Domination (Risk Board Game)

Lawrence Celis (#6359175)

Jorge Cook (#5646774)

Eric Karemera (#5734223)

Marouen Lamiri (#6226744)

Emmanuel Tsapekis (#5849004)

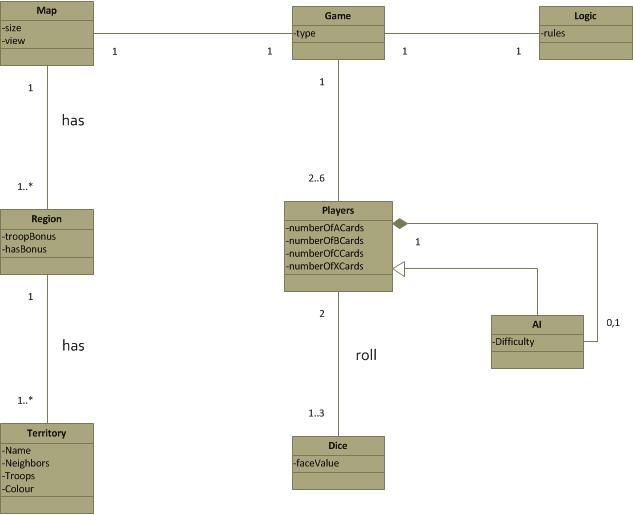
Due: Wed, Oct 30 at 1pm

# Summary of Project

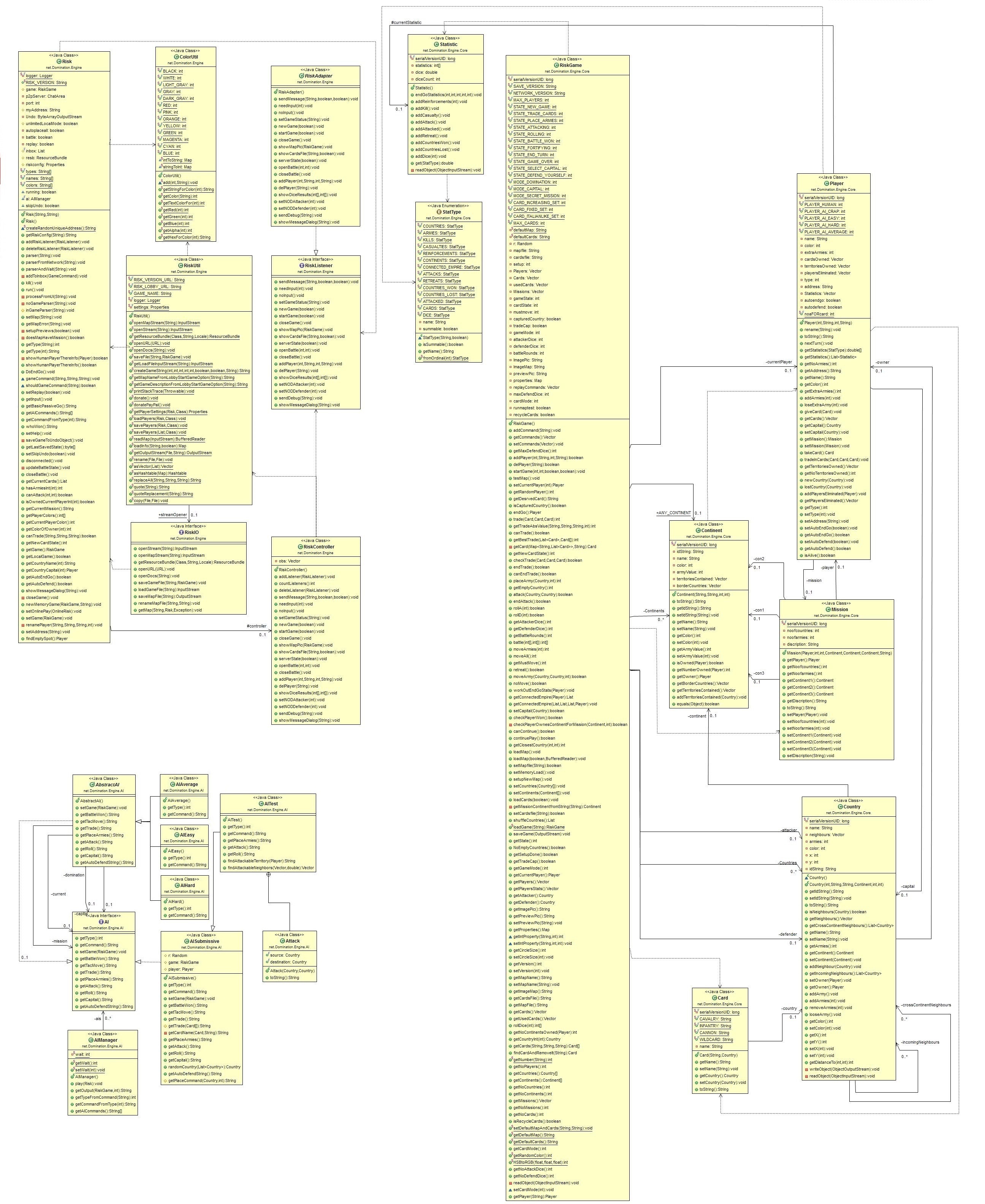
Domination is an open source turn-based strategy game that is written in Java. It’s based on the very popular Risk Board game where the players try to conquer the world using their armies, which increase in number over time. The game can be played in single player mode with AIs or with other humans in the multiplayer mode. Currently, the game is available for the Windows and Android platforms; however, our team would be working exclusively on the Windows version of the game.

