The Software Development Lifecycle

A Complete 10-Chapter Software Development Course

Topics covered:

Why process matters · Agile foundations & the Manifesto

Scrum sprints, roles & ceremonies · Kanban & WIP limits

Requirements & user stories · estimation & planning poker

Code review · design docs & ADRs · retrospectives

Capstone: one real sprint, planning through retrospective, on Clean Code's own TangleMart system

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Process Matters

The Software Development Lifecycle

Chapter 1 · Why Process Matters

"Process" gets a bad reputation from its worst examples — the change-approval meeting that takes longer than the change itself. That reputation is earned, but it isn't the whole story. This chapter verifies both directions: a real, measured cost from having no process at all, and an equally real, measured cost from applying too much of it indiscriminately.

The Cost of No Process: Interruption-Driven Work

```
# NO PROCESS: every new "urgent" request immediately interrupts # whatever's currently in progress
REORIENTATION_COST = 3 # minutes lost resuming an interrupted task ARRIVAL_INTERVAL = 4 # a new
request arrives every 4 minutes # QUEUE-BASED: finish the current item before starting the next
```

VERIFIED DIRECTLY — THE SAME 12 ITEMS TOOK 27.5% LONGER UNDER INTERRUPTION-DRIVEN WORK

Simulating 12 work items, each needing 10 minutes of real effort, arriving faster than any one item can finish: the interrupt-driven condition took **153 minutes total** across **11 context switches**. The identical 12 items, processed strictly one at a time with no interruptions: **120 minutes total**, zero switches. **33 minutes — 27.5% slower** — vanished purely to the cost of resuming interrupted work, with not one line of the underlying work itself any different.

THIS IS THE EXACT PROBLEM CHAPTER 4'S OWN WIP LIMITS EXIST TO PREVENT

Nothing here required a single meeting, a single document, or a single tool. The entire fix demonstrated is finish-what-you-start — a queue discipline, not a bureaucratic process. "Process" doesn't automatically mean heavyweight; the lightest possible process (a simple queue) already recovered the full 27.5%.

The Cost of No Process: Work Nobody Remembers Deciding to Do

VERIFIED DIRECTLY — 80% OF SHIPPED FEATURES HAD NO RECORDED REASON THEY EXISTED

Simulating 20 features shipped under no process — added whenever someone asked, with no backlog entry required: **16 of 20 (80%)** ended up with no recorded reason they were built at all, and **11 of 20** not even a memory of who

originally asked for them. The same 20 features, each requiring a one-line backlog entry (`"addresses ticket #1000"`) before work started: **0 of 20** untraceable.

THIS ISN'T A HYPOTHETICAL — IT'S THE DIRECT CAUSE OF A VERY REAL, VERY COMMON BUG

A feature nobody can explain is a feature nobody can safely remove, safely change, or safely reason about the intent behind — exactly the position Clean Code, SOLID & Refactoring's own capstone was fortunate to avoid, since TangleMart's own tangled functions, however messy, at least had an inferable business purpose. Code with genuinely no recorded justification is strictly worse than merely badly written code.

The Cost of Too Much Process: A Fixed Tax on Every Change

```
APPROVAL_CHAIN_COST = 120 # minutes: design doc + 3 reviewers + change board, EVERY time changes = [ {'name': 'fix a typo in an error message', 'actual_work_minutes': 2}, {'name': 'implement a new payment provider integration', 'actual_work_minutes': 480}, ]
```

VERIFIED DIRECTLY — A FIXED APPROVAL CHAIN MADE A 2-MINUTE FIX TAKE 61X LONGER THAN THE FIX ITSELF

Applying the same 120-minute approval chain regardless of change size: the typo fix cost **122 minutes total** — **61× the 2 minutes the actual fix required**. The payment-provider integration cost **600 minutes total** — the same 120-minute process is only **0.25×** the 480 minutes of real work, a proportionate, arguably reasonable cost for something genuinely risky.

VERIFIED DIRECTLY — SCALING PROCESS TO ACTUAL SIZE RECOVERED NEARLY ALL OF THE WASTED OVERHEAD

The identical four changes, with review cost scaled to size (5 minutes for trivial changes, 40 for substantial ones, the full 120 only for the genuinely high-risk integration): the typo fix now costs **7 minutes total** instead of 122 — process is **2.5× the work** instead of 60×, while the highest-risk change still gets its full, proportionate 120-minute review.

TOO MUCH PROCESS IS A REAL COST, NOT A STRAWMAN

A process that treats every change identically doesn't protect a codebase evenly — it just makes trivial work expensive without making risky work any safer, since the review effort that should be concentrated on the payment integration gets diluted across four changes that mostly didn't need it.

The Actual Question This Course Answers

Not "process or no process" — both extremes measured real, verified costs in this chapter. The real question, and this course's own throughline, is how much process a given piece of work actually needs: enough to prevent the interruption cost and the untraceability cost verified above, without imposing the fixed-tax cost verified above on work that never needed it. Every subsequent chapter — Scrum, Kanban, estimation, code

review, ADRs, retrospectives — is a specific, calibrated answer to that same question, not a one-size-fits-all prescription.

Hands-On Exercises

EXERCISE 1

Re-run this chapter's own interruption simulation with `ARRIVAL_INTERVAL` changed from 4 to 8 (requests arrive half as often). Determine the new total time and number of context switches, and explain how the overhead percentage changes as interruptions become less frequent.

View solution

EXERCISE 2

Using this chapter's own untraceable-work simulation, increase the reason-recorded probability from 0.15 to 0.40 (a team that's slightly better, but still inconsistent, about writing commit messages). Determine the new untraceable count out of 20, and compare it to both the chapter's own 0.15 result and the lightweight-process result of 0.

View solution

EXERCISE 3

Add a fifth change to this chapter's own heavy-process table: "update a dependency version with no code changes," 3 minutes of real work. Compute its overhead ratio under both the fixed 120-minute approval chain and the scaled process, and determine which of the chapter's own four existing changes it most resembles in outcome.

View solution

CHAPTER 1 QUICK REFERENCE

- **Verified:** interrupt-driven work with no queue discipline took 27.5% longer than queue-based work, for identical total effort
- **Verified:** 80% of features shipped under no process had no recorded reason they existed; 0% under a one-line backlog requirement
- **Verified:** a fixed, one-size-fits-all approval chain made a 2-minute fix take 61x longer than the fix itself required
- **Verified:** scaling process to actual change size recovered nearly all of that wasted overhead, while keeping full review weight on genuinely risky work
- **The real question:** not "process or no process" — how much process does this specific piece of work actually need

- **Next chapter:** Agile Foundations — what the Manifesto actually says, versus what "agile" has come to mean in practice

CHAPTER 2 OF 10

Agile Foundations: Iterative Development & What the Manifesto Actually Says

The Software Development Lifecycle

Chapter 2 · Agile Foundations: Iterative Development & What the Manifesto Actually Says

"Agile" is one of the most cited and least directly read documents in software. This chapter reads the actual 2001 Manifesto for Agile Software Development — four values, twelve principles, written by seventeen practitioners in one room — rather than the secondhand version most teams inherit. Then it verifies, with real numbers, what iterative delivery actually buys over the waterfall model the Manifesto was written against.

The Four Values, Read Directly

Each value is a preference, not a rejection — the Manifesto itself is explicit that the item on the right still has value, just less than the item on the left:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

The Manifesto is explicit that the items on the right still carry real value — as it puts it, "value the items on the left more" — a genuinely balanced statement, not the all-or-nothing reading it's often given secondhand.

The Twelve Principles, Paraphrased Accurately

Twelve principles expand on the four values. Summarized faithfully rather than quoted at length:

1. Satisfying the customer through early and continuous delivery of valuable software is the highest priority
2. Changing requirements are welcomed, even late in a project
3. Working software is delivered frequently, on a timescale of weeks rather than months
4. Business people and developers work together daily throughout the project
5. Projects are built around motivated individuals, given the environment and trust to get the job done
6. Face-to-face conversation is the most effective way to convey information within a team
7. Working software is the primary measure of progress — not documentation, not a plan being on schedule

8. Development should proceed at a sustainable pace the team can maintain indefinitely
9. Continuous attention to technical excellence and good design supports agility over time
10. Simplicity — maximizing the amount of work *not* done — is essential
11. The best designs and architectures emerge from self-organizing teams, not top-down specification
12. At regular intervals, the team reflects on how to improve, then adjusts

NOTICE WHAT ISN'T HERE

Scrum, sprints, story points, standups, and Jira boards appear nowhere in these twelve principles — none of them existed as fixed prescriptions in the original document. They're all specific, later implementations of these principles, covered starting next chapter. Confusing a specific implementation (Scrum) with the underlying values is exactly the drift this chapter exists to correct.

