Refactoring

Concordia University

Department of Computer Science and Software Engineering

Advanced Programming Practices

SOEN 6441 --- Winter 2024

Professor: Joey Paquet

Students: Team 14

Mahdieh Shekarian-40279366

Parsa MohammadRezaei-40277489

AmirMohammad RezaeiPour-40279239

Farid Farahmand-40241630

Mohammad Aminan-40277902

Omer Sayem-40226505

Potential refactoring targets.

Explain how you have identified the potential refactoring targets.

Individual Analysis:

Each team member was assigned the task of conducting an independent analysis of the codebase. We examined the code, identified potential refactoring targets, and documented our findings.

Meeting and Discussion:

Following the individual analysis, a team meeting was held to discuss the findings and select the most promising refactoring targets. During the meeting, each team member presented their identified targets, highlighting the rationale behind their choices.

Refactoring Target Selection:

After a thorough discussion, the team reached a consensus on the top five refactoring targets to prioritize. The selected targets were chosen based on their potential to significantly improve the codebase and address common pain points. Careful consideration was given to the feasibility of the refactoring tasks and their alignment with the project's current goals and timelines.

List of 5 applied refactorings:

1.

```java
public boolean parse(String p_filePath) throws IOException {

  BufferedReader l_reader = new BufferedReader(new FileReader(p_filePath));
  String l_line;
  GameMapReaderAdapter l_gameMapReaderAdapter;
  while ((l_line = l_reader.readLine())!=null) {
    l_line = l_line.trim();
    if (l_line.isEmpty() || l_line.startsWith(";")) {
      continue;
    }

    if (l_line.startsWith("[continents]")) {
      System.out.println("Domination map detected");
      l_gameMapReaderAdapter = new GameMapReaderAdapter(d_gameState, "Domination");
      if (!l_gameMapReaderAdapter.parse(p_filePath)) {
        throw new IOException("Error parsing the map file");
      }
    }

    if (l_line.startsWith("[Map]")) {
      System.out.println("Conquest map detected");
      l_gameMapReaderAdapter = new GameMapReaderAdapter(d_gameState, "Conquest");
      if (!l_gameMapReaderAdapter.parse(p_filePath)) {
        throw new IOException("Error parsing the map file");
      }
    }
  }

  // validate the map after parsing
  if (!validateMap()) {
    d_gameState.removeAction(GameState.GameAction.VALID_MAP_LOADED);
    return false;
  } else {
    d_gameState.setActionDone(GameState.GameAction.VALID_MAP_LOADED);
    d_gameState.setContinents(d_continents);
    d_gameState.setCountries(d_countries);
    return true;
  }



}
```

Before: Without try-with-resource block.

After: Try-with-resource block, will be automatically closed regardless of whether the try will complete normally or abruptly.

```
public boolean parse(String p_filePath) throws IOException {

  try (BufferedReader l_reader = new BufferedReader(new FileReader(p_filePath))) {

    String l_line;
    GameMapReaderAdapter l_gameMapReaderAdapter;
    while ((l_line = l_reader.readLine())!=null) {
      l_line = l_line.trim();
      if (l_line.isEmpty() || l_line.startsWith(";")) {
        continue;
      }

      if (l_line.startsWith("[continents]")) {
        System.out.println("Domination map detected");
        l_gameMapReaderAdapter = new GameMapReaderAdapter(d_gameState, "Domination");
        if (!l_gameMapReaderAdapter.parse(p_filePath)) {
          throw new IOException("Error parsing the map file");
        }
      }

      if (l_line.startsWith("[Map]")) {
        System.out.println("Conquest map detected");
        l_gameMapReaderAdapter = new GameMapReaderAdapter(d_gameState, "Conquest");
        if (!l_gameMapReaderAdapter.parse(p_filePath)) {
          throw new IOException("Error parsing the map file");
        }
      }
    }

    // validate the map after parsing
    if (!validateMap()) {
      d_gameState.removeAction(GameState.GameAction.VALID_MAP_LOADED);
      return false;
    } else {
      d_gameState.setActionDone(GameState.GameAction.VALID_MAP_LOADED);
      d_gameState.setContinents(d_continents);
      d_gameState.setCountries(d_countries);
      return true;
    }
  }
}
```

Before: Without separation of concern, and repeated code.

