

# ADVANCED PROGRAMMING PRACTICES

SOEN 6441 | Build 3

**Code Optimization**

Submitted By: Group 6

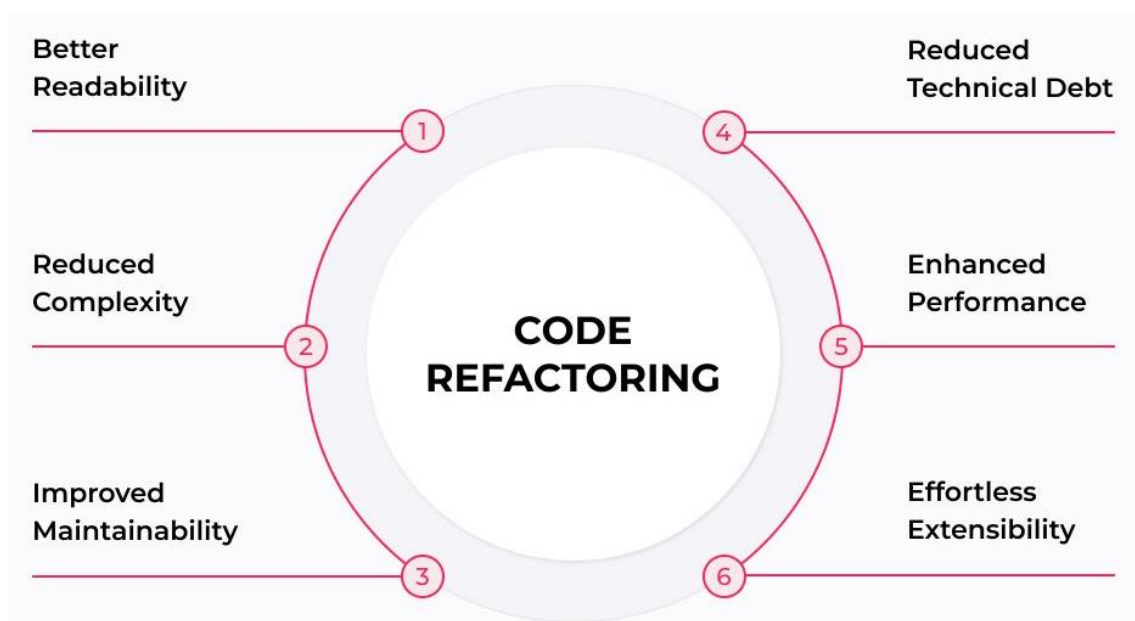
## Introduction

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. Parallely, 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.

