IUT Lyon 1
l'excellence technologique

Chap.2.2 – Paradigmes de Conception (suite)

V. Deslandres ©

Conception d'Architectures Logicielles

LP DevOps

IUT de Lyon - Université Lyon 1

Pourquoi ce souci de l'architecture

- Anticiper les évolutions futures sans faire de sur-conception
- Séparer les éléments
  - Limiter le périmètre des modifications
  - Meilleure lisibilité

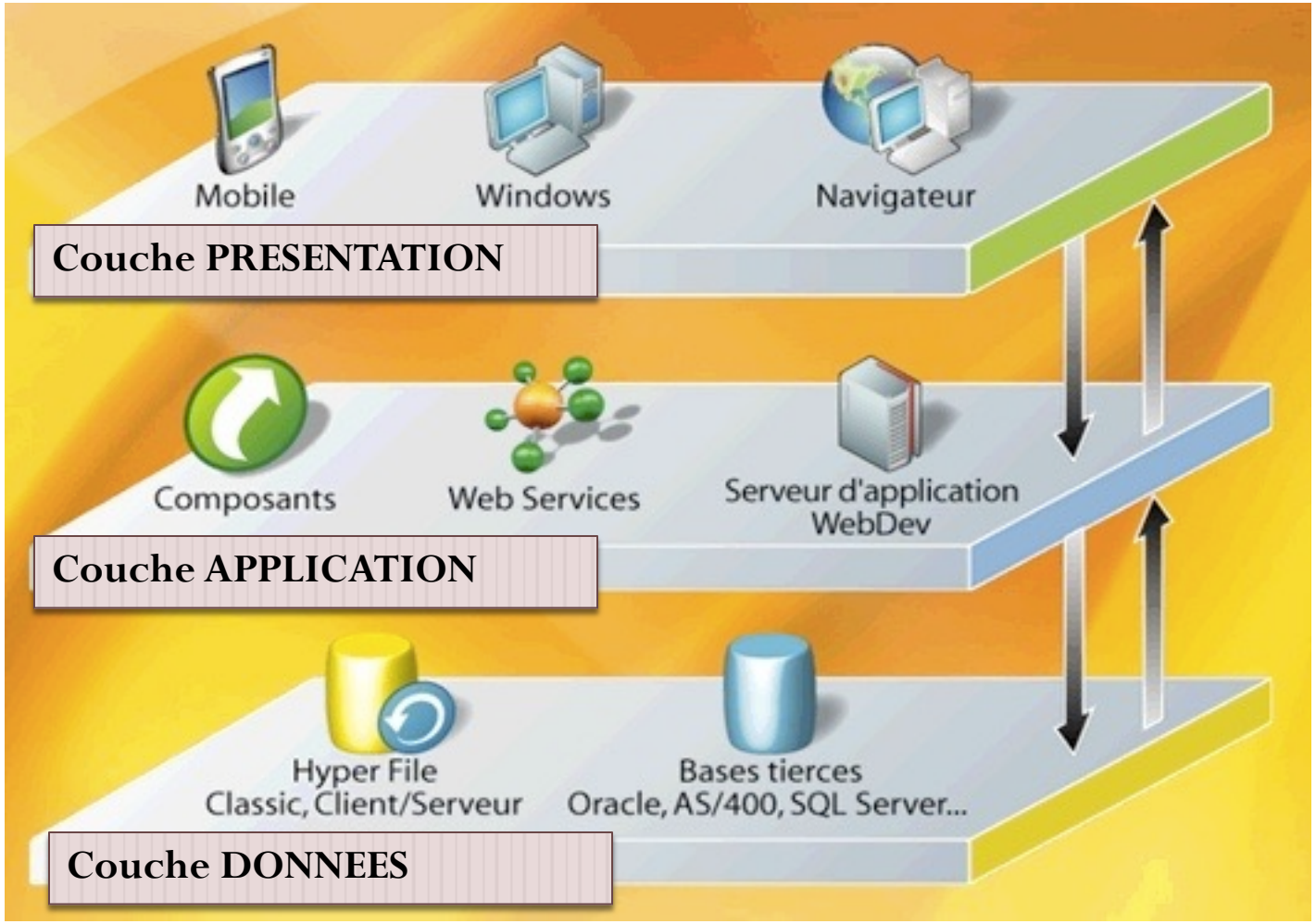
- Ca n'est pas qu'un **problème des applications monolithiques**
  - (même si le fait d'avoir de nombreuses couches est un problème)

  - Une **application distribuée** peut par exemple être impactée si on change de *driver* et donc le format des données, actuellement utilisé par tous les services.

Architecture en 3 couches

Chaque niveau n'interagit qu'avec les niveaux immédiatement proches

Niveau Logique:
Règles Métier
Ex.: Traiter un passage en Caisse



Un exemple d'architexture 5-tiers

- Présentation IHM
- Contrôleur
- ~~Application / règles~~ Métier
- ~~Domaine / Logique des~~ Données
- Accès à la Persistance

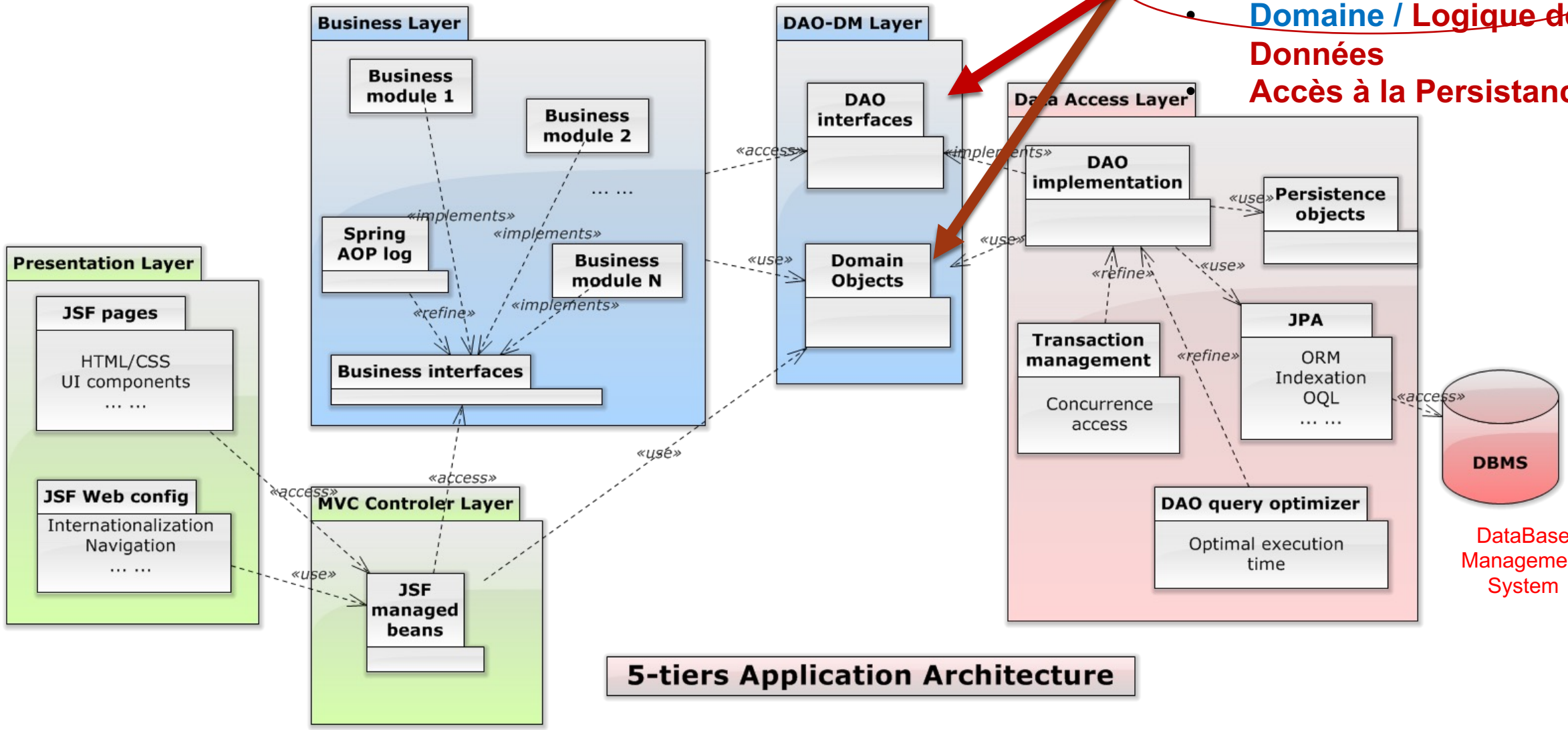


Illustration : détails

Couche Présentation (*presentation layer*)

