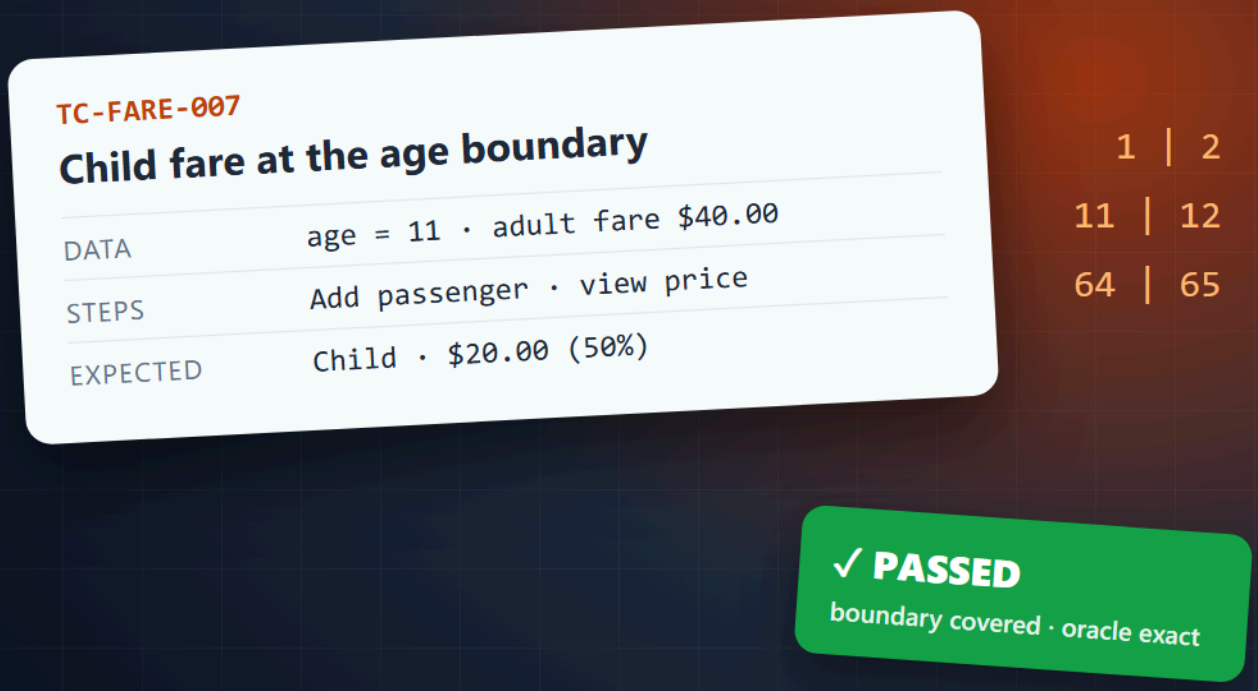


THE PRACTICAL GUIDE

Write Effective Test Cases

A Practical Guide to Test Design for QA Engineers and Developers

Partitions · Boundaries · Decision tables · State transitions
Pairwise · Exploratory · API · UI · Automation

ANALYZE · DESIGN · EXECUTE · REVIEW · MAINTAIN

THENEXBOOKS

Write Effective Test Cases

A Practical Guide to Test Design for QA Engineers and Developers

First Edition

THENEXBOOKS

Write Effective Test Cases
A Practical Guide to Test Design for QA Engineers and Developers

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Test case templates, checklists, and code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Tools and standards evolve; always check current documentation for the tools you use. The publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Harborline and all people, companies, routes, and data associated with it are fictional. Product names such as pytest, Hypothesis, Playwright, Appium, Cucumber, k6, Locust, and axe are trademarks of their respective owners and are used for identification only.

