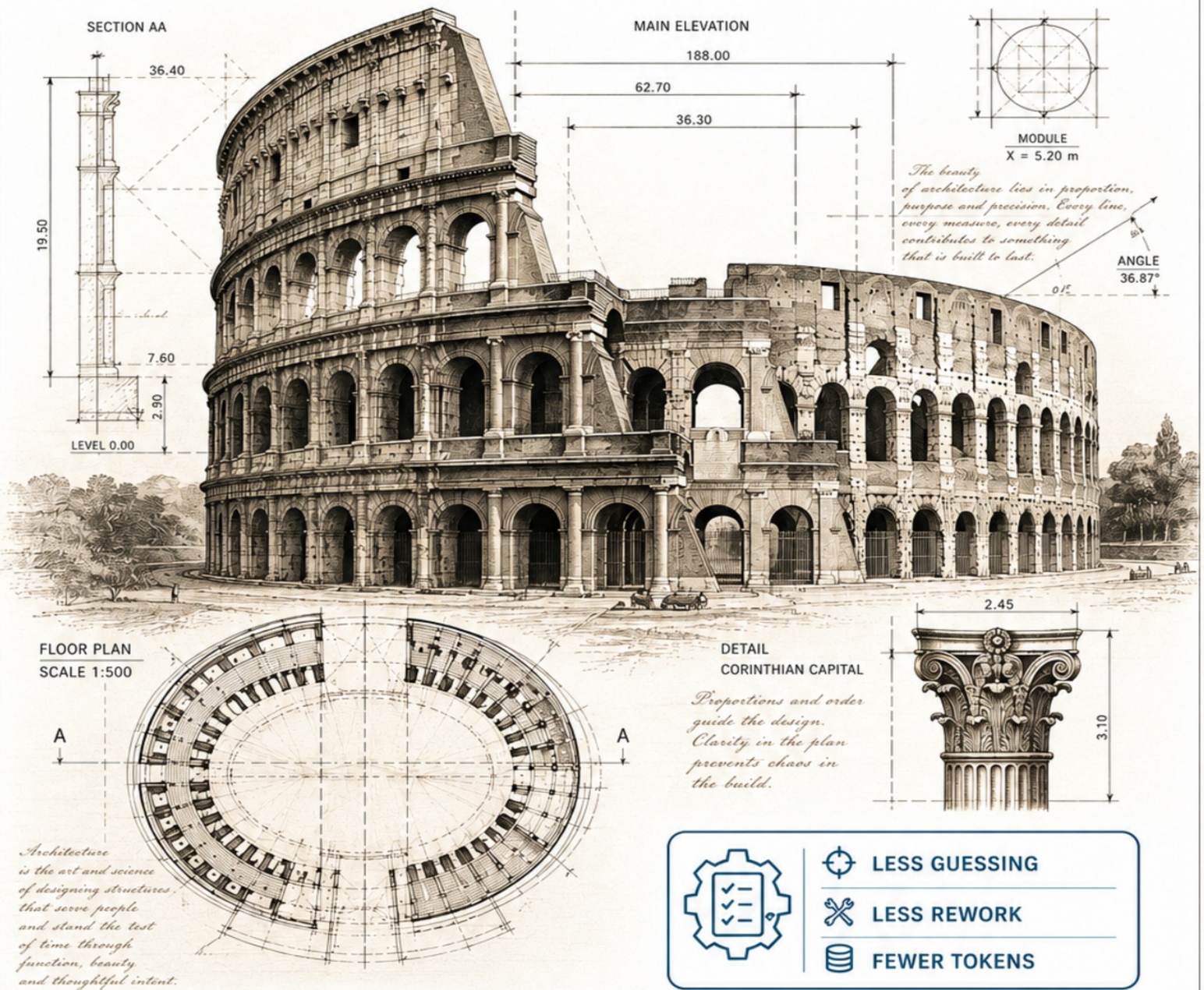


SPEC DRIVEN DEVELOPMENT

FROM VIBE CODING TO SOFTWARE ENGINEERING
BETTER SPECIFICATIONS. BETTER CODE. FEWER TOKENS.



Spec Driven Development

J.C. Ködel

Spec Driven Development

1. [About the author](#)
2. [0 - Why SDD is essential in the age of AI](#)
 1. [The thesis: knowledge and specification are leverage](#)
 2. [The three arguments: judge, decide, answer](#)
 3. [The 70% and the 30%](#)
 4. [What SDD is, in plain language](#)
 5. [SDD is not exactly new](#)
 6. [The next step](#)
3. [0b - Trilogy map: what lives in each volume](#)
4. [1 - Fundamentals: where SDD comes from](#)
 1. [Waterfall: the right instinct, the wrong cost](#)
 2. [The agile turn: learning to iterate cheaply](#)
 3. [Scrum, the pillar; XP and Kanban, the support](#)
 4. [What each method taught us](#)
 5. [SDD: the synthesis that only now became viable](#)
 6. [Why AI needs specification](#)
 7. [The cycle, now with a name](#)
 8. [The next step](#)
5. [2 - Anatomy of a specification](#)
 1. [From vague intent to a blueprint](#)
 2. [What a spec is \(and what it isn't\)](#)

3. [The why part: problem and intent](#)
4. [The scope part: in, out, and non-goals](#)
5. [The behavior part: scenarios, rules, and acceptance criteria](#)
6. [The edges: exceptions, assumptions, and dependencies](#)
7. [The whole blueprint and what makes a spec good](#)
8. [The next step](#)
6. [2b - Requirement Language: Writing What the AI Executes Without Guessing](#)
 1. [The Blueprint Is Right, the Handwriting Is Not](#)
 2. [Where Specifications Come From](#)
 3. [Before Anything: Functional or Technical](#)
 4. [EARS: The Syntax That Will Not Let the Condition Stay Implicit](#)
 5. [Given/When/Then: Behavior as a Scene](#)
 6. [Design by Contract: What Holds Before, After and Always](#)
 7. [ATDD: The Acceptance Criterion Written First](#)
 8. [Which to Use, and When](#)
 9. [What a Badly Formed Sentence Costs an Agent](#)
7. [2c - The Scope of a Spec: How Much Fits in One Specification](#)
 1. [The Question That Comes Before Writing](#)
 2. [The Criterion: A Spec Is What Fits in One Lap](#)
 3. [Completing and Reopening Fit Together; Creating and Deleting Do Not](#)
 4. [When the Feature Is Too Big](#)
 5. [When the Method Does Not Pay Off](#)
 6. [What You Take From Here](#)
8. [3 - Hands on: the SDD tools](#)

1. [From the blueprint to the wall](#)
2. [Three ways of doing the same thing](#)
3. [The cycle is the same, the incarnation changes](#)
4. [The honest comparison](#)
5. [The choice and why](#)
6. [Lean setup and first contact](#)
7. [The bridge to the full cycle](#)
9. [4 – speckit constitution: the rules before the first move](#)
 1. [The rules before the first move](#)
 2. [What a project constitution is](#)
 3. [What goes in and what stays out](#)
 4. [Running the `constitution` step on the To-Do](#)
 5. [The To-Do's constitution, principle by principle](#)
 6. [Why the stack stays out](#)
 7. [Living artifact and the bridge to the first spec](#)
0. [4.5 – Appendix: the To-Do constitution](#)
 1. [To-Do Constitution](#)
 1. [Core Principles](#)
 2. [Engineering Standards](#)
 3. [Workflow](#)
 4. [Governance](#)
 2. [5 – The complete SDD cycle](#)
 1. [Before the first `specify`, a map](#)
 2. [What you're learning is the cycle, not the app](#)
 3. [The feature lives on a branch: `main → branch → merge`](#)
 4. [The map in a single figure](#)

5. [The constitution sits above the loop](#)
6. [The loop, step by step](#)
7. [The artifacts talk to each other; that's why you clear the context](#)
8. [The map is drawn: the bridge to Ch. 6](#)
3. [6 – Create and list tasks: the first complete loop](#)
 1. [From the map to the ground: the first feature](#)
 2. [One feature at a time: the backlog](#)
 3. [Spotlight: `specify` in action](#)
 4. [The anatomy from Ch. 2, actually filled in](#)
 5. [What is spec and what is `plan`: the boundary](#)
 6. [The rest of the loop, at a follow-along pace](#)
 7. [Commit, merge, and the bridge to Ch. 7](#)
4. [6.5 – Creating and listing tasks in practice: the whole loop, file by file](#)
 1. [How to read this chapter](#)
 2. [The input: the `/speckit-specify` prompt](#)
 3. [What `specify` and `clarify` returned: `spec.md`](#)
 4. [The requirements check: `checklists/requirements.md`](#)
 5. [What `plan` decided: `plan.md`](#)
 6. [The `plan`'s design artifacts](#)
 7. [The execution list from `tasks`: `tasks.md`](#)
 8. [The code `implement` generated](#)
 9. [The portrait of the lap](#)
5. [6b – Specification Anti-Patterns: Six Ways to Get It Wrong, Over and Over](#)
 1. [Six Defects, All Taken From This App](#)

2. [1. A Rule Inherited by Reference](#)
 3. [2. Initial State Left Undeclared](#)
 4. [3. A Verb With Two Owners](#)
 5. [4. A Repeated Edge With No Answer](#)
 6. [5. Scope That Grows Out of Convenience](#)
 7. [6. Fixing at the End of the Chain](#)
 8. [The Checklist, in Six Questions](#)
6. [7 – Completing a Task: When the Obvious Hides Decisions](#)
1. [The state of git, again](#)
 2. [The feature of the moment, and `specify` at a light pace](#)
 3. ["Completing" seems obvious. It is not.](#)
 4. [The answers go back into the spec](#)
 5. [What changes down the line](#)
 6. [How to distrust the obvious](#)
 7. [End of the loop and the bridge to Ch. 8](#)
7. [7.5 – Completing and Reopening a Task in Practice: the Whole Loop, File by File](#)
1. [How to read this chapter](#)
 2. [The input: the `/speckit-specify` prompt](#)
 3. [What `specify` and `clarify` returned: `spec.md`](#)
 4. [What `plan` decided: `plan.md`](#)
 5. [The `plan` 's state model: `data-model.md`](#)
 6. [The `tasks` execution list: `tasks.md`](#)
 7. [The code `implement` generated](#)
 8. [The portrait of the lap](#)
8. [8 – Filtering Tasks: The Spotlight on `plan` and `checklist`](#)
1. [The third loop starts on `main`](#)

2. [specify and clarify](#) , at a light pace
3. [The what is already decided; the how is missing](#)
4. [Spotlight: plan decides the how](#)
5. [The constitution check has teeth](#)
6. [Spotlight: checklist as a quality gate](#)
7. [Closing the loop, at a light pace](#)
8. [The bridge to Ch. 9](#)
9. [8.5 – Filtering Tasks in Practice: the Whole Loop, File by File](#)
 1. [How to read this chapter](#)
 2. [The input: the /speckit-specify prompt](#)
 3. [What specify and clarify returned: spec.md](#)
 4. [The requirements review: checklists/requirements.md](#)
 5. [What plan decided: plan.md](#)
 6. [The plan 's view model: data-model.md](#)
 7. [The tasks execution list: tasks.md](#)
 8. [The code implement generated](#)
 9. [The portrait of the lap](#)
0. [9 – Deleting a Task: The Spotlight on tasks and analyze](#)
 1. [The fourth loop starts: 004-excluire-tarefa](#)
 2. [The early steps, at a light pace](#)
 3. [tasks : the plan turns into a list of steps](#)
 4. [Why this order: TDD and layers](#)
 5. [analyze : reading the three artifacts together](#)
 6. [A real conflict: the deletion against "completing"](#)
 7. [Fix at the source, and the cascade it triggers](#)
 8. [From the branch to the merge, and the bridge to Ch. 10](#)
1. [9.5 – Deleting a Task in Practice: the Whole Loop, File by File](#)

1. [How to Read This Chapter](#)
 2. [The Input: the `/speckit-specify` Prompt](#)
 3. [What `specify` and `clarify` Returned: `spec.md`](#)
 4. [The Completeness Check: `checklists/requirements.md`](#)
 5. [What `plan` Decided: `plan.md`](#)
 6. [The `analyze` Report: `analyze-report.md`](#)
 7. [The `plan` 's Model of the Operation: `data-model.md`](#)
 8. [The `tasks` Execution List: `tasks.md`](#)
 9. [The Code `implement` Generated](#)
 10. [The Portrait of the Lap](#)
2. [10 - Edit Task: The Spotlight on `implement`](#)
 1. [The Last Loop Starts on `main`](#)
 2. [From `specify` to `analyze` , at a Light Pace](#)
 3. [`implement` Reads the Artifacts, Not the Chat](#)
 4. [Red, Green, Refactor](#)
 5. [Reviewing What the Agent Wrote](#)
 6. [Loop Closed, App Complete](#)
3. [10.5 - Editing a Task in Practice: the Whole Loop, File by File](#)
 1. [How to Read This Chapter](#)
 2. [The Input: the `/speckit-specify` Prompt](#)
 3. [What `specify` and `clarify` Returned: `spec.md`](#)
 4. [The Completeness Check: `checklists/requirements.md`](#)
 5. [What `plan` Decided: `plan.md`](#)
 6. [The `analyze` Report: `analyze-report.md`](#)
 7. [The `plan` 's Model of the Operation: `data-model.md`](#)
 8. [The `tasks` Execution List: `tasks.md`](#)
 9. [The Code `implement` Generated](#)

10. [The Second Iteration, in the Order It Happened](#)
11. [The Portrait of the Lap, and of the Five](#)
4. [10b – Iterative Refinement: The Spec After It Already Exists](#)
 1. [No Document Is Born Finished](#)
 2. [Four Doors the Change Comes Through](#)
 3. [The Chain, in the Order Things Move](#)
 4. [What Changes and What Does Not](#)
 5. [When the Spec Stops Being Trustworthy](#)
 6. [The Document That Outlives the Project](#)
5. [11 – The Whole App Was Born From Five Loops](#)
 1. [Stop and Look Back](#)
 2. [The Whole App Is Five Merges Into](#) `main`
 3. [The Timeline in One Figure](#)
 4. [Each Feature Lit a Spotlight](#)
 5. [Together, They Covered the Whole Cycle](#)
 6. [One Constitution Governed the Five Loops](#)
 7. [The Cycle Is Yours Now](#)
 8. [What If I Did This Without SDD?](#)
 9. [The To-Do Was the Vehicle; the Process Was the Protagonist](#)
6. [Canonical Glossary](#)
 1. [Terms](#)
 2. [Rules](#)

About the author

I started programming in the 90s, writing software for video rental stores, back when renting a tape was still a business. In 1998 I built my first ERP, an integrated management system, in Visual Basic 6 with SQL Server. Real clients used it and it grew for years. It also rotted in my hands, and that experience taught me early how much it costs to build without a method.

From 2002 on I worked on systems that had no right to fail: international registries and access control at the Federal Police, international internet banking, the Basel II rollout, the risk requirements the Central Bank imposes on banks. A reusable framework I wrote back then is still in production at a large Brazilian bank almost twenty years later, without anyone having had to rewrite it.

Then came the phones. Dozens of published apps, in partnerships that included research and development projects with Microsoft. Along the way, an artificial intelligence system that analyzed 10 million calls a month for a support operation with more than 150,000 employees across 13 countries.

In 2017 I launched an app of my own, Meu Cronograma Capilar. It passed 10 million downloads, holds a 4.8 rating and has been in the category's Top 10 on the Play Store since 2018. I still take care of it alone to this day: architecture, code, tests, publishing and operation. I mention this app because it proves what no job title proves: I know how to deliver the whole cycle, alone, and sustain it for almost a decade.

That path matters here for one reason. I entered the profession when the heavy process, full of documents signed before a single line of code, was the rule. I watched agile development be born as a reaction, work, and then degenerate into ceremony. I worked under every methodology this material discusses, with the scars of someone who was there. Today I build software with AI in production every day, and this material was produced with the techniques it teaches: specification, clarification, plan, tasks, implementation. The specification tree in the repository records every step, including this text you are reading. When I claim that something works, it is because I saw it work in production or I point to whoever demonstrated it before me.

J.C.Ködel

0 - Why SDD is essential in the age of AI

You open your editor, describe in a single sentence what you need, and seconds later an artificial intelligence hands back code that compiles, runs, and even looks well made. A scene that would have been science fiction a few years ago is now routine. And it carries an uncomfortable, honest question, the one that may have brought you here: if the machine already builds this well, why would it still be worth my time to understand what is being built and to describe it carefully before asking?

It is a fair question, and AI deserves the credit: for a good share of everyday tasks, it writes quality code in seconds. But there is a more useful question than "does AI program better than I do?": what separates the people who use these tools to build solid things from the people who just paste back answers they don't understand? Whoever improvises loose requests to an AI, with no method and no clear description of what they want, is doing what is usually called *vibe-coding*: programming by feel, on a vibe, hoping the result will do. This material is the answer to that improvisation, and by the end of the chapter I hope you walk away convinced, not by me, but by yourself.

The thesis: knowledge and specification are leverage

Before any "how" we need the "why," and it fits into a single idea, the thread running through everything that follows:

Knowledge is leverage. Understanding what you want and knowing how to describe it does not compete with artificial intelligence: it multiplies what you can do with it.

A lever amplifies the strength you already have. Someone with no strength to apply lifts nothing, no matter how good the lever. It is the same with AI. It amplifies whoever hands it a clear statement of the problem and exposes whoever throws only vague phrases at it. For the person who understands what they are building and can specify, that is, say precisely what they want and why, AI is a multiplier: it delivers drafts in seconds and takes the tedium out of repetitive code. For the person who neither understands nor describes, it becomes a factory of code that looks right and nobody can judge.

Common sense says: "if AI does it, I don't need to get involved." The thesis of this material flips that: precisely because AI does it, specifying well matters more. When producing code stops being the bottleneck, the value shifts to what typing never solved on its own: knowing what to build, judging whether it is right, choosing between paths, and answering for the result.

The three arguments: judge, decide, answer

The thesis sounds nice, but it has to hold up. Here are three concrete reasons, from the most decisive to the broadest, why understanding remains the leverage even when AI does the manual labor.

First, someone has to judge what the machine produced. AI generates plausible code, and plausible is a dangerous word. Almost always what it writes is correct. The problem lives in the minority: the passage that compiles, passes the obvious test, and breaks silently in some rare case nobody thought to check, like a permission flaw where one user sees data that isn't theirs. Who spots that subtle defect? Only someone who knows what the code was supposed to do, and that comes from having specified the expected behavior beforehand. Without a clear specification in your head or on paper, judging becomes hoping the AI got it right.

Second, someone has to decide what to build and why. AI implements what you ask, but what to ask, and why that way, is still yours. Does this screen need to work without internet? Is it worth the complexity of syncing data, or does the problem not justify it? AI suggests competent options, but it doesn't carry the context of your product, your users, your budget, what will hurt to maintain two years from now. Deciding is the very act of specifying: asking for the right thing is worth more than quickly receiving the wrong one.

Third, responsibility can't be delegated. When the system goes live, the authorship is yours. If data leaks or the cloud bill explodes, there is no "the AI that wrote it." And it is impossible to answer for something you don't understand. Owning the result means being able to explain why the system is the way it is, and that only exists when there was recorded intent, a specification, rather than a pile of improvised requests.

Judge, decide, and answer: AI does none of the three for you, and specification helps you master all of them.

The 70% and the 30%

If you already use AI to program, you may recognize a scene like this. You ask for a feature, say a screen that lists items, filters by status, and updates when something changes. In seconds a huge, impressive answer comes back. The structure is there, the names make sense, much of it simply works. Those are the **70%**: the predictable work that has shown up thousands of times in thousands of similar projects. AI is extraordinary at that 70%, and it is good that it is. That is your time coming back into your pocket.

But then the **30%** begins. The list works with ten items and chokes on ten thousand. The filter ignores a status that only exists in your product. The real-time update works online and vanishes at the first dead spot in the signal. None of that 30% is about typing more code. It is about judgment: noticing what is missing, understanding why it fails, and deciding how to fix it without knocking over the rest. And there is a cruel trap: whoever can't do the 30% also can't tell it is missing. They accept the 70% as if it were 100%, ship it, and discover the hole when a user falls into it. The 70% is speed. The 30% is value, and the value lives in knowing, before you ask, what actually needs to exist.

What SDD is, in plain language

The method that captures this value has a name: **SDD**, short for *Spec-Driven Development*. The idea is simple: you describe clearly what you want, the problem, the rules, what counts as correct, before you ask for the code. The specification becomes the starting point, and the code comes afterward, to fulfill it.

Think about building a house. Nobody hands bricks to the bricklayer and says start. First comes the blueprint: where the walls go, how many rooms, where the water runs. The blueprint

is the specification; the build is the code. With the blueprint in hand, you can check whether the wall came out in the right place. Without it, you only find the mistake once the wall is already standing. SDD is drawing the blueprint before raising the building.

Why is this essential now? Because the age of AI made the build cheap and made the missing blueprint expensive. Without a specification, vibe-coding's improvisation produces two concrete problems. The first is expensive, unproductive prompts: you describe it badly, get back something crooked, and describe it again, fighting the machine more than calm thinking would have cost. The second is undecipherable code: AI delivers a lot, fast, and you pile up a system nobody understands or can maintain. The more AI produces, the more dangerous it is not to know what to ask for.

That first problem has a technical root that explains why improvising comes out expensive. AI processes text in *tokens* (pieces of words, the unit it reads, generates, and charges for) and keeps no memory of its own between one request and the next. Everything it needs to know about your task has to fit into the *context*: the window of text that comes back with each interaction. In vibe-coding, that context is the entire conversation, and it only grows. With each new prompt, the AI rereads an ever-larger history to guess what you want, burns more tokens on that rework, and loses precision as the conversation drags on. With SDD, the reference stops being the chat and becomes the specification: a short, stable document. The AI runs against that clear contract instead of reassembling your intent from a long conversation, which costs fewer tokens and produces less rework. SDD is the discipline that keeps the tiller in your hand.

SDD is not exactly new

A dose of honesty against the hype: specifying before building was not invented just now. The software industry spent decades experimenting with ways to do it, from *waterfall* (which wrote the whole specification at the start and only built afterward) to agile (which delivers in short cycles, adjusting the route at every step). Each of those schools got something right and stumbled on something, and SDD inherits the lessons of both. Chapter 1 tells that story properly; for now, it is enough to know that the missing piece for combining the best of both sides was the cost of rewriting, and it was exactly that cost that AI knocked down.

For those coming from the agile world: SDD does not replace your sprint. The specification becomes a living artifact, revised each cycle, and AI is what makes the rewrite cheap enough for that to be worth it.

The next step

If the thesis made sense, you already have the essentials: in the age of AI, the bottleneck stopped being producing code and became knowing what to ask for and judging what comes back.

One reasonable suspicion remains: isn't specifying before building the old way of making software, the one the world spent years trying to abandon? Chapter 1 answers by showing where SDD comes from, what lesson each methodology left behind, and what changed for this old idea to come back into play without the cost that used to sink it.

A confession before we go on: this material was written using SDD. Every chapter began from a specification before the first sentence, the way to keep cohesion, not forget details, and check at every step whether it still made sense. What you read is human text, written by me, grounded in facts; the specification served as scaffolding. The method here is the same one you will apply to software: the blueprint in hand before raising the wall, whether the build is a system or a text.

Ob - Trilogy map: what lives in each volume

This is the first book in a trilogy and you do not need the other two to finish it. Each volume stands on its own. This chapter exists for a practical reason: further along, the text will cite its siblings in a few passages, and it is better for you to know beforehand what lives in each one than to find out in the middle of an argument.

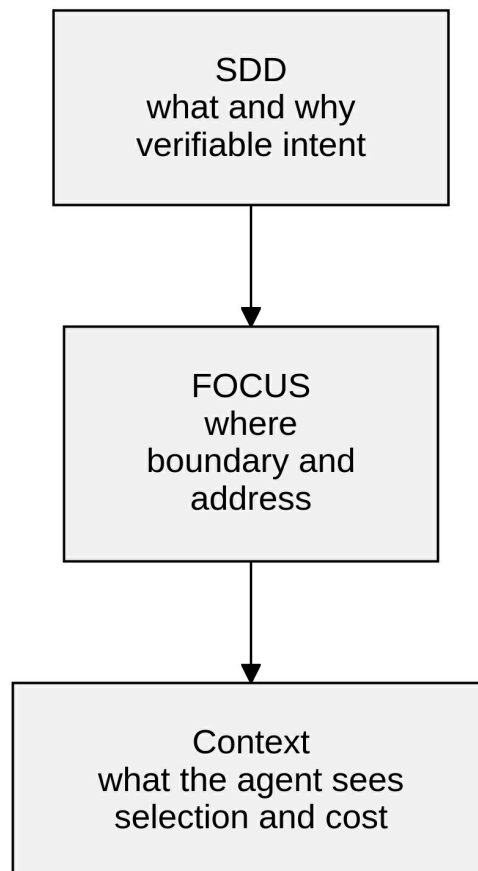
The three answer different questions about the same work.

This book answers *what* and *why*: how to turn a vague intention into a verifiable specification, so that the work, yours or an AI's, has a target to be checked against. It comes first because without it the other two have nothing to organize.

FOCUS Architecture (2026, <https://books.kodel.com.br/en/books/focus/>) answers *where*: where each rule lives and why dependencies point inward. The acronym opens up into Feature-Oriented, Clean, Unidirectional and Scalable, four adjectives for code organized by feature, with clean layers, with dependencies pointing in a single direction and with room to grow. It is the answer for when the specification is ready and you have to decide which file the thing it asks for will land in.

Context Engineering (2026, <https://books.kodel.com.br/en/books/context-engineering/>) answers *what the agent sees right now*, in the window of this one

call, and at what cost. A model does not know your project; it knows whatever fit into the conversation at that moment. Choosing what goes in there, delivering it at the right time and paying as little as possible for it is a craft of its own, and it is the subject of the third volume.



The order of the arrows is the order information travels in, and it works as a route for anyone who wants all three: the specification says what to do, the architecture says where what it asks for happens, and context carries both, in the right dose, into the model's window. None of that is binding. Whoever reads only this volume walks away with a complete method in hand.

None of the three is a prerequisite for the others, and this book was written to hold up that promise in practice. From here on, when a sibling volume shows up, it shows up as a **cross-**

reference: the name of the book, the link and whatever you need to know summarized right there, in the sentence itself. You will never have to interrupt your reading, open another book and come back. If some passage further along only closes with outside reading, that is a flaw of mine, not a design choice.

1 - Fundamentals: where SDD comes from

In Chapter 0 you walked away with one idea: specifying before you ask for the code is what separates the people who build solid things from the people who just paste back answers they don't understand. It makes sense. But you may also have walked away with an uncomfortable suspicion, and it is better to face it head-on: isn't describing everything carefully before building precisely the old way of making software, the heavy, bureaucratic one the world spent thirty years trying to abandon?

If you have ever worked on a team, you know the fatigue. A document nobody reads, a meeting to approve a meeting, a giant plan that reality runs over in the first week. Anyone who lived through that learned, rightly, to distrust whoever shows up preaching "let's plan everything up front." The suspicion is fair, and this chapter meets it head-on.

What it will do is separate two things that usually come glued together: the instinct to think before building, which has always had value, and the cost of changing late, which is what actually sank the old model. For that we need to go back in time a little and look, without rushing, at how software was made before you arrived. Not out of nostalgia: you will see that each method was born fixing the mistake of the one before it, and that SDD is the next step in that line, not a return to its beginning.

Waterfall: the right instinct, the wrong cost

Imagine building a house. Nobody hands bricks to the bricklayer and says start. First comes the blueprint: where the walls go, how many rooms, where the water runs. Only then does the structure go up, and only once it is done does the paint come. Each stage begins when the previous one ends, and going back is expensive: knocking down a wall that is already up costs far more than moving a line on the blueprint. That is the intuition behind *waterfall*: the model that specifies everything at the start and then builds in phases that flow downward in sequence, like a waterfall, never going back up.

The classic phases are these: gather requirements, design, implement, integrate, test, and maintain, one after another. This model is usually attributed to a 1970 paper by Winston Royce, "Managing the Development of Large Software Systems."¹ And here lies an irony worth knowing: Royce drew the waterfall diagram to say that it did *not* work well. He presented the pure sequence as a risky example and argued for adding back-and-forth between the phases. The industry copied the diagram he criticized and ignored the remedies he suggested.

A curiosity for history buffs: the term "waterfall" itself does not appear in Royce's paper; it caught on later, in 1970s texts that cited his work. The name that became a synonym for "the old way" was born from an incomplete reading of the author who described it.

Look at what actually happened. The waterfall instinct was right: thinking about the problem before you start building keeps you from raising the wall in the wrong place. That instinct was never the flaw. The flaw was the bet that you could get the entire specification right in one shot, at the start, and that nothing

would change afterward. Reality always changes. The client understands what they want better only once they see something finished; the market moves; a forgotten detail shows up at the end. And changing it there at the end, with the build standing on the wrong blueprint, was expensive. Estimates of the era spoke of a late fix costing dozens of times more than an early one.² The problem with waterfall, then, was never planning: it was having no way to plan again without paying a fortune.

The agile turn: learning to iterate cheaply

If changing late is expensive, the way out is not to leave the change for late. Instead of raising the whole house at once on a closed blueprint, why not put up one room first, live in it, see what bothers you, and adjust before moving on? That is the idea of *iterative and incremental* development: building in short cycles, delivering and adjusting bit by bit, rather than all at once at the end. Each cycle produces something usable, gets feedback, and corrects the route of the next cycle, while correcting is still cheap.

It may sound like a recent invention, but it isn't. There are records of iterative development back in the 1950s, and NASA's Mercury space program in the 1960s is a documented example of building and testing in small steps.³ Iterating was not born with the trend; what was missing was a name, shared values, and people willing to defend the practice against the weight of waterfall.

That name arrived in 2001. Seventeen software professionals gathered in Snowbird, Utah, and wrote the *Agile Manifesto*: a short document that set four values for developing software.⁴ Translated from the original source, they say you should value:⁵

- individuals and interactions over processes and tools;

- working software over comprehensive documentation;
- customer collaboration over contract negotiation;
- responding to change over following a plan.

Read that last line carefully, because it is the direct answer to waterfall's pain. Waterfall treated the plan as sacred and change as failure. The Manifesto flips it: change is expected, and the value is in responding to it quickly. The plan still exists; what ends is the fiction that it would be right from start to finish. *Agile* is exactly that: delivering in short cycles, with frequent feedback, adjusting the route at every step instead of betting everything on a fixed plan.

Scrum, the pillar; XP and Kanban, the support

Values need practice to become routine, and agile took shape in concrete methods. The main one, the one that most shaped how teams work to this day, is *Scrum*.

Scrum is an agile approach based on short cycles with defined roles and events and frequent feedback. The short cycle has its own name: *sprint* (a burst, a short and intense run), a fixed-length period, usually one to four weeks (two is the most common), at the end of which there is something ready to show and evaluate. Each sprint the team plans what fits in the period, works, delivers, and reviews what it did, deciding the next step based on what it learned. Instead of a single giant bet at the start, there are many small bets, each correcting the one before it. Scrum was presented publicly by Ken Schwaber and Jeff Sutherland at the 1995 OOPSLA conference.⁶ The name comes from earlier: a 1986 paper by Hirotaka Takeuchi and Ikujiro Nonaka, "The New New Product Development Game," which

compared high-performing product teams to a rugby scrum, where the team advances together, pushing in the same direction.⁷

For those already working with Scrum: it enters here only for the lesson that matters to SDD, the short cycle that makes feedback cheap. Roles like Product Owner and Scrum Master and events like the daily standup exist and are useful, but they are not the point of this chapter.

Alongside Scrum, two supporting methods added pieces that will reappear later. *Extreme programming* (XP), by Kent Beck, takes engineering good practices to the extreme. Beck developed it on Chrysler's C3 project, around 1996, and consolidated it in his 1999 work "Extreme Programming Explained."⁸ Two of its practices matter here. *Continuous integration*: merging and testing everyone's work frequently, instead of waiting for the end, so that errors show up early. And *TDD* (Test-Driven Development): writing the test before the code, so that the code is born already proving it does what it should. Both push the error close to its origin, where it is cheap to fix.

The other support is *Kanban*, formulated for software by David Anderson out of work at Corbis in the mid-2000s and described in his 2010 book, with its root in Toyota's production system.⁹ Kanban visualizes the work on a board and limits *work in progress*, or WIP: what has been started and not yet finished. Limiting WIP avoids the habit of starting a lot and finishing little; the team focuses on completing before pulling the next item. The work runs in a continuous flow, without batches.

What each method taught us

Look at the whole sequence at once and a pattern appears: a chain, each link answering the weakness of the one before it. From it come three lessons that go straight into what follows.

The first came from waterfall: **specifying gives direction**. Thinking about the problem before building keeps you from building the wrong thing competently. That instinct was right and still holds.

The second came from agile: **iterating gives adaptation**. Since reality changes, building in short cycles lets you adjust the route before the deviation gets expensive. It was the direct answer to waterfall's blind spot, which treated change as an accident instead of a rule.

The third came from Scrum, XP, and Kanban together: **early feedback reduces risk**. Short sprints, testing before coding, integrating constantly, limiting work in progress. Everything points in the same direction: finding out what is wrong as soon as possible, while the fix is still cheap. Each method refined the previous one on this point, shortening the distance between making a mistake and noticing it.

Direction, adaptation, and controlled risk. Hold on to the three. SDD doesn't pick one and discard the others; it tries to keep all three at the same time, and the rest of the chapter is about how that stopped being a dream.

SDD: the synthesis that only now became viable

Why didn't anyone simply combine the two strengths before? Why not specify with waterfall's clarity and still iterate cheaply like agile? The answer is that a piece was missing, and the piece was the cost of rewriting.

Think about the house again. If changing the blueprint meant knocking down finished walls, you would hold on to the blueprint tooth and nail and avoid touching it, exactly waterfall's reflex. If raising and knocking down walls were instant and nearly free, you would experiment freely, adjust at every visit, and the blueprint would become a living document instead of a sentence. What separated those two worlds was always the price of change. Agile lowered that price with short cycles and team discipline, but rewriting real software was still slow and expensive, done by hand, line by line.

This is where the age of AI changes the equation. When producing and redoing code stops being the bottleneck, the cost of rewriting plummets. A specification that changed can be run again in minutes, not weeks. And that unlocks the combination that didn't add up before: specifying first, with the direction waterfall taught, and still iterating cheaply, with the adaptation agile taught. **Specifying before building is an idea proven over decades; AI gives it new meaning rather than resurrecting it,** handing the old discipline of thinking first a cost of change it never had.

Let me be blunt so there is no doubt: **this is not going back to waterfall.** Waterfall froze the specification and punished anyone who changed their mind. SDD does the opposite: it treats the specification as a living artifact, made precisely to change and be run again as many times as needed. The blueprint is still valuable, but it stopped being a prison.

Why AI needs specification

It is worth closing the case for the why, now with AI at the center. Chapter 0 showed the mechanics: without a specification, the AI's context is the entire conversation, which only grows and

gets expensive; with one, the reference is a short, stable document, which the AI runs against directly.

There is also a second economy, less obvious and more lasting: the specification becomes the project's memory. The decisions, the rules, and the reasoning behind each choice are recorded in one place, which survives the end of the conversation, the change of whoever is at the keyboard, and the forgetting of six months from now. A chat with the AI evaporates; a specification stays, and it is from there that the next cycle starts. How much of that material fits into each call, and at what price, belongs to *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering/>), the third volume in this trilogy, which answers what the agent sees right now, in the window of this one call, and at what cost. You do not need it to follow along here: the rule Chapter 0 already gave is enough, that the model keeps no memory between one request and the next. That is where the edge of writing before talking comes from.

The cycle, now with a name

This synthesis has a working rhythm, and it is organized into four steps that will reappear from the beginning to the end of the material. It is worth fixing the vocabulary now, still with no tool in front of us:

- **specify:** describe clearly what you want, the problem, and what counts as correct, before asking for the code.
- **plan:** decide how it will be built, the approach and the technical decisions that hold up the specification.
- **tasks:** break the plan into concrete, executable steps, small enough to keep track of.

- **implement:** build, now with direction, fulfilling the specification and the plan.

It is the same logic as the three lessons, now in a working sequence: specifying gives direction, the plan and the tasks keep the adaptation organized, and implementing in short steps brings feedback early. There are tools that give shape to this cycle and handle the mechanical part of each step, and the material gets to them later. For now, what matters is recognizing the vocabulary: when you read specify, plan, tasks, and implement in the coming chapters, these are the four steps.

The next step

You now know where SDD comes from and why it makes sense now. What is left is to see up close the central piece of all this: the specification itself. In the next chapter we open the blueprint and examine the parts of a good specification, what it needs to contain to guide the build and what makes it clear enough for the machine and for you. From intent to the document that truly guides what will be built.

Footnotes

"Waterfall model", Wikipedia, includes the attribution to Winston W. Royce, "Managing the Development of Large Software Systems" (1970), the origin of the term, and the growing cost of late fixes: https://en.wikipedia.org/wiki/Waterfall_model

"Waterfall model", Wikipedia, includes the attribution to Winston W. Royce, "Managing the Development of Large Software Systems" (1970), the origin of the term, and the growing cost of late fixes: https://en.wikipedia.org/wiki/Waterfall_model

"Iterative and incremental development", Wikipedia, on the roots of iterative development predating 2001 (records since the 1950s and NASA's Mercury program): https://en.wikipedia.org/wiki/Iterative_and_incremental_development

"Agile software development", Wikipedia, on the writing of the Agile Manifesto in 2001 in Snowbird, Utah, by seventeen signatories: https://en.wikipedia.org/wiki/Agile_software_development

Values cited from the primary source, "Manifesto for Agile Software Development" (2001): <https://agilemanifesto.org>

"Scrum (software development)", Wikipedia, on Ken Schwaber and Jeff Sutherland, the 1995 OOPSLA presentation, and the concept of the sprint: [https://en.wikipedia.org/wiki/Scrum_\(software_development\)](https://en.wikipedia.org/wiki/Scrum_(software_development))

Hirotaka Takeuchi and Ikujiro Nonaka, "The New New Product Development Game", Harvard Business Review (1986), origin of the "scrum" metaphor: <https://hbr.org/1986/01/the-new-new-product-development-game>

"Extreme programming", Wikipedia, on Kent Beck, Chrysler's C3 project (around 1996), the 1999 work "Extreme Programming Explained", and practices like TDD and continuous integration: https://en.wikipedia.org/wiki/Extreme_programming

"Kanban (development)", Wikipedia, on David J. Anderson, the work at Corbis (mid-2000s), the 2010 book, the root in the Toyota Production System, and the work-in-progress (WIP) limit: [https://en.wikipedia.org/wiki/Kanban_\(development\)](https://en.wikipedia.org/wiki/Kanban_(development))

2 - Anatomy of a specification

From vague intent to a blueprint

At the end of the previous chapter a promise was left: to open the blueprint and examine the parts of a good specification, from intent to the document that guides what will be built. Time to keep it.

Start at the start of almost every project, a loose sentence. "I want a to-do list app to organize what I have to do." You have probably said something like it about an idea of your own. It is an honest starting point, but that is all it is, a starting point. Notice how much it leaves open. Whose tasks? One person or a team? What does "organize" mean? What does the app do when you finish a task, when the list is empty, when you type only spaces? The intent exists, but the blueprint doesn't yet. Hand that sentence to a bricklayer, human or AI, and they will fill the gaps on their own, guessing. Some guesses will please you; others will cost you rework.

This chapter is about what turns that loose sentence into a blueprint that truly guides. The question driving everything from here on is simple: what needs to be in a specification for it to work? We will answer part by part, using that same to-do list as an example that grows with each section, until the raw intent becomes a document anyone, person or machine, can follow without guessing.

What a spec is (and what it isn't)

Before listing the parts, we have to fix what a specification is, because most mistakes start here. I will use the short name that already appeared in the previous chapter: *spec*, the specification of what you want.

The first rule fits in a few words: **a spec describes the what and the why, not the how**. What the system does and why it matters go in the spec. How it does it, which language, which database, which architecture, that is another step, *plan*, the stage of the specify → plan → tasks → implement cycle where the technical approach is decided. When you write "a completed task leaves the pending list," you are describing behavior, and that is spec. When you write "store the tasks in a PostgreSQL database," you are deciding implementation, and that is plan. That is the boundary, and it is worth repeating because it is easy to cross without noticing: implementation decisions do not belong in the spec. Once it is ready, somebody still has to decide which file each rule will live in, and that belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it here: it is enough to know that the decision exists, that it comes later, and that pulling it forward is exactly the mistake this chapter wants to spare you.

This holds even for the choice of technology, and the point is important. The technology is your choice, declared in the plan, not in the spec. If at some point I say the to-do list is a web app built in React with TypeScript, treat that as an example, not a requirement. Swap in any other stack and nothing the spec describes changes, because the spec talks about the problem, not the tool that solves it.

The second rule answers a common fear. Anyone who associates "specifying" with the weight of *waterfall* fears they are signing a contract carved in stone. **A spec is a living artifact:** a document made to be revised and run again cheaply, nothing frozen about it. It is the previous chapter's thesis made flesh: because AI knocked down the cost of rewriting, changing the blueprint stopped being expensive, and the spec can change as many times as reality demands.

The third rule aims at the right target: the spec seeks testable clarity before volume. The best spec is rarely the longest; it is the one that reduces *ambiguity*, that is, reduces the passages that allow more than one reasonable reading. It says enough to guide and to stop guessing, without becoming paperwork nobody reads.

One phrase that will come back still needs explaining: the spec has to be **machine-readable**. There is nothing esoteric about it. The AI reads your spec as *context*, the text it receives in order to act, and it acts from what is written there. Where the text is clear, it executes; where it is ambiguous, it guesses. And guessing costs: it creates rework when the guess misses and burns *tokens* rereading and redoing. A machine-readable spec is just a spec with no holes for the guess to slip through. The same text that removes a person's doubt removes the AI's doubt.

One clarification is worth making, because it undoes a common misunderstanding: ambiguity does not force the AI to guess in silence. In SDD, a well-guided AI does what any serious professional would do, it asks. Faced with a passage that allows two readings, it can stop and hand the doubt back to you ("does the completed task disappear from the list or just change color?") before writing a single line. Think about how this would happen with people. If the AI were a human developer running a project in waterfall or in Scrum, they would not make up what you

meant; they would raise their hand in the meeting, send the message, close the gap by talking, because they know that building on a wrong assumption is expensive. The AI is capable of the same gesture, and the spec is where those answers get recorded instead of getting lost in the chat. So treat every question it asks as a gift: it is an ambiguity showing up early, while fixing it is still cheap, and not late, after it has become wrong code.

The why part: problem and intent

Now the parts, one by one. The first is the **why** part: the problem and the intent. What problem this solution solves, for whom, and why it matters. It seems obvious to the point of skipping, and it is exactly what gets skipped most. Without the why, everything else loses direction: you have no way to decide what goes in and what stays out, nor how to judge whether a choice is good, because you don't know what you are choosing in favor of.

Filling it in with the to-do list: the problem is that a person forgets tasks scattered across notes and in their head, and wants a single place to record what they need to do, see what is left, and check off what they finished. For whom: a person organizing their own tasks, alone, in what we will call single-user use. Why it matters: to reduce forgetting and the sense of overload. Three lines, and the loose intent from the start already has a north. Every decision from here on will measure itself against this why: does it serve one person organizing their own tasks? Then it makes sense. Doesn't serve it? Then it is probably scope too much.

This is the moment to name a word that will show up constantly: *requirement*. A requirement is a testable statement of what the solution needs to do or respect. The why itself is not a

requirement; it is the ground the requirements rest on.

The scope part: in, out, and non-goals

With the why fixed, the second part draws the boundary: the **scope**. *Scope* is the boundary of what goes in and what stays out of a solution. It has three compartments, and the third is the one most people forget.

In: what the solution does in this version. In the to-do list, that is creating a task, marking it done, editing the text, deleting, filtering by status (all, to do, done), and setting an optional due date.

Out: what is left for later. Here, accounts and login, sharing between people, notification reminders, attachments, and subtasks. None of it is forbidden forever; it just isn't in this version.

That "this version" has a name, and it is one of the most useful concepts in all of software building: the *MVP* (minimum viable product). The MVP is the smallest version of the solution that already solves the core problem end to end and can go into someone's hands. Notice the word carrying the weight: *viable*. The lean version truly works for the why you fixed, without what isn't essential yet; crippled is something else. That is why "Out" is a strategic decision, with no taste of defeat: you push to later everything that isn't needed for the first version to be worth it, precisely so you can ship, see it working, and learn from real use before investing in the rest. In the to-do list, the MVP is recording, seeing what is left, and completing tasks; login, attachments, and sharing stay out not because they are bad, but because the first version already delivers value without them.

Cutting scope early is what makes software come into existence; wanting everything in the first version is like waterfall's old trap, the giant bet that takes forever to prove whether it is any good.

And the third compartment, the decisive one: the **non-goals**. A *non-goal* is something you declare explicitly outside the target, on purpose. It differs from "out for now": it means "this is not what we are building." In the to-do list: it is not a project manager, it has no collaboration between multiple users, and it does not promise to sync across devices.

Why name what you are not going to do? Because that is how you contain the AI. Remember that it fills silence with guessing. If the spec doesn't say that multi-user collaboration is out, a well-meaning assistant might decide that "to-do list" calls for sharing and hand you accounts, permissions, and invitations you never asked for. The non-goal closes that door before it opens. Declaring what stays out is worth as much as declaring what stays in.

The behavior part: scenarios, rules, and acceptance criteria

The third part is the **behavior**: what the system does, described in two ways that complete each other, scenarios and rules.

A *scenario*, also called a *user story*, is a short description of a use situation, from the point of view of whoever uses it: what the person does and what happens in response. In the to-do list: "when creating a task with filled-in text, it appears at the top of the to-do list"; "when completing a task, it leaves the to-do view and starts counting as done." They are stories of what happens, in the language of whoever uses it, without a word about how it is built inside.

The rules are the constraints that always hold, underneath the scenarios: "the task text can't be empty or only spaces"; "the due date, when given, can't be in the past at the moment of creation." Scenarios tell what happens on the happy path; rules say what always holds, including when someone tries to step out of line.

But scenario and rule still leave a gap, and this is where the most important piece of this part comes in: the **acceptance criterion**. The *acceptance criterion* is what counts as done and correct, the verifiable condition that decides whether a requirement was met. It is the testable heart of the spec. Notice the difference: the scenario is the story (what happens); the acceptance criterion is how you know, beyond argument, that the story happened correctly.

An example makes the distinction concrete. Imagine the rule "the app must be fast." It sounds good and is useless, because nobody can say objectively whether it was met. Fast how much? Measured how? Turn it into an acceptance criterion and it becomes verifiable: "opening the list with a hundred tasks shows the first screen in under a second." Now it can be tested, and the answer is yes or no, with no opinion in the middle.

The to-do list's acceptance criteria follow the same pattern: "creating an empty task is refused, with a clear message"; "completing a task removes it from the pending count"; "the done filter shows only completed tasks." Each can be checked by anyone, without ambiguity, and it is exactly that quality, being measurable, that lives inside the acceptance criterion: an attribute of a well-written criterion, not a separate section of the spec.

The edges: exceptions, assumptions, and dependencies

What is left is the part that separates a naive spec from a robust one: the **edges**. They are three things the happy path tends to ignore: edge cases, assumptions, and dependencies.

An *edge case* is a rare or extreme situation, of emptiness, limit, or error, that the solution still has to handle well. In the to-do list, it is the empty list on first use (what does the screen show when there is nothing?), text with only spaces, a due date typed in the past, the attempt to delete a task that was already deleted, the list that grew long enough to become too long. None of these is the common use. That is why they are forgotten, and it is when they happen that they break everything. A spec that names its edges is a spec that decided, ahead of time, what to do when life goes off script.

An *assumption* is something you take as true without guaranteeing it, and which, if it changes, changes the solution. In the to-do list, the assumptions are concrete: a single user, on the same device; no need for an account in this version; the data is stored locally on the device. Writing this down keeps someone from later building on an assumption nobody agreed to.

The dependencies are the third item: what the solution depends on externally and does not control. It is worth naming the category even when it is empty, and that is the case here: in this version, the to-do list has no external dependency, and recording that is already useful information. In a future version, with the data stored on a server, dependencies would appear, and they would go here. Don't force a dependency just to fill the section; record the truth, including when the truth is "none."

The whole blueprint and what makes a spec good

Now you can see the whole blueprint at once. Gather the parts in the order they usually appear in a specification document:

1. Problem and intent (the why)
2. Scope: in, out, and non-goals
3. Behavior: scenarios and rules
4. Acceptance criteria (what counts as done and correct)
5. Edges: exception cases, assumptions, and dependencies

This is the skeleton of a spec: a document structure you can write in an ordinary text editor, with no code or tool. The names may vary from one place to another, but the anatomy is this, and it is what you will recognize later, when the material reaches the tools that give shape to this document.

With the blueprint in view, the attributes of a good spec boil down to one line: **clear** (each passage allows a single reading), **testable and measurable** (the acceptance criterion decides with a yes or a no), and **readable by human and by machine**. Each attribute came from a part you just saw; none of them depends on size. Repeat it like a motto: the spec's job is to **reduce ambiguity**, and no amount of volume replaces that.

And it is with these attributes that you gain what the chapter promised, the ability to look at a loose intent and say what is missing. Go back to the opening sentence, "I want a to-do list app." Now you don't just see an idea; you see the holes. The why is missing (solve what, for whom?), the scope is missing (what stays out?), the non-goals are missing, the acceptance criteria that would let you test are missing. You don't need the finished document to diagnose; you just pass the intent through the anatomy and mark what is blank.

For those already working with software: you must have noticed that performance, accessibility, and security barely showed up here. They exist and have a name, the non-functional requirements, and they describe not what the system does but how well it does it (fast, accessible, secure). In a spec they usually live alongside the rules and the acceptance criteria ("the first screen loads in under a second" is one of them); for this chapter's anatomy, it is enough to know they exist and where they fit, and treating them in depth is left for another time.

The next step

The blueprint is drawn, from the loose intent at the start to the document with why, scope, behavior, criteria, and edges.

One thing is missing, and it doesn't fit on the blueprint. Knowing what to build is not the same as knowing how to get it off the paper. The spec describes the destination carefully; the next step of the cycle is exactly leaving the blueprint for the build, deciding the how and getting your hands dirty. That is where we go next, putting our hands on the first tool in practice, now that the blueprint is ready to guide the way.

2b - Requirement Language: Writing What the AI Executes Without Guessing

The Blueprint Is Right, the Handwriting Is Not

The previous chapter showed what parts a specification has: the problem, the scope, the scenarios, the rules, the acceptance criteria, the edges. You know how to assemble the document. What is missing is the layer underneath, the one that decides whether each sentence inside it gets executed the way you meant or interpreted however it landed.

Two specs can have exactly the same sections filled in and yield different code. The difference is not in the structure, it is in the shape of each sentence. "The system must validate the title" has a subject, a verb and an object, fits in the functional requirements section, and says almost nothing: validate against what, when, and what happens when validation refuses. Compare it with what the To-Do's create-task spec actually says:

FR-002: The system **MUST** refuse creation when the title is blank (empty or made only of spaces), returning an explicit error result that identifies the required title as the cause, without creating the task.

Same section, same part of the blueprint. The second sentence has a trigger ("when the title is blank"), has the thing to do ("refuse creation"), has the shape of the response ("explicit error result") and has the negative guarantee ("without creating the task"). None of those four is optional for whoever implements it, and an agent handed the first version will invent all four.

That is what this chapter is about: the shapes requirements engineering invented so that a sentence cannot be read two ways. There are four, each born from a different problem, and none of them was created to talk to an AI. All of them were created because humans were already reading requirements in incompatible ways long before agents existed. What changed is that the cost of the misunderstanding now falls on an executor that never asks for clarification on its own.

Where Specifications Come From

A paragraph of history is worth it here, because it explains why the four notations are so different from one another.

The specification was born big. In the 1970s and 1980s, the requirements document was a single volume, written before any code, and the standard that formalized it, IEEE 830, went as far as fixing a recommended table of contents with dozens of sections.^{[1](#)} It was the waterfall of Chapter 1, put on paper: a long document, approved by signature, and a project that only started afterwards. The problem was not the rigor, it was the size of the loop. When the first screen appeared, two years later, half the requirements had aged out.

The agile reaction shrank the document until it nearly vanished. The user story on a card, with the conversation as the complement, was the answer to the thousand-page volume. It

worked for pace, and it opened another hole: a card that says "as a user, I want to filter my tasks" cannot be verified. The ambiguity the giant volume hid by excess, the card hid by absence.

The four notations in this chapter are attempts at the middle ground: the precision of a standard in the size of a card. They came from distinct traditions, aerospace, automated testing, programming language design, and that is why they serve distinct purposes. None replaces the others, and the To-Do's spec uses three of the four without anyone ever announcing it.

Before Anything: Functional or Technical

There is a division that comes before any notation, and it decides where a sentence lives before it decides how the sentence is written.

A **functional specification** describes observable behavior: what the system does, for whom, under what condition, and how you know it worked. It is written in the vocabulary of the problem. If you swap React for Vue, or `localStorage` for a remote database, the functional specification stays true word for word.

A **technical specification** describes construction: what layers exist, what interface each one exposes, what data structure holds the operation up, where state lives. It is written in the vocabulary of the solution, and it dies along with the stack choice.

Chapter 2 already fixed the boundary between spec and plan; this is the same boundary seen from the sentence. What matters here is the practical test, which works well when you are mid-draft and cannot tell whether that paragraph belongs in the spec: **swap the technology in your head and reread**. If the sentence still makes sense, it is functional. If it turns to nonsense, it is technical, and its place is the plan.

Apply it to the To-Do. "Created tasks are still there after the app is closed and reopened" survives swapping anything. "Tasks are written to the browser's `localStorage`" does not survive even the decision to build a phone app. Both sentences are true about the same system, and only the first is a requirement. The second is a plan decision, and the first feature's `plan.md` records it exactly that way, with the note that the domain knows nothing about `localStorage`.

Confusing the two is the origin of half the bad specs in existence. A spec that opens by saying "create a POST /tasks endpoint that writes to the `tasks` table" left nothing for `plan` to decide, and tied the feature to an architecture before anyone asked whether it was the right one.

EARS: The Syntax That Will Not Let the Condition Stay Implicit

EARS, the Easy Approach to Requirements Syntax, was born in aeronautical engineering, at Rolls-Royce, and was presented in 2009 at a requirements engineering conference.² The problem it solved was the opposite of ours: jet engine requirements, written by dozens of people, reviewed by auditors, and one of them misunderstood cost certification. The solution was to restrict the grammar. Not the vocabulary, the grammar: every requirement has to fit one of five shapes.

Ubiquitous, for what always holds, with no trigger:

The system **MUST** preserve the creation order of tasks.

Event-driven, opened by *when*, for what happens in response to something:

When the person creates a task with a filled-in, unique title, the system **MUST** register it and start showing it in the list.

State-driven, opened by *while*, for what holds during a continuous condition:

While the active view is "open", the system **MUST** show only open tasks.

Unwanted behavior, opened by *if*, for what the system does when something goes wrong:

If the title provided is blank, **then** the system **MUST** refuse creation and return an error that identifies the required title as the cause.

Optional, opened by *where*, for what only holds in a configuration or variant:

Where local storage is unavailable, the system **MUST** ...

The fifth shape is the one that shows up least in a small project, and the To-Do has no requirement of that kind. The first four cover everything the five features needed.

What the restriction buys is one single thing, and it is a big one: **the triggering condition is never left implicit**. A requirement that is not ubiquitous has, mandatorily, a clause opening the sentence that says when it holds. You cannot write "the system validates the title" and move on, because that sentence is none of

the five shapes: either it is ubiquitous, and so it validates always, including while listing, which is false; or it has a trigger, and the trigger has to show up.

