Báo cáo: PHÁT TRIỂN KỸ NĂNG VIẾT PROMPT HIỆU QUẢ ĐỂ
TẬN DỤNG TỐI ĐA KHẢ NĂNG CỦA CÁC MÔ HÌNH NGÔN NGỮ LỚN TRONG HỌC TẬP

Họ và tên : Dương Văn Huy. Mã SV: 25020175
Lớp học phần: VNU_E252014
Trường: Đại học Công Nghệ - Đại Học Quốc Gia Hà Nội. Ngành: CN1

Nội dung báo cáo

Ba tác vụ lựa chọn:

1. Tóm tắt tài liệu: Head First Java
2. Giải thích khái niệm: Design Patterns trong Java
3. Bộ câu hỏi ôn tập: chương Toán học đồng dư - Toán rời rạc

I. Tác vụ 1

1. Prompt cơ bản

Tóm tắt cuốn Head First Java

2. Prompt cải tiến

Tóm tắt cuốn Head First Java dành cho người mới học lập trình.

Yêu cầu:

- Trình bày các nội dung chính theo từng chương hoặc chủ đề
- Giải thích ngắn gọn, dễ hiểu
- Độ dài khoảng 300–500 từ
- Dùng bullet points

3. Prompt nâng cao

Bạn là một giảng viên lập trình Java có 10 năm kinh nghiệm.
Hãy tóm tắt cuốn Head First Java theo cách giúp người mới học nắm nhanh kiến thức cốt lõi.
Yêu cầu:

Chia nội dung thành các phần:

- Cơ bản Java (syntax, OOP)
- Các khái niệm quan trọng (inheritance, polymorphism, encapsulation)
- API và thư viện

Mỗi phần:

- Giải thích ngắn gọn
- Đưa ví dụ minh họa đơn giản

Cuối cùng:

- Tóm tắt 5 ý quan trọng nhất cần nhớ

Hãy trình bày rõ ràng, dễ học, ưu tiên tư duy thay vì lý thuyết khô khan.

II. Tác vụ 2

1. Prompt cơ bản

> Giải thích Design Pattern trong Java

2. Prompt cải tiến

Giải thích khái niệm Design Pattern trong Java.
Yêu cầu:

- Định nghĩa rõ ràng
- Phân loại (Creational, Structural, Behavioral)
- Mỗi loại đưa 2–3 ví dụ phổ biến
- Giải thích ngắn gọn, dễ hiểu cho sinh viên CNTT

3. Prompt nâng cao

Bạn là một Software Architect.
Hãy giải thích Design Pattern trong Java cho lập trình viên junior.

Yêu cầu:

a. Giải thích:
- Design Pattern là gì và tại sao cần dùng

b. Phân loại:
- Creational
- Structural
- Behavioral

c. Với mỗi loại:
- Chọn 1 pattern tiêu biểu (ví dụ: Singleton, Factory, Observer)
- Mô tả:
  - Vấn đề nó giải quyết
  - Cách hoạt động (có thể dùng pseudo-code hoặc Java code ngắn)
  - Khi nào nên dùng / không nên dùng

d. Áp dụng:
- Đưa ví dụ thực tế trong dự án phần mềm

Hãy giải thích theo kiểu “dạy học”, có logic rõ ràng, tránh lý thuyết quá trừu tượng.

III. Tác vụ 3

1. Prompt cơ bản

Bộ đề cương ôn tập Toán đồng dư

2. Prompt cải tiến

Tạo bộ đề cương ôn tập về Toán đồng dư trong Toán rời rạc.
Yêu cầu:

- Bao gồm các chủ đề chính (định nghĩa, tính chất, định lý)
- Có ví dụ minh họa
- Có bài tập từ dễ đến trung bình
- Trình bày dạng outline rõ ràng

3. Prompt nâng cao

Bạn là giảng viên dạy Toán rời rạc.
Hãy xây dựng một bộ đề cương ôn tập chuyên sâu về **Toán đồng dư (modular arithmetic)** cho sinh viên đại học.

