SOFTWARE PROCESS

RISK2.0

**SOEN 6441 (Advanced Programming Practice)**

**Build 1.0**

**Risk Game**

**SUBMITTED BY:**

Dong Jae Kim (40103110)

Tianyi Liu (40093667)

Natheepan Ganeshamoothy (29335838)

**What is Coding Conventions**

Coding conventions are a set of prescriptive rules that pertain to how the code should be written, including file organization, indentation, comments, declarations and naming. They are used to improve internal qualities, maximize productivity, increase sustainability of the project (Bar 2019).

**Coding standards adopted in build 1.0/2.0 of Risk Game**

***File Organization:***

* Files are organized according to MVC architecture, where controllers were grouped inside controller folder, models inside model folder and .fxml in (Bar 2019)side views folder.
* Utilities function like file parser and validator were also grouped into their own folder called utility

***Code layout:***

* Maximized visibility of the different block by having curly braces alone on their line of code
* Blank lines were added to separate code components, such as between function and methods and declarations. Concise indentation were applied for better readability

***Naming convention:***

* Constants are named with all upper case letters and underscore
* Classes are named according and structured according MVC pattern
  + Model classes will have Model keyword
  + Controller classes will have Controller keyword
* Method names start with a lower case letter and upper case letter to separate words
* Function/method’s local variables are written entirely in lower case without underscore

***Comment convention***

* Eliminated pointless comments
* Commenting is done following conventions for Java Doc and made at the beginning of each method and file.
  + Sometimes, short description was used to describe the method that contained long algorithm
  + @see is used to link to API documentation
  + @param is used to describe parameter
  + @return is used to describe the return type
  + @author is used to describe the author of the code
  + @version is used to describe the version of the code

**Scope of build 1**

Map editor

* User driven creation of map, where we can:
  + Add/delete a territory
  + Add/delete adjacent territory
* We can load the existing file or we can create a new map from scratch. s
* In both cases, we support validating the map during load and save, and only allow game to start once the map is valid.

GamePlay

* Player is able to choose number of players between (2-6)
* Territory is randomly assigned to players and initial army count is decided using using the Risk rule (based on the number of players)
  + Each player then gets to assign initial army in round robin fashion
* During reinforcement, correct number of army is calculated from number occupied territory, continent control and validity of cards traded
  + Player can then select the territory they own to assign armies
* During fortification, we show all of the player’s occupied territory its adjacent territory, such that player can select territory to fortify using valid fortification rule

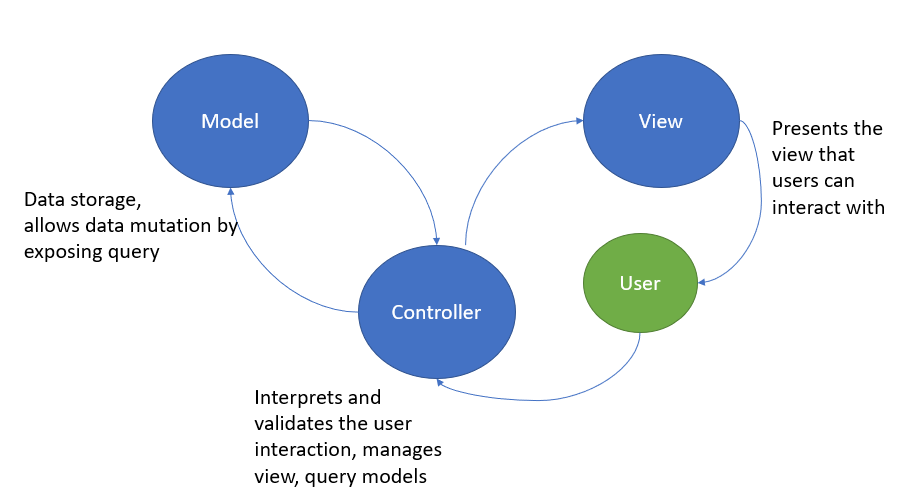
**Scope of build 2**

GamePlay

* Player is able to attack using valid attack rule
* Player is able to perform allOut move, where player proceeds with maximum amount of dice roll and only end when attacker conquers the attacked or the attacker cannot attack anymore
* Move implementation of reinforcement, attack and fortify to player class
* Implementation of phase view, card view and world domination view according to project specification

