NGUYỄN THẾ HUY

SPEC DRIVEN DEVELOPMENT

CÁCH TEAM ENGINEERING HIỆN ĐẠI LÀM VIỆC VỚI AI

Khi AI trở thành một phần của SDLC

https://huynt.dev

Chào bạn, và cám ơn bạn đã đọc ebook này của mình.

Mình là Huy, hiện tại mình là một Technical Leader/Principal Engineer với hơn 10 năm kinh nghiệm. Mình đã viết khoảng gần 10 đầu ebook khác nhau xoay quanh các kiến thức về lập trình, với trên 20.000 lượt download tại website của mình là https://huynt.dev.

Ebook này của mình nhằm cung cấp cho bạn những câu chuyện rất thật khi áp dụng Spec Driven Development: một phương pháp làm việc đang ngày càng trở nên phổ biến trong thời đại AI.

Dù bạn là người mới bắt đầu hay đã có kinh nghiệm, mình tin những kiến thức trong ebook này sẽ giúp bạn rất nhiều trong việc phát triển dự án hiệu quả hơn với AI. Đặc biệt mình hy vọng cả các bạn ở vị trí non-tech như PO hay BA cũng có thể đọc và hiểu được quy trình phát triển phần mềm hiện đại ngày nay với AI.

Chúc bạn có những giây phút thú vị, bổ ích khi đọc ebook.

Lời nói đầu

Lần đầu mình để Claude viết gần như trọn một module xác thực, mình vừa thích vừa hơi sợ.

Chỉ trong khoảng mười phút, nó tạo controller, service, repository, migration, test, rồi tiện tay viết luôn một file README khá chỉn chu. Nếu nhìn riêng pull request đó, mọi thứ trông rất ổn: code chạy, test xanh, tên hàm cũng sạch. Mình push lên, mở PR, rồi đi pha ấm trà với cảm giác vừa tiết kiệm được hai ngày làm việc.

Hai tuần sau, một người dùng không reset được mật khẩu.

Mình mở code ra đọc lại và bắt đầu thấy khó chịu. Có một nhánh `if` xử lý trường hợp token hết hạn, nhưng cách xử lý không giống thứ mình nhớ trong đầu, và mình không chắc nhánh đó đến từ đâu: do mình từng nói với Claude, do Claude tự suy ra, hay do một yêu cầu nào đó mình đã quên?

Mình hỏi lại Claude và nó trả lời rất tự tin. Vấn đề là câu trả lời lần này khác lần trước; cả hai đều nghe hợp lý, nhưng không câu nào chứng minh được ý định ban đầu của team.

Đó là lúc mình nhận ra một điều hơi đau: mình đang xây một hệ thống mà chính mình không còn thật sự sở hữu. Không phải vì code quá tệ, nhìn nó còn khá đẹp; vấn đề là ý định phía sau code đã biến mất.

Vài tháng sau, mình đọc về Spec-Driven Development. Tên nghe khá nghiêm túc, thậm chí hơi gợi lại mấy thứ thời waterfall như tài liệu dày, ký duyệt nhiều tầng, rồi mới được viết dòng code đầu tiên. Nhưng đọc kỹ hơn thì SDD không nói như vậy. Ý chính đơn giản hơn nhiều: trước khi để AI viết code, hãy viết rõ ra mình muốn gì. Rõ ở mức AI hiểu được, dev mới vào team hiểu được, và PO không code cũng có thể đọc rồi nói: "Đúng, đây là thứ mình cần."

Cuốn ebook này là những ghi chép mình ước gì có sớm hơn. Một phần đến từ AI Unified Process, một phần từ GitHub Spec Kit, một phần từ DDD và Hexagonal Architecture. Nhưng phần đáng giá nhất đến từ những lần mình và team để AI chạy quá nhanh, rồi phải quay lại tìm xem ý định ban đầu đã rơi ở đâu.

Cuốn sách này dành cho ai

Mình viết cuốn này cho ba nhóm người.

Nhóm đầu tiên là **dev và Tech Lead**. Nếu bạn đang dùng Claude Code, Copilot, Cursor, Codex hay một AI agent nào đó mỗi ngày, rất có thể bạn đã thấy cảm giác này: tuần đầu mọi thứ cực nhanh, nhưng vài tháng sau sửa một feature lại chậm bất thường. Sách này không bảo bạn dùng AI ít đi; nó chỉ đưa ra một cách để dùng AI mà vẫn giữ được ngữ cảnh.

Nhóm thứ hai là **Engineering Manager**. Bạn có thể đang bị hỏi: "Team mình dùng AI hiệu quả chưa?" Câu hỏi đó khó trả lời nếu chỉ nhìn vào số dòng code hoặc số PR. SDD cho bạn vài thứ dễ quan sát hơn: tỷ lệ use case có spec, tỷ lệ AC có test, khả năng truy ngược từ code về yêu cầu, thời gian onboard người mới, và độ an toàn khi để AI sinh lại code mà không làm lệch hành vi.

Nhóm thứ ba là **PO, BA, và những người không code**. Vai trò của bạn không nhỏ đi trong thời AI; ngược lại, nó quan trọng hơn. Khi AI có thể viết code rất nhanh, câu hỏi lớn nhất sẽ là: ai đảm bảo nó đang viết đúng thứ cần viết? Spec vì vậy không còn là tài liệu kỹ thuật dành riêng cho dev, mà là chỗ nghiệp vụ và kỹ thuật gặp nhau.

Nên đọc thế nào

Bạn có thể đọc tuần tự, nhưng không bắt buộc.

Chương 1 đến Chương 3 trả lời câu hỏi "vì sao": vì sao vibe coding nhanh nhưng nguy hiểm, spec thật ra là gì, và sáu nguyên tắc nào giúp team dùng AI có kiểm soát hơn.

Chương 4 và Chương 5 là phần thực hành. Một chương cho dự án mới bắt đầu từ con số không, một chương cho hệ thống cũ năm năm tuổi không có tài liệu. Đọc cái nào trước cũng được, tùy bạn đang đứng ở đâu.

Chương 6 là phần vận hành team: ai làm gì, repo xếp ra sao, đo bằng chỉ số nào, và áp dụng theo lộ trình thế nào để không biến SDD thành một phong trào ngắn hạn.

Phần Kết sẽ nói thật một chuyện ít người nói: khi nào không nên dùng SDD. Vì không có phương pháp nào hợp với mọi tình huống.

Cuối sách có Phụ lục gồm mẫu Use Case, mẫu Business Requirement, cheatsheet câu lệnh, và một bảng thuật ngữ dành cho người mới.

Một lời hứa

Mình sẽ không nói lý thuyết suông. Mỗi chương đều có một tình huống cụ thể, có tên nhân vật, có tên team, có deadline thật và có sai lầm thật. Các tình huống đó là tổng hợp từ nhiều dự án mình và bạn bè đã đi qua. Tên thì đổi, nhưng vấn đề thì không.

Mình cũng sẽ không tô hồng AI. AI là công cụ rất tốt khi có một cái khung để làm việc; còn khi không có khung, nó chạy rất nhanh nhưng dễ đưa cả team tới một nơi không ai thật sự chọn. Cuốn sách này là về cái khung đó.

Bắt đầu thôi.

Chương 1. Vibe Coding và cái bẫy của tốc độ

1.1. Chuyện của Nam và Linh

Cách đây hơn một năm, mình ngồi cà phê với Nam, lúc đó đang là Tech Lead của một fintech startup tám người. Team của Nam vừa nhận một nhiệm vụ khá căng: trong hai tháng phải ship MVP cho một ứng dụng cho vay tiêu dùng nhỏ.

Founder đã hứa deadline với nhà đầu tư, Linh là PO đứng giữa nghiệp vụ và kỹ thuật, còn Nam thì đứng giữa Linh và một codebase chưa tồn tại.

Nam kể hôm đó rằng anh muốn "thử AI một cách nghiêm túc", không phải chỉ autocomplete vài dòng, mà để AI làm hẳn một phần feature. Anh mở Claude Code và gõ đại ý:

> Tạo service KYC, nhận CMND và ảnh selfie, gọi API verify, trả kết quả.

Khoảng hai mươi phút sau, anh có controller, service layer, integration với provider giả lập, test và swagger doc. Nhìn qua mọi thứ rất ổn, đến mức Linh sang xem demo xong chỉ gật đầu: "Đẹp quá. Cứ thế này hai tháng là dư."

Hai tháng đó, họ ship được thật.

Team vui, founder vui, Nam đăng LinkedIn được vài trăm like. Nếu câu chuyện dừng ở đây, nó sẽ là một case study rất đẹp về AI coding.

Nhưng vấn đề bắt đầu ở tháng thứ ba.

Một khách hàng đã KYC xong nhưng app vẫn báo chưa xác thực. Nam mở service KYC ra đọc lại và thấy một flag tên `verification_status` được set ở ba chỗ khác nhau, mỗi chỗ theo một logic hơi khác. Không ai nhớ vì sao lại có ba chỗ như vậy. Spec không có, decision record cũng không, còn đoạn chat Slack nói về chuyện đó thì đã trôi đi đâu mất.

Nam hỏi lại Claude. Claude trả lời rất tự tin, nhưng đó chỉ là một lời giải thích hợp lý được dựng lại từ chính đoạn code hiện tại, không phải ý định nghiệp vụ ban đầu.

Tháng thứ tư, có một thay đổi luật nhỏ. Hồ sơ KYC cần thêm video xác thực sống. Nam estimate ba ngày, nhưng cuối cùng mất mười một ngày, chủ yếu không phải để viết code mới mà để hiểu lại phần code AI đã viết hộ mình vài tháng trước.

Linh hỏi một câu rất khó chịu:

> Sao hồi tháng 2 cùng một feature làm hai ngày, giờ làm hai tuần?

Nam không có câu trả lời gọn.

Sự thật là team đã đi rất nhanh ở đoạn đầu, nhưng không để lại đủ dấu vết để chính họ quay lại an toàn. Thứ họ thiếu không phải code, mà là ngữ cảnh.

1.2. Vibe coding là gì

Gần đây cộng đồng dev hay dùng từ **vibe coding** để chỉ kiểu làm việc như vậy. Bạn ngồi cạnh AI, nói chuyện kiểu:

> Thử thêm cái này xem.
>
> Ừ chưa đúng, đổi thành thế kia.
>
> Nhìn ổn đấy, merge thôi.

Cảm giác rất giống pair với một người cực giỏi và cực nhanh, chỉ khác ở một điểm quan trọng: người đó thường không dừng lại để hỏi bạn những câu khó.

Một dev junior cẩn thận có thể sẽ hỏi:

> Nếu user chưa đủ 18 tuổi thì reject luôn hay cho phép người bảo lãnh?

AI thì thường chọn một hướng nghe hợp lý, viết code, rồi đi tiếp. Nếu prompt thiếu rule, nó sẽ lấp chỗ trống bằng xác suất. Với prototype thì có thể chấp nhận được, nhưng với hệ thống xử lý tiền, hợp đồng, quyền truy cập, định danh hoặc dữ liệu khách hàng, cách làm đó rất nguy hiểm.

Vibe coding có ba vấn đề lớn.

Vấn đề đầu tiên là yêu cầu nằm trong đầu người, không nằm trong hệ thống. Khi Linh nói "cho vay tiêu dùng nhỏ", trong đầu cô có rất nhiều ràng buộc về tuổi, thu nhập, hạn mức, lịch sử nợ, trạng thái KYC, và chính sách từ chối. Nam nhớ được một phần, AI đoán thêm một phần, và code cuối cùng trở thành hỗn hợp giữa nghiệp vụ thật, trí nhớ của dev, và phỏng đoán của model.

Vấn đề thứ hai là AI không có ký ức dài hạn của team. Nó không nhớ cuộc họp tuần trước, không biết vì sao Linh từng nói "tạm thời chưa làm video KYC", cũng không phân biệt được một rule là bắt buộc pháp lý hay chỉ là quyết định tạm trong MVP. Nếu những điều đó không được viết ra, với AI coi như chúng không tồn tại.

Vấn đề thứ ba là test do AI sinh ra dễ tạo cảm giác an toàn giả. Khi AI viết code rồi tự viết test dựa trên chính code đó, test thường chỉ chứng minh rằng "code đang làm đúng cái code đang làm", chứ chưa chắc chứng minh được nó đang làm đúng điều business cần.

Kết quả là một loại nợ kỹ thuật mới mà mình hay gọi là **code không có ngữ cảnh**. Code có thể sạch và test có thể xanh, nhưng khi cần sửa, không ai trả lời được câu hỏi quan trọng nhất: vì sao đoạn này tồn tại?

> Nếu bạn là PO hoặc EM và không rành kỹ thuật, có một câu hỏi đáng hỏi team mỗi tháng: *"Có ai giải thích được vì sao module quan trọng nhất đang chạy như hiện tại không?"* Nếu câu trả lời thường là "Claude viết, để hỏi lại Claude", team đang ở đoạn đầu của câu chuyện Nam và Linh.

1.3. Vì sao spec quay lại trung tâm

Mười năm trước, "viết spec" là cụm từ nhiều dev không thích. Nó gợi tới waterfall, tài liệu dày, ký duyệt nhiều tầng, và cảm giác phải viết rất nhiều trước khi được làm gì thật.

Agile phản ứng lại điều đó bằng user story ngắn, hội thoại liên tục và feedback nhanh. Trong một thế giới mà dev là người trực tiếp viết phần lớn code, cách đó khá hợp lý, vì dev có thể giữ nhiều ngữ cảnh trong đầu, chạy sang hỏi PO khi cần, rồi tích lũy hiểu biết qua nhiều sprint.

Nhưng khi AI bắt đầu viết một phần đáng kể code, giả định đó yếu đi.

AI không tự sống trong team của bạn, nên nó không biết Slack thread nào quan trọng, không biết câu nói trong buổi refinement là quyết định cuối hay chỉ là suy nghĩ nháp. Mọi thứ bạn muốn AI hiểu về domain đều phải được đưa vào ngữ cảnh theo một cách nào đó.

Spec quay lại không phải vì chúng ta muốn quay về waterfall, mà vì nó trở thành giao diện làm việc giữa con người và AI. Khi spec rõ, AI có chỗ bám để sinh code, dev mới có tài liệu để đọc, PO có nơi để phản biện, và test có cơ sở để hình thành. Ngược lại, khi spec mơ hồ, mọi thứ phía sau cũng mơ hồ theo; điểm khác biệt duy nhất là sự mơ hồ đó được chuyển thành code rất nhanh.

1.4. Spec-Driven Development là gì

Nếu nói thật ngắn, mình định nghĩa SDD như sau:

> *Spec đứng ở giữa để con người thống nhất ý định; AI viết code, test và doc xoay quanh nó. Khi có gì sai, sửa spec trước rồi mới sửa code.*

Tách câu đó ra, có bốn việc đáng giữ lại.

Spec là nơi team thống nhất "mình đang làm gì", thay vì để câu trả lời nằm rải rác trong Slack thread, trí nhớ của Tech Lead, hoặc đoạn code AI vừa sinh.

AI là người thợ rất nhanh, không phải người chịu trách nhiệm cuối cùng về ý định sản phẩm. Nó có thể viết code, sinh test, tạo migration, đề xuất refactor. Nhưng rule nghiệp vụ, đánh đổi về kiến trúc, hay chính sách có hệ quả về tiền bạc hoặc pháp lý vẫn phải có con người xác nhận.

Spec không cần hoàn hảo từ ngày đầu, và SDD cũng không yêu cầu bạn ngồi ba tuần viết tài liệu rồi mới code. Spec vòng đầu chỉ cần đủ rõ để bắt đầu. Sau demo, test, bug hoặc phản hồi, nó được sửa tiếp và lớn lên cùng hiểu biết của team.

Test đóng vai trò như hàng rào bảo vệ. Khi spec đã có Acceptance Criteria, test có thể map vào từng AC, nhờ đó bạn có thể để AI regenerate code mà vẫn biết các hành vi quan trọng có bị vỡ hay không.

Toàn bộ phần còn lại của cuốn sách chỉ là mở rộng bốn ý này vào nhiều tình huống khác nhau.

