

Build 3 Risk Game Document

Team 14

Group members:

Yann Kerichard 40059813

Yueshuai Jiang 40046639

Che-Shao Chen 40055392

Contents

1. Structural design	4
1.1 Controller.....	4
1.1.1 MainController.....	4
1.1.2 GameController.....	4
1.1.3 MapEditorController	4
1.1.4 TournamentController	4
1.2 Model	4
1.2.1 GamePlay	5
1.2.2 Strategy	5
1.2.3 Map.....	5
1.2.4 Utilities	5
1.3 View	5
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.	5
1.3.1 Common	5
1.3.2 GamePlay	6
1.3.3 MapEditor.....	6
2.diagram of the architecture design	7
3.Code Standards	8
3.1 Naming should reveal the schematic	8
3.2 Name should in right length	8
3.3 Same concept should be named in same word.....	8
3.2 Naming by using words related to the domain when naming	9
3.3 Method and Attribute names start with a lowercase letter and use uppercase letters to separate words	9
3.4 Constants use upper case letters with underscores between words	10
4. Code Layout.....	10
4.1 Use the line break wisely	10
4.2 Indent style	10
4.3 Blank lines	11
4.4 Use of space.....	11
5. Commenting.....	12
5.1 Block Comments.....	12
5.2 Single-Line Comments.....	12
5.3 Trailing Comments.....	12
5.4 Documentation Comments	12
6. How you came up with a list of potential refactoring targets?	13
List of potential refactoring targets.....	13
3 selected refactoring operations	15

7. References	18
---------------------	----

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 4 controllers, they are GameController, MainController, MapEditorController, and TournamentController.

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.

1.1.4 TournamentController

This controller will be called when the player choose the tournament mode. In this mode, the player can choose 1~5 maps, how many games will play in each map, and 2~4 AI players. Once the player finished choosing those options, the mode will run the input and show the final result for each map and game to the user.

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 Strategy

Strategy is a sub-package of the Gameplay package. This package is dealing with strategy classes for players. It include 4 AI strategy classes such as Aggressive AI, Benevolent AI, Random AI, and Cheater AI. Human class in this package is dealing with the human player's actions. The ConcreteStrategy class is for the basic functions that needed to be used for other classes in this package. The Strategy class is for the combine of four strategies for players and easy to let programmer to import the strategy classes.

1.2.3 Map

The Map package contains classes like Continent.class, Country.class, Map.class, MapEditor.class, and MapChecker.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. The MapChecker class is mainly for checking whether if a map passes all the tests, and to determine if it is playable or not.

1.2.4 Utilities

Utilities package includes useful classes that can be widely reused by other classes. Such as FileHandler class for reading and writing files. Rng 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.

