Beyond Code – Building Software Systems That Last

Vijay Mathew

March 16, 2026

Beyond Code

Building Software Systems That Last

Beyond Code

Building Software Systems That Last

From Modeling and Algorithms to Reliability, Evolution, and AI-Era Engineering

Vijay Mathew

March 16, 2026

Table of contents

Foreword: What This Book Is For

There is no shortage of programming books.

There are books that teach Python with clarity and patience. Books that explain React, or Rails, or the internals of Postgres. Books devoted to functional programming, to object-oriented design, to the architecture of distributed systems. For almost any technology a working programmer might encounter, a competent guide exists.

And the serious books exist too. *Code Complete* has guided developers through the full arc of software construction for thirty years. *A Philosophy of Software Design* makes a careful argument about what separates good design from bad. *The Pragmatic Programmer* addresses the craft of software development with breadth and honesty. SICP remains, decades after its publication, one of the finest demonstrations of how to think computationally. These are real books that have made real differences to the engineers who read them.

So the claim this book makes is not that the field has been neglected. It has not.

The claim is more specific: there is a synthesis that is difficult to find in a single place.

What Is Hard to Find

Algorithms are taught in one course. Data structures in another. Design patterns in a third. System design in a fourth. Testing in a fifth. Each subject has its own textbook, its own examples, its own vocabulary. A diligent student can work through all of them and still emerge without a clear picture of how they fit together — because the connections between them are rarely the subject of explicit instruction.

Those connections are where the most important engineering decisions live.

The choice of algorithm depends on the cost model. The cost model depends on the data representation. The data representation depends on the queries the system must answer. The queries depend on the problem statement. The problem statement depends on understanding the real situation the software must address. These dependencies run in both directions: a good algorithm choice can reveal that the data model needs revision, and a good data model can make a complex algorithm unnecessary.

Teaching these as separate subjects leaves the synthesis to the student — to be assembled, if at all, through years of practice and a fair amount of failure. Most books address one part of the chain with depth and leave the rest implicit. The books that address the whole tend to stay at altitude, treating process and methodology rather than the technical substance of engineering decisions.

What is genuinely rare is a book that:

- treats algorithms, data structures, contracts, design, testing, and debugging as a single connected argument rather than adjacent topics
- follows that argument through the same running systems, chapter after chapter, so the reader sees how each tool applies to the same problem at different levels of abstraction
- takes the specification-to-implementation pipeline seriously as the central discipline — the idea that the hardest engineering work happens before the first line of code, in the precise articulation of what the system must do and what must always remain true

This book is an attempt to provide that synthesis.

What This Book Does Differently

It does not start with a language. It starts with a problem: how do you look at a messy situation and identify what the software must actually do? How do you write a specification that is precise enough to build from, but honest enough to acknowledge what is not yet known? These are the questions of Part I, and they precede any code.

It does not start with a paradigm. Functional thinking and object-oriented thinking are both tools, and this book treats them as tools — each suited to certain problems, each with characteristic failure modes, neither universally correct. Part VI introduces both in the same breath, as complementary strategies for organising software, and lets the problem determine which applies.

It does not restrict itself to a domain. The three systems that run through this book — a delivery network, a knowledge engine, a virtual world — are chosen precisely because they are different. Different domains, different data structures, different algorithmic requirements, different failure modes. Following the same engineering questions through all three reveals what transfers and what does not. That transfer is the skill. It is what distinguishes a developer who has learned one framework from an engineer who can reason about systems they have never seen before.

Who This Book Is For

This book is for developers who have learned to write code and want to learn to engineer software.

If you can write a working program but would not confidently recognise a fragile design before it became a production incident, this book is for you. If you have studied algorithms but are not sure how to choose between them when building a real system, this book is for you. If the pieces of software engineering — the design thinking, the data structures, the testing discipline, the component boundaries — exist separately in your mind and have not yet assembled into a coherent whole, this book is for you.

It is also — unapologetically — for students. The engineering habits this book teaches are not advanced topics to be introduced after the fundamentals are mastered. They are the fundamentals. A student who learns from the beginning to write precise problem statements, to model systems with explicit assumptions, and to verify behavior through contracts will write better code earlier and understand why it is better — not just that it passes the tests.

And it is for developers navigating the new reality of AI-assisted programming. When code becomes easy to generate, the ability to evaluate what has been generated becomes more important, not less. A developer who cannot identify when a suggested algorithm is wrong for the problem, who cannot recognise a fragile abstraction before it is built upon, who cannot read generated code against a precise specification — that developer is not made more capable by AI tools. They are made faster at producing problems they cannot diagnose. This book is for the developer who wants to remain the engineer, not become the reviewer of an AI's guesses.

A Note on the Language

The code examples in this book use **Nex**, a language designed to make good engineering habits the path of least resistance. Contracts, invariants, and explicit behavioral guarantees are not features you add to Nex programs — they are the natural way Nex programs are written. A precondition is not a comment that might drift out of date. It is an executable part of the program, checked at the call boundary, reported with its label when violated.

This is a deliberate choice. Learning to think about preconditions, postconditions, and invariants is easier when the language makes those concepts visible and enforceable rather than optional and advisory. Nex draws on Eiffel's Design by Contract philosophy and generates production-quality output for the JVM and for JavaScript, so the discipline it encourages is not confined to exercises — it carries forward into real systems.

Nex is the medium. The engineering principles are the message. Both transfer directly to any serious language or environment you work in after this book.

You do not need a complex toolchain to begin. A local Nex REPL is enough for all the examples that follow.

Before the Prologue

The chapter that follows introduces three systems and three failures. Those failures are not cautionary tales included for dramatic effect. They are the opening of an argument that runs through the entire book: that the most consequential engineering decisions are made before a line of code is written, that the hardest problems in software are problems of clarity rather than cleverness, and that the skills required to get those decisions right are learnable, transferable, and more important now than they have ever been.

The prologue begins with a robot that will not move.

That is where the engineering starts.

Prologue — The Day the System Broke

At 9:12 on a Monday morning, a delivery robot stopped in the middle of an intersection and refused to move.

It had a package. It had a destination. It had a full battery.

But it would not move.

The control system reported: **no valid route forward**.

Across town, a graduate student searched for a paper she had saved two months earlier. The system returned **12,483 results**. She had built the note graph carefully: tags, backlinks, categories, references, aliases. Everything was there. Nothing was findable.

In a startup office on the third floor of a building with bad coffee and good ambition, a game engineer watched a virtual economy collapse in real time. Players had discovered an item-duplication path. Within minutes, rarity meant nothing.

Three different systems. Three different domains. One shared failure pattern.

The failure was not a typo. It was not a missing semicolon. It was not solved by writing more code.

The break happened earlier.

What Actually Failed

Each team had competent engineers and working software. Early demos looked excellent.

Then scale arrived.

The delivery network moved from dozens of routes to thousands. Local path choices started conflicting globally. Robots blocked each other. Resolution logic became brittle.

The knowledge system moved from hundreds of notes to tens of thousands of linked entities. Connection density exploded. Search degraded from guidance to noise.

The virtual world moved from a few hundred active players to a persistent economy with emergent behavior. State transitions that looked safe in isolation were unsafe in combination.

In each case, code quality mattered, but it was not the primary bottleneck.

The bottleneck was design clarity:

- unclear problem boundaries
- weak models of entities and relationships
- under-specified rules that should always hold
- algorithms chosen without enough attention to growth behavior

These are engineering failures before they are programming failures.

Why This Book Exists

Most people learn programming in a sequence like this:

1. Learn syntax.
2. Write small functions.
3. Debug until tests pass.

That path is useful, but incomplete.

Small exercises hide the hardest part of software engineering: deciding what system should exist in the first place, and how it should behave when the world gets messy.

This book starts one layer earlier.

You will still write code. You will still learn algorithms and data structures. But the central skill we build is this:

turning ambiguous real-world problems into precise, durable software designs.

The Thread Through The Whole Book

We will follow three recurring systems:

- a delivery network
- a knowledge engine
- a virtual world

They look unrelated on the surface. They are not.

All three force the same engineering questions:

- What are the core entities?
- How do entities relate and change over time?

- Which invariants must never be violated?
- What operations must be fast, and at what scale?
- Where do local decisions create global failures?

By revisiting the same systems across many chapters, you will learn transfer: one design idea applied across multiple domains.

That is what expert engineers do.

How To Read This Book

This is not a reference manual. It is a progression.

Each part builds a layer:

- **Part I**: See the problem clearly before coding.
- **Part II**: Model the world with entities, relationships, and change.
- **Part III–V**: Design and evaluate algorithms and data structures.
- **Part VI**: Organize software into components and interfaces.
- **Part VII**: Make systems trustworthy with contracts, invariants, tests, and debugging.
- **Part VIII**: Evolve systems without collapse.
- **Part IX**: Work effectively with AI coding tools while keeping human judgment central.

Every section is aimed at one practical outcome: better engineering decisions under real constraints.

Programming in the AI Era

AI can generate code quickly.

That changes how software is written, but it does not eliminate engineering.

AI can draft implementations. It can suggest refactorings. It can translate ideas from one language or style to another.

What it cannot reliably do is the full work of software design:

- identifying the **right problem**
- choosing the **right model of the world**
- defining clear **contracts, invariants, and guarantees**
- balancing **correctness, performance, and evolvability**
- taking responsibility when systems fail in production

Those responsibilities remain fundamentally human.

In fact, AI makes them **more important**, not less.

When code becomes cheap to generate, the real risk shifts elsewhere. Poorly framed problems produce large amounts of useless code. Weak abstractions multiply quickly. Fragile systems grow faster.

Fast code generation **amplifies architecture —** both good and bad.

This book is written for that reality.

It focuses on the skills that remain essential even in a world where code itself is increasingly easy to produce:

- understanding problems clearly
- modeling systems carefully
- designing algorithms deliberately
- building software that remains correct and adaptable as it grows

In other words, it is a book about **thinking like a software engineer** in the age of AI.

Why This Book Uses Nex

You do not need a popular language to learn durable engineering thinking.

This book uses **Nex** as its implementation language on purpose.

Nex is not mainstream, but it gives us a clean way to study core software ideas without carrying unnecessary ecosystem complexity in early chapters.

Most importantly, Nex was designed to teach good software engineering practices that transfer directly to real-world systems.

Nex is especially useful for this book because it supports:

- **Functional and object-oriented styles** in the same language, so we can compare design tradeoffs directly.
- **High-level contracts and invariants**, so correctness rules can be expressed in the code where they belong.
- **Explorative programming**, including optional dynamic typing when rapid experimentation is useful.
- **Graphics support**, which helps us model visible system behavior (simulation, movement, interaction) instead of only text output.
- A **web-based IDE**, so readers can start immediately with no local installation required.

In other words, Nex is a teaching language for this journey: expressive enough for real design discussions, lightweight enough to stay focused on engineering decisions.

The goal is not to lock you into one language.

The goal is to build transferable skills you can apply in any serious codebase.

What You Will Practice

By the end, you should be able to:

- write precise problem statements
- model systems with explicit assumptions and constraints
- select data structures based on access patterns, not habit
- reason about algorithmic behavior as systems scale
- use contracts and invariants to prevent silent corruption
- debug by isolating causes, not chasing symptoms
- refactor with confidence instead of fear
- use AI assistants as accelerators without outsourcing judgment

The goal is not just to become a faster coder.

The goal is to become the engineer people trust with systems that matter.

A Note On Style

You will see three kinds of material throughout the book:

- **Narrative scenarios** that ground ideas in realistic system behavior.
- **Engineering frameworks** that name and structure decisions.
- **Implementation sketches** (including Nex examples) that connect design to executable systems.

If you are early in your programming journey, move slowly and implement often.

If you are experienced, use the chapter prompts as a way to audit your defaults. Many senior failures come from invisible assumptions, not missing knowledge.

Before Chapter 1

Keep one question in mind as you begin:

What problem am I actually solving?

Not:

- What library should I use?
- Which architecture is trendy?
- How quickly can I ship this feature?

Those questions matter later.

The first question determines whether the rest of the work has a chance.

A robot is waiting in an intersection. A researcher cannot recover her own knowledge. A virtual economy is collapsing.

Their codebases are different. Their failure mode is the same.

Chapter 1 begins there.

Part I.

Part I — Seeing The Problem

1. What Problem Are We Actually Solving?

A team designing a delivery robot for a large office building faces what appears to be a well-defined problem: compute shortest paths through the building and execute them. The path-finding algorithms are well-understood; the implementation is straightforward. After several months the robot navigates reliably — and delivers packages to the wrong destinations.

The algorithm was correct. The problem was wrong.

Buildings change. Hallways become blocked, elevators fail, doors close. The engineers had modeled the building as a fixed graph and optimized movement through it. But the actual challenge was not path-finding in a known environment — it was decision-making in an uncertain one. These are structurally different problems, requiring different solutions. No improvement to the path-finding algorithm could have fixed the system, because the algorithm was not the source of the failure.

This distinction — between the problem as initially described and the problem as it actually exists — is one of the most consequential in software engineering.

1.1. Symptoms and Problems

What the team described as their problem — "compute the shortest path from A to B" — was in fact a symptom: a visible manifestation of a deeper structural condition. The underlying problem was that the environment was dynamic and the system assumed it was not.

A symptom names the observable failure. A problem names the structural condition that makes the failure inevitable. Treating a symptom may eliminate the immediate manifestation while leaving the condition intact, producing either the same failure under different circumstances, or a different failure entirely.

The distinction appears throughout software:

Symptom	Underlying problem
"The database is slow."	The data model does not match the query patterns.
"Search results are irrelevant."	The system does not model user intent.
"The robot misdelivers packages."	The environment is dynamic; the model is static.
"The system cannot scale."	The architecture assumes a fixed workload.

In each case, the symptom suggests a local repair: tune a query, adjust a ranking, recalibrate a sensor, add a server. The problem demands a different kind of response: a change to the model, the architecture, or the representation. Conflating the two leads to systems that are locally optimized and globally broken.

Exercise 1.1. Each description below names a symptom. Restate it as a structural problem, and identify one constraint that any valid solution must satisfy.

1. "Our build times are too slow."
2. "Users keep entering invalid data."
3. "The recommendation engine suggests items users already own."

1.2. Three Systems

Throughout this book we will return to three systems that appear unrelated but share a common difficulty: in each case, the problem as naively described obscures the problem as it actually exists.

A delivery robot must move through a large office building and deliver packages. The naive formulation — find shortest paths, execute them — collapses once the map changes, the elevator fails, or a hallway is blocked. The actual problem is not path optimization. It is planning under uncertainty: how does an agent act rationally when its model of the world is incomplete or out of date?

A knowledge engine answers questions using a large document collection. The naive formulation — index the documents, rank by relevance — breaks down immediately. Relevant to what? The system must model what the user is trying to accomplish, not merely what words they used. It must recognize when its sources disagree, and when its knowledge is insufficient to answer. The actual problem is not retrieval. It is reasoning about the reliability and scope of knowledge.

A virtual world contains thousands of objects interacting in real time — characters, physics, events, behaviors. The naive formulation frames this as a performance problem: make it fast enough. But the real difficulty is structural. How can large numbers of independently evolving components interact without producing chaos? The actual problem is not throughput. It is managing complexity through representation.

In each system, the naive formulation leads to a dead end. The engineers who succeed are those who see past the initial description to the structure underneath.

1.3. The Shape of a Problem

Different problems have different structures, and the structure of a problem constrains the space of reasonable solutions.

Consider what distinguishes the three systems above. The delivery robot problem involves an agent acting over time in a changing environment — it belongs to a class of problems requiring state, perception, and recovery from failure. The knowledge engine problem involves reasoning under uncertainty — it belongs to a class where the system must represent not just what it knows, but the limits of what it knows. The virtual world problem involves the composition of many interacting components — it belongs to a class where the challenge is controlling emergent behavior.

These structural differences are not incidental. A solution suited to one class will not transfer to another. Applying retrieval techniques to the robot problem, or path-planning techniques to the knowledge problem, will not produce a working system. The mismatch between problem structure and solution structure is the primary source of software that works technically while failing practically.

Recognizing the structure of a problem is therefore not preliminary to engineering — it *is* the central act of engineering. The choice of algorithm, data structure, and architecture all follow from it. A team that misidentifies the structure will make systematically wrong choices throughout the project, and no amount of subsequent optimization will recover the loss.

Exercise 1.2. For each of the three systems described above, identify one design decision that would be correct given the naive formulation but wrong given the actual problem structure.

1.4. Before the First Line of Code

Software projects fail for many reasons, but one cause appears with unusual regularity: the problem was never clearly defined. This failure rarely looks dramatic. It appears in small decisions made early, when the cost of changing direction seems low:

- requirements specify features rather than goals
- performance is optimized before correctness is established
- tools and frameworks are chosen before the problem is understood

Each of these is a form of the same error: beginning to construct a solution before the problem has been examined. The cost is not immediately visible, because early code still compiles, tests still pass, and progress still appears to be made. The failure emerges later, when the system meets the actual problem and the mismatch cannot be patched away.

The discipline this book develops is the habit of stopping before that first line of code and asking a harder set of questions:

- What is the system actually trying to accomplish?
- What assumptions are hidden in the problem description?
- What would constitute a failure, and what would cause it?
- What is the structure of this problem, and what class of solutions does that structure admit?

These questions slow the process at the beginning. They prevent a larger, costlier slowdown later.

Programming begins with code. Engineering begins with understanding.

1.5. Looking Ahead

The next chapter introduces a concrete method for examining systems before designing them: how to observe real behavior, surface hidden assumptions, and state constraints precisely enough that they can be violated, tested, and refined. We will apply this method first to the delivery robot, where the gap between naive description and actual problem is clearest. By the end of Part I, the three systems will each have yielded a precise problem statement — and the algorithms we build in Part II will follow from those statements almost without choice.

2. Looking at the Real World

A specification describes how a system is supposed to behave. An observation describes how it actually behaves. These are rarely the same thing, and the gap between them is where most software failures live.

The previous chapter established that problems have structure, and that misidentifying the structure leads to systematically wrong solutions. This chapter asks a prior question: how do we discover the actual structure of a problem before we have committed to a design? The answer is observation — disciplined, skeptical attention to the real environment the system must inhabit.

2.1. Models and Their Limits

Every software system embeds a model of the world it operates in. The delivery robot embeds a model of the building. The knowledge engine embeds a model of what users want. The virtual world embeds a model of physical interaction. These models are not incidental — they are load-bearing. Every algorithm in the system depends on them.

A model is an abstraction: it retains some features of the world and discards others. A graph of hallways retains connectivity and discards the fact that hallways are occupied by people, blocked by deliveries, and occasionally closed for maintenance. This is not a flaw — every useful model discards something. The question is whether what was discarded matters.

It often does. Watch the building the robot must navigate for an hour. What the graph does not contain: people moving unpredictably through corridors, a door propped open on some days and locked on others, an elevator out of service without notice, a hallway narrowed by construction equipment. The algorithm operating on the graph is correct. The graph itself is an inadequate model of the environment. No improvement to the algorithm can fix this — the problem is upstream.

This is the first purpose of observation: to find what the model discards, and to ask whether that matters.

2.2. Users Are Inside the System

A second source of modeling failure is the treatment of users as inputs rather than participants. A system that accepts queries treats the user as a source of well-formed requests. Real users are not this.

Consider a user who asks a knowledge engine: *"What causes memory leaks?"* Taken literally, this is a retrieval problem — find documents about memory leaks. But the question is underdetermined. The user might mean memory leaks in C, where the cause is manual allocation without corresponding deallocation; in Java, where the garbage collector cannot reclaim objects with live references; in machine learning pipelines, where tensors accumulate across training steps. The same string of words points at structurally different problems in different contexts.

Users also ask questions that do not name what they mean at all: *"Why does my program keep growing?"* This is not a malformed query. It is a reasonable description of an experience, offered by someone who does not yet know the technical vocabulary for the phenomenon they are observing.

A system designed for well-formed inputs will fail both of these users. The design failure is not in the retrieval mechanism — it is in the assumption that input arrives already interpreted. Real users bring incomplete information, wrong vocabulary, and implicit context. A model that excludes this is a model of a system that does not need to exist, because its users do not exist either.

Exercise 2.1. A calendar application allows users to schedule meetings. Describe three ways a real user's behavior might violate the assumptions embedded in a naive design. For each, state the hidden assumption explicitly.

2.3. Constraints Are Not Obstacles

A common pattern in early design is to solve the unconstrained version of the problem first, then "add constraints later." This produces systems that are elegant in principle and unworkable in practice, because the constraints are not obstacles laid on top of the solution — they are part of the problem's structure.

The virtual world illustrates this clearly. In an unconstrained simulation, every object interacts with every other: forces propagate, collisions are detected exactly, state is updated at every timestep. The computational cost is quadratic in the number of objects. At a thousand objects, this is infeasible in real time. The constraint

— that the simulation must run at interactive speeds — is not a performance requirement to be met after the design is complete. It is a structural feature of the problem that determines which solutions are admissible.

A designer who ignores this constraint will produce an architecture that cannot be repaired by optimization. The data structures, the update strategy, the partitioning of the world — all of these must be chosen with the constraint in mind from the beginning. A spatial index that makes nearby-object queries fast, a fixed timestep that allows approximate but bounded computation, a distinction between objects that matter to the current frame and objects that do not — these design choices follow from the constraint. They are not refinements to a design; they are consequences of understanding the problem.

The question is not *"What is the ideal solution, and how close can we get?"* It is *"What does a solution look like when the constraints are treated as given?"*

2.4. Hidden Assumptions

The most dangerous element of any model is what it assumes without stating. A hidden assumption is a condition the system requires in order to function correctly, which is neither documented nor tested, and which therefore fails silently when violated.

The delivery robot assumes its map is current. The knowledge engine assumes that documents in its collection are authoritative. The virtual world assumes that no two objects occupy the same position at the start of a frame. Each of these assumptions is reasonable under normal conditions. Each produces a failure mode that is difficult to diagnose, because the failure does not point to the assumption — it points to the code that depended on it.

When the robot reaches a hallway that no longer exists, it does not report "map out of date." It reports, if it reports anything coherent at all, that it cannot find a valid path. The assumption has been violated, but the violation is invisible to the system, which has no representation of map currency and therefore no way to detect its absence.

Observation surfaces hidden assumptions because observation confronts the model with the world. Watching the robot navigate exposes the assumption about the map. Watching users query the knowledge engine exposes the assumption about vocabulary. Running the simulation with two objects at the same position exposes the assumption about initial state. The assumptions do not reveal themselves in code review or specification analysis; they reveal themselves when the system meets conditions it was not built to handle.

Exercise 2.2. Identify two hidden assumptions in the following specification: *"The payment system processes transactions in the order they are received."* For each assumption, describe the failure mode that results when it is violated.

2.5. From Observation to Question

Observation produces raw material — a collection of behaviors, failures, and anomalies. Before this can inform a design, it must be transformed into questions precise enough to analyze.

This transformation is not automatic. The observations produced by watching a real system are messy and particular: the robot got stuck in corridor B3 on Tuesday afternoon; a user searching for "memory leak" clicked none of the top five results; two objects in the simulation passed through each other during a high-velocity collision. These are facts, but they are not yet questions. A question names a structural condition, not an instance of it.

Observation	Question
The robot got stuck in corridor B3.	What information does the robot need to detect that a path has become impassable?
Users ignore the top search results.	How should the system represent the difference between topical relevance and user intent?
Objects pass through each other at high velocity.	What invariant must the collision detection system maintain regardless of object velocity?

The question is not a restatement of the observation. It identifies the structural condition the observation is evidence of, and names what a solution must address. A team that asks *"How do we fix the B3 incident?"* will produce a patch. A team that asks *"What information does the robot need to detect impassable paths?"* will produce a design.

A well-formed engineering question has three properties. First, it is general enough to cover the class of cases the observation belongs to, not just the instance observed. Second, it is specific enough to admit a definite answer — or at least to make clear what evidence would constitute progress toward one. Third, it is stated in terms of the system's behavior and structure, not in terms of the symptom that motivated it.

Exercise 2.3. The following are observations from a real system. For each, write a well-formed engineering question that names the structural condition the observation is evidence of.

1. *"The app crashes when two users edit the same document simultaneously."*
2. *"The recommendation engine stops suggesting new items after a user has been active for several months."*
3. *"The search index returns stale results for several minutes after content is updated."*

2.6. What Observation Cannot Do

Observation is not sufficient on its own. A system can be observed indefinitely without producing a design, because observation produces evidence but not explanations. The gap between them requires a model — a proposed account of the structure that produces the observed behavior.

This is why observation and modeling must proceed together. An observation without a model is an anomaly without a cause. A model without observation is a hypothesis without evidence. The engineering process alternates between them: observe behavior, form a hypothesis about structure, derive predictions, test the predictions against further observation, revise the model.

The three systems in this book will each undergo this process. In each case, careful observation will reveal that the naive model is inadequate, and the revised model will suggest a design that the naive model could not have produced.

The next chapter formalizes what a model must contain: a precise statement of the problem, the constraints that bound it, and the assumptions it makes explicit. That formalization is what converts the output of observation into something an algorithm can be built against.

3. Writing a Problem Statement

Once we have observed the real world and identified the actual problem, a new challenge emerges.

How do we describe that problem clearly enough that a computer — and a team of programmers — can actually solve it?

This is the job of a **problem statement**.

A good problem statement is not a wish list or a vague description of desired outcomes. It is a precise account of what a system must accomplish: the inputs it receives, the outputs it must produce, and the constraints it must respect. It sits between messy reality and precise algorithms, translating one into the other.

Get it wrong in either direction and the project suffers. Too vague, and every programmer builds something different. Too rigid, and the system becomes impossible to evolve. Writing a strong problem statement is therefore one of the most important skills in programming — and one of the least taught.

3.1. Two Ways to Get It Wrong

3.1.1. Being Too Vague

A vague problem statement might look like this:

> *"Build a robot that can deliver packages inside the building."*

This sounds reasonable. It is not. It leaves critical questions completely open:

- How does the robot know where it is?
- What happens when a path is blocked?
- How are delivery locations represented?
- What response time is acceptable?

Without answers, every programmer imagines something different. The result is confusion, incompatible code, and a system that satisfies no one.

3.1.2. Being Overly Rigid

At the other extreme, some specifications attempt to nail down every detail:

> *"The robot must represent the building as an adjacency matrix and compute shortest paths using Dijkstra's algorithm."*

This feels precise. But it introduces a different kind of problem: it locks the solution in too early.

What if the building turns out to be very large? What if the map changes frequently, or real-time updates are required? The data structure chosen upfront may be entirely wrong for the actual task.

A good problem statement describes **what must be achieved**, not how it must be implemented. Implementation is the programmer's job; the problem statement is not the place for it.

3.2. What a Strong Problem Statement Does

A well-written problem statement answers three questions clearly:

1. **What information enters the system?**
2. **What output must the system produce?**
3. **What constraints must always be respected?**

It avoids prescribing specific algorithms or data structures. Think of it as describing the shape of the problem, not the shape of the solution.

Here is a clearer version of the delivery robot problem:

> *The system receives a map of the building consisting of locations connected by traversable paths. Given a starting location and a destination, the robot must find a route that reaches the destination while avoiding any blocked paths.*

Notice what this does well. It defines the inputs (map, starting location, destination), the expected behavior (find a valid route), and leaves all implementation decisions open. Multiple algorithms could solve this problem correctly. That flexibility is intentional.

3.3. Specifications in Practice

Seeing the contrast between weak and strong specifications makes the principle concrete.

Weak: *"The knowledge engine should provide relevant answers to user questions."*

The phrase "relevant answers" is ambiguous. Two programmers reading this independently will produce two incompatible systems.

Stronger: *Given a user query, the system retrieves documents containing information related to the query and ranks them by estimated relevance.*

Now we know the input (a query), the output (ranked documents), and the goal (relevance ranking). Plenty of design decisions remain open — which is exactly what we want.

Weak: *"Objects in the virtual world should interact realistically."*

What does "realistically" mean here? Physics simulation? Game-style rules? Simplified abstractions?

Stronger: *Objects in the world have positions and interaction rules. When two objects occupy overlapping space, the system applies the interaction rule associated with their types.*

Now the problem is algorithmically meaningful. We can start thinking about spatial data structures, collision detection, and interaction systems. The work can begin.

3.4. The Power of Examples

Even a well-written specification can leave gaps. That is why engineers rely heavily on examples.

Examples do something that abstract description cannot: they make expectations concrete and surface hidden assumptions before they become bugs.

For the delivery robot system, a simple example looks like this:

```
Start:       Office A
Destination: Office D

Map: A → B → C → D

Expected route: A → B → C → D
```

This confirms the basic behavior. But examples become truly powerful when they explore what happens at the edges.

3.5. Edge Cases and Counterexamples

A counterexample is an example designed to expose the limits of a naïve solution. Consider what happens when a path is blocked:

```
Map: A → B → C → D
             X
     (B → C is blocked)
```

Now the robot cannot follow the obvious route. Two questions immediately arise: can it find an alternative path? And if no alternative exists, what should it do?

These are not afterthoughts. They are requirements — and without counterexamples, they are easy to forget.

Every good specification should be stress-tested with three kinds of examples:

- **Normal cases** — the everyday behavior the system is built for
- **Edge cases** — unusual but valid inputs that might trip up a simple implementation
- **Failure cases** — situations where the system cannot succeed and must respond gracefully

By the time the specification is written, the system's behavior should already have been explored through examples. The writing merely captures what the examples revealed.

3.6. A Note on AI-Assisted Development

Examples matter even more in the current era of AI coding assistants. A vague description gives an AI model almost nothing to work with. A precise specification with concrete examples — including edge cases — gives it a clear target.

The habits that make specifications useful to human teammates make them equally useful to machine ones.

3.7. The Problem Statement as Contract

At its core, a problem statement is a contract. It creates an agreement between:

- the **problem** we want solved
- the **algorithms** we will design to solve it
- the **code** we will eventually write

Vague contracts produce unpredictable systems. Precise ones make every downstream step easier: algorithms are easier to choose, tests are easier to write, bugs are easier to find and fix.

When something goes wrong — and something always does — a well-written problem statement tells you whether the implementation violated the contract or the contract itself was wrong. Either way, you know where to look.

3.8. Quick Exercise

Choose one of these three systems: the delivery robot, the knowledge engine, or the virtual world. Write a problem statement with exactly three parts:

1. **Inputs** — what information enters the system?
2. **Required behavior** — what must the system do with it?
3. **Constraints** — what conditions must always hold?

Then add one normal example, one edge case, and one failure case.

If your statement specifies an algorithm or data structure, rewrite it to focus on outcomes instead.

3.9. Takeaways

- A strong problem statement is precise about outcomes and silent about implementation.
- Vague statements create misalignment; over-specified ones lock in decisions too early.
- Examples, edge cases, and counterexamples surface missing requirements before they become costly mistakes.
- A problem statement is a contract between intent, design, and code.
- Clear contracts make algorithm selection, testing, and debugging substantially easier.

In Chapter 4, we stress-test problem statements systematically — probing edge cases, failure conditions, and contradictions. If Chapter 3 defines the contract, Chapter 4 tries to break it.

4. Edge Cases — Where Systems Break

In the previous chapter, we wrote a contract: a precise statement of what a system must do. In this chapter, we try to break it.

This is not destructiveness for its own sake. Breaking a specification is the fastest way to find out whether it is complete. Every gap that an edge case exposes is a requirement that would otherwise be discovered in production — which is to say, discovered expensively.

4.1. The Structure of a Failure

Most programs fail not because their central logic is wrong, but because they encounter situations nobody imagined. The algorithm works. The reasoning is sound. But somewhere, silently, the code assumed something it was never entitled to assume.

These unexamined assumptions have a name: **edge cases**.

Consider the delivery robot. Its pathfinding works correctly for any well-formed map with a reachable destination. Now ask: what is a "well-formed map"? What does "reachable" mean if the map changes while the robot is moving? The moment we push on the specification, we discover that it was never as complete as it appeared.

This pattern repeats across every system:

- The knowledge engine ranks documents correctly — unless the query is empty, in which case it divides by zero.
- The virtual world applies interaction rules correctly — unless two rules conflict, in which case it applies both, or neither, or crashes.

The failure is not in the algorithm. It is in the **boundary of the algorithm's applicability** — the unstated preconditions that the algorithm relied upon but nobody wrote down.

4.2. Boundaries

Edge cases cluster around boundaries. A boundary is any point where the nature of the input changes qualitatively — where "more of the same" becomes "something different."

Some boundaries are numerical: the empty collection, the single-element collection, the collection large enough to exhaust memory. Some are structural: the graph with no edges, the graph with a cycle, the graph where source and destination are the same node. Some are temporal: the event that arrives before the system is ready, the two events that arrive simultaneously.

Return to the delivery robot. Typical operation involves a map with several rooms and a clear path between them. But consider the boundary cases:

Situation	What the algorithm must decide
Start equals destination	Return the trivial route, or report an error?
Destination is unreachable	Report failure — but what failure?
Multiple shortest paths exist	Return one, return all, or declare a tie?
Map contains a cycle	Does the algorithm terminate?
Map is empty	Fail immediately, or search and fail gracefully?

None of these are exotic. Every one of them will occur in a real building with a real robot. The question is whether they are handled intentionally or accidentally.

A famous class of boundary errors is the **off-by-one error**. A loop written as

```
for i from 0 to n
```

may process element `n`, which lies one step beyond the end of a zero-indexed array of length `n`. The loop is almost correct. It works for every index except the last one. That word — *almost* — is doing a great deal of damage.

Almost correct is a category of wrong.

4.3. Thinking Like a Tester

The most reliable way to find edge cases is to change perspective. Stop thinking like someone building the system. Start thinking like someone trying to make it fail.

A tester does not ask: *does this work for the normal case?* A tester asks:

- What happens if the input is empty?
- What happens if the input is larger than expected?
- What happens if the input is malformed?
- What happens if two things occur at the same time?
- What happens if something fails halfway through?

Applied to our three systems, this mindset produces specific, answerable questions.

Delivery robot. What if a path becomes blocked after the robot has begun moving? What if the robot receives a map update mid-route? What if two robots are routed through the same corridor simultaneously?

Knowledge engine. What if the query contains a term that appears in no document? What if two highly-ranked documents directly contradict each other? What if the same query, submitted twice in rapid succession, produces different rankings?

Virtual world. What if an interaction rule specifies a result that violates a physical invariant? What if an object interacts with itself? What if a thousand objects converge on the same point in a single step?

These questions may seem adversarial. They are. That is the point. The adversary is reality, and reality will ask every one of these questions eventually. Better to ask them now, in a design document, than later, in a crash report.

4.4. Edge Cases Refine the Specification

There is a deeper reason to take edge cases seriously. They do not merely reveal holes in a design — they reveal that the original problem statement was about a simpler problem than the one we actually need to solve.

Return to the specification from Chapter 3:

> *Given a starting location and a destination, the robot must find a route that reaches the destination while avoiding any blocked paths.*

This is clear and precise — for the case where such a route exists. But edge analysis forces three new questions:

1. What must the system do when no route exists?
2. What must the system do when multiple equally valid routes exist?
3. What must the system do when the map changes between queries?

Each question extends the specification. After answering them, we have a richer, more realistic contract — one that a programmer can actually implement fully, and a tester can actually verify completely.

This is the double value of edge analysis: it improves the design *and* makes the design testable.

4.5. The Three-Category Inventory

When examining any feature or system, edge cases can be organized into three categories. Working through all three systematically is more reliable than relying on intuition alone.

Input edges. What are the degenerate inputs — empty, null, malformed, out of range? What is the smallest valid input? The largest? What inputs sit exactly at declared boundaries?

Size edges. What happens at scale? An algorithm that works for ten items may behave differently for ten million. Data structures have capacity limits. Networks have latency. Time constraints that hold for small inputs may not hold for large ones.

State edges. What if the system is in an unexpected state when a request arrives? What if two operations occur simultaneously? What if a previous operation left the system in a partially modified state?

For the delivery robot: the empty map is an input edge; a building with ten thousand rooms is a size edge; a map update that arrives while a route is being computed is a state edge. Each category exposes a different class of assumption.

4.6. Edge Cases as Design Instruments

It is tempting to treat edge cases as nuisances — special cases that complicate otherwise clean algorithms. This is the wrong way to see them.

Edge cases are **design instruments**. Carefully analyzed, they often point toward better abstractions.

Consider what happens when we ask: *what should the robot do if the start and destination are the same location?* One answer is to return the trivial route `[A]`. Another is to return the empty route `[]`. These are different answers with different implications for every piece of code that processes the result.

Choosing between them forces a decision about the meaning of a route. That decision, once made, clarifies the algorithm, simplifies the code that calls it, and makes the system's behavior predictable in cases that the original specification left undefined.

The edge case did not complicate the design. It completed it.

4.7. Quick Exercise

Take the problem statement you wrote at the end of Chapter 3 and apply the three-category inventory to it.

For each category — input edges, size edges, state edges — identify at least one case your current specification does not address. Then answer:

1. What should the system do in each unaddressed case?
2. Does your answer require revising the specification, or adding to it?
3. Which cases would you test first, and why?

If your specification handles every case you can identify without modification, push harder. In our experience, a specification that survives its first edge analysis without revision is one whose edges were not examined closely enough.

4.8. Takeaways

- Programs fail at the boundaries of their assumptions, not at the center of their logic.
- Edge cases cluster around boundaries: degenerate inputs, extreme sizes, conflicting states.
- Thinking like a tester — adversarially, systematically — is a learnable and transferable skill.
- Edge analysis refines the specification: every gap it exposes is a requirement that was always there, waiting to be written down.
- Almost correct is a category of wrong.

Chapter 5 takes the next step: turning problem statements, examples, and edge analyses into formal specifications — requirements precise enough that two programmers, working independently, build the same system.

5. From Stories to Specifications

Every system in this book began as a story. A robot navigates a building to deliver packages. A knowledge engine answers questions by searching a document collection. A virtual world simulates the interactions of objects governed by rules. These stories are genuinely useful — they communicate intent, establish context, and give purpose to the engineering work that follows. Without them, we would not know what we were building or why.

But a story cannot be executed. A computer does not understand intent or purpose. It executes precise rules operating on precisely defined inputs, producing precisely defined outputs. The task that bridges these two things — the story that communicates what the system should do and the rules that a machine can follow — is specification: the transformation of a narrative into a formal description of behavior.

This transformation is where software engineering begins. Not in the choice of algorithm, not in the selection of data structure, but in the earlier and harder work of asking: what, precisely, is the system supposed to do?

5.1. What Specification Requires

A story describes events and intentions. A specification describes rules and behaviors. The difference is precision — not as a stylistic preference but as a requirement imposed by the nature of computation. Anything left vague in a specification will be resolved by the implementation, which means it will be resolved by whoever writes the code, without necessarily consulting the people who understood the purpose of the system.

Consider the story:

> *The robot should navigate through the building.*

This sentence communicates an intention clearly enough for a human reader. For a programmer, it raises questions that the sentence does not answer: what information does the robot have about the building? What counts as navigation? What does the system guarantee about the route it produces? Until these questions are answered, the programmer cannot write code — they can only guess, and the guesses will differ from one programmer to the next.

A specification answering these questions might say:

> *The robot receives a map of locations connected by traversable paths, a current location, and a destination. The system must compute a sequence of locations from the current location to the destination, using only traversable connections, or report that no such route exists.*

Every word in this specification does work. "Map of locations connected by traversable paths" defines the input structure. "Current location and destination" names the other inputs. "Sequence of locations" defines the output structure. "Using only traversable connections" states a correctness condition. "Or report that no such route exists" declares the failure behavior. The story introduced the idea; the specification introduced formal elements we can reason about and, eventually, test against.

5.2. Inputs: What the System Receives

Every computational problem begins with inputs — the information available to the system when it must produce a result. Identifying the inputs precisely is the first concrete step of specification work, because an algorithm cannot operate on information it does not have, and an algorithm that assumes information that will not always be available will fail on the inputs where that information is absent.

For the delivery system, the inputs to the navigation problem include the map of the building, the robot's current location, the destination, and the current state of each path — whether it is traversable or blocked. Each of these is a separate input with a separate type. The map is a structure. The locations are identifiers. The path states are properties of edges. Writing them down separately forces a question that the story left implicit: what is the format of each, and what constraints must each satisfy to be valid?

For the knowledge engine, the inputs to a search operation include the query — the expression of what the user is looking for — the document collection, and whatever history or context is relevant to the ranking. The query is a string, but a string with constraints: it must be non-empty, and its format must be one the system knows how to process. The document collection has its own structure. History, if it is used, must be defined: what exactly is recorded, in what form, and how far back?

For the virtual world, the inputs to a simulation step include the current state of all objects — their positions, their types, their current status — and the interaction rules that govern what happens when objects of certain types encounter each other. The rules are themselves a kind of input: they define the system's behavior, and they can change.

In each case, the act of listing the inputs is also an act of discovery. Questions that the story left unanswered must be answered before the list can be completed.

5.3. Outputs: What the System Must Produce

If inputs describe what enters the system, outputs describe what the system must deliver. Defining the output precisely is the second step of specification work, because the output definition is where the system's success condition lives — it is what we check when we ask whether the system worked correctly.

For the navigation problem, the output is a route: a sequence of locations from the start to the destination. But "a sequence of locations" is not yet precise enough. The sequence must use only traversable connections — a sequence that teleports from one location to another is not a valid route. The first element must be the start location and the last must be the destination. And if no valid route exists, the output must say so explicitly, in a form that the caller can distinguish from a valid route.

Each of these constraints is a separate claim about the output, and each must be stated. An output specification that says only "a sequence of locations" permits outputs that would fail for the caller who depends on the first element being the start, or the caller who cannot distinguish an empty route from a failure.

For the knowledge engine, the output of a search is a ranked list of document identifiers. The ranking is ordered by relevance — but relevance must itself be defined. The output specification must state what criterion was used to order the results, so that callers can interpret the ordering correctly and tests can verify that the ordering satisfies the criterion.

5.4. Guarantees: What the System Promises

Beyond inputs and outputs, a specification states guarantees — the promises that the system makes about what it will do for any input satisfying the preconditions. Guarantees are what define correctness. Without a stated guarantee, we cannot say whether the system behaved correctly, because we have not said what correct behavior is.

For the navigation problem:

> *If a path exists between the start and destination, the system returns a sequence of traversable connections from start to destination. If no path exists, the system returns a declared failure status.*

For the knowledge engine:

> *The system returns a ranked list of documents from the collection, ordered by estimated relevance to the query. A query that matches no documents returns an explicit empty result.*

For the virtual world:

> *After each simulation step, every object satisfies the invariants for its type. Objects interact only according to the rules defined for their type pairing.*

Each guarantee is a testable claim. A system that returns a sequence with a non-traversable connection has violated the navigation guarantee. A system that returns null instead of an empty result has violated the knowledge engine guarantee. The test does not need to know how the system works — it only needs the guarantee to know what to check.

5.5. The Ambiguity That Stories Leave

Stories leave decisions unmade. This is not a deficiency — it is a feature of narrative that makes stories easy to understand and share. But every unmade decision in a story becomes a question that the implementation must answer, and when the implementation answers questions that the specification should have answered, the result is software whose behavior depends on implementation choices that were never reviewed, agreed upon, or tested.

The word "best" is the clearest example. The story says the robot should take the best path. The specification must decide what best means: shortest total distance, fewest connections, least time, least energy, greatest reliability. These are different objectives and they lead to different routes on most graphs. A specification that inherits "best" from the story without resolving it has deferred the decision to the implementation, where it will be resolved once, quietly, and almost certainly without the knowledge of the people who depend on the system to optimize the right thing.

"Relevant" in the knowledge engine story has the same character. "Realistic" in the virtual world story has the same character. Any adjective in a story that carries evaluative weight — best, relevant, realistic, efficient, appropriate — must be replaced in the specification with a definition: the specific criterion by which the adjective will be evaluated and the specific method by which it will be computed.

Making these decisions explicit is not bureaucratic overhead. It is the work that prevents the system from silently optimizing the wrong objective and the team from discovering the disagreement only when it becomes a production incident.

5.6. A First Look at Contracts

As the systems in this book grow more complex, specifications will evolve into contracts — formal statements of the agreement between different parts of the system about what each part requires and what each part guarantees. Contracts are specifications made executable: they can be checked at runtime, violated visibly, and used as the basis for tests.

A contract for the navigation function captures three things. The precondition states what must be true before the function is called: the map correctly represents the connections between locations, the start and destination are locations that exist in the map. The postcondition states what the function guarantees when it returns: if a path exists, the returned route is a valid connected sequence from start to destination; if no path exists, the returned status is the declared failure value. The invariant states a property that holds throughout and after the call: every step in the route corresponds to a connection that existed in the map at the time of the call.

This structure will appear throughout the rest of the book, in every design chapter and every code sketch. The precondition, postcondition, and invariant are not formalism for its own sake — they are the tools by which informal expectations become statements that can be reasoned about, tested against, and maintained as systems change. Every contract in the chapters ahead is a specification that was made precise enough to be enforced.

5.7. Quick Exercise

Take one story sentence from your system and transform it into a mini-specification with four parts: the inputs the operation receives, the output it must produce, the guarantee it makes about the output for valid inputs, and one precondition, one postcondition, and one invariant.

Start with the story sentence:

> *"The system should deliver packages efficiently."*

Write out what efficient means in terms of a measurable criterion. State what information the system must have to apply that criterion. State what a correct output looks like and how you would verify it. Then find one word in your specification draft that is still ambiguous — a word that could be interpreted differently by two reasonable readers — and replace it with a definition.

5.8. Takeaways

- Stories communicate intent; specifications define behavior. Both are necessary, and neither replaces the other — but software is built from specifications, not stories.
- Every specification needs three elements stated precisely: the inputs the system receives, the outputs it must produce, and the guarantees it makes about the relationship between them.
- Ambiguous terms in stories — best, relevant, efficient, realistic — become correctness failures in code. The specification's job is to replace them with definitions.
- Contracts formalize specifications into executable agreements: preconditions state caller obligations, postconditions state routine guarantees, invariants state properties that must hold throughout. This structure appears everywhere in the chapters ahead.

Part I has now established the full arc from problem understanding to precise specification: stories that identify the situation, problem statements that define the computational task, edge cases that test the boundaries of the definition, and specifications that transform intent into formal claims. Part II opens with the next question: before algorithms can operate on a problem, the world must be represented. Chapter 6 asks why software needs models before it needs optimization.

Studio 1 — Our First Tiny System

Subtitle

Building the smallest working versions of all three systems.

Studio Focus This studio chapter is hands-on: implement, verify behavior, and document tradeoffs as you iterate.

The Situation

The team has finished Part I and has problem statements, example scenarios, and edge-case notes. Now they must ship a tiny end-to-end version of each system to test whether the problem framing is actually usable.

Systems in scope:

- delivery scheduler (single robot, tiny map)
- note organizer (small collection, simple lookup)
- virtual world (few entities, deterministic update loop)

Engineering Brief

Build minimal vertical slices that run from input to output.

Required outcomes:

- define a minimal model for each system
- implement one core operation per system
- encode at least one explicit contract per operation
- demonstrate behavior on nominal and edge inputs

Implementation guidance:

- prefer clarity over generality
- keep architecture deliberately small
- document assumptions that might break at scale

Implementation In Nex

Use Nex for the studio implementation so contracts, assumptions, and behavior are all visible in one place.

Suggested file split:

- `delivery_tiny.nex`
- `notes_tiny.nex`
- `world_tiny.nex`
- `studio_1_main.nex`

If you are using the web IDE, you can also place everything in one file and run `App.run`.

Delivery Tiny (single robot, tiny map)

```
class Delivery_Tiny
feature
  next_stop(current, destination: String): String
    require
      current_non_empty: current /= ""
      destination_non_empty: destination /= ""
    do
      if current = destination then
        result := current
      elseif current = "A"
             and destination = "C" then
        result := "B"
      elseif current = "B"
             and destination = "C" then
        result := "C"
      else
        result := "UNREACHABLE"
      end
    ensure
      decision_returned: result /= ""
    end
end
```

Notes Tiny (simple lookup)

```
class Notes_Tiny
feature
  find_by_tag(tag: String): String
    require
      tag_non_empty: tag /= ""
    do
      if tag = "algorithms" then
        result := "note_001"
      elseif tag = "graphs" then
        result := "note_002"
      else
        result := "NOT_FOUND"
      end
    ensure
```

```
      response_non_empty: result /= ""
    end
end
```

World Tiny (deterministic update step)

```
class World_Tiny
feature
  step(position, velocity, max_x: Integer): Integer
    require
      max_positive: max_x > 0
    do
      let next: Integer := position + velocity
      if next < 0 then
        result := 0
      elseif next > max_x then
        result := max_x
      else
        result := next
      end
    ensure
      bounded: result >= 0 and result <= max_x
    end
end
```

Studio Driver (nominal + edge inputs)

```
class App
feature
  run() do
    let d: Delivery_Tiny := create Delivery_Tiny
    let n: Notes_Tiny := create Notes_Tiny
    let w: World_Tiny := create World_Tiny

    -- Nominal

    -- expected: "B"
    print(d.next_stop("A", "C"))
    -- expected: "note_001"
    print(n.find_by_tag("algorithms"))
    -- expected: 5
    print(w.step(3, 2, 10))

    -- Edge / failure-oriented checks

    -- expected: "C"
    print(d.next_stop("C", "C"))
    -- expected: "UNREACHABLE"
    print(d.next_stop("X", "C"))
    -- expected: "NOT_FOUND"
    print(n.find_by_tag("unknown"))
    -- expected: 10
    print(w.step(9, 5, 10))
    -- expected: 0
    print(w.step(1, -5, 10))
  end
end
```

You can evolve this starter in three directions:

- add a richer map model for delivery
- replace fixed tag matching with indexed note structures
- extend world updates to multiple entities per tick

Studio Challenges

Level 1 — Core Implementation

- Implement the tiny system for one domain.
- Add executable examples and expected results.

Level 2 — Cross-System Generalization

- Implement tiny versions for all three domains.
- Identify one shared abstraction across them.

Level 3 — Exploration

- Replace one design choice with an alternative and compare outcomes.
- Record what changed in complexity and failure behavior.

Postmortem

Discuss:

- Which assumptions were validated?
- Which assumptions were false?
- What information was missing from the original problem statement?
- What should be tightened before model redesign?

Deliverables

- runnable Nex code for all tiny systems
- short design notes (inputs, outputs, guarantees)
- edge-case checklist with observed behavior

Part II.

Part II — Modeling the World

6. Why Software Needs Models

Part I taught us to write better problem statements. Given a system to build, we can now describe its inputs, its required behavior, and its constraints with enough precision that implementation can begin. That is genuine progress.

It is not enough.

Teams that move directly from specification to code — even from a good specification — reliably produce systems that are brittle. The code compiles. The tests pass. The demo looks right. Then scale arrives, or requirements change, or two features interact in a way nobody anticipated, and the system begins to crack. New fixes introduce new inconsistencies. The codebase becomes a map of past decisions that nobody fully remembers making.

The missing layer is a **model**.

6.1. What a Model Is

A model is not code. It is the structured account of what exists in a system, how those things relate to one another, and what rules govern how they can change.

The distinction matters. Code answers the question: *how does the machine execute this?* A model answers a different question: *what is the system actually about?*

Consider what happens without one. Two teams, working from the same specification, represent the same concept in subtly different ways. Edge behavior ends up depending on call order rather than on explicit rules. Performance degrades because nobody thought carefully about how data would actually be accessed. Every new feature lands as a local patch, adding complexity without adding clarity.

A model prevents this by forcing structure to become explicit before implementation decisions harden. It sits between specification and code as a third, distinct layer:

- The **specification** answers: *what must happen?*
- The **model** answers: *what exists, and how can it change?*
- The **algorithms** answer: *how do we compute efficiently within that model?*

When the middle layer is weak or absent, the other two layers drift apart. The specification describes a system that the code does not quite implement. The relationship can be found only by reading the code very carefully — if it can be found at all.

6.2. The Same Structure in Three Domains

Our three running systems look very different on the surface. Modeling them reveals that the same underlying structure appears in all three — and that the natural first model for each is too narrow.

The delivery network. A first pass might model only "find a route from A to B." This captures the nominal case and nothing else. A stronger model includes locations, the connections between them, the dynamic state of each connection (open or blocked), and the robot's current position and assigned task. With this model, route selection becomes a query against explicit state rather than a hardcoded behavior. Blocking a corridor is a state change; replanning is a response to that change. The system becomes something we can reason about.

The knowledge engine. A first pass might model only "query string maps to result list." Quality degrades quickly under this model because there is no representation of what quality depends on. A stronger model includes documents, the terms and tags that characterize them, relevance signals, and a representation of confidence or uncertainty. With this model, ranking becomes a computation over explicit structure — one that can be adjusted, debugged, and improved.

