Testing Like a Senior

Patterns & Anti-Patterns Across Software QA Career Levels

modernQAcourse.com

THE MODERN QA ENGINEER SKILLSET

Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

The QA Engineer's Field Manual

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

The QA Engineer's Field Manual

Book 26 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

The QA Engineer's Field Manual: A Practical, Opinionated Guide to Testing in the Real World

Edition: First Edition, February 2026

About This Book

This book will make you uncomfortable.

Not because the material is difficult – it is not. But because it will hold a mirror up to habits you have spent years building, and it will tell you that many of them are wrong. Not slightly suboptimal. Not "could be improved." Wrong. Counterproductive. Actively harmful to the quality of the software you ship and the trajectory of your career.

Most QA engineers learn their craft through a combination of online courses, blog posts, conference talks, and on-the-job training. The problem is that most of these sources teach the same outdated playbook: write a login test first, automate everything you can, measure success by counting test cases, and retry flaky tests until they pass. This playbook was reasonable in 2015. In 2026, it is a recipe for building test suites that nobody trusts, automation frameworks that slow down releases, and careers that plateau at "senior automation engineer" with nowhere left to go.

This book offers a different path. It is organized not around tools or technologies but around **career maturity levels** – because your relationship with testing changes fundamentally as you grow. A junior engineer needs to learn how to write reliable tests. A mid-level engineer needs to learn when *not* to write tests. A senior engineer needs to learn how to

make testing a strategic asset that influences architecture, reduces risk, and accelerates the entire engineering organization.

At each level, we examine the **anti-patterns** that keep engineers stuck and the **patterns** that propel them forward. Every chapter presents a real problem, explains why the common solution is wrong, and offers a better alternative grounded in engineering principles rather than folklore.

This is not a tutorial. There are no step-by-step instructions for installing Selenium. There are no screenshots of CI dashboards. What there is: a direct, sometimes blunt conversation about what separates QA engineers who write tests from QA engineers who build quality into systems.

About the Modern QA Course Series

This book is part of the **Modern QA Course** – a series of 25 comprehensive textbooks covering every area of knowledge expected of a professional QA engineer in 2026. The full series is available at modernQA-course.com and covers topics ranging from test automation frameworks and API testing to performance engineering, security testing, CI/CD integration, and career strategy.

Each book in the series is a deep dive into a specific domain. Together, they form a complete curriculum for quality engineering professionals at every career stage:

- **Books 1-10** cover core technical competencies: browser automation, API testing, test design, performance, security, mobile testing, database testing, and more
- **Books 11-20** cover foundational engineering skills: version control, Linux, networking, containers, cloud infrastructure, and monitoring
- **Books 21-24** cover professional skills: communication, leadership, process design, and team dynamics
- **Book 25** covers interview preparation and career strategy

This book – the 26th – was written to be read *first*. It is the field manual, the big picture, the strategic overview that shows you where all the pieces fit. Think of it as the map before the territory. Read this book to understand what matters, why it matters, and where you stand. Then use the other 25 books to go deep in the areas where you need to grow.

Who This Book Is For

This book is for anyone who tests software for a living and suspects there might be a better way.

More specifically:

- **Junior QA engineers** who want to avoid the most common career traps from day one
- **Mid-level automation engineers** who have built test suites but sense that something is off – tests are passing, but quality is not improving
- **Senior engineers** who are technically strong but struggling to articulate their impact or move into leadership
- **Test leads and managers** who want a shared vocabulary of patterns and anti-patterns for their teams
- **Developers who write tests** and want to understand what separates adequate test coverage from genuine quality engineering
- **Career changers** entering QA who want to skip the years of trial-and-error learning

You do not need experience with any specific tool. The patterns in this book apply whether you use Playwright, Cypress, Selenium, or something that does not exist yet. Tools change. Thinking scales.

Prerequisites

- At least some exposure to software testing (manual or automated)
- Basic understanding of what a test suite is and how CI/CD pipelines work
- Willingness to question your current practices

That is all.

How to Use This Book

Structure

This book is organized into **six levels of career maturity**, each containing multiple chapters. The levels build on each other, but the book is designed to be useful no matter where you are in your career.

