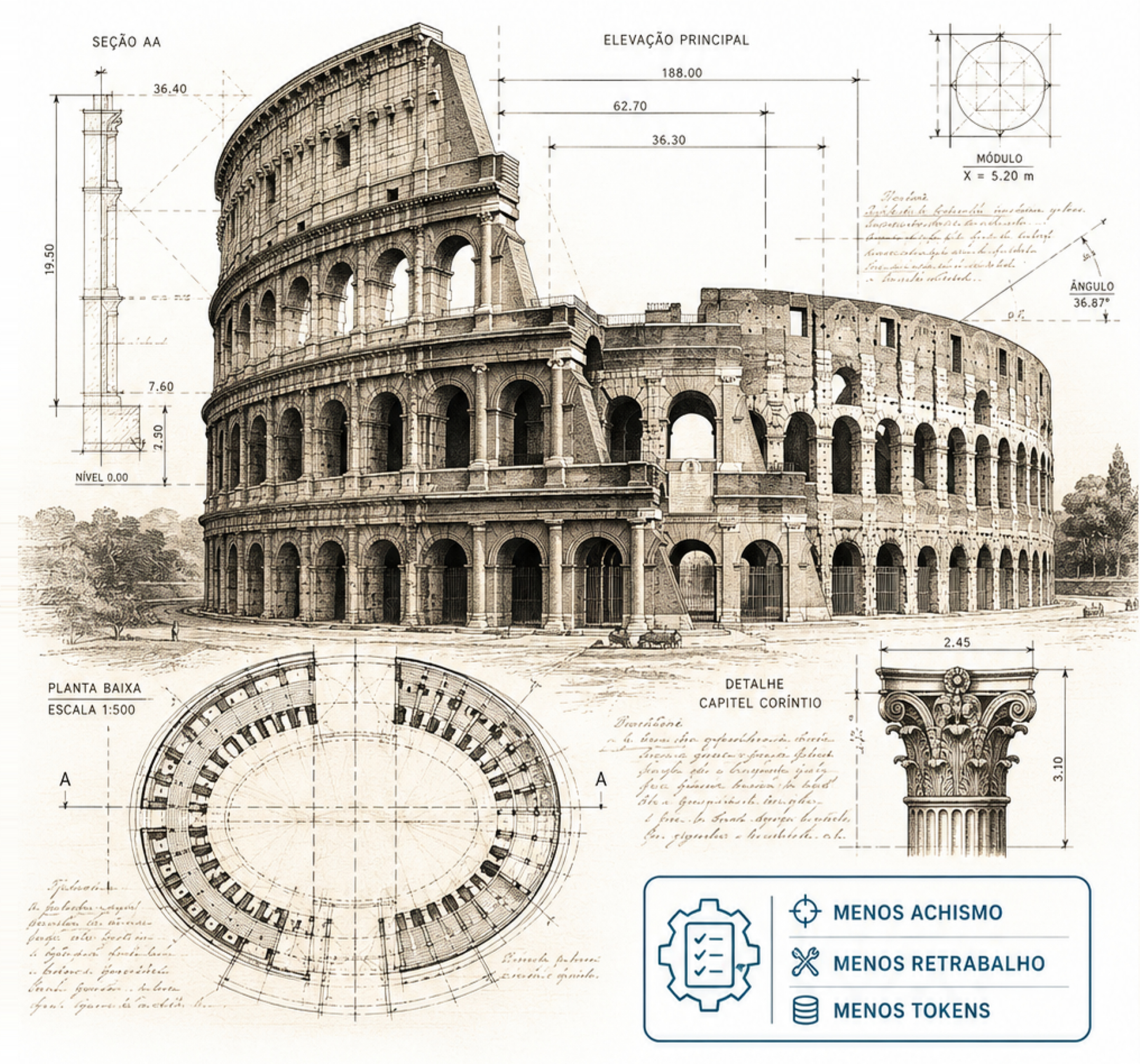
J.C. KÖDEL

SPEC DRIVEN DEVELOPMENT

DA **VIBE-CODING** À **ENGENHARIA DE** SOFTWARE

ESPECIFIQUE MELHOR. OBTENHA MELHOR CÓDIGO. GASTE MENOS TOKENS.



Índice

Esta é uma amostra gratuita. Adquira a versão completa para ler todos os capítulos.

