M4- Risk Game

M4_Risk Game: Individual Part

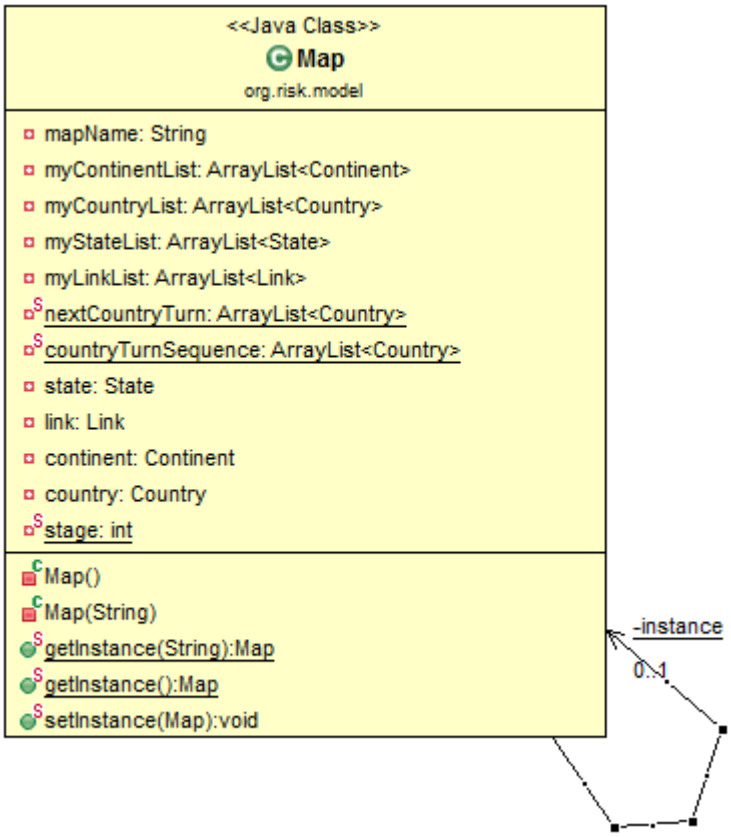
Qasim Naushad 5658624

The project we have is a game named risk, it's using model view controller as the architectural design which helps separate the Logic layer from the UI layer, it also promotes code re-usability and most of all it accommodates the developer when making changes. I used plug-in like Jeodrant and ObjectAid to better understand the structure of the system, after analysis I came up with 4 basic patterns which my project is using

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)

Singleton

The game is played on a single map, the way the map class is used is that it creates a single instance of the map and keeps using and modifying the same map object throughout the flow of system Below is the class diagram of map class, showing singleton in action



The purpose of map class is to read all the inputs in the provided map, like number of states, continents , links between them etc, the map class loads all that data into the system to use and distribute, rather than loading the input each time its needed the map class does it once. The pattern singleton provides the required functionality by making it impossible to make more than one instance of the map class, in this way the same map instance is used all around the system and keeping track of map becomes easier, rather than making more than one instances and keeping track of all of them.

The working code of the map class is shown below and you can see all the necessary methods types like static map instance, private constructor which are all a part of singleton

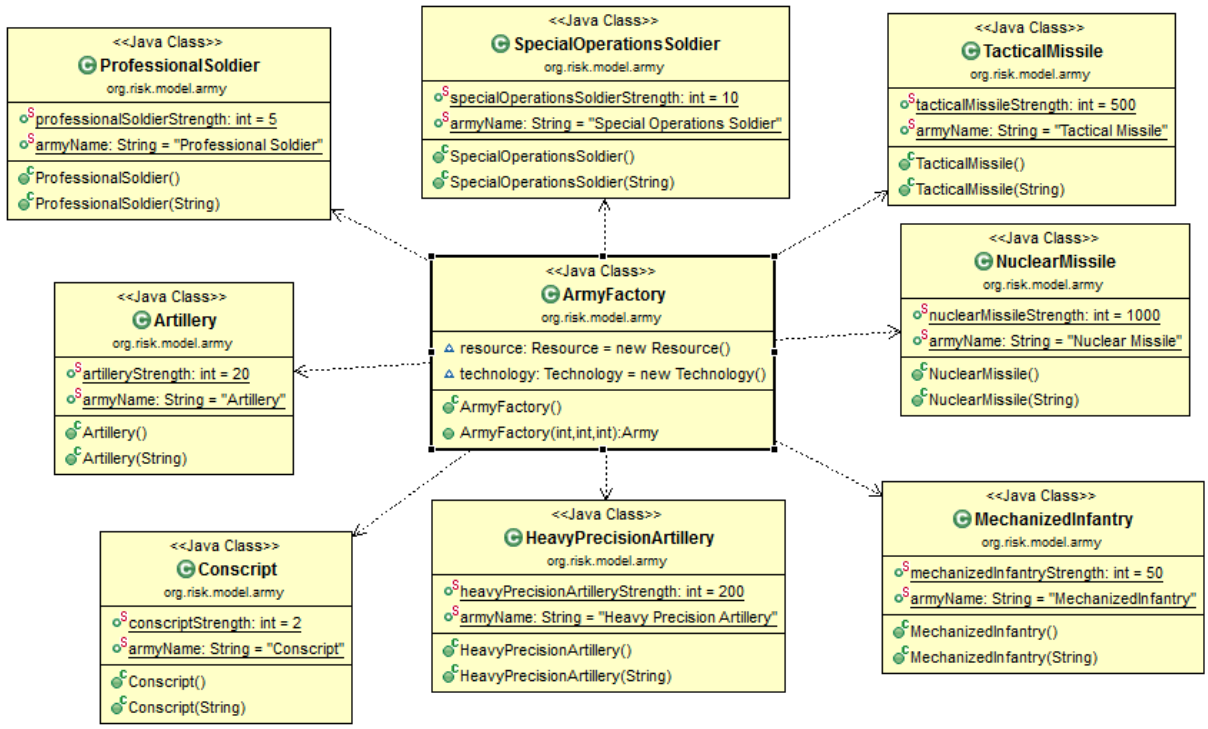
```
public class Map extends Observable {
	private static Map instance = null;
	....
	private Map() {
	.....
	}

	public static Map getInstance() {
		if (instance == null) {
			instance = new Map();
		}
		return instance;
	}
```

```
	public static void setInstance(Map map) {
		instance = map;
		}
.....
	}
```

