Build 2 – Refactoring Document

To pinpoint potential refactoring candidates in the build (#2), the focus was placed on the following criteria:

- Methods containing multiple sets of logic.
- Repeated use of similar logic in various locations.
- Classes with an abundance of methods.
- Extensive nesting and complex conditional logic structures.

Refactoring Targets:

1. Revising the "loadFile" method included implementing try-with-resources for efficient file reading and resource management in the "loadmap" function.

2. For improved error handling, consider modifying the "loadFile" method within the "loadmap" function to throw an IOException when an error occurs during file reading.

3. The "updatePlayers" method in the "playerservice" has multiple responsibilities, such as parsing a player's name, handling player-related actions (additions or removals), and updating the game state. To improve code structure, it's advisable to decompose this method into smaller, more focused functions to handle each of these tasks separately.

4. As part of the refactoring process in the "playerservice" while removing a player, consider introducing a custom exception that should be thrown when a player is not found.

5. The class variables in the "showmap" class, including `d_players`, `d_gameState`, `d_map`, `d_countries`, and `d_continents`, should be changed to private access. Additionally, it's recommended to create public getter methods to provide controlled access to these variables, ensuring encapsulation.

6. The extensive "showMap" method would benefit from decomposition into smaller, more specialized methods, each responsible for handling specific tasks.

7. Modify the "executeLoadMap" method to make use of dependency injection for the `d_mapService`. The method should no longer depend on an internally stored `d_mapService`, allowing it to accept any implementation of the `MapService` that can be provided during invocation. This change enhances flexibility and testability.

8. To adhere to the Single Responsibility Principle and improve code organization, consider breaking down the "startup" class and other similar phases into smaller, more specialized classes. Each of these new classes should be responsible for a single aspect of the game's startup phase, such as input handling, map editing, game player actions, and so on. This approach enhances code modularity and maintainability.

9. Wherever applicable, use the Java Stream API to streamline loops and operations on collections for improved readability and maintainability of your code.

10. Disallow progression in the game unless there are at least two players.

Potential Refactoring Targets:

11. The state pattern has been applied by introducing a foundational GameplayPhase class responsible for managing commands and defining abstract methods.

12. Implement command pattern for processing of orders.

13. Break the "loadMap" method into smaller, specialized functions for loading continents, countries, and borders

14. Initialisation of deploy order

15. Added edit function method in mapService class.

State Pattern:

Before: All Command-specific operations were implemented within the game engine controller

```
public class GameEngineController {

    /**
     * d_gameState stores the information about current GamePlay.
     */
    GameState d_gameState = new GameState();

    /**
     * d_mapService instance is used to handle load, read, parse, edit, and save map
     * file.
     */
    MapService d_mapService = new MapService();

    /**
     * Player Service instance is used to modify player's list and give orders.
     */
    PlayerService d_playerService = new PlayerService();

    /**
     * The main method responsible for accepting command from users and redirecting
     * those to corresponding logical flows.
     *
     * @param p_args the program doesn't use default command line arguments
     */
    Run | Debug
    public static void main(String[] p_args) {
        GameEngineController l_game = new GameEngineController();

        l_game.initGamePlay();
    }
```

After: Created GameplayPhase class that manages commands and abstract methods, shifting command-specific operations from the game engine controller to their respective gameplay phases and executes command based on different phases.

GamePlayPhase.java × | MapService.java | Map.java | Command.java ...\CommonFunctions | CommonCode.java | GameEngineCtx.java

src > main > java > Models > GamePlayPhase.java > {} Models

```java
import constants.AppConstants;

/**
 * This abstract class lists the methods required for all game phases.
 */
public abstract class GamePlayPhase {

    /**
     * d_gameState stores the information about current GamePlay.
     */
    GameState d_gameState;

    /**
     * d_gameEngineCtx manage the state of the game engine and control gameplay.
     */
    GameEngineCtx d_gameEngineCtx;

    /**
     * d_mapService instance is used to perform map operations such as load, read,
     * parse, edit, and save map file.
     */
    MapService d_mapService = new MapService();

    /**
     * Player Service instance is used to modify player's list and give orders.
     */
    PlayerService d_playerService = new PlayerService();

    /**
     * It's a flag used to determine whether the map has been loaded.
     */
    boolean l_mapLoadSuccess;

    /**
     * A constructor for initializing the current game engine's value.
     *
     * @param p_gameEngineCtx Instance of the game engine used to update the
```

