Farm Defense
Coding Standards and Guidelines

1. Introduction

1.1 Purpose

This document outlines the coding standards to be followed in this software to ensure consistency, readability, and maintainability of the codebase.

1.2 Scope

These coding standards apply to the Farm Defense software and are intended for use by developers and anyone involved in the software development process.

2. Naming Conventions

2.1 Classes and Interfaces

Use CamelCase with the first letter capitalized (e.g., MyClass, MyInterface)

2.2 Methods

Use CamelCase with the first letter lowercase (e.g., calculateTotal())

2.3 Variables

Use meaningful simple names with CamelCase (e.g., employeeCount, totalAmount)

2.4 Constants

Use uppercase with underscores (e.g., `MAX_VALUE`, `PI`)

3. Documentation

3.1 Javadoc

Provide comprehensive Javadoc for classes and methods that are not getters or setters

3.1.1 Method Javadoc template

```
/**
   * Description of method's functionality
   *
   * @param  myParameter  description of myParameters usage
   * @return         what the method is returning
  */
```

3.1.2 Class Javadoc template

```
/**
 * Description of class purpose and functionality
 *
 * @author Your Name
 * @since yyyy-mm-dd
 */
```

3.2 Inline Comments

Use inline comments sparingly and ensure they add value by explaining complex code sections. Use inline comments to explain any non-trivial class attributes.

4. Coding Practices

4.1 Encapsulation

Ensure class attributes(class variables) are only altered by their class. If you want data in a class to be altered by other classes, add getter and setter methods. Class attributes should be private or in some cases protected. Class attributes should not be public

4.2 Adhering to architecture

Be mindful of where you are creating classes and adding certain functionalities. Read over the software architecture document and adhere to the architecture of the project.

4.3 Adding features

When adding new desired features do not immediately start writing code. Plan how the feature will be structured, what classes will be leveraged and what classes need to be created. Be mindful and adhere to the architecture when adding these. If your changes reveal opportunities for refactoring existing code for better clarity or efficiency, consider adding those changes.

4.4 Avoid Code Duplication

Before writing new code, check if there are existing functionalities that can be reused. Avoid duplicating code unnecessarily.

4.5 Use utility classes

Long and repetitive code blocks should be put into utility classes to improve code readability.

5. Testing

5.1 Unit Testing

Write unit tests for all non-trivial methods and classes.