- ✅ **0** - Por que SDD é essencial na era da IA
- ✅ **1** - Fundamentos: de onde vem o SDD
- 🔒 **2** - Anatomia de uma especificação — *versão completa*
- 🔒 **3** - Mãos à obra: as ferramentas de SDD — *versão completa*
- 🔒 **4** - speckit constitution: as regras antes da primeira jogada — *versão completa*
- 🔒 **5** - O ciclo completo do SDD — *versão completa*
- 🔒 **6** - Criar e listar tarefas: o primeiro loop completo — *versão completa*
- 🔒 **6.5** - Criar e listar tarefas na prática: o loop inteiro, arquivo por arquivo — *versão completa*
- 🔒 **7** - Concluir uma tarefa: quando o óbvio esconde decisões — *versão completa*
- 🔒 **7.5** - Concluir e reabrir tarefa na prática: o loop inteiro, arquivo por arquivo — *versão completa*
- 🔒 **8** - Filtrar tarefas: o holofote no `plan` e no `checklist` — *versão completa*
- 🔒 **8.5** - Filtrar tarefas na prática: o loop inteiro, arquivo por arquivo — *versão completa*
- 🔒 **9** - Excluir tarefa: o holofote no `tasks` e no `analyze` — *versão completa*
- 🔒 **9.5** - Excluir tarefa na prática: o loop inteiro, arquivo por arquivo — *versão completa*
- 🔒 **10** - Editar tarefa: o holofote no `implement` — *versão completa*
- 🔒 **10.5** - Editar tarefa na prática: o loop inteiro, arquivo por arquivo — *versão completa*
- 🔒 **11** - A app inteira nasceu de cinco loops — *versão completa*
- 🔒 Glossário canônico — *versão completa*

0 - Por que SDD é essencial na era da IA

Você abre o editor, descreve em uma frase o que precisa e, segundos depois, uma inteligência artificial devolve código que compila, roda e até parece bem feito. A cena que há poucos anos seria ficção virou rotina. E ela traz junto uma pergunta incômoda, honesta, que talvez tenha te trazido até aqui: se a máquina já constrói tão bem, por que ainda valeria a pena eu entender o que está sendo construído e gastar tempo descrevendo isso com cuidado antes de pedir?

É uma pergunta legítima. Fingir que a IA é fraca seria desonesto: ela não é. Em boa parte das tarefas do dia a dia, ela escreve código bom, rápido e barato. Mas a pergunta certa não é "a IA programa melhor que eu?". É outra, mais útil: o que separa quem usa essas ferramentas para construir coisas sólidas de quem apenas cola respostas que não entende? Quem só improvisa pedidos soltos à IA, sem método nem descrição clara do que quer, está fazendo o que se costuma chamar de *vibe-coding*: programar no clima, na vibe, torcendo para o resultado servir. Este material é a resposta a esse improviso, e ao final do capítulo espero que você saia convencido não por mim, mas por você mesmo.

A tese: conhecimento e especificação são alavanca

Antes de qualquer "como", precisamos do "porquê", e ele cabe em uma ideia só, o fio condutor de tudo o que vem a seguir:

> Conhecimento é alavanca. Entender o que você quer e saber descrever isso não compete com a inteligência artificial: multiplica o que você consegue fazer com ela.

Uma alavanca não substitui sua força, amplifica a que você já tem. Quem não tem força nenhuma para aplicar não levanta nada, por melhor que seja a alavanca. É assim com a IA. Ela amplia quem entrega um pedido claro do problema e expõe quem só joga frases vagas. Para quem entende o que está construindo e consegue especificar, ou seja, dizer com precisão o que quer e por quê, a IA é um multiplicador: entrega rascunhos em segundos e elimina o tédio do código repetitivo. Para quem não entende nem descreve, ela vira uma fábrica de código que parece certo e ninguém sabe avaliar.

Repare na inversão. O senso comum diz: “se a IA faz, não preciso me envolver”. A tese deste material diz o oposto: justamente porque a IA faz, especificar bem importa mais. Quando produzir código deixa de ser o gargalo, o valor migra para tudo o que a digitação nunca resolveu sozinha: saber o que construir, julgar se está certo, decidir entre caminhos e responder pelo resultado. A ferramenta ficou poderosa. A alavanca, o seu entendimento transformado em especificação, é o que decide até onde essa potência te leva.

