1. Câu hỏi 1: Lớp nào dưới đây đại diện cho các đối tượng có dữ liệu và trạng thái cần quản lý trong hệ thống?
A. Lớp biên
B. Lớp điều khiển
C. Lớp thực thể
D. Lớp giao diện
2. Câu hỏi 2: Lớp biên trong hệ thống có vai trò gì?
A. Xử lý dữ liệu
B. Điều khiển luồng công việc
C. Giao tiếp với người dùng hoặc hệ thống bên ngoài
D. Lưu trữ dữ liệu
3. Câu hỏi 3: Quan hệ nào giữa các lớp thể hiện sự kế thừa?
A. Association
B. Aggregation
C. Composition
D. Inheritance
4. Câu hỏi 4: Sơ đồ lớp mô tả:
A. Quan hệ giữa các đối tượng trong một luồng xử lý
B. Các lớp và quan hệ giữa các lớp trong hệ thống
C. Giao diện người dùng của hệ thống
D. Thứ tự luồng xử lý giữa các đối tượng
5. Câu hỏi 5: Quan hệ Include giữa các use case được dùng khi:
A. Một use case mở rộng một use case khác
B. Một use case cần gọi một use case khác để hoàn thành chức năng
C. Một use case kế thừa một use case khác
D. Một use case được sử dụng bởi hệ thống bên ngoài
6. UML, sơ đồ nào giúp mô tả sự tương tác giữa các đối tượng theo thời gian?
A. Sơ đồ lớp
B. Sơ đồ tuần tự
C. Sơ đồ trạng thái
D. Sơ đồ hoạt động
7. CMMI mức 5

– Optimizing tập trung vào điều gì?

A. Kiểm soát quy trình
B. Định lượng quy trình
C. Cải tiến quy trình
D. Xác định quy trình
8. Đâu là ưu điểm chính của mô hình phát triển phần mềm Agile?

A. Linh hoạt và thay đổi nhanh chóng
B. Yêu cầu chi tiết ngay từ đầu
C. Kiểm thử chỉ diễn ra ở giai đoạn cuối
D. Phù hợp với tất cả các loại dự án
9. Nguyên tắc DRY (Don’t Repeat Yourself) trong lập trình có ý nghĩa gì?
A. Hạn chế viết code lặp lại
B. Viết code dễ đọc hơn
C. Tăng tốc độ thực thi của chương trình
D. Giảm chi phí phát triển phần mềm
10. Yếu tố nào quan trọng nhất trong việc thiết kế một lớp trong lập trình hướng đối tượng?
A. Đặt tên biến dễ hiểu
B. Đảm bảo tính đóng gói và tái sử dụng
C. Viết phương thức càng nhiều càng tốt
D. Dùng số nguyên thay vì số thực
11. Phát biểu nào đúng về kiểm thử phần mềm?
A. Kiểm thử chỉ diễn ra sau khi phát triển xong sản phẩm
B. Kiểm thử giúp tìm ra tất cả lỗi trong phần mềm
C. Kiểm thử có thể thực hiện song song với phát triển phần mềm
D. Kiểm thử không quan trọng nếu phần mềm được viết tốt

Trả lời ngắn
1.Định nghĩa phần mềm và công nghệ phần mềm.
Phần mềm là tập hợp các chương trình, dữ liệu và thông tin máy tính được thiết kế để thực hiện một nhiệm vụ cụ thể. Công nghệ phần mềm là lĩnh vực nghiên cứu, phát triển và ứng dụng các phương pháp, công cụ, quy trình để thiết kế, xây dựng, bảo trì và quản lý phần mềm.

2.Mô tả ngắn gọn về các mô hình vòng đời phát triển phần mềm phổ biến.

Các mô hình vòng đời phát triển phần mềm phổ biến bao gồm:

Mô hình thác nước (Waterfall): Phát triển theo từng giai đoạn tuần tự, mỗi giai đoạn chỉ thực hiện khi giai đoạn trước hoàn thành.

Mô hình phát triển linh hoạt (Agile): Phát triển phần mềm theo các vòng lặp ngắn, cải tiến liên tục và phản hồi từ người dùng.

Mô hình phát triển theo kiểu bánh xe (Spiral): Kết hợp yếu tố của thác nước và Agile, tập trung vào việc đánh giá rủi ro trong mỗi vòng lặp.

Mô hình V-Model**:** Mỗi giai đoạn phát triển phần mềm đi đôi với một giai đoạn kiểm thử, chú trọng vào sự chính xác ngay từ đầu.

3.Liệt kê ba loại yêu cầu phần mềm chính và giải thích từng loại.

Chức năng (Functional): Tính năng cụ thể hệ thống phải có (ví dụ: đăng nhập, thanh toán).

Phi chức năng (Non-functional): Tiêu chí về hiệu suất, bảo mật, khả năng mở rộng.

4.Mô tả vai trò của sơ đồ lớp UML trong thiết kế phần mềm.
Sơ đồ lớp UML giúp mô hình hóa cấu trúc hệ thống bằng cách biểu diễn các lớp, thuộc tính, phương thức và mối quan hệ giữa chúng. Nó hỗ trợ thiết kế hướng đối tượng, cải thiện giao tiếp giữa các thành viên nhóm và làm tài liệu tham chiếu cho lập trình viên. Ngoài ra, sơ đồ lớp còn giúp dễ dàng bảo trì, mở rộng hệ thống và đảm bảo phần mềm được thiết kế hiệu quả.
5.Tại sao kiểm thử phần mềm quan trọng trong phát triển phần mềm?

Kiểm thử phần mềm quan trọng vì:

- **Đảm bảo chất lượng sản phẩm**: Giúp phát hiện lỗi trước khi sản phẩm đến tay người dùng, giảm thiểu rủi ro.
- **Tiết kiệm chi phí sửa lỗi**: Càng phát hiện lỗi sớm, chi phí sửa chữa càng thấp. Nếu lỗi xuất hiện sau khi phát hành, chi phí sẽ cao hơn nhiều.
- **Nâng cao trải nghiệm người dùng**: Phần mềm ít lỗi, hoạt động ổn định giúp cải thiện trải nghiệm và giữ chân khách hàng.
- **Đảm bảo an toàn và bảo mật**: Giúp phát hiện lỗ hổng bảo mật như SQL Injection, XSS, CSRF, giảm nguy cơ bị tấn công.
- **Tăng tính tin cậy của phần mềm**: Đảm bảo phần mềm hoạt động đúng theo yêu cầu, tránh sự cố không mong muốn.

