Team 13 – Refactoring Document Build 3

COURSE: SOEN 6441 APP
INSTRUCTOR: Prof. JOEY PAQUET

Team members

- Jayati Thakkar 40230506
- Sushant Sinha 40261753
- Raj Kumar Ramesh 40225218
- Bhoomiben Bhatt 40291067
- Rupal Kapoor 40274905
- Rikin Chauhan 40269431

Potential Refactoring Targets:

Identify Targets:

Based on the challenges and inconsistencies encountered during the development of build 2, along with insights gathered throughout the development process of build 3, a list of refactoring objectives has been formulated. These objectives are aimed at addressing the noted issues and enhancing the overall code quality and functionality.

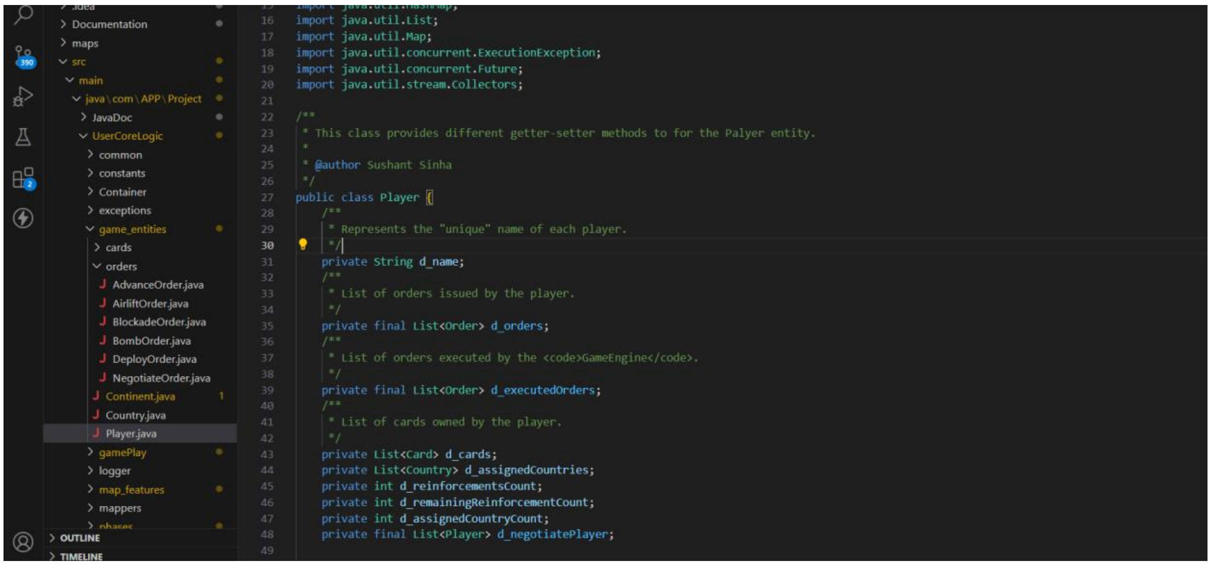
1) Utilizing the Observer pattern to streamline console log management.
2) Revamping the Adapter pattern to facilitate the loading and saving of both Domination and Conquest map formats.
3) Enhancing player behavior strategies through improvements to the Strategy pattern.
4) Improving how information is displayed on the console for better clarity.
5) Upgrading error management by optimizing logger functionalities.
6) Organizing and clearly separating the observer functionality into the view directory for better modularity.
7) Revitalizing the game to support both individual play and tournament styles.
8) Ensuring the Command pattern is correctly implemented and functional.
9) Conducting a thorough code review and modification to meet established coding standards.
10) Refactor saveMap() method and issueOrder() method
11) Revise terminal such a way that messages for commands are being reformatted to enhance user-friendliness.
12) All unnecessary print statements which were used for testing purpose.
13) Enhance player strategies and behaviors using improvements in the Strategy pattern for dynamic gameplay.
14) Remove unnecessary/ irrelevant comments to improve cleanliness, with relevant details moved to the respective function's Javadoc documentation.
15) Replace System.exit() with a return statement.

Actual Refactoring Targets:

1) **Strategy pattern:** The Strategy Pattern is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. It encapsulates different algorithms or strategies in separate classes and allows a client to choose the appropriate algorithm dynamically. The pattern promotes loose coupling by separating the selection of an algorithm from the algorithm's implementation.

