Advanced Programming Practices Project - Build 3 War Zone Game Implementation

Team

Tejaswini - 40186127

Vignesh - 40171544

Vikram - 40126852

William - 40186129

Manimaran - 40167543

Contents:

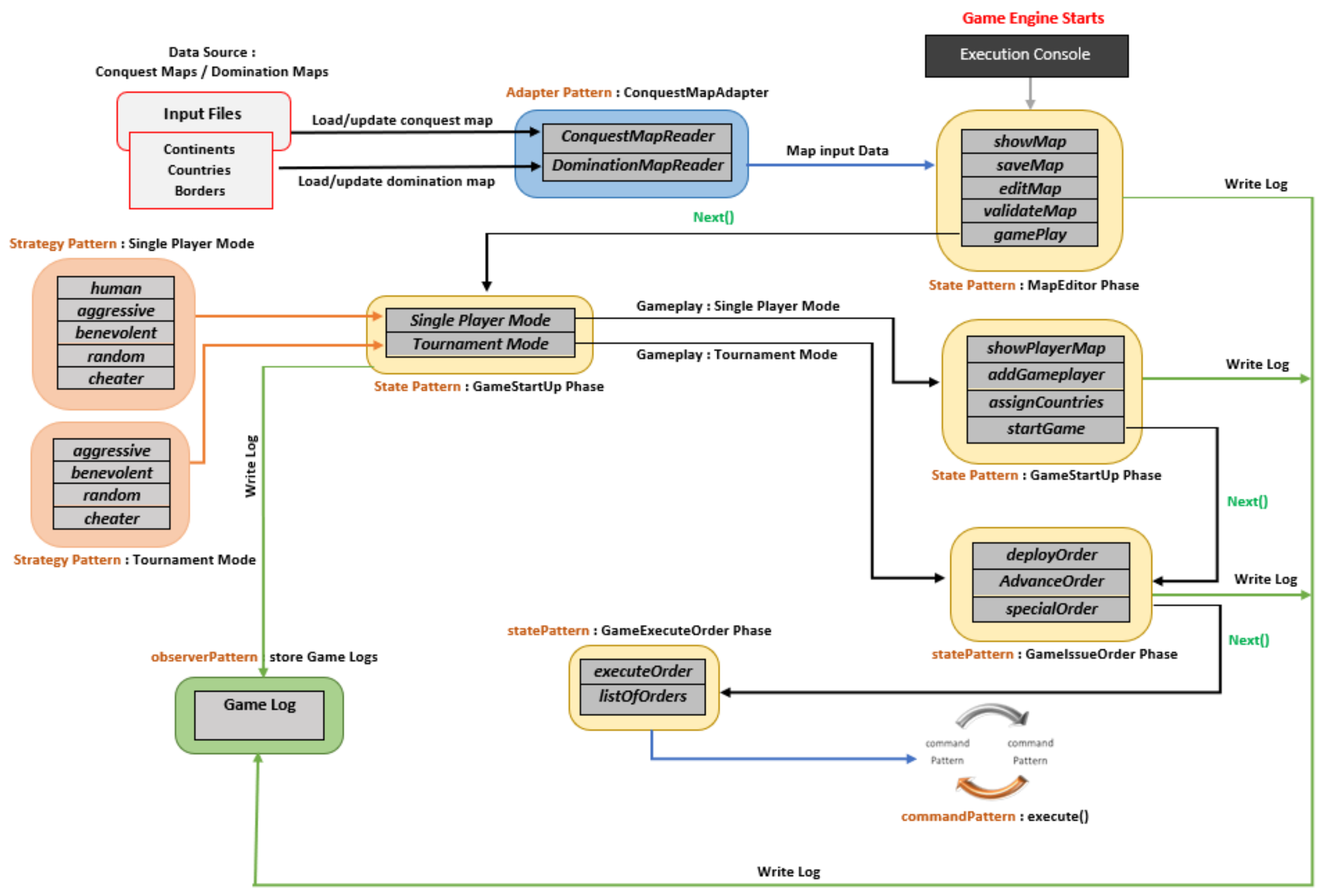
Overview

The purpose of this document is to present a detailed description of War zone game. It will explain the purpose and features along with the interfaces of the gaming system, what the game will do, the constraints under which it must operate and how the system will react to external stimuli.

