BÁO CÁO PHÂN TÍCH VÀ ĐÁNH GIÁ THIẾT KẾ HỆ THỐNG

**MỤC LỤC**

[1. Phân tích Coupling (Độ kết dính) và Cohesion (Độ liên kết) 1](#_Toc200810626)

[1.1. Giới thiệu lý thuyết 1](#_Toc200810627)

[1.2. Phân tích trong thiết kế hiện tại của dự án 2](#_Toc200810628)

[1.2.1. Về Cohesion (Độ liên kết - Đánh giá: TỐT) 2](#_Toc200810629)

[1.2.2. Về Coupling (Độ kết dính - Đánh giá: TỐT) 3](#_Toc200810630)

[1.3. Đề xuất cải thiện và hành động 4](#_Toc200810631)

[2. Phân tích mức độ đáp ứng nguyên lý SOLID 4](#_Toc200810632)

[2.1. Giới thiệu lý thuyết 4](#_Toc200810633)

[2.2. Phân tích trong thiết kế hiện tại 5](#_Toc200810634)

[2.2.1. Trước khi thêm yêu cầu mở rộng (Thiết kế hiện tại) 5](#_Toc200810635)

[2.2.2. Sau khi thêm yêu cầu mở rộng 6](#_Toc200810636)

[2.3. Đề xuất cải thiện để đáp ứng SOLID tốt hơn 7](#_Toc200810637)

[2.3.1. Cải thiện OCP/DIP cho việc tích hợp Git (REQ-EXT-2) 7](#_Toc200810638)

[2.3.2. Cải thiện OCP cho việc quản lý vai trò (REQ-EXT-1) 8](#_Toc200810639)

[2.4. Kết luận và Hành động 9](#_Toc200810640)

# 

# 1. Phân tích Coupling (Độ kết dính) và Cohesion (Độ liên kết)

## 1.1. Giới thiệu lý thuyết

* **Cohesion (Độ liên kết):** Là thước đo mức độ liên quan giữa các thành phần bên trong một module (ví dụ: một file, một class, một component). Thiết kế tốt hướng tới **High Cohesion** (Độ liên kết cao), nghĩa là một module chỉ nên thực hiện một nhiệm vụ duy nhất và các thành phần bên trong nó phải phục vụ cho nhiệm vụ đó.
* **Coupling (Độ kết dính):** Là thước đo mức độ phụ thuộc giữa các module khác nhau. Thiết kế tốt hướng tới **Loose Coupling** (Độ kết dính lỏng lẻo), nghĩa là sự thay đổi trong một module ít hoặc không ảnh hưởng đến các module khác.

## 1.2. Phân tích trong thiết kế hiện tại của dự án

Dựa trên cấu trúc thư mục, thiết kế của dự án đã thể hiện sự tuân thủ tốt các nguyên tắc về Coupling và Cohesion.

### 1.2.1. Về Cohesion (Độ liên kết - Đánh giá: TỐT)

#### Backend (High Cohesion)

* **Phân tách theo chức năng:** Cấu trúc controllers, services, models, routes là một ví dụ điển hình của việc đạt được độ liên kết chức năng (Functional Cohesion) - mức độ cao nhất.
  + userController.js chỉ xử lý các yêu cầu HTTP liên quan đến người dùng.
  + UserModel.js chỉ định nghĩa cấu trúc dữ liệu của người dùng.
  + Mỗi module đều có một mục đích rõ ràng và duy nhất.
* **Middleware:** Các middleware như verify-token.js và check-role.js có độ liên kết rất cao. Chúng chỉ làm một việc duy nhất: xác thực token và kiểm tra vai trò, và được áp dụng cho nhiều route khác nhau (cross-cutting concern).

#### Frontend (High Cohesion)

* **Cấu trúc Component:** Các component được chia nhỏ theo chức năng (auth, dashboard, class-manager) và có các component con chuyên biệt (ví dụ: CommitActivityChart, LateTasksChart). Điều này cho thấy mỗi component có một trách nhiệm duy nhất, giúp dễ bảo trì và tái sử dụng.
* **Custom Hooks:** Việc sử dụng các custom hooks (useCommitActivity.js, useTaskSummaryData.js,...) là một điểm sáng tuyệt vời. Chúng đóng gói logic stateful và các hiệu ứng phụ (side effects) liên quan đến một chức năng cụ thể (ví dụ: lấy dữ liệu cho biểu đồ). Điều này giúp component giao diện (Dashboard.js) trở nên gọn gàng, chỉ tập trung vào việc hiển thị, đạt được High Cohesion.
* **Services:** Tương tự backend, frontend cũng tách biệt logic gọi API vào thư mục services (auth-service.js, class-service.js). Điều này giúp các component không cần biết chi tiết về việc gọi API như thế nào.

