<u>Front End/User Interface:</u>

The Front end is split into several components:

- **Top Bar (Above Map):**
  - Your Name, what 'phase' the game is in, and whose turn it is.
- **Right Navigation Bar:**
  - All Player Names, color coded, the current player is outlined in gold. These can be clicked for more player info.
  - A Match Log of previous moves that have been made.
  - What moves you can currently make, and the buttons necessary to make them.
  - An option to view your cards or skip your turn.
- **Left Chat Bar:**
  - For chatting with people. Filters script tags, censors some swearing. Messages are color coded to the player that sent them.
- **Card Drop Down:**
  - Clicking the show cards button in the right navigations bar will display a dropdown from the top that displays your cards. These must be clicked to turn in.
- **Map:**
  - Using AMMaps library to simulate the map
  - Territories:
    - represented by circle
    - Each lat/long coordinate is in an array:
      - Ex: const CHINA = [36.897568, 103.375284];
    - Each "territory data structure" is a dictionary
      - Ex:
        ```
        let CHINADATA =   {
                "latitude": CHINA[0],
                "longitude":  CHINA[1],
                "type": "circle" ,
                "color": "#CC0000",
                "scale": 0.5,
                "label": "China 5",
                "labelShiftY": 2,
                 "zoomLevel": 5,
                "title": "China",
                "id": "China",
                 "description": "Occupied by Player 1. Soldiers: 5"
        };
        ```
    - This is then put into an array in which is then attached to the variable "image" in the AMCharts.createChart function
  - Borders and adjacency:
    - AMMaps has a line attribute
      - It's an array in which each index contains a dictionary with two arrays

        - The first array is associated with lat start and lat end which the second array is associated with long start and long end
        - Each dictionary is assigned to a variable
        - Example: of one index in the line array

```
let AB-OT = {
        "latitudes": [ ALBERTA[0], ONTARIO[0]],
        "longitudes": [ ALBERTA[1], ONTARIO[1] ]
        “color”: blue
}
```

- **Reflecting Changes**
    - Each map element reflects its changes, territory labels change color and number to display ownership and number of troops. When players are attacking, border connections change as well.
    - Each player is only able to see buttons and UI that are permissible on their turn.
    - All relevant info is recorded in the match log.

Networking:

Package Client:

The Client package manages organization of player sessions, and communication between the back end and the front end. It works by taking in move information from a player session, and passing it to the back end. The back will then a return a game update. This update will return whether or not the move requested by the front end was valid, the results of that move and what needs to be displayed on the board as a result, and what moves are now available for the player who just moved.

- **Matches Class**:
    - Manages and organizes all sessions into game lobbies. Holds onto player sessions, and match classes, the internal representation of a lobby.
    - Connected Method: Add this new session our sessions queue, and give it a unique id. Send update messages to this session informing it of available match lobbies that it needs to display.
    - Closed Method: Remove this session from our sessions queue, and update the lobby menu to no longer list that session as being present. Inform the match this players was in that the game is over.
    - Message Method: Respond to different message from the GUI based on the message type. Can make calls to join_player, start_game, and create_lobby, or ask a Match class for a game update. These updates are radiated out to the players in that match.

    - Join_player Method: Store a player id in the selected lobby. Update other lobbies in case this player changed the lobby they wanted to be in. Asks the GUI to see if the selected lobby is now full, and a match can begin.
    - Start_game Method: Begin a match and do not allow any new sessions to join the lobby.
    - Create_lobby Method: Creates a new lobby with the size and name inputted by the player.

- **Match Class**
    - Manages and individual game of risk. Sends and receives messages to update the GUI and ask for move updates from the back end.
    - Has getters for the match name, match id, current number of members in lobby, maximum lobby size, and player ids. Can add and remove players.
    - Start Method: Causes the match to no longer add or remove players. Sends an initial game update to matches so the game can begin. Beings accepting requests for game updates, based on player input, which is parsed, validated, and enacted in the back end. Once that is done, a set of response messages are sent to matches, which radiates them.

<u>**Game Design:**</u>

Package Game: This class contains interfaces and classes common to all games, such as the Die and GameUpdate classes as well as the player and cardpool interfaces.

- **Class Die**: has a method that simulate a regular 6 sided die roll.

- **Interface Player:** Player has only one method that must be implemented, getPlayerId() which returns a UUID, the unique id of a player used by the network to determine which game a player belongs to.

