

BE SOFTWARE ARCHITECTURE WISE · CHAPITRE 5

Styles architecturaux I : monolithes et conceptions modulaires

La plupart des logiciels, y compris certains des produits les plus importants et qui connaissent le plus de succès, fonctionnent comme une seule application déployable. Ce chapitre explique comment de telles applications doivent être structurées à l'intérieur. Il présente le vocabulaire des styles architecturaux, des patrons, des tactiques et du quantum d'architecture, puis traite de l'architecture en couches, du monolithe modulaire, des tubes et filtres, de l'architecture micronoyau et à plug-ins, des architectures hexagonale, en oignon et Clean Architecture, et de la conception stratégique pilotée par le domaine, avant de se terminer sur les cas où un monolithe est la bonne réponse. Il s'adresse aux développeurs et aux architectes qui conçoivent ou maintiennent des applications métier et qui veulent choisir et faire respecter une structure de façon délibérée.

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

EN UN COUP D'ŒIL

Vous apprendrez à

- Distinguer les styles architecturaux, les patrons, les tactiques et les architectures de référence, et expliquer pourquoi les contraintes d'un style produisent ses qualités.
- Comparer le partitionnement technique et le partitionnement par domaine, et prévoir l'effet de chacun sur le coût d'un changement typique et sur l'organisation des équipes.
- Identifier les quanta d'architecture d'une conception et expliquer pourquoi des bases de données partagées lient des services en un seul quantum.
- Décrire les styles en couches, monolithe modulaire, tubes et filtres et micronoyau, évaluer leurs caractéristiques et reconnaître leurs anti-patrons.
- Appliquer l'architecture hexagonale, en oignon ou Clean Architecture pour garder la logique métier indépendante des frameworks et de l'infrastructure.
- Utiliser la conception stratégique pilotée par le domaine pour trouver des contextes délimités et décrire leurs relations dans une carte de contextes.
- Faire respecter des règles structurelles dans le code avec des tests d'architecture et des outils de modules comme import-linter, ArchUnit et Spring Modulith.
- Décider quand un monolithe est le bon choix, comment le faire passer à l'échelle et quels signes indiquent qu'il est temps d'en extraire des services.

Avant de commencer

Le chapitre 3, en particulier la modularité, le couplage et la cohésion, ainsi que les dépendances et l'organisation en couches ; le chapitre 4, en particulier le modèle C4. Les attributs de qualité du chapitre 2 sont utilisés tout au long du chapitre.

Durée

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

Termes clés

style architectural · patron architectural · tactique · partitionnement technique · partitionnement par domaine · quantum d'architecture · couches fermées et ouvertes · gouffre d'architecture (architecture sinkhole) · monolithe modulaire · tubes et filtres · back pressure (contre-pression) · micronoyau · ports et adaptateurs · contexte délimité

Pourquoi les styles comptent

Les chapitres 3 et 4 vous ont donné les principes d'une bonne structure (modularité, couplage et cohésion, sens des dépendances) et les outils pour la décrire. Ce chapitre et le suivant transforment ces principes en formes récurrentes que prennent les systèmes réels. Un style architectural est un ensemble nommé de contraintes portant sur les types de parties d'un système et sur la façon dont elles peuvent se connecter. Les contraintes sont l'essentiel : interdisez aux parties d'un pipeline de partager un état, ou interdisez à un module métier de toucher aux tables d'un autre, et vous obtenez des avantages prévisibles, à un prix prévisible.

Connaître les principaux styles permet à un architecte de partir d'une forme dont on sait qu'elle fonctionne pour un ensemble donné de qualités, de l'expliquer en une phrase et de repérer tôt le moment où un système s'en écarte.

Ce chapitre traite des styles dans lesquels l'application entière est une seule unité de déploiement : un seul build, une seule mise en production, généralement une seule base de données, avec des parties qui s'appellent les unes les autres en mémoire. Une telle architecture monolithique évite les pannes réseau, les indisponibilités partielles et les problèmes de cohérence des systèmes distribués, que le chapitre 6 aborde. Cela ne fait pas des monolithes des systèmes primitifs. Qu'un monolithe soit un plaisir ou un calvaire à faire évoluer dépend presque entièrement de la façon dont il est découpé à l'intérieur, et c'est le sujet de ce chapitre.

Comment s'articule le chapitre

La section 5.1 pose le vocabulaire utilisé pour chaque style des chapitres 5 et 6 : styles, patrons, tactiques et architectures de référence ; la différence entre le partitionnement technique (par couche) et le partitionnement par domaine (par capacité métier) ; le déploiement monolithique et le déploiement distribué ; et le quantum d'architecture, l'unité déployable indépendamment à laquelle s'applique un seul ensemble de qualités. Elle se termine par un tableau récapitulatif de tous les styles des deux chapitres.

Les sections 5.2 à 5.5 décrivent ensuite quatre styles selon un format homogène : définition et origine, un diagramme, le fonctionnement, un exemple, un tableau des caractéristiques, quand l'utiliser et quand l'éviter, les anti-patrons et la manière de le faire respecter dans le code. L'architecture en couches (5.2) est l'empilement familier présentation, métier et persistance, avec ses couches fermées et ouvertes et l'anti-patron du gouffre. Le monolithe modulaire (5.3) découpe une application déployable unique par domaine métier, avec des API de modules explicites et une propriété claire des données, illustré par l'expérience de Shopify et par une carte des modules d'une plateforme de livraison de repas. Le style tubes et filtres (5.4) organise le traitement en une chaîne d'étapes indépendantes, des tubes Unix aux pipelines de données. Le micronoyau (5.5) construit un noyau minimal que des plug-ins étendent, comme dans les IDE, les navigateurs et les moteurs de règles.

Les sections 5.6 et 5.7 portent sur la structure à l'intérieur des modules et entre eux plutôt que sur le déploiement. Les architectures hexagonale, en oignon et Clean Architecture (5.6) placent la logique métier au centre et font dépendre d'elle les bases de données, les interfaces utilisateur et les services externes, jamais l'inverse, ce qui rend le cœur testable et son infrastructure remplaçable. La conception stratégique pilotée par le domaine (5.7) fournit la méthode pour trouver de bonnes frontières : sous-domaines, contextes délimités et relations entre eux dans une carte de contextes. Enfin, la section 5.8 se demande quand un monolithe

est la bonne réponse, jusqu'où il peut passer à l'échelle et quels signes indiquent qu'il est temps de le découper, et propose un guide de décision et un tableau comparatif des styles du chapitre.

Un exemple fil rouge

Comme dans les chapitres précédents, une plateforme de livraison de repas revient tout au long du chapitre : les clients commandent auprès de restaurants, des livreurs livrent, les paiements passent par un prestataire externe. Dans la section 5.1, elle est classée selon quatre conceptions candidates ; en 5.2, c'est une application web classique en couches ; en 5.3, elle devient huit modules avec une règle de dépendance vérifiée dans le build ; en 5.4, ses événements de commande traversent un pipeline jusqu'à un entrepôt de données, et trouver le goulet d'étranglement fait l'objet d'un calcul commenté. Les sections suivantes reviennent à la même plateforme, de la façon la plus complète dans la carte de contextes de la section 5.7, afin que les styles puissent être comparés sur un seul système familier plutôt que sur des exemples jouets sans rapport entre eux.

L'idée à retenir

Si vous ne deviez retenir qu'une chose de ce chapitre, ce serait celle-ci : **les frontières à l'intérieur d'un système comptent plus que les frontières entre unités de déploiement**. Un monolithe découpé par domaine, avec des API de modules explicites, un seul propriétaire pour chaque table et des règles de dépendance vérifiées automatiquement, peut être modifié en toute sécurité par plusieurs équipes pendant de nombreuses années, et n'importe quel module pourra plus tard devenir un service s'il en a vraiment besoin. Un ensemble de services qui partagent une base de données et appellent les éléments internes les uns des autres reste un seul système enchevêtré, avec en plus les pannes réseau. Les styles sont des outils pour créer et préserver de bonnes frontières ; aucun d'eux n'est une fin en soi.

Deux habitudes en découlent. D'abord, choisissez le style le plus simple dont les caractéristiques satisfont les exigences architecturalement significatives du système, et consignez ce choix et ses compromis dans un enregistrement de décision. Ensuite, faites respecter les règles du style avec des tests d'architecture dans le build, car tout style s'érode lorsque ses contraintes n'existent que dans la tête des gens.

SECTION 5.1 SUR 5.8 · INTERMÉDIAIRE · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Styles architecturaux, patrons et tactiques

AVANT DE LIRE

Cette section s'appuie sur :

- 2.1 Exigences fonctionnelles et de qualité
- 3.1 Modularité
- 3.7 Loi de Conway

Termes déjà vus qu'elle utilise : Attribut de qualité, Architecturalement significatif, Connecteur, Cohésion, Loi de Conway

L'IDÉE EN BREF

Pensez aux types de maisons : une maison mitoyenne, un plain-pied, une tour d'habitation. Chaque nom en dit long à un constructeur avant même qu'un plan existe : combien d'étages, comment les pièces communiquent, ce que cela coûte, où cela fonctionne bien. Une tour convient à une ville dense et serait du gaspillage dans un village. Le logiciel a ses propres formes nommées, appelées styles architecturaux : une application en couches, un ensemble de modules dans un seul programme, une chaîne d'étapes de traitement, un groupe de services séparés. Connaître les styles, et ce à quoi chacun est bon ou mauvais, permet à une équipe de choisir rapidement une forme raisonnable et d'en parler en quelques mots.

CE QU'IL FAUT SAVOIR

- Un style architectural est un ensemble nommé de types d'éléments, de types de connecteurs et de contraintes sur leur combinaison (Garlan et Shaw) ; ce sont les contraintes qui donnent à un style ses qualités prévisibles.
- Un patron architectural, dans la tradition de *Pattern-Oriented Software Architecture* (Buschmann et al., 1996), est une solution éprouvée à un problème structurel récurrent dans un contexte donné ; une tactique est une décision de conception unique visant un attribut de qualité ; une architecture de référence est un modèle pour un domaine.
- Le partitionnement technique découpe un système selon le rôle technique (présentation, logique, données) ; le partitionnement par domaine le découpe par capacité métier (commandes, paiements, livraison). Ce choix détermine jusqu'où se propage une modification typique d'une fonctionnalité et comment les équipes peuvent être organisées.

- Un style monolithique est déployé comme une seule unité ; un style distribué est déployé comme plusieurs unités qui coopèrent à travers un réseau et hérite de tous les coûts du réseau.
- Un quantum d'architecture est une partie déployable indépendamment, dotée d'une forte cohésion fonctionnelle, données comprises ; un seul ensemble de caractéristiques d'architecture s'applique par quantum, si bien que des parties aux besoins très différents doivent former des quanta séparés.
- Chaque style est un compromis, avec des forces et des faiblesses caractéristiques ; les systèmes réels combinent des styles, et les évaluations de ce cours sont des jugements relatifs pour des mises en œuvre typiques, pas des lois.

Pourquoi les architectes nomment des formes

La plupart des systèmes ne sont pas conçus à partir de rien. Les architectes expérimentés reconnaissent qu'un nouveau problème ressemble à un problème déjà rencontré et se tournent vers une forme dont on sait qu'elle fonctionne : quelques couches pour une application métier simple, une chaîne d'étapes de traitement pour un flux de données, un noyau avec des plug-ins pour un produit que les clients étendent. Nommer ces formes donne à une équipe un vocabulaire commun. Dire «c'est un monolithe modulaire avec une communication par événements entre les modules» transmet en une phrase ce qui demanderait sinon une page de diagrammes, et cela apporte avec soi tout un savoir sur ce qui tend à bien ou mal se passer.

David Garlan et Mary Shaw ont donné une base solide à cette démarche dans *An Introduction to Software Architecture* (rapport technique de Carnegie Mellon, 1994) et dans leur livre *Software Architecture: Perspectives on an Emerging Discipline* (1996). Ils ont décrit un style architectural comme un vocabulaire de types de composants et de connecteurs, accompagné d'un ensemble de contraintes sur la façon dont ils peuvent être combinés. Le style tubes et filtres, par exemple, a une seule sorte de composant (un filtre qui transforme des données) et une seule sorte de connecteur (un tube qui les transporte dans un seul sens), et il impose aux filtres de ne pas partager d'état et de ne pas savoir qui sont leurs voisins. Roy Fielding, dont la thèse (2000) a défini REST comme un style, a formulé la même idée plus nettement : un style est un ensemble coordonné de contraintes, et ce sont les contraintes qui produisent les propriétés. Interdisez aux filtres de partager un état et vous obtenez des filtres qui peuvent être testés, réutilisés et réarrangés indépendamment ; ce n'est pas de la chance, c'est une conséquence de la règle.

C'est la façon la plus utile de lire chaque style de ce chapitre et du suivant. Un style n'est pas une image à copier. C'est un ensemble de règles que vous acceptez de suivre, choisies parce que les suivre procure des caractéristiques d'architecture précises, les attributs de qualité du

chapitre 2, moyennant un prix. Une équipe qui adopte le vocabulaire d'un style mais pas ses règles paie le prix sans obtenir les avantages.

Styles, patrons, tactiques et architectures de référence

Quatre termes apparentés sont utilisés, souvent de façon approximative, pour désigner un savoir de conception réutilisable. Ils diffèrent surtout par leur portée et par leur degré de prescription.

Un patron architectural décrit une solution éprouvée à un problème structurel récurrent dans un contexte donné. L'idée de patron vient de l'architecte du bâtiment Christopher Alexander et est entrée dans le logiciel avec les patrons de conception de Gamma, Helm, Johnson et Vlissides (1994), qui opèrent à l'échelle de quelques classes. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, de Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad et Michael Stal (1996), qu'on appelle généralement POSA, a catalogué des patrons à l'échelle de systèmes entiers : Couches (Layers), Tubes et filtres (Pipes and Filters) et Tableau noir (Blackboard) (regroupés sous le thème « de la boue à la structure »), Courtier (Broker) pour les systèmes distribués, Modèle-Vue-Contrôleur et Présentation-Abstraction-Contrôle pour les systèmes interactifs, et Micronoyau (Microkernel) et Réflexion (Reflection) pour les systèmes adaptables. Chaque patron de POSA est rédigé sous la forme contexte, problème, forces et solution, avec ses conséquences, ce qui rend les compromis explicites.

Une tactique, au sens où l'entendent Len Bass, Paul Clements et Rick Kazman dans *Software Architecture in Practice*, est plus petite : une seule décision de conception qui influe sur une réponse d'un attribut de qualité, comme ajouter un heartbeat pour détecter un serveur défaillant ou un cache pour réduire la latence. La section 2.6 a utilisé des tactiques de modifiabilité comme l'encapsulation et le recours à un intermédiaire. Les patrons regroupent plusieurs tactiques : le patron Couches combine l'encapsulation, la restriction des dépendances et le recours à un intermédiaire.

Une architecture de référence est un modèle pour tout un domaine ou toute une plateforme : elle nomme les blocs de construction typiques et la façon dont ils se connectent, afin que les projets l'adaptent plutôt que de partir d'une page blanche. AUTOSAR, l'architecture logicielle en couches utilisée pour les calculateurs électroniques des voitures, en est un exemple bien connu ; les centres d'architecture publiés par les fournisseurs de cloud en offrent beaucoup de plus petites. Une architecture de référence est généralement construite à partir de plusieurs styles et patrons à la fois.

Les praticiens ne s'accordent pas sur l'endroit où passe la frontière entre style et patron. POSA appelle Couches un patron ; Garlan et Shaw appellent les systèmes en couches un style ; Mark Richards et Neal Ford, dans *Fundamentals of Software Architecture*, réservent le mot «style» à la topologie globale d'un système et le mot «patron» aux structures réutilisables de plus bas niveau qui peuvent apparaître à l'intérieur de nombreux styles. Ce cours suit l'usage courant : **style** pour la forme globale d'un système ou d'une grande partie de celui-ci, **patron** pour une solution réutilisable qui peut se situer à n'importe quel niveau. Ne dépensez pas d'énergie sur la frontière ; consacrez-la aux contraintes et à leurs conséquences.

Quatre sortes de savoir de conception réutilisable

Terme	Portée	Exemple	Question à laquelle il répond
Style architectural	Le système entier ou une grande partie	En couches, tubes et filtres, microservices	Quelles sortes de parties et de connexions sont autorisées, et selon quelles règles ?
Patron architectural	N'importe quel niveau structurel	Courtier (Broker), Modèle-Vue-Contrôleur, Micronoyau (Microkernel)	Quelle est une solution éprouvée à ce problème structurel récurrent, et que coûte-t-elle ?
Tactique	Une seule décision de conception	Heartbeat, cache, encapsulation, nouvelle tentative avec backoff	Quelle décision unique fait évoluer cet attribut de qualité dans le sens dont nous avons besoin ?
Architecture de référence	Un domaine ou une plateforme	AUTOSAR pour l'électronique automobile ; les architectures publiées par les fournisseurs de cloud	Que contiennent habituellement les systèmes de ce type, et comment sont-ils habituellement organisés ?

Deux façons de découper un système : partitionnement technique ou par domaine

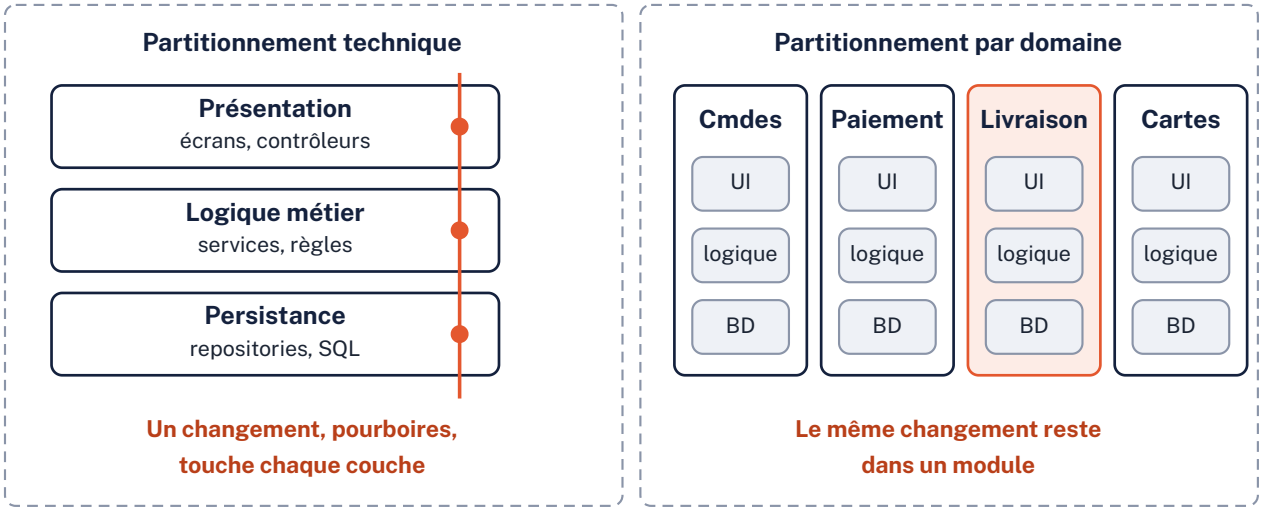
La première décision structurelle de tout style est la façon de découper le système au niveau supérieur. Il existe deux réponses de base. Le partitionnement technique regroupe le code selon le rôle technique : tout ce qui concerne l'interface utilisateur à un endroit, toutes les règles métier à un autre, tout l'accès à la base de données à un troisième. Le partitionnement par domaine regroupe le code par capacité métier : tout ce qui concerne les commandes à un endroit, les paiements à un autre, la livraison à un troisième, chacun contenant l'interface utilisateur, la logique et l'accès aux données dont il a besoin.

La différence apparaît dès qu'une fonctionnalité change. Supposons qu'une entreprise de livraison de repas permette aux clients d'ajouter un pourboire pour le livreur. Dans un système partitionné techniquement, ce changement demande un nouveau champ sur un écran (présentation), une nouvelle règle pour les plafonds de pourboire et les versements (logique métier) ainsi qu'une nouvelle colonne et une nouvelle requête (persistance) : trois parties, souvent prises en charge par trois personnes ou équipes différentes. Dans un système partitionné par domaine, le même changement se concentre presque entièrement dans le module de livraison. La plupart des demandes de changement métier sont formulées en termes de domaine, si bien que le partitionnement par domaine tend à garder un changement au même endroit. Le partitionnement technique garde au même endroit un changement *technique*, comme le remplacement de la bibliothèque d'accès à la base de données, mais ces changements sont plus rares.

Ce choix façonne aussi l'organisation. Selon la loi de Conway, un système partitionné techniquement tend à être construit par une équipe frontend, une équipe backend et une équipe base de données, et chaque fonctionnalité exige un passage de relais entre elles. Un système partitionné par domaine peut être construit par des équipes qui prennent chacune en charge une capacité métier de bout en bout. Martin Fowler a formulé la même recommandation en 2015 : les couches présentation, domaine et données sont une bonne façon d'organiser un petit logiciel, mais le niveau supérieur d'un système plus grand doit être découpé en modules orientés domaine, chacun organisé en couches à l'intérieur si cela aide.

Aucune des deux n'est mauvaise. Le partitionnement technique est simple, familier et pris en charge par la structure de dossiers par défaut de presque tous les frameworks web, et il convient aux petits systèmes où une seule équipe fait tout. Le partitionnement par domaine demande davantage de réflexion en amont, car il faut trouver de bonnes frontières de domaine (le sujet de la section 5.7), mais il s'adapte mieux à la taille du système et au nombre d'équipes.

La même fonctionnalité selon deux partitionnements



Avec un partitionnement technique, une fonctionnalité typique traverse chaque couche ; avec un partitionnement par domaine, elle se concentre dans un seul module, qui peut lui-même rester organisé en couches à l'intérieur.

Styles monolithiques et distribués

La seconde distinction de premier niveau concerne le déploiement. Une unité de déploiement est ce qui est construit, mis en production et démarré d'un seul tenant : un exécutable unique, une image de conteneur, une archive web. Dans une architecture monolithique, l'application entière est une seule unité de déploiement, généralement avec une seule base de données. Elle peut tourner en de nombreuses copies identiques derrière un équilibreur de charge, mais chaque copie contient tout le code, et mettre en production le moindre changement signifie tout mettre en production. Dans une architecture distribuée, le système se compose de plusieurs unités de déploiement qui coopèrent en échangeant des messages à travers un réseau, et chacune peut être mise en production et passer à l'échelle séparément.

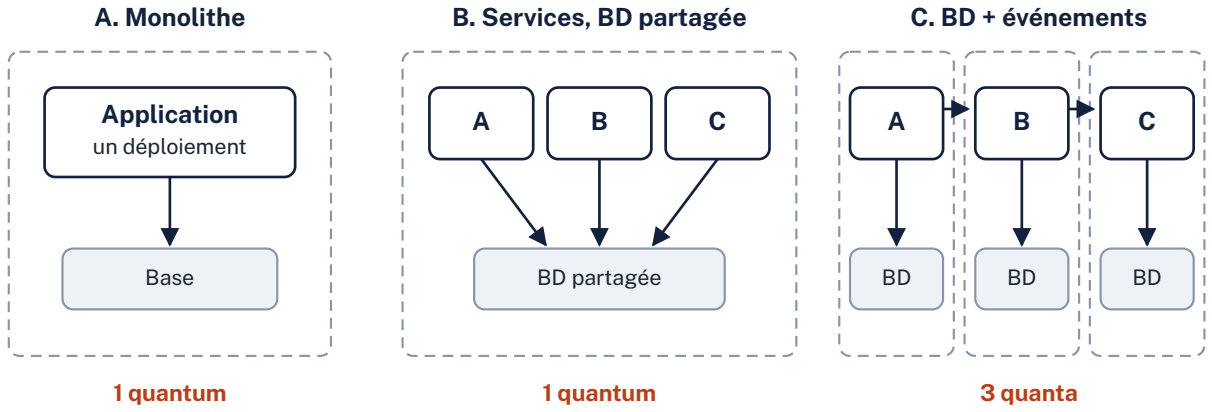
Deux précisions évitent bien des confusions. D'abord, «monolithe» décrit le déploiement, pas la qualité. Un monolithe peut être magnifiquement modulaire ou être une grosse boule de boue (le nom donné par Brian Foote et Joseph Yoder à un système sans structure discernable), et il en va de même d'un système distribué : bien des équipes ont appris que découper une boule de boue en services donne une boule de boue distribuée, ce qui est pire. Ensuite, être distribué n'est pas une amélioration gratuite. Chaque appel qui traverse un réseau peut être lent, échouer à mi-chemin ou arriver deux fois, les données ne peuvent plus être modifiées dans une seule transaction locale, et le système exige une supervision, une automatisation du déploiement et des compétences d'astreinte dont un monolithe peut se passer. Le chapitre 6 détaille ces coûts. Ce chapitre traite des styles qui les évitent : tout tourne dans une seule unité de déploiement, et l'architecture réside dans la façon dont le code y est organisé.

Le quantum d'architecture

L'opposition entre monolithique et distribué est un premier découpage utile, mais trop grossier pour les décisions réelles. Neal Ford, Rebecca Parsons et Patrick Kua ont introduit une idée plus précise dans *Building Evolutionary Architectures* (2017), affinée par Ford, Richards, Sadalage et Dehghani dans *Software Architecture: The Hard Parts* (2021) : le quantum d'architecture. Un quantum est une partie du système qui peut être déployée indépendamment, qui présente une forte cohésion fonctionnelle (elle accomplit une seule tâche cohérente pour le métier) et qui inclut tout ce dont elle a besoin pour fonctionner, en particulier sa base de données. La définition ultérieure ajoute que les parties d'un quantum sont couplées statiquement (elles partagent du code, des schémas ou une infrastructure qui doivent être déployés ensemble) et que les appels synchrones lient les quanta entre eux sur le plan opérationnel tant que dure l'appel.

Pourquoi est-ce important ? Parce que les caractéristiques d'architecture s'appliquent par quantum. Un quantum passe à l'échelle, est déployé, sécurisé et rétabli comme un tout : il a donc un seul niveau de scalabilité, un seul niveau de disponibilité, une seule cadence de mise en production. Si deux parties d'un système ont besoin de caractéristiques très différentes, par exemple un flux de positions des livreurs qui doit absorber des milliers de mises à jour par seconde et une tâche mensuelle de facturation des restaurants, elles ne peuvent les obtenir qu'en se trouvant dans des quanta différents. À l'inverse, découper du code en plusieurs services ne crée pas plusieurs quanta s'ils partagent toujours une seule base de données : aucun d'eux ne peut changer le schéma ni être restauré seul, si bien qu'ils tiennent ou tombent ensemble. Un monolithe est, par définition, un seul quantum.

Compter les quanta



Les pointillés délimitent chaque quantum : ce qui peut être déployé et exécuté seul, données comprises

Une base de données partagée lie des services en un seul quantum ; des quanta séparés exigent un déploiement séparé et des données séparées.

EXEMPLE COMMENTÉ

Exemple commenté : classer quatre conceptions

Une entreprise de livraison de repas examine quatre conceptions candidates pour sa plateforme. Pour chacune, l'architecte pose trois questions : comment le niveau supérieur est-il partitionné, combien y a-t-il d'unités de déploiement, et combien de quanta ?

Conception 1 : une application Django avec les dossiers `views`, `services` et `models`, une base de données PostgreSQL	Partitionnement technique ; une unité de déploiement ; un quantum. Simple, mais chaque fonctionnalité traverse les trois dossiers.
Conception 2 : une application avec les paquets `orders`, `payments`, `delivery` et `menus`, chacun avec ses propres tables, un serveur de base de données	Partitionnement par domaine ; une unité de déploiement ; un quantum. Les changements restent locaux, et les modules pourraient être séparés plus tard.
Conception 3 : quatre services (commandes, paiements, livraison, cartes) qui lisent et écrivent tous dans les mêmes tables de la base de données	Partitionnement par domaine de nom seulement ; quatre unités de déploiement mais un seul quantum, car un changement de schéma ou une panne de la base de données touche les quatre. Tous les coûts de la distribution, peu de ses avantages.
Conception 4 : les mêmes quatre services, chacun avec sa propre base de données, qui échangent des événements par l'intermédiaire d'un courtier de messages (message broker)	Partitionnement par domaine ; quatre unités de déploiement ; quatre quanta. Chacun peut passer à l'échelle et être mis en production indépendamment, au prix de pannes réseau et d'une cohérence à terme entre eux.

Les conceptions 1 et 2 coûtent autant à exploiter, mais la conception 2 rend les changements typiques moins coûteux et préserve les options futures.

La conception 3 est le cas d'alerte : le nombre d'unités de déploiement dit peu de chose ; le nombre de quanta, qui dépend des données et du couplage, dit ce qui peut réellement évoluer indépendamment.

SI VOUS DÉBUTEZ

Pas à pas : un quantum, un ensemble de qualités

Le point qui fait le plus souvent trébucher ici est que les caractéristiques d'architecture appartiennent à tout un quantum d'architecture, et non à chaque fonctionnalité qu'il contient. Un quantum est déployé, passe à l'échelle et est rétabli d'un seul bloc : tout ce qu'il contient partage donc un seul niveau de scalabilité, un seul niveau de disponibilité et une seule cadence de mise en production. Suivez les deux fonctionnalités contrastées de la section à travers un même monolithe.

- La plateforme de livraison de repas tourne comme une seule unité de déploiement avec une seule base de données. Deux de ses fonctionnalités ont des besoins très différents : un flux de positions des livreurs qui doit absorber des milliers de mises à jour par seconde, et une tâche mensuelle de facturation des restaurants.
- À l'heure du dîner, le flux de positions est surchargé. Le seul moyen d'ajouter de la capacité est de faire tourner davantage de copies de l'application entière, et chaque copie embarque aussi le code de facturation, qui n'a besoin d'aucune capacité supplémentaire.
- Si la tâche de facturation a un bug qui épuise la mémoire du processus, elle peut arrêter aussi le flux de positions, car les deux tournent dans le même processus.
- Lorsque le code de facturation a besoin d'une petite correction, celle-ci ne peut pas être mise en production seule : l'application entière est construite, testée et redéployée, flux de positions compris.
- Pour donner au flux sa propre mise à l'échelle, ses propres défaillances et ses propres mises en production, il doit devenir un quantum séparé, déployé seul avec ses propres données. Déplacer son code dans un service séparé ne suffit pas s'il partage toujours la base de données, car alors aucune des deux parties ne peut changer le schéma ni être restaurée seule.
- **La leçon :** ne demandez pas combien de services compte une conception ; demandez quelles parties ont besoin de caractéristiques très différentes. Ces parties ont besoin de quanta séparés, et tout le reste peut rester ensemble dans un seul.

Comment lire les sections de styles qui suivent

Les sections 5.2 à 5.5 et la majeure partie du chapitre 6 décrivent chacune un style, toujours dans le même ordre, afin que les styles puissent être comparés : une définition et l'origine du style ; un diagramme ; son fonctionnement ; un exemple concret de système ; un tableau qui

évalue ses principales caractéristiques ; quand l'utiliser et quand l'éviter ; les anti-patrons courants ; et la manière de le faire respecter dans le code, car un style qui n'est pas vérifié s'érode, comme l'a décrit la section 1.7.

Les évaluations des caractéristiques (Bas, Moyen, Élevé, chacune avec une justification) suivent l'approche des comparaisons de styles de Richards et Ford, mais ce sont les jugements propres à ce cours. Lisez-les comme relatives et typiques : elles comparent les styles tels qu'ils sont habituellement construits, par une équipe compétente, pour un système métier typique. Une équipe disciplinée peut réaliser un monolithe en couches qui se déploie plusieurs fois par jour, et une équipe négligente peut construire des microservices impossibles à déployer. Les évaluations vous indiquent où le style vous aide et où vous devrez lutter contre lui.

Deux sections de ce chapitre ne portent pas sur des styles de déploiement. Les architectures hexagonale, en oignon et Clean Architecture (section 5.6) décrivent comment organiser l'intérieur d'une application ou d'un service pour que la logique métier ne dépende pas de la technologie. La conception pilotée par le domaine (section 5.7) est une façon de trouver les frontières de domaine sur lesquelles s'appuient les monolithes modulaires et les microservices. Les deux s'appliquent aussi bien à l'intérieur d'un monolithe que d'un système distribué.

Les styles des chapitres 5 et 6 en un coup d'œil

Style	Partitionnement principal	Déploiement	Usage typique
En couches (5.2)	Technique	Monolithique	Applications métier petites à moyennes construites par une seule équipe
Monolithe modulaire (5.3)	Domaine	Monolithique	Produits en croissance, avec plusieurs équipes qui veulent un seul déploiement
Tubes et filtres (5.4)	Technique (étapes de traitement)	Monolithique ou distribué	Traitement de données : ETL, multimédia, journaux, compilateurs
Micronoyau (5.5)	Noyau technique, plug-ins souvent par domaine	Généralement monolithique	Produits étendus par des clients ou des tiers ; moteurs de règles
Hexagonale, en oignon, Clean (5.6)	Structure interne : le domaine au centre	L'un ou l'autre	Toute application dont la logique doit survivre à ses frameworks
À base de services (6.2)	Domaine	Distribué, quelques services à gros grain	Une étape pragmatique au-delà d'un monolithe, partageant souvent une base de données
Architecture orientée services (6.2)	Technique (orchestration, services d'entreprise)	Distribué	Intégration d'entreprise, aujourd'hui largement historique
Microservices (6.3)	Domaine	Distribué, nombreux services à grain fin	De nombreuses équipes qui doivent mettre en production indépendamment, à grande échelle
Pilotée par les événements (6.4)	Technique (traitement des événements)	Distribué	Réactions rapides et découplées aux événements métier
Serverless et basée sur l'espace (6.8)	Variable	Distribué	Charge en pics ou extrême ; charges de travail payées à l'usage

Les styles se combinent, et ce ne sont pas des objectifs

Les systèmes réels sont hybrides. Un monolithe modulaire peut être organisé en couches à l'intérieur de chaque module ; l'un de ses modules peut exécuter une tâche nocturne en tubes et filtres ; un microservice peut être construit en style hexagonal ; un système piloté par les événements peut utiliser un micronoyau pour permettre aux clients d'ajouter leurs propres règles. Nommer le style de chaque partie reste utile, car chaque partie hérite alors d'un ensemble connu de règles et de compromis.

L'erreur courante consiste à traiter un style comme une fin en soi : «nous passons aux microservices» comme stratégie, plutôt que comme moyen d'atteindre une qualité énoncée, par exemple le déploiement indépendant pour douze équipes. Partez des exigences architecturalement significatives du chapitre 2, choisissez le style le plus simple dont les caractéristiques les satisfont, consignez ce choix dans un enregistrement de décision et faites respecter ses règles. La section 5.8 propose un guide de décision pour les styles de ce chapitre.

