M3 Actual Architecture - Risk Game

Qasim Naushad – 5658624
Prakash Gunasekaran - 6399185
Prabhjot Kaur Sekhon - 6473318
Rajwinder Kaur - 6282490
Harsahiljit Singh - 6405975
Yash Paliwal - 6562566

Class Diagram of Actual System

Our Ideal Architecture VS Risk's Actual Architecture

Figure 1 is the domain model of our system, showing the general structure of the system which we extracted from the project itself, after the detail analysis of the project we produced the actual class diagram shown in Figure 2 by using ObjectAid plug-in in Eclipse.

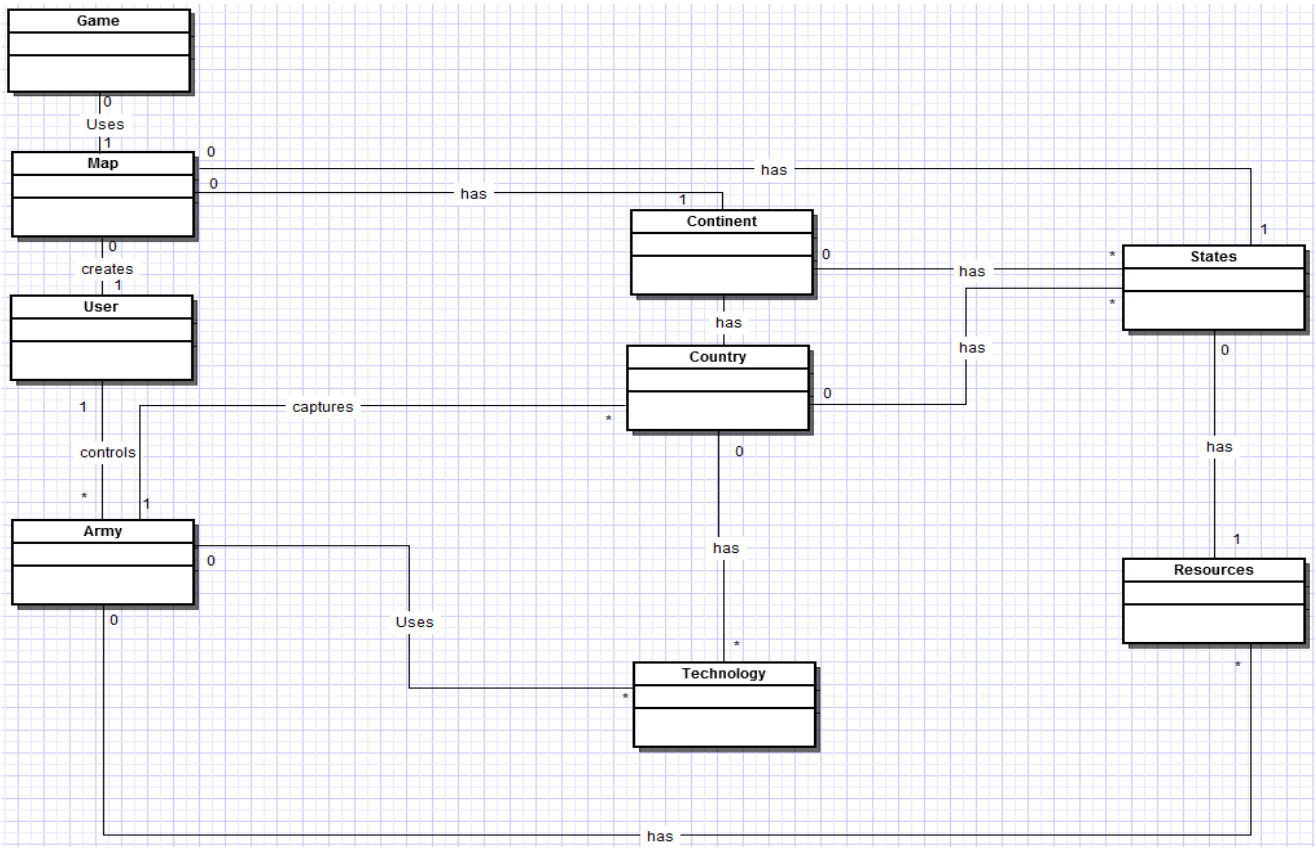


Figure 1. Our Domain Diagram from M2