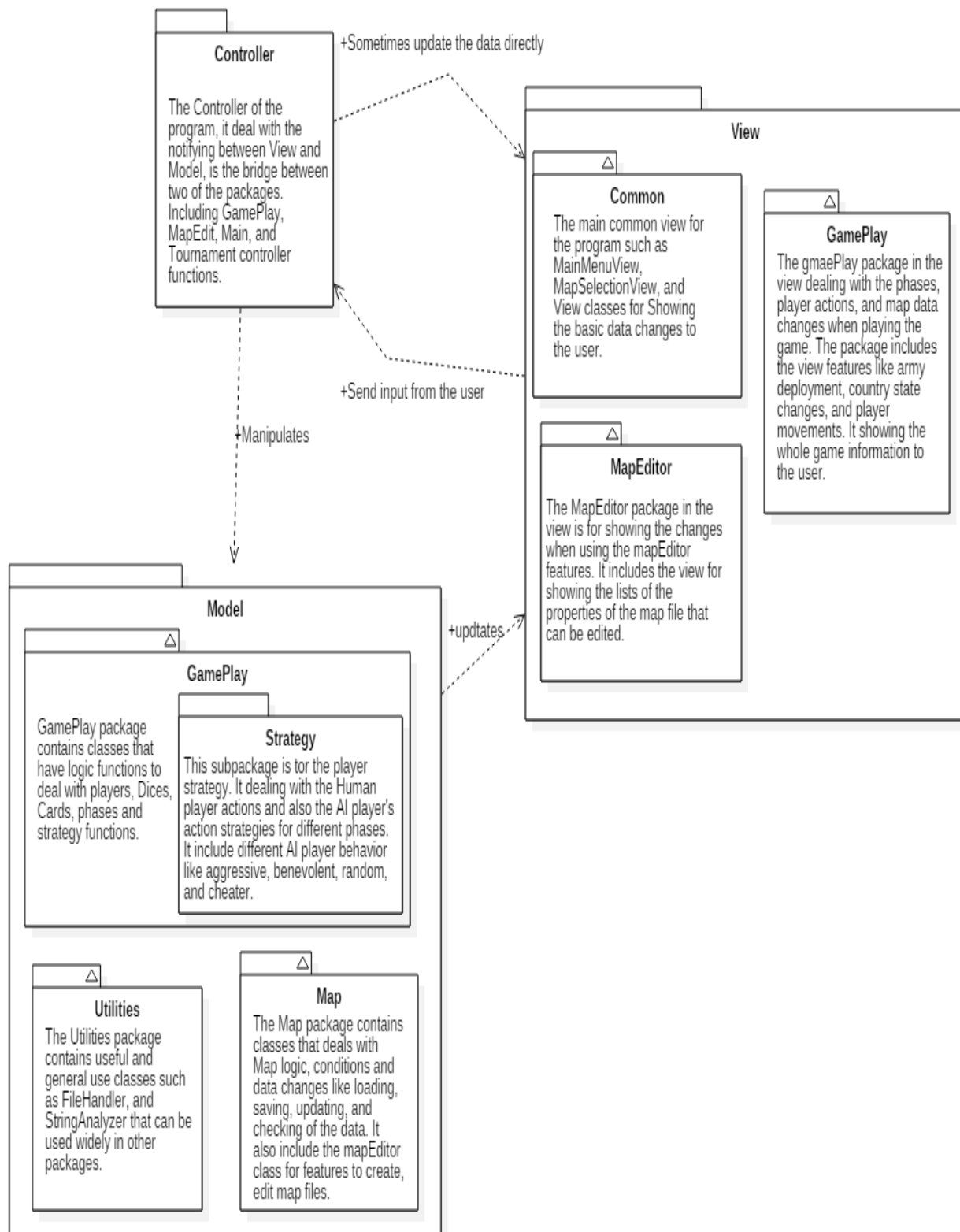
1.3.2 GamePlay

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 view class fulfill functions for showing the map properties to help users create and editing maps. It includes 3 view classes such as EditorMenuView, EditView, and MapEditorView.

2. diagram of the architecture design



3.Code Standards

Normative naming can make the code more readable.

3.1 Naming should reveal the schematic

bad:

```
4    int d;  
5    //d means the number of days  
6
```

good:

```
4    int daysSinceCreation;  
5    int daysSinceModify;  
6
```

3.2 Name should in right length

bad:

```
4    var theTeachersListWithAllTeachersInclude;  
5    var list;  
6
```

good:

```
4    var AllTeachers;  
5    var TeachersInOrder;
```

3.3 Same concept should be named in same word

bad:

```
4    void LoadSingleData();  
5    void FetchDataFiltered();  
6    Void GetAllData();  
7    // "Load", "Fetch" and "get" are the same concepts  
8  
9    void SetDataToView();  
10   void SetObjectValue(int value);  
11   // "data loading to view" and "setting a value of object" are different concpets
```


good:

```
4 void GetSingleData();
5 void GetDataFiltered();
6 void GetAllData();
7 // "Load", "Fetch" and "get" are the same concepts
8
9 void LoadDataToView();
10 void SetObjectValue(int value);
11 // "data loading to view" and "setting a value of object" are different concepts
```

3.2 Naming by using words related to the domain when naming

bad:

```
4 String addressCity;
5 String addressDoorNumber;
6 String addressPostCode;
```

good:

```
4 String City;
5 String DoorNumber;
6 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.

```
4      /*
5      *comments.
6      */
```

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.

```
10 if(a>0){
11
12     /*comments*/
13     .....
14 }
```

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.

