Bilkent University

Department of Computer Engineering

CS 319

Object Oriented Software Engineering

Game of "Risk" Design Report

Group 1-E

Mehmet Ege Acıcan-21602186

Ali Ata Kılıçlı-21301530

Ekin Üstündağ-21602770

Mehmet Akif Kılıç-21502487

Seman Arda Yuvarlak-21501402

Contents

1. Introduction

1.1 Purpose of the System

Game of "Risk" is a strategic conquest and conquer game. It' s main purpose is to make the players be in a competitive and entertaining strategic game with each other on the computer environment and therefore it is designed this way. The game of "Risk" is based on the classic board game, "Risk" and its gameplay is almost identical to the classical board game version however we added 2 different gameplay mods to the game in order to fix the time problem that the original game has. The Game of "Risk" is a game with 3 gameplay mods which are: Classical, Time Mod and Golden Territories. Classical Mod is the classic implementation of the board game while time mod adds a time limit to the classic game, declaring the player who has the most territories at the end of the clock as winner. Golden Territory mod on the other hand declares the player who occupies the adjusted "Golden Territory" as the winner. We increased the entertainment aspect by eliminating the time factor that specifically made the game "boring". Our design and our architecture will be based on subsystems that satisfies the functional requirements in our analysis report.

1.2 Design Goals

We will firstly establish our criterias and trade-offs of our design to simplify the implementation process. Therefore we are going to design the game before implementing. The main criterias and trade offs are:

1.2.1 Maintenance Criteria

Maintainability: The game design is based on object oriented programming and thus game will be consisted of systems and subsystems, which are all going to be connected with each other in hierarchical structure.

Reliability: The game should contain as less bugs and errors as possible so that players are able to play the game without any disruption. In order to achieve this goal, we are going to look at the interactions between

the User and the Interface and decrease the error potential as much as possible. Therefore, the User Interface will handle exceptions without alerting the User.

Modifiability: Our system will follow the same principles as any other software systems follow: Open to changes and updates. Therefore, Our game is going to allow modification without showing any problems. Also, classes are going to be, even though they are connected, as independent as possible so that whenever a change occurs in one, the others will be affected by it as minimum as possible.

Reusability: Our system is going to reuse specific components in order to prevent including another subsystem with the same features, creating repetition problem.

1.2.2 User Criteria

User Friendliness: Game of "Risk" is going to include user friendly interface in order to allow an easy understanding of the game and thus allowing a fun gameplay. The user will mostly use mouse in order to interact with the game. The control mechanisms such as choosing a territory to attack and distributing troops will be handled by simple clicking to the territory and dragging the mouse.

Performance: For a better gameplay, the game performance is an important aspect. Therefore, our game will try to be as fast as possible with main purpose to not make the player wait for more than 0.1 seconds in any of the operations that is in the game.

1.2.3 Trade offs

In order to allow these criterias there are specific trade-offs that should be considered:

Functionality vs User Friendliness: In order to make the game user friendly, the game will be designed in a way to make it as simple as possible. This decreases the complexity of the game and thus decreases the possibility of specific functionalities that can be added.

Efficiency vs Portability: The Game of "Risk" is going to be designed only for the Laptops and Desktops in order to provide better efficiency. Therefore, the portability factor is limited.

Cost vs Simplicity: Since the game is going to be implemented as simple as possible, The cost of time by the developers decreases. Therefore, because of not trying to add any complexity to the game, the time cost is going to be minimized.

1.3 Definitions

MVC(Model View Controller): The separation of model, view and controller. The main design pattern of the Game of "Risk". Model, View and Controllers are explicity defined.

GUI(Graphical User Interface): Interface that allows the players to interact with the software. Game of "Risk" uses GUI.

JavaFX: A software platform for delivering desktop applications. JavaFX allows us to make fxml files to manage view.

2. High Level Software Architecture

2.1 Subsystem Decomposition

This section divides the main system into specific subsystems and analyzes them separately. We plan to do this in order to possibly increase the freedom, modifiability and flexibility of our system in the future. Our design is composed of GUI layer, Gameplay Layer and Data Layer

GUI layer handles all of the GUI tasks such as displaying the map and getting mouse inputs from the user. GUI layer is composed of Game GUI and Menu GUI Managers. While Menu manager handles the menus that is offered to the user, Game Manager handles the interactions between the User Interface and the Gameplay. In other words , Map manager is responsible for adjusting the map according to the changes of game whenever it is necessary, while Menu Manager is responsible for Main menu, pre-game-settings and settings menu.