IssueOrderPhase.java ×

src > main > java > Models > IssueOrderPhase.java

```java
    protected void executeCardAction(String p_enteredCommand, Player p_player) throws IOException {
        if (p_player.getD_playerOwnedCards().contains(p_enteredCommand.split(" ")[0])) {
            p_player.handleCardActions(p_enteredCommand, d_gameState);
            d_gameEngineCtx.setD_gameEngineCtxLog(p_player.d_playerLogMessage, AppConstants.ORDER_EFFECT);
        }
        p_player.checkAdditionalOrders();
    }

    @Override
    protected void executeShowMap(Command p_command, Player p_player) throws InvalidCommand, IOException, InvalidMap
        ShowMap l_showMap = new ShowMap(d_gameState);
        l_showMap.showMap();

        askForOrder(p_player);
    }

    @Override
    protected void executeAdvance(String p_command, Player p_player) throws IOException {
        p_player.initAdvanceOrder(p_command, d_gameState);
        d_gameState.updateLog(p_player.getD_playerLogMessage(), AppConstants.ORDER_EFFECT);
        p_player.checkAdditionalOrders();
    }

    @Override
    protected void executeDeploy(String p_command, Player p_player) throws IOException {
        p_player.initDeployOrder(p_command);
        d_gameState.updateLogFile(p_player.getD_playerLogMessage(), AppConstants.ORDER_EFFECT);
        p_player.checkAdditionalOrders();
    }
```

```java
OrderExecutionPhase.java ×
src > main > java > Models > OrderExecutionPhase.java
 *
 * @param p_gameEngineController Instance of the game engine used to update the
 *                               state.
 * @param p_gameState            The current game state associated with the game
 *                               engine instance.
 */
public OrderExecutionPhase(GameEngineCtx p_gameEngineController, GameState p_gameState) {
    super(p_gameEngineController, p_gameState);
}

@Override
protected void executeCardAction(String p_enteredCommand, Player p_player) throws IOException {
    LogInvalidCommandInCurrentPhase();
}

@Override
protected void executeAdvance(String p_command, Player p_player) {
    LogInvalidCommandInCurrentPhase();
}

@Override
public void initPhase() {
    while (d_gameEngineCtx.getD_CurrentPhase() instanceof OrderExecutionPhase) {
        executeOrders();

        ShowMap l_map_view = new ShowMap(d_gameState);
        l_map_view.showMap();

        if (this.checkGameHasEnded(d_gameState))
            break;
```

```java
StartUpPhase.java ×
src > main > java > Models > StartUpPhase.java
 * @param p_gameState            The current game state associated with the game
 *                               engine instance.
 */
public StartUpPhase(GameEngineCtx p_gameEngineController, GameState p_gameState) {
    super(p_gameEngineController, p_gameState);
}

@Override
protected void executeCardAction(String p_enteredCommand, Player p_player) throws IOException {
    LogInvalidCommandInCurrentPhase();
}

/**
 * {@inheritDoc}
 */
public void initPhase() {
    BufferedReader l_reader = new BufferedReader(new InputStreamReader(System.in));

    while (d_gameEngineCtx.getD_CurrentPhase() instanceof StartUpPhase) {
        try {
            System.out.println("Enter Game Commands or type 'exit' for quitting");
            String l_commandEntered = l_reader.readLine();

            handleCommand(l_commandEntered, null);
        } catch (InvalidCommand | InvalidMap | IOException l_exception) {
            d_gameEngineCtx.setD_gameEngineCtxLog(l_exception.getMessage(),
                    AppConstants.ORDER_EFFECT);
        }
    }
```

Reason: The state pattern improves code structure and maintainability by encapsulating different states and behaviors, enabling dynamic state transitions and enhancing reusability.

Added Test Cases:

1. testValidateStartupPhase – Validates the gameplay phase when the game starts.
2. testGameEndCondition – The verifies whether a player has successfully conquered all countries, and upon achievement of this condition, initiates an immediate game exit.