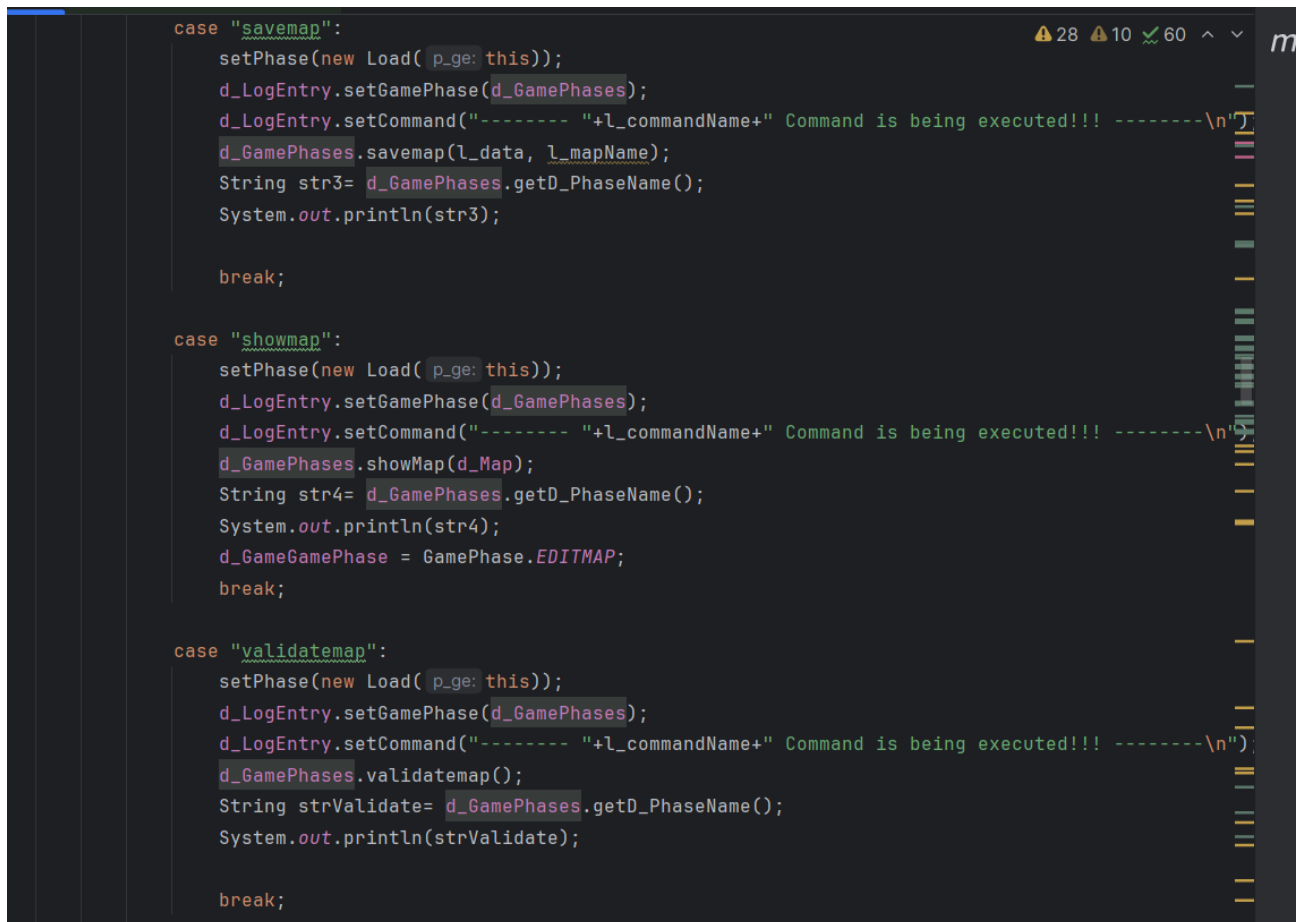
### **Documentation and Code Comments Refresh:**

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

## Optimizations Implemented

After careful consideration of the above the following 5 refactoring's were implemented:

- 1. Make methods for repeated/long code:** This is done to ensure code reusability and increase readability. The example for this is shown in the pictures below.



```
case "savemap":
    setPhase(new Load( p_ge: this));
    d_LogEntry.setGamePhase(d_GamePhases);
    d_LogEntry.setCommand("----- "+l_commandName+" Command is being executed!!! -----\\n");
    d_GamePhases.savemap(l_data, l_mapName);
    String str3= d_GamePhases.getD_PhaseName();
    System.out.println(str3);

    break;

case "showmap":
    setPhase(new Load( p_ge: this));
    d_LogEntry.setGamePhase(d_GamePhases);
    d_LogEntry.setCommand("----- "+l_commandName+" Command is being executed!!! -----\\n");
    d_GamePhases.showMap(d_Map);
    String str4= d_GamePhases.getD_PhaseName();
    System.out.println(str4);
    d_GameGamePhase = GamePhase.EDITMAP;
    break;

case "validatemap":
    setPhase(new Load( p_ge: this));
    d_LogEntry.setGamePhase(d_GamePhases);
    d_LogEntry.setCommand("----- "+l_commandName+" Command is being executed!!! -----\\n");
    d_GamePhases.validatemap();
    String strValidate= d_GamePhases.getD_PhaseName();
    System.out.println(strValidate);

    break;
```

- 2. Introduce Logging Statements:** Instead of relying on print statements for debugging, logging statements were introduced for logging in the outputs for different runs.

