

BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 3

Principes et fondamentaux de conception

Toute architecture repose sur les mêmes quelques idées de structure : comment diviser un système en parties, ce que ces parties doivent savoir les unes des autres, dans quel sens leurs dépendances doivent pointer et comment garder l'ensemble aussi simple que le problème le permet. Ce chapitre enseigne ces idées à toutes les échelles, d'une classe Python à un composant, puis à une flotte de services : modularité, couplage et cohésion, masquage de l'information, gestion des dépendances, SOLID, connascence, loi de Conway, simplicité et métriques qui mesurent la structure. Il s'adresse aux développeurs qui veulent raisonner avec précision sur la conception, et aux architectes qui ont besoin d'un vocabulaire commun pour les revues.

bewiseapps.com/fr/software-architecture-wise/design-principles/ · Version 2026-11-03. Information pédagogique uniquement, pas un conseil professionnel.

EN UN COUP D'ŒIL

Vous apprendrez à

- Expliquer le critère de Parnas pour décomposer un système en modules et l'appliquer pour comparer deux décompositions candidates.
- Classer les dépendances selon les niveaux classiques de couplage et de cohésion, et identifier le couplage temporel, de déploiement et sémantique entre services.
- Distinguer le masquage de l'information de l'encapsulation, spécifier une interface comme un contrat et concevoir des API stables qui tiennent compte des abstractions qui fuient et de la loi de Hyrum.
- Calculer l'instabilité, le degré d'abstraction et la distance à la séquence principale, et s'en servir avec les principes des dépendances acycliques, des dépendances stables et des abstractions stables pour corriger des structures de dépendances.
- Appliquer les principes SOLID et les principes de cohésion des composants à l'échelle des classes, des composants et des services, et reconnaître quand ils sont appliqués de façon dogmatique.
- Utiliser la connascence pour décrire la force, la localité et le degré du couplage, et choisir des refactorisations qui l'affaiblissent.
- Expliquer la loi de Conway, les données qui l'étayent et la manœuvre de Conway inversée, et calculer les canaux de communication d'une équipe.
- Distinguer la complexité essentielle de la complexité accidentelle, calculer la complexité cyclomatique et transformer des règles structurelles en fonctions d'aptitude sans tomber dans le piège de la loi de Goodhart.

Avant de commencer

Les chapitres 0 à 2, en particulier la section 0.3 (organiser le code), la section 1.7 (dette technique) et la section 2.6 (modifiabilité). Savoir lire de courts exemples en Python est utile.

Durée

Environ 3 heures de lecture ; 45 minutes pour le test.

Termes clés

modularité · masquage de l'information · couplage · cohésion · couplage afférent et efférent · abstraction · conception par contrat · Loi de Hyrum · Principe des dépendances acycliques · instabilité et degré d'abstraction · SOLID · connascence · Loi de Conway · complexité accidentelle

Pourquoi des principes, et pourquoi à toutes les échelles

Le chapitre 2 a décrit les qualités qu'une architecture doit fournir. La plupart d'entre elles, et la modifiabilité avant tout, dépendent moins d'une technologie particulière que de la structure : la façon dont un système est divisé en parties, ce que chaque partie sait des autres et le sens dans lequel vont les dépendances entre elles. Une équipe peut choisir le framework le plus en vogue et la meilleure plateforme cloud, et se retrouver malgré tout avec un système où chaque

changement touche dix endroits, parce que les parties ont été découpées aux mauvais endroits. Les principes de ce chapitre sont le savoir accumulé en plus de cinquante ans sur la manière de bien les découper.

Ce sont de vieilles idées, et c'est une force. Le masquage de l'information date de 1972, le couplage et la cohésion des travaux de conception structurée de la fin des années 1960 et des années 1970, la loi de Conway de 1968. Elles ont survécu parce qu'elles décrivent quelque chose qui ne change pas : le coût d'un changement dépend de la distance qu'il doit parcourir. Ce qui a changé, c'est l'échelle à laquelle les architectes les appliquent. Le même principe qui vous dit de garder privés les champs d'une classe vous dit de garder privée la base de données d'un service ; le même raisonnement qui interdit un cycle de dépendances entre deux paquets interdit deux services qui ne peuvent être mis en production qu'ensemble. Chaque section montre donc chaque idée deux fois : dans un petit morceau de code, et à l'échelle des composants et des services.

Organisation du chapitre

Les quatre premières sections posent les fondations. La section 3.1 se demande ce qu'est un module et où doivent passer ses frontières ; elle s'appuie sur l'exemple KWIC de David Parnas pour montrer pourquoi une décomposition selon les étapes de traitement échoue et une décomposition selon les changements probables réussit, et elle distingue les modules logiques des unités physiques que l'on construit et que l'on déploie. La section 3.2 donne le vocabulaire pour juger les frontières : les niveaux classiques de couplage et de cohésion, le couplage afférent et efférent, et les formes de couplage qui n'apparaissent qu'entre services. La section 3.3 examine l'interface elle-même : masquage de l'information et encapsulation, contrats, abstractions qui fuient, modules profonds et loi de Hyrum. La section 3.4 s'intéresse au sens des dépendances : les principes de Robert C. Martin pour les dépendances entre composants, les métriques d'instabilité et de degré d'abstraction, la manière de rompre les cycles, l'organisation en couches, et la façon de vérifier tout cela automatiquement.

Les quatre sections suivantes s'appuient sur elles. La section 3.5 couvre les cinq principes SOLID et les principes de cohésion des composants, chacun illustré dans le code et entre services, ainsi que les critiques qu'ils suscitent lorsqu'on les applique mécaniquement. La section 3.6 présente la connascence, une mesure plus fine du couplage, assortie d'un ensemble clair de règles pour la rendre plus faible et plus locale. La section 3.7 élargit le regard à l'organisation : la loi de Conway, les données qui l'étayent, l'arithmétique des canaux de communication et la manœuvre de Conway inversée. La section 3.8 conclut sur la simplicité et la mesure : complexité essentielle et complexité accidentelle, KISS, YAGNI et DRY bien compris, le prix du maintien des options ouvertes, les métriques structurelles comme la complexité cyclomatique, et les limites de toutes les métriques.

Une seule idée sous-jacente

Lues ensemble, les huit sections forment un seul argument. Le changement est le coût dominant du logiciel : une bonne structure est donc une structure dans laquelle les changements probables restent locaux. Chaque principe du chapitre est une manière d'y parvenir. Parnas dit de placer chaque changement probable à l'intérieur d'un seul module. Le couplage et la cohésion disent de garder proches les choses qui changent ensemble et d'éloigner celles qui changent séparément. Le masquage de l'information dit de veiller à ce que ce qui peut changer soit invisible de l'extérieur. Les principes de dépendance disent de faire pointer les dépendances vers ce qui change le moins. La connascence dit de garder aussi proches que possible deux morceaux de code liés par un accord fort. La loi de Conway ajoute que les personnes font partie de la structure : un changement qui exige l'accord de deux équipes n'est pas local, quelle que soit l'organisation du code.

Cet argument explique aussi pourquoi les principes entrent parfois en conflit et pourquoi aucun n'est absolu. Supprimer une duplication peut augmenter le couplage ; ajouter des interfaces pour inverser une dépendance peut rendre un module moins profond ; scinder un composant pour améliorer sa cohésion peut créer un saut réseau. Le savoir-faire professionnel ne consiste pas à appliquer chaque principe partout, mais à se demander, pour un changement précis qui est probable dans un système précis, quelle distance il devrait parcourir, et quel principe raccourcit cette distance au moindre coût.

Comment utiliser ce chapitre

Si la conception est nouvelle pour vous, vous pouvez suivre les sections dans l'ordre ; chacune commence par une courte explication tirée de la vie quotidienne et renvoie à la section 0.3, où les bases ont été présentées. Les ingénieurs expérimentés voudront peut-être commencer par les exemples commentés : la décomposition d'une boutique en ligne dans la section 3.1, la refactorisation des commandes et de la facturation dans la section 3.2, l'interface de paiement dans la section 3.3 et le calcul des métriques dans la section 3.4 sont conçus pour être discutés. Les principes présentés ici servent dans tout le reste du cours : les styles architecturaux des chapitres 5 et 6 sont pour une large part des réponses différentes aux questions de savoir où placer les frontières et dans quel sens pointent les dépendances, et le chapitre 11 fait des vérifications automatisées introduites ici une pratique de gouvernance grâce aux fonctions d'aptitude.

SECTION 3.1 SUR 3.8 · INTERMÉDIAIRE · 28 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Modularité : diviser un système en parties

AVANT DE LIRE

Cette section s'appuie sur :

- 0.3 Organiser le code
- 2.6 Modifiabilité et déployabilité

Termes déjà vus qu'elle utilise : Dépendance, Bibliothèque, Monolithe, Érosion de l'architecture, Fonction d'aptitude (fitness function)

L'IDÉE EN BREF

Pensez à un grand hôpital. Ce n'est pas une immense salle où tout le monde fait tout ; il est divisé en services : radiologie, pharmacie, maternité. Chacun a son propre personnel et sa propre façon de travailler, et les autres s'adressent à lui par un accueil, avec des formulaires de demande clairs. Si la pharmacie change son système de gestion des stocks, les autres services ne s'en aperçoivent pas, car ils remplissent toujours le même formulaire. Le logiciel est divisé de la même façon en modules. Tout l'art consiste à choisir où placer les murs, pour que les changements attendus restent à l'intérieur d'un seul service.

CE QU'IL FAUT SAVOIR

- Un module est une unité de décomposition dotée d'une responsabilité et d'une interface : une fonction, une classe, un paquet, un composant ou un service, selon l'échelle à laquelle vous travaillez.
- La modularité est payante grâce à la compréhension, au travail en parallèle des équipes, à l'isolation des changements, à la réutilisation et aux tests, mais seulement si les frontières sont tracées aux bons endroits.
- David Parnas (1972) a montré avec l'exemple KWIC que décomposer selon les étapes de traitement disperse chaque décision de conception dans de nombreux modules, et a proposé à la place que chaque module cache une décision difficile ou susceptible de changer.
- Les modules logiques (unités de responsabilité dans la conception) se distinguent des modules physiques (unités de construction et de déploiement) ; une même conception logique peut être déployée en une seule unité ou en de nombreux services.

- La granularité est un compromis : des modules trop grands mélangent des changements sans rapport entre eux, tandis que des modules trop petits multiplient les interfaces, la coordination et, à travers un réseau, la latence et les modes de défaillance.
- Les frontières qui ne sont pas imposées s'érodent : les professionnels utilisent donc la visibilité du langage, des systèmes de modules comme le Java Platform Module System, les modules des outils de build et des vérifications automatisées des dépendances.

Ce qu'est un module

Un module est une partie d'un système qui a une responsabilité claire et une manière définie dont les autres parties peuvent l'utiliser. Cette définition est volontairement indépendante de la taille. Dans la section 0.3, un module était un fichier Python ; dans ce chapitre, le mot couvre toute unité de décomposition, c'est-à-dire l'action de diviser un système en parties : une fonction, une classe, un paquet, une bibliothèque, un composant à l'intérieur d'une application, ou un service déployé séparément. Les mêmes questions se posent à toutes les échelles : de quoi cette partie est-elle responsable, qu'offre-t-elle aux autres, que garde-t-elle pour elle, et de quoi dépend-elle ?

Tout module a deux faces. Son interface est ce que les autres modules doivent savoir pour l'utiliser : noms et paramètres des fonctions, une API HTTP, les événements qu'il publie, mais aussi les promesses informelles, comme sa rapidité de réponse et les erreurs qu'il peut renvoyer. Son implémentation est tout le reste : le code, les structures de données, les tables de la base de données, les bibliothèques tierces qu'il se trouve utiliser. Une bonne modularité garde l'interface petite et stable, et laisse l'implémentation changer librement.

David Parnas, dont l'article de 1972 est le fondement de cette section, définissait un module comme une *attribution de responsabilité* plutôt que comme un morceau de texte de programme. C'est la façon la plus utile d'y penser quand on est architecte. Un module est la promesse qu'une partie du système, et en général une équipe, est responsable d'un ensemble de décisions. Que le code finisse dans un fichier ou dans quarante, dans un processus ou dans dix, est une question distincte, traitée plus bas à propos des modules logiques et physiques.

SI VOUS DÉBUTEZ

Si vous débutez

Imaginez une application de livraison de repas. Quelque part dans son code se trouve la logique qui affiche les cartes, celle qui encaisse le paiement, celle qui suit les livreurs et celle qui envoie les notifications. La modularité signifie simplement que chacune de ces tâches vit dans sa propre partie, clairement séparée, avec un petit nombre de points d'entrée convenus. Le reste de cette section explique où placer les murs et comment empêcher les gens de les escalader.

Pourquoi modulariser

Un système de quelques centaines de lignes peut être compris dans son ensemble. Un système de quelques millions de lignes ne le peut pas, et l'organisation qui le construit ne peut pas non plus travailler comme un seul groupe indivisé. La modularité est la propriété d'être divisé en modules qui peuvent être compris, modifiés et analysés en grande partie isolément. C'est l'une des cinq sous-caractéristiques de la maintenabilité dans ISO/IEC 25010 (section 2.6), et elle procure au moins cinq avantages distincts.

La compréhension. Un développeur peut étudier un module et ses interfaces sans lire le reste du système. **Le travail en parallèle.** Les équipes peuvent travailler sur des modules différents en même temps avec peu de besoin de coordination, parce qu'elles se sont mises d'accord à l'avance sur les interfaces. **L'isolation des changements.** Un changement portant sur une décision de conception, par exemple les règles fiscales applicables, est fait dans un seul module et ne se propage pas. **La réutilisation.** Un module bien délimité, comme un composant de validation d'adresses, peut être utilisé par plusieurs produits. **Les tests.** Un module doté d'une interface claire peut être testé isolément, ses dépendances étant remplacées par des doublures de test.

Parnas en a cité trois en 1972 : l'avantage managérial (un temps de développement raccourci, parce que des groupes distincts travaillent sur des modules distincts en communiquant peu), la flexibilité (des changements radicaux dans un module sans modifier les autres) et la compréhensibilité (étudier le système un module à la fois). Un demi-siècle plus tard, ce sont toujours les raisons. Mais remarquez qu'aucun de ces avantages ne découle automatiquement du découpage d'un système en morceaux. Un système peut être divisé en cinquante modules et ne procurer aucun de ces avantages si chaque changement en touche dix. Les avantages dépendent entièrement de l'endroit *où* les frontières sont tracées.

La séparation des préoccupations

Edsger Dijkstra a donné son nom à cette idée dans un court essai de 1974, « On the role of scientific thought » (EWD 447). Il décrivait une façon de penser avant une façon de construire : vous étudiez en profondeur un aspect d'un problème, par exemple si un programme est correct, tout en sachant que d'autres existent, par exemple s'il est efficace ou si quelqu'un en veut. Il appelait cela avoir l'esprit « à la fois à une seule voie et à plusieurs voies », et il admettait que la séparation n'est jamais parfaite. Une *préoccupation* est simplement quelque chose à quoi vous devez penser.

En conception logicielle, la **séparation des préoccupations** consiste à organiser un système de sorte que chaque partie traite une seule préoccupation et puisse être comprise et modifiée sans garder les autres à l'esprit. Sur la plateforme de livraison de repas, calculer le prix d'une commande, estimer son délai de livraison, encaisser le paiement et envoyer des notifications sont des préoccupations distinctes. Si elles sont emmêlées dans une longue fonction, modifier le texte d'une notification oblige à lire les règles de tarification, et un test de tarification doit mettre en place un paiement. Une fois séparée, chacune peut être raisonnée, testée et remplacée isolément. La même réflexion sous-tend beaucoup de structures familières : HTML pour le contenu et CSS pour la présentation, le modèle et la vue, une règle métier à part de la base de données qui stocke son résultat.

C'est le plus large des principes de ce chapitre, et les autres le précisent. La modularité fournit les unités. Le masquage de l'information (section 3.3) indique où tracer la ligne : autour des décisions susceptibles de changer. Le principe de responsabilité unique (section 3.5) indique aux besoins de qui une partie doit répondre.

Deux mises en garde s'imposent. D'abord, certaines préoccupations refusent de tenir en un seul endroit. La journalisation, l'autorisation et les nouvelles tentatives touchent presque tout ; on les appelle donc des **préoccupations transversales** ; elles sont généralement prises en charge par un middleware partagé, une passerelle ou un service mesh (section 8.5), afin que le code métier n'ait pas à les répéter. Ensuite, une mauvaise séparation peut aggraver les choses. Diviser strictement en couches techniques sépare l'interface utilisateur des données, mais disperse une préoccupation métier, comme le pourboire au livreur, dans chaque couche, si bien qu'une seule fonctionnalité les touche toutes. C'est pourquoi la section 5.3 organise les modules autour des capacités métier. Demandez-vous quelles préoccupations changent indépendamment, et séparez celles-là.

Parnas et l'exemple KWIC

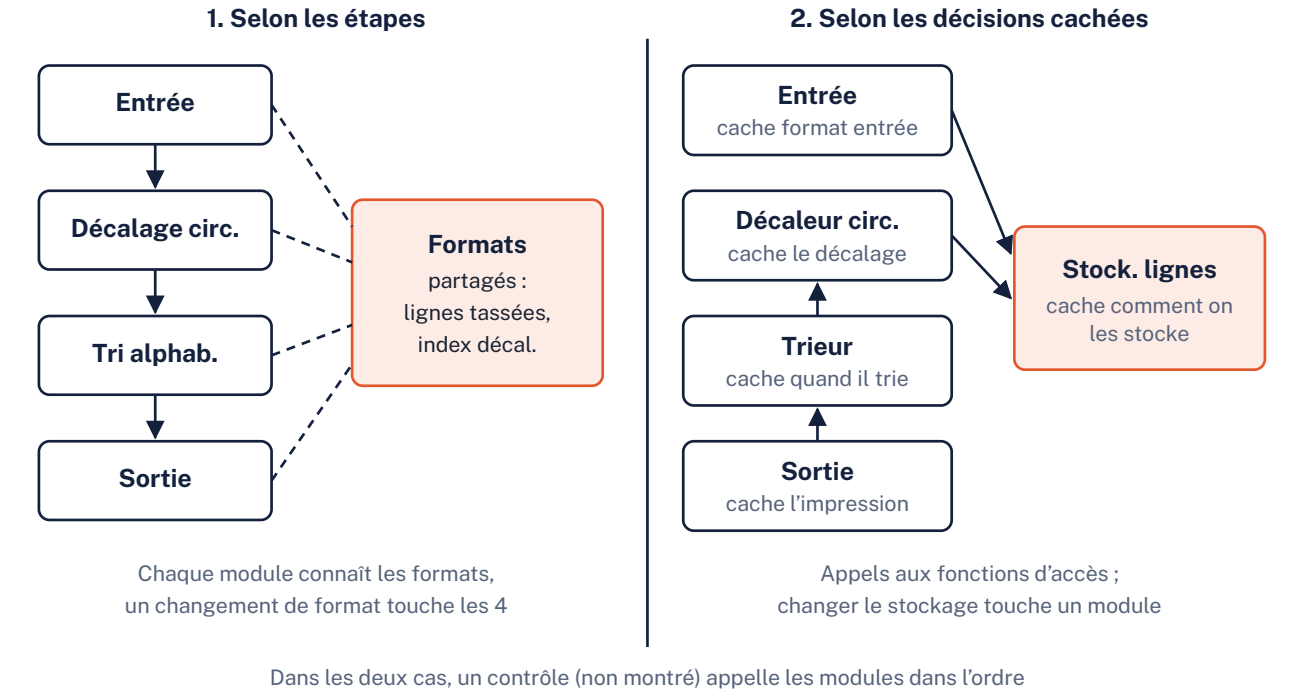
Parnas a illustré son propos avec un petit programme appelé KWIC (key word in context, mot clé dans son contexte), qui produit un index de titres. Il lit des lignes de texte, génère chaque décalage circulaire de chaque ligne (en déplaçant à plusieurs reprises le premier mot à la fin), trie tous les décalages par ordre alphabétique et les imprime. La ligne `Design of Software` produit `Design of Software`, `of Software Design` et `Software Design of`, de sorte qu'un lecteur peut trouver le titre sous n'importe lequel de ses mots.

La première décomposition qu'il décrit est celle que produiraient la plupart des programmeurs en dessinant un organigramme : un module Entrée qui lit les lignes et les stocke en mémoire dans un format compacté, un module Décalage circulaire qui construit un index indiquant où commence chaque décalage, un module Tri alphabétique qui trie cet index, un module Sortie qui imprime le résultat, et un module de contrôle principal qui les appelle dans l'ordre. C'est une conception parfaitement raisonnable. Chaque module est petit et les interfaces entre eux sont les formats de données en mémoire.

La seconde décomposition paraît plus étrange. Elle comprend un module Stockage des lignes qui offre des fonctions comme *donner le caractère de la ligne r, mot w, position c* et cache la façon dont les lignes sont réellement stockées ; un module Entrée qui lit les lignes et les confie au Stockage des lignes ; un Décaleur circulaire qui présente les décalages *comme s'ils* étaient stockés, au moyen de fonctions d'accès similaires ; un Trieur qui offre *donne-moi le i-ème décalage dans l'ordre* sans dire quand ni comment il trie ; un module Sortie ; et un contrôle principal. Les modules ne correspondent plus aux étapes du traitement.

Parnas a ensuite listé les décisions discutables et susceptibles de changer : le format d'entrée ; le fait de garder toutes les lignes en mémoire, ce qui n'est pas réaliste pour de très grandes entrées ; le fait de compacter plusieurs caractères dans chaque mot machine ; le fait de stocker un index des décalages plutôt que les décalages eux-mêmes ; et le fait de tout trier une fois plutôt que de rechercher à la demande. Dans la première décomposition, la plupart de ces changements touchent plusieurs modules, car chaque module connaît les formats de données partagés. Dans la seconde, chaque changement reste confiné au seul module qui cache la décision correspondante. La seconde conception était aussi plus facile à développer en parallèle, parce que les équipes n'avaient à s'accorder que sur des noms de fonctions et des paramètres, et non sur des structures de tables détaillées, et plus facile à comprendre un module à la fois.

Les deux décompositions de KWIC selon Parnas



Dans les deux cas, un contrôle (non montré) appelle les modules dans l'ordre

Dans la décomposition par organigramme, tous les modules partagent les formats de données ; dans la décomposition par masquage de l'information, chaque module cache une décision derrière des fonctions d'accès.

Le critère : décomposer selon les décisions susceptibles de changer

Parnas concluait qu'il est «presque toujours incorrect» de commencer à décomposer un système à partir d'un organigramme. Il proposait plutôt de partir d'une liste de décisions de conception difficiles, ou de décisions de conception susceptibles de changer, et de concevoir chaque module pour qu'il cache l'une de ces décisions aux autres. Il a appelé ce principe le masquage de l'information (information hiding) ; la section 3.3 le traite en profondeur.

L'idée paraît simple, mais elle va à l'encontre de l'instinct de la plupart d'entre nous. Un organigramme décrit le *temps* : d'abord ceci, ensuite cela. Les décisions de conception traversent le temps. La décision sur la façon de représenter une commande sert quand la commande est reçue, tarifée, payée, enregistrée et confirmée. Si les modules suivent les étapes, cette décision unique est répartie entre toutes, et la changer impose de toutes les changer. Si un module possède la décision, toutes les étapes s'adressent à ce module, et la décision peut changer en un seul endroit.

Deux précisions comptent en pratique. D'abord, le critère porte sur les changements *probables*, ce qui signifie qu'un architecte doit faire une supposition éclairée sur l'avenir : ce qui a changé dans des systèmes similaires, ce que l'entreprise prévoit, quels fournisseurs et quelles réglementations sont instables. Les quatre questions de modifiabilité de la section 2.6 (qu'est-

ce qui change, avec quelle probabilité, qui le change, à quel coût) sont exactement les données d'entrée dont il a besoin. Ensuite, Parnas distinguait une bonne décomposition d'une hiérarchie. Un système peut être proprement organisé en couches et pourtant mal décomposé, et inversement ; la section 3.4 traite séparément du sens des dépendances.

D'OÙ CELA VIENT

D'où cela vient

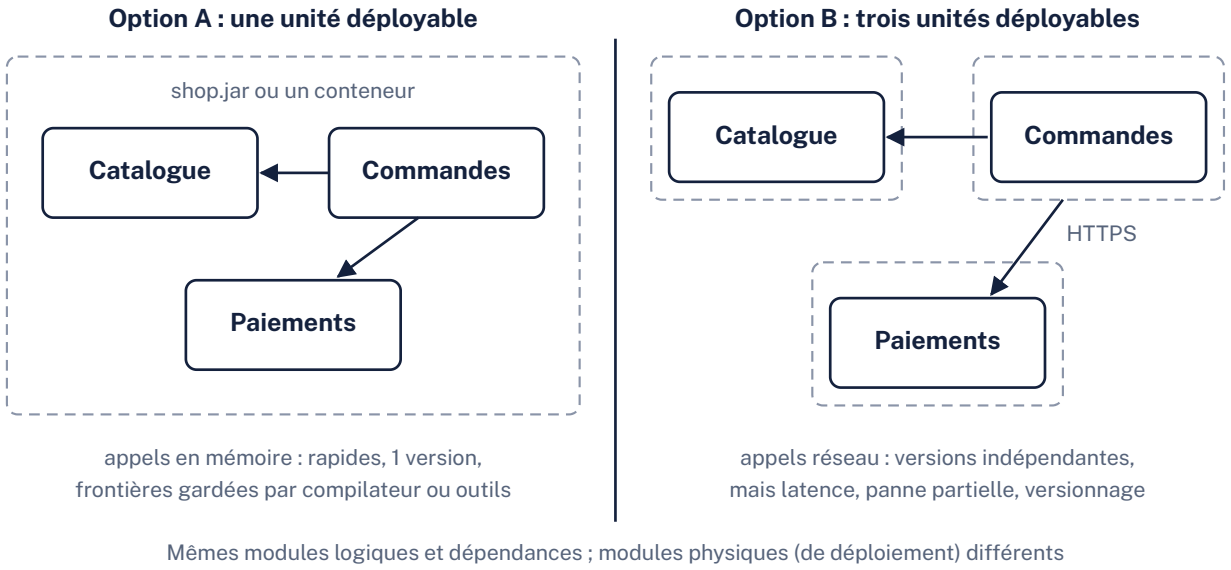
L'article de Parnas, *On the Criteria To Be Used in Decomposing Systems into Modules*, a paru dans *Communications of the ACM* en décembre 1972 (volume 15, numéro 12). À l'époque, la programmation modulaire était largement recommandée, mais il existait peu de conseils sur la façon de choisir les modules. L'article ne fait que six pages, se lit très facilement et reste l'un des plus cités du génie logiciel. Parnas y relevait même un défaut de sa propre seconde conception : l'interface du décaleur circulaire fixait l'ordre des décalages, et en révélait donc plus que nécessaire.

Modules logiques et modules physiques

Un module logique est une unité de responsabilité dans la conception : Catalogue, Commandes, Paiements. Un module physique est une unité que l'on construit, que l'on versionne ou que l'on déploie : un fichier JAR Java, un wheel Python, un paquet npm, une image de conteneur, une fonction serverless. La section 1.1 les a appelées structures de modules et structures d'allocation. Les deux sont liées, mais ce n'est pas la même chose, et les confondre est à l'origine de certaines des erreurs les plus coûteuses de l'architecture moderne.

Une même conception logique peut être projetée sur des modules physiques de façons très différentes. Tous les modules logiques peuvent vivre dans une seule unité déployable et s'appeler les uns les autres au sein du même processus : c'est un monolithe modulaire. Ou bien chacun peut être déployé comme un service à part, qui appelle les autres par le réseau : ce sont des microservices. Les frontières logiques, et les décisions que cache chaque module, peuvent être identiques dans les deux cas. Ce qui change, c'est le coût du franchissement d'une frontière (un appel de fonction prend des nanosecondes, un appel réseau des millisecondes, et il peut échouer), la possibilité de mettre en production et de faire passer à l'échelle des parties de façon indépendante, et le travail d'exploitation.

Une conception logique, deux conceptions physiques



Mêmes modules logiques et dépendances ; modules physiques (de déploiement) différents

Les modules logiques décrivent des responsabilités ; les modules physiques décrivent ce qui est construit et déployé, et une même conception logique peut être empaquetée d'une façon ou de l'autre.

Une erreur courante et coûteuse consiste à traiter la séparation physique comme si elle créait la modularité logique. Découper en services un système mal décomposé ne cache aucune décision ; cela transforme des appels internes entre modules enchevêtrés en appels réseau entre services enchevêtrés, ce que l'on appelle parfois un monolithe distribué. L'erreur inverse existe aussi : une équipe garde une seule unité déployable mais laisse ses modules internes s'imbriquer les uns dans les autres, si bien qu'elle ne pourra jamais les séparer par la suite. Le conseil pratique de nombreux praticiens est de commencer par bien définir les frontières logiques, de les faire respecter, et seulement ensuite de décider lesquelles méritent de devenir des frontières physiques.

La granularité et ses coûts

Quelle taille doit avoir un module ? La granularité est la taille des modules par rapport au système, et il n'existe pas de bonne réponse universelle, car les coûts tirent dans des directions opposées. Les modules plus grands ont moins d'interfaces à concevoir et à maintenir, mais chacun contient davantage de raisons de changer sans rapport entre elles, davantage de personnes qui le modifient en même temps, et davantage de choses à comprendre avant de faire un changement. Les modules plus petits sont chacun plus faciles à comprendre, mais le système a plus d'interfaces, plus de dépendances à gérer et plus d'endroits où une même fonctionnalité doit être assemblée morceau par morceau.

Le coût d'une interface dépend aussi du type de frontière. Entre deux classes d'un même programme, une interface supplémentaire coûte un peu de temps de lecture. Entre deux services, elle coûte un saut réseau, de la sérialisation, une politique de versionnage, de la supervision, un mode de défaillance et peut-être l'accord d'une seconde équipe pour chaque changement. C'est pourquoi la bonne granularité pour les services est généralement bien plus grossière que la bonne granularité pour les classes.

Symptômes d'une mauvaise granularité

Trop grossière	Trop fine
Des fonctionnalités sans rapport modifient le même module ; de nombreuses personnes modifient les mêmes fichiers et entrent en conflit dans la gestion de versions.	Une seule petite fonctionnalité exige des changements coordonnés dans de nombreux modules ou services.
De longs temps de build et de test pour chaque changement ; personne ne comprend le module dans son ensemble.	De nombreuses interfaces triviales de simple transmission qui n'apportent aucune abstraction ; des chaînes d'appels qui traversent plusieurs modules.
Les équipes font la queue pour mettre en production, parce que tout est livré ensemble.	À travers un réseau : une latence élevée due à des appels trop bavards, des défaillances partielles et des transactions distribuées.
Le nom du module est vague, comme `core`, `common` ou `manager`.	Chaque module est superficiel : son interface est presque aussi complexe que son implémentation (voir la section 3.3).

Faire respecter les frontières des modules dans le code

Une frontière qui n'existe que sur un schéma ne survivra pas à l'épreuve d'une échéance. Tôt ou tard, quelqu'un importera une classe interne parce que c'est plus rapide, et dès qu'un raccourci existe, d'autres suivent ; la section 1.7 a appelé cela l'érosion. Les langages et les outils de build offrent plusieurs moyens de faire de la frontière voulue le chemin de moindre résistance, ou de faire échouer le build quand on la franchit.

Python repose sur des conventions : les noms qui commencent par un tiret bas sont considérés comme internes (PEP 8), et le fichier `__init__.py` d'un paquet peut ne réexporter que ses fonctions publiques. Rien n'empêche d'importer `shop.payments._stripe_client` : les équipes

soucieuses de leurs frontières utilisent donc un vérificateur comme import-linter pour faire échouer le build quand un import interdit apparaît. Java offre la visibilité package-private (la visibilité par défaut quand aucun modificateur n'est indiqué) et, depuis Java 9 en 2017, le Java Platform Module System : un fichier `module-info.java` déclare de quels modules celui-ci a besoin (`requires`) et lesquels de ses paquets il exporte (`exports`) ; les paquets qui ne sont pas exportés ne peuvent pas être utilisés de l'extérieur, ni à la compilation ni à l'exécution. Go considère tout paquet situé sous un répertoire nommé `internal` comme importable uniquement depuis le code enraciné dans le parent de ce répertoire. C# dispose de `internal`, visible uniquement au sein d'un assembly ; Rust, de `pub(crate)`.

```
# shop/payments/__init__.py  (the public face of the package)
from shop.payments.api import charge, refund, PaymentResult

__all__ = ["charge", "refund", "PaymentResult"]

# shop/payments/_provider_client.py   internal: leading underscore
# shop/payments/_retry.py             internal
# shop/payments/api.py                the only module callers should use

# Elsewhere in the shop:
from shop.payments import charge          # fine
from shop.payments._retry import backoff  # works, but breaks the boundary
```

Un paquet Python qui publie une petite interface et garde ses modules d'implémentation internes par convention.

```
// module-info.java for the payments module
module shop.payments {
    requires shop.orders.api;        // what this module depends on
    exports shop.payments.api;       // the only package others can use
    // shop.payments.provider and shop.payments.retry are not exported,
    // so code in other modules cannot compile against them
}
```

La même frontière en Java, imposée par le compilateur et l'environnement d'exécution au moyen d'un descripteur de module.

Moyens de faire respecter une frontière

Mécanisme	Exemples	Force
Convention de nommage	Tiret bas initial en Python ; `internal` dans un nom d'espace de noms	Faible : repose sur la discipline et la revue de code
Visibilité du langage	Package-private en Java, `internal` en C#, `pub(crate)` en Rust, répertoires `internal` en Go	Forte au sein d'un même build ; vérifiée par le compilateur
Système de modules du langage	Java Platform Module System (`requires` , `exports`) ; le champ `exports` d'un `package.json` Node.js	Forte : un accès non déclaré échoue à la compilation ou au chargement
Modules des outils de build	Builds multimodules Maven ou Gradle ; règles de visibilité Bazel ; paquets séparés dans un monorepo	Forte : un module ne peut utiliser que ce qu'il déclare comme dépendance
Vérifications automatisées de l'architecture	import-linter, ArchUnit, dependency-cruiser (section 3.4)	Forte si elles sont exécutées dans le pipeline de build comme fonction d'aptitude
Frontière de processus et de réseau	Services séparés avec leurs propres stockages de données, accessibles uniquement par une API	La plus forte, et la plus coûteuse à franchir et à modifier

EXEMPLE COMMENTÉ

Exemple commenté : décomposer une boutique en ligne par organigramme et par changement probable

Le processus de commande d'une boutique en ligne est d'abord conçu comme un organigramme : Recevoir la commande, Valider le panier, Calculer le prix (promotions et TVA), Encaisser le paiement, Enregistrer la commande, Envoyer la confirmation. Les six étapes lisent et écrivent toutes un même enregistrement de commande partagé, un dictionnaire dont chaque étape connaît les clés. La conception alternative compte six modules, chacun cachant une décision : Commande (comment une commande est représentée et stockée), Tarification (règles de promotion), Taxes (règles de TVA), Passerelle de paiement (quel prestataire et comment l'appeler), Notifications (canaux et modèles de messages), et un Flux de validation qui ne fait qu'enchaîner les appels, comme le contrôle principal de Parnas. Cinq changements que l'entreprise prévoit pour l'année à venir sont comptés pour les deux conceptions.