Project description

War zone game allows the user to select the continent. Once the continent is selected by the user, then the countries belongs to the map will be assigned randomly to each of the player respectively. Initially same number of armies are issued to all the players. Player can deploy the armies in same country to increase the own country's army count. Thereafter, during the execution phase he can transfer the armies to his own countries to increase the reinforcement of the country or he can deploy the armies on enemies to attach and conquer the enemy country. Once the player wins in battle then the country will belong to his own and he can add his own armies in the conquered country.

Architectural Design:



Input for our entire game will be a map, comprised of several Continents and the corresponding Countries. Here a map is represented with the help of text files.

- continent text file containing information about continent id and continent name.
- Country text file containing information about country id, country name and continent id.
- Borders text file containing information about the country id and the corresponding borders of the country.

The entry point will be the main engine class and where user is allowed to choose the game play operation of his choice. Based on the user choice we either load the map or load the game.

Load map:

We compose a map, by reading the data from input files (Continent file, Country file, Border file) with the help of BeanReader class.

Load game:

If the player chooses the load game option, then game will be loaded.

<u>Implementation</u>

1. ComposeGraphImpl:

Compose Graph Impl class contains two functionalities for creating continent map and country map.

We use graph class as a utility to create continent map and country map. Graph class has a functionality to add and remove the nodes and edges. Here node represents the countries and edges represents the connection between the countries. Once the graph is composed if the user's choice is show map, then this generated map info will be shown.

Once the main engine loads the game for game play, assignReinforcement, issueOrder, executeOrder class will execute to assign the armies to the player. Player instances will show the current countries owned by the player and also armies correspond to the country.

2. EditCommandsImpl:

Edit Command Impl class is used to make changes and apply functionalities in the map files, the map files consists of Continents, Countries, Borders.

The different map files are stored under different packages for the varying purposes, It bolster's the gameplay based on the user interaction and the different patterns and strategies used. It gets the command from the user and does operations mainly like editing the continents, it's respective countries and it's borders.

After getting the command from the user, executing it's operations, it transmits the values to the design patterns, where the differences between the maps are analyzed and processed further.

<u>**The main methods involved in this class are:**</u>

- **writeContinentFile(),**
- **writeCountryFile(),**
- **writeBordersFile(),**
- **editContinentMap(),**
- **editCountryMap(),**
- **editNeighbourMap().**

It also notifies the user if invalid command is given and the process is repeated until get the valid command for editing the maps.

3. InputProcessor:

The Input Processor class is used to process the values from the command line interrupt.

It is used to evaluate the commands given during the gameplay startup phase and continues to work while executing and issue order phases too.

The main objective of this class is to split the commands and transfers the required values to the next phase on the gameplay based on the **split-command" ".**

For eg, if user wants to add a country to the map and gives the command add countryid continentid, it splits the values between the spaces and transfers the output value to the next functionalities.

<u>The main methods involved in this class are:</u>

- **getAddContinentInput(),**
- **getAddCountryInput(),**
- **getString(),**
- **getStartUpPhase().**

4. MapReader:

The whole game logic is based on the values we going to fetch from the maps, those maps should be read, implemented, saved, executed and edited changes should be made.

But initially to start the game logic we need to read the values from the maps and use it in the game logic. For that purpose, we are using this class.

The map values are determined based on the values we get from the user like what type of map, and what war-zone map should be needed on that particular time. So, the functionalities written in this class are used to execute it's operations based on the values it fetches.

It's functionalities does return any bogus values, since it is totally dependable on user commands.