Gameplay layer handles the interaction between the GUI and the Data Manager, thus allowing the gameplay. The games layers

connection between the layers is handled by this layer. Input manager handles the receiving the inputs of the users. Sound Manager adjusts the sound and movement manager handles the changes according to the inputs of the users

Data Manager layer is responsible for handling the data kept in the system. These data include Player data such as player territories and current player troops. These are handled specifically in the player data manager. Territory data manager specifically holds information about the territories. Game data manager handles the saving and loading games.

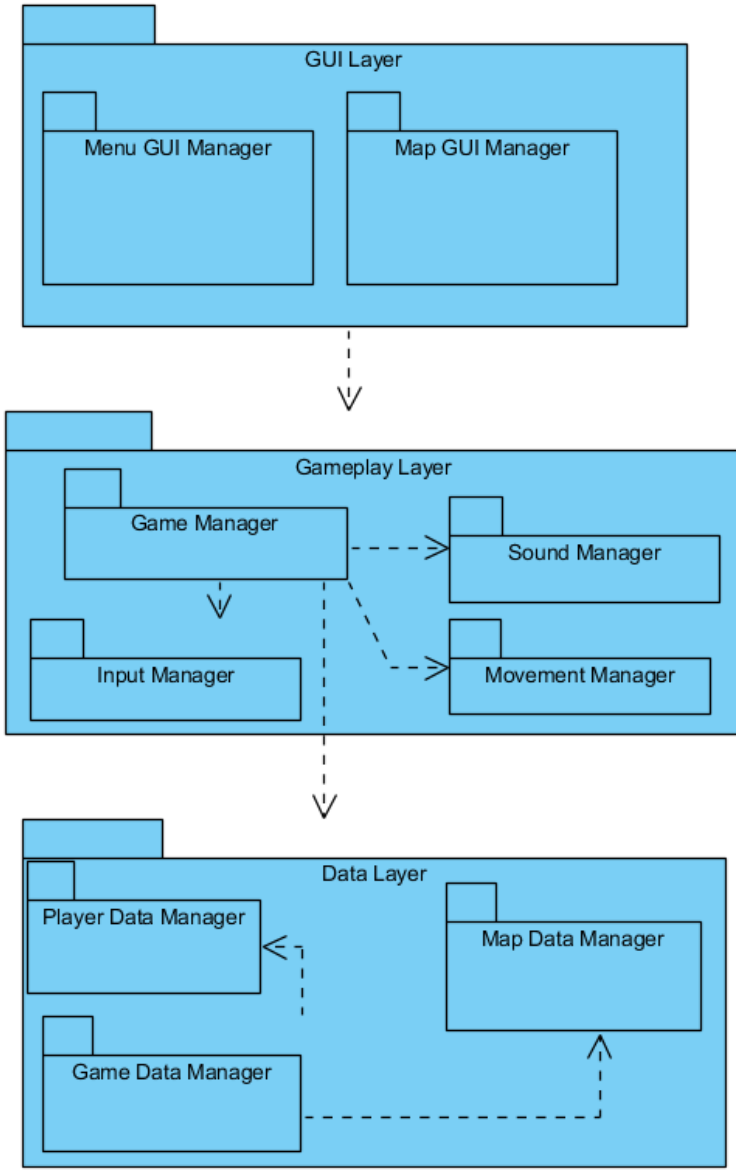


Figure 1: Package Diagram for showing layers

2.2 Hardware/Software Mapping

Game of "Risk" will be implemented in Java programming language and use Javafx libraries to handle the GUI. Also, the

game will use JRM so it will be playable on multiple operating systems. A mouse and a keyboard is sufficient for interacting the game since the mouse is going to handle all of the GUI inputs for Game and menus while keyboard is going to handle the name giving to the game and storing the name to the data. Data manager subsystem is working in order to handle the saving and loading the games.

2.3 Data Management

In our game, we will use Java libraries serialization and deserialization libraries functionality to save and load the game. The saved games will include the current state of the Map(who holds which territory) and remaining time(if the loaded game is in Time Mod).

2.4 Access Control and Security

Since our game does not hold any unique and important data from the players, We do not prioritize Access Control and Security.

2.5 Boundary Conditions

Shutdown: Players are able to exit the game in the main menu or they can exit the game from the pause menu which appears if they pause the game while they are playing. The game will automatically save before exiting.

Error Behaviour and Exception Handling: If there occurs a run time error such as the game not being able to load the data of the previously saved games, We will display a run time error message. The game will try to handle exceptions without alerting the user.

Starting the Game: As the user starts the game, main menu will be shown without any requirements. After clicking the play button, after the player chooses a new game, adjusts gameplay mod and the pre-game-settings, the system will create a new slot for the players new game, save it and then initialize the game. If the player chooses load game option, if no problem

occurs, the system will initialize the loaded game that the player selects from the load games. If one of the loaded games is deleted from the slots, then the slot will be free and a new game is going to be free to start. Otherwise, the slots will show which saved game that they contain along with the last time that they had been played. If one of the loaded files is corrupted, the game will show an error message.

