinforcedProvince: ProvinceHitArea
        + The province that will be the destination of reinforcement in reinforce phase.
      * clickedHitArea: ProvinceHitArea
        + The province the player clicked.
      * draftReinforceProvince: Province
        + The province that will get draft phase reinforcements
      * manualPlacement: boolean
        + If false, game will place troops randomly in the beginning of the game, else players will do.
    - Operations
      * GameManager()
        + Constructor
      * tick(): void
        + Update method of this object
      * render(DoaGraphicsContext): void
        + Empty method, exists because DoaEngine wants it.
      * nextPhase(): void
      * claimProvince(Province): void
        + Gives the passed province to the current playing player.
      * numberOfReinforcementsForThisTurn(): int
        + Returns numberOfReinforcementsForThisTurn.
      * areAllProvincesClaimed(): boolean
        + Self-explanatory.
      * markAttackerProvince(ProvinceHitArea): void
        + Marks the attacker province.
      * markDefenderProvince(ProvinceHitArea): void
        + Marks the defender province.
      * getAttackerProvince(): ProvinceHitArea
        + Returns the attacker province.
      * draftReinforcement(int): void
        + Puts the passed amount of troops to draftReinforcementProvince.
      * toss(int): void
        + Tosses int amount of die.
      * blitz(): void
        + Tosses infinite amount of die. Stops when unable to do so.
      * occupyProvince(Province): void
        + Gives the passed province to the current playing player.
      * getReinforcingProvince(): ProvinceHitArea
        + Returns the reinforcing province.
      * markReinforcedProvince(ProvinceHitArea): void
        + Marks the reinforced province.
      * reinforce(int): void
        + Takes passed amount of troops from reinforcing province and puts them into reinforced province.
      * randomPlacement(): void
        + Randomly places all starting troops to the map.
      * setDraftReinforceProvince(Province): void
        + Sets the draft reinforcement province.
      * saveGame(String): boolean
        + Saves the game.
      * loadGame(String): boolean
        + Loads the game.
      * updateGameManager(GameInstance): void
        + Updates the game manger.
      * saveGameVariables(GameInstance): void
        + Saves the game variables.
      * getReinforcedProvince(): ProvinceHitArea
        + Returns the reinforced province.
      * markReinforcingProvince(ProvinceHitArea): void
        + Marks the reinforcing province.

#### Enum TurnPhase

* + - Attributes
      * DRAFT
        + Represents the draft phase of a turn.
      * ATTACK
        + Represents the attack phase of a turn.
      * REINFORCE
        + Represents the reinforcement phase of a turn.

#### Class Dice

* + - Attributes
      * values: int[]
        + Face values of the dice.
      * ATTACK\_DICE\_3: Dice
        + A triplet of dice that will be used for attacking.
      * ATTACK\_DICE\_2: Dice
        + A pair of dice that will be used for attacking.
      * ATTACK\_DICE\_1: Dice
        + A single die that will be used for attacking.
      * DEFENCE\_DICE\_2: Dice
        + A pair of dice that will be used for defending.
      * DEFENCE\_DICE\_1: Dice
        + A single die that will be used for defending.
    - Operations
      * Dice(int)
        + Constructor.
      * rollAllAndGetAll(): int[]
        + Rolls all the dice and returns their face values.
      * rollOne(int): void
        + Roll the die at the passed index.
      * rollAll(): void
        + Rolls all dice.
      * randomlyGenerate(int): Dice
        + Create and roll passed amount of dice.
      * getAllValues(): int[]
        + Returns the face values of dice.

