ADVANCED PROGRAMMING PRACTICES

SOEN 6441 | Build 2

Code Optimization

Submitted By: Group 6

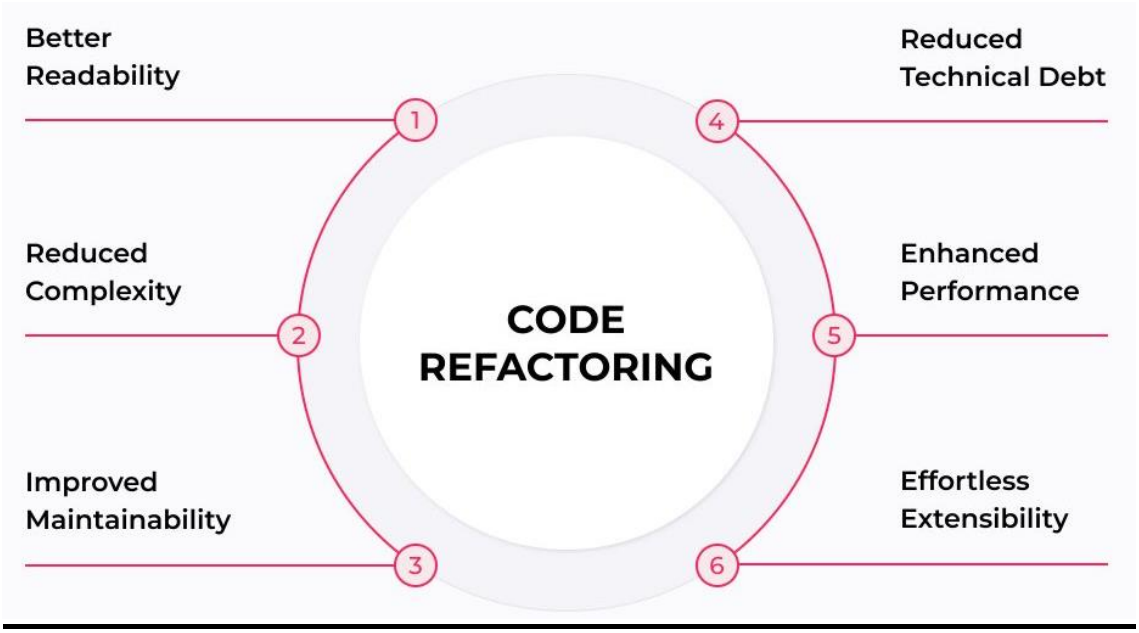
Introduction

Code optimization is a crucial practice in software engineering aimed at refining the internal architecture of software without altering its external functionality. It's advisable to engage in code optimization practices after each development phase to ensure the codebase remains clean and efficient. Recognizing the potential for improvement, we embarked on a systematic journey to refine our code, ensuring it not only meets its functional objectives but does so with elegance and efficiency.

Through collaborative code analysis sessions, we dissected our codebase, scrutinizing it for common pitfalls such as redundancy, lack of coherence, and deviation from best practices. Parallelly, we employed static code analysis tools to uncover hidden issues and ensure no stone was left unturned.

The identified optimization targets spanned a wide range of aspects, from enhancing code readability through better naming conventions and modularization to improving runtime efficiency by optimizing data structures and algorithms. Each target was carefully chosen for its potential to make the code more intuitive, maintainable, and robust, thereby laying a solid foundation for the application's future evolution.

This report outlines the strategy and actions taken by Team 06 during the second build of our project to optimize our code.



Identification of Optimization Opportunities

To ensure our codebase remains robust and efficient, we embarked on a comprehensive review, identifying several key areas where optimization could significantly enhance performance and maintainability:

Collaborative Code Analysis:

Through team reviews, we pinpointed sections of code plagued by inefficiencies such as redundant logic, excessively lengthy methods, and complex code constructs that hindered readability and maintenance.

Redundant Code Elimination:

We identified and removed superfluous variables and methods, which cluttered the codebase without serving a functional purpose.

Code Style Uniformity Check:

We meticulously checked for consistent application of coding standards and conventions throughout the codebase, rectifying instances of inconsistent naming conventions to prevent confusion among developers.