#### Điểm cần xem xét/cải thiện

* groupsController.js và groupController.js: Sự tồn tại của hai file này có thể là dấu hiệu của sự trùng lặp hoặc một file cũ chưa được xóa. Điều này làm giảm độ liên kết vì logic quản lý nhóm bị phân tán hoặc không rõ ràng.
* Các component lớn: Một số component như Dashboard.js hoặc UserManagerPage.js có thể trở nên quá lớn nếu chúng chứa quá nhiều logic con. Cần đảm bảo chúng chỉ đóng vai trò "container" và ủy thác công việc cho các component con chuyên biệt.

### 1.2.2. Về Coupling (Độ kết dính - Đánh giá: TỐT)

* **Backend - Frontend (Loose Coupling):**
  + Hai phần này hoàn toàn độc lập và chỉ giao tiếp với nhau thông qua một giao diện chung là API (được định nghĩa trong routes). Backend không biết gì về cách frontend được triển khai (React, Vue,...) và ngược lại. Đây là một ví dụ điển hình của **Data Coupling** (truyền dữ liệu qua API), là một dạng kết dính lỏng lẻo và mong muốn.
* **Trong Backend (Loose Coupling):**
  + **Controller và Service:** Controller gọi đến Service để thực thi logic nghiệp vụ. Sự phụ thuộc này là cần thiết, nhưng vì Service không biết gì về req, res của Express, nó có thể được tái sử dụng ở nơi khác (ví dụ: trong một cron job). Điều này giữ cho độ kết dính ở mức chấp nhận được.
  + **Service và Model:** Service phụ thuộc vào Model để tương tác với cơ sở dữ liệu. Đây là một sự phụ thuộc tự nhiên.
* **Trong Frontend (Loose Coupling):**
  + **Component và Service:** Các component không trực tiếp dùng fetch hay axios. Thay vào đó, chúng gọi các hàm từ services. Nếu sau này muốn đổi thư viện HTTP từ axios sang fetch, ta chỉ cần sửa trong file api-client.js hoặc các file service tương ứng mà không cần động đến hàng loạt component.
  + **Component và Hooks:** Như đã nói ở trên, component sử dụng hook mà không cần biết chi tiết triển khai bên trong hook. Ví dụ, Dashboard.js chỉ cần gọi useTaskSummaryData() và nhận về dữ liệu, nó không quan tâm dữ liệu đó đến từ đâu, giúp giảm sự phụ thuộc.

## 1.3. Đề xuất cải thiện và hành động

1. **Rà soát và hợp nhất groupController.js và groupsController.js:**
   * **Hành động:** Kiểm tra nội dung của hai file. Giữ lại file có tên nhất quán với các controllers khác (groupController.js) và hợp nhất toàn bộ logic vào đó. Xóa file còn lại để tránh nhầm lẫn và đảm bảo **High Cohesion**.
   * **Source code:** Sửa lại các require trong group-routes.js để trỏ đến controller duy nhất.
2. **Duy trì "Thin Controller, Fat Service" trên Backend:**
   * **Hành động:** Thường xuyên review code để đảm bảo các controllers chỉ làm nhiệm vụ parse request, validate input cơ bản và gọi service. Toàn bộ logic nghiệp vụ (tính toán, xử lý dữ liệu phức tạp, gọi nhiều model) phải nằm trong services. Điều này duy trì **Loose Coupling** giữa tầng giao vận (HTTP) và tầng nghiệp vụ. Hiện tại nhiều controller vẫn xử lý logic nghiệp vụ.
3. **Chia nhỏ các Component lớn trên Frontend:**
   * **Hành động:** Nếu một component như Dashboard.js hoặc UserManagerPage.js có file mã nguồn dài hơn 200-300 dòng và chứa nhiều useState, useEffect, hãy xem xét tách các phần giao diện và logic liên quan ra thành các component con. Điều này tăng **High Cohesion** cho từng component.

# 2. Phân tích mức độ đáp ứng nguyên lý SOLID

## 2.1. Giới thiệu lý thuyết

SOLID là bộ 5 nguyên lý thiết kế hướng đối tượng giúp phần mềm trở nên dễ hiểu, linh hoạt và dễ bảo trì.

* **S - Single Responsibility Principle (SRP):** Mỗi class/module/function chỉ nên chịu một trách nhiệm duy nhất.
* **O - Open/Closed Principle (OCP):** Có thể mở rộng (open for extension) nhưng đóng với việc sửa đổi (closed for modification).
* **L - Liskov Substitution Principle (LSP):** Các đối tượng của lớp con có thể thay thế các đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình.
* **I - Interface Segregation Principle (ISP):** Client không nên bị buộc phải phụ thuộc vào các interface mà chúng không sử dụng.
* **D - Dependency Inversion Principle (DIP):** Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả hai nên phụ thuộc vào một trừu tượng (abstraction).

