LIBRARY

The Developer's Great Book

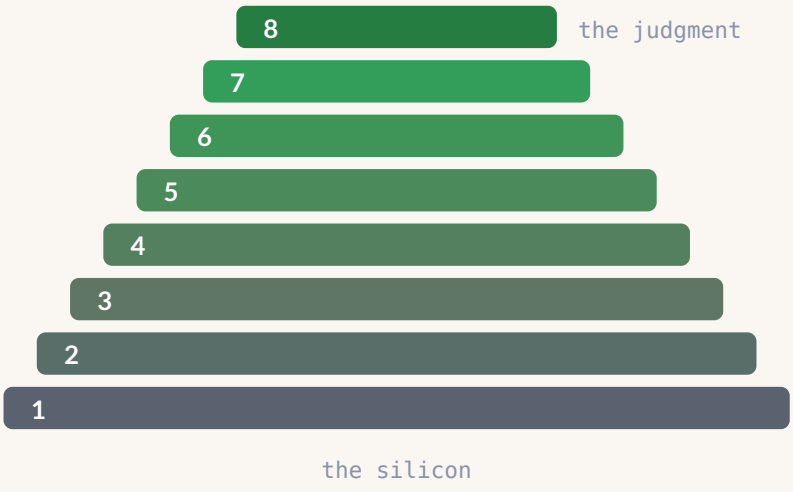
The whole craft woven into one thread

From bits and Big O up to clean code, architecture, networking, delivery, human-centred software and the AI era: the marrow of dozens of tech books, recombined into a single thread.

by

HUGONNOT Odilon

Edition of 30 October 2026
web-developpeur.com



Contents

1 The machine only understands numbers 4

1.1 Everything is a number 5
1.2 Memory is a pyramid 15
1.3 Cost has a unit: Big O 21
1.4 Think like the machine, write high-level 26

2 Language, a tool for thinking 32

2.1 The three moves: chain, repeat, choose 33
2.2 Values and references: what the variable really holds 39
2.3 Types are sets of values 44
2.4 Functions are values 51
2.5 Recursion: solving a problem with a smaller version of itself 58
2.6 Naming a concept changes what you can think 62
2.7 Three ways to write a program 67

3 Writing for humans 70

3.1 The name reveals the intent 71
3.2 The deep function, not just the short one 73
3.3 The comment says why, not what 75
3.4 Define the error out of existence 77
3.5 Don't repeat yourself, and make change easy 81
3.6 Refactoring is changing form without changing behavior 83

4 Giving the system a shape 87

4.1 Compose rather than inherit 88
4.2 A pattern answers a need, not a goal 91
4.3 Depend on abstractions, invert the dependencies 95
4.4 No "best practice", only the least-bad trade-off 102
4.5 Conceptual integrity: one mind 104
4.6 Store to find again: the database 107

5 Two machines talk 112

5.1 Distance does not compress 113
5.2 The hourglass: a tiny contract in the middle 114
5.3 TCP: building reliable out of unreliable 116
5.4 Congestion: sharing without a boss 118
5.5 The web pays the toll 120

6 Prove it, ship it, hold the load 123

6.1 Test first 124
6.2 Debugging is refusing to believe 127
6.3 Versioning is a content-addressable system 129
6.4 Flow beats effort 132
6.5 Shipping often is less scary than shipping rarely 133
6.6 Reads scale by copying, writes by splitting 136
6.7 At scale, guarantees have a price 138

7 Software is for humans 141

7.1 Don't make me think 141
7.2 An API is designed for its caller 143
7.3 Accessible by construction 143
7.4 The organization shows up in the code 144
7.5 Adding people to a late project makes it later 146
7.6 Think like the attacker to defend 147

8 Coding in the age of AI 150

8.1 AI generates the probable, not the true 151
8.2 Everything turns on context 153
8.3 AI is steered in short loops 155
8.4 The surgical team, finally feasible 157
8.5 Human judgment is the last link 158

Glossary 163

Foreword

You learn to code in fragments: a language one year, a method the next month, a trick on some random Thursday night. The knowledge piles up, but nothing ties it together, and you end up applying rules without knowing where they come from.

This book does not teach you to code: it teaches you to code **well**, in any language and at any level. It strings the fragments like beads onto **a single thread**, from the current inside the silicon to the human judgment no AI replaces.

Each chapter is the **consequence of the one before**, and each idea is told only once, in its place. This is not a collection of summaries: the best of dozens of books is stitched together here, and their titles are listed at the bottom of each section. Read top to bottom and the whole field assembles itself.

The promise is simple. Whether you are just starting out or have years of code behind you, you walk away knowing not just what to do but **why** you do it, **where** it breaks and **when** to hold back.

1 The machine only understands numbers

what code costs

The machine does not think: numbers go in, numbers come out. A piece of text, an image, a condition, it all becomes numbers, and every computation has a cost. The same task can be instant or crawl for seconds, depending on how you write it. This chapter teaches you to see that cost before you pay it. Four steps: the smallest cell of memory, the layers where data lives, the unit that measures cost, and the translator (the software that converts your code into instructions for the processor chip) that handles the rest.

If you already code, do not skip this chapter: you will find out why a sorted array scans faster than the same array shuffled, why `O(1)` sometimes lies, and why "café" weighs five bytes.

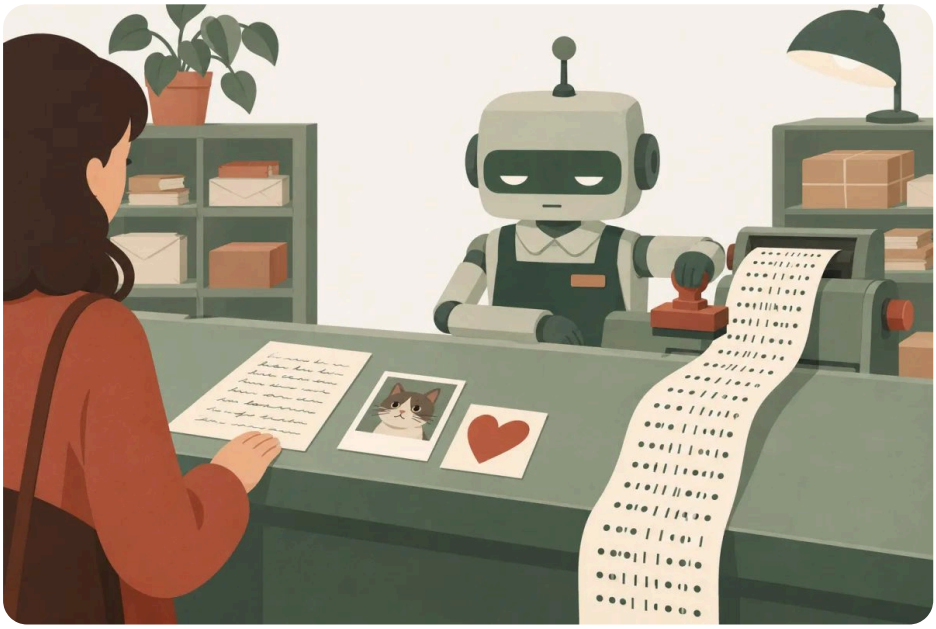


Figure 1 : A poem, a photo, a drawn heart: the counter only hands back numbers. The machine reads nothing else, and every calculation has a price.

1.1 Everything is a number

A heads-up: this is the passage closest to the machine, the most "low-level" one in the book, and that is on purpose. Don't try to memorize the 0s and 1s, aim for intuition: seeing how a computer stores and works on information, from the smallest switch up to the program. This overview will help you understand how the code written on top of it works. The rest of the book rests on this foundation.

A **bit** is a switch: off, it is worth 0, on, it is worth 1. It is the only thing the machine can physically hold. Counting with only two digits is called binary. You group the switches by eight, and a group of eight is called a **byte**.

A padlock with 4 dials of 10 digits offers 10 × 10 × 10 × 10 = 10,000 combinations, from 0000 to 9999. Counting starts at zero: the last one is 9999, not 10,000. A byte is the same padlock with 8 dials of two positions each, which is to say 8 switches: 2 × 2 × … × 2 = 256 combinations, from 0 to 255.

Laid end to end, these bytes make up memory: one huge row of slots, each tagged by a number, its **address**.

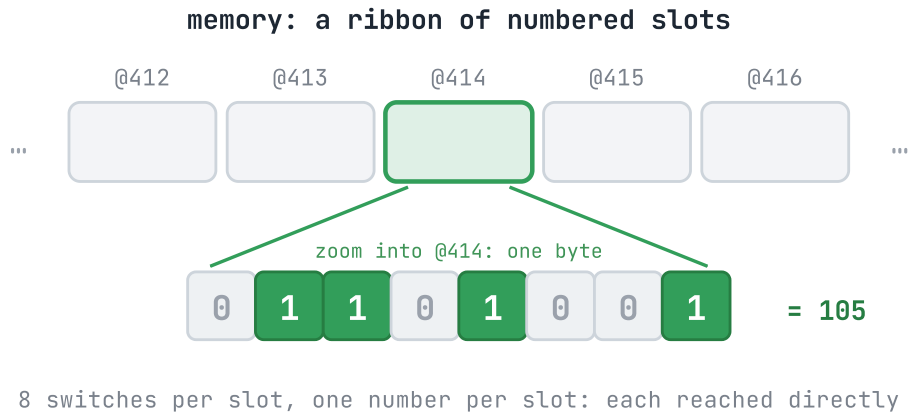


Figure 2 : All of memory fits in this picture. That slot number is the address: chapter 2 will use it to find a piece of data. Why `01101001` is 105: see "Reading a binary number", further down.

Why binary

Why does a bit have only two values, 0 or 1? It does seem primitive. Because **the physical world is noisy**. Noise is the small, unpredictable variations that shake every signal: the level of a signal in a wire drifts with heat, disturbances from nearby devices, the age of the components. Telling ten levels apart (one per decimal digit) would demand an impossible precision. Two states, by contrast, are unmistakable, "off" or "on"; it takes enormous noise to confuse a 0 with a 1.

A DVD illustrates it: its track is just a series of microscopic pits and flat surfaces, and the laser only has to tell whether the light comes straight back or scatters. A QR code does the same with black or white squares that your phone reads, even in poor light.

To write decimal with the same method, you would need to tell ten pit depths apart on the DVD: a scratch or a speck of dust would be enough to mix up a 4 and a 5. On the QR code, you would need to tell ten shades of gray apart: a mere shadow would be enough to mix up two neighbors.

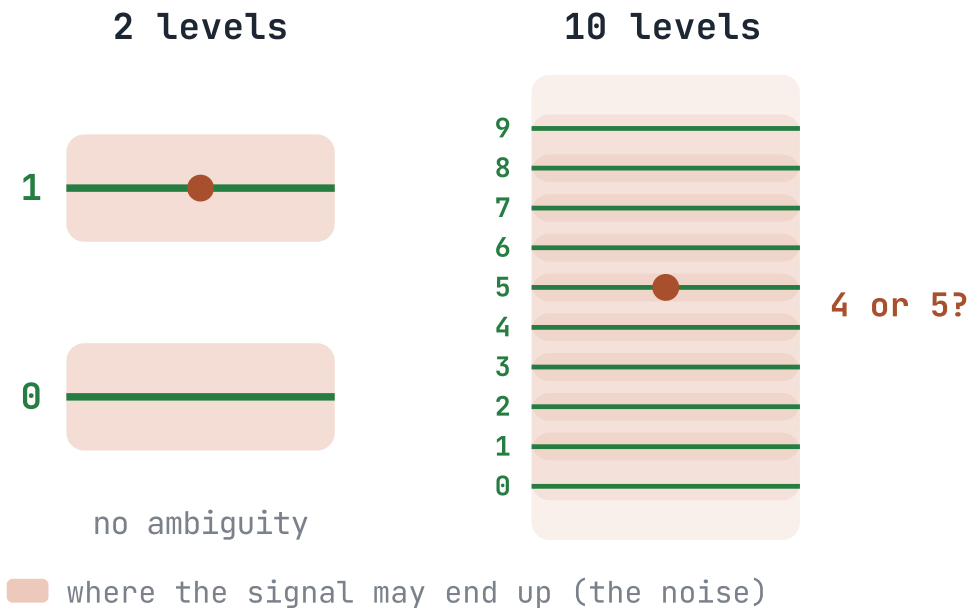


Figure 3 : Noise makes the signal (the dot) tremble: it can end up anywhere in the pink zone. With two levels, the zones stay apart. With ten, they overlap.

So the bit is the smallest choice there is: yes or no. It is also the one that best resists noise, and it lives in the simplest object there is, the switch. With only two states, you can still do everything: true or false to decide, numbers written in 0s and 1s to count. Far from primitive, it is an engineer's choice.

Other paths were actually tried. The ENIAC, one of the very first computers (completed in 1945), counted in decimal, with ten flip-flops, each a pair of electronic tubes, per digit. The Setun, a Soviet computer (completed in 1958, then built in series until 1965), counted in ternary, with three values. Both worked. Binary won out anyway: two states are the simplest thing to build and to make reliable, by the billions.

Some exceptions remain. Recent SSDs, those drives with no moving parts, take the opposite bet: they tell 4, 8 or even 16 levels apart in a single cell, so 2, 3 or 4 bits instead of one, to store more. They pay for that density with faster wear. It is the same rule: more levels, less room for noise.

Reading a binary number

That leaves how a string of 0s and 1s becomes a number. It all comes down to **position**. Start with base 10, the one made of our ten digits. `234` is not "2, 3, 4" stuck together: each column carries a weight, units, tens, hundreds.

Binary follows the very same rule, with two possible digits instead of ten: 0 and 1. Starting from the right, the columns are no longer worth 1, 10, 100, but 1, 2, 4, 8, 16…: each one doubles the one before, these are the **powers of 2**.

Reading a binary number therefore means adding up the columns that carry a 1, skipping those that carry a 0. In `101`, the 4 column and the 1 column carry a 1, the 2 column carries a 0: 4 + 0 + 1 = 5. With all eight columns at 1, you get 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255: the maximum of a byte, just like 9999 for the padlock.

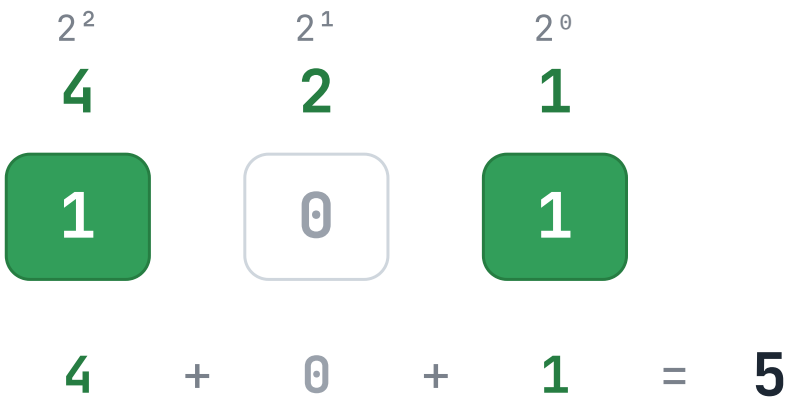


Figure 4 : Each column is a power of 2; add the ones whose bit is set to 1.

Everything becomes 0s and 1s

Now that you can read a binary number, let's see what the machine stores in them. Text, image, sound: everything comes down to numbers, hence to 0s and 1s. A letter fits in one byte: `'A'` is the number 65, by a convention every machine shares, ASCII (American Standard Code for Information Interchange). A word takes a few: "Hi!" takes three.



Figure 5 : The ASCII dictionary: one letter, one number. The H and the i of "Hi!" are in green.

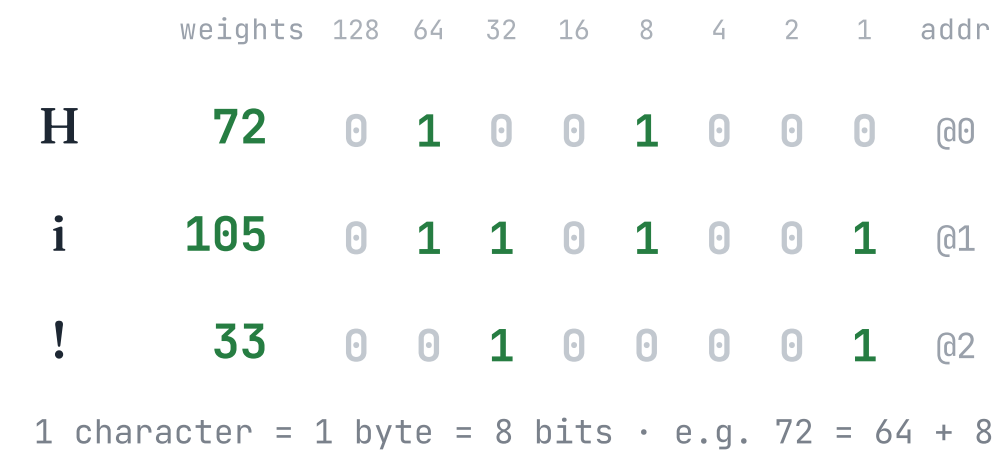


Figure 6 : The word "Hi!" in memory: one letter per byte. Read the number by adding the columns whose bit is 1.

One byte per letter was true for English: a byte offers only 256 values, enough for its alphabet. But the world has over 150,000 characters, from accents to emojis. **UTF-8** is the convention that stores each character on 1 to 4 bytes: `'A'` keeps its single byte, `'é'` takes two, `'🎉'` four. Hence a very real trap: the "length" of a text depends on what you count, bytes or characters. The word `café` has 4 characters, but 5 bytes. Cutting `café` after the fourth byte leaves you half of an `é`.

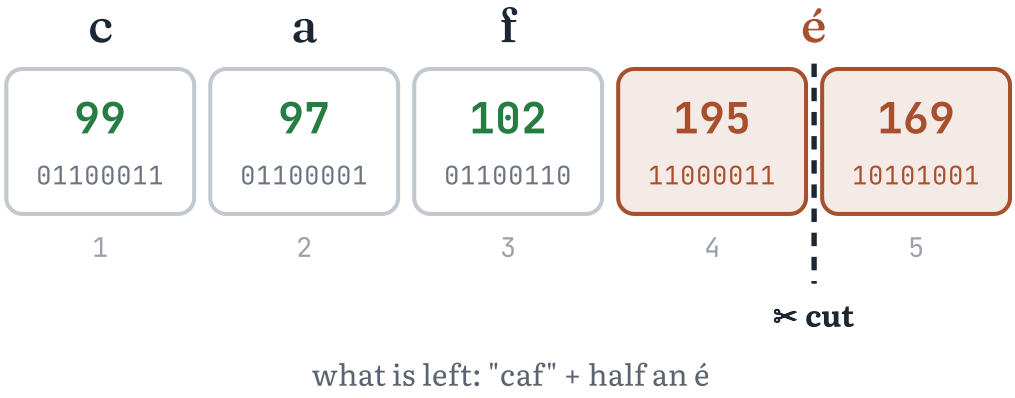


Figure 7 : "café" in bytes: the letters in green fit in one byte, the é (in terracotta) takes two. The dashed line marks the cut.

The same trap exists when reading. If a program reads the two bytes of the `é` (195 and 169) with an old convention, one byte per letter, it sees two symbols, and "café" becomes "cafÃ©". That is the cause of the strange signs that replace accents in a file opened with the wrong format.

On to images: an image is a mosaic of tiny dots, the pixels. A photo has millions of them. Each has its own color, recomposed by mixing red, green and blue, three colors that are enough to make almost all the others. Yellow, for instance, is red and green at full strength, with no blue. So you need three numbers per pixel, one byte each, from 0 to 255.

The simplest case is a black-and-white image. Take a heart drawn on a grid 8 columns wide and 7 rows tall: each square, a pixel, is lit (1) or unlit (0), and each row is one byte. The first, `01100110`, equals 64 + 32 + 4 + 2 = 102. Seven numbers are enough to draw the heart.

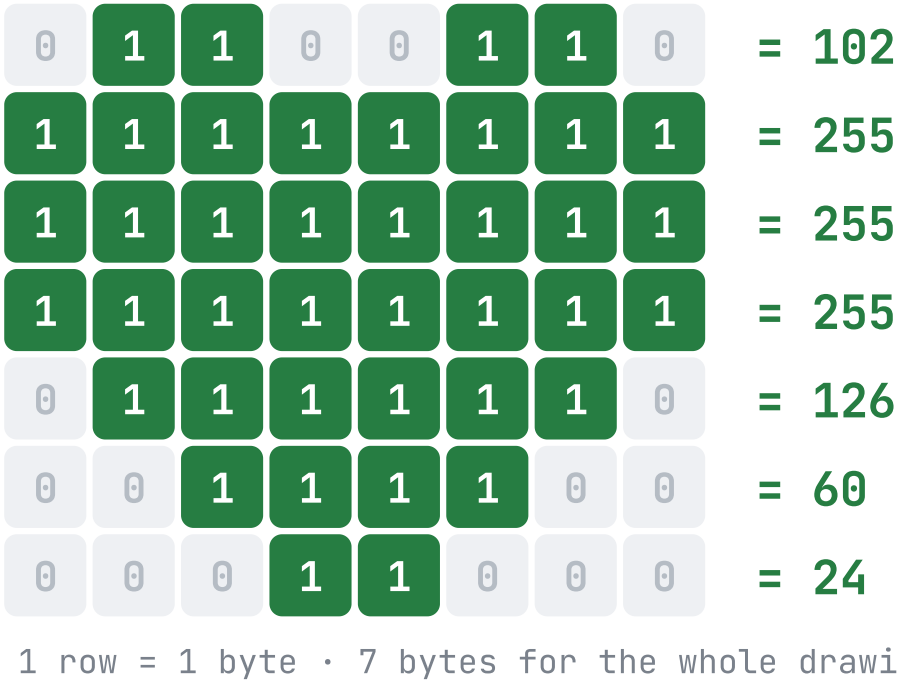


Figure 8 : A black-and-white heart: a lit square is 1, an unlit one is 0. Each row is a byte, hence a number.

Eight digits per byte is long to read. **Hexadecimal** is a way of writing numbers in base 16, where one digit, 0 to 9 then a to f, stands for four bits: two digits per byte.

For example: the terracotta color of this book's figures, `#a84f2d`, is written in the web's color notation. It reads that way: three bytes, `a8`, `4f`, `2d`, that is 168 of red, 79 of green, 45 of blue (the three numbers of each pixel, seen

above). Same column rule as in binary, with 16 in place of 2: `a8` is 10 × 16 + 8 (`a` is worth 10, `f` is worth 15).

Sound is stored as numbers too. A microphone measures the vibration of the air, and its value is noted 44,100 times a second: a song becomes a long list of numbers. A video is just images and sound playing in sequence. Photo, music, film: nothing magical, just an enormous pile of tiny switches, on or off.

A date too. The computer stores it as a number of seconds elapsed since January 1, 1970 at midnight UTC (Greenwich time): `1780963200` reads as "June 9, 2026". The number does not say where you are: the same date reads 2 am in Paris, or 8 pm the day before in New York, depending on the time zone reading it. Hence a golden rule: store a date in UTC, the world reference time, and only convert it to local time when displaying it.

Integers and their limits

Back to the numbers themselves, the ones built from columns of powers of 2. An integer, meaning a number with no decimal point, is an exact sum of powers of 2: it stores without error, as long as its bytes suffice. A byte stops at 255: to go further, you chain several together. Two bytes reach 65,535, four reach over 4 billion (with no negatives), and each extra byte multiplies the capacity by 256.

To store negatives too, the machine sets aside half of the patterns: on one byte, `00000000` to `01111111` mean 0 to 127, and `10000000` to `11111111` mean −128 to −1. So the pattern `11111111` is 255 or −1 depending on how the program reads it. This convention is called two's complement.

But the number of bytes is chosen in advance, and that is where it breaks. Like the padlock from the start, which rolls from 9999 back to 0000, a full byte starts over: 255 + 1 equals 0 (with negatives, 127 + 1 gives −128). An **integer overflow** is a computation whose result no longer fits in the bytes set aside for it: in languages with fixed-size integers, the number then usually wraps around to the bottom of the range, without warning.

In C, Go or Java, an integer has a fixed size, and the trap is real. In 2014, YouTube's view counter, a 32-bit integer (4 bytes), hit its ceiling, 2,147,483,647, on *Gangnam Style*. Half the patterns of 4 bytes go to negatives, hence this ceiling of $2^{31} - 1$. In C, for a signed integer, the result is even undefined: anything can happen. Python, for its part, grows its integers on demand: it has no such limit.

Numbers with a decimal point

After the point, the column rule keeps going. In base 10, you move on to tenths (1/10), hundredths (1/100)... and `0.1` means "1 in the tenths column". The dot does not make the value, **the column does**. In binary, the columns become fractions: 1/2, 1/4, 1/8, 1/16... To be stored without error, a fractional number has to be written as a sum of those fractions: `0.75`, for instance, is 1/2 + 1/4, or `0.11` in binary.

`0.1`, however, cannot be written that way, and that is the root of a famous bug: in almost every language, `0.1 + 0.2` does not equal `0.3`. A comparison helps. In base 10, 1/3 is written `0.333…` without ever ending. In binary, it is 1/10 that never ends: `0.0001100110011…` A fraction lands exactly in binary only if, once simplified, the denominator (the bottom number) is a power of 2, like 2, 4 or 8. But 10 is not one: it hides a factor of 5, which multiplying 2 by itself never produces.

So how does a machine store such numbers? By moving the point. Stay in base 10 to see the trick: `1234.5` can be written 1.2345×10^{3}, and `0.00012345` becomes 1.2345×10^{-4}. This way of writing has a name, scientific notation: it is the one on your calculator. The same digits in both cases, plus an exponent saying how many places the point moved.

In binary, the machine stores the digits on one side, called the **mantissa**, and on the other the **exponent**, the power of 2 to multiply them by. A **float** is a number stored as a mantissa and an exponent: the point has no fixed place, the exponent slides it, hence the name "floating point". With the same number of digits, you hold both the huge and the tiny.



Figure 9 : A floating-point number in memory: 64 switches in three roles. One for the sign, eleven for the exponent, fifty-two for the digits. That is where the room runs out and the pattern gets cut (here, `0.1`'s).

▾ The exact bits of 0.1, for the curious (optional)

For `0.1`, let's read the three pieces. The **sign** first: 0, because `0.1` is positive (1 = negative). The **mantissa** next: take `0.0001100110011…` and slide the point to just after the first 1, which gives `1.100110011… × 2⁻⁴`, whose digits repeat **1001** forever. A number normalized this way always starts with "1.", so that leading 1 is never stored: a free bit.

The **exponent** last, −4. Here, the 11 bits that store it do not use the two's complement seen earlier: they read as a number from 0 to 2047, but −4 is negative. The fix: add a fixed offset of 1023, which always makes it positive. So the machine stores −4 + 1023 = 1019, written in binary as `01111111011` (and subtracts 1023 when reading to get −4 back).

The float trap

Back to `0.1`, whose binary pattern never ends. The room kept for digits, by contrast, is **finite**: about fifty bits. So we cut the endless pattern, and the last digit kept gets **rounded**. That small rounding leftover, let us call it the **crumb**:

stored `0.1` is a little more than 0.1. `0.2` suffers the same, the crumbs pile up, and the total misses `0.3`:

```
// in JavaScript (=== tests equality, same results in Python and PHP):
// what the machine actually stores (the "crumb"):
0.1        → 0.1000000000000000055…   // a bit too big
0.2        → 0.2000000000000000111…   // a bit too big
0.3        → 0.2999999999999999888…   // a bit too small

0.1 + 0.2  → 0.3000000000000000444…   // overshoots stored 0.3
0.1 + 0.2 === 0.3   // false → 0.30000000000000004
0.1 + 0.3 === 0.4   // true… but by luck!
0.1 + 0.7 === 0.8   // false, just like 0.1 + 0.2
```

That is the whole trap: **0.1 and 0.2 lean a touch too high, 0.3 a touch too low**. Their sum overshoots the stored 0.3. The sum and the stored 0.3 are simply not the same number, so the equality is false.

And this miss is nothing special. The crumbs sometimes add up, sometimes cancel out: comparing two floats with an equality test is therefore a **lottery**, as the last two lines of the code above show.

To remember: an integer is stored without error as long as it fits in its bytes. A number with a decimal point is exact only if it is a sum of 1/2, 1/4, 1/8… short enough to fit in the float's digits, like 0.75 (1/2 + 1/4). Hence two reflexes.

Never compare two floats with an equality test: check instead that they are close enough, relative to their size (in Python, `math.isclose(a, b)`). And store money **in cents**, as integers: €19.99 becomes 1999. An exact decimal type, such as Python's `Decimal` or Java's `BigDecimal`, works too. The math becomes reliable again, and a whole class of rounding bugs disappears.

The program is a number too

One last example, the most unsettling: your program itself is made only of numbers. A program is a list of **instructions**, tiny orders the processor, the computing chip, knows how to carry out: "add these two numbers", "copy this cell elsewhere", "jump to number 40 if the result is zero".

These orders are stored one behind the other, each at its own number, like the cells of memory. The chip reads the first, runs it, moves on to the next, and so on. A jump only tells it "carry on at number 40 instead". Going back means repeating. Jumping ahead means skipping a piece, for instance when the result is zero.

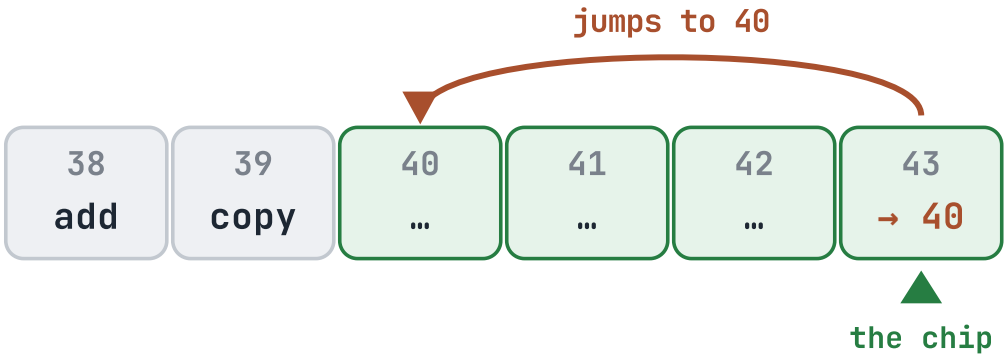


Figure 10 : The chip reads the orders one by one, each at its own number. At 43, a jump takes it back to 40: it repeats.

Orders act on **data**, meaning numbers such as a price, a letter or a pixel. Yet orders and data are both numbers, stored in the same memory: nothing in the number itself says which is which. It all depends on the reading: 65 can be read as the letter "A", as a price or as an order. That ambiguity has a cost: in 7.6, an attacker will exploit it to slip an order where a piece of data was expected.

Modern chips add a lock: every area of memory is marked "orders" or "data", and the chip refuses to run the latter. But that lock only sees memory cells. The attacker hides the order inside a text (a name typed in a form, say) that your program copies as is into the middle of a command: no mark separates data from order there. It is the same confusion, one notch higher.

↳ Write Great Code, vol. 1 · Eloquent JavaScript

1.2 Memory is a pyramid

So far, memory was just a uniform row of bytes. In reality it has **layers**, stacked like a pyramid: small and fast at the top, huge and slow at the bottom. Why? Because no memory is at once big, fast and cheap: the fast kind costs a lot and stays small. And the processor, or **CPU** (the chip that runs your

instructions one by one) only computes fast on what sits right next to it. The **registers** are a few slots inside the CPU, the closest to it: the only place it actually computes.

Everything else lives further away, in layers: the **cache** (a fast reserve glued to the chip, keeping a copy of recent data), then **RAM** (the working memory, the memory sticks in your machine), then **disk**. The cache itself splits into three tiers, L1, L2 and L3, from nearest to the chip to farthest. To be used, a piece of data must therefore first climb to a register, layer by layer. Each layer you go down is bigger, but roughly ten times slower: the trip takes that much longer.

Between RAM and the hard disk, however, it is no longer one more step, it is a chasm: ×100,000. An SSD closes a good part of it, but is still about a thousand times slower than RAM.

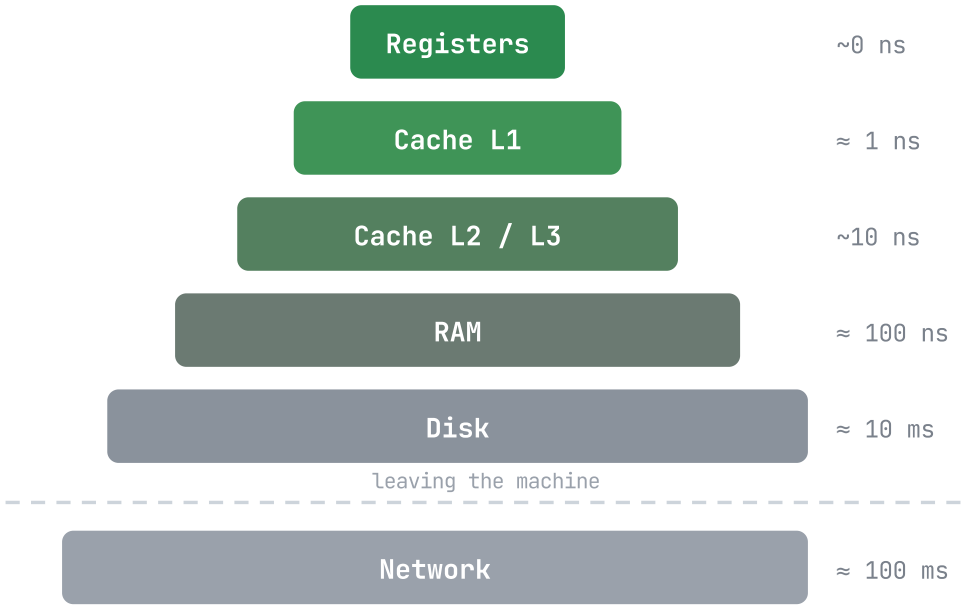


Figure 11 : The memory pyramid: at the top, small and near-instant; at the bottom, huge and slow. Below the dashed line, you have left the machine.

The pyramid does not stop at the machine's edge either: below the disk sits the network, another ten times slower when the data crosses an ocean. That layer gets its own chapter (chapter 5).



Figure 12 : The machine as a city: the further data lives from the town hall (the CPU), the longer the courier walks.

The orders of magnitude are dizzying. A cache read takes about 1 ns, one billionth of a second. Scale it up to **1 second**: RAM would then answer in **~1 min 40**, and the hard disk in **~4 months**. A value you read often must therefore stay **as high as possible**.

Reading neighboring cells

The machine never fetches a single byte: like a courier filling a crate, it pulls a whole **cache line** (about 64 bytes) at once. Hence another reflex: **reading neighboring cells is almost free**; jumping all over is costly.

These two reflexes come from one and the same tendency. **Locality** is the tendency of a program to stay close to what it has just read. It takes two forms:

- You soon read again what you have just read (in time).
- You then read the cells next to the one you have just read (in space).

The cache bets on both: recent data stays at the top of the pyramid, its neighbors arrive in the same crate. Honor the bet and you run fast: scan cells in order rather than hopping from address to address, keep together what is used together. Break it and you pay for every jump.

Defragmenting a hard drive applies the same idea. A file is often scattered in pieces, and the read head, the mechanical arm that reads the disk, must jump from one piece to the next. Defragmenting glues the pieces back side by side. An SSD, an electronic drive with no moving parts, has no jump to avoid: defragmenting no longer helps there.

The classic case of locality is the **two-dimensional array**: a grid of cells, in rows and columns. You meet one every day without naming it: a spreadsheet, a chessboard, or the pixel heart of 1.1, which is nothing but a grid of pixels.

Memory, though, stays one long row of bytes, with no rows and no columns. The grid is therefore **flattened** into it: the first row, then the second laid right behind, and so on. Asking for the cell at row `i`, column `j` (what languages

write `a[i][j]`) is not a double lookup: it is computing a single position, `i × width + j`. In a grid 3 columns wide, the cell at row 2, column 1 sits at position 7 (2 × 3 + 1), counting from 0.

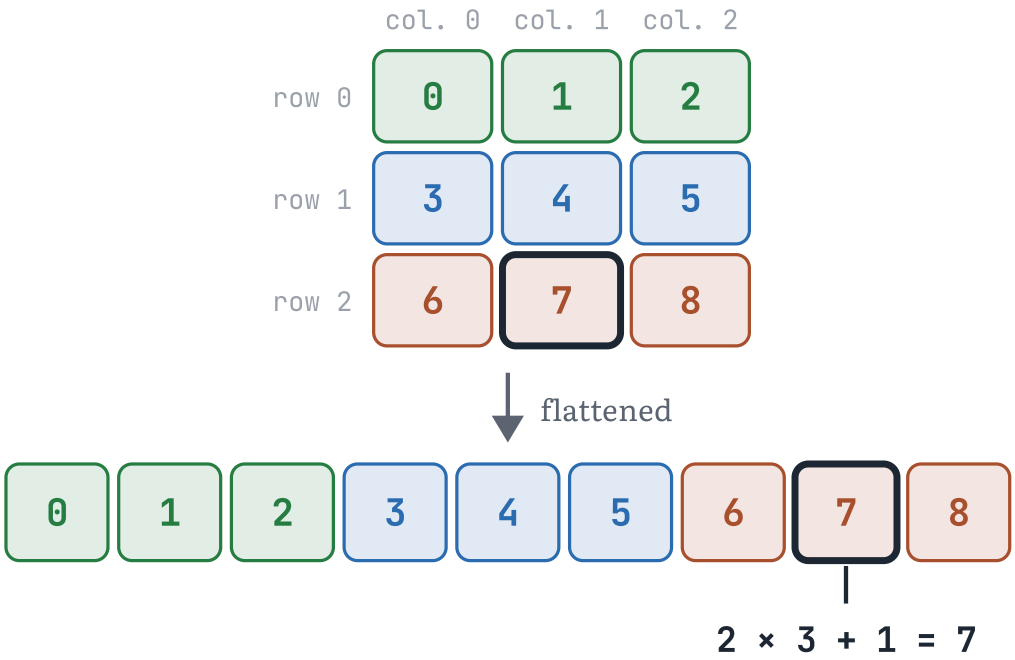


Figure 13 : A grid 3 columns wide unrolled into one row: each row keeps its color. The cell at row 2, column 1 (circled) lands at position 7.

From there, two traversals with opposite costs. Traversing a **row** means stepping from one neighboring cell to the next: each is already in the cache's 64 bytes, so free. Traversing a **column** means jumping a whole grid width each step: you land on a new cache line at every step, and once the grid is wide, the previous line is pushed out of the cache before it is used again. That is slow.

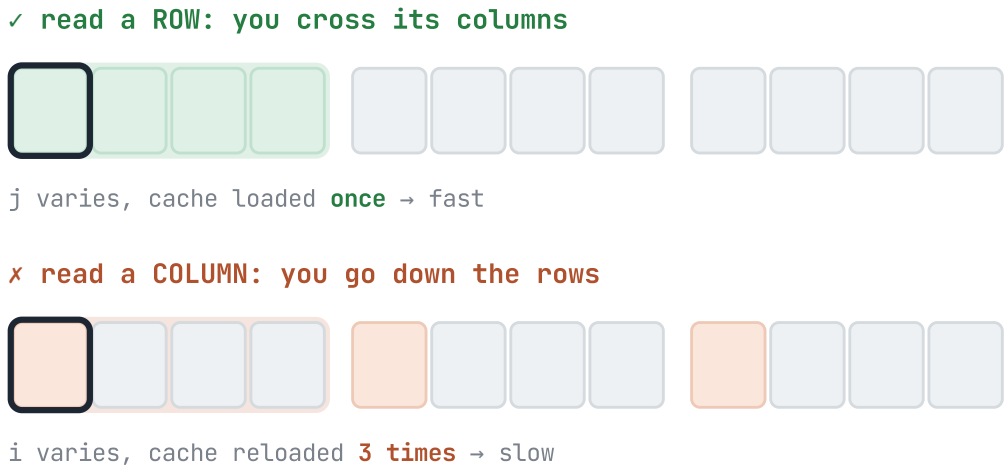


Figure 14 : The black frame is the reader, the colored zone the loaded cache line.

