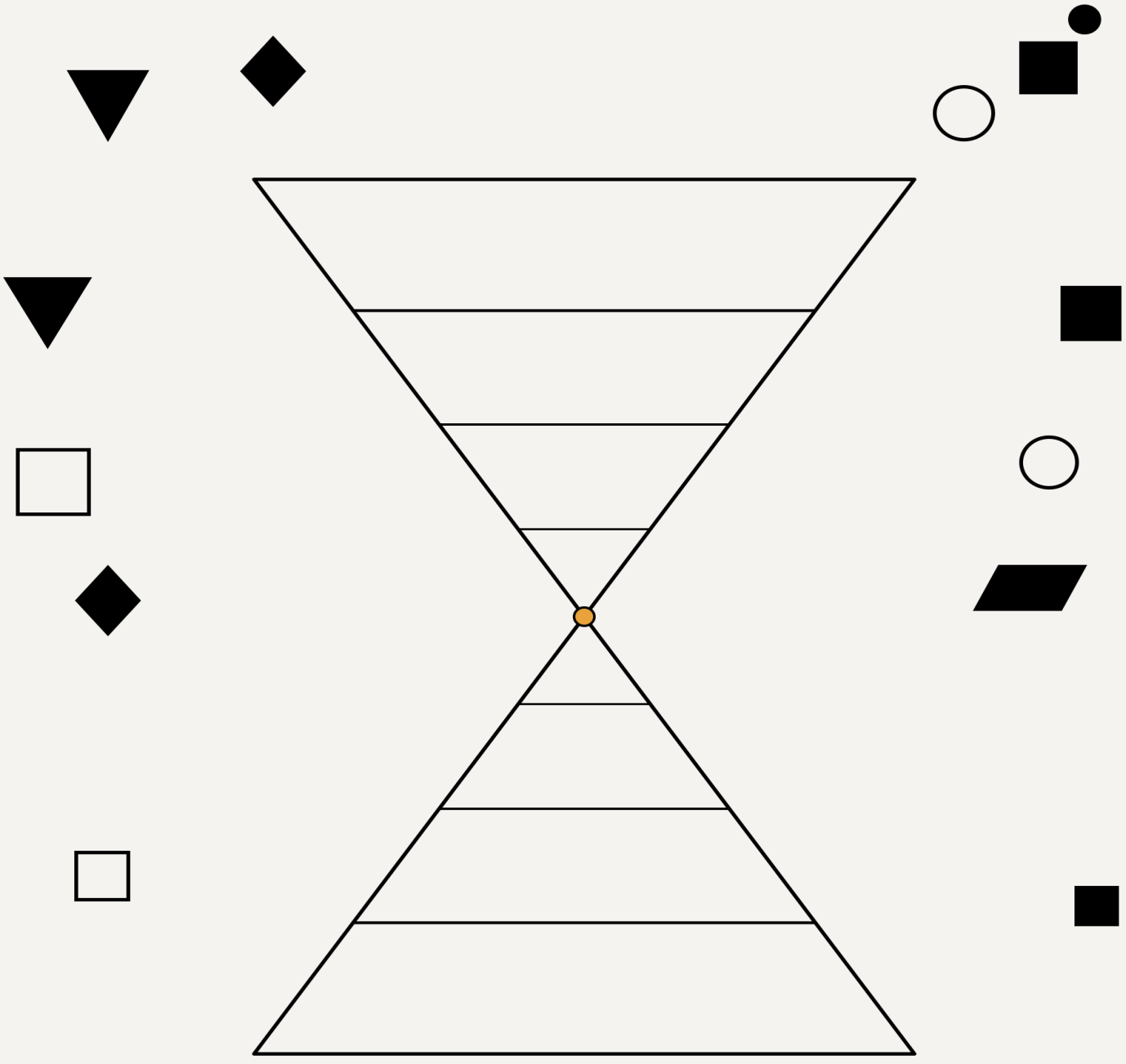
Process-First Design

Less art, more engineering



Sergiy Yevtushenko

Process-First Design

Less art, more engineering

Sergiy Yevtushenko

Version 1.0.0 — June 2026

Contents

Introduction

Software design is the last craft in the building. The chips were standardized, then the languages, the protocols, the deployment — and the shape of the code was left to taste, so that the same business problem yields a different structure in every shop and every sprint. This book is about the part that never got standardized, and the quiet evidence that it is standardizing anyway.

The thesis

Process-First Design is a way of deciding what software is made of. Its claim is one sentence: **the unit of design is the process, not the entity** — what the software *does*, not what it *is*. You design a thing that happens, a trigger producing an outcome, and you shape the data around the processes that use it rather than around a model of the domain that exists before any use.

That is a small statement with a large consequence, because the dominant tradition runs the other way. Entity-first design starts from a shared model of each domain concept — one `Customer`, one `Order`, one `Product` — and builds behavior on top of it. Process-first inverts the order: the process comes first, and types belong to processes. A "customer" in one process is the shape that process needs; what is genuinely common across processes is a small, shared kernel of value objects, and nothing else is shared by default.

Underneath the inversion is a simpler reason, one the book develops as it goes: a process is an act of knowledge gathering, and modeling around what a process needs to *know* rather than what a system happens to *store* is what produces per-process types at all. Entities answer "what data exists?"; a process answers "what does this one need to know?" — and the second question is the one that yields the methodology's types.

The book's subtitle is *less art, more engineering*, and it is meant literally. Most of what makes software design feel like art is the absence of a method for the decisions it forces — so the decisions get answered by taste, and taste varies by person, by mood, by deadline. A method does not remove the judgment that genuinely belongs to design; it removes the judgment that never should have been judgment, the dozen small structural decisions that have a right shape and were being re-litigated every time. What remains for human judgment is the part worth a human: the architecture, the domain, the hard trade-offs. The mechanical part becomes engineering.

This is the methodology layer. It is language-neutral: the principles hold in Scala, Kotlin, Rust, C#, TypeScript as readily as in Java. Where the book shows code, it shows Java, because the Java implementation — Java Backend Coding Technology — is the most fully developed instance and the audience is largely Java. But the methodology is the *why*; JBCT is one *how*. A reader can adopt process-first design in any of those languages without opening the JBCT book.

A convergence, not an invention

The methodology does not claim to have discovered process-first design. It claims to have *named* something practitioners keep arriving at independently.

Across the last several years, designers working in different languages, different domains, and different communities have converged on the same structural move: organize around business operations rather than shared entities; shape types to the process; compose small, pure, well-typed operations instead of layering services over a data model. Scott Wlaschin reached it in F#, modeling domains as workflows with typed inputs and outputs. Debasish Ghosh reached it in Scala through algebraic composition. Jimmy Bogard reached it in .NET by organizing code in vertical slices. Sandro Mancuso reached it from craftsmanship, letting the domain emerge from use. Rico Fritzsche reached it building Request Processing Units, self-contained units that each handle one domain request around a functional core. None cites the others as the source; they were solving the same problem and the same structure fell out. From a different direction entirely — formal rather than empirical — Yannick Loth's Independent Variation Principle reaches the same partition by asking what causes each element to change. The full survey of this convergence is a story in itself, developed in the author's *The Quiet Consensus*; here it is enough to know that the convergence exists and that it is wide.

That convergence is this book's anchor, and it is worth being precise about what an anchor is. It is not proof. Six practitioners agreeing does not make a methodology correct; the book's actual case rests on the methodology *working*, demonstrated on real code through the spiral that follows, not on the company it keeps. The convergence does something narrower and still valuable: it suggests the structure is real — discovered, not imposed — because independent people keep discovering it. A method that merely recognizes what good designers already do, and makes it teachable and consistent, is on firmer ground than one inventing a new way to think.

It also reframes what the book is doing. It is not arguing that the industry *ought* to standardize on process-first design. It is observing that the industry is *already* converging on it, and offering a vocabulary for what is emerging. The reader is invited to observe, not to be persuaded — the same way every other engineering discipline industrialized, moving from craft, where each artifact is unique and taste-driven, to standardized practice, where parts are interchangeable and the process is predictable, without ever losing the room for genuine design. The author's *Software's Industrialization Moment* develops that parallel; this book is what the standardized practice looks like for backend software design.

What this book commits to

Five principles run through every chapter. They are stated here and enforced throughout; where the book seems to make a choice, it is usually one of these.

1. **Legibility first.** Code is written to be read — by the next developer, by the author a year later, by the reviewer who met neither. Every structural choice

is a concession to that reader. Extra words that aid the reader are purchased legibility, not ceremony.

2. **Show possibilities, don't make claims.** The book describes what the methodology makes possible and demonstrates it on real code. It does not promise productivity multipliers, fewer bugs, or compliance readiness. Where evidence is absent, it says so plainly, and leaves the reader to map the possibilities onto their own situation.
3. **More time for the interesting work.** The discipline offloads the mechanical decisions — where failures live, how state threads through a process — so attention moves up to architecture and domain judgment. Most developers want more of that work, not less; this is how the structure buys it for them.
4. **Methodology fits the work, not the reverse.** The vocabulary scales with complexity. Trivial code stays trivial; the methodology earns its weight only where complexity demands it. It is a tool, not a recipe to be applied at uniform strength to everything.
5. **Show it's happening; don't argue it should.** The book reveals a convergence rather than advocating a position. The reader becomes an observer of something already underway, not the target of a case for adoption.

There is also a discipline of scope, load-bearing enough to state once and keep: this methodology is bounded to enterprise backend software — systems large enough and long-lived enough for structural coupling to become the dominant cost. It does not claim to be the right tool for real-time control loops, numerical kernels, game engines, or throwaway scripts. Knowing where a method does not apply is part of applying it honestly.

