DEGUSTAÇÃO GRATUITA

SDD EM DELPHI

Como especificar, projetar e desenvolver software Delphi com apoio de IA

RÉGYS BORGES DA SILVEIRA

Uma amostra para conhecer o livro

A Inteligência Artificial pode acelerar a escrita de código, mas não corrige um problema mal compreendido. Quanto mais rápida se torna a implementação, mais importante fica a clareza sobre objetivos, regras, limites e critérios de aceitação.

SDD em Delphi apresenta o Desenvolvimento Orientado por Especificações como uma forma prática de conectar intenção, arquitetura, código, testes, revisão e evidências. A IA participa como parceira, enquanto as decisões técnicas e a responsabilidade continuam humanas.

Esta degustação foi selecionada para mostrar tanto a base conceitual quanto a aplicação do método. Ela preserva a diagramação da edição impressa e pode ser compartilhada gratuitamente.

O que está incluído

- Como usar este livro e sumário completo
- Capítulo 1 - O que é SDD, na íntegra
- Trecho do Capítulo 4 - Anatomia de uma boa especificação
- Figuras, exemplos e prompts no padrão editorial do livro

Como usar este livro

Este livro pode ser lido de forma sequencial ou consultiva.

Na leitura sequencial, a Parte 1 cria a base conceitual do SDD, a Parte 2 ensina a transformar demandas em especificações, a Parte 3 aprofunda o uso de IA e prompts, a Parte 4 conecta especificação e arquitetura Delphi, a Parte 5 trata da implementação assistida, a Parte 6 aborda qualidade, governança e segurança, e a Parte 7 reúne estudos de caso.

Na leitura consultiva, o leitor pode ir diretamente ao tipo de problema que precisa resolver: requisitos, regras de negócio, contratos, prompts, arquitetura VCL, FireMonkey, APIs, FireDAC, testes, documentação, revisão de código ou modernização de legado.

Os exemplos foram escritos para desenvolvedores Delphi que precisam lidar com sistemas reais. Em alguns momentos o texto menciona VCL, FireMonkey, FireDAC, DEXT, banco de dados, APIs, testes e sistemas legados. Quando um termo técnico aparece pela primeira vez, a explicação fica no próprio texto, em nota ou no glossário.

Os prompts não devem ser tratados como fórmulas mágicas. Eles são exemplos de como transformar especificações em instruções mais verificáveis para ferramentas de IA. O ponto central do livro não é copiar prompts, mas aprender a pensar, especificar, revisar e validar melhor.

Roteiros de leitura por perfil

Se você está começando em Delphi ou ainda não tem familiaridade com SDD, leia o livro em ordem. A Parte 1 cria a base conceitual, a Parte 2 mostra como transformar demanda em especificação, e só depois as partes seguintes entram em prompts, arquitetura, implementação e governança. Essa ordem evita a armadilha de começar pela ferramenta antes de entender o método.

Se você já trabalha com Delphi legado, comece pela Parte 1 e avance com atenção especial aos capítulos sobre regras, critérios de aceitação, FireDAC, VCL, sistemas legados e estudos de caso. O objetivo, nesse percurso, é aprender a mapear comportamento existente antes de pedir refatoração, geração de código ou modernização com IA.

Se você atua como arquiteto, líder técnico ou analista, pode ler a Parte 1, a Parte 2 e depois saltar para os capítulos de arquitetura, governança, métricas e fluxo completo em equipe. Esse roteiro ajuda a transformar o SDD em prática de time: decisões registradas, padrões, revisão, evidências e limites para agentes de IA.

Se você quer aplicar rapidamente com agentes de código, leia primeiro os capítulos de anatomia da especificação, engenharia de prompts, context engineering, geração controlada, testes derivados da especificação e governança de IA. Depois volte aos estudos de caso para observar o ciclo inteiro funcionando.

Se você procura exemplos práticos, use o mapa de exemplos no catálogo de templates e consulte o repositório público de materiais complementares. Os exemplos são complementares ao texto: servem para estudo, revisão e futura adaptação a projetos Delphi 13 Florence / RAD Studio 13 Florence.

Código e materiais complementares

Os exemplos oficiais deste livro estão disponíveis em:

https://github.com/regyssilveira/sdd-em-delphi-exemplos

O repositório reúne regras de negócio, testes DUnitX, persistência com FireDAC e SQLite, contratos de API, um percurso completo de especificação até evidências executáveis e uma API construída com DEXT. O `README` apresenta os requisitos e os comandos de validação, enquanto o mapa do livro relaciona cada exemplo aos capítulos correspondentes.

Para reproduzir exatamente o código associado à primeira edição impressa, utilize a release `livro-1.0`. A branch principal

poderá receber correções posteriores sem alterar a versão que serviu de referência para estas páginas.

Sumário

Parte 1 - Fundamentos do SDD

Capítulo 1 - O que é SDD 15
Capítulo 2 - Por que Delphi precisa de especificações melhores 38
Capítulo 3 - IA como parceira, não como autora final 59
Capítulo 4 - Anatomia de uma boa especificação 77

Parte 2 - Construindo Especificações

Capítulo 5 - Do problema de negócio ao escopo técnico 108
Capítulo 6 - Requisitos funcionais em Delphi 132
Capítulo 7 - Requisitos não funcionais 161
Capítulo 8 - Regras de negócio e critérios de aceitação 187
Capítulo 9 - Dados, contratos e integrações 208
Capítulo 10 - Diagramas úteis sem burocracia 227

Parte 3 - IA, Prompts e Contexto

Capítulo 11 - Engenharia de prompts para Delphi 242
Capítulo 12 - Context engineering para agentes de código 251
Capítulo 13 - Frameworks SDD e fluxos orientados por especificações 273
Capítulo 14 - Prompts para análise, arquitetura e decisão 284
Capítulo 15 - Prompts para geração de código Delphi 291
Capítulo 16 - Prompts para revisão, refatoração e documentação 299

Parte 4 - Arquitetura Delphi Orientada por Especificações

Capítulo 17 - Organização de projetos Delphi 309
Capítulo 18 - Arquitetura para aplicações VCL 316
Capítulo 19 - Arquitetura para FireMonkey e multiplataforma 322
Capítulo 20 - APIs e serviços com Delphi 327
Capítulo 21 - FireDAC, banco de dados e transações 333
Capítulo 22 - Sistemas legados e modernização assistida por IA 339

Parte 5 - Implementação Assistida por IA

Capítulo 23 - Da especificação ao backlog técnico 347
Capítulo 24 - Geração controlada de código 356
Capítulo 25 - Testes derivados da especificação 365
Capítulo 26 - Documentação gerada e documentação validada 375
Capítulo 27 - Revisão de código com IA 383

Parte 6 - Qualidade, Governança e Segurança

Capítulo 28 - Qualidade de software no fluxo SDD 390
Capítulo 29 - Segurança e privacidade 399
Capítulo 30 - Governança de IA em equipes Delphi 407
Capítulo 31 - Métricas, produtividade e melhoria contínua 422

Parte 7 - Estudos de Caso

Capítulo 32 - Cadastro corporativo em VCL 434
Capítulo 33 - API REST com autenticação e contrato 443
Capítulo 34 - Modernização de módulo legado 451
Capítulo 35 - Aplicação multiplataforma com FireMonkey 458
Capítulo 36 - Fluxo completo SDD em equipe 466

Encerramento

Conclusões finais 474
Glossário 480
Catálogo de templates SDD 484
Guia prático de adoção SDD 496
Estudo comparativo: prompt isolado, SDD leve e SDD rigoroso 510
Caderno de exercícios por parte 517
Referências 522
Índice remissivo 526
Sobre o autor 528

Capítulo 1 - O que é SDD

Ponto de partida

O desenvolvimento de software depende de uma relação delicada entre intenção e implementação. Alguém precisa de um sistema, uma tela, uma API, um relatório, uma integração ou uma automação. O desenvolvedor interpreta essa necessidade, toma decisões técnicas e transforma o entendimento em código. Quando esse entendimento é incompleto, ambíguo ou informal, o código pode até compilar, mas passa a carregar suposições invisíveis.

Imagine uma solicitação aparentemente simples: “precisamos de um cadastro de clientes”. Essa frase pode significar apenas salvar nome e documento, mas também pode envolver bloqueio de duplicidade, auditoria, inativação, permissões, integração com pedidos e regras fiscais. Se essas decisões não aparecem antes da implementação, alguém as tomará no meio do caminho, muitas vezes sem perceber.