# Part 1

# Domain Diagram from Milestone 2



# Class Diagram of the Actual system



# Description of Class Diagram:

Regarding the UML class diagram, we can observe that there are three major groups of classes that are all connected to each other via the GUI of the game. The first one has the class “Risk” and “RiskUtil” as the two main classes. “ColorUtil” and “RiskListener” are the two dependencies of the “Risk” class; “Risk” also has a directed association with the class “RiskController” with multiplicity 0..1, which has two dependencies itself: “RiskListener” and “RiskUtil”. This last class has two dependencies: the “ColorUtil” class, and the “Risk” class, and one directed association with “RIskIO” with multiplicity 0..1, which depends on the “Risk” class.

The second group has the class “RiskGame” as its core. This class has three dependencies: “Mission”, “Statistic”, and “Card” and also the three classes associated to it: “Player”, “Continent”, and “Country”. “RiskGame” takes care of dealing with the logic for each game turn. It has an association with ‘Player’ and ‘Continent’ classes of multiplicity 0..1, and 0..\* respectively. This makes sense because the player essentially plays the game entirely by means of selecting countries, represented by “Country” class, which is associated to “Continents” with a multiplicity of 0..1. In summary, in addition to associations mentioned above, the “RiskGame” class is also connected with “Statistic”, “Mission”, and “Card” classes.

The third group of classes deal with the A.I. of the game. The Abstract class ‘AbstractAI’ is extended by the concrete classes ‘AIAverage’, ‘AIEasy’, and ‘AIHard’. The ‘AbstractAI’ class is associated (with multiplicity 0..1) and uses the “AI” class as an interface. “AItest” extends the class “AISubmissive” which itself also implements the “AI” interface. Finally “AIManager” is also associated with the “AI” class with a multiplicity of 0..\*.

# Comparison with Domain Diagram in Milestone 2

When comparing the actual diagram with the conceptual classes we did for milestone 2, the main difference is the way we thought of relating the “Player” class with the “AI” class. As reflected in the Domain Diagram, we pictured the “Player” class as being the one dealing with all types of players and the “AI” class extending “Player” with the purpose of using its methods and refining them for the use of an AI player. On the other hand, the class diagram shows how the actual implementation of the game has a group of A.I. classes separated from the rest of the game logic, and a “Player” class is more related to gameplay logic, associated with “Statistic” and “Mission”, and with relationships with other classes like “Continent”, “Country”, and “RiskGame”. In addition, the actual class diagram shows a single class “Cards” for the cards held by a player in the game; on the other hand, in our Domain Diagram, we had “Player” dialing directly with the cards in the game.