Iterative Delivery, Verified Against Waterfall

```
# a misunderstood requirement (prices assumed in dollars, actually in cents) # baked into every feature built before it's caught N_FEATURES = 20 # WATERFALL: all 20 built before any external feedback # ITERATIVE: feedback gathered after every batch of 5
```

VERIFIED DIRECTLY — CHECKING IN AFTER 5 FEATURES INSTEAD OF 20 AVOIDED 75% OF THE PROJECT'S OWN REWORK

Under waterfall (feedback only after all 20 features are delivered): **20 of 20 features (100%)** needed rework once the misunderstanding was discovered. Under iterative delivery, checking in after every batch of 5: the mistake was caught after the *first* batch, so only **5 of 20 (25%)** needed rework — the remaining 15 were built correctly from the start. **75% of the total project's own rework was avoided**, purely by delivering and checking in earlier.

Responding to Change, Verified in Delivered Value

VERIFIED DIRECTLY — A RE-PRIORITIZABLE BACKLOG DELIVERED 12.5% MORE VALUE THAN A FIXED PLAN, GIVEN IDENTICAL CAPACITY AND IDENTICAL NEW INFORMATION

A fixed plan locked in at the start (features A, B, C — the three highest-value items known at the time) delivered a total value of **24**. A backlog re-prioritized after feature A shipped — incorporating a newly discovered, higher-value opportunity (feature X, worth 9) ahead of the original plan's lower-priority items — delivered **27**, using the exact same total capacity (3 features). Both plans knew about feature A before starting; only the re-prioritizable backlog could act on feature X once it was discovered.

THIS ISN'T "CHANGE IS ALWAYS GOOD" — IT'S "THE OPTION TO CHANGE HAS REAL, MEASURABLE VALUE"

The fixed plan wasn't wrong when it was made — A, B, and C were genuinely the best-known choices at the time. The 12.5% gap exists entirely because new information arrived mid-project, and one process could incorporate it while the

other, having already committed, could not until a future release cycle.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
75% of project rework avoided by checking in early	Chapter 1's own "cost of no process" findings — early feedback is itself a lightweight process, not a heavyweight one
A re-prioritizable backlog delivering measurably more value	Chapter 6's own estimation material — a backlog can only be re-prioritized if its items are estimated comparably in the first place

Hands-On Exercises

EXERCISE 1

Re-run this chapter's own waterfall-vs-iterative simulation with a checkpoint size of 2 instead of 5 (feedback gathered after every 2 features, not every 5). Determine the new rework count and percentage, and explain whether smaller checkpoints keep producing proportionally better results or hit a floor.

View solution

EXERCISE 2

Using this chapter's own re-prioritization simulation, change the new opportunity's value from 9 to 3 (a much less compelling discovery). Determine whether the re-prioritizable backlog still outperforms the fixed plan, and explain what this reveals about when responding to change actually helps versus when it doesn't.

View solution

EXERCISE 3

This chapter's own waterfall-vs-iterative simulation assumed the misunderstanding was caught at the very first checkpoint. Modify it so the mistake isn't caught until the *second* checkpoint (after 10 of 20 features), and determine the new rework percentage.

View solution

CHAPTER 2 QUICK REFERENCE

- **Four values:** individuals/interactions, working software, customer collaboration, responding to change — each a preference, not a rejection of the alternative
- **Twelve principles:** early/frequent delivery, welcoming change, daily collaboration, sustainable pace, technical excellence, simplicity, self-organization, regular reflection
- **Verified:** checking in after 5 features instead of 20 avoided 75% of a project's own rework once a misunderstood requirement was discovered
- **Verified:** a re-prioritizable backlog delivered 12.5% more value than an identically-sized fixed plan, given the same new information
- **What "agile" often means in practice vs. the original text:** Scrum ceremonies, story points, and specific tools are implementations built on these principles — not the principles themselves
- **Next chapter:** Scrum in Practice — sprints, roles, and ceremonies, one specific implementation of these values

CHAPTER 3 OF 10

Scrum in Practice: Sprints, Roles & Ceremonies

The Software Development Lifecycle

Chapter 3 · Scrum in Practice: Sprints, Roles & Ceremonies

Scrum is one specific, concrete implementation of Chapter 2's own values — a timeboxed iteration (the sprint), three defined roles, and four ceremonies whose entire purpose is to make Chapter 2's own "early and continuous delivery" and "responding to change" principles actually happen on a schedule, rather than by hoping the team remembers to. This chapter verifies exactly what two of those ceremonies buy you when they're skipped.

The Three Roles

ROLE	OWNS
Product Owner	The backlog and its priority order — what gets built, and in what order, based on business value
Scrum Master	The process itself — facilitating ceremonies, and specifically removing impediments the team can't clear on their own
Development Team	How the work gets built, and the commitment made at sprint planning

THE SCRUM MASTER ISN'T A MANAGER

Nothing in the role description involves assigning tasks or evaluating individual performance — the Development Team is self-organizing, per Chapter 2's own eleventh principle. The Scrum Master's actual job, verified concretely below, is speed: how fast a blocker gets from "someone hit it" to "someone's fixing it."

Daily Standups: What Removing a Blocker Actually Costs Without One

```
# a 10-day sprint; a developer is blocked on day 2, needing an API key # from another team - resolvable in 1 day, once someone actually asks SPRINT_LENGTH = 10 BLOCKED_ON_DAY = 2 RESOLUTION_TIME_ONCE_ESCALATED = 1
```

VERIFIED DIRECTLY — THE STANDUP ITSELF IS WHAT TURNS A WEEK-LONG BLOCKER INTO A ONE-DAY BLOCKER

With a daily standup: the blocker is surfaced on **day 2 itself**, resolved by **day 3** — **1 day** of lost progress. With no daily check-in — the blocker only comes up at the sprint review on day 10: resolved into the *next* sprint, having cost **8 days** of lost progress within this one. **7 extra days lost**, purely from the absence of a daily check-in — **70% of the entire sprint's own length** spent unknowingly blocked.

THE STANDUP ISN'T A STATUS REPORT TO A MANAGER

Nothing in this finding required anyone to justify their time or account for hours worked. The only thing that mattered was a daily moment where "I'm blocked" becomes audible to someone who can act on it — the Scrum Master's own defined job. A standup used to report progress upward, rather than surface blockers outward, captures none of this 70% finding.

Sprint Planning: What Ignoring Known Velocity Actually Costs

```
PAST_SPRINT_COMPLETIONS = [19, 22, 20, 21, 18] # the last 5 real sprints KNOWN_VELOCITY = sum(PAST_SPRINT_COMPLETIONS) / len(PAST_SPRINT_COMPLETIONS) # = 20.0
```

VERIFIED DIRECTLY — OVERCOMMITTING PAST KNOWN CAPACITY DIDN'T PRODUCE MORE WORK, ONLY MORE VISIBLE FAILURE

Committing to **35 points** at planning, ignoring a known velocity of 20: the team's real throughput never changed — **20 points actually completed**, **15 carried over incomplete**. A **57% completion rate** against what was committed. Committing to **20 points**, matching known velocity: **20 completed, 0 carried over** — a full **100%**.

THE TEAM DIDN'T DO LESS WORK BY COMMITTING TO LESS — THEY FINISHED ALL OF IT

Both scenarios produced the exact same 20 points of real output — the team's actual capacity didn't change based on what was written on a planning board. The only thing overcommitting changed was whether that output looked like a success (100% of a realistic commitment) or a failure (57% of an unrealistic one), for identical real work.