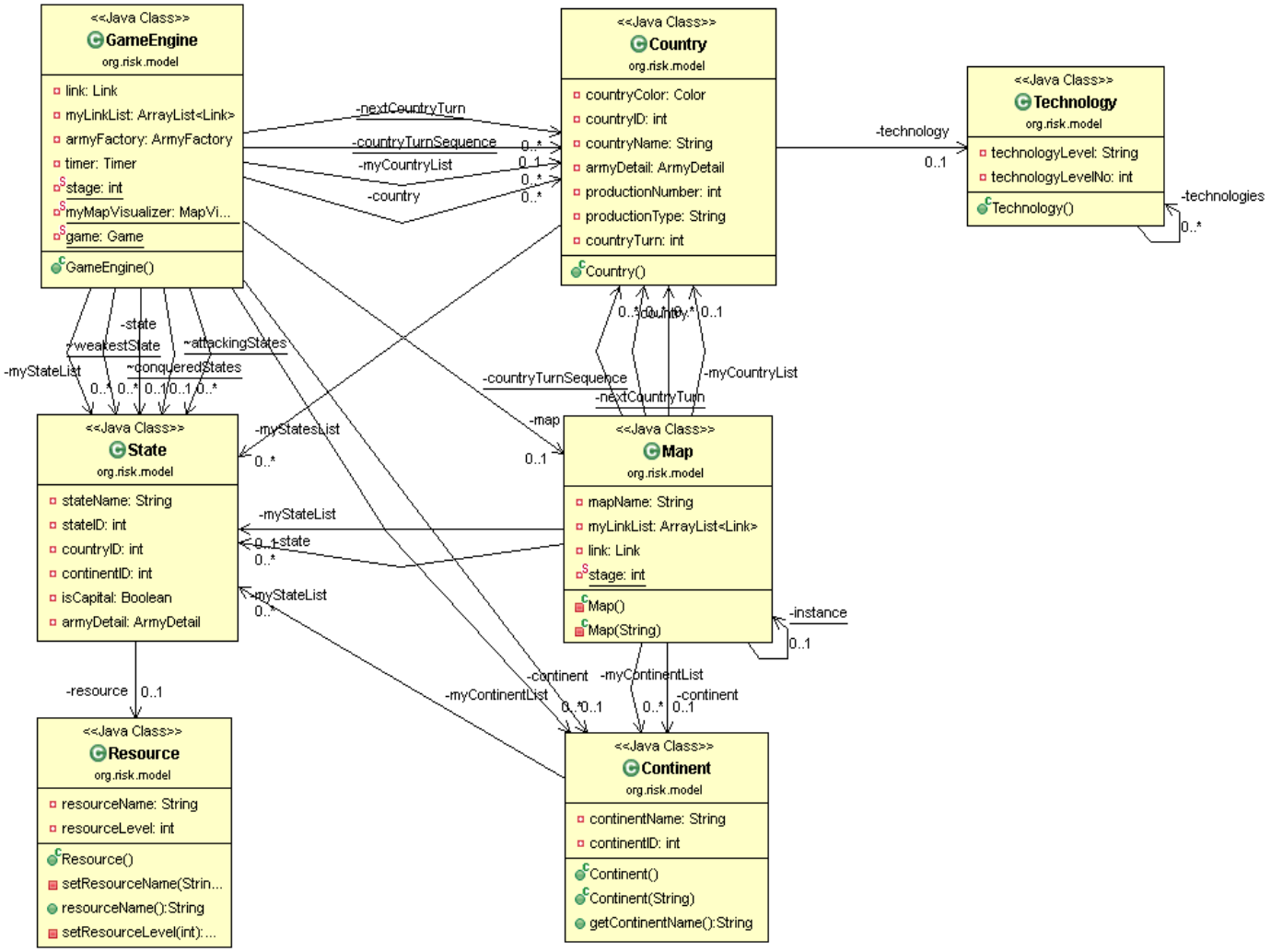


Figure 2. Risk's Actual Architecture

Most of the classes were mapped from domain model to class diagram for our project, some of the examples are, state class, continent class, Game Engine and map class etc, but some of the very important conceptual classes like Player (user) were not mapped, and all of the responsibilities of a player were meshed up with Country class which was making the architecture very hard to comprehend. Conceptual class army and army types were implemented in the project by using factory pattern design to take advantage of polymorphism because there were about ten different types of armies.

