J.C. KÖDEL

SPEC DRIVEN DEVELOPMENT

FROM VIBE CODING TO SOFTWARE ENGINEERING

BETTER SPECIFICATIONS. BETTER CODE. FEWER TOKENS.

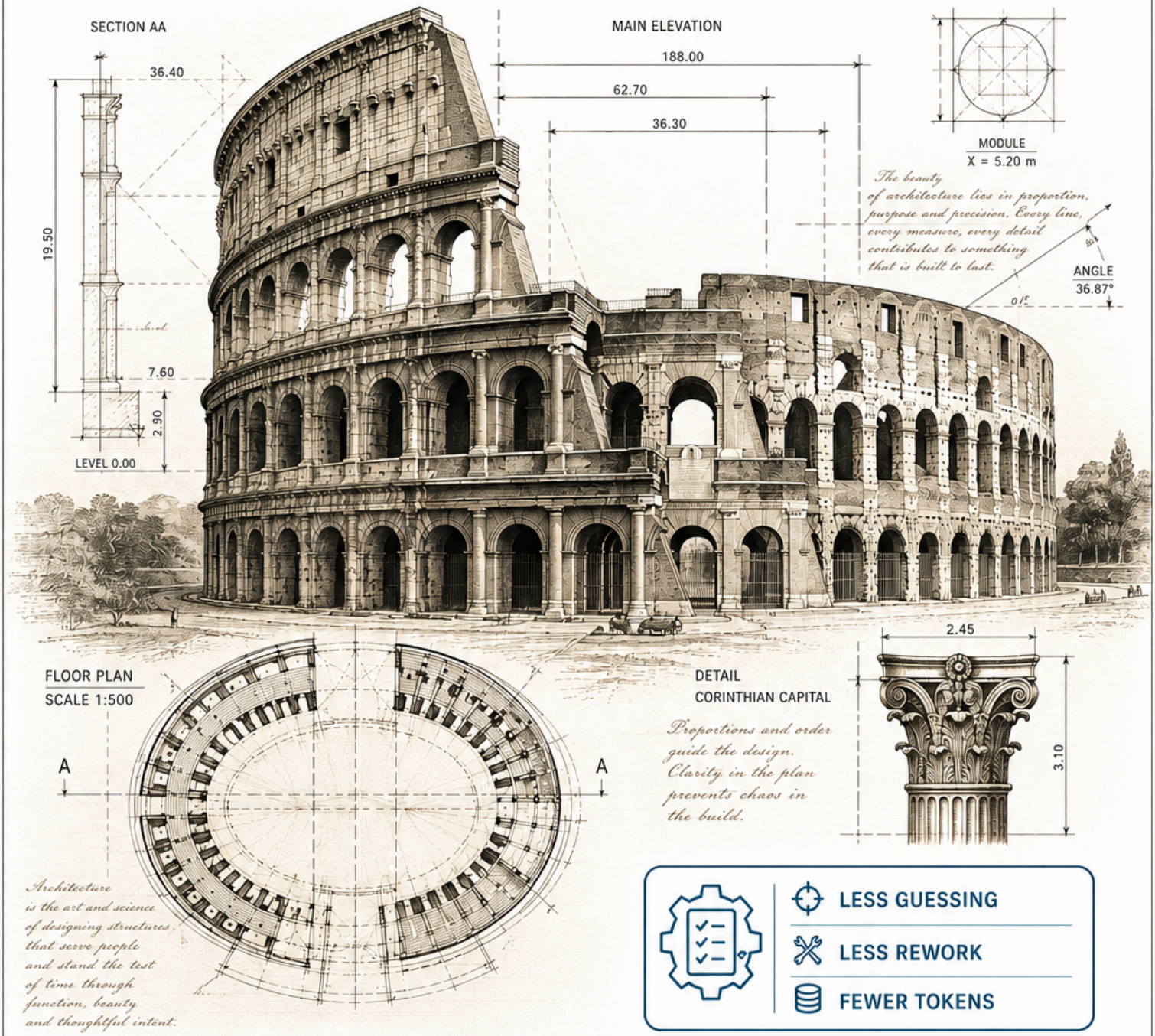

Table of Contents

This is a free sample. Get the complete edition to read every chapter.

0 - Why SDD Is Essential in the AI Era

You open your editor, describe what you need in a single sentence, and seconds later an artificial intelligence hands back code that compiles, runs, and even looks well made. What would have been science fiction a few years ago is now routine. It also brings 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. Pretending that AI is weak would be dishonest, because it is not. For a large share of everyday tasks, it writes good code, quickly and cheaply. But the right question is not "does AI program better than I do?" There is a more useful one: what separates someone who uses these tools to build solid things from someone who merely pastes back answers they do not understand? Anyone who just improvises loose requests to an AI, with no method and no clear description of what they want, is doing what people call *vibe-coding*: programming by feel, on vibes, 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 your own reasoning.