ATTENTION

Malentendus courants

- **Monolithe veut dire système hérité.** Cela veut dire une seule unité de déploiement. De nombreux produits à succès fonctionnent comme des monolithes bien structurés.
- **Plus de services veut dire plus modulaire.** La modularité est une affaire de frontières et de dépendances ; des services qui partagent une base de données forment un seul quantum, quel que soit leur nombre.
- **Un style se choisit une fois pour toutes.** Les styles évoluent : une application en couches peut devenir un monolithe modulaire, puis perdre un module au profit d'un service séparé lorsque ses besoins divergent.
- **Utiliser les dossiers du framework, c'est choisir un style.** Une structure de dossiers par défaut est un partitionnement technique choisi par les auteurs du framework pour de petites applications, pas une décision concernant votre système.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Selon Garlan et Shaw, quels sont les deux éléments qui définissent un style architectural ?**

Un vocabulaire de types de composants et de connecteurs, et un ensemble de contraintes sur la façon dont ils peuvent être combinés. Ce sont les contraintes qui produisent les qualités caractéristiques du style.

▼ **Quelle est la différence entre une tactique et un patron ?**

Une tactique est une seule décision de conception visant une réponse d'un attribut de qualité, comme un heartbeat ou un cache. Un patron est une solution documentée plus vaste, qui combine généralement plusieurs tactiques.

▼ **Pourquoi quatre services qui partagent une seule base de données pourraient-ils compter comme un seul quantum d'architecture ?**

Parce qu'aucun d'eux ne peut changer le schéma partagé, être restauré ou passer à l'échelle au niveau des données sans affecter les autres ; en pratique, ils sont déployés et tombent en panne ensemble, si bien qu'ils partagent un seul ensemble de caractéristiques opérationnelles.

▼ **Quelle sorte de changement le partitionnement technique garde-t-il au même endroit ?**

Un changement technique, comme le remplacement de la bibliothèque d'accès à la base de données ou du framework d'interface utilisateur. Les changements métier typiques, en revanche, traversent toutes les couches techniques.

▼ **À vous : Une plateforme de billetterie a trois services déployés séparément, Réservation, Paiements et Rapports, qui utilisent tous une même base de données partagée. Comment est-elle partitionnée, combien compte-t-elle d'unités de déploiement et de quanta ? Pourquoi la lourde charge de Rapports pourrait-elle avoir de l'importance ?**

Il est partitionné par domaine, avec trois unités de déploiement mais un seul quantum, car la base de données partagée les lie : un quantum inclut ses données. Le système n'est donc pas aussi indépendant qu'il y paraît. Les requêtes lourdes de Rapports peuvent ralentir Réservation le jour de la mise en vente, et un seul ensemble de caractéristiques de qualité s'applique aux trois. Pour donner à Rapports des besoins différents, faites-en un quantum séparé, par exemple avec son propre magasin de données de reporting.

SOURCES

- Garlan and Shaw: An Introduction to Software Architecture (SEI, 1994): https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- Roy Fielding: Architectural Styles and the Design of Network-based Software Architectures, chapter 1: https://ics.uci.edu/~fielding/pubs/dissertation/software_arch.htm
- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: Architecture styles: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/

SECTION 5.2 SUR 5.8 · INTERMÉDIAIRE · 24 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Architecture en couches

AVANT DE LIRE

Cette section s'appuie sur :

- 3.4 Dépendances
- 5.1 Styles et patrons

Termes déjà vus qu'elle utilise : Partitionnement technique, Unité de déploiement, Quantum d'architecture (architecture quantum), Loi de Conway

L'IDÉE EN BREF

Pensez à un restaurant. Les clients ne parlent qu'aux serveurs. Les serveurs transmettent les commandes à la cuisine et n'entrent jamais eux-mêmes dans la réserve. Les cuisiniers obtiennent les ingrédients auprès du personnel de la réserve, et seul le personnel de la réserve traite avec les fournisseurs. Chaque groupe a une seule tâche et ne parle qu'au groupe voisin : vous pouvez donc former à nouveau les serveurs sans toucher à la cuisine. Une application en couches fonctionne de la même manière : les écrans parlent aux règles métier, les règles métier parlent au code d'accès aux données, et seul ce code parle à la base de données.

CE QU'IL FAUT SAVOIR

- Une architecture en couches organise le code en couches horizontales selon le rôle technique, généralement présentation, métier, persistance et base de données, et ne permet à chaque couche de dépendre que des couches situées en dessous.
- Les couches fermées ne doivent pas être contournées, ce qui crée des couches d'isolation : un changement à l'intérieur d'une couche ne se propage pas aux autres tant que son interface tient. Une couche ouverte, comme une couche de services partagés, peut être contournée.
- L'organisation en couches est un partitionnement technique : une fonctionnalité métier typique touche donc toutes les couches, ce qui ralentit le changement et incite à des structures d'équipes découpées par couche.
- L'anti-patron du gouffre d'architecture désigne les requêtes qui traversent des couches qui ne font que les transmettre ; en règle générale, une minorité de telles requêtes est normale, une large majorité suggère que le style n'est pas le bon.

- Le style obtient une note élevée en simplicité et en faible coût, et une note basse en déployabilité, scalabilité, élasticité et tolérance aux pannes, car le système entier est une seule unité de déploiement.
- Les règles des couches s'érodent si on ne les fait pas respecter automatiquement, par exemple avec import-linter en Python ou ArchUnit en Java, sous forme de tests qui font échouer le build.

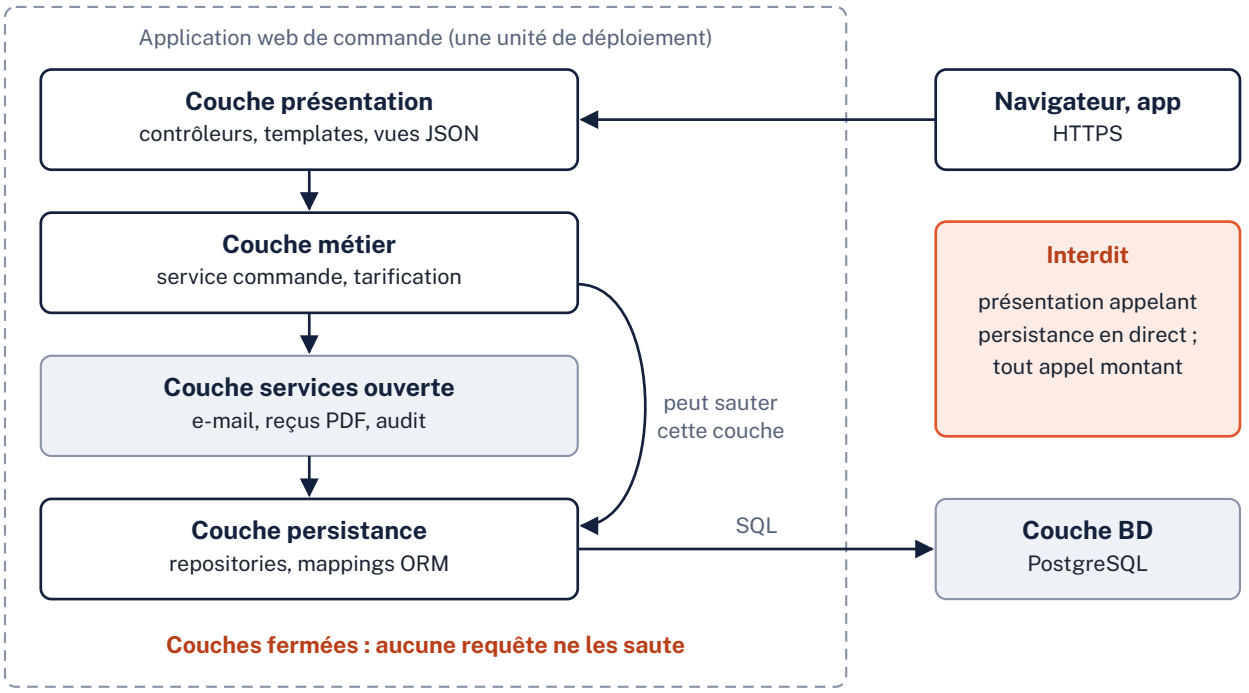
Définition et origine

Une architecture en couches organise une application en couches horizontales, dont chacune a une responsabilité technique unique, offre des services à la couche supérieure et utilise les services de la couche inférieure. C'est le style le plus utilisé pour les applications métier, souvent par défaut plutôt que par décision, car la plupart des frameworks web génèrent un projet avec des dossiers pour les vues, la logique métier et les modèles.

L'idée est ancienne et vient du logiciel système. Edsger Dijkstra a décrit le système d'exploitation multiprogrammé THE (Communications of the ACM, 1968) comme une hiérarchie de niveaux, chacun construit uniquement sur ceux qui se trouvent en dessous, afin que chaque niveau puisse être raisonné et testé isolément. Les piles de protocoles réseau comme le modèle de référence OSI appliquent le même principe. Dans le logiciel de gestion, l'organisation en couches est arrivée avec les systèmes client-serveur et à trois niveaux des années 1990, et POSA (1996) l'a cataloguée sous le nom de patron Couches (Layers). Garlan et Shaw ont rangé les systèmes en couches parmi leurs styles de base, avec la contrainte que les interactions soient limitées aux couches adjacentes.

L'application métier typique comporte quatre couches. La **couche de présentation** gère l'interface utilisateur et le protocole : requêtes HTTP, écrans, JSON. La **couche métier** contient les règles et les cas d'utilisation : comment le prix d'une commande est calculé, quand elle peut être annulée. La **couche de persistance** transforme les objets métier en données stockées et inversement, généralement au moyen de repositories et d'un outil de mapping objet-relationnel (ORM). La **couche base de données** est la base de données elle-même. Les petites applications fusionnent parfois métier et persistance ; les plus grandes ajoutent des couches, comme une couche application qui coordonne les cas d'utilisation au-dessus d'une couche domaine qui contient le modèle central. Le style ne fixe pas le nombre de couches ; il fixe le sens des dépendances.

Une application de commande de repas en couches



Les appels vont uniquement vers le bas ; les couches fermées ne peuvent pas être sautées, alors que la couche de services ouverte peut l'être.

Fonctionnement : couches fermées, couches ouvertes et isolation

Chaque couche peut être fermée ou ouverte. Une requête doit traverser une couche fermée : la couche de présentation ne peut pas passer outre la couche métier pour atteindre la couche de persistance, même quand ce serait plus rapide à écrire. Une couche ouverte peut être contournée. L'exemple habituel est une couche de services techniques partagés, comme l'envoi d'e-mails, la génération de reçus PDF ou l'écriture d'enregistrements d'audit. Seules certaines opérations métier en ont besoin : forcer chaque requête à traverser cette couche ajouterait donc du code de relais inutile. Dans le diagramme, la couche métier peut appeler la couche de services ou passer directement à côté pour atteindre la persistance.

Les couches fermées créent des couches d'isolation. Comme la couche de présentation ne connaît que l'interface de la couche métier, l'équipe peut remplacer l'outil de mapping objet-relationnel, restructurer les tables ou passer d'un produit de base de données à un autre sans toucher à un seul contrôleur ni à un seul template, tant que l'interface de la couche métier reste la même. C'est le masquage de l'information (David Parnas, 1972) appliqué à toute une strate du code, et c'est le principal avantage qu'offre le style. Chaque couche ouverte, et chaque exception qui permet à une couche d'en sauter une autre, l'affaiblit : si les contrôleurs interrogent directement la base de données, un changement de schéma se propage désormais aussi à la couche de présentation.

Le sens des dépendances est l'autre moitié de la règle. Les couches inférieures ne doivent rien savoir des couches supérieures : la couche de persistance ne doit jamais importer un contrôleur, et la couche métier ne doit jamais construire de HTML. Lorsqu'une couche inférieure doit notifier une couche supérieure, elle le fait au moyen d'une abstraction qu'elle possède, comme une interface de rappel (callback) ou un événement, afin que la dépendance pointe toujours vers le bas dans le code. La section 3.4 traite en profondeur du sens des dépendances, des cycles et de l'organisation en couches stricte ou souple.

Il est utile de bien distinguer couches et niveaux. Une couche est un regroupement logique dans le code. Un niveau (tier) est une frontière physique à l'exécution, comme un serveur ou un processus séparé. Un système classique à trois niveaux fait tourner le navigateur, un serveur d'applications et un serveur de base de données comme trois niveaux, alors que le serveur d'applications à lui seul peut contenir trois ou quatre couches. Transformer une couche en niveau place un réseau entre deux couches qui s'appelaient auparavant en mémoire, ce qui multiplie le coût de chaque appel ; cela vaut rarement la peine en soi.

Exemple : une application web classique

Prenons le site de commande d'une entreprise régionale de livraison de repas : les clients parcourent les cartes, passent commande et paient ; le personnel des restaurants voit les commandes arriver. Une équipe de cinq personnes le construit avec un framework web courant et une base de données PostgreSQL. Une structure en couches est le point de départ évident, et l'organisation du code rend les couches visibles.

```
foodshop/
  presentation/          # HTTP only: parse requests, render responses
    order_views.py
    menu_views.py
    templates/
  business/              # rules and use cases; no HTTP, no SQL
    order_service.py
    pricing.py
  services/              # open layer: shared technical services
    email.py
    receipts.py
  persistence/           # all database access
    order_repository.py
    menu_repository.py
    models.py
  settings.py
```

Structure des paquets de l'application de commande de repas en couches : un paquet de premier niveau par couche.

EXEMPLE COMMENTÉ

Exemple commenté : suivre une requête à travers les couches

Un client appuie sur **Commander**. La requête arrive sous la forme `POST /orders`, avec le panier et l'adresse de livraison.

Présentation	`order_views.place_order` analyse le JSON, vérifie qu'il est bien formé et que l'utilisateur est connecté, et appelle `order_service.place(customer_id, items, address)`. Elle ne sait rien des tables ni des prix.
Métier	`order_service.place` vérifie que le restaurant est ouvert, calcule le prix du panier avec `pricing.py` (remises, frais de livraison, taxes) et demande à `order_repository` d'enregistrer la commande. Il appelle `receipts.send` dans la couche de services ouverte pour envoyer un reçu par e-mail.
Persistance	`order_repository.save` fait correspondre la commande à des lignes dans `orders` et `order_lines` et les écrit dans une seule transaction de base de données.
Base	PostgreSQL valide la transaction et rend la main. Le résultat remonte : le service renvoie la nouvelle commande, la vue renvoie `201 Created` avec le numéro de commande.
Demande de changement	L'équipe produit demande un pourboire pour le livreur. Le développeur ajoute un champ de pourboire à la requête et à la réponse (présentation), une règle de pourboire maximal et le pourboire dans le total (métier), ainsi qu'une colonne et un mapping (persistance), plus une migration (base de données). Quatre couches pour une petite fonctionnalité.

Chaque couche a accompli une seule sorte de tâche, et chacune pouvait être testée seule, la couche inférieure étant remplacée par une doublure de test.

Le pourboire montre le coût du partitionnement technique : même un petit changement métier traverse chaque couche, si bien qu'il touche plus de fichiers, plus de relecteurs et, si les équipes sont découpées par couche, plus d'équipes.

Les conséquences du partitionnement technique

Comme l'organisation en couches est un partitionnement technique, le domaine métier est éparpillé à travers les couches. Il n'existe pas d'endroit unique appelé «livraison» : il y a des contrôleurs de livraison, des services de livraison et des repositories de livraison, entremêlés avec tout le reste dans chaque couche. Un développeur qui veut comprendre le fonctionnement des pourboires doit tous les parcourir. À mesure que le système grandit, chaque couche devient un vaste ensemble peu cohésif de code sans rapport : la couche métier contient côte à

côte les règles des commandes, des paiements, des cartes et des livreurs, et rien n'empêche le code de paiement d'appeler le code des cartes en profondeur. Avec le temps, les couches restent propres, mais l'intérieur de chaque couche devient une grosse boule de boue.

Il y a aussi un coût organisationnel. Organiser les équipes par couche (une équipe frontend, une équipe backend, une équipe base de données) signifie que presque chaque fonctionnalité exige un passage de relais et une file d'attente entre équipes, que la loi de Conway grave ensuite dans l'architecture. Martin Fowler qualifie pour cette raison les équipes organisées par couche d'anti-patron : les développeurs n'ont pas besoin d'être full-stack, mais les équipes, si. Une architecture en couches n'oblige pas à découper les équipes par couche, mais elle en fait la voie de moindre résistance.

Enfin, un monolithe en couches est un seul quantum d'architecture : une seule unité de déploiement et généralement une seule base de données. Chaque changement, aussi petit soit-il, implique de construire, tester et redéployer l'application entière, et l'application entière passe à l'échelle et tombe en panne d'un seul bloc. C'est souvent parfaitement acceptable, mais cela fixe un plafond.

Un autre découpage : l'architecture en tranches verticales

L'**architecture en tranches verticales** (vertical slice architecture) de Jimmy Bogard (2018) organise le code autour des requêtes individuelles et non des couches. Chaque fonctionnalité, comme *passer commande* ou *annuler la commande*, contient son propre traitement, sa validation et son accès aux données, de l'entrée de la requête jusqu'à la base de données. Son objectif est de « minimiser le couplage entre les tranches et maximiser le couplage dans une tranche », pour qu'un changement touche une fonctionnalité et non toutes les couches. Le prix à payer est la structure partagée qu'offraient les couches : la duplication peut s'accumuler et les gestionnaires peuvent enfler, et Bogard précise que l'approche suppose une équipe qui reconnaît les code smells et qui refactorise. Les tranches se combinent bien aussi avec les modules : les modules divisent les domaines, et les tranches divisent les fonctionnalités à l'intérieur d'un module.

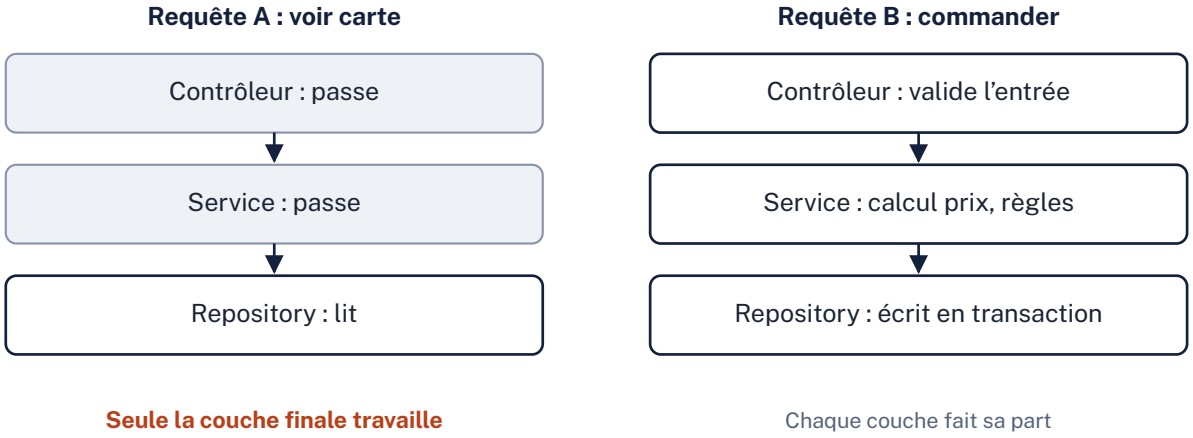
Le gouffre d'architecture et autres anti-patrons

L'anti-patron du gouffre d'architecture, un nom popularisé par Mark Richards, désigne les requêtes qui traversent des couches qui ne font rien d'autre que les transmettre. Consulter une carte en est le cas typique : le contrôleur appelle une méthode de service qui appelle une méthode de repository qui exécute une requête, et le contrôleur comme le service n'ajoutent rien. Chaque couche de simple relais ajoute du code à écrire, à tester et à lire, ainsi qu'un peu de latence. Un certain relais est le prix de l'isolation et ne pose pas de problème. Richards propose une règle empirique : si environ un cinquième des requêtes sont de simples relais, c'est

normal ; si c'est le cas de la plupart, l'application consiste probablement surtout en accès aux données avec peu de logique métier, et l'organisation en couches n'est pas le bon style, ou bien les couches devraient être ouvertes. Ouvrir des couches fait cependant renoncer à l'isolation : décidez-le donc délibérément et consignez pourquoi.

D'autres anti-patrons courants méritent d'être reconnus tôt. **Les couches qui fuient** : du SQL dans les contrôleurs, des objets de requête HTTP passés aux services, ou des entités ORM rendues directement en JSON, si bien qu'un changement dans une couche se propage aux autres. **Les dépendances montantes** : un repository qui appelle un service, ou un objet de domaine qui met en forme du HTML. **La couche métier anémique** : des services qui ne font que déplacer des données pendant que les vraies règles vivent dans les contrôleurs ou dans des procédures stockées ; Fowler a nommé en 2003 l'anti-patron voisin Anemic Domain Model (modèle de domaine anémique). **Le service divin** : un unique `OrderService` de plusieurs milliers de lignes que tous les autres services appellent. **La couche utilitaire dont tout dépend** : un paquet `common` qui accumule peu à peu de la logique métier et couple tout à tout.

Gouffre ou organisation en couches utile



Les requêtes de simple relais coûtent du code et de la latence sans ajouter de comportement ; quelques-unes sont normales, une majorité est un signal d'alerte.

Caractéristiques du style en couches

Caractéristique	Évaluation	Justification
Simplicité	Élevé	Familier à presque tous les développeurs et structure par défaut de la plupart des frameworks ; facile à expliquer et à démarrer.
Coût global	Élevé (peu coûteux)	Une seule base de code, un seul déploiement, une seule base de données ; infrastructure et exploitation minimales.
Testabilité	Moyen	Chaque couche peut être testée en remplaçant la couche inférieure par une doublure de test, mais des tests réalistes exigent souvent toute la pile et une base de données.
Modifiabilité pour les changements métier	Bas à moyen	Les changements techniques restent dans une couche, mais la plupart des changements de fonctionnalités traversent toutes les couches.
Déployabilité	Bas	Tout changement implique de redéployer l'application entière ; le risque de mise en production croît avec la taille.
Scalabilité et élasticité	Bas	L'application entière passe à l'échelle comme une seule unité ; vous ne pouvez pas faire passer à l'échelle uniquement la partie chargée, et le démarrage d'une grande application ralentit la mise à l'échelle élastique.
Tolérance aux pannes	Bas	Une seule fuite de mémoire ou une requête qui s'emballe peut faire tomber tout le processus ; il n'y a aucune isolation entre les fonctionnalités.
Performances	Moyen	Les appels internes au processus sont rapides, mais les couches supplémentaires et le mapping des objets ajoutent un surcoût, et les gouffres en ajoutent davantage.
Évolutivité à grande échelle	Bas	La faible cohésion à l'intérieur de chaque couche et le couplage entre fonctionnalités rendent les grands systèmes en couches difficiles à restructurer.

SI VOUS DÉBUTEZ

Pas à pas : une couche qui semble ne rien faire

L'idée la plus difficile de cette section est le compromis qui se cache derrière le gouffre d'architecture. Une couche qui ne fait que transmettre une requête ressemble à du code gaspillé, et pourtant la règle qui force les requêtes à la traverser est ce qui maintient l'isolation des couches. Suivez deux requêtes à travers l'application de commande de repas.

- Un client consulte une carte. Le contrôleur appelle une méthode de la couche métier, qui appelle simplement le repository des cartes, qui exécute la requête. Le contrôleur et la méthode métier ne font que transmettre la requête.
- Un développeur propose de laisser le contrôleur appeler directement le repository. Cela économiserait du code, mais ferait de la couche métier une couche ouverte : une couche que les requêtes peuvent sauter.
- Des mois plus tard, l'équipe restructure les tables des cartes. En passant par la couche métier, le contrôleur n'aurait pas besoin de changer, tant que l'interface de cette couche resterait la même. Avec le raccourci, le changement atteint aussi le contrôleur.
- Passer une commande est différent : le contrôleur vérifie l'entrée, la couche métier calcule le prix du panier et vérifie les règles, et le repository enregistre la commande dans une seule transaction. Chaque couche accomplit un vrai travail.
- La question utile est donc de savoir combien de requêtes ressemblent à celle de la carte. La règle empirique de Richards : qu'environ un cinquième soient de simples relais est normal. Si c'est le cas de la plupart, l'application consiste probablement surtout en accès aux données, avec peu de règles. Un autre style peut alors mieux lui convenir, ou l'équipe peut ouvrir certaines couches à dessein et consigner pourquoi, en acceptant la perte d'isolation.
- **La leçon :** un certain relais est le prix des couches d'isolation. Mesurez-le plutôt que de le supprimer à vue, et considérez une large majorité de requêtes de simple relais comme le signe que le style ne convient pas.

Quand il convient, et quand il ne convient pas

L'architecture en couches est un bon choix pour les petites et moyennes applications construites par une seule équipe, pour les produits dont le domaine n'est pas encore bien compris (le partitionnement par domaine exige des frontières que vous ne connaissez pas encore), pour les budgets et les délais serrés, et pour les applications qui consistent surtout en formulaires sur des données, avec des règles modestes. C'est aussi une structure interne

raisonnable pour un module ou un service unique qui fait lui-même partie d'un système partitionné par domaine : un module `delivery` en couches à l'intérieur d'un monolithe modulaire profite du meilleur des deux.

C'est un mauvais choix comme structure de premier niveau d'un grand système à plusieurs équipes, car chaque équipe travaille dans chaque couche et les fonctionnalités entrent en collision. C'est un mauvais choix lorsque des parties du système ont besoin d'une scalabilité ou d'une disponibilité très différentes, car il n'y a qu'un seul quantum. Et c'est un mauvais choix lorsque la logique métier est riche et centrale, car les règles sont dispersées entre contrôleurs, services et procédures stockées ; dans ce cas, une structure interne centrée sur le domaine comme l'architecture hexagonale ou Clean Architecture garde le cœur indépendant de la base de données et du framework web.

Le chemin d'évolution le plus courant va d'un monolithe en couches à un monolithe modulaire : conserver un seul déploiement, mais réorganiser le niveau supérieur par domaine et garder des couches à l'intérieur de chaque module.

Faire respecter les règles des couches dans le code

Les règles des couches se dégradent vite. Sous la pression des délais, un développeur importe le repository directement dans un contrôleur parce que cela lui fait gagner vingt minutes, personne ne le remarque en revue, et le développeur suivant copie ce modèle. La défense est un test d'architecture : une vérification automatisée des dépendances qui s'exécute dans le build et le fait échouer lorsqu'une règle est enfreinte, ce qui fait de la règle une fonction d'aptitude.

En Python, import-linter lit des contrats dans un fichier de configuration. Son contrat `layers` énumère les couches de la plus haute à la plus basse et interdit à toute couche inférieure d'importer une couche supérieure. À lui seul, il ne rend pas les couches fermées (une couche supérieure peut importer n'importe quelle couche inférieure) : un contrat `forbidden` ajoute donc la règle selon laquelle la présentation ne doit pas importer directement la persistance ; il faut activer `allow_indirect_imports`, sans quoi la chaîne normale de la présentation vers la persistance en passant par le métier compterait comme une violation. En Java, ArchUnit exprime les mêmes règles sous forme de test unitaire : `layeredArchitecture()` déclare les couches par paquet et indique, par exemple, que la couche de persistance ne peut être accédée que par la couche de services. Des outils similaires existent pour d'autres écosystèmes, comme dependency-cruiser pour JavaScript et TypeScript.

Deux pratiques font tenir les vérifications. Introduisez-les en enregistrant les violations existantes comme des exceptions connues, afin que le build passe immédiatement au vert et que seules les nouvelles violations le fassent échouer ; puis réduisez les exceptions au fil du

temps. Et gardez la configuration à côté du code et relisez ses modifications comme du code, car assouplir une règle est une décision d'architecture qui mérite un enregistrement de décision.

```
[importlinter]
root_package = foodshop

[importlinter:contract:layers]
name = Layers only depend downwards
type = layers
layers =
    foodshop.presentation
    foodshop.business
    foodshop.services
    foodshop.persistence

[importlinter:contract:closed-business]
name = Presentation must go through business
type = forbidden
source_modules = foodshop.presentation
forbidden_modules = foodshop.persistence
# presentation → business → persistence is the normal path
allow_indirect_imports = True
```

Une configuration import-linter (.importlinter) pour l'application de commande de repas : les couches ne peuvent dépendre que vers le bas, et la présentation ne peut pas sauter la couche métier.

POUR ALLER PLUS LOIN

Les couches à l'intérieur d'autres styles

L'organisation en couches survit à l'intérieur de la plupart des autres styles. Chaque module d'un monolithe modulaire, chaque microservice et chaque plug-in peut être organisé en couches en interne. Les architectures hexagonale et Clean Architecture peuvent se lire comme une organisation en couches avec un changement crucial : la dépendance entre la logique métier et la persistance est inversée, de sorte que le code de base de données dépend d'interfaces définies par la couche métier plutôt que l'inverse. C'est cette inversion qui permet de tester le cœur métier sans base de données et de lui faire survivre à ses frameworks.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre une couche fermée et une couche ouverte ?**

Une requête doit traverser une couche fermée et ne peut pas la sauter ; une couche ouverte peut être contournée par la couche supérieure, ce qui est typique d'une couche de services partagés dont seules certaines requêtes ont besoin.

▼ **Pourquoi une application en couches a-t-elle tendance à exiger des changements dans chaque couche pour une nouvelle fonctionnalité ?**

Parce qu'elle est partitionnée techniquement : le code de chaque capacité métier est réparti entre les couches de présentation, métier et persistance, et la plupart des fonctionnalités sont définies en termes métier.

▼ **Quelle est la différence entre une couche et un niveau (tier) ?**

Une couche est un regroupement logique du code par responsabilité ; un niveau est une frontière physique à l'exécution, comme un serveur séparé. Plusieurs couches tournent généralement à l'intérieur d'un seul niveau.

▼ **Pourquoi un contrat layers d'import-linter ne rend-il pas, à lui seul, les couches fermées ?**

Il interdit seulement aux couches inférieures d'importer les couches supérieures ; une couche supérieure peut toujours importer n'importe quelle couche inférieure, si bien qu'un contrat forbidden séparé est nécessaire pour empêcher la présentation d'importer la persistance.

▼ **À vous : Vous tracez 500 requêtes dans une app de billetterie en couches, et 400 traversent directement les couches métier et de persistance sans qu'aucune logique ne s'y ajoute. Comment cela s'appelle-t-il, que cela suggère-t-il, et que vérifieriez-vous avant d'agir ?**

C'est l'antipattern du gouffre d'architecture. Un certain transit est le prix normal de l'isolation, environ un cinquième selon la règle empirique habituelle, mais ici il atteint 80 %, ce qui suggère que le style en couches n'est peut-être pas le bon. Vérifiez d'abord si ces requêtes sont surtout de simples lectures et si le reste du système tire réellement profit de l'isolation. Les options comprennent l'ouverture d'une couche ou le passage à un partitionnement par domaine. Ne réécrivez pas sur la foi d'un seul échantillon.

SOURCES

- Martin Fowler: Presentation Domain Data Layering: https://martinfowler.com/bliki/PresentationDomainDataLayering.html
- Microsoft Azure Architecture Center: N-tier architecture style: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier
- import-linter documentation: Layers contract: https://import-linter.readthedocs.io/en/stable/contract_types/layers/
- ArchUnit user guide (layer checks): https://www.archunit.org/userguide/html/000_Index.html
- Jimmy Bogard: Vertical Slice Architecture (2018): https://www.jimmybogard.com/vertical-slice-architecture/

SECTION 5.3 SUR 5.8 · INTERMÉDIAIRE · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Le monolithe modulaire

AVANT DE LIRE

Cette section s'appuie sur :

- 3.3 Masquage de l'information
- 5.1 Styles et patrons
- 5.2 En couches

Termes déjà vus qu'elle utilise : Unité de déploiement, Partitionnement par domaine, Grosse boule de boue (big ball of mud), Test d'architecture, Transaction

L'IDÉE EN BREF

Pensez à un grand magasin. C'est un seul bâtiment, avec une seule porte d'entrée, des horaires d'ouverture communs et une seule équipe de sécurité, mais à l'intérieur il est divisé en rayons : alimentation, vêtements, électronique. Chaque rayon a son propre personnel, sa propre réserve et sa propre caisse, et si le rayon vêtements a besoin de quelque chose du rayon alimentation, il le demande au comptoir de l'alimentation au lieu d'aller se servir dans sa réserve. Un monolithe modulaire est un logiciel construit comme ce magasin : un seul programme, déployé d'un seul bloc, mais divisé à l'intérieur en rayons métier qui gardent leurs propres données et ne traitent entre eux que par des comptoirs convenus.

CE QU'IL FAUT SAVOIR

- Un monolithe modulaire est une unité de déploiement unique dont le niveau supérieur est partitionné par domaine métier en modules, chacun doté d'une petite API publique explicite, d'éléments internes privés et de ses propres données.
- Il conserve la simplicité opérationnelle d'un monolithe (un seul build, un seul déploiement, des appels internes au processus, des transactions locales) tout en offrant une grande partie des avantages de modularité que l'on recherche dans les microservices.
- Les frontières s'érodent si on ne les fait pas respecter : visibilité du langage, tests d'architecture (ArchUnit, import-linter), vérification de Spring Modulith pour les applications Spring, Packwerk pour Ruby et schémas de base de données séparés pour chaque module.
- Chaque table appartient à exactement un module ; les autres modules utilisent son API ou réagissent à ses événements, jamais à ses tables. Les jointures entre modules sont remplacées par des appels d'API ou des modèles de lecture.

- Les modules peuvent communiquer par des appels internes au processus (simples, immédiats, transactionnels) ou par des événements internes au processus (couplage plus lâche, mais le travail de l'écouteur se fait séparément et peut échouer séparément).
- Shopify a réorganisé son grand monolithe Ruby on Rails en composants de domaine à partir de 2017 et l'a décrit en 2019 ; ses articles ultérieurs montrent à la fois l'intérêt de l'approche et la difficulté d'une isolation complète.
- Un module bien délimité peut ensuite être extrait en service pour un effort modeste, car son API, ses données et ses dépendances sont déjà explicites.