Os três argumentos: avaliar, decidir, responder

A tese soa bonita, mas precisa se sustentar. Vejamos três razões concretas, da mais decisiva à mais ampla, pelas quais o entendimento continua sendo a alavanca, mesmo quando a IA faz o trabalho braçal.

Primeiro, alguém precisa avaliar o que a máquina produziu. A IA gera código plausível, e plausível é uma palavra perigosa. Quase sempre o que ela escreve está certo. O problema mora na minoria: o trecho que compila, passa no teste óbvio e quebra em silêncio num caso raro que ninguém pensou em verificar, como uma falha de permissão em que um usuário enxerga dados que não são dele. Quem percebe esse defeito sutil? Apenas quem sabe o que o código deveria fazer, e isso vem de ter especificado o comportamento esperado antes. Sem uma especificação clara na cabeça ou no papel, avaliar vira torcer para a IA ter acertado.

Segundo, alguém precisa decidir o que construir e por quê. A IA implementa o que você pede, mas o que pedir, e por que daquele jeito, continua sendo seu. Esta tela precisa funcionar sem internet? Vale a complexidade de sincronizar dados ou o problema não justifica? A IA sugere opções competentes, mas não carrega o contexto do seu produto, seus usuários, seu orçamento, o que vai doer manter daqui a dois anos. Decidir é o próprio ato de especificar: pedir a coisa certa vale mais do que receber rápido a coisa errada.

Terceiro, a responsabilidade não se delega. Quando o sistema vai ao ar, a autoria é sua. Se dados vazam ou a conta de nuvem dispara, não existe “a IA que escreveu”. E é impossível responder por algo que você não entende. Assumir o resultado exige conseguir explicar por que o sistema é como é, e isso só existe quando houve intenção registrada, uma especificação, e não um amontoado de pedidos improvisados.

Três argumentos, uma só tese: avaliar, decidir e responder são exatamente o que a IA não faz por você, e exatamente o que a especificação te ajuda a dominar.

O exemplo dos 70% e dos 30%

Se você já usa IA para programar, talvez reconheça uma cena assim. Você pede uma funcionalidade, digamos uma tela que lista itens, filtra por situação e atualiza quando algo muda. Em segundos vem uma resposta enorme e impressionante. A estrutura está lá, os nomes fazem sentido, boa parte simplesmente funciona. Esses são os **70%**: o trabalho previsível, que já apareceu milhares de vezes em milhares de projetos parecidos. A IA é extraordinária nesses 70%, e é ótimo que seja. É tempo seu que volta para o bolso.

Mas então começam os **30%**. A lista funciona com dez itens e engasga com dez mil. O filtro ignora uma situação que só existe no seu produto. A atualização em tempo real funciona online e some no primeiro ponto cego de sinal. Nenhum desses 30% é sobre digitar mais código. São sobre julgamento: perceber o que falta, entender por que falha e decidir como corrigir sem derrubar o resto. E há uma armadilha cruel: quem não consegue fazer os 30% também não percebe que eles estão faltando. Aceita os 70% como se fossem 100%, entrega, e descobre o buraco quando o usuário cai nele. Os 70% são velocidade. Os 30% são valor, e o valor mora em saber, antes de pedir, o que de fato precisa existir.

O que é SDD, em linguagem simples

O método que captura esse valor tem nome: **SDD**, sigla em inglês para *Spec Driven Development*, ou desenvolvimento guiado por especificação. A ideia é simples: você descreve com clareza o que quer, o problema, as regras, o que conta como certo, antes de pedir o código. A especificação vira o ponto de partida, e o código vem depois, para cumpri-la.

Pense em construir uma casa. Ninguém entrega tijolos ao pedreiro e manda começar. Primeiro vem a planta: onde ficam as paredes, quantos cômodos, por onde passa a água. A planta é a especificação; a obra é o código. Com a planta na mão, dá para conferir se a parede saiu no lugar certo. Sem ela, você só descobre o erro quando a parede já está de pé. SDD é desenhar a planta antes de levantar a obra.

Por que isso é essencial agora? Porque a era da IA barateou a obra e encareceu a falta de planta. Sem especificação, o improviso do vibe-coding produz dois problemas concretos. O primeiro são prompts caros e improdutivos: você descreve mal, recebe algo torto e descreve de novo, brigando com a máquina mais do que pensar com calma teria custado. O segundo é o código indecifrável: a IA entrega muito, rápido, e você acumula um sistema que ninguém entende nem consegue manter. Quanto mais a IA produz, mais perigoso é não saber o que pedir.

