ĐẠI HỌC BÁCH KHOA HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Học phần: Phát triển phần mềm theo chuẩn kỹ năng ITSS

Đề tài: Hệ thống quản lý phòng tập Gym

GVHD: ThS. Nguyễn Mạnh Tuấn

Nhóm 29

1. Đỗ Mạnh Phương 20225660
2. Đàm Thanh Bách 20225600
3. Nguyễn Phúc Anh 20225784
4. Lê Đồng Cảnh Phú 20225755
5. Lê Thị Ngọc Thảo 20225673

Hà Nội, tháng 6 năm 2025

NGUYÊN LÝ THIẾT KẾ

7.1 Áp dụng Design Concepts

7.1.1 Coupling

-Couling là một cách để miêu tả sự phụ thuộc lẫn nhau giữa các modules, hay còn gọi là tính liên kết giữa các modules.

-Để có một bản thiết kế tốt thì cần phải thiết kế sao cho coupling lỏng để khi có sự thay đổi ở một module thì ảnh hưởng ít nhất có thể đến các module khác. Coupling càng lỏng lẻo, càng tốt.

1.1 Content coupling

- Đây là vi phạm coupling ở mức cao nhất được định nghĩa là khi một component có thể truy cập trực tiếp vào hoạt động bên trong của một component khác ví dụ như truy xuất data, thay đổi data của một component khác. Hay nói đơn giản là việc một component sử dụng code của một component khác.

-Đối với tiêu chí này thì nhóm em không vi phạm Content Coupling. Bởi vì dự án bọn em truy cập đến các method ở class khác thông qua Interface của class đó, ngoài ra các thuộc tính (attributes) cũng được để thành private và có hàm getter setter để truy cập vào những thuộc tính này.

1.2 Common Coupling (ko)

- Đây là trường hợp 2 module cùng chia sẻ chung dữ liệu, 2 global structure có thể cùng vào chỉnh sửa, truy cập hoặc có chung những khối mã nguồn thì sẽ vi phạm common coupling.

- Tuy nhiên, object-oriented programming không có common data. Tất cả data đều thuộc về lớp. Ví dụ khi dùng C, bạn có data structure khai báo là global, sau đó sẽ có một số phương thức ghi vào structure, các phương thức khác đọc từ đó, trường hợp này sẽ vi phạm common coupling. Còn trong Object Oriented, không có bất cứ data nào mà không thuộc về lớp. Do đó, project hiện tại (C#) không vi phạm common coupling.

- Design ko vi phạm Coupling này.

1.3 Control Coupling

- Một module vi phạm control coupling khi nó truyền tham số điều khiển cho các module khác thông qua việc gọi method. Điều này không tốt vì thành phần được

gọi sẽ biết được cấu trúc bên trong thành phần gọi và khi cấu trúc này bị thay đổi thì thành phần được gọi sẽ phải thay đổi theo.

-Nhóm em thiết kế các chức năng CRUD bằng các phương thức riêng biệt, chứ không gộp lại thành một phương thức to, vì vậy nhóm em không vi phạm tiêu chí này.

1.4 Stamp coupling

-Stamp coupling xảy ra khi tham số truyền vào cho module là thừa, việc xử lý có thể chỉ cần một vài trường dữ liệu của object nhưng chúng ta truyền nguyên object vào. Chú ý rằng, stamp coupling là mức có thể chấp nhận được.

- Do bọn em có thời gian để tiến hành việc code nên chúng em chia tách các method sao cho các method chỉ sử dụng đúng các tham số truyền vào. Nếu cần truyền các tham số khác nhau thì sẽ tạo các method khác nhau.

1.5 Data coupling

- Là coupling ở mức thấp nhất, khi mà các module tương tác với nhau chỉ qua tham số truyền vào.

-Nhóm em đã đạt được tiêu chí này ở các module như AuthController và AuthService.

1.6 Uncoupled

- Các module là uncoupled khi chúng tồn tại độc lập và riêng rẽ, không có mỗi quan hệ với các module khác.

- Design của bọn em không vi phạm uncoupled.

7.1.2 Cohesion

Cohension là thể hiện mức độ liên kết ở mức chi tiết giữa các thành phần ở bên trong một module. Cohension càng cao thì thiết kế càng tốt và ngược lại

2.1 Coincidental cohesion

- Các subcomponent nằm trong một component một cách ngẫu nhiên, không liên quan đến mục tiêu thể hiện chung của component.