#### Class Player

* + - Attributes
      * playerColor: Color
        + Color of the player.
      * playerName: String
        + Name of the player.
      * cards: Set<Card>
        + Cards the player has.
      * NAME\_PLAYER: Map<String, Player>
        + Map that maps the name of the player to the player.
      * number: int
        + Static variable for ID assignment.
      * isInTurn: boolean
        + True if it’s this player’s turn, false otherwise.
      * id: it
        + ID of the player.
      * isLocalPlayer: boolean
        + True if this player is not connected via a socket connection or is an AI Player, false otherwise.
    - Operations
      * Player(String, Color)
        + Constructor.
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Empty method.
      * turn(): void
        + Sets the isInTurn variable to true.
      * findStartingTroopCount(int): int
        + Returns the amount of troops that this player will have at the beginning of the game.
      * calculateReinforcementsForThisTurn(Player): int
        + Returns the amount of troops a player will get when their turn comes.
      * getProvinces(): Set<Province>
        + Returns the province owned by this player.
      * addProvince(Province): boolean
        + Adds a province to this players province set.
      * removeProvince(Province): boolean
        + Removes a province from this players province set.
      * getTroopsIn(Province): int
        + Returns the amount of troops inside a province.
      * getName(): String
        + Returns the name of the player.
      * getColor(): Color
        + Returns the color of the player.
      * isLocalPlayer(): boolean
        + Returns isLocalPlayer variable.
      * getID(): int
        + Returns the id of the player.
      * endTurn(): void
        + Sets the isInTurn variable to false.

#### Class AIPlayer

* + - Attributes
      * difficulty: int
        + An int denoting the difficulty level of the AI, higher is harder.
    - Operations
      * AIPlayer(String, color, int)
        + Constructor.
      * tick(): void
        + Update method of this object.
      * attack(): void
        + Determines the kind of attack action to be performed depending on the AI’s difficulty, and performs it.

#### Class Card

* + - Operations
      * Card()
        + Constructor.

#### Class Continent

* + - Attributes
      * name: String
        + Name of the continent.
      * NAME\_CONTINENT: Map<String, Continent>
        + Map that maps the name of a continent to a continent.
      * color: Color
        + Color of the continent.
      * captureBonus: int
        + Capture bonus of the continent.
      * abbreviation: String
        + Abbreviation of the continent.
      * provinces: Set<Province>
        + Province that belong to this continent.
    - Operations
      * getName(): String
        + Returns the name of the continent.
      * getProvinces(): Set<Province>
        + Returns the provinces of the continent.
      * setName(String): Continent
        + Sets the name of the continent.
      * SetProvinces(List): Continent
        + Sets the provinces of the continent.
      * getColor(): Color
        + Returns the color of the continent.
      * getCaptureBonus(): int
        + Returns the capture bonus of the continent.
      * getAbbreviation(): String
        + Returns the abbreviation of the continent.
      * setCaptureBonus(int): Continent
        + Sets the capture bonus of the continent.
      * setAbbreviation(String): Continent
        + Set sthe abbreviation of the continent.

#### Class Province

* + - Attributes
      * name: String
        + Name of the province.
      * continent: Continent
        + Continent the province belongs to.
      * neighbours: List<Province>
        + List of the provinces neighbouring the province
      * meshes: List<Mesh2D>
        + Physical representation of the province
      * center: DoaVectorI
        + Centre coordinates for the province
      * ALL\_PROVINCES: List<Province>
        + List of all provinces in the game
      * UNCLAIMED\_PROVINCES: List<Province>
        + List of all unclaimed provinces in the game
      * isClaimed: boolean
        + Flag showing whether the province is claimed or not
      * owner: Player
        + Which player owns the province
      * troops: int
        + The number of troops the province houses
    - Operations
      * Province()
        + Constructor
      * getContinent(): Continent
        + Gets the continent the province is in
      * getName(): String
        + Gets the name of the province
      * getNeighbours(): List<Province>
        + Gets a list of all neighbours the province has
      * getMeshes(): List<Mesh2D>
        + Gets the physical representation of the province
      * getCenter(): DoaVectorI
        + Gets the centre coordinates for the province
      * setCenter(DoaVectorI): void
        + Sets centre coordinates for the province
      * setContinent(Continent): Province
        + Sets the continent the province belongs to
      * setName(String): Province
        + Sets the name of the province
      * setNeighbours(List<Province>): Province
        + Sets a list of provinces as neighbours for the province
      * setMeshes(List<Mesh2D>): Province
        + Sets a physical representation for the province
      * addMesh(Mesh2D): void
        + Adds a new mesh to the existing physical representation of the province
      * toString(): String
        + Prints object hashcode
      * printAllProvinces(): void
        + Prints all of the provinces
      * isClaimed(): boolean
        + Returns whether the province belongs to someone or not
      * getClaimedBy(Player): void
        + Gives the province to a player at the start of the game
      * getTroops(): int
        + Gets the number of troops housed on the province
      * addTroops(int): void
        + Adds a number of troops to the current garrison on the province
      * removeTroops(int): void
        + Removes a number of troops from the current garrison on the province
      * isOwnedBy(Player): boolean
        + Checks whether the province is owned by a certain player or not
      * getOwner(): Player
        + Gets the player that owns the province
      * getRandomUnclaimedProvince(): Province
        + Gets a random province that doesn’t belong to anyone
      * getOccupiedBy(Player): void
        + Gives the province to a player during an attack
      * getProvinceHitArea(): ProvinceHitArea
        + Gets the hit area of the province on the game map