Link to singleton pattern: http://sourcemaking.com/design_patterns/singleton

Factory pattern

in our project each country can command different types of armies, and each army has its own properties, but the basic structure of the classes is the same, each type of army at most adds on extra method to itself for that reason all types of army classes extend (inherit) from Army class taking use of polymorphism (OOP), in this scenario using factory pattern was a good choice, using polymorphism and class hierarchy to extend functionality rather than hard coding every class with all methods and dynamically deciding which class to initialize. The dynamic initialization is where factory pattern comes in. The diagram below shows how the package under observation is interacting within and using factory pattern.



The code below shows how the factory method depends on two attributes, technology and resources and based on that it chooses which army type to initialize

```
public class ArmyFactory {

	......

	public Army ArmyFactory(int resourceLevel, int technologyLevel,int
numberOfStates) {
```

```
                if (resourceLevel ==resource.noResource().resourceLevel()) {
                return new Volunteer();
            }

            else if (resourceLevel == resource.metalResource().resourceLevel()
                        && technologyLevel == technology.technologyLevelBasic()
                                    .technologyLevelNo()) {
                return new Conscript();
            }


            else if (resourceLevel == resource.energyResource().resourceLevel()
                        && technologyLevel == technology.technologyLevelBasic()
                                    .technologyLevelNo()) {
                return new ProfessionalSoldier();
            }

        .......
            else {
                return new Army();
            }
        .......
}
```

Link to factory pattern: http://sourcemaking.com/design_patterns/factory_method

M4- Risk Game (Individual Part) Submitted By Prakash Gunasekaran (Student ID- 6399185)

