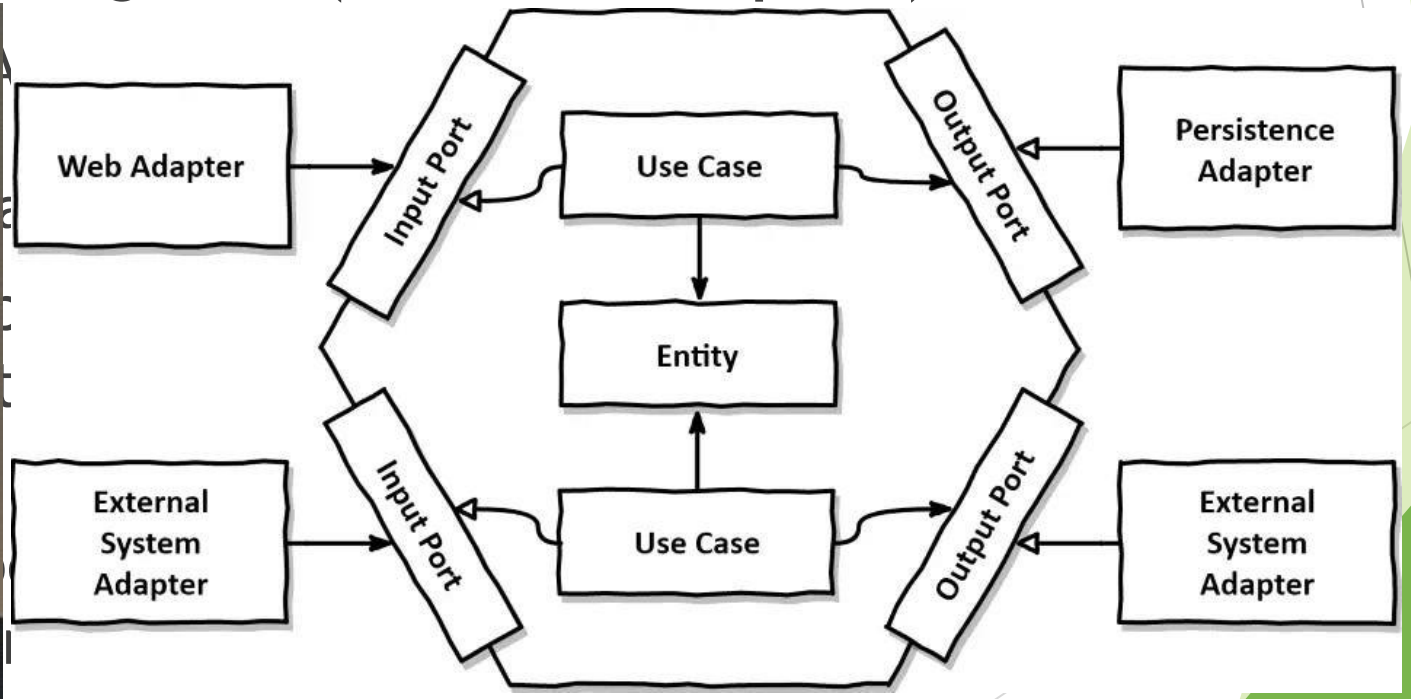
Chapitre 5
Clean Architecture

Contexte

- Au cours des dernières décennies, un large éventail d'idées relatives à l'architecture des systèmes a émergé. Parmi celles-ci, on peut citer :
- **L'architecture hexagonale (Ports and Adapters):**



Contexte

- **Le paradigme DCI (Data-Context-Interaction):**
  - un modèle d'architecture logicielle proposé par Trygve Reenskaug (également à l'origine de MVC).
  - vise à mieux représenter le comportement du système tel qu'il est vécu par l'utilisateur, en séparant clairement les données, le contexte d'exécution, et les interactions.
- Le **modèle BCE (Boundary-Control-Entity):**
  - présenté par Ivar Jacobson dans son ouvrage Object-Oriented Software Engineering: A Use Case Driven Approach.
  - issu de l'analyse orientée objet, dans le cadre de la méthode OOSE.
  - Il permet de structurer un système autour des cas d'utilisation en séparant clairement les responsabilités