1.5. Vibe Coding và SDD đặt cạnh nhau

Bảng dưới đây là cách mình hay giải thích nhanh cho team mới:

Khía cạnh	Vibe Coding	Spec-Driven Development
Tốc độ tuần 1	Rất nhanh	Chậm hơn một chút vì phải viết spec đầu tiên
Tốc độ tháng 3	Chậm dần vì ngữ cảnh mất	Ổn định hơn vì ngữ cảnh nằm trong spec
Onboard dev mới	"Đọc code rồi hỏi mình"	"Đọc `/specs`, rồi pair một buổi"
Refactor	Sợ, vì không biết phá rule nào	Tự tin hơn, vì test bảo vệ AC
Đổi AI tool	Dễ phụ thuộc vào chat history hoặc tool cũ	Spec vẫn giữ được, tool có thể đổi
Vai trò của PO	Người đặt hàng từ xa	Đồng tác giả của spec
Khi có bug production	Đọc code, hỏi AI, đoán	Đọc spec, tìm AC, sửa spec/test/code theo thứ tự

Điểm cần nói thẳng là SDD thường chậm hơn vibe coding ở tuần đầu, vì bạn phải ngồi viết BR, use case, entity model trước khi để AI sinh code. Đầu tư đó trả lại ở tháng thứ ba trở đi, khi refactor không còn sợ, khi dev mới onboard trong vài ngày thay vì vài tuần, và khi bug production có thể truy ngược về một mục cụ thể trong spec. Nếu dự án của bạn chỉ chạy một tuần rồi bỏ, đầu tư này không kịp thu hồi; mình sẽ nói kỹ chuyện đó ở Phần Kết.

1.6. SDD không phải quay về waterfall

Khi mình nói chữ "spec" với một dev đã quen agile, phản ứng đầu tiên thường là dị ứng. Họ hình dung ra mấy quyển tài liệu in giấy A4 dày hai trăm trang, ký duyệt qua ba tầng, mất sáu tháng mới được viết dòng code đầu tiên. Đó không phải SDD, và mình hiểu vì sao có sự dị ứng đó.

SDD trong thực hành nhẹ hơn rất nhiều so với hình dung trên. Spec viết bằng markdown, sống trong git, review qua pull request như code. Một spec hoàn chỉnh đầu tiên chỉ cần một đến hai trang, không phải một trăm. Khi spec sai thì sửa, không phải "duyệt lại" qua các tầng. Và spec được kiểm chứng bằng Acceptance Criteria, không phải bằng chữ ký.

SDD giữ lại phần đáng giá của requirement engineering: rõ ràng, truy được, và có thể kiểm thử. Nhưng nó bỏ đi gánh nặng của waterfall: phê duyệt cứng nhắc, thiết kế trước mọi chi tiết, và sửa đổi tốn kém. Cái làm cho điều này khả thi chính là AI, vì AI gánh phần "viết và sinh lại code" mà trước đây khiến việc viết spec chi tiết trở nên không kinh tế.

1.7. Ba câu hỏi để mang về team

Mình không yêu cầu bạn tin SDD ngay lập tức. Mình chỉ xin bạn dành năm phút trả lời ba câu hỏi với team của mình.

Câu thứ nhất: nếu một dev mới vào team tuần sau, làm sao họ biết được logic nghiệp vụ của module quan trọng nhất? Nếu câu trả lời là "đọc code rồi hỏi", đó chính là vibe coding với một cái tên khác.

Câu thứ hai: nếu bạn cần refactor một service mà AI từng viết, bạn dựa vào đâu để biết mình không phá hỏng gì? Nếu câu trả lời là "test có sẵn", hãy kiểm tra xem các test đó đang kiểm thử hành vi nghiệp vụ, hay chỉ kiểm tra cách cài đặt cụ thể.

Câu thứ ba: nếu PO sửa một yêu cầu, bạn có biết những file code nào cần đụng tới không? Nếu không, bạn đang thiếu khả năng truy ngược từ code về yêu cầu, và đó là một trong sáu nguyên tắc mình sẽ nói ở Chương 3.

Nếu cả ba câu đều khiến bạn hơi ngập ngừng, đừng lo. Đa số team mình từng gặp đều bắt đầu từ chỗ này. Chương 2 sẽ đi từ thứ cơ bản nhất: spec thật ra là cái gì, và vì sao mấy khái niệm tưởng cũ như DDD và Hexagonal lại bỗng dưng cực kỳ hợp với thời AI.

Chương 2. Spec là gì, và vì sao DDD với Hexagon trở lại đúng lúc

2.1. Chuyện user story ba dòng

Linh, PO trong câu chuyện Chương 1, sau vài tháng làm với AI đã cẩn thận hơn. Khi chuẩn bị feature thanh toán bằng QR, cô viết một user story:

> *Là khách hàng, tôi muốn đặt hàng bằng QR Code, để thanh toán nhanh hơn.*

Ba dòng, rất quen thuộc. Nếu làm agile lâu, bạn đã thấy kiểu user story này rất nhiều lần.

Dev đọc thấy ổn, AI đọc cũng không có vẻ gì bối rối. Hai tuần sau demo, cả hai bên ngồi nhìn màn hình:

> "Sao QR hết hạn sau 5 phút? Chính sách của mình là 15 phút mà."
>
> "Linh có nói đâu."
>
> "Tưởng default là 15 phút chứ."
>
> "Default của ai?"
>
> "Claude chọn…"

Đây là kiểu lỗi mình gặp ngày càng nhiều trong các team dùng AI. Code không hẳn sai theo nghĩa kỹ thuật; nó chạy đúng theo một giả định nào đó. Vấn đề là mỗi bên lại đang ngầm hiểu một kiểu: PO nghĩ chính sách là 15 phút, dev tưởng chưa nói thì để mặc định, còn AI chọn một con số nghe hợp lý.

User story ba dòng không sai, nhưng nó chỉ là điểm bắt đầu cho một cuộc trò chuyện. Trước đây, cuộc trò chuyện đó diễn ra giữa PO và dev; bây giờ, nếu AI là người viết code, cuộc trò chuyện ấy cần được ghi lại rõ hơn, nếu không AI sẽ tự điền phần còn thiếu.

2.2. Bốn tầng của yêu cầu

Một lỗi phổ biến là gọi mọi tài liệu trong dự án là "spec". Thật ra có vài tầng khác nhau. Bạn không cần biến chúng thành bốn bộ tài liệu cồng kềnh, nhưng nên hiểu mỗi tầng đang trả lời một câu hỏi riêng.

Tầng 1: Business Requirement (BR)

BR trả lời câu hỏi: **vì sao làm việc này?**

Ở tầng này, nội dung nên ngắn, đo được, và chưa cần nói nhiều tới giải pháp.

```
# BR-007: Tăng tỷ lệ checkout thành công

## Goal
Tăng tỷ lệ đặt hàng thành công của khách mới từ 62% lên 80%
trong Q3.

## Success Metric
- conversion_rate (paid_orders / started_checkouts)
- Đo qua dashboard Analytics, segment "new_customer = true"

## Out of Scope
- Khách cũ (sẽ làm ở BR-008)
- Thanh toán quốc tế
```

BR chủ yếu là việc của PO hoặc người phụ trách nghiệp vụ. Dev nên đọc được nó, nhưng ở tầng này chưa cần nghĩ tới code.

Tầng 2: Use Case

Use Case trả lời câu hỏi: **ai làm gì với hệ thống?**

Một BR thường sinh ra một hoặc vài use case. Mỗi use case nên có actor, trigger, main flow, alternative flow, exception.

```
# UC-042: Đặt hàng bằng QR Code

## Actor
Khách hàng (đã có app, chưa hoặc đã login)

## Trigger
```

```
Khách bấm "Thanh toán QR" ở giỏ hàng

## Main Flow
1. Hệ thống tạo QR code chứa order_id + amount + expiry
2. Hệ thống hiển thị QR + bộ đếm thời gian còn lại
3. Khách dùng app ngân hàng quét QR
4. Hệ thống nhận webhook từ payment gateway
5. Hệ thống xác nhận order, gửi email + push notification

## Alternative Flows
- 2a. Khách chưa login: chuyển sang flow đăng nhập nhanh (UC-005)
- 4a. Khách thanh toán ngoài app: webhook delay > 30s, hiển thị
  "đang xác nhận"

## Exceptions
- E1. QR hết hạn (>15 phút): hiển thị "QR đã hết hạn, vui lòng tạo lại"
- E2. Số tiền không khớp: refund tự động, log incident
- E3. Webhook không tới sau 30 phút: chuyển order sang trạng thái
  "manual_review"
```

Ở tầng này, AI đã cần đọc để viết code, còn PO vẫn có thể góp ý mà không cần hiểu database, framework hay kiến trúc hệ thống.

Tầng 3: Entity Model

Entity Model trả lời câu hỏi: **hệ thống đang nói về những "danh từ" nào?**

Entity model không phải ERD; nó chưa cần nói về index, datatype hay foreign key, mà tập trung vào tên gọi, ý nghĩa, trạng thái và quan hệ.

```
# Entity Model — Checkout Context

## Order
- Đại diện: một lần khách đặt hàng
- Trạng thái: draft, awaiting_payment, paid, cancelled, manual_review
```

```
- Có nhiều: OrderItem
- Có một: Payment (sau khi paid)

## OrderItem
- Một dòng trong đơn
- Thuộc: Order
- Tham chiếu: Product

## Payment
- Một giao dịch thanh toán
- Thuộc: Order
- Phương thức: qr_code, credit_card, wallet
- Trạng thái: pending, success, failed, refunded

## QRSession
- Một lần khách yêu cầu QR
- Thuộc: Order
- Hết hạn sau 15 phút
- Một Order có thể có nhiều QRSession (nếu tạo lại)
```

Tầng này thường bị đánh giá thấp, nhưng lại ảnh hưởng rất nhiều tới chất lượng code AI sinh ra. Nếu spec dùng đúng tên nghiệp vụ, AI có cơ hội sinh ra class, function và test gần với cách team nói chuyện hằng ngày hơn.

Tầng 4: Acceptance Criteria

Acceptance Criteria trả lời câu hỏi: **làm sao biết là xong đúng?**

Tầng này nối spec trực tiếp sang test.

```
# UC-042 — Acceptance Criteria

## AC-1: Tạo QR thành công
Given: khách có giỏ hàng > 0 đồng, đã chọn QR
When: khách bấm "Thanh toán QR"
```

```
Then: hệ thống tạo QRSession với expiry = now + 15 phút
And: trả về ảnh QR + bộ đếm thời gian

## AC-2: Thanh toán thành công trong thời gian hợp lệ
Given: QRSession còn 5 phút
When: payment gateway gửi webhook success với đúng order_id, đúng amount
Then: Order chuyển trạng thái paid
And: Email xác nhận được gửi trong vòng 10 giây

## AC-3: QR hết hạn
Given: QRSession đã quá 15 phút
When: payment gateway gửi webhook success
Then: Refund tự động được khởi tạo
And: Order vẫn ở trạng thái awaiting_payment
And: Log incident kèm tracking_id
```

Bốn tầng nghe có vẻ nhiều, nhưng thật ra chỉ xoay quanh bốn câu hỏi:

- Vì sao làm? BR.
- Ai làm gì? Use Case.
- Làm với những khái niệm nào? Entity Model.
- Biết đúng bằng cách nào? Acceptance Criteria.

Spec hiện tại và spec thay đổi

Một điểm mình thấy rất hay từ OpenSpec là họ tách rõ hai thứ:

- `specs/`: mô tả hệ thống hiện tại đang phải hành xử thế nào.
- `changes/`: mô tả một thay đổi đang được đề xuất, gồm lý do làm, phần spec thay đổi, hướng thiết kế và danh sách việc cần làm.

Điểm hay ở đây là feature mới không cần chen thẳng vào "bộ spec chính" ngay từ đầu. Khi có ý tưởng mới, bạn tạo một thư mục thay đổi riêng để ghi lý do làm, phần yêu cầu được thêm/sửa/xóa, hướng thiết kế và checklist triển khai. Chỉ sau khi làm xong và được review, thay đổi đó mới được lưu vào archive, còn phần spec mới được nhập lại vào spec chính.

Cách nghĩ này rất hợp với SDD vì nó giữ quy trình mềm. Spec chính vẫn là nơi mô tả hệ thống hiện tại, còn thay đổi mới có chỗ riêng để thảo luận mà chưa làm rối phần baseline. Với brownfield, cách này càng hữu ích, vì bạn có thể ghi rõ "đây là hành vi hiện tại" và "đây là đề xuất để đổi hành vi", thay vì trộn hai thứ vào một file rồi không ai biết đâu là sự thật đang chạy.

2.3. Một use case sống được cần những gì

Một use case đủ tốt không cần dài, nhưng vẫn cần có vài thành phần tối thiểu.

Đầu tiên là **ID**, ví dụ `UC-042`, vì nếu không có ID thì rất khó lần ngược từ spec sang code, test, commit, PR hay incident.

Tiếp theo là **Actor**. Actor giúp AI và dev biết ai đang thực hiện hành động này, bởi user thường, admin, scheduled job, webhook từ hệ thống ngoài hay customer support đều dẫn tới permission và validation khác nhau.

Trigger nói use case bắt đầu khi nào; một button click, một webhook, một cron job hay một event nội bộ sẽ dẫn tới thiết kế code khác nhau.

Main Flow là happy path, và đây là phần AI thường làm khá ổn nếu được viết rõ.

Alternative Flows là những biến thể hợp lệ, cũng là nơi nhiều bug xuất hiện vì business thường nghĩ "cái đó đương nhiên" còn dev và AI thì không.

Exceptions là các tình huống lỗi đã có quyết định xử lý. Đừng trộn exception với bug: "QR hết hạn thì hiển thị lỗi và cho tạo lại" là exception, còn "app crash khi QR hết hạn" là bug.

Cuối cùng là **Acceptance Criteria**. Không có AC, spec mới chỉ là mô tả; có AC, spec bắt đầu trở thành thứ có thể kiểm thử.

Một mẹo nhỏ là đừng cố nghĩ hết Alternative và Exception ngay từ vòng đầu. Hãy viết đủ những case rõ nhất để bắt đầu, rồi bổ sung tiếp khi review, test hoặc user phản hồi. Spec tốt không phải spec hoàn hảo từ ngày đầu, mà là spec chịu được việc học thêm.

2.4. Entity Model là ngôn ngữ chung

Thử đưa AI hai prompt sau cho cùng một việc.

Prompt A:

> Tạo service quản lý `tbl_dh_v2` với các cột id, ma_kh, tong_tien, tt, ngay_tao, ngay_cap_nhat.

Prompt B:

> Tạo service quản lý Order. Một Order có nhiều OrderItem, thuộc về một Customer, có một Payment sau khi thanh toán, và có trạng thái draft, awaiting_payment, paid, cancelled.

Prompt B thường cho code tốt hơn, không phải vì AI ghét tiếng Việt hay thích tiếng Anh, mà vì prompt B cho nó nhiều móc bám hơn.

`Order`, `OrderItem`, `Customer`, `Payment` là những khái niệm AI đã gặp rất nhiều; quan hệ "một Order có nhiều OrderItem" giúp nó hiểu cấu trúc, còn danh sách trạng thái giúp nó hiểu vòng đời của đơn hàng. Từ đó, nó có thể sinh guard clause, validation và test case hợp lý hơn.

Đó chính là Ubiquitous Language trong DDD: business và dev dùng cùng một ngôn ngữ. Thời AI, ngôn ngữ đó có thêm một người nghe nữa là model; nếu bạn dùng tên mơ hồ, AI sẽ đoán, còn nếu bạn dùng tên rõ, nó có nhiều cơ hội làm đúng hơn.

Đừng để một từ có quá nhiều nghĩa

Một cái bẫy rất hay gặp: cùng một từ `Order`, nhưng ở checkout nó nghĩa là đơn hàng, ở shipping nó nghĩa là đơn giao, ở finance nó có thể gần với khoản phải thu.

Nếu bạn nhét tất cả vào một class `Order`, bạn sẽ tạo ra một god object mà dev khó hiểu và AI càng dễ regenerate sai.