Same array, same work: only the order of the two **loops** changes, blocks of code you repeat, one for each row, the other for each column:

```
// same logic, two loop orders over a 2D array
for (i...) for (j...) a[i][j]
// ✓ we move along a row: the cache line is reused
for (j...) for (i...) a[i][j]
// ✗ we step down a row each time: often several times slower, sometimes
// dozens
```

This gain assumes **cells of numbers side by side**, one row after another. That is the case in C, in Go with a `[3][3]int` array, in NumPy by default and, row by row, in Java with `int[][]`. Fortran, Julia or MATLAB store by columns, and the winning order flips there. The gain disappears when each cell only holds the address of a piece of data stored elsewhere, as in a Python list or a Java `Integer[]`.

The trap in everyday code

The same trap springs every day in ordinary code, in the shape of an innocent-looking "**load it all into memory**". Pulling 100,000 rows from a **database** (the software that stores data and finds it on demand) to display only ten means moving 10,000 times too much data: it crosses the network, is decoded one row at a time, fills RAM, and almost all of that work is thrown away. The cost comes from the moving, as with the column traversal. **Moving data costs more than computing.** The reflex: ask the database for only the ten rows you need (`LIMIT 10`).

↳ Write Great Code, vol. 1 · vol. 2

1.3 Cost has a unit: Big O

Writing code that runs is good; knowing whether it holds when the database goes from a hundred rows to ten million is better. We saw the price of one access; what remains is counting how many we make. Imagine looking up a name in an unsorted notebook: you read the names one by one. With 1,000 names, that is 1,000 steps; with ten million, ten million steps: the work follows the size.

Big O is the unit that says how the number of steps grows with the input size, written `n` (here, the number of names). It **ignores the constants**: it does not matter whether your machine is slow or fast, only the way it grows counts. For the notebook, it is written `O(n)`, the `O` meaning "on the order of": ten times more data means ten times more work. It is not a duration in seconds, it is the shape of a curve. Four families show up everywhere:

- **O(1): constant** time, whatever the size (the fastest);
- **O(log n): logarithmic** time, climbing ever more slowly: you halve the problem at each step;
- **O(n): linear** time, twice the data means twice the work;
- **O(n^2): quadratic** time, twice the data means four times the work (the worst here).

The word "exponential" is often misapplied to `O(n²)`. True exponential time, `O(2ⁿ)`, is far worse: every element added **doubles** the work, and forty elements already pass a thousand billion operations.

A shape is not a speed: an `O(1)` can still be slow if each step is a hard-disk read, a hundred thousand times costlier than a RAM read.

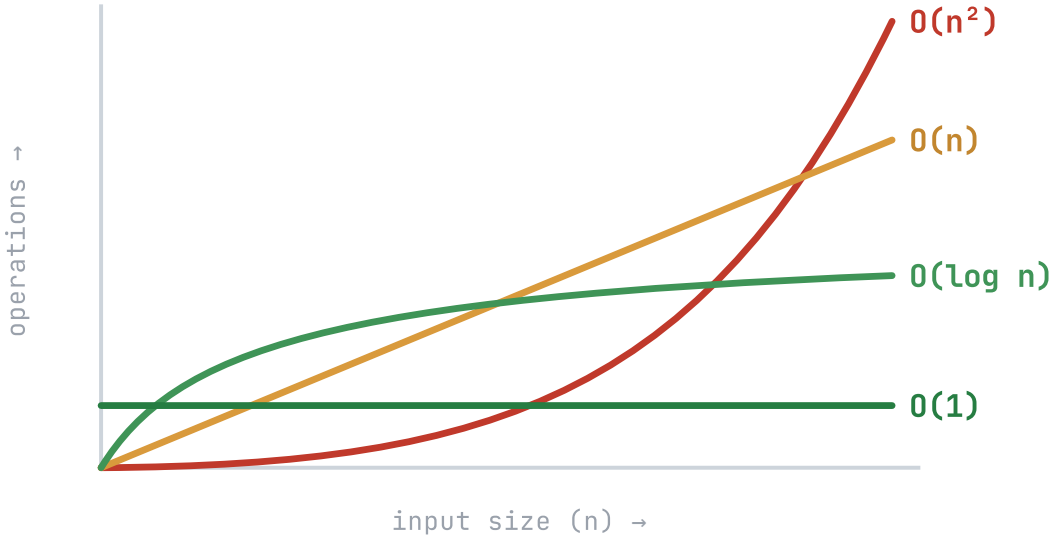


Figure 15 : As the input grows, the family decides everything: O(n²) explodes where O(1) does not move.

Three strategies for one search

The word logarithm is scary, the idea is not. `log n` is the number of times you can halve `n` before landing on 1. A million halved twenty times in a row reaches 1, so `log` of a million is about 20. Take the notebook again, but with the names in order, as in a phone directory: you open it in the middle, keep the right half, and repeat. Same job, other strategy, other family: a directory of one billion names is combed through in about thirty steps. The curve flattens: doubling the data adds just one more step, and a thousand times more names add only ten.

A third strategy for the same notebook: stop searching altogether. That gives O(1), one of the most profitable ideas in everyday code. A **hash table** stores each **value** under a **key**, a customer's record under their name for instance: to

find it, you give the name and get the record at once, in constant time (on average), even over a million entries. You already use one: Python's `dict`, JavaScript's object, PHP's associative `array`.

The mechanism is simple. The key goes through a **hash function** that turns it into a cell number: you search for nothing, you compute where the value sits and go straight there, whatever the number of entries.

Concretely, the table is a big array of numbered cells. Take the key "Alice": the hash function turns it into, say, the number 7, and the record is stored in cell 7. To find it again, you redo the same calculation and read cell 7, without scanning anything.

Two keys can produce the same number: that is a **collision**. The two records then share the cell or its neighbors, and the keys are compared until the right one turns up. As long as the table is not too full and the function spreads the keys well, collisions stay rare: that is why O(1) holds "on average". When the table fills up, it is copied into a bigger array, which is costly but rare. In exchange for this speed, the table stores keys in no particular order: to get "all the names from A to C", you have to scan everything, whereas the sorted list hands them over in one block.

Same question, three strategies, over a million names (in Python, for the example):

```
name in list
// O(n): read the names one by one, up to 1,000,000 steps
bisect_left(sorted_list, name)
// O(log n): open in the middle, about twenty steps, returns the position
name in dictionary
// O(1): compute where it sits, a single step
```

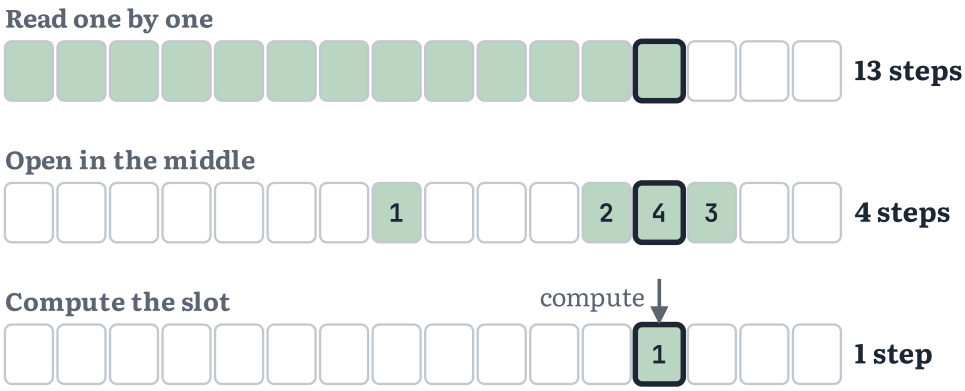


Figure 16 : The same name searched among 16: green cells are the ones consulted, the circle is the name found. With a million names: up to 1,000,000 steps, 20, 1.

One detail hides a cost: the directory in the second strategy was already in order. Who put it there? An efficient sort takes about n × log n steps: that is the **O(n log n)** family, between the straight line of O(n) and the blow-up of $O(n^2)$. For a million names, that makes twenty million steps, or twenty searches that read everything. Sorting only pays off if you then search often: after about twenty searches, it has paid for itself. Pay dearly to prepare, pay little to search: you will make that trade again and again, all the way to a database index, which is just a directory kept up to date to answer fast.

Estimate before coding

These gaps between families are not theoretical. A computer from the 1970s, thousands of times slower than yours, beats a modern, blazing-fast machine as soon as the data grows, provided the old one runs `O(n)` and the new one `O(n³)`. An **algorithm** is a method of computing: a series of steps that solves a problem. **No amount of computing power makes up for a bad algorithm.**

Hence a habit that pays off: before writing a line, roughly estimate the number of operations. On ten million items, a single pass (`O(n)`) is ten million operations, a fraction of a second. Comparing them all pairwise (`O(n²)`) is ten million × ten million = **a hundred trillion**, hours of compute. The gap is **ten million times more work**: that thirty-second estimate tells you which one is viable before you even code.

To turn these operation counts into seconds, remember a single equivalence: a billion simple operations take about one second. The rest follows, since each time the number of operations is multiplied by a thousand, the duration is too. A million operations take a thousandth of a second, ten million a hundredth. A hundred trillion take a hundred thousand seconds, close to thirty hours. The rule is rough (it assumes data that fits in the cache, and in Python you should count ten to a hundred times more), and that is its strength: it sticks, and it separates the feasible from the impossible.

Choosing a data structure

Big O also helps choose a **data structure**, the way a collection is laid out in memory. An **array** puts its elements side by side, like numbered lockers: reaching the 500th is instant, you compute its address and go, but inserting in the middle forces you to **shift every following element** by one slot. Each one is copied a notch further along, and copying takes time: that is the price of moving data.

A **linked list** works like a treasure hunt: each element tells you where to find the next. Inserting only **re-wires two links**, copying nothing, but to reach the 500th you must **follow the trail from the start**. In Big O, it is a mirror: the ar-

ray reads in `O(1)` and inserts in the middle in `O(n)`, the linked list reads in `O(n)` and inserts in `O(1)` once it has reached the right spot. In practice, the array almost always wins: its neighboring cells fit in the cache (1.2).

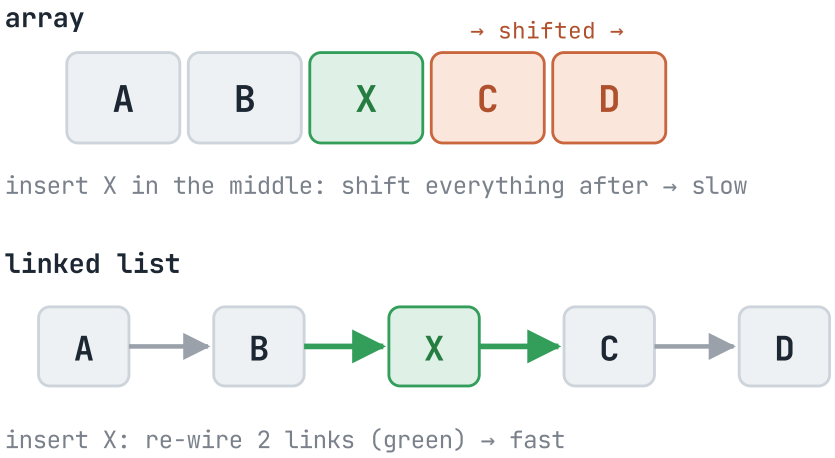


Figure 17 : Inserting in the middle: the array shifts everything after, the list re-wires two links (in green).

No strategy is "the best" in the absolute, neither among algorithms nor among structures: you choose according to the problem. Before choosing, ask how much data you will have, what the program will do most often (read, insert, search) and what must stay fast. This thinking needs no code, as long as you know how the machine stores data, moves it and counts its steps. Once the approach is sized up, what remains is writing it cleanly.

↳ Grokking Algorithms · Programming Pearls

1.4 Think like the machine, write high-level

These days we write in clear languages, called high-level (PHP, JavaScript, Python...), far from the machine's raw language, the low level. The chip, in fact, only understands its own list of instructions, those tiny orders written as numbers, seen in 1.1: "add", "copy", "jump to 40". A single line like `total = price * quantity` becomes several of these orders: read the price, read the quantity, multiply them, store the result. Writing everything at that level would be endless, and would have to be redone for each family of chips, each with its own list. Hence a translator between the two.

A program's path

Here is a program's path, from writing to running. It starts with text you type, with three words to know:

- A **code editor** is software for typing text, like a word processor but made for code (VS Code, for example).
- A **file** is like a sheet of paper kept on the disk, with a name. The typed text is saved in it, `checkout.py` say. The ending ".py" says it is Python.
- The **source code** is the text of the program, as the human wrote it. The machine cannot understand it yet: it must be translated first.

Running the program means handing `checkout.py` to a translator: software that turns the text into instructions the chip understands. You go through the terminal for that.

- A **terminal** is a window where you type your requests to the machine, one line at a time.
- The line `python checkout.py` tells Python's translator: "read `checkout.py` and do what it says".

Editing the file changes nothing until you save it and run it again.

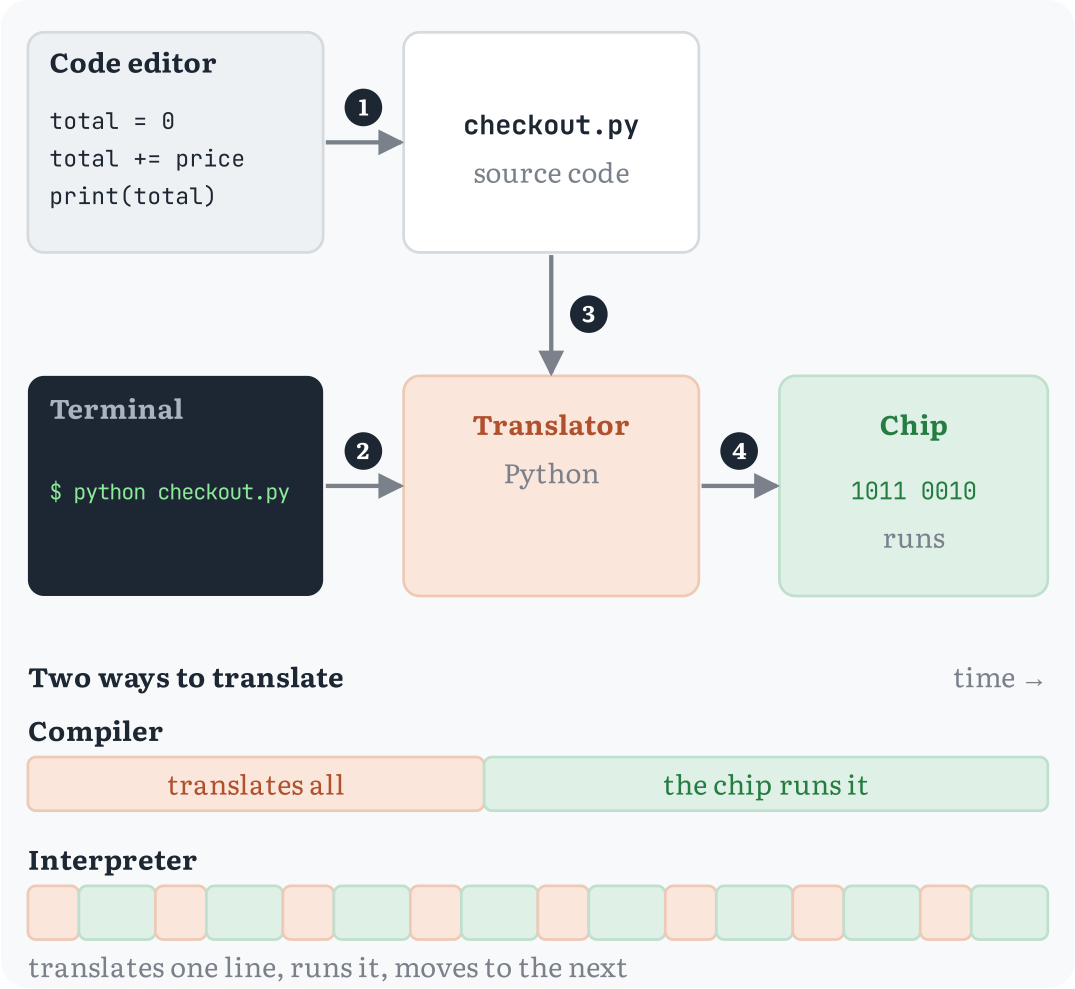


Figure 18 : A program's path: ① you type and save, ② you launch it in the terminal, ③ the text of the file arrives at the translator, ④ it hands its instructions to the chip. Below, the compiler translates everything before running, the interpreter alternates the two.

Compiler or interpreter

There are two ways to translate:

- The **compiler** translates all the text of the program before it runs, like a book translated in full before it is published. Then the chip runs the result.
- The **interpreter** translates while the program runs, like a conference interpreter: sentence after sentence.

The line is blurry, because most languages mix the two. Python, PHP and Java first translate everything into an intermediate form, easier to run. Java and JavaScript then translate the most used parts again into chip instructions,

during the run. "Compiled" or "interpreted" therefore says how a language is run, not what it is.

Three reasons to know the machine

The lessons that follow hold for both ways of translating. Knowing this translator a little, and the machine it targets, still pays off, for three reasons.

First, knowing it helps you **smell a hidden cost** behind a plain-looking line. Here is a loop in JavaScript, a block of code that repeats, which recounts the words of a text on every turn, even though the text never changes. The machine cannot always tell: for n words, the loop makes n turns, and each turn rereads all n words, which makes n × n steps. That is the `O(n²)` family. The fix fits in one line: count once, before the loop.

```
for (let i = 0; i < wordCount(text); i++) { … }
// ✗ recounted on EVERY pass
const n = wordCount(text);                          // ✓ counted once
for (let i = 0; i < n; i++) { … }
```

In this code, `for` repeats the block as long as the condition `i < n` stays true, `i` gaining 1 on each pass. `let` and `const` create a named cell. `wordCount(text)` gives the number of words. Chapter 2 covers these moves in detail.

Second, you learn what the translator does for you, and what it leaves to you. It optimizes small details on its own: it computes `3 × 4` once and for all, say, instead of redoing it every run. But the translator's optimizations stay **local**: it will not turn a slow search into a fast one. **The algorithm stays your job; the machine only polishes what you hand it.**

Finally, the low level holds surprises. To go fast, the CPU **bets** ahead of time on the result of every test, those "if this condition, then..." written `if` in the code. This is branch prediction. The CPU starts working without waiting for the real answer: if the bet is good, nothing is lost, and if it is bad, that head start is thrown away.

This gives an absurd but real result. To count the numbers above 128 in an array, the CPU asks the same question at every cell. If the array is sorted, the answer is false, false, false..., then true, true, true...: the bet "same answer as last time" is almost always good. Shuffled, the answer changes at random and the bet is lost about one time in two: the same scan can be several times slower. This case became famous on Stack Overflow, the developers' help site. But it all depends on the compiler: some turn the test into a plain calculation, with no bet possible, and the gap vanishes.

Measure, don’t guess

No one would ever guess that, and there are many effects like it: on performance, intuition is often wrong. The only reliable way to know whether code is fast is therefore to **measure** it, actually time it. You estimate before coding, we measure after.

Measuring has its tool and its trap. The **profiler** is a program that watches yours while it runs and says where the time goes, by function (a named chunk of code) or by line. Its verdict almost always surprises: a few lines eat most of it, rarely the ones you suspected.

That leaves the classic trap: the **microbenchmark**. It is a stopwatch placed on a single line of code, cut off from the rest of the program. The time it reads is often wrong, for three reasons:

- The compiler sees that the result of the line is never used, and removes it. You are then timing nothing.
- The cache keeps what the line has just read. From the second run on, everything is within reach (1.2), whereas in the real program the data comes from far away.
- Java and JavaScript translate the most used parts again during the run, but only after watching them run for a while. At the start of the measurement, that work is not done yet.

The fix is simple: measure the real program, with real data, and more than once.

↳ Write Great Code, vol. 2

↳ WHICH LEADS TO CHAPTER 2

The machine speaks only numbers and stores its data in layers. Every step has a price: you can now estimate it before coding, and you know it must be measured after. A translator does the rest, but it only reads code, and someone has to write it. What remains is choosing the words in which you express your intent: the **programming language**. *It is not only for giving orders, it becomes a tool to think: its words decide what you are able to conceive.*

"The book of nature is written in mathematical language." — Galileo, *Il Saggiatore*, 1623

2 Language, a tool for thinking

expressing

You know how the machine stores its numbers and what each access costs. Now you have to learn to talk to it, and the words we use matter: a language makes some ideas simple, others painful. Adding up prices takes one word in the Python language, `sum(prices)`, while in the C language you must write the repetition yourself: take a price, add it to the total, move on to the next. One idea runs under the whole chapter: naming a thing widens what you can think.



Figure 19 : Until a thing has a name, you cannot handle it. Once named, you can file it, compare it, combine it.

Two words keep coming back, and each names a thing, so let us define them now. A **variable** is a name stuck on a slot in memory, like a label on a locker. From now on we will say locker. The name is tied to the locker's address (1.1), and the locker holds a piece of data, 0s and 1s that stand for a number, a price, a letter. You store the value, find it again by its name, and can replace it: the name stays, the content changes, hence "variable".

A **function** is a recipe written once and for all: you hand it ingredients, its **arguments**, it hands back a dish, its **result**. The function `subtotal(price, quantity)` carries its arguments in parentheses: it takes a price and a quantity, and gives back what those items cost together, €90 for two sweaters at €45. You call it (you launch it) as often as you like, it does the same work every time.

The function also lives in memory: its instructions are stored somewhere, and its name is a label stuck on the address of the first one. Calling it means telling the chip to jump to that address, then, once the work is done, to come back to the instruction that follows the call. Naming a recipe thus means being able to call it back with a single word.

2.1 The three moves: chain, repeat, choose

A program is a sequence of instructions stored in memory. The chip runs them one after the other, in the order of their numbers. With variables to hold the values, three moves are enough to write any program: chain, repeat, choose. That is not an opinion but a theorem, proved in 1966 by Corrado Böhm and Giuseppe Jacopini.

That theorem changed habits. In the 1960s, languages offered `goto`, "go to line N", a jump to any place in the program. In 1968, Edsger Dijkstra, in an article titled *Go To Statement Considered Harmful*, argued that such a program becomes impossible to follow: you no longer know which way the chip came.

The theorem therefore proved that you can do without `goto`. Without it, a program reads from top to bottom, like a book. Each loop and each choice has a clearly marked beginning and end: you see where you go in and where you come out.

The chip, for its part, still jumps from one instruction to another. But the language places those jumps for you, cleanly. Every mainstream language offers these three moves, only the way they are written changes.

Think of a shop checkout. Each item goes past the barcode reader, and two identical sweaters make a single beep, followed by "× 2". The total climbs with each beep. At payment time, the till takes off 10% if the total tops €100.

Pseudo-code describes a calculation in sentences close to plain English: you then rewrite it in the language of your choice, and that is the good habit, think first, translate after. Here is the checkout in pseudo-code, the `cart_total` function, then, in the drawing below, the chip's path for two sweaters at €45 and a book at €15. The `cart` there is the sequence of scanned items, `item.price` reads "the price of this item" and `item.quantity` "how many there are".

```
function cart_total(cart) :
    total = 0
    // chain: one line, then the next
    for each item in cart :
    // repeat: a loop, one round per beep
        total = total + item.price × item.quantity
    if total > 100 :                          // choose: yes or no
        total = total × 0.9                   // 10% off
    return total
```

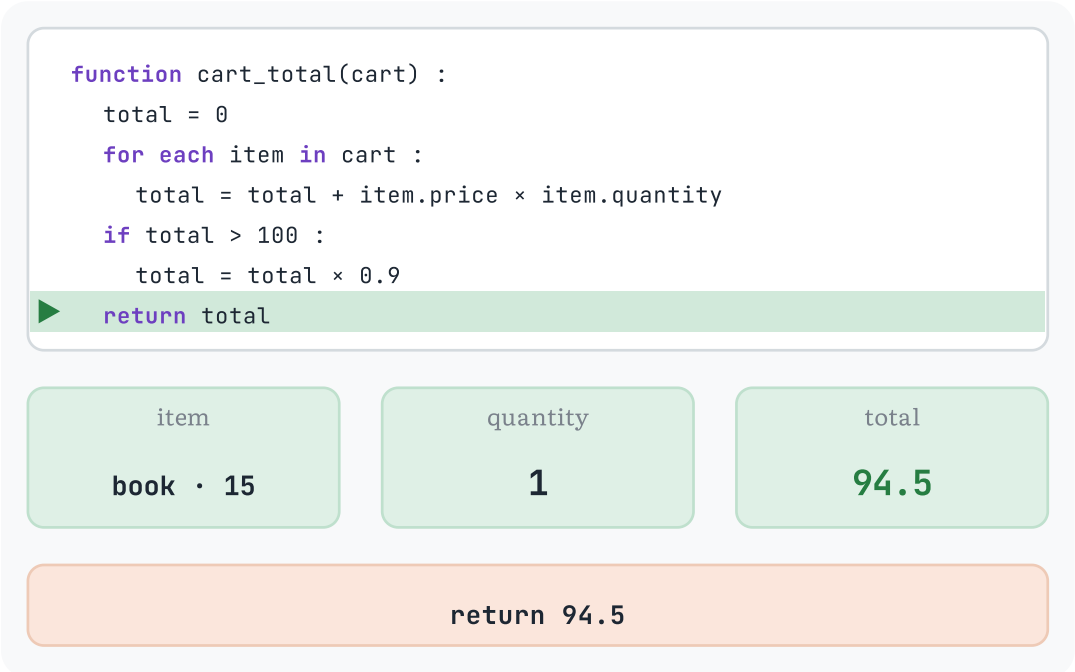


Figure 20 : The chip plays the program one line at a time. The green band marks the current line, the bottom area says what it has just done, and the three lockers show their content. Note that the band goes back up to "for each" after every round: that is the jump the language hides. Interactive version on the website.

Let us read this code the way the chip would. It starts from the first instruction, `total = 0` : it finds the locker named `total` and writes 0 into it, which replaces whatever the locker held before. The sign `=` therefore does

not mean "equals": it means "put into". The chip always works out the right side first, then stores the result in the locker named on the left. So `total = 0` reads "put 0 into the locker `total`".

To **chain** is to run the instructions from top to bottom, in the order they are written: each one starts when the previous one is done. Order matters: put the "if" before the loop, and the chip asks whether the total exceeds 100 while it is still 0. The discount never fires, and the customer at €105 pays full price.

Next line, `for each`. We **repeat** with a **loop**: a piece of program that the chip redoes once per element to process. Here, it makes one round per beep. Each round takes the next item, reads the `total` locker, adds price times quantity, and writes the result back into the same locker. That is why `total = total + …` makes sense. Then the chip climbs back to the `for each` line: that is the jump back of 1.1. The `sum(prices)` of the introduction hides exactly this loop.

After the sweaters, the locker holds 90. After the book, 105, that is `01101001`, the same pattern of 0s and 1s as the letter "i" of 1.1. The locker does not know what it keeps: the program decides whether to read it as a price or as a letter. Two beeps, two rounds, then the chip moves on to the line below. A thousand beeps, a thousand rounds: the cost grows with the cart, that is the `O(n)`.

The discount is left. We **choose** with an "if": the chip asks a question with a yes or no answer, then runs the piece of program that follows or skips it. It reads the `total` locker and compares its content to 100. It holds 105, the answer is yes, the line `total = total × 0.9` runs and rewrites the locker, which now holds 94.5. With a single sweater, the answer would have been no, and the chip would have skipped that line, straight to `return total`: that is the jump forward.

This hand-run, where you play the chip one instruction at a time, is the best way to understand a program you are reading, and the first reflex when a result surprises you.

Where the cart comes from, where the total goes

The cart was not born inside the program, and 94.5 must not stay there. At the till, items come in through the barcode reader, and the total goes out to the screen and the receipt printer. A program is the same. The **input** of a program is everything it receives from outside: a key pressed on the keyboard, a file on the disk, a barcode, a message from another machine over the network (chapter 5). The **output** of a program is everything it sends outside: text on the screen, a file written, a message over the network. In between, the calculation, which touches nothing but lockers.

```
cart = read_barcodes()        // input: the data comes from outside
total = cart_total(cart)      // calculation: nothing but lockers
show(total)                   // output: the result goes out to the screen
```

Every language has its own words for input and output. Here are Python's:

- `print(total)` plays the role of our `show`. The word comes from "print out", yet it writes to the screen, in the terminal.
- `input()` waits for a line typed on the keyboard, then hands it back. It comes back as text: for Python, "45" is then two characters, not the number forty-five. So you must convert it to a number before calculating.

Input, calculation, output: it is the "numbers go in, numbers come out" from the opening of chapter 1, seen from the program's side.

Do not mix up returning and showing: `return total` hands the result back to the code that called the function, which can reuse it. `show(total)` sends it out of the program: the rest of the calculation can no longer get it back. When a result is wrong, look first at what went in: a correct calculation on wrong data gives a wrong result, and the machine does not complain.

Where a variable lives, and how far

The `total` locker served for the length of one calculation. How long does it live, and who can touch it? In `cart_total`, the variable `total` is born when the function starts and vanishes when it returns its result: each call re-

serves its own lockers of memory and frees them on the way out. A variable's **scope** is the part of the program where its name is visible. Another function, `order_total` for instance, can therefore have its own `total`: same label, different locker, like two locker rooms that each have a locker marked "Paul".

A variable set outside any function, called global, is visible almost everywhere in the program. Imagine `total` were global: another part of the program, computing delivery fees, could write into it between the end of the calculation and the display, and the screen would show a wrong total, with no error at all. The day the value is wrong, nobody knows who touched it. Hence the rule: give every variable the smallest scope possible. The exact scope rules vary from one language to another, and they are one of the first things to learn about yours: that is where the surprises hide.

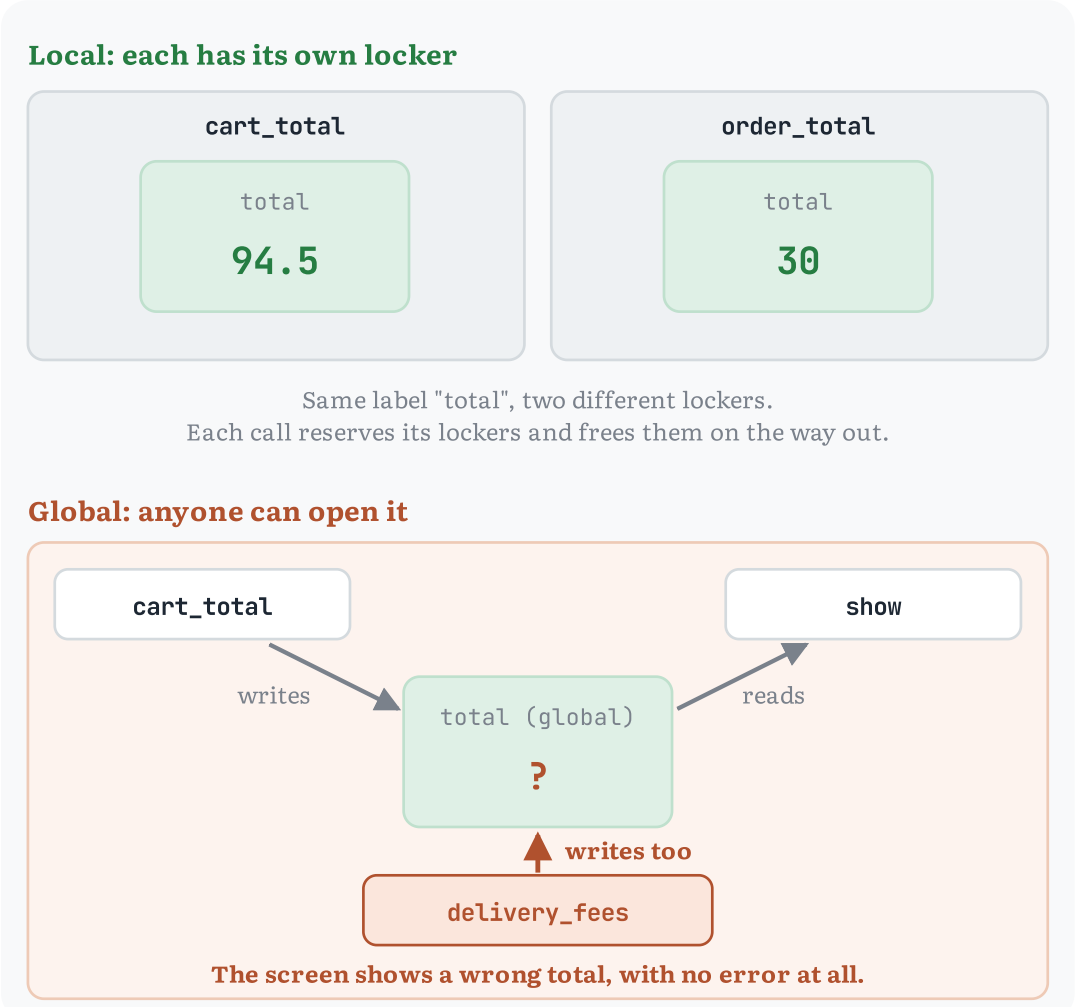


Figure 21 : At the top, two functions each have their own "total" locker. Below, one locker in the hallway: any function can write to it, and nobody knows who touched it.

When an instruction fails

Some instructions are impossible to run. Take the average price of an empty cart: you must share €0 among 0 items, and sharing among nobody makes no sense. Many languages then stop the program and print an **error**, a message saying what failed and on which line. Here is Python's for this calculation:

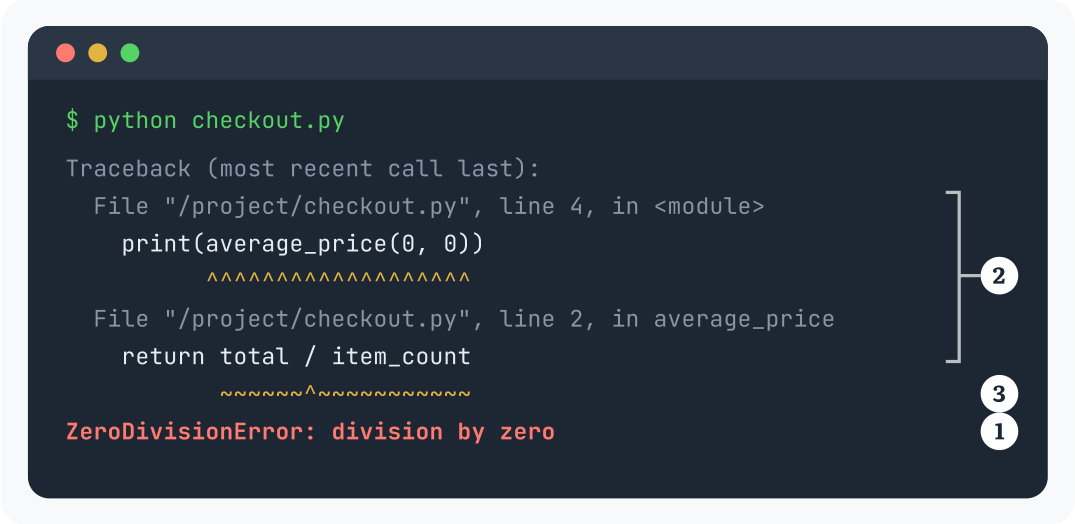


Figure 22 : A real Python stack trace, run on the checkout program (the file path is shortened). The numbers match the list: 1 what failed, 2 where, 3 the exact spot.

A **stack trace** is the message the language prints when a program stops on an error: the list of calls in progress, then the cause. It reads from the bottom up:

1. The last line says what failed: `ZeroDivisionError: division by zero`.
2. Above it, the list of calls in progress, the most recent last: the program starts at line 4 and calls `average_price`. The failure happens at line 2, inside that function.
3. The `^` and `~` signs underline the exact spot in the line.

The wording changes from one language to another, and some do not stop at all. In JavaScript, `0 / 0` returns `NaN`, "not a number", with no error at all. That `NaN` flows through the following calculations and ends up on the screen (2.3). Stopping dead is therefore a service: better a program that fails than a screen showing a wrong total.

One question remains open. When the chip reads `item.price`, what does it find in the locker of `item`: the price itself, or the address to fetch it from? The answer changes everything that follows.

2.2 Values and references: what the variable really holds

A variable's locker is a small space of memory, filled with 0s and 1s (1.1). It can hold two kinds of things: the data itself, or the address of another space of memory where the data is to be found. In most languages, it depends on the kind of data.

By value: the data sits in the locker

A **number**, like the €45 price, fits whole in the locker: its 0s and 1s are right there. A **boolean**, a value that can only be true or false, fits too. We then say the variable holds a **value**.

By reference: the locker holds an address

An item at the till, on the other hand, does not fit in a single number: it has a name, a price and a quantity. To keep it in one piece, languages offer the object.

An **object** is a bundle that gathers, under a single name, everything that describes a thing: the information it carries and the actions you can ask of it. Here, the thing is the item:

- The object's pieces of information are called **fields**: `name`, `price`, `quantity`.
- The object's actions are called **methods**, functions kept inside the object: `applyDiscount()`.

The point: handle the whole item at once, without chasing its name, its price and its quantity separately. The whole of chapter 4 will rest on objects.

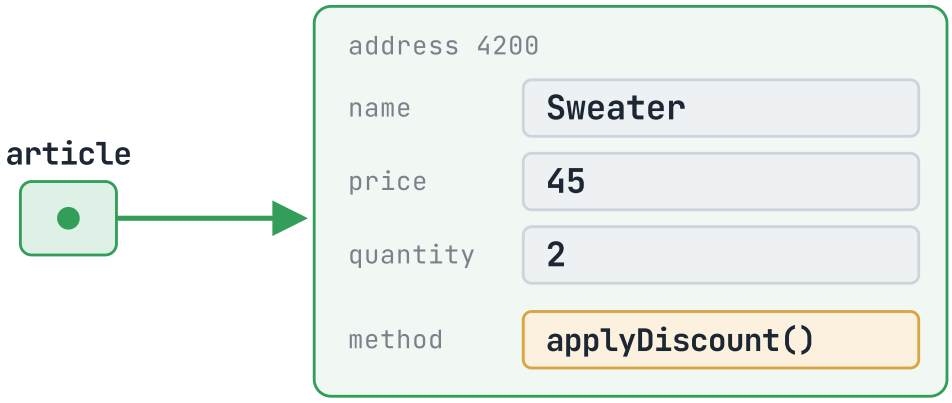


Figure 23 : An object is a group of lockers (its fields) and actions (its methods) kept under a single address. The variable article only holds that address.

A **list** stores values one after another, a **dictionary** finds them by a name. In most languages, for an object as for a list or a dictionary, the variable holds only the address of the data, its cell number.

An address stored in a variable is called a **reference**. Some languages, such as C, say a **pointer**. Hence the answer to the question left open in 2.1: the locker of `item` holds a reference. To read `item.price`, the chip jumps to that address and reads the price there.

Copying by value: two separate lockers

Copying a variable means writing `y = x` : "put into the locker `y` what the locker `x` holds". If `x` holds the number 5, `y = x` **duplicates the value:** `y` gets its own 5, and changing `y` does not touch `x` . This is copying "by value", like a photocopy: you can scribble on it, the original does not move.

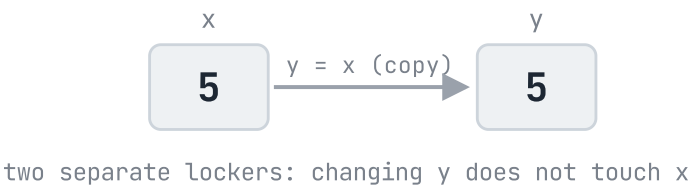


Figure 24 : The value is copied into a second locker: the two live separately.

Copying by reference: one content, two names

If `a` names a list, like `[1, 2, 3]`, its locker holds an address. Then `b = a` copies that address, nothing more. The list still exists only once in memory, and the variables `a` and `b` are **two names for one piece of data.**

This is copying "by reference", like the link to a shared document: if anyone edits the document, everyone sees the change, because there is only one. So `b[0] = 9` changes the first item of the list (counting starts at 0), and `a[0]` is 9 too.

