Build 1 Risk Game Document

Team 14

Group members:

Yann Kerichard 40059813

Yueshuai Jiang 40046639

Che-Shao Chen 40055392

Contents

The Design of The Architecture of The Risk Game

1. Structural design

In our project, we are using the MVC architecture design. The structure separates into 3 main parts for the system, called Controller, Model, and View.

1.1 Controller

Controller is the for interface between view and model, receiving input from user and initiate response by calling corresponding objects. After receiving the inputs, the controller will call other models to complete the process according to the input. In the risk game, we have made 3 controllers, they are GameController, MainController, and MapEditorController.

1.1.1 MainController

In the MainController, it is the menu model of the system when first launched, it gives players 2 options at the beginning, either play game or edit map.

1.1.2 GameController

In the GameController, there is one method , excute , to perform the whole phase of the gameplay, including startup phase, reinforcement phase, attack phase and fortification phase.

1.1.3 MapEditorController

This controller will be called when the player choose to edit map. Any operation related to map editing, will be implemented in this controller.

1.2 Model

Model is the component where all the data related logic respond to, the model will manage all behavior and data during the whole process of the application, including responding request to access the state of data from view, and change state request from controller.

1.2.1 GamePlay

In the GamePlay package, we created classes and their related attributes that will be needed during the game , which includes card, dices, phase and player.

1.2.2 Map

The Map package contains classes like Continent.class, Country.class, Map.class, and MapEditor.class. The first two classes deal with the basic data and logic for two main map properties in the game. The Map class is one of the most important class in the project, since no operation can run without map, it carries the most functions among other classes, which includes functions for checking validation of maps, data change, saving, and loading maps, set country armies and so on. The functions in MapEditor class can let users create, editing, and save map files.

1.2.3 Utilities

Utilities package includes useful classes that can be widely reused by other classes. Such as FileHandler class for reading and writing files. Message class for showing messages to the user. Random class dealing with random index values. The StringAnalyzer class focus on the string function logic.

1.3 View

View is the component that interact directly with players and model, acting as an interface to show players the information and possible option they can choose.

1.3.1 Common

The Common package contains the most generalized views in the game, which includes View, MainMenuView, MapSelectionView.
View is an abstract class which contains the input validation.
MainMenuView is the main menu interface
MapSelectionView is the interface when player is selecting maps.