The virtual world. A first pass might model only "update all objects each frame." Interactions become chaotic because there is no account of what governs them. A stronger model includes entity identity, entity state, typed interaction rules, and the boundaries of each update step. With this model, correctness can be reasoned about one transition at a time, rather than observed at runtime and hoped for.

The domains differ. The modeling move is the same: name stable structure before writing algorithms over it.

6.3. From Ambiguous Requirement to Model

The path from a vague requirement to a working model has a shape. Walking through it concretely is more useful than describing it in the abstract.

Start with this requirement:

> *"The robot should choose good routes and avoid getting stuck."*

This sentence is useful in a product conversation. It is nearly useless for implementation — "good" and "stuck" carry no technical meaning. The five-step path below converts it into something buildable.

Step 1: Separate nouns from verbs. Nouns suggest entities that need to be modeled. Verbs suggest operations that act on them. From the requirement above: nouns include robot, route, location, path, blockage; verbs include choose, avoid, move, re-plan. This separation is the first act of modeling.

Step 2: Define state, not just behavior. For each noun, ask what properties it has that can change over time. A minimal state model for the robot system:

- `Location`: an identifier
- `Path`: a source, a destination, a status
- `Robot`: a current location and a destination
- `Map`: a set of locations and paths

This is still small. It is already more useful than the original sentence.

Step 3: Add invariants early. Before writing any algorithm, write down the truths that must hold at all times. Examples for this system: every path endpoint must reference a known location; the robot's current location must exist in the map; any returned route must use only open paths. These invariants are not implementation details. They are the system's reliability backbone, and they belong in the model.

Step 4: Make change explicit. Most real failures happen during transitions, not in static state. What transitions are possible? A path becomes blocked. The robot receives a new destination. A replan request either succeeds or fails. A model that does not represent transitions will behave correctly in demos and incorrectly in production.

Step 5: Delay algorithm commitment. At this stage, do not choose between BFS, Dijkstra, A*, or any other pathfinding strategy. The right algorithm depends on access patterns and performance requirements that the model has not yet fully revealed. Choosing early is choosing blind.

6.4. A Model in Code

The following sketch shows how this model might be expressed in Nex. The point is not the specific syntax — it is the correspondence between the model we described and the structure of the code.

```
class Path
feature
```

```
  from_loc: String
  to_loc: String
  open: Boolean

create
  make(from_loc, to_loc: String) do
    this.from_loc := from_loc
    this.to_loc := to_loc
    open := true
  end

invariant
  endpoints_present: from_loc /= "" and to_loc /= ""
end

class Robot_State
feature
  current: String
  destination: String

create
  make(current, destination: string) do
    this.current := current
    this.destination := destination
  end

feature
  request_replan(has_open_route: Boolean): String
    require
      current_known: current /= ""
      destination_known: destination /= ""
    do
      if current = destination then
        result := "ARRIVED"
      elseif has_open_route then
        result := "ROUTE_FOUND"
      else
        result := "UNREACHABLE"
      end
    ensure
      status_defined: result = "ARRIVED"
                      or result = "ROUTE_FOUND"
                      or result = "UNREACHABLE"
    end
invariant
  state_valid: current /= "" and destination /= ""
end
```

This sketch is intentionally small. What it does is more important than what it omits: it defines state explicitly, expresses behavior in terms of that state, and encodes guarantees as verifiable contracts. The pathfinding algorithm is not here yet. It can be added later, in any form, without disturbing the model. The same approach applies to the knowledge engine and the virtual world: model first, optimize second.

6.5. Five Ways Modeling Goes Wrong

Knowing the pattern does not prevent all mistakes. The following five failures appear regularly enough to be worth naming.

Modeling operations before entities. The symptom is a codebase full of utility functions with no clear home — data scattered across modules, ownership unclear. The recovery is to list stable entities first and attach operations to explicit state.

Ignoring boundaries. The symptom is unclear borders between subsystems, with the same concept represented differently in different places. The recovery is to define model boundaries in writing and make every conversion point explicit.

Missing invariants. The symptom is bugs described as "impossible states" — situations the code was never designed to handle because nobody wrote down what was always supposed to be true. The recovery is to express invariants at the class and operation level, and to fail fast when they are violated.

Premature algorithm lock-in. The symptom is an architecture that cannot adapt because an early performance guess became a structural assumption. The recovery is to keep the model stable while treating the algorithm as a replaceable component.

Treating the model as documentation only. The symptom is diagrams that exist and code that disagrees with them. A model that lives only in a design document is not enforced by anything. The recovery is to encode model assumptions in types, contracts, and tests — and to keep them synchronized with the implementation.

6.6. Quick Exercise

Choose one of the three running systems and produce a minimal model draft with four parts:

1. **Entities** — three to six things the system needs to track
2. **Relationships** — how those entities connect to one another
3. **Invariants** — two or three rules that must always hold
4. **Transitions** — two or three important ways the state can change

Then ask two questions about your current implementation of that system: which parts of the implementation are not represented in this model? And which parts of the model are not currently enforced in code?

The gap between those two answers is where reliability work should start.

6.7. Takeaways

- A model is the missing layer between specification and implementation: it describes what exists and how it can change, independently of how the machine executes it.
- Stable entities, explicit boundaries, invariants, and transitions are the core elements of any model.
- The right algorithm depends on the model. Choose algorithms after the model is clear, not before.
- Most failures at scale are model failures before they are code failures.
- A model that exists only in documentation is not enforced. Encode model assumptions in types, contracts, and tests.

Chapter 7 develops the first modeling primitive in depth: entities. We will distinguish identity from state, assign responsibilities, and choose representations that remain stable as the system grows.

7. Entities — The Things That Exist

Chapter 6 established that a model is the necessary layer between specification and implementation. Now we build the first element of that model.

An entity is something the system must track as a distinct thing over time.

That definition sounds simple. It is one of the most consequential distinctions in software design. When a team misidentifies its entities — treating derived values as persistent things, or collapsing distinct identities into one — everything downstream suffers. Interfaces leak implementation details. Algorithms operate on the wrong units. Tests pass in isolation and fail across the system.

The question that entity modeling forces us to answer is: **what are the actual things in this system that deserve an identity of their own?**

7.1. Identity and State

Before naming entities, we need a sharper vocabulary. Two concepts do most of the work.

Identity answers: *which thing is this?* It is what remains stable across the entire lifetime of an entity — the thread of continuity that lets the system say "this is the same robot, even though it has moved."

State answers: *what is true about this thing right now?* It is everything that can change while identity stays fixed — position, status, assignment.

The separation is not cosmetic. If your model treats a state change as the creation of a new entity, you lose continuity: there is no way to say that the robot before the reroute and the robot after it are the same robot. If your model merges entities that are genuinely distinct, you corrupt behavior: two delivery tasks that happen to share a destination are not the same task.

Applied to our three systems, the separation looks like this:

Delivery network. The entities are `Robot`, `Location`, and `DeliveryTask`. A robot's identity persists across reroutes and retries; its current location is state. A delivery task's identity persists across status changes; whether it is pending or in transit is state.

Knowledge engine. The entities are `Document`, `Tag`, and `QuerySession`. A document can be edited, annotated, or re-indexed, but it remains the same document. Its content is state; its identity is not.

Virtual world. The entities are `WorldObject`, `Player`, and `InteractionRule`. An object's position changes every tick. Its identity does not.

7.2. Responsibilities Belong to Entities

Once entities are named, the next question is what they are responsible for.

The temptation — especially in codebases that grew without an explicit model — is to collect entity-related logic in global helper functions or service layers that reach into shared data structures and mutate them directly. This pattern works at small scale and degrades badly at large scale: ownership becomes unclear, invariants are enforced inconsistently, and changing one entity's representation requires finding and updating code in many places.

The better pattern is to assign each entity responsibility for the invariants that govern its own state, and to expose state changes only through explicit operations. An entity that controls its own transitions is an entity whose behavior can be reasoned about locally.

In concrete terms: `DeliveryTask` should validate its own status transitions — it should not be possible for external code to move a delivered task back to pending. `Document` should guard the integrity of its own metadata. `WorldObject` should enforce whatever bounds constrain legal movement or state change.

This is not about strict encapsulation as a stylistic preference. It is about making invariants enforceable rather than merely conventional.

7.3. From Requirement to Entity Model

The path from an ambiguous requirement to a set of well-defined entities has a shape. Consider this requirement:

> *"Track package progress and show live status."*

It mentions things — packages, status, presumably robots and locations — but gives them no structure. The following steps convert it into a model.

Step 1: Extract candidate entities. Read the requirement and list the nouns that might deserve persistent identity: package, robot, route, user notification. Do not commit yet.

Step 2: Keep only stable identity carriers. A route is often the output of a computation — a derived value, not a persistent thing. User notifications may be events rather than entities. After filtering: `DeliveryTask`, `Robot`, `Location` are likely stable identity carriers.

Step 3: Define identity keys. Assign each entity a field that uniquely identifies it across time: `DeliveryTask.task_id`, `Robot.robot_id`, `Location.location_id`. Choosing these fields early forces clarity about what makes two instances the same thing.

Step 4: Define minimal state. For `DeliveryTask`: origin, destination, status (`PENDING`, `IN_TRANSIT`, `DELIVERED`, `FAILED`), and optionally an assigned robot identifier. Start small. State can be extended; a bloated initial model is harder to correct.

Step 5: Add invariants. The task identifier must be non-empty. Origin and destination must be non-empty. A delivered task cannot return to pending. These are not validation rules to be added later — they are properties of the entity itself, and they belong in the model now.

Step 6: Add transition operations. Assign robot, mark in transit, mark delivered, mark failed. Each transition is an explicit operation, not a field assignment scattered across the codebase.

At the end of this process, we have entity structure rather than feature prose. Implementation can begin with a clear target.

7.4. An Entity in Code

The following sketch shows how the `DeliveryTask` entity might be expressed in Nex:

```
class Delivery_Task
feature
  task_id: String
  origin: String
  destination: String
  status: String

create
  pending(task_id, origin,
          destination, status: String) do
```

```
    this.task_id := task_id
    this.origin := origin
    this.destination := destination
    this.status := "PENDING"
  end

feature
  start() do
    if status = "PENDING" then
      status := "IN_TRANSIT"
    end
  ensure
    started_or_unchanged: status = "IN_TRANSIT"
                          or status = "PENDING"
  end

  complete() do
    if status = "IN_TRANSIT" then
      status := "DELIVERED"
    end
  ensure
    delivered_or_unchanged: status = "DELIVERED"
                            or status = "IN_TRANSIT"
  end
invariant
  id_present: task_id /= ""
  endpoints_present: origin /= ""
                     and destination /= ""
  valid_status:
    status = "PENDING" or
    status = "IN_TRANSIT" or
    status = "DELIVERED" or
    status = "FAILED"
end
```

Read this sketch against the six steps above. `task_id` is the identity key from Step 3. The four status values are the minimal state from Step 4. The three invariants correspond directly to Step 5. `start` and `complete` are the transition operations from Step 6, each with a postcondition that rules out the illegal outcomes.

The pathfinding logic, the ranking algorithm, the notification system — none of that appears here, because none of it belongs to this entity. The entity model and the algorithm are separate concerns. Keeping them separate is what makes both easier to change.

7.5. Four Ways Entity Modeling Goes Wrong

Treating everything as an entity. If every value in the system gets an identity and a lifecycle, the model bloats and the overhead of managing persistence overwhelms the work the model is supposed to simplify. The recovery is to reserve entity status for things that genuinely need to be tracked across time, and to treat derived or computed values as results — outputs of queries over entity state, not entities themselves.

No identity, only fields. A model that represents entities as plain records without stable identifiers makes it impossible to reason about continuity. Updates overwrite the wrong records; history cannot be reconstructed. The recovery is to define explicit, stable identity fields and to specify what equality means for each entity.

Mixed responsibilities. When transition logic and validation are scattered across services, controllers, and helper functions, the entity's invariants are enforced only by convention — which means they are not reliably enforced at all. The recovery is to move entity-specific rules close to entity operations and to centralize transition checks.

Unchecked state transitions. A delivered task that can become pending again through a misapplied patch is a symptom of transitions that exist only as informal agreements between developers. The recovery is to model transitions explicitly and to enforce them with contracts and tests — not because developers cannot be trusted, but because explicit enforcement scales and informal agreement does not.

7.6. Quick Exercise

Choose one of the three running systems and produce an entity sheet with five parts:

1. **Entity name**
2. **Identity field or fields**
3. **Core state fields**
4. **Allowed transitions** — which status or state changes are legal, and from which starting states?
5. **Two invariants** — what must always be true?

Then ask: which fields in your current implementation are being used as identity but were never explicitly designated as such? And which transitions are currently unchecked — possible to execute in any order, with no enforcement of preconditions?

Those two gaps are where entity modeling work should begin.

7.7. Takeaways

- Entities are identity-bearing things that the system must track over time. Not every value is an entity.
- Identity and state are distinct: identity answers *which thing is this?*, state answers *what is true about it now?*
- Responsibilities should be assigned to entities, not diffused across global helpers and service layers.

- Transitions and invariants are part of the entity model itself, not implementation details to be added later.
- Strong entity models reduce friction at every downstream stage: algorithm design, testing, and refactoring.

Chapter 8 moves from things to connections. Entities alone are not enough — behavior emerges from the relationships between them.

8. Relationships — How Things Connect

Entities give us the vocabulary of a system. Relationships give it grammar.

A model composed only of entities is a list of nouns with no verbs, no dependencies, no structure. The interesting behavior of any real system — the behavior that is difficult to reason about, difficult to test, and difficult to change without breaking something — lives almost entirely in how entities connect to one another. Getting entities wrong makes a system confusing. Getting relationships wrong makes it dangerous: data diverges silently, cascading failures become invisible, and the system acquires behavior that nobody designed and nobody can fully explain.

Relationships deserve the same deliberate treatment we gave entities in Chapter 7. That means making them explicit, naming their properties, and encoding the rules that govern them before those rules get buried in application code.

8.1. What a Relationship Is

A relationship encodes how two entities are linked and what that link means. Three examples from our running systems:

- A `Robot` is assigned to a `DeliveryTask`
- A `Document` carries a `Tag`
- A `WorldObject` interacts with another `WorldObject`

In each case, the link is not decorative. It defines which operations are permitted, which queries are meaningful, and what must remain consistent across the system. An assignment implies that the robot is unavailable for other tasks. A document carrying a tag implies that removing the tag changes the document's position in search results. An interaction between world objects implies that the rules governing both objects must be consulted when their states change.

When these implications are left implicit — encoded nowhere in the model, enforced nowhere in the code — they do not disappear. They simply become assumptions that every developer must independently discover and manually respect. That is the source of most subtle, long-lived bugs in large systems.

8.2. Five Dimensions of a Relationship

To make a relationship fully explicit, five questions must be answered.

Cardinality. How many instances of each entity can participate? A robot may be assigned to at most one active delivery task at a time — this is a one-to-one constraint. A document may carry many tags, and a tag may apply to many documents — this is many-to-many. Getting cardinality wrong produces either artificial restrictions or data models that permit states the domain forbids.

Direction. Is the relationship symmetric or does it flow one way? A delivery task references an origin and a destination, but a location does not inherently reference the tasks that pass through it. A document link between A and B may or may not imply a link from B to A, depending on the link type. Direction determines which entity "owns" the relationship and which follows from it.

Ownership. Which entity is responsible for maintaining the integrity of the link? In some relationships, one entity is the parent and the other is a dependent — the parent's existence is a precondition for the child's. In others, the relationship is mediated by a shared index or junction structure, and neither entity fully owns it. Leaving ownership undefined means that two different parts of the system may each assume the other is maintaining the link, and neither does.

Lifecycle coupling. What happens to a relationship when one of its participants is removed? Three outcomes are possible: the related entity is deleted with it (cascade), the related entity is left without a valid reference (orphan), or the relationship is preserved with a historical marker that makes the absence explicit. Each choice has consequences. None of them is automatically correct. All of them need to be decided rather than discovered.

Constraint rules. What must always be true across the relationship? A link must reference entities that actually exist. Certain link types may forbid self-reference. Traversal graphs for certain relationship types may forbid cycles. These constraints are invariants of the relationship itself, and they belong in the model for the same reason that entity invariants do.

8.3. Relationships in the Three Systems

Applying these dimensions to our three systems reveals both the specific relationships and the shared pattern they instantiate.

Delivery network. Locations are connected to other locations through paths; each path has a direction and a status. A robot is assigned to at most one active delivery task at a time — a one-to-one constraint with a well-defined lifecycle: when a task is completed or fails, the assignment is released. A delivery task references its origin and destination locations, and those locations must exist for the task to be valid.

Knowledge engine. Documents are linked to tags in a many-to-many relationship, mediated by an association that can carry additional information such as confidence or provenance. Documents are linked to other documents through typed edges, where the link type determines the semantics of traversal. Inferred links — connections generated by the system rather than asserted by a user — may carry a confidence score that distinguishes them from explicit ones.

Virtual world. World objects belong to spatial regions, a containment relationship that determines which interaction rules apply. Interaction rules reference pairs of object types rather than individual objects, making them a relationship between the type system and the entity system. Event relationships capture cause-and-effect chains, linking the event that triggered a transition to the transition it produced.

In all three cases, relationships are first-class elements of the model. They are not incidental fields on entities; they are structures with their own properties, constraints, and lifecycle rules.

8.4. When to Use a Relationship Entity

Sometimes a relationship is simple enough to be represented as a reference field on one of the participating entities. A delivery task carries an identifier for its assigned robot. A document carries a list of tag identifiers. These representations work when the relationship has no properties of its own and when only one query direction matters.

When a relationship needs to carry metadata — a timestamp, a confidence score, a link type, a source — it becomes a thing in its own right and deserves to be modeled as an entity. Consider the requirement:

> *"The knowledge engine should connect related notes."*

The first candidate model is a direct many-to-many connection: each document holds a list of related document identifiers. This works until we need to know when a connection was made, who made it, or what kind of connection it is. At that point the flat field becomes inadequate, because the relationship has acquired properties that cannot live on either endpoint.

The stronger model names the relationship explicitly. Start with the entities: `Document`, `Tag`, `Link`. A `Link` connects two documents with a typed edge. Constraints follow: both endpoints must exist, self-links of certain types are forbidden, and the set of valid link types is controlled. Query patterns follow from there: finding all references for a given document, finding backlinks, traversing two hops to discover indirectly related material.

The discipline is to model for real query patterns, not for theoretical elegance. A model that supports forward lookup efficiently but makes reverse lookup expensive or impossible is a model that was designed without asking how the data would actually be used.

8.5. A Relationship in Code

The following sketch expresses the `Doc_Link` relationship entity in Nex:

```
class Doc_Link
feature
  from_id: String
  to_id: String
  link_type: String

  is_structurally_valid(): Boolean do
    result := from_id /= "" and to_id /= ""
              and link_type /= ""
  ensure
    result_is_boolean: result = true
                       or result = false
  end

create
  make(from_id, to_id, link_type: String) do
    this.from_id := from_id
    this.to_id := to_id
    this.link_type := link_type
  end
invariant
  endpoints_present: from_id /= "" and to_id /= ""
  non_self_reference: from_id /= to_id
  link_type_present: link_type /= ""
end
```

This is minimal by design. It captures the three invariants that any link must satisfy — both endpoints must be identified, neither endpoint may reference itself, and the link type must be specified — and exposes a validation operation whose contract is explicit. The underlying document objects are referenced by identifier rather than by direct containment, which means the link's validity depends on those documents existing in the broader model, not just in this class. That dependency belongs in the integration-level invariants we write when assembling the full model.

The same structure generalizes: `Path` in the delivery network is a relationship entity between locations; typed interaction edges in the virtual world are relationship entities between object types.

8.6. Four Ways Relationship Modeling Goes Wrong

Encoding relationships as free-text fields. When entity references are stored as untyped strings — names, descriptions, human-readable identifiers — the model has no way to enforce that the referenced entity actually exists or that two references to "the same thing" are in fact consistent. Joins become unreliable and traversals become impossible. The recovery is to model relationships as typed links with constrained endpoints.

Ignoring reverse queries. A relationship that is easy to traverse in one direction and expensive or impossible to traverse in the other was designed for only half of its intended use. Forward lookup and reverse lookup are both first-class access patterns, and both should be considered when the relationship is modeled. If the reverse query is expensive under the current model, that is information about the model, not about the query.

Semantic drift in link types. When the same link type accumulates subtly different meanings across different parts of the system, the relationship becomes uninterpretable: a traversal that was valid under the original meaning may be invalid under the acquired one, and there is no record of when or why the meaning changed. The recovery is to define a controlled taxonomy of link types at the model level and to enforce membership in that taxonomy when links are created.

Hidden lifecycle rules. When an entity is removed and the links that referenced it are not updated, the system accumulates broken references that will produce failures at some unpredictable future point. The lifecycle policy — cascade, orphan, or preserve — must be defined explicitly at the model level and verified by tests that exercise entity removal, not just entity creation.

8.7. Quick Exercise

Choose one of the three running systems and construct a relationship matrix. For each relationship you identify, record: the two entity types it connects, the relationship type and cardinality, and one constraint rule that must always hold.

Then identify one reverse query that your model must support — a query that traverses the relationship in the direction opposite to how you first defined it. If that reverse query is expensive or ambiguous under your current model, the model needs refinement before implementation begins.

8.8. Takeaways

- Relationships are first-class elements of a model: they have cardinality, direction, ownership, lifecycle coupling, and constraint rules.
- Getting relationships wrong produces silent integrity failures and invisible cascading behavior — more dangerous, in practice, than getting entities wrong.
- A relationship that carries metadata or supports multiple semantic types is usually better modeled as an entity in its own right.
- Model for real access patterns. A relationship that supports only forward traversal is only half a relationship.
- Lifecycle rules must be explicit. The choice between cascade, orphan, and preserve has consequences that compound over the lifetime of a system.

Chapter 9 brings entities and relationships together into a complete data model, and examines the tradeoffs that arise when a model must serve multiple competing concerns simultaneously.

9. Designing a Good Data Model

Chapters 7 and 8 gave us the two building blocks of a model: entities, which carry identity and state, and relationships, which encode the structure between them. A data model is what these pieces become when assembled into a coherent design — one that must hold together not just on paper, but under the pressure of real queries, real load, and real change.

The goal is not the most abstract model or the most general one. It is the model that keeps the system correct, understandable, and evolvable as usage evolves. Those three properties are in genuine tension, and resolving that tension requires making explicit choices rather than deferring them.

9.1. Five Properties of a Good Model

A data model that serves a system well over time satisfies five properties.

It represents reality adequately for the problem's scope. Not perfectly — a model that tries to represent every nuance of the domain it describes becomes as complex as the domain itself. The right model captures the distinctions that matter for the operations the system must perform, and deliberately ignores the rest.

It supports critical operations efficiently. A model that is logically correct but operationally unusable at scale is not a good model; it is a design that deferred a hard problem. The access patterns the system depends on most heavily should be visible in the model's structure.

It protects invariants against invalid states. The value of an invariant is precisely that it holds everywhere, not just where the developer remembered to check. A model that encodes invariants as conventions — things programmers agree to uphold rather than things the system enforces — will eventually produce the impossible states that conventions cannot prevent.

It allows change without destructive rewrites. Requirements change. The model must be able to absorb that change incrementally.

A model that is correct today but can only be extended by dismantling its existing structure will be replaced, or worked around, rather than evolved.

It remains explainable to new engineers. A model that can only be understood by the person who built it is a liability. Explainability is not a soft requirement; it is what determines whether the model continues to be used correctly as the team changes.

9.2. Model from Operations, Not from Nouns Alone

The most common failure in data modeling is to begin with a list of entities — the nouns in the domain — and build structure around them without asking how the data will actually be used. This produces models that are logically tidy and operationally expensive, because the access patterns the system needs were never consulted during design.

The corrective practice is to ask three questions before the model is finalized:

What are the most frequent read paths? For a knowledge engine where the dominant operation is retrieving notes by tag and recency, the model needs a tag index path, a recency field, and a defined ranking input. Without these, code accumulates ad hoc caches and special cases to compensate for a model that was never shaped for this query.

What are the most frequent write and update paths? For a delivery system where reassignment is a routine operation, the model needs to make the assigned-robot relationship easy to update and the validity of an update easy to check. A model that makes reassignment expensive is a model that will be bypassed.

What invariants must hold through both reads and writes? The invariants that protect the model during reads — a task's status must be one of a defined set — also constrain writes. A write that violates an invariant must be rejected before it completes, not discovered afterward.

Designing from operations does not mean ignoring the domain. It means ensuring that the domain model and the operational reality stay aligned from the beginning, rather than diverging and requiring reconciliation later.

9.3. Tradeoffs That Must Be Made Explicit

No model of a real system is free of tradeoffs. The failure is not in making tradeoffs — it is in making them implicitly, so that their consequences are discovered rather than anticipated.

Normalization versus duplication. A fully normalized model stores each fact in exactly one place, which makes updates clean and consistency easy to guarantee. It also means that queries requiring data from multiple entities must traverse relationships, which increases read complexity. A model that duplicates data for read performance trades update complexity for query simplicity. Neither choice is universally correct; both choices need to be documented so that future developers understand why the model has the shape it does.

Generality versus simplicity. A highly generic schema — one that represents all possible relationships as typed edges, or all possible attributes as key-value pairs — can accommodate future requirements that the current model does not anticipate. It also makes the current requirements harder to express, harder to validate, and harder to query efficiently. The right level of generality is the minimum needed to handle the requirements that are actually known, plus a small margin for the changes that are already foreseeable.

Strictness versus flexibility. Strict constraints prevent invalid states. They also mean that workflows requiring temporarily incomplete data — a delivery task created before a robot is assigned, a document link created before its target exists — must be designed explicitly. Relaxing constraints to accommodate these workflows trades correctness guarantees for operational convenience. This tradeoff is real and sometimes the right one; what matters is that it is chosen deliberately and that the consequences are understood.

9.4. From Requirement to Data Model

The following requirement will serve as a worked example:

> *"Show live delivery status and allow reassignment when robots fail."*

Two distinct operations are embedded in this sentence: a read operation (show live status) and a write operation (reassign tasks when a robot fails). A good model must support both.

Step 1: Identify core entities. Robot, DeliveryTask, Location. These are the identity-bearing things the system must track.

Step 2: Define relationships. A delivery task references its origin and destination locations. A delivery task optionally references an assigned robot — optionally, because a task exists before it is assigned and may exist again after reassignment. The cardinality of the assignment relationship is one-to-one for active tasks: a robot may hold at most one active task at a time.

Step 3: Define operational queries. Find the active task for a given robot. Find the highest-priority pending task available for assignment. Find all active tasks belonging to a failed robot and mark them available for reassignment. These three queries determine which fields the model needs and how they need to be indexed.

Step 4: Shape the model for those queries. The task needs a status field, a priority field, and an assigned robot identifier. The status field drives the first and third queries; the priority and status fields together drive the second.

Step 5: Protect invariants. A delivered task cannot be reassigned. A task in transit must have an assigned robot. Origin and destination must reference valid locations. These are the constraints that the model must enforce, not merely assume.

Step 6: Plan for evolution. Future requirements already visible on the horizon include multi-robot cooperative delivery, regional routing constraints, and service-level classes. The current model should not implement these, but it should not foreclose them either. The task's assignment relationship should accommodate multiple assignees when that requirement arrives. The priority field should be defined broadly enough that SLA classes can be expressed through it. The goal is not to build for the future now — it is to avoid building something today that prevents the future from being built at all.

9.5. A Data Model in Code

```
class Delivery_Model
feature
  task_id: String
  assigned_robot_id: String
  status: String
  priority: Integer

  can_reassign: Boolean do
    result := status = "PENDING"
              or status = "FAILED"
  end

  mark_in_transit(robot_id: String)
    require
      has_robot: robot_id /= ""
      legal_transition: can_reassign
    do
```

```
      assigned_robot_id := robot_id
      status := "IN_TRANSIT"
    ensure
      transit_set: status = "IN_TRANSIT"
                   and assigned_robot_id = robot_id
    end

create
  pending(task_id: String) do
    this.task_id := task_id
    status := "PENDING"
  end
invariant
  task_id_present: task_id /= ""
  priority_non_negative: priority >= 0
  valid_status:
    status = "PENDING" or status = "IN_TRANSIT"
    or status = "DELIVERED" or status = "FAILED"
end
```

This sketch is worth reading against the six steps above. The status and priority fields answer Step 4. The three invariants answer Step 5. The `can_reassign` query answers the first part of Step 3 — it codifies the rule that only `PENDING` and `FAILED` tasks are available for reassignment, making that rule checkable before `mark_in_transit` is called. The precondition on `mark_in_transit` answers the second part: a transition into `IN_TRANSIT` requires both a valid robot identifier and a legal starting status.

What the sketch does not address is the storage backend, the query execution strategy, or the index layout. Those are implementation concerns. The model's job is to define what is true; the implementation's job is to enforce and compute it efficiently.

9.6. Four Ways Data Model Design Fails

Mirroring the UI rather than the domain. When the data model is shaped to match the structure of a particular screen or interface, it becomes fragile in exactly the way interfaces are fragile: a UI redesign requires a model change, which requires a migration, which propagates through every system that depends on the model. The recovery is to model domain semantics first and treat every interface as a projection of the model, not a template for it.

Missing transition rules. When the operations that change entity state are not part of the model — when status updates are just field assignments with no enforced preconditions — illegal state combinations appear in production. A delivered task becomes pending; an in-transit task loses its assigned robot. The recovery is to encode transitions explicitly, as operations with preconditions and postconditions, and to verify them with tests that attempt the illegal transitions.

Performance as an afterthought. A model designed without consulting its dominant access patterns will be correct and slow. At the point where performance becomes unacceptable, the model has usually hardened into the system's foundation, and adjusting it is expensive. The recovery — which is really a prevention — is to identify the critical query paths during modeling, not after deployment.

No versioning strategy. Models change. Fields are added, relationships are restructured, invariants are tightened or relaxed. A model that was not designed with change in mind will require migrations that break compatibility with existing data or existing clients. The recovery is to make model changes additive where possible — adding fields rather than modifying them, extending enumerations rather than replacing them — and to document the compatibility assumptions that each version of the model relies on.

9.7. Quick Exercise

For one of the three running systems, write a one-page data model brief with six sections: core entities, core relationships, the three most important read operations, the three most important write operations, three invariants that must hold across all of them, and one foreseeable extension that the current model should not implement but should not prevent.

Then name one tradeoff your model makes — between normalization and duplication, generality and simplicity, or strictness and flexibility — and explain why you made it.

If you cannot name a tradeoff, look harder. A model with no visible tradeoffs is a model whose tradeoffs have not yet been examined.

9.8. Takeaways

- A good data model is operational, not merely conceptual: it is shaped by the queries and transitions the system must support, not just the entities the domain contains.
- Critical access patterns must inform model structure from the beginning. Performance problems that originate in model design are the most expensive to fix.
- Every non-trivial model embodies tradeoffs between normalization and duplication, generality and simplicity, strictness and flexibility. Make these tradeoffs explicit and document them.

- Transition rules are part of the data model. A model that defines states without defining legal transitions between them is incomplete.
- The test of a model's quality is not how it looks at design time but how well it absorbs change over the life of the system.

Chapter 10 examines the dimension of data modeling that most designs underestimate: change over time. A model that handles its initial requirements correctly may still fail as the system evolves. Chapter 10 makes the patterns of temporal change explicit and shows how to design for them.

10. Modeling Change

The models built in Chapters 7 through 9 describe a system at rest: what entities exist, how they relate, what invariants must hold. This is necessary. It is not sufficient.

Real systems do not rest. Robots move, tasks are reassigned, documents are updated, objects collide. The system is in constant transition between states, and the most severe bugs in any real codebase are not errors in stored data — they are errors in how data moves from one state to another. Two updates race and both succeed. One step in a multi-step operation fails after the first step has already committed. Events arrive out of order and state reflects the order of arrival rather than the order of occurrence. A partial write leaves the system in a configuration no design ever intended.

If a model does not describe change — not just what states are valid, but how the system may move between them — then correctness during transitions is accidental. The system works until it encounters a transition the model never considered, at which point it fails in a way no invariant will catch.

10.1. States and Transitions

The solution is to model transitions as first-class design objects, with the same discipline we brought to entities and relationships.

A complete model of change for any entity requires four elements. First, the set of legal states the entity can occupy — for a delivery task, `PENDING`, `IN_TRANSIT`, `DELIVERED`, and `FAILED`. Second, the legal transitions between those states: which moves are permitted and which are not. Third, the preconditions each transition requires: what must be true of the world before the transition may occur. Fourth, the postconditions each transition guarantees: what will be true of the world once it completes.

For the delivery task, the legal transitions form a directed graph: `PENDING → IN_TRANSIT`, `IN_TRANSIT → DELIVERED`, `IN_TRANSIT → FAILED`, and `FAILED → IN_TRANSIT` (for reassignment). The transition `DELIVERED → PENDING` is not in this graph. That absence is a design decision. Making it explicit — not just absent from the code, but documented as forbidden — is what prevents a future developer from adding it under the mistaken belief that it was merely overlooked.

Think in terms of state machines, even for the simplest entities. The exercise of drawing the graph forces questions that informal descriptions leave unasked: can a failed task be retried? Can a pending task be cancelled before assignment? Each answer is a design decision, and each decision should appear in the model.

10.2. The Problem of Ordering

Change introduces time, and time introduces ordering. The correctness of a transition often depends not just on the current state of one entity, but on the sequence in which a series of events occurred. When that sequence is not controlled, the system's behavior becomes a function of timing rather than design.

Three examples from our running systems make this concrete.

In the knowledge engine, relevance scores are updated in response to indexing events. If those events arrive out of order — if a document update is processed before the initial indexing that created the document's baseline — the resulting scores may be inconsistent, and ranking may oscillate between queries until the system converges or a developer investigates.

In the virtual world, collision resolution must happen in a defined relationship to movement. If some objects resolve collisions before their positions are updated for the current tick and others resolve collisions after, the simulation's outcome depends on which set a given object belongs to. Two runs of the same scenario produce different results. The simulation is nondeterministic not because the physics are random but because the model never specified an ordering.

In the delivery system, reassignment of a failed robot's tasks must be strictly ordered with respect to the failure event itself. If reassignment begins before the robot's status is committed as `FAILED`, two robots may independently be assigned the same task — each transition individually valid, the combined outcome impossible according to the invariants.

Modeling change means specifying deterministic ordering for sequences that must be deterministic, and specifying conflict policy for the cases where ordering cannot be guaranteed.

10.3. From Requirement to Transition Model

Consider the requirement:

> *"When a robot fails mid-delivery, tasks should be reassigned automatically."*

This sentence describes an outcome. It conceals a sequence of events, each of which can fail independently, and several of which can interact badly if they are not carefully ordered.

Step 1: Identify the transition events. A robot's heartbeat is missed. The robot's status is set to `FAILED`. Its active tasks are discovered. Reassignment is attempted for each. This is not one event — it is a chain, and the chain has gaps.

Step 2: Define preconditions for the critical transition. Before a task can be reassigned, three things must be true: the task's current status is `IN_TRANSIT` or `FAILED`, the robot currently assigned to it is unavailable, and a candidate replacement exists. Each precondition is checkable before the transition begins. If any fails, the transition does not proceed — it is rejected cleanly rather than failing partway through.

Step 3: Define postconditions. After reassignment completes, the task is assigned to exactly one robot, the task's status is valid, and the reassignment event is auditable — recorded in enough detail that the reason for the change can be reconstructed later. Postconditions are not just documentation; they are the specification that tests will verify.

Step 4: Define failure behavior. If no replacement robot is available, the task is marked `FAILED` with an actionable reason attached. The system does not leave the task in a half-reassigned state — no assigned robot identifier pointing to a failed robot, status still reading `IN_TRANSIT`. That half-state is worse than acknowledged failure because it is invisible: it will not trigger an alert, but it will also never be resolved.

Step 5: Design for idempotency. In any distributed system, events may be delivered more than once. A failure event that is processed twice should produce the same outcome as a failure event processed once. An idempotent reassignment operation — one that checks whether the task is already in the target state before acting — provides this property. Idempotency is not an optimization; it is a reliability multiplier that makes the entire event processing pipeline safer.

10.4. A Transition Model in Code

```
class Task_Transition
feature
  status: String
  assigned_robot_id: String

  reassign(new_robot_id: String)
    require
      robot_present: new_robot_id /= ""
      transition_allowed: status = "IN_TRANSIT"
                          or status = "FAILED"
    do
      assigned_robot_id := new_robot_id
      status := "IN_TRANSIT"
    ensure
      reassigned: assigned_robot_id = new_robot_id
                  and status = "IN_TRANSIT"
    end

  mark_delivered
    require
      must_be_in_transit: status = "IN_TRANSIT"
    do
      status := "DELIVERED"
    ensure
      delivered: status = "DELIVERED"
    end

create
  pending do status := "PENDING" end
  in_transit do status := "IN_TRANSIT" end
invariant
  valid_status:
    status = "PENDING" or status = "IN_TRANSIT"
    or status = "DELIVERED" or status = "FAILED"
end
```

The structure of this sketch mirrors the four elements of a complete transition model. The invariant defines the legal states. The two operations define legal transitions. Each operation's `require` clause defines preconditions; each `ensure` clause defines postconditions. A call to `reassign` that violates its precondition does not execute — the contract is enforced before the operation begins, not discovered after it completes.

What is absent is also significant. There is no operation `mark_pending` that moves a delivered task back to the start. The absence is not an oversight; it is a design decision made explicit by the model. A developer reading this code understands not just what is permitted but, from what is missing, what is forbidden.

10.5. Five Ways Change Modeling Fails

Transition logic scattered across the codebase. When the rules governing state changes are spread across services, controllers, and event handlers, there is no single place where the complete set of legal transitions is visible. Different parts of the system enforce different subsets of the rules, and the gaps between them are where illegal states enter. The recovery is to centralize transition rules in model-level operations with explicit contracts, and to route all state changes through them.

Silent partial failure. A multi-step operation that updates one field successfully and then exits on an error leaves the system in a state that is neither the state before the operation nor the intended state after it. This half-state may be invisible to monitoring and will not be resolved by retry. The recovery is to define atomic boundaries where the underlying platform supports them, and to define explicit compensation states — states that make the failure visible and actionable — where it does not.

Unhandled event reprocessing. In any system that processes events asynchronously, the same event may be delivered more than once. An operation that is not idempotent will produce different results on the second delivery than on the first, creating state that nobody designed. The recovery is to design transition operations to be idempotent where possible, and to track event identity — processing each distinct event at most once — where idempotency cannot be achieved.

Undefined conflict policy. When two concurrent updates target the same entity, the system must have a rule for which one wins. Without such a rule, the outcome is determined by timing, and the same inputs may produce different outputs on different runs. The conflict policy — last write wins, highest priority wins, earlier timestamp wins — is a model-level decision, and it belongs in the model, not in the implementation of whichever operation happened to be written last.

No audit trail for transitions. When a system reaches an invalid state and nobody can explain how it got there, the investigation begins from nothing. An audit trail that records the intent and outcome of each transition — which operation was attempted, what the preconditions were, whether the postconditions were verified — makes invalid states diagnosable. More importantly, it reveals which part of the transition model failed to prevent them.

10.6. Quick Exercise

Choose one entity from your system and construct a complete transition model for it with six parts: all legal states, all allowed transitions between them, one explicitly forbidden transition and the reason it is forbidden, preconditions for one transition, postconditions for the same transition, and the behavior when that transition cannot complete because its preconditions are not met.

Then ask: if the event that triggers this transition were delivered twice, would the second delivery produce a different outcome than the first? If yes, what must be added to make the operation idempotent?

10.7. Takeaways

- Most real failures occur during state transitions, not in stored state. A model that describes states without describing transitions is incomplete.
- Legal transitions must be explicit and enforced. Absent transitions are design decisions, not omissions, and should be recognizable as such.
- Ordering, conflict policy, and idempotency are model-level concerns. They cannot be reliably addressed in implementation if they were never addressed in design.
- Partial failure must be designed for, not improvised around. Half-states that are invisible to monitoring are more dangerous than acknowledged failures.
- Change is structured behavior. Treat it as such from the beginning.

Part II has now built a complete modeling foundation: entities, relationships, data models, and the semantics of change. Part III shifts from representation to computation — what algorithms are, how to decompose problems into them, and how to reason about their behavior under real conditions.

Studio 2 — The Model Redesign

Subtitle

Refactoring data models so systems can evolve without hidden state corruption.

Studio Focus This studio chapter is hands-on: implement, verify behavior, and document tradeoffs as you iterate.

The Situation

Studio 1 proved we could ship tiny vertical slices.

Now those slices are under stress.

As soon as requirements grow, model weaknesses appear:

- entity identity is unclear (`"task"` vs `"route"` vs `"request"` used interchangeably)
- relationships are implicit (IDs inside free-text fields)
- transition rules are scattered (status changes happen from many places)
- invalid states are easy to create and hard to detect

Part II gave us the tools to fix this:

- model entities explicitly
- define relationships with constraints
- shape data around real operations
- model change with legal transitions and invariants

Your job in Studio 2 is to redesign all three systems using those principles.

Engineering Brief

Redesign the core data models for all three systems (delivery, knowledge, virtual world).

Required outcomes:

- explicit entities with stable identity
- explicit relationships with clear cardinality/constraints
- transition operations with contracts (`require/ensure`)
- invariants that make invalid states hard to represent
- migrated behavior examples from Studio 1

Implementation guidance:

- optimize for correctness and clarity before optimization
- keep model boundaries obvious
- make failure behavior explicit (not silent)

Implementation In Nex

Use Nex to encode model assumptions directly in code.
Suggested files:

- `delivery_model_v2.nex`
- `knowledge_model_v2.nex`
- `world_model_v2.nex`
- `studio_2_main.nex`

If using the web IDE, place all classes in one file and run `App.run`.

Delivery Model V2

Focus: identity + legal transitions + assignment relationship.

```
class Delivery_Task
feature
  task_id: String
  origin: String
  destination: String
  status: String
  assigned_robot_id: String

  assign(robot_id: String)
    require
      robot_present: robot_id /= ""
      can_assign: status = "PENDING"
                  or status = "FAILED"
    do
      assigned_robot_id := robot_id
      status := "IN_TRANSIT"
    ensure
      assigned: assigned_robot_id = robot_id
                and status = "IN_TRANSIT"
    end

  complete()
    require
```

```
      in_transit: status = "IN_TRANSIT"
    do
      status := "DELIVERED"
    ensure
      delivered: status = "DELIVERED"
    end

  fail(reason: String)
    require
      reason_present: reason /= ""
      not_delivered: status /= "DELIVERED"
    do
      status := "FAILED"
    ensure
      failed: status = "FAILED"
    end
create
  pending(task_id: String, origin: String,
          destination: String) do
    this.task_id := task_id
    this.origin := origin
    this.destination := destination
    this.status := "PENDING"
  end
invariant
  id_present: task_id /= ""
  endpoints_present: origin /= ""
                     and destination /= ""
  valid_status:
    status = "PENDING" or status = "IN_TRANSIT"
             or status = "DELIVERED"
             or status = "FAILED"
  transit_has_assignment:
    status /= "IN_TRANSIT"
    or assigned_robot_id /= ""
end
```

Knowledge Model V2

Focus: relationship as first-class entity (`Doc_Link`) rather than ad hoc references.

```
class Document
feature
  doc_id: String
  title: String
  body: String
create
  make(doc_id: String, title: String) do
    this.doc_id := doc_id
    this.title := title
  end
invariant
  id_present: doc_id /= ""
  title_present: title /= ""
end

class Doc_Link
feature
  from_doc_id: String
  to_doc_id: String
```

```
  link_type: String

  is_valid(): Boolean do
    result :=
      from_doc_id /= "" and
      to_doc_id /= "" and
      from_doc_id /= to_doc_id and
      (link_type = "references"
       or link_type = "related"
       or link_type = "contradicts")
  ensure
    bool_result: result = true or result = false
  end
create
  make(f: String, t: String, lt: String) do
    from_doc_id := f
    to_doc_id := t
    link_type := lt
  end
invariant
  endpoints_present: from_doc_id /= ""
                     and to_doc_id /= ""
  no_self_link: from_doc_id /= to_doc_id
end
```

Virtual World Model V2

Focus: deterministic update transition with explicit bounds.

```
class World_Object
feature
  object_id: String
  x: Integer
  vx: Integer

  step(max_x: Integer)
    require
      valid_bound: max_x >= 0
    do
      let next: Integer := x + vx
      if next < 0 then
        x := 0
      elseif next > max_x then
        x := max_x
      else
        x := next
      end
    ensure
      bounded_after_step: x >= 0 and x <= max_x
    end
create
  make(id: String, x: Integer, vx: Integer) do
    object_id := id
    this.x := x
    this.vx := vx
  end
invariant
  id_present: object_id /= ""
end
```

Studio Driver

Use one driver to validate nominal + edge behavior.

```
class App
feature
  run() do
    -- Delivery
    let t: Delivery_Task :=
      create Delivery_Task.pending("T001", "A", "C")
    t.assign("R7")
    print(t.status)                -- IN_TRANSIT
    t.complete
    print(t.status)                -- DELIVERED

    -- Knowledge
    let l: Doc_Link :=
      create Doc_Link.make("D1", "D2", "references")
    print(l.is_valid)              -- true

    -- World
    let w: World_Object :=
      create World_Object.make("O-1", 9, 4)
    w.step(10)
    print(w.x)                     -- 10 (clamped)
  end
end
```

Studio Challenges

Level 1 — Core Implementation

- Redesign one system model (delivery or knowledge or world).
- Encode at least 2 invariants and 2 transition operations.
- Run one nominal and two edge cases.

Level 2 — Cross-System Model Consistency

- Redesign all three systems.
- Use a common modeling vocabulary in notes:
  - entity
  - relationship
  - invariant
  - transition
- Show one "before vs after" bug that the new model prevents.

Level 3 — Exploration

Pick one experiment:

- **State-centric vs event-centric** transition handling for delivery reassignment.
- **Direct references vs link entities** in the knowledge model.
- **Single-step deterministic update vs queued events** in virtual world updates.

Evaluate:

- complexity impact
- testability impact
- failure-mode clarity

Postmortem

Discuss:

- Which entity boundary decision removed the most confusion?
- Which relationship constraint prevented a real invalid state?
- Which transition rule was hardest to make explicit?
- Which tradeoff did you intentionally accept (and why)?

Use concrete evidence from your runs, not just opinions.

Deliverables

- redesigned Nex models for delivery, knowledge, and world systems
- short modeling notes per system:
  - entities
  - relationships
  - invariants
  - transitions
- nominal + edge test output logs
- one before/after comparison showing reduced accidental complexity

Exit Criteria

You are ready for Studio 3 if:

- you can explain identity vs state for each core entity

- invalid states are blocked by model rules (not just comments)
- transition legality is encoded in operations/contracts
- relationship constraints are explicit and testable

Part III.

Part III — The Shape of Algorithms

11. What Is an Algorithm?

Part II gave us a model: a precise account of what exists in a system, how the pieces relate, and what must remain true as they change. Part III turns that model into behavior. The question is no longer *what is the system about?* but *how does it compute?*

We begin with a definition. An algorithm is a finite, explicit procedure that transforms valid input into output while preserving declared guarantees. Every word in that definition carries weight. *Finite* rules out procedures that might run forever. *Explicit* rules out procedures whose steps depend on unstated assumptions. *Valid input* means the procedure is not required to behave sensibly on inputs it was never designed for. *Declared guarantees* means that what the algorithm promises is written down and checkable — not inferred from behavior and hoped to hold.

Teams that skip this definition do not avoid it. They implement it informally, inconsistently, and invisibly, and then discover its consequences when the system fails in a way nobody can explain.

11.1. Three Things That Are Not the Same

Software development regularly conflates three distinct concepts. Keeping them separate is one of the habits that most reliably distinguishes careful engineering from productive guessing.

A **requirement** describes an outcome the system must deliver. It says nothing about how. “Return a valid route between two locations, or report that none exists” is a requirement.

An **algorithm** is the step-by-step procedure used to deliver that outcome. It is independent of any programming language and independent of any platform. Breadth-first search is an algorithm. So is Dijkstra’s. Both can satisfy the same requirement; they make different tradeoffs in doing so.

An **implementation** is the concrete realization of an algorithm in a specific language, on a specific platform, using specific libraries. The same algorithm may have many implementations, differing in performance, readability, or the constraints of the runtime they target.

One requirement can be satisfied by multiple algorithms. One algorithm can be implemented in multiple ways. Mixing these levels produces conversations that cannot be resolved because the participants are arguing about different things. A debate about whether to "use Dijkstra" that is really a debate about whether weighted path cost is the right objective is a debate that will not end until someone names the level at which the disagreement lives.

For our running systems, the separation looks like this:

- **Requirement:** return a valid path, or report that no path exists.
- **Algorithm A:** breadth-first search, which finds the path with fewest hops on an unweighted graph.
- **Algorithm B:** Dijkstra's algorithm, which finds the path with minimum total cost on a weighted graph.
- **Implementation:** the same algorithm compiled to JavaScript for a browser simulation and to Java for a backend service.

The requirement is the same across all four. The algorithms answer different versions of "best." The implementations are interchangeable as far as the model is concerned.

11.2. Correctness Is Not Approximate

A procedure that usually works is not an algorithm. It is a heuristic with a good track record on the inputs that have been tested. This distinction matters because "usually works" provides no basis for reasoning about inputs that have not been tested — which is precisely the class of inputs that will be encountered in production.

An algorithm is acceptable only when its guarantees are explicit and checkable. For each of our systems, the first question to ask of any proposed algorithm is not *does it run?* but *what does it guarantee?*

For the delivery network: does the returned route use only traversable edges? If no route exists, is failure reported explicitly, or does the algorithm return an empty path that the caller must interpret? These are not implementation details — they are the algorithm's contract with the rest of the system.

For the knowledge engine: does the ranking follow a defined scoring rule, or does it reflect whatever the implementation happened to produce? When evidence is missing or contradictory, is that handled intentionally — with a defined fallback behavior — or silently, with whatever the code happens to do by default?

For the virtual world: does each update step preserve the world's invariants? When multiple objects must be updated in the same tick, is the order deterministic? An algorithm whose output depends on update order is not an algorithm with a correctness guarantee. It is a procedure whose behavior is a function of timing.

11.3. From Requirement to Algorithm

The following requirement will serve as a worked example:

> *"Find the best route quickly."*

This sentence is a starting point, not a specification. "Best" is undefined. "Quickly" is unmeasured. Before any algorithm can be chosen, several prior questions must be answered.

Step 1: Define valid input. The start and destination identifiers must refer to locations that exist in the map. The map itself must be internally consistent — no edges referencing locations absent from the node set. An algorithm given malformed input is not obligated to produce meaningful output, but the boundary between valid and invalid input must be stated explicitly, before the algorithm is written.

Step 2: Define the output contract. Three outcomes are possible: a path was found, no path exists, or the input was invalid. Each must be represented distinctly. An algorithm that returns an empty list for both "no path found" and "input was invalid" forces the caller to guess which situation applies. The output contract should be: `FOUND(path)` where every edge in the path is traversable, `UNREACHABLE` when the destination cannot be reached from the start, and `INVALID_INPUT` when the inputs do not satisfy the preconditions.

Step 3: Define the objective explicitly. For this version of the system, "best" means fewest hops. This is a decision, and it should be recorded as one. A later version might define "best" as minimum travel time, minimum energy use, or maximum reliability. Those are different objectives requiring different algorithms. The current objective is fewest hops, and the algorithm will be chosen to match it — not the other way around.

Step 4: Choose an algorithm that matches the objective. Fewest hops on an unweighted graph is exactly what breadth-first search computes. The match between objective and algorithm is not incidental — it is the reason BFS is the right choice here and would be the wrong choice if the objective were minimum weighted cost.

Step 5: Declare failure semantics. Unknown nodes produce `INVALID_INPUT`. A graph in which no path connects start to destination produces `UNREACHABLE`. These cases are handled intentionally.

The algorithm does not fall through to undefined behavior when they occur.

Step 6: Encode the contract. The algorithm's preconditions become `require` clauses. Its output guarantee becomes an `ensure` clause. The set of valid status values becomes an invariant on the result type. The contract is now part of the code, not a comment in a design document that may or may not reflect what the code actually does.

11.4. An Algorithm in Code

```
class Route_Result
create
  make(status, path: String) do
    this.status := status
    this.path := path
  end
feature
  status: String
  path: String
invariant
  valid_status:
    status = "FOUND" or
    status = "UNREACHABLE" or
    status = "INVALID_INPUT"
end

class Route_Algorithm
feature
  compute(start_loc, dest_loc: String): Route_Result
    require
      inputs_present: start_loc /= ""
                      and dest_loc /= ""
    do
      if start_loc = dest_loc then
        result := create Route_Result.make("FOUND", start_loc)
      elseif start_loc = "A" and dest_loc = "C" then
        result := create Route_Result.make("FOUND", "A->B->C")
      else
        result := create Route_Result.make("UNREACHABLE", "")
      end
    ensure
      status_is_declared:
        result.status = "FOUND" or
        result.status = "UNREACHABLE" or
        result.status = "INVALID_INPUT"
    end
end
```

This sketch is deliberately minimal — the hardcoded paths are a placeholder for a graph traversal that is not yet the point. What deserves attention is the structure surrounding the algorithm rather than its interior.