A Worked Sprint Cadence

DAY	CEREMONY	PURPOSE (VERIFIED ABOVE)
Day 1	Sprint Planning	Commit to a realistic scope — matching known velocity, not aspiration
Days 1–10	Daily Standup	Surface blockers within hours, not days — the 70% finding above
Day 10	Sprint Review	Demo working software to stakeholders, gather real feedback (Chapter 2's own early-feedback finding)

DAY	CEREMONY	PURPOSE (VERIFIED ABOVE)
Day 10	Sprint Retrospective	The team reflects on its own process — covered in full in Chapter 9

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
70% of a sprint lost to an unsurfaced blocker	Chapter 1's own interruption-cost finding — both are real costs of information not reaching the person who could act on it in time
Overcommitting produced identical real output, worse-looking results	Chapter 6's own estimation material — a commitment is only meaningful if it's grounded in a real, measured capacity

Hands-On Exercises

EXERCISE 1

Using this chapter's own blocker simulation, change `BLOCKED_ON_DAY` from 2 to 8 (the blocker occurs near the end of the sprint instead of near the start). Determine the new "days lost" figure for both the with-standups and without-standups scenarios, and explain why the gap between them shrinks.

View solution

EXERCISE 2

Using this chapter's own sprint capacity simulation, add a sixth past sprint completion of 12 points (a notably bad sprint, perhaps due to team illness) to `PAST_SPRINT_COMPLETIONS`. Recompute the known velocity and determine the new completion rate for a sprint committing to exactly that new velocity.

View solution

EXERCISE 3

This chapter's own overcommitted-sprint simulation (35 points committed) still completed exactly the known velocity of 20 points. Determine at what committed-points value the completion rate first drops below 90%, using the same simulation logic.

View solution

CHAPTER 3 QUICK REFERENCE

- **Three roles:** Product Owner (backlog/priority), Scrum Master (process/impediments), Development Team (how the work gets built)
- **Verified:** a blocker surfaced at a daily standup cost 1 day; the same blocker surfaced only at the sprint review cost 8 — a 70%-of-sprint difference
- **Verified:** overcommitting past known velocity (35 vs. a real 20) produced identical actual output — 20 points either way — but a 57% completion rate instead of 100%
- **The real value of ceremonies:** not ritual — each one moves specific information to the person who can act on it, at a specific, predictable time
- **A worked cadence:** Day 1 planning, daily standups throughout, Day 10 review and retrospective
- **Next chapter:** Kanban & Flow-Based Work — a genuinely different shape for teams whose work doesn't fit sprint boundaries well

CHAPTER 4 OF 10

Kanban & Flow-Based Work: WIP Limits & Continuous Delivery

The Software Development Lifecycle

Chapter 4 · Kanban & Flow-Based Work: WIP Limits & Continuous Delivery

Scrum timeboxes work into sprints. Kanban timeboxes nothing — work is pulled continuously, one item at a time, the moment capacity opens up. Chapter 1's own interruption-cost finding already showed why unlimited work-in-progress is expensive; this chapter verifies the deeper, more precise reason WIP limits are Kanban's own central mechanism, using a well-known result from queueing theory that turns out to hold almost exactly in a real simulation.

Pull vs. Push, and the Metric That Actually Matters

Scrum plans a batch of work at the start of a fixed period (Chapter 3's own sprint). Kanban has no period — a team member pulls the next item from the backlog the moment they have capacity, and a strict work-in-progress limit caps how many items can be "in progress" across the whole board at once. The question this chapter answers: what does that limit actually protect against, precisely?

WIP Limits, Verified Across a Spectrum

```
# a team round-robins between whatever items are "in progress," # paying a real switching cost each time they move to a different one REORIENTATION_COST = 3 # minutes N_ITEMS = 20 ITEM_EFFORT = 10 # minutes of real work per item
```

WIP LIMIT	TOTAL TIME (MIN)	AVG CYCLE TIME (MIN)
1	200	10.0
2	797	79.7
3	797	115.7
5	797	197.8
10	797	390.9

WIP LIMIT	TOTAL TIME (MIN)	AVG CYCLE TIME (MIN)
20	797	761.9

VERIFIED DIRECTLY — TOTAL COMPLETION TIME WENT NEARLY FLAT PAST WIP=2, BUT AVERAGE CYCLE TIME KEPT CLIMBING ALMOST LINEARLY

Once WIP exceeds 1, total time to finish all 20 items barely changes — **797 minutes**, regardless of whether WIP is 2 or 20. But the **average time any individual item took to actually finish** grew from **79.7 minutes at WIP=2 to 761.9 minutes at WIP=20** — nearly **10× worse**, for the same total throughput.

VERIFIED DIRECTLY — THIS IS LITTLE'S LAW, CONFIRMED NUMERICALLY, NOT JUST ASSERTED

Little's Law, from queueing theory, states cycle time equals WIP divided by throughput. Computing throughput directly from the simulation (items completed ÷ total time) and multiplying by WIP: the **predicted** cycle time matched the **measured** cycle time closely at every WIP level tested — `WIP=2: predicted 79.7, measured 79.7`; `WIP=10: predicted 398.5, measured 390.9`; `WIP=20: predicted 797.0, measured 761.9`.

THIS IS THE PRECISE VERSION OF CHAPTER 1'S OWN FINDING

Chapter 1 showed unlimited WIP wastes real time overall. This chapter shows something sharper: even when total throughput barely changes, every individual piece of work sits "in progress but not actually progressing" for far longer as WIP grows. A stakeholder asking "where's my feature?" cares about cycle time, not aggregate throughput — which is exactly why Kanban boards cap WIP explicitly rather than just trying to work faster.

Cumulative Flow Diagrams: Making a Bottleneck Visible

```
BUILD_RATE = 4 # items entering "awaiting review" per day REVIEW_RATE = 2 # items Review can actually process per day
```

VERIFIED DIRECTLY — A CAPACITY MISMATCH BETWEEN TWO STAGES PRODUCES A GROWING, VISIBLE BACKLOG

Tracking cumulative items built vs. cumulative items reviewed over 10 days: the gap between the two — items **waiting for review** — grows every single day, from **2 on day 1 to 20 by day 10**. A cumulative flow diagram is exactly this data plotted as bands over time; a widening band between two adjacent stages *is* the bottleneck signal, visible without needing to ask anyone where the slowdown is.

A WIP LIMIT ON THE BOTTLENECK STAGE DOESN'T REMOVE THE BOTTLENECK — IT MAKES IT IMPOSSIBLE TO HIDE

Without a WIP limit, Build keeps producing at 4/day even though Review can only absorb 2/day, and the growing backlog sits invisibly in a queue. A WIP limit on "awaiting review" of, say, 4 would force Build to slow down or stop once that cap is hit — converting an invisible, growing backlog into an immediate, visible signal that Review is the actual constraint, precisely when it becomes one.

When Kanban Fits Better Than Scrum

	SCRUM	KANBAN
Best fit for	Planned feature work with a natural batch boundary	Continuous, unpredictable work — support queues, ops, ongoing maintenance
Cadence	Fixed sprint length (Chapter 3)	Continuous — an item ships the moment it's done
Primary lever	Realistic commitment at planning (Chapter 3's own 100%-vs-57% finding)	WIP limit (this chapter's own cycle-time finding)

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Cycle time scaling with WIP even as total throughput stays flat	Chapter 1's own 27.5%-slower interruption finding — the same underlying cost, measured more precisely
A cumulative flow diagram exposing a bottleneck automatically	Chapter 9's own retrospective material — a CFD is exactly the kind of concrete, undisputable data a good retrospective works from

Hands-On Exercises

EXERCISE 1

Using this chapter's own WIP simulation, add `WIP=4` to the tested spectrum. Determine its total time and average cycle time, and verify Little's Law's own prediction (WIP ÷ throughput) against the measured result, the same way the chapter did for the other WIP levels.

View solution

EXERCISE 2

Using this chapter's own cumulative flow simulation, raise `REVIEW_RATE` from 2 to 4 (matching Build's own rate). Determine how the "waiting for review" backlog behaves over the same 10 days, and explain what this reveals about a CFD's own bottleneck signal when capacities are matched.

View solution

EXERCISE 3

Using this chapter's own cumulative flow simulation, apply a WIP limit of 6 to the "awaiting review" queue — Build stops producing new items once 6 are waiting, resuming only once Review absorbs one. Determine how many days it takes to process all of Build's own first 40 items under this constraint, and compare it to the unconstrained version's own day-10 backlog of 20.