- **Interface CardPool:** Represents a generic pool of cards. It has three methods that must be implemented: handoutCard(), handinCard() and isEmpty(). handoutCard returns a card to handout, handinCard() takes in a card to turn in and isEmpty() returns a boolean indicating if the cardpool is empty. handinCard() can throw an UnsupportedOperationException by implementing classes that do not allow cards to be added back into the pool.

- **Class GameUpdate:** The GameUpdate class is used to store information to send to players in a game. It can hold information about an action that was executed, the next valid action for a player, if a player lost the game or if a player won the game. It also can store information about handing out cards and whether or not an error occurred when attempting to execute an action. This information is sent via the network to all of the players in the game and is used to update the game view/state in the frontend. The valid action object sent is used by the frontend to restrict a player's from taking any action that is illegal as defined by the object.

Package Risk: The risk package contains the data structures most fundamental to the operation of the risk game. The package is divided between classes related to execution of the game (actual gameplay) and classes controlling the logic, flow of control, and rules of the game.

- **Class RiskBoard**: models the game board. The board knows about the territories on it, which territories are neighbors and holds the actual Territory objects that store all of the necessary information for a Territory.

- **Class RiskPlayer**: represents a RiskPlayer. A RiskPlayer is identified by a unique UUID assigned by the network and knows which territories and continents they control as well as the cards they have.

- **Class Referee**: The referee keeps track of the game state and determines whose turn it is and which type of action is legal at any given moment in the game. It holds onto a global validMove object, an instance of ValidAction, which defines the legal moves at any given time. Referee deals with validating actions, switching players and determining if a player has lost or won a game. For each type of action, it has a validate method, which first determines if the type of action is valid at the time (based on the type of ValidAction validMove is) and then calls the validate method of validMove. If the validate method returns false, the move is invalid. Otherwise, the move is valid. Referee also has methods that deal with calculating the next valid action after an action is executed. These methods are called by the RiskActionProcessor.

- **Class RiskActionProcessor**: A RiskActionProcessor is associated with a Referee object, which it uses to determine if Risk actions should be executed. For each type of action, it has a process action method, which takes in the action. In these methods, the RiskActionProcessor calls on the referee to validate the action. If the referee returns true, the executeAction method is called which actually carries out the action and updates the state of the game. After an action is executed, the process methods ask the referee class to return the next valid action for the current player. If the next valid action is null, the processor than asks the referee to switch the player whose turn it is, as the current player has not actions available. All public methods of RiskActionProcessor return a GameUpdate object, which stores the move just executed, the next valid action for a player, if someone won/lost the game, and a card to hand out to a player. If the Action was not validated by the referee, RiskActionProcessor returns a GameUpdate message with an error and the valid action object defining what a player can do.

- **Class RiskCardPool:** This class represent the CardPool in Risk. It implements the CardPool interface. There are 42 cards in Risk, 30 one-star cards and 12 two-star cards. Cards can only be handed out in risk, but not returned to the pool. Once the pool of cards it empty, the handoutCard method returns -1 to indicate it is empty. The handinCard method throws an UnsupportedOperationException.

- **Class RiskMessageAPI:** This class is used to translate game update objects into json messages to send to players in the frontend. It is also used to translate messages received by the frontend into Action objects to send to the RiskActionProcessor.

Package RiskWorld: This package contains classes and enums relevant to the territories and continents used in Risk.

- **Class Territory**: The territory class represents a territory in Risk. It knows how many troops occupy it and the player who owns the territory (if any). Troops can be added and removed from a Territory and a Territory can change owners if the number of troops is zero and if no one owns the territory. A territory also knows which continent it belongs to.

- **Enum TerritoryEnum**: Each territory is represented by a unique id defined by a TerritoryEnum, which is sent into the constructor of Territory. TerritoryEnum has a public method called getContinent(), which returns the Continent a TerritoryEnum belongs to by calling the getTerrs() method of ContinentEnum.

- **Enum ContinentEnum:** Each Continent is represented by a unique id defined by a ContinentEnum. ContinentEnum has two public methods: getContinentalBonus() and getTerrs(). getContinentalBonus returns the bonus value associated with a Continent and getTerrs returns the set of TerritoryEnum's that belong to a given Continent.

Package RiskAction: The RiskAction package contains the data structures necessary to define each possible type of action during Risk. There are are two types of classes in this package. One implements the Action interface and defines the actual action to execute/executes the action. The other type implements the ValidAction interface, which defines all of the possible moves of a particular action type that can be made.

