

COMPANION VOLUME • STAGE 09 OF THE ENGINEER'S SYLLABUS

# The Engineer's Craft

Everything before this taught you to make code work. This volume is about making code that a team can still change safely two years from now: how to name it, shape it, test it, change it, review it, debug it, and write about it.

---

| CHAPTERS | CODE EXAMPLES | EXERCISES | STAGE BUDGET |
|----------|---------------|-----------|--------------|
| 19       | 83            | 53        | 220 h        |

---

# Contents

|                                                           |            |
|-----------------------------------------------------------|------------|
| How to use this volume .....                              | 4          |
| <b>Part I · Code that reads well .....</b>                | <b>5</b>   |
| Chapter 01 · Naming .....                                 | 6          |
| Chapter 02 · Functions .....                              | 11         |
| Chapter 03 · Comments .....                               | 18         |
| Chapter 04 · Complexity and deep modules .....            | 23         |
| <b>Part II · Design .....</b>                             | <b>30</b>  |
| Chapter 05 · SOLID, read critically .....                 | 31         |
| Chapter 06 · Composition and dependency injection .....   | 38         |
| Chapter 07 · Design patterns worth knowing .....          | 44         |
| <b>Part III · Tests .....</b>                             | <b>52</b>  |
| Chapter 08 · What tests are for .....                     | 53         |
| Chapter 09 · Test doubles .....                           | 60         |
| Chapter 10 · TDD, properties, snapshots and mutants ..... | 66         |
| <b>Part IV · Changing code safely .....</b>               | <b>73</b>  |
| Chapter 11 · Refactoring .....                            | 74         |
| Chapter 12 · Legacy code .....                            | 83         |
| <b>Part V · Working with other people .....</b>           | <b>91</b>  |
| Chapter 13 · Code review .....                            | 92         |
| Chapter 14 · Version control and releasing .....          | 99         |
| <b>Part VI · When things go wrong .....</b>               | <b>105</b> |
| Chapter 15 · Systematic debugging .....                   | 106        |
| Chapter 16 · Error handling .....                         | 112        |
| Chapter 17 · Performance: measure first .....             | 120        |
| <b>Part VII · Beyond the code .....</b>                   | <b>126</b> |
| Chapter 18 · Technical writing .....                      | 127        |

|                                                        |     |
|--------------------------------------------------------|-----|
| Chapter 19 · Estimation, decomposition and scope ..... | 133 |
| The nineteen rules .....                               | 139 |
| Reading list .....                                     | 141 |

Every chapter has the same shape. It opens with its idea in one sentence, explains it through worked examples, lists the traps people fall into, and closes with exercises and a checklist you can apply to real code the same week.

The exercises matter more than the reading. Do them in your own editor, not in your head. Each has a solution folded underneath it: try first, then open it. Where your answer differs, decide whether it is worse or merely different. In this subject it is often the second.

Examples are in Python because it reads almost like pseudocode, but every idea transfers to other languages, and the text says so where a language feature changes the advice. Some early examples use floating-point numbers for money to stay short. Chapter 11 explains why real code must not.

### THE ONE QUESTION

Nearly everything in this volume answers one question: **how expensive will the next change be?**

Good names make the next change cheaper to understand. Deep modules let you change one thing without touching five. Tests make change safe, and refactoring stops its cost creeping upwards. Review, writing and estimation spread understanding across a team so no change depends on one person's memory.

When two pieces of advice conflict, and several in this book do, ask which one makes the next change cheaper for the next reader. That reader is usually you, six months from now, with none of today's context. *The Pragmatic Programmer* compresses the whole idea into three letters: ETC, easier to change.

## PART I

# Code that reads well

Code is read far more often than it is written. These four chapters are about the reader: the names they see first, the functions they step through, the comments that tell them what code cannot, and the shape of modules that let them understand one part without holding the whole system in their head.

# Naming

2 exercises

*A name is the smallest unit of design. A good one is a compressed explanation. A bad one is a small lie that every future reader has to see through.*

## Why names carry so much weight

When you open an unfamiliar file, names are what you read first and most. You skim identifiers to build a map before you read a single expression. A precise name lets the reader skip the body entirely. A vague one forces them into it. A misleading one sends them the wrong way with confidence.

Naming is hard because it forces you to decide what a thing *is*. If you can't name a function without the word "and", or you can only reach for "manager" or "helper", the difficulty is usually telling you something about the design rather than about your vocabulary. Treat naming trouble as a design signal.

## Principles that hold up

- **Name for intent, not for type or mechanism.** `pending_orders` tells the reader what they are holding. `order_list` tells them only the container, and `data` tells them nothing at all.
- **Make length proportional to scope.** A loop index that lives for two lines can be `i`. A constant read from forty files deserves a full, unambiguous name. Short names in wide scopes and long names in tiny ones are both mistakes.
- **Use the words your domain uses.** If the finance team says an invoice is *voided*, write `invoice.void()`, not `invoice.set_status(3)`. Domain-driven design calls this a ubiquitous language: the same terms in conversation, in tickets, and in code.
- **Put units in the name when the type doesn't carry them.** `timeout_seconds`, `price_cents`, `distance_km`. In 1999 NASA lost the Mars Climate Orbiter because one team's software reported thrust in pound-force seconds and another's read it as newton-seconds.
- **Name booleans as predicates.** `is_active`, `has_shipped` and `can_retry` read naturally after `if`. Avoid negatives: `if not is_disabled` makes every reader perform a double negation in their head.
- **Name functions for what they return or what they do.** Queries are nouns or questions: `total_price()`, `is_eligible()`. Commands are verbs: `send_invoice()`, `cancel_order()`. A `get_user()` that also creates a user when one is missing is lying. Call it `get_or_create_user()`.

- **One word per concept, one concept per word.** If `fetch`, `get`, `load` and `retrieve` all mean "read from the database" in one codebase, readers will assume they differ. Pick one. Equally, don't let `account` mean a login in one module and a billing entity in another.
- **Avoid words that say nothing.** `data`, `info`, `item`, `manager`, `handler`, `processor`, `helper`, `util`. Each is a placeholder for a decision not yet made. A `userManager` usually turns out to be three things: a repository, a password policy, and a notification sender.

## A worked example

Here is a function that works correctly and communicates nothing.

**BEFORE** pricing.py

```
def calc(d, t, f=False):
    r = 0
    for x in d:
        if x["s"] == "A" or f:
            r += x["p"] * x["q"]
    return r * (1 + t)
```

**AFTER** pricing.py

```
def order_total(line_items, tax_rate, include_inactive=False):
    subtotal = 0
    for item in line_items:
        if item["status"] == "ACTIVE" or include_inactive:
            subtotal += item["unit_price"] * item["quantity"]
    return subtotal * (1 + tax_rate)
```

Nothing about the logic changed, and the function can now be understood without its call sites. Notice what the rename exposes, too. A boolean argument that switches what a function does is usually two functions sharing a body; chapter 2 comes back to that. Renaming often reveals design problems that bad names were hiding.

## Renaming is cheap, so do it often

Modern editors rename safely across an entire codebase in seconds. Treat every name as a draft that improves as your understanding does. The moment you realise what a thing actually is, rename it, because otherwise that understanding is lost when you close the file.

The exception is names that escape your codebase: public API fields, database columns, event names, configuration keys. Those are contracts with other people, and chapter 14 covers how to change them.

## COMMON TRAPS

### Names that encode the whole implementation.

`is_least_relevant_multiple_of_larger_prime_factor` came up in a public exchange between John Ousterhout and Robert Martin. It is precise and nearly unreadable. When a name gets that long, the idea probably wants a comment or a better abstraction.

**Private abbreviations.** `cstmr_acct_bal` saves eight keystrokes and costs every reader a decoding step. Widely known abbreviations such as `id`, `url` and `http` are fine.

**Names that drift.** A `send_email` that now also writes an audit record has a stale name. Stale names are worse than vague ones, because readers trust them.

**Cute names.** `yeet_order()` is funny once, to one person.

## EXERCISES

### EXERCISE 1.1 Critique the names

Each name below has a problem. Say what it is and propose a better name. Assume a small online shop.

```
data = fetch_orders(status="pending")           # 1
flag = response.status_code in (502, 503, 504)   # 2
def process_user(user): ... # deactivates the account and emails the owner # 3
TIMEOUT = 30                                     # 4
if not is_not_verified: ...                      # 5
def get_customer(email): ... # inserts a new row when none is found         # 6
class OrderManager: ... # loads orders, prices them, sends receipts         # 7
newsletter_recipient_email_addresses_list = []   # 8
```

### SOLUTION

1. `data` says nothing. Use `pending_orders`.
2. The flag means "this failure is worth retrying". Use `is_retryable`.
3. `process` hides two actions. Name the behaviour, `deactivate_and_notify(user)`, and consider splitting it, because the "and" is a design smell.
4. The unit is missing. Use `TIMEOUT_SECONDS = 30`.
5. A double negative. Rename the variable `is_verified` and write `if is_verified:`.
6. The name hides a side effect. Use `get_or_create_customer(email)`.

7. Three responsibilities in one name. Split into an `OrderRepository`, a `price_order` function, and a `ReceiptSender`.
8. Too long, and the plural already implies a list. Use `newsletter_recipients`.

### EXERCISE 1.2 Rename without changing logic

Rename everything in this function so it can be understood without its call sites. Don't change the logic. Hint: `a` is a login attempt.

```
def chk(u, lst, n):
    c = 0
    for a in lst:
        if a.uid == u.id and a.ok is False:
            c += 1
    return c >= n
```

#### SOLUTION

```
def has_too_many_failed_logins(user, login_attempts, max_failures):
    failures = 0
    for attempt in login_attempts:
        if attempt.user_id == user.id and attempt.succeeded is False:
            failures += 1
    return failures >= max_failures
```

With good names, a simplification becomes visible that was invisible before: the loop is a count, and could be a `sum()` over a generator.

One trap is worth noticing. The original tests `a.ok is False`, which excludes `None`. If an attempt can have an unknown outcome, rewriting it as `not attempt.succeeded` would count unknowns as failures and change behaviour. A rename must never change logic, so check before you tidy.

#### CHECKLIST

- ☐ Can a reader tell what each name holds or does without reading its definition?
- ☐ Does every boolean read naturally after `if`?
- ☐ Are units in the name wherever the type doesn't carry them?
- ☐ Is there exactly one word for each concept in this module?

- ☐ Did you rename the things you now understand better than when you named them?

GO DEEPER

BOOK      **A Philosophy of Software Design** John Ousterhout · the chapter on choosing names

---

BOOK      **Domain-Driven Design** Eric Evans · the origin of "ubiquitous language"

---

FREE      **A Philosophy of Software Design vs Clean Code** Ousterhout and Martin disagree in public, usefully

# Functions

3 exercises

*A function should do one thing, at one level of abstraction, and its signature should tell a caller everything they need without opening the body.*

## What "one thing" means

The rule is old and usually misread as "short". A better test is whether you can describe what the function does in one sentence without the word "and". If you can, it probably does one thing, whatever its length. If the sentence needs "validates the cart *and* charges the card *and* emails a receipt", you have three functions sharing a body.

The second test is levels of abstraction. A function that reads like a sequence of business steps shouldn't suddenly drop into string slicing or HTTP details. Mixing levels forces the reader to change zoom constantly.

**BEFORE** checkout.py · four jobs at three levels

```
def checkout(cart, card, email):
    if not cart.items:
        raise ValueError("cart is empty")
    for item in cart.items:
        if item.quantity <= 0:
            raise ValueError(f"bad quantity for {item.sku}")
    subtotal = sum(i.unit_price * i.quantity for i in cart.items)
    if cart.coupon and cart.coupon.expires_at > datetime.now():
        subtotal -= subtotal * cart.coupon.percent_off / 100
    total = subtotal + round(subtotal * 0.2, 2)
    resp = requests.post(
        "https://payments.example.com/charges",
        json={"card": card.token, "amount": int(total * 100)},
        timeout=10,
    )
    if resp.status_code != 201:
        raise PaymentFailed(resp.text)
    body = f"Thanks! You paid {total:.2f} for {len(cart.items)} items."
    smtplib.SMTP("mail.example.com").sendmail("shop@example.com", [email], body)
    return resp.json()["charge_id"]
```

**AFTER** checkout.py · the top level reads as the business process

```
def checkout(cart, card, email):
    validate(cart)
    total = price(cart, now=datetime.now())
    charge_id = payments.charge(card, total)
    send_receipt(email, cart, total)
    return charge_id
```

```
def validate(cart):
    if not cart.items:
        raise EmptyCart()
    for item in cart.items:
        if item.quantity <= 0:
            raise InvalidQuantity(item.sku)

def price(cart, now):
    subtotal = sum(i.unit_price * i.quantity for i in cart.items)
    if cart.coupon and cart.coupon.is_valid_at(now):
        subtotal = cart.coupon.apply(subtotal)
    return subtotal + tax_on(subtotal)
```

`checkout` now reads as the four steps of the process, and each step can be understood, tested and changed alone. `price` takes `now` as an argument instead of reading the clock itself, which makes it deterministic; chapter 6 explains why that matters so much. The payment and email details moved behind `payments.charge` and `send_receipt`, where they live with the rest of their kind.

## How long is too long?

There is no line count. Robert Martin argues for functions of a handful of lines. John Ousterhout argues that aggressive splitting produces *conjoined* functions that can only be understood together, so a reader must jump between five of them to follow a single idea. They debated this in public and did not converge.

The useful synthesis is two questions. Can a caller use this function without reading its body? Can a reader understand this function without reading the functions it calls? When splitting a function makes either answer worse, don't split it.

## Arguments

- **Fewer is better.** Every parameter is something a caller must understand and supply. Zero to three is comfortable. More usually means some of them belong together.
- **Flag arguments are two functions in disguise.** `render(invoice, True)` means nothing at the call site. Split it into `render_draft` and `render_final`, or at least make the flag keyword-only so the call reads `render(invoice, draft=True)`.
- **Group parameters that travel together.** If `street`, `city`, `postcode`, `country` appear in five signatures, they are an `Address`. Introducing that object is one of the most common refactorings in chapter 11.
- **Don't mutate your arguments** unless the name says so. A caller who passes a list to `total_of(items)` does not expect to find it sorted afterwards.

**EXAMPLE** keyword-only flags in Python

```
# The bare * forces callers to name every argument after it.
```

```
def export_orders(orders, *, include_refunds=False, as_csv=True):
    ...

export_orders(orders, include_refunds=True)    # reads clearly
export_orders(orders, True)                   # TypeError: rejected at the call
```

## Commands and queries

Bertrand Meyer's **command-query separation** says a function should either change state or return information, never both. A query can be called any number of times without consequence, so it is safe in a log line, a debugger or a test. A command's name warns you that something happens.

Functions that do both surprise people: a `get_next_invoice_number()` that also increments a counter, or an `is_valid()` that records the check. The standard library breaks the rule where it's genuinely convenient, `list.pop()` being the famous case, and that's fine as long as the name makes the side effect obvious.

## Guard clauses over nesting

### BEFORE

```
def shipping_cost(order):
    if order.items:
        if order.country in SUPPORTED_COUNTRIES:
            if order.total < FREE_SHIPPING_THRESHOLD:
                return rate_for(order.country, order.weight_kg)
            else:
                return 0
        else:
            raise UnsupportedCountry(order.country)
    else:
        return 0
```

### AFTER

```
def shipping_cost(order):
    if not order.items:
        return 0
    if order.country not in SUPPORTED_COUNTRIES:
        raise UnsupportedCountry(order.country)
    if order.total >= FREE_SHIPPING_THRESHOLD:
        return 0
    return rate_for(order.country, order.weight_kg)
```

Each special case is handled and dismissed at the top, and the main path runs unindented to the end. The reader never has to track which `else` belongs to which `if`.

## Functional core, imperative shell

Gary Bernhardt's name for a structure that makes most code easy to test. Put decisions in pure functions that take values and return values: no clock, no network, no database. Wrap them in a thin outer layer that gathers inputs, calls the core, and performs the effects. The core is where the bugs would be, and it needs no mocks to test. The shell is so simple it barely needs testing at all.

**EXAMPLE** reminders.py

```
# Core: pure. Values in, values out.
def overdue_reminders(invoices, today):
    return [
        Reminder(inv.customer_email, inv.number, (today - inv.due_date).days)
        for inv in invoices
        if not inv.paid and inv.due_date < today
    ]

# Shell: gathers inputs, calls the core, performs the effects.
def send_overdue_reminders(db, mailer, clock):
    invoices = db.unpaid_invoices()
    for reminder in overdue_reminders(invoices, clock.today()):
        mailer.send(reminder.to, render(reminder))
```

---

**COMMON TRAPS**

**Extracting until nothing is left.** Ten three-line functions that each call the next are harder to follow than one thirty-line function. Inline them back.

---

**Names that restate the body.** Extracting `x > 0 and y > 0` into `check_x_and_y()` adds a jump and no meaning. `is_in_first_quadrant()` adds meaning.

---

**Hidden state threading.** Helpers that read and write instance variables set by other helpers are conjoined. Pass values explicitly.

---

**Returning `None` for failure.** Every caller must remember to check, and the ones who forget crash later, far from the cause. Raise instead. Chapter 16 covers this.

---

**EXERCISES**

**EXERCISE 2.1 Flatten with guard clauses**

Rewrite this so the main path is unindented. Behaviour must not change.

```
def can_publish(post, user):
    if user is not None:
        if user.is_active:
```

```

        if post.author_id == user.id or user.is_editor:
            if not post.is_locked:
                return True
            else:
                return False
        else:
            return False
    else:
        return False
else:
    return False

```

#### SOLUTION

```

def can_publish(post, user):
    if user is None or not user.is_active:
        return False
    if post.is_locked:
        return False
    return post.author_id == user.id or user.is_editor

```

Checking `is_locked` before ownership changes nothing here, because both paths return `False`. In general, though, reordering guards can change which error a caller sees, so verify equivalence rather than assuming it. A quick way is a table-driven test over every combination of inputs, run against both versions.

### EXERCISE 2.2 Spot the separation violations

Which of these break command-query separation, and what would you rename or split?

```

def next_order_number(self):
    self._counter += 1
    return f"ORD-{self._counter:06d}"

def is_valid(self, coupon):
    coupon.times_checked += 1
    return coupon.expires_at > self.clock.now()

def total(self):
    return sum(line.amount for line in self.lines)

```

#### SOLUTION

- `next_order_number` changes state and returns a value, but "next" signals consumption, much like `pop`. It's an acceptable exception. Renaming it `allocate_order_number()` makes the command even clearer.

- `is_valid` is the real problem: a query's name hiding a side effect. Calling it in a log line or a debugger changes data. Remove the counter update from it, or move that into an explicit `record_check(coupon)` command.
- `total` is a pure query. It's fine.

### EXERCISE 2.3 Separate the core from the shell

Extract a pure function that decides who to notify and with what message, then write a test for it that needs no database and no mailer.

```
def notify_expiring_subscriptions():
    today = date.today()
    for sub in db.query("SELECT * FROM subscriptions WHERE active"):
        days_left = (sub.ends_on - today).days
        if days_left in (7, 1):
            mailer.send(sub.email, f"Your plan ends in {days_left} day(s).")
```

#### SOLUTION

```
def expiry_notices(subscriptions, today):
    notices = []
    for sub in subscriptions:
        days_left = (sub.ends_on - today).days
        if days_left in (7, 1):
            notices.append((sub.email, f"Your plan ends in {days_left} day(s)."))
    return notices

def notify_expiring_subscriptions(db, mailer, today):
    for email, message in expiry_notices(db.active_subscriptions(), today):
        mailer.send(email, message)

def test_notices_go_out_seven_days_and_one_day_before_expiry():
    today = date(2026, 3, 1)
    subs = [
        Sub("a@x.com", ends_on=date(2026, 3, 8)), # 7 days away
        Sub("b@x.com", ends_on=date(2026, 3, 2)), # 1 day away
        Sub("c@x.com", ends_on=date(2026, 3, 5)), # 4 days away: no notice
    ]
    assert expiry_notices(subs, today) == [
        ("a@x.com", "Your plan ends in 7 day(s)."),
        ("b@x.com", "Your plan ends in 1 day(s)."),
    ]
```

The test needs no mocks, no database and no patched clock. That is the entire payoff of the pattern.

## CHECKLIST

- ☐ Can you describe each function in one sentence without "and"?
- ☐ Does each function stay at one level of abstraction?
- ☐ Are there boolean flags at call sites that should be keyword-only, or separate functions?
- ☐ Do any queries have side effects?
- ☐ Is the decision-making code pure, with I/O pushed to the edges?

## GO DEEPER

- |             |                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------|
| <b>TALK</b> | <b>Boundaries</b> Gary Bernhardt, 2012 · the origin of functional core, imperative shell                |
| <hr/>       |                                                                                                         |
| <b>BOOK</b> | <b>A Philosophy of Software Design</b> John Ousterhout · the chapter on when to split and when to join  |
| <hr/>       |                                                                                                         |
| <b>BOOK</b> | <b>Object-Oriented Software Construction</b> Bertrand Meyer · command-query separation, from the source |

# Comments

2 exercises

*Comments should say what the code cannot: why a decision was made, what an interface promises, and what a reader would otherwise have to learn the hard way.*

## The disagreement worth understanding

Few topics split experienced engineers like comments. Robert Martin treats a comment as a small failure: a sign the code didn't express itself, and a liability that will drift out of date. John Ousterhout treats comments as essential. Code can say what happens, but it cannot say why, and it cannot describe an abstraction without the reader reading its implementation. Their public exchange ended with each holding the opposite default: Martin distrusts a comment until it proves useful, and Ousterhout assumes it helps until it proves wasteful.

Both are right, about different kinds of comment. Comments that narrate what the code does are noise, and Martin's instinct is correct for them. Comments that describe an interface or record a reason are irreplaceable, and Ousterhout's instinct is correct there. The skill is telling the kinds apart.

## Comments worth writing

| KIND             | WHAT IT SAYS                                                              | EXAMPLE                                                                           |
|------------------|---------------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| <b>Interface</b> | What a function promises its caller: inputs, output, errors, side effects | A docstring on a public function                                                  |
| <b>Why</b>       | The reason for a decision that looks wrong or arbitrary                   | "Retry twice: the provider returns 502s for about a second during their deploys." |
| <b>Warning</b>   | A consequence the reader must know before changing the code               | "Not thread-safe. Callers must hold the order lock."                              |
| <b>Reference</b> | Where the full explanation lives                                          | A link to the spec, the ticket, or the paper                                      |
| <b>Intent</b>    | The idea behind code that can only execute it                             | "Both lists are sorted, so advance whichever head is smaller."                    |

And the comments worth deleting:

- **Narration.** `# loop over the orders` above `for order in orders:` .

- **Commented-out code.** Version control remembers it. Nobody will ever be sure it's safe to delete, so everybody leaves it, forever.
- **Journal comments.** Change history belongs in commit messages, where it's attached to the change.
- **Banners.** `# ===== HELPERS =====` usually means the file wants to be two files.
- **Apologies.** `# sorry, hack` without saying what the hack works around. Replace it with a why comment and a ticket link.

## Write the interface comment first

Ousterhout suggests writing the interface comment before the implementation. It works as a design tool: if the comment is long, full of conditions and caveats, the abstraction is probably wrong, and it's cheaper to find out before the code exists. A good interface comment describes *what*, never *how*. The how belongs to the implementation, which must stay free to change.

**BEFORE** narration, and no interface

```
def reserve(sku, qty):
    # get the stock row
    row = db.get_stock(sku)
    # check if enough
    if row.available < qty:
        # not enough, return false
        return False
    # subtract
    row.available -= qty
    row.reserved += qty
    # save
    db.save(row)
    return True
```

**AFTER** an interface a caller can rely on

```
def reserve(sku: str, qty: int) -> bool:
    """Hold `qty` units of `sku` for an order that hasn't been paid yet.

    Returns False, and changes nothing, if fewer than `qty` units are
    available. Reserved units stop counting as available until they are
    released with `release()` or converted to a sale with `commit()`.

    Not safe to call concurrently for the same SKU: callers must hold the
    per-SKU lock from `stock_lock(sku)`.
    """
    row = db.get_stock(sku)
    if row.available < qty:
        return False
    row.available -= qty
    row.reserved += qty
    db.save(row)
    return True
```

---

Every narration comment is gone, and the body is no harder to read without them. The docstring now tells a caller three things the code alone could not: that failure leaves state unchanged, what the reservation's lifecycle is, and the locking rule. None of that is visible in seven lines of implementation.

## The comment that protects a fence

G. K. Chesterton described a fence across a road. A reformer wants it removed because he can see no use for it. The wiser reply: go and find out why it was put there, and then you may take it down. Code is full of fences, such as an odd sleep, a check that looks redundant, or a sort that seems unnecessary. A one-line why comment is the difference between the next engineer understanding the fence and removing it in a cleanup that brings back a production bug from three years ago.