=> Kiểm thử phần mềm giúp nâng cao chất lượng, giảm rủi ro, tối ưu chi phí và đảm bảo phần mềm hoạt động ổn định, an toàn.

6.Định nghĩa nguyên lý SOLID trong lập trình hướng đối tượng.

SOLID là 5 nguyên tắc giúp thiết kế mã nguồn dễ bảo trì, mở rộng và tránh lỗi trong lập trình hướng đối tượng (OOP):

- **Single Responsibility Principle (SRP) - Nguyên tắc trách nhiệm duy nhất**: Mỗi class chỉ nên có một lý do để thay đổi, tức là chỉ chịu trách nhiệm cho một chức năng duy nhất. Ví dụ, không nên viết một class vừa quản lý user vừa gửi

email, mà nên tách thành `UserManager` và `EmailService`.

- **Open/Closed Principle (OCP) - Nguyên tắc đóng/mở**: Class nên có thể mở rộng nhưng không được sửa đổi trực tiếp. Điều này giúp hệ thống linh hoạt hơn, có thể thêm chức năng mới mà không làm ảnh hưởng đến mã nguồn cũ.

- **Liskov Substitution Principle (LSP) - Nguyên tắc thay thế Liskov**: Các class con phải có thể thay thế class cha mà không làm thay đổi hành vi của chương trình. Nếu một class con phá vỡ chức năng của class cha, tức là vi phạm nguyên tắc này.

- **Interface Segregation Principle (ISP) - Nguyên tắc phân tách interface**: Một interface không nên ép buộc class triển khai các phương thức không cần thiết. Thay vì một interface lớn chứa nhiều chức năng, nên tách thành nhiều interface nhỏ hơn để class chỉ cần triển khai những gì thực sự cần thiết.

- **Dependency Inversion Principle (DIP) - Nguyên tắc đảo ngược sự phụ thuộc**: Module cấp cao không nên phụ thuộc trực tiếp vào module cấp thấp. Thay vào đó, cả hai nên phụ thuộc vào abstraction (interface hoặc abstract class) để tăng tính linh hoạt.

=> Những nguyên tắc này giúp viết mã dễ bảo trì, dễ mở rộng và hạn chế lỗi phát sinh khi thay đổi hoặc mở rộng phần mềm.

7.Mô tả sự khác biệt giữa kiểm thử hộp đen và kiểm thử hộp trắng.
Kiểm thử hộp đen (Black Box Testing):

- Kiểm tra phần mềm dựa trên đầu vào và đầu ra mà không cần biết mã nguồn bên trong.
- Người kiểm thử chỉ quan tâm đến chức năng và phản hồi của hệ thống.
- Các kỹ thuật phổ biến: kiểm thử giá trị biên, kiểm thử bảng quyết định, kiểm thử dựa trên đặc tả yêu cầu.
- Ví dụ: Kiểm tra form đăng nhập bằng cách nhập tài khoản đúng/sai và quan sát kết quả.

Kiểm thử hộp trắng (White Box Testing):

- Kiểm tra logic bên trong của mã nguồn, cấu trúc chương trình.
- Người kiểm thử cần hiểu về mã nguồn và thuật toán của hệ thống.
- Các kỹ thuật phổ biến: kiểm thử dòng lệnh, kiểm thử luồng dữ liệu, kiểm thử điều kiện.

- Ví dụ: Kiểm tra một hàm xử lý đăng nhập bằng cách kiểm tra tất cả các nhánh điều kiện trong mã.

8.Mô tả quy trình thiết kế cơ sở dữ liệu từ sơ đồ lớp UML.

Xác định các lớp liên quan đến dữ liệu

- Chọn các lớp có đặc điểm lưu trữ dữ liệu (không bao gồm lớp điều khiển, giao diện).
- Ví dụ: Lớp Bệnh nhân, Hồ sơ khám, Bác sĩ trong hệ thống HIS.

Xác định thuộc tính và kiểu dữ liệu

- Chuyển thuộc tính lớp thành các cột trong bảng.
- Xác định kiểu dữ liệu phù hợp với hệ quản trị CSDL (MySQL, PostgreSQL,...).

Xác định khóa chính (Primary Key - PK)

- Chọn thuộc tính duy nhất để làm khóa chính (ID, mã số, ...).
- Nếu cần, dùng khóa chính tổng hợp từ nhiều thuộc tính.

Xác định quan hệ giữa các bảng

- Dựa vào liên kết trong sơ đồ lớp UML (association, aggregation, composition).
- Xác định khóa ngoại (Foreign Key - FK) để liên kết bảng.

Chuẩn hóa dữ liệu

- Kiểm tra các dạng chuẩn (1NF, 2NF, 3NF) để loại bỏ dư thừa dữ liệu.

Tạo sơ đồ ERD (Entity-Relationship Diagram)

- Chuyển đổi từ UML sang ERD để dễ dàng triển khai CSDL.

Viết lệnh SQL tạo bảng

- Chuyển đổi mô hình dữ liệu thành script SQL (CREATE TABLE, ALTER TABLE).

Kiểm thử và tối ưu hóa

- Chạy thử với dữ liệu mẫu, kiểm tra hiệu suất truy vấn, thêm chỉ mục nếu cần.

9.Nêu ba ưu điểm của việc sử dụng mô hình phát triển phần mềm
Agile.

Tính linh hoạt cao: Agile cho phép điều chỉnh yêu cầu và ưu tiên trong suốt quá trình phát triển, giúp đáp ứng nhanh chóng với sự thay đổi từ khách hàng hoặc thị trường.
Tăng cường sự hợp tác: Nhờ các vòng lặp ngắn (sprint) và giao tiếp thường xuyên giữa đội phát triển và khách hàng, Agile đảm bảo sản phẩm cuối cùng sát với mong đợi của người dùng.
Giảm rủi ro: Việc phát triển theo từng giai đoạn nhỏ và kiểm tra liên tục giúp phát hiện lỗi sớm, từ đó giảm thiểu rủi ro và chi phí sửa chữa ở giai đoạn sau.