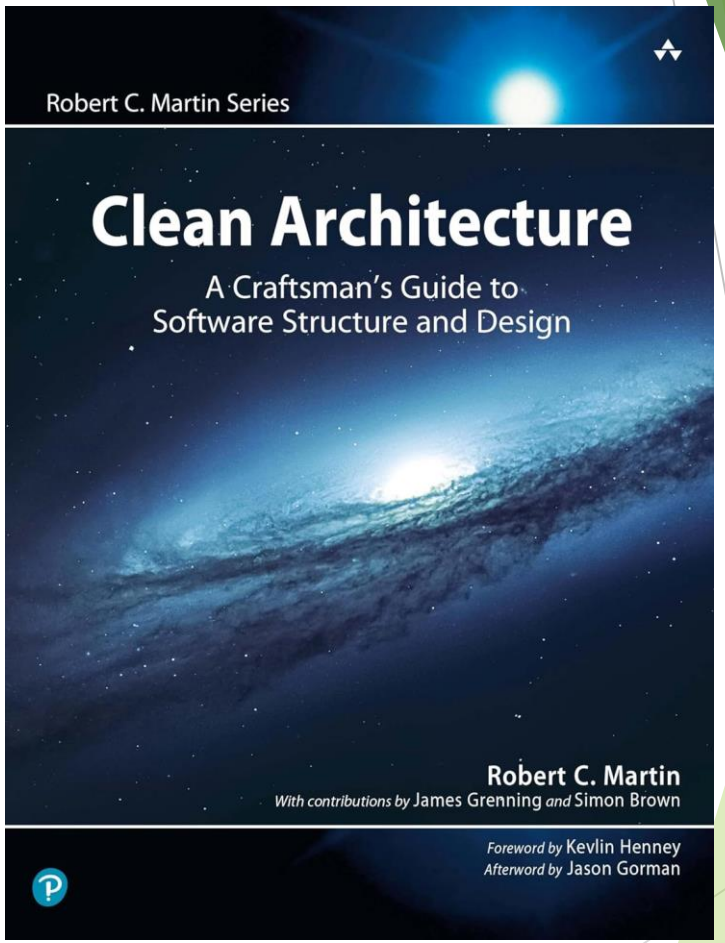
- Bien que ces architectures présentent certaines différences dans leurs modalités de mise en œuvre, elles demeurent fondamentalement proches.
- Elles poursuivent toutes un objectif commun, à savoir la **séparation des préoccupations (separation of concerns)**.
- Cette séparation est réalisée par une structuration du logiciel en couches distinctes.
- Chacune de ces approches comporte au minimum une couche dédiée aux règles métier, ainsi qu'une autre consacrée aux interfaces utilisateur et aux interactions avec les systèmes externes.

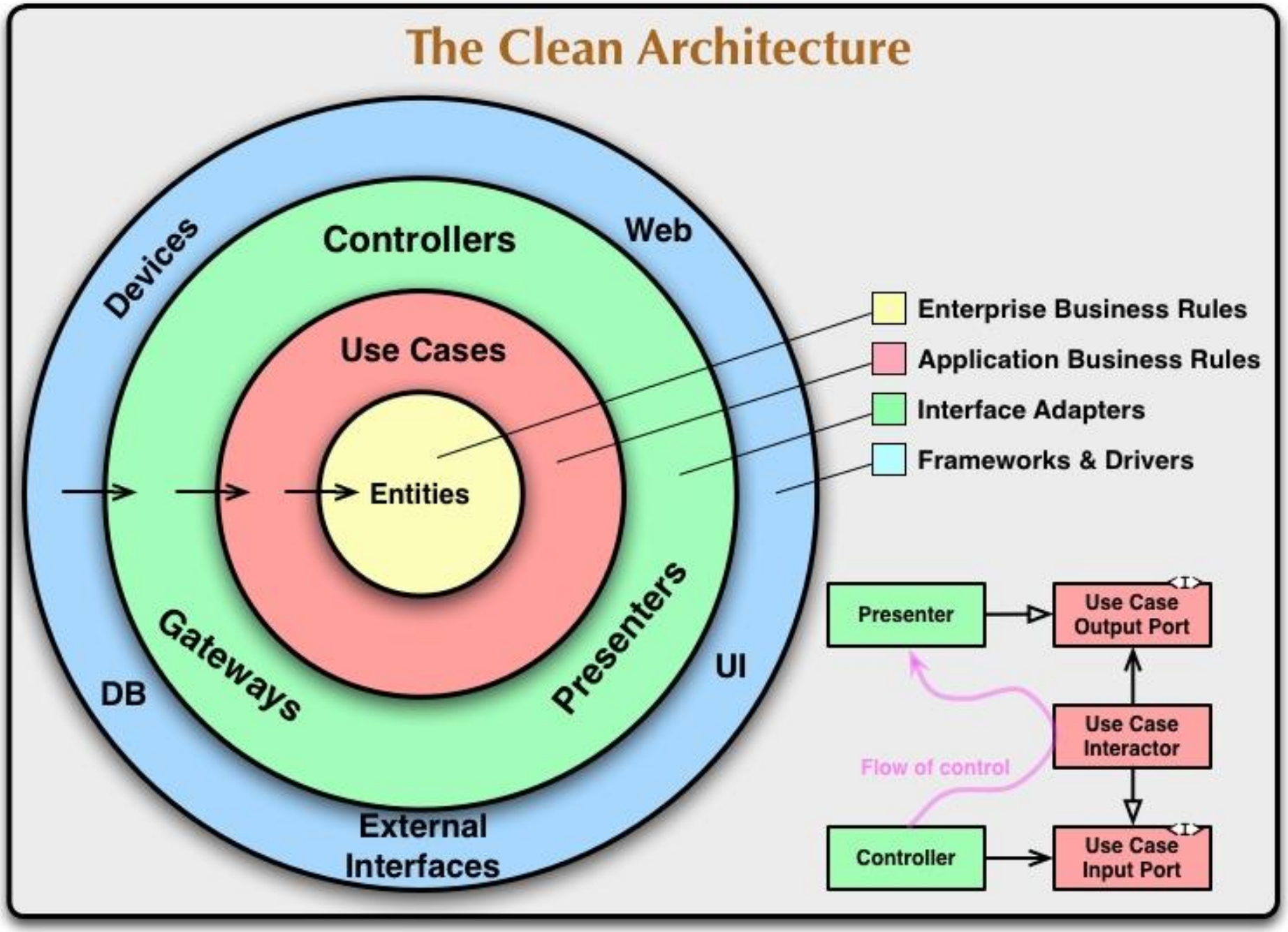
- Chacune de ces architectures conduit à la conception de systèmes présentant les caractéristiques suivantes :
- **Indépendance vis-à-vis des frameworks :** l'architecture ne dépend pas de l'existence d'une bibliothèque logicielle riche en fonctionnalités. Les frameworks peuvent ainsi être utilisés comme de simples outils, sans contraindre la conception du système à leurs limites.
- **Testabilité :** les règles métier peuvent être testées de manière isolée, indépendamment de l'interface utilisateur, de la base de données, du serveur web ou de tout autre élément externe.
- **Indépendance vis-à-vis de l'interface utilisateur :** l'interface utilisateur peut évoluer ou être remplacée sans impact sur le reste du système.
  - Par exemple, une interface web peut être substituée par une interface en ligne de commande sans modifier les règles métier.