Before Example: there was no logic for logging, we were using just print statements for debugging.

After Example:

```

11 public class WriteLogEntry implements Observer {
12     String d_fileName="log.txt";
13     private static String d_bufferData;
14     private FileWriter d_fileWriter;
15     private BufferedWriter d_bufferedWriter;
16
17
18     public WriteLogEntry() {
19
20         try{
21             d_fileWriter = new FileWriter(d_fileName);
22             d_bufferedWriter = new BufferedWriter(d_fileWriter);
23         } catch(IOException e) {
24             e.printStackTrace();
25         }
26     }
27
28
29     @Override
30     public void update(Observable o) {
31
32         LogBuffer LogBuff = (LogBuffer)o;

```

**3. Implement Strategy Pattern:** To introduce different types of Players, a Strategy pattern was implemented.

Before Example: There was only 1 player class for human players as shown below which has an `issue_order()` method.

After Example: Strategy pattern was introduced and before all the functions of this class the word 'Human' was added for clarity.

```

3 public abstract class PlayerStrategy {
4     GameMap d_Map;
5     Player d_Player;
6     protected PlayerStrategy(Player p_player, GameMap p_map){
7         d_Player = p_player;
8         d_Map = p_map;
9     }
10    public abstract order1 createOrder();
11    protected abstract TerritoryDetails toAttackFrom();
12    protected abstract TerritoryDetails toAttack();
13    protected abstract TerritoryDetails toMoveFrom();
14 }

```

**4. Implemented Adapter Pattern:** An adapter pattern was introduced to enable the application to read/write from/to a file.

Before Example: The game could work only with Domination Maps

After Example: ConquestMapIOAdapter was introduced so that the game could work with both Domination and Conquest maps

```
1 Hetul +  
> public class MapAdapter extends DominationMap{  
  
    3 usages  
    ConquestMap d_ConquestMap;  
  
    /**  
     * constructor  
     * @param p_conquestMap conquest map  
     */  
    2 usages 1 Hetul  
> public MapAdapter(ConquestMap p_conquestMap) { this.d_ConquestMap = p_conquestMap; }  
  
    /**  
     * method to read domination map  
     * @return game map  
     */  
    2 usages 1 Hetul  
> public GameMap readDominationMap(String p_mapName) { return d_ConquestMap.readConquestMap(p_mapName); }  
  
    /**  
     * method to save map  
     * @return true if map is saved successfully  
     */  
    no usages 1 Hetul  
> public boolean saveMap(GameMap p_map, String p_fileName) { return d_ConquestMap.saveMap(p_map, p_fileName); }  
}
```

**5. Review of various code validations/verifications/formatting:** Ensuring that the code after Build 2 could adapt to the various changes introduced in Build 3 like the different Player strategies.

Before Example: There was no check needed because the only player type was human, checks were added as part of introducing various patterns in the code to keep the behaviour the same for the end user.

After Example: example of the changes made to the main Player class (now for human players)

```
cmd.setGamePhase(l_gamePhase);  
while (l_gamePhase != GamePhase.TURN) {  
    if (l_p.getO_isHuman()) {  
        l_cmd = sc.nextLine();  
        l_gamePhase = cmd.parseCommand(l_p, l_cmd);  
    } else {  
        l_gamePhase = cmd.parseCommand(l_p, p_newCommand: "");  
    }  
}  
l_traversalCounter++;  
}  
l_gamePhase = GamePhase.ISSUE_ORDERS;  
cmd.setGamePhase(l_gamePhase);  
l_traversalCounter = 0;  
}  
} else {  
    System.out.println("---- it is not valid command so enter 1 to load game and enter 2 to  
    continue;  
}
```