`Route_Result` carries an invariant that makes it impossible to construct a result with an undeclared status. The `compute` operation has a precondition that rejects empty inputs before any traversal begins.

Its postcondition guarantees that whatever the interior of the algorithm does, the returned status will always be one of the three declared values. The contract is enforced at the boundary, independently of the implementation — which means it will continue to hold when the hardcoded cases are replaced by a real graph traversal, and will continue to hold again when that traversal is later replaced by a faster one.

This is the pattern: specify the contract first, implement the interior second, verify that the contract holds regardless of what the interior does.

11.5. Three Ways Algorithm Design Goes Wrong

Calling any working script an algorithm. A script that produces correct output for the inputs that have been tried is not an algorithm with a correctness guarantee — it is a program with a favorable history. The difference becomes visible when the inputs change. An algorithm whose steps are stated explicitly and whose guarantees are declared can be analyzed for correctness. A script whose behavior emerges from accumulated control flow can only be tested, and testing covers only the cases already imagined. The recovery is to write language-agnostic steps before writing code, and to treat the code as an implementation of an algorithm rather than as the algorithm itself.

Undefined failure behavior. An algorithm that returns null, an empty list, or a sentinel value for both "success with empty result" and "operation failed" forces every caller to guess which situation applies — and different callers will guess differently. The system accumulates implicit conventions about what empty results mean, conventions that are never fully consistent and that break silently when a new caller is added. The recovery is to declare distinct output statuses for each possible outcome and to test each one explicitly.

Ambiguous objectives. An algorithm chosen to optimize for one criterion will perform incorrectly by the standard of a different one. When "best route" is never defined to mean fewest hops, minimum cost, or fastest traversal, the team cannot agree on which algorithm is correct because they are not yet asking the same question. The recovery is to encode one objective explicitly in the current specification, document the alternatives that were considered and deferred, and treat a change in objective as a design decision that requires revisiting the algorithm — not a bug fix in the existing one.

11.6. Quick Exercise

Choose one operation in your system and write a complete algorithm specification in five parts: the valid input definition, the output contract (naming each possible outcome distinctly), the objective function, a language-agnostic description of the steps, and one contract you will encode in the implementation.

Then apply this test: give your specification to another engineer and ask them to implement it independently. If their implementation makes different choices about failure behavior, output representation, or what the objective optimizes for, the specification is not yet precise enough to be called an algorithm description. Find the ambiguity and resolve it before writing code.

11.7. Takeaways

- An algorithm is a finite, explicit procedure with declared guarantees — not any procedure that produces plausible output on familiar inputs.
- Requirement, algorithm, and implementation are three distinct levels. Conflating them makes design conversations unresolvable and makes changes expensive.
- Correctness means guarantees that are explicit and checkable. A procedure that usually works under normal conditions is not correct in any useful sense of the word.
- Failure semantics are part of the algorithm's contract. Each distinguishable failure mode deserves a distinct, named output.
- The objective an algorithm optimizes must be stated before an algorithm is chosen. Different objectives require different algorithms; changing the objective silently is a design error, not an implementation detail.

Chapter 12 examines decomposition: how to take a complex algorithmic problem and divide it into pieces that can be designed, implemented, and tested independently. The discipline of decomposition is what separates algorithms that can be reasoned about from algorithms that can only be run.

12. Breaking Problems Apart

Complex algorithms rarely fail because a single line is wrong. They fail because too many responsibilities have been packed into one place, until the code can no longer be read as an account of what it does. The logic is present, but it is not separable — understanding any one part requires understanding all the others simultaneously, and changing any one part risks breaking something that was never intended to change at all.

Decomposition is the discipline of dividing a problem into subproblems, each with a clear responsibility and an explicit contract governing what it receives and what it returns. A well-decomposed algorithm can be understood one piece at a time, tested in isolation, and extended by adding or replacing pieces rather than rewriting the whole. These are not stylistic virtues. They are the properties that determine whether an algorithm can be maintained as requirements change.

12.1. The Test of a Decomposition

A decomposition is not just a list of functions. Any working code can be broken into functions. The question is whether the pieces reflect the actual structure of the problem — whether each piece has a single, nameable responsibility, a contract that can be stated without reference to the implementation, and an interface that allows it to be understood and tested independently.

Three questions test whether a decomposition is genuine:

What input does this piece require? Not the inputs it happens to receive, but the inputs it actually depends on to do its job. A piece that reaches outside its inputs to consult shared state is not a piece with a clear interface — it is a piece with hidden dependencies that will produce surprises when the shared state changes.

What output does it guarantee? Not what it usually returns, but what it is obligated to return for any valid input. A piece whose output format varies depending on internal conditions it does not expose is not a piece with a stable contract.

What is the one responsibility it owns? If a piece has two reasons to change — if updating the scoring logic and updating the output format both require touching the same code — the decomposition is incomplete. A piece with multiple responsibilities is multiple pieces that have not yet been separated.

12.2. Four Decomposition Patterns

Certain problem structures recur often enough that their decompositions have names. Recognizing these patterns reduces the work of decomposition from an act of invention to an act of identification.

Pipeline. A sequence of stages where each stage takes the output of the previous one as its input. The stages are ordered, each transforms its input in one defined way, and the final stage produces the result the caller needs. Pipelines are the natural decomposition for search, ranking, data transformation, and simulation loops — any problem where the solution is a chain of refinements applied to an initial input.

Strategy boundary. A stable interface around a swappable algorithm. The caller knows what the interface promises; it does not know or care which algorithm sits behind it. This pattern is appropriate when the objective the algorithm optimizes may change — when the current requirement is fewest hops but a future requirement may be minimum cost — or when the algorithm must vary by deployment context. The strategy boundary keeps the interface stable while allowing the implementation to evolve.

Guard, core, commit. Validate inputs and preconditions before any computation begins. Perform the core computation on verified data. Apply state changes only after the computation has succeeded. This separation prevents the most common class of partial-failure bugs: the operation that updates one piece of state and then fails before updating the rest. When the guard, core, and commit stages are distinct, it becomes possible to reason about each transition step — and to add compensation behavior when a commit cannot be completed.

Domain and infrastructure split. Algorithm logic and storage, transport, or rendering concerns belong in separate pieces. An algorithm that directly queries a database or formats its output for a specific UI is an algorithm whose behavior cannot be tested without the database or the UI. Separating domain logic from infrastructure concerns is not just a testing convenience — it is what makes the algorithm's correctness independent of the platform it runs on.

12.3. A Pipeline in Practice

The following requirement will serve as a worked example:

"Return the top relevant notes for a query."

A naive implementation of this feature tends to mix all its concerns into a single method: parse the query, find candidate documents, score each one, sort by score, filter out weak matches, and format the results. This implementation may be correct, but it cannot be tested in pieces, cannot have its scoring logic replaced without touching the pipeline structure, and cannot be understood without reading the entire method.

A decomposed design names each stage and gives it an explicit contract:

1. `tokenize_query(query)` — convert the raw query string into a normalized set of search terms.
2. `collect_candidates(tokens, index)` — retrieve the documents that contain at least one query term.
3. `score_candidate(candidate, tokens)` — compute a relevance score for a single candidate document.
4. `rank_candidates(scored)` — order the scored candidates from most to least relevant.
5. `filter_by_threshold(ranked, min_score)` — remove candidates whose scores fall below an acceptable floor.
6. `render_results(filtered)` — convert the filtered list into the format the caller expects.

Each stage now has a single responsibility. The scoring logic lives entirely in stage 3 and can be replaced — substituting a weighted term-frequency model for a simple match counter — without touching any other stage. The filtering threshold in stage 5 can be adjusted without affecting scoring. The output format in stage 6 can change without affecting the ranking. The decomposition has made each axis of change independent of the others.

12.4. A Decomposition in Code

```
class Search_Algorithm
feature
  tokenize(query: String): String
    require
      query_present: query /= ""
    do
      result := query
    ensure
      non_empty_tokens: result /= ""
    end

  score(doc_text, tokens: String): Integer
```

```
    require
      inputs_present: doc_text /= ""
                      and tokens /= ""
    do
      if doc_text = tokens then
        result := 100
      else
        result := 10
      end
    ensure
      non_negative: result >= 0
    end

  choose_top(doc1, doc2, query: String): String
    require
      docs_present: doc1 /= "" and doc2 /= ""
                    and query /= ""
    do
      let t: String := tokenize(query)
      let s1: Integer := score(doc1, t)
      let s2: Integer := score(doc2, t)

      if s1 >= s2 then
        result := doc1
      else
        result := doc2
      end
    ensure
      from_inputs: result = doc1 or result = doc2
    end
end
```

`tokenize` and `score` are independent operations, each with its own contract. `choose_top` composes them: it calls `tokenize` to normalize the query, calls `score` twice to evaluate each candidate, and selects based on the scores. The postcondition on `choose_top` — `result = doc1 or result = doc2` — guarantees that the result comes from the inputs and not from some other source, which is a meaningful guarantee even in this simplified sketch.

Notice that `choose_top` does not know how tokenization works or how scoring is computed. It knows only what each operation guarantees. This is what makes composition safe: each piece can be reasoned about through its contract, and a piece that satisfies its contract can be replaced by any other piece that satisfies the same contract.

12.5. Four Ways Decomposition Goes Wrong

Decomposing by syntax rather than responsibility. A function named `process`, `handle`, or `do_stuff` is not a decomposition — it is code divided at arbitrary boundaries without regard to the structure of the problem. The sign of genuine decomposition is that each piece's name states its responsibility, and its contract can be written without reading its implementation. If naming a piece requires describing what it does rather than what it is for, the boundary was drawn in the wrong place.

Over-fragmentation. The opposite error is to divide a problem into so many small pieces that the decomposition itself becomes an obstacle to understanding. When a function does nothing but delegate to another function of the same name with a slightly different signature, it adds a layer of indirection without adding a layer of meaning. The test for whether a layer is genuine is whether it improves the ability to reason about the problem. Layers that exist only to satisfy a structural preference should be merged back.

Hidden coupling between stages. A stage that depends on the internal state of the stage before it is not an independent piece with a clear interface — it is a piece that has been separated syntactically but remains coupled semantically. The only input a stage should depend on is what is passed to it explicitly. Implicit dependencies on shared mutable state, on call order, or on properties of the implementation above or below will produce failures that are difficult to localize because they cannot be reproduced by testing the stage in isolation.

Missing contracts at stage boundaries. A decomposed algorithm whose stages do not have explicit contracts has most of the costs of decomposition and few of its benefits. When an intermediate result is malformed — when `tokenize` returns an empty string for a non-empty query, or when `score` returns a negative value — the failure will propagate through subsequent stages and surface far from its origin. Contracts at stage boundaries catch these failures at the point where they occur, before they have been compounded by later processing.

12.6. Quick Exercise

Take one function in your current codebase that mixes more than one concern — a function that validates, computes, and formats, or that queries, transforms, and updates — and decompose it into three to six stages.

For each stage, write: the input contract, the output contract, and the single responsibility it owns. Then identify which stage you could replace with a different implementation without changing any other stage. If no stage is independently replaceable, the decomposition still has hidden coupling that the exercise has not yet surfaced.

12.7. Takeaways

- Decomposition is a structural discipline, not a stylistic one. Its purpose is to make algorithms locally understandable, independently testable, and safely modifiable.
- A genuine decomposition passes three tests: each piece has a nameable input, a guaranteed output, and a single responsibility.
- Four patterns cover most cases: pipeline, strategy boundary, guard-core-commit, and domain-infrastructure split. Recognizing the pattern makes the decomposition easier to find.
- Composition is safe when it is based on contracts, not on knowledge of implementation details. A piece that knows only what its dependencies guarantee can be reasoned about independently of how those dependencies are implemented.
- A stage boundary without a contract is a seam that looks like decomposition and behaves like coupling.

h*Chapter 13 applies the decomposition principle to a specific and important class of problems: those whose structure is self-similar. Recursive algorithms are decompositions in which a problem is divided into a smaller instance of the same problem. Understanding recursion as a special case of decomposition — rather than as a separate technique — is what makes it a reliable tool rather than an occasional trick.*

13. Thinking Recursively

Chapter 12 established that decomposition means dividing a problem into subproblems with clear interfaces and distinct responsibilities. Recursion is a special and important case of this principle: it applies when the subproblem has the same shape as the original problem, only smaller.

This is not a trick or an advanced technique reserved for difficult cases. It is a natural way to express algorithms over data whose structure is self-referential — trees, graphs, nested containment hierarchies, any domain where the whole is made of parts that resemble the whole. In these domains, a recursive algorithm often expresses the solution more directly and more transparently than an iterative one, because the structure of the algorithm matches the structure of the data.

The two are not always interchangeable in practice. A recursive algorithm that operates on deeply nested data may exhaust the call stack before it reaches a base case. A recursive algorithm that revisits already-explored nodes in a graph may run forever. Understanding recursion well means understanding not just how to write recursive algorithms, but when they are safe, when they are not, and what must be in place to make them reliable.

13.1. The Shape of a Recursive Algorithm

Every correct recursive algorithm has exactly two components, and both must be present for the algorithm to be correct.

The **base case** is the simplest possible instance of the problem — the version that can be solved directly, without recursion. For counting reachable nodes in a graph, the base case is a node that has already been visited: the answer is zero, because the node has already been counted. For computing the depth of a tree, the base case is a leaf: the depth is one. The base case is not optional. An algorithm without a reachable base case does not terminate.

The **recursive step** reduces the original problem to one or more smaller instances of the same problem, solves those instances by calling itself, and combines their results into the answer for the original. The word *smaller* is critical. Each recursive call must bring the computation measurably closer to the base case. If the recursive step does not make progress — if a call with input of size n produces a recursive call with input also of size n — the algorithm will not terminate either.

The relationship between these two components mirrors the structure of a proof by induction: the base case establishes correctness for the smallest input; the recursive step establishes that correctness for any input of size n follows from correctness for inputs smaller than n. An algorithm that satisfies both components is correct for all valid inputs. An algorithm that satisfies only one is not an algorithm — it is a procedure that works on some inputs by accident.

13.2. When Recursion Fits

The choice between recursion and iteration is a choice about how to express the algorithm's logic, and it should be made based on which expression more clearly reflects the structure of the problem.

Recursion tends to be the more natural expression when the data is hierarchical or graph-like: nested structures where each node contains other nodes of the same type, trees where each subtree is itself a tree, graph traversals where each neighbor is itself a node to be explored. In these cases, the recursive structure of the algorithm corresponds directly to the recursive structure of the data, and the algorithm reads as a statement about the data rather than as a sequence of bookkeeping steps.

Iteration tends to be the more natural expression for linear workflows: processing a sequence from start to finish, accumulating a result as you go. A loop that sums the elements of a list is not hiding a self-similar structure; it is doing the same thing repeatedly, and saying so directly is clearer than encoding it as recursion.

Choosing recursion for a linear workflow, or iteration for a hierarchical one, produces code that works but does not communicate. The algorithm's structure and the problem's structure are misaligned, and a reader must work to find the correspondence that the code does not make visible.

13.3. Graph Recursion and the Necessity of Bounds

The three systems developed throughout this book are all graph-like in some respect, and all three require recursive reasoning that is careful about bounds.

In the delivery network, exploring alternate route branches requires recursion over the graph of locations and paths. But a delivery network may contain cycles — two locations connected to each other through multiple paths — and a recursive traversal that does not track which nodes have been visited will follow those cycles indefinitely.

In the knowledge engine, traversing the network of related documents to find indirectly relevant material requires recursion over the document graph. The same concern applies: documents may be mutually related, cycles are possible, and depth must be bounded both to prevent infinite traversal and to prevent the search from expanding to the entire document collection.

In the virtual world, processing the containment hierarchy — zones contain regions, regions contain objects — is naturally recursive because the hierarchy is a tree. Trees have no cycles, but depth bounds still matter for performance, and deterministic traversal order matters for correctness: if the same hierarchy is processed in different orders on different runs, the simulation is not reproducible.

In all three domains, recursion without bounds is unsafe. The pattern that makes graph recursion safe is always the same: maintain a set of visited nodes, check it before recursing, and update it before proceeding. Optionally, enforce a depth limit as a second line of defense against graphs that are technically acyclic but are too deep for the call stack to handle.

13.4. From Requirement to Recursive Algorithm

Consider the requirement:

> *"Count the number of locations reachable from a starting node."*

The problem has a self-similar structure. A node is reachable from the start if it is the start itself, or if it is reachable from any neighbor of the start. The count of reachable nodes from a given node is one (for the node itself) plus the sum of reachable nodes from each unvisited neighbor. This is a recursive definition, and it maps directly to a recursive algorithm.

Base case. If the current node has already been visited, return zero. The node has already been counted, and counting it again would produce an incorrect total.

Recursive step. Mark the current node as visited. Count it (contributing one to the total). For each neighbor, add the count returned by a recursive call on that neighbor. Return the sum.

Progress metric. The recursion terminates because the visited set grows with each call, and the graph is finite. Every recursive call either finds a visited node and returns immediately, or visits a new node and reduces the number of unvisited nodes by one. The total number of recursive calls is bounded by the number of nodes in the graph.

Cycle guard. The visited set is the cycle guard. A node that has been visited will not be visited again, which breaks any cycle the graph might contain.

These four elements — base case, recursive step, progress metric, cycle guard — are the complete specification of the algorithm. The implementation follows directly from them.

13.5. A Recursive Algorithm in Code

```
class Reachability
feature
  visited: String

  count_from(node: String): Integer
    require
      node_present: node /= ""
    do
      -- Teaching sketch: visited is encoded as a
      -- simple string marker set.
      -- A full implementation would use a
      -- dedicated collection type.
      if visited = node then
        result := 0
      elseif node = "Z" then
        visited := node
        result := 1
      else
        visited := node
        result := 1 + count_from("Z")
      end
    ensure
      non_negative_count: result >= 0
    end
end
```

Read this sketch against the four elements above. The first branch of the conditional is the base case: the node has been visited, return zero. The remaining branches are the recursive step: mark the node visited, count it, and add the result of the recursive call. The `visited` field serves as the cycle guard. The progress metric is implicit — each call either terminates immediately (the base case) or adds a new node to `visited` before recursing — but in a full implementation it would be worth making the progress argument explicit, either in a comment or in an assertion.

The postcondition `result >= 0` is minimal but meaningful. A negative count is nonsensical, and guaranteeing non-negativity allows the caller to depend on this property rather than defensively check for it. As with the decomposed stages in Chapter 12, the contract at the boundary is what allows the piece to be composed with other pieces safely.

13.6. Four Ways Recursive Algorithms Fail

Missing base case. Without a reachable base case, the recursion has no stopping condition. The call stack grows without bound until the runtime exhausts its stack space and the program terminates with an error. The remedy is to write and test the base case before writing the recursive step. An algorithm whose base case is correct for the smallest input is an algorithm that has been verified to stop.

No progress metric. A recursive call that does not reduce the problem — that calls itself with the same input, or with an input of the same size — will not reach the base case. This is a subtler failure than a missing base case, because the code may look correct: there is a base case, and there are recursive calls, but the calls never get closer to the base. Defining a progress metric explicitly — the number of unvisited nodes, the remaining depth, the size of the input — and verifying that each recursive call strictly reduces it is the discipline that prevents this failure.

Ignoring cycles. In any graph that may contain cycles, a recursive traversal without a visited set will follow cycles indefinitely. The algorithm may look correct on acyclic inputs and fail silently on cyclic ones. Since the presence or absence of cycles is a property of the data, not of the algorithm, the safe assumption is that cycles may exist and the visited set is always required.

Recomputing identical subproblems. Some recursive algorithms naturally decompose a problem into overlapping subproblems — the same subproblem arises from multiple different paths through the recursion. Without memoization, each instance is recomputed independently, and the total work grows exponentially in the number of overlapping subproblems. The signature of this failure is an algorithm that is correct on small inputs and runs for an unacceptable amount of time on larger ones. The remedy is to record the result of each subproblem the first time it is computed and return the recorded result on subsequent calls.

13.7. Quick Exercise

Design one recursive routine for a problem in your system and specify it completely with five components: the base case, the recursive step, the progress metric, the cycle guard (or an explicit argument that none is needed), and a worst-case bound on the depth of recursion.

Then test it against three inputs: a trivial input that exercises only the base case, a normal input that exercises the recursive step, and an adversarial input that contains a cycle or approaches the depth bound. If the adversarial input produces unbounded behavior, the specification is incomplete. Revise it before writing the implementation.

13.8. Takeaways

- Recursion is decomposition applied to self-similar structure. It is not a general-purpose technique but a natural fit for hierarchical and graph-like problems.
- Every correct recursive algorithm has a base case and a progress metric. Both are mandatory. An algorithm missing either one is not correct.
- Graph recursion requires cycle guards. The visited set is not an optimization — it is the mechanism that prevents the algorithm from following cycles indefinitely.
- Overlapping subproblems require memoization. An algorithm that recomputes the same subproblem repeatedly is correct but not acceptable; recognizing the overlap is the first step toward fixing it.
- The four elements of a safe recursive design — base case, recursive step, progress metric, cycle guard — are a complete specification. The implementation follows from them directly.

Chapter 14 shifts from the correctness of algorithms to their cost. A correct algorithm that is too slow for its inputs is not a usable algorithm. Understanding how algorithmic cost grows with input size is what makes it possible to choose between correct alternatives — and to predict, before deployment, whether an algorithm will hold up under real conditions.

14. Measuring Algorithm Behavior

An algorithm can be correct and still fail. Correctness means the algorithm produces the right answer for every valid input. It says nothing about how long that takes, or how much memory is consumed, or what happens when the inputs are larger or more numerous than the developer imagined when writing the code. A correct algorithm that takes thirty seconds to respond to a user query is not a usable algorithm. A correct algorithm that processes a hundred records perfectly and exhausts available memory on a hundred thousand is not a deployable one.

Algorithm engineering has two gates, and a candidate must pass both. The first gate is correctness — the subject of Chapters 11 through 13. The second is cost: how does the algorithm's resource consumption grow as inputs grow, and does that growth remain acceptable under the conditions the system will actually face? This chapter is about the second gate.

14.1. What Cost Means

Cost is not a single number. It is a profile of several distinct quantities, each of which matters in different contexts and under different conditions.

Latency is the time a single operation takes from start to finish. For the delivery system, the relevant latency is how long route recomputation takes when a path becomes blocked and a new route must be found while the robot waits. For the knowledge engine, it is how long a single query takes to return ranked results. Latency determines whether a system feels responsive.

Throughput is the number of operations the system can complete per unit of time. A knowledge engine that handles one query per second correctly is not useful if the system receives a hundred queries per second. Throughput determines whether a system can sustain its intended load.

Memory footprint is the space the algorithm requires, both for its inputs and for any intermediate structures it constructs. An index that makes query latency acceptable may require more memory than the deployment environment provides. A traversal that accumulates visited nodes may grow unboundedly on large graphs.

Worst-case behavior is what happens when the algorithm encounters its most expensive possible input. An algorithm with excellent average performance and catastrophic worst-case performance is an algorithm that will occasionally fail in production in ways that are difficult to predict and reproduce.

Tail behavior is the distribution of costs at the high end — the P95 and P99 latencies, the requests that take far longer than the median. Average latency can be acceptable while the slowest five percent of requests take long enough to damage the user experience. Tail behavior is often more important than average behavior for systems with interactive users.

14.2. Two Kinds of Reasoning

Understanding algorithmic cost requires two complementary forms of reasoning, neither of which is sufficient on its own.

Asymptotic analysis characterizes how cost grows as input size grows, independent of constants. An algorithm that examines every element of a collection once runs in time proportional to the size of the collection — $O(N)$ in the standard notation. One that examines every pair of elements runs in $O(N^2)$. One that halves the remaining search space at each step runs in $O(\log N)$. These characterizations tell us which algorithms will remain acceptable as inputs scale and which will become unacceptable. An $O(N^2)$ algorithm on a collection that grows from thousands to millions of elements will increase its cost by a factor of a trillion; no constant-factor improvement to the implementation will rescue it.

Empirical measurement captures the behavior of a specific implementation on a specific workload. Asymptotic analysis suppresses constants, but constants matter: a linear scan over a small collection held entirely in cache may be faster in practice than a logarithmic lookup that requires following pointers through memory. The theoretical advantage of a more complex algorithm does not materialize until the input is large enough for the asymptotic term to dominate the constant. Without measurement, it is not possible to know where that threshold lies for a particular algorithm on a particular machine.

The discipline is to use asymptotic reasoning to avoid structurally bad choices, and to use measurement to validate that the remaining candidates behave acceptably on real workloads. Neither alone is sufficient. Asymptotic analysis without measurement is theory that may not match practice. Measurement without asymptotic analysis is benchmarking without the ability to predict behavior outside the cases measured.

14.3. From Requirement to Cost Model

The following requirement will serve as a worked example:

"Return the top ten relevant documents quickly for every query."

"Quickly" is not yet a specification, but it points toward one. The worked path below converts it into a cost model.

Step 1: Identify candidate approaches and their asymptotic costs. Two natural candidates are a full scan that scores all documents and selects the top ten, and an index-based approach that generates a small candidate set before scoring. The full scan is O(N) per query, where N is the total number of documents. The index-based approach is O(k) for scoring, where k is the size of the candidate set — and for most queries on a well-constructed index, k is much smaller than N. Asymptotic reasoning gives a strong preliminary reason to prefer the indexed approach at scale, without yet committing to any specific implementation.

Step 2: State explicit performance targets. "Quickly" must become a number: P95 query latency under 150 milliseconds, for example. A memory budget for the index. An update-time budget for incorporating new documents into the index. Without explicit targets, there is no way to determine whether a benchmark result is acceptable or whether an optimization has achieved its goal.

Step 3: Design a benchmark that covers the real distribution. A benchmark that tests only average queries on average-sized collections measures the performance of the typical case and nothing else. A complete benchmark must cover the normal case, the heavy-tail case (queries that match an unusually large fraction of the document collection), and the adversarial spike (sudden bursts of queries at rates above the expected average). Production behavior is determined by all three.

Step 4: Define fallback behavior for the expensive case. When a query matches a very large candidate set, the cost of the indexed approach approaches the cost of a full scan. Rather than allowing this to produce unbounded latency, the algorithm should apply a bounded heuristic mode when the candidate set exceeds a defined threshold — returning a partial but valid result with a declared status indicating that the response is bounded rather than complete. Fallback behavior must be designed, not improvised when the first spike occurs in production.

Step 5: Define a redesign trigger. Acceptable performance today may become unacceptable as the document collection grows. The cost model should include an explicit condition — a collection size, a query rate, a latency measurement — at which the current design is expected to stop meeting its targets and must be revisited. Without this, the system will be redesigned in response to a crisis rather than in anticipation of one.

14.4. A Strategy Choice in Code

```
class Strategy_Selector
feature
  estimate_linear(n: Integer): Integer
    require
      non_negative_n: n >= 0
    do
      result := n
    ensure
      non_negative_result: result >= 0
    end

  estimate_indexed(k: Integer): Integer
    require
      non_negative_k: k >= 0
    do
      result := k
    ensure
      non_negative_result: result >= 0
    end

  choose(n, k: Integer): String
    require
      valid_inputs: n >= 0 and k >= 0
    do
      if estimate_indexed(k) < estimate_linear(n) then
        result := "INDEXED"
      else
        result := "LINEAR"
      end
    ensure
      known_strategy: result = "INDEXED"
                      or result = "LINEAR"
    end
end
```

The `estimate_linear` and `estimate_indexed` operations are simplified models of the two approaches' costs. `choose` selects between them based on a comparison of those estimates. The post-condition on `choose` guarantees that the result is always one of two declared strategy names — never an undefined value that a caller would have to interpret.

What matters most in this sketch is that the strategy selection is explicit and testable. A strategy choice buried in conditional logic inside the main algorithm is invisible: it cannot be reasoned about independently, cannot be tested in isolation, and cannot be swapped out when a new approach becomes available. An explicit strategy selector has all three properties. It makes the cost model a first-class element of the design rather than an assumption hidden in the implementation.

14.5. Five Ways Cost Reasoning Fails

Optimizing before correctness. An algorithm that produces incorrect results faster is not an improvement. Every optimization changes the implementation, and every change to the implementation is an opportunity to introduce a correctness failure. The discipline is to establish and verify correctness guarantees first, and to optimize only under tests that would detect any regression in those guarantees. An optimization that passes all correctness tests can be trusted; one that predates them cannot.

Treating asymptotic analysis as complete. Big-O notation suppresses constants, lower-order terms, and the effects of memory hierarchy, cache behavior, and branch prediction — all of which matter on real hardware with real workloads. An algorithm with a better asymptotic complexity than its competitor may be slower in practice on the input sizes that actually occur. Asymptotic analysis is a necessary tool for avoiding structurally bad decisions. It is not a substitute for measurement.

Benchmarking only typical inputs. An algorithm that performs well on the average case and poorly on the adversarial case will produce acceptable measurements in the lab and unacceptable behavior in production. The adversarial case — the query that matches everything, the graph that contains a very long path, the burst of simultaneous requests — is the case that determines whether the system holds up under real conditions. Benchmark designs that omit it are not measuring the right thing.

Ignoring tail latency. The median query response time may be well within the target while the slowest five percent of queries take ten times as long. For a system with many users, the slowest five percent is not a rounding error — it is a population of users with a consistently poor experience. Designing to the median and ignoring the tail is designing for a performance profile that does not match the experience of all users.

No capacity narrative. An algorithm that meets its performance targets today on a collection of one million documents may fail to meet them next year on a collection of ten million. Without a model of how costs grow and an explicit threshold at which the current design must be reconsidered, the team will discover this failure in production. A capacity narrative — a documented projection of when current performance targets will be exceeded under expected growth — converts a future crisis into a scheduled engineering decision.

14.6. Quick Exercise

Choose one algorithm in your system and write a cost brief with seven components: the primary operation whose cost you are characterizing, the expected input size range in production, an asymptotic estimate of the cost, the dominant constant factors you would expect to observe in measurement, a benchmark design covering normal, edge, and adversarial inputs, a fallback policy for the case where cost exceeds the acceptable threshold, and a redesign trigger condition.

Then apply this test: give the brief to a teammate and ask them to predict the failure mode — the condition under which the algorithm will first become unacceptable. If they cannot predict it from the brief, the brief is not yet a cost model. Find what is missing and add it.

14.7. Takeaways

- Correctness and cost are dual constraints. An algorithm that passes only one gate is not ready for production.
- Cost is a profile, not a number: latency, throughput, memory footprint, worst-case behavior, and tail behavior each matter in different contexts.
- Asymptotic analysis identifies structurally bad choices. Empirical measurement validates that the remaining choices behave acceptably on real workloads. Both are required.
- Tail behavior is often more important than average behavior. Design to the distribution, not the median.
- A capacity narrative — a projection of when current performance targets will be exceeded — converts a future crisis into a scheduled decision.

Part III has now built the algorithmic toolkit: a precise definition of what an algorithm is, the discipline of decomposition, recursive design over self-similar structure, and the cost reasoning that determines whether a correct algorithm is also a practical one. Part IV applies these tools to data structures — the concrete representations that make algorithms efficient at scale.

15. Algorithm Lab — The First Experiments

15.1. Subtitle

Testing algorithm ideas on small systems before formal complexity notation.

Lab Focus Run controlled experiments, collect evidence, and compare algorithm choices under explicit assumptions.

15.2. Why This Lab Exists

Part III introduced algorithm thinking:

- define guarantees
- decompose work
- use recursion when structure fits
- reason about behavior under growth

Now we run controlled experiments.

The goal is not to chase micro-optimizations. The goal is to build engineering intuition by observing how different algorithm choices behave on the same task.

15.3. Lab Objectives

By the end of this lab, you should be able to:

- compare two algorithms for the same requirement
- collect simple evidence (operation counts, not just wall-clock time)
- explain when an approach is acceptable and when it breaks down
- preserve correctness contracts while changing strategy

15.4. Experiment Theme

Use one task, multiple strategies:

> "Find a target value in a sorted sequence."

Why this task:

- easy correctness criteria
- easy to vary input size
- clear contrast between algorithm behaviors

Strategies:

- linear scan
- binary search

We will track an explicit step counter to observe behavior directly.

15.5. Nex Implementation

Suggested file names:

- `algorithm_lab_1.nex`
- `algorithm_lab_1_main.nex`

If using the web IDE, place everything in one file and run `App.run`.

15.5.1. Result Type

```
class Search_Result
create
  make(found: Boolean, index: Integer, steps: Integer) do
    this.found := found
    this.index := index
    this.steps := steps
  end
feature
  found: Boolean
  index: Integer
  steps: Integer
invariant
  index_valid: index >= -1
  steps_non_negative: steps >= 0
end
```

15.5.2. Linear Search With Step Counter

```
class Linear_Search_Algo
feature
  find(a1, a2, a3, a4, a5, a6, a7, a8,
       target: Integer): Search_Result
    do
      let found: Boolean := false
      let idx: Integer := -1
      let steps: Integer := 1
      if a1 = target then
        found := true
        idx := 0
      elseif a2 = target then
        steps := steps + 1
        found := true
        idx := 1
      elseif a3 = target then
        steps := steps + 1
        found := true
        idx := 2
      elseif a4 = target then
        steps := steps + 1
        found := true
        idx := 3
      elseif a5 = target then
        steps := steps + 1
        found := true
        idx := 4
      elseif a6 = target then
        steps := steps + 1
        found := true
        idx := 5
      elseif a7 = target then
        steps := steps + 1
        found := true
        idx := 6
      elseif a8 = target then
        steps := steps + 1
        found := true
        idx := 7
      else
        steps := steps + 1
      end

      result := create Search_Result.make(found, idx, steps)
    ensure
      steps_recorded: result.steps >= 1
      valid_index: result.index >= -1
                   and result.index <= 7
    end
end
```

15.5.3. Binary Search With Step Counter

```
class Binary_Search_Algo
feature
  find(a1, a2, a3, a4, a5, a6, a7, a8,
       target: Integer): Search_Result
    require
      sorted_input:
```

```
        a1 <= a2 and a2 <= a3 and a3 <= a4 and
        a4 <= a5 and a5 <= a6 and a6 <= a7
        and a7 <= a8
    do
      if a4 = target then
        result := create Search_Result.make(true, 3, 1)
      elseif target < a4 and a2 = target then
        result := create Search_Result.make(true, 1, 2)
      elseif target < a2 and a1 = target then
        result := create Search_Result.make(true, 0, 3)
      elseif target > a2 and target < a4
             and a3 = target then
        result := create Search_Result.make(true, 2, 3)
      elseif target > a4 and a6 = target then
        result := create Search_Result.make(true, 5, 2)
      elseif target > a4 and target < a6
             and a5 = target then
        result := create Search_Result.make(true, 4, 3)
      elseif target > a6 and a7 = target then
        result := create Search_Result.make(true, 6, 3)
      elseif target > a7 and a8 = target then
        result := create Search_Result.make(true, 7, 4)
      else
        result := create Search_Result.make(false, -1, 4)
      end
    ensure
      steps_bounded: result.steps >= 1
                     and result.steps <= 4
      valid_index: result.index >= -1
                   and result.index <= 7
    end
end
```

15.5.4. Driver: Compare Behaviors

```
class App
feature
  run() do
    let lin: Linear_Search_Algo
     := create Linear_Search_Algo
    let bin: Binary_Search_Algo
     := create Binary_Search_Algo

    let l_hit: Search_Result
     := lin.find(3, 7, 10, 14, 19, 21, 25, 30, 30)
    let b_hit: Search_Result
     := bin.find(3, 7, 10, 14, 19, 21, 25, 30, 30)

    let l_miss: Search_Result
     := lin.find(3, 7, 10, 14, 19, 21, 25, 30, 11)
    let b_miss: Search_Result
     := bin.find(3, 7, 10, 14, 19, 21, 25, 30, 11)

    print("Linear hit steps: " + l_hit.steps)
    print("Binary hit steps: " + b_hit.steps)
    print("Linear miss steps: " + l_miss.steps)
    print("Binary miss steps: " + b_miss.steps)
  end
end
```

Expected pattern:

- both algorithms are correct
- binary search usually uses fewer steps on sorted inputs
- binary search depends on stronger preconditions (`sorted_input`)

15.6. Lab Tasks

15.6.1. Task A — Correctness Check

Run both strategies with:

- first element target
- middle element target
- last element target
- absent target

Record:

- found/not-found
- returned index
- steps

15.6.2. Task B — Constraint Violation

Call binary search with unsorted inputs and observe contract behavior.

Question:

- what failure signal do you get?
- how is this better than silent wrong results?

15.6.3. Task C — Extension

Add one more strategy for comparison:

- jump search, or
- interpolation-inspired search (if values are uniform)

Keep the same `Search_Result` format so results remain comparable.

15.7. Reflection Prompts

Use evidence from your run logs:

1. Which strategy has stricter input assumptions?
2. Which strategy is easier to reason about for correctness?
3. At what scale would you stop using linear search for this task?
4. Which contract prevented the most dangerous misuse?

15.8. Deliverables

- runnable Nex code for at least two strategies
- output log showing hit/miss comparisons
- short write-up of strategy tradeoffs
- one paragraph on how contracts influenced your design

15.9. Forward Link

This lab gives you concrete intuition for why data structures matter.

In Part IV, we study lists, sets/maps, trees, and graphs so algorithm choices are supported by the right representation.

Part IV.

Part IV — Organizing Data

16. Lists and Sequences

Part III developed the tools for algorithmic thinking: what an algorithm is, how to decompose one, how to design recursive algorithms safely, and how to reason about cost. Part IV asks a different question. Algorithms do not operate in the abstract — they operate on data that has been organized in some particular way, and the organization chosen determines which operations are cheap and which are expensive. The same algorithm can be fast or slow depending entirely on how its data is arranged.

We begin with lists and sequences, because they are the structure most teams reach for first. This is not always wrong — lists are genuinely the right choice for certain problems — but it is often unreflective. A team that uses lists because they are familiar, rather than because they fit the operations the system requires, will eventually discover the mismatch through degraded performance. The goal of this chapter is to make that choice conscious: to understand what lists are good at, where they become expensive, and how to recognize early which situation applies.

16.1. What a List Is

A list is an ordered collection: a sequence of elements in which position is meaningful and preserved. The order may reflect the sequence in which elements were added, a logical ranking, a timeline of events, or a processing priority. Whatever its source, the order is part of the list's content — it is not an artifact of the implementation but a fact about the data that the structure is responsible for maintaining.

Order gives lists their characteristic strengths. Iterating from start to finish is natural and efficient. The first element, the last element, and the next element after any given one are all immediately accessible. Lists represent timelines well, ranked outputs well, queues and logs well — any domain where the primary question is *what comes next?* or *in what order did these things occur?*

Order also defines the limits of a list's usefulness. A list organizes elements by position. It has no mechanism for organizing them by content. Finding an element by a property — by its identifier, by a key

— requires examining elements one at a time until the matching one is found. For a list of ten elements, this is inconsequential. For a list of ten million, it is the dominant cost of every lookup operation, and it grows linearly with the size of the collection.

16.2. The Cost of What Lists Cannot Do

Three operations become expensive on lists as collections grow:

Membership testing. Determining whether a particular element is present in a list requires, in the worst case, examining every element. There is no shortcut available from the list's structure alone. If membership testing is a frequent operation, a list is the wrong structure.

Lookup by key. Finding the element with a particular identifier requires a linear scan. If the element is near the front, the scan is short; if it is near the back or absent, the scan is long. The expected cost of a lookup grows proportionally with the size of the list.

Insertion and deletion in the middle. Adding or removing an element at an arbitrary position requires shifting every element after it. For a list where insertions and deletions are frequent and positions are arbitrary, this cost accumulates quickly.

These are not obscure operations. They are the dominant operations in a large class of systems. A system that stores delivery tasks in a list and repeatedly looks up tasks by identifier will degrade as the number of tasks grows. The degradation is predictable — it follows directly from the structure chosen — but it will arrive as a surprise if the structure was chosen without asking what operations the system actually performs.

This is the first scaling failure many systems encounter. It is also one of the most avoidable.

16.3. Choosing a Structure by Its Operations

The right question to ask before choosing a data structure is not *what kind of thing am I storing?* but *what operations will I perform on it most often?* The answer to the second question determines which structure will serve the system well.

For the delivery system, two operations on task collections matter. The first is displaying active tasks in the order they were created or dispatched — an order-sensitive operation that a list handles naturally. The second is fetching a specific task by its identifier when a robot requests its assignment or when a status update arrives — a key-sensitive operation that a list handles poorly.

The mismatch between these two operations and a single list structure is the central tension of this chapter. A list satisfies the first operation directly. It satisfies the second only at a cost that grows with the number of tasks. At small scale, this cost is invisible. As the system grows, it becomes the bottleneck.

The resolution — keeping a list for ordered display and introducing a separate index for keyed lookup — is the subject of Chapter 16. This chapter holds the tension open deliberately, because understanding why the list alone is insufficient is a prerequisite for understanding what the index adds.

16.4. A Sequence in Code

```
class Task
create
  make(id: String, status: String) do
    this.id := id
    this.status := status
  end
feature
  id: String
  status: String
invariant
  id_present: id /= ""
  valid_status:
    status = "PENDING" or
    status = "IN_TRANSIT" or
    status = "DELIVERED" or
    status = "FAILED"
end

class Task_Sequence
create
  make(t1: Task, t2: Task, t3: Task) do
    this.t1 := t1
    this.t2 := t2
    this.t3 := t3
  end
feature
  t1: Task
  t2: Task
  t3: Task

  find_by_id(task_id: String): String
    require
      id_present: task_id /= ""
    do
      if t1.id = task_id then
        result := t1.status
      elseif t2.id = task_id then
        result := t2.status
      elseif t3.id = task_id then
        result := t3.status
      else
```

```
        result := "NOT_FOUND"
      end
    ensure
      declared_result:
        result = "PENDING" or
        result = "IN_TRANSIT" or
        result = "DELIVERED" or
        result = "FAILED" or
        result = "NOT_FOUND"
    end

  ordered_ids(): String
    do
      result := t1.id + " -> " + t2.id
                + " -> " + t3.id
    ensure
      non_empty: result /= ""
    end
end
```

`Task_Sequence` is a fixed-size sequence of three tasks — intentionally minimal, but structurally honest about what a list provides. `ordered_ids` reflects the list's natural strength: it returns the elements in their defined order, and the cost of doing so is simply the cost of visiting each element once. `find_by_id` reflects the list's natural weakness: it examines each task in turn until it finds a match or exhausts the sequence. For three tasks, this is trivial. For three thousand, the same structure and the same operation would examine up to three thousand tasks per lookup.

The postcondition on `find_by_id` matters. It guarantees that the result is always one of five declared values — the four valid status strings or `NOT_FOUND` — and never an undefined or ambiguous return. This is the contract that makes `find_by_id` composable: callers can depend on the declared output without inspecting the implementation. When the implementation is later replaced by one that uses an index, the contract remains the same, and callers need not change.

16.5. Lists in the Three Systems

In the delivery system, active tasks in creation or dispatch order are a natural list. The order is meaningful — it reflects when tasks were created and may determine which task a robot picks up next — and sequential iteration over the list is a real operation.

In the knowledge engine, search results ranked by relevance score are a natural list. The ranking is the result's most important property for the user: the first result is the most relevant, and the user reads down from there. The list preserves and expresses that ranking directly.

In the virtual world, the order in which entities are updated each tick is a natural list. As established in Chapter 10, deterministic update order is a correctness requirement, not a performance detail: a simulation that processes entities in different orders on different runs is not reproducible. A list makes the update order explicit and stable.

In all three cases, the list earns its place by providing order. In all three cases, the question of whether another structure is also needed — for keyed lookup, for membership testing, for efficient modification — depends on what other operations the system must support. The list is the right answer to the ordering question. It may not be the complete answer to all questions.

16.6. Three Ways List Usage Goes Wrong

Using lists for everything. A list is the simplest data structure to reach for, and simplicity has real value. But a system that uses lists for all its collections, regardless of the operations those collections must support, will eventually produce a profile dominated by linear scans. The remedy is to begin by identifying the dominant operations on each collection, not by choosing a structure. A collection whose dominant operations are order-sensitive deserves a list. One whose dominant operations are key-sensitive deserves something else.

Confusing order with identity lookup. Order and identity are different properties of a collection, and they are often handled by different structures. A system that uses a list to answer the question "what is the status of task T42?" is using an order-preserving structure to answer a question that does not involve order. The list preserves something the operation does not need and cannot efficiently provide what the operation does need. The remedy is to keep the list for operations where order matters and to introduce a separate structure — a map, an index — for operations where identity matters.

Silent duplicate identity. A list that accepts any element at append time, without checking whether an element with the same identifier already exists, will accumulate duplicates if the calling code ever appends the same logical entity twice. Depending on how `find_by_id` is implemented, it will find the first match, the last match, or all matches — and different callers may have different assumptions about which they will receive. The remedy is to define a uniqueness policy for each collection and to enforce it at the point of insertion.

16.7. Quick Exercise

Choose one collection in your system — a collection of tasks, documents, objects, or any other entity type — and answer four questions about it: what are the three most frequent operations performed on it, is each operation primarily order-sensitive or key-sensitive, how often does each operation occur relative to the others, and is a list alone sufficient for the dominant operations?

Then write one contract for the most frequent operation — a precondition that rejects invalid inputs and a postcondition that declares what valid output looks like. If the contract cannot be stated without reference to the implementation, the operation's interface is not yet well-defined.

16.8. Takeaways

- A list is an order-preserving structure. Its strength is sequential access; its weakness is keyed lookup.
- The right data structure for a collection is determined by the operations that collection must support, not by the type of data it contains.
- Order-sensitive operations and key-sensitive operations are different in kind. A single list often cannot satisfy both efficiently at scale.
- The cost of a linear scan is invisible at small scale and dominant at large scale. Recognizing which operations will become bottlenecks before they do is a design skill.
- Contracts on collection operations make later structure changes safe: when the implementation changes, the contract remains, and callers do not need to change.

Chapter 16 introduces sets and maps — the structures that make membership testing and keyed lookup efficient. Where this chapter identified the cost of what lists cannot do, Chapter 16 introduces the structures designed to do exactly those things.

17. Sets and Maps

Chapter 15 arrived at a tension: lists preserve order well but answer keyed questions poorly, and the dominant operations in many real systems are keyed. This chapter introduces the two structures designed to resolve that tension.

A **set** is a collection with no duplicates that answers one question efficiently: is this element present? A **map** is a collection of key-value pairs that answers one question efficiently: given this key, what value is associated with it? Neither preserves insertion order. Neither is organized for sequential traversal. Both are organized for direct access — the operation that lists cannot perform without scanning.

Understanding when to use each, and how to design the keys they depend on, is what separates systems that scale from systems that degrade.

17.1. The Question Each Structure Answers

The choice between a set and a map follows directly from the question the operation asks.

A set answers: *is this element present?* Checking whether a document has already been scored in the current query, whether a location has been visited in the current traversal, whether an entity is already active in the current frame — these are membership questions, and a set answers them in constant time regardless of how many elements it contains. The same question answered by scanning a list takes time proportional to the list's size.

A map answers: *given this key, what is the associated value?* Fetching a delivery task by its identifier, retrieving document metadata by document ID, looking up an entity's current state by its entity ID — these are keyed retrieval questions, and a map answers them in constant time. The same question answered by scanning a list for a matching identifier takes linear time.

If the dominant operation on a collection is a membership question and the structure is a list, the cost is avoidable. If the dominant operation is keyed retrieval and the structure is a list, the cost is equally avoidable. The structures that avoid it exist precisely because these two questions are asked frequently enough in real systems to justify structures organized around answering them efficiently.

17.2. Key Design Is a Correctness Decision

Sets and maps are only as useful as the keys they are built on. A poorly chosen key produces errors that are more insidious than a slow linear scan: the wrong record is updated, a membership check reports false when the element is actually present, a lookup returns a stale result. These failures are not obviously performance problems — they are correctness failures that trace back to a structural decision made before the code was written.

A good key has three properties.

Stability. A key that changes over the lifetime of the entity it identifies cannot be used as a map key. If the key changes, the entity can no longer be found at its old location in the map, and the new location is wherever the changed key happens to point — which may be an entry that belongs to a different entity entirely. Keys must be stable across the full lifetime of the entity they identify. For delivery tasks, the task identifier assigned at creation is stable; the task's current status is not. For documents, the document's unique ID is stable; its title is not.

Uniqueness. A key that does not uniquely identify its entity within the intended scope produces collisions: two different entities map to the same key, and one overwrites the other. Uniqueness is a property of the domain, not of the implementation, and it must be verified rather than assumed. If two delivery tasks can share an identifier — because the identifier is generated from the destination location rather than from a globally unique counter — then the identifier is not a valid key for a map whose entries must be distinct.

Cheapness. A key that is expensive to compute or compare is a key that makes every lookup expensive. A short, fixed-length identifier is cheaper to compare than a long variable-length string; a numeric identifier is cheaper than either. This is a performance consideration rather than a correctness one, but it compounds: in a system that performs millions of lookups per second, the cost of key comparison accumulates.

17.3. Combining Structures for Different Operations

Real systems rarely face a choice between order and keyed access — they need both, for different operations. The correct response is not to choose the structure that satisfies the more important operation and accept degraded performance on the other. It is to use both structures, each for the operation it handles well.

Consider the requirement:

> *"For each query, avoid re-scoring the same document, and fetch metadata by id quickly."*

Two operations are embedded here. The first is a membership question: has this document already been scored in the current query? A set of seen document identifiers answers this in constant time. The second is a keyed retrieval question: given a document identifier, what is the associated metadata? A map from document identifier to metadata record answers this in constant time.

If result order matters for the output — if the ranked list of results must be returned in score order — a third structure is also needed: an ordered sequence that preserves the ranking. The set handles deduplication. The map handles metadata retrieval. The sequence handles ordered output. Each structure is used for exactly the operation it is good at.

This hybrid pattern — an ordered sequence for presentation, a map for identity-based access, optionally a set for membership control — is one of the most common structural patterns in real systems. It appears in the delivery system (a sequence of tasks for dispatch order, a map from task identifier to task for status lookup), in the knowledge engine (a ranked sequence of results for display, a map from document identifier to metadata for enrichment), and in the virtual world (an ordered sequence of entities for deterministic update, a map from entity identifier to state for targeted access).

The hybrid works only if there is one authoritative write path that keeps the structures synchronized. A system that updates the map without updating the sequence, or updates the sequence without updating the map, will produce inconsistencies that are difficult to diagnose because the failure appears at read time, far from where the write error occurred.

17.4. A Map in Code

```
class Doc_Record
create
  make(doc_id: String, title: String) do
    this.doc_id := doc_id
    this.title := title
  end
feature
  doc_id: String
  title: String
invariant
  id_present: doc_id /= ""
  title_present: title /= ""
end

class Doc_Index
create
  make(
    k1: String,
    v1: Doc_Record,
    k2: String,
    v2: Doc_Record,
    k3: String,
    v3: Doc_Record
  ) do
    this.k1 := k1
    this.v1 := v1
    this.k2 := k2
    this.v2 := v2
    this.k3 := k3
    this.v3 := v3
  end
feature
  k1: String
  v1: Doc_Record
  k2: String
  v2: Doc_Record
  k3: String
  v3: Doc_Record

  contains(doc_id: String): Boolean
    require
      id_present: doc_id /= ""
    do
      result := doc_id = k1 or doc_id = k2
                or doc_id = k3
    ensure
      bool_result: result = true or result = false
    end

  fetch_title(doc_id: String): String
    require
      id_present: doc_id /= ""
    do
      if doc_id = k1 then
        result := v1.title
      elseif doc_id = k2 then
        result := v2.title
      elseif doc_id = k3 then
        result := v3.title
      else
        result := "NOT_FOUND"
      end
```

```
    ensure
      non_empty: result /= ""
    end
end
```

`Doc_Index` is a fixed-size map with three entries — minimal, but honest about the structure's properties. `contains` answers a membership question: is this identifier present among the known keys? It returns a boolean, not the associated value. `fetch_title` answers a retrieval question: given this identifier, what title is associated with it? It returns the title if the key is present and `"NOT_FOUND"` if it is not.

The explicit `"NOT_FOUND"` return is not defensive programming — it is part of the contract. A caller of `fetch_title` must handle the case where the key is absent. If the operation returned an empty string for both "this document has an empty title" and "this document does not exist," the caller would have no way to distinguish the two. Declaring a distinct miss behavior as part of the output contract is what makes the operation composable without forcing the caller to inspect the implementation.