3. Subsystem Services

3.1 GUI Layer

Map GUI Manager: The map and the changes in the map according to the game is going to be handled by fxml files and libraries which are connected to the classes that connects game datas and user inputs.

Menu GUI Manager: Menu part of the GUI manager consists of all the menus that the user encounters. These include: The main menu, new-game-settings menu, pre-game-settings menu, play-menu, general settings menu. Main menu is the opening menu that allows users to choose to play the game, settings and directly exit the game. Play menu allows the user to load a game or start a new game. New-game-settings menu allows the user to choose the gameplay mod, pre-game-settings menu allows the user to adjust the pre-game conditions, general settings menu allows the user to adjust the sound and brightness of the game.

Menu

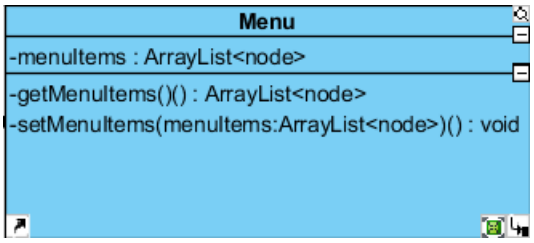


Figure 2: Menu Class

Attributes:
menuItems: An arbitrary list that contains menu Items
Methods:
getMenuItems(),setMenuItems(menuItems:ArrayList<node>)
These are the getters and the setters of the menu

Main Menu Controller

Figure 3:MainMenuController class

Constructor:

Public mainMenuController(): This constructor allows to construct the objects in terms of configuration of the fxml files. For this case, it adjusts or changes labels.

Play-Menu-Controller

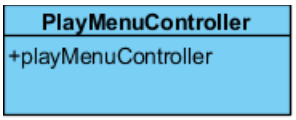


Figure 4:PauseMenuController class

Constructor:

Public playMenuController(): This constructor allows to construct the objects in terms of configuration of the fxml files. For this case, it adjusts or changes labels.

Pause-Menu-Controller

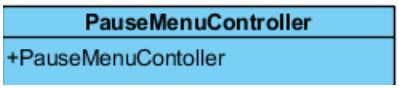


Figure 5: Pause Menu Controller

Constructor:

Public PauseMenuController(): This constructor allows to construct the objects in terms of configuration of the fxml files. For this case, it adjusts or changes labels.

New-Game-Settings Menu Controller

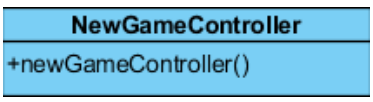


Figure 6:NewGameController class

Constructor:

Public newGameController(): This constructor allows to construct the objects in terms of configuration of the fxml files. For this case, it adjusts or changes labels.

Pre-Game-Settings Menu Controller

Figure 7:PreGameSettingsController class

Constructor:

Public preGameController(): This constructor allows to construct the objects in terms of configuration of the fxml files. For this case, it adjusts or changes labels.

Settings

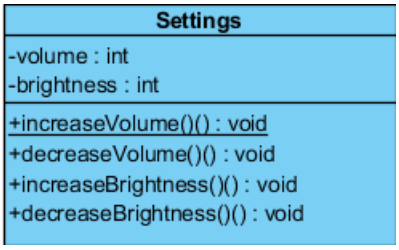


Figure 8:Settings Class

Attributes:

volume: This attribute holds the current volume degree of the game

brightness: This attribute holds the current brightness degree of the game.

Methods:

increaseVolume(): this method allows to increase the volume by 1 unit

decreaseVolume(): this method allows to decrease the volume by 1 unit.

increaseBrightness(): this method allows to increase the brightness by 1 unit.

decreaseBrightness(): this method allows to decrease the brightness by 1 unit

Map

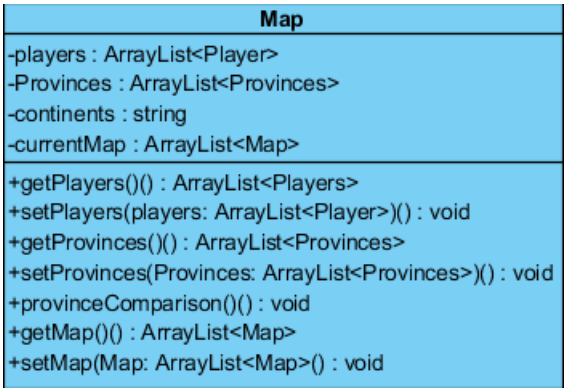


Figure 9: Map class

Attributes:

players: This attribute stores the players of the game in an arraylist.

Provinces: This attribute stores the provinces of the game in an arraylist.

