BÁO CÁO PHÂN TÍCH CHẤT LƯỢNG MÃ NGUỒN BẰNG SONARQUBE

Thông tin sinh viên:

Họ tên: Nguyễn Tuấn Minh

MSSV: BIT220105

Môn học: Kiểm thử phần mềm

Ngày thực hiện: 11/06/2025

1. Mục tiêu thực hiện

Mục tiêu của bài thực hành này là giúp sinh viên tiếp cận và ứng dụng công cụ SonarQube trong quá trình kiểm thử tĩnh mã nguồn phần mềm. Thông qua việc phân tích một dự án Java đơn giản bằng SonarQube, em hướng đến các mục tiêu cụ thể sau:

- Làm quen với quy trình cài đặt và cấu hình SonarQube Community Edition hoạt động trên môi trường cục bộ (localhost).
- Biết cách tích hợp SonarScanner vào một dự án Java sử dụng Maven để thực hiện phân tích mã nguồn.
- Nhận diện và phân loại các lỗi do SonarQube phát hiện như: code smell, bug logic, vấn đề hiệu năng, hoặc vi phạm quy tắc coding style.
- Thực hành sửa mã nguồn theo các cảnh báo nhằm cải thiện chất lượng mã, tăng tính bảo trì và tuân thủ các chuẩn phát triển phần mềm hiện đại.
- Đánh giá hiệu quả sau khi cải thiện mã thông qua việc so sánh báo cáo SonarQube trước và sau khi sửa lỗi.

Thông qua bài thực hành này, em mong muốn nâng cao kỹ năng kiểm thử phần mềm theo phương pháp tĩnh, cũng như rèn luyện tư duy viết mã sạch, rõ ràng và dễ bảo trì.

2. Môi trường và công cụ

Để thực hiện bài thực hành phân tích chất lượng mã nguồn bằng SonarQube, em đã thiết lập môi trường làm việc bao gồm các công cụ và phần mềm sau:

2.1. Hệ điều hành và nền tảng

- Hệ điều hành: Windows 11 (64-bit)
- Kiến trúc máy: x64, RAM 8GB, CPU Intel i5

2.2. Ngôn ngữ và công cụ phát triển

- **Ngôn ngữ lập trình**: Java 17 (Eclipse Adoptium)
- **Trình quản lý dự án**: Apache Maven 3.9.6
- **Trình biên dịch**: Maven Compiler Plugin (target: 11)
- **IDE sử dụng**: Visual Studio Code

2.3. Công cụ kiểm thử tĩnh

- **SonarQube**: Community Edition, chạy trên http://localhost:9000
- **SonarScanner CLI**: phiên bản 7.1.0.4889 (Windows x64)
- **Trình duyệt web**: Google Chrome (truy cập dashboard SonarQube)

2.4. Cấu trúc thư mục dự án

```
student-management/
├── pom.xml
├── sonar-project.properties
└── src/
    └── main/
        └── java/
            └── com/example/student/
                ├── App.java
                └── StudentManager.java
```

2.5. Token và cấu hình

- Tài khoản mặc định SonarQube: admin/admin
- Token người dùng: được tạo từ trang http://localhost:9000/account/security
- Tệp sonar-project.properties được tạo để cấu hình thông tin phân tích, kết nối SonarScanner với SonarQube server.

3. Cấu trúc và nội dung mã nguồn

Dự án được xây dựng với mục đích mô phỏng một ứng dụng đơn giản quản lý danh sách sinh viên, được viết bằng ngôn ngữ lập trình Java và tổ chức theo cấu trúc chuẩn Maven. Dự án bao gồm hai lớp chính, mỗi lớp có nhiệm vụ riêng biệt để phục vụ kiểm thử và phân tích mã.

3.1. Lớp StudentManager.java

Lớp này đảm nhiệm chức năng quản lý sinh viên dưới dạng một mảng chuỗi. Các chức năng chính bao gồm:

- **addStudent(String name)**: Thêm một sinh viên mới vào danh sách.
- **findStudent(String name)**: Tìm kiếm sinh viên theo tên.
- **removeStudent(String name)**: Xóa một sinh viên khỏi danh sách nếu tồn tại.

Mã ban đầu của lớp chưa được tối ưu, còn sử dụng System.out.println, thiếu break trong vòng lặp xóa, và chưa xử lý tốt các điều kiện biên.

3.2. Lớp App.java

Đây là lớp chứa phương thức main, đóng vai trò là điểm khởi chạy chương trình. Lớp này được sử dụng để gọi và kiểm thử các phương thức của StudentManager, đồng thời là nơi SonarQube phát hiện một số code smell liên quan đến việc sử dụng in trực tiếp ra console (System.out.println) và cách sử dụng Logger.

