

BIBLIOTHÈQUE

Le grand livre du développeur

Tout le métier en un seul fil

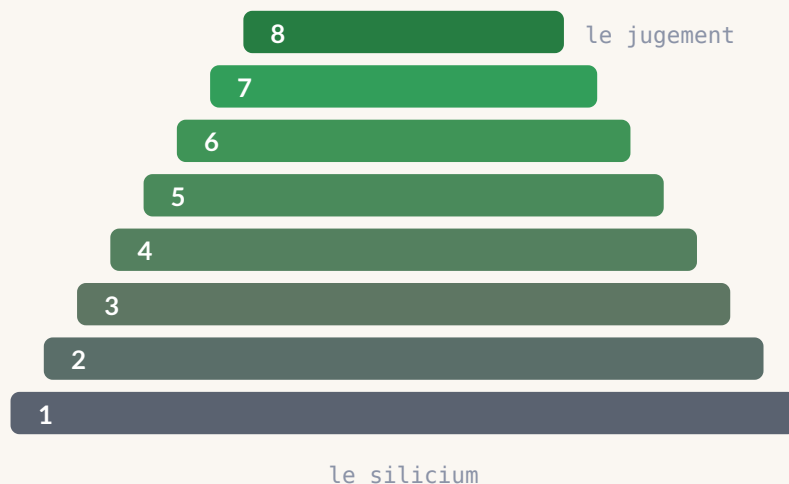
Des bits et du Big O jusqu'au code propre, l'architecture, le réseau, la livraison, les humains et l'ère de l'IA : la moelle de dizaines de livres tech, recousue en un seul fil.

par

HUGONNOT Odilon

Édition du 10 octobre 2026

web-developpeur.com



Sommaire

1	La machine ne comprend que des nombres	4
1.1	Tout est nombre	5
1.2	La mémoire est une pyramide	16
1.3	Le coût a une unité: le Big O	21
1.4	Penser la machine, écrire haut niveau	26
2	Le langage, outil de pensée	30
2.1	Les trois gestes: enchaîner, répéter, choisir	31
2.2	Valeurs et références: ce que la variable contient vraiment	35
2.3	Les types sont des ensembles de valeurs	39
2.4	Les fonctions sont des valeurs	44
2.5	La récursivité: résoudre un problème avec une version plus petite de lui-même	49
2.6	La permissivité a un prix	53
2.7	Nommer un concept change ce qu'on peut penser	55
2.8	Trois façons d'écrire un programme	59
3	Écrire pour les humains	62
3.1	Le nom révèle l'intention	63
3.2	La fonction profonde, pas seulement courte	65
3.3	Le commentaire dit le pourquoi, pas le quoi	67
3.4	Faire disparaître l'erreur	69
3.5	Ne te répète pas, et facilite le changement	73
3.6	Refactorer, c'est changer la forme sans changer le comportement	75
4	Donner une forme au système	79
4.1	Composer plutôt qu'hériter	80
4.2	Un pattern répond à un besoin, pas à un but	84
4.3	Dépendre d'abstractions, inverser les dépendances	87
4.4	Pas de « best practice », seulement le moins mauvais des compromis	94
4.5	L'intégrité conceptuelle: un seul esprit	96
4.6	Ranger pour retrouver: la base de données	99
5	Deux machines se parlent	104
5.1	La distance ne se compresse pas	105
5.2	Le sablier: un contrat minuscule au milieu	106
5.3	TCP: fabriquer du fiable sur du peu fiable	108
5.4	La congestion: partager sans chef	110
5.5	Le web paie le péage	112
6	Prouver, livrer, tenir la charge	115
6.1	Le test d'abord	116
6.2	Déboguer, c'est refuser de croire	119
6.3	Versionner, c'est un système adressable par contenu	122
6.4	Le flux bat l'effort	124
6.5	Déployer souvent fait moins peur que déployer rarement	126
6.6	La lecture tient en copiant, l'écriture en découpant	128
6.7	À l'échelle, les garanties se paient	131
7	Le logiciel est pour des humains	134
7.1	Ne me fais pas réfléchir	134
7.2	Une API se conçoit pour celui qui l'appelle	136
7.3	Accessible par construction	137
7.4	L'organisation se reflète dans le code	138
7.5	Ajouter des gens à un projet en retard le retarde	139
7.6	Penser comme l'attaquant pour défendre	140
8	Coder à l'ère de l'IA	144
8.1	L'IA génère du probable, pas du vrai	145
8.2	Tout se joue dans le contexte	147
8.3	L'IA se pilote en boucle courte	149
8.4	L'équipe chirurgicale, enfin réalisable	151
8.5	Le jugement humain est le dernier maillon	152
	Lexique	157

Avant-propos

On apprend à coder par morceaux : un langage cette année, une méthode le mois suivant, une astuce un jeudi soir. Le savoir s'empile, mais rien ne le relie, et on finit par appliquer des règles sans savoir d'où elles viennent.

Ce livre n'apprend pas à coder : il apprend à coder **bien**, dans n'importe quel langage et à n'importe quel niveau. Il enfile les morceaux comme des perles sur **un seul fil**, du courant dans le silicium jusqu'au jugement humain qu'aucune IA ne remplace.

Chaque chapitre est la **conséquence du précédent**, et chaque idée n'est racontée qu'une fois, à sa place. Ce n'est pas un recueil de résumés : le meilleur de dizaines de livres y est recousu, et leurs titres sont cités au bas de chaque section. Lisez de haut en bas, et le métier s'assemble.

La promesse est simple. Que vous débutiez ou que vous codiez depuis des années, vous repartez en sachant non seulement quoi faire, mais **pourquoi** on le fait, **où** ça casse et **quand** s'en abstenir.

1 La machine ne comprend que des nombres

le coût

La machine ne pense pas : des nombres entrent, des nombres sortent. Un texte, une image, une condition, tout devient des nombres, et chaque calcul a un coût. La même tâche peut être instantanée ou ramer pendant des secondes, selon la façon dont vous l'écrivez. Ce chapitre vous apprend à voir ce coût avant de le payer. Quatre marches : la plus petite case de mémoire, les étages où vivent les données, l'unité qui mesure le coût, et le traducteur (le logiciel qui convertit votre code en instructions pour la puce de calcul) qui s'occupe du reste.

Si vous codez déjà, ne sautez pas ce chapitre : vous y trouverez pourquoi un tableau trié se parcourt plus vite que le même tableau en désordre, pourquoi le `0(1)` ment parfois, et pourquoi « café » pèse cinq octets.

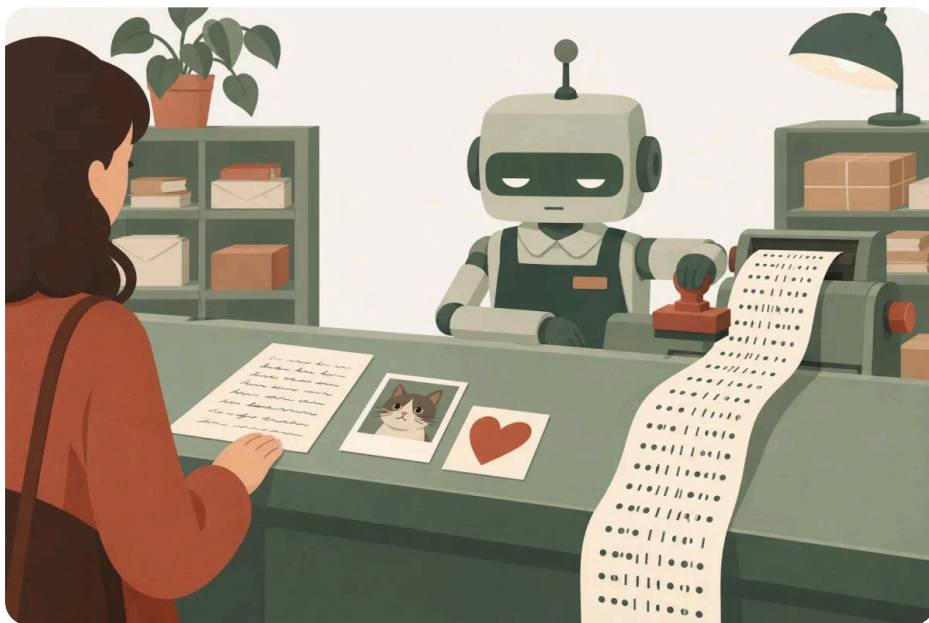


Figure 1 : Un poème, une photo, un cœur dessiné : le guichet ne rend que des nombres. La machine ne lit rien d'autre, et chaque calcul se paie.

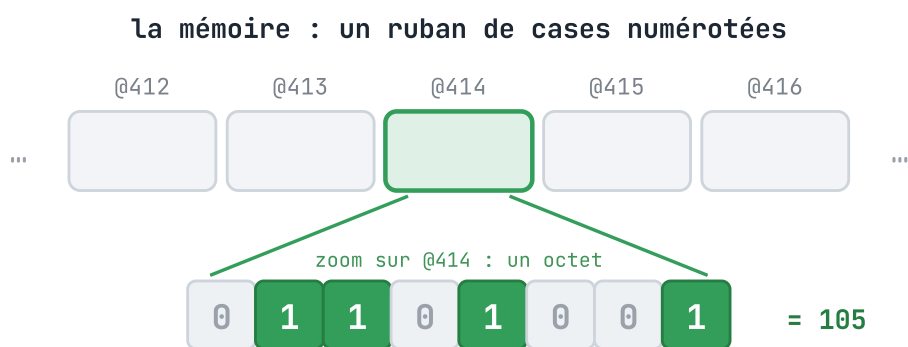
1.1 Tout est nombre

Petit avertissement : c'est le passage le plus proche de la machine, le plus « bas niveau » du livre, et c'est voulu. Ne cherchez pas à retenir les 0 et les 1 par cœur, visez l'intuition : voir comment un ordinateur range et fait travailler l'information, du plus petit interrupteur jusqu'au programme. Cette vue d'ensemble vous servira à comprendre comment marche le code qu'on écrit par-dessus. Tout le reste du livre repose sur ce socle.

Un **bit**, c'est un interrupteur : éteint, il vaut 0, allumé, il vaut 1. C'est la seule chose que la machine sait physiquement retenir. Compter avec deux chiffres seulement, c'est le binaire. On groupe les interrupteurs par huit, et un groupe de huit s'appelle un **octet**.

Un cadenas à 4 molettes de 10 chiffres offre $10 \times 10 \times 10 \times 10 = 10\,000$ combinaisons, de 0000 à 9999. On part de zéro : la dernière est 9999, pas 10 000. Un octet est le même cadenas avec 8 molettes à deux positions, c'est-à-dire 8 interrupteurs : $2 \times 2 \times \dots \times 2 = 256$ combinaisons, de 0 à 255.

Mis bout à bout, ces octets forment la mémoire : une immense rangée de cases, chacune repérée par un numéro, son **adresse**.



8 interrupteurs par case, un numéro par case : on va droit à chacune

Figure 2 : Toute la mémoire tient dans cette image. Ce numéro de case, c'est l'adresse : le chapitre 2 s'en servira pour retrouver une donnée. Pourquoi 01101001 vaut 105 : voir « Lire un nombre binaire », plus bas.

Le choix du binaire

Pourquoi un bit n'a-t-il que deux valeurs, 0 ou 1 ? Ça paraît primitif. Parce que **le monde physique est bruyé**. Le bruit, ce sont les petites variations imprévisibles qui secouent tout signal : le niveau du signal dans un fil dérive avec la chaleur, les perturbations des appareils voisins, l'âge des composants. Distinguer dix niveaux (un par chiffre décimal) demanderait une précision intenable. Deux états, eux, sont sans ambiguïté, « éteint » ou « allumé ». Il faut un bruit énorme pour confondre un 0 et un 1.

Un DVD l'illustre : sa piste n'est qu'une suite de microscopiques creux et de surfaces plates, et le laser n'a qu'à dire si la lumière revient droit ou se disperse. Un code QR fait de même avec des carrés noirs ou blancs que votre téléphone lit, même sous une lumière médiocre.

Pour écrire en décimal avec le même procédé, il faudrait distinguer dix profondeurs de creux sur le DVD : une rayure ou un grain de poussière suffirait à confondre un 4 et un 5. Sur le QR code, il faudrait distinguer dix nuances de gris : une simple ombre suffirait à confondre deux voisines.

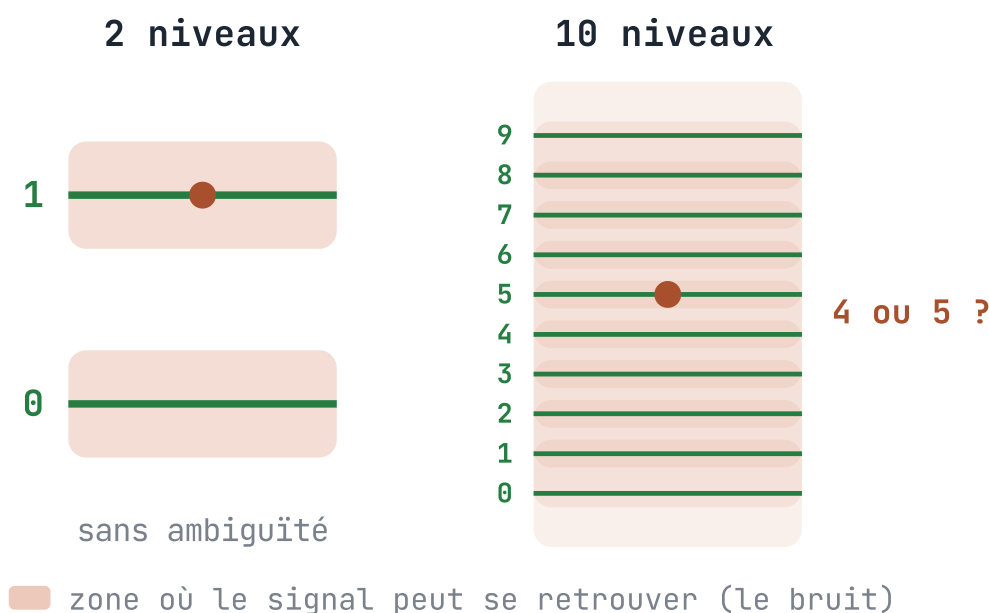


Figure 3 : Le bruit fait trembler le signal (le point) : il peut se retrouver n'importe où dans la zone rose. À deux niveaux, les zones restent séparées. À dix, elles se chevauchent.

Le bit est donc le plus petit choix possible : oui ou non. C'est aussi celui qui résiste le mieux au bruit, et il se loge dans l'objet le plus simple qui soit, l'interrupteur. Avec seulement deux états, on sait pourtant tout faire : vrai ou faux pour décider, des nombres écrits en 0 et en 1 pour compter. Loin d'être primitif, c'est un choix d'ingénieur.

On a d'ailleurs essayé d'autres choses. L'ENIAC, l'un des tout premiers ordinateurs (achevé en 1945), comptait en décimal, avec dix bascules, des paires de tubes électroniques, par chiffre. Le Setun, un ordinateur soviétique (achevé en 1958 puis fabriqué en série jusqu'en 1965), comptait en ternaire, avec trois valeurs. Les deux marchaient. Le binaire s'est pourtant imposé : deux états, c'est ce qu'il y a de plus simple à fabriquer et à fiabiliser, en milliards d'exemplaires.

Il reste des exceptions. Les SSD récents, ces disques sans pièce mobile, font le pari inverse : ils distinguent 4, 8 ou même 16 niveaux dans une même cellule, donc 2, 3 ou 4 bits au lieu d'un, pour stocker plus. Ils paient cette densité par une usure plus rapide. C'est la même règle : plus de niveaux, moins de marge pour le bruit.

Lire un nombre binaire

Reste à savoir comment une suite de 0 et de 1 devient un nombre. Tout se joue sur la **position**. Partons de la base 10, celle de nos dix chiffres. 234, ce n'est pas « 2, 3, 4 » collés : chaque colonne porte un poids, les unités, les dizaines, les centaines.

Le binaire suit exactement la même règle, avec deux chiffres possibles au lieu de dix : 0 et 1. En partant de la droite, les colonnes ne valent donc plus 1, 10, 100, mais 1, 2, 4, 8, 16... : chacune double la précédente, ce sont les **puissances de 2**.

Lire un nombre binaire revient alors à additionner les colonnes qui portent un 1, en sautant celles qui portent un 0. Dans `101`, la colonne des 4 et celle des 1 portent un 1, celle des 2 porte un 0 : $4 + 0 + 1 = 5$. Avec huit colonnes toutes à 1, on obtient $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$: c'est le maximum d'un octet, comme 9999 pour le cadenas.

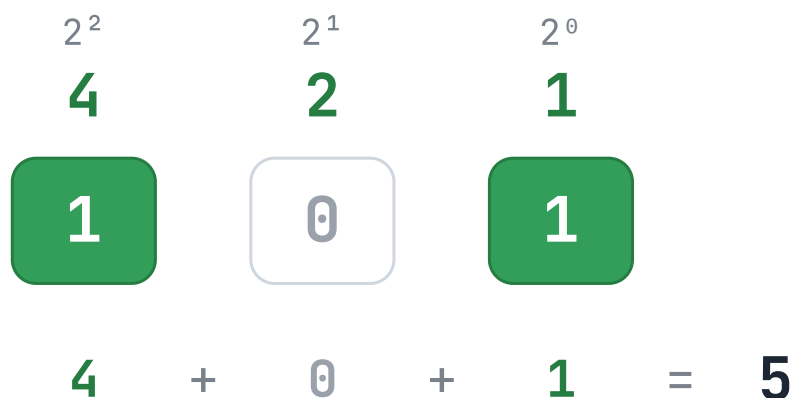


Figure 4 : Chaque colonne vaut une puissance de 2. On additionne celles dont le bit est à 1.

Tout devient des 0 et des 1

Maintenant qu'on sait lire un nombre binaire, voyons ce que la machine y range. Texte, image, son : tout se ramène à des nombres, donc à des 0 et des 1. Une lettre tient dans un octet : le `'A'` est le nombre 65, par une convention que toutes les machines partagent, l'ASCII (code américain standard pour l'échange d'information). Un mot en occupe quelques-uns : «Hi!» en prend trois.

majuscules									
A 65	B 66	C 67	D 68	E 69	...	H 72	...	Z 90	
minuscules									
a 97	b 98	c 99	d 100	e 101	...	i 105	...	z 122	

Figure 5 : Le dictionnaire de l'ASCII : une lettre, un numéro. Le H et le i de «Hi!» sont en vert.

		poids	128	64	32	16	8	4	2	1	adr.
H	72	0	1	0	0	1	0	0	0	0	@0
i	105	0	1	1	0	1	0	0	1	0	@1
!	33	0	0	1	0	0	0	0	0	1	@2

1 caractère = 1 octet = 8 bits · ex. 72 = 64 + 8

Figure 6 : Le mot « Hi! » en mémoire : une lettre par octet. On lit le nombre en additionnant les colonnes dont le bit vaut 1.

Un octet par lettre, c'était vrai pour l'anglais : un octet n'offre que 256 valeurs, assez pour son alphabet. Mais le monde compte plus de 150 000 caractères, des accents aux emojis. L'**UTF-8** est la convention qui range chaque caractère sur 1 à 4 octets : 'A' garde son octet, 'é' en prend deux, '🎉' quatre. D'où un piège bien réel : la « longueur » d'un texte dépend de ce qu'on compte, des octets ou des caractères. Le mot **café** compte 4 caractères, mais 5 octets. Couper **café** après le quatrième octet, c'est laisser la moitié d'un **é**.

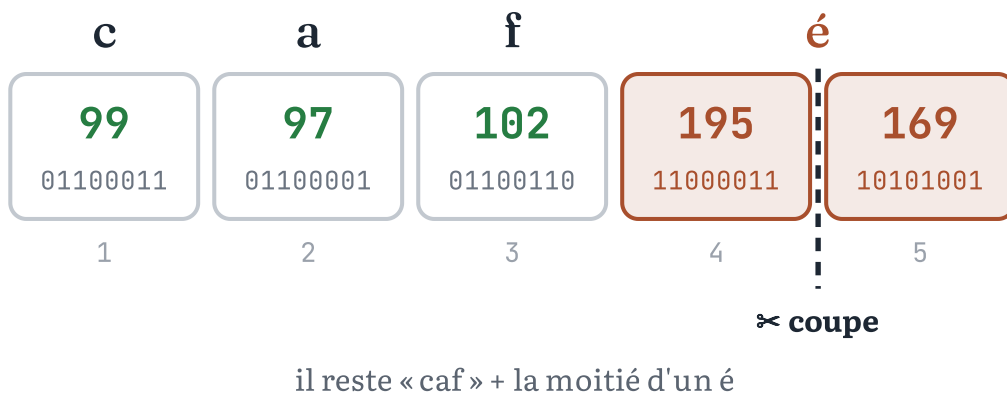


Figure 7 : « café » en octets : les lettres en vert tiennent dans un octet, le é (en terracotta) en prend deux. Le trait en pointillé marque la coupe.

Le même piège existe à la lecture. Si un programme lit les deux octets du **é** (195 et 169) avec une ancienne convention, un octet par lettre, il y voit deux symboles, et « café » devient « cafÃ© ». C'est la cause des signes bizarres qui remplacent les accents dans un fichier ouvert avec le mauvais format.

Passons à l'image : c'est une mosaïque de petits points, les pixels. Une photo en compte des millions. Chacun a sa couleur, qu'on recompose en dosant du rouge, du vert et du bleu, trois couleurs qui suffisent à fabriquer presque toutes les autres. Le jaune, par exemple, c'est du rouge et du vert à fond, sans bleu. Il faut donc trois nombres par pixel, un octet chacun, de 0 à 255.

Le cas le plus simple est une image en noir et blanc. Prenez un cœur dessiné sur une grille de 8 colonnes et 7 lignes : chaque case, un pixel, est allumée (1) ou éteinte (0), et chaque ligne est un octet. La première, `01100110`, vaut $64 + 32 + 4 + 2 = 102$. Sept nombres suffisent pour dessiner le cœur.



1 ligne = 1 octet · 7 octets pour tout le dessin

Figure 8 : Un cœur en noir et blanc : une case allumée vaut 1, éteinte vaut 0. Chaque ligne est un octet, donc un nombre.

Huit chiffres par octet, c'est long à lire. L'**hexadécimal** est une écriture des nombres en base 16, où un chiffre, de 0 à 9 puis de a à f, remplace quatre bits : deux chiffres par octet.

Exemple, la couleur terracotta des figures de ce livre, `#a84f2d`, notation des couleurs du web, se lit ainsi : trois octets, `a8`, `4f`, `2d`, soit 168 de rouge, 79 de vert, 45 de bleu, les trois nombres de chaque pixel en couleur, vus plus haut.

Même règle de colonnes qu'en binaire, avec 16 à la place de 2 : `a8`, c'est $10 \times 16 + 8$ (le `a` vaut 10, le `8` vaut 15).

Le son se range lui aussi en nombres. Un micro mesure la vibration de l'air, et on note sa valeur 44 100 fois par seconde : une chanson devient une longue liste de nombres. Une vidéo n'est que des images et du son qui défilent. Photo, musique, film : rien de magique, juste énormément de petits interrupteurs, allumés ou éteints.

Une date aussi. L'ordinateur la range comme un nombre de secondes écoulées depuis le 1er janvier 1970 à minuit, heure de Londres : `1780963200` se lit «9 juin 2026». Le nombre ne dit pas où l'on est : la même date se lit 2 h du matin à Paris, ou 20 h la veille à New York, selon le fuseau horaire qui la lit. D'où une règle d'or : on range une date en UTC, l'heure de référence mondiale, et on ne la convertit en heure locale qu'au moment de l'afficher.

Les entiers et leurs limites

Revenons aux nombres eux-mêmes, ceux des colonnes en puissances de 2. Un entier, c'est-à-dire un nombre sans virgule, est une somme exacte de puissances de 2 : il se range sans erreur, tant que ses octets suffisent. Un octet s'arrête à 255 : pour aller plus loin, on en réunit plusieurs. Deux octets montent jusqu'à 65 535, quatre jusqu'à plus de 4 milliards (sans négatifs), et chaque octet de plus multiplie la capacité par 256.

Pour ranger aussi les négatifs, la machine réserve la moitié des motifs : sur un octet, `00000000` à `01111111` valent 0 à 127, et `10000000` à `11111111` valent -128 à -1. Le motif `11111111` vaut donc 255 ou -1 selon la lecture qu'en fait le programme. Cette convention s'appelle le complément à deux.

Mais le nombre d'octets est choisi d'avance, et c'est là que ça casse. Comme le cadenas du début, qui repasse de 9999 à 0000, un octet plein repart du début : $255 + 1$ vaut 0 (avec des négatifs, $127 + 1$ donne -128). Un **dépassement d'entier** est un calcul dont le résultat ne tient plus dans les octets réservés : dans les langages à entiers de taille fixe, le nombre repart en général du bas de l'échelle, sans prévenir.

En C, Go ou Java, où un entier a une taille fixe, le piège est réel (en C, pour un entier signé, le résultat n'est même pas défini : tout peut arriver) : en 2014, le compteur de vues de YouTube, un entier de 32 bits, soit 4 octets, a buté sur son maximum, 2 147 483 647 ($2^{31} - 1$: la moitié des motifs de 4 octets sert aux négatifs), devant *Gangnam Style*. Python, lui, agrandit ses entiers à la demande : il n'a pas cette limite.

Les nombres à virgule

Après la virgule, la règle des colonnes continue. En base 10, on passe aux dixièmes ($1/10$), aux centièmes ($1/100$)... et `0.1` signifie « 1 dans la colonne des dixièmes ». En code, le point remplace la virgule. Ce n'est pas le point qui fait la valeur, **c'est la colonne**. En binaire, les colonnes deviennent des fractions : $1/2$, $1/4$, $1/8$, $1/16$... Pour se ranger sans erreur, un nombre à virgule doit s'écrire comme une somme de ces fractions : `0.75`, par exemple, c'est $1/2 + 1/4$, soit `0.11` en binaire.

`0.1`, lui, n'y arrive pas, et c'est l'origine d'un bug célèbre : dans presque tous les langages, `0.1 + 0.2` ne fait pas `0.3`. Une comparaison aide à comprendre. En base 10, $1/3$ s'écrit `0.333...` sans jamais finir. En binaire, c'est $1/10$ qui ne finit jamais : `0.0001100110011...` Une fraction ne tombe juste en binaire que si, une fois simplifiée, le dénominateur (le nombre du bas) est une puissance de 2, comme 2, 4 ou 8. Or 10 n'en est pas une : il cache un facteur 5, que multiplier 2 par lui-même ne produit jamais.

Comment ranger de tels nombres, alors ? En déplaçant la virgule. Restons en base 10 pour voir le procédé : `1234.5` peut s'écrire `1.2345 × 103`, et `0.00012345` s'écrit `1.2345 × 10-4`. Cette écriture a un nom, la notation scientifique : c'est celle de votre calculatrice. Les mêmes chiffres dans les deux cas, et un exposant qui dit de combien de rangs la virgule a bougé.

En binaire, la machine range d'un côté les chiffres, qu'on appelle la **mantisse**, et de l'autre l'**exposant**, la puissance de 2 par laquelle il faut les multiplier. C'est ça, la « virgule flottante » : la virgule n'a pas de place fixe, l'exposant la fait glisser. Un nombre rangé ainsi s'appelle un **flottant**. Avec le même nombre de chiffres, on code ainsi l'immense comme l'infime.



Figure 9 : Un nombre à virgule en mémoire : 64 interrupteurs répartis en trois rôles. Un pour le signe, onze pour l'exposant, cinquante-deux pour les chiffres. C'est là que la place manque et que le motif se coupe (ici, celui de `0.1`).

▼ Les bits exacts de `0.1`, pour les curieux (facultatif)

Pour `0.1`, lisons les trois morceaux. Le **signe** d'abord : 0, car `0.1` est positif (1 = négatif). La **mantisse** ensuite : on reprend `0.0001100110011...` et on glisse la virgule jusqu'au premier 1, ce qui donne `1.100110011... × 2-4`, dont les chiffres répètent **1001** sans fin. Un nombre ainsi normalisé commence toujours par «1.», donc ce 1 de tête n'est jamais stocké : un bit gratuit. L'**exposant** enfin, -4. Les 11 bits qui le stockent ne savent coder que des nombres positifs (0 à 2047), or -4 est négatif. La parade : on lui ajoute un décalage fixe de 1023, ce qui le rend toujours positif. La machine range donc $-4 + 1023 = 1019$, écrit en binaire `01111111011` (et resoustrait 1023 à la relecture pour retrouver -4).

Le piège des flottants

Revenons à `0.1`, dont le motif binaire ne finit jamais. La place réservée aux chiffres, au contraire, est **finie** : une cinquantaine de bits. On coupe donc le motif infini, et le dernier chiffre gardé est **arrondi**. Ce petit reste d'arrondi, ap-

pelons-le la **miette**: **0.1** stocké vaut un peu plus que 0.1. **0.2** subit le même sort, les miettes s'ajoutent, et le total rate **0.3** :

```
// (=== teste l'égalité) ce que la machine stocke vraiment (la « miette »)
// :
0.1      → 0.1000000000000000055...    // un peu trop grand
0.2      → 0.2000000000000000111...    // un peu trop grand
0.3      → 0.2999999999999999888...    // un peu trop petit

0.1 + 0.2 → 0.3000000000000000444...    // dépasse le 0.3 stocké
0.1 + 0.2 === 0.3    // false → 0.30000000000000004
0.1 + 0.3 === 0.4    // true... mais par chance !
0.1 + 0.7 === 0.8    // false, comme 0.1 + 0.2
```

Tout est là: **0.1** et **0.2** penchent un peu trop haut, **0.3** un peu trop bas. La somme des deux dépasse le 0.3 rangé en mémoire. La somme et le 0.3 rangé ne sont donc pas le même nombre : l'égalité est fausse.

Ce ratage n'est pas isolé. Les miettes tantôt s'ajoutent, tantôt se compensent : comparer deux flottants avec un test d'égalité est donc une **loterie**, comme le montrent les deux dernières lignes du code ci-dessus.

À retenir: un entier se range sans erreur tant qu'il tient dans ses octets. Un nombre à virgule, lui, n'est exact que s'il est une somme de $1/2$, $1/4$, $1/8$... assez courte pour tenir dans les chiffres du flottant, comme 0.75 ($1/2 + 1/4$). D'où deux réflexes.

Ne jamais comparer deux flottants avec un test d'égalité: on vérifie plutôt qu'ils sont assez proches, en proportion de leur taille (en Python, `math.isclose(a, b)`). Et un montant d'argent se range **en centimes**, des entiers: 19,99 € devient 1999. Un type décimal exact, comme `Decimal` en Python ou `BigDecimal` en Java, convient aussi. Le calcul redevient fiable, et toute une classe de bugs d'arrondi disparaît.

Le programme est un nombre aussi

Dernier exemple, le plus troublant: votre programme lui-même n'est fait que de nombres. Un programme est une liste d'**instructions**, de minuscules ordres que le processeur, la puce de calcul, sait exécuter: « additionne ces

deux nombres», « copie cette case ailleurs», « saute au numéro 40 si le résultat vaut zéro ».

Ces ordres sont rangés les uns derrière les autres, chacun à son numéro, comme les cases de la mémoire. La puce lit le premier, l'exécute, passe au suivant, et ainsi de suite. Un saut ne fait que lui dire « continue plutôt au numéro 40 ». Revenir en arrière, c'est répéter. Sauter en avant, c'est passer un morceau, par exemple quand le résultat vaut zéro.

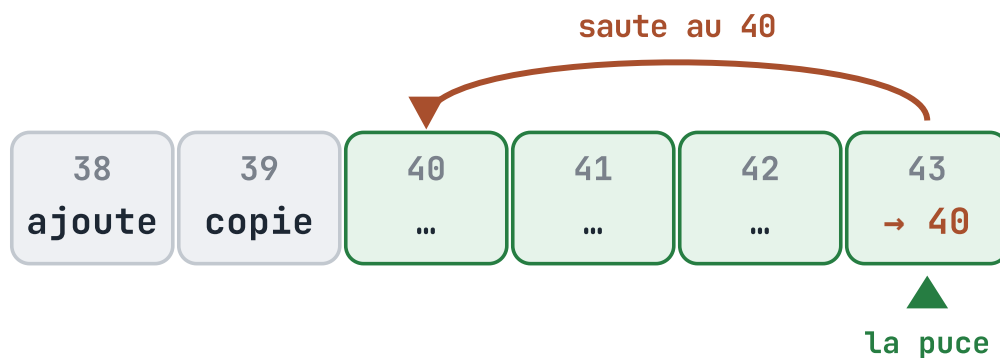


Figure 10 : La puce lit les ordres un par un, chacun à son numéro. Au 43, un saut la ramène au 40 : elle répète.

Les ordres agissent sur des **données**, c'est-à-dire des nombres comme un prix, une lettre ou un pixel. Or ordres et données sont tous deux des nombres, rangés dans la même mémoire : la machine ne les distingue pas. Tout dépend de la lecture : le 65 peut se lire comme la lettre « A », comme un prix ou comme un ordre. Cette ambiguïté a un coût : au 7.6, un attaquant en profitera pour glisser un ordre là où l'on attendait une donnée.

Les puces modernes se protègent de cette confusion : chaque zone de mémoire porte une marque, « ordres » ou « données » (c'est le bit NX, pour « non exécutable »), et la puce refuse d'exécuter une case marquée « données ». Mais ce verrou ne voit que les cases de mémoire. L'attaquant cache son ordre dans un texte (un nom saisi dans un formulaire, par exemple) que votre programme recopie tel quel au milieu d'une commande : aucune marque ne sépare alors la donnée de l'ordre. C'est la même confusion, un cran plus haut.

↳ [Write Great Code, vol. 1](#) · [Eloquent JavaScript](#)

1.2 La mémoire est une pyramide

Jusqu'ici, la mémoire n'était qu'une rangée d'octets uniforme. En vrai, elle a des **étages**, empilés comme une pyramide : petit et rapide en haut, immense et lent en bas. Pourquoi ? Parce qu'aucune mémoire n'est à la fois grande, rapide et bon marché : la rapide coûte cher et reste petite. Et le **processeur** (la puce qui exécute vos instructions une par une) ne calcule vite que sur ce qui est tout près de lui. Le plus près, ce sont ses **registres**, quelques cases au cœur de la puce : les seules où il calcule.

Tout le reste vit plus loin, par étages : le **cache** (une réserve rapide collée à la puce, qui garde une copie des données récentes), puis la **RAM** (la mémoire vive, les barrettes de votre machine), puis le **disque**. Le cache lui-même se découpe en trois paliers, L1, L2 et L3, du plus proche de la puce au plus lointain. Pour être utilisée, une donnée doit donc d'abord monter jusqu'à un registre, étage après étage. Chaque étage qu'on descend est plus grand, mais environ dix fois plus lent : le voyage dure d'autant plus longtemps.

Entre la RAM et le disque dur, en revanche, ce n'est plus une marche de plus, c'est un gouffre : $\times 100\,000$. Un SSD en comble une grande partie, mais reste environ mille fois plus lent que la RAM.

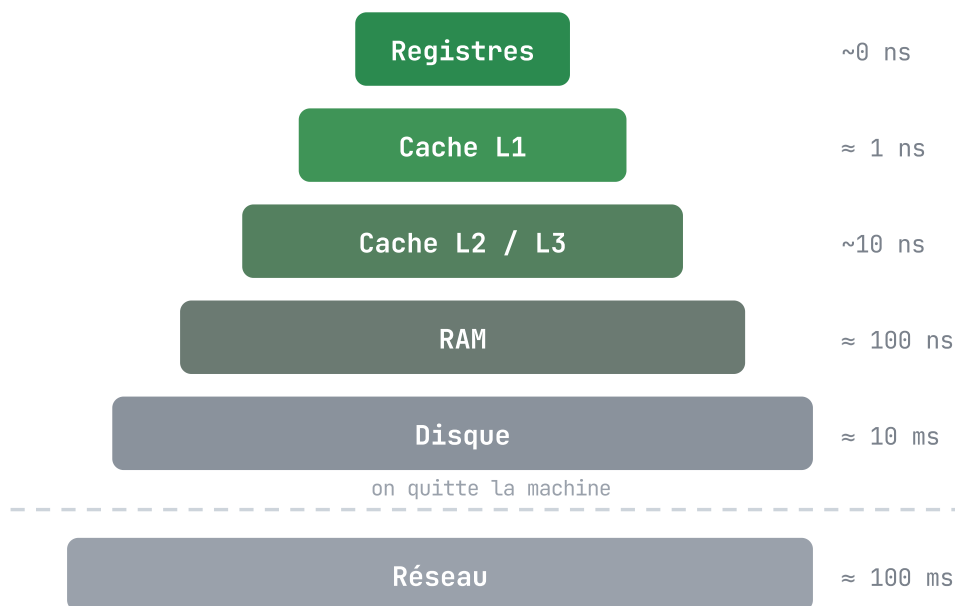


Figure 11 : La pyramide mémoire : en haut, petit et quasi instantané. En bas, immense et lent. Sous le pointillé, on a quitté la machine.

La pyramide ne s'arrête d'ailleurs pas au bord de la machine : en dessous du disque, il y a le réseau, encore dix fois plus lent quand la donnée traverse un océan. Cet étage-là a son chapitre ([chapitre 5](#)).



Figure 12 : La machine vue comme une ville : plus la donnée habite loin de la mairie (le processeur), plus le coursier marche longtemps.

Les ordres de grandeur donnent le vertige. Une lecture en cache dure environ 1 ns, un milliardième de seconde. Ramenez-la à **1 seconde** : la RAM répondrait alors en **~1 min 40**, et le disque dur en **~4 mois**. Une donnée qu'on relit souvent doit donc rester **le plus haut possible**.

Lire des cases voisines

Autre réflexe : comme un coursier qui remplit une caisse, la machine ne rapporte jamais un seul octet, elle tire toute une **ligne de cache** (environ 64 octets) d'un coup. **Lire des cases voisines est donc presque gratuit**. Sauter partout coûte cher.

La **localité** est la tendance d'un programme à relire bientôt ce qu'il vient de lire (localité temporelle), et à lire ensuite les cases voisines de celle qu'il lit (localité spatiale). Les deux réflexes ci-dessus s'appuient dessus. Le cache parie sur les deux : la donnée récente reste en haut de la pyramide, ses voisines arrivent

dans la même caisse. Respecter ce pari, c'est aller vite : parcourir des cases dans l'ordre plutôt que de sauter d'une adresse à l'autre, garder ensemble ce qui sert ensemble. Le casser, c'est payer chaque saut.

La défragmentation d'un disque dur applique la même idée. Un fichier y est souvent éparpillé en morceaux, et la tête de lecture, le bras mécanique qui lit le disque, doit sauter de l'un à l'autre. Défragmenter recolle les morceaux côte à côte. Un SSD, disque électronique sans pièce mobile, n'a aucun saut à éviter : la défragmentation n'y sert plus.

Le cas d'école de la localité est le **tableau à deux dimensions** : une grille de cases, en lignes et en colonnes. Vous en croisez tous les jours sans le nommer : une feuille de calcul, un damier, ou le cœur en pixels du 1.1, qui n'est qu'une grille de pixels.

La mémoire, elle, reste une seule longue rangée d'octets, sans lignes ni colonnes. La grille y est donc **aplatie** : la première ligne, puis la deuxième posée juste derrière, et ainsi de suite. Demander la case ligne **i**, colonne **j** (le

`a[i][j]` des langages), ce n'est pas faire un double accès : c'est calculer une seule position, `i × largeur + j`. Dans une grille de 3 colonnes, la case ligne 2, colonne 1 se trouve à la position 7 ($2 \times 3 + 1$), en comptant depuis 0.

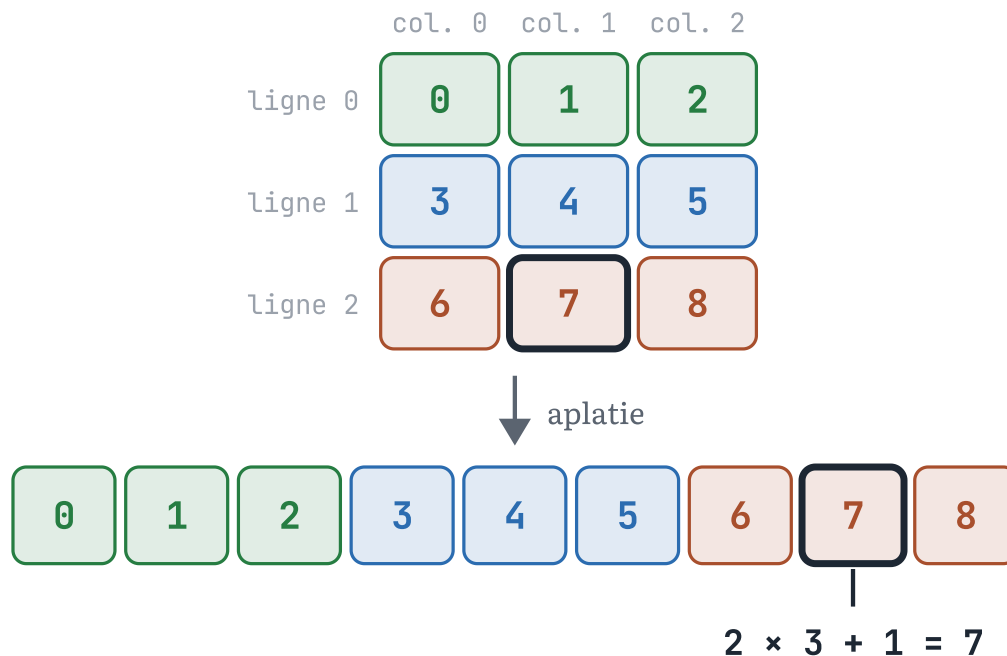


Figure 13 : Une grille de 3 colonnes déroulée en une rangée : chaque ligne garde sa couleur. La case ligne 2, colonne 1 (cerclée) tombe à la position 7.

De là, deux parcours aux coûts opposés. Parcourir une **ligne**, c'est aller de case en case voisine : chacune est déjà dans les 64 octets ramenés par le cache, donc gratuit. Parcourir une **colonne**, c'est sauter une largeur entière de grille à

chaque pas : on tombe sur une nouvelle ligne de cache à chaque pas, et dès que la grille est large, la ligne précédente est chassée du cache avant d'avoir servi. C'est lent.

✓ lire une LIGNE : on traverse les colonnes



j varie, cache chargé 1 fois → rapide

x lire une COLONNE : on descend les lignes



i varie, cache rechargé 3 fois → lent

Figure 14 : Le cadre noir est le lecteur, la zone colorée la ligne de cache chargée.

Même tableau, même calcul : seul change l'ordre des deux **boucles**, des blocs de code qu'on répète, l'un pour chaque ligne, l'autre pour chaque colonne :

```
// même logique, deux ordres de boucles sur un tableau 2D
for (i...) for (j...) a[i][j]
// ✓ on avance dans une ligne : la ligne de cache est réutilisée
for (j...) for (i...) a[i][j]
// x on descend d'une ligne à chaque pas : jusqu'à 10× plus lent
```

Ce gain suppose des **cases de nombres côte à côte**, une ligne après l'autre : c'est le cas en C, en Go avec un tableau `[3][3]int`, en NumPy par défaut et, ligne par ligne, en Java avec `int[][]`. Fortran, Julia ou MATLAB rangent par colonnes, et l'ordre gagnant s'y inverse. Le gain disparaît quand chaque case ne contient que l'adresse d'une donnée rangée ailleurs, comme dans une liste Python ou un `Integer[]` de Java.

Le piège au quotidien

Le même piège se referme tous les jours sur du code ordinaire, sous la forme d'un « **je charge tout en mémoire** » anodin. Ramener 100 000 lignes d'une **base de données** (le logiciel qui range les données et les retrouve à la demande) pour

n'en afficher que dix, c'est faire voyager 10 000 fois trop de données : elles traversent le réseau, sont décodées une à une, remplissent la RAM, et presque tout ce travail est jeté. Le coût vient du déplacement, comme pour le parcours en colonne. **Déplacer des données coûte plus cher que calculer.** Le réflexe : demander à la base seulement les dix lignes voulues (`LIMIT 10`).

↳ [Write Great Code, vol. 1](#) · [vol. 2](#)

1.3 Le coût a une unité : le Big O

Écrire du code qui tourne, c'est bien. Savoir s'il tiendra quand la base passe de cent à dix millions de lignes, c'est mieux. On a vu le prix d'un accès. Reste à compter combien on en fait. Imaginez chercher un nom dans un carnet non trié : on lit les noms un par un. Avec 1 000 noms, c'est 1 000 étapes. Avec dix millions, dix millions d'étapes : le travail suit la taille.

Le **Big O** est l'unité qui dit comment le nombre d'étapes grandit avec la taille de l'entrée, notée `n` (ici, le nombre de noms). Il **ignore les constantes** : peu importe que votre machine soit lente ou rapide, seule compte la façon de grandir. Pour le carnet, on l'écrit `O(n)`, le `O` voulant dire « de l'ordre de » : dix fois plus de données, dix fois plus de travail. Ce n'est donc pas une durée en secondes, c'est une forme de courbe. Quatre familles reviennent partout :

- › **`O(1)`** : temps **constant**, quelle que soit la taille (le plus rapide) ;
- › **`O(log n)`** : temps **logarithmique**, qui grimpe de moins en moins vite : on coupe le problème en deux à chaque étape ;
- › **`O(n)`** : temps **linéaire**, deux fois plus de données font deux fois plus de travail ;
- › **`O(n²)`** : temps **quadratique**, deux fois plus de données en font quatre fois plus (le pire ici).