Continents: This attribute defines the continents in the game

currrentMap: : This attribute stores possible playable maps of the game in an arraylist.

Methods:

getPlayers(): this function gets the players of the game.

setPlayers(players: Arraylist<Player>) : this function sets the players of the game.

getProvinces(): this function gets the provinces of the game.

setProvinces(Provinces:Arraylist<Provinces>): this function sets the provinces of the game.

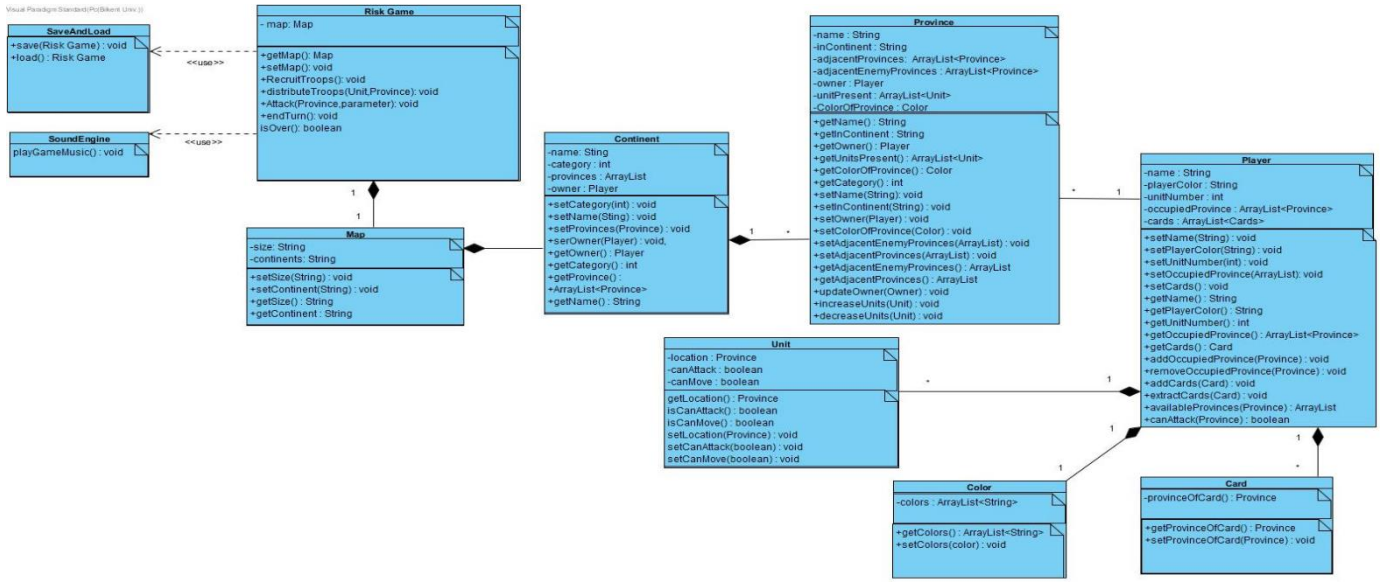
provinceComparison(): This function gets how many provinces each player holds in their hands and compares them.

getMap(): this function gets the current map of the game.

setMap(Map:ArrayList <Map>) : This function sets the map that is going to be played on.

Gameplay Layer

This package has all the GUI-independent game classes storing game play data and providing game play operations. In addition, associations among objects in the package are presented.



SoundEngine

This class provides the static method to perform background music.

SaveAndLoad

save(RiskGame): Method that saves the current game.

load(): Method that loads a game to contuinue playing an old game

RiskGame

Attributes

map: The map of the game that contains all continents and provinces.

Methods

getMap(): Returns the map object of the game.

setMap(): Changes the map object of the game.

RecruitTroops(): Creates Troops objects for players.

distributeTroops(Unit,Province): Distribute Troops objects to players.

Attack(Province,parameter): Performs an attack to given province with given parameter modifier.

endTurn(): Ends the turn of the current player .

isOver(): Checks if the game is finished.

Map

Attributes

size: the size of the map contained in a string.

continents: Continents in the map.