- **Indépendance vis-à-vis de la base de données :** il est possible de remplacer un système de gestion de base de données (tel qu'Oracle ou SQL Server) par une autre solution (comme MongoDB ou BigTable) sans affecter les règles métier, celles-ci n'étant pas couplées à la technologie de stockage.
- **Indépendance vis-à-vis des systèmes externes :** les règles métier sont totalement découplées des interactions avec le monde extérieur et ne dépendent d'aucune interface externe.
- La **clean architecture** constitue une tentative d'intégration de ces différentes architectures au sein d'un cadre conceptuel unifié et opérationnel.

La clean architecture

- La **Clean Architecture** est un style d'architecture logicielle proposé par Robert C. Martin, qui vise à construire des systèmes **maintenables, testables et indépendants des technologies** (frameworks, base de données, UI…).
- La Clean Architecture repose sur l'idée de séparer les préoccupations en organisant le code en couches distinctes, chacune ayant un rôle et une responsabilité claire.
- Cela permet de minimiser les dépendances directes entre les différentes parties du système, ce qui rend le code plus facile à tester, à comprendre et à modifier.


Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to
Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman


The Clean Architecture
Devices
Web
Controllers
Use Cases
Entities
Gateways
Presenters
DB
UI
External Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case Output Port
<I>
Use Case Interactor
Flow of control
Controller
Use Case Input Port
<I>

- Les cercles concentriques représentent différentes zones du logiciel.
- De manière générale, plus on se rapproche du centre, plus le niveau d'abstraction du logiciel est élevé.
- Les cercles externes correspondent aux mécanismes, tandis que les cercles internes représentent les politiques.
- La règle fondamentale qui assure le bon fonctionnement de cette architecture est la suivante :
- **Règle de dépendance :** Les dépendances du code source doivent être orientées uniquement vers l'intérieur, c'est-à-dire en direction des politiques de plus haut niveau.

- Ainsi, aucun élément appartenant à un cercle interne ne doit avoir connaissance d'un élément situé dans un cercle externe.
- En particulier, le code des couches internes ne doit faire référence à aucun élément déclaré dans les couches externes, qu'il s'agisse de fonctions, de classes, de variables ou de toute autre entité logicielle nommée.
- De même, les formats de données définis dans les couches externes ne doivent pas être utilisés dans les couches internes, en particulier lorsqu'ils sont générés par des frameworks situés dans ces couches externes.
- L'objectif est d'empêcher toute influence des couches externes sur les couches internes.

Entités

- Les entités encapsulent les règles métier critiques à l'échelle de l'entreprise.
- Une entité peut prendre la forme d'un objet doté de méthodes, ou d'un ensemble de structures de données et de fonctions. L'essentiel est qu'elle puisse être réutilisée par différentes applications au sein de l'organisation.
- Dans le cas où il ne s'agit pas d'un système d'entreprise mais d'une application unique, ces entités correspondent aux objets métier de l'application.
- Elles encapsulent les règles les plus générales et les plus abstraites.
- Par conséquent, elles sont les moins susceptibles d'être affectées par des changements externes.
- À titre d'exemple, une modification de la navigation des pages ou des mécanismes de sécurité ne devrait pas impacter ces objets.
- Plus largement, aucune évolution opérationnelle propre à une application particulière ne devrait affecter la couche des entités.