Esse primeiro problema tem uma raiz técnica que explica por que o improviso sai caro. A IA processa texto em *tokens* (pedaços de palavra, a unidade que ela lê, gera e cobra) e não guarda memória própria entre um pedido e o outro. Tudo o que ela precisa saber da sua tarefa tem que caber no *contexto*: a janela de texto que volta junto a cada interação. No vibe-coding, esse contexto é a conversa inteira, e ela só cresce. A cada novo prompt, a IA relê um histórico cada vez maior para adivinhar o que você quer, queima mais tokens nesse retrabalho e perde precisão conforme a conversa se alonga. Com SDD, a referência não é o bate-papo, é a especificação: um documento curto, estável e organizado. A IA executa contra esse contrato claro em vez de remontar sua intenção a partir de uma conversa longa. Menos idas e vindas, menos texto reprocessado, menos tokens para o mesmo lugar. SDD é a disciplina que mantém o leme na sua mão.

SDD não é exatamente novo

Vale uma dose de honestidade contra o hype: especificar antes de construir não foi inventado agora. Décadas atrás, a forma dominante de fazer software era o *waterfall*, ou cascata, um modelo que escrevia toda a especificação no início e só depois construía, em fases que desciam em sequência como uma cachoeira. SDD herda dele a parte boa: pensar no problema antes de sair codando.

O waterfall puro, porém, fracassou, e por um motivo claro. Ele apostava que dava para acertar a especificação inteira de uma vez, no começo, e que nada mudaria. A realidade muda sempre, e voltar atrás custava caro: a obra já estava erguida sobre a planta errada. Foi para corrigir essa rigidez que surgiram o *agile* e o *scrum*, abordagens que entregam em ciclos curtos, com feedback frequente, ajustando a rota a cada passo em vez de apostar tudo num plano fixo.

E o que a era da IA muda nessa equação? Ela derruba o custo que afundou o waterfall. Reescrever ficou barato. Se a especificação precisa mudar, a IA refaz o código em minutos. SDD, hoje, não é o waterfall rígido de volta: é especificar com intenção e, ainda assim, iterar barato, juntando a clareza da planta com a leveza dos ciclos curtos. O melhor dos dois mundos ficou viável.

> Para quem vem do mundo ágil: SDD não substitui sua sprint. A especificação vira um artefato vivo, revisado a cada ciclo, e a IA é quem torna a reescrita barata o bastante para isso valer a pena.

O próximo passo

Se a tese fez sentido, você já tem o essencial: na era da IA, o gargalo não é mais produzir código, e sim saber o que pedir e avaliar o que volta. Entender e especificar é a alavanca que amplia tudo o que a máquina entrega. É isso que separa quem constrói coisas sólidas de quem só cola respostas.

Antes de dar forma prática a esse método, vale uma pergunta honesta: especificar antes de construir não seria o jeito antigo de fazer software, aquele que o mundo passou anos tentando abandonar? No próximo capítulo você verá de onde o SDD vem. A ideia de especificar antes já provou seu valor por décadas, e cada metodologia, do waterfall ao ágil, deixou uma lição que o SDD herda. O que mudou foi o momento: a IA recoloca essa ideia comprovada em jogo, agora sem o custo que a afundava.

> Uma confissão antes de seguir: este material foi escrito usando SDD. Cada capítulo nasceu de uma especificação antes da primeira frase, a forma de manter coesão, não esquecer detalhes e checar, a cada passo, se ainda fazia sentido. O que você lê é texto humano, escrito por mim, embasado em fatos: a especificação foi o andaime, não o autor. SDD não é vibe-writing, assim como não é vibe-coding. É o contrário disso: a planta na mão antes de levantar a parede, seja a obra um software ou um texto.

---

1 - Fundamentos: de onde vem o SDD

No capítulo anterior você ficou com uma ideia na mão: especificar antes de pedir o código é o que separa quem constrói coisas sólidas de quem só cola respostas que não entende. Faz sentido. Mas talvez tenha ficado também uma suspeita incômoda, e é melhor encará-la de frente: descrever tudo com cuidado antes de construir não é justamente o jeito antigo de fazer software, aquele pesado e burocrático que o mundo passou trinta anos tentando abandonar?

