Về phương thức học của em thì khi gặp 1 vấn đề hay 1 bài toán kinh điển của dev mà em chưa được tiếp cận, thay vì lên gg research thì em sẽ lên tiktok để research, em xem 1 vài video thì nó sẽ cụ thể hơn thay vì tìm 1 đống kiến thức miên man ở trên gg, youtube, AI. Vì vẫn chưa thạo kĩ năng đọc docx ở gg mạnh nên việc vừa xem video vừa nghe phân tích và giải thích đối với em là tốt hơn.

Còn khi tìm hiểu về 1 kiến thức mới, đầu tiên em sẽ lên đọc qua lý thuyết của nó, sau đó em sẽ chat với AI để hỏi về case study và best practice rồi đọc kĩ hơn về 5W1H, cùng với nhìn qua code mẫu và tự tư duy, nếu vẫn chưa hiểu thì sẽ hỏi tiếp AI, hỏi sâu hết 1 vấn đề này rồi qua vấn đề khác. Sau khi nắm được hết được kiến thức thì em sẽ hỏi thêm các câu hỏi mở rộng.

Dưới đây là phần em tổng hợp lại:

1. Association & Aggregation & Composition

Association: Hai đối tượng độc lập. Ví dụ: `Manager` và `Project`. Manager có thể quản lý Project, nhưng nếu Manager nghỉ việc, Project vẫn tồn tại (và ngược lại).

Aggregation: "Cái toàn thể" chứa "Cái bộ phận", nhưng bộ phận có thể tồn tại độc lập. Ví dụ: `Library` và `Book`. Nếu Library bị giải thể, Book vẫn có thể đem sang thư viện khác.
Composition: Bộ phận là một phần máu thịt của toàn thể. Ví dụ: `Room` bên trong `House`. Nếu House bị sập, các Room biến mất theo.

2. Coupling & Cohesion

Cohesion: Một Class/Module nên tập trung làm **một việc và làm thật tốt**. Cohesion cao giúp code dễ hiểu, dễ test.
Coupling: Các Class biết về nhau càng ít càng tốt.

- *Tight Coupling:* Sửa Class A làm Class B lỗi.
- *Loose Coupling:* Thay đổi implementation của Class A không ảnh hưởng gì đến B nhờ giao tiếp qua Interface.

3. Composition over Inheritance

Tại sao kế thừa (Inheritance) thường là "cái bẫy"?

- **Fragile Base Class:** Khi thay đổi class cha, hàng loạt class con có thể bị hỏng hành vi mà không lường trước được.
- **White-box reuse:** Kế thừa làm lộ chi tiết bên trong của cha cho con, vi phạm tính đóng gói.
- **Giải pháp:** Dùng Composition.

4. Interface over Abstract Class

Với sự xuất hiện của `default` và `private` methods trong Interface, ranh giới đã mờ dần, nhưng bản chất vẫn khác biệt:

- **Interface (What it can do):** Định nghĩa một **năng lực/hành vi**. Một Class có thể có nhiều năng lực (Multiple inheritance of behavior).
- **Abstract Class (What it is):** Định nghĩa một **bản sắc/chủng loại**. Dùng khi các class con thực sự chia sẻ chung mã nguồn và trạng thái (fields).
  Best practice: Luôn bắt đầu bằng Interface. Chỉ chuyển sang Abstract Class khi thấy mình đang copy-paste quá nhiều code logic giống nhau giữa các implementation.

5. SOLID

Nguyên lý	Tóm tắt tư duy	Case study thực tế
SRP (Single Responsibility)	Một class chỉ có một lý do để thay đổi.	Đừng để class `User` vừa chứa data, vừa có hàm `saveToDB()`, vừa có hàm `sendEmail()`.
OCP (Open/Closed)	Mở rộng tính năng bằng cách viết code mới, không sửa code cũ.	Dùng Interface/Strategy. Khi thêm loại phí ship mới, chỉ cần tạo Class mới, không sửa `if-else` trong hàm tính phí.
LSP (Liskov Substitution)	Class con không được làm thay đổi kỳ vọng của Class cha.	Nếu `Bird` có hàm `fly()`, thì `Ostrich` (đà điểu) không nên kế thừa `Bird` vì đà điểu không bay được.
ISP (Interface Segregation)	Thà nhiều Interface nhỏ còn hơn một Interface lớn.	Đừng bắt `Worker` interface phải có hàm `code()` nếu `Janitor` (tạp vụ) cũng implement interface đó.
DIP (Dependency Inversion)	Đừng phụ thuộc vào Concrete Class, hãy phụ thuộc vào Abstraction.	`BusinessLogic` không nên gọi trực tiếp `MySQLDatabase`. Nó nên gọi thông qua `IDatabase`.

Github: [VuDaiVillage/fsa-fr-oop-advanced]