Com a chegada da Inteligência Artificial ao desenvolvimento de software, essa relação ficou ainda mais importante. Hoje é possível pedir a uma ferramenta que gere uma classe, uma tela, um serviço de API, um teste automatizado ou uma documentação inicial. A velocidade aumentou. Mas a pergunta fundamental continua a mesma: a ferramenta entendeu corretamente o que precisa ser construído?

Este livro apresenta o SDD, Specification-Driven Development, ou Desenvolvimento Orientado por Especificações, como uma resposta prática a esse cenário. A proposta não é criar mais burocracia nem transformar todo projeto em um conjunto pesado de documentos. É colocar a especificação no centro do processo, para que pessoas e ferramentas trabalhem a partir de um entendimento mais claro, verificável e rastreável.

Podemos começar com uma definição de trabalho: SDD é desenvolver software tomando a especificação como referência

principal para decidir, implementar, testar e revisar. Nesse sentido, a especificação não é apenas um registro posterior do que foi feito. Ela é o artefato que orienta o que será feito.

Ao final desta introdução, você deve conseguir explicar o que é SDD, por que ele é diferente de apenas escrever prompts melhores e como começar a pensar em especificações antes de pensar em código. Essa é a mudança central: deixar de enxergar a IA como um atalho para gerar implementação e passar a usá-la como parte de um processo técnico orientado por uma descrição verificável do que precisa ser construído.

Ao longo destas primeiras páginas, vamos construir a base conceitual do SDD, discutir por que ele se tornou especialmente relevante com IA e mostrar por que a mentalidade de "criar bons prompts" é necessária, mas insuficiente. Bons prompts ajudam. Mas, sem uma especificação clara, eles continuam apoiados em entendimento frágil.

A relação que este livro procura reorganizar pode ser vista de forma simples: entre a intenção de negócio e a implementação existe uma camada de entendimento. O SDD fortalece essa camada para que o código não seja produzido apenas a partir de impressões, conversas soltas ou prompts improvisados.

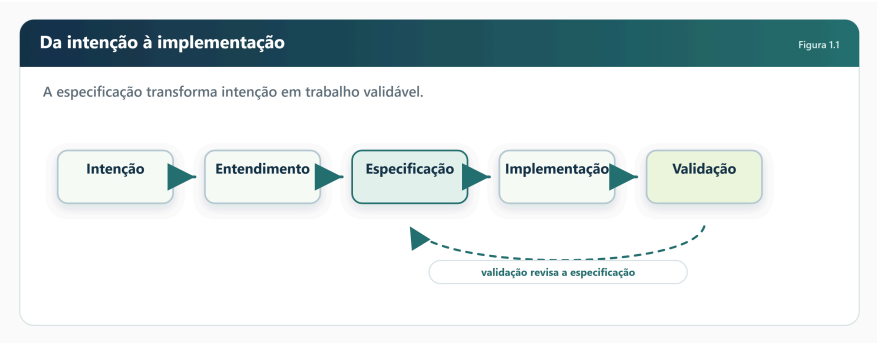

Figura 1.1 - Da intenção à implementação. A especificação organiza o entendimento antes da implementação e continua sendo revisada a partir da validação.

O leitor deste livro

Este livro foi escrito para o desenvolvedor Delphi que já conhece a pressão de transformar demandas vagas em sistemas que precisam funcionar por anos. Talvez você trabalhe com telas desktop, módulos de dados extensos, regras de negócio espalhadas entre eventos, consultas, relatórios e integrações. Talvez esteja modernizando módulos antigos, criando APIs, experimentando aplicações multiplataforma ou tentando introduzir testes em uma base que nunca foi desenhada para isso.

Alguns nomes específicos do ecossistema Delphi aparecerão ao longo do livro. Eles serão explicados quando forem realmente necessários.[1] Neste início, o mais importante não é memorizar ferramentas, bibliotecas ou componentes. É entender o problema que vem antes deles: quando a necessidade não está clara, qualquer tecnologia pode ser usada para construir a solução errada.

Também foi escrito para quem começou a usar IA no dia a dia e percebeu um padrão curioso: algumas respostas ajudam muito, outras parecem corretas, mas escondem problemas sérios. A IA pode gerar uma estrutura convincente, usar nomes plausíveis e até sugerir uma arquitetura aparentemente organizada. Ainda assim, se a solicitação inicial for vaga, a ferramenta preencherá lacunas com suposições próprias. Em desenvolvimento profissional, essas suposições precisam ser controladas.

Por fim, o livro também se dirige a arquitetos, líderes técnicos, analistas e consultores que precisam criar uma ponte entre negócio, equipe e implementação. Em projetos Delphi corporativos, muitas falhas não nascem da falta de capacidade técnica dos desenvolvedores. Elas nascem da ausência de uma descrição comum sobre o que o sistema deve fazer, quais regras precisa respeitar, quais decisões foram tomadas e como a equipe saberá que a entrega está correta.

Se você está começando, não precisa dominar todos os termos de engenharia de software antes de acompanhar a ideia. Nesta introdução, usaremos alguns conceitos em sentido bem direto: requisito é uma necessidade que o sistema deve atender; regra de negócio é uma condição do domínio que o sistema precisa

respeitar; critério de aceitação é uma forma de verificar se aquilo foi atendido; rastreabilidade é a capacidade de ligar uma necessidade ao que foi decidido, implementado e testado.

Se você trabalha com Delphi legado, a proposta também se aplica a você. SDD não começa exigindo reescrita, troca de framework ou abandono do sistema existente. Em um projeto novo, SDD ajuda a especificar antes de construir. Em um projeto legado, SDD ajuda a entender antes de modificar. Essa diferença é importante: muitas vezes, o primeiro passo não é gerar código novo, mas recuperar e registrar o comportamento que já existe.

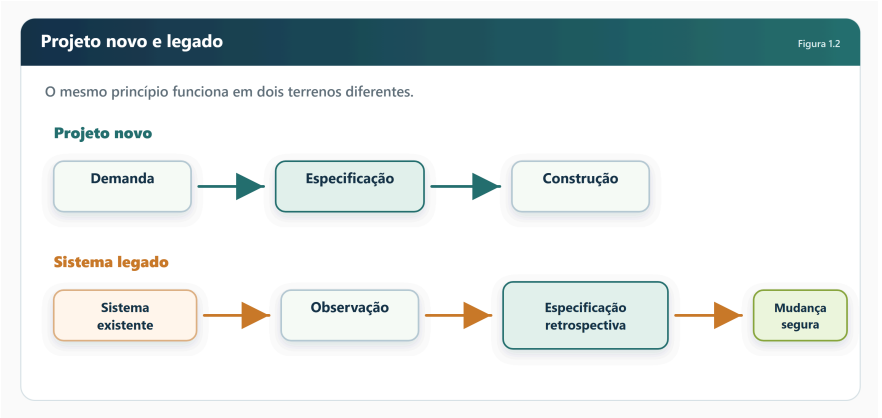

Figura 1.2 - SDD em projeto novo e em sistema legado. Em projetos novos, a especificação antecede a construção; em sistemas legados, ela ajuda a compreender o comportamento existente antes da mudança.

O ponto de partida deste livro é simples: você não precisa abandonar sua experiência Delphi para usar IA. Pelo contrário. Quanto mais clareza você tiver sobre requisitos, arquitetura, código legado, componentes, banco de dados, regras de negócio e restrições do ambiente, melhor será sua capacidade de orientar a IA e validar o que ela produz.

O caminho inicial

Antes de aprofundar a definição de SDD, precisamos entender por que ele é necessário. A tentação, quando surge uma nova abordagem, é procurar imediatamente um modelo, um template ou uma sequência de passos. Isso ajuda em algum momento, mas não deve ser o começo. Se o leitor entender apenas a forma externa do SDD, corre o risco de transformar a metodologia em mais um documento para preencher.

Por isso, começaremos pelas bases: requisitos, especificação, qualidade de software, rastreabilidade, contexto para IA e particularidades do ecossistema Delphi. A partir daí, fica mais fácil entender os ganhos concretos do SDD. A pergunta central será: se um desenvolvedor já sabe programar em Delphi e já consegue pedir ajuda a uma IA, por que mudar o processo? A resposta passa por reduzir ambiguidades, criar prompts melhores, aumentar o controle sobre código gerado, melhorar a comunicação da equipe, facilitar testes e diminuir riscos de manutenção.

Depois disso, a mudança de mentalidade fica mais evidente: sair da lógica do prompt isolado e entrar na lógica da especificação. Prompt engineering continua sendo útil, mas o prompt não é o fundamento do desenvolvimento. O prompt é apenas uma interface de comunicação com a IA. O fundamento é a especificação que descreve o comportamento esperado, as regras, as restrições e os critérios de validação.