Level	Title	Chapters	Who Benefits Most
0	Entry Shock	1	Engineers in their first two weeks at a new company
1	Test Executor	2-5	Junior engineers writing their first automation
2	Automation Engineer	6-9	Mid-level engineers building and maintaining test suites
3	Quality Engineer	10-13	Senior engineers thinking about risk and strategy
4	Systems Thinker	14-17	Staff-level engineers influencing architecture
5	QA Leader	18-22	Leads, principals, and managers building organizations
Advanced	Deep Dives	23-24	Anyone curious about AI in testing and the future of QA

Reading Approaches

If you are just starting your career: Read the whole book front to back. Everything ahead of your current level is a preview of where you are going. Everything at your level is immediately actionable.

If you are mid-career: Start at Level 2 or 3. Read forward, then circle back to Level 0 and 1 to see which anti-patterns you have unconsciously adopted.

If you are a team lead: Read the whole book, then assign specific chapters to specific team members based on where they are struggling. Use the "Core Question" at the end of each level as a coaching prompt.

Conventions

Throughout this book, you will encounter several types of callout boxes:

> **Anti-Pattern:** A common practice that feels right but produces bad outcomes. These are the habits this book exists to break.

> **Pattern:** The better alternative. A practice grounded in engineering principles that produces reliable, scalable results.

> **War Story:** A real-world scenario (anonymized) that illustrates the pattern or anti-pattern in action.

> **Core Question:** A self-assessment prompt at the end of each level that helps you gauge whether you have truly internalized the material or are still operating at a lower level.

Table of Contents

LEVEL 0

ENTRY SHOCK

CHAPTER 1

The First Two Weeks: Your Reputation Is Being Built

Why This Chapter Comes First

Most QA books start with test design techniques or tool installation guides. This one starts with your first day at work. Here is why.

The technical skills you need to succeed as a QA engineer are learnable. Any motivated person can pick up Playwright in a week, learn API testing in a month, and become competent with CI/CD pipelines in a quarter. What is not so easily learned – and what determines more of your career trajectory than any tool – is how you establish yourself in a new organization. The habits you form in your first two weeks at a new job will shape how your team perceives you for years.

Every senior engineer has seen it: the new hire who dives into writing tests on day one, breaks the build on day three, and spends the rest of their first month apologizing. They were eager. They were skilled. But they skipped the most important step in any new role: understanding the system before trying to change it.

This chapter is about that step. Not the system as in the application under test – though that matters too – but the *entire* system: the code, the people, the processes, the politics, the unwritten rules, and the cultural signals that tell you what this organization actually values versus what it claims to value.

System Reconnaissance

Your first task in any new role is not to write code. It is to gather intelligence.

Think of your first two weeks as an anthropological field study. You are observing a complex system – part technical, part social – and your job is to build an accurate mental model of how it works before you attempt to contribute. This is not passive. Good reconnaissance is deliberate, structured, and exhausting.

The Codebase

Before you open a single test file, you need to understand what you are testing. This sounds obvious, but the number of QA engineers who start writing tests against an application they barely understand is staggering.

Start with architecture. Ask your manager or a senior developer for a system architecture diagram. If one does not exist – and it often does not – that is your first data point about the organization's documentation culture. Sketch one yourself. Identify the major services, their responsibilities, how they communicate (REST, GraphQL, message queues, gRPC), and where the data lives. You do not need to understand every endpoint on day one. You need the big picture.

Then move to the codebase itself. Clone the main repositories. Read the README files – if they exist. Look at the directory structure. Identify the tech stack: what languages, what frameworks, what build tools. Find the test directories. Are there unit tests? Integration tests? End-to-end tests? How many? When were they last modified? Are there tests that have been commented out? Tests with names like `test_login_BROKEN_DO_NOT_RUN`? Every detail tells a story.

In 2026, you have AI-powered tools that can accelerate this process dramatically. Feed the repository to an AI coding assistant and ask it to summarize the architecture, identify the main entry points, map the API endpoints, and list the external dependencies. This is not cheating. This is using the tools available to you. A senior engineer who joined the same team would do exactly the same thing.

The API and UI Structure

Map the application from the user's perspective. Open the application and use it. Click through every major flow. Submit forms. Break things. Note which flows are smooth and which feel fragile. Pay attention to error handling – or the lack of it.

Then map the API surface. If there is an OpenAPI/Swagger specification, read it cover to cover. If there is not, use browser DevTools to observe the network traffic as you use the application. Build a mental catalog of the main API endpoints, their request/response shapes, and their authentication requirements.

This dual perspective – UI behavior and API contracts – is your testing foundation. Everything you do later builds on this understanding.