Classes and Relationships between them

Game engine class handles the work flow of whole project. Map class carries whole information about map like continents, countries, states, links etc. The game engine class uses the map class to get map information and distribute it to other classes. Now as the class game engine holds all the data of map other classes instead of going to map classes uses game engine class to get map information which reduces redundancy.

Source Code

```
import java.util.Set;

import javax.sound.midi.Receiver;

……
public class GameEngine {
	private Map map;
	private ArrayList<Continent> myContinentList;
	private ArrayList<Country> myCountryList;
	private ArrayList<State> myStateList;
	private ArrayList<Link> myLinkList;
	……
	private void populateMap() {
		map = Map.getInstance();
		myContinentList = map.getMyContinentList();
		myCountryList = map.getMyCountryList();
		myStateList = map.getMyStateList();
		myLinkList = map.getMyLinkList();
		nextCountryTurn = map.getNextCountryTurn();
		countryTurnSequence = map.getCountryTurnSequence();
		stage = map.getStage();
	}
	……
	public class Map extends Observable {
	private String mapName;
	private ArrayList<Continent> myContinentList;
	private ArrayList<Country> myCountryList;
	private ArrayList<State> myStateList;
	private ArrayList<Link> myLinkList;
```

```
public ArrayList<Continent> getMyContinentList() {
        return myContinentList;
 }

public ArrayList<Country> getMyCountryList() {
        return myCountryList;

 }public ArrayList<State> getMyStateList() {

        return myStateList;
 }

……
```

Code Smell and Possible Refactoring

To find code smells in the project we used a plug-in named Jdeodrant, it found two types of code smells which were in abundance in our project, Feature envy, and Long methods. Besides those code smells there was another major problem with the architecture of the system which was the country class, it was playing two roles, first to handle the information related to countries in the map, and the 2nd was to handle the player responsibilities like taking turns. The last code smell which we found out was that the class named GameEngine which is responsible to control the whole flow of the simulation is very poorly structured, the if else statements alone make the code hard to read and understand. We noticed that the project has no sense of responsibility in its architecture, there are few classes with all the code in them and the rest of them are fairly empty. After analyzing the project manually and by using tools like objectAid, Jdeodrant and McCabe we came up with a very basic solution, to evenly distribute responsibilities among classes.

See: http://www.objectaid.com/download

See: http://marketplace.eclipse.org/content/jdeodorant#.UVCwPhyk968

See: http://marketplace.eclipse.org/content/eclipse-metrics-plugin-continued#.UVCx2Ryk968

Our Proposed Refactoring:

1. Extracting another class named player from country class to divide responsibilities like taking turns and keeping country information
2. Extracting another class named Attack from GameEngine which will take over the responsibility of attack methods because the class GameEngine is overwhelmed.
3. Moving methods like (remove army strength(),addArmyDetail() from class State to class Army to eliminate feature envy
4. Moving methods from class Country to Army to eliminate feature envy
5. Lastly we extracted methods from the If-Else statements from the GameEngine, continent and state classes to make the code more readable