- Rõ ràng, chúng ta có thể thấy loại cohesion này trong class Utils trong package utils

- Ngoài những trường hợp trên thì Design không vi phạm cohesion trên.

2.2 Logical cohesion

- Các thành phần trong 1 module có liên quan đến nhau nhưng về mặt logic chứ không phải chức năng.

- Quan sát lại phần Control Coupling bên trên, nếu chúng ta chia đoạn code thành 2 methods và đưa chúng vào trong cùng một lớp, chúng ta có thể đối mặt với logical cohesion. Do đó, chúng ta cần xem xét đưa chúng vào trong những class hoặc package khác.

- Do sử dụng Service-Repository pattern nên chúng em đã tránh đc cohesion này.

2.3 Informational cohesion

- Các operation có tính độc lập (có input và output riêng nhưng chúng có thể thao tác trên một tập dữ liệu chung là attribute của lớp đó)

- Nhóm đạt được mức cohesion này.

7.2 Áp dụng Design Principles SOLID

Nguyên tắc SOLID là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc thiết kế hướng đối tượng, giúp cho lập trình viên viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain. 5 nguyên tắc đó bao gồm: Single responsibility principle, Open-Closed principle, Liskov substitution principle, Interface segregation principle, và Dependency inversion principle.

Áp dụng SOLID vào thiết kế của nhóm :

Nguyên tắc số 1: Single-responsibility principle (SRP):

Mỗi lớp chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể.

Theo nguyên lý này, một class chỉ nên giữ một trách nhiệm duy nhất. Một class nếu có quá nhiều chức năng thì sẽ trở nên khó đọc, các chức năng không quá rạch ròi dẫn đến khó khăn trong việc đọc hiểu code.

Ví dụ trong controller chỉ chứa các method thực thi các nhiệm vụ như: create, read, update, delete,..

Nguyên tắc này đã được áp dụng trong code của nhóm em như sau:

STT	Related modules	Mô tả	Cải thiện
1	Auth Controller	Trong AuthController, có các chức năng: Đăng nhập, đăng ký, yêu cầu kích hoạt tài khoản, đổi mật khẩu, yêu cầu quên mật khẩu, thiết lập lại mật khẩu	Tách các chức năng liên quan đến việc kích hoạt và quản lý mật khẩu ra 2 controller riêng biệt đó là ActivationController và PasswordController
2	AuthService	Trong AuthService chỉ có các chức năng liên quan đến xác thực người dùng như: Đăng nhập, đăng ký	
3	ActivationService	Trong ActivationService chỉ có các chức năng liên quan đến kích hoạt tài khoản như: Lấy mã kích hoạt cho tài khoản, kích hoạt tài khoản	
4	PasswordService	Trong password service có những chức năng như: Quên mật khẩu, gửi yêu cầu thiết lập lại mật khẩu, thiết lập lại mật khẩu	

Nguyên tắc số 2: Open-closed principle (OCP):

Theo nguyên lý này, mỗi khi ta muốn thêm chức năng cho chương trình thì nên viết class mới mở rộng từ class cũ (như bằng cách kế thừa) chứ không nên sửa đổi class cũ. Việc viết class mới mở rộng từ class cũ này dẫn tuy có thể sẽ phát sinh nhiều class, nhưng có điểm lợi là chúng ta sẽ không cần phải test lại các class cũ nữa, mà chỉ tập trung vào test các class mới vừa tạo ra. Thông thường để mở rộng thêm chức năng thì ta cần phải viết thêm code, vậy để thiết kế ra một module

có thể dễ dàng mở rộng nhưng lại phải hạn chế sửa đổi code thì ta cần tách những phần dễ thay đổi ra khỏi phần ít thay đổi sao cho đảm bảo không ảnh hưởng đến phần còn lại.

Như việc có thể thêm bất kỳ một Controller mới nào vào trong Project, nhưng tất cả các Controller đều được Extend duy nhất từ class gốc là BaseController.

Nguyên tắc này được nhóm em áp dụng như sau:

STT	Related modules	Mô tả	Cải thiện
1	Controller	Các controller như Auth, User,Course,Device, v.v đều có đủ 4 chức năng CRUD và các chức năng liên quan đến HttpRequest	Tạo một controller cha có các phương thức abstract về HttpRequest

Nguyên tắc số 3: Liskov substitution principle (LSP):

Các đối tượng kiểu class con có thể thay thế các đối tượng kiểu class cha mà không gây ra lỗi.