Preface

Almost every software team writes test cases, and almost every team has a test suite it does not fully trust. Some cases say "verify the page works". Some check twenty things at once and fail for reasons nobody can explain. Some were written for a screen that was redesigned two years ago. Some pass every night while customers report the exact defect they were supposed to catch. The problem is rarely effort. It is **design**.

An effective test case is small, precise, and purposeful. It targets a specific risk, uses data chosen on purpose, and states an expected result that leaves no room for interpretation. A good set of test cases is not the largest possible set; it is the smallest set that gives you real confidence — and it tells you what it does *not* cover.

This book teaches you how to write such test cases, systematically. We start with the anatomy of a test case and how to derive it from requirements, then work through the classic test design techniques — equivalence partitioning, boundary value analysis, decision tables, state transitions, pairwise testing, and scenarios — before moving to experience-based and white-box techniques. We then apply them at every level, from unit tests to API, UI, mobile, performance, security, and accessibility, and finish with the practices that keep a suite healthy: test data, risk-based prioritization, traceability, reviews, maintenance, and automation, including the responsible use of AI assistants.

Who This Book Is For

- **Manual and exploratory testers** who want a systematic way to design and document tests.
- **Automation engineers and SDETs** who want their automated checks to test the right things.
- **Developers** who write unit, API, and end-to-end tests and want them to find real defects.
- **Test leads, QA managers, and business analysts** who plan, review, and report on testing.

What You Will Learn

- What makes a test case effective, and how to write clear steps and precise expected results.
- How to analyze requirements, find gaps early, and derive test conditions.
- Equivalence partitioning, boundary value analysis, decision tables, state transition, pairwise, and scenario-based testing.
- Error guessing, exploratory testing, white-box coverage, data-driven and property-based tests.
- Test cases for units, APIs, UIs, mobile apps, performance, reliability, usability, security, and accessibility.
- Test data, risk-based prioritization, regression suites, traceability, metrics, reviews, maintenance, automation, and AI assistance.
- Two complete case studies you can adapt to your own product.

How This Book Is Organized

Part I — Foundations of Test Cases (Chapters 1–5) covers what test cases are, their anatomy, analysis of the test basis, writing steps and expected results, and choosing a format. **Part II — Black-Box Test Design Techniques** (Chapters 6–11) covers the core specification-based techniques. **Part III — Experience-Based and White-Box Techniques** (Chapters 12–15) adds error guessing, exploratory testing, code coverage, and data-driven and property-based tests. **Part IV — Test Cases Across Levels and Layers** (Chapters 16–21) applies everything to unit, API, UI, mobile, non-functional, security, and accessibility testing. **Part V — Managing Test Cases** (Chapters 22–26) covers test data, prioritization, traceability, reviews, maintenance, and automation. **Part VI — Case Studies** (Chapters 27–28) brings it all together. The appendices provide a cheat sheet, templates, a review checklist, and a glossary.

A Note on Terminology and Tools

We use widely accepted testing terminology — test basis, test condition, test case, test oracle, coverage — consistent with common industry glossaries such as the ISTQB glossary and the ISO/IEC/IEEE 29119 series. You do not need any certification to use this book. Code examples use Python (pytest, Hypothesis) and Playwright because they are popular and readable, but the techniques apply to any language, framework, or test management tool.

The Running Example: Harborline

Throughout the book we test **Harborline**, a fictional ferry and coastal-transit ticketing platform with a web app, mobile apps, station kiosks, a crew boarding app, and a public API. Harborline has exactly the kind of rules that hide defects: fares that depend on passenger age, booking windows, vehicle lengths with oversize surcharges, promo codes, loyalty tiers, partial payment with points, seat holds that expire, time-based refund rules, weather cancellations, multiple languages, and offline boarding scans. Every technique in this book is applied to these rules, so you can see not only how a technique works but how the resulting test cases fit together.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, terminology, or a useful aside.

> **WARNING**
>
> A common mistake that lets defects slip through or makes a suite hard to maintain.