- Cette couche contient essentiellement les pages Web sous forme de JSF (*Java Server Faces*) : éléments HTML + les composants JSF.
- La mise en page est assurée par les feuilles de style CSS.
- La configuration de l'environnement Web est assurée par les fichiers XML. On peut configurer de multiples langages (internationalisation).
- Les navigations entre les pages peuvent être aussi définies de façon explicite dans ces fichiers XML.

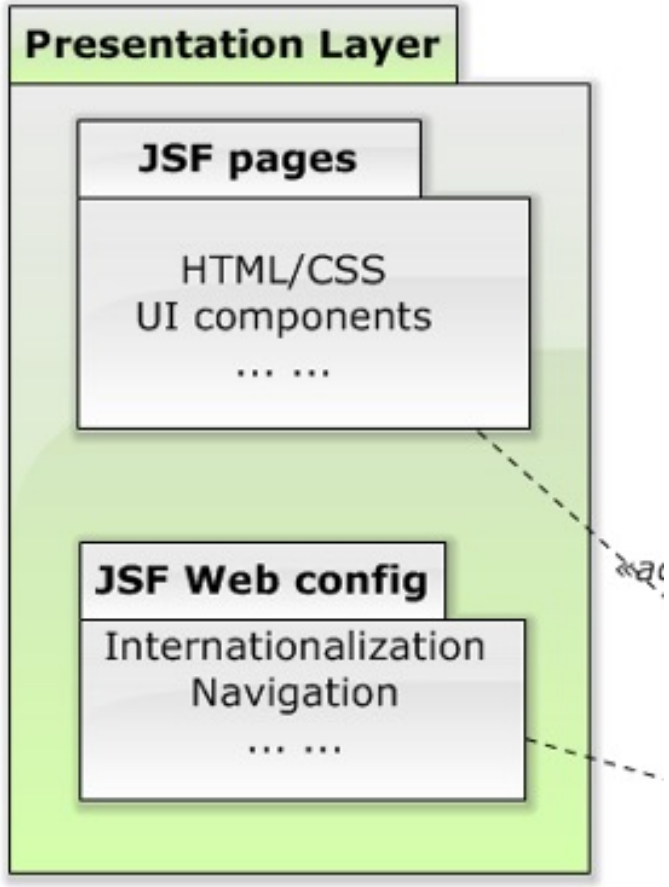
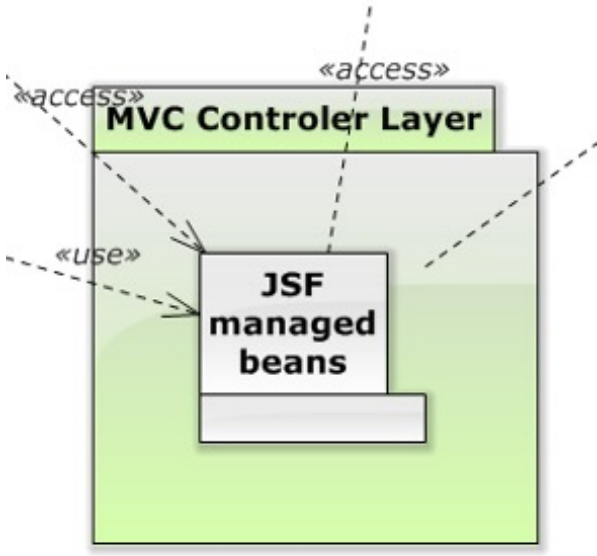


Illustration : détails (2)

Couche Contrôleur (*MVC - controller layer*)

- Cette couche contient les classes utilisées comme *contrôleur* du modèle MVC. Le framework JSF est lui-même de nature MVC : on utilise les classes *JavaBean* (*Managed Beans*) pour répondre aux requêtes venant des pages JSF (Vue), en appelant les services (méthodes) fournis par la couche métier (Modèle).



- **Cette couche intermédiaire assure un découplage entre la couche Présentation et la couche Métier**, qui travaillent de façon indépendante, avec une possibilité de gestion des traitements et des données demandées par la Vue.
- Cette couche doit avoir **accès aux classes** du domaine (*Domain Object*) définis dans la couche DAO-DM (*Domain Management*).

Détails (3) - Couche métier (*Business layer*)

- Cette couche s'occupe de la **partie fonctionnelle** (le métier) de l'application.
- La couche s'organise en plusieurs sous-modules fonctionnels permettant de réaliser les services (*Business Interfaces*) accessibles par les contrôleurs.
- Pour implémenter ces services, on doit avoir un accès aux services d'accès et de traitement de données, **exposés par les interfaces DAO** (*Data Access Objects*).
- Les **objets de domaine** sont également utilisés ici, même s'ils figurent dans la couche DAO - DM.

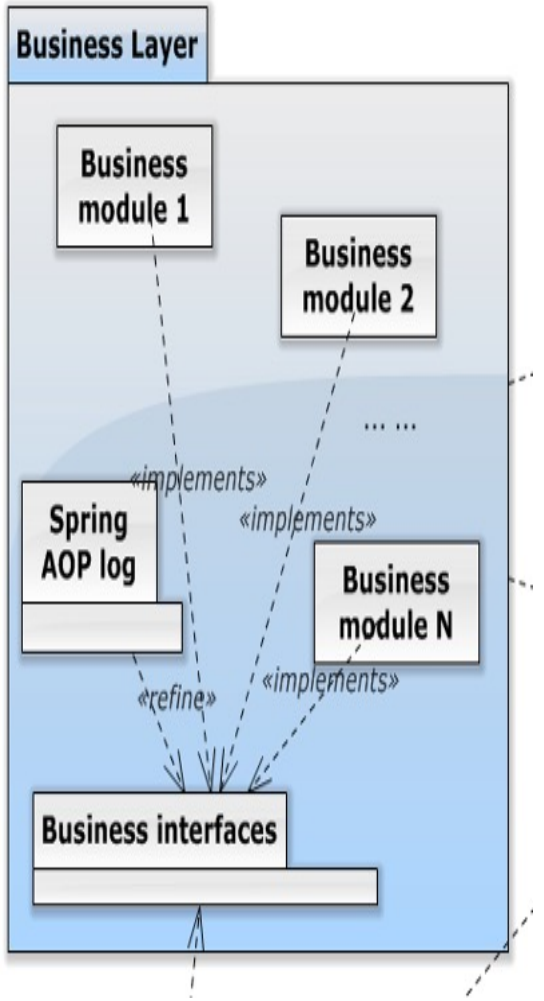


Illustration : détails (4) - Couche DAO-DM

DAO signifie « *Data Access Objects* » et DM « *Domain Management* »

- Cette couche intermédiaire, entre la couche métier et la couche d'accès aux données, fournit un **ensemble d'interfaces** permettant d'effectuer les opérations sur les données gérées par l'application.
  - Typiquement la création, lecture, mise à jour et suppression des données (CRUD)
- Dans cette couche, on ne définit **que les interfaces DAO :** elles sont implémentées dans la couche d'Accès aux Données.
- De plus, cette couche contient un **ensemble d'objets du domaine,** qui sont utilisés et visibles par toutes les couches sauf la couche d'Accès aux Données.
  - Le principe est d'avoir un découplage entre le modèle « logique » de données et le modèle « physique » de données. Ce dernier est géré par la couche d'Accès aux Données sous forme d'objets persistants.