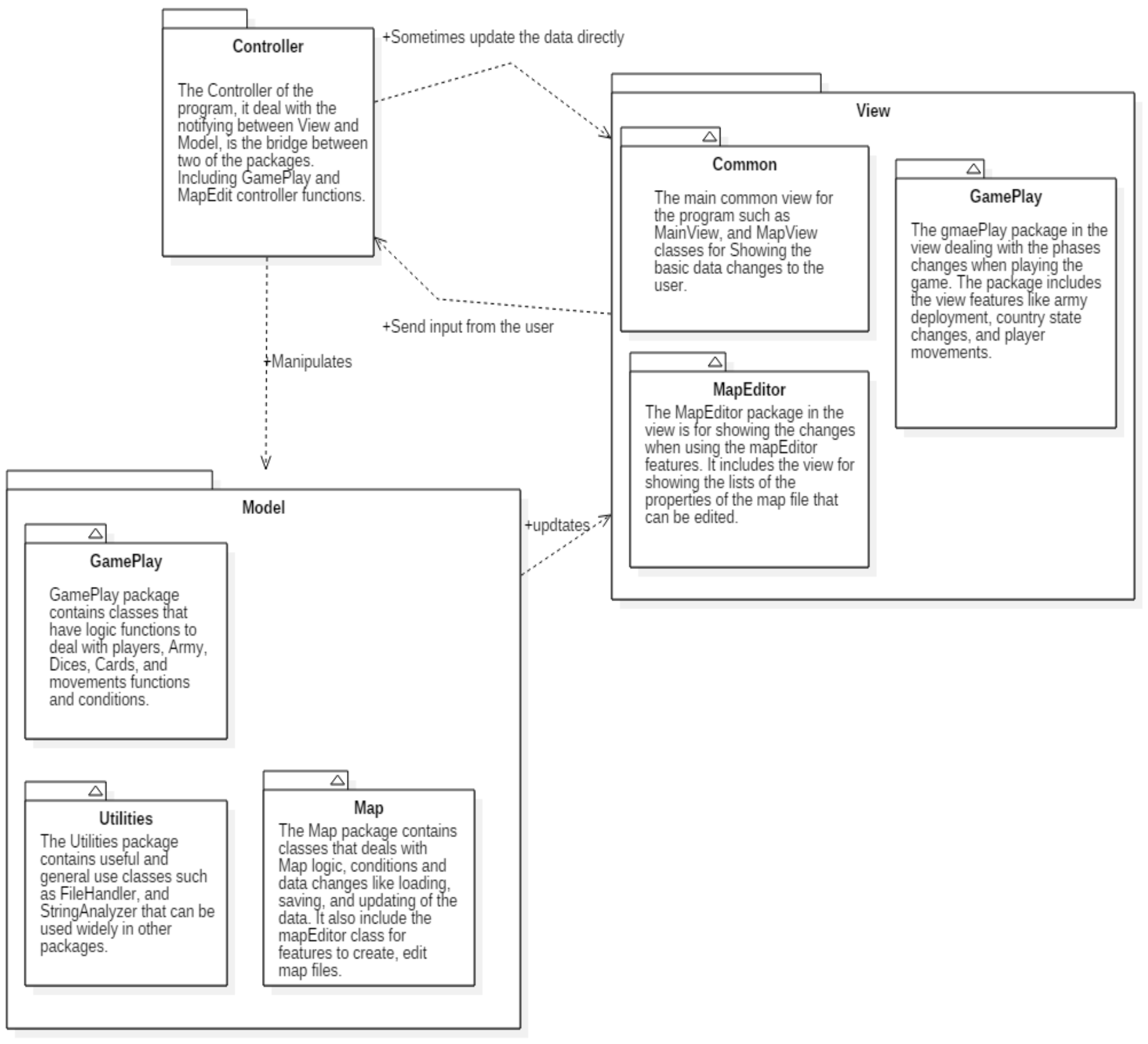
1.3.2 GamePlay

The gamePlay package contains all interfaces during the game play, which includes startup, reinforcement, attack, fortification, phase change, card exchange, map viewing, world domination view, winner view.

1.3.3 MapEditor

The MapEditor class fulfill functions for showing the map properties to help users create and editing maps.

2. diagram of the architecture design

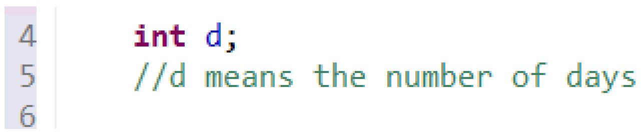

Controller
The Controller of the program, it deal with the notifying between View and Model, is the bridge between two of the packages. Including GamePlay and MapEdit controller functions.
+Sometimes update the data directly
View
Common
The main common view for the program such as MainView, and MapView classes for Showing the basic data changes to the user.
GamePlay
The gmaePlay package in the view dealing with the phases changes when playing the game. The package includes the view features like army deployment, country state changes, and player movements.
+Send input from the user
+Manipulates
MapEditor
The MapEditor package in the view is for showing the changes when using the mapEditor features. It includes the view for showing the lists of the properties of the map file that can be edited.
Model
+updtates
GamePlay
GamePlay package contains classes that have logic functions to deal with players, Army, Dices, Cards, and movements functions and conditions.
Utilities
The Utilities package contains useful and general use classes such as FileHandler, and StringAnalyzer that can be used widely in other packages.
Map
The Map package contains classes that deals with Map logic, conditions and data changes like loading, saving, and updating of the data. It also include the mapEditor class for features to create, edit map files.

