**Phần 1. JWT**

1. JWT có thể được sử dụng cho mục đích gì?

* **Xác thực người dùng**: JWT thường được sử dụng để xác thực trong các hệ thống web và API.
* **Ủy quyền (Authorization)**: Sau khi người dùng đăng nhập, JWT được dùng để cấp quyền truy cập vào tài nguyên.
* **Trao đổi thông tin an toàn**: JWT có thể chứa dữ liệu đã được mã hóa và xác minh tính toàn vẹn bằng chữ ký số.
* Cung cấp thông tin giữa các service trong các kiến trúc MicroServices

1. Cấu trúc JWT gồm những gì?

JWT bao gồm ba phần, được phân tách bởi dấu chấm (.):

* Header: Chứa loại token (JWT) và thuật toán mã hóa (HS256, RS256, ...).
* Payload: Chứa các thông tin (claims) như user ID, thời gian hết hạn.
* Signature: Chữ ký số để đảm bảo tính toàn vẹn của token.

1. Hãy mô tả việc xác thực bằng JWT theo cách bạn hiểu?

* Người dùng đăng nhập bằng tài khoản và mật khẩu.
* Máy chủ xác thực thông tin đăng nhập, tạo JWT chứa thông tin người dùng.
* Người dùng gửi JWT này trong các request tiếp theo (thường qua header Authorization).
* Máy chủ kiểm tra tính hợp lệ của JWT bằng cách xác minh chữ ký.
* Nếu JWT hợp lệ, máy chủ cho phép truy cập tài nguyên.

**Phần 2. Architecture**

1. **Ưu – nhược điểm các mô hình**

**Phần 1: Kiến trúc đơn khối - Kiến trúc phân tán**

1. Monolithic Architecture
2. Ưu điểm:

* Design: thiết kế đơn giản chỉ 1 code base duy nhất với nhiều function phụ thuộc lẫn nhau
* Development: khi một ứng dụng được xây dựng với một code base, việc phát triển (develop) sẽ dễ dàng hơn.
* Deployment: chỉ thực thi (executable) 1 file hoặc 1 directory nên việc triển khai (deployment) dễ dàng hơn.
* Debugging: tất cả code ở cùng 1 nơi nên việc the dõi yêu cầu (request) và tìm sự cố (issue) dễ dàng hơn
* Testing: Có thể test toàn bộ ứng dụng một cách dễ dàng và tìm kiếm lỗi trong môi trường đồng nhất.
* Performance - Hiệu suất cao trong các ứng dụng ít phức tạp: Các ứng dụng ít phức tạp có thể chạy hiệu quả với kiến trúc monolithic do giảm thiểu giao tiếp mạng và độ trễ

1. Nhược điểm

* Slower development speed: một ứng dụng lớn, nguyên khối khiến quá trình phát triển trở nên phức tạp và chậm hơn.
* Scalability: không thể mở rộng các thành phần riêng lẻ.
* Reliability: nếu có lỗi trong bất kỳ mô-đun nào, lỗi đó có thể ảnh hưởng đến tính khả dụng của toàn bộ ứng dụng.
* Barrier to technology adoption (Rào cản đối với việc áp dụng công nghệ): bất kỳ thay đổi nào trong khuôn khổ hoặc ngôn ngữ đều ảnh hưởng đến toàn bộ ứng dụng, khiến việc thay đổi thường tốn kém và mất thời gian.
* Lack of flexibility (Thiếu tính linh hoạt): bị hạn chế bởi các công nghệ đã được sử dụng trong khối nguyên khối.
* Deployment: một thay đổi nhỏ đối với một ứng dụng nguyên khối đòi hỏi

phải triển khai lại toàn bộ hệ thống.

1. Distributed Architecture
2. Ưu điểm

* Khả năng chịu lỗi và tin cậy: nếu một dịch vụ bị lỗi, các dịch vụ khác có thể tiếp tục thực hiện các yêu cầu dịch vụ.
* Khả năng thích ứng: Chức năng của ứng dụng được chia thành các đơn vị phần mềm được triển khai riêng biệt, dễ dàng xác định vị trí và áp dụng thay đổi hơn, phạm vi thử nghiệm được giảm xuống chỉ còn dịch vụ bị ảnh hưởng và rủi ro triển khai giảm đáng kể vì thường chỉ triển khai dịch vụ bị ảnh hưởng.
* Có khả năng mở rộng: vì chứa một tập hợp các máy độc lập nên có thể mở rộng theo chiều ngang dễ dàng.
* Độ trễ thấp: do có nhiều máy chủ và có khả năng phản hồi đến gần người dùng hơn để giải quyết truy vấn, do đó mất rất ít thời gian để giải quyết truy vấn của người dùng.