Yêu cầu:

+ Lý thuyết:
  - Định nghĩa đồng dư
  - Các tính chất quan trọng
  - Định lý Fermat nhỏ, Euler
+ Ví dụ:
  - Minh họa từng khái niệm bằng bài toán cụ thể
+ Bài tập:
  - 5 bài cơ bản
  - 5 bài trung bình
  - 3 bài nâng cao
  - Có đáp án hoặc hướng dẫn giải
+ Ứng dụng:
  - Ứng dụng trong mã hóa (cryptography)

Hãy trình bày như một tài liệu học tập hoàn chỉnh, rõ ràng, logic và dễ ôn thi.

? Phân tích hiệu quả prompt: Tại sao một số prompt lại hiệu quả hơn prompt khác ?

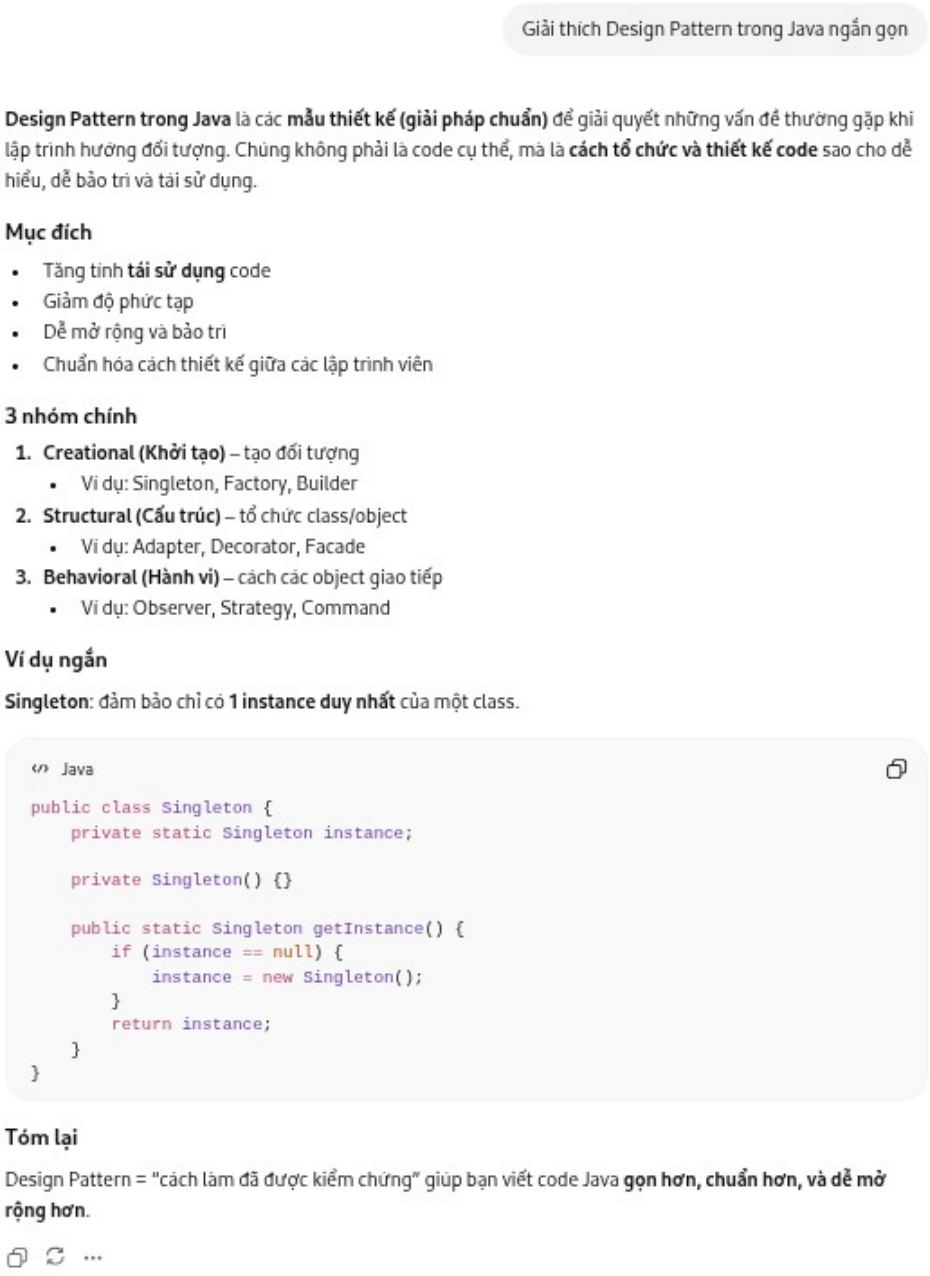
1. Bản chất Prompt

Prompt đóng vai trò quan trọng, tạo nên ảnh hưởng lớn đến chất lượng đầu ra của câu trả lời. Prompt định hình không gian suy luận mà AI được phép hoạt động. Vì sao?

Nếu Prompt chung chung thì sẽ có phạm vi mơ hồ, không gian suy luận rộng, output của AI sẽ trung bình, an toàn, chung chung

Nếu Prompt cấu trúc tốt, rõ ràng, không gian suy luận sẽ được ép hình, oupt sâu hơn, chính xác hơn

Dưới đây là ví dụ cho 3 kiểu prompt với tác vụ cụ thể: giải thích khái niệm design pattern trong java



Ảnh 1. Prompt cơ bản

Giải thích khái niệm Design Pattern trong Java.
Yêu cầu:

Định nghĩa rõ ràng
Phân loại (Creational, Structural, Behavioral)
Mỗi loại đưa 2–3 ví dụ phổ biến
Giải thích ngắn gọn, dễ hiểu cho sinh viên CNTT

Design Pattern trong Java là các "mẫu thiết kế" (giải pháp chuẩn) dùng để giải quyết những vấn đề thường gặp trong lập trình hướng đối tượng. Chúng không phải là code cụ thể, mà là **cách tổ chức code** sao cho dễ bảo trì, mở rộng và tái sử dụng.