The running example

One example carries the book's main arc: a ticketing platform. It starts as small as software gets — a customer buys a ticket for a specific seat at a specific event — and grows, pass by pass, into a multi-venue, multi-tenant platform. Across the spiral and the synthesis that follows it, the domain never changes; only its scope expands.

This is deliberate. A methodology's claims are easy to make on a fresh toy in each chapter and hard to make on the same material at increasing magnification. Carrying one domain up through every altitude — use case, workflow, subsystem, system — is the honest test: the reader watches the methodology handle the same seats, holds, and payments at four scales and can judge whether it holds together or frays. Event ticketing is chosen because it is universally recognizable, because its concepts (seats, holds, reservations, prices, events) are rich enough to exercise every part of the methodology, and because nobody needs a domain expert to follow it.

One domain is carried this way on purpose, and one chapter just as deliberately breaks from it. Brownfield, at the end, applies the methodology in reverse to a second domain — a payroll system the book never designed. That is not a lapse in the single-example

discipline; it is the other half of the test. Carrying one domain up through every altitude shows the methodology under magnification; turning it loose on an unfamiliar one shows that it travels, rather than having been quietly shaped to fit the first.

How to read this book

The book is built to be read in one sitting — start to finish in an afternoon, not a chapter a week. That target is not a courtesy; methodology read in fragments loses its connective tissue, and a reader who cannot hold the whole vocabulary in working memory cannot see the through-line. The structure serves that reading:

- **Spiral 0** comes before any methodology: the decisions a single use case forces, and why you are already answering them. It is the problem the rest of the book solves.
- **Foundations** names the vocabulary — four shapes, six patterns, six properties, the telescope, three recovery classes, two axes of cohesion. Read it once; refer back as needed.
- **The spiral** is the core: four passes applying that vocabulary to the running example at successive altitudes, use case through system. The passes get shorter as they climb, and that is the point rather than an accident — each altitude reuses the vocabulary and adds only its own small delta. By the last pass you will have learned almost nothing new, which is the methodology demonstrating its central claim on itself.
- **Architecture Synthesis** then pays the debts the spiral defers: how to turn a design into an architecture, the full decision framework the passes only gesture at.
- **Brownfield** applies all of it in reverse — to a system you inherited rather than one you start clean.
- **Closing** looks at what the methodology does not cover, and what to expect.

Every chapter names, at its foot, which of the book's narrative threads it advances; a second reading can follow a thread rather than the chapters. The prose keeps a surface that lands on first read, with the deeper structure available to a reader who returns. None of it requires the second read; all of it rewards one.

The spiral begins where the work begins — at the smallest unit of it, one customer buying one ticket. But before the methodology, the problem it answers: the decisions that use case forces whether or not anyone has a method for them.

Spiral 0 — The Decisions a Use Case Forces

Ask three competent teams to build "buy a ticket" and you get three codebases that pass review and disagree on nearly everything that matters. Not because anyone was wrong — because the use case forces a dozen small decisions, every one of them has a defensible answer, and nothing made them answer the same way. This chapter is those decisions, before any methodology is in the room.

What this chapter does

This is a warm-up, not a pass. The spiral that follows walks a real methodology through four altitudes; before it starts, it is worth seeing the problem the methodology exists to solve — not as an argument, but as a list of decisions you already make.

Take the smallest unit of real work: a customer buys a ticket for a specific seat at a specific event. One trigger, one outcome. It looks trivial. It is not, and the reason it is not has nothing to do with difficulty. The work is easy. What is hard is that the use case quietly forces a series of decisions, each one has several defensible answers, and without a shared method you answer them by instinct — differently on Tuesday than you did on Monday, differently from the developer at the next desk, differently in this use case than in the one beside it.

The chapter walks those decisions. It does not show code, because the point is not that someone wrote bad code; competent people write defensible code for every one of these. The point is that the decisions are unavoidable and the consistency is not automatic. That gap is what the methodology closes.

The decisions

Where do the failures live? Buying a ticket can fail: the seat was taken a moment ago, the payment was declined, the event closed sales, the customer is not eligible. Those are not edge cases bolted on later; they are part of what "buy a ticket" *means*. So where do they go? Exceptions thrown and caught somewhere up the stack? Error codes returned and checked? A result object carrying success or a named failure? Some of each? Every answer is defensible, and you must pick one — there is no version of this use case that does not decide where failure lives. Most codebases decide it more than once, differently each time, and the documentation drifts from the code the first week.

Is this "customer" the same "customer" as everywhere else? The use case needs the customer who is buying. Is that the same `Customer` that registration creates, that profile-update mutates, that billing charges — one shared class used everywhere — or a shape specific to buying? Pick the shared class and it grows a field for every use case that ever touched it, until no use case wants all of it. Pick a local shape and you have to say what relates it to the others. You must decide, and the decision compounds: every use case that mentions a customer inherits it.

What happens to the reservation when payment fails after it? To take the money you first hold the seat. Then the payment is declined. Now there is a held seat and no sale. Release it immediately? Leave it for a cleanup job to sweep later? Trust an expiry to handle it? Forget it exists? You must decide — the held seat is real whether or not you decided on purpose — and this is the decision most often made implicitly, which is to say made by whoever later debugs the orphaned holds.

If the same logic has to fire from a queue tomorrow, what moves? Today buying a ticket is an HTTP endpoint. Tomorrow a partner integration needs the same booking driven from a message queue, or an admin tool needs it from a script. How much of what you wrote was about *buying a ticket* and how much was about *being an HTTP request*? You must decide where the trigger ends and the business begins — and if you never decided, the answer is "they are tangled," and the second trigger means copy-paste.

Where do logging, retries, and audit go? Every operation needs to be observable, some calls need to be retried, money movements need an audit trail. In the method body, inline with the business steps? Scattered across helpers? In a wrapper around the call? You must decide, and without a convention the answer is "wherever each developer put it," which is to say inconsistently, which is to say the observability is uneven exactly when you need it to be uniform.

None of these is exotic. There are five here and the real use case has more. Every one is small. Every one has a reasonable answer. And every one is answered, every time, by everyone who builds the use case — there is no opting out.

You are already answering these

That is the whole point, and it is easy to miss because the decisions feel too small to notice. You are already answering all of them. The only question is whether you answer them *consistently* — the same way in this use case as the last, the same way as the developer beside you — or *ad hoc*, by individual judgment, freshly each time.

Without a shared method, it is ad hoc. Not through carelessness; through the absence of anything that would make it otherwise. Each developer answers each decision reasonably, and the reasonable answers differ, and the differences accumulate. The cost is not in any single answer — any one of them is fine. The cost is the *drift*: a codebase where failure lives in three places, where "customer" means four things, where compensation is a pattern in one corner and a bug in another, where half the use cases are weldable to their trigger and half are not. That drift is what "this codebase is hard to work in" actually is, decomposed.

This is why architectural arguments feel endless. They are often these decisions, surfaced as preference — one developer prefers exceptions, another result types, and both are defending a reasonable answer to a question that has several. The argument does not resolve because nothing about the question forces a single answer. What ends it is not winning the argument; it is a method that answers the question once, the same way every time, so the question stops being asked.

Why the answers can be simple

Here is the part that is easy to disbelieve: the methodology that follows answers every one of these decisions, and the answers are as simple as the questions.

Failures live in the return type, as a closed set of named outcomes the caller must address. “Customer” is a small shared value object where it means the same thing everywhere, plus a shape local to each use case where it does not. The held seat’s fate is a named compensation, decided when the use case is designed rather than when it is debugged. The trigger is one of the use case’s six properties, named separately from the business so a second trigger costs nothing. Logging, retries, and audit are quarantined — business code stays clean, the runtime supplies them uniformly. Each answer fits in a sentence.

The simplicity is not a happy accident; it is the mechanism. A methodology you can apply from memory is one you actually apply — every use case, the same way, without opening a manual. A methodology you have to look things up to follow gets applied unevenly, or abandoned under deadline. The power is not despite the simplicity; it is *because* of it. Simple decisions, answered the same way every time, are the only kind that get answered consistently across a team and a codebase and a year — and consistency, not cleverness, is what makes a large system stay legible. The decisions in this chapter are trivial. That is exactly why a small, fixed set of answers can dissolve them.

Into the methodology

The decisions do not go away. There is no design so clever that “buy a ticket” stops needing to know where its failures live or what becomes of a held seat. The methodology does not remove the decisions; it answers them once, simply, and the same way every time, so that you stop re-deciding them and the codebase stops drifting.

Surfacing a decision early does one more thing, quieter than consistency: it is a chance to fix the business behind it, not only the code in front of it. The held seat with no sale is a gap in the business process itself, and naming it while that process is still being designed is when it is cheapest to close. Answering these decisions by construction is a way to harden the business before any of it is written down.

The chapters that follow supply the answers. The next names the vocabulary — the small fixed set of shapes, patterns, and properties the answers are built from. The spiral after it walks that vocabulary through the same use case you just met, then up through workflows, subsystems, and the whole system, applying the same answers at every scale. Watch, as you read, how few new things you have to learn after the first pass. That is the methodology keeping the promise this chapter makes: the decisions are small, the answers are small, and small is what lets them hold.

Foundations

Spiral 0 ended on a promise: the decisions a use case forces have a small, fixed set of answers. This chapter is that set. It is the shortest kind of chapter to write and the easiest to underestimate — a vocabulary, not an argument. Everything the spiral does afterward is this vocabulary, applied.