Identified Pattern in Army package (org.risk.model.army) as Factory Pattern

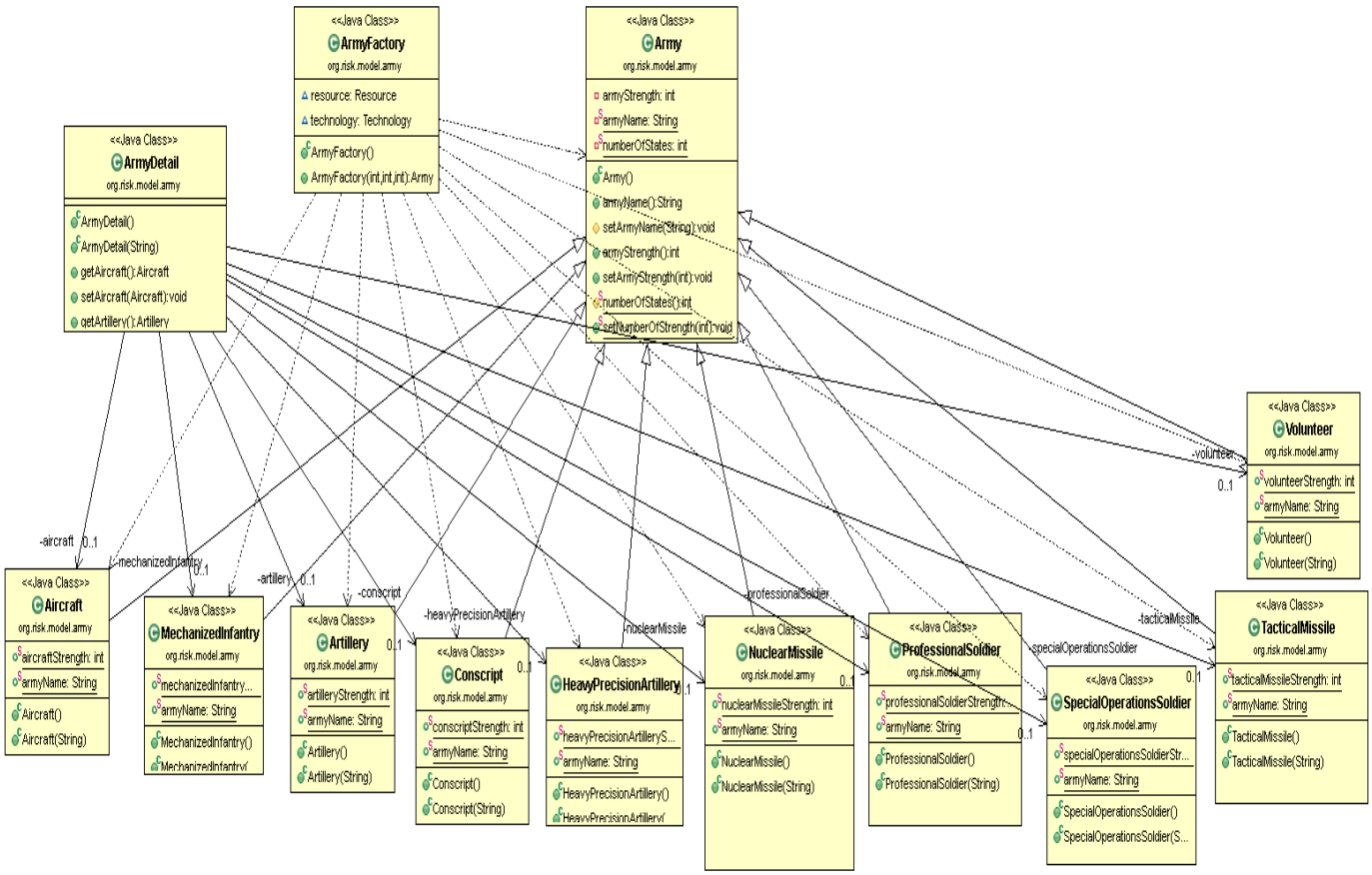


Fig 1 - UML Diagram for Factory pattern

Identified Pattern in Map.java (org.risk.model) as Singleton Pattern

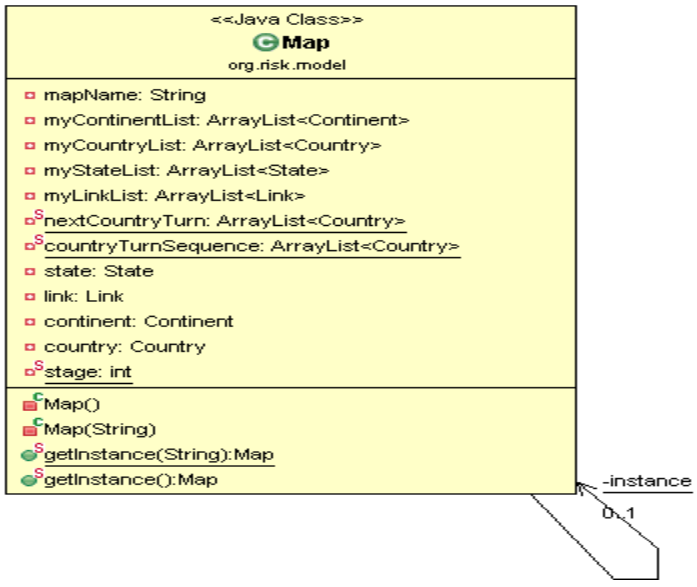
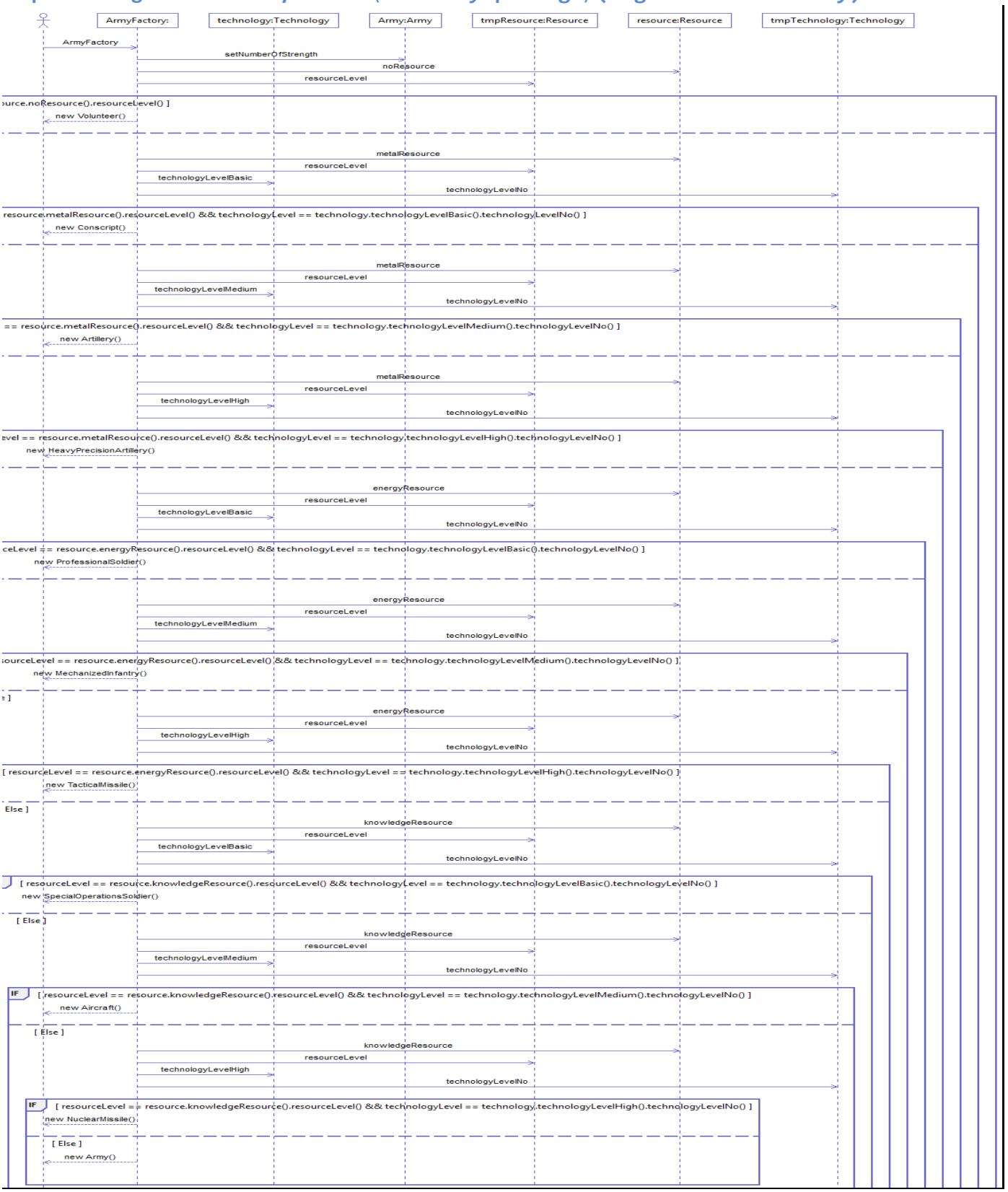


Fig 2 - UML Diagram for Singleton pattern

Tool Used to genetate UML Diagrams, See http://www.objectaid.com/download

Sequence Diagram for Factory Pattern (For Army package) (org.risk.model.army)



Tool Used to genetate Sequence Diagram, See http://www.objectaid.com/download

From Army package (ArmyFactory.java)

```
package org.risk.model.army;
import org.risk.model.Resource;
import org.risk.model.Technology;
public class ArmyFactory {
    // The class is responsible for which type of the army to call.
       Resource resource = new Resource();
       Technology technology = new Technology();
public Army ArmyFactory(int resourceLevel, int technologyLevel,
                  int numberOfStates) {
             Army.setNumberOfStrength(numberOfStates);
             // If a country has no Resource it will produce Volunteer as Army
             if (resourceLevel == resource.noResource().resourceLevel()) {
                  return new Volunteer();
             }
             // if a country has metal as a resource and Basic technology, it will
             // produce Conscript as Army
             else if (resourceLevel == resource.metalResource().resourceLevel()
                        && technologyLevel == technology.technologyLevelBasic()
                                    .technologyLevelNo()) {
                  return new Conscript();}
.................
}
```

From Army package (Army.java)

```
package org.risk.model.army;
import javax.xml.bind.annotation.XmlAccessType;
public class Army {
       private int armyStrength;
       private static String armyName;
       private static int numberOfStates;
public String armyName(){
             return armyName;
       } ..............
}
```

From Army package (ArmyDetail.java)

```
package org.risk.model.army;
import java.util.Enumeration;
public class ArmyDetail {
       private Aircraft aircraft;
       private Artillery artillery;
        ……..
public ArmyDetail(String name) {
             aircraft = new Aircraft(name);
             artillery = new Artillery(name); ….}
public Aircraft getAircraft() {
             return this.aircraft;
       }
private int getNuclearMissileCount(ArmyDetail armyDetail) {
             return (armyDetail.getNuclearMissile().armyStrength() /
NuclearMissile.nuclearMissileStrength);
       }…….
.…}
```