Changement 1 : ajouter un second prestataire de paiement	Organigramme : Encaisser le paiement, plus Enregistrer la commande, qui stocke le format de référence de transaction du prestataire = 2 modules. Par changement : Passerelle de paiement = 1.
Changement 2 : les règles de TVA d'une catégorie de produits changent	Organigramme : Calculer le prix, Enregistrer la commande (colonnes de TVA) et Envoyer la confirmation (détail de la TVA) = 3. Par changement : Taxes = 1.
Changement 3 : découper l'adresse de livraison en champs structurés	Organigramme : chaque étape lit l'enregistrement de commande partagé = 6. Par changement : Commande = 1 (les autres appellent ses méthodes).
Changement 4 : ajouter des confirmations par SMS	Organigramme : Envoyer la confirmation = 1. Par changement : Notifications = 1.
Changement 5 : nouvelle fonctionnalité, un message cadeau sur les commandes	Organigramme : Recevoir la commande, Enregistrer la commande, Envoyer la confirmation = 3. Par changement : Flux de validation, Commande, Notifications = 3.
Total des modules touchés	Organigramme : 2 + 3 + 6 + 1 + 3 = 15. Par changement : 1 + 1 + 1 + 1 + 3 = 7.

Les changements portant sur des décisions existantes (prestataire, règles fiscales, représentation) restent à l'intérieur d'un seul module dans la conception fondée sur le changement ; c'est exactement ce que Parnas avait prédit.

Certains changements coûtent autant dans les deux conceptions (SMS), et une fonctionnalité réellement nouvelle (message cadeau) traverse plusieurs modules dans les deux. Le masquage de l'information protège les décisions que vous avez anticipées, pas toutes les fonctionnalités possibles.

Ce décompte est un moyen rapide et reproductible de comparer des décompositions candidates lors d'une revue de conception : listez d'abord les changements probables, puis comptez les modules que chacun touche.

Erreurs courantes

Décomposer uniquement par couche technique. Des dossiers appelés `controllers`, `services` et `repositories` sont des modules d'une certaine sorte, mais un changement dans le fonctionnement de la TVA touchera les trois. Les couches sont utiles (section 3.4), mais une décomposition qui n'a que des couches est un organigramme déguisé. La plupart des systèmes matures combinent un découpage par capacité métier et une organisation en couches à l'intérieur de chaque partie.

Tout partager. Un module appelé `common` ou `utils` que tous les autres modules importent devient l'endroit où s'accumulent des décisions sans rapport entre elles, et tout changement qui le touche se propage partout. **Des frontières sans contrôle.** Les schémas montrent des modules propres alors que le code contient des centaines d'imports croisés. **Des modules définis par l'organigramme d'une réorganisation passée** plutôt que par les décisions que le système doit cacher ; la section 3.7 explique pourquoi les organisations et les frontières des modules s'influencent mutuellement. **La distribution prématurée.** Transformer chaque module logique en service avant que les frontières aient prouvé leur stabilité rend coûteuses les corrections ultérieures de frontières.

Liste de contrôle : examiner une proposition de décomposition

- [ ] **Nommez la décision que cache chaque module**
  Si vous ne pouvez pas dire en une phrase ce qu'un module garde secret, il s'agit probablement d'une étape de traitement ou d'un fourre-tout.
- [ ] **Listez les changements probables**
  Notez cinq à dix changements que l'activité ou la technologie risquent d'apporter, et comptez les modules que chacun toucherait.
- [ ] **Vérifiez la taille de l'interface**
  L'interface doit être bien plus petite que l'implémentation. De grandes interfaces avec peu de comportement derrière elles suggèrent un mauvais découpage.
- [ ] **Séparez le logique du physique**
  Mettez-vous d'abord d'accord sur les modules logiques ; décidez séparément lesquels méritent leur propre unité déployable, et pourquoi.
- [ ] **Décidez comment la frontière sera imposée**
  Choisissez un mécanisme de visibilité ou une vérification automatisée, et ajoutez-le au build avant que le premier raccourci n'apparaisse.
- [ ] **Alignez les frontières sur la responsabilité des équipes**
  Chaque module devrait avoir une seule équipe propriétaire ; des modules partagés par plusieurs équipes pour l'essentiel de leur travail sont un signal d'alerte.

Comment les professionnels appliquent la modularité

Les architectes expérimentés considèrent la structure des modules comme l'artefact de conception le plus important pour la maintenabilité, et ils y reviennent régulièrement. Au début de la vie d'un produit, quand le domaine est mal compris, ils préfèrent des modules moins nombreux et plus grands, avec de solides frontières internes, parce que fusionner et scinder des modules au sein d'une même base de code coûte peu. À mesure que le domaine se stabilise et que les changements probables deviennent clairs, ils affinent les frontières et ce n'est qu'alors qu'ils se demandent lesquelles devraient être déployées séparément. Les techniques de la conception pilotée par le domaine, comme les contextes délimités (bounded contexts), aident à trouver des frontières qui suivent le langage du métier.

Ils mesurent aussi. Compter les modules touchés par les changements récents, ce que l'historique de la gestion de versions rend facile, montre si les frontières fonctionnent ; la section 3.8 présente le couplage de changement et les hotspots (points chauds). La section

suivante, la section 3.2, donne le vocabulaire pour décrire à quoi ressemblent de bonnes et de mauvaises frontières : le couplage entre modules et la cohésion à l'intérieur de chacun.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi Parnas appelait-il un module une « attribution de responsabilité » plutôt qu'un sous-programme ?**

Parce que ce qui compte, ce sont les décisions de conception dont un module est responsable et qu'il cache, et non la façon dont son code est empaqueté ; une responsabilité peut s'étendre sur plusieurs routines, et les routines n'ont pas à correspondre aux étapes de traitement.

▼ **Dans l'exemple KWIC, pourquoi le changement de la façon de stocker les lignes touchait-il plusieurs modules dans la première décomposition ?**

Tous les modules partageaient directement les formats de données en mémoire : la décision de stockage était donc connue de chaque module au lieu d'être cachée dans un seul.

▼ **Quelle est la différence entre un module logique et un module physique ?**

Un module logique est une unité de responsabilité dans la conception ; un module physique est une unité que l'on construit ou que l'on déploie, comme un JAR, un paquet, un conteneur ou un service. Une même conception logique peut être empaquetée physiquement de nombreuses façons.

▼ **Citez deux signes indiquant que des services sont trop fins.**

De petites fonctionnalités exigent des changements coordonnés dans plusieurs services, et les requêtes passent par des chaînes bavardes d'appels réseau, avec une latence élevée et des défaillances partielles.

▼ **À vous : Le paiement d'un billet est découpé en étapes, Sélection, Blocage, Prix, Paiement et Émission, qui partagent toutes un même enregistrement de réservation dont chaque étape connaît les champs. Des promotions et une nouvelle façon de stocker les blocages arrivent. Critiquez ce découpage, puis proposez-en un meilleur.**

C'est une décomposition en organigramme : chaque décision de conception est répartie sur toutes les étapes, si bien que changer l'enregistrement de réservation ou le stockage des blocages touche les cinq. Comme le préconise Parnas, faites en sorte que chaque module cache une décision susceptible de changer : Réservation cache la forme de l'enregistrement, Blocages cache la façon dont les blocages sont stockés, Tarification cache les règles de promotion et Paiements cache le prestataire. Chacun offre une interface étroite : les promotions ne changent donc qu'un module. Faites respecter les frontières, sinon elles s'éroderont.

SOURCES

- D. L. Parnas, On the Criteria To Be Used in Decomposing Systems into Modules (Communications of the ACM, 1972): https://doi.org/10.1145/361598.361623
- JEP 261: Module System (OpenJDK): https://openjdk.org/jeps/261
- Go 1.4 release notes: internal packages: https://go.dev/doc/go1.4
- Import Linter documentation: https://import-linter.readthedocs.io/en/stable/
- Dijkstra: On the role of scientific thought (EWD 447): https://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html

Couplage et cohésion

AVANT DE LIRE

Cette section s'appuie sur :

- 0.3 Organiser le code
- 2.6 Modifiabilité et déployabilité
- 3.1 Modularité

Termes déjà vus qu'elle utilise : Module, Interface, Appel synchrone, Messagerie asynchrone, Refactorisation (refactoring)

L'IDÉE EN BREF

Pensez à deux voisins qui partagent un jardin. S'ils se prêtent seulement un outil de temps en temps, quand l'autre le demande, chacun peut réaménager son propre jardin sans histoires : ils sont faiblement couplés. S'ils partagent un abri, un tuyau d'arrosage et une clé cachée sous un pot de fleurs, chaque changement exige une conversation : ils sont fortement couplés. Pensez maintenant à un tiroir de cuisine. S'il ne contient que des couverts, vous savez où chercher : il est cohésif. S'il contient des couverts, des piles, de la ficelle et de vieux tickets de caisse, il ne l'est pas. Un bon logiciel a des parties cohésives, faiblement couplées les unes aux autres.

CE QU'IL FAUT SAVOIR

- Le couplage est le degré de dépendance d'un module envers un autre ; la cohésion est le degré auquel les éléments d'un module vont ensemble. Visez un couplage faible et une cohésion forte, qui vont généralement de pair.
- Les échelles de la conception structurée classent le couplage du couplage de contenu, commun, externe, de contrôle et par estampille jusqu'au couplage de données (du plus fort au plus faible), et la cohésion de la cohésion accidentelle, logique, temporelle, procédurale, communicationnelle et séquentielle jusqu'à la cohésion fonctionnelle (de la plus faible à la plus forte).
- Le couplage afférent (Ca) compte les dépendances entrantes et le couplage efférent (Ce) les dépendances sortantes ; un module dont le Ca est élevé est risqué à modifier, et un module dont le Ce est élevé est exposé aux changements des autres.

- Les systèmes distribués ajoutent le couplage temporel (les deux côtés doivent être disponibles en même temps), le couplage de déploiement (ils doivent être mis en production ensemble) et le couplage sémantique (ils doivent partager une signification cachée), qu'une API propre ne suffit pas à supprimer.
- Parmi les tactiques qui réduisent le couplage figurent les interfaces étroites, les intermédiaires comme les adaptateurs et les courtiers, les événements à la place des commandes, et une propriété stricte des données, chacune avec son propre coût.
- Le couplage ne peut pas être éliminé, seulement choisi : une partie du couplage est la raison d'être du système ; le but est donc de le rendre explicite, faible et local là où le changement est probable.

Deux mesures de la structure

Le couplage est le degré de dépendance d'un module envers un autre, tel qu'un changement de l'un peut imposer un changement de l'autre. La cohésion est le degré auquel les éléments d'un module vont ensemble parce qu'ils servent un même objectif. Ces deux idées ont été introduites par Larry Constantine à la fin des années 1960, dans le cadre du mouvement de la conception structurée, et publiées en 1974 dans l'article *Structured Design* de Wayne Stevens, Glenford Myers et Constantine paru dans l'*IBM Systems Journal*, puis plus longuement dans le livre d'Edward Yourdon et Constantine, *Structured Design* (1979). La section 0.3 les a présentées de manière informelle ; cette section les précise.

Les deux sont liées. Si les choses qui changent ensemble vivent dans le même module (cohésion forte), alors moins de changements traversent les frontières des modules, et les modules ont besoin d'en savoir moins les uns sur les autres (couplage faible). Yourdon et Constantine allaient jusqu'à dire qu'en pratique, la cohésion était la plus importante des deux. Les tactiques de modifiabilité de la section 2.6 (augmenter la cohésion, réduire le couplage) reposent exactement sur ce couple.

Une précision d'emblée : le couplage n'est pas un défaut. Un système dont les modules ne dépendraient jamais les uns des autres ne ferait rien d'utile. Martin Fowler, qui écrivait sur le sujet en 2001, avertissait que chercher à supprimer entièrement le couplage ne fait que tout repousser dans un seul module géant, où le couplage est caché plutôt que disparu. Le but est de choisir le couplage délibérément : peu de dépendances, passant par des interfaces étroites et stables, orientées dans des directions sensées, et aussi faibles que la tâche le permet.

Les échelles classiques



Les échelles de la conception structurée : le couplage, du couplage de contenu (le plus fort) au couplage de données (le plus faible), et la cohésion, de la cohésion accidentelle (la plus faible) à la cohésion fonctionnelle (la plus forte).

Les niveaux de couplage

La conception structurée a classé les manières dont deux modules peuvent être reliés, de la plus nocive à la plus bénigne. La forme à six niveaux ci-dessous remonte généralement aux livres de Glenford Myers sur la conception composite du milieu des années 1970, et a ensuite été popularisée par Meilir Page-Jones ; les noms et les limites varient légèrement selon les sources, mais c'est la forme que l'on trouve dans la plupart des manuels. Ces niveaux ont été définis pour les procédures d'un même programme, mais chacun a un équivalent direct entre composants et entre services, et c'est là que les architectes les rencontrent aujourd'hui.

Niveaux de couplage, du plus fort au plus faible

Niveau	Ce que cela signifie	Dans le code	Entre services
Contenu	Un module pénètre dans les éléments internes d'un autre : il modifie ses données privées ou saute dans son code.	Modifier `cart._items` depuis l'extérieur ; remplacer à chaud (monkey-patching) la fonction d'un autre module.	Lire ou écrire directement dans les tables de base de données d'un autre service.
Commun	Plusieurs modules lisent et écrivent les mêmes données globales.	Un dictionnaire global `config` que n'importe quelle fonction peut modifier.	Plusieurs services partageant un même schéma de base de données ou un même espace de clés de cache.
Externe	Des modules partagent un format, un protocole ou un périphérique imposé de l'extérieur.	Deux modules analysent tous deux le même format de fichier d'un fournisseur.	Plusieurs services dépendant de la structure de fichier d'un partenaire ou d'une version précise d'un protocole de messagerie.
Contrôle	Un module passe un indicateur qui dit à l'autre quoi faire, si bien que l'appelant connaît la logique interne de l'appelé.	`process(order, mode="refund")`	Une API avec un champ `action` qui sélectionne des comportements complètement différents.
Stamp	Un module reçoit toute une structure de données mais n'en utilise qu'une partie.	Passer l'objet `Customer` complet à une fonction qui n'a besoin que de l'adresse e-mail.	Envoyer l'enregistrement client complet dans chaque événement alors que les consommateurs ont besoin de deux champs.
Données	Les modules ne communiquent que par des paramètres de données simples, tous nécessaires.	`send_receipt(email, order_id, total)`	Une API ou un événement étroit et documenté, avec seulement les champs dont les consommateurs ont besoin.

Deux mises en garde. D'abord, l'échelle est un guide du risque, pas une loi. Le couplage par estampille (stamp coupling) est souvent un compromis raisonnable : passer un objet entier est plus simple que d'en extraire six champs, et le seul coût est que le destinataire peut désormais casser quand on modifie des champs qu'il n'utilise même pas. Ensuite, l'échelle ne dit rien du

nombre de dépendances ni de la fréquence à laquelle l'autre côté change. Un unique appel couplé par contrôle vers un module qui ne change jamais peut être inoffensif ; un couplage de données vers un module qui change chaque semaine peut faire mal. Plus loin dans ce chapitre, la connascence offre un vocabulaire plus fin, et *Balancing Coupling in Software Design* (2024), de Vlad Khononov, soutient que la force du couplage doit toujours être jugée en même temps que la distance entre les modules et leur volatilité.

```
# Control coupling: the caller chooses the callee's internal branch
def handle_payment(order, mode):
    if mode == "charge":
        ...  # call the provider, store the transaction
    elif mode == "refund":
        ...  # look up the transaction, reverse it
    elif mode == "dry_run":
        ...  # validate only

handle_payment(order, "refund")

# Data coupling: separate operations, each taking only what it needs
def charge(order_id: str, amount_pence: int) -> str: ...
def refund(transaction_id: str, amount_pence: int) -> None: ...

refund("tx_81f2", 1999)
```

Couplage de contrôle (en haut) contre couplage de données (en bas) : l'indicateur oblige l'appelant à savoir comment l'appelé est organisé à l'intérieur.

SI VOUS DÉBUTEZ

Pas à pas : un reçu, quatre sortes de couplage

L'idée la plus difficile de cette section est de comprendre pourquoi certaines sortes de couplage sont pires que d'autres. En gros, plus un module s'appuie sur des choses que d'autres modules contrôlent ou peuvent modifier, plus leurs changements risquent de le casser. Suivez une petite tâche, l'envoi d'un reçu, écrite de quatre façons (quatre des six niveaux du tableau, du plus fort au plus faible).

- Le code du reçu a besoin de l'adresse e-mail d'un client, du numéro de commande et du total. Dans la première version, il pénètre dans les données privées du module client et lit directement le champ e-mail : c'est un couplage de contenu. Renommez ce champ et le reçu casse, alors que personne n'avait promis qu'il resterait inchangé.
- Dans la deuxième version, le processus de commande place l'adresse e-mail dans un dictionnaire global que n'importe quelle fonction peut lire ou modifier, et le code du reçu la lit à cet endroit : c'est un couplage commun. Désormais, tout code qui écrit dans ce dictionnaire peut envoyer le reçu à la mauvaise adresse, et rien dans le code du reçu n'indique lequel.
- Dans la troisième version, le code du reçu reçoit l'objet Customer complet et n'utilise que l'adresse e-mail : c'est un couplage par estampille. Il ne pénètre plus à l'intérieur, mais il peut encore casser quand on modifie des champs qu'il n'utilise pas.
- Dans la quatrième version, il ne reçoit que les trois valeurs simples dont il a besoin : c'est un couplage de données, le niveau le plus faible. Il ne sait rien de la façon dont les clients ou les commandes sont stockés, si bien que le module client peut se réorganiser librement.
- **La leçon :** chaque pas vers l'extrémité « données » de l'échelle signifie que le code du reçu s'appuie sur moins de choses contrôlées par d'autres modules, si bien que moins de leurs changements peuvent le casser. L'échelle est un guide du risque, pas une loi : passer l'objet entier est souvent un compromis raisonnable.

Les niveaux de cohésion

La cohésion demande pourquoi les éléments d'un module sont ensemble. Yourdon et Constantine ont décrit sept sortes d'association, là encore de la pire à la meilleure. Les niveaux inférieurs regroupent des choses pour des raisons sans rapport avec le problème (elles ont été écrites au même moment, ou elles s'exécutent au même moment) ; les niveaux supérieurs regroupent des choses parce qu'elles travaillent sur les mêmes données ou vers un même objectif unique. Les synthèses ultérieures des manuels s'accordent sur le classement général :

les cohésions accidentelle et logique sont médiocres, les cohésions communicationnelle et séquentielle sont bonnes et la cohésion fonctionnelle est la meilleure, même si les écarts entre niveaux ne sont pas réguliers.

Niveaux de cohésion, du plus faible au plus fort

Niveau	Les éléments sont regroupés parce que…	Exemple
Accidentelle	ils n'ont absolument aucune relation significative.	Un module `utils` contenant le formatage des dates, un décorateur de nouvelle tentative et un taux de TVA.
Logique	ils font le même genre de chose, choisie par un indicateur.	Un module `exports` doté d'une fonction qui écrit du CSV, du PDF ou du XML selon un paramètre.
Temporelle	ils s'exécutent au même moment.	Une routine `startup()` qui ouvre la base de données, préchauffe les caches et envoie un ping de supervision.
Procédurale	ils se suivent dans une séquence de contrôle, mais sur des données différentes.	Une fonction qui valide un formulaire, puis journalise l'adresse IP de l'utilisateur, puis rafraîchit un menu.
Communicationnelle	ils opèrent sur les mêmes données.	Un module qui calcule plusieurs rapports à partir du même historique de commandes.
Séquentielle	la sortie de l'un est l'entrée du suivant.	Analyser un relevé bancaire, puis classer ses lignes par catégorie, puis totaliser chaque catégorie.
Fonctionn.	ensemble, ils accomplissent une tâche bien définie et rien d'autre.	Un module `Tax` qui calcule la TVA d'un panier, et uniquement cela.

À l'échelle de l'architecture, la cohésion revient à se demander si un composant ou un service a une seule raison d'exister. Un service appelé `CustomerService` qui gère la connexion, les points de fidélité et les préférences marketing a au mieux une cohésion logique : trois équipes le modifieront pour trois raisons sans rapport. Une bonne heuristique consiste à décrire l'objectif d'un module en une phrase sans employer *et* ; la section 3.5 développe cette idée sous la forme du principe de responsabilité unique, défini en fonction des personnes qui demandent les changements.

Couplage afférent et efférent

Les niveaux ci-dessus décrivent la *nature* d'une dépendance. Les architectes ont aussi besoin de les compter. Le couplage afférent (Ca) est le nombre de modules extérieurs à un composant qui dépendent de lui : ses flèches entrantes. Le couplage efférent (Ce) est le nombre de modules extérieurs dont il dépend : ses flèches sortantes. Ces termes viennent de l'anatomie, où les nerfs afférents transportent les signaux vers l'intérieur et les nerfs efférents vers l'extérieur. Robert C. Martin les a définis pour les paquets en 1994 en comptant des classes ; les outils actuels comptent des classes, des modules ou des paquets : vérifiez donc toujours ce que compte un outil avant de comparer des chiffres.

Les deux nombres racontent des histoires différentes. Un composant dont le Ca est élevé, comme une bibliothèque partagée `money` utilisée par quarante modules, est *responsable* : tout changement qui le touche peut en casser beaucoup d'autres, il doit donc changer avec prudence et rarement. Un composant dont le Ce est élevé, comme un flux de validation de commande qui appelle la tarification, le stock, le paiement, la livraison et les notifications, est *exposé* : il peut devoir changer chaque fois que l'un d'eux change. Ni l'un ni l'autre n'est un problème en soi ; le problème apparaît quand un composant est à la fois très sollicité par d'autres et lui-même dépendant de nombreux éléments volatils. La section 3.4 transforme Ca et Ce en la métrique d'instabilité de Martin et en règles fondées sur elle.

EXEMPLE COMMENTÉ

Exemple commenté : compter Ca et Ce dans une application de livraison de repas

Sept modules (Validation, Carte, Tarification, Paiements, Notifications, Suivi des livreurs et Administration) ont les dépendances suivantes (« A dépend de B » s'écrit « A vers B ») : Validation vers Carte, Validation vers Tarification, Validation vers Paiements, Validation vers Notifications ; Suivi des livreurs vers Notifications ; Tarification vers Carte ; Administration vers Carte, Administration vers Tarification.

Carte	Ca = 3 (Validation, Tarification, Administration) ; Ce = 0
Tarification	Ca = 2 (Validation, Administration) ; Ce = 1 (Carte)
Notifications	Ca = 2 (Validation, Suivi des livreurs) ; Ce = 0
Validation	Ca = 0 ; Ce = 4 (Carte, Tarification, Paiements, Notifications)
Lecture	Carte est le module dont on dépend le plus : son interface demande donc le plus de soin et les changements les plus lents. Validation est le module qui dépend le plus des autres : il devrait donc être le plus facile à modifier, et celui qui absorbe les changements venus des autres.

Le décompte prend quelques minutes avec un outil d'analyse des dépendances et transforme une impression vague (« Carte est un module central ») en un chiffre que vous pouvez suivre dans le temps.

Le couplage dans les systèmes distribués

Quand les modules deviennent des services qui communiquent par un réseau, les niveaux classiques s'appliquent toujours, mais trois autres sortes de couplage prennent de l'importance. Aucune n'est visible dans la signature d'une fonction, ce qui explique pourquoi les équipes qui découpent un système en services sont souvent surprises de voir à quel point les services restent couplés.

Le couplage temporel signifie qu'une partie ne fonctionne que si une autre est disponible et réactive au même moment. Un appel HTTP synchrone de Validation vers Paiements couple leur disponibilité : si Paiements est en panne, Validation l'est aussi, et la disponibilité combinée est le produit des deux (section 2.5). Des messages asynchrones passant par une file suppriment ce couplage, au prix de résultats obtenus à terme plutôt qu'immédiatement. (En conception au niveau du code, le même terme désigne parfois le fait que des appels doivent se faire dans un ordre fixe, comme `open()` avant `read()` ; le contexte rend en général le sens évident.)

Le couplage de déploiement signifie que deux parties doivent être mises en production ensemble, ou dans un ordre fixe, parce qu'un changement dans l'une casse l'autre. Si chaque mise en production de Commandes exige une mise en production simultanée de Facturation, les deux forment en pratique une seule unité déployable, quoi qu'en dise le schéma. Les changements rétrocompatibles, les API versionnées et le patron expansion-contraction (expand and contract) sont les remèdes habituels.

Le couplage sémantique signifie que deux parties doivent s'accorder sur une signification qu'aucune interface n'exprime : qu'une commande avec `status = 4` est expédiée, que les prix sont en pence et TVA comprise, qu'un « client actif » a commandé au cours des 90 derniers jours. De tels accords se rompent en silence quand l'un des côtés change son interprétation. Le remède consiste à rendre la signification explicite dans le contrat (états nommés, unités dans les noms de champs, schémas publiés) ou à garder la règle en un seul endroit que les autres interrogent.

Tactiques pour réduire le couplage

Des interfaces étroites. N'exposez que le minimum d'opérations et de champs dont les clients ont besoin. Chaque champ supplémentaire dans une réponse d'API est une chose dont un client peut finir par dépendre (voir la section 3.3 sur la loi de Hyrum). **Des intermédiaires.** Placez quelque chose entre les deux parties pour qu'aucune ne connaisse les détails de l'autre : un adaptateur ou une couche anticorruption qui traduit le modèle d'un fournisseur, une API gateway (passerelle d'API), ou un courtier de messages (message broker) qui découple l'émetteur du destinataire dans le temps et dans l'identité. **Des événements plutôt que des commandes.** Une commande (*Facturation, crée une facture*) oblige l'émetteur à connaître le destinataire et sa tâche ; un événement (*une commande a été passée*) permet à l'émetteur d'énoncer un fait et à un nombre quelconque de destinataires d'y réagir, si bien que de nouveaux consommateurs peuvent être ajoutés sans modifier le producteur. **La propriété des données.** Chaque donnée a un seul propriétaire qui l'écrit ; les autres modules interrogent le propriétaire par son interface ou gardent leur propre copie alimentée par des événements, sans jamais lire les tables du propriétaire (section 7.1).

Chaque tactique a un prix. Les intermédiaires ajoutent un saut et un élément de plus à exploiter. Les événements renoncent à la cohérence immédiate et rendent le déroulement d'un processus métier plus difficile à voir en un seul endroit. Les copies de données peuvent être périmées. La propriété des données oblige à construire et à versionner des API qu'une table partagée vous donnait gratuitement. Le compromis vaut la peine là où le changement est fréquent ou les équipes séparées, et souvent pas entre deux classes maintenues par une seule personne.

La loi de Déméter : ne parlez qu'à vos amis immédiats

La **loi de Déméter** est issue du projet Demeter de la Northeastern University, où Ian Holland a identifié la règle en 1987 ; Karl Lieberherr, Holland et A. Riel l'ont publiée à la conférence OOPSLA en 1988. Sous sa forme générale, chaque unité ne doit avoir qu'une connaissance limitée des autres unités, celles qui lui sont étroitement liées : « ne parlez qu'à vos amis immédiats ». En termes orientés objet, une méthode peut envoyer des messages à son propre objet, à ses arguments, aux objets qu'elle crée et aux parties immédiates de l'objet, mais pas à des inconnus atteints par leur intermédiaire.

Prenons une routine d'affectation des livreurs de la plateforme de livraison de repas qui contient `order.getCustomer().getAddress().getPostcode()`. Elle a besoin d'une seule chose, un code postal, et pourtant elle sait désormais qu'une commande a un client, qu'un client a une adresse et qu'une adresse a un code postal. Chacun de ces faits est une dépendance. Si les clients peuvent enregistrer plusieurs adresses, du code qui n'a rien à voir avec les clients casse. Écrit `order.deliveryPostcode()`, le même appel dépend d'une seule opération, et la structure qui se trouve derrière peut changer librement. Un principe voisin est « tell, don't ask » (dites, ne demandez pas) : au lieu d'extraire des données d'un objet pour agir sur elles, demandez à l'objet de faire le travail. Dans les termes de la section 3.6, la longue chaîne répand la connaissance de la structure loin de sa place.

L'idée vaut aussi à plus grande échelle. Un client qui suit une chaîne d'appels (récupérer la commande, puis son restaurant, puis le compte de versement de ce restaurant) dépend de la façon dont trois services sont reliés. Lui donner une seule opération pour ce dont il a besoin, éventuellement via un backend for frontend, supprime cette connaissance, comme le recommandent les interfaces étroites vues plus haut.

Le nom en promet trop. Martin Fowler a dit qu'il préférerait qu'on l'appelle la *Suggestion* de Déméter, et il met en garde contre l'idée de faire de « tell, don't ask » une raison de supprimer toute méthode de requête. Compter les points est un mauvais test : une chaîne sur des données simples, ou un builder fluide qui renvoie sans cesse le même objet, ne révèle rien sur des collaborateurs cachés. Et appliquée aveuglément, la règle remplit les classes de fines méthodes de transfert, ce qui élargit leurs interfaces. Utilisez-la comme une invitation à vous demander pourquoi ce code a besoin d'en savoir autant.

La composition plutôt que l'héritage

En 1994, Gamma, Helm, Johnson et Vlissides en ont fait le deuxième principe de *Design Patterns* : « Privilégiez la composition d'objets plutôt que l'héritage de classes. » Leur raisonnement portait sur la visibilité. L'héritage est une réutilisation en *boîte blanche* : la sous-classe voit les détails de son parent et en dépend. La composition est une réutilisation en *boîte*

noire : un objet n'en utilise un autre qu'au travers de son interface. Dans les termes de cette section, l'héritage est l'un des couplages les plus forts qu'offre un langage, et il est figé à l'écriture du code, si bien qu'il ne peut pas être changé pendant l'exécution du programme. Un changement dans un parent peut casser des sous-classes auxquelles personne n'a touché, un problème généralement appelé **classe de base fragile**.

La plateforme de livraison de repas montre comment cela tourne mal. Une équipe modélise `ExpressOrder` et `ScheduledOrder` comme des sous-classes d'`Order`. Puis arrive la commande groupée, et quelqu'un a besoin d'une commande groupée programmée livrée en express. L'héritage simple oblige à choisir un parent, et ajouter des classes pour chaque combinaison fait vite grossir le code. Avec la composition, une `Order` *a* une `FulfilmentPolicy` et une `PricingRule`. Chacune est une petite interface avec plusieurs implémentations, les combinaisons n'exigent aucune nouvelle classe, et une politique peut être remplacée à l'exécution ou par une doublure dans un test. C'est la forme de l'interface de paiement de la section 3.3.

Le principe s'est diffusé dans la conception des langages. Go n'a aucune hiérarchie de types : les types satisfont automatiquement les interfaces, et l'embedding offre quelque chose d'analogue, mais non identique, au sous-classement. À l'échelle architecturale, la même préférence se manifeste dans les plug-ins derrière une interface stable (section 5.5), plutôt que dans un framework que chaque équipe doit étendre. Une bibliothèque de base partagée dont chaque service doit hériter peut recréer la classe de base fragile à l'échelle de toute une organisation.

Le mot est *privilégier*, pas *toujours*. L'héritage est raisonnable pour une véritable relation de sous-type qui respecte la substitution de Liskov (section 3.5), surtout à l'intérieur d'un module dont les propriétaires contrôlent les deux côtés. La composition a ses propres coûts : du code de transfert, davantage de petits objets et un comportement plus difficile à suivre en lisant une seule classe. Demandez-vous si le nouveau type *est* vraiment l'ancien à tous égards, ou s'il veut simplement réutiliser une partie de son code. Dans le second cas, composez.

Exemple commenté : refactoriser un couple commandes-facturation fortement couplé

Le module Commandes d'une boutique en ligne appelle directement Facturation chaque fois qu'une commande est passée. Facturation lit ensuite la table des commandes pour en trouver les détails, interprète les codes numériques de la colonne de statut et recalcule la TVA à partir des prix des lignes. Le code ci-dessous en est une version simplifiée ; il montre trois des problèmes à la fois.

```
# orders.py
def place_order(basket, customer_id):
    order_id = db.insert("orders", customer_id=customer_id, status=1,
                         lines=basket.lines)
    billing.process(order_id, mode="invoice")   # Orders waits for Billing
    return order_id

# billing.py
def process(order_id, mode):
    row = db.query_one("SELECT * FROM orders WHERE id = %s", order_id)
    if row["status"] != 1:              # 1 means 'placed', by convention
        return
    net = sum(l["price"] * l["qty"] for l in row["lines"])
    vat = round(net * 0.2)               # VAT rule duplicated here
    if mode == "invoice":
        db.insert("invoices", order_id=order_id, net=net, vat=vat)
    elif mode == "credit_note":
        db.insert("credit_notes", order_id=order_id, net=-net, vat=-vat)
```

Avant : couplage de contrôle (l'indicateur de mode), couplage commun (les deux modules utilisent la table des commandes) et couplage sémantique (codes de statut et règles de TVA connus des deux).

Toute modification de la structure de la table des commandes, des codes de statut ou des règles de TVA doit désormais être faite dans les deux modules à la fois. Commandes ne peut pas accepter une commande si Facturation est lent ou en panne. Et les deux comportements de Facturation sont choisis par un indicateur que Commandes doit connaître. La refactorisation se déroule en quatre étapes : Commandes cesse d'appeler Facturation et publie à la place un événement `OrderPlaced` qui transporte exactement les faits dont Facturation a besoin, y compris les totaux que Commandes a déjà calculés ; Facturation s'y abonne et enregistre les factures dans son propre stockage ; les deux comportements deviennent deux opérations ; et la table des commandes devient privée à Commandes.