Modified Test Cases (if any): None

Command Pattern:

Before: The system previously supported the processing of orders in a single class OrderImpl.

```java
     * Set the count of armies to be allocated.
     *
     * @param p_armiesToAllocate count of armies to be allocated
     */
    public void setD_armiesToAllocate(Integer p_armiesToAllocate) {
        this.d_armiesToAllocate = p_armiesToAllocate;
    }

    /**
     * Executes the order based on the given game state and player information.
     *
     * @param p_gameStateInfo The current game state information.
     * @param p_player        The player initiating the order.
     */
    public void executeOrder(GameState p_gameStateInfo, Player p_player) {
        if (this.d_orderInstruction.equals(anObject:"deploy")) {
            if (this.checkCountryOwnership(p_player, this)) {
                this.deployArmies(this, p_gameStateInfo, p_player);
                System.out.println("\nExecution of the order was successful. " + this.getD_armiesToAllocate()
                        + " armies have been deployed to the following country: "
                        + this.getD_targetCountry());
            } else {
                System.out.println(
                        "\nThe order was not executed because the target country specified in the deployment command does not belong to the player: "
                                + p_player.getD_playerName());
            }
        } else {
            System.out.println(x:"The order was not executed because it was an invalid order command.");
        }
    }

    /**
     * Checks if a player owns a country based on the target country name.
     *
     * @param p_player The player to check ownership for.
     * @param p_order  The order containing the target country name to check
```

After: A command pattern has been implemented to facilitate the processing of orders. This means separate classes such as advance and deploy classes have been implemented to process orders.

```java
     * Constructor receiving all necessary parameters for executing the order.
     *
     * @param p_playerInitiator       Player initiating the order
     * @param p_sourceCountryName     Country from which armies are transferred
     * @param p_targetCountry         Country receiving the new armies
     * @param p_numberOfArmiesToPlace Number of armies to be added
     */
    public Advance(Player p_playerInitiator, String p_sourceCountryName, String p_targetCountry,
            Integer p_numberOfArmiesToPlace) {
        this.d_targetCountry = p_targetCountry;
        this.d_sourceCountry = p_sourceCountryName;
        this.d_playerInitiator = p_playerInitiator;
        this.d_armiesToAllocate = p_numberOfArmiesToPlace;
    }

    /**
     * Executes the order and makes necessary changes to the game state.
     *
     * @param p_gameState Current state of the game
     */
    @Override
    public void execute(GameState p_gameState) {
        if (isValid(p_gameState)) {
            Player l_targetCountryPlayer = fetchTargetCountryPlayer(p_gameState);
            if (l_targetCountryPlayer.d_negotiatedWith.contains(this.d_playerInitiator)) {
                this.setD_orderExecutionLog(
                        "Advance Command cannot be executed as Player:" + this.d_playerInitiator.getD_playerName()
                                + " negotiated terms with Player:" + l_targetCountryPlayer.getD_playerName(),
                        AppConstants.LOG_MSG);
                p_gameState.updateLog(orderExecutionLog(), AppConstants.ORDER_EFFECT);
            } else {
                Country l_targetCountry = p_gameState.getD_map().getCountryByName(d_targetCountry);
                Country l_sourceCountry = p_gameState.getD_map().getCountryByName(d_sourceCountry);
                Integer l_sourceArmyCountUpdate = l_sourceCountry.getD_armyCount() - this.d_armiesToAllocate;
                l_sourceCountry.setD_armyCount(l_sourceArmyCountUpdate);
                processOrder(p_gameState, l_targetCountryPlayer, l_targetCountry, l_sourceCountry);
            }
        } else {
```

```java
    /**
     * parameterized constructor.
     *
     * @param p_playerInitiator  player that created the order
     * @param p_targetCountry    country that will receive the new armies
     * @param p_armiesToAllocate number of armies to be added
     */
    public Deploy(Player p_playerInitiator, String p_targetCountry, Integer p_armiesToAllocate) {
        this.d_targetCountry = p_targetCountry;
        this.d_playerInitiator = p_playerInitiator;
        this.d_armiesToAllocate = p_armiesToAllocate;
    }

    /**
     * Executes the deploy order.
     *
     * @param p_gameState current state of the game.
     */
    @Override
    public void execute(GameState p_gameState) {
        if (isValid(p_gameState)) {
            for (Country l_country : p_gameState.getD_map().getD_countries()) {
                if (l_country.getD_countryName().equalsIgnoreCase(this.d_targetCountry)) {
                    Integer l_armiesToUpdate = l_country.getD_armyCount() == null ? this.d_armiesToAllocate
                            : l_country.getD_armyCount() + this.d_armiesToAllocate;
                    l_country.setD_armyCount(l_armiesToUpdate);
                    this.setD_orderExecutionLog(l_armiesToUpdate
                            + " armies have been deployed successfully on country : " + l_country.getD_countryName(),
                            AppConstants.LOG_MSG);
                }
            }
        } else {
            this.setD_orderExecutionLog("Deploy Order = " + "deploy" + " " + this.d_targetCountry + " "
                    + this.d_armiesToAllocate + " is not executed since Target country: "
                    + this.d_targetCountry + " given in deploy command does not belongs to the player : "
                    + d_playerInitiator.getD_playerName(), AppConstants.ERROR_LOG_MSG);
```