The Thesis: Knowledge and Specification Are Leverage

Before any "how", we need the "why", and it fits into a single idea, the thread that runs 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 does not replace your strength. It amplifies the strength you already have. Apply no force at all and you lift nothing, however good the lever is. AI works the same way. It amplifies anyone who hands it a clear statement of the problem, and it exposes anyone who only tosses out vague phrases. For someone who understands what they are building and can specify it, that is, say precisely what they want and why, AI is a multiplier: it produces

drafts in seconds and removes the tedium of repetitive code. For someone who neither understands nor describes, it becomes a factory of code that looks right and that no one can judge.

Notice the inversion. Common sense says: “if the AI does it, I do not need to get involved.” This material argues the opposite: precisely because the AI does it, specifying well matters more. Once producing code stops being the bottleneck, the value shifts to everything typing alone never solved: knowing what to build, judging whether it is right, choosing between paths, and answering for the result. The tool became powerful. The lever, your understanding turned into a specification, is what decides how far that power carries you.

The Three Arguments: Evaluate, 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 stays the lever even when the AI does the heavy lifting.

First, someone has to evaluate what the machine produced. AI generates plausible code, and plausible is a dangerous word. Almost everything it writes is correct. The trouble lives in the minority: the snippet that compiles, passes the obvious test, and fails silently in some rare case nobody thought to check, like a permission bug where one user sees data that belongs to someone else. Who notices a defect that subtle? Only someone who knows what the code was supposed to do, and that knowledge comes from having specified the expected behavior beforehand. Without a clear specification in your head or on paper, evaluating turns into hoping the AI got it right.

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

Third, responsibility cannot be delegated. When the system goes live, the authorship is yours. If data leaks or the cloud bill explodes, there is no “the AI wrote it” to fall back on. And you cannot answer for something you do not 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 heap of improvised requests.

Three arguments, one thesis: evaluating, deciding, and answering are exactly what the AI does not do for you, and exactly what a specification helps you master.

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 them by status, and updates when something changes. In seconds a huge, impressive answer arrives. The structure is there, the names make sense, much of it simply works. That is the **70%**: the predictable work that has shown up thousands of times across thousands of similar projects. AI is extraordinary at that 70%, and it is great that it is. That is your own time landing back in your pocket.

Then the **30%** begin. The list works with ten items and chokes on ten thousand. The filter ignores a situation that exists only in your product. The real-time update works online and vanishes at the first dead zone in signal. None of that 30% is about typing more code. It is about judgment: spotting 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 cannot do the 30% also cannot see that 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. No one hands bricks to a bricklayer and says start. First comes the blueprint: where the walls go, how many rooms, where the plumbing runs. The blueprint is the specification; the building is the code. With the blueprint in hand, you can check whether a wall ended up in the right place. Without it, you only discover the mistake once the wall is already standing. SDD is drawing the blueprint before you put up the building.

Why is this essential now? Because the AI era made the building cheap and the missing blueprint expensive. Without a specification, the improvisation of vibe-coding produces two concrete problems. The first is costly, unproductive prompts: you describe it badly, get back something crooked, and describe it again, wrestling with the machine far longer than

thinking it through calmly would have taken. The second is undecipherable code: the AI delivers a lot, fast, and you pile up a system nobody understands or can maintain. The more the 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 gets expensive. The AI processes text in *tokens* (chunks of words, the unit it reads, generates, and bills 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 inside the *context*: the window of text that travels with every 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 is not the chat, it is the specification: a short, stable, organized document. The AI works against that clear contract instead of reassembling your intent from a long conversation. Fewer round trips, less reprocessed text, fewer tokens spent to reach the same place. SDD is the discipline that keeps the helm in your hand.

SDD Is Not Exactly New

A dose of honesty against the hype: specifying before building was not invented yesterday. Decades ago, the dominant way to make software was *waterfall*, a model that wrote the entire specification at the start and only then built, in phases that flowed down in sequence like a waterfall. SDD inherits the good part of it: thinking about the problem before rushing to code.

Pure waterfall, though, failed, and for a clear reason. It bet that you could get the entire specification right in one shot, at the start, and that nothing would change. Reality always changes, and backtracking was expensive: the building was already standing on the wrong blueprint. It was to fix that rigidity that *agile* and *scrum* appeared, approaches that deliver in short cycles, with frequent feedback, adjusting course at every step instead of betting everything on a fixed plan.