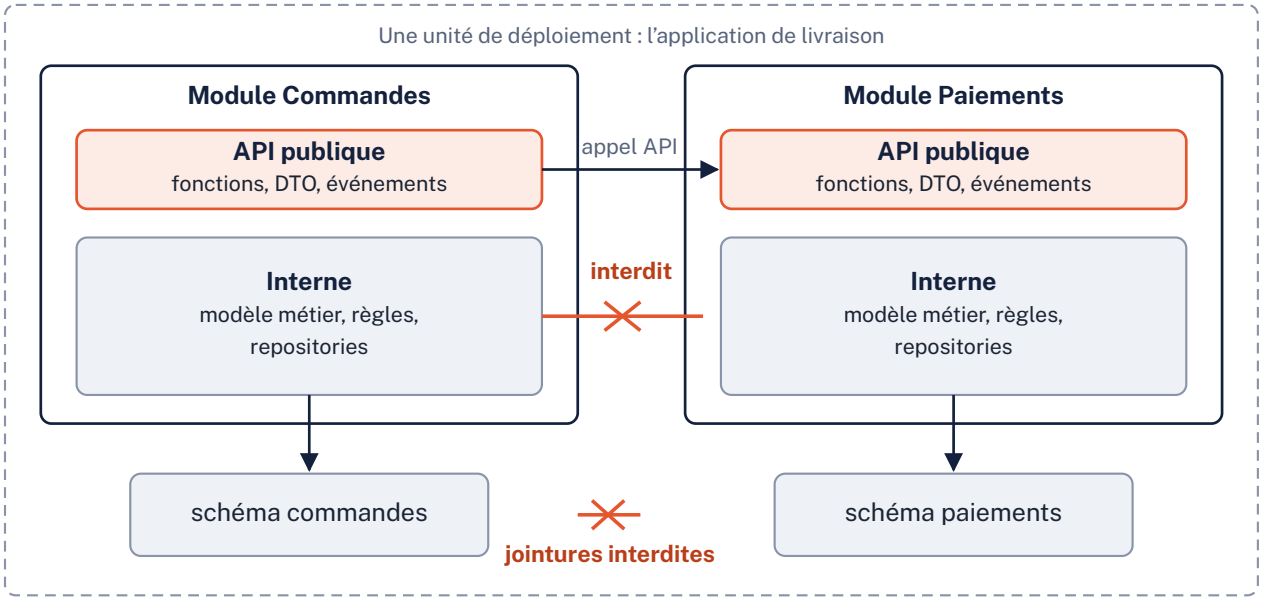
Définition : un seul déploiement, plusieurs domaines

Un monolithe modulaire est une application construite et déployée comme une seule unité, mais dont le code est divisé, au niveau supérieur, en modules qui suivent les frontières métier : commandes, paiements, livraison, cartes. Chaque module possède une petite API publique délibérée que les autres modules peuvent utiliser, une partie interne plus grande à laquelle personne d'extérieur ne doit toucher, et ses propres données. Il combine le partitionnement par domaine des microservices avec l'unité de déploiement unique d'un monolithe classique.

L'idée n'est pas nouvelle. Les modules aux éléments internes cachés remontent au masquage de l'information de David Parnas (1972), et de grands monolithes bien structurés ont toujours existé. Ce qui est plus récent, c'est le nom et l'attention qu'on lui porte. Après une décennie au cours de laquelle de nombreuses organisations ont découpé leurs systèmes en microservices et découvert le coût de la distribution, des praticiens comme Simon Brown et Sam Newman ont soutenu que le monolithe modulaire est une option sous-estimée, et que beaucoup d'équipes seraient mieux servies par un monolithe modulaire que par un ensemble de services. Richards et Ford lui consacrent un chapitre entier dans la deuxième édition de *Fundamentals of Software Architecture* (2025).

La différence avec le style en couches tient au découpage du niveau supérieur. Une application en couches est divisée en présentation, métier et persistance, chaque domaine étant réparti sur les trois. Un monolithe modulaire est divisé en domaines, et chaque module peut être organisé en couches à l'intérieur. La différence avec les microservices tient au déploiement : les modules partagent un seul processus, une seule mise en production et généralement un seul serveur de base de données, et ils s'appellent les uns les autres en mémoire.

Anatomie d'un monolithe modulaire



Chaque module expose une petite API publique et possède son propre schéma ; accéder aux éléments internes ou aux tables d'un autre module est interdit.

API des modules et code interne

La règle la plus importante est qu'un module ne s'utilise qu'à travers son API. L'API est l'ensemble des fonctions, des objets de transfert de données (DTO) et des événements que les responsables du module s'engagent à garder stables ; tout le reste, y compris le modèle de domaine, les repositories et les tables de la base de données, est interne et peut changer à tout moment. En pratique, l'API est petite : un module de paiements peut exposer `authorise_payment`, `refund`, un type `PaymentResult` et un événement `PaymentCaptured`, tout en cachant des dizaines de classes pour les prestataires, les nouvelles tentatives, les registres comptables et le rapprochement.

Les bonnes API de module ont plusieurs traits communs. Elles parlent la langue du domaine (`authorise_payment`, et non `insert_payment_row`). Elles transmettent des données simples, comme des identifiants et de petits enregistrements immuables, plutôt que des objets internes, pour que les appelants ne puissent pas passer par un objet renvoyé pour atteindre les éléments internes du module. Elles sont assez grossières pour qu'un cas d'utilisation typique n'ait besoin que d'un ou deux appels, et non de vingt. Et elles sont conçues comme si un réseau devait un jour s'interposer entre l'appelant et l'appelé : pas d'objets mutables partagés, pas de dépendance à la transaction de base de données de l'appelant, des erreurs explicites. C'est ce dernier trait qui préserve la possibilité d'extraire le module plus tard.

À l'intérieur de la frontière, tout ce que l'équipe du module juge utile est permis : une structure en couches, une structure hexagonale, un modèle de domaine riche ou de simples scripts de transaction. Des modules différents peuvent faire des choix différents, car rien à l'extérieur n'en dépend.

```
# delivery_platform/payments/__init__.py  (the module's public API)
from .api import authorise_payment, refund, PaymentResult
from .events import PaymentCaptured

__all__ = ["authorise_payment", "refund", "PaymentResult", "PaymentCaptured"]

# delivery_platform/ordering/internal/checkout.py
from delivery_platform import payments             # allowed: public API
# from delivery_platform.payments.internal import ledger   # forbidden

def place_order(order, card_token):
    result = payments.authorise_payment(order.id, order.total, card_token)
    if not result.ok:
        raise PaymentDeclined(result.reason)
    order.confirm(result.payment_id)
    events.publish(OrderPlaced(order.id, order.restaurant_id))
```

Un paquet Python comme module : le paquet payments exporte une petite API, et ordering n'importe que cette API.

Faire respecter les frontières

Dans un monolithe, chaque classe peut techniquement appeler n'importe quelle autre, donc les frontières ne tiennent que si quelque chose les fait respecter. Sans contrôle, un raccourci pris sous la pression d'une échéance vers les éléments internes d'un autre module passe inaperçu en revue, et un monolithe modulaire se dégrade en grosse boule de boue (big ball of mud) en un an ou deux. Il existe plusieurs lignes de défense, à utiliser de préférence ensemble.

Visibilité du langage. Utilisez le masquage le plus fort que le langage propose : classes à visibilité package en Java, `internal` en Kotlin et en C# (qui n'impose de frontières qu'entre modules de build ou assemblies, et non entre packages d'un même projet), les `exports` du Java Platform Module System, des modules de build séparés (sous-projets Maven ou Gradle, projets .NET) dont les dépendances sont déclarées explicitement. Python et Ruby n'offrent que des conventions (un trait de soulignement initial, un sous-paquet `internal`), donc ce sont les outils qui doivent faire le travail.

Tests d'architecture. Les tests d'architecture vérifient les dépendances dans le build. ArchUnit (Java) peut affirmer qu'aucune classe extérieure à un module ne touche à ses paquets internes et que les dépendances entre modules sont exemptes de cycles. import-linter (Python) propose un contrat `forbidden`, qui peut empêcher les autres modules d'importer le paquet interne

d'un module tout en les laissant utiliser son API publique, un contrat `layers` dont les couches peuvent contenir plusieurs éléments frères mutuellement indépendants séparés par `|`, ce qui rend vérifiable un ordre de dépendance à sens unique entre modules, et un contrat `independence` pour les paquets qui ne doivent pas du tout dépendre les uns des autres, même indirectement.

Support des frameworks. Spring Modulith, pour les applications Spring Boot, traite chaque sous-paquet direct du paquet principal de l'application comme un module, considère le paquet de base du module comme son API et ses sous-paquets comme internes, et propose une vérification en une ligne (`ApplicationModules.of(Application.class).verify()`) qui échoue lorsqu'un module fait référence aux éléments internes d'un autre module, lorsque les dépendances entre modules forment un cycle ou, quand un module liste ses dépendances autorisées, lorsqu'il en utilise une qui ne figure pas dans la liste. Il peut aussi générer des diagrammes de composants de style C4 pour les modules. Dans le monde Ruby, Packwerk, l'outil open source de Shopify, analyse statiquement les références entre paquets, chacun déclaré par un fichier `package.yml`, et signale les dépendances non déclarées.

Frontières des données. Donnez à chaque module son propre schéma de base de données (ou au moins ses propres tables clairement préfixées) et, idéalement, son propre utilisateur de base de données, qui ne peut accéder qu'à ce schéma. La revue de code seule n'empêchera pas un `JOIN` commode entre modules ; les permissions de la base de données, si.

```
[importlinter:contract:payments-internals]
name = Only payments may use payments internals
type = forbidden
source_modules =
    delivery_platform.ordering
    delivery_platform.delivery
    delivery_platform.reporting
forbidden_modules =
    delivery_platform.payments.internal
# going through the public API (which itself uses internals) is fine
allow_indirect_imports = True

[importlinter:contract:module-order]
name = Module dependencies point one way
type = layers
layers =
    delivery_platform.reporting | delivery_platform.notifications
    delivery_platform.delivery
    delivery_platform.ordering | delivery_platform.restaurants
    delivery_platform.payments | delivery_platform.menus | delivery_platform.customers
```

Contrats import-linter (un par module en pratique) : les autres modules ne peuvent pas importer les éléments internes de payments, et les dépendances entre modules doivent suivre un seul sens, les modules d'une même ligne étant indépendants les uns des autres.

La propriété des données au sein d'un monolithe

La propriété des données est le point où les monolithes modulaires échouent le plus souvent. Partager un serveur de base de données ne pose pas de problème ; partager des tables, si. Si le module de livraison lit directement la table `orders`, l'équipe des commandes ne peut plus modifier cette table sans se coordonner avec la livraison, et les deux modules sont aussi étroitement couplés que s'ils n'en formaient qu'un. La règle est simple à énoncer : chaque table a exactement un module propriétaire, seul le propriétaire y écrit, et les autres modules obtiennent les données par l'API du propriétaire ou à partir des événements qu'il publie.

La règle a des conséquences que les équipes doivent accepter délibérément. Un rapport qui était autrefois une seule requête SQL joignant commandes, paiements et livreurs nécessite désormais soit des appels à trois API de modules, soit un modèle de lecture : un module de reporting qui s'abonne aux événements des autres et tient sa propre copie dénormalisée, optimisée pour les requêtes. Les clés étrangères entre schémas de modules sont généralement supprimées, car elles rendraient le schéma d'un module dépendant de celui d'un autre ; l'intégrité entre modules est alors assurée par l'application. Ces coûts sont réels, mais ce sont les mêmes que ceux qu'imposerait un passage ultérieur aux services, payés tôt et dans un cadre moins coûteux.

Il reste un avantage que les systèmes distribués perdent : des modules dans un même processus peuvent, quand c'est vraiment nécessaire, partager une seule transaction locale de base de données. Utilisez-le avec parcimonie. Chaque utilisation couple opérationnellement les données des modules et devra être défaite si un module est un jour extrait.

SI VOUS DÉBUTEZ

Pas à pas : un propriétaire pour chaque table

L'idée qui demande le plus d'attention ici est la propriété des données au sein d'un même programme. Les modules partagent un seul serveur de base de données, et pourtant chaque table a un seul propriétaire : seul ce module peut la lire ou l'écrire directement. Suivez un raccourci et un rapport à travers la plateforme de livraison de repas.

- Le module Livraison a besoin des détails de chaque nouvelle commande. La table des commandes se trouve dans la même base de données, donc le code le plus rapide à écrire est une requête qui la lit directement.
- Des mois plus tard, Commandes réorganise cette table et remplace sa colonne de statut par des horodatages distincts de passage et de paiement. La requête de Livraison échoue alors, et dès ce moment les deux équipes doivent planifier ensemble chaque modification de la table, aussi étroitement couplées que si elles formaient un seul module.
- La règle supprime le raccourci. Seul Commandes touche à ses tables ; les autres modules utilisent l'API publique de Commandes ou les événements internes au processus qu'il publie, des messages internes au programme annonçant que quelque chose s'est produit. Dans la conception présentée plus loin dans cette section, Livraison écoute OrderPlaced au lieu d'interroger la table.
- Les rapports sont plus difficiles. Un rapport sur les annulations qui était autrefois une seule requête SQL joignant commandes, remboursements et clients irait puiser dans les tables de trois modules.
- Il y a deux solutions : appeler les API des trois modules et combiner les réponses, ou laisser Reporting tenir un modèle de lecture : il écoute les événements des modules, comme OrderCancelled, et stocke sa propre copie des seules données dont le rapport a besoin.
- Les clés étrangères entre les tables des modules sont généralement supprimées elles aussi ; c'est l'application, et non la base de données, qui maintient ces liens corrects.
- **La leçon :** partager un serveur de base de données ne pose pas de problème ; partager des tables, si. Les événements et les modèles de lecture représentent un travail supplémentaire, mais c'est le même travail qu'exigerait un passage ultérieur aux services, fait plus tôt et à moindre coût.

Communication entre modules : appels ou événements

Les modules communiquent de deux façons, toutes deux en mémoire. Un **appel interne au processus** à l'API d'un autre module est simple, rapide et immédiat : Commandes appelle `payments.authorise_payment` et obtient le résultat aussitôt, et une exception se propage normalement. Il crée une dépendance directe, de l'appelant vers l'appelé, et l'appelant ne peut pas terminer avant l'appelé. Utilisez des appels quand l'appelant a besoin d'une réponse pour continuer, par exemple pour savoir si le paiement a été autorisé.

Un événement interne au processus inverse la dépendance. Commandes publie `OrderPlaced` sans savoir ni se soucier de qui écoute ; Livraison, Notifications et Reporting s'y abonnent. Ajouter une nouvelle réaction revient à ajouter un écouteur, sans toucher à Commandes. Utilisez des événements quand l'émetteur n'a pas besoin de réponse, et quand plusieurs modules réagissent au même fait.

Les événements introduisent des subtilités qui échappent souvent aux équipes qui les découvrent. Si un écouteur s'exécute dans la transaction de l'émetteur, une défaillance de l'écouteur peut annuler la commande ; s'il s'exécute après le commit, la commande est enregistrée mais le travail de l'écouteur peut être perdu si le processus plante entre les deux. Spring Modulith traite ce problème avec un registre de publication d'événements qui enregistre chaque paire événement-écouteur dans la même transaction que la modification métier et ne la marque comme terminée que lorsque l'écouteur a réussi, afin que les paires inachevées puissent faire l'objet d'une nouvelle tentative ; un équivalent fait main est l'outbox transactionnelle décrite à la section 6.7. Les écouteurs doivent aussi être idempotents, pour qu'un événement rejoué ne cause aucun dommage. Aucun des deux styles n'est meilleur ; la plupart des monolithes modulaires utilisent des appels pour les requêtes et les commandes qui exigent une réponse, et des événements pour signaler des choses qui se sont produites.

Migrations et tests par module

La propriété ne dure que si l'outillage la respecte. Donnez à chaque module son propre schéma de base de données et son propre dossier de migrations, afin que les changements de Commandes vivent à côté de son code et ne puissent pas modifier discrètement les tables de Livraison ; Flyway, par exemple, peut s'exécuter une fois par schéma, avec des emplacements de scripts et des tables d'historique distincts. Une connexion à la base par module, autorisée sur son seul schéma, transforme une requête égarée en erreur. Les tests suivent les mêmes lignes : testez chaque module par son API publique, ses voisins étant remplacés par des doublures. Le `@ApplicationModuleTest` de Spring Modulith, par exemple, démarre par défaut un seul module. Gardez quelques tests de toute l'application pour le câblage. Comme tout se déploie encore ensemble, l'expansion-contraction reste nécessaire lors d'une mise en production progressive.

Le monolithe modulaire de Shopify

L'application principale de Shopify est l'une des plus grandes bases de code Ruby on Rails existantes, sur laquelle plus d'un millier de développeurs ont travaillé pendant plus de dix ans. Dans «Deconstructing the Monolith» (Shopify Engineering, février 2019), Kirsten Westeinde a expliqué pourquoi l'entreprise ne l'a pas découpée en microservices et l'a plutôt transformée, à partir de 2017, en monolithe modulaire. Les difficultés étaient connues : le code de domaines métier sans rapport était enchevêtré, les modifications avaient des effets inattendus ailleurs, et l'intégration des nouveaux arrivants était lente. Les microservices ont été envisagés et jugés porteurs de leurs propres coûts (pipelines et infrastructures séparés, latence et défaillances du réseau, refactorisation plus difficile entre services) ; l'équipe a donc choisi d'obtenir la modularité sans multiplier les unités de déploiement.

La première étape a été de réorganiser le code, de la structure Rails standard (modèles, vues et contrôleurs) en composants nommés d'après les domaines métier, comme les commandes, l'expédition, les stocks et la facturation, en une seule grande modification automatisée. La suivante a été l'isolation : chaque composant devait avoir une API publique et la propriété exclusive de ses données, et un outil interne, Wedge, suivait les progrès de chaque composant en enregistrant les appels qui traversaient les frontières pendant l'exécution de la suite de tests. Un point d'étape de septembre 2020 faisait état de 37 composants dans un monolithe central d'environ 2,8 millions de lignes de Ruby, et le même mois Shopify a publié en open source Packwerk, un outil d'analyse statique permettant de déclarer et de vérifier les dépendances entre paquets.

Cette histoire est instructive parce que ce n'est pas un simple succès. Le point d'étape de 2020 indiquait que les idées d'origine tenaient toujours mais que presque tous les détails avaient changé, et que presque aucun composant n'avait encore de frontière complète et solide. Un bilan rétrospectif de Packwerk publié en 2024 rapportait que ses contrôles de confidentialité avaient été supprimés, que certains paquets organisés par nom plutôt que par comportement à l'exécution induisaient les développeurs en erreur, et qu'isoler complètement ne serait-ce qu'un paquet de base avait pris de nombreux mois ; Packwerk était devenu un garde-fou sur les frontières clés plutôt que le cœur de la stratégie. La leçon pour les architectes est qu'un monolithe modulaire est une direction à suivre qui exige un investissement soutenu, et qu'introduire après coup des frontières dans une grande base de code existante est beaucoup plus difficile que de les préserver dès le départ.

EXEMPLE COMMENTÉ

Exemple commenté : la plateforme de livraison de repas en monolithe modulaire

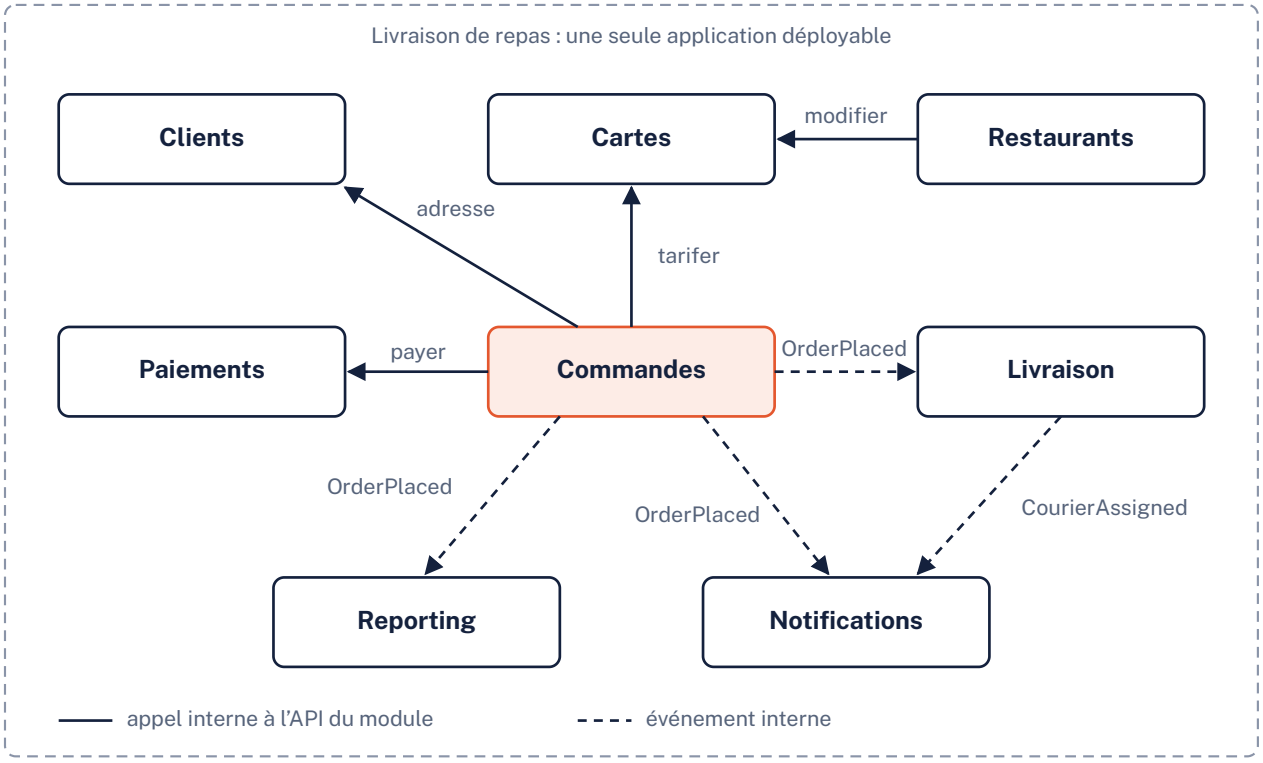
Une start-up de livraison de repas comptant quatre équipes (environ 25 ingénieurs) veut une propriété indépendante sans le coût d'exploitation des microservices. L'architecte propose huit modules dans une seule application Python déployable, avec un seul serveur PostgreSQL et un schéma par module.

Clients	Comptes, adresses, préférences. API : `get_customer`, `get_delivery_address`. Équipe A.
Restaurants ; Cartes (deux modules)	Restaurants gère les comptes des restaurants et leurs horaires d'ouverture, et met à jour les cartes par l'API de Cartes ; Cartes fournit les prix et la disponibilité à tous les autres (`price_items`, `is_open`). Équipe B.
Commandes	Panier, étape de paiement, cycle de vie de la commande. Appelle Clients, Cartes et Paiements ; publie `OrderPlaced` et `OrderCancelled`. Équipe C.
Paiements	Autorisation, capture et remboursements via un prestataire externe. API : `authorise_payment`, `refund` ; publie `PaymentCaptured`. Équipe C.
Livraison	Affectation des livreurs et suivi. Écoute `OrderPlaced` ; publie `CourierAssigned` et `Delivered`. Équipe D.
Notifications et Reporting	Purs écouteurs : envoient des messages et tiennent à jour des modèles de lecture. Ils n'appellent aucun autre module. Équipe A.
Règle de dépendance	Paiements, Cartes et Clients ne dépendent de personne ; Commandes et Restaurants les appellent ; Livraison réagit aux événements de Commandes ; Notifications et Reporting se contentent d'écouter. Vérifié par import-linter dans le build.
Déploiement	Une seule image de conteneur, exécutée en six réplicas derrière un équilibreur de charge, plus un processus worker (même code) qui gère les écouteurs d'événements et les tâches d'arrière-plan.

Chaque équipe possède des modules entiers, donc la plupart des modifications ne touchent qu'un module et qu'une équipe.

La règle de dépendance rend les cycles impossibles : si Paiements a un jour besoin de connaître les commandes, il reçoit un identifiant de commande, et non un import de Commandes.

Carte des modules de la plateforme de livraison de repas



Les flèches pleines sont des appels à l'API d'un autre module ; les flèches en pointillés sont des événements internes au processus. Commandes appelle trois modules, et ses écouteurs ne dépendent que des événements qu'il publie.

Caractéristiques du monolithe modulaire

Caractéristique	Évaluation	Justification
Simplicité	Moyen à élevé	Un seul déploiement et des appels internes au processus, mais trouver de bonnes frontières de modules demande de la réflexion.
Coût global	Élevé (peu coûteux)	Infrastructure et exploitation d'un monolithe ; un peu d'effort supplémentaire pour l'outillage des frontières.
Modifiabilité pour les changements métier	Élevé	Le partitionnement par domaine garde la plupart des modifications dans un seul module, détenu par une seule équipe.
Testabilité	Moyen à élevé	Les modules peuvent être testés par leurs API en remplaçant leurs voisins ; les tests du système complet s'exécutent dans un seul processus.
Déployabilité	Bas à moyen	Toujours une seule unité de déploiement : chaque mise en production livre tous les modules, même si un rayon d'impact plus petit par modification aide.
Scalabilité et élasticité	Bas à moyen	Les réplicas font passer à l'échelle toute l'application ; un module ne peut pas passer à l'échelle seul, même si des processus worker peuvent isoler la charge d'arrière-plan.
Tolérance aux pannes	Bas	Un seul processus : un plantage ou une fuite de mémoire dans n'importe quel module les affecte tous.
Performances	Élevé	Les appels entre modules se font en mémoire, sans sérialisation ni latence réseau.
Évolutivité	Élevé	Des API explicites et la propriété des données rendent les modules faciles à restructurer ou à extraire plus tard.

Quand l'utiliser, quand l'éviter, et les anti-patterns

Un monolithe modulaire est un bon choix par défaut pour les nouveaux produits dont le domaine est raisonnablement clair, pour les organisations comptant jusqu'à quelques équipes, pour les systèmes qui ont besoin d'une forte cohérence entre domaines métier, et pour les équipes dont la capacité d'exploitation est limitée. C'est aussi la cible naturelle pour moderniser un monolithe enchevêtré : y introduire des frontières est généralement moins coûteux et moins risqué que de le découper en services d'un seul coup. Il convient moins lorsque des parties du système ont besoin d'attributs de qualité réellement différents (un flux de positions de livreurs qui doit absorber d'énormes volumes d'écriture et un module de back-office à charge modeste), lorsque des dizaines d'équipes doivent mettre en production

indépendamment plusieurs fois par jour, ou lorsque des parties doivent être écrites dans des langages différents ou s'exécuter dans des lieux différents pour des raisons de résidence des données. Ces besoins appellent des quanta d'architecture distincts.

Anti-patterns courants : **les modules de nom seulement**, où les dossiers suivent les domaines mais où n'importe quoi importe n'importe quoi ; **la base de données partagée en libre-service**, où les modules gardent leur propre code mais font des jointures sur les tables des autres ; **la frontière bavarde**, une API si fine qu'un cas d'utilisation nécessite des dizaines d'appels, ce qui sera pénible si le module est un jour extrait ; **le noyau partagé qui grossit**, un module `common` qui absorbe peu à peu de la logique métier et y couple chaque module ; et **les modules par entité**, un découpage par entité de données (un module Client, un module Commande, un module Produit dont chaque cas d'utilisation a besoin en même temps) au lieu d'un découpage par capacité métier, ce qui produit des modules incapables de faire quoi que ce soit seuls.

Préserver la possibilité d'extraire des services

Le monolithe modulaire est parfois présenté comme un tremplin vers les microservices, parfois comme une destination. Il peut être l'un ou l'autre, et l'essentiel est que la décision peut être reportée jusqu'au dernier moment responsable. Si un module a plus tard besoin de passer à l'échelle séparément ou d'être mis en production indépendamment, l'extraction est bien plus facile quand le module a déjà une API explicite, possède ses données et ne dépend des autres qu'à travers des API et des événements. Le travail devient en grande partie mécanique : placer une interface réseau devant l'API, déplacer le schéma dans sa propre base de données, remplacer les événements internes au processus par un courtier de messages (message broker), et remplacer toute transaction partagée par une saga explicite ou une compensation.

Sam Newman défend la même thèse dans *Monolith to Microservices* (2019) : un monolithe bien modularisé est le meilleur point de départ pour une décomposition, car la partie difficile, trouver les frontières et les faire respecter, est déjà faite. La section 5.8 examine les signes indiquant qu'il est temps d'extraire, et la section 11.6 couvre les techniques de migration incrémentale comme le strangler fig.

Un découpage plus grossier : les systèmes autonomes (self-contained systems)

Les **systèmes autonomes** (self-contained systems, SCS) divisent un grand système en quelques applications web autonomes, chacune avec ses propres données, sa logique et son interface utilisateur, et chacune détenue par une seule équipe. Ils sont en général plus grands et moins nombreux que les microservices : le site SCS en suggère peut-être 5 à 25 pour une boutique en ligne, là où les microservices se compteraient par centaines. Chaque système doit assurer seul sa fonction principale ; les appels aux autres restent donc hors du cycle requête-

réponse, et les interfaces se relient par des liens plutôt qu'en partageant du code. Sur la plateforme de livraison de repas, la prise de commande, le portail des restaurants et les outils des livreurs pourraient en être chacun un. Ce style convient aux domaines clairs dotés d'interfaces web, et chaque système peut lui-même être un monolithe modulaire. Les écrans qui ont besoin de données de nombreux domaines sont plus difficiles.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Qu'est-ce qui distingue un monolithe modulaire d'un monolithe en couches ?**

Son niveau supérieur est partitionné par domaine métier plutôt que par couche technique ; chaque module a une API explicite, des éléments internes privés et ses propres données, et peut être organisé en couches à l'intérieur.

▼ **Pourquoi les API de module doivent-elles transmettre des identifiants et des enregistrements simples plutôt que des objets internes du domaine ?**

Les objets internes renvoyés permettent aux appelants d'atteindre les éléments internes du module et les couplent à son modèle ; des données simples laissent les éléments internes libres de changer et l'API prête à traverser un réseau plus tard.

▼ **Un rapport a besoin de données des schémas commandes et paiements. Quelles sont deux options qui respectent la propriété des données ?**

Appeler les API des deux modules et combiner les résultats, ou construire un modèle de lecture dans un module de reporting qui écoute les événements des deux modules et tient sa propre copie dénormalisée.

▼ **Quel risque introduit un écouteur d'événement interne au processus qui s'exécute après le commit de la transaction de l'émetteur ?**

Si le processus plante après le commit mais avant l'exécution de l'écouteur, le travail de l'écouteur est perdu, à moins que l'événement ne soit enregistré durablement, par exemple dans un registre de publication d'événements ou une outbox, puis rejoué.

▼ **À vous : Dans un monolithe modulaire de livraison de repas, le module Rapports fait directement des jointures sur les tables du module Paiements pour une synthèse nocturne. Cela fonctionne et c'est rapide. Pourquoi s'y opposer, par quoi le remplaceriez-vous, et comment éviteriez-vous que cela se reproduise ?**

Chaque table appartient à un seul module. La jointure couple Rapports au schéma privé de Paiements : un changement dans Paiements peut donc la casser en silence, et une extraction ultérieure devient plus difficile. Remplacez-la par un appel à l'API publique de Paiements ou par un modèle de lecture alimenté par les événements de Paiements, en acceptant un peu de travail en plus et de données périmées. Donnez à chaque module son propre schéma et ajoutez un test d'architecture, avec import-linter ou des droits de base de données par exemple, afin que le build échoue à toute récidive.

SOURCES

- Shopify Engineering: Deconstructing the Monolith (2019): https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity
- Shopify Engineering: Under Deconstruction: The State of Shopify's Monolith (2020): https://shopify.engineering/shopify-monolith
- Shopify Engineering: A Packwerk Retrospective (2024): https://shopify.engineering/a-packwerk-retrospective
- Spring Modulith reference documentation: Fundamentals: https://docs.spring.io/spring-modulith/reference/fundamentals.html
- import-linter documentation: Forbidden contract: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/
- Redgate: Handling multiple schemas in the same database with Flyway: https://www.red-gate.com/blog/handling-multiple-schemas-in-the-same-database-with-flyway/
- Spring Modulith reference: integration testing application modules: https://docs.spring.io/spring-modulith/reference/testing.html
- Self-Contained Systems: https://scs-architecture.org/
- SCS compared with microservices: https://scs-architecture.org/vs-ms.html

SECTION 5.4 SUR 5.8 · INTERMÉDIAIRE · 23 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Tubes et filtres, et architectures en pipeline

AVANT DE LIRE

Cette section s'appuie sur :

- 2.4 Performances et scalabilité
- 5.1 Styles et patrons

Termes déjà vus qu'elle utilise : Débit (throughput), Idempotent, Test d'architecture, Schéma

L'IDÉE EN BREF

Pensez à une station de lavage automatique. Chaque voiture passe par une série de postes : trempage, brossage, rinçage, séchage. Chaque poste fait un seul travail, ne sait rien des autres et se contente de faire passer la voiture au suivant sur un tapis roulant. Vous pouvez ajouter un poste de cirage sans modifier les autres, et si le séchage est lent, les voitures commencent à faire la queue devant lui. Le style tubes et filtres fonctionne de la même façon pour les données : chaque enregistrement passe par une chaîne d'étapes simples (les filtres), transporté de l'une à l'autre par un tapis roulant (le tube).

CE QU'IL FAUT SAVOIR

- Le style tubes et filtres organise le traitement en une chaîne de filtres indépendants reliés par des tubes à sens unique ; les filtres ne partagent aucun état et ne connaissent pas leurs voisins, seulement le format des données qu'ils lisent et écrivent.
- Les filtres sont de quatre sortes : producteurs (sources), transformateurs, testeurs (qui laissent passer ou écartent des enregistrements) et consommateurs (puits).
- Doug McIlroy a proposé de raccorder les programmes comme des tuyaux d'arrosage dans une note de 1964 ; Ken Thompson a ajouté les tubes à Unix en 1973, et la philosophie Unix des petits outils reliés par des tubes a rendu le style célèbre.
- Les usages typiques sont l'ETL et les pipelines de données, les phases de compilation, le traitement multimédia et le traitement des journaux ; la même structure s'exécute en mode batch (entrée bornée) ou en streaming (entrée non bornée).
- Le débit d'un pipeline est fixé par son filtre le plus lent, et un producteur rapide submergera un consommateur lent, à moins que les tampons ne soient bornés et que le back pressure ne ralentisse le producteur.

- Le style obtient une note élevée en simplicité, en modularité et en réutilisation des filtres, et une note faible en interactivité ainsi qu'en gestion d'un état partagé ou d'une reprise sur erreur complexe entre étapes.

