Cập nhật tháng 8 năm 2024

[Bài đọc] Refactoring kết hợp TDD - Hoạt động và lợi ích

1. **Refactoring Là Gì?**
   - Định nghĩa:
     - Refactoring là quá trình **cải thiện cấu trúc bên trong của mã nguồn** mà **không làm thay đổi hành vi bên ngoài** của chương trình
   - Mục tiêu:
     - Làm cho code **sạch hơn, dễ hiểu hơn, dễ bảo trì hơn** và **dễ mở rộng hơn**

2. **Vì Sao Phải Refactor?**
   - Code phình to, khó đọc
   - Code lặp lại ở nhiều nơi
   - Class, hàm quá dài hoặc làm nhiều việc
   - Khó mở rộng, sửa một chỗ dễ gây lỗi chỗ khác
   - Chất lượng giảm khi thêm tính năng liên tục

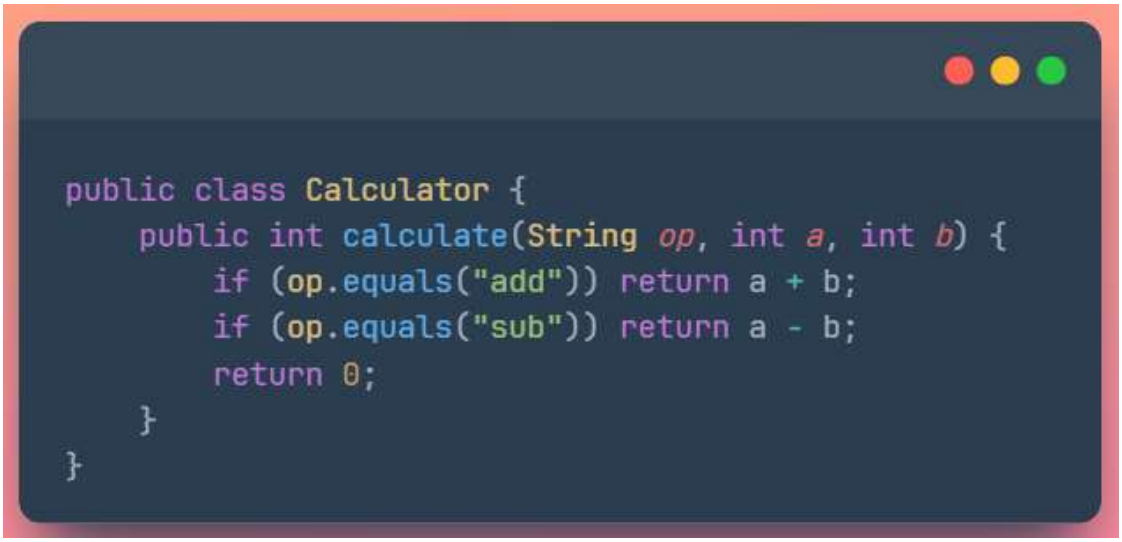
3. **Mối Quan Hệ Giữa TDD và Refactoring**
   - TDD và Refactoring luôn song hành:
     - **TDD tạo ra bộ test đáng tin cậy**, giúp bạn yên tâm **refactor** mà không lo phá vỡ chức năng
     - **Refactoring** giúp giữ cho code sạch, dễ đọc và dễ mở rộng trong suốt vòng đời dự án

   - Quy trình TDD có Refactoring ở bước thứ 3:
     - **Red:** Viết test thất bại
     - **Green:** Viết code đơn giản để test pass
     - **Refactor:** Cải thiện code (dựa trên test đảm bảo hành vi không đổi)

4. **Hoạt Động Refactoring Kết Hợp TDD Trong Java**
   - Quy trình Refactoring điển hình
     - Viết test để xác nhận chức năng (nếu chưa có)
     - Đảm bảo tất cả test **pass trước khi refactor**
     - Tiến hành refactor:
       - Đổi tên biến, hàm, class cho rõ nghĩa
       - Tách hàm dài thành các hàm nhỏ hơn
       - Loại bỏ code lặp lại
       - Tách class quá tải thành nhiều class nhỏ
       - Thay thế cấu trúc điều kiện phức tạp bằng polymorphism hoặc pattern
     - Chạy lại toàn bộ test