Data Structure Analysis:

We scrutinized our use of concrete data structures like ArrayList and HashMap, considering the switch to more abstract types like List and Map to increase the code's flexibility and adherence to object-oriented design principles.

Logic Encapsulation:

To foster code maintainability and lessen interdependencies, we encapsulated validation and business logic within their respective classes, reducing the sprawl of tangled logic across the codebase.

Performance Bottlenecks:

By profiling the application, we identified performance bottlenecks, particularly in areas involving intensive computations or data processing, which could be mitigated through algorithm optimization or parallel processing.

Code Smell Identification:

Using static analysis tools, we detected various code smells such as "God Objects," "Feature Envy," and "Spaghetti Code," indicating sections of the code that required refactoring for better modularity and coherence.

Optimization Targets Identified

Through our analysis, we identified areas where optimization could significantly benefit code readability and maintainability:

Constant Usage for Literal Values:

Implement constants for frequently used literal values across the code, ensuring consistency and ease of updates.

Function Extraction for Complex Logic:

Identify sections with intricate logic and extract them into separate, well-named functions to enhance readability and testability.

Main Procedure Decomposition:

Break down the central processing loop into smaller, manageable classes or methods, facilitating easier understanding and modularization.

Embedded Validation in Domain Classes:

Embed validation logic directly within domain-specific classes to promote encapsulation and reduce external dependencies.

Dynamic Typing with Interface References:

Replace concrete class references in variables, method signatures, and return types with their corresponding interfaces to increase flexibility and reduce coupling.

Method Simplification:

Simplify complex methods by breaking them into smaller methods, each performing a single, focused task, thereby adhering to the Single Responsibility Principle.

Consistent Naming Scheme:

Ensure all methods, variables, and classes follow a consistent naming scheme that reflects their purpose and scope, improving code clarity.

Iterative Constructs Modernization:

Update traditional loop constructs to modern, more readable equivalents (like enhanced for-loops or streams in Java) where applicable.

Redundant Code Removal:

Eliminate unnecessary methods and variables that do not contribute to the application's functionality to streamline the codebase.

Optimizing Data Structures Usage:

Review and optimize the use of data structures for efficiency, choosing the most appropriate type for each use case to improve performance.

Error Handling Consolidation:

Standardize error handling across the application to ensure a consistent strategy for managing exceptions and errors, improving robustness.

Dependency Reduction:

Reduce external dependencies within classes by applying Dependency Inversion and Injection principles, facilitating easier testing and maintenance.

Comment and Documentation Update:

Revise and update comments and documentation to accurately reflect the current state of the codebase, removing outdated information.

Concurrency and Parallelism:

Investigate opportunities to apply concurrency or parallelism in processing tasks to improve performance, particularly in computationally intensive operations.

Code Smell Resolution:

Address and resolve identified code smells, such as "Large Class," "Long Method," and "Primitive Obsession," through appropriate refactoring techniques to enhance code quality.

Refactoring for Testability:

Improve the structure of the code to make it more amenable to automated testing, ensuring that each component can be tested in isolation.

Documentation and Code Comments Refresh:

Update and improve the internal documentation and code comments to reflect the current state of the system and facilitate easier maintenance.

Optimizations Implemented

Following the identification of optimization targets, we implemented the following refactoring's:

1. Optimization of Map Loading Logic:

Before Refactoring: The map loading logic initially conducted a file existence check deep within nested conditionals. This not only cluttered the method but also intertwined the map validation logic with file existence checks, leading to a less clear flow of execution and a mix of concerns within the method.

```
public GameMap loadMap(String p_mapName) {

    String l_filePath = "src/main/resources/maps/" + p_mapName;
    GameMap l_gameMap;
    File l_file = new File(l_filePath);
    if(l_file.exists())
    {
        LoadMap l_loadMap = new LoadMap();
        l_gameMap = l_loadMap.readMap(l_filePath);
        l_gameMap.d_MapName = p_mapName;
        if(validateMap(l_gameMap)) {
            l_gameMap.d_Valid = true;
        }
        else {
            System.out.println(x:"The map is inappropriate for gameplay. To proceed, make necessary changes to the current map or sel
            l_gameMap.d_Valid = false;
        }
    }else {
        System.out.println("There is no such map named " + p_mapName + " . Try loading again, or make a map using 'editMap'.\n");
        return null;
    }
    return l_gameMap;
```