Somente depois disso entraremos em um exemplo prático simples, baseado em uma funcionalidade de cadastro e inativação de clientes. O objetivo do exemplo não será ensinar detalhes de interface, acesso a dados ou banco de dados nesse momento, mas demonstrar como uma demanda aparentemente trivial contém perguntas que precisam ser respondidas antes da implementação.

Uma objeção natural deve aparecer desde já: isso não deixa o desenvolvimento mais lento? A impressão inicial pode ser essa, porque o SDD desloca parte do esforço para antes do código. Mas essa mudança não existe para atrasar a entrega. Ela existe porque corrigir entendimento antes da implementação costuma ser muito mais barato do que corrigir uma arquitetura, uma tela, uma API ou

uma regra espalhada pelo sistema depois que tudo já foi construído.

As bases do desenvolvimento orientado por especificações

Todo software nasce de uma intenção. Uma empresa quer controlar clientes, emitir pedidos, integrar sistemas, reduzir erros operacionais, atender a uma exigência fiscal ou disponibilizar uma API. Essa intenção inicial raramente aparece pronta para ser codificada. Ela costuma vir em conversas, mensagens, documentos incompletos, exemplos de telas antigas, planilhas, regras lembradas por usuários-chave e decisões implícitas que ninguém registrou.

A engenharia de requisitos existe para transformar essa intenção em entendimento compartilhado. Um requisito não é apenas “aquilo que o usuário pediu”. Ele precisa ser compreendido, refinado, limitado e validado. Uma frase simples como “preciso de um cadastro de clientes” pode esconder regras sobre CPF, duplicidade, dados obrigatórios, consentimento, auditoria, permissões, integração com outros módulos e retenção de histórico.

Em Delphi, essa descoberta tardia é especialmente comum em sistemas corporativos de longa vida. Muitas aplicações cresceram ao longo de anos, recebendo novas telas, novos campos, novas consultas e novas exceções. Como Delphi oferece grande produtividade para criar interfaces e conectar dados, é tentador começar pela tela, pelo módulo de acesso a dados ou pela consulta ao banco. O problema é que, quando a regra de negócio ainda não foi explicitada, a implementação inicial costuma se tornar o lugar onde decisões de negócio são tomadas sem registro.

É aqui que a especificação ganha importância. Uma especificação não deve ser entendida apenas como um texto formal colocado ao lado do projeto. Ela é um modelo do comportamento esperado do software. Esse modelo pode ser escrito em linguagem natural, tabelas de regras, critérios de aceitação, diagramas,

contratos de API, exemplos de entrada e saída, testes ou prompts versionados. O formato importa menos do que a função: permitir que pessoas e ferramentas entendam o que precisa acontecer, em quais condições, com quais dados e sob quais restrições.

Uma boa especificação responde a perguntas que o código sozinho nem sempre explica bem. O que está dentro do escopo? O que ficou fora? Que regra não pode ser violada? Qual comportamento é esperado em caso de erro? O que significa "salvar com sucesso"? Como saber se uma alteração futura quebrou uma regra existente? Essas perguntas são importantes em qualquer linguagem, mas se tornam críticas em ambientes onde sistemas precisam ser mantidos por muito tempo.

Qualidade de software também precisa ser tratada como parte dessa base. Qualidade não significa apenas "não apresentar erro". Um sistema pode compilar e ainda ser difícil de testar, inseguro, lento, frágil, acoplado ou impossível de manter. Atributos como manutenibilidade, testabilidade, segurança, desempenho, confiabilidade, usabilidade e rastreabilidade precisam aparecer cedo nas decisões. Se segurança, desempenho ou auditabilidade não aparecem na especificação, dificilmente aparecerão de forma consistente no código gerado por IA.

Outro conceito fundamental é rastreabilidade. Em um fluxo orientado por especificações, precisamos conseguir ligar uma necessidade de negócio a uma decisão técnica, um prompt, um trecho de código, um teste e uma evidência de validação. Essa ligação pode ser simples, mas precisa existir. A rastreabilidade permite entender por que uma classe foi criada, qual regra um teste cobre e qual critério de aceitação justifica determinado comportamento.

No contexto da IA generativa, essa base se torna ainda mais importante. Modelos de IA produzem respostas a partir do contexto recebido e de padrões aprendidos. Eles não têm, por padrão, compreensão garantida do negócio, da história do sistema, das restrições da equipe ou das decisões arquiteturais já tomadas. Quando a especificação é vaga, a ferramenta tende a preencher lacunas. Às vezes isso ajuda. Muitas vezes, produz uma solução plausível, mas desalinhada com o projeto.

Por isso, no desenvolvimento assistido por IA, a especificação passa a ser parte do contexto operacional da ferramenta. Ela orienta a geração, limita escolhas, explicita critérios e facilita a revisão. O desenvolvedor continua responsável por decidir, validar e manter a qualidade. A IA participa do processo, mas não substitui o julgamento técnico.

Essa discussão encontra terreno fértil no ecossistema Delphi. Delphi continua presente em sistemas administrativos, fiscais, industriais, comerciais e corporativos. Muitos desses sistemas têm anos ou décadas de evolução. Suas regras podem estar em eventos de componentes, DataModules, stored procedures, relatórios, integrações ou bibliotecas internas. A IA pode ajudar a compreender e modernizar esse cenário, mas apenas se houver método. Sem especificação, a ferramenta passa a interagir com fragmentos de código. Com especificação, ela passa a trabalhar com intenção, restrições e critérios.

Para projetos novos, essa especificação nasce antes da implementação. Para projetos legados, ela pode nascer depois, como uma especificação retrospectiva. Nesse caso, o desenvolvedor observa o comportamento atual, consulta o código, conversa com usuários, identifica regras implícitas e registra o que o sistema já faz antes de alterar sua estrutura. Esse cuidado evita refatoração cega, reduz regressões e permite usar IA como apoio para compreensão, não como mecanismo de mudança automática.

O custo da ambiguidade

A pergunta mais honesta neste ponto é: se já conseguimos desenvolver software e se a IA já consegue gerar código, por que adotar SDD? A resposta é direta: SDD não existe para tornar o desenvolvimento mais lento. Ele existe para reduzir o custo da ambiguidade.

Ambiguidade é cara. Ela faz uma tela ser refeita porque uma regra não foi considerada. Faz uma API mudar contrato depois que outro sistema já começou a consumi-la. Faz uma validação ser implementada na interface quando deveria estar no domínio. Faz a

equipe descobrir tarde demais que um campo precisa ser auditado, que uma exclusão física não deveria existir ou que determinado dado pessoal não poderia ser enviado a uma ferramenta externa.

Quando começamos pelo código, muitas dessas decisões são tomadas de forma implícita. Quando começamos por uma especificação suficiente, elas aparecem antes. Isso não significa que tudo será previsto. Software muda. Requisitos evoluem. O ponto é diminuir decisões acidentais e tornar as mudanças mais conscientes.

O primeiro ganho do SDD é, portanto, reduzir ambiguidade antes da implementação. Uma demanda como “criar cadastro de clientes” parece simples, mas não informa se CPF é obrigatório, se e-mail pode repetir, se existe inativação, se há auditoria, se a listagem deve ocultar inativos ou se a regra será compartilhada com uma API. Uma especificação inicial não precisa resolver o universo inteiro, mas deve tornar essas perguntas visíveis.

O segundo ganho é melhorar a interação com IA. Muitos desenvolvedores começam perguntando: “como faço a IA gerar esse código?”. Essa pergunta é compreensível, mas limitada. Uma pergunta melhor é: “como descrevo corretamente o comportamento que este código precisa implementar?”. Quando a especificação vem antes, o prompt deixa de ser uma tentativa isolada e passa a ser derivado de um artefato técnico. A IA recebe objetivo, contexto, restrições, formato esperado e critérios de aceitação.

Há uma diferença profunda entre pedir “crie uma tela de cadastro de clientes em Delphi” e pedir a geração de uma estrutura inicial com campos definidos, regras de validação, separação entre Form e serviço de aplicação, persistência via FireDAC, banco de dados do exemplo em SQLite e critérios de aceitação claros. O segundo pedido ainda não garante qualidade, mas reduz o espaço de suposições e facilita a revisão.