Test case IDs look like `TC-FARE-007`: an area code and a number. Code, data, and field names appear in `monospace`. Numbers labeled *hypothetical* or *illustrative* are examples, not measurements.

Contents

Preface

PART I — FOUNDATIONS OF TEST CASES

1 What Makes a Test Case Effective

2 Anatomy of a Test Case: Fields, Formats, and Templates

3 From Requirements to Test Conditions: Analyzing the Test Basis

4 Writing Clear Steps and Expected Results

5 Choosing a Format: Scripted Cases, Checklists, BDD Scenarios, and Charters

PART II — BLACK-BOX TEST DESIGN TECHNIQUES

6 Equivalence Partitioning

7 Boundary Value Analysis

8 Decision Tables and Cause-Effect Thinking

9 State Transition Testing

10 Combinatorial and Pairwise Testing

11 Use Case, User Story, and Scenario-Based Testing

PART III — EXPERIENCE-BASED AND WHITE-BOX TECHNIQUES

12 Error Guessing and Fault Attacks

13 Exploratory Testing with Charters and Session Notes

14 White-Box Test Design: Statement, Branch, and Condition Coverage

15 Data-Driven and Property-Based Test Cases

PART IV — TEST CASES ACROSS LEVELS AND LAYERS

16 Unit Test Cases Developers Actually Maintain

17 API Test Cases

18 UI and End-to-End Test Cases

19 Mobile and Cross-Platform Test Cases

20 Non-Functional Test Cases: Performance, Reliability, and Usability

21 Security and Accessibility Test Cases

PART V — MANAGING TEST CASES

22 Test Data Design and Management

23 Risk-Based Prioritization and Regression Suites

24 Traceability, Coverage, and Metrics That Matter

25 Reviewing and Maintaining Test Cases

26 Automating Test Cases and Using AI Assistants Responsibly

PART VI — CASE STUDIES

27 Case Study: Test Cases for Harborline Booking and Payments

28 Case Study: Rescuing a Legacy Test Suite

APPENDICES

A Test Design Cheat Sheet

B Templates

C Test Case Review Checklist

D Glossary

PART I

Foundations of Test Cases

What an effective test case is, what goes into one, how to derive it from requirements, and how to write steps and expected results that anyone can execute and judge.

1 · What Makes a Test Case Effective
2 · Anatomy of a Test Case
3 · From Requirements to Test Conditions
4 · Writing Clear Steps and Expected Results
5 · Choosing a Format

CHAPTER 1 · PART I — FOUNDATIONS OF TEST CASES

What Makes a Test Case Effective

A test case is a small, precise question you ask a system, together with the answer you expect. This chapter shows what separates a test case that finds defects and earns trust from one that merely fills a spreadsheet, and introduces Harborline, the ferry ticketing platform we test throughout the book.

A Test Case Is a Question with an Answer Key

Every team that tests software writes test cases, yet few teams agree on what one is. Some people mean a row in a spreadsheet. Others mean a function decorated with `@pytest.mark`. Others mean a paragraph in a wiki that says "check the booking page". All of these can be test cases, but only if they share a small set of essential parts.

In this book we use the widely accepted definition: a **test case** is a set of preconditions, inputs, actions, expected results, and postconditions, developed from one or more test conditions. Put more plainly, a test case says:

- **Where you start** — the state the system and its data must be in before you begin (preconditions).
- **What you do** — the actions you take and the data you feed in (steps and inputs).
- **What should happen** — the observable outcome that tells you whether the system behaved correctly (expected results).
- **Where you end** — the state the system should be left in (postconditions).
- **Why it exists** — the test condition, requirement, or risk it covers.

The expected result is the answer key. Without it, you are not running a test; you are taking a tour. A tour can be valuable — exploratory testing, covered in Chapter 13, is a disciplined form of it — but a tour does not let a different person, on a different day, reach the same pass or fail verdict. A test case does.