**EXAMPLE** a fence with its reason attached

```
# Sort by (created_at, id), not created_at alone: the bulk importer writes
# thousands of rows with identical timestamps, and pagination skips or
# repeats rows unless the order is total. See INC-2291.
rows.sort(key=lambda r: (r.created_at, r.id))
```

---

### COMMON TRAPS

**Docstrings that repeat the types.** `sku (str):` the `sku` adds nothing a type hint doesn't. Describe meaning, units and constraints instead: "a catalogue SKU, case-sensitive".

---

**Stale comments.** A comment the code has outgrown is worse than none. Review comments as carefully as the code beside them.

---

**Comments excusing bad names.** `x = 86400 # seconds in a day` should be `SECONDS_PER_DAY = 86400`.

---

**Explaining how in an interface comment.** It couples callers to the implementation you'll want to change later.

---

### EXERCISES

#### EXERCISE 3.1 Keep, delete or rewrite

For each comment, decide and give a reason.

```
# 1
```

```

i += 1 # increment i

# 2
# Use a set: this runs per request and the blocklist has ~50k entries.
blocked = set(load_blocklist())

# 3
# old approach
# for u in users: send(u)

# 4
time.sleep(0.2) # wait

# 5
# Returns None if the user doesn't exist.
def find_user(email): ...

# 6
# TODO fix this

```

## SOLUTION

1. **Delete.** Pure narration.
2. **Keep.** It records why a set was chosen, with the numbers that justify it. A reader who "simplifies" it to a list would make every request slower.
3. **Delete.** Commented-out code. Version control has it.
4. **Rewrite.** The code already says it waits. Say why, and why this long: "The label printer drops jobs sent within ~150 ms of each other."
5. **Keep, and move it into the docstring.** It's interface information. Better still, declare the return type as `User | None` so the type checker enforces it.
6. **Rewrite.** Say what's wrong and link a ticket: "TODO(INV-412): rounding is per line; finance wants rounding on the invoice total."

## EXERCISE 3.2 Write the interface comment

Write the docstring a caller needs. Describe behaviour, not implementation.

```

def merge_customer_records(primary_id, duplicate_id):
    primary = repo.get(primary_id)
    dup = repo.get(duplicate_id)
    for order in repo.orders_of(dup.id):
        order.customer_id = primary.id
        repo.save(order)
    primary.loyalty_points += dup.loyalty_points
    if not primary.phone:
        primary.phone = dup.phone
    repo.save(primary)
    repo.delete(dup.id)

```

## SOLUTION

```
def merge_customer_records(primary_id: int, duplicate_id: int) -> None:
    """Fold a duplicate customer into the primary record, then delete it.

    Every order of the duplicate is reassigned to the primary and loyalty
    points are summed. The duplicate's phone number is kept only if the
    primary has none; all its other fields are discarded.

    Irreversible: the duplicate record is deleted. Both ids must exist.
    Not atomic: if it fails part way, some orders may already have moved,
    so call it inside a transaction.
    """
```

Writing this surfaces two design questions that the code hides: should this be atomic on its own, and is silently discarding the duplicate's other fields acceptable? That's the comment doing design work before anyone has been hurt by the answer.

## CHECKLIST

- ☐ Does every public function have an interface comment that says what, not how?
- ☐ Is there a why comment on every line that looks wrong but isn't?
- ☐ Did you delete narration, commented-out code and journal comments?
- ☐ Do the comments still match the code after your change?

## GO DEEPER

- |       |                                                                                                      |
|-------|------------------------------------------------------------------------------------------------------|
| BOOK  | A Philosophy of Software Design John Ousterhout · the chapters on comments and comments-first design |
| <hr/> |                                                                                                      |
| FREE  | The Ousterhout and Martin exchange the section on comments is the best argument on both sides        |
| <hr/> |                                                                                                      |
| BOOK  | The Thing G. K. Chesterton, 1929 · where the fence first appears                                     |

# Complexity and deep modules

3 exercises

*Complexity is anything that makes a system hard to understand or change. The strongest defence is the deep module: a lot of functionality behind a small, simple interface.*

## What complexity looks like

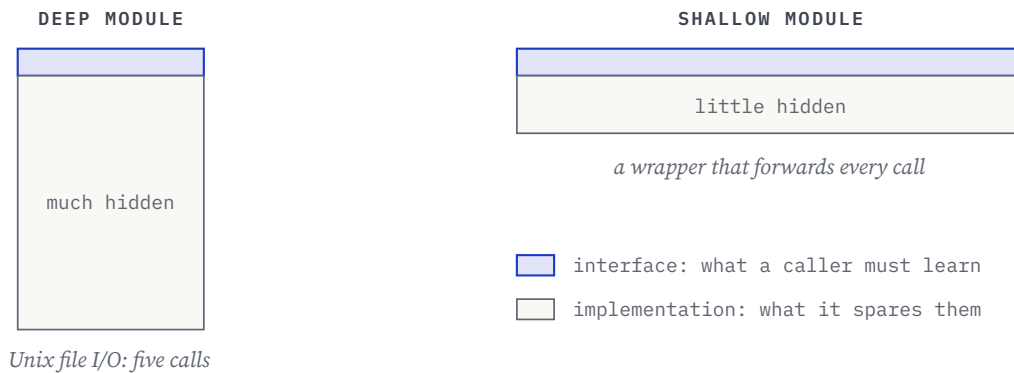
John Ousterhout defines complexity by its effects, which makes it recognisable in practice. It shows up in three ways.

- **Change amplification.** A simple change requires edits in many places. Changing the currency symbol means touching forty templates.
- **Cognitive load.** How much a developer must hold in mind to make a change safely. A function that only works if you remember to call `init()` first and `flush()` afterwards has high cognitive load.
- **Unknown unknowns.** It isn't obvious which code must change, or what you need to know to change it. This is the worst of the three, because you can't see the cost until something breaks.

Its two root causes are **dependencies**, where code can't be understood or changed in isolation, and **obscurity**, where important information isn't obvious. Most of the rest of this volume is about reducing one or the other.

## Deep and shallow modules

Picture every module as a rectangle. The top edge is its interface: everything a caller must learn to use it. The area beneath is the functionality it provides. A deep module has a narrow top and a large area. It gives a lot and asks little. A shallow module has a wide top and a small area, and its interface costs nearly as much to learn as its implementation would to write.



*The top edge is the cost a module imposes on every caller; the area is the value it gives back. Aim for narrow and tall.*

Unix file I/O is the classic deep module. Five calls, `open`, `read`, `write`, `lseek` and `close`, hide disk layout, caching, permissions, device drivers and concurrent access. A class whose methods each forward to one method on another class is the classic shallow module: an interface to learn, and nothing behind it.

## A worked example: a rate limiter

**BEFORE** shallow: every caller must know the algorithm

```
class TokenBucketStore:
    def get_bucket(self, key): ...
    def save_bucket(self, key, bucket): ...

class TokenBucket:
    def __init__(self, capacity, tokens, last_refill): ...
    def refill(self, now, rate_per_second): ...
    def try_consume(self, n=1): ...

# Repeated in every endpoint:
bucket = store.get_bucket(user_id) or TokenBucket(100, 100, time.monotonic())
bucket.refill(time.monotonic(), rate_per_second=5)
allowed = bucket.try_consume()
store.save_bucket(user_id, bucket)
if not allowed:
    return too_many_requests()
```

**AFTER** deep: one question, one answer

```
class RateLimiter:
    """Allow bursts of up to `burst` requests per key, refilling at
    `per_second`. Thread-safe. Unknown keys start with a full allowance."""

    def __init__(self, per_second: float, burst: int, store, clock=time.monotonic):
        ...

    def allow(self, key: str) -> bool:
        ...
```

```
# Every endpoint:
if not limiter.allow(user_id):
    return too_many_requests()
```

Both implement the same token bucket. In the shallow design, every endpoint repeats five lines of algorithm, and a caller who forgets `refill` or `save_bucket` gets a limiter that is subtly wrong. Switch to a sliding window and you edit every caller: change amplification. The deep design hides the algorithm, the storage, the clock and the locking behind one method. The interface got smaller, and the module got more powerful.

## Information hiding, and how it leaks

David Parnas made the case in 1972: divide a system by the design decisions most likely to change, and give each module one decision to hide. The opposite is **information leakage**, where one decision is reflected in several modules. If both the CSV exporter and the CSV importer know the column order, changing the format means changing both, and nothing forces you to remember.

A common cause is **temporal decomposition**: structuring code by the order things happen. A read step, a parse step and a write step feel natural, but if reading and writing both need to know the file format, the format leaks into both. Structure by knowledge, not by time. One module owns the format and offers both directions.

**BEFORE** the same knowledge in two places

```
# exporter.py
writer.writerow([o.id, o.customer_email, o.total_cents, o.created_at.isoformat()])

# importer.py
order_id, email, total_cents, created_at = row    # a second copy of the format
```

**AFTER** order\_csv.py · the only module that knows the format

```
COLUMNS = ("id", "customer_email", "total_cents", "created_at")

def to_row(order):
    return [order.id, order.customer_email, order.total_cents,
            order.created_at.isoformat()]

def from_row(row):
    fields = dict(zip(COLUMNS, row))
    return Order(
        id=int(fields["id"]),
        customer_email=fields["customer_email"],
        total_cents=int(fields["total_cents"]),
        created_at=datetime.fromisoformat(fields["created_at"]),
    )
```

## Pull complexity downwards

When complexity is unavoidable, it's better for the module to absorb it than to push it onto every caller. A module has one author and many users. Sensible defaults, handling common edge cases internally, and retrying transient failures are all ways of pulling complexity down. A configuration option for something the module could work out itself does the opposite: it pushes a decision onto every user.

## Define errors out of existence

Every error a module can raise is part of its interface, and every caller must deal with it. Sometimes the cheapest fix is to redefine the operation so the error can't happen. Python's standard library has good examples.

**EXAMPLE** errors that were designed away

```
Path("reports").mkdir(exist_ok=True)    # "already exists" is not an error
settings.get("theme", "light")          # a missing key has a default
"hello"[10:20]                          # slicing past the end returns ""
cache.pop(key, None)                    # removing a missing key is a no-op
```

Each one takes a case that would otherwise be an exception and gives it a sensible meaning. The same move works in your own code: make `unsubscribe` succeed when the user is already unsubscribed, and every caller loses a `try` block. Use judgement, though. Some errors must stay loud, like a payment that failed.

## Tactical and strategic programming

Tactical programming aims to get the current feature working as fast as possible. Each shortcut is small and seems reasonable, and complexity accumulates one reasonable decision at a time. Strategic programming treats working code as necessary but not sufficient: the goal is a good design that also works. Ousterhout suggests investing something like ten to twenty percent of development time in design improvements. It pays back within months, because every later change is cheaper.

---

### COMMON TRAPS

**Classitis.** Splitting everything into tiny classes because small feels good produces many shallow modules, each with an interface to learn and little behind it.

---

**Pass-through methods.** A method that only calls another method with the same signature adds interface without adding depth. Merge the layers, or give each a distinct job.

---

**Configuration as an escape hatch.** Every option you expose is a decision every user must now make.

---

**Pretending abstractions don't leak.** Joel Spolsky's law says all non-trivial abstractions leak to some degree. Design for the leaks you can't avoid, such as network failure, and document them in the interface.

---

## EXERCISES

### EXERCISE 4.1 Make it deep

This module makes every caller manage a connection. Redesign its interface so callers can't get it wrong. Write the new signatures and one example call.

```
class Mailer:
    def connect(self): ...
    def authenticate(self, user, password): ...
    def build_message(self, to, subject, body): ...
    def send_message(self, message): ...
    def disconnect(self): ...

# Every caller:
m = Mailer()
m.connect()
m.authenticate(SMTP_USER, SMTP_PASSWORD)
msg = m.build_message(customer.email, "Your receipt", body)
m.send_message(msg)
m.disconnect()
```

---

### SOLUTION

```
class Mailer:
    """Sends email through the configured SMTP relay. Connections are
    opened, reused and closed internally. Transient failures are retried
    up to 3 times before MailDeliveryError is raised."""

    def __init__(self, host: str, user: str, password: str): ...

    def send(self, to: str, subject: str, body: str) -> None: ...

# Every caller:
mailer.send(customer.email, "Your receipt", body)
```

Credentials are supplied once, when the application is wired together. Connections, message building and retries are hidden. The caller's cognitive load drops from five ordered steps to one call, and the forgotten- `disconnect()` bug is gone because it can no longer be written.

### EXERCISE 4.2 Define the error away

This function raises in a case that callers nearly always treat as success. Redesign it.

```
def remove_from_cart(cart, sku):
    if sku not in cart.items:
        raise ItemNotInCart(sku)
    del cart.items[sku]
```

#### SOLUTION

Make removal idempotent. Removing an item that isn't there leaves the cart exactly as the caller wanted it:

```
def remove_from_cart(cart, sku):
    cart.items.pop(sku, None)
```

If some caller genuinely needs to know whether anything was removed, return a boolean rather than raising. Most callers will ignore it, which is exactly what they want.

### EXERCISE 4.3 Find the leak

Two modules share a design decision. Name it, and restructure so only one module knows it.

```
# signup.py
def create_user(email, password):
    salt = os.urandom(16)
    digest = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, 600_000)
    db.insert_user(email, salt + digest)

# login.py
def check_password(user, password):
    salt, digest = user.password_hash[:16], user.password_hash[16:]
    candidate = hashlib.pbkdf2_hmac("sha256", password.encode(), salt, 600_000)
    return hmac.compare_digest(candidate, digest)
```

#### SOLUTION

The leaked decision is the password storage format: the algorithm, the iteration count, the salt length and the byte layout. Both modules know all four. Raise the iteration count in one and every login breaks.

```
# passwords.py: the only module that knows the storage format.
ITERATIONS = 600_000
SALT_BYTES = 16
```

```
def hash_password(password: str) -> bytes:
    salt = os.urandom(SALT_BYTES)
    return salt + _derive(password, salt)

def verify_password(password: str, stored: bytes) -> bool:
    salt, digest = stored[:SALT_BYTES], stored[SALT_BYTES:]
    return hmac.compare_digest(_derive(password, salt), digest)

def _derive(password: str, salt: bytes) -> bytes:
    return hashlib.pbkdf2_hmac("sha256", password.encode(), salt, ITERATIONS)
```

A production version should also record the algorithm and iteration count inside the stored hash, so the parameters can be upgraded without breaking existing passwords. Encoded formats such as Argon2's do exactly that, which is one more reason to use a vetted library here.

---

#### CHECKLIST

- ☐ For each module you touched, is its interface much simpler than its implementation?
- ☐ Would changing one design decision touch one module, or several?
- ☐ Did you push onto callers a decision the module could have made?
- ☐ Can any error you raise be defined out of existence?
- ☐ Did part of this change improve the design, or did it only add to it?

#### GO DEEPER

- |       |                                                                                       |
|-------|---------------------------------------------------------------------------------------|
| BOOK  | A Philosophy of Software Design John Ousterhout · short, and this chapter is its core |
| <hr/> |                                                                                       |
| PAPER | On the Criteria To Be Used in Decomposing Systems into Modules David Parnas, 1972     |
| <hr/> |                                                                                       |
| ESSAY | The Law of Leaky Abstractions Joel Spolsky, 2002                                      |

## PART II

# Design

Three chapters on how pieces of code depend on each other: the principles people cite, the structures that keep dependencies swappable, and the named patterns that recur often enough to be worth recognising on sight.

# SOLID, read critically

2 exercises

*SOLID is five heuristics about managing dependencies in object-oriented code. Learn the problem each one solves, because that is how you'll know when it doesn't apply.*

Robert Martin assembled these principles around the turn of the millennium, and they became an interview staple. They're genuinely useful at the boundaries of a system, where change is likely. Applied everywhere, they produce an interface for every class and five files for every feature. Treat each one as the answer to a specific question.

## S: single responsibility

The original wording was "a class should have only one reason to change". Martin later sharpened it: a module should be responsible to one *actor*, meaning one group of people who might ask for changes. The point isn't that a class does one tiny thing. It's that changes requested by different people shouldn't collide in the same code.

**BEFORE** three sets of stakeholders, one class

```
class Invoice:
    def total(self): ...           # finance decides the rules
    def to_pdf(self): ...         # the brand team decides the layout
    def email_to_customer(self): ... # support decides wording and timing
```

A tax rule change and a logo change now touch the same class, the same tests and the same review. Split along the lines of who asks for change: keep `total` on `Invoice`, move rendering to an `InvoicePdf`, and sending to a `ReceiptSender`.

## O: open for extension, closed for modification

You should be able to add behaviour by adding code, rather than by editing code that already works. In practice, that means finding the axis along which variation actually happens and making it pluggable.

**BEFORE** every new promotion edits this function

```
def discount(order):
    if order.coupon_type == "PERCENT":
        return order.subtotal * order.coupon_value / 100
    elif order.coupon_type == "FIXED":
        return min(order.coupon_value, order.subtotal)
    elif order.coupon_type == "BOGO":
        return cheapest_item_price(order)
```

**AFTER** new promotions are new entries

```
DISCOUNT_RULES = {
    "PERCENT": lambda o: o.subtotal * o.coupon_value / 100,
    "FIXED": lambda o: min(o.coupon_value, o.subtotal),
    "BOGO": cheapest_item_price,
}

def discount(order):
    return DISCOUNT_RULES[order.coupon_type](order)
```

The caveat is the important part. Do this only where variation has actually happened. Building plug-in points for variations nobody has asked for is speculative generality, and it makes code harder to read for no benefit. YAGNI, "you aren't gonna need it", is the counterweight to this principle.

## L: Liskov substitution

Barbara Liskov's principle: anywhere code uses a type, it must be able to use any subtype without noticing. A subtype must accept at least what its parent accepts, promise at least what its parent promises, and raise no new surprises. The textbook violation is a `Square` that inherits from `Rectangle` and breaks the expectation that setting the width leaves the height alone. The one you'll actually meet lives in storage backends and test fakes.

**BEFORE** two implementations, two contracts

```
class UserRepository(Protocol):
    def get(self, user_id: int) -> User:
        """Raises UserNotFound if no such user exists."""

class SqlUserRepository:
    def get(self, user_id):
        row = self.db.fetch_one("SELECT * FROM users WHERE id = %s", user_id)
        if row is None:
            raise UserNotFound(user_id)
        return User.from_row(row)

class InMemoryUserRepository:
    def get(self, user_id):
        return self.users.get(user_id)  # returns None: breaks the contract
```

Code tested against the in-memory version never sees `UserNotFound`, so the error path goes untested and breaks in production. Chapter 9 shows the fix: one set of contract tests, run against both implementations.

## I: interface segregation

Clients shouldn't depend on methods they don't use. A function that only reads a file shouldn't demand an object that can also delete buckets and rotate credentials. That's harder to call, harder to fake, and coupled to changes it doesn't care about. In Python, a `Protocol` states exactly what a function needs.

#### EXAMPLE

```
class Readable(Protocol):
    def read(self, key: str) -> bytes: ...

def load_price_list(source: Readable) -> PriceList:
    return PriceList.parse(source.read("prices.json"))

# Anything with a read() method now works: an S3 client, a local
# directory, or a dictionary wrapper in a test.
```

## D: dependency inversion

High-level policy shouldn't depend on low-level detail. Both should depend on an abstraction, and the policy side should own it. Your order logic shouldn't import a payment provider's SDK. It should declare what it needs from a payment gateway, and a small adapter should provide that on top of the provider.

#### EXAMPLE the arrow of dependency points inwards

```
# Owned by the domain.
class PaymentGateway(Protocol):
    def charge(self, card_token: str, amount: Money) -> ChargeId: ...

class OrderService:
    def __init__(self, payments: PaymentGateway):
        self.payments = payments

# Lives at the edge, and is the only code that knows about Stripe.
class StripeGateway:
    def charge(self, card_token, amount):
        intent = stripe.PaymentIntent.create(
            amount=amount.cents, currency=amount.currency.lower(),
            payment_method=card_token, confirm=True,
        )
        return ChargeId(intent.id)
```

Imports now point from the Stripe adapter towards your domain, not the other way round. Switching providers, or using a fake in tests, touches one small class.

## The critique

Two criticisms are worth taking seriously. First, SOLID was framed for the class-based languages of the 1990s. In a language with first-class functions, several of the principles collapse into "pass a function", and in data-oriented code some barely apply. Second, principles phrased as rules invite mechanical application: an interface for every class, including those with one implementation that will never change.

In 2022 Dan North proposed **CUPID** as an alternative framing: properties to move towards rather than rules to obey. Code should be **composable**, follow the **Unix philosophy** of doing one thing well, be **predictable**, be **idiomatic** for its language, and be **domain-based** in its names and structure. You needn't adopt it. It's a useful reminder that the goal is code that's pleasant to change, not code that passes a checklist.

| PRINCIPLE                    | QUESTION IT ANSWERS                  | SMELL WHEN IGNORED                     | SMELL WHEN OVERDONE                    |
|------------------------------|--------------------------------------|----------------------------------------|----------------------------------------|
| <b>Single responsibility</b> | Whose changes collide here?          | Unrelated edits conflict in one file   | Logic scattered across tiny classes    |
| <b>Open/closed</b>           | Where does variation happen?         | An if/elif chain edited for every case | Plug-in points nobody has ever used    |
| <b>Liskov substitution</b>   | Can I trust any implementation?      | Subclasses that raise "not supported"  | Rarely overdone                        |
| <b>Interface segregation</b> | What does this caller actually need? | Fakes that stub twenty unused methods  | A one-method interface for everything  |
| <b>Dependency inversion</b>  | Which way should imports point?      | Business logic importing vendor SDKs   | Interfaces for classes that never vary |

## COMMON TRAPS

**Reciting instead of reasoning.** In reviews and interviews, naming a principle isn't an argument. Say what change it makes cheaper.

**An interface with one implementation, forever.** An abstraction you never substitute costs a file and a jump and buys nothing. Add it when the second implementation, or the test fake, arrives.

**Inheriting to reuse code.** Liskov violations nearly always start here. Chapter 6 covers the alternative.

## EXERCISES

## EXERCISE 5.1 Diagnose the strain

Which principle does each snippet strain, and what would you change?

```
# A
class ReadOnlyList(list):
    def append(self, item):
        raise NotImplementedError("read-only")

# B
def ship(order):
    label = FedExClient(api_key=os.environ["FEDEX_KEY"]).create_label(order)
    ...

# C
class ReportGenerator:
    def query_sales(self): ...
    def compute_trends(self): ...
    def render_html(self): ...
    def upload_to_s3(self): ...
    def schedule_nightly(self): ...

# D
def print_summary(repo: FullOrderRepository):  # uses only repo.count()
    print(f"{repo.count()} orders")
```

### SOLUTION

- **A, Liskov substitution.** Code that accepts a `list` may call `append`. A read-only sequence shouldn't be a list subtype. Use a tuple, or implement `collections.abc.Sequence`, which has no mutating methods.
- **B, dependency inversion.** Shipping policy constructs a vendor client and reads its credentials. Inject a `Carrier` abstraction and build the FedEx adapter at the edge.
- **C, single responsibility.** Data access, analysis, presentation, storage and scheduling are each requested by different people. Split them so a chart change can't break the nightly upload.
- **D, interface segregation.** Accept a protocol with only `count()`, or simply pass the number: `print_summary(order_count: int)`. The simplest interface is often a plain value.

## EXERCISE 5.2 Invert the dependency

Make `OrderService.place` testable without charging real cards and without patching anything.

```
class OrderService:
```

```
def place(self, cart, card_token):
    total = cart.total()
    intent = stripe.PaymentIntent.create(
        amount=total.cents, currency="usd",
        payment_method=card_token, confirm=True,
    )
    return Order(cart=cart, charge_id=intent.id)
```

## SOLUTION

```
class OrderService:
    def __init__(self, payments: PaymentGateway):
        self.payments = payments

    def place(self, cart, card_token):
        charge_id = self.payments.charge(card_token, cart.total())
        return Order(cart=cart, charge_id=charge_id)

class FakePayments:
    def __init__(self):
        self.charges = []

    def charge(self, card_token, amount):
        self.charges.append((card_token, amount))
        return f"ch_fake_{len(self.charges)}"

def test_placing_an_order_charges_the_cart_total():
    payments = FakePayments()
    cart = Cart.with_items(("SKU-1", Money.usd("12.50"), 2))

    order = OrderService(payments).place(cart, "tok_visa")

    assert payments.charges == [("tok_visa", Money.usd("25.00"))]
    assert order.charge_id == "ch_fake_1"
```

The Stripe call moves into the `StripeGateway` shown earlier in the chapter. The service is now tested in milliseconds, with no network and no patching, and the test reads as a statement about behaviour.

