SOFTWARE PROCESS

RISK1.0

**SOEN 6441 (Advanced Programming Practice)**

**Build 1.0**

**Risk Game**

**SUBMITTED BY:**

Dong Jae Kim ()

Tianyi Liu ()

Natheepan Ganeshamoothy ()

Ayush Dave ()

**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 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

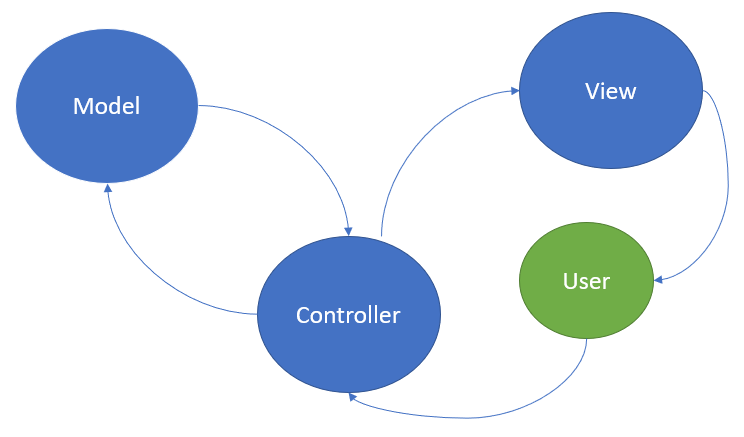
* User driven creation of map from scratch
  + Add/delete a continent
  + Add/delete a territory
  + Add/delete adjacent territory
* Editing currently existing map
* Check for the validity of the map using 3 incorrect map files

GamePlay

* Player is able to choose number of players
* Territory is assigned to players using the Risk rule
  + The initial army based on number of player is assigned automatically to players in the round robin fashion in the implementation
* Calculate correct number of reinforcement based on number of occupied territory, continent control and number of cards
* Player can assign country to reiniforcement to territory
* Player can assigh army with valid fortification rules

**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 interal 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**

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**

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.
    - 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 assigned correctly to the the number of players that could be 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 the next Player object.

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

*GamePhaseController*

* + Mediator between GamePhaseModel and all of the phases of the game play like reinforcement, attack and fortification
    - It acts as an Observer for GamePhaseModel observable.
    - Updates the view to the correct phase when the state of GamePhaseModel changes

*MapEditorController*

* + Mediator between MapSelectorView (MapSelector.fxml) and MapModel and PlayerModel class.
  + It is generates the correct MapModel and PlayerModel from the MapSelectorView (MapSelector.fxml), which will be used for the rest of the game
    - It’s responsible for parsing map, validating map and determining number of players and assigning territories to the players with correct number of armies
    - It also determines the round robin scheduling of the player turns.

*ReinforcementController*

* + Mediator between ReinforcementView (Reinforcement.fxml) and MapModel and PlayerModel class.
    - It is responsible for getting the next player in the turn and that player’s occupied territories from PlayerModel class. It subsequently calculates the correct number reinforcement (Continent control, occupied territory and traded cards) based the information given from PlayerModel and updating the army count in the MapModel, during which all of the state changes are rendered to the ReinforcementView.

*AttackController*

* + Mediator between AttackView (Attack.fxml) and the MapModel and PlayerModel class.

*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 countries that you own that is arrived by the available path.
    - It then limits one fortification move

**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 reading the map format and parsing it into MapModel
  + It alsom checks for duplicate countries and invalid file format

*Validate*

* + This class is responsible for validating the MapModel 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 outputting the edited map after it has been validated

**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