Le mot « exponentiel » s'emploie souvent à tort pour `O(n²)`. Le vrai temps exponentiel, `O(2n)`, est bien pire : chaque élément ajouté **double** le travail, et quarante éléments suffisent à dépasser mille milliards d'opérations.

Une forme n'est pas une vitesse : un $O(1)$ peut rester lent, si chaque étape est une lecture sur disque dur, cent mille fois plus coûteuse qu'une lecture en RAM.

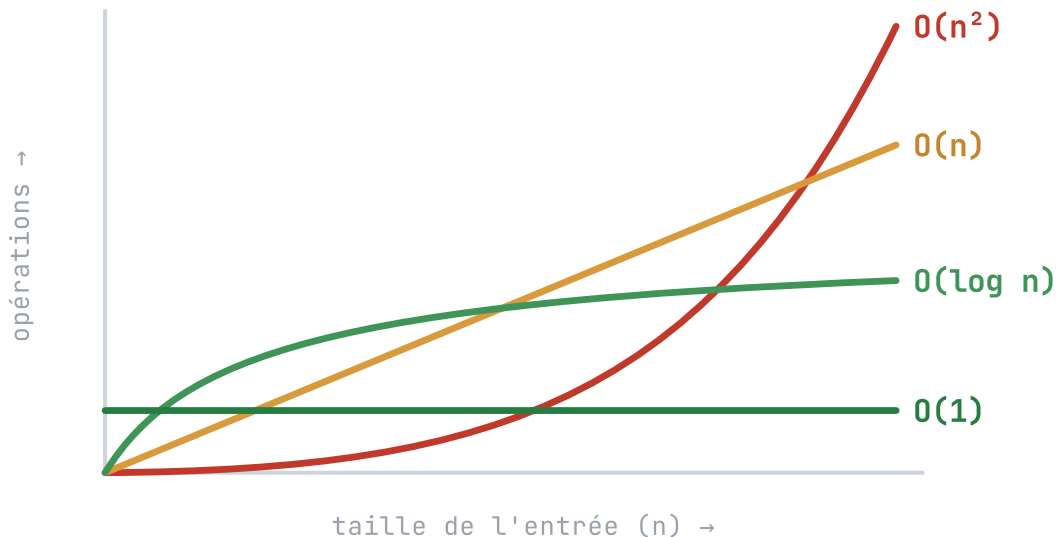


Figure 15 : Quand l'entrée grandit, c'est la famille qui décide tout : $O(n^2)$ explose là où $O(1)$ ne bouge pas.

Trois stratégies pour une même recherche

Le mot logarithme fait peur, l'idée non. $\log n$, c'est le nombre de fois qu'on peut couper n en deux avant de tomber sur 1. Un million coupé en deux vingt fois de suite arrive à 1, donc $\log$ d'un million vaut environ 20. Reprenons le carnet, mais avec les noms rangés dans l'ordre, comme dans un annuaire : on l'ouvre au milieu, on garde la bonne moitié, et on recommence. Même travail, autre stratégie, autre famille : un annuaire d'un milliard de noms se fouille en une trentaine d'étapes. La courbe s'aplatit : doubler les données n'ajoute qu'une seule étape, et mille fois plus de noms n'en ajoutent que dix.

Troisième stratégie pour le même carnet : ne plus chercher du tout. Cela donne du $O(1)$, l'une des idées les plus rentables de la programmation courante. Une **table de hachage** range chaque **valeur** sous une **clé**, par exemple la fiche d'un client sous son nom : pour la retrouver, on donne le nom et on obtient la fiche d'un coup, en temps constant (en moyenne), même sur un million d'entrées. Vous en utilisez déjà : le `dict` de Python, l'objet de JavaScript, le tableau associatif de PHP.

Le mécanisme est simple. La clé passe dans une **fonction de hachage** qui la transforme en numéro de case : on ne cherche rien, on calcule où la valeur est rangée et on y va directement, quel que soit le nombre d'entrées.

Concrètement, la table est un grand tableau de cases numérotées. Prenons la clé « Alice » : la fonction de hachage en fait, par exemple, le numéro 7, et la fiche est rangée dans la case 7. Pour la retrouver, on refait le même calcul et on lit la case 7, sans rien parcourir.

Deux clés peuvent donner le même numéro : c'est une **collision**. Les deux fiches se partagent alors la case ou ses voisines, et l'on compare les clés jusqu'à trouver la bonne. Tant que la table n'est pas trop pleine et que la fonction répartit bien les clés, les collisions restent rares : voilà pourquoi le $O(1)$ vaut « en moyenne ». Quand la table se remplit, on la recopie dans un tableau plus grand, ce qui coûte cher mais arrive rarement. En échange de cette vitesse, la table range les clés en vrac : pour obtenir « tous les noms de A à C », il faut tout parcourir, alors que la liste triée les donne d'un seul bloc.

Même question, trois stratégies, sur un million de noms (en Python, pour l'exemple) :

```
nom in liste
//  $O(n)$  : on lit les noms un par un, jusqu'à 1 000 000 d'étapes
bisect_left(liste_triee, nom)
//  $O(\log n)$  : on ouvre au milieu, une vingtaine d'étapes, renvoie la
// position
nom in dictionnaire
//  $O(1)$  : on calcule où il est rangé, une seule étape
```

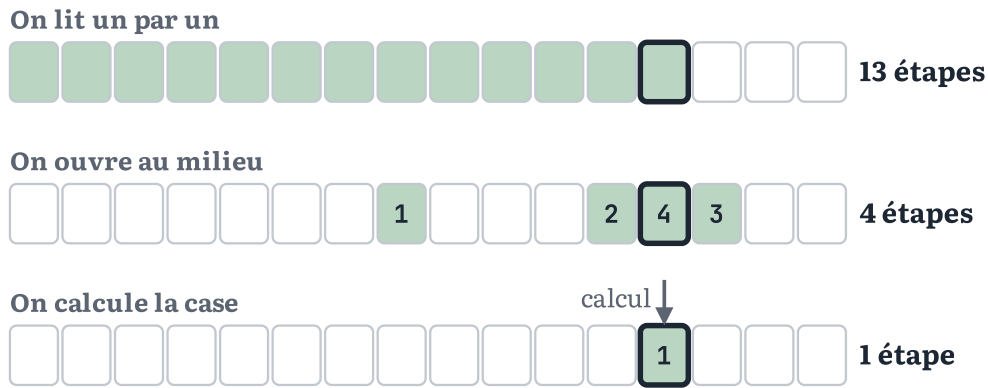


Figure 16 : Le même nom cherché parmi 16 : les cases vertes sont celles qu'on consulte, le cercle est le nom trouvé. Avec un million de noms : jusqu'à 1 000 000 d'étapes, 20, 1.

Un détail cache un coût : l'annuaire de la deuxième stratégie était déjà dans l'ordre. Qui l'y a mis ? Un tri efficace prend environ $n \times \log n$ étapes : c'est la famille **$O(n \log n)$** , entre la droite de $O(n)$ et l'explosion de $O(n^2)$. Pour un million de noms, cela fait vingt millions d'étapes, soit vingt recherches qui lisent tout. Trier ne se paie donc que si l'on cherche ensuite souvent : après une vingtaine de recherches, le tri est rentabilisé. Préparer cher pour chercher pas cher : vous referez cet échange sans cesse, jusqu'à l'index d'une base de données, qui n'est qu'un annuaire tenu à jour pour répondre vite.

Estimer avant de coder

Ces écarts entre familles ne sont pas des vues de l'esprit. Un ordinateur des années 1970, des milliers de fois plus lent que le vôtre, bat une machine moderne ultra-rapide dès que les données grossissent, à condition que le vieux tourne en $O(n)$ et le neuf en $O(n^3)$. Un **algorithme** est une méthode de calcul : une suite d'étapes qui résout un problème. **Aucune puissance de calcul ne rattrape un mauvais algorithme.**

D'où une habitude qui paie : avant d'écrire une ligne, estimez à la louche le nombre d'opérations. Sur dix millions d'éléments, un seul passage ($O(n)$) fait dix millions d'opérations, soit une fraction de seconde. Les comparer deux à deux ($O(n^2)$) en fait dix millions $\times$ dix millions = **cent mille milliards**, des heures de calcul. L'écart est de **dix millions de fois plus de travail** : ce petit calcul, fait en trente secondes, vous dit lequel est jouable avant même de coder.

Pour convertir ces nombres d'opérations en secondes, reprenez une seule équivalence : un milliard d'opérations simples, c'est environ une seconde. Le reste s'en déduit, car chaque fois que le nombre d'opérations est multiplié par mille, la durée l'est aussi. Un million d'opérations passe en un millième de seconde, dix millions en un centième. Cent mille milliards font cent mille secondes, près de trente heures. Le repère est grossier (il suppose des données qui tiennent dans le cache, et en Python il faut compter dix à cent fois plus), et c'est sa force : il se retient, et il sépare le jouable de l'impossible.

Choisir sa structure de données

Le Big O sert aussi à choisir une **structure de données**, la façon dont on range une collection en mémoire. Un **tableau** pose ses éléments côte à côte, comme des casiers numérotés : atteindre le 500^e est instantané, on calcule son adresse et on y va, mais insérer au milieu oblige à **décaler tous les suivants** d'une case. Chacun est recopié un cran plus loin, et recopier prend du temps : c'est le prix de déplacer des données.

Une **liste chaînée** ressemble à une chasse au trésor : chaque élément dit où trouver le suivant. Insérer ne coûte que de **rebrancher deux liens**, sans rien recopier, mais pour atteindre le 500^e il faut **suivre la piste depuis le départ**. En Big O, c'est un miroir : le tableau lit en `O(1)` et insère au milieu en `O(n)`, la

liste chaînée lit en $O(n)$ et insère en $O(1)$ une fois arrivée au bon endroit. En pratique, le tableau gagne presque toujours : ses cases voisines tiennent dans le cache (1.2).

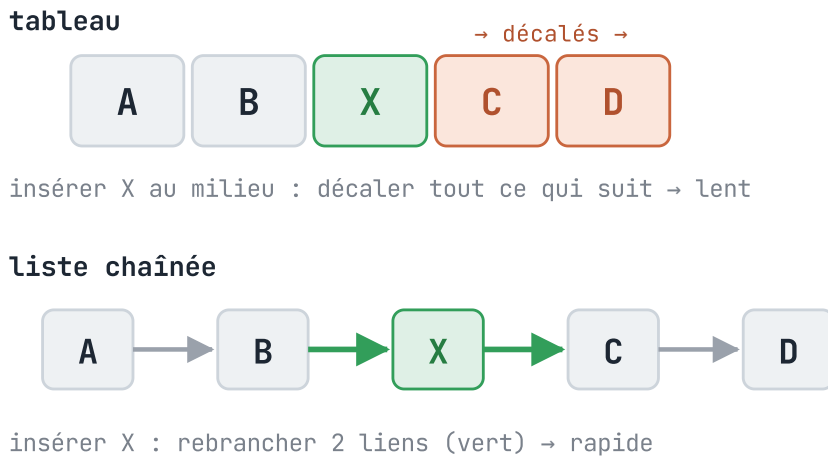


Figure 17 : Insérer au milieu : le tableau décale tout ce qui suit, la liste rebranche deux liens (en vert).

Aucune stratégie n'est « la meilleure » dans l'absolu, ni parmi les algorithmes ni parmi les structures : on choisit selon le problème. Avant de choisir, demandez-vous combien de données vous aurez, ce que le programme fera le plus souvent (lire, insérer, chercher) et ce qui doit rester rapide. Cette réflexion se fait sans coder, à condition de savoir comment la machine range les données, les déplace et compte ses étapes. Une fois l'approche dimensionnée, il reste à l'écrire proprement.

↳ [Grokking Algorithms](#) · [Programming Pearls](#)

1.4 Penser la machine, écrire haut niveau

On code aujourd'hui en langage clair, dit de haut niveau (PHP, JavaScript, Python...), loin du langage brut de la machine, le bas niveau. La puce ne comprend en effet que sa propre liste d'instructions, ces petits ordres en nombres vus en 1.1 : « additionne », « copie », « saute au 40 ». Une simple ligne comme `total = prix * quantite` devient plusieurs de ces ordres : lire le prix, lire

la quantité, les multiplier, ranger le résultat. Tout écrire à ce niveau serait interminable, et à refaire pour chaque famille de puces, chacune ayant sa liste. D'où un traducteur entre les deux.

Voici le trajet d'un programme, de l'écriture à l'exécution. Un **éditeur de code** est le logiciel dans lequel on tape des lignes de code, en texte brut (VS Code, par exemple). Un **fichier** est un contenu rangé sous un nom sur le disque : le texte tapé y est enregistré, `caisse.py` par exemple, où «.py» dit «du Python». Le **code source** est le texte d'un programme tel qu'un humain l'a écrit, avant toute traduction.

Lancer le programme, c'est confier `caisse.py` au traducteur, le logiciel qui le convertit en instructions pour la puce. Un **terminal** est une fenêtre où l'on tape ses demandes à la machine, ligne par ligne. Dans le terminal, la ligne `python caisse.py` demande au traducteur de Python de lire `caisse.py` et d'exécuter ce qu'il y trouve. Modifier le fichier ne change rien tant qu'on ne l'a pas enregistré puis relancé.

Le **compilateur** transforme tout le code en instructions pour la puce avant l'exécution, comme un livre traduit en entier avant d'être publié. L'**interpréteur** traduit au fil de l'exécution, comme un interprète de conférence, phrase après phrase. La frontière est floue : la plupart des langages mélangent les deux (Python, PHP et Java compilent d'abord vers une forme intermédiaire, puis Java et JavaScript compilent à la volée les passages les plus utilisés). «Compilé» ou «interprété» décrit donc la façon dont un langage est exécuté, pas le langage lui-même.

Les leçons qui suivent valent pour les deux. Connaître un peu ce traducteur, et la machine qu'il vise, reste payant pour trois raisons.

D'abord, connaître ce traducteur aide à **flairer un coût caché** derrière une ligne banale. Voici une boucle, un bloc de code répété, qui recompte les mots d'un texte à chaque tour, alors que le texte ne change pas. La machine ne s'en

aperçoit pas toujours : pour n mots, la boucle fait n tours, et chaque tour relit les n mots, soit $n \times n$ étapes. C'est la famille $O(n^2)$. Le remède tient en une ligne : compter une fois, avant la boucle.

```
for (let i = 0; i < nbMots(texte); i++) { ... }  
// x recomptés à CHAQUE tour  
const n = nbMots(texte); // ✓ comptés une fois  
for (let i = 0; i < n; i++) { ... }
```

Dans ce code, `for` répète le bloc tant que la condition `i < n` reste vraie, `i` gagnant 1 à chaque tour. `let` et `const` créent une case nommée. `nbMots(texte)` donne le nombre de mots. Le [chapitre 2](#) détaille ces gestes.

Ensuite, on sait ce que le compilateur fait pour vous, et ce qu'il vous laisse. Il optimise les petits détails tout seul : il calcule par exemple 3×4 une fois pour toutes, au lieu de le refaire à chaque exécution. Mais les optimisations du compilateur restent **locales** : il ne transformera pas une recherche lente en recherche rapide. **L'algorithme reste votre travail. La machine ne fait que polir ce que vous lui donnez.**

Enfin, le bas niveau réserve des surprises. Pour aller vite, le processeur **parie** à l'avance sur le résultat de chaque test, ces « si telle condition, alors... » qui s'écrivent `if` dans le code. C'est la « prédiction de branche ». Il commence à travailler sans attendre la vraie réponse : si le pari est bon, rien n'est perdu, et s'il est mauvais, ce travail d'avance est jeté.

Cela donne un résultat absurde mais réel. Pour compter les nombres supérieurs à 128 dans un tableau, le processeur pose la même question à chaque case. Si le tableau est trié, la réponse est faux, faux, faux..., puis vrai, vrai, vrai... : le pari « même réponse que la dernière fois » est presque toujours bon. En désordre, la réponse change au hasard et le pari est perdu environ une fois sur deux : le même parcours peut être plusieurs fois plus lent. Ce cas est devenu célèbre sur Stack Overflow, le site d'entraide des développeurs. Mais tout dépend du compilateur : certains transforment le test en simple calcul, sans pari possible, et l'écart disparaît.

Personne ne devinerait ça, et des effets de ce genre, il y en a beaucoup : sur la performance, l'intuition se trompe souvent. Le seul moyen fiable de savoir si un code est rapide, c'est donc de le **mesurer**, c'est-à-dire de le chronométrer pour de vrai. On estime avant de coder, on mesure après.

La mesure a son outil et son piège. Le **profileur** est un programme qui observe le vôtre pendant qu'il tourne et dit où passe le temps, par fonction (un morceau de code nommé) ou par ligne. Son verdict surprend presque toujours : quelques lignes mangent l'essentiel, rarement celles qu'on soupçonnait.

Reste le piège classique. Un **microbenchmark** chronomètre une ligne seule, hors contexte. Il peut tromper de trois façons :

- › le compilateur peut supprimer un calcul dont le résultat ne sert à rien ;
- › le cache, déjà rempli par les tours précédents, répond plus vite qu'en vrai (1.2) ;
- › un langage qui compile à la volée n'a pas encore optimisé le passage mesuré.

Mesurez le vrai programme sur de vraies données, plusieurs fois.

↳ [Write Great Code, vol. 2](#)

↳ CE QUI MÈNE AU [CHAPITRE 2](#)

*La machine ne parle que des nombres et range ses données par étages. Chaque pas se paie : on sait maintenant l'estimer avant de coder, et qu'il faudra le mesurer après. Un traducteur fait le reste, mais il ne lit que du code, et quelqu'un doit bien l'écrire. Reste à choisir les mots dans lesquels on exprime son intention : le **langage de programmation**. Il ne sert pas seulement à donner des ordres, il devient un outil pour penser : ses mots décident de ce qu'on est capable de concevoir.*

« Le livre de la nature est écrit en langage mathématique. » — Galilée, *Il Saggiatore*, 1623

2 Le langage, outil de pensée

exprimer

Vous savez comment la machine range ses nombres et ce que chaque accès coûte. Il faut maintenant apprendre à lui parler, et les mots qu'on emploie comptent : un langage rend certaines idées simples, d'autres pénibles. Additionner des prix tient en un mot dans le langage Python, `sum(prix)`, alors qu'en langage C il faut écrire soi-même la répétition : prendre un prix, l'ajouter au total, passer au suivant. Une idée court sous tout le chapitre : nommer une chose, c'est agrandir ce qu'on peut penser.



Figure 18 : Tant qu'une chose n'a pas de nom, on ne sait pas la manipuler. Nommée, on la range, on la compare, on la combine.

Deux mots reviennent sans cesse, et chacun nomme une chose : autant les poser tout de suite. Une **variable** est un nom collé sur une case de la mémoire, comme une étiquette sur un casier. Nous dirons désormais casier. Le nom est rattaché à l'adresse du casier (1.1), et le casier contient une donnée, des 0 et des 1 qui disent un nombre, un prix, une lettre. On range la valeur, on la retrouve par son nom, on peut la remplacer : le nom reste, le contenu change, d'où « variable ».

Une **fonction** est une recette écrite une fois pour toutes : on lui donne des ingrédients, ses **arguments**, elle rend un plat, son **résultat**. La fonction `sous_total(prix, quantité)` porte ses arguments entre parenthèses : elle reçoit un prix et une quantité, et rend ce que coûtent ces articles ensemble, 90 € pour deux pulls à 45 €. On l'appelle (on la lance) autant de fois qu'on veut, elle refait le même travail.

La fonction, elle aussi, vit en mémoire : ses instructions sont rangées quelque part, et son nom est une étiquette collée sur l'adresse de la première. L'appeler, c'est dire à la puce de sauter à cette adresse puis, le travail fini, de revenir à l'instruction qui suit l'appel. Nommer une recette, c'est donc pouvoir la rappeler d'un mot.

2.1 Les trois gestes : enchaîner, répéter, choisir

Un programme est une suite d'instructions rangées en mémoire. La puce les exécute l'une après l'autre, dans l'ordre de leurs numéros. Avec des variables pour garder les valeurs, trois gestes suffisent à écrire n'importe quel programme : enchaîner, répéter, choisir. Ce n'est pas un avis mais un théorème, démontré en 1966 par Corrado Böhm et Giuseppe Jacopini.

Ce théorème a changé les habitudes. Dans les années 1960, les langages offraient le `goto`, « va à la ligne N », un saut vers n'importe quel endroit du programme. En 1968, Edsger Dijkstra, dans un article titré *Go To Statement Considered Harmful* (« le goto jugé nuisible »), soutient qu'un tel programme devient impossible à suivre : on ne sait plus par où la puce est passée.

Le théorème montrait qu'on pouvait s'en passer. Un programme se lit alors de haut en bas, chaque répétition et chaque choix ayant un début et une fin. Sous le capot, la puce saute toujours, mais le langage le fait pour vous, de façon rangée. Tous les langages courants offrent ces trois gestes, seule leur écriture change.

Pensez à une caisse de magasin. Chaque article passe devant le lecteur de codes-barres, et deux pulls identiques ne font qu'un bip, suivi de « × 2 ». Le total monte à chaque bip. Au moment de payer, la caisse retire 10 % si le total dépasse 100 €.

Un **pseudo-code** décrit un calcul en phrases presque françaises : on le réécrit ensuite dans le langage de son choix, et c'est la bonne habitude, penser d'abord, traduire après. Voici la caisse en pseudo-code, la fonction `total_panier`, suivie de ce que fait la puce pour deux pulls à 45 € et un livre à 15 €. Le `panier` y est la suite des articles scannés, `article.prix` se lit « le prix de cet article » et `article.quantité` « combien il y en a ».

```

fonction total_panier(panier) :
    total = 0
    // enchaîner : une ligne, puis la suivante
    pour chaque article du panier :
        // répéter : une boucle, un tour par bip
        total = total + article.prix × article.quantité
    si total > 100 :                                // choisir : oui ou non
        total = total × 0,9                        // 10 % de remise
    rendre total

// la puce, pas à pas, avec 2 pulls à 45 € et 1 livre à 15 € :
// total = 0
// tour 1, pull : total = 0 + 45 × 2 = 90
// tour 2, livre : total = 90 + 15 × 1 = 105
// plus d'article, la boucle s'arrête
// 105 > 100 ? oui, donc total = 105 × 0,9 = 94,5
// rendre 94,5

```

Lisons ce code comme le ferait la puce. Elle part de la première instruction, `total = 0` : elle repère le casier nommé `total` et y écrit 0, qui remplace ce que le casier contenait avant. Le signe `=` ne veut donc pas dire « égale » : il veut dire « mets dans ». La puce calcule toujours la droite d'abord, puis range le résultat dans le casier nommé à gauche. Ainsi `total = 0` se lit « mets 0 dans le casier `total` ».

Enchaîner, c'est exécuter les instructions de haut en bas, dans l'ordre où elles sont écrites : chacune démarre quand la précédente est finie. L'ordre compte : placez le « si » avant la boucle, et la puce se demande si le total dépasse 100 alors qu'il vaut encore 0. La remise ne se déclenche jamais, et le client à 105 € paie plein tarif.

Ligne suivante, `pour chaque`. On **répète** grâce à une **boucle** : un morceau de programme que la puce refait une fois par élément à traiter. Ici, elle fait un tour par bip. Chaque tour prend l'article suivant, lit le casier `total`, y ajoute

prix fois quantité, et réécrit le résultat dans le même casier. C'est pourquoi `total = total + ...` a un sens. Puis la puce remonte à la ligne `pour chaque` : c'est le saut en arrière de 1.1. Le `sum(prix)` de l'introduction cache exactement cette boucle.

Après les pulls, le casier contient 90. Après le livre, 105, soit `01101001`, le même motif de 0 et de 1 que la lettre «i» de 1.1. Le casier ne sait pas ce qu'il garde : c'est le programme qui décide de le lire comme un prix ou comme une lettre. Deux bips, deux tours, puis la puce passe à la ligne du dessous. Mille bips, mille tours : le coût grandit comme le panier, c'est le `0(n)`.

Reste la remise. On **choisit** avec un «si» : la puce pose une question dont la réponse est oui ou non, puis exécute le morceau de programme qui suit ou le saute. Elle lit le casier `total` et compare son contenu à 100. Il contient 105, la réponse est oui, la ligne `total = total × 0,9` s'exécute et réécrit le casier, qui contient maintenant 94,5. Avec un seul pull, la réponse aurait été non, et la puce aurait sauté par-dessus la ligne, droit vers `rendre total` : c'est le saut en avant.

Ce déroulé à la main, où l'on joue la puce une instruction à la fois, est le meilleur moyen de comprendre un programme qu'on lit, et le premier réflexe quand un résultat surprend.

D'où vient le panier, où va le total

Le panier n'est pas né dans le programme, et 94,5 ne doit pas y rester. À la caisse, les articles arrivent par le lecteur de codes-barres, et le total part vers l'écran et l'imprimante du ticket. Un programme est pareil. L'**entrée** d'un programme, c'est tout ce qu'il reçoit du dehors : une touche du clavier, un fichier sur le disque, un code-barres, un message d'une autre machine par le réseau

(chapitre 5). La **sortie** d'un programme, c'est tout ce qu'il envoie au dehors : un texte à l'écran, un fichier écrit, un message sur le réseau. Entre les deux, le calcul, qui ne touche que des casiers.

```
panier = lire_codes_barres()    // entrée : la donnée vient du dehors
total = total_panier(panier)    // calcul : rien que des casiers
afficher(total)                // sortie : le résultat part vers l'écran
```

Chaque langage a ses mots pour l'entrée et la sortie : en Python, `print(total)` (de l'anglais « imprimer », mais il écrit à l'écran, dans le terminal) joue le rôle de notre `afficher`, et `input()` attend une ligne tapée au clavier et la rend sous forme de texte, même si on a tapé 45 : il faut la convertir en nombre avant de calculer. Entrée, calcul, sortie : c'est le « des nombres entrent, des nombres sortent » de l'ouverture du [chapitre 1](#), vu depuis le programme.

Ne confondez pas rendre et afficher : `rendre total` remet le résultat au code qui a appelé la fonction, qui peut s'en resservir. `afficher(total)` l'envoie hors du programme : le reste du calcul ne le retrouve plus. Quand un résultat est faux, regardez d'abord ce qui est entré : un calcul juste sur une donnée fausse donne un résultat faux, et la machine ne s'en plaint pas.

Où vit une variable, et jusqu'où

Le casier `total` a servi le temps d'un calcul. Combien de temps vit-il, et qui peut y toucher ? Dans `total_panier`, la variable `total` naît quand la fonction démarre et disparaît quand elle rend son résultat : chaque appel réserve ses propres casiers de mémoire et les libère en partant. La **portée** d'une variable est la zone du programme où son nom est visible. Une autre fonction, `total_commande` par exemple, peut donc avoir son propre `total` : même étiquette, autre casier, comme deux vestiaires qui ont chacun un casier marqué « Paul ».

Une variable posée hors de toute fonction, qu'on dit globale, est visible de presque partout dans le programme. Imaginez que `total` le soit : un autre morceau du programme, qui calcule des frais de livraison, pourrait y écrire

entre la fin du calcul et l’affichage, et l’écran montrerait un total faux, sans aucune erreur. Le jour où la valeur est fausse, on ne sait plus qui l’a touchée. D’où la règle : donner à chaque variable la plus petite portée possible. Les règles exactes de portée varient d’un langage à l’autre, et c’est l’une des premières choses à connaître du vôtre : c’est là que se cachent les surprises.

Quand une instruction échoue

Certaines instructions sont impossibles à exécuter. Calculer le prix moyen d’un panier vide, par exemple, revient à diviser 0 € par 0 article. Face à une instruction impossible, beaucoup de langages arrêtent le programme et affichent une **erreur**, un message qui dit ce qui a échoué et à quelle ligne. Le texte change d’un langage à l’autre : Python écrit par exemple `ZeroDivisionError: division by zero`, précédé du fichier et de la ligne fautive. En JavaScript, au contraire, `0 / 0` rend `NaN`, « pas un nombre », sans aucune erreur, et ce `NaN` voyage jusqu’à l’écran (2.6). S’arrêter net est un service : mieux vaut un programme qui échoue que l’affichage d’un total faux.

Une question reste ouverte. Quand la puce lit `article.prix`, que trouve-t-elle dans le casier d’`article` : le prix lui-même, ou l’adresse où aller le chercher ? La réponse change tout ce qui suit.

2.2 Valeurs et références : ce que la variable contient vraiment

Le casier d’une variable peut contenir deux sortes de choses : la donnée elle-même, ou l’adresse où aller la chercher. Dans la plupart des langages, cela dépend de la donnée. Un **nombre**, comme le prix de 45 €, tient tout entier dans le casier de sa variable. Un **booléen**, une valeur qui ne peut être que vrai ou faux, tient lui aussi dans un seul casier.

Un article de la caisse, en revanche, ne tient pas dans un seul nombre. Un **objet** regroupe des données et les actions qu’on peut lui demander : l’article en est un, avec son nom, son prix et sa quantité. Les données qu’un objet porte sont

ses **champs** (`nom`, `prix`, `quantité`), les actions qu'il sait faire sont ses **méthodes**, des fonctions rangées dans l'objet (`appliquerRemise()`). Tout le chapitre 4 reposera sur les objets.

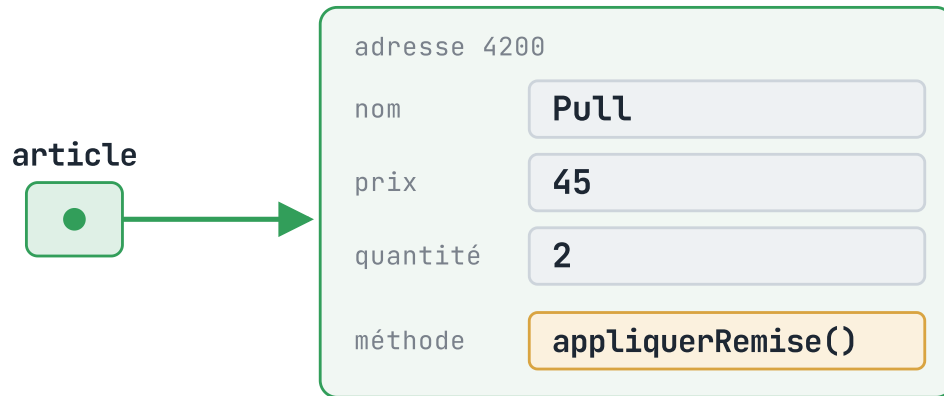


Figure 19 : Un objet est un groupe de casiers (ses champs) et d'actions (ses méthodes) rangé sous une seule adresse. La variable `article` ne garde que cette adresse.

Une **liste** range des valeurs à la suite, un **dictionnaire** les retrouve par un nom. Dans la plupart des langages, pour un objet comme pour une liste ou un dictionnaire, la variable ne contient que l'adresse de la donnée, son numéro de case. Une adresse rangée dans une variable s'appelle un **pointeur**, ou une référence. D'où la réponse à la question laissée ouverte en 2.1: le casier d'`article` contient une adresse. Pour lire `article.prix`, la puce saute à cette adresse et y lit le prix.

Copier une variable : par valeur ou par référence

Copier une variable, c'est écrire `y = x` : « mets dans le casier `y` ce que contient le casier `x` ». Si `x` contient le nombre 5, `y = x` **duplique la valeur** : `y` reçoit son propre 5, et changer `y` ne touche pas `x`. C'est la copie « par valeur », comme une photocopie : on peut griffonner dessus, l'original ne bouge pas.

Si `a` désigne une liste, comme `[1, 2, 3]`, son casier contient une adresse, et `b = a` recopie cette adresse, rien de plus. Après `b = a`, la liste n'existe toujours qu'une fois en mémoire, et les variables `a` et `b` sont **deux noms pour une seule donnée**. C'est la copie « par référence », comme le lien d'un do-

cument partagé : si quelqu'un modifie le document, tout le monde voit la modification, parce qu'il n'y en a qu'un. Ainsi `b[0] = 9`, qui change le premier élément de la liste (on compte à partir de 0), change aussi `a[0]`.

Écrire ensuite `b = [7, 8]`, en revanche, ne touche pas `a`. Cette ligne ne suit pas l'adresse pour modifier la liste : elle fabrique une nouvelle liste ailleurs en mémoire et range son adresse dans le casier de `b`, à la place de l'ancienne. Le casier de `a` n'a pas bougé et contient toujours l'adresse de la première liste. Retenez la différence : `b[0] = 9` suit l'adresse et change la liste, que les deux noms voient, alors que `b = [7, 8]` change seulement ce que contient le casier de `b`.

En résumé, deux cas. Ce qui les sépare n'est pas la taille de la donnée mais une question : peut-on la modifier sur place, c'est-à-dire changer son contenu sans en fabriquer un nouveau ? Une donnée qu'on ne peut pas modifier sur place est **immuable**. Pour elle, partager ou dupliquer revient au même, et le langage choisit ce qui lui coûte le moins.

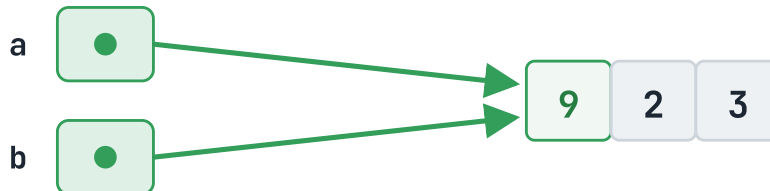
- › par valeur, pour les données qu'on ne modifie pas sur place (nombre, booléen, texte : mettre un texte en majuscules en rend un nouveau, l'original reste intact) : copier la variable en fait une seconde, et les deux vivent séparément ;
- › par référence, pour les données modifiables sur place (objet, liste, dictionnaire, dans la plupart des langages) : le casier contient l'adresse, copier la variable ne recopie que l'adresse, et une modification faite sous un nom se voit sous l'autre.

valeur (un nombre, un booléen)



deux casiers séparés : changer y ne touche pas x

référence (objet, liste, dictionnaire)



une seule donnée, deux noms : `b[0] = 9` change aussi `a[0]`

Figure 20 : En haut, la valeur se recopie dans un second casier. En bas, a et b mènent à la même donnée.

```
const a = [1, 2, 3];
// en JavaScript, const fige l'adresse, pas le contenu
const b = a;           // b reçoit la même adresse : la MÊME liste
b[0] = 9;
// a[0] vaut 9 aussi : une seule donnée derrière deux noms
```

Passer à une fonction, copier en profondeur

Dans la plupart des langages (JavaScript, Python, Java), un objet, une liste ou un dictionnaire donné en argument à une fonction n'est pas recopié : si la fonction le modifie, elle modifie l'**original**. Confondre valeur et référence est une source inépuisable de bugs « j'ai changé une copie et l'original a bougé ». Le réflexe, quand on code : avant de modifier un objet reçu, se demander qui d'autre tient son adresse. C'est le défaut de la variable globale de [2.1](#) sous un autre habit : plusieurs mains sur la même donnée. Python en donne un piège classique : `[[0] * 3] * 3` ne fabrique pas trois lignes, mais trois fois la même ligne.

Ce partage d'adresse a une conséquence pratique : copier une liste ne la recopie pas toujours en entier. Une copie superficielle, `[...a]` en JavaScript ou `a.copy()` en Python, ne recopie que la liste elle-même : si elle contient des objets, la copie contient les mêmes adresses, donc les mêmes objets. Pour tout

recopier jusqu'au fond, il faut demander une copie profonde: `structuredClone(a)` en JavaScript pour une liste ou un dictionnaire, `copy.deepcopy(a)` en Python.

↳ [Learning Go](#) · [PHP 8 Objects](#)

2.3 Les types sont des ensembles de valeurs

Dans la section précédente, on a croisé des nombres, du texte, des listes, des objets. Chaque sorte de donnée est un **type**. Presque tous les langages offrent les mêmes briques de base, sous des noms qui varient :

- › le nombre, entier ou à virgule (`number` , `int` , `float`) : `42` ou `3.14` ;
- › le booléen (`boolean`) : vrai ou faux ;
- › la chaîne de caractères (`string`) : du texte, comme `"bonjour"` , une suite de lettres qui sont chacune un nombre (l'ASCII et l'UTF-8 de 1.1) ;
- › la liste, ou tableau (`array` , `list`) : des données côte à côte, comme `[1, 2, 3]` ;
- › le dictionnaire (`map` , `dict`) : des données retrouvées par un nom, comme `{nom: "Léa"}` (la table de hachage de 1.3) ;
- › l'objet (`object`) : des données et des actions regroupées.

Le type décide de ce qu'on peut faire d'une valeur: `2 + 2` vaut 4, mais `"2" + "2"` , du texte, donne `"22"` dans la plupart des langages (PHP, lui, convertit et répond 4). Mêmes signes, autre type, autre résultat. Ce que fait un langage devant un type inattendu varie, et c'est une source classique de surprises : en cas de doute, on teste l'opération dans le langage qu'on utilise.

Pourquoi écrire ses types

À quoi sert de déclarer ses types, c'est-à-dire d'écrire noir sur blanc ce qu'on attend de chaque variable ? Beaucoup de langages, comme JavaScript, Python ou PHP, permettent de s'en passer : ils ne vérifient qu'à l'exécution, et on découvre la faute quand le code tourne.

Deux questions distinctes se cachent là, et on les confond souvent. Le typage **statique** vérifie les types avant que le programme tourne (Java, ou TypeScript, expliqué plus bas). Le typage **dynamique** les vérifie pendant qu'il tourne (Python, JavaScript). Et devant un type inattendu, le langage refuse-t-il avec une erreur ou convertit-il en silence ? Python et JavaScript sont tous deux dynamiques, mais le premier refuse `"5" - 1` quand le second répond 4 (2.6) : on dit l'un **strict**, l'autre **permissif** (typage fort et faible, dans le jargon). Les deux axes sont indépendants.

Voici une petite fonction `total`, le prix fois la quantité, écrite en JavaScript et appelée avec de mauvaises entrées :

```
function total(prix, quantite) { return prix * quantite; }
total("12", 3)
// 36 : un "12" tapé dans un formulaire arrive en texte, et marche par
// accident
total("douze", 3) // NaN, « pas un nombre », qui voyage jusqu'à l'écran
```

Un type déclaré est un engagement : « ici, il y aura un nombre ». TypeScript, c'est JavaScript plus ces engagements, écrits dans le code :

```
function total(prix: number, quantite: number) { return prix * quantite; }
total("12", 3)
// erreur signalée avant de tourner : « 12 » est du texte
total("douze", 3)
// erreur signalée aussi : « douze » n'est pas un nombre
```

Le compilateur de TypeScript (vu en 1.4) relit l'engagement avant que le code tourne, une relecture que JavaScript seul n'avait pas, et signale chaque engagement trahi avant le premier lancement. Vous ne gagnez pas en vitesse de frappe, vous gagnez le **moment** où la faute apparaît : devant vous, pas devant l'utilisateur.

Un type rend un second service, moins visible et plus rentable au quotidien : il documente. `envoyer(destinataire, message)` ne dit pas ce qu'il attend. `envoyer(client: Client, message: string)` le dit, et l'éditeur de code connaît `Client` et vous propose ses champs pendant que vous tapez. La pre-

mière ligne d'une fonction, son nom suivi de ce qu'elle attend, s'appelle sa **signature**: avec des types, elle devient une documentation que le compilateur vérifie. Tant que les types sont respectés, elle ne ment pas.

Un type est un ensemble de valeurs

Un type n'est pas qu'une étiquette (`int`, `string`, `boolean`): c'est l'**ensemble des valeurs** qu'une variable peut prendre. Cette idée simple suffit à expliquer comment on combine des types, et comment un langage décide qu'un objet y entre.

- › le type `boolean` ne contient que **deux** valeurs: vrai et faux;
- › le type `"rouge" | "vert" | "bleu"`, où la barre se lit « ou » (un texte forcément l'un de ces trois mots), en contient **trois**;
- › le type `string`, une **infinité**.

Un type qui énumère ses valeurs une à une, comme les trois couleurs, est une **énumération** (`enum` dans beaucoup de langages). Le statut d'une commande en est une, `"en attente" | "payée" | "annulée"` et rien d'autre: la faute de frappe `"payee"` est refusée avant même que le code tourne. C'est la forme qui rend la faute impossible, ce que 3.4 appellera un détrompeur. Un état qu'on ne peut pas écrire est un bug qu'on ne peut pas avoir.

Un ensemble manque encore à la liste: l'absence. Un casier peut n'avoir pas encore de valeur, `null` en Java et en JavaScript, `None` en Python, `nil` en Go, et un casier « client ou rien » est deux types en un. Oublier le second est le bug le plus répandu de l'histoire du logiciel: Tony Hoare, qui a inventé la référence nulle en 1965, l'a appelée son « erreur à un milliard de dollars ». Vu en ensembles, le remède s'écrit tout seul: `Client | null` dit l'absence dans le type, et le compilateur exige le test avant d'ouvrir le casier, ce que d'autres langages nomment `Optional` ou `Option`. On verra en 3.4 quand il vaut mieux qu'il n'y ait rien à tester.

Sous cet angle, on combine des types comme des ensembles. L'**union** (`A | B`) réunit toutes les valeurs des deux types. L'**intersection** (`A & B`) ne garde que les valeurs qu'ils ont en commun. Elle sert à exiger deux choses à la fois, par

exemple un objet qui a les champs d'un client et ceux d'un compte connecté : plus on exige de champs, moins il y a d'objets qui conviennent, donc l'ensemble rétrécit.

un type = un ensemble de valeurs

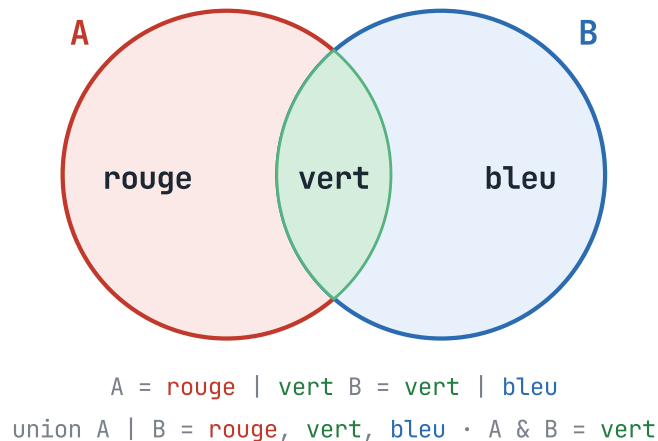


Figure 21 : Deux types, A et B : l'union garde tout, l'intersection ne garde que le chevauchement.

Une **interface** est la liste de ce qu'on exige d'un objet : des champs, comme `x` et `y` pour un `Point`, ou des actions, comme lire et se fermer. En Go, toute valeur qui sait faire les actions d'une interface la satisfait, sans l'avoir déclaré : un fichier et une connexion réseau savent tous deux lire et se fermer, donc le même code peut lire l'un ou l'autre.

La vision en ensembles explique aussi le **typage structurel** : un objet entre dans un type selon sa forme, pas selon l'étiquette qu'il porte. C'est une serrure qui ne demande pas qui a fabriqué la clé, seulement si elle a la bonne forme. Un objet qui a un `x` et un `y` est un `Point`, même s'il n'a jamais entendu parler de `Point`.

```
interface Point { x: number; y: number }
function longueur(p: Point) {...}
// longueur attend un Point : un x et un y
const p = { x: 3, y: 4, z: 5 };
longueur(p)
// ✓ ok : la forme {x, y} y est, le z en trop est ignoré
```

Le **typage nominal** (Java, PHP) fait l'inverse: l'objet doit déclarer noir sur blanc `implements Point`, «je suis un Point», pour être accepté. Avec le typage structurel, la **forme suffit**.

« Je ne sais pas » vaut mieux que « fais-moi confiance »

Dernière idée, tirée d'*Effective TypeScript* (Dan Vanderkam): elle dépasse de loin TypeScript. Une donnée venue du dehors, la réponse d'un serveur (un ordinateur distant) ou un fichier, arrive en texte JSON (du texte structuré, le format d'échange des serveurs). `JSON.parse` le transforme en objet, et personne ne garantit ce qu'il contient.

Face à une donnée dont on ignore la forme, il n'y a que deux postures. Pré-tendre qu'on sait, et le langage se tait. Admettre qu'on ne sait pas, et il exige une vérification avant le premier usage. C'est un inconnu à la porte: on le laisse entrer sans rien demander, ou on lui demande d'abord ses papiers.

```
// txt : un texte JSON reçu du dehors
let a: any = JSON.parse(txt); a.client.nom
// x « fais-moi confiance » : aucune vérification. Si client manque,
// erreur à l'exécution
let b: unknown = JSON.parse(txt); b.client
// ✓ erreur à la compilation : prouve d'abord ce que c'est
```

- › `any` éteint la vérification du compilateur: c'est un « fais-moi confiance ». `any` **contamine en silence** tout ce qu'il touche: ce qu'on en tire, comme `a.client`, devient `any` à son tour.
- › `unknown` dit l'inverse: « je ne sais pas encore ce que c'est, je vérifierai avant de m'en servir ». Avec `unknown`, le compilateur **reste en garde**: il refuse tout usage tant que le type n'est pas prouvé par un test écrit dans le code, comme `si b est un nombre`. Les types disparaissent à l'exécution: seul ce test protège des données réelles.

On retrouvera cette posture devant une saisie d'utilisateur en [7.6](#), puis devant une réponse d'IA en [8.1](#): ne jamais traiter comme sûr ce qu'on n'a pas vérifié.