3. Code Standards

Normative naming can make the code more readable.

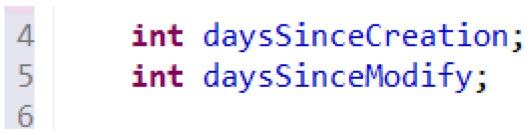
3.1 Naming should reveal the schematic

bad:



```
int d;
//d means the number of days
```

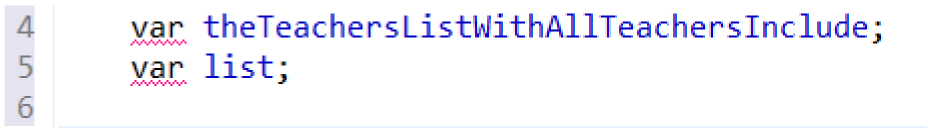
good:



```
int daysSinceCreation;
int daysSinceModify;
```

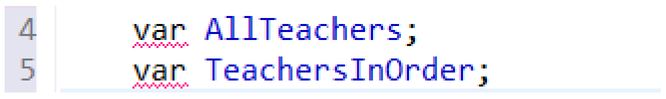
3. 2 Name should in right length

bad:



```
var theTeachersListWithAllTeachersInclude;
var list;
```

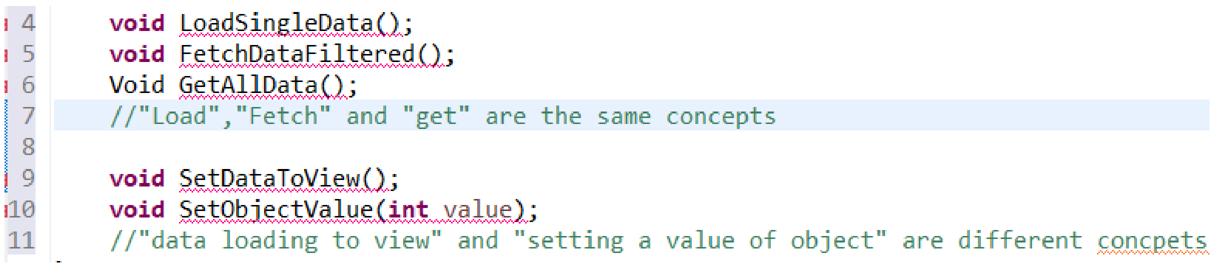
good:



```
var AllTeachers;
var TeachersInOrder;
```

3. 3 Same concept should be named in same word

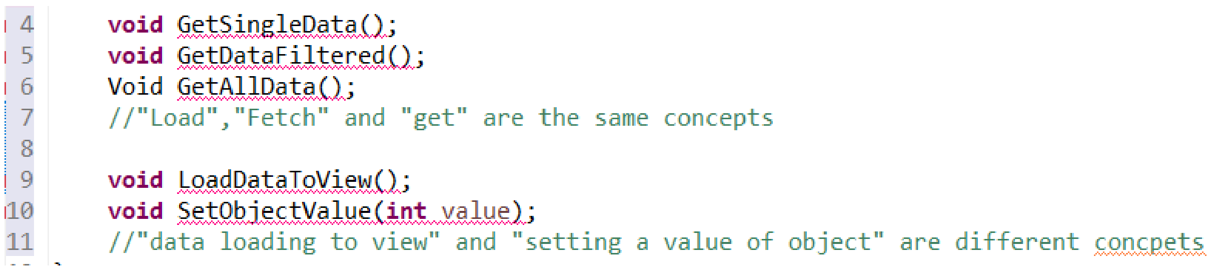
bad:



```
void LoadSingleData();
void FetchDataFiltered();
Void GetAllData();
//"Load","Fetch" and "get" are the same concepts

void SetDataToView();
void SetObjectValue(int value);
//"data loading to view" and "setting a value of object" are different concpets
```

good:

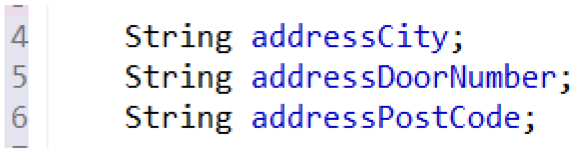


```
    void GetSingleData();
    void GetDataFiltered();
    Void GetAllData();
    //"Load","Fetch" and "get" are the same concepts

    void LoadDataToView();
    void SetObjectValue(int value);
    //"data loading to view" and "setting a value of object" are different concpets
```

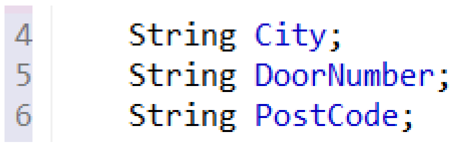
3. 2 Naming by using words related to the domain when naming

bad:



```
    String addressCity;
    String addressDoorNumber;
    String addressPostCode;
```

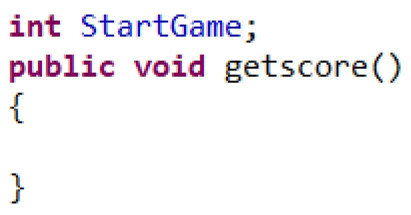
good:



```
    String City;
    String DoorNumber;
    String PostCode;
```

3.3 Method and Attribute names start with a lowercase letter and use uppercase letters to separate words

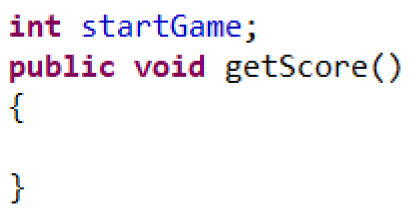
bad:



```
int StartGame;
public void getscore()
{

}
```

good:

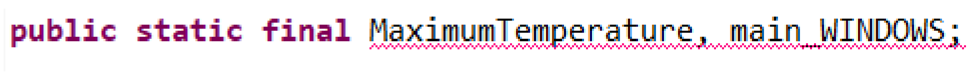


```
int startGame;
public void getScore()
{

}
```

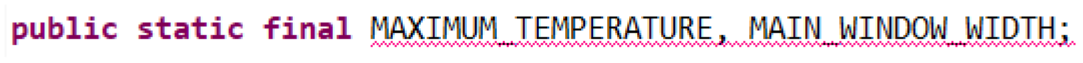
3. 4 Constants use upper case letters with underscores between words

bad:



```
public static final MaximumTemperature, main_WINDOWS;
```

good:



```
public static final MAXIMUM_TEMPERATURE, MAIN_WINDOW_WIDTH;
```

4. Code Layout

4. 1 Use the line break wisely

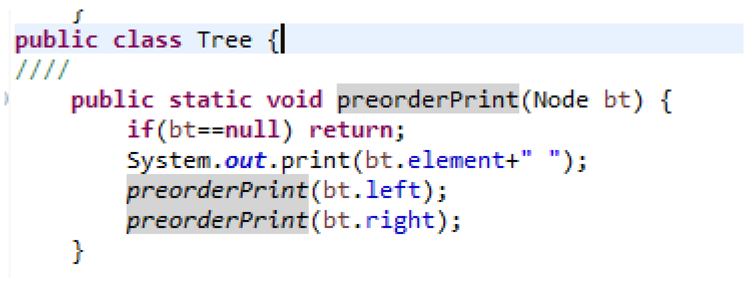
Breaking the code lines only when: The statements exceed the column limit, or,

One statement is over.

