SOEN 6441 - Advanced Programming Practices
Risk Game
Coding Conventions[1]

Nasim Adabi[1],Michael Hanna[2],Pinkal Shah[3] and Wasim Alayoubi[4]

[1]40079444 nasim.adabi@gmail.com
[2]40075977 michael_baligh@yahoo.com
[3]40102983 pinkalshah270594@gmail.com
[4]40053306 wasim.alayoubi@mail.concordia.ca

Group ID Group_U_D

[1] Updated 2019-12-01 (Delivery 3)

1 Package Structure

In our coding convention the project packages are structured to follow the MVC Architectural style. This makes it easy for the team members to know exactly where to find different classes of the game.

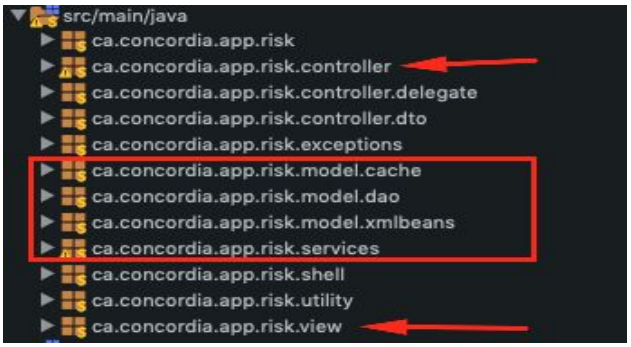

Fig 1. Project Structure

2 Class Names

The classes are named using the 'camel-case' notation with the **exception** that the **first letter** of the class name is also in **uppercase**. For example,

GameService.java
GameBusinessDelegate.java

3 Variables and Constants

3.1 Variables

All the variables are self-sufficient to infer their meaning from the name itself and follow the 'camel-case' coding convention. For example,

numberOfArmies
countryName
player2Add
player2Remove

The use of global variables has been discouraged since our architecture depends on the model's cache, hence, whenever anything global needs to be changed, we access the RunningGame instance and operate on the values from there.

3.2 Constants

The constants are all uppercase with an underscore(_) in between to connect the names. The naming is self-sufficient to understand the meaning. For example,

NONE_DEFAULT_VALUE
COMMAND_EXECUTED_SUCCESSFULLY

4 Exception Classes

All the error messages are propagated through the custom exception class. Moreover, 'stacktrace' command can be used in case of more clarity required for any exception/error raised during runtime. The class is:

RiskGameRuntimeException.java

5 Code layout

We have used the approach of minimizing code length by appending the curly braces to the statement that precedes it. Moreover, all the names of classes, methods, variables and constants are defined so that they are self-explanatory without the use of any unnecessary comments.

```
public void reinforceInitialization(int playerID) {

  int numberOfCountries = RunningGame.getInstance().getCountries().getList().size();

  PlayerDaoImpl playerDaoImpl = new PlayerDaoImpl();

  PlayerModel activePlayerModel = playerDaoImpl.findById(RunningGame.getInstance(),
      RunningGame.getInstance().getCurrentPlayerId());

  int reinforcementArmies = 0;
  boolean fullContinentOccupy = false;

  if (Math.floor((float) numberOfCountries / 3) > 3) {
    reinforcementArmies = Math.floorDiv(numberOfCountries, 3);
  } else {
    reinforcementArmies = 3;
  }
```

Fig 2. Curly braces on the same line

6 Unit Tests

We are using the latest version of JUnit,JUnit 5 for the testing framework. The test folder structure is inline with the implementation of the project. This helps in easily identifying which chunk of code is being unit tested.

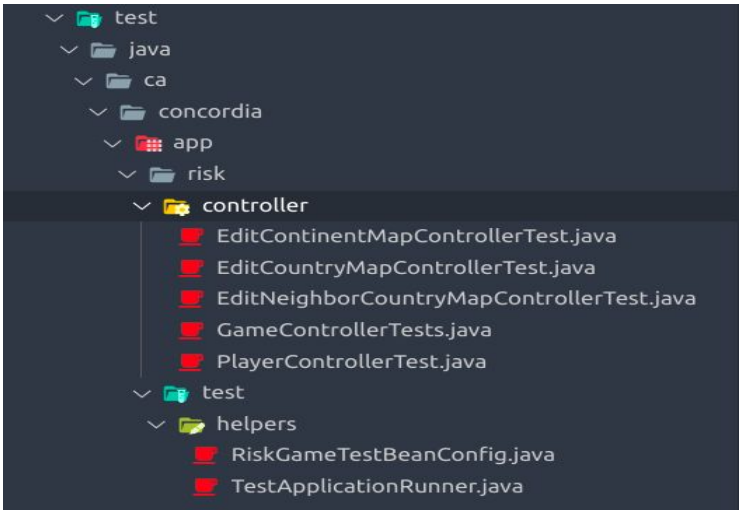

Fig 3. Test Folder Structure

7 Observer classes

We are using the observer pattern to help notifying GUI interfaces components whenever a model state change happens. We should always use the Java built-in observer implementation whenever applicable.