↳ [Effective TypeScript](#) · [Learning Go](#)

2.4 Les fonctions sont des valeurs

Un type dit ce qu'on peut faire d'une valeur. Or, dans les langages modernes, une fonction est elle aussi une valeur, qui se manipule comme n'importe quelle donnée : on la range dans une variable ou dans un objet (c'est ce qu'est une méthode), on la passe en argument, on la rend comme résultat. Rien de magique : on a vu en [introduction de ce chapitre](#) qu'elle vit en mémoire, son nom n'étant qu'une étiquette sur son adresse. On peut donc ranger cette adresse ailleurs, comme n'importe quelle donnée. Deux exemples en JavaScript :

```
bouton.addEventListener('click', maFonction)
// « quand on clique, exécute celle-ci »
[1, 2, 3].map(doubler)
// « map (ici : transformer) applique doubler (qui multiplie par 2) à
// chaque élément » : [2, 4, 6]
```

Dans les deux cas, une fonction voyage comme une donnée. Cette seule idée débloque trois outils, qui paraissent magiques tant qu'on n'en a pas vu le mécanisme :

- › la **closure** : une fonction qui repart avec **son propre casier**. Elle garde la portée où elle est née : les variables de cet endroit lui restent accessibles d'un appel à l'autre (un compteur qui garde son total privé) ;
- › le **décorateur** : un **post-it agrafé** sur une fonction. Il ajoute un comportement sans raturer une seule ligne de l'original (chronométrer, garder un résultat déjà calculé, vérifier des droits) ;
- › le **générateur** (`yield`) : une fonction qui vous **tend une valeur, s'arrête, et reprend à la suivante** quand vous en redemandez. Elle n'a jamais toute la suite en main (parcourir un fichier énorme sans le charger en entier).

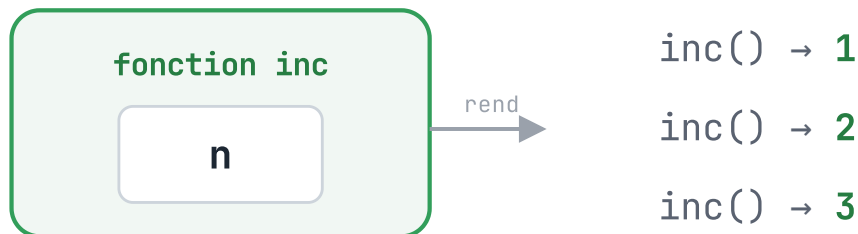
La closure : une fonction qui se souvient

La closure surprend le plus. Un exemple en Python :

```
def compteur():
    n = 0                # n vit à l'intérieur de compteur()
    def inc():
        nonlocal n      # « le n du dessus, pas un nouveau »
        n += 1
        return n
    return inc           # on rend la FONCTION, pas son résultat

c = compteur()          # c est maintenant inc
c()                      # 1
c()                      # 2 ← n a survécu d'un appel à l'autre
```

une closure : inc emporte son n privé



n est privé à inc et survit entre les appels, même après que compteur() a fini

Figure 22 : À chaque appel, inc ajoute 1 à son `n` et s'en souvient, d'où 1, puis 2, puis 3. C'est ça, une closure.

Ici, `inc` est écrite à l'intérieur de `compteur()` et utilise son `n` : voilà ce qui fait d'elle une closure. En 2.1, les casiers d'une fonction disparaissaient à la fin de l'appel. Là, `compteur()` a fini, et pourtant `n` survit : `inc`, qu'on a gardée (ici en la renvoyant), l'emporte avec elle. La closure existe dans presque tous les langages : JavaScript et Go gardent l'accès à la variable de l'endroit où la fonction est née tout seuls, Python aussi mais exige `nonlocal` pour la modifier. Pour mesurer son intérêt, voyons ce qu'elle remplace.

Pour qu'une fonction se souvienne d'une valeur d'un appel au suivant, on passe sinon par un objet (2.2), par une variable globale, visible de partout, ou, dans certains langages comme le C, par un `static`, un casier fixe attaché à la fonction. Les valeurs ainsi gardées forment l'état de la fonction :

- › avec un `static` : un seul état, partagé par tous les appels de la fonction ;
- › avec une closure : un état **neuf à chaque appel** de `compteur()`, donc autant de compteurs indépendants qu'on veut.

Au fond, une closure, c'est une fonction qui emporte avec elle **un casier à elle** : `n` reste caché du reste du programme, et chaque `compteur()` en fabrique un neuf (deux closures nées du même appel partagent, elles, le même casier).

Si elle paraît dense, c'est qu'elle réunit trois idées :

- › une fonction se range dans une variable ;
- › une variable a une portée ;
- › une donnée reste en mémoire tant que quelqu'un s'en sert.

Le **ramasse-miettes** est le mécanisme du langage qui libère la mémoire que plus aucune variable n'atteint : quand plus rien ne tient `inc`, elle disparaît, et son `n` avec elle.

Le décorateur : emballer sans toucher

Le décorateur en découle directement. Chronométrer une fonction sans toucher à son code, par exemple, toujours en Python :

```
from time import time
# time() : l'heure en secondes, fournie avec Python
def chrono(f):
    # prend une fonction, en rend une autre
    def emballage(*a, **kw):
        # *a, **kw : tous les arguments de f, quels qu'ils soient
        t = time()
        r = f(*a, **kw)          # on exécute la fonction d'origine
        print(time() - t)       # et on affiche le temps qu'elle a pris
        return r
    return emballage

@chrono
# @ : emballe lent(), qui est maintenant chronométré
def lent(): ...

lent()
# s'exécute normalement, et affiche sa durée au passage
```

La ligne `@chrono` est un raccourci pour `lent = chrono(lent)` : l'étiquette `lent` est recollée sur `emballage`, une closure qui se souvient de la vraie fonction, `f`. Chaque appel à `lent()` passe donc par l'emballage. Des outils web en Python comme Flask ou Django s'en servent pour vérifier, en une ligne (`@login_required`), que l'utilisateur est connecté avant d'exécuter une fonction.

Le générateur : une valeur à la fois

Le générateur, lui, évite le piège « je charge tout en mémoire » de la [section 1.2](#) en rendant ses valeurs une à une, à la demande, en Python :

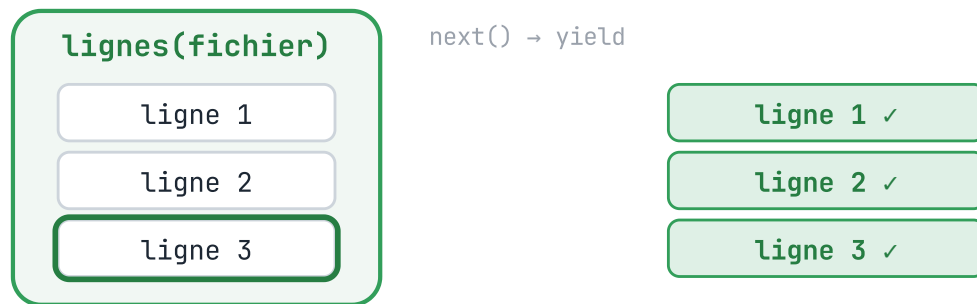
```
# x une liste : toutes les lignes nettoyées d'un coup, en mémoire
def lignes(fichier):
    return [ligne.strip() for ligne in fichier]

# ✓ un générateur : une ligne nettoyée à la fois
def lignes(fichier):
    for ligne in fichier:
        yield ligne.strip()
    # rend une ligne, se met en pause, reprend à la suivante

for l in lignes(mon_fichier):
    # les lignes arrivent une par une, à la demande
    traiter(l)
# → on traite un fichier de 50 Go sans jamais le charger entier en mémoire
```

À chaque `yield`, la fonction rend une ligne puis s'arrête sans libérer ses cases : elle garde sa place dans la boucle. Quand la boucle `for` lui en redemande une (le `next()` du schéma), elle reprend à l'instruction suivante. Piège : un générateur ne se parcourt qu'une fois, et relire le fichier demande d'en créer un nouveau.

un générateur : une valeur, puis pause, puis la suivante



une valeur à la fois. Entre deux, la fonction garde sa place, en pause

Figure 23 : Les valeurs sortent au compte-gouttes, jamais toutes d'un coup : c'est ce qui permet de lire un fichier de 50 Go sans le charger.

Les fonctions des autres

Reste l'usage le plus courant des fonctions : se servir de celles des autres. On n'écrit pas toutes ses recettes soi-même. Le `sum(prix)` de l'introduction, qui additionne des prix, n'est pas de vous : Python est fourni avec des fonctions toutes prêtes (`sum`, `print` et bien d'autres), et vous les appelez comme celles que vous écrivez. Une **bibliothèque** est un lot de fonctions écrites par d'autres, qu'un programme ajoute pour s'en servir sans les réécrire.

La ligne `import random` (« fais venir la bibliothèque random », celle du hasard), placée au début du programme, donne accès à `random.randint(1, 6)` : cette fonction rend un nombre entier au hasard de 1 à 6, bornes comprises, comme un dé.

Une **bibliothèque standard** est le lot de fonctions toutes prêtes fourni avec un langage : `random` fait partie de la bibliothèque standard de Python, et `sum` marche même sans `import`. Toute autre bibliothèque se télécharge et s'installe d'abord, par exemple pour dessiner un graphique, retoucher une photo

ou lire un code-barres. Une **dépendance** est du code que vous n'avez pas écrit mais dont votre programme a besoin pour fonctionner : toute bibliothèque téléchargée en est une, et ses bugs et ses mises à jour deviennent les vôtres.

Si une bibliothèque téléchargée change ou disparaît, votre programme casse avec elle. En 2016, l'auteur d'une bibliothèque JavaScript de onze lignes, `left-pad`, qui complète un texte avec des espaces à gauche, l'a retirée du registre public (le site où se publient les bibliothèques JavaScript) : des milliers de programmes ont cessé de s'installer le jour même, la plupart sans l'avoir jamais nommée, parce qu'une de leurs bibliothèques en dépendait. Une dépendance a ses propres dépendances, et on hérite de toute la chaîne. D'où l'usage de figer les versions exactes employées (le fichier de verrouillage), pour qu'une mise à jour ne casse pas le programme sans prévenir. Le [chapitre 4](#) reprendra cette question de qui dépend de qui, pour votre propre code.

↳ [Fluent Python](#) · [Eloquent JavaScript](#)

2.5 La récursivité : résoudre un problème avec une version plus petite de lui-même

Si une fonction est une recette, la **récursivité** est une recette qui se cite elle-même. « Pour vider un carton : sortez les objets. Si vous trouvez un carton à l'intérieur, appliquez cette recette au carton trouvé. S'il n'y en a pas, vous avez fini. » Une **fonction récursive** est donc une fonction qui s'appelle elle-même. On croit d'abord à une boucle infinie, mais non. Deux temps suffisent :

- le **cas d'arrêt** : le carton qui ne contient plus de carton. On le vide, c'est réglé, sans relancer d'appel ;
- le **cas général** : le carton qui en contient un autre. On ramène le problème à une version plus petite de lui-même.

Comme chaque carton trouvé est plus petit que celui d'où il sort, on retombe toujours sur le cas d'arrêt. Sans lui, on ouvrirait des cartons pour l'éternité, comme deux miroirs face à face.

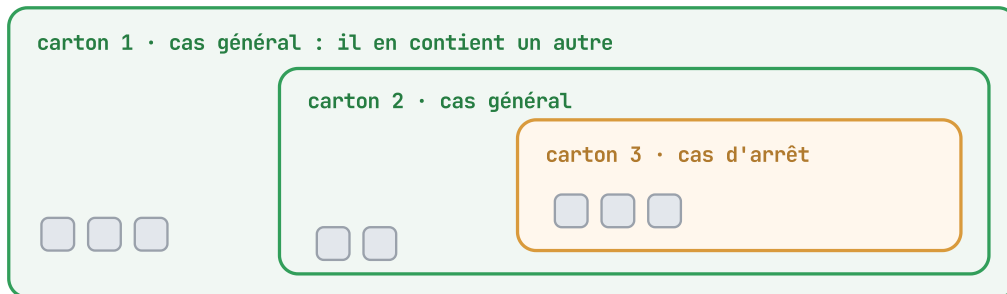


Figure 24 : Chaque carton ouvert en cache un plus petit (vert), jusqu'à celui qui ne contient que des objets : le cas d'arrêt (orange).

Voir la récursivité à l'œuvre : la factorielle

Le secret pour écrire une fonction récursive sans se nouer le cerveau : régler le cas d'arrêt, puis **faire confiance** à l'appel plus petit, en supposant qu'il rend déjà le bon résultat. Vous n'avez pas à imaginer toute la cascade de cartons emboîtés, seulement celui que vous tenez. L'exemple classique est la factorielle, la multiplication de tous les entiers de 1 à n (`fact(4)` vaut $4 \times 3 \times 2 \times 1$). Ce n'est pas le cas d'usage le plus utile, c'est le plus petit où le mécanisme se voit en entier, ici en Python :

```
def fact(n):
    if n <= 1: return 1          # cas d'arrêt
    return n * fact(n - 1)      # réduit à un problème plus petit
# fact(3) = 3 × fact(2) = 3 × 2 × fact(1) = 6
```

Chaque appel de `fact` est un carton : `fact(3)` contient `fact(2)`, qui contient `fact(1)`, le carton sans carton dedans. C'est le cas d'arrêt.

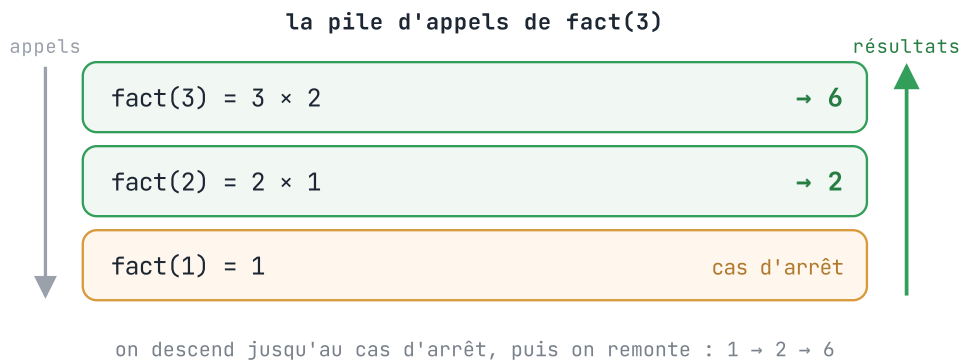


Figure 25 : On descend jusqu'au cas d'arrêt (orange), puis les résultats remontent : 1, 2, 6 (vert).

Ce que la machine retient : la pile d'appels

Sous le capot, chaque appel **attend** le résultat du suivant. Comme on l'a vu en introduction de ce chapitre, appeler une fonction, c'est sauter à ses instructions en mémoire, puis revenir. Chaque appel doit donc noter l'instruction où reprendre et ses propres valeurs (le `n` qui vaut 3, puis 2), et ces notes s'empilent, une par carton ouvert, tant que le plus petit n'est pas vidé : c'est la **pile d'appels**. Comme dans une pile d'assiettes, la dernière note posée est la première reprise : la pile se **dépile** en remontant les résultats.

Cette pile, vous l'avez déjà vue imprimée. Quand un programme s'arrête sur une erreur (2.1), le message est suivi de la liste des appels en cours, un par carton ouvert : c'est la **trace d'appels** (*stack trace*). Python l'écrit du plus ancien au plus récent, JavaScript et Java dans l'autre sens. Reprenons le prix moyen d'un panier vide :

```
Traceback (most recent call last):
  File "caisse.py", line 12, in <module>
    afficher(prix_moyen(panier))
    # l'appelant : il a passé un panier vide
  File "caisse.py", line 7, in prix_moyen
    return total / len(panier)      # la ligne qui a échoué
ZeroDivisionError: division by zero
```

Lire une trace, c'est lire la pile : la ligne qui a échoué est dans l'appel le plus récent, mais la cause est souvent un cran plus haut, chez l'appelant qui a passé une mauvaise valeur. Avant de chercher dans le code, on lit la trace : le message d'abord, la ligne qui a échoué ensuite, puis l'appelant.

D'où une limite bien réelle : une fonction récursive qui s'appelle trop de fois avant d'atteindre le cas d'arrêt fait **déborder la pile** (en anglais *stack overflow*). Python refuse par défaut d'aller au-delà d'environ 1 000 appels emboîtés et signale une `RecursionError`. JavaScript signale lui aussi une erreur, à une limite qui dépend du navigateur. Un cas d'arrêt oublié, ou un appel qui ne rapetisse pas le problème, y mène tout droit.

Le piège des appels répétés

Autre piège, plus discret : les appels qui se répètent. Dans la suite de Fibonacci, chaque nombre est la somme des deux précédents, et la fonction naïve tient en deux lignes (le `if` et le `return` du code ci-dessous, sans la ligne `@cache`). Mais `fib(40)` appelle `fib(38)` deux fois, `fib(37)` trois fois, et plus de 300 millions d'appels en tout : chaque cran de plus multiplie le travail par environ 1,6, une explosion exponentielle, de la famille du $O(2^n)$ de 1.3.

Le remède tient en une ligne, un décorateur de la bibliothèque standard, nommé `cache` comme la mémoire rapide de 1.2. Il garde en mémoire les résultats déjà calculés (on dit la *mémoïsation*) : chaque valeur n'est alors calculée qu'une fois.

```
from functools import cache
@cache
# garde chaque résultat : fib(38) n'est calculé qu'une fois
def fib(n):
    if n < 2: return n          # cas d'arrêt
    return fib(n - 1) + fib(n - 2)
fib(40)    # 102334155, en 41 calculs au lieu de 331 millions d'appels
```

Là où elle excelle : les arbres

La récursivité est l'outil naturel pour tout ce qui est **arborescent** (en forme d'arbre : une branche qui se redivise en branches plus petites) :

- › parcourir un dossier et ses sous-dossiers, version numérique des cartons du début;
- › une page web : son texte HTML (le langage qui décrit la page) est fait de balises emboîtées, un paragraphe dans une section dans la page, que le navigateur range en arbre, le **DOM** (Document Object Model);
- › un JSON imbriqué, des listes dans des listes.

```
def taille(dossier):  
    total = 0  
    for e in dossier:  
        if est_dossier(e): total += taille(e)  
        # un dossier dedans : on s'appelle  
        else: total += e.octets  
        # un fichier : rien à ouvrir  
    return total
```

Quand un dossier contient plusieurs sous-dossiers, la fonction va jusqu'au fond du premier, remonte d'un cran, puis ouvre le suivant : c'est un parcours en profondeur. Le code épouse la **forme des données** : un dossier contient des fichiers et d'autres dossiers, la fonction s'appelle sur ces derniers, et le cas d'arrêt est le dossier qui n'en contient plus. Avec une boucle seule, il faudrait tenir soi-même la liste des dossiers restant à visiter. Une donnée faite de plus petites versions d'elle-même, comme un dossier de dossiers, appelle une fonction qui s'appelle elle-même.

Le prix : chaque appel occupe une place dans la pile, donc pour parcourir une simple liste, une boucle reste plus économe.

↳ [Grokking Algorithms](#) · [Eloquent JavaScript](#)

2.6 La permissivité a un prix

Après la mécanique des appels, une question de confiance : que fait le langage quand on lui donne une valeur du mauvais type ? Certains acceptent presque tout, sans jamais protester : ce sont ceux que la section 2.3 a dits permissifs. Leur prix se paie en silence. JavaScript, l'exemple le plus connu, convertit tout

seul une valeur d'un type à l'autre (texte, nombre, booléen) : c'est la **coercition**. Dès qu'un des deux côtés est du texte, le même `+` qui additionnait colle bout à bout.

Dans le code ci-dessous, `==` et `===` posent tous deux la question « ces deux valeurs sont-elles égales ? » et répondent `true` (vrai) ou `false` (faux). Les deux dernières lignes montrent la coercition à l'œuvre sur un texte, `"5"`.

```
0 == false    // true  (== convertit false en 0, puis compare)
0 === false   // false (=== compare sans convertir : 0 n'est pas false)
"5" - 1       // 4     (- n'a aucun sens sur du texte → "5" devient 5)
"5" + 1       // "6"   (+ voit du texte → 1 devient "1", puis les deux sont collés)
// "51"      (+ voit du texte → 1 devient "1", puis les deux sont collés)
```

Ce `"5"` n'a rien d'artificiel : un champ de formulaire web rend toujours du texte, même quand l'utilisateur y a tapé un nombre. Le client tape 2 dans « quantité », le code en ajoute 1, et le panier affiche 21 articles, sans la moindre erreur.

Le remède tient en un caractère : toujours `===`, qui compare sans rien convertir, jamais `==`, qui convertit d'abord et compare ensuite (seule tolérance courante : `x == null`, vrai quand `x` vaut `null` ou `undefined`, c'est-à-dire n'a pas de valeur).

TypeScript, présenté à la [section 2.3](#), signale comme erreurs `0 == false` et `"5" - 1` avant même que le code tourne. Mais il laisse passer `"5" + 1`, puisque coller du texte reste permis : la parade est de convertir soi-même le champ dès qu'on le lit, avec `Number(champ.value)` (le texte tapé dans le champ, changé en nombre). Connaître la permissivité d'un langage, c'est connaître ses bugs avant de les écrire, et chaque langage permissif a sa propre table de conversions.

↳ [Eloquent JavaScript](#)

2.7 Nommer un concept change ce qu'on peut penser

Le vrai pouvoir d'un langage, c'est de **donner un nom à une idée compliquée**. Une fois qu'on peut dire « un thread », « une promesse », « une interface », on raisonne dessus sans repenser aux rouages en dessous. Trois noms, chacun plus loin de la machine que le précédent : le thread cache le va-et-vient de la puce entre plusieurs tâches, la promesse cache une attente, l'interface cache un type concret. À chaque cran, la machine ne voit rien changer. C'est vous qui pensez plus haut.

Le thread : un fil d'instructions

Premier nom : le thread. Un **thread** (fil d'exécution) est une suite d'instructions que la puce suit l'une après l'autre. Pensez-y comme à un cuisinier, qui prépare un plat à la fois : ce sera notre comparaison. Par défaut, JavaScript fait tourner votre code sur un seul thread : c'est son seul cuisinier.

Votre ordinateur garde des dizaines de programmes ouverts en même temps, mais sa puce n'a que quelques **cœurs**, des unités de calcul dont chacune ne suit qu'un thread à la fois. Le **système d'exploitation** (Windows, Linux ou macOS) est le programme qui gère la machine, un peu comme un chef de cuisine. Pour chaque programme lancé, le système d'exploitation crée un **processus** : le programme en cours d'exécution, avec sa propre zone de mémoire, partagée par tous ses threads. Le système d'exploitation fait passer les threads tour à tour sur les cœurs, si vite qu'ils semblent avancer ensemble.

Deux termes qu'on confond souvent se séparent ici. La **concurrency**, c'est avoir plusieurs tâches en cours, même si une seule avance à un instant donné, comme un cuisinier seul qui, tour à tour, fait avancer chaque commande. Le **parallelisme**, c'est plusieurs tâches qui avancent au même instant, une par cœur, comme deux cuisiniers côte à côte. Par défaut, votre code JavaScript est concurrent sans être parallèle : un seul cuisinier fait avancer les commandes à tour de rôle. Au quotidien, ces rouages restent cachés derrière le mot « thread » : c'est l'intérêt d'un nom.

La promesse : une valeur qui arrivera plus tard

Deuxième nom : la promesse, qui fait trébucher tout débutant JavaScript. Certaines opérations prennent du temps, par exemple aller chercher des données sur un serveur (un ordinateur distant qui répond). JavaScript ne reste pas figé pendant l'attente : il continue, et rappelle votre code quand le résultat arrive. Un code qui continue sans attendre la réponse est dit **asynchrone**.

Comment JavaScript fait-il pour continuer ? Son seul thread, c'est-à-dire son seul cuisinier, prend les tâches, comme des commandes, dans une file d'attente, une par une. La **boucle d'événements** est le mécanisme qui prend sans cesse la tâche suivante dans la file d'attente des tâches (un clic, la réponse d'un serveur, un minuteur : chacun y dépose une tâche). Beaucoup de langages ont la leur, Python et C# par exemple.

Voilà pourquoi un calcul très long fige la page : tant qu'il dure, personne d'autre n'avance dans la file. L'attente du réseau, elle, ne fige rien, parce que le cuisinier ne la fait pas : il confie la requête au navigateur, qui s'occupe du réseau, et il passe à la tâche suivante. Quand le serveur répond, le navigateur ajoute à la file une nouvelle tâche, « traiter la réponse » : elle attend son tour derrière celles qui sont déjà là, puis s'exécute.

Pour récupérer la réponse du serveur, on passe traditionnellement une **fonction de rappel** (en anglais *callback*), une fonction qu'on donne pour qu'elle soit exécutée plus tard : « quand tu auras fini, exécute ceci ». Mais dès qu'une requête (une demande envoyée au serveur) a besoin du résultat de la précédente, on emboîte un callback dans un callback dans un callback : le code part vers la droite et devient illisible. C'est le fameux « callback hell ».

```
// x avec des callbacks : un callback dans un callback dans un callback
// (r1 => ... est une fonction anonyme : sans nom, écrivez là où on s'en
// sert, appelée quand r1 arrive. La flèche se lit « donne »)
get(a, r1 => {
  get(r1, r2 => {
    get(r2, r3 => afficher(r3)); // chaque étape s'enfonce d'un niveau
  });
});
```

Une **promesse** est une valeur qui n'est pas encore là mais qui arrivera plus tard: une fonction comme `get` en renvoie une, au lieu de la valeur elle-même. Le mot-clé `await` veut dire «attends ici que la valeur arrive, puis donne-la-moi comme une variable normale». Seule la fonction qui contient ce `await` se met en pause: pendant ce temps, le cuisinier traite les autres tâches de la file, et la page ne se fige pas.

Cette pause est celle du générateur de [2.4](#): la fonction s'arrête en gardant ses casiers, et reprend à l'instruction suivante quand la valeur est là. Les trois requêtes à la suite se lisent alors de haut en bas, comme du code ordinaire:

```
// dans une fonction marquée async (le mot qui autorise await)
const r1 = await get(a);    // attends le 1er résultat
const r2 = await get(r1);   // puis le 2e, qui dépend du 1er
afficher(await get(r2));    // puis le 3e
```

Ici, chaque requête a besoin de la précédente, d'où les trois `await` à la suite. Si elles étaient indépendantes, les attendre une à une les mettrait bout à bout: trois fois plus long. Mieux vaut les lancer toutes, puis attendre les trois d'un coup avec `Promise.all`, une promesse qui se réalise quand toutes les autres le sont:

```
const p1 = get(a), p2 = get(b), p3 = get(c);
// les trois partent d'un coup : trois promesses
const [r1, r2, r3] = await Promise.all([p1, p2, p3]);
// on attend les trois ensemble
```

Si l'une des trois échoue, `Promise.all` échoue aussitôt, et l'erreur ressort à la ligne du `await`, comme une instruction impossible de [2.1](#) (on verra en [3.4](#) comment l'attraper).

Attention, lire de haut en bas ne veut pas dire que rien d'autre ne s'exécute entre les lignes. Des threads qui partagent la mémoire pourraient modifier la même variable au même instant: avec un seul thread, JavaScript écarte ce

risque. Reste l'**entrelacement**: des tâches qui avancent par morceaux, à tour de rôle. Entre deux `await`, une autre tâche peut modifier la variable, et votre code reprend avec une valeur périmée (même famille de bug qu'en 6.7).

L'interface: la forme des trous

Les deux premiers noms cachent du temps (un fil, une attente). Le troisième cache un type. Troisième nom: l'interface, la liste de ce qu'on exige d'un morceau de code, déjà croisée avec les types (2.3). Une prise électrique en donne l'image: le mur ne dit pas quel appareil viendra, seulement la forme des trous, et tout ce qui a la bonne fiche se branche, lampe ou aspirateur.

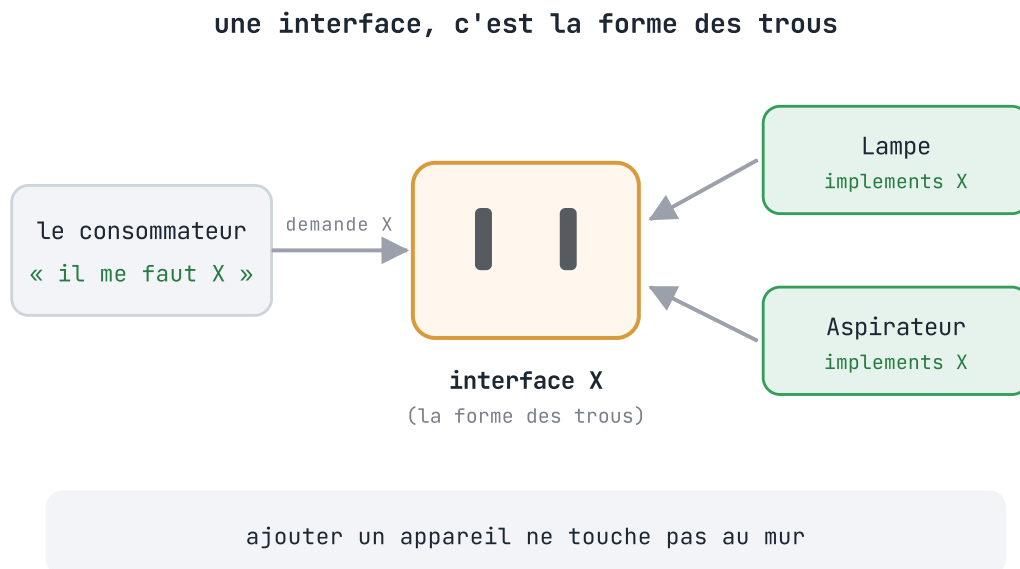


Figure 26 : En orange, l'interface X: le consommateur ne connaît qu'elle. En vert, les appareils qui s'y branchent: lampe, aspirateur ou celui qu'on inventera demain, sans changer une ligne du consommateur.

Ce seul mot change la relation entre deux morceaux de code, l'un qui utilise (le consommateur), l'autre qui fournit (le fournisseur). Le consommateur dit « il me faut quelque chose qui sait faire X » sans connaître le type concret derrière. Le fournisseur, lui, n'a qu'à savoir faire X, et à le dire d'un mot (`implements`) là où le langage l'exige. Ce nom suffit à rendre les deux indépendants: chacun peut changer sans que l'autre bouge. C'est le point de départ du chapitre 4.

Pour la machine, ces trois noms ne changent rien : elle exécute toujours des instructions sur des octets, qu'on parle de thread, de promesse ou d'interface. Seule votre manière d'y penser change. Un langage ne se contente donc pas de traduire votre pensée : il décide de ce que vous pouvez penser sans effort.

↳ [Learning Go](#) · [Eloquent JavaScript](#)

2.8 Trois façons d'écrire un programme

Un dernier mot, que vous croiserez dans les offres d'emploi. Un **paradigme** est une façon de penser et d'écrire un programme. Le chapitre en a pratiqué trois sans les nommer :

- › la **programmation procédurale** écrit le programme comme une suite d'instructions suivies dans l'ordre, regroupées en fonctions ;
- › la **programmation orientée objet** (POO) regroupe, dans un même objet, des données et les actions qu'on peut lui demander, comme l'article de la caisse (2.2, puis tout le [chapitre 4](#)) ;
- › la **programmation fonctionnelle** fait voyager les fonctions comme des données (2.4) et veut qu'elles ne modifient jamais ce qu'elles reçoivent : au lieu de changer la liste reçue, une fonction en rend une nouvelle, avec le changement fait. Le piège de 2.2, où une liste modifiée sous un nom change aussi sous l'autre, disparaît.

Le style fonctionnel repose sur deux mots. Un **effet de bord** est tout ce qu'une fonction change hors de son résultat : un texte affiché, une variable globale, une liste reçue qu'elle modifie sur place. Une **fonction pure** ne fait qu'une chose : calculer son résultat à partir de ses arguments, sans effet de bord et sans rien lire de caché.

En JavaScript, `noms.sort()` trie la liste sur place, alors que `noms.toSorted()` rend une liste triée et laisse l'original intact : même travail, l'un avec effet de bord, l'autre pur. Une fonction pure est aussi plus facile à tester : mêmes arguments, même résultat, donc on compare ce qu'elle rend à

ce qu'on attend, comme `prixTTC(100)` qui doit rendre 120. Aucun programme utile n'est sans effet de bord (afficher, enregistrer, envoyer) : on les range en bordure, et on garde un cœur pur.

PHP, JavaScript et Python permettent de mélanger les trois façons. Java est d'abord orienté objet, le style fonctionnel n'y étant arrivé qu'avec Java 8, et le C est surtout procédural. On choisit, pour chaque problème, celle qui l'énonce le plus simplement : un panier se modélise en objets, une liste de prix à transformer s'écrit en fonctionnel.

Robert Martin, auteur de *Clean Architecture*, un livre sur la conception des logiciels, fait une remarque surprenante : aucun paradigme n'a ajouté de pouvoir, chacun en a retiré un. La procédurale, avec les trois gestes de 2.1 (Martin parle de programmation structurée), retire le `goto`, le saut vers n'importe où. L'objet retire le choix de la fonction à appeler : au lieu de le faire soi-même, on demande à l'objet, avec `forme.dessiner()`, et c'est lui, cercle ou carré, qui sait comment. Le fonctionnel retire l'affectation, le « mets dans » : une donnée ne change plus, on en fabrique une autre.

Chaque retrait est un gain, parce que ce qu'on ne peut plus faire, on n'a plus à le surveiller. Un paradigme est une discipline : il vous prive d'une liberté pour que le programme reste lisible. Il y a pourtant une limite : on retire tant que le programme peut encore faire ce qu'on lui demande. Cette limite a un nom, la règle du moindre pouvoir, que Tim Berners-Lee, l'inventeur du web, a érigée en principe : pour un même résultat, mieux vaut l'outil qui peut le moins. C'est pourquoi une page se décrit en HTML, qu'on ne peut que lire, plutôt qu'en JavaScript, qu'il faut exécuter.

Un langage inconnu se découvre par ses pièges avant ses bugs. Cinq questions suffisent à les trouver :

- › que recopie-t-il quand on copie une variable, la valeur ou l'adresse (2.2) ?
- › jusqu'où voit un nom (2.1) ?
- › que fait-il d'une valeur du mauvais type, refuse-t-il ou convertit-il en silence (2.3, 2.6) ?

- › combien de cuisiniers travaillent ensemble, et partagent-ils la mémoire (2.7) ?
- › que devient un entier quand il est plein (1.1) ?

Le chapitre en a donné les réponses pour quelques langages. Les poser à un autre, c'est le connaître en une heure.

↳ Clean Architecture

↳ CE QUI MÈNE AU CHAPITRE 3

Trois gestes (enchaîner, répéter, choisir) suffisent à tout calculer. Le chapitre leur a donné des mots : des casiers qui tiennent une donnée ou une adresse, des types qui bornent ce qu'ils acceptent (sauf si le langage convertit en silence), des fonctions qui se transmettent comme des nombres ou s'appellent elles-mêmes, des noms qui cachent les rouages, et trois façons d'écrire un programme. Chaque chose nommée agrandit ce qu'on peut penser.

*On sait maintenant tout exprimer. Mais connaître tous les mots d'une langue n'a jamais fait un bon écrivain : une phrase peut être grammaticalement parfaite et rester illisible, comme ces courriers administratifs qu'on relit trois fois. Un programme aussi peut être exact et rester un casse-tête. Le prochain enjeu n'est plus le pouvoir du langage mais la **clarté**.*

*« Ce que l'on conçoit bien s'énonce clairement, et les mots pour le dire arrivent aisément. » — Boileau, *L'Art poétique*, 1674*

3 Écrire pour les humains

la propreté

Le code est lu bien plus souvent qu'il n'est écrit : Robert Martin, dans *Clean Code*, compte plus de dix minutes de lecture pour une minute d'écriture.

La machine, elle, se moque de l'élégance : un nom à une lettre comme `x` s'exécute aussi vite qu'un nom parlant comme `prixTTC`. La propreté du code, c'est-à-dire sa clarté, n'est donc pas pour la machine : c'est de la **communication** avec le prochain humain qui ouvrira le fichier.

Ce prochain humain, c'est souvent vous. Une recette griffonnée pour soi marche très bien, jusqu'au jour où quelqu'un d'autre doit la suivre, ou qu'on la relit six mois plus tard. Cette propreté se construit en six habitudes : bien nommer, bâtir des fonctions profondes, commenter le pourquoi, faire disparaître l'erreur, ne pas se répéter, et retoucher un code sans changer ce qu'il fait.



Figure 27 : Écrit en été, relu six mois plus tard : on ne reconnaît plus son propre code.

3.1 Le nom révèle l'intention

Le nom est le premier signe de clarté, et quand il ne dit pas assez, on est tenté de l'expliquer à côté. Un **commentaire** est une note écrite pour les humains au milieu du code, après `//` ou `#` selon le langage, comme les lignes grises des exemples de ce livre : la machine l'ignore. Mais un commentaire qui ne fait que traduire un nom obscur répare au mauvais endroit : c'est le nom qu'il faut corriger. Selon Robert Martin dans *Clean Code*, un bon nom répond à trois questions d'un coup :

- › **pourquoi** ça existe ;
- › **ce que** ça fait ;
- › **comment** on s'en sert.

Comparez la condition brute et la même condition rangée sous un nom :

```
// x à décoder à chaque lecture : majeur (>= 18), ET (&&) solde positif,  
// ET pas (!) suspendu  
if (age >= 18 && solde > 0 && !suspendu) { ... }  
  
// ✓ la règle, écrite une seule fois, sous un nom  
function peutPasserCommande(client) {  
    return client.age >= 18 && client.solde > 0 && !client.suspendu;  
}  
  
// ✓ partout ailleurs, le nom EST l'explication  
if (peutPasserCommande(client)) { ... }
```

Lu à voix haute, `peutPasserCommande` répond aux trois questions. Pourquoi il existe : bloquer une commande interdite avant qu'elle parte. Ce qu'il fait : dire si ce client a le droit de commander. Comment s'en servir : le début `peut...` annonce une réponse oui ou non, à placer dans un `if`.

Quand on code, le gain se voit le jour où la règle change. Si un client doit aussi avoir une adresse e-mail vérifiée, on corrige `peutPasserCommande`, et chaque `if` qui l'utilise suit sans qu'on y touche. Le pouvoir de nommer de la [section 2.7](#) redescend ici à l'échelle d'une seule ligne : le code dit « passer commande », dans les mots de l'activité, et non plus « 18, solde, suspendu ».

Un détail cloche encore à l'intérieur de la fonction : le 18. Un **nombre magique** est un nombre écrit tel quel dans le code, sans rien qui dise ce qu'il représente. Pourquoi 18 ? C'est la majorité, mais il faut le deviner. Et le jour où la boutique s'ouvre dans un pays où l'on est majeur à 21 ans, il faut retrouver tous les 18 du programme, sans toucher à ceux qui comptent autre chose.

Le remède : donner un nom à ce nombre. Une **constante** est une variable qu'on remplit une fois et qu'on ne réécrit jamais, souvent écrite en majuscules pour qu'on la reconnaisse d'un coup d'œil. En JavaScript, le mot `const` interdit de réécrire la variable : on écrit `const AGE_MAJORITE = 18`, puis `client.age >= AGE_MAJORITE`, et la ligne se lit sans deviner. La caisse de 2.1 cachait deux nombres magiques, le seuil de 100 € et le coefficient 0,9 (10 % de remise), qui méritent chacun une constante. Un texte écrit tel quel pose le même problème : une faute de frappe sur `"payee"` passe sans bruit, et le remède est le même, une constante, ou mieux un type qui n'admet que ces mots (2.3).

La constante soigne la lecture, pas la règle. La majorité est une règle de l'activité que sert le programme, et elle change selon le pays : 18 ans en France, 21 à Singapour. Dès que le programme sert plusieurs pays, l'âge n'est plus un nombre gravé dans le code mais une donnée, rangée dans la base de données, pays par pays, que `peutPasserCommande` va chercher. Le nom reste, la valeur a déménagé.

Le nom doit aussi survivre aux changements du code. Prenons une fonction `trouverClient`, qui cherche un client dans la base de données et le renvoie. Un jour, on lui fait créer le client quand elle ne le trouve pas. Son nom ment alors : le code qui l'appelle croit lire, alors que la fonction écrit. Un nom qui ment coûte plus cher qu'un nom vague, parce qu'on le croit sans vérifier. On le renomme dans la foulée, `trouverOuCreerClient`, et chaque ligne qui l'appelle se relit d'un œil neuf. Mieux encore, on sépare la lecture (`trouverClient`) de l'écriture (`creerClient`) : c'est la séparation commande-requête de Bertrand Meyer, selon laquelle une fonction qui répond à une question ne modifie rien.

Un nom qui survit au changement ne suffit pas encore. Robert Martin demande aussi qu'il tienne à l'échelle du programme entier. Si le code dit ici `trouverClient`, là `chercherClient` et ailleurs `recupererClient` pour la même opération, le lecteur croit à trois opérations différentes et cherche une nuance qui n'existe pas. **Un mot par idée, partout**: un seul verbe pour «aller chercher», un seul nom pour «client», et les mêmes d'un fichier à l'autre. Et si ces mots sont aussi ceux des gens de l'activité (vendeurs, comptables), tout le monde parle la même langue, ce qu'Eric Evans appelle l'*Ubiquitous Language* (4.5).

Une partie de ces règles ne demande aucun jugement, et on la confie à des outils. Le **formateur** réécrit les espaces, les retours à la ligne et l'indentation pour que tout le projet ait la même allure, quel qu'en soit l'auteur. Le **linter** relit le code sans l'exécuter et signale ce qu'une machine sait voir: une variable jamais lue, un nom qui ne suit pas la convention du projet, un `==` là où il fallait `===` (2.6). On ne débat plus de la forme en relecture, et l'attention reste pour le sens, les noms et les règles qu'aucun outil ne vérifie.

↳ [Coder proprement](#)

3.2 La fonction profonde, pas seulement courte

Un bon nom cache une règle derrière un mot. Reste à savoir jusqu'où découper le code en fonctions, et là deux maîtres s'affrontent :

- › Clean Code (Robert Martin) : les fonctions doivent être petites, puis plus petites encore ;
- › A Philosophy of Software Design (John Ousterhout) : passé un point, découper encore n'aide plus personne.

La vraie question n'est pas « combien de lignes ? », mais « est-ce plus simple pour l'appelant ? », c'est-à-dire pour le code qui se sert de votre fonction. Ce code appelant, c'est souvent vous qui l'écrirez, dans trois mois, quand vous aurez oublié ce que la fonction cache. Il faut donc pouvoir s'en servir sans rien savoir de son intérieur: au restaurant, vous commandez « un tiramisu », pas « des œufs, du sucre, du mascarpone, et je monterai le tout moi-même ».

Prenons le calcul d'un prix, en quatre étapes : lire le prix hors taxes (HT), retirer la remise, ajouter la TVA, ajouter les frais de port.

```
// x morceaux exposés : l'appelant doit connaître les 4 étapes et leur
// ordre
const ht          = lirePrix(commande);
const apresRemise = retirerRemise(ht, commande);
const avecTVA     = appliquerTVA(apresRemise);
const total       = ajouterPort(avecTVA, commande);

// ✓ une fonction "profonde" : un seul appel, les 4 étapes cachées dedans
const total = prixFinal(commande);
```

Le découpage en lui-même n'est pas le problème : `prixFinal` peut très bien appeler ces quatre étapes en interne, comme un cuisinier enchaîne les étapes d'un plat. Mais ces quatre étapes deviennent privées : une fonction **privée** est réservée au code qui la contient, et le reste du programme ne peut pas l'appeler. Il ne voit que `prixFinal`, comme le client ne voit que le plat, jamais la cuisine. Selon le langage, la protection vient d'un mot-clé (`private` en Java) ou d'une simple convention (un tiret bas devant le nom en Python).

L'appelant n'a plus qu'un nom à connaître au lieu d'enchaîner les étapes lui-même. Le jour où la TVA change, on ne touche qu'à l'intérieur de `prixFinal` : c'est le principe de dissimulation d'information (information hiding), posé par David Parnas en 1972.

La fonction **profonde** cache beaucoup de travail derrière une toute petite **surface**, c'est-à-dire ce que l'appelant doit connaître pour s'en servir (ici un nom et un argument) : une ligne sur la carte, tout le reste en cuisine. **Découper, oui. Exposer les morceaux, non.**

Découper a pourtant une limite. Une fonction qui ne cache presque rien est dite **peu profonde** : `appliquerTVA(prix)`, si elle se résume à `prix * 1.2`, ne simplifie rien. L'appelant aurait pu écrire le calcul lui-même, et il a un nom de plus à retenir.

Parfois, découper devient même contre-productif : quand deux étapes sont si soudées qu'on ne comprend pas l'une sans l'autre, le lecteur doit sauter de l'une à l'autre pour suivre le calcul, et le code est plus difficile à lire qu'avant. Deux

fonctions qu'on ne peut lire qu'ensemble sont des **fonctions siamoises** (*conjoined methods* dans *A Philosophy of Software Design*), et mieux vaut les fusionner en une seule.

