**Possible Refactoring Targets :-**

Below are the 15 potential refactoring targets that were considered for this build. The first 5 targets have been implemented and is discussed in detail in the next section of this document :-

**1)** Implement Strategy Pattern

**2)** Implement Adapter Pattern

**3)** Modify gameplayer command to pass strategy as an argument

**4)** Modify savemap command to pass Map type (Conquest or Domination) to save map

**5)** Refactored saveMap() and editMap() methods to explicitly handle different map types

**6)** Private Members: Instead of having public fields, they (E.g., d\_playerName, d\_holdingCountries, d\_currentArmyCount, d\_deployingPlayers, etc) can be encapsulated, and getter and setter methods can be provided. This would provide better control over the access and modification of the class's internal state. **[Player.java, Deploy.java]**

**7)** Create Constants: The probability numbers (e.g., 0.6, 0.7) in the generateBooleanWithProbability method can be replaced with named constants or parameters to improve code readability and maintainability. **[Advance.java]**

*private static final double DEFENDING\_PROBABILITY = 0.6;*

*private static final double ATTACKING\_PROBABILITY = 0.7;*

**8)** Improve Variable Naming: The variable names can be improved for clarity. For example, *l\_defendingNum* and *l\_attackingNum* could be renamed to something more descriptive like *l\_defendingArmies* and *l\_ attackingArmies*. **[Advance.java]**

**9)** Extract Methods: Complex parts of the execute method can be extracted into separate methods with meaningful names. This improves the readability of the execute method and allows for better organization. **[Advance.java]**

*private void handleSameCountryHolderCase() {*

*// Logic for when the source and target countries have the same owner*

*}*

*private void handleDifferentCountryHolderCase() {*

*// Logic for when the source and target countries have different owners*

*}*

**10)** Simplify Constructor Logic: Instead of subtracting *d\_armyCount* directly in the *Deploy()* constructor, we can update it within the execute method. This separation of concerns can improve clarity. **[Deploy.java]**

*public Deploy(Player p\_player, int p\_deployArmyCount, Country p\_targetCountry) {*

*d\_deployingPlayer = p\_player;*

*d\_armyCount = p\_deployArmyCount;*

*d\_targetCountry = p\_targetCountry;*

*d\_isExecuted = false;*

*}*

**11)** Create constant: The minimum army count, i.e. *0* can be replaced with a named constant or parameter. Then *MIN\_ARMY\_COUNT* can be used in the *isValid()* method. **[Deploy.java]**

*private static final int MIN\_ARMY\_COUNT = 0;*

*if (d\_armyCount <= MIN\_ARMY\_COUNT) {*

*System.out.println("Army count is less than or equal to zero..");*

*return false;*

*}*

**12)** Reduce Method Length: The *createOrders()* method is quite lengthy. It can be broken down into smaller methods with specific responsibilities to improve readability and maintainability. **[CreateOrders.java]**

*private void checkForGameConclusion(List<Player> currPlayingPlayers) { ... }*

*private void clearDiplomacyList(List<Player> currPlayingPlayers) { ... }*

*private void takeOrdersFromPlayers(List<Player> currPlayingPlayers) { ... }*

**13)** Constants for File Paths: Hard-coded file paths can be replaced with constants or configuration values **[GameUtils.java]**

*// Hard-coded path*

*File myObj = new File("src/main/resources/maps/" + arguments.get(0) + ".map");*

*// To replace with a constant*

*final String MAPS\_DIRECTORY = "src/main/resources/maps/";*

*File myObj = new File(MAPS\_DIRECTORY + arguments.get(0) + ".map");*

**14)** Refactor Random Number Generation: using a shared Random instance instead of creating a new one each time can be considered. **[GameUtils.java]**

*// To create static final Random instance*

*private static final Random random = new Random();*

*// To use a shared instance in the generateRandomNumber method*

*public static int generateRandomNumber(int minValue, int maxValue) {*

*int range = maxValue - minValue + 1;*

*return random.nextInt(range) + minValue;*

**15)** Use Try-With-Resources: try-with-resources can be used for handling resources like Scanner. **[GameUtils.java]**

*try (Scanner fileScanner = new Scanner(myObj)) {*

*// Existing code...*

*} catch (Exception e) {*

*// Handle exceptions.*

*}*

**Refactoring Implementation: -**

1. **Implemented Strategy Pattern**

Our code implements the Strategy Pattern using PlayerStrategy, where different strategies for creating orders are encapsulated in separate strategy classes.

Both adapters in our code (MapFileAdapter and MapWriterAdapter) bridge the gap between the existing classes (ConquestFileParser and ConquestFileWriter) and the target interfaces (MapFileParser and MapFileWriter). They allow the client code to work with the target interfaces while using the functionality provided by the existing classes through adaptation.