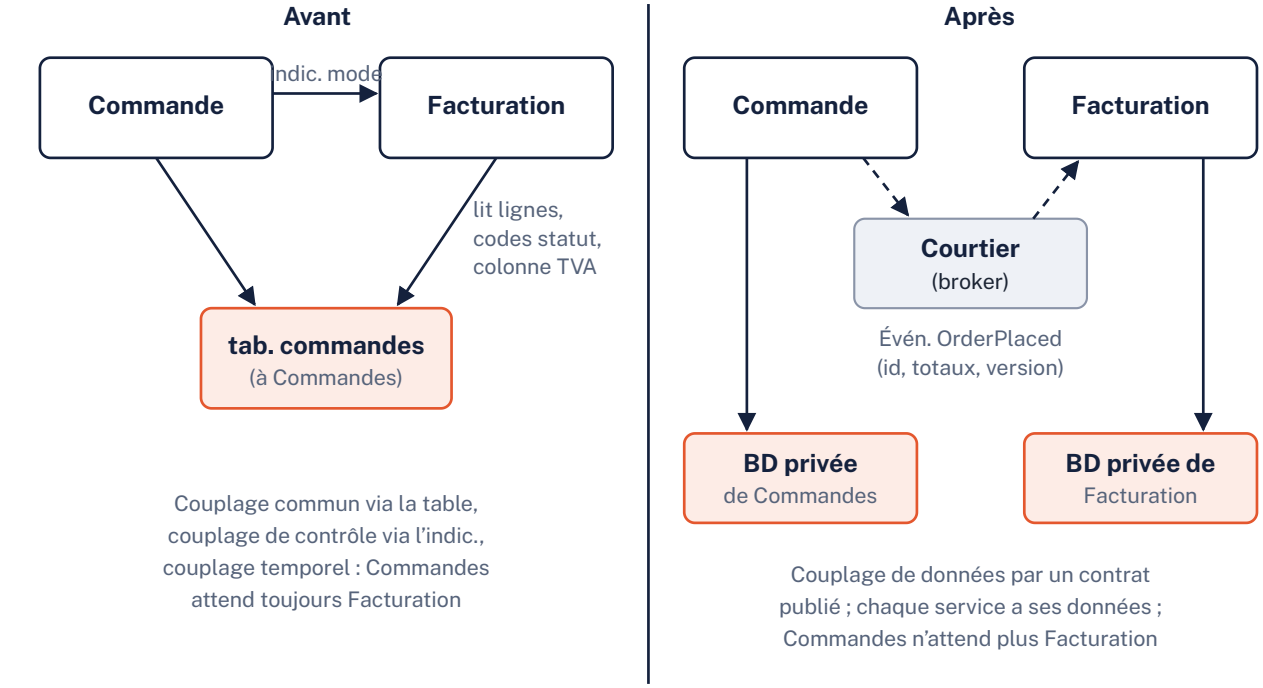
```
# orders.py
def place_order(basket, customer_id):
    order = Order.create(customer_id, basket)      # computes net and VAT once
    repository.save(order)
    events.publish("OrderPlaced", {
        "schema_version": 1, "order_id": order.id,
        "customer_id": customer_id,
        "net_pence": order.net_pence, "vat_pence": order.vat_pence})
    return order.id

# billing.py
@events.subscribe("OrderPlaced")
def on_order_placed(event):
    create_invoice(event["order_id"], event["net_pence"], event["vat_pence"])

def create_invoice(order_id, net_pence, vat_pence): ...
def create_credit_note(order_id, net_pence, vat_pence): ...
```

Après : Commandes énonce un fait par un événement versionné ; Facturation possède ses propres données et ne connaît plus les tables, les codes ni les règles fiscales de Commandes.

Le couple commandes-facturation avant et après



La refactorisation remplace un appel direct avec indicateur et une table partagée par un événement publié et des stockages de données séparés.

EXEMPLE COMMENTÉ

Ce qui a changé, mesuré

Comparaison des deux versions à l'aide du vocabulaire du couplage présenté dans cette section.

Sorte de couplage	Avant : quasi-couplage de contenu et couplage commun (table partagée), de contrôle (indicateur de mode), sémantique (statut 1, règle de TVA). Après : couplage de données par un événement versionné à cinq champs.
Couplage efférent de Commandes	Avant : Ce inclut Facturation. Après : Commandes ne dépend que du contrat de l'événement et du courtier ; il ne sait pas que Facturation existe.
Couplage temporel	Avant : une commande échoue si Facturation est en panne. Après : l'événement attend dans le courtier jusqu'à ce que Facturation soit rétabli.
Couplage de déploiement	Avant : une modification de la table des commandes exige une mise en production conjointe. Après : Commandes peut modifier sa table librement ; seules les modifications de l'événement demandent de la prudence, et le versionnage s'en charge.
Nouveaux coûts	Un courtier à exploiter ; les factures apparaissent quelques secondes plus tard au lieu d'instantanément ; le schéma de l'événement est désormais un contrat public à versionner ; le processus est plus difficile à tracer sans une bonne observabilité.

L'événement transporte les totaux calculés par Commandes : la TVA est donc calculée en un seul endroit. Facturation n'a plus besoin de connaître les règles fiscales ni la structure des lignes de commande.

La refactorisation a supprimé un choix, pas tout couplage : Facturation dépend toujours de la signification de `OrderPlaced`. Cette dépendance est désormais explicite, étroite et versionnée, et c'est ce que signifie un couplage faible en pratique.

ATTENTION

Idées reçues courantes

- Les microservices ne sont pas automatiquement faiblement couplés. Des services qui partagent une base de données ou qui doivent être mis en production ensemble sont fortement couplés, quelle que soit la façon dont ils sont déployés.
- Supprimer une duplication peut augmenter le couplage. Deux services qui partagent une bibliothèque pour une règle métier changent désormais ensemble ; parfois, un peu de duplication donne la conception la moins couplée (voir la section 3.8 sur DRY).
- La messagerie asynchrone supprime le couplage temporel, pas le couplage sémantique. Un consommateur qui interprète mal un champ d'événement est tout aussi défaillant, simplement plus tard.
- Un couplage faible n'est pas la même chose que peu de traits entre les boîtes d'un schéma. Une seule dépendance envers une interface volumineuse, instable et mal spécifiée peut être pire que cinq dépendances envers des interfaces petites et stables.

Comment les professionnels utilisent ces idées

Les concepteurs expérimentés citent rarement le nom des niveaux lors des revues, mais ils les appliquent constamment : ils remettent en question chaque paramètre indicateur, recherchent les tables partagées et l'état global, demandent quel est l'objectif unique de chaque module, et observent dans la gestion de versions quels modules changent ensemble. Aux frontières entre équipes et entre services, ils sont stricts, car le couplage y est coûteux à défaire ; à l'intérieur d'un petit module détenu par une seule personne, ils sont plus souples, car il y coûte peu.

Ils rendent aussi le couplage visible. Un graphe de dépendances généré à partir du code, un décompte de Ca et Ce par composant et une liste des tables de base de données auxquelles accède plus d'un service font partie des vues les plus instructives d'une architecture, et tous peuvent être produits automatiquement et suivis dans le build. La section suivante aborde le principe qui maintient le couplage faible dès le départ : cacher les décisions derrière des interfaces.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une fonction dotée d'un paramètre de mode qui sélectionne des comportements complètement différents est-elle un exemple de couplage de contrôle ?

Parce que l'appelant doit savoir comment l'appelé est organisé à l'intérieur et choisir sa branche interne ; modifier la logique de l'appelé peut imposer des modifications chez les appelants.

▼ Un module a Ca = 12 et Ce = 1. Qu'est-ce que cela suggère ?

De nombreux modules dépendent de lui et il dépend de peu de choses : les modifications qui le touchent sont donc risquées et doivent être rares et prudentes ; son interface mérite une stabilité particulière.

▼ Deux services ne communiquent que par des événements, mais doivent toujours être mis en production ensemble. Quelle sorte de couplage subsiste ?

Le couplage de déploiement, généralement causé par des modifications non rétrocompatibles de l'événement partagé ou par des accords sémantiques cachés.

▼ Citez un coût du remplacement d'un appel direct par un événement.

Les résultats deviennent cohérents à terme plutôt qu'immédiatement ; vous héritez aussi d'un courtier à exploiter, d'un schéma d'événement à versionner et d'un déroulement plus difficile à tracer.

▼ À vous : Réservation dépend de Places, Paiements et E-mail. Remboursements dépend de Paiements et E-mail. Rapports dépend de Places et Paiements. Calculez Ca et Ce pour Paiements et Réservation. Lequel est le plus risqué à modifier, et lequel est le plus exposé ?

Paiements : Ca = 3 (Réservation, Remboursements, Rapports), Ce = 0. Réservation : Ca = 0, Ce = 3 (Places, Paiements, E-mail). Paiements est plus risqué à changer, car trois modules ressentent tout changement de son interface : gardez donc celle-ci étroite et stable. Réservation est plus exposé, puisqu'un changement dans l'une de ses trois dépendances peut le casser, mais changer Réservation lui-même est sans danger, car rien n'en dépend. Cette répartition du risque est saine.

SOURCES

- Martin Fowler, Reducing Coupling (IEEE Software, 2001): https://martinfowler.com/ieeeSoftware/coupling.pdf

- W. P. Stevens, G. J. Myers and L. L. Constantine, Structured Design (IBM Systems Journal, 1974): https://doi.org/10.1147/sj.132.0115
- Microservices: database per service pattern (microservices.io): https://microservices.io/patterns/data/database-per-service.html
- Martin Fowler, What do you mean by Event-Driven?: https://martinfowler.com/articles/201701-event-driven.html
- Law of Demeter: general formulation (Northeastern University): https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html
- Law of Demeter: object formulation (Northeastern University): https://www2.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/object-formulation.html
- Object-oriented programming: an objective sense of style (record): https://scilit.com/publications/8b7cd3e2eaafe26d184e99c8c9e37165
- Law of Demeter (Wikipedia, for authors and project): https://en.wikipedia.org/wiki/Law_of_Demeter
- Fowler: Mocks Aren't Stubs: https://martinfowler.com/articles/mocksArentStubs.html
- Fowler: TellDontAsk: https://martinfowler.com/bliki/TellDontAsk.html
- When did people favor composition over inheritance?: https://sicpers.info/2025/11/when-did-people-favor-composition-over-inheritance
- The Go FAQ: https://go.dev/doc/faq

Masquage de l'information, abstraction et interfaces

AVANT DE LIRE

Cette section s'appuie sur :

- 0.3 Organiser le code
- 0.6 API et intégration
- 3.1 Modularité

Termes déjà vus qu'elle utilise : Module, API (interface de programmation d'application), Classe et objet, Changement cassant (breaking change)

L'IDÉE EN BREF

Quand vous envoyez un colis, vous le remettez au guichet et recevez un numéro de suivi. Vous ne savez pas, et cela vous est égal, s'il voyage en camionnette, en train ou en avion, ni par quel dépôt il passe. La poste peut changer tout cela dès demain sans vous prévenir, car l'accord passé au guichet reste le même. Les modules logiciels fonctionnent au mieux comme ce guichet : ils offrent une promesse petite et claire, et cachent la façon dont ils la tiennent. Les ennuis commencent quand les clients se mettent à compter sur des détails que personne n'a promis, comme le fait que le colis arrive toujours avant dix heures.

CE QU'IL FAUT SAVOIR

- Le masquage de l'information (Parnas, 1972) signifie que chaque module cache une décision de conception difficile ou susceptible de changer derrière une interface qui en révèle le moins possible.
- L'encapsulation est un mécanisme du langage (champs privés, modules, contrôle d'accès) ; le masquage de l'information est un principe de conception qui porte sur les décisions à cacher. Vous pouvez encapsuler sans rien cacher d'utile.
- Une interface est un contrat ; la conception par contrat (Bertrand Meyer) le rend précis avec des préconditions, des postconditions et des invariants, et attribue la responsabilité de chaque défaillance à l'appelant ou au fournisseur.
- Toutes les abstractions non triviales fuient dans une certaine mesure (Joel Spolsky, 2002) ; les bons concepteurs exposent donc honnêtement les défaillances, les coûts et les caractéristiques de performances au lieu de faire comme s'ils n'existaient pas.

- Les modules profonds d'Ousterhout offrent beaucoup de fonctionnalités derrière une petite interface ; les modules superficiels ajoutent le coût d'une interface sans cacher grand-chose.
- La loi de Hyrum dit que tout comportement observable d'une API largement utilisée finira par être utilisé comme une garantie ; les API stables ont donc besoin de surfaces minimales, d'un versionnement explicite, de changements par ajout et de tests de contrat.

Le principe : cacher ce qui est susceptible de changer

Dans la section 3.1, l'exemple KWIC de David Parnas a montré que décomposer un programme par étapes de traitement dispersait chaque décision de conception entre plusieurs modules. Son alternative, le masquage de l'information (information hiding), confie à chaque module une décision à garder secrète : comment les lignes sont stockées, comment les décalages sont calculés, à quel moment le tri a lieu. L'interface du module est choisie pour révéler le moins possible de cette décision. Tout le reste du système ne dépend que de l'interface, si bien que la décision peut changer sans que les autres parties s'en aperçoivent.

La liste de Parnas des choses qui méritent d'être cachées reste une bonne liste de contrôle un demi-siècle plus tard : les structures de données et leur mode de stockage ; l'ordre dans lequel les opérations sont effectuées ; les détails du matériel, des périphériques et des systèmes d'exploitation ; les formats d'entrée et de sortie ; et les algorithmes qui pourraient être remplacés par de meilleurs. Les ajouts modernes suivent la même logique : quel fournisseur tiers est utilisé, quelle base de données, quel courtier de messages, quelle version d'une API externe, et quelles règles métier sont en cours de révision.

Le principe s'applique de la même façon à toutes les échelles. Une classe cache ses champs ; un paquet cache ses modules internes ; un service cache sa base de données, son langage et son déploiement ; l'API publique d'une organisation cache les services qui existent derrière elle. Dans chaque cas, ce qui est caché peut changer à peu de frais, et ce qui est exposé devient coûteux à changer, parce que d'autres en dépendent.

L'encapsulation n'est pas le masquage de l'information

Les deux termes sont souvent employés comme synonymes, mais ils ne désignent pas la même chose. L'encapsulation est un mécanisme : regrouper les données avec les opérations qui les manipulent et restreindre l'accès direct, au moyen de champs privés, de frontières de modules ou de modificateurs d'accès (section 0.3). Le masquage de l'information est un principe de conception : décider *quelles* décisions garder secrètes, et concevoir l'interface de façon qu'elle ne les révèle pas. L'encapsulation est l'un des outils qui servent à cacher l'information ; elle ne décide pas à votre place de ce qu'il faut cacher.

Il est facile d'encapsuler parfaitement tout en ne cachant rien. Une classe dont les champs sont privés, avec un getter et un setter publics pour chacun, est entièrement encapsulée, et pourtant chaque appelant sait exactement quels champs existent et de quels types ils sont. Changez la représentation interne, et tous les appelants cassent. L'inverse se produit aussi : Python n'impose aucune confidentialité, et pourtant un module Python bien conçu peut cacher très efficacement ses décisions grâce à un ensemble de fonctions soigneusement choisi.

```
# Encapsulated, but the representation leaks through the interface
class Price:
    def __init__(self, pounds: float):
        self._pounds = pounds
    def get_pounds(self) → float:          # callers now depend on 'float pounds'
        return self._pounds
    def set_pounds(self, value: float):
        self._pounds = value

# Information hiding: callers ask for what they need, not for the field
class Money:
    def __init__(self, minor_units: int, currency: str):
        self._minor, self._currency = minor_units, currency
    def add(self, other: "Money") → "Money": ...
    def allocate(self, ratios: list[int]) → list["Money"]: ...
    def format(self, locale: str) → str: ...
# Switching from pence to a decimal type later changes only this class.
```

Les deux classes sont encapsulées ; seule la seconde cache la décision sur la façon dont un prix est stocké.

SI VOUS DÉBUTEZ

Pas à pas : des champs privés qui ne cachent rien

L'idée qui demande le plus d'attention ici est qu'une classe peut garder toutes ses données privées et ne rien cacher d'utile pour autant. L'encapsulation met les données sous clé ; le masquage de l'information décide quelle décision de conception l'interface ne doit pas trahir. Suivez les deux classes ci-dessus à travers une modification.

- La première conception de la boutique est la classe Price. Son montant se trouve dans un champ privé : elle est donc correctement encapsulée, mais get_pounds renvoie le montant sous forme de nombre de livres à virgule flottante, et set_pounds en reçoit un.
- L'étape de paiement, les factures et les remboursements appellent chacun get_pounds et font leurs propres calculs avec le résultat : additionner des prix, ou répartir un remboursement entre plusieurs articles.
- Le chapitre 0 a expliqué que les nombres à virgule flottante ne sont qu'approximatifs. Certains totaux commencent à sortir faux d'un penny après arrondi ; l'équipe décide donc de stocker l'argent en pence entiers.
- Modifier le champ privé est facile, mais ne règle rien : get_pounds renvoie toujours des flottants, et les calculs qui tombent faux se trouvent chez chaque appelant. Pour corriger le problème, l'équipe doit changer l'interface et chaque appelant qui fait des calculs. Le champ privé cachait l'endroit où le nombre était conservé, mais pas la décision qui comptait : que l'argent est un flottant exprimé en livres.
- Dans la seconde conception, Money, les appelants ne reçoivent jamais le nombre brut. Ils demandent à Money d'additionner deux montants, d'en répartir un selon des proportions (allocate) ou de se mettre en forme pour l'affichage. Les calculs sur les montants vivent à l'intérieur de Money, si bien qu'un changement de représentation ne touche que cette classe.
- **La leçon :** les champs privés sont le verrou ; le masquage de l'information consiste à choisir ce qui se trouve derrière. Nommez d'abord la décision la plus susceptible de changer, puis vérifiez qu'aucune opération de l'interface ne la trahit.

L'abstraction et les interfaces comme contrats

Une abstraction est une vue simplifiée de quelque chose, qui garde ce qui compte pour un objectif donné et laisse de côté le reste. Un fichier est une abstraction qui recouvre des blocs de disque, des caches et des pilotes ; un *paiement* est une abstraction qui recouvre un réseau de cartes, l'API d'un prestataire et le processus de règlement d'une banque. L'abstraction et le

masquage de l'information sont les deux faces d'un même geste : choisir une abstraction décide de ce que montre l'interface ; cacher le reste permet de se fier à l'abstraction en toute sécurité.

Une interface est donc un contrat entre un module et ses utilisateurs. Sa partie formelle est ce qu'un vérificateur de types ou un schéma d'API peut voir : noms des opérations, types des paramètres et des valeurs de retour, types d'erreurs, noms des champs d'un corps JSON. Sa partie informelle est tout aussi contraignante : ce que fait l'opération, ses effets de bord, si elle peut être relancée sans risque, combien de temps elle prend, quelles erreurs sont possibles et ce qu'elles signifient. John Ousterhout souligne que la partie informelle ne peut être consignée que dans la documentation, et que si les utilisateurs doivent lire le code d'un module pour apprendre à s'en servir, le module ne fournit aucune véritable abstraction.

Conception par contrat

La conception par contrat, développée par Bertrand Meyer dans les années 1980 avec le langage Eiffel et décrite dans son livre *Object-Oriented Software Construction* et dans son article de 1992 *Applying 'Design by Contract'*, rend le contrat précis. Chaque opération a une précondition, que l'appelant doit rendre vraie avant l'appel ; une postcondition, que l'opération garantit à son retour si la précondition était satisfaite ; et chaque classe ou module a des invariants, des conditions qui restent vraies entre deux opérations quelles qu'elles soient. Le contrat répartit les responsabilités : si une précondition est violée, le bug est chez l'appelant ; si une postcondition est violée, il est chez le fournisseur.

Meyer s'est opposé à l'habitude inverse, la programmation défensive, dans laquelle chaque routine vérifie tout au cas où quelqu'un l'appellerait mal. Les vérifications redondantes ajoutent du code, de la complexité et de nouveaux chemins de défaillance. Chaque condition devrait plutôt relever clairement de la responsabilité d'un seul côté. Une routine dont la précondition est exigeante est plus simple ; une routine dont la précondition est tolérante est plus facile à appeler ; ce choix est une décision de conception, prise explicitement. Aux frontières d'un système, là où les entrées viennent d'utilisateurs non fiables ou d'autres organisations, la validation n'est pas facultative : le contrat y prévoit que le module frontière vérifie lui-même tout (chapitre 8).

```
class Account:
    """Invariant: balance_pence ≥ -overdraft_limit_pence."""

    def withdraw(self, amount_pence: int) → None:
        # Preconditions (the caller's obligations)
        if amount_pence ⩽ 0:
            raise ValueError("amount must be positive")
        if self.balance_pence - amount_pence < -self.overdraft_limit_pence:
            raise InsufficientFunds(self.id)
        old = self.balance_pence
        self.balance_pence -= amount_pence
        # Postcondition (the supplier's guarantee)
        assert self.balance_pence == old - amount_pence
        assert self.balance_pence ≥ -self.overdraft_limit_pence  # invariant
```

Un contrat mis par écrit : préconditions vérifiées à l'entrée, postcondition et invariant énoncés et vérifiés par des assertions.

Les trois mêmes idées s'appliquent aux services. Une spécification d'API, comme un document OpenAPI, énonce les préconditions (champs obligatoires, formats, authentification), les postconditions (la réponse et le changement d'état, par exemple *une commande existe avec le statut PLACED*) et les invariants sur lesquels un client peut compter (*le total d'une commande ne change jamais après le paiement*). Les tests de contrat pilotés par le consommateur, dans lesquels chaque client publie les attentes sur lesquelles il compte et le fournisseur les exécute dans son build, vérifient ces contrats automatiquement.

Les abstractions qui fuient

En novembre 2002, Joel Spolsky a publié *The Law of Leaky Abstractions*, qui énonce que «toutes les abstractions non triviales fuient, dans une certaine mesure.» Une abstraction qui fuit (leaky abstraction) est une abstraction dont les détails cachés affectent encore ses utilisateurs. TCP présente un flux fiable par-dessus un réseau non fiable, jusqu'à ce qu'un câble soit coupé et qu'il n'y parvienne plus. SQL vous permet d'énoncer ce que vous voulez, et non comment l'obtenir, jusqu'à ce que deux requêtes logiquement équivalentes diffèrent en vitesse d'un facteur mille et que vous deviez lire le plan d'exécution. Un mapping objet-relationnel (ORM) cache SQL jusqu'à ce qu'une boucle lance discrètement une requête par ligne.

La conséquence pour les architectes n'est pas que les abstractions sont mauvaises, mais qu'une interface doit être honnête sur les détails qui vont fuir. L'exemple classique est l'appel distant qui ressemble exactement à un appel de fonction local. Les appels réseau sont plus lents de plusieurs ordres de grandeur, peuvent dépasser leur délai d'expiration, peuvent réussir sur le serveur alors que la réponse se perd, et peuvent échouer indépendamment de l'appelant ; la section 6.1 les présente comme les illusions de l'informatique distribuée. Une bonne interface distante l'expose : elle prend un délai d'expiration en paramètre, renvoie des

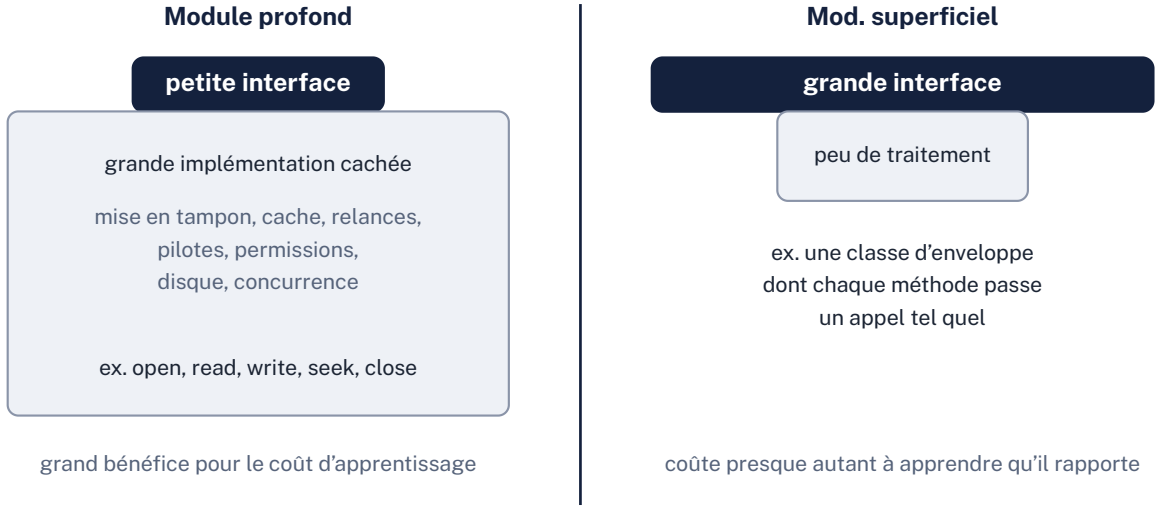
erreurs qui distinguent *refusé* de *résultat inconnu*, et accepte une clé d'idempotence pour que les nouvelles tentatives soient sans risque. Cacher complètement le réseau, c'est une fuite qui n'attend que son heure.

Modules profonds et modules superficiels

A Philosophy of Software Design de John Ousterhout (publié pour la première fois en 2018 ; deuxième édition en 2021) propose une manière utile de juger une interface. Représentez-vous chaque module comme un rectangle : le bord supérieur est son interface, que chaque utilisateur doit apprendre, et la surface est la fonctionnalité qu'il fournit. Un module profond a une petite interface au-dessus de beaucoup de fonctionnalités : il rembourse donc bien des fois le coût de son apprentissage. Un module superficiel a une interface presque aussi complexe que ce qu'il fait : il ajoute donc un coût sans cacher grand-chose.

Son exemple préféré de profondeur est l'entrée-sortie de fichiers sous Unix : une poignée d'appels (open, read, write, seek, close) cachent les systèmes de fichiers, l'organisation du disque, la mise en cache, les permissions, les périphériques et les accès concurrents. Son exemple de superficialité est une classe dont chaque méthode transmet un appel à une autre classe, ou une classe de liste chaînée dont l'interface expose presque tout ce qu'elle fait. Ousterhout soutient que le conseil courant de garder les classes et les méthodes très petites peut produire ce qu'il appelle la classite : de nombreuses classes superficielles dont les interfaces, mises bout à bout, représentent plus de complexité qu'un seul module plus profond.

Modules profonds et modules superficiels



L'image d'Ousterhout : la largeur du bord supérieur est l'interface que les utilisateurs doivent apprendre ; la surface en dessous est la fonctionnalité qu'il cache.

Tout le monde ne suit pas Ousterhout aussi loin. Les partisans des petites classes et des petites fonctions, comme Robert C. Martin dans *Clean Code*, soutiennent que les unités minuscules sont plus faciles à nommer, à tester et à réutiliser ; Ousterhout et Martin ont débattu de la question dans un échange écrit publié en 2025. Les deux camps s'accordent sur l'objectif de fond, qui est de réduire la complexité qu'un lecteur doit garder en tête. À l'échelle de l'architecture, le test de profondeur est particulièrement précieux : un service dont l'API reflète ses tables de base de données une à une, avec création, lecture, mise à jour et suppression pour chacune, est superficiel, et reporte toute la vraie logique sur ses clients.

La loi de Hyrum : tout ce qui est observable devient le contrat

La loi de Hyrum, nommée d'après Hyrum Wright, alors ingénieur logiciel chez Google, par son collègue Titus Winters, et popularisée par *Software Engineering at Google* (2020), constate qu'avec un nombre suffisant d'utilisateurs d'une API, peu importe ce que vous promettez dans le contrat : tous les comportements observables de votre système finiront par être utilisés par quelqu'un. L'ordre dans lequel une fonction renvoie les éléments d'un ensemble non ordonné, la formulation exacte d'un message d'erreur, un champ toujours présent bien que documenté comme facultatif, le fait qu'un appel prenne 50 millisecondes : si les utilisateurs peuvent le voir, certains d'entre eux s'y fieront.

Cette loi explique pourquoi le masquage de l'information doit être conçu dès le départ, et non espéré. Chaque comportement exposé par accident fait partie du contrat de fait, et le modifier cassera quelque chose chez quelqu'un, aussi clairement que la documentation ait indiqué qu'il n'était pas spécifié. Certaines équipes rendent délibérément aléatoires les comportements non spécifiés, comme l'ordre d'itération, pour que personne ne puisse en venir à en dépendre ; Go rend aléatoire l'ordre d'itération des maps pour cette raison. D'autres gardent des réponses légères, pour qu'il y ait moins de choses dont dépendre.

La loi de Postel et sa critique

Alors que la loi de Hyrum décrit ce qui arrive aux interfaces, la loi de Postel vous dit comment vous y comporter. Jon Postel a dirigé la rédaction des premières spécifications d'internet, et l'idée apparaît en 1980 dans la RFC 760, sur le protocole IP : « une implémentation doit être conservatrice dans son comportement d'envoi et libérale dans son comportement de réception. » La RFC 761, sur TCP, a donné la forme habituellement citée : « soyez conservateur dans ce que vous faites, soyez libéral dans ce que vous acceptez des autres. » On l'appelle aussi le **principe de robustesse**. C'est une maxime de conception, pas une loi de la nature. Elle convenait à son époque : de nombreuses implémentations indépendantes de spécifications imparfaites devaient fonctionner ensemble.

Elle reste utile sous une certaine forme. Le **lecteur tolérant** de Martin Fowler l'applique à l'intégration : un consommateur ne doit lire que ce dont il a besoin et ignorer le reste. Si le service Notifications lit deux champs d'un événement `OrderPlaced` et ignore les champs inconnus, Commandes peut ajouter des champs sans le casser, et c'est ce qui permet aux changements compatibles de la section 7.7 de fonctionner.

La critique porte sur ce qui se passe avec le temps. Associez un récepteur tolérant à la loi de Hyrum, et chaque erreur acceptée devient quelque chose sur quoi d'autres s'appuient. Supposons que l'intégration d'un restaurant envoie des prix sous la forme `"12,50"` et que la plateforme l'accepte sans rien dire. L'émetteur n'apprend jamais son erreur, d'autres partenaires la copient, et la bizarrerie devient une partie non écrite du contrat, que toute réécriture devra reproduire.

L'Internet Architecture Board l'a exposé dans la RFC 9413, *Maintaining Robust Protocols* (juin 2023), un document à caractère informatif, et non une norme, dont l'approbation a été large mais pas unanime. Elle soutient que tolérer les entrées inattendues mène à la dégradation des protocoles : les fautes restent cachées, les erreurs deviennent des normes de fait, et les nouvelles implémentations doivent copier les bogues des anciennes pour interopérer. Son exemple est celui de serveurs TLS qui échouaient sur des extensions vides, ce qui a obligé les clients à les contourner. Elle recommande une maintenance active des protocoles, afin que les ambiguïtés soient résolues dans la spécification et non absorbées, et une *intolérance vertueuse* : lever une erreur fatale à toute violation de la spécification, afin que les bogues apparaissent tôt.

Une règle pratique réconcilie les deux. Ne soyez tolérant que pour ce que le contrat vous autorise à ignorer, comme des champs supplémentaires inconnus. Soyez strict pour tout ce que le contrat spécifie : un champ obligatoire manquant, un type erroné ou une valeur non valide reçoit une erreur claire, pas une supposition. C'est surtout important pour l'argent, l'identité et la sécurité, où un récepteur qui « fait de son mieux » peut interpréter un message autrement que le système voisin.

Concevoir des API stables entre composants

Entre composants et services, les interfaces sont ce qu'il y a de plus coûteux à modifier, car les changer exige de se coordonner avec chaque consommateur. Les pratiques ci-dessous découlent de tout ce qui précède ; la section 0.6 a présenté les API et la section 7.7 traite en profondeur de la compatibilité des schémas.

Pratiques pour des API de composants et de services stables

Pratique	Ce que cela signifie	Pourquoi cela fonctionne
Surface minimale	N'exposez que les opérations et les champs dont un consommateur connu a besoin ; ajoutez-en sur demande.	Moins de choses dont dépendre, donc moins de choses à casser (loi de Hyrum).
Modéliser le domaine, pas le stockage	Proposez des opérations comme `place_order` et `cancel_order` plutôt qu'une création, une mise à jour et une suppression calquées sur les tables.	Le stockage peut changer sans que le contrat change ; le module reste profond.
Changement par ajout	Ajoutez des champs facultatifs et de nouvelles opérations ; ne renommez, ne supprimez et ne changez jamais le sens des éléments existants au sein d'une même version.	Les anciens clients continuent de fonctionner ; les nouveaux utilisent les nouvelles parties.
Lecteur tolérant (tolerant reader)	Les clients ignorent les champs qu'ils ne comprennent pas et ne dépendent pas de l'ordre des champs.	Les fournisseurs peuvent ajouter des champs sans rien casser chez personne.
Versionnement explicite	Introduisez une nouvelle version majeure (dans l'URL, un en-tête ou le schéma d'événement) pour les changements cassants, et faites tourner l'ancienne et la nouvelle côte à côte pendant une période annoncée.	Les consommateurs migrent à leur propre rythme ; le couplage de déploiement est évité.
Expansion-contraction (expand and contract)	Ajoutez la nouvelle forme, faites migrer les consommateurs, confirmez par la supervision que plus personne n'utilise l'ancienne forme, puis supprimez-la, dans des mises en production distinctes.	Les changements cassants se font sans mise en production conjointe.
Tests de contrat	Les consommateurs publient leurs attentes ; le build du fournisseur les exécute.	Les changements cassants accidentels sont détectés avant le déploiement.
Sémantique de défaillance honnête	Documentez les délais d'expiration, la possibilité de nouvelle tentative, l'idempotence et le sens des erreurs comme parties du contrat.	Évite les pires fuites de l'abstraction réseau.

Exemple commenté : une interface de paiement qui cache le prestataire

Une entreprise de livraison de repas encaisse ses paiements par l'intermédiaire du prestataire A. Le service financier veut ajouter le prestataire B pour réduire les frais dans un pays, et pouvoir changer de prestataire en moins d'une journée si l'un d'eux subit une panne.

Aujourd'hui, le code de l'étape de paiement appelle directement le SDK du prestataire A à onze endroits, utilise ses codes d'erreur pour décider quoi afficher au client et stocke ses objets de transaction dans la table des commandes. Chacun de ces points est une décision qui a fui.