Methods

setSize(): sets the size of the map

getSize(): returns the size of the map

setContinents(): sets the continents in the map

getContinents(): returns the continets in the map

Continent

Attributes

name: name of the continent

category: category of the continent in terms of int

provinces: an ArrayList of provinces that are in the continent

owner: A Player that is full owner of the continent with all of the provinces in it

Methods

setName(String): Sets the name of the Continent

SetCategory(int): Sets the category of the Continent

SetProvinces(ArrayList<Province>): Sets all of the provinces of the Continent

SetOwner(Player): Sets the owner of the continent, when there is

getName(): Returns the name of the Continent

getCategory(): Returns the catefory of the Continent

getProvinces(): Returns the Provinces ArrayList of the Continent

getOwner(): Returns the owner of the Continent, if there is

Province

Attributes

Name: name of the Province

Category: category of the province in terms of int

inContinent: string that holds the continent that the Province in

AdjacentProvinces: An ArrayList of Provinces that this Province is adjacent

AdjacentEnemyProvinces: An ArrayList of enemy Provinces that this Province is adjacent

Owner: the Player that holds this Province

UnitPresent: An ArrayList of Units that are on this Province

ColorOfProvince: the color of the Province in the game

Methods

GetName(): Returns a string that is the name of the province

GetInContinent(): Returns a string that is the Continent the Province in

GetOwner(): Returns a Player that owns the Province

GetUnitsPresent: Returns An ArrayList of Units

GetColorOfProvince(): Returns the color of the province

GetCategory(): Returns the catefory of the Province

SetName(): Sets the name of the Province

SetInContinent(): Sets the Continent that the Province in

SetOwner(): Sets the Owner of the Province

SetUnitsPresent(): Sets the Units on the Province

SetColorOfProvince(): Sets the color of the province

SetCategory(): Sets the category of the Province

UpdateOwner(Player): Updates the curren owner of the province if it is changed

IncreaseUnits(Unit): Increases the number of Units on the Province

DecreaseUnit(Unit): Decreases the number of Units on the Province

Player

Attributes

Name: Name of the Player

PlayerColor: Color of the Player

UnitNumber: number of Units of the Player

OccupiedProvinces: An ArrayList of the Provinces that Player has

Cards: An ArrayList of Cards of the Player

Methods

SetName(): Sets the Name of the Player

SetPlayerColor(): Sets the Color of the Player

SetUnitNumber(): Sets the number of Units of the Player

SetOccupiedProvinces(): Sets the OccupiedProvinces of the Player

SetCards(): Sets the Cards ArrayList of the Player

GetName(): Returns the Name of the Player

GetPlayerColor(): Returns the color of the Player

GetUnitNumber(): Returns the unitNumber the Player

GetOccupiedProvinces(): Returns occupiedProvinces of the Player

GetCards(): Returns the Cards of the Player

AddOccupiedProvince(): adds a province to occupiedProvinces ArrayList

RemoveOccupiedProvince(): removes a province from occupiedProvinces ArrayList

AddCards(): adds a card to cards ArrayList

extractCards(): removes a card from cards ArrayList

AvailableProvinces(): Returns an ArrayList of available provinces

CanAttack(): Returns true if the Player is able to attack

Unit

Attributes

Location: the Province unit is on.

CanAttack: a boolean which represents whether the unit can attack or not.

CanMove: a boolean which represents whether the unit can move or not.

Methods

GetLocation(): Returns the the Province unit is on.

IsCanAttack(): Returns a boolean which represents whether the unit can attack or not.

IsCanMove(): Returns a boolean which represents whether the unit can move or not.

SetLocation(): sets the location of the unit.

SetCanAttack(): sets the boolean canAttack.

SetCanMove(): sets the boolean canMove.

Color

Attributes

colors: an ArrayList contains colors as strings

Methods

GetColors(): Returns the colors ArrayList

SetColors(): Sets the colors ArrayList

Card

Attribute

ProvinceOfCard: the province the card is attached

Methods

GetProvinceOfCard: Returns the province the card is attached

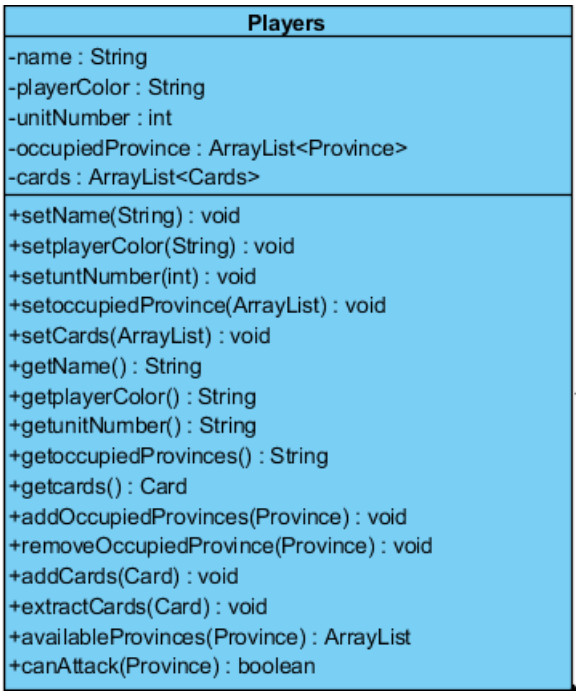
SetProvinceOfCard: Sets the province the card is attached

Data Layer

3.3 DATA LAYER

3.3.1 Player Data Manager

3.3.1.1 Player



Attributes:

private String name: name of the player

private String playerColor : color of the player

private int unitNumber: a total unit number that player has

Arrayist<Province> occupiedProvince: provinces that player occupied

ArrayList<Cards> cards: cards that player have

Constructors:

Public Players() : default constructor for a Players class.