And what does the AI era change in that equation? It knocks down the cost that sank waterfall. Rewriting became cheap. If the specification has to change, the AI redoes the code in minutes. SDD today is not rigid waterfall returning: it is specifying with intent and still iterating cheaply, joining the clarity of the blueprint with the lightness of short cycles. The best of both worlds became viable.

> If you come from the agile world: SDD does not replace your sprint. The specification becomes a living artifact, revised every cycle, and the AI is what makes rewriting cheap enough to be worth it.

The Next Step

If the thesis made sense, you already have the essentials: in the AI era, the bottleneck is no longer producing code, but knowing what to ask for and evaluating what comes back. Understanding and specifying is the lever that amplifies everything the machine delivers. That is what separates someone who builds solid things from someone who just pastes back answers.

Before giving this method a practical shape, an honest question is worth asking: isn't specifying before building the old way of making software, the one the world spent years trying to leave behind? In the next chapter you will see where SDD comes from. The idea of specifying first has proven its worth for decades, and every methodology, from waterfall to agile, left a lesson that SDD inherits. What changed is the timing: AI puts that proven idea back in play, now without the cost that used to sink it.

> A confession before we go on: this material was written using SDD. Every chapter began as a specification before the first sentence, the way to keep it coherent, not forget details, and check at each step whether it still made sense. What you are reading is human text, written by me, grounded in facts: the specification was the scaffold, not the author. SDD is not vibe-writing, just as it is not vibe-coding. It is the opposite: the blueprint in hand before you raise the wall, whether the build is software or a text.

---

1 - Fundamentals: Where SDD Comes From

In the previous chapter you walked away with one idea in hand: specifying before you ask for the code is what separates someone who builds solid things from someone who just pastes back answers they do not 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 you build exactly the old way of making software, that heavy, bureaucratic way the world spent thirty years trying to leave behind?

If you have ever worked on a team, you know the fatigue. A document no one reads, a meeting to approve a meeting, a giant plan that reality runs over in the first week. Anyone who lived through that learned, and rightly so, to distrust whoever shows up preaching "let's plan everything first." The suspicion is legitimate, and this chapter will not pretend it does not exist.

What it will do is separate two things that usually come glued together: the instinct to think before building, which always had value, and the cost of changing late, which is what actually sank the old model. To do that we need to go back a little in time and look, without rushing, at how software was made before you arrived. Not out of nostalgia. Quite the opposite: 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

Picture building a house. No one hands bricks to a bricklayer and says start. First comes the blueprint: where the walls go, how many rooms, where the plumbing runs. Only then does the structure go up, and only once it is finished does the painting come. Each stage begins when the previous one ends, and going back is expensive: knocking down a wall that is already standing 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 down 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 lives 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 bit of trivia 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 very author who described it.

Notice what actually happened. The waterfall instinct was right: thinking about the problem before you start building keeps you from putting up 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 when they see something finished; the market moves; a forgotten detail shows up at the end. And changing things at the end, with the structure already raised on the wrong blueprint, was expensive. Estimates from the time spoke of fixing an error late costing dozens of times more than fixing it early.[2] The problem with waterfall, then, was not 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, what if you put up one room first, live in it, see what bothers you, and adjust before moving on? That is the idea behind *iterative/incremental* development: building in short cycles, delivering and adjusting little by little, instead of everything at once at the end. Each cycle produces something usable, gets feedback, and corrects the course of the next cycle, while correcting is still cheap.

It may look like a recent invention, but it is not. There are records of iterative development as far back as 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 met in Snowbird, in the state of Utah, and wrote the *Agile Manifesto*: a short document that fixed four values for developing software.[4] As stated in 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 inverts it: change is expected, and the value lies in responding to it fast. It is not about abandoning the plan, it is about no longer pretending the plan would be right from start to finish. *Agile* is exactly that: delivering in short cycles, with frequent feedback, adjusting course 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 short, intense burst, a quick 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 previous one. Scrum was presented publicly by Ken Schwaber and Jeff Sutherland at the OOPSLA conference in 1995.[6] The name comes from earlier: a 1986 paper by Hirotaka Takeuchi and Ikujiro Nonaka, "The New New Product Development Game," which compared high-performance product teams to a rugby scrum formation, where the team moves forward together, pushing in the same direction.[7]

> For those already working with scrum: here it comes in 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 meeting 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 good engineering practices to the extreme. Beck developed it on Chrysler's C3 project, around 1996, and consolidated it in his work "Extreme Programming Explained," from 1999.[8] Two of its practices matter here. *Continuous integration*: merging and testing everyone's work frequently, instead of waiting until the end, so 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 work, with roots in Toyota's production system.[9] Kanban visualizes work on a board and limits *work in progress*, or WIP (Work In Progress): 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. It is flow, not batch.