What a test case is not

It helps to be just as clear about what a test case is *not*, because many weak test cases are really something else wearing a test case's clothes.

- **It is not a restated requirement.** "Verify that child fares are 50% of the adult fare" repeats the rule. It does not say which child, which route, which fare, or what number should appear on the screen.

- **It is not a bug report.** A bug report describes a failure that already happened. A test case describes a check that can be run again and again, whether or not it has ever failed.
- **It is not a whole test plan.** "Test the payment module" is a goal or a scope statement. A test case is small enough to run in minutes and to produce a single verdict.
- **It is not a user manual.** Steps that explain every click to a new user, without any expected result, teach people how to use the product but do not test it.
- **It is not the same thing as automation code.** Code can *implement* a test case, and a good automated test is a test case in executable form. But a script that clicks through pages and asserts nothing meaningful is not a test case at all, no matter how green it turns.

Why We Write Test Cases

Test cases cost time to design, write, review, run, and maintain. They earn that cost back through three purposes. Keeping all three in mind is the simplest way to decide how much detail a test case needs.

1. Detect defects

The first purpose is to find problems before customers do. A test case that can never fail is useless for this purpose. An effective test case is designed so that a plausible defect — a wrong discount, an off-by-one boundary, a missing email — would cause it to fail visibly. Much of Part II of this book is about choosing inputs that have the highest chance of exposing such defects.

2. Provide evidence

The second purpose is to provide evidence about quality. When a release manager asks "can we ship the new refund policy?", a list of executed test cases with clear verdicts is evidence. When an auditor asks how you know that senior discounts are applied correctly, a traceable test case with recorded results is evidence. Evidence only counts if the test case is precise enough that its verdict means something. "Refunds work — Passed" is not evidence; "Cancellation 10 hours before departure refunds $50.00 of a $100.00 booking — Passed" is.

3. Document behavior

The third purpose is documentation. Requirements drift, people leave, and the knowledge of why the system behaves as it does fades quickly. A well-written test case records the agreed behavior in concrete terms: this input, this action, this result. Months later, when someone asks whether a 64-year-old passenger pays the adult or the senior fare, a test case can answer faster than a search through old meeting notes.

These purposes pull in slightly different directions. Defect detection rewards cleverly chosen inputs. Evidence rewards traceability and recorded results. Documentation rewards readability. The effective test case balances all three for its context.

Meet Harborline

Abstract advice about test cases is easy to agree with and hard to apply. So throughout this book we test one realistic system: **Harborline**, a fictional ferry and coastal-transit ticketing platform. Harborline sells tickets through a web app, iOS and Android apps, self-service kiosks at stations, and a public REST API used by travel agents. Crew members use a separate crew app to scan tickets at the dock, even when the dock has no network connection.

Harborline was chosen because its business rules are rich, numeric, and full of boundaries — exactly the kind of rules that produce defects and that reward careful test design. Here are the rules we will use in every chapter. Fares and IDs are illustrative, and the currency is USD unless stated otherwise.

Fares by passenger type

Passenger type	Age	Fare rule
Infant	0–1	Free
Child	2–11	50% of adult fare
Adult	12–64	Full fare
Senior	65+	30% off adult fare

On the route from Northport to Gull Island, which we use as our default route, the adult fare is $40.00. That makes a child fare $20.00, a senior fare $28.00, and an infant fare $0.00. A family of two adults, a seven-year-old, and a one-year-old therefore pays $100.00.

Booking, vehicles, and promo codes

- A booking holds 1 to 9 passengers.
- Departures can be booked from 60 days ahead until 30 minutes before departure.
- A booking may add a car, motorbike, or bicycle. A car up to 5.5 m long is standard; 5.51–7.00 m is oversize and costs 40% more; a car over 7.00 m cannot be booked online.
- Promo codes are 6–12 uppercase alphanumeric characters. Only one code applies per booking. Some codes are limited to certain routes, and some expire.