Notice that the To-Do's spec never uses the words "ubiquitous" or "event-driven" anywhere, and still keeps the discipline. The FR-002 that opened this chapter is pure unwanted behavior: condition, action, shape of the response, negative guarantee. FR-005 of the filtering feature ("the system **MUST** open, on startup, in the default 'open' view") is event-driven, with "on startup" as the trigger. You do not need to announce the notation to reap its benefit. You need to know it so you notice when the sentence you just wrote fits no shape at all, which is the sign that it is incomplete.

One vocabulary detail the standard brought along, and one the To-Do's spec uses on every line: the **MUST** in capitals. It comes from the RFC tradition and separates obligation from suggestion. **MUST** is what the system has to do; **MUST NOT** is what it cannot do under any circumstance; **SHOULD** is a recommendation, and it is precisely because it is weak that it hardly appears in a good spec. If something is a **SHOULD**, ask why it is in the spec.

Given/When/Then: Behavior as a Scene

Given/When/Then came from somewhere else. It was born in BDD, Behaviour-Driven Development, formulated by Dan North out of the practice of TDD, and the original intent was pedagogical: people learning TDD did not know where to start writing a test, and writing the sentence before the code unblocked them.³ The format caught on because it serves both ends. People who do not program can read it and disagree; the test tool can execute it.

The shape has three parts, and each one answers a question:

- **Given:** what state the world is in beforehand. It is the setup, not the action.
- **When:** the single gesture that triggers the behavior.
- **Then:** what became true afterwards.

A real scenario from the To-Do's first feature, copied from the spec:

Given a task that already exists with a certain title, **When** the person tries to create another task with that same title, **Then** creation is refused with an error result explaining that the title already exists, and no new task is created.

Three things in that scenario deserve attention. The first is that the *Given* declares the initial state, and declaring initial state is where specs fail most. The second is that the *When* has a single action; a scenario with two *Whens* is two badly separated scenarios. The third is that the *Then* asserts two things, the error and the absence of a side effect, and the second is the one that actually catches the defect. A system that shows the error message and creates the task anyway passes half the scenario.

The relationship with EARS is not one of competition. EARS gives shape to the **rule**; Given/When/Then gives shape to the **example that proves the rule**. A good spec usually has both, and the To-Do's does: the functional requirements section is EARS without saying the name, the acceptance scenarios section is Given/When/Then saying the name out loud. When they disagree, either the rule is wrong or the example is wrong, and finding that out while reading costs one conversation. Finding it out later costs a whole lap.

Design by Contract: What Holds Before, After and Always

The third notation comes from programming, not documentation. **Design by Contract** was created by Bertrand Meyer along with the Eiffel language in the 1980s, and the idea is to treat every operation as a contract between the caller and the executor.⁴ Three clauses:

- **Precondition:** what has to be true for the operation to be callable at all. The caller's responsibility.
- **Postcondition:** what the operation guarantees will be true when it finishes successfully. The executor's responsibility.
- **Invariant:** what is true before and after, always, and which the operation has no license to break.

Why does this matter in a spec, if it is an idea from language design? Because the three clauses are questions most specs forget to answer, and the agent answers them on its own when they are missing.

Take the operation of editing a task, the To-Do's fifth feature, and read its spec through that lens:

- **Precondition:** a task with the given id exists. The spec says that, and it also says what happens when the precondition fails (error as value, not an exception), which is the decision that turns a precondition into specified behavior instead of into a crash.
- **Postcondition:** the task's title and description become the ones provided, and the title is still required and not repeated among the other tasks.
- **Invariant:** the task's position in the list and its creation date do not change. That holds before, during and after any edit,

and the spec states it as a requirement of its own, FR-005.

The invariant is the easiest one to forget and the most expensive one to discover late, because it belongs to no operation in particular: it belongs to the system. Nobody spontaneously writes "editing must not reorder the list", because reordering never crosses the mind of someone thinking about editing. It crosses the mind of whoever implements it, when the simplest way to save the change is to remove and reinsert.

One question worth asking of every spec before closing it: **what has to stay true once this feature exists?** The answers are the invariants, and each one deserves an explicit line.

ATDD: The Acceptance Criterion Written First

The fourth one is less a notation and more an order of work. **ATDD**, Acceptance Test-Driven Development, is the practice of writing the acceptance test before building, together with whoever asked for the feature, and using that test as the definition of done.⁵ The TDD Chapter 1 introduced does this at the level of a unit of code; ATDD does it at the level of the feature, and the difference in level changes who takes part: the unit test is written by whoever programs, the acceptance test is written by whoever knows what the thing needs to do.

In the flow this book walks, ATDD shows up without that name in two places. The **success criteria** of the spec, which are the verifiable statements about the final result. And the `quickstart.md` each feature produces, with its numbered list of manual validations that someone runs with the app open in front of them.

Notice what is specific about that order. Writing the criterion after building is describing what got finished, and that never fails anything, because the criterion is born molded to the result. Writing it first is taking on a commitment that can fail. It was item 4 of the fifth feature's `quickstart.md`, "edit only the description", written before the code existed, that failed the implementation and forced the spec to change. Had that list been drafted afterwards, it would have had five green items and one live defect.

Which to Use, and When

The four coexist in a single spec, and each occupies a different place in the document.

Notation	Answers	Where it goes in a spec	Sign that it is missing
EARS	Under what condition the rule holds	Functional requirements	A requirement with no trigger that looks like always holds and does not
Given/When/Then	What the example that proves the rule looks like	Acceptance scenarios	A rule with no concrete example, or an example with initial state
Design by Contract	What holds before,	Rules, edges and assumptions	An invariant nobody declared, and

	after and always		that the implementati breaks unnoticed
ATDD	How you know it is finished	Success criteria and manual validation	A criterion written after t code, that nev fails anything

The common mistake is not picking the wrong notation. It is believing that one of them makes the others unnecessary. A spec with nothing but Given/When/Then scenarios says nothing about the cases nobody wrote a scenario for; a spec with nothing but EARS requirements has not a single example to check whether the rule was understood; a spec with both can still declare no invariant at all and leave the list free to reorder.

What a Badly Formed Sentence Costs an Agent

Close the chapter with the case the rest of the book will meet again in other clothes.

Suppose the edit-task spec said only this about the title:

FR-003 (hypothetical version): The edited title **MUST** remain non-repeated, on the same criterion as creation.

The sentence looks complete. It has a subject, an obligation and a reference to a rule that exists in another document of the same project. It passes an inattentive reading, and it did: that was the original wording.

What it does not say is whether the task being edited counts in the comparison. A human reading that probably assumes it does not, because keeping your own title while fixing a description is obviously allowed. The agent assumed the opposite, compared against all tasks, and editing only the description started failing as a duplicate.

The real text, after the defect showed up in manual validation, is this:

FR-003: The edited title **MUST** remain non-repeated among the **other** tasks. Keeping the task's own title while editing does **NOT** count as a duplicate (it is the case of fixing only the description).

The difference between the two versions is one word and one exclusion clause. The first inherited the rule by reference; the second states it. None of the four notations in this chapter would have stopped the first version from being written, but all four would have raised the question: EARS asks for the explicit condition, Given/When/Then asks for a scenario covering the case of keeping the title, Design by Contract asks what the operation's exact postcondition is, and ATDD would have put "edit only the description" on the validation list before any code existed. The fourth is the one that caught it.

Chapter 6b comes back to this episode under another name and with another job. Here it served to show what an incomplete sentence costs. There it is the first of six patterns that repeat in specs of every kind, with the fix set beside each one.

Before that, one question this chapter did not touch is still open: given that you know how to write a requirement sentence, how many of them fit in a single spec? That is the next chapter.

Footnotes

"IEEE 830", the *Recommended Practice for Software Requirements Specifications* standard (1984, 1993 and 1998 editions), superseded by ISO/IEC/IEEE 29148. Entry "Software requirements specification", Wikipedia:

https://en.wikipedia.org/wiki/Software_requirements_specification

Alistair Mavin, Philip Wilkinson, Adrian Harwood and Mark Novak, "Easy Approach to Requirements Syntax (EARS)", *17th IEEE International Requirements Engineering Conference (RE'09)*, 2009, work developed in the context of aeronautical requirements at Rolls-Royce. The author's page, with a description of the patterns:

<https://alistairmavin.com/ears/>

Dan North, "Introducing BDD" (originally published in *Better Software*, 2006), on the origin of Behaviour-Driven Development out of teaching TDD and on the Given/When/Then format: <https://dannorth.net/introducing-bdd/>

Bertrand Meyer, *Design by Contract*, formulated along with the Eiffel language in the second half of the 1980s, with preconditions, postconditions and class invariants. Entry "Design by contract", Wikipedia: https://en.wikipedia.org/wiki/Design_by_contract

"Acceptance test-driven development", Wikipedia, on the practice of deriving acceptance tests from the customer's criteria before construction:

https://en.wikipedia.org/wiki/Acceptance_test-driven_development

2c - The Scope of a Spec: How Much Fits in One Specification

The Question That Comes Before Writing

You know what parts a spec has and you know how to shape each sentence inside it. What is missing is the decision that comes before both: **how much** goes into a single document.

It looks administrative and it is not. A spec that is too big produces a `tasks.md` of sixty items that the agent runs for three hours before you find out the third decision was wrong. A spec that is too small produces ceremony: seven artifact files for a change that was two lines. Both fail the same way, delivering late something nobody can review in one sitting anymore.

The To-Do became five specs, not one and not twenty. None of that was accidental, and the criterion that produced that number is what this chapter is about.

The Criterion: A Spec Is What Fits in One Lap

The unit is not the screen, nor the database table, nor the "module". The unit is **one capability the user can exercise from start to finish, and that you can review whole before approving.**

Three questions settle most cases.

Can you state it in one sentence, with no "and" in the middle?

"The person creates a task and sees their list" passes, because creating without seeing the result is no capability at all; the listing is what makes creation observable. "The person creates a task and gets an email reminder" does not pass: those are two independent capabilities that share a sentence only because they were remembered together.

If you shipped only this, could anyone use it? The To-Do's first feature shipped an app that already served a purpose: you can write down what needs doing and look at the list later. The second shipped completing and reopening. Neither depended on the other existing to be worth something.

Can you read the whole spec and disagree with it in ten minutes? This is the test that shows up least in books and decides the most in practice. The spec exists to be reviewed by you before it turns into code. A document you cannot finish in one sitting is a document you will approve by skimming, and approving by skimming is the same as not having written it.

Notice what those three questions do not ask: how many hours it takes, how many files it touches, how many lines of code come out. Implementation effort is a terrible slicing criterion, because anyone estimating effort before the plan exists is guessing, and because a capability that is small to describe can be expensive to build without ceasing to be a single capability.

Completing and Reopening Fit Together; Creating and Deleting Do Not

Watch the criterion work on the case it settled itself.

The To-Do's second feature is "complete and reopen a task". Two actions, one document. They stayed together because they are the same capability seen from both sides: the same state field, the same transition rule, and one without the other leaves the person stuck. A task that gets completed and never comes back is a task you cannot have checked off by mistake. The pair is the capability; each half alone is half a feature.

Creating and deleting, on the other hand, are actions that touch the same place and do not form a pair. You can ship creation without deletion and the app works. That is exactly what happened: deleting became the fourth feature, three laps later, with a spec of its own that brought in subjects creation never had, such as confirmation before an irreversible effect.

The quick test for cases like this is to look at **state**. If the two actions write to the same field with rules that depend on each other, they are probably one capability. If each touches a different place, or if one makes sense alone, they are two.

When the Feature Is Too Big

Sometimes you look at what has to be done and none of the three questions answers yes. The sentence has three "and"s, the spec takes more than ten minutes to read, and there is no way to ship half of it without shipping all of it. Time to split, and splitting well is the hard part.

The wrong cut is the **cut by layer**. One spec for the database, another for the logic, another for the screen. Each passes the size test and none of them delivers any capability: you get three complete laps of the cycle before anybody can use anything, and

the first one can only be validated by someone who can read a database schema. It is the waterfall of Chapter 1, now sliced horizontally.

The cut that works is **by complete path, from shallowest to deepest**. You take the whole capability and pull out of it the leanest version that still crosses every layer. Then the next spec fattens that version up.

The To-Do is the example, because it started big. The initial request was a single sentence, "I want a to-do list app", and that fits in no lap at all: it holds creation, listing, completion, filtering, deletion and editing inside it, with rules that did not even exist when the sentence was spoken.

The cut by complete path is what the whole book walks. First create and list, and the app already serves for writing things down. Then complete and reopen, and it starts serving for keeping track. Then filter, delete, edit. Each of those five crosses domain, storage and screen, and each leaves one more thing the person can do. You can stop after any of them and still have a whole app, just a smaller one.

The cut by layer, applied to the same request, would give three specs: the `Task` entity with `localStorage` persistence, then all the use cases, then all the screens. Add the three up and the result is the same app. The difference is that in the first two laps there is nothing to open and look at. Manual validation, which in the real lap closed the first feature with somebody typing a repeated title and checking the error message on screen, would have no way of existing before the third spec. And manual validation is always what catches whatever slipped past the automatic steps.

There are three signs that you cut in the wrong place:

- **One of the parts is not demonstrable.** If you cannot open the app and show what changed, that part turned into a chunk of

implementation instead of a slice.

- **The order between the parts is mandatory in both directions.** A dependency in one direction is normal. If A needs B and B needs A, you cut through the middle of one thing.
- **The same rule appears in both specs.** A duplicated rule is a badly drawn boundary, and the two copies will diverge by the third week.

A spec that is too big is almost always more than one slice, and where to draw the boundary between slices is where this book stops. The criterion that holds that cut up, the one that decides a responsibility belongs on this side and not that one, is the subject of *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. For slicing the To-Do, the three questions in this section were enough, and they will be enough for most of what you are going to write.

When the Method Does Not Pay Off

A book that argues for a way of working owes you the places where that way is a waste. SDD costs time before it saves time, and there are situations where the arithmetic does not work out.

A throwaway script. Renaming two hundred files, converting a spreadsheet, scraping a page once. If the program dies after it runs, specifying is writing documentation for a corpse. Ask directly and read the result.

Real exploration. You do not know what you want and you are going to find out by poking around. The spec is hostile here, because it asks you to declare up front what you will only know afterwards. Explore freely, throw away what you made, and write

the spec when you know what the question is. Nobody errs by exploring. The error is keeping the exploration's code after you found the answer.

A one-off fix with a known cause. An inverted condition, a field missing from the screen, a wrong label. If you know which line it is and you know what it should say, the spec adds nothing. The warning sign is when the "one-off fix" is the third one in the same place: at that point the cause was never known at all, and it is worth stopping to specify.

A proof of concept with an expiration date. A prototype for Friday's meeting that nobody will maintain. Same as the throwaway script, with one extra caution: a prototype that survives the meeting becomes a product without ever having had a spec, and that is how half the unmaintainable software in the world gets born.

In three other cases the method is misapplied for a different reason: it is not the size of the task that gets in the way, it is the absence of someone who decides. A spec written by someone with no authority to answer "is this behavior really the one we want?" turns into a document of questions. If you cannot get the answers, the problem is not the method.

Outside those situations, the arithmetic tends to work out on the very first lap, and for a simple reason: the cost of writing the spec is yours, once; the cost of not writing it is the agent's, every time, multiplied by every assumption it has to make on its own.

What You Take From Here

One spec per usable capability, reviewable in one sitting. Splitting by complete path, never by layer. And the honesty not to specify what dies in an hour.

With the blueprint drawn, the handwriting settled and the size of the sheet decided, what is missing is the tool that turns all this into a repeatable process. That is what comes next.

3 - Hands on: the SDD tools

From the blueprint to the wall

In Chapter 2 you drew the blueprint. You started from a loose sentence, "I want a to-do list app," and arrived at an anatomy that gives direction: the why, the scope, the behavior, the acceptance criteria, the edges. The two chapters after it settled the handwriting of each requirement and the size of the sheet. The spec is ready to guide. But a blueprint, however careful, raises no wall on its own. At some point someone picks up the blueprint and starts laying brick.

That moment arrives now. After three chapters of foundation, you touch a real tool, one that takes the `specify → plan → tasks → implement` cycle and turns it into commands you type and artifacts that appear on your screen. Leaving the concept and getting your hands dirty changes the question that drives the material. Until now the question was "what is a good spec?". From here on it becomes another: **with which tool, and why?**

There is more than one answer, and none of them is magic. There is a handful of SDD tools mature enough to take on a real project, each with its own way of embodying the same cycle, each strong in one place and heavy in another. This chapter won't shove a choice down your throat in the first line. It will first show the terrain: three tools, side by side, for what they really are. Only then, with the terrain in view, do we decide which one raises the wall of this material, and why.

Three ways of doing the same thing

Start by getting to know the three by what matters first: the **mental model** of each one, that is, the unit of work you reason with, and the **real order** of the steps it makes you follow. All three are operated through a *CLI* (command-line interface), the way to command a program by typing instructions in the terminal instead of clicking buttons; and, in practice, through commands you give to the AI assistant itself inside the editor. What changes is the shape the cycle takes in each one.

spec-kit, from GitHub, thinks in terms of **feature**. The unit is a feature, usually isolated on its own Git branch, with a specs folder next to it holding the documents for that feature. The real order of the flow is `constitution → specify → (clarify) → plan → (check) → tasks → (analyze) → implement`, with a few optional steps in the middle.¹ You install a CLI called `specify`, initialize the project, and then talk to the assistant through commands in the `/speckit.<command>` format (in some agents they show up with a hyphen, `/speckit-<command>`). Hold on to the first step of that order, `constitution`, because we come back to it in the very next section.

OpenSpec, from Fission-AI, thinks in terms of **change**. A *change* is OpenSpec's unit of work: a package that describes an alteration (the why, the specs that change, the tasks) before applying it to the project. The real order lives in a set of commands under the `/opsx:` prefix, and goes `/opsx:explore → /opsx:propose → /opsx:apply → /opsx:archive`; you can even skip straight to `/opsx:propose` when you already know what you want.² Each change carries only the part that changes, which you could call a *spec delta*, the altered slice of the specification; when the change is archived, that slice is merged into the project's living spec. It is a lighter, looser approach: there is no rules document governing the whole project, and you iterate proposal by proposal.

BMAD Method, from the `bmad-code-org` community, thinks in terms of **epic** and **story**. An *epic* is a large block of work; a *story* is a small, implementable slice of it, handled one at a time. It is the most ceremonious of the three. In the current version, V6, the work goes through four phases, Analysis, Planning, Solutioning, and Implementation, and is driven by *agent personas*, specialized roles the assistant takes on at each stage (an analyst, an architect, a developer, and so on).³ Along the way comes the *PRD* (product requirements document), the document that says what you want to build and why before discussing architecture. It is a lot of apparatus, and some projects ask for exactly that.

For those who want to look inside: in OpenSpec, a change is literally a folder in `openspec/changes/` with a `proposal.md`, the requirements, and the tasks; archiving moves that folder to a history and updates the living specs. In BMAD V6, the default set is six named personas (Analyst, Product Manager, Architect, Developer, UX Designer, and Technical Writer); separate Scrum Master, QA, or Product Owner roles, which appeared in older versions, are not part of that default. Skipping this box doesn't lose you the thread: what matters above is each tool's mental model, not the name of each file or persona.

The cycle is the same, the incarnation changes

A doubt may hit here, and it is a healthy one: if each tool has different commands and names, what is left of the `specify → plan → tasks → implement` cycle you learned at the start of the material? The answer is the key to this chapter: **the cycle is the same; the tool is just the shape it takes**. Specify, plan, break into tasks, and build are still the four steps, in any of the three. What the tool

does is give that cycle a concrete body: commands, files, an order to follow. Switching tools switches the packaging; the content stays.

One detail of spec-kit tends to startle at first sight, and it is worth defusing the startle now. Its flow doesn't open on `specify` ; it opens on `constitution` . It looks like a step invented outside the cycle you learned, but it is a **governance** step that comes before it: the *constitution* fixes the principles that every later decision has to respect, and it prepares the ground for `specify` to happen within agreed limits. The next chapter is entirely about it; for now, hold on to why it exists: the AI assistant, left loose, tends to build beyond what was asked, and the constitution is the leash that holds that impulse back. The cycle doesn't change from tool to tool; what changes is the accent each one pronounces it with.

The honest comparison

You already have the mental model. Now the part that decides the choice: where each tool shines and where each one weighs, by the same criteria, without selling any as the single solution. The table below covers the three by the six criteria that matter most when choosing. Right after it, a paragraph of reading per tool, because the meaning of a comparison can't depend on you memorizing a grid.

Criterion	spec-kit	OpenSpec	BMAD Method
Unit of work	one feature at a time, isolated on a project branch	one change proposal at a time	epics and stories driven by personas
Real order	governs,	explores,	analyzes,

	specifies, plans, breaks into tasks, and builds	proposes, applies, and archives	plans, designs, and implements
Weight / ceremony	medium; with a governance step up front	light; no mandatory steps	heavy; four phases and many roles
Where it shines	organized flow, without tying down the technology, switching assistants with no rework	quick, small adjustments on a project that already exists	large projects, deep planning, multiple domains
Weakness	young and experimental; an "eager" agent	less governance; risk when merging changes	steep learning curve; overkill on a small project
Maturity / adoption	from GitHub; less than a year old; very popular	popular; community asks for more documentation	stable V6 version; consolidated adoption

spec-kit is the balance. It gives you a structured flow, with one artifact per phase, without locking you into a language or a specific assistant: switch agents without redoing the work. It shines when you want discipline without marrying a *stack* (the project's pile of technologies: language, database, framework). The price is its youth. It is an experimental project, with less than a year on the road, and it carries the vice the constitution exists

to contain: the agent, left loose, is eager and generates beyond scope. Containing that is a matter of instruction and context, and assembling what the agent sees before it decides belongs to *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering/>), the third volume in this trilogy, which answers what the agent sees right now, in the window of this one call, and at what cost. To pick a tool here you do not need it: it is enough to know that spec-kit answers that vice with the constitution, the gate Chapter 4 lays out. It is more ceremony than OpenSpec asks for, and less than BMAD imposes.

OpenSpec is the lightness. No constitution, no phase gates, you open a change, describe what changes, and go. It shines on an existing codebase, in that work of adjusting what is already standing in small, incremental changes, without the ceremony of a full flow. The lightness has its cost: less governance means less safety net. There are reports of scenarios lost in silence when two changes touch the same requirement and are archived, and the documentation is still thin for the more complicated flows. Freedom with fewer rails.

BMAD Method is the depth. Four phases, several personas (the specialized roles the assistant takes on at each stage, like analyst, architect, and developer), expensive planning done carefully before a line of code. It shines on a large project, with many people and a lot to coordinate, where skipping the planning costs more than doing it. For a to-do list app, though, it is overkill, and it doesn't hurt to say so plainly: setting up four phases and six roles to record and complete tasks is using a truck to deliver a letter. Calling it overkill here does not diminish BMAD; it just means the tool was made for a scale our example does not have. On the right project, all that weight is precisely its strength.

Notice what the table doesn't have: a "best" column. None of the three is the answer to everything. Each one solves one kind of problem well and gets in the way on another, and choosing is matching the tool to your case, not crowning a universal champion.

The choice and why

With the terrain in view, the material makes its choice: from here on, it builds with **spec-kit**. The decision comes with a reason, and there are three explicit criteria holding it up.

The first is *dogfooding*, the practice of using the very tool you recommend. I don't recommend spec-kit secondhand: I actually use it, in the work that matters most, building software. I run several development projects with this methodology, from **constitution** to **implement**, and it is in that daily practice, and not in a marketing brochure, that I learned where it helps and where it gets in the way. I saw the structured flow avoid rework on a serious project, and I also saw the agent get eager and want to build beyond what was asked, in the way I described in the weaknesses. When this chapter says something shines or something weighs, it is the account of someone who took hits and got it right with the tool in hand, in code that went to production. This material itself is also built with spec-kit, by the same cycle you will operate, but that is the smallest of the reasons: the weight of the recommendation comes from the software projects, not from the text. Recommending what you use every day is more honest than recommending what you admire from afar.

The second criterion is the direct mapping of the vocabulary. The cycle you have carried since the first chapter is **specify → plan → tasks → implement**, and spec-kit's core commands are exactly **specify**,

`plan`, `tasks`, and `implement`. There is no mental translation on the way: what you learned in concept is what you type in practice, with the same name. For a material that teaches the cycle, that one-to-one correspondence is worth gold, because it removes a whole layer of friction between understanding and doing.

The third criterion is what I call the *middle path*: the deliberate choice of the intermediate option, neither the simplest nor the most robust, when both extremes charge too high a price. Look at the comparison. OpenSpec is light, but the lightness becomes a lack of net when the project grows; BMAD is powerful, but the power becomes ceremony that drowns a small project. spec-kit sits in the middle: it has enough structure to contain the eager agent, without the paraphernalia that stalls someone who just wants to start. Far from lukewarm, that balance is the position that serves the widest variety of projects, and it was what, in my practice, made me settle on it and not on the extremes.

Now the part that matters as much as the choice: it is a **teaching convention**. If your case calls for OpenSpec's lightness or BMAD's depth, go with it head held high; the cycle you are learning is the same in all three, and almost everything that follows translates to any of them.

Lean setup and first contact

Enough talk: time to install. The path here is the minimum to go from zero to a standing project, and it stops exactly at initialization. You won't run any flow command yet; that is a matter for the next chapters. The goal of this section is only to leave the tool installed and the To-Do ready to receive the first command.

One caveat before the steps, and it is important: tools like spec-kit ship versions in a matter of days. That is why the up-to-date installation path lives in the official documentation, not here. What follows is the general shape, checked and working, but the living and definitive source is the official spec-kit repository (<https://github.com/github/spec-kit>).⁴ When some detail diverges, trust the documentation, not this page.

spec-kit runs on top of a CLI called `specify`, installed with `uv`, a package manager from the Python world that downloads and runs tools like this one. It is not this material's job to teach you how to install `uv` or how to deal with Python and its managers: that would be a detour from the main thread, and they are well-documented steps that you resolve with a quick search or by asking the AI assistant itself to explain what `uv` is and how to use it.

With `specify` available, initializing the project is a single command:

```
specify init todo --integration claude
```

`specify init` creates the project and, along the way, asks two questions that matter. The first is which AI assistant you use. In the example above I fixed **Claude Code** with `--integration claude`, but that is just an example: spec-kit supports dozens of assistants (Copilot, Gemini, Cursor, Codex, and many others), and changing the value doesn't change anything you learn here. The assistant is your choice, as the app's stack always was. The second question is the project's script type, *bash/zsh* for Linux and macOS, *PowerShell* for Windows.

What appears after the command is the *scaffolding*, the skeleton of files and folders the tool generates for you to start from. It is worth opening it and recognizing what came:

```
todo/
├── .specify/          # the heart of the tool: scripts, templates, and the pr
object's memory
│   └── memory/
│       └── constitution.md # the constitution, still blank, waiting for th
e first command
├── .claude/           # the integration with the chosen assistant (here, Clau
de Code)
└── CLAUDE.md          # the project's instructions for the assistant
```

None of this is your app yet. There is no screen, database, or to-do list code; there is the scaffolding that will hold up the build. Notice the `constitution.md` inside `.specify/memory`: it is the file the first flow command will fill in, that governance that precedes specifying. It sits there, empty, precisely to remind you that the next step has a name.

And the setup ends here, on purpose. The tool is installed, the To-Do is initialized, the scaffolding is in view. You still haven't written a line of spec or run `constitution`; you only prepared the ground.

For anyone who wants to try the other two: OpenSpec is installed via npm and initialized with `openspec init`; BMAD is installed with `npx bmad-method install`. Both have their own documentation and their own flows, and nothing stops you from initializing a test project with them to feel the difference. Here, we go on with spec-kit.

The bridge to the full cycle

You entered this chapter with a blueprint in hand and leave with a tool installed and the To-Do project initialized, ready to receive commands.

The ground is prepared, and the first command has a name. The next step is to run `constitution`, fix the principles that govern the project, and from there carry the spec forward through the entire cycle: specify, plan, break into tasks, and, at last, build. The tool is in hand and the To-Do is waiting for the first command.

Footnotes

GitHub Spec Kit, official repository and documentation, flow `constitution → specify → (clarify) → plan → tasks → (analyze/checklist) → implement` and `specify init`: <https://github.com/github/spec-kit> · <https://github.github.io/spec-kit/> (checked on 2026-06-25, `specify` CLI v0.11.8).

OpenSpec (Fission-AI), official repository, `/opsx:` namespace and the `explore → propose → apply → archive` flow with change/archive: <https://github.com/Fission-AI/OpenSpec> (checked on 2026-06-25).

BMAD Method (bmad-code-org), official repository and documentation, version V6 with four phases (Analysis → Planning → Solutioning → Implementation) and six default personas (Analyst, Product Manager, Architect, Developer, UX Designer, Technical Writer): <https://github.com/bmad-code-org/BMAD-METHOD> · <https://docs.bmad-method.org> (checked on 2026-06-25).

GitHub Spec Kit, official repository and documentation, flow `constitution → specify → (clarify) → plan → tasks → (analyze/checklist) → implement` and `specify init`: <https://github.com/github/spec-kit> · <https://github.github.io/spec-kit/> (checked on 2026-06-25, `specify` CLI v0.11.8).

4 - speckit constitution: the rules before the first move

The rules before the first move

In the previous chapter you got the ground ready: the tool installed, the To-Do project initialized, the spec-kit scaffolding in view. Coming back to the construction metaphor that opens this material, it is like having the lot cleared, the hoarding up, and the mixer running. What is missing is the first work order. And here comes the surprise that tends to unsettle anyone arriving from improvisation: the first command in the flow writes no feature at all. It doesn't draw the list screen, it doesn't create the add-task button, it doesn't touch the database. The first command writes the **rules of the project**.

Remember the `constitution.md` file, sitting empty inside `.specify/memory`, that Chapter 3 left waiting? That is what steps in now. Before you specify what the To-Do does, spec-kit asks you to say under which principles it will be built.

That raises a fair question, and it is the question that runs through the whole chapter: why is the first thing the tool asks for the rules, and not the specification? Why spend energy on government before you even know what the first feature is? Hold on to the question. The answer starts with a word you already use outside programming without noticing.

What a project constitution is

You live alongside constitutions your whole life, even without calling them that. A country has a constitution: the text that stands above any ordinary law and that every new law has to respect in order to hold. A condo has bylaws: you can decorate your apartment however you like, but you can't change the façade or do noisy construction on a Sunday, because there are general rules that apply to every unit. A city has a building code: each building project is different, but all of them obey the same limits on height, setback, and safety.

Notice the shared pattern. In each case there is a set of general, lasting rules, decided once and rarely touched, that frame every specific decision that comes later. You don't re-argue the building code with each floor that goes up. It sits there, in the background, governing.

A software project constitution is exactly that. It is the project's **governance layer**: the set of non-negotiable principles that every spec, every plan, every task list, and every line of implementation has to respect. Governance, here, is the level of the rules that sit above the day-to-day decisions and that decide what is acceptable across all of them.

Now the hook's question answers itself. The rules of the game have to exist before the first move. If you were to specify the To-Do's first feature without having fixed the principles, each spec would invent its own rules, and the project would turn into a patchwork of local decisions. The constitution comes before **specify** because it is what defines the limits within which every spec will be written.

It is worth carefully separating the constitution from the other artifacts in the flow, because they are easy to confuse. The **constitution** is the general, lasting rules of the whole project. The

spec is the *what* of a specific feature. The **plan** is the *how* and the technology. The **tasks** are the execution steps. The constitution is the only one that cuts across everything and that rarely changes: specs come and go with each feature, but the constitution governs all of them, from the start of the project to the end.

What goes in and what stays out

Once you know what the constitution is, the next practical doubt is what to write inside it. A good constitution usually holds four kinds of content.

First, the **principles**, each with its **rationale**. A principle without the why is an arbitrary order, easy to ignore when the deadline bites. With the rationale, it becomes an argument: you know which problem the rule avoids and so you respect it even under pressure. "Every operation that can fail returns an explicit result" is a principle; "because that way the error path becomes part of the contract, and not a surprise at runtime" is the rationale that holds it up. Second, the **engineering and quality standards**: the technical practices the project demands of itself, such as SOLID, the five design principles that keep classes and modules cohesive, decoupled, and easy to change, Clean Architecture, and the TDD you already know from Chapter 1. Third, the **workflow**: how the team (or you alone with the AI) carries each change through. Fourth, the **governance** proper: how the constitution is versioned and by what criterion it is amended, the subject that closes the chapter.

Just as important as what goes in is what **stays out**. These do not belong in the constitution: the programming language, the framework, the library, the state management pattern, the

database, the infrastructure, and any implementation detail. No "use PostgreSQL," no "React 19," no "global state with Redux."

Why this strictness? Because those choices are technology, and technology is a decision for another step, the `plan`, not for the constitution. It is the principle of **technology-agnostic specification**, one of the central ideas of SDD: intent and lasting rules stay separate from implementation. A constitution swollen with stack decisions recreates vibe-coding with more ceremony; an empty constitution governs nothing. The balance point is to fix principles and structure, and to leave technology for later.

Notice that this section, on its own, already works as a checklist: for any line you think of putting in the constitution, ask whether it is a lasting rule or an implementation choice. If it is implementation, it lives in the `plan`.

Constitution, spec, or plan? The rule that settles the doubt

In practice, three questions resolve almost every case. **What** the app does (the user can mark a task as done, can filter by date) is a functional requirement, and that lives in the **spec**. **How** it is built and under which standard (layered architecture, TDD, error as value, and even a general scope limit like "this app is single-user") are rules that hold for every feature, and that is **constitution**. **With which** technology (the language, the framework, the state library, the database) is **plan**.

It is worth undoing here a confusion inherited from systems analysis, because it trips up experienced people. In scope prioritization, the analyst separates out what is "MUST HAVE": the list of essential features. It is tempting to think the constitution is that document, but it isn't. That list of features is spec. The constitution's "MUST" is of another nature: instead of "this feature must exist," it says "every feature must obey this."

One describes the features; the other describes the rules above all of them. That is why a functional requirement, however mandatory, does not go in the constitution.

And your preferences? They pass through the same filter. If the preference is a quality standard you want across the whole project (always test first, always isolate data access), it is a principle and goes in the constitution with its rationale. If it is a tool taste (this framework, that database), it is stack and goes to the `plan`. If it is a whim with no why that survives deadline pressure, it goes nowhere.

Running the `constitution` step on the To-Do

With the theory solid, let's run the step for real on the To-Do. In spec-kit, creating or updating the constitution is a step in the flow that starts from a template and writes a real file of the project.^{[1](#)}

The mechanism is simple to describe. There is a template with fill-in markers at `.specify/templates/constitution-template.md`, with fields like the project name and the names and descriptions of the principles. From your answers, the step fills that mold and writes the constitution to `.specify/memory/constitution.md`, the same file that sat empty waiting for you. At the top of the file the command keeps a **Sync Impact Report**, a short account of what changed from one version to the next, applies **semantic versioning** to the version number, and propagates the adjustments to the dependent templates (plan, spec, and tasks), so that none of them ends up talking about a rule the constitution no longer has.

I won't transcribe every option of the command here, and for a reason of method: command-line instructions change with each release of the tool. What ages well is the understanding of the

step (template filled, file written, version controlled); the exact detail of the command you always check in the official spec-kit documentation, which is the living source.[2](#)

What you write in the command

But what, in the end, do you provide in this step? The command doesn't invent your project's rules: you describe them in plain language and the tool organizes them into the constitution's format. The input you hand over is the list of principles you want, each with its rationale, plus what should stay out.

So this doesn't stay abstract, here is the text you provide to the `constitution` step, word for word, to generate a constitution like the To-Do's that you are about to read. Copy it and adapt it to your project, swapping the principles for your own:

```
Create the Constitution for the To-Do project, a personal task-list app (single-user).
```

```
The Constitution must define only permanent engineering and architecture principles. It must not specify technologies, languages, frameworks, libraries, state management patterns, databases, or implementation details. Those decisions belong to the Plan, not to the Constitution.
```

```
Produce a lean, clear, normative constitution, starting at version 1.0.0.
```

```
The constitution must contain exactly six principles, each made up of:
```

- * a short title;
- * objective rules using normative language ("MUST", "MUST NOT", "SHOULD", etc.);
- * a brief *Rationale* explaining which problem the principle avoids.

```
The six principles must address the following themes:
```

1. Layered Architecture

```
The application must clearly separate responsibilities into layers, keeping the direction of dependencies always pointing toward the application domain. B
```

business logic must remain independent of the user interface and the infrastructure. Dependency composition must happen only at the edge of the application.

The Constitution must define only the architectural principles. It must not impose frameworks, specific state management patterns, dependency injection mechanisms, or implementation details.

2. Isolated Business Logic

Every business rule must reside in a layer of its own in the application, independent of the user interface and the infrastructure.

The interface only collects user input and presents results. No business decision must exist in the UI.

3. Error as Value

Predictable failures must be represented by explicit success or error results, never by exceptions propagated across layers.

Exceptions must remain confined to the boundaries with infrastructure.

4. Test-Driven Development

Every new behavior must be specified by automated tests before or during its implementation.

Development must follow a flow that encourages small iterations, continuous validation, and safe refactoring.

5. Simplicity

The project must stay deliberately simple.

Being a personal, single-user app, any feature not needed for the scope must be avoided (YAGNI). The code must prioritize readability, low coupling, and ease of maintenance.

6. Technology Agnosticism

The Constitution must never impose a language, framework, library, database, interface architecture, state management pattern, or any specific technology.

Those choices belong exclusively to the Plan.

After the principles, you must include the following sections:

Engineering Standards

Consolidate the general principles used by the project, including:

- * SOLID;
- * Clean Architecture;
- * Test-Driven Development;
- * Error as Value;
- * Separation of Concerns.

Workflow

Briefly describe the expected development flow:

1. Constitution
2. Spec
3. Plan
4. Tasks
5. Implementation
6. Tests
7. Review

Governance

Include:

- * semantic versioning;
- * a process for creating amendments;
- * a compatibility rule between versions;
- * an obligation to review the Constitution whenever a permanent architectural change occurs.

The result must be a professional, objective, and lasting document, focused on engineering principles, avoiding implementation decisions or details specific to any technology.

Write each rule the way you would explain it to a colleague joining the project tomorrow: the rule, the why behind it, and the limit of what it does not cover. The clearer the request, the less the tool has to guess and the better the first result comes out.

One note before you read the result, and it holds for everything you build from here on: **version the project with git from the start**. The constitution is the first of several files the flow will write into the repository, and all of them are text that evolves. Without version control you lose the history of each amendment and the why behind it, which is exactly what makes these artifacts living and auditable. Initialize git in the To-Do project before moving on, and commit the constitution as soon as it is born. Each project has exactly **one** constitution, in its own `.specify`, and it is that one you look after from now on.

The step ran and the file was born. Let's read it.

The To-Do's constitution, principle by principle

The constitution that was born for the To-Do is **lean and complete**: six principles, enough to govern a real app without turning into a treatise. I'll comment on each one in excerpts, with the why of its being there. The whole file, with the supporting sections and the version header, is reproduced in this chapter's appendix; here you get the commented cuts, not the file dumped all at once.

The first principle is the structural heart and deserves the most care. Note the word **MUST**, placed by spec-kit: it means *DEVE* (and **MUST NOT**, *NÃO DEVE*). AI agents generally have no trouble with mixed languages in the same document:

I. Layered Architecture. The application **MUST** clearly separate responsibilities into layers, keeping the direction of dependencies always pointing toward the application domain. Business logic **MUST** remain independent of the

user interface and the infrastructure. Dependency composition **MUST** happen only at the edge of the application.

This constitution **MUST NOT** impose frameworks, specific state management patterns, dependency injection mechanisms, or implementation details. It defines only the architectural principles.

Several new terms here, and each is worth a definition. **Layered architecture** is organizing the code into bands with distinct responsibilities, instead of everything mixed together. The **domain** is where the business rule lives, the heart of the application. The **direction of dependency** is the rule of who may know whom: the arrows always point inward, toward the domain, and never the other way. The outer layers (the interface, the infrastructure) know the inner ones; the domain ignores whoever is outside. **Dependency composition** is the moment the concrete pieces get wired to one another; keeping it "at the edge" means assembling everything at a single entry point of the application, leaving the core free of details. That is the essence of **Clean Architecture**: the business rule at the center, frameworks and details at the edge.

A concrete way to embody this, which you will see in the next chapters, is to separate the screen (view), the band that holds the screen's state, the use cases that run the rules, and data access isolated in a repository. But notice: the constitution does **not** descend to that level. It fixes the principle (layers, direction toward the domain, composition at the edge) and stops there.

And there is a deliberate detail in that "stops there." The principle says, in so many words, that the constitution does not impose a state management pattern or a dependency injection mechanism. It doesn't order "use the Redux pattern." How you

manage state follows whatever is idiomatic in the stack you choose, and that is decided later, in the `plan`. It could be Redux or Zustand in React, BLoC in Flutter, NgRx in Angular: the constitution doesn't choose for you. What it guarantees is the structure; how the state is embodied is free. Which of those patterns to adopt, and what each one charges in return, belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it here: in the To-Do that choice shows up in the first cycle's `plan`, and all the constitution demands is that business rules not live in the screen.

The other five principles complete the mold:

II. Isolated Business Logic. Every business rule **MUST** reside in a layer of its own in the application, independent of the user interface and the infrastructure. The interface **MUST** only collect user input and present results. No business decision **MUST** exist in the UI.

Concentrating the rule in one place makes it testable without the screen and predictable for humans and for the AI.

III. Error as Value. Predictable failures **MUST** be represented by explicit success or error results, and **MUST NOT** be propagated as exceptions across layers. Exceptions **MUST** remain confined to the boundaries with infrastructure.

This is **error handling as value** (sometimes called the Result type): the failure becomes part of the function's contract, and the caller is forced to handle it, instead of being surprised at runtime.

IV. Test-Driven Development. Every new behavior **MUST** be specified by automated tests before or during its implementation. Development **SHOULD** follow a flow of small iterations, with continuous validation and safe refactoring.

It is the TDD from Chapter 1, now raised to a rule of the To-Do: specifying behavior by test stops being an option and becomes law.

V. Simplicity. The project **MUST** stay deliberately simple. Being a personal, single-user app, any feature not needed for the scope **MUST** be avoided (YAGNI). The code **SHOULD** prioritize readability, low coupling, and ease of maintenance.

The app exists to teach the method; needless complexity only steals the focus.

VI. Technology Agnosticism. This constitution **MUST NOT** impose a language, framework, library, database, interface architecture, state management pattern, or any specific technology. Those choices **MUST** belong exclusively to the `plan`.

It is the technology-agnostic specification from the previous section, now written as a rule of the To-Do.

The five engineering standards the project consolidates (SOLID, Clean Architecture, TDD, error as value, and separation of concerns) appear gathered in a supporting section of the file,

without repeating the text of the principles. And a necessary bit of honesty: going deep on this layered architecture is not the focus of this chapter. It is the subject of *FOCUS Architecture* (<https://books.kodel.com.br/en/books/focus/>), where the same separation appears drawn layer by layer. You do not need it to go on from here: what is enough now is to understand that the constitution fixes the structure, and the detail of how to draw each layer is left for later.

Checking and improving the result

Once the file is generated, don't accept it in the dark: read it as a reviewer, because the result is yours and may come out different from what I showed here. Three checks are enough. The first is **coverage**: is each principle you asked for present, with a clear name and a rationale that truly justifies the rule, instead of repeating it in other words? The second is **leakage**: did any line fix a technology by accident? Look for names of languages, frameworks, libraries, or databases; they can appear only as an example of "follows the stack," never as a requirement. In the architecture principle, confirm that it fixes the layers and the direction of dependency, but does not tie down a state pattern. The third is **form**: is the version header there, starting at 1.0.0, and does the Sync Impact Report record what was born?

If something came out vague, missing, or wrong, you have two ways out. You can run the step again with a sharper request (a more concrete rationale, the principle that was missing) or edit the file by hand and bump the version according to the size of the change. The tool gives the first shape; the owner of the constitution is you.

Why the stack stays out

Maybe you felt a tension reading the first principle. If the constitution fixes the architecture, isn't it, deep down, fixing technology? And doesn't that contradict the very principle of technology agnosticism that the To-Do's constitution just declared?

There is no contradiction, and untying that knot is the point of this section. Architecture and stack are things of different natures. **Architecture** is a structural, lasting principle: the idea of separating view, state, use cases, and data access, and of making the dependencies point inward, holds regardless of any tool. It is governance. **Stack** is an implementation choice: which language, which framework, which state library, which database. It is a **plan** decision, and it is your choice.

The proof is in portability. The same To-Do constitution serves a React app, a desktop in Delphi, a system in Java or C#. What changes from one to another is only the **embodiment** of the architecture: the separation into layers exists in all of them, but the way to write a data repository in Java is not the way to write it in Dart, and the way to manage state in React is not the way you do it in Flutter. The structure is the same; the clothes it wears are the stack's.

Hence the postponement being deliberate. Fixing the stack in the constitution would jump the gun on a decision the method tells you to make later, with the right information in hand, in the **plan**. The constitution locks down the **intent** (how the project is organized and what it demands of quality); the **plan** locks down the **implementation** (with what this will be built). That separation between intent and implementation is exactly what gives SDD its strength: you decide each thing at the right moment, without tying down too early what can still change.

Living artifact and the bridge to the first spec

One last idea is missing so the constitution doesn't look like a foundation stone you lay and forget. It is a **living artifact**, in the same spirit as the spec from Chapter 2. It was born at version 1.0.0 and evolves by **semantic versioning**: an **amendment** that removes or redefines a principle in an incompatible way bumps the major version, a new principle bumps the middle one, a simple clarification bumps the minor one. Every amendment enters with a criterion and with its rationale, recorded in the Sync Impact Report. The constitution changes little, but when it changes, it changes in a controlled and traceable way.

And it is not forgotten after being written: further along the flow it comes back to demand conformance, in a check called the **constitution check** that the next chapter presents at the right point in the cycle.

Clear the context at the end of each step

A habit worth adopting from now on, and one many people forget. When you finish each step of the flow (`constitution` , `specify` and `clarify` , `plan` and `check` , `tasks` and `analyze` , `implement`), run a `/clear` to reset the agent's context before moving on to the next. It seems counterintuitive to throw away everything the agent "learned" in the conversation, but that is exactly the point of SDD: what rules is the **generated documentation**, never what stayed in the conversation's memory. The constitution, the spec, the plan, and the tasks are the source of truth; the next step reads those files and works from them, not from a chat history that kept piling up.

Clearing the context brings two gains. The first is quality: the agent restarts each step looking at the artifact, without dragging along assumptions, dead ends, and misunderstandings from the

previous step. The second is cost: huge contexts burn a lot of tokens on every interaction, and carrying the whole conversation from one end of the flow to the other is expensive for no reason. If the documentation is good, it is enough. And if it isn't enough, the problem is in the documentation, not in the lost context, and that is where you should go back to fix it.

The To-Do's rules are set; the ground, which was already prepared, now also has its laws. The constitution, however, sits above the flow: it is written once and governs everything, but it is not part of the cycle that repeats with each feature. Before running the first `specify`, it is worth climbing to a high point and seeing this whole cycle from above: what path a feature travels, from the git branch to the integrated code. That is the map the next chapter draws, so that only then do we come down to the ground and write the To-Do's first spec.

Footnotes

GitHub Spec Kit, official repository and documentation, flow `constitution → specify → (clarify) → plan → tasks → (analyze/checklist) → implement` and `specify init`:
<https://github.com/github/spec-kit> · <https://github.github.io/spec-kit/> (checked on 2026-06-25, `specify` CLI v0.11.8).

GitHub Spec Kit, official repository and documentation, flow `constitution → specify → (clarify) → plan → tasks → (analyze/checklist) → implement` and `specify init`:
<https://github.com/github/spec-kit> · <https://github.github.io/spec-kit/> (checked on 2026-06-25, `specify` CLI v0.11.8).

4.5 - Appendix: the To-Do constitution

This appendix reproduces, in full and unedited, the `.specify/memory/constitution.md` file actually generated for the To-Do project in its isolated spec-kit workspace, with `specify` version 0.11.8. It is the faithful copy of the real artifact commented on principle by principle in the body of Chapter 4. The comment block at the top (Sync Impact Report) is part of the generated file and is kept here as is.

To-Do Constitution

Core Principles

I. Layered Architecture

The application **MUST** clearly separate responsibilities into layers, keeping the direction of dependencies always pointing toward the application domain. Business logic **MUST** remain independent of the user interface and the infrastructure. Dependency composition **MUST** happen only at the edge of the application.

This constitution **MUST NOT** impose frameworks, specific state management patterns, dependency injection mechanisms, or implementation details. It defines only the architectural principles.

Rationale: what is lasting and governable is the structure, that is, the layers, the direction of dependency, and the isolation of the domain. Names of frameworks and state patterns are details that change with the stack and do not belong in a permanent document.

II. Isolated Business Logic

Every business rule **MUST** reside in a layer of its own in the application, independent of the user interface and the infrastructure. The interface **MUST** only collect user input and

present results. No business decision **MUST** exist in the UI.

Rationale: concentrating the rule in a single place makes the behavior testable without a UI, predictable for humans and agents, and independent of the presentation technology.

III. Error as Value

Predictable failures **MUST** be represented by explicit success or error results, and **MUST NOT** be propagated as exceptions across layers. Exceptions **MUST** remain confined to the boundaries with infrastructure.

Rationale: making the error path part of the operation's contract forces the caller to handle the failure and eliminates runtime surprises that hide behind unforeseen exceptions.

IV. Test-Driven Development

Every new behavior **MUST** be specified by automated tests before or during its implementation. Development **SHOULD** follow a flow of small iterations, with continuous validation and safe refactoring.

Rationale: the method this project practices demands the discipline it preaches. The test fixes the intent before the code and protects continuous refactoring.

V. Simplicity

The project **MUST** stay deliberately simple. Being a personal, single-user app, any feature not needed for the scope **MUST** be avoided (YAGNI). The code **SHOULD** prioritize readability, low coupling, and ease of maintenance.

Rationale: the app exists to serve as an example of the method. Needless complexity steals the focus and turns the example into noise.

VI. Technology Agnosticism

This constitution **MUST NOT** impose a language, framework, library, database, interface architecture, state management pattern, or any specific technology. Those choices **MUST** belong exclusively to the Plan.

Rationale: the constitution locks down the intent, not the implementation. The same structure serves any stack; choosing technology here would jump the gun on a decision the method tells you to defer to the Plan.

Engineering Standards

The project consolidates the following general engineering principles, which hold up the principles above without duplicating them:

- **SOLID** as the basis for the design of classes and modules.
- **Clean Architecture:** business rule at the center, frameworks and details at the edge.
- **Test-Driven Development:** tests before or during implementation.
- **Error as Value:** explicit success or failure result, with exceptions confined to *infra*.
- **Separation of Concerns:** each layer with a single, well-defined purpose.

Workflow

Development follows the expected flow below:

1. Constitution
2. Spec
3. Plan
4. Tasks
5. Implementation
6. Tests
7. Review

Governance

This constitution supersedes other practices of the project: in case of conflict, it prevails.

- Amendments are versioned by **semantic versioning** (MAJOR for the incompatible removal or redefinition of a principle, MINOR for a new principle or section, PATCH for a clarification).
- Every amendment follows an **explicit process**: proposal, justification (rationale), and a record of the change in the version history.
- Changes **MUST** respect the **compatibility rule between versions**: incompatible changes require a MAJOR version bump and clear communication of the impact.
- The constitution **MUST** be reviewed whenever a **permanent architectural change** occurs, ensuring that the principles keep reflecting the reality of the project.

Version: 1.0.0 | Ratified: 2026-06-25 | Last Amended: 2026-06-25

5 - The complete SDD cycle

Before the first `specify` , a map

In the previous chapter you wrote the To-Do's constitution. The project's rules are set, the ground is prepared, and the temptation now is obvious: open the terminal, run the first `specify` , and start building. Hold that impulse for one more chapter.

Before laying the first brick, an experienced builder opens the house plan and traces the whole path with a finger: where you come in, how the rooms connect, where the plumbing runs. That time pays for itself: it is what keeps you from tearing down a wall later. With specification-guided software the same holds. Before taking on the first feature, it is worth climbing to a high point and seeing, from above, the path a feature travels in full: from nothing, when it is only an idea, to code running and integrated back into the project.

That is the map this chapter draws. The question it answers is simple to state and easy to underestimate: what journey does a feature travel, from the first sentence written about it to the line of code that delivers it? What are the points it passes through, in what order, and why that order and not another?

What comes out of here is the map: the drawing of the whole path, so that when you come down to the ground and start walking, you always know where you are and what comes next. Whoever sees the path from above doesn't get lost in the middle of it.

What you're learning is the cycle, not the app

Here is the thesis that holds up the rest of this material, and it is worth saying plainly: what you are learning is not the To-Do. It is the cycle.

The To-Do is a vehicle. It is deliberately simple, so that the mechanics of the flow never stay hidden behind the complexity of the problem. You will see it born feature by feature in the next chapters, but the app itself is disposable: nobody needs one more task app in the world. What is not disposable is the path each of its features travels, because that path is the same for any software. Swapping the To-Do for a banking system, a game, or a logistics dashboard changes the content of each step, but doesn't change the shape of the cycle. It is the shape you are learning.

And what is the unit of this cycle? The **feature**. Not the whole project at once, not a loose line of code, but the feature: a new, coherent capability the system comes to have. "Create and list tasks" is a feature. "Mark a task as done" is another. Each is a unit of work that is born as an idea, travels the whole cycle, and ends up integrated into the rest of the system, ready to use. When one reaches the end, the next restarts the same path from scratch.

That is why the cycle matters more than any specific feature. You are not going to memorize how to build "create and list tasks." You are going to internalize the path, and then you will be able to travel it with any feature, in any project, for the rest of your developing life. Memorizing commands is fragile; understanding the cycle is what stays.

The feature lives on a branch: `main → branch → merge`

This cycle doesn't happen in a vacuum. It happens inside git, which here stops being a technical detail and becomes the frame of the whole feature, from start to finish.

It works like this. Your project has a main line, the `main`: the stable version, the one considered good and sound at any moment. When you go to start a new feature, you don't touch `main` directly. You create a **branch**: a parallel, isolated line of work that starts from `main` and carries the feature's name. Think of it as a separate workbench, where you can saw, sand, and make mistakes freely without spreading sawdust in the main room. The whole cycle of the feature happens inside that branch.

When the feature is ready and sound, you do the **merge**: you join the branch's work back into `main`. `main` takes in the new feature and goes back to being the stable version, now a bit more complete. The branch has done its job and can be discarded. The next feature starts from `main` again, on a new branch, and the path restarts.

There are three edges easy to name: **leaving** `main` by creating the branch, **running the cycle** inside it, **merging back**. That is the frame, and it repeats identically for each feature.