Cas d’utilisation

- La couche des cas d’utilisation regroupe les règles métier spécifiques à l’application.
- Elle encapsule et implémente l’ensemble des cas d’utilisation du système.
- Ces derniers orchestrent les flux de données entre les entités et le reste du système, tout en sollicitant les règles métier critiques portées par les entités afin d’atteindre les objectifs fonctionnels.
- Les modifications apportées à cette couche ne sont pas censées impacter les entités.
- De même, elle ne devrait pas être affectée par des changements liés aux éléments externes tels que la base de données, l’interface utilisateur ou les frameworks.
- La couche des cas d’utilisation est ainsi isolée de ces préoccupations techniques.
- En revanche, toute évolution dans le fonctionnement de l’application est susceptible d’affecter les cas d’utilisation et, par conséquent, le code de cette couche.
- Ainsi, toute modification dans les exigences ou les scénarios d’usage entraîne nécessairement des adaptations au niveau de cette couche.

Adaptateurs d’interface

- La couche des adaptateurs d’interface regroupe un ensemble de composants chargés de convertir les données depuis un format adapté aux cas d’utilisation et aux entités vers un format approprié pour les systèmes externes, tels que la base de données ou les interfaces web.
- C’est notamment dans cette couche que s’inscrit l’architecture MVC d’une interface graphique.
- Les composants tels que les présentateurs, les vues et les contrôleurs y sont localisés.
- Les modèles, quant à eux, correspondent généralement à de simples structures de données échangées entre les contrôleurs et les cas d’utilisation, puis restituées aux présentateurs et aux vues.
- De manière analogue, cette couche assure la transformation des données depuis leur forme interne — adaptée aux entités et aux cas d’utilisation — vers une forme compatible avec les mécanismes de persistance, notamment les frameworks d’accès aux bases de données.

Adaptateurs d’interface

- Aucun élément situé dans les couches internes ne doit avoir connaissance de la base de données.
- Ainsi, si celle-ci repose sur un modèle relationnel, tout le code SQL doit être strictement confiné à cette couche, en particulier dans les composants dédiés à la persistance.
- Enfin, cette couche inclut également tout autre adaptateur nécessaire pour convertir des données issues de sources externes (par exemple, des services tiers) vers le format interne utilisé par les cas d’utilisation et les entités.

Frameworks et pilotes

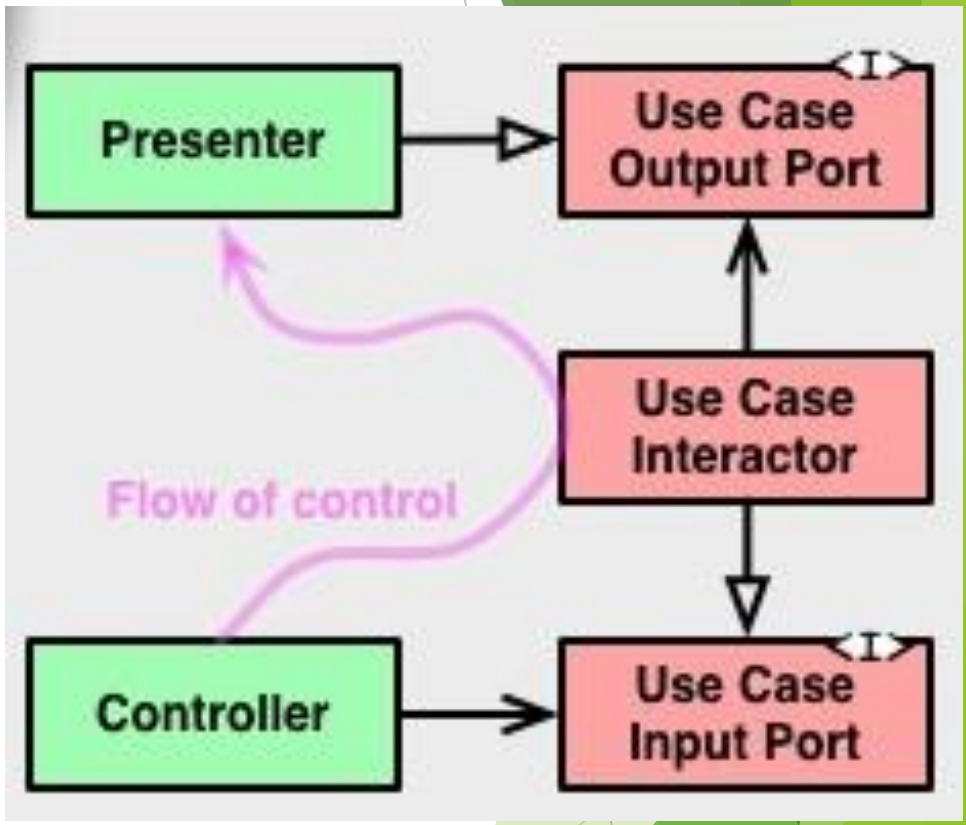
- La couche la plus externe du modèle est généralement constituée de frameworks et d'outils tels que les systèmes de gestion de bases de données ou les frameworks web.
- En règle générale, cette couche contient peu de code métier ; elle se limite essentiellement à du code d'intégration (« glue code ») assurant la communication avec la couche immédiatement interne.
- Cette couche regroupe l'ensemble des aspects techniques considérés comme des détails d'implémentation.
- Le web, par exemple, constitue un détail, tout comme la base de données. Ces éléments sont volontairement placés à la périphérie de l'architecture afin de minimiser leur impact sur le cœur du système.

