Kha Vĩnh Thuận – 22127408

1. Single Responsibility Principle (SRP) - Nguyên lý Trách nhiệm đơn:

1.1. **Định nghĩa chi tiết:**

Nguyên lý SRP nói rằng mỗi lớp hoặc module trong phần mềm chỉ nên có một trách nhiệm duy nhất và chỉ có một lý do duy nhất để thay đổi. Điều này có nghĩa là mỗi lớp nên giải quyết chỉ một vấn đề cụ thể, và nếu có sự thay đổi về yêu cầu hoặc đặc tính của phần mềm, chỉ có phần mã liên quan đến trách nhiệm đó cần phải thay đổi. Việc này giúp tránh sự phức tạp và rối rắm trong hệ thống.

1.2. Ví dụ vi phạm SRP:

Giả sử có một lớp trong hệ thống quản lý người dùng, lớp này vừa xử lý việc lưu trữ dữ liệu người dùng vào cơ sở dữ liệu, vừa gửi email xác nhận sau khi người dùng đăng ký.

Khi có yêu cầu thay đổi cách gửi email, bạn sẽ phải thay đổi cả lớp xử lý lưu trữ và gửi email. Điều này tạo ra sự phụ thuộc không cần thiết giữa các phần của hệ thống, làm mã nguồn trở nên phức tạp và dễ gây lỗi.

1.3. Ví dụ tuân thủ SRP:

Để tuân thủ SRP, ta có thể tách lớp này thành hai lớp riêng biệt: một lớp chỉ chịu trách nhiệm lưu trữ người dùng vào cơ sở dữ liệu, và một lớp khác chỉ gửi email. Việc thay đổi cách gửi email giờ chỉ ảnh hưởng đến lớp gửi email mà không làm ảnh hưởng đến lớp lưu trữ người dùng. Điều này làm giảm độ phức tạp và tăng tính dễ bảo trì của phần mềm.

1.4. Lợi ích của SRP:

- Dễ bảo trì: Khi mỗi lớp chỉ có một trách nhiệm duy nhất, việc bảo trì trở nên đơn giản hơn. Nếu có sự thay đổi yêu cầu, bạn chỉ cần thay đổi phần có liên quan mà không làm ảnh hưởng đến các phần khác của hệ thống.
- Dễ mở rộng: Khi muốn thêm tính năng mới, bạn có thể dễ dàng mở rộng phần có trách nhiệm mà không làm ảnh hưởng đến các phần khác trong mã nguồn.
- Dễ hiểu: Mã nguồn trở nên rõ ràng hơn khi mỗi lớp có trách nhiệm riêng biệt. Điều này giúp lập trình viên dễ dàng theo dõi và hiểu được mục đích của từng phần mã.

2. Don't Repeat Yourself (DRY) - Nguyên tắc Đừng Lặp Lại Chính Mình:

2.1. **Định nghĩa chi tiết:**

Nguyên tắc DRY yêu cầu tránh việc lặp lại mã nguồn hoặc logic trong phần mềm. Mọi thông tin, logic hoặc đoạn mã có thể tái sử dụng nên được viết một lần duy nhất và được gọi lại khi cần thiết. Mục tiêu là giảm thiểu sự lặp lại mã nguồn, giúp phần mềm dễ bảo trì và mở rộng.

2.2. **Ví dụ vi phạm DRY:**

Giả sử trong một ứng dụng, bạn có nhiều hàm xử lý yêu cầu từ người dùng và mỗi hàm đều có logic giống nhau để kiểm tra thông tin người dùng và xác nhận mật khẩu. Nếu mã kiểm tra này lặp lại ở nhiều nơi, khi có thay đổi yêu cầu về logic xác thực, bạn sẽ phải sửa lại ở tất cả các hàm, điều này dễ dẫn đến lỗi hoặc quên sửa ở một số nơi.

2.3. **Ví dụ tuân thủ DRY:**

Thay vì lặp lại mã kiểm tra xác thực ở mỗi hàm, bạn có thể tạo một hàm duy nhất để xử lý logic này và tái sử dụng hàm đó ở tất cả các nơi cần thiết. Việc này giúp đảm bảo rằng khi cần thay đổi logic xác thực, bạn chỉ cần sửa ở một nơi duy nhất.

Ví dụ, bạn có một hàm validateUser() để kiểm tra người dùng và mật khẩu, và thay vì viết lại mã này trong từng hàm, bạn chỉ cần gọi validateUser() khi cần.