What this chapter does

The previous chapter raised the decisions every use case forces and claimed they have simple, consistent answers. This chapter names the pieces those answers are built from. It defines; it does not demonstrate. Demonstration is the spiral's job — four passes that take this vocabulary and apply it at use-case, workflow, subsystem, and system altitude. Read this once and refer back to it; nothing here needs to be memorized, because the spiral will use every piece often enough to make it stick.

The vocabulary is deliberately small. Four shapes, six patterns, six properties, three recovery classes, two axes of cohesion, one organizing structure. That smallness is the point Spiral 0 made: a vocabulary you can hold in your head is one you apply from memory, the same way every time. The list below is the whole of it. The reason the book can be read in one sitting is that the list does not grow as the altitudes climb.

Process-first

The stance the whole methodology rests on, stated plainly: **the unit of design is the process, not the entity.** What you design is a thing that *happens* — a trigger producing an outcome — not a thing that *is*. Data structures are shaped by the processes that use them, not by a model of the domain that exists prior to any use.

The immediate consequence is that types belong to processes. A "customer" in *buy ticket* is the shape buying needs; a "customer" in registration is the shape registering needs; they are different types that happen to share a noun. What is genuinely common — an identifier, a money amount, a thing that means exactly the same wherever it appears — is a small shared value object. Everything else is local to the process that uses it.

This is a bet, and it is worth naming as one. Entity-first design — one shared model of each domain concept, used everywhere — has a real advantage: no duplication, one place to look. Process-first gives that up; it pays in per-process types and in the discipline of deciding what is genuinely shared. The bet is that the coupling cost of the shared model grows faster than the duplication cost of per-process types, so that past a certain scale process-first is cheaper to maintain. The spiral does not argue this bet; it shows the methodology operating and lets the reader judge whether the result is the kind of code they want to maintain. The bet's scope is bounded: enterprise backend, where systems are large and long-lived enough for the coupling cost to bite.

The methodology does not claim to be the right tool below that scale or outside that domain.

None of this is a ban on entities. An entity earns its place when a business invariant genuinely spans more than one field — when a whole assembled from individually-valid fields can still be invalid, so the combination needs guarding, not each field alone. Its role is to enforce that cross-field invariant during persistence: it is the composite every write to that state passes through, so the store can never hold a combination the invariant forbids. Process-first changes only the default — the process is the unit of design, and an entity is introduced where a cross-field invariant demands one, not placed at the centre of the domain because a concept happens to have a name.

The six properties of a process

A process has six properties, and they are the same six at every altitude — that is what makes the telescope below possible. Their granularity changes as you climb; the list does not.

- **Trigger** — what makes the process run: an incoming request, a scheduled tick, an event, a human approval resolving. The trigger is not the transport that carries it; HTTP versus a queue is not a different trigger. A process needs at least one trigger and may have several (a fulfilment process can begin from a purchase or from a restock), but its *outcome* is what defines it: the same outcome reached through different triggers is one process, while the same steps performed for a different outcome are different processes.
- **Typed input** — what the process needs to begin, as a precise type carrying exactly that and no more.
- **Typed output** — what the process produces on success.
- **Typed failures** — the enumerable, closed set of ways it can fail, each a named domain fact, part of the specification rather than an exception raised elsewhere.
- **Steps** — its internal sequence of operations, named in domain terms and visible at the altitude they belong to.
- **Dependencies** — the graph between the steps: what must precede what, what can run in parallel, what is conditional.

Name these six for any process and it is, in the methodology's terms, specified. The spiral walks the same six at each altitude, with the steps becoming use cases, then workflows, then subsystems as it climbs.

A process gathers knowledge

Underneath the six properties is a simpler way to see what a process is: an act of knowledge gathering. A process begins knowing only its trigger and its input, and each step acquires one more piece of what it needs — *buy ticket* learns that the request is well-formed, then that the event is selling, then that the seat is held, then that the payment cleared. The process ends, success or failure, the moment it knows enough to

answer. A declined payment is not a missing outcome but knowledge in its own right, enough to answer *the ticket cannot be sold*, so the process stops there rather than gather what it no longer needs. Typed failures and short-circuiting are not machinery bolted on; they are the process recognizing it is done.

This is also why the types belong to the process. Ask of a domain "what data exists?" and the answer is entities: one `Customer`, one `Order`, one shared shape every process must accept. Ask instead "what does *this* process need to know?" and the answer is the per-process types the methodology produces — the smallest input the trigger carries, the typed knowledge each step adds, the closed set of facts, failures included, that let the process answer. Process-first is what falls out when you model around the knowledge a process gathers rather than the data a system stores.

Not every step adds to that store. Some only confirm a precondition (the caller is entitled, the request within its rate) and nothing downstream reads their result; they gate the process without enriching it. The test is whether a later step needs the outcome: when it does, the step must carry it forward as knowledge the next step consumes, for a result some later step depends on, discarded once tested, is the loss *parse, don't validate* warns against, one boundary inward. When nothing does, the check is a pure gate, and its single fact — *the process may continue* — is spent the moment it does.

The four shapes

Every value a process handles has one of four shapes, and the shape is a domain statement, not a stylistic choice. They are type-honest: the type says what the domain knows about the value. A type's capacity to carry a business statement rather than merely a layout is its *semantic potential* — the term is William Jackson's — and the four shapes are the first place the methodology spends it.

- **`T`** — the value exists, unconditionally. No absence, no failure, no waiting.
- **`Option<T>`** — the value may or may not exist, and its absence is a domain fact, not an error.
- **`Result<T>`** — the operation may or may not have produced the value; failure is a typed outcome carried in the type, synchronous.
- **`Promise<T>`** — the value arrives later, and may fail; `Promise<T>` carries both the asynchrony and the failure.

Asynchrony emerges from leaves: any operation that touches I/O is a `Promise`, and the `Promise` propagates outward through everything that depends on it. It is not a decision made at the top; it is a fact that rises from the bottom.

A type carries a claim — a `CustomerId` claims to be a valid identifier, a `Money` claims to be a non-negative amount in a known currency — and construction is where the claim is enforced. This is *parse, don't validate*: the value cannot exist in an invalid state, so code that receives it can trust it without re-checking. The enforcement has four levels, strongest to weakest, and the methodology's commitment is identical across all of them; only the teeth differ:

1. **Type-level** — a refinement or dependent type the compiler rejects when invalid. Strongest; available only where the language supplies it.
2. **Construction-level** — a non-public constructor and a factory that returns `Result<T>`, so no invalid value can be built. This is the level the Java implementation uses.
3. **Runtime-level** — validation at the boundary, catching invalid values when they occur rather than preventing their construction.
4. **Convention-level** — discipline and review, with nothing in the code enforcing it.

The shape and the enforcement together are why a process body is free of defensive checks: the types arriving have already made their claims good.

The six patterns

Composition has six primitives, and the same six compose a process at every altitude. Each maps to a recognizable shape of work; together they are sufficient, and the methodology adds no others.

- **Leaf** — an atomic unit: a boundary crossing (I/O, an external call) or a pure computation. The bottom of composition; everything else composes Leaves.
- **Sequencer** — steps in order, each feeding the next, short-circuiting on the first failure.
- **Fork-Join** — independent steps run in parallel and joined.
- **Condition** — a branch on a business fact, routing between legitimate alternatives. Typed, never a bare boolean where the fact carries meaning.
- **Iteration** — a step applied across a collection.
- **Aspects** — cross-cutting concerns that wrap operations uniformly: business cross-cutting (audit, compliance — part of the design) and technical cross-cutting (logging, tracing, retries — supplied by the runtime).

Each function implements exactly one pattern; mixing patterns is the signal to split. The patterns are not invented by the methodology — practitioners arrived at them independently, and they parallel the constructs business-process notations have used for decades. The methodology recognizes the structure rather than imposing it.

How those steps reach one another is a second question, and it has two answers. In **direct step composition**, a step calls the next and composes on the result it returns: the chain is written out, output feeding input, the shape every pass of this book uses by default. In **event-based step composition**, a step publishes a typed fact and the next is triggered by it: the same steps, wired through events rather than return values. The two are not a timing distinction — direct composition is still `Promise`-based and non-blocking; the difference is whether coordination rides the return path or a published fact. It is not an altitude distinction either: a use case's steps can compose either way without becoming anything other than that use case. Which substrate a given composition uses is an architectural choice, decided in Architecture Synthesis against the system's Phase-4 inputs; the methodology names both and prefers neither.

The telescope

The methodology's organizing structure is a telescope: the same vocabulary at successive scales. A **use case** is one business operation — one trigger, one outcome. A **workflow** is a composition of use cases for one business outcome. A **subsystem** is a coherent business concern, a cluster of workflows. A **system** is the composition of subsystems, and the top of the telescope. There is a rung above it, the enterprise, but the book stops at the system on purpose: above the system the composing force is organizational and strategic, Conway's law and the shape of the business itself, not change-driver cohesion the code can express. The telescope reaches as far as its mechanism holds, and no further.

Altitudes are not imposed; they emerge from multiplicity. One use case is one use case. When several cohere, a workflow appears. When several workflows cohere, a subsystem appears. The methodology lets the emergence happen rather than forcing a hierarchy in advance.

Two operations recur at every altitude, and keeping them distinct is essential:

- **Within-altitude composition** — how the units *at* an altitude compose into one unit — is the six patterns. It happens at every altitude, including the use case (its steps compose via Sequencer, Leaf, Fork-Join).
- **Cross-altitude grouping** — how units of the level below *cohere to form* a unit at this level — happens only where a level is formed from a lower one. It is **change-driver cohesion**: units cohere when a single business force governs them, when one change would force them all to change together.