O terceiro ganho é manter controle sobre a arquitetura. Quando uma IA recebe uma tarefa genérica, ela pode escolher onde colocar regras, como nomear classes, como acessar dados e que padrões seguir. Em Delphi, isso pode resultar em regras dentro de eventos de botão, DataModules com responsabilidades excessivas,

queries acopladas à interface, ausência de testes ou tratamento frágil de erro.

O SDD permite orientar a ferramenta com limites claros: a regra deve ficar fora do Form, o serviço de aplicação deve orquestrar o caso de uso, o repositório deve esconder detalhes de persistência, e os critérios de aceitação devem ser testáveis. Mesmo que o leitor ainda não use todos esses padrões no dia a dia, a ideia principal já deve ficar clara: a IA precisa receber fronteiras técnicas, não apenas uma tarefa.

O quarto ganho é melhorar a comunicação entre pessoas. SDD não é útil apenas para IA. Ele ajuda desenvolvedores, analistas, usuários-chave, líderes técnicos, testadores e equipes de sustentação a trabalharem sobre uma referência comum. Em projetos de longa duração, isso é decisivo. A pessoa que mantém o sistema amanhã pode não ser a mesma que escreveu o código hoje.

O quinto ganho aparece nos testes. Quando critérios de aceitação são definidos cedo, os testes deixam de nascer apenas do código pronto e passam a nascer do comportamento esperado. A regra “CPF é obrigatório” pode gerar um critério de aceitação, que pode gerar um teste unitário, um teste de integração ou um roteiro de teste manual. O importante é que o teste deixa de ser uma atividade desconectada e passa a validar uma regra especificada.

O sexto ganho está na manutenção. Sistemas reais mudam. Regras fiscais mudam, integrações surgem, clientes pedem exceções, bancos são migrados e telas são adaptadas. Quando existe rastreabilidade, a equipe consegue perguntar qual requisito será afetado, que teste precisa mudar, que decisão técnica justificou o desenho atual e que comportamento existente precisa ser preservado. Isso é especialmente valioso em sistemas legados, onde muitas regras vivem apenas no código.

Por fim, SDD ajuda a trocar documentação decorativa por documentação operacional. Documentação decorativa existe, mas raramente é consultada. Documentação operacional orienta decisões, reduz dúvidas, alimenta prompts, gera testes, explica regras, registra exceções e apoia a manutenção. Se um documento não ajuda nenhuma dessas atividades, ele deve ser simplificado, removido ou transformado em algo mais útil.

Do prompt isolado à especificação

A popularização da IA levou muitos desenvolvedores a começarem pelo prompt. Isso é natural. A interação é simples, a resposta é rápida e a sensação de produtividade é imediata. Pedidos como “crie uma classe em Delphi”, “gere uma tela VCL”, “crie uma API REST” ou “refatore este código” podem produzir resultados úteis.

O limite dessa mentalidade aparece quando o prompt vira o centro do processo. O desenvolvedor passa a melhorar a frase, ajustar a instrução, pedir novamente, comparar respostas e aproveitar trechos. Isso pode funcionar para tarefas pequenas, mas é frágil quando estamos falando de sistemas corporativos, regras de negócio importantes, código legado ou decisões arquiteturais.

No SDD, o prompt continua existindo, mas muda de lugar. Ele deixa de ser o ponto de partida e passa a ser uma consequência da especificação. A ordem se torna mais disciplinada: entender o problema, especificar comportamento e restrições, revisar a especificação, transformar partes dela em prompts, gerar ou alterar código, validar o resultado e registrar decisões.

A distinção central é esta: o prompt é a interface de comunicação com a IA; a especificação é o fundamento técnico do que precisa ser comunicado. Melhorar prompts é útil, mas insuficiente. Um prompt bem escrito sobre um problema mal entendido ainda pode produzir uma solução errada.

Podemos enxergar essa evolução em três níveis. No primeiro nível, o desenvolvedor pede código para a IA e avalia a resposta depois. No segundo, ele melhora o prompt, adicionando mais detalhes, formato de saída e algumas restrições. No terceiro, ele cria uma especificação que orienta não apenas o prompt, mas também a implementação, os testes, a revisão e a manutenção. O SDD começa de fato nesse terceiro nível.

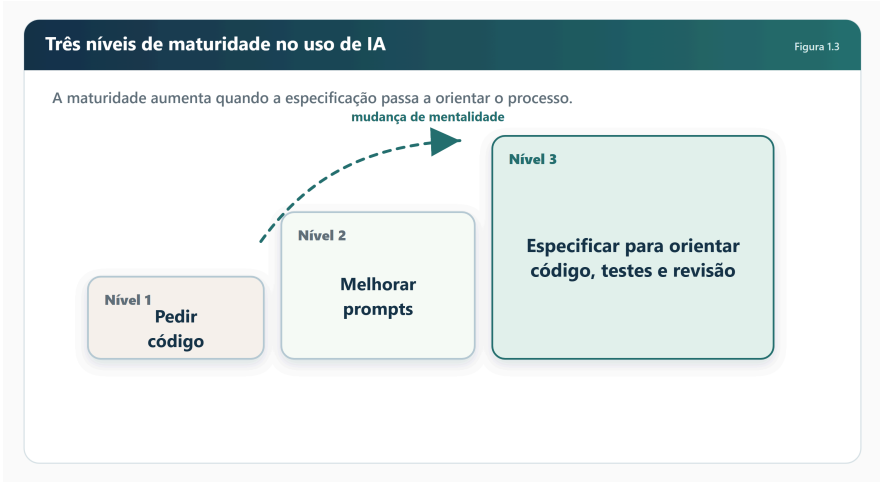

Figura 1.3 - Três níveis de maturidade no uso de IA. O SDD começa quando a especificação passa a orientar o processo, e não apenas a formulação do prompt.

Essa mudança altera a responsabilidade do desenvolvedor. Escrever código continua importante, mas deixa de ser a única habilidade central. Formular problemas, identificar ambiguidades, definir critérios de aceitação, impor restrições arquiteturais, revisar código gerado, criar testes, proteger dados e registrar decisões tornam-se habilidades ainda mais importantes. Elas sempre foram relevantes; a IA apenas tornou mais visível quando estão ausentes.

Quando uma funcionalidade simples não é simples

Imagine uma demanda comum: a área comercial pede uma rotina para inativar clientes que não devem mais receber novos pedidos. Em um fluxo tradicional, é tentador abrir o RAD Studio, ajustar um Form VCL, posicionar campos, adicionar botões, conectar uma query e implementar a gravação. A produtividade inicial parece alta. Em pouco tempo existe uma tela visível, pronta para demonstração.

Mas a simplicidade inicial esconde perguntas relevantes. CPF é obrigatório? Pode existir mais de um cliente com o mesmo e-mail? O sistema deve permitir exclusão física ou apenas inativação? Alterações precisam ser auditadas? Clientes inativos aparecem na pesquisa padrão? Há dados pessoais que exigem cuidado pela LGPD? Essa tela será a única forma de cadastro ou a mesma regra será usada por uma API? As validações pertencem ao Form, a um serviço de aplicação, ao banco de dados ou a mais de uma camada?

O problema, nesse caso, não é falta de código. É falta de especificação. A IA poderia gerar uma tela rapidamente, mas provavelmente teria que assumir respostas para essas perguntas. O desenvolvedor também poderia assumir essas respostas manualmente. Em ambos os casos, o risco é o mesmo: decisões importantes seriam tomadas antes de serem explicitadas.

O SDD propõe mudar esse ponto de partida. Antes de pedir código, criamos uma especificação inicial suficiente para orientar a implementação. Ela não precisa ser perfeita nem definitiva. Precisa ser clara o bastante para reduzir ambiguidades, permitir revisão e criar critérios de validação.

> **DICA**
>
> Antes de perguntar qual prompt usar, pergunte o que precisa estar especificado para que a resposta possa ser revisada.

O que significa desenvolver orientado por especificações

Specification-Driven Development é uma abordagem em que a especificação orienta as decisões de desenvolvimento. Isso significa que requisitos, regras de negócio, restrições técnicas, critérios de aceitação, exemplos, contratos, decisões arquiteturais e riscos são tratados como artefatos de primeira classe.

Na prática, desenvolver orientado por especificações significa que uma funcionalidade não começa apenas com “vou criar uma tela” ou “vou pedir uma classe para a IA”. Ela começa com uma descrição verificável do comportamento esperado. Essa descrição pode ser pequena no início, mas precisa ser suficiente para guiar decisões e permitir validação.

No SDD, a especificação não é um documento que acompanha o projeto depois que as decisões já foram tomadas. Ela é o principal instrumento de projeto. É a partir dela que a equipe define tarefas, cria prompts, gera código, escreve testes, revisa implementação e mantém documentação.