Se você já trabalhou em equipe, conhece a fadiga. Documento que ninguém lê, reunião para aprovar reunião, um plano gigante que a realidade atropela na primeira semana. Quem viveu isso aprendeu, com razão, a desconfiar de qualquer um que chegue pregando "vamos planejar tudo antes". A suspeita é legítima, e este capítulo não vai fingir que ela não existe.

O que ele vai fazer é separar duas coisas que costumam vir grudadas: o instinto de pensar antes de construir, que sempre teve valor, e o custo de mudar tarde, que foi o que de fato afundou o modelo antigo. Para isso a gente precisa voltar um pouco no tempo e olhar, sem pressa, como o software foi feito antes de você chegar. Não por saudade. Pelo contrário: você vai ver que cada método nasceu corrigindo o erro do anterior, e que o SDD é o próximo passo dessa linha, não um retorno ao começo dela.

Waterfall: o instinto certo, o custo errado

Imagine construir uma casa. Ninguém entrega tijolos ao pedreiro e manda começar. Primeiro vem a planta: onde ficam as paredes, quantos cômodos, por onde passa a água. Só depois se levanta a estrutura, e só depois de pronta vem a pintura. Cada etapa começa quando a anterior termina, e voltar atrás é caro: quebrar uma parede já erguida custa muito mais do que mover um risco na planta. Essa é a intuição por trás do *waterfall*, ou cascata: o modelo que especifica tudo no início e depois constrói em fases que descem em sequência, como uma cachoeira, sem voltar para cima.

As fases clássicas são essas: levantar requisitos, projetar, implementar, integrar, testar e manter, uma depois da outra. Esse modelo costuma ser atribuído a um artigo de Winston Royce de 1970, "Managing the Development of Large Software Systems" ("Gerenciando o Desenvolvimento de Grandes Sistemas de Software").[1] E aqui mora uma ironia que vale

conhecer: Royce desenhou o diagrama da cascata para dizer que ele *não* funcionava bem. Ele apresentou a sequência pura como um exemplo arriscado e defendia adicionar idas e vindas entre as fases. A indústria copiou o diagrama que ele criticava e ignorou os remédios que ele sugeria.

> Curiosidade para quem gosta de história: o próprio termo “waterfall” não aparece no artigo de Royce; ele se popularizou depois, em textos dos anos 1970 que citavam o trabalho dele. O nome que virou sinônimo de “jeito antigo” nasceu de uma leitura incompleta do autor que o descrevia.

Repare no que aconteceu de fato. O instinto do waterfall estava certo: pensar no problema antes de sair construindo evita levantar a parede no lugar errado. Esse instinto nunca foi o defeito. O defeito foi a aposta de que dava para acertar a especificação inteira de uma vez, no começo, e que nada mudaria depois. A realidade muda sempre. O cliente entende melhor o que quer só quando vê algo pronto; o mercado se move; um detalhe esquecido aparece no fim. E mudar lá no fim, com a obra erguida sobre a planta errada, custava caro. Estimativas da época falavam em corrigir tarde um erro custando dezenas de vezes mais do que corrigi-lo cedo.[2] O problema do waterfall, então, não era planejar. Era não ter como planejar de novo sem pagar uma fortuna.

A virada ágil: aprender a iterar barato

Se mudar tarde é caro, a saída é não deixar para mudar tarde. Em vez de erguer a casa inteira de uma vez sobre uma planta fechada, e que tal levantar primeiro um cômodo, morar nele, ver o que incomoda e ajustar antes de seguir? É essa a ideia do desenvolvimento *iterativo/incremental*: construir em ciclos curtos, entregando e ajustando aos poucos, em vez de tudo de uma vez no fim. Cada ciclo produz algo utilizável, recebe feedback e corrige a rota do ciclo seguinte, enquanto corrigir ainda é barato.

Pode parecer invenção recente, mas não é. Há registros de desenvolvimento iterativo já nos anos 1950, e o programa espacial Mercury, da NASA, nos anos 1960, é um exemplo documentado de construir e testar em pequenos passos.[3] Iterar não nasceu com a moda; o que faltava era nome, valores comuns e gente disposta a defender a prática contra o peso do waterfall.

Esse nome chegou em 2001. Dezessete profissionais de software se reuniram em Snowbird, no estado de Utah, e escreveram o *Manifesto Ágil*: um documento curto que fixou quatro valores para desenvolver software.[4] Traduzidos da fonte original, eles dizem que se deve valorizar:[5]

- indivíduos e interações mais que processos e ferramentas;
- software em funcionamento mais que documentação abrangente;
- colaboração com o cliente mais que negociação de contratos;
- responder a mudanças mais que seguir um plano.