```
16 if(a>0) {
17     a++;          /*comments*/
18 } else {
19     a--;          /*comments*/
20 }
```

5.4 Documentation Comments

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

```

4=  /**
5   * The Math class provides ...
6   */
7=  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 potential refactoring targets

- **Map** : This class is the class that contains the most code in our project, so obviously, we have to consider it as a primary target. It is interacting with a lot of other components and it is a core class of the program. We found some **dead code** inside, **wrong naming** methods, and **duplicate conditional fragments** that should be consolidated. It is also subject to be a **Bloater** : **Long class**, with **Long Methods**.

Example of duplicate conditional fragment :

```

public boolean checkConnectedContinents()
{
    if(this.countries == null || this.countries.size() == 0)    return false;
    if(this.continents == null || this.continents.size() == 0)  return false;
}

```

- **GameController** class : Another really important class that plays a major role in the whole program. This controller contains very complex logic, with a lot of conditional expressions that coordinates the interactions between the model classes and the views during the game, increasing the probability to find some mistakes.

Example of duplicate conditional fragment :

```

if(attackMode == 1) //All-out
{
    p.attack(map, attackerCtry, defenderCtry);
    phase.setAction("P"+p.getNumber()+" used " + attackerCtry.getName() + " to attack "
}
else {
    //Classic
    Dices dices = new Dices(attackerCtry.getArmyNumber(), defenderCtry.getArmyNumber());
    int attackerDices = attackView.askAttackerDices(p, dices.getAttackerMaxDices());
    int defenderDices = attackView.askDefenderDices(defenderCtry.getPlayer(), dices.getI
    dices.setDicesNumber(attackerDices, defenderDices);

    p.attack(map, attackerCtry, defenderCtry, dices);
    phase.setAction("P"+p.getNumber()+" used " + attackerCtry.getName() + " to attack "
}

```

Examples of Dead code :

```

} else {
    int selectedArmies = reinforcementView.askArmiesNumber(p);
    int index = map.countries.get(countryNumber-1).getArmyNumber(); Index is not used, the whole line can be removed !
    p.reinforcement(map, countryNumber, selectedArmies);
    phase.setAction("P"+p.getNumber()+" reinforced "+ selectedArmies+" army in "+map.countries.get(countryNumber-1).getName()+"\n");
}

104         else
105         {
106             int ctryId = startUpView.askCountry(p);
107             Country c = map.countries.get(ctryId-1);
108             map.addArmiesToCountry(ctryId, 1);
109         }

```

- **Player** class : This class contains the logic for all the phases of the game, so it is pretty important to optimize it. Moreover, this class is a very good potential refactoring target since build 3 requires to bring new features working around this class (players with different strategies).

Example of a simplification of the Conditional :

```

public void attack(Map map, Country attackerCtry, Country defenderCtry)
{
    do{
        Dices dices = new Dices(attackerCtry.getArmyNumber(), defenderCtry.getArmyNumber());
        battle(map, dices, attackerCtry, defenderCtry);
    }while(attackerCtry.getArmyNumber() > 1 && defenderCtry.getArmyNumber() > 0); // Continue until no attack is possible

    /* Resolving battle result : conquering a country */
    if(defenderCtry.getArmyNumber() == 0) {
        conquer(defenderCtry);
    }
}

```

Same condition : If statement could be pulled up inside the while loop, with a return statement if the condition occurs. So we can remove the duplicate condition in the while

- **MapEditorController** class : Since this class contains the logic flow of the map editor, we may still need to consider it as a refactoring target. Its role is not negligible. Moreover, we may find some mistakes because the class was made at the beginning of the build 1 and may need an update regarding the evolution of other classes.