Reason: The Command Pattern promotes decoupling, extensibility, and flexibility in handling requests by encapsulating them as objects, making it easier to add new commands and support undo/redo operations.

Added Test Cases:

AirLiftTest:

1. testAirliftExecution: - Test positive Airlift command execution.
2. testNegativeAirLiftCommand:- Test negative validation of airlift order..

BolckadeTest:

1. testBlockadeExecution: - Test Bolckade Card Execution.
2. testinvalidBlockade : - Test validation of Bolckade card.

BombTest:

1. testBombCardExecution: - Test Bomb Card Execution.
2. testValidBombOrder : - Test validation of bomb card.

DiplomacyTest:

1. testDiplomacyExecution: - Tests to check the functionality of diplomacy.
2. testAdvancePostNegotiation: - Tests whether advance attack orders work after negotiation.
3. testBombPostNegotiation : - Tests whether bomb attack orders work after negotiation.

Modified Test Cases (if any): testExecuteOrder() in OrderImpl moved to respective command test files.

Breaking Down Loadmap:

Before: - The Loadmap method was designed to execute various functionalities.

MapService.java

src > main > java > Services > MapService.java

```java
 * @param p_loadFileName map file name.
 * @return Map object after processing map file.
 */
public Map loadMap(GameState p_gameState, String p_loadFileName) {
    Map l_map = new Map();
    List<String> l_linesOfFile = loadFile(p_loadFileName);
    if (null != l_linesOfFile && !l_linesOfFile.isEmpty()) {

        // Processes the file and stores data in objects.
        List<String> l_continentData = retrieveMetaData(l_linesOfFile, "continent");
        List<Continent> l_continentObjects = analyzeContinentsMetaData(l_continentData);
        List<String> l_countryData = retrieveMetaData(l_linesOfFile, "country");
        List<String> l_bordersMetaData = retrieveMetaData(l_linesOfFile, "border");
        List<Country> l_countryObjects = analyzeCountriesMetaData(l_countryData);

        // Modifies the neighbors of countries within objects.
        l_countryObjects = analyzeBorderMetaData(l_countryObjects, l_bordersMetaData);
        l_continentObjects = linkCountryContinents(l_countryObjects, l_continentObjects);
        l_map.setD_continents(l_continentObjects);
        l_map.setD_countries(l_countryObjects);
        p_gameState.setD_map(l_map);
    }
    return l_map;
}

/**
 * The 'loadFile' method is used to handle the loading and reading of map files.
 *
 * @param p_loadFileName map file name to load.
```

After: - In the new code, we have divided this method into three sections to enhance code comprehension and readability.

Added Test Cases:

testCorrectMapValidate(): checks whether a loaded map is getting validated correctly or not.

Modified test cases: None

MapService.java

src > main > java > Services > MapService.java

```java
    /**
     * The 'loadmap' method handles the processing of map files.
     *
     * @param p_gameState    current state of game.
     * @param p_loadFileName map file name.
     * @return Map object after processing map file.
     */
    public Map loadMap(GameState p_gameState, String p_loadFileName) {
        Map l_map = new Map();
        List<String> l_linesOfFile = loadFile(p_loadFileName);
        if (isValidFileContent(l_linesOfFile)) {
            processMapContent(l_linesOfFile, l_map, p_gameState);
        }
        return l_map;
    }

    /**
     * Checks if the file content is valid.
     *
     * @param p_fileContent List of lines from the file
     * @return true if the file content is valid, false otherwise
     */
    private boolean isValidFileContent(List<String> p_fileContent) {
        return (null != p_fileContent && !p_fileContent.isEmpty());
    }

    /**
```