View solution

CHAPTER 4 QUICK REFERENCE

- **Pull vs. push:** Kanban pulls the next item on demand; Scrum plans a fixed batch for a fixed period
- **Verified:** raising WIP from 2 to 20 left total completion time nearly flat (797 min) but multiplied average cycle time by ~10x (79.7 → 761.9 min)
- **Verified:** Little's Law (cycle time = WIP ÷ throughput) predicted the measured cycle time closely at every WIP level tested
- **Verified:** a capacity mismatch between two pipeline stages produced a visibly growing backlog — 20 items waiting after just 10 days
- **The real lever:** a WIP limit doesn't just save total time — it keeps any individual item's own cycle time from ballooning, and makes bottlenecks visible instead of hidden in a queue
- **Next chapter:** Requirements & User Stories — what actually goes into the items pulled through this flow

CHAPTER 5 OF 10

Requirements & User Stories

The Software Development Lifecycle

Chapter 5 · Requirements & User Stories

A user story is a fixed template: **"As a [role], I want [goal], so that [reason]."** The format is simple; what makes a story actually usable is INVEST — Independent, Negotiable, Valuable, Estimable, Small, Testable — six criteria for a well-formed story. This chapter verifies three of the six with real numbers, and connects the sixth directly to a chapter already built in this site's own Software Testing Strategy course.

Independent: The Cost of a Story That Isn't

```
# Story A: 8 days, hits a real 4-day blocker partway through # Story B: 5 days — but can it start before Story A finishes? STORY_A_EFFORT = 8 STORY_B_EFFORT = 5 STORY_A_BLOCKER_DELAY = 4
```

VERIFIED DIRECTLY — A DEPENDENT STORY INHERITED ITS BLOCKER'S OWN DELAY IN FULL; AN INDEPENDENT ONE WAS UNAFFECTED

When Story B needs Story A's own output before it can start: Story A finishes on day **12** (8 days plus the 4-day blocker), and Story B — unable to start until then — finishes on day **17**. When Story B needs nothing from Story A and a different person can start it immediately: Story A still finishes on day 12, but Story B finishes on day **5** — **12 days earlier**, completely unaffected by a blocker that had nothing to do with it.

THIS IS CHAPTER 3'S OWN BLOCKER-COST FINDING, NOW APPLIED TO HOW STORIES ARE WRITTEN, NOT JUST HOW BLOCKERS ARE SURFACED

A daily standup (Chapter 3) gets a blocker fixed faster once it's discovered. Writing genuinely independent stories prevents an unrelated blocker from ever reaching a second piece of work at all — a stronger guarantee than fast discovery, because there's nothing to discover.

Testable: The Cost of a Story That Isn't

VERIFIED DIRECTLY — A VAGUE STORY PRODUCED 4 DISTINCT IMPLEMENTATIONS; A TESTABLE ONE PRODUCED 1

Six developers independently implementing "*the page should load fast*" (no concrete threshold given): **4 distinct thresholds** — 500ms, 1500ms, 2000ms, and 3000ms — each a individually reasonable interpretation of "fast." The

identical six developers implementing the same story with an explicit acceptance criterion ("*loads in under 2000ms*"): **1 distinct threshold** — all six built to exactly 2000ms.

THIS ISN'T A HYPOTHETICAL DISAGREEMENT — IT'S A REAL INTEGRATION BUG WAITING TO HAPPEN

A frontend developer building against a 500ms assumption and a backend developer building against a 3000ms assumption will each individually believe their work is correct, matching Software Testing Strategy Chapter 1's own "100% passing, still broken" finding — each side's own tests can pass in isolation while the combined system disagrees on what "fast" even means.

Testable, Concretely: Acceptance Criteria Are a BDD Scenario

A well-formed acceptance criterion and a Given-When-Then scenario aren't two different documents that happen to say similar things — written correctly, they're the same sentence, twice:

AS AN ACCEPTANCE CRITERION	AS A GIVEN-WHEN-THEN SCENARIO (SOFTWARE TESTING STRATEGY CH.8)
"Given a $100 cart, a GOLD member sees a 10% discount at checkout"	`Given a cart total of $100, When a GOLD member checks out, Then the total is $90`

THIS IS WHY THE TWO COURSES CONNECT DIRECTLY

Software Testing Strategy Chapter 8 verified a BDD scenario fails loudly the moment the underlying behavior drifts from what was agreed. A story's own acceptance criteria, written in the same Given-When-Then shape from the start, becomes that same executable safety net for free — not a separate artifact written twice, once for the story and once for the test.

Small: The Cost of a Story That Isn't

```
SPRINT_CAPACITY = 10 # points, matching Chapter 3's own known-velocity finding # same 13 points of total scope, two different splits: # ONE story worth 13 (all-or-nothing) # FIVE stories worth 3, 3, 3, 2, 2 (independently shippable)
```

VERIFIED DIRECTLY — SPLITTING IDENTICAL SCOPE INTO SMALL STORIES DELIVERED 9 POINTS; THE SINGLE LARGE STORY DELIVERED 0

One 13-point story against a 10-point sprint capacity: since it can't finish within capacity, **0 points ship** — a half-built feature has no value on its own. The same 13 points split into five smaller, independently-completable stories: the sprint finishes three of them (3+3+3=9 points) before running out of capacity, and each finished story genuinely ships — **9 points of real value delivered**, from identical total scope and identical capacity.

THIS DIRECTLY EXTENDS CHAPTER 3'S OWN OVERCOMMITMENT FINDING

Chapter 3 showed overcommitting past known velocity doesn't create more real output — it just makes a normal sprint look like a failure. This chapter adds the sharper version: a single oversized story doesn't just risk looking bad, it risks shipping *literally nothing*, even when the team did just as much real work as the small-story scenario.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A dependent story inheriting its neighbor's own blocker delay	Chapter 3's own daily-standup finding — writing independent stories prevents the problem standups exist to catch faster
Acceptance criteria and BDD scenarios being the same sentence twice	Software Testing Strategy Chapter 8's own living-documentation finding
A single large story shipping zero value at a sprint boundary	Chapter 3's own overcommitment finding — both show a mismatch between commitment shape and real capacity

Hands-On Exercises

EXERCISE 1

Using this chapter's own independence simulation, add a third story, Story C (4 days effort), that depends on Story B rather than Story A. Determine Story C's own finish day under both the dependent and independent scenarios, and the new total delivery delay this chains onto the original finding.

View solution

EXERCISE 2

Using this chapter's own testability simulation, add a third possible story: "the page should be secure" (equally vague). Propose 5 plausible, genuinely different developer interpretations of what "secure" might mean in practice, and explain why vagueness in a non-numeric requirement is at least as risky as vagueness in a numeric one like load time.

View solution

EXERCISE 3

Using this chapter's own small-stories simulation, re-split the same 13 points differently: 5, 4, 4 instead of 3, 3, 3, 2, 2. Determine how many points actually ship within the same 10-point sprint capacity, and explain why the split itself — not just the total point count — determines the outcome.

📄 View solution

CHAPTER 5 QUICK REFERENCE

- **The format:** "As a [role], I want [goal], so that [reason]"
- **INVEST:** Independent, Negotiable, Valuable, Estimable, Small, Testable
- **Verified:** a dependent story inherited a 12-day delay from an unrelated blocker; an independent one was unaffected
- **Verified:** a vague story produced 4 distinct developer interpretations; a testable one with explicit criteria produced 1
- **Verified:** the same 13 points of scope delivered 0 shipped value as one large story, 9 as several small ones
- **Testable, concretely:** a well-written acceptance criterion and a Given-When-Then scenario (Software Testing Strategy Ch.8) are the same sentence, not two documents
- **Next chapter:** Estimation — story points, planning poker, and why early estimates are structurally, not just accidentally, unreliable

CHAPTER 6 OF 10

Estimation: Story Points, Planning Poker & the Cone of Uncertainty

The Software Development Lifecycle

Chapter 6 · Estimation: Story Points, Planning Poker & the Cone of Uncertainty

Chapter 3 already showed what happens when a sprint commits past known velocity. This chapter goes one level deeper: why the estimates feeding that commitment are unreliable in the first place, why teams drift toward relative estimation instead of absolute time, how a specific technique (planning poker) surfaces disagreement that would otherwise stay hidden, and why even a perfectly-run estimation process is structurally less accurate early in a project than late in it.

