TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN – ĐHQG HCM
KHOA CÔNG NGHỆ THÔNG TIN

MÔN HỌC: THIẾT KẾ PHẦN MỀM
BÁO CÁO LÝ THUYẾT BROKEN WINDOW VÀ BOY SCOUT RULE

LỚP 22_3

NHÓM ARCEUS
22120271- Dương Hoàng Hồng Phúc
22120286 - Lê Võ Minh Phương
22120287 - Nguyễn Mạnh Phương
22120326 - Nguyễn Trường Tân

CHƯƠNG TRÌNH CHÍNH QUY - KHÓA 22

MỤC LỤC

I. The Broken Window Theory - Lý thuyết cửa sổ bị vỡ

1. Lý thuyết cửa sổ bị vỡ

Lý thuyết Cửa Sổ Bị Vỡ (Broken Window Theory) ra đời vào năm 1982, dựa trên một quan sát thú vị:

- Những căn nhà hoặc ô tô có cửa sổ bị vỡ, nếu không được sửa chữa kịp thời, sẽ dần dần trở thành mục tiêu cho sự phá hoại.
- Tương tự, những khu vực công cộng như đường đi bộ hay đất trống nếu bị xả rác mà không được dọn dẹp sẽ trở thành nơi tập trung rác thải.
- Nguyên nhân là do khi thấy cửa sổ không được sửa, mọi người sẽ nghĩ rằng "nhà này chắc bỏ hoang, đập phá cũng không ai quan tâm." Khi thấy rác chất chồng, họ sẽ nghĩ "chỗ này ai cũng xả rác, mình xả thêm có sao đâu ?"
- Điều này không chỉ đúng trong xã hội mà còn có thể áp dụng trong lĩnh vực lập trình và phát triển phần mềm. Những vấn đề nhỏ trong mã nguồn, nếu không được xử lý kịp thời, có thể dẫn đến những hệ quả nghiêm trọng hơn.

2. Ứng dụng trong lập trình

2.1 Những dòng code tồi tệ dần theo thời gian

Hãy tưởng tượng một dự án phần mềm nơi một nhóm lập trình viên đang phát triển một ứng dụng quản lý bán hàng. Ban đầu, mã nguồn của họ rất rõ ràng, sạch sẽ và có cấu trúc tốt. Tuy nhiên, khi áp lực thời gian gia tăng để ra mắt sản phẩm, nhóm buộc phải viết mã một cách vội vàng để hoàn thành các tính năng mới mà không có thời gian để viết tài liệu hay kiểm thử.

Chức năng đầu tiên được phát triển một cách ẩu tả, không có tài liệu và không có kiểm thử. Mặc dù ứng dụng thành công và được người dùng yêu thích, nhưng điều này đã tạo ra một "cửa sổ vỡ" trong mã nguồn. Khi các lập trình viên thấy rằng mã vẫn chạy mà không có tài liệu, họ bắt đầu lơ là trong việc viết mã sạch và tài liệu cho các tính năng sau.

2.2 Hệ quả của việc bỏ qua những vấn đề nhỏ

Khi những tính năng được phát triển một cách ẩu tả, mã nguồn của dự án dần trở thành một "đống hổ lốn." Các lập trình viên không còn cảm thấy cần thiết phải duy trì chất lượng mã, và điều này dẫn đến việc mã trở nên khó hiểu, khó bảo trì và dễ phát sinh lỗi. Tình trạng này giống như việc để cửa sổ vỡ mà không sửa chữa, dẫn đến việc mọi người nghĩ rằng không ai quan tâm đến chất lượng.