1. Khuyết điểm

* Tăng đáng kể độ phức tạp của kiến trúc: quản lý nhiều chương trình nhỏ hơn rõ ràng phức tạp hơn so với một ứng dụng monolithic.
* Yêu cầu phải deploy tự động.
* Giảm tổng thể về hiệu suất: kiến trúc microservices thường dẫn đến mức tiêu tốn tài nguyên cao hơn so với kiến trúc monolithic có cùng quy mô.
* Các microservice đòi hỏi nhiều bộ nhớ hơn, tốn nhiều chu kỳ CPU hơn và cần băng thông mạng lớn hơn.
* Quản lý các giao dịch phân tán, tính nhất quán cuối cùng, quản lý quy trìnhlàm việc, xử lý lỗi, đồng bộ hóa dữ liệu
* Khó khắc phục sự cố và gỡ lỗi: xác định và giải quyết các vấn đề trong kiến trúc microservices cũng phức tạp tương tự như việc tổng hợp các file theo dõi và log. Khi một yêu cầu thất bại đi qua nhiều microservice được lưu trữ trong các môi trườnchạy riêng biệt, việc xác định chính xác vị trí và nguyên nhân của sự cố có thể rất khó khăn.
* Chi phí cao:

• Tăng tổng dung lượng bộ nhớ

• Tài nguyên bị trùng lặp khi sử dụng nhiều container hoặc máy ảo.

• Tiêu tốn băng thông khi gọi các dịch vụ web RESTful …

**Phần 2. Architecture Patterns**

*1. Layered Architecture*

* ***Ưu điểm:*** tách biệt logic giữa các tầng
* ***Khuyết điểm:*** kiến trúc phân lớp vẫn có một yếu điểm dễ thấy là logic phải được truyền theo thức tự các tầng, bất kể tầng đó có xử lý dữ liệu hay không. Điều này gây ra sự lãng phí tài nguyên.
* **Trường hợp nên dùng kiến trúc phân lớp:** Nếu dự án có những hạn chế nhất định về thời gian và ngân sách, kiến trúc phân lớp là một sự lựa chọn thích hợp vì các lý do chính sau:

• Thuộc loại kiến trúc monolithic nên không phức tạp

• Dựa trên mức độ phổ biến của kiến trúc phân lớp, hầu hết developer điều quen thuộc, việc vận hành, phát triển dự án trở nên dễ dàng hơn.

• Kiến trúc phân lớp thuộc nhóm kiến trúc phân vùng kỹ thuật (Technical Partitioning) nên việc sử dụng kiến trúc này phù hợp với nhiều tổ chức khác nhau do các nhóm được chia thành nhóm chuyên trách kỹ thuật như nhóm FE, nhóm BE, nhóm database, ...

*2.Client – Server Architecture*

a. Ưu điểm

* Quản lý tập trung trên một Server, kiến trúc này đơn giản hóa việc bảo trì, cập nhật và quản lý bảo mật. Người quản trị có thể giám sát và quản lý dữ liệu, áp dụng các bản cập nhật và thực thi chính sách bảo mật hiệu quả từ một vị trí duy nhất.
* Khả năng mở rộng: khi số lượng Client tăng lên, có thể thêm Server hoặc mở rộng dung lượng Servẻ hiện có mà không làm thay đổi đáng kể kiến trúc hệ thống tổng thể.
* Tối ưu hóa tài nguyên: mô hình này cho phép phân bổ tài nguyên được tối ưu hóa. Server được thiết kế để xử lý chuyên sâu và lưu trữ dữ liệu lớn, trong khi Client được tối ưu hóa cho các tương tác và yêu cầu của người dùng. Sự tách biệt này đảm bảo sử dụng hiệu quả tài nguyên hệ thống.
* Độ tin cậy và tính khả dụng: với cơ sở hạ tầng máy chủ mạnh, hệ thống Client-Server có thể đảm bảo độ tin cậy và tính khả dụng cao.

b.Nhược điểm

* Kiểm soát tập trung có thể dẫn đến khả năng xảy ra lỗi cao hơn. Khi nhiều Client gửi yêu cầu đồng thời đến Server, Server có thể quá tải và làm chậm hiệu suất. Điều này cũng có thể dẫn đến lỗi Server, khiến toàn bộ hệ thống ngừng hoạt động.
* Server mạnh hơn Client, nghĩa là Server đắt hơn. Server cũng yêu cầu người có kiến thức về mạng và cơ sở hạ tầng để quản lý hệ thống.
* Kiến trúc Client-Server dễ bị tấn công Từ chối dịch vụ (DoS) vì số lượng Server thường nhỏ hơn số lượng Client.
* Các gói dữ liệu có thể bị thay đổi trong quá trình truyền, dẫn đến khả năng mất thông tin hữu ích.