10.Liệt kê các giai đoạn chính trong quá trình chuẩn hóa cơ sở dữ liệu.
Dạng chuẩn thứ nhất (1NF - First Normal Form): Loại bỏ các giá trị lặp lại trong bảng, đảm bảo mỗi ô chỉ chứa một giá trị duy nhất và xác định khóa chính.
Dạng chuẩn thứ hai (2NF - Second Normal Form): Đáp ứng 1NF và loại bỏ sự phụ thuộc hàm không đầy đủ, tức là mọi thuộc tính không khóa phải phụ thuộc hoàn toàn vào toàn bộ khóa chính.
Dạng chuẩn thứ ba (3NF - Third Normal Form): Đáp ứng 2NF và loại bỏ sự phụ thuộc bắc cầu, nghĩa là các thuộc tính không khóa chỉ phụ thuộc trực tiếp vào khóa chính, không qua thuộc tính trung gian khác.

Thảo luận nhóm
1.So sánh mô hình phát triển phần mềm Agile và mô hình Waterfall.

Dưới đây là sự so sánh giữa mô hình Agile và mô hình Waterfall:

Phương pháp phát triển:

- Waterfall: Phát triển theo từng giai đoạn tuần tự, mỗi giai đoạn chỉ bắt đầu khi giai đoạn trước hoàn thành.
- Agile: Phát triển linh hoạt, chia thành các vòng lặp (sprint) ngắn, mỗi vòng mang lại một phần sản phẩm có thể sử dụng được.

Linh hoạt:

- Waterfall: Ít linh hoạt, khó thay đổi yêu cầu khi đã bắt đầu một giai đoạn mới.
- Agile: Linh hoạt cao, dễ dàng thay đổi yêu cầu trong suốt quá trình phát triển dựa trên phản hồi từ người dùng.

Quản lý dự án:

- Waterfall: Dễ dàng quản lý vì các giai đoạn rõ ràng và được xác định trước.
- Agile: Quản lý phức tạp hơn, yêu cầu giao tiếp thường xuyên giữa các thành viên và khách hàng để điều chỉnh kịp thời.

Tiến độ:

- Waterfall: Thời gian phát triển dài hơn, do phải hoàn thành tất cả các giai đoạn theo trình tự.
- Agile: Tiến độ nhanh hơn, vì phần mềm có thể được phát hành và sử dụng sau mỗi vòng lặp ngắn.

Kiểm thử:

- Waterfall: Kiểm thử chỉ diễn ra sau khi phát triển hoàn tất, có thể gặp khó khăn trong việc phát hiện lỗi sớm.
- Agile: Kiểm thử diễn ra liên tục trong mỗi vòng lặp, giúp phát hiện lỗi sớm và điều chỉnh kịp thời.

Khách hàng:

- Waterfall: Khách hàng chỉ được tham gia vào giai đoạn đầu (yêu cầu) và cuối (kiểm thử) của dự án.
- Agile: Khách hàng được tham gia thường xuyên, phản hồi liên tục trong suốt quá trình phát triển.

2.Lợi ích của việc sử dụng UML trong thiết kế phần mềm là gì?

Lợi ích của việc sử dụng UML (Unified Modeling Language) trong thiết kế phần mềm bao gồm:

Mô tả rõ ràng và trực quan: UML cung cấp các sơ đồ trực quan giúp mô tả hệ thống phần mềm, dễ dàng hiểu và giao tiếp giữa các thành viên trong nhóm phát triển.

Tiết kiệm thời gian và chi phí: Việc sử dụng UML giúp xác định được các yêu cầu và thiết kế hệ thống rõ ràng từ đầu, giảm thiểu rủi ro và chi phí phát sinh do thay đổi trong quá trình phát triển.

Tính nhất quán và chuẩn hóa: UML cung cấp một hệ thống ký hiệu chuẩn, giúp đảm bảo tính nhất quán trong việc mô tả các thành phần của hệ thống phần mềm.

Dễ dàng bảo trì và mở rộng: UML hỗ trợ việc tái cấu trúc và mở rộng hệ thống dễ dàng hơn, vì mô hình rõ ràng giúp người phát triển hiểu rõ cấu trúc của hệ thống.

Giao tiếp hiệu quả: UML tạo ra một ngôn ngữ chung cho các nhà phát triển, người phân tích, và khách hàng, giúp cải thiện sự giao tiếp và giảm thiểu hiểu lầm.

Hỗ trợ các phương pháp phát triển phần mềm: UML có thể được áp dụng trong nhiều mô hình phát triển phần mềm như Waterfall, Agile, hay Spiral, hỗ trợ thiết kế hệ thống một cách linh hoạt và hiệu quả.

3.Làm thế nào để đảm bảo phần mềm có thể bảo trì tốt trong tương lai?

Viết mã nguồn rõ ràng và có cấu trúc tốt:

- Sử dụng các quy ước đặt tên biến, hàm dễ hiểu và nhất quán.
- Chia nhỏ mã thành các hàm hoặc module có chức năng cụ thể, tuân theo nguyên tắc "trách nhiệm duy nhất" (Single Responsibility Principle).
- Tránh lặp lại mã (DRY - Don't Repeat Yourself) bằng cách tái sử dụng các thành phần chung.

Tài liệu hóa mã nguồn và quy trình:

- Viết chú thích (comments) trong mã để giải thích các phần phức tạp hoặc logic quan trọng.
- Duy trì tài liệu hướng dẫn (README, wiki) về cách cài đặt, cấu hình và bảo trì phần mềm.
- Ghi lại các quyết định thiết kế lớn để người khác có thể hiểu lý do đằng sau.

Sử dụng kiểm soát phiên bản (Version Control):

- Dùng các công cụ như Git để theo dõi thay đổi mã nguồn, giúp dễ dàng quay lại phiên bản cũ nếu cần.
- Cam kết (commit) thường xuyên với thông điệp rõ ràng để mô tả từng thay đổi.

Áp dụng kiểm thử tự động (Automated Testing):

- Viết các bài kiểm tra đơn vị (unit tests), kiểm tra tích hợp (integration tests) và kiểm tra đầu cuối (end-to-end tests) để đảm bảo tính đúng đắn của phần mềm sau mỗi thay đổi.
- Tự động hóa quy trình kiểm tra bằng các công cụ CI/CD (Continuous Integration/Continuous Deployment).

Thiết kế phần mềm linh hoạt và mô-đun:

- Sử dụng các mẫu thiết kế (design patterns) phù hợp để tăng tính mở rộng và dễ bảo trì.
- Tách biệt các lớp chức năng (ví dụ: giao diện người dùng, logic nghiệp vụ, truy cập dữ liệu) để dễ thay đổi một phần mà không ảnh hưởng đến toàn bộ hệ thống.