4. 2 Indent style

Indent size is 4 columns, nested continuation indent may add 4 columns at each level

and always use tab instead of space

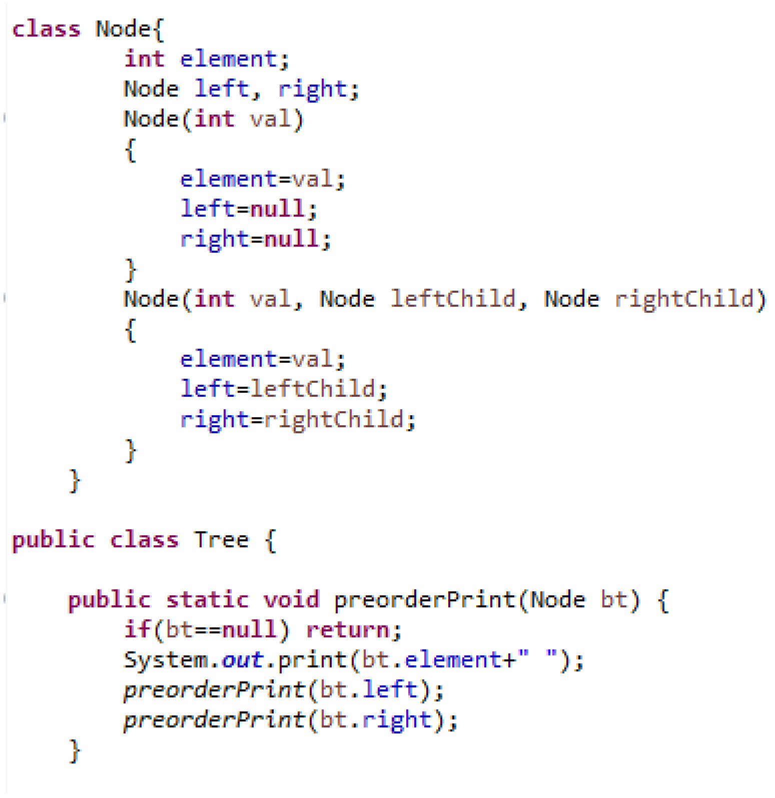


```
public class Tree {
////
    public static void preorderPrint(Node bt) {
        if(bt==null) return;
        System.out.print(bt.element+" ");
        preorderPrint(bt.left);
        preorderPrint(bt.right);
    }
```

4. 3 Blank lines

Blank lines can be added between major sections of a long and complicated function,

between public, protected, and private sections of a class declaration, between class declarations in a file , and, between function and method definitions.



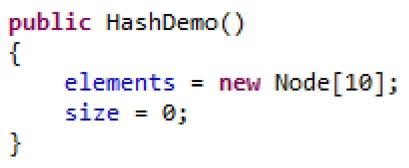
```
class Node{
        int element;
        Node left, right;
        Node(int val)
        {
            element=val;
            left=null;
            right=null;
        }
        Node(int val, Node leftChild, Node rightChild)
        {
            element=val;
            left=leftChild;
            right=rightChild;
        }
    }

public class Tree {

    public static void preorderPrint(Node bt) {
        if(bt==null) return;
        System.out.print(bt.element+" ");
        preorderPrint(bt.left);
        preorderPrint(bt.right);
    }
```

4. 4 Use of space

use space between operators and equals.

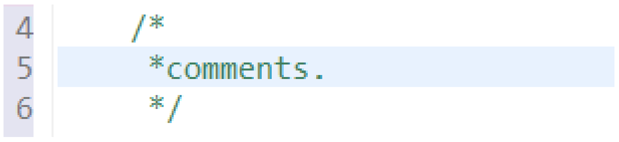


```
public HashDemo()
{
    elements = new Node[10];
    size = 0;
}
```

5. Commenting

5. 1 Block Comments

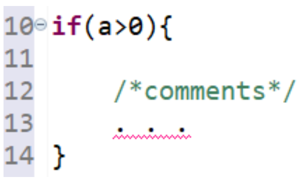
The block comment should be preceded by a blank line and have an asterisk “*” at the beginning of each line. The block comments usually appear at the beginning of each file and before each method. If the block comments inside a function or method then it should be indented to the same level as the code they describe.



```
    /*
     *comments.
     */
```

5. 2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can’t be written in a single line, it should follow the block comment format.