Methods:

public void setName(String):

This method is a simple setter method that sets player name to the given string.

public void setplayerColor(String):

Setter method that sets player color to the given string.

public void setunitNumber(int):

Setter method that sets unit number to the given int.

public void setoccupiedProvinces(ArrayList):

Setter method that sets occupied provinces.

public void setCards(ArrayList):

Setter method that sets cards that player has.

public String getName():

This method returns player's name.

public String getplayerColor();

This method returns player's color.

public int getunitNumber():

This method returns player's unit number.

public ArrayList<Province> getoccupiedProvinces():

This method returns provinces that player occupied.

public ArrayList<Card> getCards():

This method returns cards that player has.

public void addOccupiedProvince(Province):

This method adds a province to ArrayList<Province> occupiedProvince, if the player occupies a new province. Same province can not be added twice.

public void removeOccupiedProvince(Province):

This method removes province from ArrayList<Province> occupiedProvinces if another player captures that province.

public void addCards(Card):

This method add a new card to ArrayList<Cards> cards.

public void extractCards(Card):

This method extracts a card from ArrayList<Cards> cards.

public ArrayList<Province> availableProvinces(Province):

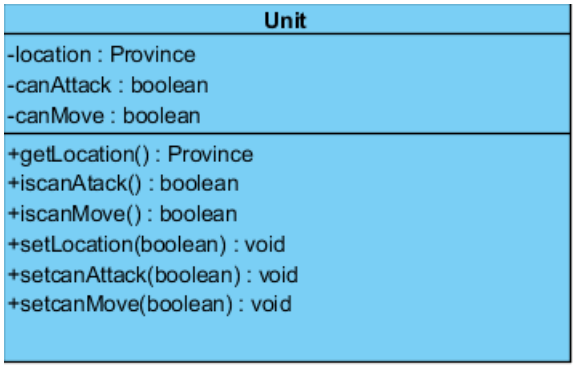
This method returns available provinces that player can attack.

public boolean canAttack(Province):

This method returns true if enemy province is adjacent to player's provinces and player has enough unit on that province to attack. If these conditions are not satisfied , returns false.

3.3.2 Game Data Manager

3.3.2.1 Unit



Attributes:

private Province location: location of the unit

private boolean canAttack: unit can attack or not.

private boolean canMove: unit can move or not.

Constructors:

public Unit(): a default constructor for Unit class.

Methods:

public void setLocation(Province):

This method sets location of the unit. It puts unit to given province.

public void setcanAttack(boolean):

This method sets an unit can attack or not.

public void setcanMove(boolean):

This method sets an unit can move or not.

public Province getLocation():

This method returns a province that unit stays.

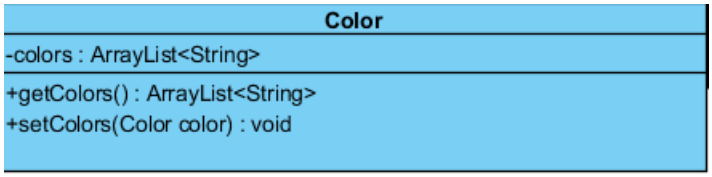
public boolean iscanAttack():

This method returns true if that unit can attack , false otherwise.

public boolean iscanMove():

This method returns true if that unit can walk , false otherwise.

3.3.2.2 Color



Attributes:

ArrayList<Color> colors: A collection of colors.

Constructor:

Public Color(): a default constructor for color class.

Methods:

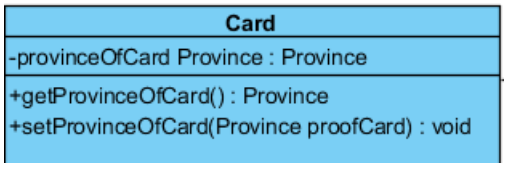
public void setColors(Color color):

This method adds a new color to ArrayList<Color> color.

public ArrayList<Color> getColors():

This method returns collection of colors.

3.3.2.3 Card



Attributes:

Province provinceOfCard: a card that province has.

Constructor:

public Card(): a default constructor for Card class.