## CHECKLIST

- ☐ Can you name the concrete change each abstraction makes cheaper?
- ☐ Do your domain modules import vendor SDKs?
- ☐ Does every implementation of an interface honour the same contract, errors included?
- ☐ Are there interfaces with one implementation and no fake?

## GO DEEPER

ESSAY

CUPID: for joyful coding Dan North, 2022

---

BOOK

Agile Software Development: Principles, Patterns, and Practices Robert C. Martin · where SOLID was first set out

---

PAPER

A Behavioral Notion of Subtyping Barbara Liskov & Jeannette Wing, 1994

# Composition and dependency injection

2 exercises

*Build behaviour by combining small objects instead of inheriting it, and hand objects their dependencies instead of letting them find their own. You get code you can rearrange, and seams you can test through.*

## What goes wrong with inheritance

Inheritance couples a subclass to its parent's implementation, not just its interface. Change the internals of a parent method, and a subclass that overrode a different method can break. This is the fragile base class problem. Hierarchies also force a single axis of classification onto things that vary along several.

**BEFORE** a class for every combination

```
class Notifier: ...
class EmailNotifier(Notifier): ...
class SmsNotifier(Notifier): ...
class RetryingEmailNotifier(EmailNotifier): ...
class RetryingSmsNotifier(SmsNotifier): ...
class LoggedRetryingEmailNotifier(RetryingEmailNotifier): ...
# two channels x two optional behaviours, and still counting
```

**AFTER** each behaviour written once, combined where used

```
class EmailChannel:
    def send(self, to, message): ...

class SmsChannel:
    def send(self, to, message): ...

class Retrying:
    def __init__(self, inner, attempts=3):
        self.inner, self.attempts = inner, attempts

    def send(self, to, message):
        for attempt in range(1, self.attempts + 1):
            try:
                return self.inner.send(to, message)
            except TransientError:
                if attempt == self.attempts:
                    raise

class Logged:
    def __init__(self, inner, log):
        self.inner, self.log = inner, log
```

```
def send(self, to, message):
    self.log.info("sending", extra={"to": to})
    return self.inner.send(to, message)

notifier = Logged(Retrying(SmsChannel()), log)
```

Adding a push channel is now one class, not four. The *Design Patterns* authors gave this advice in 1994: favour object composition over class inheritance. Inheritance still has its place, in a genuine "is-a" relationship with a stable parent, or in a framework explicitly designed to be subclassed. It just shouldn't be the default.

## Dependency injection, without a framework

Dependency injection sounds like a technology. It is only this: an object receives the things it depends on instead of constructing or looking them up. Pass them to the constructor.

**BEFORE** the job finds its own dependencies

```
class ReminderJob:
    def run(self):
        db = Database(os.environ["DATABASE_URL"])
        mailer = Smtplib.SMTP("smtp.example.com")
        today = date.today()
        for invoice in db.overdue_invoices(today):
            mailer.send(invoice.email, reminder_text(invoice, today))
```

**AFTER** dependencies arrive; one place wires them

```
class ReminderJob:
    def __init__(self, invoices, mailer, clock):
        self.invoices, self.mailer, self.clock = invoices, mailer, clock

    def run(self):
        today = self.clock.today()
        for invoice in self.invoices.overdue(today):
            self.mailer.send(invoice.email, reminder_text(invoice, today))

# main.py, the composition root: the only place that knows concrete classes.
def build_reminder_job():
    return ReminderJob(
        invoices=SqlInvoices(Database(os.environ["DATABASE_URL"])),
        mailer=Smtplib.SMTP("smtp.example.com"),
        clock=SystemClock(),
    )
```

All construction happens in one place, near the program's entry point. Everything else receives what it needs. You can read `main.py` to see how the application is wired, and a test can wire it differently.

## Time and randomness are dependencies

Code that calls `datetime.now()`, `random.random()` or `uuid4()` directly is nondeterministic, and nondeterministic code produces flaky tests. Treat the clock and the random source like any other dependency.

**EXAMPLE** a test that controls time

```
class FixedClock:
    def __init__(self, today):
        self._today = today

    def today(self):
        return self._today

def test_reminder_says_how_many_days_overdue():
    outbox = RecordingMailer()
    job = ReminderJob(
        invoices=InMemoryInvoices([invoice(due=date(2026, 5, 1))]),
        mailer=outbox,
        clock=FixedClock(date(2026, 5, 11)),
    )
    job.run()
    assert "10 days overdue" in outbox.sent[0].body
```

## Seams

Michael Feathers defines a **seam** as a place where you can change what a program does without editing it at that place. Constructor injection creates seams on purpose:

`ReminderJob` behaves differently depending on what you pass it, and not one line inside it changes.

Python also has weaker seams. `monkeypatch` and `unittest.mock.patch` swap names at runtime. They work, and they're essential for legacy code, but they couple tests to module structure: move an import and the patch silently stops applying. Prefer seams you design over seams you patch.

## The same idea at architecture scale

Alistair Cockburn's **hexagonal architecture**, also called ports and adapters, applies this to a whole application. The core declares *ports*, the interfaces it needs. *Adapters* implement them for real infrastructure: a Postgres adapter, a payment provider adapter, an HTTP controller. Tests plug in fakes. It's the dependency inversion from chapter 5, repeated at every boundary.

---

### COMMON TRAPS

**A constructor with nine parameters.** Injection makes dependencies visible, and sometimes what you see is a class doing too much. Split it rather than hiding the list inside a container.

**Service locators.** A global registry you ask for things, like `registry.get("mailer")`, hides the dependencies again. It looks like injection and delivers none of the benefits.

**Framework first.** Containers help in large applications. In small ones, plain constructor arguments are clearer. Understand the plain version before reaching for the magic.

**Wrappers nobody can read.** `Logged(Retrying(Metered(Cached(Timed(client)))))` is hard to debug. Build it once, in the composition root, and give it a name.

## EXERCISES

### EXERCISE 6.1 Replace the hierarchy

This hierarchy exists only to add behaviour. Rewrite it with composition so caching and logging combine freely with either source.

```
class PriceSource:
    def price(self, sku): ...

class ApiPriceSource(PriceSource): ...
class CsvPriceSource(PriceSource): ...
class CachedApiPriceSource(ApiPriceSource): ...
class CachedLoggedApiPriceSource(CachedApiPriceSource): ...
```

### SOLUTION

```
class Cached:
    def __init__(self, inner):
        self.inner, self._cache = inner, {}

    def price(self, sku):
        if sku not in self._cache:
            self._cache[sku] = self.inner.price(sku)
        return self._cache[sku]

class Logged:
    def __init__(self, inner, log):
        self.inner, self.log = inner, log

    def price(self, sku):
        result = self.inner.price(sku)
        self.log.debug("price lookup", extra={"sku": sku, "price": str(result)})
        return result

api_prices = Cached(Logged(ApiPriceSource(), log))
csv_prices = Cached(CsvPriceSource("prices.csv"))
```

The order of wrapping matters. `Cached(Logged(...))` logs only cache misses, while `Logged(Cached(...))` logs every lookup. Composition makes that choice explicit and visible; the inheritance version buried it in the hierarchy.

## EXERCISE 6.2 Make it deterministic

This function can't be tested reliably. Change its signature, not its behaviour, so a test can control both the time and the generated code.

```
def issue_gift_card(amount):
    code = "".join(random.choices(string.ascii_uppercase + string.digits, k=12))
    expires = datetime.now(timezone.utc) + timedelta(days=365)
    return GiftCard(code=code, amount=amount, expires_at=expires)
```

### SOLUTION

```
def issue_gift_card(amount, *, now, rng):
    alphabet = string.ascii_uppercase + string.digits
    code = "".join(rng.choices(alphabet, k=12))
    return GiftCard(code=code, amount=amount, expires_at=now + timedelta(days=365))

def test_gift_card_expires_one_year_after_issue():
    issued_at = datetime(2026, 1, 15, tzinfo=timezone.utc)
    card = issue_gift_card(Money.usd("50"), now=issued_at, rng=random.Random(42))
    assert card.expires_at == datetime(2027, 1, 15, tzinfo=timezone.utc)
    assert len(card.code) == 12
```

`random.Random(42)` is a seeded generator with the same interface as the module functions. In production, pass `random.SystemRandom()`, which draws from the operating system's secure source. That matters here, because gift card codes must be unguessable, and the original code's use of the default generator was a security bug hiding in plain sight.

### CHECKLIST

- ☐ Is behaviour added by combining objects rather than subclassing?
- ☐ Does each class receive its dependencies instead of constructing them?
- ☐ Is there one composition root that wires the application?
- ☐ Are the clock, random sources and ID generators injectable?

- ☐ Would a test need `patch` to control this code? Could it take an argument instead?

#### GO DEEPER

**BOOK**      [Working Effectively with Legacy Code](#) Michael Feathers · the chapter on seams

---

**ESSAY**      [Hexagonal Architecture](#) Alistair Cockburn · ports and adapters from the source

---

**BOOK**      [Design Patterns](#) Gamma, Helm, Johnson & Vlissides · the introduction's case for composition

# Design patterns worth knowing

3 exercises

*Patterns are names for solutions that keep recurring. Knowing them lets you recognise a problem's shape and discuss it in one word. The deeper skill is noticing when your language already solves the problem, or when you don't have the problem at all.*

## Where patterns came from

*Design Patterns*, published in 1994 by Gamma, Helm, Johnson and Vlissides, catalogued twenty-three recurring solutions in object-oriented code. Many of them work around limits of C++ and early Java, languages without first-class functions. In Python, JavaScript or Kotlin several shrink to a line: Strategy is passing a function, Command is a closure, Iterator is a generator, Singleton is a module.

That doesn't make the catalogue obsolete. The problems are still real, and the names are still the fastest way to describe a structure in a design review. What changes is how much ceremony the solution needs.

## Adapter

Wrap something you don't control behind an interface you do. Every third-party API your domain touches should go through one. It isolates the vendor's quirks, gives you one place to handle their errors, and gives tests something to fake.

EXAMPLE geocoding.py

```
class GeocoderAdapter:
    """Our interface: coordinates for an address, or None if it's unknown."""

    def __init__(self, client):
        self.client = client

    def locate(self, address: Address) -> Coordinates | None:
        try:
            result = self.client.geocode(query=address.one_line(), limit=1)
        except VendorRateLimited as e:
            raise GeocodingUnavailable() from e
        if not result.features:
            return None
        lon, lat = result.features[0].center    # this vendor returns lon first
        return Coordinates(lat=lat, lon=lon)
```

The vendor's longitude-first quirk now lives in exactly one place.

## Strategy

Choose an algorithm at runtime. In languages with first-class functions, a strategy is simply a function you pass in.

### EXAMPLE

```
def rank_products(products, score):
    return sorted(products, key=score, reverse=True)

rank_products(catalog, score=lambda p: p.units_sold_30d)
rank_products(catalog, score=lambda p: p.rating * math.log1p(p.review_count))
```

## Decorator

Add behaviour around an object or function without changing it. The `Retrying` and `Logged` wrappers in chapter 6 are decorators. Python's `@` syntax applies the same idea to functions.

### EXAMPLE

```
def retry(attempts=3, on=(TransientError,)):
    def wrap(fn):
        @functools.wraps(fn)
        def inner(*args, **kwargs):
            for attempt in range(1, attempts + 1):
                try:
                    return fn(*args, **kwargs)
                except on:
                    if attempt == attempts:
                        raise
            return inner
        return wrap

@retry(attempts=5)
def fetch_exchange_rates(): ...
```

Keep decorators few and obvious. A stack of them that changes a function's signature or control flow makes it hard to see what a call actually does.

## Facade

One simple entry point in front of a complicated subsystem. It's chapter 4's `deep` module under another name: the `Mailer.send()` of exercise 4.1 is a facade over connection handling, authentication, message building and retries.

## Observer

Let parts of a system react to an event without the event's source knowing about them. When an order ships, send an email, record a metric and notify the warehouse, without the shipping code importing any of them.

#### EXAMPLE

```
class Events:
    def __init__(self):
        self._handlers = defaultdict(list)

    def subscribe(self, event_type, handler):
        self._handlers[event_type].append(handler)

    def publish(self, event):
        for handler in self._handlers[type(event)]:
            handler(event)

events.subscribe(OrderShipped, send_shipping_email)
events.subscribe(OrderShipped, record_shipment_metric)
events.publish(OrderShipped(order_id=42, carrier="UPS"))
```

The cost is invisible control flow. Reading `publish` tells you nothing about what will happen, and one failing handler can surprise the publisher. Use it where the decoupling is worth that, and keep subscriptions in one easy-to-find place.

## Factory

Put complex or conditional construction in one place. A function that returns the right object for a configuration is a factory. You rarely need a class for it.

#### EXAMPLE

```
def payment_gateway(config) -> PaymentGateway:
    if config.environment == "test":
        return FakePayments()
    if config.provider == "stripe":
        return StripeGateway(api_key=config.stripe_key)
    if config.provider == "adyen":
        return AdyenGateway(api_key=config.adyen_key, merchant=config.merchant)
    raise UnknownProvider(config.provider)
```

## Repository

Not in the 1994 book, but everywhere since Martin Fowler's catalogue of enterprise patterns and Eric Evans' domain-driven design. A repository offers a collection-like interface over storage, such as `orders.get(id)`, `orders.add(order)` and `orders.overdue(today)`, so domain code never sees SQL. It's also what chapter 9's fakes replace.

## State machine

When an entity moves through a lifecycle, make its states and legal transitions explicit instead of scattering status checks across the codebase. The classic version uses a class per state. A transition table is often clearer.

**EXAMPLE** `order_lifecycle.py`

```
TRANSITIONS = {
    "pending": {"pay": "paid", "cancel": "cancelled"},
    "paid": {"ship": "shipped", "refund": "refunded"},
    "shipped": {"deliver": "delivered", "return": "returned"},
    "delivered": {"return": "returned"},
}

def transition(order, action):
    allowed = TRANSITIONS.get(order.status, {})
    if action not in allowed:
        raise IllegalTransition(order.status, action)
    order.status = allowed[action]
```

Every legal move is visible in one table. "Can a cancelled order be shipped?" is answered by reading, not by searching.

## One anti-pattern: Singleton

Singleton promises one instance and a global access point. The first is occasionally needed. The second is global state under a respectable name, which brings hidden dependencies, tests that depend on order, and code that can't run with two configurations. If you need one instance, create one in the composition root and pass it to whatever needs it.

## Pattern-itis

The Spring framework for Java contains a real class called `AbstractSingletonProxyFactoryBean`. It's often cited as the logical end point of applying patterns because they exist rather than because a problem needs them. The test for a pattern is the one this volume applies everywhere: does it make the next change cheaper? If you can't name the change, don't add the pattern.

| PATTERN          | PROBLEM IT SOLVES                      | USUAL PYTHON FORM                           | SKIP IT WHEN                              |
|------------------|----------------------------------------|---------------------------------------------|-------------------------------------------|
| <b>Adapter</b>   | A vendor's API doesn't fit your domain | A small class wrapping the client           | Almost never, for third-party APIs        |
| <b>Strategy</b>  | The algorithm varies by context        | A function parameter                        | Only one algorithm exists                 |
| <b>Decorator</b> | Behaviour needed around a call         | A wrapper class, or <code>@decorator</code> | The behaviour belongs inside the function |

|                      |                                         |                                      |                                        |
|----------------------|-----------------------------------------|--------------------------------------|----------------------------------------|
| <b>Facade</b>        | A subsystem is complicated to use       | A deep module with few methods       | The subsystem is already simple        |
| <b>Observer</b>      | Many reactions to one event             | Handlers registered by event type    | There's one reaction: call it directly |
| <b>Factory</b>       | Construction is conditional or complex  | A function returning an interface    | Construction is one line               |
| <b>Repository</b>    | The domain shouldn't know about storage | A class with collection-like methods | A small script with a few queries      |
| <b>State machine</b> | A lifecycle with rules about moves      | A transition table                   | Two states and one transition          |

## COMMON TRAPS

**Pattern first, problem second.** Starting a design by choosing patterns is backwards. Good patterns emerge from refactoring towards a problem you actually have.

**Java written in Python.** A `StrategyFactory` returning `Strategy` subclasses is several files of ceremony for what a dictionary of functions does.

**Hidden control flow.** Observers, decorators and injection frameworks all make it harder to see what runs when. Budget for that cost.

## EXERCISES

### EXERCISE 7.1 Collapse the ceremony

Rewrite this with the same behaviour and a fraction of the code.

```
class ShippingStrategy(ABC):
    @abstractmethod
    def cost(self, weight_kg): ...

class StandardShipping(ShippingStrategy):
    def cost(self, weight_kg):
        return 4.99 + 0.5 * weight_kg

class ExpressShipping(ShippingStrategy):
    def cost(self, weight_kg):
        return 12.99 + 1.2 * weight_kg

class ShippingStrategyFactory:
    def create(self, name):
        return {"standard": StandardShipping, "express": ExpressShipping}[name]()
```

```
cost = ShippingStrategyFactory().create(order.shipping).cost(order.weight_kg)
```

## SOLUTION

```
SHIPPING_RATES = {
    "standard": lambda kg: 4.99 + 0.5 * kg,
    "express": lambda kg: 12.99 + 1.2 * kg,
}

cost = SHIPPING_RATES[order.shipping](order.weight_kg)
```

If rates later need their own state or several methods, promote them back into classes. That's a refactoring, not a rewrite, and you'll be doing it because you have the problem rather than in case you do.

## EXERCISE 7.2 Write an adapter

Your domain needs `send_sms(to: PhoneNumber, text: str) -> None`, raising `SmsUnavailable` on transient failure and `InvalidRecipient` for a bad number. The vendor SDK behaves like this, and you can't change it:

```
client = TextBlastClient(api_key)
resp = client.messages.create(dest="+15551234567", body="hi", sender_id="SHOP")
# resp.status is "queued", "rejected" or "throttled"
# network problems raise textblast.ConnectionFailure
```

## SOLUTION

```
class TextBlastSms:
    def __init__(self, client, sender_id="SHOP"):
        self.client, self.sender_id = client, sender_id

    def send_sms(self, to: PhoneNumber, text: str) -> None:
        try:
            resp = self.client.messages.create(
                dest=to.e164(), body=text, sender_id=self.sender_id)
        except textblast.ConnectionFailure as e:
            raise SmsUnavailable() from e
        if resp.status == "throttled":
            raise SmsUnavailable()
        if resp.status == "rejected":
            raise InvalidRecipient(to)
```

The adapter translates three things: the phone number format, the vendor's status strings, and the vendor's exceptions. Domain code sees none of them, and any object with a `send_sms` method can stand in for it in a test.

### EXERCISE 7.3 Make the lifecycle explicit

Status checks for support tickets are scattered across a codebase. From these four snippets, build the transition table, then find the transitions the code allows that the business almost certainly doesn't want.

```
if ticket.status == "open": ticket.status = "in_progress"           # assign
if ticket.status in ("open", "in_progress"): ticket.status = "resolved" # resolve
if ticket.status == "resolved": ticket.status = "closed"           # close
if ticket.status != "closed": ticket.status = "open"               # reopen
```

#### SOLUTION

```
TRANSITIONS = {
    "open": {"assign": "in_progress", "resolve": "resolved", "reopen": "open"},
    "in_progress": {"resolve": "resolved", "reopen": "open"},
    "resolved": {"close": "closed", "reopen": "open"},
    "closed": {},
}
```

The suspect rule is `reopen`, which applies to every status except closed. On an open ticket it's a pointless no-op. On an in-progress ticket it silently un-assigns work someone is doing. Scattered checks hid both; the table makes them obvious and gives you one place to decide.

#### CHECKLIST

- ☐ For each pattern in your change, can you name the problem it solves here?
- ☐ Could a function or a dictionary replace a class hierarchy?
- ☐ Does every third-party API pass through an adapter you own?
- ☐ Are lifecycle rules in one place?
- ☐ Did you create any global state?

#### GO DEEPER

**FREE** [Python Design Patterns](#) Brandon Rhodes · which classic patterns still matter in Python

**BOOK** [Design Patterns](#) Gamma, Helm, Johnson & Vlissides · read each pattern when you meet it



## PART III

# Tests

Tests are what make every other chapter safe to act on. These three cover what tests are for and how to shape a suite, how to replace dependencies without writing tests that lie, and four techniques that each catch a kind of mistake the others miss.

# What tests are for

3 exercises

*A test suite exists to let you change code with confidence. A good test fails only when behaviour you care about breaks, and when it fails, it tells you what broke.*

## Four jobs a test suite does

- **Catch regressions.** What worked yesterday still works today.
- **Make refactoring safe.** Without tests, every structural improvement is a gamble, so people stop making them, and complexity wins.
- **Document behaviour.** A well-named test is an executable example of what the code promises. Unlike a comment, it can't silently drift.
- **Give design feedback.** Code that's painful to test is usually painful to use: too many dependencies, hidden state, mixed responsibilities.

## What makes a good test

Kent Beck lists twelve desirable properties of tests, and notes that they trade off against each other. Six matter most day to day.

- **Behavioural.** It checks what the code does, not how. Change the implementation without changing behaviour, and the test still passes.
- **Specific.** When it fails, the cause is obvious from its name and message.
- **Deterministic.** Same code, same result, every run. No dependence on time, randomness, ordering or the network.
- **Fast.** Fast enough that you run it after every small change.
- **Isolated.** It doesn't need other tests to run first, and leaves nothing behind.
- **Readable.** A newcomer can see what's being claimed without reading the code under test.

## Anatomy of a test

Arrange, act, assert: set up a situation, perform one action, check the outcome. Behaviour-driven teams say given, when, then. Either way, one test checks one behaviour, and its name states that behaviour as a fact.

**BEFORE** three claims, a private method, and a vague name

```
def test_coupon():
    c = Coupon("SAVE10", percent=10, expires=date(2026, 1, 1))
    o = Order(subtotal=Money.usd("100"))
    assert c.apply(o, today=date(2025, 12, 31)).total == Money.usd("90")
    assert c._expired(date(2026, 1, 2)) is True
    o2 = Order(subtotal=Money.usd("0"))
    assert c.apply(o2, today=date(2025, 12, 31)).total == Money.usd("0")
```

**AFTER** one behaviour each, through the public interface

```
def test_valid_coupon_takes_percentage_off_subtotal():
    coupon = Coupon("SAVE10", percent=10, expires=date(2026, 1, 1))
    order = Order(subtotal=Money.usd("100"))

    discounted = coupon.apply(order, today=date(2025, 12, 31))

    assert discounted.total == Money.usd("90")

def test_expired_coupon_is_rejected():
    coupon = Coupon("SAVE10", percent=10, expires=date(2026, 1, 1))
    order = Order(subtotal=Money.usd("100"))

    with pytest.raises(CouponExpired):
        coupon.apply(order, today=date(2026, 1, 2))
```

When the first version fails, its name says only "coupon", and it has reached into a private method that a refactoring may rename. When one of the second pair fails, the failure reads like a bug report.

## Table-driven tests

When one behaviour needs many examples, parametrize instead of copying.

**EXAMPLE** boundaries, named

```
@pytest.mark.parametrize("subtotal, expected", [
    ("0.00", "0.00"),      # nothing to discount
    ("49.99", "49.99"),    # just below the threshold
    ("50.00", "45.00"),    # exactly at the threshold
    ("200.00", "180.00"),  # well above
])
def test_ten_percent_off_orders_of_fifty_or_more(subtotal, expected):
    assert apply_bulk_discount(Money.usd(subtotal)) == Money.usd(expected)
```

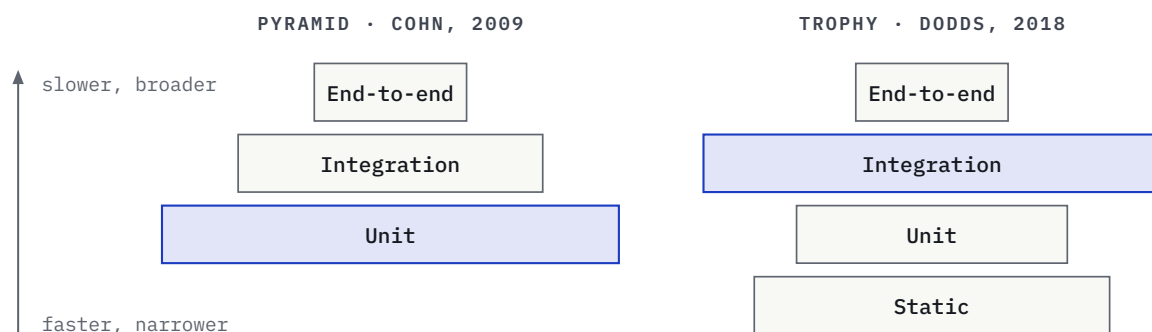
Each comment names why the case exists. Just below, at, and just above a boundary is where bugs live.