Example of Dead code :

```

private void deleteCountry()
{
    boolean deleted = false;
    int maxInput = mapEditor.getMaxInputNumber();
    int ctryNumber = editView.askCountryNumber(maxInput);
    deleted = mapEditor.deleteCountry(ctryNumber);

    if(!deleted)
    {
        // editView.errorDeletingCountry();
    }
}

```

Example of replacing the variable with method call :

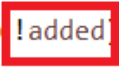
```

public void addContinent()
{
    boolean added = false;

    String countinentName = editView.askContinentName();
    int bonus = editView.askBonus();
    added = mapEditor.addContinent(countinentName, bonus);

    if(!added)
    {
        editView.errorAddingContinent();
    }
}

```

 We can remove added and put the method instead

- **MapEditor** class : This class did not move since the beginning of the project. It may need to be refreshed. It is subject to contain bad smells such as **nested conditional with guard clauses**, and many other mistakes.
- **Country** class : Also an important class that is called everywhere in the project.
- **Continent** class : Same reason as for Country class.

3 selected refactoring operations

We decided to focus our refactoring operations on 3 targets : **Map**, **GameController** and **Player**. We are going to give you an example of the operation for each of the 3 targets and let you see the solution we applied.

We decided to focus on **Map** and **GameController** because those 2 classes are the 2 cornerstones of the gameplay. The map is constantly used and has to interact with a lot of components, and the game controller is controlling the execution flow of the game. These 2 classes really need to be as clean as possible because their impact is huge on the game. Moreover, those 2 classes are containing a lot of code and we need to ensure they are not bloated.

Also, we decided to include the **Player** class in our selection because the build 3 is all around this class. It asks to implement new player strategies, and in the build 2 we brought a lot of changes to this class, which probably brought some bad smells. So, because of all the changes brought by build 2 and the importance of this class in build 3 it makes a really good refactoring target. Time has come to clean it up and to refactor the code inside.

Map : Duplicate conditional fragment

```

public boolean checkConnectedContinents()
{
    if(this.countries == null || this.countries.size() == 0) return false;
    if(this.continents == null || this.continents.size() == 0) return false;
}

```

Solution :

```

if(this.countries == null || this.countries.size() == 0
|| this.continents == null || this.continents.size() == 0) return false;

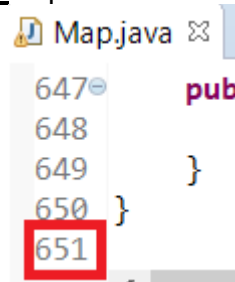
```

Benefits :

Eliminates duplicate control flow code. Combining multiple conditionals that have the same "destination" helps to show that you are doing only one complicated check leading to one action.

Large class (Extract class)

Map class is wearing too many (functional) hats.



```

647 pub
648
649 }
650 }
651

```

Solution :

Extracting map class verification methods to a new MapChecker class.

```

> Map.java
> MapChecker.java

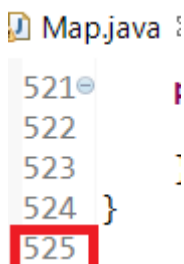
```

MapChecker contains all the methods that ensure that the map is good and playable.

```

6 public class MapChecker {
7     private Map map;
8
9     public MapChecker(Map newMap) {}
10
11
12     * check if the map is valid by:
13
14     public boolean check() {}
15
16
17     * check if there is any country and continent in the map
18
19     public boolean checkPlayableMap() {}
20
21
22     * The method checks if there are empty continents which does not have any country signed under those continents
23
24     public boolean checkNoEmptyContinent() {}
25
26
27     * To check if the map is connected : each country must be able to access all other countries.
28
29     public boolean checkConnectedMap() {}
30
31
32     * To check if the continents are connected : each continent must be able to access all other continent.
33
34     public boolean checkConnectedContinents() {}
35 }
36

```



```

521
522
523
524 }
525

```


Benefits :

High cohesion (GRASPs) : Single-responsibility classes are more reliable and tolerant of changes.

This refactoring method will help maintain adherence to the Single Responsibility Principle. The code of your classes will be more obvious and understandable.