The recognition test for grouping is one question, asked at each transition: *what business change would force all of these to change together?* If a single force rewrites them all, they cohere. The mechanism is identical at every transition; the *kind* of driver differs by altitude:

Transition	Units grouped	Change-driver character
use cases → workflow	use cases	a business policy (reservation rules, refund rules)
workflows → subsystem	workflows	a domain concern (the booking domain, the pricing model)
subsystems → system	subsystems	the product/platform boundary + operational envelope

The use case is the floor of this ladder: the smallest business-meaningful unit, not formed by grouping a lower *business* altitude. Its steps are internal composition, not a lower altitude — so the grouping question begins at the workflow and recurs upward, while the use case has only within-altitude composition.

That question has two directions, and a grouping is right only when both hold. **Completeness**: is every unit this driver governs inside the group, or are some scattered elsewhere, so that one change has to chase them across modules — the smell teams know as shotgun surgery? **Purity**: is only what this driver governs inside, or is a foreign unit riding along, so that its unrelated changes leak in as accidental coupling? The sharpened test asks both at once: does this one change force *all* of these, and *only* these, to change? Pass both and the group is cohesive; pass one and it is not.

The same criterion, derived independently and given formal shape, is Yannick Loth's Independent Variation Principle: unify elements with the same change-driver assignment, separate those with different ones. Process-First Design reaches the criterion from the process side; the Independent Variation Principle reaches the same partition from the change-driver side — two paths to one territory. The methodology uses change-driver cohesion as its own; it recognizes IVP as corroboration, not foundation.

Loth gives the cohesion behind the test its own formal account in *On the Nature of Cohesion*: cohesion as correctness relative to a modularization principle, along exactly these two axes. Process-First Design's change driver is one such principle, so the convergence IVP named now reaches cohesion itself — and the same account explains why the old structural metrics never captured it, since they measure what code shares rather than what governs it.

One consequence of the telescope is worth stating because the spiral relies on it: a unit's composition at one altitude is a Leaf at the altitude above. A workflow is a composition of use cases, and a Leaf to its subsystem. A subsystem is a composition of workflows, and a Leaf to its system. The patterns recur because the structure is fractal.

Because the structure is fractal, it is also how the code is organized. The altitudes are not only the order in which the design is discovered; they are the order in which a reader navigates the codebase, browsing from system to subsystem to workflow to use case — which is the answer to the worry about a flat list of a thousand use cases. The companion volume, Java Backend Coding Technology, names the package realization of this the *telescope rule*: each altitude becomes a package level that appears only when the design discovers it, the use cases below moving under it, while shared code settles at the lowest altitude that covers all its users.

The recovery triple

When something is invalidated — a step fails after earlier steps have changed state — there are three responses, and the methodology names all three where most discourse names only the first.

- **BER (Backward Error Recovery)** — compensate by an inverse action. Release the held seat, void the authorization, reverse the entry. The classic rollback or saga shape.
- **FER (Forward Error Recovery)** — continue with degraded state rather than undoing. Defaults under partial failure, a notification queued for retry while the booking stands, a value allowed to decay through `fresh` → `stale` → `expired`.

- **Design-out** — change the model so the invalidation cannot arise. An immutable log corrected by appending rather than overwriting; a reservation model where two bookings of one seat is structurally impossible; an idempotent operation safe to repeat.

Which applies is a judgment across four axes — reversibility, forward-progress value, domain shape, coordination cost — and mixed strategies are normal: a system can use BER for money, FER for telemetry, and design-out for collaborative state, coherently, at once. The spiral surfaces which response each altitude reaches for; the full selection mechanism is the Architecture Synthesis module's work.

Reading the spiral

That is the whole vocabulary. The spiral now applies it — four passes through one running example, event ticketing, at successive altitudes. Pass 1 takes a single use case, *buy a ticket*, and answers the decisions Spiral 0 raised, in full. Pass 2 lets workflows emerge as use cases multiply. Pass 3 lets subsystems emerge as workflows cluster. Pass 4 reaches the whole platform.

One thing to watch as you read: how little is new after the first pass. The vocabulary does not grow; each altitude reuses it and adds only the small delta that altitude makes visible. The passes get shorter as they climb, and that is deliberate — not the book running out of things to say, but the telescope doing exactly what it claims. Keep that in view; by the end it will be the clearest evidence the methodology offers about itself.

The spiral begins at the smallest unit of the work: one customer, one seat, one event.

Spiral Pass 1 — Use Case Altitude

A customer wants a seat. The system has eight steps to give them one. The methodology has six patterns, four shapes, and one recovery class to spend on the work — and one of those patterns won't show up at all.

What this pass does

Foundations named the pieces. This pass walks them through one use case, *buy a ticket for a specific seat at a specific event*: one trigger, one outcome, eight steps. By the end the six properties will have applied at the altitude where they are concrete and immediate, the per-process types will have been invented, the patterns that show up at this altitude will have, and the patterns that do not — compensation, time-as-decay, Aspects beyond runtime ones — will be visible as absences, clues to what the next altitude will earn. The methodology is not the pieces; it is what the pieces do when a team uses them on real work. This pass shows that.

The use case, opened

Trigger

The trigger for *buy ticket* is an incoming request from a customer-facing channel; for this pass, synchronous HTTP. In a typical deployment that means an HTTP POST to a specific endpoint, but the trigger is not the protocol; the trigger is what the protocol carries. Different trigger sources (scheduled job, message-bus event, human approval flow) make the same logical work into different use cases with different signatures, different observability requirements, and different failure modes for the caller-not-waiting case. That multiplicity earns its place at workflow altitude; at use-case altitude the trigger is fixed.

For *buy ticket* the customer is waiting. Latency matters in single-digit seconds. The use case must respond with success or with a structured failure that the customer-facing client can render.

Typed input

The input is what *buy ticket* needs to begin. Three things, in this use case:

- The customer placing the order. Specifically the customer's identifier, not the customer's full record. The use case needs to know who is buying; the use case does not need to know the customer's address, payment history, marketing preferences, or anything else stored on the customer. The minimal input is the smallest input that the use case actually needs.
- The event the customer is buying for. Again the identifier, not the event record. The use case will look up event-relevant facts (is the event still selling tickets, what is the current price for this seat category) at the point where those facts matter, not by passing them in.

- The specific seat the customer wants. At use-case altitude this is a seat location — a section, a row, a number — not a free-text description and not a database row identifier. The seat is named in the venue's terms.

The input type is process-local. It does not exist elsewhere in the system as a shared definition. It is a structure of raw fields, the shape that arrives at the trigger boundary before any parsing has happened:

```
BuyTicket.Request:
    customer: String
    event: String
    section: String
    row: String
    number: int
```

(A note on notation, since this is the chapter's first code block: type shapes appear in pseudo-code for readability; composition snippets later in the chapter use Java, to stay faithful to the JBCT idioms they depend on.)

That is the use case's input type, in full. It carries what *buy ticket* needs and nothing more. If a different use case (cancel ticket, check availability) needs different information, that other use case has its own input type. The fact that both involve "a customer" and "an event" does not produce a shared parent type. Sharing without evidence of common meaning is a category error that produces structural problems downstream; the use cases involve customers and events but they involve them differently.

`CustomerId`, `EventId`, and `SeatLocation` — the validated forms the use case parses these raw fields into — work the other way. They are value objects, and they must be defined identically across every use case that references them. The system's integrity depends on it: when *buy ticket* and *cancel ticket* both name `CustomerId`, they must refer to the *same* customers, with the *same* construction rules and the *same* equality semantics, not to use-case-local notions that happen to share a name. Value objects are the kernel of the type vocabulary that travels intact across processes; per-process types are the surface each use case shapes for its own needs. The discipline is recognizing which is which, and the rule is structural: per-process types must stay local because the same noun means different things in different processes; value objects must stay common because the same noun means exactly the same thing wherever it appears, and the system's invariants depend on that identity.

Typed output

The output is what the use case produces when it succeeds. *Buy ticket* succeeds when the customer has a confirmed claim to a specific seat at a specific event and payment has settled. The output structure says exactly that:

```
BuyTicket.Response:
    ticket: TicketId
    seat: SeatLocation
    eventStartsAt: ZonedDateTime
    receipt: ReceiptId
```

The customer's client renders this. The ticket identifier is the load-bearing piece — it is what the customer presents at the venue. The seat is echoed back so the client can display where the customer is sitting. The event start time is bound to a location with a specific time zone; the field type reflects that. The receipt identifier is what the customer uses to retrieve a printed proof of payment.

This output type is also process-local. The "ticket" referenced here is the customer's claim — it carries the identifier and presentation-relevant fields. A different use case (validate ticket at venue gate) has its own notion of what a ticket is for its purposes; that notion shares the identifier but adds different fields.

Typed failures

The use case can fail in specific, enumerable ways. They are not exceptions to be raised somewhere; they are part of the use case's specification.

For *buy ticket*:

- **PaymentDeclined** — the payment provider rejected the authorization.
- **SeatUnavailable** — the seat is already booked or currently held for another customer.
- **EventNotSelling** — the event has closed sales (sold out, cancelled, completed, paused).
- **CustomerIneligible** — the customer's account state prevents booking (suspended, age-restricted for this event, region-restricted, exceeded purchase limit).
- **PaymentProviderUnavailable** — the payment provider returned a transient error rather than a definitive accept or reject.
- **ConcurrentBooking** — another booking process committed the seat between this use case's check and its reservation attempt.

Each failure is a domain fact. Each one has a known set of responses (retry, fall back, surface to the customer, alert operations). The set is closed at design time; it does not grow at runtime through unanticipated paths. A new failure mode is a new entry, added deliberately, with its own response policy. In Java with JBCT, the closed set is encoded as a sealed interface extending `Cause`: fixed-message variants live in a nested enum, variants with data live in nested records. Failures are lifted into the return type via fluent construction (`cause.result()`, `cause.promise()`), never via `Result.failure(...)` or `throw`.