Identified Patterns:
Factory pattern is used in the org.risk.model.army package:

In an Army Package, I found Factory Pattern. From Figure1 and Figure 3, we can easily identify this pattern. Figure1 explains the UML diagram of an Army Package. Figure 3 explains sequence diagram of an Army Package. In an Army Package, ArmyFactory class focus the Army class into other classes like Aircraft.java, MechanizedInfantry.java, Artillery.java, Conscript.java, HeavyPrecisionArtillery.java, NuclearMissile.java, ProfessionalSolider.java, SpecialOperationsSoldier, TacticalMissile.java and Volunteer.java classes. All these classes have been inherited from Army.java class. Theses subclasses decide which class to instantiate. That is polymorphism extending its functionalities. While using Factory Pattern, dynamical initialization performs it. So, by creating one or more objects, it all shares the same interface and behaviours.

The source code above shows how the factory method depends on two attributes, technology and resources and based on that it chooses which army type to initialize

Definition is taken, See: http://sourcemaking.com/design_patterns

Identified Pattern in Map.java(org.risk.model) as Singleton pattern

From Figure 2, it creates a single instance of the map. In Map.java class, from the whole operation, it modifies the same map object.

Definition is taken, See: http://sourcemaking.com/design_patterns

Some other identified patterns are:

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

See: http://sourcemaking.com/design_patterns

M4_Risk Game: Individual Part

Rajwinder Kaur: 6282490

Our Risk Game project is designed by following Model View Controller (MVC) architectural design. Plugins like Jeodrant and ObjectAid has been used in order to perform reverse engineering of the code for better code comprehensibility. The project has implementation of many patterns like Singleton Pattern, Factory Pattern & Observer Pattern etc.

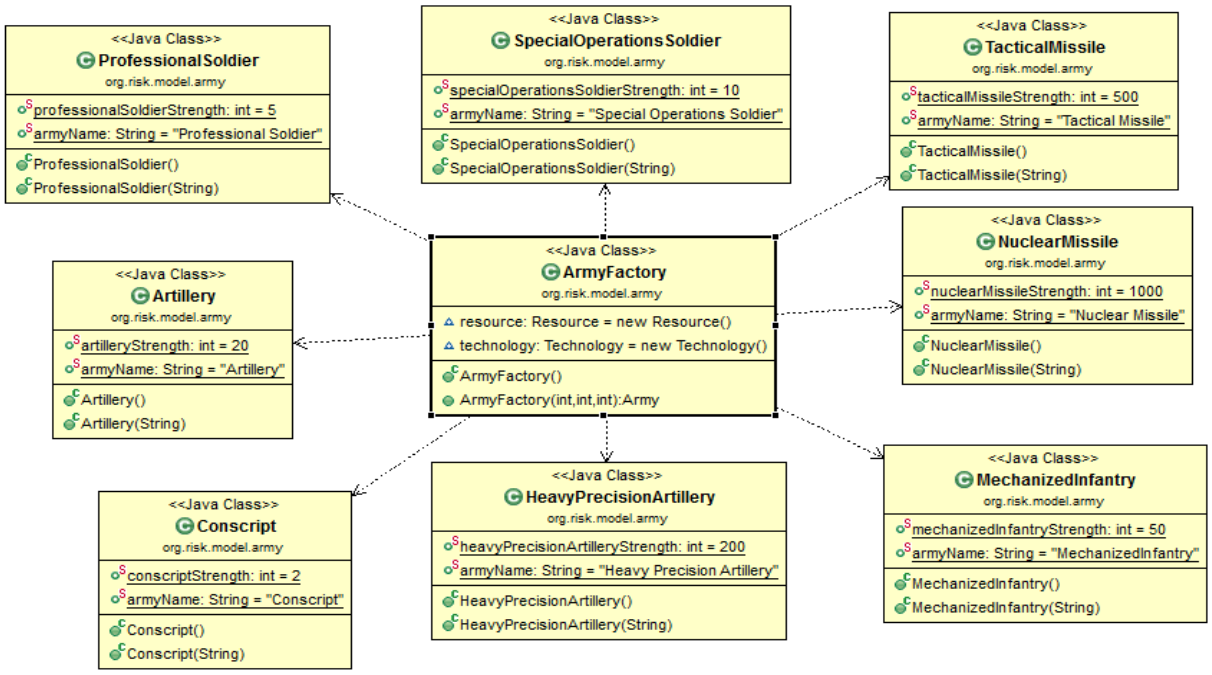
Brief description of one of the pattern named as Factory Pattern is as follows:

Factory Pattern

According to definition, Factory pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses. In general, it allows us to create one or more context specific objects that all share the same Interface and basic functionalities.