## The kinds of test

| KIND            | WHAT IT EXERCISES                                                        | TYPICAL SPEED         | WHAT IT CATCHES                                        |
|-----------------|--------------------------------------------------------------------------|-----------------------|--------------------------------------------------------|
| Static analysis | Types and lint rules, without running code                               | Instant               | Typos, wrong types, dead code                          |
| Unit            | One unit of behaviour, often one module                                  | Milliseconds          | Logic errors                                           |
| Integration     | Your code plus one real dependency, such as a database or an HTTP server | Tens of ms to seconds | Wrong queries, serialisation mismatches, configuration |
| Contract        | Whether a provider still meets what its consumers expect                 | Seconds               | Breaking API changes between services                  |
| End-to-end      | The whole system through its real interface                              | Seconds to minutes    | Wiring, deployment, cross-service flows                |

The word "unit" is contested. Martin Fowler distinguishes **solitary** unit tests, which replace every collaborator with a double, from **sociable** ones, which let real collaborators run and replace only those that are slow or nondeterministic. Much of the argument about testing strategy is people using the same word for these two different things.

## Pyramids, trophies and honeycombs



*The same kinds of test with different emphasis. Width shows roughly how many of each kind; the highlighted layer is where each model puts most of its effort.*

Mike Cohn's test pyramid says: many unit tests, fewer integration tests, very few end-to-end tests, because the higher you go, the slower and more brittle tests become. Kent C. Dodds' testing trophy puts integration tests at the widest point and adds static analysis as the base,

arguing they give the most confidence per test in modern applications. Spotify's honeycomb makes a similar case for microservices, where most bugs live at the boundaries between services.

Martin Fowler's verdict on the debate is the useful one. The shapes differ mostly because people define "unit" differently, and the ratio matters far less than quality. Most teams would gain more from tests that run fast, fail reliably, and fail only for useful reasons than from any change of shape.

## Test behaviour, not implementation

A test that asserts on private methods, internal data structures, or the exact sequence of internal calls fails whenever the implementation changes, even when behaviour doesn't. Those failures teach a team that refactoring breaks tests, and so the team stops refactoring. Test through the same interface your callers use.

## Coverage

Coverage reports which lines ran during the tests. It's a useful negative signal: uncovered code is certainly untested. It's a weak positive signal, because a line can run without any assertion checking what it did. Setting a coverage target turns it into a goal, and Goodhart's law applies: when a measure becomes a target, it stops being a good measure. Chapter 10 describes mutation testing, which measures what coverage only pretends to.

## Flaky tests

A flaky test passes and fails on the same code. It's worse than having no test, because it teaches everyone to ignore red. The usual causes are real time, unseeded randomness, state shared between tests, test ordering, real network calls, and asynchronous work checked before it finishes. Fix a flaky test the day you find it, or quarantine it with a ticket and an owner. Never simply rerun until it's green.

---

### COMMON TRAPS

**Recomputing the expected value.** `assert total(items) == sum(i.price * i.qty for i in items)` repeats the implementation's logic, so it repeats its bugs. Use literal expected values.

---

**Asserting on log output.** Logs are for people, and they change for good reasons. Assert on outcomes.

---

**Testing the framework.** A test that checks your ORM saves a field is a test of the ORM.

---

**One giant test per feature.** It fails for twenty reasons and names none of them.

---

**Sleeping to wait.** `time.sleep(2)` is either too long or, on a slow CI machine, too short. Wait for

a condition, with a timeout.

## EXERCISES

### EXERCISE 8.1 Find the boundaries

List the test cases you'd write for this function, with a short reason for each. Don't write the tests yet.

```
def password_problems(password: str) -> list[str]:  
    """Return the rules this password breaks; an empty list means it's fine.  
  
    Rules: 12 to 128 characters long; at least one letter and one digit;  
    not on the list of common passwords.  
    """
```

### SOLUTION

- 11 characters: one below the minimum.
- 12 characters: exactly the minimum.
- 128 characters: exactly the maximum.
- 129 characters: one above.
- Letters only: missing a digit.
- Digits only: missing a letter.
- The empty string: breaks several rules at once. Are all of them reported?
- `password1234` : twelve characters, has both letters and digits, yet is common. It tests the list rule independently of the others.
- Non-ASCII letters such as `é` or `ß` : do they count as letters?
- Multi-byte characters near the length limits: is length measured in characters or in bytes?
- Leading or trailing spaces: are they stripped, or do they count?

The last three are questions, not answers. Writing test cases is how you discover that a specification is incomplete, and it's far cheaper to discover it now than from a support ticket.

### EXERCISE 8.2 Repair a brittle test

Name everything that couples this test to the implementation, then rewrite it to test behaviour.

```
def test_register(caplog):
    svc = RegistrationService(repo, mailer)
    svc.register("ana@example.com", "correct horse battery")
    assert svc._pending[0]["email"] == "ana@example.com"
    assert svc._hash_password.call_count == 1
    assert repo.save.call_args[0][0].email == "ana@example.com"
    assert "Welcome" in caplog.text
```

## SOLUTION

Four couplings: `_pending` is private state; `_hash_password.call_count` asserts how, not what; `repo.save.call_args[0][0]` depends on an exact call signature; and `caplog.text` asserts on logging.

```
def test_registered_user_can_log_in_and_receives_welcome_email():
    users, outbox = InMemoryUsers(), RecordingMailer()
    service = RegistrationService(users, outbox)

    service.register("ana@example.com", "correct horse battery")

    assert service.authenticate("ana@example.com", "correct horse battery")
    assert outbox.sent_to("ana@example.com")[0].subject == "Welcome"
```

The new test states two outcomes that a user would recognise. The password could be hashed differently, the pending list could disappear, and the repository could rename its methods, and the test would keep passing for as long as the behaviour holds.

## EXERCISE 8.3 Diagnose the flake

This test fails about once a week in CI. `is_expired()` is implemented as `datetime.now() - self.last_seen > timedelta(minutes=30)`. Why does it flake, and what should replace it?

```
def test_session_expires_after_thirty_minutes():
    session = Session.start(user_id=7)
    session.last_seen = datetime.now() - timedelta(minutes=30)
    assert session.is_expired()
```

## SOLUTION

There are two calls to `datetime.now()`, at slightly different moments. Usually a few microseconds pass between them, so the difference exceeds thirty minutes and the test passes. When both calls land on the same clock tick, the difference is exactly thirty minutes, `>` is false, and the test fails.

The flake is hiding a real question: should a session be expired at exactly thirty minutes? Inject a clock and pin the boundary on purpose.

```
def test_session_is_still_valid_one_second_before_timeout():
    clock = FakeClock(datetime(2026, 3, 1, 12, 0, 0))
    session = Session.start(user_id=7, clock=clock)
    clock.advance(minutes=29, seconds=59)
    assert not session.is_expired()

def test_session_expires_at_thirty_minutes():
    clock = FakeClock(datetime(2026, 3, 1, 12, 0, 0))
    session = Session.start(user_id=7, clock=clock)
    clock.advance(minutes=30)
    assert session.is_expired()
```

The second test fails against the current implementation, because it uses `>`. That's the point: you've found and decided the boundary instead of leaving it to a race. Change the comparison to `>=`.

#### CHECKLIST

- ☐ Does each test check one behaviour, named as a fact?
- ☐ Would the tests survive a refactoring that doesn't change behaviour?
- ☐ Are expected values literal rather than recomputed?
- ☐ Did you test the boundaries: empty, one, just below, at, just above?
- ☐ Does anything depend on real time, randomness, ordering or the network?

#### GO DEEPER

**ESSAY**      On the Diverse and Fantastical Shapes of Testing Martin Fowler, 2021

**ESSAY**      Test Desiderata Kent Beck · the twelve properties, with short videos

**ESSAY**      The Testing Trophy and Testing Classifications Kent C. Dodds

# Test doubles

3 exercises

*Replace a real dependency only when it's slow, nondeterministic, or has effects you can't allow in a test. When you do, prefer the double that couples the test least to how your code works.*

## The vocabulary

Gerard Meszaros coined "test double", after the stunt double who stands in for an actor, and named five kinds. People use "mock" loosely for all of them, which causes real confusion in code review.

| KIND         | WHAT IT DOES                                                   | TYPICAL USE                                     | WHAT IT VERIFIES               |
|--------------|----------------------------------------------------------------|-------------------------------------------------|--------------------------------|
| <b>Dummy</b> | Fills a parameter; never actually used                         | A logger the tested path never calls            | Nothing                        |
| <b>Stub</b>  | Returns canned answers                                         | An exchange-rate service that always says 1.1   | Nothing; it feeds inputs in    |
| <b>Spy</b>   | Records how it was called                                      | A mailer that remembers what it sent            | Whatever you assert afterwards |
| <b>Mock</b>  | Pre-programmed with expected calls; fails if they don't happen | Checking an audit event is emitted exactly once | Interactions                   |
| <b>Fake</b>  | A working, simplified implementation                           | An in-memory repository                         | State, through normal use      |

## State or behaviour?

A test can check the outcome, meaning the state of the world after the action, or it can check the interactions, meaning which calls were made along the way. Martin Fowler's essay *Mocks Aren't Stubs* describes two schools. **Classicists** use real objects where practical and verify state. **Mockists** replace every collaborator and verify interactions. Both schools produce working software, but the classicist default ages better, because interaction tests encode *how* the code works, and how is exactly what refactoring changes.

## How over-mocking goes wrong

**BEFORE** a test that restates the implementation

```
def test_place_order():
    repo, payments, mailer = Mock(), Mock(), Mock()
    payments.charge.return_value = "ch_1"
    service = OrderService(repo, payments, mailer)

    service.place(cart, "tok_visa")

    payments.charge.assert_called_once_with("tok_visa", cart.total())
    repo.save.assert_called_once()
    mailer.send_receipt.assert_called_once()
```

This test fails if you batch the save, rename `send_receipt`, or move the email into an event handler, and none of those change behaviour. Worse, it passes when the code is broken. If the real `repo.save` requires a field that `place` never sets, the mock accepts the call happily.

**AFTER** outcomes the business would recognise

```
def test_placed_order_is_stored_charged_and_receipted():
    orders, payments, outbox = InMemoryOrders(), FakePayments(), RecordingMailer()
    service = OrderService(orders, payments, outbox)

    order = service.place(cart, "tok_visa")

    assert orders.get(order.id).status == "paid"
    assert payments.total_charged_to("tok_visa") == cart.total()
    assert outbox.sent_to(cart.customer_email)[0].subject.startswith("Receipt")
```

Refactor freely underneath. This test fails only if an order stops being stored, charged, or receipted.

## Don't mock what you don't own

When you mock a third-party library directly, the mock encodes your belief about how the library behaves. If the belief is wrong, or the library changes, your tests keep passing while production breaks. The fix is the adapter from chapter 7. Wrap the library in a small class you own, fake that class in most tests, and test the adapter itself against the real library, or the vendor's sandbox, in a handful of integration tests. The idea comes from Steve Freeman and Nat Pryce, who helped invent mock objects in the first place.

## Keeping fakes honest

A fake is only useful if it behaves like the real thing, including its errors. The technique that guarantees it is a shared contract test: one suite, run against every implementation. It's chapter 5's Liskov substitution, enforced by the test runner.

**EXAMPLE** `test_orders_contract.py`

```
@pytest.fixture(params=["memory", "postgres"])
```

```
def orders(request):
    if request.param == "memory":
        return InMemoryOrders()
    return PostgresOrders(request.getfixturevalue("postgres_db"))

def test_missing_order_raises_not_found(orders):
    with pytest.raises(OrderNotFound):
        orders.get(order_id=999)

def test_saved_order_round_trips(orders):
    order = make_order(total=Money.usd("19.99"))
    orders.add(order)
    assert orders.get(order.id) == order
```

Every test in this file runs twice. If the in-memory fake ever drifts from Postgres, a test fails before anyone relies on the difference. The database is only started for the Postgres runs, because `getfixturevalue` requests it lazily.

## When a mock is the right tool

Checking interactions is right when the interaction *is* the behaviour: an outbound message with no other observable effect. Even then, a spy or a recording fake usually reads better than a strict mock, because you assert on what was sent after the fact instead of scripting expectations up front. Save strict mocks for cases where the exact protocol matters, such as asserting that a retry loop calls a failing dependency exactly three times.

## Python specifics that bite

- **Patch where the name is looked up, not where it's defined.** If `billing.py` does `from payments import charge`, you must patch `billing.charge`. Patching `payments.charge` replaces a name that `billing` no longer uses. This is the most common `mock.patch` bug there is.
- **Use autospec.** A plain `Mock` accepts any method with any arguments, typos included. `create_autospec(RealClass)`, or `patch(..., autospec=True)`, rejects calls the real object would reject.
- **Beware mocks returning mocks.** `client.get().json()["items"]` on a `MagicMock` quietly returns another mock. If a test passes through a chain like that, check it's asserting anything real.

**EXAMPLE** the patch target

```
# billing.py
from payments import charge

def bill(customer):
    return charge(customer.card, customer.balance)
```

```
# test_billing.py
@patch("payments.charge")           # wrong: billing holds its own reference
def test_bill_wrong(mock_charge): ...

@patch("billing.charge", autospec=True)  # right: patch the name billing uses
def test_bill_right(mock_charge): ...
```

## COMMON TRAPS

**Mocking the code under test.** If you patch a method on the object you're testing, you're testing your patch.

**One mock per collaborator, by reflex.** Ask first whether the real collaborator is actually slow or nondeterministic. Often it's a plain function that should simply run.

**Tests that pass with the code deleted.** If a test is all `assert_called` checks on mocks returning mocks, delete the body of the function under test and see whether it notices.

## EXERCISES

### EXERCISE 9.1 Name the double

Classify each as a dummy, stub, spy, mock or fake.

```
# A
class AlwaysApproves:
    def check(self, transaction):
        return FraudVerdict.APPROVED

# B
class InMemoryInventory:
    def __init__(self):
        self.stock = {}
    def add(self, sku, qty):
        self.stock[sku] = self.stock.get(sku, 0) + qty
    def reserve(self, sku, qty):
        if self.stock.get(sku, 0) < qty:
            raise OutOfStock(sku)
        self.stock[sku] -= qty

# C
audit = Mock(spec=AuditLog)
...
audit.record.assert_called_once_with("refund", order_id=42)

# D
service = ReportService(db, logger=None)  # this path never logs

# E
class RecordingSlack:
    def __init__(self):
        self.posts = []
    def post(self, channel, text):
```

```
self.posts.append((channel, text))
```

#### SOLUTION

A is a stub. B is a fake. C is a mock, because it verifies an expected interaction. D is a dummy. E is a spy.

### EXERCISE 9.2 Replace the mocks

Rewrite this test with fakes so that it checks outcomes, not calls.

```
def test_refund():
    orders, payments = Mock(), Mock()
    orders.get.return_value = Order(id=5, status="paid", charge_id="ch_9",
                                    total=Money.usd("30"))
    RefundService(orders, payments).refund(order_id=5)
    payments.refund.assert_called_once_with("ch_9", Money.usd("30"))
    orders.save.assert_called_once()
```

#### SOLUTION

```
def test_refunding_a_paid_order_returns_the_money_and_marks_it_refunded():
    orders = InMemoryOrders([
        Order(id=5, status="paid", charge_id="ch_9", total=Money.usd("30")),
    ])
    payments = FakePayments(existing_charges={"ch_9": Money.usd("30")})

    RefundService(orders, payments).refund(order_id=5)

    assert orders.get(5).status == "refunded"
    assert payments.refunded("ch_9") == Money.usd("30")
```

Two more tests are now cheap to write, and they matter more than this one: refunding an order that's already refunded, and refunding when the payment provider fails. With fakes, each is a few lines.

### EXERCISE 9.3 Find the patch bug

This test passes, but real emails go out every time the suite runs. Why?

```
# notifications.py
from mailer import send_email

def notify_shipped(order):
    send_email(order.email, "Your order has shipped")

# test_notifications.py
@patch("mailer.send_email")
```

```
def test_notify_shipped_does_not_crash(mock_send):
    notify_shipped(order)
```

#### SOLUTION

`notifications` copied `send_email` into its own namespace at import time, so patching `mailer.send_email` replaces a name that `notify_shipped` no longer looks at. Patch `notifications.send_email` instead.

The deeper lesson is that the test passed because it asserted nothing. Add `mock_send.assert_called_once_with(order.email, "Your order has shipped")` and the broken patch shows up immediately. Better still, inject the mailer and drop the patch altogether.

#### CHECKLIST

- ☐ Is each double replacing something slow, nondeterministic or effectful?
- ☐ Do assertions check outcomes rather than call sequences, where possible?
- ☐ Do your fakes share contract tests with the real implementations?
- ☐ Are third-party libraries wrapped in adapters you own?
- ☐ Are patches applied where the name is looked up, with `autospec`?

#### GO DEEPER

**ESSAY**      [Mocks Aren't Stubs](#) Martin Fowler

**BOOK**        [xUnit Test Patterns](#) Gerard Meszaros · the source of the vocabulary

**BOOK**        [Growing Object-Oriented Software, Guided by Tests](#) Freeman & Pryce · the best case for the mockist style

# TDD, properties, snapshots and mutants

3 exercises

*Four techniques, each catching a different kind of mistake. Test-driven development shapes the design; property-based testing finds the inputs you didn't imagine; snapshots guard large outputs; mutation testing tests your tests.*

## Test-driven development

Kent Beck's cycle has three steps. **Red**: write a small test for behaviour that doesn't exist yet, and watch it fail. **Green**: write the simplest code that makes it pass. **Refactor**: clean up both, with the tests as your net. Then repeat, in cycles of a few minutes.

What it gives you: every line of production code exists because a test demanded it, so coverage comes free. Each test is written from the caller's side, so an awkward interface is felt immediately. And you're never more than a few minutes away from working code.

Here is the rhythm, building a parser for durations like `1h30m`.

**CYCLE 1** red, then the simplest green

```
def test_pares_minutes():
    assert parse_duration("45m") == 45 * 60

def parse_duration(text):
    return int(text[:-1]) * 60          # deliberately naive: it passes
```

**CYCLE 2** a second unit forces a lookup table

```
def test_pares_hours():
    assert parse_duration("2h") == 2 * 3600

UNIT_SECONDS = {"h": 3600, "m": 60, "s": 1}

def parse_duration(text):
    return int(text[:-1]) * UNIT_SECONDS[text[-1]]
```

**CYCLE 3** combined units force real parsing

```
def test_pares_combined_units():
    assert parse_duration("1h30m") == 5400

TOKEN = re.compile(r"(\d+)([hms])")
```

```
def parse_duration(text):
    return sum(int(n) * UNIT_SECONDS[u] for n, u in TOKEN.findall(text))
```

**CYCLE 4** malformed input forces a decision

```
@pytest.mark.parametrize("bad", ["", "10", "5x", "1h 30m", "h"])
def test_rejects_malformed_input(bad):
    with pytest.raises(ValueError):
        parse_duration(bad)

def parse_duration(text: str) -> int:
    """Seconds in a duration such as '1h30m', '45s' or '2h'."""
    tokens = TOKEN.findall(text)
    if not tokens or "".join(n + u for n, u in tokens) != text:
        raise ValueError(f"not a duration: {text!r}")
    return sum(int(n) * UNIT_SECONDS[u] for n, u in tokens)
```

Notice what the tests drove. The malformed-input test forced a decision that the happy-path tests never raised: is `1h 30m`, with a space, valid? Here the answer is no, and that decision is now written down where the next person will find it.

**The criticism.** John Ousterhout argues that TDD's tiny increments encourage tactical programming: you optimise each step and never stand back to design the whole. Robert Martin replies that the refactor step is exactly where design happens. The practical middle is to test-drive behaviours rather than methods, and to stop and think about structure every few cycles.

TDD shines on well-understood logic, and on bug fixes, where writing the failing test first proves you've actually reproduced the bug. It struggles with exploratory work where you don't yet know what you're building. For that, write a throwaway spike first, learn from it, delete it, and then test-drive the real thing.

## Property-based testing

Example-based tests check the cases you thought of. A property-based test states something that must hold for *every* input, and lets a library generate hundreds of inputs looking for a counterexample. When it finds one, it **shrinks** it to the smallest input that still fails. The idea began with QuickCheck for Haskell in 2000. In Python the library is Hypothesis; in JavaScript, fast-check.

Four shapes of property cover most needs:

- **Round trip.** Decoding what you encoded gives back the original: `parse(format(x)) == x`.
- **Invariant.** Something always true of the output: a sorted list is ordered, and has the same elements as the input.

- **Oracle.** A simple, slow, obviously correct implementation agrees with the fast one.
- **Idempotence.** Doing it twice equals doing it once: `normalize(normalize(s)) == normalize(s)`.

Suppose we add `format_duration`, the inverse of our parser, and state the round trip.

**EXAMPLE** a property finds what examples missed

```
def format_duration(seconds: int) -> str:
    hours, rest = divmod(seconds, 3600)
    minutes, secs = divmod(rest, 60)
    parts = []
    if hours:
        parts.append(f"{hours}h")
    if minutes:
        parts.append(f"{minutes}m")
    if secs:
        parts.append(f"{secs}s")
    return "".join(parts)

from hypothesis import given, strategies as st

@given(st.integers(min_value=0, max_value=10**7))
def test_format_then_parse_round_trips(seconds):
    assert parse_duration(format_duration(seconds)) == seconds
```

Hypothesis fails almost at once and reports the smallest counterexample: `seconds=0`. `format_duration(0)` returns an empty string, and the parser rightly rejects it. Nobody writing examples by hand thought to try zero. The fix is a decision, format zero as `0s`, after which the property holds for every value in the range.

## Snapshot tests

A snapshot, or golden, test saves the output of a run to a file and fails whenever later output differs. It's the cheapest way to guard something large and structured: a rendered email, an invoice's text, a command-line tool's help output, a JSON response. Approval testing, a technique popularised by Llewellyn Falco, is the same idea with a human approving each change.

**EXAMPLE** using the syrupy plugin for pytest

```
def test_invoice_email_matches_approved_snapshot(snapshot):
    email = render_invoice_email(sample_invoice())
    assert email == snapshot
```

Their weakness is social. When a snapshot fails, the easy move is to regenerate it without reading the diff, and a test that's updated without being read tests nothing. Keep snapshots small enough to review, commit them, and treat a snapshot diff in a pull request as code to

be reviewed.

## Mutation testing

Coverage says a line ran. Mutation testing asks whether any test would notice if that line were wrong. A tool makes many small changes to your code, one at a time: `<` becomes `<=`, `+` becomes `-`, a call is deleted. Each changed program is a **mutant**. If some test fails, the mutant is **killed**. If every test still passes, the mutant **survived**, and you've found behaviour your suite doesn't check.

**EXAMPLE** 100% coverage, and a hole

```
def qualifies_for_free_shipping(total_cents):  
    return total_cents >= 5000  
  
def test_large_order_ships_free():  
    assert qualifies_for_free_shipping(10_000)  
  
def test_small_order_pays_shipping():  
    assert not qualifies_for_free_shipping(1_000)
```

Mutate `>=` into `>`, and both tests still pass. The mutant survives, because the suite never checks the boundary, which is exactly where a customer with a basket of precisely fifty dollars will find the bug. Add `assert qualifies_for_free_shipping(5000)` and it dies.

Tools include mutmut and Cosmic Ray for Python, Stryker for JavaScript, TypeScript, C# and Scala, PIT for Java, and cargo-mutants for Rust. Mutation testing is slow, since it runs the suite once per mutant, so point it at critical modules or at changed code rather than everything on every commit.

---

### COMMON TRAPS

**TDD as dogma.** Test-first is a technique for getting feedback, not a moral position. Use it where the feedback is worth having.

**Weak properties.** `assert isinstance(result, str)` holds for every input and proves almost nothing. A good property would fail for a plausible bug.

**Rubber-stamped snapshots.** If your team updates snapshots in bulk without reading them, delete the snapshots.

**Chasing a perfect mutation score.** Some mutants are *equivalent*: the change doesn't alter behaviour, so no test can kill them. Aim to understand the survivors, not to eliminate every one.

**EXERCISE 10.1 Test-drive a slug**

Using red, green, refactor, build `slugify(title)` for blog URLs: lower case, words joined by single hyphens, only a-z and 0-9, and no leading or trailing hyphen. Write one failing test at a time, and keep a note of each cycle.