Quản lý phụ thuộc (Dependencies):

- Giữ các thư viện và framework được cập nhật, nhưng kiểm tra kỹ trước khi nâng cấp để tránh xung đột.
- Sử dụng công cụ quản lý phụ thuộc (như npm, Maven, hoặc pip) để theo dõi và duy trì phiên bản.

Theo dõi và ghi log:

- Tích hợp hệ thống ghi log để dễ dàng phát hiện và sửa lỗi khi phần mềm gặp sự cố.
- Sử dụng các công cụ giám sát (monitoring) để theo dõi hiệu suất và trạng thái phần mềm trong thời gian thực

4.Tại sao các công ty phần mềm thường sử dụng mô hình phát triển lặp (Iterative Development)?

Linh hoạt và thích nghi tốt – Cho phép thay đổi yêu cầu trong quá trình phát triển mà không ảnh hưởng lớn đến toàn bộ dự án.

Phát hiện lỗi sớm – Kiểm thử và đánh giá từng phiên bản giúp phát hiện và sửa lỗi nhanh hơn.

Cải thiện chất lượng sản phẩm – Phản hồi từ khách hàng sau mỗi vòng lặp giúp tối ưu hóa sản phẩm liên tục.

Giảm rủi ro – Phát triển theo từng giai đoạn giúp kiểm soát tiến độ và giảm nguy cơ thất bại.

Đáp ứng nhanh nhu cầu thị trường – Có thể phát hành các phiên bản nhỏ trước khi hoàn thiện toàn bộ sản phẩm.

5.Vai trò của kiến trúc phần mềm trong việc xây dựng một hệ thống phần mềm phức tạp.

Kiểm thử phần mềm quan trọng vì:

- **Đảm bảo chất lượng sản phẩm**: Giúp phát hiện lỗi trước khi sản phẩm đến tay người dùng, giảm thiểu rủi ro.
- **Tiết kiệm chi phí sửa lỗi**: Càng phát hiện lỗi sớm, chi phí sửa chữa càng thấp. Nếu lỗi xuất hiện sau khi phát hành, chi phí sẽ cao hơn nhiều.
- **Nâng cao trải nghiệm người dùng**: Phần mềm ít lỗi, hoạt động ổn định giúp cải thiện trải nghiệm và giữ chân khách hàng.
- **Đảm bảo an toàn và bảo mật**: Giúp phát hiện lỗ hổng bảo mật như SQL Injection, XSS, CSRF, giảm nguy cơ bị tấn công.
- **Tăng tính tin cậy của phần mềm**: Đảm bảo phần mềm hoạt động đúng theo yêu cầu, tránh sự cố không mong muốn.

=> Kiểm thử phần mềm giúp nâng cao chất lượng, giảm rủi ro, tối ưu chi phí và đảm bảo phần mềm hoạt động ổn định, an toàn.

6.Làm thế nào để đảm bảo rằng một hệ thống phần mềm đáp ứng được yêu cầu bảo mật?

SOLID là 5 nguyên tắc giúp thiết kế mã nguồn dễ bảo trì, mở rộng và tránh lỗi trong lập trình hướng đối tượng (OOP):

- **Single Responsibility Principle (SRP) - Nguyên tắc trách nhiệm duy nhất**: Mỗi class chỉ nên có một lý do để thay đổi, tức là chỉ chịu trách nhiệm cho một chức năng duy nhất. Ví dụ, không nên viết một class vừa quản lý user vừa gửi email, mà nên tách thành `UserManager` và `EmailService`.

- **Open/Closed Principle (OCP) - Nguyên tắc đóng/mở**: Class nên có thể mở rộng nhưng không được sửa đổi trực tiếp. Điều này giúp hệ thống linh hoạt hơn, có thể thêm chức năng mới mà không làm ảnh hưởng đến mã nguồn cũ.

- **Liskov Substitution Principle (LSP) - Nguyên tắc thay thế Liskov**: Các class con phải có thể thay thế class cha mà không làm thay đổi hành vi của chương trình. Nếu một class con phá vỡ chức năng của class cha, tức là vi phạm nguyên tắc này.

- **Interface Segregation Principle (ISP) - Nguyên tắc phân tách interface**: Một interface không nên ép buộc class triển khai các phương thức không cần thiết. Thay vì một interface lớn chứa nhiều chức năng, nên tách thành nhiều interface nhỏ hơn để class chỉ cần triển khai những gì thực sự cần thiết.

- **Dependency Inversion Principle (DIP) - Nguyên tắc đảo ngược sự phụ thuộc**: Module cấp cao không nên phụ thuộc trực tiếp vào module cấp thấp.

Thay vào đó, cả hai nên phụ thuộc vào abstraction (interface hoặc abstract class) để tăng tính linh hoạt.

=> Những nguyên tắc này giúp viết mã dễ bảo trì, dễ mở rộng và hạn chế lỗi phát sinh khi thay đổi hoặc mở rộng phần mềm.

7.So sánh kiểm thử đơn vị (Unit Testing) và kiểm thử tích hợp (Integration Testing).

Kiểm thử đơn vị (Unit Testing):

- Kiểm tra từng đơn vị nhỏ nhất của phần mềm, như hàm, lớp, module.
- Do lập trình viên thực hiện trong quá trình phát triển.
- Tập trung vào logic nội bộ của từng đơn vị.
- Công cụ hỗ trợ: JUnit (Java), PyTest (Python), Google Test (C++), NUnit (C#),
- Phát hiện lỗi logic, lỗi biên dịch, lỗi thuật toán.
- Thực hiện sớm trong vòng đời phát triển phần mềm.

Kiểm thử tích hợp (Integration Testing):

- Kiểm tra sự tương tác giữa nhiều đơn vị/module trong hệ thống.
- Do nhóm kiểm thử thực hiện sau khi các đơn vị đã được kiểm thử đơn vị.
- Tập trung vào giao tiếp giữa các module, API, cơ sở dữ liệu, hệ thống bên ngoài.
- Công cụ hỗ trợ: Selenium, Postman, JMeter, TestNG, ...
- Phát hiện lỗi giao tiếp giữa các module, lỗi truyền dữ liệu, lỗi giao thức.
- Thực hiện sau kiểm thử đơn vị và trước kiểm thử hệ thống.

8.Những thách thức chính trong việc thu thập yêu cầu phần mềm là gì?

Yêu cầu không rõ ràng hoặc thay đổi liên tục

- Khách hàng không xác định rõ nhu cầu hoặc thay đổi yêu cầu trong quá trình phát triển.

