Question:

In a social media platform, a UserProfile class is responsible for managing user data, handling friend requests, generating activity feeds, and sending notifications. What problems might arise from this design, and how would you refactor it to adhere to the Single Responsibility Principle (SRP) and Separation of Concerns (SOC)?

Answer:

The UserProfile class is a God Object because it handles multiple unrelated responsibilities, violating the Single Responsibility Principle (SRP). This leads to a tightly coupled design that is hard to maintain, test, and extend. For example, if the logic for sending notifications changes, it could inadvertently affect the friend request handling or activity feed generation.

Refactoring:

Break the class into smaller, focused classes

UserDataManager: Manages user data (e.g., name, email, profile picture).

FriendRequestHandler: Handles friend request logic.

ActivityFeedGenerator: Generates activity feeds

NotificationService: Sends notifications.

This refactoring adheres to SRP and SOC, making the system more modular, easier to maintain, and less prone to unintended side effects.

Question 2

A food delivery app includes a complex "meal planning" feature that was built for a future update but is never used. This has added significant overhead to the project, slowing down development and increasing costs. Which software design principles can help prevent this issue, and how would you apply them to justify removing it?

Answer:

The **YAGNI (You Aren't Gonna Need It)** and **KISS (Keep It Simple, Stupid)** principles can help prevent this issue by ensuring that only necessary features are developed and that the system remains simple and maintainable.

- **YAGNI** emphasizes that developers should only implement features required at the moment, avoiding unnecessary complexity and resource consumption.
- **KISS** promotes simplicity in design, making the system easier to develop, test, and maintain.

Keeping an unused **"meal planning"** feature increases development and maintenance costs, as developers must manage, debug, and update unused code, diverting time and resources away from essential functionality. Additionally, unnecessary complexity can introduce hidden dependencies, slowing down future updates.

To address this, the best approach is to **remove the unused feature now** and reconsider implementing it **only if real user demand justifies it in the future**. This keeps the app lightweight, efficient, and focused on delivering value to users without unnecessary overhead.