1. *Pipeline Architecture*

* ***Ưu*** điểm: dễ dàng thay đổi/bảo trì/dùng lại từng filter của hệ thống, phù hợp với nhiều hoạt động nghiệp vụ, dễ dàng nâng cấp bằng cách thêm filter mới.
* Khuyết điểm: 2 filter kề nhau cần tuân thủ định dạng dữ liệu chung
* Tình huống **nên dùng**: trong các ứng dụng xử lý dữ liệu mà dữ liệu nhập cần được xử lý bởi nhiều công đoạn khác nhau và có tính độc lập cao trước khi tạo ra kết quả cuối cùng.

1. *Event-Driven Architecture – EDA*
2. Ưu điểm

* Tính linh hoạt cao: Các dịch vụ không phụ thuộc chặt chẽ vào nhau, giúp hệ thống dễ mở rộng và bảo trì.
* Khả năng mở rộng tốt: Do xử lý sự kiện bất đồng bộ nên dễ dàng mở rộng theo chiều ngang (scale-out).
* Tốc độ phản hồi nhanh: Các sự kiện được xử lý ngay khi phát sinh, không cần chờ phản hồi từ các dịch vụ khác.
* Giảm tải cho hệ thống: Chỉ xử lý khi có sự kiện, tránh việc kiểm tra liên tục (polling).
* Tính **đàn hồi cao**: Hệ thống có thể tiếp tục hoạt động ngay cả khi một số dịch vụ bị lỗi, miễn là hàng đợi sự kiện (event queue) vẫn hoạt động.

1. Nhược điểm

* **Độ phức tạp cao**: Khó theo dõi và gỡ lỗi do hệ thống phi tập trung.
* Khó kiểm soát thứ tự xử lý: Do các sự kiện được xử lý bất đồng bộ, có thể xảy ra tình trạng xử lý không đúng thứ tự mong muốn.
* Cần cơ chế quản lý sự kiện tốt: Cần sử dụng message broker (Kafka, RabbitMQ) để đảm bảo tin nhắn không bị mất hoặc trùng lặp.
* Độ trễ có thể tăng: Nếu hàng đợi quá tải hoặc gặp sự cố, các sự kiện có thể bị xử lý chậm hơn mong đợi.

1. *Service-Oriented Architecture – SOA*
2. Ưu điểm

* **Tái sử dụng dịch vụ cao**: Các dịch vụ có thể được tái sử dụng trong nhiều ứng dụng khác nhau.
* **Khả năng tích hợp tốt**: Hỗ trợ giao tiếp với các hệ thống cũ (legacy systems) thông qua các giao thức tiêu chuẩn như SOAP, REST.
* **Dễ bảo trì và phát triển**: Các dịch vụ có thể được phát triển và triển khai độc lập.
* **Quản lý bảo mật tốt hơn**: Do sử dụng các giao thức như WS-Security để đảm bảo an toàn dữ liệu.

1. Nhược điểm

* **Hiệu suất thấp hơn do có quá nhiều giao tiếp qua mạng**: Các dịch vụ liên tục trao đổi dữ liệu với nhau, gây tốn băng thông.
* **Cấu trúc phức tạp**: Cần một **Enterprise Service Bus (ESB)** để quản lý luồng dữ liệu giữa các dịch vụ, làm tăng độ phức tạp.
* **Khó mở rộng**: Mô hình này chủ yếu tập trung vào việc tích hợp hơn là tối ưu khả năng mở rộng.
* **Độ trễ cao**: Việc xử lý dữ liệu thông qua ESB có thể gây chậm trễ.