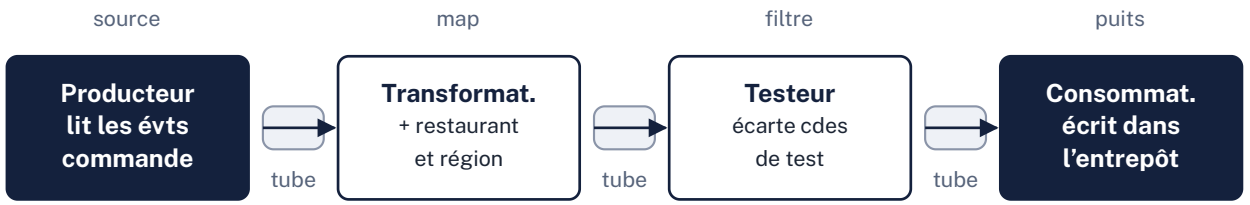
Définition et origine

Dans le style tubes et filtres (pipes and filters), le traitement est une chaîne d'étapes indépendantes. Chaque filtre lit des données sur son entrée, effectue un seul travail sur elles et écrit le résultat sur sa sortie. Chaque tube (pipe) transporte les données dans un seul sens d'un filtre au suivant, généralement avec un tampon entre les deux. La description de Garlan et Shaw (1994) énonce les contraintes qui donnent au style ses propriétés : les filtres doivent être indépendants, ne doivent pas partager d'état avec d'autres filtres et ne doivent pas connaître l'identité des filtres situés à chaque extrémité de leurs tubes. Un filtre ne connaît que le format de ce qu'il lit et de ce qu'il écrit.

Le terrain d'élection le plus connu du style est Unix. Dans une note datée du 11 octobre 1964, Doug McIlroy écrivait qu'il devrait exister des moyens de raccorder les programmes comme un tuyau d'arrosage, en vissant un autre segment quand les données doivent être retravaillées d'une autre manière. Ken Thompson a implémenté l'appel système `pipe` et les tubes dans le shell de la Version 3 d'Unix en 1973, et la notation `|` a suivi dans la Version 4. La philosophie Unix qui en est née, écrire des programmes qui font une seule chose et la font bien, et qui collaborent par une interface texte universelle, est le style tubes et filtres appliqué à l'ensemble des outils d'un système d'exploitation. POSA (1996) a catalogué Pipes and Filters comme patron architectural, et *Enterprise Integration Patterns* (2003) de Hohpe et Woolf l'a appliqué à la messagerie entre systèmes.

Richards et Ford appellent ce même style l'architecture en pipeline, et nomment quatre sortes de filtres. Un **producteur** (ou source) démarre le flux, en lisant dans un fichier, une base de données ou une file. Un **transformateur** modifie chaque enregistrement : analyse syntaxique, conversion, enrichissement. Un **testeur** vérifie une condition et transmet l'enregistrement ou l'écarte, parfois en l'aiguillant vers l'une de plusieurs sorties. Un **consommateur** (ou puits) termine le flux, en écrivant dans une base de données, un fichier ou sur un écran.

Filtres et tubes



Chaque filtre ne connaît que ses formats d'entrée et de sortie, jamais ses voisins

Les tubes stockent et transmettent à sens unique

Quatre sortes de filtres reliés par des tubes ; chaque filtre peut être développé, testé et remplacé indépendamment.

Comment ça marche

Les contraintes produisent directement les avantages. Comme un filtre ne partage aucun état et ne connaît pas ses voisins, il peut être écrit et testé isolément, avec une liste d'enregistrements en entrée et une liste de sorties attendues ; il peut être réutilisé dans d'autres pipelines ; et les filtres peuvent être ajoutés, retirés ou réordonnés en modifiant uniquement la façon dont le pipeline est assemblé. Comme les enregistrements circulent de façon incrémentale, un filtre peut commencer à traiter le premier enregistrement avant que le filtre précédent ait fini le dernier : les étapes s'exécutent donc de façon concurrente, et l'ensemble du pipeline n'a besoin de mémoire que pour quelques enregistrements à la fois, et non pour tout le jeu de données.

Le tube masque la façon dont les données circulent. Dans un shell Unix, c'est un tampon du système d'exploitation entre deux processus ; sous Linux, il contient 64 Kio par défaut, et un écrivain qui le remplit se bloque simplement jusqu'à ce que le lecteur rattrape son retard. Dans un programme Python, ce peut être un générateur, de sorte que chaque filtre tire l'enregistrement suivant du précédent. Dans une plateforme de données, ce peut être une file de messages ou un topic dans un journal comme Apache Kafka, ce qui permet aux filtres de s'exécuter sur des machines différentes et de survivre aux redémarrages. Le code des filtres change à peine d'une solution à l'autre ; les propriétés opérationnelles, elles, changent beaucoup.

La forme stricte connaît des variantes. Un pipeline peut se ramifier (fan-out : un testeur aiguille les enregistrements vers deux consommateurs différents) ou converger (fan-in : deux producteurs alimentent un même transformateur). Il peut être linéaire, ce qui est le plus simple à raisonner, ou former un graphe orienté acyclique d'étapes, comme dans les outils de workflow tels qu'Apache Airflow. À strictement parler, un filtre doté d'un état interne, par exemple un

filtre qui compte ou qui trie, rompt la propriété incrémentale, car il doit voir toute l'entrée avant de produire une sortie ; `sort` dans un pipeline Unix est un tel filtre. Ces filtres sont autorisés, mais ils deviennent les points où la mémoire, la latence et les redémarrages demandent réflexion.

Exemples : où l'on trouve des pipelines

ETL et pipelines de données. Les traitements Extract, transform, load (extraire, transformer, charger) déplacent les données des systèmes opérationnels vers un entrepôt de données : extraire les commandes de la veille, les nettoyer et les convertir, les joindre aux données des restaurants, les charger dans des tables de reporting. Chaque étape est un filtre, et les plateformes de données modernes (Apache Beam, Spark, modèles dbt, services cloud de flux de données) sont en grande partie des outils pour construire et exécuter de tels pipelines à grande échelle.

Compilateurs. Un compilateur classique est un pipeline de phases : l'analyse lexicale transforme des caractères en lexèmes (tokens), l'analyse syntaxique transforme les lexèmes en arbre syntaxique, l'analyse sémantique vérifie les types et les noms, l'optimisation réécrit une représentation intermédiaire, et la génération de code émet du code machine. Chaque phase consomme une représentation et produit la suivante, si bien que les phases peuvent être développées et remplacées séparément ; le projet LLVM, par exemple, permet à de nombreux frontends de langages de partager un même optimiseur et un même ensemble de backends. Les vrais compilateurs assouplissent le style strict, car les phases partagent une table des symboles.

Traitement multimédia. Le transcodage vidéo et le traitement audio enchaînent décodeurs, redimensionneurs, filtres et encodeurs. GStreamer construit explicitement les applications multimédias comme des pipelines d'éléments reliés par des pads, et les graphes de filtres de FFmpeg fonctionnent de la même façon.

Traitement des journaux et des événements. Les expéditeurs de journaux comme Logstash et Fluent Bit se configurent en étapes d'entrée, de filtrage et de sortie : lire les lignes de journal, les découper en champs, écarter le bruit des contrôles de santé, masquer les données personnelles et les envoyer vers un index de recherche. Les données d'observabilité de la plateforme de livraison de repas circulent de cette façon.

Pipelines de caractéristiques et d'inférence en apprentissage automatique. La préparation des données d'entraînement, ainsi que le prétraitement et le post-traitement autour d'un modèle, sont couramment construits comme des pipelines, auxquels revient le chapitre 12.

```
# Top ten URLs that returned HTTP 500 in today's access log
grep '" 500 ' access.log \
  | awk '{print $7}' \
  | sort \
  | uniq -c \
  | sort -rn \
  | head -10
```

Un pipeline Unix : six petits outils, chacun étant un filtre, trouvent les URL qui ont le plus souvent échoué avec une erreur HTTP 500 (dans le format courant des journaux d'accès, le code de statut suit la ligne de requête entre guillemets).

```
import json

def read_lines(path):                # producer
    with open(path) as f:
        for line in f:
            yield line

def parse(lines):                    # transformer
    for line in lines:
        yield json.loads(line)

def only_failed(events):             # tester
    for e in events:
        if e["status"] == "payment_failed":
            yield e

def count_by_provider(events):       # consumer
    counts = {}
    for e in events:
        counts[e["provider"]] = counts.get(e["provider"], 0) + 1
    return counts

events = only_failed(parse(read_lines("orders.jsonl")))
print(count_by_provider(events))
```

La même idée avec des générateurs Python : chaque fonction est un filtre, et les enregistrements circulent un par un, si bien que le fichier n'est jamais chargé en mémoire d'un seul coup.

Batch et streaming

La même structure s'exécute selon deux modes. Le traitement par lots traite une entrée bornée, comme toutes les commandes de la veille, s'exécute jusqu'au bout et s'arrête. Il est simple à raisonner : si une exécution échoue, corrigez le problème et relancez-la, à condition que chaque exécution remplace sa sortie au lieu de s'y ajouter. Les versements nocturnes aux

restaurants, les factures mensuelles et les chargements de l'entrepôt de données sont des pipelines batch typiques. Leur faiblesse est le délai : les résultats ont l'âge de la dernière exécution.

Le traitement de flux (stream processing) traite une entrée non bornée, en traitant chaque enregistrement peu après son arrivée : positions des livreurs, événements de paiement, clics. Les résultats sont frais, mais de nouvelles questions apparaissent. Un flux ne se termine jamais, donc les agrégats doivent être calculés sur des fenêtres (commandes par restaurant par tranche de cinq minutes) ; des enregistrements peuvent arriver en retard ou dans le désordre ; et le pipeline doit fonctionner en continu, se remettre des plantages sans perdre ni compter deux fois des enregistrements, et faire face aux pics de débit en entrée. Des frameworks comme Kafka Streams, Apache Flink et Apache Beam fournissent des fenêtres, des points de reprise (checkpoints) et des magasins d'état pour ces problèmes. Le modèle de Beam traite le batch comme un cas particulier du streaming, avec une entrée bornée, si bien qu'une même définition de pipeline peut s'exécuter dans les deux modes.

Beaucoup d'organisations utilisent les deux : un flux pour des tableaux de bord opérationnels frais mais approximatifs, et un traitement batch qui recalcule les chiffres exacts pendant la nuit. La section 7.8 examine ce choix du point de vue de l'architecture des données.

EXEMPLE COMMENTÉ

Exemple commenté : trouver le goulet d'étranglement

La plateforme de livraison de repas charge ses événements de commande dans son entrepôt de données au moyen de quatre filtres. Mesuré sur un seul worker, chaque filtre peut traiter à lui seul le nombre d'enregistrements par seconde indiqué ci-dessous.

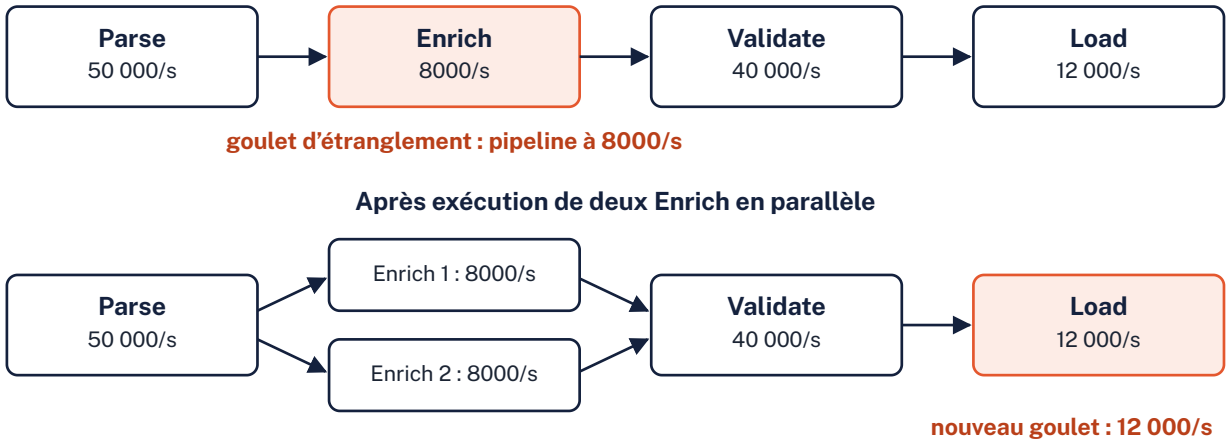
Capacités	Parse 50 000/s ; Enrich (recherche le restaurant et la région) 8000/s ; Validate 40 000/s ; Load 12 000/s.
Débit du pipeline	Un pipeline ne peut pas aller plus vite que son filtre le plus lent : min(50 000 ; 8000 ; 40 000 ; 12 000) = 8000 enregistrements par seconde.
Durée pour les données d'une journée	Un jour de forte activité, la plateforme produit 1 000 000 d'événements de commande. 1 000 000 ÷ 8000 = 125 secondes.
Faire passer le goulet d'étranglement à l'échelle	Enrich est sans état, donc deux instances peuvent s'exécuter en parallèle, chacune prenant la moitié des enregistrements : 2 × 8000 = 16 000/s à cette étape.
Nouveau débit	min(50 000 ; 16 000 ; 40 000 ; 12 000) = 12 000/s. Load est désormais le goulet d'étranglement. 1 000 000 ÷ 12 000 ≈ 83 secondes, soit un gain de 33 % et non des 50 % qu'une lecture naïve laisserait supposer.
Ce que font les tampons	Sans back pressure, Parse pousserait des enregistrements dans le tube précédant Enrich à 50 000/s alors qu'Enrich n'en écoule que 8000/s : le tampon grossirait donc de 42 000 enregistrements par seconde jusqu'à épuisement de la mémoire.

Accélérer un autre filtre que le goulet d'étranglement ne change rien ; mesurez d'abord.

Seuls les filtres sans état peuvent être parallélisés aussi simplement. Un filtre qui conserve un état, comme des totaux cumulés par restaurant, doit partitionner son entrée par clé pour que chaque instance voie tous les enregistrements de ses clés.

C'est le même raisonnement que la loi d'Amdahl et le taux d'utilisation de la section 2.4 : l'étape séquentielle la plus lente fixe la limite.

Faire passer un pipeline à l'échelle déplace le goulet d'étranglement



Le débit est fixé par le filtre le plus lent ; le paralléliser aide jusqu'à ce qu'un autre filtre devienne le plus lent.

Back pressure et gestion des erreurs

Quand un producteur est plus rapide qu'un consommateur, quelque chose doit céder. Les options sont de mettre en tampon (ce qui ne fait que repousser le problème, sauf si le tampon est borné), d'écarter ou d'échantillonner des enregistrements (acceptable pour certaines métriques, jamais pour les paiements), ou d'appliquer un back pressure (contre-pression) : obliger l'étape rapide à ralentir jusqu'à la vitesse de l'étape lente. Les tubes Unix le font automatiquement, car un tube plein bloque l'écrivain. Les conceptions en mode pull, comme les générateurs Python, le font par construction, car rien n'est produit tant que l'étape suivante ne le demande pas. Les pipelines distribués doivent l'intégrer dès la conception : files bornées, consommateurs qui tirent les données à leur propre rythme depuis un journal, ou protocoles comme la spécification Reactive Streams de la JVM, dans laquelle un abonné signale combien d'enregistrements il est prêt à recevoir. La section 9.3 traite en profondeur des files d'attente et du back pressure.

Les erreurs exigent une politique pour chaque filtre, décidée à l'avance. Un **enregistrement malformé** (un message empoisonné, ou poison message) ne doit pas arrêter le pipeline : aiguillez-le vers une file de lettres mortes (dead-letter queue) ou un fichier d'erreurs avec le motif, comptez-le, et déclenchez une alerte si le taux augmente. Une **défaillance transitoire**, comme un service de recherche qui dépasse son délai d'expiration, donne lieu à de nouvelles tentatives avec backoff. Un **plantage** du pipeline exige un point de reprise : les traitements batch sont relancés depuis le début ou depuis un point de reprise (checkpoint), et les frameworks de streaming enregistrent périodiquement leur position et leur état. Comme les redémarrages impliquent que certains enregistrements soient traités deux fois, les

consommateurs qui écrivent des résultats doivent être idempotents, par exemple en faisant un upsert sur une clé naturelle plutôt qu'une insertion à l'aveugle. Les erreurs sont le point faible du style : les filtres s'ignorent délibérément les uns les autres, si bien qu'une défaillance découverte à l'étape quatre ne peut pas facilement annuler ce que l'étape deux a déjà fait, et un pipeline qui a besoin de ce type de retour arrière coordonné relève probablement d'un autre style.

Caractéristiques des tubes et filtres (déploiement unique)

Caractéristique	Évaluation	Justification
Simplicité	Élevé	Chaque filtre fait une seule chose ; le flux est facile à dessiner, à lire et à expliquer.
Modularité et réutilisation	Élevé	Les filtres ne partagent aucun état et ne connaissent que des formats de données, donc ils peuvent être réutilisés, réordonnés et remplacés.
Testabilité	Élevé	Un filtre se teste avec des exemples d'entrée et la sortie attendue, sans ses voisins.
Coût global	Élevé (peu coûteux)	Un seul programme ou traitement ; peu d'infrastructure, sauf en cas de distribution.
Débit (throughput)	Moyen à élevé	Les étapes s'exécutent de façon concurrente et les étapes sans état se parallélisent, mais le filtre le plus lent plafonne l'ensemble.
Latence et interactivité	Bas	Conçu pour des flux de données, pas pour des échanges requête-réponse avec des utilisateurs ; chaque étape ajoute du délai.
Déployabilité	Bas à moyen	Dans un seul programme ou traitement, toute modification redéploie l'ensemble du pipeline ; des filtres répartis sur des files peuvent être mis en production séparément.
Tolérance aux pannes	Bas à moyen	Dans un seul processus, un plantage arrête l'ensemble du pipeline ; points de reprise, nouvelles tentatives et files de lettres mortes doivent être ajoutés.
État partagé et transactions	Bas	Les filtres sont délibérément sans état et indépendants ; la cohérence ou le retour arrière entre étapes est malaisé.
Scalabilité	Faible à moyen dans un seul processus	Plus élevé quand les filtres sont répartis sur des files, au prix d'un effort d'exploitation et de garanties de livraison à gérer.

SI VOUS DÉBUTEZ

Pas à pas : quand un pipeline échoue à mi-chemin

Le point qui fait le plus souvent trébucher ici est ce qui se passe quand un pipeline échoue en cours de route. Les filtres ne savent rien les uns des autres, il n'existe donc pas de moyen simple d'annuler le travail déjà fait, et un redémarrage traite certains enregistrements deux fois. Suivez le chargement d'une journée dans l'entrepôt de données à travers trois sortes de défaillances.

- Une fois par jour, la plateforme de livraison de repas charge dans l'entrepôt de données les enregistrements des livraisons effectuées par ses livreurs, au moyen de quatre filtres : Parse, Enrich (qui recherche la zone de chaque livreur), Validate et Load.
- Un enregistrement est malformé et Parse ne peut pas le lire. Plutôt que d'arrêter tout le traitement pour un seul mauvais enregistrement, Parse le met de côté dans une file de lettres mortes, note pourquoi, le compte et continue.
- Ensuite, le service de recherche dont dépend Enrich cesse de répondre pendant quelques secondes. C'est une défaillance transitoire, donc Enrich fait de nouvelles tentatives avec backoff, en attendant un peu plus longtemps à chaque fois.
- Puis le traitement plante après que Load a écrit 300 000 des 450 000 enregistrements de la journée. Ces lignes restent : les filtres ne se coordonnent pas, donc rien dans le pipeline ne peut facilement les annuler.
- La reprise la plus simple consiste à relancer le traitement depuis le début. Mais si Load se contente d'insérer des lignes, les 300 000 premières livraisons apparaissent maintenant deux fois, et chaque rapport construit sur elles les compte deux fois.
- On rend donc Load idempotent : il fait un upsert sur l'identifiant propre à chaque livraison, en ajoutant une ligne si la livraison est nouvelle et en la remplaçant si elle existe déjà. Relancer le traitement laisse alors exactement une ligne par livraison.
- **La leçon :** décidez de la politique d'erreur de chaque filtre avant la première défaillance. Mettez de côté les mauvais enregistrements, réessayez en cas de défaillance brève, et faites en sorte que la dernière étape puisse être répétée sans risque, car les redémarrages traiteront certains enregistrements deux fois.

Quand l'utiliser, quand l'éviter, et les anti-patterns

Utilisez les tubes et filtres quand le problème est naturellement un flux de données à sens unique à travers des transformations distinctes : ingestion, conversion, enrichissement, validation, agrégation, export. C'est le choix par défaut en ingénierie des données, pour le traitement de fichiers et de contenus multimédias, et pour tout traitement dont les étapes

changent plus souvent que le flux global. Il apparaît aussi à l'intérieur d'autres styles : un traitement de reporting nocturne au sein d'un monolithe modulaire, ou un service de traitement de flux au sein d'un système piloté par les événements.

Évitez-le comme structure principale des applications interactives, où les utilisateurs attendent des réponses immédiates et conversationnelles et où le travail n'est pas un flux ; pour une logique qui exige que de nombreuses étapes partagent et modifient le même état ; et pour les processus qui nécessitent un retour arrière coordonné entre étapes, qui appellent une transaction ou une saga.

Anti-patterns courants : **le filtre divin**, une seule étape qui analyse, enrichit, valide et écrit, ce qui fait perdre les avantages du style ; **l'état partagé caché**, des filtres qui communiquent par une table de base de données ou une variable globale partagée au lieu du tube, ce qui réintroduit le couplage que le style supprime ; **la dérive de format**, un producteur qui change silencieusement son format de sortie et casse tous les filtres en aval, ce qu'empêchent des schémas explicites et des vérifications de schéma (voir la section 7.7) ; **les tampons non bornés**, qui transforment un consommateur lent en plantage par manque de mémoire ; et **les puits non idempotents**, qui comptent en double après chaque nouvelle tentative.

Faire respecter le style dans le code

Les contraintes du style peuvent être rendues explicites. Donnez à chaque filtre la même signature simple (un itérable d'enregistrements en entrée, un itérable en sortie), pour que la composition soit mécanique. Définissez le format des enregistrements à chaque tube sous forme de schéma (une dataclass, un JSON Schema, une définition Avro ou Protocol Buffers) et validez-le à la frontière. Gardez les filtres exempts d'état mutable partagé : un test d'architecture peut interdire aux modules de filtres d'importer la couche de base de données ou de s'importer entre eux, de sorte que le seul moyen pour les données de passer de l'un à l'autre soit le tube. Testez chaque filtre avec de petites fixtures d'entrée et de sortie, et testez le pipeline assemblé de bout en bout avec un échantillon représentatif, enregistrements malformés compris, pour prouver que la politique d'erreur fonctionne.

Dans les pipelines distribués, surveillez le débit par étape, la profondeur des files et le retard des consommateurs (consumer lag) : une file qui grossit devant une étape est le signe visible d'un goulet d'étranglement ou d'un filtre défaillant, et c'est la première métrique sur laquelle alerter.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Quelles trois contraintes Garlan et Shaw ont-ils imposées aux filtres ?

Les filtres doivent être des entités indépendantes, ne doivent pas partager d'état avec d'autres filtres et ne doivent pas connaître l'identité des filtres situés en amont ou en aval.

▼ Pourquoi la commande Unix sort rompt-elle la propriété incrémentale d'un pipeline ?

Elle doit lire toute son entrée avant de pouvoir produire la première ligne, donc elle garde l'ensemble des données et retarde tout ce qui la suit jusqu'à la fin de son entrée.

▼ Un pipeline a des étapes qui peuvent traiter 20 000, 5000 et 15 000 enregistrements par seconde. Quel est son débit, et que se passe-t-il si l'étape du milieu est doublée ?

5000 par seconde, fixé par l'étape la plus lente. La doubler à 10 000 fait fonctionner le pipeline à 10 000 par seconde, toujours en dessous des deux autres étapes.

▼ Pourquoi le consommateur final d'un pipeline redémarrable doit-il être idempotent ?

Parce qu'un redémarrage ou une nouvelle tentative peut livrer certains enregistrements deux fois ; une écriture idempotente, comme un upsert sur une clé naturelle, donne le même résultat quel que soit le nombre de fois où un enregistrement arrive.

▼ À vous : Un pipeline d'événements de commande a les filtres Parse 40 000/s, Enrich 6000/s, Validate 30 000/s et Load 10 000/s, avec 9000 enregistrements/s qui arrivent au pic. Que se passe-t-il ? Combien de workers Enrich règlent le problème, et qu'est-ce qui limite ensuite le débit ?

Le débit est fixé par le filtre le plus lent, Enrich à 6000/s : un arriéré croît donc de 3000 enregistrements par seconde en pointe. Deux workers Enrich donnent 12 000/s, à supposer que la montée en charge soit linéaire. Load devient alors la limite à 10 000/s, ce qui dépasse encore les 9000/s mais ne laisse qu'environ 10 % de marge. Gardez les tampons bornés, avec de la back pressure, afin qu'une étape lente ne puisse pas épuiser la mémoire.

SOURCES

- Doug McIlroy's 1964 memo on coupling programs (cat-v.org): https://doc.cat-v.org/unix/pipes/
- Enterprise Integration Patterns: Pipes and Filters: https://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html
- Microsoft Azure Architecture Center: Pipes and Filters pattern: https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters
- Python documentation: Functional Programming HOWTO (generators): https://docs.python.org/3/howto/functional.html

SECTION 5.5 SUR 5.8 · INTERMÉDIAIRE · 26 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Architectures micronoyau et à plug-ins

AVANT DE LIRE

Cette section s'appuie sur :

- 0.6 API et intégration
- 3.3 Masquage de l'information
- 5.1 Styles et patrons

Termes déjà vus qu'elle utilise : Unité de déploiement, Changement cassant (breaking change), Interface

L'IDÉE EN BREF

Pensez à une console de jeux. La console elle-même fait quelques tâches générales : elle démarre, affiche un menu, lit les manettes et dessine les images. Chaque jeu est une cartouche qui s'insère par un connecteur standard. Vous pouvez ajouter de nouveaux jeux pendant des années sans ouvrir la console, et un jeu mal conçu ne peut pas en modifier le câblage. Une architecture micronoyau construit le logiciel de la même façon : un noyau petit et stable fait le travail général, et des plug-ins ajoutent les parties qui varient, par un connecteur convenu appelé contrat.

CE QU'IL FAUT SAVOIR

- Une architecture micronoyau (à plug-ins) comporte un système central minimal et des plug-ins indépendants qui ne s'y connectent que par des contrats publiés, découverts au moyen d'un registre.
- Elle convient aux produits dont le flux de travail est stable et dont certaines étapes ont de nombreuses variantes, comme les règles fiscales par pays, les moyens de paiement, les importateurs de fichiers ou les langages d'un éditeur.
- Les plug-ins ne doivent dépendre que du contrat, jamais les uns des autres ni des éléments internes du noyau, et le noyau ne doit jamais contenir de cas particuliers pour des plug-ins nommés.
- L'isolation est un compromis : les plug-ins internes au processus sont rapides mais partagent le sort de l'hôte ; les processus séparés, les sandboxes ou les appels distants protègent le noyau, mais ont un coût.

- Versionnez le contrat séparément, vérifiez la compatibilité au chargement des plug-ins et privilégiez les changements additifs ; la loi de Hyrum implique que tout ce que les plug-ins peuvent observer devient une partie du contrat.
- Les micronoyaux applicatifs visent l'extensibilité ; les micronoyaux de systèmes d'exploitation comme L4 ou QNX visent l'isolation et un petit noyau de confiance.

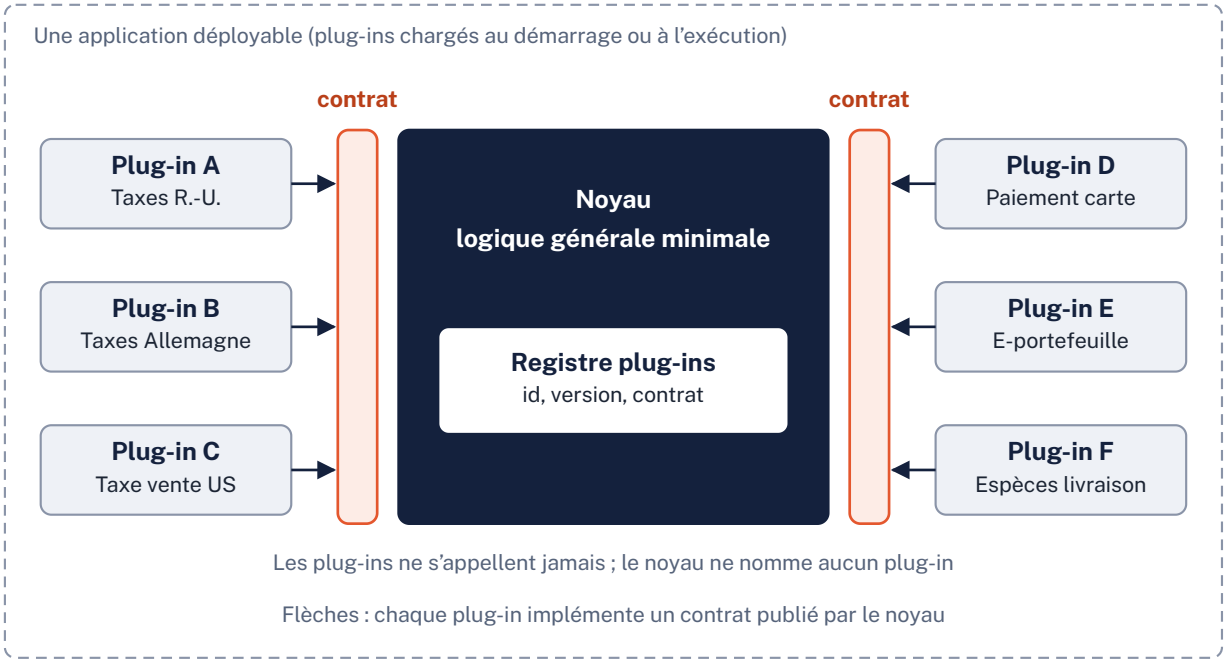
Définition et origine

Une architecture micronoyau, aussi appelée architecture à plug-ins, divise une application en deux sortes de parties : un système central qui contient les fonctionnalités minimales et générales nécessaires pour fonctionner, et des plug-ins qui ajoutent ou modifient des comportements sans que le noyau soit modifié. Les plug-ins ne se connectent au noyau que par des contrats publiés, et le noyau les trouve au moyen d'un registre. Mark Richards et Neal Ford le cataloguent parmi les styles d'architecture classiques dans *Fundamentals of Software Architecture*, et il correspond à la description d'un style architectural donnée à la section 5.1 : un ensemble nommé de contraintes portant sur les éléments (noyau, plug-ins) et les connecteurs (contrats) d'un système.

L'idée est née dans les logiciels produits : des outils installés par de nombreux clients, chacun voulant quelque chose de légèrement différent. Plutôt que de maintenir une version distincte pour chaque client, l'éditeur livre un seul noyau et laisse les clients, les partenaires ou ses propres équipes ajouter des plug-ins. Les environnements de développement intégrés, les navigateurs web, les éditeurs graphiques et les logiciels de comptabilité fonctionnent tous ainsi. Le nom est emprunté aux systèmes d'exploitation, où un micronoyau est un très petit noyau autour duquel tout le reste s'exécute à l'extérieur ; les deux idées sont liées mais ne doivent pas être confondues, comme expliqué plus loin.

Une application micronoyau forme normalement une seule unité de déploiement ; elle appartient donc aux styles monolithiques de ce chapitre : une seule unité de déploiement, un seul processus, un seul ensemble de préoccupations d'exploitation. Les plug-ins peuvent être installés séparément, mais à l'exécution ils vivent à l'intérieur du noyau, ou juste à côté.

La forme d’un système micronoyau



Le système central possède le flux de travail général et un registre ; chaque plug-in implémente un contrat publié par le noyau, et les plug-ins ne dépendent jamais les uns des autres.

Fonctionnement : noyau, plug-ins, registre et contrats

Le système central contient ce dont chaque installation a besoin : le flux de travail principal, des services partagés comme la journalisation ou la configuration, et les points d’extension où se branchent les plug-ins. Richards et Ford le décrivent comme le chemin nominal à travers l’application, avec peu ou pas de traitement personnalisé ; dans l’étape de paiement d’une plateforme de livraison de repas, cela consiste à calculer le prix d’un panier, encaisser le paiement et confirmer la commande. Un bon noyau est volontairement ennuyeux. Chaque cas particulier qui varie selon le client, le pays ou la gamme de produits est un candidat à en sortir.

Les plug-ins sont des modules indépendants et autonomes qui contiennent un traitement spécialisé. Ils ne doivent dépendre que du contrat, ni les uns des autres ni des classes internes du noyau. Quand un plug-in a besoin de quelque chose qu’un autre fournit, ce besoin doit être satisfait par le noyau (sous la forme d’un service qu’il expose par le contrat), ou bien les deux plug-ins doivent être fusionnés ; les dépendances de plug-in à plug-in transforment discrètement une étoile bien nette en enchevêtrement.

Le registre enregistre quels plug-ins sont disponibles, comment chacun s’appelle, quelle est sa version, quel contrat il implémente et comment l’atteindre. Il peut être aussi simple qu’un dictionnaire rempli au démarrage, un répertoire que le noyau parcourt à la recherche de manifestes, des entrées dans les métadonnées d’un paquet (comme avec les points d’entrée

Python), une table de base de données ou un service distant qui liste des plug-ins déployés ailleurs. C'est le registre qui permet au noyau de dire « donne-moi les règles fiscales de l'Allemagne » sans savoir qu'une classe appelée `GermanTax` existe.

Les contrats définissent ce qu'un plug-in doit fournir et ce sur quoi il peut compter : une interface avec des signatures de méthodes, les structures de données qui la traversent, le comportement en cas d'erreur et une version. Dans Eclipse, un contrat est un point d'extension déclaré dans le manifeste d'un plug-in avec un schéma ; dans Visual Studio Code, c'est l'API d'extension plus les points de contribution listés dans le `package.json` d'une extension ; dans votre propre code, c'est généralement une interface (ou un `Protocol` Python) placée dans un petit paquet à part. Lorsque le plug-in vient d'une autre organisation et ne peut pas suivre votre contrat, un adaptateur fait la traduction entre les deux, exactement comme dans l'architecture hexagonale.