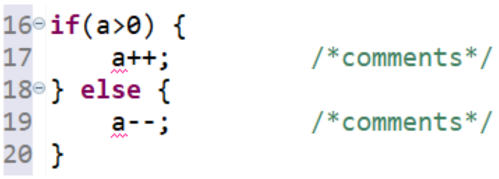


```
if(a>0){

    /*comments*/
    . . .
}
```

5. 3 Trailing Comments

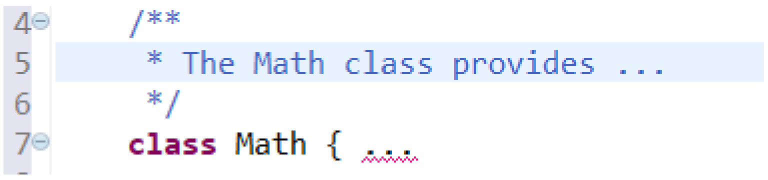
The comments can appear on the same line as the code they describe if they are very short, but they should be set apart from the code. If more than one short comment appears in a code block, they should all be indented to the same tab setting.



```
if(a>0) {
    a++;            /*comments*/
} else {
    a--;            /*comments*/
}
```

5. 4 Documentation Comments

Documentation comments should be set inside the comment delimiters /**...*/ and appear before the declaration.



```
    /**
     * The Math class provides ...
     */
    class Math { ...
```

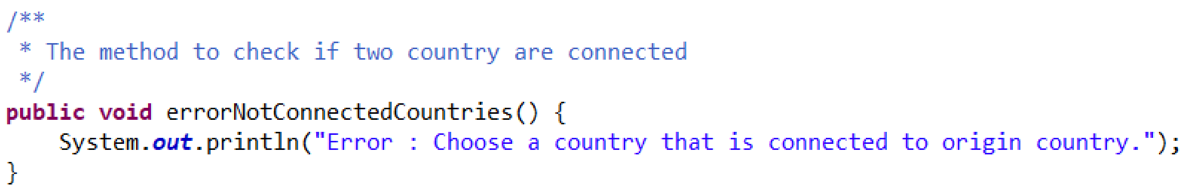
6. How you came up with a list of potential refactoring targets?

We executed the following process just after the release of build 2, and before beginning to implement the new requirements:

- Inspecting most **important logical units**
- Inspecting **requirements asked**
- Analyzing and inspecting classes having **relationships** with other
- **Defining responsibilities** and discussing about what each class should be able to do or not to do. Then, we compared with the actual responsibilities of the classes in our project
- Looking inside each class and to ensure the **methods names** quickly give an idea of what is going on
- Finally, by inspecting each file in each package and checking if **condition statements** could be refactored. Also checking warnings in Eclipse allowed us to remove/change unused lines of code. All refactoring operations were hitting parts of the system for which tests were designed.
- After refactoring, we launched test suites.

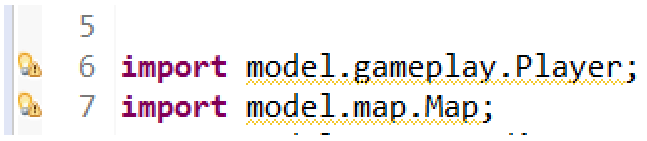
List of 3 refactoring operations :

- fortificationView.errorNotConnected() becomes fortificationView.errorNotConnectedCountries();



```
/**
 * The method to check if two country are connected
 */
public void errorNotConnectedCountries() {
    System.out.println("Error : Choose a country that is connected to origin country.");
}
```