Leia a última linha com atenção, porque ela é a resposta direta à dor do waterfall. O waterfall tratava o plano como sagrado e a mudança como fracasso. O Manifesto inverte: a mudança é esperada, e o valor está em responder a ela rápido. Não é abandonar o plano, é parar de fingir que ele estaria certo do começo ao fim. *Agile*, ou ágil, é exatamente isso: entregar em ciclos curtos, com feedback frequente, ajustando a rota a cada passo em vez de apostar tudo num plano fixo.

Scrum, o pilar; XP e kanban, o apoio

Valores precisam de prática para virar rotina, e o ágil ganhou corpo em métodos concretos. O principal deles, o que mais moldou a forma como as equipes trabalham até hoje, é o *scrum*.

Scrum é uma abordagem ágil baseada em ciclos curtos com papéis e eventos definidos e feedback frequente. O ciclo curto tem nome próprio: *sprint* (em inglês, uma arrancada, uma corrida curta e intensa), um período de duração fixa, em geral de uma a quatro semanas (duas é o mais comum), ao fim do qual existe algo pronto para mostrar e avaliar. A cada sprint a equipe planeja o que cabe no período, trabalha, entrega e revisa o que fez, decidindo o próximo passo com base no que aprendeu. Em vez de uma única aposta gigante no início, são muitas apostas pequenas, cada uma corrigindo a anterior. Scrum foi apresentado publicamente por Ken Schwaber e Jeff Sutherland na conferência OOPSLA de 1995.[6] O nome vem de antes: um artigo de 1986 de Hirotaka Takeuchi e Ikujiro Nonaka, “The New New Product Development Game” (“O Novo Novo Jogo do Desenvolvimento de Produtos”), que comparava equipes de produto de alto desempenho à formação de scrum do rúgbi, em que o time avança junto, empurrando na mesma direção.[7]

> Para quem já trabalha com scrum: aqui ele entra só pela lição que interessa ao SDD, o ciclo curto que torna o feedback barato. Papéis como Product Owner e Scrum Master e eventos como a reunião diária existem e são úteis, mas não são o ponto deste capítulo.

Ao lado do scrum, dois métodos de apoio agregaram peças que vão reaparecer mais adiante. O *extreme programming* (XP), ou programação extrema, de Kent Beck, leva ao extremo boas práticas de engenharia. Beck o desenvolveu no projeto C3, da Chrysler, por volta de 1996, e o consolidou no trabalho "Extreme Programming Explained" ("Programação Extrema Explicada"), de 1999.[8] Duas práticas dele importam aqui. A *integração contínua*: juntar e testar o trabalho de todos com frequência, em vez de esperar o fim, para que erros apareçam cedo. E o *TDD* (Test-Driven Development), ou desenvolvimento guiado por testes: escrever o teste antes do código, de modo que o código nasça já provando que faz o que deveria. Os dois empurram o erro para perto da sua origem, onde é barato consertar.

O outro apoio é o *kanban*, formulado para software por David Anderson a partir de trabalho na Corbis em meados dos anos 2000 e descrito em seu trabalho de 2010, com raiz no sistema de produção da Toyota.[9] Kanban visualiza o trabalho em um quadro e limita o *trabalho em curso*, ou WIP (Work In Progress): o que já foi começado e ainda não terminou. Limitar o WIP evita o vício de começar muita coisa e terminar pouca; a equipe foca em concluir antes de puxar o próximo item. É fluxo, não lote.

O que cada método nos ensinou

Olhe a sequência inteira de uma vez e um padrão aparece. Não é uma pilha de modas soltas; é uma corrente, cada elo respondendo à fraqueza do anterior. Dela saem três lições que vão direto para o que vem a seguir.

A primeira veio do waterfall: **especificar dá direção**. Pensar no problema antes de construir evita erguer a coisa errada com competência. Esse instinto estava certo e continua valendo.

A segunda veio do ágil: **iterar dá adaptação**. Como a realidade muda, construir em ciclos curtos permite ajustar a rota antes que o desvio fique caro. Foi a resposta direta ao ponto cego do waterfall, que tratava a mudança como acidente em vez de regra.

