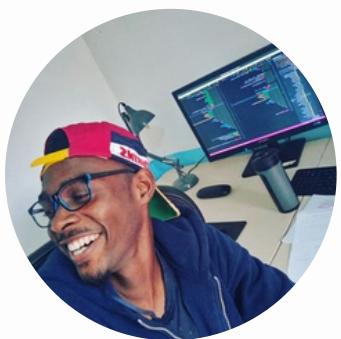
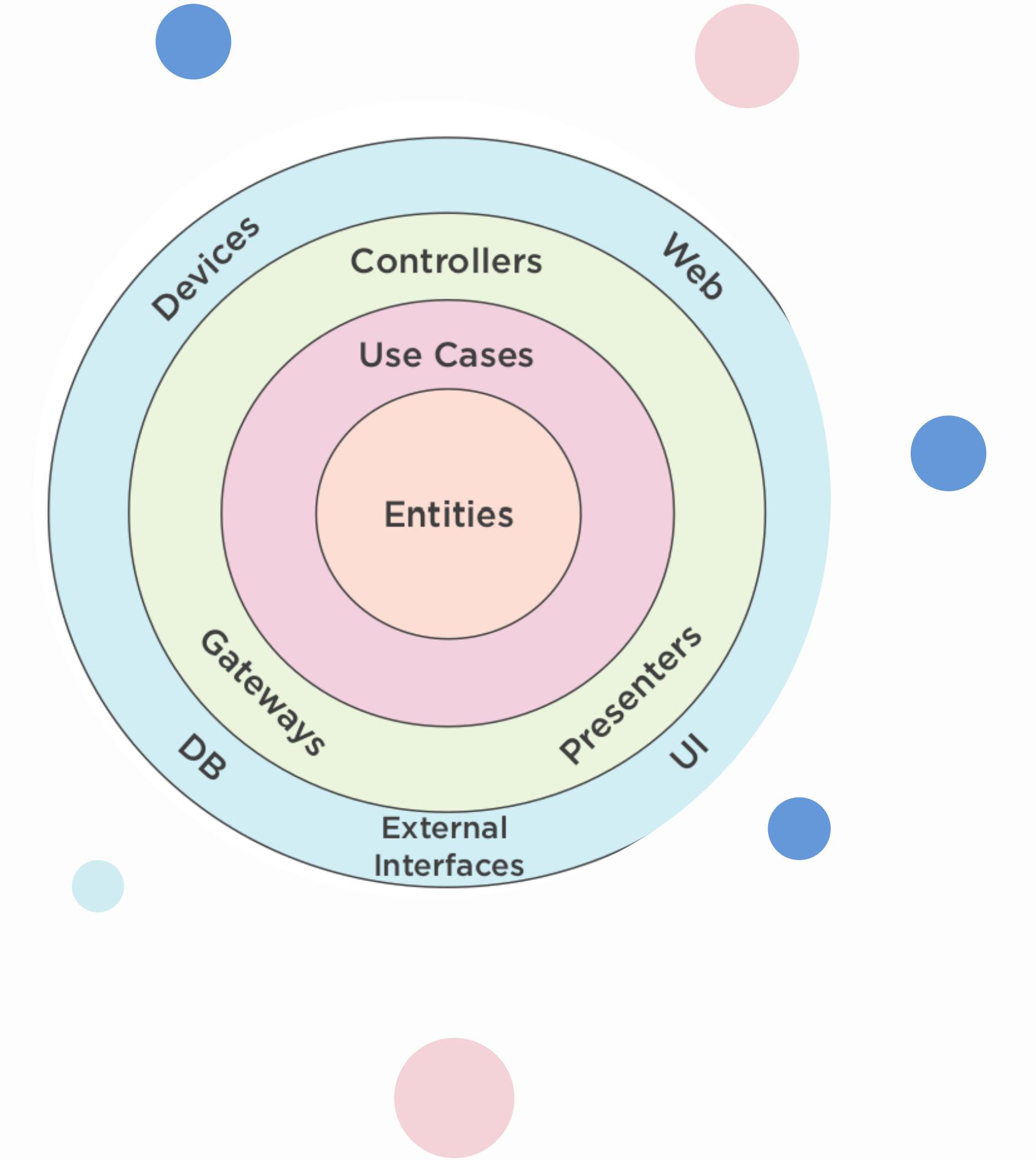


Faire le ménage Architecture: Modèles, pratiques et principes



Régis ATEMENGUÉ

@regis_ate www.regisatemengue.com



Objectifs du cours

Comprendre le nettoyage
Architecture.

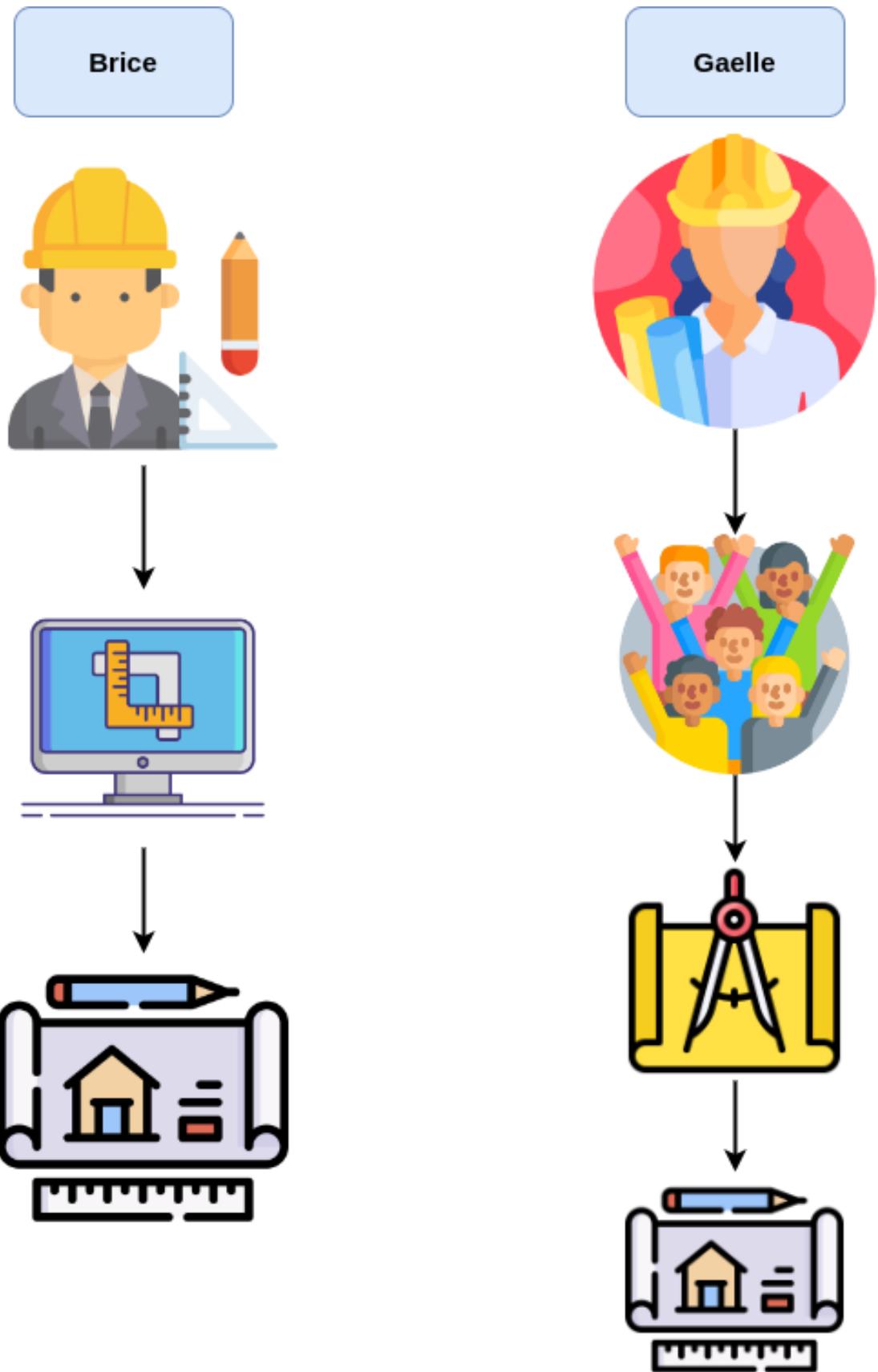
utiliser les meilleures pratiques pour créer
des systèmes maintenables et testables
applications

L'allégorie

Imaginez-moi, si vous voulez, que nous ayons deux architectes. On appellera Brice et Gaëlle. Brice et Gaëlle ont à peu près les mêmes années d'expérience, le même titre de poste et coûteront tous deux le même prix à l'embauche. Nous les embauchons tous les deux pour construire deux nouveaux bâtiments pour notre entreprise dans deux villes différentes

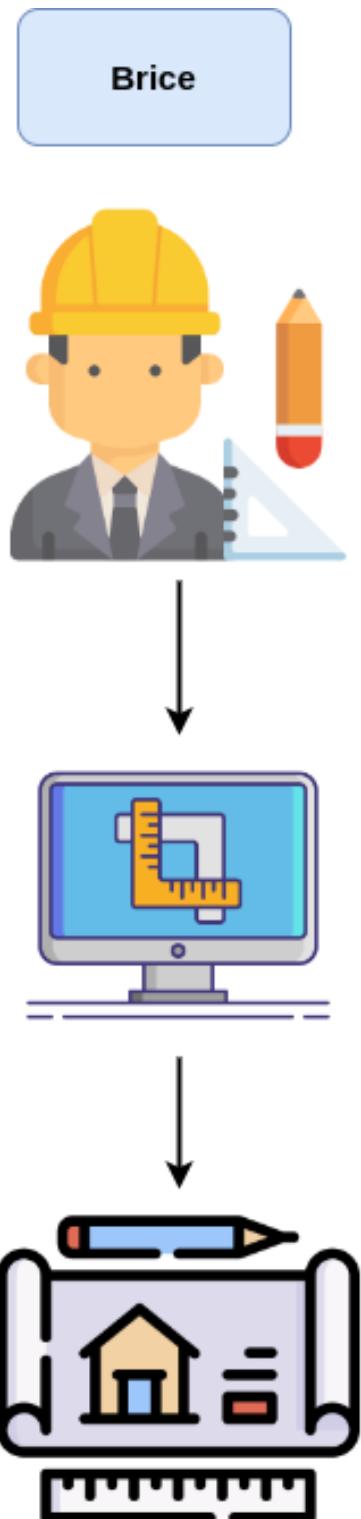
Brice est un expert en architecture classique. Il repousse constamment les limites de la conception architecturale classique en utilisant une technologie de pointe. De plus, il utilise tous ses outils et techniques préférés sur chacun de ses nouveaux projets. Son design est élégant. C'est un design classique, mais à la pointe de la technologie, et il l'a conçu exactement comme il l'aime. En fait, son design impressionne complètement le conseil d'administration. Une fois qu'il a terminé, il remet les plans à l'équipe de construction, leur souhaite bonne chance et passe à la conception de son prochain chef-d'œuvre.

Puis, une fois le bâtiment terminé, il se présente à la grande cérémonie d'ouverture pour récolter les fruits de son accomplissement.



L'allégorie

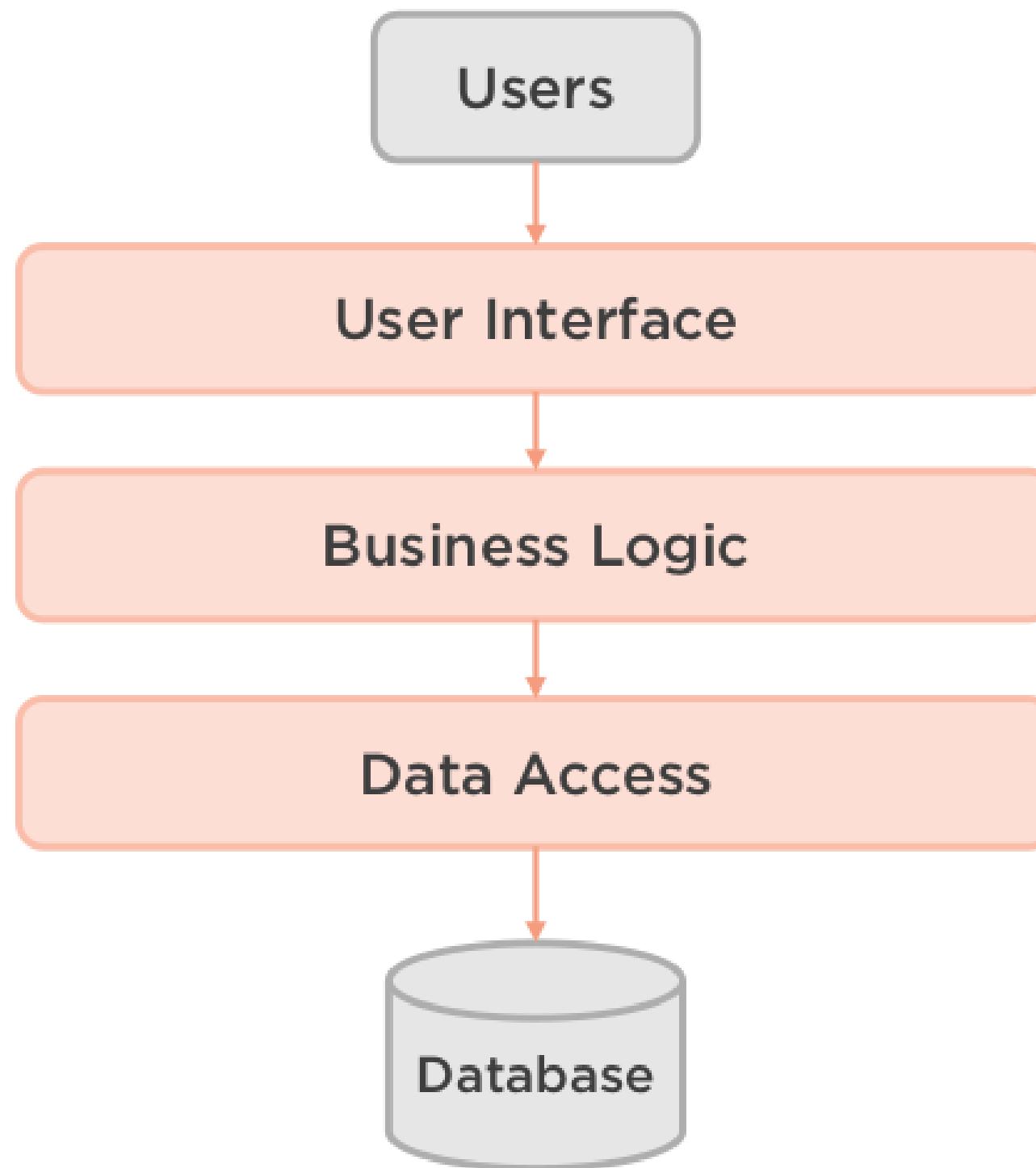
Gaelle a cependant une approche très différente de l'architecture. D'abord, elle commence par discuter avec les futurs habitants de l'immeuble, c'est-à-dire les employés, l'équipe d'entretien, et elle discute même avec le contremaître de construction. Elle découvre quels sont leurs besoins, ce qu'ils trouvent précieux et quel est le niveau d'expérience de l'équipe de construction. Elle conçoit ensuite l'architecture du bâtiment en pensant aux habitants, en tenant compte de leurs besoins à chaque décision prise. Une fois qu'elle a terminé les plans, elle ne se contente pas de les confier à l'équipe de construction et de disparaître. Au lieu de cela, elle travaille côté à côté avec eux chaque jour pour les aider à comprendre et à mettre en œuvre la conception, à recueillir des commentaires et à ajuster les plans si nécessaire, à mesure qu'ils apprennent de meilleures façons de résoudre les problèmes qui surviennent.



Architecture logicielle

l'architecture logicielle est simplement la combinaison de **architecture des applications** et **Architecture du système**, encore une fois par rapport à la structure et à la vision. En d'autres termes, il s'agit de tout ce qui concerne la conception d'un système logiciel ; de la structure du code et de la compréhension du fonctionnement de l'ensemble du système logiciel à un niveau élevé, jusqu'à la manière dont ce système logiciel est déployé sur l'infrastructure.

Haut niveau
Structure
Couches
Composants
Des relations



Architecture désordonnée ou propre



SPAGHETTI



LASAGNE

Qu'est-ce qu'une mauvaise architecture ?



Complexé

Incohérent

Strié

Fragile

Intestable

Impossible à maintenir

SPAGHETTI

Premièrement, une architecture mauvaise ou désordonnée nous apparaît comme des spaghettis. C'est juste un tas de code désordonné avec une boulette de viande occasionnelle jetée ici ou là. Il est presque impossible de naviguer du début à la fin d'une nouille, et il est très difficile d'ajouter ou de remplacer une nouille sans perturber toutes les nouilles voisines.

Qu'est-ce qu'une bonne architecture ?



Simple

Compréhensible

Flexible

Émergent

Testable

Maintenable

LASAGNE

Pour nous, une architecture épurée ressemble à une lasagne. Il présente de belles limites de nouilles horizontales et cohérentes qui divisent uniformément les différentes couches de garniture. Lorsqu'un morceau est trop gros, nous le découpons en composants plus petits avec des lignes de démarcation nettes, nettes et orthogonales, et il comporte une belle couche de présentation reposant sur une pile de fonctionnalités comestibles.

Mauvaise et bonne architecture

Alors, en termes plus techniques, qu'est-ce qu'une architecture mauvaise ou désordonnée ? Eh bien, c'est une architecture qui est **complexe**, mais en raison d'une complexité accidentelle plutôt que d'une complexité nécessaire. C'est **incohérent** dans le sens où les pièces ne semblent pas s'emboîter. C'est **rigide**, c'est-à-dire que l'architecture résiste au changement ou rend difficile son évolution au fil du temps. C'est fragile, toucher une partie du code ici pourrait casser une autre partie du code ailleurs, c'est **intestable**, c'est-à-dire que vous aimeriez vraiment écrire des tests unitaires et des tests d'intégration, mais l'architecture vous combat à chaque étape, et finalement, toutes ces choses conduisent à une architecture qui est **inmaintenable** sur la durée de vie du projet.

D'un autre côté, nous savons que nous avons une architecture bonne et propre lorsqu'elle est **simple** ou du moins, c'est aussi complexe que nécessaire, et cette complexité n'est pas accidentelle. C'est **compréhensible**, c'est ça facilera raisonner sur le logiciel dans son ensemble, c'est **flexible**, on peut facilement adapter le système pour répondre à l'évolution des besoins, c'est émergent, l'architecture évolue au cours de la vie du projet, c'est **testable**, l'architecture fait **tester plus facilement**, pas plus difficile, et finalement tout cela conduit à une architecture plus facile à maintenir tout au long de la durée de vie du projet.

Qu'est-ce que l'architecture propre ?

L'architecture propre est une architecture conçue pour le **habitants** de l'architecture, pas pour l'architecte ou pour la machine. Une architecture propre est **une philosophie de l'essentialisme architectural**. Il s'agit de se concentrer sur ce qui est véritablement essentiel à l'architecture du logiciel plutôt que sur ce qui n'est qu'un détail d'implémentation. Par concevoir pour les habitants, nous entendons les personnes qui vivront au sein de l'architecture pendant la durée de vie du projet.