After Refactoring: The refactored code now performs the file existence check at the very beginning of the map loading process. By immediately returning null if the map file does not exist, the method avoids deeper nesting of conditionals and simplifies the logic flow. This early return pattern not only clarifies the method's primary execution path but also enhances its readability and maintainability.

```
public GameMap loadMap(String p_mapName) {
    String l_filePath = "src/main/resources/maps/" + p_mapName;
    File l_file = new File(l_filePath);
    if (!l_file.exists()) {
        System.out.println("There is no such map named " + p_mapName + ". Try loading again, or make a map using 'editMap'.\n");
        return null;
    }
    LoadMap l_loadMap = new LoadMap();
    GameMap l_gameMap = l_loadMap.readMap(l_filePath);
    l_gameMap.d_MapName = p_mapName;
    if (validateMap(l_gameMap)) {
        l_gameMap.d_Valid = true;
    } else {
        System.out.println(x:"The map is inappropriate for gameplay. To proceed, make necessary changes to the current map or select
        l_gameMap.d_Valid = false;
    }
    return l_gameMap;
}
```

Implemented Changes: We introduced a guard clause to check for the file's existence, which exits early if the map file is not found. This practice adheres to the fail-fast philosophy and reduces the cognitive complexity for future developers reading the code.

The map reading and validation logic were decoupled from the file existence check, resulting in a cleaner separation of concerns. Now, the readMap method is only called when we are sure the file exists, focusing this method on its core responsibility.

Impact: This optimization reduces the depth of conditional logic, thus simplifying the understanding of the code.

It improves the maintainability by ensuring that each part of the method has a single, clear purpose.

The error message provided to the user is now more immediate and context-specific, potentially reducing user confusion and improving the user experience.

2. Refinement of Map Loading and Creation Logic:

Before Refactoring:

The initial implementation for loading and editing maps involved checking the existence of the map file directly within the editMap method. This led to a somewhat cluttered approach, with map validation and creation logic intermingled, making the method responsible for multiple actions.

```
/**
 * loading the map for playing and also create the new map if it does not present
 * @param p_mapName Name of the map to be created
 * @return  l_gameMap represents the existing map
 */
public GameMap editMap(String p_mapName) {

    String l_filePath = "src/main/resources/maps/" + p_mapName;
    GameMap l_gameMap;
    File l_file = new File(l_filePath);
    if(l_file.exists()) {
        System.out.println(" Map " + p_mapName + " exist and it can also be editted.");
        LoadMap l_loadMap = new LoadMap();
        l_gameMap = l_loadMap.readMap(l_filePath);
        l_gameMap.d_MapName = p_mapName;
    }
    else {
        System.out.println(p_mapName + " does not exist.");
        System.out.println("New Map created named " + p_mapName);
        l_gameMap = new GameMap(p_mapName);
    }
    return l_gameMap;
}
```

Optimization Steps:

We optimized the map loading process by abstracting the logic into a dedicated method called loadExistingMap. This refactoring aimed to segregate responsibilities, thereby adhering to the Single Responsibility Principle, which states that a function should have only one reason to change.

After Refactoring:

Post-optimization, the editMap method became more concise, offloading the task of reading and validating an existing map to the new loadExistingMap function. This not only made the editMap method more readable but also modularized the code for better maintainability. The method handleInvalidMap was introduced to deal specifically with scenarios where map validation fails, encapsulating the error-handling logic.

```
private GameMap loadExistingMap(String filePath, String mapName) {
    LoadMap l_loadMap = new LoadMap();
    GameMap l_gameMap = l_loadMap.readMap(filePath);
    l_gameMap.d_MapName = mapName;
    if (!validateMap(l_gameMap)) {
        handleInvalidMap();
    }
    return l_gameMap;
}

private void handleInvalidMap() {
    System.out.println(x:"The map is inappropriate for gameplay. To proceed, make necessary changes to the current map or select an a
}
```

```
/**
 * loading the map for playing and also create the new map if it does not present
 * @param p_mapName Name of the map to be created
 * @return  l_gameMap represents the existing map
 */
public GameMap editMap(String p_mapName) {

    String l_filePath = "src/main/resources/maps/" + p_mapName;
    GameMap l_gameMap;
    File l_file = new File(l_filePath);
    if(l_file.exists()) {
        System.out.println(" Map " + p_mapName + " exist and it can also be editted.");
        l_gameMap = loadExistingMap(l_filePath, p_mapName);

    }
    else {
        System.out.println(p_mapName + " does not exist.");
        System.out.println("New Map created named " + p_mapName);
        l_gameMap = new GameMap(p_mapName);
    }
    return l_gameMap;
```