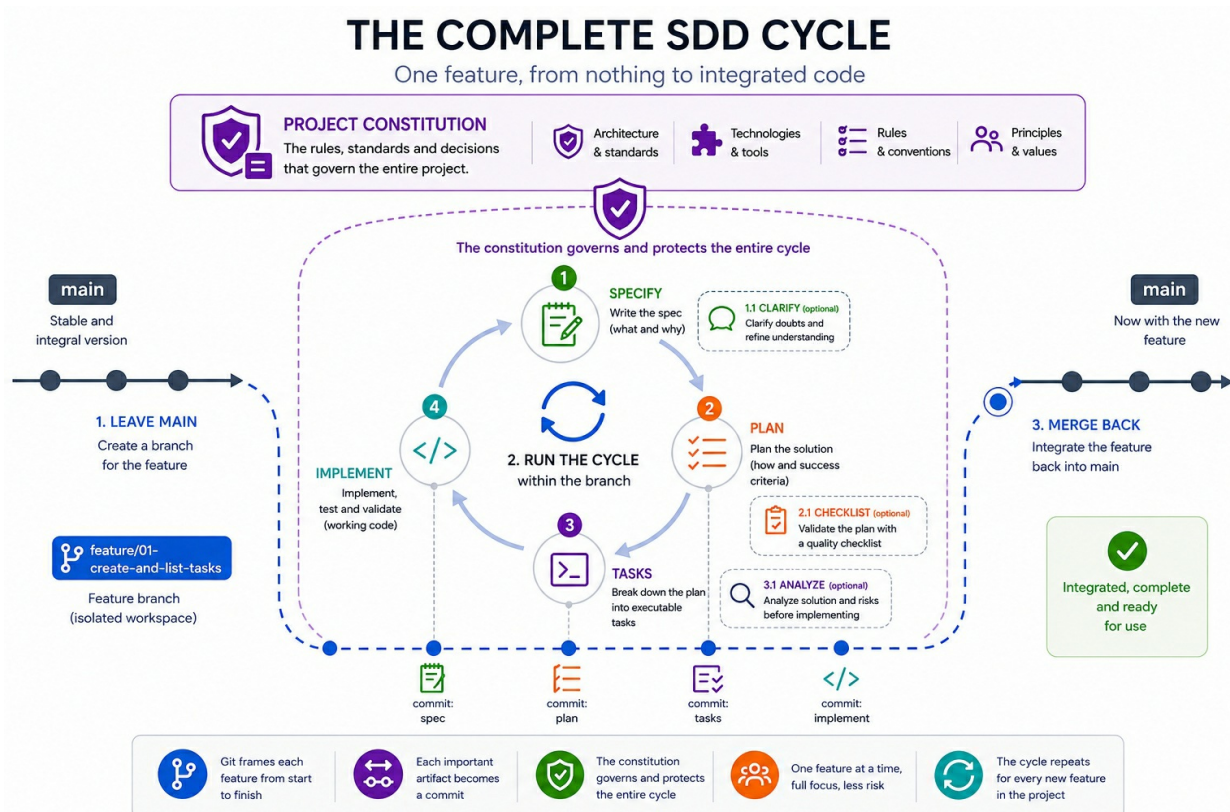
An honest caveat: technically spec-kit doesn't force you to work on branches; it is possible to run the cycle straight on `main`. The "one feature, one branch" frame is a recommendation of this material. But it isn't a loose recommendation: the tool itself ships a git extension that creates, for each new feature, a numbered branch with its name. The structure is optional, and even so the tool considers it valid enough to offer it ready-made. Adopting it is following the path spec-kit itself paves.

Inside the branch, the work doesn't become a single undistinguished block. Each important artifact the cycle produces becomes a commit, a saved point in the branch's

history. You will see further on that the core of the cycle produces four artifacts, and the rule of thumb is one commit per artifact: the branch comes to tell its own story, step by step, instead of dumping everything at once at the end. We won't turn this into a git tutorial; what matters is the shape. The feature is born on a branch, matures inside it in successive commits, and integrates through `main` at the merge. Keep that, and the rest of the map fits together.

The map in a single figure

Now put it all into a single image. The git you just saw and the cycle we are about to detail aren't two separate subjects: they are the same figure, seen from above. This is the diagram that anchors not only this chapter, but all the next ones. It is worth reading calmly once; after that it becomes a reference.



Read the diagram out loud once. At the top, the constitution, above everything, governing the cycles without being part of them. In the middle, `main` leaving on a branch and receiving the merge back. Inside the branch, the cycle: a straight line of four steps, the core, and three checks hanging off it as optional. At the bottom, the chain of artifacts that each step writes and the next reads. It is the whole chapter in one figure. Keep the image: the next chapters come back to it and only shine a spotlight on the step at hand, without redrawing the map.

The constitution sits above the loop

One thing jumps out of the diagram: the constitution is not inside the cycle. It is above it, and that is on purpose.

The constitution you wrote in the previous chapter is a once-per-project decision. You write it once, at the start, and it comes to govern all the features that will come. It isn't born again with each feature; it isn't a step that repeats. That is why it lives above the cycle, and not as one more step inside it. The general rules of the project are stable; what repeats is the building of each feature under those rules.

From that follows something important for reading the rest of the map: the cycle that repeats with each feature **starts at** `specify`, not at `constitution`. When you go to build the To-Do's first feature, you won't rewrite the constitution; it is already there. You go straight to specifying what you want. The `constitution` was the founding act of the project, prior to and above the repeated work.

But the constitution doesn't stay forgotten in a drawer after being written. It reappears inside the cycle, at a single, precise point: in the `plan`, as the **constitution check**. When you plan a feature and decide the technology and the approach, the flow

checks whether that plan respects the principles you fixed in the constitution. If the plan broke the layered architecture, or tried to smuggle in a decision the constitution forbids, it is in that check that the conflict shows up. The rules written once come back to demand conformance, always at the same spot in the cycle. Above the loop as governance, inside the loop only as a check: that is how the constitution takes part.

The loop, step by step

We reach the heart of the map. The cycle each feature travels has seven steps, in this order:

specify → clarify → plan → checklist → tasks → analyze → implement

Let's go through them one by one. The depth here is deliberately medium and even: each step gets enough for you to know what it does, what it produces, and when to run it, without any of them stealing the scene. The spotlight on each step, with the fine detail of how it behaves in practice, is what the next chapters will give, one step at a time. Here you are seeing the whole set.

Before the names, one piece of guidance that holds for all the steps: each of them is an **iterative conversation** with the agent, and never a button you press once. You run the command, read what came out, point out what turned out wrong or incomplete, ask for an adjustment, reread. Repeating a step's command to revise it is normal use of the flow, as legitimate as running it the first time. And the posture that most improves the result fits in an instruction worth giving the agent at any step: **when in doubt, ask; never assume**. An agent that asks hands the decision back to whoever it belongs to, you; an agent that assumes hides the decision inside the artifact, where it costs more to be found.

`specify` opens the cycle. It is where you describe the **what** and the **why** of the feature: the problem it solves, who benefits, what counts as done. It is the anatomy of the specification you saw in Chapter 2, applied to a concrete feature. Notice what `specify` doesn't do: it doesn't decide technology, it doesn't talk about code. It describes the need. The artifact it writes is the **spec**.

`clarify` comes right after. Every spec, however good, leaves loose ends: passages that allow more than one reading. `clarify` is the step that asks about what turned out ambiguous and **writes the answers back into the spec**, before any planning. Resolving the ambiguity now, on paper, costs a paragraph; resolving it later, in code, costs rework.

`plan` decides **the how**. It is here that technology enters: the language, the database, the framework, the architecture. The spec said the what; the `plan` decides with what and in what way to build. And it is inside the `plan` that the **constitution check** we saw runs: the plan is born already checked against the project's rules. The artifact written is the **plan**.

Right after the `plan` (and sometimes also after the `specify`), you will notice in the sessions an extra command running on its own: `/speckit-agent-context-update`. It updates the agent's context file (the `CLAUDE.md` or the equivalent of the assistant you use), pointing to the most recent plan, so that any future session starts already knowing which feature is under way and with which technical decisions. It is a maintenance step, run as an automatic hook; there is no decision of yours involved, and it is enough to know why it shows up.

`checklist` is a quality gate. Before breaking the work into tasks, it validates whether the spec's **requirements** are **complete and clear** enough to build on top of them, reading the plan as supporting context. Think of it as a unit test for the written

requirements: it doesn't check whether the code works, but whether the specification is well written. It is the question "is this ready to become work?" asked systematically, item by item.

tasks breaks the plan into **executable, ordered steps**: the concrete list of what to do, in the right sequence, with dependencies respected. Since this material adopts test-driven development, it is here that the tests enter the list, before the code they cover. A warning is worth it: spec-kit treats tests as **optional** and only includes them in the tasks when the TDD approach is asked for explicitly, whether by the constitution or by the spec itself. It isn't an automatic effect: it is a consequence of the project having declared that it wants tests. Since the To-Do's constitution adopts TDD, the test tasks show up; in a project that didn't ask for them, they simply wouldn't be generated. The artifact written is the set of **tasks**.

analyze is the last check before building. It checks the **consistency** among the three artifacts you already have: do the spec, the plan, and the tasks talk to one another? Does a task contradict the spec? Was a requirement left with no task to cover it? **analyze** catches that kind of mismatch before it becomes wrong code.

The **analyze** report is only worth something if you act on it. The correct practice is to **fix every point raised before running**

implement: ask the agent to apply each correction in the source artifact (the spec, the plan, or the tasks, as the case may be), run **analyze** again if the change was big, and only then release the build. Implementing with an open report of inconsistencies is paying to turn each inconsistency into code.

implement closes the cycle. It executes the tasks and produces the **code** that delivers the feature and passes the tests. It is the only step in which the software is actually born; all the previous ones exist so that this one is cheap, safe, and free of surprises.

The core and the three checks

Look again at the order and notice a skeleton inside it. Four steps form the **core loop**: the minimum path, without which there is no feature.

`specify → plan → tasks → implement`

Specify, plan, break into tasks, build. That is the irreducible route, and it is what produces the four versioned artifacts, one commit each. The other three, `clarify`, `checklist`, and `analyze`, are interleaved checks: control points strongly recommended, but optional with judgment. The default recommendation is to run them; the freedom is being able to omit them when the feature is trivial and crystal clear. Think of them as the safety net you decide to stretch according to the risk (the deep-dive box further on gives the practical criterion of when to skip each one).

And it is worth fixing what tells the three apart, because they are easy to confuse. `clarify` attacks **ambiguity** (what is misunderstood?). `checklist` attacks **completeness** (is everything here, ready to build?). `analyze` attacks **consistency** (do the pieces fit together?). Three different questions, at three different moments in the cycle. Whoever understands this division of labor never swaps one for another.

A validity warning, finally. These names and this order are spec-kit's at the date this material was produced. Tools evolve: a command may be renamed, the sequence may gain or lose a step. The stable path, the one worth learning, is the shape of the cycle: specify, clarify, plan, validate, break down, check, build. For the exact names and the options of each command in the version you have in hand, the source is always the official spec-kit documentation.

Deep dive (optional). The *constitution check* runs embedded in the `plan`, with no command of its own. In practice, when generating the plan, the flow opens the constitution, confronts each principle with the plan's decisions, and records the verdict in the plan's own artifact. That is why it appears in the diagram glued to the `plan`, and not as a loose box: it is a section of the plan, not a step of the cycle.

Deep dive (optional). The pairing between check and commit follows the logic of "one commit per core artifact". Since `clarify`, `checklist`, and `analyze` don't create a new artifact (they refine or check the existing ones), they don't get a commit of their own: `clarify` goes into the spec's commit, `checklist` into the plan's commit, `analyze` into the tasks' commit. The branch ends up with four clean commits, one per artifact, and each check travels along with the step it serves.

This commit structure is a choice of this material, not an obligation of the tool, but it is a choice spec-kit itself endorses. Its git extension offers automatic commits per step, with ready-made messages like "*Add specification*", "*Add implementation plan*", "*Add tasks*", and "*Implementation progress*": exactly one commit per core artifact. The checks, when they have an automatic commit, use messages like "*Clarify specification*", which add to the artifact that already exists instead of creating a new one. In other words: the convention is opinion, but opinion the tool considers valid enough to ship built in (off by default, you just turn it on).

Deep dive (optional). When to skip a check? A practical criterion: skip `clarify` only when the spec has no point you would reread twice; skip `checklist` only when the spec and plan are short and obvious; skip `analyze` only when there are few tasks and none touch a sensitive area. When in doubt, run it. The cost of a check is minutes; the cost of skipping the wrong one is redoing work already built.

The artifacts talk to each other; that's why you clear the context

There is a thread stitching all these steps together, and it is what turns the list of seven steps into a real system. Each step writes an artifact, and the next step reads that artifact and works from it. The spec feeds the `plan`; the plan feeds the `tasks`; the tasks guide the `implement`. It is a chain:

`spec → plan → tasks → code`

Notice the decisive detail: each step reads the **artifact** the previous one wrote, not the history of the conversation that produced it. The `plan` doesn't reread the chat that generated the spec; it reads the spec. The `tasks` don't reconstruct the reasoning of the planning; they read the plan. The state of the work doesn't live in the memory of the conversation with the agent. It lives in the versioned files. That is why you can switch models, and even agents, in the middle of a loop without losing anything: a more robust model for the complex step, a more economical one for the simple step, and the chain of artifacts stays whole.

From that follows the thesis that governs the way of working in SDD: **the documentation is the source of truth, not the chat.** What counts is what got written in the artifacts, because that is

what the next step will read. The conversation is the scaffolding that helped produce the artifact; once the artifact is good, the scaffolding can come down.

That is exactly why the `/clear` between steps, that habit the previous chapter asked you to adopt, is safe. Clearing the agent's context at the end of each step doesn't throw work away, because the work wasn't in the context: it was in the artifact, written and committed. The next step will restart by reading the file, without dragging along the dead ends of the previous conversation. And since each core step commits its artifact, the branch's own history is the proof that nothing was lost: at any moment, what matters is saved to disk, versioned, and not in the window of a conversation that grows and gets more expensive with each interaction. Assembling that window on purpose, deciding what goes into each call and at what price, belongs to *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering/>), the third volume in this trilogy, which answers what the agent sees right now, in the window of this one call, and at what cost. You do not need it here: the whole cycle in this book runs on the simple rule I just gave, clear the context at the end of each step and let the artifact speak for the next one.

The map is drawn: the bridge to Ch. 6

The map is complete: you have the whole path in your head, seen from above.

Now we come down to the ground. The next chapter travels this cycle for the first time, from start to finish, building the To-Do's first real feature: **create and list tasks**. And it travels it with a spotlight lit on the first step, `specify`. You will see, in detail and in practice, what it means to specify a feature well: how the spec is

born, what questions it answers, where it tends to fail. The other steps appear, because the cycle runs in full, but it is `specify` that gets the focus.

And that is the shape of the next chapters. Each one travels the complete cycle of a feature and shifts the spotlight one step ahead: one chapter lights up the `plan`, another the `tasks`, another the `implement`. The map you just kept doesn't change; what changes is the step under the light. That is why it was worth drawing it now, calmly, before walking.

Among those chapters are the ones with `.5` in the number. They redo the same lap file by file, with the real text of each artifact beside the code that came out of it. One of them deserves an explicit address right away: **Chapter 10.5** is the one that records the complete lap, from requirement to second iteration, because it was on the fifth feature that manual validation failed something and the fix had to travel back up to the spec before touching the code. If at any point you want to see the whole cycle at once, defect and repair included, that is where you go.

From the next chapter on, we walk.

6 - Create and list tasks: the first complete loop

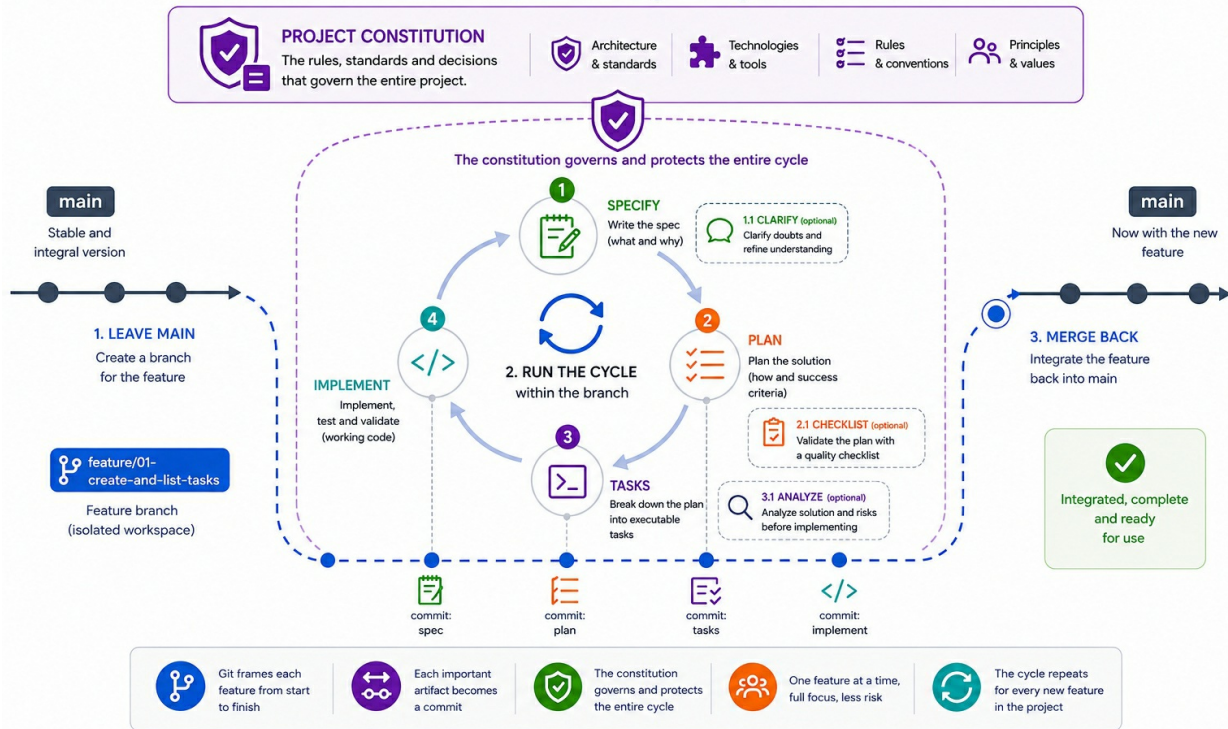
From the map to the ground: the first feature

In the previous chapter you climbed to a high point and saw the whole path from above. Now we come down to the ground and walk it for the first time. The map you kept still holds, word for word; what changes is that, from here on, each step stops being a drawing and becomes a file written to disk.

It is worth reopening the map one last time before the first step, because it is inside it that everything that follows happens:

THE COMPLETE SDD CYCLE

One feature, from nothing to integrated code



It is the same diagram from Chapter 5; what changes is that this time you step into it. The spotlight of this lap falls on the first step, **specify**, which had not yet been seen up close because in the previous chapter the whole map was in play.

The unit of work, you remember, is the **feature**: a new, coherent capability the system comes to have. The one you are going to build now is "create and list tasks," the first of the To-Do, the founding act without which there is nothing to complete, edit, or filter later. It is born as a sentence, travels the whole cycle, and ends as running code, integrated back into **main**.

Before the first command, a short reminder worth gold: have a git repository ready, with **main** in a stable and clean state. The whole cycle is going to happen inside git, and it only works well if there is firm ground underneath. You don't need to become a git

expert for this, you just need the repository initialized and `main` with no half-finished work. The first step of the cycle takes care of the rest: it creates the feature branch itself.

You will follow the whole path, from `specify` to `merge`, and you will see why the output of the first step weighs so much: it is where the feature stops being an idea and gains an outline, and it is that text every later step reads.

One feature at a time: the backlog

Look at the whole To-Do for a moment. It will need to create tasks, mark as done, edit, filter, and delete. Five capabilities. The temptation for whoever is in a hurry is to specify all five at once, "to get ahead." It is exactly what you don't do.

That queue of capabilities waiting has a name: the **backlog**, the list of what the system will still gain, in order of priority, without any of them being built before its turn. The backlog is where "complete," "edit," "filter," and "delete" stay kept, named and waiting, while you work on a single feature from start to finish.

Why one at a time? Because the feature is the unit of the cycle, and mixing several breaks that unit. A spec that tries to describe create, complete, and filter at the same time becomes a document that doesn't close: the title-uniqueness rule of creation bumps into editing, which also touches the title; the filter asks for states that completion hasn't even defined yet; each answer opens new questions in neighboring features. Specifying one at a time is the constitution's Simplicity and YAGNI (*You Aren't Gonna Need It*) in action: you solve the problem in front of you, with the scope the right size, and leave the rest in the queue until its time comes. The backlog keeps the other four in view, named and ordered, so you don't have to carry them in your head while working on the

first. No ceremonious document and no special tool: an ordered list of what comes next, plus the discipline of not pulling the next item before finishing the current one.

In practice, that backlog can be a plain text file in the repository. I use a `draft.md`: every time that, in the middle of a specification, I remember a feature the system will need ("it would be good to be able to archive old tasks"), the idea goes into `draft.md` on one line, and the specification in progress carries on without a detour. That gesture solves both sides of the problem: the idea isn't lost, and it also doesn't invade the spec of the wrong feature. When a lap around the cycle ends, `draft.md` is the queue the next specification comes out of. A different thing is remembering something that belongs to the **current** feature (a forgotten edge case, a requirement that was left out): that you don't note down for later; you remind the agent at the step where the hole is, asking it to include what was missing in the spec, the plan, or the tasks, as the case may be. An idea for another feature goes to the draft; a hole in the feature under way goes back into its artifact.

Maybe this sounds familiar: specifying everything at once is vibe-coding coming back through the back door, now disguised as a giant document. The backlog is the ordered queue of the slices waiting for the next lap, and it is what holds that temptation back.

And which to choose first? "Create and list" imposes itself. It is the base: without creating a task, there is nothing to complete; without listing, creating has no visible effect. The other four depend on this one existing. That is why it is first in the queue, and the rest wait in the backlog, each with its own loop ahead.

Spotlight: `specify` in action

Here the light comes on. The other steps will appear in this chapter, because the cycle runs in full, but it is on `specify` that the focus falls, because it is where the feature gains shape, and because everything that comes after reads what it writes.

You invoke the step by passing, in one sentence, what you want to build:

```
/speckit-specify Create and list tasks in a personal to-do list. The user can create a task by giving a title and a description, and can see the list of all tasks already created. On creation, two rules hold: the title is required (it can't be left blank) and the title can't repeat that of a task that already exists. Out of this feature: completing, editing, filtering, and deleting tasks, as well as categories, due dates, priority, users or login, and syncing .
```

Notice what that sentence already carries: what you want (create and see tasks), the rules (required and non-repeated title) and, with equal care, what stays out. Saying what doesn't go in is part of specifying well, and not a detail.

When it runs, the step does two things. First, it **creates the feature branch** (in our case, `001-criar-tarefa`, starting from `main`). From here on, all of the feature's work lives on that isolated branch. Second, it generates a first `spec.md`, the skeleton of the specification filled in from your sentence, following the tool's template.[1](#)

And here is the most important thesis of this step: **what comes out of `specify` is a draft, not the finished spec**. The tool organizes your sentence into a structure (scenarios, requirements, criteria), but the content is still yours to review. It gets the skeleton right and guesses at the flesh; you read, correct, complete. That is why, in this first loop, we are going to open that output more calmly than in the following steps.

It is worth seeing what that skeleton brings. For the first user story, `specify` didn't return just a title: it returned the story, its priority, an independent test, and the acceptance scenarios already written in the Given/When/Then format (*Given* a context, *When* someone does an action, *Then* such a result happens).

In the artifact, this appears under the label **User Story**: the *user story* you already met in Chapter 2. It is a short narrative, from the point of view of whoever uses the app: what the person wants to do and the result they expect. Don't confuse it with a *user journey*, which is the map of the path the person travels through the interface, UX/UI territory. In specification, the usual term is story: a small, verifiable slice of behavior, not the whole route through the screen.

```
### User Story 1 - Create a task (Priority: P1)
```

```
**Acceptance Scenarios**:
```

1. **Given** the task list (empty or with tasks), **When** the person creates a task with a filled-in and unique title, **Then** the task is created and comes to exist among the tasks.
2. **Given** the intent to create a task, **When** the title is given blank (empty or only spaces), **Then** the creation is refused with an error result explaining that the title is required, and no task is created.
3. **Given** an already existing task with a certain title, **When** the person tries to create another task with that same title, **Then** the creation is refused with an error result explaining that the title already exists, and no new task is created.

The **Given / When / Then** markers stay in English because they come from the template, but nothing in the document needs to be translated: you simply read them as *Given / When / Then*. In the

first scenario, for example: *Given* the task list, *When* the person creates a task with a filled-in and unique title, *Then* the task is created and comes to exist among the tasks.

This is more than a title and much less than the final truth. The skeleton got the shape right (the three scenarios that matter are there, in the right format), but it is you who checks whether they say what the problem demands. It was reading these scenarios that made it clear, for example, that something the original sentence didn't say was still to be decided: do the tasks need to survive closing the app, or is it enough for them to exist during the session? The draft exposed the question without answering it.

Reviewing that draft is different from rewriting it from scratch: it is running your eye over each part asking "is this true for my problem?". You read each requirement and check whether it matches the rule you have in your head; you flag what the skeleton assumed and you don't want; you add what was missing because your sentence didn't say it. That review is the iterative conversation Chapter 5 recommended, applied to the first step: point out each correction to the agent, ask for the adjustment in the spec, and, if you changed a lot, run `/speckit-specify` again over the result, without guilt, because repeating a step to revise it is normal use of the flow. It is a job of critical reading, quick when the feature is small like this one, and it is where your knowledge of the product, which the model doesn't have, enters the specification.

Optional box: why a draft and not the final version? An AI model doesn't know your product: it knows your sentence. The skeleton it fills in is a plausible hypothesis about what you meant, not the truth about what you need. The work of

reviewing the spec is where your knowledge of the problem enters, and it is exactly that work Chapter 7 is going to light up, with the `clarify` step.

The anatomy from Ch. 2, actually filled in

In Chapter 2 you saw the anatomy of a good specification in the abstract: the problem and the intent, the scope, the non-goals, the scenarios, the rules, and the acceptance criteria. Now the same anatomy appears filled in for a concrete feature, like opening the hood of a car that runs after studying the engine diagram. We won't re-explain each part, we will recognize it in the real artifact.

The **scope** is minimal and explicit: create a task with a title and a description, and see the list of all of them. The central entity too:

```
- Task: represents a to-do recorded by the person. Essential attributes:  
a title  
  (required and non-repeated among existing tasks) and a description (free text, optional).
```

The **business rules** are two, and the spec fixes them as verifiable requirements. The **FR** prefix that numbers each one comes from *Functional Requirement*: a testable statement of what the system needs to do. The numbering (FR-001, FR-002...) only serves to reference each requirement without ambiguity across the other artifacts.

The spec has seven requirements in total, and throughout this chapter we will look at each one at the moment it matters, instead of dumping the whole list at once. **FR-001** is the basic

capability (create a task from a title and an optional description), which already appeared in the scope above; listing and persistence come further on. The **two rules** that interest us now are the constraints that come right after:

- **FR-002**: The system MUST refuse the creation when the title is blank (empty or made up only of spaces), returning an explicit error result that identifies the required title as the cause, without creating the task.
- **FR-003**: The system MUST refuse the creation when a task with the same title already exists (compared after trimming spaces at the ends), returning an explicit error result that identifies the duplication as the cause, without creating the task.

Notice something subtle and important in those two requirements: they don't say "throw an error" or "raise an exception." They say "returning an explicit error result." That is the To-Do's constitution reflected in the spec: the principle of **error as value**, which you fixed back there, showing up here as the natural way to describe the two predictable failures of creation. The spec didn't invent exotic failure flows; it just named the two cases the rule itself produces and said both are expected results, never accidents. One more requirement makes this explicit for any reader:

- **FR-004**: The system MUST represent the predictable creation failures (missing required title; duplicate title) as explicit success or error results, and MUST NOT signal them as exceptions across layers.

And the **non-goals**, which so many specs forget, are written in so many words:

- Explicitly out of scope: **categories**, due dates, priority, users/login, and syncing **(YAGNI).**

The **acceptance scenarios** already appeared in the previous step's skeleton, written in Given/When/Then: it is where each abstract rule becomes a concrete situation that can be staged. And the **acceptance criteria** close the anatomy by turning each rule into a measurable result that a test can verify with no room for interpretation. They come prefixed with **SC**, for *Success Criteria*, numbered like the requirements (SC-001, SC-002...); again, we show only the ones that matter for this section's rules:

- **SC-002**: An attempt to create a task with a blank title is refused with a clear error result, and the task count doesn't change.
- **SC-003**: An attempt to create a task with an already existing title is refused with a clear error result, and the task count doesn't change.
- **SC-005**: Tasks created in a session keep appearing in the list after the app is closed and reopened.

Notice that each criterion is observable from the outside: "the task count doesn't change," "keep appearing." None of them talks about code, class, or database; they talk about what the person using the app can verify with their own eyes. It is the what, never the how, taken to the point of becoming an acceptance test that either passes or doesn't.

Each part of Chapter 2's anatomy has a counterpart here: the problem in the task story, the scope in the entity and the scenarios, the rules in the FRs, the non-goals in the backlog queue, the acceptance criteria in the measurable SCs. The specification stopped being a mold and became the document of a feature that exists.

What is spec and what is `plan`: the boundary

There is a question that decides whether your spec will age well or turn into a straitjacket: what goes into the specification and what is left for the `plan`? The rule you already know from Chapter 2 (the spec describes the what and the why; the `plan` decides the how); what this first loop adds is the practical test of applying it to real sentences. Take some candidates and classify each one:

- "The task needs a title" → **spec**. It is the what: a rule of the problem, true in any technology.
- "The list shows the existing tasks, from oldest to newest" → **spec**. Again the what: the observable behavior, independent of implementation.
- "Store the tasks in `localStorage`" → `plan`. It is the how: a decision about the storage medium.
- "Use React and Zustand" → `plan`. Pure how: language, framework, state pattern.

A mental test resolves any doubtful sentence: would it still be true if you swapped the entire technology? "The task needs a title" keeps holding in React, in Flutter, in a terminal app, or on paper: it is spec. "Use Zustand for the state" disappears the instant you switch frameworks: it is `plan`. If the sentence

survives the stack swap, it describes the problem; if it dies along with the technology, it describes the solution, and its place is in the plan.

See the real specification following that boundary to the letter. About persistence, it says the what and declares, in its own text, that the how is a decision of another step:

```
- **FR-006**: The system MUST preserve the created tasks between sessions of use, so that they keep appearing when you see the list after the app is closed and reopened. The storage medium is decided in the Plan.
```

The final sentence of that requirement is the boundary drawn inside the document: "the storage medium is decided in the Plan." The spec requires the tasks to survive closing and reopening, but doesn't say a word about a file, database, or `localStorage`. That keeps the specification technology-agnostic, as the constitution's Technology Agnosticism demands, and leaves the ground free for the `plan` to choose the stack without rewriting the problem.

Optional box: why does that boundary matter so much?

When the "how" leaks into the spec, you tie the problem to a solution too early. Switch the database, the framework, or the state pattern, and a contaminated spec has to be rewritten along with it. A spec that talks only about the what and the why survives all those switches, because it describes something that didn't change: what the user needs.

That boundary isn't an invention of SDD; it just names something that always existed in software development. Think of the conversation between a systems analyst and the client who commissioned the system. The client talks about their problem (what they need to record, which rule can't be broken, what counts as done) and almost never knows, nor cares to know, whether it will run on Postgres or SQLite, in React or in Flutter. Stack is the domain of whoever builds; the analyst themselves, when they know the subject, understands it at a high level. The spec is the client's side of that conversation: the what and the why, in the language of the problem. The `plan` is the developer's side: the how, in the language of the solution. In a small project like the To-Do both voices are yours, but in a more complex environment they are different people, in different roles, and keeping the spec technology-agnostic is exactly what lets those two voices talk without one invading the other's territory.

The rest of the loop, at a follow-along pace

With the spec reviewed, the spotlight goes off and the other steps pass, each reading the artifact the previous one wrote. Here the pace is follow-along: what ran, what went in, what came out.

`clarify` came right after the spec, asking about what it had left ambiguous. Two questions, and their answers were written back into the spec itself:

- Q: Should the created tasks survive closing and reopening the app, or is it enough for them to exist during the session? → A: The tasks are preserved between sessions of use.
- Q: In what order does the list present the existing tasks? → A: In order of creation, from oldest to newest.

That step will have its own spotlight in Chapter 7; for now, it is enough to see that it exists to resolve ambiguity before it becomes a wrong decision in the plan.

plan read the clarified spec and decided the how: React with TypeScript in the interface, Zustand for the state, **localStorage** behind a repository layer for persistence. It is here too that the *constitution check* runs, and the plan came out reflecting the To-Do's constitution: the business rule isolated in a domain layer, independent of the interface; the error as value materialized in a result type; the layers separated, with composition happening only at the edge.

checklist validated whether the spec and plan were complete and clear before becoming tasks. **tasks** broke the plan into an ordered list of small tasks and, faithful to the constitution's TDD, each implementation block comes after its test. You can see this literally in the list, in the pair that handles the creation rule:

```
- **T009** test/domain/create-task.test.ts: tests for CreateTask (success; empty title; only spaces; duplicate; optional description). Fails first.  
- **T010** src/domain/usecases/create-task.ts: implement until T009 passes.
```

The test task is numbered before the code task, and it says "fails first": you write the test, watch it fail for lack of the implementation, and only then write the code that makes it pass. The constitution asked for TDD; **tasks** translated the principle into execution order, and **implement** followed that order. The same happened with the layers: the plan put the business rule in a domain layer that imports nothing from the interface, and **localStorage** in a data layer behind a repository, so that swapping React for something else, or **localStorage** for a database, doesn't

touch the rule. Why the dependency runs in that direction and not the opposite one belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it to follow this lap: what matters here is that the decision came out of the plan, not out of improvisation. The constitution wasn't reopened for discussion; it appeared, already applied, inside the artifacts.

`analyze` checked the consistency among the three artifacts. And `implement` ran through the tasks until the code existed for real, with the tests green.

Between one step and the next, the `/clear`: that habit of clearing the agent's context at the end of each step, safe for the reason the previous chapter fixed. The clarified spec is in the file; the plan is in the file; the tasks are in the file. Between the `plan` and the `tasks`, for example, you can erase everything the agent "knew" about the plan discussion without worry, because `tasks` doesn't need that conversation: it needs the written `plan.md`.

At the end of that sequence, the feature that started as a sentence is code that runs: it creates tasks, refuses the invalid ones with a clear error result, lists what exists in the right order, and survives closing and reopening the app.

Commit, merge, and the bridge to Ch. 7

The feature is ready on the `001-criar-tarefa` branch, with its tests passing. What is missing is the gesture that closes the cycle: integrating back. You commit the work and do the `merge` into `main`. The feature branch has done its job; `main` goes back to being

the stable version, now with one more capability than it had before. The repository is sound, and the To-Do's first feature exists for real.

That gesture carries weight beyond the symbolic. While the feature lived on the branch, it could be half-done without getting in anyone's way; on entering `main`, it becomes part of the version considered good, and that is why it only crosses that door with the tests green. `main` keeps being what it always was: the place where nothing is half-done.

It is the whole lap around the diagram: `main → branch → cycle → merge → main`. We left a stable `main`, opened a branch for a feature, traveled the cycle from `specify` to `implement`, and came back to a stable `main` again. The map from Ch. 5 stopped being a drawing and became history written on the branch.

And the next step? The backlog is there, waiting. The next feature is **completing a task**: marking as done what you recorded here. The cycle will be the same, from `specify` to `merge`; what changes is the step under the light. In Chapter 7, the spotlight shifts one square ahead, to `clarify`, and you can already feel why it deserves its own spotlight: it was `clarify` that caught the ambiguity of persistence (survive closing the app, or not?) that the spec, on its own, had left open. Here that step passed quickly, in the follow-along; there, it becomes the center. Same path, new spotlight. We keep walking.

Footnotes

Command names, the order of the steps, and the exact form of invocation may evolve between versions of spec-kit; the excerpts in this chapter were generated with version 0.11.8. For details that change per release (flags, subcommands, internal paths), check the official spec-kit documentation instead of fixing them from memory.

6.5 - Creating and listing tasks in practice: the whole loop, file by file

How to read this chapter

Chapter 6 followed the first lap around the cycle with the spotlight on `specify`. Here you see the same lap in full, but from the other side: not the didactics of the step, but what got written to disk when it ended. Everything that follows is a faithful reproduction of what is versioned in the example app's repository, in the `001-criar-tarefa` feature: the prompt that went in, the specification that came out, the plan, the design artifacts, the tasks, and each code and test file the agent produced.

This is a reference chapter, made to come back to when you need it, more than to read in one sitting. The idea is that you can, at any moment, compare what you asked for with what you received, and see how a sentence becomes a layered system without anyone deciding that along the way. The other fractional-numbered chapters (7.5, 8.5, 9.5, and 10.5) do the same for the following laps, and it is by comparing one with another that what matters most becomes visible: the architecture doesn't change from one feature to the next.

The specification and plan files appear as running text, the way Spec Kit wrote them (the internal titles were lowered one level so as not to compete with the material's titles). The code appears in

blocks, each preceded by a short sentence saying what it is and why it exists.

The input: the `/speckit-specify` prompt

The whole lap starts with a sentence. It doesn't have to be born finished: in the flow Chapter 6 described, sentences like this come out of the backlog (the `draft.md` where the ideas waited their turn), polished at the moment of pulling the feature from the queue. This was the one, recorded literally in the specification's `Input` field:

"Create and list tasks in a personal to-do list. The user can create a task by giving a title and a description, and can see the list of all tasks already created. On creation, two rules hold: the title is required (it can't be left blank) and the title can't repeat that of a task that already exists. Out of this feature: completing, editing, filtering, and deleting tasks, as well as categories, due dates, priority, users or login, and syncing."

Notice that the sentence describes behavior and boundaries, without a word of technology: no React, no database, no "text field." That is exactly the cut `specify` demands, and it is from it that the document below comes.

What `specify` and `clarify` returned: `spec.md`

The specification is the first written artifact. The two `clarify` questions (persistence between sessions and the order of the list) already come folded into the `Clarifications` section, and each

decision became a traceable functional requirement.

Feature Specification: Create and list tasks

Feature Branch: 001-criar-tarefa

Created: 2026-06-27

Status: Draft

Input: User description: "Create and list tasks in a personal to-do list. The user can create a task by giving a title and a description, and can see the list of all tasks already created. On creation, two rules hold: the title is required (it can't be left blank) and the title can't repeat that of a task that already exists. Out of this feature: completing, editing, filtering, and deleting tasks, as well as categories, due dates, priority, users or login, and syncing."

Clarifications

Session 2026-06-27

- Q: Should the created tasks survive closing and reopening the app, or is it enough for them to exist during the session? → A: The tasks are preserved between sessions of use: they survive closing and reopening the app. The storage medium is a Plan decision (Technology Agnosticism).
- Q: In what order does the list present the existing tasks? → A: In order of creation, from oldest to newest.

User Scenarios & Testing (mandatory)

User Story 1 - Create a task (Priority: P1)

The person using the To-Do wants to record something they need to do. They give a title and, if they want, a description, and the task comes to exist in the list. It is the founding act of the app: without creating, there is nothing to list nor, later on, to complete, edit, or filter.

Why this priority: It is the core of the feature and the base of all future features. Delivered on its own, it already produces value: the person can capture tasks.

Independent Test: It can be tested in isolation by creating a task with a valid title and verifying that it comes to appear among the existing tasks.

Acceptance Scenarios:

1. **Given** the task list (empty or with tasks), **When** the person creates a task with a filled-in and unique title, **Then** the task is created and comes to exist among the tasks.
2. **Given** the intent to create a task, **When** the title is given blank (empty or only spaces), **Then** the creation is refused with an error result explaining that the title is required, and no task is created.
3. **Given** an already existing task with a certain title, **When** the person tries to create another task with that same title, **Then** the creation is refused with an error result explaining that the title already exists, and no new task is created.
4. **Given** the creation of a task with a valid title, **When** the description is left blank, **Then** the task is created normally (the description is optional).

User Story 2 - See the task list (Priority: P1)

The person wants to see everything they have recorded so far. They open the list and see the existing tasks, with their titles and descriptions. It is the visible return of the act of creating and the starting point of the following features.

Why this priority: Without seeing the list, creating a task has no noticeable effect. Together with creation, it forms the minimum pair that makes the To-Do usable.

Independent Test: It can be tested in isolation by verifying that the list presents all the tasks already created and that, when there are none, the list presents itself empty.

Acceptance Scenarios:

1. **Given** one or more tasks already created, **When** the person sees the list, **Then** all the existing tasks appear, each with its title and its description, in the order they were created.
2. **Given** no task created, **When** the person sees the list, **Then** the list presents itself empty, with no error.
3. **Given** tasks created in a previous session, **When** the person closes and reopens the app and sees the list, **Then** the tasks created before are still present.

Edge Cases

- **Title with only spaces:** a title made up only of blank spaces is treated as empty and refused by the required-title rule.
- **Duplicate title:** the duplication comparison ignores spaces at the ends of the title; two titles that are equal after that adjustment are considered the same title.
- **Missing description:** the description is optional; its absence doesn't prevent creation.
- **Empty list:** seeing the list with no task created is a valid state, not an error.

Requirements (mandatory)

Functional Requirements

- **FR-001:** The system **MUST** allow creating a task from a title and an optional description.
- **FR-002:** The system **MUST** refuse the creation when the title is blank (empty or made up only of spaces), returning an explicit error result that identifies the required title as the cause, without creating the task.
- **FR-003:** The system **MUST** refuse the creation when a task with the same title already exists (compared after trimming spaces at the ends), returning an explicit error result that identifies the duplication as the cause, without creating the task.
- **FR-004:** The system **MUST** represent the predictable creation failures (missing required title; duplicate title) as explicit success or error results, and **MUST NOT** signal them as exceptions across layers.
- **FR-005:** The system **MUST** allow seeing the list of all existing tasks, presenting, for each one, its title and its description, in order of creation (from oldest to newest).
- **FR-006:** The system **MUST** preserve the created tasks between sessions of use, so that they keep appearing when you see the list after the app is closed and reopened. The storage medium is decided in the Plan.
- **FR-007:** The business rule of creation and listing **MUST** reside in a layer of its own, independent of the user interface; the interface only collects the input and presents the result.

Key Entities (include if feature involves data)

- **Task:** represents a to-do recorded by the person. Essential

attributes: a **title** (required and non-repeated among existing tasks) and a **description** (free text, optional). There is, in this feature, no completion state, due date, priority, category, or owner.

Success Criteria (mandatory)

Measurable Outcomes

- **SC-001:** The person creates a task with a valid and unique title and finds it among the existing tasks when seeing the list.
- **SC-002:** An attempt to create a task with a blank title is refused with a clear error result, and the task count doesn't change.
- **SC-003:** An attempt to create a task with an already existing title is refused with a clear error result, and the task count doesn't change.
- **SC-004:** When seeing the list, the person finds all the tasks they created, each with a title and description, in order of creation, and a list with no tasks presents itself empty without error.
- **SC-005:** Tasks created in a session keep appearing in the list after the app is closed and reopened.

Assumptions

- The To-Do is a **personal, single-user** app; there are no users, login, or separation of data by owner in this feature.
- The feature is the **first** of the app; "completing," "editing," "filtering," and "deleting" tasks are features of their own, out of this scope, each with its own cycle.
- Explicitly out of scope: **categories, due dates, priority, users/login, and syncing** (YAGNI).

- The choices of **language, framework, database, interface architecture, and state management pattern** don't belong to this specification; they are decided in the Plan (Technology Agnosticism).
 - The title-duplication comparison considers the text of the title after trimming spaces at the ends; further refinements (for example, ignoring case differences) are not assumed in this version.
-

The requirements check:

`checklists/requirements.md`

Before planning, a checklist checks whether the specification is ready to become a plan. It works like an automated peer review of the spec itself: each item points to something that needed to be resolved before any line of code.

Checklist: Requirements Quality

Check of `spec.md` before implementing. All items must be checked.

- Each functional requirement (FR-001..FR-007) is verifiable and unambiguous.
- The two business rules (required title; non-repeated title) are explicit, with corresponding acceptance scenarios.
- The predictable failures are described as an error result, not an exception (FR-004).
- The listing order is defined (creation, oldest first, FR-005).
- Persistence between sessions is defined without fixing technology (FR-006).

- The non-goals are explicit (no completing/editing/filtering/deleting, categories, due dates, priority, login, sync).
 - The spec stays agnostic about language/framework/database/state (decisions in the plan).
 - Each Success Criterion (SC-001..SC-005) is measurable and traceable to an FR.
-

What `plan` decided: `plan.md`

It is in the plan, and only in it, that technology enters. Note the Constitution Check table: each principle of the project's constitution is confronted with a concrete decision. It is that document that anchors the architecture all the following laps will inherit.

Implementation Plan: Create and list tasks

Branch: `001-criar-tarefa` | Spec: `spec.md`

Input: Feature specification at `specs/001-criar-tarefa/spec.md`

Summary

Implement the To-Do's first feature, create a task (required and unique title, optional description) and see the list of all tasks in order of creation, with persistence between sessions. The business rule stays isolated in a domain layer (pure TypeScript), with the predictable failures represented as **error as value**; the React interface only collects input and presents the state.

Technical Context

Language/Version: TypeScript 5.x on Node 20+; execution target: browser (React web).

Main dependencies:

- **React 18**, UI library (web, not React Native).
- **Vite**, bundler and dev server.
- **Zustand**, presentation state management (a thin store that orchestrates the domain use cases).

Persistence: browser `localStorage`, behind the domain's `TaskRepository` interface (web equivalent of "persist on the device," FR-006). The domain doesn't know `localStorage`.

Tests: Vitest for the domain (use cases, no DOM) and for the store; React Testing Library optional for the components.

Target platform: modern browser (single-page app).

Project type: single-page web application, single-user, no backend.

Constitution Check

Principle	How the plan meets it
I. Layered Architecture	<code>domain</code> (pure) $\leftarrow$ <code>data</code> + <code>presentation</code> ; composition only at the edge (<code>main.tsx</code>).
II. Isolated Business Logic	Creation/listing rules live in pure TS use cases; React and Zustand decide nothing.
III. Error as Value	<code>Result<S, F></code> and <code>CreateTaskFailure</code> (discriminated union); no exceptions across layers. Exceptions only at the I/O

	<code>edge (localStorage).</code>
IV. TDD	Use case and store tests written before the implementation (see tasks.md).
V. Simplicity/YAGNI	No categories, due dates, priority, login, or sync; minimalist Zustand store.
VI. Technology Agnosticism	The spec doesn't mention React/Zustand; the stack is decided here, in the plan.

Result: PASS, no violation; no entry in the complexity table.

Project Structure

```

todo-app/
├── index.html
├── package.json
├── tsconfig.json
├── vite.config.ts
├── src/
│   ├── domain/
│   │   ├── result.ts                # Result<S, F> (Success | Failure)
│   │   ├── entities/task.ts        # Task type
│   │   ├── failures/create-task-failure.ts
│   │   ├── repositories/task-repository.ts  # interface (port)
│   │   └── usecases/
│   │       ├── create-task.ts
│   │       └── list-tasks.ts
│   ├── data/
│   │   └── local-storage-task-repository.ts  # implements TaskRepository
│   ├── presentation/
│   │   └── store/task-store.ts        # Zustand store (orchestrates use cases)
│   └── components/
│       ├── TaskForm.tsx
│       ├── TaskList.tsx
│       └── App.tsx
├── main.tsx                        # composition: repo → use cases → store
└── test/
    └── UI

```

```
├── domain/
│   ├── create-task.test.ts
│   └── list-tasks.test.ts
├── presentation/
│   └── task-store.test.ts
└── helpers/in-memory-task-repository.ts
```

Phase 0 - Research

Technical decisions recorded in research.md.

Phase 1 - Design

- Domain entities and contracts in data-model.md.
- Contracts of the ports and use cases in contracts/.
- Manual validation in quickstart.md.

The `plan`'s design artifacts

The `plan` doesn't produce only the `plan.md`. It generates a small set of supporting documents that detail the decisions: why each technology, what the exact shape of the data is, what each piece's contract is, and how to validate by hand. They are what make the `implement` almost mechanical afterward.

The technical decisions: `research.md`

Each technology decision is recorded with the chosen option, the justification, and what was discarded. It is the record of the "why," not just the "what."

Research: Create and list tasks

Technical decisions that hold up the plan.md. Each decision records the chosen option, the justification, and the discarded alternatives.

D1 - State management: Zustand

- **Decision:** use Zustand as the presentation store.
- **Justification:** minimalist (store as a hook, near-zero boilerplate), popular in the React ecosystem, and aligned with Simplicity/YAGNI (Principle V). The store only orchestrates the domain use cases and exposes the state to the UI.
- **Discarded alternatives:** Redux Toolkit (more ceremony than this feature demands); Context + useReducer (would reinvent part of what Zustand already delivers).

D2 - Error as value: discriminated union `Result<S, F>`

- **Decision:** represent success/failure with `Result<S, F> ( Success | Failure )` and the creation failures with `CreateTaskFailure ( TitleRequired | DuplicateTitle )`.
- **Justification:** it makes the predictable failures part of the use cases' signature (Principle III); TypeScript forces exhaustive handling via the discriminated union.
- **Discarded alternatives:** throwing exceptions across layers (forbidden by the constitution); returning `null` /boolean (loses the cause of the failure).

D3 - Persistence: `localStorage` behind the interface

- **Decision:** `LocalStorageTaskRepository` serializes the tasks as JSON in `localStorage`, implementing the domain's `TaskRepository` interface.

- **Justification:** it meets FR-006 (survive closing/reopening) without a backend; it is the web equivalent of "persist on the device." The domain doesn't know `localStorage`.
- **Discarded alternatives:** IndexedDB (overkill for a simple single-user app); in-memory state only (doesn't survive the reload).

D4 - Build and test tool: Vite + Vitest

- **Decision:** Vite as the bundler/dev server and Vitest for tests.
- **Justification:** the current standard for React web; Vitest runs the domain tests without a DOM and integrates with the same Vite configuration.
- **Discarded alternatives:** Create React App (discontinued); Jest standalone (extra configuration for ESM/TS that Vitest avoids).

D5 - Ordering by creation

- **Decision:** order the tasks by `createdAt` ascending when reading from the repository.
- **Justification:** it meets FR-005/SC-004 (oldest first) in a stable way, independent of the serialization order.
- **Discarded alternatives:** relying on the JSON insertion order (fragile to external edits of the storage file).

The shape of the data: `data-model.md`

The domain model in pure TypeScript: the entity, the error-as-value types, the use case signatures, the persistence repository, and the direction of the dependencies.

Data Model: Create and list tasks

Domain model (pure TypeScript), independent of React, Zustand, and `localStorage`.

Entity: Task

Field	Type	Rules
<code>id</code>	<code>string</code>	Identity generated on creation.
<code>title</code>	<code>string</code>	Required; non-repeated (compared after <code>trim</code>).
<code>description</code>	<code>string</code>	Optional (empty string when absent).
<code>createdAt</code>	<code>string</code> (ISO 8601)	Moment of creation; defines the display order.

```
export type Task = {  
  
  readonly id: string;  
  
  readonly title: string;  
  
  readonly description: string;  
  
  readonly createdAt: string;  
  
};
```

Error as value

```
// Generic Result

export type Result<S, F> =

  | { readonly kind: "success"; readonly value: S }

  | { readonly kind: "failure"; readonly error: F };

// Predictable creation failures

export type CreateTaskFailure =

  | { readonly kind: "title-required" }

  | { readonly kind: "duplicate-title" };
```

Use cases

- **CreateTask:** (input: { title: string; description?: string }) => Promise<Result<Task, CreateTaskFailure>>
 1. `trim` the title; if empty → `failure(title-required)`.
 2. If a task with the same title already exists (after `trim`) → `failure(duplicate-title)`.
 3. Otherwise, create `Task` and persist → `success(task)`.

- **ListTasks:** `() => Promise<Task[]>`, returns all the tasks in order of creation (oldest first). An empty list is a valid state.

Port: TaskRepository

```
export interface TaskRepository {  
  
  getAll(): Promise<Task[]>; // ordered by createdAt asc  
  
  add(task: Task): Promise<void>;  
  
}
```

Direction of the dependencies

```
presentation (React + Zustand)      data (localStorage)  
    \                               /  
     v                             v  
      domain (Task, Result, use cases, TaskRepository)
```

The domain imports nothing from `presentation` or from `data`. The composition (instantiating the concrete repository and injecting it into the use cases) happens only in `main.tsx`.

The contracts: `contracts/`

The contracts describe the expected behavior of each piece before there is any code. There are two: the one for the use cases and the one for the persistence repository.

Contract: CreateTask and ListTasks use cases

CreateTask

```
createTask(input: { title: string; description?: string })  
  
: Promise<Result<Task, CreateTaskFailure>>
```

Applies the two business rules in the order below and returns error as value:

1. **Required title (FR-002):** after `trim`, if the title is empty → `failure({ kind: "title-required" })`. No task is created.
2. **Non-repeated title (FR-003):** if a task whose title, after `trim`, is equal already exists → `failure({ kind: "duplicate-title" })`. No task is created.
3. **Success (FR-001):** creates `Task` with generated `id` and `createdAt`, optional `description` (empty string when absent), persists via `TaskRepository.add`, and returns `success(task)`.

Invariant: creation NEVER throws an exception for a rule violation; the failures are typed values (Principle III).

ListTasks

```
listTasks(): Promise<Task[]>
```

- Returns all the tasks in **order of creation** (oldest first, FR-005).

- An empty list is a valid state, not an error (SC-004).
 - Doesn't filter, doesn't paginate, doesn't order by another criterion (YAGNI).
-

contracts/task-repository.md

Contract: TaskRepository (port)

Interface declared by the domain and implemented by the data layer.

```
export interface TaskRepository {  
  
  getAll(): Promise<Task[]>;  
  
  add(task: Task): Promise<void>;  
  
}
```

getAll

- **Returns:** all the persisted tasks, **ordered by** `createdAt` **ascending** (oldest first).
- **Empty list:** valid state; returns `[]`, never an error.

add

- **Receives:** a `Task` already validated by the use case (required and unique title already guaranteed by `CreateTask`).

- **Effect:** persists the task so that it survives closing/reopening the app (FR-006).
- **Doesn't validate business rules:** the title's uniqueness is the use case's responsibility, not the repository's.

Implementations

- `LocalStorageTaskRepository` (production): serializes as JSON in `localStorage`.
- `InMemoryTaskRepository` (tests): keeps the tasks in an in-memory array.

Both respect the same contract, including the ordering by `createdAt`.

The manual validation: `quickstart.md`

Finally, the plan leaves a script for validating by hand, mapped to the Success Criteria. It is what a person would do to check, without automated tests, that each criterion actually happens on the screen.

Quickstart: Create and list tasks

Run

```
npm install
```

```
npm run dev      # opens the app in the browser (Vite)
```


Test

```
npm test          # runs the domain and store tests (Vitest)

npm run build     # type-check + production build
```

Manual validation (mapped to the Success Criteria)

1. **SC-001:** create a task with a valid and unique title → it appears in the list.
 2. **SC-002:** try to create with a blank title → "required title" error; the list doesn't change.
 3. **SC-003:** try to create with an already existing title → "title already exists" error; the list doesn't change.
 4. **SC-004:** see the list with tasks → they all appear with a title and description, in order of creation; with no tasks → empty list, no error.
 5. **SC-005:** create tasks, reload the page (close/reopen) → the tasks stay in the list (persistence in `localStorage`).
-

The execution list from `tasks` : `tasks.md`

`tasks` turns the plan into an ordered sequence, in TDD order: each test comes before the implementation that makes it pass. The `[P]` marks tasks that could run in parallel because they touch distinct files. The `[X]` are the record that the lap was completed in full.

Tasks: Create and list tasks

Branch: `001-criar-tarefa` | Plan: plan.md

TDD order: tests before the corresponding implementation. `[P]` marks tasks that can run in parallel (distinct files, no dependency).