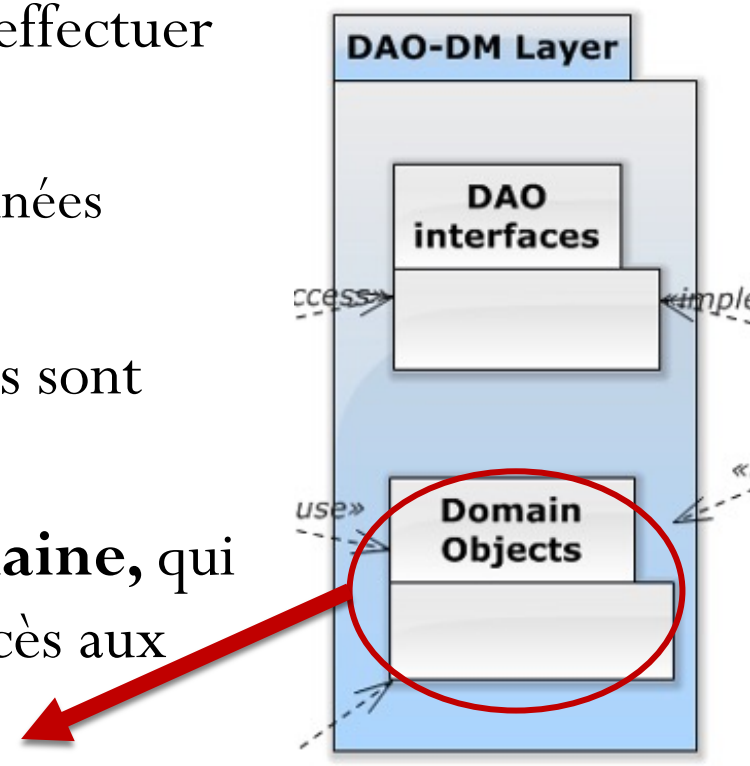
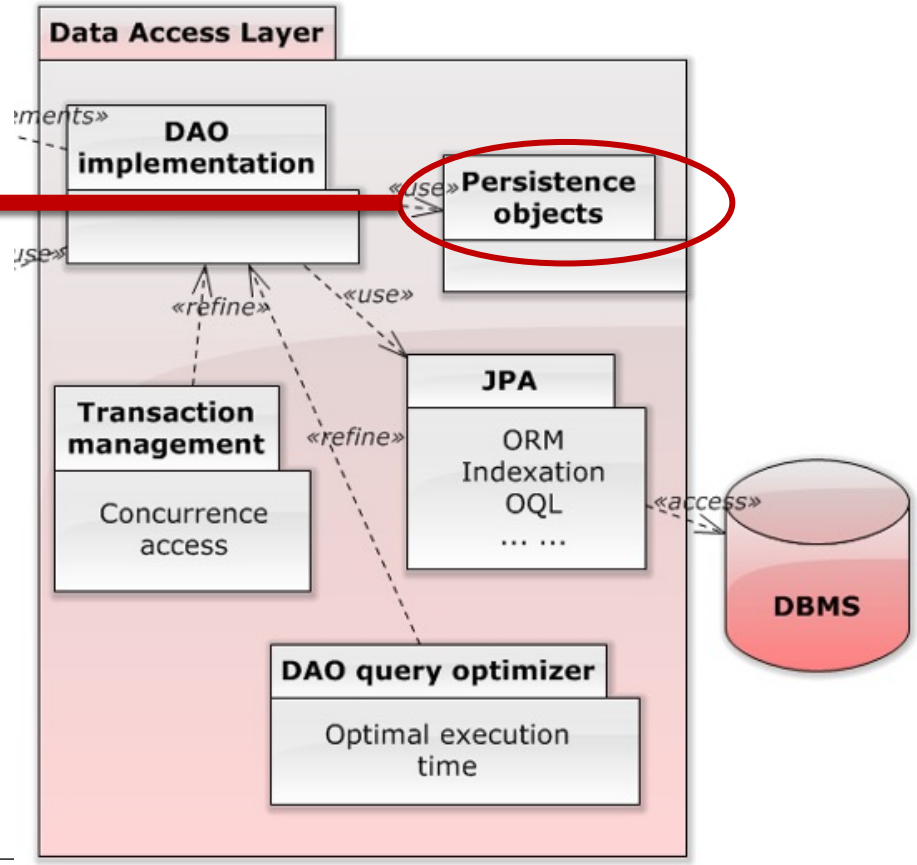


Illustration : détails (5) - Couche d'Accès aux Données

- ***Data Access layer***
- Cette couche s'occupe de tout ce qui concerne **l'interaction avec le système de persistance** (ici un SGBD relationnel). Les classes de base sont celles qui implémentent les services DAO (interfaces). Pour ceci, on a besoin de plusieurs composants :

✓Les classes de persistance qui correspondent aux données physiques via l'ORM (*Object-Relational Mapping*). Avec les annotations JPA (*Java Persistence APIs*), on spécifie les informations liées à la persistance:

clé primaire, relations *one-to-one, one-to-many, many-to-many*, gestion de l'héritage, données embarquée (*embedded entity*), etc.



*Donnée **logique** = une classe CompteUtilisateur*
*Donnée **physique** = la table Compte*

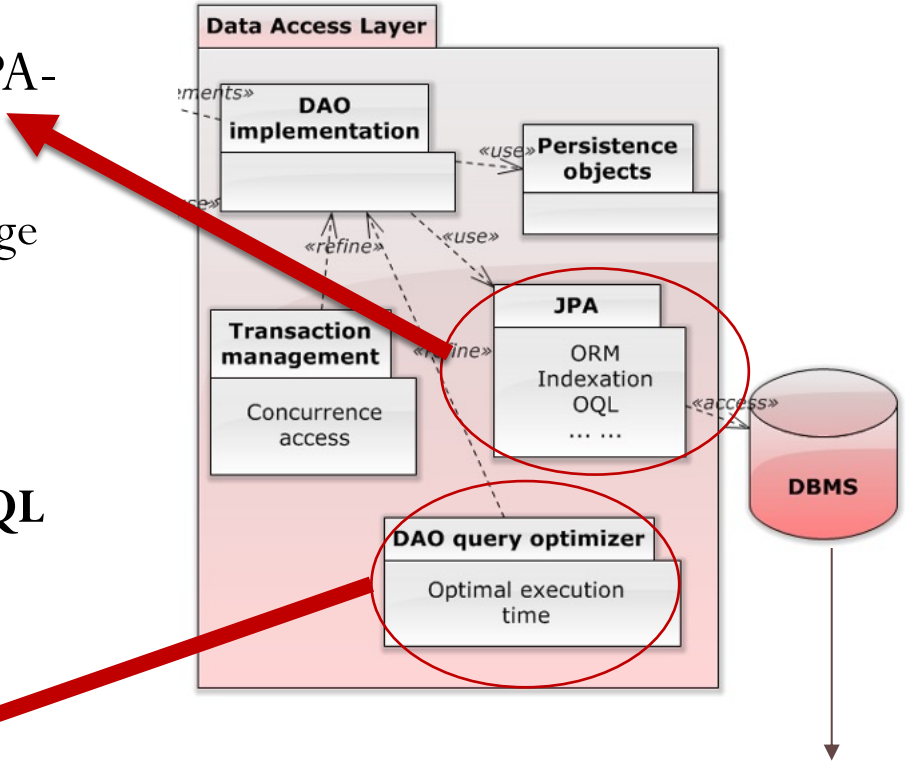
Illustration : détails (6) - Couche d'Accès aux Données

- ✓ La gestion des transactions s'effectue à l'aide du framework JPA-Hibernate.