Cela signifie que les utilisateurs du système, **les bâtiment des développeurs** le système et **les développeurs maintiennent** le système. En ne concevant pas pour l'architecte, nous entendons que l'architecte doit mettre de côté ses propres désirs, préférences et souhaits, et ne considérer que ce qui est le mieux pour les habitants de l'architecture à chaque décision prise. En ne concevant pas pour la machine, nous entendons que nous devons d'abord optimiser l'architecture pour les besoins des habitants, c'est-à-dire les utilisateurs et les développeurs, et n'optimiser pour la machine que lorsque le coût des performances pose problème aux utilisateurs, qui sont des habitants de la machine. L'architecture, l'emporte sur l'avantage d'un design épuré pour les développeurs qui sont également des habitants de l'architecture.



Pourquoi investir dans du code propre ?

Coût/bénéfice

La principale justification de l'investissement dans une architecture propre est principalement un argument coût-bénéfice. Notre objectif en tant qu'architectes logiciels est de minimiser le coût de création et de maintenance du logiciel tout en maximisant l'avantage qu'est la valeur commerciale apportée par le logiciel.

Minimiser les coûts

La seconde architecture propre construit uniquement ce qui est nécessaire lorsque cela est nécessaire. Nous essayons de créer uniquement les fonctionnalités et l'architecture correspondante qui sont nécessaires pour répondre aux besoins immédiats des utilisateurs, dans l'ordre de la valeur commerciale perçue de chaque fonctionnalité. Nous essayons de le faire sans créer de complexité accidentelle, de fonctionnalités inutiles, d'optimisations prématurées des performances ou d'embellissements architecturaux. Cela contribue à réduire le coût de création du système. Troisièmement, une architecture propre optimise la maintenabilité

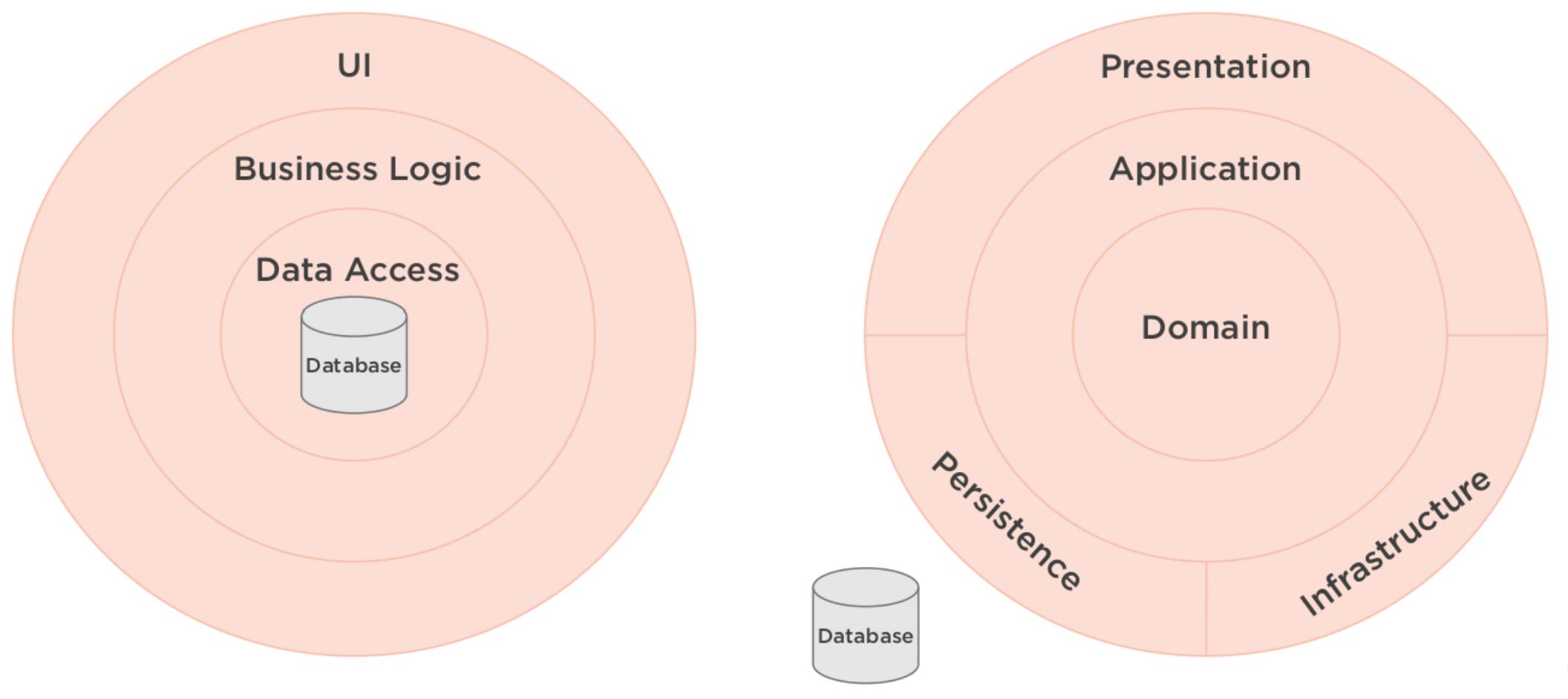
Maximiser la valeur

Pour une application d'entreprise moyenne avec un cycle de vie suffisamment long, disons environ 10 ans, nous consacrons beaucoup plus de temps et d'argent à la maintenance du système qu'à sa création. Cette concentration sur les activités à valeur ajoutée et à réduction des coûts tente de maximiser le retour sur investissement du logiciel dans son ensemble. En outre, une architecture propre offre plusieurs autres avantages en matière de valeur ajoutée et de réduction des coûts ;

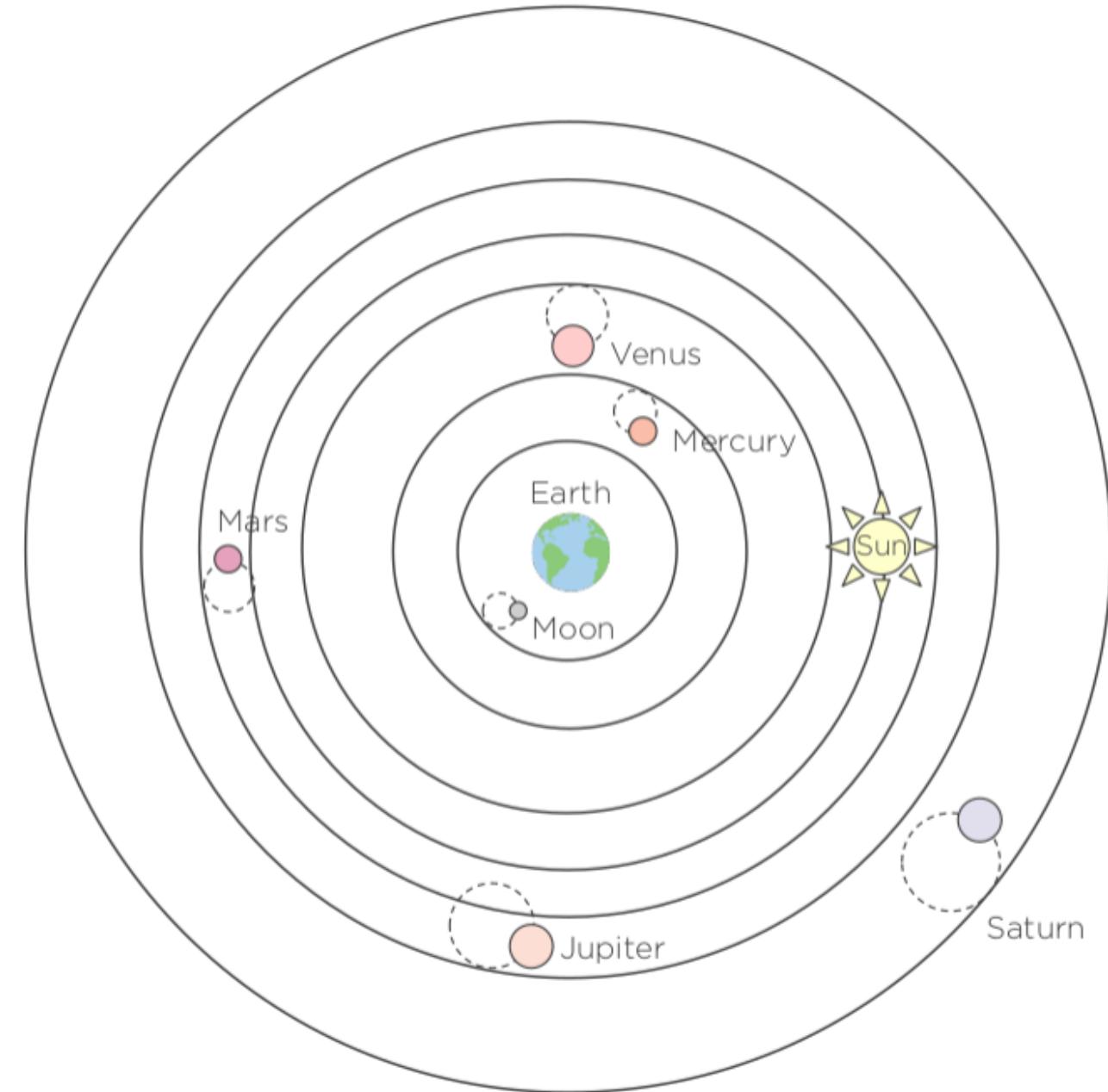
Maximiser le retour sur investissement

Architecture centrée sur le domaine

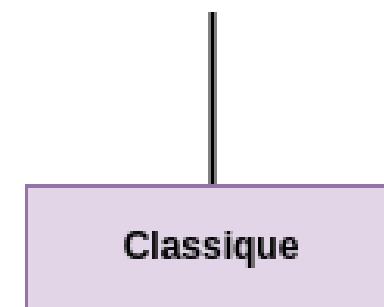
Database-centric vs.
Domain-centric Architecture



Architecture centrée sur le domaine



Geocentric Model

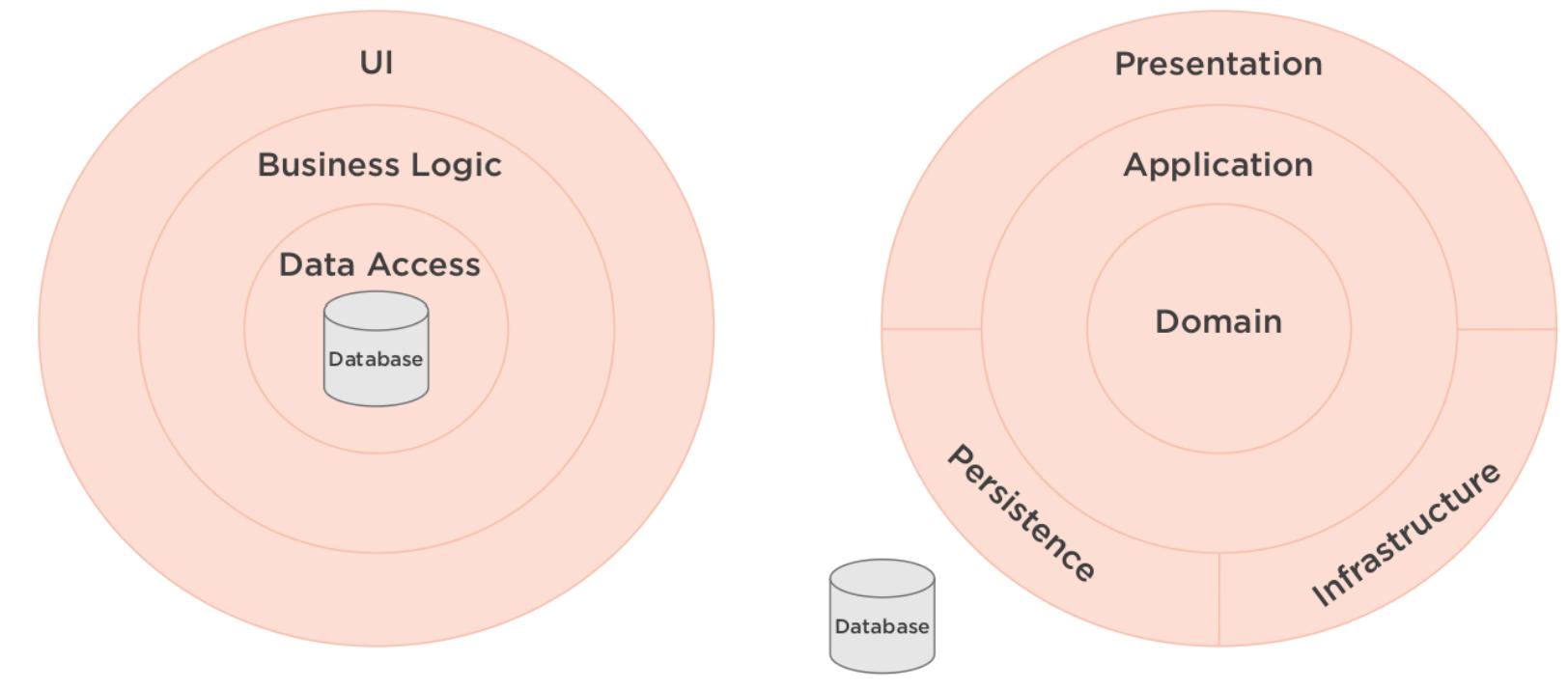


Heliocentric Model



Architecture centrée sur le domaine

Database-centric vs.
Domain-centric Architecture



L'évolution de la pensée a lieu dans le monde de l'architecture logicielle. Nous avons les architectures classiques centrées sur les bases de données à trois couches. Sa principale caractéristique est que l'interface utilisateur, la logique métier et la couche d'accès aux données tournent autour de la base de données. La base de données est **essentielle** et est donc au cœur de cette architecture.

Une nouvelle perspective a changé la façon dont beaucoup d'entre nous regardent notre architecture. Plutôt que de placer la base de données au centre de notre architecture, certains d'entre nous mettent le **domaine au centre**, et faire de la base de données simplement un détail d'implémentation en dehors de l'architecture. **Ici le domaine est essentiel, et la base de données n'est qu'un détail.**



*“The first concern of the architect
is to make sure that the house is
usable, it is not to ensure that the
house is made of brick.”*

- Uncle Bob



Essentiel vs détail

L'espace est**essentiel**



Le matériau de construction est un**détail**

L'ornementation est un**détail**

Le domaine est**essentiel**

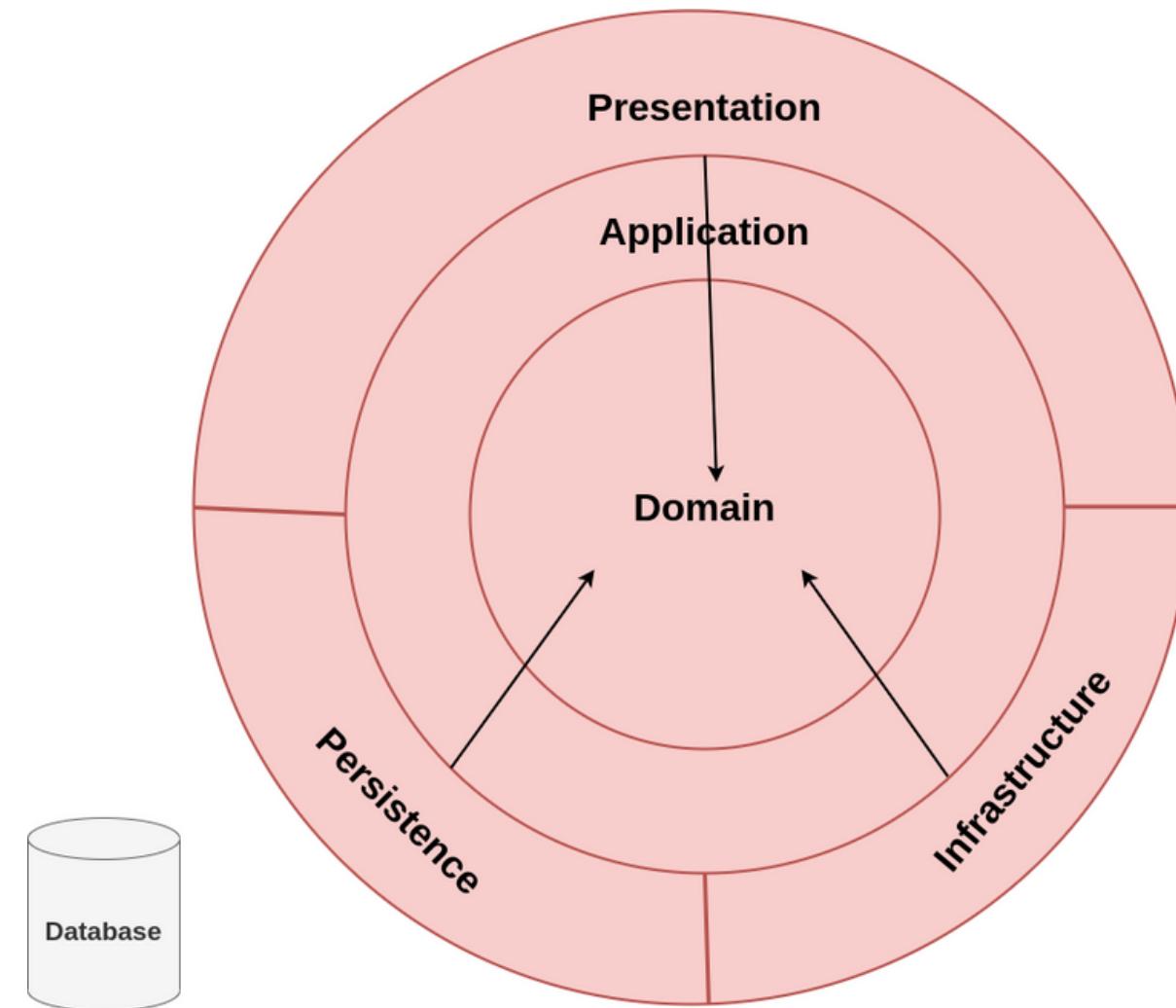
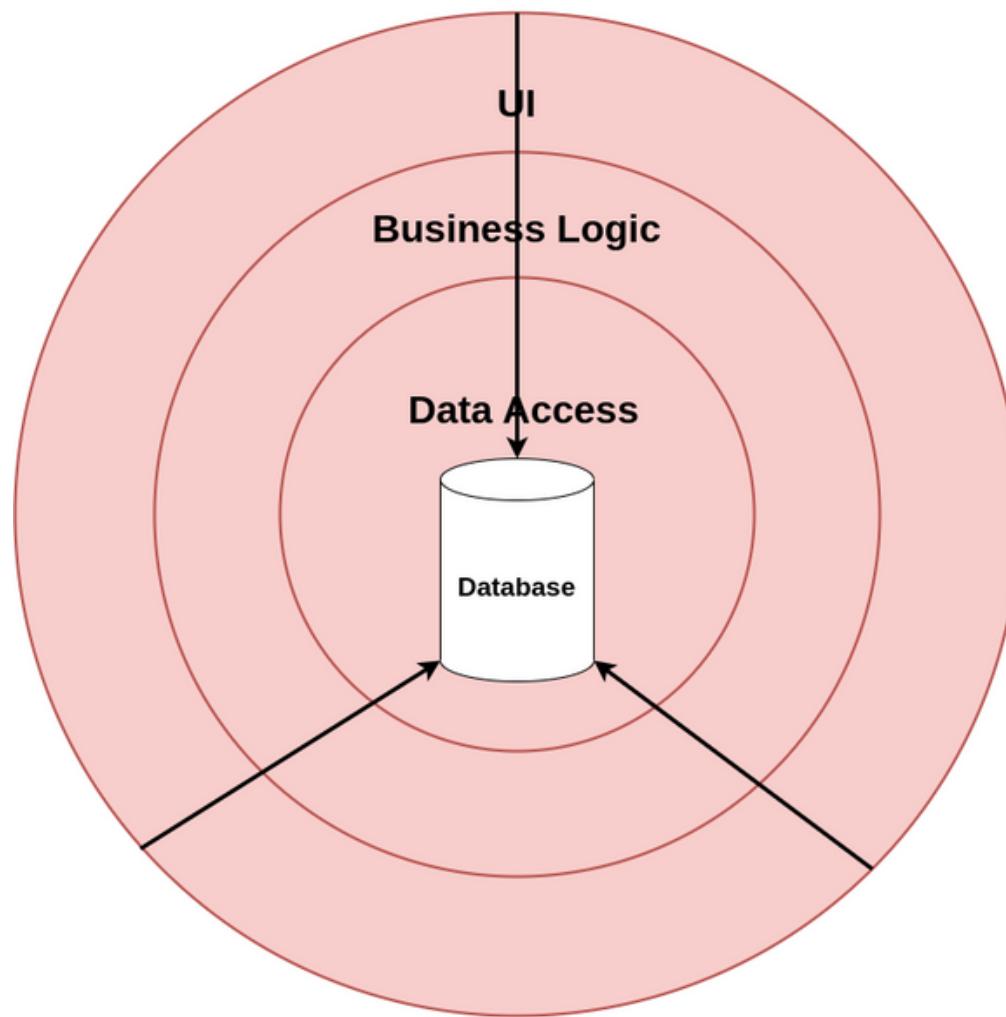