A terceira veio de scrum, XP e kanban juntos: **feedback cedo reduz risco**. Sprints curtos, testar antes de codar, integrar sempre, limitar o trabalho em curso. Tudo aponta para a mesma direção: descobrir o que está errado o quanto antes, quando o conserto ainda é barato. Cada método refinou o anterior nesse ponto, encurtando a distância entre cometer um erro e percebê-lo.

Direção, adaptação e risco controlado. Guarde os três. O SDD não escolhe um e descarta os outros; ele tenta ficar com os três ao mesmo tempo, e o resto do capítulo é sobre como isso deixou de ser sonho.

SDD: a síntese que só agora ficou viável

Por que ninguém simplesmente juntou as duas forças antes? Por que não especificar com a clareza do waterfall e, ainda assim, iterar barato como o ágil? A resposta é que faltava uma peça, e a peça era o custo de reescrever.

Pense de novo na casa. Se mudar a planta significasse derrubar paredes prontas, você seguraria a planta com unhas e dentes e evitaria mexer, exatamente o reflexo do waterfall. Se erguer e derrubar paredes fosse instantâneo e quase de graça, você experimentaria à vontade, ajustaria a cada visita, e a planta viraria um documento vivo em vez de uma sentença. O que separava esses dois mundos sempre foi o preço da mudança. O ágil baixou esse preço com ciclos curtos e disciplina de equipe, mas reescrever software de verdade continuava lento e caro, feito a mão, linha por linha.

É aqui que a era da IA muda a equação. Quando produzir e refazer código deixa de ser o gargalo, o custo de reescrever despenca. Uma especificação que mudou pode ser executada de novo em minutos, não em semanas. E isso libera a combinação que antes não fechava a conta: especificar antes, com a direção que o waterfall ensinou, e ainda assim iterar barato, com a adaptação que o ágil ensinou. **Especificar antes de construir é uma ideia comprovada por décadas, e a IA não a ressuscita: ela a ressignifica**, dando à velha disciplina de pensar antes um custo de mudança que ela nunca teve.

Vou ser direto para não deixar dúvida: **isso não é voltar ao waterfall**. O waterfall congelava a especificação e punia quem mudasse de ideia. O SDD faz o oposto: trata a especificação como artefato vivo, feito justamente para mudar e ser reexecutada quantas vezes for preciso. A planta continua valiosa, mas deixou de ser uma prisão. Não estamos resgatando um método antigo nem forçando o passado de volta; estamos usando uma alavanca nova, a IA, para finalmente ficar com o melhor de cada herança ao mesmo tempo.

Por que a IA precisa de especificação

Vale fechar a conta do porquê, agora com a IA no centro. Quando produzir código deixa de ser o gargalo, o valor migra para saber o que pedir, e a especificação é o que torna esse pedido preciso e barato. Lembre de como a IA trabalha: ela lê tudo em *tokens*, a unidade de texto que ela processa e cobra, e não guarda memória própria entre um pedido e o outro. Tudo o que ela precisa saber tem que caber no *contexto*, a janela de texto reenviada a cada interação. No improviso do *vibe-coding*, esse contexto é a conversa inteira, que só cresce: a cada pedido a IA relê um histórico maior, queima mais tokens nesse retrabalho e perde precisão conforme a conversa se alonga. Com uma especificação, a referência deixa de ser o

bate-papo e passa a ser um documento curto, estável e organizado, contra o qual a IA executa direto. Menos idas e vindas, menos texto reprocessado, mais precisão pelo mesmo custo.

Há ainda uma segunda economia, menos óbvia e mais duradoura: a especificação vira a memória do projeto. As decisões, as regras e o porquê de cada escolha ficam registrados em um lugar só, que sobrevive ao fim da conversa, à troca de quem está no teclado e ao esquecimento de daqui a seis meses. Um bate-papo com a IA evapora; uma especificação fica, e é dela que o próximo ciclo parte. É por isso que este material insiste tanto no processo: especificar bem não é papelada, é o que mantém a IA precisa hoje e o projeto compreensível amanhã.

O ciclo, agora com nome

Essa síntese tem um ritmo de trabalho, e ele se organiza em quatro etapas que vão reaparecer do início ao fim do material. Vale fixar o vocabulário agora, ainda sem nenhuma ferramenta na frente:

- **specify** (especificar): descrever com clareza o que se quer, o problema e o que conta como certo, antes de pedir o código.
- **plan** (planejar): decidir como aquilo será construído, a abordagem e as decisões técnicas que sustentam a especificação.
- **tasks** (tarefas): quebrar o plano em passos concretos e executáveis, pequenos o bastante para acompanhar.
- **implement** (implementar): construir, agora com direção, cumprindo a especificação e o plano.