After: Split into functions with **separation of concern**, **min repeated** code.

```
public boolean parse(String p_filePath) throws IOException {

  try (BufferedReader l_reader = new BufferedReader(new FileReader(p_filePath))) {

    String l_line;
    GameMapReaderAdapter l_gameMapReaderAdapter;
    while ((l_line = l_reader.readLine())!=null) {
      l_line = l_line.trim();
      if (l_line.isEmpty() || l_line.startsWith(";")) {
        continue;
      }

      if (l_line.startsWith("[continents]")) {
        parseMap(p_filePath, DOMINATION_MAP_TYPE);
      }

      if (l_line.startsWith("[Map]")) {
        parseMap(p_filePath, CONQUEST_MAP_TYPE);
      }
    }

    return postParseValidation();
  }
}

/**
 * Parses the map based on the map type.
 *
 * @param p_filePath The file path of the game map file.
 * @param mapType The type of the map.
 * @throws IOException If an I/O error occurs.
 */

private void parseMap(String p_filePath, String mapType) throws IOException {
  System.out.println(mapType + " map detected");
  GameMapReaderAdapter l_gameMapReaderAdapter = new GameMapReaderAdapter(d_gameState, mapType);
  if (!l_gameMapReaderAdapter.parse(p_filePath)) {
        throw new IOException("Error parsing the map file");
    }
}

/**
 * Validates the map after parsing.
 *
 * @return True if the map is valid, false otherwise.
 */

private boolean postParseValidation() {
    if (!validateMap()) {
        d_gameState.removeAction(GameState.GameAction.VALID_MAP_LOADED);
        return false;
    } else {
        d_gameState.setActionDone(GameState.GameAction.VALID_MAP_LOADED);
        d_gameState.setContinents(d_continents);
        d_gameState.setCountries(d_countries);
        return true;
    }
}
```

2.

```
public boolean validateMap() {

    // check if the map has atleast one continent
    if (d_continents.isEmpty()) {
        System.err.println("The map does not contain any continents.");
        printMapRules();
        return false;
    }

    // check if the continents have atleast one country
    if (d_countries.isEmpty()) {
        System.err.println("The map does not contain any countries.");
        printMapRules();
        return false;
    }

    //check if the coutry has atleast one connection
    for (Country country : d_countries.values()) {
        if (country.getAdjacentCountries().isEmpty()) {
            System.err.println("The country '" + country.getCountryId() + "' does not have any connections.");
            printMapRules();
            return false;
        }
    }
    // Check if map is a connected graph
    GameGraphUtils l_graphUtils = new GameGraphUtils();
    if (!GameGraphUtils.isGraphConnected(d_countries)) {
        System.err.println("The map is a disconnected graph.");
        printMapRules();
        return false;
    }

    //check if the contries have self loop using l_graphUtils.isSelfLoop
    if (l_graphUtils.hasSelfLoop(d_countries)) {
        printMapRules();
        return false;
    }

    // Check if each continent is a connected subgraph
    for (Continent continent : d_continents.values()) {
        if (!GameGraphUtils.isContinentConnected(d_countries, continent.getContinentId())) {
            CustomPrint.println(String.valueOf(continent.getContinentId()));
            System.err.println("The continent '" + continent.getContinentName() + "' is a disconnected subgraph.");
            printMapRules();
            return false;
        }
    }

    // If all validations pass
    return true;
}
```

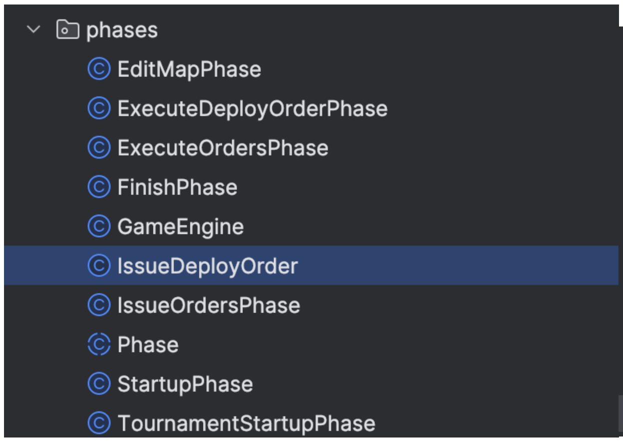
Before: Long method, no separation.