La conception introduit une interface `PaymentGateway` qui appartient à la boutique, et non à un prestataire. Elle s'exprime dans les termes propres à la boutique : des montants en entiers d'unités mineures avec une devise, l'identifiant de commande de la boutique, une clé d'idempotence et trois résultats possibles. Chaque prestataire reçoit un adaptateur qui implémente l'interface et traduit vers et depuis l'API, les codes d'erreur et les particularités de ce prestataire. La configuration choisit l'adaptateur selon le pays. C'est le même geste que l'intermédiaire de la section 2.6, vu cette fois comme du masquage de l'information : la décision cachée est *quel prestataire, et comment l'appeler*.

```
from typing import Protocol, Literal
from dataclasses import dataclass

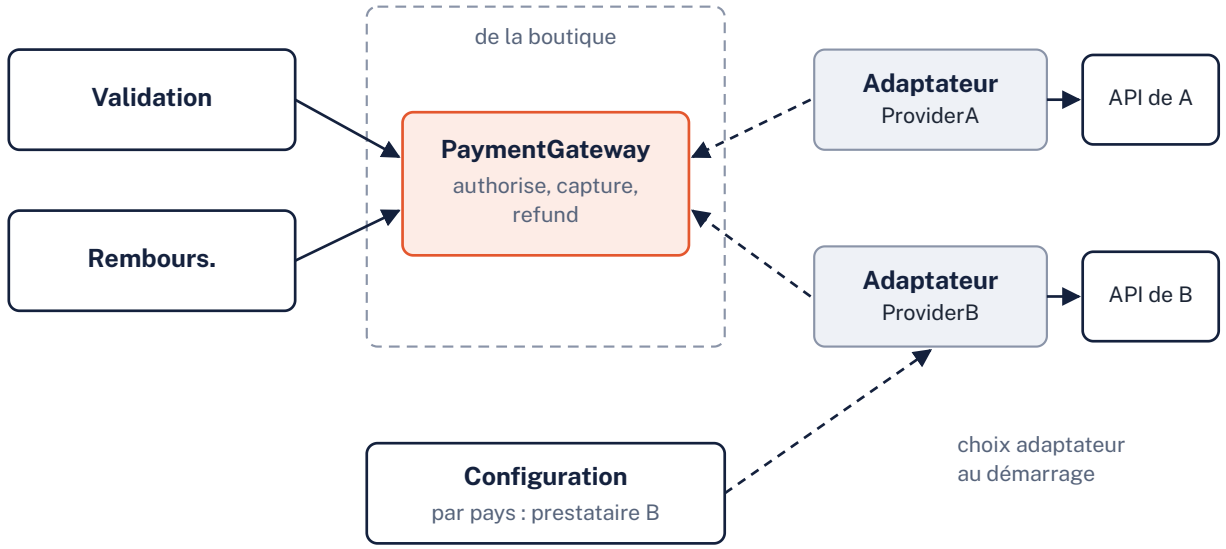
@dataclass(frozen=True)
class PaymentResult:
    outcome: Literal["approved", "declined", "unknown"]  # unknown: safe to retry
    reference: str | None        # our reference, not the provider's object

class PaymentGateway(Protocol):
    def authorise(self, order_id: str, amount_minor: int, currency: str,
                  idempotency_key: str) → PaymentResult: ...
    def capture(self, reference: str) → PaymentResult: ...
    def refund(self, reference: str, amount_minor: int) → PaymentResult: ...

class ProviderBAdapter:            # implements PaymentGateway
    def authorise(self, order_id, amount_minor, currency, idempotency_key):
        resp = self._client.create_charge(value=amount_minor / 100,   # B wants decimals
                                          currency=currency, key=idempotency_key)
        return PaymentResult(_map_status(resp.code), _store_ref(resp.id))
```

L'interface appartient à la boutique et parle sa langue ; seuls les adaptateurs savent qu'un prestataire existe.

L’interface de paiement et ses adaptateurs



Les appelants ne voient que l’interface ; prestataires, codes d’erreur, relances et règles de devise restent aux adaptateurs
Flèches pleines : dépend de ou appelle. Pointillés : implémente ou sélectionne.

Les appelants dépendent d’une interface qui appartient à la boutique ; les adaptateurs l’implémentent, et la configuration choisit celui qui s’exécute.

EXEMPLE COMMENTÉ

Ce que l'interface cache, et ce qu'elle ne cache délibérément pas

Vérification de la conception à l'aune des idées de cette section.

Décision cachée	Quel prestataire, son SDK et sa version d'API, ses codes d'erreur, ses règles de nouvelle tentative, ses montants décimaux ou entiers, ses identifiants de transaction.
Contrat (préconditions)	Un montant positif en unités mineures, une devise ISO 4217 prise en charge, une clé d'idempotence unique par tentative de paiement.
Contrat (postconditions)	`approved` signifie que les fonds sont réservés et qu'une référence est renvoyée ; `declined` signifie qu'aucun argent n'a bougé ; `unknown` signifie que le résultat n'a pas pu être déterminé et que le même appel peut être relancé avec la même clé.
Absence délibérée de masquage	L'interface expose `unknown` au lieu de prétendre que les paiements réussissent ou échouent toujours : la fuite du réseau fait partie du contrat, ce n'est pas une surprise.
Coût d'un nouveau prestataire	Avant : des modifications à 11 endroits d'appel, plus la table des commandes. Après : une nouvelle classe d'adaptateur, ses tests et une entrée de configuration.
Profondeur	Trois opérations cachent une part importante de logique propre à chaque prestataire : profonde plutôt que superficielle.

Méfiez-vous du piège du plus petit dénominateur commun : si le prestataire B offre une fonctionnalité utile qui manque à A, comme le paiement en plusieurs fois, soit l'interface s'enrichit d'une capacité facultative, soit la fonctionnalité reste indisponible. Les abstractions au-dessus de fournisseurs sacrifient toujours une partie des points forts de chacun en échange de la liberté d'en changer.

La loi de Hyrum s'applique toujours : si l'étape de paiement se met à dépendre de la latence habituelle du prestataire A, un prestataire B plus lent le révélera. Inscrivez les délais d'expiration dans le contrat.

ATTENTION

Erreurs courantes

- Des getters et des setters pour chaque champ : la classe est encapsulée, mais la représentation est entièrement exposée.
- Des interfaces nommées d'après l'implémentation actuelle, comme `StripePaymentService` ou `S3FileStore`, ce qui va à l'encontre du but même de l'interface.
- Des couches intermédiaires qui ajoutent une interface mais ne cachent rien, rendant le système plus superficiel et plus long à lire.
- Cacher les défaillances : des appels distants déguisés en appels locaux, sans délai d'expiration et sans moyen de distinguer un résultat inconnu d'un refus.
- Renvoyer des objets internes, comme des lignes de base de données ou la réponse d'un prestataire, par une API publique, ce qui les fait discrètement entrer dans le contrat.

Comment les professionnels l'appliquent

Les concepteurs expérimentés partent des besoins des appelants pour concevoir une interface, en l'écrivant dans le langage de l'appelant, et ne conçoivent l'implémentation qu'ensuite. Ils tiennent la liste des décisions que cache chaque module important et examinent les nouvelles interfaces au regard de cette liste. Ils traitent chaque API publique comme un produit doté d'un cycle de vie (proposition, revue, versionnement, dépréciation) parce que, d'après la loi de Hyrum, tout ce qui est publié sera utilisé. Et ils acceptent que les abstractions fuient, si bien qu'ils rendent les fuites explicites : budgets de latence, sémantique des erreurs et règles de nouvelle tentative figurent dans le contrat.

Le masquage de l'information vous dit ce que chaque module doit garder pour lui. Il ne vous dit pas dans quel sens doivent pointer les dépendances entre modules, ni ce qui se passe quand elles forment des boucles. C'est le sujet de la section 3.4.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi une classe dont tous les champs sont privés, avec des getters et des setters publics pour chacun, peut-elle quand même échouer à masquer l'information ?**

Parce que l'interface révèle la représentation interne exacte : les appelants en dépendent, et tout changement de représentation les casse. L'encapsulation est présente, mais aucune décision n'est cachée.

▼ **En conception par contrat, qui est en faute si une précondition est violée ?**

L'appelant, car satisfaire la précondition est son obligation ; une postcondition violée signale un bug chez le fournisseur.

▼ **Qu'est-ce qui rend un module profond au sens d'Ousterhout ?**

Il fournit beaucoup de fonctionnalités derrière une interface petite et simple, si bien que le coût d'apprentissage de l'interface est faible par rapport à ce qu'elle cache.

▼ **Pourquoi la loi de Hyrum plaide-t-elle pour des réponses d'API réduites ?**

Tout champ ou comportement observable finira par être utilisé par un client ; tout ce qui est exposé devient donc difficile à changer, et exposer moins laisse plus de liberté.

▼ **À vous : Critiquez cette interface de notification : sendSms(phone, text, vendorRetryCount), qui renvoie un objet VendorXResponse que les appelants inspectent pour y trouver les codes d'erreur du fournisseur. Que laisse-t-elle fuiter, et que changeriez-vous, tout en restant honnête sur les défaillances ?**

Elle laisse fuiter le réglage des nouvelles tentatives, le type de réponse et les codes d'erreur du fournisseur : changer de fournisseur casserait donc tous les appelants. Définissez une interface qui appartient à votre entreprise, comme send_message(recipient, text), renvoyant sent, rejected ou try-later. Cachez les nouvelles tentatives, le SDK et les codes dans un adaptateur. Ne faites pas comme si elle ne pouvait pas échouer : le contrat doit indiquer que la remise peut être retardée ou échouer, et à quelle vitesse elle répond. Les abstractions fuient : exposez donc ces faits honnêtement.

SOURCES

- Joel Spolsky, The Law of Leaky Abstractions (2002): https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/
- Hyrum's Law: https://www.hyrumslaw.com/
- Bertrand Meyer, Applying 'Design by Contract' (IEEE Computer, 1992): https://se.inf.ethz.ch/~meyer/publications/computer/contract.pdf
- John Ousterhout, A Philosophy of Software Design (book page): https://web.stanford.edu/~ouster/cgi-bin/book.php
- Martin Fowler, Tolerant Reader: https://martinfowler.com/bliki/TolerantReader.html
- RFC 9413: Maintaining Robust Protocols: https://www.rfc-editor.org/rfc/rfc9413.html
- RFC 761: Transmission Control Protocol: https://www.rfc-editor.org/rfc/rfc761.html

SECTION 3.4 SUR 3.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Gérer les dépendances : sens, cycles et stabilité

AVANT DE LIRE

Cette section s'appuie sur :

- 0.3 Organiser le code
- 3.1 Modularité
- 3.2 Couplage et cohésion

Termes déjà vus qu'elle utilise : Couplage afférent (Ca), Couplage efférent (Ce), Inversion de contrôle, Fonction d'aptitude (fitness function), Interface

L'IDÉE EN BREF

Pensez à un immeuble. Les étages supérieurs reposent sur les étages inférieurs, et les fondations ne reposent sur rien de ce qui se trouve au-dessus. Vous pouvez refaire la décoration du dernier étage sans toucher aux fondations, mais si vous creusez sous les fondations, tout ce qui est au-dessus est menacé. Imaginez maintenant deux étages qui se soutiennent mutuellement : vous ne pourriez jamais reconstruire ni l'un ni l'autre. Les modules logiciels reposent les uns sur les autres de la même façon. Les bonnes structures placent en haut les parties qui changent souvent, posées sur des parties solides et rarement modifiées en dessous, et ne laissent jamais deux parties se soutenir l'une l'autre en boucle.

CE QU'IL FAUT SAVOIR

- Une dépendance pointe du module qui a besoin de quelque chose vers le module dont il a besoin ; les changements se propagent à contre-sens des flèches, si bien que le sens des dépendances décide qui est affecté par les changements de qui.
- Le principe des dépendances acycliques interdit les cycles entre composants, car tous les composants d'un cycle doivent être construits, testés et mis en production ensemble.
- Le principe des dépendances stables dit de dépendre dans le sens de la stabilité ; l'instabilité vaut I = Ce / (Ca + Ce), de 0 (uniquement des dépendances entrantes) à 1 (uniquement des dépendances sortantes).
- Le principe des abstractions stables dit que les composants stables doivent être abstraits ; le degré d'abstraction A est le nombre de types abstraits divisé par le nombre total de types, et la distance à la séquence principale vaut D = |A + I - 1|.

- On casse les cycles par l'inversion des dépendances (le module dont on dépend possède une interface que l'autre implémente) ou en extrayant la partie commune dans un nouveau composant, plus stable.
- L'organisation en couches stricte n'autorise les appels que vers la couche immédiatement inférieure, l'organisation souple vers n'importe quelle couche inférieure ; ces deux règles, comme toute autre règle de dépendance, devraient être vérifiées automatiquement avec des outils comme ArchUnit, dependency-cruiser ou import-linter.

Pourquoi le sens compte

Une dépendance existe quand un module a besoin d'un autre pour compiler, s'exécuter ou se comporter correctement. Représentez-la par une flèche qui va du module qui dépend vers le module dont il dépend : de Paiement vers Tarification. La flèche vous apprend quelque chose d'important sur le changement. Si Tarification change d'interface ou de comportement, Paiement devra peut-être changer ; si Paiement change, Tarification n'en a cure. Les changements se propagent à contre-sens des flèches. Le sens de chaque dépendance est donc une décision sur qui sera dérangé par les changements de qui.

Les architectes s'intéressent à deux sortes de sens faciles à confondre. Le *flot de contrôle* indique qui appelle qui à l'exécution. La *dépendance de code source* indique qui importe, référence ou compile contre qui. En général, les deux pointent dans le même sens, mais ce n'est pas obligatoire. Un framework appelle votre code sans en dépendre (la section 0.3 a appelé cela l'inversion de contrôle), et grâce à une interface, un module de haut niveau peut appeler un module de bas niveau alors que la dépendance de code source pointe dans l'autre sens. Cette liberté de choisir le sens de la dépendance de code source indépendamment du flot de contrôle est l'outil le plus puissant de cette section.

Entre services, le sens se manifeste sous forme de connaissance. Si Commandes appelle l'API de Facturation, Commandes dépend de Facturation et doit changer quand l'API de Facturation change. Si Commandes publie un événement que Facturation consomme, la dépendance s'inverse : Facturation dépend du contrat d'événement de Commandes, et Commandes ne sait même pas que Facturation existe (section 3.2). Choisir entre un appel et un événement, c'est donc en partie choisir le sens d'une dépendance.

Le principe des dépendances acycliques

Robert C. Martin a formulé trois principes pour les dépendances entre composants, d'abord dans des articles au milieu des années 1990, puis dans *Agile Software Development: Principles, Patterns, and Practices* (2002) et *Clean Architecture* (2017). Le premier, le principe des

dépendances acycliques (ADP), dit que le graphe des dépendances entre composants ne doit contenir aucun cycle : ce doit être un graphe orienté acyclique.

Un cycle de dépendances, par exemple Commandes dépend de Notifications, Notifications dépend de Clients et Clients dépend de Commandes, transforme trois composants en un seul. Aucun ne peut être compilé ou testé sans les autres ; un changement dans l'un peut faire le tour de la boucle et revenir jusqu'à lui ; aucun ne peut être mis en production seul ; et personne ne peut dire lequel construire en premier. Martin introduit le principe avec ce qu'il appelle le *syndrome du lendemain matin* : vous rentrez chez vous avec du code qui fonctionne et revenez le lendemain pour le trouver cassé, parce que quelqu'un d'autre a modifié quelque chose dont vous dépendez. Son remède consiste à découper le système en composants mis en production séparément, et cela ne fonctionne que si les dépendances entre eux ne contiennent aucun cycle. Les cycles empêchent aussi d'extraire plus tard un composant sous forme de bibliothèque ou de service, car il entraîne toute la boucle avec lui.

Le Java Platform Module System impose l'ADP entre modules : la Java Language Specification fait une erreur de compilation du fait qu'un module dépende, directement ou indirectement, de lui-même. La plupart des autres langages autorisent les cycles entre paquets ; les équipes les détectent donc avec des outils. Les cycles entre classes à l'intérieur d'un petit composant sont moins nuisibles, et Martin Fowler, entre autres, les y tolère ; le principe compte surtout entre des composants que des personnes différentes construisent et mettent en production.

Casser les cycles

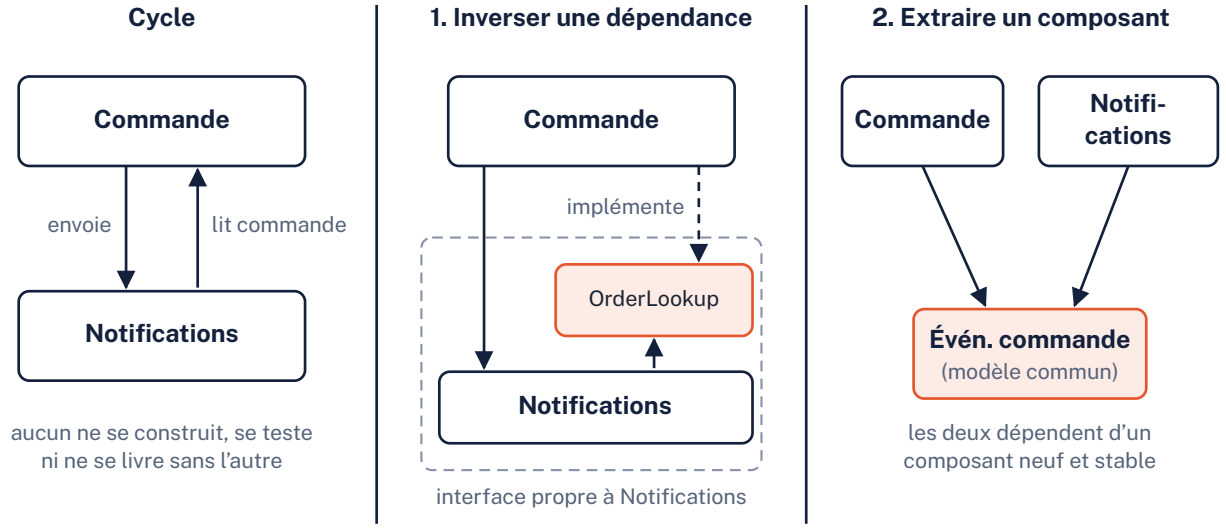
Tout cycle peut être cassé, et il existe deux techniques classiques. Le schéma ci-dessous les montre toutes deux pour un cycle dans lequel Commandes appelle Notifications pour envoyer une confirmation, et Notifications rappelle Commandes pour lire les détails de la commande.

Inversion des dépendances. Notifications déclare une interface qui décrit exactement ce dont il a besoin, par exemple `OrderLookup` avec une seule méthode, et Commandes l'implémente. La flèche de Notifications vers Commandes pointe maintenant dans l'autre sens : Commandes dépend de Notifications (pour l'appeler et pour implémenter son interface), et Notifications ne dépend de rien dans Commandes. C'est l'inversion des dépendances, que la section 3.5 traite comme le D de SOLID. Le détail clé est la propriété : l'interface appartient au module qui l'*utilise*, et non à celui qui l'implémente.

Extraire un nouveau composant. Repérez ce dont les deux côtés ont réellement besoin l'un de l'autre, souvent un petit modèle ou un contrat commun, et placez-le dans un nouveau composant dont les deux dépendent. Ici, Commandes publie un type d'événement défini dans un composant *événements de commande*, et Notifications lit les détails de la commande dans

l'événement au lieu de rappeler Commandes. Le nouveau composant est utilisé par les deux et ne dépend d'aucun, si bien qu'il est plus stable que chacun d'eux : c'est exactement ce que demande le principe suivant.

Deux façons de casser un cycle



1. Commandes appelle Notifications, qui exprime son besoin par une interface ; Commandes l'implémente.
2. La partie commune passe dans son propre composant ; Notifications ne lit plus Commandes.

Les flèches vont du module qui dépend vers le module dont il dépend. Pointillés : implémente.

On peut casser un cycle en inversant une dépendance au moyen d'une interface qui appartient à son utilisateur, ou en extrayant ce que les deux côtés partagent dans un nouveau composant.

SI VOUS DÉBUTEZ

Pas à pas : même appel, flèche inversée

Le point qui fait le plus souvent trébucher ici est que « qui appelle qui pendant l'exécution du programme » et « quel code mentionne quel code » sont deux choses distinctes, et qu'une interface vous permet de choisir la seconde. « Casser les cycles » vient de montrer la solution pour Commandes et Notifications ; suivez un e-mail de confirmation à travers elle deux fois, une fois tel qu'il s'exécute et une fois tel qu'il est écrit.

- Avant la solution, Commandes appelait Notifications pour envoyer la confirmation, et Notifications rappelait Commandes pour obtenir les détails de la commande. Le code de chaque module nommait l'autre : un cycle de dépendances.
- Après la solution, regardez-le s'exécuter. Commandes appelle Notifications comme avant. Au démarrage, Notifications a reçu un OrderLookup, qui est en fait l'implémentation propre à Commandes. Lui demander les détails de la commande exécute donc le code de Commandes, comme avant : le flot de contrôle n'a pas changé.
- Lisez maintenant le code. Notifications ne nomme qu'OrderLookup, qu'il déclare lui-même, et ne nomme jamais Commandes. Ainsi, à l'exécution, l'appel va de Notifications vers le code de Commandes, tandis que la flèche du code source va de Commandes vers Notifications.
- Cet écart, c'est le gain. Notifications peut désormais être compilé et testé sans rien de Commandes dans le build, à l'aide d'un petit faux OrderLookup qui renvoie des détails de commande inventés.
- C'est la propriété qui fait que cela fonctionne. Si OrderLookup avait été déclaré dans Commandes, Notifications devrait encore nommer le code de Commandes pour s'en servir, et le cycle subsisterait. L'interface doit vivre avec Notifications, le côté qui en a besoin.
- **La leçon :** l'appel à l'exécution et la dépendance de code source peuvent pointer dans des sens opposés. Une interface qui appartient à son utilisateur vous permet d'orienter la dépendance comme il le faut : c'est l'inversion des dépendances à laquelle cette section revient sans cesse.

Le principe des dépendances stables et l'instabilité

Certains composants changent souvent (interfaces utilisateur, intégrations avec des partenaires, nouvelles fonctionnalités) ; d'autres devraient changer rarement (concepts centraux du domaine, contrats partagés, gestion de l'argent). Le principe des dépendances

stables (SDP) dit : dépendez dans le sens de la stabilité. Un composant censé être facile à changer ne devrait pas être utilisé par un composant difficile à changer, sinon il deviendra lui aussi difficile à changer.

Martin mesure la stabilité par les dépendances, et non par la fréquence à laquelle le code est modifié. Un composant dont beaucoup d'autres dépendent est *stable*, au sens où il est coûteux à changer : chaque changement risque de casser ce qui dépend de lui. Un composant qui dépend de beaucoup d'autres et dont rien ne dépend est *instable* : rien ne l'empêche de changer, et ses dépendances peuvent l'y obliger. À partir du couplage afférent (Ca, entrant) et du couplage efférent (Ce, sortant) de la section 3.2, l'instabilité vaut :

I = Ce / (Ca + Ce)

I va de 0 à 1. I = 0 signifie uniquement des dépendances entrantes : stabilité maximale, des responsabilités envers beaucoup, aucune dépendance. I = 1 signifie uniquement des dépendances sortantes : instabilité maximale, rien ne dépend de lui. Exprimé sous forme de métrique, le SDP dit que I doit diminuer le long de chaque flèche de dépendance : chaque composant doit dépendre de composants dont le I est plus faible que le sien. Une dépendance d'un composant stable (I = 0,1) vers un composant volatil (I = 0,8) est un signal d'alerte ; le remède habituel est là encore l'inversion des dépendances, en plaçant une interface à un endroit stable.

Le principe des abstractions stables et la séquence principale

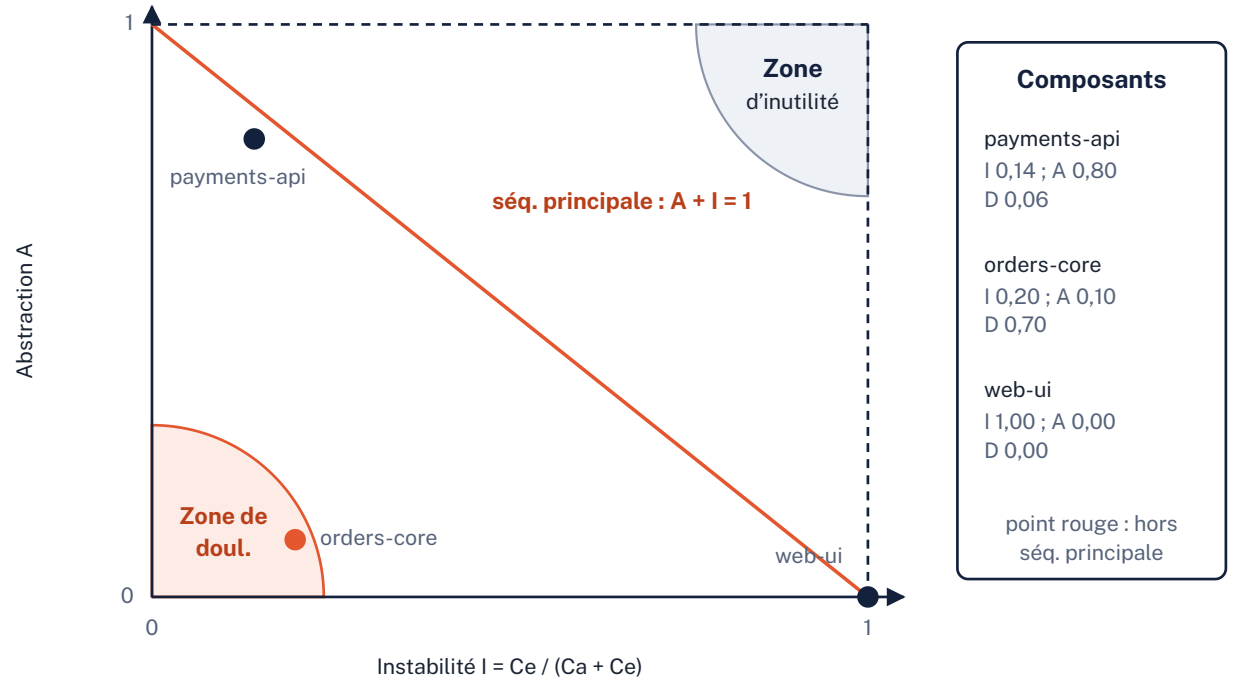
La stabilité a un coût : un composant dont le I est proche de 0 est difficile à changer. S'il est rempli de code concret, le système devient rigide. Le principe des abstractions stables (SAP) résout ce problème : un composant doit être aussi abstrait qu'il est stable. Les composants stables doivent se composer surtout d'interfaces et de types abstraits, pour que leur comportement puisse être étendu par de nouvelles implémentations ailleurs sans les modifier ; les composants instables peuvent être concrets, car ils sont de toute façon faciles à changer. Ensemble, le SDP et le SAP reviennent au principe d'inversion des dépendances appliqué à des composants entiers : les dépendances pointent vers la stabilité, et la stabilité va de pair avec l'abstraction.

Le degré d'abstraction se mesure par A = Na / Nc, où Na est le nombre de classes abstraites et d'interfaces du composant et Nc le nombre total de classes. A = 0 signifie entièrement concret ; A = 1 signifie uniquement des abstractions. Reporter A en fonction de I donne un carré. La droite qui va de (I = 0, A = 1) à (I = 1, A = 0) est la séquence principale (main sequence) : stable et abstrait à une extrémité, instable et concret à l'autre, avec des mélanges équilibrés entre les deux. La distance à cette droite vaut :

D = |A + I - 1|

D va de 0 (sur la droite) à 1 (dans un coin éloigné). Le coin proche de (0, 0) est la *zone de douleur* : stable, parce que beaucoup de choses en dépendent, et concret, si bien que tout changement est à la fois nécessaire et douloureux. Un schéma de base de données que chaque module lit directement se trouve là. Martin note que certains composants y ont leur place sans dommage, comme une bibliothèque utilitaire concrète et mature qui n'a jamais besoin de changer ; la zone ne fait mal que pour les composants volatils. Le coin proche de (1, 1) est la *zone d'inutilité* : des abstractions dont rien ne dépend, typiquement des interfaces laissées à l'abandon après un refactoring ou conçues pour une extension qui n'est jamais venue.

La séquence principale



Les composants proches de la séquence principale équilibrent degré d'abstraction et stabilité ; la zone de douleur contient du code stable et concret, et la zone d'inutilité des abstractions que personne n'utilise.

EXEMPLE COMMENTÉ

Exemple commenté : calculer I, A et D pour trois composants

Une boutique compte trois composants. Selon la définition de Martin, Ca compte les classes extérieures au composant qui dépendent de classes situées à l'intérieur, et Ce compte les classes intérieures qui dépendent de classes extérieures. Les chiffres proviennent d'un outil d'analyse des dépendances.

payments-api (5 classes, dont 4 interfaces)	Ca = 6, Ce = 1. I = 1 / (6 + 1) = 0,14. A = 4 / 5 = 0,80. D = |0,80 + 0,14 - 1| = 0,06.
orders-core (20 classes, dont 2 abstraites)	Ca = 8, Ce = 2. I = 2 / (8 + 2) = 0,20. A = 2 / 20 = 0,10. D = |0,10 + 0,20 - 1| = 0,70.
web-ui (12 classes, aucune abstraite)	Ca = 0, Ce = 9. I = 9 / (0 + 9) = 1,00. A = 0 / 12 = 0,00. D = |0 + 1 - 1| = 0,00.
Vérifier le SDP le long des flèches	web-ui (1,00) dépend d'orders-core (0,20) et de payments-api (0,14) ; orders-core (0,20) dépend de payments-api (0,14). I diminue le long de chaque flèche : le SDP est donc respecté.
Diagnostic	orders-core a D = 0,70 : stable (beaucoup de dépendants) mais concret. Si ses règles changent souvent, chaque changement met en risque huit classes dépendantes. Il se trouve dans la zone de douleur.
Remède et nouveau calcul	Extrayez les quatre types qu'utilisent ses dépendants dans orders-api (3 interfaces et 1 type valeur : A = 0,75 ; Ca vaut 8 ou plus, puisque les classes d'implémentation en dépendent désormais aussi, et Ce = 0, donc I = 0 ; D = |0,75 + 0 - 1| = 0,25). Les 16 classes concrètes restantes deviennent orders-impl, dont seul dépend le câblage de démarrage (Ca = 1) et qui dépend d'orders-api et de deux autres composants (Ce = 3) : I = 3 / 4 = 0,75, A = 0, D = 0,25.

Le remède rapproche nettement les deux parties de la séquence principale (de 0,70 à 0,25 chacune). Les dépendants s'appuient désormais sur un composant petit, stable et surtout abstrait, et le code concret volatil se trouve là où il est peu coûteux à changer.

Les chiffres sont des heuristiques, pas des objectifs. Un D de 0,70 est une raison d'aller voir, pas la preuve d'un problème ; si orders-core ne changeait jamais, il ne ferait aucun tort là où il est.

ATTENTION

Utiliser les métriques avec discernement

- Les outils diffèrent par ce qu'ils comptent (classes, fichiers, paquets) et par le fait d'inclure ou non les bibliothèques tierces ; ne comparez que des chiffres issus du même outil et de la même configuration.
- Dans les langages sans classes abstraites, comme Python avec le typage canard (duck typing), A est approximatif ; comptez les Protocols et les classes de base abstraites, et traitez la métrique comme un point de départ pour la discussion.
- Les tendances sont plus instructives que les instantanés : un composant dont le D augmente version après version est en train de dériver vers les ennuis.
- Les métriques peuvent être manipulées, par exemple en ajoutant des interfaces inutilisées pour augmenter A. La section 3.8 revient sur la loi de Goodhart.

Organisation en couches stricte et souple

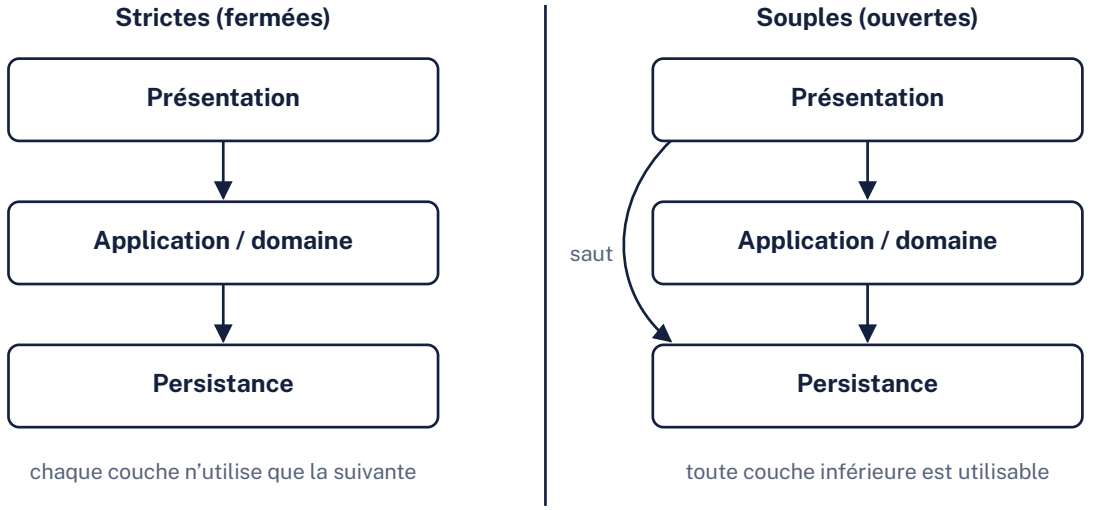
L'organisation en couches (layering) est la règle de dépendance la plus ancienne et la plus courante : disposer les modules en couches, comme la présentation, l'application ou le domaine, et la persistance, et n'autoriser chaque couche à utiliser que les couches situées en dessous. Les couches inférieures ne savent rien des couches supérieures, si bien qu'on peut ajouter une nouvelle interface utilisateur sans modifier le domaine. L'organisation en couches est un style architectural à part entière (section 5.2) ; ici, ce qui nous intéresse est la règle de dépendance qu'elle impose.

Dans l'organisation en couches **stricte** (fermée), chaque couche ne peut utiliser que la couche immédiatement inférieure. Chaque couche cache alors tout ce qui se trouve en dessous d'elle, et remplacer une couche inférieure n'affecte qu'une seule autre couche ; le prix à payer est du code de simple transmission, où la couche intermédiaire se contente de relayer des appels auxquels elle n'a rien à ajouter. Dans l'organisation en couches **souple** (ouverte), une couche peut utiliser n'importe quelle couche inférieure. Cela supprime le code répétitif, mais un changement dans la couche du bas peut alors affecter toutes les couches au-dessus. Beaucoup de systèmes réels mélangent les deux, en marquant certaines couches comme ouvertes, par exemple une couche partagée de journalisation ou d'utilitaires que tout le monde peut utiliser.