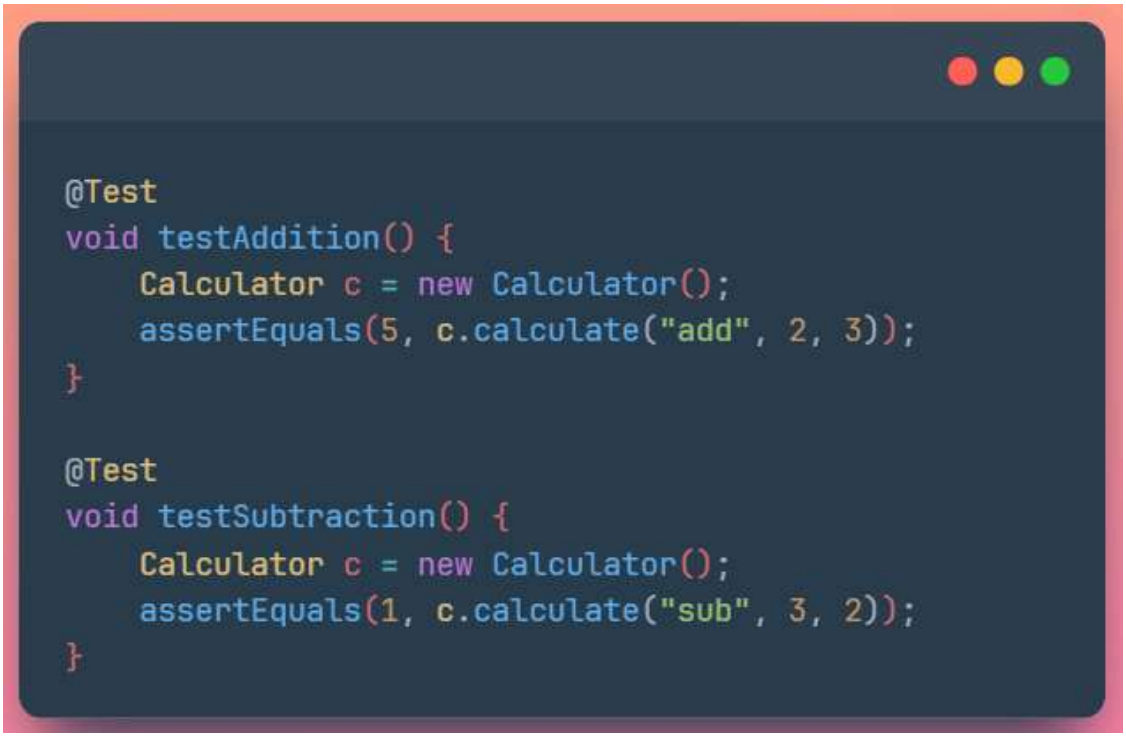
    - Nếu test pass, refactor thành công

- Ví dụ Refactoring với TDD
    - Trước refactor



```
public class Calculator {
    public int calculate(String op, int a, int b) {
        if (op.equals("add")) return a + b;
        if (op.equals("sub")) return a - b;
        return 0;
    }
}
```