Loyalty, payments, changes, and refunds

- **Harbor Points:** members earn 1 point per whole currency unit spent. Tiers are Blue (0–999 points), Silver (1,000–4,999), and Gold (5,000+).
- **Payments:** card, digital wallet, and Harbor Points, with 3-D Secure for cards. A booking can be paid partly with points.
- **Changes and refunds:** changes are free up to 24 hours before departure; cancelling between 24 and 2 hours before departure refunds 50%; cancelling under 2 hours before departure refunds nothing; a cancellation caused by weather always refunds 100%.

Booking states

A booking moves through a lifecycle. The main path is shown below; Chapter 9 turns it into a full state transition model.

Harborline booking lifecycle (main path)

```
Draft --> Held --> Paid --> Checked-in --> Boarded --> Completed
            |         |
            |         +--> Cancelled / Refunded
            +--> Expired   (seat hold lasts 15 minutes)
```

Harborline also offers seat maps, accessibility assistance requests, a user interface in English, Spanish, and Thai, tickets delivered by email and SMS with QR codes, and offline boarding scans. Each of these features appears in later chapters. You do not need to memorize the rules now; we will restate the relevant ones whenever we use them.

> **NOTE**
> Harborline's rules contain deliberate gaps and ambiguities — for example, how old is a passenger who turns 12 on the day of travel, and what is the refund when you cancel more than 24 hours ahead? Real requirements are like this. Chapter 3 shows how to find and resolve such questions before you write test cases.

A First Weak → Effective Pair

Here is a test case that could be found in almost any organization's test repository.

Weak — TC-FARE-001

```
Title:    Check child fare
Steps:    1. Go to the booking page
          2. Add a child
          3. Check the price
Expected: Price should be correct
```

It looks harmless, but it fails every purpose we just described. It is unlikely to detect a defect, because "correct" is left for the tester to decide on the day. It provides weak evidence, because two testers could run it with different children, different routes, and different ideas of "correct", and both would mark it passed. It documents nothing: a reader cannot learn from it what a child fare actually is. Here is an effective version.

Effective — TC-FARE-001

```
ID:            TC-FARE-001
Title:         Child aged 7 pays 50% of adult fare on one-way ticket
Covers:        Fare rule: Child (2-11) = 50% of adult fare
Preconditions: Route Northport -> Gull Island; adult fare $40.00
               Departure 10 days from today, seats available
```

```
Test data:     1 passenger, date of birth giving age 7 on departure date
Steps:         1. Start a one-way booking for the departure above
               2. Add one passenger with the test date of birth
               3. Continue to the price summary
Expected:      Passenger type shown as "Child"
               Fare line shows $20.00
               Booking total shows $20.00
Priority:      High
```

Notice what changed. The title states the condition and the outcome. The test data is concrete. The expected result is a number that anyone can compare with the screen. The link to the fare rule tells a reviewer why the case exists. None of this took special talent; it took the habit of asking, at each line, "could someone else run this and reach the same verdict?"

The Qualities of an Effective Test Case

There is no single perfect template, and Chapter 2 discusses several. But effective test cases share eight qualities, regardless of format. We will refer back to them throughout the book, and Appendix C turns them into a review checklist.

Quality	Question it answers
Clear	Can anyone read it the same way?
Atomic	Does it check one thing?
Traceable	Why does it exist?
Repeatable	Same inputs, same verdict?
Independent	Can it run alone, in any order?
Precise oracle	Is pass/fail unambiguous?
Maintainable	Is it cheap to update?
Risk-focused	Does it target what matters?

Clear

A clear test case uses plain words, consistent terms, and no private shorthand. If your team calls the same thing a "reservation", an "order", and a "booking", choose one term — Harborline uses *booking* — and use it everywhere. Clarity also means the reader knows the goal before reading the steps. A title such as "Refund test 3" forces the reader to reverse-engineer the purpose; "Cancellation 10 h before departure refunds 50%" does not.