**SOLUTION**

One reasonable sequence of cycles:

1. `slugify("Hello") == "hello"` drives lower-casing.
2. `slugify("Hello World") == "hello-world"` drives joining words.
3. `slugify("Hello, World!") == "hello-world"` drives dropping punctuation and collapsing runs.
4. `slugify(" Hello ") == "hello"` drives trimming.
5. `slugify("Crème brûlée")` forces a decision about accents: strip them to get `creme-brulee`, or reject them.
6. `slugify("!!!")` forces a decision about empty results.

```
import re
import unicodedata

def slugify(title: str) -> str:
    ascii_title = (unicodedata.normalize("NFKD", title)
                  .encode("ascii", "ignore").decode())
    words = re.findall(r"[a-z0-9]+", ascii_title.lower())
    if not words:
        raise ValueError(f"no usable characters in {title!r}")
    return "-".join(words)
```

Cycles 5 and 6 are the valuable ones. They are questions a product owner should answer, and TDD surfaced them before a user did.

**EXERCISE 10.2 Find the properties**

Write at least two properties for `dedupe(items)`, which removes duplicates while keeping each item's first occurrence, in order.

**SOLUTION**

```
@given(st.lists(st.integers()))
def test_dedupe_properties(items):
    result = dedupe(items)
```

```

assert len(result) == len(set(result))    # no duplicates remain
assert set(result) == set(items)         # nothing lost, nothing invented
assert dedupe(result) == result           # idempotent

# Oracle: slow, but obviously correct. Quadratic time is fine in a test.
assert result == [x for i, x in enumerate(items) if x not in items[:i]]

```

The first two properties would pass for a `dedupe` that scrambled the order. Only the oracle pins the ordering, which is why it's worth having even though it's slow.

### EXERCISE 10.3 Hunt the survivors

All tests pass and coverage is 100%. Imagine a tool that tries every other comparison operator in place of `<=`. Which mutants survive, which of them matter, and what test would kill it?

```

def late_fee_cents(days_late: int) -> int:
    if days_late <= 0:
        return 0
    return min(days_late * 100, 2000)

def test_no_fee_when_on_time():
    assert late_fee_cents(0) == 0

def test_fee_grows_per_day():
    assert late_fee_cents(3) == 300

def test_fee_is_capped():
    assert late_fee_cents(90) == 2000

```

#### SOLUTION

`<= 0` into `< 0` survives, and it's equivalent. With zero days, the formula returns `min(0, 2000)`, which is zero anyway, so no test can ever tell the difference. Ignore it.

`<= 0` into `== 0` survives, and it matters. A negative `days_late`, from a due date in the future or a clock problem, would return a negative fee: a credit. Kill it with `assert late_fee_cents(-5) == 0`.

While you're there, pin the start of the cap with `late_fee_cents(19) == 1900` and `late_fee_cents(20) == 2000`. Those tests document exactly where the rule changes.

- ☐ For a bug fix, did you write the failing test before the fix?
- ☐ For parsing, formatting or serialisation code, is there a round-trip property?
- ☐ Are snapshots small enough that someone actually reads the diff?
- ☐ For your most critical module, have you run mutation testing and read the survivors?

#### GO DEEPER

**BOOK**      Test-Driven Development: By Example Kent Beck · short, and still the best introduction

---

**FREE**      Hypothesis documentation property-based testing for Python

---

**FREE**      Stryker Mutator the clearest introduction to mutation testing

---

**TALK**      Is TDD Dead? Beck, Fowler and Heinemeier Hansson, 2014

## PART IV

# Changing code safely

Most of an engineer's working life is spent changing code that already exists. These two chapters cover improving structure without changing behaviour, and getting a safety net under code that has none.

# Refactoring

3 exercises

*Refactoring is changing the structure of code without changing what it does, in steps small enough that you're never more than a minute or two from passing tests.*

## What refactoring is, and isn't

Martin Fowler's definition is strict on purpose. A refactoring preserves behaviour. Fixing a bug while restructuring isn't refactoring, and neither is a rewrite. Kent Beck describes two hats: when you add function, you don't restructure, and when you restructure, you don't add function. You can swap hats every few minutes, but you wear one at a time, and ideally each gets its own commit.

The discipline is what makes it safe. Each step is small and mechanical, the tests run after every one, and if anything goes red you undo the last step rather than debugging it.

## When to refactor

- **Preparatory.** Before adding a feature, reshape the code so the feature is easy to add. Beck's version: first make the change easy, then make the easy change.
- **Comprehension.** When you've worked out what confusing code does, encode that understanding in names and structure before you forget it.
- **Litter-picking.** Leave code a little better than you found it. Small, continuous, and part of normal work rather than a separate project.
- **The rule of three.** The first time, just write it. The second time, wince at the duplication. The third time, refactor.

When not to: code that works, is rarely read, and never changes. Refactoring pays off through future changes. If there won't be any, there's no payoff.

## Smells, and the refactorings that fix them

Fowler and Beck called the symptoms of design problems **code smells**. They aren't bugs. They're signs that the next change will cost more than it should. The second edition of *Refactoring* lists twenty-four. These twelve account for most of what you'll meet.

| SMELL                  | WHAT YOU NOTICE                                    | USUAL REFACTORING |
|------------------------|----------------------------------------------------|-------------------|
| <b>Mysterious name</b> | You can't tell what it is without reading its body | Rename            |

|                            |                                                           |                                                         |
|----------------------------|-----------------------------------------------------------|---------------------------------------------------------|
| <b>Duplicated code</b>     | The same logic in two places, slowly drifting apart       | Extract function, after sliding the statements together |
| <b>Long function</b>       | You scroll, and comments mark the sections                | Extract function; decompose conditional                 |
| <b>Long parameter list</b> | Calls are hard to read and easy to get wrong              | Introduce parameter object; pass the whole object       |
| <b>Global data</b>         | Anything can change it, and nothing says what does        | Encapsulate the variable behind functions               |
| <b>Mutable data</b>        | A value changes here and breaks something there           | Split variable; separate query from modifier            |
| <b>Divergent change</b>    | One module changes for several unrelated reasons          | Split phase; extract class                              |
| <b>Shotgun surgery</b>     | One change needs small edits in many modules              | Move function, combining scattered code into one module |
| <b>Feature envy</b>        | A function uses another object's data more than its own   | Move the function to where the data lives               |
| <b>Data clumps</b>         | The same few fields always travel together                | Introduce a parameter object or value type              |
| <b>Primitive obsession</b> | Money as a float, emails as strings, statuses as integers | Replace primitive with object                           |
| <b>Repeated switches</b>   | The same if/elif on a type in several places              | Replace conditional with polymorphism, or a table       |

## A worked refactoring

This function produces a text receipt. It works. The task is to add an HTML receipt, and the current structure makes that painful because calculating and formatting are woven together in one loop.

**BEFORE** receipt.py

```
def receipt(order, customer):
    total = 0
    points = 0
```

```

lines = [f"Receipt for {customer.name}"]
for item in order.items:
    amount = item.unit_price * item.quantity
    if item.category == "clearance":
        amount = amount * 0.5
    points += int(amount // 10)
    if item.category == "books":
        points += item.quantity
    total += amount
    lines.append(f"  {item.name} x{item.quantity}: {amount:.2f}")
lines.append(f"Total: {total:.2f}")
lines.append(f"You earned {points} loyalty points")
return "\n".join(lines)

```

- 01 **Extract function** for a line's amount, so the clearance rule gets a name. Run the tests. Green. Commit.
- 02 **Extract function** for a line's loyalty points, for the same reason. Green. Commit.
- 03 **Split the loop.** One loop was doing three jobs: summing money, summing points and formatting. Give each its own. If that worries you for performance, remember there are a handful of items, and you can measure later if it ever matters.
- 04 **Replace temporary variables with queries.** `total` and `points` become values computed from the order.
- 05 **Split phase.** Separate calculating the receipt's data from rendering it.

**AFTER** receipt.py · five green steps later

```

def line_amount(item):
    amount = item.unit_price * item.quantity
    if item.category == "clearance":
        amount = amount * 0.5
    return amount

def line_points(item):
    points = int(line_amount(item) // 10)
    if item.category == "books":
        points += item.quantity
    return points

@dataclass(frozen=True)
class ReceiptLine:
    name: str
    quantity: int
    amount: float

@dataclass(frozen=True)
class ReceiptData:
    customer_name: str
    lines: tuple[ReceiptLine, ...]
    total: float
    points: int

```

```

def receipt_data(order, customer) -> ReceiptData:
    lines = tuple(ReceiptLine(i.name, i.quantity, line_amount(i)) for i in order.items)
    return ReceiptData(
        customer_name=customer.name,
        lines=lines,
        total=sum(line.amount for line in lines),
        points=sum(line_points(i) for i in order.items),
    )

def render_text(data: ReceiptData) -> str:
    rows = [f"Receipt for {data.customer_name}"]
    rows += [f"  {line.name} x{line.quantity}: {line.amount:.2f}" for line in data.lines]
    rows.append(f"Total: {data.total:.2f}")
    rows.append(f"You earned {data.points} loyalty points")
    return "\n".join(rows)

def receipt(order, customer):
    return render_text(receipt_data(order, customer))

```

The original function still exists with the same signature, so no caller changed. Checked against the original on two thousand random orders, it printed the same receipt for almost all of them, but about one in twenty totals came out a cent different. Since Python 3.12, `sum()` adds floats more accurately than a running total does, so the split loop rounds differently. A change that alters output, even by a cent, isn't a pure refactoring, and only a check that broad would have caught it. The cure is the next section's: stop keeping money in floats. Adding `render_html(data)` is now one new function that shares all of the calculation. That's preparatory refactoring: five small steps, each green, and a feature that was awkward has become easy.

## Replace primitive with object: money

Primitive obsession deserves its own example, because its most common form is also a real bug. Binary floating point can't represent most decimal fractions exactly.

**EXAMPLE** a Python session

```

>>> 0.1 + 0.2
0.30000000000000004
>>> sum([0.1] * 10)
0.9999999999999999
>>> round(2.675, 2)           # you probably expected 2.68
2.67

```

Represent money as integer minor units or as `Decimal`, and wrap it in a type so the rules live in one place.

**AFTER** money.py

```

from dataclasses import dataclass

```

```

from decimal import Decimal, ROUND_HALF_EVEN

CENT = Decimal("0.01")

@dataclass(frozen=True)
class Money:
    amount: Decimal
    currency: str

    @classmethod
    def of(cls, amount: str, currency: str) -> "Money":
        return cls(Decimal(amount).quantize(CENT, ROUND_HALF_EVEN), currency)

    @classmethod
    def usd(cls, amount: str) -> "Money":
        return cls.of(amount, "USD")

    @property
    def cents(self) -> int:
        return int(self.amount * 100)

    def __add__(self, other: "Money") -> "Money":
        if other.currency != self.currency:
            raise CurrencyMismatch(self.currency, other.currency)
        return Money(self.amount + other.amount, self.currency)

    def times(self, factor) -> "Money":
        exact = self.amount * Decimal(str(factor))
        return Money(exact.quantize(CENT, ROUND_HALF_EVEN), self.currency)

```

Adding dollars to euros is now an error instead of a silent nonsense number. Rounding happens in one place, by one rule. Every signature that takes `Money` says what it means. And the factory takes a string on purpose, because `Decimal(0.1)` would faithfully preserve the float's error.

## Habits that keep it safe

- **Let the editor do it.** Rename, extract, inline and move are automated in every serious editor, and automated refactorings preserve behaviour by construction.
- **Commit after every green step.** Small commits make it trivial to undo the step that went wrong.
- **Keep refactoring commits apart from behaviour changes.** A reviewer can verify a pure refactoring quickly when they know nothing is meant to change.
- **Refactor in the main line.** A refactoring branch that lives for weeks conflicts with everything. Small pieces, merged alongside feature work, actually land.

---

### COMMON TRAPS

**Refactoring without tests.** Without a net, a refactoring is a rewrite with extra confidence. Get tests first; chapter 12 shows how when there are none.

---

**Big-bang restructures.** A two-week refactoring branch usually dies. Small steps merge.

---

**"Refactoring" as a euphemism.** If behaviour changes, call it a change. Stakeholders and reviewers both deserve an accurate word.

---

**Abstracting too early.** Two similar pieces of code aren't yet a pattern. Sandi Metz's advice: duplication is far cheaper than the wrong abstraction.

---

## EXERCISES

### EXERCISE 11.1 Name the smell

Name the main smell in each snippet, and the refactoring you'd reach for.

```
# A
def ship(street, city, postcode, country, name, phone): ...
def quote(street, city, postcode, country, weight_kg): ...

# B
def loyalty_tier(customer):
    spend = customer.account.ledger.totals.lifetime_spend
    ...

# C
if user.role == 3: ...           # in fourteen different files

# D
class Order:
    def shipping_label(self):
        return (f"{self.customer.name}\n{self.customer.address.street}\n"
                f"{self.customer.address.city} {self.customer.address.postcode}")
```

---

### SOLUTION

- **A: data clumps**, plus long parameter lists. The four address fields always travel together. Introduce an `Address` value object.
- **B: message chain**. The function knows the internal structure of four objects. Hide the delegate: `customer.lifetime_spend()`.
- **C: primitive obsession and repeated switches**. `3` means something. Replace it with an enum, `Role.ADMIN`, and if the checks are all "may this user do X?", gather them into one permission function.
- **D: feature envy**. `shipping_label` uses the customer's and address's data, not the order's. Move it to where the data lives, such as `address.label_for(name)`.

## EXERCISE 11.2 Split the phase

This function parses a CSV and applies business rules in the same loop. Split it into a parsing phase that produces typed records, and an applying phase that works only with those records. Behaviour must not change.

```
def import_prices(path):
    updated = 0
    with open(path) as f:
        for row in csv.reader(f):
            sku, price = row[0].strip().upper(), row[1]
            if not price.replace(".", "", 1).isdigit():
                continue
            if float(price) <= 0:
                continue
            catalog.set_price(sku, float(price))
            updated += 1
    return updated
```

## SOLUTION

```
@dataclass(frozen=True)
class PriceUpdate:
    sku: str
    price: float

def parse_price_rows(lines) -> tuple[list[PriceUpdate], list[list[str]]]:
    updates, rejects = [], []
    for row in csv.reader(lines):
        sku, raw_price = row[0].strip().upper(), row[1]
        if not raw_price.replace(".", "", 1).isdigit() or float(raw_price) <= 0:
            rejects.append(row)
            continue
        updates.append(PriceUpdate(sku, float(raw_price)))
    return updates, rejects

def apply_price_updates(catalog, updates) -> int:
    for update in updates:
        catalog.set_price(update.sku, update.price)
    return len(updates)

def import_prices(path):
    with open(path) as f:
        updates, _rejects = parse_price_rows(f)
    return apply_price_updates(catalog, updates)
```

Behaviour is identical: the same rows are accepted and the same count is returned. But the parser now accepts any iterable of lines, so it can be tested with a list of strings and no file, and it hands back the rejected

rows. Reporting those rejects, and replacing the float with `Money`, are the natural next steps. Each is a behaviour change, so each gets its own commit and its own tests. One hat at a time.

### EXERCISE 11.3 Build a value object

Write an `EmailAddress` type that rejects strings without exactly one `@`, rejects an empty local part or domain, and treats `Ana@Example.COM` and `Ana@example.com` as equal.

#### SOLUTION

```
@dataclass(frozen=True)
class EmailAddress:
    local: str
    domain: str

    @classmethod
    def parse(cls, raw: str) -> "EmailAddress":
        text = raw.strip()
        if text.count("@") != 1:
            raise InvalidEmail(raw)
        local, domain = text.split("@")
        if not local or not domain:
            raise InvalidEmail(raw)
        return cls(local, domain.lower())

    def __str__(self) -> str:
        return f"{self.local}@{self.domain}"
```

The local part keeps its case, because the email standards allow a mail server to treat it as case-sensitive, while domains are always case-insensitive. That's exactly the kind of rule you want written once, rather than as forty scattered `.lower()` calls. Chapter 16 returns to this pattern under its better-known name: parse, don't validate.

#### CHECKLIST

- ☐ Were the tests green before you started, and after every step?
- ☐ Is each refactoring in a commit that changes no behaviour?
- ☐ Did you refactor towards the change you need, not a hypothetical one?
- ☐ Is money a type, rather than a float, everywhere you touched it?

#### GO DEEPER

BOOK      *Refactoring*, 2nd edition Martin Fowler: the catalogue, with worked examples

Refactoring, 2nd Edition Martin Fowler · the catalogue, with worked examples

**FREE** The refactoring catalogue every refactoring in the book, summarised online

**ESSAY** The Wrong Abstraction Sandi Metz, 2016

**BOOK** Tidy First? Kent Beck, 2023 · tiny, practical, on when to tidy and why

# Legacy code

3 exercises

*Michael Feathers defines legacy code as code without tests. The craft is getting a safety net around it before you change it, one small seam at a time, without stopping the business while you do.*

## Why legacy code is hard

The problem isn't age or style. It's that you can't change the code safely, because you can't tell whether a change broke something. And you can't easily add tests, because the code wasn't written to be tested: it reads the clock, calls the database in the middle of a calculation, and constructs its own collaborators. Feathers calls this the legacy code dilemma. To change code safely you need tests, and to add tests you often need to change code.

## The legacy change algorithm

- 01 Identify change points.** Where exactly does the change need to go?
- 02 Find test points.** Where can you observe the effect of that code?
- 03 Break dependencies.** Make just enough careful, minimal edits to get the code into a test harness.
- 04 Write tests.** Pin down the current behaviour around the change points.
- 05 Make the change, and refactor.** Now, with a net.

Step 3 is the dangerous one, because you're changing code without tests. Keep those edits tiny and mechanical, lean on your editor's automated refactorings, and review them twice.

## Characterisation tests

A characterisation test records what the code *actually* does, not what it should do. The recipe: call the code with some input, assert something you know is wrong, run the test, and let the failure tell you the real output. Paste that in as the expected value. Repeat with inputs that exercise different branches.

**EXAMPLE** nobody fully understands it, and finance depends on it

```
def compute_commission(sale, rep, month):  
    ... # 140 lines of nested conditions
```

```
# Step 1: assert something deliberately wrong.
def test_characterise_commission_for_a_standard_rep():
    sale = Sale(amount=12_000, product="enterprise", region="EMEA")
    rep = Rep(level=2, tenure_months=30)
    assert compute_commission(sale, rep, month=11) == 0

# The run fails with: assert 1380.0 == 0
# Step 2: paste the real value in. The test now pins current behaviour.
def test_characterise_commission_for_a_standard_rep():
    sale = Sale(amount=12_000, product="enterprise", region="EMEA")
    rep = Rep(level=2, tenure_months=30)
    assert compute_commission(sale, rep, month=11) == 1380.0
```

Some characterised values will look wrong. Don't fix them now. Note them, keep the test as it is, and fix the behaviour later as a separate, deliberate change, agreed with whoever owns it. Someone may be depending on the bug. When the output is large, such as a generated report, combine this with the snapshot tests of chapter 10. Approval-testing tools are built for exactly this job.

## Getting code into a test harness

Most legacy code resists tests because of one dependency it reaches for directly. Feathers catalogues two dozen dependency-breaking techniques. Three of them cover most cases in Python.

**Parameterise the dependency.** Add a parameter whose default preserves the old behaviour. Existing callers don't change, and tests can pass something else.

**AFTER** callers unchanged; tests can pass both arguments

```
def overdue_accounts(today=None, accounts=None):
    today = today or date.today()
    accounts = db.accounts() if accounts is None else accounts
    return [a for a in accounts if a.due < today and a.balance > 0]
```

**Extract and override.** Move the awkward call into its own method, then override that method in a test subclass. It's a crude seam, but it's safe to create and needs no changes to any caller.

### EXAMPLE

```
class InvoiceRun:
    def execute(self):
        for customer in self.customers_to_bill():
            self.bill(customer)

    def customers_to_bill(self):
        # extracted; was an inline query
        return db.query("SELECT * FROM customers WHERE billing_day = %s",
                        date.today().day)

class TestableInvoiceRun(InvoiceRun):
```

```
def __init__(self, customers):
    self._customers = customers

def customers_to_bill(self):
    return self._customers
```

**Wrap the global.** Code that reaches for a module-level singleton, such as `settings.FEATURE_X` or `cache.client`, can first be pointed at a small function. That function can then be replaced in tests with `monkeypatch`. It's a weaker seam than injection, but on the first day it's often the only safe one.

## Adding a feature without making it worse

When the code around a change is too tangled to test yet, don't add your new logic to the tangle. Two techniques keep new code clean and tested from its first line.

**Sprout.** Write the new behaviour as a new, fully tested function, and call it from one line inside the old code.

### EXAMPLE

```
def process_order(order):
    # ... 200 lines of untested legacy code ...
    order.gift_wrap_fee = gift_wrap_fee(order.items)    # sprouted: one new line
    # ... more legacy code ...

def gift_wrap_fee(items):                               # new, pure, fully tested
    wrapped = [i for i in items if i.gift_wrap]
    return Money.usd("3.50").times(len(wrapped))
```

**Wrap.** When new behaviour must happen before or after every call to old code, rename the old function and put a new one, with the old name, in front of it.

### EXAMPLE

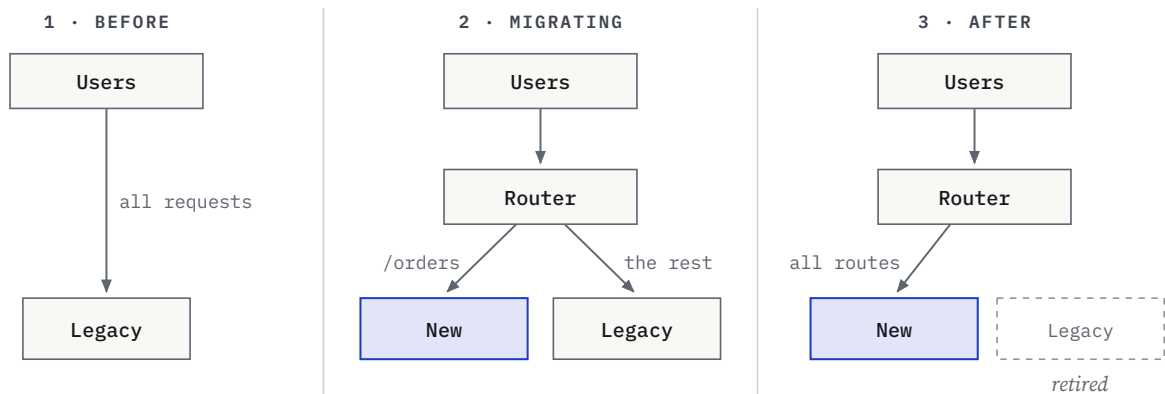
```
def charge_card(card, amount):                          # new function, old name
    if fraud_score(card, amount) > FRAUD_THRESHOLD:
        raise SuspectedFraud(card.last4)
    return _charge_card_legacy(card, amount)

def _charge_card_legacy(card, amount):                  # the untouched original, renamed
    ...
```

## Scratch refactoring

To understand a baffling piece of code, refactor it aggressively with no intention of keeping the result. Rename everything, extract functions, delete whatever looks dead. You'll learn how it works. Then throw the changes away with `git restore .` and do it properly, with tests.

## Replacing a whole system: the strangler fig



*Traffic moves from the old system to the new one a route at a time. At every stage the whole system works, and any step can be reversed by changing the router.*

In 2004 Martin Fowler named this pattern after the strangler figs he saw in Queensland, which grow around a host tree until they can stand on their own. Put a routing layer in front of the old system. Build new functionality, or rebuild old functionality, as a new system behind the same router. Move routes across one at a time. Eventually the old system handles nothing and can be switched off.

Inside a single codebase, the same move is called **branch by abstraction**. Introduce an interface in front of the old component, make every caller use it, build the new implementation behind it, switch over, and delete the old one. Both approaches let you migrate in production, in small, reversible steps.

## Large changes: the Mikado method

For a change too big to make in one step, try it naively and note what breaks. Each breakage is a prerequisite. Revert, then attempt each prerequisite the same way, building a graph of what depends on what. When you reach changes that succeed on their own, make them, commit, and work back up the graph. Ola Ellnestam and Daniel Brolund named it after the pick-up-sticks game, where you remove sticks from the top of the pile without disturbing the rest.

## The rewrite temptation

Faced with a large legacy system, most engineers want to start again. Joel Spolsky's 2000 essay *Things You Should Never Do* made the case against, using Netscape's decision to rewrite its browser from scratch: years passed with nothing shipped while competitors moved on. Old code is full of fixes for bugs nobody remembers, and a rewrite rediscovers them one production incident at a time. Rewrites do occasionally succeed, usually when the scope is small or the platform underneath is truly dead. The default should be incremental replacement.

---

## COMMON TRAPS

**Fixing bugs while characterising.** You will find odd behaviour. Record it; don't fix it mid-harness. Change one thing at a time.