After: **Modeler validation function**.

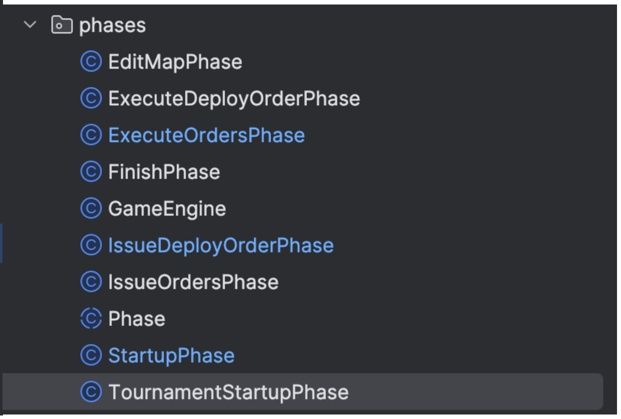
```
/**
 * Validates if the continents exist in the map.
 *
 * @return True if the continents exist, false otherwise.
 */

private boolean validateContinentsExist() {
    if (d_continents.isEmpty()) {
        System.err.println("The map does not contain any continents.");
        printMapRules();
        return false;
    }
    return true;
}

/**
 * Validates if the countries exist in the map.
 *
 * @return True if the countries exist, false otherwise.
 */

public boolean validateCountriesExist() {
    if (d_countries.isEmpty()) {
        System.err.println("The map does not contain any countries.");
        printMapRules();
        return false;
    }
    return true;
}



/**
 * Validates if the countries have connections.
 *
 * @return True if the countries have connections, false otherwise.
 */

public boolean validateCountryConnections() {
    for (Country country : d_countries.values()) {
        if (country.getAdjacentCountries().isEmpty()) {
            System.err.println("The country '" + country.getCountryId() + "' does not have any connections.");
            printMapRules();
            return false;
        }
    }
    return true;
}

/**
 * Validates the graph connectivity.
 *
 * @return True if the graph is connected, false otherwise.
 */
```

```java
private boolean validateGraphConnectivity() {
    if (!GameGraphUtils.isGraphConnected(d_countries)) {
        System.err.println("The map is a disconnected graph.");
        printMapRules();
        return false;
    }
    return true;
}

/**
 * Validates the graph for self loops.
 *
 * @return True if the graph does not have self loops, false otherwise.
 */

private boolean validateGraphSelfLoops() {
    if (GameGraphUtils.hasSelfLoop(d_countries)) {
        System.err.println("The map contains self-loops.");
        printMapRules();
        return false;
    }
    return true;

}

/**
 * Validates the continent connectivity.
 *
 * @return True if the continent is connected, false otherwise.
 */

private boolean validateContinentConnectivity() {
    for (Continent continent : d_continents.values()) {
        if (!GameGraphUtils.isContinentConnected(d_countries, continent.getContinentId())) {
            System.err.println("The continent '" + continent.getContinentName() + "' is a disconnected subgraph.");
            printMapRules();
            return false;
        }
    }
    return true;
}



/**
 * Validates the map by checking if it is a connected graph and if each continent is a connected subgraph.
 *
 * @return True if the map is valid, false otherwise.
 */

public boolean validateMap() {

   return validateContinentsExist() && validateCountriesExist() && validateCountryConnections() &&
         validateGraphConnectivity() && validateGraphSelfLoops() && validateContinentConnectivity();
}
```

- **Change Issue Order class to Issue Order Phase**
    - To avoid confusion and increase readability we added Phase to the name of the class like the other Phases.