Similarly, in our Risk Game project, package org.risk.model.army has the implementation of Factory pattern. The ArmyFactory class is created and specializes the army into different types of other classes like Conscript.java, ProfessionalSoldiers.java etc., each with their own behavior & properties. All these army classes are inherited from the main base class army i.e we have an inheritance hierarchy that exercises polymorphism extending its functionalities. That's why a static factory method in the base class was added hence implementing the Factory Pattern.

Class diagram is shown below that implements factory pattern:



These classes are inherited from the ArmyFactory.java, so whenever any army class is called, as per the conditions defined, instead of directly accessing the desired class, methods of the base class will be called. This refactors the code and makes it easier to differentiate between the called classes and provides the output accordingly.

The code below shows how the factory method depends on two attributes, technology and resources and based on that it chooses which army type to initialize

```
public class ArmyFactory {


      ......


      public Army ArmyFactory(int resourceLevel, int technologyLevel,int
numberOfStates) {
```

```
                if (resourceLevel ==resource.noResource().resourceLevel()) {
                return new Volunteer();
            }

            else if (resourceLevel == resource.metalResource().resourceLevel()
                        && technologyLevel == technology.technologyLevelBasic()
                                    .technologyLevelNo()) {
                return new Conscript();
            }



            else if (resourceLevel == resource.energyResource().resourceLevel()
                        && technologyLevel == technology.technologyLevelBasic()
                                    .technologyLevelNo()) {
                return new ProfessionalSoldier();
            }

        .......
                else {
                return new Army();
                }
        .......
}
```

Link to factory pattern:

http://sourcemaking.com/design_patterns/factory_method

http://patterns.instantinterfaces.nl/current/Refactoring-and-Design-Patterns-INTRDP-GMDV.html

M4_Risk Game: Individual Part

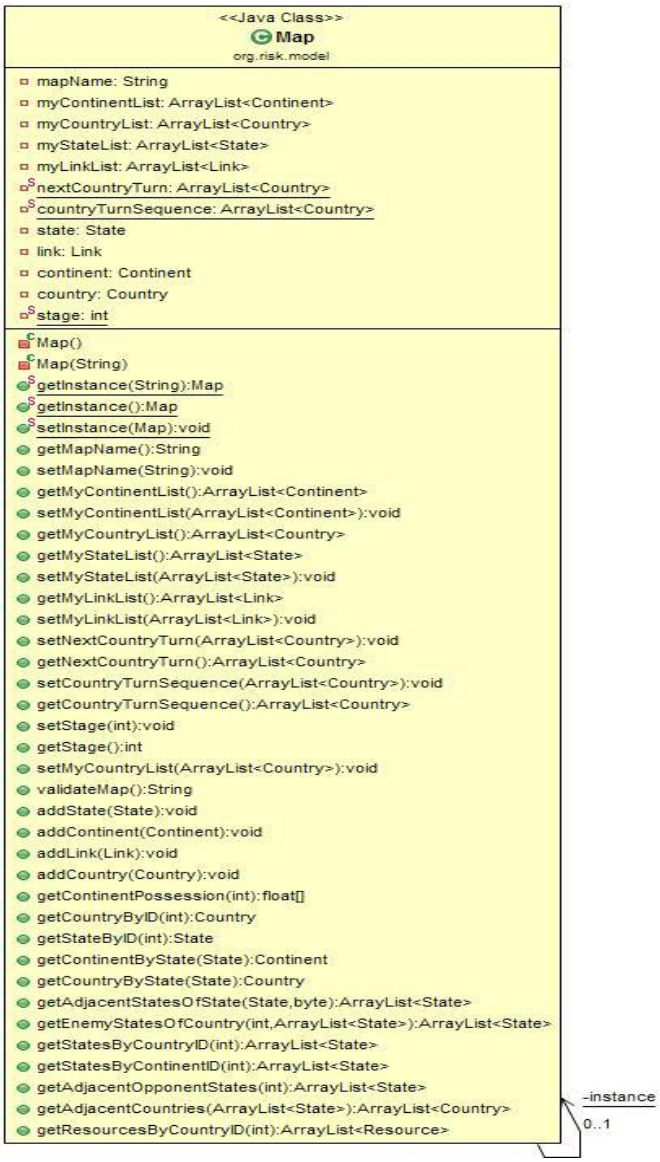
Harsahiljit Singh - 6405975

Singleton Pattern

This pattern ensures that class only has one instance and provide global point of access to it. In our project this pattern is implemented in class Map.java This class contains whole information about the map like Continents, States, and Number of links and provides this information to other classes like the class Game engine uses this class to get information about whole map and further provides this information to other classes.

This pattern is very helpful in our project as the Map class only has one instance so all classes which want to interact with Map class uses only one instance. Singleton pattern makes it very easy for other classes to interact with Map class.