The CI/CD Pipeline

Find the CI/CD configuration files. They are usually in the root of the repository: `.github/workflows/`, `Jenkinsfile`, `.gitlab-ci.yml`, `bitbucket-pipelines.yml`, or similar. Read them carefully.

What triggers a build? Every push? Only pull requests? Only merges to `main`? What stages does the pipeline have? Which tests run where? How long does a full pipeline take? Is there a separate nightly test run? What happens when tests fail – does the build break, or does someone get a notification they ignore?

The CI/CD pipeline is the truth about what the organization actually tests, as opposed to what it says it tests. If the pipeline only runs unit tests and skips integration tests, that tells you where the gaps are. If the pipeline takes 45 minutes and developers routinely skip it, that tells you about the friction in the development process.

Flaky Test Culture

Every organization has a relationship with flaky tests. Your job is to figure out what it is.

Ask these questions, directly or by observation:

- How many tests fail on any given run without a code change?
- Does the team have a process for investigating flaky failures, or do they just re-run the build?
- Is there a "quarantine" list of known-flaky tests?

- When was the last time someone fixed a flaky test versus disabling it?
- Do developers trust the test results, or do they merge despite failures?

The answers reveal more about the organization's quality culture than any mission statement. A team that actively investigates and fixes flaky tests is a team that takes quality seriously. A team that has a "just re-run it" culture is telling you that their test infrastructure is a liability, not an asset.

Social Architecture Mapping

The technical system is only half the picture. The other half is the people.

Who Makes Decisions

Every organization has a formal hierarchy and an informal power structure, and they are rarely the same thing. The org chart says who reports to whom. The informal structure says who actually influences what gets built, how it gets tested, and what gets shipped.

Identify the key players:

- **The architects and tech leads** who shape technical decisions. Their opinions about testing will constrain what you can do.
- **The product managers** who decide what gets built. Their priorities determine what needs testing most urgently.
- **The other QA engineers** (if they exist). What is their approach? What tools do they use? What is their relationship with the development team? Are they embedded in feature teams or centralized?
- **The developers who care about testing.** Every team has at least one developer who writes thorough tests and values quality. Find that person. They will be your most important ally.
- **The developers who do not care about testing.** Every team has these too. They push code without tests, resist code reviews, and view QA as an obstacle. Do not fight them on day one. Understand their perspective first.

Cultural Signals

Pay attention to the small things. They reveal the culture more accurately than any company values poster.

Do developers write unit tests, or is testing entirely a QA responsibility? When a bug reaches production, is the reaction "let us figure out how this escaped our process" or "QA missed this"? Are QA engineers included in design reviews and architecture discussions, or are they brought in after the feature is built? Do sprint demos include test coverage alongside feature demos?

These signals tell you whether quality is a shared value or a department. That distinction shapes everything about your role.

The Unwritten Rules

Every workplace has rules that nobody writes down but everyone follows. Some examples from QA:

- "We do not block releases for test failures in the nightly run" (meaning the nightly run is informational, not gating)
- "We write tests for new features but not for bug fixes" (meaning regression coverage is growing more slowly than technical debt)
- "The mobile team does their own testing" (meaning there is a testing silo you need to navigate)
- "We use Selenium but we are thinking about migrating to Playwright" (meaning there is political context around tooling decisions)

You discover these rules by listening more than talking in your first two weeks. Attend every meeting you are invited to. Read Slack channels. Look at pull request comments. Read old Jira tickets and their discussions.

AI-Assisted Codebase Digestion

In 2026, one of the most powerful tools in your reconnaissance kit is an AI coding assistant. Used correctly, it can compress weeks of codebase exploration into days.

Here is a structured approach to using AI for codebase understanding:

Step 1: Repository Summary. Feed the repository (or a representative sample of files) to an AI assistant and ask: "Summarize the architecture

of this application. What are the main services, their responsibilities, and how do they communicate?"

Step 2: Test Infrastructure Audit. Ask: "Analyze the test directory. What test frameworks are used? How are tests organized? What is the ratio of unit to integration to end-to-end tests? Are there any tests that appear to be disabled or broken?"

Step 3: API Surface Mapping. Ask: "List all API endpoints in this application, organized by domain. Include the HTTP method, path, and a brief description of each."

Step 4: Dependency Analysis. Ask: "What are the external dependencies of this application? Which third-party APIs does it call? What databases does it use? What message queues or event systems are involved?"

Step 5: Risk Identification. Ask: "Based on the codebase, what areas appear to be the highest risk? Where is the code most complex? Where is test coverage thinnest? Where are the most recent bug fixes concentrated?"