### 4.3.2 Package Risk UI

#### Class BlitzButton

* + - Attributes
      * idleImage: DoaSprite
        + The image that will be rendered while the button is not being interacted by the user.
      * hoverImage: DoaSprite
        + The image that will be rendered while the button is hovered.
      * clickImage: DoaSprite
        + The image that will be rendered while the button is clicked.
    - Operations
      * render(DoaGraphicsContext): void
        + Method that renders the button on to the screen.

#### Class TextImageButton

* + - Attributes
      * text: String
        + The text this button will display.
      * textColor: Color
        + Color of the text this button will display.
      * hoverTextColor: Color
        + Color of the text when this button is hovered.
    - Operations
      * render(DoaGraphicsContext): void
        + Method that renders the button on to the screen.
      * setText(String): void
        + Sets the text this button will display

#### Class SpinnerCenterPiece

* + - Attributes
      * text: String
        + The text this button will display.
      * textColor: Color
        + Color of the text this button will display.
      * hoverTextColor: Color
        + Color of the text when this button is hovered.
    - Operations
      * render(DoaGraphicsContext): void
        + Method that renders the button on to the screen.
      * setText(String): void
        + Sets the text this button will display

#### Class RiskGameScreenUI

* + - Attributes
      * dicePanel: DicePanel
        + The panel that is used to throw dice.
    - Operations
      * initUI(): void
        + Initializes the UI elements.

#### Class DicePanel

* + - Attributes
      * MAX: DoaVectorF
        + How much the dice panel pans into the screen
      * MIN: DoaVectorF
        + How much the dice panel pans out of the screen
      * ACCELERATION: float
        + How fast the panel speeds up while panning in or out of the screen
      * one: DoaImageButton
        + Button for when a single die is to be rolled
      * two: DoaImageButton
        + Button for when two dice are to be rolled
      * three: DoaImageButton
        + Button for when three dice are to be rolled
      * blitz: DoaImageButton
        + Button for when dice are to be rolled nonstop until either province runs out of usable troops
      * Moving: boolean
        + Flag to check if the panel is moving
    - Operations
      * hide(): void
        + Hides the dice panel.
      * tick(): void
        + Update method of this object.
      * show(): void
        + Unhides the dice panel.
      * render(DoaGraphicsContext): void
        + Method that draws this object onto the screen.

#### Class TopPanel

* + - Attributes
      * threeSecondGuard: DoaTaskGuard
        + A conditional that only becomes true once every 3 seconds for one tick.
      * alpha: float
        + Alpha of the component.
      * delta: float
        + Delta of the alpha of the component.
    - Operations
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Method that draws this object onto the screen.

#### Class BottomPanel

* + - Attributes
      * MIDDLE: DoaSprite
        + A sprite.
      * LEFT: DoaSprite
        + A sprite.
      * RIGHT: DoaSprite
        + A sprite.
      * spipnnerValues: List<Integer>
        + Values of the spinner.
      * index: int
        + Index of the spinner.
    - Operations
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Method that draws this object onto the screen.
      * updateSpinnerValues(int, int): void
        + Updates the spinner values, first parameter is min, second is max.
      * incrementIndex(): void
        + Increments the index.
      * decrementIndex(): void
        + Deccrements the index.
      * nullSpinner(): void
        + Sets the spinner values to null.

#### Class MainMenu

* + - Attributes
      * FLEUR\_DE\_LIS: String
        + Texture name.
      * FLEUR\_WIDTH: int
        + Texture width.
      * FLEUR\_HEIGHT: int
        + Texture height.
    - Operations
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Method that draws this object onto the screen.

