Introduction:

Our system uses a standard MVC setup. We have a MainView class that contains our GUI design for the users of our program to interact with. Our main program is class called RiskView that instantiates a RiskModel and MainView, both of which are utilized to instantiate a RiskController. RiskController receives information about the user's interaction with the view and passes that information to the RiskModel which performs tasks based on the user input.

RiskController is setup to execute commands based on which of the numerous buttons contained in MainView are clicked. Once a button is clicked, RiskController checks with the model to determine what state that the system is in and then properly calls methods of the model to execute the action the user is attempting to accomplish. RiskModel is composed of a number of methods that directly communicate with our other classes to handle playing a game of Risk.

User Stories: As a player I want to be able to…

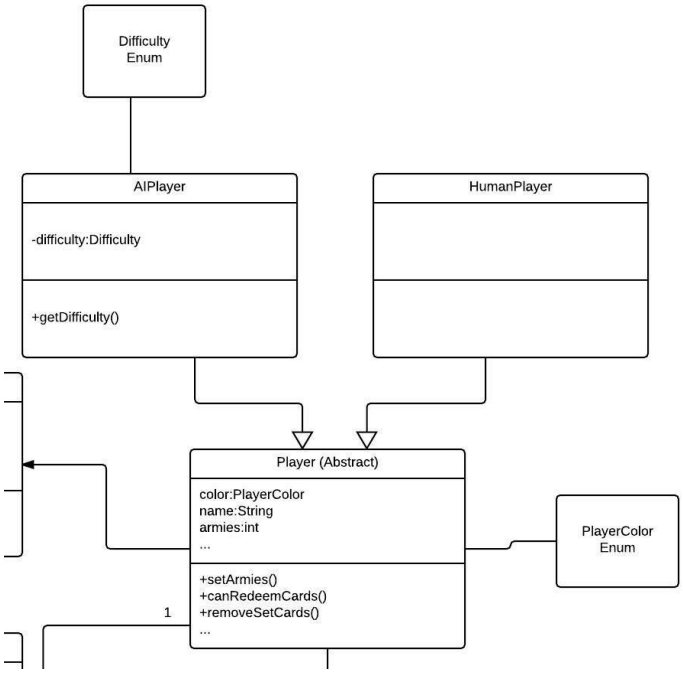
● See a Title Screen on startup ● Start a new game ● Select the number of players for the game ● Display options regarding the name of players ● Select the Number of Human and AI playres ● Display the game board ● See the options I have during a turn	● Receive armies each turn ● Fortify my territories with with armies I receive ● Roll dice for combat ● Attack enemy territories ● Defend my territories from attack ● Conquer territories in combat ● Transfer armies between territories ● Acquire Risk cards from attacking territories

● See the number of armies on the board ● Know what the turn order will be for the game ● See the turn order and what player is currently playing	● Trade my Risk cards for additional armies ● Play a full game against AI opponents ● See a game over screen when a player has won the game

Our system enables us to accomplish these user stories through the use of our MCV design. The MainView class provides the user with visuals of everything that is needed to be seen such as the game board, the total players playing, the turn order, option buttons, territory buttons, the dice panel, and army labels. This class therefore fulfills the user stories dealing with visuals. Our RiskModel class contains methods such as handleDiceRoll, handleLosses, acquireCard, conquerTerritory, and nextTurn that emulute aspects of a Risk board game. This allows for the satisfaction of the user stories such as “Conqer territories in combat” that deal with the mechanics of the game.

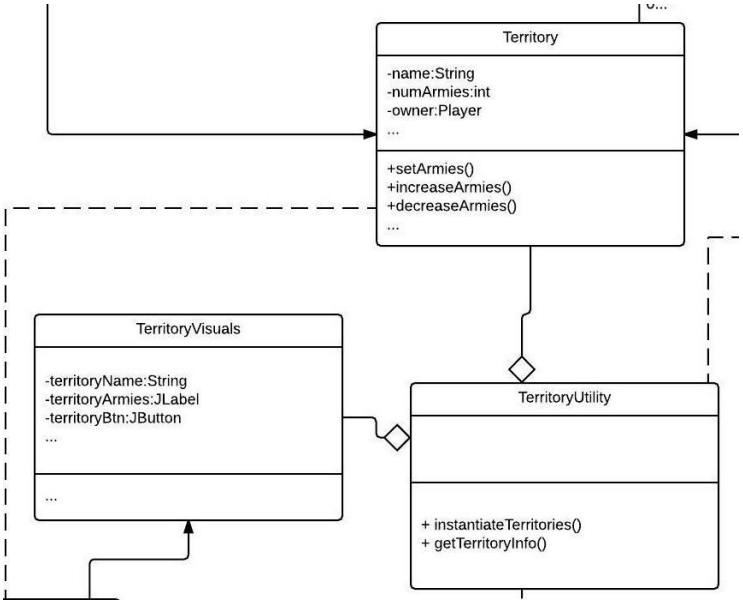
OOD:

Player	
• Abstract class • Knows color • Knows name • Knows number armies to add • Knows cards	• Territory • Card • RiskModel • AIPlayer • HumanPlayer



One of our most important classes is the Player class. Player is an abstract class that models a player participating in the game of Risk. Player possesses attributes of its color, name, armies, owned territories, and owned Risk cards. In our design, we noticed that we would only ever instantiate either an AIPlayer or a HumanPlayer, both of which are subclasses of the Player class. As a result of this, we decided that to refactor Player into an abstract class that contains methods that can be used for both AIPlayer and HumanPlayer. It contains methods that allow a Player to receive armies, Risk cards, and claim territories. Through communication with the model, an instance of a Player is able to emulate the actions a player would take when playing the board game Risk.

TerritoryVisuals	
• Indirectly connects territory objects to view • Knows territory button • Knows territory name • Knows territory icon/armies	• RiskController

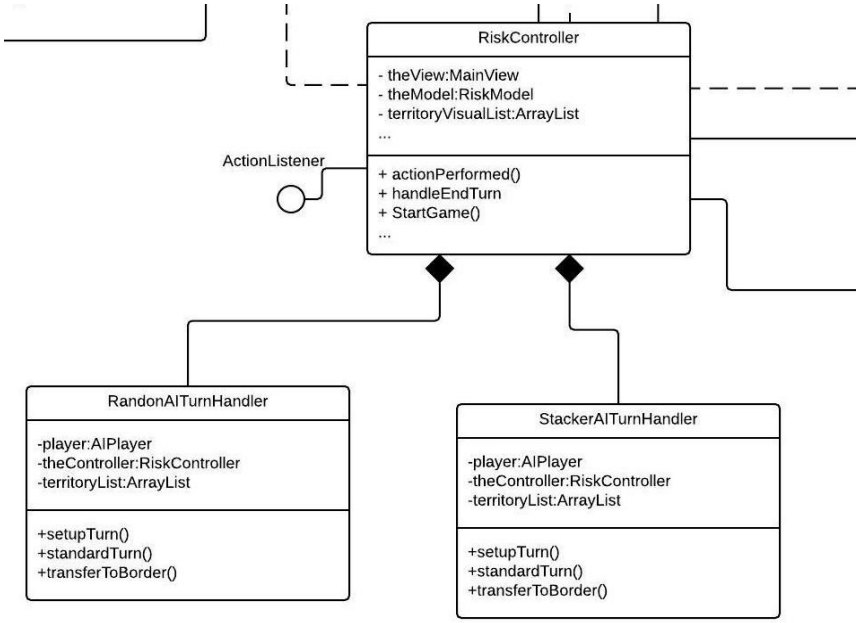


Another important but strange class is the class we called TerritoryVisuals. This class is a grouping of the JLabels and JButton that relate to a single territory which is kept track of through the String name of that territory. For example, a TerritoryVisuals object whose name attribute is “Alaska” would contain a JLabelfor the cannon image on Alaska, the JLabel text that represents the number of armies on the territory, and the button that lays on top of the territory. Having a class that groups these JLabels and JButton together allowed us to update the changes made to a territory much easier than it would have been to do so without it. This class is a major component to our system in that our process of acquiring the button pressed and then performing the actions and updates on the territory relies heavily on this particular way in which we chose to group these elements. In hindsight, it would have been much better to have a Territory object itself be an attribute of the TerritoryVisuals class as opposed to the name of the

territory since this would have saved us the trouble of having to find the proper territory through comparing names. However, by the time we realized this, this particular format of the TerritoryVisuals class had become so important to our system that changing it was a tedious and unproductive option.

RandomAITurnHandler	
• Plays turn for easy (random) AI	• RiskController

StackerAITurnHandler	
• Plays turns for hard (non-random) AI	• RiskController



Two classes important to our goals for the game were RandomAITurnHandler and StackerAITurnHandler. We first implemented RandomAITurnHandler in order to determine how we could implement AI control of turns from within the game itself. This AI did little other than claim random territories in during the setup phase, but was still important for testing purposes, both in decreasing the setup time when we wanted to test features like attacking and risk cards,

but also in understanding how we could implement a smarter AI that actually made decisions based on the game state. Eventually we phased out the RandomAIHandler and made the full switch over to StackerAIHandler, which plays the game with basic goals such as moving armies to territories that border enemies, and attempting to claim entire continents to get bonus armies each turn. This fulfills the user story of being able to play against an AI opponent while having it be more than just a simple AI actor.