Bounded Context là cách DDD nói rằng: trong context này, một từ có một nghĩa cụ thể; sang context khác, nó có thể là khái niệm khác.

Trong repo spec, cách làm rất đơn giản:

```
/specs
  /checkout-context
    entities.md       (Order = đơn hàng)
    UC-042-PlaceOrderQR.md
  /shipping-context
    entities.md       (Order = đơn giao hàng)
    UC-101-CreateShipment.md
  /finance-context
    entities.md       (Order = receivable)
    UC-203-IssueInvoice.md
```

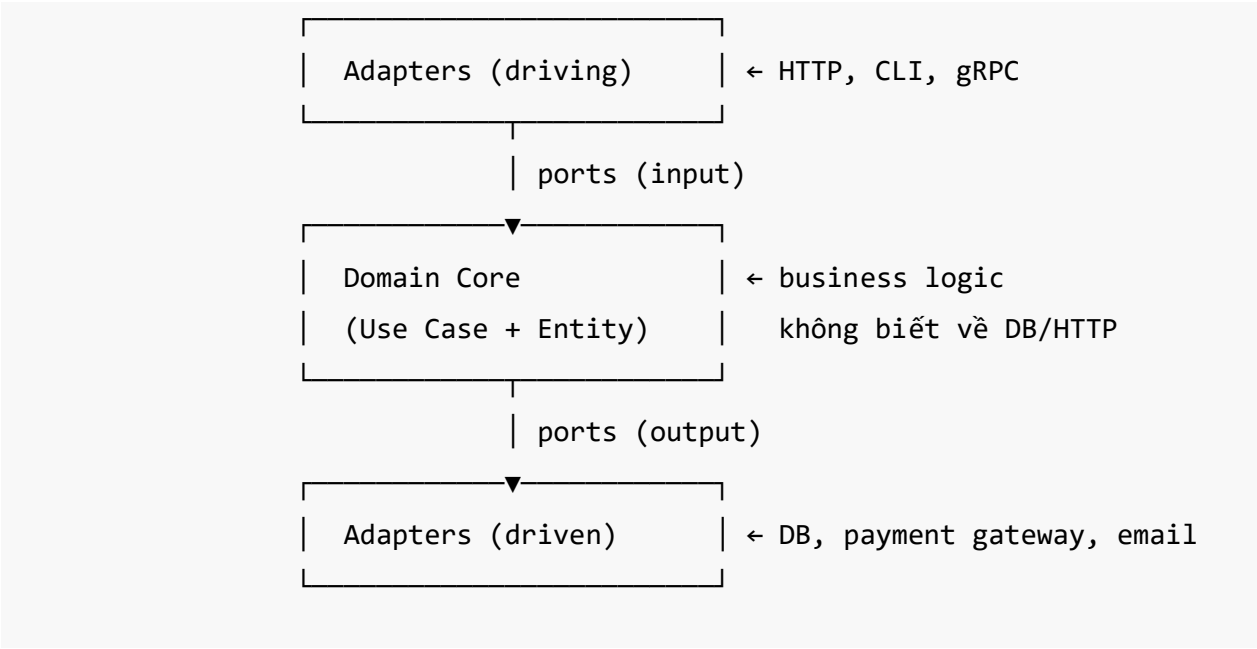
Khi AI đọc spec trong `/checkout-context`, nó không bị kéo sang nghĩa của `/shipping-context`. Đó là cách bạn ngăn một số lỗi kiến trúc từ rất sớm.

> Nếu bạn không quen DDD, chỉ cần nhớ một quy tắc: cùng một từ mà team dùng với hai nghĩa khác nhau thì sớm muộn cũng sinh lỗi. Khi nghe team tranh cãi "Order chỗ này khác Order chỗ kia", đó là lúc nên tách context.

2.5. Hexagonal Architecture, một kiến trúc cũ bỗng hợp thời AI

DDD giúp team đặt tên và chia nghĩa, còn Hexagonal Architecture giúp team đặt code vào đúng chỗ.

Ý tưởng cốt lõi của Hexagonal, hay Ports and Adapters, là tách lõi nghiệp vụ khỏi thế giới bên ngoài.



Domain Core chứa rule nghiệp vụ như Order, PlaceOrder, PaymentPolicy, hoặc rule "QR hết hạn sau 15 phút"; nó không cần biết Postgres, REST, Stripe hay Sendgrid.

Port là interface cho những nhu cầu của domain, chẳng hạn lưu Order, gửi email hoặc verify payment.

Adapter là phần hiện thực cụ thể của các port đó, ví dụ `PostgresOrderRepository`, `SendgridEmailAdapter`, `StripePaymentAdapter`.

Vì sao nó hợp với AI

Vì bạn có thể để AI viết lại adapter mà không động vào lõi nghiệp vụ.

Ví dụ, ba tháng sau launch, team muốn đổi từ Stripe sang Adyen. Nếu logic Stripe nằm rải rác trong service, AI phải sửa rất nhiều file và có thể vô tình đổi business rule. Nhưng nếu code theo hexagonal, bạn viết adapter mới, đổi config, domain gần như không đổi.

Spec vẫn nói "thanh toán bằng QR", acceptance test và domain test vẫn giữ nguyên, còn AI chỉ làm việc ở phần kết nối.

Đó là vùng AI rất giỏi, vì code có ranh giới rõ, input/output rõ và ít quyết định nghiệp vụ.

SDD và Hexagonal khớp nhau tự nhiên

Khi bạn viết spec theo SDD, bạn đang gián tiếp vẽ ra cấu trúc hexagonal. Mỗi Use Case trong spec map gần như một đối một với một use case trong domain core. Mỗi Entity trong spec là một aggregate root hoặc value object. Mỗi tích hợp ra ngoài, từ database tới payment gateway hay email service, là một port với adapter có thể đổi theo môi trường.

AI sinh ra code có cấu trúc như vậy thì cũng dễ test hơn, dễ bảo trì hơn, và dễ tái sinh hơn khi spec đổi. Đó là lý do mình nói hai cách tiếp cận này không phải tình cờ trùng nhau, mà bổ trợ nhau khá tự nhiên.

2.6. Khi nào chưa cần đến DDD và Hexagonal đầy đủ

Để cho công bằng, không phải dự án nào cũng đáng đầu tư đầy đủ Bounded Context, Aggregate Root, Port và Adapter. Một số trường hợp có thể nhẹ hơn nhiều.

Script một lần dùng thì spec đơn giản viết imperatively là đủ. CRUD admin panel thuần thì entity với flow đơn giản là được, không cần tách port. Prototype thí nghiệm thì vibe coding với AI nhanh hơn nhiều so với dựng hexagonal đầy đủ.

Quy tắc mình hay dùng là thế này: nếu hệ thống có rule nghiệp vụ không tầm thường, sẽ sống quá sáu tháng, hoặc có hơn hai dev cùng làm, hãy đầu tư DDD nhẹ kèm Hexagonal. Nếu không nằm trong nhóm đó, đừng thiết kế quá tay.

2.7. Một thí nghiệm nhỏ trước khi sang Chương 3

Trước khi đọc tiếp, bạn có thể thử thí nghiệm này với chính codebase team mình. Mở năm file ngẫu nhiên trong `src/`, sau đó đếm hai con số.

Con số thứ nhất: bao nhiêu file có class hoặc function chính mang tên giống tên một use case nghiệp vụ, ví dụ `PlaceOrder`, `IssueRefund`, hay `VerifyKYC`.

Con số thứ hai: bao nhiêu file mang tên kỹ thuật thuần, ví dụ `OrderService`, `OrderHandler`, hay `OrderManager`.

Tỷ lệ càng nghiêng về phía thứ hai, codebase của bạn càng xa ngôn ngữ nghiệp vụ, và càng khó để AI làm việc hiệu quả với spec. Chương 3 sẽ cho bạn sáu nguyên tắc để từng bước thay đổi điều đó, mỗi nguyên tắc đi kèm một tình huống cụ thể.

Chương 3. Sáu nguyên tắc Spec-Driven

Chương này lấy cảm hứng từ sáu nguyên tắc của AI Unified Process (AIUP), nhưng mình sẽ không bắt đầu bằng định nghĩa, vì định nghĩa thường đọc xong rất nhanh quên. Thứ dễ nhớ hơn là tình huống: một field không ai nhớ vì sao tồn tại, một rule refund bị AI đoán sai, hoặc một test suite xanh nhưng production vẫn lỗi.

Vì vậy, mỗi nguyên tắc trong chương này sẽ đi theo cùng một nhịp: chuyện xảy ra trong team, bài học rút ra, rồi cách áp dụng vừa đủ. Đọc xong chương này, bạn không cần thuộc tên sáu nguyên tắc; chỉ cần nhớ sáu câu hỏi ở cuối chương là đã đủ để mang về team.

3.1. Nguyên tắc 1: Requirements-Driven

> *Spec dẫn dắt code, không phải code dẫn dắt spec.*

Tình huống

Một team e-commerce sáu người đang sprint review. Dev Hùng demo feature mới:

> Em thêm trường `is_premium` vào Order. Khách VIP có thể đặt hàng trước.

PO Mai nghe thấy hợp lý nên gật đầu, và feature được merge.

Ba tháng sau, production có bug: một số khách thường vẫn nhận thông báo "đơn ưu tiên". Mai tìm lại Slack thì không thấy quyết định nào rõ, Hùng đã chuyển team, còn dev mới chỉ thấy trong database có cột `is_premium` nhưng không biết nó được set theo rule nào. Trong code có bốn chỗ set field đó, mỗi chỗ một điều kiện hơi khác.

Không ai nhớ vì sao field tồn tại; spec không có, comment không có, chỉ còn code.

Bài học

Nếu một khái niệm mới xuất hiện trong code, nó nên xuất hiện trong spec trước hoặc ít nhất cùng lúc.

Nghe có vẻ cứng, nhưng thực tế chỉ là một thói quen nhỏ: khi bạn hoặc AI định thêm field, status, flow, permission hay policy, hãy dừng lại hai phút và ghi vào spec.

Ví dụ:

```
## Entity: Order
- field `is_premium` (boolean): set TRUE khi customer.tier = "VIP"
  tại thời điểm tạo order. Không thay đổi sau đó. Phục vụ BR-012.
```

Hai phút này có thể tiết kiệm rất nhiều giờ sau đó, nhất là khi người viết code ban đầu không còn ở team.

Cách áp dụng

Đưa "Spec updated" vào Definition of Done, và trong code review, nếu PR thêm field, status hoặc flow mới mà không link tới spec thì gắn label `needs-spec`. Commit message cũng nên có UC hoặc BR ID, ví dụ:

```
feat(UC-042): add QR expiry validation
```

Quan trọng nhất: đừng biến nguyên tắc này thành thủ tục nặng. Với thay đổi nhỏ, một dòng spec là đủ. Mục tiêu là giữ ý định lại, không phải viết văn bản cho đẹp.

3.2. Nguyên tắc 2: AI-Assisted

> *AI làm phần lặp lại; con người giữ phần quyết định.*

Tình huống

Trang được giao build CRUD admin cho mười hai entity nội bộ. Nếu làm tay, cô ước lượng khoảng hai tuần, mà đây lại là kiểu việc vừa nhiều vừa chán: form, table, filter, validation cơ bản, quyền truy cập.

Trang đưa cho Claude Code file `/specs/entities.md`, trong đó có tên entity, field và quan hệ. Ba mươi phút sau, cô có controller, service và test suite cho mười hai entity; thêm một buổi review để sửa vài relationship AI hiểu sai, một việc hai tuần rút xuống còn khoảng một ngày.

Cùng tuần đó, ở team khác, dev Tú nhờ AI viết rule refund:

> Nếu khách hủy đơn trong 24h, refund 100%. Sau đó refund 50%.

AI viết xong và không ai kiểm lại kỹ. Ba tuần sau, kế toán phát hiện rule thật không phải vậy: sau 24h không refund, trừ khi đơn chưa bắt đầu shipping. Team mất tiền thật.

Bài học

AI rất hợp với việc có khuôn rõ: CRUD, mapping, boilerplate, rename, extract method, sinh test theo AC đã có sẵn, viết doc từ code và spec.

AI không nên tự quyết những thứ có hệ quả business lớn: chính sách refund, hạn mức tín dụng, quyền truy cập dữ liệu, trade-off kiến trúc lớn, rule liên quan pháp lý hoặc tiền bạc.

AI có thể đề xuất, nhưng con người phải quyết.

Cách áp dụng

Trong sprint planning, chia task thành hai nhóm:

- AI-driven: phần có pattern rõ, ít rủi ro nghiệp vụ.
- Human-driven, AI-assisted: phần cần con người quyết định, AI chỉ hỗ trợ.

Khi review code business-critical, hỏi một câu rất thẳng:

> Rule này đến từ spec nào?

Nếu không trả lời được, đừng merge.

3.3. Nguyên tắc 3: Iterative Improvement

> *Spec không cần hoàn hảo ngày đầu. Nó cần tốt lên sau mỗi vòng.*

Tình huống

Team của Linh viết spec cho UC-042: đặt hàng bằng QR. Vòng đầu chỉ có Main Flow và hai Exception rõ nhất, đủ để AI viết code, test pass và demo được.

Khi người dùng thử, một case mới lộ ra: khách quét QR, thanh toán xong, nhưng đúng lúc đó mạng trên điện thoại bị đứt. Khi app reconnect, đơn bị xử lý duplicate.

Spec vòng hai thêm AC về idempotency, AI regenerate handler và test mới pass. Đến vòng ba, kế toán phát hiện khi hủy đơn đã thanh toán cần ghi sổ ngược, nên spec lại thêm exception và flow refund. Nhìn như vậy không có nghĩa là spec vòng đầu thất bại; team chỉ đang học domain qua thực tế.

Bài học

Spec là tài liệu sống, nên đừng cố viết mọi thứ từ ngày đầu. Bạn sẽ dễ viết quá kỹ những phần chưa chắc quan trọng, trong khi vẫn bỏ sót những thứ chỉ lộ ra khi user dùng thật. Điều nguy hiểm không phải là spec phải sửa, mà là code thay đổi liên tục còn spec thì đứng yên.

OpenSpec diễn đạt tinh thần này khá gọn: quy trình nên linh hoạt, không cứng; lặp lại để học, không phải waterfall. Mình thích cách nghĩ đó vì nó nhắc team rằng `/plan`, `/tasks`, `/implement` hay `/opsx:propose`, `/opsx:apply` chỉ là hành động hỗ trợ, không phải cổng kiểm soát để mắc kẹt. Nếu đang triển khai mà phát hiện spec sai, quay lại sửa artifact trước đó là chuyện bình thường.

Cách áp dụng

Hãy time-box buổi viết spec đầu tiên; với một use case vừa phải, một đến hai giờ là đủ cho vòng đầu.

Mỗi spec nên có `## History`:

```
## History
- v1: initial
- v2: added AC-4 for idempotency after UAT
- v3: clarified refund flow for paid orders
```

Khi spec đổi, cập nhật test theo AC mới. Nếu AC cũ không còn đúng, đánh dấu deprecated thay vì xóa sạch, vì phần lịch sử đó giúp người mới hiểu vì sao rule hiện tại tồn tại.

> Nếu bạn là EM, đừng chỉ hỏi “spec đã đầy đủ chưa?”. Hãy hỏi “sau mỗi sprint, spec có học được gì mới không?”.

3.4. Nguyên tắc 4: Test-Protected

Test là hàng rào để AI được phép viết lại mà không làm lệch hành vi.

Tình huống

Sau sáu tháng, team Nam có hơn hai trăm test. Claude đề xuất refactor service layer, tách logic thanh toán thành strategy pattern; Nam chạy test thì 196 pass, 4 fail. Đọc kỹ, anh thấy hai test cũ cần sửa vì mock quá sát implementation, còn hai test còn lại bắt đúng regression, nên team sửa rồi merge khá tự tin.