#### Class UIInit

* + - Attributes
      * BUTTON\_IDLE\_SPRITE
        + A Sprite.
      * BUTTON\_HOVER\_SPRITE
        + A Sprite.
      * MM\_BUTTON\_SIZE: DoaVectorI
        + Main menu button size.
      * MM\_PLAY\_OFFLINE\_LOCATION: DoaVectorF
        + Play offline button location.
      * MM\_PLAY\_ONLINE\_LOCATION: DoaVectorF
        + Play online button location.
      * MM\_SETTING\_LOCATION: DoaVectorF
        + Settings button location.
      * MM\_RULES\_LOCATION: DoaVectorF
        + Rules button location.
      * MM\_EXIT\_LOCATION: DoaVectorF
        + Exit button location.
      * UI\_FONT: Font
        + Global UI Font.
      * FONT\_COLOR: Color
        + Global Font Color.
      * HOVER\_FONT\_COLOR: Color
        + Button Font color when hovered.
    - Operations
      * initUI(): void
        + Initializes the UI elements.

#### Class ProvinceHitArea

* + - Attributes
      * meshes : List<GeneralPath>
        + The list of GeneralPaths generated by using the meshes of every province.
      * selectedMesh : BufferedImage
        + The texture that will be rendered when the mesh is selected.
      * selectedBorder : BufferedImage
        + The texture that will be rendered when the mesh is selected.
      * emphasizedBorder : BufferedImage
        + The texture that will be rendered when the mesh is emphasized.
      * highlightBorder : BufferedImage
        + The texture that will be rendered when the mesh is highlighted.
      * owner : Province
        + The province which owns that ProvinceHitArea.
      * HINTS : Map<RenderingHints.Key, Object>
        + RenderingHints for the graphics object.
      * minX : int
        + The minimum x value of the province
      * minY : int
        + The minimum y value of the province
      * maxX : int
        + The maximum x value of the province
      * maxY : int
        + The maximum y value of the province
      * isAttacker : boolean
        + True if this province is about to attack another province.
      * centerX : double
        + The center x value of the province
      * centerY : double
        + The center y value of the province
      * isDefender : boolean
        + True if this province is about to defend an attack from another province.
      * isEmphasized : boolean
        + True if this province is emphasized.
      * isHighlighted : boolean
        + True if this province is highlighted.
      * isSelected : boolean
        + True if this province is selected.
      * isReinforcing : boolean
        + True if this province is about to reinforce another province.
      * isReinforced : boolean
        + True if this province is about to be reinforced by another province.
      * selectedMeshAlpha : float
        + The transparency of the selected province’s visual effect.
      * selectedMeshAlphaDelta : float
        + The speed of change in the transparency of the selected province’s visual effect.
      * playerOwnedMeshes : Map<Color, BufferedImage>
        + It maps the province’s with their owner’s colors.
    - Operations:
      * ProvinceHitArea(Province)
        + Constructor.
      * tick() : void
        + Update method of this oobject.
      * render(DoaGraphicsContext) : void
        + Method that draws this object to the screen.
      * cacheMeshAsImage() : void
        + In order to gain performance, we cache the meshes.
      * isMouseClicked() : boolean
        + Returns true if the user clicks this province.
      * selectAsAttacker() : void
        + Selects the clicked province as the attacker province.
      * selectAsDefender() : void
        + Selects the clicked province as the defender province.
      * deselectAsAttacker() : void
        + Deselects the attacker province if that province is no longer the attacker province.
      * deselectAsDefender() : void
        + Deselects the defender provinceif that province is no longer the defender province.
      * emphasizeForAttack() : void
        + Emphasizes the provinces which can be attacked.
      * deemphasizeForAttack() : void
        + Deemphasizes the provinces which can be attacked if the attacker province is no longer the attacker province.
      * selectAsReinforcing() : void
        + Selects the clicked province as the reinforcing province.
      * selectAsReinforced() : void
        + Selects the clicked province as the reinforced province.
      * deselectAsReinforcing() : void
        + Deselects the reinforcing province if that province is no longer the reinforcing province.
      * deselectAsReinforced() : void
        + Deselects the reinforced province if that province is no longer the reinforced province.