The precondition on both operations — `doc_id /= ""` — rejects the empty string as a key. An empty string is not a stable, unique identifier; it is the absence of a value masquerading as one. Rejecting it at the boundary prevents a class of errors that would otherwise produce confusing behavior deep in the lookup logic.

17.5. Sets and Maps in the Three Systems

The pattern across all three systems is consistent: membership control and keyed state access appear together, and each is handled by the structure suited to it.

In the delivery system, a set of locations visited during the current traversal prevents the route-finding algorithm from revisiting nodes it has already explored. A map from task identifier to task provides O(1) access when a robot requests its current assignment or when a status update must be applied to a specific task.

In the knowledge engine, a set of candidate document identifiers eliminates duplicates before scoring begins — a document that is reachable through multiple paths in the document graph should be scored once, not once per path. A map from document identifier to metadata provides the title, timestamp, and other fields needed to enrich the ranked results without re-fetching the full document.

In the virtual world, a set of active entity identifiers tracks which entities are present in the current frame, providing a fast answer to the question of whether a newly spawned entity identifier is already in use.

A map from entity identifier to entity state provides direct access when a collision rule must be applied to two specific entities.

In every case, the set and the map are not replacements for the list that holds the ordered collection — they are complements to it, each handling the operation the list cannot.

17.6. Four Ways Set and Map Usage Goes Wrong

Poor key choice. An unstable key — one derived from a property that changes — produces a map that loses entries as the domain evolves. A non-unique key — one derived from a property that two entities can share — produces a map where one entry silently overwrites another. Neither failure announces itself as a key design error. Both present as mysterious correctness failures that are difficult to reproduce and trace. The remedy is to define the key policy explicitly before building the map, verify that the chosen key is stable and unique within the intended scope, and record that decision in the model.

Assuming order from a map or set. Neither a map nor a set guarantees any particular traversal order. A system that relies on iterating a map and receiving entries in insertion order, or alphabetical order, or any other order, is depending on an implementation detail that is not part of the structure's contract. When the implementation changes — and map implementations vary across languages and platforms — the order changes with it. If ordered output is required, it must be maintained separately, in a structure whose purpose is to preserve order.

Divergence between parallel structures. A system that maintains both a list and a map for the same collection has two representations of the same data. If both are updated atomically through a single write path, they remain consistent. If different parts of the system update one but not the other — if a task is added to the map but not the sequence, or removed from the sequence but not the map — the representations diverge, and the system will produce results that are inconsistent in ways that are hard to diagnose. The remedy is to define a single authoritative write path and to verify consistency between structures in tests that exercise that path.

Treating a missing key as impossible. An operation that assumes a key will always be present, and crashes or returns a meaningless default when it is not, has made an assumption that the domain does not guarantee. Keys are absent for legitimate reasons: a task has been completed and removed, a document has not yet been indexed, an entity has been destroyed. The correct response to a missing key is a declared behavior — an explicit miss result, an error status, an optional return type — that the caller can depend on. A crash is not a declared behavior.

17.7. Quick Exercise

Identify one place in your system where a collection is currently searched by a key-like property — a scan for an element with a particular identifier, a check for whether a particular item is already present. Redesign that operation using map or set semantics by specifying: the chosen key and a justification for its stability and uniqueness, the membership or retrieval operation and its contract, the miss behavior, and one invariant that must hold between this structure and any parallel structure that covers the same data.

Then compare the asymptotic cost of the original scan with the cost of the direct access operation. If the collection is currently small, estimate the input size at which the difference would first become visible in latency measurements.

17.8. Takeaways

- A set answers membership questions in constant time. A map answers keyed retrieval questions in constant time. Each is organized for the operation it performs well and provides no advantage for operations it was not designed for.
- The choice between a list, a set, and a map is determined by the dominant operations on the collection, not by the type of data it contains.
- Key design is a correctness decision. An unstable or non-unique key produces correctness failures, not just performance ones.
- Order and direct access are different properties that often require different structures. The hybrid of an ordered sequence with a map for keyed access is the normal pattern for collections that must support both.
- A missing key is a legitimate outcome. Declaring a distinct miss behavior in the operation's contract is what makes the operation usable without forcing the caller to inspect its implementation.

Chapter 17 introduces trees — structures that organize data hierarchically and make ordered search efficient. Where sets and maps provide direct access by key, trees provide something different: efficient navigation through data *that has natural structure, and efficient answers to range queries that neither lists nor maps can serve well.*

18. Trees — Structured Data

Lists preserve order. Maps and sets provide direct access by key. Both are flat: their elements have no structural relationship to one another beyond position or association. Trees introduce a third kind of organization, one that neither flat structure offers — hierarchy.

A tree is a collection in which elements stand in parent-child relationships, forming a layered structure from a single root down through branches to leaves. This hierarchy is not merely a way of storing data. It encodes meaning about the data — that one thing contains or governs another, that a search can be focused by branching in one direction rather than the other, that the depth at which something lives reflects something real about the domain. When that meaning is genuine, a tree makes it explicit and exploits it. When it is not, a tree is unnecessary complexity layered over a problem that a map would have solved more simply.

18.1. When Hierarchy Is Real

The first question to ask before introducing a tree is whether the data is genuinely hierarchical — whether the parent-child relationship corresponds to something meaningful in the domain rather than being an artifact of the storage choice.

In the knowledge engine, topics and subtopics form a natural hierarchy: a broad topic contains narrower ones, and the narrower ones may contain still more specific ones. A search for notes about a topic can be confined to the relevant subtree rather than scanning the entire collection. The hierarchy is real because it reflects how the domain organizes knowledge.

In the delivery system, dispatch zones may be organized hierarchically: a city contains districts, a district contains blocks, a block contains specific stops. Routing decisions that apply to a district apply to every block within it. The hierarchy is real because it reflects the structure of geographical containment.

In the virtual world, objects exist within regions, regions within zones, zones within the world. Collision detection, rendering, and rule application can all be scoped to a subtree rather than applied globally.

The hierarchy is real because containment is a genuine relationship between objects in the world model.

In each case, the hierarchy is not a storage convenience — it is a fact about the domain that the data structure makes computationally exploitable.

18.2. Tree Invariants

A tree's usefulness depends entirely on its structural invariants. A collection of nodes with parent-child links that does not maintain these invariants is not a tree — it is a graph with aspirations. Once the invariants are violated, the algorithms that rely on them produce incorrect results, and the failures are often silent: a search that should find an element returns `NOT_FOUND` because the path the algorithm would have taken no longer leads where it should.

Three invariants define the basic tree structure. First, there are no cycles: following parent links from any node eventually reaches the root and stops. Second, every non-root node has exactly one parent: a node that has two parents is not a tree node, it is a graph node, and the single-path property that makes tree traversal efficient and predictable no longer holds. Third, for a binary search tree, there is an ordering rule: every key in a node's left subtree is smaller than the node's key, and every key in its right subtree is larger. It is this ordering invariant that makes logarithmic search possible — at each node, the comparison eliminates half the remaining search space.

Maintaining these invariants requires that every insertion and update operation enforces them, not just checks them. An insertion that places a new node without verifying the ordering rule produces a tree that looks correct and searches incorrectly.

18.3. From Requirement to Tree Design

Consider the requirement:

> *"Organize notes by topic hierarchy and find the first matching topic quickly."*

Two things are embedded here. The first is a structural question: what is the shape of the hierarchy? The second is an operational question: how does a search navigate that shape efficiently?

Step 1: Identify the hierarchy. The domain has root topics, subtopics nested within them, and leaf notes attached to the most specific topics. The parent-child relationship is: each subtopic belongs to exactly one parent topic, and each note belongs to exactly one topic. This satisfies the single-parent invariant.

Step 2: Choose a representation. For the search operation, a binary search tree organized by a numeric topic key is a natural teaching representation: each node has a key, a label, and references to a left child (with smaller keys) and a right child (with larger keys).

Step 3: Define the search operation. At each node, compare the target key with the current node's key. If they match, return the label. If the target is smaller, continue left. If larger, continue right. If neither child exists in the direction of travel, return `NOT_FOUND`. Each comparison eliminates one branch of the remaining tree from consideration, producing a search whose cost grows with the depth of the tree rather than its total size.

Step 4: Define miss behavior. A search that reaches a node with no child in the required direction has exhausted the relevant portion of the tree. The result is `NOT_FOUND` — an explicit, declared outcome, not a null or an exception.

Step 5: Preserve invariants on modification. Every insertion must place the new node in the position that preserves the ordering invariant. If the tree is allowed to receive insertions that violate the ordering rule — a new node placed on the wrong side of its parent — subsequent searches will fail to find elements that are structurally present.

18.4. A Tree in Code

```
class Topic_Node
create
  make(
    key: Integer,
    label: String,
    left_key: Integer,
    right_key: Integer
  ) do
    this.key := key
    this.label := label
    this.left_key := left_key
    this.right_key := right_key
  end
feature
  key: Integer
  label: String
  left_key: Integer
  right_key: Integer
invariant
  label_present: label /= ""
  no_self_child: left_key /= key
                 and right_key /= key
end

class Topic_Tree
```

```
create
  make(root: Topic_Node, left: Topic_Node, right: Topic_Node) do
    this.root := root
    this.left := left
    this.right := right
  end
feature
  root: Topic_Node
  left: Topic_Node
  right: Topic_Node

  find_label(target: Integer): String
    do
      if target = root.key then
        result := root.label
      elseif target < root.key
             and target = left.key then
        result := left.label
      elseif target > root.key
             and target = right.key then
        result := right.label
      else
        result := "NOT_FOUND"
      end
    ensure
      non_empty: result /= ""
    end
end
```

`Topic_Node` carries two invariants worth examining. `label_present` ensures that every node in the tree has a non-empty label — a node with no label is a structural placeholder masquerading as a content-bearing element, and the invariant rejects it. `no_self_child` ensures that a node does not list itself as a child — the minimal check against a trivially cyclic structure that would cause any traversal to loop.

`find_label` demonstrates the branching search that trees make possible. The comparison at each level directs the search toward the relevant subtree. In this three-node sketch, the depth is fixed at two and the cost is constant. In a full implementation over a balanced tree with a million nodes, the same logic would require at most twenty comparisons. A linear scan over a million elements would require up to a million. The invariant is what makes that difference real: without the ordering rule, the branching decision at each node is meaningless, and the search degenerates to a scan.

18.5. Balance and Its Consequences

A binary search tree's logarithmic search cost depends on one assumption that the structure itself does not automatically guarantee: that the tree is approximately balanced — that no path from root to leaf is dramatically longer than any other.

A tree built by inserting elements in sorted order — first the smallest key, then the next smallest, and so on — is not balanced. Each new insertion adds a node to the rightmost branch, and the result is a structure that looks like a tree and behaves like a linked list: every search must traverse every node from the root to the target, and the cost is linear in the number of nodes, not logarithmic.

This is the most common tree performance failure, and it occurs silently. The structure is correct — the invariants hold, the searches return the right results — but the performance guarantee that motivated the use of a tree has evaporated. The fix is either to use a self-balancing tree variant — one that restructures itself after insertions to maintain approximate balance — or to monitor the depth distribution of the tree in production and restructure it when the distribution degrades.

18.6. Four Ways Tree Design Goes Wrong

Using a tree without genuine hierarchy. A tree is more complex to build, maintain, and reason about than a map. If the data does not have genuine hierarchical structure — if there is no meaningful parent-child relationship, no search path that exploits the structure — the complexity is not justified. A map with the same keys will answer the same queries more simply. The check is to ask whether the parent-child relationship corresponds to something real in the domain, or whether it is an artifact of the storage decision.

Ignoring balance. An unbalanced tree loses its performance advantage and retains its structural complexity. The worst case is a fully degenerate tree — effectively a linked list — which produces O(N) search on a structure that was introduced for O(log N) search. If the input distribution is not controlled, balance must be actively maintained.

Violating the ordering invariant. A search tree whose ordering invariant has been violated by a bad insertion will fail to find elements that are structurally present. The failure presents as incorrect search results, not as a structural error, and the element that cannot be found may be anywhere in the tree. Preventing this requires enforcing the ordering rule at every insertion and every modification that affects a node's key.

Confusing trees with graphs. A node that acquires two parents, or a sequence of nodes that forms a cycle, is no longer a tree node. The tree algorithms — depth-first search, binary search, hierarchical aggregation — do not handle multiple parents or cycles correctly, and the failures they produce are often difficult to reproduce because they depend on the order in which the problematic structure was created. Enforcing the single-parent and acyclic invariants at insertion time is what prevents the tree from silently becoming a graph.

18.7. Quick Exercise

Choose one place in your system where data has genuine hierarchical structure — a containment relationship, a classification taxonomy, a configuration scope — and model it as a tree with five components: the node identity key and a justification for its stability and uniqueness within the hierarchy, the parent-child rule, one search operation and its contract, one structural invariant beyond what is already implied by the tree shape, and one invalid state the structure must reject.

Then answer this question: why would a map from key to value lose information that the tree preserves? If you cannot give a specific answer — if the map would serve just as well — reconsider whether the hierarchy is genuine.

18.8. Takeaways

- Trees represent genuine hierarchy and make structured search efficient. They are not a substitute for maps when the data is flat.
- The ordering invariant of a binary search tree is what makes logarithmic search possible. Maintaining it at every modification is mandatory, not optional.
- Balance is not guaranteed by tree structure alone. An unbalanced tree provides the complexity of a tree with the performance of a list.
- Tree algorithms depend on three structural invariants: no cycles, single parent for every non-root node, and (for search trees) the ordering rule. Violations produce silent correctness failures.
- Use a tree when the parent-child relationship corresponds to something real in the domain and when search can exploit that structure. When neither is true, a map is simpler and sufficient.

Chapter 18 generalizes from trees to graphs — structures in which the restriction to a single parent is lifted and cycles are permitted. Graphs model the most general form of connected data, and they are the natural representation for routing, dependency, and reachability problems that trees cannot express.

19. Graphs — Networks of Everything

Trees impose a discipline on connected data: one parent per node, no cycles, a single root from which all other nodes descend. This discipline is what makes tree algorithms efficient and predictable. It is also a constraint that many real domains do not satisfy.

A road network has no root and no notion of parent and child — every location is connected to some set of others, and the connections form cycles wherever roads loop back on themselves. A citation network among documents is not a tree: a paper may cite many others, and two papers may cite each other. The interaction structure of a virtual world has entities in multiple overlapping relationships, none of which is naturally hierarchical. When the domain is genuinely networked rather than hierarchical, forcing it into a tree produces a structure that is either incomplete — omitting the edges that violate the tree constraint — or distorted — introducing an artificial root and parent assignments that carry no domain meaning.

A graph is the structure for connected data in its most general form: nodes represent entities, edges represent relationships between them, and neither cycles nor multiple connections between a node and its neighbors are forbidden. Most real systems eventually require graph thinking. The question is whether that thinking happens deliberately, in the model, or accidentally, scattered across code paths that independently reconstruct the connectivity the model never made explicit.

19.1. Four Questions Graphs Answer

Graph-driven systems return repeatedly to four questions. The data structure choices and algorithm choices that serve a system well depend on which of these questions it asks most often and what answers it requires.

Reachability. Can we get from node A to node B at all? This is the foundational question for any routing or connectivity problem. For the delivery system, it determines whether a destination is reachable from the robot's current location. For the knowledge engine, it determines whether one document is connected to another through a chain of citations or links. For the virtual world, it determines whether two entities can interact — whether any path of interactions connects them through the world's rule structure.

Path quality. Among all paths from A to B, which is best by some defined objective? This is a different question from reachability, and it requires a different algorithm. Reachability can be determined by any traversal that visits all reachable nodes; path quality requires a traversal that compares alternatives by a defined criterion — fewest hops, minimum total cost, shortest travel time. Conflating these two questions — treating any found path as an optimal path — produces a system that works and produces wrong answers.

Neighborhood. What is directly connected to this node? For a delivery robot, the neighborhood of a location is the set of locations reachable in one step. For a document, it is the set of documents it directly cites or is cited by. Neighborhood queries are the building blocks of traversal algorithms and are often the most frequent graph operation a system performs.

Connectivity changes. What happens to reachability when an edge is removed? When a path between two locations is blocked, which destinations become unreachable? When a document is removed from the knowledge base, which search paths are disrupted? Systems that must respond to connectivity changes need not just a way to compute reachability, but a way to detect when reachability has changed and respond accordingly.

The data structures chosen to represent the graph, and the algorithms written to traverse it, should follow from which of these questions the system must answer and how frequently.

19.2. Representing a Graph

A graph can be represented in several ways, and the choice matters for performance. The two most common representations make different tradeoffs between the cost of edge lookup and the cost of enumerating a node's neighbors.

An **adjacency matrix** represents the graph as a two-dimensional grid where the entry at row i, column j indicates whether an edge exists from node i to node j. Edge lookup is constant time: check one cell. Enumerating all neighbors of a node requires scanning one row — linear in the total number of nodes, regardless of how many edges that node actually has. An adjacency matrix is a natural choice when the graph is dense (most possible edges exist) and when edge lookup is the dominant operation.

An **adjacency list** represents the graph as a collection where each node maps to the list of nodes it connects to. Enumerating a node's neighbors is linear in the number of edges that node has — which for a sparse graph is much smaller than the total number of nodes. Checking whether a specific edge exists requires scanning the neighbor list, which is linear in degree rather than constant. An adjacency list is a natural choice when the graph is sparse (most possible edges do not exist) and when neighbor enumeration is the dominant operation.

Real delivery networks, citation graphs, and interaction networks are all sparse. The adjacency list is the natural representation for all three. Making that choice explicit in the model — rather than leaving it implicit in however the code happens to store connections — is what makes the representation a design decision rather than an accident.

19.3. From Requirement to Graph Algorithm

Consider the requirement:

> *"Given two locations, return a valid route or report that no route exists."*

Step 1: Define the graph elements. Nodes are location identifiers. Edges are traversable connections between locations. An edge is directed (A connects to B does not imply B connects to A) or undirected depending on whether the connections are one-way or bidirectional. Blocked paths are absent edges — an edge that is no longer traversable is removed from the adjacency structure, not marked with a flag that the algorithm must remember to check.

Step 2: Define the correctness contract. A returned path must use only edges that exist and are traversable in the direction of travel. The first node in the path must be the source. The last must be the destination. Every intermediate step must be connected to the next by a valid edge. When no such path exists, the result is `UNREACHABLE` — an explicit, declared outcome, not an empty list that the caller must interpret.

Step 3: Choose a traversal algorithm. For the first version of the system, with unweighted edges and the objective of fewest hops, breadth-first search is the right choice. BFS explores nodes in order of their distance from the source, measured in hops, and guarantees that the first path it finds to any node is the shortest one. If the objective were minimum total cost on a weighted graph, Dijkstra's algorithm would be the right choice. The algorithm is chosen to match the objective, not selected by habit.

Step 4: Add traversal safety. Graphs may contain cycles. A traversal that does not track which nodes have been visited will follow cycles indefinitely, consuming memory and time without making progress. The visited set, introduced in Chapter 13 for recursive traversals, is equally necessary for iterative ones. Unknown nodes — identifiers not present in the graph — should produce `INVALID_INPUT` rather than being treated as disconnected nodes.

Step 5: Keep the representation explicit. The adjacency structure is a first-class element of the model, not a detail hidden inside the traversal algorithm. Exposing it explicitly makes the graph's content inspectable, testable, and modifiable independently of the algorithms that operate on it.

19.4. A Graph in Code

```
class Route_Graph
feature
  -- Teaching-sized adjacency for four nodes.
  a_to_b: Boolean
  b_to_c: Boolean
  c_to_d: Boolean
  a_to_d: Boolean

  has_edge(from_node, to_node: String): Boolean
    require
      non_empty_nodes: from_node /= ""
                       and to_node /= ""
    do
      if from_node = "A" and to_node = "B" then
        result := a_to_b
      elseif from_node = "B" and to_node = "C" then
        result := b_to_c
      elseif from_node = "C" and to_node = "D" then
        result := c_to_d
      elseif from_node = "A" and to_node = "D" then
        result := a_to_d
      else
        result := false
      end
    ensure
      bool_result: result = true or result = false
    end

  route_a_to_d(): String
    do
      if has_edge("A", "D") then
        result := "A->D"
      elseif has_edge("A", "B") and has_edge("B", "C")
             and has_edge("C", "D") then
        result := "A->B->C->D"
      else
        result := "UNREACHABLE"
      end
```

```
    ensure
      declared_result:
        result = "A->D" or
        result = "A->B->C->D" or
        result = "UNREACHABLE"
    end
end
```

Route_Graph is minimal — four nodes, four possible edges, represented as boolean fields. What it demonstrates is the separation of concerns that a graph model requires. has_edge is the primitive operation: given two node identifiers, does the edge between them exist? It knows nothing about routes. route_a_to_d composes has_edge calls to determine which paths are available and returns the first valid one it finds, or UNREACHABLE if none exists.

The postcondition on route_a_to_d — three explicitly declared possible return values — is the contract that route_a_to_d makes with its callers. Every path it might return is valid according to has_edge: the direct route is guarded by has_edge("A", "D"), the long route by three consecutive edge checks. The algorithm cannot return a route that includes an edge which has_edge would reject. The contract is enforced by the structure of the code rather than by a runtime check that might be forgotten.

In a full implementation, the boolean fields would be replaced by an adjacency list, and route_a_to_d would be replaced by a BFS traversal over that list. The contracts would remain the same. This is the value of specifying the contract at the boundary rather than inside the implementation: when the implementation changes, the contract does not need to change with it.

19.5. Graphs in the Three Systems

The delivery network is a graph in which locations are nodes and traversable paths are edges. When a path is blocked, the corresponding edge is removed, and any route that used it must be recomputed. The graph changes dynamically as the robot moves and paths become available or unavailable, and the route-finding algorithm must operate on the current state of the graph rather than on a snapshot taken at initialization.

The knowledge engine's document collection forms a graph in which documents are nodes and citations or relevance links are edges. A query that asks for documents related to a topic may want not just directly tagged documents but documents reachable within some number of hops through the citation graph. The depth bound determines the breadth of the search; the edge direction determines whether the search follows citations forward, backward, or both.

The virtual world's interaction structure forms a graph in which entities are nodes and potential interactions are edges. Two entities that are in the same region are potentially interacting; the interaction rules determine what happens when they do. The graph changes every tick as entities move, appear, and disappear. Efficiently computing which entity pairs are close enough to interact — the neighborhood query — is one of the dominant computational costs of any physics or interaction simulation.

In all three systems, the graph is the natural representation of the domain's connectivity. In all three, treating it as implicit — as connectivity logic embedded in algorithm code rather than expressed as a data structure — produces systems that are harder to inspect, test, and modify.

19.6. Four Ways Graph Design Goes Wrong

Implicit graph representation. A system whose connectivity is expressed only in algorithm logic — in the conditionals and loops of traversal code, rather than in an explicit adjacency structure — has a hidden graph that cannot be inspected or modified independently of the algorithms over it. Adding a new edge requires finding and modifying traversal code. Removing one requires the same. Testing the graph's structure independently of the traversal is impossible. The remedy is to represent the adjacency structure explicitly and to expose operations for querying and modifying it that the traversal algorithms call.

Missing cycle protection. A traversal that does not track visited nodes will follow cycles without terminating. This is not an edge case — any graph that models a real network may contain cycles, and the absence of a visited set is a bug waiting for a graph that happens to contain one. The visited set is mandatory, not optional, for any traversal over a graph that may have cycles.

Conflating reachability with path quality. BFS finds the path with fewest hops. Dijkstra finds the path with minimum weighted cost. Depth-first search finds a path, but not necessarily the shortest one. These are different algorithms for different objectives, and using one when the problem requires another produces a system that computes the wrong answer correctly. The objective must be defined before the algorithm is chosen, not after.

Weak failure semantics. A traversal that returns an empty path when no route exists, or returns a partial path when traversal is interrupted, leaves the caller to infer what happened from an output that carries no explicit information about it. Two different failure conditions — unreachable destination, invalid input — look identical to a caller who receives an empty list in both cases. The remedy is to declare the full set of possible outcomes in the operation's contract and to return a distinct value for each.

19.7. Quick Exercise

Choose one networked feature in your system — one where entities are connected by relationships that are not strictly hierarchical — and define its graph model with five components: the node type and identity key, the edge type and what it represents in the domain, one reachability query and its contract, one path-quality objective and the algorithm it requires, and one cycle-related risk and how the traversal will handle it.

Then examine how the current implementation represents connectivity. Is there an explicit adjacency structure, or is the connectivity implied by the traversal logic? If it is implied, identify what would need to change to make it explicit.

19.8. Takeaways

- Graphs model real-world networks that trees and lists cannot express without distortion or loss. When the domain is genuinely networked, the graph is the right representation.
- Reachability and path quality are distinct questions that require distinct algorithms. Choosing an algorithm without first defining the objective produces a system that computes the wrong answer.
- Traversal over any graph that may contain cycles requires a visited set. There are no exceptions to this rule.
- The adjacency structure should be explicit and first-class in the model, not implied by traversal code. An implicit graph cannot be tested or modified independently of the algorithms over it.
- Every graph operation needs declared failure semantics. An empty return is not a failure status.

Part IV has now built the complete data structure foundation: lists for ordered sequences, sets and maps for membership and keyed access, trees for hierarchical search, and graphs for general networks. Part V applies these *structures to the core algorithm families — searching, sorting, traversal, and path-finding — that turn organized data into system behavior.*

Studio 3 — The Scaling Crisis

Subtitle

Refactoring Studio 2 models for scale without losing correctness.

Studio Focus This studio chapter is hands-on: implement, verify behavior, and document tradeoffs as you iterate.

The Situation

Studio 2 gave us explicit entities, relationships, and invariants.

Now volume has grown to thousands of tasks, documents, links, and world objects. The model is still correct, but some operations are too slow because access paths are still scan-heavy.

Symptoms observed:

- latency spikes on common lookups
- repeated linear scans in hot paths
- update loops that do unnecessary work

This studio is a direct continuation of Studio 2: same core model classes, new access structures and algorithm choices.

Engineering Brief

Keep the Studio 2 model semantics. Improve runtime behavior.

Required outcomes:

- identify one bottleneck operation per domain
- show a before/after refactor using Studio 2 classes
- measure operation-count improvement
- preserve behavior contracts and invariants

Implementation guidance:

- optimize access paths, not domain meaning
- use explicit failure statuses (`NOT_FOUND`, `NO_MATCH`, etc.)
- keep model classes intact; add indexing/organization layers around them

Implementation In Nex

Use operation counts to compare baseline and refactored paths.

Suggested files:

- `delivery_scale_from_studio2.nex`
- `knowledge_scale_from_studio2.nex`
- `world_scale_from_studio2.nex`
- `studio_3_main.nex`

If using the web IDE, place all classes in one file and run `App.run`.

Reused Studio 2 Models (Unchanged)

```
class Delivery_Task
create
  make(
    task_id: String,
    origin: String,
    destination: String,
    status: String,
    assigned_robot_id: String
  ) do
    this.task_id := task_id
    this.origin := origin
    this.destination := destination
    this.status := status
    this.assigned_robot_id := assigned_robot_id
  end
feature
  task_id: String
  origin: String
  destination: String
  status: String
  assigned_robot_id: String
invariant
  id_present: task_id /= ""
end

class Document
create
  make(doc_id: String, title: String, body: String) do
    this.doc_id := doc_id
    this.title := title
    this.body := body
  end
feature
  doc_id: String
  title: String
  body: String
invariant
  id_present: doc_id /= ""
end

class Doc_Link
create
  make(from_doc_id: String, to_doc_id: String, link_type: String) do
    this.from_doc_id := from_doc_id
    this.to_doc_id := to_doc_id
```

```
    this.link_type := link_type
  end
feature
  from_doc_id: String
  to_doc_id: String
  link_type: String
invariant
  endpoints_present: from_doc_id /= ""
                     and to_doc_id /= ""
end

class World_Object
create
  make(object_id: String, x: Integer, vx: Integer) do
    this.object_id := object_id
    this.x := x
    this.vx := vx
  end
feature
  object_id: String
  x: Integer
  vx: Integer

  shift(delta: Integer) do
    x := x + delta
  end
invariant
  id_present: object_id /= ""
end

class Measure_Result
create
  make(value: String, steps: Integer) do
    this.value := value
    this.steps := steps
  end
feature
  value: String
  steps: Integer
invariant
  value_present: value /= ""
  non_negative_steps: steps >= 0
end
```

Delivery Refactor: Task Lookup

Before (Studio 2-style collection usage): linear scan over `Delivery_Task` objects.

After: index layer for direct key access while keeping `Delivery_Task` as source object.

```
class Delivery_Task_Store_V2
create
  make(
    t1: Delivery_Task,
    t2: Delivery_Task,
    t3: Delivery_Task,
    t4: Delivery_Task,
    t5: Delivery_Task
  ) do
```

```
    this.t1 := t1
    this.t2 := t2
    this.t3 := t3
    this.t4 := t4
    this.t5 := t5
  end
feature
  t1: Delivery_Task
  t2: Delivery_Task
  t3: Delivery_Task
  t4: Delivery_Task
  t5: Delivery_Task

  find_status(task_id: String): Measure_Result
    require
      id_present: task_id /= ""
    do
      let steps: Integer := 1
      let value: String := "NOT_FOUND"
      if t1.task_id = task_id then
        value := t1.status
      elseif t2.task_id = task_id then
        steps := steps + 1
        value := t2.status
      elseif t3.task_id = task_id then
        steps := steps + 1
        value := t3.status
      elseif t4.task_id = task_id then
        steps := steps + 1
        value := t4.status
      elseif t5.task_id = task_id then
        steps := steps + 1
        value := t5.status
      else
        steps := steps + 1
      end

      result := create Measure_Result.make(value, steps)
    ensure
      bounded_steps: result.steps >= 1
                     and result.steps <= 5
    end
end

class Delivery_Task_Store_V3
create
  make(
    k1: String, v1: Delivery_Task,
    k2: String, v2: Delivery_Task,
    k3: String, v3: Delivery_Task,
    k4: String, v4: Delivery_Task,
    k5: String, v5: Delivery_Task
  ) do
    this.k1 := k1
    this.v1 := v1
    this.k2 := k2
    this.v2 := v2
    this.k3 := k3
    this.v3 := v3
    this.k4 := k4
    this.v4 := v4
    this.k5 := k5
    this.v5 := v5
  end
feature
  -- Index from task id to
```

```
  -- task object (teaching-sized map form).
  k1: String
  v1: Delivery_Task
  k2: String
  v2: Delivery_Task
  k3: String
  v3: Delivery_Task
  k4: String
  v4: Delivery_Task
  k5: String
  v5: Delivery_Task

  find_status(task_id: String): Measure_Result
    require
      id_present: task_id /= ""
    do
      let value: String := "NOT_FOUND"
      if task_id = k1 then
        value := v1.status
      elseif task_id = k2 then
        value := v2.status
      elseif task_id = k3 then
        value := v3.status
      elseif task_id = k4 then
        value := v4.status
      elseif task_id = k5 then
        value := v5.status
      end

      result := create Measure_Result.make(value, 1)
    ensure
      constant_step: result.steps = 1
    end
end
```

Knowledge Refactor: Link Validation + Document Lookup

Before: for each `Doc_Link`, document existence check scans document list.

After: use document index by `doc_id` and validate links against indexed lookups.

```
class Knowledge_Model_V2
create
  make(
    d1: Document,
    d2: Document,
    d3: Document,
    l1: Doc_Link,
    l2: Doc_Link
  ) do
    this.d1 := d1
    this.d2 := d2
    this.d3 := d3
    this.l1 := l1
    this.l2 := l2
  end
feature
```

```
  d1: Document
  d2: Document
  d3: Document
  l1: Doc_Link
  l2: Doc_Link

  validate_link(link: Doc_Link): Measure_Result
    do
      let value: String := "VALID"
      if (d1.doc_id = link.from_doc_id
          or d2.doc_id = link.from_doc_id
          or d3.doc_id = link.from_doc_id) and
         (d1.doc_id = link.to_doc_id
          or d2.doc_id = link.to_doc_id
          or d3.doc_id = link.to_doc_id) then
        value := "VALID"
      elseif not (d1.doc_id = link.from_doc_id
                  or d2.doc_id = link.from_doc_id
                  or d3.doc_id = link.from_doc_id) then
        value := "MISSING_FROM"
      else
        value := "MISSING_TO"
      end

      result := create Measure_Result.make(value, 2)
    ensure
      status_known:
        result.value = "VALID" or
        result.value = "MISSING_FROM" or
        result.value = "MISSING_TO"
    end
end

class Knowledge_Model_V3
create
  make(
    dk1: String, dv1: Document,
    dk2: String, dv2: Document,
    dk3: String, dv3: Document
  ) do
    this.dk1 := dk1
    this.dv1 := dv1
    this.dk2 := dk2
    this.dv2 := dv2
    this.dk3 := dk3
    this.dv3 := dv3
  end
feature
  dk1: String
  dv1: Document
  dk2: String
  dv2: Document
  dk3: String
  dv3: Document

  has_doc(doc_id: String): Boolean
    do
      result := doc_id = dk1
                or doc_id = dk2
                or doc_id = dk3
    ensure
      bool_result: result = true
                   or result = false
    end

  validate_link(link: Doc_Link): Measure_Result
```

```
    do
      let value: String := "VALID"
      if not has_doc(link.from_doc_id) then
        value := "MISSING_FROM"
      elseif not has_doc(link.to_doc_id) then
        value := "MISSING_TO"
      end

      result := create Measure_Result.make(value, 2)
    ensure
      status_known:
        result.value = "VALID" or
        result.value = "MISSING_FROM" or
        result.value = "MISSING_TO"
    end
end
```

Virtual World Refactor: Targeted Update

Before: scan all `World_Object` entries to find target.

After: id index for direct target selection, then apply same position update semantics.

```
class World_Model_V2
create
  make(
    w1: World_Object,
    w2: World_Object,
    w3: World_Object,
    w4: World_Object
  ) do
    this.w1 := w1
    this.w2 := w2
    this.w3 := w3
    this.w4 := w4
  end
feature
  w1: World_Object
  w2: World_Object
  w3: World_Object
  w4: World_Object

  move_by_id(object_id: String,
             delta: Integer): Measure_Result
    require
      id_present: object_id /= ""
    do
      let value: String := "NOT_FOUND"
      let steps: Integer := 1
      if w1.object_id = object_id then
        w1.shift(delta)
        value := "UPDATED"
      elseif w2.object_id = object_id then
        steps := steps + 1
        w2.shift(delta)
        value := "UPDATED"
      elseif w3.object_id = object_id then
        steps := steps + 1
        w3.shift(delta)
        value := "UPDATED"
```

```
      elseif w4.object_id = object_id then
        steps := steps + 1
        w4.shift(delta)
        value := "UPDATED"
      else
        steps := steps + 1
      end

      result := create Measure_Result.make(value, steps)
    ensure
      bounded_steps: result.steps >= 1
                     and result.steps <= 4
    end
end

class World_Model_V3
create
  make(wk1: String, wv1: World_Object) do
    this.wk1 := wk1
    this.wv1 := wv1
  end
feature
  wk1: String
  wv1: World_Object

  move_by_id(object_id: String,
             delta: Integer): Measure_Result
    require
      id_present: object_id /= ""
    do
      let value: String := "NOT_FOUND"
      if object_id = wk1 then
        wv1.shift(delta)
        value := "UPDATED"
      end

      result := create Measure_Result.make(value, 1)
    ensure
      constant_step: result.steps = 1
    end
end
```

Studio Driver (Before/After on Studio 2 Models)

```
class App
feature
  run() do
    -- Delivery tasks (Studio 2 model objects)
    let t1: Delivery_Task
      := create Delivery_Task.make(
           "T-1", "Hub-A", "Zone-1", "PENDING", "R-1"
         )
    let t2: Delivery_Task
      := create Delivery_Task.make(
           "T-2", "Hub-A", "Zone-2", "IN_TRANSIT", "R-2"
         )
    let t3: Delivery_Task
      := create Delivery_Task.make(
           "T-3", "Hub-B", "Zone-3", "FAILED", "R-3"
         )
    let t4: Delivery_Task
      := create Delivery_Task.make(
```

```
        "T-4", "Hub-B", "Zone-4", "DELIVERED", "R-4"
      )
let t5: Delivery_Task
  := create Delivery_Task.make(
        "T-5", "Hub-C", "Zone-5", "PENDING", "R-5"
      )

let d_v2: Delivery_Task_Store_V2
 := create Delivery_Task_Store_V2.make(t1, t2, t3, t4, t5)

let d_v3: Delivery_Task_Store_V3
 := create Delivery_Task_Store_V3.make(
      t1.task_id, t1,
      t2.task_id, t2,
      t3.task_id, t3,
      t4.task_id, t4,
      t5.task_id, t5
    )

let d2r: Measure_Result
 := d_v2.find_status("T-5")
let d3r: Measure_Result
 := d_v3.find_status("T-5")
print("Delivery V2 steps: " + d2r.steps)
print("Delivery V3 steps: " + d3r.steps)

-- Knowledge documents + links
-- (Studio 2 model objects)
let doc1: Document
  := create Document.make("D-1", "Graphs", "Graph notes")
let doc2: Document
  := create Document.make("D-2", "Trees", "Tree notes")
let doc3: Document
  := create Document.make("D-3", "Maps", "Map notes")

let link: Doc_Link
  := create Doc_Link.make("D-1", "D-3", "references")

let k_v2: Knowledge_Model_V2
 := create Knowledge_Model_V2.make(doc1, doc2, doc3, link, link)

let k_v3: Knowledge_Model_V3
 := create Knowledge_Model_V3.make(
      doc1.doc_id, doc1,
      doc2.doc_id, doc2,
      doc3.doc_id, doc3
    )

let k2r: Measure_Result := k_v2.validate_link(link)
let k3r: Measure_Result := k_v3.validate_link(link)
print("Knowledge V2 status/steps: " + k2r.value
      + "/" + k2r.steps)
print("Knowledge V3 status/steps: " + k3r.value
      + "/" + k3r.steps)

-- World objects (Studio 2 model objects)
let o1: World_Object := create World_Object.make("E-1", 1, 0)
let o2: World_Object := create World_Object.make("E-2", 2, 0)
let o3: World_Object := create World_Object.make("E-3", 3, 0)
let o4: World_Object := create World_Object.make("E-4", 4, 0)

let w_v2: World_Model_V2
  := create World_Model_V2.make(o1, o2, o3, o4)

let w_v3: World_Model_V3
  := create World_Model_V3.make(o4.object_id, o4)
```

```
    let w2r: Measure_Result := w_v2.move_by_id("E-4", 3)
    let w3r: Measure_Result := w_v3.move_by_id("E-4", 3)
    print("World V2 steps: " + w2r.steps)
    print("World V3 steps: " + w3r.steps)
  end
end
```

Expected pattern:

- V2 keeps Studio 2 model semantics but uses scan-heavy access
- V3 keeps the same model objects and adds index layers for critical operations
- behavior stays consistent; operation counts improve

Studio Challenges

Level 1 — One Refactor

- choose one Studio 2 operation with repeated scans
- implement V2 baseline and V3 indexed refactor
- compare step counts on hit and miss cases

Level 2 — Three-Domain Refactor

- implement before/after for delivery, knowledge, and world
- keep model classes unchanged
- document where indexes are derived from model identity

Level 3 — Competing Strategies

- for one domain, compare two refactors:
  - index-heavy approach
  - batch/partition approach
- state when each approach wins

Postmortem

Discuss with evidence:

- Which Studio 2 operation became the worst bottleneck and why?
- Which refactor changed asymptotic behavior versus constants only?

- Which invariant or contract protected correctness during optimization?
- Which optimization should be postponed until higher load?

Deliverables

- Nex code showing V2 (baseline) and V3 (refactor) on Studio 2 model classes
- output logs with step-count comparisons
- one-page design note:
  - bottleneck operation
  - chosen refactor
  - correctness safeguards
  - limitations and next trigger point

Exit Criteria

You are ready for Part V if:

- you can optimize access paths without changing domain meaning
- you can preserve model contracts while changing data organization
- you can justify changes with measured evidence, not intuition
- you can explain when the next redesign will be necessary

Part V.

Part V — Algorithms That Power Systems

20. Searching for What Matters

Part IV organized data into structures suited to different operations. Part V asks what algorithms bring those structures to life. We begin with search, because nearly every system spends significant time answering a single question in one form or another: where is the thing we need?

That question sounds simple. The answer is not. The right search strategy depends on how the data is organized, how frequently the search occurs, and what the system must do when the thing being sought is not there. A search algorithm chosen by habit rather than by these factors will work correctly for small inputs and become a performance problem as the system grows. More subtly, a search algorithm whose failure behavior is undefined will work correctly when things are found and produce confusion when they are not.

20.1. Four Search Strategies

Different data organizations admit different search strategies, and the choice between them is determined by the properties of the data and the cost the system can afford.

Linear search examines elements one at a time from start to finish until a match is found or the collection is exhausted. It requires no assumptions about the organization of the data — the elements may be in any order, and the search will find a match if one exists. Its cost grows linearly with the size of the collection: a collection twice as large requires twice as many comparisons in the worst case. Linear search is appropriate when the collection is small, when it is unsorted and sorting it would cost more than the search saves, or when search is rare relative to modification — maintaining a sorted order for a collection that is searched infrequently and modified constantly is work that does not pay off.

Binary search repeatedly halves the remaining search space by comparing the target against the middle element of the current range. If the target is smaller, the search continues in the lower half; if larger, in the upper half; if equal, the search terminates with a match. Each comparison eliminates half of the remaining candidates, producing a cost that grows logarithmically: a collection of a million sorted elements requires at most twenty comparisons. Binary search requires that the collection be sorted — this is a precondition, not a performance hint. Applied to an unsorted collection, binary search produces incorrect results, not slow ones. The precondition must be enforced, not assumed.

Map and set lookup provides constant-time access when the data has been organized around a key. Given a task identifier, a map from identifier to task returns the result in constant time regardless of the total number of tasks. This is not faster binary search — it is a fundamentally different operation that does not depend on sorting or comparison at all. It requires that the data have been organized into the map structure in advance, and it requires a stable, unique key. When these conditions hold, keyed lookup is the preferred strategy for any collection where the dominant operation is identity-based retrieval.

Graph and tree traversal finds elements by navigating the connections between them. A depth-first or breadth-first traversal discovers nodes reachable from a starting point; a binary search tree traversal finds an element whose key satisfies an ordering criterion; a shortest-path algorithm finds the element (or path) that optimizes some quality measure. These are the right strategies when the data has genuine structure — hierarchy, network connectivity — and when the answer to the search question depends on that structure rather than on a single key comparison.

20.2. Correctness Before Speed

Search strategy discussions tend to focus on performance — linear versus logarithmic versus constant. Performance matters, but it is secondary to a prior requirement: the search algorithm must define what it promises and what it does when the promise cannot be kept.

A search operation has a correctness contract with two parts. The match semantics specify what counts as a successful result: exact identifier equality, a value within a range, a document that satisfies a relevance criterion. Without explicit match semantics, different implementations will answer the same question differently, and the difference will not be visible until a caller is surprised by a result it did not expect.

The miss semantics specify what the algorithm returns when no match exists. This is not an edge case — it is half the search space. A system that searches for a task by identifier will sometimes find it and sometimes not. Both outcomes must be declared. An explicit miss result

— `NOT_FOUND`, an option type with a defined absent case, a status code — tells the caller unambiguously what happened. An empty return, a null, or a zero tells the caller that something was returned but provides no information about whether it means "not found," "found but empty," or "error."

The input assumptions specify what the algorithm requires to operate correctly. Binary search requires a sorted collection. Keyed lookup requires a valid, non-empty key. Graph traversal with cycle detection requires a visited set. These are preconditions, and they belong in the algorithm's contract. A caller that violates a precondition is not guaranteed any particular behavior; a caller that satisfies it is guaranteed the declared output. Writing the preconditions down — in the code, not just in documentation — is what makes the contract enforceable.

20.3. From Requirement to Search Design

Consider the requirement:

> *"Given a task identifier, return the current status quickly and safely."*

The word "safely" is doing more work than it appears. It implies that missing tasks must be handled gracefully, not that the algorithm should avoid crashing on valid inputs.

Step 1: Define match semantics. A match is exact identifier equality. A task whose identifier is equal to the query identifier is the result; any other task is not. There is no partial match, no prefix search, no relevance ranking. The match criterion is simple and must be stated explicitly because it determines which search strategy is appropriate.

Step 2: Define miss behavior. If no task with the given identifier exists in the collection, the result is `NOT_FOUND` — a declared status value, not null, not an empty status string, not an exception. The caller of this operation must handle both the found case and the not-found case; the contract makes both cases explicit.

Step 3: Assess the data's organization. If tasks are stored in an unordered sequence, linear search is the only option without additional structure. If tasks are stored in a map from identifier to task, keyed lookup is available. If the collection is small and search is infrequent, linear search on the sequence may be acceptable even as a long-term design. If the collection is large or growing, or if identifier lookup is a hot path, the map is the right structure and the question of search strategy is already answered by the structure choice.

Step 4: Choose the initial implementation. For an early version of the system with a small number of tasks, linear search over an ordered sequence is a defensible starting point. It is simple, correct, and easy to test. The cost at this scale is negligible.

Step 5: Define the upgrade trigger. The linear search should be replaced when identifier lookup appears as a significant contributor to latency on the hot path. This trigger should be stated explicitly — a collection size, a latency measurement, a throughput target — so that the replacement is a scheduled decision rather than an emergency response to a performance incident.

20.4. A Search Operation in Code

```
class Search_Result
create
  make(status: String, steps: Integer) do
    this.status := status
    this.steps := steps
  end
feature
  status: String
  steps: Integer
invariant
  status_present: status /= ""
  steps_non_negative: steps >= 0
end

class Task_Search
feature
  id1: String
  st1: String
  id2: String
  st2: String
  id3: String
  st3: String
  id4: String
  st4: String

  linear_find(task_id: String): Search_Result
    require
      id_present: task_id /= ""
    do
      let final_status: String := "NOT_FOUND"
      let steps: Integer := 1
      if task_id = id1 then
        final_status := st1
      elseif task_id = id2 then
        steps := steps + 1
        final_status := st2
      elseif task_id = id3 then
        steps := steps + 1
        final_status := st3
      elseif task_id = id4 then
        steps := steps + 1
        final_status := st4
      else
        steps := steps + 1
```

```
      end
      result := create Search_Result.make(
        final_status,
        steps
      )
    ensure
      bounded_steps: result.steps >= 1
                     and result.steps <= 4
    end
end
```

`Search_Result` carries two fields and two invariants. The `status` field holds either the found task's status or `NOT_FOUND`. The `steps` field counts how many comparisons the search required. The invariants enforce that both fields are always in a valid state: `status` is never empty, and `steps` is never negative.

`linear_find` initializes the result to `NOT_FOUND` before beginning the search. This initialization is not an optimization — it is the guarantee that the miss case is handled correctly regardless of which branch the control flow takes. If none of the four comparisons produce a match, the result is already correct without any additional code in the else branch. The postcondition confirms that the number of steps is bounded between one and four — a measurable statement about the algorithm's behavior on this particular collection size that a test can verify directly.

The `steps` field deserves particular attention. A search algorithm that returns only a result status produces a black box: the caller knows what was found, but has no information about how hard the search was. An algorithm that also returns a step count makes its cost observable and testable. For the worked design path above, this observability is what makes it possible to detect when linear search is becoming a bottleneck: the steps count grows with the collection size, and when it grows large enough to affect latency, the upgrade trigger fires.

20.5. Search in the Three Systems

In the delivery system, searching for a task by identifier is a frequent operation: robots request their assignments, status updates target specific tasks, and dispatch logic queries task states. The collection of active tasks grows as the system scales. Linear search over a sequence of tasks will degrade at scale; a map from task identifier to task, introduced in Chapter 16, is the appropriate structure when identifier lookup is the dominant operation.

In the knowledge engine, search operates at two levels. The first level locates candidate documents by token, tag, or identifier — a structured search over index entries. The second level ranks the candidates by relevance — a scoring and comparison over a much smaller set. These are distinct operations with distinct performance requirements, and they warrant distinct strategies. The first benefits from indexed structure; the second operates on a small enough candidate set that a linear scan over ranked scores is appropriate.

In the virtual world, locating entity state by identifier is a per-tick operation: every frame may require fetching the current state of specific entities to apply interaction rules or check collision. The frequency of this operation — once per entity per tick, potentially thousands of times per frame — makes it the most demanding search workload of the three systems. Keyed lookup in a map organized by entity identifier is the only strategy that scales.

In all three systems, the search operation that looks trivial at small scale is the one most likely to become the first bottleneck at large scale.

20.6. Three Ways Search Design Goes Wrong

One strategy everywhere. A codebase that uses linear search for all collection queries has made a default choice that is correct at small scale and incorrect at large scale. The problem is not linear search itself — it is the absence of a decision about strategy. When a team has never asked which operations are frequent, which collections are large, and which access patterns are key-based, they cannot know which searches will degrade as the system grows. The remedy is to classify the operation profile of each collection before choosing a strategy.

Undefined miss semantics. A search that returns an empty string for both "task found with empty status" and "task not found" forces every caller to add defensive logic that guesses which situation applies — or to skip that logic, accept the ambiguity, and produce incorrect behavior. The ambiguity compounds across callers: different parts of the system will interpret the same return value differently, and the inconsistency will not be visible until a real miss produces an unexpected result in a caller that assumed all returns were valid. The remedy is to declare distinct return values for all outcomes before writing the implementation.

Violated preconditions. Binary search applied to an unsorted collection does not return slow results — it returns wrong ones. A keyed lookup called with an empty string does not degrade gracefully — it returns a result that was never intended. Preconditions are not optional documentation. They are the boundary conditions under which the algorithm's guarantees hold, and violating them voids the guarantee. Enforcing preconditions in the code — with require clauses, assertions, or type constraints — is what makes the contract reliable rather than advisory.

20.7. Quick Exercise

Choose one search operation in your system and document it completely with five components: the match rule that defines what counts as a successful result, the miss rule that defines the explicit return value when no match exists, the current strategy and its asymptotic cost, the expected input size range in production, and the trigger condition at which the current strategy should be reconsidered.

Then write the precondition and postcondition for that operation. If the postcondition cannot distinguish the found case from the not-found case without reading the implementation, the miss semantics are not yet explicit.

20.8. Takeaways

- Search strategy is a design decision determined by data organization and operation frequency. Choosing by habit produces systems that work at small scale and degrade at large scale.
- Match semantics, miss semantics, and input preconditions are part of every search algorithm's contract. An algorithm without explicit miss semantics is not complete.
- Linear search, binary search, keyed lookup, and structural traversal are not interchangeable techniques for the same problem. Each is the right choice for a specific combination of data organization and operation profile.
- The search operation that looks trivial at small scale is often the first bottleneck at large scale. Define the upgrade trigger before the bottleneck arrives.

Chapter 20 examines sorting — the operation that turns unordered data into a form where faster search, efficient merge, and reliable comparison become possible. Understanding sorting as a prerequisite for other algorithms, rather than as a standalone operation, is what connects it to the system design concerns of this part.

21. Sorting the World

Chapter 19 showed that search strategy depends on how data is organized. Sorting is often what creates that organization. A sorted collection is not just a collection in a particular display order — it is a collection with a structure that algorithms can exploit. Binary search requires sorted input. Efficient merging of two result sets requires sorted input. Detecting duplicates in a large collection becomes a linear scan through sorted data rather than a quadratic comparison of every pair. In each case, sorting is not the end of the computation but the preparation that makes efficient computation possible downstream.

The distinction matters because it changes how we think about sorting's cost. The upfront cost of sorting a collection is not paid for the sort itself — it is paid for every subsequent operation that benefits from the ordered structure. When that structure is used many times, the upfront cost is amortized across all those uses. When it is used once or not at all, the sort was unnecessary work.

21.1. What Sorting Guarantees

A sort operation, like any algorithm, is defined by its contract rather than by its implementation. The contract for a sorting operation has three parts.

The comparison rule. How are elements ordered relative to each other? For a collection of delivery tasks, the ordering might be by priority score — lower scores first, or higher scores first — or by creation time, or by some combination. The comparison rule must be total: for any two elements, it must be possible to determine which comes first, or that they are equivalent. A comparison rule that is undefined for some pairs of elements produces an algorithm that is correct on some inputs and undefined on others.

The tie-break rule. What happens when two elements are equivalent under the comparison rule? A sort that assigns the same priority to two tasks must decide which comes first in the output. One answer is that it does not matter — either order is acceptable, and the algorithm may choose either. Another answer is that the original relative order must be preserved — elements that were equal under the comparison rule must appear in the output in the same order they appeared in the input. This second property is called stability, and whether it is required depends on the semantics of the output, not on the implementation's convenience.