Les cas d'utilisation sont**essentiel**

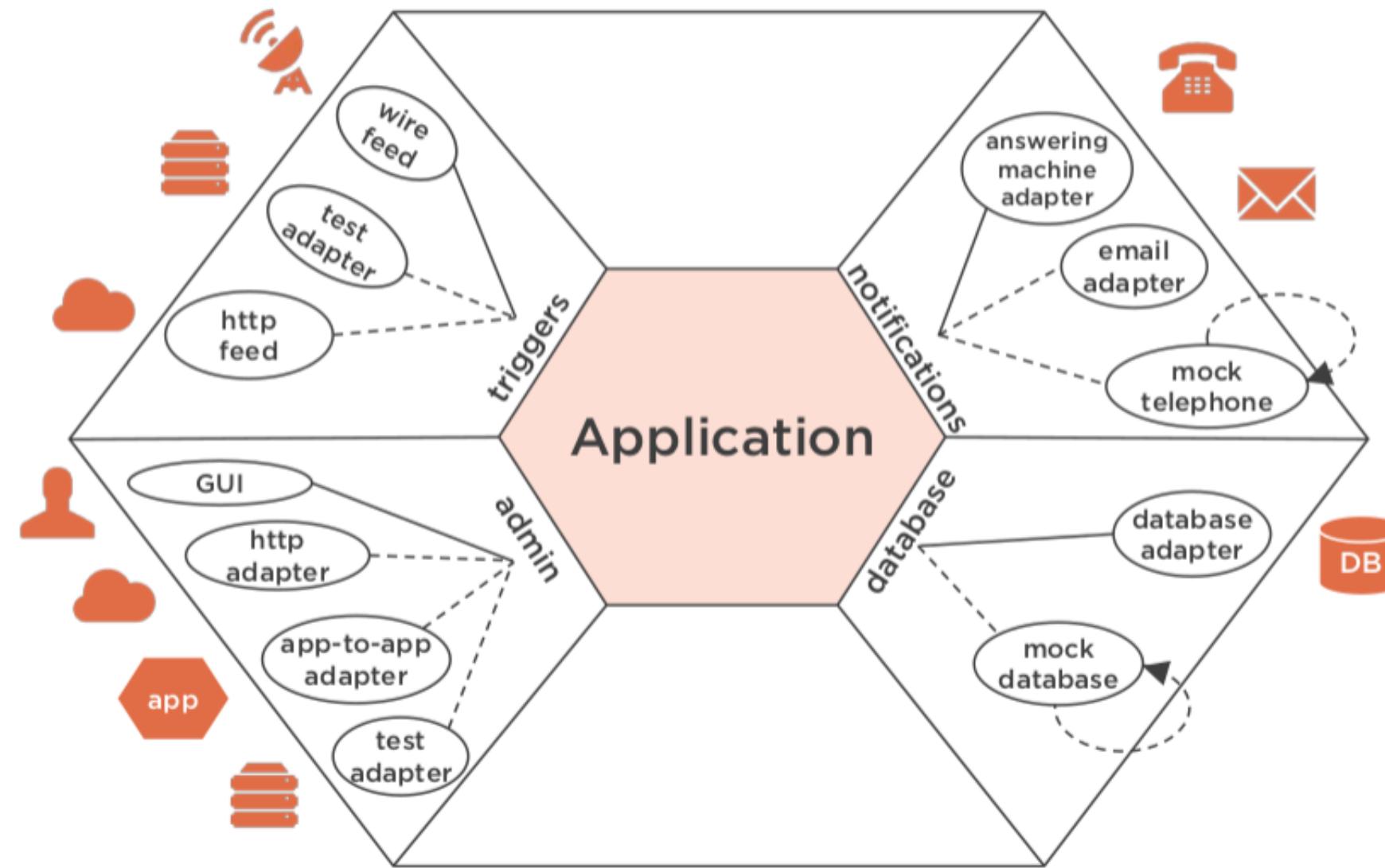
La présentation est un**détail**

La persistance est un**détail**

Architecture centrée sur le domaine ou centrée sur la base de données



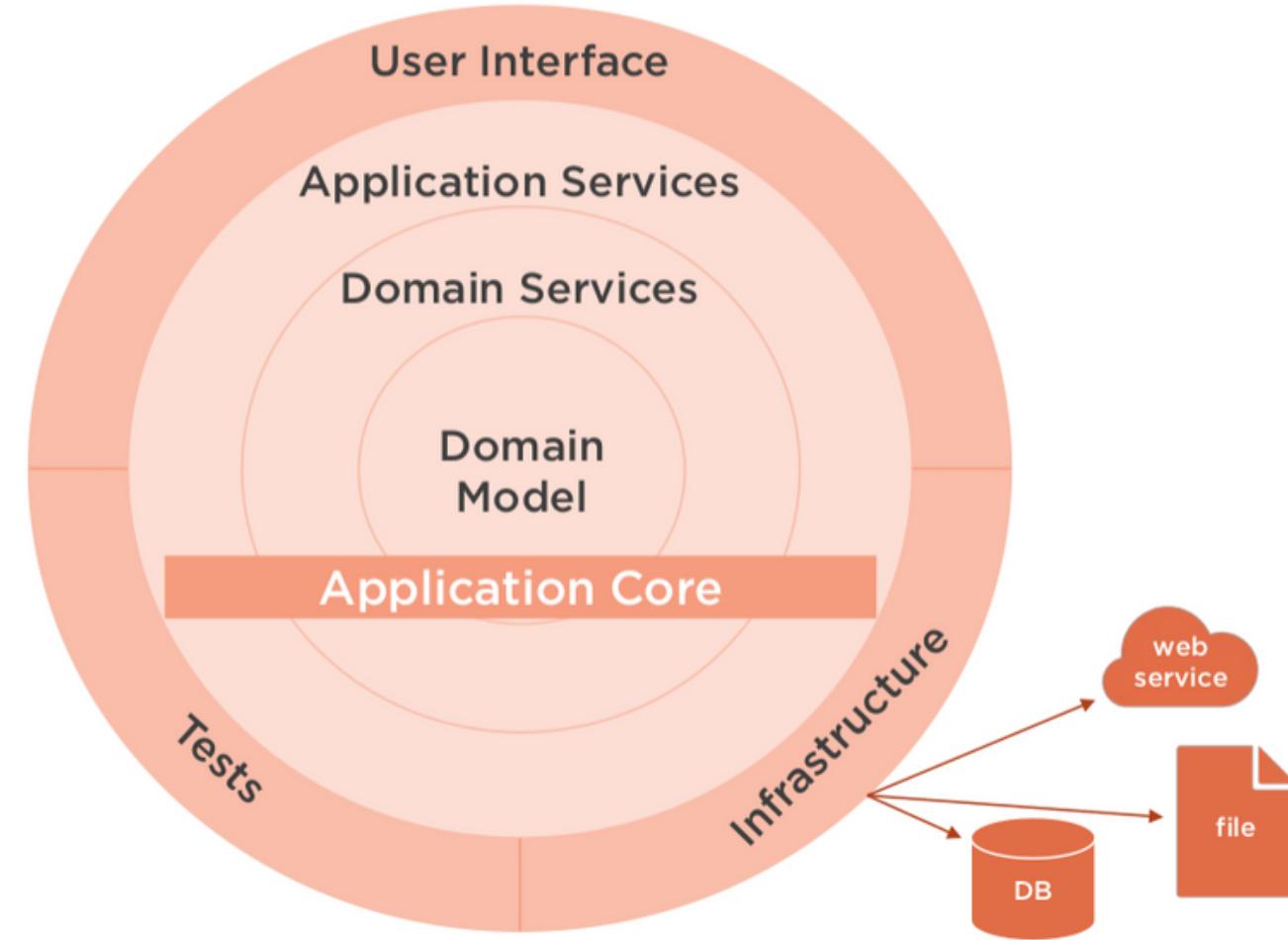
Avec les architectures centrées sur la base de données, la base de données est essentielle, elle est donc au centre de l'application et toutes les dépendances pointent vers la base de données. Avec les architectures centrées sur le domaine, le domaine et les cas d'utilisation sont essentiels, et la présentation et la persistance ne sont qu'un détail. Le domaine est donc au centre de l'application enveloppé dans une couche d'application et toutes les dépendances pointent vers le domaine.



Original source: <http://alistair.cockburn.us/Hexagonal+architecture>

L'architecture hexagonale d'Alistair Cockburn. Il s'agit d'une architecture en couches avec la couche d'application, et donc transitoirement, le domaine au centre de l'architecture. En outre, il s'agit d'une architecture de plugins qui comprend des ports et des adaptateurs. Essentiellement, les couches externes de l'architecture adaptent la couche d'application interne aux différents supports de présentation, supports de persistance et systèmes externes. Vous pouvez exécuter, et donc tester, une isolation dans cette architecture d'application complète sans interface utilisateur, sans base de données et sans dépendances externes.

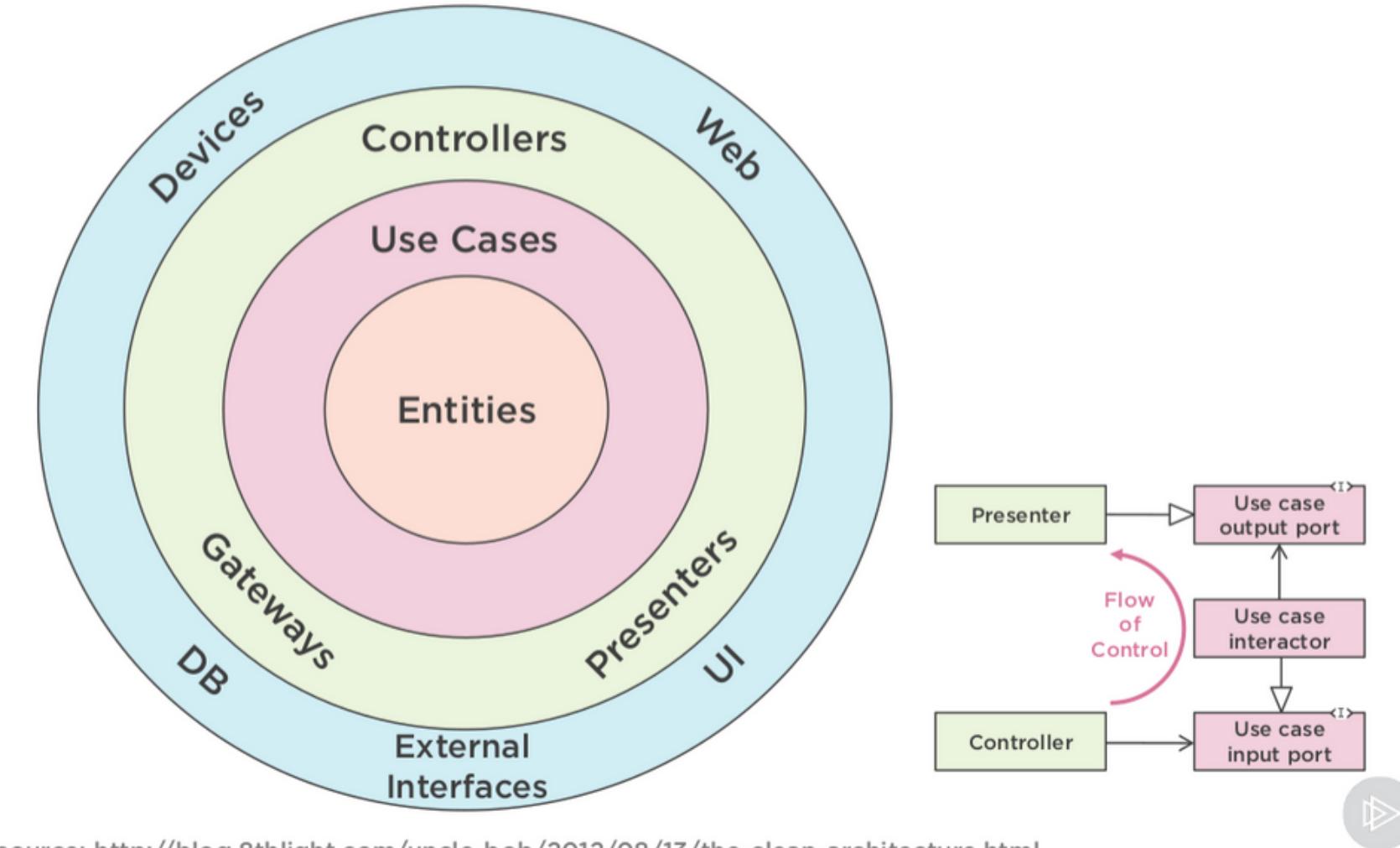
Onion Architecture



Original source: <http://jeffreypalermo.com/blog/the-onion-architecture-part-2/>

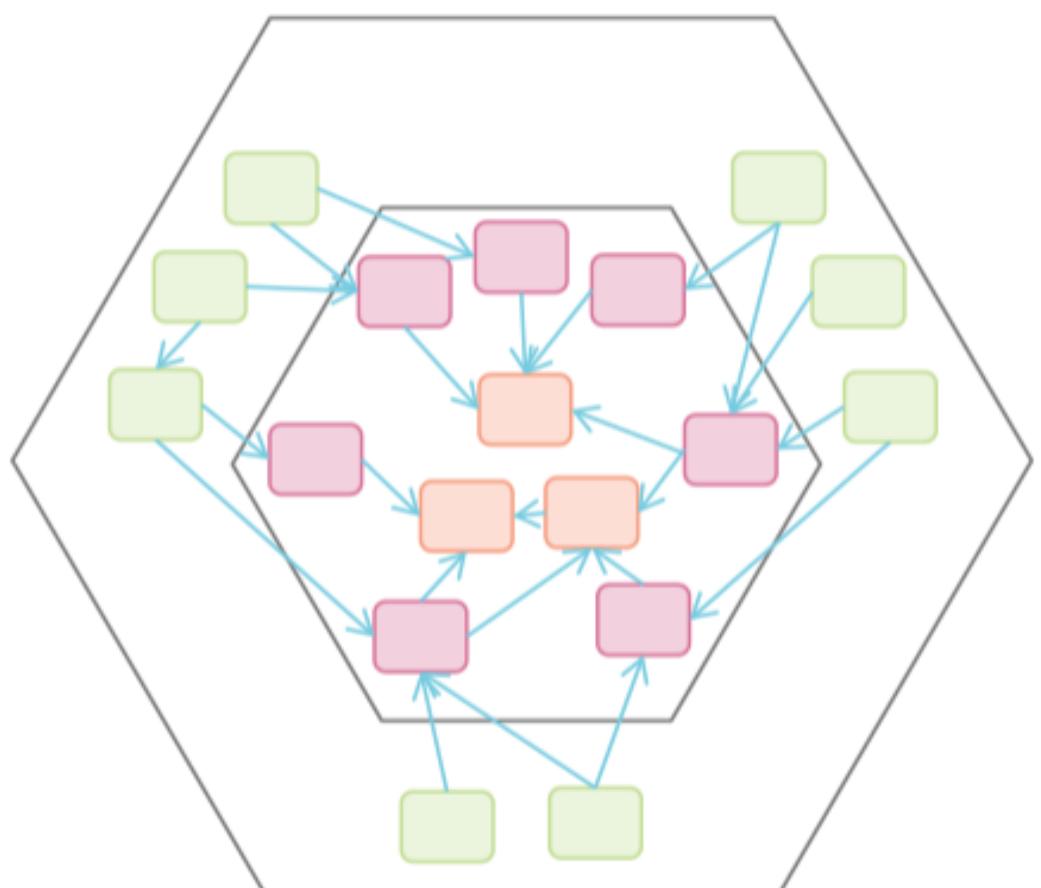
L'architecture de l'oignon par Jeffrey Palermo. Il s'agit également d'une architecture en couches avec le domaine au centre entouré d'une couche d'application. Les couches externes se composent d'une interface utilisateur mince comme couche de présentation et d'une couche d'infrastructure, qui inclut la persistance. De plus, toutes les dépendances pointent vers le centre de l'architecture, c'est-à-dire qu'aucune couche interne ne connaît l'existence d'une couche externe. Encore une fois, vous pouvez tester cette architecture d'application de manière isolée sans interface utilisateur, base de données ou dépendances externes.