Atomic

An atomic test case checks one test condition, or one tightly related group of observations about a single behavior. When it fails, you know what broke. A test case that books a trip, applies a promo code, pays with points, changes the date, and then cancels will fail for dozens of reasons, and the failure tells you almost nothing. Atomic does not mean trivial: one behavior may produce several observable results — a price, a message, an email — and it is fine to check all of them in one case. Atomic means one *reason* to exist.

Traceable

A traceable test case links to the thing it verifies: a user story, an acceptance criterion, a business rule, a risk, or a defect that must never return. Traceability lets you answer two questions quickly: "which tests cover this requirement?" and "which requirement does this failing test protect?" Chapter 24 covers traceability in depth; for now, remember that every test case should be able to say why it exists in one line.

Repeatable

A repeatable test case gives the same verdict every time it runs against the same system, regardless of who runs it. Repeatability dies when test data is vague ("use any senior passenger"), when dates are hard-coded and drift into the past, or when the result depends on timing. A Harborline test that books "the 09:00 sailing on 14 March" will stop working after 14 March. A test that books "a departure 10 days from today" keeps working, and still falls safely inside the 60-day booking window.

Independent

An independent test case does not rely on another test case having run first. It sets up what it needs, either through its own steps or through preconditions that can be established directly. Independence matters for manual testing, where testers pick cases in different orders, and it matters even more for automation, where tests run in parallel. A test case that says "use the booking created in TC-BKG-014" is a hidden chain; if TC-BKG-014 fails or is skipped, this one fails too, and the report shows two failures for one defect.

With a precise oracle

A **test oracle** is the source of the expected result — the means by which you decide whether an outcome is correct. A precise oracle turns the verdict into a comparison rather than an opinion. "$28.00" is precise. "Discount applied" is less precise — which discount, how much? "Looks right" is not an oracle at all. Chapter 4 is devoted to writing precise expected results and handling cases where a precise oracle is hard to obtain.

Maintainable

Systems change; test cases must change with them. A maintainable test case avoids details that do not matter to its purpose. If a test checks the senior discount, it should not depend on the exact label of a button three screens earlier. Maintainable test cases also avoid duplication: if the steps to create a paid booking appear in forty test cases, a change to checkout means forty edits. Referencing a shared precondition ("a paid booking exists for 2 adults") keeps the cost of change low.

Risk-focused

You can never test everything, so an effective test case targets something that matters: a rule that is easy to get wrong, a feature customers use constantly, a calculation involving money, or an area with a history of defects. Refund calculations at Harborline move real money, so they deserve careful cases at every boundary. The color of the footer does not. Chapter 23 shows how to prioritize systematically.

> **TIP**
> When reviewing a test case, read the title and the expected result first, then skip the steps. If those two lines alone tell you what is being checked and how to judge it, the test case is probably in good shape.

Common Weaknesses and How to Fix Them

Most weak test cases fail in one of four ways. Learning to spot them is the fastest way to improve a test suite, whether you wrote it or inherited it.

Weakness 1: Vague expected results

Vague expected results are the most common weakness and the most damaging, because they make every other quality irrelevant. A perfectly traceable, independent, atomic test case with the expected result "system works as expected" can never fail in a meaningful way.

Weak — TC-REF-004

```
Title:    Cancel booking
Steps:    Cancel a booking close to departure
Expected: Refund is processed correctly
```

Effective — TC-REF-004

```
Title:         Cancellation 10 h before departure refunds 50% of total
Preconditions: Paid booking, total $100.00 by card
               (2 adults $40.00 each, 1 child $20.00, 1 infant $0.00)
               Departure is 10 h from now
Steps:         1. Open the booking and choose "Cancel booking"
               2. Confirm the cancellation
Expected:      Refund amount shown as $50.00
```

```
                Booking status becomes "Cancelled"
                Refund of $50.00 recorded against the original card
                Cancellation email sent, stating the $50.00 refund
```

The effective version picks a moment that falls unambiguously in the 50% window, fixes the booking total so the refund can be computed in advance, and lists every observable consequence: the amount, the state, the payment record, and the email.