Methods:

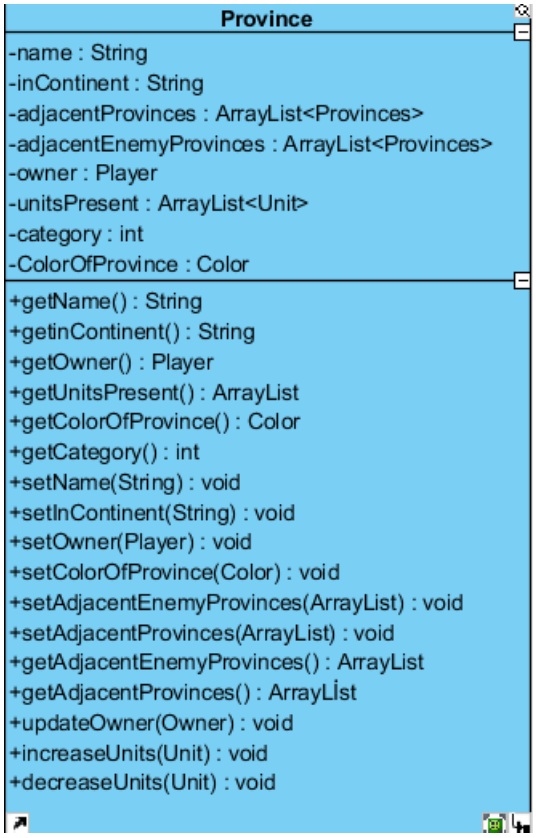
public void setProvinceOfCard(Province):

This method assigns a card to desired province.

public Province getProvinceOfCard():

This method returns a province that card has assigned.

3.3.2.4 Province



Attributes:

String name: name of the province.

String inContinent: name of the continent that includes province.

ArrayList<Provinces> adjacentProvinces: a collection of adjacent provinces.

ArrayList<Provinces> adjacentEnemyProvinces: a collection of provinces that both adjacent and enemy.

Player owner: owner of the province.

ArrayList<Unit> unitsPresent: units that stays on the province.

int category: indicator of the importance of the province.

Color ColorOfProvince: colof of the province

Constructors:

public Province(): A default constructor for Province class.

Methods:

public String getName():

This method returns name of the province.

public String getinContinent():

This method returns the continent that province belongs to.

public Player getOwner():

This method returns the owner of the province.

public ArrayList<Unit> getUnitsPresent():

This method returns units that are on that province.

public Color getColorOfProvince():

This method returns color of the province.

public int getCategory():

This method returns category of the province. Category indicates importance of the province. It ranges from 1 to 5.

public void setName(String):

This method sets name of the province.

public void setInContinent(String):

This method assigns the province to the continent.

public void setOwner(Player):

This method assigns a player(owner) to the province.

public void setColorOfProvince(Color):

This method assigns a color to the province.

public void setAdjacentEnemyProvince(ArrayList):

This method sets adjacent-enemy provinces to this province.

public void setAdjacentProvinces(ArrayList):

This method set adjacent provinces to this province.

public ArrayList<Province> getAdjacentEnemyProvinces():

This method returns the collection of provinces that are both adjacent and enemy.

public ArrayList<Province> getAdjacentProvince():

This method returns the collection of provinces that are adjacent.

public void updateOwner(Owner):

This method updates the owner of the province.

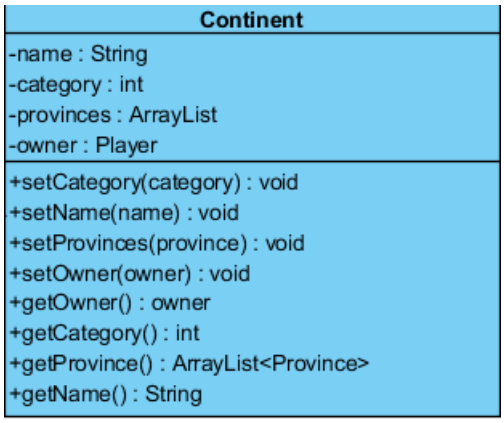
public void increaseUnits(Unit):

This method increases the number of units that are on the province.

public void decreaseUnits(Unit):

This method decreases the number of units that are on the province.

3.3.2.5 Continent



Attributes:

String name: name of the continent.

int category: category of the continent. Same as province.

ArrayList<Province> provinces: a list of provinces that belongs to this continent.

Player owner: owner of the continent.

Constructors:

public Continent(): a default for Continent class.

Methods:

public void setCategory(category):

This method sets category of the continent.

public void setName(name):

This method assigns name to the continent.

public void setProvinces(province):

This method assigns provinces to the continent.

public void setOwner(owner):

This method assigns a new owner to the continent.

public Player getOwner():

This method returns owner of the continent.

public int getCategory():

This method returns category – same as continent- of the continent.