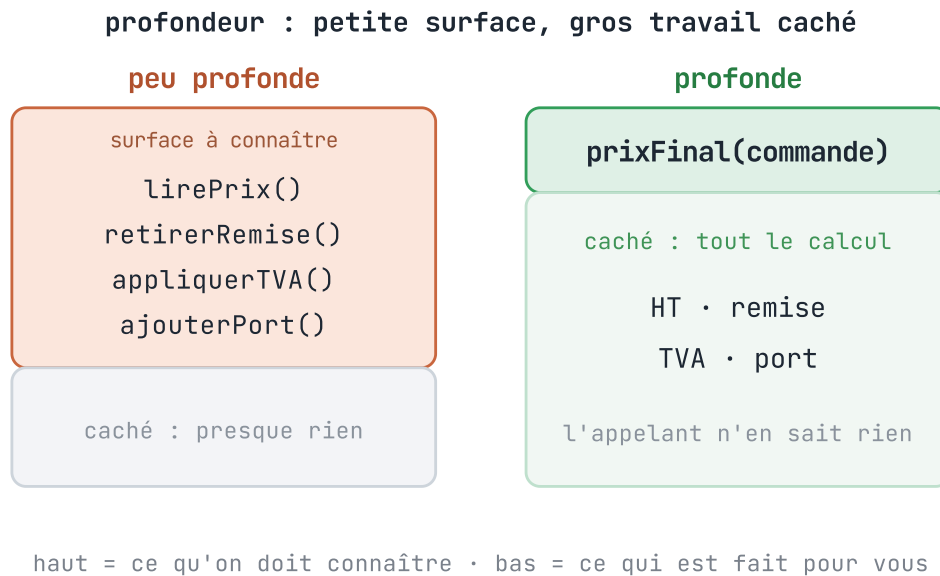


Figure 28 : En couleur vive, ce que l'appelant doit connaître. En pâle, ce qui lui est caché. La fonction profonde expose un seul nom (`prixFinal(commande)`), la peu profonde ses quatre étapes.

L'exemple le plus parlant d'Ousterhout dépasse la fonction : les fichiers d'Unix, le système dont Linux et macOS reprennent les appels. Pour travailler sur un fichier, le programmeur n'a que cinq appels à connaître : ouvrir (`open`), lire (`read`), écrire (`write`), se déplacer dans le fichier (`lseek`), fermer (`close`). Derrière, des centaines de milliers de lignes gèrent les disques, les droits d'accès et le matériel, sans qu'il en voie rien. C'est la fonction profonde à grande échelle : cinq mots sur la carte, une cuisine entière. Ousterhout appelle cette surface une « interface ».

↳ [Coder proprement](#) · [A Philosophy of Software Design](#)

3.3 Le commentaire dit le pourquoi, pas le quoi

Le nom est choisi, la fonction découpée : reste le commentaire, la note écrite à côté du code. Un bon nom dit déjà ce que fait le code. Faut-il alors se passer de commentaires ?

Sur les commentaires, *Clean Code* est cinglant : Robert Martin va jusqu'à écrire que **les commentaires sont toujours des échecs**, la preuve qu'on n'a pas su s'exprimer dans le code. Un commentaire qui ne fait que **paraphraser** le code, c'est-à-dire redire ce qu'il fait, vieillit et finit par mentir : le code change et le commentaire non. Sur ce cas, Martin a raison.

Les commentaires sont comme les notes dans la marge d'une recette : « ne pas ouvrir le four avant 20 minutes » sauve le gâteau, « mélanger les œufs » écrit à côté de la ligne qui dit de mélanger les œufs ne sert à rien.

Dans le code, c'est pareil. La première note ci-dessous répète sa ligne. La seconde dit ce qu'aucune ligne ne peut dire : pourquoi le `timeout`, le délai au-delà duquel on cesse d'attendre une réponse, vaut 29 000 millisecondes, soit 29 secondes (écrit `29_000`, le tiret bas ne sert qu'à la lisibilité) et pas 30 000.

```
user.deactivate();  
// x désactive l'utilisateur          (le code le dit déjà)  
timeout = 29_000;  
// ✓ 29 s : juste sous les 30 s du load balancer, qui coupe la connexion  
// au-delà
```

La raison n'est écrite nulle part dans le code : un serveur placé devant le nôtre, le *load balancer* (celui qui répartit les demandes des visiteurs), coupe toute connexion sans réponse au bout de 30 secondes. Notre code abandonne donc une seconde plus tôt, pour afficher sa propre erreur claire au lieu d'une coupure sans explication. Seul un commentaire peut le dire.

Martin lui-même ne bannit pas tous les commentaires : il garde ceux qui expliquent une intention ou préviennent d'un danger. A *Philosophy of Software Design*, de John Ousterhout, conteste l'idée que le bon code se passe de commentaires et pose une règle : le bon commentaire dit ce que le code ne peut pas dire. Trois genres méritent qu'on les écrive :

- › **le pourquoi** : la décision non évidente, le compromis derrière le code.

```
// en centimes : zéro arrondi flottant
```

- › **l'avertissement** : le piège, l'ordre qu'on ne doit pas casser.

```
// NE PAS réordonner : valider avant d'enregistrer
```

- › **le contrat** : ce qu'une fonction garantit à son appelant et ce qu'elle exige de lui, pour qu'il n'ait pas à lire les lignes qu'elle contient. Les types en disent déjà une partie, gratuitement : ce qu'elle attend et ce qu'elle rend ([section 2.3](#)). Le commentaire ne prend le relais que pour le reste, ce que les types ne savent pas dire. `// écrit dans la base de données, suppose la liste triée`

Un pourquoi, pourtant, n'a pas sa place dans un commentaire : celui du changement. Pourquoi cette ligne a-t-elle été modifiée mardi ? La réponse raconte l'histoire du code, pas son état, et chaque retouche ajouterait une ligne au commentaire. Cette histoire a son propre carnet : le message de commit, la note qu'on écrit pour décrire un changement quand on l'enregistre dans Git. Git est l'outil qui garde l'historique du code. Le livre le présente en [6.3](#), il suffit ici de savoir qu'il existe. La commande `git blame` affiche en face de chaque ligne le message de son dernier changement.

Le commentaire sert même de détecteur. Quand il devient long et pénible à écrire, c'est « le canari dans la mine », l'oiseau que les mineurs emportaient parce qu'il tombait au premier gaz, avant eux. Un commentaire pénible à écrire signale une fonction mal découpée : sa surface, ce que l'appelant doit connaître, est trop compliquée pour se dire simplement ([3.2](#)). Ousterhout va jusqu'à écrire ses commentaires avant le code, comme outil de conception : si l'explication ne vient pas, c'est la conception qu'il faut revoir, pas la phrase.

↳ [Coder proprement](#) · [A Philosophy of Software Design](#)

3.4 Faire disparaître l'erreur

Un commentaire peut avertir d'un piège. Mieux vaut qu'il n'y ait pas de piège. Une carte SIM a un coin coupé : elle ne rentre que dans un sens, donc personne n'a besoin d'être prévenu, et personne ne se trompe. Une forme qui rend le mauvais geste impossible s'appelle un **détrompeur** (*poka-yoke* en japonais). C'est une idée sous-estimée : la meilleure façon de gérer une erreur, c'est qu'elle ne puisse pas arriver.

Le code peut faire pareil, mais il fait souvent l'inverse. Prenons l'opération «prends les 10 premiers caractères» d'un texte. Que répondre pour «hi», qui n'en a que 2 ? Java s'arrête sur une erreur. Python change la définition de l'opération en «prends jusqu'à 10 caractères» : le cas qui faisait exploser n'est plus une erreur, c'est un cas normal. Voici les deux (l'exemple Java est celui de John Ousterhout, le Python est ajouté ici) :

```
// x Java : s'arrête sur une erreur, à prévenir par une vérification avant
// chaque appel
"hi".substring(0, 10);
// 🌟 StringIndexOutOfBoundsException (position hors limites)

# ✓ Python : la tranche [0:10] (les positions 0 à 9) se borne à ce qui
# existe, sans erreur
"hi"[0:10]           # → "hi"
"hi"[10]
# 🌟 IndexError : une position seule n'a pas de réponse juste
```

Cette erreur de Java s'appelle une exception. Quand une fonction n'y arrive pas, beaucoup de langages l'obligent à **lever une exception** : elle s'arrête net et envoie un signal d'erreur, l'exception, à son appelant. L'appelant doit alors l'attraper et décider quoi faire (c'est le rôle d'un bloc `try` / `catch` en Java, `try` / `except` en Python). S'il ne le fait pas, l'exécution en cours s'arrête : c'était le cas du `ZeroDivisionError` de 2.1, une exception que personne n'avait prévue. Chaque appelant prudent de `substring` doit donc vérifier la longueur avant d'appeler, et c'est ce coût, répété partout, que la version Python supprime.

Cette idée vaut aussi pour votre propre code, pas seulement pour ce que les langages fournissent. Une recherche de clients par nom qui ne trouve personne n'a pas besoin de lever une exception : une liste vide est une réponse juste, et la boucle de l'appelant fait simplement zéro tour.

```
// x aucun client de ce nom ? exception, et chaque appelant doit vérifier
// avant d'appeler
si existeClient("Dupont") :
// à répéter devant chaque appel
    clients = chercherClients("Dupont") // sinon ✨ AucunClient

// ✓ aucun client ? une liste vide : la boucle fait zéro tour, le cas
// d'erreur a disparu
clients = chercherClients("Dupont") // → []
pour chaque client dans clients :
    afficher(client.nom)
```

La même idée vaut pour un visiteur non connecté. Plutôt que de renvoyer « rien » (le `null` de 2.3), qu'il faut tester à chaque appel, on renvoie un utilisateur « invité » sans aucun droit : plus aucun appelant n'a de cas spécial à tester. Mais l'idée s'arrête là où l'absence est anormale : si le compte d'un client qui vient de commander est introuvable, répondre « invité » cacherait un bug.

Attention : on ne fait disparaître une erreur que si le cas a une réponse juste. Jusqu'à 10 caractères de « hi », c'est « hi ». Une recherche sans résultat, c'est une liste vide. Mais la 11e lettre de « hi » n'existe pas, et le prix moyen d'un panier vide (0 € divisé par 0 article, le calcul de 2.1) n'a pas de réponse non plus : là, l'erreur doit rester, et faire du bruit.

Ce bruit a un nom. **Échouer tôt** (*fail fast*, formule popularisée par Jim Shore en 2004) est la règle qui veut qu'un programme s'arrête dès qu'il rencontre un cas sans réponse juste, plutôt que de continuer avec une valeur fausse. L'erreur éclate alors au plus près de sa cause, et non trois écrans plus loin.

Échouer tôt a un outil, l'**assertion** : une ligne qui vérifie ce qui doit toujours être vrai, et arrête tout sinon, comme `assert total >= 0`. Un **invariant** est ce qui doit rester vrai quoi que fasse le programme : un total de caisse n'est jamais négatif, une commande payée a un montant. L'assertion ne vérifie pas une entrée, qui prévoit un cas possible comme la liste vide de tout à l'heure :

elle crie si l'impossible arrive, donc si le code a un bug. Le contrat d'une fonction (3.3) tient en trois lignes de ce genre : ce qu'elle exige à l'entrée, ce qu'elle garantit à la sortie, ce qu'elle laisse toujours vrai entre les deux.

```
total = prixHT - remise
assert total >= 0, "total négatif : bug de calcul"
// l'invariant, vérifié ici
afficher(total)
// sans l'assert, -12 € irait jusqu'au ticket
```

Faire disparaître l'erreur et échouer tôt visent deux erreurs différentes. L'erreur venue du monde extérieur (un serveur muet, un fichier absent) échappe à notre contrôle et arrivera de toute façon : on la prévoit, et quand le cas a une réponse juste, on la fait disparaître. L'erreur venue du code lui-même est un bug, comme la moyenne d'un panier vide : on veut l'entendre tout de suite, avant qu'un total faux ait voyagé jusqu'au ticket de caisse.

Quand l'erreur reste : la porter, pas la taire

Une exception qui reste a besoin de deux choses, un contexte et une destination. Le contexte, c'est ce que le message emporte en remontant : « fichier introuvable » ne dit rien, « config.json introuvable dans /etc/caisse » se corrige en une minute. Chaque étage ajoute ce qu'il sait et qu'en bas personne ne savait. La destination, c'est l'endroit où l'on attrape : le code métier lève et laisse remonter, et l'on attrape aux frontières, là où le programme touche le dehors (2.1) : l'écran, qui traduit en message pour l'humain, l'API, qui répond un code d'erreur, le journal, qui garde tout. Entre les deux, une attrape vide est la pire ligne d'un programme : l'erreur disparaît sans avoir reçu de réponse juste, l'exact contraire de ce que cette section appelle faire disparaître l'erreur.

Go et Rust ont fait un autre choix : la fonction renvoie son résultat ou son erreur, comme deux valeurs, et l'appelant doit regarder laquelle il a reçue, si bien que l'oubli se voit dans le code au lieu de se produire à l'exécution. Enfin, une erreur qui remonte saute des lignes, et parmi elles celle qui refermait le fichier ou rendait la connexion. Le ramasse-miettes (2.4) ne libère que la mé-

moire, pas un fichier ouvert, une connexion à la base ou un verrou : le bloc qui s'exécute quoi qu'il arrive, `finally` en Java, `with` en Python, `defer` en Go, est là pour ça.

Faire disparaître l'erreur vient de John Ousterhout, qui la range sous un principe plus large : « tirer la complexité vers le bas ». Il s'agit d'absorber la difficulté à l'intérieur de la fonction, comme `prixFinal` cachait les quatre étapes du calcul, plutôt que de la repousser sur ceux qui l'appellent (3.2). Mieux vaut que l'auteur de la fonction peine une fois, dedans, que mille appelants, dehors.

↳ [A Philosophy of Software Design](#)

3.5 Ne te répète pas, et facilite le changement

Le taux de TVA est écrit à trois endroits du programme. Le jour où il change, on en corrige deux et on oublie le troisième : chaque copie vieillit à son rythme. **DRY** (« Don't Repeat Yourself », ne te répète pas) est le principe qui veut qu'un même savoir, comme le taux de TVA, n'existe qu'à un seul endroit du programme. C'est l'un des plus mal compris : il ne dit pas « ne copie-colle pas du code ». Ses auteurs, Andrew Hunt et David Thomas, le tiennent en une phrase : « **chaque savoir doit avoir une représentation unique, non ambiguë, qui fait autorité dans le système** ». Le mot qui compte est savoir, pas code, et deux conséquences contre-intuitives en découlent.

Du code identique n'est pas toujours une duplication. Si deux fragments se ressemblent par hasard et évoluent pour des raisons sans rapport, les fusionner les couple à tort : les voilà attachés l'un à l'autre, et toucher à l'un oblige désormais à se soucier de l'autre.

Le prix d'un article se valide avec `prix > 0`, sa quantité en stock avec `quantite > 0` : on est tenté de tout réunir dans un seul `estPositif()`. Mais demain le stock devra accepter 0 (rupture de stock), pas le prix : c'étaient deux règles, identiques par coïncidence.

La programmeuse Sandi Metz en tire une maxime, dans son article *The Wrong Abstraction* (2016) : « la duplication coûte bien moins cher que la mauvaise abstraction ». Une abstraction, c'est un regroupement de cas sous un seul nom, ici `estPositif()`. Avec deux copies, le jour où le stock doit accepter 0, on modifie la copie du stock, et c'est fini. Avec un seul `estPositif()`, on ne peut pas le changer sans casser le prix : on ajoute donc un cas particulier (« si c'est le stock, accepter 0 »), puis un autre pour le cas suivant, et la fonction devient un enchevêtrement que personne n'ose plus toucher. Dans le doute, gardez la copie.

On se répète aussi sans copier une seule ligne. Le même savoir se retrouve à deux endroits, de trois façons courantes. Premier cas : un nombre écrit dans le code et repris dans un texte. Le code accorde la livraison gratuite dès 100 €, et la bannière du site l'annonce aussi : le jour où le seuil passe à 120 €, on pense au code et on oublie la bannière. La bannière doit lire le seuil dans le code au lieu de le retaper. Deuxième cas : la forme d'une fiche de la base de données (nom, prix, stock), écrite une fois dans la base et une seconde fois dans le code. La parade est de générer l'une à partir de l'autre. Troisième cas : un commentaire qui redit ce que le code fait déjà, exactement le piège du commentaire-paraphrase vu en 3.3. Aucun copier-coller, et pourtant deux vérités à garder synchronisées.

Quelle que soit la forme de la répétition, la parade est la même : chaque savoir a un seul endroit qui fait foi. Tout le reste va l'y chercher, au lieu d'en garder sa propre copie dans son coin.

DRY se range sous une idée plus large. **ETC** (*Easier To Change*, plus facile à changer) veut qu'à chaque choix, on se pose une seule question : « est-ce que ça rendra le système plus facile, ou plus dur, à modifier ? ». C'est une boussole plus qu'une règle. Prenons un programme qui envoie des alertes par e-mail ou

par SMS (message texte du téléphone). Chaque moyen d'envoi est un canal, et le morceau de code qui sait envoyer par un canal, un notifieur. Demain, on voudra un troisième canal : comment l'ajouter sans rouvrir tout le code ?

```
// x chaque nouveau canal rouvre la fonction
if (canal == "email") ... else if (canal == "sms") ...

// ✓ un dictionnaire {canal → notifieur} : un nouveau canal = une entrée,
// la logique ne bouge pas
notifieurs[canal].envoyer(message)
```

Cette façon de faire porte un nom, Stratégie : chaque notifieur est une manière interchangeable de faire la même chose, et le dictionnaire choisit laquelle. On y revient en 4.2, avec d'autres solutions éprouvées du même genre. Les bons noms, les fonctions profondes, DRY lui-même sont autant de cas particuliers d'ETC : chacun rend le code plus facile à changer. Reste le code déjà écrit : comment le rendre plus facile à changer sans le casser ?

↳ [The Pragmatic Programmer](#)

3.6 Refactorer, c'est changer la forme sans changer le comportement

Refactorer, c'est réécrire une phrase pour qu'elle se lise mieux sans changer ce qu'elle dit. En code : changer la structure interne sans changer le comportement observable, c'est-à-dire ce que voit quiconque utilise le code sans connaître son intérieur (les résultats renvoyés, ce qui s'affiche, ce qui s'enregistre). La preuve est un test, un petit programme qui appelle le code et vérifie sa réponse : s'il passait avant, il passe encore après. Sinon ce n'est plus un refactoring, c'est une modification.

On avance par petits pas sûrs : renommer une variable, sortir trois lignes dans leur propre fonction, vérifier, recommencer. Des « odeurs » guident ces pas, ces signes qui trahissent un code mal structuré. L'une des plus courantes, le *Data Clump* : un groupe d'arguments qui voyagent toujours ensemble, comme

la rue, la ville et le code postal passés l'un après l'autre à chaque fonction. Le remède les réunit en un seul objet, `adresse`, ce que le catalogue de Refactoring nomme *Introduce Parameter Object*.

```
expedier(rue, ville, codePostal, pays) // x 4 arguments collés partout
expedier(adresse)
// ✓ ils formaient un concept : une Adresse
```

Quatre mots qui ne se quittent jamais finissent par en appeler un seul : la langue l'a fait bien avant nous en inventant le mot *adresse*. Le code ne fait que rattraper son retard.

Le programmeur Kent Beck résume le principe clé du refactoring en une phrase : **faciliter le changement, puis faire ce changement devenu facile**. C'est la boussole ETC mise en deux temps. C'est contre-intuitif, parce qu'on commence par ne rien livrer de ce qui était demandé, et Beck prévient que ce premier temps peut être le plus dur. Vous devez brancher le paiement PayPal, alors que le code ne connaît que la carte bancaire ? D'abord vous refactorisez le `if (moyen == "carte")` en dictionnaire {moyen → payeur} (le payeur est le morceau de code qui sait encaisser par ce moyen), comme les canaux de notification de 3.5, sans toucher au comportement. Ensuite vous ajoutez PayPal presque gratuitement.

YAGNI (« You Aren't Gonna Need It », tu n'en auras pas besoin) est le contrepoids de DRY et d'ETC : la règle, venue de l'équipe de Kent Beck, qui déconseille de construire d'avance ce que personne n'a encore demandé. Tant qu'il n'y a que la carte bancaire, un dictionnaire de moyens de paiement est de la sur-ingénierie : du code compliqué par précaution, à lire et à tester, pour un besoin qui souvent ne viendra jamais, ou viendra autrement. On refactorise quand le besoin arrive, pas avant.

Martin Fowler, l'auteur de *Refactoring*, pousse la patience plus loin. La **règle de trois**, que Fowler tient de Don Roberts, tient en trois temps : la première fois on écrit, la deuxième on copie en grimaçant, la troisième on refactorise. Pourquoi attendre ? Deux cas qui se ressemblent n'ont pas forcément la même raison d'être, comme le prix et le stock de 3.5, validés de la même façon mais sou-

mis à des règles différentes. La carte bancaire et PayPal, eux, servent le même but évident, encaisser : ici deux suffisent. Dès que le point commun est certain, on peut regrouper, à deux cas comme à trois.

On ne refactorise que ce qu'on a compris, et le code à retoucher est presque toujours celui d'un autre, ou le vôtre d'il y a six mois, ce qui revient au même. Prenez la caisse dont on vous demande de changer le taux de TVA, et que vous n'avez pas écrite. Un code inconnu se lit de haut en bas, comme ce livre :

- › les noms et les signatures d'abord, qui disent l'intention sans qu'on ouvre les fonctions (3.1) ;
- › une donnée qu'on suit de l'entrée à la sortie, en déroulant à la main (2.1) ;
- › les tests, des exemples d'usage écrits par quelqu'un qui savait ;
- › l'historique, `git blame` et le message de commit, qui disent pourquoi une ligne est là (3.3).

On ne lit pas tout : on lit jusqu'à pouvoir expliquer à voix haute ce que fait le morceau qu'on va toucher, et pas une fonction de plus.

Reste à refactoriser sans rien casser. Les petits pas ne sont sûrs qu'avec un filet de sécurité. Fowler le martèle : sans une série de tests qui confirme à chaque étape que le comportement observable n'a pas bougé, le refactoring devient un pari à l'aveugle (on y revient en 6.1). Encore faut-il que ces tests regardent le comportement observable et non la structure interne : un test collé aux détails casse à chaque renommage, et on finit par ne plus refactoriser. Et si le code n'a aucun test ? On commence par en écrire, non pour vérifier qu'il est juste, mais pour fixer ce qu'il fait aujourd'hui, bugs compris. Si l'un d'eux échoue après une retouche, le comportement a bougé. Michael Feathers appelle cela des tests de caractérisation.

Un second garde-fou touche à l'organisation du travail : l'image des deux casquettes, que Fowler tient de Kent Beck. On porte celle qui ajoute un comportement ou celle qui change la forme, jamais les deux à la fois, et chaque casquette a son propre commit, son propre enregistrement dans Git (6.3). Un correctif de trois lignes noyé dans deux cents lignes déplacées ne se relit pas. Le

collègue chargé de la revue de code (relire les changements avant qu'ils entrent dans le projet) l'approuve les yeux fermés. Et le jour où le bug revient, l'historique ne dit plus quel commit l'a ramené.

Le réflexe quotidien, c'est la **règle du boy-scout** de Clean Code : comme le scout qui quitte le campement plus propre qu'il ne l'a trouvé, laissez toujours le fichier un peu plus propre qu'à votre arrivée. Pas tout nettoyer, juste une retouche en passant, un nom flou corrigé, une fonction trop longue coupée en deux, dans son propre commit, casquette de refactoring sur la tête : la propreté devient une habitude, au lieu d'un grand ménage qu'on repousse sans fin.

↳ Refactoring

↳ CE QUI MÈNE AU CHAPITRE 4

Un nom qui dit l'intention, une fonction qui cache beaucoup derrière une petite surface, un commentaire réservé au pourquoi, une erreur qu'on fait disparaître plutôt que de l'attraper partout, un seul endroit qui fait foi pour chaque savoir, des petits pas protégés par les tests pour changer la forme sans changer le comportement : le chapitre a posé six habitudes. Une même boussole les relie : garder le code facile à changer, sans rien construire d'avance. On sait maintenant écrire une fonction propre, un fichier propre.

*Mais mille fonctions propres ne font pas un système clair, comme mille phrases justes ne font pas un livre : il leur manque un plan. L'architecture est ce plan d'ensemble d'un programme : elle décide ce qui dépend de quoi et où passent les frontières entre ses parties. Les raccourcis pris ici et là s'accumulent en **dette technique** : chaque raccourci rend le changement suivant plus cher, comme les intérêts d'un emprunt (l'image date de 1992, de Ward Cunningham). Sans plan d'ensemble, personne ne sait où la rembourser, et chaque modification risque de casser autre chose.*

« Un programme doit être écrit pour que les humains le lisent, et seulement accessoirement pour que les machines l'exécutent. » — Abelson & Sussman, Structure et interprétation des programmes informatiques, 1985



Donner une forme au système

architecturer

La [section 1.2](#) regardait la machine d'en haut, comme une ville vue du ciel. Ici, on regarde votre programme d'en haut : ses quartiers, c'est-à-dire ses grands groupes de fichiers, les échanges d'un groupe à l'autre, et ce qu'on peut démolir sans que le reste s'écroule.

À l'échelle du système, une question domine : qui dépend de qui, et qu'est-ce qui peut changer sans tout casser ? Prenons une boutique, avec sa règle de remise (10 % au-delà de 100 €) et sa base de données, où elle range ses commandes (on l'ouvre en [4.6](#)). Si on change de base de données, faut-il réécrire la règle de remise ? Idéalement non, parce que la règle ne devrait pas dépendre de l'endroit où l'on range les données. L'architecture, ce plan d'ensemble, fixe la forme des dépendances avant qu'elles ne se décident toutes seules, en désordre. Six questions la découpent.

1. Comment un morceau de code réutilise-t-il le travail d'un autre ? ([4.1](#))
2. Comment reconnaître les problèmes que d'autres ont résolus mille fois avant nous ? ([4.2](#))
3. Dans quel sens faire pointer les dépendances ? ([4.3](#))
4. Comment trancher quand deux bonnes idées s'opposent ? ([4.4](#))
5. Qui garde la forme de l'ensemble ? ([4.5](#))
6. Et où vivent les données quand le programme s'arrête ? ([4.6](#))

La cinquième a une réponse courte : un seul esprit. La sixième ouvre enfin la pièce que tout le chapitre cite sans y entrer : la base de données.

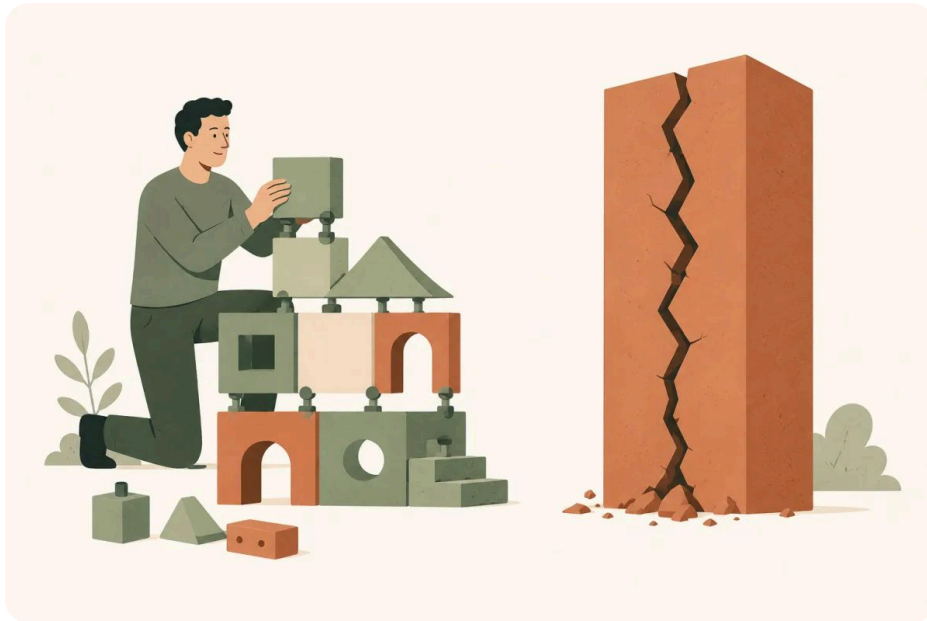


Figure 29 : Composer des pièces aux jointures nettes tient. Un bloc monolithique rigide se fend au premier changement.

4.1 Composer plutôt qu'hériter

Trois mots d'abord, car la suite du chapitre s'appuie dessus : ceux de la programmation orientée objet, la POO vue en [2.8](#).

- Une **classe** est un moule : la classe `Utilisateur` fixe qu'un utilisateur a un nom, un e-mail, et qu'il sait se connecter. Elle ne contient personne, comme un moule à gâteau ne contient pas de gâteau.
- Fabriquer un exemplaire à partir du moule s'appelle **instancier** la classe, ce que fait le mot-clé `new` dans beaucoup de langages (Java, C#, PHP, JavaScript).
- Une **instance** est l'exemplaire fabriqué à partir d'une classe : l'objet vu en [section 2.2](#), avec ses valeurs à lui (Dupont, dupont@exemple.fr).

```
class Utilisateur { nom; email; seConnecter() }  
// le moule : il décrit, il ne contient personne  
  
u = new Utilisateur("Dupont", "dupont@exemple.fr")  
// une instance : quelqu'un, avec ses valeurs
```

Prenez un administrateur : un utilisateur qui peut en plus bannir les autres. Recopier nom, e-mail et connexion dans un second moule serait absurde, d'où la première question de l'architecture : comment un moule réutilise-t-il le travail d'un autre ? Deux réponses dominent.

- **Hériter**, c'est dire « un administrateur est un utilisateur », ce qu'écrit le mot-clé `extends` : le moule enfant (`Administrateur`) reprend tout du moule parent (`Utilisateur`) d'un coup, sans rien demander. Il peut ensuite ajouter ses propres éléments (le pouvoir de bannir) ou réécrire certains de ceux du parent (une connexion plus stricte).
- **Composer**, c'est assembler : « un utilisateur a une façon de se connecter » veut dire qu'il contient un objet à part, issu d'un autre moule, qu'on peut remplacer sans toucher au moule `Utilisateur` . Aujourd'hui, la connexion se fait par mot de passe, avec un objet `ParMotDePasse` . Le jour où le site passe à un code reçu par texto, on le remplace par un objet `ParTexte` , et le moule `Utilisateur` ne bouge pas.

```
// hériter : l'enfant reprend tout, ajoute, réécrit
class Administrateur extends Utilisateur {
    bannir(autre) { ... }           // ajouté
    seConnecter() { ... }          // réécrit, plus strict
}

// composer : l'utilisateur contient un objet issu d'un autre moule
class Utilisateur { connexion; seConnecter() { connexion.vérifier() } }
u.connexion = new ParMotDePasse() // aujourd'hui
u.connexion = new ParTexte()
// demain : le moule Utilisateur n'a pas bougé
```

Le premier réflexe pour réutiliser, c'est l'héritage, parce qu'il ne coûte qu'un mot-clé : ni interface (2.3) à écrire, ni objet à brancher. Mais il est tout ou rien : l'enfant reprend tout du parent, qu'il le veuille ou non. L'exemple culte du livre *Design Patterns* : Tête la première le montre avec des canards plutôt que des utilisateurs. Un moule parent `Canard` écrit `voler()` une seule fois, et tous les canards en héritent. Puis débarque le canard de bain, qui en hérite aussi alors qu'il ne vole pas. On peut bien réécrire `voler()` à vide dans le canard de bain, mais il faudra y penser pour chaque canard en bois, en plastique ou en peluche

qui arrivera ensuite. Et le jour où quelqu'un améliore `voler()` chez le parent, tous les canards changent d'un coup, y compris ceux qu'on n'avait pas en tête.



Figure 30 : Tous sortent du même moule, donc tous héritent du vol. Le canard de bain aussi, et il y croit.

La **composition** sort le vol du moule du canard : `voler()` confie le travail à un objet à part, rangé dans le canard, qu'on remplace (vrai vol, ou aucun vol) sans toucher au canard, même pendant que le programme tourne.

```
// x héritage : voler est figé chez le parent
class CanardDeBain extends Canard { }
// hérite voler()... mais ne vole pas !

// ✓ composition : le canard confie le vol à un objet rangé en lui, qu'on
// peut changer
class Canard { comportementVol; voler() { comportementVol.voler() } }
canard.comportementVol = new NePeutPasVoler()
// nouvel objet, même canard
```

Le réflexe, quand on code : avant d'écrire `extends`, dire la phrase «X est un Y» et se demander si elle restera vraie pour tous les X, y compris ceux de l'an prochain. Si oui, l'héritage est permis. Si l'on dit plus naturellement «X a un Y», ou si «est un» n'est vrai que parfois, c'est de la composition. Le test sur six cas :

- › Un administrateur est un utilisateur : vrai pour tous, y compris demain. L'héritage est permis.
- › Un virement est une opération bancaire : comme le retrait et le dépôt, sans exception. L'héritage est permis.
- › Un carré est un rectangle : vrai en géométrie. Mais changer la largeur d'un rectangle laisse sa hauteur, alors qu'un carré doit suivre : le parent promet ce que l'enfant ne peut pas tenir. L'héritage est déconseillé.
- › Un employé est une personne : vrai aujourd'hui, mais il peut devenir manager l'an prochain, et la personne reste la même. Le rôle change, pas la personne : une personne a un rôle. Composition.
- › Une voiture est un moteur ? Absurde. Une voiture a un moteur. Composition.
- › Une commande est un client ? Absurde. Une commande a un client. Composition.

Reste une difficulté : au départ, on ne sait pas toujours que le canard de bain va arriver. Deux indices aident à le deviner. Ce qui varie est une action (voler, se connecter) plutôt qu'une identité (un canard), et on imagine déjà que cette action pourrait différer d'un canard à l'autre. Sans indice, on commence simple avec l'héritage et on refactorise le jour où le cas arrive, car passer de l'héritage à la composition est un refactoring classique (3.6).

Dans le doute, composer. L'héritage n'est pas interdit, mais il lie chaque enfant à son parent : une modification du parent peut casser un enfant qu'on n'a pas touché. On parle de *classe de base fragile* (*fragile base class* en anglais). La composition coûte quelques lignes de plus, pour confier le travail à l'objet, et en échange chaque objet se change seul. La maxime « Préférez la composition à l'héritage » vient du livre *Design Patterns* (1994), qui a popularisé les **patterns** : des réponses éprouvées à des problèmes qui reviennent sans cesse. La section 4.2 leur est consacrée.

↳ [Design Patterns : Tête la première](#) · [Design Patterns \(GoF\)](#)

4.2 Un pattern répond à un besoin, pas à un but

Un design pattern n'est pas une décoration qu'on plaque pour faire sérieux : c'est une réponse éprouvée à un problème récurrent. Un menuisier n'invente pas une façon d'assembler deux planches à chaque meuble : il connaît la queue d'aronde (des dents en trapèze qui s'emboîtent et résistent à la traction), le tenon-mortaise (une languette qui entre dans une encoche), et il prend l'assemblage que la contrainte réclame. Un pattern, c'est ça : un assemblage nommé, éprouvé par d'autres avant vous. L'idée vient de l'architecte Christopher Alexander (*A Pattern Language*, 1977), qui l'avait écrite pour des maisons et des villes. Les auteurs de *Design Patterns* l'ont transposée au logiciel.



Figure 31 : Un pattern est un assemblage nommé, éprouvé avant vous : on le choisit selon la contrainte, on ne le réinvente pas.

Trois patterns reviennent dans beaucoup de programmes, et vous en avez déjà croisé un :

- › **Stratégie** : changer de façon de faire pendant que le programme tourne, exactement ce qu'on vient de faire en remplaçant le vol du canard ;
- › **Observateur** : prévenir une liste d'abonnés quand une donnée change ;
- › **Décorateur** : ajouter un comportement autour d'un objet, en l'enveloppant, sans hériter de lui.

Aucun n'est sorcier. Pour l'Observateur, il suffit de savoir qu'une fonction est une valeur comme une autre, donc qu'on peut en ranger plusieurs dans une liste ([section 2.4](#)). L'objet surveillé, appelé le sujet, garde cette liste de fonctions et les appelle toutes à chaque changement. Vous le pratiquez déjà:

`addEventListener('click', fn)` abonne une fonction aux clics.

```
sujet.abonnés = [majAffichage, envoiMail]
sujet.changer(état) { abonnés.forEach(fn => fn(état)) }
// forEach : appelle chaque fonction de la liste
sujet.changer("commande payée")
// → majAffichage et envoiMail reçoivent "commande payée"
```

Le Décorateur est le post-it agrafé de la [section 2.4](#), appliqué à un objet entier: un objet en enveloppe un autre et ajoute une étape avant ou après chaque appel, sans que l'appelant s'en aperçoive.

```
service = new Log(new Cache(new Service()))
// l'appel traverse Log (il note), puis Cache (il garde les réponses déjà
// calculées), puis Service (le vrai travail)
```

Le **GoF** (le «Gang of Four», les quatre auteurs du livre fondateur *Design Patterns*) range ses 23 patterns en trois familles, et ces trois familles valent mieux que les 23 noms appris par cœur:

- › **Création** (comment créer les objets)
 - › Fabrique (*Factory Method*): choisit quelle classe instancier;
 - › Singleton: un exemplaire unique pour tout le programme, que n'importe quel code peut modifier, d'où sa mauvaise réputation;
 - › Monteur (*Builder*): construit un objet complexe étape par étape.
- › **Structure** (comment assembler les objets)
 - › Décorateur (vu plus haut);
 - › Adaptateur: traduit une interface en une autre;
 - › Composite: traite un groupe d'objets comme un seul.
- › **Comportement** (comment les faire communiquer)

- › Stratégie et Observateur (vus plus haut);
- › Commande : une action rangée dans un objet, qu'on peut mettre en attente ou annuler.

Vous ne mémorisez pas 23 solutions toutes faites : vous posez une question, « mon problème est de créer, d'assembler ou de communiquer? », et vous ne cherchez plus que dans une famille, de cinq à onze patterns au lieu de vingt-trois (la liste ci-dessus ne montre que les plus courants). Le langage compte aussi : là où les fonctions sont des valeurs, une Stratégie se réduit à passer une fonction, comme celle qui compare deux éléments quand on trie une liste (2.4). En 1996, Peter Norvig a compté que 16 des 23 patterns du GoF deviennent invisibles ou plus simples en Lisp, un langage fait de fonctions.

Au-dessus des 23 patterns, le GoF pose deux principes fondateurs : la composition plutôt que l'héritage (4.1), et **programmer pour une interface, pas pour une implémentation**. L'interface (2.3) dit ce qu'un objet sait faire, l'implémentation est la classe précise qui le fait vraiment. On retrouve la prise électrique de la section 2.7 : le mur ne connaît pas l'appareil, seulement la forme des trous. Votre code travaille avec ce qu'un objet sait faire, jamais avec ce qu'il est.

```
// x pour une implémentation : le code est soudé à une classe précise
exporter(doc) { new PDF().écrire(doc) }
// ne sait faire que du PDF

// ✓ pour une interface : « quelque chose qui sait écrire() »
exporter(doc, sortie) { sortie.écrire(doc) }
// PDF, CSV, HTML... : ce qu'elle FAIT, pas ce qu'elle EST
```

Le jour où il faudra écrire ailleurs que dans un PDF, vous ne rouvrez pas `exporter()` : vous lui passez autre chose. La fonction n'a jamais eu besoin de savoir dans quoi elle écrivait. Passer à une fonction ce dont elle a besoin (ici, la sortie), au lieu de la laisser le fabriquer, s'appelle l'**injection de dépendances**. Le nom est plus impressionnant que l'idée.

La règle finale : un pattern n'est jamais le but, on ne l'applique pas, on le **reconnaît** quand le besoin l'appelle. Le danger n°1 est la sur-application, forcer un pattern là où une solution simple suffirait. Une Fabrique qui ne fait que cacher

un seul **new**, sans rien choisir, c'est une queue d'aronde là où il n'y a qu'une seule planche.

▼ Voir les 23 patterns un par un, pour les curieux (facultatif)

Ce livre n'en détaille que trois, et c'est volontaire : on retient mieux trois familles que vingt-trois noms. Si vous voulez le catalogue entier, chaque pattern avec son problème, son schéma et le moment où il sert, un des plus clairs est [celui de Refactoring Guru](#), en français et gratuit à lire.

↳ [Design Patterns \(GoF\)](#) · [Tête la première](#)

4.3 Dépendre d'abstractions, inverser les dépendances

On a appris à programmer pour une interface. Reste à savoir qui dessine la prise. Tout se joue sur le sens des dépendances. Dépendre de quelque chose, c'est en avoir besoin pour fonctionner, donc bouger quand ça bouge (2.4). D'ordinaire, le code métier appelle la base de données : c'est lui qui dépend d'elle.

Clean Architecture partage le code en deux camps. D'un côté les **règles métier**, ce qui est propre à votre activité : comment se calcule une remise, quand une commande devient valide, qui a le droit d'annuler. Ça change peu, et c'est là qu'est la valeur. De l'autre la base de données, le framework (l'ossature de code toute prête sur laquelle on bâtit l'application), l'écran. Ça change souvent.

La **règle de dépendance** de *Clean Architecture* interdit une seule chose : que le code métier dépende de la technique. Sinon, ce qui devait être stable bouge à chaque fois que la technique bouge. D'où l'inversion : le code métier déclare une interface, et la technique s'y plie. Cette interface est l'abstraction du titre : la liste de ce dont le code métier a besoin, sans dire qui la fournit.

Chez vous, ce n'est pas le fabricant d'aspirateurs qui décide de la forme des prises : la maison fixe la forme, et tous les appareils s'y plient. Si chaque aspirateur imposait la sienne, il faudrait casser le mur à chaque achat. Le code mé-

tier tient le rôle du mur et dessine la prise : c'est l'interface. La base de données tient le rôle de l'aspirateur.

```
// le code métier déclare CE DONT il a besoin : une interface, c'est ça,  
// l'abstraction du titre  
interface DépôtCommande { sauvegarder(commande) }  
// la technique (base de données, envoi de mails...) l'implémente, et le  
// code métier ignore tout d'elle  
// → on change de base de données sans toucher une ligne de métier
```

inverser la dépendance : la flèche se retourne

✗ avant



la base change → le code métier casse

✓ après



le code métier ignore la base

Figure 32 : Le code métier déclare l'interface, la base s'y plie. Il ne dépend plus de rien de volatil.

C'est le «il me faut quelque chose qui sait faire X» de la [section 2.7](#), monté à l'échelle d'un système : les frontières protègent ce qui compte du reste.

Robert Martin, l'auteur de *Clean Architecture*, va plus loin avec une formule-choc : **la base de données et le framework sont des détails**, au même titre que la marque des câbles électriques d'une maison. Votre code métier ne devrait même pas savoir qu'il range ses commandes dans MySQL, une base de données parmi d'autres. D'où l'objectif qu'il fixe : **un bon architecte maximise le nombre de décisions non prises**. Moins on s'engage tôt sur les détails, plus on garde d'options ouvertes quand on en sait davantage.

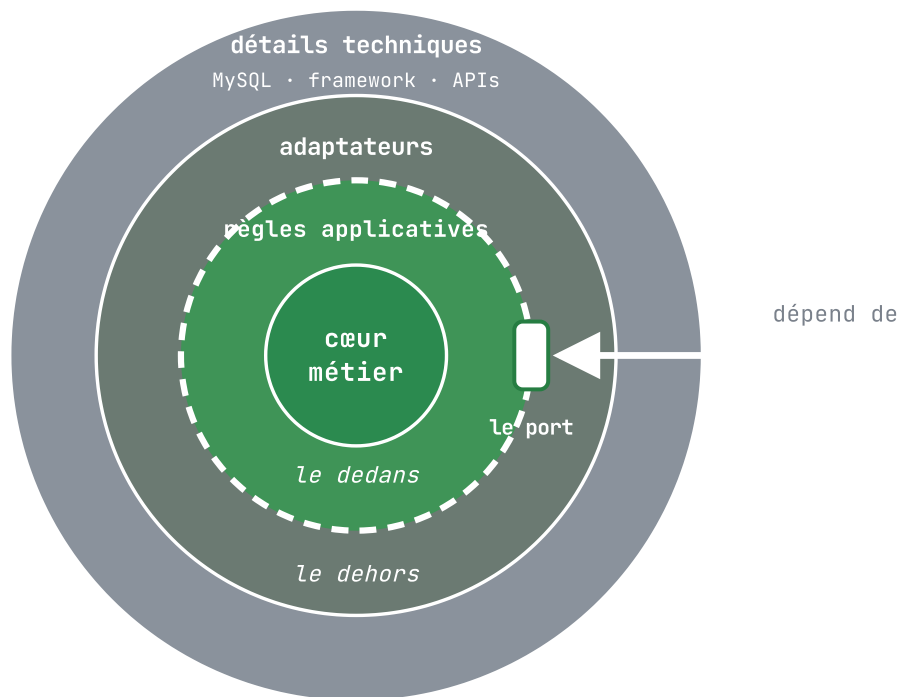
Les cercles et les ports

La règle de dépendance, appliquée à tout le système, donne les quatre cercles concentriques de Clean Architecture, héritiers de l'**architecture hexagonale** d'Alistair Cockburn (2005), dite **ports et adaptateurs**. Du centre vers l'extérieur :

1. le cœur métier : les règles que l'entreprise appliquerait avec un carnet et un stylo si le logiciel n'existait pas ;
2. les règles applicatives : l'enchaînement des étapes d'une action complète, vérifier le stock, débiter la carte, envoyer la confirmation ;
3. les adaptateurs : le code qui fait la jonction avec le dehors, celui qui reçoit les requêtes du navigateur et celui qui lit et écrit dans la base ;
4. les détails techniques : MySQL, le framework, les API tierces, c'est-à-dire le logiciel d'une autre entreprise (paiement, envoi de SMS) qu'on appelle par le réseau.