Phase 1 - Setup

- **T001** Scaffold React + TS (Vite): `package.json`, `tsconfig.json`, `index.html`, `src/main.tsx`, `src/index.css` in the plan.md structure.
- **T002** Add dependencies: `zustand`; dev: `vitest`. (Domain/store tests run in a `node` environment, no DOM, so React Testing Library/jsdom weren't needed.)
- **T003** Configure build and tests: `vite.config.ts` (React plugin) and `vitest.config.ts` (`environment: node`); `dev` / `build` / `test` scripts in `package.json`.

Phase 2 - Domain (core, pure TS)

- **T004** [P] `src/domain/result.ts` : `Result<S, F>` type with `success` / `failure` helpers.
- **T005** [P] `src/domain/failures/create-task-failure.ts` : `CreateTaskFailure` union.
- **T006** [P] `src/domain/entities/task.ts` : `Task` type.
- **T007** [P] `src/domain/repositories/task-repository.ts` : `TaskRepository` interface.
- **T008** `test/helpers/in-memory-task-repository.ts` : fake that implements `TaskRepository` (orders by `createdAt`).
- **T009** `test/domain/create-task.test.ts` : tests for `CreateTask` (success; empty title → `title-required` ; only spaces → `title-required` ; duplicate → `duplicate-title` ; optional description). Fails first.

- **T010** `src/domain/usecases/create-task.ts` : implement until T009 passes.
- **T011** `test/domain/list-tasks.test.ts` : tests for `ListTasks` (empty list; creation order). **Fails first.**
- **T012** `src/domain/usecases/list-tasks.ts` : implement until T011 passes.

Phase 3 - Data

- **T013** `src/data/local-storage-task-repository.ts` : implements `TaskRepository` over `localStorage` (JSON serialization, ordering by `createdAt`, I/O exceptions contained at the edge).
- **T013b** `test/data/local-storage-task-repository.test.ts` : persistence test between sessions with a fake `Storage` (a new instance over the same storage simulates reopening the app, SC-005).

Phase 4 - Presentation

- **T014** `test/presentation/task-store.test.ts` : Zustand store tests with `InMemoryTaskRepository` (load list; successful create updates list; failure exposes message without changing the list). **Fails first.**
- **T015** `src/presentation/store/task-store.ts` : Zustand store that orchestrates `CreateTask` / `ListTasks` and exposes state (`tasks`, `lastError`). Accompanied by `src/presentation/store/task-store-context.tsx` (provider + selector hook).
- **T016 [P]** `src/presentation/components/TaskForm.tsx` : creation form (title + description) that triggers the store's action.
- **T017 [P]** `src/presentation/components/TaskList.tsx` : task list (title + description), empty state with no error.
- **T018** `src/presentation/App.tsx` : composes `TaskForm` + `TaskList` and loads the list on mount.
- **T019** `src/main.tsx` : composition at the edge, instantiates

`LocalStorageTaskRepository` , injects it into the use cases, creates the store, and mounts the `App` .

Phase 5 - Validation

- **T020** `npm test` (13 green tests) and `npm run build` (clean type-check + production build).
 - **T021** Validation of the Success Criteria: SC-001..SC-004 covered by the domain/store tests; SC-005 (persistence on reopening) covered by the `LocalStorageTaskRepository` test (T013b).
-

The code `implement` generated

From here on it is all code produced by the agent, in the state it was left at the end of this lap. The order follows that of the layers, from the inside out: first the domain (which knows no one), then the data, then the presentation, and finally the tests. It is the same direction of dependencies that `data-model.md` drew.

Domain

The heart of the system, pure TypeScript, with no import of React, Zustand, or `localStorage` . Everything that decides anything lives here.

The `Result<S, F>` is the spine of "error as value": every operation that can fail in a predictable way returns one of these two shapes, and the `kind` forces exhaustive handling.

```
// src/domain/result.ts
```

```
/**
```

- * Explicit result of an operation that can fail in a predictable way
- .
- *
- * Error as value (constitution, Principle III): instead of throwing an exception, the operation returns `success` with the value or `failure` with the typed failure. The discriminated union by `kind` forces exhaustive handling in TypeScript
- .
- *
- /

```
export type Result<S, F> =
```

```
  | { readonly kind: "success"; readonly value: S }
```

```
  | { readonly kind: "failure"; readonly error: F };
```

```
export const success = <S, F>(value: S): Result<S, F> => ({ kind: "success", value });
```

```
export const failure = <S, F>(error: F): Result<S, F> => ({ kind: "failure",  
error });
```

The `Task` entity: four fields, all `readonly`. The immutability is deliberate: changing a task is producing another, never altering the existing one in place.

```
// src/domain/entities/task.ts

/**

 * A to-do recorded by the person
 *
 *
 * The title is required and non-repeated among existing tasks; the description
 *
 * is optional. `createdAt` (ISO 8601) defines the display order (oldest first).
 *
 */

export type Task = {

  readonly id: string;

  readonly title: string;
```

```
    readonly description: string;

    readonly createdAt: string;

};
```

The predictable creation failures, one for each business rule. Notice that there is a type just for this: the error is as first-class as the success.

```
// src/domain/failures/create-task-failure.ts

/**
 * Predictable failures when creating a task, one for each business rule
 *
 *
 * `title-required`: empty title or only spaces (FR-002)
 *
 * `duplicate-title`: a task with the same title already exists, after `trim` (FR-003).
 *
 */

export type CreateTaskFailure =

    | { readonly kind: "title-required" }
```

```
| { readonly kind: "duplicate-title" };
```

The persistence repository, declared by the domain. The domain says what it needs (`getAll` , `add`), and leaves it to the data layer to decide how.

```
// src/domain/repositories/task-repository.ts

import type { Task } from "../entities/task";

/**

 * Domain repository to persist and read tasks
 *
 *
 * The domain declares this interface; the data layer implements it. That way
 *
 * the business rule doesn't know the storage medium
 *
 */

export interface TaskRepository {

  /** All the tasks, in order of creation (oldest first). */
```



```
getAll(): Promise<Task[]>;

/** Persists a new task. */

add(task: Task): Promise<void>;

}
```

The creation use case: the two business rules, in the order of the contract, returning failure as value. `generateId` and `now` are injected as parameters with a default, a small detail that makes the use case deterministic in the tests.

```
// src/domain/usecases/create-task.ts

import type { Task } from "../../entities/task";

import type { CreateTaskFailure } from "../../failures/create-task-failure";

import type { TaskRepository } from "../../repositories/task-repository";

import { failure, success, type Result } from "../../result";

export type CreateTaskInput = { title: string; description?: string };
```

```
export type CreateTask = (input: CreateTaskInput) => Promise<Result<Task, CreateTaskFailure>>;
```

```
/**
```

```
 * Creates a task applying the two business rules and returning error as value:
```

```
 * required title (FR-002) and non-repeated title (FR-003)
 *
```

```
 *
```

```
 * `generateId` and `now` are injected to keep the use case pure and testable
 *
```

```
 *
```

```
 /
```

```
export const makeCreateTask = (
```

```
  repository: TaskRepository,
```

```
  generateId: () => string = () => crypto.randomUUID(),
```

```
  now: () => Date = () => new Date()
```

```
): CreateTask =>
```

```
  async ({ title, description = "" }) => {
```

```
const trimmedTitle = title.trim();

if (trimmedTitle.length === 0) {

    return failure({ kind: "title-required" });

}

const existing = await repository.getAll();

const isDuplicate = existing.some((task) => task.title.trim() === trimmed
Title);

if (isDuplicate) {

    return failure({ kind: "duplicate-title" });

}

const task: Task = {

    id: generateId(),

    title: trimmedTitle,
```

```
        description: description.trim(),

        createdAt: now().toISOString()

    };

    await repository.add(task);

    return success(task);

};
```

The listing use case is deliberately tiny: it just delegates to the repository, whose order already comes guaranteed. Anything beyond that would be YAGNI.

```
// src/domain/usecases/list-tasks.ts

import type { Task } from "../../entities/task";

import type { TaskRepository } from "../../repositories/task-repository";

export type ListTasks = () => Promise<Task[]>;

/**
```

```

    * Returns all the existing tasks, in order of creation (FR-005). An empty li
    st

    * is a valid state, not an error (SC-004)
    .

    *
    /

export const makeListTasks = (repository: TaskRepository): ListTasks => () =>
  repository.getAll();

```

Data

The only concrete implementation of the repository in this lap. It is here, and only here, that `localStorage` appears, with the I/O exceptions contained at the edge.

```

// src/data/local-storage-task-repository.ts

import type { Task } from "../../domain/entities/task";

import type { TaskRepository } from "../../domain/repositories/task-repository";

const STORAGE_KEY = "todo-app.tasks";

/**

```

* Persists the tasks as JSON in the browser's `localStorage`. It is the web

* equivalent of "persist on the device" (FR-006): it survives closing an

* reopening the app. The I/O exceptions stay contained at this edge

* (constitution, Principle III); the domain never sees `localStorage`
.

*
/

```
export class LocalStorageTaskRepository implements TaskRepository {
```

```
    constructor(private readonly storage: Storage = localStorage) {}
```

```
    async getAll(): Promise<Task[]> {
```

```
        const raw = this.storage.getItem(STORAGE_KEY);
```

```
        if (raw === null || raw.trim().length === 0) {
```

```
            return [];
```

```
        }
```

```

const tasks = JSON.parse(raw) as Task[];

return [...tasks].sort((a, b) => a.createdAt.localeCompare(b.createdAt));
}

async add(task: Task): Promise<void> {

  const tasks = await this.getAll();

  const updated = [...tasks, task];

  this.storage.setItem(STORAGE_KEY, JSON.stringify(updated));

}

}

```

Presentation

The outermost layer. The Zustand store orchestrates the use cases and translates the typed failure into a message; the React components only collect input and draw state.

The store contains no rule: it triggers `createTask` / `listTasks` and, at most, converts a `CreateTaskFailure` into text for the screen. Note the exhaustive `switch` in `messageFor`, it is TypeScript demanding the

handling of each failure.

```
// src/presentation/store/task-store.ts

import { createStore } from "zustand/vanilla";

import type { Task } from "../../domain/entities/task";

import type { CreateTaskFailure } from "../../domain/failures/create-task-failure";

import type { CreateTask } from "../../domain/usecases/create-task";

import type { ListTasks } from "../../domain/usecases/list-tasks";

export type TaskStoreState = {

  readonly tasks: Task[];

  readonly lastError: string | null;

  loadTasks: () => Promise<void>;

  createTask: (title: string, description: string) => Promise<boolean>;

};
```



```

const messageFor = (failure: CreateTaskFailure): string => {

  switch (failure.kind) {

    case "title-required":

      return "The title is required.";

    case "duplicate-title":

      return "A task with this title already exists.";

  }

};

/**

 * Presentation store (Zustand) that orchestrates the domain use cases and
 * exposes the state to the UI. Contains no business rule: it just triggers
 * `createTask`/`listTasks` and translates the typed failure into a message for
 * the screen
 */

```

```
*  
/
```

```
export const createStore = (createTask: CreateTask, listTasks: ListTasks)  
=>
```

```
  createStore<TaskStoreState>((set) => ({
```

```
    tasks: [],
```

```
    lastError: null,
```

```
    loadTasks: async () => {
```

```
      const tasks = await listTasks();
```

```
      set({ tasks, lastError: null });
```

```
    },
```

```
    createTask: async (title, description) => {
```

```
      const result = await createTask({ title, description });
```

```
      if (result.kind === "failure") {
```

```
        set({ lastError: messageFor(result.error) });
```

```
        return false;

    }

    const tasks = await listTasks();

    set({ tasks, lastError: null });

    return true;

}

}));
```

The context that delivers the composed store to the React tree, with a selector hook that re-renders only when the observed slice changes.

```
// src/presentation/store/task-store-context.tsx

import { createContext, useContext, type ReactNode } from "react";

import { useStore, type StoreApi } from "zustand";

import type { TaskStoreState } from "../task-store";
```

```
const TaskStoreContext = createContext<StoreApi<TaskStoreState> | null>(null)
;
```

```
/** Makes the already-composed store (with the use cases injected) available
to the tree. */
```

```
export const TaskStoreProvider = ({
```

```
  store,
```

```
  children
```

```
}: {
```

```
  store: StoreApi<TaskStoreState>;
```

```
  children: ReactNode;
```

```
}) => <TaskStoreContext.Provider value={store}>{children}</TaskStoreContext.P
rovider>;
```

```
/** Reads a slice of the store's state, re-rendering only when it changes. */
```

```
export function useTaskStore<T>(selector: (state: TaskStoreState) => T): T {
```

```
  const store = useContext(TaskStoreContext);
```

```

    if (store === null) {

        throw new Error("useTaskStore must be used within a TaskStoreProvider.");

    }

    return useStore(store, selector);

}

```

The creation form: it collects title and description, triggers the action and, on success, clears the fields. The error message comes straight from the store.

```

// src/presentation/components/TaskForm.tsx

import { useState, type FormEvent } from "react";

import { useTaskStore } from "../store/task-store-context";

/** Creation form: collects title and description and triggers the store's action. */

export function TaskForm() {

    const createTask = useTaskStore((state) => state.createTask);

```

```
const lastError = useTaskStore((state) => state.lastError);
```

```
const [title, setTitle] = useState("");
```

```
const [description, setDescription] = useState("");
```

```
const handleSubmit = async (event: FormEvent) => {
```

```
    event.preventDefault();
```

```
    const created = await createTask(title, description);
```

```
    if (created) {
```

```
        setTitle("");
```

```
        setDescription("");
```

```
    }
```

```
};
```

```
return (
```

```
<form className="task-form" onSubmit={handleSubmit}>

  <input

    aria-label="Title"

    placeholder="Task title"

    value={title}

    onChange={(event) => setTitle(event.target.value)}

  />

  <textarea

    aria-label="Description"

    placeholder="Description (optional)"

    value={description}

    onChange={(event) => setDescription(event.target.value)}

  />

  {lastError !== null && <p role="alert" className="task-form__error">{lastError}</p>}

  <button type="submit">Add task</button>
```

```
    </form>

  );

}
```

The list, which only draws the existing tasks. The empty list has its own branch, it is a valid state, with a sentence inviting you to create the first one.

```
// src/presentation/components/TaskList.tsx

import { useTaskStore } from "../../store/task-store-context";

/** Lists the existing tasks in order of creation; an empty list is a valid state. */

export function TaskList() {

  const tasks = useTaskStore((state) => state.tasks);

  if (tasks.length === 0) {

    return <p className="task-list__empty">No tasks yet. Create the first one above.</p>;

  }

}
```



```

return (

  <ul className="task-list">

    {tasks.map((task) => (

      <li key={task.id} className="task-list__item">

        <h3>{task.title}</h3>

        {task.description.length > 0 && <p>{task.description}</p>}

      </li>

    ))}

  </ul>

);

}

```

The `App`, which composes the form and the list and loads the tasks on mount.

```

// src/presentation/App.tsx

import { useEffect } from "react";

```

```
import { TaskForm } from "../components/TaskForm";

import { TaskList } from "../components/TaskList";

import { useTaskStore } from "../store/task-store-context";

export function App() {

  const loadTasks = useTaskStore((state) => state.loadTasks);

  useEffect(() => {

    void loadTasks();

  }, [loadTasks]);

  return (

    <main className="app">

      <h1>To-Do</h1>

      <TaskForm />

      <TaskList />

    </main>
  );
}
```

```
    </main>

  );

}
```

The `main.tsx` is the edge of the composition, and it is the only place where the three layers meet: it instantiates the concrete repository, injects it into the use cases, creates the store, and mounts the `App`. Everything above was written without knowing this file exists.

```
// src/main.tsx

import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import { LocalStorageTaskRepository } from "../data/local-storage-task-repository";

import { makeCreateTask } from "../domain/usecases/create-task";

import { makeListTasks } from "../domain/usecases/list-tasks";

import { App } from "../presentation/App";

import { createTaskStore } from "../presentation/store/task-store";

import { TaskStoreProvider } from "../presentation/store/task-store-context";
```

```
import "./index.css";

// Composition at the edge: the concrete repository is instantiated here and
// injected into the use cases; only at this point do the layers meet.

const repository = new LocalStorageTaskRepository();

const store = createTaskStore(makeCreateTask(repository), makeListTasks(repository));

const rootElement = document.getElementById("root");

if (rootElement === null) {

  throw new Error("Element #root not found in index.html.");

}

createRoot(rootElement).render(

  <StrictMode>

    <TaskStoreProvider store={store}>
```

```
<App />

</TaskStoreProvider>

</StrictMode>

);
```

Tests

The tests that came before the code (the order in `tasks.md` makes that explicit). They are what prove that each functional requirement actually happens, and it is their presence that lets the four following laps touch this base without fear.

Two helpers hold up all the tests. The `InMemoryTaskRepository` is an implementation of the repository that lives in memory, and `MemoryStorage` is a fake `Storage`, to test the production repository without a browser.

```
// test/helpers/in-memory-task-repository.ts

import type { Task } from "../../src/domain/entities/task";

import type { TaskRepository } from "../../src/domain/repositories/task-repository";

/** In-memory implementation of `TaskRepository` for the domain and store tests. */
```

```
export class InMemoryTaskRepository implements TaskRepository {

    private readonly tasks: Task[] = [];

    async getAll(): Promise<Task[]> {

        return [...this.tasks].sort((a, b) => a.createdAt.localeCompare(b.createdAt));

    }

    async add(task: Task): Promise<void> {

        this.tasks.push(task);

    }

}
```

```
// test/helpers/memory-storage.ts

/** In-memory `Storage` to test the repository without a browser. */

export class MemoryStorage implements Storage {

    private readonly map = new Map<string, string>();

}
```

```
get length(): number {
```

```
    return this.map.size;
```

```
}
```

```
clear(): void {
```

```
    this.map.clear();
```

```
}
```

```
getItem(key: string): string | null {
```

```
    return this.map.has(key) ? this.map.get(key)! : null;
```

```
}
```

```
key(index: number): string | null {
```

```
    return [...this.map.keys()][index] ?? null;
```

```
}
```

```

removeItem(key: string): void {

    this.map.delete(key);

}

setItem(key: string, value: string): void {

    this.map.set(key, value);

}

}

```

The `CreateTask` tests cover the two business rules and the optionality of the description. Notice how each failure case also verifies that nothing was created.

```

// test/domain/create-task.test.ts

import { describe, expect, it } from "vitest";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository"
;

```



```
describe("CreateTask", () => {

  it("creates a task with a valid and unique title", async () => {

    const repository = new InMemoryTaskRepository();

    const createTask = makeCreateTask(repository);

    const result = await createTask({ title: "Buy bread", description: "Corner bakery" });

    expect(result.kind).toBe("success");

    const all = await repository.getAll();

    expect(all).toHaveLength(1);

    expect(all[0].title).toBe("Buy bread");

    expect(all[0].description).toBe("Corner bakery");

  });

  it("refuses an empty title with title-required", async () => {
```

```
const repository = new InMemoryTaskRepository();

const createTask = makeCreateTask(repository);

const result = await createTask({ title: "" });

expect(result).toEqual({ kind: "failure", error: { kind: "title-required"
} });

expect(await repository.getAll()).toHaveLength(0);

});

it("treats a title with only spaces as empty", async () => {

const repository = new InMemoryTaskRepository();

const createTask = makeCreateTask(repository);

const result = await createTask({ title: "  " });

expect(result).toEqual({ kind: "failure", error: { kind: "title-required"
} });
```

```

});

it("refuses a duplicate title (ignoring spaces at the ends) with duplicate-
title", async () => {

  const repository = new InMemoryTaskRepository();

  const createTask = makeCreateTask(repository);

  await createTask({ title: "Study SDD" });

  const result = await createTask({ title: "  Study SDD  " });

  expect(result).toEqual({ kind: "failure", error: { kind: "duplicate-title
" } });

  expect(await repository.getAll()).toHaveLength(1);

});

it("accepts a missing description (optional)", async () => {

  const repository = new InMemoryTaskRepository();

  const createTask = makeCreateTask(repository);

```

```

    const result = await createTask({ title: "No description" });

    expect(result.kind).toBe("success");

    if (result.kind === "success") {

        expect(result.value.description).toBe("");

    }

});

});

```

The `ListTasks` tests cover the empty list and the creation order. The trick of injecting a controlled `id` and `now` makes the order deterministic.

```

// test/domain/list-tasks.test.ts

import { describe, expect, it } from "vitest";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeListTasks } from "../../src/domain/usecases/list-tasks";

```

```
import { InMemoryTaskRepository } from "../helpers/in-memory-task-repository"
;

describe("ListTasks", () => {

  it("returns an empty list when there are no tasks", async () => {

    const listTasks = makeListTasks(new InMemoryTaskRepository());

    expect(await listTasks()).toEqual([]);

  });

  it("returns the tasks in order of creation (oldest first)", async () => {

    const repository = new InMemoryTaskRepository();

    let tick = 0;

    const createTask = makeCreateTask(

      repository,

      () => `id-${tick}`,

      () => new Date(2026, 0, 1, 0, 0, tick++)

    );
```

```

    );

    await createTask({ title: "First" });

    await createTask({ title: "Second" });

    await createTask({ title: "Third" });

    const listTasks = makeListTasks(repository);

    const titles = (await listTasks()).map((task) => task.title);

    expect(titles).toEqual(["First", "Second", "Third"]);

  });

});

```

The production repository test is the one that proves persistence between sessions (SC-005): a new instance over the same `Storage` simulates reopening the app.

```

// test/data/local-storage-task-repository.test.ts

import { describe, expect, it } from "vitest";

```

```
import { LocalStorageTaskRepository } from "../../src/data/local-storage-task-repository";
```

```
import { makeCreateTask } from "../../src/domain/usecases/create-task";
```

```
import { MemoryStorage } from "../helpers/memory-storage";
```

```
describe("LocalStorageTaskRepository", () => {
```

```
  it("returns an empty list when nothing was saved", async () => {
```

```
    const repository = new LocalStorageTaskRepository(new MemoryStorage());
```

```
    expect(await repository.getAll()).toEqual([]);
```

```
  });
```

```
  it("preserves the tasks between sessions (closing and reopening the app)",  
    async () => {
```

```
    const storage = new MemoryStorage();
```

```
    let tick = 0;
```

```
    const createTask = makeCreateTask(
```

```
      new LocalStorageTaskRepository(storage),
```

```

    () => `id-${tick}`,

    () => new Date(2026, 0, 1, 0, 0, tick++)

  );

  await createTask({ title: "First" });

  await createTask({ title: "Second" });

  // A new instance over the same Storage simulates reopening the app.

  const reopened = new LocalStorageTaskRepository(storage);

  const titles = (await reopened.getAll()).map((task) => task.title);

  expect(titles).toEqual(["First", "Second"]);

});

});

```

Finally, the store tests, which verify the orchestration: load the list, create successfully, and expose the right message on each failure, without changing the list.

```
// test/presentation/task-store.test.ts
```



```
import { describe, expect, it } from "vitest";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeListTasks } from "../../src/domain/usecases/list-tasks";

import { createTaskStore } from "../../src/presentation/store/task-store";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository"
;

const buildStore = () => {

    const repository = new InMemoryTaskRepository();

    return createTaskStore(makeCreateTask(repository), makeListTasks(repository
));

};

describe("task store", () => {

    it("loads the list of existing tasks", async () => {

        const store = buildStore();

        await store.getState().createTask("Task A", "");
```

```
    await store.getState().loadTasks();

    expect(store.getState().tasks.map((task) => task.title)).toEqual(["Task A
"]);

  });

  it("a successful create updates the list and clears the error", async () =>
  {

    const store = buildStore();

    const created = await store.getState().createTask("New", "a description")
    ;

    expect(created).toBe(true);

    expect(store.getState().tasks).toHaveLength(1);

    expect(store.getState().lastError).toBeNull();

  });

  it("a required-title failure exposes a message without changing the list",
  async () => {
```

```
const store = buildStore();

const created = await store.getState().createTask("  ", "");

expect(created).toBe(false);

expect(store.getState().tasks).toHaveLength(0);

expect(store.getState().lastError).toBe("The title is required.");

});

it("a duplicate-title failure exposes a specific message", async () => {

  const store = buildStore();

  await store.getState().createTask("Duplicate", "");

  const created = await store.getState().createTask("Duplicate", "");

  expect(created).toBe(false);
```

```
    expect(store.getState().lastError).toBe("A task with this title already exists.");

    expect(store.getState().tasks).toHaveLength(1);

  });

});
```

The portrait of the lap

A sentence went in. Out came a specification with seven traceable requirements, a plan that chose the stack without contradicting any principle, five design documents, a list of twenty-two tasks in TDD order, and nineteen code and test files organized in three layers, with thirteen green tests at the end. None of it was improvised along the way: each file has a place the plan already foresaw, and each decision has a document that explains it.

Keep the shape of this skeleton, because it is exactly the one you will run into again in the next laps. In Chapter 7.5, the feature of completing and reopening a task doesn't invent a new architecture: it adds a use case, a failure type, and a few fields, fitting into the same places. It is that repetition that proves, in practice, that the method holds the architecture up. The three-layer design that showed up here is the convention this book adopts for the example, and it is justified in *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it to follow the next laps: what is enough to keep is that the plan decided this division once and that the four laps that followed fit inside it.

6b - Specification Anti-Patterns: Six Ways to Get It Wrong, Over and Over

Six Defects, All Taken From This App

The first lap is over. You watched a spec be born, be questioned, become a plan, become tasks and become code, and you saw the result recorded in the repository. Before the second lap, it is worth stopping at a place the cycle has no dedicated step to teach: **what is usually wrong with a spec that looks right.**

The six patterns in this chapter did not come from a catalog. All of them happened in the To-Do, across the five laps this book walks, and all of them were fixed. That has one consequence for how you read the chapter: some of the episodes belong to laps you have not followed yet, the fourth and the fifth. Each case here is told whole, without depending on the chapter where it appears in full, and when you get there you will recognize the defect before it is named.

The way to use this is as a checklist. Once you finish writing a spec, before sending it to `clarify`, run the six. It takes five minutes, and each of them, once, cost a good deal more than that.

A word about the examples. The To-Do's repository only holds the **right** text, because every defect was fixed. So every bad version you are about to read is a declared hypothesis, built by

subtraction from a real artifact, with the real text beside it and the address of where it lives. No bad example in this chapter is a spec anybody actually handed in.

1. A Rule Inherited by Reference

The symptom: the requirement tells you to apply "the same rule", "the same criterion", "as in such-and-such feature", instead of stating what the rule is.

It looks like economy and it is a trap. Whoever writes it has the whole rule in their head at the moment of writing, and the reference preserves perfectly the thing they are thinking. Whoever reads it gets a pointer, and resolves that pointer with whatever they themselves think the rule means.

Hypothetical version. Suppose the edit-task spec said only this about the title:

FR-003 (hypothetical): The edited title **MUST** remain non-repeated, on the same criterion as creation.

There is nothing grammatically wrong there. The creation rule exists, it is written down, and it is a document from the same project. What the sentence does not answer is whether the task being edited counts in the comparison.

The real text. In the repository, after the defect showed up, FR-003 reads:

The edited title **MUST** remain non-repeated among the **other** tasks. Keeping the task's own title while editing does **NOT** count as a duplicate (it is the case of fixing only the

description).

And the fifth clarification of the `005-editor-tarefa` spec, which you find in Chapter 10.5, records where the adjustment came from: "the earlier wording, 'on the same criterion as creation', was ambiguous on this point and made editing only the description fail as a duplicate."

The cost of letting it through. This one got through everything. `clarify` did not ask, `checklist` flagged no gap, `analyze` came out clean, the tests derived from the tasks went green. It was manual validation, with the app open, that caught it. An ambiguous requirement is undetectable by any tool that checks consistency between artifacts, because the spec was consistent with itself: it was consistent and incomplete.

The general fix: whenever you are about to write "the same criterion as X", copy the criterion. If it is too long to copy, that is a sign the rule deserves a place of its own, and then the reference points at that place instead of at another feature.

2. Initial State Left Undeclared

The symptom: the spec fixes every possible option and forgets to say which one holds when the person arrives.

This one is treacherous because the list of options looks complete. You enumerated everything, reviewed it, nothing missing. What is missing is instant zero.

Hypothetical version. Suppose the filter-by-state spec had only this:

FR-001 (hypothetical): The system **MUST** offer three views: all, open and completed. **FR-002** (hypothetical): When a view is chosen, the system **MUST** show only the matching tasks.

The three views are there, the behavior of each one is there. And nobody said what the person sees on opening the app for the first time, which leaves the agent with three equally defensible answers.

The real text. This is the only gap in the series that a step of the cycle caught on its own. `checklist` flagged it, and the `003-filtrar-tarefas` spec gained a completeness correction note and a new requirement, FR-005, fixing the default view as "open". The block is in Chapter 8.5, right after the clarifications: "the initial version of this spec fixed the three views but did not declare which one appears when the screen opens."

The cost of letting it through. Low here, because it was caught early. Had it gone through, the app would open on "all", which is the default an agent picks when it does not know, and the whole feature would lose its point: anyone who asks for a filter wants to arrive already filtered.

The general fix: for every list of options, write the default line. It goes for views, for sorting, for modes, for anything with more than one possible value. The question is always the same: **what does the person see before choosing anything?**

3. A Verb With Two Owners

The symptom: two specs use the same expression to describe different behaviors, and each one is right within itself.

This one does not show up in the spec you are writing. It shows up between yours and another one, written in another lap, which makes it the only one of the six that requires looking outside the document.

Hypothetical version. Suppose the delete spec said:

FR-002 (hypothetical): On confirming the deletion, the task **MUST** leave the list definitively.

Read alone, that sentence is flawless. Now read it alongside what the complete-task spec already said, three laps earlier: a completed task **stays in the list**, in the same position, shown as completed. "Leave the list" came to mean two things in the same project, and neither spec has any way of knowing that on its own.

The real text. `analyze` was what caught it, and classified it as a HIGH conflict, the only one at that severity in the whole book. The report is in Chapter 9.5, with item C1 pointing at `004-excluir-tarefa/spec.md` FR-002 against `002-concluir-tarefa/spec.md` FR-006, and the recommendation to fix **at the source**: clarify that "definitive" qualifies the act of deleting, as distinct from completing. The correction went into the spec and flowed down the whole chain.

The cost of letting it through. High and silent. Two specs that contradict each other produce code that is coherent with each of them separately, and the conflict only shows up when somebody uses both features in the same session. It is the kind of defect that reaches the end user.

The general fix: keep a project vocabulary. When you use a strong verb ("leave", "remove", "archive", "cancel"), check whether it has been used before with another meaning. `analyze` does that cross-check, and it is good that it does, but the cheap moment to find out is while you write.

4. A Repeated Edge With No Answer

The symptom: the spec describes the happy path and the obvious error, and does not say what happens when the operation is repeated, or when it lands on something that is no longer there.

Two questions, and they share an origin: the spec thought about the first use and not the second.

Hypothetical version. Suppose the complete-task spec stopped here:

FR-001 (hypothetical): The person **MUST** be able to mark an open task as completed. **FR-002** (hypothetical): The person **MUST** be able to reopen a completed task.

And what happens when you complete a task that is already completed? The sentence neither forbids nor permits. An agent can treat it as an error, can ignore it, can toggle the state (the worst of the three, because it turns completing into an on/off switch nobody asked for).

The real text. The two ends of that edge were closed in different laps, both times in `clarify`. In the second lap, the fourth clarification of `002-concluir-tarefa`, which you find in Chapter 7.5, settled on idempotence: "it is an idempotent no-op: the operation succeeds and the state stays the same, with no additional effect." In the fifth lap, the fourth clarification of `005-editar-tarefa`, in Chapter 10.5, answered the other half, editing something that no longer exists: "a predictable error (error as value), not an exception that breaks the app."

The cost of letting it through. Depends on which half stayed open. An unanswered repetition gives you odd but recoverable behavior. An unanswered operation on something that vanished

tends to become an unhandled exception, which is the app closing in the face of whoever is using it.

The general fix: two fixed questions for every operation that changes state. **What if it happens twice? What if the target no longer exists?** They fit in any spec and almost always reveal a line that was missing.

5. Scope That Grows Out of Convenience

The symptom: the spec absorbs the near relative of what was asked for, because "while we are in here anyway".

This is the friendliest of the six, and that makes it the hardest to refuse. Nobody proposes extra scope in bad faith; they propose it because the addition is cheap **right now**, and it is true that it is.

Hypothetical version. Suppose the filter spec had turned into this:

FR-00X (hypothetical): Besides filtering by state, the system **MUST** allow sorting the list by date or by title, and **MUST** remember the chosen view when the app is reopened.

Sorting is filtering's cousin, remembering the choice looks like care for the user, and both would cost little while the filter code is open. The problem is not the cost of building; it is that the feature stopped being reviewable as a single thing, and that persisting the view introduces new state, with a life cycle of its own, inside a spec that was about something else.

The real text. Both were refused in the clarifications of `003-filtrar-tarefas`, in Chapter 8.5. Sorting: "Not in this feature. The order stays the creation order, inherited from 001. Configurable sorting

is out of scope (YAGNI); if it is wanted, it becomes a feature of its own later." Persistence: "The filter is an ephemeral view state for the session; on reopening the app, it goes back to the default view." In the fifth lap the same move refused an edit history, with the same note that it becomes a feature of its own if it is ever wanted.

The cost of letting it through. A bloated spec does not fail right away, and that is what makes it expensive. It fails in the review, which nobody finishes; it fails in manual validation, which now has twice the items; and it fails months later, when somebody tries to work out why the filter persists and the sorting does not, and the answer is "because it was open that day".

The general fix: refuse it in the clarification and record the refusal with its reason. Writing "out of scope; if it is wanted, it becomes a feature of its own" costs one line and saves you the whole conversation again three months from now.

6. Fixing at the End of the Chain

The symptom: the defect gets patched where it surfaced, in the code or in the task, and the spec goes on saying the wrong thing.

This one is different from the five before it. The others are defects of wording; this one is a defect of reflex, and it is what ruins the whole method, because it cuts the link between the documentation and what the software does.

Hypothetical version. This one needs no example of text, it needs an example of a gesture. The suite went red because editing only the description failed as a duplicate. The patch fits in one line, and it is the right line:

```
const isDuplicate = existing.some(

  (other) => other.id !== id && other.title.trim() === trimmedTitle

);
```

Apply that and you are done: green test, correct behavior, feature shipped. And the spec goes on saying that uniqueness is "on the same criterion as creation", which is precisely the sentence that produced the defect. The next person to read the spec, or the next agent to work from it, will rebuild the same error.

The real text. In both episodes where this nearly happened, the correction climbed back to the source. In the fourth lap's vocabulary conflict, `analyze` itself recommended it in writing: "fix at the source (the delete spec) [...] Do not patch it in the tasks." In the fifth lap, the order was spec, then tasks, then code: FR-003 rewritten and Q5 added first, T002 gaining the new case next, and only then the line above. Chapter 10.5 shows that sequence file by file.

The cost of letting it through. It is a compounding cost. Every patch that does not climb back to the source widens the distance between what the spec says and what the system does, and that distance is exactly what SDD exists to keep at zero. A spec that lies is worse than no spec at all, because the next person trusts it.

The general fix: when you find a defect, before opening the code file, ask **which artifact this defect was born in**. Fix it there, let the fix flow down, and the code changes as a consequence.

The Checklist, in Six Questions

Once the spec is written, before sending it on:

1. Does any requirement tell you to apply "the same criterion" from somewhere else instead of stating the rule?
2. Does every list of options have a line saying which one holds on opening?
3. Does any strong verb in this document already mean something else in another spec in the project?
4. For every operation that changes state: what if it happens twice? What if the target no longer exists?
5. Is there anything in here that got in only because it was cheap to add right now?
6. And, when the defect turns up later: are you going to fix it where it surfaced or where it was born?

The first five you answer by reading. The sixth asks a reflex of you, and it is the only one that keeps applying after the document is finished.

The second lap starts now, and it is where the fourth question on this list gets answered live, inside `clarify`.

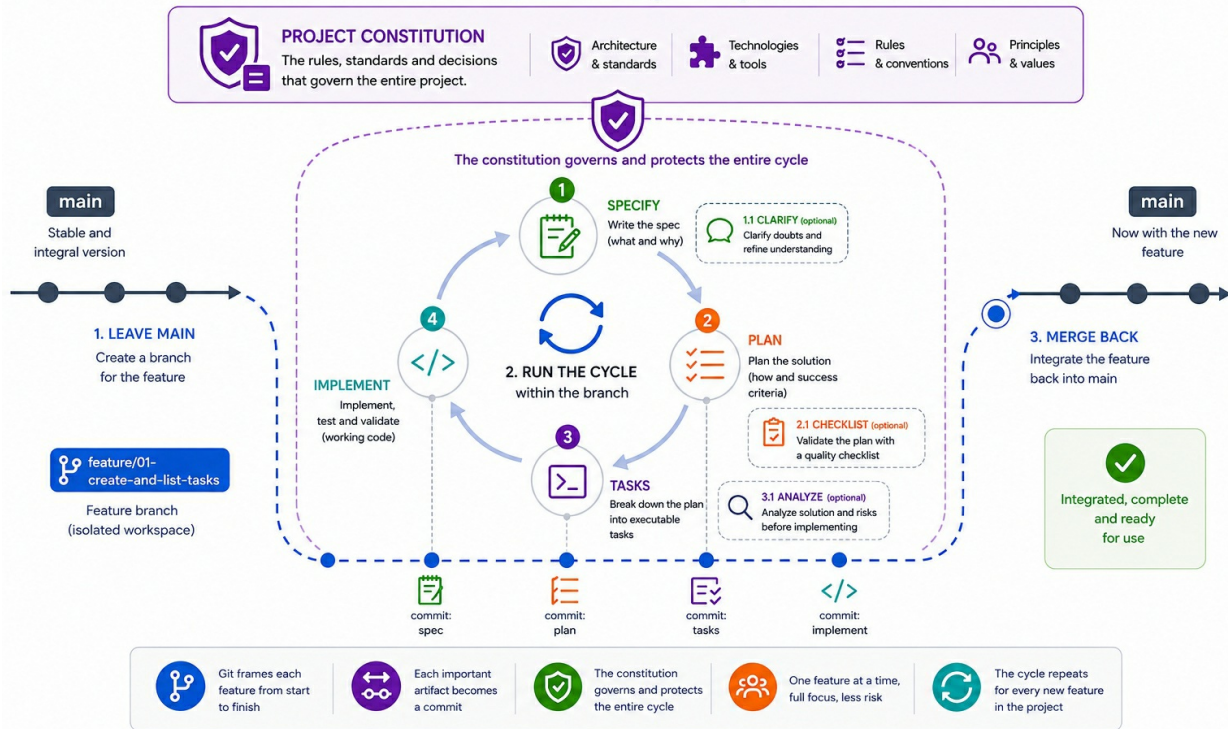
7 - Completing a Task: When the Obvious Hides Decisions

The state of git, again

In Chapter 6 you walked the whole cycle for the first time: you took "create and list tasks", carried it from `specify` to `merge`, and the first feature of the To-Do came to exist as running code. The `main` branch went stable again, with one more capability than it had before. Now you are about to take the second lap, and the good news is that the map is the same. It is worth reopening it for just a moment, not to relearn it, but to get your bearings:

THE COMPLETE SDD CYCLE

One feature, from nothing to integrated code



It is the same diagram from Chapter 5; what changes is only where you stand on it. The `main` branch is clean, with the create-task feature already integrated, and the first step of the cycle opens, on its own, the branch for the feature of the moment: `002-concluir-tarefa`. Notice that the branch belongs to the feature, numbered by the project, not by the chapter. What sets this chapter apart lies ahead, in a step that raced by in the first loop and now takes the focus.

The feature of the moment, and `specify` at a light pace

The To-Do backlog has the next slice waiting: **completing a task**, marking as done what you recorded in the previous loop. Along with it comes the action of going back, **reopening**. It is the

second feature in the queue, and it only makes sense because the first one exists: with no task created, there is nothing to complete.

You already learned how to specify in the previous loop; that is why `specify` runs light here. You describe the feature in a few lines, with what matters: the problem, the scope, and, with equal care, the non-goals.

```
/speckit-specify Complete and reopen a task in a personal to-do list. A person can mark an existing task as done and go back by reopening it. Out of scope for this feature: a history of completions, saved completion date and time, multi-level undo, filtering and editing tasks.
```

The non-goals here are the constitution's Simplicity and YAGNI acting before the first line of code even exists: you say, out loud, that you will not keep a history of completions, nor save date and time, nor build a multi-level undo. Filtering and editing are other features, from other chapters, and they stay where they are, in the backlog. It is cheap to write a non-goal and expensive to remove the code of a capability that never should have been born; the short sentence in the spec is what holds that account in check. And, as in the first loop, `specify` decides nothing about technology: the stack was already chosen and will be inherited when the time for `plan` arrives.

In a few minutes the spec is standing: short, with two user stories (complete and reopen), the rules and the non-goals. It seems nothing is left to clarify. And that is exactly where this chapter's trap lives.

"Completing" seems obvious. It is not.

Read the intent again: "mark a task as done". It sounds transparent. Anyone understands what it means. The natural reaction, faced with a sentence like that, is to say there is nothing to ask, that it is just a matter of implementing. Hold on to that feeling, because it is exactly the ground where decisions hide. The more obvious an intent seems, the more dangerous the ambiguities it carries, because no one thinks to discuss them.

This is where **clarify** comes in, and it is what the spotlight of this chapter falls on. Its job is simple to state and powerful in practice: it takes your spec, looks for the points that admit more than one reasonable interpretation (the **ambiguities**, the term you already met back in Chapter 2) and asks targeted questions, one at a time, before any plan is drawn. Run over the spec for "complete task", which seemed so clear, it raised questions no one in a hurry would have asked:

1. When a task is completed, does it **disappear** from the list or **stay** there, marked somehow?
2. Is completing **reversible**? Can you reopen what has been completed?
3. Does completion **change the order** of the list? Does the task go to the end, to a group of completed ones?
4. What happens when you try to complete a task that **is already completed**?

Each of these questions was hidden inside the verb "complete", and none of them has a single answer: the word seemed to carry the answer on its own, but it carried four decisions disguised as one.

Here is the point that holds up the entire chapter, and it is worth reading slowly: **a question you do not answer in **clarify** does not disappear. It only changes place.** If you ignore the question about the order of the list and send it off to be implemented,

someone will decide for you: the agent. When it comes time to write the code, faced with the ambiguity no one resolved, it will pick some plausible path, maybe the wrong one, and move on without warning. We have a name for this, the **agent's assumption**: the decision the agent makes on its own, in `implement`, when an ambiguity was left open. What the agent has in front of it at the moment it decides, and how that material gets there, belongs to *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering/>), the third volume in this trilogy, which answers what the agent sees right now, in the window of this one call, and at what cost. You do not need it here: this chapter's remedy comes earlier and costs less, which is keeping the ambiguity from ever reaching that point. `clarify` exists so that these decisions are yours, made early, with you in control, and not the agent's, late, in the dark.

Deep dive: what idempotency is in a command. The fourth question ("completing what is already completed") points to a property with a name of its own: **idempotency**. An action is idempotent when applying it once or many times to the same state gives the same result. Pressing the button of an elevator that is already called does not call it "even more"; completing a task that is already completed should not complete it "twice" nor throw an error. Recognizing that an operation needs to be idempotent is the kind of decision that goes unnoticed until it turns into a bug, and it is exactly what a question at the right moment brings to the surface.

The answers go back into the spec

Raising the questions is half the work. The other half, the one that turns `clarify` from an interesting conversation into a step of the flow, is what happens to the answers. You decide each one, and `clarify` records them back into the spec.

Deciding is up to you, and the criterion is the To-Do constitution, Simplicity and YAGNI up front. For our To-Do, the set of answers came out like this: the completed task **stays on the list**, shown struck through, and it stays **in the same position** as always, without reordering anything. Completing is **reversible** by an explicit reopen action. And completing what is already completed, or reopening what is already open, is an **idempotent no-op**. "No-op" is short for *no operation*: the action is accepted, it succeeds, and it simply does nothing because there is nothing to do. Completing a task that is already completed is exactly that, a success that changes nothing. No separate view of completed tasks, no history, no saved date. The minimum that answers the four questions well.

A caveat this chapter makes a point of nailing down: these are the **author's** answers for the To-Do, not the only right ones. Another project could send the completed task to the end of the list, or hide it in a tab. What `clarify` teaches, before any answer, is that **there is a decision to make**, and that it is better to make it now, with eyes open.

Now the detail that changes everything. The answers did not stay in the chat. `clarify` wrote them back into the spec itself, in a dated clarifications section, in the tool's real format:

```
## Clarifications
```

```
### Session 2026-06-27
```

```
- Q: When a task is completed, does it disappear from the list or stay visible? → A: It stays on the list, shown as completed (struck through/marked). The
```

re is no separate view of completed tasks.

- Q: Does completing a task change its position in the list? → A: No. The task stays in the same position (creation order is kept); completing does not reorder the list.

- Q: Is completing reversible? → A: Yes, by an explicit reopen action, which takes the task from completed back to open.

- Q: What happens when you complete an already completed task (or reopen an already open one)? → A: It is an idempotent no-op: the operation succeeds and the state stays the same, with no additional effect.

The artifact changed before your eyes. The spec that went into `clarify` ambiguous came out with the four decisions recorded, and each answer was also propagated to the right part of the document: the edge cases stopped being questions and became requirements, with idempotency and staying on the list written out in full. `clarify` did not talk about the spec. It **changed** the spec.

And that is why the next gesture is safe. Right after `clarify` comes `/clear`, that habit of clearing the agent's context you settled on back there. Clearing now throws nothing away, because the decision does not live in the memory of the conversation: it lives in the versioned spec, on disk, in git. The next step will start by reading the file, not the chat. You can forget the entire discussion without losing a comma of what was decided.

Deep dive: why `clarify` has a limit on questions. The tool does not ask everything it could: it works with a cap on questions per session, which forces it to prioritize the highest-impact ambiguities instead of sweeping every detail. Far from a limitation, the cap forces focus on what changes the plan and the code, and leaves the rest to your review. The exact number may change from one version of spec-kit to another, so check the official documentation when you need the precise value; what does not change is the idea of a limit that pushes prioritization.^{[1](#)}

What changes down the line

It may sound like too much effort for four little questions. The best way to measure that effort is to follow a single decision down the stream, from `clarify` to the code. Take the answer "completing is reversible".

Without that answer, the agent would have built a one-way feature: mark as done and that's it. With it, the whole stream changes. When `plan` read the clarified spec, it did not design one operation, it designed **two, inverse**: one to complete, one to reopen. The `tasks`, faithful to the constitution's TDD, broke that into pairs of test and implementation for each of the two actions, plus the tests for the idempotent case. In the code, the pair shows up literal, two symmetric operations the domain exposes:

```
complete-task.ts  open → completed  (completing an already completed one =  
no-op)  
reopen-task.ts    completed → open    (reopening an already open one      =  
no-op)
```

The other decision, "stays struck through in the same position", ran down the stream just the same. It shaped the task's state (one extra field, saying whether it is completed) and the way to display it (struck through, without moving), and it forced the repository to gain an operation to update the task **without reordering** the list. One sentence answered in `clarify`, and three artifacts down the line already knew what to do.

Now imagine the counterfactual. If the reversibility question had been left open, the agent would have assumed. Maybe it would have assumed that completing is final, and you would only discover the problem when you tried to reopen a task and realized you could not. The fix, at that point, would no longer be changing one answer in a spec: it would be redoing the plan, the tasks, the tests and the code. What cost one question here would cost, down the line, a whole round of rework.

The rest of the loop ran at a following pace, each step reading the artifact of the one before. Between `plan` and `tasks`, spec-kit runs the `checklist`, and it is worth a sentence so as not to confuse things: it is a quality gate for the **requirements**, a kind of "test for the spec", that checks whether the decisions from `clarify` came out clear and measurable before they turned into tasks: a test of what was specified, without touching code. The `analyze` checked the consistency between spec, plan and tasks, and the `implement` walked the list until the code existed, with the tests in the green. None of these steps stood under the spotlight here, because it is not their turn: each will have its own chapter further on, with the care it deserves. For now, they appear in passing, just enough for you to see the `clarify` decision propagating through all of them.

Deep dive: how "error as value" shapes the invalid case.

There is a case `clarify` did not need to raise because the constitution already answered it: what to do when you ask to complete a task that does not exist? By the error-as-value principle, this does not become an exception that blows up between the layers; it becomes an explicit failure result, which the caller is obliged to handle. Notice the fine distinction: the nonexistent task is a failure (an error result), but completing the already completed one is a success that

changes nothing (the idempotent no-op). Two similar cases, different treatments, both decided on purpose and not in a scramble.

How to distrust the obvious

You saw `clarify` at work. The question that remains is more valuable than the command: how do you generate good questions like those on your own, even before running the tool? Because what protects a project, more than the step itself, is the habit of distrusting the obvious that the step trains.

Looking at the four questions that mattered, you can distill a handful of angles that almost always hide decisions. Faced with any short intent, try running through them:

- **Behavior:** what exactly happens when the action succeeds? What does "complete" mean, on the screen and in the data?
- **Reversibility:** can it be undone? Is there an inverse action? Who can do it?
- **State:** which states can the thing take on, and which transitions between them are valid?
- **Order and position:** does the action change the order, the position, the grouping of something the person sees?
- **Repetition:** what happens if the action is repeated on the same target? Is it idempotent?
- **Edge cases:** what about when the target does not exist, is empty, or is already in the final state?

Do not memorize this as a list to tick off. The value is in acquiring the reflex of looking at an apparently simple sentence and asking "what is this not telling me?". The more natural it becomes to ask

those questions, the less you will depend on the tool to remember them for you, and the better the specs you write before `clarify` even runs. An intent like "archive a message" or "cancel an order" opens, under these angles, the same fan of hidden decisions that "complete a task" opened.

And here it is worth undoing a confusion of words. This set of angles is **not** the `checklist` step of spec-kit. The `checklist` is that requirements quality gate that showed up in the previous section, a real step of the loop. What we just assembled is only a habit of reasoning, a way of thinking, that you use in your head before and during `clarify`. One is a step of the tool; the other is the mental reflex that makes you distrust the obvious. Do not confuse the memorized list with the step of the flow: they are different things with similar names.

End of the loop and the bridge to Ch. 8

With the tests in the green, the loop closes the same way as in the previous lap. You commit the work, artifact by artifact, and **merge** the `002-concluir-tarefa` branch back into `main`. The feature that began as a sentence is now running code: it marks tasks as completed, keeps them struck through in place, lets you reopen them and does not get confused when the action repeats. The `main` branch is stable again, with two capabilities where there was one.

And the focus will move once more. The next feature in the backlog is **filtering tasks**: seeing only the open ones, only the completed ones, or all of them. That is a feature of a different nature, and the difference has a consequence: in Chapter 8, the spotlight shifts one more notch, to `plan`, the step where the how is decided.

Footnotes

Command names, the order of the steps, and the exact form of invocation may evolve between versions of spec-kit; the excerpts in this chapter were generated with version 0.11.8. For details that change per release (flags, subcommands, internal paths), check the official spec-kit documentation instead of fixing them from memory.

7.5 - Completing and Reopening a Task in Practice: the Whole Loop, File by File

How to read this chapter

Chapter 7 followed the second lap of the cycle with the spotlight on `clarify`. Here you see the whole lap, the way it was recorded in the example app's repository, in the `002-concluir-tarefa` feature. The prompt that went in, the specification that came out (with the four `clarify` answers already incorporated), the plan, the state model, the tasks and every code and test file the agent produced.

Like 6.5, this is a reference chapter. And here it has an extra purpose: to make it visible that the second feature **invents no new architecture**. It adds one field, two use cases and one repository operation, and it all fits into the same places 6.5 showed. Several files below are modifications of files you already saw on the previous lap; where that is the case, the text points out what changed.

The artifacts appear as markdown (headings lowered so as not to compete with the ones in the material); the code, in blocks, each preceded by a short sentence.

The input: the `/speckit-specify` prompt

"Complete and reopen a task in a personal to-do list. A person can mark an existing task as done and go back by reopening it. Out of scope for this feature: a history of completions, saved completion date and time, multi-level undo, filtering and editing tasks."

The sentence is short and seems obvious: mark as done. Chapter 7 showed that the obvious hid decisions.

What `specify` and `clarify` returned: `spec.md`

Notice how the four `clarify` questions (disappears or stays, reorders or not, reversible or not, what to do with the repeated operation) turned into specific functional requirements. Without them, the feature would have been implemented in four different ways, all defensible.

Feature Specification: Complete and reopen a task

Feature Branch: `002-concluir-tarefa`

Created: 2026-06-27

Status: Draft

Input: User description: "Complete and reopen a task in a personal to-do list. A person can mark an existing task as done and go back by reopening it. Out of scope for this feature: a history of completions, saved completion date and time, multi-level undo, filtering and editing tasks."

Clarifications

- Q: When a task is completed, does it disappear from the list or stay visible? → A: It stays on the list, shown as completed (struck through/marked). There is no separate view of completed tasks.
- Q: Does completing a task change its position in the list? → A: No. The task stays in the same position (creation order is kept); completing does not reorder the list.
- Q: Is completing reversible? → A: Yes, by an explicit reopen action, which takes the task from completed back to open.
- Q: What happens when you complete an already completed task (or reopen an already open one)? → A: It is an idempotent no-op: the operation succeeds and the state stays the same, with no additional effect.

User Scenarios & Testing (mandatory)

User Story 1 - Complete a task (Priority: P1)

The person using the To-Do finished a chore and wants to record that it is done. They mark the corresponding task as completed and the list starts to show that this item has been fulfilled, without it disappearing from the radar.

Why this priority: It is the core of the feature and the reason it exists: to give the person the sense of progress from closing what they finished. Delivered on its own, it already produces value.

Independent Test: It can be tested in isolation by completing an open task and checking that it moves to the completed state.

Acceptance Scenarios:

1. **Given** an open task on the list, **When** the person marks it as completed, **Then** the task moves to the completed state and

remains on the list.

User Story 2 - Reopen a completed task (Priority: P1)

The person completed a task by mistake, or the chore became necessary again. They reopen the task, which goes back to the open state, as if the completion had not happened.

Why this priority: Completing with no way to go back is a trap; reversibility makes the action safe to use. It forms a pair with completing.

Independent Test: It can be tested by completing a task and then reopening it, checking that it returns to the open state.

Acceptance Scenarios:

1. **Given** a completed task, **When** the person reopens it, **Then** the task goes back to the open state and remains on the list.

Edge Cases

- Completing a task that is already completed is an idempotent no-op: the operation succeeds and the state does not change (FR-005).
- Reopening a task that is already open is an idempotent no-op, symmetric to the previous one (FR-005).
- A completed task stays visible on the list, shown as completed, in the same position it was in (FR-006).

Requirements (mandatory)

Functional Requirements

- **FR-001:** The system **MUST** allow marking an existing task as completed.

- **FR-002:** The system **MUST** allow reopening a completed task, taking it back to the open state.
- **FR-003:** The system **MUST** represent the state of a task as one of two values: open or completed.
- **FR-004:** The system **MUST** preserve the state (open/completed) of tasks between usage sessions, together with the rest of the task's information.
- **FR-005:** The system **MUST** treat completing an already completed task (and reopening an already open one) as an idempotent operation: successful and without changing the state already in effect.
- **FR-006:** The system **MUST** keep the completed task visible on the list, in the same position (creation order preserved); completing **MUST NOT** reorder or hide the task.

Non-Goals

- Explicitly out of scope: **history of completions, saved completion date/time, multi-level undo, filtering and editing tasks (YAGNI).**

Key Entities

- **Task:** besides the title and description that already exist, it now has a **state**: open or completed. A task is born open; **completing** takes it to completed and **reopening** takes it back to open (inverse actions). A completed task stays on the list, in the same position. Completing what is already completed (or reopening what is already open) is idempotent.