Khi một lỗi nhỏ nhặt không được giải quyết ngay từ đầu, mọi người ngầm khẳng định rằng việc phá vỡ các quy tắc, tiêu chuẩn hoặc thực hành sẽ không gây ảnh hưởng tiêu cực gì. Việc thiếu nhận thức về các hậu quả tiêu cực sẽ hình thành thói quen vô tổ chức; các thực hành không tốt ngày càng được chấp nhận và chất lượng bắt đầu giảm dần.
Áp dụng quy tắc **Broken window** trong lập trình

Lấp cửa sổ vỡ "ngay trong trứng nước":

- Để ngăn chặn những vấn đề lớn hơn, các lập trình viên cần phải sửa chữa những lỗi nhỏ ngay khi chúng còn nhỏ. Nếu sau khi hoàn thành một chức năng, nhóm phát triển dành thời gian để refactor mã, viết tài liệu và kiểm thử, họ sẽ tạo ra một mã nguồn sạch sẽ và dễ bảo trì hơn cho các tính năng sau này.
- Khi mọi người trong nhóm đều có ý thức về việc duy trì chất lượng mã, không ai dám "đập vỡ cửa sổ" bằng cách viết mã ẩu tả. Điều này không chỉ giúp cải thiện chất lượng sản phẩm mà còn tạo ra một môi trường làm việc tích cực hơn.

Triển khai các "good practice" - thực hành tốt là các chuẩn được các chuyên gia khuyến nghị:

- Giải pháp cho vấn đề này không phải là dễ dàng, nhưng có thể thực hiện được. Triển khai các thực hành tốt trong nhóm là rất quan trọng. Những người có kinh nghiệm thường mong muốn thể hiện bản thân, trong khi những người ít kinh nghiệm hơn chỉ thể hiện những gì họ có thể làm được. Sự chênh lệch khối lượng công việc thực sự không thể hiện điều gì. Thay vào đó, áp dụng cách tiếp cận từ tốn mà tập trung vào chất lượng sẽ hiệu quả hơn nhiều.

3. Tạm kết

Lý thuyết Cửa Sổ Bị Vỡ không chỉ áp dụng trong việc ngăn chặn tội phạm mà còn có thể áp dụng trong thói quen làm việc hàng ngày của lập trình viên. Những thói quen xấu ban đầu, như việc bỏ qua kiểm thử hoặc không viết tài liệu, có thể dẫn đến những vấn đề lớn hơn trong tương lai. Do đó, các lập trình viên cần phải cẩn thận và chú ý đến từng chi tiết

nhỏ trong mã nguồn của mình, để không để những "cửa sổ vỡ" xuất hiện và phát triển thành những vấn đề nghiêm trọng hơn. Việc duy trì chất lượng mã không chỉ là trách nhiệm của một cá nhân mà là của cả nhóm, tạo ra một văn hóa làm việc tích cực và hiệu quả hơn.

II. Boy Scout Rule – Nguyên tắc Hướng đạo sinh

1. Khái niệm

"Boy Scout Rule" (tạm dịch: Nguyên tắc Hướng đạo sinh) là một nguyên lý trong phát triển phần mềm, được phổ biến bởi Robert C. Martin (Uncle Bob) trong cuốn "Clean Code". Nguyên tắc này lấy cảm hứng từ phương châm của Hướng đạo sinh: "Hãy để lại nơi cắm trại sạch sẽ hơn khi bạn đến".

Trong lập trình, điều này có nghĩa: Mỗi khi bạn chỉnh sửa mã nguồn, hãy cố gắng cải thiện chất lượng code (dù nhỏ) trước khi rời đi. Dù là sửa lỗi, thêm tính năng, hay chỉ đọc code, lập trình viên nên để lại code base "sạch hơn" so với trạng thái ban đầu.

2. Ứng dụng của lý thuyết vào lập trình

Nguyên tắc này được áp dụng linh hoạt trong quy trình phát triển phần mềm, tập trung vào tính bền vững và dễ bảo trì của code. Cụ thể:

2.1 Tái cấu trúc code (Refactoring)

Sửa **code "có mùi"** - code smell: Phát hiện đoạn code khó hiểu, trùng lặp, hoặc vi phạm nguyên tắc SOLID/DRY, cần tái cấu trúc ngay.