Một team khác cũng để AI hỗ trợ refactor, nhưng test suite chỉ có mười hai smoke test kiểu "API trả 200". Test pass hết, vậy mà tuần sau production vẫn có incident vì rule không refund cho một nhóm khách đã biến mất. Test không bắt được, đơn giản vì nó chưa bao giờ kiểm tra rule đó.

Khác biệt ở đây không nằm ở số lượng test, mà nằm ở việc test có bám vào Acceptance Criteria hay không.

Bài học

Test trong SDD nên kiểm tra hành vi từ spec, không chỉ kiểm tra chi tiết triển khai.

Một cấu trúc dễ hiểu:

```
tests/
  /use-cases
    UC-042-place-order-qr/
      AC-1-create-qr.test.ts
      AC-2-payment-success.test.ts
      AC-3-qr-expired.test.ts
      AC-4-idempotency.test.ts
```

Khi spec có AC-4 thì test cũng có AC-4, và khi test fail, bạn biết mình phải mở spec nào.

Cách áp dụng

Mỗi AC nên có ít nhất một test, và tên test hoặc annotation nên chứa UC ID cùng AC ID. Với Java có thể dùng annotation kiểu `@UseCase("UC-042")`; với TypeScript hoặc Python, tên describe/test cũng đủ dùng:

```
describe("UC-042 / AC-2: payment success", ...)
```

Và một nguyên tắc rất đáng giữ: test đầu tiên cho một pattern nên do người viết hoặc review kỹ. Sau đó AI có thể clone pattern. Đừng để AI tự viết code rồi tự viết test cho chính code đó mà không có điểm tựa từ spec.

3.5. Nguyên tắc 5: Stakeholder-Centric

> *Spec chỉ thật sự có giá trị khi người chịu hậu quả đã đọc nó.*

Tình huống

Khoa dành ba ngày implement feature tích điểm thành viên; demo đẹp, test xanh, UI cũng ổn.

PO Vy xem xong hỏi:

> Khi khách hủy đơn, điểm đã tích có bị trừ lại không?

Khoa khựng lại:

> Spec không nói chỗ đó.

Vy đáp:

> Vì với business thì chuyện đó đương nhiên.

Điều đó đương nhiên với Vy, vì cô đã làm với nghiệp vụ này ba năm, nhưng lại không đương nhiên với Khoa và càng không đương nhiên với AI.

Một buổi review spec ba mươi phút trước khi code có thể đã bắt được case này.

Bài học

Spec không nên chỉ là tài liệu giữa dev và AI; nó phải được đọc bởi người chịu hậu quả khi sản phẩm sai.

Với feature nghiệp vụ, thường cần hai góc nhìn:

- Một người kỹ thuật: spec có implement được không, có lệch kiến trúc không?
- Một người nghiệp vụ: spec có đúng thực tế không, có thiếu rule "đương nhiên" không?

Cách áp dụng

Spec PR nên có ít nhất một approval từ phía nghiệp vụ và một approval từ phía kỹ thuật. Ngoài ra, mỗi sprint nên có một buổi walkthrough ngắn: PO mở spec mới, đọc qua flow và AC, rồi team đặt câu hỏi. Không cần nghi lễ; mười lăm đến ba mươi phút thường đã đủ bắt được rất nhiều gap.

Với những spec quan trọng, customer support và operations cũng nên được kéo vào sớm, vì họ thường biết những edge case mà dev và PO không nhớ.

> Nếu bạn là PO, bạn không cần trở thành dev để tham gia SDD. Việc quan trọng nhất của bạn là đọc spec và hỏi: "Nếu A thì sao? B có được tính không? Mình không thấy xử lý C." Những câu đó rất giá trị.

3.6. Nguyên tắc 6: Traceable

> *Từ code tìm được spec. Từ spec tìm được test. Từ test tìm được requirement.*

Tình huống

Ba giờ sáng, production có incident: khách đặt hàng nhưng không nhận email xác nhận, trong khi on-call dev là Quân mới vào team hai tháng.

Nếu không có traceability, Quân sẽ tìm "email confirmation" trong Slack và gặp vài chục thread; grep `send_email` trong code thì thấy nhiều chỗ, nhưng đọc một hồi vẫn không biết đây là bug hay hành vi đúng theo một quyết định cũ. Senior đang ngủ, nên Quân chỉ còn cách patch tạm và hy vọng đúng.

Nếu có traceability, alert hoặc log chỉ tới `UC-042 / AC-2`. Quân mở spec:

> Email xác nhận được gửi trong vòng 10 giây sau khi payment success.

Anh mở test AC-2 thì thấy test pass, rồi check log và phát hiện payment webhook thành công nhưng email service crash. Vậy vấn đề không phải AC-2 sai, mà là spec thiếu AC về retry email khi email service fail. Quân có thể mở một ticket rất rõ:

> Cần bổ sung AC-5: retry email confirmation khi email provider fail.

Incident chưa biến mất, nhưng thời gian "không biết phải hỏi gì" giảm rất mạnh.

Bài học

Traceability cần đi được ba chiều:

```
Business Requirement (BR) ↔ Use Case (UC) ↔ Code/Test
```

BR cần biết UC nào phục vụ nó, UC cần biết code và test nào implement nó, còn test cần biết AC nào nó đang bảo vệ.

Cách áp dụng

Commit message nên có format:

```
<type>(UC-XXX): <description>
```

Folder code và test nên soi chiếu với spec:

```
/specs/use-cases/checkout/UC-042-place-order-qr.md
/src/use-cases/checkout/place-order-qr/
/tests/use-cases/checkout/place-order-qr/
```

Nếu dùng IDE hoặc plugin hỗ trợ click qua lại giữa spec và test thì tốt; nếu chưa có, chỉ riêng convention tên file và commit message cũng đã giúp rất nhiều.

3.7. Sáu câu hỏi để mang về team

Đừng cố thuộc tên nguyên tắc; hãy mang sáu câu này về hỏi team:

1. Requirements-Driven: *PR này có spec entry tương ứng không?*
2. AI-Assisted: *Đoạn nào AI được làm, đoạn nào con người phải quyết?*
3. Iterative Improvement: *Spec này đã học thêm gì sau vòng feedback gần nhất?*
4. Test-Protected: *Mỗi AC có test bảo vệ chưa?*
5. Stakeholder-Centric: *Ai từ phía nghiệp vụ đã đọc spec này?*
6. Traceable: *Nếu production lỗi đoạn này, có tìm được spec liên quan trong một phút không?*

Team trả lời được "có" cho cả sáu là team đã trưởng thành với SDD. Team mới bắt đầu thường chỉ đạt hai trên sáu, và đó cũng bình thường. Chương 6 sẽ nói về lộ trình áp dụng để dần dần đạt được tất cả trong chín mươi ngày.

Trước đó, Chương 4 và Chương 5 sẽ cho bạn thấy SDD chạy ngoài đời ra sao, một chương cho dự án mới và một chương cho legacy.

Chương 4. Greenfield, từ con số không lên một với SDD

4.1. Năm ngày, ba người, một internal HR tool

Công ty có tám mươi người, còn HR là chị Hà, người vẫn quản lý chấm công và nghỉ phép bằng Excel cộng với Google Form. Cuối mỗi tháng, chị mất gần hai ngày để tổng hợp, đối chiếu, hỏi lại manager và sửa lỗi nhập sai.

CEO nói với team Engineering:

> Làm giúp HR một internal tool đi. Không cần hoành tráng, miễn tháng sau dùng được.

Team được giao gồm ba người: Tuấn là Tech Lead, Diệp là Fullstack Dev, An là Junior Dev. Deadline chỉ có một tuần, nên stack được chọn rất nhanh: Next.js, Postgres và deploy trên Vercel, vì team đã quen sẵn.

Họ quyết định dùng SDD không phải vì dự án này quá lớn, mà vì nó đủ nhỏ để thử một cách nghiêm túc từ đầu đến cuối.

Phần dưới đây là nhật ký năm ngày. Mình kể khá cụ thể để bạn thấy SDD trong một dự án greenfield ngoài đời trông ra sao, không chỉ là khẩu hiệu "viết spec rồi cho AI code".

4.2. Ngày 1: Inception và Business Requirement Catalog

Buổi sáng ngày đầu, Tuấn, Diệp và An ngồi với chị Hà khoảng chín mươi phút. Họ không bàn database, không bàn UI framework, cũng không bàn role table, mà chỉ hỏi:

- Mỗi ngày chị đang làm gì với chấm công?
- Cuối tháng phần nào tốn thời gian nhất?
- Nếu tool chỉ làm được ba việc trong tuần đầu, chị muốn ba việc gì?
- Những thứ nào chắc chắn chưa cần làm?

Kết quả là một file BR ngắn:

```
# BR-001: Internal HR Tool — Chấm công & Nghỉ phép

## Goal
Tự động hóa quy trình chấm công và xét nghỉ phép, giảm thời gian
xử lý cuối tháng của HR từ 16 giờ xuống dưới 2 giờ.

## Success Metrics
- HR_processing_time_per_month < 2h (đo qua self-report HR)
- Employee_self_service_rate > 80% (xin nghỉ không cần email HR)
- Approval_turnaround < 24h (median)

## In Scope (MVP - Tuần 1)
- Chấm công vào/ra (web, không cần app)
- Xin nghỉ phép (annual, sick, unpaid)
- Approval flow 1 cấp (manager trực tiếp)
- Báo cáo tháng cho HR (export CSV)

## Out of Scope
- Mobile app
- Tích hợp lương
- Đa cấp duyệt
- Chấm công bằng GPS/camera
```

Điểm đáng chú ý không phải file này dài hay đẹp, mà là cả team thống nhất được "tuần này không làm gì". Với dự án greenfield, phần Out of Scope thường là thứ cứu team khỏi việc phình scope ngay ngày đầu.

Buổi chiều, Tuấn mở Claude Code trong repo mới và chạy:

```
/specify
```

Đây là lệnh của GitHub Spec Kit. Tuấn paste BR-001 vào, Spec Kit hỏi thêm vài câu về phạm vi và ràng buộc, rồi đề xuất sáu use case:

```
UC-001: Đăng nhập bằng tài khoản công ty (Google SSO)
UC-002: Chấm công vào ca
UC-003: Chấm công ra ca
UC-004: Xin nghỉ phép
UC-005: Duyệt/từ chối nghỉ phép (manager)
UC-006: Báo cáo tháng (HR)
```

Tuấn review và gộp UC-002 với UC-003 thành một use case "Chấm công vào/ra", còn lại năm use case. Hết ngày đầu, chưa có nhiều code, nhưng team đã có một thứ quan trọng hơn: phạm vi.

Commit đầu tiên:

```
git commit -m "feat(BR-001): initial requirement catalog, 5 use cases"
```

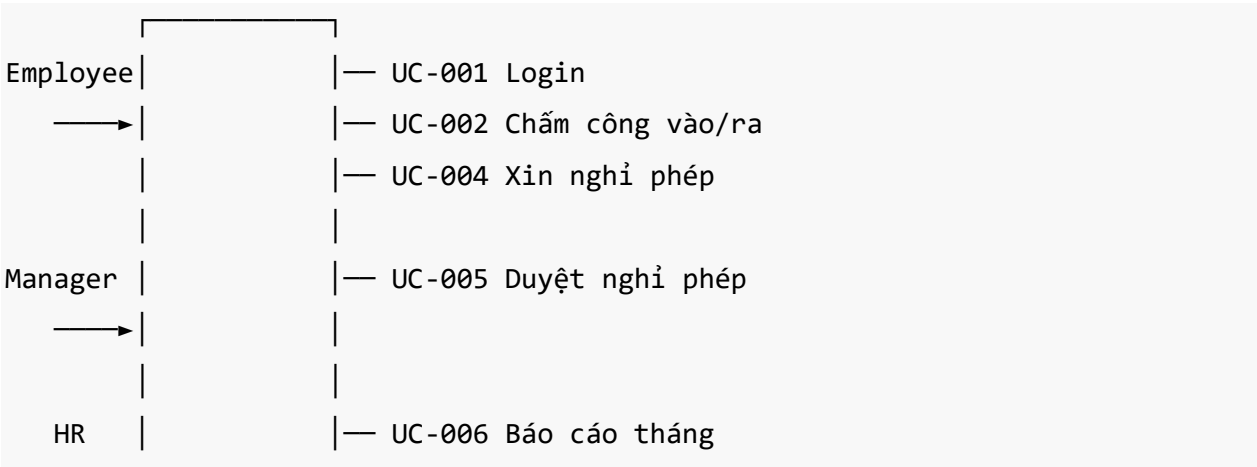
4.3. Ngày 2: Elaboration với Use Case Diagram và Entity Model

Sáng ngày hai, Diệp dùng plugin AIUP trong Claude Code:

```
/use-case-diagram
```

AI sinh PlantUML diagram với ba actor: Employee, Manager, HR. Diệp chỉnh lại tên use case, bỏ vài quan hệ `include` không cần thiết, rồi lưu diagram vào `/specs/diagrams/use-cases.puml`.

Diagram rút gọn:



```
   ──────►|            |
          └────────────┘
```

Sau đó là entity model:

```
/entity-model
```

AI đọc BR và use case list rồi đề xuất các entity. Diệp và An review lại với chị Hà, sau đó chốt bản cuối:

```
# Entity Model — HR Tool

## Employee
- id, full_name, email, department, manager_id
- annual_leave_balance (mặc định 12 ngày/năm)
- start_date
- Có nhiều: Attendance, LeaveRequest

## Attendance
- id, employee_id, check_in_at, check_out_at, date
- source ("web" | "manual_by_hr")
- Một Employee có nhiều Attendance (1 record/ngày)

## LeaveRequest
- id, employee_id, type ("annual" | "sick" | "unpaid")
- from_date, to_date, reason
- status ("pending" | "approved" | "rejected" | "cancelled")
- approver_id, approved_at, reject_reason
- Số ngày được tính: business day, trừ weekend, không trừ holiday MVP

## ApprovalLog
- Lưu mọi thay đổi status của LeaveRequest
- Phục vụ traceability cho HR
```

```
## MonthlyReport (view, không phải table)
- Tổng hợp: employee, work_days, leave_days, late_minutes
```

Tuấn phát hiện một chỗ thiếu: ngày lễ quốc gia chưa được xử lý. Chị Hà nói MVP chưa cần vì tháng đầu chỉ cần bỏ weekend, nên team ghi rõ "không trừ holiday MVP" vào entity model và Out of Scope.

Đến cuối ngày hai, repo có cấu trúc:

```
/specs
  /business-requirements
    BR-001-hr-tool.md
  /use-cases
    UC-001-login.md
    UC-002-attendance-check.md
    UC-004-request-leave.md
    UC-005-approve-leave.md
    UC-006-monthly-report.md
  /entities
    entity-model.md
  /diagrams
    use-cases.puml
```

Các file UC lúc này chỉ là stub, nghĩa là mới có ID, tên và mô tả ngắn; chưa cần đầy đủ ngay.

4.4. Ngày 3 và 4: Viết spec chi tiết rồi để AI triển khai

An nhận UC-004 "Xin nghỉ phép" làm use case dẫn dắt, vì đây là use case có nhiều rule nhất. Nếu làm tốt, nó sẽ tạo mẫu cho các case còn lại.