1. *Microservices Architecture*
2. Ưu điểm

* **Dễ mở rộng (scalability)**: Có thể mở rộng từng service riêng lẻ thay vì toàn bộ hệ thống.
* **Triển khai nhanh và linh hoạt**: Các nhóm có thể phát triển, triển khai và cập nhật từng microservice mà không ảnh hưởng đến hệ thống chung.
* **Công nghệ đa dạng**: Có thể sử dụng nhiều công nghệ khác nhau cho từng service, giúp tối ưu hiệu suất.
* **Dễ bảo trì và nâng cấp**: Do các dịch vụ nhỏ gọn, độc lập, giúp giảm thiểu ảnh hưởng khi thay đổi hoặc sửa lỗi.
* **Tăng độ ổn định**: Nếu một service bị lỗi, hệ thống tổng thể vẫn có thể tiếp tục hoạt động.

1. Nhược điểm

* **Tăng độ phức tạp**: Việc quản lý nhiều service nhỏ thay vì một hệ thống lớn yêu cầu công cụ và kỹ năng quản lý tốt hơn.
* **Khó khăn trong giao tiếp giữa các service**: Cần sử dụng API Gateway hoặc message broker để quản lý giao tiếp giữa các service
* **Vấn đề đồng bộ dữ liệu**: Vì các microservice hoạt động độc lập, cần có giải pháp để đồng bộ dữ liệu giữa chúng
* **Cần DevOps mạnh**: Việc triển khai, giám sát và quản lý microservices yêu cầu các công cụ CI/CD, containerization (Docker, Kubernetes)
* **Chi phí vận hành cao**: Mỗi service có thể yêu cầu một database riêng, nhiều instance chạy song song, làm tăng chi phí hạ tầng.

1. Mô hình khái quát
2. Công nghệ

**Phần 3. Dessign Pattern**

1. **FactoryMethod**

**1. Ứng dụng thực tế trong hệ thống thanh toán**

* Khi bạn có nhiều phương thức thanh toán (thẻ tín dụng, PayPal, Momo, VNPay), bạn có thể sử dụng Factory Method để tạo các đối tượng xử lý thanh toán phù hợp.

2. **Quản lý Logger trong ứng dụng**

* Khi ứng dụng cần hỗ trợ nhiều loại logger (ConsoleLogger, FileLogger, DatabaseLogger), Factory Method giúp tạo logger phù hợp.

3. **Tạo đối tượng Database Connection**

* Khi bạn muốn kết nối với nhiều loại cơ sở dữ liệu như MySQL, PostgreSQL, MongoDB.

4. **Ứng dụng trong hệ thống gửi thông báo (Notification System)**

* Khi bạn cần gửi thông báo bằng Email, SMS hoặc Push Notification.

**Khi nào nên sử dụng Factory Method?**

* Khi hệ thống có nhiều lớp con với cách tạo đối tượng khác nhau.
* Khi muốn giấu logic khởi tạo đối tượng khỏi người dùng.
* Khi cần tạo đối tượng mà không làm thay đổi code của lớp client.

1. **Singleton**
2. **Quản lý kết nối cơ sở dữ liệu (Database Connection)**

* Đảm bảo chỉ có một kết nối tới cơ sở dữ liệu duy nhất trong toàn bộ ứng dụng, tránh tạo nhiều kết nối dư thừa gây tốn tài nguyên.

1. **Quản lý cấu hình hệ thống (Configuration)**

* Trong ứng dụng, có thể có một lớp lưu trữ thông tin cấu hình như API keys, database URLs, settings,... chỉ cần tải một lần duy nhất.

1. Quản lý Logger (Ghi log hệ thống)

* Đảm bảo rằng tất cả các phần của ứng dụng đều sử dụng cùng một logger duy nhất để ghi log.

1. Quản lý Bộ nhớ Cache

* Một ứng dụng có thể có một lớp lưu trữ dữ liệu cache (ví dụ: dữ liệu tạm thời từ database) để truy xuất nhanh hơn.

1. **Quản lý Luồng xử lý (Thread Pool)**

* Trong các ứng dụng đa luồng, có thể sử dụng Singleton để đảm bảo chỉ có một Thread Pool duy nhất.
* **🔥 Khi nào nên sử dụng Singleton?**
* ✔ Khi một lớp cần có **duy nhất một thể hiện** trong suốt vòng đời ứng dụng.  
  ✔ Khi muốn **tiết kiệm bộ nhớ** bằng cách sử dụng chung một đối tượng thay vì tạo nhiều đối tượng mới.  
  ✔ Khi muốn **đảm bảo tính nhất quán** trong quản lý dữ liệu hoặc tài nguyên (vd: database connection, logger).
* **🚨 Khi nào KHÔNG nên sử dụng Singleton?**
* ❌ Khi ứng dụng cần mở rộng dễ dàng, vì Singleton có thể gây khó khăn khi viết test và mở rộng code.  
  ❌ Khi cần sử dụng đa luồng, Singleton có thể gây lỗi nếu không được thiết kế thread-safe.

