VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Kỹ Thuật Lập Trình (Cơ bản và nâng cao C++)

OOP Nâng Cao K24

BT2 Third-party API

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 5/2025

Mục lục

1 Đề Bài: Thiết Kế Hệ Thống Gửi Request Đến API Bên Thứ Ba

Vấn Đề

Trong các hệ thống hiện đại, việc tích hợp với các dịch vụ bên ngoài (third-party APIs) như OpenAI, DeepSeek, hoặc Google là điều phổ biến. Tuy nhiên, mỗi dịch vụ thường cung cấp giao diện khác nhau và không thống nhất, dẫn đến khó khăn trong việc mở rộng, bảo trì và kiểm thử.

Hơn nữa, việc ghi lại nhật ký (log) của các hành động gọi request và phản ứng lại khi request được gửi (ví dụ: hiển thị giao diện, gửi mail, v.v.) là yêu cầu quan trọng đối với các hệ thống có tính tương tác cao.

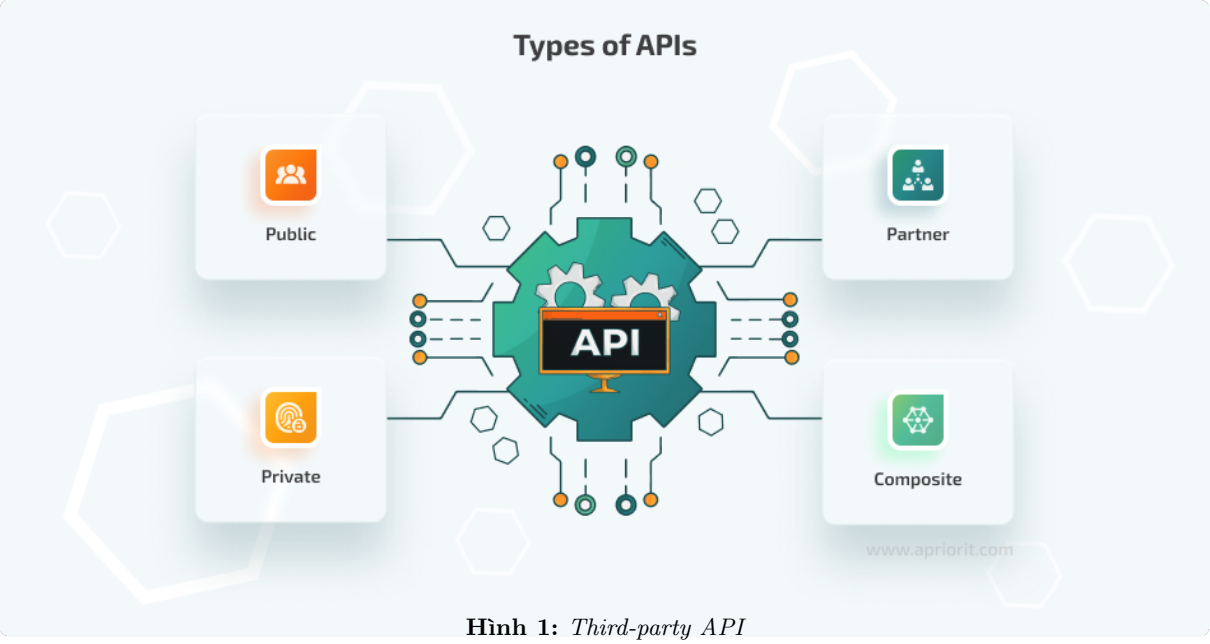
Vấn đề đặt ra:

- Làm sao để chuẩn hoá việc gọi các API có giao diện khác nhau?
- Làm sao để theo dõi mọi request được gửi, phục vụ cho log, thống kê và kiểm thử?
- Làm sao để thiết kế mở rộng, dễ thêm API mới mà không ảnh hưởng hệ thống cũ?