Weakness 2: Too many checks in one case

The opposite problem is a test case that tries to check everything. It often grows over time: someone adds "and also verify the points" to an existing case instead of writing a new one.

Weak — TC-BKG-020

```
Title: End-to-end booking test
Steps: 1. Search Northport -> Gull Island
       2. Add 2 adults, 1 senior, 1 car
       3. Select seats
       4. Apply promo code SPRING25
       5. Pay with card and 3-D Secure
       6. Check email ticket
       7. Change departure date
       8. Check Harbor Points
       9. Cancel booking
Expected: Everything works
```

If step 4 fails, steps 5 through 9 are never tested. If the case passes, you have no idea which of the many rules was actually verified, because there is no specific expected result for any of them. The fix is to split the case by behavior. Each new case gets its own preconditions and its own precise expected result:

- `TC-FARE-004` — Senior aged 70 pays $28.00 on the default route.
- `TC-VEH-002` — Standard car 4.6 m long is priced at the standard vehicle rate.
- `TC-PROMO-003` — Valid route-specific promo code reduces the total by its stated amount.
- `TC-PAY-006` — Card payment requiring 3-D Secure moves booking from Held to Paid.
- `TC-LOY-002` — Member paying $68.00 by card earns 68 Harbor Points.

There is still a place for a longer end-to-end journey — Chapter 18 discusses when it is worth it — but even then, every step that matters has its own expected result.

Weakness 3: Hidden dependencies

A hidden dependency is anything the test case needs but does not state. Common examples are data created by another test, a specific user account that "everyone knows", a feature flag, a time of day, or an environment setting.

Weak — TC-LOY-005

```
Title:    Silver tier upgrade
Steps:    1. Log in as the usual test member
          2. Book a trip
Expected: Member becomes Silver
```

Which member? How many points do they have now? The case only works if the "usual test member" happens to be just below 1,000 points, which will stop being true the first time the case passes.

Effective — TC-LOY-005

```
Title:         Member crossing 1,000 points is upgraded from Blue to Silver
Preconditions: Member account with 950 Harbor Points, tier Blue
               (created fresh for this test)
Test data:     Booking for 2 adults + 1 child + 1 infant on default route
               Total $100.00, paid fully by card
Steps:         1. Sign in as the member
               2. Complete the booking and payment
               3. Open the member dashboard
Expected:      Points earned for booking: 100
               Points balance: 1,050
               Tier shown: Silver
```

The dependency on account state is now explicit, and the case creates the account it needs, so it runs the same way every time.

Weakness 4: Happy path only

Many test suites contain dozens of cases showing that valid inputs work and almost none showing that invalid inputs are rejected safely. Defects cluster where the happy path ends: at boundaries, in error handling, and in unusual combinations. For every rule, ask what happens just inside, just outside, and far outside it.

For Harborline's passenger limit of 1–9 per booking, a happy-path suite might test 2 passengers and stop. An effective set also covers 1 passenger, 9 passengers, an attempt to add a tenth passenger, and an attempt to continue with zero passengers. For the car length rule, it covers 5.50 m, 5.51 m, 7.00 m, and 7.01 m. Chapters 6 and 7 give you systematic techniques — equivalence partitioning and boundary value analysis — for finding these values rather than guessing them.

Effective negative case — TC-BKG-011

```
Title:         Adding a tenth passenger is blocked with an explanatory message
Preconditions: Draft booking on default route with 9 adult passengers
Steps:         1. Choose "Add passenger"
Expected:      No tenth passenger form appears
               Message: "A booking can include up to 9 passengers."
               Passenger count remains 9; total remains $360.00
```

Test Conditions, Test Cases, and Test Procedures

Three terms are often used interchangeably, but separating them clarifies a great deal of test design work.