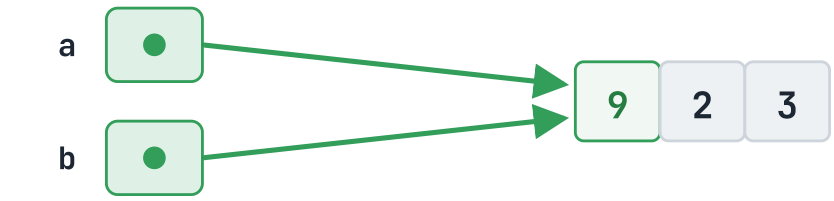


Figure 25 : a and b lead to the same list: changing it under one name changes it under the other.

```
const a = [1, 2, 3];
// in JavaScript, const freezes the address, not the content
const b = a;            // b gets the same address: the SAME list
b[0] = 9;
// a[0] is 9 too: one piece of data behind two names
```

Changing the list or replacing the locker

Let us take `a` and `b` again, which name the same list. Two lines look alike but do very different things:

- `b[0] = 9` follows the address and changes the list. Both names see the change.
- `b = [7, 8]` builds a new list elsewhere in memory and puts its address in the locker of `b`, in place of the old one. The locker of `a` has not moved: it still holds the address of the first list.

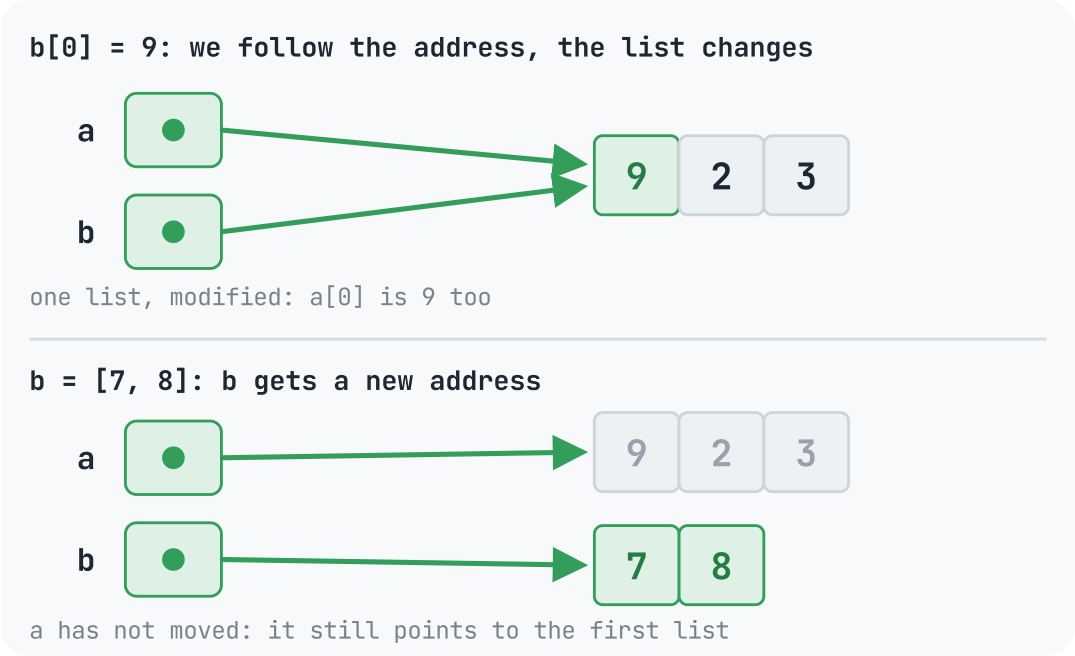


Figure 26 : Top, the arrow of b leads to the list and modifies it. Bottom, the arrow of b is replaced and the arrow of a does not move.

Keep this in mind: following the address changes the data, changing the address only changes the locker.

Which rule to remember: can it be changed in place?

To know whether a piece of data is copied by value or by reference, one question matters: can it be changed in place? Changing in place means altering the content of the data itself, like `b[0] = 9`, instead of building a new one. The size of the data makes no difference.

Data that cannot be changed in place is **immutable**: it is frozen, like printed text, and you can only build another one. For it, sharing or duplicating comes to the same thing, since nobody can change it under the other's nose. The language therefore picks whichever costs less.

- **By value**, for frozen data: number, boolean, text. Copying the variable makes a second one, and the two live separately. Putting a text in upper case builds a new one: the original stays intact.
- **By reference**, for data that can be changed: object, list, dictionary, in most languages. The locker holds the address, and copying the variable copies only the address. A change made under one name therefore shows under

the other.

One detail: in languages like Python, Java or JavaScript, a text is in fact stored by address. Since it is frozen, it nevertheless behaves like a value.

Passing to a function, copying deeply

In most languages (JavaScript, Python, Java), an object, a list or a dictionary passed as an argument to a function is not copied: if the function changes it, it changes the **original**. For example, a function `empty(cart)` that clears the contents of the list it receives also clears the real checkout cart, not a copy. Confusing value and reference is an endless source of "I changed a copy and the original moved" bugs. The reflex when coding: before changing an object you received, ask who else holds its address. It is the flaw of the global variable from 2.1 in another coat: several hands on the same data.

This shared address has one practical consequence: copying a list does not always copy what it contains. Take a cart, a list of two items. There are two kinds of copies:

- The **shallow copy** copies the list, but not its items: both lists show the same items. Changing the price of the sweater in the copy therefore changes it in the original too. In JavaScript, `[...a]`. In Python, `a.copy()`.
- The **deep copy** copies the items too, down to the last level: nothing is shared. In JavaScript, `structuredClone(a)` for a list or a dictionary. In Python, `copy.deepcopy(a)`.

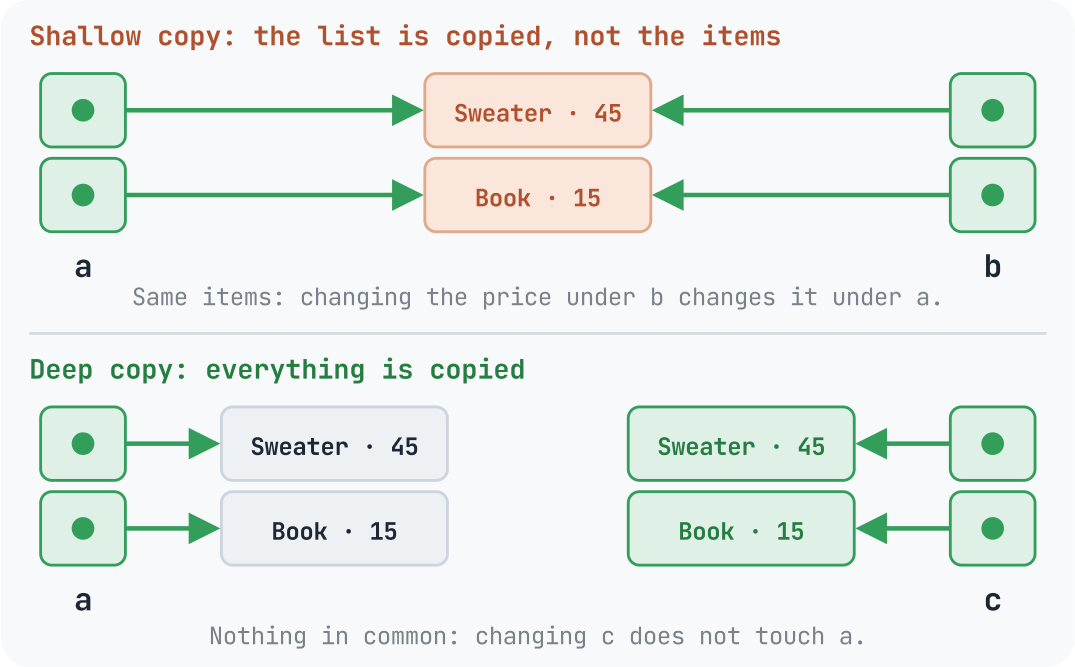


Figure 27 : Top, a and b show the same items (in orange, shared). Bottom, c has its own items (in green, copied).

↳ Learning Go · PHP 8 Objects

2.3 Types are sets of values

In the previous section we met numbers, text, lists, objects. Each kind of data is a **type**. Nearly every language offers the same basic bricks, under names that vary:

- the number, whole or with a decimal point (`number` , `int` , `float`): `42` or `3.14` ;
- the boolean (`boolean`): true or false;
- the **string**: text, like `"hello"` , a run of letters that are each a number (the ASCII and UTF-8 of 1.1);
- the list, or array (`array` , `list`): data side by side, like `[1, 2, 3]` ;
- the dictionary (`map` , `dict`): data found by a name, like `{name: "Lea"}` (the hash table of 1.3);
- the object (`object`): a bundle of data and actions.

The type decides what you can do with a value. The same `+` sign does two different things:

- between two numbers, it adds: `2 + 2` is 4.
- between two pieces of text, it glues them end to end, in most languages: `"2" + "2"` gives `"22"`.

Same signs, other type, other result. And facing an unexpected type, each language reacts in its own way: PHP, for example, converts both texts to numbers and answers 4. It is a classic source of surprises. When in doubt, test the operation in the language you use.

Why write your types

So far, types existed without anyone writing them: the language sees on its own that `12` is a number. Some languages also ask you to **declare** them, that is, to write down what you expect from each variable:

- with the type written: `int age = 12;` in Java, `let age: number = 12;` in TypeScript.
- without: `age = 12` in Python, `let age = 12;` in JavaScript, `$age = 12;` in PHP.

What is the point of adding this writing? A first answer lies in the **moment** when types are checked:

- **Static** typing checks the types before the program runs, so the mistake shows up before the first launch: Java, or TypeScript, explained below.
- **Dynamic** typing checks the types while the program runs, so you find out about the mistake when the code runs: Python, JavaScript.

Another thing varies from one language to the next, and it is often mixed up with the previous one: what does it do facing an unexpected type? Take `"5" - 1`, a text minus a number. Python refuses and stops with an error. JavaScript converts the text to a number without warning and answers 4 (below). The first is called **strict**, the second **permissive** (strongly and weakly typed, in the jargon). The two choices are independent: Python and JavaScript are both dynamic, yet one is strict and the other permissive.

Let us see what this changes on a small `total` function, price times quantity, written in JavaScript and called with bad inputs:

```
function total(price, qty) { return price * qty; }
total("12", 3)
// 36: a "12" typed into a form arrives as text, and works by accident
total("twelve", 3)
// NaN, "not a number", which travels all the way to the screen
```

A declared type is a commitment: "there will be a number here". TypeScript is JavaScript plus these commitments, written in the code:

```
function total(price: number, qty: number) { return price * qty; }
total("12", 3)       // error reported before running: "12" is text
total("twelve", 3)   // error reported too: "twelve" is not a number
```

TypeScript's compiler (a translator, like the one in 1.4) re-reads these commitments before the first run, and flags every broken one. You gain no typing speed, you gain the **moment** the mistake shows up: in front of you, not in front of the user.

The first job of a type is to make a mistake show up earlier. It does a second one, less visible and more rewarding day to day: it documents, that is, it explains the code to whoever reads it. `send(recipient, message)` does not say what it expects. `send(customer: Customer, message: string)` does: `Customer` is an object (2.2), a type you defined yourself, with its fields (name, address). The code editor knows it, so it offers you those fields as you type.

The first line of a function, its name followed by what it expects, is its **signature**. With types, it becomes documentation the compiler checks: as long as the types are respected, it does not lie.

Permissiveness has a price: coercion

A permissive language accepts almost anything without complaining. Its price is paid later: a wrong result arrives without the slightest error. JavaScript, the best-known example, converts a value from one type to an-

other on its own, for example a text into a number: that is **coercion**.

Compare the text `"5"` with the number `5`. The `==` sign asks "are these two values equal?" and answers `true` or `false`. The `===` sign asks the same question, but converts nothing.

```
5 == "5"
// true  (== converts the text "5" to the number 5, then compares)
5 === "5"
// false (=== compares without converting: a number is not a text)
"5" - 1       // 4     (- makes no sense on text: "5" becomes 5)
"5" + 1       // "51"  (+ sees text: 1 becomes "1", then both are glued)
```

That `"5"` is not artificial: a box to fill in on a web page (a form field) always hands back text, even when the user typed a number into it. The customer types 2 in "quantity", the code adds 1, and the cart shows 21 items, without a single error.

The cure fits in one character: in practice, always `===`, which compares without converting anything, never `==`, which converts first and compares after.

TypeScript, the safety net above, flags `5 == "5"` and `"5" - 1` as errors before the code even runs. But it lets `"5" + 1` through, because gluing text is still allowed. The defense: convert it yourself as soon as you read the box, with `Number(field.value)`, where `field.value` is the typed text and `Number` turns it into a number.

Knowing a language's permissiveness is knowing its bugs before you write them. Naming the trap helps you see it: once you know the word "coercion", you know to look for it in every language, each with its own conversion rules.

A type is a set of values

A type is more than a label (`int`, `string`, `boolean`). In the locker picture (2.2), it is the list of what a locker accepts: the **set of values** a variable can take. This simple idea is enough to explain how types are combined, and how a language decides that an object belongs to one.

- the type `boolean` holds only **two** values: true and false;
- the type `"red" | "green" | "blue"`, where the bar reads "or" (a text that can only be one of these three words), holds **three**;
- the type `string`, **infinitely many**.

Seen this way, types are combined like sets. Take two types of colors: A is "red" or "green", B is "green" or "blue".

- The **union** (`A | B`) gathers all the values of both types: red, green, blue.
- The **intersection** (`A & B`) keeps only the values they have in common: green.

The intersection lets you require two things at once. For example, an object must have the fields of a customer and those of a signed-in account. The more fields you require, the fewer objects qualify: the set shrinks.

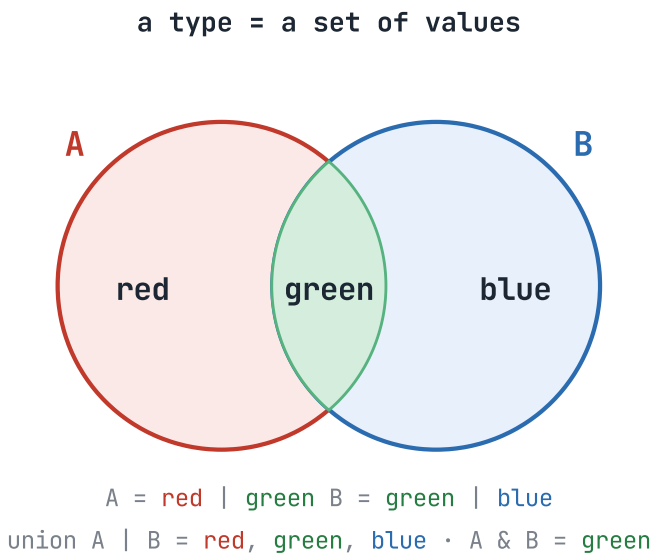


Figure 28 : Two types, A and B: the union keeps everything, the intersection keeps only the overlap.

Enumeration and absence: write down every possible case

A type that lists its values one by one, like the three colors, is an **enumeration** (`enum` in many languages). The status of an order is one: `"pending" | "paid" | "cancelled"`, and nothing else. The typo

`"payed"` is then refused before the code even runs. This shape makes the mistake impossible: 3.4 will call it a poka-yoke. A state you cannot write is a bug you cannot have.

One value is still missing from the list: absence. A locker may not hold a value yet, and each language has its own word for that:

- `null` in Java and JavaScript.
- `None` in Python.
- `nil` in Go.

A "customer or nothing" locker therefore holds two kinds of values: a customer, or the absence. Tony Hoare, who invented the null reference in 1965, an address that leads nowhere, called it his "billion-dollar mistake". Why so costly? The language lets you write "open the locker and read the customer" without warning that the locker may be empty. If you forget it, the program opens an empty locker and crashes, and nothing had flagged it beforehand. To protect themselves, people sprinkle "is the locker empty?" checks everywhere, and end up forgetting one.

Thinking in sets gives the cure. `Customer | null` reads "a customer, or nothing": the "or" bar, seen above, adds the absence to the set of customers. The compiler then refuses to open the locker until you have checked whether it is empty. Other languages call this `Optional` or `Option`. In 3.4, we will see when it is better for there to be nothing to check.

Interface and structural typing: the shape is enough

A type can also describe what you require from an object. This list of requirements is called an **interface**: a contract of the kind "to get in here, you must have this". The word covers two cases:

- **fields**: the `Point` interface requires an `x` and a `y`, two numbers.
- **actions**: in Go, a "reader" interface requires being able to read and close. A value meets it as soon as it can do these actions, without having had to declare it. A file and a network connection can both be read and closed: the same code can therefore read either one.

Section 2.6 will come back to the interface as a wall socket. Here, we look at how an object gets in.

Sets also explain **structural typing**: an object enters a type by its shape, not by the label it wears. It is a lock that does not ask who made the key, only whether it has the right shape. An object with an `x` and a `y` is a `Point`, even if it has never heard of `Point`.

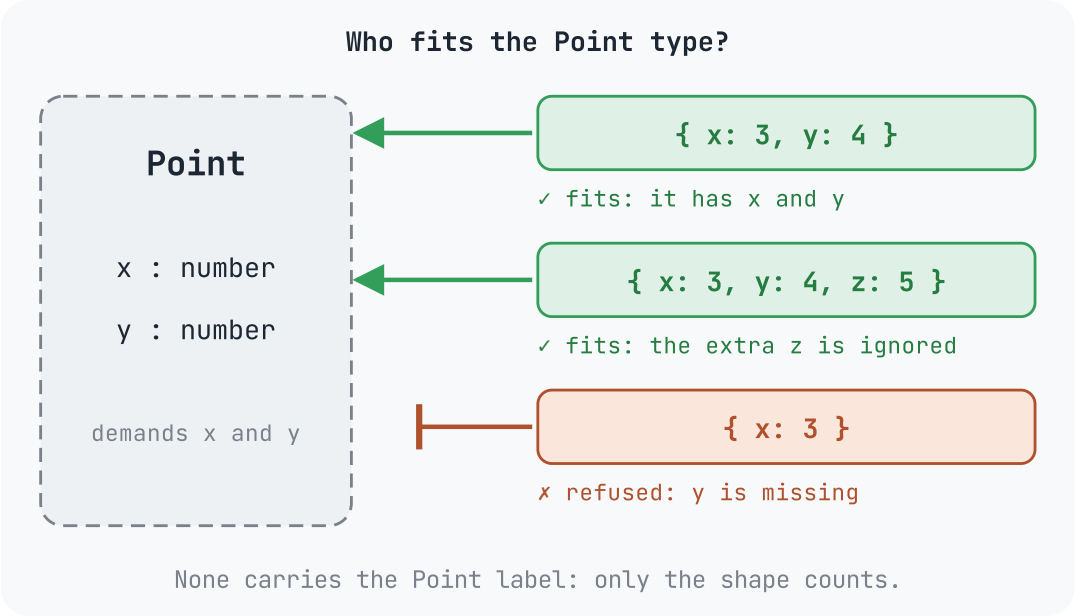


Figure 29 : The hole only demands x and y. The first two objects have that shape and fit, even with an extra z. The third has no y: it is refused.

```
interface Point { x: number; y: number }
function length(p: Point) {…}   // length expects a Point: an x and a y
const p = { x: 3, y: 4, z: 5 };
length(p)
// ✓ ok: the {x, y} shape is there, the extra z is ignored
```

Nominal typing, by name (Java, PHP), does the opposite: the object must declare in writing `implements Point`, "I am a Point", to be accepted. With structural typing, the **shape is enough**.

"I don't know" beats "trust me"

A program often receives data it did not write: a server's answer (a server is a remote computer), the content of a file, what a visitor typed. It arrives as text laid out in an agreed format, JSON. The `JSON.parse` instruction turns

that text into an object you can work with, but nobody guarantees what the object holds.

It is a stranger knocking at the door. Two reactions are possible: let them in without asking anything, or ask for their papers first. TypeScript offers both, with two types.

- `any` ("anything") switches the check off: a plain "trust me". It **spreads silently**, like an ink stain: whatever you pull out of it becomes `any` too.
- `unknown` says the opposite: "I don't know what this is yet, I'll check before using it". The compiler **stays on guard**: it refuses to read its fields or compute with it until a test written in the code has proven the type (for example "is it a number?").

Here are the two reactions, in TypeScript:

```
// txt: a JSON text received from outside
let a: any     = JSON.parse(txt);  a.customer.name
// ✗ no check: if customer is missing, the program crashes while running
let b: unknown = JSON.parse(txt);  b.customer
// ✓ the compiler refuses: prove what it is first
```

Why a written test, and not the compiler alone? Because types only serve while the program is being written. Once it runs, they no longer exist: only a test in the code looks at the real data.

You will meet this posture again facing user input in 7.6, then an AI's answer in 8.1: never treat as safe what you have not checked.

↳ Effective TypeScript · Learning Go · Eloquent JavaScript

2.4 Functions are values

A type says what you can do with a value. Now, in modern languages a function is a value too. Think of a recipe card: you can put it in a drawer, hand it to someone, give it back in exchange for something else, like any other thing. Handing over the card is not baking the cake: the function only runs when someone calls it.

Nothing magical: we saw in the introduction to this chapter that a function's name is just a label on its address in memory. Storing the function therefore means storing that address, like any piece of data. In practice, you can:

- store it in a variable or in an object (inside an object, it is called a **method**);
- pass it as an argument to another function;
- return it as a result.

Two examples in JavaScript:

```
button.addEventListener('click', myFunction)
// "when someone clicks, run this one"
[1, 2, 3].map(double)
// "map (here: transform) applies double (which multiplies by 2) to every
// element": [2, 4, 6]
```

In both cases, a function travels like a piece of data. That single idea unlocks two tools, the closure and the decorator. A third one resembles them: the generator also keeps its lockers after the call. All three look magical until you see the mechanism:

- the **closure**: a function that leaves with **its own locker**. It keeps the scope where it was born: the variables of that place stay within its reach from one call to the next (a counter keeping its private total);
- the **decorator**: a **sticky note stapled** onto a function. It adds a behavior without crossing out a single line of the original (timing, remembering results already computed, checking permissions);
- the **generator** (`yield`): a function that **hands you one value, stops, and resumes with the next** when you ask again. It never holds the whole sequence (walking a huge file without loading it whole).

The closure: a function that remembers

The closure is the most surprising. An example in Python:

```
def counter():
    n = 0                    # n lives inside counter()
    def inc():
        nonlocal n           # "the n above, not a new one"
        n += 1
        return n
    return inc               # we return the FUNCTION, not its result

c = counter()   # c is now inc
c()   # 1
c()   # 2  ← n survived between the two calls
```

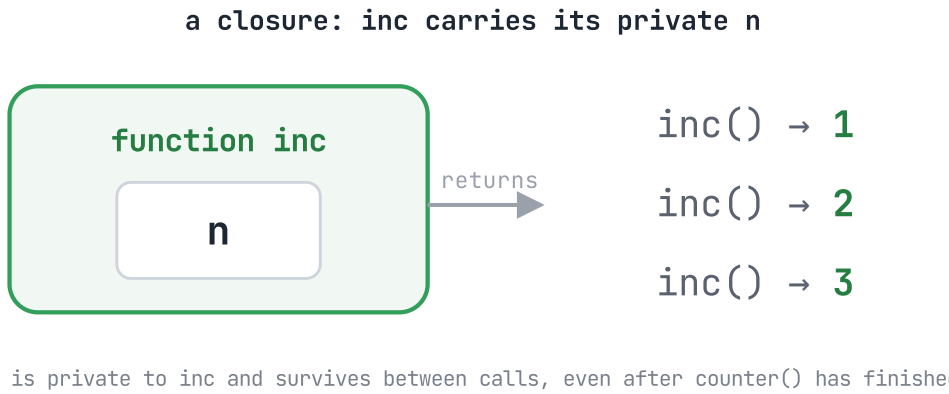


Figure 30 : Each call, inc adds 1 to its `n` and remembers it, hence 1, then 2, then 3. That is a closure.

Here, `inc` is written inside `counter()` and uses its `n` : that is what makes it a closure.

In 2.1, a function's lockers vanished when the call ended. There, `counter()` has finished, yet `n` survives. Why? Because someone still holds `inc` (the variable `c`), and `inc` carries the locker of `n` along. Returning `inc` only lets it leave `counter()` . What saves `n` is that the function is kept, in a variable or in an object.

Almost every language has closures. One detail changes: JavaScript and Go let you modify the original variable directly, Python demands `nonlocal` (seen in the code). To see what a closure is worth, look at what it replaces.

Without a closure, a function that must remember a value from one call to the next has three options:

- an object (2.2);
- a global variable, visible from everywhere, hence changeable by any part of the program;
- in some languages such as C, a `static`, a fixed locker attached to the function.

The values kept this way make up the function's **state**. The problem shows up when you want two counters, one for left clicks and one for right clicks:

- with a global or a `static`: **one** state, shared by every call. The two counters add up together;
- with a closure: a **brand-new state on every call** to `counter()`, hence as many independent counters as you want.

A closure is therefore a function that carries **a locker of its own** away with it: `n` stays hidden from the rest of the program, and each `counter()` builds a new one. Special case: two functions born from the same call share that locker.

The closure seems dense because it brings together three ideas, the last of them new:

- a function is stored in a variable (start of 2.4);
- a variable has a scope (2.1);
- a piece of data stays in memory as long as a variable keeps its address.

The **garbage collector** is the mechanism of the language that frees memory that no variable can reach anymore. It carries out this new idea: a piece of data stays as long as a variable holds it. It is a cleaning crew, like emptying a locker whose key nobody holds. Once nothing holds `inc`, it disappears, and its `n` with it.

The name does its job here: instead of describing a locker that outlives the call all over again, you say "closure" and use it as a brick. The decorator, right after, is built with that brick.

The decorator: wrapping without touching

The decorator follows directly. Timing a function without touching its code, for instance, still in Python:

```
from time import time
# time(): the clock in seconds, shipped with Python
def timer(f):                # takes a function, returns another
    def wrapper():
        start = time()
        f()                      # run the original function
        print(time() - start)    # and print the elapsed time
    return wrapper

def slow(): ...
slow = timer(slow)           # "slow" now names the wrapper
slow()                       # runs slow, then prints its duration
```

Here, `slow = timer(slow)` moves the label `slow` onto `wrapper`, a closure that remembers the real function, `f` (as `inc` remembered `n`). Every call to `slow()` now goes through the wrapper.

Python offers a shortcut: writing `@timer` on the line above `def slow():` is exactly the same as writing `slow = timer(slow)`. Web tools for Python such as Flask or Django use this to check, in one line (`@login_required`), that the user is logged in before running a function.

The generator: one value at a time

A 50 GB file does not fit in memory: loading it all at once is the trap from section 1.2. The generator gets around it by handing back its values one at a time, on demand, drop by drop. In Python:

```
# ✗ a list: every cleaned line, handed back at once
def clean_all(file):
    cleaned = []
    for line in file:
        cleaned.append(line.strip())
        # strip(): removes spaces at the start and end of the line
    return cleaned
    # the whole list arrives at once, in memory

# ✓ a generator: one cleaned line at a time
def clean_one_by_one(file):
    for line in file:
        yield line.strip()
        # yield: hands back one line, then pauses the function

for line in clean_one_by_one(my_file):   # lines arrive one at a time
    handle(line)
# → handle a 50 GB file without ever loading it whole into memory
```

The two functions share the same loop. Only what it does with each line changes: put it in a list, or hand it back right away. At each `yield`, the function hands back one line, then stops without freeing its lockers: it keeps its place in the loop. When the caller's `for` loop asks for another (this is called `next()`, "give me the next one", as the diagram below shows), it resumes right after the `yield`.

Catch: a generator can be walked only once, like a bottle you have emptied. To read the file again, you have to create a new one.

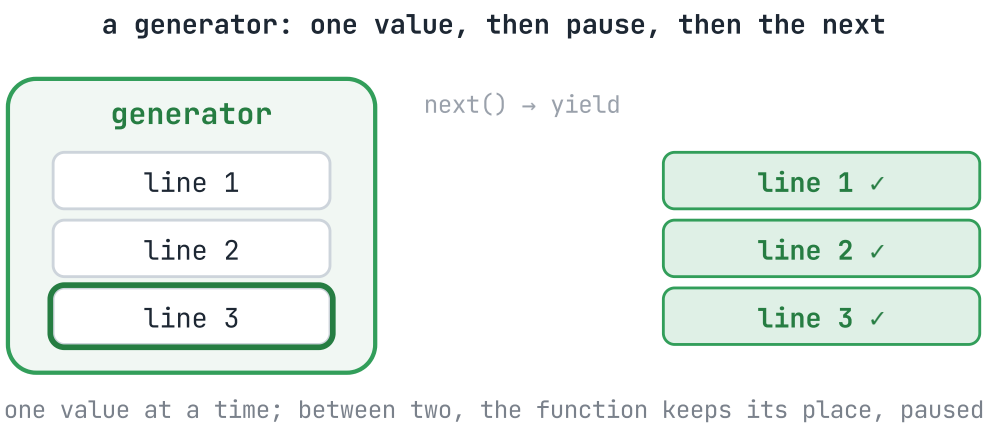


Figure 31 : Values come one drip at a time, never all at once: that is what lets you read a 50 GB file without loading it.

Libraries: other people's functions

So far, we have learned to write cleverer functions. But most of the time, you use other people's: you do not write every recipe yourself. The `sum(prices)` from the chapter introduction is not yours: Python ships with ready-made functions (`sum` , `print` and many more), called just like the ones you write. A **library** is a cookbook written by others: a collection of functions that a program adds so it can use them without rewriting them.

The line `import random` ("bring in the library called `random` , the one for chance"), placed at the start, gives access to `random.randint(1, 6)` :

```
import random                       # "bring in the library called random"
roll = random.randint(1, 6)
# a random whole number from 1 to 6, like a die
print(roll)
```

A **standard library** is the collection of ready-made functions that ships with the language: `random` is part of it, and `sum` works even without an `import`. Any other library is downloaded and installed first, for example to draw a chart, edit a photo or read a barcode.

↳ Fluent Python · Eloquent JavaScript

2.5 Recursion: solving a problem with a smaller version of itself

If a function is a recipe, **recursion** is a recipe that quotes itself. "To empty a box: take the things out. If you find a box inside, apply this recipe to that box. If there is none, you are done." A **recursive function** is therefore a function that calls itself. At first it sounds like an infinite loop, but it isn't. It all hinges on two parts:

- the **base case**: the box that holds no more boxes. You empty it and you are done, no further call;
- the **general case**: the box that holds another one. The problem reduced to a smaller version of itself.

Since each box you find is smaller than the one it came from, you always land on the base case. Without it, you would keep opening boxes forever, like two mirrors facing each other.

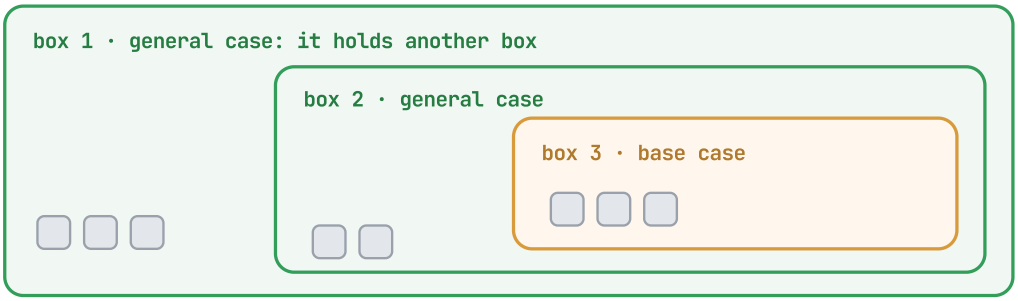


Figure 32 : Each box you open hides a smaller one (green), down to the one that holds only things: the base case (orange).

Watching recursion at work: the factorial

The trick to writing a recursive function without tying your brain in knots: handle the base case, then **trust** the smaller call, assuming it already returns the right result. You do not have to picture the whole cascade of nested boxes, only the one in your hands. The classic example is the factorial, the multiplication of every integer from 1 to `n` (`fact(4)` is 4 × 3 × 2 × 1). It is not the most useful case, it is the smallest one where the whole mechanism is visible, here in Python:

```
def fact(n):
    if n <= 1: return 1        # base case
    return n * fact(n - 1)     # reduce to a smaller problem
# fact(3) = 3 × fact(2) = 3 × 2 × fact(1) = 6
```

Each call to `fact` is a box: `fact(3)` holds `fact(2)`, which holds `fact(1)`, the box with no box inside. That is the base case.

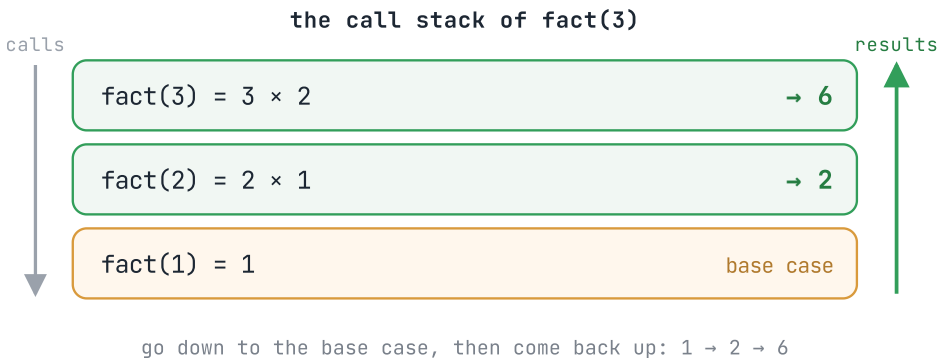


Figure 33 : We go down to the base case (orange), then the results come back up: 1, 2, 6 (green).

What the machine remembers: the call stack

Under the hood, each call **waits** for the next one's result. As seen in the introduction to this chapter, calling a function means jumping to its instructions in memory, then coming back. Each call must therefore note the instruction to resume at and its own values (the `n` that is 3, then 2), and these notes pile up, one per opened box, until the smallest one is empty: that is the **call stack**. As with a stack of plates, the last note put down is the first one picked up: the stack **unwinds**, passing results back up.

You have already seen this stack printed: it is the stack trace of 2.1, which lists the calls in progress, one per open box, then the cause of the failure. Python writes it from the oldest to the newest call, JavaScript and Java the other way round. Take the one for the average price of an empty cart:

```
Traceback (most recent call last):
  File "checkout.py", line 4, in <module>
    print(average_price(0, 0))      # the caller: it passed 0 items
          ^^^^^^^^^^^^^^^^^^^
  File "checkout.py", line 2, in average_price
    return total / item_count       # the line that failed
           ~~~~~~^~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Reading a trace means reading the stack: the line that failed is in the most recent call, but the cause is often one step up, in the caller that passed a bad value. Before searching the code, you read the trace: the message first, the failing line next, then the caller.

Hence a very real limit: a recursive function that calls itself too many times before reaching the base case **overflows the stack** (a *stack overflow*). By default Python refuses to go past about 1,000 nested calls and raises a `RecursionError`. JavaScript also throws an error, at a limit that depends on the browser. A forgotten base case, or a call that does not shrink the problem, leads straight there.

The trap of repeated calls

Another trap, a quieter one: calls that repeat. In the Fibonacci sequence each number is the sum of the two before it, and the naive function fits in two lines (the `if` and the `return` of the code below, minus the `@cache` line). But `fib(40)` calls `fib(38)` twice, `fib(37)` three times, and more than 300 million calls in all: each extra step multiplies the work by about 1.6, an exponential explosion, in the family of the `O(2ⁿ)` from 1.3.

The cure fits in one line, a decorator from the standard library, named `cache` like the fast memory of 1.2. It keeps in memory the results already computed (this is called memoization): each value is then computed only once.

```
from functools import cache
@cache
# keeps each result: fib(38) is computed only once
def fib(n):
    if n < 2: return n          # base case
    return fib(n - 1) + fib(n - 2)
fib(40)   # 102334155, in 41 computations instead of 331 million calls
```

Where it shines: trees

Recursion is the natural tool for anything **tree-shaped** (shaped like a tree: a branch that splits into smaller branches):

- walking a folder and its subfolders, the digital version of the boxes at the start;
- a web page: its HTML text (the language that describes the page) is made of nested tags, a paragraph inside a section inside the page, which the browser arranges as a tree, the **DOM** (Document Object Model);
- a nested JSON, lists inside lists.

```
def size(folder):
    total = 0
    for e in folder:
        if is_folder(e): total += size(e)
        # a folder inside: we call ourselves
        else: total += e.bytes              # a file: nothing to open
    return total
```

When a folder holds several subfolders, the function goes to the bottom of the first one, climbs back up one level, then opens the next: a depth-first walk. The code matches the **shape of the data**: a folder holds files and other folders, the function calls itself on the latter, and the base case is the folder that holds no more. With a loop alone, you would have to maintain the list of folders still to visit yourself. Data made of smaller versions of itself, like a folder of folders, calls for a function that calls itself.

The word "recursion" gives you the reflex: facing a folder of folders or a menu of submenus, you recognize the shape and know where to start.

The price: each call takes a slot on the stack, so to walk a plain list, a loop stays cheaper.

↳ Grokking Algorithms · Eloquent JavaScript

2.6 Naming a concept changes what you can think

A language's real power is to **give a name to a complicated idea**. Once you can say "a thread", "a promise", "an interface", you reason about it without re-deriving the machinery underneath. Three names, each one further from the machine than the last: the thread hides the chip's back-and-forth between several tasks, the promise hides a wait, the interface hides a concrete type. At each step, the machine sees nothing change. You are the one who thinks higher.

The thread: a line of instructions

First name: the thread. A **thread** (thread of execution) is a sequence of instructions that the chip follows one after the other. Think of it as a cook, who prepares one dish at a time: that will be our comparison. By default, JavaScript runs your code on a single thread: that is its only cook.

Your computer keeps dozens of programs open at the same time, but its chip has only a few **cores**, computing units that each follow only one thread at a time. The **operating system** (Windows, Linux or macOS) is the program that manages the machine, a bit like a head chef. For each program you launch, the operating system creates a **process**: the running program together with its own area of memory, shared by all its threads. The operating system makes the threads take turns on the cores, so fast that they seem to move together.

Two terms often mixed up part ways here. **Concurrency** is having several tasks in progress, even if only one moves at any given instant, like a lone cook moving each order forward in turn. **Parallelism** is several tasks moving at the same instant, one per core, like two cooks side by side. By default, your

JavaScript code is concurrent without being parallel: a single cook moves the orders forward in turn. Day to day, the machinery stays hidden behind the word "thread": that is the point of a name.

The promise: a value that will arrive later

Second name: the promise, the name that trips up every JavaScript beginner. Some operations take time, like fetching data from a server (a remote computer that answers). JavaScript does not freeze while it waits: it keeps going, and calls your code back once the result arrives. Code that keeps going without waiting for the answer is called **asynchronous**.

How does JavaScript keep going? Its single thread, that is, its single cook, takes the tasks, like orders, from a queue, one by one. The **event loop** is the mechanism that keeps taking the next task from the queue of tasks (a click, a server answer, a timer: each one drops a task in it). Many languages have their own, Python and C# for example.

That is why a very long computation freezes the page: while it runs, nobody else in the queue moves. Waiting on the network, though, freezes nothing, because the cook does not do the waiting: he hands the request to the browser, which takes care of the network, and moves on to the next task. When the server answers, the browser adds a new task to the queue, "handle the answer": it waits its turn behind the ones already there, then runs.

The traditional way to collect the server's answer is to hand over a **callback**, a function to be run later: "when you are done, run this". But as soon as one request (a demand sent to the server) needs the result of the previous one, you nest a callback inside a callback inside a callback: the code drifts to the right and becomes unreadable. That is the infamous "callback hell".