Les plug-ins peuvent être connectés de trois façons. Les appels **point à point, dans le processus** sont la norme : le noyau appelle une méthode sur l'objet plug-in. Les plug-ins **en processus séparé** s'exécutent dans leur propre processus et communiquent avec le noyau par des messages interprocessus, ce qui protège le noyau des plantages. Les plug-ins **distants** se trouvent derrière une interface HTTP ou de messagerie ; cela permet des plug-ins écrits dans d'autres langages ou détenus par d'autres équipes, mais ajoute de la latence réseau et des modes de défaillance, et le système commence à ressembler aux styles distribués du chapitre 6.

SI VOUS DÉBUTEZ

Pas à pas : un noyau qui ne nomme jamais ses plug-ins

L'idée la plus difficile de cette section est la manière dont un noyau peut utiliser des plug-ins sans savoir lesquels existent. Le noyau ne dépend que du contrat, et un registre lui indique à l'exécution quels plug-ins sont installés. Suivez l'étape de paiement de la plateforme de livraison de repas lorsqu'elle ajoute un pays.

- Aujourd'hui, le code de l'étape de paiement calcule lui-même la taxe, dans des branches comme : si le pays est GB, appliquer les règles du Royaume-Uni.
- Se lancer en Allemagne implique d'ouvrir le code de l'étape de paiement, d'ajouter une branche allemande, puis de retester et de remettre en production toute l'étape de paiement. Le noyau connaît chaque pays par son nom, et il grossit à chaque lancement.
- Dans la conception micronoyau, le noyau publie plutôt un contrat : une petite interface qui dit ce que tout ensemble de règles fiscales doit fournir, un moyen de calculer la taxe sur un panier dont le prix est établi, et exactement quelles données la traversent.
- L'équipe chargée de l'Allemagne écrit un plug-in qui implémente le contrat et indique quel pays il dessert, DE. Les règles du Royaume-Uni sortent de l'étape de paiement de la même façon pour devenir un plug-in GB, si bien que l'étape de paiement ne nomme plus aucun pays.
- Au démarrage, le registre trouve les plug-ins installés et enregistre le pays que chacun dessert. Quand un client allemand passe commande, le noyau demande au registre les règles fiscales de DE et les appelle par le contrat, sans savoir quel morceau de code répond.
- Ajouter les États-Unis plus tard fonctionne de la même manière : un nouveau plug-in avec ses propres tests, trouvé par le registre. Le code de l'étape de paiement lui-même ne change pas.
- **La leçon :** le noyau connaît le contrat et le registre, jamais les plug-ins. Toute ligne du noyau qui nomme un plug-in particulier est le signe qu'il manque quelque chose au contrat.

Exemples réels

Eclipse. L'IDE Eclipse est le cas d'école. Son noyau n'est guère plus qu'un plan de travail et un runtime de plug-ins (bâti sur Equinox, une implémentation de la spécification du framework OSGi). Presque tout ce que voit un développeur, des outils Java à l'intégration de la gestion de versions, est un plug-in. Chaque plug-in possède un manifeste qui déclare ce dont il a besoin,

quels points d'extension il offre aux autres et à quelles extensions il se branche. Les plug-ins sont activés de façon paresseuse, seulement quand quelque chose en a réellement besoin, afin que des centaines de plug-ins installés ne ralentissent pas le démarrage.

Visual Studio Code. Les extensions déclarent des événements d'activation (ouvrir un fichier Markdown, exécuter une commande) afin de ne se charger qu'en cas de besoin, et apportent des commandes, des langages, des thèmes et des vues par leur manifeste. Les extensions s'exécutent dans un processus hôte d'extensions séparé, si bien qu'une extension défaillante ne peut pas figer l'interface utilisateur de l'éditeur. C'est un bon exemple de choix de l'isolation par processus pour un écosystème de plug-ins qui compte des milliers d'auteurs tiers.

Navigateurs web. Les extensions de navigateur utilisent un ensemble standard d'API d'extension et doivent déclarer dans un manifeste les permissions dont elles ont besoin (quels sites elles peuvent lire, si elles peuvent voir les onglets ou les téléchargements). Le navigateur fait respecter ces permissions, car on ne peut pas, par défaut, faire confiance aux auteurs d'extensions.

Moteurs de règles métier. Le système de gestion des sinistres d'un assureur peut garder le flux de traitement des sinistres dans le noyau et placer les règles de chaque juridiction (ce qui est couvert, quels documents sont nécessaires, délais légaux) dans des plug-ins. Les moteurs fiscaux, les systèmes de paie et les plateformes de commerce en ligne dotées de plug-ins de moyens de paiement suivent le même modèle : un processus stable, avec des règles ou des prestataires qui varient selon le pays, le produit ou le client.

Outils de développement. Le framework de test pytest est construit autour de la bibliothèque de plug-ins `pluggy` ; nombre de ses propres fonctionnalités sont implémentées sous forme de plug-ins utilisant les mêmes hooks que ceux qu'utilisent les tiers. Les outils de build, les linters et les générateurs de sites statiques font couramment de même.

Micronoyau applicatif ou micronoyau de système d'exploitation

Aspect	Micronoyau de système d'exploitation	Architecture micronoyau applicative (à plug-ins)
Ce qui est minimal	Le noyau : espaces d'adressage, threads, ordonnancement et communication interprocessus	Le système central : le flux de travail général d'une application
Ce qui est hors du noyau	Pilotes de périphériques, systèmes de fichiers et piles réseau, exécutés comme serveurs séparés en espace utilisateur	Les comportements métier ou fonctionnalités variables, comme les règles fiscales, les moyens de paiement ou les langages d'un éditeur
Objectifs principaux	Fiabilité, sécurité et une petite base informatique de confiance	Extensibilité, personnalisation et un écosystème de tiers
Communication entre les parties	Passage de messages entre processus isolés	Généralement des appels dans le processus via une interface ; parfois un processus séparé ou un appel distant
Exemples	Mach, la famille L4 (dont seL4, formellement vérifié), QNX, MINIX 3	Eclipse, Visual Studio Code, navigateurs web, pytest, moteurs de règles d'assurance et fiscales

REMARQUE

Ne confondez pas les deux micronoyaux

Les micronoyaux de systèmes d'exploitation, comme Mach et la famille L4, sortent les pilotes de périphériques et les systèmes de fichiers du noyau pour les placer dans des processus séparés en espace utilisateur, afin qu'un pilote défaillant ne puisse pas faire tomber toute la machine. Linux, à l'inverse, est un noyau monolithique avec des modules chargeables. Les deux conceptions ont fait l'objet d'un célèbre débat entre Andrew Tanenbaum et Linus Torvalds sur Usenet en 1992. Un micronoyau applicatif emprunte la forme (un petit noyau, tout le reste à l'extérieur), mais son objectif habituel est l'extensibilité, pas l'isolation, et ses plug-ins s'exécutent normalement dans le même processus que le noyau.

Isolation, versionnage et compatibilité des plug-ins

Trois questions décident si une architecture à plug-ins reste saine au fil des années : à quel point les plug-ins sont isolés, comment le contrat évolue, et comment l'incompatibilité est détectée.

Isolation. Les plug-ins internes au processus partagent la mémoire, les threads et les permissions du noyau. C'est rapide et simple, mais un seul plug-in avec une boucle infinie, une fuite mémoire ou une faille de sécurité affecte tout. Les options, par ordre croissant d'isolation et de coût, sont : dans le processus avec des appels défensifs (délais d'expiration, capture des exceptions plug-in par plug-in, limites de ressources) ; un processus ou un worker séparé par plug-in ou par groupe de plug-ins ; une sandbox (bac à sable), comme un runtime WebAssembly ou un interpréteur restreint, qui limite ce que le code du plug-in peut toucher ; et un service distant. Choisissez en vous demandant qui écrit les plug-ins. Les règles fiscales de votre propre équipe peuvent s'exécuter dans le processus ; du code arbitraire provenant d'une place de marché publique ne le devrait pas.

Versionnage. Le contrat est une API, et les règles de la section 0.6 sur les changements cassants s'appliquent. Versionnez le contrat séparément du noyau, généralement avec le versionnage sémantique : une nouvelle version majeure pour les changements cassants, une version mineure pour les ajouts rétrocompatibles. Laissez chaque plug-in déclarer pour quelle version du contrat il a été construit, et laissez le noyau déclarer quelles versions il prend en charge. Les changements additifs (nouveaux champs facultatifs, nouveaux hooks facultatifs) permettent aux anciens plug-ins de continuer à fonctionner ; quand un changement cassant est inévitable, prenez en charge l'ancienne et la nouvelle version majeure côte à côte pendant une période de dépréciation annoncée.

Vérifications de compatibilité. Vérifiez la compatibilité au chargement des plug-ins, pas au moment où un client les déclenche pour la première fois. Un registre qui refuse un plug-in incompatible au démarrage, avec un message de journal clair, transforme un bogue à l'exécution en échec de déploiement. L'observation de Hyrum Wright, connue sous le nom de loi de Hyrum, mérite d'être retenue ici : avec suffisamment d'utilisateurs, chaque comportement observable d'une interface finira par être utilisé par quelqu'un, quoi que dise la documentation. Gardez le contrat petit, gardez les éléments internes du noyau hors d'atteinte (Eclipse marque pour cette raison les paquets hors API comme `internal`), et traitez tout ce qui est visible par les plug-ins comme faisant partie du contrat.

Caractéristiques du style micronoyau

Caractéristique	Évaluation	Justification
Extensibilité et modifiabilité (dans les directions prévues)	Élevé	Un nouveau comportement arrive sous la forme d'un nouveau plug-in ; le noyau et les autres plug-ins ne changent pas.
Simplicité	Moyen	L'idée est simple, mais les contrats, le registre, le chargement et les vérifications de version constituent une vraie mécanique.
Testabilité	Moyen à élevé	Chaque plug-in peut être testé seul par rapport au contrat ; le noyau peut être testé avec de faux plug-ins. Les combinaisons doivent tout de même être testées.
Déployabilité	Moyen	On peut souvent ajouter des plug-ins sans redéployer le noyau ; un changement de contrat impose des mises en production coordonnées.
Performances	Moyen à élevé	Les plug-ins internes au processus sont de simples appels de fonction ; l'isolation par processus ou par sandbox ajoute un surcoût.
Scalabilité et élasticité	Bas	C'est généralement une seule unité de déploiement, qui passe donc à l'échelle d'un bloc, comme tout monolithe.
Tolérance aux pannes	Bas à moyen	Un plug-in interne au processus peut planter ou ralentir l'hôte ; l'isolation par processus améliore ce point, mais a un coût.
Coût global	Élevé (peu coûteux)	Une seule base de code et un seul runtime à exploiter ; le coût principal est la conception et l'évolution des contrats.

Quand l'utiliser, et quand l'éviter

Bons cas d'usage. Les logiciels produits que de nombreux clients installent et personnalisent. Les systèmes au flux de travail stable mais avec de nombreuses variantes d'une étape : fiscalité et paie par pays, règles d'assurance par juridiction, moyens de paiement, importateurs de formats de fichiers, transporteurs. Les plateformes qui veulent un écosystème de tiers. Les systèmes où un régulateur ou un client modifie souvent les règles et où vous voulez que ces modifications soient de petites mises en production testées séparément. Le style fonctionne aussi au sein d'un système plus vaste : un service d'une architecture distribuée peut lui-même être un micronoyau.

Mauvais cas d'usage. Les systèmes dont vous ne savez pas encore ce qui varie. Des points d'extension conçus à l'avance pour des besoins imaginés relèvent de la généralité spéculative : ils ajoutent de la mécanique, ralentissent l'équipe et se trouvent généralement au mauvais endroit quand le vrai besoin arrive (voir la discussion sur la réversibilité à la section 1.6). Les

systèmes qui exigent une très grande scalabilité ou une isolation des pannes, puisque l'ensemble passe à l'échelle et tombe en panne comme une seule unité. Les systèmes où les plug-ins devraient se coordonner étroitement entre eux, ce qui annule le principal avantage du style.

Une règle pratique : attendez d'avoir construit deux ou trois fois le même type de variation, puis extrayez le contrat à partir des cas réels. Un contrat dérivé de trois implémentations fiscales qui fonctionnent sera bien meilleur qu'un contrat deviné avant la première.

ATTENTION

Anti-patrons courants

- **Le noyau qui sait tout.** Le noyau contient `if plugin_name == "stripe"` ou des cas particuliers pour certains plug-ins. Chacun est le signe qu'il manque quelque chose au contrat.
- **Noyau obèse.** Des règles variables reviennent en douce dans le noyau parce que c'est plus rapide que d'étendre le contrat, jusqu'à ce que le noyau redevienne l'ancien monolithe.
- **Plug-ins enchevêtrés.** Les plug-ins s'importent mutuellement ou partagent un état caché, si bien qu'ils ne peuvent plus être installés, testés ou retirés indépendamment.
- **Contrat qui fuit.** Le contrat transmet aux plug-ins des objets du domaine du noyau ou des sessions de base de données, si bien que chaque changement interne devient un changement cassant pour chaque plug-in.
- **Contrat non versionné.** Personne ne peut dire quels plug-ins casseront quand le noyau changera, si bien que les mises à niveau deviennent des événements redoutés, faits d'un seul coup.
- **Un plug-in par client.** Les plug-ins deviennent un moyen de maintenir des centaines de forks propres à chaque client. C'est la configuration, et non le code, qui doit exprimer les petites différences entre clients.

EXEMPLE COMMENTÉ

Exemple commenté : concevoir un contrat de plug-in pour des règles fiscales propres à chaque pays

La plateforme de livraison de repas utilisée tout au long de ce cours se lance en Allemagne et aux États-Unis après le Royaume-Uni. Chaque pays taxe différemment les plats, l'alcool et les frais de livraison, et la taxe sur les ventes américaine varie aussi selon l'État et la ville. Le code de l'étape de paiement contient déjà des branches `if country == "GB"`. L'architecte propose un micronoyau : l'étape de paiement est le noyau, et les règles de chaque pays sont un plug-in. Voici les décisions, dans l'ordre où elles ont été prises.

1. Nommer le point de variation	Seul le calcul de la taxe sur un panier dont le prix est établi varie selon le pays. La tarification, les remises, le paiement et les reçus restent dans le noyau. Un point de variation étroit garde le contrat petit.
2. Décider de ce qui traverse le contrat	Entrées : les lignes du panier avec une catégorie neutre (`hot-food`, `alcohol`, `delivery-fee`), des montants hors taxe en `Decimal` et une région facultative. Sortie : le total de la taxe et une ventilation étiquetée pour le reçu. Aucun objet du noyau (commandes, clients, sessions de base de données) n'est transmis, si bien que les plug-ins ne peuvent pas dépendre des éléments internes du noyau.
3. Faire du contrat un paquet séparé	`tax_contract` ne contient que des classes de données et le protocole `TaxRules`, versionné selon le versionnage sémantique. Le noyau et les plug-ins en dépendent tous deux ; aucun ne dépend de l'autre.
4. Définir la compatibilité	Un plug-in déclare la version du contrat dont il a besoin, par exemple (2, 1). Le noyau l'accepte si la version majeure correspond et si la version mineure du noyau est au moins aussi élevée. Ajouter un champ facultatif est un changement mineur ; renommer ou supprimer un champ est un changement majeur.
5. Choisir le mode de découverte	Les plug-ins sont des paquets Python ordinaires qui s'enregistrent sous le groupe de points d'entrée `fooddelivery.tax_rules`. Installer un paquet suffit ; aucun code du noyau ne liste les pays.
6. Décider du comportement en cas d'échec	Si aucun plug-in n'est enregistré pour un pays, l'étape de paiement refuse la commande avec une erreur claire plutôt que d'appliquer une taxe nulle. Un plug-in qui lève une exception ne fait échouer que ce paiement-là, et l'erreur est signalée avec le nom et la version du plug-in.
7. Décider de l'isolation	Les plug-ins sont écrits par les propres équipes pays de l'entreprise et sont dignes de confiance ; ils s'exécutent donc dans le processus, pour la rapidité. Si des éditeurs fiscaux externes étaient autorisés à fournir des plug-ins, la conception les placerait derrière un appel distant avec un délai d'expiration.

8. Spécifier les tests	Une suite de tests de contrat partagée (arrondis, paniers vides, lignes de remboursement négatives, catégories inconnues) s'exécute contre chaque plug-in dans son propre pipeline. Chaque équipe pays y ajoute ses propres cas, tirés des instructions fiscales publiées.

Le noyau ne change plus quand un pays est ajouté : le lancement d'un quatrième pays représente donc un nouveau paquet et ses tests, pas une modification de l'étape de paiement.

La décision la plus difficile est l'étape 2. Une liste de catégories trop grossière oblige les plug-ins à deviner ; une liste trop détaillée couple le contrat au droit d'un seul pays. Attendez-vous à la réviser, et planifiez alors la migration vers une nouvelle version majeure.

Le registre rejette les plug-ins incompatibles au démarrage : une incompatibilité de version apparaît donc lors d'une vérification de déploiement, et non sous la forme d'une taxe erronée sur le reçu d'un client.

```
# package: tax_contract  (version 2.1.0)
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

CONTRACT_VERSION = (2, 1)

@dataclass(frozen=True)
class TaxLine:
    sku: str
    category: str      # "hot-food", "cold-food", "alcohol", "delivery-fee"
    net: Decimal       # never float for money

@dataclass(frozen=True)
class TaxResult:
    total_tax: Decimal
    breakdown: dict[str, Decimal]   # label shown on the receipt

class TaxRules(Protocol):
    country: str                    # ISO 3166-1 alpha-2, e.g. "DE"
    requires_contract: tuple[int, int]
    def calculate(self, lines: list[TaxLine], region: str | None) → TaxResult: ...
```

Le paquet du contrat, versionné et publié séparément du noyau comme des plug-ins.

```
from importlib.metadata import entry_points
from tax_contract import CONTRACT_VERSION

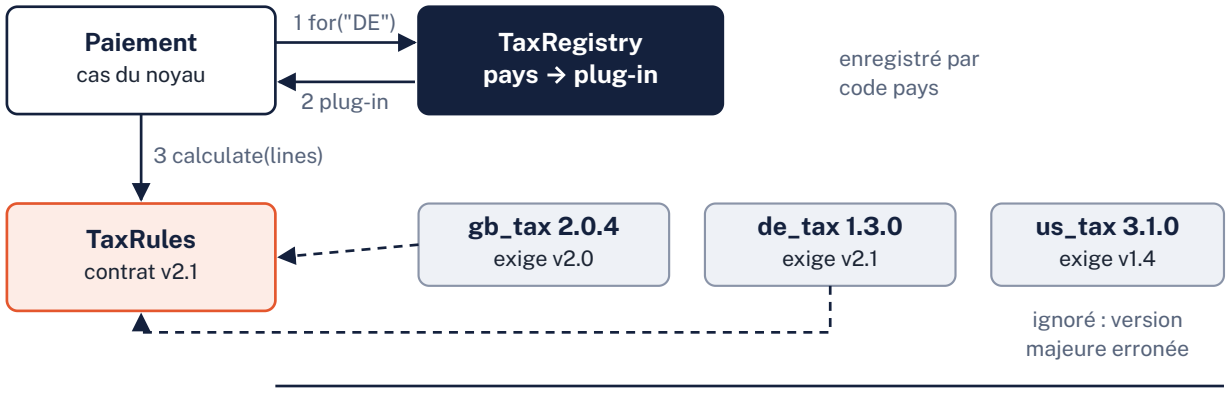
class UnsupportedCountry(Exception): pass

class TaxRegistry:
    def __init__(self, log):
        self._by_country = {}
        for ep in entry_points(group="fooddelivery.tax_rules"):
            plugin = ep.load()()
            major, minor = plugin.requires_contract
            if major ≠ CONTRACT_VERSION[0] or minor > CONTRACT_VERSION[1]:
                log.warning("skipping %s: needs contract %s", ep.name,
plugin.requires_contract)
                continue
            self._by_country[plugin.country] = plugin

    def for_country(self, country):
        if country not in self._by_country:
            raise UnsupportedCountry(country)   # fail loudly, never guess a rate
        return self._by_country[country]
```

Le registre dans le noyau : il découvre les plug-ins installés, vérifie la compatibilité des contrats et les recherche par pays.

Les plug-ins fiscaux à l'exécution



Chargement : au démarrage, le registre lit les paquets installés (points d'entrée), vérifie la version de contrat requise par chaque plug-in et l'enregistre par pays ; les plug-ins incompatibles sont journalisés et ignorés sans planter.

L'étape de paiement demande au registre le plug-in enregistré pour un pays et l'appelle par le contrat ; un plug-in construit pour une version de contrat incompatible est ignoré au démarrage.

Faire respecter le style dans le code

Une architecture à plug-ins ne le reste que si ses règles sont vérifiées automatiquement. La structure qui fonctionne dans la plupart des langages comporte trois sortes de paquets : un petit paquet de **contrat** ne contenant que des interfaces et des types de données ; le **noyau**, qui dépend du contrat ; et un paquet par **plug-in**, chacun dépendant du contrat et de rien d'autre parmi votre code.

Faites ensuite respecter les règles de dépendance avec des outils du type présenté à la section 3.4 : en Python, import-linter peut interdire au noyau d'importer un paquet de plug-in et interdire aux plug-ins de s'importer entre eux ou d'importer le noyau ; en Java, des tests ArchUnit ou des systèmes de modules (JPMS, OSGi) peuvent faire de même. Faites échouer le build quand une règle est enfreinte.

Ajoutez une **suite de tests de contrat** : un ensemble de tests écrits une fois pour toutes par rapport au contrat et exécutés contre chaque plug-in dans le pipeline propre à ce plug-in. Elle vérifie le comportement que promet le contrat (arrondis, entrée vide, types d'erreur), et non le droit fiscal de chaque pays, que couvrent les tests propres au plug-in. Enfin, ajoutez une vérification au démarrage ou un test de fumée (smoke test) de déploiement qui charge chaque plug-in installé et vérifie sa version de contrat, afin que les incompatibilités soient détectées avant que les utilisateurs ne les découvrent.

Questions fréquentes

Un micronoyau n'est-il que le patron Stratégie ?
Au niveau du code, le patron Stratégie est le même mécanisme : choisir à l'exécution une implémentation d'une interface. Le micronoyau en est la version architecturale : les plug-ins sont construits et versionnés séparément, et souvent installés séparément, un registre les découvre, et le contrat est une API publiée avec des règles de compatibilité.

Les plug-ins peuvent-ils avoir leurs propres données ?
Oui, et c'est souvent souhaitable, mais traitez-les comme les données détenues par les modules d'un monolithe modulaire : les tables ou les fichiers d'un plug-in lui appartiennent, et le noyau n'y accède que par le contrat.

La configuration doit-elle remplacer les plug-ins ?
Pour de petites différences relevant des données (un taux, un seuil, un libellé), oui : la configuration est moins coûteuse et plus sûre. Les plug-ins servent aux différences qui exigent de la logique. Beaucoup de systèmes utilisent les deux : un plug-in par pays, plus une configuration pour les taux en vigueur dans chaque pays.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Quels sont les trois éléments qu'une architecture à plug-ins doit définir en plus du noyau lui-même ?

Les plug-ins, les contrats qu'ils implémentent, et un registre qui permet au noyau de découvrir et de rechercher les plug-ins sans les nommer dans le code.

▼ Pourquoi les plug-ins ne doivent-ils pas dépendre les uns des autres ?

Parce qu'ils ne pourraient alors plus être installés, retirés, testés et versionnés indépendamment, ce qui est le principal avantage du style ; les besoins partagés relèvent du noyau ou du contrat.

▼ Pourquoi Visual Studio Code exécute-t-il les extensions dans un processus séparé ?

Pour qu'une extension lente ou qui plante ne puisse pas figer ni faire planter l'interface utilisateur de l'éditeur : il sacrifie un peu de performances et de simplicité en échange d'une isolation vis-à-vis de milliers d'auteurs tiers.

▼ Quand la configuration est-elle préférable à un plug-in ?

Quand la différence entre les variantes relève des données, comme un taux, une limite ou un libellé, plutôt que de la logique. La configuration est plus simple et plus sûre à modifier.

▼ À vous : L'étape de paiement d'une billetterie utilise des plug-ins de moyens de paiement. Le noyau contient if plugin == "voucher", et le plug-in voucher lit la table bookings du noyau et appelle directement le plug-in de carte. Critiquez cela et dites ce que le contrat devrait fournir.

Cela enfreint les règles du style. Le cœur ne doit contenir aucun cas particulier pour des plug-ins nommés, et les plug-ins ne doivent dépendre que du contrat, jamais des éléments internes du cœur ni les uns des autres. Définissez un contrat versionné, avec par exemple can_pay et take_payment, qui transmet les détails de réservation dont le plug-in a besoin. Déplacez la logique des bons derrière lui, et laissez le cœur enchaîner les moyens de paiement pour les paiements fractionnés. Tout ce que les plug-ins peuvent observer devient une partie du contrat : gardez-le donc petit.

SOURCES

- Azad Bolour: Notes on the Eclipse Plug-in Architecture (eclipse.org): https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html
- Visual Studio Code: Extension Host: https://code.visualstudio.com/api/advanced-topics/extension-host

- Python Packaging User Guide: Entry points specification: https://packaging.python.org/en/latest/specifications/entry-points/
- pluggy: the plug-in system behind pytest: https://pluggy.readthedocs.io/en/latest/
- Semantic Versioning 2.0.0: https://semver.org/

SECTION 5.6 SUR 5.8 · INTERMÉDIAIRE · 25 MIN DE LECTURE · DERNIÈRE RELECTURE : NOVEMBRE 2026

Architecture hexagonale, en oignon et Clean Architecture

AVANT DE LIRE

Cette section s'appuie sur :

- 2.6 Modifiabilité et déployabilité
- 3.5 SOLID à grande échelle
- 5.2 En couches

Termes déjà vus qu'elle utilise : Inversion des dépendances, Doublure de test (test double), Couche, Interface, Framework

L'IDÉE EN BREF

Pensez à un adaptateur de voyage. Le chargeur de votre ordinateur portable a une seule forme de fiche ; chaque pays a sa propre prise. Vous ne reconstruisez pas l'ordinateur pour chaque pays : vous emportez de petits adaptateurs qui s'adaptent à la prise d'un côté et à votre fiche de l'autre. L'architecture hexagonale et la Clean Architecture construisent le logiciel de la même façon. La partie importante, les règles métier, a ses propres prises standard (appelées ports). Les bases de données, les sites web, les sociétés de paiement et les outils de test s'y connectent par des adaptateurs, si bien que n'importe lequel peut être remplacé sans toucher aux règles.

CE QU'IL FAUT SAVOIR

- Les architectures hexagonale (ports et adaptateurs, Cockburn, 2005), en oignon (Palermo, 2008) et Clean Architecture (Martin, billet de blog en 2012, livre en 2017) partagent une même idée : la logique métier au centre, indépendante des frameworks et de l'infrastructure, avec des dépendances orientées vers l'intérieur.
- Un port est une interface qui appartient à l'application ; un adaptateur fait la traduction entre un port et une technologie. Les adaptateurs pilotes appellent l'application ; les adaptateurs pilotés implémentent des interfaces que l'application appelle.
- La règle de dépendance dit que le code intérieur ne doit jamais nommer quoi que ce soit d'un anneau extérieur ; l'inversion des dépendances permet au flux de contrôle d'aller vers l'extérieur tandis que les dépendances du code source pointent vers l'intérieur.
- Les principaux bénéfices sont des tests rapides et isolés des règles métier, et la possibilité de remplacer une infrastructure, comme un prestataire de paiement, en écrivant un nouvel adaptateur.

- Les coûts sont des interfaces supplémentaires, du code de mapping et de l'indirection, qui ne valent pas la peine pour de simples applications CRUD ou des outils à courte durée de vie.
- Faites respecter la règle automatiquement, par exemple avec import-linter, la règle onionArchitecture d'ArchUnit ou des modules de build séparés.

Une idée, trois noms

L'architecture hexagonale, l'architecture en oignon et la Clean Architecture sont trois formulations d'une même idée : placer la logique métier au centre, la rendre indépendante des interfaces utilisateur, des bases de données, des frameworks et des services externes, et faire pointer vers elle toutes les dépendances du code source. Elles décrivent la structure interne d'une application ; elles se combinent donc avec n'importe quel style de déploiement : un monolithe unique, un monolithe modulaire dont chaque module est construit de cette façon, ou un microservice.

Elles sont nées d'une frustration envers l'architecture en couches classique. Dans le style en couches de la section 5.2, la couche métier dépend de la couche de persistance, qui dépend de la base de données : les règles métier finissent façonnées par les tables et le mapping objet-relationnel, et tester une règle implique de démarrer une base de données. La famille de styles de cette section inverse cette dépendance grâce au principe d'inversion des dépendances : le code métier déclare les interfaces dont il a besoin, et le code d'infrastructure les implémente. Le principe lui-même est traité à la section 3.5 ; cette section présente les architectures qui reposent sur lui.

L'architecture hexagonale : ports et adaptateurs

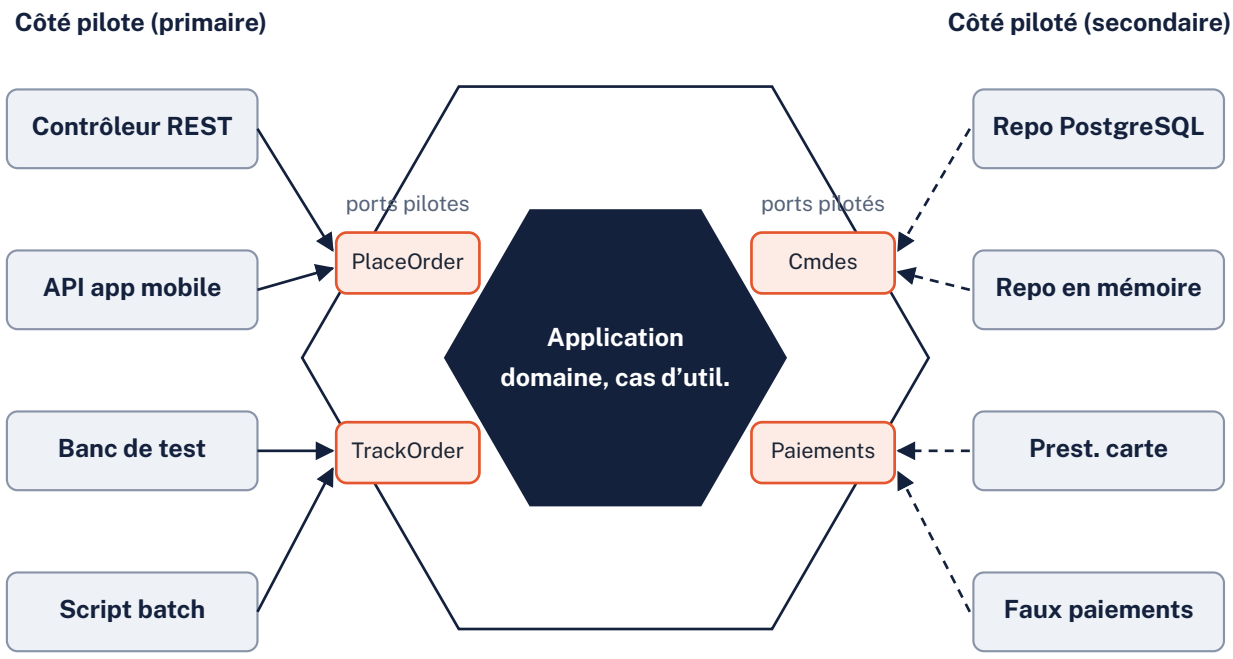
Alistair Cockburn a d'abord décrit l'idée sur le wiki du Portland Pattern Repository, puis l'a rédigée sous forme d'article en 2005 sous le nom plus descriptif de ports et adaptateurs. Son intention déclarée était de «permettre à une application d'être pilotée aussi bien par des utilisateurs que par des programmes, des tests automatisés ou des scripts batch», et de la développer et la tester indépendamment de ses futures bases de données et de ses futurs périphériques. En 2024, Juan Manuel Garrido de Paz et lui ont publié un court livre, *Hexagonal Architecture Explained*, qui reformule le patron.

Un port est une conversation intentionnelle entre l'application et le monde extérieur, exprimée sous forme d'interface : « passer une commande », « stocker les commandes », « débiter une carte ». Un adaptateur est le code qui fait la traduction entre un port et une technologie

particulière : un contrôleur HTTP, un repository SQL, le SDK d'un prestataire de paiement, un script en ligne de commande, une imitation en mémoire. Un même port peut avoir plusieurs adaptateurs. L'application, à l'intérieur de l'hexagone, ne connaît que ses ports.

L'hexagone n'a aucune signification en tant que forme ; Cockburn l'a choisi pour échapper à la représentation des couches de haut en bas et pour laisser de la place afin de dessiner plusieurs ports de chaque côté. Ce qui compte, c'est la frontière entre l'intérieur et l'extérieur, et l'asymétrie entre deux sortes de ports.

Ports pilotes et ports pilotés



Trait plein : l'adaptateur appelle un port. Pointillés : il implémente un port. Toutes les dépendances vont vers l'intérieur.

Les adaptateurs pilotes appellent l'application par des ports pilotes ; l'application appelle des ports pilotés, que des adaptateurs pilotés implémentent, si bien que la base de données et le prestataire de paiement sont des plug-ins remplaçables.

Côté pilote et côté piloté

Les ports et adaptateurs pilotes (primaires) se trouvent à gauche dans la représentation habituelle. Un adaptateur pilote reçoit une requête d'un acteur principal (le navigateur d'un utilisateur, une app mobile, un autre système, un test, une tâche planifiée) et appelle un port pilote, qui est l'API propre de l'application : en pratique, un cas d'utilisation comme `PlaceOrder`. L'adaptateur transforme du HTTP, un message ou un argument de ligne de commande en un appel avec des valeurs du domaine, et retransforme le résultat en réponse.

Les ports et adaptateurs pilotés (secondaires) se trouvent à droite. L'application les appelle quand elle a besoin de quelque chose du monde extérieur : stocker une commande, débiter une carte, envoyer une notification, lire l'heure. Le port est une interface qui appartient à l'application et qui est écrite dans ses termes (`OrderRepository.save(order)`), et un adaptateur piloté l'implémente pour une technologie (PostgreSQL, l'API d'un prestataire de paiement par carte, une passerelle SMS). C'est là que se produit l'inversion des dépendances : à l'exécution, l'application appelle l'adaptateur, mais dans le code source, c'est l'adaptateur qui dépend de l'interface de l'application, et non l'inverse.

Le texte original de Cockburn parle d'acteurs principaux et secondaires ; les termes pilote et piloté sont aujourd'hui plus courants, et il les emploie lui-même. La distinction compte pour les tests : les ports pilotes sont exercés par des bancs de test qui jouent le rôle d'utilisateurs, et les ports pilotés sont remplacés par des doublures de test. Elle compte aussi pour le changement : les nouveaux canaux (un assistant vocal, une API partenaire) arrivent sous la forme de nouveaux adaptateurs pilotes, et les nouvelles infrastructures sous la forme de nouveaux adaptateurs pilotés.