<u>The main methods involved in this class are:</u>

- **readContinentFile() - reads the continent files.**
- **readCountryMap() - reads the corresponding countries under it's continents.**
- **mapCountryBorderReader() - reads the border nations among the countries.**

5. ValidateMapImpl:

After reading the map values from the MapReader class, we should ensure that the whole values under the corresponding maps are equal or not. To check that the ValidateMapImpl class is used.

The validity of the maps are checked through whether the corresponding values of Continents, it's Countries and it's respective Borders are correct are not.

First we validate the maps based on the continents id's, continent's name, country id's and at last the whole maps are validated based on the inputs.

The main methods involved in this class are:

- **validateContinentId(),**
- **validateContinentName(),**
- **validateCountryId(),**
- **validateFullMap().**

6. Graph class:

This Graph class is used to create the graphs between the nodes and it's edges.

It is used to access the map values and attain it's total functionailes by using the graph concept.

The main methods involved in this class are:

- **addVertex(),**
- **removeVertex(),**
- **addEdge(),**
- **removeEdge().**

7. MainEngineClass:

The initial game will start under this class only.

It is totally responsible for even executing the map functionalities.

The sole responsibility of this class is to initiate the **start** method under **GameEngine** class.

8. GameEngine class:

<u>This class is used to select the options from the user like:</u>

- **Show Map,**
- **Save Map,**
- **Edit Map,**
- **Validate Map,**
- **Game Play.**

It executes based on the user input.

The main function used in this class is **Start()** method.

It also has responsibility for **assigning reinforcements, issuing orders and executing orders.**

9. GamePlayer Class:

This class is used to contain the parameters for the players who are involved in the gameplay.

Since it is a multi player game, the values or attributes should be assigned to the number of players who plays.

The attributes includes **player name, territory and countries the player owns and the no of armies the player contains.**

<u>The main methods involved in this class are:</u>

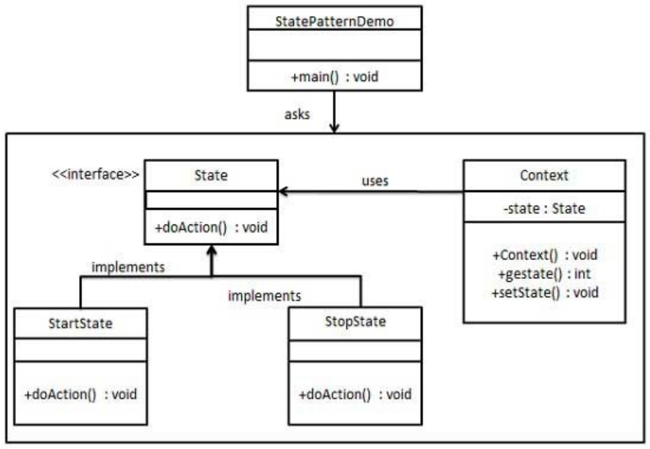
- **setStrategy(),**
- **issueOrders(),**
- **nextOrder(),**
- **pushBackOrder(),**
- **setSpecialCardValues().**

Refactoring:

Refactoring consists of improving the **internal structure** of an existing program's source code, while **preserving** its **external behavior**.

• State Pattern:

The state pattern in Java is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes.



Different phases implemented in gameplay in state pattern:

1. Startup phase.
2. Issue orders phase.
3. Order execution phase.

Gameplay implementation using State pattern:

• Phase:

- ✓ It is an abstract class which acts as a main phase or state in the state pattern.
- ✓ This abstract class acts as a parent class to extend another two-abstract class.
  1. Gameplay.
  2. Map editor.

- ✓ It contains functionalities to different phases on startup, issue and order execution.

<u>Game play:</u>

It is an abstract class extends the phase of the state pattern. The functions used in this class used to implement the showman functionality of the show game map.

The decision from the strategy pattern either single mode or tournament is used to determined the start up phase of the game play in State pattern.

Some of the functionalities include:

- showPlayerMap,
- addGamewPlayer,
- assignCountries,
- startGame.

