COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING
DEPARTMENT OF SOFTWARE ENGINEERING

Software evolution and maintenance

<u>Group members</u>	<u>ID No</u>
1. Gelila Adugna	ETS0312/12
2. Hara Birhanu	ETS0336/12
3. Kokebe Negalign	ETS0419/12
4. Mastewal Tesfaye	ETS0435/12
5. Hermela Mulugeta	ETS 0353/12

<u>SECTION B</u>

Submitted to Mr. Ashenafi

Submission date May 13,05,2024

Table of Content

Introduction

Code smells are common signs of poor code quality and potential problems in a software system. They can manifest in various forms such as duplicated code, overly complex functions, long methods, and more. Identifying code smells is crucial in order to improve the maintainability, readability, and overall quality of code.

Refactoring is the process of restructuring existing code in order to improve its design without changing its external behavior. By addressing code smells through refactoring, developers can eliminate redundancy, improve readability, and make the codebase more maintainable. Refactoring helps to ensure that the code remains clean, efficient, and adaptable as the software system evolves.

1. Dispensable- lazy class

What is the dispensable class in software evolution and maintenance?

The dispensable class in software evolution and maintenance refers to a type of class that is considered unnecessary or redundant within a software system. This class may contain code that is no longer relevant or serves no useful purpose, making it a candidate for removal or refactoring during software maintenance.

Why is the dispensable class considered a problem in software evolution and maintenance?

The dispensable class can be problematic because it can lead to inefficiencies, increased complexity, and potential errors within the software system. By removing or refactoring such classes, developers can improve code quality, reduce redundancy, and make the system more maintainable over time.

How can developers identify dispensable classes in their software systems?

Developers can identify dispensable classes by analyzing code quality metrics such as coupling, cohesion, and code that smells like dead code or unused variables. They may also review code comments and documentation to understand the purpose of each class and determine whether it is still necessary for the functioning of the software.

What are some strategies for dealing with dispensable classes during software evolution and maintenance?

Some strategies for dealing with dispensable classes include refactoring the code to remove unnecessary logic or dependencies, merging similar classes together to reduce redundancy, or completely removing the class if it serves no useful purpose. It's essential to carefully evaluate each option based on its impact on overall system stability and maintainability.

How does addressing dispensable classes contribute to overall software quality?

By addressing dispensable classes during software evolution and maintenance, developers can improve code quality by reducing redundancy, simplifying logic, and improving overall system maintainability. This results in more efficient development processes and more reliable software systems over time.

An example of how a dispensable class might be identified and refactored in a real-world scenario.

In a real-world scenario involving an e-commerce website built using PHP, developers might identify a dispensable class named "OldPaymentGateway" that contains outdated payment processing logic no longer used by the system since being replaced by a new payment gateway named "NewPaymentGateway." To address this issue, developers could refactor the "OldPaymentGateway" class by removing its outdated functionality and merging its remaining methods into either the "NewPaymentGateway" class or another relevant payment processing utility class.

How does addressing dispensable classes fit into broader strategies for maintaining large-scale software systems?

Addressing dispensable classes fits into broader strategies for maintaining large-scale software systems as part of ongoing efforts to improve code quality.

2. Data Class Code Smell:

A data class is a class that primarily holds data without providing much behavior. It typically contains only properties and simple getter/setter methods. This can lead to an anemic domain model where behavior is spread across multiple classes instead of being encapsulated within the class itself. As a result, data classes can contribute to code duplication and make it harder to maintain and reason about the codebase.

Refactoring Methods:

Encapsulate Field: This refactoring method involves encapsulating the class's fields with getter and setter methods. By encapsulating the fields, you can control access to them and introduce additional logic (such as validation) if needed in the future.

```php
// Before Refactoring
class UserData {
    public $name;
    public $email;
}

// After Refactoring
class UserData {
    private $name;
    private $email;
    public function getName() {
        return $this->name;
    }
    public function setName($name) {
        $this->name = $name;
    }

    public function getEmail() {
        return $this->email;
    }
    public function setEmail($email) {
        $this->email = $email;
    }
}
```

Replace Data Class with Value Object: If the data class represents a conceptually immutable value, it can be replaced with a value object. A value object is an object whose equality is determined by the equality of its values, and once created, its state cannot be changed.

```
// Before Refactoring
class Point {
    public $x;
    public $y;
}
// After Refactoring
class Point {
    private $x;
    private $y;
    public function __construct($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }

    public function getX() {
        return $this->x;
    }
    public function getY() {
        return $this->y;
    }
}
```

In the refactored examples, we've encapsulated the fields of the data classes (UserData and Point) with getter and setter methods, allowing for better control and potential validation of the data. Additionally, we've replaced the Point data class with a value object, ensuring immutability and encapsulating behavior related to the concept of a point in space within the object itself.

These refactoring methods help to improve the design and maintainability of the codebase by reducing the multiplication of data classes and promoting encapsulation of behavior within the appropriate classes.

3. Duplicate code

duplicate code is also called duplicate code smell or duplicated code, is a software development concept where identical or similar code is written in multiple places within a program or across different programs. It is sections of code that appear in more than one place within a software system or application.