See: http://sourcemaking.com/refactoring

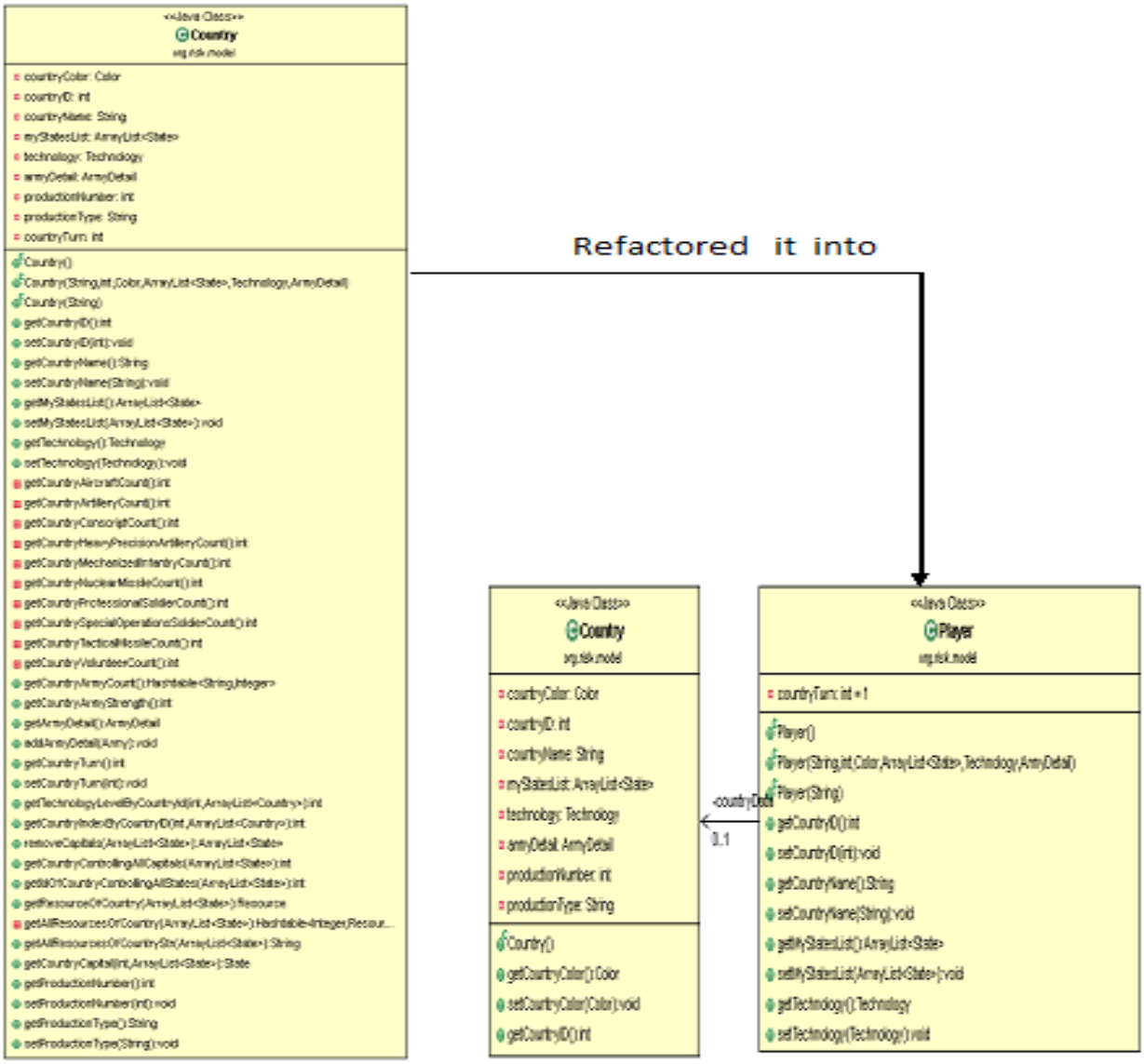
Identified Patterns:

After careful analysis of the project we identified the patterns used in it:

1. Factory pattern, used by the org.risk.model.army package
2. Singleton pattern, used by the Map.Java class
3. Observer pattern, used by the org.risk.view
4. And the whole System is based on model view controller(MVC)

See: http://sourcemaking.com/design_patterns

Before Refactoring:



Country class had both the responsibilities as we mentioned above which means it had low cohesion, and by doing the refactoring by using jdeodrant plugin we reduce feature envy, and by doing so we increase cohesion between classes. Both the classes have individual responsibilities now, and code comprehension is much easier.