Cần chú ý không nên cho những phương thức không đặc trưng, không mang tính khái quát vào lớp cha để tránh trường hợp lớp con khi kế thừa phải những phương thức đó. Với những phương thức kiểu này thì nên sử dụng interface và cho những class muốn sử dụng phương thức đó implements interface đó.

Trong một chương trình có rất nhiều các class thừa kế từ các class cha là thư viện nhưng không bị biến đổi tính đúng đắn của chương trình.

Nguyên tắc này được nhóm em áp dụng như sau:

STT	Related modules	Mô tả	Cải thiện

1	UserBaseModel, AdminModel, ManagerModel, MemberModel, TrainerModel	Class con AdminModel, ManagerModel, MemberModel, TrainerModel đều được kế thừa từ class cha UserBaseModel và thay thế class cha UserBaseModel trong các phương thức khác	

Nguyên tắc số 4: Interface-segregation principle (ISP):

Thay vì dùng một interface lớn thì nên tách thành nhiều interface nhỏ với từng mục đích cụ thể.

Nếu ta chỉ có duy nhất một interface, ở đó nhét toàn bộ phương thức, thì khi các class muốn implement sẽ phải define lại toàn bộ abstract method trong interface đó. Điều này khiến tốn thời gian và lãng phí tài nguyên không cần thiết. Vậy nên cần chia thành nhiều interface nhỏ hơn, gồm các method liên quan với nhau, thì việc implement sẽ dễ dàng hơn.

Nguyên tắc này được nhóm em áp dụng như sau:

STT	Related modules	Mô tả	Cải thiện
1	Các interface Service	Các interface service định nghĩa trước các method mà service sẽ được sử dụng	

Nguyên tắc số 5: Dependency-inversion principle (DIP):

Các module cấp cao không nên phụ thuộc vào các modules cấp thấp, mà nên phụ thuộc vào abstraction. Chi tiết nên phụ thuộc vào abstraction, abstraction không nên phụ thuộc vào chi tiết.

Những thành phần trong chương trình chỉ nên phụ thuộc vào những cái trừu tượng (abstraction). Những thành phần trừu tượng không nên phụ thuộc vào các thành phần mang tính cụ thể. Những cái trừu tượng (abstraction) là những cái ít thay đổi và biến động, tập hợp những đặc tính chung nhất, bao quát nhất của những cái cụ thể. Những cái cụ thể dù khác nhau thế nào đi nữa đều mang khuôn mẫu mà cái trừu tượng đã định ra. Việc phụ thuộc vào cái trừu tượng sẽ giúp chương trình linh động và thích ứng tốt với các sự thay đổi diễn ra liên tục.

Nguyên tắc này được nhóm em áp dụng như sau:

STT	Related modules	Mô tả	Cải thiện
1	Các service trong BE ví dụ: AuthService	AuthService có chức năng là cung cấp những Service liên quan đến việc xác thực người dùng, AuthService khi muốn sử dụng method của các class khác thì sẽ gọi đến các InterfaceService. Điều này để tránh việc 1 class phụ thuộc quá chặt chẽ vào các class khác	

* Nhờ việc áp dụng nguyên lý SOLID, code được tổ chức logic hơn, các method, class cũng được tổ chức rõ ràng, chính xác hơn.

+ Tính rõ ràng, dễ hiểu: Khi áp dụng SOLID vào thiết kế, nhất là trong khi làm việc nhóm, ta sẽ tạo ra các hàm dễ hiểu hơn, giúp code clean và dễ đọc hơn, từ đó giúp cho các bạn khác trong nhóm đọc hiểu code của nhau dễ hơn.

+ Tính dễ thay đổi: SOLID giúp tạo ra các module, class rõ ràng, mạch lạc, mang tính độc lập cao. Do vậy khi có sự yêu cầu thay đổi, mở rộng, ta cũng dễ dàng thực hiện việc thay đổi hơn.
+ Tính tái sử dụng: Nhờ tính rõ ràng, dễ hiểu và dễ thay đổi, mà nếu hệ thống khác cần sử dụng chức năng tương tự, thì nhờ SOLID cũng giúp tái sử dụng module dễ hơn.