Class diagram of Map.java class using java plugin “Objectaid”



I used java plugin WOP to find patterns http://www-ist.massey.ac.nz/wop/

```
public class Map extends Observable {
	private String mapName;

	private ArrayList<Continent> myContinentList;

	private ArrayList<Country> myCountryList;

	private ArrayList<State> myStateList;

	private ArrayList<Link> myLinkList;

	private static ArrayList<Country> nextCountryTurn;

	private static ArrayList<Country> countryTurnSequence;
```

```
        private State state;
        private Link link;
        private Continent continent;
        private Country country;
        private static int stage;


private static Map instance = null;

        private Map() {
                this.mapName = "";
                ........
                ........

        }

private Map(String name) {
                this.mapName = name;
                .......
                ......
        }



        public static Map getInstance(String name) {
                if (instance == null) {
                        instance = new Map(name);
                }
                return instance;
        }



      public static void setInstance(Map map) {
                instance = map;
        }
```

http://sourcemaking.com/design_patterns/singleton

M4_Risk Game Indiviual Part

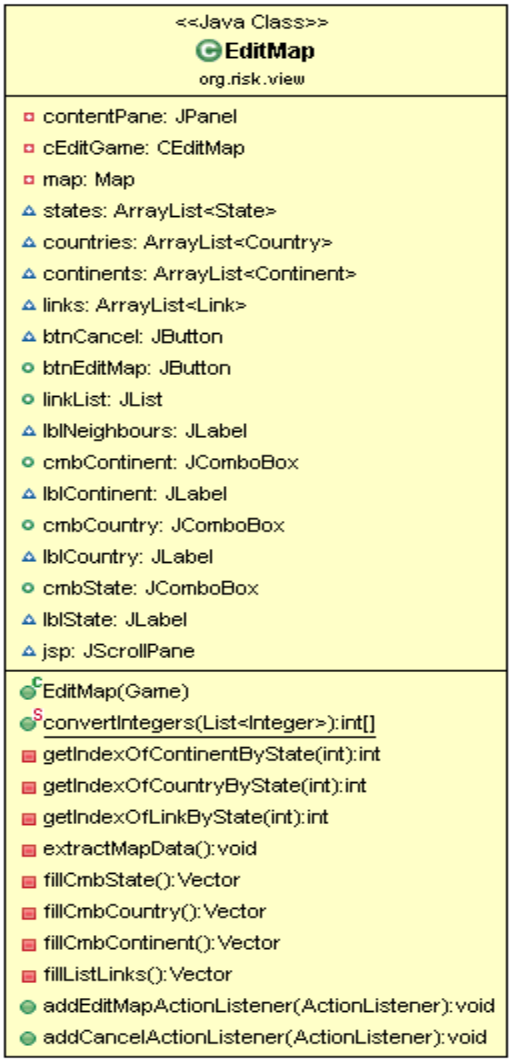
Prabhjot Kaur Sekhon 6473318

OBSERVER PATTERN:

Our project is based on MVC architecture. The View Model object is a sort of mediator between what happens on screen, between what the user does and the Model that simulates the World that forms the Game.
In the project, to understand the structure of the system, Jeodrant and Object Aid plug-ins are used. Various patterns have been used in this project, one of which is “observer pattern” used by org.risk.view.

The Observer pattern is the basis of events and event driven systems. It defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Following class, edit map, of org.risk.view uses “observer pattern”. Whenever any action is performed, all other class get notified. Classes including state, country, and continent are affected when changes in class Edit Map are done. When subject changes all the observer classes are notified to perform certain actions according to the conditions defined.

Below is the class diagram of EditMap class, showing observer pattern in action :



Below, is the working code, defining required actions according to the given condition :

```
cmbState.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
         JComboBox cbs = (JComboBox) e.getSource();

int stateID = ((Item)cbs.getItemAt(cbs.getSelectedIndex()))
                              .getId();
if (stateID >= 1) {
                    // Enable elements of the window
                    cmbContinent.setEnabled(true);
                    cmbCountry.setEnabled(true);
                    linkList.setEnabled(true);
                    btnEditMap.setEnabled(true);
        }…….
        ……
        ……
 else {
                    int[] emptyList = { -1 };
                    // Disable elements of the window
                    cmbContinent.setSelectedIndex(-1);
                    cmbCountry.setSelectedIndex(-1);
                    linkList.setSelectedIndices(emptyList);
                    cmbContinent.setEnabled(false);
                    cmbCountry.setEnabled(false);
                    linkList.setEnabled(false);
                    btnEditMap.setEnabled(false);
     }
   }
 });
```

Link to observer pattern:
http://sourcemaking.com/design_patterns/observer
http://patterns.instantinterfaces.nl/current/Refactoring-and-Design-Patterns-INTRDP-GMDV.html

Risk M4 Individual Part

YASH PALIWAL 6562566

Our project is a game developed in Java. It's name is Risk Game in which computer is the player and the game is about the battles fought between countries. The whole system is based on MVC - Model View Controller. I used plug-ins like **Jeodrant** to understand the architecture of the system and **Object-aid** to create class diagrams and **WOP** to find patterns in Eclipse.