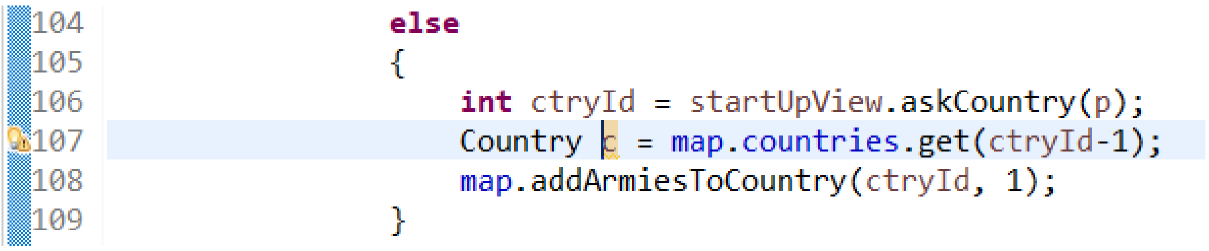
- Removing unused imports everywhere



```
import model.gameplay.Player;
import model.map.Map;
```

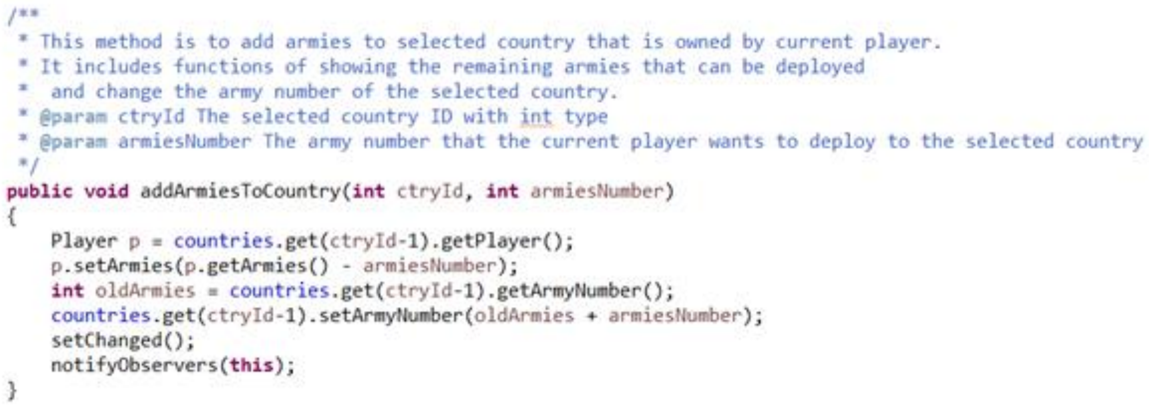
Unused imports removed in MapEditorController

- Removing unneeded computations



```
            else
            {
                int ctryId = startUpView.askCountry(p);
                Country c = map.countries.get(ctryId-1);
                map.addArmiesToCountry(ctryId, 1);
            }
```

Unneeded variable and computation in GameController class
Moving method to other location



```
/**
 * This method is to add armies to selected country that is owned by current player.
 * It includes functions of showing the remaining armies that can be deployed
 *  and change the army number of the selected country.
 * @param ctryId The selected country ID with int type
 * @param armiesNumber The army number that the current player wants to deploy to the selected country
 */
public void addArmiesToCountry(int ctryId, int armiesNumber)
{
    Player p = countries.get(ctryId-1).getPlayer();
    p.setArmies(p.getArmies() - armiesNumber);
    int oldArmies = countries.get(ctryId-1).getArmyNumber();
    countries.get(ctryId-1).setArmyNumber(oldArmies + armiesNumber);
    setChanged();
    notifyObservers(this);
}
```

Moving addArmiesToCountry method from map to country class

7. References

http://google.github.io/styleguide/javaguide.html

https://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141999.html#216