GameController : Duplicate conditional fragment & Dead code

```
if(attackMode == 1) //All-out
{
    p.attack(map, attackerCtry, defenderCtry);
    phase.setAction("P"+p.getNumber()+" used " + attackerCtry.getName() + " to attack "-
}
else {
    //Classic
    Dices dices = new Dices(attackerCtry.getArmyNumber(), defenderCtry.getArmyNumber());
    int attackerDices = attackView.askAttackerDices(p, dices.getAttackerMaxDices());
    int defenderDices = attackView.askDefenderDices(defenderCtry.getPlayer(), dices.getDefenderMaxDices());
    dices.setDicesNumber(attackerDices, defenderDices);

    p.attack(map, attackerCtry, defenderCtry, dices);
    phase.setAction("P"+p.getNumber()+" used " + attackerCtry.getName() + " to attack "-
}

else {
    int selectedArmies = reinforcementView.askArmiesNumber(p);
    int index = map.countries.get(countryNumber-1).getArmyNumber();
    p.reinforcement(map, countryNumber, selectedArmies);
    phase.setAction("P"+p.getNumber()+" reinforced " + selectedArmies+" army in "+map.countries.get(countryNumber-1).getName()+"\n");
}
```

Solution :

```
if(attackMode == 1) //All-out
{
    p.attack(map, attackerCtry, defenderCtry);
}
else {
    //Classic
    Dices dices = new Dices(attackerCtry.getArmyNumber(), defenderCtry.getArmyNumber());
    int attackerDices = attackView.askAttackerDices(p, dices.getAttackerMaxDices());
    int defenderDices = attackView.askDefenderDices(defenderCtry.getPlayer(), dices.getDefenderMaxDices());
    dices.setDicesNumber(attackerDices, defenderDices);

    p.attack(map, attackerCtry, defenderCtry, dices);

    phase.setAction("P"+p.getNumber()+" used " + attackerCtry.getName() + " to attack " + map.countries.get(defenderCtry.getNumber()-1).getName()
}
else {
    int selectedArmies = reinforcementView.askArmiesNumber(p);
    p.reinforcement(map, countryNumber, selectedArmies);
    phase.setAction("P"+p.getNumber()+" reinforced " + selectedArmies+" army in "+map.countries.get(countryNumber-1).getName()+"\n");
}
```

Benefits :

Reduced code size

Simpler support

Leading to one action

Player : Simplification of the Conditional

```
public void attack(Map map, Country attackerCtry, Country defenderCtry)
{
    do{
        Dices dices = new Dices(attackerCtry.getArmyNumber(), defenderCtry.getArmyNumber());
        battle(map, dices, attackerCtry, defenderCtry);
    }while(attackerCtry.getArmyNumber() > 1 && defenderCtry.getArmyNumber() > 0); // Continue until no attack is possible

    /* Resolving battle result : conquering a country */
    if(defenderCtry.getArmyNumber() == 0) {
        conquer(defenderCtry);
    }
}
```

Same condition : If statement could be pulled up inside the while loop, with a return statement if the condition occurs. So we can remove the duplicate condition in the while

Solution :

```

public void attack(Map map, Country attackerCtry, Country defenderCtry)
{
    do{
        Dices dices = new Dices(attackerCtry.getArmyNumber(), defenderCtry.getArmyNumber());
        battle(map, dices, attackerCtry, defenderCtry);

        /* Resolving battle result : conquering a country */
        if(defenderCtry.getArmyNumber() == 0) {
            conquer(defenderCtry);
            return;
        }
    }while(attackerCtry.getArmyNumber() > 1);    // Continue until no attack is possible
}

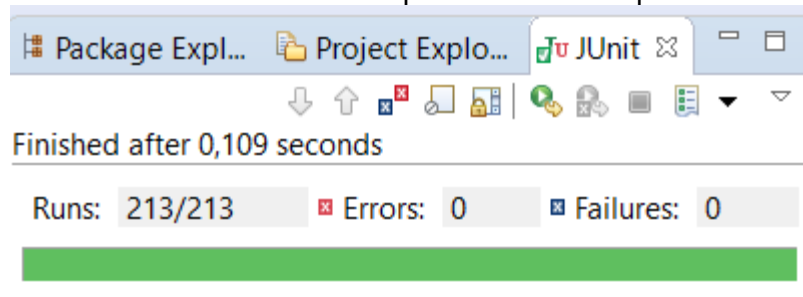
```

Benefits :

Simpler loop condition

Easier to understand

And know what ? All tests still pass after those operations !



7. References

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

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