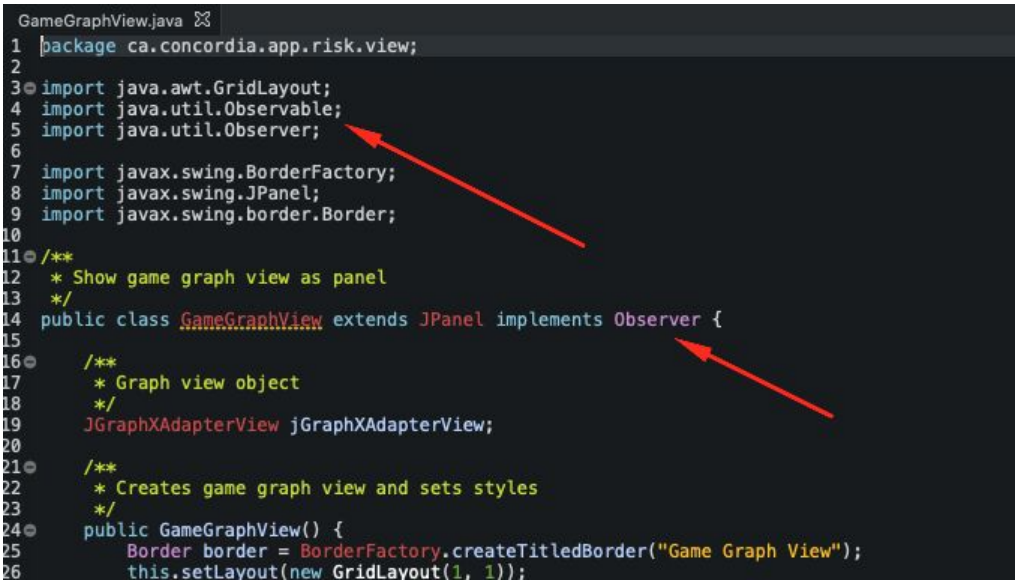

Fig 4. Observer class

8 Builder classes

The Builder design pattern should be used to build and save the game to default xml file.

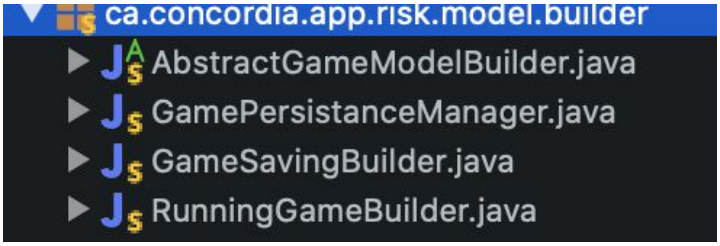

Fig 5. Builder class

9 Adapter classes

The Adapter design pattern should be used to support loading and saving map files

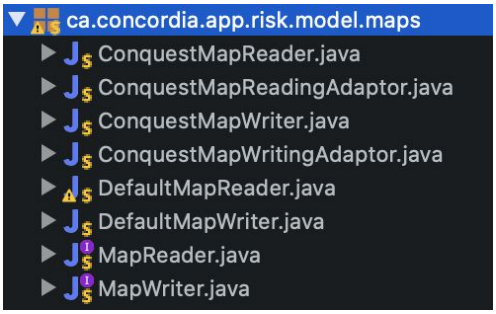

Fig 6. Observer class

8 Strategy classes

The Strategy design pattern should be used to support creating player modes (strategies)

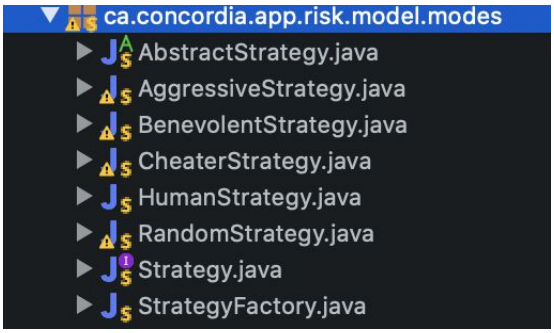

Fig 7. Strategy classes