Thiếu giao tiếp giữa các bên liên quan

- Nhà phát triển, tester và khách hàng có thể hiểu khác nhau về yêu cầu.

Giới hạn thời gian và ngân sách

- Không đủ thời gian hoặc tài nguyên để phân tích chi tiết và đầy đủ yêu cầu.

Khó khăn trong việc mô tả yêu cầu phi chức năng

- Hiệu suất, bảo mật, khả năng mở rộng khó định lượng chính xác.

Người dùng không có kiến thức kỹ thuật

- Khách hàng hoặc người dùng có thể đưa ra yêu cầu không thực tế hoặc khó hiểu về mặt kỹ thuật.

Sự phức tạp của hệ thống

- Hệ thống lớn, nhiều thành phần liên quan, gây khó khăn trong việc xác định đầy đủ yêu cầu.

Xung đột giữa các bên liên quan

- Các bộ phận khác nhau có thể có yêu cầu mâu thuẫn, cần đàm phán và thống nhất.

9.Cách áp dụng quy trình Scrum vào dự án phát triển phần mềm thực tế.

Scrum là một quy trình phát triển phần mềm theo mô hình **Agile**, giúp quản lý dự án linh hoạt, chia công việc thành các vòng lặp ngắn (**Sprint**). Để áp dụng Scrum vào thực tế, thực hiện các bước sau:

Bước 1: Xác định các vai trò trong nhóm Scrum

- **Product Owner (PO)**: Người chịu trách nhiệm về sản phẩm, định nghĩa yêu cầu và ưu tiên công việc.
- **Scrum Master**: Hướng dẫn nhóm làm việc theo Scrum, loại bỏ trở ngại.
- **Development Team**: Nhóm phát triển phần mềm (lập trình viên, tester, UI/UX designer…).

Bước 2: Xây dựng Product Backlog

- Liệt kê tất cả các tính năng và yêu cầu của sản phẩm dưới dạng **User Stories**.
- Product Owner sắp xếp mức độ ưu tiên để tập trung vào những tính năng quan trọng nhất.

Bước 3: Lập kế hoạch Sprint (Sprint Planning)

- Mỗi Sprint thường kéo dài từ **1-4 tuần**.
- Chọn các **User Stories** từ Product Backlog để thực hiện trong Sprint.
- Nhóm phát triển ước lượng thời gian và chia công việc thành các **Task** cụ thể.

Bước 4: Daily Scrum (Cuộc họp hằng ngày)

- Họp nhanh (~15 phút) để cập nhật tiến độ, kế hoạch trong ngày và các khó khăn gặp phải.

- Scrum Master hỗ trợ giải quyết vấn đề nếu có.

Bước 5: Hoàn thành Sprint & Review

- Khi Sprint kết thúc, nhóm trình bày sản phẩm cho Product Owner hoặc khách hàng.
- Tiến hành **Sprint Review** để kiểm tra kết quả và nhận phản hồi.

Bước 6: Sprint Retrospective (Cải tiến quy trình)

- Nhóm họp lại để đánh giá những gì đã làm tốt, chưa tốt và đề xuất cải tiến cho Sprint tiếp theo.

10.Tại sao việc tối ưu hóa mã nguồn lại quan trọng trong phát triển phần mềm?

Việc tối ưu hóa mã nguồn (**Code Optimization**) quan trọng vì:

1. **Cải thiện hiệu suất phần mềm**
    - Giúp ứng dụng chạy nhanh hơn, sử dụng ít tài nguyên CPU, RAM, ổ cứng hơn.
    - Đặc biệt quan trọng với hệ thống có nhiều người dùng hoặc xử lý dữ liệu lớn.
2. **Giảm tiêu thụ tài nguyên & tiết kiệm chi phí**
    - Code tối ưu giúp giảm chi phí phần cứng, cloud server và năng lượng tiêu thụ.
3. **Dễ bảo trì và mở rộng**
    - Code sạch, có cấu trúc rõ ràng giúp developer dễ đọc, hiểu và sửa lỗi.
    - Giúp mở rộng phần mềm dễ dàng hơn khi có yêu cầu mới.
4. **Giảm rủi ro lỗi & tăng tính ổn định**
    - Code rườm rà có thể gây lỗi khó phát hiện.
    - Việc tối ưu giúp loại bỏ đoạn code không cần thiết, tránh bug tiềm ẩn.
5. **Bảo mật tốt hơn**
    - Code tối ưu giúp loại bỏ những đoạn code dư thừa, tránh lỗ hổng bảo mật.
    - Ví dụ: Kiểm tra và tối ưu truy vấn SQL có thể ngăn chặn SQL Injection.

Tình huống

1. Bạn là một kỹ sư phần mềm trong một nhóm phát triển, khách hàng liên tục thay đổi yêu cầu. Bạn sẽ xử lý như thế nào?

Giao tiếp thường xuyên với khách hàng: Tôi sẽ giữ liên lạc chặt chẽ với khách hàng để hiểu rõ các yêu cầu thay đổi, lý do và ưu tiên của họ. Điều này giúp đảm bảo rằng chúng tôi đang phát triển đúng hướng và tránh hiểu lầm.

Áp dụng phương pháp Agile: Với các yêu cầu thay đổi liên tục, tôi sẽ đề xuất sử dụng mô hình phát triển Agile. Điều này giúp nhóm có thể điều chỉnh linh hoạt các yêu cầu mới trong từng vòng lặp ngắn (sprint) và dễ dàng đưa vào phản hồi của khách hàng trong quá trình phát triển.

Thiết lập phạm vi rõ ràng: Tôi sẽ cùng khách hàng xác định phạm vi và ưu tiên các yêu cầu để tránh việc thay đổi quá nhiều trong quá trình phát triển. Việc này giúp tránh làm gián đoạn tiến độ và giảm thiểu các yêu cầu không cần thiết.

Đảm bảo tài liệu và bản mô phỏng yêu cầu: Tôi sẽ ghi lại chi tiết các yêu cầu thay đổi và đảm bảo mọi thành viên trong nhóm đều hiểu rõ các thay đổi này. Việc tài liệu hóa giúp nhóm theo dõi tiến độ và đảm bảo không bỏ sót yêu cầu quan trọng.

Đánh giá tác động của thay đổi: Tôi sẽ đánh giá tác động của từng thay đổi đối với thời gian, chi phí và chất lượng phần mềm để thông báo cho khách hàng và điều chỉnh kế hoạch dự án nếu cần thiết.