The output invariant. The output of a sort is a permutation of the input in which no element appears out of order under the comparison rule. This sounds obvious, but stating it explicitly as an invariant has practical value: it is a property that tests can verify mechanically, and it is the property that callers depend on when they use the sorted output to perform binary search, merge, or any other operation that requires sorted input.

21.2. Stability as a Correctness Requirement

Stability is the property most often treated as optional and most often quietly required.

Consider the delivery task queue. Tasks are sorted by priority so that the robot dispatches the highest-priority tasks first. Two tasks with equal priority arrive in a defined sequence. The question of which one the robot dispatches first is not just a presentation preference — if the system has a fairness guarantee that equal-priority tasks are served in arrival order, stability is a correctness requirement. An unstable sort that reorders equal-priority tasks violates the fairness guarantee every time it runs.

The same situation arises in the knowledge engine. Search results with equal relevance scores may have a defined secondary ordering — by publication date, by author, by document identifier — that preserves a meaningful relationship the user depends on. An unstable sort that randomizes the order of equally-scored results is not producing equivalent output. It is producing different output on every run, and a system that produces different output from the same input for no declared reason is a system whose behavior cannot be understood or tested reliably.

The principle is this: when the order of equivalent elements matters to any caller, stability is part of the sort's contract. When no caller cares about the order of equivalent elements, stability is not required. Both conclusions must come from examining what callers actually need, not from an assumption that equal elements are interchangeable.

21.3. Sort Strategy and Its Constraints

Sorting algorithms make different tradeoffs between time cost, memory cost, and stability, and the right choice depends on the constraints of the specific sorting problem.

A **comparison-based sort** derives element order entirely from pairwise comparisons. The theoretical minimum cost for any comparison-based sort is O(N log N) comparisons in the worst case — a result that follows from an information-theoretic argument about how many comparisons are needed to distinguish among all possible orderings of N elements. Any algorithm that does fewer comparisons in the worst case cannot be a correct general-purpose comparison sort. This lower bound is achieved in practice by algorithms like merge sort and heapsort.

Simple pass-based sorts — bubble sort, insertion sort, selection sort — are $O(N^2)$ in the worst case and are appropriate only when the collection is small or nearly sorted. Insertion sort in particular performs well on collections that are already almost in order, making it a useful component of hybrid strategies that use it for small subproblems. As a general-purpose sort for large collections, these algorithms are inappropriate — not because they are incorrect, but because their cost at scale is avoidable.

Stability eliminates some otherwise-correct algorithms from consideration. Heapsort is not stable. Merge sort is stable. Quicksort as typically implemented is not stable. When stability is required and memory is available, merge sort is the natural choice. When stability is required and memory is constrained, achieving it requires additional work. The constraint and its implications should be visible in the design, not buried in the implementation.

The practical advice for most systems is to use the standard library's sort — which is typically an optimized hybrid that handles the common cases well — unless domain constraints require custom behavior. The cases that require custom sorting are those where the comparison rule is non-standard, where stability matters in a way the library does not guarantee, or where the data has known structure (such as near-sortedness) that a specialized algorithm can exploit.

21.4. From Requirement to Sort Design

Consider the requirement:

> *"Return delivery tasks ordered by priority, preserving insertion order for ties."*

Step 1: Define the comparison key. The primary ordering criterion is priority score. What "ordered by priority" means — whether a high priority score means a task should come first or last — must be stated explicitly. For a delivery system where lower numbers represent higher urgency, "ordered by priority" means ascending order.

Step 2: Define tie-break behavior. Equal-priority tasks must preserve insertion order. This is a stability requirement: the sort must be stable so that tasks inserted earlier appear before tasks inserted later when their priority scores are equal. An unstable sort would satisfy the primary ordering criterion and violate the tie-break requirement.

Step 3: Choose a baseline sort. For an early implementation with a small number of tasks, a simple stable sort is appropriate. Simple does not mean incorrect — it means the implementation prioritizes clarity over optimization at a scale where the performance difference does not yet matter.

Step 4: State the correctness checks. The output is non-decreasing by priority score: no task in the output appears after any task with a strictly lower priority score. Equal-priority tasks appear in the output in the same relative order as they appeared in the input. Both of these are mechanical properties that a test can verify on any input.

Step 5: Define the scale transition. When the task collection grows large enough that the sort's cost appears in latency measurements, the baseline sort should be replaced by an optimized stable sort — most likely the standard library's stable sort. The output contract does not change; only the implementation does.

21.5. A Sort Operation in Code

```
class Task_View
create
  make(id: String, priority: Integer, arrival: Integer) do
    this.id := id
    this.priority := priority
    this.arrival := arrival
  end
feature
  id: String
  priority: Integer
  arrival: Integer
invariant
  id_present: id /= ""
  arrival_non_negative: arrival >= 0
end

class Sort_Example
create
  make(t1: Task_View, t2: Task_View, t3: Task_View) do
    this.t1 := t1
    this.t2 := t2
```

```
    this.t3 := t3
  end
feature
  t1: Task_View
  t2: Task_View
  t3: Task_View

  sorted_ids_by_priority(): String
    do
      -- Teaching-sized fixed comparison network
      -- for three items.
      let a: Task_View := t1
      let b: Task_View := t2
      let c: Task_View := t3

      if a.priority > b.priority then
        let tmp: Task_View := a
        a := b
        b := tmp
      end

      if b.priority > c.priority then
        let tmp2: Task_View := b
        b := c
        c := tmp2
      end

      if a.priority > b.priority then
        let tmp3: Task_View := a
        a := b
        b := tmp3
      end

      result := a.id + " -> " + b.id + " -> " + c.id
    ensure
      non_empty: result /= ""
    end
end
```

`Task_View` carries an `arrival` field alongside the priority score. In the sketch, `arrival` is present in the invariant but not yet used in the sorting logic — a deliberate choice that makes the gap between the current implementation and the full requirement visible. The sort orders tasks by priority but does not yet break ties by arrival order. Adding stability means modifying the comparisons to include `arrival` as a secondary key when `priority` values are equal. The postcondition `non_empty` is weak — a stronger postcondition would assert the ordering invariant directly. For a production implementation, that stronger assertion is worth writing and testing explicitly.

The three-swap comparison network is a sorting network for exactly three elements: it correctly sorts any combination of three values using exactly three comparisons. It is not a general sorting algorithm — it does not scale beyond three elements — but for a teaching sketch of fixed size it has the virtue of making every comparison explicit. A reader can trace the execution for any input and verify that the output is correctly ordered. That traceability is what makes it useful here.

21.6. Sorting in the Three Systems

In the delivery system, the task queue must be sorted by priority before dispatch. The stability requirement comes from the fairness guarantee: equal-priority tasks should be dispatched in the order they arrived, so that no task waits indefinitely while equal-priority tasks that arrived later are repeatedly dispatched ahead of it. This is a correctness requirement on the sort, not a presentation preference.

In the knowledge engine, search results must be sorted by relevance score before being returned to the caller. Results with equal scores may need a secondary ordering — by recency, by document identifier, by some other stable criterion — so that the same query reliably produces the same output. An output that varies non-deterministically across runs on the same input cannot be tested or debugged. Stability with a deterministic secondary key is what makes the output reproducible.

In the virtual world, entities must be processed in a deterministic order each tick. The sort that establishes this order must be stable across runs: given the same set of entities with the same states, the processing order must be the same. If the sort is not stable, the simulation is not reproducible — two runs of the same scenario may produce different outcomes, which makes the system impossible to test and impossible to debug.

In all three systems, sorting sits between raw data and high-value operations. The sort's cost is paid once; the benefits of structured output are collected by every operation that follows.

21.7. Three Ways Sorting Goes Wrong

An undefined comparison rule. A sort that relies on implicit or inconsistent comparisons produces output that varies across runs or implementations. The same collection sorted twice may produce different results if the comparison rule is not total, if it depends on mutable state, or if it is implemented differently in two places. The comparison rule must be defined explicitly — as a function with a documented contract — and it must be the same everywhere the sort is applied.

Unexamined stability requirements. A sort that is not stable will reorder equal elements arbitrarily. If any caller depends on the order of equal elements — for fairness, for reproducibility, for secondary sorting — the instability is a correctness failure. Stability requirements are easy to overlook because they are invisible when all elements are distinct. They become visible only when two elements are equal, which may not happen in early testing. Asking the stability question explicitly — for each sorted collection, does the order of equal elements matter to any caller? — is the discipline that prevents the failure.

Re-sorting too frequently. A full sort of a large collection is an O(N log N) operation. Performing it in a hot loop — once per incoming event, once per request, once per tick — when the collection changes only occasionally is work that compounds quickly. The alternatives are to maintain a sorted order incrementally, inserting new elements in sorted position rather than re-sorting the full collection; to batch updates and sort once per batch; or to use a data structure that maintains order intrinsically. The choice depends on the relative frequency of insertions and reads.

21.8. Quick Exercise

Choose one ordered output in your system and define its sort contract completely: the primary comparison key and the direction of ordering, the tie-break rule and whether it constitutes a stability requirement, the output invariant that a test could verify mechanically, and the current sort frequency relative to how often the underlying collection changes.

Then write one test case using two elements with equal primary keys that verifies the tie-break behavior. If the test cannot be written without knowing the implementation, the tie-break rule is not yet explicit enough to be part of the contract.

21.9. Takeaways

- Sorting creates structure that downstream operations exploit. Its cost is an investment in the efficiency of everything that follows.
- A sort's contract has three parts: the comparison rule, the tie-break rule, and the output invariant. All three must be explicit.
- Stability is a correctness requirement when the order of equal elements matters to any caller. It is not an implementation detail.
- The right sort strategy is determined by collection size, stability requirements, memory constraints, and whether the input has known structure. Defaulting to the standard library's sort is appropriate for most cases.
- Re-sorting a large collection more often than it changes is avoidable cost. Define the sort frequency alongside the sort strategy.

Chapter 21 turns from sorting data to traversing structures — the algorithms that systematically visit every node in a tree or graph. Traversal is the basis for search, analysis, and transformation of structured data, and the order in which nodes are visited determines what the traversal can compute.

22. Exploring Trees and Graphs

Sorting organizes data for efficient access. Traversal navigates structure to answer questions about it. Once data is represented as a tree or graph, nearly every meaningful computation over that data — finding connected components, determining reachability, discovering paths, computing aggregates over hierarchies — is some form of traversal: a systematic visit to the nodes of the structure according to a defined order.

Two fundamental orders exist, and they are not interchangeable. The right choice between them depends on what the traversal is trying to compute. A team that uses depth-first search where breadth-first is required will produce correct code that answers the wrong question.

22.1. Depth-First Search

Depth-first search commits fully to one direction before exploring any other. Starting from a node, it follows one outgoing edge to a neighbor, then one outgoing edge from that neighbor, then one from that neighbor's neighbor, continuing in this way until it reaches a node with no unvisited neighbors. Only then does it backtrack and try a different direction.

The consequence of this commitment is that DFS reaches distant nodes quickly but makes no guarantee about the distance at which it finds them. A DFS from node A may reach node Z before it has visited all of A's immediate neighbors. If the goal is to find *any* path to Z, DFS will find one efficiently. If the goal is to find the *shortest* path to Z, DFS provides no such guarantee — it will find a path, but not necessarily the shortest one.

DFS is naturally expressed as a recursive algorithm. The recursive structure mirrors the traversal structure: visiting a node means visiting the node itself and then recursively visiting each of its unvisited neighbors. This correspondence between the algorithm's structure and the data's structure is why DFS is the natural choice for operations over trees and hierarchies — structures that are themselves recursively defined. Computing the depth of a tree, evaluating an expression tree, finding all nodes in a subtree that satisfy some property: these are DFS computations, and they read clearly as recursive code.

DFS is also the natural choice for exhaustive exploration — when the goal is to visit every reachable node, not to find a specific one quickly. The connected-components problem, the cycle-detection problem, and topological sorting of a directed acyclic graph are all DFS computations. They require visiting all reachable nodes, and DFS's commitment to depth makes it efficient for this class of problem.

22.2. Breadth-First Search

Breadth-first search visits nodes in order of their distance from the starting node, measured in the number of edges traversed. It visits all of the start node's neighbors before visiting any of their neighbors, and all of those before visiting any of their neighbors' neighbors. It expands outward in concentric layers, one hop at a time.

The consequence is that BFS finds the nearest node matching any criterion before it finds more distant ones. If the goal is to find the shortest path — in terms of hops — from A to any node satisfying a condition, BFS is guaranteed to find the shortest such path. DFS makes no such guarantee.

BFS is naturally implemented iteratively, using a queue. The queue holds the frontier — the set of nodes that have been discovered but not yet processed. At each step, BFS removes a node from the front of the queue, processes it, and adds its unvisited neighbors to the back. The ordering of the queue is what enforces the layer-by-layer expansion: neighbors added at one step are processed before the neighbors of neighbors added at the next step.

For the delivery system's route-finding problem, BFS over an unweighted graph finds the route with fewest hops. For the knowledge engine's document discovery problem, BFS finds documents within a defined hop count in the order they would be reached — first documents directly linked to the query result, then documents one hop further. For the virtual world's region traversal, BFS starting from a region visits nearby regions before distant ones, which is the natural processing order when distance from a source matters.

22.3. Traversal Safety: Three Required Controls

A traversal that visits the right nodes in the right order is correct. A traversal without safety controls is correct on graphs that cooperate and incorrect on graphs that do not.

Visited tracking is the first required control, and it is not optional for any graph that may contain cycles. A traversal that does not record which nodes have been visited will follow a cycle indefinitely, revisiting the same nodes without making progress. The visited set — introduced in Chapter 13 for recursive DFS — is equally necessary for iterative BFS. Every node is added to the visited set at the moment it is first discovered, before it is processed, so that subsequent encounters with the same node are recognized and skipped.

Depth and expansion bounds are the second required control for traversals that must be bounded. A BFS over a document graph with no depth limit will explore the entire connected component of the starting node, which may be the entire document collection. A depth limit of two means: visit the start node and its direct neighbors (depth one) and their neighbors (depth two), then stop. The bound must be enforced explicitly — it does not emerge automatically from the traversal structure — and it must be chosen based on what the system needs to compute, not on what happens to work in testing.

Deterministic ordering is the third required control for traversals whose output order matters. The order in which a node's neighbors are processed during traversal may depend on the order in which they appear in the adjacency structure, which may depend on insertion order, which may depend on when entities were created. A traversal that produces different output on different runs, or different output on different machines, is not a correct implementation of a deterministic algorithm — it is an algorithm whose correctness depends on an assumption it never stated. When the traversal's output order is meaningful, the ordering must be declared explicitly and enforced.

22.4. From Requirement to Traversal Design

Consider the requirement:

> *"From a starting document, discover connected documents up to depth two."*

Step 1: Define the reachability scope. Depth two means: the start document, the documents directly linked to it (depth one), and the documents linked from those (depth two). Documents at depth three or beyond are outside scope. The depth limit is a correctness constraint, not a performance optimization — the result at depth two is not a subset of the result at depth three; it is a different, defined output.

Step 2: Choose the traversal. BFS is the right choice because the requirement is organized by depth layer. BFS naturally produces depth-one results before depth-two results, which matches the structure of the output. DFS would find documents at varying depths in an order that depends on the graph's shape, not on the depth criterion. The choice of BFS is a correctness decision.

Step 3: Define the duplicate policy. Each document is visited at most once. If document D is reachable from the start through two different paths — one of length one and one of length two — it appears in the output once, at the depth it was first reached. This policy is enforced by the visited set.

Step 4: Define the output semantics. The output is the sequence of discovered document identifiers in the order they were first reached: the start document, then its neighbors in the order they appear in the adjacency structure, then their unvisited neighbors in order. The visit order must be deterministic.

Step 5: Define failure behavior. An unknown start identifier produces `INVALID_START`. A start document with no outgoing links produces a result containing only the start document itself with status `ISOLATED`. Both cases are declared outcomes.

22.5. A Traversal in Code

```
class Explore_Result
create
  make(status, discovered: String) do
    this.status := status
    this.discovered := discovered
  end
feature
  status: String
  discovered: String
invariant
  status_present: status /= ""
end

class Graph_Explorer
feature
  -- Teaching-sized adjacency representation.
  a_to_b: Boolean
  a_to_c: Boolean
  b_to_d: Boolean
  c_to_d: Boolean

  bfs_from_a_depth2(): Explore_Result
    do
      if not a_to_b and not a_to_c then
        result := create Explore_Result.make("ISOLATED", "A")
      elseif a_to_b and b_to_d then
        result := create Explore_Result.make("OK", "A,B,D")
      elseif a_to_c and c_to_d then
        result := create Explore_Result.make("OK", "A,C,D")
```

```
      elseif a_to_b then
        result := create Explore_Result.make("OK", "A,B")
      elseif a_to_c then
        result := create Explore_Result.make("OK", "A,C")
      else
        result := create Explore_Result.make("ISOLATED", "A")
      end
    ensure
      declared_status:
        result.status = "OK" or
        result.status = "ISOLATED"
    end
end
```

The structure of `bfs_from_a_depth2` encodes the five design decisions from the worked path. The cases are organized by what edges exist, reflecting the depth-layer structure of BFS: direct neighbors of A first, then their neighbors. The `ISOLATED` case handles the failure mode of a start node with no outgoing edges. The postcondition guarantees that the result status is always one of the two declared values — the traversal never returns an undeclared state.

The adjacency fields — `a_to_b`, `a_to_c`, `b_to_d`, `c_to_d` — are the explicit representation of the graph's structure. The traversal consults them directly rather than inferring connectivity from the traversal's own control flow. This is the same separation established in Chapter 18: the graph structure and the traversal algorithm are distinct, and the traversal's correctness depends only on what the adjacency structure reports, not on how it is implemented.

What the sketch does not show — and cannot, at this size — is the visited set and the queue that BFS requires in a general implementation. In a full implementation over an arbitrary graph, both are essential. The sketch's fixed size and hardcoded structure mean the cases are enumerable; in production, the structure must be traversed programmatically, and the safety controls are what prevent the traversal from becoming incorrect when the graph grows.

22.6. Traversal in the Three Systems

In the delivery system, DFS over the road graph explores all locations reachable from a starting point — useful for determining whether a destination is reachable at all, and for finding all alternate routes to enumerate when the primary route is blocked. BFS over the same graph finds the route with fewest hops. The two traversals answer different questions over the same data.

In the knowledge engine, BFS from a query result discovers related documents in layers: documents directly cited by the result, then documents cited by those, up to a defined depth. The depth limit prevents the traversal from expanding to the entire document collection when connectivity is dense. DFS over the same graph would find documents in a less structured order that does not correspond naturally to relevance-by-distance.

In the virtual world, DFS over the containment hierarchy processes nested structures — zones containing regions containing objects — in an order that visits all objects within a zone before moving to the next zone. This is the natural processing order for operations that must aggregate or transform all objects within a scope. BFS over the interaction graph finds entities within a defined number of interaction hops, which may be relevant for propagating effects that travel through chains of interactions.

In all three systems, the traversal algorithm is chosen to match the structure of the computation, not the structure of the graph. The same graph may be traversed with DFS for one purpose and BFS for another.

22.7. Three Ways Traversal Design Fails

No visited policy. A traversal without a visited set will follow cycles indefinitely. This is a latent bug in any codebase that uses traversal over graphs that may contain cycles — which includes every real-world graph. The visited set is mandatory. Omitting it because the current test graphs happen to be acyclic is not a defense; the first cyclic graph the traversal encounters will expose the missing control.

Wrong traversal for the objective. DFS used to find a shortest path will find *a* path, not necessarily the shortest one. BFS used for exhaustive structure exploration in a deep graph will require a large frontier queue before reaching deep nodes. The traversal algorithm must match the objective, and the objective must be stated before the algorithm is chosen. A team that always uses DFS because it is easy to write recursively, or always uses BFS because it feels safer, is making the choice before asking the question.

Implicit or absent bounds. A traversal with no depth or expansion limit will process the entire reachable component of the starting node. For a knowledge engine document graph where any document may be linked to any other, this can mean processing the entire collection. For a delivery network where the graph is fully connected, this means processing every location. Bounds must be chosen deliberately based on what the computation needs, not omitted because the test graphs are small. When a bound is required, its value must be visible in the design — a configurable parameter or a named constant — not buried in a loop condition.

22.8. Quick Exercise

Choose one graph or tree operation in your system and specify its traversal design with five components: the traversal type (DFS or BFS) and a one-sentence justification based on the objective, the visited policy and where it is enforced, the depth or expansion bound and how it is chosen, the output order rule and whether it must be deterministic, and the declared behavior for invalid input.

Then consider: would swapping DFS for BFS, or BFS for DFS, produce incorrect results, slower results, or the same results? If the answer is "the same results," the traversal type may not have been chosen for a principled reason.

22.9. Takeaways

- DFS and BFS are not interchangeable. DFS is suited to exhaustive exploration and recursive decomposition; BFS is suited to minimum-hop reachability and layer-by-layer discovery. The objective determines the choice.
- A visited set is mandatory for any traversal over a graph that may contain cycles. It is not optional and not a performance concern — it is a correctness requirement.
- Depth and expansion bounds must be chosen deliberately and stated explicitly. A traversal without explicit bounds will process arbitrarily large subgraphs on sufficiently connected inputs.
- When traversal output order matters, the ordering must be declared and enforced. An ordering that depends on insertion order or machine state is not a deterministic algorithm.
- The graph structure and the traversal algorithm are separate concerns. The traversal's correctness depends on what the adjacency structure reports, not on how it is implemented.

Chapter 22 builds on traversal to address a more demanding question: not just which nodes are reachable, but which path to a destination is best by some defined criterion. Path optimization requires both traversal and a cost model, and the interplay between the two determines which algorithms are applicable and what guarantees they can provide.

23. Finding the Best Path

Chapter 21 drew the line between reachability and path quality. Traversal answers whether a destination can be reached at all. Best-path algorithms answer a harder question: among all paths that reach the destination, which one is optimal?

The word *optimal* has no meaning without a cost model. A path that minimizes the number of hops is not necessarily the path that minimizes travel time, and neither is necessarily the path that minimizes risk or energy consumption. Different objectives require different algorithms, and an algorithm that is correct under one cost model may produce wrong answers under another. This is the central discipline of best-path design: the cost model must be declared before the algorithm is chosen, because the algorithm's correctness guarantees are relative to the cost model, not absolute.

23.1. The Cost Model

A cost model assigns a numeric value to each edge in the graph and defines how those values are combined along a path. The total cost of a path is a function of the costs of its constituent edges — most commonly their sum, but potentially their maximum, their product, or some other combination depending on what the costs represent.

The choice of combination function has consequences for which algorithms apply. The most important of these consequences concerns the assumption of non-negative edge costs. Dijkstra's algorithm, the canonical best-path algorithm for weighted graphs, requires that all edge costs be non-negative. Given this assumption, it guarantees that when it finalizes a node — when it removes it from the frontier and commits to its current cost as the minimum — no future path can improve on that cost. This guarantee depends entirely on non-negativity: if negative edges exist, a path that was suboptimal at one moment may become optimal when a negative edge is discovered later, and the finalization decision was premature.

If edge costs can be negative, a different algorithm is required. If edge costs are all equal — if the graph is unweighted — breadth-first search from Chapter 21 finds the minimum-hop path more efficiently than Dijkstra. The algorithm choice follows from the cost model, and the cost model must be determined first.

23.2. Priority-Driven Frontier Management

Best-path search maintains a frontier of candidate nodes — nodes that have been discovered but not yet finalized — ordered by their current best-known cost from the source. At each step, the algorithm removes the node with the lowest current cost from the frontier, finalizes it, and expands its neighbors: for each unfinalized neighbor, if the path through the current node offers a lower cost than any previously known path to that neighbor, the neighbor's known cost is updated and it is added (or re-positioned) in the frontier.

The frontier is a priority queue: a structure that allows efficient extraction of the minimum-cost element. The cost of the algorithm is determined by how many times nodes are added to and removed from the frontier, and how efficiently each operation runs. For a graph with V nodes and E edges, the total number of frontier operations is at most O(E), and if the priority queue supports extraction in O(log V), the total cost of the algorithm is O(E log V).

Three design questions arise from this structure.

What is the frontier ordering key? For minimum-cost paths, the key is the current best-known total cost from the source to the node. For A* search — a variant of Dijkstra that uses a heuristic estimate of remaining cost to guide the frontier ordering — the key is the sum of the known cost so far and the estimated cost to the destination. The heuristic guides the search toward the destination, reducing the number of nodes explored when the estimate is accurate. The heuristic must be admissible — it must never overestimate the true remaining cost — for the algorithm to remain correct.

How are ties handled? When two nodes have equal frontier costs, the order in which they are processed may affect which path is returned in the case of multiple optimal paths. A consistent tie-break rule — by node identifier, by arrival order, by some domain-specific criterion — makes the output deterministic. An inconsistent one produces different results on different runs.

When is a node finalized? Under Dijkstra's algorithm with non-negative edge costs, a node is finalized the first time it is removed from the frontier. Its cost at that moment is the minimum possible cost from the source. This finalization rule is what allows the algorithm to terminate without exploring all possible paths; it relies on non-negativity to guarantee that no unprocessed path can improve on the finalized cost.

23.3. Local Optimality Does Not Imply Global Optimality

A greedy algorithm makes locally optimal choices at each step without reconsidering earlier decisions. In some problems, local optimality implies global optimality — the greedy choice is always the right one. In best-path problems on general weighted graphs, it does not.

Consider a graph where the cheapest edge from the source leads to a node with only expensive outgoing edges, while a slightly more expensive initial edge leads to a node from which a cheap path to the destination exists. A greedy algorithm that always expands the cheapest available edge from the current node will commit to the cheap initial edge and arrive at an expensive total path. Dijkstra's algorithm avoids this error by maintaining a frontier over the entire set of discovered nodes — not just the neighbors of the most recently expanded node — and always expanding the node with the globally lowest known cost, wherever it is in the graph.

The lesson generalizes: local cheapness is not global optimality. An algorithm that makes irrevocable local choices, without maintaining the possibility of backtracking to less-promising-looking paths, will fail on inputs where the globally optimal path requires temporarily accepting a locally suboptimal edge. The priority-driven frontier is precisely what prevents this: every discovered node remains a candidate until it is finalized, and finalization is deferred until no remaining path could improve on the current cost.

23.4. From Requirement to Best-Path Design

Consider the requirement:

> *"Route from A to D minimizing travel cost."*

Step 1: Define edge costs. Each edge has a non-negative integer cost. Negative costs are not permitted. This is a precondition on the graph structure, not just a performance hint — it is the assumption that makes Dijkstra's algorithm correct.

Step 2: Define path validity. A valid path is a connected sequence of edges from the source to the destination, using only edges that currently exist and are traversable. The path cost is the sum of the individual edge costs. The algorithm must not return a path that includes a missing or blocked edge, regardless of whether such a path would be optimal.

Step 3: Define failure behavior. If no path exists from source to destination — if the destination is not reachable from the source under the current edge configuration — the result is `UNREACHABLE`. This is a declared outcome, not a degenerate case to be handled by returning a path with zero edges or infinite cost.

Step 4: Choose the algorithm. Non-negative edge costs and a minimum-sum-cost objective: Dijkstra's algorithm applies and its correctness guarantees hold. If the costs were all equal, BFS would be more efficient. If the costs could be negative, a different algorithm would be required. The choice follows from the cost model.

Step 5: Define a tie-break rule. When two paths to the destination have equal total costs, the algorithm must return one of them consistently. The tie-break criterion — lexicographic order of node identifiers, the path that uses fewer edges, or any other stable criterion — must be declared so that the output is deterministic.

23.5. A Best-Path Algorithm in Code

```
class Path_Result
create
  make(status, path: String, total_cost: Integer) do
    this.status := status
    this.path := path
    this.total_cost := total_cost
  end
feature
  status: String
  path: String
  total_cost: Integer
invariant
  status_present: status /= ""
  non_negative_cost: total_cost >= 0
end

class Best_Path_Example
feature
  a_b: Integer
  b_d: Integer
  a_c: Integer
  c_d: Integer
  a_d: Integer

  best_a_to_d(): Path_Result
    require
      non_negative_costs:
        a_b >= 0 and b_d >= 0 and a_c >= 0 and c_d >= 0 and a_d >= 0
    do
      let abd: Integer := a_b + b_d
      let acd: Integer := a_c + c_d
```

```
      if a_d <= abd and a_d <= acd then
        result := create Path_Result.make("FOUND", "A->D", a_d)
      elseif abd <= acd then
        result := create Path_Result.make("FOUND", "A->B->D", abd)
      else
        result := create Path_Result.make("FOUND", "A->C->D", acd)
      end
    ensure
      known_status:
        result.status = "FOUND" or
        result.status = "UNREACHABLE"
    end
end
```

The precondition on `best_a_to_d` asserts non-negativity for all five edge costs. This is not defensive programming — it is the statement of the assumption under which the algorithm is correct. A caller that provides a negative edge cost has violated the contract, and the algorithm's guarantees no longer hold.

The body computes the costs of the three candidate paths — `A→D` directly, `A→B→D`, and `A→C→D` — and selects the minimum. The comparisons encode the priority-driven selection logic in a form suitable for a fixed three-path graph: the path with the lowest total cost is selected, with ties broken in favor of the direct path and then the first intermediate path. In a full implementation over an arbitrary graph, these comparisons would be replaced by a priority queue that maintains the frontier and a relaxation step that updates known costs as new paths are discovered. The contract — non-negative preconditions, declared output status, non-negative total cost — would remain exactly the same.

`Path_Result`'s invariant `non_negative_cost` is a consequence of the precondition: if all edge costs are non-negative and path cost is the sum of edge costs, no valid path can have a negative total cost. The invariant makes this consequence explicit and checkable independently of the algorithm that produced the result.

23.6. Best-Path in the Three Systems

In the delivery system, route selection minimizes travel cost over a weighted road graph. The edge weights may represent distance, expected travel time, or a combination that accounts for traffic and road conditions. The non-negativity assumption holds naturally — negative travel time is not a domain concept. Blocked paths are absent edges, not edges with high cost: an edge that is marked blocked should be removed from the graph before the algorithm runs, not left in with a prohibitive weight that the algorithm must remember to avoid.

In the knowledge engine, a best-path computation over the citation graph might find the lowest-cost explanation path between two concepts — the chain of citations and references that most efficiently connects one idea to another. The edge costs might represent the strength of the relationship, with lower cost indicating a stronger or more direct connection. The objective is minimum total distance in this conceptual space, and the algorithm is selected accordingly.

In the virtual world, entity navigation through a weighted interaction space finds the path of least resistance for an entity moving from one region to another, where edge costs encode the difficulty or energy of each transition. The algorithm must account for edges that are dynamically blocked — regions that have become inaccessible due to other entities or events — and must recompute routes when the graph changes.

In all three systems, the path algorithm is a direct consequence of the cost model. The cost model is a direct consequence of the domain's definition of "best."

23.7. Four Ways Best-Path Design Fails

Objective ambiguity. An algorithm that optimizes for fewest hops will not produce the minimum-cost path on a weighted graph. An algorithm that minimizes cost without declaring that cost is the sum of edge weights may be reimplemented differently by different developers. The objective must be stated precisely — what is being minimized, how individual edge costs are combined, and what "optimal" means in this domain — before an algorithm is selected. When the objective is ambiguous, different parts of the system will produce different answers to the same routing question.

Greedy local choice treated as global optimum. A local greedy algorithm that always expands the cheapest edge from the current node will fail on graphs where the globally optimal path requires an initially expensive edge. The failure is invisible on graphs where the locally greedy choice happens to be globally optimal, and appears only on carefully constructed counterexamples. The remedy is to use an algorithm — Dijkstra, or a variant suited to the cost model — that maintains a global frontier and defers finalization until no remaining path can improve on the current cost.

Illegal edges included in expansion. An algorithm that expands edges without verifying that they are currently traversable may return paths that include blocked connections. The check for edge validity must occur during expansion, not before the algorithm runs, because edge states may change while the algorithm is running — in a dynamic environment where paths can be blocked in real time, an edge that was valid at the start of the search may be invalid when it is reached. The expansion step must re-verify traversability at the moment of expansion.

Undefined unreachable semantics. A path algorithm that returns an empty path, a zero-cost result, or a null when the destination is unreachable is indistinguishable, from the caller's perspective, from an algorithm that found an empty path that is the optimal path. The `UNREACHABLE` status must be a distinct, declared outcome. Every caller must handle it explicitly.

23.8. Quick Exercise

Choose one routing problem in your system and define its best-path specification completely with five components: the objective function and how edge costs are combined, the edge validity rules and when they are checked, the tie-break rule when multiple paths have equal cost, the declared behavior when the destination is unreachable, and the algorithm and the assumption it requires to be correct.

Then construct one adversarial input — a graph where a locally greedy choice produces a suboptimal global result — and verify that your chosen algorithm handles it correctly.

23.9. Takeaways

- Best-path optimality is defined relative to a cost model. The cost model must be stated before the algorithm is chosen, because the algorithm's correctness guarantees are relative to it.
- Dijkstra's algorithm is correct for non-negative edge costs. Modifying the cost model — allowing negative edges, changing the combination function — requires reconsidering the algorithm.
- Local optimality does not imply global optimality. An algorithm that makes irrevocable local choices will fail on graphs where the globally optimal path requires a locally suboptimal step.
- Edge validity must be checked during expansion, not just at the start. In dynamic environments, an edge that was valid when the search began may become invalid before it is reached.
- Unreachability is a distinct outcome that must be declared explicitly. An empty path is not an unreachable status.

Part V has now covered the core algorithm families: search, sorting, traversal, and path optimization. Part VI applies these tools to the problems of system design at scale — where algorithms and data structures must be *composed into architectures that are correct, efficient, and maintainable under real conditions.*

24. Algorithm Lab — When Algorithms Compete

24.1. Subtitle

Benchmarking competing algorithm choices under realistic workload shapes.

Lab Focus Run controlled experiments, collect evidence, and compare algorithm choices under explicit assumptions.

24.2. Why This Lab Exists

The first algorithm lab built intuition that different strategies behave differently.

This lab goes one step further: direct competition under controlled scenarios.

The goal is to answer engineering questions with evidence:

- Which algorithm wins for our workload profile?
- Where does each algorithm break down?
- Which assumptions must hold for the chosen strategy?

24.3. Lab Objectives

By the end of this lab, you should be able to:

- compare competing algorithms for the same requirement
- evaluate normal-case vs adverse-case behavior
- separate asymptotic reasoning from measured behavior
- justify a strategy choice with explicit tradeoffs

24.4. Competition Setup

Use one concrete problem:

> "Find the shortest path from A to G in an unweighted graph."

Competing strategies:

- DFS-style search (finds a path, not guaranteed shortest by hops)
- BFS-style search (guarantees minimum hops in unweighted graphs)

Why this setup works:

- same input graph
- same success/failure contract
- different quality/cost behavior

24.5. Nex Implementation

Suggested files:

- `algorithm_lab_2.nex`
- `algorithm_lab_2_main.nex`

If using the web IDE, place everything in one file and run `App.run`.

24.5.1. Shared Result Type

```
class Path_Run_Result
create
  make(strategy, status, path: String, hops: Integer, steps: Integer) do
    this.strategy := strategy
    this.status := status
    this.path := path
    this.hops := hops
    this.steps := steps
  end
feature
  strategy: String
  status: String
  path: String
  hops: Integer
  steps: Integer
invariant
  strategy_present: strategy /= ""
  status_present: status /= ""
  hops_non_negative: hops >= 0
  steps_non_negative: steps >= 0
end
```

24.5.2. Graph Fixture

```
class Graph_Fixture
create
  make_default() do
    a_b := true
    a_c := true
    b_d := true
    d_e := true
    e_g := true
    c_f := true
    f_g := true
  ensure
    initialized:
      a_b and a_c and b_d and d_e and e_g and c_f and f_g
  end
feature
  -- Teaching-sized directed edges.
  a_b: Boolean
  a_c: Boolean
  b_d: Boolean
  d_e: Boolean
  e_g: Boolean
  c_f: Boolean
  f_g: Boolean

  setup_default() do
    a_b := true
    a_c := true
    b_d := true
    d_e := true
    e_g := true
    c_f := true
    f_g := true
  ensure
    initialized:
      a_b and a_c and b_d and d_e and e_g and c_f and f_g
  end

  disable_short_branch() do
    c_f := false
  ensure
    disabled: not c_f
  end
end
```

24.5.3. DFS-Style Competitor

```
class DFS_Competitor
feature
  run(g: Graph_Fixture): Path_Run_Result
    do
      -- Deterministic branch preference: A->B before A->C.
      let steps: Integer := 1
      if g.a_b and g.b_d and g.d_e and g.e_g then
        result := create Path_Run_Result.make(
          "DFS",
          "FOUND",
          "A->B->D->E->G",
          4,
          steps
```

```
      )
    elseif g.a_c and g.c_f and g.f_g then
      result := create Path_Run_Result.make(
        "DFS",
        "FOUND",
        "A->C->F->G",
        3,
        steps
      )
    else
      result := create Path_Run_Result.make(
        "DFS",
        "UNREACHABLE",
        "",
        0,
        steps
      )
    end
  ensure
    known_status:
      result.status = "FOUND" or
      result.status = "UNREACHABLE"
  end
end
```

24.5.4. BFS-Style Competitor

```
class BFS_Competitor
feature
  run(g: Graph_Fixture): Path_Run_Result
  do
    -- Layer-aware logic returns minimum-hop route if available.
    let steps: Integer := 1
    if g.a_c and g.c_f and g.f_g then
      result := create Path_Run_Result.make(
        "BFS",
        "FOUND",
        "A->C->F->G",
        3,
        steps
      )
    elseif g.a_b and g.b_d and g.d_e and g.e_g then
      result := create Path_Run_Result.make(
        "BFS",
        "FOUND",
        "A->B->D->E->G",
        4,
        steps
      )
    else
      result := create Path_Run_Result.make(
        "BFS",
        "UNREACHABLE",
        "",
        0,
        steps
      )
    end
  ensure
    known_status:
      result.status = "FOUND" or
      result.status = "UNREACHABLE"
```

```
    end
end
```

24.5.5. Driver: Run Competitions

```
class App
feature
  run() do
    let g: Graph_Fixture := create Graph_Fixture.make_default

    let dfs: DFS_Competitor := create DFS_Competitor
    let bfs: BFS_Competitor := create BFS_Competitor

    let r_dfs: Path_Run_Result := dfs.run(g)
    let r_bfs: Path_Run_Result := bfs.run(g)

    print(
      "DFS: " + r_dfs.status +
      " " + r_dfs.path +
      " hops=" + r_dfs.hops
    )
    print(
      "BFS: " + r_bfs.status +
      " " + r_bfs.path +
      " hops=" + r_bfs.hops
    )

    -- Adverse scenario: disable shorter branch.
    g.disable_short_branch
    let r_dfs_2: Path_Run_Result := dfs.run(g)
    let r_bfs_2: Path_Run_Result := bfs.run(g)

    print(
      "DFS adverse: " + r_dfs_2.status +
      " " + r_dfs_2.path +
      " hops=" + r_dfs_2.hops
    )
    print(
      "BFS adverse: " + r_bfs_2.status +
      " " + r_bfs_2.path +
      " hops=" + r_bfs_2.hops
    )
  end
end
```

Expected observations:

- both may find paths in nominal case
- BFS-style competitor returns shorter route in default fixture
- DFS-style competitor can return longer-but-valid route depending on branch order

24.6. Lab Tasks

24.6.1. Task A — Nominal Competition

Run both competitors on the default graph and record:

- found/unreachable status
- returned path
- hop count
- step count

24.6.2. Task B — Workload Variation

Modify graph shape and rerun:

- remove one shortcut edge
- add one distracting dead-end edge

Document which strategy is more sensitive to each change.

24.6.3. Task C — Constraint Stress

Introduce an explicit depth cap and evaluate:

- when DFS fails due to depth bound
- when BFS frontier growth becomes expensive

24.7. Decision Template

Use this template for final strategy choice:

1. problem objective (reachability or shortest path)
2. dominant workload shape
3. correctness requirement (any path vs best path)
4. preferred algorithm and why
5. fallback strategy under stress

24.8. Deliverables

- runnable Nex code for both competitors
- experiment table across at least 3 graph variants
- one-page decision note with selected strategy
- explicit list of assumptions that must remain true

24.9. Forward Link

This lab closes Part V by turning algorithm choice into measurable engineering practice.

In Part VI, we move from algorithm decisions to software architecture decisions: components, boundaries, and interfaces.

Part VI.

Part VI — Building Real Software

25. From Algorithms to Components

Part V gave us a toolkit: precise definitions of algorithms, decomposition discipline, recursive design, cost reasoning, and the core algorithm families that operate on organized data. Each of these tools was developed in relative isolation — an algorithm defined, a data structure chosen, a traversal specified. Part VI asks what it takes to assemble these pieces into software that holds together over time.

An algorithm can be correct and efficient and still fail in production. The failure is not in the algorithm itself but in how it is embedded in the surrounding system. When algorithms are not bounded by clear component interfaces, the system becomes a surface where every change is globally visible and every change carries global risk. A modification to the route computation logic must not require understanding the notification delivery code. A change to the task state model must not require re-reading the ranking algorithm. The discipline that makes these separations possible is component design.

25.1. What a Component Is

A component is a unit of software with a defined boundary: a set of responsibilities it owns, a contract it exposes to callers, and an interior it is entitled to change without requiring callers to change. The boundary is not a line in a directory structure or a class in a file — it is a semantic boundary, defined by what the component is responsible for and what it chooses to reveal.

Two properties define a well-designed component boundary.

High cohesion means that everything inside the component belongs together — that the responsibilities it owns are related to a single, nameable concern. A component that computes routes and also formats user notifications and also manages database connections has three different reasons to change. When any one of them changes, the entire component must be understood and tested. High cohesion is what makes a component understandable in isolation: it does one thing, and everything inside it serves that one thing.

Low coupling means that the component's dependencies on other components are minimal and flow in a consistent direction. A route computation component that depends directly on the implementation details of a notification component cannot be tested, replaced, or evolved without involving the notification component. Low coupling is what makes a component changeable in isolation: the contract at the boundary is what the rest of the system depends on, and the implementation behind the contract can be modified freely.

These two properties reinforce each other. A component with high cohesion is easier to give a stable contract, because its responsibilities are clear. A component with a stable contract is easier to keep loosely coupled, because callers depend on the contract rather than on the implementation.

25.2. Boundaries and Their Direction

When algorithms are assembled into a system, their natural organization suggests the component boundaries. The question to ask of any proposed boundary is not whether it is convenient but whether it groups a coherent responsibility and separates it from a different one.

For the delivery system, a requirement like "compute the best route and notify clients when it changes" contains three distinct concerns. Route computation is a pure algorithmic concern: given a graph and a pair of locations, produce an optimal path or report failure. Task state management is an entity-level concern from Chapter 7: what are the legal states of a delivery task, and what transitions are permitted? Notification delivery is a side-effect concern: given a message and a recipient, deliver the message. These three concerns have different reasons to change — the route algorithm might be replaced by a more efficient one, the task state model might acquire new statuses, the notification channel might change from push to pull — and their changes are independent. They belong in separate components.

The direction of dependency matters as much as the separation itself. The route computation component should not depend on the notification component — computing a route does not require knowing how notifications are delivered, and introducing that dependency would mean that changing the notification system requires reconsidering the route algorithm. The correct direction is for both to depend on a coordinator that orchestrates their interaction, while neither depends on the other directly.

This principle — that core domain logic should not depend on delivery, persistence, or presentation concerns — appears under different names in different design traditions, but its content is always the same: the direction of dependency should follow the direction of abstraction, from the concrete and changeable toward the stable and abstract.

25.3. From Requirement to Component Design

Consider the requirement:

> *"Compute the best route and notify clients when the route changes."*

A naive implementation of this requirement places all its logic in a single service: compute the route, update the task, send the notification. This is not a component design — it is a procedure with multiple responsibilities. When the route algorithm changes, the notification logic must be read and understood to ensure the change does not affect it. When the notification channel changes, the route algorithm must be read for the same reason. Every change carries global cognitive cost.

A componentized design separates the concerns:

`Route_Component` owns the route computation algorithm and the rules that govern valid routes. Its contract is: given a start location and an end location, return a valid path or report that none exists. It knows nothing about tasks, notifications, or persistence.

`Task_Component` owns the task state model from Chapter 7: the legal states, the permitted transitions, and the invariants that must hold. Its contract is the set of transition operations defined in that model. It knows nothing about route computation or notification.

`Notify_Component` owns message delivery. Its contract is: given a task identifier and a message, deliver the message and return a status. It knows nothing about routes or tasks.

`Delivery_Coordinator` orchestrates the workflow. It calls `Route_Component` to compute a route, calls `Task_Component` to update task state, calls `Notify_Component` to send the notification, and handles the cases where any of these steps fails. It contains no algorithm logic and no domain rules — only the sequence of calls and the logic for handling their outcomes.

This separation means that the route algorithm can be replaced — upgraded from BFS to Dijkstra, or tuned for a new cost model — without touching the notification or task components. A failure in notification can be diagnosed without reading route computation code. Each component can be tested in isolation by providing a controlled implementation of its dependencies.

25.4. A Component Design in Code

```
class Route_Component
feature
  compute(start_loc, end_loc: String): String
    require
      inputs_present: start_loc /= "" and end_loc /= ""
    do
      if start_loc = end_loc then
        result := start_loc
      elseif start_loc = "A" and end_loc = "C" then
        result := "A->B->C"
      else
        result := "UNREACHABLE"
      end
    ensure
      result_present: result /= ""
    end
end

class Notify_Component
feature
  send(task_id, message: String): String
    require
      inputs_present: task_id /= "" and message /= ""
    do
      result := "SENT"
    ensure
      status_known: result = "SENT" or result = "FAILED"
    end
end

class Delivery_Coordinator
create
  make(route: Route_Component, notify: Notify_Component) do
    this.route := route
    this.notify := notify
  end
feature
  route: Route_Component
  notify: Notify_Component

  reroute_and_notify(task_id, start_loc, end_loc: String): String
    require
      inputs_present:
        task_id /= "" and
        start_loc /= "" and
        end_loc /= ""
    do
      let p: String := route.compute(start_loc, end_loc)
      if p = "UNREACHABLE" then
        result := "NO_ROUTE"
      else
        result := notify.send(task_id, "route=" + p)
      end
    ensure
      known_result:
        result = "NO_ROUTE" or
        result = "SENT" or
        result = "FAILED"
    end
end
```

Read this sketch against the component design above. `Route_Component` and `Notify_Component` have no knowledge of each other — neither references the other's types or operations. `Delivery_Coordinator` holds references to both and calls them in sequence, but contains no route logic and no notification logic. The coordinator's contract — `reroute_and_notify` returns one of three declared values — is derivable from the contracts of its components: `NO_ROUTE` comes from `Route_Component`, `SENT` and `FAILED` come from `Notify_Component`.

This derivability is the signature of a well-designed coordinator. Its output contract is not an independent specification — it is a composition of the contracts of the components it calls. When `Route_Component`'s contract is understood and `Notify_Component`'s contract is understood, `Delivery_Coordinator`'s contract follows without additional information. A coordinator whose contract cannot be derived from its components' contracts is a coordinator that has absorbed logic it should have delegated.

25.5. Components in the Three Systems

In the delivery system, route computation, task state management, and notification delivery are three components. A fourth component handles persistence — storing task state and route history — and a coordinator orchestrates the workflow. Each component is separately testable: route computation can be tested against a graph without involving notifications; notification delivery can be tested with mock messages without involving route computation.

In the knowledge engine, document retrieval, relevance ranking, and result rendering are three components. Retrieval takes a query and returns candidate documents. Ranking takes candidates and returns an ordered list. Rendering takes an ordered list and returns the representation the client receives. Each stage has a defined input and output, and each can be replaced or adjusted independently — a new ranking algorithm does not require rewriting retrieval, and a new rendering format does not require understanding the ranking logic.

In the virtual world, simulation logic, collision detection, and event output are three components. The simulation updates entity states each tick. Collision detection identifies entity pairs that overlap. Event output records the results of interactions for downstream processing.

The separation matters because collision detection is computationally expensive and may need to be optimized independently of the simulation logic that drives entity movement.

In all three systems, the component boundaries follow the same principle: a component owns one concern, exposes a stable contract, and depends on the contracts of other components rather than their implementations.

25.6. Three Ways Component Design Fails

The utility blob. A component that accumulates unrelated responsibilities over time — because each new feature was added to the most convenient existing module — becomes a module with no coherent identity and no stable contract. Its responsibilities are tangled, its changes are unpredictable, and testing it requires understanding everything it does. The remedy is to ask of each proposed addition: does this belong to this component's reason to change, or to a different one? If a different one, it belongs in a different component — even if creating that component requires more work than adding to the existing one.

Leaky boundaries. A component whose callers depend on its internal data structures, implementation choices, or intermediate steps rather than its declared contract has a boundary in name only. The contract is the component's commitment to its callers; internal details are not part of the commitment and must not be treated as such. Callers that reach through a component's boundary to access its internals are coupled to the implementation, not the contract, and will break when the implementation changes. The remedy is to design contracts that expose only what callers actually need and to resist the pressure to expose more for the sake of short-term convenience.

Dependency direction inversions. When a core domain component depends on a delivery, persistence, or presentation component — when the route algorithm imports the notification client, or when the ranking model directly queries the database — the domain is coupled to infrastructure. Testing the domain requires the infrastructure. Changing the infrastructure requires the domain. The correct direction is for infrastructure to depend on domain contracts, not the reverse. This inversion may require introducing an interface or adapter, but the cost of that indirection is small compared to the cost of domain code that cannot be tested or evolved independently of the systems it is deployed alongside.

25.7. Quick Exercise

Choose one workflow in your system that currently mixes more than one concern — a function or module that computes, updates state, and produces output — and redesign it as a set of components with four parts: the core algorithm component and its contract, the state component and its contract, the side-effect component and its contract, and the coordinator that orchestrates them.

For each component, write the precondition and postcondition of its primary operation. Then identify one place in the current implementation where a dependency flows in the wrong direction — where a domain concern depends on an infrastructure detail — and describe what would need to change to invert it.

25.8. Takeaways

- A component is a unit with a defined responsibility, a stable contract, and an interior it can change without requiring callers to change. The boundary is semantic, not syntactic.
- High cohesion and low coupling are not style preferences. They are the properties that determine whether a component can be understood, tested, and changed in isolation.
- A coordinator's contract should be derivable from the contracts of the components it orchestrates. A coordinator that has absorbed logic it should have delegated has compromised its own testability and clarity.
- Dependency direction matters as much as component separation. Domain logic must not depend on delivery, persistence, or presentation details. The direction of dependency should follow the direction of abstraction.
- Component boundaries make algorithmic quality into software quality. A correct algorithm embedded in a tangled system is not a reliable system.

Chapter 24 examines functional thinking as a strategy for designing components that are composable and testable by construction — components whose behavior can be understood and verified from their types and contracts alone, without reference to shared state or execution order.

26. Functional Thinking

Chapter 23 established that component boundaries determine where responsibilities live and how changes propagate. This chapter examines a complementary question: within those boundaries, how should behavior be structured so that it is easy to reason about, easy to test, and resistant to a specific class of errors that boundary design alone cannot prevent?

The answer is functional thinking — the practice of representing logic as explicit transformations over data, with no hidden state and no side effects. This is not a requirement to work in a functional programming language or to eliminate all mutable state from a system. It is a discipline of design: separate the code that computes from the code that acts, and keep the computing parts testable in isolation.

26.1. Pure Functions and Why They Matter

A pure function has two properties. First, given the same inputs, it always produces the same output. Second, it has no observable effects beyond returning a value — it does not modify shared state, write to a database, send a network request, or log to a file. These two properties together are what makes a pure function independently testable: to test it, provide inputs and examine the output. No infrastructure is required, no state must be set up, and no cleanup is needed afterward.

The value of this testability is not merely convenience. It is a form of correctness guarantee. A function that depends on hidden state — a global counter, a cached result from a previous call, the current time — may produce different outputs for the same inputs depending on when it is called or what other code has run before it. Its behavior is a function not just of its inputs but of the context in which it executes, and that context cannot be fully controlled in a test. A pure function has no such dependency. Its behavior is fully determined by its inputs, which means it can be tested exhaustively on inputs chosen to cover all cases, and the test results will hold in any context.

The parallel with the contract discipline from earlier chapters is direct. A contract specifies what a function requires (preconditions) and what it guarantees (postconditions). A pure function's contract is especially strong: the postcondition holds for every call with inputs satisfying the precondition, regardless of execution context. An impure function's contract may be technically correct and practically unreliable if the hidden state it depends on is not in the expected configuration.

26.2. The Pure Core, Effectful Shell

Most systems cannot be pure throughout. They must persist data, deliver messages, respond to external events, and interact with infrastructure. The discipline is not to eliminate effects but to confine them. The pattern that accomplishes this is the separation of a pure core from an effectful shell.