Uma especificação útil precisa ser clara, verificável e rastreável. Clareza significa que as pessoas envolvidas conseguem entender o comportamento esperado. Verificabilidade significa que é possível avaliar se a implementação atende ao que foi descrito. Rastreabilidade significa que uma regra pode ser ligada a decisões, código, testes e evidências.

Em projetos assistidos por IA, esses atributos ganham peso adicional. A especificação passa a servir também como contexto para a ferramenta. Ela informa o que deve ser construído, mas também o que não deve ser inventado. Ela reduz a liberdade da IA onde essa liberdade seria perigosa e aumenta a utilidade da IA onde a automação realmente ajuda.

O que SDD não pretende ser

SDD não é sinônimo de documentação extensa. Um documento longo, desatualizado e pouco consultado não representa SDD. Também não é uma volta rígida ao modelo cascata, em que tudo precisa ser definido de maneira definitiva antes de qualquer implementação. Especificações podem e devem evoluir.

SDD também não elimina prototipação, conversa com usuários ou experimentação técnica. Pelo contrário, protótipos e conversas podem ajudar a melhorar a especificação. A diferença é que o conhecimento descoberto precisa ser incorporado ao artefato que orienta o desenvolvimento.

Outro equívoco comum é imaginar que SDD significa criar um prompt enorme e pedir para a IA resolver tudo. Isso não é SDD. Um prompt grande pode continuar sendo vago, contraditório ou tecnicamente frágil. SDD exige decomposição, validação, critérios e rastreabilidade.

A ideia central é simples: SDD não defende uma especificação perfeita antes de qualquer ação. Defende uma especificação suficiente, explícita e evolutiva antes das decisões que terão custo de manutenção.

Por que a IA torna a especificação mais importante

IA aumenta a velocidade com que uma ideia pode virar código. Isso é poderoso, mas também perigoso. Quando a especificação é fraca, uma ferramenta pode produzir rapidamente uma solução baseada em premissas erradas. A velocidade não corrige a ambiguidade; ela apenas faz a ambiguidade chegar mais cedo ao código.

Considere dois pedidos. O primeiro é comum e tentador:

PROMPT

```
Crie uma tela de cadastro de clientes em Delphi.
```

O segundo ainda é um pedido para a IA, mas já nasce de uma especificação mínima:

PROMPT

```
Crie a estrutura inicial de uma tela VCL de
cadastro de clientes em Delphi,
considerando os campos Nome, CPF, E-mail e Status,
com validação de CPF
obrigatório, bloqueio de CPF duplicado, separação
entre Form e serviço de
```

```
aplicação, persistência via FireDAC e critérios de
aceitação listados abaixo.
```

O segundo pedido não é melhor apenas porque é maior. Ele é melhor porque contém decisões, restrições e critérios. Ele não entrega responsabilidade à IA de forma irrestrita. Ele orienta a ferramenta dentro de um espaço técnico mais controlado.

Esse é o ponto em que SDD e IA se encontram. A IA pode ajudar a explorar alternativas, revisar lacunas, gerar código, propor testes, documentar decisões e encontrar inconsistências. Mas ela precisa ser conduzida por especificações. Sem isso, o processo fica dependente de tentativa e erro.

Como SDD se relaciona com TDD, BDD e DDD

SDD não precisa competir com TDD, BDD ou DDD. Essas abordagens respondem a perguntas diferentes e podem se complementar.

Abordagem	Foco principal	Relação com SDD
SDD	Especificação como artefato orientador	Define o que deve ser construído e validado
TDD	Testes antes ou junto do código	Pode transformar critérios de aceitação em testes
BDD	Comportamento descrito em linguagem próxima do negócio	Pode expressar parte da especificação
DDD	Modelagem do domínio e linguagem ubíqua	Ajuda a estruturar regras e conceitos centrais
Documentação tradicional	Registro de informações do projeto	Pode ser útil, mas nem sempre orienta execução

TDD ajuda a criar código testável e orientado por verificações. BDD ajuda a expressar comportamento em uma linguagem mais próxima do negócio. DDD ajuda a compreender domínios complexos e organizar modelos conceituais. SDD pode usar elementos de todas essas abordagens, mas seu foco é anterior e transversal: criar uma especificação capaz de orientar análise, implementação, testes, IA e manutenção.

A especificação como contrato de desenvolvimento

Quando dizemos que a especificação funciona como contrato, não estamos usando a palavra em sentido jurídico. A ideia é um acordo técnico verificável. A especificação define o que será entregue, o que está fora do escopo, quais regras devem ser respeitadas, quais restrições existem e como a equipe saberá que a implementação está correta.

Esse contrato técnico não precisa ser pesado. Em muitos casos, uma especificação inicial de poucas páginas, bem estruturada, vale mais do que dezenas de páginas genéricas. O valor está na capacidade de orientar decisões e reduzir dúvidas.

Uma forma simples de pensar a rastreabilidade desse contrato é:

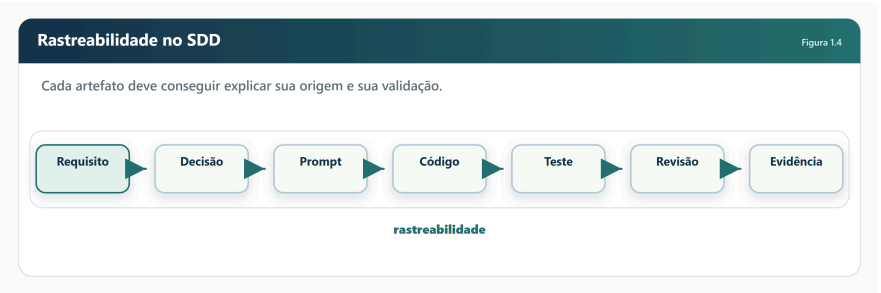

Figura 1.4 - Rastreabilidade no SDD. Um requisito deve poder ser relacionado às decisões, aos prompts, ao código, aos testes, à revisão e às evidências de validação.

Essa linha será recorrente ao longo do livro. Um requisito deve levar a decisões; decisões podem orientar prompts; prompts podem gerar ou modificar código; código precisa ser testado; testes e revisão produzem evidências. Quando essa relação existe, a equipe ganha mais controle sobre o desenvolvimento.

O ciclo SDD aplicado a Delphi

Em um projeto Delphi, o ciclo SDD pode começar com uma demanda inicial e evoluir para uma especificação revisada. A partir dela, a equipe cria tarefas técnicas e prompts, implementa em Delphi, testa, revisa e registra evidências. O ciclo não é linear de forma rígida; ele é iterativo. A validação pode revelar lacunas na especificação, e essas lacunas voltam para o início do processo.

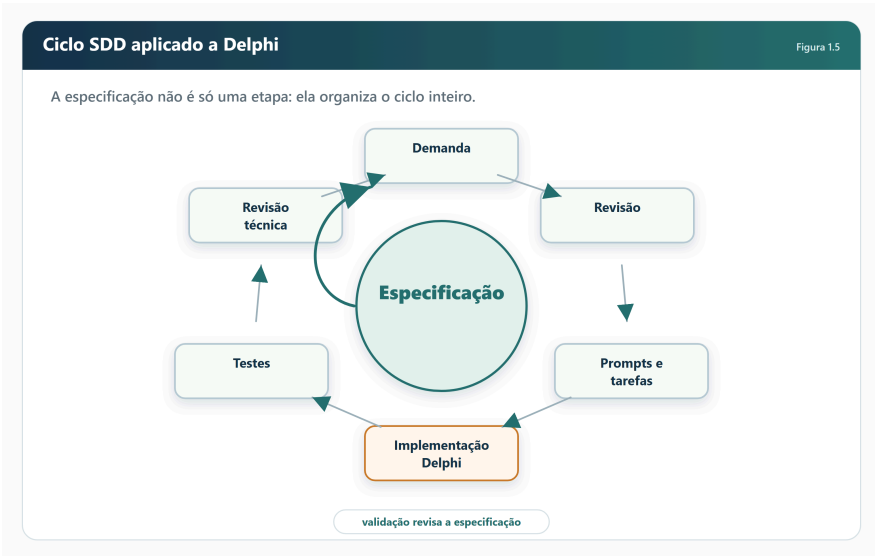

Figura 1.5 - Ciclo SDD aplicado a Delphi. A validação pode revelar lacunas na especificação, por isso o ciclo retorna ao artefato que orienta o desenvolvimento.