* Strategy Interface (PlayerStrategy):
  + PlayerStrategy is the strategy interface that declares the common methods, such as createOrder and getStrategyName, that all concrete strategies must implement.
* Concrete Strategies (AggressiveStrategy, BenevolentStrategy, CheaterStrategy, RandomStrategy):
  + Each concrete strategy class extends PlayerStrategy and provides its own implementation of the createOrder and getStrategyName methods.
  + Each strategy class encapsulates a specific behavior for creating orders.
* Context (Player):
  + The Player class contains a reference to a PlayerStrategy (d\_targetPlayer), allowing the player to switch strategies dynamically.
* Usage of Strategies:
  + The Player class delegates the responsibility of creating orders to its associated strategy by calling the createOrder method.
  + The Player class can also query the strategy name using the getStrategyName method.

A screenshot of a computer

Description automatically generated

**Corresponding Tests:** AggressiveStrategyTest.java, BenevolentStrategyTest.java, CheaterStrategyTest.java, PlayerStrategyTest.java, RandomStrategyTest.java

1. **Implemented Adapter Pattern**

The Adapter Pattern is implemented in our code using two classes: MapFileAdapter and MapWriterAdapter. When a map file is saved, the user is given the option as to which file format to use as output.

* MapFileAdapter
  + Target (MapFileParser):
    - MapFileParser is the target interface that the client code (the code using MapFileAdapter) expects to work with.
  + Adaptee (ConquestFileParser):
    - ConquestFileParser is the existing class that needs to be adapted to work with the MapFileParser interface.
  + Adapter (MapFileAdapter):
    - MapFileAdapter extends MapFileParser, acting as an adapter to translate calls from MapFileParser to ConquestFileParser.
  + Constructor:
    - The constructor of MapFileAdapter takes a ConquestFileParser as a parameter, initializing it with the name of the map.
  + Method Translation:
    - The parseMapFile method in MapFileAdapter translates calls to parseMapFile from MapFileParser to ConquestFileParser.
* MapWriterAdapter
  + Target (MapFileWriter):
    - MapFileWriter is the target interface that the client code (the code using MapWriterAdapter) expects to work with.
  + Adaptee (ConquestFileWriter):
    - ConquestFileWriter is the existing class that needs to be adapted to work with the MapFileWriter interface.
  + Adapter (MapWriterAdapter):
    - MapWriterAdapter extends MapFileWriter, acting as an adapter to allow using ConquestFileWriter as a MapFileWriter.
  + Constructor:
    - The constructor of MapWriterAdapter takes a ConquestFileWriter as a parameter, initializing it with the target map.
  + Method Delegation:
    - The saveMap method in MapWriterAdapter delegates the saveMap method to the corresponding method in ConquestFileWriter.

A screenshot of a computer

Description automatically generated

**Corresponding Tests:** ConquestFileWriterTest.java, MapWriterAdapterTest.java

1. **Modify gameplayer command to pass strategy as an argument.**

In the previous builds, the gameplayer command took only playername as argument to add or remove players. After implementing the Strategy pattern, it now has been modified to take an additional argument, i.e., the strategy type while adding players. However, this is optional and if no strategy is provided while adding the player(s), then the game will implement it as “Human” by default and will run as per the user-given commands.

A screenshot of a computer

Description automatically generated

**Corresponding Tests:** PlayerTest.java

1. **Modify savemap command to pass Map type(Conquest or Domination) to save the map**

In the previous builds, the savemap command took only one argument, i.e., file name to save the map. After implementing the Adapter pattern, it now has been modified to take an additional argument to set the format in which map is to be saved. The two formats are: Conquest and Domination.