Outcome:

As a result of these changes, we achieved a cleaner separation of concerns within our codebase. The refactoring enhanced the clarity of the map loading and creation flow, making it easier for developers to navigate and extend the code. It further isolated the error handling into a dedicated method, improving the ability to manage and modify error messages and the corresponding behaviour.

By extracting specific functionalities into their own methods, we made the codebase more organized. This refactors not only improved readability but also simplified future enhancements and debugging. Moreover, these improvements to the map handling logic have laid the groundwork for more advanced features, such as dynamic map validation and automated map repair functionalities, which can now be integrated with minimal impact on the existing code structure.

3. Refactoring Territory Management:

One significant improvement in our codebase was the optimization of the territory management logic, specifically the process of removing neighbouring territories.

Before Refactoring:

The initial implementation employed an iterator pattern to traverse a list of Territory Details. This approach, while functional, resulted in verbose and less readable code. It also required the manual management of the iterator and a separate list of territories, which increased the risk of errors during iteration and removal operations.

```
private boolean removeNeighbourTerritory(GameMap p_map, ArrayList<TerritoryDetails> p_tList, TerritoryDetails p_territory){
    Iterator<TerritoryDetails> l_iterator = p_tList.listIterator();
    while(l_iterator.hasNext()) {
        TerritoryDetails l_neighbour = l_iterator.next();
        if(!removeNeighbour(p_map, p_territory.getTerritoryID(), l_neighbour.getTerritoryID()))
            return false;
    }
    return true;
}
```

After Refactoring:

We refactored this method to utilize the enhanced for-loop construct, streamlining the code and enhancing readability. The new approach eliminates the need for an explicit iterator and list, thus reducing boilerplate code and potential for errors. Additionally, it simplifies the logic by directly iterating over the values of a map, leading to a more concise and intention-revealing implementation.

```
private boolean removeNeighbourTerritory(GameMap p_map, TerritoryDetails p_territory) {
    for (TerritoryDetails l_neighbour : p_territory.getNeighbours().values()) {
        if (!removeNeighbour(p_map, p_territory.getTerritoryID(), l_neighbour.getTerritoryID())) {
            return false;
        }
    }
    return true;
}
```

This refactoring not only enhanced the maintainability of our code but also aligned it with modern Java practices, leveraging the strengths of the Collections framework. It is a testament to our commitment to producing code that is not only efficient but also clean and elegant.

4. Refinement of Neighbour Association Logic:

Before Refactoring:

The original implementation of the addNeighbour method involved repetitive and verbose checks for territory existence and neighbour association within the GameMap class. The code was prone to errors due to repeated calls to toLowerCase() and lacked a clear separation of concerns, leading to a method that was challenging to read and maintain.

```
public boolean addNeighbour(GameMap p_map, String p_territoryID, String p_neighbourTerritoryID) {
    if(p_map.getTerritories().containsKey(p_territoryID.toLowerCase()) && p_map.getTerritories().containsKey(p_neighbour
        TerritoryDetails l_c1 = p_map.getTerritories().get(p_territoryID.toLowerCase());
        TerritoryDetails l_c2 = p_map.getTerritories().get(p_neighbourTerritoryID.toLowerCase());

        if(!l_c1.getNeighbours().containsKey(l_c2.getTerritoryID().toLowerCase())){
            l_c1.getNeighbours().put(p_neighbourTerritoryID.toLowerCase(), l_c2);
            System.out.println(p_territoryID+" added as neighbour to "+p_neighbourTerritoryID);
        }
        else
            System.out.println(x:"Already Neighbour");

        if(!l_c2.getNeighbours().containsKey(l_c1.getTerritoryID().toLowerCase())){
            l_c2.getNeighbours().put(p_territoryID.toLowerCase(), l_c1);
            System.out.println(p_neighbourTerritoryID+" added as neighbour to "+p_territoryID);
        }

        else
            System.out.println(x:"Already Neighbour");

            return true;
    }
    else {
        if(!p_map.getTerritories().containsKey(p_territoryID.toLowerCase()) && !p_map.getTerritories().containsKey(p_nei
            System.out.println(p_territoryID + " and " + p_neighbourTerritoryID + "  does not exist. Set their neighbors
        else if(!p_map.getTerritories().containsKey(p_territoryID.toLowerCase()))
            System.out.println(p_territoryID + " does not exist. Set their neighbors after creating their own territory.
        else
            System.out.println(p_neighbourTerritoryID + " does not exist. Set their neighbors after creating their own t
        return false;
    }
}
```