---

**Large untested edits.** Every dependency-breaking edit happens without tests, by definition. Keep them tiny.

---

**Strangling without a finish line.** A migration that stalls at sixty percent leaves you maintaining two systems forever. Plan the last route, and the deletion, from the start.

---

**Removing fences.** Code that looks pointless in a legacy system is often a fix for something you haven't met yet. Find out why before you delete it.

---

## EXERCISES

### EXERCISE 12.1 Characterise it

Write characterisation tests that pin this function's current behaviour, including anything that looks like a bug. Don't fix anything.

```
def shipping_band(weight_kg, country):
    if country == "US":
        if weight_kg < 1:
            return "A"
        if weight_kg < 5:
            return "B"
        return "C"
    if country in ("CA", "MX"):
        if weight_kg <= 2:
            return "B"
        return "C"
    if weight_kg > 10:
        return "D"
    return "C"
```

---

### SOLUTION

```

@pytest.mark.parametrize("weight, country, band", [
    (0.5, "US", "A"),
    (1.0, "US", "B"),      # exactly 1 kg is band B: the US branch uses <
    (4.9, "US", "B"),
    (5.0, "US", "C"),
    (2.0, "CA", "B"),      # but this branch uses <=, unlike the US one
    (2.1, "MX", "C"),
    (10.0, "FR", "C"),     # exactly 10 kg abroad is still C
    (10.1, "FR", "D"),
    (0.1, "us", "C"),      # SUSPECT: lower-case code falls through to international
    (-1, "US", "A"),       # SUSPECT: negative weight accepted without complaint
])
def test_characterise_shipping_band(weight, country, band):
    assert shipping_band(weight, country) == band

```

The last two rows are probably bugs, and a characterisation suite records them anyway, clearly marked, so that fixing them later is a visible decision rather than an accident. Note too the inconsistent boundaries between the US and the other North American branches. That's a question for whoever owns shipping prices, not something to "fix" quietly.

## EXERCISE 12.2 Sprout a rule

Add a rule: orders containing any item marked `hazardous` can't ship by air. The legacy function is untested and three hundred lines long. Show the one line you'd add, and the sprouted function with its tests.

```

def choose_shipping_method(order):
    # ... 300 lines ...
    method = cheapest_available(order, candidates)
    # ... more ...
    return method

```

## SOLUTION

```

def choose_shipping_method(order):
    # ... 300 lines ...
    candidates = without_air_if_hazardous(candidates, order.items)  # sprouted
    method = cheapest_available(order, candidates)
    # ... more ...
    return method

def without_air_if_hazardous(methods, items):
    if any(item.hazardous for item in items):
        return [m for m in methods if m.mode != "air"]
    return methods

def test_hazardous_items_rule_out_air_methods():
    methods = [Method("ground", mode="road"), Method("express", mode="air")]
    items = [Item("battery pack", hazardous=True)]

```

```
assert without_air_if_hazardous(methods, items) == [methods[0]]
```

```
def test_ordinary_orders_keep_every_method():  
    methods = [Method("ground", mode="road"), Method("express", mode="air")]  
    assert without_air_if_hazardous(methods, [Item("book")]) == methods
```

Now check the edge the sprout creates. If every candidate was an air method, the list is suddenly empty. What does `cheapest_available` do with an empty list? That's the one question about the legacy code you must answer before shipping.

### EXERCISE 12.3 Plan a strangler

A monolith serves `/catalog`, `/cart`, `/checkout`, `/account` and `/admin`, and you're moving to a new service. In a few sentences each: which route moves first, and why? What must be true before `/checkout` moves? How will you know you've finished?

#### SOLUTION

**First:** something read-heavy, low-risk and well-bounded, often `/catalog`. It has no writes to reconcile, and a mistake there is visible but cheap. Many teams also pick the area with the most planned new work, so the migration pays for itself.

**Before checkout:** the new service must read and write orders consistently with the old system, since both will be live at once. You need a rollback path at the router, and monitoring that compares outcomes before and after, such as conversion rate and payment failures, ideally through a gradual percentage rollout.

**Finished** means the router has sent no traffic to the monolith for a sustained period, its data is migrated or archived, and the old code, infrastructure and routing rules are deleted. Deletion is part of the plan, not an afterthought.

#### CHECKLIST

- ☐ Before changing untested code, did you pin its behaviour with characterisation tests?
- ☐ Were your dependency-breaking edits small, mechanical and reviewed?
- ☐ Is new behaviour in sprouted or wrapped code, with its own tests?

- ☐ For a migration: is there a router you can flip back, and a plan to delete the old path?

#### GO DEEPER

|       |                                                                                              |
|-------|----------------------------------------------------------------------------------------------|
| BOOK  | Working Effectively with Legacy Code Michael Feathers · the source of nearly everything here |
| ESSAY | Strangler Fig Application Martin Fowler                                                      |
| ESSAY | Things You Should Never Do, Part I Joel Spolsky, 2000                                        |
| BOOK  | The Mikado Method Ola Ellnestam & Daniel Brolund                                             |

## PART V

# Working with other people

Code is a team sport played over years. These chapters cover the two collaboration loops every engineer runs daily: reviewing each other's changes, and integrating and releasing them.

# Code review

3 exercises

*Review's biggest payoff is shared understanding, not bug-catching. Review to keep the codebase healthy and to teach, keep changes small enough to review properly, and remember that there's a person on the other side of every comment.*

## What review actually does

Ask engineers why they review code and most will say: to find bugs. In 2013 Alberto Bacchelli and Christian Bird studied review at Microsoft, observing and interviewing developers and classifying hundreds of real review comments. Finding defects was the most common stated motivation, yet reviews turned up fewer defects than people expected. Their main outcomes were knowledge transfer, team awareness, and alternative solutions to problems.

That changes how you should review. A review that only hunts for bugs misses its most valuable output: two people who now understand this code instead of one, and a reviewer who learned a technique or a corner of the system they didn't know.

## The standard

Google's published engineering practices set the bar clearly: approve a change once it definitely improves the overall health of the codebase, even if it isn't perfect. There's no perfect code, only better code. A reviewer who blocks for polish slows the whole team down. One who approves anything lets the codebase decay a change at a time. The balance is to hold the line on design and correctness, and to let personal preferences go.

## What to look for, most important first

- 01 Design.** Do the pieces make sense together? Does this belong here, in a library, or nowhere?
- 02 Functionality.** Does it do what the author intended, and is that good for users? Think about edge cases and concurrency, which tests often miss.
- 03 Complexity.** Could it be simpler? Is anything built for a future that may never arrive?
- 04 Tests.** Are there tests, do they test behaviour, and would they fail if the code broke?
- 05 Naming.** Do the names say what things are?
- 06 Comments.** Do they explain why, and are they needed at all?
- 07 Style and consistency.** Mostly the formatter's job. Anything a tool didn't catch is a nit.

- 08 **Documentation.** Did the user-facing or developer docs change along with the behaviour?
- 09 **Context.** Look at the whole file, not just the diff. Is this change making the file worse?
- 10 **Good things.** Say so when you see them. Review is also how people learn what good looks like.

This list is adapted from Google's. Its unwritten rule matters as much as any item: read every line you were asked to review, and if you can't understand something, say so. Code the reviewer can't follow is code the next maintainer can't follow either.

## Size is the variable that matters most

A study of code review at Cisco, covering 2,500 reviews, found that reviewers' ability to find defects fell sharply beyond about 200 to 400 lines at a time, and after 60 to 90 minutes of continuous reviewing. The 2025 DORA report found the other half of the problem: AI assistance is making pull requests larger, working against the small-batch discipline its own research links to better delivery.

Big changes don't get reviewed. They get skimmed and approved. If you want real review, send small changes. Split refactoring from behaviour. Land the database migration separately. Put new code behind a flag and merge it in slices.

## As the author

- **Review your own diff first**, in the review tool, the way the reviewer will see it. You'll catch the debug print, the stray file and the confusing name before anyone else has to.
- **Keep it small and single-purpose.** One logical change per pull request.
- **Explain the why.** The diff shows what changed. The description must say why, how you tested it, and where reviewers should look hardest.
- **Separate refactors from behaviour changes.** Reviewers can skim a pure rename. They must study a logic change. Don't make them do both at once.
- **Respond to every comment.** Change it, or explain why not. "Done" is a perfectly good reply.
- **Your code is not you.** A comment about the code isn't a verdict on your competence.

**EXAMPLE** a pull request description

### ## Why

Customers on annual plans were billed monthly after changing plan (BILL-812).

### ## What changed

- `PlanChange.apply` keeps the billing interval of the target plan
- A migration backfills `billing\_interval` for the 212 affected subscriptions

```

### How it was tested
- New unit tests for interval handling in `test_plan_change.py`
- Ran the migration on a copy of production and spot-checked 20 accounts

### Look closely at
The backfill query. It's the only irreversible part of this change.

```

## As the reviewer

- **Respond quickly.** Slow review is a major cause of large pull requests: if review takes three days, people batch their work to avoid waiting. Google's guidance is to respond within one business day at most.
- **Big picture first.** If the design is wrong, line comments are wasted effort. Say so early, and ideally in conversation.
- **Ask, don't command.** "What happens if the list is empty?" invites the author to find the bug. "This crashes on an empty list" is fine when you're sure. "Fix this" teaches nothing.
- **Label severity.** Make clear what blocks approval and what's optional. The Conventional Comments convention gives you labels such as `issue`, `suggestion`, `question`, `nit` and `praise`.
- **Explain why, and offer a way forward.** A reason turns a preference into a lesson.
- **Automate the arguments.** Formatting, import order and lint rules belong to tools. Human attention is too expensive to spend on them.

| INSTEAD OF                      | TRY                                                                                                             |
|---------------------------------|-----------------------------------------------------------------------------------------------------------------|
| "This is wrong."                | "issue: this double-charges if <code>charge()</code> times out and we retry. Could we send an idempotency key?" |
| "Why would you do it this way?" | "question: what made you choose a thread here over the existing job queue? I may be missing a constraint."      |
| "Rename this."                  | "nit: <code>data</code> to <code>pending_refunds</code> ? It took me a moment to see what it held."             |
| "Use a dict."                   | "suggestion: a dict keyed by SKU would make this lookup constant-time. Not blocking at today's sizes."          |
| Silence, when it's good         | "praise: the characterisation tests made this easy to review. Thank you."                                       |

## Disagreeing well

Most review disagreements are preference dressed up as principle. Before pushing back, ask whether you have data, a documented standard, or a concrete failure scenario. If you do, state it. If you only have a preference, label it as one, or let it go.

When a thread reaches its third round, stop typing and talk. A five-minute call resolves what twenty comments won't. If you still disagree, agree on who decides, usually the code owner or a tech lead, rather than letting the change stall, and record the outcome in the thread so the next reader sees why.

## Reviewing code an assistant wrote

The standard doesn't change; the failure modes do. Generated code is fluent, which makes its errors harder to spot: calls to APIs that don't exist, edge-case handling that looks right and isn't, security checks that are almost correct, and tests that assert whatever the implementation happens to do. Two rules help. The author must understand every line they submit, whoever wrote the first draft. And tests generated alongside code prove little on their own, so check that they encode the *intended* behaviour, not merely the actual behaviour.

---

### COMMON TRAPS

**Rubber-stamping.** "LGTM" after two minutes on 800 lines isn't a review. If you can't review properly, say so and suggest someone who can.

---

**Nitpick avalanches.** Twenty style comments bury the one design concern. Let tooling handle style, and prioritise.

---

**Gatekeeping.** Using review to enforce personal taste, or to show off, teaches people to avoid sending you reviews.

---

**Last-minute redesigns.** Proposing a different architecture the day before release, having not been involved earlier. Raise design concerns in design discussions, not at the final gate.

---

### EXERCISES

#### EXERCISE 13.1 Review this diff

Write review comments, each labelled with a severity, for this change.  
It adds an endpoint for applying discount codes.

```
+@app.post("/orders/<order_id>/discount")
+def apply_discount(order_id):
+    code = request.json["code"]
+    order = db.execute(f"SELECT * FROM orders WHERE id = {order_id}").one()
+    discount = db.execute(
```

```
+     "SELECT percent FROM discount_codes WHERE code = %s", (code,)
+ ).one()
+ order.total = order.total - order.total * discount.percent / 100
+ db.save(order)
+ return {"total": order.total}
```

#### SOLUTION

- **issue:** SQL injection. `order_id` is interpolated straight into the query. Use a parameter, as the second query already does.
- **issue:** no authorisation check. Any user can apply a discount to anyone's order.
- **issue:** applying the same code twice discounts twice. Should a code apply at most once per order? That probably needs a record of applied codes.
- **issue:** an unknown code makes `.one()` raise, so the client gets a 500. A 404 or 422 with a clear message would be better.
- **question:** can a discount be applied to an order that's already paid or shipped?
- **issue:** money as a float, with percentage arithmetic and no defined rounding. This should go through the `Money` type.
- **suggestion:** there are no tests. At minimum: a valid code, an unknown code, a repeated code, and someone else's order.

Notice that none of these is about style. With a formatter in place, review attention goes where it's needed.

#### EXERCISE 13.2 Rewrite the comments

Rewrite each so it's specific, kind and actionable. Invent plausible details.

1. "This is really inefficient."
2. "Did you even run this?"
3. "I wouldn't do it like this."
4. "Wrong."

#### SOLUTION

1. "suggestion: this scans every order once per customer, which is quadratic. At our volumes that's billions of comparisons a night. Grouping orders by customer id first would make it linear."

2. "question: I get a `NameError` on line 42 when I run this locally, because `total_cents` isn't defined in that scope. Is a commit missing?"
3. "nit, not blocking: I'd probably have used the existing `retry` decorator instead of a hand-written loop, mostly for consistency. Happy either way."
4. "issue: this returns the pre-tax total, but the invoice needs the amount after tax. `Invoice.total_with_tax` does that."

### EXERCISE 13.3 Split the pull request

A 1,900-line pull request renames `Customer` to `Account` across the codebase, adds a `tax_id` column with a migration, adds VAT calculation for EU accounts, and fixes an unrelated timezone bug in invoice dates. Propose a sequence of smaller pull requests, in merge order.

#### SOLUTION

1. The timezone fix, alone, with its regression test. It's unrelated, so it shouldn't wait for anything.
2. The rename, as a pure mechanical refactoring done by the editor. Large, but quick to review because nothing is supposed to change. Merge it fast to avoid conflicts.
3. A migration adding a nullable `tax_id` column, with no code using it yet.
4. Code that reads and writes `tax_id`, with its validation.
5. VAT calculation behind a feature flag, with tests.
6. Enable the flag for EU accounts. Later, remove the flag in a small cleanup.

Each step can be reviewed in one sitting and deployed on its own, and a problem in step 5 doesn't hold up the timezone fix.

#### CHECKLIST

- ☐ As author: is the change small, single-purpose, self-reviewed and explained?
- ☐ As reviewer: did you look at design before details, and label severity?
- ☐ Did you say something about what's good?

☐ Did any thread go past three rounds without a conversation?

GO DEEPER

**FREE** Google Engineering Practices: code review guides for both authors and reviewers

---

**PAPER** Expectations, Outcomes, and Challenges of Modern Code Review Bacchelli & Bird, 2013

---

**FREE** Conventional Comments labels that make severity explicit

---

**REPORT** State of AI-assisted Software Development DORA, 2025

# Version control and releasing

3 exercises

*Integrate small changes continuously, make every commit tell the reader why, and separate deploying code from releasing features.*

## A commit is a message to the future

Months from now, someone will run `git blame` on a confusing line and read your commit message. It's the only record of why that line is the way it is. The diff already says what changed. The message must say why.

- **One logical change per commit.** It builds, it passes the tests, and it can be reverted on its own.
- **A short summary line**, around fifty characters, in the imperative mood: "Fix double charge on payment retry", as if completing the sentence "this commit will...".
- **A body that explains why**, wrapped at about seventy-two characters. What was wrong, why this approach, and what the alternatives were.

**BEFORE** `git log --oneline`

```
fixed stuff
wip
changes per review
update payment.py
```

**AFTER** `one message, in full`

Send idempotency key on payment retries

The payment provider times out on a small fraction of charges, and our retry loop resubmitted them as new charges, so some customers were billed twice (BILL-771).

Each checkout now generates one idempotency key and reuses it for every retry, so the provider deduplicates. The key is stored on the order, so a crash between attempts can't produce a second one.

## Conventional commits

A lightweight convention that makes history machine-readable: prefix the summary with a type. `fix:` corresponds to a patch release, `feat:` to a minor release, and a `!` after the type, or a `BREAKING CHANGE:` footer, to a major one. Release tooling can then generate

changelogs and version numbers for you.

#### EXAMPLE

```
feat(checkout): accept gift cards as partial payment
fix(invoices): use the customer's timezone for due dates
refactor(orders): split pricing from rendering
feat(api)!: remove the deprecated /v1/orders endpoint
```

## Curate history before you share it

Your local history can be messy while you work. Before merging, make it tell a clean story.

Git can fold small fixes into the commits they fix:

#### EXAMPLE

```
git commit --fixup=abc1234      # record a fix for an earlier commit on this branch
git rebase -i --autosquash main  # fold every fixup into the commit it targets
```

The rule that keeps this safe: never rewrite history that other people have already pulled.

Rebase your own branch; never rebase `main`.

## Branching strategies

| STRATEGY           | HOW IT WORKS                                                                                         | SUITS                                                            | COST                                           |
|--------------------|------------------------------------------------------------------------------------------------------|------------------------------------------------------------------|------------------------------------------------|
| <b>Git Flow</b>    | Long-lived <code>develop</code> and <code>main</code> , plus feature, release and hotfix branches    | Software shipped as numbered releases: installed apps, libraries | Heavy process; long-lived branches diverge     |
| <b>GitHub Flow</b> | Short-lived branches off <code>main</code> , merged by pull request, deployed from <code>main</code> | Most web applications and services                               | Needs good CI and fast review                  |
| <b>Trunk-based</b> | Everyone integrates into <code>main</code> at least daily; any branch lives a day or two             | Teams practising continuous delivery                             | Needs feature flags and strong automated tests |

Vincent Driessen, who described Git Flow in 2010, later added a note recommending simpler workflows for continuously delivered software. DORA's research has repeatedly associated trunk-based development with higher delivery performance. The mechanism is simple: the longer a branch lives, the further it diverges, and the more painful and risky the merge. Integrating daily keeps every merge small.

## Deploying is not releasing

Trunk-based development raises an obvious question: how do you merge unfinished features every day? You merge them switched off. A feature flag lets code reach production dormant, which separates **deploying**, putting code on servers, from **releasing**, exposing behaviour to users. Pete Hodgson's taxonomy names four kinds of flag, and the distinction matters because their lifespans differ.

| TYPE              | PURPOSE                                         | LIFESPAN                                | EXAMPLE                              |
|-------------------|-------------------------------------------------|-----------------------------------------|--------------------------------------|
| <b>Release</b>    | Hide unfinished work                            | Days to weeks. Delete once launched     | <code>new_checkout_flow</code>       |
| <b>Experiment</b> | A/B test variants                               | Until the result is significant         | <code>pricing_page_variant_b</code>  |
| <b>Ops</b>        | Kill switch, or graceful degradation under load | Long-lived, by design                   | <code>disable_recommendations</code> |
| <b>Permission</b> | Features for some users only                    | Long-lived; often becomes product logic | <code>beta_program</code>            |

**EXAMPLE** a release flag, with its removal already ticketed

```
if flags.enabled("new_checkout_flow", user=current_user):
    return new_checkout(cart)      # TODO(CHK-201): delete flag after full rollout
return legacy_checkout(cart)
```

Flags are debt, because each one doubles the paths through the code it guards. On 1 August 2012, Knight Capital deployed new trading code that reused an old flag. One of its eight servers hadn't received the new code, and on that server the flag reactivated logic that had been dead for years. In about forty-five minutes the firm lost roughly \$440 million. Remove release flags as part of finishing a feature, and never reuse a flag's name.

## Versioning what others depend on

If other people depend on your code, as a library, an API or an event schema, version it. Semantic Versioning's rules are short: in `MAJOR.MINOR.PATCH`, increment `PATCH` for backward-compatible bug fixes, `MINOR` for backward-compatible features, and `MAJOR` for anything that breaks existing users. Versions below 1.0.0 promise nothing.

The hard part is knowing what counts as breaking. Hyrum Wright, a Google engineer, observed that with enough users, every observable behaviour of a system will be depended on by somebody, whatever the documentation promises: field order, the wording of error messages, timing, exactly how rounding works. So "backward compatible" really means

compatible with what people actually do, and you often only find out by changing it. Deprecate before removing: announce, warn in logs or responses, provide a migration path and a date, and then remove in the next major version.

## Changelogs

A changelog is for people, not machines. It lists what changed that a user would care about, newest first, grouped under Added, Changed, Deprecated, Removed, Fixed and Security, as in the widely used Keep a Changelog convention. A changelog generated from conventional commits is a good start, and deserves a human edit before release.

---

### COMMON TRAPS

**"Merge branch main into feature-x"**, fifteen times over. Rebase your branch onto `main` instead, or squash on merge.

**Force-pushing shared branches.** Use `--force-with-lease` if you must, and never on `main`.

**Flag graveyards.** Give every release flag a removal ticket on the day it's created.

**Breaking changes in minor versions.** It's the quickest way to teach users to pin every dependency forever.

### EXERCISES

#### EXERCISE 14.1 Rewrite the messages

Write a proper summary line and a short body for each. Invent plausible context.

1. fix bug
2. updated tests
3. refactor

---

#### SOLUTION

Reject coupons that expired while the cart was open

Coupons were checked for expiry only when added to the cart, so a coupon that expired while the cart sat open was still honoured at payment. Checkout now re-validates every coupon against the current time.

Cover leap-day renewals in subscription tests

Renewal dates were untested for subscriptions that start on 29 February.  
Adds cases for renewals in leap and non-leap years. No production code changes.

Extract tax calculation from InvoiceRenderer

Tax logic lived inside the PDF renderer, so the planned CSV export would have had to duplicate it. Moved unchanged to tax.py; output is identical, as checked by the existing snapshot tests.

### EXERCISE 14.2 Pick the version

Your library is at 2.4.1. What's the next version after each change, taken separately?

- A. Fixes a crash when parsing empty input.
- B. Adds an optional timeout parameter, defaulting to 30 seconds.
- C. Renames `fetch()` to `get()`, with no alias kept.
- D. Rewords an error message.
- E. Drops support for Python 3.9.
- F. Makes a function 40% faster, with identical results.

#### SOLUTION

- **A: 2.4.2.** A backward-compatible fix.
- **B: 2.5.0,** a backward-compatible feature. But check the default: if calls never timed out before, a 30-second default is a behaviour change that could break slow callers. Defaulting to no timeout is the safe choice.
- **C: 3.0.0.** It breaks every caller. Better: add `get()` and keep `fetch()` as a deprecated alias in 2.5.0, then remove it in 3.0.0.
- **D: formally 2.4.2.** But by Hyrum's law, somebody is matching on that string. Call it out in the changelog.
- **E: usually 3.0.0.** Some projects treat dropping an end-of-life Python version as minor. Write your policy down and apply it consistently.
- **F: 2.4.2 or 2.5.0,** depending on your policy for performance work. It isn't breaking, unless someone depended on it being slow, which Hyrum would tell you isn't impossible.

### EXERCISE 14.3 Classify the flags

For each flag, name its type and when it should be removed.

- |                          |                                              |
|--------------------------|----------------------------------------------|
| 1. show_new_onboarding   | the onboarding redesign, half built          |
| 2. checkout_button_color | testing green against blue                   |
| 3. disable_pdf_export    | the PDF service falls over under load        |
| 4. enterprise_sso        | only enterprise customers get single sign-on |

#### SOLUTION

1. **Release flag.** Remove it, and the old onboarding code, as soon as the redesign is fully rolled out.
2. **Experiment flag.** Remove it when the experiment reaches a result, keeping the winning variant.
3. **Ops flag.** Long-lived by design. Test that switching it on actually works, because an untested kill switch fails exactly when you need it.
4. **Permission flag.** Long-lived, and arguably not a flag at all. It's a product entitlement, and belongs in the plans and billing model.

#### CHECKLIST

- ☐ Does each commit build, pass the tests, and do one thing?
- ☐ Does each message say why?
- ☐ Did your branch live less than a couple of days?
- ☐ Is unfinished work behind a flag with a removal ticket?
- ☐ For anything others depend on, did you check whether the change breaks them, including undocumented behaviour?

#### GO DEEPER

**SPEC**      [Semantic Versioning 2.0.0](#)

**SPEC**      [Conventional Commits 1.0.0](#)

**ESSAY**      [Feature Toggles](#) Pete Hodgson · the four types, in depth

**FREE**      [Trunk-based development](#) DORA's summary of the research

**FREE**      [Hyrum's Law](#) the one-sentence law and its origin

## PART VI

# When things go wrong

Every system fails. These three chapters cover finding the cause when it does, deciding how failure is handled before it happens, and the discipline of measuring before you optimise.