---

1. Định nghĩa

Design Pattern là:

- Một **giải pháp tổng quát, đã được kiểm chứng** cho các vấn đề thiết kế phần mềm lặp lại.
- Giúp lập trình viên **viết code sạch hơn, dễ hiểu hơn**, và giảm lỗi.

---

2. Phân loại Design Pattern

🔹 2.1. Creational Patterns (Mẫu khởi tạo)

👉 Tập trung vào **cách tạo đối tượng** linh hoạt và hiệu quả.

Ví dụ phổ biến:

- **Singleton**
  - Đảm bảo chỉ có **1 instance duy nhất** của một class.
  - Dùng cho: logging, config, database connection.
  - Ý tưởng: constructor private + static instance.
- **Factory Method**
  - Tạo đối tượng mà không cần chỉ rõ class cụ thể.
  - Dùng khi: có nhiều loại object cùng chung interface.
  - Ví dụ: tạo `Shape` (Circle, Rectangle…) qua factory.
- **Builder**
  - Tạo object phức tạp từng bước.
  - Dùng khi: object có nhiều tham số (tránh constructor dài).
  - Ví dụ: tạo `User` với nhiều thuộc tính optional.

Get Plus ×

🔹 2.2. Structural Patterns (Mẫu cấu trúc)

👉 Tập trung vào **cách tổ chức, kết hợp các class/object**.

Ví dụ phổ biến:

- **Adapter**
  - Chuyển đổi interface của class này thành interface khác.
  - Giống như "bộ chuyển đổi".
  - Ví dụ: chuyển dữ liệu từ API cũ sang API mới.
- **Decorator**
  - Thêm chức năng cho object mà không sửa code gốc.
  - Dùng khi: muốn mở rộng hành vi linh hoạt.
  - Ví dụ: thêm tính năng cho `Coffee` (milk, sugar…).
- **Facade**
  - Cung cấp interface đơn giản cho hệ thống phức tạp.
  - Giúp người dùng chỉ gọi 1 class thay vì nhiều class.
  - Ví dụ: hệ thống thanh toán.