This is not a replacement for reading the code yourself. It is an accelerant. The AI gives you a map; you still need to walk the territory. But starting with a map is vastly better than wandering blind.

The 30-Day QA System Assessment

By the end of your first month, you should produce a document. Not because anyone asked you to – though some organizations will – but because it forces clarity and demonstrates initiative.

This document should contain:

Section 1: System Overview. A one-page summary of the application architecture, the tech stack, and the major user flows.

Section 2: Current Test Coverage. What types of tests exist? What frameworks are used? How are tests organized? What is the approximate coverage by test type (unit, integration, E2E)? Where are the obvious gaps?

Section 3: CI/CD Integration. How do tests integrate with the build pipeline? What is gating versus informational? What is the average build time? What is the flaky test rate?

Section 4: Quality Culture Assessment. How does the team think about quality? Is testing a shared responsibility or a QA-only concern?

What is the team's relationship with flaky tests? How are bugs triaged and prioritized?

Section 5: Risk Map. Based on your first month of observation, what are the highest-risk areas of the application? Where are bugs most likely to escape to production? Where would additional test coverage have the highest impact?

Section 6: Recommended Priorities. What should you focus on in your first quarter? Be specific. "Improve test coverage" is not a priority. "Add API-level integration tests for the payment processing flow, which currently has zero automated coverage and has produced three production incidents in the last quarter" is a priority.

This document serves three purposes. First, it forces you to synthesize everything you have learned into a coherent picture. Second, it gives your manager concrete evidence that you spent your first month strategically rather than aimlessly. Third, it becomes the foundation for your first quarter's work plan.

> **War Story:** A QA engineer at a fintech company produced a 30-day assessment document that identified a gap in testing around a specific API endpoint handling currency conversion. The document included data on three recent production bugs, all traced to that endpoint. The engineering manager was so impressed that the QA engineer was asked to present the findings to the entire engineering organization. Within six months, she was promoted to senior. The document took about four hours to write. The career impact lasted years.

The Reputation Framework

Your reputation in a new organization is built in concentric circles.

Week 1: Your immediate team. They see everything: when you arrive, what questions you ask, how you respond to feedback, whether you listen more than you talk. In the first week, optimize for one thing: demonstrating that you are thoughtful and observant. Ask good questions. Take notes. Do not volunteer opinions about what should change until you understand why things are the way they are.

Week 2-4: The broader engineering team. Your first pull requests, your first bug reports, your first test plans – these form the second circle. Quality matters enormously here. A single sloppy PR or a bug report that wastes a developer's time can set your reputation back weeks. Conversely,

a well-written bug report that saves a developer an hour of debugging builds trust fast.

Month 2-3: Leadership and cross-functional partners. By this point, your manager is forming opinions about your trajectory. Product managers are noticing whether you ask insightful questions about requirements. Designers are noticing whether you catch UX issues that others miss. These impressions solidify quickly and change slowly.

The core lesson: **before writing tests, understand the system.** The system is not just the software. It is the people, the processes, the politics, and the culture. Understanding all of it is not optional. It is the foundation on which everything else in this book – and in your career – is built.

> **Core Level 0 Question:** When you start a new role, do you rush to prove yourself by producing output – or do you invest in understanding the full system first?

Career Translation

Resume phrasing

- Produced a 30-day QA system assessment in the first month at a new company, identifying three critical coverage gaps in the payment processing flow and two architectural risks, which became the foundation for the team's quarterly test strategy.
- Used AI-assisted codebase analysis to map 12 microservices, 180 API endpoints, and the existing test infrastructure within the first two weeks, compressing what typically takes a quarter into days.
- Established credibility with the engineering team by delivering a structured risk map and recommended priorities before writing a single test, resulting in alignment on the first quarter's quality engineering goals.

Cover letter framing

I believe the most important thing a QA engineer does in a new role happens before they write a single test. In my first two weeks at any company, I conduct a structured reconnaissance: mapping the architecture, auditing the test infrastructure, assessing the CI/CD pipeline, and understanding the team's quality culture. By the end of my first month, I deliver a system

assessment that identifies risks, coverage gaps, and recommended priorities. This deliberate approach builds credibility fast and ensures my first tests target what actually matters.