<u>The class extends from this class are:</u>

1. Gameplay issue order.
2. Gameplay order execution.
3. Gameplay startup.

- **<u>GamePlayStartUp:</u>**

It is a class extends the game play abstract class. It has the functionalities like **loadmap**() and army reinforcements.

This class is used to convert the map file into continents, countries and its borders and assigning values is done.

It acts as one of the states or phase functionalities under gameplay phase.

- **<u>GameplayIssueOrder:</u>**

It is also a class extends the game play abstract class.

The main functionalities of this class are:

1. issueOrders(),
2. deployOrder(),
3. advanceOrder().

1. **deployOrder():**

This functionality is used to deploy the no of armies in the gameplay.

2. **advanceOrder():**

This functionality is used to issue the advanced orders like bomb order command, blockade order command, airlift order command, diplomacy order command.

- **GameplayOrderExecution:**

It is also a class extends the game play abstract class.

This class main functionality is to execute the orders which are defined in issue ordering class.

The function used in this class is **executeOrders().**

- **MapEditor:**

It is an abstract class extends the phase of the state pattern. The functions used in this class used to implement the various map editing functionalities and savemap.

The class extends from this class is MasterMapEditor class.

- **MasterMapEditor:**

It extends from MapEditor abstract class.

The main functionalities of this class are:

1. showMap().
2. loadMap().
3. saveMap().
4. validateMap().
5. editMap().

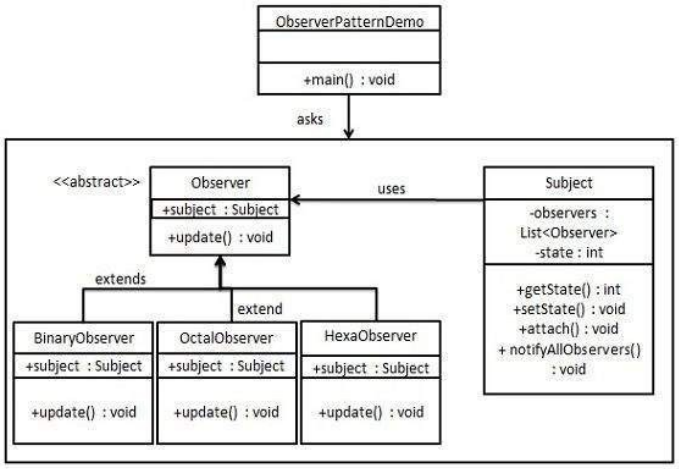
- **<u>Observer Pattern:</u>**

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

<u>The classes used in this pattern are:</u>

1. LogEntryBuffer.
2. LogWriter.
3. Observable.
4. Observer.

The observer Pattern is used to track the commands and store it in the log buffer class and writes it in the log file.



- **<u>Observable:</u>**

It is a class used to add, remove and notify the observers and it is considered as a subject, in which observer objects are notified with one to many dependencies.

- **<u>Observer</u>**:

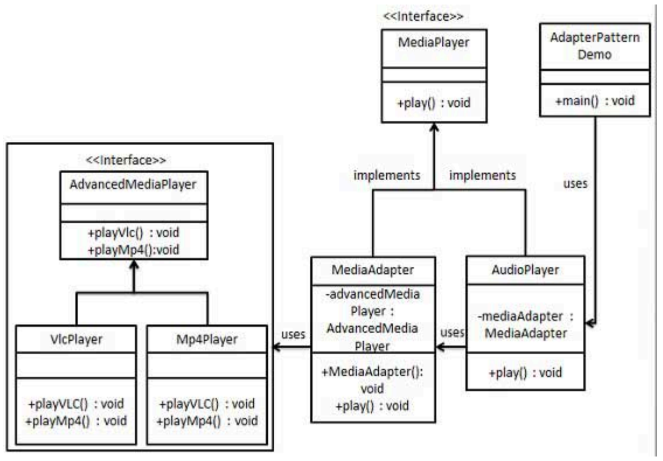
It is an interface containing update(), which used to update the observers implements from it.

- **<u>LogEntryBuffer:</u>**

It is a class extends the observable class and implements serializable interface.