🔹 2.3. Behavioral Patterns (Mẫu hành vi)

👉 Tập trung vào **cách các object giao tiếp với nhau**.

Ví dụ phổ biến:

- **Observer**
  - Khi 1 object thay đổi → các object khác được cập nhật.
  - Dùng trong: event system, UI.
  - Ví dụ: subscribe / notify.
- **Strategy**
  - Cho phép thay đổi thuật toán linh hoạt khi runtime.
  - Dùng khi: có nhiều cách xử lý cùng một vấn đề.
  - Ví dụ: nhiều cách sắp xếp, thanh toán.
- **Command**
  - Đóng gói request thành object.
  - Dùng khi: cần undo/redo hoặc queue task.
  - Ví dụ: remote control (bật/tắt thiết bị).

---

3. Tóm tắt dễ nhớ

Nhóm	Mục đích	Ví dụ
Creational	Tạo object	Singleton, Factory, Builder
Structural	Tổ chức code	Adapter, Decorator, Facade
Behavioral	Giao tiếp	Observer, Strategy, Command

---

4. Kết luận (cho sinh viên)

- Design Pattern = **kinh nghiệm thiết kế đã chuẩn hóa**
- Không bắt buộc phải dùng, nhưng:
  - Giúp code **chuyên nghiệp hơn**
  - Dễ làm việc nhóm
  - Dễ mở rộng hệ thống lớn

Nếu cần, mình có thể:

- Viết **code Java minh họa từng pattern**
- Hoặc giúp bạn **phân biệt các pattern dễ nhầm (Factory vs Builder, Strategy vs State, …)**

Ảnh 2, 3, 4. Prompt cải tiến

Tôi sẽ giải thích theo cách "đi từ thực tế → khái niệm → áp dụng", đúng mindset của một Software Architect.

1. Design Pattern là gì? Tại sao cần?

✔️ Khái niệm

Design Pattern là **mẫu thiết kế đã được kiểm chứng** để giải quyết các vấn đề lặp lại trong lập trình.

> Không phải là code cụ thể → mà là **cách tổ chức code**.

✔️ Tại sao cần?

Khi bạn code nhiều dự án, bạn sẽ gặp lại các vấn đề giống nhau:

- Tạo object sao cho hợp lý?
- Các class giao tiếp với nhau thế nào?
- Làm sao code dễ mở rộng mà không sửa nhiều?

👉 Nếu không dùng pattern:

- Code dễ **rối, khó bảo trì**
- Khó mở rộng (vi phạm OOP principles)

👉 Nếu dùng pattern:

- Code **chuẩn hóa**
- Dễ đọc (team hiểu ngay)
- Dễ mở rộng, tái sử dụng

2. Phân loại Design Pattern

Có 3 nhóm chính:

Nhóm	Mục tiêu
Creational	Tạo object
Structural	Tổ chức class/object
Behavioral	Giao tiếp giữa object

3. Creational Pattern – Singleton

✔️ Vấn đề

Bạn cần:

- Chỉ có **1 instance duy nhất**
- Dùng chung toàn hệ thống

Ví dụ:

- Config app
- Logger
- Database connection

✔️ Cách hoạt động

```java
public class AppConfig {
    private static volatile AppConfig instance;

    private AppConfig() {}

    public static AppConfig getInstance() {
        if (instance == null) {
            synchronized (AppConfig.class) {
                if (instance == null) {
                    instance = new AppConfig();
                }
            }
        }
        return instance;
    }
}
```