1. **State Pattern - Mẫu thiết kế trạng thái**

State Pattern là một mẫu thiết kế hành vi (Behavioral Pattern) cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong thay đổi. Điều này giúp mã dễ mở rộng và tránh việc sử dụng quá nhiều điều kiện if-else hoặc switch-case.

**📌 Các trường hợp nên sử dụng State Pattern**

* Khi một đối tượng có nhiều trạng thái khác nhau và mỗi trạng thái có hành vi riêng biệt.
* Khi muốn tránh sử dụng quá nhiều if-else hoặc switch-case trong mã nguồn.
* Khi trạng thái của đối tượng thay đổi theo thời gian và các hành vi tương ứng cũng thay đổi.

**1️. Ví dụ: Máy bán hàng tự động (Vending Machine)**

**Bài toán:**

Một máy bán hàng có các trạng thái:

* **Không có tiền (No Coin)**
* **Có tiền (Has Coin)**
* **Đang bán hàng (Dispensing)**
* **Hết hàng (Out of Stock)**

Mỗi trạng thái sẽ có các hành vi khác nhau khi người dùng thao tác.

2. **Ví dụ: Trạng thái tài khoản ngân hàng**

**Bài toán:**

Một tài khoản ngân hàng có thể có 3 trạng thái:

* **Tài khoản hoạt động (Active)**
* **Bị đóng băng (Frozen)**
* **Đóng tài khoản (Closed)**

Mỗi trạng thái có quy tắc riêng khi gửi tiền, rút tiền.

1. **Strategy Pattern - Mẫu thiết kế chiến lược**

Strategy Pattern là một mẫu thiết kế **hành vi (Behavioral Pattern)**, cho phép chọn thuật toán hoặc phương thức thực thi tại **thời điểm chạy (runtime)** mà không cần sửa đổi mã nguồn chính.

📌 **Khi nào nên sử dụng Strategy Pattern?**  
✔ Khi có nhiều thuật toán hoặc cách xử lý khác nhau cho cùng một tác vụ.  
✔ Khi muốn **giảm số lượng if-else hoặc switch-case** trong mã nguồn.  
✔ Khi muốn mở rộng hoặc thay đổi thuật toán dễ dàng mà **không ảnh hưởng đến phần còn lại của hệ thống**.

**1️. Ví dụ: Tính toán phí vận chuyển**

**Bài toán**

Một công ty vận chuyển có nhiều cách tính phí khác nhau:

* **Giao hàng tiêu chuẩn (Standard Shipping)**
* **Giao hàng nhanh (Express Shipping)**
* **Giao hàng siêu tốc (Same-Day Delivery)**

**2. Ví dụ: Thanh toán (Payment)**

**Bài toán**

**Một ứng dụng thương mại điện tử hỗ trợ nhiều phương thức thanh toán:**

* **Thanh toán bằng thẻ tín dụng**
* **Thanh toán bằng ví điện tử (PayPal, MoMo, ZaloPay)**
* **Thanh toán bằng tiền mặt khi nhận hàng (COD)**

**3. Ví dụ: Sắp xếp danh sách (Sorting)**

**Bài toán**

**Một ứng dụng hỗ trợ nhiều cách sắp xếp danh sách:**

* **Sắp xếp theo Bubble Sort**
* **Sắp xếp theo Quick Sort**
* **Sắp xếp theo Merge Sort**

1. **Decorator Pattern - Mẫu thiết kế trang trí**

**Decorator Pattern** là một mẫu thiết kế thuộc nhóm **cấu trúc (Structural Pattern)**, cho phép **thêm hành vi mới vào một đối tượng mà không làm thay đổi mã nguồn của nó**.

**📌 Khi nào nên sử dụng Decorator Pattern?**

✔ Khi muốn mở rộng tính năng cho một đối tượng mà **không cần sửa đổi mã nguồn gốc**.  
✔ Khi cần **thêm/xóa linh hoạt** các chức năng trong runtime.  
✔ Khi muốn **tránh kế thừa lớp (class inheritance)** quá sâu.

**1️. Ví dụ: Trang trí đồ uống (Beverage)**

**Bài toán**

Một quán cà phê bán nhiều loại đồ uống như:

* Cà phê đen
* Cà phê sữa
* Cà phê thêm topping (đường, sữa, trân châu, v.v.)