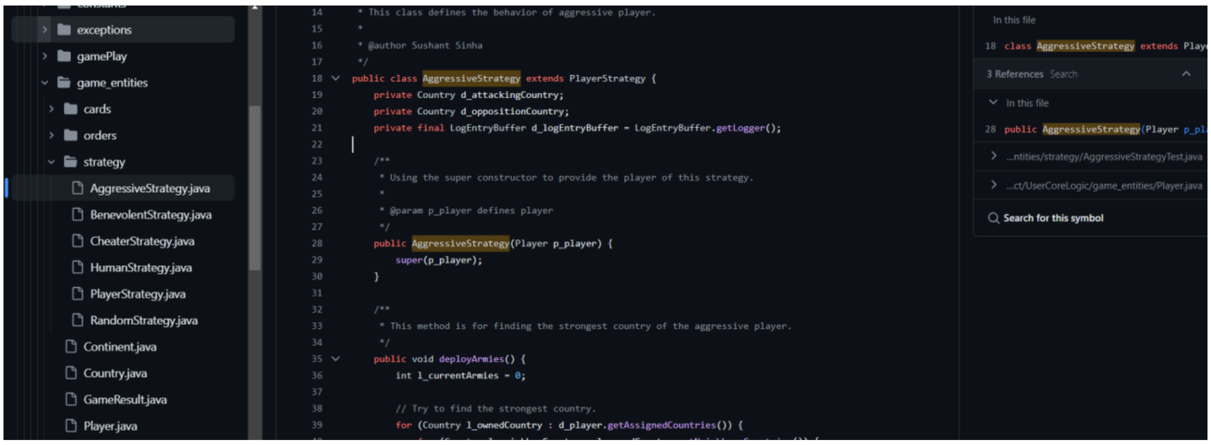
   Refactoring the Player class's issueOrder() method to adopt the Strategy pattern: In line with the requirements for build 3, the issueOrder() function in the Player class has been updated to employ unique strategies for each player type.

Before



```java
import java.util.List;
import java.util.Map;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;
import java.util.stream.Collectors;

/**
 * This class provides different getter-setter methods to for the Palyer entity.
 *
 * @author Sushant Sinha
 */
public class Player {
    /**
     * Represents the "unique" name of each player.
     */
    private String d_name;
    /**
     * List of orders issued by the player.
     */
    private final List<Order> d_orders;
    /**
     * List of orders executed by the <code>GameEngine</code>.
     */
    private final List<Order> d_executedOrders;
    /**
     * List of cards owned by the player.
     */
    private List<Card> d_cards;
    private List<Country> d_assignedCountries;
    private int d_reinforcementsCount;
    private int d_remainingReinforcementCount;
    private int d_assignedCountryCount;
    private final List<Player> d_negotiatePlayer;

```

After

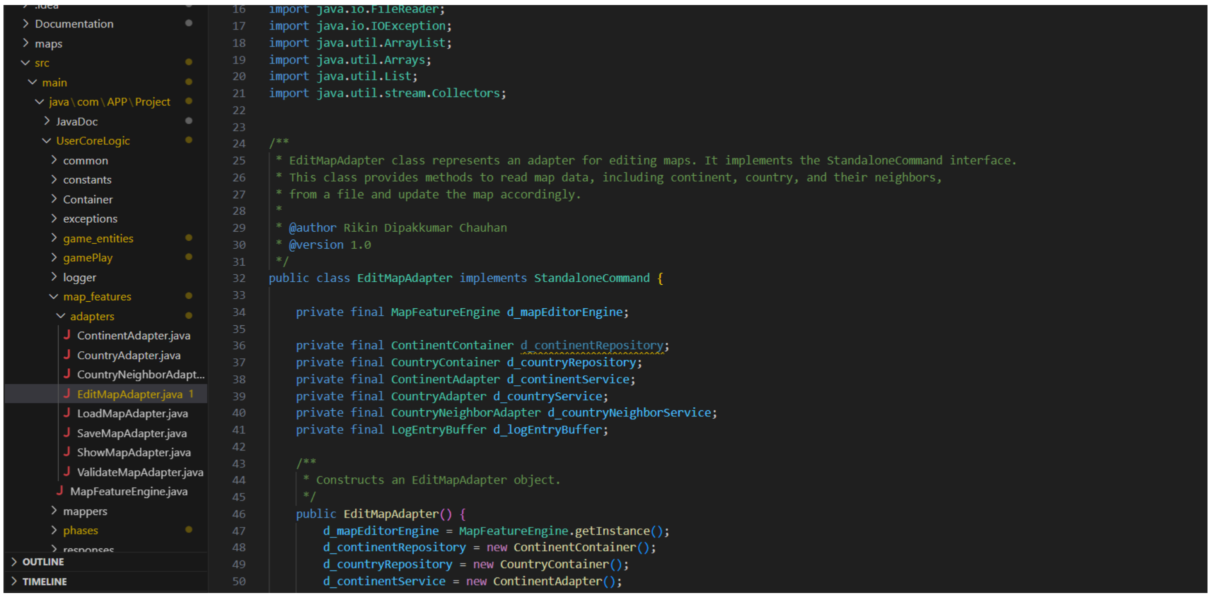


```java
 * This class defines the behavior of aggressive player.
 *
 * @author Sushant Sinha
 */
public class AggressiveStrategy extends PlayerStrategy {
    private Country d_attackingCountry;
    private Country d_oppositionCountry;
    private final LogEntryBuffer d_logEntryBuffer = LogEntryBuffer.getLogger();

    /**
     * Using the super constructor to provide the player of this strategy.
     *
     * @param p_player defines player
     */
    public AggressiveStrategy(Player p_player) {
        super(p_player);
    }

    /**
     * This method is for finding the strongest country of the aggressive player.
     */
    public void deployArmies() {
        int l_currentArmies = 0;

        // Try to find the strongest country.
        for (Country l_ownedCountry : d_player.getAssignedCountries()) {
```

```java
18 class AggressiveStrategy extends Playe
28 public AggressiveStrategy(Player p_pl
```