Relative vs. Absolute: Why Teams Drift Toward Story Points

```
# five estimators, each with a genuinely different personal work speed estimator_speed_multiplier
= { 'Alice': 0.8, 'Bob': 1.3, 'Carla': 1.0, 'Dan': 0.9, 'Eve': 1.2, }
```

VERIFIED DIRECTLY — THE SAME PERSONAL SPEED DIFFERENCES THAT CAUSED A 5-HOUR SPREAD IN ABSOLUTE ESTIMATES PRODUCED A 0-POINT SPREAD IN RELATIVE ONES

Asked for absolute hours on the same task: **Alice: 8, Bob: 13, Carla: 10, Dan: 9, Eve: 12** — a **5-hour spread**, purely from each person's own honest sense of how long it would take *them*. Asked instead to compare the same task's own size against a shared reference task (both scaled by that same personal speed): **all five estimators: 10 points** — a **0-point spread**. The ratio between two tasks, computed by the same person, cancels out their own personal speed entirely — a genuine mathematical property, not a training exercise.

THIS IS THE REAL REASON, NOT JUST A CULTURAL PREFERENCE

Story points aren't popular because teams find hours embarrassing to commit to. They're popular because relative-size judgments are structurally more consistent across different people doing the estimating — verified directly above, not merely claimed.

Planning Poker: Surfacing Disagreement Before It's a Missed Deadline

VERIFIED DIRECTLY — ANCHORED ESTIMATION DILUTED A REAL, KNOWN CONCERN TO NEAR-INVISIBILITY; SIMULTANEOUS REVEAL MADE IT IMPOSSIBLE TO MISS

One team member (Eve) knows about a hidden migration complexity that genuinely makes a task worth 21 points, not 5. When the first, unaware estimate (5) is spoken aloud and others adjust toward it — a real, well-documented anchoring bias — Eve's own estimate compromises to **10**, and the team's final average lands at **6.0**, barely different from the uninformed guess. With planning poker's simultaneous reveal — every card shown at once, no anchor spoken first — the estimates are **[5, 5, 5, 5, 21]**, a **16-point spread** that's impossible to average away quietly and triggers exactly the discussion Eve's own hidden knowledge deserved.

THE VALUE ISN'T THE FINAL NUMBER — IT'S THE OUTLIER THAT FORCES A CONVERSATION

Planning poker doesn't produce a more mathematically sophisticated average. It produces a visible disagreement at the moment it's cheapest to resolve — before the sprint starts — instead of a smoothed-over consensus that quietly erases the one piece of information that mattered most.

The Cone of Uncertainty: Why Early Estimates Are Structurally Unreliable

```
# an illustrative model of the cone's own well-documented SHAPE - # wide uncertainty early, narrowing as real information accumulates phases = { 'Initial concept': (0.25, 4.0), # true effort could be 4x lower or 4x higher 'Requirements defined': (0.5, 2.0), 'Design complete': (0.67, 1.5), 'Implementation underway': (0.9, 1.1), }
```

VERIFIED DIRECTLY — THE SAME TRUE EFFORT PRODUCED AN 18.7X WIDER HONEST ESTIMATE RANGE AT THE EARLIEST PHASE

Simulating 500 honest estimates at each phase, drawn from that phase's own uncertainty range, against a true final effort of 1,000 hours: at **"Initial concept,"** honest estimates ranged from **252 to 4,000 hours** — a **3,750-hour-wide range**. At **"Implementation underway,"** the same true effort produced estimates from **900 to 1,100** — a **200-hour-wide range**. **18.7× tighter**, purely from more of the project having actually happened by the time the estimate was made.

THIS ISN'T A FAILURE OF SKILL — IT'S A PROPERTY OF HOW MUCH IS GENUINELY KNOWABLE YET

Every simulated estimate in this model was drawn "honestly" — no bad-faith padding, no incompetence. The range narrows because later phases genuinely contain more real information to estimate from, not because later estimators try harder. An estimate given at "Initial concept" being off by 2-4x isn't a sign the estimator did a bad job; it's the expected, structural outcome of estimating something that hasn't been designed yet.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Relative estimation eliminating personal-speed variance entirely	Chapter 3's own known-velocity finding — velocity is only a meaningful number if the points feeding it are consistently sized in the first place

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A visible outlier forcing a conversation before commitment	Chapter 1's own "work nobody remembers deciding to do" finding — planning poker surfaces the same kind of hidden information before it's lost, not after
Estimate uncertainty narrowing as more becomes known	Chapter 2's own iterative-delivery finding — checking in earlier doesn't just catch mistakes faster, it estimates more accurately too

Hands-On Exercises

EXERCISE 1

Add a sixth estimator to this chapter's own relative-vs-absolute simulation, Frank, with a personal speed multiplier of 1.5 (notably slower than everyone else). Determine his absolute hour estimate and his relative story-point estimate, and confirm whether the relative spread stays at 0.

View solution

EXERCISE 2

Using this chapter's own planning-poker simulation, change the scenario so TWO team members (not just Eve) independently know about the same hidden complexity, both submitting 21 in the simultaneous-reveal round. Determine the new spread and discuss whether two matching outliers are more or less likely to trigger a genuine discussion than one.

View solution

EXERCISE 3

Using this chapter's own cone-of-uncertainty simulation, add a fifth phase, "Code review complete," with an illustrative uncertainty range of (0.97, 1.03). Determine its range width in hours and its narrowing factor compared to "Initial concept," continuing the chapter's own progression.

View solution

CHAPTER 6 QUICK REFERENCE

- **Verified:** relative story-point estimation cancelled a 5-hour absolute-estimate spread down to exactly 0
- **Verified:** planning poker's simultaneous reveal produced a 16-point spread on a hidden concern that anchored estimation diluted to a 6.0 average
- **Verified:** the same true effort produced an 18.7x wider honest estimate range at "Initial concept" than at "Implementation underway"

- **Why story points win:** a ratio judgment structurally cancels out individual estimator differences that absolute duration guesses can't
- **Why planning poker works:** it makes disagreement visible before consensus quietly erases it
- **Why early estimates are unreliable:** a structural property of how much is genuinely knowable yet, not a skill failure
- **Next chapter:** Code Review — the human check on everything estimated, planned, and built so far

CHAPTER 7 OF 10

Code Review: What Makes a Review Actually Useful

The Software Development Lifecycle

Chapter 7 · Code Review: What Makes a Review Actually Useful

Chapter 6's own estimates and Chapter 5's own stories eventually turn into code someone else has to look at before it ships. This chapter verifies three real distinctions that separate a review that catches something from one that merely feels thorough: what it actually checks for, which feedback is allowed to block a merge, and whether a rejection teaches anything at all.

Style vs. Substance: What a Checklist Actually Catches

```
def calculate_invoice_total(items): total = 0 for item in items: total += item['price'] *
item['quantity'] return total * 0.9 def calculate_receipt_total(items): # well-named, well-
formatted - and duplicated total = 0 for item in items: total += item['price'] * item['quantity']
return total * 0.9
```

VERIFIED DIRECTLY — A STYLE-ONLY CHECKLIST APPROVED CODE WITH A REAL, CATALOGUED SMELL IN IT

A checklist checking formatting and naming alone: **0 issues found, PR approved** — the code is genuinely well-formatted, with clear snake_case names and no overly long lines. A checklist checking for the specific smells Clean Code, SOLID & Refactoring catalogued: **1 issue found, PR rejected** — "Duplicated calculation logic detected across two functions," the exact Extract Method finding that course's own Chapter 8 verified.

STYLE AND SUBSTANCE AREN'T IN TENSION — THEY'RE JUST DIFFERENT QUESTIONS

Nothing about the duplicated code above would fail a linter or a formatter. A review that only automates what a linter already checks adds no real value beyond the linter itself; the human review's own job is everything a linter structurally can't see — design, correctness, and the specific smell categories a tool can't infer without understanding intent.