For *buy ticket* the split is mechanical. `SeatUnavailable` and `ConcurrentBooking` carry no extra data and live in the nested enum. The other four carry data the caller needs to render or retry (the provider's decline reason, the event's closure type, which eligibility restriction applies, the transient error's category) and live as nested records.

The use case's signature is short:

```
Promise<BuyTicket.Response> buyTicket(BuyTicket.Request request)
```

`Promise<T>` is the methodology's async shape, used by any use case that performs I/O — even when the trigger is synchronous, as it is here. Reads from stores, calls to

the payment provider, writes to the ticket record, and notifications are each asynchronous Leaves, and the asynchrony propagates outward. `Promise<T>` carries both the async modality and the failure modality: it resolves to either the success value (`BuyTicket.Response`) or one of the enumerated `BuyTicket.Failure` variants. A caller pattern-matches on both branches; the compiler insists that both are addressed. The HTTP handler that calls this use case awaits the Promise's resolution and renders the response to the customer as a synchronous response from the client's point of view.

Steps

The steps are the use case's internal structure. *Buy ticket* has a small number of steps; the methodology's discipline keeps them named, ordered, and visible.

1. Validate the request — well-formed identifiers, the seat location is real for the event's venue, no obvious structural problems.
2. Check that the event is still selling tickets.
3. Check that the customer is eligible to buy a ticket for this event.
4. Reserve the seat — claim it for this customer with a short time-bounded hold.
5. Authorize payment — ask the payment provider to commit the customer's money to this order.
6. Confirm the seat — convert the reservation hold into a permanent assignment.
7. Issue the ticket — create the customer's claim record.
8. Notify the customer — send the confirmation through whatever channel the customer prefers.

Eight steps. Each one named in domain terms. The reader looking at the use case implementation sees these names in order; the implementation is not hidden behind framework calls or scattered across helper classes. The methodology's structural commitment is that the steps are visible at the altitude they belong to, named in the domain's vocabulary.

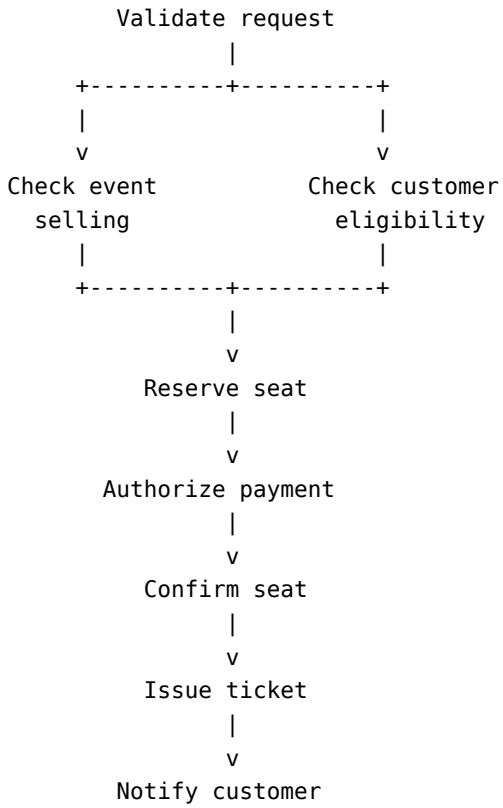
Dependencies between steps

The dependency graph is what makes the steps a composition rather than a list. Some steps must complete before others can begin. Some steps can run in parallel. Some steps are conditional on the outcome of prior steps.

For *buy ticket*:

- Validation must complete before any other step runs. An invalid request goes no further. The dependency is enforced structurally: validation produces a new type (`BuyTicket.ValidRequest`) only when the request is valid, and subsequent steps accept only this type. Invalid requests cannot reach later steps because no value of the required type can be constructed from them. This is the discipline Alexis King named *parse, don't validate* in 2019, applied at the use case's entry point — and the same pattern holds at every step boundary that follows: each step produces a new type whose existence is the precondition for the next.
- Event-selling check and customer-eligibility check are independent of each other; both depend only on validation. They can run in parallel.
- Seat reservation depends on event-selling and customer-eligibility both succeeding.

- Payment authorization depends on seat reservation succeeding. A reserved seat is the precondition for taking money.
- Seat confirmation depends on payment authorization succeeding.
- Ticket issuance depends on seat confirmation succeeding.
- Customer notification depends on ticket issuance succeeding.



That is the graph. It has two parallel branches early (event-selling and customer-eligibility), then a linear sequence. The graph is small enough to hold in the reader's head. It is also small enough that the implementation can be read and the graph can be reconstructed without external documentation.

The six properties are now stated. The use case is, in methodology terms, specified. Implementation follows.

Per-process types: where the methodology earns its first concrete commitment

The six properties named types: `BuyTicket.Request`, `BuyTicket.Response`, `BuyTicket.Failure`, plus the shared value objects `CustomerId`, `EventId`, `SeatLocation`, `TicketId`, `ReceiptId`. Inventing these types is where the methodology's process-first commitment becomes visible.

The commitment is small to state and consequential to apply: **types belong to processes, not to the domain.** A "customer" in *buy ticket* is a customer-buying-

something, identified by `CustomerId` and needing nothing else from the customer concept at this use case's level. A "customer" in customer-registration is a customer-being-created, with a different set of fields. A "customer" in customer-profile-update is a customer-with-mutable-state, with a third set. The same noun, three uses, three types.

In the entity-first tradition this collapse is the default: there is one `Customer` class, used everywhere, with whatever union of fields the various callers have asked for over the years. The result is well-documented and structurally consistent across teams that have used the pattern long enough: god-objects that no individual use case actually wants, or anemic carriers whose behavior has been pushed out into services that operate on the carriers from outside.

Process-first reverses the default. Each use case owns the types it needs. The shared layer is small — value objects that genuinely carry one piece of well-defined meaning across processes. The seat example makes this concrete.

The seat across three processes

A seat is a real domain concept. Three processes interact with it.

Buy ticket needs to know where the customer is sitting. The seat in this use case is a location — section, row, number, within a specific venue. The booking process records that this customer is in row L, seat 14, for the event being held. The information needed about the seat is its physical position. The behavior performed is selecting it and committing it to a customer.

```
SeatLocation:
    section: SectionId
    row: RowId
    number: SeatNumber
```

Check seat availability is a different process. It is not buying anything; it is answering "is this seat currently bookable for this event?" The seat in this process is an availability status — held until a certain time, currently for sale, sold, blocked, withdrawn from sale. The information needed is the seat's current state and, if held, when the hold expires.

```
SeatAvailability:
    seat: SeatId
    status: AvailabilityStatus
    holdExpiresAt: Option<Instant>
```

A hold may or may not be present; both cases are legitimate from the business perspective. The `Option<Instant>` expresses that fact explicitly: a seat for sale carries no hold expiration, a held seat does, and the type makes the difference unmissable rather than encoding it as a null or a sentinel instant.

Quote a price for a seat is a third process. Before the customer commits to a purchase, they ask: what does this specific seat at this specific event cost? The seat in this process is a price tier — premium, standard, economy, accessible, restricted-view — paired with the pricing rules for this event's tier schedule. The information needed

is the seat's price tier and the event's pricing rules; the physical location of the seat is irrelevant to pricing. Pricing is a per-tier concern, not a per-seat-location concern.

```
SeatPricing:
    seat: SeatId
    tier: PriceTier
    eventSchedule: PriceScheduleId
```

Three processes. Three different things called "seat." Different fields. Different operations. Different invariants. If a single shared `Seat` class is forced over all three uses, the class either grows every field every use ever needed (and no use wants all of it) or stays thin and pushes the differences into mapping code that has to evolve as each use evolves.

Process-first declines the shared class. The three processes each get their own type. The processes are connected at the persistence layer (there is a physical seat in a venue, persisted, that all three processes read from in their own projections) and at the identifier layer (`SeatId` travels between processes opaquely; possessing the identifier is not the same as possessing the seat's data).

What stays shared

The reframe is sometimes mistaken for a position that nothing should be shared. That is not the position. Some types are genuinely shared because they genuinely mean the same thing everywhere they appear.

`CustomerId`, `EventId`, `SeatId`, `TicketId` — opaque identifiers, used the same way wherever they appear. A typed wrapper around a string or a UUID. They travel; the things they identify do not need to travel as full objects with them.

`Money` — an amount and a currency, with arithmetic rules and comparison operations. The same `Money` in pricing means the same thing as the same `Money` in payment authorization. A shared `Money` value object is appropriate.

`Instant` — a point in time, in the system's chosen representation. Used wherever a time is mentioned.

These are small. Few fields. Well-defined invariants. No behavior beyond enforcing those invariants and supporting basic operations. They are value objects in the term-of-art sense — equality is by content, instances are immutable, construction is validated, the type carries one piece of domain meaning and no more.

The distinction the methodology makes is between *value objects* and *entities*. Value objects can be shared because they carry one well-defined piece of domain meaning. Entities — concepts that mean different things in different processes — cannot. The discipline is recognizing which is which, which is usually obvious once the question is asked: does this concept carry one piece of validated meaning across all uses, or does it carry different meanings in different uses? The first is a value object. The second is an entity, and it does not become a shared type just because the noun is the same.

Construction enforces the type's claim

A type that carries domain meaning makes a claim. A `CustomerId` claims to be a valid customer identifier. A `Money` claims to be a non-negative amount in a known currency. A `SeatLocation` claims to be a real position in a real venue.