```
// ✗ with callbacks: a callback inside a callback inside a callback
// (r1 => … is an anonymous function: nameless, written where it is used,
// called when r1 arrives. The arrow reads "gives")
get(a, r1 => {
  get(r1, r2 => {
    get(r2, r3 => show(r3));   // each step sits one level deeper
  });
});
```

A **promise** is a value that is not there yet but will arrive later: a function like `get` returns one, instead of the value itself. The keyword `await` means "wait here until the value arrives, then hand it to me as a normal variable". Only the function containing that `await` pauses: meanwhile the cook handles the other tasks in the queue, and the page does not freeze.

That pause is the generator's pause from 2.4: the function stops while keeping its lockers, and resumes at the next instruction once the value is there. The three requests in a row then read top to bottom, like ordinary code:

```
// inside a function marked async (the word that allows await)
const r1 = await get(a);    // wait for the 1st result
const r2 = await get(r1);   // then the 2nd, which depends on the 1st
show(await get(r2));         // then the 3rd
```

Here, each request needs the previous one, hence the three `await` in a row. If they were independent, awaiting them one by one would put them end to end: three times as long. Better to start them all, then wait for the three at once with `Promise.all`, a promise that comes true when all the others do:

```
const p1 = get(a), p2 = get(b), p3 = get(c);
// all three leave at once: three promises
const [r1, r2, r3] = await Promise.all([p1, p2, p3]);
// we wait for the three together
```

If one of the three fails, `Promise.all` fails at once, and the error comes out at the line of the `await`, like an impossible instruction in 2.1 (we will see in 3.4 how to catch it).

Careful: reading top to bottom does not mean nothing else runs between the lines. Threads that share memory could change the same variable at the same instant: with a single thread, JavaScript rules out that risk. What remains is **interleaving**: tasks moving forward in pieces, taking turns. Between two `await`, another task can change the variable, and your code resumes with a stale value (same family of bug as in 6.7).

The interface: the shape of the holes

The first two names hide time (a thread, a wait). The third hides a type. Third name: the interface, the list of what you demand of a piece of code, already met with types (2.3). A wall socket gives the picture: the wall does not say which appliance will be plugged in, only the shape of the holes, and anything with the right plug fits, lamp or vacuum cleaner.

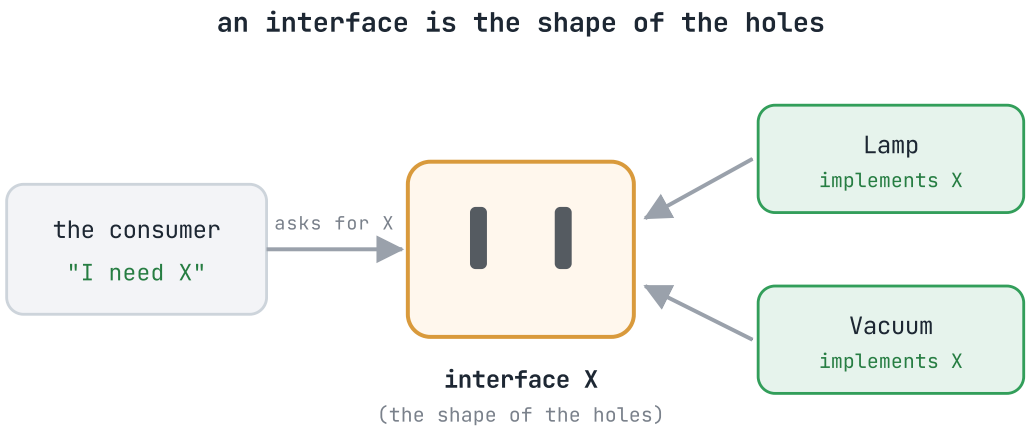


Figure 34 : In orange, interface X: the only thing the consumer knows. In green, the appliances that plug into it: lamp, vacuum or the one invented tomorrow, without changing a line of the consumer.

That one word changes the relationship between two pieces of code, one that uses (the consumer) and one that provides (the provider). The consumer says "I need something that can do X" without knowing the concrete type behind it, and the provider only has to be able to do X, and to say so in one word (`implements`) where the language requires it. This name is enough to make the two independent: either can change without the other noticing. It is the starting point of chapter 4.

For the machine, these three names change nothing: it still runs instructions on bytes, whether you say thread, promise or interface. Only the way you think about it changes. So a language does not merely translate your thinking: it decides what you can think without effort.

↳ Learning Go · Eloquent JavaScript

2.7 Three ways to write a program

One last word, which you will meet in job ads. A **paradigm** is a way of thinking about and writing a program. The chapter practiced three of them without naming them:

- **procedural programming** writes the program as a sequence of instructions followed in order, gathered into functions;
- **object-oriented programming** (OOP) bundles data and the actions you can ask of it into a single object, like the checkout item (2.2, then the whole of chapter 4);
- **functional programming** lets functions travel like data (2.4) and wants them never to change what they receive: instead of altering the list it was given, a function returns a new one with the change made. The trap of 2.2, where a list changed under one name changes under the other too, disappears.

The functional style rests on two words. A **side effect** is anything a function changes outside its result: a text shown on screen, a global variable, a received list it alters in place. A **pure function** does only one thing: compute its result from its arguments, with no side effect and without reading anything hidden.

In JavaScript, `names.sort()` sorts the list in place, whereas `names.toSorted()` returns a sorted list and leaves the original intact: same job, one with a side effect, the other pure. A pure function is also easier to test: same arguments, same result, so you compare what it returns with what you expect, like `priceWithTax(100)` which must return 120. No useful program is free of side effects (showing, saving, sending): they are pushed to the edges and the core stays pure.

PHP, JavaScript and Python let you mix all three ways. Java is object-oriented first, the functional style having arrived only with Java 8, and C is mostly procedural. For each problem, you pick the one that states it most simply: a shopping cart is modelled with objects, a list of prices to transform is written in the functional style.

Robert Martin, author of *Clean Architecture*, a book on how to design software, makes a surprising remark: no paradigm has added a power, each has taken one away. Procedural style, with the three moves of 2.1 (Martin calls it structured programming), takes away `goto`, the jump to anywhere. Objects take away the choice of which function to call: instead of making it yourself, you ask the object, with `shape.draw()`, and it is the object, circle or square, that knows how. The functional style takes away assignment, the "put into": a piece of data no longer changes, you make another one.

Each removal is a gain, because what you can no longer do, you no longer have to watch for. A paradigm is a discipline: it deprives you of a freedom to keep the program readable. There is a limit, though: you remove as long as the program can still do what you ask of it. That limit has a name, the rule of least power, which Tim Berners-Lee, the inventor of the web, raised to a principle: for the same result, prefer the tool that can do the least. This is why a page is described in HTML, which you can only read, rather than in JavaScript, which you must run.

An unknown language reveals its traps before its bugs. Five questions are enough to find them:

- what does it copy when you copy a variable, the value or the address (2.2)?
- how far does a name reach (2.1)?
- what does it do with a value of the wrong type, does it refuse or convert in silence (2.3)?
- how many cooks work together, and do they share the memory (2.6)?
- what becomes of an integer when it is full (1.1)?

The chapter gave the answers for a few languages. Asking them of another is how you get to know it in an hour.

↳ Clean Architecture

↳ WHICH LEADS TO CHAPTER 3

Three moves (chain, repeat, choose) are enough to compute anything. The chapter gave them words: lockers holding data or an address, types that bound what they accept (unless the language converts silently), functions passed around like numbers or calling themselves, names that hide the machinery, and three ways of writing a program. Each named thing enlarges what you can think.

We can now express anything. But knowing every word of a language has never made anyone a good writer: a sentence can be grammatically perfect and still unreadable, like those official letters you have to read three times. A program too can be exact and still a headache. The next challenge is no longer the language's power but **clarity**.

"What is well conceived is clearly stated, and the words to say it arrive with ease." — Boileau, *L'Art poétique*, 1674

3 Writing for humans

cleanliness

Code is read far more often than it is written: Robert Martin, in *Clean Code*, counts more than ten minutes of reading for every minute of writing. The machine does not care about elegance: a one-letter name like `x` runs just as fast as a meaningful one like `priceWithTax`. So the cleanliness of code, its clarity, is not for the machine: it is **communication** with the next human who opens the file.

That next human is often you. A recipe scribbled for yourself works fine, right up to the day someone else has to follow it, or you reread it six months later. This cleanliness is built in six habits: naming well, building deep functions, commenting the why, making the error disappear, not repeating yourself, and reworking code without changing what it does.



Figure 35 : Written in summer, reread six months later: you no longer recognize your own code.

3.1 The name reveals the intent

The name is the first sign of clarity, and when it does not say enough, you are tempted to explain it on the side. A **comment** is a note written for humans in the middle of the code, after `//` or `#` depending on the language, like the gray lines in this book's examples: the machine ignores it. But a comment that only translates an obscure name repairs the wrong place: fix the name. According to Robert Martin in *Clean Code*, a good name answers three questions at once:

- **why** it exists;
- **what** it does;
- **how** you use it.

Compare the raw condition with the same condition stored under a name:

```
// ✗ decode it on every read: adult (>= 18), AND (&&) positive balance,
// AND not (!) suspended
if (age >= 18 && balance > 0 && !suspended) { ... }

// ✓ the rule, written once, under a name
function canPlaceOrder(customer) {
  return customer.age >= 18 && customer.balance > 0 && !customer.suspended;
}

// ✓ everywhere else, the name IS the explanation
if (canPlaceOrder(customer)) { ... }
```

Read aloud, `canPlaceOrder` answers all three questions. Why it exists: to stop a forbidden order before it goes through. What it does: say whether this customer may order. How to use it: the `can…` prefix signals a yes-or-no answer, to put in an `if`.

When you code, the payoff shows the day the rule changes. If a customer must also have a verified email address, you fix `canPlaceOrder`, and every `if` that uses it follows without being touched. The power to name from section 2.6 comes down here to the scale of a single line: the code says "place an order", in the language of the business, no longer "18, balance, suspended".

One detail still jars inside the function: the 18. A **magic number** is a number written as is in the code, with nothing to say what it stands for. Why 18? It is the age of majority, but you have to guess. And the day the shop opens in a country where adulthood starts at 21, you must hunt down every 18 in the program, without touching the ones that count something else.

The cure: give the number a name. A **constant** is a variable you fill once and never write to again, often spelled in capitals so that it stands out at a glance. In JavaScript, the word `const` forbids writing to the variable again: you write `const LEGAL_AGE = 18`, then `customer.age >= LEGAL_AGE`, and the line reads without guessing. The till of 2.1 hid two magic numbers, the €100 threshold and the 0.9 multiplier (10% off), each worth a constant. A text written as is raises the same problem: a typo on `"payee"` slips through silently, and the cure is the same, a constant, or better a type that admits only those words (2.3).

The constant fixes the reading, not the rule. Adulthood is a rule of the activity the program serves, and it varies by country: 18 in France, 21 in Singapore. As soon as the program serves several countries, the age is no longer a number carved into the code but a piece of data, stored in the database, country by country, which `canPlaceOrder` goes and fetches. The name stays, the value has moved out.

The name must also survive changes to the code. Take a function `findCustomer`, which looks a customer up in the database and returns it. One day, it is made to create the customer when it cannot find one. Its name now lies: the code that calls it thinks it is reading, but the function is writing. A lying name costs more than a vague one, because people trust it without checking. You rename it right away, `findOrCreateCustomer`, and every line that calls it gets read again with fresh eyes. Better still, split reading (`findCustomer`) from writing (`createCustomer`): this is Bertrand Meyer's *command-query separation*, which says a function that answers a question should change nothing.

A name that survives change is not enough yet. Robert Martin also asks that it hold across the whole program. If the code says `findCustomer` here, `fetchCustomer` there and `getCustomer` elsewhere for the same operation, the reader believes in three different operations and looks for a nuance that does not exist. **One word per idea, everywhere:** a single verb for "go and get", a single noun for "customer", and the same ones from one file to the next. And if those words are also the ones the people who run the business use (sales staff, accountants), everyone speaks the same language, which Eric Evans calls the *Ubiquitous Language* (4.5).

Some of these rules call for no judgment at all, and we hand them to tools. The **formatter** rewrites spaces, line breaks and indentation so the whole project looks the same, whoever wrote it. The **linter** rereads the code without running it and flags what a machine can see: a variable never read, a name that breaks the project's convention, a `==` where `===` was needed (2.3). Nobody argues about form in review any more, and attention is kept for meaning, names and the rules no tool checks.

↳ Clean Code

3.2 The deep function, not just the short one

A good name hides a rule behind a word. The next question is how far to split code into functions, and here two masters clash:

- *Clean Code* (Robert Martin): functions should be small, then smaller than that;
- *A Philosophy of Software Design* (John Ousterhout): past a point, cutting further helps no one.

The real question is not "how many lines?" but "is it simpler for the **caller**?", meaning the code that uses your function. You will often write that calling code yourself, three months from now, once you have forgotten what the function hides. So you must be able to use it without knowing anything about its inside: at a restaurant, you order "a tiramisu", not "eggs, sugar, mascarpone, and I will whip it up myself".

Take a price to compute, in four steps: read the pre-tax price, subtract the discount, add the VAT, add the shipping.

```
// ✗ exposed pieces: the caller must know the 4 steps and their order
const pretax     = readPrice(order);
const discounted = applyDiscount(pretax, order);
const taxed      = applyVat(discounted);
const total      = addShipping(taxed, order);

// ✓ a "deep" function: one call, the 4 steps hidden inside
const total = finalPrice(order);
```

The decomposition itself is not the problem: `finalPrice` can perfectly well call those four steps internally, the way a cook goes through the steps of a dish. But those four steps become private: a **private** function is reserved to the code that contains it, and the rest of the program cannot call it. It sees only `finalPrice`, as the customer sees only the dish, never the kitchen. Depending on the language, the protection comes from a keyword (`private` in Java) or from a mere convention (a leading underscore in Python).

The caller has one name to know instead of chaining the steps by hand. The day the VAT rule changes, you touch only the inside of `finalPrice`: this is the *information hiding* principle, set out by David Parnas in 1972.

The **deep** function hides a lot of work behind a tiny **surface**, meaning what the caller must know to use it (here one name and one argument): one line on the menu, everything else in the kitchen. **Split, yes; expose the pieces, no.**

Cutting has a limit, though. A function that hides almost nothing is called **shallow**: `applyVAT(price)`, if it boils down to `price * 1.2`, simplifies nothing. The caller could have written the calculation itself, and now has one more name to remember.

Sometimes cutting even backfires: when two steps are so tightly bound that neither makes sense without the other, the reader has to jump back and forth to follow the calculation, and the code is harder to read than before. Two func-

tions you can only read together are **conjoined functions** (*A Philosophy of Software Design* calls them *conjoined methods*), and you are better off merging them into one.

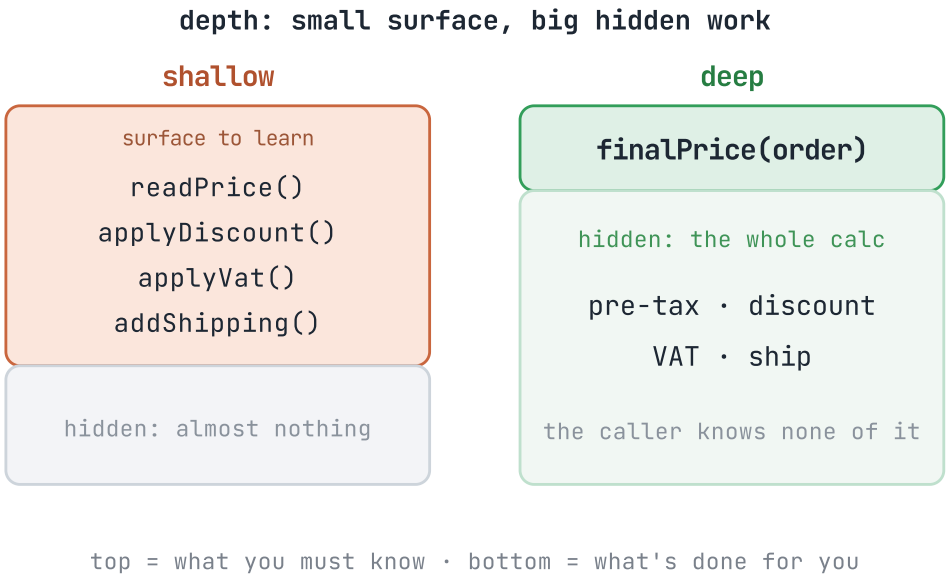


Figure 36 : Bright: what the caller must know. Pale: what is hidden from it. The deep function exposes one name (`finalPrice(order)`), the shallow one its four steps.

Ousterhout's most telling example goes beyond the function: the files of Unix, the system whose calls Linux and macOS reproduce. To work on a file, the programmer has only five calls to know: `open` , `read` , `write` , `lseek` (move around inside the file) and `close` . Behind them, hundreds of thousands of lines handle the disks, the access rights and the hardware, without the programmer seeing any of it. It is the deep function at large scale: five words on the menu, a whole kitchen. Ousterhout calls this surface an "interface."

↳ Clean Code · A Philosophy of Software Design

3.3 The comment says why, not what

The name is chosen, the function is split: what remains is the comment, the note written beside the code. A good name already says what the code does. Should we then do without comments?

On comments, *Clean Code* is scathing: Robert Martin goes as far as writing that **comments are always failures**, proof you couldn't express yourself in the code. A comment that merely **paraphrases** the code, that is, says again what it does, ages and ends up lying, because the code changes and the comment does not. In that case, Martin is right.

Think of the notes in a recipe's margin: "do not open the oven before 20 minutes" saves the cake, "beat the eggs" written next to the line that says to beat the eggs is worth nothing.

Code works the same way. The first note below repeats its line. The second says what no line can: why the `timeout`, the delay past which you stop waiting for an answer, is 29,000 milliseconds, or 29 seconds (written `29_000`, the underscore is only for readability) and not 30,000.

```
user.deactivate();
// ✗ deactivates the user          (the code already says this)
timeout = 29_000;
// ✓ 29 s: just under the load balancer's 30 s, which cuts the connection
// past that
```

The reason is written nowhere in the code: a server placed in front of ours, the load balancer (the one that spreads the visitors' requests), cuts any connection left without an answer after 30 seconds. So our code gives up one second earlier, to show its own clear error instead of an unexplained cut. Only a comment can say so.

Yet Martin himself does not ban every comment: he keeps those that explain an intent or warn of a danger. *A Philosophy of Software Design*, by John Ousterhout, disputes the idea that good code needs no comments and sets a rule: the good comment says what the code cannot. Three kinds are worth writing:

- **the why**: the non-obvious decision, the trade-off behind the code.
  `// in cents: zero floating-point rounding`
- **the warning**: the trap, the order you must not break.
  `// do NOT reorder: validate before saving`

- **the contract**: what a function guarantees its caller and what it requires of the caller, so the caller never has to read the lines inside it. Types already say part of it for free: what it expects and what it returns (section 2.3). The comment only takes over for the rest, what types cannot say.
  `// writes to the database, assumes the list is sorted`

One why, though, does not belong in a comment: the why of a change. Why was this line modified on Tuesday? The answer tells the code's history, not its state, and every edit would add a line to the comment. That history has its own notebook: the commit message, the note you write to describe a change when you save it in Git. Git is the tool that keeps the code's history. The book presents it in 6.3, and for now it is enough to know it exists. The `git blame` command shows, next to each line, the message of its last change.

The comment even works as a detector. When it grows long and painful to write, it is "the canary in the coal mine", the bird miners carried because it succumbed to the gas before they did. A comment that is painful to write signals a badly split function: its surface, what the caller must know, is too complicated to say simply (3.2). Ousterhout even writes his comments before the code, as a design tool: if the explanation will not come, it is the design that needs fixing, not the sentence.

↳ Clean Code · A Philosophy of Software Design

3.4 Define the error out of existence

A comment can warn of a trap. Better if there is no trap. A SIM card has one cut corner: it only goes in one way, so nobody needs a warning, and nobody gets it wrong. A shape that makes the wrong move impossible is called a **poka-yoke** (Japanese for "mistake-proofing"). It is an underrated idea: the best way to handle an error is for it to be unable to happen.

Code can do the same, but it often does the opposite. Take the operation "take the first 10 characters" of a text. What should it answer for "hi", which has only 2? Java stops on an error. Python changes the definition of the operation to

"take up to 10 characters": the case that used to blow up is no longer an error, it is a normal case. Here are both (the Java example is John Ousterhout's, the Python one is added here):

```
// ✗ Java: stops on an error, to be prevented by a check before every call
"hi".substring(0, 10);
// 💥 StringIndexOutOfBoundsException (index out of range)

# ✓ Python: the slice [0:10] (positions 0 to 9) clamps to what exists, no
# error
"hi"[0:10]              # → "hi"
"hi"[10]
# 💥 IndexError: a single position has no right answer
```

This Java error is called an exception. When a function cannot do its job, many languages make it **throw an exception**: it stops right there and hands an error signal, the exception, to its caller. The caller must then catch it and decide what to do (that is the job of a `try` / `catch` block in Java, `try` / `except` in Python). If it does not, the current run stops: that was the case of the `ZeroDivisionError` from 2.1, an exception nobody had planned for. Every careful caller of `substring` therefore has to check the length before calling, and that cost, repeated everywhere, is what the Python version removes.

This idea applies to your own code too, not just to what languages provide. A customer search by name that finds nobody has no need to throw an exception: an empty list is a right answer, and the caller's loop simply runs zero times.

```
// ✗ no customer by that name? exception, and every caller must check
// before calling
if customerExists("Smith"):                 // to repeat before every call
    customers = findCustomers("Smith")   // otherwise 💥 NoSuchCustomer

// ✓ no customer? an empty list: the loop runs zero times, the error case
// is gone
customers = findCustomers("Smith")       // → []
for each customer in customers:
    show(customer.name)
```

The same idea works for a visitor who is not logged in. Rather than returning "nothing" (the `null` of 2.3), which has to be checked on every call, return a "guest" user with no rights at all: no caller has a special case left to test. But the idea stops where the absence is abnormal: if the account of a customer who just ordered cannot be found, answering "guest" would hide a bug.

Careful: you only define an error away when the case has a right answer. Up to 10 characters of "hi" is "hi". A search with no result is an empty list. But the 11th letter of "hi" does not exist, and the average price of an empty basket (€0 divided by 0 items, the calculation of 2.1) has no right answer either: there, the error must stay, and stay loud.

That noise has a name. **Fail fast** (a phrase popularized by Jim Shore in 2004) is the rule that a program should stop as soon as it meets a case with no right answer, rather than carry on with a wrong value. The error then blows up as close as possible to its cause, not three screens later.

Failing early has a tool, the **assertion**: a line that checks what must always be true, and stops everything otherwise, like `assert total >= 0`. An **invariant** is what must stay true whatever the program does: a till total is never negative, a paid order has an amount. The assertion does not check an input, which plans for a possible case like the empty list from earlier: it shouts if the impossible happens, so if the code has a bug. A function's contract (3.3) fits in three lines of this kind: what it requires on the way in, what it guarantees on the way out, what it leaves always true in between.

```
total = priceBeforeTax - discount
assert total >= 0, "negative total: calculation bug"
// the invariant, checked here
show(total)
// without the assert, -€12 would reach the receipt
```

Defining errors away and failing fast aim at two different errors. The error that comes from the outside world (a silent server, a missing file) is beyond our control and will come anyway: you plan for it, and when the case has a right

answer, you define it away. The error that comes from the code itself is a bug, like averaging an empty basket: you want to hear it at once, before a wrong total has travelled all the way to the receipt.

When the error stays: carry it, do not silence it

An exception that stays needs two things, a context and a destination. The context is what the message carries as it travels up: "file not found" says nothing, "config.json not found in /etc/till" is fixed in a minute. Each level adds what it knows and nobody below could know. The destination is where you catch it: business code raises and lets it rise, and you catch at the borders, where the program touches the outside (2.1): the screen, which turns it into a message for the human, the API, which answers with an error code, the log, which keeps everything. In between, an empty catch is the worst line in a program: the error vanishes without getting a right answer, the exact opposite of what this section calls defining the error away.

Go and Rust made another choice: the function returns its result or its error, as two values, and the caller has to look at which one it got, so a slip shows in the code instead of happening at run time. Finally, an error that travels up skips lines, among them the one that closed the file or returned the connection. The garbage collector (2.4) only frees memory, not an open file, a database connection or a lock: the block that runs whatever happens, `finally` in Java, `with` in Python, `defer` in Go, is there for that.

Defining errors away comes from John Ousterhout, who files it under a broader principle: "pull complexity downwards." It means absorbing the difficulty inside the function, the way `finalPrice` hid the four steps of the calculation, rather than pushing it onto its callers (3.2). Better for the function's author to struggle once, inside, than for a thousand callers to struggle, outside.

↳ A Philosophy of Software Design

3.5 Don’t repeat yourself, and make change easy

The VAT rate is written in three places in the program. The day it changes, you fix two and forget the third: each copy ages at its own pace. **DRY** (*Don’t Repeat Yourself*) is the principle that a given piece of knowledge, like the VAT rate, should live in only one place in the program. It is one of the most misread: it does not say "don’t copy-paste code". Its authors, Andrew Hunt and David Thomas, put it in one sentence: "**every piece of knowledge must have a single, unambiguous, authoritative representation within a system**". The word that counts is knowledge, not code, and two counterintuitive consequences follow.

Identical code is not always duplication. If two fragments look alike by chance and will evolve for unrelated reasons, merging them couples them wrongly: they are now tied together, and touching one forces you to worry about the other.

An item’s price is validated with `price > 0`, its stock quantity with `quantity > 0`: it is tempting to merge both into a single `isPositive()`. But tomorrow stock must allow 0 (out of stock), the price must not: they were two rules, identical by coincidence.

Programmer Sandi Metz draws a maxim from it, in her 2016 article *The Wrong Abstraction*: "duplication is far cheaper than the wrong abstraction". An abstraction is a grouping of cases under a single name, here `isPositive()`. With two copies, the day stock must accept 0, you edit the stock copy, and you are done. With a single `isPositive()`, you cannot change it without breaking the price: so you add a special case ("if it is stock, accept 0"), then another for the next case, and the function becomes a tangle nobody dares touch. When in doubt, keep the copy.

You also repeat yourself without copying a single line. The same knowledge leaks into a second place in three common ways. First, a number written in the code and repeated in a text. The code grants free delivery from €100, and the site banner announces it too: the day the threshold moves to €120, you remember the code and forget the banner. The banner should read the threshold

from the code instead of retyping it. Second, the shape of a database record (name, price, stock), written once in the database and again in the code; the cure is to generate one from the other. Third, a comment restating what the code already does, the very comment-paraphrase trap from 3.3. No copy-paste, yet two truths to keep in sync.

Whatever form the repetition takes, the cure is the same: each piece of knowledge gets one authoritative home, and the rest goes there to look it up instead of keeping its own copy on the side.

DRY sits under a broader idea. **ETC** (*Easier To Change*) says that at every choice, you ask one question: "will this make the system easier, or harder, to change?" It is a compass more than a rule. Take a program that sends alerts by email or text message. Each way of sending is a channel, and the piece of code that knows how to send through a channel is a notifier. Tomorrow, a third channel will be wanted: how do you add it without reopening all the code?

```
// ✗ every new channel reopens the function
if (channel == "email") ... else if (channel == "sms") ...

// ✓ a dictionary {channel → notifier}: a new channel = one entry, the
// logic stays put
notifiers[channel].send(message)
```

This way of doing it has a name, Strategy: each notifier is an interchangeable way of doing the same thing, and the dictionary picks which. We come back to it in 4.2, with other proven solutions of the same kind. Good names, deep functions, DRY itself are all special cases of ETC: each makes code easier to change. What remains is code that already exists: how do you make it easier to change without breaking it?

↳ The Pragmatic Programmer

3.6 Refactoring is changing form without changing behavior

Refactoring is rewriting a sentence so it reads better without changing what it says. In code: reshaping the internal structure without changing observable behavior, that is, what anyone using the code can see without knowing its inside (the results returned, what shows on screen, what gets saved). The proof is a test, a small program that calls the code and checks its answer: if it passed before, it still passes after. Otherwise it is no longer a refactoring, it is a modification.

You move in small safe steps: rename a variable, pull three lines into their own function, check, repeat. "Smells" guide those steps, the signs that betray badly structured code. One of the most common, the *Data Clump*: a group of arguments that always travel together, like the street, the city and the postal code passed one after another to every function. The remedy gathers them into a single object, `address`, which the *Refactoring* catalog calls *Introduce Parameter Object*.

```
sendParcel(street, city, zip, country)
// ✗ 4 arguments glued together everywhere
sendParcel(address)
// ✓ they were one concept: an Address
```

Four words that never part end up calling for a single one: language did it long before we did, by inventing the word *address*. Code is only catching up.

The key idea of refactoring fits in a line from programmer Kent Beck: **make the change easy, then make the easy change.** It is the ETC compass in two steps. It is counterintuitive, because you start by delivering none of what was asked, and Beck warns that this first step can be the hard one. Need to wire up PayPal, when the code only knows bank cards? First you turn the `if (method == "card")` into a dictionary {method → payer} (the payer being the piece of code that knows how to charge through that method), like the notification channels in 3.5, behavior untouched; then you add PayPal almost for free.

YAGNI (*You Aren't Gonna Need It*) is the counterweight to DRY and ETC: the rule, from Kent Beck's team, that advises against building ahead what nobody has asked for yet. As long as there is only the bank card, a dictionary of payment methods is over-engineering: code made complicated as a precaution, to read and to test, for a need that often never comes, or comes in another shape. You refactor when the need arrives, not before.

Martin Fowler, author of *Refactoring*, takes patience further. The **rule of three**, which Fowler owes to Don Roberts, comes in three steps: the first time you just write it, the second time you copy it and wince, the third time you refactor. Why wait? Two cases that look alike do not necessarily have the same reason to exist, like price and stock in 3.5, checked the same way but subject to different rules. Card and PayPal, though, serve the same obvious purpose, taking payment: here two are enough. As soon as the common ground is certain, you can group them, at two cases as at three.

You only refactor what you have understood, and the code to touch is almost always someone else's, or your own from six months ago, which comes to the same thing. Take the till whose VAT rate you are asked to change, and which you did not write. Unknown code is read from top to bottom, like this book:

- names and signatures first, which tell the intent without opening the functions (3.1);
- a piece of data followed from input to output, by tracing it by hand (2.1);
- the tests, examples of use written by someone who knew;
- the history, `git blame` and the commit message, which tell why a line is there (3.3).

You do not read everything: you read until you can explain out loud what the piece you are about to touch does, and not one function more.

What remains is refactoring without breaking anything. The small steps are only safe with a safety net. Fowler hammers it: without a suite of tests confirming at each step that observable behavior hasn't moved, refactoring becomes a blind bet (more on that in 6.1). Those tests must watch observable behavior, not internal structure: a test glued to the details breaks at every re-

name, and soon nobody refactors. What if the code has no tests at all? You start by writing some, not to check that it is right, but to pin down what it does today, bugs included. If one of them fails after an edit, the behavior has moved. Michael Feathers calls these *characterization tests*.

A second safeguard is about how you organize the work: the image of the two hats, which Fowler owes to Kent Beck. You wear the one that adds behavior or the one that changes the shape, never both at once, and each hat gets its own commit, its own record in Git (6.3). A three-line fix drowned in two hundred moved lines cannot be reviewed. The colleague doing the code review (reading the changes before they enter the project) approves it with eyes shut. And the day the bug comes back, the history no longer says which commit brought it.

And the daily reflex fits in one image, *Clean Code*'s **boy-scout rule**: like the scout who leaves the campsite cleaner than he found it, always leave the file a little cleaner than you found it. Not scrub everything, just one small fix on the way, a vague name corrected, a long function cut in two, in its own commit, refactoring hat on: cleanliness becomes a habit, instead of a spring-clean you postpone forever.

↳ `Refactoring`

↳ WHICH LEADS TO CHAPTER 4

A name that states the intent, a function that hides a lot behind a small surface, a comment kept for the why, an error made to disappear instead of caught everywhere, one place that is the authority for each piece of knowledge, small steps guarded by tests to change the shape without changing the behavior: the chapter laid down six habits. One compass ties them together: keep the code easy to change, without building anything ahead of need. We can now write a clean function, a clean file.

But a thousand clean functions still do not make a clear system, just as a thousand well-turned sentences do not make a book: they lack a plan. **Architecture** *is the overall plan of a program: it decides what depends on what, and where the boundaries between its parts run. Shortcuts taken here and there pile up into* **technical debt***: each shortcut makes the next change more expensive, like interest on a loan (the image dates from 1992, from Ward Cunningham). Without an overall plan, nobody knows where to repay it, and every change risks breaking something else.*

"Programs must be written for people to read, and only incidentally for machines to execute." — Abelson & Sussman, *Structure and Interpretation of Computer Programs*, 1985

4 Giving the system a shape

architecture

Section 1.2 looked at the machine from above, like a city seen from the sky. Here we look down on your program: its districts, meaning its big groups of files, the exchanges between one group and another, and what can be torn down without the rest collapsing.

As soon as one group of code uses the work of another, it depends on it. A **dependency** is a piece of code that another needs in order to work. When it comes from elsewhere, like a downloaded library (2.4), its bugs and updates become yours: if it changes or vanishes, your program breaks with it.

A famous case: in 2016, the author of `left-pad`, a small JavaScript library, pulled it from the public registry (the site where JavaScript libraries are published). It was eleven lines long and padded a text with spaces on the left. That same day, thousands of programs failed to install. Most had never named it: one of their own libraries depended on it.

A dependency has dependencies of its own, and you inherit the whole chain. Hence the habit of pinning the exact versions in a **lock file**, a list that records the precise version of each library. That way, an update cannot break the program unannounced.

At the scale of the whole system, one question dominates: who depends on whom, and what can change without breaking everything? Take a shop, with its discount rule (10% above €100) and its database, where it keeps its orders (we open it in 4.6). If you switch databases, must you rewrite the discount rule? Ideally not, because the rule should not depend on where the data is kept. Architecture, that overall plan, sets the shape of the dependencies before they decide themselves, in disorder. Six questions carve it up.

1. How does one piece of code reuse another's work? (4.1)
2. How do you recognize the problems others have solved a thousand times before you? (4.2)
3. Which way should the dependencies point? (4.3)

4. How do you decide when two good ideas pull against each other? (4.4)
5. Who keeps the shape of the whole? (4.5)
6. And where does the data live once the program stops? (4.6)

The fifth has a short answer: a single mind. The sixth finally opens the room the whole chapter keeps pointing at without entering: the database.



Figure 37 : Composing pieces with clean joints holds; one rigid monolithic block cracks at the first change.

4.1 Compose rather than inherit

Three words first, because the rest of the chapter leans on them: those of object-oriented programming, the OOP seen in 2.7.

- A **class** is a mold: the `User` class states that a user has a name, an email, and knows how to sign in. It holds nobody, the way a cake pan holds no cake.
- Making a copy from the mold is called **instantiating** the class, which is what the `new` keyword does in many languages (Java, C#, PHP, JavaScript).
- An **instance** is the copy made from a class: the object seen in section 2.2, with values of its own (Smith, smith@example.com).

```
class User { name; email; signIn() }
// the mold: it describes, it holds nobody

u = new User("Smith", "smith@example.com")
// an instance: somebody, with values of their own
```

Take an admin: a user who can also ban others. Copying name, email and sign-in into a second mold would be absurd, hence architecture's first question: how does one mold reuse another's work? Two answers dominate.

- **Inheriting** says "an admin *is a* user", which the `extends` keyword writes: the child mold (`Admin`) takes everything from the parent mold (`User`) at once, without asking. It can then add its own elements (the power to ban) or rewrite some of the parent's (a stricter sign-in).
- **Composing** means assembling: "a user *has a* way of signing in" means it holds a separate object, from another mold, which can be replaced without touching the `User` mold. Today, sign-in is by password, with a `ByPassword` object. The day the site moves to a code sent by text, you replace it with a `ByText` object, and the `User` mold does not move.

```
// inherit: the child takes everything, adds, rewrites
class Admin extends User {
  ban(other) { ... }              // added
  signIn() { ... }                // rewritten, stricter
}

// compose: the user holds an object from another mold
class User { signInMethod; signIn() { signInMethod.check() } }
u.signInMethod = new ByPassword()   // today
u.signInMethod = new ByText()
// tomorrow: the User mold did not move
```

The first reflex for reuse is inheritance, because it costs a single keyword: no interface (2.3) to write, no object to plug in. But it is all or nothing: the child takes everything from the parent, whether it wants to or not. The classic example from the book *Head First Design Patterns* shows it with ducks instead of users. A parent mold `Duck` writes `fly()` once, and every duck inherits it. Then the rubber duck shows up and inherits it too, though it cannot fly. You can rewrite `fly()` as empty inside the rubber duck, but someone must re-

member to do it for every wooden, plastic or plush duck that comes later. And the day someone improves `fly()` in the parent, every duck changes at once, including the ones nobody had in mind.

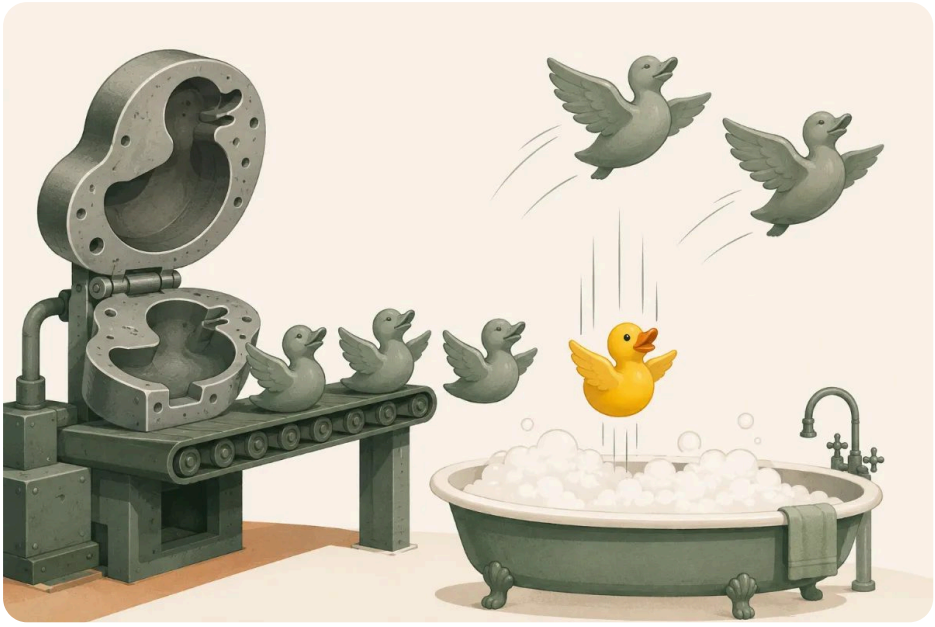


Figure 38 : They all come out of the same mold, so they all inherit flight. The rubber duck does too, and it believes it.

Composition takes flight out of the duck's mold: `fly()` hands the work to a separate object stored inside the duck, which you swap (real flight, or none) without touching the duck, even while the program is running.

```
// ✗ inheritance: flying is frozen in the parent
class RubberDuck extends Duck { }  // inherits fly()… but cannot fly!