**Architecture Design**

The Risk 1.0 was build following the model view controller pattern (MVC), where we have divided the application into three interconnected parts. This is done to separate the internal representation of the data and its behavior from the way its presented in the view, and eventually to the user. Namely, the loose coupling of the model from the view allows efficient code reuse, and further maintenance and sustainability by making the code more modular.



**Figure 1: Model – View - Controller architecture used in Risk build 1.0 and 2.0**

The diagram above shows the MVC architecture of Risk build 1. The controller listens for events triggered by the user. Upon activation, the controller can get or set the state of the model, and also create or update the view. Model represents the mutable data storage and has actions that allows controller to have access to its state, allowing controller to upate the view.

**Architecture modules (Updated to include build 2.0)**

Below describes the modules of the MVC architecture for Risk build 1.

**Model**

*GamePhaseModel*

* + Class representing the state of the game phase. This class is an observable and attaches GamePhaseController observer, which is the game engine.
    - When the state of this class changes it notify the GamePhaseController to update to a new view, such as reinforcement, attack or fortification.

*MapModel*

* + Class representing the map of the RISK game after map has been validated and randomly assigned correctly to the the number of players between 2 to 6.
  + This MapModel will hold the state of the map during the game play for reinforcement, attack and fortification.

*PlayerModel*

* + Class representing the state of the players during the game play. It’s responsible for getting Player object of the current turn.

*DeckModel*

* + Class representing the state of the decks during the game play. It’s responsible sending a randomly shuffled card to the player.

*ActionModel*

* + Class representing the state of the message in different phase of the game. It’s responsible for updating the message view when an action that took place in the game has been added to its state.

**Controllers** – all the controller have access to all the models

*GamePhaseController*

* + This is the game engine that is able to change its nested controller depending on the phase of the game, as well as the controller responsible for phase view
    - It acts as an Observer for GamePhaseModel observable, which updates the view/controller to the correct phase when the state of GamePhaseModel changes
    - This controller is also responsible for phase view represented by Layout.fxml, which shows the current player in the phase and what type of phase.

*MapEditorController*

* + Mediator between MapSelectorView (MapSelector.fxml) and MapModel and PlayerModel class.
    - It’s responsible for parsing map, validating map and determining number of players and assigning territories to the players with the correct number of initial armies

*setUpController*

* + Mediator between setUpView (setup.fxml) and the PlayerModel
    - It’s responsible for allowing players to assign their initial army one by one to the their territory following round robin fashion

*ReinforcementController*

* + Mediator between ReinforcementView (Reinforcement.fxml) and MapModel and PlayerModel class.
    - It is responsible for getting the Player object of the current turn from PlayerModel
    - It then calculates the correct number reinforcement from continent control, number of occupied territory and traded cards based on information given in the Player object.
    - It is also responsible for handling user inputs of assigning armies, which is stored into the model and updates the view to show the change in the state.

*CardController*

* + Nested controller within the reinforcement controller. This is a mediator between CardView.fxml and the PlayerModel class
    - This is responsible for handling user interaction with CardView.fxml. Namey, it only allows user to select 3 cards at a time
    - It also calls validation method to make sure that the card is valid

*AttackController*

* + Mediator between AttackView (Attack.fxml) and the MapModel and PlayerModel class.
    - Currently this class is placeholder class for build 2

*ConqueredController*

* + This is a mediator between ConqueredView and PlayerModel.
    - When a player conquers a country this controller is automatically initialized to allow user to move their army to the conquered country

*FortificationController*

* + Mediator between FortificationView (Fortification.fxml) and the MapModel and PlayerModel
    - It is responsible for getting the occupied territory from PlayerModel and determines for each territory, all of the adjacent territories that you could fortify your armies.