The methodology asks that construction enforce the claim. A `CustomerId` cannot exist as an empty string. A `Money` cannot exist with a negative amount or an unknown currency. A `SeatLocation` is constructed by lookup against the venue's seating plan, not by accepting arbitrary section, row, and number values. Construction is the choke point; invalid values do not survive it.

Parse, don't validate, as the prior subsection cited. The construction site does the work once. Every caller that receives a constructed value can trust it. The use case's body is free of defensive checks because the types it operates on have already been validated at the boundary.

Implementation choices vary across languages — some compilers enforce these invariants directly via refinement types; others enforce through factory-method discipline. The Foundations section named four enforcement levels. The methodology's structural commitment is the same across all four; the teeth differ. *Buy ticket* in Java with JBCT uses the factory-method pattern: the type has a non-public constructor and a public static factory that returns `Result<T>`. The factory performs validation; success wraps a valid instance, failure carries the named cause. Callers must address both branches — the type system insists. Invalid instances cannot exist as values of the type because no caller can construct one without the factory's `Result`. Same discipline as in languages with refinement-type support; the teeth here are structural through the factory's `Result` return rather than syntactic through the type signature itself.

Composite construction accumulates. When `BuyTicket.ValidRequest` is built from independently validated fields (`CustomerId`, `EventId`, `SeatLocation`, each with its own factory returning `Result<T>`), the JBCT-canonical composition is

```
Result.all(customerIdResult,
           eventIdResult,
           seatLocationResult)
      .map(BuyTicket.ValidRequest::validRequest);
```

The composition collects per-field failures rather than short-circuiting at the first; a caller receiving the structured rejection sees every invalid field at once, which is what a customer-facing client needs to render. Async equivalents (`Promise.all`, `Promise.allOrCancel`) accumulate the same way for I/O-touching factories.

Types evolve within the use case

Per-process types are not static. Inside the use case body the types appear in sequence: each step accepts the prior step's typed output and produces a new typed value that the next step accepts. `BuyTicket.ValidRequest` is the entry condition; downstream steps produce a `Reservation`, then an `Authorization`, then a `ConfirmedSeat`, then a `Ticket`, then a `NotifiedTicket`. The sequence of types *is* the use case's spine. A reader scanning the steps' return types sees what each step contributes.

This pattern is canonical in JBCT under the name **growing context**: each pipeline stage receives the prior context, adds information, and passes an enriched context forward. The discipline is structural (every step's output type is the precondition for the next step's input), so the compiler enforces the order. Reordering the chain accidentally produces a type error at the first misaligned step, not a behavioral surprise at runtime. Per-process types are not just a between-processes commitment; they are the within-process record of growing knowledge.

The same discipline lets the Fork-Join at the front of the use case carry information forward rather than acting as a pure gate. The event-selling and customer-eligibility checks each succeed with a typed value (the event's current sale state, the customer's booking context) that combines with the validated request into the reservation context the next step consumes. Success enriches; failure carries the typed `BuyTicket.Failure` variant that propagates to the caller and short-circuits the chain. The pattern holds at every step boundary.

Composition: the six patterns at use-case altitude

The methodology names six composition primitives: Leaf, Sequencer, Fork-Join, Condition, Iteration, Aspects. The claim from the Foundations section is that the same six apply at every altitude. At use-case altitude, some of them show up immediately; some are present but spare; some do not earn their place until later altitudes. Which appear and which do not is itself information about the use case.

Sequencer

A Sequencer is the most common pattern at use-case altitude. *Buy ticket*'s linear spine (reserve, then authorize, then confirm, then issue, then notify) is a Sequencer. Each step's output feeds the next; failure at any step short-circuits the rest.

The implementation reads as a chain of typed operations. Each step takes the prior step's output as input. Each step is a Leaf to an external resource (or composes other Leaves) and therefore returns `Promise<T>`. The chain composes through `flatMap`, which short-circuits at the first failure — the failure variant propagates to the use case's caller without further steps running.

```
return reserveSeat.apply(validRequest)
                  .flatMap(authorizePayment::apply)
                  .flatMap(confirmSeat::apply)
                  .flatMap(issueTicket::apply)
                  .flatMap(notifyCustomer::apply);
```

The shape is recognizable. The reader following the chain sees the use case's spine. Each step is named in domain terms. The failure mode at each step is part of the step's typed return, not raised as an exception that disappears up the stack.

The chain stays short. JBCT's monadic-chain discipline caps direct chains at four or five steps before extracting named sub-Sequencers. *Buy ticket*'s full pipeline (validation, the Fork-Join checks, this five-step booking spine) exceeds that limit as one inline

chain; the use case body composes named sub-pipelines instead, each itself a short Sequencer that the outer Sequencer flatMaps over.

Sequencer appears in every non-trivial use case at this altitude. If a use case has no Sequencer it is doing a single thing, and the methodology's response is to ask whether it is actually a use case at all or something smaller — a query, a probe, a leaf operation.

Condition

A Condition is the methodology's name for branching on a business decision — a domain fact that selects between alternative continuations, both of which are legitimate. The Condition pattern routes on values like *which discount tier applies to this purchase*, *which shipping method does this order use*, *do these miles get added to the customer's account at the standard rate or the promotion rate*. The use case is asking which path this case takes.

For *buy ticket* as described, explicit Condition is rare. The use case is mostly Sequencer + Leaf + Fork-Join: validate, check, reserve, authorize, confirm, issue, notify. Each step has a clear next step on success; the use case is not differentiated by branching business rules at this altitude.

A Condition would appear if the use case branched on, say, the event's sale type: member-restricted events take a membership-check path before reserving; general-sale events skip the membership check. The methodology accommodates such variations cleanly — the branch is encoded as a typed sum (`EventSaleType.MemberOnly | EventSaleType.GeneralSale`), and the use case dispatches on the variant.

Other natural Conditions in the booking domain include applying a customer's eligible discount tier (student, senior, military), routing on whether a valid promo code is present, and triggering a group-discount path when the booking covers a party of N or more. Each is a branch on a domain fact selecting between alternative continuations — different price calculations, different validation rules.

The discipline is keeping the branch typed. A boolean rarely carries enough domain meaning to make the branch self-describing at the call site; the typed sum carries the business fact the branch depends on. `EventSaleType.MemberOnly` says what is being decided; `true` says only that something was decided.

Condition earns more visibility at workflow altitude (Pass 2), where workflows compose around branching business decisions: which fulfillment flow this booking follows, which refund policy applies to this cancellation, which notification channel reaches this customer best.

Leaf

A Leaf is the methodology's name for an atomic execution unit. It is the bottom of composition — either a boundary crossing (I/O, external library call, non-deterministic API) or a pure computation (deterministic input-to-output transformation with no side effects). Anything that composes other operations is a different pattern (Sequencer, Fork-Join, Condition); anything that has hidden state is forbidden in business code. *Buy ticket* has several Leaves, almost all of them boundary crossings:

- Reading the venue's seating plan to validate the seat location.

- Reading the event's selling-status from the event-management subsystem.
- Reading the customer's eligibility state from the customer subsystem.
- Calling the seat-reservation store to claim the seat with a hold.
- Calling the payment provider to authorize the customer's payment method.
- Calling the seat-reservation store again to convert the hold to a permanent assignment.
- Writing the ticket to the ticket store.
- Calling the notification service to send the customer confirmation.

Each one is a Leaf. The use case's body composes them; the body itself is not a Leaf. The discipline at this altitude is that Leaves are named in domain terms (`ReservationStore.claim`, `PaymentProvider.authorize`) rather than technical terms (`Database.execute`, `Http.post`). The technical detail lives behind the domain-named interface. The use case body knows nothing about whether the reservation store is a Postgres table, a Redis key, a remote service, or a hybrid; it knows only that it can ask the reservation store to claim a seat.

The cross-cutting distinction the Foundations section drew applies cleanly here. The use case's body operates in a quarantined zone where only domain concepts appear. Technical concerns live one step away, behind domain-named interfaces. The reader of the use case body sees business logic uninterrupted by framework or infrastructure concerns.

Fork-Join

A Fork-Join is the pattern for parallel steps that converge. *Buy ticket* has one natural Fork-Join: the event-selling check and the customer-eligibility check are independent. Both depend on validation completing; neither depends on the other. They can run concurrently and their results join before reservation begins. The Fork-Join also grows context, as the prior subsection named: each check on success returns a typed status that the reservation step needs.

```
return Promise.all(
        checkEventSelling.apply(validRequest.event()),
        checkCustomerEligibility.apply(
                validRequest.customer(),
                validRequest.event()))
    .map((selling, eligibility) ->
            ReservationContext.reservationContext(
                    validRequest, selling, eligibility))
    .flatMap(reserveSeat::apply);
```

`checkEventSelling` returns `Promise<EventSaleStatus>`; `checkCustomerEligibility` returns `Promise<CustomerBookingContext>`. `Promise.all` runs both concurrently and joins them into a tuple, propagating either's typed failure if it occurs. The `.map` builds the next step's context: `ReservationContext` combines the validated request with both statuses. `reserveSeat::apply` is a single-arg method reference taking that context. The step interface stays single-method, single-arg, as the methodology asks; the growing context absorbs what the Fork-Join produced.

The implementation depends on the runtime's concurrency primitives (futures, promises, fibers, threads, whatever the language and stack provide). The methodology's commitment is independent of the primitive: the pattern is Fork-Join, the implementation honors what the host language calls it.

Fork-Join at use-case altitude is sometimes present, sometimes absent. *Buy ticket* has one occurrence; some use cases have none. Its presence depends on what is genuinely parallelizable in the use case's step graph, and that depends on the domain, not on the developer's preference.