The Clean Architecture

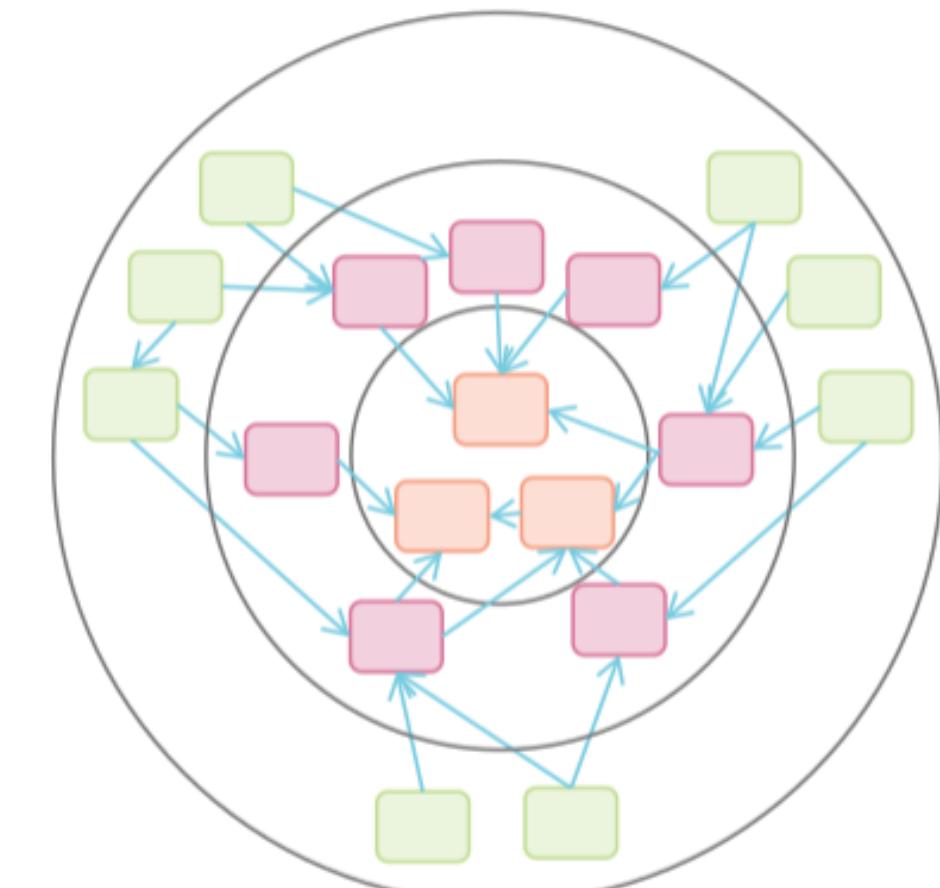


l'architecture épurée d'Oncle Bob. Encore une fois, il s'agit d'une architecture en couches avec le domaine, c'est-à-dire les entités, au centre entouré d'une couche application, c'est-à-dire les cas d'utilisation. La couche externe est constituée de ports et d'adaptateurs adaptant le cœur de l'application aux dépendances externes via des contrôleurs, des passerelles et des présentateurs. De plus, Uncle Bob va encore plus loin en incorporant le modèle d'architecture BCE d'Ivar Jacobson pour expliquer comment la couche de présentation et la couche d'application doivent être connectées.

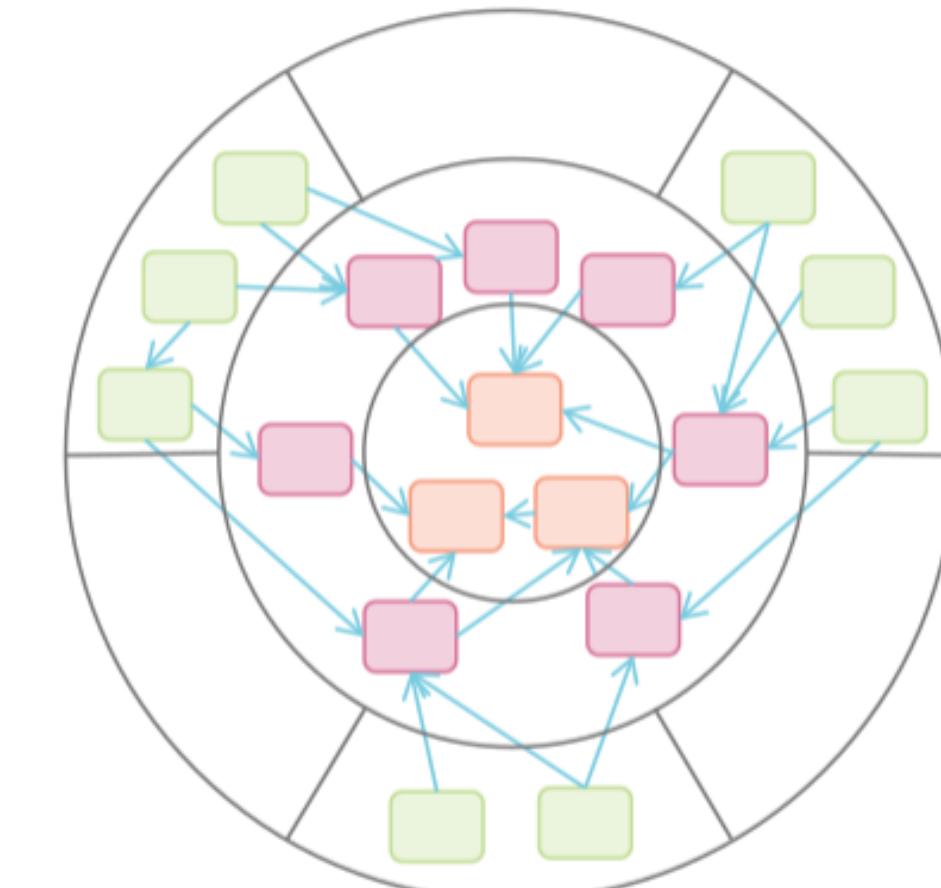
It's All the Same Thing



Hexagonal



Onion

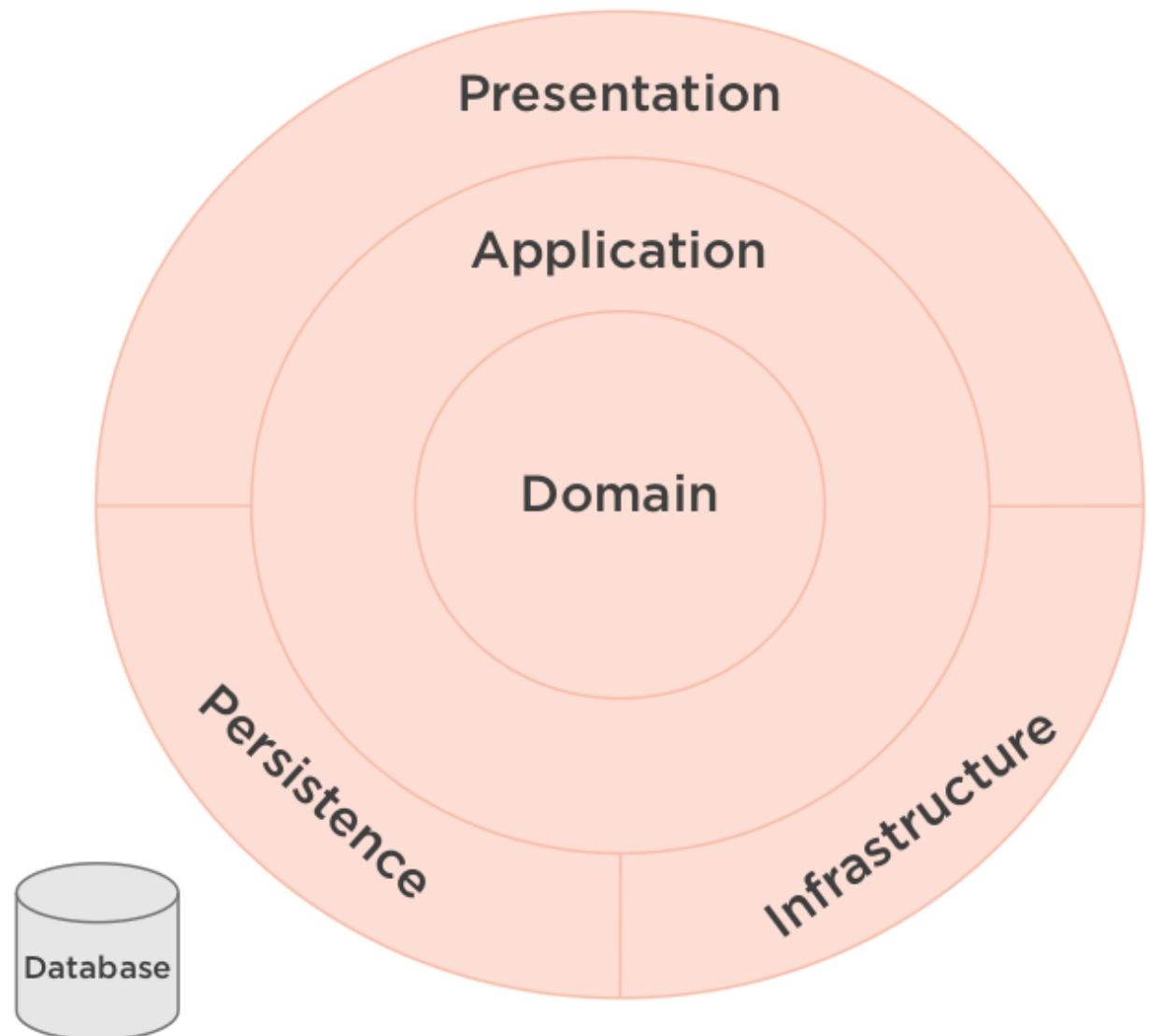


Clean

Pourquoi utiliser une architecture centrée sur le domaine ?

Premièrement, l'**accent est mis sur le domaine**, ce qui est essentiel pour les habitants de l'architecture, c'est-à-dire les utilisateurs et les développeurs, et offre plusieurs avantages et réductions de coûts pour nos logiciels.

Deuxièmement, il y a **moins de couplage** entre la logique du domaine et les détails d'implémentation, tels que la présentation, la base de données et le système d'exploitation. Cela permet au système d'être plus flexible et adaptable, et nous pouvons faire évoluer l'architecture beaucoup plus facilement au fil du temps. Troisièmement, l'utilisation d'une architecture centrée sur le domaine nous permet de **de intégrer la conception pilotée par domaine (DDD)** qui est un excellent ensemble de stratégies développées par Eric Evans pour gérer des domaines commerciaux d'un haut degré de complexité.



Conception pilotée par domaine (DDD)

La conception pilotée par domaine est une approche du développement logiciel qui centre le développement sur la programmation d'un modèle de domaine doté d'une riche compréhension des processus et des règles d'un domaine.

Martin Fowler

Proposition de valeur de DDD

**Principles and patterns to
solve difficult problems**

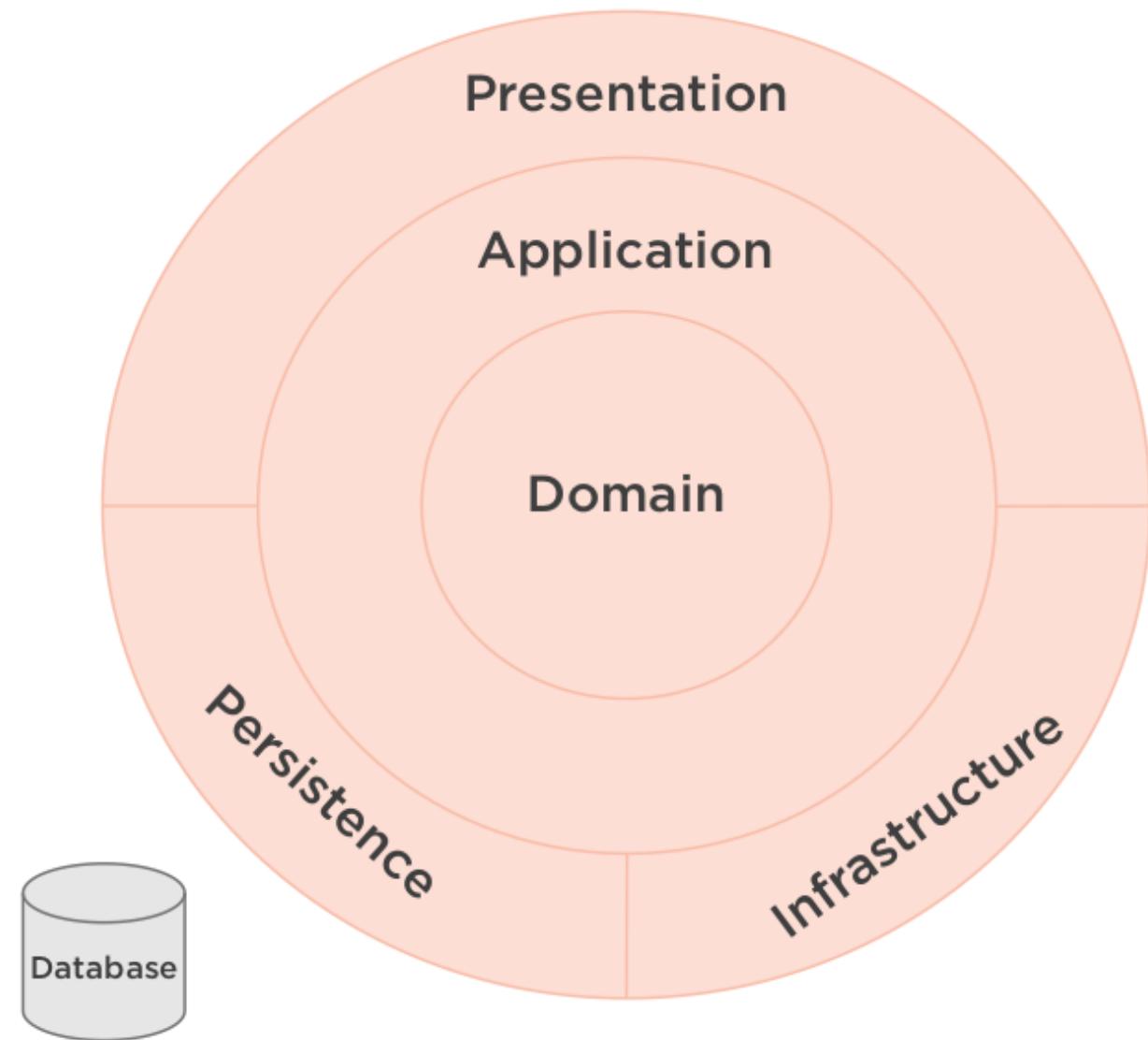
**History of success
with complex projects**

**Aligns with practices
from our own **experience****

**Clear, readable, testable code
that represents the domain**

pourquoi ne voudrions-nous pas utiliser une architecture centrée sur le domaine ?

D'abord, **le changement est difficile.** La plupart des développeurs sortent de l'université après avoir seulement appris l'architecture traditionnelle centrée sur les bases de données à trois couches. En outre, il peut également s'agir du seul modèle architectural que l'architecte connaît suffisamment bien pour pouvoir donner des conseils. Deuxièmement, **il nécessite plus de réflexion pour mettre en œuvre une conception centrée sur le domaine.** Vous devez savoir quelles classes appartiennent à la couche de domaine et quelles classes appartiennent à la couche d'application plutôt que de simplement tout jeter dans une couche de logique métier. Troisièmement, il a un **coût initial plus élevé** pour mettre en œuvre cette architecture par rapport à une architecture traditionnelle centrée sur les bases de données à trois couches.



Principe de conception CleanMicroservice

Principe de conception CleanMicroservice

La conception épurée des microservices favorise la conception orientée objet avec une séparation des préoccupations obtenue en divisant le logiciel en couches à l'aide de **le principe d'inversion de dépendances (programmation sur interfaces)**.

Une conception épurée des microservices présente les avantages suivants :

- Non lié à un cadre unique
- Non lié à une technologie API unique comme REST ou GraphQL
- Unité testable
- Non lié à un client spécifique (fonctionne avec les clients Web, de bureau, de console et mobiles) • Non lié à une base de données spécifique
- Ne dépend pas d'une implémentation de service externe spécifique

Principe de conception CleanMicroservice

Important Design Principles

Dependency
inversion

Separation of
concerns

Single responsibility

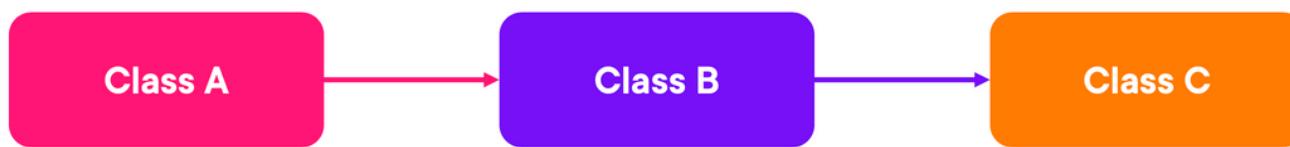
DRY

Persistence
ignorance

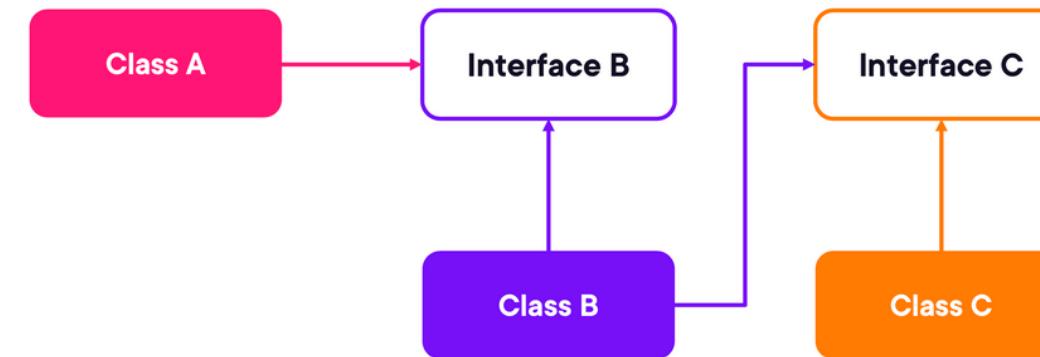
Inversion de dépendance : principe ProgramAgainst des interfaces (principe d'inversion de dépendance généralisée)

N'écrivez pas de programmes dans lesquels les dépendances internes sont des types d'objets concrets, mais programmez plutôt sur des interfaces. Une exception à cette règle concerne les classes de données sans comportement (sans compter les simples getters/setters)

Typical Approach



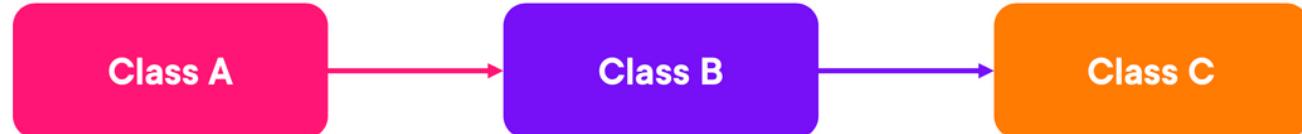
Adding Dependency Inversion



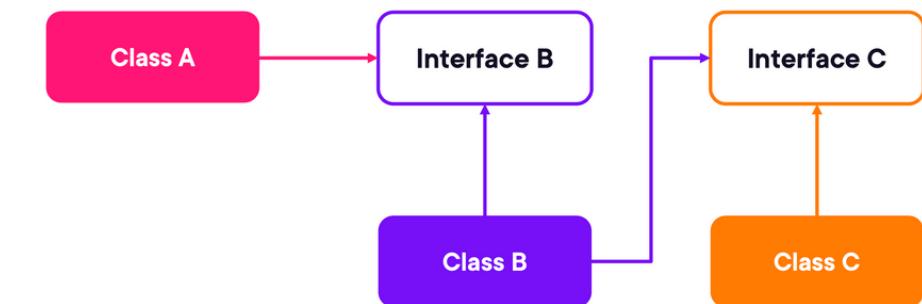
Inversion de dépendance : principe ProgramAgainst des interfaces (principe d'inversion de dépendance généralisée)

Une interface est utilisée pour définir un type de base abstrait. Diverses implémentations peuvent être introduites pour implémenter l'interface. Lorsque vous souhaitez modifier le comportement d'un programme, vous créez une nouvelle classe qui implémente une interface, puis utilisez une instance de cette classe. De cette façon, vous pouvez mettre en pratique le principe ouvert-fermé. Vous pouvez considérer ce principe comme une condition préalable à l'utilisation du **principe ouvert-fermé** effectivement. Le principe du programme contre les interfaces est une généralisation du principe d'inversion de dépendance du **SOLID** des principes:

Typical Approach



Adding Dependency Inversion



Inversion de dépendance : principe ProgramAgainst des interfaces (principe d'inversion de dépendance généralisée)

Le principe d'inversion de dépendance est une méthodologie pour **accouplement lâche** cours de logiciels.