✔️ Khi nào nên dùng

- Tài nguyên dùng chung (config, cache)
- Tránh tạo nhiều instance gây lỗi

❌ Khi không nên dùng

- Khi cần test (Singleton khó mock)
- Khi không cần global state

✔️ Ví dụ thực tế

Trong backend:

- `DatabaseConnection.getInstance()`
- `Logger.getInstance()`

4. Structural Pattern – Adapter

✔️ Vấn đề

Bạn có:

- 2 class **không tương thích interface**
- Nhưng vẫn muốn chúng làm việc với nhau

✔️ Cách hoạt động

```java
}

// Class cũ (không tương thích)
class OldPaymentSystem {
    void makePayment() {
        System.out.println("Paid using old system");
    }
}

// Adapter
class PaymentAdapter implements Payment {
    private OldPaymentSystem oldSystem;

    public PaymentAdapter(OldPaymentSystem oldSystem) {
        this.oldSystem = oldSystem;
    }

    public void pay() {
        oldSystem.makePayment();
    }
}
```

✔️ Khi nào nên dùng

- Tích hợp API cũ
- Dùng thư viện bên thứ 3

❌ Khi không nên dùng

- Nếu có thể sửa trực tiếp class gốc
- Nếu hệ thống đơn giản

5. Behavioral Pattern – Observer

✔️ Vấn đề

Bạn cần:

- Khi 1 object thay đổi → nhiều object khác tự cập nhật

✔️ Cách hoạt động

```java
import java.util.*;

interface Observer {
    void update(String message);
}

class User implements Observer {
    public void update(String message) {
        System.out.println("Received: " + message);
    }
}

class Channel {
    private List<Observer> subscribers = new ArrayList<>();

    void subscribe(Observer o) {
        subscribers.add(o);
    }

    void notifyAllUsers(String msg) {
        for (Observer o : subscribers) {
            o.update(msg);
```

✔️ Khi nào nên dùng

- Event system
- Notification
- UI update

❌ Khi không nên dùng

- Nếu số lượng observer quá lớn → khó debug
- Quan hệ quá phức tạp

✔️ Ví dụ thực tế

- YouTube:
  - User subscribe channel
  - Channel đăng video → notify user
- Backend:
  - Order → gửi email + log + analytics

6. Tóm tắt tư duy (quan trọng nhất)

Pattern	Ý tưởng cốt lõi
Singleton	1 instance duy nhất
Adapter	Biến interface cho tương thích
Observer	Khi A đổi → B, C, D cập nhật

7. Cách học đúng (rất quan trọng cho bạn)

Đừng học kiểu thuộc lòng.

Hãy học theo flow:

1. Gặp vấn đề thực tế
2. Tự code cách naive
3. Nhận ra vấn đề
4. Áp dụng pattern

👉 Pattern = kinh nghiệm đóng gói

Nếu bạn muốn, tôi có thể:

- Cho bạn **bài tập thực chiến từng pattern**
- Hoặc áp dụng trực tiếp vào **project backend bạn đang làm (auction system)**

Ảnh 5,6,7,8. Prompt nâng cao

Prompt không chỉ là đầu vào, mà là cơ chế điều hướng cách mô hình suy luận và lựa chọn thông tin. Nó hoạt động như một dạng "điều kiện hóa xác suất", làm thu hẹp hoặc mở rộng không gian câu trả lời. Prompt càng cụ thể thì phân phối đầu ra càng tập trung, nhưng nếu quá chi tiết có thể gây nhiễu hoặc làm mô hình bám hình thức hơn nội dung.

Cấu trúc prompt ảnh hưởng trực tiếp đến cách lập luận. Prompt tuyến tính (một câu hỏi chung) khiến mô hình tự chọn chiến lược, thường dẫn đến câu trả lời an toàn, ít chiều sâu. Prompt phân tầng (chia bước rõ ràng) buộc mô hình triển khai suy luận có tổ chức, từ đó tăng độ mạch lạc và tính phân tích. Vì vậy, cấu trúc không chỉ là hình thức mà là công cụ kiểm soát tiến trình tư duy của AI.