It can occur due to

- Copy-paste code: When developers copy and paste code from one location to another, often without modifying it, to reuse it in a similar context.
- Similar but not identical code: When developers write similar code in different places, but with minor variations, to achieve a similar functionality.

Duplicate code can lead to several issues, including:

- · Maintenance and update challenges
- Code duplication:
- Debugging difficulties
- Code quality decline

To avoid code duplication, use refactoring.

Refactoring is the process of restructuring existing code to improve its maintainability, scalability, and efficiency, while preserving its original functionality the most common methods are.

- Extract Method/function: Break down a large method into smaller, more manageable methods. This can help eliminate duplicate code by identifying and extracting reusable code blocks.
- Extract Class/Module: When there is a group of related methods or functions that are duplicated across different classes or modules, wecan extract them into a separate class or module.
- Replace Duplicate Code with Inheritance: Use inheritance to create a base class that contains common functionality, and then extend it to create subclasses that inherit the common behavior.
- Replace Duplicate Code with Composition: Use composition to create objects that contain other objects, allowing you to reuse code without duplicating it.
- Remove Dead Code: Remove code that is no longer used or is unreachable, as it can lead to confusion and maintenance issues.

Example for code duplication

```
//For checking the user is authorized to or not
//In one function use
function isAuthorized(username, resource) {
    if (username === "admin") {
        return true;
    } else if (username === "user") {
        return resource === "resource1" || resource === "resource2";
    } else {
        return false;
    }
}


//duplicate code in another function
function isAuthorized2(username, resource) {
    if (username === "admin") {
        return true;
    } else if (username === "user") {
        return resource === "resource1" || resource === "resource2";
    } else {
        return false;
    }
}
//But instead of use the same code multiple times use code refactoring method of
 function authorize (username, resource) {
    if (username === "admin") {
        return true;
    } else if (username === "user") {
        return resource === "resource1" || resource === "resource2";
    } else {
        return false;
    }
}


function isAuthorized(username, resource) {
    return authorize (username, resource);
}
```

3. Dead code

Dead code is code that is never executed or has no effect. It is also known as "dead code" or "unreachable code." It includes variables, functions, or entire blocks of code that have become redundant due to changes in requirements, logic, or refactoring.

- Unused variables

Eg let un=”hello”   if the un is declared and not used in the program then it is dead code.

The variable is not contributing to the program's functionality.

The memory allocated for the variable is wasted.

- Unreachable code

Example: if (false) {console.log('This will never be executed'); }

- Unused functions
- Unused modules
- Commented codes.

Impacts of dead code on the program

- ·Code Complexity:
- Increased Compile Time:
- Reduced Code Readability:
- Memory Waste

4. Speculative generality

Speculative generality refers to the tendency to over-engineer code by adding functionality that is not currently needed but might be needed in the future. This results in unnecessary complexity and can make the code harder to understand, maintain, and extend.

Refactoring methods to resolve speculative generality typically involve simplifying the code by removing unused or unnecessary abstractions, classes, methods, or other elements. Here are some common refactoring techniques to address speculative generality:

4.1. Remove Unused Code: Identify and remove any code that is not currently being used in the application. This includes unused classes, methods, variables, and imports.

For example:

```
// Before refactoring
                public class UnusedClass {
                public void unusedMethod() {
                        // Unused method implementation
                          }
          // After refactoring
                      // UnusedClass and unusedMethod are removed
```

4.2. Inline Methods: If a method is only called in one place and does not provide any additional clarity or abstraction, consider removing the method and placing its logic directly in the calling code.

For example:

```
// Before refactoring
                public void doSomething() {
                helperMethod();
```

```
                    }
             private void helperMethod() {
              // Helper method implementation
                     }
          // After refactoring
              // Inline helperMethod into doSomething
                public void doSomething() {
                 // Helper method implementation directly here
                       }
```

4.3. Simplify Class Hierarchy: If there are unnecessary inheritance hierarchies or abstract classes that do not serve a clear purpose, simplify or remove them.

For example:

```
// Before refactoring
                 public abstract class MyBaseClass {
               // Abstract methods and common behavior
                     }
               public class MySubClass extends MyBaseClass {
                   // Subclass-specific behavior
                         }
            // After refactoring
            // Remove MyBaseClass and directly implement behavior in MySubClass
                 public class MySubClass {
                // Combined behavior of MyBaseClass and MySubClass
                        }
```

5.4. Reduce Overgeneralization: Avoid creating overly generic or abstract classes or methods that do not have a clear and immediate purpose. Instead, favor simplicity and clarity in your code design.

For example:

```
// Before refactoring
                              public interface GenericService<T> {
                                 void performAction(T item);
                                            }
                    // After refactoring
                         // If there's no real need for genericity, make it specific
                             public interface SpecificService {
                                 void performAction(SpecificType item);      }
```

5. YAGNI Principle (You Ain't Gonna Need It): Apply the YAGNI principle to avoid adding functionality until it is actually needed. This helps prevent speculative generality by focusing on current requirements rather than potential future needs.

.