#### Class ProvinceConnector

* + - Attributes:
      * \_this : ProvinceConector
        + Single instance of this object.
      * provinceHitAres : ProvinceHitArea[]
        + Path this object will render.
      * dashArray : float
        + A variable to draw dashed line.
    - Operations:
      * ProvinceConnector()
        + Constructor.
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Method that draws a dashed line onto the screen.
      * setPath(ProvinceHitArea…) : void
        + Sets the path.
      * getInstance() : ProvinceConnectors
        + Returns the single instance of this object.

#### Class PlayerColorBank

* + - Attributes:
      * colors : Color[]
        + Possible colors of players.
    - Operations:
      * size() : int
        + Size of the list.
      * get(int) : Color
        + Getter.

#### Class GameBoard

* + - Attributes:
      * \_this : GameBoard
        + Single instance of this object.
    - Operations:
      * GameBoard()
        + Constructor.
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Method that draws a the world map onto the screen.
      * getInstance() : GameBoard
        + Returns the single instance of this object.

#### Class GameScreenVisual

* + - Attributes:
      * OFFSET : int
        + Offset of the visual elements from the borders of the screen.
    - Operations:
      * render(DoaGraphicsContext) : void
        + Method that draws this object onto the screen.

#### Class SeasonEffect

* + - Attributes:
      * godrayAlpha : float
        + Transparency value of the godray sprite
      * godrayAlphaDelta : float
        + The rate of change of the transparency value of the godray sprite
      * godrayAngle : double
        + Current angle of the godray sprite
    - Operations:
      * tick(): void
        + Update method of this object.
      * render(DoaGraphicsContext): void
        + Method that draws this object onto the screen.

#### Enum Season

* + - Attributes:
      * WINTER
      * SPRING
      * SUMMER
      * FALL
        + Enum showing the individual seasons

#### Class PlayOfflineButtonAction

* + - Attributes:
      * mm: MainMenu
        + An instance of the main menu
    - Operations:
      * execute() : void
        + Takes the user to the Play Offline Menu

#### Class PlayOnlineButtonAction

* + - Operations:
      * execute() : void
        + Takes the user to the Play Online Menu

#### Class RulesButtonAction

* + - Operations:
      * execute() : void
        + Takes the user to the Rules Menu

#### Class SettingsButtonAction

* + - Operations:
      * execute() : void
        + Takes the user to the Settings Menu

#### Class ExitButtonAction

* + - Operations:
      * execute() : void
        + Takes the user to the Exit Menu

#### Class NextPhaseButtonAction

* + - Operations:
      * execute() : void
        + Advances the player to the next phase in their turn

#### Class BlitzButtonAction

* + - Operations:
      * execute() : void
        + Rolls dice for attacker until unable to do so.

#### Class CenterPieceButtonAction

* + - Operations:
      * execute() : void
        + Solidifies troop count choice where relevant

#### Class DecrementButtonAction

* + - Operations:
      * execute() : void
        + Decrements the troop count on the troop count spinner

#### Class IncrementButtonAction

* + - Operations:
      * execute() : void
        + Increments the troop count on the troop count spinner

#### Class DiceButtonAction

* + - Attribute:
      * diceCount : int
        + The number of dice that will be rolled when that dice button is pressed
    - Operations:
      * DiceButtonAction()
        + Constructor
      * execute() : void
        + Rolls diceCount amount of dice for an attack.

### 4.3.3 Package Exceptions

#### Class RiskException

* + - Operations:
      * RiskException()
        + Constructor.
      * RiskException(String)
        + Constructor.
      * RiskException(String, Throwable)
        + Constructor.
      * RiskException(String, Throwable, boolean, boolean)
        + Constructor.

#### Class DiceException

* + - Operations:
      * DiceException(String)
        + Constructor.

#### Class DiceAccessException

* + - Operations:
      * DiceAccessException(int, int)
        + Constructor.