An chạy:

```
/use-case-spec UC-004
```

AI mở rộng stub thành bản spec đầy đủ. An review, sửa vài chỗ, gọi Tuấn hỏi thêm những điểm chưa chắc, rồi chốt:

```
# UC-004: Xin nghỉ phép

## Actor
Employee (đã login)

## Trigger
Employee bấm "Xin nghỉ phép" ở trang Dashboard

## Preconditions
- Employee đã login
- Employee có annual_leave_balance > 0 (cho type "annual")

## Main Flow
1. Hệ thống hiển thị form: type, from_date, to_date, reason
2. Employee điền và submit
3. Hệ thống validate (xem Acceptance Criteria)
4. Hệ thống tạo LeaveRequest với status = "pending"
5. Hệ thống gửi email tới Manager kèm link duyệt
6. Hệ thống quay về Dashboard với toast "Đã gửi yêu cầu"

## Alternative Flows
- 4a. Type = "sick" + có file đính kèm: lưu file vào storage,
  link trong record.

## Exceptions
- E1. from_date < hôm nay: reject, hiển thị "Không thể xin nghỉ
  ngày quá khứ" (trừ HR đại diện đăng ký hộ – phase 2).
- E2. to_date < from_date: reject, hiển thị "Ngày kết thúc phải
  sau ngày bắt đầu".
- E3. Số ngày xin > balance (cho type "annual"): reject, hiển thị
  "Số ngày phép còn lại: X".
- E4. Có LeaveRequest pending khác overlap thời gian: reject.
```

```
## Acceptance Criteria

### AC-1: Submit yêu cầu hợp lệ
Given: Employee có 8 ngày phép, không pending request
When: submit annual leave từ 2025-03-10 đến 2025-03-12 (3 business days)
Then: tạo LeaveRequest pending
And: gửi email tới manager_id của employee
And: balance KHÔNG bị trừ (chỉ trừ khi approved)

### AC-2: Validate from_date quá khứ
Given: hôm nay là 2025-03-15
When: submit from_date = 2025-03-10
Then: reject với message "Không thể xin nghỉ ngày quá khứ"
And: KHÔNG tạo record

### AC-3: Validate vượt balance
Given: Employee có 2 ngày phép còn lại
When: submit annual leave 5 ngày
Then: reject với message "Số ngày phép còn lại: 2"

### AC-4: Validate overlap
Given: Employee có pending request 2025-03-10 → 12
When: submit request 2025-03-11 → 13
Then: reject với message "Đã có yêu cầu nghỉ chồng lấn thời gian"

### AC-5: Sick leave có thể vượt balance
Given: Employee có 0 ngày phép
When: submit SICK leave 3 ngày
Then: tạo LeaveRequest (sick không trừ vào annual balance)

## History
- v1 (Ngày 3): initial
```

Bây giờ mới để AI đi sâu vào code:

```
/plan UC-004
```

AI đề xuất plan: route Next.js, API handler, DB migration cho `leave_requests`, email service stub, test theo AC.

An đọc plan trước khi cho chạy tiếp và thấy một điểm AI đề xuất sai: trừ `annual_leave_balance` ngay khi submit, trong khi spec nói rõ chỉ trừ khi approved. Cô sửa plan rồi mới đi tiếp.

Tiếp theo:

```
/tasks UC-004
```

AI tách thành tám task:

```
1. Create migration: leave_requests, approval_logs
2. Create entity types (TypeScript)
3. Implement API POST /api/leaves
4. Implement validation layer
5. Implement email notification (stub for MVP)
6. Implement UI form
7. Write test suite (5 AC)
8. Update Dashboard to show user's requests
```

Rồi mới:

```
/implement
```

Claude Code đi qua từng task, còn An review từng commit nhỏ. Khi AI hỏi một quyết định nghiệp vụ mà spec chưa nói, An không để nó đoán; cô mở spec, bổ sung rule còn thiếu, rồi chạy lại.

Hết ngày ba, UC-004 và UC-001 đã xong, Diệp đang làm UC-005, còn Tuấn setup CI/CD.

Ngày bốn, team xong UC-002, UC-005 và UC-006. Test suite có hai mươi tám test case, trong đó hai mươi sáu pass và hai fail.

Fail đầu tiên đến từ overlap validation: spec chưa nói rõ khi `end_date` của request mới bằng `start_date` của request cũ thì có tính là overlap không. Team quyết định vẫn tính là overlap, rồi update AC-4, sửa test và sửa code.

Fail thứ hai là approve request không gửi email khi `manager_id` null. Spec chưa nói manager null thì sao, nên chị Hà giải thích rằng nếu không có manager thì HR director duyệt; team thêm rule đó vào UC-005.

Hai lỗi này không phải dấu hiệu quy trình hỏng. Ngược lại, đây là lúc SDD làm đúng việc của nó: ép hiểu biết mới quay lại spec trước khi code trôi tiếp.

4.5. Ngày 5: Deploy thử và vòng phản hồi

Sáng ngày năm, team deploy staging. Chị Hà và năm nhân viên dùng thử.

Những phản hồi đầu tiên:

Feedback	Cách xử lý
“Mình xin nghỉ trùng holiday Tết, hệ thống vẫn tính như business day”	Đã Out of Scope MVP, ghi vào BR-001 v2
“Email duyệt link không mở được trên mobile”	Bug, fix trong 30 phút
“Báo cáo CSV nên có cột phòng ban”	UC-006 chưa có, bổ sung AC rồi regen export
“Nút Submit ở form xin nghỉ nhỏ quá”	UX fix

Chiều ngày năm, deploy production.

Tổng kết lại, ba dev làm trong năm ngày, tương đương khoảng mười lăm ngày công. Codebase có bốn mươi bảy file và khoảng ba nghìn dòng code; test có ba mươi hai case, mỗi AC trong MVP có ít nhất một test; còn specs gồm một BR, năm use case, một entity model và một diagram.

Quan trọng hơn: khi chị Hà hỏi "sau này thêm nghỉ lễ thì sửa ở đâu?", team trả lời được ngay: BR-001 v2, entity LeaveRequest, UC-004 và UC-006.

4.6. Tool trong câu chuyện này

Bạn không bắt buộc dùng đúng các tool dưới đây, nhưng nên hiểu mỗi tool hợp với đoạn nào trong nhịp làm việc.

Tool	Mạnh ở	Khi nào chọn
GitHub Spec Kit	Luồng Spec → Plan → Tasks → Implement. Hỗ trợ nhiều AI agent. Ít bị khóa vào một tool.	Khi team mới làm SDD và cần khung rõ.
OpenSpec	Quy trình dựa trên artifact: proposal, delta specs, design, tasks; có `specs/` cho baseline và `changes/` cho thay đổi đang làm.	Khi muốn quy trình nhẹ, linh hoạt, đặc biệt khi cần quản lý nhiều thay đổi song song hoặc audit lịch sử.
Claude Code cộng AIUP marketplace	Slash command theo từng giai đoạn: `/requirements`, `/entity-model`, `/use-case-spec`, `/reverse-engineer`.	Khi team đã dùng Claude Code và muốn đi sâu hơn.
IntelliJ AIUP Navigator	Link spec ↔ test qua annotation như `@UseCase`.	Với stack JVM, nhất là Java/Kotlin.
Cursor / Copilot / Codex CLI	Làm việc tốt với Markdown spec nếu team đã quen IDE đó.	Khi không muốn đổi tool chính của dev.

Lời khuyên thực tế là đừng dành hai tuần chỉ để chọn tool "chuẩn". Hãy chọn một tool, làm một use case từ spec tới code tới test, rồi đánh giá. SDD tốt ở chỗ spec vẫn nằm trong repo, nên tool có thể đổi sau.

Nếu dùng OpenSpec cho case HR tool ở trên, nhịp có thể là:

```
openspec init
/opsx:propose request-leave-flow
/opsx:apply
/opsx:archive
```

Điểm khác biệt là thay đổi "Xin nghỉ phép" sẽ sống trong `openspec/changes/request-leave-flow/` trước. Khi hoàn tất và đã apply xong, phần spec mới được merge vào `openspec/specs/` qua bước archive. Cách này hữu ích khi team có nhiều thay đổi chạy song song và vẫn muốn baseline phản ánh đúng hệ thống đang chạy.

4.7. Mấy lỗi hay gặp khi mới làm SDD greenfield

Lỗi đầu tiên là cố viết spec quá chi tiết ngay từ ngày một. Triệu chứng là ba ngày chưa xong spec, dev chưa được code, còn PM thì mất kiên nhẫn. Cách chữa là time-box: vòng đầu cho một use case chỉ nên một đến hai giờ, đủ Main Flow và vài AC chính. Phần chi tiết hơn để các vòng sau.

Lỗi thứ hai là spec không có Acceptance Criteria. Spec có thể dài, đẹp, nhiều đoạn văn, nhưng nếu thiếu AC thì khi AI viết code xong, không ai biết "xong đúng" nghĩa là gì. Quy ước rất đơn giản: không AC thì không bắt đầu code.

Lỗi thứ ba là AI viết code trước, spec viết sau cho đủ tài liệu. PR có cả code và spec nhưng spec rõ ràng được viết sau code. Cách chữa là tách commit: spec trước, code sau. Reviewer nhìn timeline là biết.

Lỗi thứ tư là một mình dev viết spec rồi tự duyệt. Đây là chỗ team nhỏ hay vướng nhất. Cách chữa quay lại nguyên tắc Stakeholder-Centric ở Chương 3: mỗi spec PR cần ít nhất một reviewer từ phía nghiệp vụ.

4.8. Checklist trước khi bắt đầu greenfield với SDD

Trước khi bắt đầu một dự án greenfield mới với SDD, mình hay dùng checklist này. Bạn có thể copy về team mình:

- ☐Đã viết BR (một trang) với Goal, Success Metric, Out of Scope chưa?

- ☐Đã liệt kê năm đến mười use case chính (ID kèm một dòng mô tả) chưa?
- ☐Đã có Entity Model sơ bộ với tên đúng nghiệp vụ chưa?
- ☐Đã chọn tool SDD (Spec Kit, OpenSpec, Claude Code + AIUP, hay Cursor) chưa?
- ☐Đã quy ước commit message có UC ID chưa?
- ☐Đã quy ước test name map với AC chưa?
- ☐Đã có ít nhất một reviewer từ phía nghiệp vụ cho mỗi spec PR chưa?

Tích đủ bảy là bạn sẵn sàng cho Ngày 1.

Nhưng thực ra đa số chúng ta không làm greenfield thường xuyên. Chúng ta thừa kế hệ thống cũ. Chương 5 dành cho ngày đầu tiên bạn mở `git clone` và thấy năm năm legacy mà không có một dòng doc nào.

Chương 5. Brownfield, áp SDD vào legacy code

5.1. Năm năm legacy, không doc, dev gốc đã đi

Tâm là Tech Lead mới của team Billing ở một công ty telecom hai trăm người. Ngày đầu nhận việc, sếp đưa cho anh repo `billing-engine`: Spring Boot 1.5, khoảng ba trăm hai mươi nghìn dòng code, chạy từ năm 2020.

Tài liệu duy nhất là một README bốn dòng:

> *Billing engine. Run `./gradlew bootRun`. Configs in `application.properties`.*
> *Contact: thaonm (đã nghỉ).*

Sếp nói:

> Tháng sau cần thêm tính năng tính phí roaming quốc tế. Em xem rồi làm theo cách em quen.

Tâm mở file `BillingCalculator.java` và thấy một nghìn tám trăm dòng code. Bên trong có function `calculate()` dài tám trăm dòng, mười bảy nhánh `if/else`, cùng nhiều biến đặt tên kiểu `flag_v2`, `isNewPolicy`, `specialCase`.

Đây mới là đời thật của nhiều team, vì không phải lúc nào chúng ta cũng bắt đầu từ một repo sạch. Phần lớn công việc kỹ thuật là duy trì và mở rộng những hệ thống đã tồn tại.

SDD có giúp được không? Có, nhưng không theo kiểu viết một bản spec thật đẹp rồi cho AI rewrite toàn bộ.

5.2. Hai sai lầm điển hình với legacy cộng AI

Sai lầm đầu tiên là tin vào "big bang rewrite với AI".

Nhìn một file tám trăm dòng, ai cũng muốn vứt đi. Tâm có thể nghĩ rằng chỉ cần để Claude đọc code cũ, viết lại từ đầu, rồi mình review là xong. Vấn đề là không ai còn nắm trọn yêu cầu: dev gốc đã nghỉ, người phụ trách nghiệp vụ đổi hai lần, còn hệ thống thì vẫn đang tính

tiền cho khách hàng mỗi ngày. Nói cách khác, nó đang đúng, hoặc ít nhất là đủ đúng, theo nhiều rule chưa từng được ghi lại.

Big bang rewrite thường đi theo một quỹ đạo quen thuộc: vài tháng đầu rất phấn khích, sau đó team phát hiện hệ thống cũ có "tính năng ẩn" mà khách hàng đang dùng, timeline bắt đầu kéo dài, team nản dần và phía nghiệp vụ mất niềm tin.

Rewrite không sai; nguy hiểm nằm ở việc rewrite khi chưa có baseline.

Sai lầm thứ hai là "viết spec từ code rồi coi đó là sự thật".

AI có thể đọc code và sinh spec, và việc đó hữu ích, nhưng nó chỉ cho bạn biết code đang làm gì chứ không cho bạn biết code có đang làm đúng hay không. Nếu một bug đã sống trong code ba năm và khách hàng đã quen workaround, spec sinh từ code có thể vô tình biến bug thành feature.

Cả hai sai lầm đều bỏ qua một bước: kiểm chứng lại với phía nghiệp vụ.

Vì vậy, SDD trong brownfield phải bắt đầu bằng việc đào lại sự thật hiện tại, rồi hỏi người vận hành xem sự thật đó có thật sự đúng không.

Ở đây, cách nghĩ của OpenSpec rất đáng học: hãy tách "spec hiện tại" khỏi "thay đổi đang đề xuất". Với legacy, `specs/` nên mô tả hành vi đang chạy, kể cả khi hành vi đó xấu hoặc mang nợ lịch sử. Còn `changes/` mới là nơi nói "ta muốn sửa gì". Nếu trộn hai thứ, team rất dễ mất khả năng phân biệt đâu là bug, đâu là feature, đâu mới chỉ là đề xuất.

5.3. Chiến lược Brownfield SDD trong năm bước

Trong ba tháng tiếp theo, Tâm đi theo năm bước dưới đây. Ngoài đời, các bước có thể chồng lên nhau, nhưng tách ra sẽ dễ hình dung hơn.

Bước 1: Reverse-engineer Entity Model từ DB

DB schema thường là nơi rẻ nhất để bắt đầu. Nó không kể hết nghiệp vụ, nhưng cho bạn danh từ, quan hệ và một phần lịch sử của hệ thống.

Tâm chạy:

```
/reverse-engineer --target=entity-model --source=postgres
```

AI sinh bản entity model sơ bộ:

```
# Reverse-engineered Entity Model — Billing

## Subscription (bảng: tbl_sub_v3)
- Tên cũ "tbl_sub_v3" — đề xuất rename ngôn ngữ là "Subscription"
- Có nhiều: BillingCycle, UsageRecord

## BillingCycle (bảng: bill_cyc)
- Một tháng billing cho 1 subscription
- Có nhiều: Charge

## Charge (bảng: chrg_item)
- Một dòng tính phí (cuộc gọi, SMS, data, phụ phí, giảm giá)
- type: voice, sms, data, addon, discount, roaming, ...
- Có cột `flag_v2`, `flag_v3` chưa rõ nghĩa
```

Tâm không dừng ở đây mà ngồi với hai nhân viên vận hành billing để hỏi từng chỗ chưa rõ.

Họ giải thích rằng `flag_v3` là chính sách phụ phí mới áp dụng cho hợp đồng ký sau 2022-07, còn `flag_v2` vẫn tồn tại cho hợp đồng cũ.

Tâm ghi lại:

```
## Charge
- flag_v2 (deprecated, vẫn dùng cho legacy contract trước 2022-07)
- flag_v3 (default cho contract sau 2022-07, áp dụng chính sách phụ phí mới)
- TODO Phase 2: rename → contract_policy_version
```

Mất gần một tuần cho bước này, nhưng đổi lại team có file `entity-model.md` với khoảng bốn mươi entity đã được người vận hành xác nhận. Đó là nền móng chắc hơn rất nhiều so với việc lao vào refactor ngay.

Bước 2: Reverse-engineer Use Case từ code, log và UI

Code legacy thường không được tổ chức theo use case, mà theo controller, service, helper, util, hoặc đôi khi đơn giản là theo thói quen của người viết ban đầu.

Tâm dùng ba nguồn:

- Controller: endpoint nào thường là cửa vào của use case.
- Production log: hành động nào thật sự được dùng.
- UI flow: người vận hành bấm gì trong ngày làm việc.

Anh chạy:

```
/reverse-engineer --target=use-cases --source=src/controllers/
```

AI đề xuất danh sách use case, còn Tâm đọc lại, gộp những endpoint chỉ là chi tiết của cùng một flow và bỏ các endpoint internal ít liên quan.