En suivant ce principe, les relations de dépendance conventionnelles entre les classes de haut niveau et les classes de bas niveau sont inversées, rendant ainsi les classes de haut niveau indépendantes des détails d'implémentation de bas niveau.

Le principe d'inversion de dépendance stipule :

- Les classes de haut niveau ne doivent rien importer des classes de bas niveau.
- Les abstractions (= interfaces) ne doivent pas dépendre d'implementations concrètes (classes) •

Les implementations concrètes (classes) doivent dépendre d'abstractions (= interfaces)

Inversion de dépendance : principe ProgramAgainst des interfaces (principe d'inversion de dépendance généralisée)

Le principe d'inversion de dépendance est une méthodologie pour **accouplement lâche** cours de logiciels.

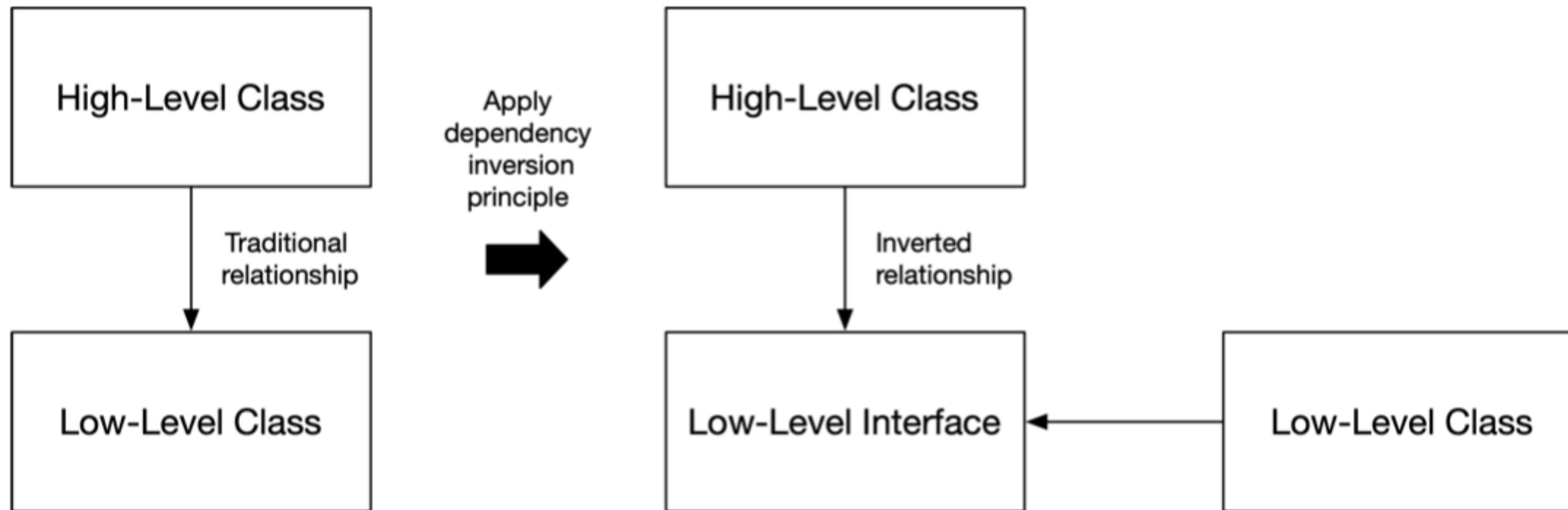
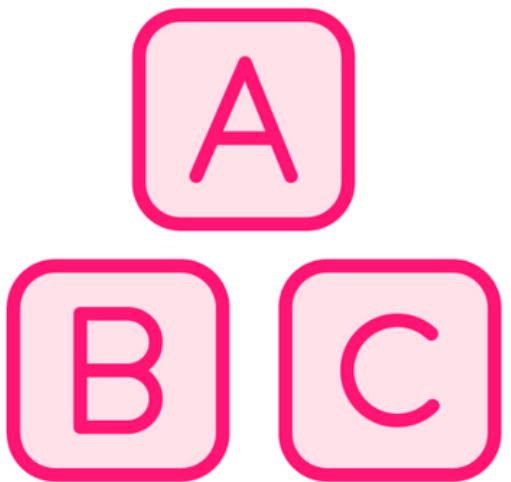


Fig 3.1 Dependency Inversion Principle

Séparation des préoccupations

La séparation des préoccupations est un modèle/principe de conception d'architecture logicielle permettant de séparer une application en sections distinctes, de sorte que chaque section répond à une préoccupation distincte. Essentiellement, la séparation des préoccupations est une question d'ordre. L'objectif global de la séparation des préoccupations est d'établir un système bien organisé où chaque partie remplit un rôle significatif et intuitif tout en maximisant sa capacité d'adaptation au changement.

Separation of Concerns



Split into blocks of functionality

- Each covering a concern

More modular code

- Encapsulation within a module

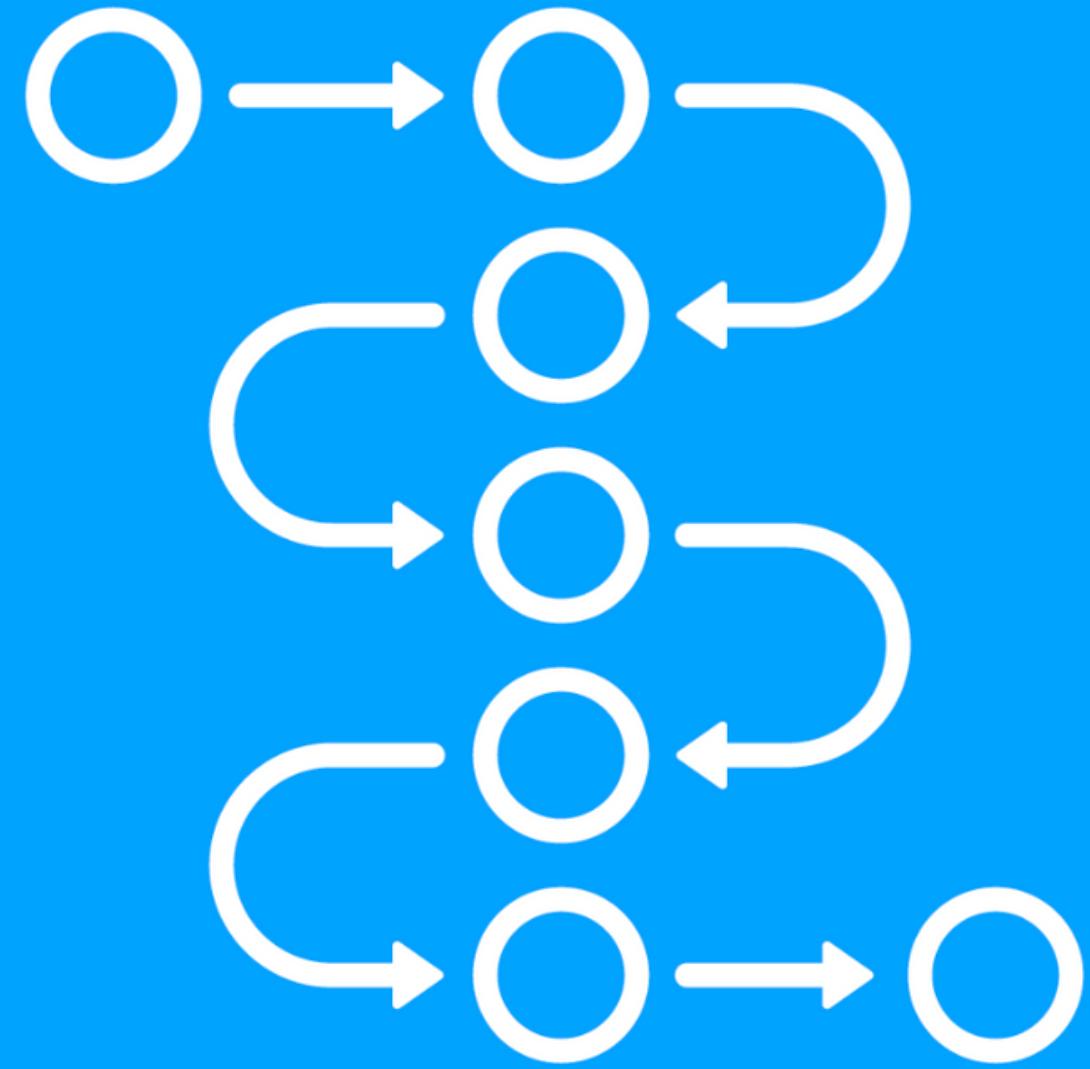
Typical layered application

Easier to maintain

Séparation des préoccupations

Une entité logicielle ne devrait avoir qu'une seule responsabilité à son niveau d'abstraction.

Un système logiciel se situe au plus haut niveau de la hiérarchie logicielle et doit avoir un seul objectif dédié. Par exemple, il peut y avoir un système logiciel de commerce électronique ou de paie. Mais il ne devrait pas y avoir de système logiciel qui gère à la fois les activités liées au commerce électronique et à la paie. Si vous étiez un éditeur de logiciels et aviez créé un système logiciel de commerce électronique, il serait facile de le vendre à des clients souhaitant une solution de commerce électronique. Mais si vous aviez créé un système logiciel qui englobe à la fois les fonctionnalités de commerce électronique et de paie, il serait difficile de le vendre aux clients qui souhaitent uniquement une solution de commerce électronique, car ils disposent peut-être déjà d'un système logiciel de paie et, bien sûr, n'en ont pas. je n'en veux pas un autre



DRY
(aka Don't Repeat Yourself)

Less code repetition

Easier to make changes

Ignorance persistante

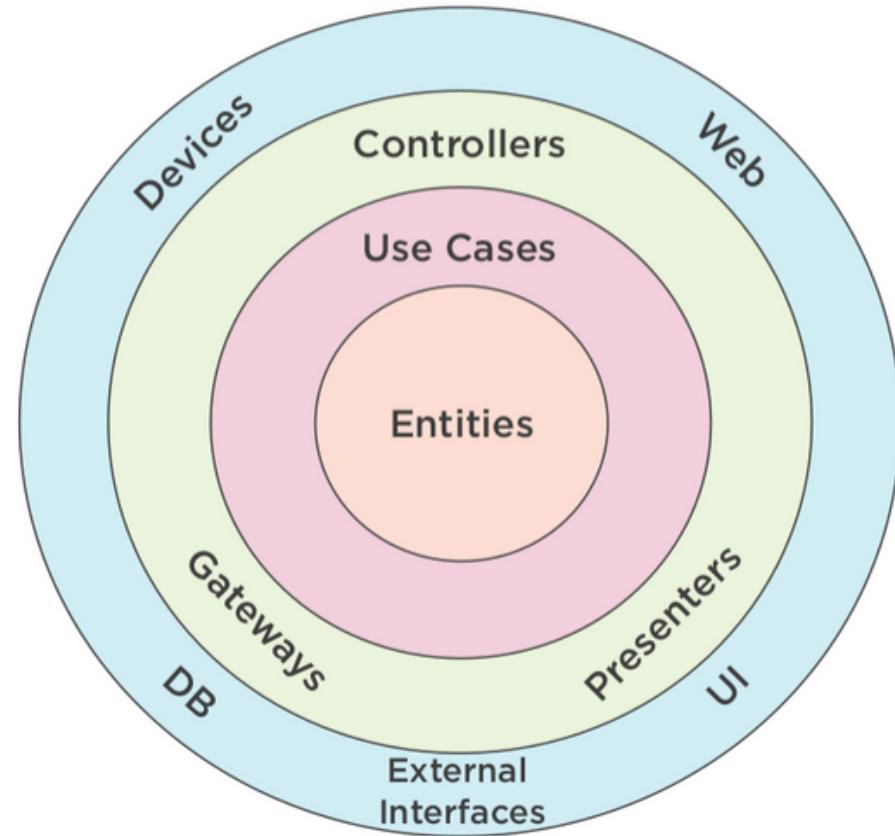
Le principe de l'ignorance de persistance (PI) stipule que les classes modélisant le domaine métier dans une application logicielle ne devraient pas être affectées par la manière dont elles pourraient être conservées. Ainsi, leur conception doit refléter aussi fidèlement que possible la conception idéale nécessaire pour résoudre le problème commercial en question, et ne doit pas être entachée de préoccupations liées à la manière dont l'état des objets est sauvegardé puis récupéré.

Persistence Ignorance
POCO
Domain classes
shouldn't be
impacted by how
they are persisted
**Typically requires base
class or attributes**

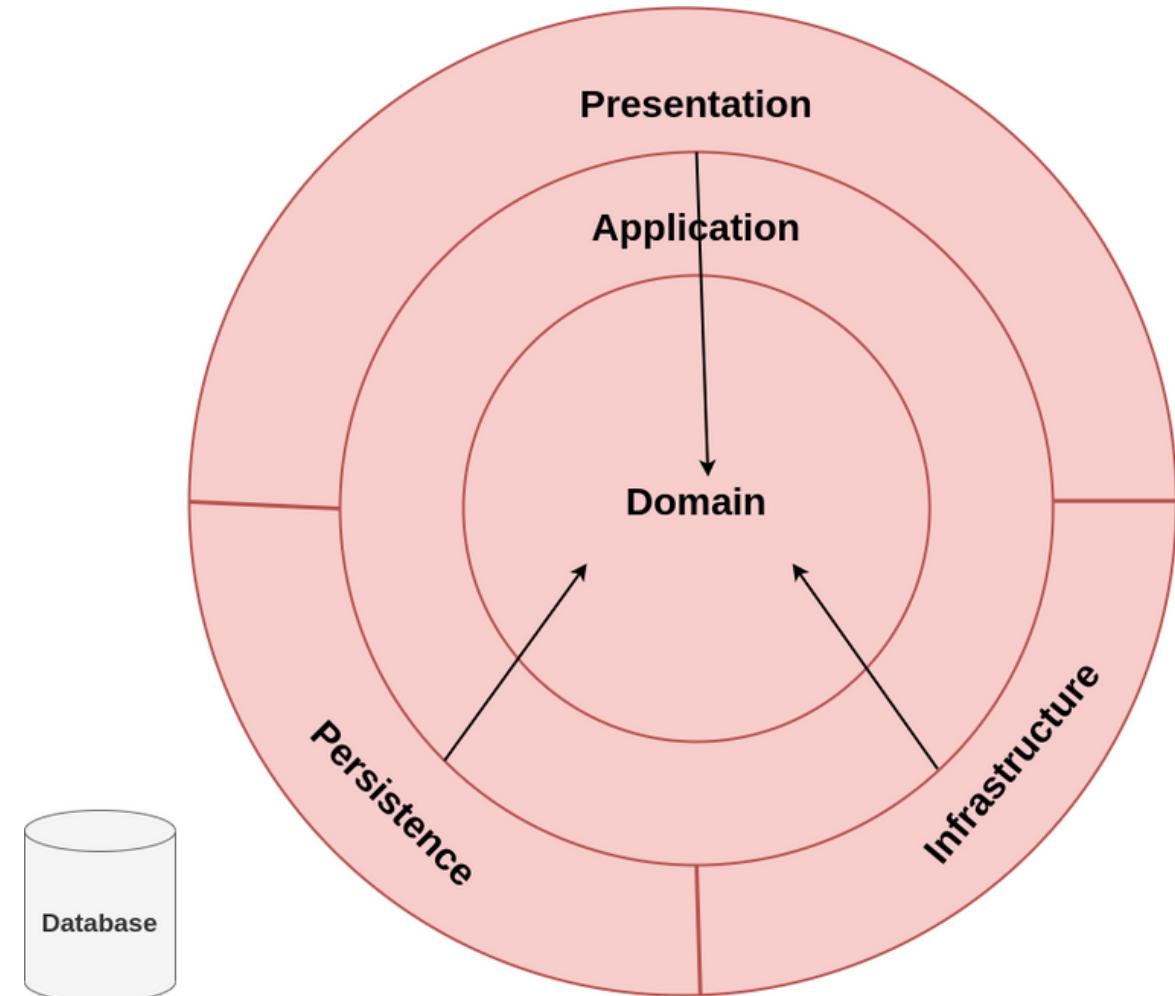
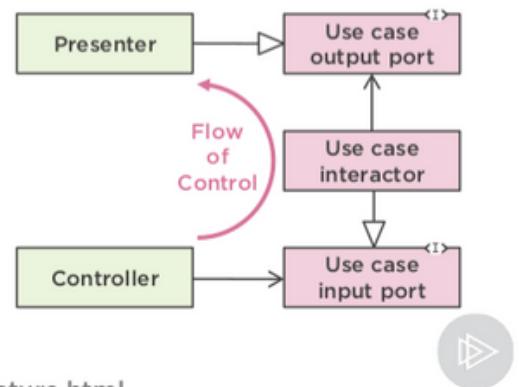