Uniquement quatre cercles ?

- Les cercles représentés ont une vocation schématique : il est tout à fait possible que davantage de couches soient nécessaires dans une application réelle.
- Aucune règle n'impose de se limiter strictement à ces quatre niveaux.
- Cependant, la règle de dépendance demeure toujours applicable : les dépendances du code source doivent être orientées vers l'intérieur.
- À mesure que l'on progresse vers le centre, le niveau d'abstraction et de politique augmente.
- Le cercle le plus externe est constitué de détails concrets de bas niveau.
- En se rapprochant du centre, le logiciel devient progressivement plus abstrait et intègre des règles métier de plus haut niveau.
- Le cercle le plus interne correspond ainsi au niveau d'abstraction le plus élevé et le plus général du système.

Franchissement des frontières

- Dans la partie inférieure droite de la figure, un exemple illustre la manière dont les frontières entre les couches sont franchies.
- On y observe les contrôleurs et les présentateurs interagissant avec les cas d'utilisation situés dans la couche suivante.
- Il convient de noter le sens du flux de contrôle : celui-ci débute dans le contrôleur, traverse le cas d'utilisation, puis se poursuit jusqu'à l'exécution dans le présentateur.
- En parallèle, les dépendances du code source sont orientées vers l'intérieur, en direction des cas d'utilisation.
- Cette apparente contradiction est généralement résolue par l'application du principe d'inversion des dépendances (Dependency Inversion Principle).
- Dans un langage tel que Java, par exemple, on organise les interfaces et les relations d'héritage de manière à ce que les dépendances du code source soient opposées au flux de contrôle, précisément aux points de franchissement des frontières.



Franchissement des frontières

- Ainsi, si un cas d'utilisation doit invoquer un présentateur, cet appel ne peut être direct, car cela violerait la règle de dépendance : aucun élément d'une couche externe ne doit être référencé par une couche interne.
- Dans ce cas, le cas d'utilisation dépend d'une interface située dans la couche interne (appelée, dans le schéma, « port de sortie du cas d'utilisation »), tandis que le présentateur, situé dans la couche externe, implémente cette interface.
- Cette même technique est utilisée pour franchir toutes les frontières dans ces architectures.
- Le polymorphisme dynamique est exploité afin d'inverser les dépendances du code source par rapport au flux de contrôle, garantissant ainsi le respect de la règle de dépendance, quelle que soit la direction du flux d'exécution.

Quels types de données traversent les frontières

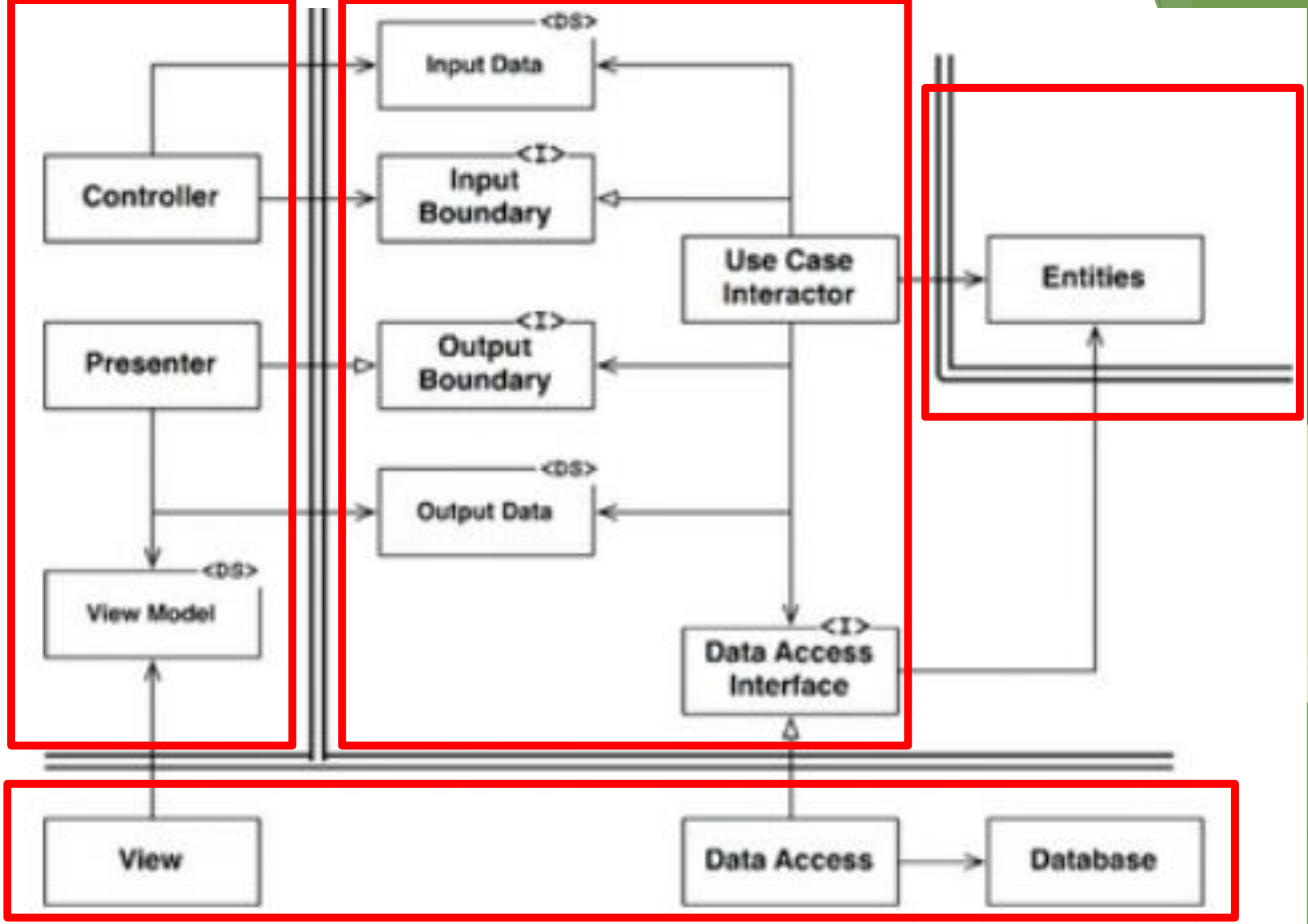
- En général, les données qui traversent les frontières entre les couches prennent la forme de structures de données simples.
- Il peut s'agir de structures élémentaires (structs), d'objets de transfert de données (DTO), d'arguments de fonctions, ou encore de collections génériques telles que des tables de hachage, voire d'objets construits spécifiquement pour cet échange.
- L'élément essentiel est que ces données restent simples et isolées lorsqu'elles sont transmises entre les couches.
- Il est important d'éviter toute forme de contournement consistant à transmettre des objets métier de type entité ou des lignes issues directement de la base de données.