Xây dựng bản demo hoặc prototype: Nếu có thể, tôi sẽ phát triển bản demo hoặc prototype để khách hàng có thể thấy trước sản phẩm và dễ dàng đưa ra phản hồi sớm hơn, giúp giảm thiểu việc thay đổi yêu cầu muộn trong quá trình phát triển.

2. Trong quá trình kiểm thử, bạn phát hiện một lỗi nghiêm trọng nhưng trưởng nhóm quyết định không sửa chữa. Bạn sẽ làm gì?

Nếu trong quá trình kiểm thử tôi phát hiện một lỗi nghiêm trọng nhưng trưởng nhóm quyết định không sửa chữa, tôi sẽ làm như sau:

- Xác định mức độ nghiêm trọng của lỗi: Tôi sẽ đánh giá lại mức độ ảnh hưởng của lỗi đối với sản phẩm, người dùng và các tính năng khác của hệ thống để đảm bảo rằng nó thực sự nghiêm trọng và cần được giải quyết ngay lập tức.

- Trình bày lại vấn đề với trưởng nhóm: Nếu lỗi nghiêm trọng, tôi sẽ cố gắng giải thích lại với trưởng nhóm về tác động của lỗi đối với chất lượng phần mềm, hiệu suất và trải nghiệm người dùng, đồng thời chỉ ra những nguy cơ nếu lỗi

không được sửa chữa.

- Đề xuất giải pháp tạm thời: Nếu trưởng nhóm vẫn quyết định không sửa chữa, tôi có thể đề xuất giải pháp tạm thời hoặc biện pháp giảm thiểu tác động của lỗi, nhằm đảm bảo rằng hệ thống vẫn hoạt động ổn định trong khi chờ sửa chữa sau này.

- Ghi nhận và báo cáo lại lỗi: Tôi sẽ ghi nhận lỗi và tác động của nó trong hệ thống theo dõi lỗi (bug tracking system), đồng thời yêu cầu xem xét lại lỗi trong các phiên bản sau hoặc khi có cơ hội.

- Thông báo với các bên liên quan: Nếu lỗi này có thể ảnh hưởng đến khách hàng hoặc người dùng cuối, tôi sẽ báo cáo lại cho các bên liên quan hoặc khách hàng (nếu cần) để họ có thể quyết định liệu có nên tiến hành sửa chữa ngay hay không.

3. Một dự án phần mềm gặp tình trạng chậm tiến độ do thay đổi yêu cầu liên tục từ khách hàng. Bạn sẽ đề xuất giải pháp gì?

Sử dụng mô hình phát triển linh hoạt (Agile) – Chia dự án thành các vòng lặp ngắn (Sprint) để thích ứng nhanh với thay đổi.

Xác định yêu cầu rõ ràng ngay từ đầu – Thực hiện các buổi họp với khách hàng để thống nhất yêu cầu cốt lõi, tránh thay đổi không cần thiết.

Ưu tiên hóa yêu cầu – Áp dụng phương pháp MoSCoW (Must-have, Should-have, Could-have, Won’t-have) để tập trung vào những tính năng quan trọng nhất.

Thiết lập quy trình kiểm soát thay đổi – Yêu cầu khách hàng xác nhận bằng văn bản với mỗi thay đổi để đánh giá tác động trước khi thực hiện.

Giao tiếp hiệu quả với khách hàng – Thường xuyên cập nhật tiến độ và giải thích tác động của việc thay đổi yêu cầu đến thời gian và chi phí.

4. Nhóm của bạn đang thiết kế cơ sở dữ liệu cho một hệ thống thương mại điện tử. Làm thế nào để đảm bảo thiết kế cơ sở dữ liệu không bị dư thừa?

Hiểu rõ yêu cầu nghiệp vụ

- Xác định các thực thể chính trong hệ thống (ví dụ: Khách hàng, Sản phẩm, Đơn hàng, Danh mục, Thanh toán).

- Phân tích mối quan hệ giữa các thực thể (ví dụ: Một khách hàng có thể đặt nhiều đơn hàng, một đơn hàng chứa nhiều sản phẩm).
- Điều này giúp bạn tránh lưu trữ dữ liệu không cần thiết ngay từ đầu.

Áp dụng chuẩn hóa (Normalization)

Chuẩn hóa là quá trình tổ chức dữ liệu để loại bỏ dư thừa và đảm bảo tính toàn vẹn. Các bước cơ bản bao gồm:

- **Chuẩn 1NF (First Normal Form)**:
  - Đảm bảo mỗi cột trong bảng chứa giá trị nguyên tử (không chứa danh sách hay tập hợp).
  - Ví dụ: Thay vì lưu "Sản phẩm mua: áo, quần" trong một cột, tạo bảng chi tiết đơn hàng với từng sản phẩm trên một dòng.
- **Chuẩn 2NF**:
  - Loại bỏ phụ thuộc hàm một phần bằng cách tách dữ liệu vào các bảng riêng nếu cần.
  - Ví dụ: Nếu bảng Đơn hàng có cột "Tên khách hàng" và "Địa chỉ", hãy tách thông tin khách hàng vào bảng Khách hàng riêng.
- **Chuẩn 3NF**:
  - Loại bỏ phụ thuộc bắc cầu.
  - Ví dụ: Nếu "Mã bưu điện" liên quan đến "Thành phố", hãy tách thông tin địa lý ra bảng riêng thay vì lặp lại trong bảng Đơn hàng.

Thông thường, chuẩn hóa đến 3NF là đủ cho hầu hết hệ thống thương mại điện tử, cân bằng giữa hiệu suất và tính không dư thừa.

Sử dụng khóa chính và khóa ngoại

- Mỗi bảng nên có một **khóa chính** (Primary Key) duy nhất để định danh bản ghi (ví dụ: ID khách hàng, ID đơn hàng).
- Sử dụng **khóa ngoại** (Foreign Key) để liên kết các bảng thay vì sao chép dữ liệu.
  - Ví dụ: Bảng Đơn hàng chỉ cần chứa "ID khách hàng" thay vì toàn bộ thông tin khách hàng.

Tránh lưu trữ dữ liệu trùng lặp

- Không lưu dữ liệu có thể tính toán được.
  - Ví dụ: Thay vì lưu cột "Tổng giá trị đơn hàng" trong bảng Đơn hàng, tính giá trị này từ bảng chi tiết đơn hàng (số lượng × giá sản phẩm).
- Tách dữ liệu tĩnh và động.