L'organisation en couches classique a une faiblesse notoire : la couche domaine dépend de la couche persistance, si bien que les règles métier les plus importantes dépendent de détails de base de données. Les architectures hexagonale, en oignon et Clean Architecture (section 5.6)

appliquent l'inversion des dépendances pour y remédier : le domaine possède des interfaces comme `OrderRepository`, et le code de persistance les implémente, si bien que toutes les dépendances de code source pointent vers l'intérieur, vers le domaine.

Organisation en couches stricte et souple



L'organisation en couches stricte n'autorise les appels que vers la couche immédiatement inférieure ; l'organisation souple permet aussi à une couche de sauter des couches situées en dessous.

Vérifier automatiquement les règles de dépendance

Les règles de dépendance écrites uniquement dans un wiki sont enfreintes en quelques semaines. Le moyen fiable de les faire respecter est de les exprimer sous forme de vérifications exécutables qui tournent dans le build et le font échouer quand une règle est enfreinte. Une telle vérification est une fonction d'aptitude (fitness function) architecturale, une idée présentée dans la section 2.8 et développée dans la section 11.3. Trois outils open source largement utilisés couvrent les principaux écosystèmes.

ArchUnit (Java et autres langages de la JVM) vous permet d'écrire des règles d'architecture sous forme de tests unitaires ordinaires : quels paquets peuvent accéder à quels autres, définitions de couches, et absence de cycles entre tranches de la base de code. **dependency-cruiser** (JavaScript et TypeScript) analyse les imports et les valide par rapport aux règles d'un fichier de configuration, et sa configuration de départ interdit déjà les dépendances circulaires. **import-linter** (Python) vérifie des *contrats* déclarés dans un fichier de configuration ; ses types de contrats couvrent notamment les imports interdits, les couches, l'indépendance entre modules frères, les modules protégés et l'acyclicité entre modules frères.

```
@ArchTest
static final ArchRule layers = layeredArchitecture()
    .consideringAllDependencies()
    .layer("Web").definedBy("..web..")
    .layer("Domain").definedBy("..domain..")
    .layer("Persistence").definedBy("..persistence..")
    .whereLayer("Web").mayNotBeAccessedByAnyLayer()
    .whereLayer("Domain").mayOnlyBeAccessedByLayers("Web", "Persistence")
    .whereLayer("Persistence").mayNotBeAccessedByAnyLayer();

@ArchTest
static final ArchRule noCycles =
    slices().matching("com.shop.(*)..").should().beFreeOfCycles();
```

ArchUnit (Java) : une règle de couches et une vérification de cycles écrites sous forme de tests.

```
// .dependency-cruiser.js
module.exports = {
  forbidden: [
    { name: "no-circular", severity: "error",
      from: {}, to: { circular: true } },
    { name: "domain-not-to-db", severity: "error",
      comment: "Domain code must use repository interfaces, not the database",
      from: { path: "^src/domain" },
      to: { path: "^src/db" } },
  ],
};
```

dependency-cruiser (JavaScript et TypeScript) : interdire les cycles et empêcher le domaine d'importer la couche base de données.

```
# .importlinter
[importlinter]
root_package = shop

[importlinter:contract:layers]
name = Web above domain above adapters
type = layers
layers =
    shop.web
    shop.domain
    shop.adapters

[importlinter:contract:features-independent]
name = Feature packages do not import each other
type = independence
modules =
    shop.catalogue
    shop.ordering
    shop.payments
```

import-linter (Python) : un contrat de couches et un contrat d'indépendance, vérifiés en exécutant lint-imports dans le pipeline.

Remarquez que la règle ArchUnit ci-dessus permet à Persistence d'accéder à Domain, mais pas l'inverse : c'est la disposition centrée sur le domaine qui vient d'être décrite, et non l'organisation en couches classique. Des règles comme celles-ci coûtent peu à ajouter, s'exécutent en quelques secondes et transforment un schéma d'architecture en quelque chose que le build défend. Quand une base de code existante enfreint déjà une règle, la plupart des outils peuvent enregistrer les infractions actuelles comme référence et n'échouer que sur les nouvelles, si bien que la règle empêche la situation de s'aggraver dès le jour où elle est introduite.

GUIDE DE DÉCISION

Guide de décision : que faire d'une dépendance indésirable ?

Utilisez ce guide quand une vérification de dépendances échoue ou qu'une revue découvre une dépendance qui pointe dans le mauvais sens.

Q1 **La dépendance fait-elle partie d'un cycle entre composants ?**

- **Oui** Cassez d'abord le cycle (ADP) : poursuivez ci-dessous pour choisir comment.
- **Non** Passez à la question de la stabilité.

Q2 **Un côté n'a-t-il besoin que d'un service petit et bien défini de la part de l'autre ?**

- **Oui** Inversez la dépendance : le côté qui utilise le service possède une interface, et l'autre côté l'implémente.
- **Non** Passez à la question suivante.

Q3 **Les deux côtés dépendent-ils d'un modèle, d'un contrat ou d'un élément de logique partagé ?**

- **Oui** Extrayez-le dans un nouveau composant dont les deux dépendent, et faites en sorte que ce composant soit petit, stable et surtout abstrait.
- **Non** Demandez-vous si les deux composants n'en forment pas réellement un seul : fusionnez-les, puis redécoupez-les selon une meilleure frontière (section 3.1).

Q4 **Un composant stable (I faible) dépend-il d'un composant volatil (I élevé) ?**

- **Oui** Cela enfreint le SDP. Placez une interface dans le composant stable, ou dans un nouveau composant stable, et déplacez le code volatil derrière elle.
- **Non** Le sens est acceptable. Inscrivez la règle dans une fonction d'aptitude pour qu'il le reste.

Comment les professionnels gèrent les dépendances

Les architectes expérimentés gardent une image du graphe des dépendances entre composants et la consultent régulièrement, en la générant à partir du code plutôt qu'en la dessinant à la main. Ils décident quels composants sont censés être stables (concepts du domaine, contrats partagés) et lesquels sont volatils (interfaces utilisateur, intégrations), et font pointer les flèches du volatil vers le stable. Ils traitent les nouveaux cycles comme des échecs du build, et non comme des questions de style. Et ils appliquent le même raisonnement entre services : les bibliothèques partagées et les schémas d'événements partagés sont les composants stables d'un système distribué, qui méritent des surfaces minimales et un versionnement soigné, tandis que l'intérieur de chaque service reste libre de changer.

Rien de tout cela n'exige de calculer les métriques avec précision. Leur valeur tient aux questions qu'elles obligent à poser : dans quel sens cette flèche doit-elle pointer, qu'arrive-t-il à tout ce qui est en amont si ceci change, et ce composant est-il aussi abstrait que l'exige le nombre d'éléments qui dépendent de lui ? La section 3.5 poursuit avec les principes SOLID, qui appliquent les mêmes idées à l'intérieur des composants et entre eux.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Pourquoi tous les composants d'un cycle de dépendances doivent-ils être mis en production ensemble ?**

Chacun dépend, directement ou indirectement, de tous les autres : un changement dans l'un peut donc exiger des changements dans les autres, et aucun ne peut être construit ou testé sans les autres.

▼ **Un composant a Ca = 3 et Ce = 9. Quelle est son instabilité, et que signifie-t-elle ?**

I = 9 / 12 = 0,75. Il dépend de bien plus d'éléments qu'il n'en a qui dépendent de lui : il est donc relativement instable, facile à changer, mais exposé aux changements de ses dépendances.

▼ **Qu'est-ce qui se trouve dans la zone de douleur, et quand cela fait-il vraiment mal ?**

Les composants à la fois stables (beaucoup de dépendants) et concrets (peu d'abstractions). Cela fait mal quand un tel composant doit changer souvent ; un composant concret qui ne change jamais n'y fait guère de tort.

▼ **Quelle est la différence entre organisation en couches stricte et souple ?**

Dans l'organisation stricte, une couche ne peut utiliser que la couche juste en dessous d'elle ; dans l'organisation souple, elle peut utiliser n'importe quelle couche inférieure.

▼ **À vous : Le composant pricing-rules a Ca = 8, Ce = 0 et 12 classes, aucune abstraite. Le composant reports a Ca = 2, Ce = 6 et 10 classes, dont 5 abstraites. Calculez I, A et D pour chacun, et dites lequel vous inquiète le plus.**

pricing-rules : I = 0 / 8 = 0, A = 0, D = |0 + 0 - 1| = 1, en plein dans la zone de douleur : beaucoup de choses en dépendent, et pourtant il est concret, si bien que chaque changement est douloureux. reports : I = 6 / 8 = 0,75, A = 0,5, D = 0,25, raisonnablement près de la séquence principale. C'est pricing-rules qui inquiète. Extrayez des interfaces, afin que les dépendants s'appuient sur des abstractions stables. Des règles qui changent souvent rendent cela urgent.

SOURCES

- Robert C. Martin, OO Design Quality Metrics: An Analysis of Dependencies (1994): https://bccdev.ime.usp.br/tccs/2010/lucianna-joao/arquivos/referencias/oodmetrics.pdf
- ArchUnit User Guide: https://www.archunit.org/userguide/html/000_Index.html
- Import Linter: contract types: https://import-linter.readthedocs.io/en/stable/contract_types/
- dependency-cruiser (GitHub): https://github.com/sverweij/dependency-cruiser
- The Java Language Specification, Chapter 7: Packages and Modules: https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html

Les principes SOLID appliqués aux composants

AVANT DE LIRE

Cette section s'appuie sur :

- 3.2 Couplage et cohésion
- 3.3 Masquage de l'information
- 3.4 Dépendances

Termes déjà vus qu'elle utilise : Conception par contrat, Précondition, Invariant, Loi de Hyrum, Principe des dépendances stables

L'IDÉE EN BREF

Pensez à un groupe de restaurants bien géré. Chaque cuisine rend compte à un seul chef, si bien que deux patrons ne lui donnent jamais d'ordres contradictoires. On ajoute de nouveaux plats à la carte sans reconstruire la cuisine. Un chef remplaçant doit préparer chaque plat exactement comme le chef habituel, sinon les clients s'en aperçoivent. Chaque fournisseur ne reçoit que le bon de commande dont il a besoin, pas tout le manuel de l'entreprise. Enfin, le siège fixe les normes et les fournisseurs s'y adaptent, et non l'inverse. SOLID, ce sont ces cinq habitudes, transposées au logiciel.

CE QU'IL FAUT SAVOIR

- SOLID désigne cinq principes rassemblés par Robert C. Martin : responsabilité unique, ouvert-fermé, substitution de Liskov, ségrégation des interfaces et inversion des dépendances ; chacun s'applique aux classes, aux composants et aux services.
- Responsabilité unique signifie un acteur par module : le code modifié à la demande de groupes de personnes différents doit être séparé, et cela ne signifie pas qu'un module ne peut faire qu'une seule chose.
- La substitution de Liskov exige une compatibilité comportementale : pas de préconditions plus fortes, pas de postconditions plus faibles et des invariants préservés, ce qui est exactement le critère d'une version d'API rétrocompatible.
- La ségrégation des interfaces plaide pour de petites interfaces de rôle dans le code et, entre services, pour des API propres à chaque client, des événements ciblés et de petites bibliothèques.
- L'inversion des dépendances fait dépendre les détails d'abstractions détenues par le côté métier ; savoir qui détient l'interface compte plus que le fait qu'elle existe.

- REP, CCP et CRP décident de ce qui entre dans un composant et tirent dans des sens opposés ; le bon équilibre se déplace de CCP vers REP et CRP à mesure qu'un système mûrit.
- Appliqué de façon dogmatique, SOLID produit des abstractions superficielles et spéculatives ; il fonctionne mieux comme vocabulaire pour diagnostiquer les conceptions qui causent de vrais problèmes.

Cinq principes et leur histoire

SOLID est un moyen mnémotechnique désignant cinq principes de conception que Robert C. Martin a rassemblés et affinés à la fin des années 1990, de la façon la plus complète dans son article de 2000 *Design Principles and Design Patterns*. Michael Feathers a remarqué que leurs initiales pouvaient être réarrangées pour former le mot SOLID, et l'acronyme est resté. Les principes ont été écrits pour les classes orientées objet, mais le livre ultérieur de Martin, *Clean Architecture* (2017), soutient que chacun s'applique aussi aux composants, et les praticiens les appliquent couramment aux services et aux API.

C'est pourquoi ils ont leur place dans un cours d'architecture. Chaque principe est en réalité un énoncé sur **l'endroit où le changement doit atterrir** et **le sens dans lequel les dépendances doivent pointer**, les deux préoccupations des sections précédentes sur le couplage et la cohésion et la gestion des dépendances. Cette section passe en revue chaque principe, le montre en quelques lignes de Python, puis montre ce qu'il signifie lorsque la « classe » est un composant déployable ou un service détenu par une autre équipe. Elle se termine par les principes de cohésion au niveau des composants et par les critiques que suscite SOLID, qui sont substantielles et méritent d'être connues.

SI VOUS DÉBUTEZ

Si vous débutez

Vous avez besoin de trois idées vues dans les sections précédentes. Un **module** est toute unité de code dotée d'une frontière : une fonction, une classe, un paquet, un composant ou un service. Une **interface** est l'ensemble des opérations qu'un module offre aux autres, un contrat qui cache la manière dont le travail est fait. Le **couplage** mesure ce qu'un module doit savoir d'un autre ; plus il en sait, plus un changement d'un côté risque de casser l'autre. Le `Protocol` de Python, utilisé ci-dessous, est simplement une façon de nommer une interface : toute classe dotée des méthodes correspondantes la satisfait.

S : responsabilité unique signifie un seul acteur

Le principe de responsabilité unique est le plus cité et le plus mal compris des cinq. Il ne dit *pas* qu'un module ne doit faire qu'une seule chose ; c'est une bonne règle pour les fonctions, mais c'est une règle différente. La formulation de Martin lui-même a évolué : il disait qu'une classe ne devrait avoir qu'une seule raison de changer, puis, dans *Clean Architecture*, qu'un module devrait être responsable devant un, et un seul, acteur. Un acteur est un groupe de personnes (utilisateurs ou parties prenantes) qui veulent que le système change de la même manière. Le principe porte sur les personnes et sur la direction des changements futurs.

L'illustration classique est une classe `Employee` dotée de trois méthodes : `calculate_pay`, spécifiée par la finance ; `report_hours`, spécifiée par les opérations ; et `save`, spécifiée par les administrateurs de bases de données. Si le calcul de la paie et le rapport d'heures partagent une fonction utilitaire pour calculer les heures normales, une modification demandée par la finance altère silencieusement le rapport sur lequel s'appuient les opérations. Personne n'a été négligent ; le code mélangeait simplement les préoccupations de deux acteurs, si bien qu'un changement demandé par l'un pouvait nuire à l'autre.

```
from dataclasses import dataclass

@dataclass
class EmployeeRecord:          # shared data, no behaviour
    id: str
    hourly_rate: float
    hours: list[float]

class PayCalculator:           # actor: finance
    def pay(self, e: EmployeeRecord) → float:
        regular = sum(min(h, 8) for h in e.hours)
        overtime = sum(max(h - 8, 0) for h in e.hours)
        return (regular + 1.5 * overtime) * e.hourly_rate

class HoursReporter:           # actor: operations
    def weekly_report(self, e: EmployeeRecord) → str:
        return f"{e.id}: {sum(e.hours):.1f} hours"

class EmployeeRepository:      # actor: database administrators
    def save(self, e: EmployeeRecord) → None: ...
```

Découpage par acteur : chaque classe ne change qu'à la demande de ses propres parties prenantes, et les données partagées restent un simple enregistrement.

À l'échelle des composants et des services, le principe devient un test pour les frontières : **qui demande les changements de ce composant ?** Dans une plateforme de livraison de repas, un service « Restaurant » unique qui détient les cartes (modifiées à la demande des restaurateurs), les règles de commission (modifiées par l'équipe financière) et le classement des résultats de

recherche (modifié par l'équipe croissance) sera modifié pour trois raisons sans rapport, par trois groupes de personnes, à trois rythmes de mise en production. Chaque mise en production fait courir un risque aux deux autres préoccupations, et l'équipe propriétaire devient une file d'attente. Découper selon les acteurs, par exemple en un service de carte, un composant de commissions au sein du domaine financier et un service de classement, aligne chaque frontière sur une seule source de changement. C'est aussi l'idée qui sous-tend l'alignement des services sur les capacités métier et, en conception pilotée par le domaine, sur les contextes délimités (traités au chapitre 5).

Notez que le principe ne dit pas « faites des services petits ». Un gros service qui sert bien un seul acteur ne pose pas de problème ; un tout petit service que deux départements ne cessent de modifier dans des directions contradictoires en pose un.

O : ouvert à l'extension, fermé à la modification

Le principe ouvert-fermé a été énoncé pour la première fois par Bertrand Meyer dans *Object-Oriented Software Construction* (1988) : un module doit être ouvert à l'extension (on peut ajouter un nouveau comportement) mais fermé à la modification (son code source existant et testé n'a pas besoin d'être modifié pour cela). Meyer pensait à l'héritage ; Martin l'a réinterprété en termes d'interfaces et de polymorphisme. Sa forme pratique est la suivante : **quand un type de variation est attendu, donnez-lui un point d'extension**, afin que la variante suivante soit du nouveau code plutôt qu'une branche de plus dans l'ancien code.

```
from typing import Protocol

class Carrier(Protocol):
    def quote(self, weight_kg: float) → int: ...   # price in pence

class RoyalMail:
    def quote(self, weight_kg: float) → int:
        return 350 + int(weight_kg * 120)

class Courier:
    def quote(self, weight_kg: float) → int:
        return 600 + int(weight_kg * 80)

CARRIERS: dict[str, Carrier] = {"royal_mail": RoyalMail(), "courier": Courier()}

def cheapest(weight_kg: float) → tuple[str, int]:
    return min(((name, c.quote(weight_kg)) for name, c in CARRIERS.items()),
               key=lambda pair: pair[1])
```

Ajouter un transporteur revient à enregistrer une nouvelle classe ; le code de calcul des devis ne change jamais.

À l'échelle de l'architecture, la même idée apparaît sous la forme de **points d'extension que d'autres équipes peuvent utiliser sans vous demander de modifier votre code**. Un service des commandes qui publie un événement `OrderPlaced` est ouvert à l'extension : l'équipe fidélité peut ajouter un nouveau consommateur qui attribue des points, et le service des commandes n'est ni modifié, ni retesté, ni redéployé. Une architecture à plug-ins (le style micronoyau) fait la même promesse pour les fonctionnalités. Pour les API, le principe se traduit par une évolution additive : de nouveaux champs facultatifs, de nouveaux endpoints et de nouveaux types d'événements étendent un contrat sans modifier ce sur quoi s'appuient les clients existants.

Le piège, c'est la prédiction. Un point d'extension n'a de valeur que pour les types de changement qui surviennent réellement. Un registre de transporteurs vaut la peine si la boutique ajoute des transporteurs tous les quelques mois ; un « moteur de règles de tarification » générique construit parce que la tarification *pourrait* un jour varier est une machinerie spéculative. Les professionnels ajoutent le point d'extension la deuxième ou la troisième fois qu'une variation apparaît, pas avant.

L : substitution de Liskov et contrats comportementaux

Le principe de substitution de Liskov vient de l'exposé principal de Barbara Liskov, *Data Abstraction and Hierarchy*, lors de la conférence OOPSLA de 1987, et a été précisé par Liskov et Jeannette Wing dans *A Behavioral Notion of Subtyping* (ACM Transactions on Programming Languages and Systems, 1994). De manière informelle : si du code fonctionne avec des objets de type T, il doit continuer à fonctionner lorsqu'on lui donne un objet de n'importe quel sous-type S, sans savoir lequel il a reçu. Avoir les bons noms de méthodes ne suffit pas ; le *comportement* doit être compatible.

Les règles du sous-typage comportemental se rattachent directement à la conception par contrat (section 3.3). Un sous-type ne peut **pas renforcer les préconditions** (exiger davantage de l'appelant), ne peut **pas affaiblir les postconditions** (promettre moins en retour) et doit **préserver les invariants** (les faits toujours vrais pour le type). Liskov et Wing ont ajouté une contrainte d'historique : un sous-type ne doit pas permettre des changements d'état que le supertype interdit, ce qui explique pourquoi un point mutable ne peut pas se faire passer pour un point immuable.

```
class Discount:
    """Postcondition: 0 ⩽ result ⩽ total."""
    def apply(self, total: int) → int:
        return total

class PercentOff(Discount):
    def __init__(self, pct: int):
        self.pct = pct
    def apply(self, total: int) → int:
        return total - total * self.pct // 100      # keeps the contract

class FixedVoucher(Discount):
    def __init__(self, pence: int):
        self.pence = pence
    def apply(self, total: int) → int:
        return total - self.pence                    # can go below zero!

# FixedVoucher(500).apply(300) == -200: the checkout now issues a refund.
# Fix: return max(total - self.pence, 0), honouring the base postcondition.
```

Le contrat de base dit qu'une remise ne rend jamais le total négatif. Un sous-type qui le rompt met en échec les appelants qui faisaient confiance au contrat.

Le symptôme d'une violation du LSP est la vérification de type chez les appelants : `if isinstance(d, FixedVoucher): ...`. Chaque vérification de ce genre signifie qu'une abstraction n'est plus digne de confiance, et la connaissance de ses exceptions se répand dans toute la base de code.

À l'échelle des services, la substituabilité est la question qui se cache derrière chaque **changement d'API compatible**. Si la version 2.3 d'une API de paiement doit remplacer la 2.2 sans que les clients s'en aperçoivent, elle est un sous-type comportemental de la 2.2 et les mêmes règles s'appliquent. Elle ne doit pas renforcer les préconditions : pas de nouveau champ obligatoire dans la requête, pas de validation plus stricte des champs existants, pas de limite de débit plus basse. Elle ne doit pas affaiblir les postconditions : pas de champs de réponse supprimés, pas de temps de réponse garanti plus lent, pas de perte d'idempotence. Elle doit préserver les invariants : un montant est toujours exprimé en unités mineures, un identifiant de commande, comme avant, ne change jamais. Ajouter une nouvelle valeur à une énumération dans une *réponse* est un cas subtil : cela élargit ce que le client peut recevoir, ce qui casse les clients qui supposaient l'ancien ensemble complet. Le même raisonnement s'applique lorsqu'une implémentation d'un service en remplace une autre derrière un équilibreur de charge, ou lorsque vous remplacez l'adaptateur d'un prestataire de paiement par celui d'un autre. Les tests de contrats pilotés par le consommateur sont, en pratique, des vérifications de Liskov automatisées, et la loi de Hyrum (section 3.3) rappelle que les clients peuvent s'appuyer sur un comportement que vous n'avez jamais promis.

I : ségrégation des interfaces et API propres à chaque client

Le principe de ségrégation des interfaces dit qu'aucun client ne doit être forcé de dépendre d'opérations qu'il n'utilise pas. Une interface trop large couple chaque client à chaque opération : quand une opération change, tous les clients sont touchés, ne serait-ce que parce qu'ils doivent être recompilés, retestés ou redéployés, même ceux qui ne l'appellent jamais. Dans les langages compilés, ce couplage est littéral (une modification d'un fichier d'en-tête ou d'un JAR partagé oblige à reconstruire tout ce qui en dépend) ; en Python, il se manifeste par des doublures de test qui doivent implémenter vingt méthodes pour tester du code qui en appelle deux.

```
from typing import Protocol

class OrderReader(Protocol):
    def get(self, order_id: str) → dict: ...
    def list_for_customer(self, customer_id: str) → list[dict]: ...

class OrderWriter(Protocol):
    def save(self, order: dict) → None: ...
    def cancel(self, order_id: str) → None: ...

def monthly_statement(orders: OrderReader, customer_id: str) → int:
    return sum(o["total"] for o in orders.list_for_customer(customer_id))

# The reporting job cannot cancel an order, and its tests
# need a fake with two read methods, not the whole repository.
```

Chaque client ne dépend que du rôle dont il a besoin ; une même classe peut toujours implémenter les deux rôles.

À l'échelle des services, la ségrégation des interfaces est l'argument en faveur des **API propres à chaque client**. Une API unique et généraliste qui sert l'application mobile, la boutique en ligne, le système d'entrepôt et les intégrations des partenaires tend à devenir une conception au plus petit dénominateur commun : chaque client reçoit des champs qu'il ignore, et chaque changement doit être négocié avec tous. Le patron backend for frontend (nommé chez SoundCloud et décrit par Phil Calçado et Sam Newman) donne à chaque type de client sa propre couche d'API légère, détenue par l'équipe qui construit ce client, et qui compose les appels aux services sous-jacents. GraphQL s'attaque au même problème autrement, en laissant chaque client sélectionner exactement les champs dont il a besoin. Pour les événements, le principe suggère de publier plusieurs types d'événements ciblés (`OrderPlaced`, `OrderShipped`) plutôt qu'un seul événement `OrderChanged` contenant tous les champs, qui force chaque consommateur à tout analyser et filtrer.

La version architecturale s'applique aussi aux **bibliothèques partagées**. Une bibliothèque `common` commune à toute l'entreprise, qui contient la journalisation, la gestion des montants, la génération de PDF et des clients HTTP, couple chaque service à chacune de ses parties : un correctif de sécurité dans le code PDF force une nouvelle version de services qui ne génèrent jamais de PDF. La découper en petites bibliothèques que les clients adoptent indépendamment, c'est appliquer la ségrégation des interfaces au niveau des paquets, et cela anticipe le principe de réutilisation commune présenté plus bas.

D : l'inversion des dépendances et qui détient l'interface

Le principe d'inversion des dépendances comporte deux parties : les modules de haut niveau (les règles métier) ne doivent pas dépendre des modules de bas niveau (les détails comme les bases de données, les courtiers de messages et les frameworks web) ; les deux doivent dépendre d'abstractions. Et les abstractions ne doivent pas dépendre des détails. Le mot « inversion » désigne le sens de la dépendance dans le code source : dans une conception naïve, les règles métier appellent le détail et donc l'importent ; après l'inversion, le détail importe une interface que les règles métier définissent, si bien que la flèche du code source pointe dans le sens opposé au flux de contrôle.

Le point crucial, et souvent négligé, est la **propriété**. L'interface appartient au composant de haut niveau et s'exprime dans ses termes (`OrderRepository.save(order)` , et non `PostgresClient.execute(sql)`). Si l'interface se trouve dans le paquet de persistance, vous avez ajouté une abstraction mais vous n'avez rien inversé : les règles métier dépendent toujours du paquet de persistance.

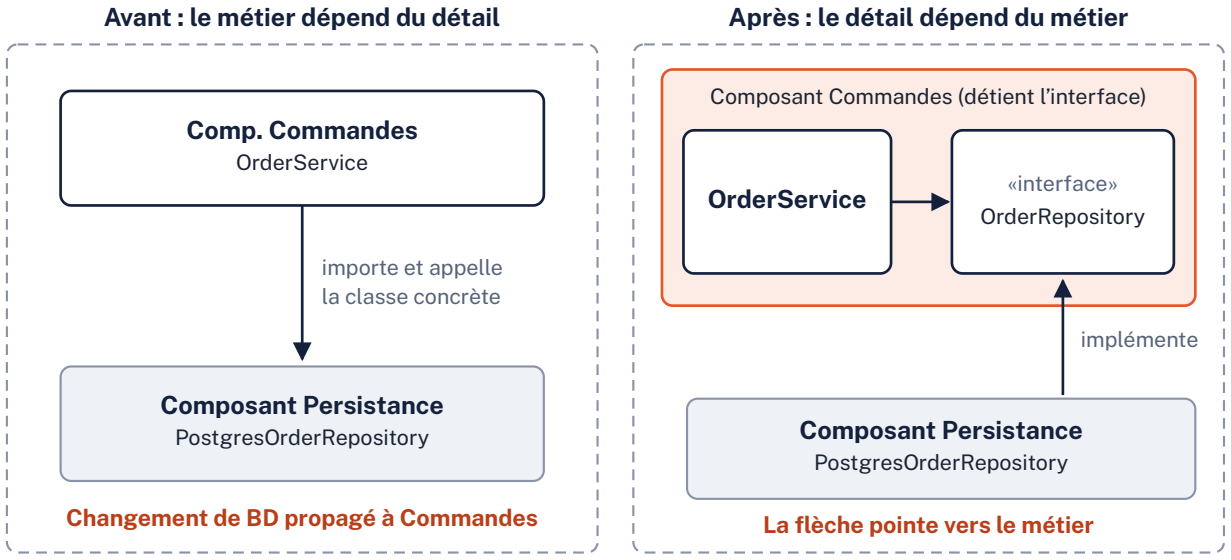
```
# orders/ports.py  (high-level policy owns this)
from typing import Protocol
class PaymentPort(Protocol):
    def charge(self, order_id: str, pence: int) → str: ...  # returns payment id

# orders/service.py
class Checkout:
    def __init__(self, payments: PaymentPort):
        self.payments = payments
    def pay(self, order_id: str, pence: int) → str:
        return self.payments.charge(order_id, pence)

# infrastructure/stripe_adapter.py  (detail depends on policy)
from orders.ports import PaymentPort
class StripeAdapter(PaymentPort):
    def charge(self, order_id: str, pence: int) → str:
        ...  # call the provider's API, map its errors to ours
```

Le paquet orders définit le port dont il a besoin ; l'infrastructure l'implémente et y est branchée au démarrage.

Inverser une dépendance à travers une frontière de composant



Placer l'interface dans le composant Commandes inverse la dépendance du code source, si bien que la persistance peut changer sans toucher aux règles métier.

L'inversion des dépendances n'est pas la même chose que l'injection de dépendances, même si elles vont souvent de pair. L'injection est une technique pour fournir à un module ses collaborateurs depuis l'extérieur (l'argument de constructeur `payments` ci-dessus) ; l'inversion est une décision sur le sens dans lequel pointent les dépendances du code source. Vous pouvez injecter une classe concrète sans rien inverser.

À l'échelle de l'architecture, l'inversion des dépendances est la règle qui organise les architectures hexagonale, en oignon et Clean Architecture : le domaine est au centre et définit des ports ; les adaptateurs pour les bases de données, les files d'attente et les interfaces utilisateur sont à l'extérieur et dépendent vers l'intérieur. Entre services, la même question se pose sous la forme « qui détient le contrat ? ». Si un service central de facturation définit un schéma `BillingEvent` et que chaque équipe produit doit s'y adapter, la facturation incarne les règles stables et les produits en dépendent, ce qui est souvent juste. Si au contraire la facturation lit les tables internes de chaque produit, la dépendance va du stable vers le volatil, et chaque modification de schéma d'un produit casse la facturation. La section 3.4 a montré comment quantifier cela avec le principe des dépendances stables.

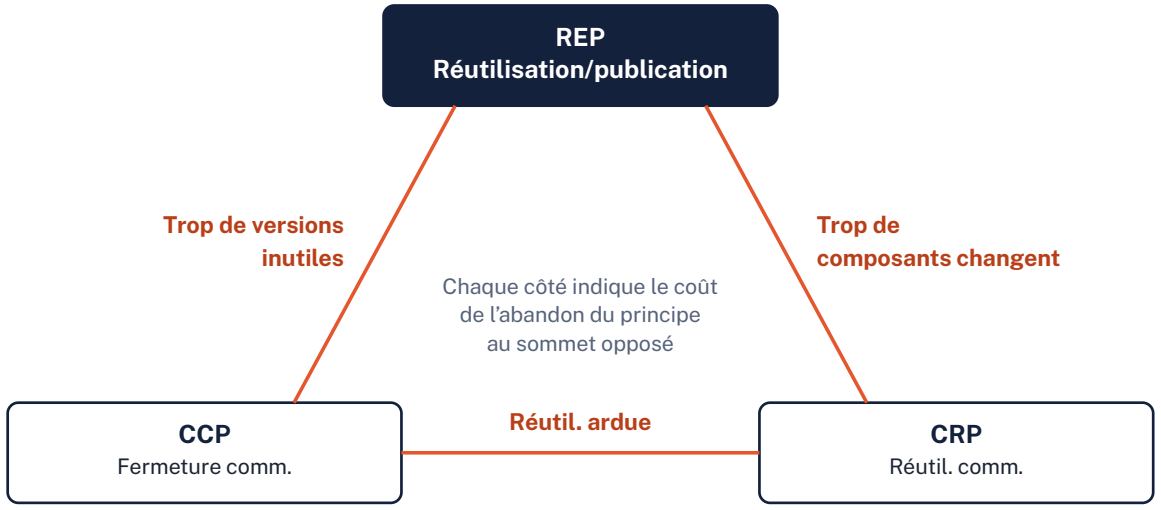
Les principes de cohésion des composants : REP, CCP et CRP

SOLID vous indique comment façonner les classes et leurs dépendances. Martin l'associe à trois principes pour décider **quelles classes vont ensemble dans un même composant**, un composant étant une unité de publication et de déploiement comme un JAR, un paquet Python ou un assembly .NET.

Le principe d'équivalence réutilisation/publication (REP) dit que l'unité de réutilisation est l'unité de publication : si d'autres doivent réutiliser un composant, il lui faut des numéros de version, des notes de version et un thème cohérent, donc les classes qu'il contient doivent avoir un sens à être publiées ensemble. Le principe de fermeture commune (CCP) est le principe de responsabilité unique appliqué aux composants : regroupez dans un même composant les classes qui changent pour les mêmes raisons et aux mêmes moments, afin qu'un changement typique ne touche qu'un composant et que seul celui-ci doive être publié. Le principe de réutilisation commune (CRP) est le principe de ségrégation des interfaces appliqué aux composants : les classes utilisées ensemble vont ensemble, et celles qui ne le sont pas ne doivent pas être empaquetées ensemble, car dépendre d'un composant, c'est dépendre de tout son contenu.

Les trois tirent dans des sens opposés. REP et CCP sont *inclusifs* (ils font grossir les composants) ; CRP est *exclusif* (il les fait rapetisser). Martin les représente par un triangle dont chaque côté indique le prix à payer quand on abandonne le principe situé au sommet opposé.

Le triangle des tensions



Privilégier deux principes de cohésion, quels qu'ils soient, entraîne le coût inscrit sur le côté qui les relie ; les architectes choisissent une position, et cette position évolue à mesure que le projet mûrit.