Success Criteria (mandatory)

Measurable Outcomes

- **SC-001:** The person can mark an open task as completed and see that state reflected on the list.
- **SC-002:** The person can reopen a completed task and see it go back to the open state.
- **SC-003:** The state of each task stays correct after closing and reopening the app.

Assumptions

- The feature extends the create-and-list-tasks one (001-criar-tarefa), already integrated into `main`; it reuses the Task entity, the repository and the existing layered architecture.
 - The To-Do constitution governs the feature: Simplicity/YAGNI, layered architecture, logic isolated from the UI, error as value and TDD.
 - The storage medium stays the one decided in the previous feature's Plan; here we only add the new state to what is already persisted.
-

What `plan` decided: `plan.md`

The high point of this plan is explicit traceability: a whole table links each `clarify` decision to a concrete effect in the design. That is how the conversation of the previous step survives: by turning into structure. Two of those decisions, error as value and the boundary between layers, get treatment of their own in *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it to read the plan below: it carries each decision with the reasoning beside it, which is all this chapter asks of you.

Implementation Plan: Complete and reopen a task

Branch: `002-concluir-tarefa` | **Spec:** `spec.md`

Input: Feature specification in `specs/002-concluir-tarefa/spec.md`

Summary

Add to the task a **state** (open/completed) and two inverse actions, **complete** and **reopen**, keeping the completed task visible on the list, in the same position. The decisions fixed in `clarify` shape the design: since completing is **reversible**, the plan foresees a **reopen** use case symmetric to complete (one action alone would not be enough); since the state needs to **persist** and the task **stays in the same position**, the repository gains an **update** operation that preserves the order; and since completing what is already completed is **idempotent**, the use cases handle the no-op without error. The rule stays isolated in the domain layer, with the UI only reflecting the state and triggering the actions.

Technical Context

Language/Version: TypeScript 5.x on Node 20+; target: browser (React web). Inherited from 001.

Main dependencies: React 18, Vite, Zustand. No new dependencies.

Persistence: `localStorage` behind `TaskRepository`, as in 001. The new state field goes into the same persisted JSON; old tasks without the field are read as open (backward compatibility).

Tests: Vitest for the use cases (no DOM) and for the store.

Project type: single-page, single-user web application, no backend.

Constitution Check

Principle	How the plan meets it
I. Layered Architecture	New use cases <code>complete-task</code> / <code>reopen-task</code> in the domain; <code>data</code> gains <code>update</code> ; <code>presentation</code> only reflects.
II. Isolated Business Logic	The state transition and idempotency live in the pure-TS use cases; React/Zustand decide nothing.
III. Error as Value	The predictable failure (nonexistent task) is <code>Result</code> / <code>CompleteTaskFailure</code> ; the idempotent no-op is success , not error.
IV. TDD	Tests of the use cases (complete, reopen, idempotency, nonexistent) before the implementation (see tasks.md).
V. Simplicity/YAGNI	No history, no saved date/time, no multi-level undo, no filter or editing; a single state field.
VI. Technology Agnosticism	The spec and the clarify do not name a stack; the decision (a boolean field persisted in the same medium) belongs to this plan.

Result: PASS, no violation.

State modeling decision

The state is represented by a boolean field `completed` on the `Task` entity (open = `false`, completed = `true`). Two states, with no elaborate state machine (YAGNI). The alternative of an enum `"open" | "done"` was considered and dropped as overkill for two values; a named boolean is enough and keeps serialization trivial.

Traceability clarify → plan

clarify decision	Effect on the plan
Completing is reversible	Two inverse use cases: <code>complete-task</code> and <code>reopen-task</code> .
Stays in the same position	<code>getAll</code> keeps ordering by <code>createdAt</code> ; <code>update</code> swaps the item in place, without reordering.
Stays visible, struck through	<code>TaskList</code> renders the completed state (without hiding it); the action toggles complete/reopen.
Idempotency	Completing an already completed one and reopening an already open one return success without changing the state.

Project Structure (additions)

```
src/
├── domain/
│   ├── entities/task.ts           # + completed field
│   ├── failures/complete-task-failure.ts # new: task-not-found
│   ├── repositories/task-repository.ts  # + update(task)
│   └── usecases/
│       ├── complete-task.ts          # new
│       └── reopen-task.ts            # new
└── data/
```

```
├── local-storage-task-repository.ts # + update; backward-compatible read
├── presentation/
│   ├── store/task-store.ts          # + completeTask/reopenTask
│   └── components/TaskList.tsx      # + action and completed styling
├── test/
│   └── domain/
│       ├── complete-task.test.ts    # new
│       └── reopen-task.test.ts      # new
```

Phase 1 - Design

- State model in data-model.md.
 - Manual validation in quickstart.md.
-

The `plan`'s state model: `data-model.md`

On this lap, the design artifact that matters is the state model. It draws the transitions and, above all, fixes that the idempotent no-op is **success**, not error, a distinction Chapter 7 highlighted and that here becomes a contract.

State model: Complete and reopen a task

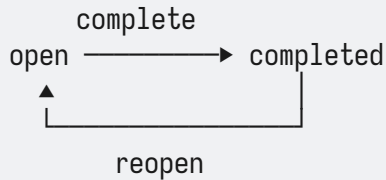
Task entity (addition)

The `Task` gains a `completed: boolean` field.

- `false` → **open** task (the initial state of every created task).
- `true` → **completed** task.

Tasks persisted by the previous feature do not have the field; when read, the absent one is interpreted as `false` (open). Backward compatibility with no migration.

Transitions



- **complete:** open → completed. On an already completed one, no-op (idempotent).
- **reopen:** completed → open. On an already open one, no-op (idempotent).
- The position on the list does not change in any transition (order by `createdAt` preserved).

Predictable failure

- `task-not-found` : the target task (by id) does not exist. Represented as a `Result` /value error, not an exception. The idempotent no-op is NOT a failure: it is success with no state change.

The `tasks` execution list: `tasks.md`

The task list is shorter than 001's, because the setup already exists. Notice that it starts with the domain (field, failure, contract), crosses data and ends in presentation, always with the

test before the implementation.

Tasks: Complete and reopen a task

Branch: `002-concluir-tarefa` | Plan: plan.md

TDD order: tests before the corresponding implementation. [P]
marks tasks that can run in parallel (distinct files, no dependency).

Phase 1 - Domain (core, pure TS)

- **T001** `src/domain/entities/task.ts` : add `completed: boolean` field.
- **T002** `src/domain/failures/complete-task-failure.ts` : `CompleteTaskFailure` union (`task-not-found`).
- **T003** `src/domain/repositories/task-repository.ts` : add `update(task: Task): Promise<void>` and `getById(id)` to the contract.
- **T004** `test/helpers/in-memory-task-repository.ts` : implement `update` / `getById` in the fake, preserving the order.
- **T005** `test/domain/complete-task.test.ts` : tests for `CompleteTask` (open→completed; **idempotent** on already completed; **nonexistent** → `task-not-found`). **Fails first**.
- **T006** `src/domain/usecases/complete-task.ts` : implement until T005 passes.
- **T007** `test/domain/reopen-task.test.ts` : tests for `ReopenTask` (completed→open; **idempotent** on already open; **nonexistent** → `task-not-found`). **Fails first**.
- **T008** `src/domain/usecases/reopen-task.ts` : implement until T007 passes.

Phase 2 - Data

- **T009** `src/data/local-storage-task-repository.ts` : implement `update` (replaces in place, without reordering) and `getById` ; backward-compatible read (`completed` absent → `false`).

Phase 3 - Presentation

- **T010** `test/presentation/task-store.test.ts` : store tests (complete updates the state; reopen reverts; idempotency does not break). **Fails first.**
- **T011** `src/presentation/store/task-store.ts` : `completeTask` / `reopenTask` actions that orchestrate the use cases and reload the list.
- **T012** `src/presentation/components/TaskList.tsx` : show the completed task as struck through, in the same position, with the action toggling complete/reopen.
- **T013** `src/main.tsx` : compose the new use cases at the edge.

Phase 4 - Validation

- **T014** `npm test` (all green) and `npm run build` (clean type-check).
 - **T015** Validation of Success Criteria SC-001..SC-003.
-

The code `implement` generated

From here on it is all code, in the state it was in at the end of this lap. Where a file already existed on the previous lap, the text points out what changed.

Domain

The `Task` entity gains a single new field, `completed` , with the rest identical to 6.5: a boolean is enough for two states.

```
// src/domain/entities/task.ts
```

```
/**
```

```
 * A chore recorded by the person
```

```
 *
```

```
 *
```

```
 * The title is required and not repeated among the existing tasks; the
```

```
 * description is optional. `createdAt` (ISO 8601) defines the display order
```

```
 * (oldest first). `completed` holds the state: `false` is open (the initial
```

```
 * state of every created task), `true` is completed
```

```
 *
```

```
 *
```

```
 /
```

```
export type Task = {
```

```
  readonly id: string;
```

```
  readonly title: string;
```

```
  readonly description: string;
```

```
readonly createdAt: string;

readonly completed: boolean;

};
```

The predictable failure of the two new actions. It is shared by complete and reopen, and the comment makes explicit what is **not** a failure: the idempotent operation.

```
// src/domain/failures/complete-task-failure.ts

/**

 * Predictable failure when completing or reopening a task
 *
 *
 *
 * `task-not-found`: no task exists with the given id. The idempotent no-op
 *
 * (completing the already completed one, reopening the already open one) is
 * NOT
 *
 * a failure: it is success with no state change
 *
 */
```

```
export type CompleteTaskFailure = { readonly kind: "task-not-found" };
```

The repository grows: besides `getAll` and `add`, it now declares `getById` (to find the target task) and `update` (to swap it in place, without reordering). The contract changes; whoever implements it, in the data layer, has to keep up.

```
// src/domain/repositories/task-repository.ts

import type { Task } from "../entities/task";

/**
 * Domain repository for persisting and reading tasks
 *
 *
 * The domain declares this interface; the data layer implements it. That way the
 * business rule does not know the storage medium
 *
 */

export interface TaskRepository {

  /** All tasks, in creation order (oldest first). */
```

```

getAll(): Promise<Task[]>;

/** A task by id, or `null` if it does not exist. */

getId(id: string): Promise<Task | null>;

/** Persists a new task. */

add(task: Task): Promise<void>;

/** Updates an existing task in place, without changing the order of the list. */

update(task: Task): Promise<void>;

}

```

The complete use case, the direct translation of the state model: a nonexistent task is a failure; an already completed one is success with no effect (idempotency); an open one becomes completed through an immutable copy.

```

// src/domain/usecases/complete-task.ts

import type { Task } from "../../entities/task";

```

```
import type { CompleteTaskFailure } from "../failures/complete-task-failure";
```

```
import type { TaskRepository } from "../repositories/task-repository";
```

```
import { failure, success, type Result } from "../result";
```

```
export type CompleteTask = (id: string) => Promise<Result<Task, CompleteTaskFailure>>;
```

```
/**
```

```
 * Marks a task as completed (FR-001)
```

```
 *
```

```
 *
```

```
 * Completing an already completed task is an idempotent no-op: it returns success
```

```
 * without touching the state (FR-005). The nonexistent task is error as value, not
```

```
 * an exception (FR-003 of the constitution). The position on the list does not
```

```
 * change: the repository updates in place
```

```
 *
```

```
 *
```

```
 /
```

```
export const makeCompleteTask = (repository: TaskRepository): CompleteTask =>

  async (id) => {

    const task = await repository.getById(id);

    if (task === null) {

      return failure({ kind: "task-not-found" });

    }

    if (task.completed) {

      return success(task);

    }

    const completed: Task = { ...task, completed: true };

    await repository.update(completed);

    return success(completed);
```

```
};
```

The reopen use case is the exact mirror of the previous one. The symmetry is no coincidence: the plan foresaw two inverse actions, and the code writes them almost identical, swapping only the target value of `completed`.

```
// src/domain/usecases/reopen-task.ts

import type { Task } from "../entities/task";

import type { CompleteTaskFailure } from "../failures/complete-task-failure";

import type { TaskRepository } from "../repositories/task-repository";

import { failure, success, type Result } from "../result";

export type ReopenTask = (id: string) => Promise<Result<Task, CompleteTaskFailure>>;

/**

 * Reopens a completed task, taking it back to open (FR-002). It is the inverse of

 * `CompleteTask`

 * .
```

*

* Reopening an already open task is an idempotent no-op (FR-005); the nonexistent

* task is error as value (`task-not-found`). The position on the list does not change.

*

/

```
export const makeReopenTask = (repository: TaskRepository): ReopenTask =>
```

```
  async (id) => {
```

```
    const task = await repository.getById(id);
```

```
    if (task === null) {
```

```
      return failure({ kind: "task-not-found" });
```

```
    }
```

```
    if (!task.completed) {
```

```
      return success(task);
```

```
    }
```



```

const reopened: Task = { ...task, completed: false };

await repository.update(reopened);

return success(reopened);

};

```

`CreateTask` changes in a single line: the new task is born with `completed: false`. Everything else is what 6.5 showed.

```

// src/domain/usecases/create-task.ts

import type { Task } from "../entities/task";

import type { CreateTaskFailure } from "../failures/create-task-failure";

import type { TaskRepository } from "../repositories/task-repository";

import { failure, success, type Result } from "../result";

export type CreateTaskInput = { title: string; description?: string };

export type CreateTask = (input: CreateTaskInput) => Promise<Result<Task, CreateTaskFailure>>;

```

```
/**  
  
 * Creates a task applying the two business rules and returning error as value:  
  
 * required title (FR-002) and non-repeated title (FR-003)  
 *  
  
 * `generateId` and `now` are injected to keep the use case pure and testable  
 *  
 */
```

```
export const makeCreateTask = (  
  
  repository: TaskRepository,  
  
  generateId: () => string = () => crypto.randomUUID(),  
  
  now: () => Date = () => new Date()  
  
) => CreateTask =>  
  
  async ({ title, description = "" }) => {  
  
    const trimmedTitle = title.trim();
```

```
if (trimmedTitle.length === 0) {

    return failure({ kind: "title-required" });

}

const existing = await repository.getAll();

const isDuplicate = existing.some((task) => task.title.trim() === trimmed
Title);

if (isDuplicate) {

    return failure({ kind: "duplicate-title" });

}

const task: Task = {

    id: generateId(),

    title: trimmedTitle,

    description: description.trim(),

    createdAt: now().toISOString(),
```

```

        completed: false

    };

    await repository.add(task);

    return success(task);

};

```

Data

The `localStorage` repository gains `getById` and `update`, and the read starts treating old tasks (without `completed`) as open, the backward compatibility the plan promised, solved with a `?? false` in the deserialization.

```

// src/data/local-storage-task-repository.ts

import type { Task } from "../../domain/entities/task";

import type { TaskRepository } from "../../domain/repositories/task-repository";

const STORAGE_KEY = "todo-app.tasks";

/**

```

```
* Persists the tasks as JSON in the browser's `localStorage`. It is the we  
b  
  
* equivalent of "persist on the device" (FR-006): it survives closing an  
d  
  
* reopening the app. The I/O exceptions stay contained at this boundar  
y  
  
* (constitution, Principle III); the domain never sees `localStorage`  
.  
  
*  
/
```

```
export class LocalStorageTaskRepository implements TaskRepository {  
  
  constructor(private readonly storage: Storage = localStorage) {}  
  
  async getAll(): Promise<Task[]> {  
  
    const raw = this.storage.getItem(STORAGE_KEY);  
  
    if (raw === null || raw.trim().length === 0) {  
  
      return [];  
  
    }  
  }  
}
```

```
const stored = JSON.parse(raw) as Array<Omit<Task, "completed"> & { completed?: boolean }>;
```

```
// Tasks written by the previous feature do not have `completed`; reading the
```

```
// absent one as `false` (open) keeps compatibility with no migration.
```

```
const tasks: Task[] = stored.map((task) => ({ ...task, completed: task.completed ?? false }));
```

```
return tasks.sort((a, b) => a.createdAt.localeCompare(b.createdAt));
```

```
}
```

```
async getById(id: string): Promise<Task | null> {
```

```
const tasks = await this.getAll();
```

```
return tasks.find((task) => task.id === id) ?? null;
```

```
}
```

```
async add(task: Task): Promise<void> {
```

```

    const tasks = await this.getAll();

    const updated = [...tasks, task];

    this.storage.setItem(STORAGE_KEY, JSON.stringify(updated));

}

async update(task: Task): Promise<void> {

    const tasks = await this.getAll();

    const updated = tasks.map((existing) => (existing.id === task.id ? task :
existing));

    this.storage.setItem(STORAGE_KEY, JSON.stringify(updated));

}

}

```

Presentation

The store gets two new actions (`completeTask` / `reopenTask`) that just call the use cases and reload the list. `messageFor`, the error-translation part, stays exactly as in 6.5.

```
// src/presentation/store/task-store.ts

import { createStore } from "zustand/vanilla";

import type { Task } from "../../domain/entities/task";

import type { CreateTaskFailure } from "../../domain/failures/create-task-failure";

import type { CompleteTask } from "../../domain/usecases/complete-task";

import type { CreateTask } from "../../domain/usecases/create-task";

import type { ListTasks } from "../../domain/usecases/list-tasks";

import type { ReopenTask } from "../../domain/usecases/reopen-task";

export type TaskStoreState = {

  readonly tasks: Task[];

  readonly lastError: string | null;

  loadTasks: () => Promise<void>;

  createTask: (title: string, description: string) => Promise<boolean>;

  completeTask: (id: string) => Promise<void>;
```



```
reopenTask: (id: string) => Promise<void>;

};

const messageFor = (failure: CreateTaskFailure): string => {

  switch (failure.kind) {

    case "title-required":

      return "The title is required.";

    case "duplicate-title":

      return "A task with this title already exists.";

  }

};

/**

 * Presentation store (Zustand) that orchestrates the domain use cases and ex
 poses

 * the state to the UI. It contains no business rule: it only trigger
 s
```

```
* `createTask`/`listTasks` and translates the typed failure into a message f  
or
```

```
* the screen  
.
```

```
*  
/
```

```
export const createTaskStore = (
```

```
  createTask: CreateTask,
```

```
  listTasks: ListTasks,
```

```
  completeTask: CompleteTask,
```

```
  reopenTask: ReopenTask
```

```
) =>
```

```
  createStore<TaskStoreState>((set) => ({
```

```
    tasks: [],
```

```
    lastError: null,
```

```
    loadTasks: async () => {
```

```
      const tasks = await listTasks();
```

```
    set({ tasks, lastError: null });

},

createTask: async (title, description) => {

    const result = await createTask({ title, description });

    if (result.kind === "failure") {

        set({ lastError: messageFor(result.error) });

        return false;

    }

    const tasks = await listTasks();

    set({ tasks, lastError: null });

    return true;

},
```

```

completeTask: async (id) => {

  await completeTask(id);

  const tasks = await listTasks();

  set({ tasks });

},

reopenTask: async (id) => {

  await reopenTask(id);

  const tasks = await listTasks();

  set({ tasks });

}

});

```

`TaskList` starts drawing the state: a checkbox that toggles complete/reopen and a `--done` class for the struck-through look. The completed task stays in the same position, never disappears. The `aria-label` changes with the state, an accessibility touch.

```
// src/presentation/components/TaskList.tsx

import { useTaskStore } from "../../store/task-store-context";

/**
 * Lists the existing tasks in creation order; an empty list is a valid state
 *
 * The completed task stays in the same position, shown as struck through, with
 * the action toggling between complete and reopen
 *
 */

export function TaskList() {

  const tasks = useTaskStore((state) => state.tasks);

  const completeTask = useTaskStore((state) => state.completeTask);

  const reopenTask = useTaskStore((state) => state.reopenTask);

  if (tasks.length === 0) {
```

```
    return <p className="task-list__empty">No tasks yet. Create the first one  
    above.</p>;
```

```
  }
```

```
  return (
```

```
    <ul className="task-list">
```

```
      {tasks.map((task) => (
```

```
        <li
```

```
          key={task.id}
```

```
          className={task.completed ? "task-list__item task-list__item--done"  
: "task-list__item"}
```

```
        >
```

```
          <label className="task-list__check">
```

```
            <input
```

```
              type="checkbox"
```

```
              checked={task.completed}
```

```
              aria-label={task.completed ? `Reopen ${task.title}` : `Complete  
${task.title}`}
```

```

        onChange={() => (task.completed ? reopenTask(task.id) : completeTask(task.id))}

    />

    <span>

        <h3>{task.title}</h3>

        {task.description.length > 0 && <p>{task.description}</p>}

    </span>

</label>

</li>

    )})

</ul>

);

}

```

`main.tsx` only composes the two new use cases at the edge and injects them into the store. The composition pattern is the same as 6.5's, with two more lines.

```
// src/main.tsx
```

```
import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import { LocalStorageTaskRepository } from "../data/local-storage-task-repository";

import { makeCompleteTask } from "../domain/usecases/complete-task";

import { makeCreateTask } from "../domain/usecases/create-task";

import { makeListTasks } from "../domain/usecases/list-tasks";

import { makeReopenTask } from "../domain/usecases/reopen-task";

import { App } from "../presentation/App";

import { createTaskStore } from "../presentation/store/task-store";

import { TaskStoreProvider } from "../presentation/store/task-store-context";

import "../index.css";

// Composition at the edge: the concrete repository is instantiated here and
// injected into the use cases; only at this point do the layers meet.

const repository = new LocalStorageTaskRepository();
```



```
const store = createStore(

  makeCreateTask(repository),

  makeListTasks(repository),

  makeCompleteTask(repository),

  makeReopenTask(repository)

);

const rootElement = document.getElementById("root");

if (rootElement === null) {

  throw new Error("Element #root not found in index.html.");

}

createRoot(rootElement).render(

  <StrictMode>

    <TaskStoreProvider store={store}>
```

```
    <App />

    </TaskStoreProvider>

    </StrictMode>

  );
```

Tests

The `InMemoryTaskRepository` fake gains `getById` and `update`, mirroring what the real repository started offering. The two need to stay in sync, otherwise the tests would stop proving the production behavior.

```
// test/helpers/in-memory-task-repository.ts

import type { Task } from "../../src/domain/entities/task";

import type { TaskRepository } from "../../src/domain/repositories/task-repository";

/** In-memory implementation of `TaskRepository` for domain and store tests. */

export class InMemoryTaskRepository implements TaskRepository {

  private readonly tasks: Task[] = [];
```

```
    async getAll(): Promise<Task[]> {

        return [...this.tasks].sort((a, b) => a.createdAt.localeCompare(b.createdAt));

    }

    async getById(id: string): Promise<Task | null> {

        return this.tasks.find((task) => task.id === id) ?? null;

    }

    async add(task: Task): Promise<void> {

        this.tasks.push(task);

    }

    async update(task: Task): Promise<void> {

        const index = this.tasks.findIndex((existing) => existing.id === task.id)
        ;

        if (index >= 0) {
```

```
        this.tasks[index] = task;

    }

}

}
```

The `CompleteTask` tests cover the four cases of the state model, including the one that proves completing does not reorder the list, and the one that proves idempotency.

```
// test/domain/complete-task.test.ts

import { describe, expect, it } from "vitest";

import { makeCompleteTask } from "../../src/domain/usecases/complete-task";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository";

const seedOpenTask = async (repository: InMemoryTaskRepository) => {

    const createTask = makeCreateTask(repository, () => "id-1");

    const result = await createTask({ title: "Wash the dishes" });
```

```
    if (result.kind === "failure") {  
  
        throw new Error("seed failed");  
  
    }  
  
    return result.value;  
  
};  
  
describe("CompleteTask", () => {  
  
    it("takes an open task to the completed state", async () => {  
  
        const repository = new InMemoryTaskRepository();  
  
        const task = await seedOpenTask(repository);  
  
        const completeTask = makeCompleteTask(repository);  
  
        const result = await completeTask(task.id);  
  
        expect(result.kind).toBe("success");  
    }  
});
```

```

    expect((await repository.getById(task.id)).completed).toBe(true);

});

it("is idempotent: completing an already completed task stays success, with
out changing the state", async () => {

    const repository = new InMemoryTaskRepository();

    const task = await seedOpenTask(repository);

    const completeTask = makeCompleteTask(repository);

    await completeTask(task.id);

    const result = await completeTask(task.id);

    expect(result.kind).toBe("success");

    expect((await repository.getById(task.id)).completed).toBe(true);

});

it("does not change the task's position on the list when completing", async
() => {

```

```

const repository = new InMemoryTaskRepository();

let tick = 0;

const createTask = makeCreateTask(repository, () => `id-${tick}`, () => new Date(2026, 0, 1, 0, 0, tick++));

await createTask({ title: "First" });

await createTask({ title: "Second" });

await createTask({ title: "Third" });

const completeTask = makeCompleteTask(repository);

await completeTask("id-1");

const titles = (await repository.getAll()).map((task) => task.title);

expect(titles).toEqual(["First", "Second", "Third"]);

});

it("returns task-not-found when the task does not exist", async () => {

const repository = new InMemoryTaskRepository();

```

```

    const completeTask = makeCompleteTask(repository);

    const result = await completeTask("nonexistent");

    expect(result).toEqual({ kind: "failure", error: { kind: "task-not-found"
    } });

    });

});

```

The `ReopenTask` tests, symmetric to the complete ones.

```

// test/domain/reopen-task.test.ts

import { describe, expect, it } from "vitest";

import { makeCompleteTask } from "../../src/domain/usecases/complete-task";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeReopenTask } from "../../src/domain/usecases/reopen-task";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository"
;

```



```
const seedCompletedTask = async (repository: InMemoryTaskRepository) => {

    const createTask = makeCreateTask(repository, () => "id-1");

    const result = await createTask({ title: "Wash the dishes" });

    if (result.kind === "failure") {

        throw new Error("seed failed");

    }

    await makeCompleteTask(repository)(result.value.id);

    return result.value;

};

describe("ReopenTask", () => {

    it("takes a completed task back to the open state", async () => {

        const repository = new InMemoryTaskRepository();
```

```
const task = await seedCompletedTask(repository);

const reopenTask = makeReopenTask(repository);

const result = await reopenTask(task.id);

expect(result.kind).toBe("success");

expect((await repository.getById(task.id)).completed).toBe(false);

});

it("is idempotent: reopening an already open task stays success, without changing the state", async () => {

const repository = new InMemoryTaskRepository();

const createTask = makeCreateTask(repository, () => "id-1");

const created = await createTask({ title: "Already open" });

const taskId = created.kind === "success" ? created.value.id : "";

const reopenTask = makeReopenTask(repository);
```

```
const result = await reopenTask(taskId);

expect(result.kind).toBe("success");

expect((await repository.getById(taskId)).completed).toBe(false);

});

it("returns task-not-found when the task does not exist", async () => {

  const repository = new InMemoryTaskRepository();

  const reopenTask = makeReopenTask(repository);

  const result = await reopenTask("nonexistent");

  expect(result).toEqual({ kind: "failure", error: { kind: "task-not-found"
} });

});

});
```

The production repository test gains two new cases: the `update` that does not reorder and the backward-compatible read of old tasks without `completed`.

```
// test/data/local-storage-task-repository.test.ts

import { describe, expect, it } from "vitest";

import { LocalStorageTaskRepository } from "../../src/data/local-storage-task-repository";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { MemoryStorage } from "../../helpers/memory-storage";

describe("LocalStorageTaskRepository", () => {

  it("returns an empty list when nothing was saved", async () => {

    const repository = new LocalStorageTaskRepository(new MemoryStorage());

    expect(await repository.getAll()).toEqual([]);

  });
});
```

```
it("preserves the tasks between sessions (closing and reopening the app)",
  async () => {

    const storage = new MemoryStorage();

    let tick = 0;

    const createTask = makeCreateTask(

      new LocalStorageTaskRepository(storage),

      () => `id-${tick}`,

      () => new Date(2026, 0, 1, 0, 0, tick++)

    );

    await createTask({ title: "First" });

    await createTask({ title: "Second" });

    // A new instance over the same Storage simulates reopening the app.

    const reopened = new LocalStorageTaskRepository(storage);

    const titles = (await reopened.getAll()).map((task) => task.title);

    expect(titles).toEqual(["First", "Second"]);
```

```
});
```

```
it("update persists the new state without reordering the list", async () =>  
{
```

```
    const storage = new MemoryStorage();
```

```
    let tick = 0;
```

```
    const repository = new LocalStorageTaskRepository(storage);
```

```
    const createTask = makeCreateTask(repository, () => `id-${tick}`, () => new  
Date(2026, 0, 1, 0, 0, tick++));
```

```
    await createTask({ title: "First" });
```

```
    await createTask({ title: "Second" });
```

```
    const second = await repository.getById("id-1");
```

```
    await repository.update({ ...second!, completed: true });
```

```
    const reopened = new LocalStorageTaskRepository(storage);
```

```
    const all = await reopened.getAll();
```

```
    expect(all.map((task) => task.title)).toEqual(["First", "Second"]);
```

```

    expect(all.find((task) => task.id === "id-1")?.completed).toBe(true);

});

it("reads old tasks without the completed field as open", async () => {

    const storage = new MemoryStorage();

    storage.setItem(

        "todo-app.tasks",

        JSON.stringify([
            { id: "x", title: "Old", description: "", createdAt: "2026-01-01T00:00:00.000Z" }
        ])
    );

    const tasks = await new LocalStorageTaskRepository(storage).getAll();

    expect(tasks[0].completed).toBe(false);

});

});

```

Finally, the store tests gain two cases (complete reflects; reopen reverts) on top of the four that already existed from 6.5, and `buildStore` starts injecting the two new use cases.

```
// test/presentation/task-store.test.ts

import { describe, expect, it } from "vitest";

import { makeCompleteTask } from "../../src/domain/usecases/complete-task";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeListTasks } from "../../src/domain/usecases/list-tasks";

import { makeReopenTask } from "../../src/domain/usecases/reopen-task";

import { createTaskStore } from "../../src/presentation/store/task-store";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository"
;

const buildStore = () => {

  const repository = new InMemoryTaskRepository();

  return createTaskStore(
```



```
    makeCreateTask(repository),

    makeListTasks(repository),

    makeCompleteTask(repository),

    makeReopenTask(repository)

);

};

describe("task store", () => {

    it("loads the list of existing tasks", async () => {

        const store = buildStore();

        await store.getState().createTask("Task A", "");

        await store.getState().loadTasks();

        expect(store.getState().tasks.map((task) => task.title)).toEqual(["Task A"]);

    });
});
```

```
it("creating successfully updates the list and clears the error", async ()
=> {

    const store = buildStore();

    const created = await store.getState().createTask("New", "a description")
;

    expect(created).toBe(true);

    expect(store.getState().tasks).toHaveLength(1);

    expect(store.getState().lastError).toBeNull();

});

it("a required-title failure exposes a message without changing the list",
async () => {

    const store = buildStore();

    const created = await store.getState().createTask("  ", "");

    expect(created).toBe(false);
```

```
expect(store.getState().tasks).toHaveLength(0);

expect(store.getState().lastError).toBe("The title is required.");

});

it("a duplicate-title failure exposes a specific message", async () => {

  const store = buildStore();

  await store.getState().createTask("Duplicate", "");

  const created = await store.getState().createTask("Duplicate", "");

  expect(created).toBe(false);

  expect(store.getState().lastError).toBe("A task with this title already exists.");

  expect(store.getState().tasks).toHaveLength(1);

});

it("completing a task reflects the completed state on the list", async () => {
```

```
const store = buildStore();

await store.getState().createTask("Finish the report", "");

const taskId = store.getState().tasks[0].id;

await store.getState().completeTask(taskId);

expect(store.getState().tasks[0].completed).toBe(true);

});

it("reopening a completed task returns it to the open state", async () => {

  const store = buildStore();

  await store.getState().createTask("Finish the report", "");

  const taskId = store.getState().tasks[0].id;

  await store.getState().completeTask(taskId);

  await store.getState().reopenTask(taskId);
```

```
    expect(store.getState().tasks[0].completed).toBe(false);  
  
  });  
  
});
```

The portrait of the lap

A one-line sentence went in. `clarify` pulled out of it four decisions that were hidden, and each one turned into a requirement, then design, then code. The domain grew without contorting itself, and the data and presentation layers followed behind, fitting into the same places as always.

The comparison with 6.5 is what is worth keeping: the architecture did not move. In Chapter 8.5, the filtering feature shows an even leaner case, in which almost nothing new is needed in the domain, because the right decision is to derive the view from the state that already exists, instead of duplicating it.

8 - Filtering Tasks: The Spotlight on `plan` and `checklist`

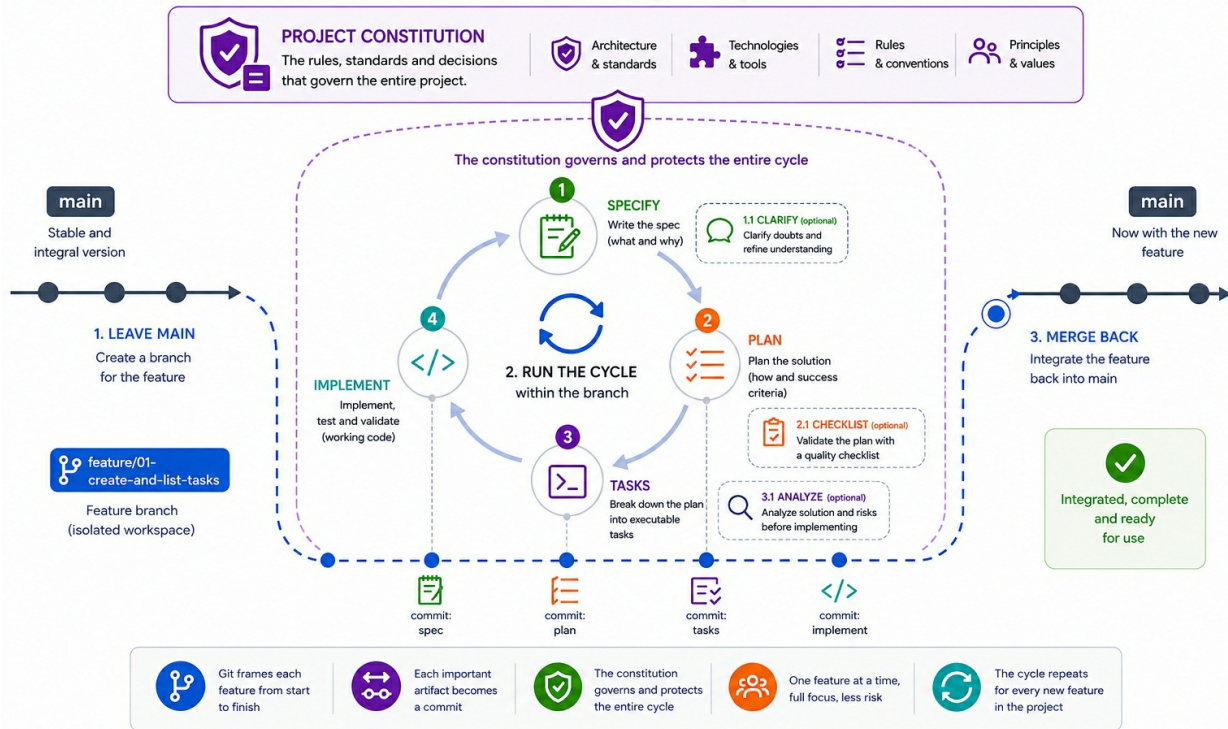
The third loop starts on `main`

The `main` branch is stable again. Two features live in it: creating and listing tasks, completing and reopening tasks. You are going to start the third one the same way you started the other two: not by touching `main`, but by leaving it. `specify` creates the branch `003-filtrar-tarefas`, and it is inside it that the whole cycle happens, until the merge hands the result back to the main line.

The map has not changed. It is worth reopening it for just a moment, not to relearn it, but to get your bearings:

THE COMPLETE SDD CYCLE

One feature, from nothing to integrated code



It is the same diagram from Chapter 5; what changes is only where you stand on it. This time the spotlight does not fall on a single step, but on two neighboring ones, **plan** and **checklist**, which went unexamined in the first two loops.

The feature of the moment is **filtering tasks by state**: seeing only the open ones, only the completed ones, or all of them. It is the best possible showcase for this pair of steps, and the reason lies in the nature of the problem. Completing a task, in the previous chapter, hid decisions about the **what**. Filtering lays bare decisions about the **how**: where to keep the active view, how to derive the visible list, which layers take part. These are questions of construction, and that is exactly what **plan** exists to answer. The **checklist**, right beside it, makes sure nothing was left half done before it turned into tasks.

Why two steps under the same light, and not one at a time? Because they work as a pair. `plan` produces the design of the construction; `checklist` is the gate that examines that design (along with the spec) before it turns into tasks. Seeing the two in the same chapter makes visible something that separating them would hide: planning well and checking the plan are neighboring moves, and it is on the border between them that the cheapest errors to fix show up.

`specify` and `clarify` , at a light pace

You already know how to run the first steps, so let us go through them at a light pace, without re-explaining the commands. The `specify` of the filter is short, because the intent is small: the person wants to look at the list through a slice of state. Three views, one purpose. The spec describes what each view shows and why, and it stops on purpose before saying how.

`clarify` raises the loose ends. The most revealing one here is ordering: within a view, in what order do the tasks appear? It could be by date, by title, by priority. The clarified answer closes the door without inventing a mechanism: the order is the creation order, oldest first, the same as always, and ordering some other way is **out of scope**. Notice the pattern: `clarify` is not choosing technology, it is removing ambiguity from the intent. Technology is the next step's business.

There is a second end `clarify` ties off, with the same discipline: does the chosen view persist between sessions? The answer is no. Switching to "completed" is a gesture of the current session, not a saved preference; when the app reopens, you go back to the default slice. The filter's spec leaves `clarify` short and agnostic:

three views, creation order, no memory between sessions, and not a word about framework or data structure. Spec approved, `/clear`, and the next step starts from the artifact.

The what is already decided; the how is missing

We arrive at the thesis that holds up the chapter. There is a sharp border between what the spec decides and what `plan` decides, and filtering tasks is the perfect case for seeing it, because the problem sits right on top of that line.

The rule is practical and fits in one sentence: **perception and intent belong to the spec; mechanism belongs to `plan`**. Everything that describes what the person perceives or wants (which states they see, what each view means, which slice appears when the screen opens) belongs to the **what**, and the spec already fixed that. Everything that describes how the machine carries it out (where the view's state lives, how the list is derived, which layer does what) belongs to the **how**, and that is still open.

It is worth exercising with concrete pairs from the filter itself. "The person sees only the open tasks" is perception: spec. "The visible list is a projection of the complete list, computed on each render" is mechanism: `plan`. "On opening, the open ones appear" is intent: spec. "The active view is kept in the presentation layer, outside `localStorage`" is mechanism: `plan`. Notice that neither side is more important; they are answers to different questions, asked at different times. Mixing them is the classic mistake: when the how leaks into the spec, it starts to talk about a solution instead of the problem, and it locks down decisions that did not even need to be made yet.

With the border in mind, the spotlight can turn on over the right side of it.

Spotlight: `plan` decides the how

`plan` reads the clarified spec and turns it into a plan of construction. It invents no new requirement: it starts from the **what** already fixed and answers only the **how**. In the To-Do, this has a consequence worth stating right away, because it tends to confuse.

Technology lives in `plan`, but the stack was already chosen

By the project's constitution, the choice of technology belongs to `plan`, not the spec (it is the principle of technology agnosticism: the spec names no framework). But the To-Do's stack is not new: it was decided in the `plan` of the first feature, back in Chapter 6, when React with TypeScript in the interface, Zustand for the state and `localStorage` behind a repository layer were settled. The filter's `plan` **inherits** that stack, without reopening the discussion. You do not re-choose a language for every feature; you choose once and keep the coherence. That is why the spotlight of this `plan` does not fall on "which technology", but on the design decisions the new feature requires.

Deep dive: why the choice of technology does not belong in the spec. Keeping the stack out of the spec is what lets it survive a change of technology, and purism has nothing to do with it. A good spec of the filter stays true if tomorrow the To-Do swaps React for something else: it talks about views and states, not components. If the framework had leaked into it, every change of technology would force a rewrite of

the document that describes the problem, and the problem did not change. Deciding technology once, in `plan`, and letting it be inherited by the following features keeps each layer of decision in its place and each artifact with the useful life it deserves.

The three how decisions

There are three questions the spec left open, and `plan` answers each one. In the real artifact of the filter, they show up like this:

1. **Where the active view lives:** in the presentation layer (store), as `filter`, initialized to "open". It is screen state, not business rule, and it is not persisted.
2. **How the visible list is derived:** by a pure projection in the domain (`selectByFilter`), which takes the complete list and the criterion and returns the subset, on each render; with no second list kept nor a new query to the repository on each view switch.
3. **Which layers take part:** the view chooses the view and displays; the presentation layer keeps the active view; the domain decides which tasks match. The repository does not change.

Look at what each decision does. The first separates two things the word "state" mixes: the state of a task (open or completed, which is a business rule and lives in the entity) and the state of the view (which slice is on the screen right now, which is a presentation detail and vanishes when the session ends). The second chooses to derive instead of duplicate: the visible list is not a second source of truth that has to be kept in sync, it is a function of the complete list. The third distributes the work

across the layers Chapter 4 named, each in its place. Three construction answers, none of them belonging to the spec, all necessary before writing the first line of code.

To feel why these decisions are not detail, follow a concrete interaction. You are on the "open" view and mark a task as completed. What happens to the screen? Since the visible list is derived from the complete list on each render, and the task now no longer matches the "open" criterion, it simply disappears from the view at that same instant, with no special code to remove it from there. Switch to "all" and it reappears, struck through, in the same place as before. That whole behavior comes for free from the decision to derive instead of keep: there is no list to update, no synchronization to write, no risk of the view and the data disagreeing. The choice of mechanism, made in `plan`, is what makes the behavior simple. A worse choice here (two lists, updated by hand) would make the same feature demand twice the code and open the door to subtle bugs.

And this is where the "how" shows its most important face: it depends on the choices made for **this** project. Deriving the list by filtering in memory is the right answer because the To-Do keeps everything in `localStorage` and already loads all the tasks at once; with the whole list in hand, filtering is just a cheap projection. Swap the persistence for a real database, with thousands of tasks that do not fit (nor are worth loading) at once, and the best decision becomes another: a **new method on the repository** would make much more sense, something like "bring only the open ones", letting the database filter with a query and return only the slice. The criterion (which tasks match each view) would still be a domain rule, but the **place** where it runs would change from an in-memory projection to a query on the database. Notice what this reveals: `plan` does not have a universal answer for "how to filter". It has the right answer for the stack **this** project chose. Change the foundation in `plan`, and the

construction decisions that rest on it change along with it. That is why the "how" lives in `plan`, and not in the spec: the filter's spec ("see only the open ones") would stay identical in both worlds; only the plan of construction gets rewritten when the technology underneath it is different.

Deep dive: why the view is presentation state, and not task state. Trying to keep the active view together with the data (in `localStorage`, for example) seems harmless, but it confuses two lifetimes. The tasks' data lasts between sessions; the view you are looking at right now is ephemeral, it dies when you close the tab and is reborn in the default slice. Treating it as screen state keeps that distinction honest and avoids meaningless questions, like "which view was active last time" when no one asked for it to be remembered.

The constitution check has teeth

`plan` is not approved just because it solves the problem. Before going on, it passes through the **constitution check**: the verification, built into `plan` itself, that the plan respects the project's rules.

It is worth pausing on the title of this section, because the image recurs further on. To say a verification "has teeth" is to say it bites: that it has real power to **reject** and **halt progress**, not just to log a warning that everyone reads and ignores. The opposite, a "toothless" verification, is the one that points out a problem but lets it through anyway, becoming a rubber stamp: a formality fulfilled with no consequence. The question that matters about any control gate is that one: can it say no and hold the work right

there, or does it only grumble while it lets things go by? To show that the constitution check is of the first kind, it is worth following a draft that was **rejected** by it.

The draft that puts the rule in the view

The shortest way out to filter is also the most tempting: let the list component decide. The view (the rendering layer) would receive all the tasks and, right there in the interface code, would choose which ones to render according to the view. It works, it runs, it shows the right thing on the screen. And even so the constitution check blocks it, with two objections logged in the plan:

- It breaks **Principle II (Isolated Business Logic)**: "which tasks appear" is a business decision and it was in the interface. The constitution requires that no business decision live in the UI.
- It breaks **Principle I (Layered Architecture)**: the rule would cross the layer boundary, gluing the domain to the view and breaking the direction of dependency.

Pause a moment on the first point, because it is subtle. "Which tasks appear" sounds like a screen matter, after all it is the screen that shows them. But deciding the criterion (a task matches the "open" view when it is not completed) is a rule about the meaning of the data, and that is business, not presentation. The To-Do's constitution was written precisely so that the interface only collects input and displays a result, without making that kind of decision. The draft violated this without seeming to violate it.

The correction gives the rule back to the domain

The fix is straightforward, and the plan logs it:

The selection rule goes back to the domain, as the pure projection `selectByFilter`. The interface is responsible only for choosing the active view and displaying the result the domain returns.

After the correction, each layer does only its own job: the domain decides which tasks match, the presentation keeps which view is active, the view paints the list it receives. Where to draw that boundary when the case is less clear-cut than this one belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it here: the test the To-Do used fits in one question, whether the thing decides the meaning of the data or only its appearance. The constitution check passes, and passes for real, not as a concession. That is the chapter's point about this step: far from being a poster on the wall that everyone agrees with and no one follows, the constitution reappears at an exact moment of the cycle, with the power to reject a plan that was going to work, and forces the correction while it is still cheap, in text, before it turns into code.

Spotlight: `checklist` as a quality gate

Plan corrected, there is still a gate before breaking the work into tasks: the `checklist`. If `plan` answers the how, `checklist` asks something else: are the spec and the plan **complete, clear and**

consistent enough to turn into tasks without rework? It is a quality gate, and it works paired with `plan`, checking the whole before construction begins.

To show that this gate also has teeth, it needs to catch something. And it did, a small, concrete hole. The spec fixed the three views with care, but it had a silence: it did not say **which view appears when the screen opens**. The checklist item is literal:

The initial view on opening the screen is declared.

Completeness hole found: the spec fixed the three views but did not say which one appears when the app opens. Fixed in the spec with FR-005 (default view = "open"); a To-Do opens showing what is left to do.

It is a hole easy to miss. Each view in isolation is well defined; what was missing was the starting state, the question "and when they have not chosen anything yet?". Without the `checklist`, that silence would turn into code anyway: someone, at implementation time, would pick a default on impulse, and the product decision (a To-Do opens showing what is left to do, so the default is the open ones) would be made by accident, hidden in a line of implementation. The gate forces the decision to rise to where it belongs, the spec, and to cost almost nothing: one corrected sentence instead of a redone behavior.

Corrected in the `checklist`, the hole cost one sentence in the spec, written in seconds. Discovered later, with the app running, it would cost a whole investigation: report, diagnosis, a rushed decision, touched code, redone test. That is the sense of calling the `checklist` a gate **with teeth**: the cheap place to pay a debt that, ignored, only gets more expensive.

It is worth separating this gate from `clarify`, because the two ask questions and can be confused. `clarify` acts **before**, on the spec alone, and attacks **ambiguity**: what here admits more than one reading? `checklist` acts **after**, on the spec and plan together, and attacks **completeness and consistency**: what here is missing or contradicts itself? Ordering was a `clarify` case (there were competing readings); the initial view was a `checklist` case (there was no ambiguity, there was a gap). Different questions, different moments, the same discipline of resolving it in the text before paying for it in the code.

Deep dive: why the filtered list is derived, and not kept. The plan chose to compute the visible list on each render, instead of keeping a second synchronized list. The difference matters beyond style: two lists that need to agree are two chances to disagree. Deriving from a single source (the complete list, filtered by a pure function) eliminates the whole class of bugs where the view shows something the data no longer says. It is the same logic of "a single source of truth", applied to the presentation layer.

Closing the loop, at a light pace

From here the path is known, and we cover it fast. The validated plan turns into **tasks**: the list of steps in order, tests before the code, each file in its place. For the filter, the tasks fall almost by themselves out of the decisions already made, because the corrected `plan` and the closed `checklist` left no doubt: a test and the selection function in the domain, the active view in the store with the "open" default, the selector in the interface, the derived list in the display. When the plan is clean, the tasks step is almost transcription. The **analyze** checks the consistency between spec,

plan and tasks, looking for the mismatch left over (a requirement with no task, a task with no requirement). And the **implement** executes: the filter function in the domain, the active view in the store, the selector in the interface, the tests in the green. The feature that began as a sentence now runs.

With the tests passing, you commit the artifacts, one by one, and **merge** the branch back into `main`. The main line is stable again, now with three capabilities where there were two.

The bridge to Ch. 9

The next feature in the backlog is **deleting a task**, and it brings a new difficulty: deleting is a delicate action, one that asks for careful steps and can conflict with what the previous features already did. That is why, in Chapter 9, the spotlight shifts to `tasks` and `analyze`, the steps that break the work into safe steps and check whether the whole stays coherent.

8.5 - Filtering Tasks in Practice: the Whole Loop, File by File

How to read this chapter

Chapter 8 followed the third lap of the cycle with the spotlight on `plan`. Here you see the whole lap, the way it was recorded in the example app's repository, in the `003-filtrar-tarefas` feature.

This lap has two peculiarities worth noting. The first: in the app's repository, it was recorded in a **single commit** (`b3f1cc1`), with specification, plan, tasks and implementation together. That is why the chapter treats it as a single lap, but the order of the loop stays visible in the artifacts. The second: it is the lap where `plan` works the most. It shows two episodes Chapter 8 highlighted, the `checklist` catching a completeness hole in the spec (which is the default view?) and the **constitution check rejecting the first draft of the plan** (which put the filter rule inside the widget). Both were logged in the artifacts, and you read them below, exactly as the agent wrote them.

As in the previous laps, the artifacts appear as markdown (headings lowered) and the code in blocks with a context sentence before each one.

The input: the `/speckit-specify` prompt

"Filter the task list by state: see only the open ones, only the completed ones, or all of them. It rests on the complete/reopen feature. Out of scope for this feature: text search, combined filters, saved filters and configurable ordering."

What `specify` **and** `clarify` **returned:** `spec.md`

Notice the box right after the clarifications: it records, within the spec itself, the completeness correction the `checklist` prompted.

Feature Specification: Filter tasks by state

Feature Branch: `003-filtrar-tarefas`

Created: 2026-06-27

Status: Draft

Input: User description: "Filter the task list by state: see only the open ones, only the completed ones, or all of them. It rests on the complete/reopen feature. Out of scope for this feature: text search, combined filters, saved filters and configurable ordering."

Clarifications

Session 2026-06-27

- Q: Besides filtering by state, should the feature allow ordering the list (by date, by title)? → A: Not in this feature. The order stays the creation order (oldest first), inherited from 001. Configurable ordering is out of scope (YAGNI); if it is wanted,

it becomes its own feature later.

- Q: Should the chosen filter survive closing and reopening the app? → A: No. The filter is an ephemeral view state of the session; when the app reopens, it goes back to the default view. Persisting the chosen view would be extra state with no proven need (YAGNI).
- Q: Does the count or the label of each view need to appear (e.g. "Open (3)")? → A: No. Just the three view selectors, without count badges; the minimum needed for the person to switch views.

Completeness correction (incorporated after the `checklist`, see `checklists/requirements.md`): the initial version of this spec fixed the three views but **did not declare which of them appears when the screen opens**. The `checklist` pointed out the gap; the correction fixes the **default view as "open"** (FR-005).

User Scenarios & Testing (mandatory)

User Story 1 - See only the open tasks (Priority: P1)

The person has a list that mixes what they already did and what is left to do. They want to focus on what is still pending. On choosing the "open" view, the list starts to show only the open tasks, hiding the completed ones without deleting them.

Why this priority: It is the central reason for the feature and the most frequent view of use of a To-Do: looking at what is still left. Delivered on its own, it already produces value.

Independent Test: It can be tested in isolation by marking some tasks as completed, choosing the "open" view and checking that only the open ones appear.

Acceptance Scenarios:

1. **Given** a list with open and completed tasks, **When** the person chooses the "open" view, **Then** the list shows only the open tasks, in creation order.

User Story 2 - See only the completed ones or all (Priority: P2)

The person wants to review what they already finished, or go back to seeing the whole list. They switch to the "completed" view (only what is done) or "all" (the complete list, open and completed together).

Why this priority: It completes the set of views and gives the feature symmetry, but the "open" view already delivers the central value. Hence P2.

Independent Test: It can be tested by switching between "completed" and "all" and checking that each view shows exactly the right subset.

Acceptance Scenarios:

1. **Given** a list with open and completed tasks, **When** the person chooses the "completed" view, **Then** the list shows only the completed ones.
2. **Given** the same list, **When** the person chooses the "all" view, **Then** the list shows all the tasks, in creation order.

Edge Cases

- When no task satisfies the chosen view (e.g. "completed" with none completed), the list appears **empty** within that view; an empty list is a valid state, not an error (inherited from 001).
- Completing or reopening a task while a view is active **re-evaluates** the visible list: a task completed in the "open" view stops appearing there at once; reopening it brings it back.

- The feature **does not reorder** the list: within any view, the order is the creation order (oldest first), preserved from 001.

Requirements (mandatory)

Functional Requirements

- **FR-001:** The system **MUST** offer three views of the list by state: **open, completed** and **all**.
- **FR-002:** The system **MUST**, in the "open" view, show only open tasks; in the "completed" view, only completed ones; in the "all" view, all the tasks.
- **FR-003:** The system **MUST** preserve, within each view, the creation order of the tasks (oldest first), without reordering.
- **FR-004:** The system **MUST** re-evaluate the visible list when a task changes state (complete/reopen), reflecting the active view immediately.
- **FR-005:** The system **MUST** open, on startup, in the **default view "open"**. (Added in the `checklist` correction: the spec did not declare the initial view.)
- **FR-006:** The system **MUST** treat a view with no matching tasks as an empty list (a valid state), not as an error.

Non-Goals

- Explicitly out of scope: **text search, combined filters (several criteria at once), saved filters, configurable ordering, counters/badges per view and persistence of the chosen view between sessions (YAGNI).**

Key Entities

- **Task:** reused from 001/002 without change. The already existing state (open/completed) is the filter criterion; this

feature **does not** add fields to the task.

- **View (filter):** which subset of the list is on display: **open**, **completed** or **all**. It is a **view** state of the session, not an attribute of the task nor persisted data. The default view is "open".

Success Criteria (mandatory)

Measurable Outcomes

- **SC-001:** The person can, in a mixed list, see only the open tasks by choosing the "open" view.
- **SC-002:** The person can see only the completed ones and go back to seeing all, by switching the view.
- **SC-003:** On opening the app, the initial view is "open".
- **SC-004:** Completing a task in the "open" view removes it from the visible list at once; reopening it brings it back.

Assumptions

- The feature extends the create/list (001) and complete/reopen (002) ones, already integrated into `main`; it reuses the Task entity, the repository and the existing layered architecture, without touching the persistence.
 - The To-Do constitution governs the feature: Simplicity/YAGNI, layered architecture, business logic isolated from the UI, error as value and TDD.
 - The chosen view's state is ephemeral (session); it is not persisted, as clarified.
-

The requirements review:

checklists/requirements.md

This is the artifact that caught the hole. See the bold item in the middle of the list: it describes the gap found, the correction applied and the fact that it was rechecked only after being fixed. The `checklist` changes the spec; nothing in it is decorative.

Checklist: Requirements Quality

Review of `spec.md` and `plan.md` before generating the tasks. It validates completeness, clarity and consistency; an item that fails is corrected [here](#), before it turns into task and code.