What Each Method Taught Us

Look at the whole sequence at once and a pattern appears. It is not a pile of loose trends; it is a chain, each link responding to the weakness of the one before it. From it come three lessons that go straight into what comes next.

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 course before the detour gets expensive. It was the direct answer to waterfall's blind spot, which treated change as an accident rather than 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 all three. SDD does not 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 did no one simply join 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 tearing down finished walls, you would hold on to the blueprint tooth and nail and avoid touching it, exactly waterfall's reflex. If raising and tearing down walls were instant and almost free, you would experiment freely, adjust on 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 AI era 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 frees up the combination that did not 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, and AI does not resurrect it: it gives it new meaning**, granting the old discipline of thinking first a cost of change it never had.

Let me be direct 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 re-run as many times as needed. The blueprint stays valuable, but it stopped being a prison. We are not rescuing an old method or forcing the past back; we are using a new lever, AI, to finally keep the best of each inheritance at the same time.

Why AI Needs a Specification

It is worth closing the case for why, now with AI at the center. When producing code stops being the bottleneck, the value shifts to knowing what to ask for, and the specification is what makes that request precise and cheap. Remember how AI works: it reads everything in *tokens*, the unit of text it processes and bills for, and it keeps no memory of its own between one request and the next. Everything it needs to know has to fit inside the *context*, the window of text resent with every interaction. In the improvisation of *vibe-coding*, that context is the entire conversation, which only grows: with each request the AI rereads a larger history, burns more tokens on that rework, and loses precision as the conversation

drags on. With a specification, the reference is no longer the chat but a short, stable, organized document, against which the AI runs directly. Fewer round trips, less reprocessed text, more precision for the same cost.

There is also a second saving, less obvious and more lasting: the specification becomes the project's memory. The decisions, the rules, and the reason for each choice are recorded in one place, one that survives the end of the conversation, the change of whoever is at the keyboard, and the forgetting six months from now. A chat with the AI evaporates; a specification stays, and it is from there that the next cycle starts. That is why this material insists so much on the process: specifying well is not paperwork, it is what keeps the AI precise today and the project understandable tomorrow.

The Cycle, Now With a Name

This synthesis has a working rhythm, and it organizes into four stages that will reappear from start to finish in this material. It is worth fixing the vocabulary now, still without any tool in front of you:

- **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 support the specification.
- **tasks**: break the plan into concrete, executable steps, small enough to follow.
- **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 stage, and this 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 stages.

The Next Step

To recap the path: waterfall got it right by asking for direction and got it wrong by charging dearly for changes; agile taught us to iterate cheaply and to seek feedback early; scrum, XP, and kanban fine-tuned that search. SDD inherits all three lessons at once, and that only

became viable now because AI knocked down the cost of rewriting. The idea of specifying first is old and has already proven its worth; what is new is the timing, which finally lets that idea deliver everything it promised.

You now know where SDD comes from and why it makes sense now. What is left is to look closely at 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.

---

1. "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 rising cost of late corrections: https://en.wikipedia.org/wiki/Waterfall_model
2. "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 rising cost of late corrections: https://en.wikipedia.org/wiki/Waterfall_model
3. "Iterative and incremental development", Wikipedia, on the roots of iterative development predating 2001 (records from the 1950s and NASA's Mercury program): https://en.wikipedia.org/wiki/Iterative_and_incremental_development
4. "Agile software development", Wikipedia, on the drafting of the Agile Manifesto in 2001 in Snowbird (Utah) by seventeen signatories: https://en.wikipedia.org/wiki/Agile_software_development
5. Values cited from the primary source, "Manifesto for Agile Software Development" (2001): https://agilemanifesto.org
6. "Scrum (software development)", Wikipedia, on Ken Schwaber and Jeff Sutherland, the 1995 OOPSLA presentation, and the sprint concept: https://en.wikipedia.org/wiki/Scrum_(software_development)
7. 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
8. "Extreme programming", Wikipedia, on Kent Beck, Chrysler's C3 project (around 1996), the work "Extreme Programming Explained" (1999), and practices like TDD and continuous integration: https://en.wikipedia.org/wiki/Extreme_programming
9. "Kanban (development)", Wikipedia, on David J. Anderson, the work at Corbis (mid-2000s), the 2010 work, the roots in the Toyota Production System, and the work-in-progress (WIP) limit: https://en.wikipedia.org/wiki/Kanban_(development)