Martin observe que la position d'un projet dans le triangle devrait évoluer dans le temps. Au début, la facilité de développement compte le plus, donc les équipes privilégient CCP : regrouper par changement, en acceptant que les composants soient difficiles à réutiliser. À mesure qu'un système mûrit et que d'autres équipes commencent à dépendre de ses parties, l'équilibre se déplace vers REP et CRP. La même tension apparaît entre les services. Fusionner deux services qui changent toujours ensemble (CCP) réduit les mises en production

coordonnées mais rend le résultat moins réutilisable ; découper un service pour que les consommateurs ne dépendent que de ce qu'ils utilisent (CRP) augmente le nombre d'unités déployables que touche un seul changement métier.

EXEMPLE COMMENTÉ

Exemple commenté : revue d'un composant partagé dans une boutique en ligne

Une boutique en ligne a un paquet `shop-core` utilisé par cinq services. Il contient des règles de tarification, le calcul des taxes, la validation des adresses, le rendu des factures PDF et un utilitaire de base de données. Au cours du dernier trimestre, il a connu 40 versions. Appliquez les principes.

Acteurs (SRP, CCP)	La tarification est modifiée par l'équipe commerciale, les taxes par la finance quand les taux changent, la mise en page des PDF par le marketing, l'utilitaire de base de données par l'équipe plateforme. Quatre acteurs, c'est quatre raisons indépendantes de changer : le paquet ne respecte donc pas la fermeture commune.
Utilisateurs (ISP, CRP)	Le service d'entrepôt n'utilise que la validation des adresses, et pourtant chacune des 40 versions l'a obligé à se mettre à jour et à refaire ses tests. Selon CRP, environ 40 versions, moins les quelques-unes qui touchaient aux adresses, lui étaient inutiles.
Versions (REP)	Les notes de version n'ont aucun thème cohérent, si bien que les consommateurs ne peuvent pas juger si une version les concerne.
Dépendances (DIP, SDP)	L'utilitaire de base de données importe les modèles ORM des services, si bien qu'un paquet stable et largement utilisé dépend de code de service volatil : la flèche pointe dans le mauvais sens.
Découpage proposé	`pricing` (détenu par le domaine commercial), `tax` (finance), `address`, `invoice-pdf`, et l'utilitaire de base de données déplacé dans chaque service ou dans une bibliothèque de plateforme sans aucun import de domaine.
Effet attendu	Chaque service ne dépend que des paquets qu'il utilise. Un changement de taux de taxe ne publie qu'un seul paquet et ne touche que les services qui calculent la taxe.

Les principes n'ont pas dicté la réponse ; ils ont fourni un vocabulaire pour expliquer pourquoi le paquet actuel pose problème et ce que coûte chaque option.

Le découpage augmente le nombre de paquets à versionner et à publier : c'est le côté « trop de composants changent » du triangle. C'est acceptable ici, car les changements sont pour la plupart indépendants.

Critiques et limites

SOLID est largement enseigné et largement critiqué, et les deux réactions ont leurs mérites. Voici les critiques les plus sérieuses.

Les principes sont assez vagues pour justifier presque n'importe quoi. Dan North a défendu cette thèse dans des conférences et dans son essai de 2022 présentant CUPID (composable, Unix philosophy, predictable, idiomatic, domain-based). Il soutient que « une seule raison de changer » est arbitraire, puisque l'on peut dire de presque n'importe quel code qu'il change pour de nombreuses raisons, et que l'appliquer avec zèle crée des coutures artificielles entre du code qui change en réalité ensemble, comme le contenu d'un rapport et son format. Il propose des propriétés à viser plutôt que des règles à suivre. Martin a répondu à cette ligne de critique dans un billet de 2020 sur la pertinence de SOLID, en soutenant que les principes restent valables parce que les problèmes sous-jacents de gestion des dépendances n'ont pas changé, et que dire aux gens de « faire simple » sans les guider produit un fouillis.

L'application dogmatique produit une sur-abstraction. Une interface pour chaque classe, une fabrique pour chaque interface et un point d'extension pour chaque variation imaginable mènent à du code où suivre un seul comportement oblige à ouvrir dix fichiers. La distinction de John Ousterhout entre modules profonds et modules superficiels (section 3.3) offre un contrepoids utile : une abstraction devrait cacher beaucoup derrière une petite interface, et beaucoup de classes inspirées de SOLID sont superficielles. Les points d'extension spéculatifs violent YAGNI (section 3.8) et ont un coût réel.

Les principes ont été écrits pour un seul paradigme. Dans les langages dotés de fonctions de première classe et d'un typage structurel, de nombreux patrons SOLID se réduisent à passer une fonction ou à s'appuyer sur le duck typing. Les idées sous-jacentes (séparer par acteur, dépendre de contrats stables, garder les interfaces petites) restent valables ; le cérémonial, souvent, non.

Le point de vue professionnel équilibré consiste à traiter SOLID comme un **vocabulaire de diagnostic** : quand une conception pose problème, les principes aident à expliquer pourquoi et suggèrent une direction. Ce n'est pas une liste de contrôle que chaque classe doit satisfaire, et « viole l'OCP » n'est pas en soi un argument ; « chaque nouveau transporteur a exigé de modifier cette fonction et a causé deux incidents » en est un.

GUIDE DE DÉCISION

Guide de décision : dois-je ajouter une abstraction ici ?

Utilisez-le quand un principe semble réclamer une nouvelle interface, un point d'extension ou un découpage.

Q1 **Ce type de variation s'est-il réellement produit au moins deux fois, ou est-il certain (par exemple, un contrat avec un second prestataire est signé) ?**

- **Oui** Un point d'extension est probablement justifié ; continuez.
- **Non** Gardez le code concret, faites en sorte qu'il reste facile à refactoriser, et attendez des preuves réelles (YAGNI).

Q2 **Les parties que vous voulez séparer sont-elles modifiées par des acteurs différents ou selon des calendriers différents ?**

- **Oui** Séparez-les (SRP, CCP) ; la frontière réduira la coordination.
- **Non** Gardez-les ensemble ; découper ajouterait des coutures que chaque changement devrait franchir.

Q3 **Des règles métier de haut niveau importent-elles actuellement un détail volatil, comme une base de données, le SDK d'un fournisseur ou un framework ?**

- **Oui** Définissez une interface détenue par le côté métier et implémentez-la dans le détail (DIP).
- **Non** Aucune inversion n'est nécessaire ; une dépendance directe envers un module stable ne pose pas de problème.

Q4 **La nouvelle interface cachera-t-elle une complexité importante derrière quelques opérations ?**

- **Oui** C'est un module profond ; il mérite d'exister.
- **Non** C'est un simple relais superficiel ; reconsidérez-le, ou fusionnez-le avec son appelant.

SOLID à trois échelles

Principe	Échelle de la classe	Échelle du composant ou du service
Responsabilité unique	Un acteur par classe	Un service ou un composant aligné sur une capacité métier et une équipe propriétaire
Ouvert-fermé	Un nouveau comportement grâce à de nouvelles implémentations d'une interface	Des événements, des plug-ins et des changements d'API additifs qui permettent aux autres d'étendre sans vous modifier
Substitution de Liskov	Les sous-classes respectent le contrat de la classe de base	Les nouvelles versions d'API et les implémentations de remplacement respectent l'ancien contrat : pas de nouvelles entrées obligatoires, pas de sorties supprimées
Ségrégation des interfaces	De petites interfaces de rôle	Des API propres à chaque client (backend for frontend), des types d'événements ciblés, de petites bibliothèques partagées
Inversion des dépendances	Dépendre d'abstractions que vous détenez	Le domaine définit des ports ; les adaptateurs et l'infrastructure dépendent vers l'intérieur ; les propriétaires stables définissent les contrats

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Une classe met en forme une facture et calcule aussi sa taxe. Cela viole-t-il automatiquement le principe de responsabilité unique ?**

Pas automatiquement. C'est le cas si la mise en forme et les règles fiscales sont modifiées par des acteurs différents (par exemple le marketing et la finance), car alors les changements demandés par l'un peuvent casser l'autre ; si un seul groupe détient les deux, la classe peut très bien convenir.

▼ **Pourquoi l'ajout d'un nouveau champ obligatoire à une requête d'API rompt-il la substituabilité ?**

Il renforce la précondition : les appelants qui respectaient l'ancien contrat échouent désormais, si bien que la nouvelle version ne peut plus remplacer l'ancienne.

▼ **Quelle est la différence entre l'inversion des dépendances et l'injection de dépendances ?**

L'inversion concerne le sens des dépendances du code source, le côté métier détenant l'abstraction ; l'injection est une technique pour fournir des collaborateurs depuis l'extérieur. Vous pouvez injecter une classe concrète sans rien inverser.

▼ **Une équipe regroupe les classes strictement selon ce qui change ensemble, et d'autres équipes se plaignent que les composants sont impossibles à réutiliser. Sur quel côté du triangle des tensions se trouve-t-elle ?**

Le côté CCP-CRP, qui porte la mention difficile à réutiliser : elle a négligé le principe d'équivalence réutilisation/publication.

▼ **À vous : La version 2 d'une API interne de remboursement rejette désormais les montants supérieurs à £100, que la version 1 acceptait, et ne renvoie plus le champ reason. L'équipe la dit compatible parce que l'URL n'a pas changé. Servez-vous de la substitution de Liskov pour en juger.**

Elle n'est pas compatible. La substitution de Liskov demande si la nouvelle version peut remplacer l'ancienne sans que les appelants s'en aperçoivent. Rejeter les montants supérieurs à £100 renforce une précondition, et supprimer le champ de motif affaiblit une postcondition ; chacun de ces changements peut casser des appelants existants. Une URL inchangée ne prouve rien. Soit vous conservez l'ancien comportement en ne faisant que des ajouts, soit vous publiez une nouvelle version et faites migrer les appelants, la limite de £100 ne s'appliquant qu'à ceux qui l'acceptent.

SOURCES

- Robert C. Martin: The Single Responsibility Principle (Clean Coder blog): https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
- Robert C. Martin: Solid Relevance (Clean Coder blog, 2020): https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html
- Barbara Liskov and Jeannette Wing: A Behavioral Notion of Subtyping (1994): https://www.cs.columbia.edu/~wing/publications/LiskovWing94.pdf
- Dan North: CUPID, for joyful coding (2022): https://dannorth.net/blog/cupid-for-joyful-coding/
- Sam Newman: Backends for Frontends pattern: https://samnewman.io/patterns/architectural/bff/

SECTION 3.6 SUR 3.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

La connascence : une mesure plus fine du couplage

AVANT DE LIRE

Cette section s'appuie sur :

- 3.2 Couplage et cohésion
- 3.3 Masquage de l'information

Termes déjà vus qu'elle utilise : Couplage, Encapsulation, Couplage sémantique, Couplage temporel, Invariant

L'IDÉE EN BREF

Quand deux amis conviennent de se retrouver « au même endroit et à la même heure que d'habitude », ils sont liés par un accord que personne d'autre ne peut voir ; si l'un d'eux change ses plans, l'autre doit le savoir. Le logiciel est plein d'accords de ce genre : des noms, l'ordre des valeurs, ce qu'un nombre signifie en secret, l'étape qui doit avoir lieu en premier. La connascence est une façon de recenser ces accords, de juger du risque de chacun et de rendre les plus risqués plus faibles, plus visibles ou plus rapprochés.

CE QU'IL FAUT SAVOIR

- Deux éléments sont connascents si une modification de l'un exige une modification de l'autre pour que le système reste correct ; Meilir Page-Jones a introduit cette idée en 1992.
- Les formes statiques (nom, type, signification, position, algorithme) sont visibles dans le code source ; les formes dynamiques (exécution, timing, valeur, identité) n'apparaissent qu'à l'exécution et sont généralement plus fortes.
- La force mesure la difficulté à trouver et à modifier une forme, la localité la distance entre les éléments, et le degré le nombre d'éléments qui partagent l'accord.
- Convertissez les formes fortes en formes plus faibles, gardez locale toute connascence forte, et réduisez le degré en faisant passer une convention par une seule définition ou un seul propriétaire.
- Remplacer les paramètres positionnels par des paramètres nommés, et les valeurs magiques par des énumérations nommées, transforme des accords forts et invisibles en accords faibles que les outils peuvent vérifier.

> - Entre services, seules les formes faibles, comme le nom et le type dans un schéma publié, sont saines ; une connascence de valeur ou d'identité entre services signale une frontière mal placée.

Pourquoi le couplage a besoin d'un vocabulaire plus fin

La section 3.2 a décrit le couplage à l'aide des niveaux de la conception structurée (contenu, commun, externe, contrôle, estampille, données) et des formes distribuées qui comptent entre services. Ces niveaux classent la *manière* dont les modules sont reliés. Ils en disent moins sur deux questions que les ingénieurs se posent chaque jour : **à quel point un changement sera-t-il douloureux**, et **quelle refactorisation réduirait la douleur** ? La connascence répond aux deux.

La connascence (du latin, « né ensemble ») a été introduite par Meilir Page-Jones dans un article de 1992 des *Communications of the ACM*, « Comparing techniques by means of encapsulation and connascence », puis développée dans son livre *What Every Programmer Should Know About Object-Oriented Design* (Dorset House, 1995), remanié plus tard sous le titre *Fundamentals of Object-Oriented Design in UML* (1999). Sa définition est simple : deux éléments sont connascents si une modification de l'un exigerait une modification de l'autre pour que le système reste correct. Les éléments peuvent être n'importe quoi : deux fonctions, une fonction et ses appelants, un producteur et les consommateurs de ses messages, une colonne de base de données et chaque rapport qui la lit.

Deux caractéristiques rendent l'idée utile aux architectes. D'abord, la connascence se présente sous des **formes** nommées que l'on peut classer par force, si bien que l'on peut dire non seulement « ces éléments sont couplés », mais « ils sont couplés par position, ce qui est pire qu'un couplage par nom, et voici comment convertir l'un en l'autre ». Ensuite, elle possède deux dimensions supplémentaires, la **localité** et le **degré**, qui indiquent si une forme donnée est acceptable là où elle se trouve.

SI VOUS DÉBUTEZ

Si vous débutez

Chaque fois que deux morceaux de code doivent s'accorder sur quelque chose, ils sont connascents. Si une fonction en appelle une autre par son nom, les deux doivent s'accorder sur ce nom : renommez l'une et vous devez renommer l'autre. C'est inévitable et sans danger. Le danger réside dans les accords difficiles à voir (un nombre qui signifie en secret « expédié »), difficiles à vérifier (deux services qui doivent s'exécuter dans un ordre précis) ou dispersés en de nombreux endroits. La connascence est une façon de nommer ces accords cachés pour pouvoir les rendre visibles, plus faibles ou plus rapprochés.

Formes statiques : visibles dans le code

La connascence statique peut être repérée en lisant le code source, souvent à l'aide d'outils, sans exécuter le programme. De la plus faible à la plus forte, dans l'ordre le plus souvent donné :

Connascence de nom : les éléments doivent s'accorder sur un nom. Un appelant utilise `calculate_tax` ; un consommateur JSON lit `customer_id`. C'est la forme la plus faible, car le renommage est mécanique et les outils trouvent chaque utilisation.

Connascence de type : les éléments doivent s'accorder sur un type. Une fonction attend un nombre entier de pence ; lui passer un flottant fonctionne en Python mais corrompt le résultat. Les vérificateurs de types et les schémas la rendent visible.

Connascence de signification (ou de convention) : les éléments doivent s'accorder sur ce que signifie une valeur. Le statut `3` signifie « expédié » ; `-1` signifie « introuvable » ; une chaîne vide signifie « utiliser la valeur par défaut ». De telles valeurs magiques sont invisibles pour les outils : rien ne relie le `3` du code de l'entrepôt au `3` du service d'e-mail.

Connascence de position : les éléments doivent s'accorder sur l'ordre des valeurs. `create_user("Ann", "Smith", "ann@example.com")` dépend de l'ordre des paramètres ; un fichier CSV dépend de l'ordre des colonnes ; un tuple `(lat, lon)` casse silencieusement si quelqu'un suppose `(lon, lat)`, comme le fait GeoJSON.

Connascence d'algorithme : les éléments doivent s'accorder sur un algorithme. L'émetteur et le récepteur d'un webhook doivent calculer la même signature HMAC ; un client et un serveur doivent appliquer le même schéma de hachage des mots de passe ou la même somme de contrôle ; chaque service qui route par client doit utiliser la même fonction de hachage pour choisir un shard.

Formes dynamiques : visibles seulement à l'exécution

La connascence dynamique dépend du comportement du système à l'exécution : elle est donc plus difficile à repérer et généralement plus forte.

Connascence d'exécution : l'ordre d'exécution compte. Une connexion doit être ouverte avant l'envoi des données ; `begin()` doit précéder `commit()` ; une commande doit être créée avant que son paiement soit encaissé.

Connascence de timing : le moment de l'exécution compte. Le délai d'expiration d'un client doit être plus long que le temps de traitement normal du serveur ; un test qui attend deux secondes espère qu'une tâche asynchrone est terminée ; deux threads se font la course pour mettre à jour un solde.

Connascence de valeur : plusieurs valeurs doivent changer ensemble, généralement à cause d'un invariant. La liste des codes de pays valides dans le frontend web et dans le backend doit concorder ; le total d'une commande doit être égal à la somme de ses lignes ; les débits et les crédits d'une écriture comptable doivent s'équilibrer.

Connascence d'identité : les éléments doivent faire référence à la même entité, et pas seulement à une entité égale. Deux dépôts doivent partager la même transaction de base de données pour que leurs écritures soient atomiques ; deux composants doivent publier dans la *même* instance de file d'attente, et non dans deux files ayant la même configuration.

Page-Jones a aussi nommé la relation opposée, la **contranascence** : des éléments qui doivent *différer*, comme deux classes d'un même espace de noms qui ne doivent pas porter le même nom, ou deux services qui ne doivent pas utiliser le même port. Elle obéit aux mêmes règles.

Les neuf formes, avec un exemple de chacune

Forme	Nature	Exemple dans une application de livraison de repas	Façon habituelle de l'affaiblir
Nom	Statique	Le service des commandes appelle `restaurant.get_menu()`	Déjà la forme la plus faible ; gardez des noms parlants
Type	Statique	Frais de livraison transmis en pence entiers, et non en livres sous forme de flottant	Annotations de type, schémas, un type `Money`
Signification	Statique	`status == 4` signifie « livreur affecté »	Constantes nommées ou énumération, partagées via une seule définition
Position	Statique	`quote(2.5, 51.5, -0.12, True)`	Arguments nommés, objet paramètre, champs JSON nommés
Algorithme	Statique	L'app et le serveur calculent la même somme de contrôle du panier	Une bibliothèque partagée unique, ou laisser un seul côté calculer et envoyer le résultat
Exécution	Dynamique	Le paiement doit être autorisé avant que le restaurant soit notifié	Encapsuler la séquence en un seul endroit (un workflow ou une saga)
Timing	Dynamique	L'app du livreur abandonne au bout de 5 s alors que l'affectation prend jusqu'à 8 s	Rendre l'interaction asynchrone ; dériver les délais d'expiration des SLO
Valeur	Dynamique	Le frontend et le backend détiennent chacun la liste des zones de livraison	Un seul propriétaire des données ; les autres l'interrogent ou s'y abonnent
Identité	Dynamique	Deux modules doivent écrire dans la même transaction de base de données	Confier à un seul module la responsabilité de l'unité de travail

Force, localité et degré

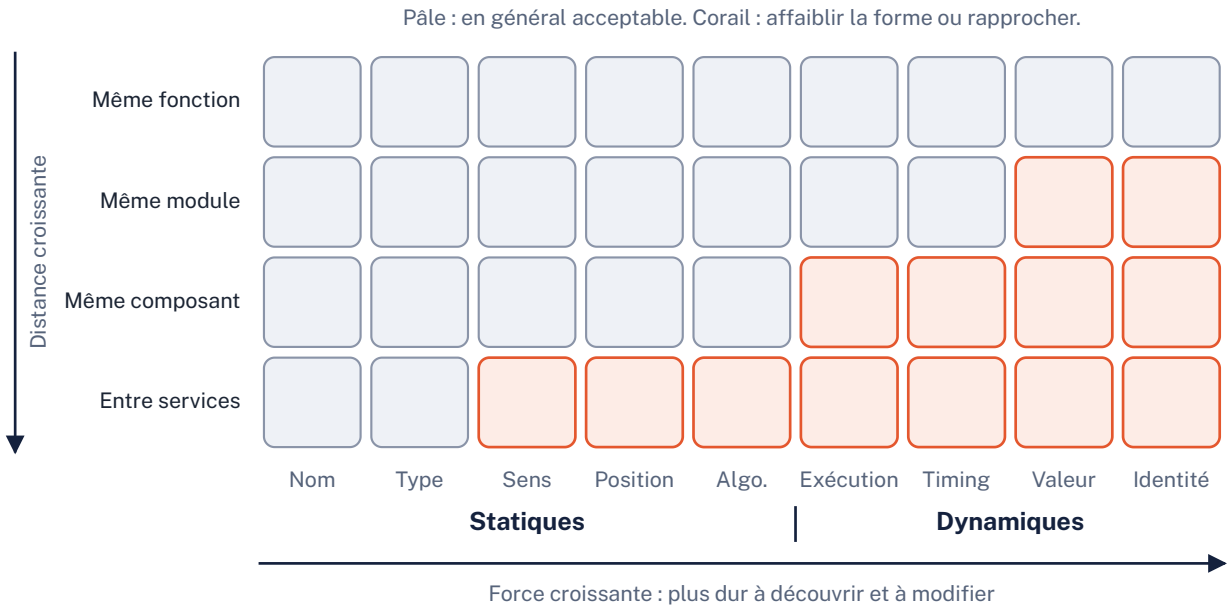
Chaque occurrence de connascence peut être jugée sur trois propriétés. La force mesure la difficulté à découvrir et à modifier la forme : le nom est faible, car les outils le trouvent et le renomment ; la signification est plus forte, car rien ne signale les valeurs magiques ; le timing

et l'identité sont les plus fortes, car elles peuvent n'apparaître que sous charge ou en production. L'ordre du tableau est le classement conventionnel, même si les auteurs divergent légèrement sur les formes statiques (certains placent la position sous la signification).

La localité mesure la proximité des éléments connascents. Des paramètres positionnels entre deux fonctions utilitaires privées d'un même fichier sont sans danger : la personne qui modifie l'une voit l'autre. Des champs positionnels dans un message échangé entre services détenus par des équipes différentes sont dangereux, car les personnes qui doivent changer ensemble ne travaillent pas ensemble, ne publient pas ensemble et peuvent même ignorer l'existence les unes des autres.

Le degré mesure le nombre d'éléments concernés. Un nombre magique partagé par deux fonctions est une gêne ; le même nombre utilisé par quarante fichiers dans six services est un projet de migration. Le degré peut se compter : si le code de statut `3` apparaît à 14 endroits dans le service d'entrepôt, à 9 dans le service de notifications et à 5 dans les vues de la base de données de reporting, le degré est de 28, et changer sa signification implique de coordonner 28 modifications entre trois équipes et au moins deux déploiements, avec une migration de données pour les vues.

La force doit diminuer quand la distance augmente



Les formes fortes de connascence sont tolérables au sein d'une fonction ou d'une classe ; à travers les frontières des composants et des services, visez les formes les plus faibles.

Les règles

Les conseils de Page-Jones peuvent se résumer en trois énoncés : minimiser la connascence globale en découpant un système en éléments encapsulés ; minimiser la connascence qui franchit les frontières d'encapsulation ; et maximiser la connascence à l'intérieur de ces frontières. Le dernier surprend, mais ce n'est que la cohésion reformulée : ce qui doit changer ensemble doit vivre ensemble.

Jim Weirich, développeur Ruby surtout connu comme l'auteur de l'outil de build Rake, a fait redécouvrir l'idée à une nouvelle génération dans des conférences données à partir de 2009, dont « The Building Blocks of Modularity » (présentée lors de plusieurs conférences Ruby en 2009, parmi lesquelles MountainWest RubyConf), qu'il appelait en plaisantant une grande théorie unifiée du développement logiciel, et « Connascence Examined ». Il a condensé la pratique en deux règles. La **règle de localité** : plus deux éléments sont éloignés, plus la connascence entre eux doit être faible. La **règle du degré** : convertissez les formes fortes de connascence en formes plus faibles, surtout lorsque de nombreux éléments sont concernés.

En pratique, cela donne trois actions, à peu près par ordre de préférence :

Affaiblir la forme. Remplacer la signification par le nom (une constante nommée), la position par le nom (arguments nommés, champs nommés), l'algorithme par le nom (une seule fonction partagée que les deux côtés appellent), l'exécution par l'encapsulation (une seule méthode qui effectue les étapes dans le bon ordre).

Augmenter la localité. Si deux éléments doivent partager une forme forte, rapprochez-les : dans la même classe, le même module, le même composant ou le même service, détenu par la même équipe.

Réduire le degré. Si de nombreux éléments partagent une convention, faites-les passer par une seule définition, un seul propriétaire ou un seul adaptateur, afin que seuls quelques endroits connaissent le détail.

EXEMPLE COMMENTÉ

Exemple commenté : remplacer des paramètres positionnels et des valeurs magiques

Une application de VTC possède une fonction utilisée par l'API de réservation, le planificateur et un outil d'administration. Sa signature a grossi en deux ans : `book(rider_id, 51.501, -0.142, 51.515, -0.072, 2, True, 0)`. Analysez la connascence et refactorisez.

Position	Huit arguments positionnels. Intervertir les coordonnées de prise en charge et de dépose passe toujours la vérification de types et envoie silencieusement le chauffeur au mauvais endroit.
Signification	`2` est la catégorie de véhicule (1 standard, 2 grand, 3 accessible) ; `True` signifie « payer par carte » ; `0` signifie « réserver maintenant », tandis que tout autre nombre est une heure programmée en secondes depuis l'epoch.
Degré	Une recherche trouve 31 sites d'appel dans trois composants : chaque nouveau paramètre implique donc 31 modifications à une position fixe.
Étape 1 : le nom plutôt que la position	Introduire un type `Location` et des arguments uniquement nommés, afin que les appelants écrivent `pickup=Location(lat=51.501, lon=-0.142)`.
Étape 2 : le nom plutôt que la signification	Remplacer `2` par `VehicleClass.LARGE`, `True` par `Payment.CARD`, et `0` par `when=None`, qui signifie maintenant.
Étape 3 : réduire le degré	Regrouper les arguments dans un objet `BookingRequest` ; ajouter plus tard un champ facultatif ne touche que le code qui en a besoin.
Résultat	La connascence de position et de signification (forte, degré 31) devient une connascence de nom et de type (faible), que le vérificateur de types et l'IDE peuvent vérifier et renommer automatiquement.

Rien n'a changé dans le comportement ; la refactorisation a seulement changé les accords que les appelants doivent respecter et la visibilité de ces accords.

Effectuez le changement par étapes qui laissent fonctionner les anciens appelants, par exemple en acceptant les deux formes pendant une version et en journalisant l'utilisation de l'ancienne.

```
from dataclasses import dataclass
from datetime import datetime
from enum import Enum

class VehicleClass(Enum):
    STANDARD = "standard"
    LARGE = "large"
    ACCESSIBLE = "accessible"

class Payment(Enum):
    CARD = "card"
    CASH = "cash"

@dataclass(frozen=True, kw_only=True)
class Location:
    lat: float
    lon: float

@dataclass(frozen=True, kw_only=True)
class BookingRequest:
    rider_id: str
    pickup: Location
    dropoff: Location
    vehicle: VehicleClass = VehicleClass.STANDARD
    payment: Payment = Payment.CARD
    when: datetime | None = None          # None means now

def book(request: BookingRequest) → str: ...
```

Après la refactorisation : chaque accord entre appelant et appelé est un nom ou un type (kw_only fait d'un appel positionnel une erreur).

La connascence à travers les frontières des services

Mark Richards et Neal Ford, dans *Fundamentals of Software Architecture* (2020 ; seconde édition 2025), présentent la connascence, aux côtés de la cohésion et du couplage, comme l'une des mesures fondamentales de la modularité, et soutiennent qu'elle est aussi utile entre services qu'au sein du code. Lisez les formes en pensant à un système distribué : chacune correspond à un problème familier.

Le **nom et le type** entre services sont les accords qu'un schéma publié rend explicites : noms et types des champs dans un document OpenAPI, un schéma Avro ou Protobuf, ou un catalogue d'événements. Ce sont les formes que vous voulez, et les tests de contrat peuvent les vérifier.

La **signification** entre services, c'est le code de statut partagé ou l'indicateur inexpliqué : ce que la section 3.2 appelait le couplage sémantique. Préférez les valeurs auto-descriptives (`"status": "rider_assigned"`) aux nombres, et les unités dans les noms de champs

(`fee_pence`, `timeout_ms`) aux suppositions. Les unités méritent une attention particulière : la sonde Mars Climate Orbiter de la NASA a été perdue en 1999 parce que le logiciel au sol d'une équipe produisait des valeurs d'impulsion en livres-force secondes, alors que le logiciel de navigation qui les consommait attendait des newtons-secondes : une connascence de signification entre organisations que personne n'avait rendue explicite.

La **position** entre services apparaît dans les échanges de fichiers à largeur fixe ou CSV et dans les formats de messages positionnels ; les champs nommés la suppriment. L'**algorithme** apparaît lorsque deux services doivent calculer indépendamment le même prix, la même remise ou le même hachage ; le remède consiste généralement à faire calculer la valeur par un seul service, qui envoie le résultat.

L'**exécution et le timing** entre services sont les problèmes difficiles des systèmes distribués, et le timing est la connascence qui se cache derrière le couplage temporel de la section 3.2 : un consommateur qui suppose que l'écriture en base du producteur est visible avant l'arrivée de l'événement, une chaîne d'appels synchrones dont les délais d'expiration ne sont pas correctement imbriqués, un workflow qui ne fonctionne que si les messages arrivent dans l'ordre. Rendre les interactions asynchrones, idempotentes et tolérantes au désordre affaiblit ces formes ; des patrons comme les sagas et l'outbox (chapitre 6) existent en grande partie pour les gérer.

La **valeur et l'identité** entre services sont les signes d'une frontière mal placée. Si deux services doivent garder une valeur cohérente (le total de la commande dans le service des commandes et le total de la facture dans la facturation), ou doivent agir sur la même ligne dans la même transaction, alors soit un service doit détenir ces données et l'autre les lui demander, soit les deux services devraient probablement n'en faire qu'un. Une connascence forte à travers un réseau, c'est la définition même d'un monolithe distribué.

ATTENTION

Erreurs courantes

- Traiter tout couplage comme mauvais. La connascence de nom est le prix de toute collaboration ; l'objectif est une connascence faible, locale et de faible degré, pas une connascence nulle.
- Affaiblir la forme mais augmenter le degré, par exemple en plaçant toutes les constantes de tous les services dans une seule bibliothèque partagée « constants » que tous les services doivent mettre à jour ensemble.
- Corriger les formes statiques et ignorer les formes dynamiques. Une API magnifiquement typée dont les appelants doivent invoquer trois endpoints dans un ordre précis présente toujours une forte connascence d'exécution.
- Oublier les éléments qui ne sont pas du code : les fichiers de configuration, les vues de base de données, les tableaux de bord et les règles d'alerte sont connascents avec le code qu'ils décrivent.

Le couplage équilibré : force, distance et volatilité

La connascence décrit le type de connaissance que deux parties partagent. *Balancing Coupling in Software Design* (2024) de Vlad Khononov, mentionné à la section 3.2, s'appuie sur la conception structurée et la connascence et pose une autre question : ce couplage est-il un problème *ici* ? Il soutient que le couplage n'est pas un mal à réduire au minimum, mais le ciment qui fait d'un système un système, et il le juge sur trois dimensions à la fois.

La **force d'intégration** mesure la quantité de connaissance que les parties partagent. Par ordre décroissant, ses quatre niveaux sont *intrusif* (s'appuyer sur les éléments internes privés d'un autre, comme ses tables de base de données), *fonctionnel* (partager la connaissance d'exigences métier, comme avec une règle dupliquée), *de modèle* (partager un modèle de domaine) et *de contrat* (partager seulement une interface explicite et stable qui masque le reste). La **distance** indique à quel point les parties sont éloignées : même fichier, même module, services différents. Elle augmente le coût de les modifier ensemble, et elle est en partie organisationnelle, car deux services détenus par une même équipe sont en pratique plus proches que deux services détenus par des équipes différentes, comme le prédirait la loi de Conway. La **volatilité** est la probabilité qu'une partie change. La conception pilotée par le domaine aide à l'estimer : les sous-domaines cœurs changent le plus, les sous-domaines de soutien et génériques beaucoup moins.

Le modèle est souvent résumé par une formule simplifiée :

```
BALANCE = (STRENGTH XOR DISTANCE) OR NOT VOLATILITY
```

Lisez-la en deux temps. La force et la distance doivent se compenser : les parties qui partagent beaucoup ont leur place côte à côte (c'est la cohésion), et les parties éloignées doivent partager peu. Quand les deux sont élevées, un changement est coûteux et fréquent, et c'est le monolithe distribué. Quand les deux sont faibles, des choses sans rapport cohabitent sans raison, ce qui est aussi de la complexité. La seconde étape est la clause de sauvegarde : si aucune des parties ne doit changer, un déséquilibre peut être sans importance.

Sur la plateforme de livraison de repas, supposons que Livraison lise directement les tables de Commandes. C'est une force intrusive à travers une frontière de service et d'équipe, et la prise de commande est au cœur de l'activité : le couplage est donc déséquilibré. Il existe deux corrections honnêtes : réduire la force en publiant un contrat, ou réduire la distance en fusionnant les deux. À l'inverse, une intégration légère avec un ancien système financier qui change rarement peut être intrusive et distante, tout en restant tolérable tant que ce système ne bouge pas.

Traitez le modèle comme un outil de réflexion, pas comme une mesure. La formule est une simplification binaire du traitement plus complet qu'offre le livre. La volatilité est une prévision, et l'historique des commits peut induire en erreur : des commits fréquents peuvent traduire une mauvaise conception, et du code rarement touché peut simplement être trop effrayant à modifier. Un jugement de faible volatilité doit donc être consigné, par exemple dans un enregistrement de décision, et réexaminé.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Une fonction prend `(width, height)` et un appelant passe `(height, width)`. Quelle forme de connascence a été rompue, et quelle est la façon la plus simple de l'affaiblir ?**

La connascence de position. Utilisez des arguments nommés ou un objet paramètre, afin que l'accord devienne un accord de nom.

▼ **Pourquoi la connascence de signification est-elle plus forte que la connascence de nom ?**

Les outils et les lecteurs peuvent trouver chaque utilisation d'un nom, mais rien ne relie les utilisations séparées d'une valeur magique comme 3 : les modifications sont donc faciles à manquer.