The rest of the classes do somehow follow an idea similar to the one we reflected in the domain diagram. With “Continent” and “Country” related to each other, and both associated with the logic of the game, compared to our “Territory” and “Region” associated to “Map”, which itself is connected to “Logic”.

In terms of the architecture of the program, the actual diagram shows the classes subdivided into two defined groups: logic, and A.I. However, some of the classes, such as “Statistic”, might be better placed somewhere else other than with the logic of the game. Furthermore, a subdivision of the classes dealing with the gameplay information would have been more suitable. On the other hand, in the actual architecture, gameplay information and logic classes are bound together into a single group of classes, and the A.I group is separated from the rest.

# Description of the tool used

The tool we used to obtain the domain diagram from the actual code was “ObjectAid UML Explorer”, a plugin for “Eclipse” which showed to be an excellent choice given the fact that it provided most of the classes’ details without much effort. One downside, however, is the fact that the relationships shown by the tool are to some extent messy, and must be arranged manually, which can take a long time in the case of a big class diagram such as the one shown.

# Code of ‘Continent’ and ‘Country’

**public class Continent implements Serializable {**

**private String name;**

**private int color, armyValue;**

**private Vector territoriesContained = new Vector();**

**private transient Vector borderCountries;**

**public Continent(String id, String n, int noa, int c)**

**public boolean isOwned(Player p)**

**public int getNumberOwned(Player p)**

**public Vector getBorderCountries()**

**public Vector getTerritoriesContained()**

**public void addTerritoriesContained(Country t)**

**public class Country implements Serializable {**

**private Vector neighbours;**

**private Player owner;**

**private int armies, color;**

**private Continent continent;**

**private transient List<Country> crossContinentNeighbours;**

**private transient List<Country> incomingNeighbours = new ArrayList<Country>(2);**

**public Country (int p, String id, String n, Continent c, int a, int b)**

**public boolean isNeighbours(Country t)**

**public List<Country> getCrossContinentNeighbours()**

**public void addNeighbour(Country t)**

**public List<Country> getIncomingNeighbours()**

**public void addArmy()**

**public void addArmies(int n)**

**public void removeArmies(int lessArmies)**

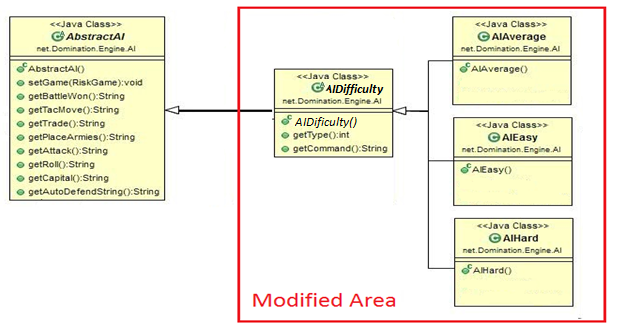
**public void looseArmy()**

# Code Smell and Possible Refactoring

**Refactoring #1:**

We noticed that the 3 difficulty classes (AIAverage, AIEasy, AIHard) had exactly the same methods, the methods being **getType()** and **getCommand()**. To improve readability of the code, and to decrease repetition, we propose the following refactoring: “**Extract Superclass**”

* Step 1: Create an **AIDifficulty** class which will be the base class for the 3 difficulty classes, it will also be a subclass to the AbstractAI class
* Step 2: Make this class into an abstract class
* Step 3: Create the following methods in AIDifficulty class: getType() & getCommand()
* Step 4: Delete the following methods from the 3 difficulty classes: getType() & getCommand()
* Step 5: Compile and test



Code for Refactoring 1:

**public** **class** **AIHard** **extends** AbstractAI

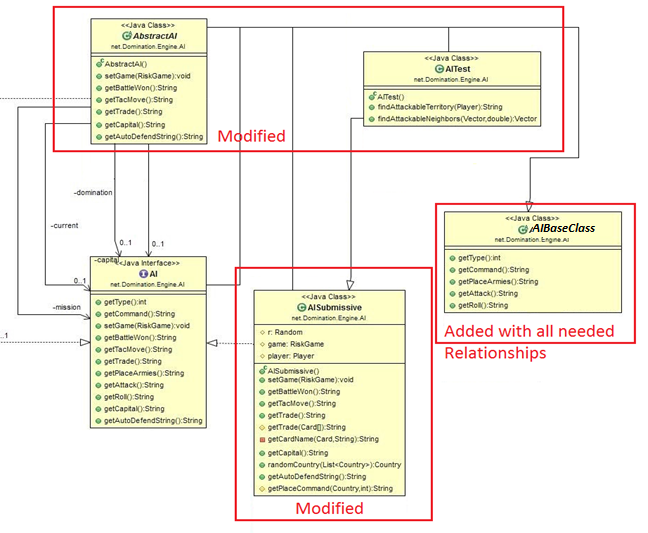
**public** **int** **getType()**

**public** String **getCommand()**

The above is the source code for the ‘AIHard’ class. The ‘AIAverage’ and ‘AIEasy’ classes are very similar in that they also have a ‘getType()’ method that returns a hardcoded ‘int’ and a ‘getCommand()’ that returns a hardcoded ‘string’. The new superclass that we have proposed, ‘AIDifficulty’, will contain these two methods, but because each class contains a different hardcoded return value for both methods, field attributes must be added to allow the methods to return the attributes (rather than the hardcoded values) in the superclass that will be overridden in the various subclasses.

**Refactoring #2:**

We noticed that several methods in the AI section of the code had the same names in different classes but some had different implementation. To improve the code, and to make the classes smaller in size, we propose the following refactoring:

* Step 1: Create an abstract class called **AIBaseClass** which is the base class of all the other class
* Step 2: Retrieve all the methods to be inserted in **AIBaseClass** from the other classes, while modifying the code for different cases of usage of a method (since some methods are used differently in each classes)
* Step 3: Delete all the methods in the other classes that have been put in **AIBaseClass**
* Step 4: Compile and test

**Refactoring #3:**

We noticed that there is a huge class called “**RiskGame**”, which contains several attributes and methods related to the core mechanics of the game that could be extended and separated into several classes in order to reduce its size. A huge class with several attributes and methods could be detrimental to the maintenance of the program since huge classes tend to be less readable and harder to maintain. Additionally, the Weighted Methods per Class (WMC) metric will be reduced as a result of our proposition. We propose the “RiskGame” class to be extended by 3 new classes called “**Fortify**”, “**Attack**”, and “**TroopPlacement**”, in which all features that are seldom used in the original class are placed into the newly created classes. The refactoring method is called “**Extract Subclass**” and will contain the following steps:

* Step 1: Create a new class called “**Fortify**” that extends the class “RiskGame”
* Step 2: Use the refactoring technique “**Push Down Method**” to the methods moveArmy() and noMove(), which are the methods involved for the fortification state of the game, from the “RiskGame” class to the “Fortify” class
* Step 3: Use the “**Push Down Field**” refactoring technique to all the necessary attributes depending to those methods to the new “Fortify” class from the “RiskGame” class
* Step 4: Verify that all traces of the moved methods and attributes within the “RiskGame” class are gone in order to finalize the extraction
* Step 5: Create a new class called “**Attack**” that extends the class “RiskGame”
* Step 6: Use Pull Down Method to the methods attack() and endAttack() from the “RiskGame” class to the “Attack” class
* Step 7: From the “RiskGame” class, use Push Down Field to all the necessary attributes that depends on the new methods within the “Attack” class
* Step 8: Repeat Step 4
* Step 9: Create a new class called “**TroopPlacement**” that extends the class “RiskGame”
* Step 10: Use Pull Down Method to the methods placeArmy() and getEmptyCountry() from the “RiskGame” class to the “TroopPlacement” class
* Step 11: From the “RiskGame” class, use Push Down Field to all the necessary attributes that depends on the new methods within the “TroopPlacement” class
* Step 12: Repeat Step 4
* Step 13: Compile and test