- **Enum MoveType:** This enum is used to give an id to each type of move: setup, setup reinforce, reinforce, turn in a card, attack, defend, claim territory and move troops.
- **Interface Action**: All classes that execute actions/moves implement this interface. It has four methods that must be implemented: getMoveType(), getMovePlayer(), isActionExecuted() and executeAction(). getMoveType() returns the MoveType the action represents, getMovePlayer() returns the player making the move, isActionExecuted() returns whether or not the action has been executed and executeAction executes the actual move.

- **Class SetupAction**: This class represents selecting a territory to claim at the beginning of the game when each player selects unoccupied territories to place one troop at.

- **Class SetupReinforceAction**: This class represents the reinforcement phase after all territories have been claimed at the beginning of the game. One territory is selected at which one troop is added to the territory.
- **Class CardTurnInAction:** This class represents a card turn in action, in which a player turns in one or multiple cards. Executing this action removes the cards from the player set, and the number of troops given to reinforce a player's territories is increased by the value of the cards handed in.

- **Class ReinforceAction**: This class represents the reinforcement phase each player carries out at the beginning of their turn. Executing this action reinforces territories with troops as defined by the Map received in the constructor.

- **Class AttackAction:** This class represents an attack. It takes in the territory a player is attacking from, the territory being attacked and the number of dice the attacker is going to roll. Executing this action rolls the dice.

- **Class DefendAction:** This class represents a defend action. It rolls the number of dice the defender chose to roll and compares the roll results of the attacker with the defender. Based on the comparison, troops are removed from the attacker's territory or the defender's territory. If the defender lost the territory, it sets a boolean indicating the territory was lost.

- **Class ClaimTerritoryAction:** This class deals with an attacker claiming a territory defeated in battle. It removes the specified number of troops from the attacking territory and adds them to the territory being claimed, changing the territory's owner.

- **Class MoveTroopsAction:** This class deals with transferring a specified number of troops from one player's territory to another territory controlled by the player.

- **Interface ValidAction**: All classes that define the possible moves of an action type implement this interface. It has three methods that must be implemented: actionAvailable(), getMoveType() and getMovePlayer(). actionAvailable() returns whether or not the action itself has any possible moves (determined by the constructor of classes that implement this interface). getMoveType() returns the type of action this class represents and getMovePlayer returns which player can make this type of action.

- **Class ValidSetupAction:** This class deals with determining which territories are available to be claimed (unoccupied) at the beginning of the game selectable and which player can claim make this move. It has a validate method which takes in a SetupAction and determines whether or not the SetupAction represents a valid action as defined by this class. The action is unavailable if all of the territories have been claimed.

- **Class ValidSetupReinforceAction:** This class deals with determining which territories a player can reinforce after the SetupAction phase, which is just the set of territories owned by the player. It has a validate method which takes in a SetupReinforceAction and determines whether or not the SetupReinforceAction represents a valid action as defined by this class. The action is unavailable if the player has placed all of their initial reinforcements.

- **Class ValidCardTurnInAction**: This class deals with determining if a player can turn in any cards and which cards can be turned in. It has a validate method that determines if a CardAction is valid. The action is unavailable if a player has no cards to turn in.

- **Class ValidReinforceAction**: This class deals with determining the number of troops a player can reinforce with and where the troops can be placed on the board. It has a validate method that determines if a given ReinforceAction is valid. ReinforceAction is always available for players that have territories and have not lost the game.

- **Class ValidAttackAction**: This class deals with determining which territories a player can attack from, which territories a player can attack and from which territories, and the number of dice a player can roll when attacking from a specific territory. It has a validate method that determines if a given AttackAction is valid. The action is unavailable if the player has no territories that can attack based on their location and the total number of troops occupying the territory.

- **Class ValidDieDefendAction:** This class deals with determining the number of dice a defending player can roll based on the number of troops on the defending territory. It has a validate method that determines if a given DefendAction is valid. The action is always available for players being attacked.

- **Class ValidClaimTerritoryAction**: This class deals with determing how an attacker can claim a territory they defeated in battle. It knows which territory it is claiming, the attacking territory and the total number of troops that can be moved to the territory to claim from the attacking one. It has a validate method that determines if a given ClaimTerritoryAction is valid. This action is always available for players that have won a territory.

- **Class ValidMoveTroopsAction**: This class determines from which territories to where a player can transfer troops and how many troops can be transferred from a given territory. It has a validate method that can determine if a given MoveTroopsAction is valid. The action is not available if a player has no territories from which troops can be transferred.