É a mesma lógica das três lições, agora em sequência de trabalho: especificar dá direção, o plano e as tarefas mantêm a adaptação organizada, e a implementação em passos curtos traz o feedback cedo. Existem ferramentas que dão forma a esse ciclo e cuidam da parte mecânica de cada etapa, e o material chega a elas mais adiante. Por ora, o que importa é reconhecer o vocabulário: quando você ler specify, plan, tasks e implement nos próximos capítulos, são estas quatro etapas.

O próximo passo

Recapitulando o trajeto: o waterfall acertou ao pedir direção e errou ao cobrar caro por mudanças; o ágil ensinou a iterar barato e a buscar feedback cedo; scrum, XP e kanban afinaram essa busca. O SDD herda os três aprendizados de uma vez, e isso só ficou viável agora porque a IA derrubou o custo de reescrever. A ideia de especificar antes é antiga e já provou seu valor; o que é novo é o momento, que finalmente deixa essa ideia render tudo o que ela prometia.

Você já sabe de onde o SDD vem e por que ele faz sentido agora. Falta ver de perto a peça central de tudo isso: a própria especificação. No próximo capítulo a gente abre a planta e examina as partes de uma boa especificação, o que ela precisa conter para guiar a construção e o que a torna clara o bastante para a máquina e para você. Da intenção ao documento que orienta, de verdade, o que será construído.

---

1. "Waterfall model", Wikipédia (em inglês), inclui a atribuição a Winston W. Royce, "Managing the Development of Large Software Systems" (1970), a origem do termo e o custo crescente de correções tardias: https://en.wikipedia.org/wiki/Waterfall_model · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Waterfall_model?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩

2. "Waterfall model", Wikipédia (em inglês), inclui a atribuição a Winston W. Royce, "Managing the Development of Large Software Systems" (1970), a origem do termo e o custo crescente de correções tardias: https://en.wikipedia.org/wiki/Waterfall_model · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Waterfall_model?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩

3. "Iterative and incremental development", Wikipédia (em inglês), sobre as raízes do desenvolvimento iterativo anteriores a 2001 (registros desde a década de 1950 e o programa Mercury da NASA): https://en.wikipedia.org/wiki/Iterative_and_incremental_development · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Iterative_and_incremental_development?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩

4. "Agile software development", Wikipédia (em inglês), sobre a redação do Manifesto Ágil em 2001 em Snowbird (Utah) por dezessete signatários: https://en.wikipedia.org/wiki/Agile_software_development · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Agile_software_development?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩

5. Valores citados da fonte primária, “Manifesto para Desenvolvimento Ágil de Software” (2001): https://agilemanifesto.org · tradução oficial em português: https://agilemanifesto.org/iso/ptbr/manifesto.html↩
6. “Scrum (software development)”, Wikipédia (em inglês), sobre Ken Schwaber e Jeff Sutherland, a apresentação na OOPSLA de 1995 e o conceito de sprint: https://en.wikipedia.org/wiki/Scrum_(software_development) · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Scrum_(software_development)?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩
7. Hirotaka Takeuchi e Ikujiro Nonaka, “The New New Product Development Game”, Harvard Business Review (1986), origem da metáfora “scrum”: https://hbr.org/1986/01/the-new-new-product-development-game↩
8. “Extreme programming”, Wikipédia (em inglês), sobre Kent Beck, o projeto C3 da Chrysler (por volta de 1996), o trabalho “Extreme Programming Explained” (1999) e práticas como TDD e integração contínua: https://en.wikipedia.org/wiki/Extreme_programming · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Extreme_programming?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩
9. “Kanban (development)”, Wikipédia (em inglês), sobre David J. Anderson, o trabalho na Corbis (meados dos anos 2000), o trabalho de 2010, a raiz no Sistema Toyota de Produção e o limite de trabalho em curso (WIP): https://en.wikipedia.org/wiki/Kanban_(development) · versão com tradução automática para português: https://en-wikipedia-org.translate.goog/wiki/Kanban_(development)?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=en&_x_tr_pto=wapp↩