▼ **Deux services doivent toujours mettre à jour la limite de crédit d'un client dans la même transaction. Qu'est-ce que cela suggère ?**

Une forte connascence d'identité ou de valeur à travers une frontière réseau : les données ont probablement besoin d'un propriétaire unique, ou les deux services vont ensemble.

▼ **Pourquoi une connascence forte au sein d'une petite fonction peut-elle être parfaitement acceptable ?**

Parce que la localité est élevée : la personne qui modifie un élément voit immédiatement l'autre, si bien que le coût et le risque de les garder cohérents sont minimes.

▼ **À vous : L'appel hold(customer, "A", 14, 1, True) est utilisé par le site web, la billetterie et un outil d'administration, où 1 signifie place pour fauteuil roulant. Nommez les types de connascence en jeu et dites comment vous les affaibliriez.**

Connascence de position : chaque appelant doit garder les arguments dans le bon ordre, et la rangée et la place pourraient être interverties en silence. Connascence de signification : 1 et True sont des valeurs magiques que les trois appelants doivent interpréter de la même façon. Le degré est de trois et les éléments sont éloignés, ce qui aggrave les choses. Utilisez des paramètres nommés ou un petit type de requête, et une énumération nommée telle que SeatType.WHEELCHAIR. Les deux deviennent une connascence de nom, que les outils peuvent vérifier.

SOURCES

- connascence.io: reference to the forms and properties of connascence: https://connascence.io/
- connascence.io: connascence of position: https://connascence.io/position.html
- Practicing Ruby: Connascence as a software design metric (on Jim Weirich's talk): https://practicingruby.com/articles/connascence
- Balanced Coupling (coupling.dev): https://coupling.dev/
- Balanced Coupling: Balance: https://coupling.dev/posts/core-concepts/balance/
- Balanced Coupling: Integration strength: https://coupling.dev/posts/dimensions-of-coupling/integration-strength/
- Balanced Coupling: Distance: https://coupling.dev/posts/dimensions-of-coupling/distance/
- Balanced Coupling: Volatility: https://coupling.dev/posts/dimensions-of-coupling/volatility/
- Pearson: Balancing Coupling in Software Design: https://www.pearson.com/en-nz/subject-catalog/p/balancing-coupling-in-software-design-successful-software-architecture-in-general-and-distributed-systems/P200000000372/9780137353576

SECTION 3.7 SUR 3.8 · INTERMÉDIAIRE · 22 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

La loi de Conway et la forme des organisations

AVANT DE LIRE

Cette section s'appuie sur :

- 1.3 Pourquoi c'est important
- 3.1 Modularité
- 3.2 Couplage et cohésion

Termes déjà vus qu'elle utilise : Interface, Couplage, Dépendance

L'IDÉE EN BREF

Si quatre groupes de bâtisseurs qui ne se parlent jamais construisent chacun une partie d'une maison, vous obtiendrez quatre parties qui s'assemblent mal, avec les jointures exactement là où les groupes se sont rencontrés. Le logiciel fonctionne de la même manière : les parties d'un système et les liens entre elles tendent à reproduire les équipes qui se parlent. Ainsi, quand vous concevez un système, vous concevez aussi, que vous le vouliez ou non, qui doit parler à qui, et l'inverse est tout aussi vrai.

CE QU'IL FAUT SAVOIR

- L'article de Melvin Conway paru en 1968 dans Datamation soutenait que les organisations qui conçoivent des systèmes sont contraintes de produire des conceptions qui copient leurs propres structures de communication.
- La raison est mécanique : une interface entre deux sous-systèmes n'existe que là où leurs concepteurs l'ont négociée, si bien que le graphe du système reflète le graphe de communication de l'organisation.
- MacCormack, Baldwin et Rusnak (2012) ont constaté que les produits d'organisations faiblement couplées étaient systématiquement plus modulaires que des produits comparables issus d'organisations fortement couplées, même si une revue de 2016 a jugé l'effet miroir fréquent plutôt qu'universel.
- Un groupe de n personnes compte n(n-1)/2 canaux de communication : le coût de coordination croît donc à peu près comme le carré de la taille de l'équipe, et de petites équipes responsables de parties correspondantes du système le maintiennent gérable.
- La manœuvre de Conway inversée façonne les équipes et leurs interactions en fonction de l'architecture voulue, mais elle n'agit que lentement face à une base de code existante fortement couplée.

> - La charge cognitive de l'équipe limite la part d'un système qu'une équipe peut prendre en charge ; les frontières doivent donner à chaque équipe une partie qu'elle peut entièrement comprendre, construire et exploiter.

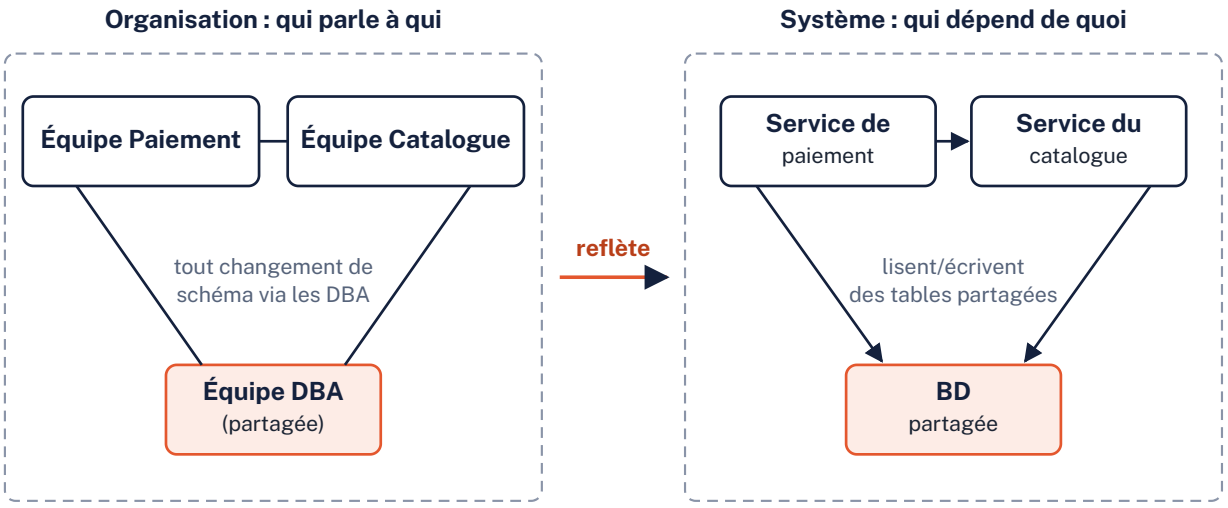
Ce que Conway a réellement dit

En avril 1968, le magazine *Datamation* a publié « How Do Committees Invent? » de Melvin Conway, programmeur et consultant en informatique. Sa conclusion est la phrase aujourd'hui connue sous le nom de loi de Conway : les organisations qui conçoivent des systèmes sont «contraintes de produire des conceptions qui sont des copies des structures de communication de ces organisations». Le nom est venu plus tard : Fred Brooks a cité l'article dans *The Mythical Man-Month* (1975) et a appelé l'idée la loi de Conway.

L'argument de Conway est plus précis que le slogan. Un système se compose de sous-systèmes reliés par des interfaces. Pour que deux sous-systèmes se connectent, les personnes qui les conçoivent doivent avoir négocié et convenu de cette interface : elles doivent donc avoir communiqué. Si les concepteurs de deux sous-systèmes ne communiquent pas, il n'y a aucune interface sur laquelle s'entendre. Il existe donc une correspondance qui préserve la structure (Conway employait le terme mathématique d'homomorphisme) entre le graphe du système et le graphe de l'organisation qui l'a conçu. Son exemple : une société de recherche sous contrat disposait de huit personnes pour écrire un compilateur COBOL et un compilateur ALGOL. Cinq ont été affectées au COBOL et trois à l'ALGOL. Le compilateur COBOL fonctionnait en cinq phases ; le compilateur ALGOL en trois.

Deux autres points de l'article sont souvent oubliés. D'abord, Conway précisait explicitement que « système » s'entend au sens large : cela s'applique à tout objet conçu, pas seulement au logiciel. Ensuite, sa conclusion pratique portait sur la **flexibilité**. Comme la première conception est rarement la meilleure, et comme une organisation tend à figer sa structure tôt (une fois les équipes formées, la conception est déjà en partie décidée), il soutenait que les organisations de conception devaient rester légères et souples, et s'organiser selon la communication dont la conception a réellement besoin.

Le miroir



La structure du système copie la structure de la communication

Quand deux équipes ne peuvent se coordonner que par l'intermédiaire d'une équipe partagée chargée de la base de données, le système qu'elles construisent tend à se coupler par une base de données partagée.

SI VOUS DÉBUTEZ

Si vous débutez

Imaginez que trois services d'une collectivité locale qui ne se parlent jamais construisent chacun une partie d'un site web. Vous obtiendrez probablement trois rubriques avec des menus différents, des connexions différentes et des formulaires qui vous demandent votre adresse trois fois. Personne ne l'a planifié ; c'est ce qui arrive quand les seules jointures d'un produit se trouvent là où les personnes qui le construisent se sont réellement parlé. La loi de Conway affirme qu'il ne s'agit pas d'un accident, mais d'une tendance que l'on peut prévoir, et donc utiliser.

Les données probantes : l'hypothèse du miroir

Pendant des décennies, la loi de Conway s'est surtout appuyée sur des anecdotes. Depuis les années 2000, des chercheurs l'ont mise à l'épreuve, généralement sous le nom d'hypothèse du miroir (mirroring hypothesis) : les dépendances techniques d'un produit correspondront aux liens organisationnels (communication, proximité géographique, appartenance à un même employeur) entre les personnes qui le construisent.

L'étude la plus connue dans le domaine du logiciel est celle d'Alan MacCormack, Carliss Baldwin et John Rusnak, publiée dans *Research Policy* en 2012 sous le titre « Exploring the duality between product and organizational architectures: a test of the mirroring hypothesis ». Ils ont comparé des paires de produits aux fonctions similaires, l'un construit par une organisation fortement couplée (une entreprise commerciale avec une équipe réunie sur un même site) et l'autre par une organisation faiblement couplée (une communauté open source distribuée) : par exemple deux logiciels de gestion financière, deux traitements de texte, deux bases de données, et Linux comparé à des noyaux de systèmes d'exploitation commerciaux. Ils ont mesuré la modularité par le *coût de propagation*, c'est-à-dire la part moyenne des fichiers d'un système qu'une modification d'un fichier peut affecter directement ou indirectement. Dans chaque paire, le produit de l'organisation faiblement couplée était plus modulaire et, dans la paire la plus extrême, l'écart était de plusieurs fois. Les auteurs y voient la preuve que des équipes distribuées, qui ne peuvent pas facilement se coordonner, sont poussées vers des conceptions qui exigent moins de coordination.

Une étude de Microsoft Research sur Windows Vista, menée par Nachiappan Nagappan, Brendan Murphy et Victor Basili (2008), a abordé la question sous l'angle de la qualité. Des métriques décrivant l'organisation qui avait construit chaque binaire (combien d'ingénieurs y avaient touché, combien étaient partis, à quelle distance dans la hiérarchie se trouvaient les contributeurs) prédisaient mieux quels binaires étaient sujets aux défaillances que les métriques de code classiques comme le taux de modification (churn), la complexité, la couverture et les dépendances.

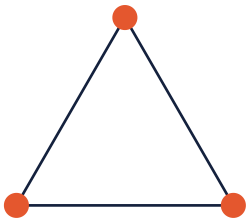
Les données ne sont pas unanimes, et un résumé honnête compte. Une revue de 142 études empiriques réalisée en 2016 par Lyra Colfer et Carliss Baldwin a constaté que l'effet miroir était fréquent dans les études portant sur des secteurs et des entreprises, mais pas universel, et que les études de projets collaboratifs ouverts ne le confirmaient souvent *pas*, sans doute parce que des dépôts de code partagés, des outils de suivi des tickets et des discussions ouvertes permettent à des contributeurs distribués de se coordonner sans les liens organisationnels habituels. L'effet miroir est une tendance forte, pas une loi de la physique, et de bons outils et de bonnes pratiques peuvent en partie le contrer.

Les canaux de communication et l'importance de la taille des équipes

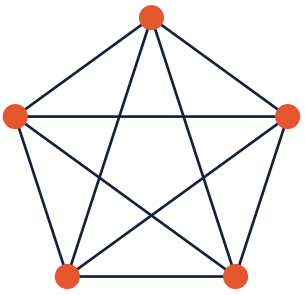
Un groupe de n personnes compte $n(n-1)/2$ canaux de communication possibles entre deux personnes. Ce nombre croît à peu près comme le carré de la taille du groupe : doubler l'équipe quadruple les canaux. Brooks s'est servi du même calcul pour expliquer pourquoi ajouter des personnes à un projet en retard le retarde davantage : chaque nouvel arrivant doit être formé et ajoute de nouveaux canaux que tout le monde doit entretenir.

Sur le plan de l'architecture, ce calcul explique deux choses. D'abord, une équipe unique ne peut pas grandir indéfiniment tout en gardant une compréhension partagée de son code ; au-delà d'une certaine taille, elle se fragmente en sous-groupes informels et, selon la loi de Conway, le code se fragmente selon les mêmes lignes, que quelqu'un ait conçu ces coutures ou non. Ensuite, le nombre de canaux entre les *équipes* compte autant que le nombre de canaux en leur sein : si chaque équipe doit parler à toutes les autres pour livrer une modification, la structure du système forme en pratique une seule grande unité couplée, quel que soit son nombre de services.

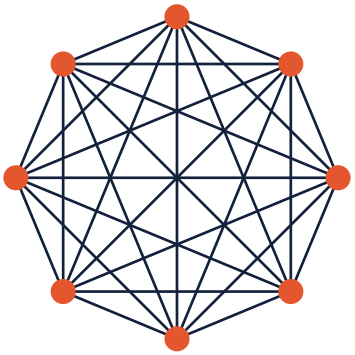
Les canaux croissent bien plus vite que les personnes



3 membres
3 canaux



5 membres
10 canaux



8 membres
28 canaux

Chaque paire de personnes est un canal de communication potentiel : 3 personnes en ont 3, 5 en ont 10 et 8 en ont 28.

EXEMPLE COMMENTÉ

Exemple commenté : les canaux de communication dans un groupe produit en croissance

Le produit de commande d'une entreprise de livraison de repas est construit par une seule équipe de 8 ingénieurs. Elle prévoit de passer à 24 ingénieurs. Comparez une grande équipe unique avec trois équipes de 8 aux responsabilités clairement définies.

Une équipe de 8	8 × 7 / 2 = 28 canaux.
Une équipe de 24	24 × 23 / 2 = 276 canaux : environ dix fois plus pour trois fois plus de personnes.
Trois équipes de 8, canaux internes	3 × 28 = 84 canaux au sein des équipes.
Trois équipes, canaux entre équipes	Si chaque équipe a un interlocuteur unique avec chacune des autres équipes (par exemple la personne responsable technique), cela ajoute 3 × 2 / 2 = 3 canaux entre équipes. Total : environ 87.
Trois équipes, tout le monde se coordonne avec tout le monde	Si les responsabilités sont floues et que chaque modification mobilise tous les ingénieurs, le graphe redevient en pratique celui à 276 canaux.
Ce que cela montre	Répartir les personnes en équipes n'aide que si le système est découpé de la même façon, de sorte que la plupart des modifications ne demandent qu'une communication interne à l'équipe. C'est l'architecture qui rend possible le graphe plus petit.

La formule donne des canaux potentiels, pas des conversations réelles ; l'important est la vitesse à laquelle le coût de coordination peut croître, pas une prédiction précise.

Team Topologies (Skelton et Pais) recommande des équipes durables d'environ 5 à 9 personnes, ce qui concorde avec le calcul : assez petites pour que les canaux restent gérables.

La manœuvre de Conway inversée

Si les organisations tendent à produire des systèmes qui reflètent leur structure, vous pouvez utiliser cette tendance délibérément : **concevez la structure des équipes et les canaux de communication en fonction de l'architecture que vous voulez**, et laissez le système prendre cette forme en grandissant. Jonny LeRoy et Matt Simons ont nommé cette approche la

manœuvre de Conway inversée dans un article du *Cutter IT Journal* en décembre 2010, et elle est depuis devenue une technique courante dans les programmes de microservices et de plateformes.

En pratique, il s'agit de décider d'abord des frontières visées (par exemple les commandes, les paiements, le suivi des livraisons et l'intégration des restaurants comme services distincts), puis de former une équipe durable autour de chacune, de lui confier la responsabilité du code, des données et du pipeline de déploiement, et de faire en sorte que les équipes interagissent principalement par des API et des événements publiés plutôt que par du code ou des bases de données partagés. Là où un contact entre équipes est réellement nécessaire, rendez-le explicite : une équipe plateforme qui propose un produit en libre-service, une équipe facilitatrice qui apporte son aide pour une durée limitée.

La manœuvre a ses limites. Elle n'épouse le grain d'un système existant que lentement : réorganiser les équipes ne restructure pas une base de code fortement couplée, et pendant un temps les nouvelles équipes lutteront contre l'ancienne architecture. Martin Fowler note qu'elle ne peut pas réparer rapidement un système existant rigide. Elle suppose aussi que vous connaissez les bonnes frontières ; si vous les tracez mal puis les figez dans des structures d'équipes, la loi de Conway reproduira fidèlement l'erreur et la rendra coûteuse à corriger. C'est pourquoi les équipes commencent souvent par un monolithe modulaire et déplacent les frontières avant de les inscrire dans l'organisation.

Charge cognitive de l'équipe

La loi de Conway vous dit que les frontières entre équipes deviennent des frontières du système. Une seconde contrainte vous dit quelle taille peut avoir la partie de chaque équipe. La charge cognitive de l'équipe, un concept que Matthew Skelton et Manuel Pais ont introduit dans la conception des organisations logicielles avec *Team Topologies* (2019 ; deuxième édition en 2025), est la quantité totale qu'une équipe doit avoir en tête pour construire, exploiter et faire évoluer ce dont elle est responsable : le domaine métier, le code, l'infrastructure, les outils, les astreintes.

Ils empruntent à la psychologie la distinction entre la charge **intrinsèque** (la difficulté inhérente à la tâche, comme comprendre le domaine), la charge **extrinsèque** (l'effort imposé par l'environnement, comme une procédure de déploiement manuelle ou un build obscur) et la charge **pertinente** (l'effort d'apprentissage qui améliore le travail futur). L'objectif de conception est de réduire au minimum la charge extrinsèque, par exemple grâce à des plateformes et à l'automatisation, afin que la capacité aille aux charges intrinsèque et pertinente.

Pour les architectes, la conséquence est directe. **Une frontière de service ou de composant doit donner à l'équipe qui en est responsable une partie du système qu'elle peut entièrement comprendre.** Si une équipe est responsable de douze services répartis sur quatre domaines, elle est surchargée, et la qualité, la vitesse ou la santé des astreintes en pâtira, aussi propre que soit le code de chaque service. À l'inverse, un domaine si compliqué qu'aucune équipe orientée flux ne peut l'absorber (un moteur de tarification à modèles actuariels, un codec vidéo) peut justifier une équipe spécialisée et une frontière claire autour de son sous-système.

Laisser la structure des équipes façonner les frontières

Prises ensemble, ces idées donnent aux architectes un ensemble de règles pratiques :

Alignez les frontières sur la responsabilité des équipes. Chaque module, service, stockage de données et pipeline doit avoir une seule équipe responsable. En pratique, une responsabilité partagée signifie une absence de responsabilité, et le code reflétera ce dont le dernier contributeur avait besoin.

Réduisez au minimum la communication qu'exige chaque modification. Une frontière est bien placée si la plupart des modifications d'une capacité métier peuvent être faites, testées et déployées par une seule équipe. Surveillez les modifications qui exigent régulièrement la coordination de trois équipes : elles montrent que la structure du système et celle des équipes ne concordent pas.

Rendez les interfaces entre équipes explicites et stables. Les interfaces entre équipes méritent un versionnage, une documentation et des tests de contrat ; les interfaces au sein d'une équipe peuvent rester informelles et évoluer librement.

Réexaminez les deux ensemble. Les réorganisations sont des décisions d'architecture ; les décisions d'architecture sont des décisions organisationnelles. Un architecte que l'on ne consulte pas sur les changements d'équipes, ou un manager qui réorganise sans tenir compte du code, ignore une force qui agira quoi qu'il arrive.

Les modèles de types d'équipes et de modes d'interaction, ainsi que la manière de les faire évoluer, sont traités dans Team Topologies et conception sociotechnique.

Quand l’organigramme et l’architecture ne concordent pas

Situation	Ce que vous voyez dans le système	Remède habituel
Équipes organisées par couche technique (frontend, backend, base de données)	Chaque fonctionnalité exige trois équipes et trois passages de relais ; les couches développent des interfaces épaisses et génériques ; les mises en production sont coordonnées	Équipes pluridisciplinaires alignées sur les capacités métier, chacune responsable d’une tranche verticale
Des microservices avec une équipe par service, mais une équipe centrale des données responsable de tous les schémas	Un système distribué couplé par une base de données partagée ; les changements de schéma attendent des semaines	Donner à chaque équipe de service la responsabilité de ses données ; l’équipe des données fournit une plateforme et des normes
Deux équipes partagent la responsabilité d’un même service	Des conventions contradictoires dans une même base de code, des conflits de fusion, personne d’astreinte pour les parties difficiles	Découper le service selon les préoccupations des équipes, ou le confier à une seule équipe
Une équipe est responsable de nombreux services sans rapport entre eux	Des services négligés, des dépendances obsolètes, une réponse lente aux incidents : une surcharge cognitive	Regrouper des services, en transférer certains à d’autres équipes, ou investir dans l’outillage de plateforme
Un prestataire externe construit un sous-système dans le cadre d’un contrat à prix forfaitaire	Une interface rigide, très spécifiée, dont la modification coûte très cher	Traiter la frontière comme une porte à sens unique ; la placer là où le changement est le moins probable
Une réorganisation transfère une capacité à une nouvelle équipe, mais le code reste dans le dépôt de l’ancienne équipe	La nouvelle équipe contourne un code dont elle n’est pas responsable ; érosion de la frontière prévue	Transférer le code, les pipelines et l’astreinte en même temps que la responsabilité

ATTENTION

Malentendus courants

- La loi de Conway n'est pas une recommandation de copier l'organigramme actuel ; c'est la description d'une force. La décision consiste à laisser l'organisation existante façonner le système, ou à façonner l'organisation pour le système dont vous avez besoin.
- Les microservices n'échappent pas à la loi. Cent services construits par des équipes qui doivent toutes se coordonner se comportent comme un seul système, avec en plus le coût d'un réseau.
- La loi porte sur la structure de communication, pas sur les liens hiérarchiques officiels. Deux équipes sous des managers différents qui travaillent côte à côte et se parlent tous les jours peuvent produire des parties étroitement intégrées.
- Le travail à distance et distribué n'abolit pas la loi ; il change la communication qui coûte peu. Les interfaces asynchrones, écrites et bien documentées deviennent les frontières naturelles.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi, dans l'argument de Conway, la structure du système doit-elle ressembler à la structure de l'organisation ?

Parce que chaque interface entre sous-systèmes doit être négociée par les personnes qui les conçoivent ; là où les personnes ne communiquent pas, aucune interface n'est conçue, si bien que les liens du système suivent les liens entre les personnes.

▼ Combien de canaux de communication une équipe de 12 personnes compte-t-elle ?

12 × 11 / 2 = 66.

▼ Une entreprise se réorganise en une équipe par microservice, mais conserve une seule base de données partagée que toutes les équipes modifient. Que prédit la loi de Conway ?

Les services resteront couplés par la base de données, et les modifications exigeront une coordination entre toutes les équipes, car la structure de communication (tout le monde touche au même schéma) n'a pas changé.

▼ Qu'est-ce que la charge cognitive extrinsèque, et comment l'architecture peut-elle la réduire ?

L'effort imposé par l'environnement plutôt que par le problème, comme des déploiements manuels ou un outillage obscur ; les plateformes, l'automatisation et des frontières claires la réduisent pour que les équipes puissent se concentrer sur leur domaine.

▼ **À vous : Un groupe de 10 personnes dans la billetterie passera à 30 ingénieurs. Comparez les chemins de communication d'une équipe unique de 30 et de trois équipes de 10 ayant un lien entre chaque paire d'équipes. Que doit-il aussi changer dans le logiciel ?**

Une équipe de 10 a 10 x 9 / 2 = 45 chemins. Une équipe de 30 en a 30 x 29 / 2 = 435. Trois équipes de 10 en ont 3 x 45 = 135 à l'intérieur des équipes, plus 3 liaisons, soit 138. Mais cela ne fonctionne que si chaque équipe possède une partie du système qui lui correspond, avec des interfaces claires. Sinon, les équipes doivent se coordonner sans cesse : planifiez donc ensemble les frontières des modules et le découpage des équipes.

SOURCES

- Melvin Conway: How Do Committees Invent? (Datamation, April 1968): https://www.melconway.com/Home/Committees_Paper.html
- Martin Fowler: Conway's Law: https://martinfowler.com/bliki/ConwaysLaw.html
- MacCormack, Baldwin and Rusnak: Exploring the duality between product and organizational architectures (Research Policy, 2012): https://doi.org/10.1016/j.respol.2012.04.011
- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Thoughtworks Technology Radar: Inverse Conway Maneuver: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

SECTION 3.8 SUR 3.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Simplicité, réversibilité et mesure de la structure

AVANT DE LIRE

Cette section s'appuie sur :

- 1.6 Décisions et compromis
- 1.7 Dette technique
- 3.4 Dépendances

Termes déjà vus qu'elle utilise : Couplage afférent (Ca), Instabilité (I), Cycle de dépendances, Couplage, Refactorisation (refactoring)

L'IDÉE EN BREF

Certaines choses sont difficiles parce que la tâche est difficile : une déclaration d'impôts est compliquée parce que le droit fiscal l'est. D'autres sont difficiles à cause de la façon dont nous avons choisi de les faire : un système de classement à sept dossiers là où deux suffiraient. Les bons concepteurs acceptent le premier type et combattent le second. Ils construisent ce qui est nécessaire maintenant, consignent chaque fait à un seul endroit, évitent les choix qu'ils ne peuvent pas défaire et vérifient leur travail avec quelques mesures honnêtes, sans oublier qu'un chiffre peut être manipulé s'il devient l'objectif.

CE QU'IL FAUT SAVOIR

- Fred Brooks distinguait la complexité essentielle, qui appartient au problème, de la complexité accidentelle, qui vient de la manière dont la solution est construite ; les architectes placent la première et éliminent la seconde.
- YAGNI met en garde contre le fait de construire pour des besoins présumés, en raison des coûts de construction, de retard, de maintien et de réparation, mais il n'excuse pas de négliger les tests, la refactorisation ou les décisions qui sont des portes à sens unique.
- DRY, tel que le définissent Hunt et Thomas, veut que chaque élément de connaissance ait une seule représentation faisant autorité ; fusionner du code qui se ressemble seulement crée un couplage néfaste.
- La flexibilité est une option qui a un prix : payez-la quand le changement est probable ou très coûteux sans elle, et que l'option est bon marché, et comptez sur une bonne modularité pour le reste.

- La complexité cyclomatique, M = E − N + 2P, compte les chemins indépendants à travers une fonction et vaut, pour du code structuré, le nombre de décisions plus un ; McCabe suggérait une limite d'environ 10.
- Le LCOM, les métriques de couplage et de stabilité, le couplage de changement et les hotspots révèlent chacun des problèmes structurels différents, et tous deviennent trompeurs lorsqu'on en fait des objectifs (loi de Goodhart).
- Les fonctions d'aptitude transforment des décisions structurelles, comme des dépendances interdites, en vérifications automatisées qui font échouer le build lorsqu'elles ne sont pas respectées.

Deux types de complexité

Dans « No Silver Bullet: Essence and Accidents of Software Engineering » (présenté au congrès mondial de l'informatique de l'IFIP en 1986 et publié dans *IEEE Computer* en 1987), Fred Brooks a séparé les difficultés de la construction du logiciel en deux types. La complexité essentielle appartient au problème : les vraies règles de la TVA, la vraie diversité des produits d'assurance, les vraies façons dont une livraison peut échouer. Aucune conception ne peut l'éliminer sans changer ce que fait le système. La complexité accidentelle vient de la manière dont nous construisons la solution : langages, outils, frameworks, mécanismes de déploiement, contournements, et une structure plus compliquée que ne l'exige le problème. Brooks soutenait que les langages de haut niveau et d'autres progrès avaient déjà éliminé une grande partie de la difficulté accidentelle, si bien qu'aucune nouvelle technique isolée n'apporterait un gain d'un facteur dix en une décennie.

Pour les architectes, cette distinction est un outil quotidien. La complexité essentielle doit être *placée* : mettez-la là où les personnes qui la comprennent peuvent y travailler, derrière une frontière qui l'empêche de se répandre partout ailleurs (un module de tarification, un service fiscal). La complexité accidentelle doit être *éliminée ou évitée* : chaque service, file d'attente, cache, couche d'abstraction et option de configuration supplémentaire en ajoute, et chacun doit mériter sa place en résolvant un vrai problème. L'essai de Ben Moseley et Peter Marks « Out of the Tar Pit » (2006) a poussé l'idée plus loin en désignant l'état mutable et le flux de contrôle comme des sources majeures de complexité accidentelle, ce qui explique en partie la popularité des conceptions fonctionnelles et pilotées par les événements qui limitent l'état partagé.

Le reste de cette section présente les principes qui contiennent la complexité accidentelle (KISS, YAGNI, DRY bien compris, réversibilité) et les mesures qui vous aident à voir la structure de façon objective, avec leurs limites.

KISS et YAGNI

KISS, généralement développé en « keep it simple, stupid » (« reste simple, idiot »), est une maxime de conception attestée dans l'US Navy en 1960 et associée à Kelly Johnson, l'ingénieur en chef des Skunk Works de Lockheed, qui aurait exigé que les avions à réaction puissent être réparés sur le terrain par un mécanicien moyen muni d'une poignée d'outils. La lecture logicielle est la même : une conception est bonne si les personnes qui doivent l'exploiter et la réparer, souvent à 3 h du matin et sous pression, peuvent la comprendre. Simple ne veut pas dire facile ni naïf. Cela signifie peu de concepts, peu de pièces mobiles et peu de cas particuliers pour le problème à résoudre.

YAGNI (« you aren't gonna need it », « vous n'en aurez pas besoin ») vient de l'Extreme Programming, à la fin des années 1990 : ne construisez pas de capacités pour des besoins seulement présumés. L'analyse de Martin Fowler est utile, car elle nomme les coûts liés au non-respect de ce principe. Il y a le **coût de construction** de la fonctionnalité présumée ; le **coût du retard**, parce qu'une chose utile n'a pas été construite à la place ; le **coût de maintien**, parce que chaque modification ultérieure doit composer avec le code supplémentaire ; et, si l'hypothèse était fausse, le **coût de réparation**. Même quand l'hypothèse s'avère juste, la fonctionnalité a été construite avec moins de connaissances que vous n'en auriez eu plus tard.

Deux nuances empêchent YAGNI de devenir une excuse. D'abord, il s'applique aux *fonctionnalités et à la flexibilité spéculative*, pas aux pratiques qui rendent le changement peu coûteux. Les tests automatisés, la refactorisation, la livraison continue et des frontières de modules claires sont ce qui permet de reporter des fonctionnalités en toute sécurité ; les abandonner au nom de YAGNI transforme le report en dette. Ensuite, certaines décisions sont des portes à sens unique (section 1.6). Choisir un modèle de données, la forme d'une API publique, un modèle de multilocation ou une frontière de sécurité n'est pas une fonctionnalité que l'on peut ajouter plus tard au même coût. Pour ces décisions, anticiper n'est pas de la spéculation : c'est le travail de l'architecte.

DRY porte sur la connaissance, pas sur le texte

Andy Hunt et Dave Thomas ont introduit DRY dans *The Pragmatic Programmer* (1999 ; édition du 20e anniversaire en 2019) avec cette définition : «Chaque élément de connaissance doit avoir une représentation unique, non ambiguë et faisant autorité au sein d'un système.» Le mot clé est *connaissance*. Si le taux de TVA, la règle de validité d'un code postal ou la signification d'un statut de commande est écrit à plusieurs endroits, le modifier exige de retrouver chaque copie, et les copies divergeront. L'édition de 2019 souligne que DRY porte sur la duplication de la connaissance et de l'intention, pas sur des lignes de code qui se trouvent se ressembler.

Le mauvais usage courant va dans l'autre sens. Deux fonctions qui se ressemblent sont fusionnées parce qu'elles se ressemblent, alors qu'elles représentent des connaissances différentes qui changeront pour des raisons différentes : la règle des frais de livraison et la règle de commission des restaurants multiplient toutes deux un total par un pourcentage aujourd'hui, mais elles relèvent d'acteurs différents (la finance et les partenariats) et divergeront. Les fusionner couple deux concepts que le domaine sépare : les deux dépendent désormais d'une seule fonction partagée, si bien qu'une modification demandée pour l'une change l'autre en silence. Dans les termes de la section 3.6, cela crée une connascence là où le problème n'en a aucune (partager une fonction n'est le bon remède que si les deux côtés doivent réellement calculer la même chose) ; c'est aussi une violation de la responsabilité unique (section 3.5). Le conseil bien connu de Sandi Metz est de préférer la duplication à la mauvaise abstraction, car une mauvaise abstraction partagée accumule paramètres et conditions jusqu'à ce que personne n'ose plus la modifier. Beaucoup d'équipes appliquent la règle de trois : tolérer une deuxième copie, et extraire une abstraction quand la troisième apparaît et que la connaissance commune est claire.

À l'échelle de l'architecture, le compromis devient explicite. Une bibliothèque partagée que chaque service doit utiliser supprime le code dupliqué, mais couple les cycles de mise en production des services. Beaucoup d'organisations à microservices acceptent délibérément du *code* dupliqué (de petits utilitaires copiés dans chaque service) tout en exigeant un responsable unique pour la *connaissance* dupliquée (la fiche client de référence, les règles fiscales). Posez la question pour toute duplication : si cette règle change, toutes les copies doivent-elles changer ensemble ? Si oui, c'est de la connaissance dupliquée, qui doit avoir un seul responsable. Si non, c'est une ressemblance fortuite, qui peut rester.