Em Delphi, a implementação pode envolver uma tela VCL, uma interface FireMonkey, um serviço de aplicação, uma API com DEXT, acesso a dados via FireDAC, testes com DUnitX ou scripts SQL para SQLite e Firebird. O detalhe técnico varia, mas o princípio permanece: a implementação deve responder à especificação, não a uma sequência improvisada de decisões.

Da demanda vaga à especificação inicial

Voltemos à demanda do cadastro de clientes. Em vez de começar pelo Form, podemos criar uma especificação inicial.

O objetivo seria permitir o cadastro, consulta, edição e inativação de clientes em uma aplicação Delphi VCL. O escopo incluiria nome, CPF, e-mail e status, validação de campos obrigatórios, bloqueio de CPF duplicado, listagem de clientes cadastrados e inativação sem exclusão física. Ao mesmo tempo, a especificação deveria declarar o que está fora do escopo naquele momento, como integração com Receita Federal, envio de e-mail, sincronização com API externa ou controle avançado de permissões.

As regras de negócio poderiam ser registradas em uma tabela simples:

Código	Regra
RN-001	CPF é obrigatório.
RN-002	CPF não pode ser duplicado.
RN-003	Cliente inativo não deve aparecer por padrão nas pesquisas operacionais.
RN-004	Exclusão física não será permitida.

Os critérios de aceitação tornariam essas regras verificáveis:

Código	Critério
CA-001	Ao tentar salvar cliente sem CPF, o sistema deve impedir a gravação.
CA-002	Ao tentar cadastrar CPF já existente, o sistema deve exibir erro claro.
CA-003	Ao inativar cliente, o registro deve permanecer no banco com status inativo.
CA-004	A listagem padrão deve mostrar apenas clientes ativos.

Algumas observações técnicas também já poderiam aparecer: a interface principal será VCL, a persistência usará FireDAC, o banco de dados do exemplo poderá ser SQLite, as regras não devem ficar concentradas no Form e os testes devem cobrir as validações principais. Esses detalhes não encerram a arquitetura, mas impedem que a implementação comece sem direção.

Essa ainda não é uma especificação completa. Ela não define todos os campos, mensagens, índices, estrutura de tabelas ou classes. Mesmo assim, já muda a qualidade da conversa. O desenvolvedor pode revisar regras antes de codificar. A IA pode receber contexto melhor. Os testes podem ser derivados dos critérios. E a implementação deixa de começar no escuro.

Armadilhas comuns

Um erro comum é confundir especificação com excesso de documento. Quando isso acontece, a equipe cria arquivos longos, pouco objetivos e difíceis de manter. SDD exige clareza, não volume.

Outro erro é especificar apenas tela e campos, ignorando comportamento. Em sistemas corporativos, os problemas mais importantes geralmente estão nas regras, exceções, integrações e consequências de cada operação.

Também é comum escrever prompts sem critérios de aceitação. A IA pode gerar algo visualmente adequado, mas tecnicamente desalinhado. Sem critérios, a revisão vira opinião.

Em projetos Delphi, um erro recorrente é colocar regra de negócio diretamente em eventos de componentes. Isso pode parecer produtivo no início, mas dificulta teste, reaproveitamento, manutenção e uso da mesma regra por uma API ou outro módulo.

Por fim, há o risco de tratar a especificação como algo fixo. Uma especificação deve evoluir quando o entendimento melhora. O importante é que essa evolução seja registrada e que código, testes e documentação acompanhem a mudança.

O que levamos deste capítulo

Nesta introdução, vimos que SDD não é apenas uma técnica de documentação nem um nome novo para escrever prompts mais longos. SDD é uma forma de organizar o desenvolvimento em torno de especificações claras, verificáveis e rastreáveis. A especificação passa a orientar decisões, prompts, código, testes, revisão e manutenção.

Também vimos que o valor do SDD aumenta em um cenário com IA. Quando uma ferramenta consegue gerar código rapidamente, a qualidade do resultado passa a depender ainda mais da clareza do pedido, do contexto fornecido e dos critérios usados para validar a resposta. A IA pode acelerar a implementação, mas não elimina a necessidade de compreender o problema.

A principal mudança de mentalidade é deixar de perguntar primeiro “como faço a IA gerar este código?” e passar a perguntar “qual comportamento precisa ser descrito, validado e preservado?”. Essa mudança vale tanto para projetos novos quanto para sistemas Delphi legados. Em projetos novos, SDD ajuda a

especificar antes de construir. Em sistemas legados, ajuda a entender antes de modificar.

Mais importante do que memorizar a sigla é perceber a mudança de postura. A IA não elimina a necessidade de pensar. Ela aumenta o valor de pensar com clareza. SDD é a disciplina que transforma essa clareza em artefatos utilizáveis: especificações, prompts, código, testes, revisões e decisões rastreáveis.

Para o iniciante, isso significa aprender a programar sem tratar o código como primeiro e único lugar do raciocínio. Para o desenvolvedor de Delphi legado, significa criar uma forma segura de compreender, registrar e evoluir sistemas existentes. Em ambos os casos, a mentalidade é a mesma: antes de acelerar a implementação, precisamos melhorar a definição do que deve ser implementado.

No próximo capítulo

Com essa base, agora aproximamos essa discussão da realidade específica do ecossistema Delphi. Aplicações VCL, FireMonkey, DataModules, FireDAC, APIs, integrações e sistemas legados têm características próprias que influenciam a forma como especificamos, implementamos e mantemos software.

O objetivo será entender por que especificações melhores são especialmente importantes em projetos Delphi, sobretudo em sistemas corporativos de longa duração. Antes de avançarmos para modelos de especificação, prompts e geração assistida por IA, precisamos reconhecer o terreno onde essas práticas serão aplicadas.

NOTAS DO CAPÍTULO

1. VCL é a biblioteca visual tradicional do Delphi para aplicações desktop Windows. FireMonkey é a biblioteca visual multiplataforma. DataModule é um módulo usado para organizar componentes não visuais, como conexões e consultas. FireDAC é a biblioteca de acesso a dados. RAD Studio é o ambiente de desenvolvimento da Embarcadero.

Capítulo 4 - Anatomia de uma boa especificação

Ponto de partida

No capítulo anterior, posicionamos a Inteligência Artificial no lugar correto: uma parceira técnica, útil para analisar, organizar, sugerir, gerar e revisar, mas incapaz de assumir a responsabilidade final pelo software. Essa conclusão nos leva a uma pergunta prática. Se a IA trabalha melhor quando recebe contexto, restrições e critérios, onde esse contexto deve ser organizado? A resposta, no SDD, é a especificação.

Até aqui, falamos da especificação como centro do processo. Agora precisamos abri-la por dentro. Uma especificação útil não é apenas um texto descrevendo o que alguém deseja. Também não é um documento longo preenchido por obrigação. Ela é um conjunto de informações organizadas para permitir decisão, implementação, teste, revisão e manutenção.

Essa distinção é importante porque muitos desenvolvedores já tiveram contato com documentos de requisitos que envelheceram mal. Alguns eram grandes demais para serem lidos. Outros eram genéricos demais para orientar código. Outros descreviam telas, mas não explicavam regras. Outros registravam regras, mas não traziam exceções. Em muitos projetos, o documento existia, mas o comportamento real continuava sendo descoberto no código, no banco, no usuário experiente ou na memória de alguém da equipe.

O SDD exige outro tipo de especificação. Ela precisa ser suficientemente clara para orientar uma pessoa, suficientemente estruturada para orientar uma IA e suficientemente verificável para permitir que o resultado seja avaliado. Isso não significa escrever tudo antes de fazer qualquer coisa. Significa escrever o necessário para reduzir ambiguidades perigosas antes que elas se transformem em código, retrabalho ou defeito.

Neste capítulo, vamos observar a anatomia de uma boa especificação. O objetivo não é apresentar ainda um template definitivo, nem aprofundar todos os tipos de requisito. Isso virá nos capítulos seguintes. O objetivo agora é formar uma base mental sólida: entender quais partes precisam existir, por que elas existem e como se conectam em um fluxo profissional de desenvolvimento Delphi orientado por especificações.

O leitor e a especificação como ferramenta de trabalho

Para quem está começando, especificação pode parecer uma etapa anterior ao desenvolvimento, quase separada dele. Primeiro alguém escreve o que deve ser feito; depois o desenvolvedor implementa. Na prática, essa separação costuma ser ilusória. O desenvolvedor interpreta requisitos, toma decisões, escolhe nomes, define validações, organiza dados, decide onde uma regra ficará e imagina cenários que não foram descritos. Quando a especificação é fraca, essas decisões continuam acontecendo, mas de forma invisível.