Ràng buộc trong prompt (định dạng, độ dài, phong cách) giúp tăng độ khớp với kỳ vọng, nhưng tạo ra trade-off. Quá ít ràng buộc dẫn đến mơ hồ; quá nhiều ràng buộc làm giảm tính linh hoạt và có thể khiến câu trả lời trở nên cơ học. Prompt hiệu quả nằm ở điểm cân bằng giữa định hướng và tự do.

Prompt cũng mang "ngữ cảnh ẩn" như giọng điệu, đối tượng, mục đích. Mô hình rất nhạy với các tín hiệu này và tự điều chỉnh độ phức tạp, cách diễn đạt. Điều này cho thấy ngay cả những yếu tố không được nói rõ vẫn có ảnh hưởng đáng kể đến chất lượng đầu ra.

Một điểm quan trọng là prompt tạo ra hiệu ứng định khung. Cách đặt câu hỏi có thể dẫn dắt câu trả lời theo hướng tích cực hoặc tiêu cực. Vì vậy, prompt không trung lập; nó định hình góc nhìn mà mô hình sử dụng.

Cuối cùng, có mối quan hệ phi tuyến giữa độ dài prompt và hiệu quả. Sau một ngưỡng nhất định, thêm thông tin không cải thiện mà còn làm giảm chất lượng do nhiễu. Do đó, prompt tốt không phải là dài nhất, mà là rõ ràng, có cấu trúc và đủ định hướng để kiểm soát suy luận.

? Tổng hợp

Báo cáo này tổng hợp các nguyên tắc viết prompt hiệu quả, dựa trên phân tích mối quan hệ giữa cấu trúc prompt và chất lượng câu trả lời. Trọng tâm không chỉ là "viết gì", mà là "thiết kế cách AI sẽ suy nghĩ".

Báo cáo rút gọn các nguyên tắc viết prompt hiệu quả, tập trung vào chiều sâu và tính ứng dụng:

1. Xác định mục tiêu rõ ràng
   Prompt phải nói rõ làm gì, mức độ nào, kết quả ra sao.
   Ví dụ: thay vì "phân tích prompt", dùng "phân tích sâu ảnh hưởng của cấu trúc prompt và đưa nhận định phản biện".
   → Giảm mơ hồ, tăng độ chính xác.

2. Cấu trúc hóa suy luận
   Chia bước (định nghĩa → cơ chế → ví dụ → nhận định) giúp AI lập luận logic hơn.
   → Cấu trúc prompt quyết định cấu trúc tư duy.

3. Kiểm soát định dạng đầu ra
   Yêu cầu như “ngắn gọn, không in đậm” làm câu trả lời cô đọng và ít lan man hơn.
   → Định dạng ảnh hưởng trực tiếp đến cách tổ chức nội dung.

4. Ràng buộc có chọn lọc
   Ràng buộc vừa đủ giúp bám mục tiêu; quá nhiều làm câu trả lời cơ học.
   → Tối ưu là cân bằng giữa định hướng và linh hoạt.

5. Cung cấp ngữ cảnh
   Liên kết với yêu cầu trước (như bài này) giúp câu trả lời nhất quán và sâu hơn.
   → Prompt tốt là một phần của chuỗi, không phải câu lệnh rời.

6. Dùng ví dụ để “neo” cách hiểu
   Yêu cầu có ví dụ buộc AI cụ thể hóa thay vì nói chung.
   → Ví dụ đóng vai trò hướng dẫn ngầm.

7. Nhận diện hiệu ứng định khung
   Cách đặt câu hỏi dẫn dắt hướng trả lời (tiêu cực/tích cực).
   → Prompt không trung lập.

8. Tối ưu độ dài
   Ngắn quá thì thiếu, dài quá thì nhiễu.
   → Quan trọng là mật độ thông tin.

Kết luận: prompt hiệu quả là prompt thiết kế được cách AI suy nghĩ, không chỉ nội dung AI trả lời.