// ✓ composition: the duck hands flying to an object stored inside it,
// swappable
class Duck { flyBehavior; fly() { flyBehavior.fly() } }
duck.flyBehavior = new CannotFly()   // new object, same duck
```

The reflex when you code: before writing `extends`, say the sentence "X is a Y" and ask whether it will stay true for every X, including next year's. If so, inheritance is allowed. If "X has a Y" is the more natural thing to say, or if "is a" is only sometimes true, it is composition. The test on six cases:

- An admin is a user: true for all of them, tomorrow's included. Inheritance is allowed.

- A transfer is a bank operation: like a withdrawal and a deposit, without exception. Inheritance is allowed.
- A square is a rectangle: true in geometry. But changing a rectangle's width leaves its height alone, while a square must follow: the parent promises what the child cannot keep. Inheritance is a bad idea.
- An employee is a person: true today, but they may become a manager next year, and the person stays the same. The role changes, not the person: a person has a role. Composition.
- A car is an engine? Absurd. A car has an engine. Composition.
- An order is a customer? Absurd. An order has a customer. Composition.

One difficulty remains: at the start, you do not always know the rubber duck is coming. Two clues help guess it. What varies is an action (flying, signing in) rather than an identity (a duck), and you can already imagine that action differing from one duck to another. With no clue, you start simple with inheritance and refactor the day the case arrives, since moving from inheritance to composition is a classic refactoring (3.6).

When in doubt, compose. Inheritance is not forbidden, but it ties each child to its parent: a change in the parent can break a child nobody touched. This is the *fragile base class* problem. Composition costs a few more lines, to hand the work over to the object, and in return each object can be changed alone. The maxim "Favor composition over inheritance" comes from the book *Design Patterns* (1994), which popularized **patterns**: proven answers to problems that keep coming back. Section 4.2 is devoted to them.

↳ Head First Design Patterns · Design Patterns (GoF)

4.2 A pattern answers a need, not a goal

A design pattern is not decoration you slap on to look serious: it is a proven answer to a recurring problem. A carpenter does not invent a way to join two boards for every piece of furniture: they know the dovetail (trapezoid teeth that interlock and resist being pulled apart), the mortise and tenon (a tongue that slides into a slot), and they take the joint the constraint calls for. A pattern

is that: a named joint, proven by others before you. The idea comes from the architect Christopher Alexander (*A Pattern Language*, 1977), who wrote it for houses and towns. The authors of *Design Patterns* carried it over to software.



Figure 39 : A pattern is a named joint, proven before you: you pick it for the constraint, you do not reinvent it.

Three patterns turn up in many programs, and you have already met one:

- **Strategy**: swap one way of doing a task for another while the program runs, exactly what we just did by replacing the duck's flight;
- **Observer**: notify a list of subscribers when a piece of data changes;
- **Decorator**: add a behavior around an object, by wrapping it, without inheriting from it.

None is wizardry. For the Observer, it is enough to know that a function is a value like any other, so you can put several in a list (section 2.4). The watched object, called the subject, keeps that list of functions and calls them all on every change. You already practice it: `addEventListener('click', fn)` subscribes a function to clicks.

```
subject.subscribers = [refreshView, sendMail]
subject.change(state) { subscribers.forEach(fn => fn(state)) }
// forEach: calls each function in the list
subject.change("order paid")
// → refreshView and sendMail receive "order paid"
```

The Decorator is the sticky note from section 2.4, applied to a whole object: one object wraps another and adds a step before or after each call, without the caller noticing.

```
service = new Log(new Cache(new Service()))
// the call goes through Log (it records), then Cache (it keeps answers
// already computed), then Service (the real work)
```

The **GoF** (the "Gang of Four", the four authors of the founding *Design Patterns* book) sorts its 23 patterns into three families, and those three families beat learning the 23 names by heart:

- **Creational** (how to create objects)
  - Factory Method: picks which class to instantiate;
  - Singleton: a single instance for the whole program, which any code can change, hence its bad reputation;
  - Builder: assembles a complex object step by step.
- **Structural** (how to assemble objects)
  - Decorator (seen above);
  - Adapter: translates one interface into another;
  - Composite: treats a group of objects as one.
- **Behavioral** (how to make them communicate)

- Strategy and Observer (seen above);
- Command: an action stored in an object, which can be queued or undone.

You don't memorize 23 ready-made solutions: you ask one question, "is my problem to create, to assemble, or to communicate?", and you only search one family, five to eleven patterns instead of twenty-three (the list above shows only the most common ones). Your language matters too: where functions are values, a Strategy boils down to passing a function, like the one that compares two items when you sort a list (2.4). In 1996, Peter Norvig counted that 16 of the GoF's 23 patterns become invisible or simpler in Lisp, a language built on functions.

Above all 23 patterns, the GoF sets two founding principles: composition over inheritance (4.1), and **program to an interface, not an implementation.** The interface (2.3) says what an object can do, the implementation is the specific class that actually does it. This is the electrical socket from section 2.6: the wall knows nothing about the appliance, only the shape of the holes. Your code works with what an object can do, never with what it is.

```
// ✗ to an implementation: the code is welded to a specific class
export(doc) { new PDF().write(doc) }          // can only do PDF

// ✓ to an interface: "something that can write()"
export(doc, output) { output.write(doc) }
// PDF, CSV, HTML…: what it DOES, not what it IS
```

The day it must write somewhere other than a PDF, you do not reopen `export()`: you hand it something else. The function never needed to know what it was writing into. Handing a function what it needs (here, the output), instead of letting it build that itself, is called **dependency injection.** The name is more impressive than the idea.

And the final rule: a pattern is never the goal; you do not apply one, you **recognize** it when the need calls for it. The number-one danger is over-application, forcing a pattern where a simple solution would do. A Factory that only hides a single `new`, choosing nothing, is a dovetail where there is only one board.

▾ See all 23 patterns one by one, for the curious (optional)

This book details only three, on purpose: three families are easier to remember than twenty-three names. If you want the whole catalog, each pattern with its problem, its diagram and the moment it earns its place, one of the clearest is Refactoring Guru's, free to read.

↳ Design Patterns (GoF) · Head First

4.3 Depend on abstractions, invert the dependencies

We have learned to program to an interface. One question remains: who draws the socket? Everything turns on the direction of the dependencies. Depending on something means needing it to work, and therefore moving when it moves (start of the chapter). Normally the business code calls the database: the business is the one that depends.

Clean Architecture splits code into two camps. On one side the **business rules**, whatever is specific to your activity: how a discount is computed, when an order becomes valid, who may cancel it. That changes little, and that is where the value sits. On the other side the database, the framework (the ready-made code skeleton the application is built on), the screen. That changes often.

The **dependency rule** of *Clean Architecture* forbids one single thing: the business code depending on the technology. Otherwise what was meant to stay stable moves every time the technology moves. Hence the inversion: the business code declares an interface, and the technology conforms to it. That interface is the abstraction of the title: the list of what the business code needs, without saying who provides it.

In your home, no vacuum cleaner maker decides the shape of the sockets: the house sets the shape, and every appliance conforms. If each vacuum imposed its own, you would have to break the wall open with every purchase. Business

code plays the wall and draws the socket: that is the interface. The database plays the vacuum cleaner.

```
// the business code declares WHAT it needs: an interface, that is the
// abstraction of the title
interface OrderRepository { save(order) }
// the technology (database, email sending…) implements it; the business
// code knows nothing of it
// → you swap databases without touching a line of business code
```

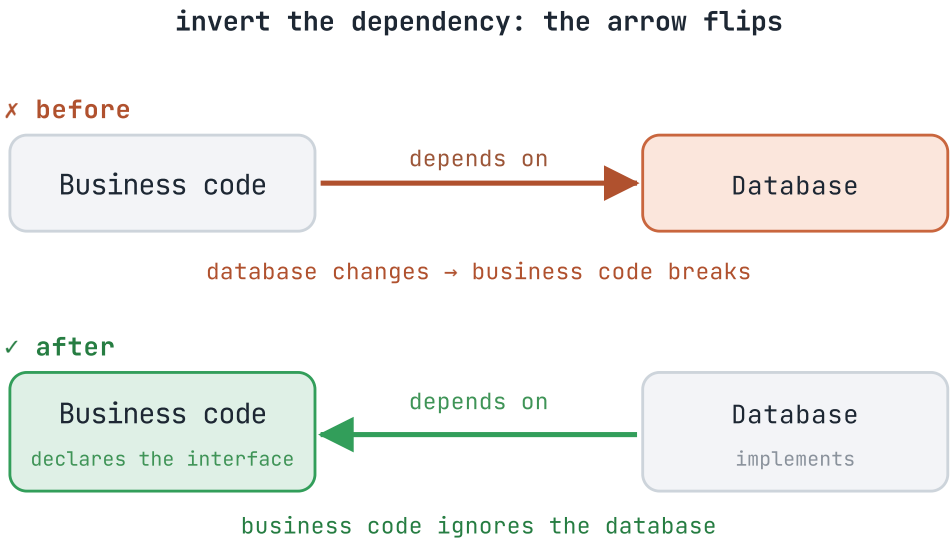


Figure 40 : Business code declares the interface, the database conforms to it. It no longer depends on anything volatile.

It is the "I need something that can do X" of section 2.6, scaled up to a whole system: boundaries protect what matters from the rest.

Robert Martin, the author of *Clean Architecture*, goes further with a striking line: **the database and the framework are details**, just like the brand of the electrical wiring in a house. Your business code should not even know that its orders live in MySQL, one database among others. Hence the goal he sets: **a good architect maximizes the number of decisions not made**. The less you commit to details early, the more options you keep open once you actually know more.

Circles and ports

Applied to the whole system, the dependency rule gives *Clean Architecture*'s four concentric circles, heirs to Alistair Cockburn's **hexagonal architecture** (2005), also known as **ports and adapters**. From the center outward:

1. the business core: the rules the company would apply with a notebook and a pen if the software did not exist;
2. the application rules: the sequence of steps of one complete action, check the stock, charge the card, send the confirmation;
3. the adapters: the code that joins the inside to the outside, the code that receives the browser's requests and the code that reads and writes the database;
4. the technical details: MySQL, the framework, third-party APIs, that is, another company's software (payment, text messages) that you call over the network.

One rule holds it together: an outer circle may depend on an inner one, never the reverse. When the inside (the two central circles) needs the outside (the two outer circles), to save or to send an email, it goes through an interface it draws itself.

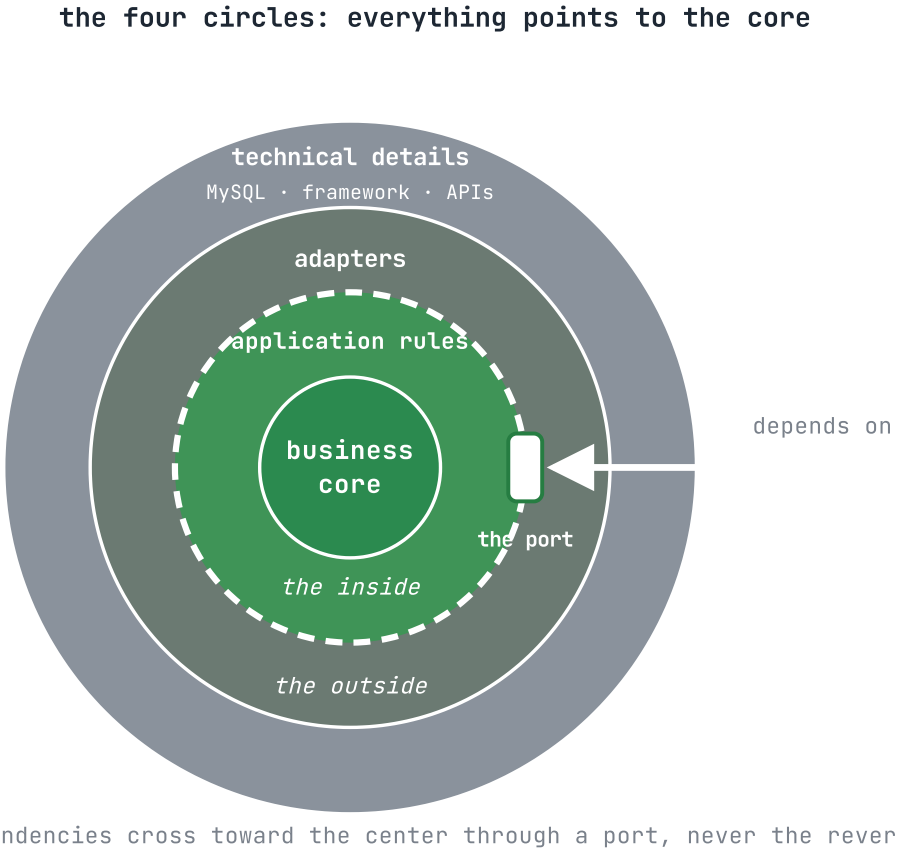


Figure 41 : Four circles, one rule: the outside depends on the inside, never the reverse. The port (the white socket) sits on the boundary of the inside (the green circles), and the adapter, just outside, plugs into it. A real system exposes several, one per need.

In practice: the core defines an interface `OrderRepository` with a `save()` method. That is the port: the socket the core exposes outward, whose shape it defines. The adapter `MySQLOrderRepository` implements that port. The test adapter `InMemoryOrderRepository` implements the same port with a plain in-memory list, so the tests run without any database. The core code calls `repo.save(order)` without knowing what is plugged in. Moving from MySQL to PostgreSQL, another database of the same kind: plug in a different adapter, the core does not change. The split costs one more interface: keep it for the boundaries that really move, or it becomes the over-application of section 4.2.

One question remains: who plugs the adapter in? The startup code of the program, the one place that knows MySQL: it creates the adapter and hands it to the core as an argument, which is the dependency injection of 4.2. The injection is the technique, the inversion is the principle: whoever owns the interface. At run time, the calls still go from the core to the database. Only the dependency, meaning who must know whom, is flipped.

```
// the port: defined by the core
interface OrderRepository { save(order) }

// two adapters for the same port
class MySqlOrderRepository implements OrderRepository { save(o) { /* writes
to MySQL */ } }
class InMemoryOrderRepository implements OrderRepository { save(o) {
list.add(o) } }

// the core only knows the port
class Shop {
  constructor(repo) { this.repo = repo }
  // constructor runs at new, this.repo is a locker of this object: the
  // repository comes from outside, this is the injection
  place(order) { this.repo.save(order) }
}

// the startup code plugs the adapter in
shop = new Shop(new MySqlOrderRepository())
// for real, in front of customers
shop = new Shop(new InMemoryOrderRepository())  // in tests
```

Even a language gets plugged into

Languages themselves apply the same rule, and Python shows it most simply. It does not know your `Basket` object, and yet `len(Basket())` works: `len` gives the length of a thing (the number of letters in a text, of items in a

list), and it does so by looking in the object's mold for a method with a name set in advance, `__len__`. The language provides the interface, your object conforms to it.

```
class Basket:
    def __init__(self):
        self.items = ["bread", "milk", "coffee"]
    def __len__(self):
        return len(self.items)

len(Basket())   # 3
```

Here, `class Basket` opens the mold of a basket, `__init__` runs each time a basket is built and fills it with three articles, and `self` stands for the basket being measured. The `len` in the `return` measures the inner list, not the basket: this is not recursion (2.5). `__len__` is a method, a function stored in the object's mold (the double underscores mark a name reserved by Python): when you write `len(something)`, Python calls it.

The same interface exists for `for`, `+` or `print`, each with its own reserved name. *Fluent Python* (Luciano Ramalho) calls this set Python's data model: the interfaces of the language itself. The lesson: you do not configure a language, you **plug into** it, following the interfaces it provides.

The D in SOLID, and the other four

Dependency inversion carries a letter: it is the D of **SOLID**, five design principles gathered under one acronym.

- **S** (*Single Responsibility*): a module (a source file) answers to one **actor** only, one group of people who drive the same kind of changes;
- **O** (*Open/Closed*): open to extension, closed to modification, like the `export()` from section 4.2 we never had to reopen to write elsewhere;
- **L** (*Liskov Substitution*): a child mold must be able to replace its parent mold without surprises, which the rubber duck from section 4.1, inheriting `fly()` without flying, cannot do;

- **I** (*Interface Segregation*): no code should depend on what it does not use: several small interfaces beat one huge one, otherwise it is a socket with thirty holes for an appliance that uses two;
- **D** (*Dependency Inversion*): business code does not depend on the technology, both depend on an interface, and the business code is the one that draws it.

The example for the S is Martin's own: an `Employee` class that computes the pay, produces the hours report and saves itself to the database. Three tasks, but three actors: accounting for the pay, HR for the report, the database administrators for the save. The day accounting has a calculation changed that the report also uses, HR's report breaks without anyone asking for it.

It is the most misunderstood of the five. Many reduce it to "a function does one thing", which is a separate, lower-level rule from *Clean Code*. Martin corrects this himself in *Clean Architecture*, chapter 7: what you count is the number of groups served, not the number of things the code does.

Between two objects: the Law of Demeter

The same idea holds at the smallest scale, between two objects: the less an object knows about the others, the less it breaks when they change. The **Law of Demeter** says "only talk to your immediate neighbors": an object only calls its own methods, those of the objects handed to it and those of the objects it keeps inside. No chain like `order.getCustomer().getAddress().getCity()`, which dives into the internal structure of three strangers: the day one of them changes shape, the caller breaks. The law targets objects that hide their inside, not a plain data structure, which exists to be read.

Two classics name the same flaw. *Clean Code* calls a chain of calls coupled like railway cars a **train wreck**. Each car is a **getter**, a method that only hands back a value. *Refactoring* calls it a **message chain** and fixes it with Hide Delegate, ex-

posing a method that says what you want instead of the path to reach it. At the scale of a system as between two objects, the rule does not change: depend only on what you are entitled to know.

```
// ✗ a chain that walks through three strangers
order.getCustomer().getAddress().getCity()

// ✓ ask for what you want, not for the path to it
order.deliveryCity()
```

↳ Clean Architecture · Clean Code · Refactoring · Fluent Python

4.4 No "best practice", only the least-bad trade-off

The higher up you look at a system, the fewer universal answers there are. The book *Software Architecture: The Hard Parts* (Neal Ford and co-authors) puts it bluntly: every architecture problem is a snowflake, no two alike. The skill is not picking the right pattern, it is **weighing** trade-offs. Two tools for that.

The first tool, the **architecture quantum**: the smallest piece of a system you can put online and test alone, and that can fall without dragging anything else down. A service is one part of the application put online on its own. Two services sharing the same database are like two flats behind a single circuit breaker: one trips the breaker, the other is in the dark. They are a single quantum.

The test, for any diagram: "if this piece changes or crashes, how many others fall with it?" That piece and everything that falls with it make up your quantum. The question does not need a big system to be useful: on a first project it already applies to one file that three pages call. The shared database is not the only link: a service that waits for another's answer before it can go on (the call

is said to be *synchronous*) stays stuck when the other falls, so the two are still a single quantum. Twenty "microservices", deliberately tiny services, that share a database or wait for one another? A single quantum.

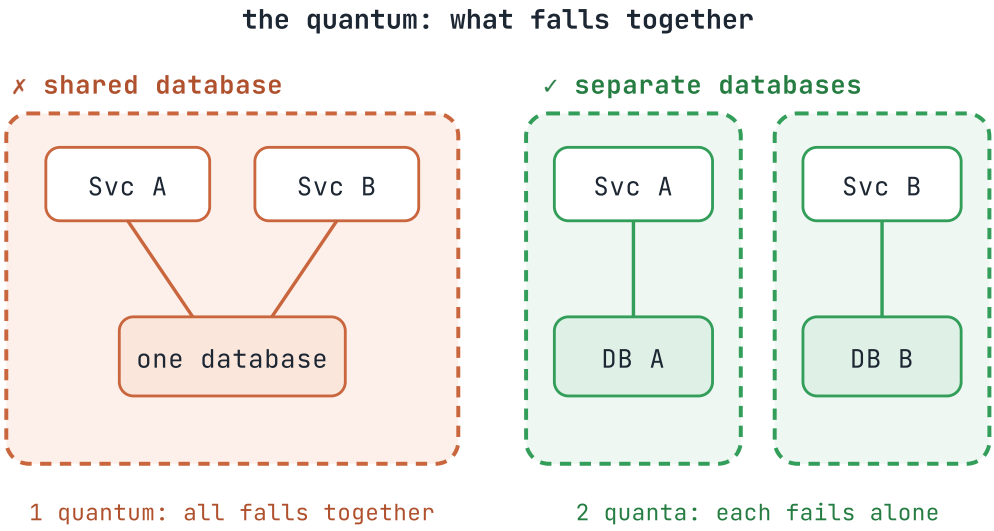


Figure 42 : Each dashed frame is a quantum. In orange, the shared database welds the two services into one. In green, each service falls alone.

The second tool: "**reuse is coupling**", coupling being the tie that forces two pieces of code to change together. Take the file that three pages call: the quantum asked what falls with it, here the question becomes what changes with it. Change it for the first page, and the other two change too, without anyone asking. At system scale it is the same: sharing a business class across services propagates every change everywhere at once.

Hence a counterintuitive reflex: you sometimes duplicate on purpose. Share only what is genuinely one piece of knowledge that must stay consistent (the DRY of section 3.5); for two bits that merely look alike but will evolve apart, a little copying beats a bad coupling.

Between a small quantum and sharing, no principle decides. A small quantum buys **independence**, sharing buys **consistency**, and each is paid for: one in duplication and network calls, since separate parts have to talk to each other at a distance (chapter 5), the other in coupling. You choose case by case, by what matters most here: no rule, only the least-bad trade-off.

↳ Software Architecture: The Hard Parts

4.5 Conceptual integrity: one mind

Weighing trade-offs case by case is not enough if everyone weighs alone: the finest architecture dies if forty teams stack their ideas without coordination. The textbook case is the API. An **API** (application programming interface) is an application's front counter: other programs drop off their requests there, with a form to fill in and a set response format (section 7.2 comes back to it). Forty teams that never talk, and that counter becomes Asterix's "place that sends you mad", the one with permit A38: every desk demands its own form, here the identifier changes name, there the date changes format. Every desk works. The whole sends you mad.

```
// GET /address: the request at the counter, →: its reply
// ✗ three teams, three names for the same id, three ways to write the same
June 9 (the last in seconds since 1970)
GET /users   → { id,     "2026-06-09" }
GET /orders  → { userId, "06/09/2026" }
GET /cart    → { uid,    1780963200 }

// ✓ one vocabulary, one format: the caller can guess everything
GET /users · /orders · /cart → { id, "2026-06-09" }
```

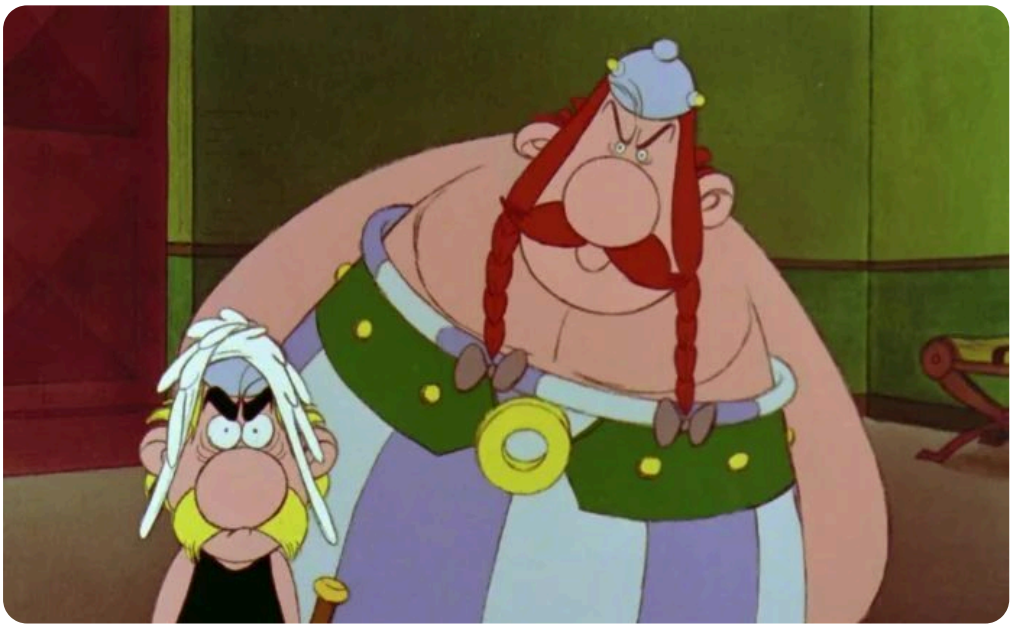


Figure 43 : Permit A38: every desk sends you to the next one (*The Twelve Tasks of Asterix*, 1976).

Against this disorder, *The Mythical Man-Month*, by Fred Brooks, sets this rule: "conceptual integrity is the most important consideration in system design." Conceptual integrity means a system that seems thought up in one piece: the same names, the same formats, the same rules everywhere. It must come from a single mind, or a very small group, or it becomes the tower of Babel.

One mind doesn't mean one person coding it all: that small group decides the form, the others fill it in. And Brooks insists: "form is liberating". Once the structure is fixed, everyone knows where their piece fits, and codes faster, not slower.

The model lives in the code

But a single vision does not survive in a document: within the first few weeks, the diagram filed in the team's documentation no longer matches the code, because the code moves and the diagram doesn't. So the vision has to live in the code. Eric Evans's **Model-Driven Design** asks that the model, meaning the description of the business with its objects and rules, be written directly into the code.

Take a business rule: "a paid order can no longer be cancelled". As long as it stays scattered, every screen with a Cancel button redoes the check on its own, and one of them will eventually forget it. With Evans, the rule lives in the `Order` object itself: call `order.cancel()` on a paid order, the object refuses. The status is the enumeration of 2.3, and `cancel()` is only allowed from certain of its states: the order is a **state machine**, an object that moves from one state to another only through transitions written once, in the code. That object does more than carry the fields of a database row: it knows the

rules of a real order. Martin Fowler calls an object reduced to its fields an *Anemic Domain Model* and counts it as a flaw. The model no longer sits beside the code: it is the code.

```
// ✗ the scattered rule: every screen redoes it (and one forgets)
if (order.status != "paid") { order.status = "cancelled" }

// ✓ the rule lives in the object: the screen asks, the object refuses
class Order { cancel() { if (paid) refuse("already paid") else status =
"cancelled" } }
```

Words matter as much as rules. In a shipping company, the business expert (the person who knows the company's real activity) says "cargo", "port of call", "itinerary", while the code says "row 42", "Boolean", "status2". Every meeting then becomes a translation, and translation always loses something. Evans asks the team and the expert to agree on the exact words of the business and to use them everywhere, in meetings as in code: that is the **Ubiquitous Language** ("language present everywhere"), one shared language. It is the "one word per idea" of section 3.1, extended to the people who run the business. "Everywhere" means within one team's domain: section 7.4 shows where that language stops.

Entity or value object: Evans's sorting

A model made of objects needs sorting. Evans distinguishes two kinds of objects, and a single question is enough to sort them: this one, do I follow it over time or do I replace it? It is the value/reference distinction from section 2.2, carried over to the business.

- An **entity** has an identity that outlives its contents: customer 42 is still customer 42 after moving house and changing name, you follow it over time.
- A **value object** is nothing but its contents: two €10 notes are interchangeable, two identical addresses are the same address.

Ask that question of every object in the application: an invoice's amount, you replace it, a value object. The invoice itself, for the accountants who follow it over time under its number, is an entity. One question, and every object finds

its place, but the answer depends on who is using it: for the customer who only keeps a copy, the same invoice is just a document, a value.

And that sorting drives two concrete moves.

- **Comparing.** Two entities compare by identity: two customers with the same name are still two customers, and the code compares their `id`. Two value objects compare by content, field by field.
- **Changing.** A value object is not modified, it is replaced. Editing an address shared by two orders means falling into the reference trap: both orders move house at once (2.2). Hand the right order a fresh address, and the trap is gone.

One consequence of "the model is the code" remains, and it is uncomfortable. Many teams split the roles: the architect draws the plans, the developers build them, and Brooks himself defended that split. But if the model is the code, it no longer holds: designing means coding. The architect who never touches the keyboard ends up drawing plans that cannot be built. Evans calls this **Hands-on Modelers**, designers who keep their hands in the code: the same head holds the pencil and the keyboard.

↳ The Mythical Man-Month · Domain-Driven Design · Learning DDD

4.6 Store to find again: the database

The chapter has talked about the database without opening it: a detail to keep at arm's length in 4.3, a piece two services should not share in 4.4. Let us look inside. The lockers of memory (RAM) empty out when the program stops or the power goes out. Yet an order placed at 2 pm must still exist tomorrow morning: it has to be written to disk, into a file (an output of the program, 2.1). So why not a plain file?

Three troubles a file cannot settle

A text file holds a list of orders perfectly well, until the day three troubles arrive, often in the same week:

- Two programs write into it at the same time, and their lines get tangled.
- The power fails in the middle of a write, and the file is left half-written, with nobody the wiser.
- Finding customer Smith's order means reading the file from start to finish: the unsorted notebook of 1.3, `O(n)`, reread for every question.

A **database** is a set of data kept on disk by a specialized program that, unlike a text file, keeps three promises:

- several programs can use it without tangling their writes;
- each change is written in full or not at all;
- one row can be found without rereading everything.

That program is called the database engine: PostgreSQL and MySQL are two examples, SQLite a third, which runs inside your own program. Your code talks to the engine instead of touching the file.

The first two promises rest on the **transaction**: a group of changes that succeeds in full or not at all. Keeping the writes of several programs from mixing is **isolation**. All or nothing is **atomicity**: recording an order and taking the item out of stock, both steps or neither. These two words match two of the four letters of the acronym ACID, which section 6.7 details, with a trap in store. The third promise, finding one row without rereading everything, has nothing to do with transactions: it is the one you rely on at every read, and it is the one that matters here.

The grid and the key: the relational model

First, how to store. The relational model, laid down in 1970 by Edgar Codd, stores each kind of thing in a **table**: a grid with fixed columns, one row per thing, a Customers table, an Orders table. (Nothing to do with the hash table of 1.3: same word, different thing.) Each row carries a number of its own, its **key**: what you hand over to get the row. An order does not copy its customer: it keeps their key (called a foreign key). Order 7 does not contain "Smith" but 42, the number of Smith's row in Customers.

Gluing an order and its customer back together when reading is a **join**: for each order, go fetch the customer row that carries that number.

The list of tables, of their columns and of each column's type forms the **schema**: the shape of the data, decided before the first row, which the engine enforces on every write. These rules are declared as constraints: a column that cannot stay empty (NOT NULL), a value that can appear only once (UNIQUE), a foreign key that must point to a row that exists. A rule checked only in the application can be bypassed: a script, another program, a bug, and it is gone. Set in the engine, it applies to every write going through it, wherever it comes from.

Normalization is the DRY of section 3.5 applied to data: each fact is written once, the customer's phone number in the customer's row, never in the orders. Otherwise the customer gets a new phone number, and their orders contradict each other: some keep the old one, like a shelf price that differs from the checkout price.

You can also repeat on purpose to read faster: that is denormalization. Every write gets heavier, and section 6.6 shows that trade.

An index costs what a sort costs

Next, how to search. Finding Smith among ten million customers forces the engine to read the whole table, the unsorted notebook. An **index** is the sorted directory of section 1.3, kept up to date by the engine for a given column, so that a search on that column drops to a few steps instead of ten million. Most often that directory is a balanced tree (a *B-tree*): three or four disk reads are enough for ten million rows.

The detail that matters is the price. Every time a row comes in, the engine must also file it in every directory of the table, at its place in the order. Martin Kleppmann, author of *Designing Data-Intensive Applications*, sums it up as a law: "well-chosen indexes speed up read queries, but every index slows down writes". So you do not index "just in case": you index the columns you search often, and you pay the write for the read, exactly the sorting trade of section 1.3.

Say what you want: the query and its plan

The standard language for talking to the engine is **SQL** (Structured Query Language), and an SQL query, meaning a question put to the database, says what you want, not how to get it. You write "the name of the customer whose e-mail is smith@example.com", and the engine alone picks the method: read the whole table or go through an index. The **execution plan** is the method the engine has chosen to answer a query.

Placed in front of a query, `EXPLAIN` shows this plan without running the query, in PostgreSQL as in MySQL (its variant `EXPLAIN ANALYZE` does run it for real: on an `UPDATE` or a `DELETE`, it really changes the data).