Interview framing

"I treat my first two weeks like an anthropological field study. I map the technical system – architecture, APIs, test infrastructure, CI pipeline – and the social system – who makes decisions, what the team values, what the unwritten rules are. I use AI tools to accelerate codebase understanding but always validate by walking the territory myself. By day thirty, I deliver a written assessment that shows I understand the system deeply enough to contribute strategically, not just tactically."

What not to say

- "I like to dive right in and start writing tests on day one" – eagerness without understanding leads to wasted effort and broken builds
- "I just follow whatever the team is already doing" – passive adoption does not demonstrate strategic thinking
- "I spent my first month getting set up" – sounds passive; the reconnaissance should be active and produce a tangible artifact
- "I don't need to understand the business, I just need to test the code" – testing without business context means testing the wrong things
- "I asked my manager what to test" – shows dependency rather than initiative; a QA engineer should proactively assess risk

Interview Depth Check

Question 1

Prompt: You start a new job on Monday. Walk me through your first two weeks.

What a strong answer should cover:

- Week 1: Technical reconnaissance (architecture, codebase, test infrastructure, CI/CD pipeline)
- Week 1: Social reconnaissance (key stakeholders, decision makers, quality culture)

- Week 2: Deeper exploration (flaky test culture, unwritten rules, risk areas)
- Week 2: Begin drafting the 30-day assessment document

Example answer:

- "Day 1-2: I get access to repositories, clone the main codebases, and use an AI assistant to generate an architecture summary. I identify the tech stack, the test frameworks, and the CI/CD configuration. I also ask my manager for an architecture diagram – or note that one does not exist."
- "Day 3-5: I map the application from the user's perspective: major flows, API surface, error handling patterns. I read the CI pipeline configuration to understand what is actually tested versus what the team says is tested. I attend every meeting I am invited to and mostly listen."
- "Week 2: I deep-dive into the test suite: coverage gaps, flaky test rate, skipped tests, test organization. I identify the key stakeholders and their relationship with quality. I start drafting my 30-day assessment."
- "By day 14, I have a mental model of the full system – technical and social – and a draft document that will become my recommended priorities for the first quarter."

Question 2

Prompt: How do you use AI tools to accelerate your understanding of a new codebase?

What a strong answer should cover:

- Repository summary (architecture, services, communication patterns)
- Test infrastructure audit (frameworks, coverage, broken or skipped tests)
- API surface mapping (endpoints, methods, dependencies)
- Risk identification (complex code, thin coverage, recent bug concentrations)

- AI as accelerant, not replacement for human understanding

Example answer:

- "I use a structured five-step approach. Step 1: Feed the repository to an AI and ask for an architecture summary – main services, their responsibilities, how they communicate. Step 2: Audit the test directory – frameworks used, test organization, ratio of unit to integration to E2E tests, any disabled or broken tests."
- "Step 3: Map the API surface – all endpoints organized by domain with methods and descriptions. Step 4: Analyze external dependencies – third-party APIs, databases, message queues. Step 5: Identify risk areas – complex code, thin test coverage, concentrated bug fixes."
- "This gives me a map in hours rather than weeks. But I always validate the AI's output by exploring the actual code, because AI can miss nuances like deprecated endpoints or dead code that is still tested."

Question 3

Prompt: Describe what your 30-day system assessment document looks like and how you use it.

What a strong answer should cover:

- Six sections: system overview, current test coverage, CI/CD integration, quality culture, risk map, recommended priorities
- Specific and actionable, not generic
- Serves three purposes: forces synthesis, demonstrates initiative, becomes a work plan
- Shared with manager and used to align on first-quarter goals

Example answer:

- "The document has six sections. System overview: a one-page architecture summary. Current test coverage: what types of tests exist, what frameworks are used, where the gaps are. CI/CD integration: what is gating versus informational, pipeline runtime, flaky rate."

- "Quality culture assessment: is testing a shared responsibility or a QA silo? How does the team handle flaky tests? How are bugs triaged? Risk map: highest-risk areas based on bug history, code complexity, and business impact. Recommended priorities: specific, actionable items for my first quarter."
- "A good priority is not 'improve test coverage.' It is 'Add API-level integration tests for the payment processing flow, which has zero automated coverage and produced three production incidents last quarter.'"
- "I share the document with my manager and use it to align on first-quarter goals. It demonstrates that I spent my first month strategically, and it gives my manager concrete evidence of my assessment skills."

LEVEL 1

TEST EXECUTOR

This is where most automation courses stop.

About This Sample

You have just read **Chapter 1 of Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-26-testing-like-a-senior/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.