Một vài đặc điểm cho thấy là một đoạn ***code smell:***

- Trùng lặp mã (**Duplicated Code**): Các đoạn mã giống nhau xuất hiện ở nhiều nơi, làm tăng nguy cơ lỗi và khó bảo trì khi thay đổi logic.
- Hàm/Method quá dài (**Long Methods**): Hàm thực hiện quá nhiều nhiệm vụ, gây khó hiểu, giảm khả năng tái sử dụng và kiểm thử.

- Lớp "Chúa tể" (**God Class**): Lớp ôm đồm quá nhiều trách nhiệm, vi phạm nguyên tắc Single Responsibility, khiến hệ thống kém linh hoạt và gây khó khăn cho những người đọc khác, thậm chí cả tác giả của đoạn code đó.
- Điều kiện phức tạp (**Complex Conditionals**): Các câu lệnh if-else lồng nhau nhiều tầng hoặc logic Boolean rối rắm, dễ gây hiểu nhầm và sai sót.
- **Magic Number/String**: Sử dụng trực tiếp giá trị số/chữ không giải thích (ví dụ: 86400 thay vì `SECONDS_PER_DAY`), khiến mã khó đọc và khó cập nhật.

Đặt tên có ý nghĩa: Thay thế tên biến/hàm mơ hồ (ví dụ: `x, temp`) bằng tên mô tả rõ chức năng (ví dụ: `userEmail, calculateTotalPrice`).

Giảm độ phức tạp: Chia nhỏ hàm dài thành các hàm con, tách lớp phức tạp thành module độc lập.

2.2 Dọn dẹp code "Rác"

Xóa code comment không cần thiết, code chết (dead code- code không sử dụng nữa), hoặc đoạn debug tạm thời.

Cập nhật thư viện lỗi thời hoặc phiên bản không an toàn.

2.3 Cải thiện test cases

Thêm unit test cho đoạn code thiếu coverage ,nghĩa là chúng ta cần viết thêm test case để kiểm tra nhánh code chưa được trước đó.

Ví dụ:

```
// Hàm cần test
function positiveSum(a, b) {
  if (a < 0 || b < 0) throw new Error("Số âm không hợp lệ");
  return a + b;
}
// Test case HIỆN TẠI (chỉ test số dương)
test('Tổng 2 số dương', () => {
  expect(positiveSum(2, 3)).toBe(5);
});
// Code coverage = 50% (nhánh "số âm" chưa được test)
```

Chúng ta cần viết thêm test cho số âm để đảm bảo code luôn hoạt động tốt trong mọi input:

```
test('Raise lỗi nếu số âm', () => {
  expect(() => sum(-1, 5)).toThrow("Số âm không hợp lệ");
});
//Code coverage tăng lên 100% (đã test cả 2 nhánh)
```

2.4 Tích hợp vào quy trình nhóm

Áp dụng trong **code review**: Yêu cầu thành viên chỉ ra và sửa code "bẩn" khi phát hiện.

Kết hợp với **Continuous Integration (CI):** Đảm bảo cải tiến code không phá vỡ build.

Continuous Integration (CI) là quy trình tự động hóa việc tích hợp code từ nhiều thành viên vào kho lưu trữ chung (như Git). Mỗi khi code được đẩy lên, hệ thống CI tự động: Build (biên dịch) code, Chạy test (unit test, integration test),Báo cáo lỗi nếu phát hiện vấn đề.

Tất cả hoạt động trên nhằm giúp phát hiện lỗi sớm, đảm bảo code luôn ổn định và sẵn sàng triển khai nhanh chóng cho khách hàng.

3. Ví dụ cụ thể về áp dụng Boy Scout Rule

Ngữ cảnh: Dự án web quản lý đặt phòng khách sạn.