MapService.java

src > main > java > Services > MapService.java

```java
     *
     * @param p_fileContent List of lines from the file
     * @param p_map         Map object to update with processed data
     * @param p_gameState   GameState to update with the processed map
     */
    private void processMapContent(List<String> p_fileContent, Map p_map, GameState p_gameState) {
        List<String> l_continentData = retrieveMetaData(p_fileContent, AppConstants.CONTINENT);
        List<String> l_countryData = retrieveMetaData(p_fileContent, AppConstants.COUNTRY);
        List<String> l_bordersMetaData = retrieveMetaData(p_fileContent, AppConstants.BORDER);

        List<Continent> l_continentObjects = analyzeContinentsMetaData(l_continentData);
        List<Country> l_countryObjects = analyzeCountriesMetaData(l_countryData);
        l_countryObjects = analyzeBorderMetaData(l_countryObjects, l_bordersMetaData);
        l_continentObjects = linkCountryContinents(l_countryObjects, l_continentObjects);

        p_map.setD_continents(l_continentObjects);
        p_map.setD_countries(l_countryObjects);
        p_gameState.setD_map(p_map);
    }

    /**
     * The 'loadFile' method is used to handle the loading and reading of map files.
     *
     * @param p_loadFileName map file name to load.
     * @return List of lines from map file.
     */
    public List<String> loadFile(String p_loadFileName) {

        String l_filePath = CommonCode.getMapFilePath(p_loadFileName);
```

Initialization of deploy order:-

Before: Initially, the **initDeployOrder** method resided in the **PlayerService** class. Consequently, we had to pass an instance of the **Player** class when issuing orders.

PlayerService.java

src > main > java > Services > PlayerService.java

```java
    */
    public void initDeployOrder(String p_command, Player p_playerObj) {
        // Retrieve the list of player orders or create a new list if it's empty
        List<OrderImpl> l_orderList = CommonCode.isCollectionEmpty(p_playerObj.getD_playerOrder()) ? new ArrayList<>
                : p_playerObj.getD_playerOrder();

        // Extract the country name and number of armies from the command
        String l_country = p_command.split(" ")[1];
        String l_armyCount = p_command.split(" ")[2];

        // Check if the player has enough unallocated armies for the deployment
        if (checkDeployArmyCount(p_playerObj, l_armyCount)) {
            // Print an error message if the deployment order can't be executed due to
            // insufficient armies
            System.out.println(
                    "The deploy order exceeds the player's available unallocated armies and cannot be executed");
        } else {
            // Create an OrderImpl object representing the deployment order
            OrderImpl l_orderObject = new OrderImpl(p_command.split(" ")[0], l_country,
                    Integer.parseInt(l_armyCount));

            // Add the order to the player's list of orders
            l_orderList.add(l_orderObject);
            p_playerObj.setD_playerOrder(l_orderList);

            // Update the player's unallocated armies count
            Integer l_unallocatedArmies = p_playerObj.getD_unallocatedArmyCount() - Integer.parseInt(l_armyCount);
            p_playerObj.setD_unallocatedArmyCount(l_unallocatedArmies);
```

After: - Now that we have relocated the **initDeployOrder** method to the **Player** class, we can effortlessly access the player object using the **this** keyword instead of passing it from the upper layer.

Player.java

src > main > java > Models > Player.java

```java
     *
     * @param p_command The deploy command entered by the player.
     */
    public void initDeployOrder(String p_command) {

        try {
            // Extract the country name and number of armies from the command
            String l_country = p_command.split(" ")[1];
            String l_armyCount = p_command.split(" ")[2];

            // Check if the player has enough unallocated armies for the deployment
            if (checkDeployArmyCount(this, l_armyCount)) {
                // Print an error message if the deployment order can't be executed due to
                // insufficient armies
                this.setD_playerLog(
                        "The deploy order exceeds the player's available unallocated armies and cannot be executed",
                        AppConstants.ERROR_LOG_MSG);
            } else {
                this.d_playerOrder.add(new Deploy(this, l_country, Integer.parseInt(l_armyCount)));
                // Update the player's unallocated armies count
                Integer l_unallocatedArmies = this.getD_unallocatedArmyCount() - Integer.parseInt(l_armyCount);
                this.setD_unallocatedArmyCount(l_unallocatedArmies);

                // Print a message indicating that the order has been added to the execution
                // queue
                this.setD_playerLog("Order Queued for player: " + this.d_playerName, AppConstants.LOG_MSG);

            }
        } catch (Exception l_exception) {
            this.setD_playerLog("Invalid command for deploy", AppConstants.ERROR_LOG_MSG);
```

Added test cases: None

Modified test cases:

testInitDeployOrder(): moved this test from playerServiceTest to player class and call using player object. It is used to check whether armies are updated after deploy order execution

Reason: - Moving the **initDeployOrder** method to the **Player** class and utilizing **this** keyword for player object access enhances code readability, reduces dependencies, and promotes better encapsulation.