Sau hai tuần, team có bốn mươi hai use case ở ba mức độ:

- 18 use case rõ: có endpoint, có log, có người vận hành xác nhận.
- 16 use case đoán được: có code, nhưng ý định nghiệp vụ chưa chắc.
- 8 use case bí ẩn: có code, nhưng không ai biết còn ai dùng không.

Tám use case bí ẩn được đánh dấu `legacy-unverified`. Team thêm analytics để xem còn request không và cam kết không đụng trong bốn tuần. Sau bốn tuần, năm cái không còn request được đề xuất deprecate, còn ba cái có khách lớn dùng thì phải đi tìm người hiểu nghiệp vụ để xác nhận.

Với brownfield, đào lại tri thức cũ là công việc thật, không phải phần phụ trước khi code.

Bước 3: Kiểm chứng baseline với phía nghiệp vụ

Nhiều team hay bỏ qua bước này.

Tâm tổ chức bốn workshop, mỗi buổi chín mươi phút:

- Subscription

- Billing Cycle
- Charges
- Reporting

Người tham gia không chỉ có dev, mà còn có trưởng vận hành billing, finance và customer service lead.

Cách làm của Tâm khá thủ công: mở spec stub do AI sinh từ code, đọc từng AC và hỏi:

> Cái này có đúng thực tế không?
>
> Có thiếu case nào không?
>
> Có rule nào đang "đương nhiên" với mọi người nhưng không nằm trong spec không?

Mỗi buổi phát hiện vài chỗ thiếu, trong đó có một chỗ rất đáng nhớ: khách hàng doanh nghiệp được giảm 10% cuối tháng nếu tổng usage vượt năm mươi triệu, áp dụng từ năm 2021. Code có làm, nhưng không ai trong team dev hiện tại biết hàm nào chịu trách nhiệm.

Tâm grep `0.9`, `discount`, rồi tìm ra `applyEnterpriseDiscount()` trong `LegacyBillingHelper.java`, được gọi từ đúng một nhánh trong `calculate()`, không test và không comment.

Từ đó team viết UC-018: "Áp dụng chiết khấu doanh nghiệp", có AC, có owner nghiệp vụ.

Tâm không sửa thẳng vào spec chính ngay, mà tạo một change folder kiểu OpenSpec:

```
openspec/changes/document-enterprise-discount/
├── proposal.md     # vì sao cần ghi lại rule này
├── specs/          # delta: thêm UC-018 và các scenario
├── design.md       # tác động tới BillingCalculator
└── tasks.md        # characterization test, AC test, refactor sau
```

Sau workshop và test xác nhận, change này mới được archive; lúc đó UC-018 trở thành một phần của baseline. Làm vậy chậm hơn việc sửa trực tiếp một file Markdown vài phút, nhưng lịch sử quyết định rõ hơn nhiều.

Bước 4: Bọc test quanh hành vi hiện tại

Với legacy, trước khi refactor, bạn cần biết hành vi hiện tại là gì.

Characterization test là test ghi lại hành vi hiện tại. Nó không nói rằng code đúng, mà chỉ nói rằng nếu refactor xong test vẫn pass thì hành vi cũ chưa bị đổi.

Tâm lấy năm mươi record billing thật từ production tháng trước, đã anonymized; mỗi record gồm input và output hiện tại. Anh đưa bộ dữ liệu đó cho AI để sinh test:

```
@Test
void characterization_case_001_subscription_v2_with_roaming() {
    // Input từ production trace #B-2024-09-14-001
    var input = loadFixture("case-001.json");
    var expected = loadFixture("case-001-expected.json");

    var actual = calculator.calculate(input);

    assertThat(actual).isEqualTo(expected);
}
```

Năm mươi test này giống như năm mươi ảnh chụp. Chúng không chứng minh billing đúng theo nghiệp vụ, nhưng tạo ra một lưới an toàn để refactor.

Với legacy, đôi khi chỉ cần biết "mình chưa làm đổi hành vi cũ" đã là một bước tiến rất lớn.

Bước 5: Refactor với spec và test bảo vệ

Bây giờ Tâm mới thật sự đụng vào code.

Quy tắc của anh:

- Refactor không đổi hành vi.
- Đổi hành vi phải có spec.
- Một PR không vừa refactor lớn vừa đổi rule nghiệp vụ.
- Characterization test phải pass.

- AC test mới phải pass.

Với function `calculate()` tám trăm dòng, Tâm và AI làm từng phần, bắt đầu bằng việc nhận diện bốn use case bên trong:

- UC-019: Tính phí cơ bản
- UC-020: Phụ phí cuối tuần
- UC-021: Giảm giá thành viên
- UC-022: Làm tròn hóa đơn

Mỗi use case có spec ngắn và test theo AC, sau đó AI hỗ trợ tách code thành bốn strategy class. Code cũ và code mới chạy song song qua feature flag trong hai tuần để so kết quả; khi có lệch, team xem spec và fixture để quyết định bên nào đúng.

Cuối cùng, `calculate()` không còn là một khối tám trăm dòng nữa, mà được tách thành nhiều file nhỏ hơn, mỗi file có spec và test đi kèm.

5.4. Vài nguyên tắc thực dụng cho Brownfield SDD

Đừng cố viết spec cho một trăm phần trăm codebase. Legacy thường có ba vùng:

- Module nóng: đổi mỗi sprint.
- Module hơi nóng (ấm): vài tháng mới đổi.
- Module lạnh: lâu rồi không ai đụng.

Hãy đầu tư theo mức độ: module nóng cần spec đầy đủ, characterization test và AC test; module ấm cần spec stub và test khi chuẩn bị đụng; còn module lạnh thì cứ để yên, trừ khi có lý do nghiệp vụ thật.

Cách tìm module nóng:

```
git log --since=6.months.ago --name-only | sort | uniq -c | sort -rn
```

File nào xuất hiện nhiều nhất trong sáu tháng gần đây là nơi nên bắt đầu.

Nguyên tắc thứ hai là bounded context trước, chi tiết sau. Đừng cố hiểu toàn bộ hệ thống trong tuần đầu; với billing, Tâm chia trước thành Subscription, BillingCycle, Charge, Invoice, Payment, Reporting, rồi context nào team đang đụng thì đào sâu context đó.

Nguyên tắc thứ ba là test trước refactor, vì code xấu nhìn rất ngứa tay nhưng refactor không test thì chẳng khác nào đánh bạc.

Nguyên tắc thứ tư là sống chung với phần chưa hoàn hảo. Bạn không cần biến toàn bộ legacy thành kiến trúc đẹp; chỉ cần mỗi lần đụng vào một module, bạn để lại nó rõ hơn một chút so với lúc nhận.

Nguyên tắc thứ năm là spec brownfield có quyền xấu. Nó có thể có note "TODO: kiểm chứng với phía nghiệp vụ", có comment lịch sử, có thuật ngữ cũ bên cạnh thuật ngữ mới; đừng đợi spec đẹp mới viết, vì brownfield cần spec thật trước khi cần spec đẹp.

5.5. Tool hữu ích cho brownfield

Một số tool hoặc kỹ thuật đặc biệt hữu ích:

Tool/kỹ thuật	Dùng cho việc gì
AIUP `/reverse-engineer`	Sinh spec stub từ code, entity model từ DB
AI sinh characterization test	Đưa input/output production, nhờ AI viết test
AST-based refactor tools như jscodeshift, rope, OpenRewrite	Refactor cơ học an toàn hơn
Feature flag như LaunchDarkly, Unleash	Cho code mới và cũ chạy song song
Git archaeology như `git log -L`, `git blame`	Tìm lịch sử "vì sao có dòng này"

AI cũng khá hữu ích khi bạn cho nó danh sách class và hỏi:

> Theo bạn các class này thuộc những bounded context nào?

Kết quả đó không nên được tin ngay, nhưng thường là điểm bắt đầu tốt cho cuộc thảo luận với team.

5.6. Khi nào dừng

Tech Lead brownfield sớm muộn cũng phải hỏi: bao giờ là đủ?

Không có công thức chung, nhưng có ba dấu hiệu đáng nhìn:

1. Dev mới có thể làm việc độc lập trong một module bất kỳ trong vòng một tuần.
2. Module nóng có tỷ lệ spec coverage trên bảy mươi phần trăm, mỗi use case quan trọng có AC test.
3. Khi phía nghiệp vụ hỏi "có làm được không?", Tech Lead trả lời được trong một giờ thay vì một tuần.

Đạt được cả ba nghĩa là legacy đã chuyển từ trạng thái "gánh nặng" sang "tài sản có thể duy trì". Đầu tư thêm sau điểm này thường chỉ là làm đẹp; nếu chi phí cao hơn lợi ích, hãy dừng.

5.7. Một lời cho người đang ngập trong legacy

Brownfield ít khi được kể trên blog post hay hội thảo, vì nó không có khoảnh khắc "ship MVP trong năm ngày" để chụp ảnh đăng LinkedIn. Nhưng đa số chúng ta sống ở đây, và phần lớn giá trị kỹ thuật cũng nằm ở đây.

Một câu mình hay tự nhắc khi làm brownfield: SDD trên legacy không phải để trả hết nợ, mà để dừng vay thêm. Mỗi spec mới bạn viết, mỗi test characterization bạn thêm, mỗi UC ID gắn vào commit message là một dòng nợ ngưng tăng. Sau sáu tháng nhìn lại, trận chiến không kết thúc, nhưng nó đã bắt đầu được kiểm soát.

Chương 6 sẽ nói về phần "team" của SDD: ai làm gì, repo xếp ra sao, code review thế nào, đo bằng chỉ số gì, và áp dụng cho team chưa biết SDD trong ba mươi, sáu mươi, chín mươi ngày.

Chương 6. Vận hành team Spec-Driven hằng ngày

6.1. Một cảnh sprint review điển hình

Thứ Sáu cuối sprint, team Tuấn họp review. Không khí khá bình thường, không có gì giống một "quy trình mới" đang được trình diễn.

An mở UC-008 và đọc qua các Acceptance Criteria. Chị Hà, người không code, góp ý hai chỗ; Diệp mở test report cho thấy mười bốn trên mười bốn AC đã pass. Trong sprint có hai AC mới được phát hiện từ feedback, và cả hai đã được merge vào spec v3.

Tuấn nhìn spec PR còn pending của An và hỏi:

> AC-3 chưa rõ nếu manager đang nghỉ phép thì ai duyệt thay?

An ghi lại thành open question.

Năm phút cuối, EM nhìn dashboard: Spec Coverage 87%, Regen Success Rate 92%, Trace Ratio 100%. Không ai nói nhiều, vì chỉ số đang ổn thì cứ để nó yên.

Cảnh này không có gì đặc biệt; nó chỉ là kết quả của vài quy ước nhỏ được giữ đều: spec có owner, AC có test, PR có UC ID, và phía nghiệp vụ đọc spec trước khi code đi quá xa.

Chương này nói về những quy ước đó.

6.2. Ai làm gì trong SDD

Một hiểu nhầm phổ biến là spec là việc của PO hoặc BA. Cách hiểu đó không đúng, vì trong SDD, spec là nơi cả team gặp nhau và mỗi vai trò đóng góp một phần khác nhau.

PO và BA: làm rõ "làm gì" và "vì sao"

PO hoặc BA chịu trách nhiệm chính cho BR: mục tiêu là gì, đo bằng gì, cái gì nằm ngoài scope. Họ cũng cùng viết Use Case spec, nhất là các phần Actor, Trigger, Main Flow, ngoại lệ nghiệp vụ, và review những spec PR có ảnh hưởng tới hành vi nghiệp vụ.

Phần khó nhất nằm ở những câu tưởng như "đương nhiên":

Hủy đơn thì điểm tích lũy có bị trừ không?

Manager nghỉ phép thì ai duyệt thay?

Khách doanh nghiệp có rule riêng không?

Những câu đó dev không thể đoán, và AI càng không thể đoán.

PO không cần biết markdown đẹp, vì template đã lo phần đó, cũng không cần biết code. Nhưng PO cần đọc spec và nói được: "Đúng, đây là rule nghiệp vụ."

Tech Lead: giữ spec và kiến trúc

Tech Lead review spec từ góc nhìn kỹ thuật:

Cái này triển khai được không?

Có lệch bounded context không?

Có đang ép domain biết quá nhiều về database hoặc provider không?

Tech Lead cũng quyết định đoạn nào nên để AI làm và đoạn nào con người phải giữ tay lái. CRUD, mapping, adapter có thể giao nhiều cho AI, trong khi rule nghiệp vụ, security boundary và kiến trúc lõi phải được review kỹ hơn.

Một việc nữa của Tech Lead là giữ ADR liên kết với spec. Nếu quyết định "dùng cron job thay vì pub-sub cho monthly billing", ADR nên link tới UC hoặc BR liên quan.

Dev: hiện thực hóa spec, viết test và dẫn AI

Dev không chỉ "nhận spec rồi code"; họ đọc spec, phát hiện gap, viết test map với AC, rồi dùng AI để triển khai nhanh hơn.

Vai trò của dev giống người dẫn việc cho AI: đưa context, giới hạn phạm vi, đọc lại phần AI sinh ra, chỉnh tên biến hoặc tên hàm, kiểm tra test, và không để AI tự quyết rule nghiệp vụ.

Một thói quen tốt là khi đang triển khai mà phát hiện spec thiếu, đừng âm thầm quyết trong code; hãy mở spec PR nhỏ, tag PO hoặc TL, rồi quay lại code sau.

Engineering Manager: giữ nhịp quy trình và chỉ số

EM không cần review từng dòng spec, vì việc chính của EM là đảm bảo quy trình không bị nghẽn.

Ai đang bị quá tải review spec? PO có phản hồi trong vòng 24 giờ không? Dev mới có hiểu cách viết UC không? Tool có làm team chậm lại không? Ban lãnh đạo có hiểu vì sao tuần đầu SDD chậm hơn vibe coding không?

EM cũng nên theo dõi vài chỉ số đơn giản: Spec Coverage, AC Coverage, Trace Ratio, onboarding time, bug rate ở module đã áp dụng SDD.

QA hoặc Tester: soi spec từ góc nhìn kiểm thử

Nếu team có QA, QA nên vào ngay từ giai đoạn viết spec, không đợi code xong mới test.

QA đọc AC và hỏi:

AC này test được không?

Given/When/Then có đủ rõ không?

Edge case nào người dùng thật sẽ chạm vào?

Trong SDD, QA giúp bắt bug khi nó còn là một câu mơ hồ trong spec. Đó là lúc sửa rẻ nhất.

Khi team nhỏ không có đủ vai trò

Startup nhỏ thường một người đội nhiều mũ: Tech Lead có thể viết cả spec nghiệp vụ, dev có thể kiêm QA, và chuyện đó vẫn ổn.

Nhưng đừng tự duyệt mọi thứ của mình. Nếu không có PO, hãy kéo founder, customer, người vận hành hoặc support vào đọc spec, vì bạn cần một người không viết code nhìn vào và hỏi những câu đời thường.

6.3. Cấu trúc repo gợi ý

Bạn không cần copy cấu trúc này y nguyên, nhưng nên giữ ý tưởng: spec, code và test soi chiếu được nhau.

```
project-root/
├── /specs                          ← nơi giữ source of truth nghiệp vụ
│   ├── /business-requirements
│   │   ├── BR-001-hr-tool.md
│   │   └── BR-002-...
│   ├── /use-cases
│   │   ├── /checkout-context
│   │   │   ├── UC-042-place-order-qr.md
│   │   │   └── UC-043-cancel-order.md
│   │   ├── /shipping-context
│   │   │   └── UC-101-create-shipment.md
│   │   └── _legacy
│   ├── /entities
│   │   ├── checkout-context.md
│   │   └── shipping-context.md
│   ├── /diagrams
│   │   ├── use-cases.puml
│   │   └── architecture.puml
│   └── README.md
│
├── /docs
│   ├── /adr
│   │   ├── 001-chose-postgres.md
│   │   └── 002-event-driven-payment.md
│   ├── runbooks/
│   └── onboarding.md
│
├── /src
│   ├── /use-cases
│   │   ├── /checkout
│   │   │   ├── place-order-qr/
│   │   │   └── cancel-order/
│   │   └── /shipping
```

```
│   ├── /domain
│   ├── /ports
│   └── /adapters
│
├── /tests
│   ├── /use-cases
│   │   └── /checkout
│   │       └── /place-order-qr
│   │           ├── AC-1.test.ts
│   │           ├── AC-2.test.ts
│   │           └── ...
│   └── /characterization
│
└── README.md
```

Khi PR đụng UC-042, reviewer biết cần xem ba chỗ:

- `/specs/.../UC-042.md`
- `/src/.../place-order-qr/`
- `/tests/.../place-order-qr/`

AI cũng lấy context dễ hơn khi folder được xếp theo cùng một nhịp.