```
SELECT name FROM customers WHERE email = 'smith@example.com'
-- EXPLAIN (PostgreSQL): Seq Scan on customers      ✗ reads all ten million
rows
CREATE INDEX customers_email ON customers (email)
-- EXPLAIN: Index Scan using customers_email        ✓ a few steps
SELECT name FROM customers WHERE lower(email) = 'smith@example.com'
-- EXPLAIN: Seq Scan on customers      ✗ the function hides the column, the
index goes unused
```

The classic trap is on the last line: the index exists, and the engine does not use it. As soon as a function wraps the column, `lower(email)` or a date you extract the year from, most engines no longer find the directory, which is sorted on the e-mails as written, not on their lowercase version. The fix: index the expression itself (a functional index: PostgreSQL has long offered it, MySQL since 8.0.13), or store the e-mails already in lowercase and compare the column as it is.

Markus Winand's online guide, *Use The Index, Luke*, catalogs such cases where an index exists but goes unused: a function around the column, an implicit type conversion, a wildcard at the start of the pattern. The reflex, when you code: faced with a slow query, read its plan before touching the code.

The second trap shows up in no plan, because each query, taken alone, is perfect. Picture a page that shows a hundred orders with their customer's name. The code asks for the orders first, then repeats, inside a loop, one query per order to find its customer: a hundred and one queries, where one join would do.

The **N+1** is a loop that fires one query per turn instead of a single join, and ORMs (object-relational mappers), the tools that write SQL on your behalf, produce it without warning.

The fix: fetch the customers of all the orders in a single join, or in one grouped second query, which ORMs call *eager loading*.

Each query pays a round trip: the question travels to the engine, the answer travels back, and the engine often lives on another machine.

↳ Designing Data-Intensive Applications

↳ WHICH LEADS TO CHAPTER 5

Composing pieces instead of inheriting from a mold, recognizing in a pattern an assembly others have already tested, making dependencies point toward the business rules, weighing every split by asking what falls with it, entrusting the shape of the whole to a single mind. And storing the data in a database where every index speeds up reads and slows down writes. Those are the chapter's six decisions. One question ties them together: what can change without dragging the rest along? The shape is set, the boundaries drawn.

But this system will not live on one machine, as sections 4.4 and 4.6 already hinted: the browser is here, the server elsewhere, the database often on another machine, sometimes on another continent, and every one of the N+1's hundred and one queries makes the trip. Between them stretches the **network***, the cables and relays that carry messages from one machine to another: a world we have not looked at yet, with physical laws of its own. Before shipping anything, you need to know how two machines talk.*

"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway." — Andrew Tanenbaum, *Computer Networks*, 1996

5 Two machines talk

the network

Until now, we looked inside a single machine. The pyramid of section 1.2 already announced it: below the disk there is one more floor, the slowest of all, about a million times the duration of a RAM access. Here we are. One click is enough: the request, the message that asks for the page, leaves the browser, crosses cables, routers (relays that steer messages) and sometimes an ocean, and the answer comes back. That journey has laws of its own, and they look like nothing we have seen: distance costs, messages get lost, and nobody directs the traffic.

The journey unfolds in five stages: what distance really costs, the tiny contract that holds the Internet together, the reliability built on top of it, sharing with nobody in charge, and what the web pays on every page.

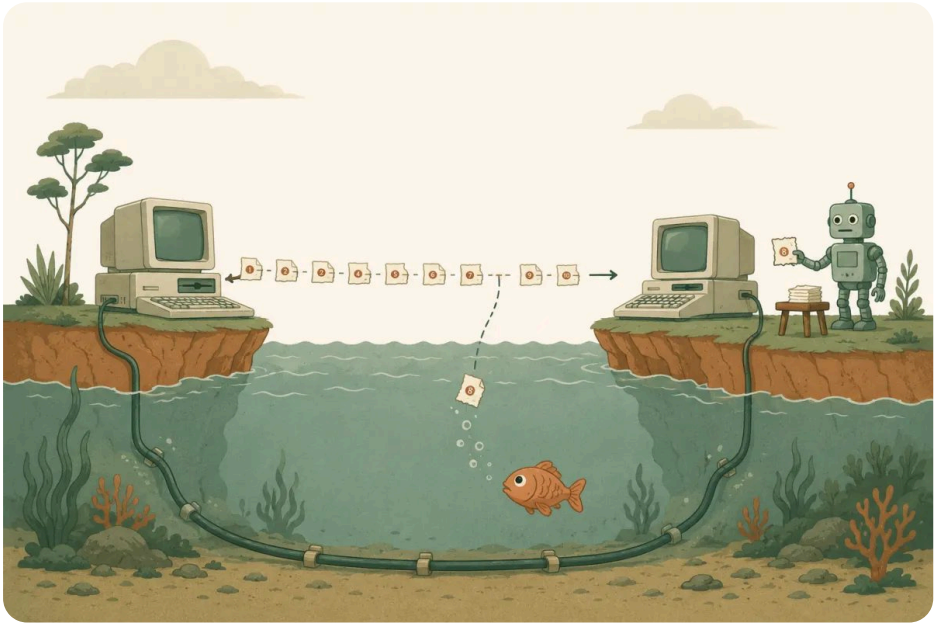


Figure 44 : The network cuts, numbers, and loses pieces along the way. The whole art of this chapter: acting as if nothing happened, by sending a copy.

5.1 Distance does not compress

Two numbers describe a network, and they get confused all the time. Picture a highway. **Bandwidth** is the number of lanes: how much data passes abreast every second. **Latency** is the length of the drive: how long a message takes to make the round trip. Reread Tanenbaum's station wagon, it is already driving on that highway. Picture it loaded with tapes by the thousand and set off on a drive of many hours: the tapes give it huge bandwidth, the hours on the road a terrible latency. Your provider's ads only ever brag about bandwidth, all "2 Gbit/s fiber". Yet it is latency that keeps you waiting.

The difference is physical. Bandwidth can be bought: one more fiber, more frequencies, more lanes. Latency cannot, you do not move cities closer: it is bounded by the speed of light. In fiber, the signal travels at about two thirds of the speed of light, and a Paris–New York round trip will never drop much below 60 ms (70 to 80 ms in practice, since cables do not run in a straight line). No subscription shortens the Atlantic.

Yet latency is what governs the web. Mike Belshe, one of the designers of SPDY (the ancestor of HTTP/2, the language browsers and servers speak), measured this at Google, and *High Performance Browser Networking* takes up his result: past a few megabits per second, **doubling the bandwidth changes almost nothing** in page load time. Cutting latency speeds it up almost linearly. The reason: an average page is dozens of small resources (images, stylesheets, fonts, scripts), hence dozens of round trips, many of them in sequence (the browser only discovers an image after reading the page), that add up. The web is not limited by the width of the road, it is limited by its length. A video or a big download, though, does need width.

When you code, count round trips before bytes: the hundred and one queries to the database of the N+1 in 4.6, at 60 ms each, cost six seconds, while a single call that groups them costs only about sixty milliseconds.

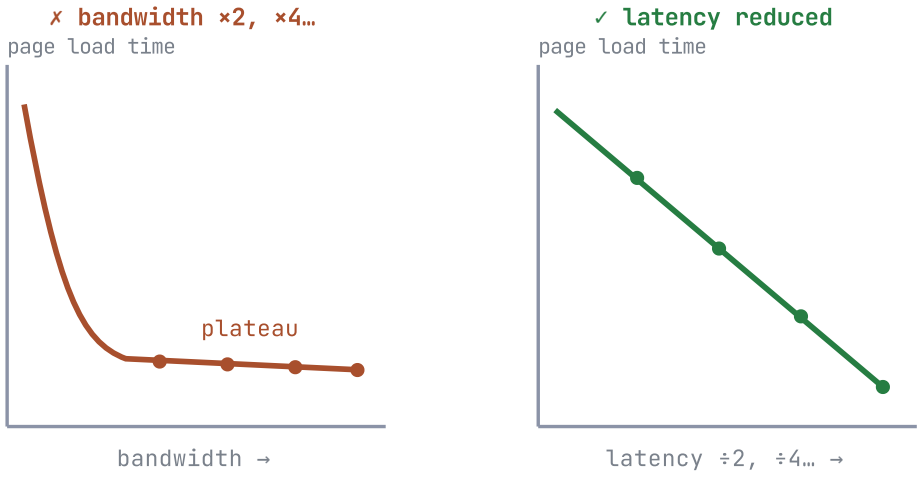


Figure 45 : Belshe's measurement: past a few megabits, widening the road no longer helps. Shortening it always does.

↳ High Performance Browser Networking

5.2 The hourglass: a tiny contract in the middle

What happens to a message during those 60 ms of travel? First, the network never carries a message in one piece. It cuts it into **packets**: small independent chunks, like the pages of a book mailed in separate envelopes. Each packet carries the destination address and is forwarded hop by hop by **routers**, the relays that steer it, the first being the modem-router your provider installed at home. Two packets of the same message may take different routes and arrive out of order.

Second, the genius of the Internet lies in what it does NOT promise. In the middle of the network, every machine shares one single **protocol**, a conversation rule they all agree on: **IP** (*Internet Protocol*), which gives each machine an

address and moves packets toward it. Its contract is tiny: "I do my best" (*best-effort*). A packet may be lost, duplicated, delayed, delivered out of order, and IP has promised nothing but to try.

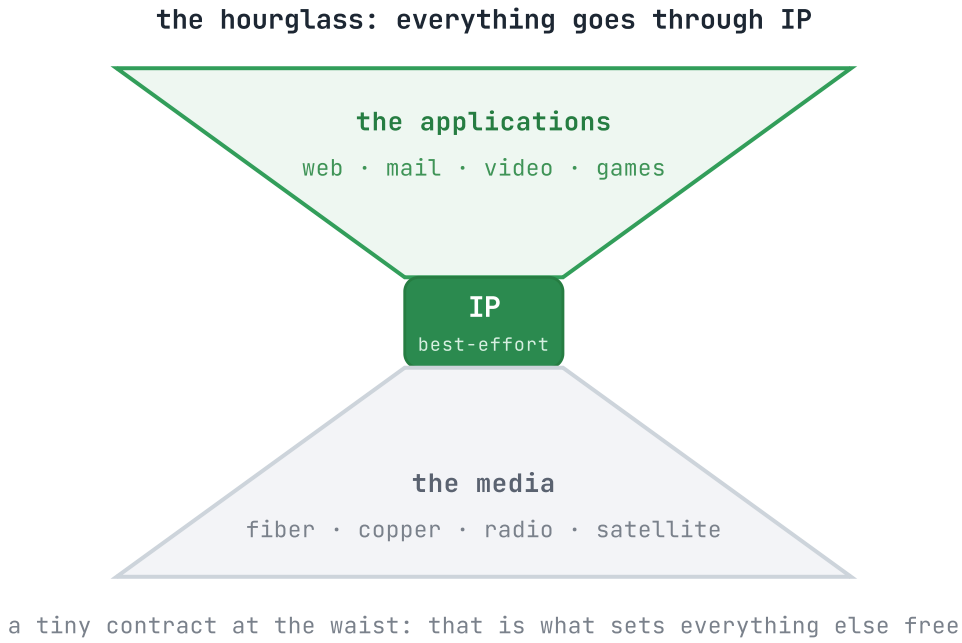


Figure 46 : Below, the media have changed for 50 years. Above, so have the applications. The waist has not moved: IP and its "I do my best".

Why promise so little? Because a simple middle is a universal middle. Reliability is guaranteed **at the endpoints**, sender and receiver, not in the middle: only they know whether the whole message arrived (the so-called *end-to-end* argument, set out by Saltzer, Reed and Clark). The hourglass waist stays lean enough to run over any medium, including those that did not exist at its birth: fiber, 5G and satellite arrived without its contract changing. You will recognize here the principle of 4.2, "program to an interface, not an implementation": IP is the interface, and on both sides, applications and physical media alike, everyone stays free to vary.

That tiny contract climbs all the way into your code. A network call is not a function call: it can fail, drag, never answer, and that is no accident, it is the contract set at the waist of the hourglass. Hence three developer reflexes, from the very first line that crosses the network. Set a delay beyond which you stop waiting: that is the **timeout**. Plan to retry, but without applying the operation twice: a timeout does not tell you whether the transfer went through, and

a transfer sent again must not be debited twice. An operation that can be repeated without a double effect is called **idempotent**. And you retry politely, waiting longer and longer between two attempts (1 s, 2 s, 4 s), or else a thousand clients retrying at once rebuild the outage (5.4). Finally, plan the error message, rather than a page that spins forever.

↳ Computer Networks: A Systems Approach

5.3 TCP: building reliable out of unreliable

IP does its best, but its best is not enough for everything: a bank transfer cannot afford to get lost along the way. Something has to bridge the gap between what IP promises and what an application actually needs. That is the job of **TCP** (*Transmission Control Protocol*), running on the two endpoint machines, right above IP. Its mechanism copies registered mail. Every byte sent carries a **tracking number**. The receiver replies with a **proof of receipt** saying what actually arrived. And if a receipt does not come back in time, the sender **sends the packet again**, without asking why it went missing.

Number, receipt, resend: these three moves repair loss, duplicates (the number lets the receiver discard them) and disorder (it lets the receiver put packets back in order). The application above sees none of it. It believes it is driving on a perfect road, while the network underneath is nothing of the sort. This is reliability built at the endpoints, exactly as the hourglass intended.

This service has two prices. The first is paid at the door. Before a single useful byte, the two machines exchange a **three-way handshake**: "I want to talk", "got it, me too", "got it" (SYN for "synchronize", ACK for "acknowledge": SYN, SYN-ACK, ACK in the diagram). That handshake costs one full round trip, so 60 ms lost on Paris–New York before anything begins. A **connection** is the conversation two machines open with a handshake, each keeping track in memory of where it stands. And every new connection pays that price again.

The second price is subtler: how fast should you send? Waiting for each packet's acknowledgment would leave the road nearly empty. So TCP sends a whole convoy of packets, trucks on our highway, before hearing back about

the first one, rather than one truck, silence, then the next. The number of packets allowed on the road without news is called the **window**.

The right size of the window is the **bandwidth × latency product**: multiply the bandwidth by the round-trip time, and you get what the road holds at any instant, every truck that has left and whose confirmation has not yet come back. At 100 Mbit/s, or 12.5 MB per second (a byte is 8 bits), with a 60 ms round trip between Paris and New York, about 750 KB are permanently in flight, already sent, not yet confirmed. But TCP does not know this figure in advance: it has to guess it along the way (5.4).

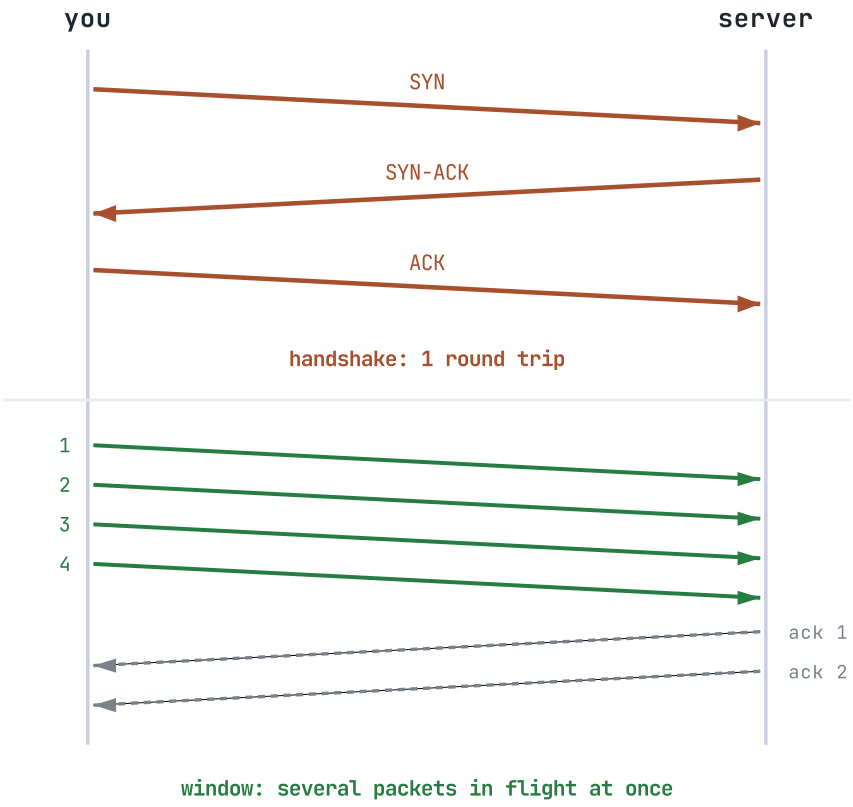


Figure 47 : In orange, the handshake costs one round trip before the first useful byte. In green, the numbered packets leave without waiting for one acknowledgment per packet (dotted grey): several travel abreast, that is the window.

Thanks to TCP, you never handle a lost or out-of-order packet: your code receives the bytes repaired and in order, or an error if the connection breaks. The catch: a lost packet holds back every packet behind it, even those already arrived, until it is resent (this is *head-of-line blocking*). And two costs remain that

code cannot remove, only spread out: the handshake (reusing a connection already open avoids paying it again) and the window (a connection that has already found its right window size does not start from zero).

↳ Computer Networks: A Systems Approach · High Performance Browser Networking

5.4 Congestion: sharing without a boss

TCP's window leaves one question open (5.3): how big should it be? Millions of connections share the same cables, and nobody hands out speaking turns. Every router holds waiting packets in a queue, like cars before a toll booth. If everyone sends at full speed, the queue overflows and the router drops the extra packets. Everyone then resends what they lost, which makes the jam even longer.

Congestion collapse is the vicious circle in which resent packets lengthen the jam that made them go missing. It happened for real: in 1986, a link between the University of California at Berkeley and the nearby Lawrence Berkeley lab dropped from 32 useful kbit/s to 40 bit/s, 800 times less. No cable was cut, no machine was down, and yet nothing was getting through. The engineer Van Jacobson drew from it the rules that follow.

The remedy is a rule of the road that every machine applies to itself, with no referee. Speed up **gently**, one window notch more per round trip. Then, at the first lost packet, **cut your speed in half** at once. It is how a careful driver behaves: you pick up speed little by little, and you lift off the moment you see

brake lights ahead, without knowing what is blocking further on. Loss is no longer just an accident to repair: it becomes the **signal** that the road is saturating, and everyone brakes on their own upon seeing it.

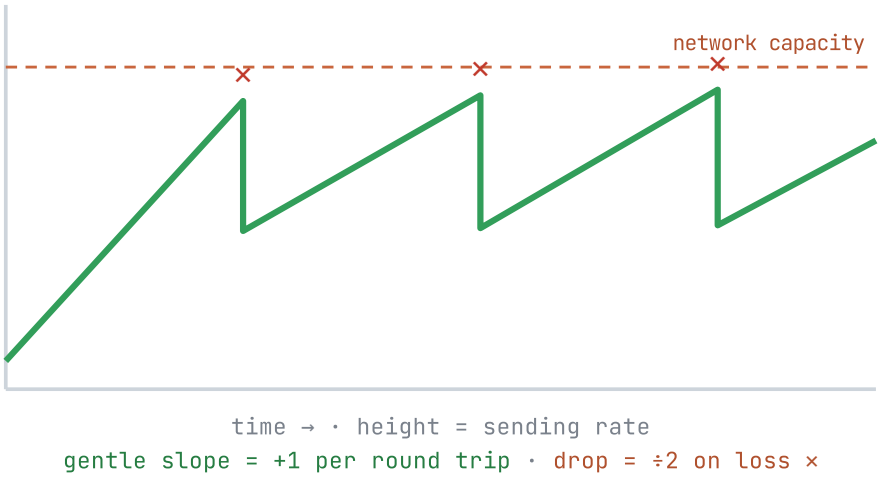


Figure 48 : Climb gently, halve on loss: the sawtooth hugs the capacity without collapsing. The Internet holds because every machine brakes on its own.

This coded politeness is called **AIMD** (*Additive Increase, Multiplicative Decrease*: add a little when all goes well, multiply by a fraction on loss). It has a catch: very long router queues delay the loss, hence the signal, and latency swells long before anyone brakes (this is *bufferbloat*). Modern variants refine the curve (CUBIC climbs back faster, BBR measures the road instead of waiting for loss), but the principle has not moved since Van Jacobson's 1988 paper: a commons with no policeman, held together by everyone's restraint.

You have already seen that caution at work. Start a large download: the speed on screen climbs over the first few seconds instead of starting at full throttle. The connection does not know the capacity of the road yet, so it begins small (about ten packets, roughly 14 KB) and doubles its window on every round trip: this is **slow start** (slow only compared with sending everything at once).

Keep that figure in mind: a page that fits in those 14 KB, compressed (shrunk with gzip or Brotli), arrives in a single burst, without waiting for an acknowledgment. At the first incident, the connection falls back to the cautious step-by-step climb seen above. A warm connection, one that has just served traffic

and widened its window, starts much faster (left idle, it forgets): hence the value of opening as few connections as possible, which the next section will put numbers on.

↳ Computer Networks: A Systems Approach · High Performance Browser Networking

5.5 The web pays the toll

HTTP (*HyperText Transfer Protocol*) is the language browsers and servers speak: ask for a page, answer with its contents. But an HTTP request never travels alone: it is text handed to a TCP connection, itself carried packet by packet by IP. Take a visitor who types your site's address. Before the first byte of the page, everything this chapter has laid out gets paid, round trip by round trip: that is the web's toll. Four steps follow one another, each waiting for the previous one.

- **DNS** (*Domain Name System*): the web's directory turns the site's name into an IP address, because a router knows how to forward towards an address, never towards a name. One round trip to a name server, often close by, and none if the answer is already remembered.
- **The TCP handshake**: the two machines agree to open a reliable connection (5.3). One round trip.
- **TLS** (*Transport Layer Security*): encryption, the padlock of *https* (the "s" stands for "secure"). The server proves its identity with a certificate, then the two machines agree on a secret key, so that nobody along the way can read or alter what follows. One round trip with TLS 1.3, two with TLS 1.2.
- **The request**: at last, the browser asks for the page. One final round trip before the first useful byte.

At 60 ms of latency, count at worst 240 to 300 ms before the first response (depending on the TLS version). On mobile, add the radio waking up: to save battery, the phone's radio drops its link with the antenna as soon as it is idle, and has to re-establish it before sending anything, which costs tens to hundreds of milliseconds more.

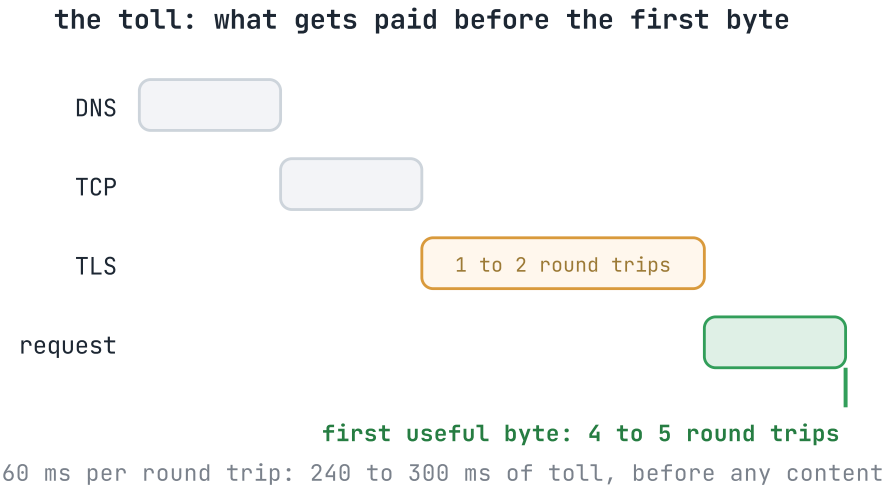


Figure 49 : Each step waits for the previous one: the toll adds up in round trips. That is what HTTP/2 and then HTTP/3 set out to shave.

That toll explains a decade of workarounds. **HTTP/1** can only ask for one resource at a time per connection, and a browser opens only six of them per domain (per site name, such as mysite.com or images.mysite.com). So developers worked around it in two directions. First, cutting down the number of requests, by gluing files end to end (*bundling*) and packing every icon into a single image (a *sprite*). Then, dodging the limit, by spreading resources across several domains to earn six more connections each time.

HTTP/2 fixed the connection problem: every request travels interleaved over **one single** connection, so one handshake only, and a window that stays open and wide. Those old tricks now do little, or harm: spreading across domains reopens exactly the connections the protocol had just saved.

HTTP/3 goes further still: it drops TCP for a new transport, **QUIC** (pronounced "quick"), because one single TCP connection has a flaw. A lost packet makes every request queued behind it wait, even those whose packets have already arrived. QUIC, built on UDP (sending packets with no acknowledgment,

so QUIC rebuilds reliability itself), recovers each request separately and also merges the handshake and the encryption into a single round trip instead of two or three.

The chapter's moral is the golden rule of web performance: **the fastest request is the one you never make**. Three moves follow from it, each answering one lesson of these five sections. Cache, so you never ask twice for what you already have: a setting line attached to the server's reply (a *header*), such as `Cache-Control: max-age=31536000, immutable`, tells the browser it may keep the file for a year without asking again. With a file name that changes on every version, a new version still reaches everyone. Reuse one connection, so you pay the handshake and the encryption only once. And move the content closer to the visitor with a **CDN** (*Content Delivery Network*), meaning relay servers spread around the world, since no subscription shortens the Atlantic. Cache and CDN come back in 6.6, at the scale of the whole service.

↳ High Performance Browser Networking

↳ WHICH LEADS TO CHAPTER 6

Five laws emerge. Distance is paid for in round trips, and neither money nor code shortens them. In the middle, IP only promises to do its best, and TCP rebuilds at the edges the reliability that is missing. With no boss, every machine brakes on its own as soon as a packet is lost. Finally, every web page pays its toll of DNS, handshake and encryption before the first byte. One lesson ties them together: the network is slow, fallible and shared, and code must assume it.

But two machines that can talk do not yet make a service that holds. An architecture, however elegant, is only a **hypothesis** *until it has been put to the test: the rule of 4.3, which makes dependencies point towards the business rules, cannot be checked by eye: it takes a test that fails as soon as a boundary is crossed. Prove it, ship it, hold it under real load: the shape laid down in chapter 4 must meet the world.*

"Talk is cheap. Show me the code." — Linus Torvalds, 2000

6 Prove it, ship it, hold the load

flow

Between code that "works on my machine" and a service that holds in **production**, the machine open to the public where thousands of people actually use it, lies a chasm. Crossing it is not about coding more: it is about installing a **flow** that proves, ships and holds, without heroics and without all-nighters.

Seven links make up that flow. Proving first: writing the test before the code, then hunting bugs without taking anything on trust. Shipping next: keeping the history of everything you write, making work circulate instead of piling it up, and shipping so often that it stops being an event. Holding the load at last: taking a million readers, and finding out what guarantees cost once the data no longer fits on one machine.

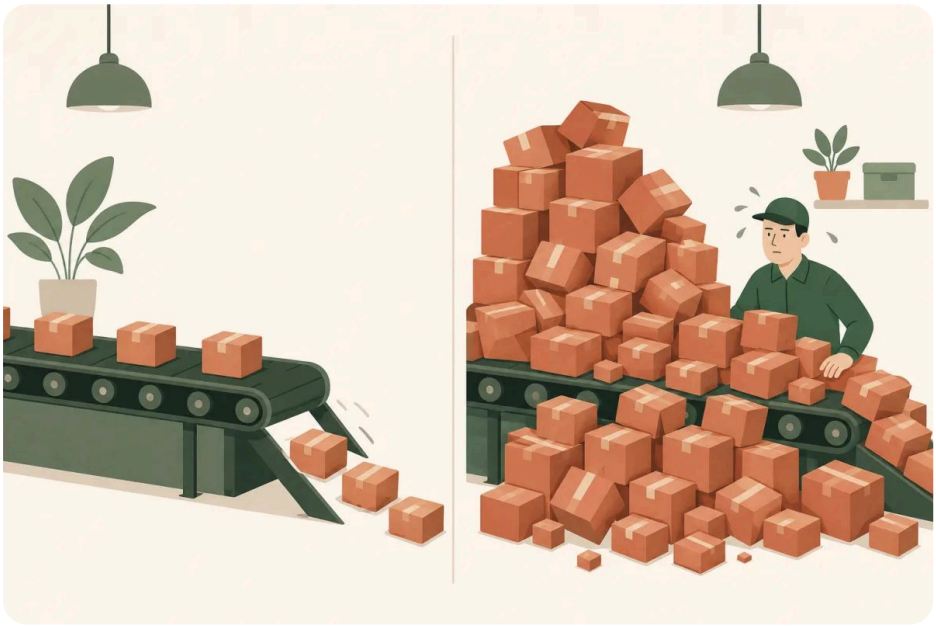


Figure 50 : Steady flow ships effortlessly; a saturated queue freezes everything, even when everyone is flat out.

6.1 Test first

The test of 3.6 called the code after the fact, to check that a refactoring had broken nothing. Let us write it before the code, "`sum(5, 3)` must return 8", and that changes everything: you define the expected result before knowing how to get it. The cycle repeats endlessly: **Red** (a failing test), **Green** (the dumbest code that passes, even hard-coded), **Refactor** (clean without breaking).

```
// 1. RED: the test BEFORE the code
test('5 + 3 = 8', () => expect(sum(5, 3)).toBe(8))   // ✗ fails

// 2. GREEN: the dumbest code that passes (yes, "8" hard-coded)
function sum(a, b) { return 8 }
// ✓ green, no shame

// 3. RED then GREEN: a 2nd test breaks the "8", you generalize
function sum(a, b) { return a + b }
// ✓ green for both tests
```

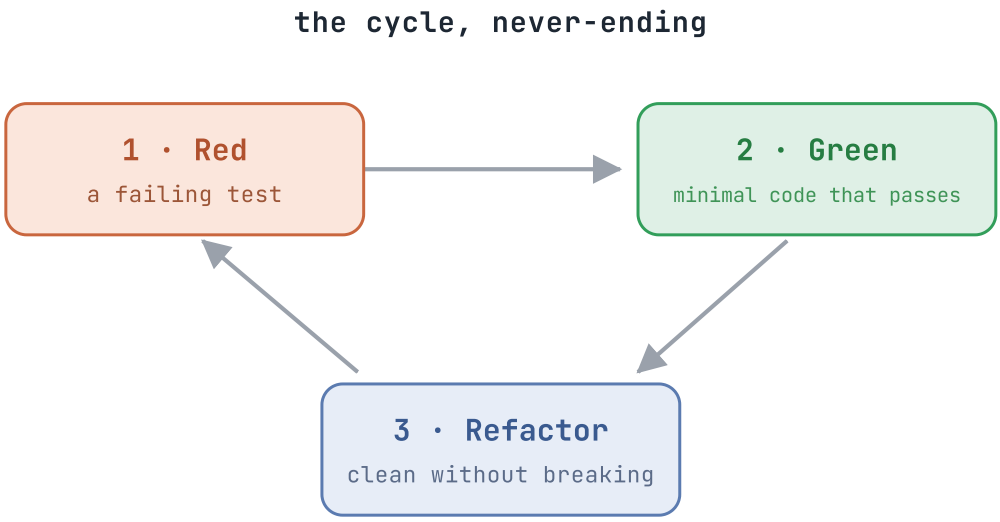


Figure 51 : You loop forever: a failing test, the minimal code that makes it pass, then you clean up, and on to the next.

You never clean up while a test is failing: green first, tidying afterwards. The goal, in four words: clean code that works.

Which test to write next? The same cases come back for every function. Take the `cart_total` of 2.1:

› nothing: an empty cart must return 0;

- one: a single item returns its price;
- several: three items return their sum;
- the edges: €99.99, €100 and €100.01, around the discount threshold. The test at €100 asks a question nobody had put to the salesperson, "exceeds" or "reaches", and it is the edge case that asks it, not the code;
- the invalid: a negative quantity or a price in text must fail early (3.4) rather than return an absurd total.

One test per case, each named for what it checks: the list of tests becomes the most honest description of what the function does.

But the deepest effect of **TDD** (*Test-Driven Development*) is not catching bugs, it is **emergent design**: writing the test first, you design the API, that front counter from section 4.5, from the point of view of whoever comes to drop off a request, not of whoever staffs the desk. The code becomes modular and decoupled because it has to be testable.

You don’t draw the architecture up front, you let it emerge, test after test. The short functions of chapter 3 and the clean dependencies of chapter 4 then become consequences rather than acts of willpower.

One difficulty puts beginners off: how do you test a function that sends an email, writes to the database or calls a remote service? A test that touches the real world is slow, it fails for reasons that have nothing to do with your code (the network, the clock, somebody else’s database), and it sends real emails.

The answer is the wall socket of section 2.6: instead of plugging the real service into the function, you hand it in as a parameter. Michael Feathers calls this a **seam**, a place where you can change behavior without modifying the function itself:

```
function register(email, send) {
// send comes from outside: that is the seam
  send(email, "Welcome!");
}
// in production: register(mail, realMailSender)
const calls = [];
// in test: a fake appliance records what it is asked
register("a@b.fr", (to, msg) => calls.push([to, msg]));
// → calls holds [["a@b.fr", "Welcome!"]]: nothing was sent, the test is
// instant
```

A fake appliance plugged into the socket is called a **mock** (a fake that answers what it is told to answer and also records what it is asked). It is the dependency inversion of section 4.3 seen from the test side: code you can test without the world is code whose dependencies are clean. Two limits. With too many fakes, you test the scenery rather than the code. And faced with old code that has no tests, you first write a **characterization test**: it records what the code really does, not what it should do, before you change it. For Feathers, code without tests is already *legacy*: code nobody dares touch, whatever its age.

A test that isolates a function behind fakes is a **unit test**. The one that plugs the real pieces together, database included, is an integration test, and the one that plays the whole journey, from the click to the receipt, an end-to-end test: slower and more fragile in that order, so many of the first, some of the second, a handful of the third.

Time and chance are two hidden inputs of the same kind: a discount "until midnight" tested at 11:59 pm passes, at 12:01 am fails, without a line having changed. A test must give the same answer on every run, so you pass the clock and the die as parameters, like `send` above: it is the seam again.

And the economics back the discipline: the same defect costs ten to a hundred times more when you find it in production rather than here, at the keyboard (*Code Complete*). Testing early isn't zeal, it's the cheaper path.

↳ Test-Driven Development: By Example · Working Effectively with Legacy Code · Code Complete

6.2 Debugging is refusing to believe

Tests catch most bugs; the survivors are the ones that look impossible. A programmer who can log in sitting down but never standing up. A Chicago banking terminal that crashes the instant a customer types "Quito". If a bug looks impossible, it means one of your certainties is false and you do not yet know which one. That is all Bentley's rule says: "debugging is usually about refusing to believe." Refusing to believe what you take for granted, the line you have reread ten times, the configuration you are sure you deployed, the data you assume is clean. Until those certainties are checked one by one, the explanation stays hidden behind one of them.

Rick Lemons, whom Bentley quotes, said the best debugging lesson of his life had been a magic show. Six impossible tricks in a row, not one of them actually impossible: the magician was simply pulling your eyes to the side where nothing was happening. An impossible bug works the same way, you just have to look at the other hand.



Figure 52 : An "impossible" bug works like a magic trick: you stare at the flashy hand, and the action is in the other one.

Our two examples then untangle themselves. Sitting down, the programmer typed his password without looking at the keyboard; standing up, he looked at the keys, and two of them had been swapped. As for the terminal, it checked whether the input started with `quit`, its command to exit: "Quito" starts with "quit".

Your own impossible bug will look more like this: everything works on your machine, and nothing works in production. Stop staring at the code that "cannot" be wrong, and ask the only useful question: what is different between the two? The answer is almost always outside the code. **Configuration** is a setting the program reads at startup instead of carrying it in its code, and its values often differ between the two machines. Other suspects: a cache serving an old answer, two requests arriving in the opposite order, a value forgotten at de-

ploy time. That question is not settled from memory. You read it in what the application wrote about itself while it ran: its log. How to keep one is the business of section 6.5.

Refusing to believe is practiced with method. Reproduce first: as long as you cannot make the bug appear at will, you cannot prove it has gone either. Then reduce: remove items from the cart, lines from the file, steps from the journey, down to the smallest case that still fails, because a bare bug can be read and a bug drowned in a thousand lines cannot. The Chicago terminal: "Quito" crashes, "Quit" crashes too, "Qui" does not, and the border points at the guilty line. Then one hypothesis at a time, checked before the next, instead of changing three things at once and no longer knowing which one acted.

The **debugger** stops the program on a chosen line, the **breakpoint**, and shows the contents of every locker: it is the by-hand trace of 2.1, run by the machine. When the bug showed up "one day" and nobody knows which, Git searches for you: `git bisect` replays the history by halves, like the directory of 1.3, and finds the guilty commit among a thousand in about ten tries (6.3). The last move is the first of 6.1: write the test that reproduces the bug, Red, before fixing it. The fixed bug joins the safety net and will not come back without showing itself.

And one move works absurdly well once you are stuck: explain the bug out loud, to a colleague or even a rubber duck. McConnell calls it *confessional debugging*: developers routinely find the answer mid-sentence, before the listener says a word. Reconstructing the causal chain for someone else forces your brain from "hunting" to "understanding".

↳ Programming Pearls · Code Complete

6.3 Versioning is a content-addressable system

Versioning is the flow's **safety net**: you change anything, you experiment, you work with others without colliding, and you can always go back. Proof by fear: you just wiped out three days of commits with a mis-aimed

`git reset --hard`. A commit is a dated snapshot of the whole project, with its author's name: Git's basic unit. Panic. Except in Git, almost nothing truly disappears, and understanding why changes everything.

Git is not a history folder: it is a database of objects, each addressed by its **fingerprint**: a short code computed from the content by the SHA-1 function, called a "hash". The same content always yields the same hash, so nothing is lost or silently forged. Everything chains by pointers:

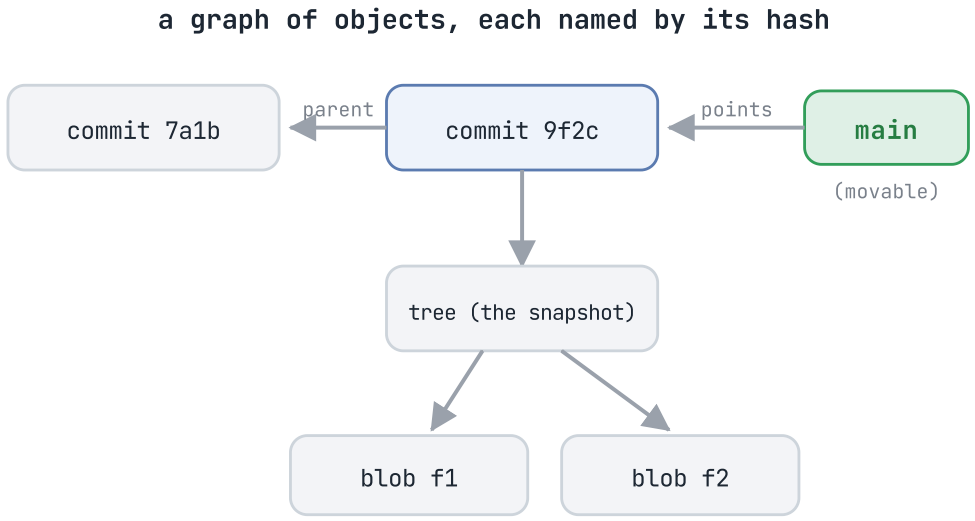


Figure 53 : Every object is immutable, named by its hash. A branch is just a pointer you move; nothing truly vanishes.

Five words are enough to read this diagram. A **blob** is a file's content. A **tree** is the list of files in a folder at a given moment. A **commit** points to a tree and to its parent. A **branch**, here `main`, is just a pointer to a commit, moved forward with each new commit. And `HEAD` is the "you are here" pointer: the commit you are working on.

```
git cat-file -p HEAD
# shows this commit: its tree, its parent, the author
```

And before entering that graph, a file passes through **three zones**, which is why `git add` exists:



Figure 54 : `git add` photographs a version for the next commit; `git commit` seals it into the graph. Hence you can commit only part.

`git add` saves nothing: it photographs the exact version of a file for the next commit. The commit seals that photo into the graph. That is why you can commit only part of your changes.

That leaves the opening promise: working with others without colliding. Each person advances their own **branch**, one pointer among others, without touching anyone else's. When the work is ready, a **merge** glues the two histories back together: Git creates a commit with two parents, one per branch.

Between the branch and the merge, there is a step Git does not do: someone reads. You propose your branch, which is called a *pull request* (a request to merge), with the exact list of changed lines, and a colleague reads it before it is glued back. That is **code review**: the next human of chapter 3 shows up sooner than planned, not in six months but tomorrow morning. The numbers agree. In *Code Complete*, McConnell reports that formal code inspections find about six defects in ten, more than most forms of testing. A pull request review is lighter, but points the same way. And *Accelerate* found that teams that review each other's work deliver faster than those that wait for an outside committee's approval, which slows things down without making them safer. When you code: a short branch gets read in ten minutes, a month-long branch does not get read, it gets signed with eyes closed.

And your three wiped days? The commit before the `reset` is still there, an immutable object in the database; `git reflog` lists the recent hashes (Git keeps them for about three months), you point a branch back at it, and it all re-

turns. The net only catches what was committed: a change you never committed really is gone. Hence the reflex: commit often. Understanding the graph is how you ship without panic.

The net has a flip side: nothing disappears, not even what you would like to see disappear. A database password committed by mistake stays readable in an object of the graph, for anyone who clones the repository, even erased from the file the next day. A **secret**, a password or an API key, is a value that never lives in the code: it arrives through the configuration, an environment variable or a file Git ignores. And a secret committed even once is not erased, it is changed.

↳ Pro Git · Code Complete · Accelerate

6.4 Flow beats effort

The Phoenix Project teaches, through a novel, a merciless factory law: a task's wait time **explodes** as a **resource** (a server, a team, a person everything depends on) approaches 100% utilization.

```
wait time ≈ % busy ÷ % free   // the queueing law
```

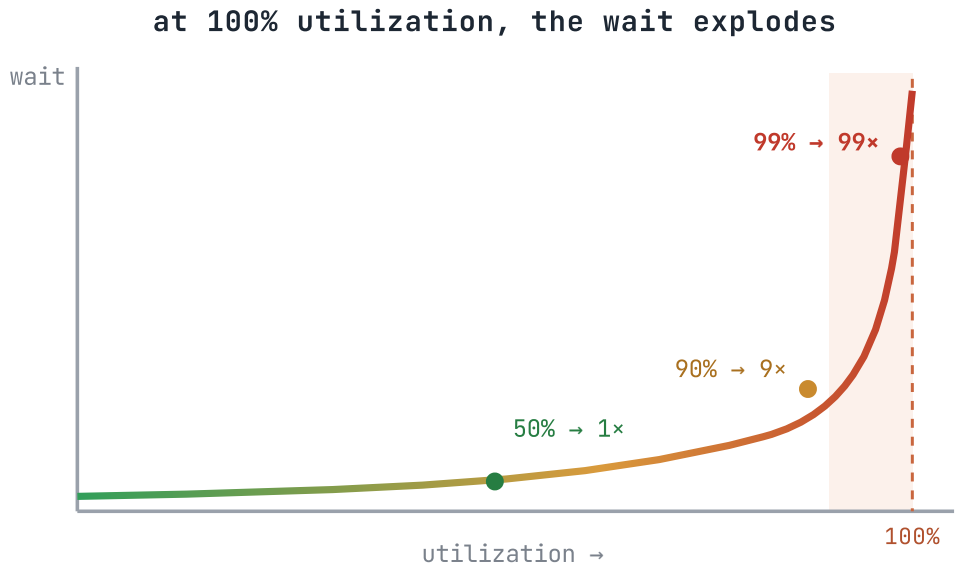


Figure 55 : Nearly flat, then vertical: 50/50 = 1×, 90/10 = 9×, 99/1 = 99×. Slack is not waste; it is what lets work flow.

Why the wall? At 99% utilization, no slack is left to absorb the unexpected: one task that drags, one burst of arrivals, and the queue swells with no way to drain. At 50%, the spare time soaks up those bumps as they come.

"Everyone is flat out" and "nothing moves" are therefore the same sentence. The remedy is counterintuitive: limit work in progress, stop starting things to actually finish some. Slack is not waste: it is what lets work flow.

A second flow hides under the same word: the one in your own head. Designing or coding takes fifteen minutes of climbing to lock in, and an interruption does not cost five minutes, it costs the whole climb back, near twenty (DeMarco and Lister measured it in *Peopleware*). Slack protects this flow too: a calendar pinned at 100%, like a server at 100%, lets nothing truly move.

And the data is blunt: in the Coding War Games (a tournament of hundreds of developers the same authors ran), the biggest productivity gap came not from language or experience but from the work environment. Working somewhere quiet, rarely interrupted, the best had a third more chance of delivering work without a single bug. It is the environment that makes the gap, not the talent.

↳ The Phoenix Project · Peopleware

6.5 Shipping often is less scary than shipping rarely

The big release prepared over months is a cannon shot: once the ball has left, nothing can be corrected, and the target has had months to move. A burst of small deliveries does the opposite: each shot shows where it lands, and the next one corrects the aim.

CI/CD is the automatic chain that makes the burst possible. Continuous integration (CI) is a conveyor belt: on every commit, it replays the build and all the tests, and nobody merges on red. Continuous delivery (CD) extends that belt: every version that goes green is packaged, ready to ship to production in one click. Deploying stops being an event; it is the belt's normal output.

It remains to prove that the burst beats the cannon. *Accelerate*, the book by Nicole Forsgren, Jez Humble and Gene Kim, measures it: teams that deploy often and with **loose coupling** (shipping without asking another team's permission) are both faster and more stable. That loose coupling is the quantum of 4.4. The expected trade-off (move fast = break more) just isn't there in the data.

You get there with tools that make mistakes cheap: *feature flags* (deploy code switched off, flip it on, flip it back off at the first sign of trouble) or *blue-green* deployment (two versions ready side by side, a reversible switch from one to the other). Both rest on the same principle, set down by Jez Humble and David Farley in *Continuous Delivery* back in 2010: deploying (putting the code in place) is not releasing (making it visible to users). That is what removes the fear: a blunder is fixed with a flip, not an all-nighter.