- A **test condition** is a testable aspect of the system: something that could be verified by one or more test cases. "Child fare is 50% of adult fare" is a test condition. So is "booking is rejected when departure is less than 30 minutes away". A test condition says *what* to test, not how.
- A **test case** turns a test condition into concrete preconditions, inputs, actions, and expected results. One test condition often needs several test cases — for the child fare, you might test ages 2, 7, and 11, and routes with different adult fares.
- A **test procedure** is a sequence of test cases, with any setup and teardown actions, in the order they are to be executed. A procedure might say: create a member account, run `TC-LOY-001`, then `TC-LOY-005`, then clean up. In automation, a test procedure often corresponds to a test script or suite.

Above all three sits the **test basis**: the body of knowledge — requirements, user stories, designs, regulations, and so on — from which test conditions are derived. Chapter 3 is about analyzing the test basis. The relationship looks like this:

From test basis to test procedure

```
Test basis        Harborline fare rules (story HL-102)
    |
    v
Test condition    TCOND-FARE-02: Child (2-11) pays 50% of adult fare
    |
    +--> TC-FARE-001  Child aged 7, adult fare $40.00 -> $20.00
    +--> TC-FARE-002  Child aged 2 on departure date -> $20.00
    +--> TC-FARE-003  Child aged 11 on departure date -> $20.00
    |
    v
Test procedure    "Fare regression, default route":
                  set up route data -> run TC-FARE-001..012 -> clean up
```

Why does the distinction matter? Because it separates two different kinds of thinking. Deciding *which* conditions deserve testing is an analysis activity; you do it with the product owner and developers, and you can review it before a single test case is written. Deciding *how* to test each condition — which values, which steps, which oracle — is a design activity. Mixing the two tends to produce test suites that are thorough on the parts that were easy to think about and silent on everything else.

> **NOTE**
>
> The vocabulary in this book follows common usage in the ISTQB syllabi and the ISO/IEC/IEEE 29119 family of software testing standards. You do not need either to use this book, but if your organization does, the terms will match.

The Cost of Poor Test Cases

It is tempting to treat test case quality as a matter of style: nice to have, but secondary to "actually testing". In practice, poor test cases cost teams in ways that are easy to recognize once named. We will not quote industry figures here; the costs below are ones you can observe in your own team, and Chapter 24 shows how to measure the ones that matter to you.

False confidence

The most serious cost is a green report that means nothing. When test cases have vague oracles, they pass even when the system is wrong. A release goes out with a senior discount of 25% instead of 30%, and the regression suite reported "Fares — 12 of 12 passed". The suite did not merely fail to help; it actively told people the product was fine.

Wasted execution time

Unclear steps slow testers down. Each ambiguous line causes a pause, a question to a colleague, or a guess. Duplicated test cases are executed twice. Tests with hidden dependencies fail for reasons unrelated to the product, and someone spends time finding out why before re-running them.

Noise and lost trust

Flaky or dependent tests produce failures that are not defects. After a few weeks of this, people stop reading failure reports carefully. The day a real defect appears, it is dismissed as "probably the test again".

Maintenance drag

Test cases full of incidental detail break whenever the interface changes. Teams then face a choice between spending days updating cases or quietly abandoning them. Abandoned test cases are worse than none, because they still appear in coverage reports.

Missed defects and weak evidence

Happy-path suites miss the defects that live at boundaries and in error handling. And when an auditor, a customer, or a regulator asks for evidence, a test suite of "works correctly — Passed" rows does not provide it. For a platform that handles payments and refunds, like Harborline, that gap can matter well beyond the test team.

> **WARNING**
>
> Do not respond to these costs by writing longer test cases. Length and quality are different things. A three-line test case with a precise oracle beats a thirty-step script ending in "verify everything is correct".