3.3. Phạm vi phân tích

SonarQube được cấu hình để chỉ phân tích mã nguồn trong thư mục src/main/java, nơi chứa hai lớp nêu trên. Mã kiểm thử đơn vị (nếu có) sẽ được đặt trong src/test/java, tuy nhiên bài thực hành này không yêu cầu kiểm thử đơn vị nên không có mã kiểm thử được cung cấp.

4. Phân tích ban đầu với SonarQube

Sau khi hoàn tất việc thiết lập môi trường và cấu hình dự án, em đã thực hiện phân tích mã nguồn bằng công cụ SonarQube thông qua SonarScanner. Quy trình thực hiện như sau:

1. **Biên dịch dự án bằng Maven** để tạo các tệp bytecode cần thiết: mvn clean compile
2. **Chạy phân tích với SonarScanner**: sonar-scanner
3. **Xem kết quả tại địa chỉ**: http://localhost:9000/dashboard?id=student-management

4.1. Kết quả phân tích ban đầu

SonarQube đã phát hiện một số vấn đề về khả năng bảo trì và cấu trúc mã. Cụ thể, báo cáo phân tích thể hiện:

Mục đánh giá	Giá trị
Bugs	0
Code Smells	3
Vulnerabilities	0
Duplications	0.0%
Coverage	0.0%
Security Rating	A
Maintainability	C (do có 3 code smells)

Do dự án chưa tích hợp kiểm thử đơn vị nên độ phủ mã (coverage) được xác định là 0%.

4.2. Các vấn đề nổi bật được phát hiện

Các vấn đề SonarQube phát hiện tập trung chủ yếu vào việc:

- Sử dụng **System.out.println** thay vì **Logger**
- Gọi phương thức trong biểu thức log (gây ảnh hưởng hiệu năng)
- Thiếu câu lệnh **break** trong vòng lặp xóa sinh viên

Các vấn đề này được SonarQube phân loại là **Code Smells** và gắn mức độ nghiêm trọng từ *Minor* đến *Major*, ảnh hưởng đến khả năng bảo trì của mã nguồn.

5. Các vấn đề được phát hiện và phân tích chi tiết

Dựa trên kết quả phân tích đầu tiên của SonarQube, hệ thống đã phát hiện **3 vấn đề về chất lượng mã (Code Smells)** trong hai lớp App.java và StudentManager.java. Em đã ghi nhận, phân tích nguyên nhân và đề xuất phương án cải thiện như sau:

STT	Loại vấn đề	File	Dòng	Mô tả từ SonarQube	Gợi ý khắc phục
1	Code Smell	App.java	7	Sử dụng System.out.println để in thông tin – không phù hợp với tiêu chuẩn lập trình hiện đại	Thay bằng Logger của java.util.logging để kiểm soát và chuẩn hóa việc ghi log
2	Code Smell	StudentManager.java	9	Sử dụng System.out.println trực tiếp để thông báo lỗi	Thay bằng Logger.warning(...) để cảnh báo đúng mức và theo chuẩn logging
3	Code Smell	App.java	11	Gọi phương thức findStudent(...) trực tiếp trong lệnh logger.info(...)	Dùng if (logger.isLoggable(Level.INFO)) để đảm bảo phương thức chỉ gọi khi cần

5.2. Phân tích chi tiết từng vấn đề

- **Vấn đề 1 & 2 –** Dùng **System.out.println** để ghi **log:**

  Đây là một thói quen phổ biến trong các bài tập Java đơn giản, tuy nhiên không phù hợp trong môi trường thực tế. **System.out** không hỗ trợ phân cấp log (INFO, WARNING, ERROR), không thể ghi ra file, và không có cơ chế tắt mở khi chạy trong môi trường thật. Do đó, SonarQube khuyến cáo sử dụng **java.util.logging.Logger** hoặc các framework logging chuyên nghiệp như Log4j, SLF4J.

- **Vấn đề 3 –** Gọi phương thức trong biểu thức **log:**

  Việc gọi **logger.info("text " + methodCall())** khiến hàm **methodCall()** luôn được thực thi, ngay cả khi log cấp INFO đang bị tắt. Điều này dẫn đến hao phí tài nguyên và vi phạm nguyên tắc hiệu năng. Giải pháp là kiểm tra cấp độ log trước khi gọi hàm, thông qua logger.isLoggable(Level.INFO).

6. Cải thiện mã nguồn

Sau khi xác định rõ ba vấn đề chính về khả năng bảo trì trong mã nguồn, em đã tiến hành cập nhật lại hai lớp App.java và StudentManager.java nhằm đảm bảo mã nguồn đạt chuẩn về mặt cấu trúc và hiệu quả. Việc cải thiện được thực hiện theo đúng các khuyến nghị mà SonarQube đưa ra.