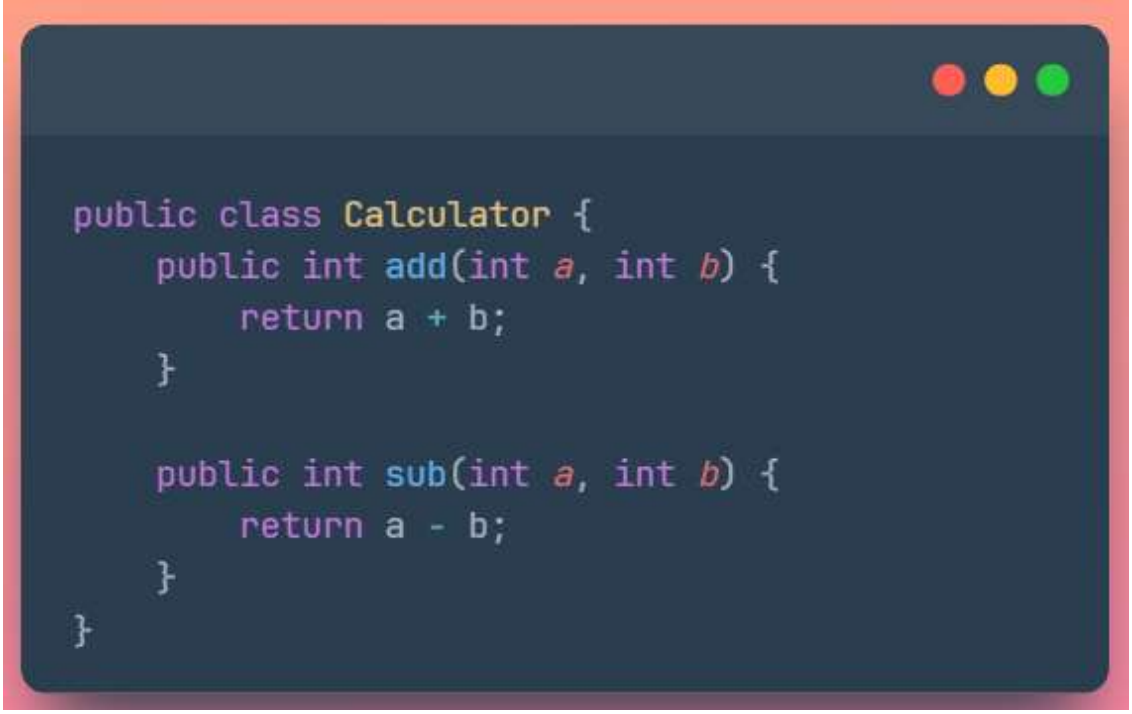
    - Test



```
@Test
void testAddition() {
    Calculator c = new Calculator();
    assertEquals(5, c.calculate("add", 2, 3));
}

@Test
void testSubtraction() {
    Calculator c = new Calculator();
    assertEquals(1, c.calculate("sub", 3, 2));
}
```

    - Sau refactor



```java
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public int sub(int a, int b) {
        return a - b;
    }
}
```

5. **Lợi Ích Của Refactoring Kết Hợp TDD**

Lợi ích	**Ý nghĩa**
An toàn khi thay đổi	Test đảm bảo code vẫn đúng sau refactor
Code dễ hiểu	Đổi tên rõ nghĩa, hàm nhỏ, class đúng chức năng
Loại bỏ nợ kỹ thuật	Giảm lỗi tích lũy, tránh code bẩn
Dễ mở rộng	Code sạch → dễ thêm tính năng
Tiết kiệm thời gian dài hạn	Giảm thời gian fix bug, bảo trì dễ hơn

6. **Nguyên Tắc Refactoring Hiệu Quả Với TDD**
- **Luôn có test bao phủ trước khi refactor**
- **Refactor từng bước nhỏ** – Thay đổi nhỏ, kiểm tra ngay
- **Chạy toàn bộ test sau mỗi lần thay đổi**
- **Tách biệt rõ Refactor và thêm tính năng:**
    - Hoặc **Refactor trước khi thêm tính năng**
    - Hoặc **Thêm tính năng xong rồi refactor ngay**

7. Các Kỹ Thuật Refactoring Thường Dùng
- Đổi tên (Rename)
- Tách hàm (Extract Method)
- Tách class (Extract Class)
- Di chuyển hàm hoặc trường (Move Method / Move Field)
- Loại bỏ code trùng lặp (Remove Duplication)
- Thay thế điều kiện bằng polymorphism
- Inline variable (Bỏ biến trung gian không cần thiết)

Danh sách các bài học