Para quem trabalha com Delphi há muitos anos, a questão aparece de outro modo. Em sistemas VCL, FireMonkey, serviços, integrações e rotinas de banco, grande parte do conhecimento costuma estar distribuída.[1] Uma validação pode estar no evento de um botão, outra em uma consulta, outra em uma trigger, outra em uma rotina antiga de importação, outra em uma planilha usada pelo suporte. O sistema funciona porque uma rede de decisões foi acumulada ao longo do tempo. O problema é que essa rede nem sempre está explícita.

Em projetos legados, especificar não significa necessariamente começar de uma folha em branco. Muitas vezes significa recuperar conhecimento que já existe, mas está escondido. A especificação passa a funcionar como um mapa: ela revela o que o sistema faz, o que deve continuar fazendo, o que pode mudar e o que precisa ser protegido.

Para líderes técnicos, arquitetos, consultores e equipes que pretendem usar IA de forma séria, a especificação tem ainda outra função. Ela cria alinhamento. Uma boa especificação permite que analistas, desenvolvedores, testadores, revisores, usuários-chave e agentes de IA trabalhem sobre uma base comum. Quando surge uma dúvida, a equipe deixa de depender apenas de opinião ou memória e passa a comparar a solução com um artefato explícito.

Por isso, neste livro, especificação não será tratada como burocracia. Ela será tratada como ferramenta de trabalho. Seu valor não está em existir, mas em orientar decisões melhores.

O caminho deste capítulo

Vamos começar por uma ideia simples: uma especificação boa não é definida pelo tamanho. Ela é definida pela capacidade de orientar e validar. A partir daí, examinaremos os elementos que normalmente precisam aparecer em uma especificação profissional: objetivo, escopo, atores, regras de negócio, fluxos, dados, restrições técnicas, exceções, critérios de aceitação e rastreabilidade. Rastreabilidade é a capacidade de ligar uma necessidade às decisões, ao código, aos testes e às evidências que demonstram o comportamento.

Cada elemento será apresentado com uma função clara. O objetivo explica por que a funcionalidade existe. O escopo define fronteiras. Os atores mostram quem usa ou é afetado pelo comportamento. As regras de negócio dizem o que não pode ser violado. Os fluxos mostram comportamento em movimento. Os dados e contratos descrevem informações de entrada, saída e persistência. As restrições técnicas aproximam a especificação da realidade do Delphi. As exceções impedem que o sistema seja desenhado apenas para o caso feliz. Os critérios de aceitação tornam o resultado verificável. A rastreabilidade conecta tudo a decisões, prompts, código, testes e evidências.

Ao final, reuniremos esses elementos em uma especificação pequena, mas utilizável. A intenção é que o leitor termine este capítulo sabendo reconhecer quando uma especificação é apenas

uma descrição vaga e quando ela começa a se tornar um artefato operacional.

Especificação boa não é especificação grande

Um erro comum é associar qualidade de especificação ao volume de texto. Documentos grandes parecem mais completos, mas podem ser pouco úteis. Um arquivo com dezenas de páginas pode não responder às perguntas essenciais: qual problema será resolvido, o que está dentro do escopo, quais regras precisam ser preservadas, quais exceções existem, que dados são envolvidos e como saberemos se a implementação ficou correta.

Também existe o erro oposto: reduzir especificação a uma frase solta. “Criar cadastro de clientes” é curto, mas deixa quase tudo em aberto. Cliente pessoa física e jurídica? CPF ou CNPJ obrigatórios? Pode excluir cliente? Existe inativação? Cliente inativo aparece em consulta? Cliente bloqueado pode gerar pedido? A tela precisa funcionar offline? A regra vale também para API? Haverá auditoria? Quem pode alterar limite de crédito?

O ponto não é escrever muito ou pouco. O ponto é escrever o suficiente para orientar decisão e validação. Uma especificação boa tende a ter cinco características.

Ela é clara porque reduz interpretações incompatíveis. Quem lê entende o problema, o comportamento esperado e as fronteiras principais.

Ela é suficiente porque contém o contexto necessário para a etapa atual. Não tenta prever todos os detalhes do futuro, mas não omite informações que mudariam a implementação.

Ela é verificável porque permite confirmar se o resultado atende ao que foi pedido. Requisitos vagos produzem validações vagas; requisitos verificáveis produzem testes, revisão e aceite mais objetivos, em linha com a engenharia de requisitos descrita pela ISO/IEC/IEEE 29148:2018 (ISO; IEC; IEEE, 2018).

Ela é rastreável porque permite ligar uma necessidade a requisitos, regras, critérios, código, testes e evidências. Isso não

precisa começar com uma ferramenta complexa. Pode começar com identificadores simples, tabelas e disciplina.

Ela é evolutiva porque aceita mudança sem perder histórico. Sistemas mudam. A especificação precisa acompanhar essa mudança, não ser congelada como uma fotografia antiga que ninguém mais confia.

Uma boa especificação, portanto, não é um romance técnico. Também não é uma lista de desejos. Ela é um instrumento de redução de ambiguidade.

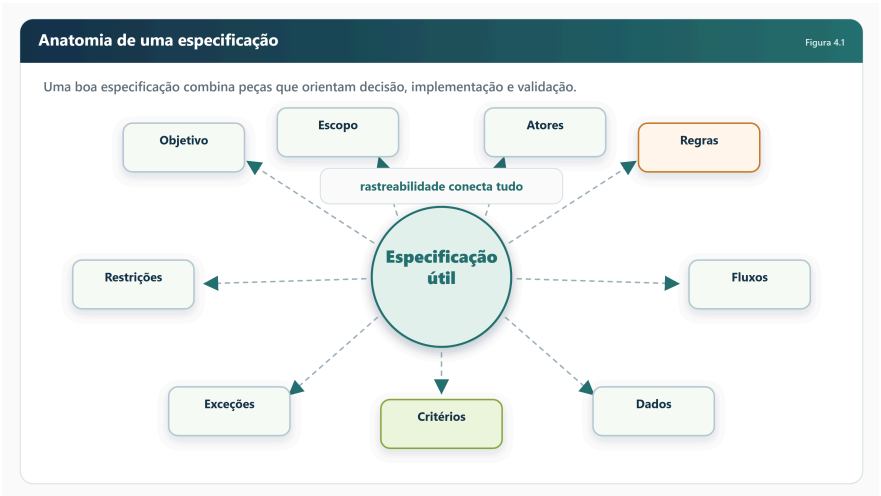

Figura 4.1 - Anatomia de uma especificação. Uma especificação útil combina intenção, fronteiras, comportamento, dados, restrições, validação e rastreabilidade.

Componentes de uma especificação

A partir daqui, vamos decompor a especificação em seus componentes principais. Cada parte cumpre uma função diferente, mas nenhuma deve ser vista isoladamente. O valor aparece quando objetivo, escopo, atores, regras, fluxos, dados, restrições, exceções, critérios e rastreabilidade trabalham juntos para reduzir ambiguidade.

Para tornar essa anatomia mais concreta, pense em cada elemento como uma resposta a uma pergunta que o desenvolvedor, o revisor ou a IA precisará fazer em algum momento. Se a especificação não responde, alguém terá que inferir. E inferência não documentada é um dos lugares onde nascem bugs, retrabalho e respostas ruins de IA.

Alguns termos técnicos aparecerão nos exemplos, especialmente quando a especificação tocar APIs, transporte de dados e persistência.[2]

Elemento	O que precisa ficar explícito	Por que isso importa para IA e desenvolvimento
Objetivo	Por que a funcionalidade existe e que problema resolve.	Evita que a IA gere uma solução tecnicamente correta para o problema errado.
Escopo	O que entra, o que fica fora, dependências e premissas.	Impede que a resposta cresça para funcionalidades não solicitadas ou ignore limites reais.
Atores	Quem usa, quem dispara, quem aprova e quem é afetado.	Ajuda a diferenciar tela, rotina automática, API, suporte, administrador e usuário final.
Regras de negócio	Condições que não podem ser violadas.	Dá à IA restrições comportamentais que devem ser preservadas em qualquer implementação.
Fluxos	Sequência principal, alternativas e pontos de decisão.	Mostra comportamento em movimento, não apenas nomes de telas ou campos.