Mục Tiêu

Xây dựng một hệ thống C++ cho phép:

- Gửi request đến các API bên thứ ba (ví dụ: **ChatGPT** và **DeepSeek**).
- Chuẩn hoá giao diện gọi các API bằng **Adapter Pattern**.
- Ghi log tất cả các hành động gửi request bằng **Singleton Logger**.
- Cho phép theo dõi và phản ứng khi request được gửi bằng **Observer Pattern**.
- Viết các test case để kiểm thử các tình huống như attach/detach observer, gọi nhiều model khác nhau,...



Hình 1: *Third-party API*

2 Kiến thức yêu cầu

2.1 Adapter Pattern

Định nghĩa: Adapter Pattern là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern), cho phép các lớp có giao diện không tương thích có thể làm việc cùng nhau bằng cách chuyển đổi giao diện của một lớp thành giao diện mà client mong đợi.

Ví dụ thực tế:

- Bạn có một thiết bị điện với phích cắm 3 chấu (chuẩn Mỹ), nhưng ổ điện nhà bạn là 2 chấu (chuẩn Việt Nam). Bạn cần một bộ chuyển đổi (adapter) để dùng được thiết bị đó.

Trong lập trình:

- Hệ thống cần gọi phương thức `run()`, nhưng thư viện cũ chỉ cung cấp `execute()`. Bạn viết một lớp Adapter để ánh xạ `run()` → `execute()`.

Tóm tắt:

- Adapter giúp **kết nối** hai thành phần không tương thích.
- Rất hữu ích khi **tái sử dụng mã cũ** hoặc **kết hợp thư viện bên ngoài**.
- Tuân thủ nguyên lý **Open/Closed** – mở rộng mà không sửa đổi mã nguồn gốc.

Đọc thêm: https://refactoring.guru/design-patterns/adapter

2.2 Singleton Pattern

Định nghĩa: Singleton Pattern đảm bảo rằng một lớp chỉ có **một thể hiện duy nhất** (instance), và cung cấp một điểm truy cập toàn cục đến thể hiện đó.

Ví dụ thực tế:

- Hệ thống chỉ có một `Database Connection`, hoặc một `Logger`, nên cần đảm bảo chỉ tạo một instance trong toàn hệ thống.

Đặc điểm:

- Constructor là `private`.
- Dùng phương thức `getInstance()` để lấy thể hiện duy nhất.

Tóm tắt:

- Kiểm soát số lượng instance (chỉ một).
- Dễ quản lý tài nguyên toàn cục.
- Có thể gây khó khăn cho việc kiểm thử nếu lạm dụng.

Đọc thêm: https://refactoring.guru/design-patterns/singleton

2.3 Observer Pattern

Định nghĩa: Observer Pattern thiết lập mối quan hệ **một-nhiều** giữa các đối tượng: khi một đối tượng (Subject) thay đổi trạng thái, tất cả các đối tượng phụ thuộc (Observers) sẽ được thông báo và cập nhật tự động.

Ví dụ thực tế:

- Ứng dụng thời tiết: nhiều màn hình (observers) cùng theo dõi một nguồn dữ liệu thời tiết (subject). Khi dữ liệu thay đổi, tất cả giao diện được cập nhật.

Thành phần chính:

- `Subject`: quản lý danh sách observers và thông báo thay đổi.

- `Observer`: giao diện chung cho các đối tượng theo dõi.

Tóm tắt:

- Giảm sự phụ thuộc chặt giữa các thành phần.
- Dễ mở rộng thêm nhiều đối tượng theo dõi.
- Phù hợp cho các hệ thống sự kiện, GUI, MVC.

Đọc thêm: https://refactoring.guru/design-patterns/observer

3 Hướng giải quyết

Để giải quyết bài toán tích hợp nhiều API bên thứ ba khác nhau với yêu cầu mở rộng, ghi log, và theo dõi hành vi, ta triển khai hệ thống theo các bước sau:

3.1 Bước 1: Chuẩn hoá giao diện gọi API bằng Adapter

- Định nghĩa một interface chuẩn `ThirdPartyAPI` với hàm `sendRequest(std::string)`.
- Tạo các lớp adapter như `ChatGPTAdapter` và `DeepSeekAdapter`, kế thừa `ThirdPartyAPI`.
- Trong mỗi adapter, chuyển đổi lời gọi `sendRequest(msg)` thành lời gọi thực tế của SDK gốc (`callGPT()`, `askDeepSeek()`,...).
- Nhờ đó, các thành phần phía trên (ví dụ `RequestManager`) có thể sử dụng nhiều API khác nhau một cách thống nhất.