# Systematic debugging

3 exercises

*Debugging is the scientific method applied to a program: observe, hypothesise, predict, test, narrow. Changing code at random until the symptom disappears isn't debugging. It's moving the bug somewhere you haven't looked yet.*

## Where bugs live

Every bug lives in the gap between what you believe the code does and what it actually does. So the first move is always to distrust a belief. "That function can't return `None`." "The config is definitely loaded." "This branch is never reached." The belief you're most sure of is usually the one you haven't checked.

## The loop

- 01 **Reproduce it** reliably, ideally as a failing automated test. If you can't reproduce it, you can't know you've fixed it.
- 02 **Read the evidence.** The whole error message, the whole stack trace, the logs around it. The answer is in the error message more often than people expect.
- 03 **Form a hypothesis**, specific and falsifiable: "the discount is applied twice because the handler runs in both the web and the worker process".
- 04 **Predict.** If the hypothesis is true, what else must be true? What would you see if you looked?
- 05 **Test the prediction** with the cheapest experiment available: a log line, a breakpoint, a one-line change.
- 06 **Narrow, and repeat.** Every experiment should cut the search space.
- 07 **Fix the cause**, not the symptom.
- 08 **Keep the reproduction** as a regression test.
- 09 **Look for siblings.** The same mistake is often made in more than one place.

## Binary search, everywhere

When you can't reason your way to the bug, halve the search space. It works along three axes.

**Across history.** If it worked last month, the bug is in a commit since then, and `git bisect` will find which one. Given a script, it runs unattended.

#### EXAMPLE

```
git bisect start
git bisect bad           # the current commit is broken
git bisect good v2.3.0   # this release was fine
git bisect run pytest tests/test_invoices.py -x -q
# git checks out commits, runs the tests, and names the first bad commit
git bisect reset
```

Across a thousand commits, that takes about ten test runs.

**Across input.** A 40,000-line CSV crashes the importer. Delete the second half. Still crashing? Keep halving. In a few minutes you have the one line that matters. This is **delta debugging**, the same shrinking that Hypothesis does automatically.

**Across code.** Add a checkpoint halfway along the suspect path: is the data already wrong at this point? The first place it's wrong is where to look.

## Use a debugger

Print debugging works, and in production, logs may be all you have. Locally, though, a debugger shows the whole state at once, and lets you ask questions you didn't think of in advance.

#### EXAMPLE stop inside a loop

```
def allocate_stock(order):
    for line in order.lines:
        breakpoint()           # opens pdb here, with everything in scope
        reserve(line.sku, line.quantity)
```

When something crashes, inspect the moment it died:

#### EXAMPLE post-mortem debugging

```
python -m pdb -c continue import_prices.py prices.csv
# when the exception is raised, you are left in the frame that raised it
```

Inside `pdb`, `p expr` prints a value, `n` steps over a line, `s` steps into a call, `u` and `d` move up and down the stack, and `w` shows where you are. A conditional breakpoint, such as `b invoices.py:88, total < 0`, stops only when the interesting case happens.

## Check the boring things first

A surprising share of debugging time is spent on a program that isn't the one you think is running.

- **Is the code you're editing the code that's running?** Stale builds, cached bytecode, the wrong virtual environment, a container image that wasn't rebuilt.
- **Is it the environment you think?** Staging or production, which config file, which feature flags.
- **What changed?** Deploys, dependency updates, configuration, data, traffic. Diff the working environment against the broken one.
- **Is the input what you think?** Print it exactly, with `repr()`, so you can see the trailing space and the invisible character.

## Explain it out loud

*The Pragmatic Programmer* popularised rubber-duck debugging: explain the problem, line by line, to a rubber duck. It works because explaining forces you to state every assumption, and one of them is wrong. A colleague works even better, and often doesn't need to say a word.

## The minimal reproduction

Shrink the failing case until removing anything makes the bug disappear. The bug often becomes obvious along the way. If it doesn't, you now have the perfect bug report: something another person can run in thirty seconds. An issue with a minimal reproduction is far more likely to be fixed quickly than one without.

## Symptom or cause?

A null check that stops a crash is often a fix for the symptom. The value shouldn't have been null, and whatever produced it upstream is still wrong. Keep asking why until you reach a cause you can fix at its source. The "five whys" of Toyota's production system is a structured version of this habit. Don't stop at the first answer that makes the error go away.

---

### COMMON TRAPS

**Shotgun debugging.** Changing several things at once until it works. You won't know which change fixed it, and the others may have broken something new.

---

**Fixing without reproducing.** "I think this should fix it" ships a guess.

---

**Stopping at the symptom.** The crash is gone; the corrupt data upstream is not.

---

**Heroic all-nighters.** After an hour stuck, take a break or ask someone. Fresh eyes are the

cheapest debugging tool there is.

**Heisenbugs.** Bugs that vanish when observed, usually from timing or concurrency. Adding a print changes the timing. Use timestamped logging, or tools built for races, rather than breakpoints.

## EXERCISES

### EXERCISE 15.1 Reason it out

Users report that the second cart they create already contains items. Find the bug without running the code.

```
class Cart:
    def __init__(self, items=[]):
        self.items = items

    def add(self, sku):
        self.items.append(sku)

a = Cart()
a.add("SKU-1")
b = Cart()
print(b.items)           # ['SKU-1']
```

### SOLUTION

Python evaluates default arguments once, when the function is defined, not on every call. Every `Cart()` created without an argument shares one list object, so adding to one cart adds to all of them.

```
class Cart:
    def __init__(self, items=None):
        self.items = list(items) if items is not None else []
```

Note the `list(items)` copy as well. Without it, a caller's list would be aliased into the cart, which is the same bug in a different costume. The belief to distrust here was "each cart has its own list".

### EXERCISE 15.2 Bisect it

An invoice total became wrong somewhere in the last 300 commits.

`python scripts/invoice_total.py fixtures/case-17.json` prints the total, and the correct total is `1234.56`. Write the script you'd hand to `git bisect run`, and the commands to start it.

## SOLUTION

```
#!/usr/bin/env bash
# check_total.sh. Exit 0 = good, 1 = bad, 125 = this commit can't be tested.
[ -f scripts/invoice_total.py ] || exit 125
output=$(python scripts/invoice_total.py /tmp/case-17.json) || exit 1
[ "$output" = "1234.56" ]
```

```
cp fixtures/case-17.json /tmp/case-17.json # bisect will check out older trees
git bisect start HEAD HEAD~300
git bisect run ./check_total.sh
```

Exit code 125 tells bisect to skip a commit that can't be tested, here one where the script didn't exist yet, instead of wrongly marking it bad. A crash, by contrast, exits 1, because a crash may well be the bug. The fixture is copied outside the repository first because bisect checks out old versions of the whole tree, which may predate the file. Three hundred commits take about nine runs.

## EXERCISE 15.3 Shrink the input

Importing `orders.csv`, 20,000 rows, fails with `ValueError: day is out of range for month`. Describe how you'd find the offending row in under ten minutes, and what you'd do once you had.

## SOLUTION

1. Keep the header, split the rows in half, and import each half. One fails.
2. Repeat on the failing half. Fifteen halvings take 20,000 rows down to one.
3. Say the row contains `2026-02-29`, and 2026 isn't a leap year. Or `31/04/2026`, from a customer whose export writes the day first.
4. Now reason about the cause. Is the data wrong, or is the parser assuming a date format? Check other rows from the same source.
5. Add the row, minimised, to the importer's tests as a regression case.
6. Decide the behaviour deliberately: reject that row with a clear message and carry on, rather than failing the whole import.

Then make the importer include the line number in its errors. Next time, this exercise takes ten seconds.

## CHECKLIST

- ☐ Can you reproduce it reliably, ideally as a failing test?
- ☐ Did you read the entire error message and stack trace?
- ☐ Is each experiment testing one stated hypothesis?
- ☐ Did you fix the cause, and keep the reproduction as a regression test?
- ☐ Did you look for the same mistake elsewhere?

## GO DEEPER

**BOOK**      [The Debugging Book](#) Andreas Zeller · interactive, in Python, including delta debugging

---

**BOOK**      [Debugging: The 9 Indispensable Rules](#) David Agans · short and practical

---

**BOOK**      [Why Programs Fail](#) Andreas Zeller · the systematic approach in depth

---

**DOCS**      [git bisect](#) including `run` and exit code 125

# Error handling

3 exercises

*Decide deliberately where each failure is detected, where it's handled, and what the user and the operator each get to see. Most error-handling bugs come from never having made those decisions.*

## Three kinds of failure

| KIND                     | EXAMPLE                                                         | RIGHT RESPONSE                                               | WHO NEEDS TO KNOW                                 |
|--------------------------|-----------------------------------------------------------------|--------------------------------------------------------------|---------------------------------------------------|
| <b>Programmer error</b>  | <code>None</code> where a value must exist; an impossible state | Fail fast and loudly. Don't catch it; fix the code           | Developers, through the crash and its stack trace |
| <b>Operational error</b> | Timeout, connection refused, disk full, rate limited            | Expected. Retry if transient and safe; degrade; report       | Operators, through logs and metrics               |
| <b>User error</b>        | Invalid email, expired card, unknown coupon                     | Validate at the boundary; return a specific, helpful message | The user, in words they understand                |

Handling all three the same way, usually with one broad `except`, is the root of most error-handling mistakes.

## Fail fast on bugs

When a program is in a state its author said was impossible, the safest thing is to stop. Carrying on means corrupting data, or acting on it in ways nobody intended. Checks at a function's entry catch bugs close to their cause, where they're cheap to diagnose. The most dangerous lines in Python look like this:

**BEFORE** every failure, bug or outage, becomes silence

```
try:
    sync_inventory()
except Exception:
    pass
```

Someone will discover, weeks later, that inventory hasn't synced since March.

## Handle errors where you have context

Catch an exception only where you can do something meaningful about it: retry, fall back, translate it into a message for the user, or add context and re-raise. Everywhere else, let it propagate. Low-level code usually can't handle anything; the request handler at the top usually can.

When an exception crosses a layer boundary, translate it into the vocabulary of the layer above, and chain the original so nothing is lost.

**EXAMPLE** payments.py

```
class PaymentError(Exception): ...
class CardDeclined(PaymentError): ...
class PaymentProviderUnavailable(PaymentError): ...

def charge(card, amount):
    try:
        return provider.charges.create(card=card.token, amount=amount.cents)
    except provider_sdk.CardError as e:
        raise CardDeclined(e.decline_code) from e
    except (provider_sdk.Timeout, provider_sdk.ServiceUnavailable) as e:
        raise PaymentProviderUnavailable() from e
```

Callers catch `CardDeclined` to show the customer a message, `PaymentProviderUnavailable` to retry or queue, or `PaymentError` for both. Nobody outside this module knows which SDK sits underneath, and `from e` keeps the original traceback attached for whoever debugs it.

## Parse, don't validate

Alexis King's phrase, from a 2019 essay, for a pattern that removes a whole category of bug. A validator checks input and returns a boolean, and the rest of the program has to trust that someone called it. A parser checks input and returns a new type that can only exist if the input was valid. Downstream code takes that type, so it can't receive unchecked data.

**BEFORE** validate, then hope

```
def is_valid_quantity(raw) -> bool:
    return raw.isdigit() and 1 <= int(raw) <= 99

def add_to_cart(cart, sku, raw_quantity):
    if not is_valid_quantity(raw_quantity):
        raise BadRequest("quantity must be 1 to 99")
    cart.add(sku, int(raw_quantity)) # everything below must trust this happened
```

**AFTER** parse once, at the edge

```
@dataclass(frozen=True)
```

```

class Quantity:
    value: int

    def __post_init__(self):
        if not 1 <= self.value <= 99:
            raise ValueError(f"quantity must be 1 to 99, got {self.value}")

    @classmethod
    def parse(cls, raw: str) -> "Quantity":
        if not raw.strip().isdigit():
            raise ValueError(f"quantity must be a whole number, got {raw!r}")
        return cls(int(raw))

def add_to_cart(cart, sku, quantity: Quantity):    # can't be handed junk
    cart.add(sku, quantity)

```

Parse at the edge of the system, as early as possible, and let the types carry the proof inwards. The `EmailAddress` of chapter 11, and `Money`, are the same pattern.

## Retries, done safely

Retrying is the right response to transient failures and the wrong response to everything else. Four rules:

- **Retry only what's transient.** Timeouts, 503 responses, connection resets. Never a validation error or a declined card, which will fail the same way every time.
- **Retry only what's safe to repeat.** Reads are safe. A payment is not, unless the provider supports an idempotency key and you send the same key on every attempt.
- **Back off exponentially, with jitter.** Retrying immediately hammers a service that's already struggling, and if thousands of clients back off on the same schedule, they come back in synchronised waves. Marc Brooker's analysis for AWS showed that randomising the delay, "full jitter", spreads them out.
- **Retry at one layer, with a cap.** If the client, the service and the database driver each retry three times, one user request can become twenty-seven attempts on the database. This is retry amplification.

**EXAMPLE** `retry.py`

```

import random
import time

def call_with_retries(fn, *, attempts=4, base=0.2, cap=5.0,
                     retry_on=(TimeoutError, ConnectionError)):
    for attempt in range(attempts):
        try:
            return fn()
        except retry_on:
            if attempt == attempts - 1:
                raise
            # Full jitter: sleep a random time between zero and the

```

```
# exponential ceiling, so clients don't retry in lockstep.
ceiling = min(cap, base * 2 ** attempt)
time.sleep(random.uniform(0, ceiling))
```

## Every network call needs a timeout

A call without a timeout can wait forever, and a worker waiting forever holds its resources forever. Several popular HTTP clients default to no timeout at all, including the widely used `requests`. Set one explicitly on every call, sized to how long the caller can genuinely afford to wait.

### EXAMPLE

```
requests.get(url) # may hang indefinitely
requests.get(url, timeout=(3.05, 10)) # 3.05 s to connect, 10 s to read
```

## Logging errors

- **Log once, where you handle it.** Logging and re-raising at every layer produces five copies of one error, and nobody can tell whether it was one incident or five.
- **Log structured context.** An order id, a user id and a request id turn a log line into something you can search. `log.error("charge failed", extra={"order_id": order.id})` beats string formatting.
- **Never log secrets or personal data.** Card numbers, passwords, tokens, and often email addresses. Logs are copied to more places, with weaker controls, than your database.
- **Make levels mean something.** Error: a person should look at this. Warning: unexpected but handled, worth watching in aggregate. Info: business events. Debug: detail for diagnosis, normally off in production.

## What the user sees

Users need to know what happened, in their terms, and what they can do next. They don't need a stack trace, which is also an information leak to anyone probing your system. Give them a reference they can quote to support, and log the same reference with full detail, so a support ticket leads straight to the log entry.

### EXAMPLE a last-resort handler, in Flask

```
@app.errorhandler(Exception)
def unexpected_error(error):
    ref = uuid4().hex[:8]
    log.exception("unhandled error", extra={"ref": ref, "path": request.path})
    message = ("Something went wrong on our side. Please try again, or "
               "contact support and quote reference {ref}.")
    return {"error": message, "reference": ref, 500}
```

---

## Design errors away, again

Chapter 4's advice applies here with extra force. Every error case you design away is one nobody has to handle: idempotent deletes, defaults for missing optional values, operations that clamp instead of raising. Keep exceptions for the situations a caller genuinely needs to know about.

---

### COMMON TRAPS

**Catching `Exception` mid-stack.** It catches your bugs too. Catch the specific exceptions you can actually handle.

---

**Returning `None` or `-1` for failure.** Callers forget to check, and the failure surfaces later, somewhere unrelated.

---

**Retrying non-idempotent operations.** Double charges, duplicate emails, repeated shipments.

---

**Silent background jobs.** A job that fails quietly is worse than one that crashes, because a crash at least gets noticed.

---

**Messages that blame.** "Invalid input" helps nobody. "Postcode must be 5 digits; you entered 4" tells the user how to fix it.

---

### EXERCISES

#### EXERCISE 16.1 Find every problem

List every error-handling problem in this function.

```
def sync_order_to_warehouse(order):
    for attempt in range(10):
        try:
            resp = requests.post(WAREHOUSE_URL, json=order.to_dict())
            if resp.status_code == 200:
                return True
        except:
            print("error")
            time.sleep(1)
    return False
```

---

#### SOLUTION

1. No timeout on the POST, so a hung warehouse hangs this worker forever.

2. A bare `except:` catches everything, including bugs in `to_dict()` and even `KeyboardInterrupt`.
3. The error is thrown away. `print("error")` carries no detail and no order id, and may never reach a log.
4. Non-200 responses aren't treated as errors. A 500 loops again with no pause at all, and a 400, which means our payload is wrong, is retried ten times though it can never succeed.
5. A 201 Created or 202 Accepted is treated as failure, because only 200 counts as success.
6. A fixed one-second sleep: no backoff, no jitter.
7. If the warehouse received the order but the response was lost, the retry may create a duplicate shipment. It needs an idempotency key, such as one derived from the order id.
8. It returns `False` on failure, which callers can ignore. It should raise.

### EXERCISE 16.2 Fix it

Rewrite `sync_order_to_warehouse` using this chapter. You may use `call_with_retries` from above.

#### SOLUTION

```
class WarehouseRejected(Exception): ...
class WarehouseUnavailable(Exception): ...

def sync_order_to_warehouse(order, session=requests):
    def attempt():
        try:
            resp = session.post(
                WAREHOUSE_URL,
                json=order.to_dict(),
                headers={"Idempotency-Key": f"order-{order.id}"},
                timeout=(3.05, 10),
            )
        except (requests.Timeout, requests.ConnectionError) as e:
            raise WarehouseUnavailable() from e
        if resp.status_code >= 500:
            raise WarehouseUnavailable(f"status {resp.status_code}")
        if resp.status_code >= 400:
            raise WarehouseRejected(f"status {resp.status_code}: {resp.text[:200]}")

    call_with_retries(attempt, attempts=4, retry_on=(WarehouseUnavailable,))
```

Transient failures retry with backoff and jitter. Permanent ones fail at once, carrying the detail needed to fix them. The idempotency key makes a retry after a lost response safe. And failure raises, rather than

returning a value that can be ignored. Logging is left to the caller that decides what to do next, so the error is logged exactly once.

### EXERCISE 16.3 Design the hierarchy

Design exception classes for a shipping-label module that calls a carrier's API. Callers must be able to tell apart three situations: the address is wrong and the customer must correct it; the carrier is down and the job should retry later; our account with the carrier is misconfigured and an engineer must step in.

#### SOLUTION

```
class LabelError(Exception):
    """Base class: anything that stopped a label being created."""

class InvalidAddress(LabelError):
    """The customer must correct the address. Not retryable."""

    def __init__(self, field: str, reason: str):
        super().__init__(f"{field}: {reason}")
        self.field, self.reason = field, reason

class CarrierUnavailable(LabelError):
    """Transient. Safe to retry later."""

class CarrierMisconfigured(LabelError):
    """Our credentials or account set-up is wrong. Alert an engineer."""
```

Each class maps to exactly one response, and its docstring says which.

`InvalidAddress` carries the field, so the interface can highlight it. A caller can catch `LabelError` to handle everything generically, or the subclasses to act precisely.

#### CHECKLIST

- ☐ Is every `except` catching a specific exception you can actually handle?
- ☐ Does every network call have a timeout?
- ☐ Are retries limited to transient, idempotent operations, with backoff and jitter, at one layer?
- ☐ Is each error logged once, with context, and without secrets?
- ☐ Does the user get a helpful message and a reference, never a stack trace?

## GO DEEPER

**ESSAY** Parse, don't validate Alexis King, 2019

---

**ESSAY** Exponential Backoff and Jitter Marc Brooker, AWS Architecture Blog

---

**ESSAY** Timeouts, retries, and backoff with jitter Amazon Builders' Library

# Performance: measure first

3 exercises

*Most code is fast enough. When it isn't, find the slow part by measuring, fix that part, and measure again. Intuition about where time goes is wrong often enough that it isn't worth consulting.*

## The whole quotation

"Premature optimization is the root of all evil" is one of the most quoted lines in programming, nearly always without its context. Donald Knuth wrote it in 1974, and the same passage goes on to say that we shouldn't pass up our opportunities in the critical three percent. His point wasn't that performance doesn't matter. It was that performance matters in a small part of a program, and you can't guess which part.

## Decide what "fast enough" means

Before optimising, set a target in terms users experience: "search returns in under 300 ms for 95% of requests", not "make search faster". Without a target, you don't know when to stop.

Measure percentiles, not averages. If 99 requests take 50 ms and one takes 5 seconds, the average is about 100 ms, which describes nobody's experience. The 99th percentile of 5 seconds describes the user who gave up. At scale the slow tail is hit constantly, because a page that makes twenty backend calls is slowed down by whichever one hits its tail.

## Profile before you touch anything

A profiler tells you where time actually goes. In Python:

### EXAMPLE

```
python -m cProfile -s cumulative report.py | head -25 # exact, per function
py-spy record -o profile.svg -- python report.py      # sampled, draws a flame graph
py-spy top --pid 12345                               # live view of a running process
```

cProfile ships with Python and counts every call, which adds overhead but gives exact numbers. py-spy samples a running program from the outside with little overhead, so it's safe to point at production. In a flame graph, width is time: look for the widest bars in code you own.

For a small piece of code, benchmark with `timeit`, which repeats the measurement to average out noise.

#### EXAMPLE

```
python -m timeit -s "items = list(range(10_000)); wanted = set(range(0, 10_000, 7))" \
    "[i for i in items if i in wanted]"
```

## Amdahl's law

Speeding up part of a program only helps in proportion to that part's share of the total. If a request spends 90% of its time waiting on the database, making your Python ten times faster saves at most 9%. That's why profiling comes first: it tells you which part is worth speeding up at all.

## Where the big wins usually are

Roughly in order of how often they matter in application code:

- 01 **Round trips.** The N+1 query: fetch a list, then query once for each item. Each query is fast; a thousand of them aren't. Batch them.
- 02 **Algorithmic complexity.** A list lookup inside a loop is quadratic. A set or a dict makes it linear.
- 03 **Doing less work.** Paginate, filter in the database rather than in Python, compute lazily, and skip what the caller doesn't use.
- 04 **Caching.** Powerful, and it brings invalidation, staleness and memory problems with it. Cache only what you've measured to be slow and repeated.
- 05 **Concurrency for waiting.** If time goes on waiting for the network, overlapping the waits helps. If it goes on computation, threads won't help in standard CPython; you need processes or native code.
- 06 **Micro-optimisations.** Caching attribute lookups in hot loops and the like. Real, small, and almost never the answer.

**BEFORE** N+1: 101 queries, in Django

```
orders = Order.objects.filter(status="paid")[:100]
for order in orders:
    print(order.customer.email)      # one extra query per order
```

**AFTER** one query with a join

```
orders = Order.objects.filter(status="paid").select_related("customer")[:100]
for order in orders:
    print(order.customer.email)
```

SQLAlchemy's equivalents are `joinedload` and `selectinload`. Turn on query logging in development and count the queries each request makes. The N+1 is one of the most common performance bugs in web applications, and one of the easiest to fix once seen.

**BEFORE** rebuilds a list of ids for every customer

```
def customers_without_orders(customers, orders):  
    return [c for c in customers if c.id not in [o.customer_id for o in orders]]
```

**AFTER**

```
def customers_without_orders(customers, orders):  
    has_ordered = {o.customer_id for o in orders}  
    return [c for c in customers if c.id not in has_ordered]
```

With 10,000 customers and 100,000 orders, the first version performs billions of operations. The second performs about 110,000.

## Keep optimised code readable

Optimised code is often less obvious. Leave a why comment with the measurement that justified it, so the next reader doesn't "simplify" it back. Ideally, keep a benchmark in the repository, so a regression shows up in CI rather than in production.

---

### COMMON TRAPS

**Optimising without a profile.** You'll speed up the part that wasn't slow.

---

**Benchmarking badly.** One run, a cold cache, a laptop on battery, a debug build. Repeat, warm up, and compare against a baseline on the same machine.