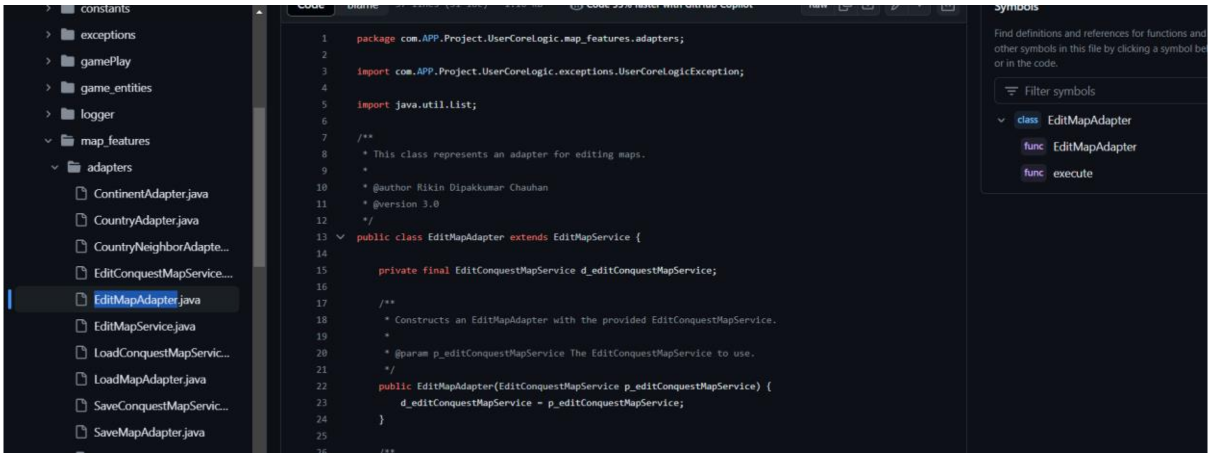
2) **Adapter Pattern:** The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two different interfaces, converting the interface of a class into another interface that clients expect.

When pattern not implemented:



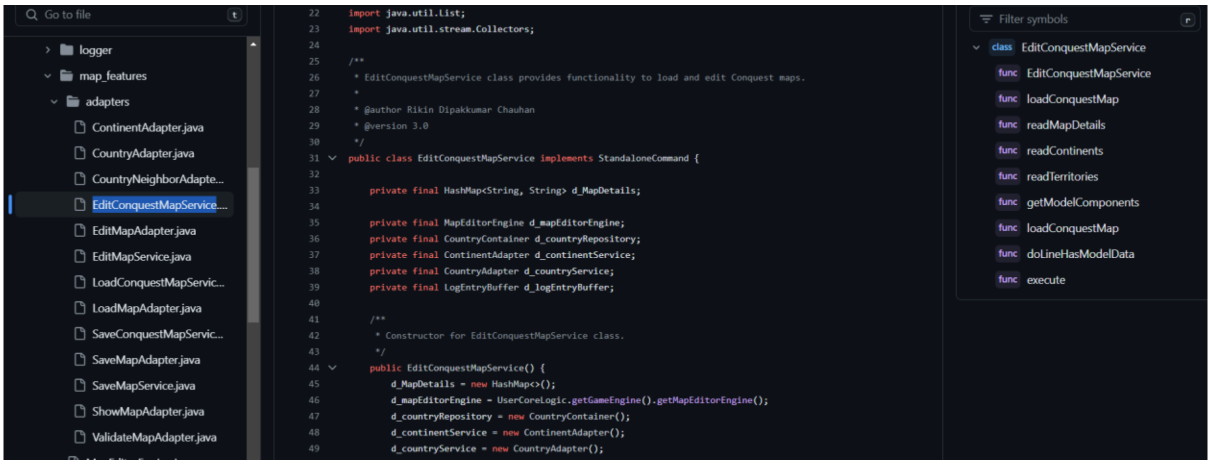
```java
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;


/**
 * EditMapAdapter class represents an adapter for editing maps. It implements the StandaloneCommand interface.
 * This class provides methods to read map data, including continent, country, and their neighbors,
 * from a file and update the map accordingly.
 *
 * @author Rikin Dipakkumar Chauhan
 * @version 1.0
 */
public class EditMapAdapter implements StandaloneCommand {

    private final MapFeatureEngine d_mapEditorEngine;

    private final ContinentContainer d_continentRepository;
    private final CountryContainer d_countryRepository;
    private final ContinentAdapter d_continentService;
    private final CountryAdapter d_countryService;
    private final CountryNeighborAdapter d_countryNeighborService;
    private final LogEntryBuffer d_logEntryBuffer;

    /**
     * Constructs an EditMapAdapter object.
     */
    public EditMapAdapter() {
        d_mapEditorEngine = MapFeatureEngine.getInstance();
        d_continentRepository = new ContinentContainer();
        d_countryRepository = new CountryContainer();
        d_continentService = new ContinentAdapter();
```

Bridge :



```java
package com.APP.Project.UserCoreLogic.map_features.adapters;

import com.APP.Project.UserCoreLogic.exceptions.UserCoreLogicException;

import java.util.List;

/**
 * This class represents an adapter for editing maps.
 *
 * @author Rikin Dipakkumar Chauhan
 * @version 3.0
 */
public class EditMapAdapter extends EditMapService {

    private final EditConquestMapService d_editConquestMapService;

    /**
     * Constructs an EditMapAdapter with the provided EditConquestMapService.
     *
     * @param p_editConquestMapService The EditConquestMapService to use.
     */
    public EditMapAdapter(EditConquestMapService p_editConquestMapService) {
        d_editConquestMapService = p_editConquestMapService;
    }
```

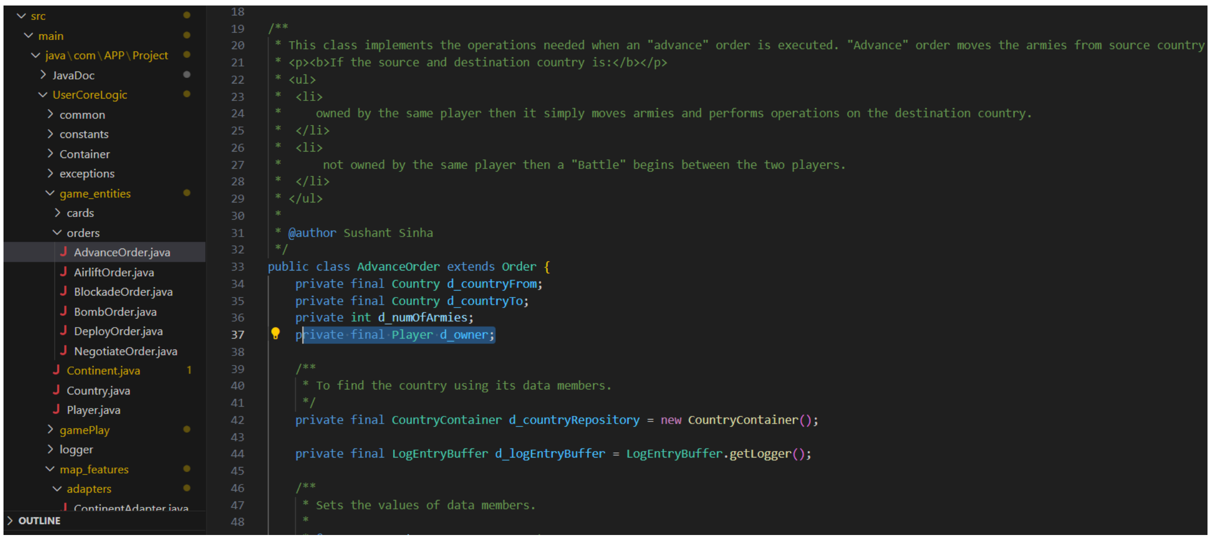
After :



```java
import java.util.List;
import java.util.stream.Collectors;

/**
 * EditConquestMapService class provides functionality to load and edit Conquest maps.
 *
 * @author Rikin Dipakkumar Chauhan
 * @version 3.0
 */
public class EditConquestMapService implements StandaloneCommand {

    private final HashMap<String, String> d_MapDetails;

    private final MapEditorEngine d_mapEditorEngine;
    private final CountryContainer d_countryRepository;
    private final ContinentAdapter d_continentService;
    private final CountryAdapter d_countryService;
    private final LogEntryBuffer d_logEntryBuffer;

    /**
     * Constructor for EditConquestMapService class.
     */
    public EditConquestMapService() {
        d_MapDetails = new HashMap<>();
        d_mapEditorEngine = UserCoreLogic.getGameEngine().getMapEditorEngine();
        d_countryRepository = new CountryContainer();
        d_continentService = new ContinentAdapter();
        d_countryService = new CountryAdapter();
```