  On peut programmer, configurer et manipuler les aspects de verrouillage des transactions sur les données, en respectant les propriétés des **transactions ACID** (Atomique, Cohérente, Isolée et Durable).

  Les transactions sont gérées à l'aide du **multi-threadings**.

  Pour effectuer les opérations sur les données, on utilise **le langage OQL (*Object Query Language*)**.



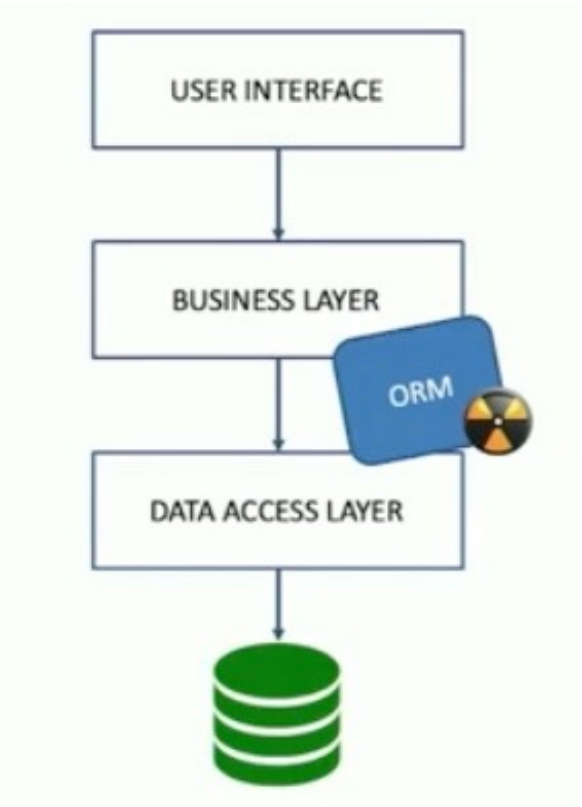
- ✓ **L'optimisation de requêtes** est ici **statique**, puisque les requêtes DAO sont déjà définies dans la couche DAO-DM.

  Il s'agit ainsi de choisir la meilleure façon pour exécuter (implémenter) les requêtes DAO : diviser en sous-requêtes, traitement en mémoire, choix de stratégie de « *fetch* » pour les collections, etc.

- ✓ Le **SGBD** utilisé ici est **PostgreSQL** avec toutes les propriétés usuelles (schéma, indexation, contraintes …) permettant une bonne exploitation de l'accès aux données.

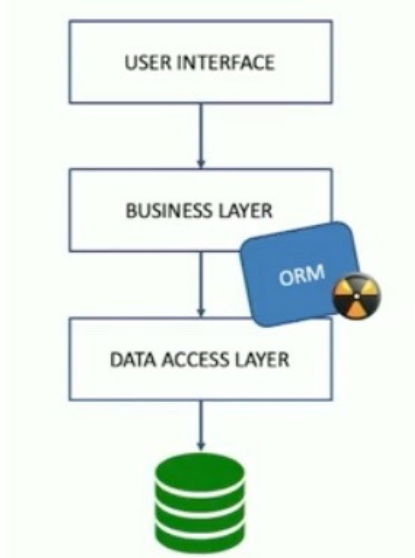
Exercice : Bonne ou Mauvaise pratique ?

- Chaque service applicatif dépend de l'objet **racine** de l'arbre décrivant les objets du domaine
- Utiliser des **requêtes dynamiques** sur le modèle objet
- Faire de **gros tests unitaires**, avec une vraie conception de test
- Ecrire des objets Métier avec **tous les get/set en public** des attributs



Exercice : Bonne ou Mauvaise pratique ? (corrigé)

- Chaque service applicatif dépend de la **racine** de l'arbre du domaine
  - ça signifie qu'on va devoir gérer toute la hiérarchie des objets ➔ lourd, nombreux tests
- Utiliser des **requêtes dynamiques** sur le modèle objet
  - traitement plus ciblé, tests et évolutions de code plus faciles



- Faire de **gros tests unitaires**, avec une vraie conception
  - un test compliqué à coder = mauvais test
- Ecrire des objets Métier avec **tous les get/set** des attributs
  - C'est un '*anti-pattern*' selon Martin Fowler, pas de l'OO
  - setAdresse() OK, avec tous les contrôles basiques des champs de l'adresse
    - Préférer une méthode **demenager()** pour la classe Personne : plus riche au niveau Métier, décrit les règles de comportement

MVC vs. Architecture 3-tier

L'architecture 3-tier découpe une Application entière en 3 parties :

- Présentation
- Logique de l'application
- Persistance

- La couche Présentation n'accède jamais directement au niveau Persistance.
- Chaque partie est modifiable sans perturber le reste de l'application.

MVC est un **modèle de conception** de la couche Présentation

- MVC découpe les composants de la partie Présentation comme expliqué précédemment en :
  - Vue (affichage des composants et des données),
  - Modèle (les données brutes et les règles de gestion),
  - Contrôleur (lien entre la Vue et le Modèle)

MVC pour un composant graphique

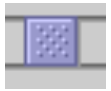
Par ex. un *slider* en Java SWING :

- *Modèle* :
  - valeur minimale = 0
  - valeur courante = 15
  - valeur maximale = 100