## 2.2. Phân tích trong thiết kế hiện tại

### 2.2.1. Trước khi thêm yêu cầu mở rộng (Thiết kế hiện tại)

* **S - Single Responsibility Principle (SRP) - Tuân thủ TỐT:**
  + **Backend:** Mỗi file trong controllers, services, models đều có một trách nhiệm duy nhất. Ví dụ: authController.js chỉ lo về đăng nhập/đăng ký, verify-token.js chỉ lo xác thực token.
  + **Frontend:** LoginForm.js chỉ là form đăng nhập. TaskChart.js chỉ vẽ biểu đồ công việc. useTaskSummaryData.js chỉ lấy dữ liệu tổng quan công việc. Thiết kế này tuân thủ rất tốt SRP.
* **O - Open/Closed Principle (OCP) - Tuân thủ KHÁ:**
  + **Điểm tốt:**
    - **Middleware:** Để thêm một vai trò mới (ví dụ: "TA"), ta chỉ cần thêm logic vào check-role.js mà không cần sửa đổi các controllers. Nếu check-role được viết tốt (ví dụ: nhận một mảng các vai trò được phép), nó sẽ rất dễ mở rộng.
    - **Routes/Services:** Để thêm một tính năng mới, ví dụ "Quản lý Sprint", ta chỉ cần tạo sprint-routes.js, sprintController.js, sprintService.js mà không cần sửa các file hiện có.
* **L - Liskov Substitution Principle (LSP) - Tuân thủ:**
  + Nguyên lý này chủ yếu áp dụng cho kế thừa trong OOP. Trong JavaScript/React, nó có thể được hiểu qua component composition.
  + Ví dụ: ClassCard.js và InstructorClassCard.js. Nếu InstructorClassCard được thiết kế để có thể thay thế cho ClassCard ở mọi nơi mà không gây lỗi (ví dụ, nó chấp nhận cùng các props cơ bản), thì nó tuân thủ LSP.
* **I - Interface Segregation Principle (ISP) - Tuân thủ TỐT:**
  + **Backend:** Các file route riêng biệt (user-routes.js, class-routes.js) hoạt động như các "interface" chuyên biệt. Client (frontend) có thể chọn chỉ sử dụng các endpoint từ class-routes.js mà không cần biết đến sự tồn tại của các routes khác.
  + **Frontend:** Việc chia nhỏ các service (auth-service.js, class-service.js) là một ví dụ tuyệt vời về ISP. Một component chỉ cần import class-service sẽ không bị phụ thuộc vào các hàm của user-service.
* **D - Dependency Inversion Principle (DIP) - Tuân thủ KHÁ:**
  + **Điểm tốt (Frontend):** Đây là điểm sáng của thiết kế. Các component cấp cao (Dashboard.js) không phụ thuộc trực tiếp vào axios hay fetch (module cấp thấp). Thay vào đó, chúng phụ thuộc vào một "trừu tượng" là các custom hooks (use...Data) và các hàm trong services. Đây là một ứng dụng rất thực tế và hiệu quả của DIP trong React.

### 2.2.2. Sau khi thêm yêu cầu mở rộng

Giả sử có các yêu cầu mở rộng sau:

1. **REQ-EXT-1:** Thêm vai trò mới "Trợ giảng" (TA) có một số quyền của Giảng viên (GV) và một số quyền của Sinh viên (SV).
2. **REQ-EXT-2:** Tích hợp với một nền tảng khác ngoài GitHub, ví dụ GitLab, để lấy dữ liệu contribution.

* **Phân tích tác động:**
  + **SRP:** Vẫn được duy trì tốt. Ta sẽ tạo các module mới cho GitLab (gitlabController, gitlabService).
  + **OCP:**
    - **REQ-EXT-1 (Vai trò TA):** Thiết kế hiện tại sẽ bị thách thức. Nếu check-role.js dùng if-else, ta phải *sửa đổi* nó. Nếu logic phân quyền nằm rải rác trong các services, ta phải sửa nhiều nơi.
    - **REQ-EXT-2 (GitLab):** githubController và githubService đang được viết chuyên cho GitHub. Để hỗ trợ GitLab, ta phải tạo gitlabController, gitlabService,... và có thể sẽ có nhiều code bị trùng lặp. Đây là dấu hiệu vi phạm OCP.
  + **ISP, LSP:** Ít bị ảnh hưởng.
  + **DIP:**
    - **REQ-EXT-2 (GitLab):** contributionService (cấp cao) hiện đang phụ thuộc trực tiếp vào githubService (cấp thấp). Khi thêm GitLab, contributionService sẽ phải chứa logic if (source === 'github') { ... } else if (source === 'gitlab') { ... }. Điều này vi phạm cả OCP và DIP.