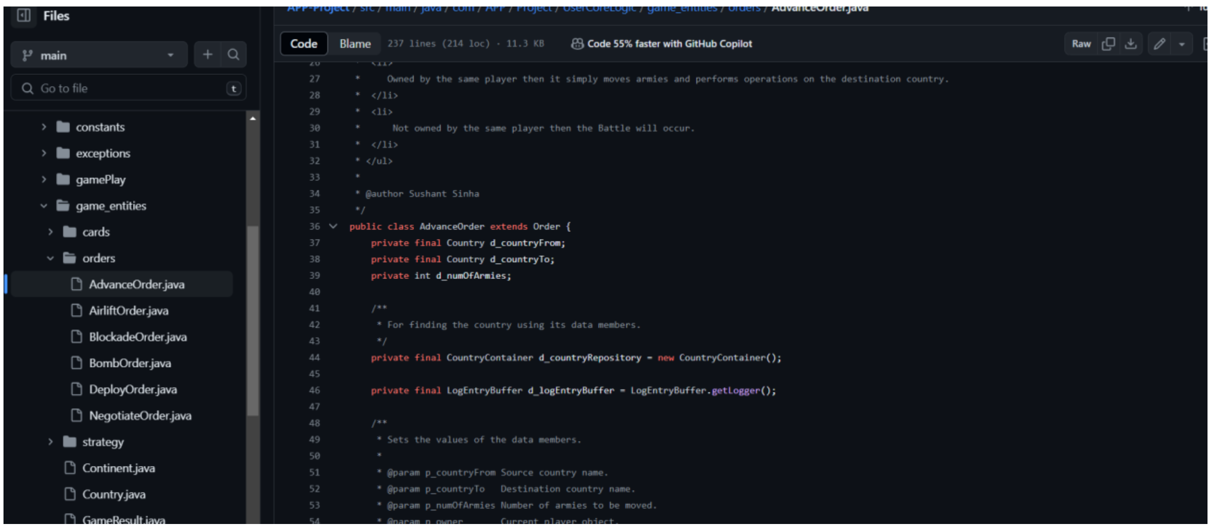
3) **Removal of unused variable:** The removal of unused variables during refactoring is a clean-up process aimed at improving code readability and maintainability by eliminating elements that serve no purpose. Unused variables are those declared in a program but never used in any operation or computation. They can clutter code, leading to confusion and potential errors, especially in large codebases.

Before



```java
/**
 * This class implements the operations needed when an "advance" order is executed. "Advance" order moves the armies from source country
 * <p><b>If the source and destination country is:</b></p>
 * <ul>
 *  <li>
 *     owned by the same player then it simply moves armies and performs operations on the destination country.
 *  </li>
 *  <li>
 *      not owned by the same player then a "Battle" begins between the two players.
 *  </li>
 * </ul>
 *
 * @author Sushant Sinha
 */
public class AdvanceOrder extends Order {
    private final Country d_countryFrom;
    private final Country d_countryTo;
    private int d_numOfArmies;
    private final Player d_owner;

    /**
     * To find the country using its data members.
     */
    private final CountryContainer d_countryRepository = new CountryContainer();

    private final LogEntryBuffer d_logEntryBuffer = LogEntryBuffer.getLogger();

    /**
     * Sets the values of data members.
     *
```