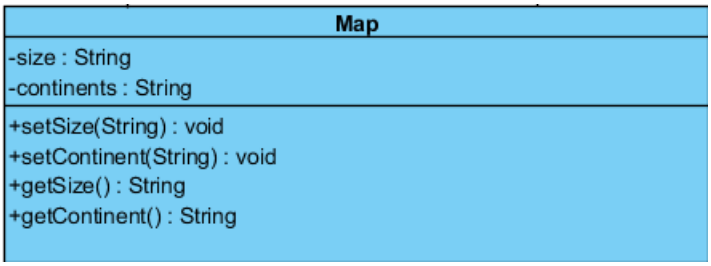
public ArrayList<Province> getProvince():

This method returns the collection of provinces that belongs to that continent.

public String getName():

This method returns name of the continent.

3.3.2.6 Map



Attributes:

String size: size of the map

String Continents: Continents that belongs to map.

Constructors:

public Map(): default constructor for Map class.

Method:

public void setSize(String):

This method sets size of the map.

public void setContinent(String):

This method assigns continents to map.

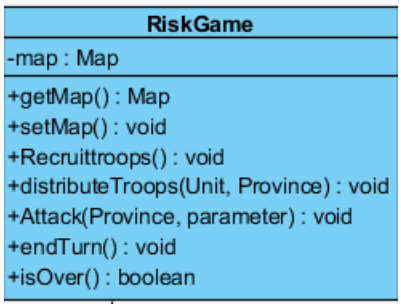
public String getsize():

This method returns size of the map.

public String getContinent():

This method returns the continents that are on the map.

3.3.2.7 RiskGame



Attributes:

Map map: map that is needed for a game

Constructors:

public RiskGame(): a default constructor for RiskGame class.

Methods:

public Map getMap():

This method returns map.

public void setMap():

This method sets a new map to initialize game.

public void distributeTroops(Unit,Province):

This method distributes troops to the players with respect to provinces and continents they have.

public void attack(Province, player):

This method enables players to attack enemy provinces.

public void endTurn():

This method ends turn and starts a new turn if the game is not over.

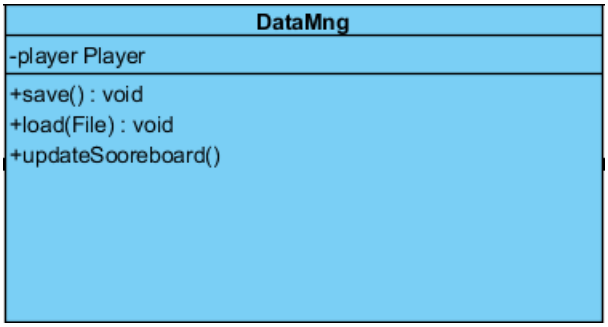
public boolean isOver():

This method returns false if the game is not over, true otherwise.

3.3.3 Data Manager

The data manager subsystem consists of the DataMng class. DataMng class is responsible for saving and loading games, initializing a loaded game and recording or uğdating some instances such as high score.

3.3.1.1 DataMng



Attributes:

Player player: a player that is needed to saved for scoreboard.

Constructors:

public DataMng(): a default constructor for DataMng class.

Methods:

public void save():

This method saves the game.

public load(File):

This method loads the game.

public void updateScoreboard().

This method updates the scoreboard.

a.

4. Low Level Design

4.1 Object Design Trade-offs

Loose Coupling vs Tight Coupling: In order to decrease dependence of the classes with each other and make it easier to test for specific code errors, We will use Loose Coupling instead of Tight Coupling.

Also, we plan to increase modifiability and maintainability of our code with the Loose Coupling.

Inheritence vs Composition: We design to make our objects in our code as independent from each other as possible. Therefore, we prefer Composition over Inheritence. Thus, our modifiability and maintainability will increase as well

4.2 Final Object Design

4.3 Packages

Java. util: This package is going to be used for data management. Data management is required in order to hold the datas such as which player holds which territory at the current stage of the game and how many troops does each player has.

JavaFXscene: This package includes sub-packages such as input, layout and image. We will use these subpackages to use handlers for mouse and keyboard events, to fit GUI objects,to support images in GUI.. Input, layout and image will accomplish these events in a respected order.

JavaFXevent: This package handles the events such as key presses and mouse clicks.

4.4 Class Interfaces

ActionListener: This interface catches any action with corresponding action events. This interface will be used to enable buttons in the game.

MouseListener: This interface catches mouse clicks. This interface is going to be used to implement the user input into the game. These inputs include: RecruitStage (for recruiting the newly gained troops), AttackStage(for attacking to different territories), DistributeStage(Distributing the remaining troops), endTurn.

KeyListener: This interface catches keyboard actions. The users entries of name for a save slot will be detected by this listener.

Javax.swing.undo: This interface allows undo/redo functions in the game if necessary.

5. References