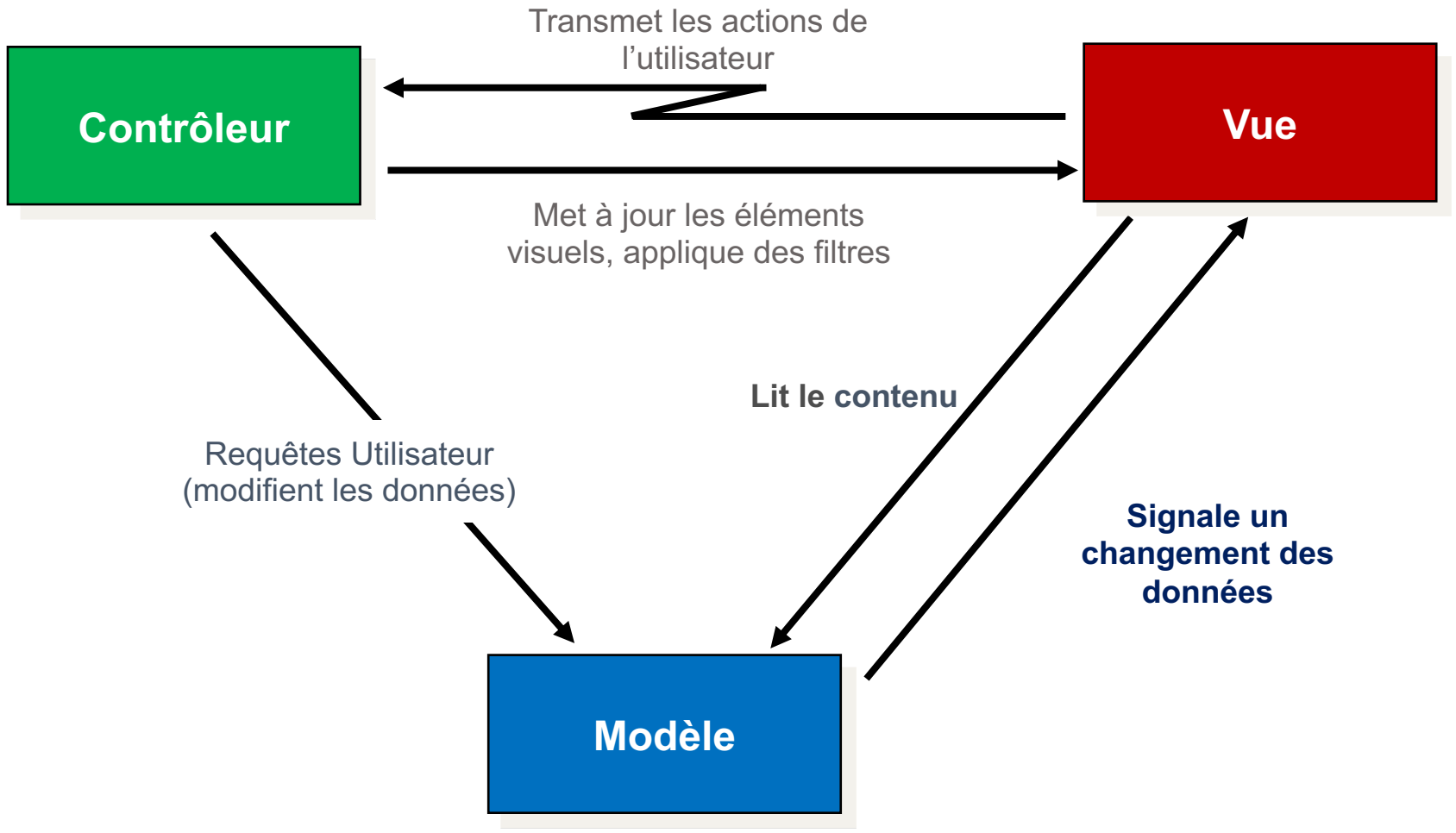
- *Vue* : afficher les données du composant graphique :

- *Contrôleur* :
  - Transmettre les données du modèle vers la VUE
  - Transmettre les actions utilisateur vers le MODELE :
    - Les clics de souris sur les boutons terminaux

      - Les *drags* de souris sur l'ascenseur :



Architecture MVC pour une Interface Utilisateur



Fiche TD#1

Souffler un peu….

Principes élémentaires

Bonne écriture de code

Préambule : exercice

Quelles sont les **3 mauvaises habitudes de code bien connues,** qu'il vous faut absolument éviter dorénavant (en BUT2) ?

1. Mal _____ ses variables, classes, fonctions

   Les _____ choisis aident à comprendre le code en plus des commentaires.

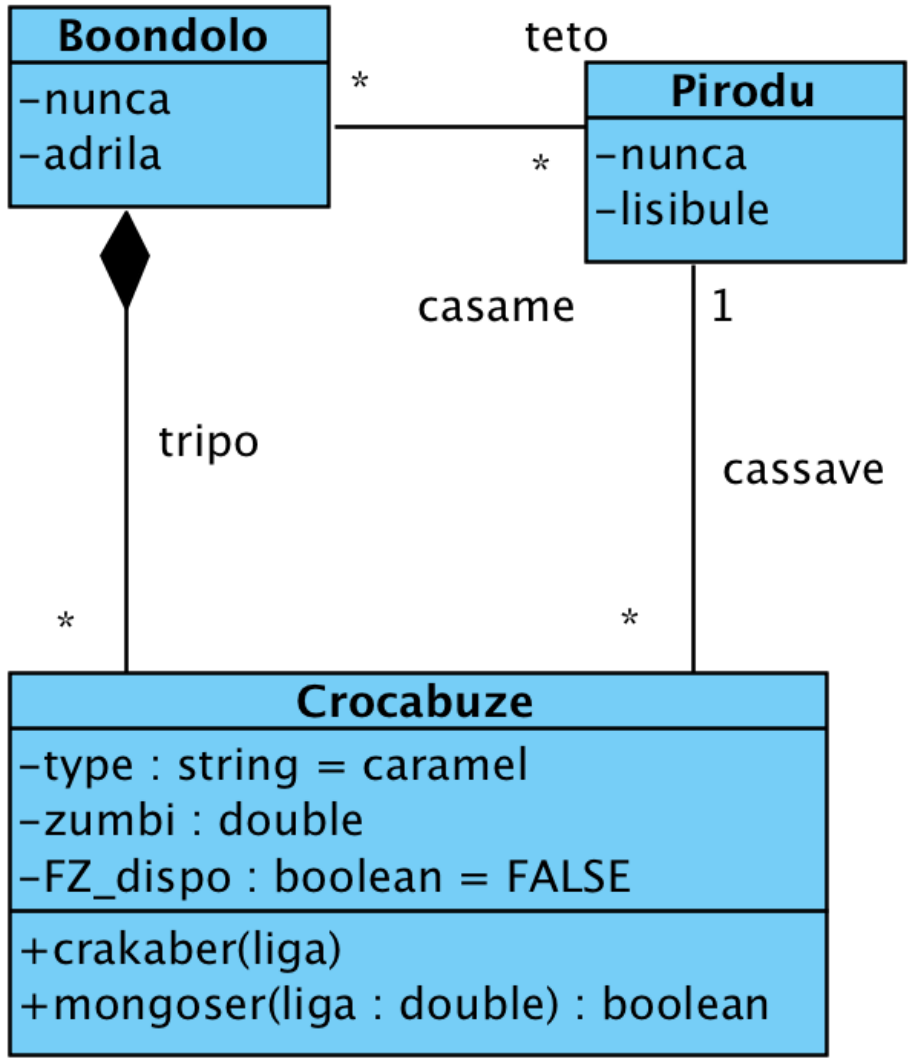
2. Ecrire des _____ __ _____ trop longs·longues

   Comme un article de presse qui utilise les _____, nos _____ doivent soigner nos yeux : on ne peut pas lire une _____ _____ de code.