Nếu dùng kế thừa (extends), ta sẽ có nhiều lớp như BlackCoffeeWithMilk, BlackCoffeeWithSugar, BlackCoffeeWithMilkAndSugar, … **gây rắc rối khi mở rộng**.  
Thay vào đó, ta dùng **Decorator Pattern** để linh hoạt thêm các topping.

2. **Ví dụ: Thêm chức năng cho Notifier (Thông báo)**

**Bài toán**

Hệ thống gửi thông báo **có thể nâng cấp linh hoạt**:

* Gửi **qua email**
* Gửi **qua SMS**
* Gửi **qua Facebook Messenger**

Người dùng có thể nhận thông báo **qua nhiều kênh cùng lúc**.

3. **Ví dụ: Thêm chức năng bảo mật cho file**

**Bài toán**

Ta có một hệ thống **lưu file**, cần thêm các tính năng như:

* **Mã hóa file**
* **Nén file**
* **Ghi log khi lưu file**

1. **Observer Pattern - Mẫu thiết kế quan sát**

**Observer Pattern** là một mẫu thiết kế thuộc nhóm **hành vi (Behavioral Pattern)**, cho phép **một đối tượng (Subject) thông báo đến nhiều đối tượng khác (Observers) khi có sự thay đổi trạng thái**.

**📌 Khi nào nên sử dụng Observer Pattern?**

✔ Khi **một đối tượng thay đổi trạng thái**, các đối tượng liên quan cần được thông báo **tự động**.  
✔ Khi **có nhiều đối tượng quan sát**, nhưng không muốn chúng bị ràng buộc chặt chẽ vào đối tượng chủ.  
✔ Khi cần **giảm sự phụ thuộc giữa các thành phần trong hệ thống** (**loose coupling**).

**1️. Ví dụ: Hệ thống thông báo thời tiết**

**Bài toán**

Hệ thống dự báo thời tiết cần cập nhật thông tin **nhiệt độ, độ ẩm, áp suất** và thông báo đến nhiều **bảng hiển thị** khác nhau như:

* **Màn hình hiển thị hiện tại**
* **Màn hình thống kê**
* **Màn hình dự báo**

Khi nhiệt độ thay đổi, tất cả màn hình phải cập nhật **tự động**.

2. **Ví dụ: Hệ thống thông báo khi có đơn hàng mới**

**Bài toán**

Trong hệ thống bán hàng online, khi có đơn hàng mới, hệ thống cần thông báo đến:

* **Bộ phận kho hàng**
* **Bộ phận giao hàng**
* **Khách hàng**

Mỗi đối tượng có thể tự động nhận thông báo khi đơn hàng được tạo.

1. **Abstract Factory Pattern - Mẫu thiết kế nhà máy trừu tượng**

**Abstract Factory** là một mẫu thiết kế thuộc nhóm **Creational Pattern (mẫu khởi tạo)**, giúp **tạo ra một nhóm đối tượng liên quan mà không cần chỉ rõ lớp cụ thể của chúng**.

**📌 Khi nào nên sử dụng Abstract Factory?**

✔ Khi hệ thống cần tạo ra **một nhóm đối tượng liên quan với nhau**.  
✔ Khi muốn **giữ tính đóng mở (Open/Closed Principle)** để **dễ dàng mở rộng mà không sửa đổi code hiện có**.  
✔ Khi muốn **giảm sự phụ thuộc giữa các lớp**, giúp code linh hoạt hơn.

**1️. Ví dụ: Hệ thống giao diện UI đa nền tảng (Windows vs MacOS)**

**Bài toán**

Ứng dụng cần hỗ trợ **giao diện cho cả Windows và MacOS**, mỗi hệ điều hành có **các thành phần UI riêng biệt**:

* **Windows**: Button & Checkbox có phong cách Windows.
* **MacOS**: Button & Checkbox có phong cách MacOS.

Ứng dụng cần **tạo các thành phần UI phù hợp** mà không cần kiểm tra hệ điều hành trực tiếp.

2. **Ví dụ: Hệ thống quản lý cơ sở dữ liệu (SQL vs NoSQL)**

**Bài toán**

Ứng dụng cần hỗ trợ **nhiều loại cơ sở dữ liệu** (SQL & NoSQL), mỗi loại có cách thao tác khác nhau:

* **SQL Database** (MySQL, PostgreSQL) sử dụng **câu lệnh SQL**.
* **NoSQL Database** (MongoDB) sử dụng **document-based API**.

Ứng dụng cần **tạo kết nối phù hợp** mà không cần kiểm tra loại database trực tiếp.