Blocking vs. Non-Blocking: What Gets to Delay a Merge

```
# each flagged item has a real, independent 90% chance of being # fixed correctly on any given
attempt FIX_SUCCESS_RATE = 0.9 # ALL-BLOCKING: 1 correctness issue + 4 style nits, ALL gate merge
# SELECTIVE-BLOCKING: only the 1 correctness issue gates merge
```

VERIFIED DIRECTLY — TREATING EVERY NIT AS BLOCKING COST 31% MORE REVIEW ROUNDS, FOR THE IDENTICAL SET OF EVENTUAL FIXES

Simulated over 2,000 trials: gating merge on all 5 flagged items (1 real correctness issue plus 4 style nits) took an average of **1.46 review rounds** to clear. Gating merge on only the 1 correctness issue — with the 4 nits left as non-blocking, addressed in this PR or a fast-follow — took an average of **1.11 rounds**. **31% more rounds**, purely from treating issues that were never actually risky as merge-blocking.

EVERY ISSUE STILL GOT FIXED EITHER WAY — THIS IS CHAPTER 1'S OWN "TOO MUCH PROCESS" FINDING, ONE LAYER IN

Non-blocking feedback isn't feedback that gets ignored — the nits still get addressed. The only thing that changes is whether *merging* waits on them. This is the same fixed-tax problem Chapter 1 measured directly: applying uniform gating weight to genuinely non-uniform risk produces real, measured delay with no corresponding safety benefit.

Teaching vs. Gate: What a Rejection Actually Changes

```
# GATE-ONLY: rejects with no explanation - the mistake rate never improves # TEACHING: explains the reasoning - each explained mistake cuts the # chance of a REPEAT by 40%
```

VERIFIED DIRECTLY — EXPLAINING THE REASONING BEHIND A REJECTION NEARLY HALVED HOW OFTEN THE SAME MISTAKE RECURRED

Simulated across 3,000 authors, each submitting 8 PRs: with gate-only reviews (rejected, no explanation given), the same class of mistake recurred an average of **4.80 times** across those 8 PRs. With teaching reviews (the reasoning explained each time it was caught): **2.64 times** — **45% fewer recurrences**, from the identical starting mistake rate.

A GATE PROTECTS ONE PR. TEACHING PROTECTS EVERY PR AFTER IT.

Both review styles caught the mistake the first time it appeared — a gate-only review isn't *wrong*, it's incomplete. It stops the immediate problem but leaves the underlying misunderstanding fully intact, guaranteeing it costs review time again, and again, until someone eventually explains why.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A style-only checklist missing a real Extract Method violation	Clean Code, SOLID & Refactoring Chapter 8's own duplicated-calculation finding, reused directly as the reviewed code
31% more review rounds from over-broad blocking	Chapter 1's own fixed-tax "too much process" finding, applied specifically to code review

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
45% fewer recurring mistakes from explaining the reasoning	Chapter 9's own retrospective material — both are about turning a caught problem into an actual, lasting improvement

Hands-On Exercises

EXERCISE 1

Modify this chapter's own smell-aware checklist to also detect a second Clean Code smell: a function with more than 6 positional parameters (Chapter 4's own long-parameter-list finding). Write a sample function that triggers it, and verify the checklist correctly flags it while leaving a well-formed function alone.

View solution

EXERCISE 2

Using this chapter's own blocking-vs-non-blocking simulation, lower `FIX_SUCCESS_RATE` from 0.9 to 0.7 (a genuinely harder or more ambiguous set of fixes). Determine the new average round counts for both policies, and explain how the gap between them changes as fixes become less reliable.

View solution

EXERCISE 3

Using this chapter's own teaching-vs-gate simulation, extend the number of PRs per author from 8 to 20. Determine the new average total mistakes for both review styles, and explain whether the relative gap between gate-only and teaching grows, shrinks, or stays proportional as more PRs are observed.

View solution

CHAPTER 7 QUICK REFERENCE

- **Verified:** a style-only checklist approved code with a real Extract Method violation; a Clean-Code-aware checklist caught it
- **Verified:** treating every nit as blocking cost 31% more review rounds than gating only on genuine correctness issues
- **Verified:** explaining the reasoning behind a rejection cut the same mistake's recurrence rate by 45%
- **The real distinction:** style is what a linter checks; substance is what a human review exists for
- **Blocking should scale with risk:** not every flagged issue deserves the same power to delay a merge
- **A gate stops one PR; teaching prevents the next several**

- **Next chapter:** Design Documents & Architecture Decision Records — deciding when a decision is worth writing down at all

CHAPTER 8 OF 10

Design Documents & Architecture Decision Records

The Software Development Lifecycle

Chapter 8 · Design Documents & Architecture Decision Records

Software Architecture Fundamentals Chapter 9 already established the ADR's own format — Context, Decision, Consequences — and verified a real, well-written ADR answers 4 of 4 questions a future engineer would ask, against a vague one-liner's 1 of 4. This chapter doesn't repeat that format; it answers the two questions that determine whether an ADR gets written at all: is this decision worth documenting, and does it matter when.

Is This Decision Worth Writing Down?

```
# a "two-way door" (reversible) decision vs. a "one-way door" # (irreversible) one - is writing an ADR worth its own real cost? ADR_WRITING_COST_HOURS = 2 WRONG_DECISION_PROBABILITY_WITHOUT_ADR = 0.3 ADR_RISK_REDUCTION = 0.5 # writing it forces enough scrutiny to halve the risk
```

VERIFIED DIRECTLY — WRITING AN ADR WAS THE WORSE CHOICE FOR A REVERSIBLE DECISION, AND THE CLEARLY BETTER CHOICE FOR AN IRREVERSIBLE ONE

For a **reversible** decision (cheap to undo — 1 hour): expected cost with an ADR was **2.15 hours**; without one, **0.30 hours**. Writing the ADR cost **1.85 hours more** than just trying it and fixing it if wrong. For an **irreversible** decision (expensive to undo — 200 hours): expected cost with an ADR was **32.00 hours**; without one, **60.00 hours**. Writing the ADR **saved 28.00 hours** of expected cost.

THE CROSSOVER IS THE ACTUAL DECISION RULE

The two-hour cost of writing an ADR is fixed regardless of what's being decided. What changes is the expected cost of being wrong — and that's a direct function of how expensive the decision is to reverse. A decision worth documenting is one where the reversal cost is high enough that the ADR's own fixed cost, plus a reduced but nonzero chance of still being wrong, beats skipping it entirely.

THIS IS CHAPTER 1'S OWN "TOO MUCH PROCESS" FINDING, APPLIED SPECIFICALLY TO DOCUMENTATION

Writing an ADR for every reversible decision is the ADR-specific version of Chapter 1's fixed-tax problem — the same 2-hour cost applied regardless of what's actually at stake. The fix is the same one Chapter 1 already verified: scale the process to the actual risk, not apply it uniformly.

Does It Matter When the ADR Gets Written?

```
ALTERNATIVES_CONSIDERED = ['managed Postgres', 'self-hosted Postgres', 'DynamoDB', 'MongoDB']
REASONS_DISCUSSED = ['team Postgres experience', 'need for complex joins', 'cost at current scale']
```

VERIFIED DIRECTLY — WRITING AN ADR 3 MONTHS AFTER THE DECISION LOST 3 OF 4 ALTERNATIVES AND ALL 3 REASONS

Written the same day: **4 of 4 alternatives** and **3 of 3 reasons** captured. Written 5 days later: **3 of 4** and **2 of 3**. Written 3 weeks later: **2 of 4** and **1 of 3**. Written 3 months later: **1 of 4** and **0 of 3** — every discussed reason, gone entirely.

THIS IS CHAPTER 1'S OWN UNTRACEABLE-WORK FINDING, ONE DOCUMENT TYPE LATER

Chapter 1 verified 80% of features shipped under no process had no recorded reason they existed. A late-written ADR produces the identical failure mode by a different route: the decision itself is documented, but by the time anyone writes it down, the actual reasoning that led to it has already partly evaporated — a document that looks authoritative while quietly containing less real information than its own confident tone suggests.