A screenshot of a computer

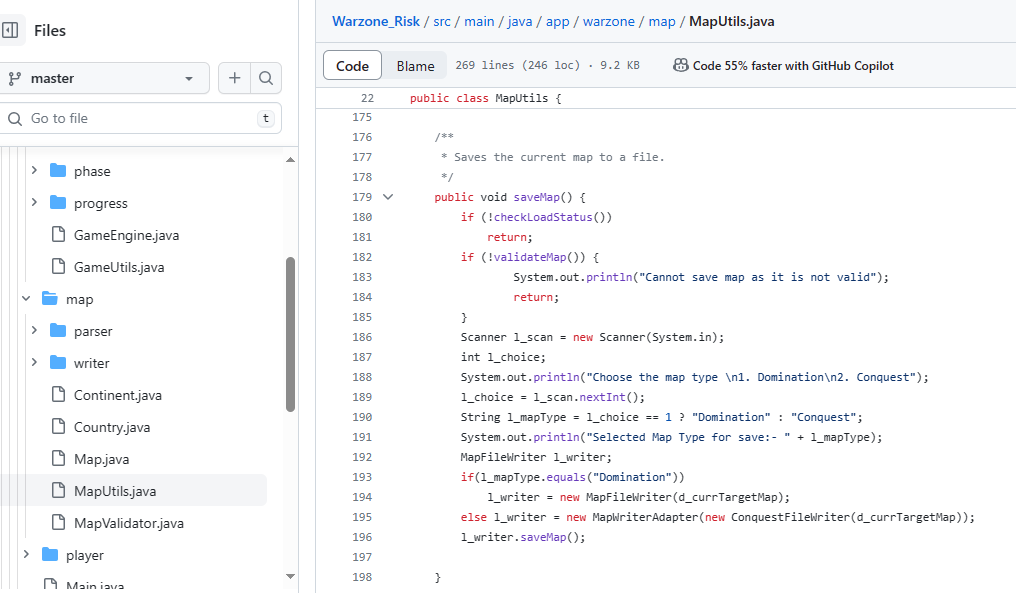
Description automatically generated

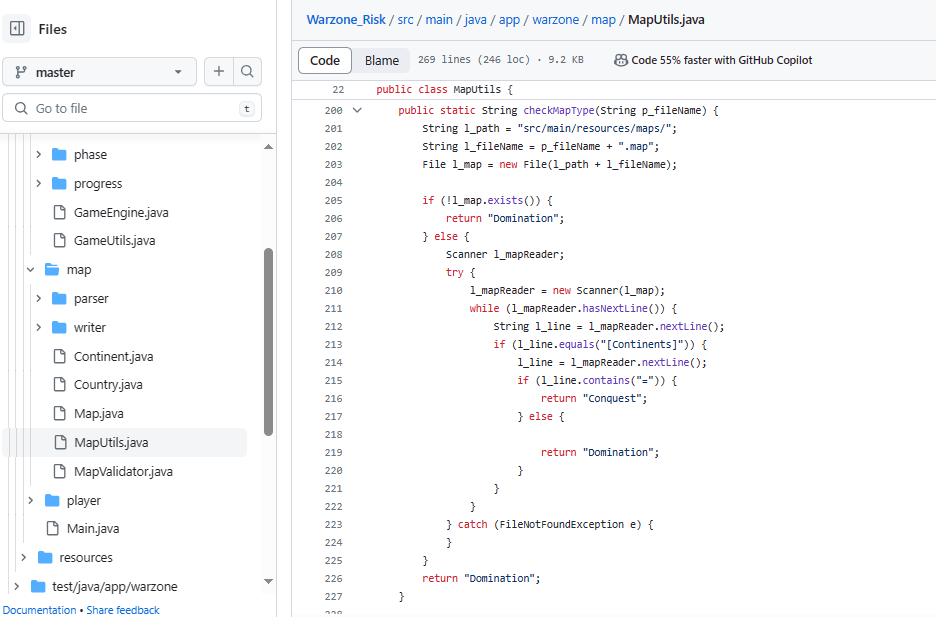
**Corresponding Tests:** MapFileParserTest.java, MapFileAdapterTest.java

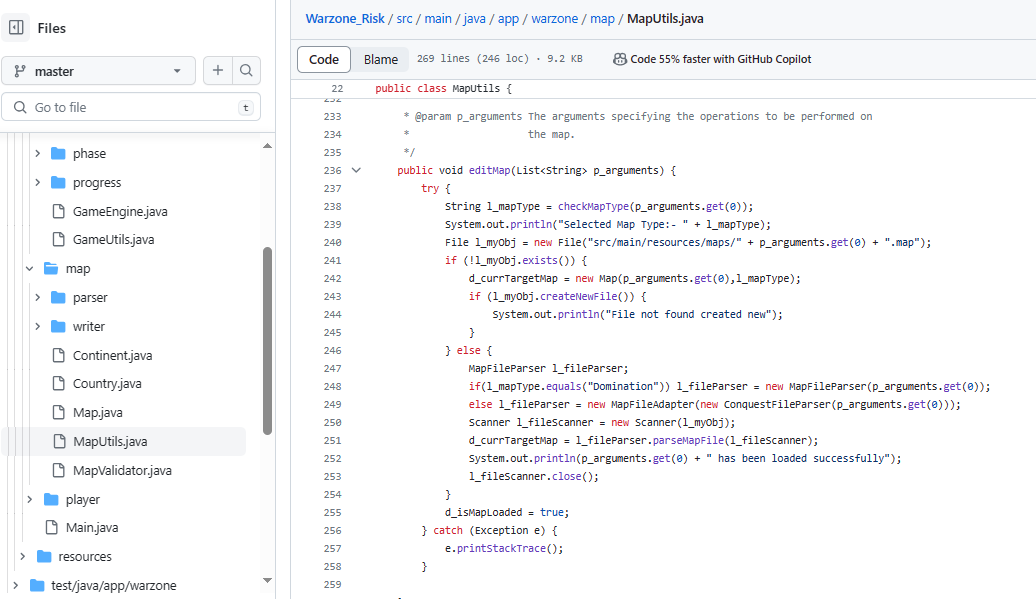
1. **Modify saveMap() method to explicitly handle different map types**

The *saveMap()* method has been modified to use *MapFileWriter* for Domination maps and an adapter (*MapWriterAdapter*) to adapt *ConquestFileWriter* for Conquest maps. It also involves user interaction through the console (Scanner class) to select the map type.

The *checkMapType()* method has also been implemented to check if the map file already exists by inspecting the content of an existing map file before determining its type. If the file exists, it reads the file to determine the map type (Domination or Conquest). If the file does not exist, it returns the default type ("Domination").







**Corresponding Tests:** MapUtilsTest.java