```
if (flags.newCart) {          // a config value, not code
    showNewCart();
    // the code is already deployed, but switched off
} else {
    showOldCart();
}
// switching on = changing flags.newCart in the config, no redeploy
```

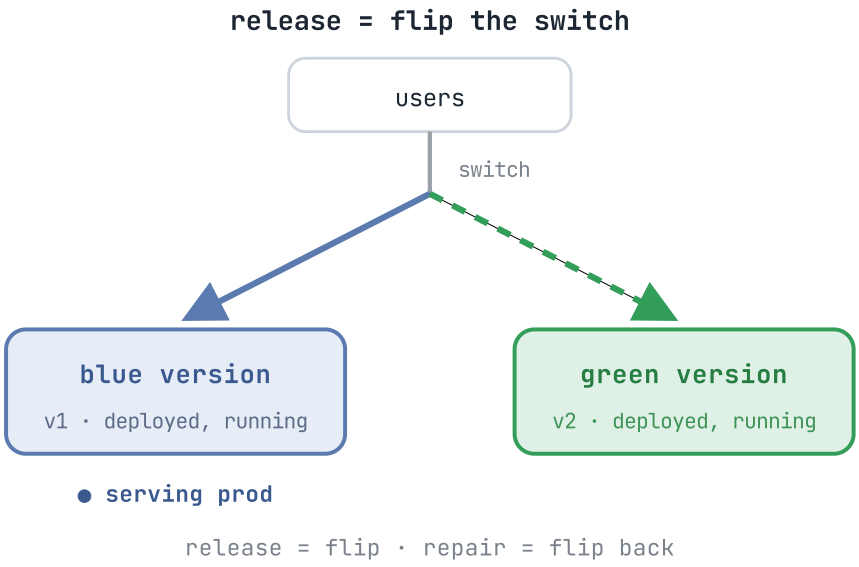


Figure 56 : Deployment already happened: both versions are running. Release is just a flip of the switch, and a blunder is fixed the other way, with one flip back. Solid line: before the switch. Dashed: after.

And you, where do you stand? *Accelerate* gives four numbers to find out objectively, the **DORA metrics** (*DevOps Research and Assessment*, the research team behind the book):

- **lead time**: time from a commit to production. Measured with two timestamps Git and the CI already have. The best: under an hour;
- **deployment frequency**: how often you ship. Counted from the CI logs. The best: on demand, several times a day;
- **MTTR** (*Mean Time To Recovery*): time to restore service after an outage, from the start of the incident back to normal. The best: under an hour;
- **change fail rate**: deployments followed by a rollback (going back to the previous version), a hotfix (a fix pushed in a hurry) or an incident, divided by the total. The best: 0 to 15%.

These four numbers are read together, with no formula or combined score: the first two measure speed, the last two stability. And when you plot the four numbers from thousands of teams, the teams sort themselves into three families: high, medium, low performance. The best win on all four at once while the weakest ship every six months (State of DevOps surveys, 2014-2017). The gain is human too: "deployment pain" leads to team exhaustion if left unchecked, and that pain is exactly what these practices (shipping often, small, reversible) drive down.

One last link conditions all the rest: **seeing**. MTTR assumes you know there is an outage. Without logs (the journal of what the application does), metrics (its vital signs: requests, errors, latency) and an alert that wakes someone up, you learn about it from an angry tweet. You only hold what you can see.

Site Reliability Engineering, the book written by Google engineers, boils these vital signs down to four: latency (the response time), traffic (how many requests arrive), errors (how many fail) and saturation (how full the resource is). That last one is the queue of section 6.4, seen from the machine room. These four figures tell you something is wrong. **Observability** goes further: it is being able to say what is happening inside a system from the outside, by reading

what it writes about itself, without opening it or redeploying it. When you code, it is decided at every log line: "error" is useless, "payment 8123 declined, card expired, 412 ms" can still be read at 3 a.m.

↳ Continuous Delivery · Accelerate · The Phoenix Project

6.6 Reads scale by copying, writes by splitting

Deployment is settled. The load remains: a million readers on a single database, and every query waits in the same line.

The tables of 4.6 are paid for at read time: an order does not copy its customer, it keeps its key, and the join that glues the two back together can cost more and more as the tables grow.



Figure 57 : The order stores only the customer's number. The join fetches the matching row and glues the two together at read time.

The grid is not the only way to file things. A document database (MongoDB is the well-known example), one of the families called **NoSQL** ("not only SQL"), puts each order in a folder: one block of JSON that holds everything, the customer, the lines, the address. Reading an order means opening a single folder, no join. In exchange, the same customer is copied into each of their orders (the denormalization of 4.6): if they move, you have to fix it everywhere.

It is a trade-off with no winner, like those of section 4.4. The folder is fast when you always read the same things together. The grid is safe when the same data gets crossed in a thousand ways. Then there is the phrase "schemaless": the schema always exists, the only question is who enforces it. The grid checks it on write, like the static typing of section 2.3. The folder leaves the application to check it on read, like dynamic typing.

On the read side, you defend in three steps. First the index of 4.6, the sorted directory that avoids scanning the whole table. Then the **cache**, which keeps already-computed answers close by, ready to be served again as-is. Redis, for example, is an ultra-fast shared memory placed in front of the database. The CDN of 5.5 does the same for files, images and scripts, with copies kept close to the visitors. It is the pyramid of 1.2, at the scale of a datacenter (the building that houses the servers). When that is no longer enough, you **copy**: **replication** duplicates the database across several machines, and each one serves its share of the reads.

The price of copying: replicas often run a little behind, because copying happens in the background rather than making every write wait. You post a comment (written on the primary machine), you reload the page (read from a copy that has not caught up yet): it is gone. The usual fix: after a write, read from the primary machine for a few seconds. The cache pays the same price: an out-of-date copy serves an old answer, and knowing when to throw it away is said to be one of the two hardest problems in the trade, along with naming (3.1).

A replica is not a backup. It keeps you running when a machine fails. It does not take you back after a mistake. On January 31, 2017, a GitLab engineer rebuilding a replica whose replication had fallen behind accidentally deleted the files of the main database, and the empty replica had nothing to give back. The daily export had been failing quietly (the export tool was older than the server, and its alert e-mails were being rejected) and the backup storage was empty. They had to restart from a snapshot, a frozen image of the database, six hours old, and it took eighteen hours to bring back. Their own conclusion: nobody was in charge of testing the restore. A backup never restored remains a hypothesis.

Copying does not help writes: every copy would have to absorb every write. Writes you **split. Sharding** spreads the data in slices (customers A-M here, N-Z there), and each machine only takes its share. Choosing the cut matters: by letter, the slices are uneven (far more names start with M than with Y) and one machine groans while the other sleeps. Cutting on a hashed key spreads the load better, at the price of range searches ("all customers from A to C"). Splitting is the last resort: every step adds a possible failure, so you measure before moving to the next.

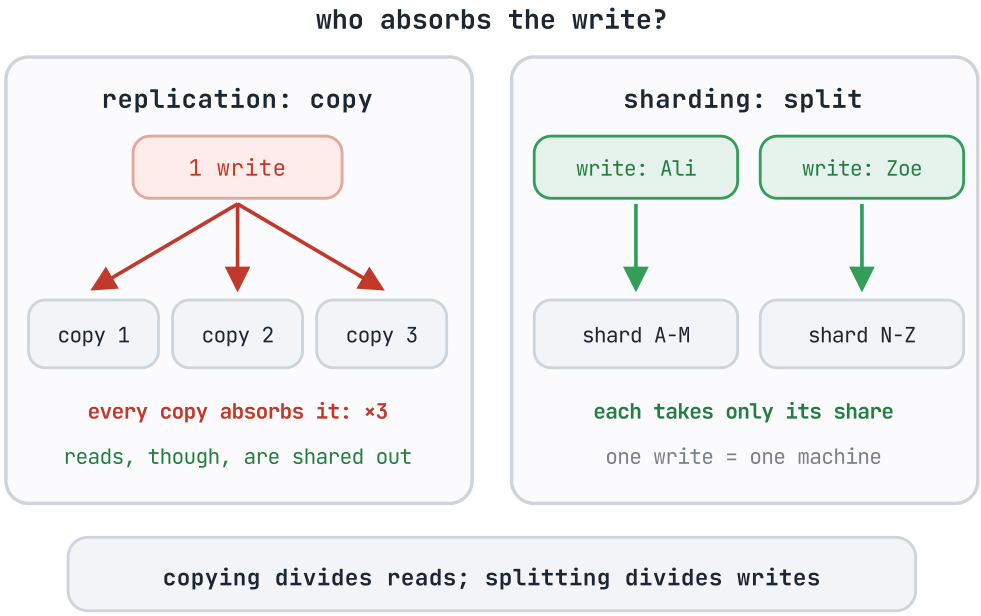


Figure 58 : The asymmetry in one picture: with copies, every write has to reach all of them; with slices, it only lands on its own.

And this is where the trouble starts: data scattered across several machines is exactly the situation where the engine's guarantees stop coming for free.

↳ Designing Data-Intensive Applications

6.7 At scale, guarantees have a price

On a single database, you live protected without knowing it. The transaction of 4.6 is all-or-nothing there: the transfer debits AND credits, or does nothing at all. That is the **ACID** contract: atomic (all or nothing), consistent (the database's rules stay true), isolated (each transaction believes it is alone),

durable (once committed, it is written). The engine gives it to you for free: if step 2 fails, it undoes step 1 on its own (the database calls it *rollback*, a different kind of going back than the one in 6.5).

Then the service grows, the data spreads across several machines, and that contract no longer comes for free: undoing "everything" takes coordination between machines, slow and fragile, which almost nobody pays for. Each machine only sees its own piece.

Anomalies appear, which most databases let through even on a single machine, for lack of full isolation. The nastiest one, **write skew**: two transactions, each perfectly valid, that break a rule together. The book's example: a hospital requires at least one doctor on call. Alice and Bob, the last two on call, sign off at the same moment. Alice's transaction checks "is Bob still there? yes" and commits. Bob's checks "is Alice still there? yes" and commits. Each saw a world where the rule held; together they leave zero doctors. No error was ever raised anywhere.

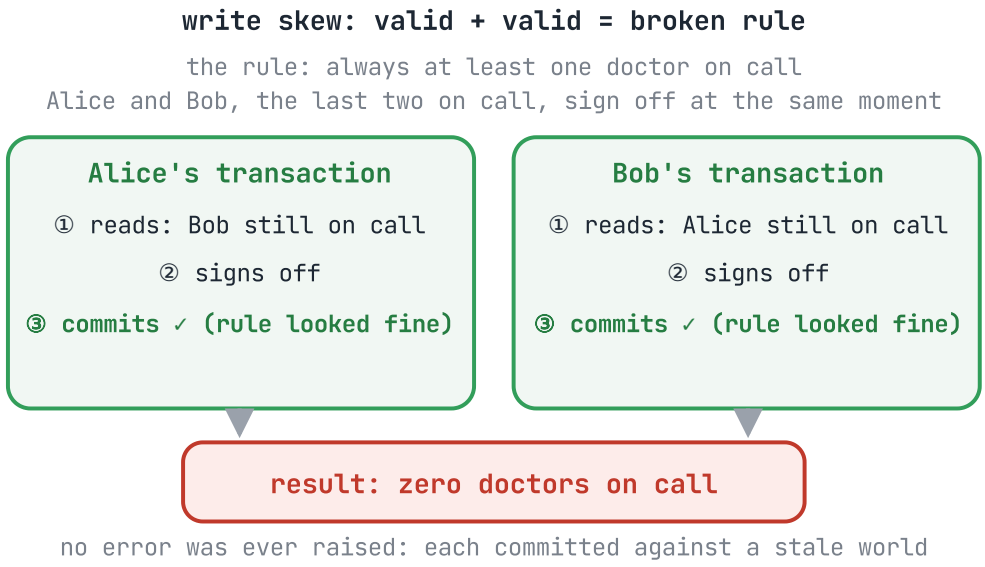


Figure 59 : Green + green = red: each transaction committed against a world where the rule still held; together they break it without triggering a single error.

With no global undo, you write the undo by hand: that is the **saga**. Placing an order = reserve the stock ①, charge the card ②, create the shipment ③; if ③ fails, your own code triggers the refund of ② then the release of ①. What the engine used to do for free becomes your job, step by step.

This trap is not waiting for you only at the scale of big distributed databases. Two threads of execution inside the same program (two threads, the cooks of section 2.6, or two goroutines, Go's lightweight threads) reading and writing the same variable replay write skew in miniature: each acts on a world already stale, and that is called a **race condition**. Same remedy as in the database: lock the access (a *mutex*, the lock that lets one thread through at a time), or better, do not share at all.

The distributed-systems moral fits in one line: the network lies, the clock lies; "suspicion and paranoia pay off".

↳ Designing Data-Intensive Applications · The Hard Parts

↳ WHICH LEADS TO CHAPTER 7

A system proven, shipped and held under load is finally ready to meet its real users. And that is exactly where technical certainties collide with reality: a hurried human who does not read, a team that grows, an attacker who probes. The tech was only the means; the **product for humans** *is the end.*

"Design is not just what it looks like and feels like. Design is how it works." — Steve Jobs, 2003

7 Software is for humans

product, team, attacker

Everything above serves one purpose: a human, at the screen, who wants to get something done. And around them, other humans: the other developer calling your API, the team that builds, and the attacker hunting for the flaw. This chapter looks at code through those people's eyes. The examples come from the web, where the most humans meet software, but each idea holds just as well for a checkout screen, a phone app or a terminal.

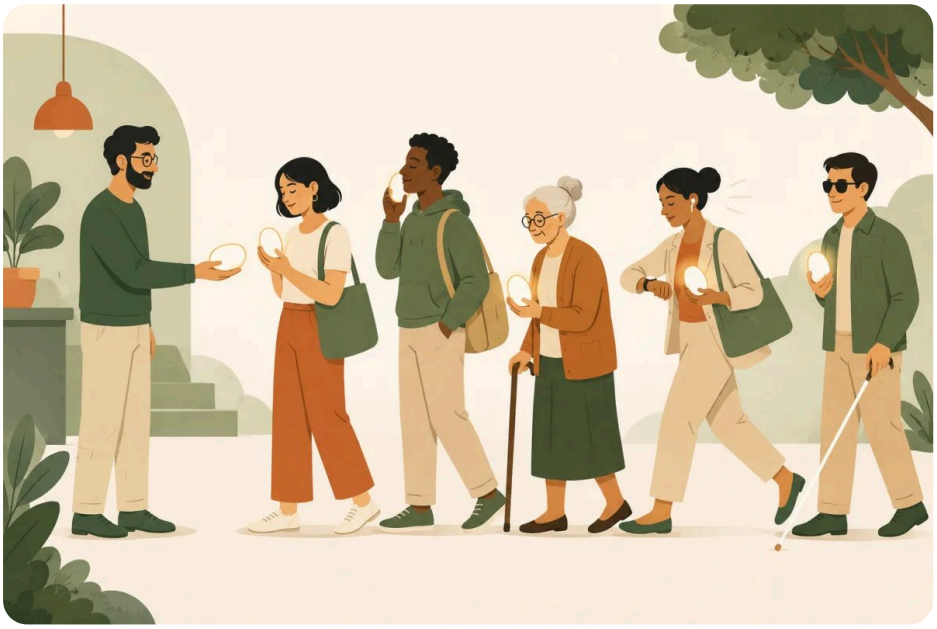


Figure 60 : Good software is grasped without a manual, by everyone, including the people we forget.

7.1 Don't make me think

A user does not read a page, they scan it, and every half-second of hesitation is a friction that drives them away. Hence the law usability consultant Steve Krug laid down in 2000, which gives the book its title: don't make me think. **Conventions beat creativity** (the magnifier top-right, the cart next to it) because the user finds them without thinking, on every site they already know.

In code, these conventions are written in HTML. A `search`-type field gives a "Search" key on the phone keyboard. A `label` ties each field to its caption and enlarges the click area. And `autocomplete` spares retyping something the browser already knows:

```
<input type="search">                          <!-- keyboard adapted on
mobile -->
<label for="ville">Ville</label>
<input id="ville" autocomplete="address-level2">   <!-- the city fills in
by itself -->
```

Navigation is not a feature of your site: it is the site, the same way the building, the aisles, and the cash registers are not extras added to a store: they are the store. Without them, there is nothing to enter, nothing to find, nowhere to pay.

The reason: the web has no physics. In a store you know appliances are "at the back left" because you walked there. Online there is no back, no left, no up. Navigation stands in for all three. A visitor who does not know where they are, where they came from, or how to search cannot do anything at all.

The trunk test reveals whether it works. Imagine being blindfolded, driven around, and dropped on a random page inside a site. That is exactly what happens to anyone who arrives from Google on your page 7. Squint: what site is this? What page? What are the major sections? Where am I in them? How do I search? If those do not pop off the page in twenty seconds, the navigation has failed. Finding those problems costs nothing: three users, one morning a month, a debrief over lunch. Krug observes that the first three users already run into most of the serious problems. Frequency beats ceremony.

Navigation is only part of it. Every visitor arrives with a **reservoir of goodwill** that each bad decision drains. You spent twenty minutes filling your cart: $45. You click "checkout". Shipping: $12. You close the tab. That is the reservoir run dry, by one number hidden too late. Filling it is the opposite: be transparent, forgive a format slip, never block the way with an animation.

↳ Don’t Make Me Think

7.2 An API is designed for its caller

The API, the front counter of 4.5, is the contract by which one program calls another, most often over the web. Its universal vocabulary is the **HTTP verbs**: `GET` to read, `POST` to create, `PUT` / `PATCH` to update, `DELETE` to remove. They state intent without reading the docs: `GET /users/42` guesses itself. And the golden rule of Arnaud Lauret in *The Design of Web APIs* is *consumer-first*: you don't start from your database, you start from what the caller wants to do, draw the ideal response for them, then build backwards, all the way to the database.

Three rules round out the contract:

- **Predictable names** grant superpowers: whoever has seen one route (an address of the API, like `/users/42`) can guess all the others.
- **Generous errors**: say what is wrong, where, and all at once, not one complaint per attempt.
- **Minimal data**: the safest data is the data you never send: expose the bare minimum.

```
// ✗ guessed by nobody          // ✓ guessed by everyone, and standard
{ "ACTBLNDFPRTF": true }       { "overdraftFacility": { "active": true } }
```

↳ The Design of Web APIs

7.3 Accessible by construction

A blind person does not see your button; their screen reader (the software that reads the page aloud, such as VoiceOver or NVDA) announces it, as three pieces of information: a **name** (the text read out), a **role** (what kind of thing it

is), a **state** (checked, open, disabled). When a native HTML element cannot express one of these, the **ARIA** (Accessible Rich Internet Applications) attributes, prefixed `aria-*`, let you supply it explicitly:

```
<button aria-pressed="true">Favorite</button>
// announced: "Favorite, button, pressed"
```

A clickable `<div>` has no role and no state: to a screen reader, it does not exist. Hence the first rule of ARIA: use the right HTML element, which provides all three for free. All of this is framed by a global standard, the **WCAG** (Web Content Accessibility Guidelines): four principles (Perceivable, Operable, Understandable, Robust) and three levels (A, AA, AAA). Two concrete moves cover the essentials. First, enough contrast: the 4.5:1 ratio on text is a measure computed from the luminance of the two colors, from 1:1 for identical colors to 21:1 for black on white. Second, keyboard navigation, which requires that you can see the **focus**, the element that receives the keystrokes, shown by a visible outline.

```
/* ✗ the most common mistake: removing the focus outline */
button:focus { outline: none; }
/* ✓ a visible focus: the keyboard user sees where they are */
button:focus { outline: 2px solid #005fcc; }
```

The ultimate free test: drop the mouse, walk your page with the Tab key. If you lose track, a keyboard user does too.

↳ Web Accessibility Cookbook

7.4 The organization shows up in the code

So far we have looked at software from the outside: the user, including the one we forget, then the developer who calls the API. Now let us look at who builds it. The organization itself is an interface. **Conway's law**: a system copies the communication structure of the organization that builds it. Four teams that

talk poorly will produce four modules that fit poorly, like it or not.

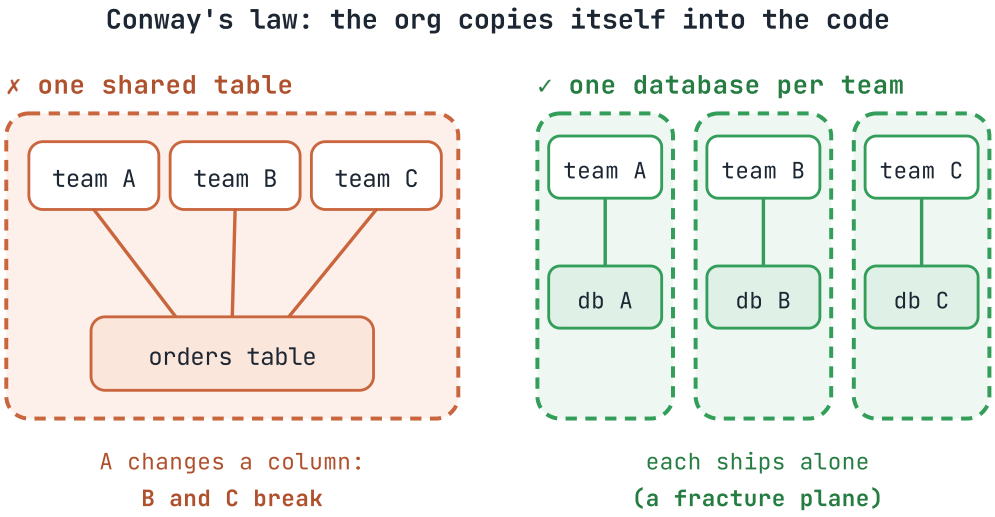


Figure 61 : The same drawing as the quantum (ch. 4), on the human side: one shared table force-couples the three teams; three databases, and each ships alone.

Team Topologies turns the law into a lever: if you want a certain architecture, organize the teams for it first. The hidden constraint is **cognitive load**: a team can only hold a bounded amount of domain. So you split the system along its **fracture planes**, its natural seams, most often the business domain. And the book gives the playbook, four team types:

- **stream-aligned**: one team = one product, shipped end to end (the default case);
- **platform**: provides internal tooling so the others ship without waiting;
- **enabling**: helps a team level up, then steps away;
- **complicated-subsystem**: maintains a piece too specialized to share (a calculation engine, say).

That fracture line along the business has a twin on the model side: Eric Evans's **Bounded Context**, a boundary within which every domain word keeps a single meaning. Two teams that share an "Order" object without that boundary end up with two incompatible definitions in the same table (the purchasing team means a purchase order, the warehouse means a delivery order) and the

monthly report that crashes. Drawing the boundary, one team, one model, a database of its own, is what lets "Order" mean something else elsewhere without breaking anything.

Either way around, the lesson is the same: drawing the teams is already drawing the system.

↳ Team Topologies · Domain-Driven Design · Learning DDD

7.5 Adding people to a late project makes it later

Intuition says: project is late, add developers. It is false, and it has a name, **Brooks's law**. The work divides badly, each newcomer must be trained (by the veterans, whom you therefore slow down), and above all they multiply the communication channels.

```
communication channels = n × (n - 1) ÷ 2
 5 people → 10 channels        // manageable
15 people → 105 channels       // 105 channels to maintain
```

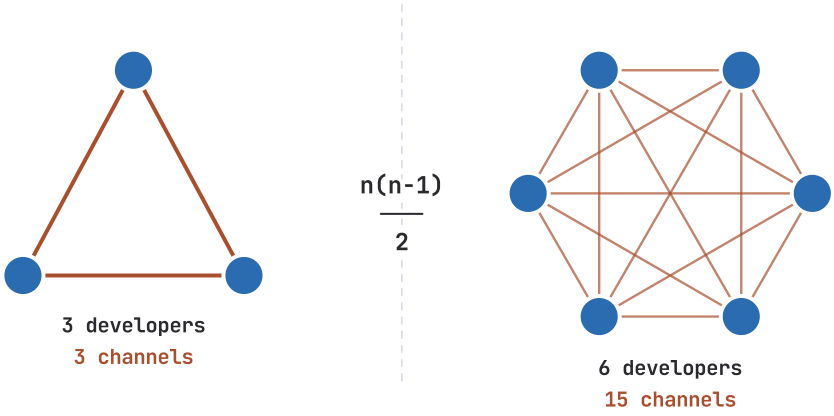


Figure 62 : From 3 to 6 people, the channels jump from 3 to 15: every newcomer multiplies the lines, not just the hands. At 15 people: 105 channels.

"The man-month as a unit for measuring the size of a job is a dangerous and deceptive myth. It implies that men and months are interchangeable." They are not: nine women do not make a baby in one month.

If headcount does not drive performance, what does? Cohesion. DeMarco and Lister call a **jelled team** one so tightly knit that the whole beats the sum of its parts: few people leaving, a shared identity, shared pride. You cannot manufacture one on demand, only create the conditions. But you can kill it fast, and they have a word for it: **teamicide**, anything that breaks a team up or keeps it from jelling. Watching people instead of trusting them, scattering them into separate offices, imposing deadlines everyone knows are fake, breaking up a team that works the moment a project ends: any one of these management reflexes is enough to crack it.

↳ The Mythical Man-Month · Peopleware

7.6 Think like the attacker to defend

The chapter's last figure, the least friendly: the attacker. First surprise: they did not choose you. Attacks are industrial: bots scan the web around the clock, with no particular target, and you are not attacked because you are interesting, but because you are reachable. Hence the thesis of *Web Application Security*, Andrew Hoffman's book: you only defend well what you know how to attack. The book is itself organized like a real attack: **reconnaissance** first (map the application, look for the service entrance rather than the front door), offense next, defense last.

What does the attacker find? Almost always the same sin: data coming from the user that the code treats as code, or takes at its word. **SQL injection** shows the whole mechanism. Your search engine pastes whatever the visitor types into a query. The attacker does not type a name: they type a quote that closes your text, then their own command:

```
search typed  : '; DROP TABLE users --
query executed: SELECT * FROM products WHERE name = ''; DROP TABLE users --
'
                // ✗ the quote closes the text: what follows becomes an
                // ORDER
```

The users table was just deleted by a search form. (Many drivers, the components that carry your queries to the database, refuse to chain two statements. The attacker then reads the whole table instead of destroying it. Same flaw.) The two cousins play the same note. **XSS** (*Cross-Site Scripting*) injects not SQL but a bit of script, slipped into the page through an HTML tag, which will run in other visitors' browsers. And **mass assignment** slips an extra field into the request, which the server saves without a second thought:

```
POST /api/profile  { "name": "Alice", "isMember": true }
user.update(req.body)   // ✗ isMember goes through → Alice self-promotes
```

The defense reads as a mirror image and holds in one principle: never trust the input. Against injection, **prepared queries**: the input travels separately from the query and can never become code again. Against XSS, escape everything you display: turn special characters into harmless text. A tag's `<` becomes `<`, which the browser displays instead of executing. Against mass assignment, a whitelist of accepted fields. And on top, **defense in depth**: each layer protects itself, from the browser to the database, so that if one gives way, the others hold.

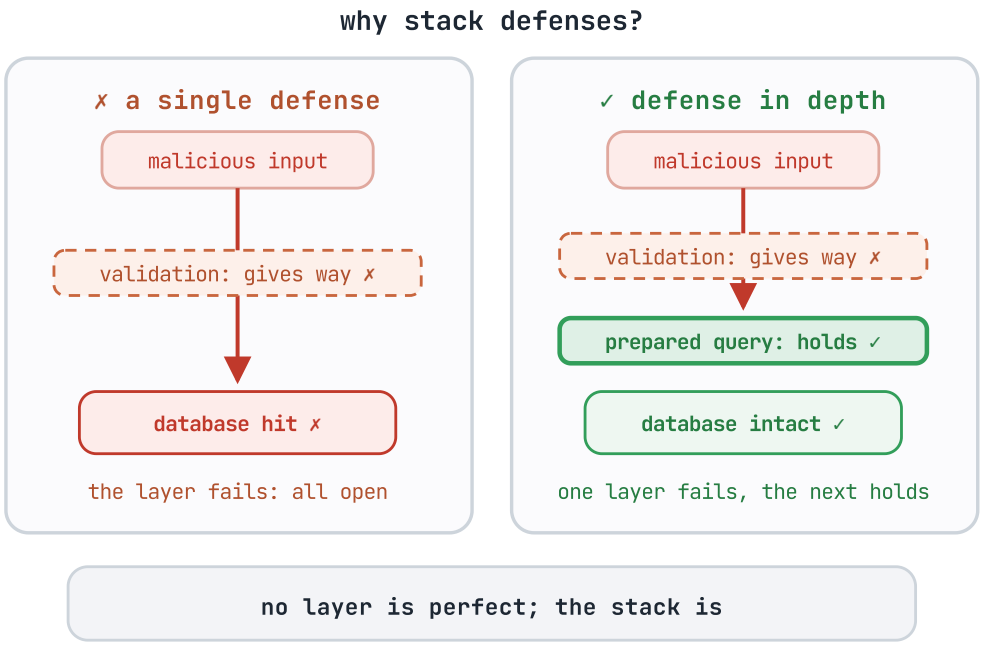


Figure 63 : Same malicious input, same layer giving way: only the stacking changes the outcome. That is the whole bet of defense in depth.

Injection is only the first door. The second is even simpler, and it is the most widespread flaw on the web in the OWASP ranking: `GET /users/42` returns your record, so the attacker types `/users/43` and reads the neighbor's. **Authentication** checks who is speaking. **Authorization** checks whether they have the right to. Here, the server did the first and forgot the second: every request must check that the requested record belongs to whoever asks.

Third door, passwords. They are never stored in clear, nor even encrypted, because a key can be stolen: you keep only the hash, the result of a hash function (1.3), a summary impossible to reverse, computed by a function built to be slow (bcrypt, argon2), so that a stolen database yields no password. Finally **least privilege**: each program and each account has only the rights it needs. The account the shop uses to talk to its database has no reason to be able to drop a table, and the injection from earlier would have failed for that reason alone. It is the least power of 2.7, applied to rights.

One last reflex to copy from the people in the trade: they say "mitigations", never "fixes". You reduce the risk, you do not erase it; no defense is ever final.

↳ Web Application Security

↳ WHICH LEADS TO CHAPTER 8

This whole craft, from the bits up to the team, has just been shaken by an entirely new actor able to write code on demand: AI. It does not replace the prior knowledge, it makes it **more necessary** *than ever: someone has to judge what it produces, and judging well demands precisely everything we have just climbed.*

"Perfection is achieved not when there is nothing more to add, but when there is nothing left to take away." — Saint-Exupéry, *Wind, Sand and Stars*, 1939

8 Coding in the age of AI

the last link

A new floor has settled on the seven below: a machine that writes code on demand. The question is no longer "can AI code?" (yes, often), but "what is left for the human?". One expects a technical answer, some corner the machine cannot code yet. The answer lies elsewhere: what is left for the human is deciding whether what the machine produces is correct, safe and in the right place. And that ability to decide is exactly what the seven floors you just climbed have built: you know how to recognize an honest name, an architecture that holds and a test that proves something. You did not make that climb for nothing: everything you learned on the way up is precisely what the arrival of AI makes indispensable.

Before climbing this last floor, one confusion to clear. "Doing AI" today almost never means training a model: it means calling an already-trained one and building around it. Nobody redraws the map of the world to show directions on their site: you embed Google Maps. The AI engineer does the same: the model is their map of the world, drawn by someone else. Their work starts after that: what you send the model, what you do with its answer, and how you

check the whole thing. That is what *engineering* means in *AI Engineering*, the book by Chip Huyen that feeds this chapter: everything plays out around the model, not inside it.

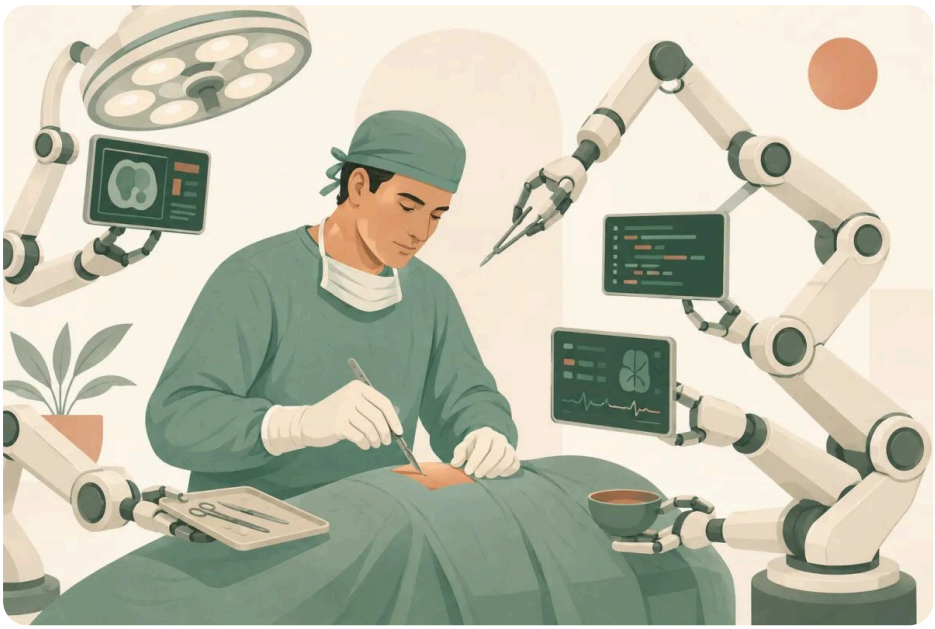


Figure 64 : AI hands over the instruments at full speed; the human holds the scalpel and decides the move.

8.1 AI generates the probable, not the true

A **language model** is the program behind ChatGPT or Claude: it was trained on enormous amounts of text to predict the next word. It does not know what is true: it produces, word after word, the most probable continuation of what came before. **Hallucination** (a false statement delivered with the confidence of a true one) is therefore not a bug to be fixed one day, it is the very mechanism

that makes it work: "anything with a non-zero probability, no matter how far-fetched or wrong, can be generated by AI". Ask it for the author of an obscure book: with the very same confidence, it will hand you a plausible, false name.

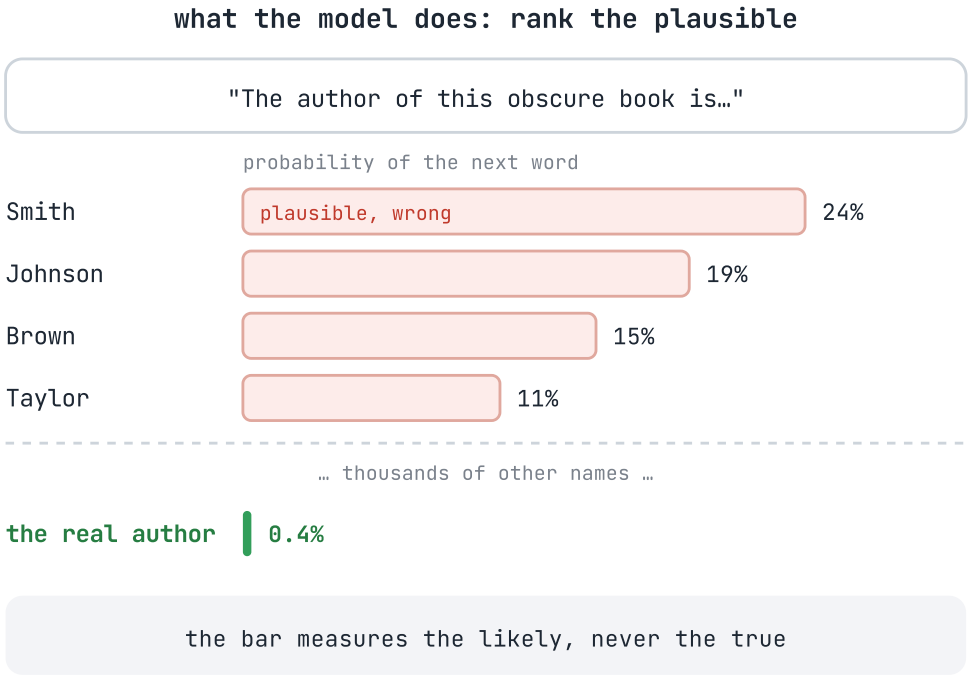


Figure 65 : The percentages are illustrative, the scale is faithful: the model has no way to see that the smallest bar is the right one.

Trust therefore gets calibrated by the nature of what you are reading. The logic of an argument can be judged on the spot: everything is right in front of you. A reference (a function name, an option, an API), on the other hand, always gets verified: a name that "sounds right" is exactly what a generator of the probable knows how to produce. Good news, execution does not lie: call a function that does not exist and it crashes, compiled language or not. The line still has to run, though: an error only shows up on the path you actually take, and that is exactly what a test guarantees.

Configuration options, though, are the exception: an invented option is often silently ignored, because many libraries skip unknown keys without complaint. The runtime error remains your best ally against the model's confidence; silence, on the other hand, proves nothing.

The consequence governs the whole chapter: you never trust blindly, you verify, and you design the system around that uncertainty rather than against it.

↳ AI Engineering · Programming Pearls

8.2 Everything turns on context

A model fails first when it lacks information. Giving it the right context at the right moment has become the central skill of the trade. The go-to pattern is called **RAG** (*Retrieval-Augmented Generation*: generate while leaning on retrieved documents). The principle: fetch the relevant documents from an external base and paste them into the **prompt** (the message you send the model), so it leans on supplied facts, not its fuzzy memory.

Why fetch pieces instead of handing over everything? A model only reads a limited amount of text at once: its **context window**. It is measured in **tokens**, pieces of words (a common word is one or two). Beyond that, the rest simply does not fit.

Concretely, this is what your coding assistant does when it answers correctly about a whole project: the project does not fit in the context window, so the assistant goes and fetches the right pieces. Each tool has its method: Claude Code (Anthropic's AI coding assistant) fires `grep` searches (looking for the exact word across all files), Cursor (an AI-powered code editor) indexes the

project as vectors to search by meaning. Searching by meaning rests on the `embed` function: it turns a text into a vector, a list of numbers encoding its meaning, and two texts about the same thing get neighboring vectors:

```
question = "Where is VAT calculated in this project?"

excerpts = project.search(embed(question), top_k=3)
# 1. top_k=3: keep the 3 passages of YOUR code
#    whose meaning is closest to the question
# → ["function grossPrice(net) { return net * 1.2; }", …]
#    found without the word "VAT": search by meaning

prompt = f"Answer from these excerpts: {excerpts}\n\n{question}"
# 2. paste the excerpts into the prompt, with the question

answer = model(prompt)
# 3. the model answers by reading your code, not guessing
```

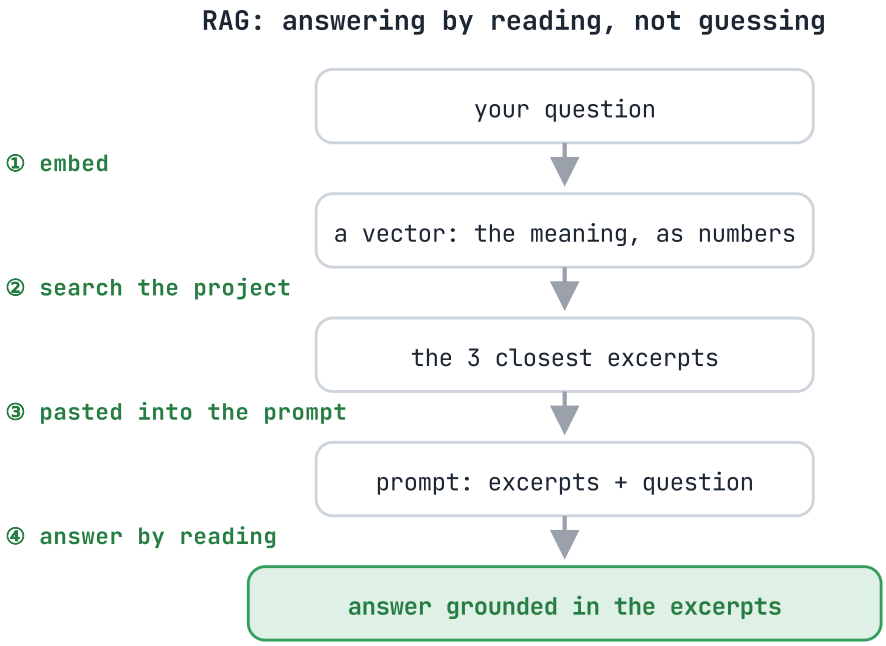


Figure 66 : The model no longer digs through its fuzzy memory: it reads the excerpts just handed to it, then answers. From one tool to the next, only step ① changes: grep matches the exact word, embed matches the meaning.

Context is also shaped in the *prompt* itself, and the difference is brutal:

```
# ✗ vague → the model guesses, unpredictable answer
"it's broken, fix it"

# ✓ role + context + enforced format → usable answer
"You are a senior PHP dev. This test fails (message below).
 Propose the minimal fix, as a diff, without rewriting the rest."
```

A session with an assistant starts from scratch: every time, you would have to repeat the project's conventions, the commands to run, the known traps. The fix: write those instructions once and for all in a file at the root of the project, Claude Code's `CLAUDE.md`, which the assistant rereads at the start of every session. It is the difference between re-explaining the job to an intern every morning and setting up a desk where everything is already in place. The trade has named that work *context engineering*: preparing the assistant's working environment, so that even a mediocre question produces a good result.

One last word you will meet in the trade: **fine-tuning**, retraining a model on your own data. Chip Huyen's criterion fits in one formula: "finetuning is for form, and RAG is for facts". Fixing the form is a job for teams building AI products, costly in data, compute and maintenance: a developer coding with the model will probably never touch it. Your remedy is always called context: the right excerpts, the right instructions, at the right moment.

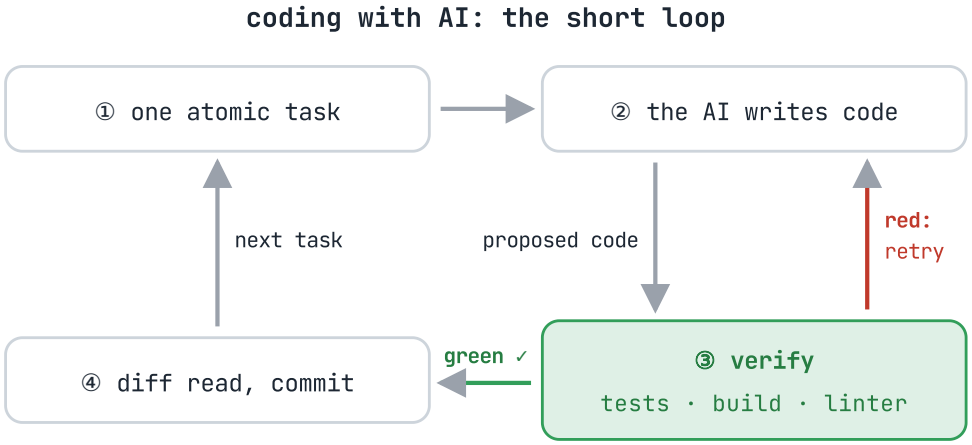
↳ AI Engineering · Programming Pearls

8.3 AI is steered in short loops

Hand AI a whole feature and you get two thousand plausible lines back. If they do not work, nobody can tell which of the dozens of stacked choices is at fault: not you, not the AI. Coding with AI is not a sprint, it is a short loop: one atomic task (one small thing at a time), the tests green, the diff read (the exact list of lines the AI added or removed), a commit, then the next one.

Every link of that loop comes down from the previous floors. Test-first becomes an executable contract (6.1): it turns "make something that works" into "make these assertions pass" (the checks written in the test), a target the machine can aim at and rerun on its own. Git (6.3), for its part, takes back its role as the net: you commit a clean state before letting the AI loose, and the diff tells you what the AI really did, not what it claims it did.

The most advanced tool, the **autonomous agent** (Claude Code is one), is just that loop automated: gather the context (the ticket, the files involved, the last error), act, verify, repeat. Everything plays out on the third beat: the verification. It must be an external, objective signal: the tests, the build (the automatic assembly of the application), the linter of 3.1. Never the agent's opinion of itself: it will always tell you it succeeded, with the same aplomb it puts into inventing a function that does not exist. Having a second agent reread the work with fresh eyes (the code review of section 6.3, handed to a machine) helps, because it has nothing to defend. But its verdict is still an opinion. Tests have no opinion.