Une seule règle les tient : un cercle extérieur peut dépendre d'un cercle intérieur, jamais l'inverse. Quand le dedans (les deux cercles du centre) a besoin du dehors (les deux cercles extérieurs), pour sauvegarder ou envoyer un mail, il passe par une interface qu'il dessine lui-même.

les quatre cercles : tout pointe vers le cœur



Les dépendances traversent vers le centre par un port, jamais l'inverse

Figure 33 : Quatre cercles, une règle : l'extérieur dépend de l'intérieur, jamais l'inverse. Le port (la prise blanche) se pose sur la frontière du dedans (les cercles verts), et l'adaptateur, juste dehors, s'y branche. Un vrai système en expose plusieurs, un par besoin.

En pratique : le cœur définit une interface `DépôtCommande` avec une méthode `sauvegarder()`. C'est le port : la prise que le cœur expose vers l'extérieur, dont il définit la forme. L'adaptateur MySQL `DépôtCommandeMySQL` implémente ce port. L'adaptateur de test `DépôtCommandeMémoire` implémente le même port avec une simple liste en mémoire, si bien que les tests tournent sans base de données. Le code du cœur appelle `depot.sauvegarder(commande)` sans jamais savoir ce qui est branché. Passer de MySQL à PostgreSQL, une autre base du même genre : brancher un

autre adaptateur, le cœur ne bouge pas. Ce découpage coûte une interface de plus : on le réserve aux frontières qui bougent vraiment, sinon c'est la sur-application de la [section 4.2](#).

Reste une question : qui branche l'adaptateur ? Le code de démarrage du programme, le seul endroit qui connaisse MySQL : il crée l'adaptateur et le passe au cœur en argument, c'est l'injection de dépendances vue en [4.2](#). L'injection est la technique, l'inversion est le principe : celui qui possède l'interface. À l'exécution, les appels vont toujours du cœur vers la base. Seule la dépendance, c'est-à-dire qui doit connaître qui, est retournée.

```
// le port : défini par le cœur
interface DépôtCommande { sauvegarder(commande) }

// deux adaptateurs pour le même port
class DépôtCommandeMySQL implements DépôtCommande { sauvegarder(c) { /*
écrit dans MySQL */ } }
class DépôtCommandeMémoire implements DépôtCommande { sauvegarder(c) {
liste.ajouter(c) } }

// le cœur ne connaît que le port
class Boutique {
    constructor(depot) { this.depot = depot }
    // constructor s'exécute au new, this.depot est un casier de cet objet :
    // le dépôt arrive de l'extérieur, c'est l'injection
    valider(commande) { this.depot.sauvegarder(commande) }
}

// le code de démarrage branche l'adaptateur
boutique = new Boutique(new DépôtCommandeMySQL())
// pour de vrai, face aux clients
boutique = new Boutique(new DépôtCommandeMémoire()) // dans les tests
```

Même un langage s'y branche

Les langages eux-mêmes appliquent la même règle, et Python la montre le plus simplement. Il ne connaît pas votre objet `Panier`, et pourtant `len(Panier())` marche : `len` donne la longueur d'une chose (le nombre de

lettres d'un texte, d'éléments d'une liste), et il le fait en cherchant, dans le moule de l'objet, une méthode au nom prévu d'avance, `__len__`. Le langage prévoit l'interface, votre objet s'y plie.

```
class Panier:
    def __init__(self):
        self.articles = ["pain", "lait", "café"]
    def __len__(self):
        return len(self.articles)

len(Panier())    # 3
```

Ici, `class Panier` ouvre le moule d'un panier, `__init__` s'exécute à chaque panier fabriqué et le remplit de trois articles, et `self` désigne le panier dont on demande la longueur. Le `len` du `return` mesure la liste interne, pas le panier : ce n'est pas une récursion (2.5). `__len__` est une méthode, une fonction rangée dans le moule de l'objet (les doubles tirets bas marquent un nom réservé par Python) : quand on écrit `len(quelque_chose)`, Python l'appelle.

La même interface existe pour `for`, `+` ou `print`, chacun avec son nom réservé. *Fluent Python* (Luciano Ramalho) appelle cet ensemble le modèle de données de Python : les interfaces du langage lui-même. La leçon : on ne configure pas un langage, on s'y **branche**, en suivant les interfaces qu'il prévoit.

Le D de SOLID, et les quatre autres

L'inversion de dépendance porte une lettre : c'est le D de **SOLID**, cinq principes de conception réunis sous un acronyme.

- › **S** (Single Responsibility) : un module (un fichier de code) ne répond qu'à un seul **acteur**, c'est-à-dire un groupe humain qui demande le même genre de changements ;
- › **O** (Open/Closed) : ouvert à l'extension, fermé à la modification, comme l' `exporter()` de la [section 4.2](#) qu'on n'a pas eu à rouvrir pour écrire ailleurs ;

- › **L (Liskov Substitution)** : un moule enfant doit pouvoir remplacer son moule parent sans surprise, ce que le canard de bain de la [section 4.1](#), qui hérite de `voler()` sans voler, ne fait pas ;
- › **I (Interface Segregation)** : aucun code ne doit dépendre de ce qu'il n'utilise pas : mieux vaut plusieurs petites interfaces qu'une énorme, sinon c'est une prise à trente trous pour un appareil qui n'en utilise que deux ;
- › **D (Dependency Inversion)** : le code métier ne dépend pas de la technique, les deux dépendent d'une interface, et c'est le code métier qui la dessine.

L'exemple du S est de Martin lui-même : une classe `Employé` qui calcule la paie, produit le rapport d'heures et s'enregistre en base. Trois tâches, mais trois acteurs : la compta pour la paie, les RH pour le rapport, les administrateurs de la base pour l'enregistrement. Le jour où la compta fait modifier un calcul que le rapport utilise aussi, le rapport des RH casse sans que personne ne l'ait demandé.

C'est le principe le plus mal compris des cinq. Beaucoup le réduisent à « une fonction fait une chose », qui est un autre principe, plus bas niveau, venu de Clean Code. Martin corrige lui-même dans Clean Architecture, chapitre 7 : ce qu'on compte, c'est le nombre de groupes servis, pas le nombre de choses que fait le code.

Entre deux objets : la loi de Déméter

La même idée vaut à la plus petite échelle, entre deux objets : moins un objet en sait sur les autres, moins il casse quand ils changent. La **Loi de Déméter** dit « ne parle qu'à tes voisins immédiats » : un objet n'appelle que ses propres méthodes, celles des objets qu'on lui passe et celles des objets qu'il garde en lui. Pas de chaîne comme `commande.getClient().getAdresse().getVille()`, qui plonge dans la structure interne de trois inconnus : le jour où l'un d'eux change de forme, l'appelant casse. La loi vise les objets qui cachent leur intérieur, pas une simple structure de données, faite justement pour être lue.

Deux classiques nomment le même défaut. Clean Code appelle **catastrophe ferroviariaire** (train wreck en anglais) une chaîne d'appels accrochés les uns aux autres comme des wagons. Chaque wagon y est un **getter**, une méthode qui ne fait que renvoyer une valeur. Refactoring parle de **chaîne de messages** et la corrige avec Hide Delegate, en exposant une méthode qui dit ce qu'on veut plutôt que le chemin pour l'obtenir. À l'échelle du système comme entre deux objets, la règle ne change pas : ne dépendre que de ce qu'on a le droit de connaître.

```
// x une chaîne qui traverse trois inconnus
commande.getClient().getAdresse().getVille()

// ✓ on demande ce qu'on veut, pas le chemin pour l'obtenir
commande.villeDeLivraison()
```

↳ [Clean Architecture](#) · [Clean Code](#) · [Refactoring](#) · [Fluent Python](#)

4.4 Pas de « best practice », seulement le moins mauvais des compromis

Plus on regarde le système de haut, moins il y a de réponses universelles. Le livre *Software Architecture: The Hard Parts* (Neal Ford et ses coauteurs) le pose net : chaque problème d'architecture est un flocon de neige, il n'y en a pas deux pareils. La compétence n'est pas de choisir le bon pattern, c'est de **peser** les compromis. Deux outils pour ça.

Premier outil, le **quantum d'architecture** : le plus petit morceau d'un système qu'on peut mettre en ligne et tester seul, et qui peut tomber sans rien entraîner d'autre. Un service, c'est un morceau de l'application mis en ligne de son côté. Deux services qui partagent la même base de données, c'est comme deux appartements derrière un seul disjoncteur : l'un fait sauter le disjoncteur, l'autre est dans le noir. Ils ne forment qu'un seul quantum.

Le test, devant n'importe quel schéma : « si ce morceau change ou tombe en panne, combien d'autres tombent avec lui ? » Ce morceau et tous ceux qui tombent avec lui forment votre quantum. La question ne demande pas un gros système pour être utile : sur un premier projet, elle se pose déjà devant un fichier que trois pages appellent. La base commune n'est pas le seul lien : un ser-

vice qui attend la réponse d'un autre pour continuer (on dit que l'appel est synchrone) reste bloqué quand l'autre tombe, et les deux ne forment encore qu'un seul quantum. Vingt «microservices», des services volontairement minuscules, qui se partagent une base ou s'attendent les uns les autres ? Un seul quantum.

le quantum : ce qui tombe ensemble

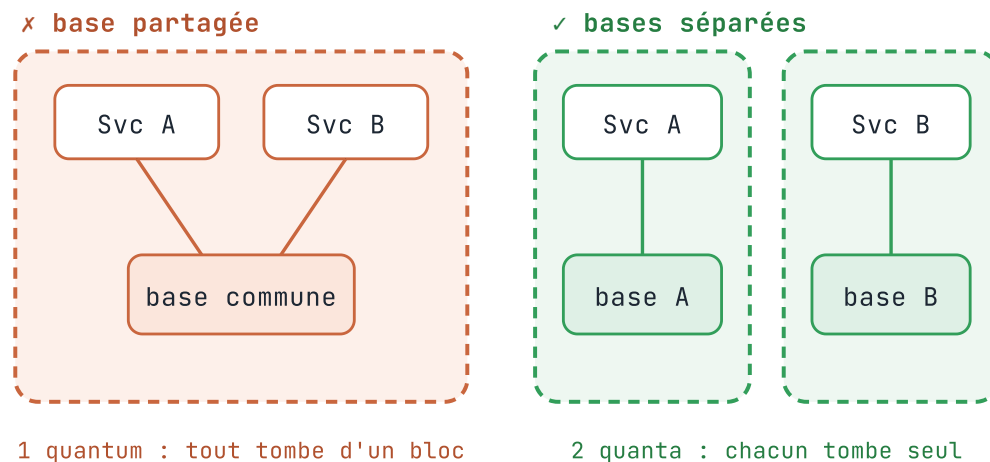


Figure 34 : Chaque cadre pointillé est un quantum. En orange, la base commune soude les deux services en un seul. En vert, chaque service tombe seul.

Second outil : « la **réutilisation, c'est du couplage** », le couplage étant le lien qui oblige deux morceaux de code à changer ensemble. Reprenez le fichier que trois pages appellent : le quantum demandait ce qui tombe avec lui, ici la question devient ce qui change avec lui. Si on le modifie pour la première page, les deux autres pages changent aussi, sans que personne l'ait demandé. À l'échelle d'un système, c'est pareil : partager une classe métier entre services propage chaque changement partout à la fois.

D'où un réflexe contre-intuitif : on duplique parfois exprès. On ne met en commun que ce qui est vraiment un seul savoir, tenu de rester cohérent (le DRY de la [section 3.5](#)). Pour deux bouts qui se ressemblent mais évolueront à part, un peu de copie vaut mieux qu'un mauvais couplage.

Entre petit quantum et mise en commun, aucun principe ne tranche. Le petit quantum achète de l'**indépendance**, la mise en commun achète de la **cohérence**, et chacune se paie : l'une en duplication et en appels réseau, puisque des

morceaux séparés doivent se parler à distance ([chapitre 5](#)), l'autre en couplage. On choisit au cas par cas, selon ce qui compte le plus ici : pas de règle, seulement le moins mauvais des compromis.

↳ [Software Architecture: The Hard Parts](#)

4.5 L'intégrité conceptuelle : un seul esprit

Peser les compromis au cas par cas ne suffit pas si chacun pèse dans son coin : la plus belle architecture meurt si quarante équipes empilent leurs idées sans coordination. Le cas type est l'API. Une **API** (interface de programmation) est le guichet d'une application : les autres programmes y déposent leurs demandes, avec un formulaire à remplir et un format de réponse imposé (la [section 7.2](#) y revient). Quarante équipes sans se parler, et ce guichet devient « la maison qui rend fou » d'Astérix, celle du laissez-passer A38 : chaque bureau réclame son propre formulaire, ici l'identifiant change de nom, là la date change de format. Chaque bureau marche. L'ensemble rend fou.

```
// GET /adresse : la demande au guichet, → : sa réponse
// x trois équipes, trois noms pour le même identifiant, trois écritures du
// même 9 juin (la dernière en secondes depuis 1970)
GET /users   → { id,      "2026-06-09" }
GET /orders  → { userId, "09/06/2026" }
GET /cart    → { uid,    1780963200 }

// ✓ un seul vocabulaire, un seul format : l'appelant devine tout
GET /users · /orders · /cart → { id, "2026-06-09" }
```

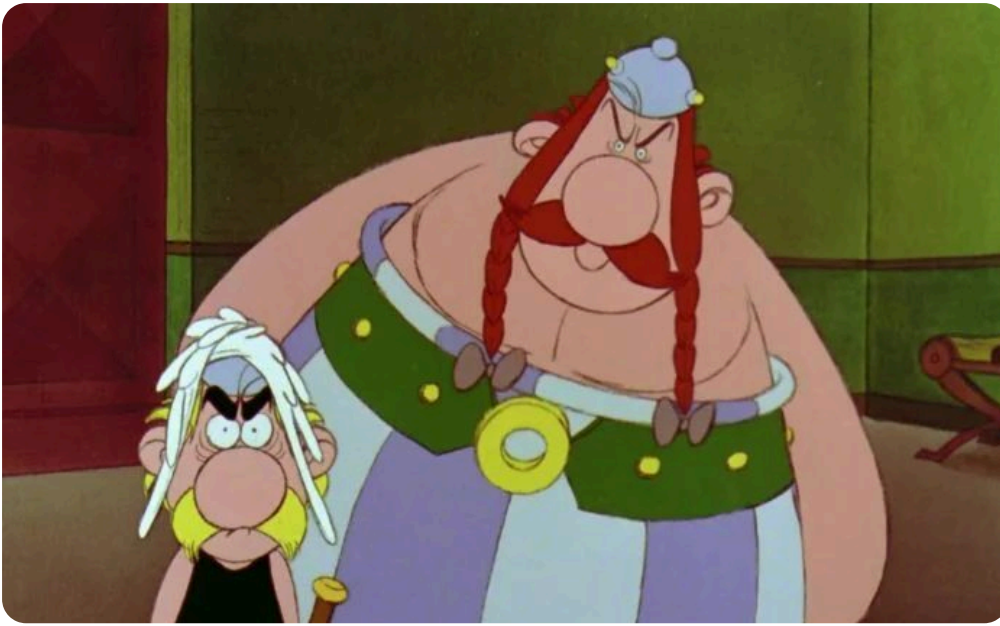


Figure 35 : Le laissez-passer A38 : chaque bureau vous renvoie au suivant (*Les Douze Travaux d'Astérix*, 1976).

Contre ce désordre, *Le Mythe du mois-homme*, de Fred Brooks, pose cette règle : « l'intégrité conceptuelle est la considération la plus importante dans la conception d'un système ». L'intégrité conceptuelle, c'est qu'un système paraisse pensé d'un seul tenant : mêmes noms, mêmes formats, mêmes règles partout. Elle doit sortir d'un seul esprit, ou d'un très petit groupe, sinon c'est la tour de Babel.

Un seul esprit ne veut pas dire une seule personne qui code tout : ce petit groupe décide de la forme, les autres la remplissent. Et Brooks insiste : « la forme libère ». Une fois la structure fixée, chacun sait où sa pièce s'emboîte, et code plus vite, pas plus lentement.

Le modèle vit dans le code

Reste qu'une vision unique ne survit pas dans un document : dès les premières semaines, le schéma rangé dans la documentation de l'équipe ne correspond plus au code, parce que le code bouge et le schéma non. La vision doit donc vivre dans le code. Le **Model-Driven Design** (la conception guidée par le modèle) d'Eric Evans demande que le modèle, c'est-à-dire la description du métier avec ses objets et ses règles, s'écrive directement dans le code.

Prenez une règle métier: «une commande payée ne s'annule plus». Tant qu'elle est éparpillée, chaque écran qui affiche un bouton Annuler refait la vérification dans son coin, et l'un d'eux finira par l'oublier. Chez Evans, la règle vit dans l'objet `Commande` lui-même: demandez `commande.annuler()` sur une commande payée, l'objet refuse. Le statut est l'énumération de 2.3, et `annuler()` n'est permis que depuis certains de ses états: la commande est une **machine à états**, un objet qui ne passe d'un état à l'autre que par des transitions écrites une fois, dans le code. Cet objet ne se contente pas de porter les champs d'une ligne de la base de données: il connaît les règles d'une vraie commande. Martin Fowler appelle *Anemic Domain Model* («modèle anémique») un objet réduit à ses champs, et y voit un défaut. Le modèle n'est plus à côté du code: il est le code.

```
// x la règle éparpillée : chaque écran la refait (et l'un l'oublie)
si (commande.statut != "payée") { commande.statut = "annulée" }

// ✓ la règle vit dans l'objet : l'écran demande, l'objet refuse
class Commande { annuler() { si (payée) refuser("déjà payée") sinon statut
= "annulée" } }
```

Les mots comptent autant que les règles. Dans une compagnie maritime, l'expert du métier (celui qui connaît l'activité réelle de l'entreprise) dit «cargo», «escale», «itinéraire», pendant que le code dit «fiche 42», «booléen», «statut2». Chaque réunion devient alors une traduction, et la traduction perd toujours quelque chose. Evans demande que l'équipe et l'expert choisissent ensemble les mots exacts du métier et les emploient partout, en réunion comme dans le code: c'est l'**Ubiquitous Language** («langage omniprésent»), une seule langue partagée. C'est le «un mot par idée» de la [section 3.1](#), étendu aux gens du métier. «Partout» s'entend dans le domaine d'une équipe: la [section 7.4](#) montre où cette langue s'arrête.

Entité ou objet-valeur: le tri d'Evans

Un modèle fait d'objets, ça se trie. Evans distingue deux sortes d'objets, et une seule question suffit à les trier: celui-là, je le suis dans le temps ou je le remplace? C'est la distinction valeur/référence de la [section 2.2](#), transposée au métier.

- › Une **entité** a une identité qui survit à tous ses changements : le client 42 reste le client 42 s'il déménage et change de nom, on le suit dans le temps.
- › Un **objet-valeur** n'est que son contenu : deux billets de 10 € sont interchangeables, deux adresses identiques sont la même adresse.

Posez la question à chaque objet de l'application : le montant d'une facture, on le remplace, c'est un objet-valeur. La facture elle-même, pour la comptabilité qui la suit dans le temps sous son numéro, est une entité. Une question, et chaque objet trouve sa place, mais la réponse dépend de qui s'en sert : pour le client qui n'en garde qu'une copie, la même facture n'est qu'un document, une valeur.

Ce tri dicte deux gestes concrets.

- › **Comparer.** Deux entités se comparent par leur identité : deux clients qui portent le même nom restent deux clients, et le code compare leurs `id`. Deux objets-valeurs se comparent par leur contenu, champ par champ.
- › **Modifier.** Un objet-valeur ne se modifie pas, il se remplace. Retoucher une adresse partagée par deux commandes, c'est tomber dans le piège de la référence : les deux commandes déménagent d'un coup (2.2). Donner une adresse neuve à la bonne commande, et le piège disparaît.

Reste une conséquence de « le modèle est le code », et elle dérange. Beaucoup d'équipes séparent les rôles : l'architecte dessine les plans, les développeurs construisent, et Brooks lui-même défendait cette séparation. Mais si le modèle est le code, elle ne tient plus : concevoir, c'est coder. L'architecte qui ne touche jamais au clavier finit par dessiner des plans impossibles à construire. Evans appelle ça les **Hands-on Modelers**, les concepteurs qui gardent les mains dans le code : la même tête tient le crayon et le clavier.

↳ [Le Mythe du mois-homme](#) · [Domain-Driven Design](#) · [Learning DDD](#)

4.6 Ranger pour retrouver : la base de données

Le chapitre a parlé de la base de données sans l'ouvrir : un détail à tenir à distance en 4.3, une pièce que deux services ne devraient pas partager en 4.4. Regardons dedans. Les casiers de la mémoire vive (la RAM) se vident quand le

programme s'arrête ou que le courant coupe. Or une commande passée à 14 h doit encore exister demain matin : il faut l'écrire sur le disque, dans un fichier (une sortie du programme, 2.1). Alors pourquoi pas un simple fichier ?

Trois ennuis qu'un fichier ne règle pas

Un fichier texte tient très bien une liste de commandes, jusqu'au jour où trois ennuis arrivent, souvent la même semaine :

- › Deux programmes écrivent dedans en même temps, et leurs lignes se mêlent.
- › Le courant tombe au milieu d'une écriture, et le fichier reste à moitié écrit, sans que personne le sache.
- › Retrouver la commande du client Dupont oblige à lire le fichier du début à la fin : c'est le carnet non trié de 1.3, $O(n)$, relu à chaque question.

Une **base de données** est un programme qui garde des données sur le disque et qui, contrairement à un fichier texte, tient trois promesses :

- › plusieurs programmes peuvent s'en servir sans mêler leurs écritures ;
- › chaque changement s'écrit en entier ou pas du tout ;
- › une ligne se retrouve sans tout relire.

Ce programme s'appelle le moteur de la base : PostgreSQL et MySQL en sont deux exemples, SQLite un troisième, qui tourne à l'intérieur de votre propre programme. Votre code parle au moteur au lieu de toucher au fichier.

Les deux premières garanties viennent de la **transaction**, un groupe de changements qui réussit en entier ou pas du tout, sans se mêler à ceux des autres. Exemple : enregistrer une commande et retirer l'article du stock, les deux gestes ou aucun. On les nomme atomicité et isolation, deux lettres du sigle ACID que la section 6.7 détaille, avec un piège à la clé. Ici, c'est la troisième garantie qui compte, retrouver une ligne sans tout relire, celle dont on profite à chaque lecture.

La grille et la clé : le modèle relationnel

Ranger d'abord. Le modèle relationnel, posé en 1970 par Edgar Codd, range chaque sorte de chose dans une **table** : une grille à colonnes fixes, une ligne par chose, une table Clients, une table Commandes. (Rien à voir avec la table de hachage de [1.3](#) : même mot, autre objet.) Chaque ligne porte un numéro à elle, sa **clé** : ce qu'on donne pour obtenir la ligne. Une commande ne recopie pas son client, elle garde la clé du client (dite clé étrangère). Recoller une commande et son client à la lecture est une **jointure** : pour chaque commande, aller chercher la ligne du client qui porte ce numéro.

La liste des tables, de leurs colonnes et du type de chaque colonne forme le **schéma** : la forme des données, décidée avant la première ligne, que le moteur fait respecter à chaque écriture. La **normalisation** est le DRY de la [section 3.5](#) appliqué aux données : chaque fait s'écrit une fois, le téléphone du client dans sa ligne, jamais dans ses commandes. Sinon le client change de téléphone, et ses commandes se contredisent : certaines gardent l'ancien, comme un prix en rayon qui diffère du prix en caisse.

On se répète parfois exprès pour lire plus vite : c'est la dénormalisation, au prix d'une écriture plus lourde, et la [section 6.6](#) montre cet échange.

L'index se paie comme le tri

Chercher ensuite. Trouver Dupont parmi dix millions de clients oblige le moteur à lire la table entière, le carnet non trié. Un **index** est l'annuaire trié de la [section 1.3](#), tenu à jour par le moteur pour une colonne donnée, afin qu'une recherche sur cette colonne descende en quelques étapes au lieu de dix millions. Le plus souvent, cet annuaire est un arbre équilibré (B-tree) : trois ou quatre lectures de disque suffisent pour dix millions de lignes.

Le détail qui compte est le prix. Chaque fois qu'une ligne entre, le moteur doit aussi la ranger dans chaque annuaire de la table, à sa place dans l'ordre. Martin Kleppmann, auteur de *Designing Data-Intensive Applications*, le résume en une loi : « des index bien choisis accélèrent les lectures, mais chaque index ralentit

les écritures». On n'indexe donc pas « au cas où » : on indexe les colonnes qu'on cherche souvent, et on paie l'écriture pour la lecture, exactement l'échange du tri de la [section 1.3](#).

Dire ce qu'on veut : la requête et son plan

Le langage standard pour parler au moteur est **SQL** (Structured Query Language, langage de requête structuré), et une requête SQL, c'est-à-dire une question posée à la base, dit ce qu'on veut, pas comment l'obtenir. On écrit « le nom du client dont l'e-mail vaut dupont@exemple.fr », et le moteur choisit seul la méthode, lire toute la table ou passer par un index. Le **plan d'exécution** est la méthode que le moteur a choisie pour répondre à une requête. Écrit devant une requête, **EXPLAIN** affiche ce plan sans exécuter la requête, chez PostgreSQL comme chez MySQL (sa variante **EXPLAIN ANALYZE**, elle, l'exécute pour de bon : sur un **UPDATE** ou un **DELETE**, elle modifie vraiment les données).

```
SELECT nom FROM clients WHERE email = 'dupont@exemple.fr'
-- EXPLAIN (PostgreSQL) : Seq Scan on clients      x lit les dix millions
de lignes
CREATE INDEX clients_email ON clients (email)
-- EXPLAIN : Index Scan using clients_email        ✓ quelques étapes
SELECT nom FROM clients WHERE lower(email) = 'dupont@exemple.fr'
-- EXPLAIN : Seq Scan on clients      x la fonction masque la colonne,
l'index reste inutilisé
```

Le piège classique est à la dernière ligne : l'index existe, et le moteur ne s'en sert pas. Dès qu'une fonction enveloppe la colonne, **lower(email)** ou une date dont on extrait l'année, la plupart des moteurs ne retrouvent plus l'annuaire, trié sur les e-mails tels qu'ils sont écrits, et non sur leur version en minuscules. Le remède : indexer l'expression elle-même (un index fonctionnel : PostgreSQL le propose depuis longtemps, MySQL depuis la 8.0.13), ou stocker les e-mails déjà en minuscules et comparer la colonne telle quelle.

Le guide en ligne de Markus Winand, [Use The Index, Luke](#), recense ces cas où un index existe mais reste inutilisé : une fonction autour de la colonne, une conversion de type implicite, un joker en début de motif. Le réflexe, quand on code : devant une requête lente, lire son plan avant de toucher au code.

Le second piège ne se voit dans aucun plan, parce que chaque requête, prise seule, est parfaite. Imaginez une page qui affiche cent commandes avec le nom de leur client. Le code demande d'abord les commandes, puis répète pour chacune, dans une boucle, une requête qui cherche son client : cent une requêtes, là où une jointure en fait une. Le **N+1** est une boucle qui lance une requête par tour au lieu d'une seule jointure, et les ORM (« mapping objet-relationnel »), ces outils qui écrivent le SQL à votre place, le produisent sans prévenir. Le remède : demander les clients de toutes les commandes en une seule jointure, ou en une seconde requête groupée, ce que les ORM appellent le chargement anticipé (*eager loading*).

Chaque requête paie un aller-retour : la question part vers le moteur, la réponse revient, et le moteur habite souvent une autre machine.

↳ Designing Data-Intensive Applications

↳ CE QUI MÈNE AU CHAPITRE 5

Composer des pièces plutôt qu'hériter d'un moule, reconnaître dans un pattern un assemblage déjà éprouvé, faire pointer les dépendances vers les règles métier, peser chaque découpage en demandant ce qui tombe avec lui, confier la forme de l'ensemble à un seul esprit. Et ranger les données dans une base où chaque index accélère les lectures et ralentit les écritures. Voilà les six décisions du chapitre. Une même question les relie : qu'est-ce qui peut changer sans entraîner le reste ? La forme est posée, les frontières tracées.

Mais ce système n'habitera pas une seule machine, les sections 4.4 et 4.6 l'ont laissé voir : le navigateur est ici, le serveur ailleurs, la base souvent sur une autre machine, parfois sur un autre continent, et chacune des cent une requêtes du N+1 fait le voyage. Entre eux s'étend le réseau, les câbles et les relais qui transportent les messages d'une machine à l'autre : un monde qu'on n'a pas encore regardé, avec ses lois physiques à lui. Avant de livrer quoi que ce soit, il faut savoir comment deux machines se parlent.

*« Ne sous-estimez jamais la bande passante d'un break rempli de cassettes fonçant sur l'autoroute. » — Andrew Tanenbaum, *Computer Networks*, 1996*

5 Deux machines se parlent

le réseau

Jusqu'ici, nous regardions l'intérieur d'une machine. La pyramide de la [section 1.2](#) l'annonçait déjà : sous le disque, il reste un étage, le plus lent de tous, environ un million de fois la durée d'un accès à la RAM. Nous y voici. Un clic suffit : la requête, votre demande de page, part du navigateur, traverse des câbles, des routeurs (des relais qui aiguillent les messages) et parfois un océan, puis la réponse revient. Ce voyage a ses lois, et elles ne ressemblent à rien de ce qu'on a vu : la distance se paie, les messages se perdent, et personne ne dirige la circulation.

Le voyage se raconte en cinq étapes : ce que coûte vraiment la distance, le contrat minuscule qui fait tenir Internet, la fiabilité fabriquée par-dessus, le partage sans chef d'orchestre, et ce que le web paie à chaque page.

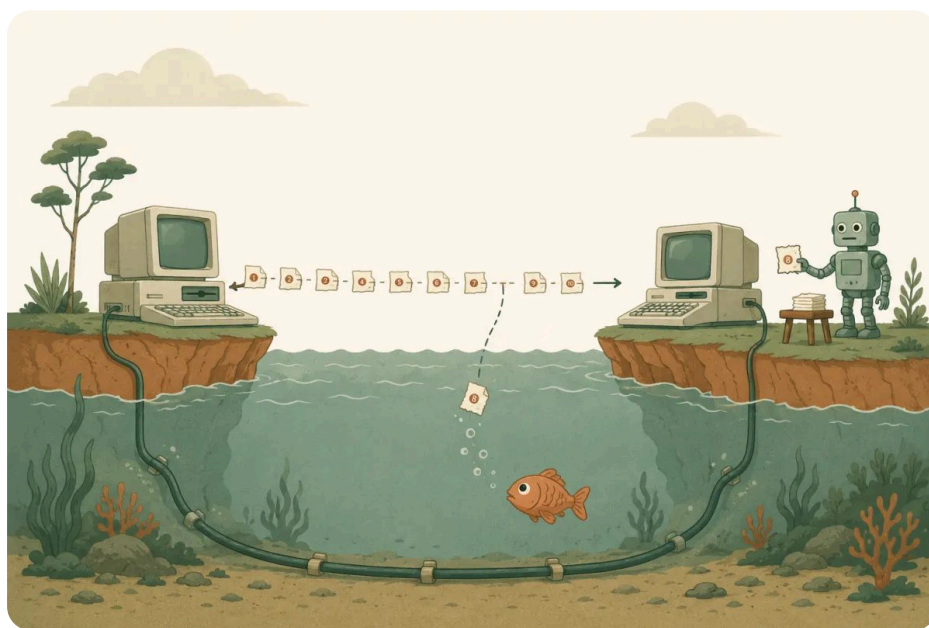


Figure 36 : Le réseau découpe, numérote, et perd des morceaux en route. Tout l'art du chapitre : faire comme si de rien n'était, en renvoyant une copie.

5.1 La distance ne se compresse pas

Deux nombres décrivent un réseau, et on les confond tout le temps. Imaginez une autoroute. La **bande passante**, c'est le nombre de voies : combien de données passent de front chaque seconde. La **latence**, c'est la longueur du trajet : le temps qu'un message met à faire l'aller-retour. Relisez le break de Tanenbaum, il roule déjà sur cette autoroute. Imaginez-le chargé de cassettes par milliers et parti pour des heures d'autoroute : les cassettes lui font une bande passante énorme, les heures de route une latence terrible. Les pubs de votre fournisseur d'accès ne vantent que la bande passante, à grands renforts de « fibre 2 Gbit/s ». Pourtant, c'est la latence qui vous fait attendre.

La différence est physique. La bande passante s'achète : une fibre de plus, des fréquences de plus, des voies en plus. La latence, elle, ne s'achète pas, on ne rapproche pas les villes : elle est bornée par la vitesse de la lumière. Dans la fibre, le signal avance à environ deux tiers de la vitesse de la lumière, et un aller-retour Paris–New York ne descendra jamais très en dessous de 60 ms (70 à 80 ms en pratique, car le câble ne suit pas une ligne droite). Aucun abonnement ne raccourcit l'Atlantique.

Or c'est la latence qui commande le web. Mike Belshe, l'un des concepteurs de SPDY (l'ancêtre de HTTP/2, la langue que parlent navigateurs et serveurs), l'a mesuré chez Google, et *High Performance Browser Networking* reprend son résultat : passé quelques mégabits par seconde, **doubler la bande passante ne change presque rien** au temps de chargement d'une page. Réduire la latence l'accélère presque linéairement. La raison : une page moyenne, c'est des dizaines de petites ressources (images, feuilles de style, polices, scripts), donc des dizaines d'allers-retours, dont beaucoup à la file (le navigateur ne découvre une image qu'après avoir lu la page), qui s'additionnent. Le web n'est pas limité par la largeur de la route, il est limité par sa longueur. Une vidéo ou un gros téléchargement, eux, réclament bien de la largeur.

Quand vous codez, comptez donc les allers-retours avant les octets : les cent une requêtes vers la base du N+1 de la 4.6, à 60 ms chacune, coûtent six secondes, alors qu'un seul appel qui les regroupe n'en coûte qu'une soixantaine de millisecondes.

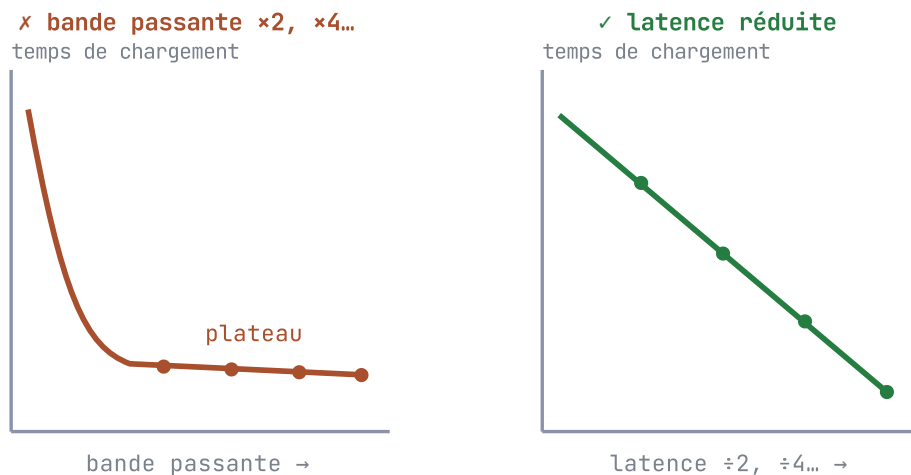


Figure 37 : La mesure de Belshe : passé quelques mégabits, élargir la route ne sert plus à rien. La raccourcir, toujours.

↳ High Performance Browser Networking

5.2 Le sablier : un contrat minuscule au milieu

Que devient un message pendant ces 60 ms de voyage ? D'abord, le réseau ne transporte jamais un message d'un bloc. Il le découpe en **paquets** : des petits morceaux indépendants, comme les pages d'un livre expédiées dans des enveloppes séparées. Chaque paquet porte l'adresse de destination et passe de proche en proche par des **routeurs**, les relais qui l'orientent, dont le premier est la box de votre fournisseur d'accès. Deux paquets du même message peuvent prendre des routes différentes et arriver dans le désordre.

Ensuite, le génie d'Internet est dans ce qu'il ne promet PAS. Au milieu du réseau, toutes les machines partagent un seul **protocole** commun, une règle de conversation qu'elles acceptent toutes : **IP** (Internet Protocol), qui donne une adresse à chaque machine et fait avancer les paquets vers elle. Son contrat

est minuscule: «je fais de mon mieux» (*best-effort*). Un paquet peut être perdu, dupliqué, retardé, remis dans le désordre, et IP n'a rien promis d'autre que d'essayer.

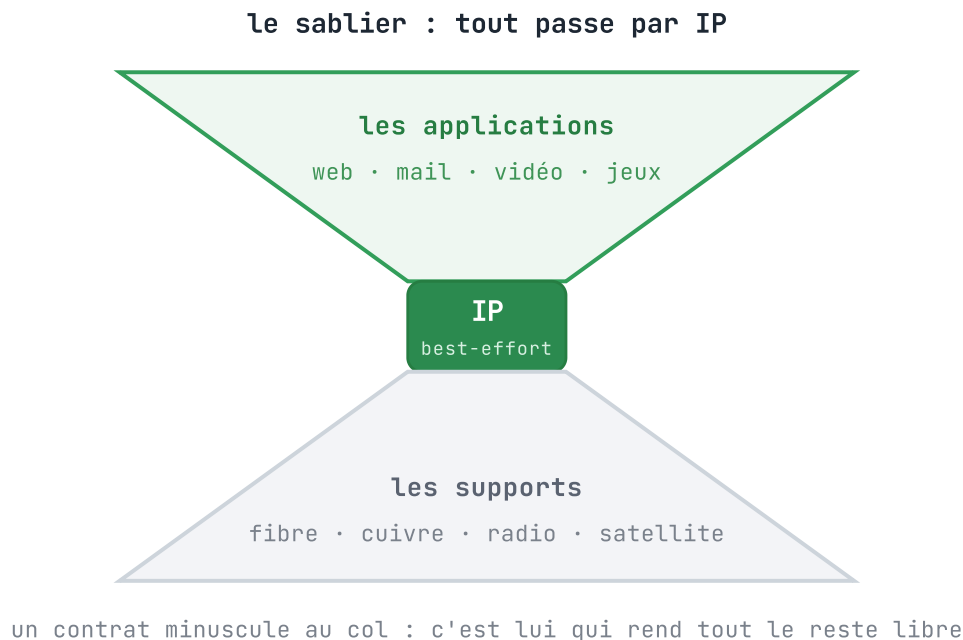


Figure 38 : En bas, les supports changent depuis 50 ans. En haut, les applications aussi. Le col, lui, n'a pas bougé: IP et son « je fais de mon mieux ».

Pourquoi promettre si peu ? Parce qu'un milieu simple est un milieu universel. La fiabilité se garantit **aux extrémités**, chez l'expéditeur et le destinataire, pas au milieu : eux seuls savent si le message est arrivé entier (c'est l'argument dit *end-to-end*, formulé par Saltzer, Reed et Clark). Le col du sablier reste assez pauvre pour courir sur n'importe quel support, y compris ceux qui n'existaient pas à sa naissance : la fibre, la 5G et le satellite sont arrivés sans que son contrat change. Vous y reconnaîtrez le principe de la 4.2, « programmer pour une interface, pas pour une implémentation » : IP est l'interface, et de part et d'autre, applications comme supports physiques, chacun reste libre de varier.

Ce contrat minuscule remonte jusque dans votre code. Un appel réseau n'est pas un appel de fonction : il peut échouer, traîner, ne jamais répondre, et ce n'est pas un accident, c'est le contrat posé au col du sablier. D'où trois réflexes de développeur, dès la première ligne qui traverse le réseau. Fixer un délai au-delà duquel on n'attend plus : c'est le **timeout**. Prévoir de réessayer, mais sans appliquer l'opération deux fois : un délai dépassé ne dit pas si le virement est

parti, et un virement renvoyé ne doit pas être débité deux fois. Une opération qu'on peut répéter sans effet double est dite **idempotente**. Et on réessaie poliment, en attendant de plus en plus longtemps entre deux essais (1 s, 2 s, 4 s), faute de quoi mille clients qui retentent d'un coup refabriquent la panne (5.4). Enfin, prévoir le message d'erreur, plutôt qu'une page qui tourne sans fin.

↳ Computer Networks: A Systems Approach

5.3 TCP : fabriquer du fiable sur du peu fiable

IP fait de son mieux, mais son mieux ne suffit pas à tout : un virement bancaire ne peut pas se permettre de se perdre en route. Il faut donc combler l'écart entre ce qu'IP promet et ce dont une application a vraiment besoin. C'est le rôle de **TCP** (*Transmission Control Protocol*), installé dans les deux machines des extrémités, juste au-dessus d'IP. Son mécanisme copie le recommandé postal. Chaque octet envoyé porte un **numéro de suivi**. Le destinataire répond par un **accusé de réception** pour dire ce qui lui est bien parvenu. Et si un accusé n'arrive pas dans le délai prévu, l'expéditeur **renvoie le paquet**, sans chercher à savoir pourquoi il s'est perdu.

Numéro, accusé, renvoi : ces trois gestes réparent la perte, le doublon (le numéro permet de l'écarter) et le désordre (il permet de remettre les paquets en ordre). L'application au-dessus, elle, n'en voit rien. Elle croit rouler sur une route parfaite, alors que le réseau en dessous ne l'est pas du tout. C'est la fiabilité fabriquée aux extrémités, exactement comme le voulait le sablier.

Ce service a deux prix. Le premier se paie à l'ouverture. Avant le moindre octet utile, les deux machines échangent une **poignée de main en trois temps** : « je veux parler », « reçu, moi aussi », « reçu » (SYN pour « synchroniser », ACK pour « acknowledge », accuser réception : SYN, SYN-ACK, ACK sur le schéma). Cette poignée de main coûte un aller-retour complet, donc 60 ms perdues sur Paris–New York avant même de commencer. Une **connexion** est la conversation que deux machines ouvrent par une poignée de main, chacune gardant en mémoire où elle en est. Et chaque nouvelle connexion repaie ce prix.

Le second prix est plus subtil : à quel rythme envoyer ? Attendre l'accusé de chaque paquet laisserait la route presque vide. TCP envoie donc toute une file de paquets, des camions sur notre autoroute, avant d'avoir des nouvelles du premier, plutôt qu'un camion, silence, puis le suivant. On appelle **fenêtre** le nombre de paquets autorisés à rouler sans nouvelles.

La bonne taille de la fenêtre est donnée par le **produit bande passante × latence** : on multiplie la bande passante par le temps d'aller-retour, et on obtient ce que la route contient à cet instant, tous les camions partis dont la confirmation n'est pas encore revenue. Sur 100 Mbit/s, soit 12,5 Mo par seconde (un octet compte 8 bits), avec 60 ms d'aller-retour entre Paris et New York, environ 750 ko voyagent en permanence, déjà partis, pas encore confirmés. Mais TCP ne connaît pas ce chiffre à l'avance : il doit le deviner en chemin (5.4).

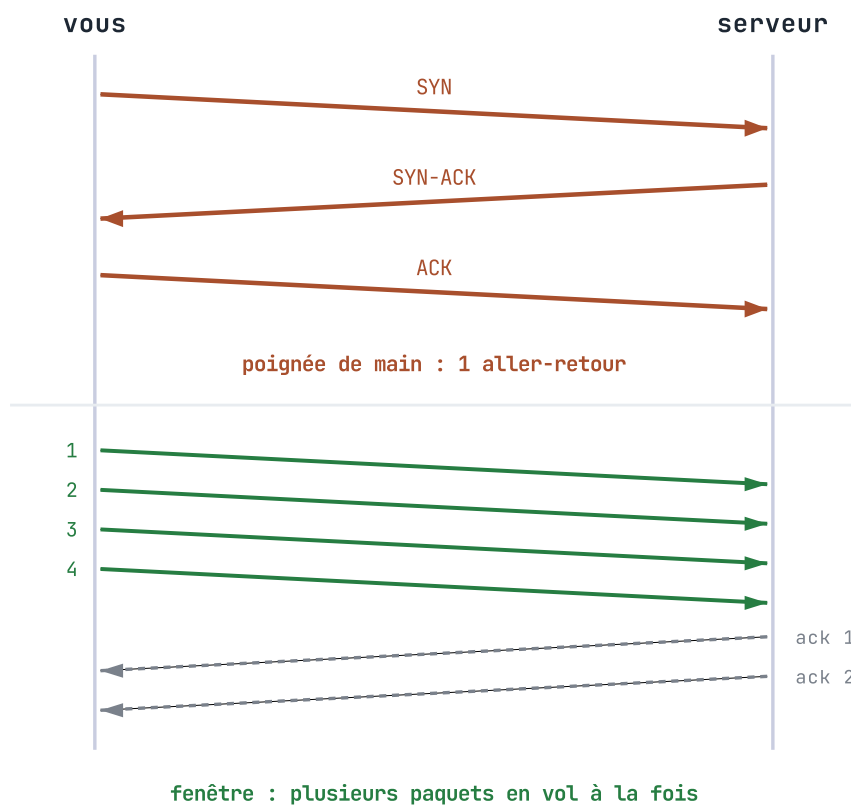


Figure 39 : En orange, la poignée de main coûte un aller-retour avant le premier octet utile. En vert, les paquets numérotés partent sans attendre un accusé par paquet (en gris pointillé) : plusieurs voyagent de front, c'est la fenêtre.

Grâce à TCP, vous ne gérez jamais un paquet perdu ou arrivé dans le désordre : votre code reçoit les octets réparés et dans l'ordre, ou une erreur si la connexion casse. Le revers : un paquet perdu retient tous ceux qui le suivent, même déjà arrivés, le temps de son renvoi (c'est le blocage en tête de file, en anglais *head-of-line blocking*). Et il reste deux coûts que le code ne supprime pas, il peut seulement les amortir : la poignée de main (réutiliser une connexion déjà ouverte évite de la repayer) et la fenêtre (une connexion qui a déjà trouvé la bonne taille de fenêtre ne repart pas de zéro).