Nếu team thích mô hình OpenSpec, repo có thể có thêm một lớp `openspec/`:

```
openspec/
├── specs/                ← baseline: hệ thống hiện tại phải hành xử thế nào
│   ├── auth/
│   │   └── spec.md
│   └── payments/
│       └── spec.md
├── changes/              ← mỗi thay đổi một folder riêng
│   ├── add-2fa/
│   │   ├── proposal.md
│   │   ├── specs/
```

```
│   │   ├── design.md
│   │   └── tasks.md
│   └── archive/
└── config.yaml
```

Mình không nghĩ mọi team đều cần đúng layout này, nhưng cách nghĩ phía sau rất đáng lấy: spec chính mô tả hiện tại, còn thay đổi mới sống trong change folder cho tới khi review, triển khai và archive xong. Nhờ vậy, team có thể làm song song nhiều thay đổi mà không làm baseline rối.

6.4. Quy trình code review trong thời SDD

Code review truyền thống chủ yếu nhìn code. Trong SDD, reviewer cần hỏi thêm một câu: code có khớp spec không?

Quy tắc 1: spec PR tách khỏi code PR

Spec PR chỉ chứa file `.md` trong `/specs`, với người review thường là PO/BA và Tech Lead. Code PR thì chứa code và test, với người review thường là dev. Việc tách ra giúp phía nghiệp vụ review spec mà không bị ngợp bởi diff code, đồng thời giúp dev review code sau khi ý định đã rõ.

Nếu dùng OpenSpec, tương đương với việc review `proposal.md`, delta spec, `design.md`, `tasks.md` trong `openspec/changes/<change-name>/` trước khi `/opsx:apply`. Sau khi code và test xong, `/opsx:archive` mới merge delta vào spec chính và giữ lại lịch sử change.

Quy tắc 2: code PR phải link UC

Tiêu đề PR nên có UC ID:

```
feat(UC-042): implement QR expiry validation
```

Phần mô tả link tới spec PR hoặc file spec. Test mới nên có UC ID trong tên describe/test.

PR đụng tới hành vi nghiệp vụ mà không link UC thì gắn `needs-spec`.

Quy tắc 3: review spec trước, code sau

Với PR lớn, mở spec trước. Đọc UC mất vài phút. Hỏi:

- Flow đã rõ chưa?
- AC có test được không?
- Exception có thiếu case hiển nhiên không?

Sau đó mới mở code. Nếu làm ngược lại, reviewer rất dễ sa vào style, naming, abstraction, rồi bỏ qua rule nghiệp vụ.

Quy tắc 4: test phải map AC

Spec có năm AC thì test nên có ít nhất năm test case tương ứng. Không cần máy móc tuyệt đối, nhưng nếu AC quan trọng mà không có test, phải có lý do rõ.

Quy tắc 5: hỏi "AI quyết hay bạn quyết?"

Khi thấy một block logic phức tạp, reviewer có thể hỏi:

> Đoạn này AI tự đề xuất hay bạn chủ động quyết?

Câu hỏi này không nhằm bắt lỗi AI, mà giúp team biết nguồn gốc quyết định để sau này debug.

6.5. Mấy lỗi thường gặp và cách chữa

Lỗi	Dấu hiệu	Cách chữa
Spec viết sau code	"Viết doc cho đẹp"	Chặn merge nếu code đụng hành vi mà spec chưa có
Spec quá chi tiết từ đầu	Spec session kéo dài, dev idle	Time-box 1-2h cho UC mới
AI bypass spec	Code có rule spec không nói	Code review trả lại, yêu cầu cập nhật spec
Test mock domain	Test pass nhưng prod fail	Test AC ở tầng hành vi, không chỉ mock HTTP

PO không review spec	Spec chỉ có dev đọc	Spec PR cần ít nhất một người nghiệp vụ
Spec không có history	Không biết rule thêm từ khi nào	Mỗi spec có `## History`
Tool lock-in	Spec phụ thuộc format của một tool	Giữ spec ở Markdown đọc được
Spec coverage ảo	Nhiều spec nhưng AC không có test	Đo AC Coverage, không chỉ đếm file spec

6.6. Chỉ số để biết SDD có đang có ích không

Đừng đo SDD bằng số dòng spec, vì dòng nhiều không có nghĩa là rõ. Mình thường dùng bốn chỉ số.

Spec Coverage: phần trăm use case đang hoạt động có spec đủ dùng, có link BR và có AC.

Mục tiêu gợi ý:

- Greenfield: gần 100% cho MVP.
- Brownfield module nóng: khoảng 70% sau một quý.
- Toàn bộ legacy: đừng ép 100%, sẽ tốn sai chỗ.

AC Coverage: phần trăm AC có ít nhất một test tương ứng.

Đây thường là chỉ số quan trọng hơn code coverage, vì nếu AC không có test, việc để AI regenerate code sẽ rủi ro hơn.

Regen Success Rate: phần trăm lần AI sinh lại code mà test suite pass, không cần sửa tay quá nhiều.

Nếu tỷ lệ này thấp, thường có hai khả năng: spec chưa đủ rõ hoặc test chưa đủ đúng.

Trace Ratio: phần trăm PR hoặc commit có UC ID hợp lệ.

Khi thiếu trace, mỗi lần production lỗi team sẽ lại grep code và đoán.

Có vài chỉ số nên tránh:

- Lines of spec: khuyến khích viết dài.
- Số UC đóng mỗi tuần: khuyến khích chia nhỏ giả tạo.
- Code coverage đơn thuần: vẫn hữu ích, nhưng không thay thế AC Coverage.

6.7. Lộ trình áp dụng 30/60/90 ngày

Đừng bắt cả team chuyển sang SDD trong một ngày; hãy thử ở phạm vi đủ nhỏ để học, nhưng đủ thật để có dữ liệu.

Ngày 1 đến 30: Thử trong phạm vi nhỏ

Chọn một use case nhỏ nhưng thật, cùng một dev có ảnh hưởng trong team và một PO/BA chịu ngồi cùng.

Setup tối thiểu:

- folder `/specs`
- template UC
- quy ước UC ID trong commit
- một tool AI team đã quen

Mục tiêu cuối tháng là có một module với spec, code và test map AC, đồng thời người tham gia thử có thể kể lại ba điều: cái gì hiệu quả, cái gì vướng, cái gì cần đổi.

Ngày 31 đến 60: Mở rộng

Chọn một module brownfield "ấm", không quá nguy hiểm nhưng có giá trị thật.

Reverse-engineer entity hoặc use case, viết spec baseline, thêm characterization test nếu cần, rồi bắt đầu đo Spec Coverage và AC Coverage, dù lúc đầu đo thủ công cũng được.

Mục tiêu: ít nhất 60% PR mới có UC ID, PO đã review vài spec thật, và team bắt đầu có ngôn ngữ chung.

Ngày 61 đến 90: Quyết định

Tổ chức retrospective riêng cho SDD. Hỏi thẳng:

- Cái gì giúp team?
- Cái gì chỉ tạo thêm việc?
- Tool nào cản trở?
- Bug rate, onboarding time, velocity có tín hiệu gì không?

Sau đó chọn một trong ba đường:

- Chuẩn hóa: dùng SDD làm mặc định cho code mới và module nóng.
- Áp dụng một phần: chỉ dùng cho khu vực như payment, billing, compliance.
- Dừng lại: chưa hợp team lúc này.

Dừng cũng được, miễn là quyết định dựa trên dữ liệu thay vì cảm giác mệt sau hai tuần đầu.

6.8. Một ngày bình thường trong team Spec-Driven

Chín giờ sáng, standup bắt đầu, và mỗi dev nói UC ID mình đang làm:

> Hôm nay mình tiếp UC-019.

Chín rưỡi, An phát hiện spec UC-019 thiếu trường hợp manager nghỉ phép, nên cô mở spec PR và tag PO Hà cùng TL Tuấn.

Mười giờ, Hà comment:

> Manager nghỉ phép thì cấp trên của manager duyệt.

An cập nhật AC.

Mười rưỡi, spec PR merge. An mở Claude Code, paste UC-019 đã update và chạy:

```
/implement
```

Mười một rưỡi, An đọc lại phần AI sinh ra: test có sáu method, map đúng sáu AC, chỉ có một biến đặt tên khó hiểu nên cô sửa lại.

Hai giờ chiều, Diệp review PR. Anh hỏi:

> Đoạn này AI quyết hay bạn quyết?

An giải thích đó là phần cô viết lại từ gợi ý của AI, và Diệp approve.

Bốn giờ, Tuấn làm một buổi viết spec ngắn với PO Hà cho UC-021. Sau bốn mươi lăm phút, team có spec stub v1 và bốn AC, đủ để dev bắt đầu ngày mai.

Không có gì kịch tính ở đây; spec chỉ trở thành một phần của nhịp làm việc.

6.9. Ba việc nên làm tuần sau

Nếu bạn là EM hoặc TL và muốn thử SDD, tuần sau chỉ cần làm ba việc.

Một là chọn một use case nhỏ, một dev và một PO để đi trọn một vòng SDD trong một sprint, thay vì chuyển cả team ngay.

Hai là tạo folder `/specs` và commit một template UC vào repo; hành động nhỏ này báo hiệu rằng team nghiêm túc.

Ba là đo một chỉ số duy nhất trong tuần đầu: bao nhiêu PR có UC ID.

Sau bốn tuần, bạn sẽ biết SDD có hợp team mình không. Phần Kết sẽ nói nốt một chuyện quan trọng: khi nào không nên dùng SDD.

Phần Kết. Khi nào không nên dùng SDD, và một góc nhìn về tương lai

7.1. SDD không phải thuốc chữa mọi bệnh

Sáu chương vừa rồi nói khá nhiều về lý do nên thử SDD, nhưng nếu dừng ở đó thì không công bằng.

SDD không phải lúc nào cũng đáng dùng; có những trường hợp nó chỉ làm team chậm hơn, thêm việc hơn, và tạo cảm giác "quy trình vì quy trình".

Hiểu rõ ranh giới vì vậy quan trọng không kém hiểu phương pháp.

Khi nào không nên dùng

Prototype dùng một lần dưới một tuần. Nếu bạn cần demo cho founder vào thứ Hai, code chạy một lần rồi bỏ, đừng dựng đầy đủ BR, UC, AC; vibe coding với AI có thể là lựa chọn đúng.

Spike kỹ thuật. Khi mục tiêu là học, ví dụ thử GraphQL Federation, vector database hay một framework mới, spec chi tiết thường chưa cần; sau khi spike chứng minh đáng làm tiếp, lúc đó viết spec cũng được.

Script một lần dùng. Với migration nhỏ, report ad-hoc hoặc batch sửa dữ liệu nội bộ, nếu chạy xong là bỏ thì spec đầy đủ thường không đáng.

Side project một người. Nếu chỉ có bạn làm, codebase nhỏ và gần như không có turnover, thì spec với chính mình có thể hơi gượng; vài README ngắn để nhắc lại ý định là đủ.

Tổ chức chưa có ai chịu trách nhiệm làm rõ nghiệp vụ. Nếu không có PO, BA, founder, customer, hoặc người dùng nội bộ đủ rõ để trả lời "đúng là gì", spec rất dễ biến thành tranh luận nội bộ. Trường hợp này nên sửa vấn đề ownership trước, SDD sau.

Khi nào nên dùng

SDD đáng cân nhắc khi có một hoặc nhiều dấu hiệu sau:

- Hệ thống sẽ sống hơn sáu tháng.

- Có hơn hai dev cùng làm codebase.
- Có rule nghiệp vụ không tầm thường: tiền, hợp đồng, pháp lý, quyền truy cập, quy trình vận hành.
- Team có khả năng đổi người.
- AI đang sinh một phần đáng kể code.
- Đây là phần lõi của sản phẩm hoặc nghiệp vụ, không phải tool dùng rồi bỏ.

Một quy tắc ngắn:

> Nếu chi phí bảo trì và onboard về sau lớn hơn chi phí viết spec từ đầu, hãy cân nhắc dùng SDD.

7.2. Vài hiểu nhầm cần gỡ

“SDD là waterfall mới.”

Không hẳn. Waterfall yêu cầu spec đầy đủ trước, code sau và ít quay lại; SDD chỉ yêu cầu spec vừa đủ để bắt đầu, rồi để spec lớn lên qua feedback. Nếu spec sai thì sửa spec, nếu học thêm rule mới thì thêm vào spec, vì nhịp của SDD vẫn là lặp lại và học tiếp.

“SDD chậm hơn.”

Tuần đầu thường chậm hơn thật, vì team phải viết BR, use case, AC và entity model. Nhưng tháng thứ ba mới là nơi SDD trả lại giá trị: onboard nhanh hơn, refactor tự tin hơn, AI regenerate an toàn hơn. Nếu dự án quá ngắn, bạn sẽ không kịp thu hồi khoản đầu tư đó.

“SDD chỉ dành cho enterprise.”

Không phải vậy. Startup ba người vẫn dùng được nếu phần họ làm có rule nghiệp vụ thật. Doanh nghiệp lớn có thêm lợi ích về governance và traceability, nhưng SDD không bắt buộc phải nặng.

“SDD bắt phải dùng DDD và Hexagonal.”

Không bắt buộc. DDD và Hexagonal chỉ rất hợp với SDD, còn bạn hoàn toàn có thể bắt đầu nhỏ hơn bằng cách đặt tên entity đúng, tách use case rõ và viết AC có test. Sau một thời gian, team thường tự thấy cần bounded context và port/adapter.

"AI càng giỏi thì càng không cần spec."

Mình nghĩ ngược lại: AI càng giỏi viết code, câu hỏi "viết đúng cái gì?" càng quan trọng. Spec là cách team trả lời câu hỏi đó một cách bền hơn prompt rời rạc.

7.3. Vai trò engineer sẽ đổi ra sao

Mình không có quả cầu pha lê, nhưng có vài hướng thay đổi khá rõ.

Dev sẽ không còn được đánh giá chủ yếu bằng tốc độ gõ code, vì AI gõ nhanh hơn bất kỳ ai. Giá trị của dev sẽ nằm nhiều hơn ở khả năng hiểu domain, viết spec đủ rõ, đọc lại phần AI sinh ra, và thiết kế kiến trúc sao cho AI hỗ trợ được mà không phá hệ thống.

PO và BA sẽ kỹ thuật hơn một chút. Họ không cần biết framework, nhưng cần viết hoặc review spec rõ hơn, hiểu Acceptance Criteria, edge case, và tác động của một rule bị thiếu.

Dev cũng sẽ gần phía nghiệp vụ hơn một chút, vì họ không thể chỉ nói "spec không ghi nên em không biết". Trong thế giới AI, dev phải biết hỏi câu đúng trước khi để AI viết quá xa.

Tech Lead sẽ giống "spec architect" hơn. Quyết định quan trọng không chỉ là chọn Postgres hay MongoDB, REST hay gRPC. Quyết định quan trọng hơn có thể là: bounded context chia thế nào, use case nhỏ đến đâu, spec repository tổ chức ra sao, traceability giữ ở mức nào.

QA sẽ đi sớm hơn vào vòng đời sản phẩm. Bug rẻ nhất là bug bắt được khi AC còn là một câu trong spec, còn bug đắt nhất là bug do khách hàng bắt trong production; SDD kéo QA về phía sớm hơn.