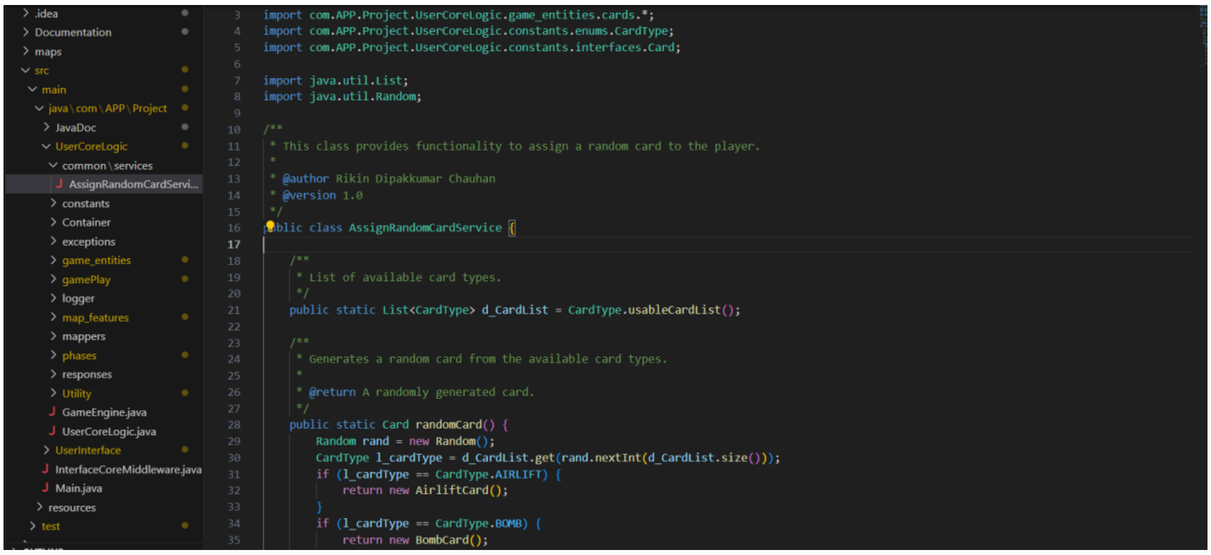
After



```java
 *     Owned by the same player then it simply moves armies and performs operations on the destination country.
 * </li>
 * <li>
 *      Not owned by the same player then the Battle will occur.
 * </li>
 * </ul>
 *
 * @author Sushant Sinha
 */
public class AdvanceOrder extends Order {
    private final Country d_countryFrom;
    private final Country d_countryTo;
    private int d_numOfArmies;

    /**
     * For finding the country using its data members.
     */
    private final CountryContainer d_countryRepository = new CountryContainer();

    private final LogEntryBuffer d_logEntryBuffer = LogEntryBuffer.getLogger();

    /**
     * Sets the values of the data members.
     *
     * @param p_countryFrom Source country name.
     * @param p_countryTo   Destination country name.
     * @param p_numOfArmies Number of armies to be moved.
```

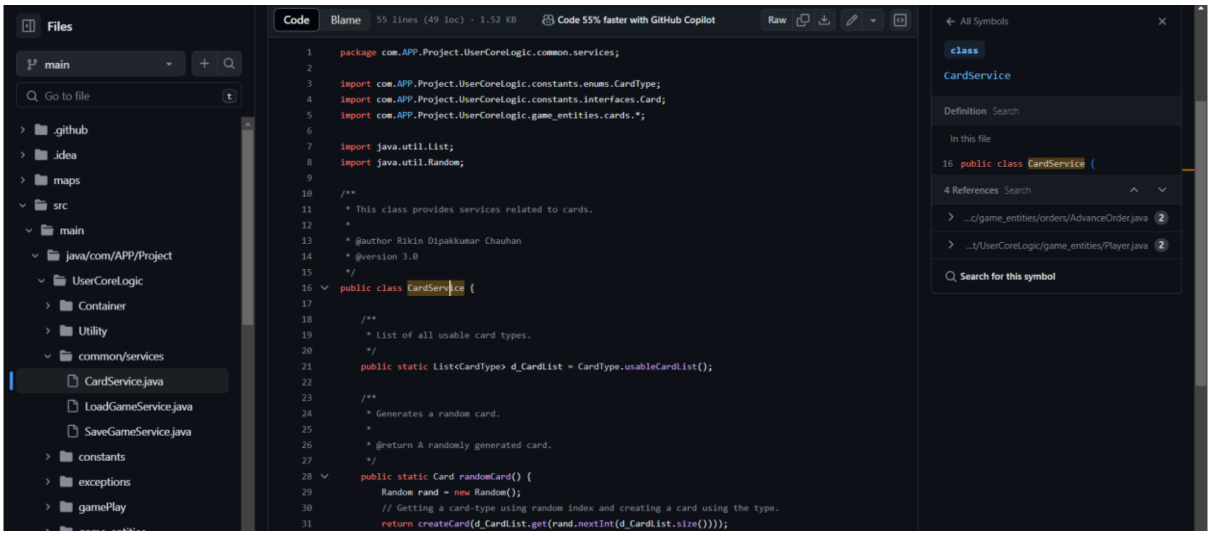
4) **Renaming:** It is a fundamental refactoring technique aimed at improving the clarity, readability, and maintainability of code. It involves changing the names of variables, methods, classes, or even namespaces to better reflect their purpose, functionality, or usage within the application.