**Entity**

*Player*

* + Blueprint to create a player

*Dice*

* + Contains operation on rolling a random number and getting maximum number of dice the attacker and defender can roll

*Continent*

* + Blueprint to create a continent

*Country*

* + Blueprint to create a country

*Card*

* + Blueprint to create a card

*Deck*

* + Contains operation on initializing deck of cards, shuffling cards and getting a card for a player

**View**

*MapSelector.fxml*

* + View for controlling UI during MapSelection phase

*Reinforcement.fxml*

* + View for controlling UI during reinforcement phase

*Attack.fxml*

* + View for controlling UI during attack phase

*Fortification.fxml*

* + View for controlling UI during fortification phase

**Utility**

*FileParser*

* + This class is responsible for parsing the input file that follows our custom format

*Validate*

* + This class is responsible for validating the Map after parsing to make sure that the country limit in the continent is fixed, map is connected and finally, the we cannot assign country to a continent that does not exist

*Output*

* + This class is responsible for creating a new file that follows the same format

**Exceptions**

*CannotFindException*

* + Custom exception related to the invalid file format

*CountLimitException*

* + Custom exception related to the invalid country limit in the continent

*DuplicatesException*

* + Custom exception related to the duplicated country assigned in the continents

**Tools and technologies used for the development of the game**

* Eclipse – IDE for game development
* JavaFX – Graphics library used to build client side UI components of Risk Game
* FXML scene builder – GUI used to speed up development of Risk UI components
* Junit5 – Testing framework used for Java development
* Javadoc – API documentation framework for automatic generation of highly browseable documentation

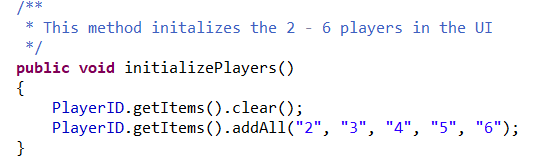
**Build 2.0 Explanation of move to the player class**

* In the build 2.0, we could not move the methods of attack phase involving valid country of attack and valid country of defend to the player class because our model already maintains a list of occupied territory for the current player, and for each the player’s territory, it also maintains list of neighbouring enemy territories and list of neighbouring occupied territories. Thus, our controller only outputs these list to the view, facilitating the human player to make valid moves.
* We also could not move methods of fortification phase involving valid country of origin to the valid country to neighbors. Similary, our model already maintains the list of the occupied territories and for each territory it also maintains valid list of neighbours, facilitating the human player to make valid fortification moves.

**Refactoring for build 2.0**

**List of refactoring targets:**

[1] Remove hardcoded number of players



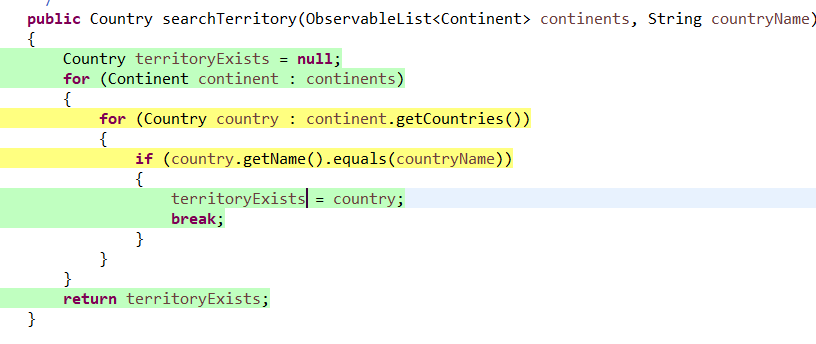
[2] Remove unnecessary for loop because .remove() method of ArrayList goes through the arraylist and removes first occurrence of the array



[3] Remove unnecessary for loop because .remove() method of ArrayList goes through the arraylist and removes first occurrence of the array