## 2.3. Đề xuất cải thiện để đáp ứng SOLID tốt hơn

### 2.3.1. Cải thiện OCP/DIP cho việc tích hợp Git (REQ-EXT-2)

**Vấn đề:** Phụ thuộc cứng vào githubService.

**Giải pháp (Strategy Pattern):** Tạo một "interface" trừu tượng cho dịch vụ Git. Trong JS, đây có thể là một class cha hoặc một quy ước về các phương thức.

// services/git/IGitService.js (quy ước)

class IGitService {

getContributions(repoUrl, user) {

throw new Error("Method not implemented!");

}

getCommitActivity(repoUrl) {

throw new Error("Method not implemented!");

}

}

Triển khai các phiên bản cụ thể.

// services/git/githubService.js

class GitHubService extends IGitService {

getContributions(...) { /\* logic for GitHub API \*/ }

// ...

}

// services/git/gitlabService.js

class GitLabService extends IGitService {

getContributions(...) { /\* logic for GitLab API \*/ }

// ...

}

Tạo một GitServiceFactory để quyết định dùng service nào dựa trên input.

// services/git/gitServiceFactory.js

const GitHubService = require('./githubService');

const GitLabService = require('./gitlabService');

function getGitService(project) {

if (project.source.includes('github.com')) {

return new GitHubService();

}

if (project.source.includes('gitlab.com')) {

return new GitLabService();

}

throw new Error("Unsupported Git provider");

}

contributionService (cấp cao) sẽ sử dụng factory này.

// services/contributionService.js

const gitServiceFactory = require('./git/gitServiceFactory');

class ContributionService {

async analyzeContributions(project) {

const gitService = gitServiceFactory.getGitService(project); // DIP

const contributions = await gitService.getContributions(...);

// ...

}

}   
**Lợi ích:** contributionService giờ phụ thuộc vào "trừu tượng" (interface của service Git). Khi muốn thêm Bitbucket, ta chỉ cần tạo BitbucketService.js và cập nhật factory mà **không cần sửa** contributionService. Điều này tuân thủ OCP và DIP.

### 2.3.2. Cải thiện OCP cho việc quản lý vai trò (REQ-EXT-1)

**Vấn đề:** Logic phân quyền có thể bị cứng.

**Giải pháp:** Sử dụng một cấu trúc dữ liệu để định nghĩa quyền hạn thay vì if-else.  
// middleware/permissions.js

const ROLES = {

ADMIN: 'admin',

INSTRUCTOR: 'instructor',

STUDENT: 'student',

TA: 'ta' // Thêm vai trò mới ở đây

};

const PERMISSIONS = {

[ROLES.ADMIN]: ['manage\_users', 'manage\_all\_classes'],

[ROLES.INSTRUCTOR]: ['create\_class', 'manage\_own\_classes', 'grade\_projects'],

[ROLES.STUDENT]: ['view\_class', 'submit\_project'],

[ROLES.TA]: ['manage\_own\_classes', 'grade\_projects'] // Cấp quyền cho TA

};

// middleware/check-role.js

function checkPermission(permission) {

return (req, res, next) => {

const userRole = req.user.role;

if (PERMISSIONS[userRole] && PERMISSIONS[userRole].includes(permission)) {

return next();

}

return res.status(403).send('Forbidden: Insufficient permissions');

};

}

// routes/class-routes.js

// router.post('/', verifyToken, checkPermission('create\_class'), classController.create);

**Lợi ích:** Khi thêm vai trò "TA", ta chỉ cần thêm định nghĩa trong permissions.js. Khi thêm một quyền mới, ta cũng chỉ cần thêm vào đó. checkPermission không bao giờ cần phải sửa đổi. Đây là một ví dụ điển hình của việc tuân thủ OCP.

## 2.4. Kết luận và Hành động

Thiết kế tổng thể của dự án là rất tốt, hiện đại và tuân thủ phần lớn các nguyên tắc thiết kế phần mềm quan trọng. Các điểm mạnh lớn nhất là sự phân tách rõ ràng giữa frontend và backend, và việc áp dụng các pattern tốt như service-layer, custom hooks.

**Các bước cần thực hiện để cải thiện:**

1. **Ngắn hạn (Refactor ngay):**
   * Hợp nhất groupController.js và groupsController.js.
   * Review các component lớn và chia nhỏ nếu cần.
2. **Dài hạn (Chuẩn bị cho mở rộng):**
   * Áp dụng **Strategy Pattern** và **Factory** cho các dịch vụ bên ngoài như GitHub/GitLab để tuân thủ OCP và DIP.
   * Xây dựng lại hệ thống phân quyền dựa trên cấu hình (config-based) thay vì logic if-else để dễ dàng mở rộng vai trò và quyền hạn.