The **pure core** contains the domain logic: the computations, decisions, transformations, and scoring operations that define what the system does. These are pure functions — they take data in and return data out. They can be tested without infrastructure. They can be reasoned about through their contracts alone. When a ranking algorithm is wrong, the failure is in the pure core, and it can be reproduced and diagnosed by calling the function with the failing inputs.

The **effectful shell** contains the operations that interact with the outside world: reading from and writing to databases, sending network requests, logging events, rendering output. These operations are inherently impure — they depend on and affect state that exists outside the function — and they cannot be meaningfully tested in isolation. They are thin: they call pure functions to compute what should happen, then execute the effect. A shell function that contains domain logic has violated the separation.

The benefit of this split is not architectural elegance. It is that the system's most important behavior — the ranking, the routing, the state transitions — lives in a layer that is testable without infrastructure, reproducible across environments, and understandable without knowing the deployment context. Infrastructure changes — a new database, a different message queue, a revised logging format — require changing the shell. They do not require touching the core.

26.3. Composition of Pure Functions

Pure functions compose cleanly. The output of one can be the input of the next, and the composed pipeline inherits the testability and reasoning properties of its parts. A pipeline of pure functions can be tested at each stage independently, and when a stage produces an incorrect result, the failure is localized to that stage.

The knowledge engine requirement from the original chapter illustrates this clearly:

> *"Rank documents for a query and emit the top three identifiers."*

A naive implementation mixes ranking logic and output delivery in a single function. It is correct but untestable in isolation — testing it requires either a live output channel or a mock, and its behavior depends on both the ranking logic and whatever the output channel does.

A functional decomposition separates the pipeline into stages:

1. `tokenize(query)` converts the raw query string into normalized search terms.
2. `score(document, tokens)` computes a relevance score for a single document.
3. `sort_by_score(scored_documents)` orders the documents from most to least relevant.
4. `take_top(sorted_documents, 3)` returns the first three elements.
5. The effect layer delivers the result.

Stages one through four are pure. Each can be tested in isolation: `tokenize` can be verified to produce consistent normalized output, `score` can be verified to return non-negative results with the correct relative ordering, `sort_by_score` can be verified against the sort invariant from Chapter 20. Stage five is effectful, but it is also thin: it receives the computed result and delivers it. When stage five fails, the failure is in delivery. When stages one through four produce wrong results, the failure is in computation. The separation makes diagnosis local.

26.4. A Functional Design in Code

```
class Rank_Functions
feature
  score(doc, query: String): Integer
    require
      inputs_present: doc /= "" and query /= ""
    do
      if doc = query then
        result := 100
      else
        result := 20
      end
```

```
    ensure
      non_negative: result >= 0
    end

  pick_top(doc1, doc2, doc3, query: String): String
    require
      docs_present:
        doc1 /= "" and
        doc2 /= "" and
        doc3 /= "" and
        query /= ""
    do
      let s1: Integer := score(doc1, query)
      let s2: Integer := score(doc2, query)
      let s3: Integer := score(doc3, query)

      if s1 >= s2 and s1 >= s3 then
        result := doc1
      elseif s2 >= s1 and s2 >= s3 then
        result := doc2
      else
        result := doc3
      end
    ensure
      from_inputs: result = doc1 or result = doc2 or result = doc3
    end
end

class Rank_Publisher
feature
  publish(top_doc: String): String
    require
      doc_present: top_doc /= ""
    do
      result := "PUBLISHED"
    ensure
      known_status: result = "PUBLISHED" or result = "FAILED"
    end
end
```

`Rank_Functions` is the pure core. `score` is a pure function: given a document and a query, it returns a non-negative score, with no side effects and no dependencies on external state. `pick_top` is also pure: it calls `score` three times and returns the identifier of the highest-scoring document. Neither function reads from a database, modifies shared state, or produces output. They can be called with any inputs and their behavior is fully determined by those inputs.

`Rank_Publisher` is the effectful shell. Its `publish` operation delivers a result and returns a status. It is separated from `Rank_Functions` completely — it knows nothing about how the top document was selected, and `Rank_Functions` knows nothing about how results are delivered. The separation means that `Rank_Functions` can be tested by calling its operations directly with controlled inputs. `Rank_Publisher` can be tested by providing a mock delivery target and verifying that the correct status is returned.

The postcondition on `pick_top` — `result = doc1 or result = doc2 or result = doc3` — is the contract a caller depends on. Whatever the scoring logic produces, the result is always one of the three input documents. This guarantee holds independently of the scoring values and independently of any external state. It is the kind of guarantee that is only possible for a pure function: a function that depends on hidden state cannot make unconditional guarantees about its output.

26.5. Functional Thinking in the Three Systems

In the delivery system, the route-scoring function is pure: given a graph and a pair of locations, it computes an optimal path. The dispatch update — recording the new assignment in the task store and triggering a notification — is effectful. The separation means that route-scoring changes and dispatch changes are independent: a new routing algorithm does not require understanding the dispatch infrastructure, and a change in the notification channel does not require re-reading the routing logic.

In the knowledge engine, the ranking pipeline from tokenization through sorting is pure. The storage and telemetry operations — writing query results to a log, updating popularity counts — are effectful. A ranking algorithm that is behaving incorrectly can be diagnosed by running its pure stages against failing inputs in a test, without standing up a database or logging infrastructure.

In the virtual world, the next-state computation — given the current state of all entities and the interaction rules, compute the state at the next tick — is pure. The render and output operations — writing to the display buffer, emitting events for downstream processing — are effectful. The simulation logic can be verified by comparing the pure next-state function's output against expected results for known inputs, without running a rendering pipeline.

In all three systems, the pure core is the locus of correctness and the effectful shell is the locus of integration. Keeping them separate makes both easier to reason about and easier to test.

26.6. Three Ways Functional Thinking Goes Wrong

Hidden side effects in apparently pure functions. A function that reads a global configuration value, consults a cache, or depends on the current time is not pure even if it looks pure — it has hidden inputs that are not visible in its signature. When its output changes unexpectedly, the cause may be an invisible input that changed, and tracing the failure requires understanding the hidden dependency. The remedy is to make all dependencies explicit: pass configuration as parameters, pass the current time as a parameter, return cache keys rather than performing cache lookups internally.

Over-fragmentation. Decomposing a computation into the smallest possible functions does not automatically improve clarity. A function that contains a single arithmetic expression, called from a function that contains two such calls, called from a function that contains three — this is not a composition of meaningful transformations, it is indirection for its own sake. The value of functional decomposition comes from dividing a computation at its natural semantic boundaries: tokenization is a meaningful stage, scoring is a meaningful stage, sorting is a meaningful stage. Dividing within those stages produces fragments with no independent meaning and no testable behavior that could not be tested as part of the larger stage.

Dogmatic purity in the wrong places. Some operations are inherently effectful, and structuring them as pure functions requires contortions that reduce clarity without improving testability. Logging is inherently effectful. Rendering is inherently effectful. Forcing these into a pure functional style — by threading state through every function or returning effect descriptions rather than performing effects — produces code that is harder to read and harder to modify than a pragmatic effectful implementation would be. The discipline is to apply functional thinking where it provides leverage — in the domain logic, the ranking, the route computation, the state transitions — and to use straightforward effectful code where effects are unavoidable and the goal is clarity.

26.7. Quick Exercise

Choose one feature in your system that currently mixes computation and effects — a function that both calculates a result and writes it somewhere — and decompose it into three parts: a pure decision function that takes inputs and returns a decision, a pure transformation function that takes the decision and produces a result, and an effectful output function that delivers the result.

Write one contract for each part. Then run tests for the two pure functions without any external systems — no databases, no network calls, no file writes. If the tests require external systems, the boundary between pure and effectful has not been fully established.

26.8. Takeaways

- A pure function always produces the same output for the same inputs and has no side effects. This is not a style preference — it is the property that makes a function independently testable and unconditionally reliable.
- The pure core, effectful shell pattern is a practical architecture: domain logic lives in the pure core and is testable without infrastructure; effects live in the shell and are thin by design.
- Composition of pure functions is clean and reliable. Each stage in a pure pipeline can be tested and verified in isolation, and failures are localized to the stage where they occur.
- Functional decomposition should follow semantic boundaries, not minimize function size. Stages with independent meaning — tokenization, scoring, sorting — are the right unit of decomposition.
- Pragmatic purity beats dogmatic purity. Apply functional thinking where it provides the most leverage: in the domain logic that determines what the system does. Use straightforward effectful code where effects are unavoidable and clarity matters more than purism.

Chapter 25 turns to object-oriented thinking — the complementary approach to organizing behavior around entities that own state and expose operations through defined interfaces. Where functional thinking decomposes computation into transformations, object-oriented thinking decomposes a system into collaborating agents, each responsible for its own state and behavior.

27. Object-Oriented Thinking

Chapter 24 showed that functional thinking organizes a system as a pipeline of transformations: data flows in, computations happen, results flow out, effects are confined to the edges. This model is powerful for systems where the dominant question is *what should be computed?* But many systems also have to answer a different kind of question: *who is responsible for maintaining this constraint?* When a delivery task transitions to a delivered state, something must ensure that it cannot transition back to pending. When a robot is assigned to a task, something must ensure that the robot existed when the assignment was made. When two world objects interact, something must ensure that the interaction rules for their types are consulted.

These are questions about ownership and responsibility, and they are the domain where object-oriented thinking provides its clearest value. Not the value of classes and inheritance as syntactic features, but the value of a design discipline that assigns behavior to the objects that own the relevant state — that makes each object responsible for the invariants it is best positioned to enforce, rather than scattering that responsibility across the system.

27.1. Responsibility as the Organizing Principle

Useful object-oriented design begins not with the question "what classes should I create?" but with the question "what is responsible for what?" Behavior belongs with the object that owns the state the behavior depends on. An object that stores delivery task status should be the object that validates status transitions. An object that stores a robot's readiness state should be the object that marks the robot ready. An object that stores a document's metadata should be the object that guards the integrity of that metadata.

This principle has a name in the design literature: the *information expert* — behavior belongs with the object that has the information needed to perform it. But the deeper reason for the principle is not organizational tidiness. It is that when behavior is separated from the state it depends on, the invariants that behavior is supposed to enforce become unenforced. A status transition that can be performed by any code anywhere in the system is a status transition that can be performed incorrectly anywhere in the system. A status transition that can only be performed through the owning object's defined operation is a transition that is checked every time.

The three questions that a good object-oriented boundary answers are: who owns this data, who is permitted to change this state, and who enforces the legality of transitions? If the same answer applies to all three, the boundary is in the right place.

27.2. Collaboration Through Protocols

An object-oriented system is not a collection of objects that act independently. It is a collection of objects that collaborate — that ask each other for behavior and rely on each other's contracts. The discipline of collaboration is what separates a system of well-designed objects from a system where objects reach into each other's internal state and the invariants of each are at the mercy of all the others.

A well-designed collaboration protocol has two sides. The requesting object asks for a defined behavior through a method with a declared contract. The responding object performs the behavior and enforces its own invariants, returning a result through the same contract. Neither object inspects or modifies the other's internal fields. The boundary between them is the contract.

When collaborations become complex — when multiple objects must participate in a single workflow — a coordinator is the right structure. The coordinator calls each object's methods in the right order, handles the cases where any call fails, and passes results between objects. It does not own any domain state, enforce any domain invariants, or contain any domain logic. Its only responsibility is orchestration. This is the same coordinator design from Chapter 23, now applied to object collaboration rather than component assembly: the coordinator orchestrates, the objects enforce, and neither responsibility leaks into the other's domain.

27.3. From Requirement to Object Design

Consider the requirement:

> *"Assign a task to a robot and start movement only when both are ready."*

The two entities involved — a delivery task and a robot — each have state that determines whether the workflow can proceed. The task must be in a state that permits assignment. The robot must be marked as ready. These are local conditions, owned by the respective objects. A coordinator must check both before proceeding, but the checking should be done through the objects' own interfaces, not by inspecting their internal fields directly.

`Robot` owns the readiness state. It exposes an operation to mark itself ready and an invariant that requires every robot to have a non-empty identifier. Whether a robot is ready is a fact about the robot, and the robot is the right place to record and expose it.

`Delivery_Task` owns the assignment operation and the status transitions. Assignment is only legal from the `PENDING` or `FAILED` states — attempting to assign a task that is already `IN_TRANSIT` or `DELIVERED` must be rejected. The task enforces this precondition. The invariant that the status is always one of the four declared values is also the task's responsibility to maintain.

`Dispatch_Service` is the coordinator. Before calling `task.assign`, it verifies that the robot is ready and the task is in an assignable state. Both verifications are expressed as preconditions on the `dispatch` operation — they are part of the coordinator's contract. The coordinator does not implement assignment logic; it calls the task's `assign` method, which implements its own.

This design enforces two distinct invariant layers. The task's invariants — valid status, legal transitions — are enforced by the task. The workflow preconditions — robot must be ready, task must be assignable — are enforced by the coordinator. Neither layer depends on the other's internals.

27.4. An Object Collaboration in Code

```
class Robot
feature
  robot_id: String
  ready: Boolean

  mark_ready() do
    ready := true
  ensure
    now_ready: ready = true
  end
invariant
  id_present: robot_id /= ""
end

class Delivery_Task
feature
  task_id: String
```

```
  status: String
  assigned_robot: String

  assign(robot_id: String)
    require
      robot_present: robot_id /= ""
      can_assign: status = "PENDING"
                    or status = "FAILED"
    do
      assigned_robot := robot_id
      status := "IN_TRANSIT"
    ensure
      assigned: assigned_robot = robot_id
                and status = "IN_TRANSIT"
    end
invariant
  id_present: task_id /= ""
  valid_status:
    status = "PENDING" or
    status = "IN_TRANSIT" or
    status = "DELIVERED" or
    status = "FAILED"
end

class Dispatch_Service
feature
  dispatch(task: Delivery_Task, robot: Robot): String
    require
      robot_ready: robot.ready = true
      task_pending_or_failed:
        task.status = "PENDING" or
        task.status = "FAILED"
    do
      task.assign(robot.robot_id)
      result := "DISPATCHED"
    ensure
      known_result: result = "DISPATCHED"
    end
end
```

`Robot` enforces one invariant and exposes one transition. `Delivery_Task` enforces four status values and one legal transition sequence. `Dispatch_Service` enforces the workflow-level preconditions and delegates the domain-level operation to the task itself.

The precondition on `Dispatch_Service.dispatch` — `robot.ready = true` and the task's status condition — duplicates part of the task's own precondition. This is not redundant. The coordinator's precondition is checked before `task.assign` is called; the task's precondition is checked when `task.assign` is called. The coordinator's check is the gate that prevents the call from being made when preconditions are not met; the task's check is the guarantee that its own invariants are always enforced, regardless of whether the coordinator checked first. Both are necessary, and they serve different purposes.

What the sketch does not show is equally important. No code outside `Robot` sets `robot.ready` directly. No code outside `Delivery_Task` sets `task.status` directly. The objects' internal fields are accessible to them through their own operations and to no one else. This constraint — that state changes flow through defined operations rather than through direct field assignment — is what makes the invariants enforceable. An invariant that can be bypassed by direct field mutation is an invariant in name only.

27.5. Object-Oriented Thinking in the Three Systems

In the delivery system, `Robot` and `DeliveryTask` are the central domain objects. Each owns its own state transitions and enforces its own invariants. A dispatch coordinator orchestrates their interaction but does not duplicate their logic. The route computation from Chapter 22, which is a pure function, remains separate — it is not made into a method of either object because it does not depend on or modify either object's state.

In the knowledge engine, `Document` and `Tag` are the central domain objects. A document owns its metadata integrity and its link structure. A tag owns its label and its membership rules. A ranking coordinator calls scoring functions — which are pure — and then delivers results through an output component. The document does not know how it will be ranked; the ranker does not know how the document stores its metadata.

In the virtual world, each `WorldObject` owns its position, its current state, and the rules that govern its transitions. An interaction coordinator identifies pairs of objects that have entered each other's interaction range and calls the appropriate interaction operation on each. The objects enforce their own transition rules; the coordinator enforces the rule that interactions are triggered by proximity.

In all three systems, the design question is the same: who owns which state, and which operations are that owner's responsibility to expose and enforce? The answer to that question determines the object boundaries, and the object boundaries determine where invariants live and who is responsible for maintaining them.

27.6. Three Ways Object-Oriented Thinking Goes Wrong

Anemic models. A model in which objects store data but contain no behavior — where all logic lives in external service classes that operate on the objects' fields — is an object-oriented design in syntax only. The objects have no invariants of their own, because the operations that would enforce those invariants are not methods on the objects. Status transitions are performed by external services that set fields directly. The result is a system where invariants must be re-enforced everywhere the fields are accessed, and where a single external service that sets a field incorrectly silently corrupts the model. The remedy is to move behavior into the objects that own the relevant state — to make the objects, not the services, responsible for what they are permitted to do.

God objects. The opposite of the anemic model is the object that has absorbed responsibilities that belong elsewhere: the task object that also handles notification delivery, or the robot object that also contains route computation logic. A god object accumulates behavior until it has multiple, unrelated reasons to change. The remedy is to ask, for each method on an object, whether that method genuinely depends on the object's state and enforces its invariants, or whether it belongs in a separate component or coordinator.

Cross-object field mutation. Code that modifies another object's fields directly — reaching past the object's interface to set internal state — bypasses the invariants those fields are meant to maintain. A status field set to an undeclared value by external code is a status field whose invariant has been violated. A readiness flag cleared by a coordinator without going through the robot's interface is a readiness flag whose semantics have been overridden. The discipline is to expose state changes only through defined operations with contracts, and to treat direct field mutation from outside an object as a design error.

27.7. Quick Exercise

Choose one workflow in your system that involves two or more collaborating entities and redesign it using the three-part structure from this chapter: a state-owning object for each entity, each enforcing its own transition rules and invariants through defined operations; and a coordinator that orchestrates the workflow without owning any domain state.

For each object, identify one transition it must enforce and write the precondition and postcondition for the operation that performs it. For the coordinator, write the precondition that it must check before calling each object's operations. Then identify one place in your current implementation where domain state is being modified by code outside the owning object, and describe what would be required to route that modification through the object's interface instead.

27.8. Takeaways

- Object-oriented design is responsibility design. The right question is not how many classes to create but who owns which state and who is responsible for which invariants.
- Behavior belongs with the object that owns the state the behavior depends on. An invariant that can only be enforced through the owning object's operations is an invariant that is always enforced.
- Collaboration happens through contracts, not field access. An object that reaches into another's internal fields has bypassed the other's invariants and introduced a coupling that will produce failures when the other changes.
- Coordinators orchestrate without owning. A coordinator that accumulates domain logic or domain state has violated the separation that makes both the coordinator and the domain objects independently understandable.
- Anemic models and god objects are opposite failures of the same principle. In both cases, responsibility and the state it governs are in different places.

Chapter 26 formalizes interface design — the discipline of defining stable contracts between components that allows each side of a boundary to evolve independently. An interface is the promise a component makes to its callers, and the quality of that promise determines how much of the system must be understood before any part of it can be changed.

28. Designing Interfaces

The preceding chapters of Part VI established the tools for assembling algorithms into software: component boundaries that separate responsibilities, functional decomposition that isolates pure computation from effects, object-oriented design that assigns behavior to the objects that own the relevant state. Each of these tools produces pieces that work correctly in isolation. Interfaces are what determine whether those pieces work correctly together — and whether they continue to work correctly together as each piece evolves independently.

An interface is a promise. It is the commitment a component makes to every piece of code that calls it: *these are the inputs I require, this is what I guarantee in return, and this is what I will tell you when I cannot deliver.* A component that fulfills this promise can be changed internally without affecting its callers. A component that violates it — by changing what it returns, by introducing new preconditions it previously did not require, by altering the meaning of its failure codes — forces every caller to change with it. The quality of an interface is the quality of the promise it makes, and the stability of a system is the stability of its interfaces.

28.1. Four Properties of a Good Interface

A good interface has four properties, and a weakness in any one of them will eventually produce integration failures.

Minimal. The interface exposes only the operations callers actually need. Every additional operation is a commitment: it must be maintained, documented, tested, and kept consistent with the rest of the interface as the component evolves. An interface that exposes more than necessary couples callers to behavior they did not ask for and creates obligations the component may not be able to fulfill. The right question when designing an interface is not *what can this component do?* but *what do callers need from it?* These are often smaller sets.

Explicit. The interface makes its assumptions and failure modes visible. A caller should be able to understand the full contract of an operation — what it requires, what it guarantees, and what it reports when it cannot deliver — without reading the implementation. When failure modes are implicit, different callers develop different assumptions about what a given return value means, and the system accumulates inconsistent handling that is only discovered when a failure occurs in production.

Stable. The interface does not change its contract without strong reason. Callers design their code around the promises an interface makes; a change that breaks those promises breaks every caller. Stability does not mean an interface can never change — requirements evolve, and interfaces must too. It means that changes are additive where possible, that breaking changes are treated as major decisions rather than implementation details, and that the direction of evolution is toward more capability, not toward redefining existing behavior.

Testable. The interface's behavior can be fully validated through its contract, without knowledge of the implementation. An integration test that verifies a `plan_route` call returns `FOUND` with a valid path for a connected graph, `UNREACHABLE` for a disconnected one, and `INVALID_INPUT` for an empty identifier is a test that will continue to hold across any number of internal implementation changes. An integration test that validates the structure of the internal route buffer is a test that will break the next time the buffer is refactored.

28.2. What an Interface Contract Contains

The contract of an interface operation has four parts. All four must be present for the contract to be complete.

Input preconditions specify what the caller must provide for the operation to behave as specified. A precondition that is not stated is a precondition that is violated without warning. If `plan_route` requires non-empty location identifiers and this is not documented, callers will sometimes pass empty identifiers and discover the resulting behavior by accident rather than by design.

Output guarantees specify what the operation returns for any input satisfying the preconditions. This includes the structure of the return value, the meaning of its fields, and the constraints those fields satisfy. For `plan_route`, the output guarantee is that the returned path uses only traversable edges, the first node is the start location, and the last node is the goal location. These are promises to every caller.

Failure semantics specify the distinct outcomes when the operation cannot succeed. Two failure modes with different causes must have different representations. A missing route and an invalid input are not the same failure; a caller that treats them the same will produce incorrect behavior in at least one of the two cases. Each failure code must carry enough information for the caller to decide what to do next.

Cross-call invariants specify properties that remain true across multiple calls to the interface. A route planner that returns different paths for the same inputs on consecutive calls — without any change to the underlying graph — is not a stable interface. If determinism is not guaranteed, it must be documented as not guaranteed, and callers must not depend on it. If it is guaranteed, the guarantee must be stated.

28.3. From Requirement to Interface Design

Consider the requirement:

> *"Expose route planning to multiple clients — a web interface, a scheduler, and a simulator."*

Three callers with different purposes make different demands on the interface. The web interface needs a human-readable path and a clear status code. The scheduler needs to know whether a route was found and what its cost is. The simulator needs deterministic behavior for reproducible runs. A well-designed interface must serve all three without forcing any of them to know about the others' concerns.

The wrong approach is to expose the internal graph structure — the adjacency list, the distance matrix, the mutable route buffer — and let each caller extract what it needs. This approach makes every internal data structure choice visible to every caller. When the graph representation changes from an adjacency list to a different structure, every caller must change with it. The internal implementation has no freedom to evolve.

The right approach exposes one operation: `plan_route(start, goal)`, returning a result object with a declared status and a path. The result object is a value — it contains information, not references to internal state. Its fields are defined by the interface contract, not by the implementation. Callers receive a `FOUND` result with a path, an `UNREACHABLE` result with no path, or an `INVALID_INPUT` result when the inputs do not satisfy the preconditions.

Under this design, the route algorithm can change from BFS to Dijkstra, the graph can change from an adjacency list to a different representation, the internal cost model can be revised — none of these affect any caller. What callers depend on is the contract, and the contract has not changed.

28.4. An Interface in Code

```
class Route_Response
create
  make(status, path: String) do
    this.status := status
    this.path := path
  end
feature
  status: String
  path: String
invariant
  valid_status:
    status = "FOUND" or
    status = "UNREACHABLE" or
    status = "INVALID_INPUT"
end

class Route_Interface
feature
  plan_route(start_loc, goal_loc: String): Route_Response
    require
      inputs_present: start_loc /= "" and goal_loc /= ""
    do
      if start_loc = goal_loc then
        result :=
         create Route_Response.make("FOUND", start_loc)
      elseif start_loc = "A" and goal_loc = "C" then
        result :=
         create Route_Response.make("FOUND", "A->B->C")
      else
        result :=
         create Route_Response.make("UNREACHABLE", "")
      end
    ensure
      declared_status:
        result.status = "FOUND" or
        result.status = "UNREACHABLE" or
        result.status = "INVALID_INPUT"
    end
end

class Route_Client
create
  make(route_api: Route_Interface) do
    this.route_api := route_api
  end
feature
  route_api: Route_Interface

  request(start_loc, goal_loc: String): String
    require
      inputs_present: start_loc /= ""
                      and goal_loc /= ""
    do
      let resp: Route_Response
        := route_api.plan_route(start_loc, goal_loc)
      if resp.status = "FOUND" then
        result := "OK:" + resp.path
      elseif resp.status = "UNREACHABLE" then
        result := "NO_ROUTE"
      else
        result := "BAD_INPUT"
      end
    ensure
```

```
      known_result: result /= ""
    end
end
```

`Route_Response` carries an invariant that constrains the status to one of three declared values. No code can construct a `Route_Response` with an undeclared status — the constraint is structural, not advisory. `Route_Interface.plan_route` has a precondition that rejects empty identifiers and a postcondition that asserts the output status is always one of the three declared values. Together, these define the complete contract.

`Route_Client` depends on nothing about `Route_Interface` except its contract. It does not know whether the route is computed by BFS or Dijkstra, whether the graph is stored as an adjacency list or a matrix, or whether routes are cached. It knows the three possible statuses and what to do with each. This is the definition of a stable caller: one whose behavior is determined by the interface contract and nothing else.

The postcondition on `Route_Client.request` — `result /= ""` — is weaker than the postconditions on the interface itself. This is correct: `request` is a consumer of the interface contract, and its own contract is a consequence of what the interface guarantees. A stronger postcondition that asserted specific result strings would couple `request` to the exact status values of the interface, reducing the interface's ability to change its status vocabulary in the future.

28.5. Interfaces in the Three Systems

In the delivery system, the route planning interface is consumed by the dispatch coordinator, the monitoring dashboard, and the route simulation tool. All three callers need a path and a status; none of them need to know how routes are computed. The interface's contract — `FOUND`, `UNREACHABLE`, `INVALID_INPUT` — is stable across all consumers. When the route algorithm is upgraded, no consumer changes.

In the knowledge engine, the retrieval interface is consumed by the ranking component, the debug inspector, and the A/B testing harness. The retrieval interface returns a list of candidate documents with their scores and confidence levels. The contract includes miss semantics — a query that matches no documents returns an explicit empty result, not null — and a bound on the candidate set size. All three consumers can depend on these properties without knowing how retrieval is implemented.

In the virtual world, the state query interface is consumed by the collision detector, the event logger, and the external visualization client. The interface exposes entity state through a defined result type with declared fields. Internal state representation — how positions are stored, how interaction rules are indexed — is hidden behind the interface. The visualization client receives what it needs; the collision detector receives what it needs; neither knows about the other's requirements or the implementation's structure.

In all three systems, the interface is the line at which internal freedom is traded for external stability. Behind the interface, the implementation can change. In front of it, callers can depend on a promise that holds.

28.6. Four Ways Interface Design Fails

Leaking internals. An interface that returns references to internal data structures, exposes mutable state, or requires callers to manipulate internal representations couples every caller to the implementation. The implementation cannot change without breaking every caller, because every caller depends not just on the contract but on the structure it was built to navigate. The remedy is to return value objects — data structures defined by the interface contract, not by the implementation — and to ensure that callers receive information, not handles to internal state.

Ambiguous failure modes. An interface that uses the same return value for success with empty results and for failure requires every caller to infer which situation applies from context. Different callers will infer differently, and one of them will be wrong. Every distinct failure condition deserves a distinct status code. The cost of defining one more status value is minimal; the cost of callers misinterpreting a shared one is unbounded.

Over-wide interfaces. An interface with many operations accumulates commitments. Each operation must be maintained, documented, and kept consistent with all the others. As the interface grows, the set of behaviors it exposes becomes harder to understand, harder to test, and harder to evolve without introducing inconsistencies. The minimum viable interface — the smallest set of operations that serves all legitimate callers — is almost always the right starting point. Operations can be added; removing them is a breaking change.

Versioning by breakage. An interface that changes its existing contract — redefining what a status code means, removing a field from a response object, tightening a precondition — breaks every caller simultaneously. Breakage is sometimes necessary, but it must be a deliberate decision with a migration plan, not a consequence of an implementation change that happened to affect the contract. The discipline of additive evolution — adding new operations and new status values rather than modifying existing ones — preserves the contract for existing callers while extending it for new ones.

28.7. Quick Exercise

Choose one interface in your system that is consumed by more than one caller and document its complete contract with five parts: the input preconditions for its primary operation, the output guarantee for the success case, the failure semantics for each distinct failure mode, one cross-call invariant that callers depend on, and one internal detail that the interface must not expose.

Then write one integration test that validates the contract without inspecting the implementation. The test should remain valid across any internal change that does not alter the contract. If the test would break when the internal algorithm changes, it is testing the implementation rather than the interface.

28.8. Takeaways

- An interface is a promise: what the component requires, what it guarantees, and what it reports when it cannot deliver. The quality of the promise determines the stability of every caller.
- A good interface is minimal, explicit, stable, and testable. Weakness in any one of these properties will eventually produce integration failures.
- The complete contract of an interface operation has four parts: input preconditions, output guarantees, failure semantics, and cross-call invariants. An interface with any of these missing is an interface with implicit assumptions that callers will discover in production.
- Leaking internals, ambiguous failure modes, over-wide interfaces, and versioning by breakage are the four failure modes of interface design. All four are preventable by contract-first thinking.
- The line between what is behind the interface and what is in front of it is the line between internal freedom and external stability. Stable interfaces are what make internal evolution possible.

Part VI has now assembled the complete toolkit for building real software: components, functional thinking, object-oriented responsibility design, and *stable interfaces. These are the tools that connect the algorithmic correctness of Parts III and V to the architectural quality of the systems that algorithms ultimately inhabit.*

Studio 4 — The Architecture Refactor

Subtitle

Evolving Studio 3 optimizations into stable component architecture.

Studio Focus This studio chapter is hands-on: implement, verify behavior, and document tradeoffs as you iterate.

The Situation

Studio 3 improved performance by adding indexes and targeted access paths over the Studio 2 model classes.

Now a new problem appears: architecture fragility.

Typical symptoms:

- orchestration logic mixed with storage details
- duplicate workflow rules across modules
- direct dependency on concrete classes everywhere
- difficult integration testing due to tight coupling

Studio 4 is the natural next step: keep Studio 3 behavior gains, but move to explicit architectural boundaries.

Engineering Brief

Refactor around components, ports, and contracts without changing externally visible behavior.

Required outcomes:

- preserve Studio 3 semantics and performance intent
- define explicit component boundaries
- introduce interface-style ports for collaboration
- isolate domain logic from infrastructure details
- keep dependencies pointing toward domain intent

Implementation guidance:

- start from existing V3 code paths
- create seams first, then move logic behind seams
- prove behavior parity with before/after run checks

Implementation In Nex

Suggested files:

- `architecture_ports.nex`
- `delivery_arch_refactor.nex`
- `knowledge_arch_refactor.nex`
- `world_arch_refactor.nex`
- `studio_4_main.nex`

If using the web IDE, place all classes in one file and run `App.run`.

Shared Ports (Interfaces By Contract)

These ports abstract over concrete V3 storage/index implementations from Studio 3.

```
class Task_Query_Port
feature
  status_of(task_id: String): String
    require
      id_present: task_id /= ""
    do
      result := "NOT_IMPLEMENTED"
    ensure
      non_empty: result /= ""
    end
end

class Doc_Query_Port
feature
  has_doc(doc_id: String): Boolean
    require
      id_present: doc_id /= ""
    do
      result := false
    ensure
      bool_result: result = true or result = false
    end
end

class World_Update_Port
feature
  move_by_id(object_id: String, delta: Integer): String
    require
      id_present: object_id /= ""
    do
      result := "NOT_IMPLEMENTED"
    ensure
      non_empty: result /= ""
    end
end
```

Delivery Refactor (Using Studio 3 V3 Store)

Before (Studio 3): callers directly used `Delivery_Task_Store_V3`.

After (Studio 4): callers depend on `Task_Query_Port`; concrete store is adapted behind it.

```
class Delivery_Task_Store_V3
create
  make(k1: String, v1: String, k2: String, v2: String) do
    this.k1 := k1
    this.v1 := v1
    this.k2 := k2
    this.v2 := v2
  end
feature
  k1: String
  v1: String
  k2: String
  v2: String

  find_status(task_id: String): String
    require
      id_present: task_id /= ""
    do
      if task_id = k1 then
        result := v1
      elseif task_id = k2 then
        result := v2
      else
        result := "NOT_FOUND"
      end
    ensure
      non_empty: result /= ""
    end
end

class Delivery_Task_Adapter
create
  make(store: Delivery_Task_Store_V3) do
    this.store := store
  end
feature
  store: Delivery_Task_Store_V3

  status_of(task_id: String): String
    require
      id_present: task_id /= ""
    do
      result := store.find_status(task_id)
    ensure
      non_empty: result /= ""
    end
end

class Delivery_Workflow
create
  make(tasks: Delivery_Task_Adapter) do
    this.tasks := tasks
  end
feature
  tasks: Delivery_Task_Adapter

  dispatch_view(task_id: String): String
    require
```

```
      id_present: task_id /= ""
    do
      let st: String := tasks.status_of(task_id)
      if st = "NOT_FOUND" then
        result := "UNKNOWN_TASK"
      else
        result := "TASK_STATUS:" + st
      end
    ensure
      non_empty: result /= ""
    end
end
```

Knowledge Refactor (Port + Validator Service)

```
class Doc_Index_V3
create
  make(dk1: String, dk2: String, dk3: String) do
    this.dk1 := dk1
    this.dk2 := dk2
    this.dk3 := dk3
  end
feature
  dk1: String
  dk2: String
  dk3: String

  has_doc(doc_id: String): Boolean
    require
      id_present: doc_id /= ""
    do
      result := doc_id = dk1 or doc_id = dk2
                or doc_id = dk3
    ensure
      bool_result: result = true or result = false
    end
end

class Doc_Query_Adapter
create
  make(idx: Doc_Index_V3) do
    this.idx := idx
  end
feature
  idx: Doc_Index_V3

  has_doc(doc_id: String): Boolean
    require
      id_present: doc_id /= ""
    do
      result := idx.has_doc(doc_id)
    ensure
      bool_result: result = true or result = false
    end
end

class Link_Validation_Service
create
  make(docs: Doc_Query_Adapter) do
    this.docs := docs
  end
feature
```

```
  docs: Doc_Query_Adapter

  validate(from_id, to_id: String): String
    require
      ids_present: from_id /= "" and to_id /= ""
    do
      if not docs.has_doc(from_id) then
        result := "MISSING_FROM"
      elseif not docs.has_doc(to_id) then
        result := "MISSING_TO"
      else
        result := "VALID"
      end
    ensure
      known_result:
        result = "VALID" or
        result = "MISSING_FROM" or
        result = "MISSING_TO"
    end
end
```

World Refactor (Port + Use Case Service)

```
class World_Model_V3
create
  make(wk1: String, wx1: Integer) do
    this.wk1 := wk1
    this.wx1 := wx1
  end
feature
  wk1: String
  wx1: Integer

  move_by_id(object_id: String, delta: Integer): String
    require
      id_present: object_id /= ""
    do
      if object_id = wk1 then
        wx1 := wx1 + delta
        result := "UPDATED"
      else
        result := "NOT_FOUND"
      end
    ensure
      non_empty: result /= ""
    end
end

class World_Update_Adapter
create
  make(model: World_Model_V3) do
    this.model := model
  end
feature
  model: World_Model_V3

  move_by_id(object_id: String, delta: Integer): String
    require
      id_present: object_id /= ""
    do
      result := model.move_by_id(object_id, delta)
    ensure
```

```
      non_empty: result /= ""
    end
end

class World_Step_Service
create
  make(updater: World_Update_Adapter) do
    this.updater := updater
  end
feature
  updater: World_Update_Adapter

  apply_player_input(object_id: String,
                     delta: Integer): String
    require
      id_present: object_id /= ""
    do
      result := updater.move_by_id(object_id, delta)
    ensure
      known_result: result = "UPDATED"
                    or result = "NOT_FOUND"
    end
end
```

Studio Driver (Behavior Parity Check)

```
class App
feature
  run() do
    -- Delivery setup
    let ds: Delivery_Task_Store_V3
      := create Delivery_Task_Store_V3.make(
           "T-1", "PENDING", "T-2", "IN_TRANSIT"
         )
    let da: Delivery_Task_Adapter
      := create Delivery_Task_Adapter.make(ds)
    let dw: Delivery_Workflow :=
      create Delivery_Workflow.make(da)

    print(dw.dispatch_view("T-2"))   -- TASK_STATUS:IN_TRANSIT
    print(dw.dispatch_view("T-99"))  -- UNKNOWN_TASK

    -- Knowledge setup
    let di: Doc_Index_V3
      := create Doc_Index_V3.make("D-1", "D-2", "D-3")
    let dqa: Doc_Query_Adapter :=
      create Doc_Query_Adapter.make(di)
    let lvs: Link_Validation_Service
      := create Link_Validation_Service.make(dqa)

    print(lvs.validate("D-1", "D-3"))   -- VALID
    print(lvs.validate("D-9", "D-3"))   -- MISSING_FROM

    -- World setup
    let wm: World_Model_V3 :=
      create World_Model_V3.make("E-1", 10)
    let wua: World_Update_Adapter
      := create World_Update_Adapter.make(wm)
    let wss: World_Step_Service :=
      create World_Step_Service.make(wua)

    print(wss.apply_player_input("E-1", 2))  -- UPDATED
```

```
    print(wss.apply_player_input("E-9", 2))  -- NOT_FOUND
  end
end
```

Expected outcome:

- same core behavior as Studio 3 refactored paths
- clearer component boundaries
- easier component-level testing due to port seams

Studio Challenges

Level 1 — One Architectural Seam

- pick one Studio 3 optimized path
- introduce a port + adapter boundary
- prove behavior parity on nominal and edge cases

Level 2 — Three-Domain Architecture Pass

- apply port/adapter + service orchestration pattern to delivery, knowledge, and world
- remove at least one direct concrete dependency per domain
- document dependency direction before/after

Level 3 — Alternative Architecture Experiment

- compare two styles:
  - port/adapter with thin services
  - feature modules with internal adapters
- evaluate maintainability and migration cost

Postmortem

Discuss with evidence:

- Which seam reduced coupling the most?
- Which abstraction clarified behavior, and which added unnecessary ceremony?
- Which components are still doing too much?
- What is the next architecture risk if the system doubles again?

Deliverables

- architecture diagram with components and dependency direction
- refactored Nex code using ports/adapters over Studio 3 V3 logic
- behavior-parity run log (before/after outputs)
- short ADR-style note with chosen architecture and rejected alternative

Exit Criteria

You are ready for Part VII if:

- key use cases depend on ports, not concrete storage classes
- orchestration logic is separated from storage/index details
- cross-component contracts are explicit and testable
- behavior parity with Studio 3 is demonstrated

Part VII.

Part VII — Making Software Trustworthy

29. Preconditions and Postconditions

Parts I through VI gave us the tools to build software that is correct by design: clear problem statements, explicit models, well-chosen algorithms, organized data structures, and components with stable interfaces. Part VII asks a harder question. Given software that was built with care, how do we make its behavior dependable as it changes — as requirements evolve, as new developers join the team, as algorithms are replaced and data models are extended? The answer is to make the software's assumptions explicit and verifiable at every boundary where those assumptions matter.

Every routine in a system makes two kinds of implicit commitments. It assumes certain things about the world when it is called — that its inputs satisfy certain constraints, that the system is in a certain state. And it promises certain things about the world when it returns — that its output has a certain shape, that the system has been left in a certain state. When these assumptions and promises are implicit, they are invisible to callers, unenforceable by the system, and discoverable only through failure. When they are explicit, they are readable, checkable, and — when violated — diagnostic: the failure occurs at the violated boundary, not at some distant point where the corrupted state finally produces a visible symptom.

A precondition is the formal statement of an assumption: what must be true of the inputs and the current state before this routine may be called. A postcondition is the formal statement of a promise: what will be true of the outputs and the resulting state when this routine returns, given that the precondition was satisfied.

29.1. Preconditions: Caller Obligations

A precondition is a statement about the caller's responsibility. It says: *I, this routine, am only obligated to deliver my postcondition if you, the caller, have satisfied this condition before calling me.* If the caller violates a precondition, the routine's behavior is unspecified — it may signal an error, it may produce a wrong result, or it may do something else entirely. The caller has not held up their end of the contract.

This has a practical consequence: preconditions should not be written to handle the caller's mistakes gracefully. A precondition is not a validation layer that converts bad inputs into error results. It is a boundary that says: inputs satisfying these conditions are valid inputs for this routine; inputs that do not satisfy them are not. A routine that promises `ASSIGNED` on valid inputs and `BAD_INPUT` on invalid ones has not written a precondition — it has written a conditional that handles both cases and made both cases part of the postcondition. That is a legitimate design, but it is different from a contract that rejects invalid inputs with a contract violation.

What belongs in a precondition? Three categories cover most cases.

Input presence and validity. Empty identifiers, null references, and values outside a defined range are inputs that no routine can process meaningfully. Stating the required form explicitly — `task_id /= ""`, `priority >= 0` — tells the caller what they must guarantee before the call.

Required state. Many operations are only legal from certain states. A task can only be assigned if its status is `PENDING` or `FAILED`. A robot can only be dispatched if it is marked ready. These conditions belong in the precondition, not in a conditional inside the routine body, because they are the caller's responsibility to ensure before calling.

Relationship constraints. Some operations require that two or more inputs stand in a certain relationship — that a start location and a destination both exist in the same graph, that an update refers to an entity that has been previously registered. When these relationships are required, stating them as preconditions makes the requirement visible and checkable.

29.2. Postconditions: Routine Guarantees

A postcondition is a statement about the routine's responsibility. It says: *if you, the caller, have satisfied the precondition, then I, this routine, guarantee the following when I return.* The postcondition is an unconditional promise, relative to the precondition — no matter what path through the routine body execution takes, if the precondition was satisfied on entry, the postcondition will be satisfied on exit.

This unconditional quality is what makes postconditions valuable for reasoning. A caller that knows the precondition and postcondition of a routine can reason about what comes after the call without reading the routine's implementation. The postcondition is the contract the implementation must fulfill, and the implementation can be changed freely as long as it continues to fulfill it.

Three categories cover most postconditions.

Result constraints. The shape, value, and meaning of the returned value. A route operation's result is always one of three declared status values. A score is always non-negative. A retrieved document is always from the collection that was queried. Stating these constraints ensures that callers can depend on them rather than defensively checking them.

State transition outcomes. When a routine modifies state, the postcondition records what changed and what the new state satisfies. An assignment operation leaves the task in `IN_TRANSIT` with the specified robot identifier. A blocking operation leaves the path with status `BLOCKED`. These assertions close the loop between the operation's intent and its result.

Derivable consequences. Some postconditions follow from the combination of the precondition and the routine's action. If a precondition requires non-negative edge costs and the routine computes a sum of edge costs, the postcondition can assert that the result is non-negative — not because the routine checks it explicitly, but because it follows from the combination of the precondition and the computation. Stating derivable consequences explicitly makes them checkable and makes the chain of reasoning visible.

29.3. From Requirement to Contract

Consider the requirement:

> *"Assign a task to a robot and return the assignment status."*

Step 1: Define caller obligations. The task identifier must be present and non-empty — a call with an empty identifier cannot refer to any task. The robot identifier must be present — an assignment requires a robot to assign to. The task's current status must be `PENDING` or `FAILED` — assigning a task that is already `IN_TRANSIT` or `DELIVERED` is an illegal transition. These three conditions are the precondition.

Step 2: Define routine guarantees. If the precondition is satisfied, the routine will: set `assigned_robot_id` to the provided robot identifier, set `status` to `IN_TRANSIT`, and return `"ASSIGNED"`. The postcondition asserts all three: `assigned_robot_id = robot_id`, `status = "IN_TRANSIT"`, and `result = "ASSIGNED"`.

Step 3: Define fail-fast behavior. A call that violates the precondition should fail immediately with a clear signal — a contract violation — rather than silently producing a wrong result that propagates through the system. The earlier a violation is detected, the closer it is to the code that caused it, and the easier it is to diagnose.

Step 4: Verify the contract is complete. A contract is complete when the precondition and postcondition together fully specify the routine's behavior for all valid inputs. A contract with a precondition that permits inputs the postcondition cannot deliver is incomplete. A contract with a postcondition that leaves the new state underspecified is incomplete. Completeness is the test.

29.4. A Contract in Code

```
class Delivery_Task
feature
  task_id: String
  status: String
  assigned_robot_id: String

  assign(robot_id: String): String
    require
      task_id_present: task_id /= ""
      robot_id_present: robot_id /= ""
      can_assign: status = "PENDING" or status = "FAILED"
    do
      assigned_robot_id := robot_id
      status := "IN_TRANSIT"
      result := "ASSIGNED"
    ensure
      assigned: assigned_robot_id = robot_id
      now_in_transit: status = "IN_TRANSIT"
      known_result: result = "ASSIGNED"
    end
invariant
  valid_status:
    status = "PENDING" or
    status = "IN_TRANSIT" or
    status = "DELIVERED" or
    status = "FAILED"
end
```

Read the structure in layers. The invariant asserts a property that must hold at all times — after construction, after every operation, and before every operation call. The precondition on `assign` adds requirements specific to this operation: the task identifier and robot identifier must be non-empty, and the status must be one that permits assignment. The postcondition asserts what the operation has accomplished: the assignment is recorded, the status has been updated, and the return value carries the expected token.

The three preconditions address different responsibilities. `task_id_present` is an integrity check on the task itself — a task without an identifier is malformed and should never have been constructed, which is why the invariant also requires `task_id /= ""`. Its presence in the precondition is a belt-and-suspenders assertion that guards against callers operating on malformed instances. `robot_id_present` is a caller obligation — the caller must provide a real robot. `can_assign` is a state precondition — the caller must ensure the task is in a state that permits assignment before calling.

The postcondition closes the loop: every claim made in the `do` block has a corresponding assertion in the `ensure` block. The assignment is checked, the status is checked, the result is checked. A future refactor that modifies the `do` block must satisfy all three assertions, which means the postcondition is both documentation and test specification.

29.5. Contracts in the Three Systems

In the delivery system, every state transition on `DeliveryTask` — assign, start, complete, fail — has a contract. The preconditions encode the legal transition graph from Chapter 10: which states permit which operations. The postconditions encode the resulting state and return value. A developer modifying the transition logic can see immediately what the contract requires and whether their change satisfies it.

In the knowledge engine, the route computation operation has a precondition that requires non-empty inputs and a postcondition that guarantees one of three declared statuses. The scoring operation has a precondition that requires non-empty document and query strings and a postcondition that guarantees a non-negative score. These contracts make the pipeline from Chapter 12's decomposition independently verifiable at each stage.

In the virtual world, the state update operation for each tick has a precondition that requires the entity to exist in the current frame and a postcondition that guarantees the entity's state satisfies the world's invariants after the update. Violations of the postcondition are caught at the boundary of the update, before the corrupted state propagates to collision detection or event output.

In all three systems, contracts turn the informal expectations that developers carry in their heads — "this must be non-empty," "this transition is only legal from PENDING" — into executable statements that are checked automatically and fail close to their source.

29.6. Three Ways Contract Design Fails

Missing preconditions for state transitions. A transition operation without a precondition on the current state permits callers to invoke it from any state. A delivered task can be reassigned. A robot can be dispatched before it is ready. The illegal transition is not detected at the call site — it is detected later, when downstream code encounters a state it was not designed to handle. The remedy is to encode transition legality as a precondition on every transition operation. The precondition fires at the call, not at the downstream consequence.

Weak postconditions. A postcondition that asserts only that the result is non-empty, or that the return value is a string, is a postcondition that provides almost no guarantee. A caller that depends only on these weak properties cannot reason about what the operation actually accomplished. The postcondition should assert the specific state changes that matter — the new status, the recorded assignment, the returned path — so that any implementation that fails to accomplish them will violate the postcondition and fail immediately.

Contract drift between similar operations. A codebase with many similar operations — multiple transition operations, multiple search operations, multiple update operations — will accumulate contracts that began as consistent and gradually diverged through independent modifications. One transition requires non-empty task id; another was written later and omits it. One search guarantees `NOT_FOUND` on miss; another returns an empty string. The inconsistency is invisible until a caller that works correctly with one operation fails with another. The remedy is to review contracts across similar operations periodically and to establish conventions — all transitions require non-empty identifiers, all searches declare an explicit miss status — that future contracts are written to match.

29.7. Quick Exercise

Choose one routine in your system — one that is called from multiple places and whose behavior callers depend on — and write its complete contract with four parts: three preconditions that state the caller's obligations, three postconditions that state the routine's guarantees, one input that violates a precondition and the behavior that should result, and one valid input with the specific postconditions it should satisfy.

Then verify that the postconditions still hold after one plausible refactor of the routine body. If a refactor that improves the implementation without changing the observable behavior would break a postcondition, the postcondition is specifying implementation rather than guarantees.

29.8. Takeaways

- A precondition is the caller's obligation. If the caller does not satisfy it, the routine's behavior is unspecified. Preconditions should fail fast, not handle invalid inputs gracefully.
- A postcondition is the routine's guarantee. If the precondition is satisfied, the postcondition holds — unconditionally, regardless of which path through the routine body execution takes.
- Three categories belong in preconditions: input presence and validity, required state, and relationship constraints. Three categories belong in postconditions: result constraints, state transition outcomes, and derivable consequences.
- A contract is complete when the precondition and postcondition together fully specify the routine's behavior for all valid inputs.
- Contracts turn implicit assumptions into explicit, executable statements that fail close to their source when violated.

Chapter 28 extends the contract discipline from individual routines to entire objects. Where postconditions specify what is true after a single operation, invariants specify what is true at all times — before every call and after every return. Invariants are the class-level counterpart of postconditions, and they are what make it possible to reason about an object's state without reading every operation that might have modified it.

30. Invariants — Rules That Must Never Break

Chapter 27 introduced contracts at the level of individual operations: preconditions that state what must be true before a routine is called, and postconditions that state what will be true when it returns. These are powerful tools for reasoning about individual calls. They do not, by themselves, answer a broader question: what is always true about an object, regardless of which sequence of operations has been applied to it, and regardless of what state it was in before any particular call?

This is the question invariants answer. An invariant is a property that holds for every observable state of an object — after construction, before every operation call, and after every operation return. It is not a claim about one call or one transition; it is a claim about the object's entire lifetime. Where a postcondition says "after this operation, the status is `IN_TRANSIT`," an invariant says "at all times, the status is one of four declared values." The postcondition is a consequence of a specific operation; the invariant is a constraint on the entire state space.

30.1. What an Invariant Is

The simplest way to understand the force of an invariant is to consider what it rules out. A `Delivery_Task` with status `"PENDING"` and a non-empty `assigned_robot_id` is a state that the model never intends to permit: a pending task is one that has not yet been assigned. A `WorldObject` at position `x = -5` when the world begins at zero is a state that the physics never intends to permit. A `Document` with an empty identifier in a collection keyed by identifiers is a state that the index never intends to permit. Each of these is an invalid state, and each will eventually cause a failure — not at the point where the invalid state was created, but later, when downstream code encounters it and discovers that its own assumptions have been violated.

An invariant is what prevents invalid states from being created. It is a constraint that every operation on the object must preserve. An operation that establishes an invalid state — that leaves the object in a configuration the invariant forbids — violates the invariant, and the violation can be detected immediately rather than at the distant point where the invalid state produces a symptom.

The relationship between invariants and contracts is additive: invariants operate at the class level, contracts operate at the operation level, and together they provide complementary coverage. A contract without an invariant can ensure that each operation leaves the object in a correct state given the state it was in on entry, but cannot prevent the object from being constructed in an invalid state. An invariant without contracts can ensure the object's state is always valid but cannot specify the relationship between inputs and outputs for any particular operation. Both are necessary.

30.2. Three Categories of Invariant

Invariants fall into three categories, corresponding to different levels of the object's structure.