### 4.3.4 Package Loaders & Globals

#### Class MapLoader

* + - Attributes:
      * PROVINCE\_DATA\_PATH : String
        + Path to retrieve the province data.
      * CONTINENT\_DATA\_PATH : String
        + Path to retrieve the continent data.
      * NEIGHBOURS\_DATA\_PATH : String
        + Path to retrieve the graph data.
      * VERTICES\_DATA\_PATH : String
        + Path to retrieve the vertex data.
    - Operations:
      * MapLoader()
        + Constructor.
      * readMapData() : void
        + Reads the map data.
      * createProvinces() : void
        + Reads the province data and creates provinces.
      * groupProvinces() : void
        + Reads the continent data and groups provinces.
      * connectProvinces() : void
        + Reads the graph data and connects provinces.
      * solidifyProvince() : void
        + Reads the vertex data and creates hitboxes for provinces.

#### Class Utils

* + - Operations:
      * Utils()
        + Constructor.
      * mapXCoordinateByZoom(float) : float
        + Maps the X coordinate by zoom.
      * mapYCoordinateByZoom(float) : float
        + Maps the Y coordinate by zoom.
      * mapCoordinateByZoom(float, float) : DoaVectorF
        + Maps a point by zoom.
      * mapCoordinateByZoom(DoaVectorF) :DoaVectorF
        + Maps a point by zoom.
      * mapMouseCoordinatesByZoom() : DoaVectorF
        + Maps the mouse pointer point by zoom.
      * computeRectangleArea(Rectangle2D) : double
        + Computes a rectangles area.
      * paintImage(DoaSprite, Color) : void
        + Paints a cingle color image.
      * complementaryColor(Color) : Color
        + Returns the complementary color of the argument.
      * euclideanDistance(DoaVectorF, DoaVectorF) : float
        + Computes euclidean distance between two points.
      * connectedComponents(ProvinceHitArea) : List<ProvinceHitArea>
        + Finds the connected ally components of the passed province.
      * explodeFrom(ProvinceHitArea, List<ProvinceHitArea>) : void
        + Modified version of floodfill algorithm. Recurses into every possible direction.
      * shortestPath(ProvinceHitArea, ProvinceHitArea) : ProvinceHitArea[]
        + Computes all the paths and finds the shortest path between two provinces.

#### Class AssetLoader

* + - Operations:
      * AssetLoader()
        + Constructor.
      * initializeAssets() : void
        + Loads all assets into the game’s memory.

#### Class Globals

* + - Attributes:
      * PROVINCE\_UNOCCUPIED : Color
        + Colour value for an unoccupied province
      * PROVINCE\_UNOCCUPIED\_BORDER : Color
        + Colour value for an unoccupied border
      * PROVINCE\_SELECTED\_BORDER : Color
        + Colour value for a selected border
      * PROVINCE\_EMPHASIZED : Color
        + Colour value for a province border that can be attacked/reinforced
      * PROVINCE\_HIGHLIGHT : Color
        + Colour value for the hover effect on a province
    - Operations:
      * Globals()
        + Constructor.
      * initializeGlobals() : void
        + Initialize globals.

#### Class GameInstance

* + - Attributes:
      * currentDate : String
        + The date when the game was saved.
      * players : player[]
        + List of player in that game instance.
      * numberOfPlayers : int
        + Number of players in that game instance.
      * currentPhase : TurnPhase
        + Current turn phase in that game instance.
      * currentPlayer : Player
        + The player who has the current turn in that game instance.
      * turnCount : int
        + Turn count in that game instance.
      * NAME\_CONTINENT : Map<String, Continent>
        + Map that maps continent names to actual continents in that game instance
      * ALL\_PROVINCES : List<Province>.
        + List of all provinces in that game instance.
      * reinforcementForThisTurn : int
        + The number of troops a player will receive for deployment at the start of their turn in that game instance.
    - Operations:
      * saveNow(GameInstance) : boolean
        + Game instance values are serialized into a save file
      * loadNow(String) : GameInstance
        + Game instance values are deserialized from file a game from this information is booted

### 4.3.5 Package Networking

#### Class Message

* + - Attributes:
      * sender : String
        + Name of the sender of the message
      * data : String
        + Data to be sent in the message
    - Operations:
      * Message(String, String, MessageType)
        + Message constructor
      * getSender() : String
        + Gets sender name
      * getData() : String
        + Gets the data
      * getType() : MessageType
        + Gets the type of the message