Quelqu'un doit créer les adaptateurs et les brancher sur l'application. Cela se fait dans une seule racine de composition, généralement le code de démarrage du programme, qui est la seule partie du système à connaître chaque adaptateur. Un framework d'injection de dépendances peut faire ce travail, mais de simples appels de constructeurs dans `main` sont souvent plus clairs.

SI VOUS DÉBUTEZ

Pas à pas : appeler vers l'intérieur et appeler vers l'extérieur

L'idée qui demande le plus d'attention ici est la différence entre les deux côtés de l'hexagone. Du côté pilote, un adaptateur appelle l'application ; du côté piloté, l'application appelle un adaptateur. Pourtant, dans le code source, chaque dépendance pointe toujours vers l'intérieur. La section 3.5 a présenté l'inversion des dépendances ; suivez une commande pour voir où elle se produit.

- Un client passe une commande. Le contrôleur REST, un adaptateur pilote, reçoit la requête HTTP, la transforme en valeurs simples comme l'identifiant de la commande et le total, et appelle le cas d'utilisation PlaceOrder. L'appel va vers l'intérieur.
- PlaceOrder enregistre d'abord la commande, puis doit autoriser la carte. Il appelle authorise sur PaymentGateway, une interface que l'application déclare elle-même, dans ses propres termes. Il n'a aucune idée de la société de paiement qui se trouve derrière.
- L'adaptateur du prestataire de paiement par carte, à l'extérieur de l'hexagone, est un adaptateur piloté : il implémente PaymentGateway à l'aide de la boîte à outils propre au prestataire. À l'exécution, l'appel va vers l'extérieur, du cas d'utilisation vers l'adaptateur, mais dans le code source, c'est l'adaptateur qui fait référence à l'interface de l'application, jamais l'inverse.
- L'enregistrement fonctionne de la même façon. PlaceOrder appelle save sur OrderRepository ; PostgresOrderRepository l'implémente avec du SQL, et une version en mémoire implémente le même port pour les tests.
- Il faut bien que quelque chose choisisse les adaptateurs. Cela se fait une seule fois, dans la racine de composition au démarrage, qui crée les adaptateurs PostgreSQL et du prestataire de paiement par carte et les remet à PlaceOrder. Un test fait de même avec des adaptateurs en mémoire et factices.
- **La leçon :** du côté pilote, l'extérieur appelle vers l'intérieur ; du côté piloté, l'intérieur appelle vers l'extérieur, par une interface qui lui appartient. Dans les deux cas, l'application ne nomme jamais une technologie, et c'est ce qui permet de remplacer et de tester les adaptateurs.

L'architecture en oignon

Jeffrey Palermo a décrit l'architecture en oignon dans une série de billets de blog à partir de juillet 2008, à l'intention des applications métier à longue durée de vie au comportement substantiel (il la déconseillait pour les petits sites web). Il a dessiné des anneaux concentriques : le modèle de domaine au centre, les services de domaine autour, les services applicatifs autour de ceux-ci, et un anneau extérieur contenant l'interface utilisateur,

l'infrastructure (y compris la base de données) et les tests. Ses principes clés sont que l'application est construite autour d'un modèle objet indépendant, que les couches intérieures définissent des interfaces que les couches extérieures implémentent, que tout couplage est orienté vers le centre et que le cœur de l'application peut être compilé et exécuté séparément de l'infrastructure.

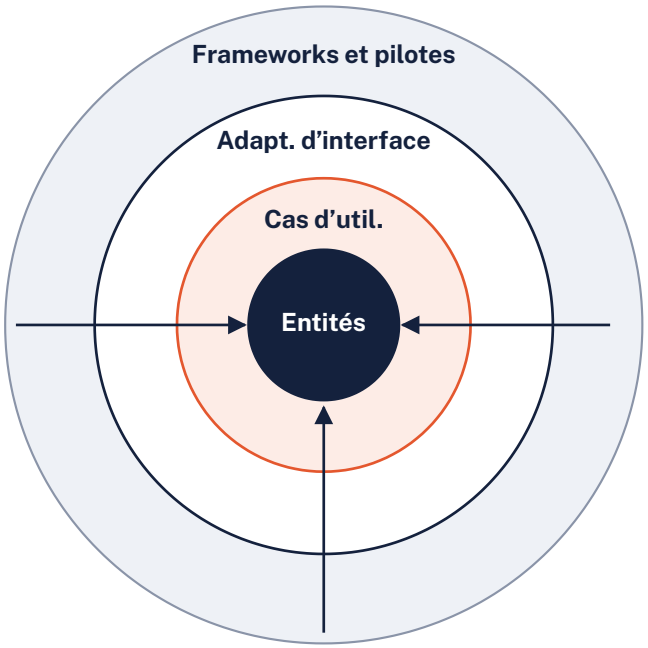
Par rapport à l'architecture hexagonale, l'architecture en oignon ajoute des anneaux nommés à l'intérieur de l'application et rend explicite un point que la représentation de Cockburn laisse implicite : la base de données n'est pas le fondement du système, mais un service externe situé en bordure. Les interfaces de repository vivent dans le cœur ; leurs implémentations vivent dans l'anneau d'infrastructure.

La Clean Architecture et la règle de dépendance

Robert C. Martin a publié « The Clean Architecture » sur son blog en août 2012, comme synthèse de l'architecture hexagonale, de l'architecture en oignon et d'autres propositions similaires, puis l'a développée dans le livre *Clean Architecture* en 2017. Sa représentation compte quatre anneaux. Les **entités** portent les règles métier valables pour toute l'entreprise. Les **cas d'utilisation** portent les règles propres à l'application, qui orchestrent les entités. Les **adaptateurs d'interface** convertissent les données entre la forme dont ont besoin les cas d'utilisation et celle dont ont besoin les agents externes : contrôleurs, présentateurs et passerelles. Les **frameworks et pilotes** forment l'anneau extérieur : le framework web, la base de données, l'interface utilisateur.

La règle de dépendance fait tenir l'ensemble : les dépendances du code source ne pointent que vers l'intérieur, et rien dans un anneau intérieur ne peut nommer quoi que ce soit de déclaré dans un anneau extérieur, qu'il s'agisse d'une classe, d'une fonction ou d'un format de données. Quand le contrôle doit aller vers l'extérieur (un cas d'utilisation qui enregistre une commande), l'anneau intérieur déclare une interface et l'anneau extérieur l'implémente. Les données qui franchissent une frontière doivent être des structures simples, exprimées dans les termes de l'anneau intérieur, jamais une ligne de base de données ni un objet de requête HTTP. Martin soutient aussi que la structure de premier niveau doit révéler la finalité du système (commandes, paiements, livraisons) plutôt que son framework, une idée qu'il appelle screaming architecture (l'architecture qui crie).

Les anneaux de la Clean Architecture



Extérieur : framework web, ORM, BD, courtier de messages, UI

Adapt. d'interface : contrôleurs, présentateurs, impl. des repositories

Cas d'utilisation : règles propres à l'app, ex. PlaceOrder

Entités : règles métier globales, ex. Order, Money

La règle de dépendance : le code ne dépend que vers l'intérieur

Chaque anneau ne peut dépendre que des anneaux situés à l'intérieur ; les entités ne dépendent de rien, et les frameworks et pilotes se trouvent en bordure, là où ils peuvent être remplacés.

Les architectures hexagonale, en oignon et Clean Architecture comparées

Aspect	Hexagonale (ports et adaptateurs)	En oignon	Clean Architecture
Auteur et date	Alistair Cockburn ; rédigée sous forme d'article en 2005	Jeffrey Palermo ; série de billets de blog à partir de juillet 2008	Robert C. Martin ; billet de blog en août 2012, livre en 2017
Représentation	Un hexagone avec un intérieur (l'application) et un extérieur (les adaptateurs), le côté gauche pilote, le côté droit piloté	Anneaux concentriques : modèle de domaine, services de domaine, services applicatifs, puis interface utilisateur, infrastructure et tests	Anneaux concentriques : entités, cas d'utilisation, adaptateurs d'interface, frameworks et pilotes
Règle centrale	L'application ne communique avec le monde que par des ports ; les adaptateurs font la traduction	Tout couplage est orienté vers le centre ; le cœur définit des interfaces que l'anneau extérieur implémente	La règle de dépendance : les dépendances du code source ne pointent que vers l'intérieur
Structure interne prescrite	Aucune : seulement l'intérieur et l'extérieur	Oui : des anneaux intérieurs nommés pour le domaine	Oui : entités et cas d'utilisation, plus des recommandations sur les présentateurs et les données qui franchissent les frontières
Accent principal	Symétrie : utilisateurs, tests, scripts et autres programmes peuvent tous piloter l'application	Placer le modèle de domaine, et non la base de données, au centre d'une application métier	Indépendance vis-à-vis des frameworks, de l'interface utilisateur, de la base de données et des agents externes ; des règles métier testables
Vocabulaire typique	Port, adaptateur, pilote (primaire), piloté (secondaire), acteur	Modèle de domaine, services de domaine, services applicatifs, infrastructure	Entité, cas d'utilisation (interactor), passerelle (gateway), présentateur, contrôleur

Un cas d'utilisation avec un port et deux adaptateurs

L'exemple Python suivant implémente le cas d'utilisation `PlaceOrder` de la plateforme de livraison de repas à la manière hexagonale. Le premier extrait est l'intérieur : une petite classe du domaine, deux ports pilotés écrits sous forme de classes `typing.Protocol`, et le cas d'utilisation, qui joue le rôle de port pilote. Le deuxième extrait est l'extérieur : un adaptateur

PostgreSQL et une doublure de test en mémoire qui implémentent le même port. Le troisième montre pourquoi c'est important : un test unitaire rapide, et la racine de composition où les adaptateurs de production sont branchés.

```
from dataclasses import dataclass
from decimal import Decimal
from typing import Protocol

@dataclass
class Order:
    id: str
    total: Decimal
    status: str = "pending-payment"

class OrderRepository(Protocol):        # driven port
    def save(self, order: Order) → None: ...

class PaymentGateway(Protocol):         # driven port
    def authorise(self, order_id: str, amount: Decimal,
                  idempotency_key: str) → bool: ...

class PlaceOrder:                       # driving port + use case
    def __init__(self, orders: OrderRepository, payments: PaymentGateway):
        self.orders, self.payments = orders, payments

    def __call__(self, order_id: str, total: Decimal) → Order:
        if total ⩽ 0:
            raise ValueError("empty basket")
        order = Order(order_id, total)
        self.orders.save(order)                 # save first: a crash cannot lose a
charged order
        approved = self.payments.authorise(order_id, total, idempotency_key=order_id)
        order.status = "placed" if approved else "payment-failed"
        self.orders.save(order)
        return order
```

Dans l'hexagone : une règle du domaine, deux ports pilotés et le cas d'utilisation PlaceOrder. Rien ici n'importe de framework web, de pilote de base de données ni de SDK de paiement. La commande est enregistrée avant l'autorisation de la carte, et l'ID de commande sert aussi de clé d'idempotence : une panne ou une nouvelle tentative ne peut donc ni débiter un client sans commande, ni le débiter deux fois.

```
class PostgresOrderRepository:          # real driven adapter
    def __init__(self, conn):
        self.conn = conn
    def save(self, order):
        with self.conn.cursor() as cur:
            cur.execute(
                "INSERT INTO orders (id, total, status) VALUES (%s, %s, %s) "
                "ON CONFLICT (id) DO UPDATE SET status = EXCLUDED.status",
                (order.id, order.total, order.status))
        self.conn.commit()

class InMemoryOrderRepository:          # test double, same port
    def __init__(self):
        self.saved = {}
    def save(self, order):
        self.saved[order.id] = order
```

À l'extérieur de l'hexagone : un véritable adaptateur et une doublure de test pour le même port. Le cas d'utilisation ne peut pas les distinguer.

```
class DecliningPayments:
    def authorise(self, order_id, amount, idempotency_key): return False

def test_failed_payment_is_recorded():
    orders = InMemoryOrderRepository()
    place = PlaceOrder(orders, DecliningPayments())
    place("A-17", Decimal("23.50"))
    assert orders.saved["A-17"].status == "payment-failed"

# composition root (main.py): the only place that knows every adapter
# place_order = PlaceOrder(PostgresOrderRepository(conn), CardProviderGateway(key))
# app.add_route("/orders", make_rest_controller(place_order))   # driving adapter
```

Un test unitaire pilote directement le cas d'utilisation, sans serveur web, sans base de données ni réseau ; le câblage de production se fait à un seul endroit, au démarrage.

Remarquez ce que le cas d'utilisation ne contient pas : pas de SQL, pas de codes de statut HTTP, pas de types d'exception propres à un prestataire, pas de décorateurs de framework. Le test s'exécute en quelques millisecondes et couvre un cas (un paiement refusé) qu'il serait malcommode de mettre en place avec la sandbox d'un vrai prestataire. L'adaptateur PostgreSQL a besoin de ses propres tests, mais ce sont des tests d'intégration ciblés d'une seule classe face à une vraie base de données, pas des tests de règles métier. C'est le bénéfice de testabilité décrit à la section 2.6 : la contrôlabilité et l'observabilité grâce aux dépendances injectées.

EXEMPLE COMMENTÉ

Exemple commenté : changer de prestataire de paiement

La plateforme de livraison de repas doit passer d'un prestataire de paiement par carte à un autre en six semaines, à cause d'un changement de tarifs. Comparez le travail dans deux versions du système : l'une où le code de l'étape de paiement appelle directement le SDK de l'ancien prestataire à 23 endroits, et l'autre construite avec un port piloté `PaymentGateway` et un adaptateur.

Appels directs : mesurer l'impact	Une recherche trouve le SDK dans l'étape de paiement, les remboursements, les abonnements et trois écrans d'administration. Les types d'exception propres au SDK sont capturés à 11 endroits, et son objet `Charge` est stocké dans une colonne de base de données.
Appels directs : la modification	Chaque endroit d'appel change, plus une migration de données pour les objets stockés. Chaque test qui touchait aux paiements a été écrit pour la sandbox de l'ancien SDK et doit être réécrit. Estimation : 5 à 6 semaines, avec une bascule risquée d'un seul coup.
Ports et adaptateurs : mesurer l'impact	Une seule classe d'adaptateur (environ 200 lignes) implémente `PaymentGateway`. Le noyau ne connaît que `charge`, `refund` et son propre type `PaymentResult`.
Ports et adaptateurs : la modification	Écrire un second adaptateur pour le nouveau prestataire ; exécuter contre lui les tests de contrat existants du port ; choisir l'adaptateur commande par commande avec un feature flag, de sorte que 1 %, puis 10 %, puis 100 % des paiements passent par le nouveau. Estimation : 2 semaines, réversible à chaque étape.
Ce que le port a coûté	Concevoir le port et l'adaptateur au départ a pris environ 3 jours de plus, et chaque nouvelle fonctionnalité de paiement exige une petite modification du port en plus de l'adaptateur.

La valeur d'un port est proportionnelle à la probabilité et au coût du changement qu'il isole. Les prestataires de paiement, les courtiers de messages et les API tierces changent souvent ; votre propre moteur de base de données principal ne changera peut-être jamais, et l'envelopper dans des ports élaborés est plus difficile à justifier.

Le port doit parler la langue du domaine (`charge`, `PaymentResult`), et non celle de l'ancien prestataire. Un port qui calque l'API d'un fournisseur ne protège de rien.

Caractéristiques de la famille ports et adaptateurs

Caractéristique	Évaluation	Justification
Testabilité	Élevé	Les règles métier peuvent être testées en mémoire, rapidement et de façon déterministe, en branchant des doublures de test sur les ports pilotés.
Modifiabilité de l'infrastructure	Élevé	Remplacer une base de données, un prestataire de paiement ou un framework web revient à écrire un nouvel adaptateur ; le noyau ne change pas.
Modifiabilité des règles métier	Moyen à élevé	Les règles vivent à un seul endroit, libres de tout code de framework, mais une fonctionnalité traverse tout de même le port, le cas d'utilisation et l'adaptateur.
Simplicité	Bas à moyen	Plus de types, d'interfaces et de code de mapping que dans une application en couches classique ; plus difficile au début pour les nouveaux venus.
Vitesse de développement (au début)	Moyen	La structure supplémentaire ralentit les premières fonctionnalités d'une simple application CRUD ; elle est rentabilisée à mesure que le domaine grandit.
Coût global	Faible à moyen	Aucune infrastructure supplémentaire ; le coût est le temps que les développeurs passent à écrire et à maintenir les ports, les adaptateurs et le code de mapping.
Performances	Moyen à élevé	Les appels d'interface ne coûtent presque rien ; les vrais surcoûts sont le mapping d'objets entre les couches et la perte de certaines commodités de l'ORM.
Scalabilité, déployabilité, tolérance aux pannes	Non déterminées par ce style	Elles dépendent du style de déploiement (monolithe ou services) dans lequel s'inscrit l'hexagone.

Coûts, critiques et quand l'utiliser

Les bénéfices sont réels, et les coûts aussi. Chaque port est une interface supplémentaire ; chaque frontière peut exiger du code de mapping pour convertir, par exemple, entre une entité ORM, un objet du domaine et une réponse d'API. Pour une application qui se contente pour l'essentiel de déplacer des champs de formulaire vers des tables avec peu de logique entre les deux, cette structure ajoute des fichiers sans ajouter de protection, et les conventions d'un framework (un modèle active record, une interface d'administration générée) peuvent donner un meilleur résultat plus vite. Certains praticiens expérimentés soutiennent que les versions strictes de la Clean Architecture produisent trop d'indirection et que la plupart des équipes ne

changent jamais de base de données ; d'autres répondent que le principal bénéfice n'est pas de changer de technologie, mais d'avoir des tests rapides et fiables et un code métier lisible sans connaître le framework. Les deux ont raison, dans des contextes différents.

Utilisez-la quand la logique métier est substantielle et survivra aux frameworks actuels ; quand le système s'intègre à des services externes susceptibles de changer ; quand des tests rapides et isolés des règles métier sont importants ; et comme structure interne des modules d'un monolithe modulaire ou des services construits avec la conception pilotée par le domaine (section 5.7). **Soyez prudent** avec les simples applications CRUD, les prototypes et les outils à courte durée de vie, pour lesquels la structure par défaut du framework suffit généralement. Une voie médiane pragmatique est courante : des ports pour les services externes et pour tout ce qui est susceptible de changer, et les commodités propres au framework ailleurs.

ATTENTION

Anti-patrons courants

- **Ports calqués sur la technologie.** Un port appelé `SqlQueryExecutor`, ou un port qui calque le SDK d'un fournisseur, conserve le couplage qu'il devait supprimer. Les ports doivent parler la langue du domaine.
- **Fuite du framework.** Des cas d'utilisation qui acceptent des objets de requête HTTP ou renvoient des entités ORM ont des dépendances vers l'extérieur déguisées.
- **De la logique dans les adaptateurs.** Les règles métier placées dans des contrôleurs ou des classes de repository sont invisibles pour les tests des cas d'utilisation et dupliquées d'un adaptateur à l'autre.
- **Une interface pour chaque classe.** Les ports ont leur place à la frontière avec le monde extérieur, pas entre chaque paire de classes internes.
- **Cas d'utilisation anémiques qui ne font que relayer.** Les cas d'utilisation qui se contentent de transmettre les appels à un repository sont le gouffre d'architecture de la section 5.2 sous une nouvelle forme ; acceptez-les pour les lectures simples, mais remarquez s'ils sont majoritaires.
- **Une racine de composition partout.** Des adaptateurs créés au plus profond du code métier (`PostgresOrderRepository()` dans un cas d'utilisation) ruinent la substitution sur laquelle repose le style.

Faire respecter la règle de dépendance

Comme pour les couches, la règle s'érode si une machine ne la vérifie pas. En Python, import-linter peut déclarer les anneaux sous la forme d'un contrat *layers* (adaptateurs au-dessus de l'application, elle-même au-dessus du domaine) et ajouter des contrats *forbidden* qui

empêchent les paquets du domaine et de l'application d'importer des frameworks web, des pilotes de base de données ou des SDK. En Java, ArchUnit propose une règle toute prête, `onionArchitecture()`, dans laquelle vous nommez les paquets du modèle de domaine, des services de domaine, des services applicatifs et des adaptateurs, et qui fait échouer un test si une dépendance pointe vers l'extérieur. Découper le code en modules de build séparés (un module de domaine sans aucune dépendance à un framework) rend les dépendances vers l'extérieur impossibles à compiler, ce qui est la vérification la plus forte de toutes. Ces vérifications sont des exemples des fonctions d'aptitude traitées au chapitre 11.

Une structure de paquets qui rend la règle visible aide les relecteurs : par exemple `orders/domain`, `orders/application` (cas d'utilisation et ports) et `orders/adapters/{rest,postgres,payments}`. S'organiser d'abord par domaine métier, puis par anneau, garde la structure alignée sur les frontières du domaine des sections 5.3 et 5.7.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Un contrôleur REST est-il un adaptateur pilote ou piloté ? Et un repository SQL ?**

Un contrôleur REST est un adaptateur pilote : il reçoit des requêtes et appelle l'application. Un repository SQL est un adaptateur piloté : il implémente une interface que l'application appelle.

▼ **Dans la Clean Architecture, un cas d'utilisation peut-il renvoyer une entité ORM à un contrôleur ?**

Non. L'ORM appartient à l'anneau extérieur, donc un cas d'utilisation qui le nomme a une dépendance vers l'extérieur. Il doit renvoyer une structure simple ou un objet du domaine défini dans un anneau intérieur.

▼ **Où les adaptateurs sont-ils créés et connectés à l'application ?**

Dans une racine de composition, généralement le code de démarrage, qui est le seul endroit à connaître chaque adaptateur.

▼ **Qu'est-ce que l'architecture en oignon ajoute à la représentation hexagonale ?**

Des anneaux nommés à l'intérieur de l'application (modèle de domaine, services de domaine, services applicatifs) et l'affirmation explicite que la base de données est une préoccupation externe située en bordure, et non le fondement.

▼ **À vous : Un produit de billetterie a des règles complexes d'attribution des places et un écran d'administration simple pour modifier les coordonnées des salles. Utiliseriez-vous l'architecture hexagonale pour les deux ? Expliquez ce que vous gagneriez ou perdriez dans chaque cas.**

Utilisez-la uniquement pour l'attribution des places. Ces règles y gagnent des tests rapides et isolés, et la base de données ou le courtier peuvent être remplacés en changeant d'adaptateur, ce qui convient à une logique importante et changeante. L'écran d'administration est un simple CRUD : les ports et le code de mappage ajouteraient une indirection pour bien peu à protéger. Utilisez là une approche simple, en la gardant séparée du cœur. Appliquez le pattern là où la complexité justifie son coût, et faites respecter automatiquement la règle de dépendance.

SOURCES

- Alistair Cockburn: Hexagonal architecture: https://alistair.cockburn.us/hexagonal-architecture/
- Jeffrey Palermo: The Onion Architecture, part 1 (2008): https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- Robert C. Martin: The Clean Architecture (2012): https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- ArchUnit user guide (layered and onion architecture rules): https://www.archunit.org/userguide/html/000_Index.html
- import-linter documentation: https://import-linter.readthedocs.io/en/stable/

Conception stratégique pilotée par le domaine : contextes délimités et cartes de contextes

AVANT DE LIRE

Cette section s'appuie sur :

- 3.1 Modularité
- 3.7 Loi de Conway
- 5.6 Hexagonale et Clean Architecture

Termes déjà vus qu'elle utilise : Adaptateur, Monolithe modulaire, Grosse boule de boue (big ball of mud), Loi de Conway

L'IDÉE EN BREF

Dans un hôpital, le mot « patient » ne signifie pas la même chose pour l'accueil (nom, adresse, rendez-vous), le service de soins (lit, médicaments, observations) et le service de facturation (assureur, facture). Si un seul formulaire géant essayait de servir tout le monde, il serait énorme et chacun se le disputerait. Cela fonctionne mieux quand chaque service tient son propre dossier simple et que tous s'accordent sur la façon de se transmettre les informations. La conception pilotée par le domaine applique cette idée au logiciel : elle repère les services naturels d'une entreprise, donne à chacun son propre modèle et son propre vocabulaire, et rend explicites les liens entre eux.

CE QU'IL FAUT SAVOIR

- La conception pilotée par le domaine (Eric Evans, 2003) construit le logiciel autour d'un modèle du domaine métier élaboré avec les experts du domaine et exprimé dans un langage omniprésent, utilisé aussi bien à l'oral que dans le code.
- Les sous-domaines sont cœur (là où l'entreprise se distingue de la concurrence), de soutien (spécifiques mais non différenciants) ou génériques (problèmes déjà résolus) ; ce classement détermine où investir l'effort de conception et ce qu'il faut acheter.
- Un contexte délimité est une frontière à l'intérieur de laquelle un modèle et un langage s'appliquent de façon cohérente ; un même mot peut sans risque désigner des choses différentes dans des contextes différents.
- Une carte de contextes montre les relations entre les contextes : partenariat, noyau partagé, client-fournisseur, conformiste, couche anticorruption, service hôte ouvert, langage publié et chemins séparés.

- L'event storming (Alberto Brandolini) découvre les contextes en plaçant des événements de domaine sur une ligne du temps avec les personnes qui connaissent le métier.
- Les agrégats sont des frontières de cohérence modifiées par l'intermédiaire d'une racine qui fait respecter les invariants ; les contextes délimités sont le meilleur guide pour tracer les frontières des modules et des services.

Pourquoi la conception pilotée par le domaine a sa place dans un chapitre sur la structure

Chaque style de ce chapitre a besoin de frontières. Un monolithe modulaire a besoin de frontières entre modules ; les microservices du chapitre 6 ont besoin de frontières entre services ; même une application hexagonale doit savoir ce qui a sa place à l'intérieur de l'hexagone. La conception pilotée par le domaine (DDD) est la méthode la plus utilisée pour trouver ces frontières dans le domaine métier lui-même, plutôt que dans des couches techniques.

Eric Evans l'a présentée dans *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley, 2003). Selon lui, dans la plupart des logiciels de gestion, les problèmes les plus difficiles ne sont pas techniques : ils tiennent à la compréhension du domaine, c'est-à-dire à ce qu'est vraiment un contrat d'assurance, un sinistre, une livraison ou un remboursement, et aux règles qui les régissent. Le logiciel devrait donc être construit autour d'un modèle du domaine conçu délibérément, élaboré conjointement par les développeurs et les experts du domaine, et ce modèle devrait être protégé de toute confusion avec d'autres modèles. En 2015, Evans a publié une courte *DDD Reference* qui résume les patrons et en ajoute quelques-uns, comme le partenariat et la grosse boule de boue (big ball of mud) en tant que patrons de carte de contextes, et les événements de domaine en tant que bloc de construction.

La DDD comporte deux volets. La **conception stratégique** divise un grand domaine en sous-domaines et en contextes délimités, et cartographie les relations entre eux ; c'est de l'architecture, et c'est le sujet principal ici. La **conception tactique** fournit des blocs de construction (entités, objets valeur, agrégats et autres) pour écrire le modèle à l'intérieur d'un contexte ; c'est de la conception, et elle est traitée brièvement.

Langage omniprésent (ubiquitous language)

Un langage omniprésent (ubiquitous language) est un vocabulaire partagé, construit ensemble par les développeurs et les experts du domaine, et employé de manière cohérente dans les conversations, les documents, les tests et le code. Si les répartiteurs disent qu'un livreur est

« empilé » lorsqu'il transporte deux commandes du même restaurant, alors le code dit `stack`, et non `multiOrderAssignment`. Quand un terme se révèle ambigu ou maladroit, l'équipe change le langage et le code en même temps.

Cela ressemble à une simple question de nommage, mais les conséquences sont architecturales. C'est dans la traduction entre les mots du métier et les mots du code que naissent les malentendus, et donc les défauts. Et dès que deux groupes emploient le même mot avec des sens différents, vous tenez la preuve de l'existence de deux modèles différents, ce qui est le point de départ des contextes délimités.

Les sous-domaines : cœur, de soutien et génériques

Le domaine métier dans son ensemble se divise en sous-domaines : des domaines d'activité comme les cartes, les commandes, les paiements et l'affectation des livreurs. Les praticiens les classent en trois catégories, en s'appuyant sur la distinction d'Evans entre le domaine cœur et les sous-domaines génériques ; la répartition en trois, avec les sous-domaines de soutien, a été popularisée par des auteurs ultérieurs, notamment Vaughn Vernon dans *Implementing Domain-Driven Design* (2013).

Le domaine cœur (core domain) est ce qui distingue l'entreprise de ses concurrents et la raison pour laquelle le logiciel vaut la peine d'être construit. Pour une entreprise de livraison de repas, c'est probablement l'affectation : associer les commandes aux livreurs et prédire les heures d'arrivée. Un sous-domaine de soutien (supporting subdomain) est propre à l'entreprise et nécessaire, mais n'est pas une source d'avantage : la gestion des cartes, l'intégration des livreurs. Un sous-domaine générique est un problème que partagent de nombreuses entreprises, pour lequel il existe de bonnes solutions : les paiements, l'identité, l'e-mail et les SMS.

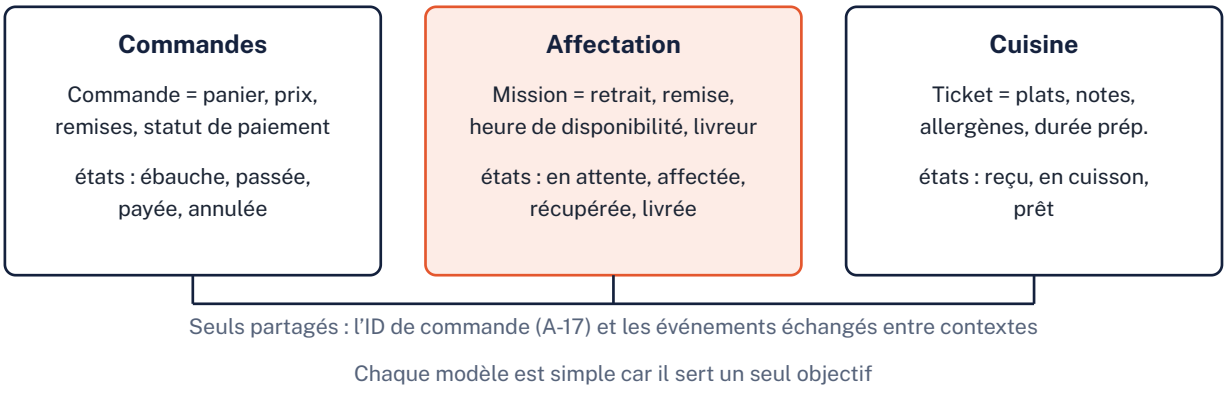
Ce classement est une décision d'investissement. Le cœur mérite les personnes les plus expérimentées, la modélisation la plus riche et du code que vous possédez entièrement, car c'est là qu'un meilleur logiciel se traduit directement en avantage pour l'entreprise. Les sous-domaines de soutien méritent des conceptions saines mais plus simples, souvent un module en couches classique. Les sous-domaines génériques devraient généralement être achetés, utilisés sous forme de service ou construits comme de fins adaptateurs au-dessus de composants open source : construire votre propre fournisseur d'identité, c'est dépenser de l'effort là où vous ne pouvez pas gagner. Le classement évolue aussi avec le temps ; une capacité de soutien peut devenir cœur quand la stratégie de l'entreprise change.

Les contextes délimités

Un contexte délimité (bounded context) est une frontière explicite à l'intérieur de laquelle un modèle, et un langage omniprésent, s'appliquent de façon cohérente. À l'intérieur, « commande » a exactement un sens. À l'extérieur, le même mot peut signifier autre chose, et c'est acceptable, car chaque modèle est conçu pour un seul objectif. Evans a observé que vouloir unifier en un seul le modèle d'un grand système n'est ni faisable ni rentable ; la tentative produit d'énormes classes partagées qui servent mal tous les objectifs et que chaque équipe doit modifier.

Sous-domaines et contextes délimités sont des choses différentes, même s'ils sont souvent alignés. Un sous-domaine fait partie du problème : il existe dans l'entreprise, qu'il y ait un logiciel ou non. Un contexte délimité fait partie de la solution : une frontière que vous choisissez pour un modèle et, généralement, pour du code et des données qui appartiennent à une seule équipe. L'idéal est un contexte délimité par sous-domaine, mais les systèmes hérités ont souvent un seul contexte qui s'étend sur plusieurs sous-domaines, et un grand sous-domaine peut avoir besoin de plusieurs contextes.

Un mot, trois modèles



« Commande » désigne un panier avec des prix pour Commandes, une mission de retrait et de remise pour Affectation, et une liste de plats à préparer dans le contexte Cuisine ; chaque modèle reste petit parce qu'il sert un seul objectif.

SI VOUS DÉBUTEZ

Pas à pas : les parties de l'entreprise et les frontières du logiciel

Le point qui fait le plus souvent trébucher ici est la différence entre un sous-domaine et un contexte délimité. Ils coïncident souvent, si bien qu'on les confond facilement, mais l'un fait partie du problème métier, et l'autre est une frontière que l'équipe choisit dans le logiciel.

- L'affectation, qui associe les commandes aux livreurs et prédit les heures d'arrivée, est un sous-domaine de l'activité de livraison de repas. Elle existe qu'il y ait un logiciel ou non : un répartiteur muni d'un téléphone et d'un plan ferait encore ce travail.
- Un contexte délimité est une décision sur le logiciel : une frontière à l'intérieur de laquelle s'appliquent un modèle et un langage. L'équipe décide qu'à l'intérieur du contexte Affectation, l'objet traité est une « mission de livraison » avec un retrait, une remise, une heure de disponibilité et un livreur.
- Idéalement, chaque sous-domaine reçoit exactement un contexte délimité, qui appartient généralement à une seule équipe, et l'affectation et son contexte coïncident de cette façon.
- La correspondance n'est pas toujours aussi nette. Dans un système plus ancien, une seule classe Order partagée peut servir à la fois les commandes, l'affectation et la vue cuisine des restaurants : un seul modèle, donc un seul contexte, étiré sur plusieurs sous-domaines, que chaque équipe doit modifier et que personne ne parvient à garder simple.
- L'inverse se produit aussi : un grand sous-domaine peut avoir besoin de plusieurs contextes délimités, chacun avec un modèle plus petit.
- Les sous-domaines peuvent rester tels quels pendant que l'équipe redessine ses contextes, comme le fait l'exemple commenté plus loin dans cette section lorsqu'il découpe cette classe Order partagée en trois modèles.
- **La leçon :** trouvez les sous-domaines en étudiant le métier ; choisissez les contextes délimités en décidant où un modèle et un langage doivent s'appliquer. Visez un contexte par sous-domaine, et repérez les endroits où les deux ne correspondent pas.