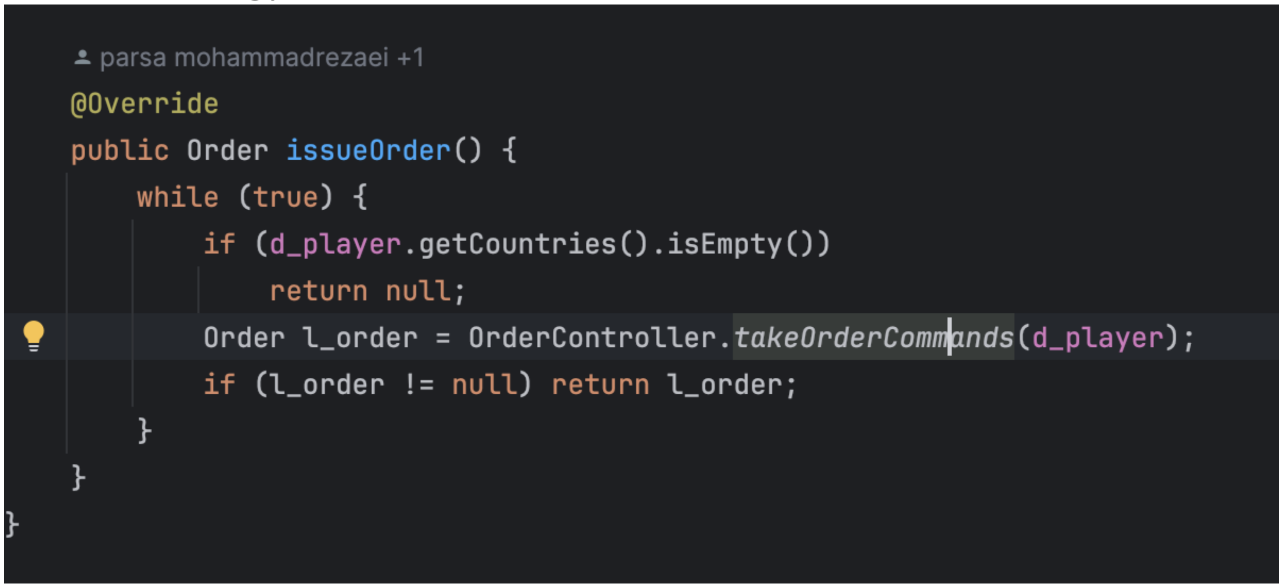


```
phases
    EditMapPhase
    ExecuteDeployOrderPhase
    ExecuteOrdersPhase
    FinishPhase
    GameEngine
    IssueDeployOrder
    IssueOrdersPhase
    Phase
    StartupPhase
    TournamentStartupPhase
```



```
phases
    EditMapPhase
    ExecuteDeployOrderPhase
    ExecuteOrdersPhase
    FinishPhase
    GameEngine
    IssueDeployOrderPhase
    IssueOrdersPhase
    Phase
    StartupPhase
    TournamentStartupPhase
```