#### Enum MessageType

* + - Attributes:
      * DEFAULT
        + Ignore message
      * CHAT
        + Chat information is coming from the server
      * DISCONNECT
        + A player is disconnecting from the server
      * GAME\_MOVE
        + A player move is coming from the server

#### Class MessageBuilder

* + - Attributes:
      * sender : String
        + Sender of this message.
      * data : String
        + The actual message.
    - Operations:
      * MessageBuilder()
        + Constructor.
      * build() : Message
        + Builds the message.
      * setSender() : MessageBuilder
        + Sets the sender of the message.
      * setData() : MessageBuilder
        + Sets the message this object is carrying.
      * setType() : MessageBuilder
        + Sets the type of the message.

#### Class SocServer

* + - Attributes:
      * serverCapacity : int
        + Capacity of the server
      * threadsFinished : AtomicInteger
        + Number of joined threads
      * server : ServerSocket
        + The server itself
      * connections : List<Socket>
        + The list of objects that connect the clients to the server
      * outputs : List<ObjectOutputStream>
        + List of data sent to buffer for the server to read from
      * inputs : List<ObjectInputStream>
        + List of data sent to buffer by the server
      * streamThreads : List<Thread>
        + Each stream needs a different thread because streams are blocking.
      * isThreadFinished : List<Boolean>
        + A list of flags showing whether each thread has joined or not.
    - Operations:
      * SocServer(int)
        + Constructor.
      * run() : void
        + Kickstarts the server
      * waitForConnection() : void
        + Server waits for connections from clients
      * setupStreams(Socket) : void
        + Initialize input & output buffers.
      * whileChatting() : void
        + Loops forever, reads from all the input buffers.
      * closeConnections() : void
        + Shuts down the server.
      * broadcast(Message) : void
        + Sends a message to all clients.

#### Class Client

* + - Attributes:
      * connection: Socket
        + Object that connects the client to the server
      * output :ObjectOutputStream
        + Data sent to buffer for the server to read from
      * input :ObjectInputStream
        + Data sent to buffer by the server
      * clientName : String
        + Name of the client
      * serverIP : String
        + Server’s IP
    - Operations:
      * Client(String, String)
        + Constructor.
      * run() : void
        + Kickstarts the client
      * connectToServer() : void
        + Connect the client to the server
      * setupStreams() : void
        + Initialize input & output buffers.
      * whileChatting() : void
        + Loops forever, reads from the input buffer.
      * closeConnections() : void
        + Disconnects from the server.
      * sendToServer(Message) : void
        + Sends a message to the server.
      * getSocket() : Socket
        + Returns the socket object.

### 4.3.6 Classes without a package:

#### Class Main

* + - Attributes:
      * gd : GraphicsDevice
        + The object that keeps the user’s monitor.
      * WINDOW\_WIDTH : int
        + Width of the window.
      * WINDOW\_HEIGHT : int
        + Height of the window.
      * e : DoaEngine
      * w : DoaWindow
    - Operations:
      * main(String[] args)
        + Entrance point of the application.
      * configureGUI() : void
        + Creates a window.

#### Class Camera

* + - Attributes:
      * KEY\_LOOK\_SPEED : float
        + Speed of scroll while using the arrow keys.
      * MOUSE\_LOOK\_SPEED: float
        + Speed of scroll while using the middle mouse mouse button.
      * LOW\_PERCENTAGE\_FOR\_MOUSE\_CAMERA: float
        + The lowest percentage for the mouse zoom.
      * HIGH\_PERCENTAGE\_FOR\_MOUSE\_CAMERA: float
        + The highest percentage for the mouse zoom.
      * topLeftBound : DoaVectorF
        + Top left corner of the map.
      * bottomRightBound : DoaVectorF
        + Bottom right corner of the map.
    - Operations:
      * Camera()
        + Constructor.
      * tick() : void
        + Update method of this object.
      * render(DoaGraphicsContext) : void
        + Empty method.

#### Class Vertex2D

* + - Operations:
      * Vertex2D(int, int)
        + Constructor.

#### Class Mesh2D

* + - Operations:
      * Mesh2D()
        + Constructor.

# 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]. |