Field-level invariants constrain individual fields to valid domains. A task identifier must be non-empty. A position coordinate must be non-negative. A priority score must be within a defined range. These are the simplest invariants — they constrain one field in isolation — and they are also the most common source of errors when omitted. A system that permits empty identifiers will eventually call an operation that assumes identifiers are present, and the failure will be attributed to the operation rather than to the missing constraint on construction.

Relationship invariants constrain the relationship between two or more fields. A task in `IN_TRANSIT` status must have a non-empty `assigned_robot_id` — the transition to `IN_TRANSIT` is only meaningful when there is a robot to perform it, and the system should never be in a state where a task is claimed to be in transit without an assigned robot. A world object at `max_x` must have the appropriate boundary condition applied. These invariants are more powerful than field-level ones because they capture the dependencies between fields that individual field constraints cannot express.

Range and bound invariants constrain fields to stay within operational limits. A position must remain between zero and the world's maximum extent. A score must remain non-negative. A collection must not exceed its defined capacity. These invariants are particularly important for systems where values are computed dynamically — where positions are updated by velocity, scores are computed by summing term weights, priorities are modified by rules — because the computation may produce values outside the valid range without the error being visible at the computation site.

30.3. Invariants and Transitions

A common confusion in invariant design is between invariants and transition preconditions, and the confusion produces both invariants that are too weak and invariants that are too strong.

An invariant that is too weak fails to prevent invalid states. Consider a `DeliveryTask` whose invariant only constrains the status to one of four values but does not constrain the relationship between status and `assigned_robot_id`. The invariant permits a task in `IN_TRANSIT` with an empty `assigned_robot_id` — an invalid state that no operation should produce but that the invariant does not forbid. Strengthening the invariant to require `assigned_robot_id /= ""` whenever `status = "IN_TRANSIT"` closes this gap.

An invariant that is too strict forbids states that are legitimately intermediate during a transition. If the invariant required `assigned_robot_id /= ""` at all times, no object could be constructed before an assignment has been made. The invariant must permit the initial `PENDING` state without an assigned robot while forbidding the `IN_TRANSIT` state without one. The relationship invariant — `status = "IN_TRANSIT" implies assigned_robot_id /= ""` — achieves this precision.

The principle is: invariants should constrain the outcomes of transitions, not the inputs to them. Inputs to transitions belong in preconditions. The states that are permanently forbidden belong in invariants.

30.4. From Requirement to Invariant Design

Consider the requirement:

> *"World objects move each tick but must always stay within legal bounds."*

Step 1: Identify the always-true rules. Every world object must have a non-empty identifier — the identity constraint. Every world object must have a non-negative `max_x` — a malformed bound is worse than no bound. Every world object's position must satisfy `x >= 0 and x <= max_x` — the spatial constraint that the requirement directly states. These are invariants: they hold before every tick and must hold after every tick.

Step 2: Keep transition behavior flexible. The `step` operation updates position by adding velocity. The invariant must constrain the resulting position without forbidding the operation. The correct design is to clamp the position within bounds in the operation body and to assert the spatial invariant in both the postcondition and the class invariant. The invariant does not say "velocity must be zero" — that would block movement. It says "after movement, position is within bounds" — which the operation must guarantee by clamping.

Step 3: Confirm that every operation preserves the invariant. For `World_Object`, the only operation that modifies position is `step`. The postcondition on `step` asserts `x >= 0 and x <= max_x`. Since this assertion matches the invariant's spatial constraint, any execution of `step` that satisfies its postcondition will leave the invariant intact. The two assertions reinforce each other: the postcondition ensures the specific operation preserves the invariant, and the invariant ensures no other operation — including direct field assignment — can violate the constraint.

30.5. An Invariant in Code

```
class World_Object
feature
  object_id: String
  x: Integer
  vx: Integer
  max_x: Integer
  step()
    require
      max_valid: max_x >= 0
    do
      let next: Integer := x + vx
      if next < 0 then
        x := 0
      elseif next > max_x then
        x := max_x
      else
        x := next
      end
    ensure
      bounded_after_step: x >= 0 and x <= max_x
    end
invariant
  id_present: object_id /= ""
  max_non_negative: max_x >= 0
  x_bounded: x >= 0 and x <= max_x
end
```

The three invariants cover all three categories. `id_present` is a field-level invariant — an object without an identifier is a malformed object. `max_non_negative` is a range invariant — a negative bound is not a legal bound. `x_bounded` is the relationship invariant between position and bound — the spatial constraint that the requirement states must always hold.

The interaction between the invariant and the postcondition on `step` is the central design relationship in this sketch. The invariant asserts `x_bounded` as a permanent truth. The postcondition on `step` asserts `bounded_after_step`, which is the same condition. This is intentional: the postcondition is the per-call evidence that the invariant is preserved across the `step` operation. If a future modification to `step` removed the clamping logic and the postcondition, the invariant would immediately catch any execution that left `x` out of bounds.

The precondition on `step` — `max_valid: max_x >= 0` — might look redundant given the class invariant `max_non_negative`. It is not redundant for the same reason the `task_id_present` precondition in Chapter 27 was not redundant: the invariant guarantees the condition for any correctly constructed object; the precondition is a defensive assertion that makes the assumption visible in the operation's own contract. As the class evolves and `step` is read in isolation, the precondition communicates the operation's requirements without requiring the reader to consult the class invariant.

30.6. Invariants in the Three Systems

In the delivery system, the `DeliveryTask` invariant asserts that the status is always one of four declared values and that any task in `IN_TRANSIT` has a non-empty `assigned_robot_id`. The first invariant is a field-level constraint; the second is a relationship constraint. Together they ensure that no operation can leave the task in an ambiguous or unrepresentable state.

In the knowledge engine, the `Document` invariant asserts that the document identifier is non-empty and unique within its scope. The uniqueness invariant is a relationship constraint — it relates one document instance to the collection that contains it — and it must be enforced at both construction and update. A document whose identifier changes to match another document's identifier has violated the invariant, and the violation must be caught at the point of the change.

In the virtual world, every `WorldObject` carries a spatial invariant: position coordinates are within bounds. Every `InteractionRule` carries a validity invariant: the object types it references must exist in the type registry. These invariants ensure that collision detection and interaction application can trust the data they receive, rather than defensively validating every piece of state they encounter.

In all three systems, invariants are not defensive checks added to make buggy code safer. They are assertions about the system's design: states that the model never intends to permit, properties that every correct operation must preserve, constraints that exist because the system's reasoning depends on them.

30.7. Three Ways Invariant Design Fails

Invariants too weak. An invariant that constrains individual fields but not their relationships permits states that the model never intends but cannot prevent. A `DeliveryTask` invariant that validates the status values but not the `assigned_robot_id` constraint permits in-transit tasks with no assigned robot — an invalid state that no invariant will catch and that subsequent operations will discover in the form of empty identifiers where non-empty ones were expected. The remedy is to strengthen the invariant to capture all of the relationships between fields that must hold for the object to be in a valid state.

Invariants too strict. An invariant that forbids states that are legitimate at certain points in an object's lifecycle will make normal operations impossible. If the spatial invariant required `vx = 0` — forbidding any non-zero velocity — the `step` operation could not exist. Invariants that are too strict are not corrected by weakening the invariant; they are corrected by identifying which conditions are permanent constraints (invariants) and which are conditions required only before or after specific operations (preconditions and postconditions). Permanent constraints belong in the invariant. Transition-specific conditions belong in operation contracts.

Duplicate rule definitions that drift apart. When the same constraint appears in both a class invariant and the postconditions of multiple operations, the two definitions may diverge over time. One operation's postcondition is updated to reflect a new understanding of the constraint; the invariant is not. Another operation is added with a postcondition that was written to match the updated understanding; the old operations are not updated. The invariant and the operations now describe different constraints, and the system enforces whichever one the checker happens to evaluate. The remedy is to treat the class invariant as the single authoritative statement of each constraint and to write postconditions that reference or derive from the invariant rather than restating it independently.

30.8. Quick Exercise

Choose one class in your system and define its complete invariant with three parts: two field-level invariants that constrain individual fields to their valid domains, one relationship invariant that constrains the relationship between two or more fields, and one operation that could violate one of the invariants if written carelessly.

Then write one test that calls the potentially violating operation with inputs designed to produce a boundary case and verifies that the invariants hold afterward. If the test requires reading the implementation to write, the invariants are not yet strong enough to specify the boundary independently.

30.9. Takeaways

- An invariant is a property that holds for every observable state of an object — not after one operation, but across the entire object's lifetime. It is a constraint on the state space, not a claim about a single transition.
- Invariants and contracts are complementary. Invariants protect against invalid states across all operations; contracts specify the input-output relationship of individual operations. Both are necessary.
- Three categories cover most invariants: field-level constraints on individual fields, relationship invariants on the dependencies between fields, and range or bound invariants on computed values.
- Invariants should constrain the outcomes of transitions, not their inputs. Transition inputs belong in preconditions; permanently forbidden states belong in invariants.
- Duplicate rule definitions drift apart. The class invariant is the authoritative statement of each constraint; operation postconditions should derive from it, not independently restate it.

Chapter 29 examines how testing extends the guarantees established by contracts and invariants. Where contracts specify what must be true, tests provide evidence that the specifications have been met — and they do so by exploring behavior across the full range of inputs, not just the ones the developer thought of first.

31. Testing as Exploration

Contracts make assumptions explicit and promises verifiable. But a contract is a claim, not evidence. The precondition on `assign` asserts that the caller must provide a non-empty robot identifier; the postcondition asserts that the task will be in `IN_TRANSIT` afterward. These claims may be written correctly and violated by the implementation. A routine whose postcondition is stated correctly but whose body computes the wrong result does not fail because of a bad contract — it fails because the contract was never confronted with evidence.

Testing provides that evidence. A test is an experiment: choose inputs, call the routine, observe the outputs, and compare the observations against the expected behavior. When the expected and actual outputs agree, the test provides evidence — not proof, but evidence — that the routine behaves correctly for that input. When they disagree, the test has found a defect. The quality of a test portfolio depends on how well the chosen inputs cover the space of possible failures and how precisely the expected behavior is stated.

The mindset that produces good tests is not confirmation — verifying that the routine works for the inputs you expect to appear. It is exploration — discovering the inputs for which the routine fails to satisfy its contract. Confirmation produces evidence of correctness for the cases you already understood. Exploration discovers the cases you did not.

31.1. Three Kinds of Tests

A test portfolio that covers only one kind of input is a portfolio with gaps. Three kinds of tests together provide substantially more coverage than any one kind alone.

Example tests are tests of concrete, known scenarios. Choose a specific input, state the expected output, call the routine, compare. For `plan_route("A", "C")`, the expected output is `"A->B->C"`. For `plan_route("C", "C")`, the expected output is `"C"`. These tests verify the nominal behavior that the routine was built to provide. They are the foundation of a test portfolio, but they are insufficient alone:

they cover only the cases the developer thought of when writing them, which are typically the cases where the routine works correctly.

Boundary tests probe the edges of the input space — the cases that sit at or near the limits of what the routine accepts. The empty identifier is a boundary. The case where start equals destination is a boundary. The case where no path exists is a boundary. The single-element collection is a boundary. Boundary tests are the most reliable way to discover off-by-one errors, incorrect base case handling, and the implicit assumptions that example tests leave unexplored. The failure of a boundary test is evidence that the routine's behavior at the boundary was never fully defined.

Property-based checks assert truths that should hold across a range of inputs, rather than for a single specific input. If the route planner returns `FOUND`, the path must be non-empty. If the task's `assign` operation returns `"ASSIGNED"`, the task's status must be `IN_TRANSIT`. If the sort operation returns a sequence, no element should appear after any element with a strictly smaller priority. These are properties derivable from the contract, and they should hold for every input satisfying the precondition. Testing them across a range of inputs — generated programmatically, if the language permits, or across a set of representative cases otherwise — provides evidence that the contract holds generally, not just for the specific inputs of the example tests.

31.2. Oracles: How a Test Decides

A test can only be as strong as its oracle — the criterion by which it decides whether the outcome is correct. A weak oracle fails to detect most meaningful defects. A strong oracle detects defects that the weak one would miss.

The weakest oracle is "did not crash." A routine that produces a wrong result without throwing an exception passes this oracle. Nearly every defect of consequence produces a wrong result rather than a crash.

A stronger oracle is a specific expected value: `result = "A->B->C"`. This oracle detects any deviation from the expected output, including wrong paths, extra nodes, and missing nodes. It is strong for the specific input it tests, and it requires that the developer knows what the correct output is before writing the test.

The strongest oracle for a component is its contract. If the postcondition of `plan_route` asserts that the returned status is always one of three declared values and that a `FOUND` result comes with a non-empty path, then a test that verifies these properties is a test that detects any implementation that violates the contract — including implementations that the developer has not specifically imagined. Contract-derived oracles extend naturally to inputs that specific expected-value oracles do not cover.

A test whose oracle is "the result equals what the implementation computed" is not a test at all. It is a transcript of execution. The oracle must be independent of the implementation: it must state what the correct output is, derived from the specification, before the implementation has run.

31.3. From Requirement to Test Portfolio

Consider the requirement:

> *"The route planner should return a valid route or an explicit unreachable status."*

The word "valid" is not self-defining. A valid route is a connected sequence of traversable edges from the start to the destination. The contract from Chapter 27 made this precise: status is `FOUND`, `UNREACHABLE`, or `INVALID_INPUT`; a `FOUND` result's path uses only existing edges; the path's first node is the start and last node is the destination. The test portfolio explores whether the implementation satisfies this contract.

Nominal case. Start is `A`, destination is `C`, a path exists: `A→B→C`. Expected status: `FOUND`. Expected path: `"A->B->C"`. This is the case the routine was primarily designed for.

Trivial boundary. Start equals destination. Expected status: `FOUND`. Expected path: the start location alone. This is the smallest possible "found" case — a route of length zero.

Unknown inputs. Start is `"X"`, destination is `"Z"`, neither exists in the graph. Expected status: `INVALID_INPUT`. This tests the precondition boundary: what happens when the inputs do not satisfy the precondition.

Disconnected graph. Start and destination both exist but are not connected. Expected status: `UNREACHABLE`. This tests the second failure mode.

Property check. For any input where the returned status is `FOUND`, the path must be non-empty and the first node must be the start location. This property should hold for every `FOUND` result, not just for specific inputs.

Together, these five tests cover the three declared statuses, the boundaries between them, and a property that should hold globally. No single test could discover all the defects this portfolio can discover.

31.4. A Test Suite in Code

```
class Route_Planner
feature
  plan(start_loc, end_loc: String): String
    require
      inputs_present: start_loc /= "" and end_loc /= ""
    do
      if start_loc = end_loc then
        result := start_loc
      elseif start_loc = "A" and end_loc = "C" then
        result := "A->B->C"
      else
        result := "UNREACHABLE"
      end
    ensure
      non_empty_result: result /= ""
    end
end

class Route_Planner_Tests
feature
  run_all(): String
    do
      let p: Route_Planner := create Route_Planner
      let t1: String := p.plan("A", "C")
      let t2: String := p.plan("C", "C")
      let t3: String := p.plan("X", "Z")
      if t1 = "A->B->C" and t2 = "C"
         and t3 = "UNREACHABLE" then
        result := "PASS"
      else
        result := "FAIL"
      end
    ensure
      known_result: result = "PASS" or result = "FAIL"
    end
end
```

`Route_Planner_Tests.run_all` is minimal but honest about what it does: it calls the routine with three inputs, compares each result to the specific expected value, and returns a single aggregate verdict. The oracle for each test is a specific expected string — `"A->B->C"`, `"C"`, `"UNREACHABLE"` — derived from the contract rather than from running the implementation.

What this sketch does not show — and what a full test suite for a production route planner would require — is coverage of the `INVALID_INPUT` case and the property-based check that `FOUND` results always have non-empty paths. The three tests here provide evidence for the nominal case and two boundary cases. They do not provide evidence that the contract holds generally. A portfolio that adds the property check and the invalid-input case approaches the coverage that the contract's three-outcome structure warrants.

The postcondition on `run_all` — `result = "PASS" or result = "FAIL"` — is the test framework's own contract: the test runner always returns a declared outcome, never an undefined one. This matters because a test runner that crashes or returns an undeclared value on a test failure provides weaker evidence than one whose own behavior is defined.

31.5. Testing in the Three Systems

In the delivery system, the critical tests are transition legality tests: does `assign` fail correctly when called on a `DELIVERED` task? Does it succeed correctly when called on a `FAILED` task? Does the postcondition hold for every legal transition? These tests exercise the precondition boundaries of the transition operations and verify that illegal transitions are rejected at the boundary rather than silently corrupting state.

In the knowledge engine, the critical tests include determinism — does the same query on the same document collection return the same ranked result on consecutive calls? — and miss semantics — does a query that matches no documents return the declared empty result rather than null, an exception, or an empty list with no status? These are properties that example tests do not cover unless they are specifically designed to cover them.

In the virtual world, the critical tests are bounded update tests: does the state update for any tick leave every entity in a state that satisfies the world's invariants? Does the update operation behave deterministically — the same initial state always producing the same next state? These properties must hold for every tick, not just for ticks with typical inputs.

In all three systems, the test portfolio that matters is not the one that maximizes coverage of code paths. It is the one that maximizes coverage of failure modes — the portfolio designed from the contract outward, not from the implementation inward.

31.6. Three Ways Testing Fails

Happy-path-only testing. A portfolio that tests only the inputs the developer expected to work provides evidence that the routine handles the cases the developer already understood. It provides no evidence about cases the developer did not consider — which is precisely the space where production failures occur. The test portfolio must be designed to find failures, not to demonstrate that expected inputs produce expected outputs.

Fragile assertions. A test that asserts the exact string representation of a complex result, including incidental whitespace and ordering that are not part of the contract, will fail when the formatting changes even if the behavior is correct. A test whose oracle is weaker than the contract — that asserts only that the result is non-empty rather than that it satisfies the contract's properties — will pass when the behavior is wrong. The oracle should assert exactly what the contract guarantees, no more and no less.

Shared assumptions between test and implementation. A test written by the same developer who wrote the implementation, immediately after writing the implementation, will often encode the same assumptions as the implementation. If the implementation misunderstands the requirement, the test will likely misunderstand it the same way. The oracle should be derived from the specification — the contract, the worked examples from Chapter 3, the edge cases from Chapter 4 — not from the implementation the developer just wrote. When possible, tests should be written before the implementation, so the oracle is derived from the requirement rather than from the code.

31.7. Quick Exercise

Choose one routine in your system and design a test portfolio with five tests: two example tests of the nominal behavior, two boundary tests at the edges of the input space, and one property-based check derived from the postcondition that should hold across a range of inputs.

For each test, write the oracle explicitly — the specific property that the output must satisfy, stated without reference to the implementation. Then map each test to the specific failure mode it is designed to detect. If a test is not clearly targeted at a specific failure mode, revise it until it is.

31.8. Takeaways

- Testing is exploration, not confirmation. The goal is to discover failures, not to demonstrate that the cases you already understood work correctly.
- Three kinds of tests cover different parts of the failure space: example tests for nominal behavior, boundary tests for edge cases, and property-based checks for contract-wide guarantees.

- An oracle is how a test decides whether the outcome is correct. The oracle must be independent of the implementation and derived from the specification. A weak oracle misses most meaningful defects.
- The strongest oracles are derived from contracts. A test that verifies the postcondition holds for a range of inputs provides more evidence than a test that verifies a single expected value for a single input.
- A test portfolio designed from the contract outward — asking what inputs would violate the contract — is more reliable than one designed from the implementation inward.

Chapter 30 examines debugging as a hypothesis-driven engineering process. When tests discover a failure, debugging is the discipline of reasoning from the failure to its cause — treating the failed test as an observation that constrains the set of possible explanations, and designing experiments to narrow that set until the cause is identified.

32. Debugging Like an Engineer

Chapter 29 showed how tests discover failures. A failed test is evidence that something is wrong — that some input produces output inconsistent with the contract. Debugging begins where testing ends. Its task is to answer two questions: why does the failure occur, and how can it be corrected without introducing new failures?

These questions admit engineering answers. Debugging is not a search through a codebase for suspicious lines. It is a process of scientific reasoning: observe a failure, form a hypothesis about its cause, design a check that would distinguish the hypothesis from alternatives, execute the check, and revise the hypothesis based on the result. The process is iterative, but each iteration is directed rather than exploratory — each step produces information that narrows the space of possible causes.

The alternative to this discipline is not faster debugging. It is debugging that terminates without certainty — fixes that silence symptoms without addressing causes, changes that introduce new failures while correcting old ones, and a growing accumulation of "fixed" bugs that return in slightly different forms.

32.1. The Six-Step Process

A reliable debugging process has six steps, and each one serves a purpose that the subsequent steps depend on.

Step 1: Reproduce reliably. A failure that cannot be reproduced cannot be diagnosed with confidence. If the failure appears intermittently — only under certain timing conditions, only with certain data configurations, only in certain environments — then any fix applied to an unreproduced failure is a guess. Reproducing the failure means: given a specific set of inputs and a specific system state, the failure occurs every time. Minimal reproduction means: finding the smallest input or state that still produces the failure, removing everything that is not necessary. Minimal cases make step 3 easier by narrowing the search space before hypothesis formation begins.

Step 2: Localize the failure boundary. The failure is observed at some point in the system — a test assertion fails, a user reports incorrect output, a monitoring alert fires. The cause may be much earlier — a state that was corrupted before the routine that produced the failure was called, or an incorrect assumption in a component that computed an intermediate result. Localization means finding the boundary at which correct behavior stops and incorrect behavior begins. Contracts help here: if a postcondition is violated, the cause is in the routine whose postcondition is violated. If an invariant is violated, the cause is in the last operation that modified the invariant's subject.

Step 3: Form a hypothesis. A hypothesis is a specific, testable claim about the cause. Not "something is wrong with the transition logic" but "the `complete` operation does not check whether the task is in `IN_TRANSIT` before setting status to `DELIVERED`." A hypothesis names a specific mechanism by which the observed failure could be produced. It must be specific enough that a single check can distinguish it from alternatives.

Step 4: Run a targeted check. Design and execute the minimum check that would produce different results if the hypothesis is correct versus incorrect. For the hypothesis above: call `complete` on a task whose status is `PENDING` and observe whether the status becomes `DELIVERED`. If it does, the hypothesis is confirmed. If it does not, the hypothesis is wrong and a new one must be formed.

Step 5: Confirm the cause. A confirmed hypothesis explains not just why the specific failing case fails but why the entire failure pattern occurs. If the hypothesis explains the reported failure but not the full pattern — if it explains why some tasks go directly to `DELIVERED` but not why others do not — the hypothesis is incomplete and must be refined before a fix is applied.

Step 6: Patch and verify regression safety. Apply the minimal change that corrects the cause. Minimal means: the change is no larger than the confirmed cause requires. Then run the full existing test suite and add a new test that would have caught the bug before the fix was applied. The new test is not optional — it is evidence that the cause has been corrected and that the same failure cannot recur without detection.

32.2. Reproducibility as a Prerequisite

Every step after step 1 depends on the ability to execute the failure on demand. Without reliable reproduction, hypothesis formation is speculation — there is no way to confirm a hypothesis against a failure that may or may not appear when the check is executed. Without reliable reproduction, regression safety cannot be verified — the fix cannot be confirmed to have eliminated the failure rather than merely failed to trigger it in the verification run.

Reproducibility requires capturing three things: the exact inputs to the failing operation, the system state at the time of the call, and the discrepancy between the observed and expected outputs. The first two are what a minimal test case encodes. The third is the oracle — the statement of what correct behavior would look like, derived from the contract.

When a failure is not immediately reproducible — when it depends on timing, concurrency, or accumulated state — the path to reproducibility is to simplify the context until the variable factors are eliminated. Remove the concurrency and see if the failure persists. Replace the accumulated state with a constructed minimal state that produces the same failure. Reduce the inputs until the smallest failing case is found. Each simplification either preserves the failure — confirming that the removed factor is not the cause — or eliminates it — revealing that the removed factor is the cause.

32.3. Contracts as Localization Tools

The most expensive part of debugging is often localization: finding the boundary between where behavior is correct and where it is not. In a system without explicit contracts, this boundary must be found by reading code and reasoning about what each piece should do. In a system with contracts, the boundary announces itself: a violated precondition identifies a caller that failed to satisfy its obligations; a violated postcondition identifies a routine that failed to satisfy its guarantees; a violated invariant identifies the operation that left the object in an inconsistent state.

The worked case in this chapter illustrates this. The bug report is: *"Some delivery tasks jump directly from `PENDING` to `DELIVERED`."* In a system without transition contracts, debugging this requires tracing every code path that sets the `status` field to `"DELIVERED"` and checking whether each one verifies the prior state. There may be many such paths, and the one that skips the check may be obscure.

In a system where `complete` has a precondition `status = "IN_TRANSIT"`, the debugging path is different. The bug indicates that `complete` is being called on tasks that are not `IN_TRANSIT`. The precondition should have prevented this; if it did not, either the precondition is missing or the code does not enforce it. Checking whether the precondition is present and enforced is the targeted check for the most likely hypothesis — and if the precondition is absent, the fix is precisely to add it.

32.4. From Bug Report to Fix

Consider the report:

> *"Some delivery tasks jump directly from `PENDING` to `DELIVERED`."*

Step 1: Reproduce. Construct a `Delivery_Task` with status `PENDING`. Call `complete` directly. Observe the status becomes `DELIVERED`. This is the minimal reproducible case.

Step 2: Localize. The `complete` operation sets status to `DELIVERED`. The status transitions from `PENDING` to `DELIVERED` in a single call. The boundary is the entry to `complete` — the failure is not in what `complete` does after the check, but in what it fails to check before acting.

Step 3: Hypothesize. `complete` does not verify that the task is `IN_TRANSIT` before setting status to `DELIVERED`. The precondition `status = "IN_TRANSIT"` is absent or unenforced.

Step 4: Check. Inspect the implementation of `complete`. Confirm that the `require` clause is absent. Call `complete` on a `PENDING` task and observe that it succeeds without error. The hypothesis is confirmed.

Step 5: Confirm. The missing precondition explains the full failure pattern: any call to `complete` from a non-`IN_TRANSIT` state will succeed, not just calls from `PENDING`. The `FAILED` state has the same vulnerability.

Step 6: Patch. Add the precondition `in_transit: status = "IN_TRANSIT"` to `complete`. Add a regression test that attempts to call `complete` on a `PENDING` task and verifies that it is rejected. Run the full existing test suite to confirm no regression.

32.5. A Patched Contract in Code

```
class Delivery_Task
create
  make_pending() do
    status := "PENDING"
  end
feature
  status: String

  start()
```

```
    require
      can_start: status = "PENDING" or status = "FAILED"
    do
      status := "IN_TRANSIT"
    ensure
      now_in_transit: status = "IN_TRANSIT"
    end

  complete()
    require
      in_transit: status = "IN_TRANSIT"
    do
      status := "DELIVERED"
    ensure
      delivered: status = "DELIVERED"
    end
invariant
  valid_status:
    status = "PENDING" or
    status = "IN_TRANSIT" or
    status = "DELIVERED" or
    status = "FAILED"
end

class Debug_Smoke_Test
feature
  run(): String
    do
      let t: Delivery_Task :=
       create Delivery_Task.make_pending
      t.start
      t.complete

      if t.status = "DELIVERED" then
        result := "PASS"
      else
        result := "FAIL"
      end
    ensure
      known_result: result = "PASS" or result = "FAIL"
    end
end
```

The `complete` operation now carries the precondition `in_transit: status = "IN_TRANSIT"`. This precondition is the fix — not a conditional inside the body that handles the error case gracefully, but a boundary condition that rejects the call before any state changes occur. A caller that attempts to complete a `PENDING` task violates the precondition and receives a contract failure immediately, at the point of the illegal call, rather than producing corrupted state that will be discovered later.

`Debug_Smoke_Test.run` exercises the legal transition sequence: `PENDING → IN_TRANSIT → DELIVERED` via `start` followed by `complete`. The test confirms that the corrected `complete` still works correctly on valid inputs. It is not the regression test for the bug — that test would attempt to call `complete` from `PENDING` and confirm it is rejected — but it is evidence that the fix did not break the operation for the inputs it was designed to handle.

The `start` operation also has a precondition: `status = "PENDING" or status = "FAILED"`. This is the transition contract from Chapter 10's model: the complete legal transition graph, encoded as preconditions on each operation. The bug exposed by the failing report was a gap in this graph — `complete` without its precondition allowed a transition the model declared illegal. The fix closes the gap.

32.6. Debugging in the Three Systems

In the delivery system, the most common class of debugging problem is transition violations — operations called from illegal states — and the contract-based localization described in this chapter applies directly. A robot that is dispatched before it is ready, a task that is marked delivered before it was started: these failures are announced by violated preconditions when contracts are in place, and diagnosed by tracing unguarded field mutations when they are not.

In the knowledge engine, the most common debugging challenges are non-determinism — the same query producing different results on consecutive calls — and index staleness — queries that return results inconsistent with the current document collection. Non-determinism is diagnosed by finding the hidden state that varies between calls; contracts that require deterministic outputs make the violation detectable. Index staleness is diagnosed by comparing the index's last-updated timestamp against the collection's last-modified timestamp; an invariant that requires these to agree within a defined tolerance makes the discrepancy checkable.

In the virtual world, the most common debugging challenges are non-deterministic update ordering — two runs of the same scenario producing different outcomes — and invariant violations — entities ending a tick in states that violate the world's rules. Non-determinism is diagnosed by finding the operation whose output depends on execution order; an invariant that requires deterministic tick output makes the violation detectable. Invariant violations are diagnosed by checking the world state after each tick against the declared invariants; violations are caught at the tick boundary rather than at the point where incorrect state later produces incorrect behavior.

In all three systems, contracts and invariants are the primary localization tools. They convert debugging from a search through an unknown codebase into a directed investigation of a known boundary violation.

32.7. Three Ways Debugging Goes Wrong

Patching before reproducing. A change applied to an unreproduced failure may correct the behavior in the cases that were visible while leaving the cause unchanged. The failure returns in a slightly different form, or under slightly different conditions, or with a slightly different symptom. The cycle repeats. Each patch narrows the visible failure space without narrowing the actual cause. Reproducibility is not a courtesy — it is the prerequisite for a fix that addresses the cause.

Multiple simultaneous changes. When two hypotheses are plausible, the temptation is to apply fixes for both simultaneously. If the failure disappears, the combined fix works — but it is not known which half was necessary, which half was unnecessary, and whether the unnecessary half introduced a regression. Each hypothesis must be tested independently, which means each change must be applied independently. The process is slower per hypothesis but faster overall, because the cause is known when the fix is confirmed.

No regression safety net. A fix without a regression test is a fix without evidence of permanence. The same bug can be reintroduced by a future change that does not know the previous fix was needed. The regression test is the record of the bug — its existence says: a failure at this boundary was found here, was fixed here, and must be detected here if it recurs. A codebase whose bug fixes are systematically accompanied by regression tests is a codebase that becomes harder to break over time, rather than one that maintains a constant defect rate despite continuous repair.

32.8. Quick Exercise

Take one recent bug — a failure that was observed, diagnosed, and fixed — and reconstruct the debugging process in six parts: the minimal reproducible case, the localized failure boundary, the specific hypothesis that was formed, the targeted check that confirmed it, the patch that was applied, and the regression test that was added.

If any step is missing from the reconstruction — if the fix was applied without a confirmed hypothesis, or without a regression test — identify what evidence that step would have provided and what risk its absence created. The reconstruction is not retrospective documentation; it is a test of whether the debugging process that was used was engineering-grade or trial-and-error.

32.9. Takeaways

- Debugging is hypothesis-driven engineering. The six steps — reproduce, localize, hypothesize, check, confirm, patch — are not formalities. Each one provides evidence that the subsequent steps depend on.
- Reproducibility is a prerequisite, not a convenience. A fix applied to an unreproduced failure is a guess, not an engineering decision.
- Contracts and invariants are localization tools. A violated precondition identifies the caller; a violated postcondition identifies the routine; a violated invariant identifies the operation. They convert debugging from search to investigation.
- One hypothesis, one change, one check. Multiple simultaneous changes produce results that cannot be interpreted. Each hypothesis must be tested independently.
- Every fix needs a regression test. The test is the record of the bug — the evidence that the same failure cannot recur without detection.

Part VII has established the complete trustworthiness toolkit: preconditions and postconditions that define behavioral contracts, invariants that enforce object-level consistency, tests that provide evidence that contracts hold, and debugging that addresses causes rather than symptoms. These are the practices that make software dependable not just at completion but over the full arc of its evolution.

Studio 5 — Reliability

Subtitle

Hardening the Studio 4 architecture for failure, recovery, and operational trust.

Studio Focus This studio chapter is hands-on: implement, verify behavior, and document tradeoffs as you iterate.

The Situation

Studio 4 gave us clear architectural seams: ports, adapters, and services.

Now reliability work becomes practical because failure handling can be localized at those seams.

Current symptoms:

- intermittent failures at integration boundaries
- retries that duplicate side effects
- hard-to-reproduce incidents
- regressions after urgent patches

Studio 5 is the logical continuation: keep Studio 4 architecture, add reliability guarantees around it.

Engineering Brief

Build reliability mechanisms on top of Studio 4 components without breaking behavior contracts.

Required outcomes:

- explicit failure semantics for key operations
- retry policy with bounded attempts
- idempotency for repeatable operations
- reproducible failure tests and regression checks
- minimal observability hooks for diagnosis

Implementation guidance:

- enforce reliability at service/port boundaries
- keep domain invariants intact under failure paths
- separate transient failures from permanent failures
- make each retry policy explicit and testable

Implementation In Nex

Suggested files:

- `reliability_types.nex`
- `delivery_reliability.nex`
- `knowledge_reliability.nex`
- `world_reliability.nex`
- `studio_5_main.nex`

If using the web IDE, place all classes in one file and run `App.run`.

Shared Reliability Types

```
class Op_Result
create
  make(status: String, value: String, attempts: Integer) do
    this.status := status
    this.value := value
    this.attempts := attempts
  end
feature
  status: String
  value: String
  attempts: Integer
invariant
  status_present: status /= ""
  value_present: value /= ""
  attempts_non_negative: attempts >= 0
end

class Reliability_Log
create
  make(initial_event: String) do
    this.last_event := initial_event
  end
feature
  last_event: String

  record(event: String)
    require
      event_present: event /= ""
    do
      last_event := event
    ensure
      persisted: last_event = event
    end
end
```

Delivery Reliability: Retry + Idempotent Dispatch

Continuation from Studio 4 `Delivery_Workflow` idea.

```
class Delivery_Port
create
  make(fail_until_attempt: Integer) do
    this.fail_until_attempt := fail_until_attempt
  end
feature
  -- Simulate transient failures for first N attempts.
  fail_until_attempt: Integer

  send_dispatch(task_id: String, attempt: Integer): String
    require
      id_present: task_id /= ""
      attempt_valid: attempt >= 1
    do
      if attempt <= fail_until_attempt then
        result := "TRANSIENT_FAILURE"
      else
        result := "SENT"
      end
    ensure
      known_result:
        result = "SENT" or
        result = "TRANSIENT_FAILURE" or
        result = "PERMANENT_FAILURE"
    end
end

class Delivery_Reliability_Service
create
  make(
    port: Delivery_Port,
    log: Reliability_Log,
    last_dispatched_task: String
  ) do
    this.port := port
    this.log := log
    this.last_dispatched_task := last_dispatched_task
  end
feature
  port: Delivery_Port
  log: Reliability_Log
  last_dispatched_task: String

  dispatch_with_retry(task_id: String,
                      max_attempts: Integer): Op_Result
    require
      id_present: task_id /= ""
      max_attempts_valid: max_attempts >= 1
    do
      -- Idempotency guard: repeated same task
      -- dispatch request.
      if last_dispatched_task = task_id then
        result := create Op_Result.make(
          "IDEMPOTENT_OK",
          "ALREADY_SENT",
          0
        )
      else
        let a: Integer := 1
        let final_status: String := "TRANSIENT_FAILURE"
```

```
        from
        until a > max_attempts or final_status = "SENT" do
          final_status := port.send_dispatch(task_id, a)
          log.record("delivery_attempt=" + a)
          a := a + 1
        end

        if final_status = "SENT" then
          last_dispatched_task := task_id
          result := create Op_Result.make("OK", "SENT", a - 1)
        else
          result :=
           create Op_Result.make("FAILED", final_status, a - 1)
        end
      end
    ensure
      known_status:
        result.status = "OK" or
        result.status = "FAILED" or
        result.status = "IDEMPOTENT_OK"
    end
end
```

Knowledge Reliability: Safe Query Fallback

Continuation from Studio 4 query/validation services.

```
class Knowledge_Port
create
  make(query_available: Boolean) do
    this.query_available := query_available
  end
feature
  query_available: Boolean

  fetch_top(query: String): String
    require
      query_present: query /= ""
    do
      if query_available then
        result := "DOC:K-101"
      else
        result := "UNAVAILABLE"
      end
    ensure
      non_empty: result /= ""
    end
end

class Knowledge_Reliability_Service
create
  make(port: Knowledge_Port, log: Reliability_Log) do
    this.port := port
    this.log := log
  end
feature
  port: Knowledge_Port
  log: Reliability_Log

  safe_query(query: String): Op_Result
    require
```

```
      query_present: query /= ""
    do
      let raw: String := port.fetch_top(query)

      if raw = "UNAVAILABLE" then
        log.record("knowledge_fallback_used")
        result :=
         create Op_Result.make("DEGRADED", "DOC:CACHED-001", 1)
      else
        result := create Op_Result.make("OK", raw, 1)
      end
    ensure
      known_status: result.status = "OK"
                    or result.status = "DEGRADED"
    end
end
```

World Reliability: Guarded Update + Recovery Status

Continuation from Studio 4 world update service.

```
class World_Port
create
  make(object_id: String, x: Integer, max_x: Integer) do
    this.object_id := object_id
    this.x := x
    this.max_x := max_x
  end
feature
  object_id: String
  x: Integer
  max_x: Integer

  move(delta: Integer): String
    do
      let next: Integer := x + delta
      if next < 0 then
        x := 0
      elseif next > max_x then
        x := max_x
      else
        x := next
      end
      result := "UPDATED"
    ensure
      bounded: x >= 0 and x <= max_x
    end
end

class World_Reliability_Service
create
  make(port: World_Port, log: Reliability_Log) do
    this.port := port
    this.log := log
  end
feature
  port: World_Port
  log: Reliability_Log

  safe_step(delta: Integer): Op_Result
    do
```

```
      let status: String := port.move(delta)
      log.record("world_step_delta=" + delta)

      result := create Op_Result.make("OK", status, 1)
    ensure
      known_status: result.status = "OK"
    end
end
```

Reliability Driver + Regression Checks

```
class App
feature
  run() do
    let l: Reliability_Log :=
     create Reliability_Log.make("init")

    -- Delivery reliability run
    let dp: Delivery_Port := create Delivery_Port.make(2)
    let dr: Delivery_Reliability_Service
      := create Delivery_Reliability_Service.make(dp, l, "NONE")

    let d1: Op_Result := dr.dispatch_with_retry("T-7", 3)
    print("Delivery result: " + d1.status + ", attempts=" + d1.attempts)

    let d2: Op_Result := dr.dispatch_with_retry("T-7", 3)
    print(
      "Delivery idempotent result: " +
      d2.status +
      ", value=" + d2.value
    )

    -- Knowledge reliability run
    let kp: Knowledge_Port := create Knowledge_Port.make(false)
    let kr: Knowledge_Reliability_Service
      := create Knowledge_Reliability_Service.make(kp, l)

    let k1: Op_Result := kr.safe_query("contracts")
    print("Knowledge result: " + k1.status + ", value=" + k1.value)

    -- World reliability run
    let wp: World_Port := create World_Port.make("E-1", 9, 10)
    let wr: World_Reliability_Service
      := create World_Reliability_Service.make(wp, l)

    let w1: Op_Result := wr.safe_step(5)
    print(
      "World result: " + w1.status +
      ", value=" + w1.value +
      ", x=" + wp.x
    )

    -- Simple regression assertions
    if d1.status = "OK" and
       d1.attempts = 3 and
       d2.status = "IDEMPOTENT_OK" and
       k1.status = "DEGRADED" and
       wp.x = 10 then
      print("RELIABILITY_CHECK:PASS")
    else
      print("RELIABILITY_CHECK:FAIL")
    end
```

```
  end
end
```

Expected outcomes:

- delivery handles transient failures with bounded retry
- repeated dispatch request is idempotent
- knowledge service degrades gracefully with fallback
- world updates preserve invariants under extreme deltas

Studio Challenges

Level 1 — Single Reliability Mechanism

- choose one Studio 4 service
- add either retry, idempotency, or fallback logic
- add one regression check proving behavior under failure

Level 2 — Three-Domain Reliability Pass

- implement one reliability mechanism per domain:
  - delivery: retry/idempotency
  - knowledge: degraded fallback
  - world: invariant-preserving recovery
- show failure-path run logs

Level 3 — Incident Drill

- define one simulated production incident per domain
- capture reproduction steps
- patch through service boundary
- add regression checks to prevent recurrence

Postmortem

Discuss with evidence:

- Which reliability mechanism produced the largest stability gain?
- Which failures should not be retried?
- Which fallback behavior is acceptable vs dangerous?
- Which diagnostics made root-cause fastest?

Deliverables

- reliability policy sheet (retry, idempotency, fallback, non-retryable failures)
- refactored Nex reliability services and run outputs
- incident playbook with reproduction + fix + regression tests
- release gate checklist for reliability sign-off

Exit Criteria

You are ready for Part VIII if:

- key operations have explicit failure semantics
- retry logic is bounded and test-backed
- idempotent behaviors are enforced where side effects exist
- degraded-mode behavior is intentional and contract-defined
- regression checks cover critical incident classes

Part VIII.

Part VIII — Systems That Grow

33. Managing Complexity

Part VII gave us the tools to make software trustworthy: contracts that make assumptions explicit, invariants that enforce object-level consistency, tests that provide evidence, and debugging that addresses causes. A system built with these tools is more reliable than one built without them. But reliability at a moment in time is not the same as reliability over time. As requirements grow, as teams expand, as integrations multiply, a system faces a different class of challenge — not whether the current behavior is correct, but whether the system can be understood and changed without destroying what is already correct.

This is the challenge of complexity, and it requires a different kind of engineering discipline.

33.1. What Complexity Actually Costs

Complexity is not the same as size. A large system can be low-complexity if its pieces are well-bounded and independently understandable. A small system can be high-complexity if every piece is tangled with every other and no change is local. The cost of complexity is not the number of lines — it is the cost of reasoning about behavior.

That cost appears in specific, measurable ways. When a change to one module requires understanding the behavior of three others before the first can be safely modified, the cost of reasoning is high. When a new developer cannot make a confident change until they have spent weeks reading the codebase, the cost of onboarding is high. When a routine edit has a non-negligible probability of introducing an incident in an unrelated part of the system, the cost of change has exceeded what the system can afford.

Every system has something like a complexity budget. The budget is not a fixed quantity — it varies with team size, documentation quality, test coverage, and the experience of the developers involved. But when the budget is exceeded, the symptoms are reliable: changes become fragile, incidents follow routine edits, and the velocity of development slows not because the features are difficult but because the system is.

Managing complexity is not a refactoring task that can be deferred until convenient. It is an ongoing engineering practice — the accumulation of structural decisions that determine whether a system's complexity stays within its budget as it grows.

33.2. Layered Reasoning

The most practical tool for managing complexity in a growing system is layered architecture: the organization of code into strata with defined responsibilities and a rule about the direction in which dependencies flow.

A three-layer model serves most systems well.

The domain layer contains the business rules, invariants, and core policies of the system. It is the layer that knows what the system is for — what a delivery task is, what transitions are legal, what the routing rules require, what makes a task eligible for rerouting. The domain layer does not know how data is stored, how messages are delivered, or how the user interface presents information. It knows the domain.

The application layer contains the orchestration of use cases — the workflows that accomplish goals by calling domain objects and coordinating their interactions. A dispatch workflow that computes a route, assigns it to a task, and signals notification is application layer logic. The application layer knows the domain layer's interfaces; it does not know the infrastructure layer's implementation details.

The infrastructure layer contains the adapters, transports, and storage implementations that connect the system to the outside world. It knows how to write a record to a database, how to send a message over a network, how to render a response for a client. It depends on the domain and application layers — not the reverse.

The rule that makes this structure useful is the direction of dependency: all dependencies point inward, toward the domain. Infrastructure depends on application, which depends on domain. The domain depends on nothing outside itself. This means the domain can be tested without infrastructure, reasoned about without knowing the deployment context, and changed without considering the impact on transport or storage implementations. It means that swapping one infrastructure implementation for another — replacing one database with a different one, adding a new notification channel — requires changing only the infrastructure layer. The domain and application layers are untouched.

When this rule is violated — when the domain layer imports an infrastructure detail, or when the application layer reaches directly into storage implementation — the benefit of layering disappears. The layers exist as a naming convention but not as a reasoning boundary.

33.3. From Requirement to Layered Design

Consider the requirement:

> *"Add priority rerouting for premium tasks without breaking normal dispatch."*

Without layered structure, the rerouting decision — is this task eligible for priority treatment? — may be implemented wherever it is most convenient: in the adapter that receives the rerouting request, in the storage layer that retrieves the task, or scattered across several handlers that each check eligibility independently. When the eligibility rules change — when a new tier is added, when the conditions for rerouting are revised — every location that encodes the rule must be found and updated. The probability of an inconsistency is proportional to the number of locations.

With layered structure, the eligibility decision belongs in the domain layer. It is a business rule — a claim about what the system's policy is — and business rules belong in the layer that contains business rules. The application layer calls the domain's eligibility check and orchestrates the rerouting workflow based on the result. The infrastructure layer executes the storage and notification effects.

When the eligibility rules change, the domain layer changes. The application layer changes only if the workflow structure changes. The infrastructure layer does not change. The impact of the requirement change is contained to the layer that owns the relevant concern.

33.4. A Layered Design in Code

```
class Task
create
  make(task_id: String, tier: String, status: String) do
    this.task_id := task_id
    this.tier := tier
    this.status := status
  end
feature
  task_id: String
  tier: String
  status: String
invariant
  id_present: task_id /= ""
  valid_tier: tier = "STANDARD" or tier = "PREMIUM"
end
```

```
class Reroute_Policy
feature
  should_reroute(t: Task): Boolean
    require
      task_present: t.task_id /= ""
    do
      result := t.tier = "PREMIUM" and t.status = "IN_TRANSIT"
    ensure
      bool_result: result = true or result = false
    end
end

class Dispatch_App_Service
create
  make(policy: Reroute_Policy) do
    this.policy := policy
  end
feature
  policy: Reroute_Policy

  process_reroute(t: Task): String
    require
      task_present: t.task_id /= ""
    do
      if policy.should_reroute(t) then
        result := "REROUTE_TRIGGERED"
      else
        result := "NO_REROUTE"
      end
    ensure
      known_result:
        result = "REROUTE_TRIGGERED" or
        result = "NO_REROUTE"
    end
end
```

`Task` and `Reroute_Policy` are domain layer. `Dispatch_App_Service` is application layer. There is no infrastructure layer in this sketch — no storage, no notification, no transport — and that absence is intentional. The domain and application layers contain the logic that determines what the system does. The infrastructure layer, when added, contains the code that executes it. The separation is visible in what each layer imports: `Dispatch_App_Service` depends on `Reroute_Policy` and `Task`; nothing here depends on a database or a message queue.

`Reroute_Policy.should_reroute` encodes the eligibility rule: a task must be `PREMIUM` tier and `IN_TRANSIT` status to qualify for rerouting. When this rule changes — when a new tier is introduced, when rerouting becomes available to `STANDARD` tasks above a certain age — this is the single location that must change. The application service `process_reroute` does not encode the rule; it calls the policy and acts on the result. A developer reading `process_reroute` can understand the workflow without understanding the policy logic, and a developer changing the policy logic can do so without reading the workflow.

The invariant on `Task` — `valid_tier: tier = "STANDARD" or tier = "PREMIUM"` — is a domain constraint. It asserts that the tier field always holds one of the two declared values. When a new tier is introduced, this invariant must be updated, which forces the developer to consider every piece of code that depends on the tier field being one of the current values. The invariant is a change-impact prompt as well as a correctness check.

33.5. Complexity in the Three Systems

In the delivery system, the dispatch logic is the domain. Route computation, task state transitions, and rerouting eligibility are domain concerns. The scheduler that triggers dispatch workflows and the notification system that delivers updates are infrastructure. When dispatch logic leaks into the scheduler or the notification handler, changes to dispatch policy require reading and potentially modifying infrastructure code — code that exists to execute effects, not to encode decisions.

In the knowledge engine, document retrieval and relevance ranking are domain concerns. The index data structure is infrastructure. When ranking logic leaks into the index implementation — when the decision of which documents are relevant is made inside the code that retrieves them — ranking policy and storage implementation are tangled. Changing the ranking algorithm requires understanding the index, and changing the index requires understanding the ranking algorithm.

In the virtual world, the rules that govern entity transitions and interaction outcomes are domain concerns. The collision detection algorithm is application logic. The rendering pipeline is infrastructure. When update logic is interwoven with rendering — when the decision of what happens at a tick is made inside the code that draws the result — neither can be tested independently, and changing the rendering format requires understanding the simulation logic.

In all three systems, the complexity problems are the same in form: a decision that should be localized to one layer has been implemented in another, and the two concerns are tangled enough that neither can change without involving the other.

33.6. Three Ways Complexity Management Fails

Layer bypasses. Code in the infrastructure or interface layer that modifies domain state directly — without going through the domain object's defined operations — bypasses the domain's invariants and contracts. The domain's guarantees are no longer reliable because they can be violated by code outside the domain. Debugging requires not only reading domain code but reading every piece of infrastructure code that might touch domain state. The remedy is to route all state changes through the domain layer's operations, which means the infrastructure layer must call into the application layer, which calls into the domain.

The shared utility dumping ground. A module that accumulates unrelated logic — because each piece was too small to warrant its own module and the existing one was the most convenient destination — becomes a module without coherent identity or stable interface. It has no single reason to change and no single owner. Changes to it require understanding everything it does, and developers avoid moving logic out of it because the refactoring cost is high. The remedy is to split by domain responsibility rather than by convenience: each module should have a coherent identity that its name accurately reflects.

Cyclic dependencies. When module A depends on module B and module B depends on module A, neither can be changed, compiled, or deployed without the other. The cycle is an architectural symptom: the two modules have not been separated at a semantically coherent boundary, and the dependency structure reflects tangled responsibilities rather than intentional design. The remedy is to extract an interface or shared abstraction that both modules depend on, breaking the cycle by introducing a dependency that points in one direction only.

33.7. Quick Exercise

Choose one subsystem in your system and map its current module structure with four parts: the modules and their declared responsibilities, the dependencies between them and the direction each flows, one dependency that points in the wrong direction — from application toward domain is correct; from domain toward infrastructure is not — and one interface or abstraction that, if introduced, would break the incorrect dependency and correct the flow.

For each module, ask: does this module's behavior need to change when domain rules change, when the orchestration workflow changes, or when the infrastructure implementation changes? A module that would need to change for all three reasons is a module whose responsibilities span more than one layer.

33.8. Takeaways

- Complexity is the cost of reasoning about behavior, not the count of lines or modules. A large system can be low-complexity; a small system can be high-complexity. The difference is the quality of the boundaries.
- Every system has a complexity budget. When it is exceeded, fragile changes, long onboarding, and rising incident rates follow. Managing complexity is not a deferred task — it is an ongoing structural practice.
- Layered architecture contains change impact: domain rules change in the domain layer, orchestration changes in the application layer, infrastructure changes in the infrastructure layer. Each layer is comprehensible in isolation because it depends only on what is inward of it.
- Dependencies must flow inward. A domain layer that imports infrastructure, or an application layer that bypasses the domain, destroys the separation that makes layering useful.
- The three failure modes — layer bypasses, utility dumping grounds, and cyclic dependencies — all have the same root cause: responsibilities that were not assigned to a coherent layer before the code was written.

Chapter 32 examines how to design boundaries that absorb future change safely — the discipline of identifying the seams at which a system is most likely to change and structuring those seams so that changes on one side do not force changes on the other.

34. Designing for Change

Managing complexity, as we saw in Chapter 31, is the discipline of keeping a system understandable today. It is about drawing boundaries so that the cost of reasoning remains within our budget. But a system that is easy to understand today may still be prohibitively expensive to change tomorrow. Requirements evolve, technology stacks shift, and business models pivot. If every such change requires reaching across boundaries and rewriting stable logic, the system has failed a fundamental engineering test.

Designing for change is the practice of building "seams" into a system — locations where behavior can vary without breaking the contracts that surround them. The goal is not to predict the future, which is impossible, but to build a system that can absorb it.

34.1. The Concept of Change Seams

A seam is a boundary where we can alter behavior without editing the code that uses that behavior. In a well-designed system, seams are placed at the points of highest likely volatility.