The patterns which we discovered in our projects are :

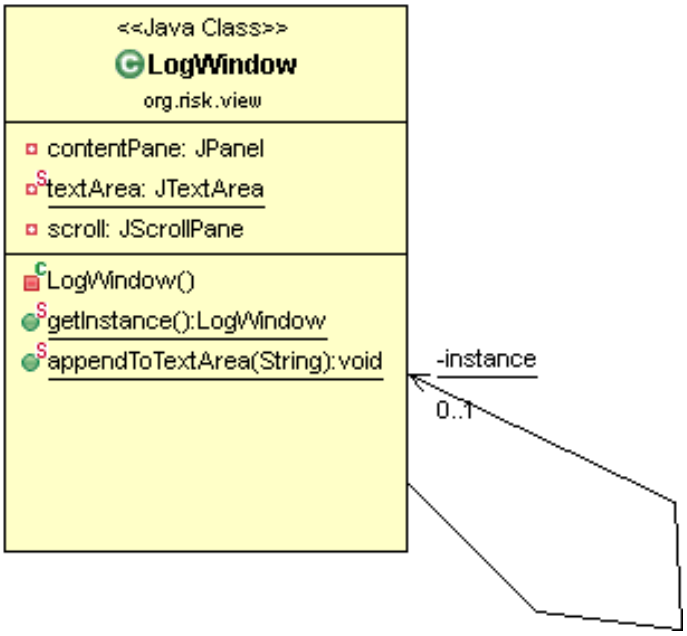
1. Factory pattern, used in the org.risk.model.army package
2. Singleton pattern, used by the Map.Java class
3. Singleton pattern, used by the LogWindow.java class
3. Observer pattern, used in the org.risk.view package

The pattern I observed is known as Singleton Pattern. I found this pattern in the LogWindow.java class in org.risk.view package. The **Singleton Pattern** is a design pattern that restricts the instantiation of a class to one object. This pattern is useful when exactly one object is needed to coordinate actions across the system. The term comes from the mathematical concept of singleton.

LogWindow.java class has the GUI (Graphic User Interface) of the log window i.e. this class is responsible for the display of the log window which is one of the important part of our game. Singleton pattern makes it easier for other classes to interact with this class by making a single instance rather than more.

Link for singleton pattern: http://sourcemaking.com/design_patterns/singleton

Class Diagram of Logwidow.java class by plug-in **Objectaid**



Link For Java plug-in ObjectAid: http://www.objectaid.com/

The source code of the loginwindow.java class is shown below and we can see that all the necessary methods like static logWindow instance, private constructor are used which are all parts of this pattern.

```
public class LogWindow extends JFrame {

     private JPanel contentPane;
     private static JTextArea textArea;
     private JScrollPane scroll;
     private static LogWindow instance = null;

     ....

     ....

/**
      *
      *
      * @return : Instance of LogWindow Class
      */
     public static LogWindow getInstance() {
          if (instance == null) {
               instance = new LogWindow();
          }
          return instance;
     }

     /**
      * Append content to Log Window
      *
      * @param msg
      *             : Log message
      */
```

Link for Java plug-in **WOP:** http://www-ist.massey.ac.nz/wop/

M4 - Implement a Refactoring - Risk Game

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

Patch set 1

- **Patchset 1 (0/3)**
  In this patch set we will pull up a method named addArmyDetails from class country and state and move it to class army, according to the reverse engineering tool (jeodrant) we found out that both classes have the same method doing the same job ,the code smells we will be eliminating are feature envy and duplicate code.
- **Patchset 1 (1/3)**
  We remove the method addArmyDetails from the country class to army class, and ran our test cases, the project was compiling properly and all the test cases passed.
- **Patchset 1 (2/3)**
  We remove the method addArmyDetails from the state class to army class, and ran the tests again and they all worked, the behaviour did not change
- **Patchset 1 (3/3)**
  As the final step we moved (method pull up) the method addArmyDetails from state and country class both and placed it in army class shown here, and ran the test cases again, they all passed showing successful method refactoring.

Patch set 2

- **Patch set 2 (0/3)**
  In this patch set we are going to extract a class named Player from class country, and after extracting the class we will move methods related to player turns into player class to divide responsibilities, country class will keep data about country and army details and player will handle player turns now
- **Patch set 2 (1/3)**
  We extracted a class named player and added turn methods to it, the diff file shows the changes done
- **Patch set 2 (2/2)**
  We moved the method player turn from gameEngine to player class