Before



```java
import com.APP.Project.UserCoreLogic.game_entities.cards.*;
import com.APP.Project.UserCoreLogic.constants.enums.CardType;
import com.APP.Project.UserCoreLogic.constants.interfaces.Card;

import java.util.List;
import java.util.Random;

/**
 * This class provides functionality to assign a random card to the player.
 *
 * @author Rikin Dipakkumar Chauhan
 * @version 1.0
 */
public class AssignRandomCardService {

    /**
     * List of available card types.
     */
    public static List<CardType> d_CardList = CardType.usableCardList();

    /**
     * Generates a random card from the available card types.
     *
     * @return A randomly generated card.
     */
    public static Card randomCard() {
        Random rand = new Random();
        CardType l_cardType = d_CardList.get(rand.nextInt(d_CardList.size()));
        if (l_cardType == CardType.AIRLIFT) {
            return new AirliftCard();
        }
        if (l_cardType == CardType.BOMB) {
            return new BombCard();
```

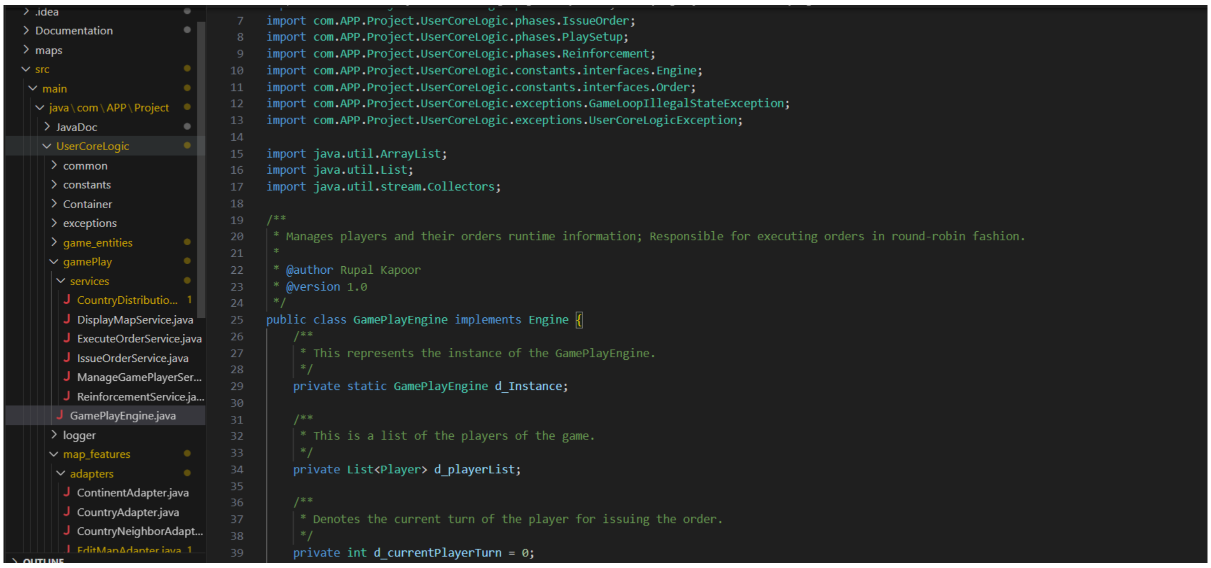
After



```java
package com.APP.Project.UserCoreLogic.common.services;

import com.APP.Project.UserCoreLogic.constants.enums.CardType;
import com.APP.Project.UserCoreLogic.constants.interfaces.Card;
import com.APP.Project.UserCoreLogic.game_entities.cards.*;

import java.util.List;
import java.util.Random;

/**
 * This class provides services related to cards.
 *
 * @author Rikin Dipakkumar Chauhan
 * @version 3.0
 */
public class CardService {

    /**
     * List of all usable card types.
     */
    public static List<CardType> d_CardList = CardType.usableCardList();

    /**
     * Generates a random card.
     *
     * @return A randomly generated card.
     */
    public static Card randomCard() {
        Random rand = new Random();
        // Getting a card-type using random index and creating a card using the type.
        return createCard(d_CardList.get(rand.nextInt(d_CardList.size())));
```