EditFunction In MapService: -

Before: - In Build 1, distinct methods—editcontinent, editcountry, and editneighbor—were implemented to individually control the flow of edit operations for continents, countries, and neighbors.

```java
    /**
     * Processing of continents specified in commands for addition or removal from
     * the selected map through the "editmap" operation.
     *
     * @param p_gameState GameState model class object
     * @param p_argument  the argument fetched from the given command
     * @param p_operation the operation to be performed as given by the user
     * @throws IOException triggered in case the file does not exist or the file
     *                     name is invalid
     * @throws InvalidMap  invalidmap exception
     */
    public void editContinent(GameState p_gameState, String p_argument, String p_operation)
            throws IOException, InvalidMap {
        String l_mapFileName = p_gameState.getD_map().getD_mapFile();
        Map l_mapToBeUpdated = (CommonCode.isNull(p_gameState.getD_map().getD_continents())
                && CommonCode.isNull(p_gameState.getD_map().getD_countries()))
                        ? this.loadMap(p_gameState, l_mapFileName)
                        : p_gameState.getD_map();

        if (!CommonCode.isNull(l_mapToBeUpdated)) {
            Map l_updatedMap = addRemoveContinents(l_mapToBeUpdated, p_operation, p_argument);
            p_gameState.setD_map(l_updatedMap);
            p_gameState.getD_map().setD_mapFile(l_mapFileName);
        }
    }

    /**
     * Builds an updated list of continents based on the provided operations
```

After: - In this build, a unified approach has been adopted by consolidating all functionalities into a single method named editFunction to handle the editing operations for continents, countries, and neighbors.

```java
MapService.java
src > main > java > Services > MapService.java
     * @throws InvalidCommand invalid command exception
     */
    public void editFunctions(GameState p_gameState, String p_operation, String p_argument, String p_switchParameter)
            throws IOException, InvalidMap, InvalidCommand {
        Map l_updatedMap;
        String l_mapFileName = p_gameState.getD_map().getD_mapFile();
        Map l_mapToBeUpdated = (CommonCode.isNull(p_gameState.getD_map().getD_continents())
                && CommonCode.isNull(p_gameState.getD_map().getD_countries()))
                        ? this.loadMap(p_gameState, l_mapFileName)
                        : p_gameState.getD_map();

        // Edit Control Logic for Continent, Country & Neighbor
        if (!CommonCode.isNull(l_mapToBeUpdated)) {
            switch (p_switchParameter) {
                case AppConstants.CONTINENT:
                    l_updatedMap = addRemoveContinents(p_gameState, l_mapToBeUpdated, p_operation, p_argument);
                    break;
                case AppConstants.COUNTRY:
                    l_updatedMap = addRemoveCountry(p_gameState, l_mapToBeUpdated, p_operation, p_argument);
                    break;
                case "neighbour":
                    l_updatedMap = addRemoveNeighbour(p_gameState, l_mapToBeUpdated, p_operation, p_argument);
                    break;
                default:
                    throw new IllegalStateException("Unexpected value: " + p_switchParameter);
            }
            p_gameState.setD_map(l_updatedMap);
            p_gameState.getD_map().setD_mapFile(l_mapFileName);
```

Added test cases: None

Modified test cases: for all the below test cases, we had to call the single editFunction method that we had implemented in place of individual methods that included editContinent, editCountry, editNeighbour.

1. testAddContinent
2. testRemoveContinent
3. testAddCountry
4. testRemoveCountry
5. testAddNeighbour
6. testRemoveNeighbour

<u>Reason: -</u> The decision to consolidate edit functionalities into a single edit method aims to enhance code reusability, simplify maintenance, and ensure a consistent and scalable approach across different edit operations.