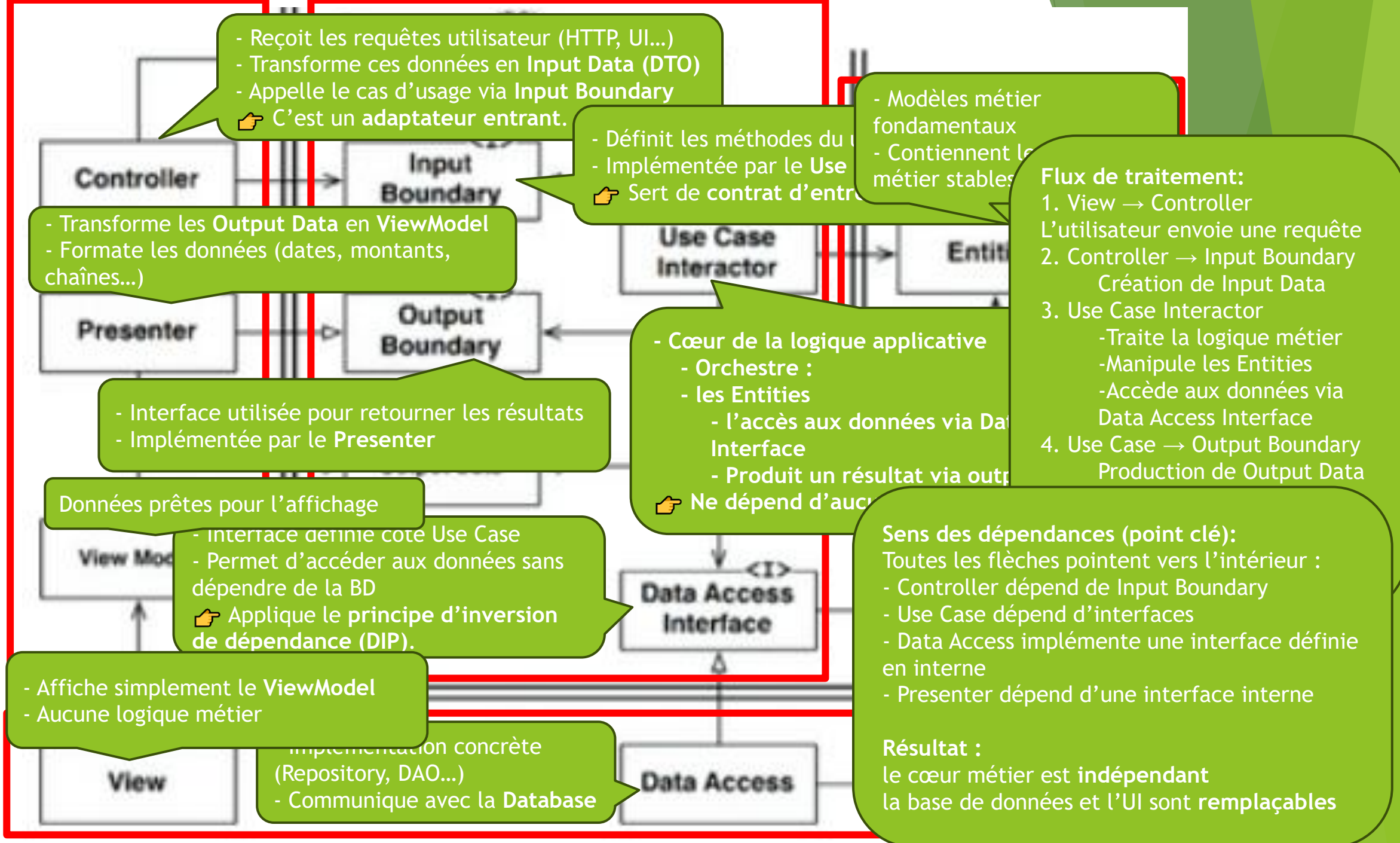
Quels types de données traversent les frontières