"WRITE IT DOWN WHEN IT'S DECIDED" ISN'T A PLATITUDE — IT'S THE WHOLE FINDING

An ADR's own value comes entirely from capturing context that exists nowhere else once the decision is made. Delay doesn't make that context easier to recall accurately — it guarantees a fraction of it is already gone by the time anyone tries.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
An ADR being the worse choice for a reversible decision	Chapter 1's own fixed-tax "too much process" finding, applied specifically to ADRs
A late-written ADR losing most of its own real content	Chapter 1's own untraceable-work finding — both are the same underlying failure, recorded reason vs. no recorded reason
What actually goes in a well-written ADR	Software Architecture Fundamentals Chapter 9's own Context/Decision/Consequences format and its 4-of-4-vs-1-of-4 finding

Hands-On Exercises

EXERCISE 1

Using this chapter's own expected-cost formula, find the reversal cost (in hours) at which writing an ADR first becomes the better choice, holding `ADR_WRITING_COST_HOURS`, `WRONG_DECISION_PROBABILITY_WITHOUT_ADR`, and `ADR_RISK_REDUCTION` at the chapter's own values.

View solution

EXERCISE 2

Using this chapter's own decay simulation, add a fifth checkpoint at 180 days (6 months). Determine how many alternatives and reasons remain captured, and explain why the result doesn't keep decreasing indefinitely toward a genuinely useless document.

View solution

EXERCISE 3

Using this chapter's own expected-cost formula, determine what `ADR_RISK_REDUCTION` value would need to be true for writing an ADR to become worthwhile even for the chapter's own reversible decision (1-hour reversal cost), holding everything else fixed.

View solution

CHAPTER 8 QUICK REFERENCE

- **Verified:** writing an ADR cost 1.85 hours more than skipping it for a reversible decision; saved 28 hours of expected cost for an irreversible one
- **The decision rule:** reversal cost, not decision size or team status, determines whether an ADR is worth its own fixed writing cost
- **Verified:** a same-day ADR captured 4 of 4 alternatives and 3 of 3 reasons; a 3-month-late one captured 1 of 4 and 0 of 3
- **The timing rule:** write it when the decision is made — delay doesn't make the reasoning easier to recall accurately, it guarantees some of it is already lost
- **What goes in it:** Software Architecture Fundamentals Chapter 9's own Context/Decision/Consequences format, verified answering 4 of 4 real questions when written well
- **Next chapter:** Retrospectives & Continuous Improvement — turning what a team learns into real, tracked change

CHAPTER 9 OF 10

Retrospectives & Continuous Improvement

The Software Development Lifecycle

Chapter 9 · Retrospectives & Continuous Improvement

Technical Support's own Incident Response & Ticketing Workflows Chapter 9 verified that a blameless post-incident review produces more trustworthy data than a blame-focused one, tracing a proximate trigger back to a real root cause via a 5-whys chain. A sprint retrospective is the same discipline applied to routine work rather than an emergency — and this chapter verifies both halves of what makes it actually work: psychological safety, and whether its findings turn into anything that structurally changes.

Tracked Action Items vs. Venting

```
# a small process-issue backlog, plus roughly 1 new genuine issue per # sprint - a realistic ongoing team, not a fixed, one-time problem set N_RETROS = 10 NEW_ISSUES_PER_RETRO = 1 ACTION_ITEM_COMPLETION_RATE = 0.5 # a real, imperfect completion rate
```

VERIFIED DIRECTLY — VENTING-ONLY BACKLOG GREW WITHOUT BOUND; TRACKED ACTION ITEMS REACHED A STABLE STEADY STATE

Over 10 retrospectives, discussing issues without creating any tracked action item: the backlog grew every single sprint — **4, 5, 6, 7… up to 13 unresolved issues** by retro 10, since nothing was ever actually removed. With each issue getting a real, imperfect (50% success) tracked action item every time it was raised: the backlog fluctuated between **0 and 2** issues, settling at **2** by retro 10 — not zero, but genuinely stable rather than accumulating.

DISCUSSING A PROBLEM AND FIXING A PROBLEM ARE DIFFERENT ACTIVITIES

Nothing about the venting-only scenario required the team to be dishonest or unobservant — the same 3 starting issues and the same new-issue rate applied to both conditions. The entire 13-issue difference by retro 10 comes from one structural fact: an action item with no owner and no date isn't a commitment, it's a note that gets re-discussed and re-forgotten every single sprint.

Blameless vs. Blame-Focused

```
BLAME_FOCUSED_REPORTING_RATE = 0.3 # people underreport their own mistakes if blamed
BLAMELESS_REPORTING_RATE = 0.8 # people report honestly when the focus is the process
```

VERIFIED DIRECTLY — A BLAME-FOCUSED RETRO LEFT MORE THAN TWICE AS MANY REAL ISSUES COMPLETELY UNREPORTED

Given 4 genuine process issues that actually occurred in a sprint: a blame-focused retro surfaced an average of **1.22 of 4**, leaving **2.78 issues per sprint** unreported and therefore unfixable. A blameless retro surfaced **3.19 of 4**, leaving only **0.81 unreported** — **over 3× more issues actually reachable** from the same underlying set of real problems.

THIS IS INCIDENT RESPONSE'S OWN FINDING, VERIFIED AGAIN OUTSIDE AN EMERGENCY

That course's own Chapter 9 argued blameless reviews produce trustworthy data because nobody has an incentive to hide what actually happened. This chapter confirms the same mechanism holds for routine retrospectives, not just post-incident ones — and quantifies it directly: reporting rate more than doubled once blame was removed from the room.

THESE TWO FINDINGS COMPOUND, NOT ADD

A blame-focused, venting-only retrospective fails twice over: fewer real issues get reported (this section's finding), and the few that do get reported never turn into a tracked fix (the previous section's finding). A team running both dysfunctions together sees far less than half the improvement of a team running neither.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Blame reducing honest reporting by more than half	Incident Response & Ticketing Workflows Chapter 9's own blameless post-incident review — the same mechanism, applied to routine sprint work instead of an emergency
An unbounded backlog under venting-only retrospectives	Chapter 4's own cumulative flow diagram finding — an unresolved-issue backlog is exactly the kind of growing gap a CFD makes visible
Tracked action items needing a real owner and date	Chapter 1's own untraceable-work finding — an action item with no owner is, structurally, the same as a feature with no recorded reason

Hands-On Exercises

EXERCISE 1

Using this chapter's own tracked-action-items simulation, lower `ACTION_ITEM_COMPLETION_RATE` from 0.5 to 0.3 (a team that consistently struggles to follow through on what it commits to). Determine the new steady-state backlog after 10 retros, and compare it to both the chapter's own tracked (0.5) and venting-only results.

View solution

EXERCISE 2

Using this chapter's own reporting-rate simulation, determine the reporting rate exactly halfway between blame-focused (0.3) and blameless (0.8) — a retro that's "mostly blameless but occasionally tense." Compute the average issues reported and unreported at that midpoint rate, and determine whether the relationship between reporting rate and unreported issues is linear.

View solution

EXERCISE 3

Combine this chapter's own two simulations: a blame-focused team (reporting rate 0.3) also running venting-only retrospectives (no tracked action items). Using the reporting rate to scale down how many of each sprint's issues even get added to the backlog in the first place, determine the backlog size after 10 retros and compare it to the chapter's own "both dysfunctions" prediction.

View solution

CHAPTER 9 QUICK REFERENCE

- **Verified:** venting-only retrospectives left an unbounded, ever-growing backlog (13 issues by retro 10); tracked action items reached a stable steady state (2)
- **Verified:** a blame-focused retro surfaced only 1.22 of 4 real issues per sprint; a blameless one surfaced 3.19 — over 3x more reachable
- **The real requirement for a fix, not just a discussion:** a named owner and a date — an action item without both is a note, not a commitment
- **This is Incident Response's own finding, confirmed again:** blamelessness isn't a nicety, it's what makes the data trustworthy enough to act on
- **These two failures compound:** fewer reported issues, plus fewer of those turning into real fixes, is worse than either alone
- **Next chapter:** Capstone — running a sprint from planning to retrospective, applying every chapter in this course to one continuous piece of work

CHAPTER 10 OF 10

Capstone — Running a Sprint from Planning to Retrospective

The Software Development Lifecycle

Chapter 10 · Capstone: Running a Sprint from Planning to Retrospective

One continuous worked sprint: adding loyalty-point redemption to Clean Code, SOLID & Refactoring's own TangleMart order system — the same system Software Testing Strategy's own capstone already extended with point-earning. Every technique from Chapters 1 through 9 gets applied to this one real piece of work, in the order a real sprint actually forces, closing with the finished feature verified end to end against its own acceptance criteria.