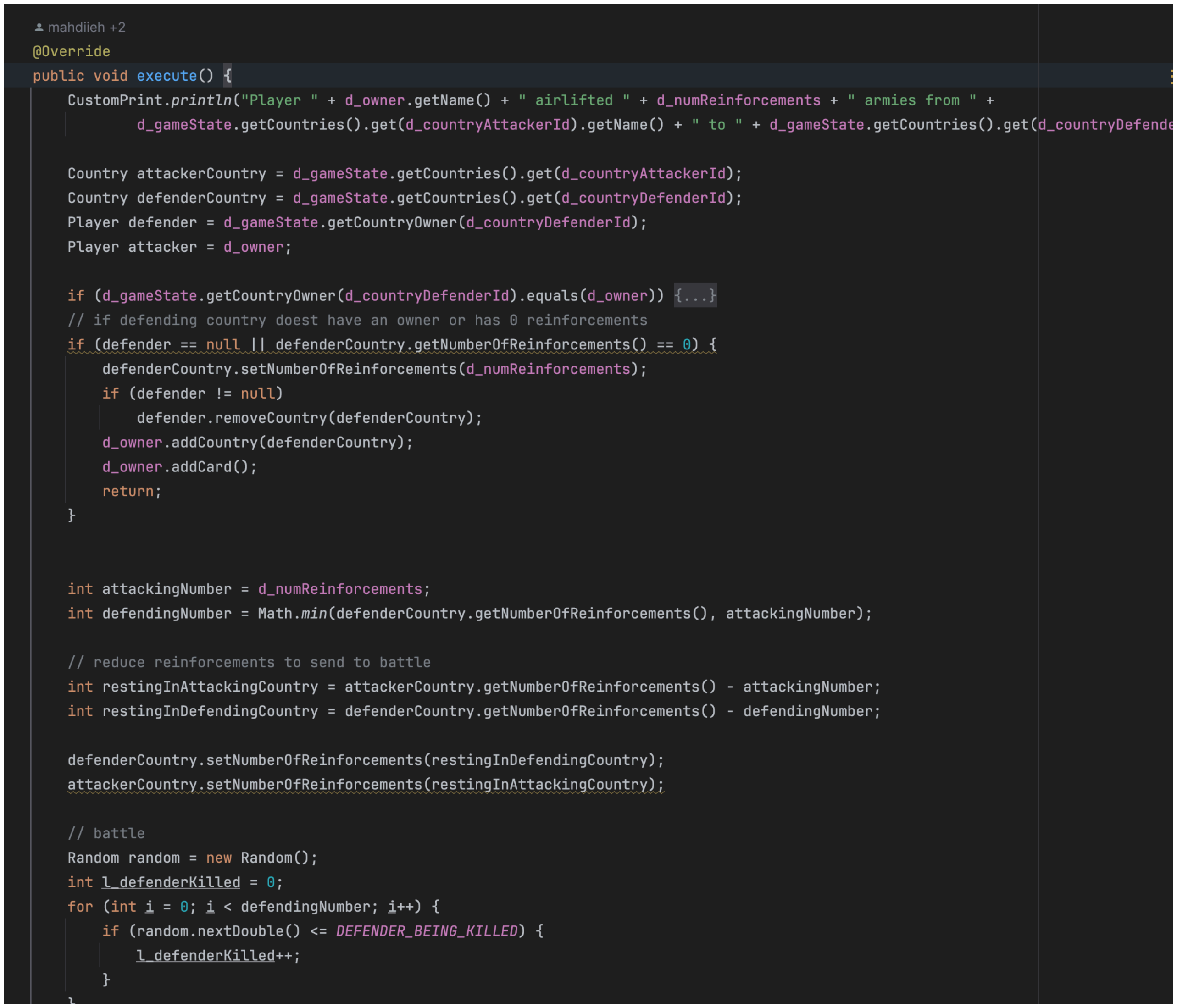
- **Refactor getting user Orders**
    - adding strategy behavior, caused a need for refactoring in getting user input. The code was moved to HumanStra tegyBehaivior.



```
    parsa mohammadrezaei +1
    @Override
    public Order issueOrder() {
        while (true) {
            if (d_player.getCountries().isEmpty())
                return null;
            Order l_order = OrderController.takeOrderCommands(d_player);
            if (l_order != null) return l_order;
        }
    }
}
```

- **Refactor AirliftOrder to use the same logic as AdvanceOrder**

- Different logic and coding style in two Orders with very similar functionalities would cause confiusion and problems while debugging



```
mahdiieh +2
@Override
public void execute() {
    CustomPrint.println("Player " + d_owner.getName() + " airlifted " + d_numReinforcements + " armies from " +
            d_gameState.getCountries().get(d_countryAttackerId).getName() + " to " + d_gameState.getCountries().get(d_countryDefende

    Country attackerCountry = d_gameState.getCountries().get(d_countryAttackerId);
    Country defenderCountry = d_gameState.getCountries().get(d_countryDefenderId);
    Player defender = d_gameState.getCountryOwner(d_countryDefenderId);
    Player attacker = d_owner;

    if (d_gameState.getCountryOwner(d_countryDefenderId).equals(d_owner)) {...}
    // if defending country doest have an owner or has 0 reinforcements
    if (defender == null || defenderCountry.getNumberOfReinforcements() == 0) {
        defenderCountry.setNumberOfReinforcements(d_numReinforcements);
        if (defender != null)
            defender.removeCountry(defenderCountry);
        d_owner.addCountry(defenderCountry);
        d_owner.addCard();
        return;
    }


    int attackingNumber = d_numReinforcements;
    int defendingNumber = Math.min(defenderCountry.getNumberOfReinforcements(), attackingNumber);

    // reduce reinforcements to send to battle
    int restingInAttackingCountry = attackerCountry.getNumberOfReinforcements() - attackingNumber;
    int restingInDefendingCountry = defenderCountry.getNumberOfReinforcements() - defendingNumber;

    defenderCountry.setNumberOfReinforcements(restingInDefendingCountry);
    attackerCountry.setNumberOfReinforcements(restingInAttackingCountry);

    // battle
    Random random = new Random();
    int l_defenderKilled = 0;
    for (int i = 0; i < defendingNumber; i++) {
        if (random.nextDouble() <= DEFENDER_BEING_KILLED) {
            l_defenderKilled++;
        }
```

List of other 14 refactoring targets:

1. **Consistent Variable Naming:**
   Some parts of code has d_continentId and some parts has d_id, both pointing to the id of the object.