After Refactoring:

We have optimized the addNeighbour method to enhance the clarity and efficiency of our codebase significantly.

```java
public boolean addNeighbour(GameMap p_map, String p_territoryID, String p_neighbourTerritoryID) {
    if (!territoriesExist(p_map, p_territoryID, p_neighbourTerritoryID)) {
        return false;
    }

    TerritoryDetails l_c1 = p_map.getTerritories().get(p_territoryID.toLowerCase());
    TerritoryDetails l_c2 = p_map.getTerritories().get(p_neighbourTerritoryID.toLowerCase());

    if (!l_c1.getNeighbours().containsKey(p_neighbourTerritoryID.toLowerCase())) {
        l_c1.getNeighbours().put(p_neighbourTerritoryID.toLowerCase(), l_c2);
        System.out.println(p_territoryID + " added as neighbour to " + p_neighbourTerritoryID);
    } else {
        System.out.println(x:"Already Neighbour");
    }

    if (!l_c2.getNeighbours().containsKey(p_territoryID.toLowerCase())) {
        l_c2.getNeighbours().put(p_territoryID.toLowerCase(), l_c1);
        System.out.println(p_neighbourTerritoryID + " added as neighbour to " + p_territoryID);
    } else {
        System.out.println(x:"Already Neighbour");
    }

    return true;
}

private boolean territoriesExist(GameMap p_map, String p_territoryID, String p_neighbourTerritoryID) {
    if (!p_map.getTerritories().containsKey(p_territoryID.toLowerCase()) && !p_map.getTerritories().containsKey(p_neighbourTerritoryI
        System.out.println(p_territoryID + " and " + p_neighbourTerritoryID + " do not exist. Set their neighbors after creating thei
        return false;
    } else if (!p_map.getTerritories().containsKey(p_territoryID.toLowerCase())) {
        System.out.println(p_territoryID + " does not exist. Set their neighbors after creating their own territory.\n");
        return false;
    } else if (!p_map.getTerritories().containsKey(p_neighbourTerritoryID.toLowerCase())) {
        System.out.println(p_neighbourTerritoryID + " does not exist. Set their neighbors after creating their own territory.\n");
        return false;
```

Encapsulation of Existence Checks:

We introduced the territoriesExist method, which consolidates the checks for the presence of territories within the map. By moving this logic into a separate, dedicated method, we have clarified the main flow of addNeighbour, making it easier to understand and reducing the duplication of existence checks.

Streamlining Logic Flow:

The refactored code now follows a clearer logical progression. It first validates the existence of both territories. If either territory does not exist, it promptly exits, thereby avoiding unnecessary processing. This change improves the method's performance by reducing the number of conditional checks performed in scenarios where one or both territories are missing.

Enhancing Code Readability:

The print statements within the conditional blocks have been cleaned up to provide more precise feedback to the user. By removing excess code and organizing the structure around the core functionality of adding neighbours, we have made the method more readable.

Preventing Case Sensitivity Errors:

By ensuring all territory IDs are treated in a case-insensitive manner consistently, we have reduced the likelihood of case sensitivity bugs. This aligns with the principle of defensive programming, where the code anticipates and safely handles potential user input variations.

Optimizing Data Structure Manipulation:

The neighbour addition process has been optimized by ensuring that the getNeighbours() method is called only once per territory, thereby minimizing lookups in the underlying data structure and improving the method's performance.

The result of these optimizations is a more maintainable and efficient method for managing the relationships between territories on the game map. The addNeighbour method now serves as a model of clean code principles applied to a critical part of the game's functionality—ensuring that the territories' relationships are accurately represented and easily modifiable, which is vital for game logic and performance.