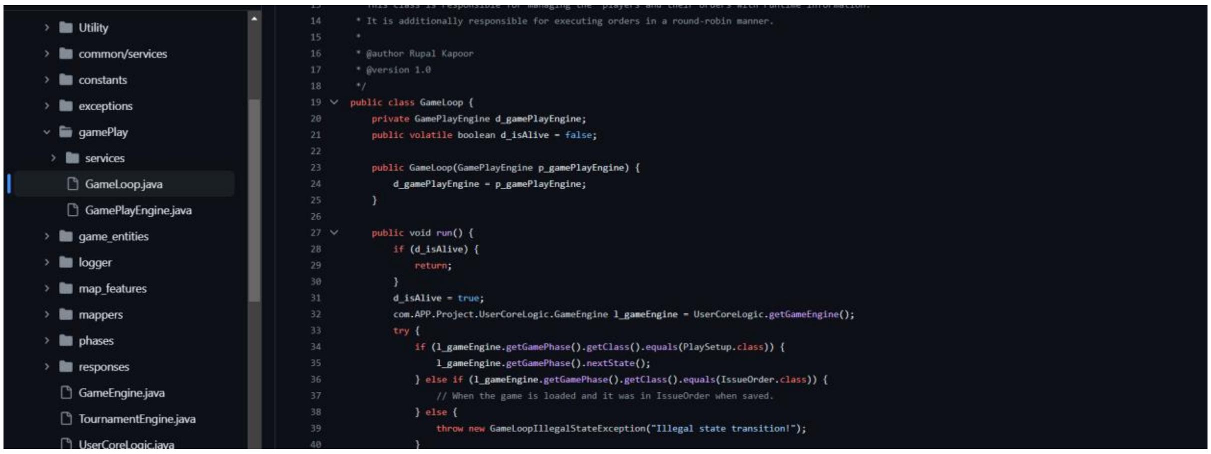
5) **Performance Optimization:** Identifying and rewriting inefficient code segments to improve application performance.
Previously the GameEngine was responsible for running the entire program into round robin manner but now a new class named game loop is created and it handle that responsibility.

Before



```java
import com.APP.Project.UserCoreLogic.phases.IssueOrder;
import com.APP.Project.UserCoreLogic.phases.PlaySetup;
import com.APP.Project.UserCoreLogic.phases.Reinforcement;
import com.APP.Project.UserCoreLogic.constants.interfaces.Engine;
import com.APP.Project.UserCoreLogic.constants.interfaces.Order;
import com.APP.Project.UserCoreLogic.exceptions.GameLoopIllegalStateException;
import com.APP.Project.UserCoreLogic.exceptions.UserCoreLogicException;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

/**
 * Manages players and their orders runtime information; Responsible for executing orders in round-robin fashion.
 *
 * @author Rupal Kapoor
 * @version 1.0
 */
public class GamePlayEngine implements Engine {
    /**
     * This represents the instance of the GamePlayEngine.
     */
    private static GamePlayEngine d_Instance;

    /**
     * This is a list of the players of the game.
     */
    private List<Player> d_playerList;

    /**
     * Denotes the current turn of the player for issuing the order.
     */
    private int d_currentPlayerTurn = 0;
```

After



```java
 * It is additionally responsible for executing orders in a round-robin manner.
 *
 * @author Rupal Kapoor
 * @version 1.0
 */
public class GameLoop {
    private GamePlayEngine d_gamePlayEngine;
    public volatile boolean d_isAlive = false;

    public GameLoop(GamePlayEngine p_gamePlayEngine) {
        d_gamePlayEngine = p_gamePlayEngine;
    }

    public void run() {
        if (d_isAlive) {
            return;
        }
        d_isAlive = true;
        com.APP.Project.UserCoreLogic.GameEngine l_gameEngine = UserCoreLogic.getGameEngine();
        try {
            if (l_gameEngine.getGamePhase().getClass().equals(PlaySetup.class)) {
                l_gameEngine.getGamePhase().nextState();
            } else if (l_gameEngine.getGamePhase().getClass().equals(IssueOrder.class)) {
                // When the game is loaded and it was in IssueOrder when saved.
            } else {
                throw new GameLoopIllegalStateException("Illegal state transition!");
            }
```