- En effet, ces structures introduiraient des dépendances contraires à la règle de dépendance, en obligeant une couche interne à dépendre de détails appartenant à une couche externe.
- À titre d'exemple, de nombreux frameworks de persistance retournent des structures de données pratiques sous forme de résultats de requêtes, souvent appelées « lignes » ou row structures.
- Ces structures ne doivent pas être propagées vers les couches internes.
- Leur utilisation violerait la règle de dépendance, car elle forcerait les couches internes à connaître des détails liés aux couches externes.
- Ainsi, les données transmises à travers les frontières doivent toujours être présentées sous une forme adaptée aux besoins de la couche interne, indépendamment de leur représentation dans les couches externes.

Exemple

- un scénario typique d'un système Java basé sur le Web et utilisant une base de données.
- Le serveur web collecte les données d'entrée fournies par l'utilisateur et les transmet au Controller.
- Celui-ci encapsule ces données dans un objet Java simple, puis fait transiter cet objet via l'interface **InputBoundary** vers le **UseCaseInteractor**.
- Le UseCaseInteractor interprète ces données et les exploite afin d'orchestrer les interactions entre les Entities.
- Il s'appuie également sur l'interface DataAccessInterface pour charger en mémoire, depuis la base de données, les informations nécessaires au fonctionnement de ces entités.
- Une fois le traitement achevé, le UseCaseInteractor collecte les données issues des entités et construit un objet de sortie (OutputData).
- Cet objet est ensuite transmis, via l'interface OutputBoundary, au Presenter.
- Le rôle du Presenter consiste à reconditionner les OutputData en une forme directement exploitable pour l'affichage, appelée ViewModel, qui est également un simple objet Java.
- Il ne reste ainsi à la **View** pratiquement aucune responsabilité, si ce n'est de transférer les données du **ViewModel** vers la page HTML.


<DS>
Input Data
<I>
Input Boundary
<I>
Output Boundary
<DS>
Output Data
Controller
Presenter
<DS>
View Model
Use Case Interactor
Entities
<I>
Data Access Interface
View
Data Access
Database


- Reçoit les requêtes utilisateur (HTTP, UI…)
- Transforme ces données en **Input Data (DTO)**
- Appelle le cas d'usage via **Input Boundary**
👉 C'est un **adaptateur entrant.**
Controller
Input Boundary
- Définit les méthodes du
- Implémentée par le **Use**
👉 Sert de **contrat d'entr**
- Modèles métier fondamentaux
- Contiennent l
métier stables
Flux de traitement:
1. View → Controller
L'utilisateur envoie une requête
2. Controller → Input Boundary
Création de Input Data
3. Use Case Interactor
-Traite la logique métier
-Manipule les Entities
-Accède aux données via Data Access Interface
4. Use Case → Output Boundary
Production de Output Data
Use Case Interactor
Entit
- Transforme les **Output Data** en **ViewModel**
- Formate les données (dates, montants, chaînes…)
Presenter
Output Boundary
- Interface utilisée pour retourner les résultats
- Implémentée par le **Presenter**
- **Cœur de la logique applicative**
- **Orchestre :**
- **les Entities**
- **l'accès aux données via Da**
Interface
- **Produit un résultat via outp**
👉 **Ne dépend d'auc**
Données prêtes pour l'affichage
View Mod
- Interface définie côté Use Case
- Permet d'accéder aux données sans dépendre de la BD
👉 Applique le **principe d'inversion de dépendance (DIP).**
<I>
Data Access Interface
Sens des dépendances (point clé):
Toutes les flèches pointent vers l'intérieur :
- Controller dépend de Input Boundary
- Use Case dépend d'interfaces
- Data Access implémente une interface définie en interne
- Presenter dépend d'une interface interne
Résultat :
le cœur métier est **indépendant**
la base de données et l'UI sont **remplaçables**
- Affiche simplement le **ViewModel**
- Aucune logique métier
View
implémentation concrète
(Repository, DAO…)
- Communique avec la **Database**
Data Access