Iteration

Iteration is the pattern for applying a step to a collection. Whether it appears at use-case altitude depends on the use case's shape. Use cases that operate on multi-item structures — checkout flows reserving stock for each line item in an order, multi-seat bookings, bulk invoice processing, order-validation passes — naturally include Iteration at use-case altitude. The trigger is still one event and the outcome is still one response, but the work inside includes "do this for each item."

Buy ticket as defined here does not iterate, because the use case handles a single ticket. The absence is a property of this specific use case, not of use-case altitude in general. A close variant — *buy multiple tickets in a single order* — would iterate naturally over the seat selections, reserving each one in turn.

Iteration also appears at higher altitudes, where the iteration is *over use cases* — a workflow processing many bookings, a subsystem handling daily batches. Both forms compose through the same primitive.

Aspects

Aspects are cross-cutting concerns: things that apply uniformly across many operations rather than belonging to any specific operation. Logging, retries, distributed tracing, correlation identifiers, idempotency-key handling, audit trail emission, circuit-breaking — these are all Aspects.

At use-case altitude, Aspects are present but mostly handled by the runtime rather than written into the use case body. The Foundations section made the distinction between business cross-cutting and technical cross-cutting. *Buy ticket* itself does not log; the runtime around it does. *Buy ticket* itself does not trace request paths; the runtime injects tracing. *Buy ticket* itself does not handle retries against the payment provider; the payment provider's Leaf wrapper handles retries policy, and the use case body sees only the typed outcome.

When multiple Aspects wrap a single operation their composition order matters, but at use-case altitude the runtime supplies the order and the use case body sees only the typed outcome. The convention itself — which Aspect wraps which, and why — earns its place at workflow altitude, where Aspects become load-bearing, and is settled fully in the Architecture Synthesis module.

The use case body is therefore short. It contains domain logic and nothing else. Aspects are visible in the use case's behavior at runtime (the logs appear, the traces propagate, the retries happen) without being visible in the use case's source. The

discipline that produces this separation is the quarantine principle plus the Leaf interface: business code calls domain-named Leaves; Leaves carry whatever technical Aspects the operation needs.

Aspects earn more visibility at higher altitudes. At subsystem altitude, business cross-cutting (audit-as-data versus audit-as-Aspect, regulatory compliance hooks) becomes load-bearing and the methodology has to make explicit choices about where the cross-cutting lives. At use-case altitude, the choices are not yet forced.

That deferral is not a reason to undervalue the separation; it is load-bearing. The composition skeleton — the Sequencer spine, the Fork-Join, the steps that compute rather than call out — is this use case's business logic in pure form, and it is testable as such, exercised against stand-in Leaves with no database, payment provider, or clock in sight. "Pure" here is meant in the business sense rather than the functional-programming one: the skeleton is a faithful account of how this use case processes, independent of the machinery that runs it.

Because the skeleton is independent of its Leaves, the same skeleton attaches to different Leaf implementations to become a running application: real adapters in production, doubles under test. That attachment is also where technical instrumentation enters. Because every Leaf and every injected dependency crosses the same uniform seam, the runtime can wrap them with logging, tracing, and metrics automatically, and the use case body names none of it. This is the technical cross-cutting the runtime already owns; the skeleton-and-Leaf structure is what lets the wrapping be applied systematically rather than written by hand. The wiring itself is assembly-time work, taken up at system altitude.

What the pattern distribution shows

Buy ticket uses Sequencer (strongly), Leaf (frequently, at every boundary), Fork-Join (once), Condition (rare or absent), Iteration (absent), Aspects (handled by runtime). Three patterns are strongly present; Condition and Iteration are absent in this specific use case; Aspects live in the runtime.

This is the distribution this use case earned. A different use case at the same altitude will have a different distribution. *Check seat availability* is mostly Leaf with a small Condition — read the seat's current state, branch on the variant. *Refund ticket* is mostly Sequencer plus Condition (full refund vs partial refund vs courtesy waiver) with one Leaf to the payment provider. *Buy multi-seat order* puts Iteration at the centre (reserve each seat in turn) alongside the same Sequencer spine. No use case at this altitude has every pattern at high frequency; the distribution carries information about the use case's shape and the business rules it embeds.

The methodology does not prescribe which patterns appear. The methodology supplies the patterns and lets the use case earn which ones show up. The reader inspecting a use case body recognizes the patterns by their shape and can reason about the use case's structure by which patterns are present and which are absent.

The four shapes earn their place

The Foundations section named four shapes that types carry: `T` (the value exists unconditionally), `Option<T>` (the value may or may not exist), `Result<T>` (the operation may or may not have produced the value), `Promise<T>` (the value arrives later). Each shape carries a domain modality. The shapes are not stylistic; they are how the type system represents what the domain says about a value.

Buy ticket makes each shape concrete.

T — the unconditional value

Several values in *buy ticket* are unconditional. They are present, they are valid, they are not waiting on anything. The validated request after construction is a `BuyTicket.ValidRequest` — not an `Option<BuyTicket.ValidRequest>`, not a `Result<BuyTicket.ValidRequest>`, just a `BuyTicket.ValidRequest`. Construction either produced it or rejected the inputs at the boundary, but downstream code receives the unconditional value.

A `CustomerId` once constructed is a `CustomerId`. A `Money` once constructed is a `Money`. A `SectionId` is a `SectionId`. These are baseline types. Wrapping them in additional containers when they cannot fail is ceremony without payload. The Foundations section named this discipline; *buy ticket* shows it in practice.

Option — the absent value

`Option` carries the absence-as-domain-fact modality. Some values legitimately may not exist. *Buy ticket* has a few:

- A seat's current hold expires at some instant, *if* the seat is currently held. If the seat is not held, there is no hold expiration. The field is `Option<Instant>`, not `Instant`. A reader looking at the field knows the absence is a domain fact, not an error condition.
- The customer's preferred notification channel might or might not be set. If unset, the use case uses a fallback. `Option<NotificationChannel>` carries the design fact.

`Option` is used where the absence carries domain meaning. Where absence would indicate a programming error (a value the use case expects to always be there but might be missing because of a bug), the answer is not `Option`; the answer is type discipline that prevents the value from being missing. `Option` is a domain claim, not a defensive catch-all.

Result — the synchronous failure modality

`Result<T>` carries failure-as-typed-outcome for operations that complete synchronously and do not touch I/O. *Buy ticket* itself returns `Promise<BuyTicket.Response>` because it performs I/O at almost every step (covered next), but `Result<T>` still earns its place inside the use case wherever a step is purely computational: pre-validation of the request structure, derivation of intermediate values from already-loaded data, structural checks against the typed input. Those steps return `Result<T>`, and the surrounding `Promise<T>` composition lifts them into the async chain where the next step is an I/O Leaf.

`Result<T>` and `Promise<T>` are not redundant. They name different modalities (synchronous-may-fail and asynchronous-may-fail), and the methodology asks the designer to be explicit about which the step is. A Sequencer of `Result<T>` operations stays synchronous; a Sequencer involving any `Promise<T>` propagates `Promise<T>` outward. The composition primitives handle the lifting uniformly, so the use case body composes both shapes without ceremony.

Promise — the asynchronous shape that includes failure

`Promise<T>` carries both the async modality and the failure modality. *Buy ticket* has multiple Leaves that touch external resources: the payment provider authorization, the reservation store calls, the ticket store writes, the notification dispatch. Each is I/O, each is asynchronous, each returns `Promise<T>`, and `T` already includes the failure case because `Promise` resolves to either a success value or a typed failure.

The propagation is automatic. Any step that depends on an async Leaf is itself in a `Promise` context, and the compositions absorb it: a Sequencer chains `Promise<T>` values via `flatMap`; the runtime handles the async sequencing while the type carries the failure modality. The reader sees the chain and reads it as a sequence; the underlying machinery does what the language and runtime supply.

The implementation depends on what the host language calls async (futures, promises, fibers, virtual threads, suspend functions, async/await). The methodology's commitment is independent of the implementation primitive. The shape is `Promise<T>`; the runtime supplies the concurrency.

Why these four

Four shapes cover the design space use-case-altitude operations care about: the value exists or it does not (T or Option), the operation succeeded or failed (T or Result), the value arrives now or later (T or Promise). Other traditions add more — list types, stream types, effect-tracking types — but the methodology declines them: four shapes are sufficient, and each additional shape is a cost the reader pays at every signature. The payoff is a signature that reads directly as domain modality. `Result<TicketId>` is a synchronous operation producing a ticket identifier or one of the named failures; `Promise<TicketId>` is the same operation asynchronously, with failure already folded in.

Recovery: BER applies cleanly at this altitude

The Foundations section named three recovery classes: BER (Backward Error Recovery — compensate via inverse action), FER (Forward Error Recovery — continue with degraded state), and design-out (alter the shape so compensation is unnecessary). At use-case altitude, BER applies cleanly to almost everything, with one design-out move that the methodology's structural choices have already made.

What can fail and what compensation looks like

Failure points in *buy ticket*:

- **Validation fails.** No state has changed. Return the structured rejection. No compensation needed because nothing happened.
- **Event-selling check fails.** No state has changed. Return the structured rejection. No compensation needed.
- **Customer-eligibility check fails.** Same as above.
- **Seat reservation fails because the seat is unavailable.** No state has changed. Return the structured rejection. No compensation needed.
- **Payment authorization fails.** The seat reservation exists — a hold has been placed on the seat. The state must be undone. Release the hold. BER applies; the inverse is mechanical.
- **Seat confirmation fails after payment authorization succeeds.** Payment has been authorized but not captured. Release the hold; void or reverse the authorization. Both are mechanical inverses on the domains they affect.
- **Ticket issuance fails after seat confirmation succeeds.** The seat is now committed and payment is authorized; the customer's ticket record does not exist. Reverse the confirmation; void the authorization; release any residual state. The inverses are still mostly mechanical, but the surface area grows with each successful step that has to be undone.
- **Notification fails after ticket issuance succeeds.** The ticket has been issued. The customer does not need to be notified for the booking itself to be valid; the failure is logged for reconciliation and the use case returns success. FER could apply here, with degraded state being "ticket issued, notification queued for retry."