    - Ví dụ: Thông tin sản phẩm (tên, giá) nên nằm trong bảng Sản phẩm, không lặp lại trong bảng Đơn hàng.

Thiết kế mối quan hệ hợp lý

- **1-1 (One-to-One)**: Dùng khi hai thực thể có quan hệ chặt chẽ nhưng tách biệt (ví dụ: Thông tin đăng nhập và Hồ sơ khách hàng).
- **1-N (One-to-Many)**: Phổ biến trong thương mại điện tử (ví dụ: Một khách hàng có nhiều đơn hàng).
- **N-N (Many-to-Many)**: Sử dụng bảng trung gian (ví dụ: Bảng "Chi tiết đơn hàng" liên kết Đơn hàng và Sản phẩm).

Xem xét hiệu suất (Denormalization có kiểm soát)

- Trong một số trường hợp, dư thừa có thể được chấp nhận để tăng tốc độ truy vấn (Denormalization).
    - Ví dụ: Lưu "Tên sản phẩm" trong bảng Chi tiết đơn hàng để tránh JOIN liên tục với bảng Sản phẩm.
- Tuy nhiên, chỉ làm điều này khi cần thiết và có cơ chế đồng bộ dữ liệu (trigger hoặc job).

Kiểm tra và tối ưu hóa

- Sử dụng biểu đồ ER (Entity-Relationship Diagram) để hình dung thiết kế và phát hiện dư thừa.
- Thực hiện các truy vấn thử nghiệm để đảm bảo không có dữ liệu lặp không cần thiết.
- Đánh giá định kỳ khi hệ thống mở rộng để điều chỉnh thiết kế nếu cần.

5. Bạn cần lựa chọn giữa hai mô hình phát triển phần mềm: Waterfall và Agile. Bạn sẽ chọn mô hình nào cho một dự án startup công nghệ? Tại sao?

Em sẽ chọn **Agile** vì lý do sau:

- **Linh hoạt và thích nghi**: Startups thường hoạt động trong môi trường thay đổi nhanh, cần thích nghi với phản hồi của khách hàng và thị trường. Agile giúp điều chỉnh sản phẩm theo từng giai đoạn.
- **Phát hành nhanh**: Agile giúp đưa sản phẩm ra thị trường sớm hơn thông qua các vòng lặp phát triển ngắn (iterations/sprints). Điều này giúp startup kiểm tra ý tưởng nhanh và giảm rủi ro.
- **Hợp tác chặt chẽ**: Agile khuyến khích tương tác liên tục giữa nhóm phát triển, khách hàng và các bên liên quan, giúp cải thiện sản phẩm theo phản hồi thực tế.

- **Tối ưu tài nguyên**: Với Agile, startup có thể tập trung phát triển các tính năng quan trọng nhất thay vì lập kế hoạch chi tiết từ đầu như Waterfall.

=> Waterfall phù hợp hơn cho các dự án có yêu cầu rõ ràng, cố định ngay từ đầu và ít thay đổi.

6. Một dự án phần mềm lớn gặp vấn đề về hiệu suất. Những bước nào cần thực hiện để tối ưu hiệu suất phần mềm?

Để tối ưu hiệu suất cho một dự án phần mềm lớn, cần thực hiện các bước sau:

1. Phân tích & Định lượng vấn đề

- Sử dụng công cụ profiling (VD: Perf, New Relic, Prometheus, Grafana) để xác định bottleneck.
- Đo lường CPU, RAM, I/O, mạng, cơ sở dữ liệu, v.v.

2. Tối ưu mã nguồn & thuật toán

- Xem xét thuật toán có thể cải tiến không (ví dụ: thay thế $O(n^2)$ bằng $O(n \log n)$).
- Giảm thiểu vòng lặp thừa, tối ưu truy vấn database.

3. Cải thiện hiệu suất hệ thống & cơ sở dữ liệu

- Caching dữ liệu (Redis, Memcached) để giảm tải database.
- Indexing hợp lý trong cơ sở dữ liệu để tăng tốc truy vấn.
- Load balancing nếu hệ thống có nhiều traffic.

4. Tối ưu hạ tầng & triển khai

- Dùng CDN để tối ưu tải dữ liệu tĩnh.
- Sử dụng kiến trúc Microservices nếu cần mở rộng dễ dàng.
- Xác định bottleneck trên cloud/server và nâng cấp tài nguyên nếu cần.

5. Kiểm thử & Giám sát liên tục

- Thực hiện stress testing, load testing để đảm bảo hệ thống chịu tải tốt.
- Thiết lập monitoring (Prometheus, ELK stack) để theo dõi hiệu suất lâu dài.

7. Một hệ thống đang hoạt động có nhiều lỗi bảo mật. Bạn sẽ làm gì để tăng cường bảo mật mà không ảnh hưởng đến người dùng hiện tại?

- **Phân tích và đánh giá rủi ro**: Xác định các lỗ hổng bảo mật quan trọng nhất thông qua kiểm thử bảo mật (penetration testing, vulnerability scanning).
- **Cập nhật và vá lỗi**: Cập nhật phần mềm, thư viện và framework mà không làm gián đoạn dịch vụ (zero-downtime patching).
- **Tăng cường xác thực và ủy quyền**: Triển khai xác thực hai yếu tố (2FA), kiểm tra và giới hạn quyền truy cập.
- **Mã hóa dữ liệu**: Bảo vệ dữ liệu nhạy cảm bằng mã hóa mạnh (AES-256, TLS 1.2/1.3).
- **Giám sát và phát hiện xâm nhập**: Sử dụng hệ thống giám sát (SIEM, IDS/IPS) để phát hiện hoạt động bất thường.
- **Tăng cường bảo mật API**: Hạn chế truy cập API, sử dụng token bảo mật (JWT, OAuth 2.0).
- **Đào tạo nhân viên**: Nâng cao nhận thức về bảo mật, tránh tấn công phishing, social engineering.
- **Kiểm thử định kỳ**: Thực hiện kiểm thử bảo mật thường xuyên để phát hiện và xử lý lỗ hổng kịp thời.

8. Khi thiết kế phần mềm, nhóm của bạn có nhiều ý kiến trái ngược nhau về cách triển khai một tính năng. Làm thế nào để đưa ra quyết định tốt nhất?