The timestamp and the various commands are monitored here.

The main functionalities of this class are:



1. returnEntry().
2. toGameString().
3. toMapEditorString().

• **LogWriter**:

It is a class implements Observer interface.

It acts as a observer for the observable or subject class.

It is used to write and display the command history when needed.

• **Adapter Pattern:**

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

The classes involved in this pattern are:

✓ ConquestMapAdapter,

✓ ConquestMapReader,

✓ DominationMapReader.

<u>The two main functionalities involved in this pattern classes are:</u>

1. loadMap() - This functionality is used to load the input values of the map which is in text format and identifies the continents, their countries and the borders of the countries.
2. saveMap() - This functionality is used to save the extracted, changed or newly added values and it's operations as the map value in respective text files.

<u>Inputs</u>:

- The inputs for this pattern(total game) are map values.
- The maps involved are conquest map and domination map.
- Both the maps consist of continents and their respective countries values.
- It is considered as initial phase since the usage of which maps are determined here.

The main objective of the adapter pattern is to differentiate the usage of which map values and directs them into map editor phase and further game play phase in state pattern.

- **<u>ConquestMapAdapter:</u>**

This is the class used as a bridge between ConquestMapReader and DominationMapReader classes or used to select on which class is used.

It is a Child class extends the DominationMapReader class and the instance of the ConquestMapReader is created and that object is used to call the functions present in there.

The functionalities involved are loadMap() and saveMap() and these are presented in its respective super or parent class and instance involved class.

- **<u>ConquestMapReader:</u>**

The instance of this class is created in ConquestMapAdapter class.

Initially it is used to deal with the Africa map values and the main functions are loadMap() and saveMap().

The main condition to activate this class is that the typeOfMAp should be "conquest".

- **DominationMapReader:**

It is a parent class of ConquestMapAdapter class.

Initially it is used to deal with the Canada map values and the main functions are loadMap() and saveMap().

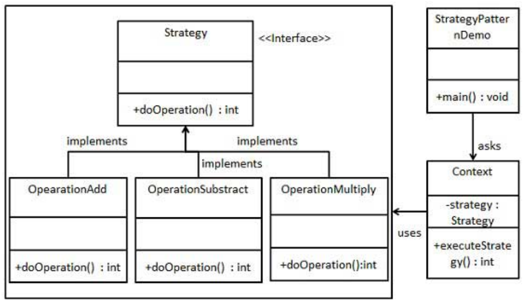
The main condition to activate this class is that the typeOfMAp should be “domination”.

- **Strategy Pattern:**

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object.

The strategy object changes the executing algorithm of the context object.



The input values for this strategy pattern is directed from map editor phase of state pattern, where different map functionalities are involved.

The result of this Design Pattern is used to set up the strategy of the player under any of the available modes and executed in game start up and execution phases in state pattern.

<u>The modes involved in the strategy pattern are:</u>

- ✓ **Single Player mode,**

The Strategies involved in the single player mode are:

- Human,
- Aggressive,
- Benevolent,
- Random,
- Cheater.

- ✓ **Tournament mode,**

The Strategies involved in the tournament mode are:

- Aggressive,
- Benevolent,
- Random,
- Cheater.

<u>The main classes involved are:</u>

- ✓ PlayerStrategy(abstract class),
- ✓ AggresiveBehavior class,
- ✓ BenevolentBehavior class,
- ✓ RandomBehavior class,
- ✓ CheaterBehavior class,
- ✓ HumanBehavior class.

The abstract class "<u>PlayerStrategy</u> <u>class</u>" is used to work on the strategies and to determine the strategies based on it's orders and the orders are listed in Order class.

Single Player mode directs to Game Start up phase in state pattern and Tournament mode directs to Game Execution phase.

The main Strategies involved during run time are:

1. Aggressive strategy,
2. Benevolent strategy,
3. Human strategy,
4. Random strategy,
5. Cheater strategy.

- **PlayerStrategy:**

It is an abstract class, which other strategy classes are extended from it.