Couches architecturales

The Clean Architecture

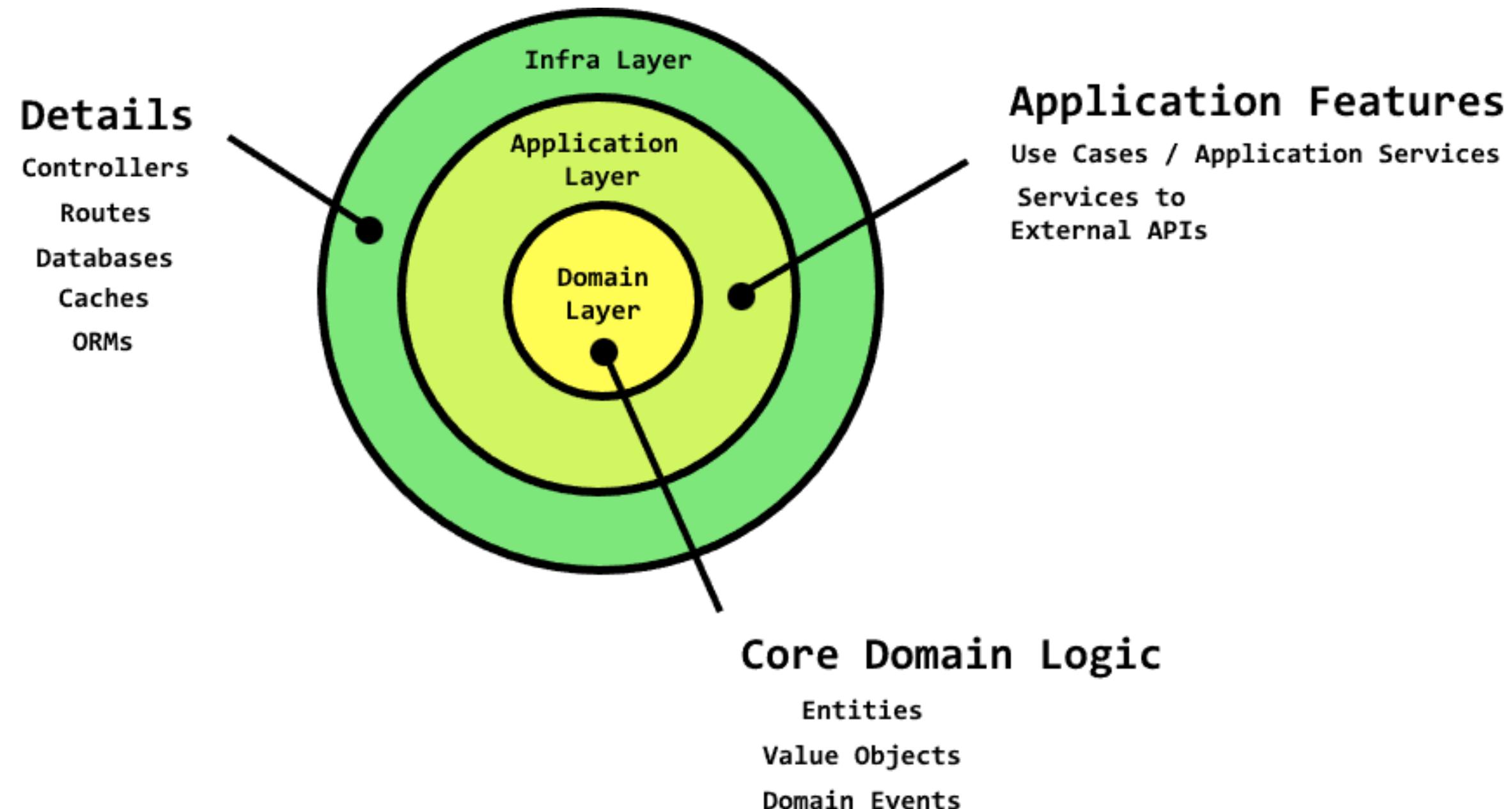


Original source: <http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html>

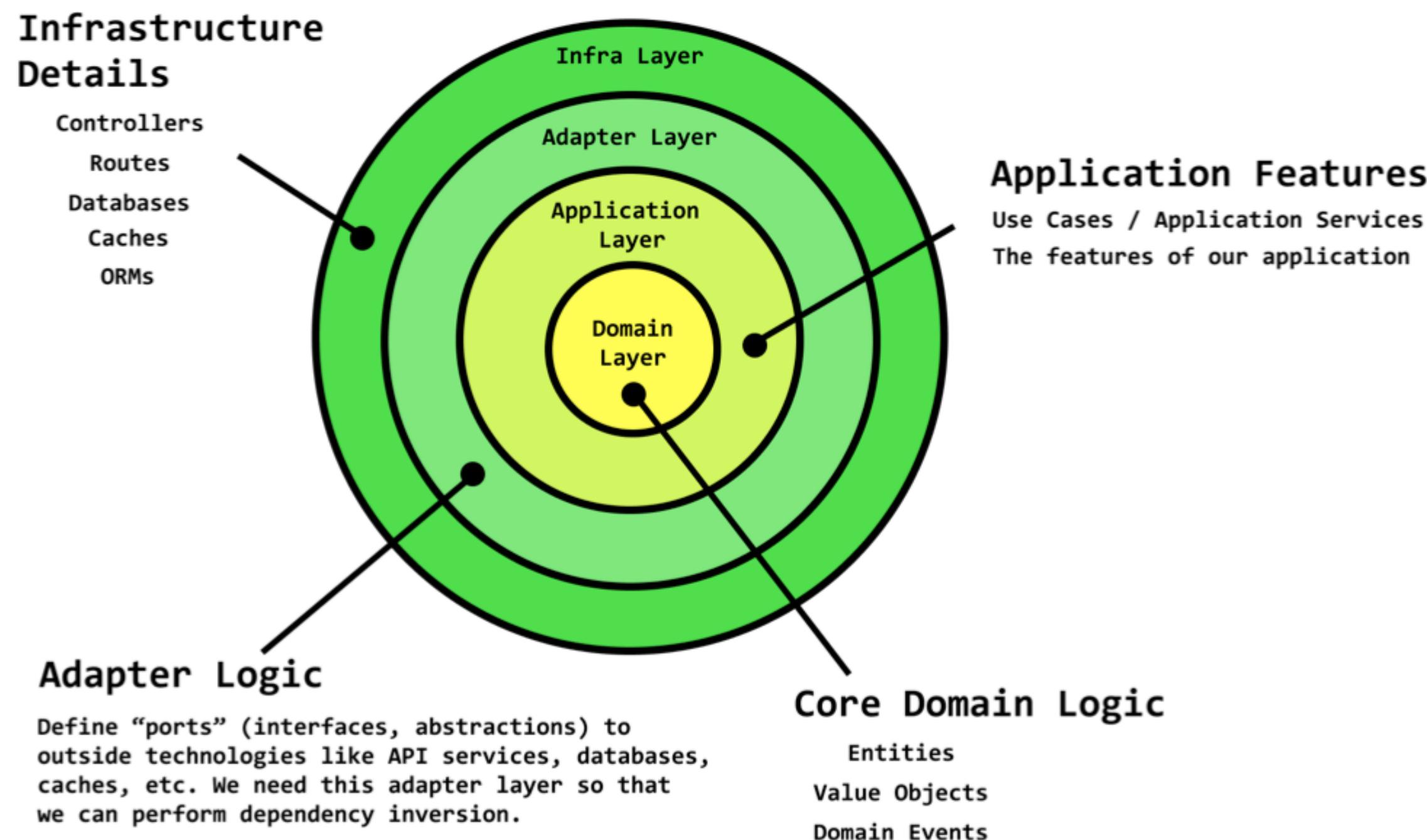


- La couche domaine, application et infrastructure
- La Dépendance Rurale

Couches de domaine



Structure propre

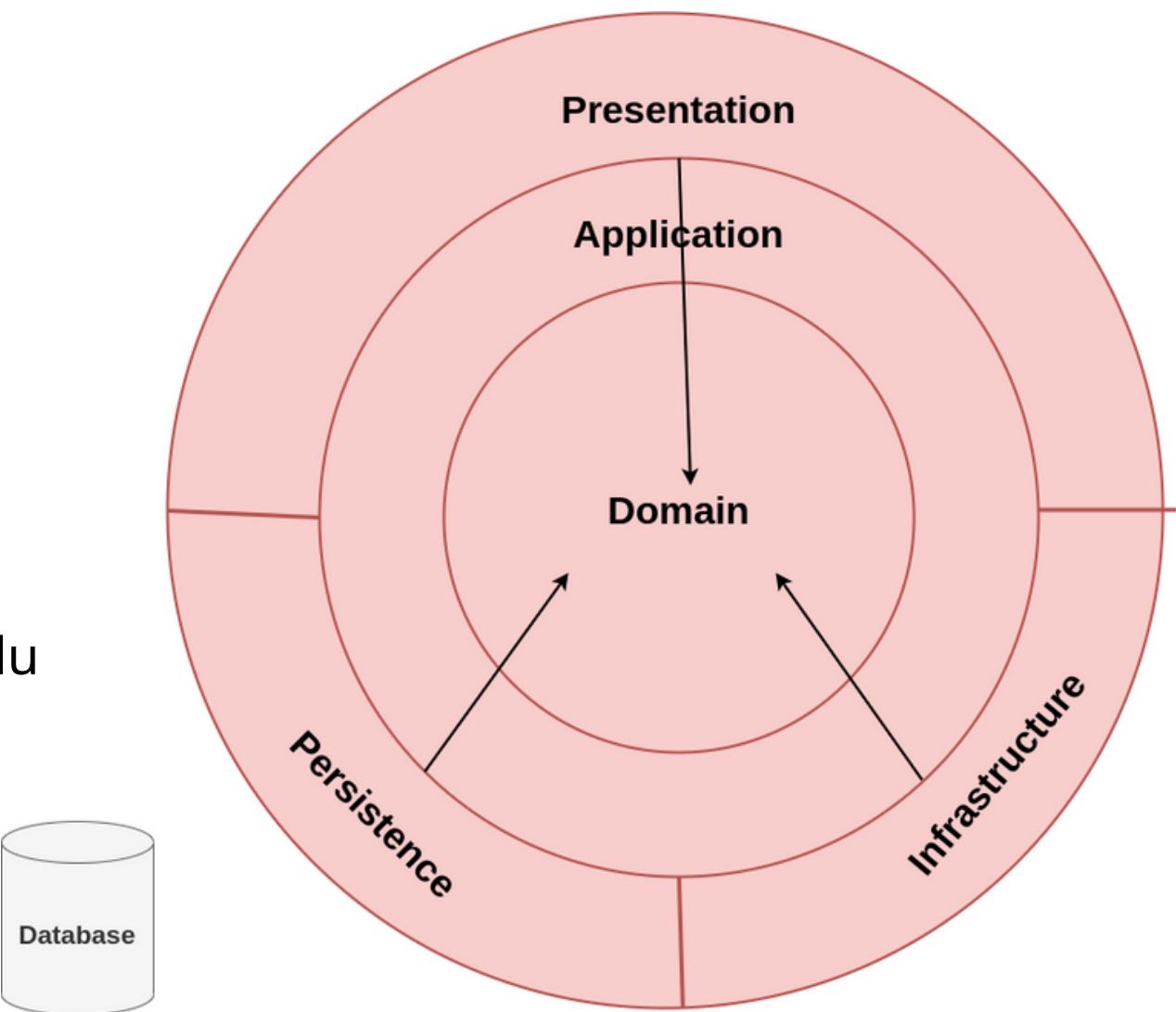


Couches de domaine

La couche de domaine est également une couche sans aucun code d'infrastructure. Il contient:

- Entités
- Objets de valeur
- Événements de domaine
- Interfaces du référentiel d'entités (modèle d'écriture)
- Services de domaine

Ces objets de domaine doivent être considérés comme des détails d'implémentation du couche d'application. En fait, la plupart de ces détails devraient rester derrière la couche application.