[3] Remove unnecessary for loop because .contains() method of ArrayList goes through the arraylist and returns true if the object exists



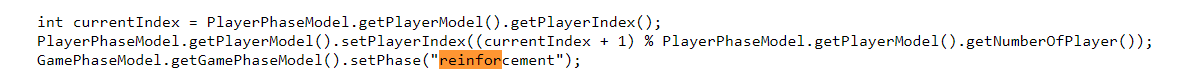
[4] Passing model into the constructor of the constroller requires a more complicate way of keeping single instance of model objects, while trying to pass these object instances to different controller during different view of the game.



[5] different phase view contains identifical component of current player and message board. Namely, improve the view architecture to allow identical components to be reused throughout different phase.



[6] Improve algorithm for getting the next valid player



**Refactored targets (Build 2.0)**

[1] Refactoring technique: Substitute Algorithm

* Why refactor?
  + This method is responsible for deleting the territory from the continent. When deleted, the territory doesn’t exist anymore, so all its connections should be deleted from other country’s neighbouring list.
  + To do this, we traversed through the neighboring list of deleted territory, and for each of the neighbours, we visited their neighbor list to find the deleted territory. When found, we removed we removed the deleted territory from their list.
  + However, I realized that this algorithm does not require another for loop to traverse through the neighbours of the neighbours because the ArrayList.remove() method already traverses the list to remove the first occurrence of the object.
  + Namely, all we needed was one for loop to go through the neighbor list of deleted territory and use arraylist.remove() to remove the country from their list.



[2] Refactoring technique: Extract super controller in the user interface

* Why refactor?
  + Three phase controllers, such as reinforcement, attack and fortification, all have common common functionality of showing player name, current game phase and game messages.
  + Therefore, I created a GamePhaseController super class, and moved all the identical fields (player name, current game phase and game message board) to the this super controller. Rest of the controllers for different phase are now nested inside this GamePhaseController, and their functionalities are more specialized for each of the phase because super controller is responsible for taking care of common functionalities.

BEFORE:



* + This is a view for reinforcement before refactoring. It has its own field for current player and its own message board. This is the same for attack and fortification

AFTER:



* + Above shows a view where we pulled up the identifical fields from the different phase controller.

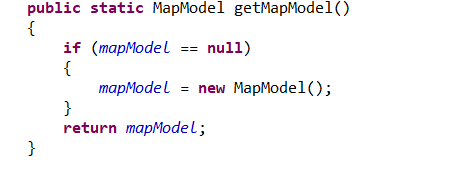
[4] Refactoring technique: Replacing model constructor with singleton factory

* Why refactor?
  + Prior to using singleton, different controller class gained access to the models by passing models into to their constructors. Namely, at different phase of the game, the model objects were passed between different controller objects to keep single instance of model classes. When a controller required multiple models objects then we had to pass all of the model objects into the constructor.
  + This made our application less flexible.

BEFORE:



* + To fix this we used singleton using public static method that returns the single instance of the model class. Namely, if it’s null create a new instance, if not just returns an existing object.
  + This allowed easier way to keep a single instance of models within our application. Morevoer. all the controller has access to the same model objects anytime in the game, making our code easier to maintain and cope with changes.

AFTER: 

[4] Refactoring technique: Substitute Algorithm

* Why refactor?
  + Hard to tell what the code is doing
  + This piece of code is used several times in the game play so its better to make it into a method
  + Current piece of code only gets the next player, but doesn’t check if the next player is valid. Namely, skips players who has already lost
* How to refactor
  + Created a method called setNextPlayer() that is easier to understand from the name of the method
  + We put the method inside PlayerPhaseModel to increase cohesion of modules (keeps together things that are related)
  + The method is basically a while loop that keeps incrementing the currentPlayerIndex until it finds a valid player that is the player who hasn’t lost yet.
  + With the updated index, all we need to do is getCurrentPlayer() to return the correct player