The pattern is: failures early in the sequence cost nothing because nothing has happened. Failures late in the sequence cost the inverse of every successful step. The Sequencer makes this visible; the reader following the chain can identify exactly which inverses are needed at which failure points.

Each inverse is mechanical at this altitude

Releasing a seat hold is the inverse of placing a seat hold. Both operations live in the same domain (the reservation store); the inverse is the operation's reverse and does not require a separate workflow to execute. The methodology calls this **domain-internal compensation**: the inverse stays inside the step's own domain.

Voiding a payment authorization is similar. The payment provider supports an inverse operation (`void` or `reverse`) that maps directly to the authorization. The inverse is mechanical in the same sense — call the inverse method on the same domain interface.

Reversing a seat confirmation (turning a confirmed assignment back into an unclaimed state) is again domain-internal. The reservation store supports this transition explicitly because the methodology asked the designer to think about it.

The discipline this surfaces is **identifying compensation requirements at design time**. When the use case is being specified, the question "what is the inverse of this step?" is asked for every step that produces state. Steps without inverses are flagged. If a step's effect cannot be undone (sending an email, charging a card without a reverse path, dispatching a physical item), the methodology surfaces that fact and the team decides whether the step belongs in this use case or whether the use case's structure needs to change.

Design-out is already in play

Buy ticket makes one design-out move that the methodology's structural choices have already taken: the seat reservation model. The use case does not naively check seat availability and then attempt to claim it (which would race against other booking attempts). The use case reserves the seat with a short time-bounded hold *before* attempting payment. The hold prevents two booking attempts from succeeding for the same seat; only one hold can exist at a time per seat.

This design-out move means "two customers attempting to book the same seat at the same time" does not need BER. The reservation model prevents the conflict from producing two inconsistent confirmed bookings. One reservation succeeds; the other's reservation attempt fails immediately with `SeatUnavailable`. There is no compensation needed for the failed attempt because the failed attempt never made any state change beyond the failed reservation try, which the reservation store handles atomically.

Design-out at use-case altitude tends to be exactly this kind of structural choice: pick a domain model where the conflicting case cannot arise. The methodology's discipline asks the designer to look for design-out opportunities before committing to compensation. If a domain model exists where the conflict is structurally impossible, that model is usually cheaper than the compensation logic that would otherwise be needed.

FER is not yet earned

Forward Error Recovery means continuing with degraded state. At use-case altitude FER applies sparingly because there is little state to degrade. The single FER candidate in *buy ticket* is the notification step — the booking can succeed even if the notification fails, with the failure logged for retry. That is a small FER application, applied at the edge of the use case where the customer's claim to the seat is already committed and the notification is a downstream concern.

The full FER discourse (time-as-decay, defaults under partial failure, self-stabilizing flows) does not earn its place until workflow altitude. At workflow altitude, multiple use cases compose, telemetry streams persist across compositions, and the design space for degraded states opens. At use-case altitude, the design space is small: the use case either succeeds and produces its output, or fails and (after compensation if needed) leaves no inconsistent state behind.

Recovery class selection at this altitude

The Foundations section named four selection axes: reversibility, forward-progress value, domain shape, coordination cost. At use-case altitude, the choice is mostly driven by the first two.

- **Reversibility.** Steps that produce reversible state (seat holds, payment authorizations, draft records) admit BER cleanly. Steps that produce irreversible state (sent emails, dispatched physical items, posted ledger entries) need design-out or accept the irreversibility.
- **Forward-progress value.** When partial success is more valuable than rollback, FER applies. The notification example is the small case at use-case altitude.

Domain shape (the third axis) determines which design-out moves are available, but at use-case altitude the design-out moves are typically built into the methodology's vocabulary (the reservation model is one such move). Coordination cost (the fourth axis) does not bite at use-case altitude because everything happens within one use case's scope.

The full selection mechanism (with all four axes weighed against each other, with mixed strategies across a workflow) lives in the Architecture Synthesis module that follows the spiral. At use-case altitude, the selection is straightforward: BER for steps with mechanical inverses, design-out where the domain model can prevent the conflict, FER for downstream effects whose failure does not invalidate the use case's primary outcome.

Architecture surfaces at this altitude

Phase-5 architecture-vector selection operates across six axes (deployment topology, composition substrate, read/write model, state storage, persistence layer configuration, recovery class), fully developed in the Architecture Synthesis module. At use-case altitude, some of these axes have decisions to make; some do not yet.

Per-use-case SLOs

The architecture decision that surfaces immediately at use-case altitude is the service-level objective (SLO) triple per use case: latency, throughput, availability. *Buy ticket* needs to respond to the customer in seconds, not minutes. It needs to handle the venue's peak booking rate, which for a popular event can be thousands of attempts per second in the first minutes of sale. It needs to remain available during exactly those peak periods.

These SLOs do not appear out of nowhere. They come from the business's understanding of what the use case is for. They are inputs to architecture, not outputs. The Phase-4 elicitation that the Architecture Synthesis module fully treats begins here: what does *this* use case need to deliver in operational terms? Answer that, in the use case's own scope, and the answer constrains the architecture choices that follow.

For *buy ticket*:

- **Latency.** P50 under 2 seconds, P99 under 5 seconds. The customer is waiting.
- **Throughput.** 1,000 attempts per second sustained during peak; 5,000 attempts per second burst.
- **Availability.** 99.95% during business hours, accepting brief degradation during scheduled maintenance.

These are concrete numbers. They are not what every use case in the system needs; *check seat availability* might have different numbers, *refund ticket* might have different numbers, *send venue capacity report* might have different numbers. Phase 4 attaches answers at three scopes (per use case, per data class, per domain), developed in the Architecture Synthesis module. Latency, throughput, and availability are per-use-case answers.

Beyond the SLO triple, the use case makes another architecture-level choice: the in-flight failure response mode. Three modes are available: synchronous fail (return the typed failure to the caller immediately), retry-with-backoff (within the use case's latency budget, transient failures retry transparently inside a Leaf), and degraded mode (return a partial success when the primary outcome cannot be achieved). For *buy ticket* the response is mostly synchronous fail; retry-with-backoff lives inside the payment-provider Leaf for transient provider errors and is invisible to the use case body; degraded mode does not apply because there is no meaningful partial success of "buying a ticket."

What is deferred

Several architecture questions do not surface at use-case altitude. They are real questions, but they do not yet have anything to decide against because the multiplicity that would force them has not arrived.

- **Persistence topology.** Does the reservation store live in a shared database with other subsystems' data, or in its own? At one use case, the question has no force. Multiple use cases across multiple workflows make the question concrete.
- **Deployment topology.** Is *buy ticket* deployed as part of a monolith, as one slice of a modulith, as its own microservice, as a function-as-a-service, or as an Aether slice on a unified runtime? At one use case, any topology works. The choice has weight when many use cases are deployed together and operational concerns (independent scaling, blast-radius isolation, deployment frequency) start to matter.
- **Cross-process consistency.** When *buy ticket* writes the ticket and another use case reads it, what consistency is required? At one use case in isolation, the question is moot. Composition with other use cases — query the customer's tickets, list event attendees, calculate venue revenue — forces it.
- **Composition substrate.** Are use cases composed directly, each calling the next and using its result, or through events that one publishes and another reacts to, or some hybrid? At one use case the trigger is fixed (synchronous HTTP); the composition question only appears when use cases call other use cases.

These deferrals are not omissions. They are honest about what the altitude has earned. The Architecture Synthesis module fully treats each axis after the spiral has shown what each altitude surfaces. The deferral here means: the question exists, the answer comes when the methodology has enough multiplicity to drive it.

Closing — multiplicity is coming

This pass has shown one use case. The methodology applied to it produced a typed input, a typed output, an enumerable failure set, a small ordered sequence of steps with explicit dependencies, per-process types specific to this use case's needs, value objects where genuine sharing applies, the composition patterns that appear at this altitude (and the ones that do not), the four shapes earning their place, and a recovery class that applies cleanly to local failures.

The methodology did not produce the design. A team designing *buy ticket* made choices. The methodology made those choices visible, named them, and kept the resulting code legible.

The next pass starts where this one ends. *Buy ticket* is one use case. The system that contains it is not just *buy ticket*. The customer who buys a ticket may want to cancel it. The customer may want a refund. The customer may want to change their seat. The customer may want to check seat availability without committing to a purchase. The customer may want to hold a seat temporarily while consulting friends. The venue operator may want to release blocked seats for last-minute sale. The pricing system may want to update tier prices based on demand. Each of these is a use case in its own right.

The use cases share resources. Multiple use cases interact with the same seat. Multiple use cases interact with the same customer. Multiple use cases produce events that other use cases consume. The composition of these use cases is not just a list; it is structured. Workflows emerge from the structure.

At workflow altitude, the methodology applies again — and this is where the patterns that stayed quiet here come alive. Iteration and Aspects earn their first strong appearances, and recovery-class selection grows more interesting as the design space opens: compensation across use cases, time-as-decay as a first-class concern, saga as a recognizable composite rather than a primitive.

The next pass walks through that.