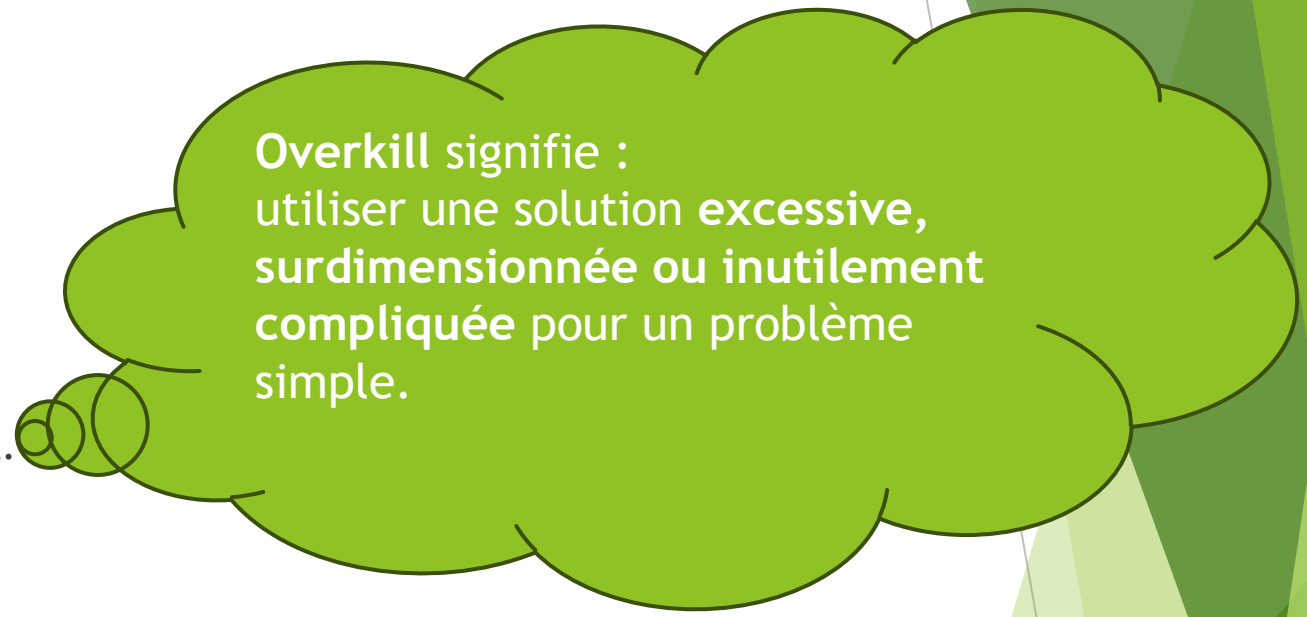
Avantages de la Clean Architecture

- **Indépendance technologique:** Le cœur métier ne dépend pas :
  - de la base de données
  - du framework (Spring, Django…)
  - de l'interface utilisateur
  - on remplacer une DB ou un framework sans impacter le métier.
- **Forte testabilité**
  - Les **Use Cases** peuvent être testés sans UI ni DB
  - Résultat : tests unitaires rapides, meilleure qualité logicielle
- **Séparation claire des responsabilités:** Chaque couche a un rôle précis :
  - Entities → règles métier
  - Use Cases → logique applicative
  - Adapters → communication externe
  - Cela réduit : le couplage et les effets de bord

- **Maintenabilité élevée**
  - Code plus lisible
  - Structure stable dans le temps
  - Idéal pour : projets long terme et systèmes complexes
- **Flexibilité et évolutivité**
  - Ajout facile de nouvelles interfaces (web, mobile, API)
  - Adaptation aux changements métier
- **Alignement avec les bonnes pratiques**
  - Compatible avec : SOLID, Hexagonal Architecture, …

Inconvénients de la Clean Architecture

- **Complexité initiale élevée**
  - Beaucoup de couches
  - Beaucoup de classes
  - Pour un petit projet, c'est souvent overkill.
- **Surcoût de développement**
  - Plus de code à écrire
  - Plus de configuration
  - Temps de développement initial plus long.
- **Courbe d'apprentissage**
  - Concepts abstraits (Use Case, Gateway, etc.)
  - Nécessite une discipline d'équipe



- **Boilerplate important**
  - DTOs
  - Interfaces
  - Mappers
  - Risque de duplication de code.
- **Risque de mauvaise application**
  - Si mal utilisée :
  - trop d'abstraction inutile
  - complexité sans bénéfice réel

Le **boilerplate** désigne :
du **code répétitif, standard et peu métier**, que l'on doit écrire encore et encore sans réelle valeur ajoutée.

Avantages	**Inconvénients**
Indépendance technologique	Complexité élevée
Testabilité	Surcoût initial
Maintenabilité	Boilerplate
Flexibilité	Courbe d’apprentissage
Faible couplage	Parfois surdimensionnée

Conclusion

- La Clean Architecture est :
  - très puissante pour les systèmes complexes
  - inutilement lourde pour les petits projets
- Recommandée:
  - système est évolutif
  - plusieurs développeurs travaillent dessus
  - le métier est complexe