SI VOUS DÉBUTEZ

Pas à pas : quand la fusion de code semblable se retourne contre vous

L'idée la plus difficile de cette section est que DRY porte sur la connaissance, pas sur du code qui se ressemble. Le chapitre 0 a montré pourquoi une règle doit être écrite une seule fois ; toute la difficulté est de distinguer une règle écrite deux fois de deux règles qui se ressemblent seulement. Suivez l'exemple de la section.

- Sur la plateforme de livraison de repas, la règle des frais de livraison et la règle de commission des restaurants multiplient toutes deux aujourd'hui le total d'une commande par un pourcentage. Le même calcul apparaît deux fois.
- Se rappelant qu'une règle doit se trouver à un seul endroit, un développeur les fusionne en une seule fonction partagée. Le code est plus court.
- Des mois plus tard, une modification est demandée pour la seule règle de commission : la commission est désormais plafonnée à un montant maximal par commande. Le développeur ajoute le plafond dans la fonction partagée, et les frais de livraison sont plafonnés eux aussi, en silence. Personne ne l'avait demandé.
- Pour réparer cela, on ajoute un indicateur afin que la fonction sache quelle règle elle calcule : c'est le couplage de contrôle de la section 3.2. Chaque demande ultérieure ajoute un paramètre ou une condition, jusqu'à ce que personne n'ose plus modifier la fonction.
- Les deux règles n'ont jamais été la même connaissance : elles relèvent d'équipes différentes (la finance et les partenariats) et changent pour des raisons différentes. Deux petites fonctions séparées étaient la meilleure conception.
- Comparez avec le taux de TVA écrit dans l'étape de paiement, la facture et l'écran de remboursement : chaque copie doit changer chaque fois que le taux change. C'est un seul élément de connaissance écrit trois fois, et il lui faut un seul responsable.
- **La leçon :** le critère est de savoir si toutes les copies doivent changer ensemble quand la règle change. Si oui, donnez à la règle un seul emplacement. Si non, la ressemblance est une coïncidence, et un peu de duplication coûte généralement moins cher que la mauvaise abstraction partagée.

La réversibilité et le coût des options

La simplicité a une dimension temporelle. Une conception simple aujourd'hui mais impossible à modifier n'est pas bon marché ; une conception pleine d'une flexibilité que personne n'utilise ne l'est pas non plus. Le concept utile est la réversibilité : combien coûterait-il d'annuler ou de modifier cette décision plus tard ? La section 1.6 a présenté les portes à sens unique et à double sens. Ici, l'idée est que la flexibilité se comporte comme une option financière : elle a un

prix maintenant (code supplémentaire, abstraction supplémentaire, coût d'exploitation supplémentaire) et une valeur qui dépend de la probabilité et du coût du changement contre lequel elle protège.

Cela donne une règle empirique qui concilie YAGNI et anticipation. **Ne payez une option que si le changement est raisonnablement probable, ou s'il serait très coûteux sans l'option, et si l'option elle-même est bon marché.** Envelopper un prestataire de paiement derrière une interface qui vous appartient coûte une journée et protège contre un changement à la fois plausible et coûteux : c'est une bonne option. Construire une couche d'abstraction multicloud pour une start-up qui n'a qu'un client est une option coûteuse contre un changement improbable. Souvent, l'option la moins chère est simplement une bonne modularité : des frontières bien placées gardent de nombreux changements futurs locaux sans que personne ait à les prédire, ce qui est l'argument originel de Parnas en faveur du masquage de l'information (section 3.1).

Mesurer la structure : la complexité cyclomatique

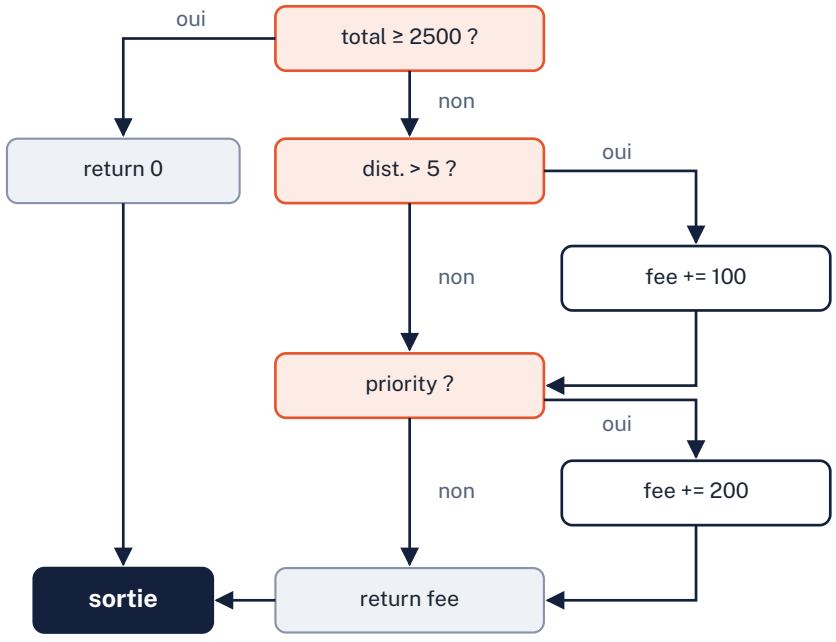
Le jugement sur la simplicité s'améliore quand il s'appuie sur la mesure. La plus ancienne métrique de code largement utilisée est la complexité cyclomatique, introduite par Thomas McCabe dans « A Complexity Measure » (*IEEE Transactions on Software Engineering*, 1976). Elle traite une fonction comme un graphe de flot de contrôle, avec des nœuds pour les blocs de code et des arêtes pour les transferts de contrôle possibles, et compte les chemins linéairement indépendants qui le traversent : $M = E - N + 2P$, où E est le nombre d'arêtes, N le nombre de nœuds et P le nombre de composantes connexes (1 pour une seule fonction). Pour du code structuré avec un seul point d'entrée, cela équivaut au nombre de points de décision plus un.

McCabe suggérait 10 comme limite supérieure raisonnable pour un module. Le rapport ultérieur du NIST sur le test structuré, d'Arthur Watson et McCabe (1996), note que des limites allant jusqu'à 15 ont été utilisées avec succès par des équipes expérimentées, et recommande une explication écrite chaque fois qu'un module dépasse la limite choisie. Des outils d'analyse statique comme `radon` ou le plugin `mccabe` pour flake8 en Python, la règle `complexity` d'ESLint en JavaScript et SonarQube la calculent automatiquement. Les outils diffèrent sur des détails, par exemple selon que chaque `and` ou `or` d'une condition compte ou non comme une décision supplémentaire : comparez donc des chiffres issus du même outil.

```
def delivery_fee(total: int, distance_km: float, priority: bool) → int:
    if total ⩾ 2500:          # decision 1: free delivery over 25 pounds
        return 0
    fee = 299
    if distance_km > 5:        # decision 2
        fee += 100
    if priority:               # decision 3
        fee += 200
    return fee
```

Trois décisions, donc une complexité cyclomatique de 4.

Le graphe de flot de contrôle et le décompte



Comptage

Nœuds N = 8

Arêtes E = 10

M = E − N + 2

M = 10 − 8 + 2 = 4

Vérif. : 3 décisions + 1 = 4

4 chemins indép.

donc au moins 4 cas de test

pour les chemins de base

Huit nœuds et dix arêtes donnent M = 10 − 8 + 2 = 4, ce qui correspond à trois décisions plus un.

EXEMPLE COMMENTÉ

Exemple commenté : quand la fonction grossit

Un an plus tard, `delivery_fee` s'est enrichie d'une boucle sur le panier et d'une condition composée. Calculez sa complexité et décidez quoi faire.

Nouveau code	`for item in items:` avec `if item.bulky: fee += 50` à l'intérieur, et `if priority and rider_available: fee += 200` à la place de `if priority`.
Décisions	Les 3 d'origine, plus la boucle `for` (1), plus le `if item.bulky` (1), plus le `and` (1, si l'outil compte les opérateurs booléens) : 6.
Complexité cyclomatique	6 + 1 = 7. En dessous de la limite de 10 de McCabe, mais elle a presque doublé.
Conséquence pour les tests	Le test des chemins de base exige au moins 7 cas de test pour parcourir une fois chaque chemin indépendant ; l'ancienne suite de 4 ne couvre plus tous les chemins.
Lecture de la conception	La fonction mêle désormais trois préoccupations de tarification : le seuil de montant du panier, les suppléments liés à la distance et aux articles, et la priorité. Elles évoluent à la demande d'acteurs différents (marketing, opérations, produit).
Refactorisation (refactoring)	Extrayez `distance_surcharge`, `bulky_surcharge` et `priority_surcharge`, chacune d'une complexité de 2 ou 3, et laissez `delivery_fee` en faire la somme. La complexité totale est similaire, mais chaque élément est petit, nommé et testable isolément.

La refactorisation réduit rarement la complexité totale ; elle la redistribue en éléments plus faciles à comprendre, à tester et à modifier indépendamment. La complexité essentielle des règles de tarification demeure.

La métrique a signalé une tendance qui méritait une discussion ; la décision de découper a été prise en lisant le code et en se demandant qui le modifie.

Mesurer la structure au-delà des fonctions

La complexité cyclomatique ne voit qu'une fonction à la fois. Les architectes ont besoin de mesures des relations entre modules.

Métriques de couplage et de stabilité. La section 3.4 (gérer les dépendances) a présenté le couplage afférent et efférent, l'instabilité $I = Ce / (Ca + Ce)$, le degré d'abstraction et la distance à la séquence principale. Ces métriques se calculent à peu de frais à partir des graphes

d'imports ; elles révèlent bien les composants dont beaucoup d'autres dépendent mais qui dépendent eux-mêmes de code volatil, et détectent les cycles de dépendances.

Métriques de cohésion. La famille LCOM (manque de cohésion des méthodes) estime si une classe porte une seule responsabilité ou plusieurs. La version originale de Shyam Chidamber et Chris Kemerer (1994) compare des paires de méthodes selon qu'elles utilisent ou non des champs communs. Une variante ultérieure très répandue, LCOM4, de Martin Hitz et Behzad Montazeri, construit un graphe dans lequel les méthodes sont reliées si elles partagent un champ ou si l'une appelle l'autre, puis compte les groupes connexes. Une classe avec LCOM4 = 1 est cohérente ; une classe avec LCOM4 = 3 est en réalité trois classes qui partagent un fichier. Par exemple, une classe `Customer` dont `update_address` et `format_address` utilisent les champs d'adresse, dont `add_points` et `points_balance` utilisent les champs de fidélité, et dont `send_newsletter` n'utilise que le champ e-mail a un LCOM4 = 3, ce qui suggère des responsabilités d'adresse, de fidélité et de marketing.

Métriques fondées sur l'historique. Le contrôle de version enregistre la façon dont le système change réellement, ce que l'analyse statique ne peut pas voir. Le couplage de changement repère les fichiers ou les services qui sont régulièrement commités ensemble : si `pricing.py` et `invoice_template.html` changent ensemble dans 70 % des commits, ils sont couplés, que l'un importe l'autre ou non, et ce couplage peut franchir une frontière de service censée les séparer. Les hotspots (points chauds) combinent la fréquence des changements et la complexité pour repérer le code à la fois difficile à manipuler et fréquemment modifié ; *Your Code as a Crime Scene* d'Adam Tornhill (2015 ; deuxième édition en 2024) a popularisé cette technique. La section 1.7, consacrée à la dette technique, a montré comment les hotspots orientent le remboursement vers la dette qui coûte le plus d'intérêts.

Les métriques structurelles en un coup d'œil

Métrique	Portée	Ce qu'elle révèle	Principal angle mort
Complexité cyclomatique	Fonction	Une logique de branchement difficile à tester et à raisonner	Ne dit rien du nommage, des données ni des relations entre fonctions
Ca, Ce, instabilité, distance	Paquet ou composant	Sens des dépendances, stabilité, cycles	Compte les dépendances, pas leur force ni leur signification
LCOM (par exemple LCOM4)	Classe	Des classes portant plusieurs responsabilités sans rapport entre elles	Sensible aux accesseurs et aux fonctionnalités du langage ; bruitée sur les petites classes
Couplage de changement	Fichier, module ou service	Un couplage caché qui n'apparaît que dans la façon dont le système change	Exige un historique suffisant ; les commits massifs créent du bruit
Hotspots	Fichier ou module	Là où une amélioration est rentabilisée le plus vite	Établit des priorités ; ne dit pas ce qui ne va pas
Connascence (évaluée en revue)	Toutes	La forme et la force de chaque dépendance	Largement manuelle ; peu d'outils mesurent les formes dynamiques

La loi de Goodhart et les limites des métriques

Chacune des métriques ci-dessus est un indicateur indirect de quelque chose qui nous importe mais que nous ne pouvons pas mesurer directement : la facilité à comprendre et à modifier le système. La loi de Goodhart, nommée d'après l'économiste Charles Goodhart (1975) et surtout connue dans la formulation de l'anthropologue Marilyn Strathern en 1997, avertit que lorsqu'une mesure devient un objectif, elle cesse d'être une bonne mesure. Les gens optimisent le chiffre, pas la propriété.

Les équipes logicielles le constatent sans cesse. Une limite stricte de complexité cyclomatique à 10 produit des fonctions découpées à des endroits arbitraires, qui passent chacune la vérification tandis que la logique devient plus difficile à suivre. Un objectif de couverture de 80 % produit des tests qui exécutent du code sans rien vérifier. Une règle interdisant à toute classe de dépasser 200 lignes produit des classes scindées en `OrderHelper` et `OrderHelper2`. Un objectif « zéro cycle » peut être atteint en déplaçant du code dans un paquet fourre-tout `common` dont tout dépend.

Les professionnels utilisent les métriques de trois manières qui résistent à la manipulation. Ils regardent les **tendances** plutôt que des seuils absolus (la complexité du module de paiements a doublé en six mois : pourquoi ?). Ils utilisent les métriques pour **lancer des discussions et**

prioriser les revues, pas pour noter les personnes. Et ils **combinent** des métriques qui tirent dans des directions différentes, comme la complexité avec la fréquence des changements, ou le couplage avec la responsabilité des équipes, afin qu'aucun chiffre isolé ne puisse être optimisé au détriment des autres.

Fonctions d'aptitude : rendre la structure vérifiable

Certaines règles structurelles sont assez claires pour être automatisées, et elles cessent alors de dépendre de la vigilance. *Building Evolutionary Architectures* de Neal Ford, Rebecca Parsons et Patrick Kua (2017 ; deuxième édition en 2022 avec Pramod Sadalage) appelle ces vérifications des fonctions d'aptitude (fitness functions) : des tests objectifs et automatisés d'une caractéristique architecturale, exécutés dans le pipeline de build ou sur le système en fonctionnement. Exemples pour la structure : aucun cycle de dépendances entre les paquets de premier niveau ; le paquet du domaine n'importe rien des paquets web ou base de données ; aucune fonction au-dessus d'une complexité de 15 sans dérogation explicite ; aucun service ne lit les tables de base de données d'un autre service.

Des outils comme ArchUnit (Java), import-linter (Python) et dependency-cruiser (JavaScript et TypeScript) expriment ces règles sous forme de code ; la section 3.4 en a donné des exemples. La différence avec une métrique de tableau de bord est qu'une fonction d'aptitude encode une *décision* que l'équipe a déjà prise : l'enfreindre fait échouer le build et impose un choix conscient, corriger le code ou modifier la règle en consignant pourquoi. Le traitement complet, y compris les fonctions d'aptitude pour les performances, la sécurité et d'autres qualités, se trouve dans Fonctions d'aptitude et architecture évolutive.

```
[importlinter]
root_package = shop

[importlinter:contract:domain-is-pure]
name = Domain must not depend on infrastructure
type = forbidden
source_modules =
    shop.domain
forbidden_modules =
    shop.infrastructure
    shop.web
```

Un contrat import-linter dans setup.cfg ou .importlinter qui fait échouer le build si la couche domaine importe l'infrastructure.

Liste de contrôle : revoir la conception d'un module

Utilisez-la pour revoir un nouveau module, composant ou service, ou une modification importante de l'un d'eux. Elle s'appuie sur toutes les sections de ce chapitre.

- [ ] **Pouvez-vous énoncer sa responsabilité en une phrase, et nommer l'acteur qui demande ses modifications ?**
  Si la phrase a besoin d'un « et », ou si plusieurs acteurs le modifient, envisagez de le découper (modularité, SRP, CCP).
- [ ] **Masque-t-il une décision de conception susceptible de changer ?**
  Un module doit détenir un secret, comme un format de données, un fournisseur ou un algorithme (masquage de l'information).
- [ ] **L'interface est-elle petite par rapport à ce qu'elle masque ?**
  Les modules profonds masquent beaucoup derrière peu ; les couches de simple transmission ajoutent un coût sans rien masquer.
- [ ] **Ses dépendances pointent-elles vers des modules plus stables et plus abstraits ?**
  Recherchez les cycles et les modules stables qui dépendent de modules volatils (ADP, SDP, DIP).
- [ ] **Que doit savoir un client pour l'utiliser correctement ?**
  Recherchez les paramètres positionnels, les valeurs magiques, les ordres d'appel imposés et les hypothèses de timing ; affaiblissez-les ou gardez-les locaux (connascence).
- [ ] **Qui en est responsable, et cette équipe peut-elle le modifier et le déployer sans se coordonner avec d'autres ?**
  Une seule équipe responsable par module ; les interfaces entre équipes doivent être explicites et versionnées (loi de Conway).
- [ ] **Une partie de sa complexité est-elle accidentelle ?**
  Interrogez chaque couche, option et mécanisme générique supplémentaire : à quelle exigence réelle répond-il (KISS, YAGNI) ?
- [ ] **Une connaissance est-elle dupliquée, ou une ressemblance fortuite est-elle partagée à tort ?**
  La connaissance dupliquée a besoin d'un seul responsable ; du code qui se ressemble mais relève de responsables différents peut rester séparé (DRY).
- [ ] **Quelles décisions qu'il contient sont difficiles à annuler, et sont-elles délibérées ?**
  Les formes des API publiques, les modèles de données et les frontières méritent un examen plus attentif que les détails internes.

- [ ] **Que disent les métriques et l'historique ?**
  Vérifiez les tendances de complexité, les métriques de couplage, le couplage de changement et les hotspots, et ajoutez une fonction d'aptitude pour toute règle qui mérite d'être conservée.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Une fonction contient deux instructions `if` et une boucle `while`, sans opérateur booléen. Quelle est sa complexité cyclomatique ?

Trois décisions plus un, donc 4.

▼ Deux services contiennent chacun une fonction identique de 20 lignes pour formater des dates. Est-ce une violation de DRY ?

Probablement pas. C'est du texte dupliqué plutôt que de la connaissance métier dupliquée, et la partager couplerait les mises en production des services ; les règles métier dupliquées, comme le calcul de l'impôt, sont une autre affaire.

▼ Pourquoi YAGNI ne s'applique-t-il pas à l'écriture de tests automatisés ?

Les tests font partie de ce qui rend le changement futur peu coûteux et sûr ; YAGNI consiste à ne pas construire de fonctionnalités ni de flexibilité spéculatives, et il repose sur de telles pratiques pour fonctionner.

▼ On impose à une équipe qu'aucune fonction ne dépasse une complexité de 10, et la complexité baisse tandis que les défauts augmentent. Que s'est-il probablement passé ?

La loi de Goodhart : les fonctions ont été découpées mécaniquement pour satisfaire la métrique, ce qui rend la logique plus difficile à suivre sans la rendre plus simple.

▼ À vous : Deux services ont chacun une vérification d'adresse de 20 lignes presque identique. L'une vérifie les adresses des restaurants pour l'enregistrement légal ; l'autre vérifie si un livreur peut atteindre une porte. Un collègue veut une bibliothèque partagée unique. Êtes-vous d'accord ?

Sans doute pas. DRY veut qu'une connaissance ait un seul foyer ; or ici le code se ressemble, mais il exprime deux règles différentes, modifiées pour des raisons différentes. Les fusionner les couplerait, et le code partagé accumulerait vite des indicateurs pour chaque cas. Gardez les deux copies, en ne partageant éventuellement qu'un analyseur de code postal neutre s'il s'agit vraiment de la même connaissance. Réexaminez la question si une troisième copie apparaît et que la connaissance partagée est claire.

SOURCES

- Fred Brooks: No Silver Bullet (UNC technical report version, 1986): https://techreports.cs.unc.edu/papers/86-020.pdf
- Martin Fowler: Yagni: https://www.martinfowler.com/bliki/Yagni.html
- Sandi Metz: The Wrong Abstraction: https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction
- Watson and McCabe: Structured Testing (NIST Special Publication 500-235, 1996): https://www.mccabe.com/pdf/mccabe-nist235r.pdf
- Martin Fowler: Beck Design Rules: https://martinfowler.com/bliki/BeckDesignRules.html

Lectures complémentaires

- *On the Criteria To Be Used in Decomposing Systems into Modules*, David L. Parnas (1972), Communications of the ACM, volume 15, issue 12. Six pages courtes qui ont introduit le masquage de l'information à travers l'exemple KWIC ; toujours le meilleur plaidoyer pour une décomposition selon les changements probables.
  https://doi.org/10.1145/361598.361623
- *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Edward Yourdon and Larry L. Constantine (1979), Yourdon Press / Prentice-Hall. Le livre qui a fait du couplage et de la cohésion un vocabulaire courant, avec l'analyse originale de leurs niveaux.
- *How Do Committees Invent?*, Melvin E. Conway (1968), Datamation (April 1968). L'énoncé original de la loi de Conway, librement accessible sur le site de l'auteur et plus nuancé que le résumé en une ligne.
  https://www.melconway.com/Home/Committees_Paper.html
- *A Philosophy of Software Design (2nd edition)*, John Ousterhout (2021), Yaknyam Press. Un livre court et engagé sur la réduction de la complexité, à l'origine des notions de modules profonds et superficiels.
  https://web.stanford.edu/~ouster/cgi-bin/book.php
- *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin (2017), Prentice Hall. Réunit en un seul endroit les principes SOLID, les principes de cohésion et de couplage des composants, et les métriques d'instabilité et de degré d'abstraction.
- *Balancing Coupling in Software Design*, Vlad Khononov (2024), Addison-Wesley. Une synthèse moderne de la conception structurée, de la connascence et de la conception pilotée par le domaine, qui juge le couplage à la fois selon sa force, la distance et la volatilité.
- *Software Engineering at Google: Lessons Learned from Programming Over Time*, Titus Winters, Tom Manshreck and Hyrum Wright (eds) (2020), O'Reilly Media. Gratuit en ligne ; traite de la loi de Hyrum et explique de façon excellente comment garder des interfaces et des dépendances saines à grande échelle.
  https://abseil.io/resources/swe-book
- *Building Evolutionary Architectures (2nd edition)*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. Montre comment protéger les décisions structurelles, comme les règles de dépendance, au moyen de fonctions d'aptitude automatisées.

Glossaire

Abstraction
Vue simplifiée de quelque chose, qui conserve ce qui compte pour un objectif donné et omet le reste, comme un « fichier » qui masque les disques, les secteurs et la mise en cache.

Abstraction qui fuit (leaky abstraction)
Abstraction dont les détails cachés affectent encore ses utilisateurs, par exemple un appel distant qui ressemble à un appel local jusqu'à ce que le réseau soit lent ou tombe en panne. Nommée par Joel Spolsky en 2002.

Backend for frontend (BFF)
API côté serveur conçue pour un type de client, comme l'app mobile, afin que chaque client reçoive des opérations et des données adaptées à ses propres besoins.

Canaux de communication
Liens individuels possibles entre les membres d'un groupe. Un groupe de n personnes en compte n(n-1)/2, de sorte que leur nombre croît à peu près comme le carré de la taille du groupe.

Charge cognitive de l'équipe
La quantité totale de connaissances et d'effort mental dont une équipe a besoin pour construire, exploiter et modifier les parties d'un système qui lui appartiennent. Lorsqu'elle dépasse la capacité de l'équipe, la qualité et la vitesse baissent.

Complexité accidentelle
Complexité introduite par la manière dont une solution est construite (outils, structure, contournements) plutôt que par le problème lui-même. Elle peut en principe être éliminée.

Complexité cyclomatique
Mesure proposée par Thomas McCabe en 1976 du nombre de chemins indépendants à travers un morceau de code, calculée à partir de son graphe de flot de contrôle ; pour du code simple, elle est égale au nombre de décisions plus un.

Complexité essentielle
Complexité propre au problème lui-même, comme les règles réelles de la fiscalité ou de l'assurance, qui ne peut être éliminée sans changer ce que fait le système.

Conception par contrat
Méthode de Bertrand Meyer consistant à spécifier chaque opération par des préconditions que l'appelant doit remplir, des postconditions que l'opération garantit et des invariants toujours vrais.

Connascence
Deux éléments sont connascents si la modification de l'un exigerait la modification de l'autre pour que le système reste correct. Manière plus fine de décrire le couplage, introduite par Meilir Page-Jones.

Connascence dynamique
Connascence qui n'apparaît qu'à l'exécution du programme, comme la dépendance à l'ordre ou au moment d'exécution, à des valeurs liées ou au partage d'une même instance d'objet par deux parties.

Connascence statique
Connascence visible dans le code source sans l'exécuter, comme l'accord sur un nom, un type, la signification d'une valeur, l'ordre des paramètres ou un algorithme.

Couplage afférent (Ca)
Nombre d'éléments extérieurs à un composant qui dépendent de lui : ses dépendances entrantes. Un Ca élevé signifie que beaucoup d'autres éléments sont affectés s'il change.

Couplage de déploiement
Situation dans laquelle deux parties doivent être publiées ensemble, ou dans un ordre fixe, parce qu'une modification de l'une casse l'autre.

Couplage efférent (Ce)
Nombre d'éléments extérieurs à un composant dont il dépend : ses dépendances sortantes. Un Ce élevé signifie qu'il est exposé aux modifications de nombreux autres.

Couplage sémantique
Dépendance à une signification ou à des règles métier partagées qui n'apparaît dans aucune interface, par exemple deux services qui doivent s'accorder sur ce que signifie « client actif ».

Couplage temporel
Dans les systèmes distribués, la dépendance au fait qu'une autre partie soit disponible et réactive au même moment, comme avec un appel synchrone. Ailleurs, le terme peut désigner un ordre d'appels imposé ou (dans l'analyse de la gestion de versions) des fichiers qui changent ensemble, ce que ce chapitre appelle couplage de changement.

Décomposition
Découpage d'un système en parties (modules) et attribution des responsabilités de chaque partie. Le critère de découpage compte davantage que le nombre de parties.

Degré (de connascence)
Nombre d'éléments qui partagent une connascence donnée. Une valeur convenue entre deux fonctions est bien plus facile à modifier qu'une valeur convenue entre quarante services.

Degré d'abstraction (A)
Métrique de Martin : nombre de classes abstraites et d'interfaces d'un composant divisé par son nombre total de classes, de 0 (entièrement concret) à 1 (entièrement abstrait).

DRY
Don't repeat yourself (ne vous répétez pas) : principe tiré de The Pragmatic Programmer selon lequel chaque élément de connaissance doit avoir une représentation unique et faisant autorité dans un système. Il porte sur la connaissance, et non sur du texte d'apparence identique.

Force (de la connascence)
La difficulté à découvrir et à refactoriser une forme de connascence. Le nom est la forme la plus faible ; les formes dynamiques comme le timing, la valeur et l'identité comptent parmi les plus fortes.

Granularité
Taille des modules ou des services par rapport à l'ensemble du système. À gros grain signifie peu de grandes parties ; à grain fin, beaucoup de petites. Chaque option a ses coûts.

Hypothèse du miroir (mirroring hypothesis)
La version issue de la recherche de la loi de Conway : les dépendances techniques d'un produit tendent à correspondre aux liens de communication et d'organisation entre les personnes qui le construisent.

Instabilité (I)
Métrique de Martin I = Ce / (Ca + Ce), de 0 (stabilité maximale : uniquement des dépendances entrantes) à 1 (instabilité maximale : uniquement des dépendances sortantes).

Invariant
Condition toujours vraie pour un module ou un objet entre deux opérations, comme « le solde d'un compte ne descend jamais sous sa limite de découvert ».

Inversion des dépendances
Fait de faire dépendre un module de haut niveau et un module de bas niveau d'une même abstraction, généralement détenue par le côté de haut niveau, de sorte que la dépendance dans le code source pointe dans le sens inverse du flux de contrôle.

KISS
Keep it simple (restez simple) : règle empirique de conception, issue de l'ingénierie des années 1960, selon laquelle il faut éviter toute complexité inutile, car elle rend un système plus difficile à comprendre, à exploiter et à réparer.

LCOM (manque de cohésion des méthodes)
Famille de métriques au niveau de la classe qui estiment la cohésion à partir de la façon dont les méthodes d'une classe partagent ses champs ; une valeur élevée laisse penser que la classe porte plusieurs responsabilités sans rapport entre elles.

Localité (de la connascence)
La proximité des éléments connascents : au sein d'une même fonction, d'un même module, d'un même composant, ou à travers un réseau. Une connascence forte n'est tolérable que si elle est locale.

Loi de Goodhart
Observation, souvent formulée ainsi : « lorsqu'une mesure devient un objectif, elle cesse d'être une bonne mesure », selon laquelle les gens optimisent le chiffre plutôt que ce qu'il était censé indiquer.

Loi de Hyrum
Observation, nommée d'après Hyrum Wright, selon laquelle, avec suffisamment d'utilisateurs d'une API, chaque comportement observable finira par être utilisé par quelqu'un, quoi que promette le contrat.

Manœuvre de Conway inversée
Façonnage délibéré des équipes et de leurs canaux de communication pour qu'ils correspondent à l'architecture que vous voulez, afin que la loi de Conway joue en votre faveur plutôt que contre vous.

Masquage de l'information (information hiding)
Principe de David Parnas (1972) selon lequel chaque module doit cacher une décision de conception difficile ou susceptible de changer, afin que cette décision puisse changer sans affecter les autres modules.

Modularité
Le degré auquel un système est divisé en parties qui peuvent être comprises, modifiées et testées en grande partie isolément. Sous-caractéristique de la maintenabilité dans l'ISO/IEC 25010.

Module logique
Unité de responsabilité dans la conception, comme Commandes ou Paiements, indépendamment de la façon dont son code est empaqueté ou déployé.

Module physique
Unité que l'on construit, versionne ou déploie, comme un fichier JAR, un paquet Python, une image de conteneur ou un service. Une même conception logique peut être empaquetée en modules physiques de bien des façons.

Module profond
Terme de John Ousterhout pour un module dont l'interface simple cache une grande quantité de fonctionnalités. Un module superficiel offre peu de comportement au regard de la complexité de son interface.

Organisation en couches (layering)
Disposer les modules en couches où chaque couche ne peut utiliser que les couches situées en dessous. Une organisation en couches stricte n'autorise que la couche immédiatement inférieure ; une organisation en couches souple autorise n'importe quelle couche inférieure.

Postcondition
Condition qu'une opération garantit vraie lorsqu'elle se termine, à condition que sa précondition ait été remplie.

Précondition
Condition qui doit être vraie avant l'appel d'une opération ; la remplir est l'obligation de l'appelant.

Principe d'équivalence réutilisation/publication (REP)

L'unité de réutilisation est l'unité de publication : les classes regroupées dans un composant doivent avoir un sens à être publiées, versionnées et réutilisées ensemble.

Principe d'inversion des dépendances (DIP)

La politique de haut niveau ne doit pas dépendre des détails de bas niveau ; les deux doivent dépendre d'abstractions, et l'abstraction doit appartenir au côté de la politique.

Principe de fermeture commune (CCP)

Les classes qui changent pour les mêmes raisons et aux mêmes moments appartiennent au même composant ; c'est le principe de responsabilité unique appliqué aux composants.

Principe de responsabilité unique (SRP)

Un module doit rendre des comptes à un seul acteur : un seul groupe de personnes qui en demandent les modifications. Le code qui change pour des raisons différentes, à la demande de personnes différentes, doit être séparé.

Principe de réutilisation commune (CRP)

Les classes utilisées ensemble vont ensemble, et les utilisateurs d'un composant ne doivent pas être contraints de dépendre de classes qu'ils n'utilisent pas.

Principe de ségrégation des interfaces (ISP)

Les clients ne doivent pas être contraints de dépendre d'opérations qu'ils n'utilisent pas ; préférez plusieurs petites interfaces propres à chaque client à une seule grande interface.

Principe de substitution de Liskov (LSP)

Tout ce qui se présente comme un sous-type doit pouvoir être utilisé partout où son supertype est attendu sans que l'appelant remarque de différence de comportement. Nommé d'après Barbara Liskov.

Principe des abstractions stables

Le principe de Robert C. Martin selon lequel un composant doit être aussi abstrait qu'il est stable, afin que les composants stables puissent malgré tout être étendus.

Principe des dépendances acycliques

Principe de Robert C. Martin selon lequel le graphe de dépendances entre composants ne doit contenir aucun cycle.

Principe des dépendances stables

Le principe de Robert C. Martin selon lequel un composant ne doit dépendre que de composants plus stables que lui : il faut dépendre dans le sens de la stabilité.

Principe ouvert-fermé (OCP)

Un module doit être ouvert à l'extension mais fermé à la modification : un nouveau comportement doit s'ajouter en ajoutant du code, par exemple une nouvelle implémentation d'une interface, et non en modifiant du code qui fonctionne. Énoncé pour la première fois par Bertrand Meyer en 1988.

Propriété des données
Règle selon laquelle chaque donnée est écrite par exactement un module ou service, que les autres doivent interroger plutôt que de lire ou de modifier directement son stockage.

Séquence principale (main sequence)
La droite A + I = 1 sur un graphique du degré d'abstraction en fonction de l'instabilité, où se situent les composants bien équilibrés. La distance à cette droite est D = |A + I - 1|.

SOLID
Moyen mnémotechnique pour cinq principes de conception orientée objet réunis par Robert C. Martin : responsabilité unique, ouvert-fermé, substitution de Liskov, ségrégation des interfaces et inversion des dépendances.

Sous-typage comportemental
Règle qui sous-tend la substitution de Liskov : un sous-type ne peut pas exiger davantage des appelants (préconditions plus fortes), promettre moins (postconditions plus faibles) ni enfreindre les invariants du supertype.

Valeur magique
Nombre ou chaîne littéral dont la signification n'est pas indiquée dans le code, par exemple un statut 3 qui signifie « expédié ». Elle crée une connascence de signification entre tout ce qui l'utilise.

YAGNI
You aren't gonna need it (vous n'en aurez pas besoin) : pratique de l'Extreme Programming consistant à ne pas construire de capacités pour des besoins seulement supposés, car elles coûtent de l'effort maintenant et de la complexité plus tard.