Les cartes de contextes

Une carte de contextes (context map) montre les contextes délimités d'un système et les relations entre eux. Ces relations décrivent à la fois l'intégration technique et les rapports de force et de communication entre les équipes : c'est pourquoi la cartographie des contextes concerne autant les organisations que le code. C'est une application directe de la loi de Conway : la carte montre où les canaux de communication de l'organisation et les chemins d'intégration du système doivent coïncider.

Beaucoup de relations sont de type **amont/aval** (upstream/downstream) : les décisions du contexte amont affectent le contexte aval, mais pas l'inverse. L'équipe aval doit alors choisir comment s'en accommoder, et c'est ce choix que nomment la plupart des patrons.

Patrons de relation de la carte de contextes

Patron	Ce que cela signifie	Qui s'adapte à qui	Exemple de livraison de repas
Partenariat	Deux équipes réussissent ou échouent ensemble : elles planifient donc conjointement et font évoluer leur interface ensemble.	Les deux, d'un commun accord	Commandes et Affectation mettent en production ensemble le passage de relais de la commande à la livraison.
Noyau partagé (shared kernel)	Deux contextes partagent une petite partie du modèle et de son code, explicitement convenue ; toute modification exige l'accord des deux équipes.	Les deux, avec prudence	Affectation et Intégration des livreurs partagent une toute petite bibliothèque de profil de livreur (type de véhicule, ID du livreur).
Client-fournisseur	L'équipe amont (le fournisseur) s'engage à répondre aux besoins de l'équipe aval (le client), qui font partie de sa planification.	L'amont sert l'aval	Le catalogue des cartes fournit les cartes et les prix à Commandes, et les besoins de Commandes entrent dans le backlog de Carte.
Conformiste	L'équipe aval adopte le modèle amont tel quel, parce qu'elle ne peut pas l'influencer et que la traduction n'en vaut pas le coût.	L'aval se conforme	Affectation utilise directement le modèle d'itinéraires et de localisation du fournisseur de cartographie.
Couche anticorruption	L'équipe aval construit une couche de traduction pour que le modèle amont ne puisse pas s'infiltrer dans le sien.	L'aval traduit	Carte traduit des dizaines de formats de caisse (point de vente) des restaurants dans son propre modèle de carte ; Paiements isole l'API du prestataire.
Service hôte ouvert (open host service)	Le contexte amont offre un protocole ou une API bien défini à tous les consommateurs, au lieu d'une intégration sur mesure pour chacun.	L'amont publie une fois	Paiements offre une API de paiement unique et documentée à Commandes, aux remboursements et aux abonnements.
Langage publié (published language)	Un format d'échange partagé et documenté, souvent utilisé avec un service hôte ouvert.	Les deux utilisent le langage	Les événements de commande (OrderPlaced, OrderCancelled) ont un schéma versionné et documenté.

Patron	Ce que cela signifie	Qui s'adapte à qui	Exemple de livraison de repas
Chemins séparés	Les contextes ne sont pas intégrés du tout, parce que le bénéfice est inférieur au coût.	Aucun	Le reporting de la paie des livreurs se fait dans un outil financier séparé, alimenté par un export mensuel, et non par une intégration en direct.

Deux de ces patrons méritent une attention particulière. Une couche anticorruption (anticorruption layer, ACL) est un ensemble d'adaptateurs et de traducteurs qu'un contexte aval construit pour qu'un modèle étranger, souvent un système hérité ou un fournisseur externe, ne puisse pas s'infiltrer dans son propre modèle. C'est la même idée que l'adaptateur piloté de la section 5.6, appliquée entre contextes, et c'est le choix par défaut pour s'intégrer à tout ce que vous ne contrôlez pas. Un service hôte ouvert (open host service) associé à un langage publié (published language) est la façon dont l'amont sert bien de nombreux consommateurs : un seul protocole documenté et versionné au lieu d'une intégration sur mesure pour chacun. Un noyau partagé (shared kernel) doit rester petit ; un noyau partagé trop grand refait de deux contextes un seul, que deux équipes se disputent, et la référence d'Evans de 2015 ajoute la grosse boule de boue comme étiquette honnête pour une zone sans aucune frontière claire, qu'il faut contenir plutôt qu'étendre.

L'event storming : découvrir les contextes

Les frontières se découvrent avec les experts du domaine ; on ne les conçoit pas seul. L'event storming, créé par Alberto Brandolini et décrit pour la première fois sur son blog en 2013, est le format d'atelier le plus populaire pour cela. Les personnes qui connaissent le métier et celles qui construisent le logiciel se tiennent devant un long mur de papier et écrivent des événements de domaine (des choses qui se sont produites, au passé : *Commande passée*, *Livreur affecté*, *Repas récupéré*) sur des notes adhésives orange, qu'elles placent sur une ligne du temps.

Le groupe ajoute ensuite les commandes qui provoquent les événements, les personnes et les systèmes externes impliqués, les politiques qui réagissent aux événements (« chaque fois qu'une commande est passée, trouver un livreur ») et, surtout, les points chauds (hot spots), là où les gens ne sont pas d'accord ou où personne ne sait. Les groupes d'événements qui partagent un langage et un objectif suggèrent des contextes délimités ; les endroits où le même mot est employé différemment suggèrent des frontières ; les endroits où se produit un

passage de relais suggèrent des points d'intégration. Une séance de vue d'ensemble (big picture) pour toute une entreprise dure un jour ou deux et ne demande presque aucune préparation, en dehors des bonnes personnes et de beaucoup de place au mur.

Les patrons tactiques en bref

À l'intérieur d'un contexte délimité, la DDD propose des blocs de construction pour le modèle. Une entité a une identité qui persiste pendant que ses attributs changent : une commande reste la même commande après un changement de statut. Un objet valeur (value object) n'a pas d'identité et n'est défini que par ses valeurs : deux objets `Money(12.50, "GBP")` sont interchangeables, et les objets valeur sont généralement immuables. Un événement de domaine enregistre quelque chose de significatif qui s'est produit dans le domaine.

Un agrégat est un groupe d'entités et d'objets valeur traités comme une seule unité pour les modifications, avec une entité comme racine. Le code extérieur ne peut détenir de références que vers la racine et ne doit modifier l'agrégat que par son intermédiaire, de sorte que la racine peut faire respecter les invariants de l'agrégat (des règles qui doivent toujours être vraies, comme « aucun article ne peut être ajouté après qu'une commande a été passée »). Une transaction ne devrait modifier qu'un seul agrégat ; les autres agrégats sont référencés par leur ID et mis à jour par des événements de domaine, ce qui fait des agrégats l'unité naturelle de cohérence et, plus tard, de distribution. Un repository charge et enregistre des agrégats entiers et se présente au domaine comme une collection ; son interface est un port piloté, en termes d'architecture hexagonale.

```
from dataclasses import dataclass, field
from decimal import Decimal

@dataclass(frozen=True)
class Money:                                # value object: equal if values equal
    amount: Decimal
    currency: str
    def __add__(self, other):
        if other.currency ≠ self.currency:
            raise ValueError("currency mismatch")
        return Money(self.amount + other.amount, self.currency)
    def times(self, n: int):
        return Money(self.amount * n, self.currency)

@dataclass
class OrderLine:                            # entity inside the aggregate
    dish_id: str
    qty: int
    price: Money

@dataclass
class Order:                                # aggregate root
    id: str
    restaurant_id: str                      # other aggregates referenced by ID only
    lines: list[OrderLine] = field(default_factory=list)
    status: str = "draft"
    events: list = field(default_factory=list)

    def add(self, dish_id, qty, price):
        if self.status ≠ "draft":
            raise ValueError("order already placed")    # invariant
        self.lines.append(OrderLine(dish_id, qty, price))

    def place(self, minimum: Money):
        total = Money(Decimal(0), minimum.currency)
        for line in self.lines:
            total = total + line.price.times(line.qty)
        if total.amount < minimum.amount:
            raise ValueError("below restaurant minimum")  # invariant
        self.status = "placed"
        self.events.append({"type": "OrderPlaced", "order_id": self.id, "total":
str(total.amount)})
```

Un agrégat Order : un objet valeur (Money), une entité (OrderLine), la racine de l'agrégat qui fait respecter les invariants, et un événement de domaine enregistré lorsque la commande est passée.

L'exemple garde l'agrégat petit, référence le restaurant par son ID au lieu de détenir un objet restaurant, et fait respecter ses règles dans des méthodes au lieu de les laisser aux appelants. Une erreur courante est l'inverse : des agrégats si grands que la modification de chaque

utilisateur verrouille les mêmes lignes, ou des objets du domaine qui ne sont que des sacs de getters et de setters, tandis que les règles vivent dans des classes de service (Martin Fowler appelle cela le modèle de domaine anémique).

EXEMPLE COMMENTÉ

Exemple commenté : une carte de contextes pour la plateforme de livraison de repas

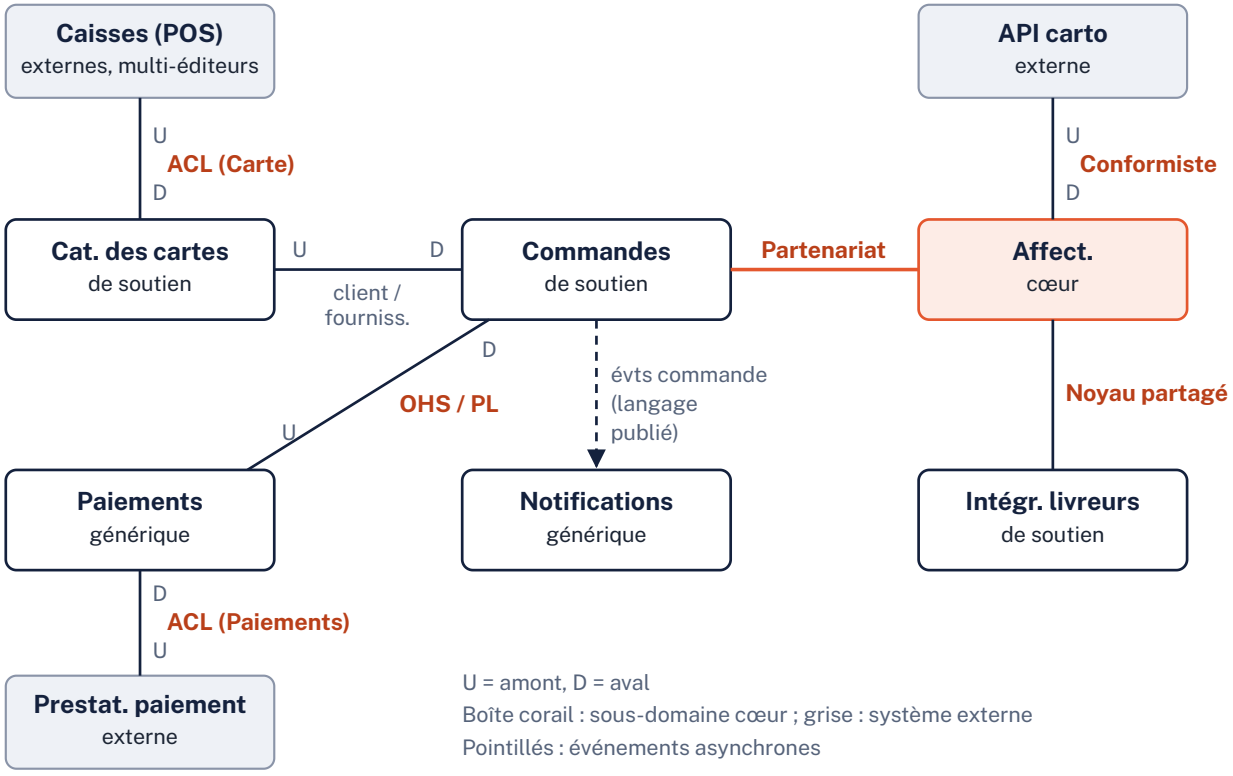
La plateforme a grandi jusqu'à environ 60 ingénieurs répartis en huit équipes, et chaque modification semble toucher la classe partagée `Order`, qui compte désormais 94 champs. L'architecte anime un atelier d'event storming de deux jours avec les chefs de produit, le personnel des opérations, le support aux restaurants et les ingénieurs, puis dessine la carte de contextes présentée ci-dessous.

1. Trouver les sous-domaines	D'après la ligne du temps des événements : gestion des cartes, commande et paiement, paiements, affectation (associer les livreurs aux commandes et prédire les heures d'arrivée), intégration des livreurs, notifications aux clients, intégration des restaurants.
2. Les classer	L'affectation est le **cœur** : des livraisons plus rapides et moins chères sont ce qui gagne les clients et les restaurants. La carte, les commandes et l'intégration des livreurs sont **de soutien** : nécessaires et propres à cette entreprise, mais ce n'est pas là qu'elle se distingue de la concurrence. Les paiements et les notifications sont **génériques** : des problèmes déjà résolus, qu'il vaut mieux acheter ou construire comme une fine couche au-dessus d'un fournisseur.
3. Dessiner les contextes délimités	Chaque sous-domaine reçoit son propre modèle. La « commande » de Commandes (panier, prix, statut de paiement) est séparée de la « mission de livraison » d'Affectation (retrait, remise, heure de disponibilité) et du « ticket » de la vue cuisine du restaurant. La classe de 94 champs devient trois modèles ciblés, reliés par l'ID de commande.
4. Nommer les relations	Commandes et Affectation : partenariat. Carte vers Commandes : client-fournisseur. Paiements : service hôte ouvert avec un langage publié, et une couche anticorruption vers le prestataire externe. Carte : couche anticorruption vers les systèmes de caisse des restaurants. Affectation : conformiste vis-à-vis de l'API de cartographie, un choix délibéré, car traduire son modèle riche coûterait plus que cela ne rapporte.
5. Décider de l'investissement	Les meilleurs ingénieurs et la modélisation la plus soignée vont à Affectation. Paiements utilise un prestataire au travers d'un fin adaptateur. Notifications utilise un service de messagerie acheté.
6. Décider du déploiement	Les contextes deviennent d'abord des modules de l'application existante, qui se transforme ainsi en monolithe modulaire (section 5.3), chacun avec son propre schéma et son interface publique. Affectation, dont les besoins de passage à l'échelle et de mise en production sont très différents, est le premier candidat à devenir plus tard un service séparé.

Classer Commandes comme sous-domaine de soutien a surpris certaines parties prenantes. Le critère n'est pas l'importance (l'entreprise ne peut pas fonctionner sans prise de commande) mais la différenciation : un meilleur modèle de commande gagnerait-il à l'entreprise plus de clients qu'un meilleur modèle d'affectation ?

La relation conformiste avec le fournisseur de cartographie est un compromis conscient, consigné dans un enregistrement de décision, et non un accident. Si le fournisseur change ou est remplacé, Affectation le ressentira.

La carte de contextes



Chaque ligne nomme le patron de relation ; U et D marquent l'amont (upstream) et l'aval (downstream), et les couches anticorruption se trouvent du côté aval, face aux systèmes que la plateforme ne contrôle pas.

Les contextes délimités comme frontières de modules et de services

Les contextes délimités sont le meilleur guide disponible pour savoir où tracer les frontières des modules dans un monolithe et celles des services dans un système distribué. Un contexte a son propre modèle, son propre langage, ses propres données et, idéalement, une seule équipe propriétaire : ce sont exactement les propriétés qui rendent un module cohésif et faiblement

couplé. Dans un monolithe modulaire, chaque contexte devient un module avec son propre schéma ; à l'intérieur, la structure hexagonale de la section 5.6 garde le modèle indépendant de l'infrastructure.

Si un contexte doit plus tard être déployé séparément, sa frontière a déjà la bonne forme pour un service, et ses relations sur la carte de contextes vous indiquent à quoi doit ressembler l'intégration. C'est pourquoi de nombreux praticiens recommandent de trouver d'abord les contextes et de décider du déploiement ensuite. L'inverse est vrai aussi : des services qui traversent les contextes délimités (un « service des commandes » dans lequel écrit chaque contexte) produisent le monolithe distribué fortement couplé étudié à la section 6.3. Notez aussi qu'un contexte délimité n'est pas automatiquement un microservice ; un contexte peut être déployé sous forme de plusieurs services, et plusieurs petits contextes peuvent partager un même déploiement.

Ce que la DDD stratégique apporte et ce qu'elle coûte

Aspect	Effet	Justification
Modifiabilité au sein d'un contexte	Élevé	Un modèle, un langage et un propriétaire : les modifications restent locales et faciles à raisonner.
Autonomie des équipes	Élevé	Les frontières des contextes s'alignent sur celles des équipes : les équipes peuvent donc modifier leurs modèles sans négocier chaque changement.
Adéquation entre le logiciel et le métier	Élevé	Le langage omniprésent et la modélisation conjointe réduisent les erreurs de traduction entre les experts et le code.
Effort initial et continu	Moyen à élevé	Les séances de modélisation, le temps des experts du domaine, les couches de traduction et la maintenance des contrats demandent tous de l'effort.
Duplication des données et intégration	Coût moyen	Chaque contexte garde sa propre copie de ce dont il a besoin, qu'il faut maintenir synchronisée au moyen d'événements ou d'API.
Intérêt pour les domaines simples	Bas	Pour les systèmes surtout CRUD, avec peu de logique, la DDD tactique en particulier ajoute du formalisme pour un faible retour.

ATTENTION

Erreurs courantes

- **La DDD tactique sans stratégie.** Les équipes adoptent partout entités, repositories et agrégats, mais ne tracent jamais de contextes délimités : elles ont le formalisme sans les frontières.
- **Un contexte par table de base de données ou par nom.** Les contextes suivent le langage et l'objectif, pas les entités de données ; un « service Client » et un « service Commande » pour chaque nom produisent des parties bavardes et couplées.
- **Modéliser sans experts du domaine.** Un langage omniprésent inventé par les seuls développeurs n'est qu'une convention de nommage.
- **Considérer la carte comme définitive.** Les contextes et les relations changent à mesure que l'entreprise et l'organisation évoluent ; revoyez la carte au moins une fois par an.
- **Sur-investir dans les sous-domaines génériques.** Les ingénieurs trouvent souvent les paiements ou l'identité plus intéressants que le cœur ; le classement existe pour résister à cette tentation.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ **Quelle est la différence entre un sous-domaine et un contexte délimité ?**

Un sous-domaine fait partie du problème métier et existe qu'il y ait un logiciel ou non ; un contexte délimité est une frontière choisie dans la solution, à l'intérieur de laquelle s'applique un modèle. Idéalement, ils sont alignés, mais ce n'est pas obligatoire.

▼ **Votre équipe s'intègre à un système de facturation hérité dont le modèle est confus et que vous ne pouvez pas modifier. Quel patron de carte de contextes protège votre modèle ?**

Une couche anticorruption : des traducteurs de votre côté qui convertissent le modèle hérité dans vos propres termes, pour qu'il ne puisse pas s'infiltrer dans votre contexte.

▼ **Pourquoi une transaction ne devrait-elle généralement modifier qu'un seul agrégat ?**

Parce que l'agrégat est l'unité dont les invariants doivent être vrais ensemble ; en modifier plusieurs à la fois crée de gros verrous et du couplage, alors que les autres agrégats peuvent être mis à jour par des événements de domaine, ce qui permet aussi de les distribuer plus tard.

▼ **Pourquoi les paiements sont-ils généralement un sous-domaine générique pour une entreprise de livraison de repas ?**

Parce qu'encaisser des paiements est un problème déjà résolu, partagé par de nombreuses entreprises, et que ce n'est pas là que l'entreprise se distingue de la concurrence : mieux vaut l'acheter ou le construire comme une fine couche au-dessus d'un prestataire que le modéliser en profondeur.

▼ **À vous : Dans une société de billetterie, « ticket » désigne une place vendue pour les Ventes et une réclamation client pour le Support, et une classe partagée Ticket a grossi jusqu'à 80 champs. Paiements vient d'un prestataire extérieur. Quels contextes et quelles relations proposeriez-vous ?**

Découpez la classe en contextes délimités, Ventes et Support, chacun avec son propre modèle et son propre langage, de sorte que « ticket » puisse sans risque y différer et que Support puisse dire « dossier ». L'attribution des places est probablement le sous-domaine cœur, Support est de soutien et l'e-mail est générique : investissez et achetez en conséquence. Protégez Ventes du prestataire de paiement externe avec une couche anticorruption. Entre Ventes et Support, une relation client-fournisseur peut convenir, en ne partageant qu'une référence de réservation. Vérifiez auprès des experts du domaine par un event storming.

SOURCES

- Eric Evans: Domain-Driven Design Reference (2015, PDF): https://www.domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf
- Martin Fowler: Bounded Context: https://martinfowler.com/bliki/BoundedContext.html
- Martin Fowler: DDD Aggregate: https://martinfowler.com/bliki/DDD_Aggregate.html
- EventStorming (Alberto Brandolini): https://www.eventstorming.com/

Quand un monolithe est la bonne réponse

AVANT DE LIRE

Cette section s'appuie sur :

- 5.1 Styles et patrons
- 5.3 Monolithe modulaire
- 5.7 Conception pilotée par le domaine

Termes déjà vus qu'elle utilise : Monolithe modulaire, Contexte délimité (bounded context), Unité de déploiement, Grosse boule de boue (big ball of mud), Équilibreur de charge (load balancer)

L'IDÉE EN BREF

Un nouveau restaurant commence généralement avec une seule cuisine, où chaque cuisinier peut voir et aider tous les autres. Ouvrir cinq cuisines séparées dans toute la ville dès le premier jour, chacune avec son personnel, ses livraisons et son responsable, coûterait bien plus cher et ferait de chaque changement de la carte une négociation. Ce n'est que lorsque l'unique cuisine est vraiment débordée, ou qu'un plat exige un équipement spécial, qu'il devient judicieux d'en ouvrir une autre. Il en va de même pour le logiciel : un programme unique et bien organisé est généralement la façon la moins chère et la plus rapide de démarrer, et le découper ne vaut la peine que lorsqu'il existe des preuves claires qu'une unité est devenue le goulot d'étranglement.

CE QU'IL FAUT SAVOIR

- Un monolithe est une décision de déploiement (une seule unité construite et publiée d'un bloc), pas un verdict sur sa qualité interne ; un monolithe bien modularisé est une architecture respectable, souvent supérieure.
- Le surcoût des microservices et MonolithFirst (2015), de Martin Fowler, soutiennent que les microservices ne sont rentables qu'au-delà d'un certain niveau de complexité, et que la plupart des systèmes de microservices réussis ont commencé comme des monolithes.
- Le monolithe majestueux de David Heinemeier Hansson (2016) plaide pour les petites équipes ; Stefan Tilkov (2015) avertit que les monolithes non modularisés sont difficiles à découper : faites donc respecter les frontières des modules dès le départ.

- Un monolithe est généralement le bon choix pour les petites équipes, les domaines incertains, les produits à leurs débuts, les forts besoins de cohérence et les capacités d'exploitation limitées.
- Les monolithes passent à l'échelle grâce à des répliques sans état derrière un équilibreur de charge, à des répliques en lecture, à la mise en cache, à des processus d'arrière-plan et, si nécessaire, au partitionnement par client (tenant).
- Découpez quand il y a des preuves : contention entre équipes, files d'attente de déploiement, besoins de passage à l'échelle ou de conformité radicalement différents ; découpez de façon incrémentale, en suivant les contextes délimités.

La question qui sous-tend le chapitre

Les styles de ce chapitre se déploient tous comme une seule unité. Le chapitre 6 pose les mêmes questions à travers un réseau, où chaque appel peut échouer, être lent ou arriver deux fois. Avant de franchir cette ligne, il vaut la peine de se demander directement : quand un monolithe est-il la bonne réponse ? Pendant une bonne partie des années 2010, la réponse par défaut du secteur était « rarement », et de nombreuses équipes ont découpé de jeunes systèmes en microservices parce que de grandes entreprises célèbres l'avaient fait. Une décennie d'expérience a produit une vision plus équilibrée, que présente cette section.

Le mot monolithe demande de la prudence. Comme l'a expliqué la section 5.1, il décrit le déploiement (une seule unité construite, testée et publiée d'un bloc), pas la qualité interne. Un monolithe peut être un système modulaire bien structuré ou une grosse boule de boue, et il en va de même d'un ensemble de services. Beaucoup des problèmes imputés aux monolithes sont en réalité des problèmes de mauvaise modularité, et découper en services un système mal modularisé produit une grosse boule de boue distribuée, ce qui est pire.

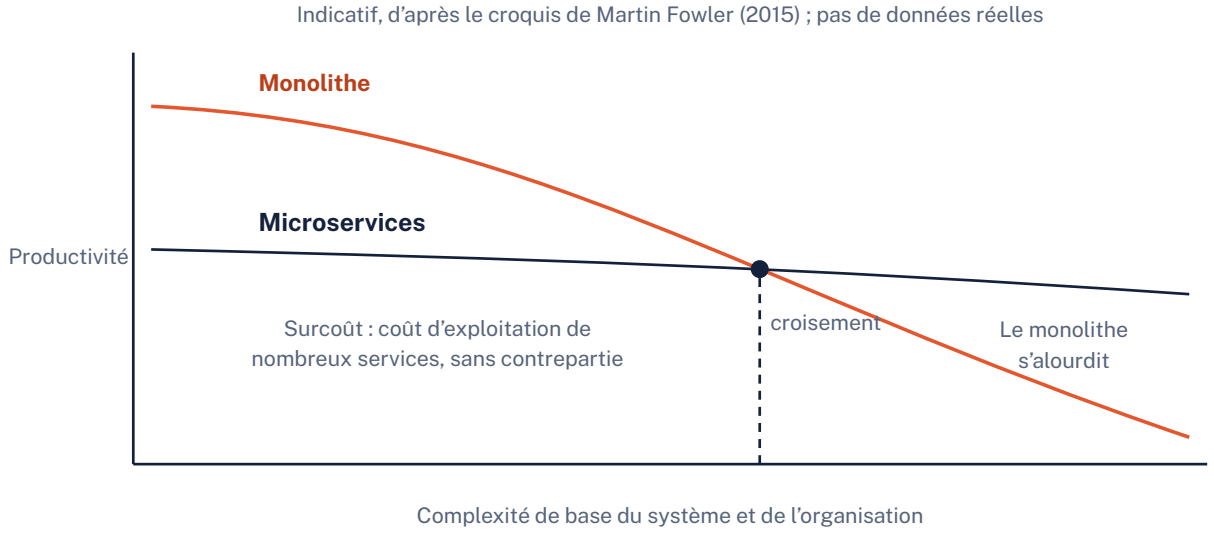
MonolithFirst et le surcoût des microservices

En mai 2015, Martin Fowler a décrit le surcoût des microservices (microservice premium) : les microservices entraînent des coûts réels qui leur sont propres (déploiement automatisé de nombreuses unités, supervision, gestion des défaillances partielles, cohérence à terme, API versionnées entre équipes), et ces coûts ralentissent une équipe, sauf si le système est assez complexe pour que les microservices les remboursent. Son conseil était de partir par défaut d'un monolithe unique et bien modularisé, et de ne passer aux microservices que lorsque la complexité l'exige.

Un mois plus tard, dans un article intitulé MonolithFirst (juin 2015), il a ajouté deux observations tirées des projets qu'il avait vus : presque tous les systèmes de microservices réussis avaient commencé comme un monolithe devenu trop gros, puis découpé, et presque

tous les systèmes construits en microservices dès le départ avaient fini en grande difficulté. L'une des raisons est qu'il est difficile de trouver de bonnes frontières entre services avant d'avoir compris le domaine, et qu'une mauvaise frontière entre services coûte bien plus cher à déplacer qu'une mauvaise frontière entre modules. Il a décrit plusieurs voies : construire un monolithe et en détacher plus tard des services par les bords ; construire un monolithe délibérément jetable ; ou commencer avec quelques services à gros grain et les découper davantage une fois les frontières stabilisées.

Le surcoût des microservices



Un croquis conceptuel : les microservices ne sont rentables qu'au-delà d'un certain niveau de complexité ; en deçà, leur coût d'exploitation est un surcoût qui ne rapporte rien.

Le monolithe majestueux, et le point de vue opposé

En février 2016, David Heinemeier Hansson, créateur de Ruby on Rails, a publié « The Majestic Monolith » sur le blog Signal v. Noise. Il y décrivait Basecamp, alors construit par une équipe d'une douzaine de programmeurs, comme une application intégrée unique servant des clients web, mobiles, de bureau et e-mail, et soutenait que les petites équipes devraient adopter un monolithe bien conçu plutôt que copier l'architecture d'organisations comptant des milliers d'ingénieurs. Sa phrase clé était «Ne distribuez pas vos traitements !», sauf si vous y êtes vraiment obligé.

Les praticiens ne sont pas tous d'accord. Également en juin 2015, Stefan Tilkov a soutenu sur le site de Fowler, dans « Don't start with a monolith », qu'il est souvent plus difficile que prévu de découper un monolithe après coup, parce que ses parties deviennent fortement couplées au travers des bibliothèques partagées, des objets du domaine et de la base de données. Si le système est manifestement assez grand pour justifier des parties séparées et que le domaine

est bien compris, il recommandait de concevoir des sous-systèmes séparés dès le départ. Les deux positions sont moins éloignées qu'il n'y paraît : toutes deux disent que la partie difficile, ce sont les frontières, et un monolithe dont les modules sont imposés dès le premier jour (section 5.3) répond à l'essentiel de la préoccupation de Tilkov.

Les cas réels vont dans les deux sens, mais partagent une leçon. Segment a décrit en 2018 comment l'entreprise avait regroupé plus de 140 petits services en un seul service, après que la charge d'exploitation et la divergence des bibliothèques partagées avaient ralenti la livraison ; Amazon Prime Video a fait passer en 2023 une charge de travail de supervision de composants serverless distribués à un processus unique (voir la section 1.3). À l'inverse, des organisations comptant des centaines d'équipes, comme les cas à grande échelle de la section 6.3, tirent des services une réelle indépendance. La leçon est l'adéquation : le style de déploiement doit correspondre à l'organisation, à la maturité du domaine et aux qualités requises.

Quand un monolithe est la bonne réponse

Une petite équipe. Avec une à quelques équipes, le coût de communication que suppriment les microservices est faible, alors que leur coût d'exploitation est le même que pour une grande organisation. Un monolithe permet à chacun de comprendre et de modifier l'ensemble du système.

Un domaine incertain ou un produit à ses débuts. Tant que vous découvrez encore ce qu'est le produit, les frontières vont bouger. Déplacer une frontière entre modules est une refactorisation ; la déplacer entre services est une migration, avec des données, des API et des déploiements.

De forts besoins de cohérence. Une seule transaction de base de données couvrant les commandes, les paiements et les stocks est simple et correcte dans un monolithe. Entre services, il faut des sagas, des outbox et de la compensation (section 6.7).

Une capacité d'exploitation limitée. Chaque service a besoin de son propre pipeline, de sa supervision, de ses alertes, de ses règles de passage à l'échelle et de connaissances pour l'astreinte. Une équipe sans plateforme pour cela passera son temps sur la tuyauterie.

Des interactions sensibles aux performances. Les appels internes au processus prennent de quelques nanosecondes à quelques microsecondes ; les appels réseau prennent des millisecondes et peuvent échouer. Des interactions bavardes entre parties du domaine coûtent bien moins cher à l'intérieur d'un seul processus.

Dans tous ces cas, la forme recommandée est un monolithe modulaire, et non un monolithe non structuré. Il garde ouverte la possibilité d'extraire des services, comme l'explique la section 5.3.

SI VOUS DÉBUTEZ

Pas à pas : déplacer une frontière tracée trop tôt

L'idée la plus difficile de cette section est de comprendre pourquoi une mauvaise frontière coûte tellement plus cher entre services qu'entre modules, l'une des principales raisons de MonolithFirst. Au début, une équipe apprend encore le domaine : certaines frontières seront donc mal placées. Suivez une donnée qui doit changer de place.

- Un jeune produit de livraison de repas comporte une partie Commandes et une partie Affectation. Au départ, l'équipe conserve dans Commandes l'heure de disponibilité de chaque commande, c'est-à-dire l'heure à laquelle la cuisine prévoit que le repas sera prêt.
- Des mois plus tard, elle découvre qu'Affectation utilise bien davantage l'heure de disponibilité, pour planifier les livreurs. Cette donnée a sa place dans Affectation, comme dans le modèle de la section 5.7.
- Dans un monolithe modulaire, la déplacer est une refactorisation : déplacez le code et la colonne vers l'autre module, mettez à jour les appelants, laissez les tests trouver ce qui est cassé, et livrez le tout en une seule version.
- Si Commandes et Affectation étaient des services séparés, le même déplacement serait une migration : modifier deux API, copier des données vivantes entre deux bases de données pendant que les commandes continuent d'arriver, mettre en production les deux services dans un ordre sûr, et maintenir l'ancien chemin en fonctionnement jusqu'à ce que chaque appelant ait basculé.
- Au début d'un produit, ce genre de déplacement se répète encore et encore, parce que l'équipe apprend encore comment fonctionne le métier. Chacun est bon marché à l'intérieur d'un seul déploiement et coûteux à travers un réseau : c'est pourquoi Fowler conseille de commencer par un monolithe.
- **La leçon :** tant que vous découvrez encore où les frontières ont leur place, gardez-les à l'intérieur d'un seul déploiement, où en déplacer une coûte peu. Imposez-les sous forme de modules dès le départ, afin que tout découpage ultérieur, fait quand il y a des preuves pour le justifier, suive des frontières qui ont déjà fait leurs preuves.

Faire passer un monolithe à l'échelle

Une croyance répandue veut que les monolithes ne puissent pas passer à l'échelle. Ils vont plus loin que ce dont la plupart des systèmes auront jamais besoin, grâce à l'axe X du cube de scalabilité (scale cube) décrit par Martin Abbott et Michael Fisher (exécuter de nombreuses copies identiques), associé à quelques tactiques classiques. Le chapitre 9 traite chacune en profondeur (section 9.1).

Des répliques horizontales derrière un équilibreur de charge. Exécutez de nombreuses copies identiques de l'application et répartissez les requêtes entre elles. Cela ne fonctionne que si l'application est sans état : les sessions, les fichiers envoyés et les caches doivent se trouver dans des stockages partagés, pour que n'importe quelle réplique puisse servir n'importe quelle requête (section 9.2). Cela apporte aussi de la redondance : la défaillance d'une réplique n'arrête plus le service.