Lợi ích của DRY:

- **Giảm sự lặp lại mã:** Mã trở nên ngắn gọn, dễ hiểu hơn khi không có sự trùng lặp. Các phần mã chung được viết một lần duy nhất, giảm sự dư thừa và giúp mã nguồn gọn gàng.
- **Dễ bảo trì:** Khi có lỗi hoặc yêu cầu thay đổi logic, bạn chỉ cần sửa chữa ở một nơi duy nhất, điều này giảm thiểu nguy cơ bỏ sót hoặc gây lỗi khi sửa mã ở nhiều nơi.
- **Tái sử dụng mã:** Khi mã đã được trích xuất ra thành các hàm hoặc module riêng biệt, chúng có thể tái sử dụng ở các phần khác của ứng dụng, giúp mã trở nên linh hoạt hơn và dễ mở rộng trong tương lai.

3. Giới Thiệu về Refactoring

Refactoring là quá trình cải tiến cấu trúc mã nguồn mà không làm thay đổi hành vi của chương trình. Mục tiêu chính của refactoring là làm cho mã nguồn trở nên dễ đọc, dễ hiểu, dễ bảo trì và mở rộng hơn, từ đó cải thiện chất lượng và khả năng duy trì của phần mềm.

3.1. Mục Tiêu và Lợi Ích của Refactoring

- **Cải thiện khả năng đọc mã**: Mã được refactor giúp lập trình viên dễ dàng hiểu và quản lý mã hơn.
- **Giảm độ phức tạp**: Refactoring giúp chia nhỏ các hàm, lớp, module quá dài thành các phần dễ hiểu và dễ quản lý hơn.
- **Tăng tính tái sử dụng và bảo trì**: Mã rõ ràng và dễ hiểu sẽ dễ dàng bảo trì và mở rộng khi có yêu cầu mới.
- **Giảm lỗi**: Quá trình refactoring giúp giảm thiểu lỗi do mã phức tạp và dễ bị bỏ sót khi thay đổi.

3.2. Các Kỹ Thuật Refactoring Phổ Biến

- **Tách hàm**: Chia một hàm quá dài hoặc quá phức tạp thành nhiều hàm nhỏ với trách nhiệm cụ thể.
- **Đổi tên biến/hàm/lớp**: Đảm bảo tên gọi của biến, hàm, lớp dễ hiểu, rõ ràng và mô tả chính xác chức năng.
- **Loại bỏ mã thừa:** Loại bỏ sự lặp lại mã và tái sử dụng mã ở nhiều nơi khác nhau.
- **Thay đổi cấu trúc dữ liệu:** Thay đổi cấu trúc dữ liệu không còn phù hợp để cải thiện hiệu suất hoặc tính rõ ràng của mã.

- **Đơn giản hóa biểu thức điều kiện**: Giảm độ phức tạp của các câu lệnh điều kiện để mã dễ hiểu hơn.

3.3. Quy Trình Refactoring

- **Hiểu mã hiện tại**: Trước khi refactor, cần phân tích và hiểu rõ cấu trúc mã hiện tại để xác định các phần cần cải tiến.
- **Chạy kiểm thử**: Đảm bảo rằng mã hiện tại có đầy đủ các bài kiểm thử (unit tests) để kiểm tra tính chính xác trước khi refactor.
- **Áp dụng refactoring dần dần**: Thực hiện cải tiến nhỏ và kiểm tra sau mỗi thay đổi để đảm bảo không ảnh hưởng đến hành vi của phần mềm.
- **Chạy kiểm thử sau khi refactor**: Kiểm tra mã sau khi refactor để đảm bảo không gây ra lỗi mới.

3.4. Khi nào nên dùng kĩ thuật Refactoring?

- **Mã khó đọc**: Khi mã trở nên quá phức tạp và khó hiểu.
- **Mã lặp lại**: Khi thấy sự lặp lại mã trong nhiều phần của ứng dụng.
- **Cải thiện hiệu suất**: Khi cần tối ưu hóa phần mềm hoặc thay đổi cấu trúc dữ liệu để tăng hiệu suất.
- **Khi cần bảo trì lâu dài**: Mã cần được refactor để dễ bảo trì và mở rộng trong tương lai.

3.5. Kết luận

Refactoring là một kỹ thuật quan trọng giúp cải thiện chất lượng mã nguồn, dễ bảo trì và mở rộng phần mềm. Quá trình này giúp giảm sự phức tạp của mã, tăng tính tái sử dụng và dễ dàng bảo trì phần mềm lâu dài. Việc áp dụng các kỹ thuật refactoring như tách hàm, đổi tên, loại bỏ mã thừa và thay đổi cấu trúc dữ liệu sẽ giúp phần mềm phát triển ổn định và dễ dàng thích nghi với các yêu cầu mới trong tương lai.