STEP 1 Lightweight Process (Chapter 1)

VERIFIED DIRECTLY — THE BACKLOG ENTRY CARRIES A RECORDED REASON BEFORE ANY WORK STARTS

`"addresses ticket #2044 - GOLD members asked how to use their points"` — recorded reason: **present**, avoiding Chapter 1's own verified 80%-untraceable outcome under no process, without imposing Chapter 1's own 61x heavy-process tax on a normal feature request.

STEP 2 Early Check-In (Chapter 2)

VERIFIED DIRECTLY — CHECKING IN AFTER 2 OF 6 SUBTASKS AVOIDED 67% OF THIS SPRINT'S OWN REWORK

Waterfall (no check-in until the end): **6 of 6 subtasks** would need rework once a redemption-rate misunderstanding surfaces. Checking in after every 2 subtasks: only **2 of 6** — **67% of rework avoided**, the same mechanism Chapter 2 verified generally, now applied to this sprint's own real subtasks.

STEP 3 Sprint Planning & Daily Standups (Chapter 3)

VERIFIED DIRECTLY — THE SPRINT COMMITTED TO EXACTLY KNOWN VELOCITY AND COMPLETED 100%; A SAME-DAY STANDUP CUT A BLOCKER FROM 7 DAYS LOST TO 1

Committing to **20 points** against a known velocity of **20.0**: **20 completed, 100%**. A blocker on day 3 (payments team must confirm points can't go negative): with a daily standup, **1 day lost**; without one, **7 days lost** — the exact mechanism Chapter 3 verified, now catching a real edge case in this sprint's own work.

STEP 4 WIP Discipline (Chapter 4)

VERIFIED DIRECTLY — A WIP LIMIT OF 1 ACROSS THIS SPRINT'S OWN 6 SUBTASKS KEPT CYCLE TIME AT LITTLE'S LAW'S OWN MINIMUM

`WIP=1`: **60 minutes total, 10.0-minute average cycle time**. `WIP=6` (juggling all subtasks at once): **237 minutes total, 229.5-minute average cycle time** — the same WIP-vs-cycle-time relationship Chapter 4 verified generally, confirmed again on this sprint's own real subtask list.

STEP 5 The User Story (Chapter 5)

VERIFIED DIRECTLY — THE STORY IS INDEPENDENT, APPROPRIATELY SMALL, AND ITS ACCEPTANCE CRITERIA ALREADY IS A BDD SCENARIO

"As a GOLD member, I want to redeem my loyalty points for a checkout discount, so that I feel rewarded for repeat purchases." Independent of other in-sprint work: **confirmed**. Sized at 8 points against a 20-point sprint: **fits safely alongside other work**. Acceptance criteria: "*Given a GOLD member with 40 points, When they redeem all 40 at checkout, Then a $4.00 discount is applied*" — already Given-When-Then, per Chapter 5's own finding.

STEP 6 Planning Poker (Chapter 6)

VERIFIED DIRECTLY — SIMULTANEOUS REVEAL SURFACED A HIDDEN COMPLEXITY ANCHORED ESTIMATION DILUTED TO INVISIBILITY

One developer knows about a rounding edge case in the points data model, worth 13 points, not the 5 everyone else assumes. Anchored estimation: final estimate **5.6** — the concern nearly vanishes. Planning poker: **[5, 5, 5, 5, 13]**, an 8-point spread that triggers discussion — the team agrees on **13**, the real complexity, not the anchored 5.6.

STEP 7 Code Review (Chapter 7)

VERIFIED DIRECTLY — A STYLE-ONLY CHECKLIST APPROVED THE EXACT COMPOSED-BUG RISK A SMELL-AWARE, TEACHING REVIEW CAUGHT AND BLOCKED

`preview_redemption_discount` and `apply_redemption_discount`, well-formatted and well-named: style-only checklist **approves**. Smell-aware checklist: **flags duplicated validation+calculation logic** — the same Extract Method risk

Clean Code Chapter 8 catalogued, and the same composed-bug shape Software Testing Strategy Chapter 1 verified. It blocks merge (a real duplication-drift risk, not a nit), with the reviewer explaining *why* rather than just rejecting.

STEP 8 The Architecture Decision (Chapter 8)

VERIFIED DIRECTLY — THE EXPECTED-COST FORMULA CORRECTLY SEPARATED A DECISION WORTH AN ADR FROM ONE THAT WASN'T

Where the points balance lives (a new field vs. a separate table) — reversal cost **40 hours**: expected cost with an ADR **8.00h** vs. without **12.00h** — **writing it is the better choice**. Which HTTP status code the endpoint returns — reversal cost **1 hour**: expected cost with an ADR **2.15h** vs. without **0.30h** — **skipping it is the better choice**. The same formula, two genuinely different real decisions from the same sprint, two correctly different answers.

STEP 9 The Retrospective (Chapter 9)

VERIFIED DIRECTLY — THE RECURRING BLOCKER PATTERN GOT A REAL, TRACKED ACTION ITEM, RAISED BLAMELESSLY

Owner: **Scrum Master**. Due: **next sprint planning**. Fix: *"flag any story touching balances for payments review at sprint planning, not during the sprint."* Raised blamelessly — crediting the planning-poker catch rather than asking why the risk wasn't obvious to everyone from the start, exactly Chapter 9's own verified reporting-rate finding in practice.

Final Integration: The Feature, Verified Against Its Own Story

VERIFIED END TO END — EVERY VALUE MATCHES THE STORY'S OWN ACCEPTANCE CRITERIA, AND THE REVIEW'S OWN FIX CLOSED THE EXACT RISK IT NAMED

The reviewed, extracted `calculate_redemption_discount()`: preview and actual both return **$4.00** for 40 redeemed points — **identical**, closing the duplication-drift risk Step 7 flagged. Negative points: **correctly rejected**, closing Step 3's own standup blocker. A full checkout — $41.00 order total, minus the $4.00 redemption — correctly totals **$37.00**.

Where This Connects

THIS CAPSTONE'S STEP	DIRECT CONNECTION
A composed-bug-shaped review finding	Software Testing Strategy Chapter 1's own $1799.10-instead-of-$17.99 finding — the same risk shape, caught here before it ever shipped
The reused TangleMart system itself	Clean Code, SOLID & Refactoring's own capstone and Software Testing Strategy's own capstone — three courses' worth of capstones building on the identical codebase

THIS CAPSTONE'S STEP	DIRECT CONNECTION
The ADR decision rule	Software Architecture Fundamentals Chapter 9's own ADR format — Chapter 8 of this course added the timing/worth-it question that format doesn't answer on its own

Hands-On Exercises

EXERCISE 1

Add a second user story to this chapter's own sprint: "As a PLATINUM member, I want to see my points balance on the order confirmation page." Determine whether it's independent of the redemption story, estimate it at a plausible point value, and verify the sprint's own total commitment still fits within the 20-point known velocity.

View solution

EXERCISE 2

Using this chapter's own ADR expected-cost formula, determine the reversal-cost threshold above which writing an ADR becomes the better choice for a decision made during this sprint, using the chapter's own risk parameters.

View solution

EXERCISE 3

Extend this chapter's own final integration check with a second redemption in the same sprint — a PLATINUM member with 25 points redeeming all 25. Verify the discount amount, verify it doesn't affect the first GOLD member's own already-completed redemption, and verify negative-point rejection still works for this second customer too.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE SUMMARY

- **Verified end to end:** one real sprint — a loyalty-points-redemption feature — carried through all 9 prior chapters' own techniques, each directly reusing that chapter's own verified formula on fresh, real inputs
- **Step 2 avoided 67% of rework** by checking in early instead of waiting for the sprint review
- **Step 6's planning poker surfaced a hidden 13-point complexity** that anchored estimation diluted to 5.6
- **Step 7's review caught the exact composed-bug risk** Software Testing Strategy Chapter 1 first verified, before it ever shipped
- **Step 8's formula correctly separated** a genuine ADR-worthy decision from a trivial one, using the same rule twice with two different real answers

- **Course complete:** process cost, Agile foundations, Scrum, Kanban, requirements, estimation, code review, ADRs, and retrospectives — all ten chapters, verified throughout
- **Subject complete:** this closes the Software Development subject's entire original seven-course scope