Elemento	O que precisa ficar explícito	Por que isso importa para IA e desenvolvimento
Dados e contratos	Entradas, saídas, persistência, identificadores e formatos.	Reduz suposições sobre campos, DTOs, consultas, payloads, banco e integração.
Restrições técnicas	Plataforma, versão, bibliotecas, arquitetura existente e limites de implantação.	Evita sugestões incompatíveis com Delphi, VCL, FireDAC, banco existente ou legado.
Exceções e casos limite	O que acontece quando o caminho ideal falha.	Obriga a IA e a equipe a considerar erro, permissão, duplicidade, indisponibilidade e concorrência.
Critérios de aceitação	Como saberemos que o comportamento foi atendido.	Transforma resposta gerada em algo revisável, testável e comparável com uma expectativa.
Rastreabilidade	Como ligar necessidade, decisão, prompt, código, teste e evidência.	Permite revisar o impacto de mudanças e entender por que uma solução existe.

Essa tabela não é um template definitivo. Ela é um mapa mental. Nos capítulos seguintes, cada elemento ganhará mais profundidade. Aqui, o objetivo é deixar claro que uma especificação útil não é uma coleção de tópicos decorativos: cada parte remove uma categoria específica de dúvida.

Níveis de especificação

Uma especificação também precisa ter o tamanho certo para a decisão que pretende orientar. Nem tudo deve ser descrito no mesmo nível de detalhe. Um projeto inteiro não cabe em um prompt de método, e um método específico não deveria carregar sozinho toda a estratégia de uma aplicação.

No SDD, é útil pensar em níveis. No projeto, a especificação define visão, objetivos, princípios, restrições globais, arquitetura macro, tecnologias adotadas, limites de segurança, políticas de dados e critérios gerais de qualidade. É nesse ponto que aparecem decisões como manter uma aplicação VCL, criar uma API DEXT, preservar FireDAC, aceitar ou não novas dependências, tratar dados sensíveis ou estabelecer como a equipe usará agentes de IA.

Em seguida, o módulo aproxima a especificação da organização do sistema. Ele descreve responsabilidades, fronteiras, integrações, dados compartilhados, dependências, eventos relevantes e regras que atravessam mais de uma tela, serviço ou rotina. Em um sistema Delphi, esse recorte ajuda a evitar que uma regra fique escondida em um Form quando também deveria valer para um serviço, uma rotina automática ou uma API.

O caso de uso descreve comportamento observável. Ele informa objetivo, atores, fluxo principal, fluxos alternativos, regras, exceções, dados de entrada e saída, permissões e critérios de aceitação. Esse costuma ser o recorte mais natural para conversar com usuários, analistas, testadores e ferramentas de IA antes de escrever código.

Quando a discussão chega à classe ou ao serviço, a especificação já entra na solução técnica. Ela define responsabilidade, dependências, métodos públicos, erros

esperados, invariantes, efeitos colaterais e relação com testes. Em Delphi, pode orientar a criação de uma classe de domínio, um serviço de aplicação, um DataModule especializado, um controller DEXT ou uma unidade de validação.

Por fim, o método é o recorte mais fino. Ele só deve ser especificado com esse detalhe quando houver risco suficiente para justificar a precisão: cálculo sensível, regra fiscal, validação crítica, concorrência, transação, compatibilidade com legado ou comportamento que precisa ser preservado com cuidado. Nesse ponto, a especificação tende a registrar pré-condições, entradas, saídas, efeitos colaterais, tratamento de erro e exemplos de teste.

Essa divisão evita dois extremos. O primeiro é especificar pouco demais e obrigar a IA ou o desenvolvedor a completar lacunas importantes. O segundo é especificar tudo no menor detalhe e transformar o SDD em burocracia. A pergunta prática é: que decisão esta especificação precisa tornar mais segura? O nível certo é aquele que responde a essa pergunta sem esconder riscos nem explicar o óbvio.

Objetivo: por que isso existe

O objetivo é o ponto de partida de uma especificação. Ele responde à pergunta: por que esta funcionalidade, rotina, serviço ou alteração precisa existir?

Essa pergunta parece simples, mas evita muitos desvios. Quando o objetivo é fraco, a equipe pode construir uma solução correta do ponto de vista técnico e errada do ponto de vista do problema. “Criar cadastro de clientes” descreve uma entrega, mas não explica a intenção. Pode significar uma tela simples para manutenção interna, um cadastro compartilhado entre módulos, um serviço central para integrações ou uma revisão de um cadastro legado com regras de bloqueio e auditoria.

Um objetivo melhor seria:

> Permitir que clientes ativos sejam cadastrados, consultados, alterados e inativados
> sem exclusão física, preservando histórico para pedidos, faturamento e auditoria.

Esse objetivo ainda não resolve tudo, mas já orienta decisões. Ele indica que exclusão física não deve ser usada. Indica que histórico importa. Indica relação com pedidos e faturamento. Indica que o cadastro não é apenas uma tela, mas uma peça dentro de um contexto maior.

Em projetos Delphi, essa diferença muda a implementação. Se o objetivo fala apenas em “criar cadastro”, um desenvolvedor pode concentrar a solução em um Form VCL com botões de incluir, alterar e excluir. Se o objetivo fala em preservar histórico e atender faturamento, a solução tende a exigir atenção a status, consultas, regras de exclusão, impacto em DataModules, rotinas existentes e talvez APIs. DataModule é um módulo usado com frequência em Delphi para organizar componentes não visuais, conexões e consultas.

Um bom objetivo não precisa ser longo. Precisa ser específico o bastante para orientar a direção. Quando a equipe não consegue escrever o objetivo com clareza, geralmente ainda não entendeu o problema.

Quando a IA entra no processo, o objetivo normalmente vira a primeira orientação do prompt. Ele diz à ferramenta qual resultado deve ser buscado antes de qualquer sugestão de classe, tela, endpoint ou consulta. Sem objetivo, a IA tende a otimizar a forma da resposta, não necessariamente a intenção do sistema.

Escopo: o que entra e o que fica fora

O escopo define fronteiras. Ele diz o que será tratado agora e, com a mesma importância, o que não será tratado. Sem escopo, demandas crescem silenciosamente. Uma tela de cadastro vira integração com Receita Federal, envio de e-mail, importação de planilha, auditoria completa, controle avançado de permissões e

painel gerencial antes que alguém perceba que o projeto mudou de tamanho.

Especificar o que entra no escopo ajuda a equipe a planejar. Especificar o que fica fora ajuda a equipe a se proteger de suposições.

Em um cadastro de clientes, o escopo inicial poderia incluir cadastro, consulta, alteração e inativação. Poderia deixar fora, nesta primeira versão, integração automática com serviços externos, envio de e-mail, revisão completa de permissões e deduplicação histórica. Isso não significa que esses itens sejam irrelevantes. Significa apenas que não fazem parte da entrega atual.

Essa distinção é especialmente importante em sistemas corporativos Delphi. Muitos sistemas têm anos de evolução e módulos interdependentes. Uma alteração aparentemente pequena pode encostar em faturamento, financeiro, estoque, relatórios e integrações. Se o escopo não estiver claro, a equipe pode descobrir tarde demais que uma mudança exigia muito mais análise.

Um bom escopo também registra dependências e premissas. Se a nova rotina depende de uma tabela existente, isso deve aparecer. Se a API será usada apenas internamente na primeira fase, isso deve aparecer. Se o sistema continuará usando FireDAC e o banco atual, isso deve aparecer. FireDAC é a biblioteca de acesso a dados usada em muitos projetos Delphi. O escopo não é apenas uma lista de funcionalidades; é uma fronteira de responsabilidade.

No trabalho com IA, o escopo funciona como cerca. Ele ajuda a dizer “não implemente integração externa agora”, “não altere permissões nesta etapa” ou “não proponha mudança de banco”. Sem escopo, a IA pode sugerir uma solução maior, mais moderna ou mais genérica do que o projeto precisa.

Atores e contexto de uso

Depois de definir objetivo e escopo, a especificação precisa dizer quem interage com o comportamento descrito. Em muitos projetos, essa parte é reduzida a “usuário”. O problema é que “usuário” raramente é suficiente.

Continue a leitura

A degustação apresentou a mudança de mentalidade que sustenta o SDD e os primeiros elementos de uma especificação utilizável. No livro completo, esse percurso avança por requisitos, arquitetura, prompts, contexto, agentes, código, testes, segurança, governança e estudos de caso Delphi.

SDD em Delphi

Como especificar, projetar e desenvolver software Delphi com apoio de IA

Régys Borges da Silveira • 528 páginas • 36 capítulos

Edições física e Kindle

CONHEÇA O LIVRO

ACESSE OS EXEMPLOS

regys.com.br

ISBN da edição física: 978-65-02-29568-7