Lab - 6: Refactoring

Team Members:

Jay Sanjaybhai Patel (jy451478@dal.ca)

Kenil Kevadiya (kn486501@dal.ca)

Date:

February 29, 2024

Subject:

Software Development Concepts

Professor:

Michael McAllister

Questions

Q - 1. In part 1, how did you determine if a particular code smell was bad enough that it warranted documenting for later fixing?

→ First, I try to find common code smells, such as long methods, large classes, improper naming, duplicate code, etc.

→ I determine that it's essential to consider the impact of these code smells on the codebase, including readability and maintainability.

→ Moreover, I also assess:

- Does code readability get impacted by code smell?
- Does it increase the chances of introducing bugs?
- Are any coding standards or best practices being violated?
- Does it affect the future modifications or additions to the code?
- Does it impact performance or scalability?

→ Once I've decided that a code smell needs fixing, I document it with details such as its file name and location in a Word file.

Q - 2. In part 1, which code smells troubled you the most? The least? Why? What does that say, if anything, about your own approach to coding and your coding style?

Code smells that troubled the most

- **Magic Number:** The existence of hardcoded numeric values in code is referred to by this term. Magic numbers are vulnerable to errors and less understandable code since it's not always clear what they mean. They may cause misunderstandings and make it difficult to maintain or change the code later on.
- **Method is too long:** Long methods may be challenging for people to understand as well as maintain the codebase, and they are often a sign of poorly organised code. Long methods can lead to code duplication, higher coupling, decreased testability, and a violation of the Single Responsibility Principle. To resolve this, we need to understand that part so that we can separate it from the lengthy method and create another small method that performs a single task.

Code smells that troubled the least

- **Unnecessary comments:** While unnecessary comments can add noise to the code, they are typically written on lines that are easily understood by any developer. Sometimes, you may find these comments after every single line in methods.
- **Repetition of comments:** Similar to unnecessary comments, repetitive comments are those that are written in every method with similar lines of code.
- **Code duplication:** It refers to some parts of the code present in many methods in the same class. We can create one separate method for that task and reduce those common lines of code from every method.

- **Inconsistent naming of variables:** It refers to variables having different names in different classes. For example, if we have one variable called NetAmount, in different classes, it may be declared as Net_Amount, netAmount, etc.
- **Global variables are public:** It refers to the situation where, in our program, in some classes, global variables are set as public access modifiers. Due to that, other classes also access those variables, which can lead to potential issues such as unintended modification or misuse by other classes, violating principles of encapsulation.

Q - 3. What were the most troublesome refactorings to do in part 2? What made them troublesome?

→ The most difficult part was refactoring the main method present in TestHarness.java. It was too long, and within it, there were many unnecessary global variables.

→ Additionally, there were numerous lengthy if-else statements, which posed challenges. These factors made it difficult to determine how to tackle these issues. Therefore, I extracted the entire if-else block and created a new method for it.

→ Then, I further subdivided each if-else block into separate methods. Moreover, the global variables were being used to make decisions based on user input. I removed them and instead converted the input to uppercase before directly comparing it to predefined strings.

→ This approach allowed me to identify code smells in the Java file and attempt to resolve them.

Q - 4. In part 3, rather than throw an exception from the constructors, we could just avoid having constructors with multiple parameters. The parameter values would then need to be set by method calls by whomever creates the object. Explain which approach (the one implemented or the one just described) would result in better code

→ Both approaches have their own advantages and disadvantages, and which one is better depends on the specific context of our code and project requirement.

1. **Throwing an exception from constructors:**

- **Advantages:** It forces the caller to provide valid parameters during object creation. This can help catch errors early in the development process and enforce the correctness of object initialization.
- **Disadvantages:** Could potentially make the code less flexible if the parameters are not always required or if there are multiple valid combinations of parameters.

2. **Avoiding constructors with multiple parameters and using method calls to set parameter values:**

- **Advantages:** Makes it easier to enforce validation rules for each parameter individually.
- **Disadvantages:** Increases the risk of forgetting to set a required parameter, leading to runtime errors if the object is used without all necessary parameters being set.

→ According to me, the second one is more appropriate most of the time because it allows for a cleaner method or constructor signature. Later, we can add values using setter methods. We can also create multiple constructors using the concept of method overloading.