2. **String Concatenation:**
   In the Continent class, in the second constructor, the line this.d_name = "" + p_continentId; uses string concatenation to convert the p_continentId to a string. It would be cleaner to use String.valueOf(p_continentId) instead.

3. **Use Constructor Chaining:**
   The constructors can be simplified and made more concise by using constructor chaining.
   For example, in the Country class, the first and second constructors can call the third constructor, passing the necessary arguments. This avoids code duplication and makes the code more maintainable.
   For example, in the Game state, instead of explicitly initializing the member variables, the constructor can call another constructor, passing the necessary arguments.

For example, the default constructor Player() is currently empty. It can be removed since it doesn't serve any purpose. Alternatively, if we want to keep it for some reason, we can use constructor chaining to call the parameterized constructor with default values.

4. **Remove Unused Import**:
   The import statement import models.orders.Order is not used in the code in Player class and can be removed.
   It seems that the handleValidateMapCommand method is not used in the provided code. If it's not needed, consider removing it to reduce code clutter.
   The config.Debug import statement is not used in the provided code and can be safely removed.
   The code imports a few classes that are not used (controllers.*, models.orders.Order). We can remove these unused imports to keep the code clean.

5. **Improve getCountryIds():**
   In the Player class, the getCountryIds() method can be simplified using Java Streams. Instead of manually iterating over the countries and adding their IDs to a list, we can use the map() function to extract the IDs and collect them into an ArrayList.

6. **Grouping Options:**
   The options -add and -remove could be grouped together in a separate class or nested class to provide more organization and structure. For example, we could define an Option class with constants ADD and REMOVE inside the Command class.

7. **Use Exceptions for Error Handling:**
   Instead of printing error messages directly to the console, we can use exceptions to handle and propagate errors. This allows for better error handling and can make the code more robust. Create custom exception classes for different types of errors and throw them when necessary.

8. **Separate Input/Output from Business Logic:**
   The CountryController class currently handles both the business logic and input/output operations. It's generally better to separate these concerns by creating separate classes for input/output and delegating the appropriate tasks to them. This improves the code's modularity and testability.

9. **Encapsulate Data:**
   Instead of directly accessing and modifying the game state and other data structures from outside the class, consider encapsulating the data and providing appropriate getter and setter methods. This helps in maintaining data integrity and allows for better control over data access.

10. **Apply the Single Responsibility Principle (SRP):**
    The CountryController class currently handles multiple responsibilities, such as assigning countries, editing neighbors, editing countries, and editing continents. Consider refactoring the class into smaller, more specialized classes, each responsible for a single task. This improves code readability, maintainability, and testability.

11. **Use Try-With-Resources:**
    In the saveMap() method, use try-with-resources to automatically close the BufferedWriter and handle any exceptions that may occur during writing. This ensures that the writer is always closed, even in the case of an exception.
12. **Separate Concerns:**
    For example, the MapController class currently handles both file I/O and game map validation. It's generally better to separate these concerns by creating a separate class for file I/O operations and delegating the appropriate tasks to it. This improves the code's modularity and testability.
    For example, the takeOrderCommands method is responsible for both parsing user input and handling different order types. Consider splitting these responsibilities into separate methods or classes to improve code readability and maintainability.
    We can split the mainGameLoop method into smaller methods or classes to separate concerns and improve readability.
    The handleGamePlayerCommand method is responsible for both parsing user input and modifying the game state. We can split these responsibilities into separate methods or classes to improve code readability and maintainability.

13. **Consider Dependency Injection:**
    Instead of creating an instance of GameMapReader within the MapController class, consider passing it as a dependency to the constructor. This allows for better decoupling and easier testing, as the dependencies can be mocked or replaced with alternative implementations.

14. **Simplify conditional logic:**
    The code currently has nested if-else statements to handle different commands. Consider using a switch statement for better readability and maintainability.