```
the autonomous agent is just this loop, automated;
the verdict stays external (tests, build, linter)
```

Figure 67 : While it is red, the loop replays; only on green do you read the diff, commit, and take the next task.

In the trade, **evaluation** is the verdict on what the model produces, coming from outside the model. It, not the model, is the real bottleneck. For code, part of the verdict is automatic: it compiles, the tests pass. But that verdict is only a floor, not a grade. Green tests say nothing about readability, about the place in the architecture, about the security flaw lying dormant. The real judgment of code is multi-criteria, and the human carries it.

For an answer in plain text, there is not even a floor: nothing ever crashes. The grid has to be built yourself, question by question:

- nothing **invented**?
- do the quoted facts come from the **supplied excerpts**?
- is the requested **format** respected?

Without a grid you iterate blind, exactly like code without tests.

↳ AI Engineering · Course · Coding with AI

8.4 The surgical team, finally feasible

AI does not just change how you write: it redraws the team. Brooks dreamed in 1975 of a **surgical team**: one brain holding the scalpel (designing, deciding), surrounded by specialized assistants. One knows all the code, another remembers every corner of the language, a third forges the scripts and tools.

The dream ran into a dilemma Brooks called cruel: a few good minds keep the system coherent, but move too slowly for large projects. AI dissolves the dilemma: you remain the single brain, it supplies the hands. The assistant knows all the code, remembers every corner of the language, forges the scripts, and it plays all three roles at once. The model imagined fifty years ago becomes practical the day the hands are a machine. Brooks's law loses nothing (7.5): these hands cost little in training, which comes down to the context you hand them (section 8.2), and they add nobody to the meeting. One cost no context removes: reading over what they produce.

Remember Conway (7.4): the shape of the team copies itself into the system. A team redrawn as one brain and machine hands will produce a different piece of software, and you are the one deciding which.

You are the chief surgeon, and the metaphor translates move by move. The instruments handed to you are the generated diffs: you accept one, reject two, send the third back with a sharper request. The moves you decide are the architecture, the boundaries, the names: the assistant proposes, it never decides. And the report you sign is the commit: your name on it, not the machine's. You never sign off on code you have not read.

↳ The Mythical Man-Month · AI Engineering

8.5 Human judgment is the last link

If AI writes the code, what is left for the human? **Everything above.** To judge whether the generated code is right, you must:

- understand what it **costs** (chapter 1);
- know whether the language **expresses** it well (chapter 2);
- judge whether it is **readable** by the next human (chapter 3);
- see whether it fits the **architecture** (chapter 4);
- know what it will pay when the data **travels** (chapter 5);
- be able to **prove** it and ship it (chapter 6);
- check that it **serves** the human, and resists the attacker (chapter 7).

AI produces the probable; **you decide what is right.** The faster it writes, the rarer your judgment becomes. The dividing line in the profession now runs here: on one side, the developer who understands, tests and can explain every line they ship. On the other, the one who accepts without reading and crosses their fingers: the field already has a name for it, vibe coding. One rule to stay on the right side: never ship a line you could not explain.

That judgment starts even before the first line, in how you frame the problem. When a programmer once asked Jon Bentley how to sort a file on disk, fifteen minutes of questions replaced a week of code: the real need was ten million distinct small integers in about a megabyte of memory. The right answer sorted nothing: one single-bit box per possible number, ticked if the number appears in the file, then you read the boxes back in order and the list comes out sorted by itself. Defining the real problem was ninety percent of the battle. An AI does not ask those questions for you: hand it "sort this file" and it will dutifully deliver the week-long sort, never the ten-second shortcut.

This whole book does not teach you to code instead of the AI: it teaches you to know when it is wrong. Nothing guarantees it will think, on its own, of storing an amount in cents (exact integers) rather than a float (approximations) (1.1): you have to be the one who knows.

↳ AI Engineering · Programming Pearls

↻ THE LOOP CLOSES

And here is the vertigo: that judgment is something AI cannot hand to you. It imitates the answers of those who know; it does not know when an answer is right. That discernment is earned by making your own mistakes and fixing them: it is the one part of the craft no assistant will ever shortcut for you. You have to climb the eight floors yourself.

"The question of whether machines can think is about as relevant as the question of whether submarines can swim." — Edsger Dijkstra



Eight levels, one thread: from the numbers in the silicon to the judgment that no model replaces. By the end you no longer decide blind: you know what to do, why, where, when and how. That is the whole field, and the notes below hold the detail of every step.

WHAT NOW?

Don't reread everything. If you are starting out, go back to the floor where the ground gave way: each chapter leans on a handful of book notes (the list is right below), and they hold the detail. If you have been coding for years, two chapters will change your next weeks more than the rest: architecture, because it is decided early and paid for over a long time, and AI, because it can be steered.

Reading is not enough: these ideas are learned through your fingers, and every floor has its training ground in the site's free interactive courses.

> **The language (chapter 2)**: the JavaScript (www.web-developpeur.com/en/learning/javascript/), PHP (www.web-developpeur.com/en/learning/php/), Python (www.web-developpeur.com/en/learning/python/) or Go (www.web-developpeur.com/en/learning/golang/) courses.
>
> **Prove it and ship it (chapter 6)**: the Testing (www.web-developpeur.com/en/learning/tests/) and Deployment (www.web-developpeur.com/en/learning/deploiement/) courses.
>
> **Software for humans (chapter 7)**: the Accessibility (www.web-developpeur.com/en/learning/accessibilite/) and Web security (www.web-developpeur.com/en/learning/securite/) courses.
>
> **Coding with AI (chapter 8)**: the course of the same name (www.web-developpeur.com/en/learning/coder-avec-ia/), nineteen lessons of method.

And keep this page within reach: it never reads the same twice. The architecture chapter will not say the same thing before and after your first real overhaul. The day a section feels obvious, you have climbed a floor.

Full book notes: web-developpeur.com/en/library/

THE BOOKS WOVEN INTO THIS THREAD

1. Write Great Code vol. 1 · vol. 2 · Grokking Algorithms · Programming Pearls
2. Eloquent JavaScript · Learning Go · Effective TypeScript · Fluent Python · PHP 8 Objects & Patterns · Domain-Driven Design · Learning DDD
3. Clean Code · A Philosophy of Software Design · The Pragmatic Programmer · Refactoring
4. Head First Design Patterns · Design Patterns (GoF) · Clean Architecture · The Hard Parts
5. High Performance Browser Networking · Computer Networks: A Systems Approach
6. TDD by Example · Working Effectively with Legacy Code · Pro Git · Continuous Delivery · The Phoenix Project · Accelerate · Data-Intensive Applications · Programming Pearls · Code Complete · Peopleware
7. Don’t Make Me Think · The Design of Web APIs · Web Accessibility Cookbook · Team Topologies · Peopleware · Domain-Driven Design · Learning DDD · The Mythical Man-Month · Web Application Security
8. AI Engineering · Programming Pearls

Glossary

Each definition is the sentence of the book that introduces the term. The number points to the section.

ACID

The transaction of 4.6 is all-or-nothing there: the transfer debits AND credits, or does nothing at all. That is the ACID contract: atomic (all or nothing), consistent (the database's rules stay true), isolated (each transaction believes it is alone), durable (once committed, it is written). 6.7 p. 138

address

Laid end to end, these bytes make up memory: one huge row of slots, each tagged by a number, its address. 1.1 p. 5

AIMD

This coded politeness is called AIMD (Additive Increase, Multiplicative Decrease: add a little when all goes well, multiply by a fraction on loss). 5.4 p. 118

API

An API (application programming interface) is an application's front counter: other programs drop off their requests there, with a form to fill in and a set response format (section 7.2 comes back to it). 4.5 p. 104

architecture quantum

The first tool, the architecture quantum: the smallest piece of a system you can put online and test alone, and that can fall without dragging anything else down. 4.4 p. 102

argument

A function is a recipe written once and for all: you hand it ingredients, its arguments, it hands back a dish, its result. ch. 2 p. 32

ARIA

When a native HTML element cannot express one of these, the ARIA (Accessible Rich Internet Applications) attributes, prefixed aria-*, let you supply it explicitly. 7.3 p. 143

array

An array puts its elements side by side, like numbered lockers: reaching the 500th is instant, you compute its address and go, but inserting in the middle forces you to shift every following element by one slot. 1.3 p. 21

assertion

Failing early has a tool, the assertion: a line that checks what must always be true, and stops everything otherwise, like assert total >= 0. 3.4 p. 77

asynchronous (async)

Code that keeps going without waiting for the answer is called asynchronous. 2.6 p. 62

Authentication

Authentication checks who is speaking. 7.6 p. 147

Authorization

Authorization checks whether they have the right to. 7.6 p. 147

autonomous agent

The most advanced tool, the autonomous agent (Claude Code is one), is just that loop automated: gather the context (the ticket, the files involved, the last error), act, verify, repeat. 8.3 p. 155

bandwidth

Bandwidth is the number of lanes: how much data passes abreast every second. 5.1 p. 113

base case

The base case: the box that holds no more boxes. 2.5 p. 58

Big O

Big O is the unit that says how the number of steps grows with the input size, written n (here, the number of names). 1.3 p. 21

bit

A bit is a switch: off, it is worth 0, on, it is worth 1. 1.1 p. 5

blob

A blob is a file's content. 6.3 p. 129

boolean

A number, like the €45 price, fits whole in the locker: its 0s and 1s are right there. A boolean, a value that can only be true or false, fits too. 2.2 p. 39

Bounded Context

That fracture line along the business has a twin on the model side: Eric Evans's Bounded Context, a boundary within which every domain word keeps a single meaning. 7.4 p. 144

boy-scout rule

The daily reflex fits in one image, Clean Code's boy-scout rule: like the scout who leaves the campsite cleaner than he found it, always leave the file a little cleaner than you found it. 3.6 p. 83

branch

A branch, here main, is just a pointer to a commit, moved forward with each new commit. 6.3 p. 129

breakpoint

The debugger stops the program on a chosen line, the breakpoint, and shows the contents of every locker: it is the by-hand trace of 2.1, run by the machine. 6.2 p. 127

Brooks's law

Intuition says: project is late, add developers. It is false, and it has a name, Brooks's law. 7.5 p. 146

byte

You group the switches by eight, and a group of eight is called a byte. 1.1 p. 5

cache

Everything else lives further away, in layers: the cache (a fast reserve glued to the chip, keeping a copy of recent data), then RAM (the working memory, the memory sticks in your machine), then disk. 1.2 p. 15

cache line

The machine never fetches a single byte: like a courier filling a crate, it pulls a whole cache line (about 64 bytes) at once. 1.2 p. 15

call stack

Each call must therefore note the instruction to resume at and its own values (the n that is 3, then 2), and these notes pile up, one per opened box, until the smallest one is empty: that is the call stack. 2.5 p. 58

callback

The traditional way to collect the server's answer is to hand over a callback, a function to be run later: "when you are done, run this". 2.6 p. 62

caller

The real question is not "how many lines?" but "is it simpler for the caller?", meaning the code that uses your function. 3.2 p. 73

CDN

Move the content closer to the visitor with a CDN (Content Delivery Network), meaning relay servers spread around the world, since no subscription shortens the Atlantic. 5.5 p. 120

characterization test

Faced with old code that has no tests, you first write a characterization test: it records what the code really does, not what it should do, before you change it. 6.1 p. 124

CI/CD

CI/CD is the automatic chain that makes the burst possible. 6.5 p. 133

class

A class is a mold: the User class states that a user has a name, an email, and knows how to sign in. 4.1 p. 88

closure

The closure: a function that leaves with its own locker. 2.4 p. 51

code editor

A code editor is software for typing text, like a word processor but made for code (VS Code, for example). 1.4 p. 26

code review

You propose your branch, which is called a pull request (a request to merge), with the exact list of changed lines, and a colleague reads it before it is glued back. That is code review: the next human of chapter 3 shows up sooner than planned, not in six months but tomorrow morning. 6.3 p. 129

coercion

JavaScript, the best-known example, converts a value from one type to another on its own, for example a text into a number: that is coercion. 2.3 p. 44

cognitive load

The hidden constraint is cognitive load: a team can only hold a bounded amount of domain. 7.4 p. 144

collision

Two keys can produce the same number: that is a collision. 1.3 p. 21

comment

A comment is a note written for humans in the middle of the code, after // or # depending on the language, like the gray lines in this book's examples: the machine ignores it. 3.1 p. 71

commit

A commit points to a tree and to its parent. 6.3 p. 129

compiler

The compiler translates all the text of the program before it runs, like a book translated in full before it is published. 1.4 p. 26

complicated-subsystem team

Complicated-subsystem: maintains a piece too specialized to share (a calculation engine, say). 7.4 p. 144

composition

Composing means assembling: "a user has a way of signing in" means it holds a separate object, from another mold, which can be replaced without touching the User mold. 4.1 p. 88

concurrency

Concurrency is having several tasks in progress, even if only one moves at any given instant, like a lone cook moving each order forward in turn. 2.6 p. 62

Configuration

Configuration is a setting the program reads at startup instead of carrying it in its code, and its values often differ between the two machines. 6.2 p. 127

Congestion collapse

Congestion collapse is the vicious circle in which resent packets lengthen the jam that made them go missing. 5.4 p. 118

conjoined functions

Two functions you can only read together are conjoined functions (A Philosophy of Software Design calls them conjoined methods), and you are better off merging them into one. 3.2 p. 73

connection

A connection is the conversation two machines open with a handshake, each keeping track in memory of where it stands. 5.3 p. 116

constant

A constant is a variable you fill once and never write to again, often spelled in capitals so that it stands out at a glance. 3.1 p. 71

context window

A model only reads a limited amount of text at once: its context window. 8.2 p. 153

Conway's law

Conway's law: a system copies the communication structure of the organization that builds it. 7.4 p. 144

core

Your computer keeps dozens of programs open at the same time, but its chip has only a few cores, computing units that each follow only one thread at a time. 2.6 p. 62

CPU

The processor, or CPU (the chip that runs your instructions one by one) only computes fast on what sits right next to it. 1.2 p. 15

data

Orders act on data, meaning numbers such as a price, a letter or a pixel. 1.1 p. 5

data structure

Big O also helps choose a data structure, the way a collection is laid out in memory. 1.3 p. 21

database

A database is a set of data kept on disk by a specialized program that, unlike a text file, keeps three promises. 4.6 p. 107

debugger

The debugger stops the program on a chosen line, the breakpoint, and shows the contents of every locker: it is the by-hand trace of 2.1, run by the machine. 6.2 p. 127

decorator

The decorator: a sticky note stapled onto a function. 2.4 p. 51

deep function

The deep function hides a lot of work behind a tiny surface, meaning what the caller must know to use it (here one name and one argument): one line on the menu, everything else in the kitchen. 3.2 p. 73

defense in depth

On top, defense in depth: each layer protects itself, from the browser to the database, so that if one gives way, the others hold. 7.6 p. 147

dependency

A dependency is a piece of code that another needs in order to work. ch. 4 p. 87

dependency injection

Handing a function what it needs (here, the output), instead of letting it build that itself, is called dependency injection. 4.2 p. 91

dependency rule

The dependency rule of Clean Architecture forbids one single thing: the business code depending on the technology. 4.3 p. 95

dictionary

A list stores values one after another, a dictionary finds them by a name. 2.2 p. 39

disk

Everything else lives further away, in layers: the cache (a fast reserve glued to the chip, keeping a copy of recent data), then RAM (the working memory, the memory sticks in your machine), then disk. 1.2 p. 15

DNS

DNS (Domain Name System): the web's directory turns the site's name into an IP address, because a router knows how to forward towards an address, never towards a name. 5.5 p. 120

DOM

A web page: its HTML text (the language that describes the page) is made of nested tags, a paragraph inside a section inside the page, which the browser arranges as a tree, the DOM (Document Object Model). 2.5 p. 58

DORA metrics

Accelerate gives four numbers to find out objectively, the DORA metrics (DevOps Research and Assessment, the research team behind the book). 6.5 p. 133

DRY

DRY (Don't Repeat Yourself) is the principle that a given piece of knowledge, like the VAT rate, should live in only one place in the program. 3.5 p. 81

dynamic typing

Dynamic typing checks the types while the program runs, so you find out about the mistake when the code runs: Python, JavaScript. 2.3 p. 44

emergent design

The deepest effect of TDD (Test-Driven Development) is not catching bugs, it is emergent design: writing the test first, you design the API, that front counter from section 4.5, from the point of view of whoever comes to drop off a request, not of whoever staffs the desk. 6.1 p. 124

enabling team

Enabling: helps a team level up, then steps away. 7.4 p. 144

entity

An entity has an identity that outlives its contents: customer 42 is still customer 42 after moving house and changing name, you follow it over time. 4.5 p. 104

enumeration

A type that lists its values one by one, like the three colors, is an enumeration (enum in many languages). 2.3 p. 44

error

Take the average price of an empty cart: you must share €0 among 0 items, and sharing among nobody makes no sense. Many languages then stop the program and print an error, a message saying what failed and on which line. 2.1 p. 33

ETC

ETC (Easier To Change) says that at every choice, you ask one question: "will this make the system easier, or harder, to change?" It is a compass more than a rule. 3.5 p. 81

evaluation

In the trade, evaluation is the verdict on what the model produces, coming from outside the model. 8.3 p. 155

event loop

The event loop is the mechanism that keeps taking the next task from the queue of tasks (a click, a server answer, a timer: each one drops a task in it). 2.6 p. 62

execution plan

The execution plan is the method the engine has chosen to answer a query. 4.6 p. 107

exponent

In binary, the machine stores the digits on one side, called the mantissa, and on the other the exponent, the power of 2 to multiply them by. 1.1 p. 5

fail fast

Fail fast (a phrase popularized by Jim Shore in 2004) is the rule that a program should stop as soon as it meets a case with no right answer, rather than carry on with a wrong value. 3.4 p. 77

field

The object's pieces of information are called fields: name, price, quantity. 2.2 p. 39

file

A file is like a sheet of paper kept on the disk, with a name. 1.4 p. 26

fine-tuning

One last word you will meet in the trade: fine-tuning, retraining a model on your own data. 8.2 p. 153

float

In binary, the machine stores the digits on one side, called the mantissa, and on the other the exponent, the power of 2 to multiply them by. A float is a number stored as a mantissa and an exponent: the point has no fixed place, the exponent slides it, hence the name "floating point". 1.1 p. 5

flow

Crossing it is not about coding more: it is about installing a flow that proves, ships and holds, without heroics and without all-nighters. ch. 6 p. 123

formatter

The formatter rewrites spaces, line breaks and indentation so the whole project looks the same, whoever wrote it. 3.1 p. 71

function

A function is a recipe written once and for all: you hand it ingredients, its arguments, it hands back a dish, its result. ch. 2 p. 32

functional programming

Functional programming lets functions travel like data (2.4) and wants them never to change what they receive: instead of altering the list it was given, a function returns a new one with the change made. 2.7 p. 67

garbage collector

The garbage collector is the mechanism of the language that frees memory that no variable can reach anymore. 2.4 p. 51

general case

The general case: the box that holds another one. 2.5 p. 58

generator

The generator (yield): a function that hands you one value, stops, and resumes with the next when you ask again. 2.4 p. 51

GoF

The GoF (the "Gang of Four", the four authors of the founding Design Patterns book) sorts its 23 patterns into three families, and those three families beat learning the 23 names by heart. 4.2 p. 91

hallucination

Hallucination (a false statement delivered with the confidence of a true one) is therefore not a bug to be fixed one day, it is the very mechanism that makes it work: "anything with a non-zero probability, no matter how farfetched or wrong, can be generated by AI". 8.1 p. 151

hash function

The key goes through a hash function that turns it into a cell number: you search for nothing, you compute where the value sits and go straight there, whatever the number of entries. 1.3 p. 21

hash table

A hash table stores each value under a key, a customer's record under their name for instance: to find it, you give the name and get the record at once, in constant time (on average), even over a million entries. 1.3 p. 21

Hexadecimal

Hexadecimal is a way of writing numbers in base 16, where one digit, 0 to 9 then a to f, stands for four bits: two digits per byte. 1.1 p. 5

HTTP

HTTP (HyperText Transfer Protocol) is the language browsers and servers speak: ask for a page, answer with its contents. 5.5 p. 120

HTTP verbs

The API, the front counter of 4.5, is the contract by which one program calls another, most often over the web. Its universal vocabulary is the HTTP verbs: GET to read, POST to create, PUT/PATCH to update, DELETE to remove. 7.2 p. 143

HTTP/1

HTTP/1 can only ask for one resource at a time per connection, and a browser opens only six of them per domain (per site name, such as mysite.com or images.mysite.com). 5.5 p. 120

HTTP/2

HTTP/2 fixed the connection problem: every request travels interleaved over one single connection, so one handshake only, and a window that stays open and wide. 5.5 p. 120

HTTP/3

HTTP/3 goes further still: it drops TCP for a new transport, QUIC (pronounced "quick"), because one single TCP connection has a flaw. 5.5 p. 120

idempotent

An operation that can be repeated without a double effect is called idempotent. 5.2 p. 114

immutable

Data that cannot be changed in place is immutable: it is frozen, like printed text, and you can only build another one. 2.2 p. 39

index

An index is the sorted directory of section 1.3, kept up to date by the engine for a given column, so that a search on that column drops to a few steps instead of ten million. 4.6 p. 107

inheritance

Inheriting says "an admin is a user", which the extends keyword writes: the child mold (Admin) takes everything from the parent mold (User) at once, without asking. 4.1 p. 88

input

The input of a program is everything it receives from outside: a key pressed on the keyboard, a file on the disk, a barcode, a message from another machine over the network (chapter 5). 2.1 p. 33

instance

An instance is the copy made from a class: the object seen in section 2.2, with values of its own (Smith, smith@example.com). 4.1 p. 88

instantiate

Making a copy from the mold is called instantiating the class, which is what the new keyword does in many languages (Java, C#, PHP, JavaScript). 4.1 p. 88

instruction

A program is a list of instructions, tiny orders the processor, the computing chip, knows how to carry out: "add these two numbers", "copy this cell elsewhere", "jump to number 40 if the result is zero". 1.1 p. 5

integer overflow

An integer overflow is a computation whose result no longer fits in the bytes set aside for it: in languages with fixed-size integers, the number then usually wraps around to the bottom of the range, without warning. 1.1 p. 5

interface

A type can also describe what you require from an object. This list of requirements is called an interface: a contract of the kind "to get in here, you must have this". 2.3 p. 44

interpreter

The interpreter translates while the program runs, like a conference interpreter: sentence after sentence. 1.4 p. 26

intersection

The intersection (A & B) keeps only the values they have in common: green. 2.3 p. 44

invariant

An invariant is what must stay true whatever the program does: a till total is never negative, a paid order has an amount. 3.4 p. 77

IP

In the middle of the network, every machine shares one single protocol, a conversation rule they all agree on: IP (Internet Protocol), which gives each machine an address and moves packets toward it. 5.2 p. 114

jelled team

DeMarco and Lister call a jelled team one so tightly knit that the whole beats the sum of its parts: few people leaving, a shared identity, shared pride. 7.5 p. 146

join

Gluing an order and its customer back together when reading is a join: for each order, go fetch the customer row that carries that number. 4.6 p. 107

key

(Nothing to do with the hash table of 1.3: same word, different thing.) Each row carries a number of its own, its key: what you hand over to get the row. 4.6 p. 107

language model

A language model is the program behind ChatGPT or Claude: it was trained on enormous amounts of text to predict the next word. 8.1 p. 151

latency

Latency is the length of the drive: how long a message takes to make the round trip. 5.1 p. 113

Law of Demeter

The Law of Demeter says "only talk to your immediate neighbors": an object only calls its own methods, those of the objects handed to it and those of the objects it keeps inside. 4.3 p. 95

lead time

Lead time: time from a commit to production. 6.5 p. 133

least privilege

Finally least privilege: each program and each account has only the rights it needs. 7.6 p. 147

library

A library is a cookbook written by others: a collection of functions that a program adds so it can use them without rewriting them. 2.4 p. 51

linked list

A linked list works like a treasure hunt: each element tells you where to find the next. 1.3 p. 21

linter

The linter rereads the code without running it and flags what a machine can see: a variable never read, a name that breaks the project's convention, a == where === was needed (2.3). 3.1 p. 71

list

A list stores values one after another, a dictionary finds them by a name. 2.2 p. 39

Locality

Locality is the tendency of a program to stay close to what it has just read. 1.2 p. 15

loop

We repeat with a loop: a piece of program that the chip redoes once per element to process. 2.1 p. 33

magic number

A magic number is a number written as is in the code, with nothing to say what it stands for. 3.1 p. 71

mantissa

In binary, the machine stores the digits on one side, called the mantissa, and on the other the exponent, the power of 2 to multiply them by. 1.1 p. 5

mass assignment

Mass assignment slips an extra field into the request, which the server saves without a second thought. 7.6 p. 147

merge

When the work is ready, a merge glues the two histories back together: Git creates a commit with two parents, one per branch. 6.3 p. 129

method

The object's actions are called methods, functions kept inside the object: applyDiscount(). 2.2 p. 39

mock

A fake appliance plugged into the socket is called a mock (a fake that answers what it is told to answer and also records what it is asked). 6.1 p. 124

Model-Driven Design

Eric Evans's Model-Driven Design asks that the model, meaning the description of the business with its objects and rules, be written directly into the code. 4.5 p. 104

MTTR

MTTR (Mean Time To Recovery): time to restore service after an outage, from the start of the incident back to normal. 6.5 p. 133

N+1

The N+1 is a loop that fires one query per turn instead of a single join, and ORMs (object-relational mappers), the tools that write SQL on your behalf, produce it without warning. 4.6 p. 107

Nominal typing

Nominal typing, by name (Java, PHP), does the opposite: the object must declare in writing implements Point, "I am a Point", to be accepted. 2.3 p. 44

normalization

Normalization is the DRY of section 3.5 applied to data: each fact is written once, the customer's phone number in the customer's row, never in the orders. 4.6 p. 107

NoSQL

A document database (MongoDB is the well-known example), one of the families called NoSQL ("not only SQL"), puts each order in a folder: one block of JSON that holds everything, the customer, the lines, the address. 6.6 p. 136

O(1)

O(1): constant time, whatever the size (the fastest). 1.3 p. 21

O(log n)

O(log n): logarithmic time, climbing ever more slowly: you halve the problem at each step. 1.3 p. 21

O(n log n)

An efficient sort takes about n × log n steps: that is the O(n log n) family, between the straight line of O(n) and the blow-up of O(n²). 1.3 p. 21

O(n)

O(n): linear time, twice the data means twice the work. 1.3 p. 21

O(n²)

O(n²): quadratic time, twice the data means four times the work (the worst here). 1.3 p. 21

object

An object is a bundle that gathers, under a single name, everything that describes a thing: the information it carries and the actions you can ask of it. 2.2 p. 39

object-oriented programming (OOP)

Object-oriented programming (OOP) bundles data and the actions you can ask of it into a single object, like the checkout item (2.2, then the whole of chapter 4). 2.7 p. 67

observability

Observability goes further: it is being able to say what is happening inside a system from the outside, by reading what it writes about itself, without opening it or redeploying it. 6.5 p. 133

observer

Observer: notify a list of subscribers when a piece of data changes. 4.2 p. 91

operating system

The operating system (Windows, Linux or macOS) is the program that manages the machine, a bit like a head chef. 2.6 p. 62

output

The output of a program is everything it sends outside: text on the screen, a file written, a message over the network. 2.1 p. 33

packet

First, the network never carries a message in one piece. It cuts it into packets: small independent chunks, like the pages of a book mailed in separate envelopes. 5.2 p. 114

paradigm

A paradigm is a way of thinking about and writing a program. 2.7 p. 67

parallelism

Parallelism is several tasks moving at the same instant, one per core, like two cooks side by side. 2.6 p. 62

platform team

Platform: provides internal tooling so the others ship without waiting. 7.4 p. 144

pointer

An address stored in a variable is called a reference. Some languages, such as C, say a pointer. 2.2 p. 39

poka-yoke

A shape that makes the wrong move impossible is called a poka-yoke (Japanese for "mistake-proofing"). 3.4 p. 77

prepared queries

Against injection, prepared queries: the input travels separately from the query and can never become code again. 7.6 p. 147

procedural programming

Procedural programming writes the program as a sequence of instructions followed in order, gathered into functions. 2.7 p. 67

process

For each program you launch, the operating system creates a process: the running program together with its own area of memory, shared by all its threads. 2.6 p. 62

production

Between code that "works on my machine" and a service that holds in production, the machine open to the public where thousands of people actually use it, lies a chasm. ch. 6 p. 123

profiler

The profiler is a program that watches yours while it runs and says where the time goes, by function (a named chunk of code) or by line. 1.4 p. 26

program to an interface, not an implementation

Above all 23 patterns, the GoF sets two founding principles: composition over inheritance (4.1), and program to an interface, not an implementation. 4.2 p. 91

programming language

What remains is choosing the words in which you express your intent: the programming language. ch. 1 p. 4

promise

A promise is a value that is not there yet but will arrive later: a function like get returns one, instead of the value itself. 2.6 p. 62

prompt

The principle: fetch the relevant documents from an external base and paste them into the prompt (the message you send the model), so it leans on supplied facts, not its fuzzy memory. 8.2 p. 153

proof of receipt

The receiver replies with a proof of receipt saying what actually arrived. 5.3 p. 116

protocol

In the middle of the network, every machine shares one single protocol, a conversation rule they all agree on: IP (Internet Protocol), which gives each machine an address and moves packets toward it. 5.2 p. 114

Pseudo-code

Pseudo-code describes a calculation in sentences close to plain English: you then rewrite it in the language of your choice, and that is the good habit, think first, translate after. 2.1 p. 33

pure function

A pure function does only one thing: compute its result from its arguments, with no side effect and without reading anything hidden. 2.7 p. 67

QUIC

HTTP/3 goes further still: it drops TCP for a new transport, QUIC (pronounced "quick"), because one single TCP connection has a flaw. 5.5 p. 120

race condition

Two threads of execution inside the same program (two threads, the cooks of section 2.6, or two goroutines, Go's lightweight threads) reading and writing the same variable replay write skew in miniature: each acts on a world already stale, and that is called a race condition. 6.7 p. 138

RAG

The go-to pattern is called RAG (Retrieval-Augmented Generation: generate while leaning on retrieved documents). 8.2 p. 153

RAM

Everything else lives further away, in layers: the cache (a fast reserve glued to the chip, keeping a copy of recent data), then RAM (the working memory, the memory sticks in your machine), then disk. 1.2 p. 15

recursion

If a function is a recipe, recursion is a recipe that quotes itself. 2.5 p. 58

register

The registers are a few slots inside the CPU, the closest to it: the only place it actually computes. 1.2 p. 15

replication

When that is no longer enough, you copy: replication duplicates the database across several machines, and each one serves its share of the reads. 6.6 p. 136

result

A function is a recipe written once and for all: you hand it ingredients, its arguments, it hands back a dish, its result. ch. 2 p. 32

router

Each packet carries the destination address and is forwarded hop by hop by routers, the relays that steer it, the first being the modem-router your provider installed at home. 5.2 p. 114

rule of three

The rule of three, which Fowler owes to Don Roberts, comes in three steps: the first time you just write it, the second time you copy it and wince, the third time you refactor. 3.6 p. 83

saga

With no global undo, you write the undo by hand: that is the saga. 6.7 p. 138

schema

The list of tables, of their columns and of each column's type forms the schema: the shape of the data, decided before the first row, which the engine enforces on every write. 4.6 p. 107

scope

A variable's scope is the part of the program where its name is visible. 2.1 p. 33

seam

Michael Feathers calls this a seam, a place where you can change behavior without modifying the function itself. 6.1 p. 124

secret

A secret, a password or an API key, is a value that never lives in the code: it arrives through the configuration, an environment variable or a file Git ignores. 6.3 p. 129

shallow

A function that hides almost nothing is called shallow: applyVAT(price), if it boils down to price * 1.2, simplifies nothing. 3.2 p. 73

sharding

Sharding spreads the data in slices (customers A-M here, N-Z there), and each machine only takes its share. 6.6 p. 136

side effect

A side effect is anything a function changes outside its result: a text shown on screen, a global variable, a received list it alters in place. 2.7 p. 67

signature

The first line of a function, its name followed by what it expects, is its signature. 2.3 p. 44

slow start

The connection does not know the capacity of the road yet, so it begins small (about ten packets, roughly 14 KB) and doubles its window on every round trip: this is slow start (slow only compared with sending everything at once). 5.4 p. 118

SOLID

Dependency inversion carries a letter: it is the D of SOLID, five design principles gathered under one acronym. 4.3 p. 95

source code

The source code is the text of the program, as the human wrote it. 1.4 p. 26

SQL

The standard language for talking to the engine is SQL (Structured Query Language), and an SQL query, meaning a question put to the database, says what you want, not how to get it. 4.6 p. 107

SQL injection

SQL injection shows the whole mechanism. 7.6 p. 147

stack overflow

Hence a very real limit: a recursive function that calls itself too many times before reaching the base case overflows the stack (a stack overflow). 2.5 p. 58

stack trace

A stack trace is the message the language prints when a program stops on an error: the list of calls in progress, then the cause. 2.1 p. 33

standard library

A standard library is the collection of ready-made functions that ships with the language: random is part of it, and sum works even without an import. 2.4 p. 51

state machine

The status is the enumeration of 2.3, and cancel() is only allowed from certain of its states: the order is a state machine, an object that moves from one state to another only through transitions written once, in the code. 4.5 p. 104

static typing

Static typing checks the types before the program runs, so the mistake shows up before the first launch: Java, or TypeScript, explained below. 2.3 p. 44

strategy

Strategy: swap one way of doing a task for another while the program runs, exactly what we just did by replacing the duck's flight. 4.2 p. 91

stream-aligned team

Stream-aligned: one team = one product, shipped end to end (the default case). 7.4 p. 144

string

The string: text, like "hello", a run of letters that are each a number (the ASCII and UTF-8 of 1.1). 2.3 p. 44

structural typing

Sets also explain structural typing: an object enters a type by its shape, not by the label it wears. 2.3 p. 44

surface

The deep function hides a lot of work behind a tiny surface, meaning what the caller must know to use it (here one name and one argument): one line on the menu, everything else in the kitchen. 3.2 p. 73

surgical team

Brooks dreamed in 1975 of a surgical team: one brain holding the scalpel (designing, deciding), surrounded by specialized assistants. 8.4 p. 157

table

The relational model, laid down in 1970 by Edgar Codd, stores each kind of thing in a table: a grid with fixed columns, one row per thing, a Customers table, an Orders table. 4.6 p. 107

TCP

Something has to bridge the gap between what IP promises and what an application actually needs. That is the job of TCP (Transmission Control Protocol), running on the two endpoint machines, right above IP. 5.3 p. 116

TDD

The deepest effect of TDD (Test-Driven Development) is not catching bugs, it is emergent design: writing the test first, you design the API, that front counter from section 4.5, from the point of view of whoever comes to drop off a request, not of whoever staffs the desk. 6.1 p. 124

teamicide

You can kill it fast, and they have a word for it: teamicide, anything that breaks a team up or keeps it from jelling. 7.5 p. 146

technical debt

Shortcuts taken here and there pile up into technical debt: each shortcut makes the next change more expensive, like interest on a loan (the image dates from 1992, from Ward Cunningham). ch. 3 p. 70

terminal

A terminal is a window where you type your requests to the machine, one line at a time. 1.4 p. 26

thread

A thread (thread of execution) is a sequence of instructions that the chip follows one after the other. 2.6 p. 62

three-way handshake

Before a single useful byte, the two machines exchange a three-way handshake: "I want to talk", "got it, me too", "got it" (SYN for "synchronize", ACK for "acknowledge": SYN, SYN-ACK, ACK in the diagram). 5.3 p. 116

throw an exception

When a function cannot do its job, many languages make it throw an exception: it stops right there and hands an error signal, the exception, to its caller. 3.4 p. 77

timeout

Set a delay beyond which you stop waiting: that is the timeout. 5.2 p. 114

TLS

TLS (Transport Layer Security): encryption, the padlock of https (the "s" stands for "secure"). 5.5 p. 120

token

It is measured in tokens, pieces of words (a common word is one or two). 8.2 p. 153

tracking number

Every byte sent carries a tracking number. 5.3 p. 116

transaction

The first two promises rest on the transaction: a group of changes that succeeds in full or not at all. 4.6 p. 107

tree

A tree is the list of files in a folder at a given moment. 6.3 p. 129

type

Each kind of data is a type. 2.3 p. 44

Ubiquitous Language

Evans asks the team and the expert to agree on the exact words of the business and to use them everywhere, in meetings as in code: that is the Ubiquitous Language ("language present everywhere"), one shared language. 4.5 p. 104

union

The union (A | B) gathers all the values of both types: red, green, blue. 2.3 p. 44

unit test

A test that isolates a function behind fakes is a unit test. 6.1 p. 124

UTF-8

UTF-8 is the convention that stores each character on 1 to 4 bytes: 'A' keeps its single byte, 'é' takes two, '🎉' four. 1.1 p. 5

value object

A value object is nothing but its contents: two €10 notes are interchangeable, two identical addresses are the same address. 4.5 p. 104

variable

A variable is a name stuck on a slot in memory, like a label on a locker. ch. 2 p. 32

WCAG

All of this is framed by a global standard, the WCAG (Web Content Accessibility Guidelines): four principles (Perceivable, Operable, Understandable, Robust) and three levels (A, AA, AAA). 7.3 p. 143

window

TCP sends a whole convoy of packets, trucks on our highway, before hearing back about the first one, rather than one truck, silence, then the next. The number of packets allowed on the road without news is called the window. 5.3 p. 116

write skew

The nastiest one, write skew: two transactions, each perfectly valid, that break a rule together. 6.7 p. 138

XSS

XSS (Cross-Site Scripting) injects not SQL but a bit of script, slipped into the page through an HTML tag, which will run in other visitors' browsers. 7.6 p. 147

YAGNI

YAGNI (You Aren't Gonna Need It) is the counterweight to DRY and ETC: the rule, from Kent Beck's team, that advises against building ahead what nobody has asked for yet. 3.6 p. 83

Colophon

Text set in Literata, headings in Fraunces, captions in Lato, code in JetBrains Mono.

This edition is taken from the online version, which remains the reference and keeps improving with each proofreading: web-developpeur.com/en/library/le-grand-livre/