ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

EXERCISE 02

SOFTWARE ANALYSIS AND DESIGN

21120280 – Lý Minh Khuê

Thành phố Hồ Chí Minh – 2024

TABLE OF CONTENTS

UNDERSTANDING THE DRY PRINCIPLE

The Don't Repeat Yourself (DRY) principle is a fundamental software development concept that encourages programmers to **reduce code or knowledge** duplication within a codebase.
As **DRY** was first introduced in the book The Pragmatic Programmer, by Dave Thomas said, "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."
In simpler terms, **DRY** emphasizes the importance of writing code in such way that one function is implemented in one place only.

For example:

```java
public class FactorialCalculator {

    4 usages
    public static int factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial( n: n - 1);
        }
    }

    public static void main(String[] args) {
        // Calculate factorial of 5
        int factorialOf5 = factorial( n: 5);
        System.out.println("Factorial of 5: " + factorialOf5);

        // Calculate factorial of 7
        int factorialOf7 = factorial( n: 7);
        System.out.println("Factorial of 7: " + factorialOf7);

        // Calculate factorial of 10
        int factorialOf10 = factorial( n: 10);
        System.out.println("Factorial of 10: " + factorialOf10);
    }
}
```

Figure 1. FactorialCalculator without DRY

```java
public class FactorialCalculator {

    2 usages
    public static int factorial(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorial( n: n - 1);
        }
    }

    3 usages
    public static void calculateAndPrintFactorial(int number) {
        int factorialResult = factorial(number);
        System.out.println("Factorial of " + number + ": " + factorialResult);
    }

    public static void main(String[] args) {
        // Calculate and print factorial of 5
        calculateAndPrintFactorial( number: 5);

        // Calculate and print factorial of 7
        calculateAndPrintFactorial( number: 7);

        // Calculate and print factorial of 10
        calculateAndPrintFactorial( number: 10);
    }
}
```

Figure 2. FactorialCalculator with DRY

As you can see, in DRY version, I introduced the `calculateAndPrintFactorial()` methods that measure the actual factorial of a given number and print the result.

Now, the `main()` is only responsible calling this method for each repetitive tasks. This seperation of concerned subjects has avoided code duplication and promote code readability and maintainability.

IMPORTANCE OF THE DRY PRINCIPLE

1. Avoiding Redundancy:

The DRY principle advocates developers to avoid duplicating code segments across their projects. Instead of repeating identical code in multiple places, developers should know to encapsulate such logic into reusable functions, methods, or modules. This not only reduces the likelihood of bugs but also makes the codebase more manageable and easier to understand while working in a team.

2. Maintainability:

By eliminating redundancy in code, DRY simplifies maintenance tasks, ensures that any changes or updates to the code only need to be implemented in one place. Consequently, maintenance tasks become less prone to errors and more efficient, saving both time and effort in the long run.

3. Readability and Understandability:

Code that follows the DRY principle tends to be more clarity and understandable. This is particularly important in collaborative projects where multiple developers may need to work on the same codebase, as well as for debugging and testing purposes.

4. Scalability:

As the project evolves in size and complexity, maintaining redundant code becomes increasingly troublesome. By applying the DRY principle, developers make a solid foundation for scalability, enabling the codebase to grow and expand.

CHALLENGES AND CONSIDERATIONS

1. Over-Abstraction:

In some cases that are rather small, designing based on DRY can be more effort-consuming than maintaining two separate copied block of code, prevent unnecessarily complex code.

2. Balancing DRY with Other Principles:

DRY is a guiding principle but should not be pursued at the cost of readability, performance, or simplicity.

3. Context Sensitivity:

Not all code duplication is supposedly bad. Contextual considerations should guide decisions on code reuse.

4. Friendly Reminder:

It's important not to blindly apply the DRY principle without considering the context. If your business logic doesn't show duplication just yet, there's no need to force it. Hasty generalization can lead to unnecessary complexity and coupling in your codebase.

Creating abstractions after estimating of future needs can be counterproductive. You might end up spending time in building abstract classes and structures that are never utilized beyond their initial context.

Understanding the differences between code reuse and duplication is crucial. DRY emphasizes avoiding redundant knowledge, not abstracting everything for potential reuse.

Functionality that appears similar at first glance can diverge significantly over time. It's preferable to duplicate code and refactor later if necessary.

Principles are not rules that we must follow. They are just guiding tools, adaptable to suit the situation at hand. These should not constraint us; However, we must understand the logic behind these principles deeply to avoid complexity.

IMPLEMENTATION STRATEGIES

1. Function and Method Extraction:

Break down the code into smaller, functions or methods that perform specific tasks. Encapsulate reusable functionalities within modules to promote code reuse and minimize redundancy.

2. Inheritance and Polymorphism:

Identify common patterns or algorithms within the code and to create hierarchies of classes to reduce repetitive logic and knowledge.

3. Use of Libraries and Frameworks:

Utilize existing libraries, frameworks, and APIs whenever possible to handle common tasks and functionalities. This reduces the need to spending time in recreating the function and helps eliminate redundant code that can be replaced with well-established solutions.

4. Design Patterns:

Employ design patterns such as Factory, Singleton, or Strategy to address recurring design problems and promote code reuse.

REFERENCES

https://www.plutora.com/blog/understanding-the-dry-dont-repeat-yourself-principle
https://www.baeldung.com/cs/dry-software-design-principle
https://thevaluable.dev/dry-principle-cost-benefit-example/
https://viblo.asia/p/dont-repeat-yourself-3Q75wBEelWb