- **Xác định tiêu chí đánh giá**: Đưa ra các tiêu chí cụ thể như hiệu suất, khả năng mở rộng, bảo trì, độ phức tạp.
- **Phân tích ưu nhược điểm**: So sánh từng giải pháp dựa trên các tiêu chí đã đặt ra.
- **Dựa vào dữ liệu thực tế**: Sử dụng số liệu, nguyên tắc thiết kế, kinh nghiệm từ dự án trước để đánh giá.
- **Nguyên mẫu (Prototype hoặc Proof of Concept - PoC)**: Xây dựng phiên bản thử nghiệm để kiểm tra tính khả thi.
- **Tham khảo ý kiến chuyên gia**: Nếu có tranh cãi lớn, có thể tham khảo ý kiến từ các chuyên gia hoặc cố vấn kỹ thuật.
- **Bỏ phiếu dân chủ**: Nếu không có phương án vượt trội, có thể để nhóm bỏ phiếu và chọn phương án phù hợp nhất.
- **Linh hoạt và thử nghiệm A/B**: Nếu có thể, triển khai thử nghiệm nhiều phương án để so sánh hiệu quả thực tế.
- **Dựa vào người ra quyết định chính**: Nếu tranh luận kéo dài, trưởng nhóm hoặc kiến trúc sư phần mềm sẽ đưa ra quyết định cuối cùng.

9. Khách hàng yêu cầu hệ thống phải có giao diện thân thiện với người dùng. Bạn sẽ làm gì để đảm bảo hệ thống đáp ứng tiêu chí này?
Nghiên cứu và hiểu người dùng:

- Tìm hiểu đối tượng người dùng chính (độ tuổi, thói quen, trình độ công nghệ) thông qua phỏng vấn khách hàng hoặc khảo sát người dùng cuối.
- Xác định nhu cầu cụ thể của họ để giao diện đáp ứng đúng mong đợi.

Thiết kế giao diện đơn giản và trực quan:

- Sử dụng bố cục rõ ràng, tránh quá tải thông tin trên một màn hình.
- Đặt các chức năng quan trọng ở vị trí dễ tiếp cận (ví dụ: nút "Lưu" hoặc "Tìm kiếm" ở nơi nổi bật).
- Áp dụng nguyên tắc tối thiểu hóa số lần nhấp chuột để hoàn thành một tác vụ.

Sử dụng màu sắc và phông chữ phù hợp:

- Chọn bảng màu hài hòa, không gây mỏi mắt (ví dụ: độ tương phản đủ cao giữa nền và chữ).
- Sử dụng phông chữ dễ đọc, kích thước phù hợp (ít nhất 16px cho nội dung chính).

Đảm bảo tính nhất quán:

- Duy trì phong cách thiết kế đồng bộ trên tất cả các màn hình (ví dụ: cùng kiểu nút, biểu tượng, cách sắp xếp).
- Sử dụng các mẫu giao diện quen thuộc để người dùng không phải học lại cách sử dụng.

Kiểm tra và lấy phản hồi:

- Tạo nguyên mẫu (prototype) giao diện và thử nghiệm với một nhóm người dùng thực tế.
- Thu thập ý kiến phản hồi để điều chỉnh trước khi phát triển hoàn chỉnh.

Hỗ trợ tính tương thích và khả năng truy cập:

- Đảm bảo giao diện hoạt động tốt trên nhiều thiết bị (máy tính, điện thoại, máy tính bảng).
- Tích hợp các tính năng hỗ trợ người khuyết tật (ví dụ: hỗ trợ đọc màn hình, phím tắt

10.Công ty bạn vừa tiếp nhận một dự án phần mềm cũ, nhưng không có tài liệu hướng dẫn. Bạn sẽ làm gì để hiểu và tiếp tục phát triển hệ thống này?

Khám phá mã nguồn: Bắt đầu bằng việc xem xét cấu trúc mã nguồn. Tôi sẽ xác định các thư mục, tệp chính, và cách tổ chức tổng thể để có cái nhìn sơ bộ về kiến trúc hệ thống.
Chạy thử hệ thống: Cài đặt và chạy phần mềm trong môi trường phát triển để quan sát cách nó hoạt động. Điều này giúp tôi hiểu chức năng chính và luồng hoạt động của hệ thống từ góc độ người dùng.
Phân tích phụ thuộc: Sử dụng các công cụ như trình quản lý gói (ví dụ: Maven, npm) hoặc tự động kiểm tra các thư viện, framework được sử dụng để hiểu công nghệ nền tảng và các thành phần bên ngoài.
Đọc mã chi tiết: Tập trung vào các phần mã quan trọng như điểm bắt đầu (entry points), các hàm chính, hoặc các lớp điều khiển (controllers) để nắm được logic cốt lõi. Tôi sẽ chú ý đến bình luận trong mã (nếu có) và đặt tên biến/hàm để suy ra ý định của lập trình viên trước đó.
Tạo sơ đồ luồng dữ liệu: Dựa trên việc đọc mã và chạy thử, tôi sẽ phác thảo cách dữ liệu di chuyển qua hệ thống, bao gồm đầu vào, xử lý, và đầu ra. Điều này giúp hình dung cách các thành phần kết nối với nhau.
Xác định các điểm không rõ ràng: Ghi chú lại những phần khó hiểu hoặc phức tạp trong mã để tập trung làm rõ sau, có thể bằng cách thử nghiệm hoặc thảo luận với nhóm.
Thử nghiệm từng phần: Viết hoặc chạy các bài kiểm tra (unit tests) nếu có sẵn, hoặc tạo các thử nghiệm nhỏ để kiểm tra chức năng của từng mô-đun. Điều này giúp xác nhận cách hoạt động của chúng mà không cần tài liệu.
Tìm kiếm thông tin bổ sung: Nếu có thể, liên hệ với những người từng làm việc trên dự án (nhà phát triển cũ, khách hàng) để hỏi về các chi tiết không rõ. Ngoài ra, tôi có thể tra cứu thông tin trên web hoặc cộng đồng nếu hệ thống sử dụng công nghệ phổ biến.
Tài liệu hóa dần dần: Trong quá trình khám phá, tôi sẽ ghi lại những gì mình hiểu được (chức năng, kiến trúc, các lưu ý) để tạo tài liệu cơ bản cho nhóm và cho chính mình khi phát triển tiếp.
Lên kế hoạch phát triển: Sau khi nắm được hệ thống, tôi sẽ xác định các ưu tiên phát triển, như sửa lỗi, tối ưu hóa, hoặc thêm tính năng, dựa trên yêu cầu mới và tình trạng hiện tại của phần mềm.