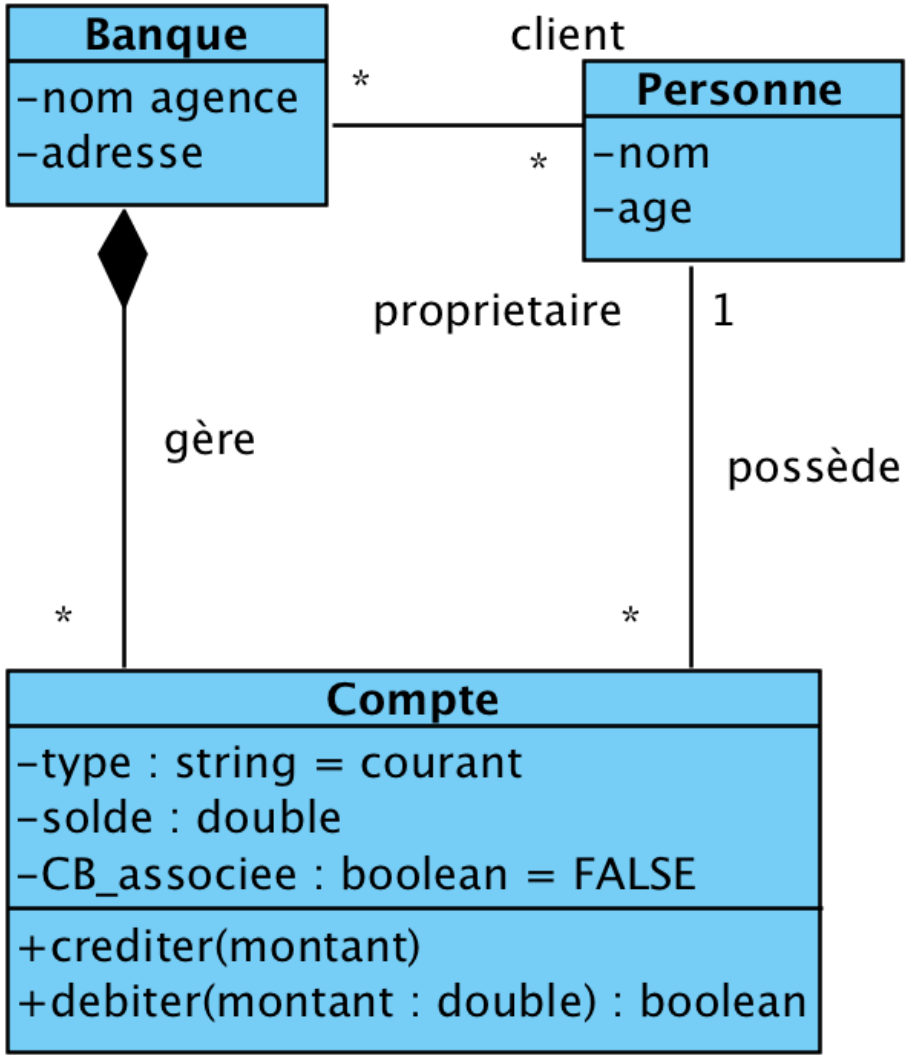
3. Avoir de trop _____ classes ou méthodes

   Se limiter à _____ en général par _____

Importance du nommage : que raconte ce diagramme ?



Ça va mieux !


Banque
-nom agence
-adresse
client
*
*
Personne
-nom
-age
proprietaire
1
gère
possède
*
*
Compte
-type : string = courant
-solde : double
-CB_associee : boolean = FALSE
+crediter(montant)
+debiter(montant : double) : boolean

Importance du nommage

A éviter	Mieux
struct Point {	typedef double **Coord**;
double x, y;	typedef double **Reel_0_1**;
double poids; // entre 0 et 1	struct Point {
};	**Coord** x, y;
	Reel_0_1 poids;
	};

Découpage d'une ligne

Si une ligne **dépasse 80 symboles**, il faut la découper, plusieurs possibilités :

- Si l'indentation est trop grande, il faut subdiviser la fonction.

- Si l'expression est trop compliquée, il faut utiliser des variables intermédiaires qui rendront le code facilement compréhensible si les noms sont bien choisis.

- Sinon, découper aux endroits logiques (opérateurs de faible priorité) en tentant de faire apparaître des alignements verticaux si c'est possible.

https://perso.univ-lyon1.fr/thierry.excoffier/COURS/COURS/TRANS_COMP_IMAGE/prog.html#performance-d-execution

Avantage d'un code court

Un code court :

- Comporte moins d'erreur.
- Est plus facilement lisible.
- Est plus facilement modifiable.
- Est plus vite écrit.

https://perso.univ-lyon1.fr/thierry.excoffier/COURS/COURS/TRANS_COMP_IMAGE/prog.html#performance-d-execution

Autres règles d'une bonne conception

1. **Ne pas mettre des accesseurs / mutateurs** pour tous les attributs systématiquement, sans conditions (règles Métier)

   On perdrait les bénéfices de l'encapsulation

Rappel : pourquoi encapsuler ?

Au sein d'une classe en POO, on encapsule pour mieux **contrôler** les attributs et méthodes :

- Rendre certains attributs en **lecture seule** (seul get() accessible en public), ou en **écriture seule** (seul set() accessible)
- **Plus de flexibilité** : le développeur peut changer une partie du code sans que cela affecte les autres parties (puisque pas d'accès direct)
- Améliorer la **sécurité** des données

C'est **exactement les mêmes raisons** qui font qu'on va encapsuler une classe dans une autre, un composant dans un autre, une application dans une autre

- Le but est de contrôler l'accès à l'élément encapsulé, cacher les détails de son implémentation

Autres règles d'une bonne conception (2)

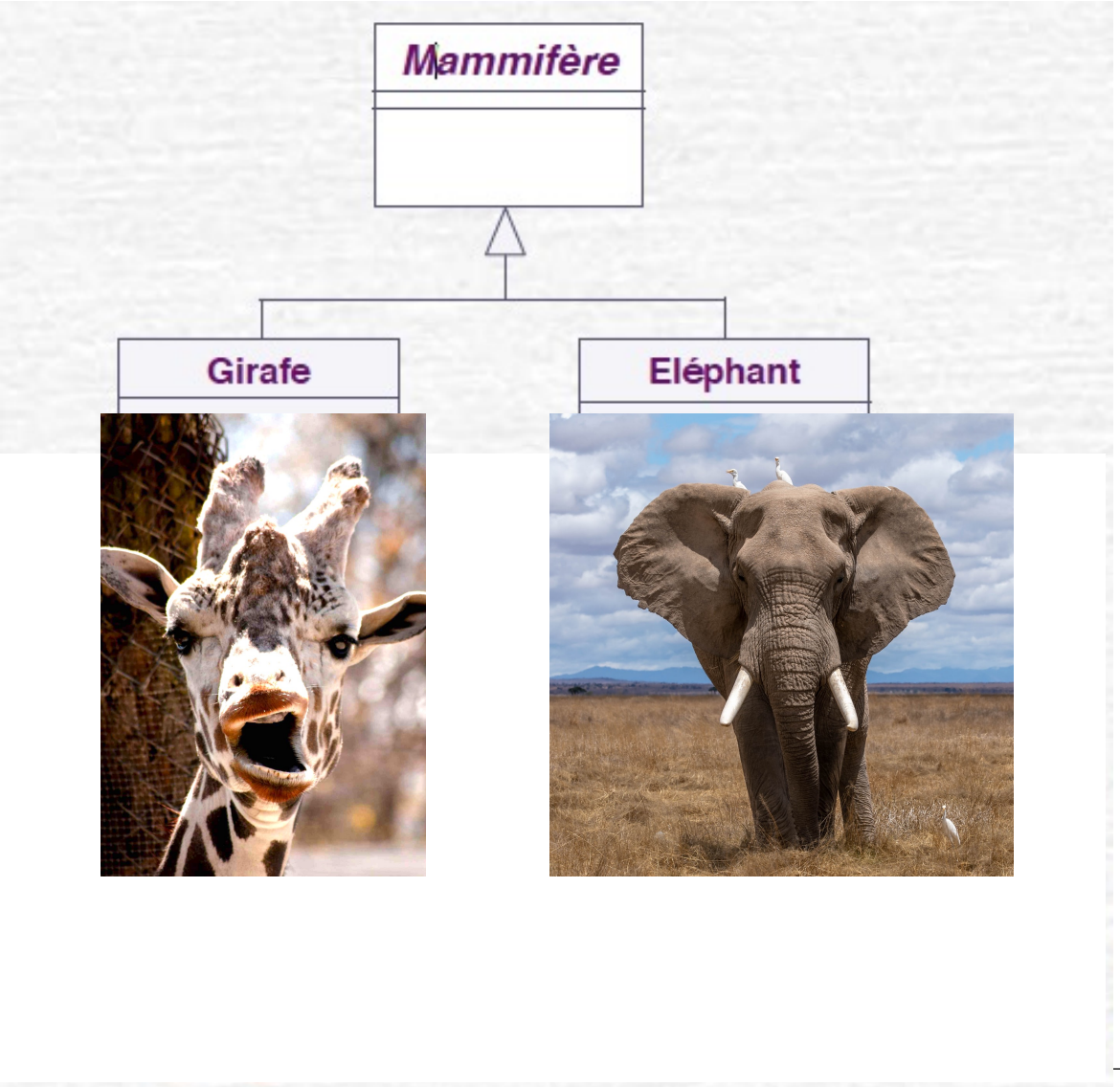
3. Ne jamais dériver une classe en n'exploitant que **certains attributs** et **méthodes**