↳ [Computer Networks: A Systems Approach](#) · [High Performance Browser Networking](#)

5.4 La congestion : partager sans chef

La fenêtre de TCP laisse une question ouverte (5.3) : de quelle taille la prendre ? Des millions de connexions se partagent les mêmes câbles, et personne ne distribue les tours de parole. Chaque routeur garde les paquets en attente dans une file, comme les voitures avant un péage. Si tout le monde envoie à fond, la file déborde et le routeur jette les paquets en trop. Chacun renvoie alors ce qu'il a perdu, ce qui rallonge encore le bouchon.

L'**effondrement de congestion** est le cercle vicieux où les paquets renvoyés rallongent le bouchon qui les a fait perdre. Il est arrivé pour de vrai : en 1986, un lien entre l'université de Berkeley et le laboratoire voisin de Lawrence Berkeley est passé de 32 kbit/s utiles à 40 bit/s, soit 800 fois moins. Aucun câble n'était coupé, aucune machine n'était en panne, et pourtant plus rien n'arrivait. L'ingénieur Van Jacobson en a tiré les règles qui suivent.

La parade est une règle de conduite que chaque machine s'applique à elle-même, sans arbitre. Accélérer **doucement**, un cran de fenêtre de plus à chaque aller-retour. Puis, au premier paquet perdu, **diviser sa vitesse par deux** d'un coup. C'est le comportement d'un conducteur prudent : on reprend de la vitesse petit à petit, et on lève le pied dès qu'on voit des feux stop devant, sans

savoir ce qui bloque plus loin. La perte n'est plus seulement un accident à réparer : elle devient le **signal** que la route sature, et chacun freine tout seul en la voyant.

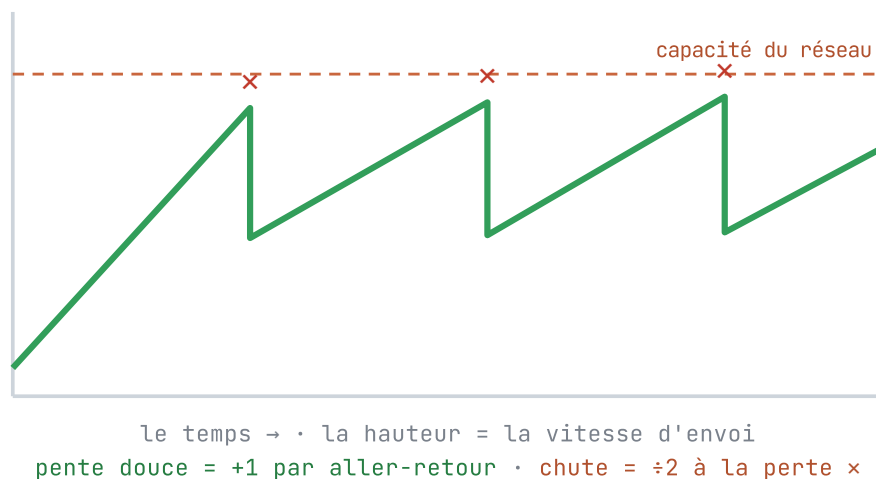


Figure 40 : Monter doucement, diviser par deux à la perte : les dents de scie frôlent la capacité sans s'effondrer. Internet tient parce que chaque machine freine toute seule.

Cette politesse codée s'appelle **AIMD** (*Additive Increase, Multiplicative Decrease* : hausse par addition, baisse par division). Elle a un piège : des files de routeur très longues retardent la perte, donc le signal, et la latence gonfle bien avant que quiconque ne freine (c'est le *bufferbloat*). Les variantes modernes affinent la courbe (CUBIC remonte plus vite, BBR mesure la route au lieu d'attendre la perte), mais le principe n'a pas bougé depuis l'article de Van Jacobson en 1988 : un bien commun sans gendarme, tenu par la retenue de chacun.

Cette prudence, vous l'avez déjà vue à l'œuvre. Lancez un gros téléchargement : la vitesse affichée grimpe pendant les premières secondes au lieu de démarrer au maximum. La connexion ne connaît pas encore la capacité de la route, alors elle commence petit (une dizaine de paquets, environ 14 ko) et double sa fenêtre à chaque aller-retour : c'est le **slow start** (démarrage lent, mais seulement par rapport à tout envoyer d'un coup).

Retenez ce chiffre : une page qui tient dans ces 14 ko, compressée (réduite avec gzip ou Brotli), arrive d'un seul envoi, sans attendre d'accusé. Au premier incident, la connexion repasse à la hausse prudente, cran par cran, vue plus haut.

Une connexion déjà chaude, qui vient de servir et a élargi sa fenêtre, démarre bien plus vite (laissée au repos, elle l'oublie) : d'où l'intérêt d'ouvrir le moins de connexions possible, ce que la section suivante va chiffrer.

↳ [Computer Networks: A Systems Approach · High Performance Browser Networking](#)

5.5 Le web paie le péage

HTTP (*HyperText Transfer Protocol*) est la langue que parlent navigateurs et serveurs : demander une page, répondre avec son contenu. Mais une requête HTTP ne voyage jamais seule : c'est du texte confié à une connexion TCP, elle-même acheminée paquet par paquet par IP. Prenons un visiteur qui tape l'adresse de votre site. Avant le premier octet de la page, tout ce que ce chapitre vient de poser se paie, aller-retour par aller-retour : c'est le péage du web. Quatre étapes se succèdent, chacune attendant la précédente.

- › **Le DNS** (*Domain Name System*) : l'annuaire du web traduit le nom du site en adresse IP, parce qu'un routeur sait acheminer vers une adresse, jamais vers un nom. Un aller-retour vers un serveur de noms, souvent proche, et aucun si la réponse est déjà en mémoire.
- › **La poignée de main TCP** : les deux machines se mettent d'accord pour ouvrir une connexion fiable (5.3). Un aller-retour.
- › **TLS** (*Transport Layer Security*) : le chiffrement, le cadenas du https (le « s » veut dire « sécurisé »). Le serveur prouve son identité par un certificat, puis les deux machines s'accordent sur une clé secrète, pour que personne, sur tout le trajet, ne puisse lire ni modifier la suite. Un aller-retour avec TLS 1.3, deux avec TLS 1.2.
- › **La requête** : enfin, le navigateur demande la page. Un dernier aller-retour avant le premier octet utile.

Sur 60 ms de latence, comptez au pire de 240 à 300 ms avant la première réponse (selon la version de TLS). Sur mobile, ajoutez le réveil de la radio : pour économiser la batterie, la radio du téléphone coupe sa liaison avec l'antenne dès qu'elle ne sert plus, et doit la rétablir avant d'envoyer quoi que ce soit, ce qui coûte des dizaines à des centaines de millisecondes de plus.

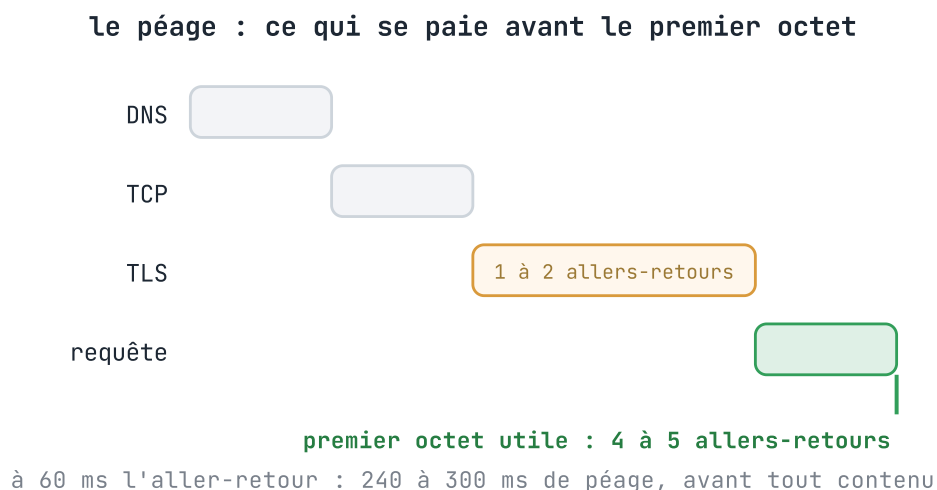


Figure 41 : Chaque étape attend la précédente : le péage s'additionne en allers-retours. C'est lui que HTTP/2 puis HTTP/3 s'emploient à raboter.

Ce péage explique une décennie de bricolages. **HTTP/1** ne sait demander qu'une ressource à la fois par connexion, et un navigateur n'en ouvre que six par domaine (par nom de site, comme `monsite.fr` ou `images.monsite.fr`). Les développeurs ont donc rusé dans deux directions. Réduire le nombre de requêtes d'abord, en collant les fichiers bout à bout (le *bundling*) et en rassemblant toutes les icônes dans une seule image (un *sprite*). Contourner la limite ensuite, en éparpillant les ressources sur plusieurs domaines pour avoir droit à six connexions de plus à chaque fois.

HTTP/2 a réglé le problème des connexions : toutes les requêtes voyagent entrelacées sur **une seule** connexion, donc une seule poignée de main, et une fenêtre qui reste ouverte et large. Ces vieilles ruses servent peu, ou nuisent : éparpiller sur plusieurs domaines rouvre précisément les connexions que le protocole venait d'économiser.

HTTP/3 va plus loin encore : il abandonne TCP pour un transport neuf, **QUIC** (prononcé « quick »), parce qu'une seule connexion TCP a un défaut. Un paquet perdu fait attendre toutes les requêtes rangées derrière lui, même celles

dont les paquets sont déjà arrivés. QUIC, bâti sur UDP (un envoi de paquets sans accusé de réception, dont QUIC refait lui-même la fiabilité), retransmet chaque requête séparément et fond aussi la poignée de main et le chiffrement en un seul aller-retour au lieu de deux ou trois.

La morale du chapitre est la règle d'or de la performance web : **la requête la plus rapide est celle qu'on ne fait pas**. Trois gestes en découlent, et chacun répond à une leçon de ces cinq sections. Mettre en cache, pour ne pas redemander ce qu'on a déjà : une ligne de réglage jointe à la réponse du serveur (un en-tête), par exemple `Cache-Control: max-age=31536000, immutable`, dit au navigateur qu'il peut garder le fichier un an sans le redemander. Avec un nom de fichier qui change à chaque version, une nouvelle version atteint quand même tout le monde. Réutiliser une même connexion, pour ne payer qu'une fois la poignée de main et le chiffrement. Rapprocher le contenu du visiteur, enfin, avec un **CDN** (Content Delivery Network), c'est-à-dire des serveurs relais répartis dans le monde, puisque aucun abonnement ne raccourcit l'Atlantique. Le cache et le CDN reviennent au [6.6](#), à l'échelle du service entier.

↳ [High Performance Browser Networking](#)

↳ CE QUI MÈNE AU [CHAPITRE 6](#)

Cinq lois se dégagent. La distance se paie en allers-retours, et ni l'argent ni le code ne les raccourcissent. Au milieu, IP promet seulement de faire de son mieux, et TCP rebâtit aux extrémités la fiabilité qui manque. Faute de chef, chaque machine freine d'elle-même dès qu'un paquet se perd. Enfin, chaque page web paie son péage de DNS, de poignée de main et de chiffrement avant le premier octet. Une même leçon les relie : le réseau est lent, faillible et partagé, et le code doit le supposer.

Mais deux machines qui savent se parler ne font pas encore un service qui tient. Une architecture, même élégante, n'est qu'une hypothèse tant qu'on ne l'a pas mise à l'épreuve : la règle du [4.3](#), qui fait pointer les dépendances vers les règles métier, ne se vérifie pas à l'œil : il faut un test qui échoue dès qu'une frontière est franchie. Prouver, livrer, tenir sous la charge réelle : la forme posée au [chapitre 4](#) doit affronter le monde.

« Les discours ne coûtent rien. Montre-moi le code. » — Linus Torvalds, 2000

6 Prouver, livrer, tenir la charge

le flux

Du code qui « marche sur ma machine » à un service qui tient en **production**, la machine ouverte au public où des milliers de gens l'utilisent pour de vrai, il y a un gouffre. Le franchir, ce n'est pas coder plus : c'est installer un **flux** qui prouve, livre et tient, sans héroïsme et sans nuits blanches.

Sept maillons composent ce flux. Prouver d'abord : écrire le test avant le code, puis traquer les bugs sans rien croire sur parole. Livrer ensuite : garder l'historique de tout ce qu'on écrit, faire circuler le travail au lieu de le stocker, et livrer si souvent que ça n'a plus rien d'un événement. Tenir la charge enfin : encaisser un million de lecteurs, et découvrir ce que coûtent les garanties quand les données ne tiennent plus sur une machine.

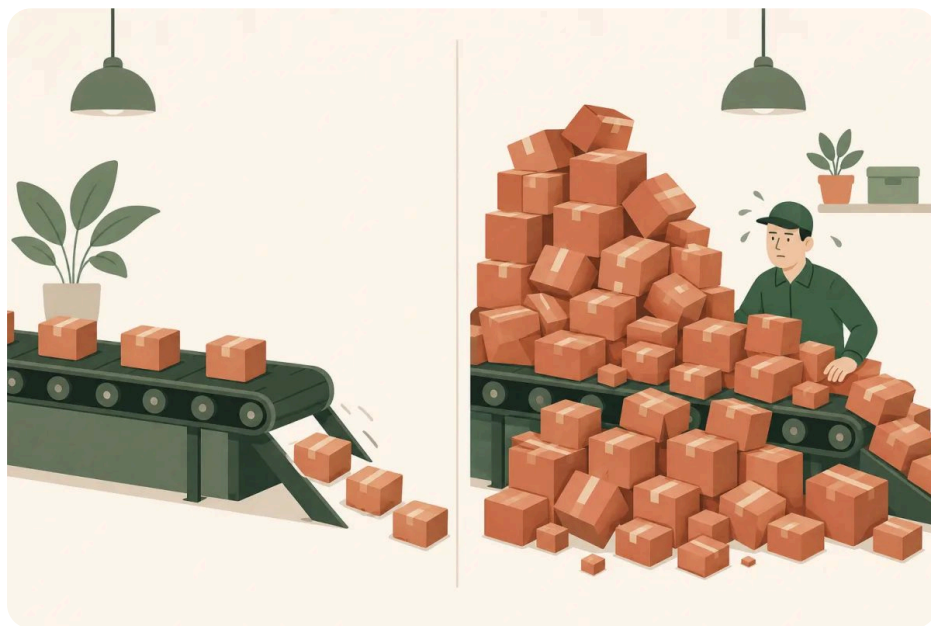


Figure 42 : Le flux régulier livre sans effort. La file saturée fige tout, même quand chacun est à fond.

6.1 Le test d'abord

Le test de 3.6 appelait le code après coup, pour vérifier qu'un refactoring n'avait rien cassé. Écrivons-le avant le code, « `somme(5, 3)` doit rendre 8 », et ça change tout : on définit le résultat attendu avant de savoir comment l'obtenir. Le cycle se répète à l'infini : **Rouge** (un test qui échoue), **Vert** (le code le plus bête qui le fait passer, même en dur), **Refactor** (nettoyer sans casser).

```
// 1. ROUGE : le test AVANT le code
test('5 + 3 = 8', () => expect(somme(5, 3)).toBe(8)) // x échoue

// 2. VERT : le code le plus bête qui passe (oui, "8" en dur)
function somme(a, b) { return 8 }
// ✓ vert, sans honte

// 3. ROUGE puis VERT : un 2e test casse le "8", on généralise
function somme(a, b) { return a + b }
// ✓ vert pour les deux tests
```

le cycle, sans fin

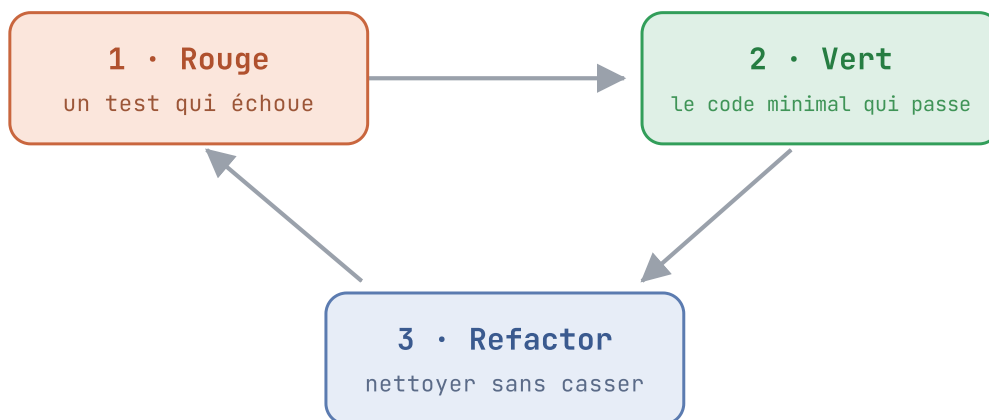


Figure 43 : On boucle sans fin : un test qui échoue, le code minimal qui le fait passer, puis on nettoie, et on enchaîne le suivant.

On ne nettoie jamais le code tant qu'un test échoue : d'abord le vert, ensuite le ménage. Le but, en quatre mots : du code propre qui marche.

Quel test écrire ensuite ? Les mêmes cas reviennent pour toute fonction. Prenons le `total_panier` de 2.1 :

- › rien : un panier vide doit rendre 0 ;

- › un seul : un article rend son prix ;
- › plusieurs : trois articles rendent leur somme ;
- › les bornes : 99,99 €, 100 € et 100,01 €, autour du seuil de remise. Le test à 100 € pose une question que personne n'avait posée au vendeur, « dépasse » ou « atteint », et c'est le cas limite qui la pose, pas le code ;
- › l'invalidé : une quantité négative ou un prix en texte doit échouer tôt ([3.4](#)) plutôt que rendre un total absurde.

Un test par cas, chacun nommé par ce qu'il vérifie : la liste des tests devient la description la plus honnête de ce que fait la fonction.

L'effet le plus profond du **TDD** (Test-Driven Development, le développement piloté par les tests) n'est pourtant pas d'attraper des bugs, c'est le **design émergent** : en écrivant le test d'abord, on conçoit l'API, ce guichet de la [section 4.5](#), du point de vue de celui qui vient y déposer sa demande, pas de celui qui tient le bureau. Le code devient modulaire et découplé parce qu'il faut qu'il soit testable.

On ne dessine pas l'architecture en avance, on la laisse apparaître, test après test. Les fonctions courtes du [chapitre 3](#) et les dépendances propres du [chapitre 4](#) deviennent alors des conséquences, pas des efforts de volonté.

Reste la difficulté qui décourage les débutants : comment tester une fonction qui envoie un mail, écrit dans la base ou appelle un service distant ? Un test qui touche le vrai monde est lent, il échoue pour des raisons qui n'ont rien à voir avec votre code (le réseau, l'heure, la base de quelqu'un d'autre), et il envoie de vrais mails.

La réponse est la prise électrique de la [section 2.7](#) : au lieu de brancher le vrai service dans la fonction, on le lui passe en paramètre. Michael Feathers appelle ça une **couture**, un endroit où l'on change le comportement sans modifier la fonction elle-même :

```
function inscrire(email, envoyer) {  
  // envoyer arrive de l'extérieur : c'est la couture  
  envoyer(email, "Bienvenue !");  
}  
// en production : inscrire(mail, vraiEnvoiDeMail)  
const appels = [];  
// en test : un faux appareil note ce qu'on lui demande  
inscrire("a@b.fr", (to, msg) => appels.push([to, msg]));  
// → appels vaut [{"a@b.fr", "Bienvenue !"}] : rien n'est parti, le test  
// est instantané
```

Un faux appareil branché sur la prise s'appelle un **mock** (un faux qui répond ce qu'on lui a dit de répondre et qui, en plus, note ce qu'on lui demande). C'est l'inversion de dépendance de la [section 4.3](#) vue côté test : du code qu'on peut tester sans le monde est du code dont les dépendances sont propres. Deux limites. Avec trop de faux, on teste le décor et non le code. Et devant du vieux code qui n'a pas de tests, on écrit d'abord un **test de caractérisation** : il note ce que le code fait vraiment, pas ce qu'il devrait faire, avant qu'on le change. Pour Feathers, du code sans tests est déjà du *legacy* : du code qu'on n'ose plus toucher, quel que soit son âge.

Un test qui isole une fonction derrière des faux est un **test unitaire**. Celui qui branche les vrais morceaux ensemble, base de données comprise, est un test d'intégration, et celui qui joue le parcours entier, du clic au ticket, un test de bout en bout : plus lents et plus fragiles dans cet ordre, donc beaucoup du premier, quelques-uns du deuxième, une poignée du troisième.

L'heure et le hasard sont deux entrées cachées du même genre : une remise « jusqu'à minuit » testée à 23 h 59 passe, à 00 h 01 échoue, sans qu'une ligne ait changé. Un test doit rendre la même réponse à chaque exécution, alors on passe l'horloge et le dé en paramètre, comme `envoyer` plus haut : c'est encore la couture.

L'économie confirme la discipline : le même défaut coûte dix à cent fois plus cher quand on le découvre en production plutôt qu'ici, au clavier (Code Complete). Tester tôt n'est pas du zèle, c'est le chemin le moins cher.

↳ [Test-Driven Development: By Example](#) · [Working Effectively with Legacy Code](#) · [Code Complete](#)

6.2 Déboguer, c'est refuser de croire

Les tests attrapent la plupart des bugs. Les survivants sont ceux qui ont l'air impossibles. Un programmeur qui peut se connecter assis mais jamais debout. Un terminal bancaire de Chicago qui plante dès qu'un client tape « Quito ». Si un bug a l'air impossible, c'est qu'une de vos certitudes est fausse, et vous ne savez pas encore laquelle. C'est tout ce que dit la règle de Bentley : « déboguer, c'est le plus souvent refuser de croire ». Refuser de croire ce qu'on tient pour acquis, la ligne qu'on a relue dix fois, la configuration qu'on est sûr d'avoir déployée, la donnée qu'on suppose propre. Tant que ces certitudes ne sont pas vérifiées une par une, l'explication reste cachée derrière l'une d'elles.

Rick Lemons, cité par Bentley, racontait que sa meilleure leçon de débogage avait été un spectacle de magie. Six tours impossibles d'affilée, et pas un seul qui le soit vraiment: le magicien attirait simplement le regard du côté où il ne se passait rien. Un bug impossible marche pareil, il suffit de regarder l'autre main.



Figure 44 : Un bug « impossible » marche comme un tour de magie : on fixe la main qui brille, et c'est dans l'autre que ça se passe.

Nos deux exemples se dénouent alors tout seuls. Le programmeur, assis, tapait son mot de passe sans regarder le clavier. Debout, il regardait les touches, et deux d'entre elles avaient été interverties. Quant au terminal, il vérifiait si la saisie commençait par `quit`, sa commande pour quitter: « Quito » commence par « quit ».

Votre bug impossible à vous ressemblera plutôt à ceci : tout marche sur votre machine, et rien ne marche en production. Arrêtez de fixer le code qui « ne peut pas » être faux, et posez la seule question utile : qu'est-ce qui change entre les deux ? La réponse est presque toujours ailleurs que dans le code. La **configuration** est un réglage que le programme lit au démarrage au lieu de le porter dans son code, et ses valeurs diffèrent souvent entre les deux machines. Autres suspects : un cache qui ressort une vieille réponse, deux requêtes qui arrivent dans l'ordre inverse, une valeur oubliée au moment de la mise en ligne. Cette

question ne se tranche pas de mémoire. On la lit dans ce que l'application a écrit sur elle-même pendant qu'elle tournait, son journal. Comment le tenir, c'est l'affaire de la [section 6.5](#).

Refuser de croire se pratique avec méthode. Reproduire d'abord : tant qu'on ne sait pas faire apparaître le bug à volonté, on ne saura pas non plus prouver qu'il a disparu. Réduire ensuite : on retire des articles du panier, des lignes du fichier, des étapes du parcours, jusqu'au plus petit cas qui échoue encore, car un bug nu se lit, un bug noyé dans mille lignes non. Le terminal de Chicago : « Quito » plante, « Quit » plante aussi, « Qui » ne plante pas, et la frontière désigne la ligne coupable. Puis une hypothèse à la fois, vérifiée avant la suivante, au lieu de changer trois choses d'un coup sans plus savoir laquelle a agi.

Le **débogueur** arrête le programme sur une ligne choisie, le **point d'arrêt**, et montre le contenu de chaque casier : c'est le déroulé à la main de [2.1](#), exécuté par la machine. Quand le bug est apparu « un jour » sans qu'on sache lequel, Git cherche à votre place : `git bisect` rejoue l'historique par moitiés, comme l'annuaire de [1.3](#), et trouve en une dizaine d'essais le commit fautif parmi mille ([6.3](#)). Le dernier geste est le premier de [6.1](#) : écrire le test qui reproduit le bug, Rouge, avant de corriger. Le bug corrigé entre au filet et ne reviendra plus sans se faire voir.

Une chose marche étonnamment bien dès qu'on est coincé : expliquer le bug à voix haute, à un collègue ou même à un canard en plastique. McConnell appelle ça le *confessional debugging* : on trouve souvent la réponse au milieu de sa phrase, avant que l'autre ait dit un mot. Reconstruire la chaîne causale pour quelqu'un d'autre force le cerveau à passer du mode « chercher » au mode « comprendre ».

↳ [Programming Pearls](#) · [Code Complete](#)

6.3 Versionner, c'est un système adressable par contenu

Le versioning est le **filet** du flux : on change tout, on tente, on travaille à plusieurs sans se marcher dessus, et on peut toujours revenir en arrière. La preuve par la peur : vous venez d'effacer trois jours de commits d'un `git reset --hard` mal visé. Un commit est une photo datée de tout le projet, avec le nom de son auteur : l'unité de base de Git. Panique. Sauf que dans Git, presque rien ne disparaît, et comprendre pourquoi change tout.

Git n'est pas un dossier d'historique : c'est une base d'objets, chacun adressé par son **empreinte** : un condensé court calculé à partir du contenu (le « hash », produit par la fonction SHA-1). Le même contenu donne toujours le même hash, donc rien ne se perd ni ne se falsifie en silence. Tout s'enchaîne par pointeurs :

un graphe d'objets, chacun nommé par son hash

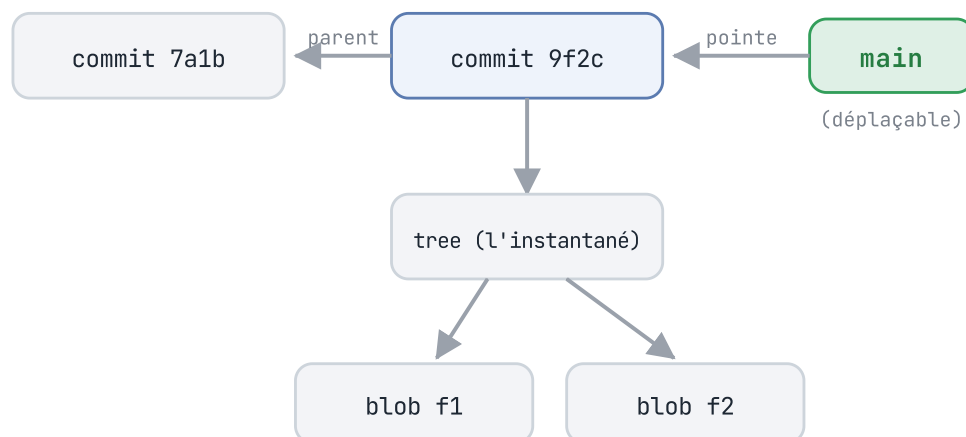


Figure 45 : Tout est un objet immuable, nommé par son empreinte. Une branche n'est qu'un pointeur qu'on déplace. Rien ne s'efface vraiment.

Cinq mots suffisent pour lire ce schéma. Un **blob** est le contenu d'un fichier. Un **tree** est la liste des fichiers d'un dossier à un instant donné. Un **commit** pointe sur un tree et sur son parent. Une **branche**, ici `main`, n'est qu'un pointeur vers un commit, qu'on avance à chaque nouveau commit. Et `HEAD` est le pointeur « vous êtes ici » : le commit sur lequel vous travaillez.

```
git cat-file -p HEAD
# affiche ce commit : son tree, son parent, l'auteur
```

Avant d'entrer dans ce graphe, un fichier traverse **trois zones**, ce qui explique pourquoi `git add` existe :

trois zones : pourquoi git add existe



Figure 46 : `git add` photographie une version pour le prochain commit. `git commit` la scelle dans le graphe. D'où la possibilité de ne commiter qu'une partie.

`git add` ne sauvegarde rien : il photographie la version exacte d'un fichier pour le prochain commit. Le commit scelle cette photo dans le graphe. C'est pour ça qu'on peut commiter une partie de ses changements seulement.

Reste la promesse du début : travailler à plusieurs sans se marcher dessus. Chacun avance sa propre **branche**, un pointeur parmi d'autres, sans toucher à celle des autres. Quand le travail est prêt, une **fusion** (*merge*) recolle les deux histoires : Git crée un commit à deux parents, un par branche.

Entre la branche et la fusion, une étape que Git ne fait pas : quelqu'un lit. On propose sa branche, ce qu'on appelle une *pull request* (une demande de fusion), avec la liste exacte des lignes changées, et un collègue la relit avant qu'elle soit recollée. C'est la **revue de code** : le prochain humain du [chapitre 3](#) arrive plus tôt que prévu, pas dans six mois mais demain matin. Les chiffres lui donnent raison. Dans *Code Complete*, McConnell rapporte que les inspections de code

formelles trouvent environ six défauts sur dix, plus que la plupart des tests. Une revue de pull request est plus légère, mais va dans le même sens. Et Accelerate a constaté que les équipes qui se relisent entre pairs livrent plus vite que celles qui attendent le feu vert d'un comité extérieur, lequel ralentit sans rendre plus sûr. Quand on code : une branche courte se relit en dix minutes, une branche d'un mois ne se relit pas, on la signe les yeux fermés.

Vos trois jours effacés ? Le commit d'avant le `reset` est toujours là, objet immuable dans la base. `git reflog` liste les hash récents (Git les garde environ trois mois), vous repointez une branche dessus, et tout revient. Le filet n'attrape que ce qui est commité : une modification jamais commitée est, elle, vraiment perdue. D'où le réflexe : commiter souvent. Comprendre le graphe, c'est livrer sans paniquer.

Le filet a un revers : rien ne disparaît, pas même ce qu'on voudrait voir disparaître. Un mot de passe de base de données commité par erreur reste lisible dans un objet du graphe, pour quiconque clone le dépôt, même effacé du fichier le lendemain. Un **secret**, mot de passe ou clé d'une API, est une valeur qui ne vit jamais dans le code : elle arrive par la configuration, une variable d'environnement ou un fichier que Git ignore. Et un secret commité une seule fois ne s'efface pas, il se change.

↳ [Pro Git](#) · [Code Complete](#) · [Accelerate](#)

6.4 Le flux bat l'effort

The Phoenix Project enseigne par le roman une loi d'usine implacable : le temps d'attente d'une tâche **explose** à mesure qu'une **ressource** (un serveur, une équipe, une personne dont tout dépend) s'approche de 100 % d'occupation.

```
temps d'attente ≈ % occupé ÷ % libre // la loi d'attente
```

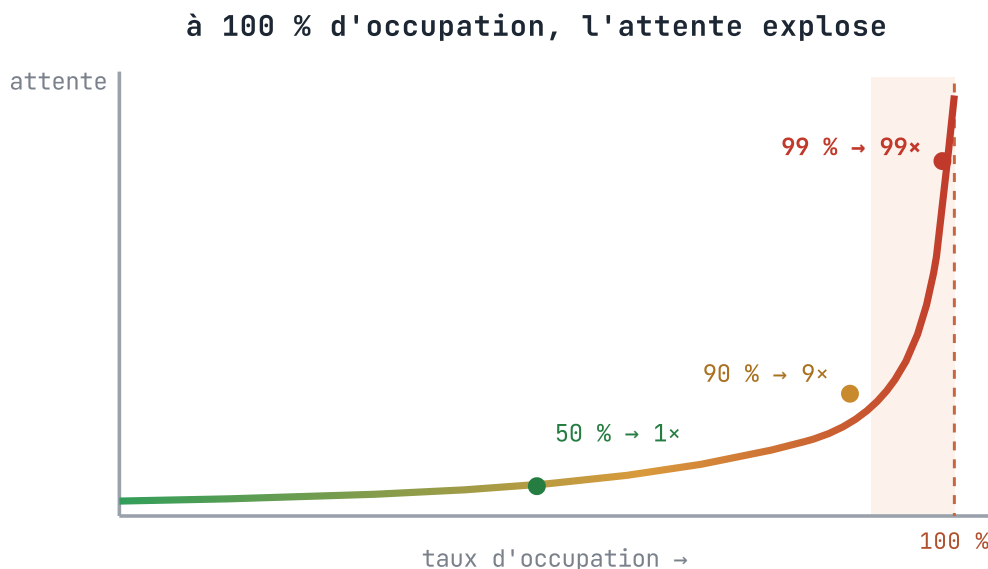


Figure 47 : Presque plat, puis vertical : $50/50 = 1\times$, $90/10 = 9\times$, $99/1 = 99\times$. La marge n'est pas du gâchis, c'est elle qui fait circuler le travail.

Pourquoi ce mur ? À 99 % d'occupation, plus aucune marge n'absorbe l'imprévu : une tâche qui traîne, un pic d'arrivées, et la file enfle sans jamais se résorber. À 50 %, le temps libre lisse ces à-coups au fil de l'eau.

« Tout le monde est à fond » et « rien n'avance » sont donc la même phrase. Le remède est contre-intuitif : limiter le travail en cours, arrêter d'en commencer pour enfin en finir. La marge n'est pas du gâchis : c'est elle qui fait circuler le travail.

Un second flux se cache sous le même mot : celui de votre tête. Concevoir ou coder demande quinze minutes de montée pour s'installer, et une interruption ne coûte pas cinq minutes, elle coûte toute la remontée, près de vingt (DeMarco et Lister l'ont mesuré dans *Peopleware*). La marge protège ce flux aussi : un agenda calé à 100 %, comme un serveur à 100 %, ne laisse plus rien avancer vraiment.

La donnée tranche : dans les Coding War Games (un tournoi de centaines de développeurs mené par les mêmes auteurs), le plus gros écart de productivité ne tenait ni au langage ni à l'expérience, mais à l'environnement. Au calme et peu interrompus, les meilleurs avaient un tiers de chances en plus de rendre un travail sans un seul bug. C'est l'environnement qui fait l'écart, pas le talent.

↳ [The Phoenix Project](#) · [Peopleware](#)

6.5 Déployer souvent fait moins peur que déployer rarement

Le grand déploiement préparé pendant des mois est un tir de canon : une fois le boulet parti, plus rien ne se corrige, et la cible a eu des mois pour bouger. La rafale de petites livraisons fait l'inverse : chaque coup montre où il atterrit, et le suivant rectifie le tir.

La **CI/CD** est la chaîne automatique qui rend la rafale possible. L'intégration continue (CI) est un tapis roulant : à chaque commit, il rejoue l'assemblage et tous les tests, et personne ne fusionne du rouge. La livraison continue (CD) prolonge ce tapis : chaque version au vert est emballée, prête à partir en production d'un clic. Déployer cesse d'être un événement. C'est la sortie normale du tapis.

Reste à prouver que la rafale bat le canon. *Accelerate*, le livre de Nicole Forsgren, Jez Humble et Gene Kim, le mesure : les équipes qui déploient souvent et avec un **couplage faible** (livrer sans demander la permission à une autre équipe) sont à la fois plus rapides et plus stables. Ce couplage faible, c'est le quantum du [4.4](#). Le compromis attendu (aller vite = casser plus) n'existe pas dans les données.

On y arrive avec des outils qui rendent l'erreur bon marché : les *feature flags* (déployer du code éteint, qu'on allume d'un interrupteur, qu'on rééteint au premier souci) ou le déploiement *blue-green* (deux versions prêtes côte à côte, un aiguillage réversible de l'une à l'autre). Les deux reposent sur le même principe, posé dès 2010 par Jez Humble et David Farley dans *Continuous Delivery* : déployer (mettre le code en place) n'est pas livrer (le rendre visible aux utilisateurs). C'est ça qui enlève la peur : une bêtise se répare d'une bascule, pas d'une nuit blanche.

```
if (flags.nouveauPanier) {           // une valeur de config, pas du code
    afficherNouveauPanier();
    // le code est déjà déployé, mais éteint
} else {
    afficherAncienPanier();
}
// allumer = changer flags.nouveauPanier dans la config, sans redéployer
```

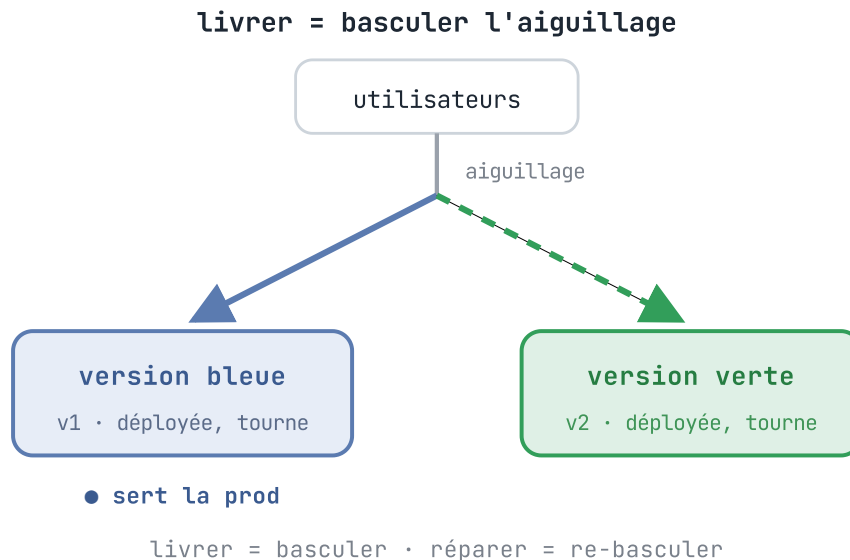


Figure 48 : Déployer a déjà eu lieu : les deux versions tournent. Livrer n'est qu'un coup d'aiguillage, et une bêtise se répare dans l'autre sens, d'une bascule. Trait plein : avant la bascule. Pointillé : après.

Voyons où vous en êtes. *Accelerate* donne quatre chiffres pour se situer objectivement, les **métriques DORA** (DevOps Research and Assessment, l'équipe de recherche derrière le livre) :

- **lead time** : temps entre un commit et sa mise en prod. Se mesure avec deux horodatages que Git et la CI ont déjà. Les meilleurs : moins d'une heure ;
- **fréquence de déploiement** : à quelle cadence on livre. Se compte dans les logs de la CI. Les meilleurs : à la demande, plusieurs fois par jour ;
- **MTTR** (Mean Time To Recovery) : temps pour rétablir le service après une panne, du début de l'incident au retour à la normale. Les meilleurs : moins d'une heure ;
- **taux d'échec des changements** : déploiements suivis d'un rollback (retour à la version d'avant), d'un hotfix (correctif poussé en urgence) ou d'un incident, divisés par le total. Les meilleurs : 0 à 15 %.

Ces quatre chiffres se lisent ensemble, sans formule ni score combiné : les deux premiers mesurent la vitesse, les deux derniers la stabilité. Et quand on trace les quatre chiffres de milliers d'équipes, celles-ci se rangent d'elles-mêmes en trois familles : haute, moyenne, basse performance. Les meilleures gagnent sur les quatre à la fois, là où les plus faibles livrent tous les six mois (enquêtes State of DevOps, 2014-2017). Le gain est aussi humain : la « douleur

de déploiement» mène à l'épuisement des équipes si on la laisse s'installer, et c'est exactement elle que ces pratiques (livrer souvent, petit, réversible) font baisser.

Un dernier maillon conditionne tout le reste : **voir**. Le MTTR suppose qu'on sait qu'il y a une panne. Sans logs (le journal de ce que fait l'application), sans métriques (ses chiffres vitaux : requêtes, erreurs, latence) et sans alerte qui réveille quelqu'un, on l'apprend par un tweet furieux. On ne tient que ce qu'on voit.

Site *Reliability Engineering*, le livre écrit par des ingénieurs de Google, ramène ces chiffres vitaux à quatre : la latence (le temps de réponse), le trafic (combien de requêtes arrivent), les erreurs (combien échouent) et la saturation (à quel point la ressource est pleine). Cette dernière, c'est la file d'attente de la section 6.4, vue depuis la salle des machines. Ces quatre chiffres disent qu'il y a un problème. L'**observabilité** va plus loin : c'est pouvoir dire ce qui se passe dans un système depuis l'extérieur, en lisant ce qu'il écrit sur lui-même, sans l'ouvrir ni le redéployer. Quand on code, ça se décide à chaque ligne de journal : « erreur » ne sert à rien, « paiement 8123 refusé, carte expirée, 412 ms » se relit en pleine nuit.

↳ [Continuous Delivery](#) · [Accelerate](#) · [The Phoenix Project](#)

6.6 La lecture tient en copiant, l'écriture en découpant

Le service tient le déploiement. Reste la charge : un million de lecteurs sur une seule base, et chaque requête fait la queue dans la même file.

Les tables de [4.6](#) se paient à la lecture : une commande ne recopie pas son client, elle garde sa clé, et la jointure qui recolle les deux coûte de plus en plus quand les tables grossissent.



Figure 49 : La commande ne stocke que le numéro du client. La jointure va chercher la ligne correspondante et recolle les deux à la lecture.

La grille n'est pas la seule façon de ranger. Une base documentaire (MongoDB en est l'exemple connu), l'une des familles dites **NoSQL** (« pas seulement SQL »), range chaque commande dans un dossier : un bloc de JSON qui contient tout, le client, les lignes, l'adresse. Lire une commande, c'est ouvrir un seul dossier, sans jointure. En échange, le même client est recopié dans chacune de ses commandes : s'il déménage, il faut corriger partout.

C'est un compromis sans camp, comme ceux de la [section 4.4](#). Le dossier est rapide quand on lit toujours les mêmes choses ensemble. La grille est sûre quand les mêmes données se croisent de mille façons. Reste l'expression « sans schéma » : le schéma existe toujours, la seule question est qui le fait respecter. La grille le vérifie à l'écriture, comme le typage statique de la [section 2.3](#). Le dossier laisse l'application le vérifier à la lecture, comme le typage dynamique.

Côté lecture, on se défend en trois temps. D'abord l'index de [4.6](#), l'annuaire trié qui évite de parcourir la table entière. Puis le **cache**, qui garde les réponses déjà calculées tout près pour les resservir telles quelles. Redis, par exemple, est une mémoire partagée ultra-rapide qu'on place devant la base. Le CDN de [5.5](#) fait pareil pour les fichiers, images et scripts, avec des copies posées près des visi-

teurs. C'est la pyramide du 1.2, à l'échelle d'un datacenter (le bâtiment qui abrite les serveurs). Quand ça ne suffit plus, on **copie** : la **réplication** duplique la base sur plusieurs machines, et chacune sert sa part des lectures.

Le prix de la copie : les répliques ont souvent un temps de retard, car la copie se fait en arrière-plan plutôt que de faire attendre chaque écriture. Vous postez un commentaire (écrit sur la machine principale), vous rechargez la page (lue depuis une copie pas encore à jour) : il a disparu. Le cache paie le même prix : une copie périmée ressort une vieille réponse, et savoir quand la jeter passe pour l'un des deux problèmes les plus durs du métier, avec le nommage (3.1).

Copier n'aide pas l'écriture : chaque copie devrait encaisser chaque écriture. Elle, on la **découpe**. Le **sharding** répartit les données par tranches (clients A-M ici, N-Z là), et chaque machine n'encaisse que sa part.

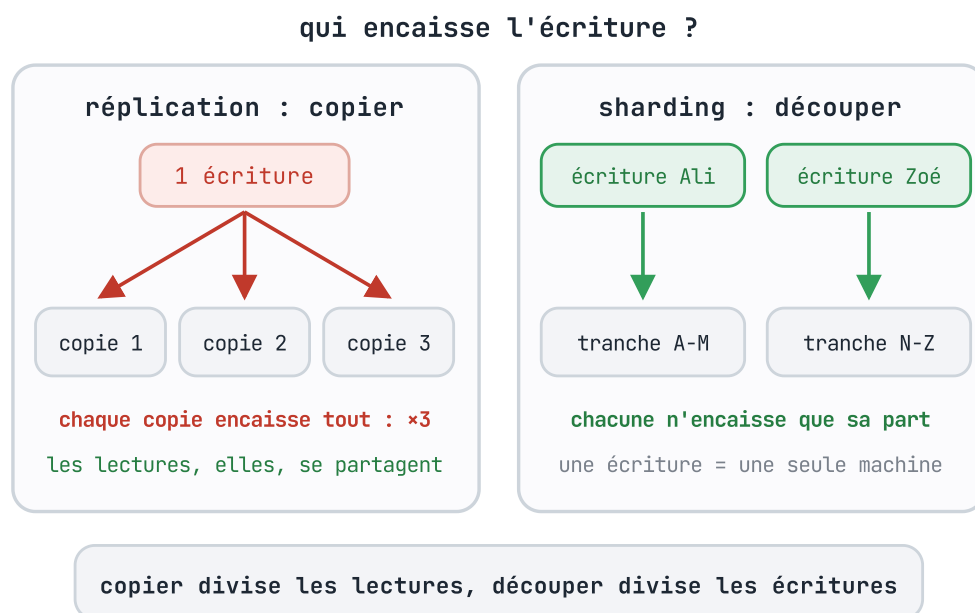


Figure 50 : L'asymétrie en une image : en copiant, chaque écriture doit toucher toutes les copies. En découpant, elle n'atterrit que sur sa tranche.