3.2 Bước 2: Gửi request qua lớp điều phối trung gian RequestManager

- Xây dựng lớp `RequestManager` làm nhiệm vụ gửi request đến adapter.
- Trước khi gửi, gọi `Logger::instance().log()` để ghi log hành động.
- Sau khi gửi xong, thông báo cho tất cả các observer đã đăng ký bằng cách gọi `observer->onRequestSent()`.

3.3 Bước 3: Ghi log toàn hệ thống bằng Singleton Logger

- Cài đặt lớp `Logger` sử dụng mẫu thiết kế Singleton để đảm bảo toàn hệ thống dùng chung một đối tượng log.
- Cho phép gọi `Logger::instance().log(msg)` từ bất kỳ đâu mà không cần truyền tham chiếu.
- Việc ghi log có thể mở rộng sang ghi file, gửi cloud, hoặc stdout tùy mục tiêu.

3.4 Bước 4: Theo dõi request bằng Observer Pattern

- Định nghĩa interface `IRequestObserver` với hàm `onRequestSent(std::string)`.
- Cho phép các đối tượng (observer) đăng ký bằng `attach()` hoặc gỡ bằng `detach()` trong `RequestManager`.
- Khi có request được gửi, vòng lặp gọi đến tất cả observer để họ phản ứng (in ra, lưu lại, gửi alert...).

3.5 Bước 5: Kiểm thử chức năng qua các test độc lập

- Viết các hàm test để mô phỏng hành vi hệ thống khi:
  - Gửi request đến từng API.
  - Gửi với các model khác nhau.
  - Gọi nhiều lần `attach()`.
  - Gọi `detach()` và xác nhận observer không còn hoạt động.
- Dùng `std::stringstream` để redirect `std::cout` và kiểm tra output thực tế với kết quả mong đợi.

4 Hiện thực

1. Cơ bản phần này chụp ảnh nộp vào kênh **Nộp terminal** để có role mới qua task sau
   (a) Hiện thực phần trên và chụp ảnh chứng tỏ đã hoành thành
2. Phần mở rộng tăng khả năng tư duy và nộp vào code vào kênh **Nộp các file code**
   (a) **Tích hợp thêm API từ Momo hoặc các cổng thanh toán khác:**

   Với kiến trúc sử dụng Adapter Pattern, việc thêm một API mới như `Momo_API` rất đơn giản. Chỉ cần:

   - Tạo lớp adapter mới: `MomoAdapter`, kế thừa từ `ThirdPartyAPI`.
   - Cài đặt lại hàm `sendRequest()` để chuyển đổi sang lời gọi của SDK Momo.

   Nhờ đó, hệ thống có thể gửi request đến Momo để kiểm tra số dư, tạo đơn hàng, hoặc xác thực giao dịch mà không làm thay đổi các phần còn lại.

   (b) **Sử dụng Observer để gắn thêm thông tin chi phí hoặc trạng thái tài chính cho mỗi request:**

   Trong các ứng dụng thực tế, việc gửi request ra bên ngoài thường liên quan đến chi phí (token, tiền). Có thể dùng một observer như `CostObserver` để:

   - Theo dõi và tính toán chi phí sau mỗi request.
   - Ghi log số tiền đã tiêu thụ.
   - Cảnh báo nếu vượt quá hạn mức.

   Việc này giúp minh bạch hoá hành vi tiêu tốn tài nguyên và hỗ trợ việc giám sát hệ thống.

   (c) **Gắn timestamp hoặc ID truy vết (trace ID) vào mỗi request** để dễ dàng theo dõi log và debug.