Couches de domaine

Implementing the Domain Layer

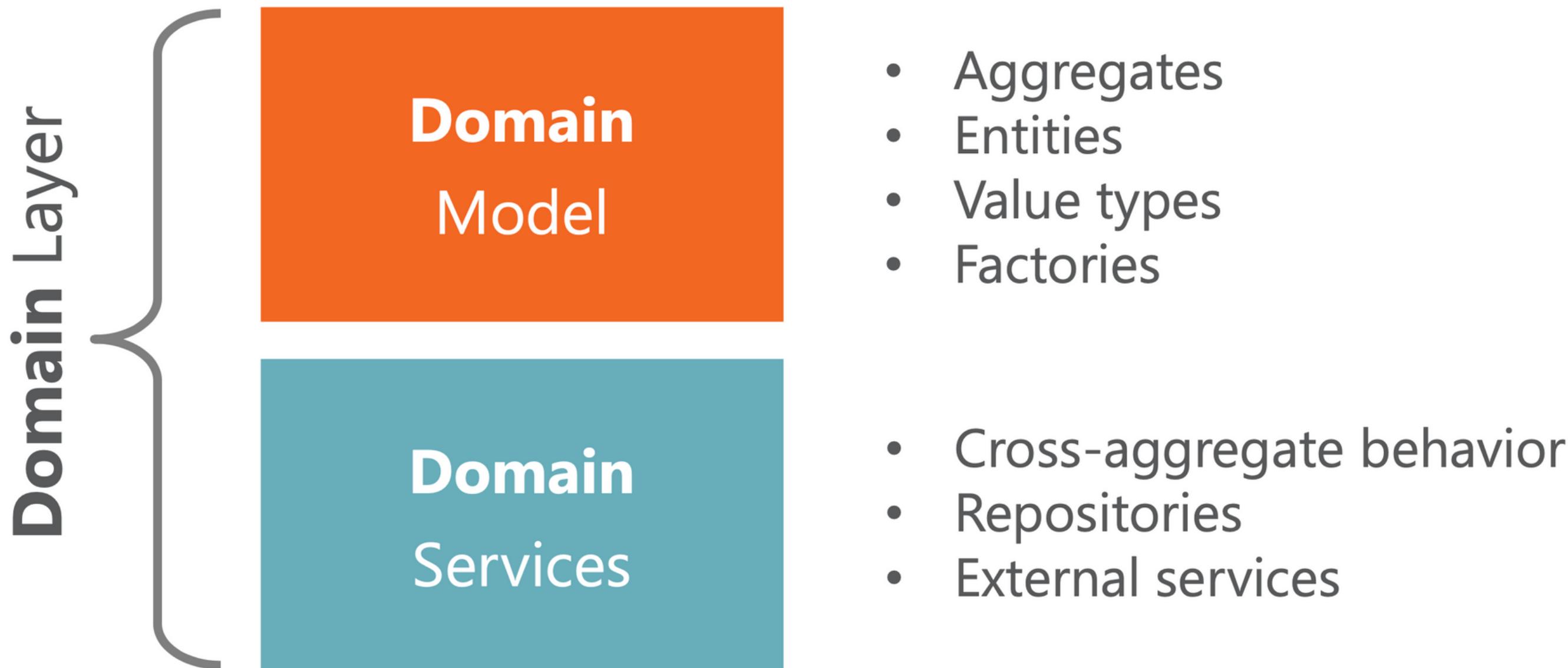
Classic Approach

Object-oriented Model

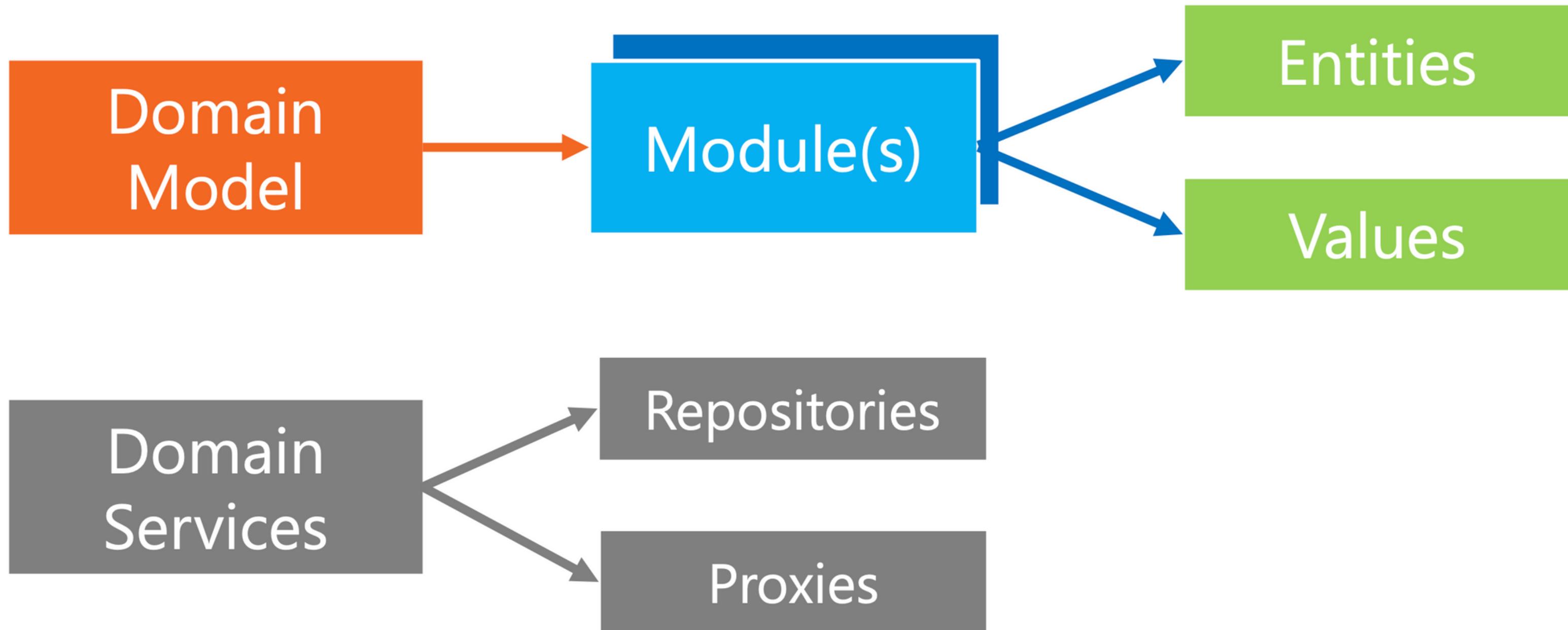
Domain Services

Couches de domaine

Domain Model Supporting Architecture



Domain Layer



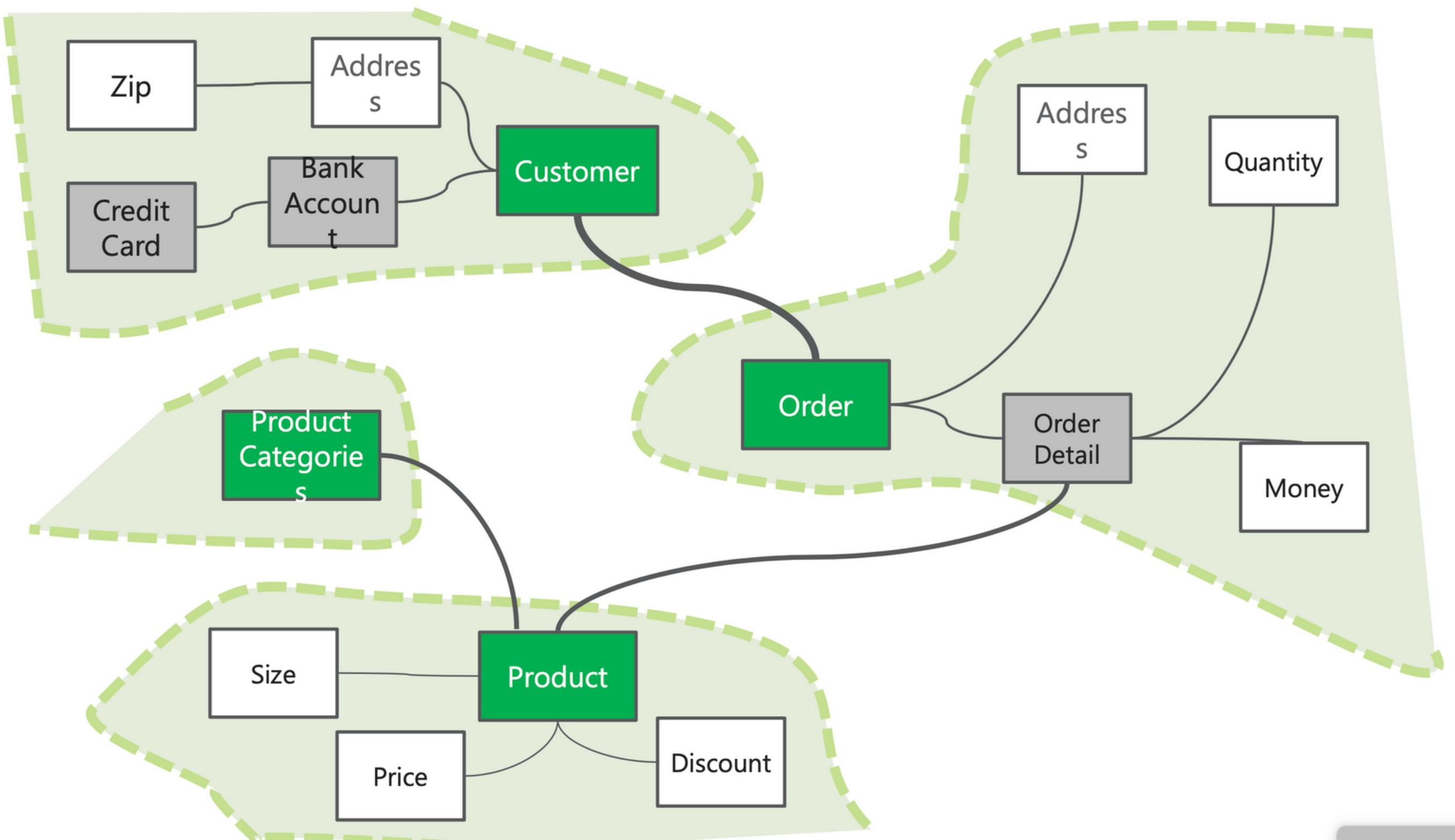
Aspects of a Domain Model Module

Value Objects

Entities

Aggregates

Couches de domaine

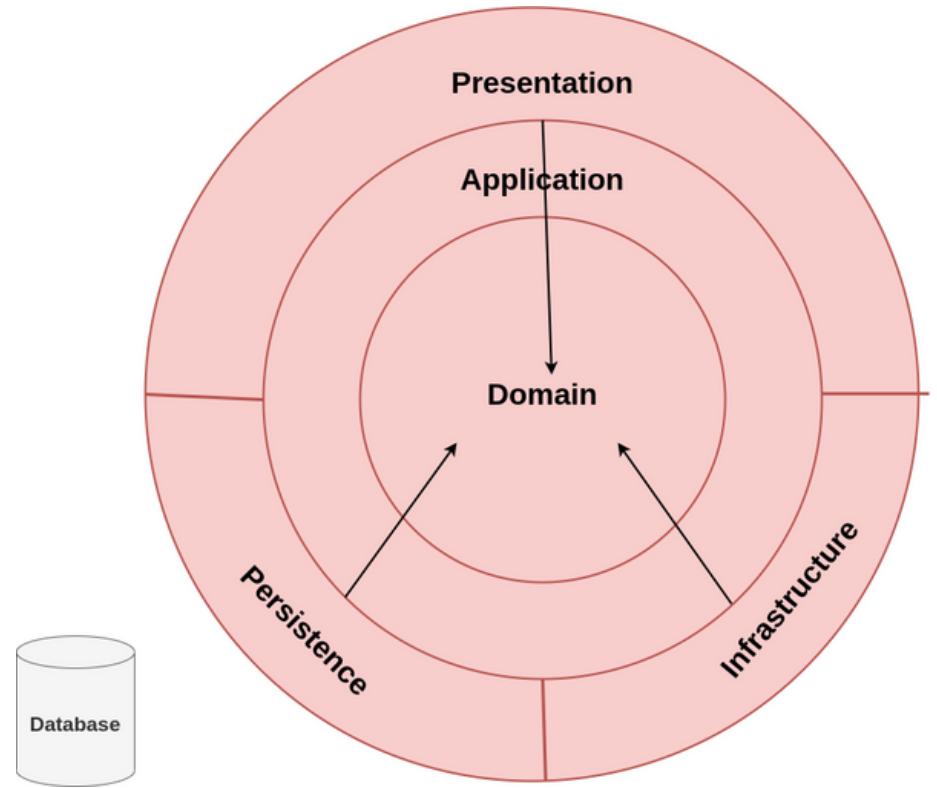


Couches d'infrastructure

Chaque morceau de code d'infrastructure que nous avons rencontré jusqu'à présent va finir par

dans la couche infrastructure. Ceci comprend:

- Contrôleurs Web
- Commandes CLI
- Implémentations de référentiels modèles d'écriture et de lecture
- Services qui se connectent à des systèmes externes, comme une API distante ou le système de fichiers
- Services qui utilisent l'heure actuelle ou génèrent des données aléatoires

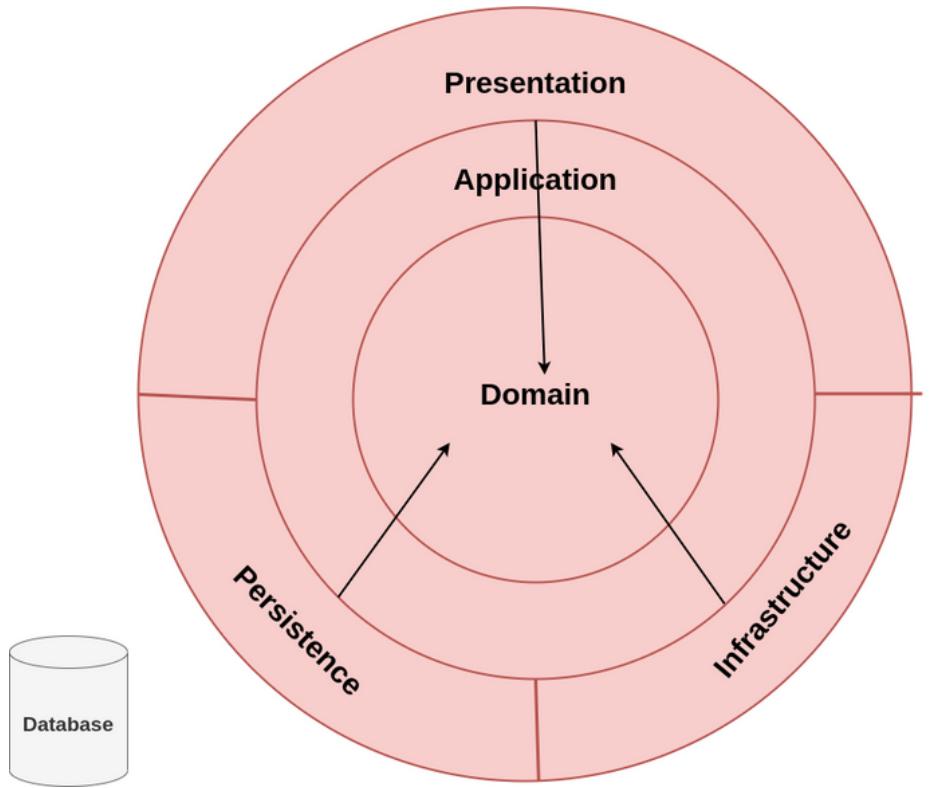


Couches d'infrastructure

Chaque morceau de code d'infrastructure que nous avons rencontré jusqu'à présent va finir par

dans la couche infrastructure. Ceci comprend:

- Contrôleurs Web
- Commandes CLI
- Implémentations de référentiels modèles d'écriture et de lecture
- Services qui se connectent à des systèmes externes, comme une API distante ou le système de fichiers
- Services qui utilisent l'heure actuelle ou génèrent des données aléatoires

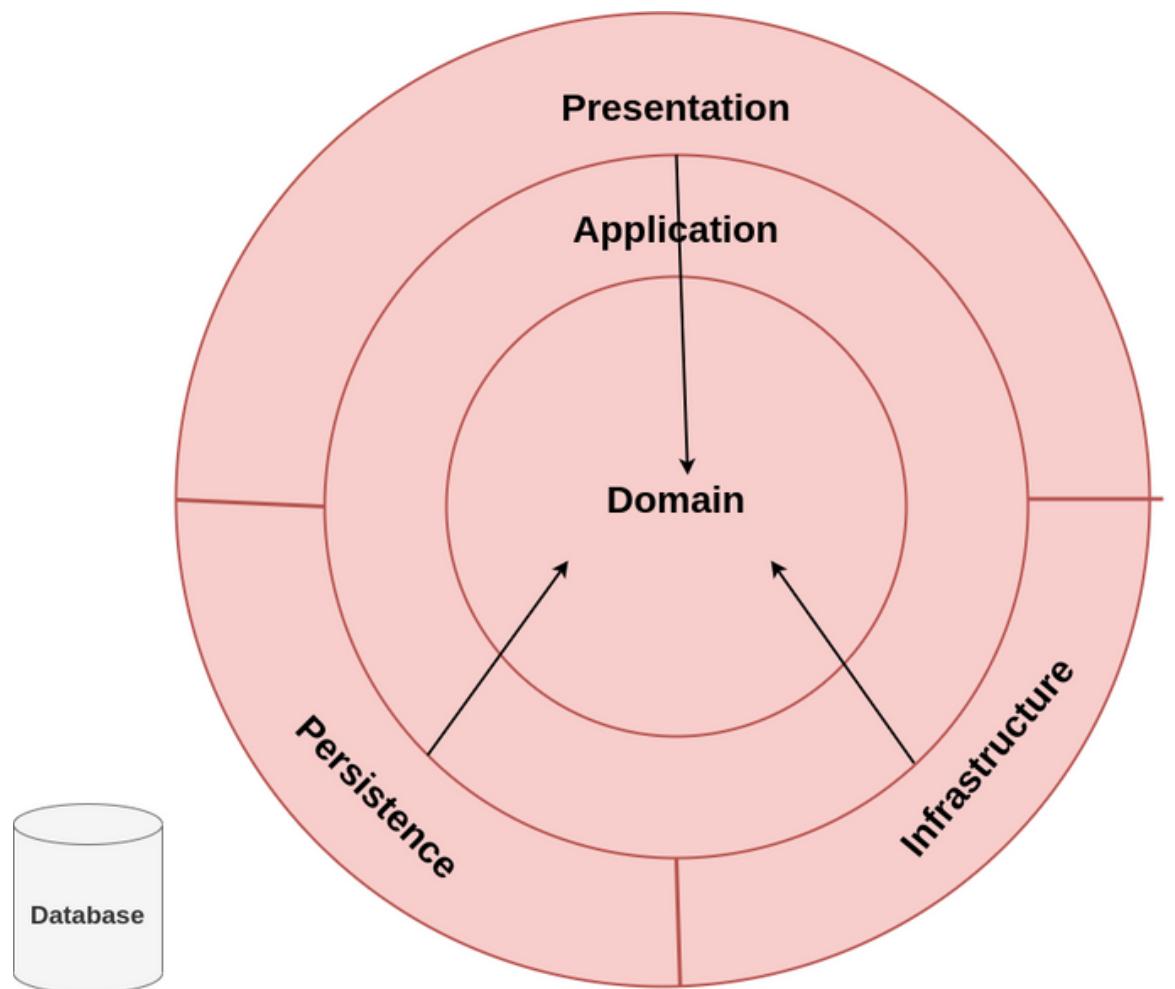


Couches d'application

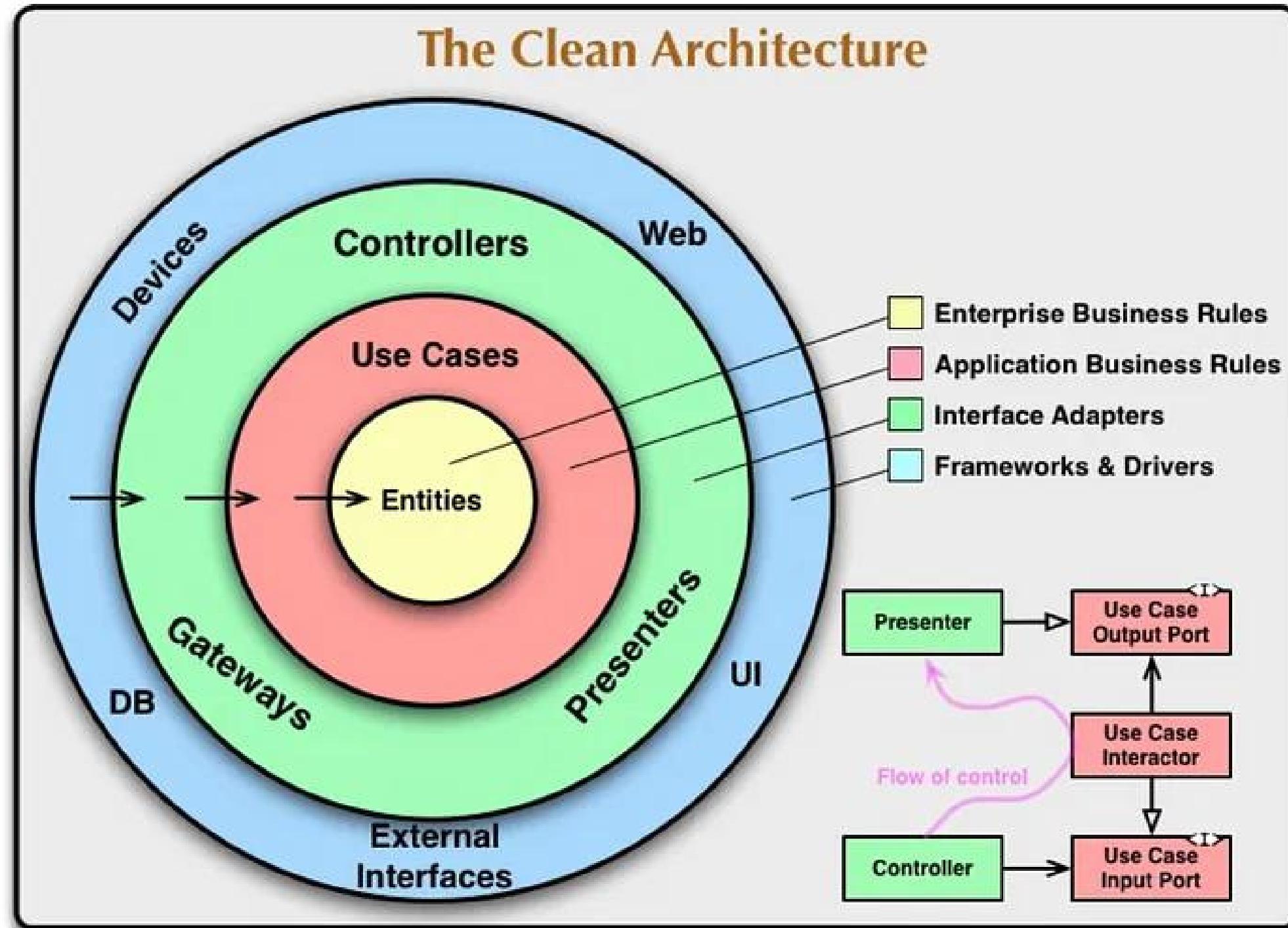
La couche application est la première couche exempte de code d'infrastructure. Ce

la couche comprend :

- Services d'application/gestionnaires de commandes et DTO de commandes
 - Afficher les interfaces du référentiel de modèles et afficher les DTO de modèles.
 - Abonnés aux événements qui écoutent les événements du domaine et effectuent des actions secondaires.
- Tâches
- Interfaces pour les services d'infrastructure



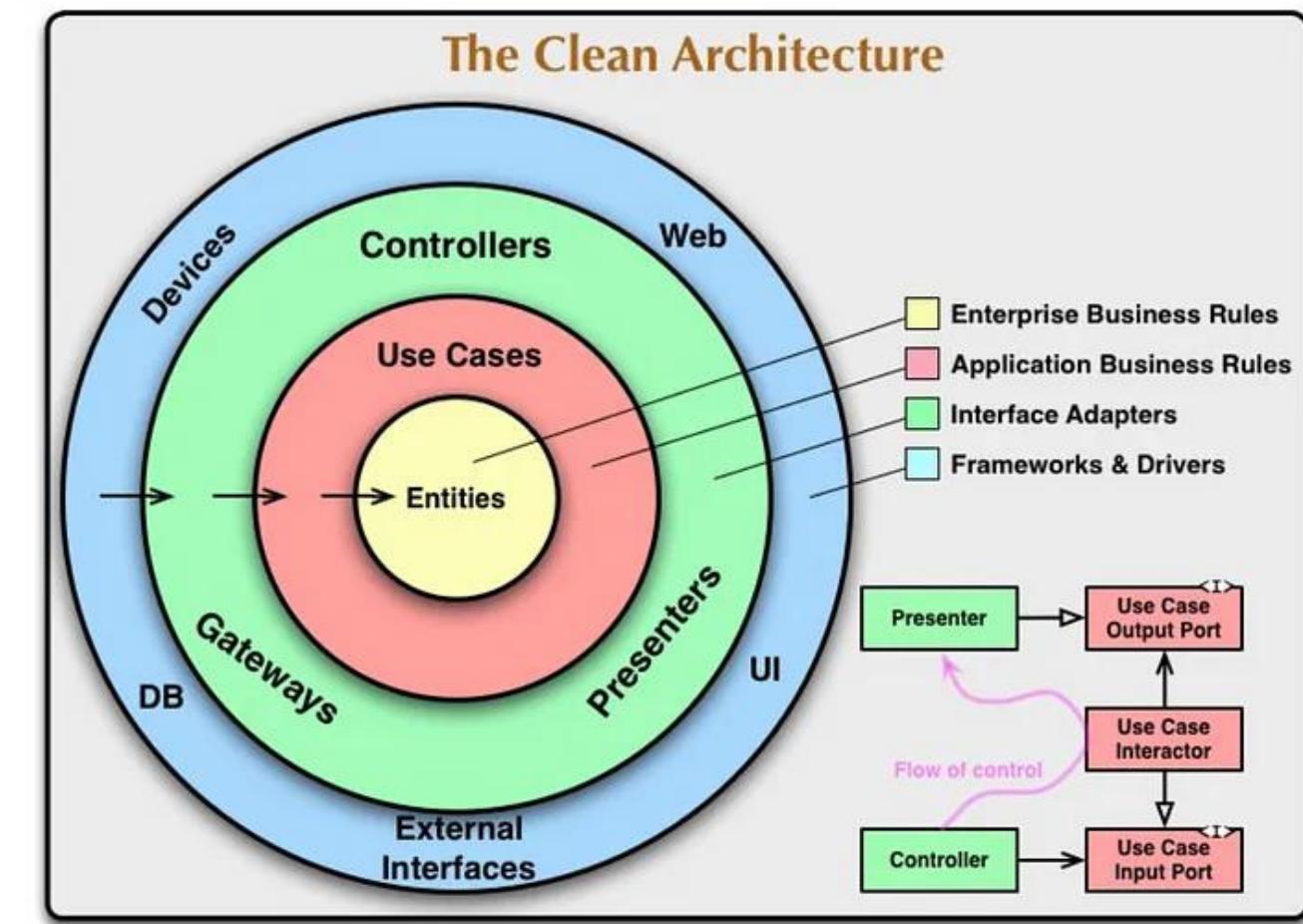
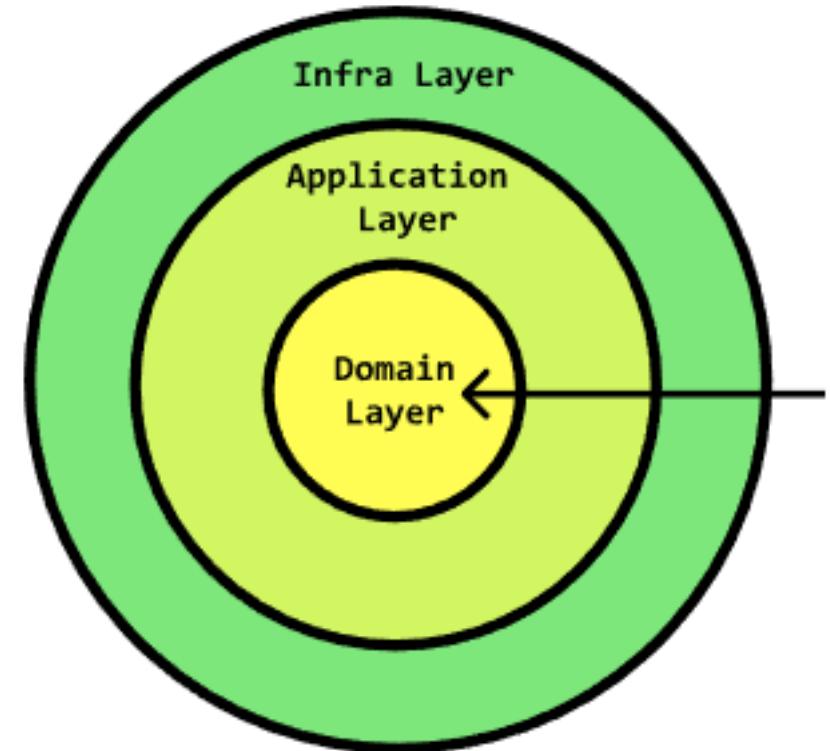
Architecture propre



Règle de dépendances

La règle de dépendance stipule que les dépendances du code source ne peuvent pointer que vers l'intérieur.