---

**Trusting averages.** Look at the 95th and 99th percentiles.

---

**Caching first.** A cache hides a slow path without fixing it, and adds a consistency problem.

---

**Optimising code that runs once.** A migration that takes four minutes instead of two doesn't matter.

---

### EXERCISES

---

#### EXERCISE 17.1 Spot the complexity

This runs nightly, and has started taking hours. What's its complexity, and what's the fix?

```
def find_duplicate_emails(users):
    duplicates = []
    for i, a in enumerate(users):
        for b in users[i + 1:]:
            if a.email.lower() == b.email.lower() and a.email.lower() not in duplicates:
                duplicates.append(a.email.lower())
    return duplicates
```

#### SOLUTION

Quadratic in the number of users. On top of that, `users[i + 1:]` copies the rest of the list on every iteration, and `not in duplicates` scans a list. At 200,000 users that's around twenty billion comparisons. Count instead:

```
from collections import Counter

def find_duplicate_emails(users):
    counts = Counter(u.email.lower() for u in users)
    return [email for email, n in counts.items() if n > 1]
```

Linear, shorter, and it returns the same list in the same order, because both versions report duplicates in order of first appearance. Confirm claims like that with a quick randomised comparison of old and new before you ship the change.

#### EXERCISE 17.2 Count the queries

A page listing 50 invoices makes 151 database queries. Explain where they come from in this template loop, and fix it.

```
invoices = Invoice.objects.all()[:50]
for invoice in invoices:
    render(invoice.number,
           invoice.customer.name,
           invoice.customer.company.name,
           invoice.lines.count())
```

#### SOLUTION

One query for the invoices. Then, for each of the 50: one for the customer, one for the customer's company, and one to count the lines. That's  $1 + 50 \times 3 = 151$ . Join the foreign keys and annotate the count, so the database does it all at once:

```

from django.db.models import Count

invoices = (Invoice.objects
            .select_related("customer__company")
            .annotate(line_count=Count("lines"))[:50])

for invoice in invoices:
    render(invoice.number,
           invoice.customer.name,
           invoice.customer.company.name,
           invoice.line_count)

```

### EXERCISE 17.3 Read the profile

Where do you look first, and what do you suspect?

| ncalls | totttime | percall | cumtime | percall | filename:lineno(function)                |
|--------|----------|---------|---------|---------|------------------------------------------|
| 1      | 0.002    | 0.002   | 14.210  | 14.210  | report.py:10(build_report)               |
| 12000  | 0.051    | 0.000   | 13.870  | 0.001   | report.py:44(price_for)                  |
| 12000  | 0.094    | 0.000   | 13.790  | 0.001   | client.py:88(get)                        |
| 12000  | 13.520   | 0.001   | 13.520  | 0.001   | {method 'recv_into' of '_socket.socket'} |
| 1      | 0.310    | 0.310   | 0.310   | 0.310   | report.py:70(render_table)               |

#### SOLUTION

Nearly all of the 14 seconds is in `recv_into`, waiting on a socket, reached through 12,000 calls to `price_for`. The report fetches one price per line over the network, at roughly a millisecond each. The Python code itself is almost free, so optimising it would be pointless: Amdahl again.

Batch the price lookups into a single request, or cache prices for the duration of the report. Rendering, at 0.3 seconds, isn't worth touching.

#### CHECKLIST

- ☐ Is there a target, stated as a percentile?
- ☐ Did a profiler, not intuition, choose what to optimise?
- ☐ Did you measure before and after, on the same machine?
- ☐ Did you check how many queries each request makes?
- ☐ Is the optimised code commented with its justification?

#### GO DEEPER

|       |                                                                                               |
|-------|-----------------------------------------------------------------------------------------------|
| PAPER | The Tail at Scale Dean & Barroso, 2013 · why the slowest requests dominate                    |
| TOOL  | py-spy a sampling profiler you can attach to production                                       |
| BOOK  | Systems Performance Brendan Gregg · the reference for everything below your code              |
| PAPER | Structured Programming with go to Statements Donald Knuth, 1974 · the famous line, in context |

## PART VII

# Beyond the code

The last two chapters are about the work around the code: writing that lets decisions outlive the meeting where they were made, and estimating and slicing work honestly enough that other people can plan around it.

# Technical writing

3 exercises

*Writing is how an engineering decision reaches people who weren't in the room, including your future self. Write for a specific reader who has less context than you, and put the conclusion first.*

## Principles

- **Know your reader, and what they'll do next.** A design doc for a sceptical senior engineer and a README for someone on their first day need different things. Name the reader before you start.
- **Lead with the conclusion.** Put the decision, recommendation or answer in the first paragraph. Busy readers stop early, so make sure they stop knowing the important part.
- **One idea per paragraph,** carried by its first sentence. Someone skimming only first sentences should still get the argument.
- **Concrete beats abstract.** "Cuts p99 checkout latency from 1.8 s to 400 ms" beats "improves performance".
- **Show, with examples.** One example of an API call explains more than a paragraph describing it.
- **Cut.** Most first drafts improve by losing a third. Delete the throat-clearing, the hedges and the repetition.

### BEFORE

In this document we will be looking at some of the various options that are available to us with regard to the question of how we might potentially go about improving the performance characteristics of the search functionality, which has been the subject of some complaints from certain users recently.

### AFTER

Search is too slow: p95 latency is 2.4 s, and support receives about ten complaints a week. This document recommends moving search to a dedicated index, which prototypes show brings p95 under 300 ms. Section 4 covers the two alternatives we considered, and why we rejected them.

## Four kinds of documentation

Daniele Procida's **Diátaxis** framework observes that documentation serves four different needs, and that most bad documentation mixes them. A tutorial that pauses to recount design history loses its learner. A reference page that tells stories is useless for looking things up.

| TYPE                | READER'S NEED              | READS LIKE                            | EXAMPLE                                    |
|---------------------|----------------------------|---------------------------------------|--------------------------------------------|
| <b>Tutorial</b>     | Learning, by doing         | A lesson with a guaranteed outcome    | "Build your first plugin"                  |
| <b>How-to guide</b> | Completing a specific task | A recipe                              | "Rotate the API keys"                      |
| <b>Reference</b>    | Looking something up       | A dictionary: accurate, complete, dry | Every configuration option and its default |
| <b>Explanation</b>  | Understanding why          | An essay                              | "Why orders use event sourcing"            |

## The README

A README's job is to take a stranger from nothing to productive. The test: can a new engineer get from cloning the repository to a running, tested build using only the README, in under fifteen minutes?

### EXAMPLE README.md

#### # order-service

Takes orders from checkout, charges payment, and hands paid orders to the warehouse. Owned by the Checkout team (#checkout).

#### ## Quick start

```
cp .env.example .env
docker compose up -d postgres
make install migrate
make test          # all green? you're set up
make run           # then open http://localhost:8000/health
```

#### ## How it fits

Checkout calls POST /orders. We charge through the payments service and publish `OrderPaid` to the `orders` topic, which the warehouse consumes.

#### ## Common tasks

- Add a migration: `make migration name=add\_tax\_id`
- Replay a failed warehouse sync: docs/runbooks/replay-sync.md

#### ## Decisions

Architecture decision records live in docs/adr/.

## Design documents

Write a design doc when getting something wrong would be expensive, or when several people need to agree. Its value lies less in the document than in the thinking it forces, and in the review it invites while changing course is still cheap. Google's practice, as described by Malte Ubl, suggests a structure like this:

- **Context and scope.** What exists today, and what this document covers.
- **Goals and non-goals.** Non-goals are things that could reasonably be goals but that you're choosing not to pursue. "Supporting multiple currencies" is a useful non-goal. "Not crashing" is not.
- **The design.** An overview first, then the details that matter: interfaces, the data model, the flows that are hard.
- **Alternatives considered.** The section that earns the document its keep. Each alternative, including doing nothing, and the trade-off that ruled it out.
- **Cross-cutting concerns.** Security, privacy, observability, migration, cost.
- **Open questions.** What you haven't decided yet, stated plainly.

Keep it as short as the decision allows. A design doc nobody finishes reading has failed at its only job.

## Architecture decision records

An ADR records one decision, briefly, in the repository, where the code's future readers will find it. Michael Nygard proposed the format in 2011: a title, a status, the context that forced a choice, the decision itself, and its consequences, good and bad. ADRs are never edited after acceptance. When a decision changes, a new ADR supersedes the old one, so the history of thinking survives.

**EXAMPLE** docs/adr/0007-row-locks-for-stock.md

### # 7. Use PostgreSQL row locks for stock reservation

Date: 2026-03-14

Status: Accepted

#### ## Context

Two checkouts can reserve the last unit of an item at the same moment, and we oversold 41 orders during the March sale. Stock lives in PostgreSQL, and reservations peak at about 300 per second.

#### ## Decision

Reserve stock with `SELECT ... FOR UPDATE` on the stock row, inside the checkout transaction. We will not introduce a separate lock service.

#### ## Consequences

- Overselling becomes impossible for single-warehouse items.
- Popular items serialise on their row. Load tests show p99 reservation time of 45 ms at peak, which we accept.
- Multi-warehouse stock, planned for Q4, will need a new decision: this approach doesn't extend to it cleanly.

---

## Runbooks and postmortems

A **runbook** is a how-to guide for someone under stress at three in the morning. Make every step a command they can copy, say what they should see, and say what to do if they don't see it.

A **postmortem** is written after an incident: a timeline, the impact, the contributing causes, what went well, and action items with owners. Write it blamelessly. The useful question is what about the system let a reasonable person make that mistake, because that's the only question whose answer prevents the next one.

## Writing, and assistants

An assistant can produce a fluent design doc in seconds, and it will read well. What it can't do is make the decisions the document exists to record, or know your system's real constraints. Much of the value of writing is that it exposes gaps in your own thinking, and a generated draft skips that step unless you interrogate every claim in it. Use tools to tighten prose you've reasoned through, not to replace the reasoning.

---

### COMMON TRAPS

**Burying the lede.** Three paragraphs of background before the recommendation.

---

**Writing for yourself.** Undefined acronyms, skipped steps, and the word "obviously".

---

**Docs that rot.** Documentation away from the code, unreviewed and unowned, decays. Keep docs in the repository and update them in the same pull request as the behaviour.

---

**Design docs as theatre.** A doc written after the code, to satisfy a process, is a description, not a design.

---

### EXERCISES

#### EXERCISE 18.1 Put the conclusion first

Rewrite this status message so a busy engineering manager gets what they need from its first two sentences.

```
So I've been looking into the payment timeout issue and it's been kind of a
rabbit hole. I tried a few things with the connection pool which didn't help,
then looked at the provider's status page which was fine. Eventually I found
that our retry logic was retrying on 4xx errors too, which was causing extra
load. I've got a fix but it needs review and I also think we should talk about
whether to add a circuit breaker, maybe next sprint? Also the fix probably
```

won't go out until Thursday because of the freeze.

## SOLUTION

Found the cause of the payment timeouts: our retry logic was retrying 4xx errors, adding load on every failure. The fix is ready for review and ships Thursday, after the freeze.

One decision needed: should we add a circuit breaker to the payment client? I'd suggest next sprint, and I'm happy to write a short proposal.

The journey, the connection pool and the status page, is gone. It mattered to the writer, not to the reader.

## EXERCISE 18.2 Write an ADR

Choose a real technical decision you've made in a project, such as a database, a framework or a library. Write an ADR for it in the format above, in under 300 words, including at least one negative consequence.

## SOLUTION

There's no single answer. Check yours against these questions:

- Could someone who joins in a year understand why this was decided, without asking you?
- Does the context describe the forces at play, not just the situation?
- Is there at least one honest downside?
- Is the decision stated as a decision, rather than as a description of the code?

## EXERCISE 18.3 Classify the documents

Which Diátaxis type is each of these, and which one is mixing types?

- A. "timeout (integer, seconds, default 30): the request timeout. retries ..."
- B. "In this guide you'll build a small to-do app and deploy it. By the end ..."
- C. "To restore from backup: 1. Stop the service. 2. Run restore --latest ..."
- D. "We chose eventual consistency for the catalogue because ..."
- E. "Step 3: install the CLI. (The CLI was first written in Perl in 2014, when the team felt that ... [four more paragraphs of history])"

## SOLUTION

A is reference. B is a tutorial. C is a how-to guide. D is explanation. E is a tutorial or how-to guide interrupted by explanation: move the history to an explanation page, and link to it from step 3.

---

#### CHECKLIST

- ☐ Is the conclusion in the first paragraph?
- ☐ Can you name the reader, and what they'll do after reading?
- ☐ Does each document serve one need: learning, doing, looking up, or understanding?
- ☐ For decisions, is there an ADR or design doc that includes the alternatives you rejected?
- ☐ Does the documentation live in the repository, updated in the same pull request as the change?

#### GO DEEPER

**FREE**      [Diátaxis](#) Daniele Procida · the four kinds of documentation

---

**ESSAY**      [Design Docs at Google](#) Malte Ubl

---

**ESSAY**      [Documenting Architecture Decisions](#) Michael Nygard, 2011

---

**COURSE**      [Technical Writing courses](#) Google · short, free, excellent

# Estimation, decomposition and scope

3 exercises

*An estimate is a forecast under uncertainty. Make it honest by breaking the work into small pieces, stating your assumptions, giving a range, and cutting scope rather than stretching time when reality disagrees.*

## Why estimates go wrong

- **The planning fallacy.** Daniel Kahneman and Amos Tversky found that people estimate by imagining the task going well. Nothing in that imagined version gets interrupted, blocked, or rewritten.
- **Hofstadter's law.** It always takes longer than you expect, even when you take Hofstadter's law into account. Correcting for optimism once isn't enough.
- **Forgotten work.** The estimate covers writing the code. It leaves out tests, review, deployment, documentation, the migration, and two days of back-and-forth about what was actually wanted.
- **Unknown unknowns.** The integration that turns out to need an API the vendor doesn't have.

## The cone of uncertainty

At the start of a project, estimates are genuinely uncertain, often by a factor of four in either direction, according to Barry Boehm's data as popularised by Steve McConnell. The range narrows as requirements, design and implementation turn unknowns into knowns. So an early estimate should be a wide range, revised as you learn. A single number at the start is false precision.

## Techniques that help

- **Decompose until the pieces are small.** A task estimated at more than a couple of days is hiding unknowns. Break it down until each piece is about a day. The decomposition is often worth more than the estimate.
- **Turn unknowns into spikes.** "Find out whether the payment provider supports partial refunds" is a time-boxed task whose deliverable is an answer. Estimate the rest once you have it.
- **Give ranges.** Three-point estimation asks for optimistic, likely and pessimistic values. The PERT formula weights them as  $(O + 4M + P) \div 6$ , but the most useful part is having said the pessimistic number out loud.

- **Take the outside view.** Kahneman's remedy for the planning fallacy is reference class forecasting: instead of imagining this task, ask how long similar tasks actually took. Record your estimates and actuals; your own history is the best reference class you'll ever have.
- **Separate estimates, targets and commitments.** An estimate is a prediction, a target is a business wish, and a commitment is a promise. McConnell's point is that they're different things, and much misery comes from a target being heard as an estimate, or an estimate as a commitment.

## Slice vertically

Split work into **vertical slices**, thin pieces of end-to-end functionality a user could see, rather than horizontal layers like "all the database work, then all the API, then all the interface". Layers deliver nothing until the last one lands, and integration problems appear at the end, when there's no time left. Slices deliver something at every step, and surface integration problems in the very first one. *The Pragmatic Programmer* calls that first slice a tracer bullet: a skeleton that goes all the way through, so you can see whether you're on target.

Take a request to let customers export their order history as a CSV file.

| SLICE                     | WHAT SHIPS                                                                                             | ESTIMATE  |
|---------------------------|--------------------------------------------------------------------------------------------------------|-----------|
| <b>1. Tracer bullet</b>   | A button that downloads a CSV of order ids and dates for the logged-in customer. End to end, and ugly. | 0.5–1 day |
| <b>2. All the columns</b> | Totals, items and status, with money formatted properly.                                               | 0.5 day   |
| <b>3. Date filter</b>     | Export a chosen date range. Needs a date picker in the interface.                                      | 1–2 days  |
| <b>4. Large histories</b> | Customers with 10,000+ orders time out, so generate the file in a background job and email a link.     | 2–4 days  |
| <b>5. Polish</b>          | Accessible controls and localised column headers.                                                      | 1 day     |

After slice 1 there's something to show. After slice 2 most customers are served. Slice 4 has the widest range because it's the least known. If time runs out, slices 3 to 5 become a conversation about scope rather than a missed deadline.

## Negotiate scope, not quality

When the estimate exceeds the time available, there are four levers: time, scope, people and quality. Fred Brooks observed that adding people to a late software project makes it later. Cutting quality, by skipping tests or review, borrows against the future at a punishing rate. That leaves time and scope, so offer options rather than a yes or no.

**EXAMPLE** a message that offers choices

We can't ship all of CSV export by the 30th. Two options:

- A. Slices 1 and 2 by the 30th: full export for all but the largest ~2% of customers. Large histories follow two weeks later.
- B. Everything, on the 14th of next month.

I'd suggest A: most customers get it sooner, and the risky part, large histories, gets the time it needs.

## Report progress honestly

- **Report in finished slices, not percentages.** "Two of five slices are in production" can be checked. "About 80% done" can't, and Tom Cargill's ninety-ninety rule applies: the first 90% of the code takes the first 90% of the time, and the remaining 10% takes the other 90%.
- **Raise risks early.** The day you learn an estimate is wrong is the day to say so. A slip announced three weeks ahead is a planning input. Announced the day before, it's a crisis.
- **Re-estimate as you learn.** A forecast that never changes isn't a forecast.

## A note on #NoEstimates

Some teams stop estimating individual items altogether. If work is sliced into small pieces of similar size, you can forecast from throughput: we finish about eight slices a week, this feature is about twenty-four slices, so roughly three weeks. It works well for teams with steady flow, and it rests on the same underlying skill as good estimation: decomposing well.

---

### COMMON TRAPS

**Giving a number on the spot.** "Let me break it down and come back to you this afternoon" is a professional answer.

**Padding silently.** Hidden buffers erode trust when they're discovered. Express uncertainty openly, as a range.

**Estimating only the coding.** Include testing, review, deployment, migration and documentation, or say explicitly that you haven't.

**Horizontal slicing.** It feels efficient and delivers nothing until the very end.

---

**Accepting a deadline as an estimate.** If someone hands you a date and asks you to confirm it, you've been asked for a commitment. Say which one you're giving.

---

## EXERCISES

### EXERCISE 19.1 Slice it vertically

Feature request: customers can save items to a wishlist, and get an email when a saved item goes on sale. Propose four or five vertical slices in delivery order, each with a range estimate and one sentence on what's possible once it ships.

---

#### SOLUTION

1. **Save and view**, 1 to 2 days. A heart button on product pages adds to a wishlist page, for logged-in customers only. Customers can save items and find them again.
2. **Remove, and move to cart**, half a day to a day. The wishlist becomes useful for buying.
3. **Sale detection**, 1 to 2 days. A daily job finds wishlisted items whose price dropped, and only logs and counts them. Nothing visible changes, but you learn how many emails you would send, which sizes the next slice.
4. **Sale emails**, 2 to 4 days. At most one email per customer per day, listing their discounted items, with unsubscribe. The promised feature is live.
5. **Guest wishlists merged at login**, 2 to 3 days, and a good candidate to cut. Guests stop losing their saved items.

Slice 3 is deliberately invisible. It's cheap, and it answers the question that slice 4's estimate depends on.

### EXERCISE 19.2 Three points, one message

You estimate a data migration at optimistic 2 days, most likely 4, and pessimistic 12, because nobody has ever fully inspected the legacy data. What's the PERT estimate, and what do you actually tell your manager?

#### SOLUTION

$(2 + 4 \times 4 + 12) \div 6 = 30 \div 6 = 5$  days. But the useful message is about the skew:

Probably four or five days, but it could take up to two and a half weeks, because nobody has inspected the legacy data. I can cut that uncertainty with a one-day spike to profile the data first, and then give you a much tighter range.

The spike is the real recommendation. The formula only tells you that the range is lopsided.

#### EXERCISE 19.3 Rewrite the status update

Rewrite this so it's verifiable and surfaces the risk.

Export feature is going well, about 75% done. Should be finished soon.

#### SOLUTION

Export: slices 1 and 2 are in production (basic CSV with all columns). Slice 3, the date filter, is in review and should ship Wednesday.

Risk: slice 4 (large histories) needs a background job, and this service has no job queue yet. That could add 3 to 5 days. I'll know by Friday whether we can reuse the billing service's queue.

#### CHECKLIST

- ☐ Is every estimate a range, with its assumptions written down?
- ☐ Are tasks about a day each, with unknowns turned into spikes?
- ☐ Is the work sliced vertically, so each slice is usable on its own?
- ☐ When time ran short, did you offer scope options rather than cutting quality?
- ☐ Is progress reported in finished slices, and risk raised the day you saw it?

#### GO DEEPER

BOOK      **Software Estimation: Demystifying the Black Art** Steve McConnell · estimates, targets, commitments, and the cone

BOOK      **Thinking, Fast and Slow** Daniel Kahneman · the planning fallacy and the outside view

---

BOOK      **The Mythical Man-Month** Fred Brooks · why adding people to a late project makes it later

---

BOOK      **The Pragmatic Programmer** Hunt & Thomas · tracer bullets and estimating

## THE NINETEEN RULES

---

If you forget everything else, keep these: one per chapter, each linked back to it. They make a decent checklist for your own pull requests.

01 Name things for what they mean, in the domain's own words, with units.

---

02 One thing per function, at one level of abstraction, with effects pushed to the edges.

---

03 Comment the why and the interface. Delete the narration.

---

04 Hide a lot behind a little, and design errors out of existence.

---

05 Apply a principle only when you can name the change it makes cheaper.

---

06 Compose behaviour and inject dependencies, including time and randomness.

---

07 Use a pattern when you have its problem, in its lightest form.

---

08 Test behaviour through public interfaces, at the boundaries, deterministically.

---

09 Fake what's slow or effectful, and keep fakes honest with contract tests.

---

10 Test first for feedback, state properties for the cases you can't imagine, and test your tests.

---

11 Refactor in small green steps, wearing one hat at a time.

---

12 Characterise before changing. Sprout and wrap. Strangle, don't rewrite.

---

13 Keep changes small, review design first, and label what blocks.

---

14 Integrate daily, explain why in every commit, and deploy dark.

---

15 Reproduce, hypothesise, bisect, and keep the reproduction as a test.

---

16 Fail fast on bugs, retry only the transient and idempotent, and time out

everything.

- 
- 17 Set a percentile target, profile first, and fix the widest bar.
- 
- 18 Lead with the conclusion, for a named reader, and record decisions beside the code.
- 
- 19 Estimate in ranges, slice vertically, and trade scope, never quality.
-

## READING LIST

---

The books behind this volume, and the chapters each one deepens. If you read only one, make it *A Philosophy of Software Design*: it's short, and it changes how you see every other chapter.

| BOOK                                                      | AUTHOR                           | DEEPENS       |
|-----------------------------------------------------------|----------------------------------|---------------|
| <b>A Philosophy of Software Design</b>                    | John Ousterhout                  | Ch. 1–4       |
| <b>The Pragmatic Programmer, 20th anniversary edition</b> | David Thomas & Andrew Hunt       | Ch. 1, 15, 19 |
| <b>Refactoring, 2nd edition</b>                           | Martin Fowler                    | Ch. 11        |
| <b>Working Effectively with Legacy Code</b>               | Michael Feathers                 | Ch. 6, 12     |
| <b>Test-Driven Development: By Example</b>                | Kent Beck                        | Ch. 10        |
| <b>xUnit Test Patterns</b>                                | Gerard Meszaros                  | Ch. 8, 9      |
| <b>Design Patterns</b>                                    | Gamma, Helm, Johnson & Vlissides | Ch. 7         |
| <b>Tidy First?</b>                                        | Kent Beck                        | Ch. 11        |
| <b>Domain-Driven Design</b>                               | Eric Evans                       | Ch. 1, 7      |
| <b>Why Programs Fail</b>                                  | Andreas Zeller                   | Ch. 15        |
| <b>Systems Performance</b>                                | Brendan Gregg                    | Ch. 17        |
| <b>Software Estimation</b>                                | Steve McConnell                  | Ch. 19        |

---

Compiled September 2026 as the companion to stage 09 of The Engineer's Syllabus. Research drew on Google's engineering practices, the Ousterhout and Martin exchange, Fowler on test shapes, CUPID, Hodgson on feature toggles, the 2025 DORA report, Bacchelli and Bird, xUnit Patterns, Diátaxis, Nygard on ADRs, SmartBear's Cisco study, Brooker on jitter, Alexis King, Hyrum's law, Conventional Commits, Semantic Versioning, the strangler fig and Ubl on design docs.

The self-contained examples in chapters 2, 10, 11, 12, 16 and 17 were run before publication to confirm they behave as the text says, including the refactoring in chapter 11, which was checked against the original on two thousand random orders. Re-checked in October 2026 with Python 3.14.7, every claim held except that one: on Python 3.12 and later the refactored receipt's total is a cent different for about one order in twenty, and chapter 11 now says so. Examples that depend on a framework or a database are illustrative.