- Each functional requirement (FR-001..FR-006) is verifiable and unambiguous.
- The three views (open / completed / all) are defined, with acceptance scenarios.
- **The initial view on opening the screen is declared.**
Completeness hole found: the spec fixed the three views but did not say which one appears when the app opens. Fixed in the spec with FR-005 (default view = "open"); a To-Do opens showing what is left to do. Item rechecked after the correction.
- The criterion of each view (which state the task needs to have) is unequivocal and covers the three cases.
- The order within each view is defined (creation, oldest first — FR-003), without reordering.
- The behavior on changing state with a view active is defined (FR-004).
- An empty list within a view is a valid state, not an error (FR-006).
- The non-goals are explicit (no search, combined filters, saved filters, ordering, counters, view persistence).

- The spec stays agnostic of language/framework/state; the view is view state, not a persisted attribute of the task (decisions in the plan).
 - Each Success Criterion (SC-001..SC-004) is measurable and traceable to an FR.
 - The `plan` keeps the filtering rule in the domain (the UI only chooses the view and displays), without violating the constitution (see Constitution Check in `plan.md`).
-

What `plan` decided: `plan.md`

Read carefully the "Conflict caught by the constitution check" section: it narrates the first draft of the plan (filter embedded in the list's JSX), why it was rejected (breaks Principles I and II) and what the correction is (the rule goes back to the domain, as a pure projection). It is `plan` correcting itself before it turns into code.

Implementation Plan: Filter tasks by state

Branch: `003-filtrar-tarefas` | **Spec:** `spec.md`

Input: Feature specification in `specs/003-filtrar-tarefas/spec.md`

Summary

Add three **views** of the list by state (open / completed / all), reusing the `completed` state the task has had since 002. The spec says the **what** (which subsets the person sees); this plan decides the **how**: where the active view lives, how the visible list is derived from the complete list and which layers take part. The

rule that decides **which tasks appear** lives in the **domain** (a pure projection of the list by a state criterion); the presentation only keeps **which view is active** and displays the result. No new data is persisted: the view is view state of the session.

Technical Context

Language/Version: TypeScript 5.x on Node 20+; target: browser (React web). **Inherited from 001/002.**

Main dependencies: React 18, Vite, Zustand. **No new dependencies** (the stack was chosen in the `plan` of 001 and is inherited here).

Persistence: no change. The active view is **not** persisted (clarification); `localStorage` behind `TaskRepository` stays the same.

Tests: Vitest for the domain filter (pure function, no DOM) and for the store (view switching).

Project type: single-page, single-user web application, no backend.

How decisions (what the spec did not fix)

1. **Where the active view lives:** in the **presentation layer** (Zustand store), as `filter: TaskFilter`, initialized to `"open"` (default view "open", FR-005). It is screen state, not a business rule, and it is not persisted.
2. **How the visible list is derived:** by a **pure projection in the domain** (`selectByFilter(tasks, filter)`), which takes the complete list (already loaded in memory) and the criterion and returns the subset. The visible list is **derived** on each render; there is no second list kept nor a new query to the repository on each view switch.

3. **Which layers take part:** `view` (TaskList/view selector) → `state/presentation layer` (the store keeps the active view) → `domain` (the `selectByFilter` rule decides which tasks match). The `repository` does not change: it keeps returning all the tasks; filtering is a derivation, not a new data query.

Constitution Check

Principle	How the plan meets it
I. Layered Architecture	The selection rule stays in the domain (<code>task-filter.ts</code>); the store only keeps the active view; the view only renders. Dependencies point toward the domain.
II. Isolated Business Logic	The decision of which tasks appear is a pure domain function, testable without a UI; React/Zustand do not decide the criterion.
III. Error as Value	There is no new predictable failure: filtering does not fail; an empty view is success (empty list), not error.
IV. TDD	Tests of the domain projection (three views, order preserved, empty) and of the store (view switching, re-evaluation on completing) before the implementation.
V. Simplicity/YAGNI	No search, combined filters, saved ones, counters, ordering nor view persistence; a single presentation <code>filter</code> field and a pure function.
VI. Technology	The spec names no stack; the stack

Agnosticism	(React/TS/Vite/Zustand) is inherited from the <code>plan</code> of 001. This plan's decisions are of design (where the view lives, how to derive), not of new technology.
-------------	---

Result: PASS — after the correction logged below.

Conflict caught by the constitution check (and the correction)

The **first draft** of this plan put the filter **inside the list widget**: the `TaskList` component would receive all the tasks and decide, right there in the JSX, which ones to render according to the view (`tasks.filter(...)` embedded in the view). It is the shortest way out, and for that reason the most tempting.

The constitution check rejected this draft:

- It breaks **Principle II (Isolated Business Logic)**: "which tasks appear" is a **business decision** and it was in the interface. The constitution requires that no business decision live in the UI.
- It breaks **Principle I (Layered Architecture)**: the rule would cross the layer boundary, gluing the domain to the view and breaking the direction of dependency.

Correction: the selection rule goes back to the **domain**, as the pure projection `selectByFilter(tasks, filter)`. The interface is responsible only for **choosing the active view** (presentation state in the store) and **displaying** the result the domain returns. The view no longer decides anything about which tasks match; it only paints the derived list.

Traceability clarify/checklist → plan

Decision (clarify/checklist)	Effect on the plan

No configurable ordering	The projection preserves the creation order; it does not reorder.
The view does not persist between sessions	<code>filter</code> lives in the presentation store, outside <code>localStorage</code> .
Default view = "open" (the checklist's hole)	<code>filter</code> initializes to <code>"open"</code> .
Filtering rule isolated (constitution check)	<code>selectByFilter</code> in the domain; the UI only chooses the view.

Project Structure (additions)

```

src/
├── domain/
│   └── entities/task-filter.ts           # new: TaskFilter type + selectByFil
ter (pure projection)
├── presentation/
│   ├── store/task-store.ts             # + filter (default "open") and setF
ilter; tasks stays = complete list
│   └── components/
│       ├── TaskFilterBar.tsx           # new: three view selectors
│       └── TaskList.tsx                # derives the visible ones via selec
tByFilter(tasks, filter) and renders
test/
├── domain/task-filter.test.ts           # new: three views, order preserved,
empty
└── presentation/task-store.test.ts     # + default view "open" and view swit
ching

```

Phase 1 - Design

- View model in data-model.md.
- Manual validation in quickstart.md.

The `plan`'s view model: `data-model.md`

The design artifact of this lap is short and revealing: the Task entity **does not change**. The view is presentation state, and the visible list is a derivation, not a kept piece of data. Telling screen state apart from business rules across a whole project, and not just in this one case, belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it to read the document below: it already says, in two lines, which side each thing landed on.

View model: Filter tasks by state

Task entity

No change. The filter reuses the `completed: boolean` field introduced in 002 (`false` = open, `true` = completed). No new field is added to the task.

View (filter)

The **view** is **presentation** state, not an attribute of the task nor persisted data.

```
type TaskFilter = "all" | "open" | "completed"
```

- "open" → only open tasks (`completed === false`). **Default view.**
- "completed" → only completed tasks (`completed === true`).
- "all" → all the tasks.

Derivation of the visible list

The visible list is a **pure projection** of the complete list by the view's criterion:

```
selectByFilter(tasks, filter) =
  filter === "all"      → tasks
  filter === "open"     → tasks where completed === false
  filter === "completed" → tasks where completed === true
```

- The creation order (by `createdAt`, oldest first) is **preserved**: the projection filters, it does not reorder.
- The projection is **derived on each render** from the complete list; there is no second list kept nor a new query to the repository.
- A view with no matches → an empty list (a valid state).

Layers

```
view (TaskFilterBar chooses the view; TaskList displays)
  | triggers setFilter / reads the visible ones
  ▼
state/presentation layer (store: active filter, default "open")
  | calls
  ▼
domain (selectByFilter decides which tasks match)
```


The direction of dependency points toward the domain. The rule of "which tasks appear" lives in the domain (Principle II); the presentation only keeps the active view and displays.

The manual validation: `quickstart.md`

Manual validation: Filter tasks by state

After `npm test` and `npm run build` are clean, validate through the UI (`npm run dev`):

1. Create three tasks and complete one of them.
 2. **SC-003:** on opening the app, the initial view is "open" — the completed one does not appear.
 3. **SC-001:** in the "open" view, only the open ones appear, in creation order.
 4. **SC-002:** switch to "completed" (only the completed one) and to "all" (all three).
 5. **SC-004:** in the "open" view, complete a visible task — it disappears from the list at once; reopen it (in the "all" or "completed" view) and it goes back to "open".
 6. A view with no matches (e.g. "completed" with none completed) shows an empty list, without an error.
-

The `tasks` execution list: `tasks.md`

The list is the leanest so far: a pure function in the domain, a field in the store, a bar component and the adjustment of the list. The domain grows by a single function.

Tasks: Filter tasks by state

Branch: `003-filtrar-tarefas` | Plan: `plan.md`

TDD order: tests before the corresponding implementation. `[P]` marks tasks that can run in parallel (distinct files, no dependency).

Phase 1 - Domain (core, pure TS)

- **T001** `test/domain/task-filter.test.ts` : tests of `selectByFilter` ("open" view only open; "completed" only completed; "all" everything; creation order preserved; empty view → empty list). **Fails first.**
- **T002** `src/domain/entities/task-filter.ts` : `TaskFilter` type and the pure projection `selectByFilter(tasks, filter)` ; implement until T001 passes.

Phase 2 - Presentation

- **T003** `test/presentation/task-store.test.ts` : store tests (initial view = "open"; `setFilter` switches the view). **Fails first.**
- **T004** `src/presentation/store/task-store.ts` : `filter` field (default "open") and `setFilter` action; `tasks` stays the complete list.
- **T005** `src/presentation/components/TaskFilterBar.tsx` : three view selectors (open/completed/all), reflecting the active view.
- **T006** `src/presentation/components/TaskList.tsx` : derive the visible tasks via `selectByFilter(tasks, filter)` (domain rule) and render them, preserving the styling and actions from 002.

- **T007** `src/presentation/App.tsx` : compose the filter bar above the list.

Phase 3 - Validation

- **T008** `npm test` (all green) and `npm run build` (clean type-check).
 - **T009** Validation of Success Criteria SC-001..SC-004.
-

The code `implement` generated

This is the lap that best proves the material's thesis: the right architectural decision makes the code almost trivial. The filter rule is a pure function of twelve lines, and the rest is fitting things into place.

Domain

The new file of this lap, and the only one in the domain.

`selectByFilter` is a pure projection: it takes the list and the view, returns the subset, with no side effect. It is here that the decision of "which tasks appear" lives, the place the constitution check demanded.

```
// src/domain/entities/task-filter.ts
```

```
import type { Task } from "../task";
```

```
/** View of the list by state. View state, not an attribute of the task. */
```

```
export type TaskFilter = "all" | "open" | "completed";

/**
 * Projects the complete list to the chosen view, preserving the creation order.
 *
 * Deciding which tasks appear is a business rule and lives here in the domain;
 *
 * the interface only chooses the view and displays the result
 *
 */

export const selectByFilter = (tasks: Task[], filter: TaskFilter): Task[] => {

  switch (filter) {

    case "all":

      return tasks;

    case "open":

      return tasks.filter((task) => !task.completed);

    case "completed":
```

```
        return tasks.filter((task) => task.completed);

    }

};
```

Presentation

The store gains a `filter` field (initialized to `"open"`, the default view the `checklist` demanded) and a `setFilter` action. Notice that `tasks` stays the complete list: the store keeps **which** view is active, never the already filtered list.

```
// src/presentation/store/task-store.ts

import { createStore } from "zustand/vanilla";

import type { Task } from "../../domain/entities/task";

import type { TaskFilter } from "../../domain/entities/task-filter";

import type { CreateTaskFailure } from "../../domain/failures/create-task-failure";

import type { CompleteTask } from "../../domain/usecases/complete-task";

import type { CreateTask } from "../../domain/usecases/create-task";

import type { ListTasks } from "../../domain/usecases/list-tasks";
```

```
import type { ReopenTask } from "../../domain/usecases/reopen-task";

export type TaskStoreState = {

  readonly tasks: Task[];

  readonly filter: TaskFilter;

  readonly lastError: string | null;

  loadTasks: () => Promise<void>;

  createTask: (title: string, description: string) => Promise<boolean>;

  completeTask: (id: string) => Promise<void>;

  reopenTask: (id: string) => Promise<void>;

  setFilter: (filter: TaskFilter) => void;

};

const messageFor = (failure: CreateTaskFailure): string => {

  switch (failure.kind) {

    case "title-required":
```

```

        return "The title is required.";

    case "duplicate-title":

        return "A task with this title already exists.";

    }

};

/**

 * Presentation store (Zustand) that orchestrates the domain use cases and ex
poses

 * the state to the UI. It contains no business rule: it only trigger
s

 * `createTask`/`listTasks` and translates the typed failure into a message f
or

 * the screen
.

 *
/

export const createTaskStore = (

    createTask: CreateTask,

```

```
listTasks: ListTasks,  
  
completeTask: CompleteTask,  
  
reopenTask: ReopenTask  
  
) =>  
  
createStore<TaskStoreState>((set) => ({  
  
  tasks: [],  
  
  filter: "open",  
  
  lastError: null,  
  
  loadTasks: async () => {  
  
    const tasks = await listTasks();  
  
    set({ tasks, lastError: null });  
  
  },  
  
  createTask: async (title, description) => {  
  
    const result = await createTask({ title, description });
```



```
    if (result.kind === "failure") {

        set({ lastError: messageFor(result.error) });

        return false;

    }

    const tasks = await listTasks();

    set({ tasks, lastError: null });

    return true;

},

completeTask: async (id) => {

    await completeTask(id);

    const tasks = await listTasks();

    set({ tasks });
```

```

    },

    reopenTask: async (id) => {

        await reopenTask(id);

        const tasks = await listTasks();

        set({ tasks });

    },

    setFilter: (filter) => {

        set({ filter });

    }

    ));

```

The filter bar: three buttons that only choose the active view. The comment makes explicit that it **does not decide which tasks match**, that is the domain's job. The `aria-pressed` communicates the active view to screen readers.

```

// src/presentation/components/TaskFilterBar.tsx

import type { TaskFilter } from "../../domain/entities/task-filter";

```

```

import { useTaskStore } from "../store/task-store-context";

const VIEWS: ReadonlyArray<{ readonly value: TaskFilter; readonly label: string }> = [

    { value: "open", label: "Open" },

    { value: "completed", label: "Completed" },

    { value: "all", label: "All" }

];

/**

 * Selector for the three views of the list. It only chooses the active view
 in the

 * store; it does not decide which tasks match (that rule lives in the domain
 , in

 * `selectByFilter`)
 .

 *
 /

export function TaskFilterBar() {

```

```
const filter = useTaskStore((state) => state.filter);

const setFilter = useTaskStore((state) => state.setFilter);

return (

  <div className="task-filter" role="group" aria-label="Filter tasks by state">

    {VIEWS.map((view) => (

      <button

        key={view.value}

        type="button"

        className={

          view.value === filter ? "task-filter__option task-filter__option-active" : "task-filter__option"

        }

        aria-pressed={view.value === filter}

        onClick={() => setFilter(view.value)}

      >
```

```

        {view.label}

      </button>

    ))}

  </div>

);

}

```

`TaskList` changes in one essential line: it calls `selectByFilter(tasks, filter)` to get the visible ones. The re-evaluation on complete/reopen (FR-004) comes for free, because the visible list is derived on each render. The rest, the struck-through style and the toggled action, is what 002 left.

```

// src/presentation/components/TaskList.tsx

import { selectByFilter } from "../../domain/entities/task-filter";

import { useTaskStore } from "../../store/task-store-context";

/**

* Lists the tasks of the active view in creation order; an empty list is a v
alid

```

```
* state. The choice of which tasks appear comes from the domain (`selectByFilter`);
```

```
* here we only display the result. The completed task is shown as struck through,
```

```
* with the action toggling between complete and reopen  
.
```

```
*  
/
```

```
export function TaskList() {
```

```
  const tasks = useTaskStore((state) => state.tasks);
```

```
  const filter = useTaskStore((state) => state.filter);
```

```
  const completeTask = useTaskStore((state) => state.completeTask);
```

```
  const reopenTask = useTaskStore((state) => state.reopenTask);
```

```
  const visibleTasks = selectByFilter(tasks, filter);
```

```
  if (visibleTasks.length === 0) {
```

```
    return <p className="task-list__empty">No tasks in this view.</p>;
```

```
  }
```

```
return (  
  
  <ul className="task-list">  
  
    {visibleTasks.map((task) => (  
  
      <li  
  
        key={task.id}  
  
        className={task.completed ? "task-list__item task-list__item--done"  
: "task-list__item"}  
  
      >  
  
        <label className="task-list__check">  
  
          <input  
  
            type="checkbox"  
  
            checked={task.completed}  
  
            aria-label={task.completed ? `Reopen ${task.title}` : `Complete  
${task.title}`}  
  
            onChange={() => (task.completed ? reopenTask(task.id) : complet  
eTask(task.id))}  
  
          />  

```

```

        <span>

            <h3>{task.title}</h3>

            {task.description.length > 0 && <p>{task.description}</p>}

        </span>

    </label>

</li>

)}}

</ul>

);

}

```

App only fits the filter bar above the list. One line of composition.

```

// src/presentation/App.tsx

import { useEffect } from "react";

import { TaskFilterBar } from "../components/TaskFilterBar";

import { TaskForm } from "../components/TaskForm";

```



```
import { TaskList } from "../components/TaskList";

import { useTaskStore } from "../store/task-store-context";

export function App() {

  const loadTasks = useTaskStore((state) => state.loadTasks);

  useEffect(() => {

    void loadTasks();

  }, [loadTasks]);

  return (

    <main className="app">

      <h1>To-Do</h1>

      <TaskForm />

      <TaskFilterBar />

      <TaskList />

    </main>
  );
}
```

```
    </main>

  );

}
```

Tests

The tests of the domain projection are pure and direct: no mock, no store, just input and output. It is the sign that the rule ended up in the right place, a pure function is the easiest thing to test there is.

```
// test/domain/task-filter.test.ts

import { describe, expect, it } from "vitest";

import type { Task } from "../../src/domain/entities/task";

import { selectByFilter } from "../../src/domain/entities/task-filter";

const task = (id: string, completed: boolean): Task => ({

  id,

  title: id,

  description: "",
```

```
    createdAt: `2026-06-27T10:00${id}:00.000Z`,

    completed

});

const tasks: Task[] = [task("1", false), task("2", true), task("3", false)];

describe("selectByFilter", () => {

    it("the 'open' view returns only the open ones", () => {

        expect(selectByFilter(tasks, "open").map((t) => t.id)).toEqual(["1", "3"]);

    });

    it("the 'completed' view returns only the completed ones", () => {

        expect(selectByFilter(tasks, "completed").map((t) => t.id)).toEqual(["2"]);

    });

    it("the 'all' view returns the whole list", () => {
```

```

        expect(selectByFilter(tasks, "all").map((t) => t.id)).toEqual(["1", "2",
"3"]);

    });

    it("preserves the creation order, without reordering", () => {

        expect(selectByFilter(tasks, "open")).toEqual([tasks[0], tasks[2]]);

    });

    it("a view with no matches returns an empty list", () => {

        const allOpen: Task[] = [task("1", false), task("2", false)];

        expect(selectByFilter(allOpen, "completed")).toEqual([]);

    });

});

```

The store tests gain two new cases (initial view "open"; `setFilter` switches the view without touching the complete list) on top of the six that already came from the previous laps.

```
// test/presentation/task-store.test.ts
```

```
import { describe, expect, it } from "vitest";

import { makeCompleteTask } from "../../src/domain/usecases/complete-task";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeListTasks } from "../../src/domain/usecases/list-tasks";

import { makeReopenTask } from "../../src/domain/usecases/reopen-task";

import { createTaskStore } from "../../src/presentation/store/task-store";

import { InMemoryTaskRepository } from "../helpers/in-memory-task-repository"
;

const buildStore = () => {

  const repository = new InMemoryTaskRepository();

  return createTaskStore(

    makeCreateTask(repository),

    makeListTasks(repository),

    makeCompleteTask(repository),

    makeReopenTask(repository)
```

```

    );

};

describe("task store", () => {

    it("loads the list of existing tasks", async () => {

        const store = buildStore();

        await store.getState().createTask("Task A", "");

        await store.getState().loadTasks();

        expect(store.getState().tasks.map((task) => task.title)).toEqual(["Task A"]);

    });

    it("creating successfully updates the list and clears the error", async ()
=> {

        const store = buildStore();

```

```
const created = await store.getState().createTask("New", "a description")
;

expect(created).toBe(true);

expect(store.getState().tasks).toHaveLength(1);

expect(store.getState().lastError).toBeNull();

});

it("a required-title failure exposes a message without changing the list",
  async () => {

    const store = buildStore();

    const created = await store.getState().createTask("  ", "");

    expect(created).toBe(false);

    expect(store.getState().tasks).toHaveLength(0);

    expect(store.getState().lastError).toBe("The title is required.");

  });
```

```
it("a duplicate-title failure exposes a specific message", async () => {

  const store = buildStore();

  await store.getState().createTask("Duplicate", "");

  const created = await store.getState().createTask("Duplicate", "");

  expect(created).toBe(false);

  expect(store.getState().lastError).toBe("A task with this title already exists.");

  expect(store.getState().tasks).toHaveLength(1);

});

it("completing a task reflects the completed state on the list", async () => {

  const store = buildStore();

  await store.getState().createTask("Finish the report", "");

  const taskId = store.getState().tasks[0].id;
```



```
    await store.getState().completeTask(taskId);

    expect(store.getState().tasks[0].completed).toBe(true);

  });

it("reopening a completed task returns it to the open state", async () => {

  const store = buildStore();

  await store.getState().createTask("Finish the report", "");

  const taskId = store.getState().tasks[0].id;

  await store.getState().completeTask(taskId);

  await store.getState().reopenTask(taskId);

  expect(store.getState().tasks[0].completed).toBe(false);

});

it("the initial view is 'open'", () => {
```

```
const store = buildStore();

expect(store.getState().filter).toBe("open");

});

it("setFilter switches the active view without touching the complete list",
  async () => {

    const store = buildStore();

    await store.getState().createTask("Task A", "");

    store.getState().setFilter("completed");

    expect(store.getState().filter).toBe("completed");

    expect(store.getState().tasks).toHaveLength(1);

  });

});
```

The portrait of the lap

This was the cheapest lap in code and the most expensive in thought, and that is no accident: when `implement` finally ran, the hard decisions were already made in the two episodes the artifacts above record. It is the central argument of the method in miniature: the work that seems to get ahead (coding the filter inside the widget right away) is the one that costs dearly later; the work that seems to slow you down (planning where the rule lives) is the one that keeps the architecture standing. In Chapter 9.5, the delete feature brings `tasks` and `analyze` to the center, and with them a conflict between requirements of different features, caught before it turns into code.

9 - Deleting a Task: The Spotlight on `tasks` and `analyze`

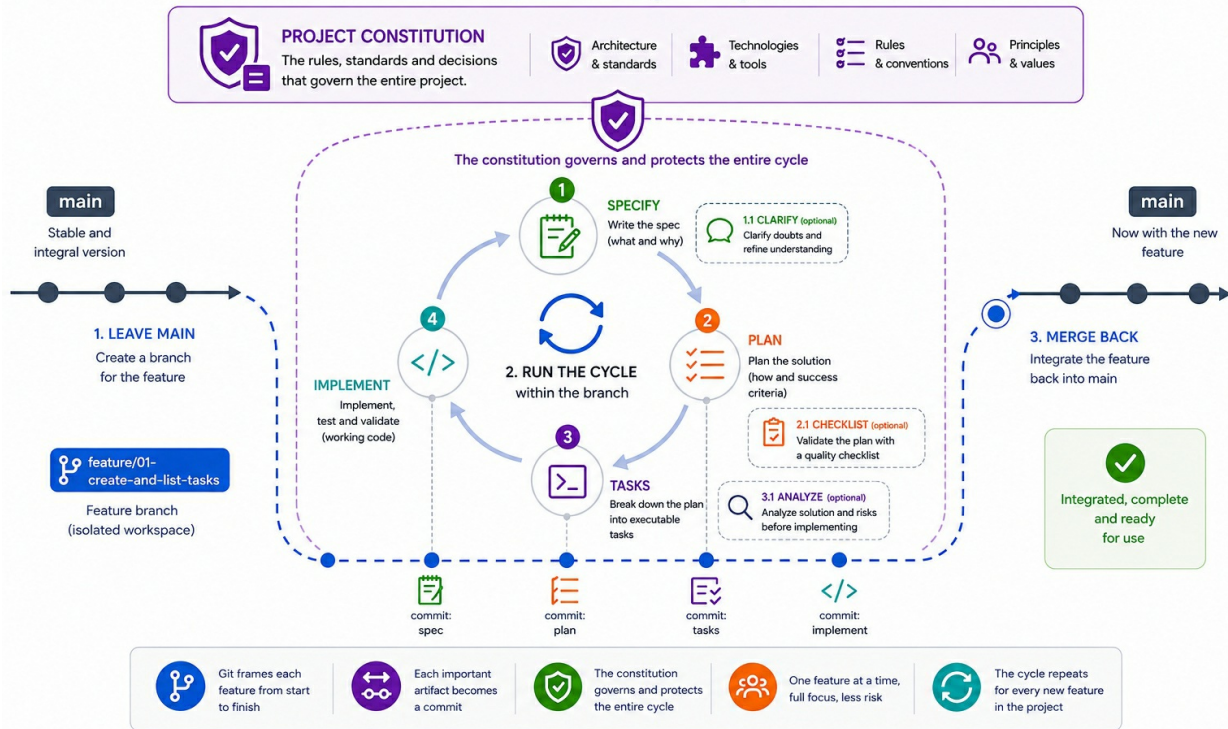
The fourth loop starts: `004-excluir-tarefa`

The `main` branch is stable with three features: create and list, complete and reopen, filter by state. The fourth starts like all the others: you leave `main` instead of touching it. You create the branch `004-excluir-tarefa` and run the whole cycle inside it, until the merge hands the result back to the main line.

The map is the usual one. Open the figure again for a second, just long enough to place yourself on it:

THE COMPLETE SDD CYCLE

One feature, from nothing to integrated code



It is the same diagram from Chapter 5; what changes is only where you stand on it. This time the spotlight falls on **tasks** and **analyze**, the two steps that break the work into steps and check whether the whole still holds together, and that ran in the dark through the three earlier laps.

The feature of the moment is **deleting a task**: the person removes an item they no longer want to see. It is the right showcase for this pair of steps for two reasons. First, deleting is a delicate action, so it matters to break it into small, safe steps, and that is what **tasks** does. Second, deleting talks to what the previous features already decided, and that is where **analyze** takes the lead: it reads spec, plan and tasks together and demands coherence between them. The early steps you already have down, so they will run light so the spotlight can turn on where the chapter wants it.

The early steps, at a light pace

The `specify` of deleting is short, because the intent is small: the person wants to remove a task from the list. The spec describes what happens when they do it and stops before saying how.

`clarify` ties off three loose ends. The first is the most important for the rest of the chapter: is the deletion reversible or final? The clarified answer chooses the simplest path the constitution approves (YAGNI): **final**, with no trash bin nor undo. The task disappears from the list and from storage; if one day reversibility is missed, it becomes its own feature, with its own cycle. The second end is a minimal safeguard: because it is a destructive action with no way back, the interface asks for a simple **confirmation** before removing. Notice that this is a minor decision, resolved in passing in `clarify` and detailed in `plan`, far from the chapter's central subject: just the basic care of not deleting on the first click. The third end closes the scope: you can delete a task in any state, open or completed, and the deletion does not look at that state.

`plan` inherits the usual stack (React with TypeScript, Zustand, `localStorage` behind a repository) and decides the how, which here is lean: the repository gains a `remove` operation, a `DeleteTask` use case orchestrates the removal treating the nonexistent task as error as value, the store gains a `deleteTask` action, and the view gains a button with that confirmation. The `checklist` passes without finding a completeness hole: each view of the deletion is declared, and the check that the artifacts **talk to each other** is left for `analyze`, right ahead.

Between one step and the next, the usual `/clear`: each step restarts by reading the approved artifact, and the chapter reaches `tasks` with a solid base instead of a vague memory.

tasks : the plan turns into a list of steps

tasks reads the plan and turns it into an ordered, verifiable list. It invents no new decision: it takes the construction design and breaks it into the concrete steps that lead to the code. The result is an artifact with the look of an executable checklist, and it is the first place where the work stops being prose and becomes a sequence of actions.

What makes a good task? Two qualities, and they go together. The first: it is **small**, a single step, the size an agent executes at once without getting lost. The second: it is **checkable**, it ends in a state you can look at and say "this is done", instead of "this is more or less on track". A task that touches half the system fails on both: it is too big to run safely and too vague to confirm. Decomposition exists so that each line of the list is a step you can take and check.

This is the real list **tasks** generated for the deletion, grouped by phase:

```
Phase 1 - Repository (data layer)
  T001 test of remove(id) in the repository          (test, fails first)
  T002 declare remove(id) in the repository interface (mechanical: contrac
t)
  T003 implement remove(id) until T001 passes
  T004 remove(id) in the in-memory fake              (mechanical: parity)

Phase 2 - Domain (use case)
  T005 type DeleteTaskFailure = { kind: "task-not-found" } (mechanical: ty
pe)
  T006 test of DeleteTask                            (test, fails first)
  T007 implement makeDeleteTask until T006 passes

Phase 3 - Presentation
  T008 test of deleteTask in the store                (test, fails first)
  T009 deleteTask action in the store until T008 passes
  T010 "delete" button with confirmation in TaskList (mechanical, view)
```

Phase 4 - Validation

T011 npm test (green) and npm run build (clean type-check)

T012 manual validation of the Success Criteria

Look at the size of each item. None says "implement the deletion". Each names a file and a step: a test that fails, a function that makes that test pass, a type, a button. Twelve steps where "delete task" would fit in a single line, and that is the point: the difference between asking an agent (or yourself a week from now) "do the whole thing" and asking "take this step, then confirm, then the next one".

Why this order: TDD and layers

The list is not in just any order; it obeys two forces, and seeing them is what lets you audit a task list instead of just trusting it.

The first force is **TDD**: the test of a behavior comes before the implementation that satisfies it. Notice the T001/T003 pair: the test of `remove` (T001) appears before the implementation of `remove` (T003), with the annotation "fails first". The same pattern repeats in T006/T007 and T008/T009. That order guarantees that each piece of behavior is born from a test that first fails and then passes, instead of a test written to confirm code that already exists.

The second force is **layered architecture**. The list rises from the inside out: first the repository (the data layer, Phase 1), then the domain (the use case, Phase 2), last the presentation (store and view, Phase 3). That direction follows the dependencies, not taste. The `DeleteTask` use case needs the repository to already know `remove`; the store needs the use case to already exist. Building from the bottom up makes each step rest on something already standing. Why the dependency runs that way, and what

happens to a project when it flips, belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it to read this list: the practical rule is enough, that nobody builds the upper floor before the one below it.

With these two forces in mind, you can **audit** a task list. Three questions are enough. Is any step too big, the kind that hides several decisions in a single line? Is any out of order, depending on something not yet built? Did any behavior get an implementation without getting a test first? Passing the deletion list through this sieve approves it: the steps are tiny, they rise in the order of the layers, and every real behavior has its test ahead of it.

A caveat, so the sieve does not turn into dogma. Not every task needs a test of its own. Look at the ones marked "mechanical": declaring the `DeleteTaskFailure` type (T005), exposing `remove` in the repository interface (T002), composing the button in the view (T010). These are steps with no decision logic, covered by the surrounding behavior tests or by the manual validation. Demanding a unit test for each of them would bloat the list without increasing safety. TDD orders tests where there is **behavior** to fix, not where there is only mechanical wiring to connect.

Deep dive: TDD orders by behavior, not by file. The practical rule is not "a test for each file", but "a test before each decision that could be wrong". Removing an item from a list is a decision (and it gets a test); declaring that the repository has a `remove` method is a signature (and it does not). When the list distinguishes the two, it stays lean without opening holes: the points where a bug could fit are

all fenced in by a test that fails first, and the points where no logic fits stay free of ceremony. Confusing the two is what makes suites swell with tests that only confirm that a line of wiring is still that line.

analyze : reading the three artifacts together

With the task list ready, the cycle reaches the last check before the code: **analyze**. It does something no earlier step did: it reads the **three** core artifacts (spec, plan and tasks) at the same time and looks for mismatches between them. Its question stops being about each artifact in isolation and becomes whether they tell the same story.

analyze classifies what it finds into three kinds of finding. A **conflict** is a contradiction: two artifacts state things that do not fit together. A **gap** is a lack: a spec requirement that no task covers, for example. An **excess** is the opposite: a task that matches no requirement, work that showed up unasked. The three answer the same question from different angles: is the chain spec → plan → tasks intact, with no loose link nor extra link?

The result comes as a **consistency report**: a table that lists each finding with its identifier, category, severity, the place where it lives and the recommendation. It is the first time this format appears, and it is deliberately dry, made for you to glance at and know what to halt before going on.

It is worth separating **analyze** from **checklist**, because the two check things and can be confused. The **checklist**, from the previous chapter, looks at **one artifact at a time** and asks whether it is complete and clear: is any decision missing, does any sentence admit two readings? **analyze** looks at the **three together** and asks whether they are consistent with each other:

does what the plan promises match what the spec asked, and do the tasks cover exactly that? Completeness and clarity of each piece are the `checklist`'s job; coherence of the whole is `analyze`'s job. Different questions, different moments.

Deep dive: gap and excess, the other two findings. The conflict is the finding this chapter stages, but the other two show up just as often in practice. A gap is an orphan requirement: the spec says "the deletion asks for confirmation", but no task creates the confirmation step. An excess is an orphan task: the list includes "export tasks to CSV" that no requirement supports, a sign of scope that leaked. `analyze` finds all three the same way, crossing the three sources; what changes is only which side of the correspondence was left without a pair.

A real conflict: the deletion against "completing"

In the filter of the previous chapter, `analyze` passed almost blank, because the clean plan left no mismatch. In the deletion, it catches a real conflict, and not one invented for the occasion: the new feature is on a collision course with a decision made back in Chapter 7.

Remember what the deletion fixed in `clarify`: it is **final**, the task disappears from the list and from storage. Now remember what "completing", in Chapter 7, left written in its spec: a completed task **stays on the list**, in the same position, shown as completed. `analyze` reads the deletion's artifacts together with those of the already integrated features, and the two sentences clash head-on. One says a completed task can disappear for good; the other says a completed task stays on the list. Which of the two holds?

The report logs the clash like this:

ID	Category	Sev.	Location	Summary
C1	Conflict between artifacts	HIGH	delete/spec.md FR-002 × complete/spec.md FR-006	"final deletion: the task disappears from the list" clashes with "a completed task stays on the list". Resolve at the source.

The wrong instinct, faced with this, is to ask "which of the two features is wrong?" and start touching the one that seems weaker. `analyze` pushes toward a better question: **which wording is ambiguous?** And here the diagnosis defuses the conflict. When "completing" said "the task stays on the list", what was it talking about? A specific case: completing a task **does not remove it on its own**, it stays there, struck through, instead of disappearing the instant you mark it. It was a promise against self-removal on completing, not a promise of eternal permanence against any future action. Read that way, the conflict evaporates: deleting can go on being final, because deleting is a deliberate, separate act, not the side effect of completing. The two features always fit together; what was missing was one of the specs saying so in full.

Fix at the source, and the cascade it triggers

Once the conflict is diagnosed, it remains to fix it, and the **where** matters as much as the **what**. The temptation is to patch it downstream: touch a task so the report stops complaining, or

adjust the code at implementation time. `analyze` recommends the opposite: fix it **at the source**, in the artifact where the truth is born.

Conflict C1 is a wording matter in the spec, so it is in the deletion's spec that the correction goes. It gains a new clarification, which distinguishes the two acts at once:

The deletion **MUST** be **final**: the task disappears from the list and from storage. "Final" qualifies the act of **deleting**, distinct from **completing**: completing marks and keeps the task on the list (002 FR-006); deleting removes it for good. The two do not contradict each other.

Notice why the source is the right place. If the fix were in a task, or worse, in a line of code, the contradiction would stay alive in the document that describes the problem: the spec would still say "disappears for good" and the completing one would still say "stays", and the next person to read the two would trip on the same clash, now without `analyze` around to flag it. Fixing at the source is what makes the fix **hold for all the downstream artifacts** at once, instead of covering the symptom at one point and letting it leak at another.

The correction in the spec triggers a **cascade**, and this time it is short. With the spec adjusted, you re-evaluate the artifacts that descend from it: the plan already modeled the deletion as a removal that holds for any state, without depending on the ambiguous sentence, so it stays valid; the tasks already covered deleting a completed task (the `DeleteTask` test in T006 includes exactly that case), so they stay valid. You run `analyze` again, and it passes clean: zero conflicts. The cascade was short precisely because the problem was in the **wording**, not in a construction decision. If the conflict were about a decision (if the plan had

chosen, say, "mark as deleted" instead of removing), the cascade would descend deeper, realigning plan, tasks and maybe tests.

`analyze` does not decide the size of the fix; it only guarantees that the fix goes down the whole chain, instead of stopping at the first file.

That is `analyze`'s role in the cycle: the last cheap safety net before `implement`. Everything it catches now costs a rewritten sentence; the same conflict, discovered with the code done and two features fighting in production, would cost a whole investigation.

From the branch to the merge, and the bridge to Ch. 10

With the report clean, `implement` executes the list in following mode. There is no surprise here, and that is the point: `tasks` made the steps so tiny and `analyze` made the whole so coherent that implementing is going down the list. The `remove` test fails and then passes; the `DeleteTask` use case is born from its test; the store gains `deleteTask`; the view gains the button with the confirmation. In the end, `npm test` closes in the green, with the new deletion tests added to those of the previous features, and `npm run build` confirms the clean type-check. The feature that began as a sentence now runs.

From there the path is the known one. You commit the core artifacts, one by one, and **merge** the `004-excluire-tarefa` branch back into `main`. The main line is stable again, now with four capabilities where there were three. `tasks` broke the work down; `analyze` caught the clash with "completing" while it was still text.

The focus moves once more, and this time to the last lap. The fifth and final feature in the backlog is **editing a task**, changing the title or the description of something that already exists.

There the specification and planning work is familiar, and the weight falls on making the change happen end to end without breaking what already works. That is why, in Chapter 10, the spotlight shifts to `implement`, the step that turns the task list into code that passes the tests, and that closes the app.

9.5 - Deleting a Task in Practice: the Whole Loop, File by File

How to Read This Chapter

Chapter 9 followed the fourth lap of the cycle with the spotlight on `tasks` and `analyze`. Here you see the whole lap, exactly as it was recorded in the example app's repository, in the `004-excluir-tarefa` feature.

This is the conflict lap. `analyze` cross-checked the deletion spec against the spec for the completion feature (002, already merged) and found a contradiction between artifacts: deletion said the task "vanishes for good," while completion promised that "a completed task stays in the list." Read together, they seemed to contradict each other. `analyze` fixed nothing on its own; it produced a report, pointed to the right place for the fix (the deletion spec, not the tasks) and demanded a second clean run before releasing `implement`. That report, with both runs, is the central artifact of this chapter, and you read it in full below.

As in the earlier laps, the artifacts appear as markdown (with headings demoted) and the code in blocks with a sentence of context before each one. Several files are modifications of files you have already seen; when that is the case, the text points out what changed.

The Input: the `/speckit-specify` Prompt

"Delete a task from the personal to-do list. The person removes a task they no longer want to see. Out of scope for this feature: bulk deletion, a trash bin, and multi-undo."

What `specify` and `clarify` Returned: `spec.md`

Notice the fourth clarification (Q4): it is born already answering the conflict that `analyze` would later formalize. The spec below is the **corrected** version, after the second run of `analyze`; FR-002 and Scenario 2 carry the caveat that tells deleting apart from completing.

Feature Specification: Delete a task

Feature Branch: `004-excluir-tarefa`

Created: 2026-06-27

Status: Draft

Input: User description: "Delete a task from the personal to-do list. The person removes a task they no longer want to see. Out of scope for this feature: bulk deletion, a trash bin, and multi-undo."

Clarifications

Session 2026-06-27

- Q: Is deletion reversible (trash/undo) or permanent? → A:

Permanent. The simplest option that respects the constitution (YAGNI): the task disappears from the list and from storage, with no trash bin and no undo. If reversibility is ever wanted, it becomes its own feature.

- Q: Does the person confirm before deleting, or is deletion immediate? → A: Confirms. Since it is a destructive action with no undo, the interface asks for a simple confirmation before removing; it is the only safeguard, decided here and without turning into an elaborate mechanism.
- Q: Can a task be deleted in any state (open or completed)? → A: Yes. Deletion does not look at state: it removes the indicated task, whether it is open or completed.
- Q: Doesn't "permanent" conflict with 002 (FR-006: "a completed task stays in the list")? → A: No. They are distinct actions. **Completing** only marks the task and keeps it visible, in place; that permanence is what 002's FR-006 speaks of (completing does not auto-remove). **Deleting** is a separate, deliberate, confirmed action that takes the task away for good. The permanence promised by 002 holds for the act of completing; it does not prevent the act of deleting.

User Scenarios & Testing (mandatory)

User Story 1 - Delete a task (Priority: P1)

The person has a task in the list they no longer want to see: it was created by mistake, it lost its point, or it is already done and now just takes up space. They trigger the deletion of that task, confirm, and the task disappears from the list for good.

Why this priority: It is the entire reason for the feature. Without it, the list only grows; the person needs a way to take out for good what no longer belongs there.

Independent Test: Can be tested by creating a task, deleting it, and checking that it no longer appears in the list and does not come back when the app is reopened.

Acceptance Scenarios:

1. **Given** a list with a few tasks, **When** the person deletes one of them and confirms, **Then** the task disappears from the list and does not return when the app is reopened.
2. **Given** a **completed** task that no longer matters, **When** the person deletes it, **Then** it is permanently removed from the list, like any other. Completing would keep it visible (002 FR-006); deleting is the deliberate act that takes it out.

Edge Cases

- Deleting a task that no longer exists (already removed in another tab, for example) is handled as a predictable error (error as value), not as an exception that crashes the application.
- Deletion is **permanent**: there is no trash bin and no undo. Once confirmed, the task disappears from the list and from storage.
- Canceling the confirmation does **not** delete: the task stays intact in the list.

Requirements (mandatory)

Functional Requirements

- **FR-001:** The system **MUST** allow deleting an existing task from the list.
- **FR-002:** Deletion **MUST** be **permanent**: the task disappears from the list and from storage, with no trash bin and no undo. "Permanent" qualifies the action of **deleting**, distinct from

completing (002): completing marks and keeps the task in the list (002 FR-006); deleting takes it out for good. The two do not contradict each other.

- **FR-003:** The system **MUST** ask for a simple confirmation before deleting; canceling the confirmation preserves the task.
- **FR-004:** The system **MUST** handle the deletion of a nonexistent task as a predictable error (error as value), without throwing an exception across layers.
- **FR-005:** Deleting a task **MUST NOT** change the position of the others in the list (creation order is preserved).

Non-Goals

- Explicitly out of scope: **bulk deletion, a trash bin, undo (multi or single), and archiving (YAGNI).**

Key Entities

- **Task:** reused from 001/002 with no field changes. Deletion removes the whole task; it adds no new field (no "deleted" as a state, since there is no trash bin).

Success Criteria (mandatory)

- **SC-001:** The person can delete a task and see it disappear from the list right away.
- **SC-002:** After closing and reopening the app, the deleted task remains absent (the deletion persisted).
- **SC-003:** Canceling the confirmation keeps the task in the list, without removing it.

Assumptions

- The feature extends those for creating/listing (001),

completing/reopening (002) and filtering (003), already integrated on `main`; it reuses the Task entity, the repository and the existing layered architecture.

- The To-Do constitution governs the feature: Simplicity/YAGNI, layered architecture, business logic isolated from the UI, error as value and TDD.
 - Deletion is permanent (clarification): the simplest option that respects the constitution. The confirmation before deleting is the only safeguard, without turning into an elaborate undo mechanism.
-

The Completeness Check:

`checklists/requirements.md`

Before `analyze`, `checklist` checks the spec on its own. Notice what it records in the notes: this time it found no completeness gaps, and it says explicitly that consistency **between artifacts** is `analyze`'s job, not its own. The two instruments have different targets, and this chapter shows both.

Specification Quality Checklist: Delete a task

Created: 2026-06-27

Feature: spec.md

Content Quality

- No implementation details (languages, frameworks, APIs)
- Focused on user value and business needs

- Written for non-technical stakeholders
- All mandatory sections completed

Requirement Completeness

- No [NEEDS CLARIFICATION] markers remain
- Requirements are testable and unambiguous
- Success criteria are measurable
- Success criteria are technology-agnostic
- All acceptance scenarios are defined
- Edge cases are identified
- Scope is clearly bounded
- Dependencies and assumptions identified

Feature Readiness

- All functional requirements have clear acceptance criteria
- User scenarios cover primary flows
- Feature meets measurable outcomes defined in Success Criteria
- No implementation details leak into specification

Notes

- `checklist` found no completeness gaps this time: the confirmation and the permanence were already pinned down in `clarify`. Consistency **between artifacts** (this spec against the completion spec) is `analyze`'s job, not `checklist`'s.
-

What `plan` Decided: `plan.md`

This lap's plan passes the constitution check clean, with no rejected draft. The operation is simple (a `remove` method on the repository, a `DeleteTask` use case), and the most interesting design decision is where the confirmation lives: in the view, because it is a UX safeguard, not a business rule.

Implementation Plan: Delete a task

Branch: `004-excluir-tarefa` | Spec: `spec.md`

Input: Feature specification in `specs/004-excluir-tarefa/spec.md`

Summary

Remove a task from the list **permanently**. The spec says **what** (the task disappears for good, after confirmation); this plan decides **how**: a new `remove(id)` method on the repository, a `DeleteTask` use case in the domain that returns `Result` (error as value for the nonexistent task), the orchestration in the store and a simple confirmation in the view before triggering the action. No new field enters the task: deleting erases the whole task, it does not mark a state.

Technical Context

Language/Version: TypeScript 5.x on Node 20+; target: browser (React web). **Inherited from 001/002/003.**

Main dependencies: React 18, Vite, Zustand. **No new dependencies** (stack inherited from the 001 `plan`).

Persistence: the deletion is written to `localStorage` behind `TaskRepository` (real removal of the item). It is the counterpart of `add`: what was persisted ceases to exist.

Testing: Vitest for the `DeleteTask` use case (domain, no DOM), for the in-memory repository's `remove` and for the store action.

Project type: single-page, single-user web application, no backend.

How-Decisions (what the spec did not pin down)

1. **Where data removal lives:** a new `remove(id: string)` method on `TaskRepository`, implemented in `LocalStorageTaskRepository` (and in the in-memory fake for the tests). It is the counterpart of `add`; the domain does not know `localStorage`.
2. **Who decides and reports the failure:** a `DeleteTask` use case in the domain. It looks the task up by id; if it does not exist, it returns `failure({ kind: "task-not-found" })` (error as value, FR-004); if it exists, it calls `remove` and returns success. The view never talks straight to the repository.
3. **Where the confirmation lives:** in the view. The confirmation is an interface gesture (a UX safeguard), not a business rule; the domain receives the delete order already confirmed. The store only orchestrates: it triggers `deleteTask` and reloads the list.

Constitution Check

Principle	How the plan meets it
I. Layered Architecture	Data removal sits behind the repository; the rule (does it exist? then remove) sits in the domain use case; the view only confirms and displays. Dependencies

	point toward the domain.
II. Business Logic Isolated	The decision "delete an existing task; nonexistent is a failure" is a pure use case, testable without a UI. The confirmation is UX, not a business decision.
III. Error as Value	A nonexistent task becomes <code>Result.failure({ kind: "task-not-found" })</code> , symmetric to <code>CompleteTask</code> ; no exception crosses layers.
IV. TDD	Tests of the use case (removes an existing one; nonexistent becomes a failure; does not affect the others) and of the repository's <code>remove</code> before implementation.
V. Simplicity/YAGNI	No trash bin, no undo, no bulk deletion, no new state on the task. One <code>remove</code> method, one use case, one confirmation.
VI. Technology Agnosticism	The spec names no stack; the stack (React/TS/Vite/Zustand) is inherited. The decisions here are about design (where removal lives, who reports the failure), not new technology.

Result: PASS. No violations; no rejected draft in this feature.

Traceability clarify/spec → plan

Decision (clarify/spec)	Effect on the plan
Permanent deletion (no	<code>remove</code> erases the item from <code>localStorage</code> ; no "deleted" state is stored.

trash/undo)	
Confirmation before deleting	Confirmation in the view; the domain receives the order already confirmed.
Deleting holds for any state	<code>DeleteTask</code> does not read <code>completed</code> : it removes the indicated task, open or completed.
Nonexistent is error as value	<code>DeleteTaskFailure = { kind: "task-not-found" } ,</code> handled in the store.

Project Structure (additions)

```

src/
├── domain/
│   ├── failures/delete-task-failure.ts    # new: { kind: "task-not-found" }
│   ├── repositories/task-repository.ts    # + remove(id): Promise<void>
│   └── usecases/delete-task.ts            # new: makeDeleteTask(repository
└── )
├── data/
│   └── local-storage-task-repository.ts    # + remove(id) (counterpart of a
└── dd)
├── presentation/
│   ├── store/task-store.ts                # + deleteTask(id)
│   └── components/TaskList.tsx            # + delete button with confirmat
└── ion
test/
├── domain/delete-task.test.ts              # new: remove; nonexistent become
├── s a failure; preserves the others
├── data/local-storage-task-repository.test.ts # + remove persists the absen
├── ce
└── presentation/task-store.test.ts          # + deleteTask reloads the list

```

Phase 1 - Design

- Model of the operation in data-model.md.

- Manual validation in quickstart.md.

The `analyze` Report: `analyze-report.md`

This is the central artifact of the chapter. Read it whole: it has two runs, the first catching the HIGH conflict (C1) and ordering a fix at the source, the second passing clean and releasing `implement`.

Consistency report (`analyze`): Delete a task

Cross-scan of `spec.md` × `plan.md` × `tasks.md` for the `004-excluir-tarefa` feature, read together with the artifacts of the features already integrated on `main` (especially `002-concluir-tarefa`). `analyze` rewrites nothing: it produces the report below and recommends where to fix.

First run (before the fix)

Specification Analysis Report

ID	Category	Severity	Location	Summary
C1	Inconsistency (conflict between artifacts)	HIGH	<code>004-excluir-tarefa/spec.md</code> FR-002, Scenario 2, Clarif. Q3 × <code>002-concluir-tarefa/spec.md</code> FR-006	The deletion spec states that a task (including a completed one) "disappears from the list and from

				storage, permanently" the completion spec states that "a completed task stays in the list, in the same position." Read together the two contradict each other on whether an item can leave the list.
A1	Underspecification (minor)	LOW	004-excluir-tarefa/spec.md FR-003	"Simple confirmation' does not pin down the means (native dialog, modal,

Coverage Summary

- Total tasks: 12
- Requirement coverage (≥ 1 task): 100% (FR-001..FR-005 mapped to T001-T010)
- Ambiguity count: 1 (LOW)
- Duplication count: 0

- Conflicts: 1 (HIGH)

Diagnosis of conflict C1

The point is not "which spec is right and which is wrong all the way through," but **which wording is ambiguous**. Checked against the real artifact `002-concluir-tarefa/spec.md` (FR-006, a finished, merged feature), completion's "stays in the list / nothing disappears" referred to a specific thing: **completing does not auto-remove** the task (it stays struck through, in place), as opposed to the idea that finishing a to-do would erase it. It was **not** a promise of eternal permanence against any action. So deletion can stay **permanent** without contradicting completion.

The artifact to adjust is the **deletion spec**: the current wording leaves "permanent" loose, without saying it describes the action of deleting (distinct from completing). That is where, at the source, the clarification goes in.

Next Actions

- HIGH conflict present: **resolve before** `/speckit-implement`.
- Suggested action: edit `004-excluir-tarefa/spec.md` (source), realign `plan.md` / `tasks.md` if needed and **rerun** `analyze` until it passes clean.
- Do **not** change a task so the report stops flagging the conflict: that would hide the contradiction instead of resolving it.

The fix cascade

The fix went in at the **source**, the deletion spec (`004-excluir-tarefa/spec.md`):

- A new clarification Q4 tells completing (marks and keeps) apart from deleting (takes it out for good).

- FR-002 now says that "permanent" qualifies the action of deleting, distinct from completing.
- Scenario 2 gains the same caveat.

From there, a cascade check of the downstream artifacts:

- `plan.md` : it already modeled deletion as `DeleteTask` removing in any state; no decision depended on the ambiguous wording. It stays valid, unchanged.
- `tasks.md` : the 12 tasks already covered deleting a completed task (T006 tests that case). Unchanged.

Since the contradiction was in the spec's **wording**, not in a technical decision, the cascade was short: with the source fixed, plan and tasks stayed coherent. If the conflict were about a decision (say, the plan having chosen "mark as deleted" instead of removing), the cascade would run down to the tasks and the code.

Second run (after the fix)

Specification Analysis Report

ID	Category	Severity	Location	Summary	Recommendation
—	—	—	—	No inconsistency, relevant ambiguity or conflict between artifacts.	No action.

Coverage Summary

- Total tasks: 12
- Requirement coverage (≥ 1 task): 100%
- Ambiguity count: 0
- Duplication count: 0
- Conflicts: 0

Next Actions

- Clean report: cleared for `/speckit-implement`.

The `plan`'s Model of the Operation: `data-model.md`

Deletion adds neither entity nor field; it models an operation on the Task that already exists. The table makes explicit that the error as value lives in the use case, and that storage idempotency (removing an absent id does not break) lives in the repository.

Data Model: Delete a task

The feature adds neither entity nor field. It models an operation on the existing `Task` entity: permanent removal.

Operation

Operation	Input	Output	Predictable failure
<code>DeleteTask</code>	<code>id:</code> <code>string</code>	<code>Result<void,</code> <code>DeleteTaskFailure></code>	<code>task-not-found</code> (no task with that id exists)

Counterpart in the repository

Method (<code>TaskRepository</code> port)	Effect
<code>remove(id: string): Promise<void></code>	Erases the task from the storage medium; counterpart of <code>add</code> . Idempotent from the storage point of view (removing an absent id does not break), but the existence check and the error as value live in the use case.

Preserved invariants

- The order of the other tasks (creation, oldest first) does not change when one is removed (FR-005).
- Deleting does not depend on the task's state: open or completed are removed alike.
- No "deleted" state is persisted: the task disappears, it is not marked (no trash bin).

The Manual Validation: `quickstart.md`

Quickstart / Validation: Delete a task

Automated tests

```
npm test      # all green, including delete-task, repository.remove and sto
re.deleteTask
```

```
npm run build # clean type-check
```


Manual validation (Success Criteria)

1. **SC-001**: create a task, trigger delete, confirm → the task disappears from the list right away.
 2. **SC-002**: delete a task, close and reopen the app → the task stays absent.
 3. **SC-003**: trigger delete and **cancel** the confirmation → the task stays in the list.
 4. Delete a **completed** task → it disappears just like an open one (FR-002, clarification).
 5. The other tasks keep their creation order after the removal (FR-005).
-

The `tasks` Execution List: `tasks.md`

Notice the opening note: purely mechanical steps (creating a failure type, composing a button in the view) **do not require a test of their own**. It is the granularity Chapter 9 discussed: `tasks` tells apart what needs a test first from what is just wiring.

Tasks: Delete a task

Branch: `004-excluir-tarefa` | **Plan:** `plan.md`

TDD order: the test for a behavior comes before the implementation that satisfies it. `[P]` marks tasks that can run in parallel (distinct files, no dependency). Purely mechanical steps (creating a failure type, composing a button in the view) do not require a test of their own.