Cela signifie que rien dans un cercle intérieur ne peut rien savoir de quelque chose dans un cercle extérieur. c'est-à-dire que le cercle intérieur ne devrait dépendre de rien dans le cercle extérieur. Les flèches noires représentées dans le diagramme montrent la règle de dépendance.



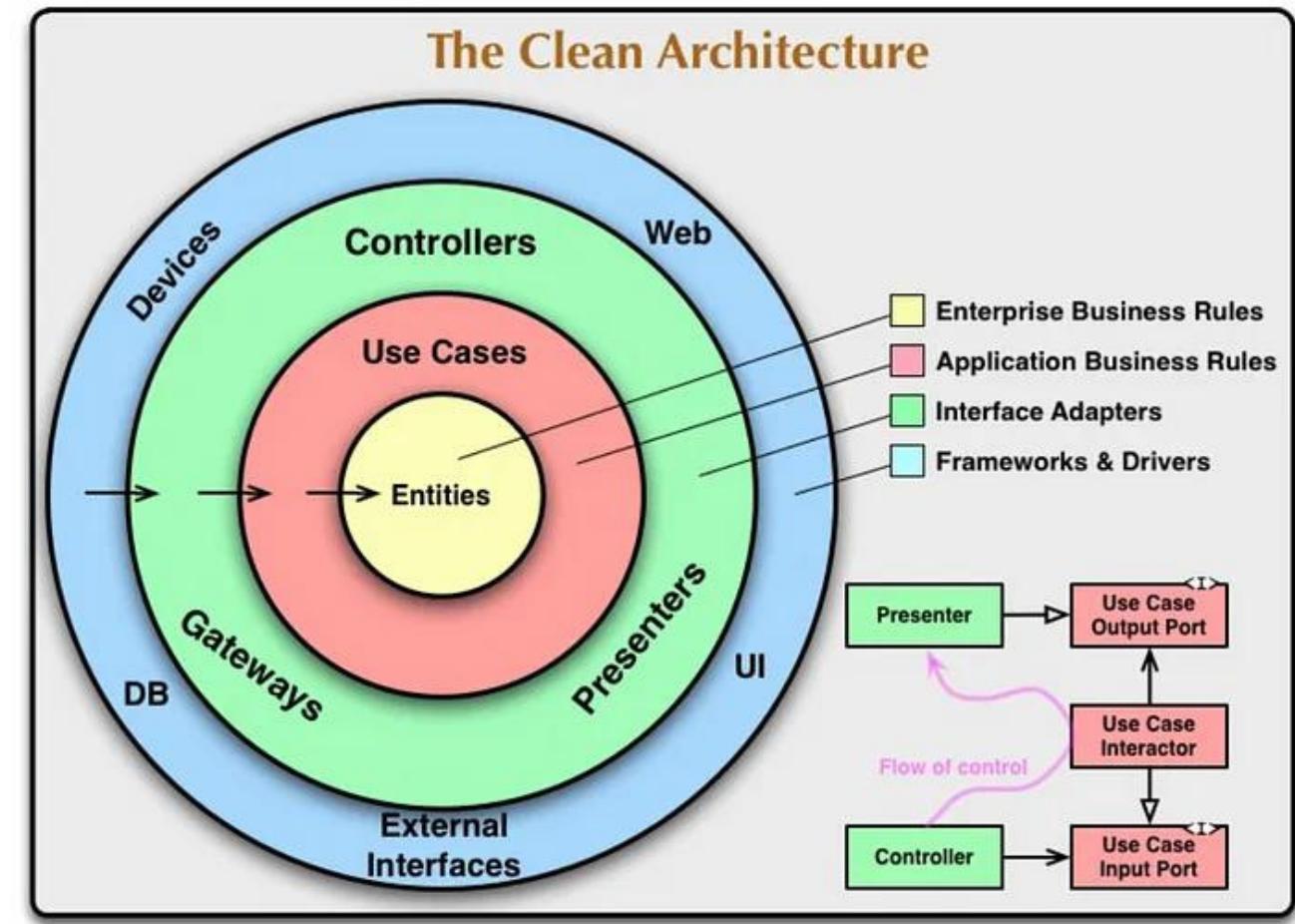
Cadres et lecteurs

Les zones logicielles qui résident à l'intérieur de cette couche sont

- Interface utilisateur
- Base de données
- Interfaces externes (ex : API plateforme native) Web (ex : Requête réseau)
- Appareils (par exemple : imprimantes et scanners)

Généralement, vous n'écrivez pas beaucoup de code dans cette couche, autre que le code de colle que le code de colle qui communique avec le cercle suivant vers l'intérieur.

La couche des cadres et des pilotes est l'endroit où se trouvent tous les détails. Le web est un détail. La base de données est un détail. Nous gardons ces choses à l'extérieur où elles peuvent faire peu de mal.

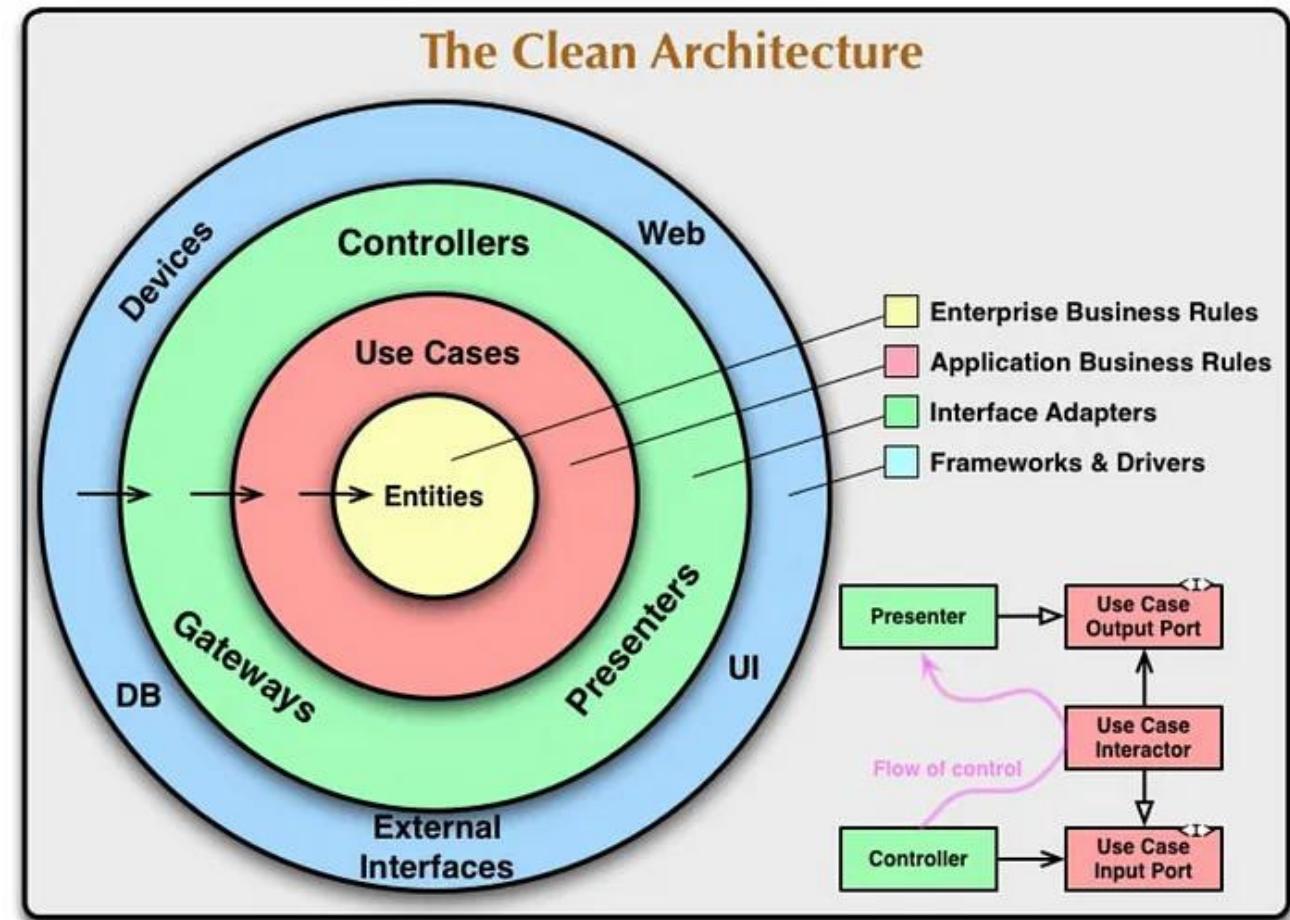


Adaptateurs d'interface

Cette couche contient

- Présentateurs (UI Logic, États)
- Contrôleurs (interface contenant les méthodes nécessaires à l'application implémentée par le Web, les périphériques ou les interfaces externes)
- Passerelles (interface qui contient chaque opération CRUD effectuée par l'application, implémentée par DB)

Le logiciel de la couche des adaptateurs d'interface est un ensemble d'adaptateurs qui convertissent les données de la manière la plus pratique pour un organisme externe tel que la base de données ou le Web.



Entités

Les entités encapsulent les règles métier critiques à l'échelle de l'entreprise. Une entité peut être un objet avec des méthodes, ou un ensemble de structures de données et de fonctions.

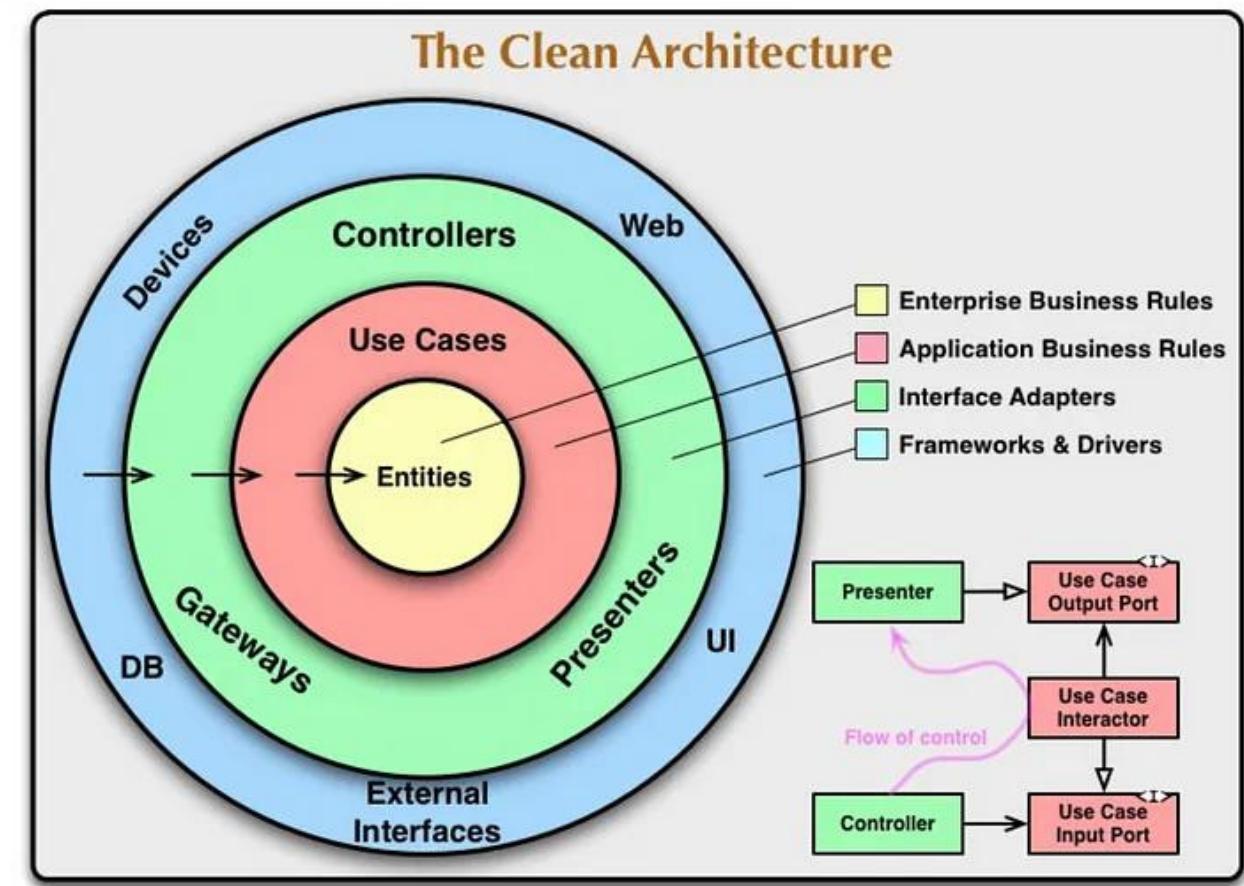
Règles commerciales d'entreprise

Il s'agit de la couche qui contient les règles métier de base ou les règles métier spécifiques à un domaine. De plus, cette couche est la moins susceptible de changer.

Les modifications apportées à une couche externe n'affectent pas cette couche. Étant donné que les règles métier ne changent pas souvent, les changements dans cette couche sont très rares.

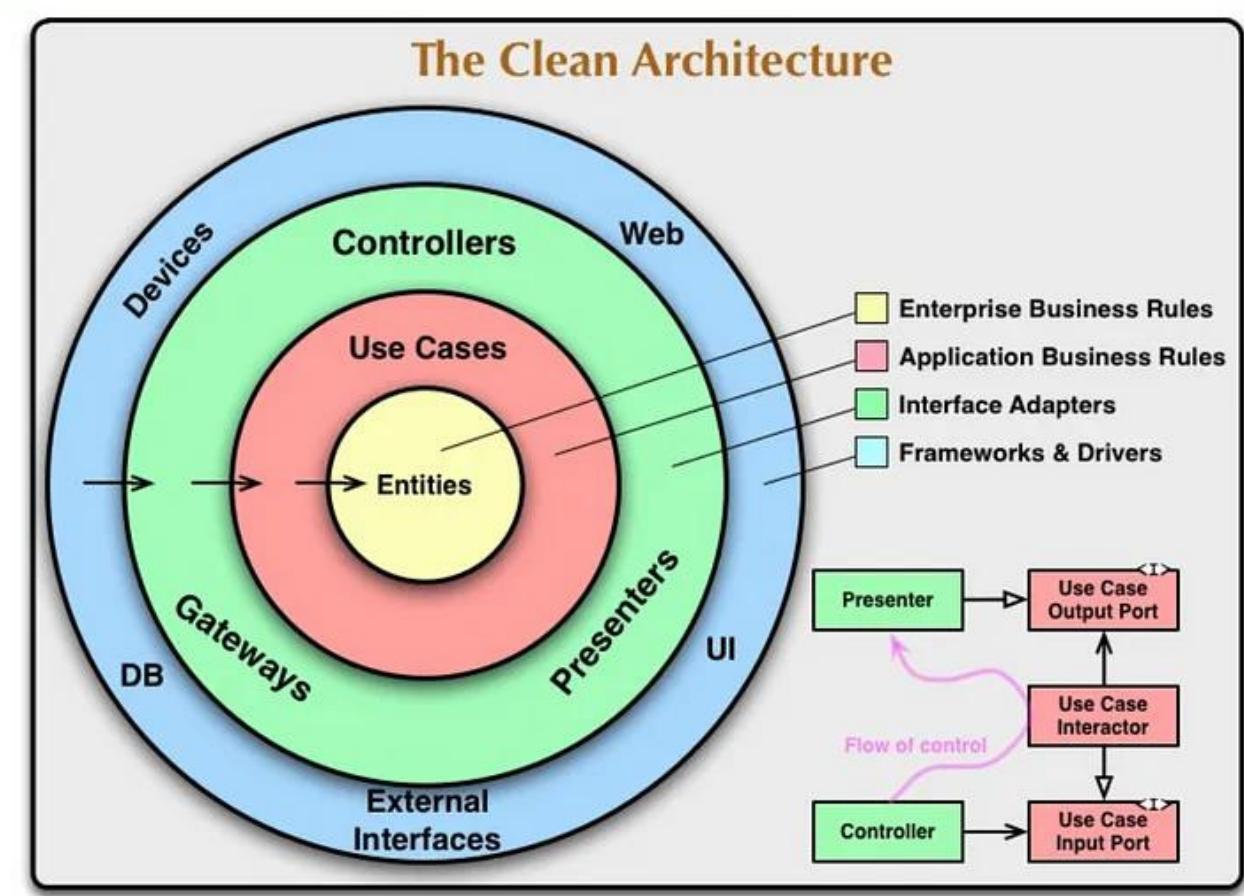
Cette couche contient les entités.

Une entité peut être soit une structure de données de base nécessaire aux règles métier, soit un objet contenant des méthodes contenant une logique métier.



CAS D'UTILISATION

Le logiciel de la couche des cas d'utilisation contient des règles métier spécifiques à l'application. Il encapsule et implémente tous les cas d'utilisation du système. Ces cas d'utilisation orchestrent le flux de données vers et depuis les entités et demandent à ces entités d'utiliser leurs règles métier critiques pour atteindre les objectifs des utilisations.



DÉMO