Tình huống: Lập trình viên A được giao sửa lỗi "Không hiển thị giá phòng sau khi áp dụng voucher" sau khi được khách hàng phản ánh

Khi xem xét code, A phát hiện đoạn xử lý tính giá trong hàm `calculateRoomPrice()` như sau:

```
// File: bookingService.js
function calculateRoomPrice(room, discount) {
  let price = room.price;
  let temp; // Biến thừa, không sử dụng
  // Xử lý giảm giá
  if (discount) {
```

```
      price = price - discount * 0.1; // Logic sai: Giảm 10% của
discount, không phải giá
    }
    // Xử lý thuế (10%)
    if (price > 100) {
      price += price * 0.1;
    } else {
      price += price * 0.05;
    }
    // Phụ phí dịch vụ
    if (room.type === 'VIP') {
      price += 50;
    } else {
      price += 20;
    }
    // ... Thêm các logic phức tạp khác
    return Math.round(price);
  }
```

Như chúng ta thấy, trong hàm `calculateRoomPrice()` đang phải làm quá nhiều tính năng trong việc tính giá phòng khách sạn, vi phạm nguyên tắc **Single Responsibility** trong **SOLID.** Ngoài ra, cách đặt tên biến chưa rõ ràng, có thể gây khó hiểu cho người đọc và bảo trì sau này và logic tính giảm giá đang sai: `discount * 0.1` thay vì đúng là `price = price - price * discount`.Các chính sách về giảm giá, thuế có thể thay đổi trong tương lai nên chúng ta không nên hardcode các giá trị như vậy.

Chúng ta có thể áp dụng **Boy Scout Rule** để chỉnh sửa như sau:

```
// Tách logic thành các hàm con tương ứng với chỉ một chức năng
cho mỗi hàm
function applyDiscount(price, voucher) {
  if (!voucher) return price;
  // Giả sử voucher là % giảm giá (ví dụ: 0.1 = 10%)
  return price - price * voucher; // Sửa logic sai thành đúng
}
function calculateTax(price) {
  const taxRate = price > 100 ? 0.1 : 0.05;
  return price * taxRate;
```

```
}
function addServiceFee(roomType) {
  return roomType === 'VIP' ? 50 : 20;
}
// Hàm chính được tái cấu trúc
function calculateRoomPrice(room, voucher) {
  let finalPrice = room.price;
  // Áp dụng từng bước riêng biệt
  finalPrice = applyDiscount(finalPrice, voucher);
  finalPrice += calculateTax(finalPrice);
  finalPrice += addServiceFee(room.type);
  return Math.round(finalPrice);
}
```

Đoạn code này đã được chia hàm lớn thành 3 hàm nhỏ:

+ `applyDiscount()`: Xử lý giảm giá (sửa logic sai).
+ `calculateTax()`: Tính thuế dựa trên giá.
+ `addServiceFee()`: Thêm phụ phí theo loại phòng.

Đặt tên có nghĩa: Thay discount → voucher để phản ánh đúng business logic.

Xóa biến temp thừa.

Thêm tính linh hoạt: Dễ dàng mở rộng từng bước tính toán (ví dụ: thêm phụ phí khác).

4. Nhận xét và lời kết cho lý thuyết Boy Scout Rule

Ưu điểm:

- Cải thiện chất lượng code liên tục, ngăn tích tụ "technical debt".
- Tạo văn hóa trách nhiệm tập thể: Mọi thành viên đều đóng góp vào code sạch.
- Phù hợp với các phương pháp Agile/DevOps.

Hạn chế:

- Đòi hỏi kỷ luật cao từ lập trình viên.
- Có thể tốn thời gian nếu cải tiến quá mức cần thiết (over-engineering).

Lời kết:

Boy Scout Rule không đòi hỏi thay đổi lớn, mà khuyến khích cải tiến từng bước nhỏ. Giống như việc dọn dẹp nhà cửa hàng ngày, nguyên tắc này giúp code base luôn "sạch sẽ", dễ mở rộng và bảo trì. Đây là kỹ năng thiết yếu để xây dựng phần mềm chất lượng cao và bền vững.

TÀI LIỆU THAM KHẢO

Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship.

Fowler, M. (2018). Refactoring: Improving the Design of Existing Code.