It creates the object for the GamePlayer class for the players involved.

It also creates an abstract function **order()** for the Order class and this function is used in different strategy classes.

- **AggressiveBehavior:**

It is an extended class of PlayerStrategy abstract class.

The order() functionality is used here and the list of of deploy and advance army functionalities are used and returned to the startup phase of the game setup.

The main strategy of this class is to attack the other country with it's owned country with **maximum forces.**

- **BenevolentBehavior:**

It is an extended class of **PlayerStrategy** abstract class.

The order() functionality is used here and the GamePlayer object is returned along with the null value from the order function.

The main strategy of this class is to **defend** it’s weak country with deploying armies on it and never attacks.

- **HumanStrategy:**

It is an extended class of PlayerStrategy abstract class.

It is not a computer player class and object for GamePlayer class is returned with the orders from the human inputs.

The main strategy is to work based on the **Human interactions**.

- **RandomBehavior:**

It is an extended class of PlayerStrategy abstract class, and returns the object for the class instantiated in the parent class and returns the value for the abstract class using super().

It is a computer player class.

The main strategy of this classs is to **randomly deploy** the armies on it’s owned countries and **attack randomly** it’s neighboring countries.

- **CheaterBehavior:**

It is a child class of PlayerStrategy class and the values are returned values for it’s order function used as abstract there.

The strategy of this class is implemented while issuing **issueOrder()** method.

The main strategy of this class is to **conquer** the immediate neighbors, **double the armies** of it’ owned countries which shares borders with the enemy nations.

It does not create any orders but will affect affect the map while **order creating** phase.

Game Modes:

- The different modes are implemented in the gameplay on using the **design patterns** .
- The input values are fetched from the different available **maps** saved in the text format.
- Initial game play play is same for both the modes until fetching the maps, but the mode usage comes into play while setting the **behavior** of the game player.
- Every strategy with different graph or maps has been implemented in the tournament mode.

✓ Single mode:

The Strategies involved in the single player mode are:

- Human,
- Aggressive,
- Benevolent,
- Random,
- Cheater.

The Single mode involves both the strategies from the computer play and the user interaction as well.

✓ Tournament mode:

The Strategies involved in the tournament mode are:

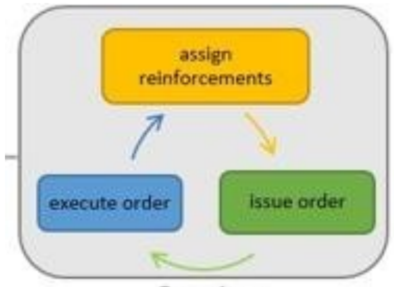
- Aggressive,
- Benevolent,
- Random,
- Cheater.

The Tournament mode involves the strategies only from the computer play.

Game Play Design:

✓ **Game main loop:**

Thecyclicprocessof**Assignreinforcement→Issueorder→Executeorder** will happen.



Game Loop

- **Assign reinforcement:**
  - ➢ It will assign armies to each player initially (5 players).
  - ➢ Further during the game play, based on the war zone rules the armies for reinforcement will be handed over to each player
  - ➢ This operation continues until all the player's area assigned with their reinforcement armies.
- **Issue order:**
  - ➢ Based on user's preference we will deploy the armies and reinforce the designated country.
  - ➢ This operation continues for each player until the armies in the player's hands gets null(zero).
  - ➢ The above operation will happen for all the players in the game.
  - ➢ The issue of army reinforcement will happen in round robin method.
- **Execute order:**
  - ➢ In execute order, the attack will happen based on the player's preference.
  - ➢ Once the attack is finished, the country will be added to the player who won in attack.
  - ➢ If the player loses the attack, then his country will be taken by enemy count.
  - ➢ This execute order will happen until the major countries are getting acquired by a single player.

Software Stack:

For this implementation we used the following stacks

- Java as a programming language
- GIT as a repository
- Eclipse as an IDE
- Project is incorporated in Gradle project