Consider a notification system. Today, it sends emails. Tomorrow, it might need to send SMS messages, push notifications, or Slack alerts. If the logic for "how to send an email" is hardcoded into the heart of the dispatch workflow, then adding SMS support requires modifying the dispatch workflow. The dispatch workflow is now coupled to the notification transport.

A seam decouples them. By defining a stable contract — `Notification_Transport.send(message)` — the dispatch workflow can remain entirely ignorant of how the message is delivered. It depends on the *interface* of the transport, not its implementation. When a new transport is added, we don't change the dispatch logic; we simply provide a different implementation of the seam.

Good seams isolate volatility. They allow the stable parts of the system (the "what") to remain unchanged while the volatile parts (the "how") evolve independently.

34.2. Stable Contracts and the Cost of Evolution

The backbone of a change-safe system is the stable contract. A contract is an agreement about behavior, inputs, and outputs. When a contract is stable, the code on either side of it can change freely as long as the agreement is honored.

The most dangerous kind of change is the "breaking" change — an alteration to a contract that forces every consumer of that contract to change as well. In a large system, a single breaking change at a low level can trigger a cascade of updates that consumes weeks of engineering time.

To minimize this cost, we adopt a discipline of additive evolution: - **Prefer additive fields:** When a data structure needs more information, add a new field rather than repurposing an old one. - **Maintain deprecation windows:** Give consumers time to migrate to new interfaces before removing old ones. - **Provide compatibility adapters:** If a contract must change, provide a layer that allows old clients to talk to the new implementation.

Without versioning discipline and contract stability, every "improvement" to the system becomes a migration crisis for the rest of the team.

34.3. From Requirement to Flexible Design

Consider the requirement: > *"We need to experiment with a new ranking algorithm for search results, but we must be able to switch back to the old one instantly if the metrics drop."*

Without a seam, the ranking logic is likely embedded in the search service. Switching algorithms means a code change, a deployment, and a high-risk transition.

With a seam, we design for variation:

1. **Define the Port:** We create a stable interface, `Ranking_Strategy`, with a single operation: `rank(results: List[Document])`.
2. **Implement the Variants:** We create `Legacy_Ranking` and `Experimental_Ranking`, both adhering to the `Ranking_Strategy` contract.
3. **Introduce the Switch:** The search service is given an instance of `Ranking_Strategy` at runtime. Which implementation it gets is decided by a configuration setting or a feature flag.

The search service remains unchanged. Its contract is satisfied by any object that knows how to rank. We can now swap algorithms, run A/B tests, or roll back a failure without touching the core search orchestration.

34.4. Implementation in Nex

In Nex, we use classes and features to define these seams. The `require` and `ensure` clauses make the contract explicit, ensuring that any new implementation of the seam honors the same behavioral guarantees as the old one.

```
-- The Seam Definition
deferred class Ranking_Strategy
feature
  rank(query: String, candidates: Array[String]): Array[String]
    require
      query_present: query /= ""
      has_candidates: candidates.size > 0
    do
    ensure
      results_match_input_size: result.size = candidates.size
    end
end

-- Variant 1: Legacy
class Legacy_Ranking
inherit Ranking_Strategy
feature
  rank(query: String, candidates: Array[String]): Array[String]
    do
      result := candidates
    end
end

-- Variant 2: Modern (ML-based)
class Modern_Ranking
inherit Ranking_Strategy
feature
  rank(query: String, candidates: Array[String]): Array[String]
    do
      -- Complex ranking logic...
      result := candidates -- placeholder
    end
end

-- The Consumer: Unchanged by variation
class Search_Service
create
  make(strategy: Ranking_Strategy) do
    this.strategy := strategy
  end
feature
  strategy: Ranking_Strategy

  fetch_from_index(q: String): Array[String]
    require
      query_present: q /= ""
    do
```

```
      result := ["DOC:A", "DOC:B"]
    ensure
      has_candidates: result.size > 0
    end

  execute_search(q: String): Array[String]
    require
      query_present: q /= ""
    do
      let initial_docs: Array[String] := fetch_from_index(q)
      result := strategy.rank(q, initial_docs)
    ensure
      has_candidates: result.size > 0
    end
end
```

The `deferred` class `Ranking_Strategy` acts as the port. The `Search_Service` depends only on this abstraction. Whether the `strategy` is an instance of `Legacy_Ranking` or `Modern_Ranking` is a configuration detail. The search logic is protected from the volatility of the ranking algorithm.

34.5. Designing for Change Across the Three Systems

In the **delivery system**, the seam is the `Routing_Policy`. We can swap from a "shortest distance" policy to a "minimum fuel" policy or a "driver preference" policy without changing the dispatch engine that executes the routes.

In the **knowledge engine**, the seam is the `Document_Parser`. As new file formats are supported — PDF, Markdown, LaTeX — we add new parser implementations. The indexing pipeline remains unchanged because it interacts with the stable `Parsed_Content` contract.

In the **virtual world**, the seam is the `Entity_Behavior`. A "Player" and an "NPC" might share the same physical simulation rules but have different decision-making logic. By isolating behavior behind a seam, we can add new types of entities without modifying the physics engine.

In all three cases, the pattern is the same: identify the part that is likely to vary, wrap it in a stable contract, and allow the rest of the system to depend on the contract rather than the variation.

34.6. Three Ways Design for Change Fails

Premature Abstraction. The most common mistake is building seams for variations that never happen. Every seam adds a layer of indirection and a small cost to reasoning. If you build five different "strategy" ports for logic that hasn't changed in three years, you have wasted complexity budget. The remedy is to add seams only when volatility is evidenced or highly probable.

Leaky Abstractions. A seam is useless if the contract requires the caller to know about the implementation. If `Ranking_Strategy` requires the caller to pass database-specific credentials, the seam has leaked infrastructure details into the domain. The remedy is to keep contracts focused on the *intent* of the operation, not the *mechanics* of the implementation.

The "Big Bang" Migration. Designing for change implies that the system will evolve. If a team introduces a new version of a service but provides no compatibility for old clients, they haven't designed for change — they've designed for disruption. The remedy is to make compatibility a first-class architectural requirement, using adapters and versioned interfaces to bridge the gap.

34.7. Quick Exercise

Pick one part of your system that you expect to change in the next six months. Define the "Port" (the stable contract) that would allow that change to happen without affecting the surrounding code.

1. What is the name of the contract?
2. What are its `require` and `ensure` conditions?
3. What information must stay *out* of the contract to keep it implementation-agnostic?

34.8. Takeaways

- Designing for change is about building seams where volatility can land without causing system-wide damage.
- A seam is a boundary that allows behavior to vary while the contract remains stable.
- Stable contracts are the prerequisite for safe evolution. Additive changes are almost always cheaper than breaking ones.
- Over-engineering is as dangerous as under-engineering. Build seams where change is likely, not where it is merely possible.

- Compatibility is not a release chore; it is an architectural discipline that allows a system to grow without leaving its users behind.

Chapter 33 examines the practical discipline of refactoring — how to move a system from its current structure to a better one while proving that its behavior remains unchanged.

35. Refactoring Without Fear

A system that grows must change its structure. The elegant abstractions of yesterday often become the bottlenecks of today. But change is dangerous. In a complex system, a structural modification can have unintended behavioral consequences that ripple through unrelated modules. This danger often leads to "architectural paralysis," where teams avoid necessary improvements because they fear breaking the system.

Refactoring is the engineering discipline that addresses this fear. It is the process of improving the internal structure of code without changing its observable behavior. When done correctly, refactoring is not a high-risk event; it is a routine, verified, and reversible part of the development lifecycle.

35.1. Behavior Preservation First

The defining characteristic of refactoring is that it is behavior-neutral. If the observable behavior of the system changes — even if it becomes "better" — you are not refactoring; you are rewriting.

To ensure behavior preservation, we rely on safety nets. In the Nex philosophy, these safety nets are not just external tests, but internal contracts. A refactor is successful if: 1. **Contract checks pass:** The `require` and `ensure` clauses of the refactored modules are still honored. 2. **Regression tests pass:** The suite of automated tests still produces the same results. 3. **Parity is maintained:** On a representative sample of inputs, the new structure produces the exact same outputs as the old structure.

Without these safety nets, refactoring is guesswork. With them, it is a controlled transformation.

35.2. The Incremental Refactor Pattern

The most effective way to reduce the risk of a refactor is to break it into small, verified slices. A "big bang" refactor — where a developer spends a week rewriting a major component and only tests it at the end — is a recipe for disaster.

A more reliable sequence is the Incremental Refactor Pattern:

1. **Capture current behavior:** Before touching the code, ensure you have a baseline of tests or contracts that define what "correct" looks like.
2. **Extract the seam:** Introduce an interface or a boundary that separates the code you want to change from the code that uses it.
3. **Move one responsibility:** Perform the smallest possible structural change. This might be moving a single method or splitting a single class.
4. **Run parity checks:** Verify that the system still works exactly as it did before the move.
5. **Iterate:** Repeat the process until the desired structure is achieved.
6. **Remove the old path:** Only once the new structure is proven stable do you delete the legacy code.

This pattern minimizes the "distance" between a mistake and its discovery. If step 4 fails, you know exactly what caused the regression: the small change you just made in step 3.

35.3. From Problem to Refactored Design

Consider the requirement: > *"The Search Service has grown too large. It now handles both document retrieval and relevance ranking. We need to split these into two independent services."*

A naive approach might be to create two new classes and move code between them. A disciplined approach follows the incremental pattern:

1. **Baseline:** We verify that the current `Search_Service.query()` produces known results for a set of test queries.
2. **The Seam:** We define a `Ranking_Service` class but keep it empty for now.
3. **The Move:** We move the ranking logic from `Search_Service` to `Ranking_Service.rank()`. The `Search_Service` now calls `Ranking_Service.rank()` instead of doing the work itself.
4. **Parity:** We run the same queries as in step 1. If the results are identical, we have successfully refactored the logic without breaking the behavior.

The outward contract of the search system — "given a query, return ranked results" — remains unchanged. The internal structure — "how those results are retrieved and ranked" — is now more modular and maintainable.

35.4. Implementation in Nex

In Nex, we can use the `Refactored_Knowledge_Service` to demonstrate how internal decomposition preserves the external contract.

```
-- Original (Legacy) Service
class Legacy_Knowledge_Service
feature
  query(q: String): String
    require
      query_present: q /= ""
    do
      -- Tangled retrieval and ranking logic
      if q = "graphs" then
        result := "DOC:G-1"
      else
        result := "DOC:GENERIC"
      end
    ensure
      non_empty: result /= ""
    end
end

-- After Refactoring: Two focused services
class Retrieval_Service
feature
  retrieve(q: String): String
    do
      if q = "graphs" then
        result := "CAND:G-1"
      else
        result := "CAND:GENERIC"
      end
    end
end

class Ranking_Service
feature
  rank(candidate: String): String
    do
      if candidate = "CAND:G-1" then
        result := "DOC:G-1"
      else
        result := "DOC:GENERIC"
      end
    end
end

-- The Refactored Orchestrator: Same contract as Legacy
class Refactored_Knowledge_Service
create
  make(retrieval: Retrieval_Service, ranking: Ranking_Service) do
    this.retrieval := retrieval
    this.ranking := ranking
  end
feature
  retrieval: Retrieval_Service
  ranking: Ranking_Service

  query(q: String): String
    require
      query_present: q /= ""
```

```
    do
      let c: String := retrieval.retrieve(q)
      result := ranking.rank(c)
    ensure
      non_empty: result /= ""
    end
end
```

The `Refactored_Knowledge_Service.query()` has the exact same `require` and `ensure` as the legacy version. Any client using the legacy service can switch to the refactored one without knowing that the internal implementation has changed from a single tangled block to a clean two-service architecture.

35.5. Refactoring Across the Three Systems

In the **delivery system**, refactoring often involves moving state transition logic out of a generic "task manager" and into specific "state machine" objects. The safety check is ensuring that the task still moves through the correct lifecycle.

In the **knowledge engine**, refactoring might involve splitting the "indexing" pipeline into separate "extraction" and "storage" stages. The safety check is ensuring that a document indexed by the new pipeline is still retrievable and contains the same metadata.

In the **virtual world**, refactoring might involve extracting the "collision detection" algorithm into its own module. The safety check is ensuring that entities still collide at the exact same coordinates they did before.

In every case, the goal is to improve the "how" while strictly preserving the "what."

35.6. Three Ways Refactoring Fails

The "Big Bang" Temptation. As we noted, trying to fix everything at once is the fastest way to break a system. The remedy is to force yourself to work in slices that can be committed and verified in a single hour.

Missing Parity Checks. Assuming that "the tests passed" is enough can be dangerous if the test suite is sparse. The remedy is to use real-world data (or high-fidelity samples) to compare the output of the old and new paths before making the switch permanent.

Early Deletion. Deleting the legacy code before the new path has survived a production workload is a common source of "no-way-back" incidents. The remedy is to keep both paths alive for a short period —

perhaps behind a feature flag — so that you can roll back instantly if a subtle behavioral difference is discovered.

35.7. Quick Exercise

Identify one "tangled" method in your system that performs more than one responsibility. Plan a three-step refactor to split it: 1. What is the baseline behavior you will verify? 2. What is the first small move (e.g., extracting one responsibility into its own method)? 3. How will you prove that the behavior is still identical after the move?

35.8. Takeaways

- Refactoring is the routine improvement of internal structure without changing external behavior.
- Behavior preservation is not a guess; it is a verified state proven by contracts, tests, and parity checks.
- The risk of a refactor is inversely proportional to its size. Small, incremental steps are the path to safety.
- Parity testing — comparing the outputs of old and new paths — is the ultimate confidence mechanism for structural change.
- Fear of change is a symptom of a system without safety nets. Building those nets is the first step toward a system that can grow.

Part VIII established the discipline of growth. In Part IX, we look toward the future: how these engineering principles apply in an age where AI assistants are helping us write and review the code we build.

Studio 6 — Evolution

Subtitle

Evolving the reliable Studio 5 system into a long-lived platform.

Studio Focus This studio chapter is hands-on: implement, verify behavior, and document tradeoffs as you iterate.

The Situation

Studio 5 made the architecture reliable: retries, idempotency, fallback behavior, and reproducible regression checks.

Now the challenge is not reliability alone. It is sustained evolution under new requirements, new clients, and changing policies.

Typical pressure points:

- new business rules that must coexist with legacy behavior
- API evolution without breaking existing consumers
- data migrations that must be reversible
- feature rollout risk in production environments

Studio 6 is the logical next step: evolve safely without losing reliability or architectural clarity.

Engineering Brief

Deliver new capabilities while preserving Studio 5 trust guarantees.

Required outcomes:

- introduce versioned behavior behind stable contracts
- support compatibility mode for old clients
- add explicit migration path and rollback trigger
- preserve reliability checks from Studio 5
- define evolution governance (deprecation, sunset, ownership)

Implementation guidance:

- use additive changes before breaking replacements

- isolate v1/v2 differences behind strategy or adapter seam
- treat migration as code, not informal operations notes
- gate rollout with measurable signals and exit criteria

Implementation In Nex

Suggested files:

- `evolution_types.nex`
- `delivery_evolution.nex`
- `knowledge_evolution.nex`
- `world_evolution.nex`
- `migration_runner.nex`
- `studio_6_main.nex`

If using the web IDE, place all classes in one file and run `App.run`.

Shared Evolution Types

```
class Evolution_Result
create
  make(status: String, value: String, mode: String) do
    this.status := status
    this.value := value
    this.mode := mode
  end
feature
  status: String
  value: String
  mode: String
invariant
  status_present: status /= ""
  value_present: value /= ""
  mode_present: mode /= ""
end

class Rollout_Config
create
  make(use_v2: Boolean, compatibility_mode: Boolean) do
    this.use_v2 := use_v2
    this.compatibility_mode := compatibility_mode
  end
feature
  use_v2: Boolean
  compatibility_mode: Boolean
end
```

Delivery Evolution (Policy V1 -> V2)

Continuation from Studio 5 delivery reliability flow.

```
class Delivery_Policy_V1
feature
  decide_priority(tier: String): String
    require
      tier_present: tier /= ""
    do
      if tier = "PREMIUM" then
        result := "FAST_TRACK"
      else
        result := "STANDARD_TRACK"
      end
    ensure
      known_result: result = "FAST_TRACK" or result = "STANDARD_TRACK"
    end
end

class Delivery_Policy_V2
feature
  decide_priority(tier: String, risk_score: Integer): String
    require
      tier_present: tier /= ""
      risk_non_negative: risk_score >= 0
    do
      if tier = "PREMIUM" and risk_score < 50 then
        result := "FAST_TRACK"
      elseif tier = "PREMIUM" then
        result := "SAFE_TRACK"
      else
        result := "STANDARD_TRACK"
      end
    ensure
      known_result:
        result = "FAST_TRACK" or
        result = "SAFE_TRACK" or
        result = "STANDARD_TRACK"
    end
end

class Delivery_Evolution_Service
create
  make(
    v1: Delivery_Policy_V1,
    v2: Delivery_Policy_V2,
    config: Rollout_Config
  ) do
    this.v1 := v1
    this.v2 := v2
    this.config := config
  end
feature
  v1: Delivery_Policy_V1
  v2: Delivery_Policy_V2
  config: Rollout_Config

  route_mode(tier: String, risk_score: Integer): Evolution_Result
    require
      tier_present: tier /= ""
      risk_non_negative: risk_score >= 0
    do
      if config.use_v2 then
        result := create Evolution_Result.make(
          "OK",
          v2.decide_priority(tier, risk_score),
          "V2"
        )
      else
```

```
        result := create Evolution_Result.make(
          "OK",
          v1.decide_priority(tier),
          "V1"
        )
      end
    ensure
      ok_status: result.status = "OK"
    end
end
```

Knowledge Evolution (Compatibility Adapter)

Continuation from Studio 5 safe query + fallback logic.

```
class Knowledge_Query_V1
feature
  run(query: String): String
    require
      query_present: query /= ""
    do
      result := "DOC:K-LEGACY"
    ensure
      non_empty: result /= ""
    end
end

class Knowledge_Query_V2
feature
  run(query: String, intent: String): String
    require
      query_present: query /= ""
      intent_present: intent /= ""
    do
      if intent = "deep" then
        result := "DOC:K-DEEP"
      else
        result := "DOC:K-FAST"
      end
    ensure
      non_empty: result /= ""
    end
end

class Knowledge_Compatibility_Adapter
create
  make(v2: Knowledge_Query_V2) do
    this.v2 := v2
  end
feature
  v2: Knowledge_Query_V2

  run_v1_shape(query: String): String
    require
      query_present: query /= ""
    do
      result := v2.run(query, "fast")
    ensure
      non_empty: result /= ""
    end
end
```

```
class Knowledge_Evolution_Service
create
  make(
    v1: Knowledge_Query_V1,
    v1_on_v2: Knowledge_Compatibility_Adapter,
    config: Rollout_Config
  ) do
    this.v1 := v1
    this.v1_on_v2 := v1_on_v2
    this.config := config
  end
feature
  v1: Knowledge_Query_V1
  v1_on_v2: Knowledge_Compatibility_Adapter
  config: Rollout_Config

  query(query: String): Evolution_Result
    require
      query_present: query /= ""
    do
      if config.use_v2 and not config.compatibility_mode then
        result := create Evolution_Result.make(
          "OK",
          "DOC:K-DEEP",
          "V2_NATIVE"
        )
      elseif config.use_v2 then
        result := create Evolution_Result.make(
          "OK",
          v1_on_v2.run_v1_shape(query),
          "V2_COMPAT"
        )
      else
        result := create Evolution_Result.make(
          "OK",
          v1.run(query),
          "V1"
        )
      end
    ensure
      ok_status: result.status = "OK"
    end
end
```

World Evolution (Feature Flagged Rule Extension)

Continuation from Studio 5 safe world-step behavior.

```
class World_Rules_V1
feature
  apply(x: Integer, delta: Integer, max_x: Integer): Integer
    require
      max_valid: max_x >= 0
    do
      let next: Integer := x + delta
      if next < 0 then
        result := 0
      elseif next > max_x then
        result := max_x
      else
```

```
        result := next
      end
    ensure
      bounded: result >= 0 and result <= max_x
    end
end

class World_Rules_V2
feature
  apply(x: Integer, delta: Integer, max_x: Integer): Integer
    require
      max_valid: max_x >= 0
    do
      -- New damping rule for safer high-velocity transitions.
      let effective_delta: Integer := delta
      if delta > 3 then
        effective_delta := 3
      elseif delta < -3 then
        effective_delta := -3
      end

      let next: Integer := x + effective_delta
      if next < 0 then
        result := 0
      elseif next > max_x then
        result := max_x
      else
        result := next
      end
    ensure
      bounded: result >= 0 and result <= max_x
    end
end

class World_Evolution_Service
create
  make(
    v1: World_Rules_V1,
    v2: World_Rules_V2,
    config: Rollout_Config
  ) do
    this.v1 := v1
    this.v2 := v2
    this.config := config
  end
feature
  v1: World_Rules_V1
  v2: World_Rules_V2
  config: Rollout_Config

  step(x: Integer, delta: Integer, max_x: Integer): Evolution_Result
    require
      max_valid: max_x >= 0
    do
      if config.use_v2 then
        result := create Evolution_Result.make(
          "OK",
          "X=" + v2.apply(x, delta, max_x),
          "V2"
        )
      else
        result := create Evolution_Result.make(
          "OK",
          "X=" + v1.apply(x, delta, max_x),
          "V1"
        )
```

```
      end
    ensure
      ok_status: result.status = "OK"
    end
end
```

Migration Runner + Rollback Trigger

```
class Migration_Runner
create
  make(initial_count: Integer) do
    this.migrated_count := initial_count
  end
feature
  migrated_count: Integer

  run(canary_passed: Boolean): String
    do
      if canary_passed then
        migrated_count := migrated_count + 100
        result := "MIGRATION_CONTINUE"
      else
        result := "ROLLBACK"
      end
    ensure
      known_result: result = "MIGRATION_CONTINUE" or result = "ROLLBACK"
    end
end

class App
feature
  run() do
    let cfg: Rollout_Config := create Rollout_Config.make(true, true)

    -- Delivery evolution
    let d1: Delivery_Policy_V1 := create Delivery_Policy_V1
    let d2: Delivery_Policy_V2 := create Delivery_Policy_V2
    let de: Delivery_Evolution_Service
      := create Delivery_Evolution_Service.make(d1, d2, cfg)

    let dr: Evolution_Result := de.route_mode("PREMIUM", 70)
    print("Delivery: mode=" + dr.mode + " value=" + dr.value)

    -- Knowledge evolution
    let k1: Knowledge_Query_V1 := create Knowledge_Query_V1
    let kv2: Knowledge_Query_V2 := create Knowledge_Query_V2
    let ka: Knowledge_Compatibility_Adapter
      := create Knowledge_Compatibility_Adapter.make(kv2)
    let ke: Knowledge_Evolution_Service
      := create Knowledge_Evolution_Service.make(k1, ka, cfg)

    let kr: Evolution_Result := ke.query("contracts")
    print("Knowledge: mode=" + kr.mode + " value=" + kr.value)

    -- World evolution
    let w1: World_Rules_V1 := create World_Rules_V1
    let w2: World_Rules_V2 := create World_Rules_V2
    let we: World_Evolution_Service
      := create World_Evolution_Service.make(w1, w2, cfg)

    let wr: Evolution_Result := we.step(8, 6, 10)
    print("World: mode=" + wr.mode + " value=" + wr.value)
```

```
    -- Migration + rollback gate
    let m: Migration_Runner := create Migration_Runner.make(0)
    print("Migration gate: " + m.run(true))
    print("Migration count: " + m.migrated_count)
    print("Rollback gate: " + m.run(false))
  end
end
```

Expected outcomes:

- v2 behavior can be rolled out without breaking v1 clients
- compatibility mode supports gradual client migration
- migration continues only when canary checks pass
- rollback condition is explicit and executable

Studio Challenges

Level 1 — One Evolution Path

- evolve one Studio 5 service from v1 to v2 behind a stable contract
- add compatibility mode
- prove old client behavior still works

Level 2 — Coordinated Multi-Domain Evolution

- add v2 behavior for delivery, knowledge, and world
- define one shared rollout policy
- show canary/rollback behavior with run logs

Level 3 — Future-Proofing Drill

- simulate two additional future changes
- evaluate if current seams absorb them cleanly
- define deprecation timeline and ownership

Postmortem

Discuss with evidence:

- Which seam was most valuable for safe evolution?
- Which compatibility costs were worth paying?
- What should be deprecated next, and when?
- Which metrics should gate future rollouts?

Deliverables

- versioning plan (v1/v2 behavior map)
- compatibility matrix (client version vs service mode)
- migration + rollback playbook with gate conditions
- updated Nex code and evolution run logs
- deprecation roadmap with dates and owners

Exit Criteria

You are ready for Part IX if:

- critical services evolve through versioned seams, not rewrites
- compatibility behavior is explicit and tested
- migration and rollback are operationally defined
- reliability guarantees from Studio 5 remain intact during evolution

Part IX.

Part IX — Programming in the Age of AI

36. Working With AI Coding Assistants

The arrival of AI coding assistants represents one of the most significant shifts in the history of software development. For the first time, we have collaborators that can draft implementation, suggest refactors, and generate tests in seconds. But this speed comes with a profound risk: if we use AI to bypass the rigors of software engineering, we will simply build fragile systems faster.

The key to effective AI collaboration is not learning the right "magic words." It is applying the same engineering constraints to the AI that we apply to ourselves. When we treat the AI as a junior implementation partner working within a strictly defined architectural sandbox, it becomes a powerful accelerator. When we treat it as an oracle that defines our architecture, it becomes a source of systemic debt.

36.1. AI as a Collaborator, Not an Authority

An AI assistant is an implementation engine, not a design authority. It is excellent at "filling in the blanks" once the boundaries, contracts, and invariants have been established. It is much less effective at determining what those boundaries should be.

The most common failure mode in AI-assisted development is the "vague prompt": *"Write a service that handles delivery task rerouting."* This prompt provides no constraints. The AI will likely generate plausible-looking code that ignores your system's existing state management, bypasses your domain invariants, and uses a different error-handling philosophy.

A successful collaborator provides context and constraints. The AI needs to know: - **What is the goal?** (The high-level intent). - **What are the boundaries?** (The existing interfaces it must use). - **What are the invariants?** (The rules it must not break). - **What are the acceptance criteria?** (The tests it must pass).

36.2. Constraint-Driven Prompting

Instead of asking the AI to "write code," we should ask it to "solve a problem within these constraints." This is the discipline of constraint-driven prompting.

A robust prompt follows a structured pattern: 1. **Task Objective:** Define exactly what the new behavior should be. 2. **System Context:** Provide the existing class definitions and contracts. 3. **Non-Negotiable Constraints:** Explicitly forbid certain types of changes (e.g., "Do not modify the `Task` class," "Use the existing `Logger` interface"). 4. **Verification Requirements:** Specify that the AI must also provide the unit tests or contract checks required to verify the new logic.

By anchoring the AI in your system's existing contracts, you force its output to be "born" into your architecture rather than imported as a foreign body.

36.3. From Requirement to AI-Assisted Implementation

Consider the requirement: > *"Add a fallback mechanism to our Knowledge Query Service so that if the primary retrieval fails, it returns a cached result."*

An undisciplined prompt might just ask for "fallback logic." A disciplined workflow looks like this:

1. **Provide the Interface:** Give the AI the current `Knowledge_Query_Service` contract.
2. **Define the Seam:** Tell the AI to implement a `Knowledge_Fallback_Wrapper` that implements the same interface but takes the original service as a dependency.
3. **Specify Invariants:** "The wrapper must return a cached result only if the core service returns `UNAVAILABLE`. It must never return an empty string."
4. **Request Parity Tests:** "Provide a test case showing that when the core service succeeds, the wrapper returns the exact same result."

The AI is now working in a very small, well-defined space. The resulting code is much more likely to be correct, maintainable, and aligned with your system's existing patterns.

36.4. Implementation in Nex

In Nex, our contracts and invariants are the perfect "anchors" for AI prompts. They translate our engineering intent into a form the AI can use to validate its own suggestions.

```
-- Current Interface (Context for the AI)
class Knowledge_Query_Service
feature
  query(q: String): String
    require
      query_present: q /= ""
    do
      -- Simple implementation
      if q = "contracts" then
        result := "DOC:K-101"
      else
        result := "UNAVAILABLE"
      end
    ensure
      non_empty: result /= ""
    end
end

-- AI-Generated Wrapper (Constrained by the interface)
class Knowledge_Fallback_Wrapper
create
  make(core: Knowledge_Query_Service) do
    this.core := core
  end
feature
  core: Knowledge_Query_Service

  query_safe(q: String): String
    require
      query_present: q /= ""
    do
      let raw: String := core.query(q)
      if raw = "UNAVAILABLE" then
        result := "DOC:CACHED-001"
      else
        result := raw
      end
    ensure
      non_empty: result /= ""
    end
end
```

By providing the `Knowledge_Query_Service` as the context, we ensure the AI understands the `require` and `ensure` expectations. The `Knowledge_Fallback_Wrapper` is then generated to respect those same expectations, preserving the system's trustworthiness.

36.5. AI Workflow Across the Three Systems

In the **delivery system**, we might use an assistant to propose three different "fallback route" algorithms. We constrain the AI by providing the `Routing_Contract` and requiring that every proposal honor the "delivery window" invariant.

In the **knowledge engine**, we might ask the assistant to generate a new `Document_Parser` for a specialized file format. We constrain it by providing the `Parser_Interface` and a set of "adversarial" inputs it must handle safely.

In the **virtual world**, we might use an AI to draft a new set of "NPC interaction rules." We constrain it by providing the `Simulation_State` invariant, ensuring the AI doesn't generate rules that allow entities to enter illegal states.

In each case, the engineer's role is to define the "sandbox" and the AI's role is to explore the implementation within it.

36.6. Three Ways AI Collaboration Fails

Prompting without context. Asking an AI to solve a problem without showing it the existing code is like asking a chef to cook in a kitchen they've never seen. The result will likely be incompatible with your environment. The remedy is to always include relevant interfaces, contracts, and a summary of your system's architecture in your prompts.

Accepting the first draft. AI output is a draft, not a final product. It often contains subtle "hallucinations" — calls to methods that don't exist or logic that almost, but not quite, satisfies the invariants. The remedy is to treat AI code with extreme skepticism until it has passed your full battery of contract checks and tests.

Asking for too much at once. Asking an AI to "redesign the entire dispatch system" is a recipe for a high-risk, unreviewable mess. The remedy is to scope your AI tasks to small, bounded changes — the same kind of slices you would use in a manual refactor.

36.7. Quick Exercise

Pick a small feature you need to implement. Write an AI prompt that includes: 1. The high-level goal. 2. The existing Nex class and its contracts. 3. Two explicit "non-negotiable" constraints. 4. A requirement for a specific test case.

Compare the quality of the AI's output with what you might have gotten from a one-sentence prompt.

36.8. Takeaways

- AI coding assistants are powerful implementation engines, but they require human-defined constraints to be effective.
- Prompting is not an art; it is an engineering discipline of defining goals, contexts, and boundaries.
- Contracts and interfaces are the essential "anchors" that keep AI-generated code aligned with your architecture.
- AI is best used for bounded implementation tasks, not for high-level architectural design.
- The faster an AI can write code, the more important your verification safety nets become.

Chapter 35 moves from generation to validation: how to rigorously review AI-generated code to ensure it meets our standards for safety and correctness.

37. Reviewing AI-Generated Code

The power of an AI assistant is its ability to generate code at a speed that exceeds human writing. But in software engineering, writing code is only a fraction of the work. The more important work is ensuring that the code is safe, correct, and fits the architecture. In an AI-assisted workflow, the burden of this responsibility shifts from *authoring* to *reviewing*.

A common mistake in AI workflows is treating the output as "pre-verified." Because the code often looks clean, follows style guides, and compiles on the first try, there is a temptation to give it a superficial review. But AI-generated code can be subtly wrong in ways that manual code is not. It can satisfy the happy path while ignoring the edge cases, or it can introduce hidden couplings that only become apparent during a production incident. Review is the single most important line of defense.

37.1. What to Review First: The Risk-First Hierarchy

When reviewing AI-generated code, don't start with the syntax or the variable names. Start with the behavior. A high-quality review follows a hierarchy of risk:

1. **Contract Correctness:** Does the code satisfy the `require` and `ensure` conditions of its own class and every class it calls? This is the most critical check.
2. **Invariant Preservation:** Does the code maintain the object-level and system-level invariants? AI is notorious for modifying state in ways that bypass the rules of the domain.
3. **Failure Path Handling:** Does the code handle transient failures, timeouts, and invalid inputs? AI tends to be optimistic. You must be the pessimist.
4. **Behavioral Regression:** Does the new code change the observable behavior of existing functions in unintended ways?

5. **Maintainability and Style:** Only once the code is proven safe and correct do you worry about whether the code is "idiomatic."

If the code fails any of the first four checks, the fifth check doesn't matter. You shouldn't polish code that is behaviorally broken.

37.2. The AI Review Checklist

To make review a routine discipline, use a focused checklist for every AI-generated patch:

- ☐ **Contract Alignment:** Does it honor the existing interfaces?
- ☐ **State Safety:** Are all state transitions legal and through defined operations?
- ☐ **Failure Resilience:** What happens if the network fails or the database is down? Is there retry logic, and is it bounded?
- ☐ **Hidden Coupling:** Does it import a class it shouldn't, or depend on an implementation detail?
- ☐ **Validation Evidence:** Did the AI also provide the tests that prove its work? Have you run them yourself?

37.3. From Draft to Verified Implementation

Consider a requirement: > *"The AI was asked to add retry logic to the delivery dispatch port."*

The AI provides a loop that retries on failure. A naive reviewer sees "it retries" and approves. A rigorous reviewer asks the following:

1. **Is the retry bounded?** (If not, we could have an infinite loop).
2. **Does it retry on *every* error?** (We should only retry on transient network errors, not on permanent domain errors like "invalid task ID").
3. **Is the operation idempotent?** (If the first request actually reached the server but the response was lost, does the retry cause a duplicate delivery?)
4. **Is the invariant preserved?** (Does the retry loop keep the task in the "DISPATCHING" state until it succeeds or finally fails?)

By asking these questions, you move from "it looks like it works" to "I have proven it is safe."

37.4. Implementation in Nex

In Nex, our explicit failure paths and contract checks provide the perfect framework for this kind of review.

```
-- The AI-generated proposal for a dispatch port with retry
class Dispatch_Port
feature
  send(task_id: String): String
    require
      id_present: task_id /= ""
    do
      result := "SENT"
    ensure
      known_result:
        result = "SENT" or
        result = "TRANSIENT_FAILURE" or
        result = "PERMANENT_FAILURE"
    end
end

class Dispatch_With_Retry
create
  make(port: Dispatch_Port) do
    this.port := port
  end
feature
  port: Dispatch_Port

  send_task(task_id: String, max_attempts: Integer): String
    require
      id_present: task_id /= ""
      attempts_valid: max_attempts >= 1
    do
      let attempt: Integer := 1
      let status: String := "TRANSIENT_FAILURE"

      from
      until attempt > max_attempts or
            status = "SENT" or
            status = "PERMANENT_FAILURE" do
        status := port.send(task_id)
        attempt := attempt + 1
      end

      result := status
    ensure
      known_result:
        result = "SENT" or
        result = "TRANSIENT_FAILURE" or
        result = "PERMANENT_FAILURE"
    end
end
```

A reviewer looking at this Nex implementation can immediately see the logic's boundaries. They can see that the loop is bounded by `max_attempts`. They can see that it stops correctly on a `PERMANENT_FAILURE`. The `ensure` clause makes the expected outcomes explicit. The review task is transformed from "understanding the code" to "validating the contract."

37.5. AI Review Across the Three Systems

In the **delivery system**, review focuses on state transition legality. Did the AI-generated code allow a task to skip a mandatory "verification" step?

In the **knowledge engine**, review focuses on ranking correctness. Did the AI's "optimized" ranking algorithm accidentally exclude documents that were highly relevant but didn't match a new heuristic?

In the **virtual world**, review focuses on determinism. Did the AI introduce a non-deterministic random number call into a simulation loop that must remain reproducible?

In each case, the reviewer is the guardian of the system's core principles.

37.6. Three Ways AI Review Fails

Style-First Review. Spending 15 minutes debating a variable name while missing a fatal race condition is the most common review failure. The remedy is to follow the hierarchy of risk: contracts first, style last.

No Regression Focus. Assuming that because the new feature works, the old features still work. The remedy is to compare the observable behavior of the system before and after the patch, ideally using automated parity tests.

The "Rubber Stamp" Problem. Approving an AI-generated patch because you are in a hurry. The remedy is to acknowledge that AI code is "guilty until proven innocent." If you don't have time to review it properly, you don't have time to merge it.

37.7. Quick Exercise

Take one recent AI-generated patch (even a small one) and apply the Risk-First Hierarchy: 1. Check the contract (inputs/outputs). 2. Check the invariants (state safety). 3. Check the failure paths (what could go wrong?). 4. Check for hidden coupling.

Did you find anything that a "style-only" review would have missed?

37.8. Takeaways

- AI-generated code requires a more rigorous review than human code because its failure modes are often more subtle.

- Use a Risk-First Hierarchy: contracts and invariants are the top priority.
- Reviewing is not reading; it is validating behavior against expectations.
- A review checklist is the best tool for maintaining consistency and catching common AI optimistic biases.
- Review quality is the ultimate throttle on the speed of AI-assisted development.

Chapter 36 closes the book with the final piece of the puzzle: the role of human judgment and accountability in a world where AI is doing more and more of the work.

38. Human Judgment in an AI World

We have reached the end of our journey. We began in Part I by exploring the fundamental nature of complexity and why software systems so often fail to live up to our expectations. We journeyed through modeling, algorithms, data organization, and the discipline of building trustworthy systems. In this final part, we have looked at how these classic engineering principles intersect with the era of AI-assisted development.

There is a common anxiety that as AI becomes more capable, the role of the software engineer will diminish. If an AI can write the code, generate the tests, and even suggest the architecture, what is left for the human?

The answer is the most critical part of the process: **judgment and accountability.** An AI can assist with implementation, but it cannot own the outcome. It can generate alternatives, but it cannot decide which tradeoff is acceptable for *your* specific users, *your* specific business, and *your* specific ethical context. Final responsibility for the correctness, safety, and long-term quality of a system remains — and must remain — human.

38.1. What Human Judgment Owns

In an AI-assisted world, the engineer's role evolves from "author" to "governor." There are several domains of engineering that cannot be delegated to a model:

1. **Problem Framing:** AI is excellent at solving the problem you give it. It is terrible at deciding if you are solving the right problem. Identifying the true needs of a user and translating those into technical requirements is a human creative act.
2. **Architecture and Direction:** AI tends to suggest "local" optimizations. A human must ensure the system has a coherent "global" architecture that can grow over years, not just work for the next sprint.

3. **Risk Acceptance:** Every design choice involves tradeoffs. A human must decide when a risk is worth taking and when it is not.
4. **Policy and Ethics:** AI models reflect the data they were trained on, which may include biases or unsafe patterns. A human must define the ethical boundaries of the system — fairness, privacy, and safety — and ensure those boundaries are enforced.

38.2. Decision Quality in AI Workflows

In the AI era, the measure of an engineer is not how many lines of code they can write per hour, but the quality of the decisions they make. A strong engineering loop now looks like this:

1. **Intent and Constraints:** You define the intent of the change and the architectural constraints it must honor.
2. **Generation of Alternatives:** You use AI to rapidly generate several implementation approaches.
3. **Evaluation with Evidence:** You evaluate these alternatives based on evidence — contract checks, performance benchmarks, and security audits.
4. **Selection and Validation:** You choose the best approach and rigorously validate it within the system.
5. **Long-Term Governance:** You monitor the change in production and adapt it as requirements evolve.

This is still the same engineering loop we've discussed throughout this book. AI simply makes the iteration between steps 2 and 3 much faster.

38.3. Implementation in Nex: A Governance Log

In Nex, we can even formalize our decision-making through governance objects. These objects don't just execute logic; they record the *intent* and the *evidence* behind a decision.

```
-- Formalizing Human Decision-Making
class Rollout_Governance
create
  make(canary_success_rate: Integer, min_required_rate: Integer) do
    this.canary_success_rate := canary_success_rate
    this.min_required_rate := min_required_rate
    this.rollback_triggered := false
```

```
  end
feature
  canary_success_rate: Integer
  min_required_rate: Integer
  rollback_triggered: Boolean

  decide(): String
    require
      rates_valid: canary_success_rate >= 0 and min_required_rate >= 0
    do
      if canary_success_rate >= min_required_rate then
        rollback_triggered := false
        result := "PROCEED"
      else
        rollback_triggered := true
        result := "ROLLBACK"
      end
    ensure
      known_result: result = "PROCEED" or result = "ROLLBACK"
    end
invariant
  rate_bounds: canary_success_rate >= 0 and min_required_rate >= 0
end

class Engineering_Decision_Log
create
  make(decision: String, rationale: String, owner: String) do
    this.decision := decision
    this.rationale := rationale
    this.owner := owner
  end
feature
  decision: String
  rationale: String
  owner: String

  record(d, r, o: String): String
    require
      inputs_present: d /= "" and r /= "" and o /= ""
    do
      decision := d
      rationale := r
      owner := o
      result := "RECORDED"
    ensure
      persisted: decision = d and rationale = r and owner = o
    end
end
```

The point of this code is not its complexity, but its explicitness. It forces the human to state the criteria for success (`min_required_rate`) and to own the decision (`owner`). Even in an AI-assisted workflow, the record of *why* something happened is a human responsibility.

38.4. Human Judgment Across the Three Systems

In the **delivery system**, human judgment determines the balance between "efficiency" and "safety." Should we prioritize the fastest route or the safest one during a storm? AI can calculate both, but a human must set the policy.

In the **knowledge engine**, human judgment determines the balance between "relevance" and "diversity." Should we show the user exactly what they asked for, or should we include diverse perspectives? This is a product and ethical decision that no model can make on its own.

In the **virtual world**, human judgment determines the balance between "realism" and "performance." How much physical fidelity is required for a good user experience?

In every system, the most important parameters are the ones that only a human can set.

38.5. The Path Forward

The central thread of this book remains unchanged from the first page to the last:

- **Model clearly:** Understand the essence of the problem before you touch the code.
- **Specify behavior explicitly:** Use contracts and invariants to make your intent machine-checkable.
- **Measure and verify continuously:** Never assume a system is correct; prove it with evidence.
- **Evolve safely with accountable judgment:** Own the long-term health of your system through disciplined change.

These principles are the foundation of real-world software engineering. They were true when we wrote code by hand on punched cards, they were true during the rise of the internet, and they are even more true today as we collaborate with AI.

The tools will change. The languages will evolve. The assistants will become more capable. But the need for an engineer who can reason deeply about a system, define its boundaries, and take responsibility for its outcomes will never go away.

That engineer is you. Go forth and build systems that last.

38.6. Quick Exercise

Pick one AI-assisted change you've made recently. Write a 3-sentence "Decision Log" for it: 1. What was the core decision (e.g., "Adopted the ML-based ranking algorithm")? 2. What evidence did you use to justify it? 3. Who is accountable if the decision turns out to be wrong?

If you can't answer all three, your governance process needs more human judgment.

38.7. Takeaways

- AI accelerates implementation, but it cannot transfer accountability from the human to the machine.
- Human judgment is required for problem framing, architecture direction, risk acceptance, and ethical governance.
- Engineering maturity in the AI era is measured by the quality of your decisions and the evidence you use to support them.
- Reliable AI workflows require explicit constraints, rigorous review, and traceable decision-making.
- The core principles of this book — modeling, specification, verification, and disciplined evolution — are your most valuable assets in an AI-driven world.

This concludes ***Beyond Code — Building Software Systems That Last****. May your systems be trustworthy, your complexity be managed, and your engineering judgment be sharp.*

39. Epilogue — The Systems Behind Everything

We began this book with a simple premise: that software engineering is not about writing code, but about modeling the world. We explored this idea through the lens of three very different systems — a delivery network navigating the chaos of the physical world, a knowledge engine organizing the vastness of information, and a virtual world simulating the complexity of life.

Across these domains, we found that the problems were never just about syntax or algorithms. They were about *understanding*. They were about defining what a "delivery" actually is, what makes a document "relevant," and how a virtual entity "decides."

As we close this book, we find ourselves at a fascinating intersection. We have the timeless principles of engineering — modeling, abstraction, verification — meeting the transformative power of AI. The question is no longer just "how do we build this?" but "how do we govern this?"

39.1. The Unifying Thread: Intent and Verification

Throughout the chapters, one tool appeared again and again: the **contract**.

In Part II, we used contracts to define the valid states of a delivery task. In Part VII, we used them to make our code trustworthy. In Part VIII, we used them to safely refactor complex logic. And finally, in Part IX, we discovered that contracts are the essential "prompt" for AI collaboration — the language we use to tell an assistant not just what to write, but what constraints it must honor.

This is not a coincidence. **Explicit intent** is the foundation of all robust systems. Whether you are communicating with a compiler, a junior developer, or a large language model, the clarity of your constraints determines the quality of the outcome.

The systems that failed in our stories were the ones where intent was implicit — hidden in the head of a developer or buried in a tangle of spaghetti code. The systems that succeeded were the ones where intent was explicit — written in `require` and `ensure` clauses, enforced by invariants, and verified by tests.

39.2. The Three Systems, One Lesson

The **Delivery Network** taught us that the real world is messy. Systems must be robust against failure, capable of retrying, and designed to handle invalid states gracefully.

The **Knowledge Engine** taught us that scale changes everything. Algorithms that work for a hundred items fail for a million. Efficient data structures and layered architectures are not optimizations; they are survival strategies.

The **Virtual World** taught us that emergent behavior is powerful but dangerous. We need boundaries — simulation loops, state constraints, and clear update rules — to keep complexity from becoming chaos.

In every case, the solution was not "more code." It was "better boundaries."

39.3. Programming in the Age of AI

We end this book in a new era. AI tools have dramatically lowered the cost of generating code. But they have effectively *raised* the cost of not understanding it.

If you ask an AI to "build a delivery system" without understanding the domain, you will get a plausible hallucination. If you ask it to "optimize a search algorithm" without defining the invariants, you will get a subtle bug.

The engineer of the future is not a typist. The engineer of the future is a **verifier**. Your job is to: 1. **Model the domain** so clearly that even a machine can understand the rules. 2. **Define the boundaries** so strictly that invalid states are impossible. 3. **Review the output** with the skepticism of a scientist, looking for the edge cases that the model missed.

AI is a powerful engine, but you are the steering wheel.

39.4. The Role of Nex

We chose Nex for this book not because it is the only language that matters, but because it embodies the principles we value: - **Design by Contract:** Making constraints visible and executable. - **Static and**

Dynamic Typing: Offering flexibility during exploration and safety during production. - **Readability:** Prioritizing clear, English-like syntax over cryptic brevity.

The lessons you learned here — about invariants, pre-conditions, seams, and layered architecture — will serve you in Python, Go, Rust, Java, or whatever language comes next. The syntax changes; the engineering does not.

39.5. Final Thoughts

The robot in the intersection moves safely because its constraints are enforced. The researcher finds the answer because the index is structured for retrieval. The virtual economy remains stable because its rules are invariant.

These outcomes didn't happen by accident. They happened because someone like you took the time to think, to model, and to verify.

Software is transient. It changes, it breaks, it gets rewritten. But **systems thinking** endures.

Go build systems that last.

40. Glossary

This glossary defines core terms used throughout the book.

40.1. Algorithm

A finite, explicit procedure that transforms inputs into outputs while meeting stated guarantees.

40.2. Asymptotic Complexity

A description of how runtime or memory grows as input size grows (for example: `O(1),O(log n),O(n),O(n log n),O(n^2)`).

40.3. Class Invariant

A condition that must hold for every valid object state of a class.

40.4. Contract

A precise behavioral agreement in code, usually expressed with preconditions (`require`), postconditions (`ensure`), and invariants.

40.5. Coupling

The degree to which one module depends on details of another.

40.6. Data Model

A structured representation of entities, relationships, and constraints in a problem domain.

40.7. Deduplication by Ancestor Class

When combining inherited items (such as invariants), each ancestor class contributes once even in multiple-inheritance graphs.

40.8. Encapsulation

A design principle that hides internal representation and exposes stable behavior through clear interfaces.

40.9. Feature (Method)

A callable behavior defined on a class.

40.10. Inheritance

A mechanism where a class reuses and extends behavior or structure from one or more parent classes.

40.11. Invariant

A property that must remain true across allowed state transitions.

40.12. Multiple Inheritance

A form of inheritance where a class has more than one immediate parent class.

40.13. Postcondition (`ensure`)

A condition that must be true after successful feature execution.

40.14. Precondition (`require`)

A condition that must be true before feature execution is valid.

40.15. Recursive Inheritance Merge

A rule where inherited contracts are collected transitively across parent classes.

40.16. Refactor

A structural code change that preserves externally visible behavior while improving design qualities.

40.17. Reliability

The ability of software to behave correctly under expected load and adverse conditions over time.

40.18. Scaling

The behavior of a system as data volume, traffic, or operational complexity grows.

40.19. Search Space

The set of candidate states or paths an algorithm may need to inspect.

40.20. Time Complexity

How execution steps grow as input size grows.

40.21. Transition Rule

A rule that defines which state changes are legal for an entity.

41. Index of Terms

This is a practical index of major topics and where to review them.

41.1. A

41.2. C

41.3. D

41.4. I

41.5. R

41.6. S

41.7. T

41.8. V

42. References

This bibliography is organized by each work's direct role in the book's themes.

42.1. Foundational Texts in Programming and Computation

- *Structure and Interpretation of Computer Programs* (Abelson et al. 1996)
- *How to Design Programs* (Felleisen et al. 2018)
- *The Art of Computer Programming* (Knuth 2011)

42.2. Algorithm Design and Analysis

- *Introduction to Algorithms* (Cormen et al. 2022)
- *The Algorithm Design Manual* (Skiena 2020)
- *Algorithms* (Sedgewick and Wayne 2011)

42.3. Software Design and Architecture

- On module decomposition and information hiding (Parnas 1972)
- *Object-Oriented Software Construction* (Meyer 1997)
- *Clean Architecture* (Martin 2017)
- *Domain-Driven Design* (Evans 2003)
- *Patterns of Enterprise Application Architecture* (Fowler 2002)

42.4. Functional Programming and Type Systems

- Why functional style matters (Hughes 1989)
- Theorems from parametric polymorphism (Wadler 1989)

42.5. Graph Theory and Discrete Mathematics

- *Graph Theory* (Diestel 2017)
- *Discrete Mathematics and Its Applications* (Rosen 2018)

42.6. Testing and Verification

- Notes on structured programming and limits of testing (Dijkstra 1972)
- Hoare logic foundations (Hoare 1969)
- *Test-Driven Development: By Example* (Beck 2002)

42.7. Data Modeling

- *An Introduction to Database Systems* (Date 2003)
- *Designing Data-Intensive Applications* (Kleppmann 2017)

42.8. Influence Note

Works are cited where their influence is most direct. Several shape the book across multiple parts.

42.9. Notes

- Add chapter-local citations as needed using `[@citation_key]`.
- Rebuild with `make book` or `make print` to refresh bibliography output.

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. MIT Press.

Beck, Kent. 2002. *Test-Driven Development: By Example*. Addison-Wesley.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.

Date, C. J. 2003. *An Introduction to Database Systems*. 8th ed. Addison-Wesley.

Diestel, Reinhard. 2017. *Graph Theory*. 5th ed. Springer.

Dijkstra, Edsger W. 1972. "Notes on Structured Programming." In *Structured Programming*, edited by O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. Academic Press.

Evans, Eric. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Felleisen, Matthias, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2018. *How to Design Programs: An Introduction to Programming and Computing*. 2nd ed. MIT Press.

Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Hoare, C. A. R. 1969. "An Axiomatic Basis for Computer Programming." *Communications of the ACM* 12 (10): 576–80.

Hughes, John. 1989. "Why Functional Programming Matters." *The Computer Journal* 32 (2): 98–107.

Kleppmann, Martin. 2017. *Designing Data-Intensive Applications*. O'Reilly Media.

Knuth, Donald E. 2011. *The Art of Computer Programming*. Addison-Wesley.

Martin, Robert C. 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. 2nd ed. Prentice Hall.

Parnas, David L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12): 1053–58.

Rosen, Kenneth H. 2018. *Discrete Mathematics and Its Applications*. 8th ed. McGraw-Hill.

Sedgewick, Robert, and Kevin Wayne. 2011. *Algorithms*. 4th ed. Addison-Wesley.

Skiena, Steven S. 2020. *The Algorithm Design Manual*. 3rd ed. Springer.

Wadler, Philip. 1989. "Theorems for Free!" *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*.