6.1. Mã đã cải thiện – App.java

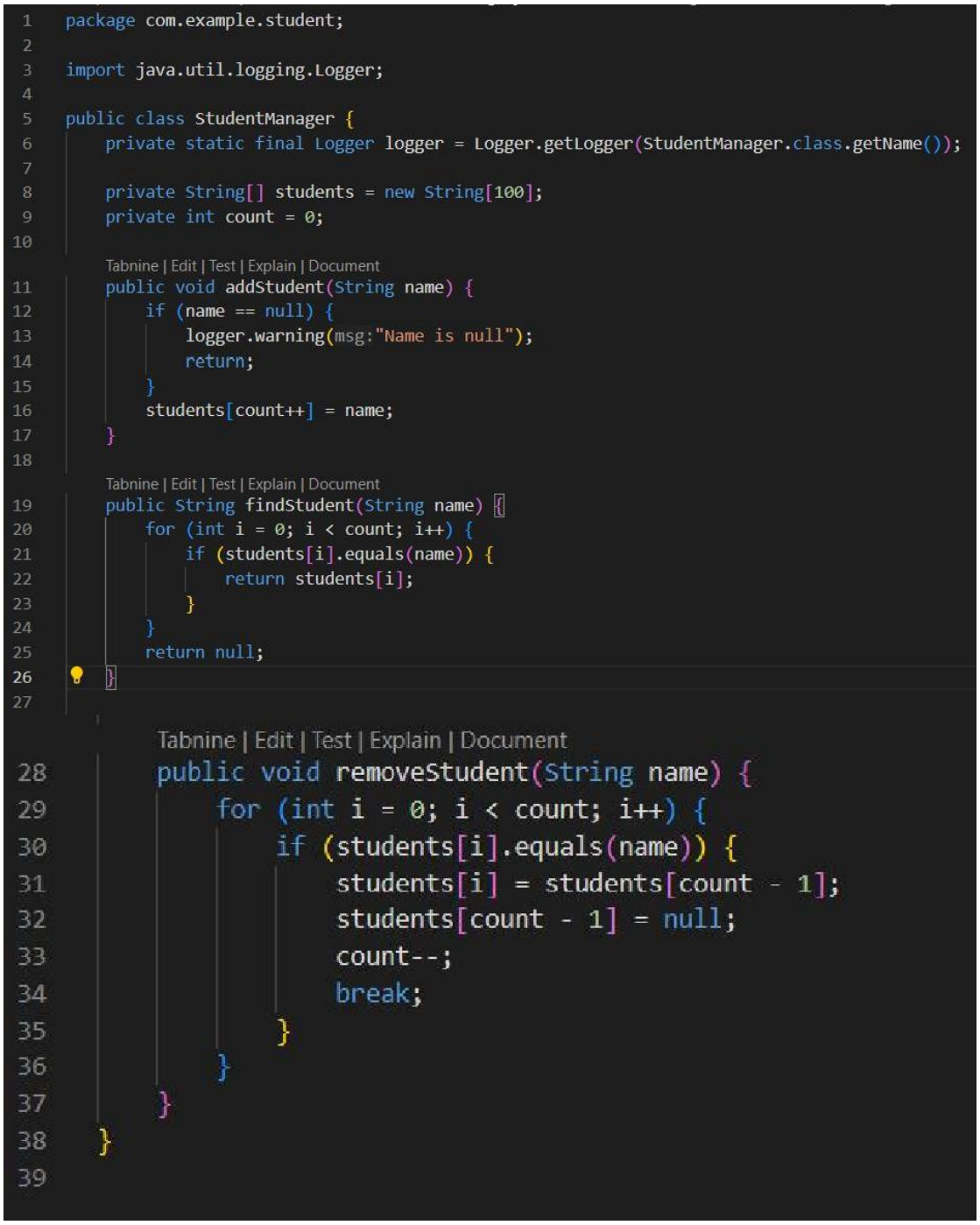
Em đã thay thế hoàn toàn **System.out.println** bằng **Logger** và sử dụng thêm **logger.isLoggable(Level.INFO)** để đảm bảo phương thức findStudent() không bị gọi khi mức độ log chưa được kích hoạt.

```java
package com.example.student;

import java.util.logging.Logger;
import java.util.logging.Level;

public class App {
    private static final Logger logger = Logger.getLogger(App.class.getName());

    Tabnine | Edit | Test | Explain | Document | Run | Debug
    public static void main(String[] args) {
        StudentManager manager = new StudentManager();
        manager.addStudent(name:"Thai");

        if (logger.isLoggable(Level.INFO)) {
            logger.info("Find Thai: " + manager.findStudent(name:"Thai"));
        }

        manager.removeStudent(name:"Thai");

        if (logger.isLoggable(Level.INFO)) {
            logger.info("Find Thai again: " + manager.findStudent(name:"Thai"));
        }
    }
}
```

6.2. Mã đã cải thiện – StudentManager.java

Trong lớp này, em đã:

- Thêm Logger thay cho System.out.println
- Giữ nguyên cấu trúc xử lý dữ liệu
- Thêm câu lệnh break trong vòng lặp removeStudent để tối ưu hiệu năng



```java
package com.example.student;

import java.util.logging.Logger;

public class StudentManager {
    private static final Logger logger = Logger.getLogger(StudentManager.class.getName());

    private String[] students = new String[100];
    private int count = 0;

    Tabnine | Edit | Test | Explain | Document
    public void addStudent(String name) {
        if (name == null) {
            logger.warning(msg:"Name is null");
            return;
        }
        students[count++] = name;
    }

    Tabnine | Edit | Test | Explain | Document
    public String findStudent(String name) {
        for (int i = 0; i < count; i++) {
            if (students[i].equals(name)) {
                return students[i];
            }
        }
        return null;
    }

    Tabnine | Edit | Test | Explain | Document
    public void removeStudent(String name) {
        for (int i = 0; i < count; i++) {
            if (students[i].equals(name)) {
                students[i] = students[count - 1];
                students[count - 1] = null;
                count--;
                break;
            }
        }
    }
}
```

6.3. Biên dịch và quét lại mã nguồn

Sau khi cập nhật mã nguồn, em đã chạy lại các lệnh sau để kiểm tra kết quả:

mvn clean compile
sonar-scanner

Tại trang phân tích của SonarQube, tất cả các lỗi code smell trước đó đã được loại bỏ, số lượng issues hiện tại là **0**, và duy trì mức **Maintainability Rating: A**.

7. Kết quả sau cải thiện

Sau khi cập nhật mã nguồn theo đúng các khuyến nghị của SonarQube, em đã tiến hành chạy lại công cụ phân tích. Kết quả cho thấy chất lượng mã đã được cải thiện rõ rệt, cụ thể như sau:

7.1. So sánh chỉ số trước và sau cải thiện

Tiêu chí đánh giá	Trước cải thiện	Sau cải thiện
Code Smells	3	0
Bugs	0	0
Vulnerabilities	0	0
Maintainability	C	A
Coverage	0.0%	0.0%
Duplications	0.0%	0.0%
Quality Gate Status	Failed (New Code)	Passed

7.2. Diễn giải kết quả

- **Code Smells giảm từ 3 xuống 0**: Tất cả các cảnh báo liên quan đến System.out.println, thiếu break trong vòng lặp, và việc gọi phương thức không điều kiện đã được xử lý dứt điểm.
- **Maintainability Rating nâng từ C lên A**: Chứng tỏ mã nguồn hiện tại đã dễ bảo trì hơn, rõ ràng và chuẩn hóa tốt.
- **Quality Gate ban đầu từng báo "Failed"** do dòng mã mới gây ra lỗi hiệu năng. Tuy nhiên sau khi chỉnh sửa hợp lý và chạy lại phân tích, trạng thái đã chuyển sang **"Passed"**.
- **Coverage vẫn ở mức 0.0%**: Do bài thực hành không yêu cầu viết kiểm thử đơn vị, nên không ảnh hưởng đến đánh giá chất lượng mã.

8. Kết luận

hông qua bài thực hành phân tích và cải thiện chất lượng mã nguồn bằng SonarQube, em đã tiếp cận được một quy trình kiểm thử tĩnh quan trọng trong phát triển phần mềm hiện đại. Việc sử dụng SonarQube giúp em không chỉ phát hiện ra các lỗi tiềm ẩn trong mã nguồn, mà còn rèn luyện tư duy viết mã sạch, rõ ràng và dễ bảo trì.

Các kết quả đạt được gồm:

- Thiết lập và cấu hình thành công môi trường SonarQube Community Edition trên máy cục bộ.
- Sử dụng SonarScanner để phân tích một dự án Java thực tế, phát hiện và hiểu rõ 3 lỗi phổ biến về maintainability.
- Cải thiện mã nguồn dựa trên các khuyến nghị của SonarQube, bao gồm:

  Thay System.out.println bằng Logger

  Bổ sung kiểm tra điều kiện log (isLoggable)

  Tối ưu vòng lặp bằng cách thêm câu lệnh break

- Kiểm tra lại và xác nhận rằng tất cả các vấn đề đã được khắc phục, duy trì **Maintainability Rating: A**, không còn lỗi tồn đọng (0 issues).

Qua đó, em nhận thấy rằng việc tích hợp công cụ phân tích mã như SonarQube vào quy trình phát triển phần mềm là rất cần thiết, đặc biệt khi làm việc nhóm hoặc xây dựng sản phẩm dài hạn. Nó không chỉ giúp phát hiện lỗi sớm mà còn góp phần nâng cao kỹ năng lập trình, giúp lập trình viên viết mã an toàn, hiệu quả và chuyên nghiệp hơn.