Phase 1 - Repository (data layer)

- **T001** `test/data/local-storage-task-repository.test.ts` : test of `remove(id)` (removing a task erases only that one from `localStorage` ; removing an absent id does not break; the order of the others is preserved). **Fails first**.
- **T002** `src/domain/repositories/task-repository.ts` : declare `remove(id: string): Promise<void>` in the repository interface. (*mechanical step: contract, no test of its own*)
- **T003** `src/data/local-storage-task-repository.ts` : implement `remove(id)` (counterpart of `add`) until T001 passes.
- **T004** `test/helpers/in-memory-task-repository.ts` : implement `remove(id)` in the in-memory fake, for the domain tests. (*mechanical step: fake parity with the repository*)

Phase 2 - Domain (use case)

- **T005** `src/domain/failures/delete-task-failure.ts` : type `DeleteTaskFailure = { kind: "task-not-found" }` . (*mechanical step: failure type, no test of its own*)
- **T006** `test/domain/delete-task.test.ts` : tests of `DeleteTask` (removes an existing task and returns success; removes a **completed** task just like an open one; a nonexistent task returns `task-not-found` ; removing one does not change the position of the others). **Fails first**.
- **T007** `src/domain/usecases/delete-task.ts` : `makeDeleteTask(repository)` (looks up by id; nonexistent becomes a failure; existing calls `remove` and returns success); implement until T006 passes.

Phase 3 - Presentation

- **T008** `test/presentation/task-store.test.ts` : test of `deleteTask` in the store (after deleting, the reloaded list no longer contains the

task). **Fails first.**

- **T009** `src/presentation/store/task-store.ts` : `deleteTask(id)` action (triggers the use case and reloads `tasks`); implement until T008 passes.
- **T010** `src/presentation/components/TaskList.tsx` : a "delete" button on each item, with **confirmation** before triggering `deleteTask` (canceling does not delete). (*mechanical view step: composition, covered by the manual validation*)

Phase 4 - Validation

- **T011** `npm test` (all green) and `npm run build` (clean type-check).
 - **T012** Manual validation of Success Criteria SC-001..SC-003 and of the quickstart.
-

The Code `implement` Generated

Since `analyze` passed clean on the second run, `implement` ran without surprises. The design was already coherent; the code is the expected wiring. Where a file already appeared in earlier chapters, the text points out what changed.

Domain

The deletion's predictable failure, symmetric to completion's. The comment recalls that deletion is permanent, with no intermediate state.

```
// src/domain/failures/delete-task-failure.ts
```

```
/**
```

```

* Predictable failure when deleting a task
.

*

* `task-not-found`: no task with the given id exists (for example, already
removed in another tab). It is error as value, not an exception (constitution,
Principle III). Deletion is permanent: there is no intermediate state
.

*
/

export type DeleteTaskFailure = { readonly kind: "task-not-found" };

```

The repository gains the `remove` method, the counterpart of `add`. The contract grows; the real repository and the test fake need to follow.

```

// src/domain/repositories/task-repository.ts

import type { Task } from "../../entities/task";

/**

```

```
* Domain repository for persisting and reading tasks
.

*

* The domain declares this interface; the data layer implements it. This way
y

* the business rule does not know the storage medium
.

*
/
```

```
export interface TaskRepository {
```

```
  /** All tasks, in creation order (oldest first). */
```

```
  getAll(): Promise<Task[]>;
```

```
  /** A task by id, or `null` if it does not exist. */
```

```
  getById(id: string): Promise<Task | null>;
```

```
  /** Persists a new task. */
```

```
  add(task: Task): Promise<void>;
```

```

    /** Updates an existing task in place, without changing the list order. */

    update(task: Task): Promise<void>;

    /** Removes a task from storage. Counterpart of `add`. */

    remove(id: string): Promise<void>;

}

```

The `DeleteTask` use case: it looks up by id, returns `task-not-found` if it does not find one, removes it and returns success if it does. It does not read `completed`, because deleting holds for any state. It is the same shape as the use cases of the earlier laps.

```

// src/domain/usecases/delete-task.ts

import type { DeleteTaskFailure } from "../failures/delete-task-failure";

import type { TaskRepository } from "../repositories/task-repository";

import { failure, success, type Result } from "../result";

export type DeleteTask = (id: string) => Promise<Result<void, DeleteTaskFailure>>>;

```

```

/**

* Deletes a task permanently (FR-001, FR-002)
.

*

* Deleting does not look at the task's state: it removes the indicated one
,

* whether open or completed (clarification 2026-06-27). A nonexistent task i
s

* error as value (`task-not-found`, FR-004), not an exception. Removing a ta
sk

* does not reorder the others (FR-005): the repository only takes the item o
ut

* of the list
.

*
/

```

```

export const makeDeleteTask = (repository: TaskRepository): DeleteTask =>

```

```

  async (id) => {

```

```

    const task = await repository.getById(id);

```

```
if (task === null) {  
  
    return failure({ kind: "task-not-found" });  
  
}  
  
await repository.remove(id);  
  
return success(undefined);  
  
};
```

Data

The `localStorage` repository gains `remove` : it reads everything, filters out the given id and rewrites. The order of the others is preserved because the filter does not reorder.

```
// src/data/local-storage-task-repository.ts  
  
import type { Task } from "../../domain/entities/task";  
  
import type { TaskRepository } from "../../domain/repositories/task-repository";  
  
const STORAGE_KEY = "todo-app.tasks";
```



```
/**  
  
 * Persists tasks as JSON in the browser's `localStorage`. It is the we  
b  
  
 * equivalent of "persist on the device" (FR-006): it survives closing an  
d  
  
 * reopening the app. I/O exceptions stay contained at this boundar  
y  
  
 * (constitution, Principle III); the domain never sees `localStorage`  
.  
  
 *  
/  

```

```
export class LocalStorageTaskRepository implements TaskRepository {  
  
    constructor(private readonly storage: Storage = localStorage) {}  
  
    async getAll(): Promise<Task[]> {  
  
        const raw = this.storage.getItem(STORAGE_KEY);  
  
        if (raw === null || raw.trim().length === 0) {  
  
            return [];  

```

```

    }

    const stored = JSON.parse(raw) as Array<Omit<Task, "completed"> & { completed?: boolean }>;

    // Tasks written by the previous feature have no `completed`; reading the
    // missing one as `false` (open) keeps compatibility without a migration.

    const tasks: Task[] = stored.map((task) => ({ ...task, completed: task.completed ?? false }));

    return tasks.sort((a, b) => a.createdAt.localeCompare(b.createdAt));

}

async getById(id: string): Promise<Task | null> {

    const tasks = await this.getAll();

    return tasks.find((task) => task.id === id) ?? null;

}

```

```
async add(task: Task): Promise<void> {  
  
    const tasks = await this.getAll();  
  
    const updated = [...tasks, task];  
  
    this.storage.setItem(STORAGE_KEY, JSON.stringify(updated));  
  
}
```

```
async update(task: Task): Promise<void> {  
  
    const tasks = await this.getAll();  
  
    const updated = tasks.map((existing) => (existing.id === task.id ? task :  
existing));  
  
    this.storage.setItem(STORAGE_KEY, JSON.stringify(updated));  
  
}
```

```
async remove(id: string): Promise<void> {  
  
    const tasks = await this.getAll();
```

```

    const remaining = tasks.filter((task) => task.id !== id);

    this.storage.setItem(STORAGE_KEY, JSON.stringify(remaining));

  }

}

```

Presentation

The store gains the `deleteTask` action, in the same mold as `completeTask` / `reopenTask`: it triggers the use case and reloads the list. Nothing new in the structure.

```

// src/presentation/store/task-store.ts

import { createStore } from "zustand/vanilla";

import type { Task } from "../../domain/entities/task";

import type { TaskFilter } from "../../domain/entities/task-filter";

import type { CreateTaskFailure } from "../../domain/failures/create-task-failure";

import type { CompleteTask } from "../../domain/usecases/complete-task";

import type { CreateTask } from "../../domain/usecases/create-task";

```

```
import type { DeleteTask } from "../../domain/usecases/delete-task";

import type { ListTasks } from "../../domain/usecases/list-tasks";

import type { ReopenTask } from "../../domain/usecases/reopen-task";


export type TaskStoreState = {

  readonly tasks: Task[];

  readonly filter: TaskFilter;

  readonly lastError: string | null;

  loadTasks: () => Promise<void>;

  createTask: (title: string, description: string) => Promise<boolean>;

  completeTask: (id: string) => Promise<void>;

  reopenTask: (id: string) => Promise<void>;

  deleteTask: (id: string) => Promise<void>;

  setFilter: (filter: TaskFilter) => void;

};
```

```

const messageFor = (failure: CreateTaskFailure): string => {

  switch (failure.kind) {

    case "title-required":

      return "The title is required.";

    case "duplicate-title":

      return "A task with this title already exists.";

  }

};

/**

 * Presentation store (Zustand) that orchestrates the domain use cases and
 * exposes the state to the UI. It holds no business rule: it only triggers
 * `createTask`/`listTasks` and translates the typed failure into a message for
 * the screen
 */

```

```
*  
/
```

```
export const createStore = (
```

```
  createTask: CreateTask,
```

```
  listTasks: ListTasks,
```

```
  completeTask: CompleteTask,
```

```
  reopenTask: ReopenTask,
```

```
  deleteTask: DeleteTask
```

```
) =>
```

```
  createStore<TaskStoreState>((set) => ({
```

```
    tasks: [],
```

```
    filter: "open",
```

```
    lastError: null,
```

```
    loadTasks: async () => {
```

```
      const tasks = await listTasks();
```

```
    set({ tasks, lastError: null });

},

createTask: async (title, description) => {

    const result = await createTask({ title, description });

    if (result.kind === "failure") {

        set({ lastError: messageFor(result.error) });

        return false;

    }

    const tasks = await listTasks();

    set({ tasks, lastError: null });

    return true;

},
```



```
completeTask: async (id) => {

  await completeTask(id);

  const tasks = await listTasks();

  set({ tasks });

},

reopenTask: async (id) => {

  await reopenTask(id);

  const tasks = await listTasks();

  set({ tasks });

},

deleteTask: async (id) => {

  await deleteTask(id);

  const tasks = await listTasks();
```

```

        set({ tasks });

    },

    setFilter: (filter) => {

        set({ filter });

    }

    ));

```

The `TaskList` gains the "delete" button and the confirmation, the latter via `window.confirm` in the view itself, exactly where the plan placed it (UX safeguard, not business rule). Canceling does not delete, because `deleteTask` only fires inside the `if`.

```

// src/presentation/components/TaskList.tsx

import { selectByFilter } from "../../domain/entities/task-filter";

import { useTaskStore } from "../../store/task-store-context";

/**

 * Lists the active view's tasks in creation order; an empty list is a valid

```

```
* state. The choice of which tasks appear comes from the domain
n

* (`selectByFilter`); here we only display the result. A completed task is
s

* shown struck through, with the action toggling between completing and
d

* reopening
.

*
/
```

```
export function TaskList() {
```

```
  const tasks = useTaskStore((state) => state.tasks);
```

```
  const filter = useTaskStore((state) => state.filter);
```

```
  const completeTask = useTaskStore((state) => state.completeTask);
```

```
  const reopenTask = useTaskStore((state) => state.reopenTask);
```

```
  const deleteTask = useTaskStore((state) => state.deleteTask);
```

```
  const visibleTasks = selectByFilter(tasks, filter);
```

```

const confirmAndDelete = (id: string, title: string) => {

    if (window.confirm(`Delete "${title}"? This action cannot be undone.`)) {

        void deleteTask(id);

    }

};

if (visibleTasks.length === 0) {

    return <p className="task-list__empty">No tasks in this view.</p>;

}

return (

    <ul className="task-list">

        {visibleTasks.map((task) => (

            <li

                key={task.id}

                className={task.completed ? "task-list__item task-list__item--done"
: "task-list__item"}

```

```

    >

    <label className="task-list__check">

        <input

            type="checkbox"

            checked={task.completed}

            aria-label={task.completed ? `Reopen ${task.title}` : `Complete
${task.title}`}

            onChange={() => (task.completed ? reopenTask(task.id) : complet
eTask(task.id))}

        />

        <span>

            <h3>{task.title}</h3>

            {task.description.length > 0 && <p>{task.description}</p>}

        </span>

    </label>

    <button

        type="button"

```

```

        className="task-list__delete"

        aria-label={`Delete ${task.title}`}

        onClick={() => confirmAndDelete(task.id, task.title)}

      >

        Delete

    </button>

  </li>

  )})

</ul>

);

}

```

The `main.tsx` composes the new use case at the boundary and injects it into the store, as in the earlier laps, with one more line.

```

// src/main.tsx

import { StrictMode } from "react";

```

```
import { createRoot } from "react-dom/client";

import { LocalStorageTaskRepository } from "../data/local-storage-task-repository";

import { makeCompleteTask } from "../domain/usecases/complete-task";

import { makeCreateTask } from "../domain/usecases/create-task";

import { makeDeleteTask } from "../domain/usecases/delete-task";

import { makeListTasks } from "../domain/usecases/list-tasks";

import { makeReopenTask } from "../domain/usecases/reopen-task";

import { App } from "../presentation/App";

import { createTaskStore } from "../presentation/store/task-store";

import { TaskStoreProvider } from "../presentation/store/task-store-context";

import "../index.css";

// Composition at the boundary: the concrete repository is instantiated here
// and

// injected into the use cases; only at this point do the layers meet.

const repository = new LocalStorageTaskRepository();
```

```
const store = createStore(  
  
  makeCreateTask(repository),  
  
  makeListTasks(repository),  
  
  makeCompleteTask(repository),  
  
  makeReopenTask(repository),  
  
  makeDeleteTask(repository)  
  
);  
  
const rootElement = document.getElementById("root");  
  
if (rootElement === null) {  
  
  throw new Error("Element #root not found in index.html.");  
  
}  
  
createRoot(rootElement).render(  
  
  <StrictMode>
```



```
<TaskStoreProvider store={store}>

  <App />

</TaskStoreProvider>

</StrictMode>

);
```

Tests

The `DeleteTask` tests cover the four cases `tasks` asked for, including the one that proves deleting a **completed** task works just like an open one (the case born from the conflict `analyze` caught).

```
// test/domain/delete-task.test.ts

import { describe, expect, it } from "vitest";

import { makeCompleteTask } from "../../src/domain/usecases/complete-task";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeDeleteTask } from "../../src/domain/usecases/delete-task";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository"
;
```

```
const seedTask = async (repository: InMemoryTaskRepository, id: string, title
: string) => {

    const createTask = makeCreateTask(repository, () => id);

    const result = await createTask({ title });

    if (result.kind === "failure") {

        throw new Error("seed failed");

    }

    return result.value;

};

describe("DeleteTask", () => {

    it("removes an existing task and returns success", async () => {

        const repository = new InMemoryTaskRepository();

        const task = await seedTask(repository, "id-1", "Wash the dishes");

        const deleteTask = makeDeleteTask(repository);
```

```
const result = await deleteTask(task.id);

expect(result.kind).toBe("success");

expect(await repository.getById(task.id)).toBeNull();

});

it("removes a completed task just like an open one", async () => {

  const repository = new InMemoryTaskRepository();

  const task = await seedTask(repository, "id-1", "Done task");

  await makeCompleteTask(repository)(task.id);

  const deleteTask = makeDeleteTask(repository);

  const result = await deleteTask(task.id);

  expect(result.kind).toBe("success");

  expect(await repository.getById(task.id)).toBeNull();
```

```

});

it("returns task-not-found when the task does not exist", async () => {

  const repository = new InMemoryTaskRepository();

  const deleteTask = makeDeleteTask(repository);

  const result = await deleteTask("nonexistent");

  expect(result).toEqual({ kind: "failure", error: { kind: "task-not-found"
} });

});

it("deleting a task does not change the position of the others", async () => {

  const repository = new InMemoryTaskRepository();

  let tick = 0;

  const createTask = makeCreateTask(repository, () => `id-${tick}`, () => new Date(2026, 0, 1, 0, 0, tick++));

  await createTask({ title: "First" });

```

```

    await createTask({ title: "Second" });

    await createTask({ title: "Third" });

    const deleteTask = makeDeleteTask(repository);

    await deleteTask("id-1");

    const titles = (await repository.getAll()).map((task) => task.title);

    expect(titles).toEqual(["First", "Third"]);

  });

});

```

The in-memory fake gains `remove`, mirroring the real repository for the domain tests.

```

// test/helpers/in-memory-task-repository.ts

import type { Task } from "../../src/domain/entities/task";

import type { TaskRepository } from "../../src/domain/repositories/task-repository";

```

```
/** In-memory implementation of `TaskRepository` for domain and store tests.
 */

export class InMemoryTaskRepository implements TaskRepository {

  private readonly tasks: Task[] = [];

  async getAll(): Promise<Task[]> {

    return [...this.tasks].sort((a, b) => a.createdAt.localeCompare(b.createdAt));

  }

  async getById(id: string): Promise<Task | null> {

    return this.tasks.find((task) => task.id === id) ?? null;

  }

  async add(task: Task): Promise<void> {

    this.tasks.push(task);

  }

}
```

```
async update(task: Task): Promise<void> {

    const index = this.tasks.findIndex((existing) => existing.id === task.id)
;

    if (index >= 0) {

        this.tasks[index] = task;

    }

}


async remove(id: string): Promise<void> {

    const index = this.tasks.findIndex((task) => task.id === id);

    if (index >= 0) {

        this.tasks.splice(index, 1);

    }

}

}
```

The production repository tests gain two new cases: `remove` erases only the indicated task and persists the absence without reordering, and removing an absent id does not break.

```
// test/data/local-storage-task-repository.test.ts

import { describe, expect, it } from "vitest";

import { LocalStorageTaskRepository } from "../../src/data/local-storage-task-repository";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { MemoryStorage } from "../../helpers/memory-storage";

describe("LocalStorageTaskRepository", () => {

  it("returns an empty list when nothing was saved", async () => {

    const repository = new LocalStorageTaskRepository(new MemoryStorage());

    expect(await repository.getAll()).toEqual([]);

  });
});
```



```
it("preserves tasks between sessions (closing and reopening the app)", async
c () => {

  const storage = new MemoryStorage();

  let tick = 0;

  const createTask = makeCreateTask(

    new LocalStorageTaskRepository(storage),

    () => `id-${tick}`,

    () => new Date(2026, 0, 1, 0, 0, tick++)

  );

  await createTask({ title: "First" });

  await createTask({ title: "Second" });

  // A new instance over the same Storage simulates reopening the app.

  const reopened = new LocalStorageTaskRepository(storage);

  const titles = (await reopened.getAll()).map((task) => task.title);

  expect(titles).toEqual(["First", "Second"]);
```

```
});
```

```
it("update persists the new state without reordering the list", async () =>  
{
```

```
    const storage = new MemoryStorage();
```

```
    let tick = 0;
```

```
    const repository = new LocalStorageTaskRepository(storage);
```

```
    const createTask = makeCreateTask(repository, () => `id-${tick}`, () => new  
Date(2026, 0, 1, 0, 0, tick++));
```

```
    await createTask({ title: "First" });
```

```
    await createTask({ title: "Second" });
```

```
    const second = await repository.getById("id-1");
```

```
    await repository.update({ ...second!, completed: true });
```

```
    const reopened = new LocalStorageTaskRepository(storage);
```

```
    const all = await reopened.getAll();
```

```
    expect(all.map((task) => task.title)).toEqual(["First", "Second"]);
```

```

    expect(all.find((task) => task.id === "id-1")?.completed).toBe(true);

});

it("reads old tasks without the completed field as open", async () => {

    const storage = new MemoryStorage();

    storage.setItem(

        "todo-app.tasks",

        JSON.stringify([
            { id: "x", title: "Old", description: "", createdAt: "2026-01-01T00:00:00.000Z" }
        ])

    );

    const tasks = await new LocalStorageTaskRepository(storage).getAll();

    expect(tasks[0].completed).toBe(false);

});

it("remove erases only the indicated task and persists the absence, without reordering the others", async () => {

```

```
const storage = new MemoryStorage();

let tick = 0;

const repository = new LocalStorageTaskRepository(storage);

const createTask = makeCreateTask(repository, () => `id-${tick}`, () => new Date(2026, 0, 1, 0, 0, tick++));

await createTask({ title: "First" });

await createTask({ title: "Second" });

await createTask({ title: "Third" });

await repository.remove("id-1");

const reopened = new LocalStorageTaskRepository(storage);

const titles = (await reopened.getAll()).map((task) => task.title);

expect(titles).toEqual(["First", "Third"]);

});

it("removing an absent id does not break or change the list", async () => {
```

```

const storage = new MemoryStorage();

const repository = new LocalStorageTaskRepository(storage);

const createTask = makeCreateTask(repository, () => "id-0", () => new Date(2026, 0, 1));

await createTask({ title: "Only one" });

await repository.remove("nonexistent");

expect((await repository.getAll()).map((task) => task.title)).toEqual(["Only one"]);

});

});

```

Finally, the store tests gain the `deleteTask` case, which proves that the reloaded list no longer contains the removed task, on top of the eight that came from the earlier laps.

```

// test/presentation/task-store.test.ts

import { describe, expect, it } from "vitest";

import { makeCompleteTask } from "../../src/domain/usecases/complete-task";

```

```
import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeDeleteTask } from "../../src/domain/usecases/delete-task";

import { makeListTasks } from "../../src/domain/usecases/list-tasks";

import { makeReopenTask } from "../../src/domain/usecases/reopen-task";

import { createTaskStore } from "../../src/presentation/store/task-store";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository"
;

const buildStore = () => {

  const repository = new InMemoryTaskRepository();

  return createTaskStore(

    makeCreateTask(repository),

    makeListTasks(repository),

    makeCompleteTask(repository),

    makeReopenTask(repository),

    makeDeleteTask(repository)
```

```

    );

};

describe("task store", () => {

    it("loads the list of existing tasks", async () => {

        const store = buildStore();

        await store.getState().createTask("Task A", "");

        await store.getState().loadTasks();

        expect(store.getState().tasks.map((task) => task.title)).toEqual(["Task A"]);

    });

    it("creating successfully updates the list and clears the error", async ()
=> {

        const store = buildStore();

```

```
const created = await store.getState().createTask("New", "a description")
;

expect(created).toBe(true);

expect(store.getState().tasks).toHaveLength(1);

expect(store.getState().lastError).toBeNull();

});

it("a title-required failure exposes a message without changing the list",
  async () => {

    const store = buildStore();

    const created = await store.getState().createTask("  ", "");

    expect(created).toBe(false);

    expect(store.getState().tasks).toHaveLength(0);

    expect(store.getState().lastError).toBe("The title is required.");

  });
```



```
it("a duplicate-title failure exposes a specific message", async () => {

  const store = buildStore();

  await store.getState().createTask("Duplicate", "");

  const created = await store.getState().createTask("Duplicate", "");

  expect(created).toBe(false);

  expect(store.getState().lastError).toBe("A task with this title already exists.");

  expect(store.getState().tasks).toHaveLength(1);

});

it("completing a task reflects the completed state in the list", async () => {

  const store = buildStore();

  await store.getState().createTask("Finish the report", "");

  const taskId = store.getState().tasks[0].id;
```

```
    await store.getState().completeTask(taskId);

    expect(store.getState().tasks[0].completed).toBe(true);

  });

it("reopening a completed task returns it to the open state", async () => {

  const store = buildStore();

  await store.getState().createTask("Finish the report", "");

  const taskId = store.getState().tasks[0].id;

  await store.getState().completeTask(taskId);

  await store.getState().reopenTask(taskId);

  expect(store.getState().tasks[0].completed).toBe(false);

});

it("the initial view is 'open'", () => {
```

```

const store = buildStore();

expect(store.getState().filter).toBe("open");

});

it("setFilter switches the active view without touching the full list", async () => {

const store = buildStore();

await store.getState().createTask("Task A", "");

store.getState().setFilter("completed");

expect(store.getState().filter).toBe("completed");

expect(store.getState().tasks).toHaveLength(1);

});

it("deleteTask removes the task and reloads the list without it", async () => {

```

```

const store = buildStore();

await store.getState().createTask("Task A", "");

await store.getState().createTask("Task B", "");

const target = store.getState().tasks.find((task) => task.title === "Task
A");

await store.getState().deleteTask(target!.id);

const titles = store.getState().tasks.map((task) => task.title);

expect(titles).toEqual(["Task B"]);

});

});

```

The Portrait of the Lap

This lap's code is as routine as the earlier ones: a method on the repository, a use case, an action in the store, a button in the view. Four laps, not one new layer. Why a stable architecture absorbs features without growing belongs to *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why

dependencies point inward. You do not need it here: for this lap it is enough to notice that the place for every piece already existed before the feature arrived. What makes it special happened **before** the code, in `analyze`. Without it, the contradiction between "the task vanishes for good" and "a completed task stays in the list" would only surface late, maybe as a reported bug, maybe as a hallway argument over which behavior was right. `analyze` found it by crossing two artifacts from different features, demanded the fix at the source (the spec's wording, not a patch in the tasks) and released the advance only after a second clean pass.

It is the central lesson of Chapter 9 in action: `tasks` breaks the plan into verifiable steps, and `analyze` ensures the artifacts do not contradict each other before a line is written. In Chapter 10.5, the last lap, the spotlight moves to `implement`, and with it a refinement that only surfaced while the code was being written: the title-uniqueness rule, on editing, cannot count the task itself. It is the kind of discovery `implement` brings out, and that goes back to the spec instead of becoming a hidden adjustment.

10 - Edit Task: The Spotlight on implement

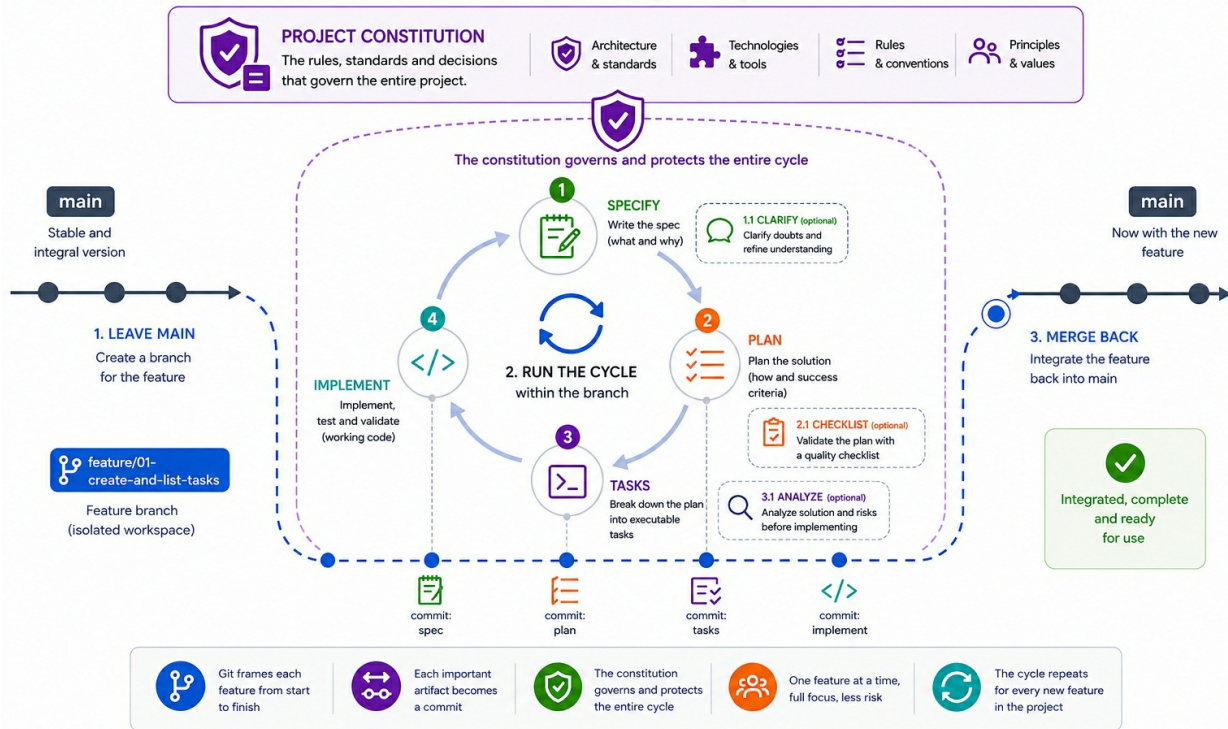
The Last Loop Starts on `main`

`main` is stable with four features: create and list, complete and reopen, filter by state, delete. The fifth repeats the usual gesture: the `005-editor-tarefa` branch cut from `main`, the whole cycle run inside it, and the merge handing the result back at the end. It is the last lap on the map, the feature that closes the app.

The map is the usual one. It is worth reopening it one last time, not to relearn it, but to get your bearings:

THE COMPLETE SDD CYCLE

One feature, from nothing to integrated code



It is the same diagram from Chapter 5; what changes is only where you stand on it. This time the spotlight falls on the step that was still unlit, `implement`, where the task list turns into code that passes the tests.

The feature this time is **editing a task**: the person fixes the text of something they already created, the title or the description. It serves `implement` precisely because it is small. The rules are already known, the entity already exists, the stack was chosen four loops ago; little is left to decide and much to execute. That frees the chapter to look closely at the act of building, which is what has not yet had the spotlight.

And it is minimal on purpose (Principle V, Simplicity). Editing changes the textual content the task **already has**, without inventing a new field. Out of scope, declared as non-goals: **an edit**

history, bulk editing and any field beyond title and description. Keeping the feature lean is what leaves the spotlight on `implement` instead of spending it arguing over scope.

From `specify` to `analyze` , at a Light Pace

You have already seen each of these steps under the spotlight, in the four earlier chapters. Here they run light, just to reach the door of `implement` with the groundwork ready.

`specify` is short: the person wants to fix the text of an existing task. `clarify` ties up the predictable loose ends: editing changes **title and description** (the text the task already has); the title rules apply again (required and not repeated, as in creation); the edit happens **in place**, without changing the position in the list or the creation date; and editing a task that no longer exists is error as value, not an exception. `plan` inherits the usual stack and decides a lean how: one new use case, `EditTask` , that looks the task up, revalidates the title and writes the corrected version. Here the first reward of accumulated simplicity shows up: the repository **gains no new method**. `getById` and `update` have existed since the second loop, and `update` already updates in place without reordering the list. Editing, in the data layer, is the same `update` that completing already used. Four loops of accumulated simplicity pay off here: the fifth feature barely asks for new infrastructure, just a use case that ties together pieces that were already in place.

`checklist` passes without flagging a hole in form. `analyze` crosses spec, plan and tasks and finds them coherent with each other: every requirement has a task, every plan decision comes from a clarification, the order is TDD's. Clean report, zero conflicts. Unlike the previous chapter, editing does not collide with any old

decision: it **reuses** the creation rules instead of reopening them. Keep that "clean report" in mind: it will come back later, when **implement** shows that clean is not a synonym for complete.

implement Reads the Artifacts, Not the Chat

implement arrives: the core-loop step that executes the tasks one by one and produces the code that passes the tests. It is the step that finally writes software, and this chapter's central point is where it draws what it needs to write.

The answer surprises anyone used to chatting with the AI: **implement** **does not reread the chat**. It starts from the versioned artifacts. It checks the prerequisites, opens **tasks.md** and the feature's available documents (the spec, the plan, the data model), and works down the task list phase by phase, executing each one and marking it done.¹ The source of truth is the documentation the earlier steps recorded, not the memory of the window. The agent reads "T002: test of **EditTask**, fails first" and goes and does exactly that, consulting the spec and the plan when it needs the detail, without depending on remembering what was said three steps ago.

This is where the **/clear** habit, repeated since Chapter 5, collects its meaning. Throughout the whole chapter you wiped the context between steps and, even so, nothing was lost, because what mattered was on disk, not in the conversation. **implement** is the final proof of this: it is able to build the feature reading only the artifacts because the artifacts were made to be read on their own. If the step depended on the chat history, **/clear** would have been a shot in the foot; because it depends on the documents, **/clear** was what kept each step lean and focused. The **spec → plan → tasks → code** chain stops being a metaphor here: it is literally the path **implement** walks, link by link.

This changes what "asking an AI for code" means. In improvisation, code comes out of a conversation: you describe, it guesses, you correct in the next reply, and the quality of the result depends on how much of the conversation still fits in the context. In `implement`, code comes out of a chain of reviewed documents, and each document has already passed your approval. The difference is in the source, not the tool.

Red, Green, Refactor

Inside `implement`, code is not born just any way: it is born under the **red-green cycle** of TDD. There are three beats. First the **red**: you write a test of the desired behavior and run it knowing it will fail, because the code does not exist yet. Then the **green**: you write the minimum of code that makes that test pass. Finally, the **refactor**: with the test passing as a safety net, you clean the code without fear, because any regression relights the red at once. It is the TDD of Chapter 1, now in action, and it is Principle IV of the To-Do constitution coming off the page.

Watch the first beat in practice. Task T002 orders writing the `EditTask` test before the implementation. One of the cases says editing must, in fact, change the task's text:

```
it("changes the title and description of an existing task", async () => {  
  
  const repository = new InMemoryTaskRepository();  
  
  const task = await seedTask(repository, "Buy bread", "id-1");  
  
  const editTask = makeEditTask(repository);
```

```
const result = await editTask({ id: task.id, title: "Buy whole-grain bread", description: "at the corner bakery" });

expect(result.kind).toBe("success");

const stored = await repository.getById(task.id);

expect(stored?.title).toBe("Buy whole-grain bread");

expect(stored?.description).toBe("at the corner bakery");

});
```

Running the suite now goes red, and the red is honest: the use case file does not exist yet.

```
FAIL test/domain/edit-task.test.ts [ test/domain/edit-task.test.ts ]
Error: Failed to load url ../../src/domain/usecases/edit-task ... Does the file exist?
```

That is the most common red at the start of a feature: not an assertion that fails, but "the thing you are about to build is not here yet." The test has already pinned the intent; the code is what is missing to meet it.

The green beat is writing that code, and only it. `EditTask` looks the task up, applies the title rules in order and writes the corrected version, returning error as value at each predictable failure:

```
export const makeEditTask = (repository: TaskRepository): EditTask =>

  async ({ id, title, description = "" }) => {

    const task = await repository.getById(id);

    if (task === null) {

      return failure({ kind: "task-not-found" });

    }

    const trimmedTitle = title.trim();

    if (trimmedTitle.length === 0) {

      return failure({ kind: "title-required" });

    }

    const existing = await repository.getAll();

    const isDuplicate = existing.some((other) => other.title.trim() === trimmedTitle);
```

```

    if (isDuplicate) {

        return failure({ kind: "duplicate-title" });

    }

    const edited: Task = { ...task, title: trimmedTitle, description: description.trim() };

    await repository.update(edited);

    return success(edited);

};

```

With the use case in place, the suite closes green:

```

✓ test/domain/edit-task.test.ts (5 tests)

```

```

Test Files  9 passed (9)
Tests      43 passed (43)

```

The third beat, the refactor, is discreet here and instructive for that reason. Notice the duplicate check: it is identical to the one in task creation, back from the first loop. `implement` did not reinvent the rule; it reused the same criterion (title compared after `trim`), keeping the behavior consistent between creating

and editing with the test suite guarding the change. Refactoring is not always rewriting; sometimes it is recognizing that the right path already exists and following it. The cycle repeats in the next phases (the `editTask` action in the store, the edit mode on the screen), always on the same beat: a test that describes the behavior, the minimum code that satisfies it, the cleanup done with the suite green confirming nothing broke.

Why does the test come before, and not after? Because the test written before describes the **intent**; the test written after tends to describe the **code you already wrote**, wrong and all. The red that comes first is the proof that the test really exercises something that does not exist yet; the green that comes next is the proof that the code does what the spec asked, and not what the programmer assumed in the heat of the moment.

Reviewing What the Agent Wrote

`implement` produces code, but it does not produce automatic trust. The generated code is **reviewed** before being accepted, and reviewing here has a precise meaning: checking, in the code, whether it honors the To-Do constitution. It is a reading against principles, where personal taste does not enter.

Look at `EditTask` again with the principles in hand. The **layered architecture** (Principle I) is respected: the use case talks to the domain and the repository, and knows neither the screen nor `localStorage`. The **business logic isolated from the UI** (Principle II) is in the right place: the three decisions (does the task exist? is the title valid? is it a duplicate?) all live in the use case, not the view; the screen only collects text and shows the result. And **error as value** (Principle III) appears at every failure exit: `task-not-found`, `title-required`, `duplicate-title` are `Result`s returned, not exceptions crossing layers. The review confirms that what the constitution

preaches is what the code demonstrates. That is the sense of reviewing: you do not ask "does the code look good?" but "does the code obey the rules we agreed on?". The full repertoire of that check, with far more items than the three the To-Do's constitution fixed, is in *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it here: this chapter's checklist is the constitution itself, and it fits on one page.

And when the review finds a deviation? That is what happened here, and the case is worth more than any principle stated in the abstract. The suite was green, the `build` clean, `analyze` had passed with no conflicts. Even so, the manual validation of the `quickstart` ran into a simple scenario: **editing only the description**, keeping the title. Reduced to a test, the result was this:

```
× EditTask keeps its own title when editing only the description (not a duplicate)
  → expected 'failure' to be 'success'
```

The cause was in plain sight in the code you just read: the duplicate check, copied from creation, compares the new title against **all** tasks, including the one being edited. Keeping your own title fell as a duplicate. The wrong reflex, at this point, would be to open `EditTask` and patch the condition right there, in a hurry. The right reflex is another: **go back to the source artifact**. The defect was not born from a typo; it was born from an incomplete spec. FR-003 said "title not repeated, on the same criterion as creation" and never said whether the task itself counted. The code just executed, faithfully, an ambiguous instruction.

So the fix goes down the chain, in the order it exists. First the **spec**: FR-003 is rewritten to say "not repeated among the **other** tasks; keeping your own title is not a duplicate," with a new clarification recording where the adjustment came from. Then the **tasks**: T002 gains the "keeping your own title returns success" case. Only then the **code**, and now with a test first: the red case above, and the one-line amendment that makes it pass.

```
const isDuplicate = existing.some(

  (other) => other.id !== id && other.title.trim() === trimmedTitle

);
```

The suite closes green again, now with the missing case covered:

```
Test Files  9 passed (9)
Tests      46 passed (46)
```

It is the "fix at the source" of Chapter 9 again, now triggered by **implement** instead of **analyze**: patching the code would have made the test pass and left the spec lying. And there is the lesson the "clean report" was keeping: **checklist** and **analyze** check form and consistency between artifacts, but they do not guess what the wording left implicit. What caught what slipped through were the test and the review. Code outside the expected, before being a programming bug, is a sign of an incomplete artifact asking for a fix back at the source.

Loop Closed, App Complete

With the suite green and the `build` clean, the final path is the familiar one. You commit the work on the branch (the artifacts and the code, including the commit that records the fix at the source) and **merge** `005-editor-tarefa` back into `main`. The main line is stable once again, now with five capabilities where, at the start of this journey, there were none.

And with the fifth feature integrated, **the app is complete**. Create, complete, filter, delete, edit: the life cycle of a personal task, whole, from birth to correction.

The whole-picture look at the five laps comes at the end of this part. Chapter 11 does not open a new loop; it turns back and does the **retrospective**: what each repetition taught, what is gained by trading improvisation for the chain of artifacts, and why the same cycle served five features so different from one another. The app is ready. What is left is to name what it taught us.

Footnotes

Command names, the order of the steps, and the exact form of invocation may evolve between versions of spec-kit; the excerpts in this chapter were generated with version 0.11.8. For details that change per release (flags, subcommands, internal paths), check the official spec-kit documentation instead of fixing them from memory.

10.5 - Editing a Task in Practice: the Whole Loop, File by File

How to Read This Chapter

Chapter 10 followed the fifth and last lap of the cycle with the spotlight on `implement`. Here you see the whole lap, exactly as it was recorded in the example app's repository, in the `005-editor-tarefa` feature.

This is the lap where `implement` discovers something no earlier artifact had noticed. The spec said the edited title should stay "not repeated, on the same criterion as creation." It seemed enough: `checklist` saw no gap, `analyze` came out clean, the tests derived from the tasks went green. It was only in the manual validation of the quickstart, editing **only the description** of a task, that the defect showed up: the naive rule treated keeping your own title as a duplicate, and the edit failed. The fix was not a patch in the code; it went back to the **source**, the spec gained a fifth clarification and FR-003 was rewritten, and only then did the code change. It is exactly the pedagogical point of Chapter 10: `implement` is part of the loop, and what it discovers feeds back into the artifacts, instead of becoming a hidden adjustment.

As in the earlier laps, the artifacts appear as markdown (with headings demoted) and the code in blocks with a sentence of context before each one. Several files are modifications of files you have already seen; when that is the case, the text points out what changed.

The Input: the `/speckit-specify` Prompt

"Edit an existing task in the personal to-do list. The person fixes the text of a task they already created. Out of scope for this feature: an edit history, bulk editing and any new field beyond the ones the task already has."

What `specify` and `clarify` Returned: `spec.md`

The spec below is the **final** version, already with the fifth clarification (Q5) and FR-003 rewritten. Notice the note at the end of Q5: it records that the refinement was discovered in `implement` and that the previous wording ("on the same criterion as creation") was ambiguous. The spec does not hide the fix; it documents it.

Feature Specification: Edit a task

Feature Branch: `005-editor-tarefa`

Created: 2026-06-27

Status: Draft

Input: User description: "Edit an existing task in the personal to-do list. The person fixes the text of a task they already created. Out of scope for this feature: an edit history, bulk editing and any new field beyond the ones the task already has."

Clarifications

- Q: Which fields does editing change (just the title, or title and description)? → A: Title and description. It is the textual content the task already has (introduced in 001); editing is changing that text, without inventing a new field (YAGNI).
- Q: Do the title rules apply again on editing? → A: Yes. The title stays required (it cannot be empty) and not repeated among the tasks, on the same criterion as creation (001). Editing does not loosen the creation rules.
- Q: Does editing change the task's position in the list? → A: No. Editing changes the text in place; `createdAt` does not change and creation order is preserved. The repository already updates in place.
- Q: Editing a task that no longer exists (removed in another tab, for example)? → A: A predictable error (error as value), not an exception that crashes the app, on the same pattern as completing/deleting.
- Q: Does the "title not repeated" rule consider the task itself being edited? → A: No. Uniqueness is against the **other** tasks. Saving an edit keeping the same title (for example, when fixing only the description) is NOT a duplicate. *(Refinement discovered in `implement`: the previous wording, "on the same criterion as creation," was ambiguous on this point and made editing only the description fail as a duplicate.)*

User Scenarios & Testing (mandatory)

User Story 1 - Fix the text of a task (Priority: P1)

The person notices a typo or wants to detail a task they already created more fully. They open the task, adjust the title and/or the description and save. The list now shows the corrected text, in the same position.

Acceptance Scenarios:

1. **Given** an existing task, **When** the person changes the title and saves, **Then** the task now displays the new title, in the same position, and the change survives reopening the app.
2. **Given** an existing task, **When** the person changes only the description and saves (keeping the same title), **Then** the new description is displayed, the title stays intact and the operation succeeds (keeping your own title is not a duplicate).

Edge Cases

- Editing and leaving the title empty (or only spaces) is rejected as a predictable error: the required-title rule applies to editing too.
- Editing and adopting the title of **another** task is rejected as a duplicate; keeping your own title (editing only the description, for example) is allowed.
- Editing a task that no longer exists is handled as a predictable error (error as value), not as an exception.

Requirements (mandatory)

Functional Requirements

- **FR-001:** The system **MUST** allow editing the textual content (title and description) of an existing task.
- **FR-002:** The edited title **MUST** stay required: an empty title or one with only spaces is rejected, on the same criterion as creation (001).
- **FR-003:** The edited title **MUST** stay not repeated among the **other** tasks. Keeping the task-in-edit's own title does **NOT** count as a duplicate (it is the case of fixing only the description).

- **FR-004:** Editing a nonexistent task MUST return a predictable error (error as value), not throw an exception across layers.
- **FR-005:** Editing MUST change the task in place, without changing its position in the list or its `createdAt`.
- **FR-006:** The edit MUST persist: the corrected text survives closing and reopening the app.

Key Entities

- **Task:** the already existing entity (001), with `title` (required, unique) and `description` (optional). Editing changes those two fields; it introduces no new field.

Success Criteria (mandatory)

- **SC-001:** After editing a task's title and reopening the app, the task appears with the new title, in the same position.
- **SC-002:** Trying to save an edit with an empty title keeps the task unchanged and signals the error.
- **SC-003:** Trying to adopt an already-used title keeps the task unchanged and signals the duplicate error.

Assumptions

- The feature extends those for creating/listing (001), completing/reopening (002), filtering (003) and deleting (004), already integrated on `main`; it reuses the Task entity, the repository and the existing layered architecture.
- The To-Do constitution governs the feature: Simplicity/YAGNI, layered architecture, business logic isolated from the UI, error as value and TDD.
- Editing is deliberately minimal (clarification): no edit history, no bulk editing and no new field. If any of those is ever

wanted, it becomes its own feature.

The Completeness Check:

`checklists/requirements.md`

`checklist` here is instructive precisely because it did **not** catch the problem. The notes at the end record it frankly: the duplicate rule was reused from creation "on the same criterion," which "seemed enough on this pass." `checklist` checks form, and the form was good. The hole was semantic, and only the test would reveal it.

Specification Quality Checklist: Edit a task

Created: 2026-06-27

Feature: spec.md

Content Quality

- No implementation details (languages, frameworks, APIs)
- Focused on user value and business needs
- Written for non-technical stakeholders
- All mandatory sections completed

Requirement Completeness

- No [NEEDS CLARIFICATION] markers remain
- Requirements are testable and unambiguous
- Success criteria are measurable

- Success criteria are technology-agnostic
- All acceptance scenarios are defined
- Edge cases are identified
- Scope is clearly bounded
- Dependencies and assumptions identified

Feature Readiness

- All functional requirements have clear acceptance criteria
- User scenarios cover primary flows
- Feature meets measurable outcomes defined in Success Criteria
- No implementation details leak into specification

Notes

- `checklist` found no form gaps: the title rules and the minimal scope (no history, no bulk, no new field) were already pinned down in `clarify`. The duplicate rule was reused from creation (001) "on the same criterion," which seemed enough on this pass.

What `plan` Decided: `plan.md`

This lap's plan has one decision worth highlighting: **no new method on the repository**. `getById` and `update` have existed since 002, and `update` updates in place without reordering, exactly what editing needs. It is the architecture paying dividends: the fifth feature reuses the repository the second one designed.

Implementation Plan: Edit a task

Branch: 005-editor-tarefa | **Spec:** spec.md

Input: Feature specification in specs/005-editor-tarefa/spec.md

Summary

Change the textual content (title and description) of an existing task, in place. The spec says **what** (fix the text, reapplying the title rules and preserving the position); this plan decides **how**: an `EditTask` use case in the domain that looks the task up by id, revalidates the title (required and not repeated), writes the changed version via the repository's `update` and returns `Result` (error as value for the nonexistent task and for the invalid title). The orchestration lives in the store and the editing in the view. **No new method on the repository:** `getById` and `update` have existed since 002, and `update` updates in place without reordering the list, exactly what editing needs. No new field enters the task.

Technical Context

Language/Version: TypeScript 5.x on Node 20+; target: browser (React web). **Inherited from 001-004.**

Main dependencies: React 18, Vite, Zustand. **No new dependencies** (stack inherited from the 001 `plan`).

Persistence: the edit is written to `localStorage` behind `TaskRepository.update`, which replaces the item in place preserving the order. It is the same path completing/reopening (002) already uses.

Testing: Vitest for the `EditTask` use case (domain, no DOM) and for the store action. The repository's `update` has had coverage since 002.

Project type: single-page, single-user web application, no backend.

How-Decisions (what the spec did not pin down)

1. **Where editing lives:** a new `EditTask` use case in the domain. It receives `{ id, title, description }`, looks the task up by id; if it does not exist, it returns `failure({ kind: "task-not-found" })` (FR-004); otherwise it revalidates the title (required, FR-002; not repeated, FR-003) and, if valid, assembles the changed task and calls `update`. The view never talks straight to the repository.
2. **Repository reuse:** no new method. `findById` locates the task and `update` writes the changed version in place (FR-005), without touching `createdAt` or the order. Both have existed since 002.
3. **The title rules:** the same as creation (001), a non-empty title after `trim` and not repeated among the tasks. The revalidation lives in the use case, not the UI.
4. **Where editing lives in the interface:** in the view. Entering edit mode, filling the fields and saving is a UX gesture; the store only orchestrates: it triggers `editTask` and reloads the list.

Constitution Check

Principle	How the plan meets it
I. Layered Architecture	The write sits behind the repository; the rule (does it exist? valid title? then update) sits in the domain use case; the view only collects the text and displays. Dependencies point toward the domain.
II. Business Logic Isolated	The decision "edit an existing task revalidating the title" is a pure use case,

	testable without a UI. Edit mode is UX, not a business decision.
III. Error as Value	A nonexistent task and an invalid title become <code>Result.failure(...)</code> ; no exception crosses layers.
IV. TDD	Tests of the use case (edits title and description; empty title becomes a failure; duplicate title becomes a failure; nonexistent becomes a failure; preserves the position) before implementation.
V. Simplicity/YAGNI	No edit history, no bulk editing, no new field, no new method on the repository . One use case, one store action, one edit mode.
VI. Technology Agnosticism	The spec names no stack; the stack (React/TS/Vite/Zustand) is inherited. The decisions here are about design (where editing lives, who revalidates), not new technology.

Result: PASS. No violations; no rejected draft in this feature.

Traceability clarify/spec → plan

Decision (clarify/spec)	Effect on the plan
Edit title and description (no new field)	<code>EditTask</code> receives <code>{ id, title, description }</code> and rebuilds the task with the same fields.
Title rules apply	The use case revalidates required (FR-002)

on editing	and not repeated (FR-003), reusing the creation criterion.
Position preserved	<code>update</code> writes in place; <code>createdAt</code> and order intact (FR-005).
Nonexistent is error as value	<code>EditTaskFailure</code> includes <code>{ kind: "task-not-found" }</code> , handled in the store.

Project Structure (additions)

```

src/
├── domain/
│   ├── failures/edit-task-failure.ts          # new: title-required | duplicat
├── e-title | task-not-found
│   └── usecases/edit-task.ts                  # new: makeEditTask(repository)
├── presentation/
│   ├── store/task-store.ts                    # + editTask(id, title, descrip
tion)
│   └── components/TaskList.tsx                # + inline edit mode
├── test/
│   ├── domain/edit-task.test.ts               # new: edits; empty/duplicate/n
onexistent become failures; preserves position
│   └── presentation/task-store.test.ts        # + editTask reloads the list

```

Phase 1 - Design

- Model of the operation in data-model.md.
- Manual validation in quickstart.md.

The `analyze` Report: `analyze-report.md`

The report has two parts that tell the whole story of the lap. The first is the normal scan: no conflict, cleared for `implement`. The second is an **addendum**, written later, that records the refinement discovered in the quickstart validation, the cause (an incomplete artifact, not a code slip), the cascade fix starting at the source, and the lesson: checklist and analyze check form and consistency, but they do not replace the test and the review, which catch what the wording left implicit.

Consistency report (`analyze`): Edit a task

Cross-scan of `spec.md` × `plan.md` × `tasks.md` for the `005-editor-tarefa` feature, read together with the artifacts of the features already integrated on `main` (especially `001-criar-tarefa`, where the title rules come from). `analyze` rewrites nothing: it produces the report below and recommends where to fix.

Specification Analysis Report

ID	Category	Severity	Location	Summary	Recommendation
A1	Underspecification (minor)	LOW	<code>005-editor-tarefa/spec.md</code> FR-001, view	The spec does not pin down the editing means (inline mode, modal, its own	According to the spec, it is a decision to be made in the plan

				screen).	
--	--	--	--	----------	--

Coverage Summary

- Total tasks: 8
- Requirement coverage (≥ 1 task): 100% (FR-001..FR-006 mapped to T001-T006)
- Ambiguity count: 1 (LOW)
- Duplication count: 0
- Conflicts: 0

Diagnosis

Unlike 004, the editing wording does not conflict with any earlier feature: it **reuses** the title rules from creation (001) instead of reopening them, and the preserved-position invariant is the same one completing (002) already honors via `update`. The three artifacts (spec, plan, tasks) are coherent with each other: every FR has a task, every plan decision comes from a clarification, and the TDD order is respected.

Next Actions

- Clean report (no conflicts): **cleared for** `/speckit-implement`.
- The only observation (A1) is a UI detail, resolved in `implement`; it does not block.

Addendum: refinement discovered in `implement`

`analyze` came out clean and the tests derived from the tasks went green, but the **quickstart validation** (item 4, "editing only the description") exposed a real defect: FR-003 inherited from

creation the "title not repeated" rule without saying whether the task itself counts. The naive implementation treated keeping the title as a duplicate, and editing only the description failed.

The cause was not a code slip, but an **incomplete artifact**. The fix went in at the **source**:

- `spec.md` : a new clarification Q5, FR-003 rewritten ("among the **other** tasks; keeping your own title is not a duplicate") and Scenario 2 and the edge case adjusted.
- `tasks.md` : T002 gained the "keeping your own title returns success" case.
- Only then the code: a new test (red) and the `other.id !== id` filter in `EditTask` (green).

Lesson recorded: checklist and analyze check form and consistency between artifacts; they do not replace the test and the review, which catch what the wording left implicit.

The `plan`'s Model of the Operation: `data-model.md`

As in the two earlier laps, editing adds neither entity nor field; it models an operation on the Task that already exists, reusing the port's `getById` and `update`.

Data Model: Edit a task

The feature adds **neither entity nor field**. It models an operation on the existing `Task` entity: changing the textual content in place.

Operation

Operation	Input	Output	Predictable failures
EditTask	<code>{ id: string; title: string; description?: string }</code>	<code>Result<Task, EditTaskFailure></code>	<code>title-required</code> (empty title after <code>trim</code>), <code>duplicate-title</code> (title already used), <code>task-not-found</code> (nonexistent id)

Repository reuse

Method (TaskRepository port)	Role in editing
<code>getId(id): Promise<Task null></code>	Locates the task to edit; <code>null</code> becomes <code>task-not-found</code> . Has existed since 002.
<code>update(task): Promise<void></code>	Writes the changed version in place, without reordering the list. Has existed since 002.

No new method is introduced: editing is, in the data layer, the same `update` that completing/reopening already uses.

Preserved invariants

- `createdAt` and the position in the list do not change when editing (FR-005): the task is updated in place.
 - The edited title obeys the same rules as creation: required (FR-002) and not repeated (FR-003).
 - `id` and `completed` are not changed by editing: it touches only `title` and `description`.
-

The Manual Validation: `quickstart.md`

Notice item 4: "editing only the description." It was this step, in the manual validation, that exposed the FR-003 defect. It was here, and in no formal step, that the refinement was born.

Quickstart / Validation: Edit a task

Automated tests

```
npm test          # all green, including edit-task and store.editTask  
  
npm run build     # clean type-check
```

Manual validation (Success Criteria)

1. **SC-001:** create a task, edit the title, save, close and reopen the app → the task appears with the new title, in the same position.
 2. **SC-002:** edit and leave the title empty → the task stays unchanged and the error is signaled.
 3. **SC-003:** edit and adopt another task's title → the task stays unchanged and the duplicate error is signaled.
 4. Edit only the description → the title stays intact and the new description is displayed (FR-001).
 5. The other tasks keep their creation order after the edit (FR-005).
-

The `tasks` Execution List: `tasks.md`

This is the version after the fix: T002 already includes the "keeping your own title, editing only the description, returns success (not a duplicate)" case, in bold. The fix cascade came down from the spec to here before touching the code.