C'est ici que les ennuis commencent : une donnée éclatée sur plusieurs machines, c'est exactement la situation où les garanties meurent.

↳ Designing Data-Intensive Applications

6.7 À l'échelle, les garanties se paient

Sur une seule base de données, vous vivez protégé sans le savoir. La transaction de [4.6](#) y est tout-ou-rien : le virement débite ET crédite, ou ne fait rien. C'est le contrat **ACID** : atomique (tout ou rien), cohérente (les règles de la base restent vraies), isolée (chaque transaction croit être seule), durable (une fois validée, c'est écrit). Le moteur vous l'offre gratuitement : si l'étape 2 échoue, il annule l'étape 1 tout seul (la base dit *rollback*, un autre retour en arrière que celui de [6.5](#)).

Puis le service grossit, la donnée se répartit sur plusieurs machines, et ce contrat ne vient plus tout seul : annuler « tout » demande une coordination entre machines, lente et fragile, que presque personne ne paie. Chaque machine ne voit que son morceau.

Des anomalies apparaissent, que la plupart des bases laissent passer même sur une seule machine, faute d'isolation totale. La plus vicieuse, le **write skew** : deux transactions, chacune parfaitement valide, qui cassent une règle ensemble. L'exemple du livre : un hôpital exige au moins un médecin de garde. Alice et Bob, les deux derniers de garde, se désinscrivent au même moment. La transaction d'Alice vérifie « Bob est encore là ? oui » et valide. Celle de Bob véri-

fie « Alice est encore là ? oui » et valide. Chacune a vu un monde où la règle tenait. Les deux ensemble laissent zéro médecin. Aucune erreur n'a été levée nulle part.

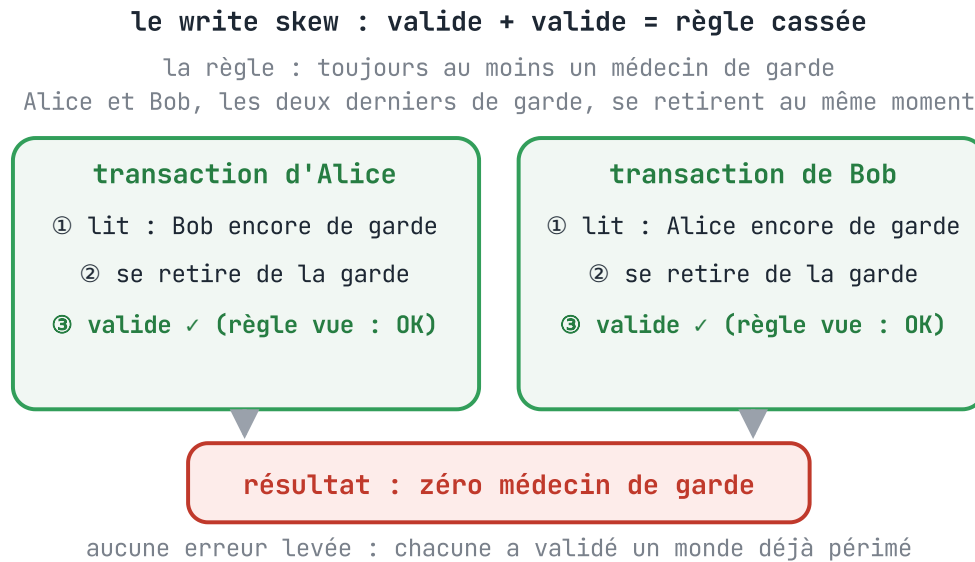


Figure 51 : Vert + vert = rouge : chaque transaction a validé un monde où la règle tenait encore. Ensemble, elles la cassent sans déclencher la moindre erreur.

Faute d'annulation globale, on écrit la marche arrière à la main : c'est la **saga**. Commander = réserver le stock ①, débiter la carte ②, créer la livraison ③. Si ③ échoue, votre code déclenche lui-même le remboursement de ② puis la libération de ①. Ce que le moteur faisait gratuitement devient votre travail, étape par étape.

Ce piège ne vous attend pas qu'à l'échelle des grandes bases réparties. Deux fils d'exécution dans le même programme (deux threads, les cuisiniers de la [section 2.7](#), ou deux goroutines, les fils légers de Go) qui lisent et écrivent la même variable rejouent le write skew en miniature : chacun agit sur un monde déjà périmé, et ça s'appelle une **race condition**. Même parade qu'en base : verrouiller l'accès (un *mutex*, le verrou qui ne laisse passer qu'un fil à la fois), ou mieux, ne pas partager du tout.

La morale du distribué : le réseau ment, l'horloge ment, et « la suspicion et la paranoïa paient ».

↳ [Designing Data-Intensive Applications](#) · [The Hard Parts](#)

↳ CE QUI MÈNE AU CHAPITRE 7

*Un système prouvé, livré et tenu sous la charge est enfin prêt à rencontrer ses vrais usagers. Et c'est précisément là que les certitudes techniques se heurtent au réel : un humain pressé qui ne lit pas, une équipe qui grandit, un attaquant qui sonde. La technique n'était que le moyen. Le **produit pour des humains**, voilà la fin.*

« Le design, ce n'est pas seulement l'apparence. Le design, c'est comment ça fonctionne. »

— Steve Jobs, 2003

7 Le logiciel est pour des humains

produit, équipe, attaquant

Tout ce qui précède sert un but unique : un humain, devant l'écran, qui veut faire quelque chose. Et autour de lui, d'autres humains : l'autre développeur qui appelle votre API, l'équipe qui construit, et l'attaquant qui cherche la faille. Ce chapitre regarde le code par les yeux de ces gens-là. Les exemples viennent du web, là où le plus d'humains rencontrent du logiciel, mais chaque idée vaut aussi pour un écran de caisse, une application de téléphone ou un terminal.



Figure 52 : Le bon logiciel se prend en main sans notice, par tout le monde, y compris ceux qu'on oublie.

7.1 Ne me fais pas réfléchir

Un utilisateur ne lit pas une page, il la scanne, et chaque demi-seconde d'hésitation est une friction qui le fait partir. D'où la loi que l'ergonome Steve Krug a posée en 2000, et qui donne son titre au livre : ne me fais pas réfléchir. Les

conventions battent la créativité (la loupe en haut à droite, le panier juste à côté) parce que l'utilisateur les trouve sans y penser, sur tous les sites qu'il connaît déjà.

Quand on code, ces conventions s'écrivent en HTML. Un champ de type `search` propose une touche « Rechercher » au clavier du téléphone. Un `label` rattache chaque champ à son intitulé et agrandit la zone de clic. Et `autocomplete` évite de retaper une information que le navigateur connaît déjà :

```
<input type="search"                                <!-- clavier adapté sur
mobile -->
<label for="ville">Ville</label>
<input id="ville" autocomplete="address-level2">    <!-- la ville se remplit
toute seule -->
```

La navigation n'est pas une fonctionnalité du site : elle est le site, de la même façon que le bâtiment, les rayons et les caisses ne sont pas des options d'un magasin : ils sont le magasin. Sans eux, il ne reste rien à entrer, rien à trouver, nulle part où payer.

La raison : le web n'a pas de physique. En magasin, vous savez que l'électroménager est « au fond à gauche » parce que vous l'avez marché. En ligne, il n'y a pas de fond, pas de gauche, pas de haut. La navigation remplace tout ça. Un visiteur qui ne sait pas où il est, d'où il vient et comment chercher ne peut rien faire du tout.

Le test du coffre révèle si ça tient. Imaginez qu'on vous bande les yeux, qu'on vous balade, et qu'on vous lâche sur une page au hasard dans un site. C'est exactement ce que vit quelqu'un qui arrive depuis Google sur votre page 7. Plissez les yeux : quel site est-ce ? Quelle page ? Quelles sections existent ? Où suis-je dedans ? Comment chercher ? Si ça ne saute pas aux yeux en vingt secondes, la navigation a échoué. Et pour trouver ces problèmes : trois utilisateurs, un matin par mois, un débrief au déjeuner. Krug observe que les trois premiers utilisateurs rencontrent déjà la plupart des problèmes sérieux. La fréquence bat la cérémonie.

La navigation n'est qu'une partie du tableau. Chaque visiteur arrive avec un **réservoir de bonne volonté** que chaque mauvaise décision vide. Vous avez mis vingt minutes à remplir votre panier : 45 €. Vous cliquez « commander ». Frais de livraison : 12 €. Vous fermez l'onglet. Voilà le réservoir à sec, en un seul chiffre caché trop tard. Le remplir, c'est l'inverse : être transparent, pardonner une faute de format, ne jamais lui barrer la route avec une animation.

↳ [Don't Make Me Think](#)

7.2 Une API se conçoit pour celui qui l'appelle

L'API, le guichet de [4.5](#), est le contrat par lequel un programme en appelle un autre, le plus souvent par-dessus le web. Son vocabulaire universel, ce sont les **verbes HTTP** : `GET` pour lire, `POST` pour créer, `PUT` / `PATCH` pour modifier, `DELETE` pour supprimer. Ils disent l'intention sans qu'on lise la doc : `GET /users/42` se devine seul. Et la règle d'or d'Arnaud Lauret dans *The Design of Web APIs* est le *consumer-first* : on ne part pas de sa base de données, on part de ce que l'appelant veut faire, on dessine la réponse idéale pour lui, puis on construit à rebours, jusqu'à la base de données.

Trois règles complètent le contrat :

- › **Noms prévisibles** : qui a vu une route (une adresse de l'API, comme `/users/42`) devine toutes les autres.
- › **Erreur généreuse** : dire ce qui ne va pas, où, et tout d'un coup, pas un reproche par essai.
- › **Donnée minimale** : la donnée la plus sûre est celle qu'on n'envoie pas : exposez le strict nécessaire.

```
// x deviné par personne  
{ "ACTBLNDFPRTF": true }
```

```
// ✓ deviné par tous, et standard  
{ "overdraftFacility": { "active": true } }
```

↳ [The Design of Web APIs](#)

7.3 Accessible par construction

Une personne aveugle ne voit pas votre bouton. Son lecteur d'écran (le logiciel qui lit la page à voix haute, comme VoiceOver ou NVDA) le lui annonce, en trois informations : un **nom** (le texte lu), un **rôle** (quel genre de chose c'est), un **état** (coché, ouvert, désactivé). Quand un élément HTML natif ne suffit pas à exprimer l'une de ces trois choses, les attributs **ARIA** (Accessible Rich Internet Applications), préfixés `aria-*`, permettent de la renseigner explicitement :

```
<button aria-pressed="true">Favori</button>  
// annoncé : "Favori, bouton, activé"
```

Un `<div>` cliquable n'a ni rôle ni état : pour le lecteur d'écran, il n'existe pas. Première règle d'ARIA, donc : utiliser le bon élément HTML, qui fournit les trois gratuitement. Tout ça est cadré par un référentiel mondial, les **WCAG** (Web Content Accessibility Guidelines) : quatre principes (Perceptible, Utilisable, Compréhensible, Robuste) et trois niveaux d'exigence (A, AA, AAA). Deux gestes concrets couvrent l'essentiel. D'abord un contraste suffisant : le ratio de 4,5:1 sur le texte est une mesure calculée sur la luminance des deux couleurs, de 1:1 pour deux couleurs identiques à 21:1 pour du noir sur blanc. Ensuite la navigation au clavier, qui suppose qu'on voie le **focus**, l'élément qui reçoit les touches, que signale un contour visible.

```
/* x l'erreur la plus répandue : on efface le contour de focus */  
button:focus { outline: none; }  
/* ✓ un focus visible : l'utilisateur au clavier voit où il est */  
button:focus { outline: 2px solid #005fcc; }
```

Le test ultime, gratuit : posez la souris, parcourez votre page à la touche Tab. Si vous perdez le fil, un utilisateur au clavier aussi.

↳ [Web Accessibility Cookbook](#)

7.4 L'organisation se reflète dans le code

Jusqu'ici, on a regardé le logiciel par le dehors : l'utilisateur, même celui qu'on oublie, puis le développeur qui appelle l'API. Voyons maintenant qui le construit. L'organisation elle-même est une interface. **Loi de Conway** : un système copie la structure de communication de l'organisation qui le construit. Quatre équipes qui se parlent mal produiront quatre modules qui s'emboîtent mal, qu'on le veuille ou non.

la loi de Conway : l'organisation se copie dans le code

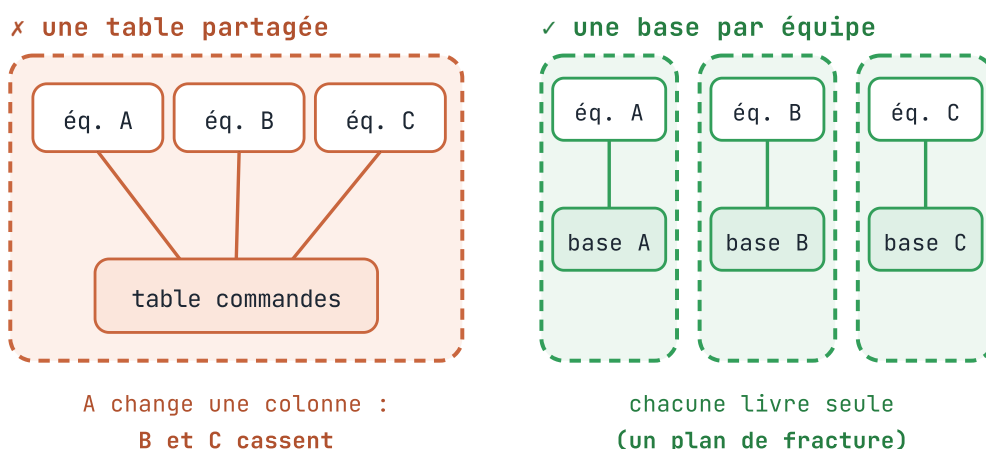


Figure 53 : Le même dessin que le quantum (ch. 4), côté humains : une table partagée couple les trois équipes de force. Trois bases, et chacune livre seule.

Team Topologies retourne la loi en levier : si vous voulez une certaine architecture, organisez d'abord les équipes pour l'obtenir. La contrainte cachée est la **charge cognitive** : une équipe ne peut tenir qu'une quantité limitée de domaine. On découpe alors le système le long de ses **plans de fracture**, ses coupures naturelles, le plus souvent le domaine métier. Et le livre donne le mode d'emploi, quatre types d'équipes :

- **alignée sur un flux** : une équipe = un produit, livré seule de bout en bout (le cas par défaut) ;
- **plateforme** : fournit des outils internes pour que les autres livrent sans attendre ;
- **habilitante** : aide une équipe à monter en compétence, puis se retire ;

- › **sous-système compliqué**: maintient un morceau trop pointu pour être partagé (un moteur de calcul, par exemple).

Ce plan de fracture le long du métier a un jumeau côté modèle: le **Bounded Context** d'Eric Evans, une frontière à l'intérieur de laquelle chaque mot du domaine garde un seul sens. Deux équipes qui se partagent un objet « Commande » sans cette frontière finissent avec deux définitions incompatibles dans la même table (l'équipe achats pense commande fournisseur, la cuisine pense commande client) et le rapport mensuel qui plante. Tracer la frontière, une équipe, un modèle, une base à elle, c'est se donner le droit que « Commande » signifie autre chose ailleurs, sans rien casser.

Dans les deux sens, la leçon est la même: dessiner les équipes, c'est déjà dessiner le système.

↳ [Team Topologies](#) · [Domain-Driven Design](#) · [Learning DDD](#)

7.5 Ajouter des gens à un projet en retard le retarde

L'intuition dit: projet en retard, on ajoute des développeurs. C'est faux, et ça porte un nom, la **loi de Brooks**. Le travail se divise mal, chaque nouvelle personne doit être formée (par les anciens, qu'on ralentit donc), et surtout elle multiplie les canaux de communication.

```
canaux de communication = n × (n - 1) ÷ 2
5 personnes → 10 canaux           // gérable
15 personnes → 105 canaux          // 105 canaux à entretenir
```

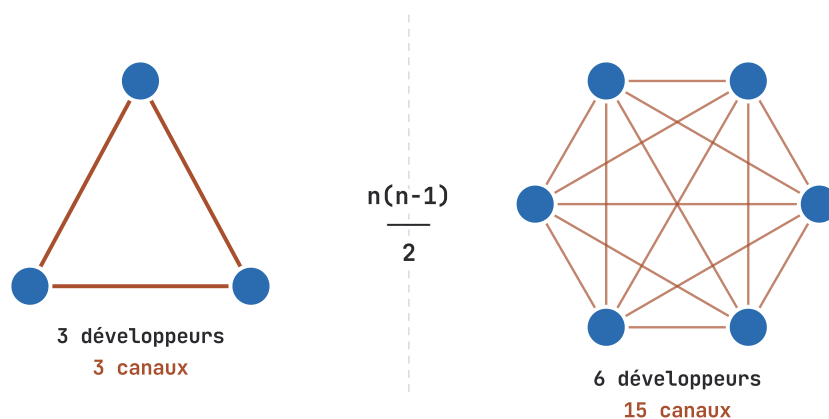


Figure 54 : De 3 à 6 personnes, les canaux passent de 3 à 15 : chaque arrivée multiplie les traits, pas seulement les bras. À 15 personnes : 105 canaux.

«Le mois-homme comme unité de mesure d'un travail est un mythe dangereux et trompeur: il laisse croire que les hommes et les mois sont interchangeables.» Ils ne le sont pas: neuf femmes ne font pas un bébé en un mois.

Si le nombre ne fait pas la performance, qu'est-ce qui la fait ? La cohésion. De-Marco et Lister appellent **équipe soudée** (*jelled team*) un groupe si lié que le tout dépasse la somme des parties : peu de départs, une identité commune, une fierté partagée. On ne la fabrique pas sur commande, on crée seulement ses conditions. En revanche, on la tue vite, et ils ont un mot pour ça : le **teamicide**, tout ce qui défait une équipe ou l'empêche de se souder. Surveiller les gens plutôt que leur faire confiance, les disperser dans des bureaux séparés, imposer des deadlines bidon, casser une équipe qui marche sitôt le projet fini : chacun de ces réflexes de management suffit à la fissurer.

↳ [Le Mythe du mois-homme](#) · [Peopleware](#)

7.6 Penser comme l'attaquant pour défendre

Dernière figure du chapitre, la moins aimable : l'attaquant. Première surprise : il ne vous a pas choisi. L'attaque est industrielle : des robots scannent le web en continu, sans cible particulière, et on n'est pas attaqué parce qu'on est intéressant, mais parce qu'on est accessible. C'est la thèse de *Web Application Security*, le livre d'Andrew Hoffman : on ne défend bien que ce qu'on sait attaquer. Le

livre s'organise d'ailleurs comme une attaque réelle : la **reconnaissance** d'abord (cartographier l'application, chercher l'entrée de service plutôt que la porte d'entrée), l'offense ensuite, la défense en dernier.

Que trouve l'attaquant ? Presque toujours le même péché : une donnée venue de l'utilisateur que le code traite comme du code, ou croit sur parole. L'**injection SQL** montre le mécanisme en entier. Votre moteur de recherche colle ce que tape le visiteur dans une requête. L'attaquant ne tape pas un nom : il tape une apostrophe qui referme votre texte, puis sa propre commande :

```
recherche tapée : ';' DROP TABLE users --
requête exécutée : SELECT * FROM produits WHERE nom = '; DROP TABLE users
--'

// x l'apostrophe referme le texte : la suite devient
// un ORDRE
```

La table des utilisateurs vient d'être supprimée par un formulaire de recherche. (Beaucoup de pilotes, ces composants qui portent vos requêtes jusqu'à la base, refusent d'enchaîner deux ordres. L'attaquant lit alors toute la table au lieu de la détruire. Même faille.) Les deux cousines jouent la même note. Le **XSS** (Cross-Site Scripting) injecte non pas du SQL mais un bout de script, glissé dans la page par une balise HTML, qui s'exécutera dans le navigateur des autres visiteurs. Et le **mass assignment** glisse un champ en trop dans la requête, que le serveur enregistre sans se méfier :

```
POST /api/profil { "nom": "Alice", "isMember": true }
user.update(req.body) // x isMember passe → Alice s'auto-promeut
```

La parade se lit en miroir : ne jamais faire confiance à l'entrée. Contre l'injection, les **requêtes préparées** : l'entrée voyage à part de la requête et ne peut jamais redevenir du code. Contre le XSS, échapper tout ce qu'on affiche : transformer les caractères spéciaux en texte inoffensif. Le `<` d'une balise devient `<`, que le navigateur affiche au lieu de l'exécuter. Contre le mass assign-

ment, une liste blanche des champs acceptés. Et par-dessus, la **défense en profondeur** : chaque couche se protège elle-même, du navigateur à la base, pour que si l'une cède, les autres tiennent.

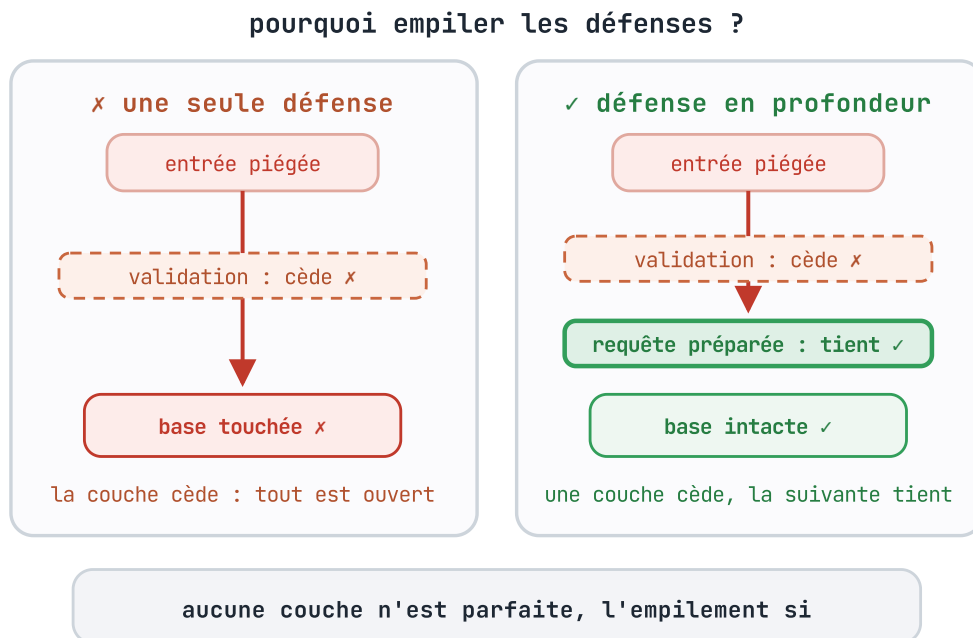


Figure 55 : La même entrée piégée, la même couche qui cède : seul l'empilement change l'issue. C'est tout le pari de la défense en profondeur.

L'injection n'est que la première porte. La seconde est plus bête encore, et c'est la faille la plus répandue du web dans le classement OWASP: `GET /users/42` renvoie votre fiche, alors l'attaquant tape `/users/43` et lit celle du voisin. L'**authentification** vérifie qui parle. L'**autorisation** vérifie s'il en a le droit. Ici, le serveur a vérifié la première et oublié la seconde : chaque demande doit contrôler que la fiche demandée appartient à celui qui la demande.

Troisième porte, les mots de passe. Ils ne se stockent jamais en clair, ni même chiffrés, car une clé se vole : on n'en garde que le hachage, le résultat d'une fonction de hachage (1.3), un résumé impossible à inverser, calculé par une fonction faite pour être lente (bcrypt, argon2), afin qu'une base volée ne livre aucun mot de passe. Enfin le **moindre privilège** : chaque programme et chaque compte n'a que les droits dont il a besoin. Le compte avec lequel la bou-

tique parle à sa base n'a aucune raison de pouvoir supprimer une table, et l'injection de tout à l'heure aurait échoué pour cette seule raison. C'est le moindre pouvoir de 2.8, appliqué aux droits.

Un dernier réflexe à copier sur les gens du métier : ils disent « mitigations », jamais « fixes ». On réduit le risque, on ne l'efface pas. Aucune défense n'est définitive.

↳ Web Application Security

↳ CE QUI MÈNE AU CHAPITRE 8

*Tout ce métier, des bits jusqu'à l'équipe, vient d'être bousculé par un acteur entièrement neuf, capable d'écrire du code à la demande : l'IA. Elle ne remplace pas le savoir d'avant, elle le rend **plus nécessaire** que jamais : quelqu'un doit juger ce qu'elle produit, et bien juger exige précisément tout ce qu'on vient de graver.*

*« La perfection est atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retirer. » — Saint-Exupéry, *Terre des hommes*, 1939*

8 Coder à l'ère de l'IA

le dernier maillon

Un nouvel étage s'est posé sur les sept précédents : une machine qui écrit du code à la demande. La question n'est plus « est-ce que l'IA sait coder ? » (oui, souvent), mais « que reste-t-il à l'humain ? ». On guette une réponse technique, un recoin que la machine ne saurait pas encore coder. La réponse est ailleurs : ce qui reste à l'humain, c'est décider si ce que la machine produit est correct, sûr et à sa place. Et cette capacité à décider, les sept étages précédents viennent justement de la construire : vous savez reconnaître un nom honnête, une architecture qui tient et un test qui prouve. Vous n'avez pas fait cette ascension pour rien : c'est ce bagage que l'arrivée de l'IA rend indispensable.

Avant de monter à ce dernier étage, une confusion à lever. « Faire de l'IA », aujourd'hui, ce n'est presque jamais entraîner un modèle : c'est en appeler un déjà entraîné et bâtir autour. Personne ne redessine la carte du monde pour afficher un plan d'accès sur son site : on intègre Google Maps. L'ingénieur IA fait pareil : le modèle, c'est sa carte du monde, dessinée par d'autres. Son travail commence après : ce qu'on envoie au modèle, ce qu'on fait de sa réponse,

et la manière de vérifier le tout. C'est le sens du mot *engineering* dans *AI Engineering*, le livre de Chip Huyen qui nourrit ce chapitre : tout se joue autour du modèle, pas dedans.



Figure 56 : L'IA tend les instruments à toute vitesse. L'humain tient le scalpel et décide du geste.

8.1 L'IA génère du probable, pas du vrai

Un **modèle de langage** est le programme derrière ChatGPT ou Claude : il a été entraîné sur d'énormes masses de texte pour prédire le mot suivant. Il ne sait pas ce qui est vrai : il produit, mot après mot, la suite la plus probable de ce qui précède. L'**hallucination** (une affirmation fausse, livrée avec l'assurance d'une vraie) n'est donc pas un bug à corriger un jour, c'est le mécanisme même qui le

fait fonctionner: « tout ce qui a une probabilité non nulle, aussi tiré par les cheveux ou faux soit-il, peut être généré par l'IA ». Demandez-lui l'auteur d'un livre obscur: il vous donnera, avec le même aplomb, un nom plausible et faux.

ce que fait le modèle : classer le plausible

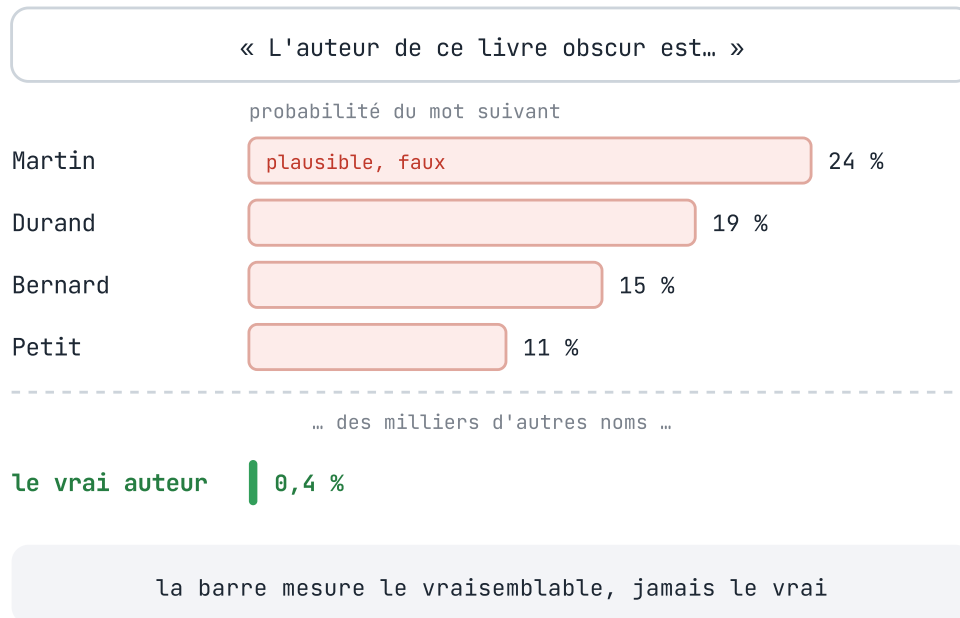


Figure 57 : Les pourcentages sont illustratifs, l'échelle est fidèle : le modèle n'a aucun moyen de voir que la plus petite barre est la bonne.

La confiance se calibre donc selon la nature de ce qu'on lit. La logique d'un raisonnement se juge sur pièce: tout est devant vous. Une référence (nom de fonction, option, API), elle, se vérifie toujours: un nom « qui sonne juste » est exactement ce qu'un générateur de probable sait produire. Bonne nouvelle, l'exécution ne ment pas: appelez une fonction qui n'existe pas et ça plante, langage compilé ou non. Encore faut-il que la ligne s'exécute: une erreur ne se déclare que sur le chemin emprunté, et c'est exactement ce qu'un test garantit.

Les options de configuration, elles, font exception: une option inventée est souvent ignorée en silence, car beaucoup de bibliothèques passent les clés inconnues sans broncher. L'erreur d'exécution reste votre meilleure alliée contre l'aplomb du modèle. Le silence, lui, ne prouve rien.

La conséquence gouverne tout le chapitre: on ne fait jamais confiance les yeux fermés, on vérifie, et on conçoit le système autour de cette incertitude plutôt que contre elle.

↳ [AI Engineering](#) · [Programming Pearls](#)

8.2 Tout se joue dans le contexte

Un modèle se trompe d'abord quand il lui manque l'information. Lui donner le bon contexte au bon moment est devenu la compétence centrale du métier. Le pattern-roi s'appelle **RAG** (*Retrieval-Augmented Generation*: générer en s'appuyant sur des documents récupérés). Le principe: chercher les documents pertinents dans une base externe et les coller dans le **prompt** (le message qu'on envoie au modèle), pour que celui-ci s'appuie sur des faits fournis, pas sur sa mémoire floue.

Pourquoi chercher des morceaux plutôt que tout donner? Un modèle ne lit qu'une quantité limitée de texte à la fois: sa **fenêtre de contexte**. On la mesure en **tokens**, des morceaux de mots (un mot courant en fait un ou deux). Au-delà, le reste n'entre tout simplement pas.

Concrètement, c'est ce que fait votre assistant de code quand il répond correctement sur un projet entier: le projet ne tient pas dans la fenêtre de contexte, alors l'assistant va chercher les bons morceaux. Chaque outil a sa méthode: Claude Code (l'assistant de code IA d'Anthropic) lance des `grep` (la recherche du mot exact dans tous les fichiers), Cursor (un éditeur de code dopé à l'IA) indexe le projet en vecteurs pour chercher par le sens. La recherche par le sens

repose sur la fonction `embed` : elle transforme un texte en vecteur, une liste de nombres qui code son sens, et deux textes qui parlent de la même chose ont des vecteurs voisins :

```
question = "Où est calculée la TVA dans ce projet ?"

extraits = projet.chercher(embed(question), top_k=3)
# 1. top_k=3 : garde les 3 passages de VOTRE code
#    dont le sens est le plus proche de la question
# → ["function prixTTC(ht) { return ht * 1.2; }", ...]
#    trouvé sans le mot « TVA » : la recherche par le sens

prompt = f"Réponds d'après ces extraits : {extraits}\n\n{question}"
# 2. on colle les extraits dans le prompt, avec la question

reponse = modele(prompt)
# 3. le modèle répond en lisant votre code, pas en devinant
```

Le RAG : répondre en lisant, pas en devinant

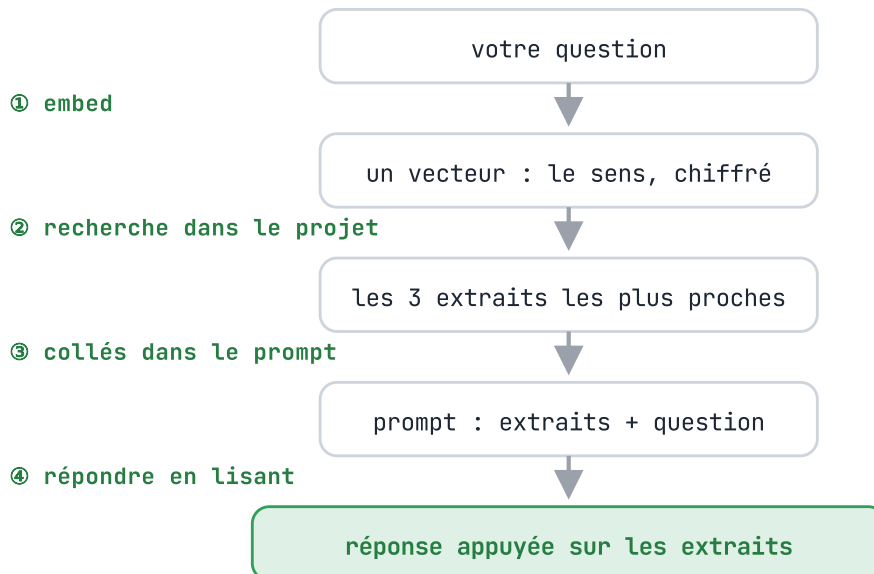


Figure 58 : Le modèle ne pioche plus dans sa mémoire floue : il lit les extraits qu'on vient de lui tendre, puis répond. D'un outil à l'autre, seule l'étape ① change : grep cherche le mot exact, embed cherche le sens.

Le contexte se soigne aussi dans le *prompt* lui-même, et la différence est brutale :

```
# x vague → le modèle devine, réponse imprévisible
"ça marche pas, corrige"

# ✓ rôle + contexte + format imposé → réponse exploitable
"Tu es dev PHP senior. Ce test échoue (message ci-dessous).
Propose le correctif minimal, en diff, sans réécrire le reste."
```

Une session avec un assistant recommence de zéro : à chaque fois, il faudrait répéter les conventions du projet, les commandes à lancer, les pièges connus. La parade : écrire ces consignes une fois pour toutes dans un fichier à la racine du projet, le `CLAUDE.md` de Claude Code, que l'assistant relit au début de chaque session. C'est la différence entre réexpliquer sa mission à un stagiaire chaque matin et lui préparer un bureau où tout est déjà en place. Le métier a baptisé ce travail *context engineering* : préparer l'environnement de travail de l'assistant, pour que même une question médiocre produise un bon résultat.

Un dernier mot, que vous croiserez dans le métier : le **fine-tuning**, ré-entraîner un modèle sur ses propres données. Le critère de Chip Huyen : « le fine-tuning corrige la forme, le RAG fournit les faits ». Corriger la forme est un chantier de constructeur de produit IA, coûteux en données, en calcul et en maintenance : celui qui code avec le modèle n'y touchera probablement jamais. Votre remède, à vous, s'appelle toujours contexte : les bons extraits, les bonnes consignes, au bon moment.

↳ [AI Engineering](#) · [Programming Pearls](#)

8.3 L'IA se pilote en boucle courte

Confiez une fonctionnalité entière à l'IA et vous recevez deux mille lignes plausibles. Si elles ne marchent pas, personne ne sait lequel des dizaines de choix empilés est fautif : ni vous, ni l'IA. Coder avec l'IA n'est pas un sprint, c'est une boucle courte : une tâche atomique (une seule petite chose à la fois), les tests au vert, le diff lu (la liste exacte des lignes que l'IA a ajoutées ou retirées), un commit, puis la suivante.

Chaque maillon de cette boucle descend des étages précédents. Le test d'abord devient un contrat exécutable (6.1) : il transforme « fais un truc qui marche » en « fais passer ces assertions » (les vérifications écrites dans le test), une cible que la machine peut viser et relancer seule. Git (6.3), lui, reprend son rôle de filet : on committe un état propre avant de lâcher l'IA, et le diff dit ce que l'IA a vraiment fait, pas ce qu'elle prétend avoir fait.

L'outil le plus avancé, l'**agent autonome** (Claude Code en est un), n'est que cette boucle automatisée : récupérer le contexte (le ticket, les fichiers concernés, la dernière erreur), agir, vérifier, recommencer. Tout se joue au troisième temps : la vérification. Elle doit être un signal extérieur et objectif : les tests, le build (la fabrication automatique de l'application), le linter de 3.1. Jamais l'avis de l'agent sur lui-même : il vous dira toujours qu'il a réussi, avec le même aplomb qu'il met à inventer une fonction inexistante. Faire relire le travail par un second agent aux yeux neufs (la revue de code de la section 6.3, confiée à une machine) aide, parce qu'il n'a rien à défendre. Mais son verdict reste un avis. Les tests, eux, n'ont pas d'avis.

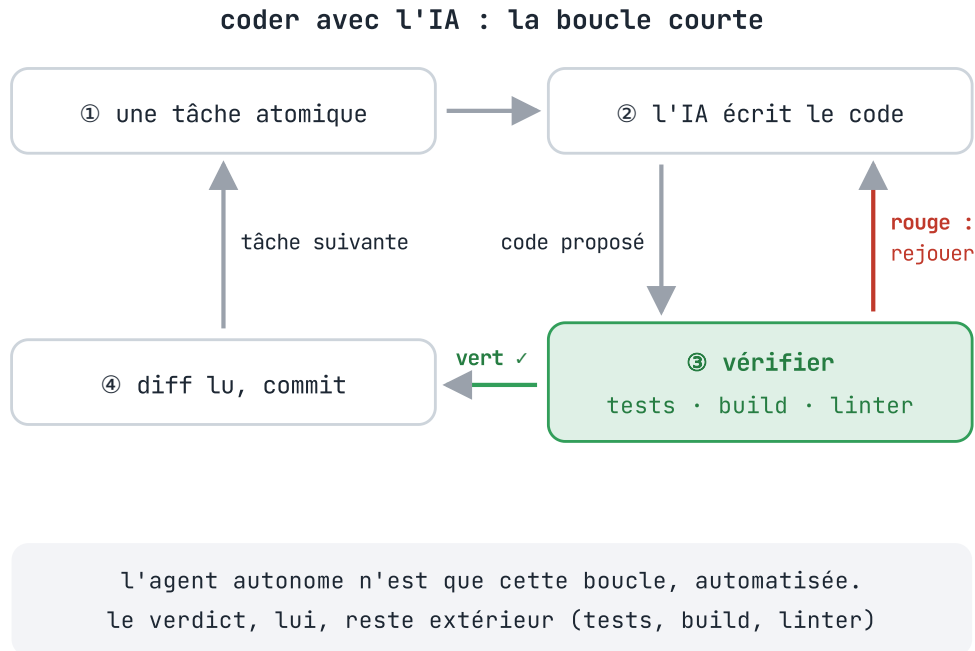


Figure 59 : Tant que c'est rouge, la boucle rejoue. Au vert seulement, on lit le diff, on committe, et on prend la tâche suivante.

Le métier appelle **évaluation** le verdict, venu de l'extérieur du modèle, sur ce qu'il produit. C'est elle, et non le modèle, le vrai goulot. Pour du code, une partie du verdict est automatique : ça compile, les tests passent. Mais ce verdict n'est qu'un plancher, pas une note. Des tests verts ne disent rien de la lisibilité, de la place dans l'architecture, de la faille qui dort. Le vrai jugement du code est multicritère, et c'est l'humain qui le porte.

Pour une réponse en texte, il n'y a même pas de plancher : rien ne plante jamais. La grille est à construire soi-même, question par question :

- › rien d'**inventé** ?
- › les faits cités sortent-ils des **extraits fournis** ?
- › le **format** demandé est-il respecté ?

Sans grille, on itère à l'aveugle, exactement comme du code sans tests.

↳ [AI Engineering](#) · [Cours](#) · [Coder avec l'IA](#)

8.4 L'équipe chirurgicale, enfin réalisable

L'IA ne change pas que la façon d'écrire : elle redessine l'équipe. Brooks rêvait en 1975 d'une **équipe chirurgicale** : un cerveau qui tient le scalpel (conçoit, décide), entouré d'assistants spécialisés. L'un connaît tout le code, un autre retient chaque recoin du langage, un troisième forge les scripts et les outils.

Le rêve butait sur un dilemme que Brooks jugeait cruel : une poignée de bons esprits garde la cohérence du système, mais avance trop lentement pour les grands projets. L'IA dissout le dilemme : vous restez le seul cerveau, elle fournit les bras. L'assistant connaît tout le code, retient chaque recoin du langage, forge les scripts, et il joue ces trois rôles à la fois. Le modèle imaginé il y a cinquante ans devient praticable le jour où les bras sont une machine. La loi de Brooks n'y perd rien (7.5) : ces bras-là coûtent peu en formation, elle se réduit au contexte que vous leur donnez (section 8.2), et ils n'ajoutent personne en réunion. Reste un coût qu'aucun contexte ne supprime : relire ce qu'ils produisent.

Souvenez-vous de Conway (7.4) : la forme de l'équipe se copie dans le système. Une équipe redessinée en un cerveau et des bras-machine produira un autre logiciel, et c'est vous qui décidez lequel.

Vous êtes le chirurgien en chef, et la métaphore se traduit geste par geste. Les instruments qu'on vous tend, ce sont les diffs générés : vous en acceptez un, vous en refusez deux, vous renvoyez le troisième en précisant la demande. Les gestes que vous décidez, ce sont l'architecture, le découpage, le nom des choses : l'assistant propose, il ne tranche jamais. Et le compte-rendu que vous signez, c'est le commit : votre nom dessus, pas celui de la machine. On ne signe jamais un code qu'on n'a pas relu.

↳ [Le Mythe du mois-homme](#) · [AI Engineering](#)

8.5 Le jugement humain est le dernier maillon

Si l'IA écrit le code, qu'est-ce qui reste à l'humain ? **Tout ce qui précède.** Pour juger si le code généré est juste, il faut :

- › comprendre ce qu'il **coûte** ([chapitre 1](#)) ;
- › savoir si le langage l'**exprime** bien ([chapitre 2](#)) ;
- › juger s'il est **lisible** par le prochain humain ([chapitre 3](#)) ;
- › voir s'il tient dans l'**architecture** ([chapitre 4](#)) ;
- › savoir ce qu'il paiera quand la donnée **voyage** ([chapitre 5](#)) ;
- › pouvoir le **prouver** et le livrer ([chapitre 6](#)) ;
- › vérifier qu'il **sert** l'humain, et résiste à l'attaquant ([chapitre 7](#)).

L'IA produit le probable. **Vous décidez du juste.** Plus elle écrit vite, plus votre jugement devient le maillon rare. La frontière du métier passe désormais là : d'un côté, celui qui comprend, teste et sait expliquer chaque ligne qu'il livre. De l'autre, celui qui accepte sans lire et croise les doigts : le métier a déjà un nom pour ça, le *vibe coding* (coder au feeling). Une seule règle pour rester du bon côté : ne livrez jamais une ligne que vous ne sauriez pas expliquer.

Ce jugement commence même avant la première ligne, dans la façon de poser le problème. Quand un programmeur demanda un jour à Jon Bentley comment trier un fichier sur disque, quinze minutes de questions ont remplacé une semaine de code : le vrai besoin tenait en dix millions de petits entiers distincts dans environ un mégaoctet de mémoire. La bonne réponse ne triait rien : une case d'un bit par numéro possible, cochée si le numéro figure dans le fichier, puis on relit les cases dans l'ordre et la liste sort triée d'elle-même. Définir le vrai problème, c'était quatre-vingt-dix pour cent de la bataille. Une IA ne pose pas ces questions à votre place : lâchez-lui « trie ce fichier » et elle livrera consciencieusement le tri d'une semaine, jamais le raccourci de dix secondes.

Ce livre entier n'apprend pas à coder à la place de l'IA : il apprend à savoir quand elle a tort. Rien ne garantit qu'elle pensera, seule, à stocker un montant en centimes (des entiers exacts) plutôt qu'en flottant (des approximations) (1.1) : il faut que vous, vous le sachiez.

↳ [AI Engineering](#) · [Programming Pearls](#)

↳ LA BOUCLE SE REFERME

Voici le vertige : ce jugement-là, l'IA ne peut pas vous le transmettre. Elle imite les réponses de ceux qui savent. Elle ne sait pas quand une réponse est juste. Ce discernement se gagne en faisant ses propres erreurs et en les corrigeant : c'est le seul apprentissage du métier qu'aucun assistant ne raccourcira pour vous. Il faut gravir les huit étages soi-même.

« La question de savoir si les machines peuvent penser est à peu près aussi pertinente que celle de savoir si les sous-marins peuvent nager. » — Edsger Dijkstra



Huit niveaux, un seul fil : des nombres dans le silicium jusqu’au jugement qu’aucun modèle ne remplace. À l’arrivée, on ne décide plus à l’aveugle : on sait quoi faire, pourquoi, où, quand et comment. C’est tout le métier, et les fiches ci-dessous portent le détail de chaque marche.