4. Préférer la **composition** à **l'héritage**

illustrés ci-après…

Règle#3 : « Ne JAMAIS dériver une classe pour certains **seulement de ses** attributs et méthodes »

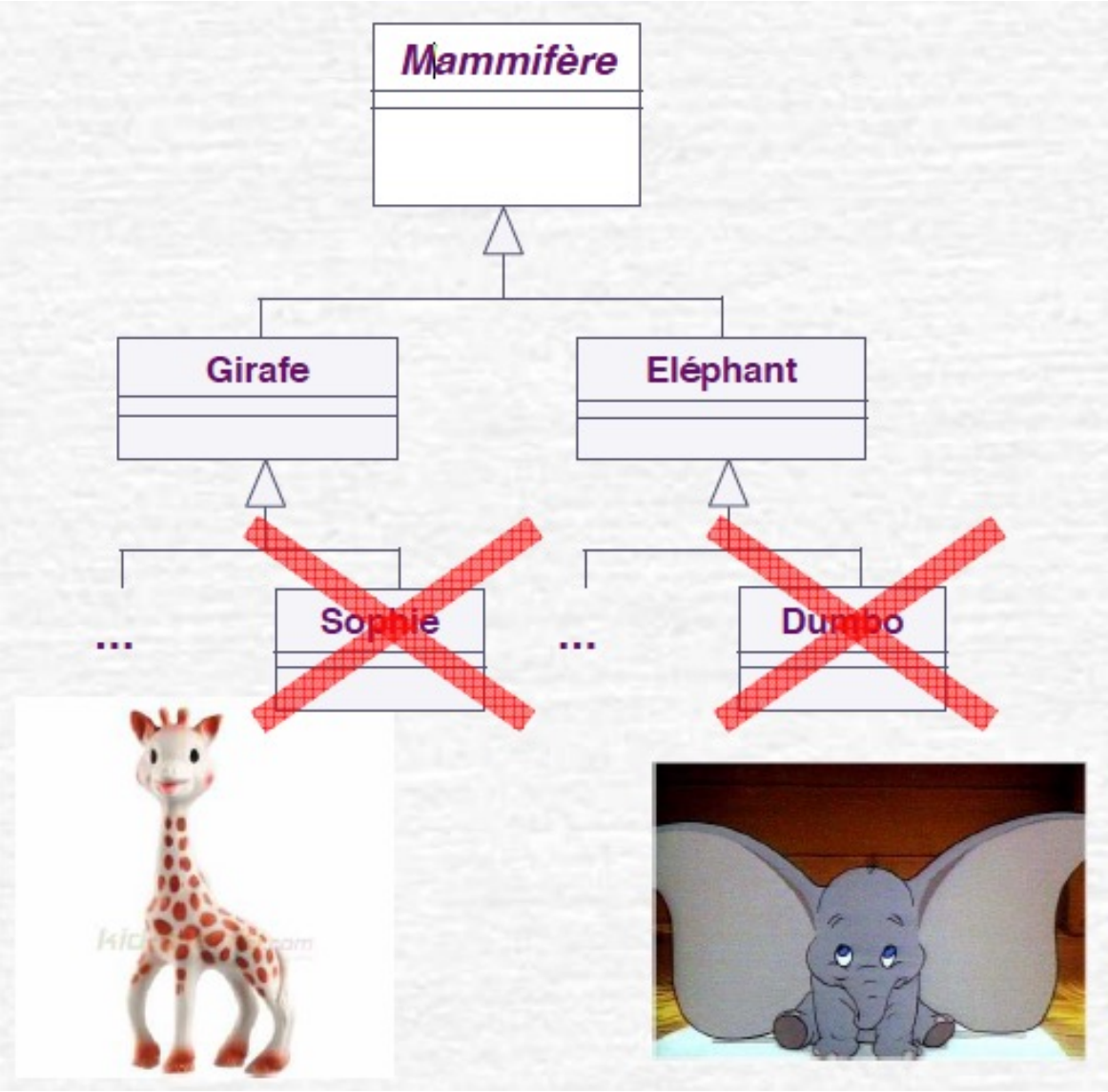
Imaginons un logiciel de simulation d'une **réserve animalière**, avec des girafes et des éléphants.



« Ne JAMAIS dériver une classe pour certains **seulement de ses** attributs et méthodes »

Pour un logiciel de moulage du jouet *Girafe Sophie,* on décide de reprendre la classe *Girafe* **pour dessiner sa robe** : on crée la sous-classe Sophie uniquement pour cela…

Que se passe-t-il si une classe Client applique le **comportement** de la classe Girafe (reproduction, alimentation) à la *Girafe Sophie* ??

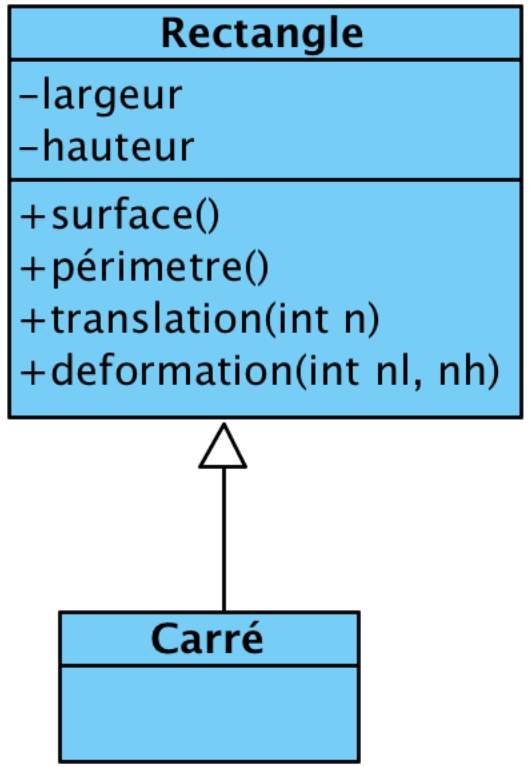


Règles en cas d'héritage

- Les **pré-conditions** définies par les sous-classes ne doivent pas être plus restrictives que celles héritées.
- Les **post-conditions** définies par les sous-classes ne doivent pas être moins larges que celles héritées
- Respecter la **règle des 100%**