Tasks: Edit a task

Branch: `005-editor-tarefa` | Plan: `plan.md`

TDD order: the test for a behavior comes before the implementation that satisfies it. `[P]` marks tasks that can run in parallel (distinct files, no dependency). Purely mechanical steps (creating a failure type, composing the edit mode in the view) do not require a test of their own.

Phase 1 - Domain (use case)

- **T001** `src/domain/failures/edit-task-failure.ts` : type `EditTaskFailure = { kind: "title-required" } | { kind: "duplicate-title" } | { kind: "task-not-found" }`. (*mechanical step: failure type, no test of its own*)
- **T002** `test/domain/edit-task.test.ts` : tests of `EditTask` (edits an existing task's title and description and returns success; **keeping your own title, editing only the description, returns success (not a duplicate — FR-003)**; an empty title returns `title-required`; a title already used by another task returns `duplicate-title`; a nonexistent task returns `task-not-found`; editing does not change the position in the list or the `createdAt`). **Fails first.**
- **T003** `src/domain/usecases/edit-task.ts` : `makeEditTask(repository)` (looks up by id; nonexistent becomes a failure; revalidates the required and not-repeated title; assembles the changed task

and calls `update`); implement until T002 passes.

Phase 2 - Presentation

- **T004** `test/presentation/task-store.test.ts` : test of `editTask` in the store (after editing, the reloaded list shows the new text in the same position). **Fails first.**
- **T005** `src/presentation/store/task-store.ts` : `editTask(id, title, description)` action (triggers the use case, translates the failure into a message and reloads `tasks`); implement until T004 passes.
- **T006** `src/presentation/components/TaskList.tsx` : inline edit mode on each item (enter editing, adjust title/description, save or cancel). (*mechanical view step: composition, covered by the manual validation*)

Phase 3 - Validation

- **T007** `npm test` (all green) and `npm run build` (clean type-check).
 - **T008** Manual validation of Success Criteria SC-001..SC-003 and of the quickstart.
-

The Code `implement` Generated

The highlight of this lap is in the `EditTask` use case, and more precisely in the line that changed after the discovery. I show the final code first and, right below, the exact diff of the refinement, so you can see the size of the fix: one condition.

Domain

The predictable failures of editing, gathering the two title rules (inherited from creation) with the nonexistent task (inherited from completing/deleting).

```
// src/domain/failures/edit-task-failure.ts

/**

* Predictable failures when editing a task
.

*

* `title-required`: an empty title or only spaces after `trim` (FR-002)
.

* `duplicate-title`: another task with the same title already exists (FR-003
).

* `task-not-found`: no task with the given id exists (FR-004)
.

*
/

export type EditTaskFailure =

| { readonly kind: "title-required" }

| { readonly kind: "duplicate-title" }

| { readonly kind: "task-not-found" };
```

The `EditTask` use case, in its final version. Notice the duplicate check: the `other.id !== id` filter is the heart of the fix, it is what makes uniqueness hold against the **other** tasks, and not against the task itself.

```
// src/domain/usecases/edit-task.ts

import type { Task } from "../../entities/task";

import type { EditTaskFailure } from "../../failures/edit-task-failure";

import type { TaskRepository } from "../../repositories/task-repository";

import { failure, success, type Result } from "../../result";

export type EditTaskInput = { id: string; title: string; description?: string };

export type EditTask = (input: EditTaskInput) => Promise<Result<Task, EditTaskFailure>>;

/**

 * Edits the textual content of an existing task, revalidating the title and
```

```
* returning error as value: nonexistent task (FR-004), required title (FR-002)
```

```
* and a title not repeated among the OTHER tasks (FR-003) – keeping your own
```

```
* title is not a duplicate. The task is updated in place, without changing the
```

```
* position in the list or the `createdAt` (FR-005)  
.
```

```
*  
/
```

```
export const makeEditTask = (repository: TaskRepository): EditTask =>
```

```
  async ({ id, title, description = "" }) => {
```

```
    const task = await repository.getById(id);
```

```
    if (task === null) {
```

```
      return failure({ kind: "task-not-found" });
```

```
    }
```

```
    const trimmedTitle = title.trim();
```

```
if (trimmedTitle.length === 0) {

    return failure({ kind: "title-required" });

}

const existing = await repository.getAll();

const isDuplicate = existing.some(

    (other) => other.id !== id && other.title.trim() === trimmedTitle

);

if (isDuplicate) {

    return failure({ kind: "duplicate-title" });

}

const edited: Task = { ...task, title: trimmedTitle, description: description.trim() };

await repository.update(edited);
```



```
return success(edited);  
  
};
```

And here is the refinement, isolated. The commit [Fix at the source: FR-003 does not count the task itself as a duplicate](#) changed six lines, of which the one that matters is the check's condition. The naive version compared only the title; the corrected one adds `other.id !== id`.

```
// src/domain/usecases/edit-task.ts  
  
-   const isDuplicate = existing.some((other) => other.title.trim() === trimmedTitle);  
  
+   const isDuplicate = existing.some(  
  
+       (other) => other.id !== id && other.title.trim() === trimmedTitle  
  
+   );
```

The fix changed one condition, but it walked the whole chain before reaching here, as the artifacts above record.

Presentation

The store gains `editTask` and, with it, a second error-translation function, `editMessageFor`, because editing has a failure creation did not have (`task-not-found`). The orchestrate-and-reload pattern is the same as the earlier laps.

```
// src/presentation/store/task-store.ts

import { createStore } from "zustand/vanilla";

import type { Task } from "../../domain/entities/task";

import type { TaskFilter } from "../../domain/entities/task-filter";

import type { CreateTaskFailure } from "../../domain/failures/create-task-failure";

import type { EditTaskFailure } from "../../domain/failures/edit-task-failure";

import type { CompleteTask } from "../../domain/usecases/complete-task";

import type { CreateTask } from "../../domain/usecases/create-task";

import type { DeleteTask } from "../../domain/usecases/delete-task";

import type { EditTask } from "../../domain/usecases/edit-task";

import type { ListTasks } from "../../domain/usecases/list-tasks";

import type { ReopenTask } from "../../domain/usecases/reopen-task";

export type TaskStoreState = {

  readonly tasks: Task[];
```

```

readonly filter: TaskFilter;

readonly lastError: string | null;

loadTasks: () => Promise<void>;

createTask: (title: string, description: string) => Promise<boolean>;

completeTask: (id: string) => Promise<void>;

reopenTask: (id: string) => Promise<void>;

deleteTask: (id: string) => Promise<void>;

editTask: (id: string, title: string, description: string) => Promise<boolean>;

setFilter: (filter: TaskFilter) => void;

};

const messageFor = (failure: CreateTaskFailure): string => {

  switch (failure.kind) {

    case "title-required":

      return "The title is required.";

```

```
        case "duplicate-title":

            return "A task with this title already exists.";

    }

};

const editMessageFor = (failure: EditTaskFailure): string => {

    switch (failure.kind) {

        case "title-required":

            return "The title is required.";

        case "duplicate-title":

            return "A task with this title already exists.";

        case "task-not-found":

            return "This task no longer exists.";

    }

};
```

```
/**  
  
 * Presentation store (Zustand) that orchestrates the domain use cases and  
 * exposes the state to the UI. It holds no business rule: it only triggers  
 * `createTask`/`listTasks` and translates the typed failure into a message for  
 * the screen  
 */
```

```
export const createTaskStore = (
```

```
  createTask: CreateTask,
```

```
  listTasks: ListTasks,
```

```
  completeTask: CompleteTask,
```

```
  reopenTask: ReopenTask,
```

```
  deleteTask: DeleteTask,
```

```
  editTask: EditTask
```

```
) =>
```

```
createStore<TaskStoreState>((set) => ({

  tasks: [],

  filter: "open",

  lastError: null,

  loadTasks: async () => {

    const tasks = await listTasks();

    set({ tasks, lastError: null });

  },

  createTask: async (title, description) => {

    const result = await createTask({ title, description });

    if (result.kind === "failure") {

      set({ lastError: messageFor(result.error) });

    }

    return false;

  }

})
```

```
}

const tasks = await listTasks();

set({ tasks, lastError: null });

return true;

},

completeTask: async (id) => {

  await completeTask(id);

  const tasks = await listTasks();

  set({ tasks });

},

reopenTask: async (id) => {

  await reopenTask(id);
```

```
    const tasks = await listTasks();

    set({ tasks });

},

deleteTask: async (id) => {

    await deleteTask(id);

    const tasks = await listTasks();

    set({ tasks });

},

editTask: async (id, title, description) => {

    const result = await editTask({ id, title, description });

    if (result.kind === "failure") {

        set({ lastError: editMessageFor(result.error) });

    }

    return false;

}
```



```

    }

    const tasks = await listTasks();

    set({ tasks, lastError: null });

    return true;

  },

  setFilter: (filter) => {

    set({ filter });

  }

}));

```

The `TaskList` gains inline edit mode: each item toggles between display and a small form with title, description and the Save/Cancel buttons. The draft state (`editingId` , `draftTitle` , `draftDescription`) is **ephemeral UI state**, it lives in the component, not in the store or the domain. Save only closes the edit if `editTask` returned success; on error, the message appears and the form stays open.

```
// src/presentation/components/TaskList.tsx
```

```
import { useState } from "react";

import { selectByFilter } from "../../domain/entities/task-filter";

import { useTaskStore } from "../../store/task-store-context";

/**

 * Lists the active view's tasks in creation order; an empty list is a valid
 * state. The choice of which tasks appear comes from the domain

 * (`selectByFilter`); here we only display the result. A completed task is
 * shown struck through, with the action toggling between completing and
 * reopening. Each item can enter inline edit mode, which triggers `editTask`
 * the store on save
 *
 */

export function TaskList() {
```

```
const tasks = useTaskStore((state) => state.tasks);

const filter = useTaskStore((state) => state.filter);

const completeTask = useTaskStore((state) => state.completeTask);

const reopenTask = useTaskStore((state) => state.reopenTask);

const deleteTask = useTaskStore((state) => state.deleteTask);

const editTask = useTaskStore((state) => state.editTask);

const lastError = useTaskStore((state) => state.lastError);


const [editingId, setEditingId] = useState<string | null>(null);

const [draftTitle, setDraftTitle] = useState("");

const [draftDescription, setDraftDescription] = useState("");


const visibleTasks = selectByFilter(tasks, filter);

const confirmAndDelete = (id: string, title: string) => {

    if (window.confirm(`Delete "${title}"? This action cannot be undone.`)) {
```

```
    void deleteTask(id);

}

};

const startEditing = (id: string, title: string, description: string) => {

    setEditingId(id);

    setDraftTitle(title);

    setDraftDescription(description);

};

const saveEditing = async (id: string) => {

    const saved = await editTask(id, draftTitle, draftDescription);

    if (saved) {

        setEditingId(null);

    }

}
```

```

};

if (visibleTasks.length === 0) {

  return <p className="task-list__empty">No tasks in this view.</p>;

}

return (

  <ul className="task-list">

    {visibleTasks.map((task) => (

      <li

        key={task.id}

        className={task.completed ? "task-list__item task-list__item--done"
: "task-list__item"}

      >

        {editingId === task.id ? (

          <div className="task-list__edit">

            <input

```

```

        aria-label="Edit title"

        value={draftTitle}

        onChange={(event) => setDraftTitle(event.target.value)}

    />

    <textarea

        aria-label="Edit description"

        value={draftDescription}

        onChange={(event) => setDraftDescription(event.target.value)}

    />

    {lastError !== null && <p role="alert" className="task-list__error">{lastError}</p>}

    <button type="button" onClick={() => saveEditing(task.id)}>Save
</button>

    <button type="button" onClick={() => setEditingId(null)}>Cancel
</button>

</div>

) : (

```

```
<>

<label className="task-list__check">

  <input

    type="checkbox"

    checked={task.completed}

    aria-label={task.completed ? `Reopen ${task.title}` : `Complete ${task.title}`}

    onChange={() => (task.completed ? reopenTask(task.id) : completeTask(task.id))}

  />

  <span>

    <h3>{task.title}</h3>

    {task.description.length > 0 && <p>{task.description}</p>}

  </span>

</label>

<button

  type="button"
```

```
        className="task-list__edit-button"

        aria-label={`Edit ${task.title}`}

        onClick={() => startEditing(task.id, task.title, task.description)}
      >

        Edit

    </button>

    <button

        type="button"

        className="task-list__delete"

        aria-label={`Delete ${task.title}`}

        onClick={() => confirmAndDelete(task.id, task.title)}

      >

        Delete

    </button>

  </>
```



```

    })

    </li>

  })}

</ul>

);

}

```

The `main.tsx` composes the last use case at the boundary. The `createTaskStore` signature reached six use cases over five laps, and the composition point stays the same, and the only one.

```

// src/main.tsx

import { StrictMode } from "react";

import { createRoot } from "react-dom/client";

import { LocalStorageTaskRepository } from "../data/local-storage-task-repository";

import { makeCompleteTask } from "../domain/usecases/complete-task";

import { makeCreateTask } from "../domain/usecases/create-task";

import { makeDeleteTask } from "../domain/usecases/delete-task";

```

```
import { makeEditTask } from "../domain/usecases/edit-task";

import { makeListTasks } from "../domain/usecases/list-tasks";

import { makeReopenTask } from "../domain/usecases/reopen-task";

import { App } from "../presentation/App";

import { createTaskStore } from "../presentation/store/task-store";

import { TaskStoreProvider } from "../presentation/store/task-store-context";

import "../index.css";

// Composition at the boundary: the concrete repository is instantiated here
and

// injected into the use cases; only at this point do the layers meet.

const repository = new LocalStorageTaskRepository();

const store = createTaskStore(

  makeCreateTask(repository),

  makeListTasks(repository),

  makeCompleteTask(repository),
```

```
    makeReopenTask(repository),

    makeDeleteTask(repository),

    makeEditTask(repository)

);

const rootElement = document.getElementById("root");

if (rootElement === null) {

    throw new Error("Element #root not found in index.html.");

}

createRoot(rootElement).render(

    <StrictMode>

        <TaskStoreProvider store={store}>

            <App />

        </TaskStoreProvider>
```

```
</StrictMode>

);
```

Tests

The `EditTask` tests include, in second place, exactly the case the refinement created: "keeps its own title when editing only the description (not a duplicate)." It is the test that was missing, and that made the defect visible as red before it turned green.

```
// test/domain/edit-task.test.ts

import { describe, expect, it } from "vitest";

import { makeCreateTask } from "../../src/domain/usecases/create-task";

import { makeEditTask } from "../../src/domain/usecases/edit-task";

import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository";

const seedTask = async (repository: InMemoryTaskRepository, title: string, id
: string) => {

    const createTask = makeCreateTask(repository, () => id);

    const result = await createTask({ title });
```

```
    if (result.kind === "failure") {

        throw new Error("seed failed");

    }

    return result.value;

};

describe("EditTask", () => {

    it("changes the title and description of an existing task", async () => {

        const repository = new InMemoryTaskRepository();

        const task = await seedTask(repository, "Buy bread", "id-1");

        const editTask = makeEditTask(repository);

        const result = await editTask({ id: task.id, title: "Buy whole-grain bread", description: "at the corner bakery" });

        expect(result.kind).toBe("success");

    });

});
```

```
const stored = await repository.getById(task.id);

expect(stored?.title).toBe("Buy whole-grain bread");

expect(stored?.description).toBe("at the corner bakery");

});

it("keeps its own title when editing only the description (not a duplicate)", async () => {

    const repository = new InMemoryTaskRepository();

    const task = await seedTask(repository, "Buy bread", "id-1");

    const editTask = makeEditTask(repository);

    const result = await editTask({ id: task.id, title: "Buy bread", description: "at the bakery" });

    expect(result.kind).toBe("success");

    const stored = await repository.getById(task.id);

    expect(stored?.title).toBe("Buy bread");
```

```

    expect(stored?.description).toBe("at the bakery");

});

it("returns title-required when the title becomes empty", async () => {

    const repository = new InMemoryTaskRepository();

    const task = await seedTask(repository, "Buy bread", "id-1");

    const editTask = makeEditTask(repository);

    const result = await editTask({ id: task.id, title: "  " });

    expect(result).toEqual({ kind: "failure", error: { kind: "title-required"
    } });

});

it("returns duplicate-title when the new title is already used by another task", async () => {

    const repository = new InMemoryTaskRepository();

    await seedTask(repository, "Buy bread", "id-1");

```

```
const second = await seedTask(repository, "Wash the dishes", "id-2");

const editTask = makeEditTask(repository);

const result = await editTask({ id: second.id, title: "Buy bread" });

expect(result).toEqual({ kind: "failure", error: { kind: "duplicate-title" } });

});

it("returns task-not-found when the task does not exist", async () => {

const repository = new InMemoryTaskRepository();

const editTask = makeEditTask(repository);

const result = await editTask({ id: "nonexistent", title: "Anything" });

expect(result).toEqual({ kind: "failure", error: { kind: "task-not-found" } });

});
```



```
it("does not change the task's position in the list or the createdAt when editing", async () => {

  const repository = new InMemoryTaskRepository();

  let tick = 0;

  const createTask = makeCreateTask(repository, () => `id-${tick}`, () => new Date(2026, 0, 1, 0, 0, tick++));

  await createTask({ title: "First" });

  await createTask({ title: "Second" });

  await createTask({ title: "Third" });

  const original = await repository.getById("id-1");

  const editTask = makeEditTask(repository);

  await editTask({ id: "id-1", title: "Second renamed" });

  const tasks = await repository.getAll();

  expect(tasks.map((task) => task.title)).toEqual(["First", "Second renamed", "Third"]);

  expect(tasks[1].createdAt).toBe(original?.createdAt);
```

```
});  
  
});
```

The store tests gain two `editTask` cases (a successful edit in the same position; a duplicate title exposes a message and does not change the list), closing the suite that grew from lap to lap without ever being rewritten.

```
// test/presentation/task-store.test.ts  
  
import { describe, expect, it } from "vitest";  
  
import { makeCompleteTask } from "../../src/domain/usecases/complete-task";  
  
import { makeCreateTask } from "../../src/domain/usecases/create-task";  
  
import { makeDeleteTask } from "../../src/domain/usecases/delete-task";  
  
import { makeEditTask } from "../../src/domain/usecases/edit-task";  
  
import { makeListTasks } from "../../src/domain/usecases/list-tasks";  
  
import { makeReopenTask } from "../../src/domain/usecases/reopen-task";  
  
import { createTaskStore } from "../../src/presentation/store/task-store";  
  
import { InMemoryTaskRepository } from "../../helpers/in-memory-task-repository";  
;
```

```
const buildStore = () => {

  const repository = new InMemoryTaskRepository();

  return createStore(

    makeCreateTask(repository),

    makeListTasks(repository),

    makeCompleteTask(repository),

    makeReopenTask(repository),

    makeDeleteTask(repository),

    makeEditTask(repository)

  );

};

describe("task store", () => {

  it("loads the list of existing tasks", async () => {

    const store = buildStore();
```

```
    await store.getState().createTask("Task A", "");

    await store.getState().loadTasks();

    expect(store.getState().tasks.map((task) => task.title)).toEqual(["Task A"]);

  });

  it("creating successfully updates the list and clears the error", async () => {

    const store = buildStore();

    const created = await store.getState().createTask("New", "a description");

    expect(created).toBe(true);

    expect(store.getState().tasks).toHaveLength(1);

    expect(store.getState().lastError).toBeNull();

  });
```

```
it("a title-required failure exposes a message without changing the list",  
  async () => {
```

```
    const store = buildStore();
```

```
    const created = await store.getState().createTask("  ", "");
```

```
    expect(created).toBe(false);
```

```
    expect(store.getState().tasks).toHaveLength(0);
```

```
    expect(store.getState().lastError).toBe("The title is required.");
```

```
  });
```

```
it("a duplicate-title failure exposes a specific message", async () => {
```

```
    const store = buildStore();
```

```
    await store.getState().createTask("Duplicate", "");
```

```
    const created = await store.getState().createTask("Duplicate", "");
```

```
    expect(created).toBe(false);

    expect(store.getState().lastError).toBe("A task with this title already exists.");

    expect(store.getState().tasks).toHaveLength(1);

  });

  it("completing a task reflects the completed state in the list", async () => {

    const store = buildStore();

    await store.getState().createTask("Finish the report", "");

    const taskId = store.getState().tasks[0].id;

    await store.getState().completeTask(taskId);

    expect(store.getState().tasks[0].completed).toBe(true);

  });

  it("reopening a completed task returns it to the open state", async () => {
```

```
const store = buildStore();

await store.getState().createTask("Finish the report", "");

const taskId = store.getState().tasks[0].id;

await store.getState().completeTask(taskId);

await store.getState().reopenTask(taskId);

expect(store.getState().tasks[0].completed).toBe(false);

});

it("the initial view is 'open'", () => {

  const store = buildStore();

  expect(store.getState().filter).toBe("open");

});
```

```

    it("setFilter switches the active view without touching the full list", async () => {

        const store = buildStore();

        await store.getState().createTask("Task A", "");

        store.getState().setFilter("completed");

        expect(store.getState().filter).toBe("completed");

        expect(store.getState().tasks).toHaveLength(1);

    });

    it("deleteTask removes the task and reloads the list without it", async () => {

        const store = buildStore();

        await store.getState().createTask("Task A", "");

        await store.getState().createTask("Task B", "");

        const target = store.getState().tasks.find((task) => task.title === "Task A");
    });

```



```

    await store.getState().deleteTask(target!.id);

    const titles = store.getState().tasks.map((task) => task.title);

    expect(titles).toEqual(["Task B"]);

  });

  it("editTask changes the task's text and reloads the list in the same position", async () => {

    const store = buildStore();

    await store.getState().createTask("Task A", "");

    await store.getState().createTask("Task B", "");

    const target = store.getState().tasks.find((task) => task.title === "Task A");

    await store.getState().editTask(target!.id, "Task A revised", "with a note");

    const titles = store.getState().tasks.map((task) => task.title);

    expect(titles).toEqual(["Task A revised", "Task B"]);
  });

```

```

    expect(store.getState().lastError).toBeNull();

});

it("editTask with a duplicate title exposes a message and does not change the list", async () => {

    const store = buildStore();

    await store.getState().createTask("Task A", "");

    await store.getState().createTask("Task B", "");

    const target = store.getState().tasks.find((task) => task.title === "Task B");

    await store.getState().editTask(target!.id, "Task A", "");

    const titles = store.getState().tasks.map((task) => task.title);

    expect(titles).toEqual(["Task A", "Task B"]);

    expect(store.getState().lastError).toBe("A task with this title already exists.");

});

```

```
});
```

The Second Iteration, in the Order It Happened

So far you have read this lap as if it ran in a straight line, from requirement to reviewed code. It did not. Between the green `implement` and the merge there was a second pass over the same artifacts, and that pass is what completes the shape of the loop: requirement, spec, plan, tasks, code, review, and then back to the start with whatever the review found.

Here is the real chronology. The suite was green and `npm run build` was clean. What was missing was the cheapest step, and the easiest one to skip: opening the app and running the five items of `quickstart.md` by hand. Items 1, 2, 3 and 5 passed. Item 4, "editing only the description," returned the message "A task with that title already exists" on screen, for a task whose title nobody had touched.

The tempting reflex, in that minute, was to open `edit-task.ts` and settle it there. The code fix is small and obvious, and you have already seen it: an id comparison in the duplicate check. Half a dozen lines, one commit, move on. What is wrong with that path is not the immediate result, which would be identical, but what it leaves behind. FR-003 would still be written the ambiguous way, T002 would still be missing the case, and the next person to read the spec in order to build anything on top of editing would inherit the same trap, now silently contradicted by the code.

So the fix went up. The order of the commits, on the `005-editor-tarefa` branch, was:

1. `spec.md` : clarification Q5 went in ("Does the not-repeated title rule consider the task being edited? No"), FR-003 was

rewritten to say "among the **other** tasks," and Scenario 2 and the matching edge case gained the same precision.

2. `plan.md` : nothing to change. The plan said "revalidates the title, reusing the creation criterion," and the criterion was still the one from creation. It was the wording of the requirement that was incomplete, not the design of the construction. Not every fix at the source travels through every artifact, and touching the plan just to keep the symmetry would be noise.
3. `tasks.md` : T002 gained the case "keeping your own title, editing only the description, returns success."
4. **The test**: written first, red, named after the rule and not after the bug.
5. **The code**: the condition fixed, the test green, the whole suite green again.

Notice what that order cost. Some fifteen minutes more than the direct patch, and four files touched instead of one. In exchange, the spec became true again. It is the same move Chapter 9 showed when `analyze` demanded coherence between artifacts, with one difference that matters: there the conflict surfaced in a formal step, with a report pointing the finger; here it surfaced with the app open on screen, in a ten-second manual test no tool ran for you. The two gates catch different things, and neither excuses the other.

That second pass is what kept `quickstart.md` from being end-of-lap paperwork. It is the last chance to discover that the spec was incomplete while the spec is still the living document of an open branch, and not a historical text nobody reads anymore.

The Portrait of the Lap, and of the Five

This last lap closes the material's argument in two ways. First, it shows that `implement` is not the mute end of the loop: it discovers things. The uniqueness rule that does not count the task itself was not in anyone's head when the spec was written; it emerged running the app, and climbed back up to the spec instead of becoming a clever hidden `if`. The artifact stays the truth.

Second, and looking at the five laps together: the `Task` entity gained one field (`completed`) and nothing more; the `TaskRepository` grew to five methods, each added when a feature needed it and reused by the following ones; the composition point in `main.tsx` stays single; the test suite only grew, was never rewritten. Five features, five laps of the same loop, and the architecture Chapter 6.5 showed is intact in 10.5. What makes an architecture survive that kind of pressure is the subject of *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume in this trilogy, which answers where each rule lives and why dependencies point inward. You do not need it to finish this book: the proof that matters here is in the five commits you have just followed. That was the thesis: the method does not guarantee you get it right the first time (005 proves it does not), it guarantees that, when you get it wrong, the fix goes in at the source and the rest stays standing.

That movement, fixing at the source, showed up here as the account of one lap. The next chapter takes it as a method: which doors a spec gets corrected through, in what order the artifacts move, and when insisting on the correction stops making sense.

10b - Iterative Refinement: The Spec After It Already Exists

No Document Is Born Finished

The cycle you followed five times treats the spec as a starting point. It gets written, reviewed, approved, and then plan, tasks, code. Read that way, from the outside, the specification looks like a document you write once.

None of the five worked like that. All of them changed after they were finished, and the last one changed after the code was already written and passing its tests. That is not a failure of the method: it is the method working. A spec that never changes is either luck, or a spec nobody is using.

What this chapter does is look at the movement itself. When a spec changes, what exactly changes in it, who found out it needed to change, and what else has to move along with it. The fifth feature's lap, file by file, is in Chapter 10.5; what matters here is the shape of the movement, which will serve your specs, and yours will have nothing to do with a to-do app.

Four Doors the Change Comes Through

A spec can be corrected at four moments in the cycle, and each one uncovers a different kind of defect. Know all four by the question each one is able to ask, because that also tells you what

none of them catches.

clarify catches the ambiguity you can see yourself on a reread. It works by closed question, one at a time, and what it uncovers are decisions the spec left open without noticing: does the filter survive closing the app? is completing what is already completed an error? The change it produces always has the same shape: a clarification recorded with its date and a requirement either added or rewritten. It is the cheapest of the four doors, because there is no plan and no code yet to undo.

checklist catches the structural gap. It knows nothing about your domain, and that is what makes it useful: it checks whether the spec has the parts a spec has to have. It was **checklist** that noticed the filter spec fixed three views and never said which one appears when the screen opens. A human reading that document would agree with it three times without spotting the hole, because the list of options looked complete.

analyze catches the contradiction between artifacts. It cross-checks spec, plan and tasks, and cross-checks the current spec against the earlier ones. It is the only one of the four doors that sees outside the document, and that is why it is the only one that can find the same verb meaning two things in different features.

Manual validation catches what was consistent and wrong.

This is the door that matters most, because it is the one left over. The three before it check form, completeness and coherence, and a spec can pass all three while being ambiguous, as long as it is uniformly ambiguous. That is what happened on the fifth lap: **checklist** with no gap, **analyze** with no conflict, green tests, and the defect waiting for the first time somebody edited only a task's description.

Hold on to that, because it is the practical conclusion of this chapter: **no automatic step can tell that a clear sentence means something else in the head of whoever reads it.** Only use tells.

The Chain, in the Order Things Move

Chapter 6b already named the defect of fixing at the end of the chain and said why it ruins the method. From there comes the question that separates the two cases, asked before you open the code file: which artifact was this defect born in? A typo was born in the code; a behavior nobody had agreed on was born in the sentence that failed to agree on it.

Suppose the answer is that it was born up top. What this chapter adds is the mechanics: fixing at the source has an order, and it is the cycle's own order. Each link moves only after the one before it is right.

The spec first. Rewrite the requirement and record the new clarification, with the reason. The record matters as much as the correction: without it, three months from now somebody will look at that odd sentence, with an explicit exception that reads like paranoia, and "clean up" the text back into the ambiguous version.

Then the plan. Often nothing changes, and checking that nothing changes is part of the work. If the correction is a matter of wording, the plan stays valid and you note that. If it touches a technical decision, the plan changes before the tasks do.

Then the tasks. A new or widened requirement usually becomes a new case in a test task that already exists, not a new task. The sign that it became a new task is that the correction opened up behavior that was not foreseen anywhere.

Then the test, red. This is the link you cannot skip. The test that reproduces the defect has to be written and has to fail before the fix exists, because that is what proves it is testing what you think it is testing.

The code last. It tends to be the smallest part. On the To-Do's fifth lap it was one more condition in a comparison.

Notice that this sequence is the same thing as the whole cycle, at a smaller scale. You are not stepping outside the method to fix an exception; you are taking a short lap inside it.

What Changes and What Does Not

Not every spec correction is the same, and it pays to tell three sizes apart, because the cost differs a lot.

Precision: the rule was right and the sentence was loose. It is the most common case and the cheapest. One line of the spec changes, one test case changes, little code changes. The To-Do's fifth lap was this size.

Decision: the spec said one thing and you concluded it was the wrong thing. Here the requirement really changes, and the plan and the tasks quite likely change with it. It is still the same feature.

Boundary: what turned up does not belong to this spec. Somebody asked for the filter to remember the last view chosen; that is a new capability, with state of its own and a life cycle of its own. The right correction is to record the refusal in the clarification, with the reason, and leave it for a future spec. A spec that absorbs someone else's boundary out of convenience stops fitting in one lap, and Chapter 6b already showed what that costs.

Confusing the three is what produces the feeling that "specifying is pointless, everything always changes". Almost nothing ever changes. One sentence changes, and the sense of chaos comes from treating all three with the same ceremony.

When the Spec Stops Being Trustworthy

There is a limit, and it deserves saying. If the same spec has come back for repair four or five times, the problem is not the wording of each sentence: it is that the document is trying to describe something that has not been understood yet.

The sign is the nature of the corrections, not their number. Several precision corrections are a document being sharpened. Decision corrections in a row, each reversing the one before, are a document being discovered, and discovery is exploration work, not specification work. The honest way out is to stop, poke at the code without ceremony until you understand what the question is, throw away what you made, and write the spec with the answer in hand.

That is not abandoning the method. It is the same good sense from Chapter 2c, applied later: SDD pays off when you know what you want and you need someone else, human or machine, to do exactly that. It does not pay off while you are still finding out what you want.

The Document That Outlives the Project

A spec maintained across five laps becomes something that was in nobody's plan: the record of why the system is the way it is.

Look at the five clarifications of the To-Do's last feature. They do not explain what the code does, which the code already explains on its own. They explain why it does it that way: why editing does not reorder the list, why editing something that vanished returns an error instead of blowing up, why keeping your own title does not count as a duplicate. Each of those answers cost a question, and none of them is in the code.

That is why keeping the spec updated is worth it even when nobody is going to read it again that week. The document is not there to guide an implementation that already happened. It is there for the next person, who may be you in six months, with no memory at all of the conversation that produced that line.

Five laps have closed. What they taught together, which none of them taught alone, is the subject of the next chapter.

11 - The Whole App Was Born From Five Loops

Stop and Look Back

The app is ready. You can open the To-Do now, create a task, complete it, filter the list, delete what no longer serves and edit what you changed your mind about. Five capabilities where, at the start of this journey, there were none. It is tempting to move on, close the editor and consider the matter closed.

Don't close it yet. Before moving on, it is worth stopping in front of the finished app and looking back, not to touch anything else, but to see how you got here. This chapter opens no sixth feature and runs no more cycles. It does the opposite of what the earlier ones did: instead of building, it consolidates. It is a retrospective.

What we are going to look at is not the To-Do's screens. Screens are forgotten, and nobody needs one more task app in the world. What we are going to look at is the path each feature walked, because it was always the same path, and it is the one you carry with you to the next project. From up here, with the whole app in view, that path becomes visible in a way no single chapter could show.

The Whole App Is Five Merges Into `main`

Look at the project through the git history, and its shape appears without evasion. `main`, the main and stable line, did not grow all at once. It received five integrations, one at a time, and between one and the next it stood still, whole, ready to use. Each integration was a **merge**, and each merge brought in a whole feature that had been built far away, on a separate **branch**.

There were five branches, and none of them existed by chance. Each branched off `main`, sheltered the development of a feature from start to finish and came back through the merge when the feature was ready and whole. Creating and listing tasks branched off, walked the path, came back. Completing and reopening, the same. Filtering, deleting, editing: each in its turn, each on its branch, each with its merge.

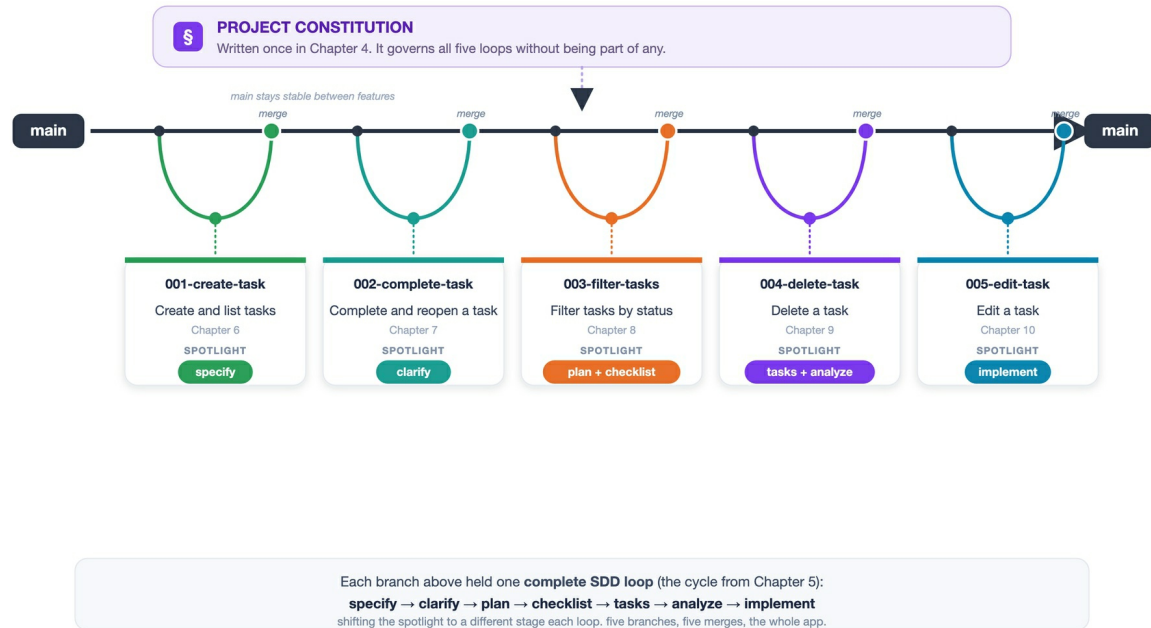
And inside each branch, always, the same cycle. The feature was not improvised in a hurry or written all at once. It was born from a specification, was planned, split into tasks and built under tests, exactly as Chapter 5 promised. Five branches, five loops, five merges. Add the three and you don't have five loose features: you have a whole app, which is the accumulated result of five disciplined repetitions of the same process. The app was not born all at once or from a burst of inspiration. It was born from one lap, repeated five times.

The Timeline in One Figure

Seen from above, this story fits in a single figure. It is the same language as the diagram in Chapter 5, `main` branching off and receiving the merge back, now applied to the To-Do's real history, with the true branch names and the five merges in the order they happened.

The whole app was born from five loops

Five branches, five merges into main: the To-Do built loop by loop



Read the figure once and it becomes a reference: it is the project's real history in the same language as the map you already know.

Each Feature Lit a Spotlight

The five laps were identical in form and different in focus. Each chapter kept the whole cycle running, but lit a spotlight on one step, so you could see it up close without the others stealing the scene. Set side by side, the five passages are the cycle seen from every angle.

Creating and listing tasks lit the spotlight on **specify**. Here you learned that what comes out of the first step is a draft, not the finished spec: the tool organizes your sentence into scenarios, requirements and criteria, gets the skeleton right and guesses at the flesh. The content stays yours to review, fix and complete, and it was that review, not the generated text, that shaped the

feature. Specifying well was less about writing a lot and more about deciding what the feature is, and what it explicitly is not, before a single line of code exists.

Completing and reopening a task lit the spotlight on `clarify`. The lesson was simple and sharp: a question you don't answer does not disappear, it just changes place. Leave an ambiguity open in the spec and the one who will decide for you is the agent, at implementation time, picking some path, plausible, maybe wrong, and moving on without warning. `clarify` exists to bring that decision back to you, and early, while it costs an answer, instead of discovering it down the line as an assumption already buried in the code.

Filtering tasks by state lit the spotlight on `plan` and `checklist`. It was where you saw intent become design before becoming code: which layers, which flow, which success criteria decide whether the thing is ready. And you saw the quality gate, `checklist`, check whether the spec and the plan were complete and clear enough to move forward, instead of discovering a hole only down the line, when fixing it already costs dearly. Planning and validating was the step that turned a good intention into a route the agent could walk without guessing.

Deleting a task lit the spotlight on `tasks` and `analyze`. `tasks` broke the plan into small, ordered, verifiable steps, each a unit the agent executes and confirms done before moving to the next. `analyze` crossed spec, plan and tasks looking for conflict, gap and excess, and taught fixing at the source: the defect is fixed in the artifact where the truth is born, the spec or the plan, so the fix comes down the whole chain, not in an isolated patch at the end, in the task or the code.

Editing a task lit the spotlight on `implement`. It was the step that turns tasks into code that passes the tests, on the beat of the red-green cycle, reading the versioned artifacts instead of the chat

history. And it was where a real defect appeared at build time, and the right reflex was not to open the code and patch the condition in a hurry, but to go back to the spec, fix at the source and let the fix come down guided by a test that first failed. Code as the consequence of a process, the exact opposite of trying again until it works. That episode is told in full in **Chapter 10.5**, in the order it happened and with the text of each artifact before and after: it is the only one of the five laps the book records from requirement to second iteration, and it is worth rereading if the mechanics of fixing at the source still feel abstract.

Together, They Covered the Whole Cycle

The spotlight, though, was never the whole cycle: on every branch, without exception, the feature walked the complete lap, and only then the merge. What changed from chapter to chapter was where the light fell, not the path walked. That is why the sum of the five passages recomposes the whole cycle in your head, and in the best possible way: through repetition. You did not see the cycle once. You saw it five times, from five angles, on five different problems. It is repetition, and not the isolated command, that gets internalized and becomes reflex.

One Constitution Governed the Five Loops

There is one piece that appeared in the diagram above everything and belongs to none of the five laps: the constitution. You wrote it a single time, back in Chapter 4, before the first feature existed. And you never rewrote it. The five features were born, walked the cycle and were integrated under the same rules of architecture, layers, error handling, quality principles. Five loops, one constitution.

That places it in a different spot from the rest. The constitution is not the first step of the cycle, which would repeat with each feature. It is **above** the cycle, governing every lap without being part of any. While `specify`, `plan` and `implement` each ran five times, it stood still at the top, stable, consulted every time `plan` needed to check whether the feature respected the fixed principles. Governance is the frame within which the repetition happens, and a frame does not run in a loop.

Going deeper: governance is not a step of the loop. It is easy to confuse the constitution with a first step of the cycle, since it comes before everything. The difference is in the frequency. A loop step runs once per feature: five features, five `specify` s. The constitution ran zero times per feature, because it was written once for all of them. It is the level of the general, lasting rules, above the day-to-day decisions. Touching it is an amendment, a rare and versioned event, not a step of the flow.

The Cycle Is Yours Now

Set the To-Do aside for a moment and look only at what is left in your hand. It is not a task app. It is a cycle: branch off `main`, write the spec of what you want, let `clarify` resolve the ambiguities, plan, validate with `checklist`, split into tasks, check consistency with `analyze`, implement under tests and integrate back into `main` through the merge. That path has nothing specific about personal tasks. It serves any software.

That is what makes the cycle transferable. Tomorrow, in a project of yours that has nothing to do with lists, you branch off, write the first spec and run the same lap. The start is always the

same: a branch and a `specify`. You already know where to begin because you have done it five times.

And in steering the agent, two reflexes you saw along the way make the biggest difference. The first: ask it to ask questions. When something in the spec admits more than one reading, force the ambiguity to surface in `clarify`, instead of letting it become a silent assumption in the code. The second: demand grounding. Do not accept a plan decision or a fix without understanding why it is this way and not another. The agent speeds up the work, but the one who governs the process is you. The cycle is what keeps that governance possible. Governing the agent is also deciding what it sees and when, and that is the subject of *Context Engineering* (2026, <https://books.kodel.com.br/en/books/context-engineering/>), the third volume in this trilogy: what the agent sees right now, in the window of this one call, and at what cost. The two reflexes above are already enough to run the cycle with discipline; the third volume only turns into method what is a habit here.

Going deeper: how to demand grounding in practice.

Demanding grounding is asking each decision to point to its origin, without gratuitous suspicion. Faced with an architecture choice, ask which constitution principle supports it. Faced with a fix, ask in which artifact the defect was born and why the fix goes there, and not in the code. When the answer is "because the spec says" or "because principle X requires it," the process is healthy. When the answer is "because it worked," you are back to improvisation.

What If I Did This Without SDD?

It is worth facing head-on the objection that may be in your head now. If I simply asked an agent "build a task app with create, complete, filter, delete and edit," wouldn't it spit out something similar in a fraction of the time? No branches, no specs, no seven steps?

For a To-Do, probably yes. A task app is the kind of thing agents do with their hands tied behind their back, because it is ubiquitous in the data they were trained on. Asking for the whole app at once, off the cuff, would come close to the result, and there is no point pretending otherwise. But that is exactly where the trap of the example lies. The To-Do is trivial on purpose, so that the mechanics of the process would never stay hidden behind the difficulty of the problem. It was never the proof of SDD's value.

The proof shows up when the project stops being trivial. In real life, software is not a task list: it grows, gains rules that contradict each other, lasts months, passes from hand to hand. That is where improvisation starts to charge. Vibe-coding loses track of why each decision was made, loses the governance that keeps the parts coherent and loses consistency as the context overflows. The cycle you learned was not invented for the To-Do. It was invented for the day the To-Do becomes a real system, and improvisation no longer copes.

The To-Do Was the Vehicle; the Process Was the Protagonist

Go back now to the uncomfortable question everything started with, back in Chapter 0: if artificial intelligence already builds on its own, why waste time specifying? The finished app is the answer, and it is in the git history, not in the rhetoric: five times

you described what you wanted before asking for the code, and each line came out of a clear spec, a decided plan, ordered tasks and tests guiding.

That is why the app mattered so little and the path mattered so much. The To-Do was the vehicle, and you can discard it without losing anything. What you don't discard is the cycle, because it is the same for any software you build afterward. The protagonist was never the task app. It was Spec-Driven Development, and the To-Do only served so that you could watch it work, end to end, five times. Each of those five slices landed in an architecture that never changed shape, and that is what *FOCUS Architecture* (2026, <https://books.kodel.com.br/en/books/focus/>), the second volume, digs into: where each rule lives and why dependencies point inward. This book closes on its own what it set out to show; the second volume answers a different question, it does not complete this one.

The material ends here. Not because there is nothing more to say on the subject, but because what it set out to show is already shown: the whole cycle, in action, from nothing to an integrated app. We did not open a final chapter comparing tools or weighing alternatives; that decision was deliberate, and the place to compare tools has already passed. What remains is simpler and more yours: a process you saw work and that you can now run on your own. The next branch is yours.

Canonical Glossary

The single source for the technical terms in this material. Each term appears here once, with the translation and the plain explanation used in the text. The chapters explain each term on its first appearance and reuse this spelling without renaming it. When you introduce a new term, record it here.

Convention: the "Ch." column shows which chapter the term entered the material in.

Terms

Term	Ch.	Definition
SDD (Spec Driven Development)	0	Development guided by a specification: describing what you want before asking for the code.
Vibe-coding	0	Programming on improvisation, on vibes, asking the AI with no method and no clear specification, hoping the result will do.
Token	0	The unit of text (a chunk of a word) that the AI reads, generates, and bills for.

Context	0	The window of text resent to the AI on every interaction, where everything it knows about the task has to fit, since it keeps no memory of its own between requests.
Waterfall	0	A model that specifies everything up front and only then builds, in sequential phases that flow down like a waterfall.
Agile	0	An approach that delivers in short cycles, with frequent feedback, adjusting course at each step instead of betting everything on a fixed plan.
FOCUS Architecture	0b	The second volume in the trilogy, on where each rule lives and why dependencies point inward. The acronym opens up into Feature-Oriented, Clean, Unidirectional and Scalable.
Context Engineering	0b	The third volume in the trilogy, on what the agent sees right now, in this call's window, and at what cost.
Cross-reference	0b	The way this book cites a sibling volume: the name, the link, and what you need to

		know summed up in the sentence itself, with no outside reading required.
Scrum	1	An agile approach based on short cycles (sprints) with defined roles and events and frequent feedback.
Iterative/incremental	1	Building in short cycles, delivering and adjusting little by little, instead of all at once at the end.
Agile Manifesto	1	A 2001 document that set four values for developing software, with a focus on people, working software, collaboration, and responding to change.
Sprint	1	A short, fixed-length cycle (usually 1 to 4 weeks) at the end of which something is ready to show and evaluate.
Work in progress / WIP	1	What has been started and not yet finished; limiting it keeps you from starting a lot and finishing little.
Continuous integration	1	Merging and testing everyone's work frequently, instead of waiting for the end, so that errors surface early.

TDD (Test-Driven Development)	1	Writing the test before the code, so the code is born already proving it does what it should.
Extreme programming (XP)	1	Kent Beck's agile approach that takes good practices to the extreme: testing first, integrating constantly, and reviewing code continuously.
Kanban	1	A method that visualizes work on a board and limits work in progress to improve the delivery flow.
The specify → plan → tasks → implement cycle	1	This material's vocabulary for the stages of SDD: specify, plan, break into tasks, and build.
Requirement	2	A testable statement of what the solution needs to do or respect.
Scope	2	The boundary of what is in and what is out of a solution.
MVP (minimum viable product)	2	The smallest version of the solution that already solves the core problem end to end and can be put in someone's hands.
Non-goal	2	What you explicitly declare to be off target, to contain expectations and keep the AI

		from inventing beyond what was asked.
Acceptance criterion	2	What counts as done and correct; the verifiable condition that decides whether a requirement was met. (Synonym cited once: acceptance criteria.)
Scenario / user story	2	A short description of a usage situation, from the point of view of the person using it, of what happens and the expected result.
Edge case	2	A rare or extreme situation (empty, limit, error) that the solution still has to handle well.
Assumption / premise	2	Something taken as true without being guaranteed, which, if it changes, changes the solution.
Ambiguity	2	A passage that allows more than one reasonable interpretation; what a good spec reduces.
Living artifact	2	A document made to be revised and rerun cheaply, not frozen like a contract.

Functional specification	2b	The one that describes observable behavior, in the vocabulary of the problem; it stays true when the technology changes.
Technical specification	2b	The one that describes construction, in the vocabulary of the solution: layers, interfaces, data structures. It dies along with the stack choice, and its place is the <code>plan</code> .
EARS (Easy Approach to Requirements Syntax)	2b	A grammar born in aeronautical engineering that forces every requirement into one of five shapes (ubiquitous, event-driven, state-driven, unwanted behavior, and optional), so the triggering condition is never left implicit.
MUST / MUST NOT / SHOULD	2b	The vocabulary of obligation inherited from the RFCs: MUST is what the system has to do, MUST NOT what it may never do, SHOULD is a recommendation.
Given/When/Then	2b	A scenario format in three parts: the state of the world before, the single gesture that triggers the behavior, and what became true afterwards.

BDD (Behaviour-Driven Development)	2b	A practice formulated by Dan North out of teaching TDD, which writes behavior in sentences a non-programmer can disagree with and a test tool can run.
Design by Contract	2b	Bertrand Meyer's idea of treating every operation as a contract between caller and executor, with a precondition, a postcondition, and an invariant.
Precondition	2b	What has to be true for the operation to be callable; the caller's responsibility.
Postcondition	2b	What the operation guarantees will be true when it finishes well; the executor's responsibility.
Invariant	2b	What is true before and after the operation, always, and which it has no license to break.
ATDD (Acceptance Test-Driven Development)	2b	Writing the acceptance test before building, together with whoever asked for the feature, and using it as the definition of done.
Capability (the unit of a spec)	2c	What the user can exercise from start to finish and you

		can review whole before approving; the measure of how much fits in one specification.
Cut by layer	2c	Splitting a big feature into one spec for the database, another for the logic, another for the screen. Each part passes the size test and none of them delivers any capability.
Cut by complete path	2c	Splitting by pulling out of the whole capability the leanest version that still crosses every layer, and fattening it up in the specs that follow.
CLI (command-line interface)	3	A way of driving a program by typing instructions in the terminal, instead of clicking buttons.
constitution	3	In spec-kit, the set of principles that govern the project; the first step of the flow, before specifying.
change / change proposal (OpenSpec)	3	OpenSpec's unit of work: a package that describes a change (the why, specs, tasks) before applying it.
Spec delta	3	The part that changes in a proposal, merged into the living specification when the

		change is archived.
Scaffolding	3	The initial structure of files and folders that a tool creates when it initializes a project.
Agent persona (BMAD)	3	A specialized role (analyst, architect, developer) that the AI agent takes on in each phase.
PRD (product requirements document)	3	A document that states what you want to build and why, before discussing architecture.
Story / epic	3	Epic: a large block of work; story: a small, implementable slice of it.
Stack	3	The project's set of technologies: language, database, framework. A choice for plan , not for the spec.
Governance	4	The level of the general, lasting rules that sit above the day-to-day decisions and define what is acceptable across all of them.
Layered architecture	4	Organizing the code into bands with distinct responsibilities (view, state/presentation, use cases, repositories) instead of everything mixed together.

Dependency direction	4	The rule for who is allowed to know whom across the layers: dependencies point inward, and the inner layers ignore the outer ones.
State/presentation layer	4	The band that holds the screen's state and reflects what it should show, between the view and the use cases.
Use case	4	The unit that runs one business rule of the app (for example, "complete a task").
Repository	4	The object that isolates data access: only it talks to the database or the network, and no one else.
Dependency inversion / injection	4	Instead of one layer creating another, it receives what it needs ready-made, which makes it testable and swappable.
Clean Architecture	4	A design that puts the business rule at the center and frameworks and details at the edge, with dependencies pointing inward.
SOLID	4	Five object-oriented design principles that keep classes and modules cohesive,

		decoupled, and easy to change.
Error handling as a value (Result)	4	Operations that can fail return an explicit success-or-failure result, instead of throwing an exception; the caller is forced to handle it.
Amendment / semantic versioning	4	The controlled change of the constitution, numbered by MAJOR.MINOR.PATCH according to its impact, always with a rationale.
constitution check	4	The check, during <code>plan</code> , that the feature's plan respects the principles set in the constitution.
branch (feature branch)	5	An isolated line of work in git where a whole feature is developed, starting from <code>main</code> .
merge	5	Joining the work of a branch back into <code>main</code> , integrating the feature into the project.
main	5	The repository's main, stable line, where each feature starts from and returns to.
unit of work	5	The feature as the thing that is born, runs through the entire cycle, and is integrated; the granularity of the loop.

core loop	5	The minimal path of the cycle, <code>specify → plan → tasks → implement</code> , without the three optional checks.
clarify	5	The stage that asks what stayed ambiguous in the spec and records the answers in it before planning.
checklist (quality gate)	5	The stage that validates whether the spec and the plan are complete and clear before they become tasks.
analyze (consistency check)	5	The stage that verifies consistency among spec, plan, and tasks before building.
manual validation	5	The check made with the app open in front of you, item by item, after the automatic steps have passed; the only one that catches what was consistent and wrong.
artifact (of the flow)	5	The document that each stage writes and the next one reads (spec, plan, tasks, code); what makes the context disposable.
agent-context-update (<code>/speckit-agent-context-update</code>)	5	A maintenance command that runs as an automatic hook after <code>plan</code> (and <code>specify</code>), updating the agent's context

		file to point at the most recent plan.
backlog	6	The queue of features the system is still going to get, named and in priority order, waiting their turn while you work on one at a time (in practice, a file like <code>rascunho.md</code>).
Anti-pattern (of specification)	6b	A way of going wrong that repeats, with a recognizable symptom, a predictable cost, and a known fix.
Rule inherited by reference	6b	The anti-pattern where the requirement says to apply "the same criterion" from somewhere else instead of saying what the rule is.
Undeclared initial state	6b	The anti-pattern where the spec fixes every possible option and forgets to say which one holds when the person arrives.
A verb with two owners	6b	The anti-pattern where two specs use the same expression for different behaviors, each one right within itself.
A repeated edge with no answer	6b	The anti-pattern where the spec never says what happens when the operation is repeated

		or lands on something that is no longer there.
Scope that grows out of convenience	6b	The anti-pattern where the spec absorbs the near relative of what was asked for because the addition is cheap right now.
Fixing at the end of the chain	6b	An anti-pattern of reflex, not of wording: the defect gets patched where it showed up and the spec goes on saying the wrong thing.
assumption (by the agent)	7	The decision the agent makes on its own, during implement , when an ambiguity in the spec was left open; what clarify exists to prevent, by bringing the decision to you and early.
idempotency	7	The property of an action that, applied once or many times to the same state, always gives the same result (completing what is already completed changes nothing and is not an error).
tasks (executable tasks)	9	The ordered list of small, verifiable steps that tasks writes from the plan; the artifact between plan and the

		code (distinct from the verb/stage of the same name).
decomposition (into tasks)	9	The act of breaking the plan into small, ordered, checkable tasks, each one a step that an agent runs and confirms as done.
consistency report	9	The output of <code>analyze</code> : a table that lists each finding (with identifier, category, severity, location, and recommendation) from cross-checking spec, plan, and tasks.
artifact conflict	9	An <code>analyze</code> finding where two artifacts, correct on their own, assert things that do not fit together (alongside gap and excess).
fix at the source	9	Fixing a conflict in the artifact where the truth originates (the spec or the plan), not in the task or the code, so the fix flows down the whole chain.
implement (execution)	10	The core-loop stage that runs the tasks in <code>tasks.md</code> and produces the code that passes the tests, reading the versioned artifacts (spec, plan, tasks) instead of the chat history.

red-green cycle	10	The beat of TDD inside <code>implement</code> : write a test that fails (red), write the minimum code that makes it pass (green), and refactor with the green suite as a safety net.
Precision / decision / boundary correction	10b	The three sizes of change in a spec already written: the rule was right and the sentence was loose; the rule was the wrong one; or what turned up does not belong to this spec.

Rules

- A concept **MUST** appear under a single name throughout the material (Principle IX).
 - Terms already recorded here **MUST** be reused without renaming in later chapters.
 - When you introduce a new term in a chapter, explain it on its first appearance and add the corresponding row to this table.
-