ET MAINTENANT ?

Ne relisez pas tout. Si vous débutez, retournez à l'étage où le sol s'est dérobé : chaque chapitre s'adosse à une poignée de fiches (la liste est juste en dessous), et c'est là que vit le détail. Si vous codez depuis des années, deux chapitres changeront vos prochaines semaines plus que les autres : l'architecture, parce qu'elle se décide tôt et se paie longtemps, et l'IA, parce qu'elle se pilote.

Lire ne suffit pas : ces idées s'apprennent dans les doigts, et chaque étage a son terrain d'entraînement dans les cours interactifs du site, gratuits.

Le langage ([chapitre 2](#)) : les cours [JavaScript](#) (www.web-developpeur.com/apprendre/javascript/), [PHP](#) (www.web-developpeur.com/apprendre/php/), [Python](#) (www.web-developpeur.com/apprendre/python/) ou [Go](#) (www.web-developpeur.com/apprendre/golang/).

Prouver et livrer ([chapitre 6](#)) : les cours [Tests](#) (www.web-developpeur.com/apprendre/tests/) et [Déploiement](#) (www.web-developpeur.com/apprendre/deploiement/).

Le logiciel pour les humains ([chapitre 7](#)) : les cours [Accessibilité](#) (www.web-developpeur.com/apprendre/accessibilite/) et [Sécurité web](#) (www.web-developpeur.com/apprendre/securite/).

Coder avec l'IA ([chapitre 8](#)) : [le cours du même nom](#) (www.web-developpeur.com/apprendre/coder-avec-ia/), dix-neuf leçons de méthode.

Gardez cette page sous la main : elle ne se lit pas deux fois pareil. Le chapitre sur l'architecture ne dira pas la même chose avant et après votre première vraie refonte. Le jour où une section vous semble évidente, c'est que vous avez monté d'un étage.

Fiches de lecture complètes : web-developpeur.com/bibliotheque/

LES LIVRES FONDUS DANS CE FIL

- 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 [Coder proprement](#) · [A Philosophy of Software Design](#) · [The Pragmatic Programmer](#) · [Refactoring](#)
- 4 [Design Patterns: Tête la première](#) · [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](#) · [Le Mythe du mois-homme](#) · [Web Application Security](#)
- 8 [AI Engineering](#) · [Programming Pearls](#)

Lexique

Chaque définition est la phrase du livre qui introduit le terme. Le numéro renvoie à la section.

accusé de réception

Le destinataire répond par un accusé de réception pour dire ce qui lui est bien parvenu. [5.3 p. 108](#)

ACID

La transaction de 4.6 y est tout-ou-rien : le virement débite ET crédite, ou ne fait rien. C'est le contrat ACID : atomique (tout ou rien), cohérente (les règles de la base restent vraies), isolée (chaque transaction croit être seule), durable (une fois validée, c'est écrit). [6.7 p. 131](#)

adresse

Mis bout à bout, ces octets forment la mémoire : une immense rangée de cases, chacune repérée par un numéro, son adresse. [1.1 p. 5](#)

agent autonome

L'outil le plus avancé, l'agent autonome (Claude Code en est un), n'est que cette boucle automatisée : récupérer le contexte (le ticket, les fichiers concernés, la dernière erreur), agir, vérifier, recommencer. [8.3 p. 149](#)

AIMD

Cette politesse codée s'appelle AIMD (Additive Increase, Multiplicative Decrease : hausse par addition, baisse par division). [5.4 p. 110](#)

API

Une API (interface de programmation) est le guichet d'une application : les autres programmes y déposent leurs demandes, avec un formulaire à remplir et un format de réponse imposé (la section 7.2 y revient). [4.5 p. 96](#)

appelant

La vraie question n'est pas « combien de lignes ? », mais « est-ce plus simple pour l'appelant ? », c'est-à-dire pour le code qui se sert de votre fonction.

[3.2 p. 65](#)

argument

Une fonction est une recette écrite une fois pour toutes : on lui donne des ingrédients, ses arguments, elle rend un plat, son résultat.

[ch. 2 p. 30](#)

ARIA

Quand un élément HTML natif ne suffit pas à exprimer l'une de ces trois choses, les attributs ARIA (Accessible Rich Internet Applications), préfixés aria-*, permettent de la renseigner explicitement. [7.3 p. 137](#)

assertion

Échouer tôt a un outil, l'assertion : une ligne qui vérifie ce qui doit toujours être vrai, et arrête tout sinon, comme assert total >= 0. [3.4 p. 69](#)

asynchrone

Un code qui continue sans attendre la réponse est dit asynchrone. [2.7 p. 55](#)

authentification

L'authentification vérifie qui parle. [7.6 p. 140](#)

autorisation

L'autorisation vérifie s'il en a le droit. [7.6 p. 140](#)

bande passante

La bande passante, c'est le nombre de voies : combien de données passent de front chaque seconde. [5.1 p. 105](#)

base de données

Une base de données est un programme qui garde des données sur le disque et qui, contrairement à un fichier texte, tient trois promesses. [4.6 p. 99](#)

bibliothèque

Une bibliothèque est un lot de fonctions écrites par d'autres, qu'un programme ajoute pour s'en servir sans les réécrire. [2.4 p. 44](#)

bibliothèque standard

Une bibliothèque standard est le lot de fonctions toutes prêtes fourni avec un langage : random fait partie de la bibliothèque standard de Python, et sum marche même sans import. [2.4 p. 44](#)

Big O

Le Big O est l'unité qui dit comment le nombre d'étapes grandit avec la taille de l'entrée, notée n (ici, le nombre de noms). [1.3 p. 21](#)

bit

Un bit, c'est un interrupteur : éteint, il vaut 0, allumé, il vaut 1. [1.1 p. 5](#)

blob

Un blob est le contenu d'un fichier. [6.3 p. 122](#)

booléen

Un booléen, une valeur qui ne peut être que vrai ou faux, tient lui aussi dans un seul casier. [2.2 p. 35](#)

boucle

On répète grâce à une boucle: un morceau de programme que la puce refait une fois par élément à traiter. [2.1 p. 31](#)

boucle d'événements

La boucle d'événements est le mécanisme qui prend sans cesse la tâche suivante dans la file d'attente des tâches (un clic, la réponse d'un serveur, un minuteur: chacun y dépose une tâche). [2.7 p. 55](#)

Bounded Context

Ce plan de fracture le long du métier a un jumeau côté modèle: le Bounded Context d'Eric Evans, une frontière à l'intérieur de laquelle chaque mot du domaine garde un seul sens. [7.4 p. 138](#)

branche

Une branche, ici main, n'est qu'un pointeur vers un commit, qu'on avance à chaque nouveau commit. [6.3 p. 122](#)

cache

Tout le reste vit plus loin, par étages: le cache (une réserve rapide collée à la puce, qui garde une copie des données récentes), puis la RAM (la mémoire vive, les barrettes de votre machine), puis le disque. [1.2 p. 16](#)

cas d'arrêt

Le cas d'arrêt: le carton qui ne contient plus de carton. [2.5 p. 49](#)

cas général

Le cas général: le carton qui en contient un autre. [2.5 p. 49](#)

CDN

Rapprocher le contenu du visiteur, enfin, avec un CDN (Content Delivery Network), c'est-à-dire des serveurs relais répartis dans le monde, puisque aucun abonnement ne raccourcit l'Atlantique. [5.5 p. 112](#)

chaîne de caractères

La chaîne de caractères (string): du texte, comme "bonjour", une suite de lettres qui sont chacune un nombre (l'ASCII et l'UTF-8 de 1.1). [2.3 p. 39](#)

champ

Les données qu'un objet porte sont ses champs (nom, prix, quantité), les actions qu'il sait faire sont ses méthodes, des fonctions rangées dans l'objet (appliquerRemise()). [2.2 p. 35](#)

charge cognitive

La contrainte cachée est la charge cognitive: une équipe ne peut tenir qu'une quantité limitée de domaine. [7.4 p. 138](#)

CI/CD

La CI/CD est la chaîne automatique qui rend la rafale possible. [6.5 p. 126](#)

classe

Une classe est un moule: la classe Utilisateur fixe qu'un utilisateur a un nom, un e-mail, et qu'il sait se connecter. [4.1 p. 80](#)

clé

(Rien à voir avec la table de hachage de 1.3: même mot, autre objet.) Chaque ligne porte un numéro à elle, sa clé: ce qu'on donne pour obtenir la ligne. [4.6 p. 99](#)

closure

La closure: une fonction qui repart avec son propre casier. [2.4 p. 44](#)

code source

Le code source est le texte d'un programme tel qu'un humain l'a écrit, avant toute traduction. [1.4 p. 26](#)

coercition

JavaScript, l'exemple le plus connu, convertit tout seul une valeur d'un type à l'autre (texte, nombre, booléen): c'est la coercition. [2.6 p. 53](#)

cœur

Votre ordinateur garde des dizaines de programmes ouverts en même temps, mais sa puce n'a que quelques cœurs, des unités de calcul dont chacune ne suit qu'un thread à la fois. [2.7 p. 55](#)

collision

Deux clés peuvent donner le même numéro: c'est une collision. [1.3 p. 21](#)

commentaire

Un commentaire est une note écrite pour les humains au milieu du code, après // ou # selon le langage, comme les lignes grises des exemples de ce livre: la machine l'ignore. [3.1 p. 63](#)

commit

Un commit pointe sur un tree et sur son parent. [6.3 p. 122](#)

compilateur

Le compilateur transforme tout le code en instructions pour la puce avant l'exécution, comme un livre traduit en entier avant d'être publié.

[1.4 p. 26](#)

composition

Composer, c'est assembler : « un utilisateur a une façon de se connecter » veut dire qu'il contient un objet à part, issu d'un autre moule, qu'on peut remplacer sans toucher au moule Utilisateur.

[4.1 p. 80](#)

concurrence

La concurrence, c'est avoir plusieurs tâches en cours, même si une seule avance à un instant donné, comme un cuisinier seul qui, tour à tour, fait avancer chaque commande. [2.7 p. 55](#)

configuration

La configuration est un réglage que le programme lit au démarrage au lieu de le porter dans son code, et ses valeurs diffèrent souvent entre les deux machines. [6.2 p. 119](#)

connexion

Une connexion est la conversation que deux machines ouvrent par une poignée de main, chacune gardant en mémoire où elle en est.

[5.3 p. 108](#)

constante

Une constante est une variable qu'on remplit une fois et qu'on ne réécrit jamais, souvent écrite en majuscules pour qu'on la reconnaisse d'un coup d'œil. [3.1 p. 63](#)

couture

Michael Feathers appelle ça une couture, un endroit où l'on change le comportement sans modifier la fonction elle-même. [6.1 p. 116](#)

débogueur

Le débogueur arrête le programme sur une ligne choisie, le point d'arrêt, et montre le contenu de chaque casier : c'est le déroulé à la main de 2.1, exécuté par la machine. [6.2 p. 119](#)

déborder la pile (stack overflow)

D'où une limite bien réelle : une fonction récursive qui s'appelle trop de fois avant d'atteindre le cas d'arrêt fait déborder la pile (en anglais stack overflow). [2.5 p. 49](#)

décorateur

Le décorateur : un post-it agrafé sur une fonction.

[2.4 p. 44](#)

défense en profondeur

Par-dessus, la défense en profondeur : chaque couche se protège elle-même, du navigateur à la base, pour que si l'une cède, les autres tiennent.

[7.6 p. 140](#)

dépassement d'entier

Un dépassement d'entier est un calcul dont le résultat ne tient plus dans les octets réservés : dans les langages à entiers de taille fixe, le nombre repart en général du bas de l'échelle, sans prévenir.

[1.1 p. 5](#)

dépendance

Une dépendance est du code que vous n'avez pas écrit mais dont votre programme a besoin pour fonctionner : toute bibliothèque téléchargée en est une, et ses bugs et ses mises à jour deviennent les vôtres. [2.4 p. 44](#)

design émergent

L'effet le plus profond du TDD (Test-Driven Development, le développement piloté par les tests) n'est pourtant pas d'attraper des bugs, c'est le design émergent. [6.1 p. 116](#)

détrompeur

Une forme qui rend le mauvais geste impossible s'appelle un détrompeur (poka-yoke en japonais).

[3.4 p. 69](#)

dette technique

Les raccourcis pris ici et là s'accumulent en dette technique : chaque raccourci rend le changement suivant plus cher, comme les intérêts d'un emprunt (l'image date de 1992, de Ward Cunningham).

[ch. 3 p. 62](#)

dictionnaire

Une liste range des valeurs à la suite, un dictionnaire les retrouve par un nom. [2.2 p. 35](#)

disque

Tout le reste vit plus loin, par étages : le cache (une réserve rapide collée à la puce, qui garde une copie des données récentes), puis la RAM (la mémoire vive, les barrettes de votre machine), puis le disque.

[1.2 p. 16](#)

DNS

Le DNS (Domain Name System) : l'annuaire du web traduit le nom du site en adresse IP, parce qu'un routeur sait acheminer vers une adresse, jamais vers un nom. [5.5 p. 112](#)

DOM

Une page web : son texte HTML (le langage qui décrit la page) est fait de balises emboîtées, un paragraphe dans une section dans la page, que le navigateur range en arbre, le DOM (Document Object Model). [2.5 p. 49](#)

donnée

Les ordres agissent sur des données, c'est-à-dire des nombres comme un prix, une lettre ou un pixel. [1.1 p. 5](#)

DRY

DRY (« Don't Repeat Yourself », ne te répète pas) est le principe qui veut qu'un même savoir, comme le taux de TVA, n'existe qu'à un seul endroit du programme. [3.5 p. 73](#)

échouer tôt

Échouer tôt (fail fast, formule popularisée par Jim Shore en 2004) est la règle qui veut qu'un programme s'arrête dès qu'il rencontre un cas sans réponse juste, plutôt que de continuer avec une valeur fausse. [3.4 p. 69](#)

éditeur de code

Un éditeur de code est le logiciel dans lequel on tape des lignes de code, en texte brut (VS Code, par exemple). [1.4 p. 26](#)

effet de bord

Un effet de bord est tout ce qu'une fonction change hors de son résultat : un texte affiché, une variable globale, une liste reçue qu'elle modifie sur place. [2.8 p. 59](#)

effondrement de congestion

L'effondrement de congestion est le cercle vicieux où les paquets renvoyés rallongent le bouchon qui les a fait perdre. [5.4 p. 110](#)

entité

Une entité a une identité qui survit à tous ses changements : le client 42 reste le client 42 s'il déménage et change de nom, on le suit dans le temps. [4.5 p. 96](#)

entrée

L'entrée d'un programme, c'est tout ce qu'il reçoit du dehors : une touche du clavier, un fichier sur le disque, un code-barres, un message d'une autre machine par le réseau (chapitre 5). [2.1 p. 31](#)

énumération

Un type qui énumère ses valeurs une à une, comme les trois couleurs, est une énumération (enum dans beaucoup de langages). [2.3 p. 39](#)

équipe alignée sur un flux

Alignée sur un flux : une équipe = un produit, livré seule de bout en bout (le cas par défaut). [7.4 p. 138](#)

équipe chirurgicale

Brooks rêvait en 1975 d'une équipe chirurgicale : un cerveau qui tient le scalpel (conçoit, décide), entouré d'assistants spécialisés. [8.4 p. 151](#)

équipe habilitante

Habilitante : aide une équipe à monter en compétence, puis se retire. [7.4 p. 138](#)

équipe plateforme

Plateforme : fournit des outils internes pour que les autres livrent sans attendre. [7.4 p. 138](#)

équipe soudée

DeMarco et Lister appellent équipe soudée (jelled team) un groupe si lié que le tout dépasse la somme des parties : peu de départs, une identité commune, une fierté partagée. [7.5 p. 139](#)

équipe sous-système compliqué

Sous-système compliqué : maintient un morceau trop pointu pour être partagé (un moteur de calcul, par exemple). [7.4 p. 138](#)

erreur

Face à une instruction impossible, beaucoup de langages arrêtent le programme et affichent une erreur, un message qui dit ce qui a échoué et à quelle ligne. [2.1 p. 31](#)

ETC

ETC (Easier To Change, plus facile à changer) veut qu'à chaque choix, on se pose une seule question : « est-ce que ça rendra le système plus facile, ou plus dur, à modifier ? ». [3.5 p. 73](#)

évaluation

Le métier appelle évaluation le verdict, venu de l'extérieur du modèle, sur ce qu'il produit. [8.3 p. 149](#)

exposant

En binaire, la machine range d'un côté les chiffres, qu'on appelle la mantisse, et de l'autre l'exposant, la puissance de 2 par laquelle il faut les multiplier. [1.1 p. 5](#)

fenêtre

TCP envoie donc toute une file de paquets, des camions sur notre autoroute, avant d'avoir des nouvelles du premier, plutôt qu'un camion, silence, puis le suivant. On appelle fenêtre le nombre de paquets autorisés à rouler sans nouvelles. [5.3 p. 108](#)

fenêtre de contexte

Un modèle ne lit qu'une quantité limitée de texte à la fois: sa fenêtre de contexte. [8.2 p. 147](#)

fichier

Un fichier est un contenu rangé sous un nom sur le disque: le texte tapé y est enregistré, `caisse.py` par exemple, où «`.py`» dit «du Python». [1.4 p. 26](#)

fine-tuning

Un dernier mot, que vous croiserez dans le métier: le fine-tuning, ré-entraîner un modèle sur ses propres données. [8.2 p. 147](#)

flottant

C'est ça, la «virgule flottante»: la virgule n'a pas de place fixe, l'exposant la fait glisser. Un nombre rangé ainsi s'appelle un flottant. [1.1 p. 5](#)

flux

Le franchir, ce n'est pas coder plus: c'est installer un flux qui prouve, livre et tient, sans héroïsme et sans nuits blanches. [ch. 6 p. 115](#)

fonction

Une fonction est une recette écrite une fois pour toutes: on lui donne des ingrédients, ses arguments, elle rend un plat, son résultat. [ch. 2 p. 30](#)

fonction de hachage

La clé passe dans une fonction de hachage qui la transforme en numéro de case: on ne cherche rien, on calcule où la valeur est rangée et on y va directement, quel que soit le nombre d'entrées. [1.3 p. 21](#)

fonction de rappel (callback)

Pour récupérer la réponse du serveur, on passe traditionnellement une fonction de rappel (en anglais `callback`), une fonction qu'on donne pour qu'elle soit exécutée plus tard: «quand tu auras fini, exécute ceci». [2.7 p. 55](#)

fonction profonde

La fonction profonde cache beaucoup de travail derrière une toute petite surface, c'est-à-dire ce que l'appelant doit connaître pour s'en servir (ici un nom et un argument): une ligne sur la carte, tout le reste en cuisine. [3.2 p. 65](#)

fonction pure

Une fonction pure ne fait qu'une chose: calculer son résultat à partir de ses arguments, sans effet de bord et sans rien lire de caché. [2.8 p. 59](#)

fonctions siamoises

Deux fonctions qu'on ne peut lire qu'ensemble sont des fonctions siamoises (`conjoined methods` dans A Philosophy of Software Design), et mieux vaut les fusionner en une seule. [3.2 p. 65](#)

formateur

Le formateur réécrit les espaces, les retours à la ligne et l'indentation pour que tout le projet ait la même allure, quel qu'en soit l'auteur. [3.1 p. 63](#)

fusion

Quand le travail est prêt, une fusion (`merge`) recolle les deux histoires: Git crée un commit à deux parents, un par branche. [6.3 p. 122](#)

générateur

Le générateur (`yield`): une fonction qui vous tend une valeur, s'arrête, et reprend à la suivante quand vous en redemandez. [2.4 p. 44](#)

GoF

Le GoF (le «Gang of Four», les quatre auteurs du livre fondateur *Design Patterns*) range ses 23 patterns en trois familles, et ces trois familles valent mieux que les 23 noms appris par cœur. [4.2 p. 84](#)

hallucination

L'hallucination (une affirmation fausse, livrée avec l'assurance d'une vraie) n'est donc pas un bug à corriger un jour, c'est le mécanisme même qui le fait fonctionner. [8.1 p. 145](#)

héritage

Hériter, c'est dire «un administrateur est un utilisateur», ce qu'écrit le mot-clé `extends`: le moule enfant (`Administrateur`) reprend tout du moule parent (`Utilisateur`) d'un coup, sans rien demander. [4.1 p. 80](#)

hexadécimal

L'hexadécimal est une écriture des nombres en base 16, où un chiffre, de 0 à 9 puis de a à f, remplace quatre bits: deux chiffres par octet. [1.1 p. 5](#)

HTTP

HTTP (`HyperText Transfer Protocol`) est la langue que parlent navigateurs et serveurs: demander une page, répondre avec son contenu. [5.5 p. 112](#)

HTTP/1

HTTP/1 ne sait demander qu'une ressource à la fois par connexion, et un navigateur n'en ouvre que six par domaine (par nom de site, comme `monsie.fr` ou `images.monsie.fr`). [5.5 p. 112](#)

HTTP/2

HTTP/2 a réglé le problème des connexions: toutes les requêtes voyagent entrelacées sur une seule connexion, donc une seule poignée de main, et une fenêtre qui reste ouverte et large. [5.5 p. 112](#)

HTTP/3

HTTP/3 va plus loin encore: il abandonne TCP pour un transport neuf, QUIC (prononcé « quick »), parce qu'une seule connexion TCP a un défaut. [5.5 p. 112](#)

idempotente

Une opération qu'on peut répéter sans effet double est dite idempotente. [5.2 p. 106](#)

immuable

Une donnée qu'on ne peut pas modifier sur place est immuable. [2.2 p. 35](#)

index

Un index est l'annuaire trié de la section 1.3, tenu à jour par le moteur pour une colonne donnée, afin qu'une recherche sur cette colonne descende en quelques étapes au lieu de dix millions. [4.6 p. 99](#)

injection de dépendances

Passer à une fonction ce dont elle a besoin (ici, la sortie), au lieu de la laisser le fabriquer, s'appelle l'injection de dépendances. [4.2 p. 84](#)

injection SQL

L'injection SQL montre le mécanisme en entier. [7.6 p. 140](#)

instance

Une instance est l'exemplaire fabriqué à partir d'une classe: l'objet vu en section 2.2, avec ses valeurs à lui (Dupont, dupont@exemple.fr). [4.1 p. 80](#)

instancier

Fabriquer un exemplaire à partir du moule s'appelle instancier la classe, ce que fait le mot-clé new dans beaucoup de langages (Java, C#, PHP, JavaScript). [4.1 p. 80](#)

instruction

Un programme est une liste d'instructions, de minuscules ordres que le processeur, la puce de calcul, sait exécuter: « additionne ces deux nombres », « copie cette case ailleurs », « saute au numéro 40 si le résultat vaut zéro ». [1.1 p. 5](#)

interface

Une interface est la liste de ce qu'on exige d'un objet: des champs, comme x et y pour un Point, ou des actions, comme lire et se fermer. [2.3 p. 39](#)

interpréteur

L'interpréteur traduit au fil de l'exécution, comme un interprète de conférence, phrase après phrase. [1.4 p. 26](#)

intersection

L'intersection (A & B) ne garde que les valeurs qu'ils ont en commun. [2.3 p. 39](#)

invariant

Un invariant est ce qui doit rester vrai quoi que fasse le programme: un total de caisse n'est jamais négatif, une commande payée a un montant. [3.4 p. 69](#)

IP

Au milieu du réseau, toutes les machines partagent un seul protocole commun, une règle de conversation qu'elles acceptent toutes: IP (Internet Protocol), qui donne une adresse à chaque machine et fait avancer les paquets vers elle. [5.2 p. 106](#)

jointure

Recoller une commande et son client à la lecture est une jointure: pour chaque commande, aller chercher la ligne du client qui porte ce numéro. [4.6 p. 99](#)

langage de programmation

Reste à choisir les mots dans lesquels on exprime son intention: le langage de programmation. [ch. 1 p. 4](#)

latence

La latence, c'est la longueur du trajet: le temps qu'un message met à faire l'aller-retour. [5.1 p. 105](#)

lead time

Lead time: temps entre un commit et sa mise en prod. [6.5 p. 126](#)

lever une exception

Quand une fonction n'y arrive pas, beaucoup de langages l'obligent à lever une exception: elle s'arrête net et envoie un signal d'erreur, l'exception, à son appelant. [3.4 p. 69](#)

ligne de cache

Autre réflexe: comme un coursier qui remplit une caisse, la machine ne rapporte jamais un seul octet, elle tire toute une ligne de cache (environ 64 octets) d'un coup. [1.2 p. 16](#)

linter

Le linter relit le code sans l'exécuter et signale ce qu'une machine sait voir : une variable jamais lue, un nom qui ne suit pas la convention du projet, un `==` là où il fallait `===` (2.6). [3.1 p. 63](#)

liste

Une liste range des valeurs à la suite, un dictionnaire les retrouve par un nom. [2.2 p. 35](#)

liste chaînée

Une liste chaînée ressemble à une chasse au trésor : chaque élément dit où trouver le suivant.

[1.3 p. 21](#)

localité

La localité est la tendance d'un programme à relire bientôt ce qu'il vient de lire (localité temporelle), et à lire ensuite les cases voisines de celle qu'il lit (localité spatiale). [1.2 p. 16](#)

loi de Brooks

L'intuition dit : projet en retard, on ajoute des développeurs. C'est faux, et ça porte un nom, la loi de Brooks. [7.5 p. 139](#)

loi de Conway

Loi de Conway : un système copie la structure de communication de l'organisation qui le construit. [7.4 p. 138](#)

loi de Déméter

La Loi de Déméter dit « ne parle qu'à tes voisins immédiats » : un objet n'appelle que ses propres méthodes, celles des objets qu'on lui passe et celles des objets qu'il garde en lui. [4.3 p. 87](#)

machine à états

Le statut est l'énumération de 2.3, et `annuler()` n'est permis que depuis certains de ses états : la commande est une machine à états, un objet qui ne passe d'un état à l'autre que par des transitions écrites une fois, dans le code. [4.5 p. 96](#)

mantisse

En binaire, la machine range d'un côté les chiffres, qu'on appelle la mantisse, et de l'autre l'exposant, la puissance de 2 par laquelle il faut les multiplier. [1.1 p. 5](#)

mass assignment

Le mass assignment glisse un champ en trop dans la requête, que le serveur enregistre sans se méfier. [7.6 p. 140](#)

méthode

Les données qu'un objet porte sont ses champs (nom, prix, quantité), les actions qu'il sait faire sont ses méthodes, des fonctions rangées dans l'objet (`appliquerRemise()`). [2.2 p. 35](#)

métriques DORA

Accelerate donne quatre chiffres pour se situer objectivement, les métriques DORA (DevOps Research and Assessment, l'équipe de recherche derrière le livre). [6.5 p. 126](#)

mock

Un faux appareil branché sur la prise s'appelle un mock (un faux qui répond ce qu'on lui a dit de répondre et qui, en plus, note ce qu'on lui demande). [6.1 p. 116](#)

Model-Driven Design

Le Model-Driven Design (la conception guidée par le modèle) d'Eric Evans demande que le modèle, c'est-à-dire la description du métier avec ses objets et ses règles, s'écrive directement dans le code. [4.5 p. 96](#)

modèle de langage

Un modèle de langage est le programme derrière ChatGPT ou Claude : il a été entraîné sur d'énormes masses de texte pour prédire le mot suivant. [8.1 p. 145](#)

moindre privilège

Enfin le moindre privilège : chaque programme et chaque compte n'a que les droits dont il a besoin. [7.6 p. 140](#)

MTTR

MTTR (Mean Time To Recovery) : temps pour rétablir le service après une panne, du début de l'incident au retour à la normale. [6.5 p. 126](#)

N+1

Le N+1 est une boucle qui lance une requête par tour au lieu d'une seule jointure, et les ORM (« mapping objet-relationnel »), ces outils qui écrivent le SQL à votre place, le produisent sans prévenir. [4.6 p. 99](#)

nombre magique

Un nombre magique est un nombre écrit tel quel dans le code, sans rien qui dise ce qu'il représente. [3.1 p. 63](#)

normalisation

La normalisation est le DRY de la section 3.5 appliqué aux données : chaque fait s'écrit une fois, le téléphone du client dans sa ligne, jamais dans ses commandes. [4.6 p. 99](#)

NoSQL

Une base documentaire (MongoDB en est l'exemple connu), l'une des familles dites NoSQL (« pas seulement SQL »), range chaque commande dans un dossier : un bloc de JSON qui contient tout, le client, les lignes, l'adresse. [6.6 p. 128](#)

numéro de suivi

Chaque octet envoyé porte un numéro de suivi. [5.3 p. 108](#)

O(1)

O(1) : temps constant, quelle que soit la taille (le plus rapide). [1.3 p. 21](#)

O(log n)

O(log n) : temps logarithmique, qui grimpe de moins en moins vite : on coupe le problème en deux à chaque étape. [1.3 p. 21](#)

O(n log n)

Un tri efficace prend environ $n \times \log n$ étapes : c'est la famille O(n log n), entre la droite de O(n) et l'explosion de O(n²). [1.3 p. 21](#)

O(n)

O(n) : temps linéaire, deux fois plus de données font deux fois plus de travail. [1.3 p. 21](#)

O(n²)

O(n²) : temps quadratique, deux fois plus de données en font quatre fois plus (le pire ici). [1.3 p. 21](#)

objet

Un objet regroupe des données et les actions qu'on peut lui demander : l'article en est un, avec son nom, son prix et sa quantité. [2.2 p. 35](#)

objet-valeur

Un objet-valeur n'est que son contenu : deux billets de 10 € sont interchangeables, deux adresses identiques sont la même adresse. [4.5 p. 96](#)

observabilité

L'observabilité va plus loin : c'est pouvoir dire ce qui se passe dans un système depuis l'extérieur, en lisant ce qu'il écrit sur lui-même, sans l'ouvrir ni le redéployer. [6.5 p. 126](#)

observateur

Observateur : prévenir une liste d'abonnés quand une donnée change. [4.2 p. 84](#)

octet

On groupe les interrupteurs par huit, et un groupe de huit s'appelle un octet. [1.1 p. 5](#)

paquet

D'abord, le réseau ne transporte jamais un message d'un bloc. Il le découpe en paquets : des petits morceaux indépendants, comme les pages d'un livre expédiées dans des enveloppes séparées. [5.2 p. 106](#)

paradigme

Un paradigme est une façon de penser et d'écrire un programme. [2.8 p. 59](#)

parallélisme

Le parallélisme, c'est plusieurs tâches qui avancent au même instant, une par cœur, comme deux cuisiniers côte à côte. [2.7 p. 55](#)

peu profonde

Une fonction qui ne cache presque rien est dite peu profonde : appliquer TVA(prix), si elle se résume à prix * 1.2, ne simplifie rien. [3.2 p. 65](#)

pile d'appels

Chaque appel doit donc noter l'instruction où reprendre et ses propres valeurs (le n qui vaut 3, puis 2), et ces notes s'empilent, une par carton ouvert, tant que le plus petit n'est pas vidé : c'est la pile d'appels. [2.5 p. 49](#)

plan d'exécution

Le plan d'exécution est la méthode que le moteur a choisie pour répondre à une requête. [4.6 p. 99](#)

poignée de main TCP

Avant le moindre octet utile, les deux machines échangent une poignée de main en trois temps. [5.3 p. 108](#)

point d'arrêt

Le débogueur arrête le programme sur une ligne choisie, le point d'arrêt, et montre le contenu de chaque casier : c'est le déroulé à la main de 2.1, exécuté par la machine. [6.2 p. 119](#)

pointeur

Une adresse rangée dans une variable s'appelle un pointeur, ou une référence. [2.2 p. 35](#)

portée

La portée d'une variable est la zone du programme où son nom est visible. [2.1 p. 31](#)

processeur

Le processeur (la puce qui exécute vos instructions une par une) ne calcule vite que sur ce qui est tout près de lui. [1.2 p. 16](#)

processus

Pour chaque programme lancé, le système d'exploitation crée un processus: le programme en cours d'exécution, avec sa propre zone de mémoire, partagée par tous ses threads. [2.7 p. 55](#)

production

Du code qui « marche sur ma machine » à un service qui tient en production, la machine ouverte au public où des milliers de gens l'utilisent pour de vrai, il y a un gouffre. [ch. 6 p. 115](#)

profileur

Le profileur est un programme qui observe le vôtre pendant qu'il tourne et dit où passe le temps, par fonction (un morceau de code nommé) ou par ligne. [1.4 p. 26](#)

programmation fonctionnelle

La programmation fonctionnelle fait voyager les fonctions comme des données (2.4) et veut qu'elles ne modifient jamais ce qu'elles reçoivent: au lieu de changer la liste reçue, une fonction en rend une nouvelle, avec le changement fait. [2.8 p. 59](#)

programmation orientée objet (POO)

La programmation orientée objet (POO) regroupe, dans un même objet, des données et les actions qu'on peut lui demander, comme l'article de la caisse (2.2, puis tout le chapitre 4). [2.8 p. 59](#)

programmation procédurale

La programmation procédurale écrit le programme comme une suite d'instructions suivies dans l'ordre, regroupées en fonctions. [2.8 p. 59](#)

programmer pour une interface

Au-dessus des 23 patterns, le GoF pose deux principes fondateurs: la composition plutôt que l'héritage (4.1), et programmer pour une interface, pas pour une implémentation. [4.2 p. 84](#)

promesse

Une promesse est une valeur qui n'est pas encore là mais qui arrivera plus tard: une fonction comme get en renvoie une, au lieu de la valeur elle-même. [2.7 p. 55](#)

prompt

Le principe: chercher les documents pertinents dans une base externe et les coller dans le prompt (le message qu'on envoie au modèle), pour que celui-ci s'appuie sur des faits fournis, pas sur sa mémoire floue. [8.2 p. 147](#)

protocole

Au milieu du réseau, toutes les machines partagent un seul protocole commun, une règle de conversation qu'elles acceptent toutes: IP (Internet Protocol), qui donne une adresse à chaque machine et fait avancer les paquets vers elle.

[5.2 p. 106](#)

pseudo-code

Un pseudo-code décrit un calcul en phrases presque françaises: on le réécrit ensuite dans le langage de son choix, et c'est la bonne habitude, penser d'abord, traduire après. [2.1 p. 31](#)

quantum d'architecture

Premier outil, le quantum d'architecture: le plus petit morceau d'un système qu'on peut mettre en ligne et tester seul, et qui peut tomber sans rien entraîner d'autre. [4.4 p. 94](#)

QUIC

HTTP/3 va plus loin encore: il abandonne TCP pour un transport neuf, QUIC (prononcé « quick »), parce qu'une seule connexion TCP a un défaut. [5.5 p. 112](#)

race condition

Deux fils d'exécution dans le même programme (deux threads, les cuisiniers de la section 2.7, ou deux goroutines, les fils légers de Go) qui lisent et écrivent la même variable rejouent le write skew en miniature: chacun agit sur un monde déjà périmé, et ça s'appelle une race condition. [6.7 p. 131](#)

RAG

Le pattern-roi s'appelle RAG (Retrieval-Augmented Generation: générer en s'appuyant sur des documents récupérés). [8.2 p. 147](#)

RAM

Tout le reste vit plus loin, par étages: le cache (une réserve rapide collée à la puce, qui garde une copie des données récentes), puis la RAM (la mémoire vive, les barrettes de votre machine), puis le disque. [1.2 p. 16](#)

ramasse-miettes

Le ramasse-miettes est le mécanisme du langage qui libère la mémoire que plus aucune variable n'atteint: quand plus rien ne tient inc, elle disparaît, et son n avec elle. [2.4 p. 44](#)

récurtivité

Si une fonction est une recette, la récursivité est une recette qui se cite elle-même. [2.5 p. 49](#)

registre

Le plus près, ce sont ses registres, quelques cases au cœur de la puce: les seules où il calcule. [1.2 p. 16](#)

règle de dépendance

La règle de dépendance de Clean Architecture interdit une seule chose: que le code métier dépende de la technique. [4.3 p. 87](#)

règle de trois

La règle de trois, que Fowler tient de Don Roberts, tient en trois temps: la première fois on écrit, la deuxième on copie en grimaçant, la troisième on refactore. [3.6 p. 75](#)

règle du boy-scout

Le réflexe quotidien, c'est la règle du boy-scout de Clean Code: comme le scout qui quitte le campement plus propre qu'il ne l'a trouvé, laissez toujours le fichier un peu plus propre qu'à votre arrivée. [3.6 p. 75](#)

réplication

Quand ça ne suffit plus, on copie: la réplication duplique la base sur plusieurs machines, et chacune sert sa part des lectures. [6.6 p. 128](#)

requêtes préparées

Contre l'injection, les requêtes préparées: l'entrée voyage à part de la requête et ne peut jamais redevenir du code. [7.6 p. 140](#)

résultat

Une fonction est une recette écrite une fois pour toutes: on lui donne des ingrédients, ses arguments, elle rend un plat, son résultat. [ch. 2 p. 30](#)

revue de code

On propose sa branche, ce qu'on appelle une pull request (une demande de fusion), avec la liste exacte des lignes changées, et un collègue la relit avant qu'elle soit recollée. C'est la revue de code: le prochain humain du chapitre 3 arrive plus tôt que prévu, pas dans six mois mais demain matin. [6.3 p. 122](#)

routeur

Chaque paquet porte l'adresse de destination et passe de proche en proche par des routeurs, les relais qui l'orientent, dont le premier est la box de votre fournisseur d'accès. [5.2 p. 106](#)

saga

Faute d'annulation globale, on écrit la marche arrière à la main: c'est la saga. [6.7 p. 131](#)

schéma

La liste des tables, de leurs colonnes et du type de chaque colonne forme le schéma: la forme des données, décidée avant la première ligne, que le moteur fait respecter à chaque écriture. [4.6 p. 99](#)

secret

Un secret, mot de passe ou clé d'une API, est une valeur qui ne vit jamais dans le code: elle arrive par la configuration, une variable d'environnement ou un fichier que Git ignore. [6.3 p. 122](#)

sharding

Le sharding répartit les données par tranches (clients A-M ici, N-Z là), et chaque machine n'encaisse que sa part. [6.6 p. 128](#)

signature

La première ligne d'une fonction, son nom suivi de ce qu'elle attend, s'appelle sa signature: avec des types, elle devient une documentation que le compilateur vérifie. [2.3 p. 39](#)

slow start

La connexion ne connaît pas encore la capacité de la route, alors elle commence petit (une dizaine de paquets, environ 14 ko) et double sa fenêtre à chaque aller-retour: c'est le slow start (démarrage lent, mais seulement par rapport à tout envoyer d'un coup). [5.4 p. 110](#)

SOLID

L'inversion de dépendance porte une lettre: c'est le D de SOLID, cinq principes de conception réunis sous un acronyme. [4.3 p. 87](#)

sortie

La sortie d'un programme, c'est tout ce qu'il envoie au dehors: un texte à l'écran, un fichier écrit, un message sur le réseau. [2.1 p. 31](#)

SQL

Le langage standard pour parler au moteur est SQL (Structured Query Language, langage de requête structuré), et une requête SQL, c'est-à-dire une question posée à la base, dit ce qu'on veut, pas comment l'obtenir. [4.6 p. 99](#)

stratégie

Stratégie: changer de façon de faire pendant que le programme tourne, exactement ce qu'on vient de faire en remplaçant le vol du canard. [4.2 p. 84](#)

structure de données

Le Big O sert aussi à choisir une structure de données, la façon dont on range une collection en mémoire. [1.3 p. 21](#)

surface

La fonction profonde cache beaucoup de travail derrière une toute petite surface, c'est-à-dire ce que l'appelant doit connaître pour s'en servir (ici un nom et un argument): une ligne sur la carte, tout le reste en cuisine. [3.2 p. 65](#)

système d'exploitation

Le système d'exploitation (Windows, Linux ou macOS) est le programme qui gère la machine, un peu comme un chef de cuisine. [2.7 p. 55](#)

table

Le modèle relationnel, posé en 1970 par Edgar Codd, range chaque sorte de chose dans une table: une grille à colonnes fixes, une ligne par chose, une table Clients, une table Commandes. [4.6 p. 99](#)

table de hachage

Une table de hachage range chaque valeur sous une clé, par exemple la fiche d'un client sous son nom: pour la retrouver, on donne le nom et on obtient la fiche d'un coup, en temps constant (en moyenne), même sur un million d'entrées. [1.3 p. 21](#)

tableau

Un tableau pose ses éléments côte à côte, comme des casiers numérotés: atteindre le 500e est instantané, on calcule son adresse et on y va, mais insérer au milieu oblige à décaler tous les suivants d'une case. [1.3 p. 21](#)

TCP

Il faut donc combler l'écart entre ce qu'IP promet et ce dont une application a vraiment besoin. C'est le rôle de TCP (Transmission Control Protocol), installé dans les deux machines des extrémités, juste au-dessus d'IP. [5.3 p. 108](#)

TDD

L'effet le plus profond du TDD (Test-Driven Development, le développement piloté par les tests) n'est pourtant pas d'attraper des bugs, c'est le design émergent. [6.1 p. 116](#)

teamicide

En revanche, on la tue vite, et ils ont un mot pour ça: le teamicide, tout ce qui défait une équipe ou l'empêche de se souder. [7.5 p. 139](#)

terminal

Un terminal est une fenêtre où l'on tape ses demandes à la machine, ligne par ligne. [1.4 p. 26](#)

test de caractérisation

Devant du vieux code qui n'a pas de tests, on écrit d'abord un test de caractérisation: il note ce que le code fait vraiment, pas ce qu'il devrait faire, avant qu'on le change. [6.1 p. 116](#)

test unitaire

Un test qui isole une fonction derrière des faux est un test unitaire. [6.1 p. 116](#)

thread

Un thread (fil d'exécution) est une suite d'instructions que la puce suit l'une après l'autre. [2.7 p. 55](#)

timeout

Fixer un délai au-delà duquel on n'attend plus: c'est le timeout. [5.2 p. 106](#)

TLS

TLS (Transport Layer Security): le chiffrement, le cadenas du https (le « s » veut dire « sécurisé »). [5.5 p. 112](#)

token

On la mesure en tokens, des morceaux de mots (un mot courant en fait un ou deux). [8.2 p. 147](#)

trace d'appels

Quand un programme s'arrête sur une erreur (2.1), le message est suivi de la liste des appels en cours, un par carton ouvert: c'est la trace d'appels (stack trace). [2.5 p. 49](#)

transaction

Les deux premières garanties viennent de la transaction, un groupe de changements qui réussit en entier ou pas du tout, sans se mêler à ceux des autres. [4.6 p. 99](#)

tree

Un tree est la liste des fichiers d'un dossier à un instant donné. [6.3 p. 122](#)

typage dynamique

Le typage dynamique les vérifie pendant qu'il tourne (Python, JavaScript). [2.3 p. 39](#)

typage nominal

Le typage nominal (Java, PHP) fait l'inverse: l'objet doit déclarer noir sur blanc implements Point, « je suis un Point », pour être accepté. [2.3 p. 39](#)

typage statique

Le typage statique vérifie les types avant que le programme tourne (Java, ou TypeScript, expliqué plus bas). [2.3 p. 39](#)

typage structurel

La vision en ensembles explique aussi le typage structurel: un objet entre dans un type selon sa forme, pas selon l'étiquette qu'il porte. [2.3 p. 39](#)

type

Chaque sorte de donnée est un type. [2.3 p. 39](#)

Ubiquitous Language

Evans demande que l'équipe et l'expert choisissent ensemble les mots exacts du métier et les emploient partout, en réunion comme dans le code: c'est l'Ubiquitous Language (« langage omniprésent »), une seule langue partagée. [4.5 p. 96](#)

union

L'union (A | B) réunit toutes les valeurs des deux types. [2.3 p. 39](#)

UTF-8

L'UTF-8 est la convention qui range chaque caractère sur 1 à 4 octets: 'A' garde son octet, 'é' en prend deux, '🍌' quatre. [1.1 p. 5](#)

variable

Une variable est un nom collé sur une case de la mémoire, comme une étiquette sur un casier. [ch. 2 p. 30](#)

verbes HTTP

L'API, le guichet de 4.5, est le contrat par lequel un programme en appelle un autre, le plus souvent par-dessus le web. Son vocabulaire universel, ce sont les verbes HTTP: GET pour lire, POST pour créer, PUT/PATCH pour modifier, DELETE pour supprimer. [7.2 p. 136](#)

WCAG

Tout ça est cadré par un référentiel mondial, les WCAG (Web Content Accessibility Guidelines): quatre principes (Perceptible, Utilisable, Compréhensible, Robuste) et trois niveaux d'exigence (A, AA, AAA). [7.3 p. 137](#)

write skew

La plus vicieuse, le write skew: deux transactions, chacune parfaitement valide, qui cassent une règle ensemble. [6.7 p. 131](#)

XSS

Le XSS (Cross-Site Scripting) injecte non pas du SQL mais un bout de script, glissé dans la page par une balise HTML, qui s'exécutera dans le navigateur des autres visiteurs. [7.6 p. 140](#)

YAGNI

YAGNI (« You Aren't Gonna Need It », tu n'en auras pas besoin) est le contrepoids de DRY et d'ETC: la règle, venue de l'équipe de Kent Beck, qui déconseille de construire d'avance ce que personne n'a encore demandé. [3.6 p. 75](#)

Colophon

Texte composé en Literata, titres en Fraunces, légendes en Lato, code en JetBrains Mono.

Cette édition est tirée de la version en ligne, qui reste la référence et s'améliore au fil des relectures : web-developpeur.com/bibliotheque/le-grand-livre/