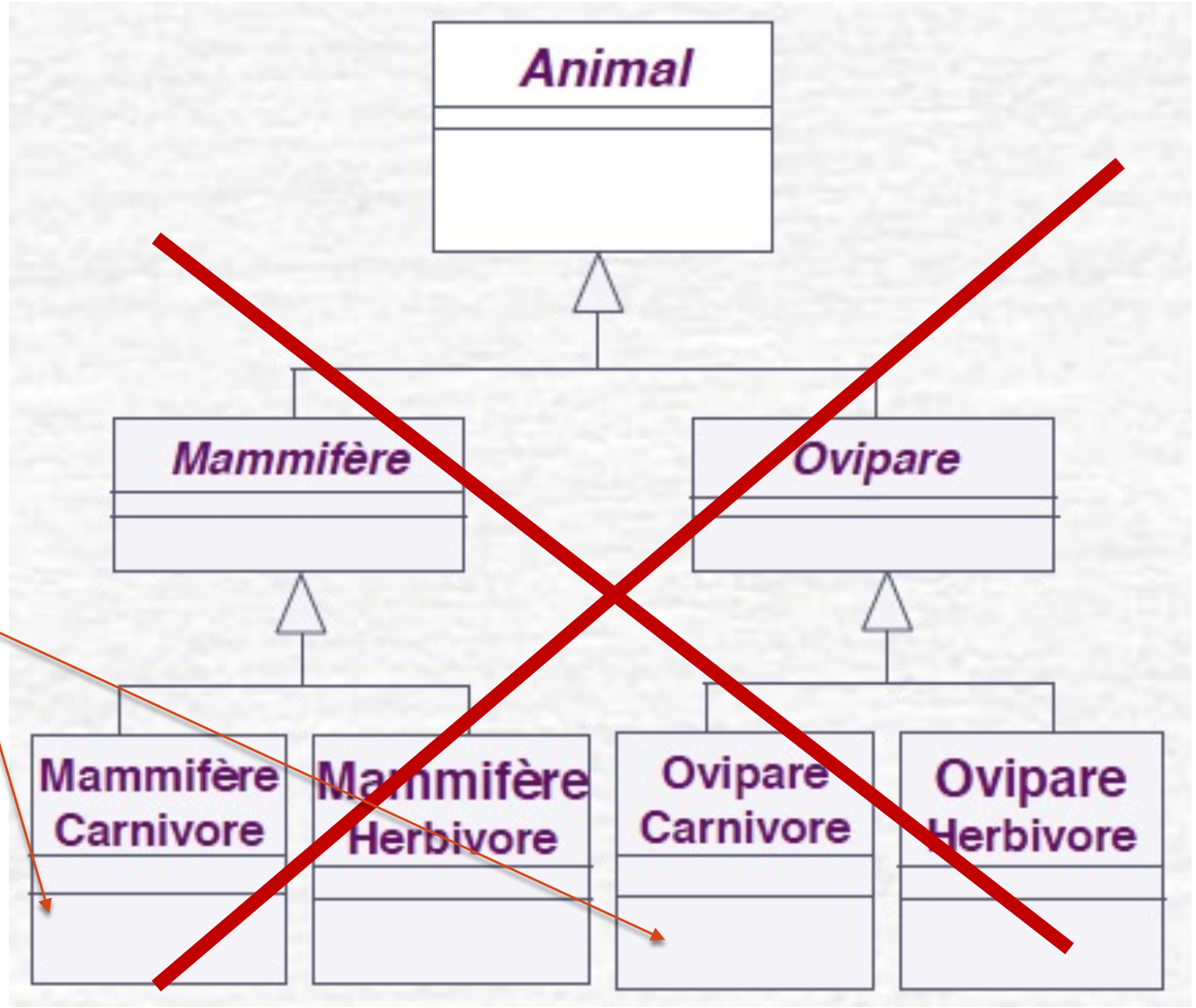
Exercice pré/post conditions

- Que penser de cette conception ?



Dans le constructeur d'un carré, on vérifie que les côtés ont la même taille

Règle#4 :« Préférer la composition à l'héritage »



code de Carnivore dupliqué

- Explosion combinatoire
- Duplication de code

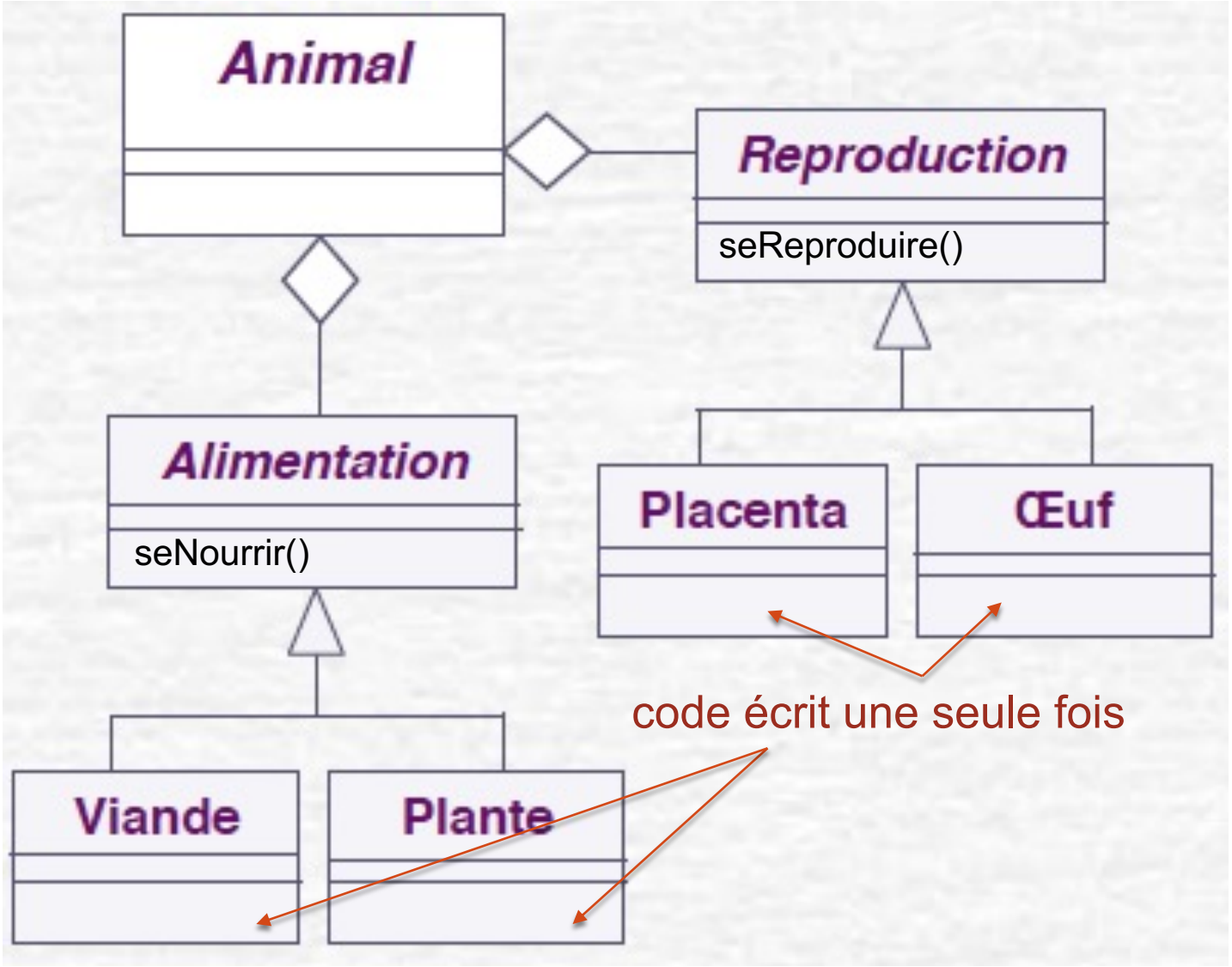
Règle#4 :« Préférer la composition à l'héritage »

On considère qu'un Animal possède 2 caractéristiques intrinsèques, qui varient d'un animal à l'autre :

- son mode d'Alimentation
- son mode de Reproduction

On définit ces modes à chaque instanciation d'Animal :

```
Animal garfield = new Animal();
garfield.setModeReproduction( new Placenta() );
garfield.setModeAlimentation( new Viande() );
…
garfield.getModeAlimentation().seNourrir();
```



Préférer la composition à l'héritage »

Encore appelé principe d'indirection ou de délégation :

on délègue à la classe **Alimentation** le comportement seNourrir() de l'Animal

La classe **Alimentation** est dite *encapsulée* dans Animal.