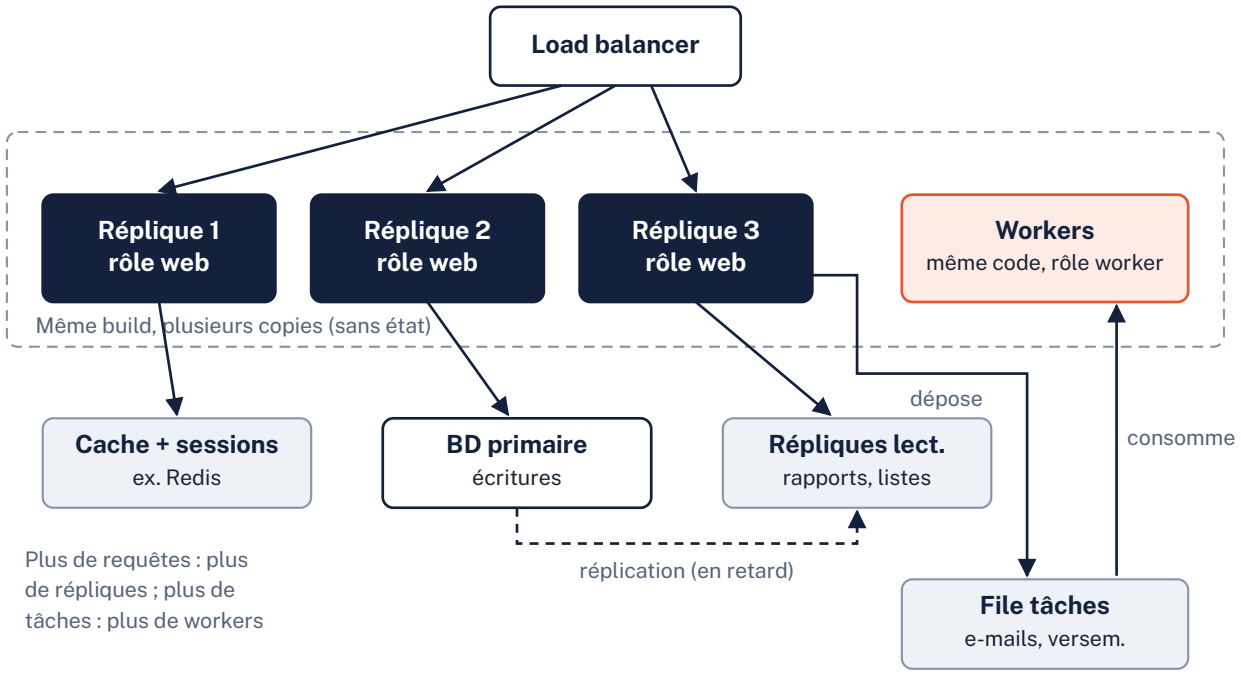
Des répliques en lecture. La plupart des systèmes lisent bien plus qu'ils n'écrivent. Les répliques en lecture de la base de données copient les modifications du primaire et servent des lectures comme l'historique des commandes et les rapports, au prix d'un retard de réplication : une réplique peut avoir une fraction de seconde de retard (section 7.4).

La mise en cache. Les cartes, les pages produits et la configuration changent rarement et sont lues en permanence ; les mettre en cache en mémoire ou dans un cache partagé supprime l'essentiel de la charge sur la base de données (section 7.6). Les ressources statiques vont sur un CDN.

Des processus d'arrière-plan. Le travail lent ou irrégulier (e-mails, factures PDF, imports, versements) est placé dans une file et traité par des processus d'arrière-plan (workers). Les workers exécutent généralement la même base de code avec une commande de démarrage différente : le système reste une seule base de code, mais les rôles web et worker passent à l'échelle séparément (section 9.3).

Des machines plus grosses et le partitionnement. La scalabilité verticale de la base de données est souvent l'étape suivante la plus simple. Pour les très grands systèmes, les données peuvent être partitionnées par client (tenant) ou par région, avec une copie complète du monolithe par partition (l'axe Z du cube de scalabilité). Stack Overflow est un exemple bien connu de la distance que peut parcourir une conception largement monolithique : selon le compte rendu de ses propres ingénieurs, un jour de février 2016, ses équilibreurs de charge ont traité environ 209 millions de requêtes HTTP, l'application principale de questions-réponses (une application multi-tenant unique pour tous ses sites de questions-réponses) tournant sur neuf serveurs web principaux, avec quatre machines SQL Server au service de tout le réseau.

Un monolithe déployé à l'échelle



Une base de code, de nombreuses copies : des répliques web sans état derrière un équilibreur de charge, un cache et des sessions partagés, une base de données primaire avec des répliques en lecture, et des workers alimentés par une file.

EXEMPLE COMMENTÉ

Exemple commenté : jusqu'où ira un seul monolithe ?

Une nouvelle start-up de livraison de repas compte 7 ingénieurs et prévoit, d'ici la fin de la deuxième année, un pic de 1200 requêtes HTTP par seconde le vendredi à l'heure du dîner. Un test de charge du monolithe modulaire montre qu'une réplique (4 vCPU) traite 180 requêtes par seconde en respectant l'objectif de 300 ms au p95. Environ 85 % des requêtes sont des lectures (cartes, suivi des commandes). Les chiffres sont fournis à titre d'illustration.

Répliques pour le pic	1200 / 180 = 6,7, donc 7 répliques au pic
Marge de capacité (headroom)	Garder le taux d'utilisation bien en dessous de la saturation (section 2.4) et survivre à la défaillance d'une zone de disponibilité sur trois : prévoir environ 50 % de marge (avec seulement deux zones, ce serait environ 100 %), donc 7 x 1,5 = environ 11 répliques au pic, avec réduction en dehors des pics
Écritures en base de données	15 % de 1200 = 180 requêtes par seconde, chacune effectuant environ 3 écritures : environ 540 écritures par seconde, largement dans les capacités d'un seul primaire PostgreSQL managé sur une instance de taille moyenne
Lectures en base de données	1020 requêtes de lecture par seconde ; un cache pour les cartes (qui changent rarement) en absorbe l'essentiel, et deux répliques en lecture prennent en charge l'historique des commandes et les rapports, en acceptant environ une seconde de retard de réplication
Travail en arrière-plan	Les e-mails de reçu, les versements aux livreurs et les rapports pour les restaurants passent à un pool de workers alimenté par une file, pour qu'un fournisseur d'e-mail lent ne puisse pas ralentir l'étape de paiement
Conclusion	Une seule base de code, déployée comme une seule unité et exécutée sous forme d'environ 11 répliques web plus des workers, atteint l'objectif à deux ans. Le budget d'architecture va aux frontières des modules et à l'absence d'état, pas au découpage.

La limite atteinte en premier est rarement le CPU. Le plus souvent, c'est la coordination des équipes à mesure que le groupe d'ingénierie grandit, ou un module unique (comme l'affectation) dont les besoins de passage à l'échelle ou de mise en production sont très différents.

Chaque chiffre ici devrait être remplacé par des mesures issues de vos propres tests de charge ; la planification de la capacité est traitée à la section 9.7.

Les signes qu'il est temps de découper

Le découpage doit répondre à des preuves, pas à la mode. Les signes qu'une unité de déploiement est devenue la contrainte sont surtout organisationnels et opérationnels :

La contention entre équipes. Plusieurs équipes modifient le même code et s'attendent les unes les autres ; les conflits de fusion, les réunions de coordination et les builds cassés par d'autres équipes se multiplient à mesure que l'organisation grandit. Si les modules sont déjà bien délimités et que la contention persiste, c'est peut-être l'indépendance de déploiement qui manque.

Les files d'attente de déploiement. Les mises en production sont regroupées en trains parce que chaque modification exige de retester tout le système ; un petit correctif attend des jours derrière des travaux sans rapport ; la modification ratée d'une équipe bloque la mise en production de tout le monde.

Des besoins de passage à l'échelle ou de disponibilité radicalement différents. Un module (par exemple, le suivi des livreurs en temps réel) a besoin de cinquante fois la capacité du reste au moment du pic, ou doit rester disponible quand tout le reste est en panne. Faire passer tout le monolithe à l'échelle pour les besoins d'un seul module devient coûteux.

Un périmètre de sécurité ou de conformité différent. Les données de carte bancaire ou les dossiers médicaux peuvent justifier une unité séparée et étroitement contrôlée, afin de garder le reste du système hors du périmètre d'audit.

Des besoins technologiques différents. Par exemple, un modèle d'apprentissage automatique qu'il vaut mieux servir depuis Python au sein d'un monolithe Java.

Certaines raisons ne sont pas de bonnes raisons à elles seules : « les monolithes, c'est du legacy », l'envie d'utiliser une nouvelle technologie, ou un code lent (qui est un problème de conception que le découpage ne résoudra pas et risque d'aggraver en ajoutant des appels réseau).

Comment découper le moment venu

Découpez de façon incrémentale, un module à la fois, en commençant par celui qui a à la fois la frontière la plus claire et la meilleure raison de partir. Modularisez d'abord si le code n'est pas encore modulaire : extraire un service d'un enchevêtrement entraîne l'enchevêtrement avec lui. Ensuite, extrayez en suivant les contextes délimités (section 5.7) : placez une API réseau devant l'API existante du module, déplacez ses données dans sa propre base de données, remplacez les événements internes au processus par un courtier de messages (message broker) et

remplacez toute transaction partagée par des sagas explicites. Dirigez progressivement le trafic vers le nouveau service et conservez l'ancien chemin jusqu'à ce que le nouveau ait fait ses preuves.

C'est l'approche strangler fig (figuier étrangleur) et ses variantes, traitées à la section 11.6. La destination n'est souvent pas « tout en microservices », mais un monolithe avec quelques services extraits là où les preuves l'exigeaient ; l'architecture à base de services de la section 6.2 est un état final courant et pragmatique.

Les styles du chapitre 5 comparés

Caractéristique	En couches	Monolithe modulaire	Tubes et filtres	Micronoyau	Hexagonale / Clean
Partitionnement principal	Technique	Domaine	Étapes de traitement	Noyau plus plug-ins	Intérieur contre extérieur
Simplicité	Élevé	Moyen à élevé	Élevé	Moyen	Bas à moyen
Coût global	Bon marché	Bon marché	Bon marché	Bon marché	Faible à moyen
Modifiabilité pour les changements métier	Bas à moyen	Élevé	Élevée au sein du flux	Élevée dans les directions prévues	Élevée pour l'infrastructure
Testabilité	Moyen	Moyen à élevé	Élevé	Moyen à élevé	Élevé
Déployabilité	Bas	Bas à moyen	Bas à moyen	Moyen	Dépend du style de déploiement
Scalabilité	Bas	Bas à moyen	Bas à moyen	Bas	Dépend du style de déploiement
Tolérance aux pannes	Bas	Bas	Bas à moyen	Bas à moyen	Dépend du style de déploiement
Usage typique	Applications petites à moyennes, une équipe	Produits en croissance, plusieurs équipes	ETL, médias, journaux, compilateurs	Produits à nombreuses variantes	Logique qui doit survivre aux frameworks

Les évaluations résument les jugements portés dans chaque section et supposent des implémentations typiques. La conception pilotée par le domaine n'apparaît pas comme colonne, car c'est une méthode pour trouver des frontières, que n'importe lequel de ces styles peut utiliser. Les systèmes réels combinent les styles : un monolithe modulaire dont les modules sont hexagonaux à l'intérieur, avec une tâche d'import en tubes et filtres et un micronoyau pour les règles fiscales, est tout à fait normal.

GUIDE DE DÉCISION

Guide de décision : choisir parmi les styles du chapitre 5

Utilisez-le pour un nouveau système ou une restructuration majeure qui fonctionnera comme une seule unité de déploiement. Parcourez les questions de haut en bas ; plusieurs styles peuvent s'appliquer à différents niveaux.

Q1 **Le système est-il principalement un flux de données passant par une série de transformations (importer, nettoyer, enrichir, exporter), plutôt que des utilisateurs qui interagissent avec un état partagé ?**

Oui Utilisez des tubes et filtres pour ce flux (section 5.4). Gardez chaque filtre sans état et le format des données entre filtres explicite.

Non Passez à la question suivante.

Q2 **Le produit a-t-il un flux de travail central stable, mais de nombreuses variantes de certaines étapes (par pays, client, fournisseur ou format de fichier), ou des tiers doivent-ils pouvoir l'étendre ?**

Oui Utilisez un micronoyau (section 5.5) pour ces points de variation, idéalement après avoir construit deux ou trois variantes, pour que le contrat repose sur des cas réels.

Non Passez à la question suivante.

Q3 **Est-il petit (une équipe, quelques dizaines d'écrans), avec des règles simples, et peu susceptible de beaucoup grandir ?**

Oui Utilisez une architecture en couches (section 5.2) avec des couches fermées imposées par un test d'architecture. N'ajoutez pas plus de structure que le problème n'en demande.

Non Passez à la question suivante.

Q4 **Plusieurs équipes y travailleront-elles, ou le domaine couvre-t-il plusieurs domaines ayant chacun leur propre langage (cartes, commandes, affectation) ?**

Oui Utilisez un monolithe modulaire (section 5.3) avec des modules tracés selon les contextes délimités (section 5.7), chacun propriétaire de ses données et de son API.

Non Une structure en couches ou modulaire convient l'une comme l'autre ; préférez des modules par domaine si une croissance est probable, car restructurer plus tard coûte cher.

Q5 **À l'intérieur de chaque module ou application, la logique métier est-elle substantielle, ou dépend-elle de services externes susceptibles de changer ?**

Oui Structurez l'intérieur avec des ports et des adaptateurs (section 5.6) : le domaine au centre, des adaptateurs pour le web, la base de données et les fournisseurs.

Non Gardez la structure par défaut du framework à l'intérieur du module ; n'ajoutez des ports qu'aux frontières susceptibles de changer.

Liste de contrôle : un monolithe qui reste sain et découpable

La plupart de ces points coûtent peu s'ils sont faits tôt, et très cher s'ils sont ajoutés après coup.

- [ ] **Les modules suivent les domaines métier**
  Les packages de premier niveau portent le nom des contextes délimités (orders, dispatch, payments), pas celui des couches techniques.
- [ ] **Chaque module a une petite API publique**
  Tout le reste est interne ; un test d'architecture fait échouer le build en cas d'import interdit.
- [ ] **Chaque table a un seul module propriétaire**
  Des schémas séparés ou des conventions de nommage rendent la propriété visible ; aucun module ne lit les tables d'un autre.
- [ ] **L'application est sans état**
  Les sessions, les fichiers envoyés et les caches se trouvent dans des stockages partagés : n'importe quelle réplique peut servir n'importe quelle requête, et l'on peut ajouter des répliques à volonté.
- [ ] **Le travail lent s'exécute dans des processus d'arrière-plan**
  Les e-mails, rapports, versements et imports passent par une file : les répliques web restent rapides, et les deux peuvent passer à l'échelle séparément.
- [ ] **Le build et les tests restent rapides**
  Suivez les durées de build et de test ; gardez le pipeline principal sous 10 à 15 minutes environ en testant les modules séparément et en parallèle.
- [ ] **Le déploiement est automatisé et fréquent**
  Des mises en production petites et fréquentes avec retour arrière (rollback) automatisé ; les feature flags découplent le déploiement de la mise en production.
- [ ] **Observabilité par module**
  Les journaux, métriques et traces sont étiquetés avec le module, pour que vous puissiez voir quel module est lent, défaillant ou coûteux.
- [ ] **Les signes de tension sont mesurés**
  Le délai de mise en œuvre des changements, la longueur de la file de déploiement, les conflits de fusion par équipe et la consommation de ressources par module sont examinés régulièrement, pour que la décision de découper repose sur des preuves.

ATTENTION

Erreurs courantes

- **Traiter « monolithe » comme une structure.** C'est une décision de déploiement. La qualité interne est une décision distincte, et c'est elle qui détermine si vous pourrez modifier ou découper le système plus tard.
- **Découper pour corriger la qualité du code.** Des services tracés autour d'une base de code enchevêtrée deviennent un monolithe distribué : tout le couplage, plus le réseau.
- **Faire passer à l'échelle ce qu'il ne faut pas.** Ajouter des répliques alors que les sessions sont stockées en mémoire, ou que la base de données est le goulot d'étranglement, ne rapporte rien.
- **Ne jamais mesurer la tension.** Sans données sur le délai de mise en œuvre, les déploiements et la contention, la décision de découper devient une affaire d'opinion et de mode.

VÉRIFIEZ VOTRE COMPRÉHENSION

▼ Pourquoi une mauvaise frontière entre services coûte-t-elle plus cher qu'une mauvaise frontière entre modules ?

Déplacer une frontière de module est une refactorisation au sein d'une seule base de code et d'une seule transaction de base de données ; déplacer une frontière de service modifie des API réseau, des stockages de données, des déploiements et souvent la répartition des responsabilités entre équipes, et doit se faire sans interruption de service.

▼ Que doit-il être vrai d'un monolithe avant que vous puissiez le faire passer à l'échelle en ajoutant des répliques ?

Il doit être sans état : les sessions, les fichiers et les caches doivent se trouver dans des stockages partagés, pour que n'importe quelle réplique puisse servir n'importe quelle requête.

▼ Citez deux signes qu'un monolithe est devenu la contrainte, et une raison de découper qui ne suffit pas à elle seule.

Signes : une contention persistante entre équipes et des files d'attente de déploiement, ou un module dont les besoins de passage à l'échelle ou de conformité sont très différents. Insuffisant à lui seul : la mode, l'envie d'une nouvelle technologie, ou un code lent.

▼ **En quoi les positions MonolithFirst et « Don’t start with a monolith » se rejoignent-elles ?**

Toutes deux disent que le problème difficile est de trouver et de conserver de bonnes frontières ; un monolithe dont les frontières de modules sont imposées répond à la préoccupation de Tilkov tout en conservant le coût plus faible défendu par Fowler.

▼ **À vous : Six ingénieurs attendent un pic de 900 requêtes par seconde, dont 80 % de lectures. Une réplique de leur monolithe modulaire en traite 150 dans l’objectif. Combien de répliques faut-il avec 30 % de marge de capacité (headroom) ? Ils veulent des microservices : que leur conseillez-vous ?**

900 / 150 = 6 répliques à la limite. Fonctionner à 70 % revient à 900 / (150 x 0,7) = 8,6, donc 9 répliques. Cela s’exécute facilement derrière un équilibreur de charge et, avec 80 % de lectures, le cache et les répliques de lecture protégeront la base de données. Rien n’indique de contention entre équipes, de files de déploiement ni de besoins de mise à l’échelle très différents, et six ingénieurs ne peuvent pas assumer le coût d’exploitation. Restez sur le monolithe, et découpez plus tard selon les contextes délimités si des preuves apparaissent.

SOURCES

- Martin Fowler: MonolithFirst (2015): https://martinfowler.com/bliki/MonolithFirst.html
- Martin Fowler: Microservice Premium (2015): https://martinfowler.com/bliki/MicroservicePremium.html
- David Heinemeier Hansson: The Majestic Monolith (2016): https://signalvnoise.com/svn3/the-majestic-monolith/
- Stefan Tilkov: Don't start with a monolith (2015): https://martinfowler.com/articles/dont-start-monolith.html
- Nick Craver: Stack Overflow, the architecture (2016 edition): https://nickcraver.com/blog/2016/02/17/stack-overflow-the-architecture-2016-edition/

Lectures complémentaires

- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. Un chapitre par style, dont les styles en couches, monolithe modulaire, pipeline et micronoyau, chacun avec une évaluation des caractéristiques ; le modèle des comparaisons de ce chapitre.
- *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal (1996), John Wiley and Sons. Le catalogue classique des patrons architecturaux, dont Couches (Layers), Tubes et filtres (Pipes and Filters) et Micronoyau (Microkernel), chacun avec son contexte, ses forces et ses conséquences.
- *An Introduction to Software Architecture*, David Garlan and Mary Shaw (1994), Carnegie Mellon University, Software Engineering Institute (CMU/SEI-94-TR-021). Le court article qui a défini les styles architecturaux comme des composants, des connecteurs et des contraintes, avec les tubes et filtres et les systèmes en couches parmi ses exemples.
  https://www.sei.cmu.edu/library/an-introduction-to-software-architecture/
- *Software Architecture: The Hard Parts*, Neal Ford, Mark Richards, Pramod Sadalage and Zhamak Dehghani (2021), O'Reilly Media. Développe le quantum d'architecture et l'analyse du couplage et de la propriété des données qui détermine où un système peut être découpé.
- *Monolith to Microservices*, Sam Newman (2019), O'Reilly Media. Un guide équilibré sur le moment, l'opportunité et la manière de décomposer un monolithe, avec un solide plaidoyer en faveur du monolithe modulaire.
- *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Eric Evans (2003), Addison-Wesley. L'origine des contextes délimités, des cartes de contextes et du langage omniprésent, utilisés pour trouver les frontières des modules et des services.
- *Hexagonal Architecture*, Alistair Cockburn (2005), alistair.cockburn.us. La description originale des ports et adaptateurs, courte et toujours l'énoncé le plus clair de l'idée.
  https://alistair.cockburn.us/hexagonal-architecture/
- *Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity*, Kirsten Westeinde (2019), Shopify Engineering. Un récit de première main de la transformation d'une très grande application Rails en monolithe modulaire plutôt qu'en microservices.
  https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity

Glossaire

Adaptateur
Code qui fait la traduction entre un port et une technologie précise, comme un contrôleur HTTP, un repository SQL ou le SDK d'un prestataire de paiement. Un même port peut avoir plusieurs adaptateurs.

Adaptateur pilote (primaire)
Adaptateur qui reçoit des requêtes venant de l'extérieur, comme un contrôleur web, un banc de test ou une tâche planifiée, et qui appelle l'application par l'intermédiaire d'un port pilote.

Adaptateur piloté (secondaire)
Adaptateur qui implémente une interface que l'application appelle lorsqu'elle a besoin de quelque chose de l'extérieur, comme stocker des données, débiter une carte ou envoyer un message.

Agrégat
Ensemble d'objets du domaine modifiés comme une seule unité par l'intermédiaire d'une unique entité racine, qui fait respecter les règles devant toujours être vraies pour l'ensemble.

Anti-patron (anti-pattern)
Solution couramment utilisée qui semble raisonnable mais qui cause invariablement plus de problèmes qu'elle n'en résout.

API du module
La petite partie, délibérément publique, d'un module que les autres modules peuvent utiliser ; tout le reste est interne et peut changer librement.

Application sans état (stateless)
Application qui ne conserve aucune donnée propre à un utilisateur dans sa mémoire entre deux requêtes, de sorte que n'importe quelle copie peut traiter n'importe quelle requête et que l'on peut ajouter ou retirer des copies librement.

Architecture de référence
Architecture modèle pour un domaine ou une plateforme, qui répartit les fonctions habituelles sur un ensemble standard de parties, et que les projets adaptent au lieu de tout concevoir de zéro.

Architecture distribuée
Architecture composée de plusieurs unités déployées séparément qui coopèrent en s'envoyant des messages par un réseau.

Architecture en couches
Style qui dispose les composants en couches horizontales, chacune avec un seul rôle technique, où chaque couche n'utilise que les couches situées en dessous.

Architecture micronoyau (microkernel)
Style architectural, aussi appelé architecture à plug-ins, dans lequel un petit système central fournit le flux de travail général et des plug-ins indépendants ajoutent ou modifient des comportements au moyen de contrats publiés.

Architecture monolithique
Architecture dans laquelle toute l'application est construite et déployée comme une seule unité, quelle que soit sa structure interne.

Back pressure (contre-pression)
Moyen pour un consommateur lent d'amener un producteur rapide à ralentir, au lieu de laisser les tampons et les files grossir sans limite.

Caractéristique d'architecture
Terme de Richards et Ford pour un attribut de qualité que l'architecture doit prendre en charge, comme la scalabilité, la déployabilité ou la tolérance aux pannes.

Carte de contextes (context map)
Représentation des contextes délimités d'un système et des relations entre eux, comme client-fournisseur ou couche anticorruption, qui couvre à la fois les relations d'intégration et les relations entre équipes.

Conception pilotée par le domaine (DDD)
Approche introduite par Eric Evans en 2003 qui construit le logiciel autour d'un modèle du domaine métier, élaboré avec les experts du domaine, et qui divise les grands domaines en modèles explicitement délimités.

Contexte délimité (bounded context)
Frontière explicite à l'intérieur de laquelle un modèle de domaine et son langage omniprésent s'appliquent de façon cohérente ; en dehors, les mêmes mots peuvent signifier autre chose.

Contrat de plug-in
L'interface publiée entre un système central et ses plug-ins : les opérations, les types de données, le comportement en cas d'erreur et la version sur lesquels les deux parties s'accordent.

Couche
Groupe de composants ayant la même responsabilité technique, qui offre des services à la couche du dessus et utilise la couche du dessous.

Couche anticorruption
Couche de traduction qu'un contexte en aval construit pour que le modèle d'un autre contexte ou système ne puisse pas s'infiltrer dans le sien.

Couche fermée
Couche que chaque requête doit traverser : la couche supérieure ne peut pas la sauter pour appeler directement une couche inférieure.

Couche ouverte
Couche que la couche du dessus est autorisée à contourner, généralement une couche de services partagés dont seules certaines requêtes ont besoin.

Couches d'isolation
Propriété selon laquelle un changement à l'intérieur d'une couche ne se propage pas aux autres couches tant que son interface reste la même ; elle suppose que les couches soient fermées.

Cube de scalabilité (scale cube)
Le modèle d'Abbott et Fisher décrivant trois façons de passer à l'échelle : l'axe X (copies identiques derrière un équilibreur de charge), l'axe Y (découpage par fonction) et l'axe Z (découpage par données, par exemple par client).

Domaine cœur (core domain)
Sous-domaine qui donne à l'entreprise son avantage concurrentiel et qui mérite donc les meilleures personnes, la modélisation la plus riche et du code détenu par l'entreprise.

Entité
Objet du domaine défini par une identité qui persiste dans le temps tandis que ses attributs changent, comme une commande ou un client.

ETL
Extract, transform, load (extraire, transformer, charger) : pipeline qui prend des données dans des systèmes sources, les nettoie et les remodèle, puis les charge dans un stockage cible comme un entrepôt de données.

Événement de domaine (domain event)
Enregistrement d'un fait significatif survenu dans le domaine métier, nommé au passé, comme OrderPlaced ou CourierAssigned.

Événement interne au processus (in-process event)
Message publié à l'intérieur d'une application en cours d'exécution afin que d'autres modules puissent réagir à un fait survenu, sans que l'émetteur sache qui écoute.

Event storming
Format d'atelier créé par Alberto Brandolini dans lequel experts du domaine et développeurs placent des événements de domaine sur une ligne du temps pour explorer un processus métier et trouver des frontières.

File de lettres mortes (dead-letter queue)
Emplacement distinct où les messages dont le traitement échoue de façon répétée sont mis de côté pour inspection, afin qu'ils ne bloquent pas le reste du flux.

Filtre
Étape de traitement autonome qui lit une entrée, puis la produit, la transforme, la teste ou la consomme, sans savoir quelles étapes la précèdent ou la suivent.

Gouffre d'architecture (architecture sinkhole)
Anti-patron des systèmes en couches dans lequel les requêtes traversent des couches qui ne font que les transmettre, ajoutant du code et du délai sans ajouter de comportement.

Grosse boule de boue (big ball of mud)
Système sans structure reconnaissable, dans lequel presque tout dépend de tout le reste ; le nom vient de Brian Foote et Joseph Yoder (1997).

Idempotent
Qualifie une opération qui a le même effet qu'elle soit appliquée une ou plusieurs fois, et qui peut donc être retentée sans risque.

Langage omniprésent (ubiquitous language)
Vocabulaire partagé par les développeurs et les experts du domaine, et employé de manière cohérente dans les conversations, les documents, les tests et le code au sein d'un même contexte délimité.

Langage publié (published language)
Format documenté et partagé pour échanger des informations entre contextes, comme un schéma d'événements versionné, souvent proposé au travers d'un service hôte ouvert.

Monolithe distribué
Système déployé sous forme de services distincts mais si fortement couplés qu'ils doivent être modifiés et publiés ensemble, ce qui cumule les coûts de la distribution sans ses avantages.

Monolithe modulaire
Application déployable unique dont le code est divisé en modules selon des frontières métier, chacun avec une interface publique explicite et ses propres données.

MonolithFirst (le monolithe d'abord)
Le conseil de Martin Fowler, en 2015, de commencer un nouveau système sous la forme d'un monolithe bien modularisé et d'en extraire des services plus tard, quand les frontières sont comprises et nécessaires.

Niveau (tier)
Frontière physique à l'exécution, comme un processus ou un serveur distinct. Une couche est une frontière logique dans le code ; plusieurs couches peuvent s'exécuter dans un même niveau (tier).

Noyau partagé (shared kernel)
Petite partie d'un modèle, et de son code, que deux contextes conviennent explicitement de partager et qui ne peut changer qu'avec l'accord des deux équipes.

Objet valeur (value object)
Objet du domaine défini uniquement par ses valeurs, sans identité propre, et généralement immuable, comme un montant d'argent ou une adresse.

Partitionnement par domaine
Division de la structure de premier niveau d'un système par capacité métier ou par sous-domaine, comme les commandes, les paiements et la livraison.

Partitionnement technique
Diviser la structure de premier niveau d'un système selon le rôle technique, comme l'interface utilisateur, la logique métier et l'accès aux données.

Patron architectural
Solution documentée et réutilisable à un problème structurel récurrent dans un contexte donné, qui nomme les parties, leurs responsabilités et leurs relations. Souvent employé comme synonyme de style.

Patron de conception (design pattern)
Solution réutilisable à un problème récurrent à l'échelle de quelques classes ou objets, comme Strategy ou Observer ; de portée plus restreinte qu'un patron architectural.

Plug-in
Module construit séparément qui ajoute un comportement à une application hôte au moyen d'un contrat publié par l'hôte, sans que le code de l'hôte soit modifié.

Point d'entrée (packaging Python)
Métadonnées d'un paquet Python installé qui annoncent un objet, comme une classe de plug-in, sous un groupe nommé afin qu'un autre code puisse le découvrir à l'exécution.

Point d'extension
Emplacement nommé qu'une application hôte ou un plug-in déclare pour que d'autres plug-ins puissent y apporter du comportement, comme dans la plateforme Eclipse.

Ports et adaptateurs (architecture hexagonale)
Le style d'Alistair Cockburn dans lequel une application ne communique avec le monde extérieur qu'au travers de ports (des interfaces qui lui appartiennent), et où des adaptateurs font la traduction entre chaque port et une technologie particulière.

Processus d'arrière-plan (background worker)
Processus qui prend dans une file des tâches lentes ou arrivant par rafales, comme des e-mails ou des rapports, et les traite en dehors de la requête qui les a créées.

Quantum d'architecture (architecture quantum)
Partie d'un système déployable indépendamment, à forte cohésion fonctionnelle, qui comprend tout ce dont elle a besoin pour fonctionner, comme sa base de données ; le périmètre au sein duquel s'applique un même ensemble de caractéristiques d'architecture.

Racine de composition
Endroit unique, généralement le code de démarrage du programme, où les adaptateurs concrets sont créés et branchés sur l'application.

Registre de plug-ins
La partie d'un système central qui enregistre les plug-ins disponibles, leurs noms, leurs versions et leurs contrats, afin que le système central puisse les trouver sans les nommer dans le code.

Règle de dépendance
Règle de Robert C. Martin pour la clean architecture : les dépendances du code source pointent uniquement vers l'intérieur, de sorte que les anneaux intérieurs ne nomment jamais rien de ce qui est déclaré dans les anneaux extérieurs.

Réplique en lecture (read replica)
Copie d'une base de données qui reçoit les modifications de la base primaire et sert les requêtes de lecture, ce qui réduit la charge sur la base primaire au prix de données légèrement périmées.

Repository (entrepôt)
Objet qui charge et enregistre des agrégats entiers et se présente au domaine comme une collection, en masquant comment et où ils sont stockés.

Service hôte ouvert (open host service)
Patron de carte de contextes dans lequel un contexte amont offre un protocole ou une API unique, bien défini et documenté, à tous ses consommateurs.

Sous-domaine
Domaine distinct de l'activité d'une entreprise, comme les cartes, les commandes, les paiements ou l'affectation des livreurs. Les sous-domaines relèvent du problème, qu'il existe ou non un logiciel.

Sous-domaine de soutien (supporting subdomain)
Sous-domaine propre à l'entreprise et nécessaire à son activité, mais qui n'est pas le terrain sur lequel elle se distingue de la concurrence ; il mérite une conception solide mais plus simple.

Sous-domaine générique
Sous-domaine dont les problèmes sont communs à de nombreuses entreprises et pour lequel de bonnes solutions existent, comme les paiements ou l'identité ; généralement acheté, ou construit de façon minimale.

Style architectural
Famille nommée d'architectures définie par les types d'éléments et de connecteurs qu'elle utilise et par les contraintes sur la manière de les combiner, comme l'architecture en couches ou les tubes et filtres.

Surcoût des microservices (microservice premium)
Le terme de Martin Fowler (2015) pour le coût et le risque supplémentaires qu'ajoutent les microservices, qui ne sont rentables qu'une fois qu'un système et une organisation sont suffisamment complexes.

Système central (core system)
Dans une architecture micronoyau, partie minimale de l'application dont chaque installation a besoin : le flux de travail principal, les services partagés et les points où se branchent les plug-ins.

Test d'architecture
Test automatisé qui fait échouer le build lorsque le code enfreint une règle structurelle, comme une dépendance interdite entre couches ou modules ; parmi les outils figurent ArchUnit et import-linter.

Traitement de flux (stream processing)
Traiter en continu un flux illimité d'enregistrements, chacun peu de temps après son arrivée.

Traitement par lots
Traitement d'un ensemble borné de données en une seule exécution, comme toutes les commandes de la veille, qui se termine ensuite.

Tube (pipe)
Canal à sens unique qui transporte les données d'un filtre au suivant, généralement avec un tampon entre les deux.

Tubes et filtres (pipes and filters)
Style dans lequel les données circulent à travers une chaîne d'étapes de traitement indépendantes (les filtres), reliées par des canaux à sens unique (les tubes).

Unité de déploiement
Ce qui est construit, publié et démarré d'un seul tenant : par exemple un exécutable, une image de conteneur ou une archive web.

Versionnage sémantique
Convention de numérotation des versions, MAJOR.MINOR.PATCH, dans laquelle le numéro majeur change pour les changements cassants, le mineur pour les ajouts rétrocompatibles et le correctif (patch) pour les corrections.