Những thay đổi này không làm dev hết việc. Nó chỉ kéo công việc ra xa phần gõ boilerplate và gần hơn với việc thiết kế ý định.

7.4. Tài liệu để đi tiếp

Phương pháp và khung làm việc

- AI Unified Process (AIUP) — phương pháp đầy đủ của Simon Martinelli, có tool và community.
- GitHub Spec Kit — toolkit open-source cho luồng Spec → Plan → Tasks → Implement.
- OpenSpec — framework nhẹ cho SDD với quy trình dựa trên artifact, delta specs và archive history.
- Microsoft Spec-Driven Development — series bài viết về Spec Kit.

Nền tảng cũ vẫn rất đáng đọc

- *Domain-Driven Design* của Eric Evans và Vaughn Vernon. Nếu mới bắt đầu, đọc Vernon trước sẽ dễ hơn.
- Hexagonal Architecture — bài gốc của Alistair Cockburn.
- *Clean Architecture* của Robert C. Martin — cùng tinh thần tách domain khỏi chi tiết bên ngoài.
- *Working Effectively with Legacy Code* của Michael Feathers — kinh điển cho brownfield. Khái niệm characterization test trong Chương 5 lấy cảm hứng từ đây.

Tool đáng thử

Tool	Dùng cho việc gì
GitHub Spec Kit	Luồng Spec → Plan → Tasks → Implement
OpenSpec	Proposal → delta specs → design → tasks → apply → archive
Claude Code cộng AIUP marketplace	Slash command theo phase
IntelliJ AIUP Navigator	Link spec ↔ test
Cursor / Copilot / Codex CLI	Làm việc với spec Markdown trong IDE quen thuộc
PlantUML / Mermaid	Diagram trong spec
LaunchDarkly / Unleash	Feature flag, rất hữu ích cho brownfield

7.5. Lời kết

Mình muốn kết bằng Nam, Tech Lead fintech ở Chương 1.

Sáu tháng sau khi team anh bị chậm lại vì vibe coding, Nam bắt đầu chuyển cách làm sang SDD. Đó không phải một cú chuyển đổi hoành tráng; anh chỉ chọn một module, viết lại spec, thêm AC test, rồi dần dần ép các PR mới link về UC.

Mình hỏi:

> Mất gì nhất khi chuyển?

Nam nói:

> Mất cái ảo tưởng rằng AI sẽ thay mình suy nghĩ. AI làm phần gõ nhanh hơn rất nhiều, nhưng phần khó nhất vẫn là diễn đạt cho rõ mình muốn gì. Trước đây phần đó dễ bị bỏ qua; giờ nó thành lợi thế cạnh tranh.

Đó cũng là điều mình muốn bạn mang theo.

Spec không phải gánh nặng, mà là nơi cả team thống nhất ý định trước khi tốc độ của AI khuếch đại mọi thứ lên. Khi spec rõ, AI giúp team đi nhanh hơn; khi spec mơ hồ, AI vẫn làm team đi nhanh hơn, chỉ là rất có thể nhanh về sai hướng.

Bạn không cần áp dụng cả cuốn sách ngay. Hãy chọn một use case, một sprint, một dev và một người phía nghiệp vụ. Viết spec vừa đủ, sinh test theo AC, để AI viết phần code, rồi đo xem điều gì xảy r a.

Sau vài vòng, bạn sẽ biết SDD hợp với team mình đến đâu.

Phụ lục

Phụ lục này để copy dùng lại. Bạn không cần đọc từ đầu đến cuối. Khi bắt đầu một repo hoặc một pilot SDD, mở phần template và chỉnh cho phù hợp với team.

A. Template `UC-XXX-Name.md`

```
# UC-XXX: <Tên use case ngắn gọn theo ngôn ngữ nghiệp vụ>

## Metadata
- **ID:** UC-XXX
- **Bounded Context:** <vd: Checkout, Shipping, Billing>
- **Liên quan tới BR:** BR-YYY
- **Status:** draft | reviewed | implemented | deprecated
- **Owner:** <PO/Dev chịu trách nhiệm>
- **Last updated:** YYYY-MM-DD

## Actor
<Ai khởi tạo use case này: người dùng, admin, system, scheduled job,
webhook...>

## Trigger
<Khi nào use case bắt đầu>

## Preconditions
- <Điều kiện phải đúng trước khi use case chạy>
- ...

## Main Flow
1. <Bước 1>
2. <Bước 2>
3. ...

## Alternative Flows
```

```
- **<N>a. <Tên biến thể>:** <mô tả>
- ...

## Exceptions
- **E1. <Tên exception>:** <mô tả + cách hệ thống xử lý>
- ...

## Postconditions
- <Trạng thái hệ thống sau khi use case thành công>

## Acceptance Criteria

### AC-1: <Tên ngắn gọn>
Given: <context>
When: <action>
Then: <expected outcome>
And: <expected outcome bổ sung>

### AC-2: ...

## Dependencies
- **Upstream UC:** <UC khác phải xong trước>
- **Downstream UC:** <UC khác phụ thuộc vào UC này>
- **External Systems:** <vd: Stripe, Sendgrid, internal LDAP>

## Notes
<Context bổ sung: quyết định lịch sử, lý do nghiệp vụ, link tới ADR, open
question>

## History
- v1 (YYYY-MM-DD, <author>): initial
- v2 (YYYY-MM-DD, <author>): added AC-4 for idempotency
- v3 (YYYY-MM-DD, <author>): clarified exception E2
```

B. Template `BR-XXX-Name.md`

```
# BR-XXX: <Tên business requirement>

## Metadata
- **ID:** BR-XXX
- **Status:** draft | approved | in-progress | done
- **Owner:** <PO/người phụ trách nghiệp vụ>
- **Stakeholders:** <list>
- **Target Quarter:** <Q3-2025>

## Background
<Vì sao có requirement này: context kinh doanh, customer feedback,
regulatory...>

## Goal
<Một câu rõ ràng. Tránh "tối ưu", "cải thiện" nếu chưa có metric>

## Success Metrics
- <Metric 1>: từ X → Y, đo qua <công cụ>, baseline tháng <N>
- <Metric 2>: ...

## In Scope
- <Những thứ làm trong phạm vi requirement này>

## Out of Scope
- <Những thứ cố ý không làm>

## Related Use Cases
- UC-XXX: ...
- UC-YYY: ...

## Constraints
```

```
- **Technical:** <vd: phải tương thích với hệ thống X>
- **Regulatory:** <vd: tuân thủ GDPR, PCI-DSS>
- **Timeline:** <deadline cứng nếu có>

## Open Questions
- [ ] <Câu hỏi chưa có câu trả lời từ phía nghiệp vụ>

## History
- v1 (YYYY-MM-DD, <author>): initial
```

C. Cheatsheet command

GitHub Spec Kit

Command	Dùng để làm gì
`/specify`	Khởi tạo spec từ ý tưởng
`/plan`	Sinh kế hoạch triển khai từ spec
`/tasks`	Tách plan thành task nhỏ
`/implement`	Để AI triển khai từng task

Luồng cơ bản:

```
/specify → review spec → /plan → /tasks → /implement
```

OpenSpec

Cài đặt nhanh:

```
npm install -g @fission-ai/openspec@latest
openspec init
```

Luồng nhanh trong profile `core`:

```
/opsx:propose → /opsx:apply → /opsx:sync → /opsx:archive
```

Command	Dùng để làm gì

`/opsx:propose`	Tạo change folder và các artifact cần trước khi triển khai
`/opsx:explore`	Khám phá vấn đề khi requirement chưa rõ, chưa tạo artifact
`/opsx:apply`	Triển khai dựa trên tasks/design/spec trong change hiện tại
`/opsx:sync`	Đồng bộ trạng thái artifact nếu có thay đổi trong quá trình làm
`/opsx:archive`	Merge delta specs vào baseline và lưu change vào archive

Expanded workflow có thêm các lệnh như `/opsx:new`, `/opsx:continue`, `/opsx:ff`, `/opsx:verify`. Dùng khi team muốn kiểm soát từng artifact thay vì đi đường nhanh.

Layout OpenSpec hay dùng:

```
openspec/
├── specs/              # baseline: hành vi hiện tại của hệ thống
├── changes/            # mỗi thay đổi một folder riêng
│   └── <change-name>/
│       ├── proposal.md # vì sao làm, scope là gì
│       ├── specs/      # delta spec: ADDED/MODIFIED/REMOVED
│       ├── design.md   # hướng kỹ thuật
│       └── tasks.md    # checklist triển khai
└── config.yaml
```

Cách nghĩ quan trọng: đừng sửa baseline một cách tùy tiện khi đang nghĩ về feature mới. Tạo change riêng, làm rõ proposal và delta spec, triển khai, verify, rồi archive để baseline phản ánh hành vi mới.

AIUP Marketplace cho Claude Code

Cài đặt:

```
/plugin marketplace add ai-unified-process/marketplace
/plugin install aiup-core
/plugin install aiup-vaadin-jooq   # nếu stack Java Vaadin+jOOQ
```

Commands `aiup-core`:

Command	Dùng để làm gì
`/requirements`	Tạo Business Requirement Catalog
`/entity-model`	Sinh entity model
`/use-case-diagram`	Sinh PlantUML use case diagram
`/use-case-spec`	Viết spec chi tiết cho một use case
`/reverse-engineer`	Brownfield: sinh spec stub từ code hiện có

Commands `aiup-vaadin-jooq` ví dụ:

Command	Dùng để làm gì
`/flyway-migration`	Tạo Flyway migration file
`/implement`	Triển khai use case với Vaadin + jOOQ
`/browserless-test`	Sinh Vaadin Browserless unit test
`/playwright-test`	Sinh Playwright E2E test

Git convention

```
<type>(UC-XXX): <description>
```

Ví dụ:

```
feat(UC-042): implement QR expiry validation
fix(UC-019): correct overlap detection edge case
docs(BR-001): clarify out-of-scope items
test(UC-042): add AC-4 idempotency test
```

PR template gợi ý

```
## UC liên quan
- UC-XXX: <link tới spec>
```

```
## Loại thay đổi
- [ ] Spec change (file `.md` only)
- [ ] Code change (link tới spec PR đã merge: #YYY)
- [ ] Bug fix (link tới incident: #ZZZ)
- [ ] Refactor (characterization test pass: <N>/<M>)

## Acceptance Criteria đã triển khai
- [ ] AC-1: <tên>
- [ ] AC-2: <tên>
- ...

## Test
- [ ] Mỗi AC có ít nhất 1 test
- [ ] Test name có UC-ID + AC-N

## AI usage
- AI gợi ý, người review: <list>
- Người viết chính: <list>
- AI sinh phần lớn, cần review kỹ: <list>
```

D. Glossary thuật ngữ

Acceptance Criteria (AC)

Điều kiện cụ thể để biết một use case đã xong đúng. Thường viết dạng Given/When/Then. Mỗi AC nên có ít nhất một test tương ứng.

Adapter

Phần code nối hệ thống với bên ngoài như database, API, email provider, payment gateway. Trong Hexagonal Architecture, adapter có thể đổi mà không làm lệch domain.

Agile

Cách làm phần mềm theo vòng ngắn, có feedback liên tục. Agile không chống lại spec; nó chống lại spec cứng nhắc và không được cập nhật.

AI Conductor

Vai trò của dev khi làm với AI: đưa spec, đưa context, kiểm tra kết quả, và giữ quyết định quan trọng. Không chỉ "prompt rồi merge".

AI Unified Process (AIUP)

Một phương pháp Spec-Driven Development do Simon Martinelli phát triển, có các giai đoạn, nguyên tắc và plugin hỗ trợ.

Brownfield

Dự án trên hệ thống có sẵn, thường là legacy. Khó hơn greenfield vì phải hiểu hành vi cũ trước khi sửa.

Bounded Context

Phạm vi mà một từ có một nghĩa rõ ràng. `Order` trong Checkout có thể khác `Order` trong Shipping.

Business Requirement (BR)

Tài liệu trả lời "vì sao làm việc này". BR nên có Goal, Success Metric, Out of Scope.

Characterization Test

Test ghi lại hành vi hiện tại của legacy code. Nó không chứng minh code đúng; nó giúp biết refactor có làm đổi hành vi cũ không.

Code Review

Quy trình dev khác đọc thay đổi trước khi merge. Trong SDD, review nhìn cả code lẫn độ khớp giữa spec và code.

DDD (Domain-Driven Design)

Phương pháp thiết kế phần mềm quanh domain nghiệp vụ. Các khái niệm hay dùng: Ubiquitous Language, Bounded Context, Entity, Aggregate.

Entity Model

Mô hình mô tả các "danh từ" trong hệ thống và mối quan hệ giữa chúng. Khác ERD ở chỗ tập trung vào ngôn ngữ nghiệp vụ, chưa đi sâu database.

Greenfield

Dự án mới hoàn toàn, chưa có code cũ. Dễ áp dụng SDD hơn brownfield vì chưa phải xử lý lịch sử.

Hexagonal Architecture

Kiến trúc tách domain core khỏi database, API, UI và các hệ thống ngoài. Rất hợp với AI vì AI có thể viết lại adapter mà không đụng domain.

Iteration

Một vòng làm việc ngắn: spec → code → test → feedback → cập nhật spec.

Legacy Code

Code đã tồn tại, thường thiếu doc hoặc test. Legacy không nhất thiết xấu; nó chỉ là code có lịch sử mà team hiện tại phải hiểu.

MVP (Minimum Viable Product)

Phiên bản nhỏ nhất có giá trị thật cho user, đủ để kiểm tra giả thuyết sản phẩm.

Port

Interface giữa domain core và adapter. Domain nói "tôi cần lưu Order", adapter quyết định lưu bằng Postgres, DynamoDB hay thứ khác.

Prompt

Câu hoặc context bạn đưa cho AI. Trong SDD, prompt tốt thường là spec + context, không phải một câu rời rạc.

Regen (Regenerate)

AI tạo lại code từ spec. Test bảo vệ việc regen để hành vi không bị lệch.

Refactor

Sửa cấu trúc code mà không đổi hành vi. Trong brownfield, nên có characterization test trước khi refactor.

Spec (Specification)

Tài liệu mô tả cần làm gì và thế nào là xong đúng. Trong SDD, spec là nơi team thống nhất ý định trước khi code chạy theo.

Spec-Driven Development (SDD)

Phương pháp đặt spec ở trung tâm. Con người thống nhất spec; AI sinh code, test, doc xoay quanh spec.

Stakeholder

Người chịu ảnh hưởng nếu sản phẩm đúng hoặc sai: user, PO, founder, customer, operations, regulator.

User Story

Cách viết yêu cầu ngắn theo công thức *"Là X, tôi muốn Y, để Z"*. Phù hợp làm điểm khởi đầu cho hội thoại, nhưng thường chưa đủ chi tiết để AI viết code an toàn, cần mở rộng thành use case đầy đủ với Acceptance Criteria.

Use Case (UC)

Mô tả một tương tác giữa actor và hệ thống để đạt một mục tiêu nghiệp vụ. Có Actor, Trigger, Main Flow, Alternative Flow, Exception và Acceptance Criteria. Là đơn vị làm việc chính trong SDD.

Vibe Coding

Cách làm việc với AI theo kiểu "chat thử rồi merge". Nhanh ngắn hạn, dễ tạo nợ kỹ thuật dài hạn. Đối lập với SDD.

Waterfall

Phương pháp truyền thống: spec đầy đủ trước, thiết kế, code, test, release tuần tự, ít quay lại. SDD không phải waterfall vì nó làm theo vòng lặp, không theo dòng chảy một chiều.

Vậy là bạn đã đọc hết cuốn “Spec Driven Development” của mình.
Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. Phát triển phần mềm trong thời đại AI là điều rất thú vị, và mình mong muốn mọi người đều có những giải pháp tốt nhất cho ứng dụng của mình. Mình hy vọng ebook này sẽ giúp đỡ bạn phần nào trên con đường nâng cấp và cải thiện kỹ năng, cũng như team của bạn làm việc luôn hiệu quả.
Cám ơn bạn rất nhiều !
Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:
Email: huynt57@gmail.com
Facebook: Tại đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !