Chap.1 – Paradigmes de Conception

V. Deslandres ©

R3-04.1 - Qualité de Conception

BUT Informatique - semestre3

IUT de Lyon - Université Lyon 1

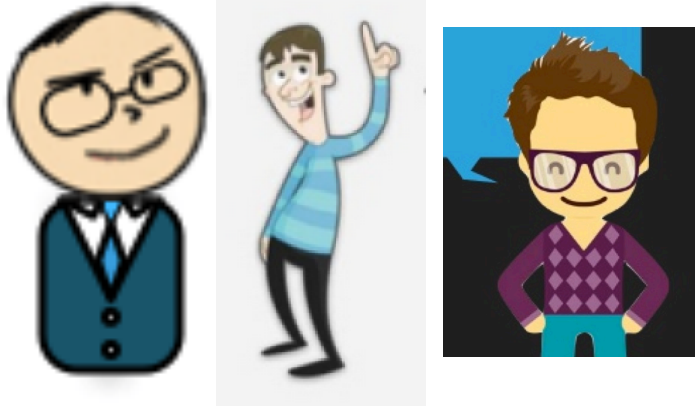
Analyse & Conception : 2 phases distinctes

Conceptual Design

ANALYSE
Conception fonctionnelle

indique au client ce que le système va faire

Client et les utilisateurs finaux



Technical Design

CONCEPTION technique

permet aux développeurs de savoir comment implémenter une solution conforme aux attentes du client

Analystes

Conception fonctionnelle vs. Technique

- **Conception fonctionnelle ou Analyse** : « dissection » du problème
  - Étude du domaine du problème
  - Définition d'un comportement observable de l'extérieur
  - Langage compréhensible par le client
  - Ex. dans l'apps Blablacar : conducteur, passager, trajet, réservation, etc.

QUOI

- **Conception technique** : fabrication d'une solution informatique
  - Détails d'implémentation réelles (IHM, stockage, accès, sécurité, fréquence/débits, performances)
  - Premiers livrables de code
  - Ex.: Découper le trajet en tronçons ; lister les trajets demandés en moins de 2 sec. ; identifier un Conducteur par son nom/prénom/dateNaissance obtenu de son permis ; archiver les entités sans interaction avec l'apps depuis plus de 3 ans

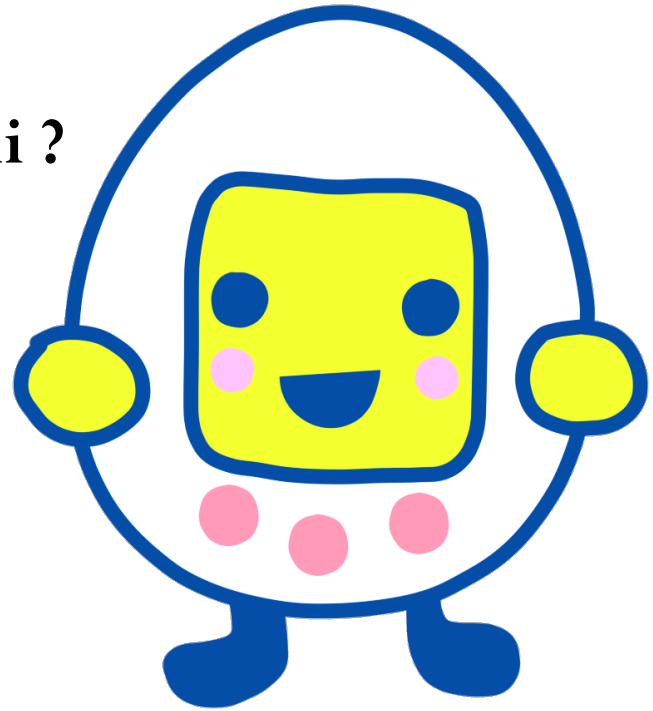
COMMENT

Sommaire de ce cours

Notions de base d'une bonne conception :
1 - Responsabilité de classe

- DEF **Responsabilités** d'une classe :
  - Ce qu'elle **SAIT**
  - Ce qu'elle est capable de **FAIRE**

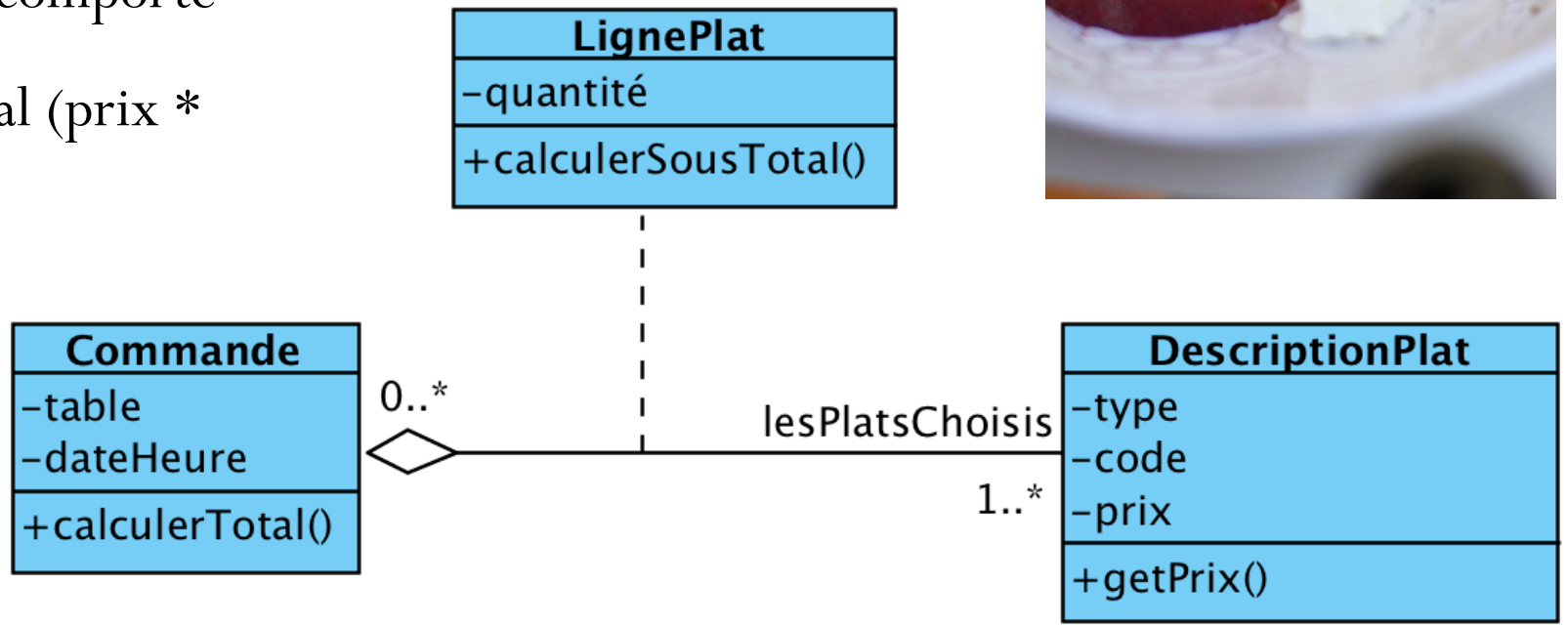
- **Par ex., pour cette classe Tamagoshi ?**



Tamagoshi
-nom -âge -race -couleur -caractère -appétit -statut -vivant : booleen -étatEnnui -étatSatiété -étatSommeil
+nourir(ingredient) +bercer(durée) +distraire(activité) +acheter() +vendre() +définir() +évaluerEtat()

Responsabilité de classe

- Pour la **classe LignePlat** suivante ?
  - Elle sait : à quel objet Commande elle appartient, et à quel objet DescriptionPlat elle correspond
  - Elle sait combien de plats elle comporte
  - Elle peut calculer son sous-total (prix * quantité de plats)

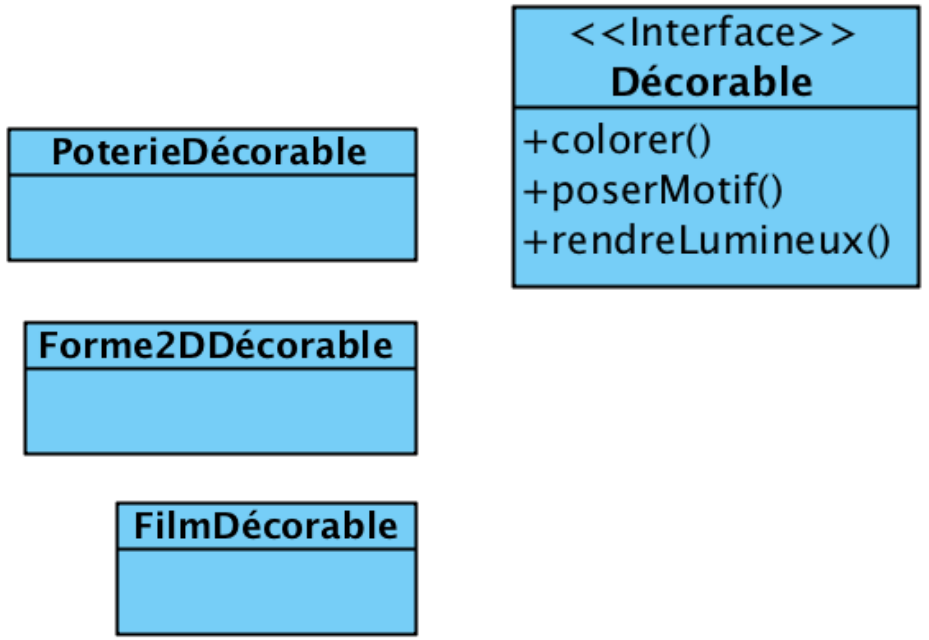


Notions de base : 2- Contrat vs Implémentation

- Le **contrat** = services rendus par une classe
  - Exprimé par la **signature** des méthodes
  - Stable
  - Masque les détails de réalisation
  - Synonymes :

  « Services », « Comportement »

Ensemble des **signatures** des méthodes

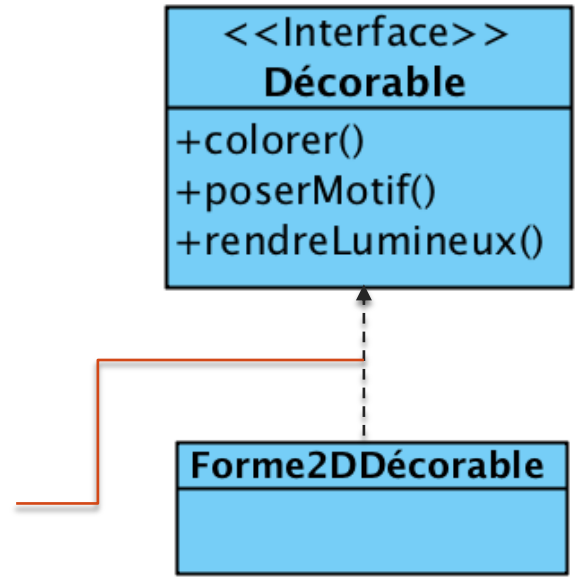


Notions de base : 2- Contrat vs Implémentation

- **L'implémentation**
  - Le code des méthodes
  - Représente les classes dites « concrètes »
  - Peut évoluer

Tous les **codes** associés aux méthodes

- Une des clefs de la maintenabilité : toujours chercher à **dissocier contrat** et **implémentation**



Relation d'implémentation ou de réalisation

Exercice : donner le **contrat** d'une table de symboles

(par ex.: une table associative ou un dictionnaire)

Que peut-on faire avec ? (par ex. une liste de contacts de son mobile)

Exercice : donner le contrat d'une table de symboles

(par ex.: une table associative ou un dictionnaire)

Que peut-on faire avec ? (par ex. une liste de contacts de son mobile)

- put(clé, valeur) : permet d'ajouter une paire (clé, valeur) ;
- get(clé) : retourne la valeur associée à une clé ;
- remove(clé) : supprime la paire correspondant à la clé donnée ;
- contains(clé) : retourne vrai si la clé se trouve dans la table, faux sinon ;
- size() : retourne le nombre de paires présentes dans la table.

Quels sont les moyens d'**implémenter** cette table de symbole ?

On peut citer :

- **Un tableau, une liste chaînée, un arbre, ou une table de hachage.**

Comment choisir ?

- Les performances (temps d'exécution ici) dépendent de la situation et de l'implémentation.
- Par ex. dans une table de hachage, les paires sont éparpillées dans un tableau, à un indice calculé par une fonction de hachage.
- Il faut donc poser des questions, par ex. : **combien de paires aurons-nous à gérer ?**

  Ainsi si on veut gérer les collisions d'indices par la fonction de hachage à l'aide d'un **sondage linéaire***, si on aura à stocker 50 clés, il suffit de prendre un tableau 2x plus grand et les performances resteront bonne.

  Ou bien envisager **d'adapter la taille de la table** en fonction du nombre de clés insérées, autrement dit l'agrandir quand il commence à être rempli (en utilisant le **facteur de charge****)

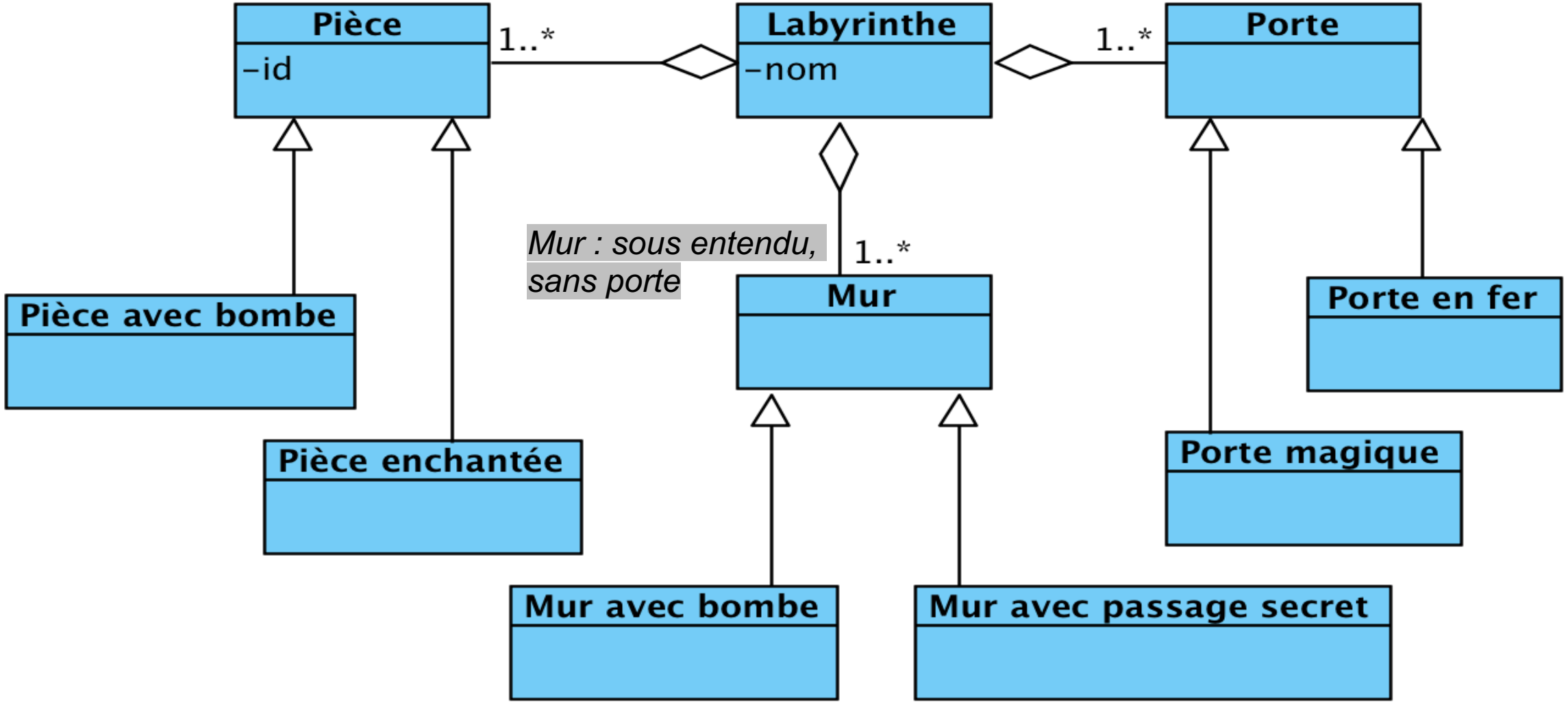
**(probing) : on parcourt les indices suivants, dès qu'un est libre, on l'occupe*
*** nb de clefs insérées / taille du tableau des paires*

Notion de base : 3 - L'abstraction

- Une quête perpétuelle en conception
- Ca signifie : **généraliser des classes**, par exemple :
  - Moto, Voiture, VéloVAE, Trottinette électrique, etc. → Véhicule
  - Capteur de température, capteur d'hygrométrie → Capteurs
  - Etc.

- Mais aussi : créer des **entités abstraites** qui ne sont pas dans l'énoncé, qui ne sont pas cités comme des objets Métier initiaux
- Exemple : jeu de **labyrinthe**
  - 2 sortes de labyrinthes : enchanté ou guerrier
  - Entités citées : pièces, portes, mur

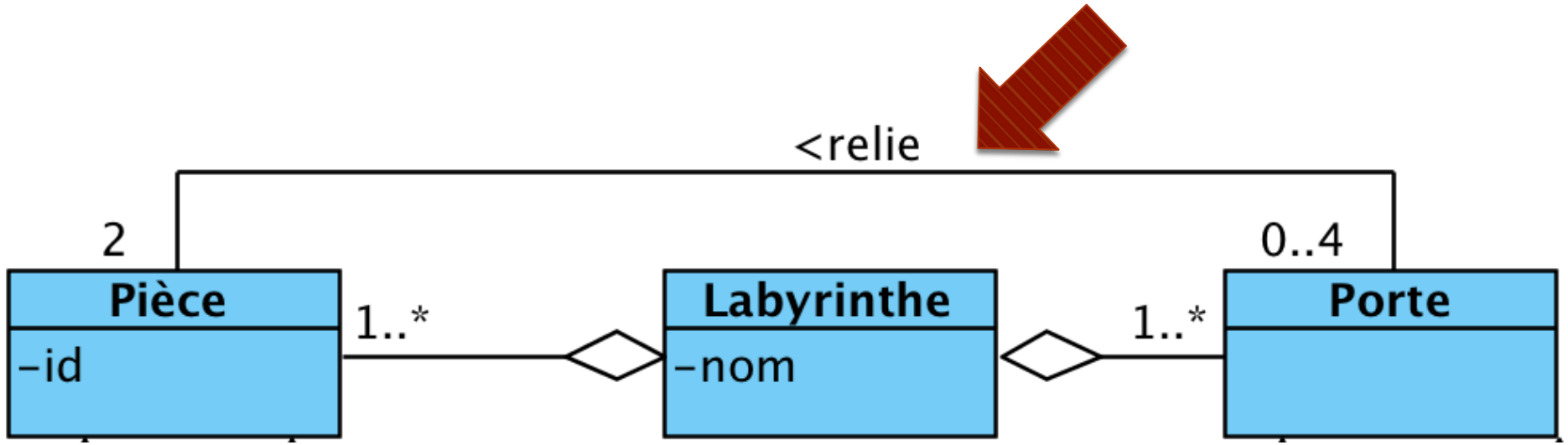
1ère modélisation



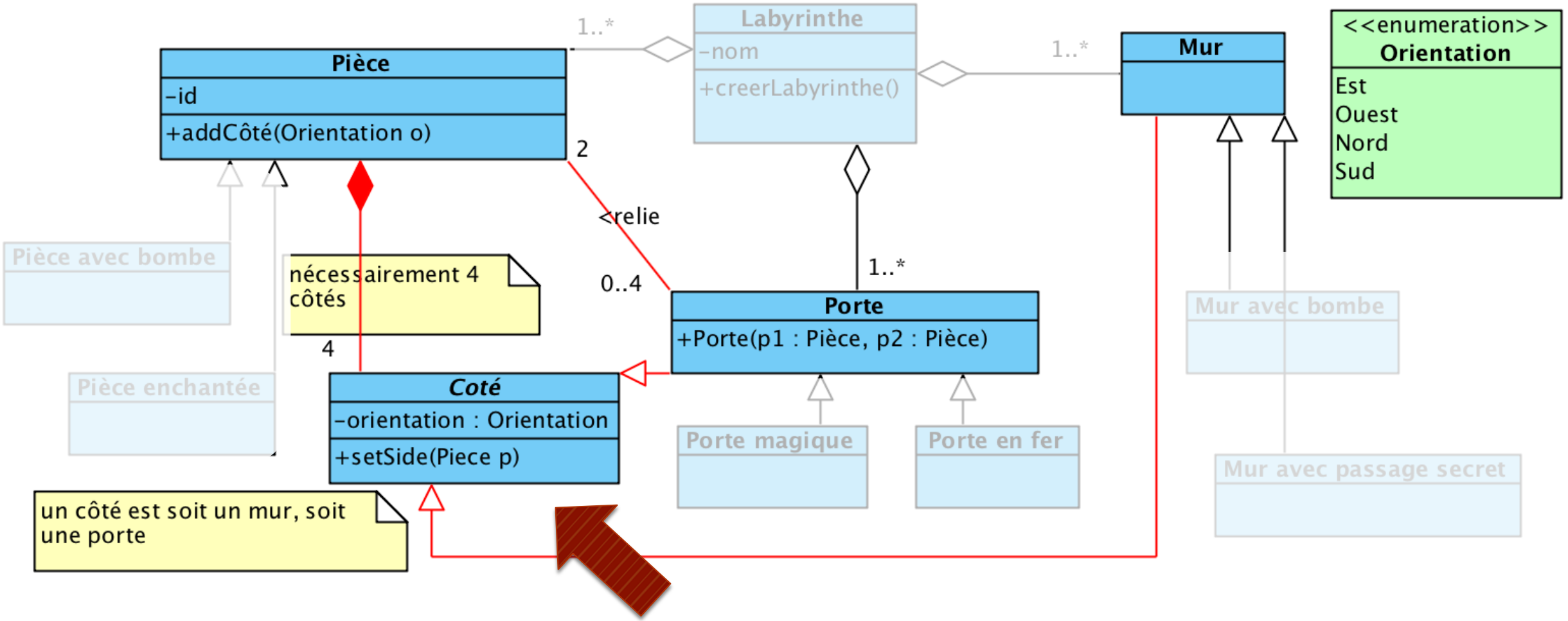
1- Que manque-t-il ici ?

2- Comment signifier qu'une pièce possède des murs et des portes ?

1- Une porte relie nécessairement 2 pièces :



2- Une pièce possède 4 côtés, chaque côté pouvant être un mur ou une porte



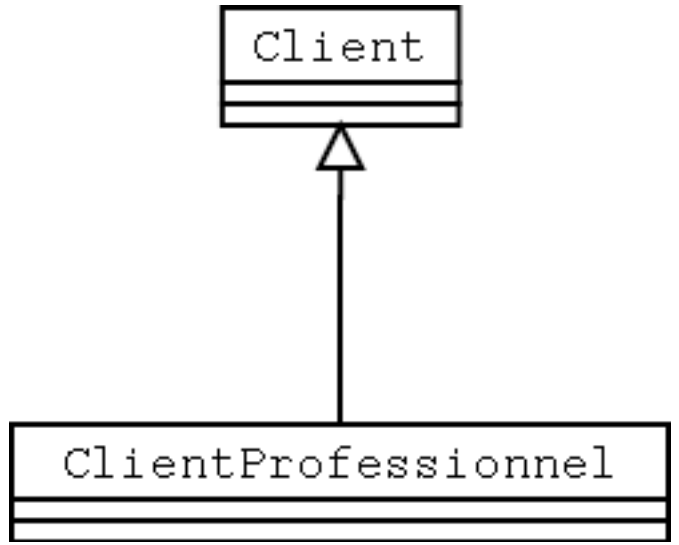
Ici Côté est une classe abstraite

Rappel : Interface vs. classe abstraite

Similaires (méthodes abstraites) mais différentes (objectifs)

- **Avantages d'une interface** : spécifie un contrat, abstraction quasi* complète, héritage multiple d'interfaces, nouvelle implémentation possible à tout moment
  - Analogie : prise de courant, son contrat = fournir du courant alternatif; comment ? pour faire quoi ? Peu importe.

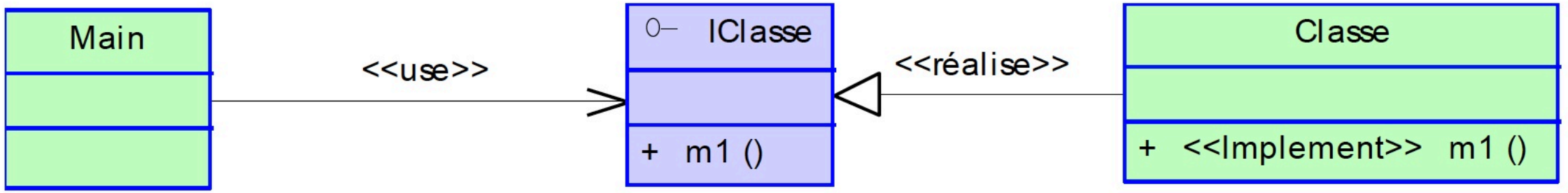
- **Avantages d'une classe abstraite** : définition partielle possible, héritage simple qui permet de spécifier des comportements
  - Ex. : le Client, par ex. avec des méthodes communes pour définir ses infos, ses accès au S.I., etc.



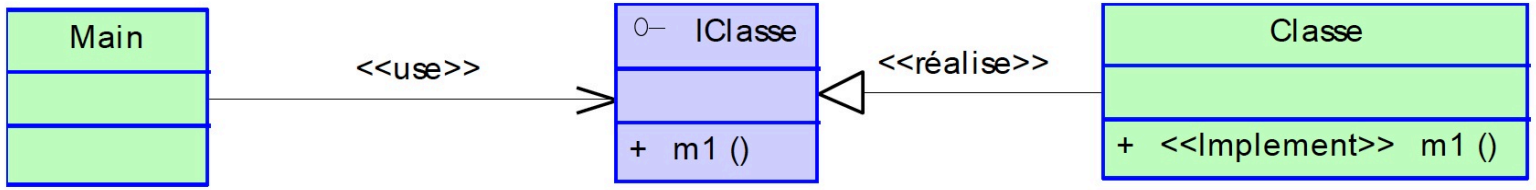
*À partir de Java 8, on peut ajouter 2 éléments dans une interface : des **méthodes statiques** et des **méthodes par défaut (publiques) avec leur implémentation**

Exercice Interface

- Donner le code Java associé à ce DCL



Corrigé exercice Interface




```
public interface IClasse {
   void m1();
}
```

Signature uniquement

```
public class Classe implement
IClasse {
   @Override
   void m1() {
         ….
   }
}
```

Implémentation de m1() pour Classe

```
public class Main {
   …
   IClasse unObjet;
   unObjet = new Classe();
   …
   unObjet.m1();
   …
}
```

Caractéristiques fondamentales de l'abstraction

- Le caractère abstrait d'un modèle doit notamment permettre de :
  - faciliter la compréhension du système étudié;
  - réduire la complexité du système étudié;
  - simuler le système étudié.

- Un modèle représente le système étudié et reproduit ses comportements.
  - *Comportement* = comment ils réagissent à des *événements* extérieurs (lancement d'un traitement, arrivée d'une nouvelle donnée…)

  - En plus des *objets du système*, on décrit les *objets du domaine*

Objets du système/ objets du domaine

Ex. Application de « Planning du bloc opératoire »

Objets du Système (de l'application)

- Les types **d'opérations** (durée moyenne, instruments et équipement requis, …),
- Les objets liés aux activités de **nettoyage** (différents déchets),
- Objets liés à la **stérilisation** des matériels,
- Eléments nécessaires aux **processus de soin** (examens pré opératoire, précautions dans la successions des opérations chirurgicales)

Objets du Domaine

- Les **personnels** (chirurgiens, infirmiers, techniciens,…),
- Les **ressources** (salles, équipement, …),
- Les plages d'ouverture et les RdV (**calendrier**)

Notion de base : 4 - Cohésion / couplage

COHESION = Esprit de famille, ciment

- Degré avec lequel les tâches d'un **module** sont fonctionnellement reliées entre elles. C'est la *colle interne* qui lie le tout de façon cohérente.

Questions à se poser pour définir la cohésion :

- Quel est le liant d'un module ?
- Quel est son objectif ?
- Fait-il une ou plusieurs choses ?
- Quelle est sa fonction au sein du système ?

Module en COO = une classe, un package, une méthode, un composant physique

- Au sein d'une classe : cohérence des relations entre **méthodes et attributs**

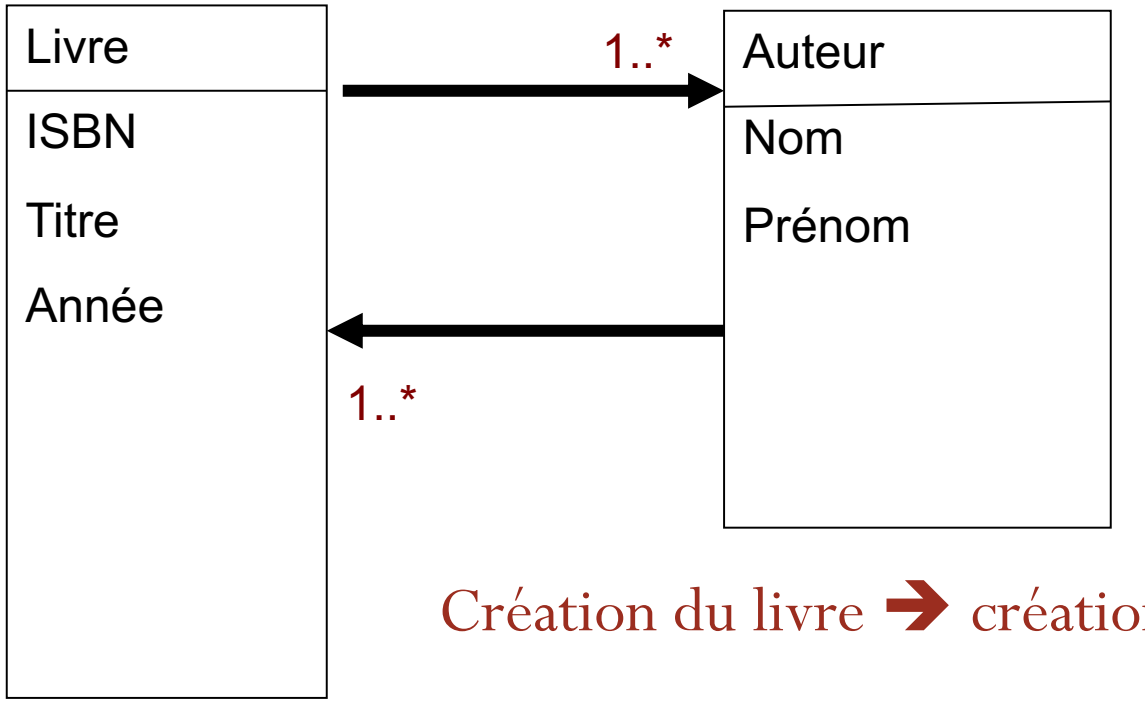
COUPLAGE = Dépendance

- Le couplage est la mesure **du degré d'interdépendance** entre les modules. Force de **l'interaction entre les modules** d'un système

- Comment les modules travaillent ensemble ?
- Qu'ont-ils besoin de savoir l'un sur l'autre ?
- Quand font-ils appel aux fonctionnalités de chacun ?

- Ex.: une classe qui **crée une instance d'une autre classe** = fort couplage
  - Ne peut pas être testée indépendamment de l'autre classe
- Autre ex.: une sous-classe dérivant d'une autre classe est en couplage fort avec son parent.

Ex. Couplage fort Livre / Auteur

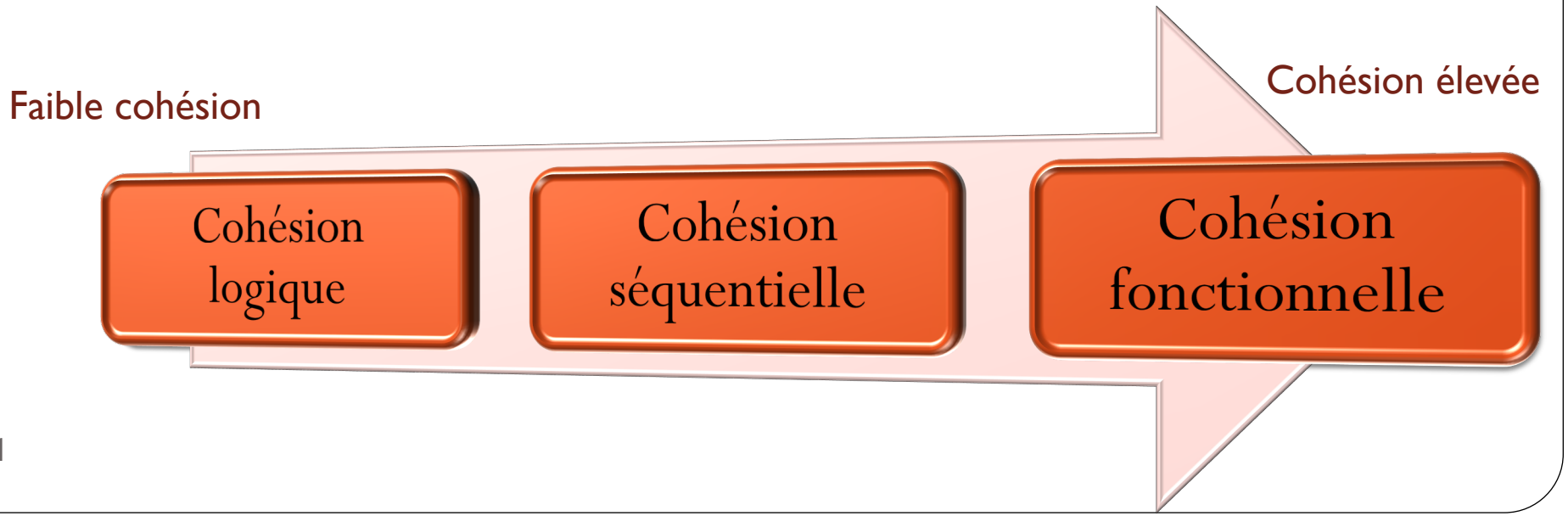


Création du livre ➔ création de l'auteur

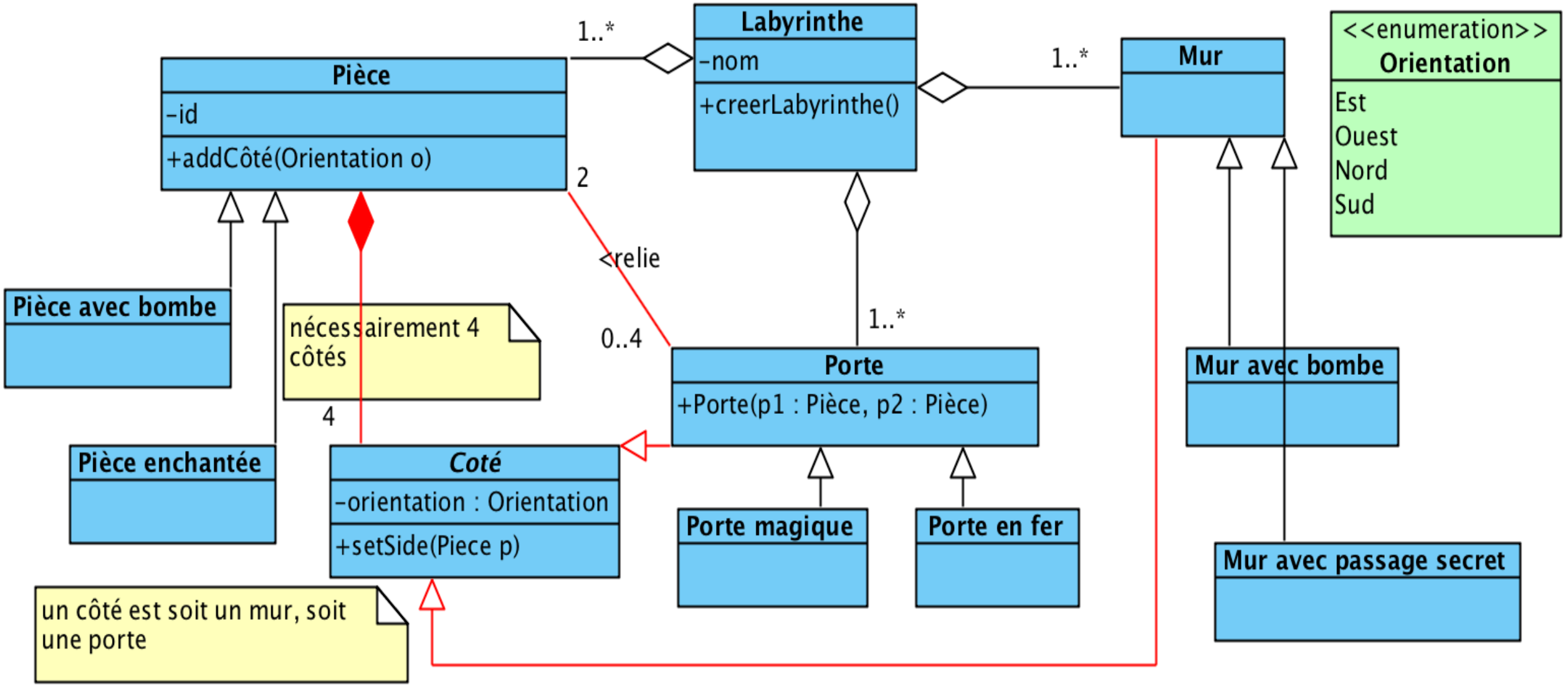
Un fort couplage n'est pas dramatique avec des composants **stables** (ex. java.util)

Que penser de la cohésion de ces composants ?

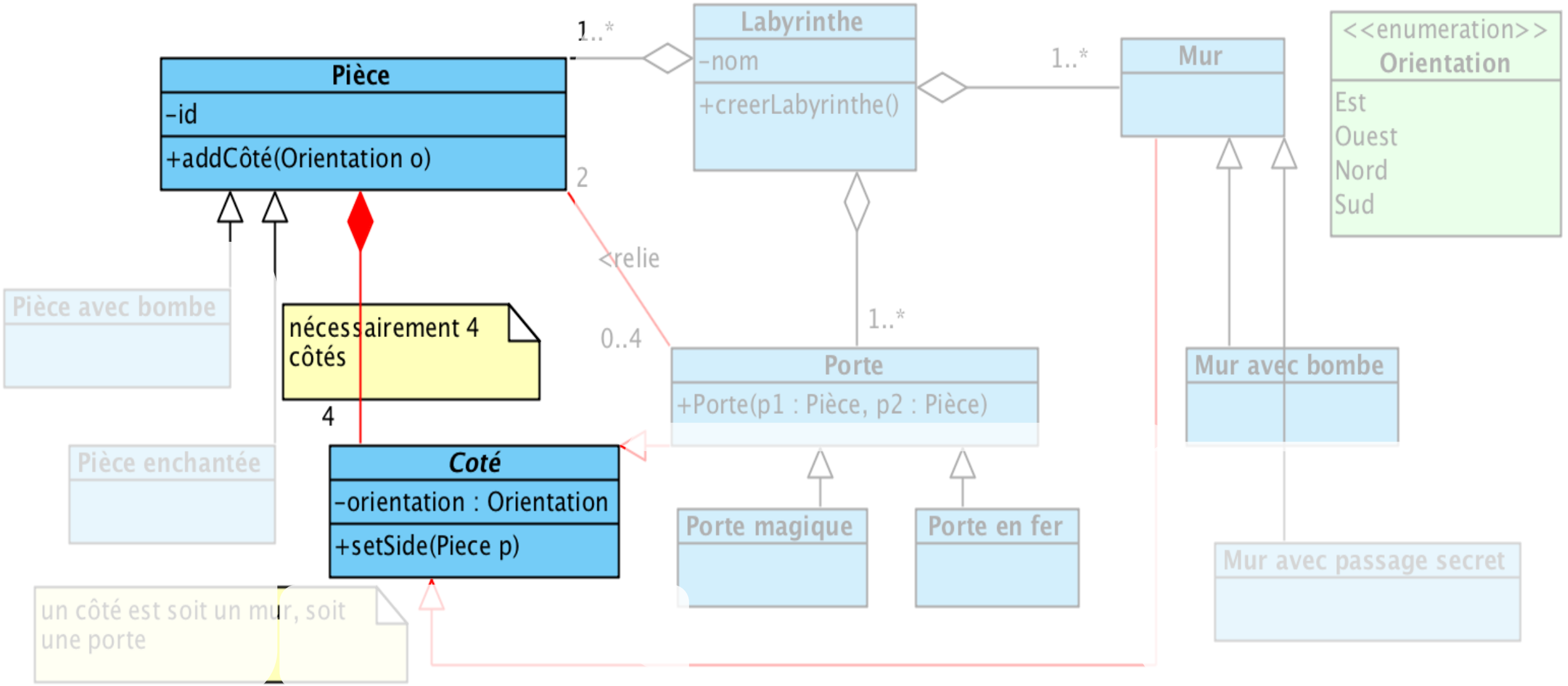
1. Un **composant A** avec une fonction qui lit les entrées d'un système à bande magnétique, une autre qui lit un disque, une dernière fonction qui lit les données du réseau
2. Un **composant B** avec un élément qui lit des entrées et génère des données qui viennent alimenter un autre élément du composant, pour sortir le résultat de la tâche du composant.



Labyrinthe : quels couplages voyez-vous ?



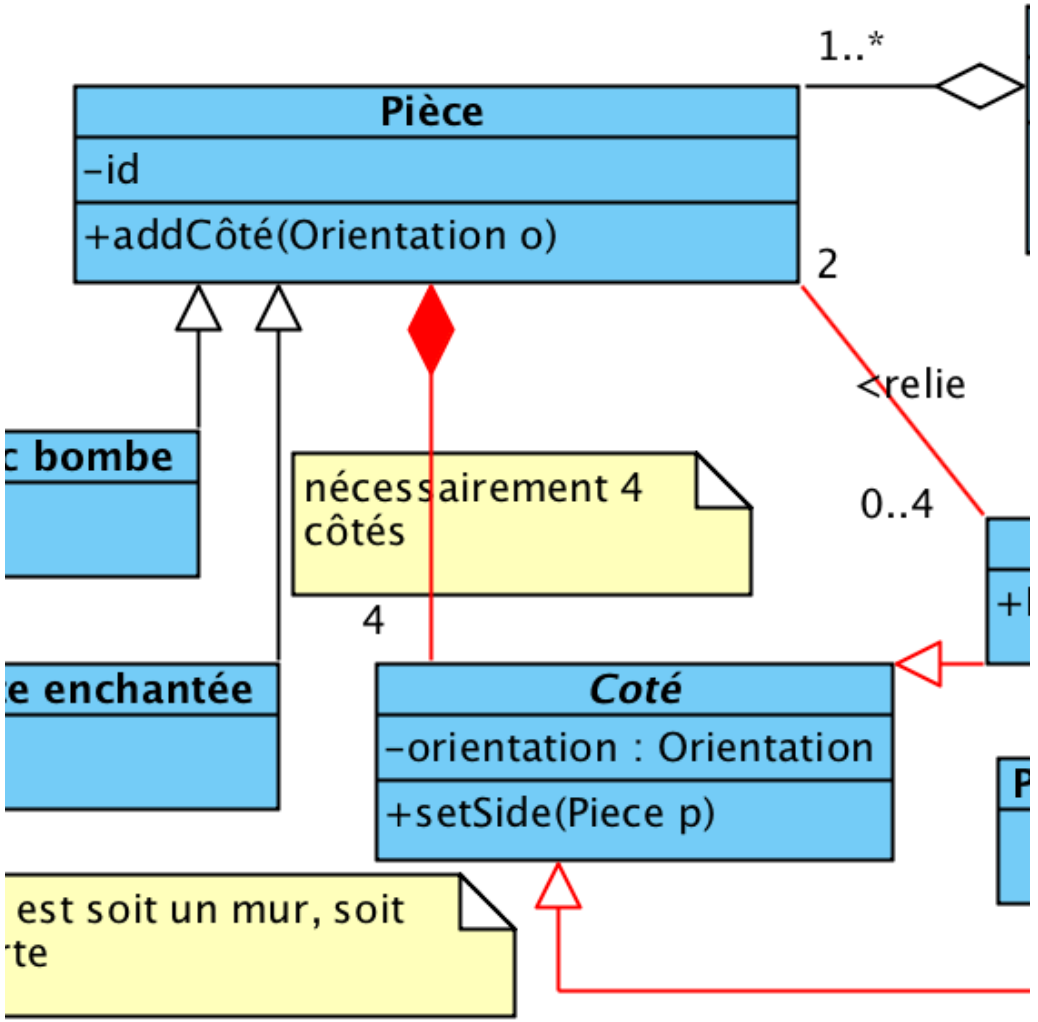
Labyrinthe : quels couplages voyez-vous ?



Labyrinthe : un couplage

- Par ex., entre la pièce et ses côtés
  - Méthodes setSide() et addCôté()

Il y aura toujours des **dépendances** entre certains modules de l'application : l'important est de les identifier et les rendre bien visible.



Cohésion / Couplage

- L'enjeu d'une **bonne conception** consiste à bien distribuer les responsabilités dans l'application.

- Les **objets** et les **modules** (package, librairies) sont des briques élémentaires permettant de réaliser des architectures avec :
  - De multiples niveaux d'abstractions,
  - De multiples modèles de calcul,
  - Des composants réutilisables

- On va chercher à avoir une **forte cohésion** entre les composants et un **faible couplage**

Cohésion / Couplage (suite)

- **Les avantages d'un faible couplage** :
  - les modifications apportées à une partie vont moins se propager dans le reste du système
  - les composants sont faciles à remplacer et à réutiliser.

- Les classes abstraites et l'héritage permettent de minimiser les interdépendances en **contrôlant le niveau de détails des services visibles** par les clients d'un objet.
  - DEF « clients d'un objet » ?

Objets qui **utilisent une classe**

par ex. les « clients » d'un objet Mur sont la classe « Labyrinthe » et la classe « Coté »

Quels sont les inconvénients d'une **faible cohésion** ?

Que penser d'une classe avec 100 méthodes et des milliers de lignes de codes ?

Manque certain de cohérence entre les variables, les traitements

Différents domaines sont couverts

Faible cohésion

Une **cohésion** médiocre altère :

- la compréhension,
- la réutilisation,
- la maintenabilité

Mais surtout : **le code est fragile**

- Il subit **toute sorte de changements** très fréquemment, comme il est très vaste
- Trop d'objets ont besoin de lui
- Risque d'introduire un **fort couplage** (car trop d'objets auront besoin de lui, trop d'interactions…)

Inconvénients d'un trop fort **couplage** ?

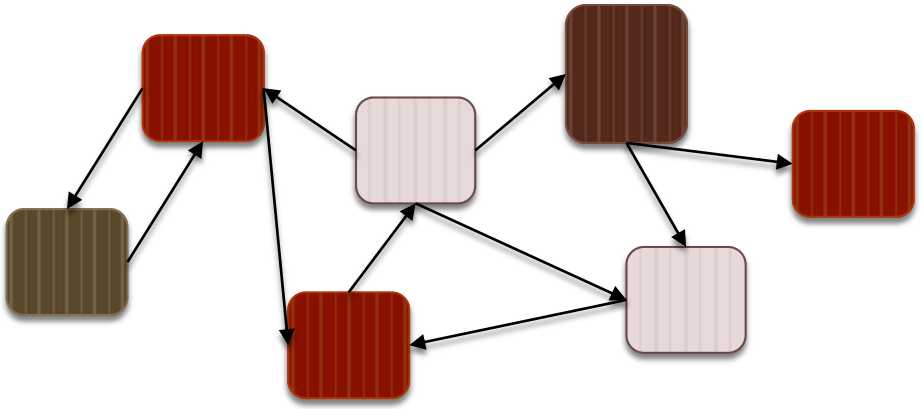
Fort couplage

- Un couplage trop élevé ? Une **assiette de spaghettis**
  - Maintenance difficile, voire impossible
  - Lisibilité faible
- C'est parfois volontaire : principe d'**obfuscation**
  - Code rendu illisible pour protéger ses sources de rétro-ingénierie

NOTA : Quelque soit le niveau de couplage d'un logiciel, il est important de savoir **quels modules sont couplés**.

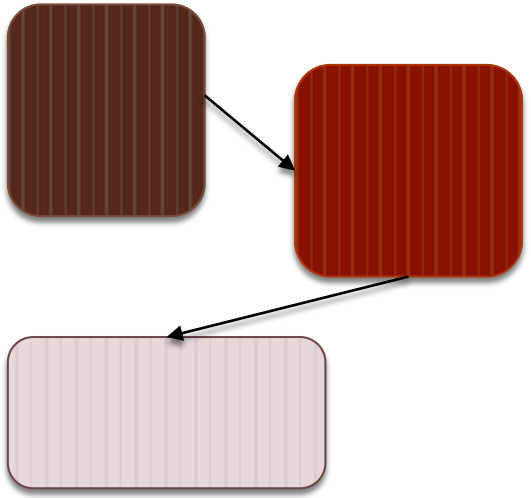
Sinon le développeur va passer du temps à tester tous les modules pour déterminer si chacun d'entre eux est affecté ou non par un changement.

2 notions antinomiques ?



- Chaque composant fait une seule chose
- Faible niveau d'abstraction
- Relations complexes
- Ultra réutilisable

→ Fort couplage



- De gros composants faiblement couplés
- Chacun plus difficile à comprendre
- Faible niveau de réutilisabilité

→ Faible cohésion interne

→ Trouver le bon équilibre

Forte cohésion : quelques règles

- Regrouper les éléments **en forte relation** dans un même package
- Regrouper les classes qui rendent des **services de même nature** aux utilisateurs
- Isoler les **classes stables** de celles qui risquent **d'évoluer** au cours du projet
- Isoler les classes **métiers** des classes **applicatives**
- Distinguer les classes dont les objets ont **des durées de vie différentes**

Un module hautement cohésif exécute une tâche ou atteint un objectif unique
« faire une chose et la faire bien »

Faible couplage : les règles

- Préférer s'adresser à une **interface**, pas à une **implémentation**
  - Via l'interface, le client ne sait pas quelle sera l'implémentation, il sait juste ce qu'il *peut* demander de faire à la classe
  - **Couplage** plus faible

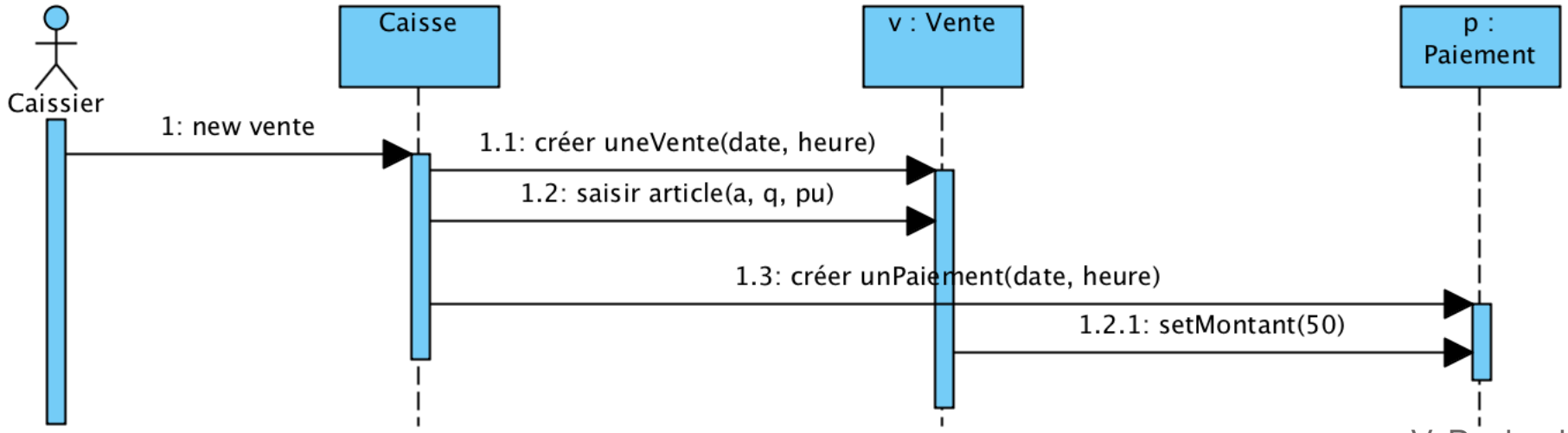
- Ne pas ajouter plus de dépendances que nécessaire

Illustration : ne mettre que les dépendances nécessaires

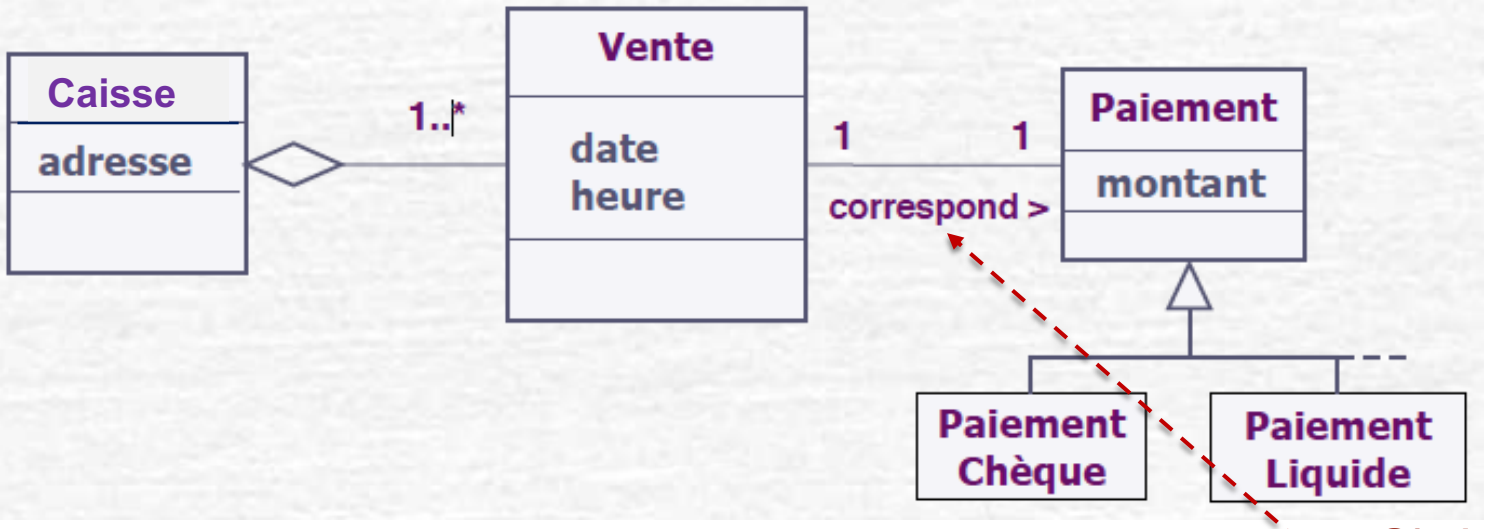
*Si c'est la Caisse qui crée le paiement, on **ajoute** un couplage de Caisse à Paiement, qui n'existait pas dans le DCL*

Imaginons qu'on ait à déterminer quel composant gère le paiement

solution 1 :



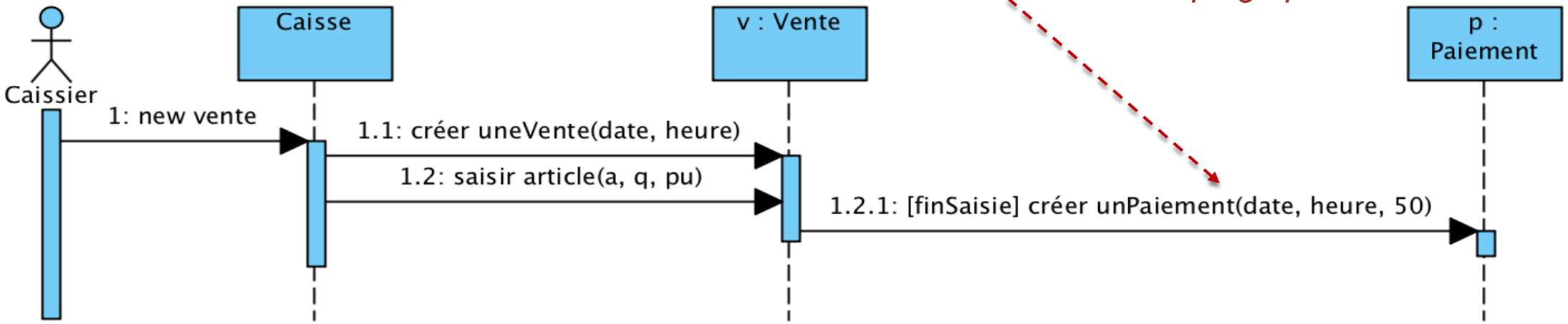
Faible couplage : règles (2)



*Le Paiement est plus **naturellement** lié à la Vente, qui peut par ailleurs avoir d'autres méthodes liées aux Paiements : choixTypePaiement, encaisser (clore la Vente), ...*

solution 2

Si c'est la vente qui crée le paiement, on traduit le couplage présent dans le DCL



Synthèse : notions de base de la conception

Compléter les **informations manquantes** !

- Des *xxxxxxxxxxx xxxxxx* :
  - Favorisent la lisibilité et la compréhension du code
  - Rendent le code plus facile à maintenir et à tester
  - Permettent le contrôle des impacts lors de changements dans le code

- Une classe a une forte *yyyy* si tous ses attributs sont employés par ses méthodes.

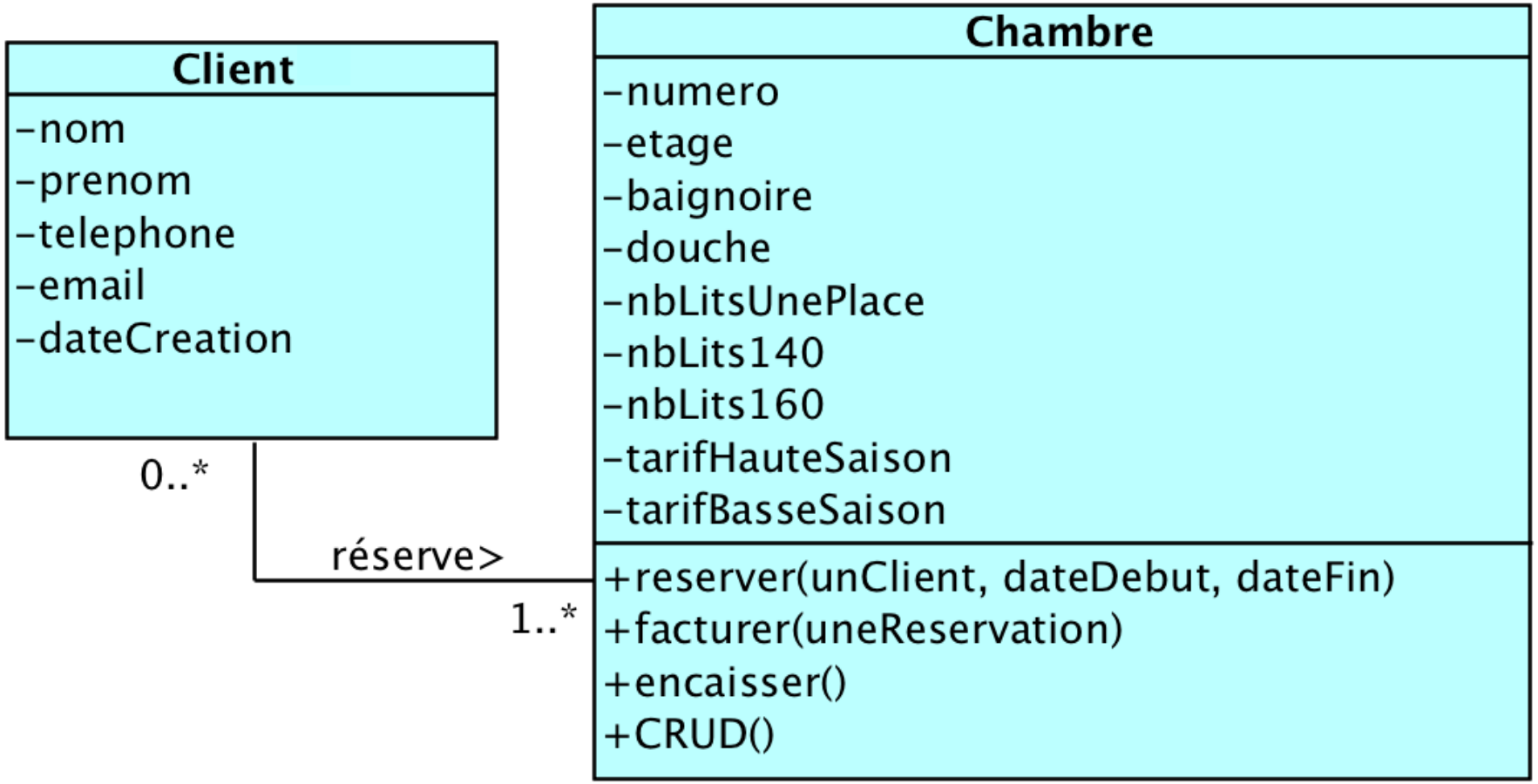
- Deux classes sont *zzzz* si un changement dans l'une exige des changements dans l'autre.

Exercice : Hôtel

- Par exemple, les hôtels génèrent des revenus en louant leurs chambres aux clients. Le concept de chambre est susceptible d'être représenté quelque part dans le système logiciel de réservation d'un hôtel, avec son numéro, étage, nombre de lits (1 place, 2 places), baignoire ou douche.
- Il peut être pratique d'utiliser la classe Chambre pour collecter et stocker les **réservations** effectuées par les clients de l'hôtel, ainsi que les données sur les **revenus** générés par ces réservations.

Donner le **diagramme de classe** avec cette conception de la classe Chambre.

Une solution



Exercice : Hôtel (suite)

- Cependant, on apprend que l'hôtel génère des revenus d'autres façons, par exemple, en servant des repas à des personnes qui ne sont pas des clients résidents. Doit-on modifier le modèle ?

- **Proposer un autre DCL**

Différents domaines sont couverts dans la classe Chambre !

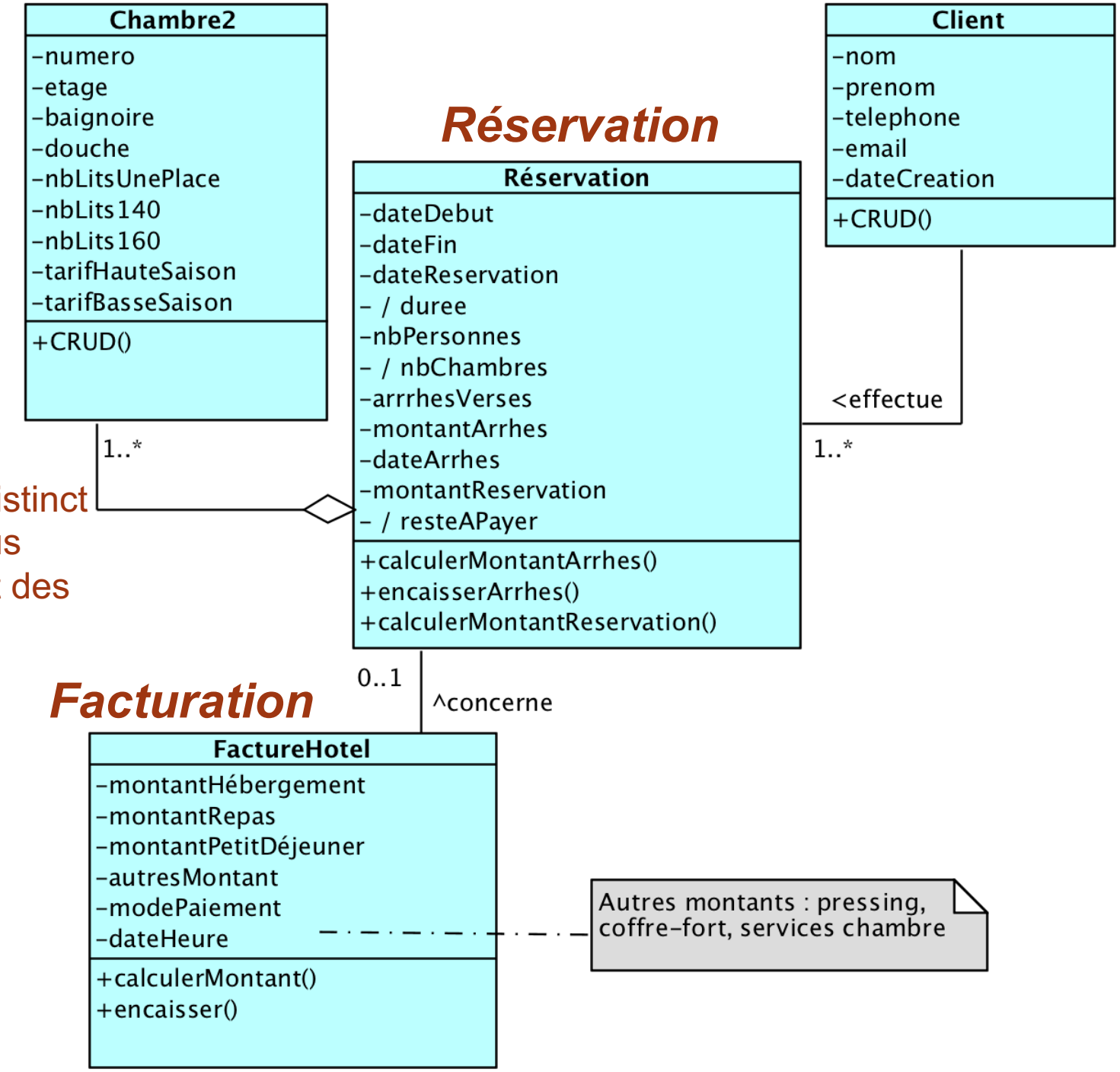
- ***Description***
- ***Réservation***
- ***Facturation***

Description

Avoir un module de facturation / paiement distinct est mieux puisque l'hôtel génère des revenus indépendamment des chambres, en servant des repas à des personnes extérieures.

➔ ***Meilleure cohésion des classes***

Réservation



Facturation

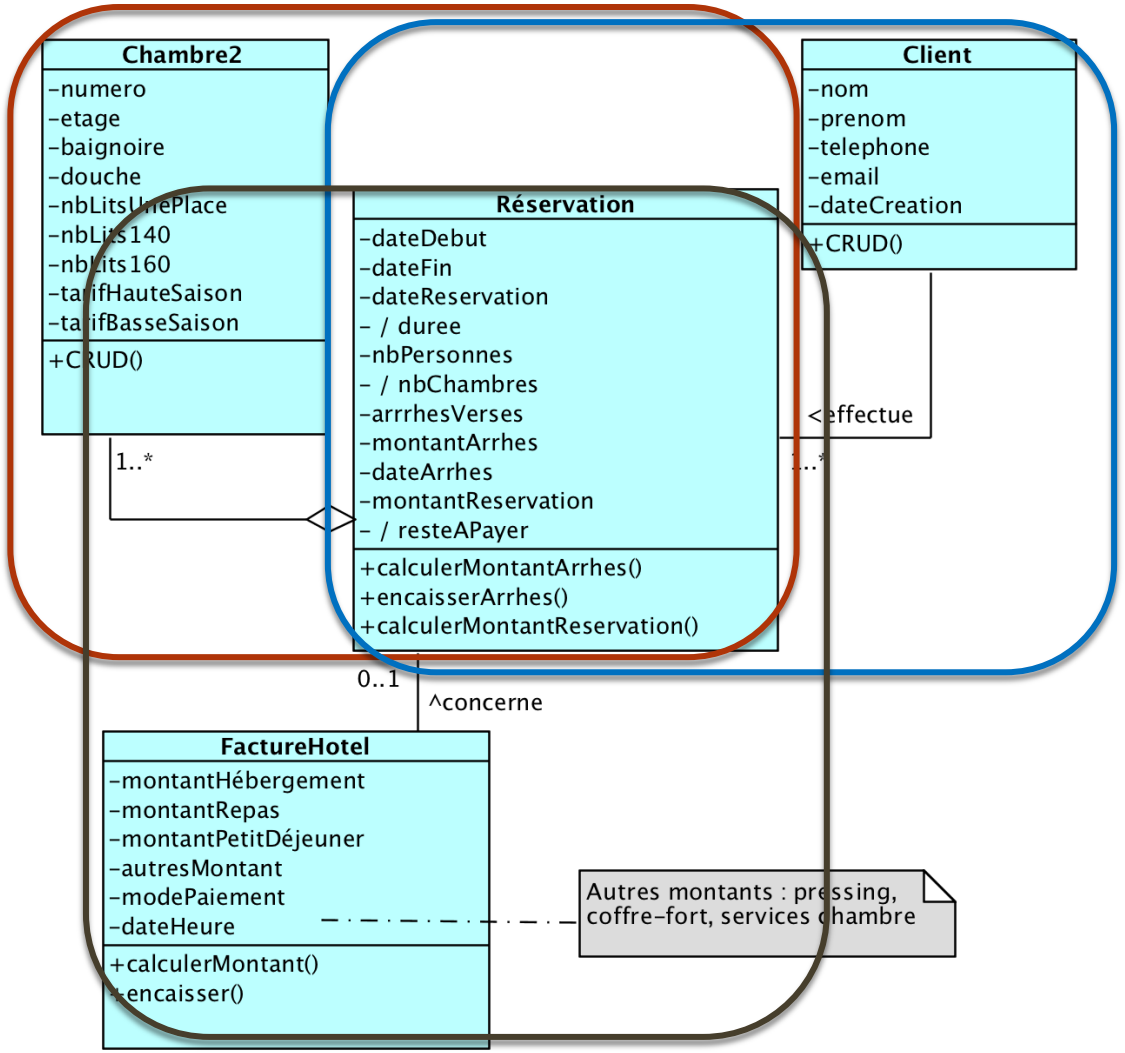
Couplage

Dans la solution précédente, **quels sont les couplages** qui existent entre les composants ?

- **Réservation** est liée à des instances de **Chambre**.
- Une **chambre** est reliée à plusieurs instances de **Réservation**.

- La **réservation** est liée à un **Client**, et réciproquement.

- Une **facture** est reliée, **ou pas**, à une **réservation**.



Quels sont les impacts des changements ?

Entre les composants reliés

1. Par ex. si le tarif d'une chambre change ?
2. Si la date de la réservation change ?
3. Si le téléphone du Client change ?

Corrigé :

1. Les futures réservations en tiendront compte (pas les anciennes, qui ont un montant de la Réservation calculé avec le tarif à la date de la Réservation) : la méthode *calculerMontantRéservation()* va chercher le tarif correspondant aux dates.
2. Nouvelles dates (avant le séjour) : les chambres affectées peuvent changer (choix du gérant), le nb de chambres aussi, la facture ne changera pas (car créée en fin de séjour). Aucun impact sur le Client.
3. Aucun impact sur les autres composants.

2 types d'impact : structurel ou dynamique (nouveau calcul)

STOP ! Fiche TD

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 grosses _____ ou _____

   Se limiter à _____ en général par _____

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

Programmer par fonctionnalité

- L'application est programmée de telle manière que l'ajout de fonctionnalité ne modifie pas le code source existant.
- Les plugins sont le premier concept permettant de réaliser cela. Mais il faut une solution pour pouvoir modifier le déroulement de fonctionnalités qui existent déjà, **sans modifier leur code source**.
- Ceci peut être réalisé par un pré-traitement sur les arguments des fonctions de la fonctionnalité précédente, et/ou un post-traitement sur leurs sorties.
- Il est possible de créer des applications complexes sur ce principe en utilisant le FyHooks framework

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)

2. **Principe DRY** - **Don't Repeat Yourself** : jamais de copier/coller de code

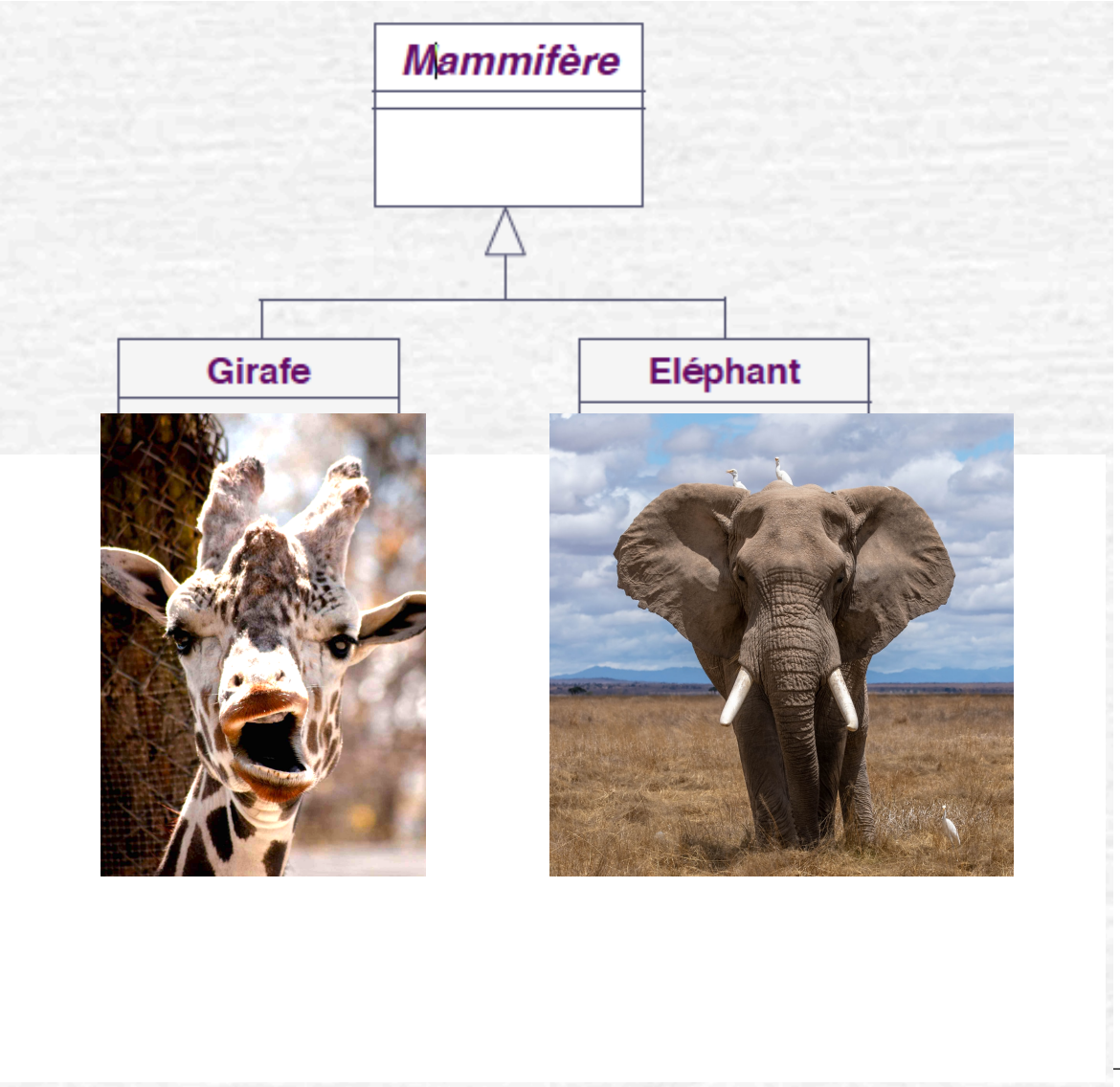
3. Ne jamais dériver une classe en n'exploitant que **certains attributs** et **méthodes**

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 »

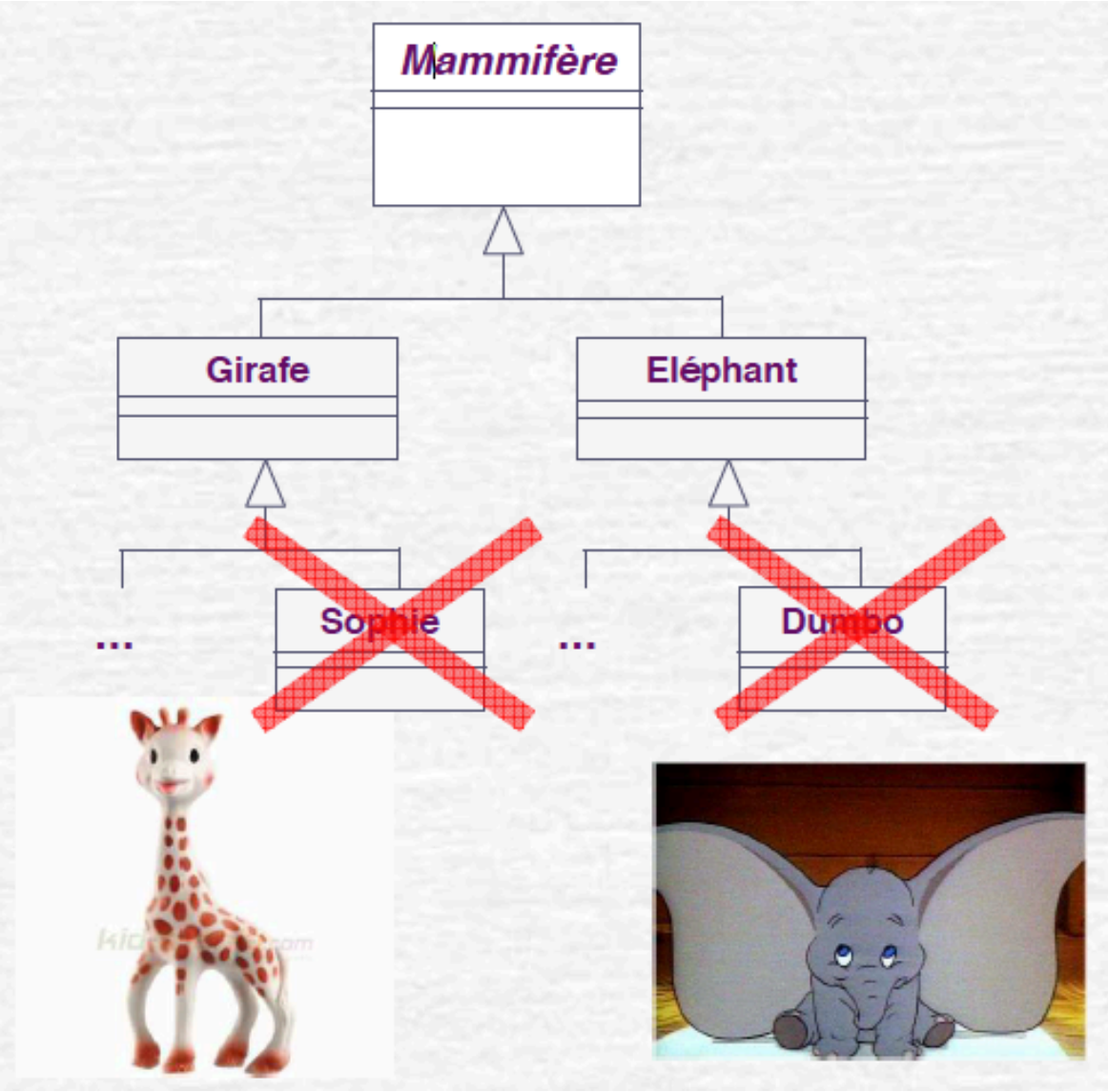
Imaginons un logiciel de simulation d'une **réserve animalière**, avec des girafes et des éléphants.



« 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* ??



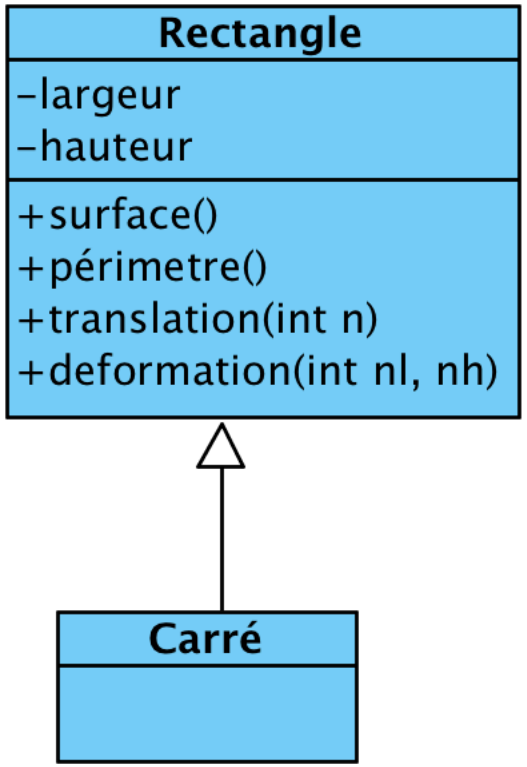
Règles en cas d'héritage

- 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 plus larges que celles héritées

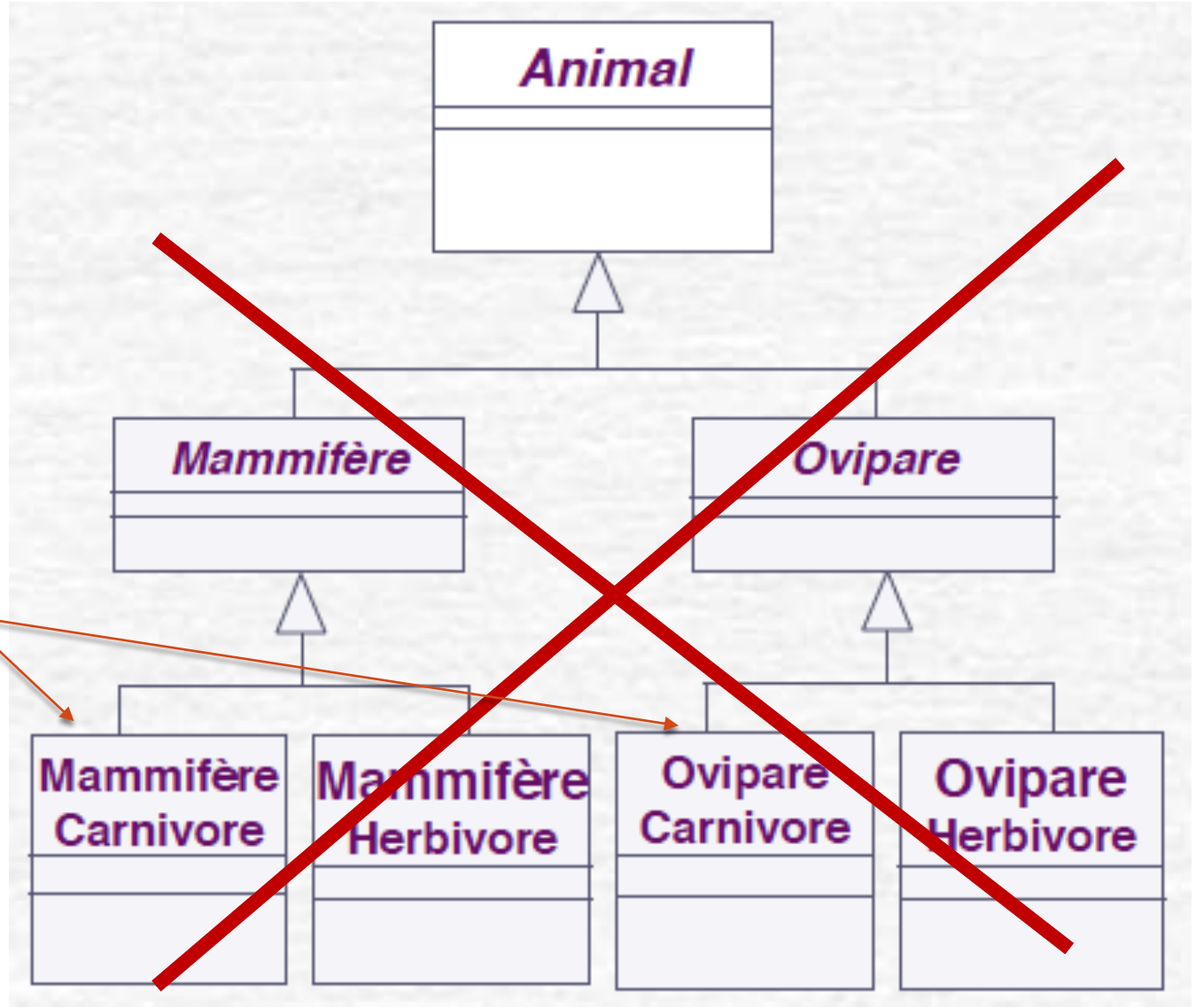
Exercice pré/post conditions

- 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

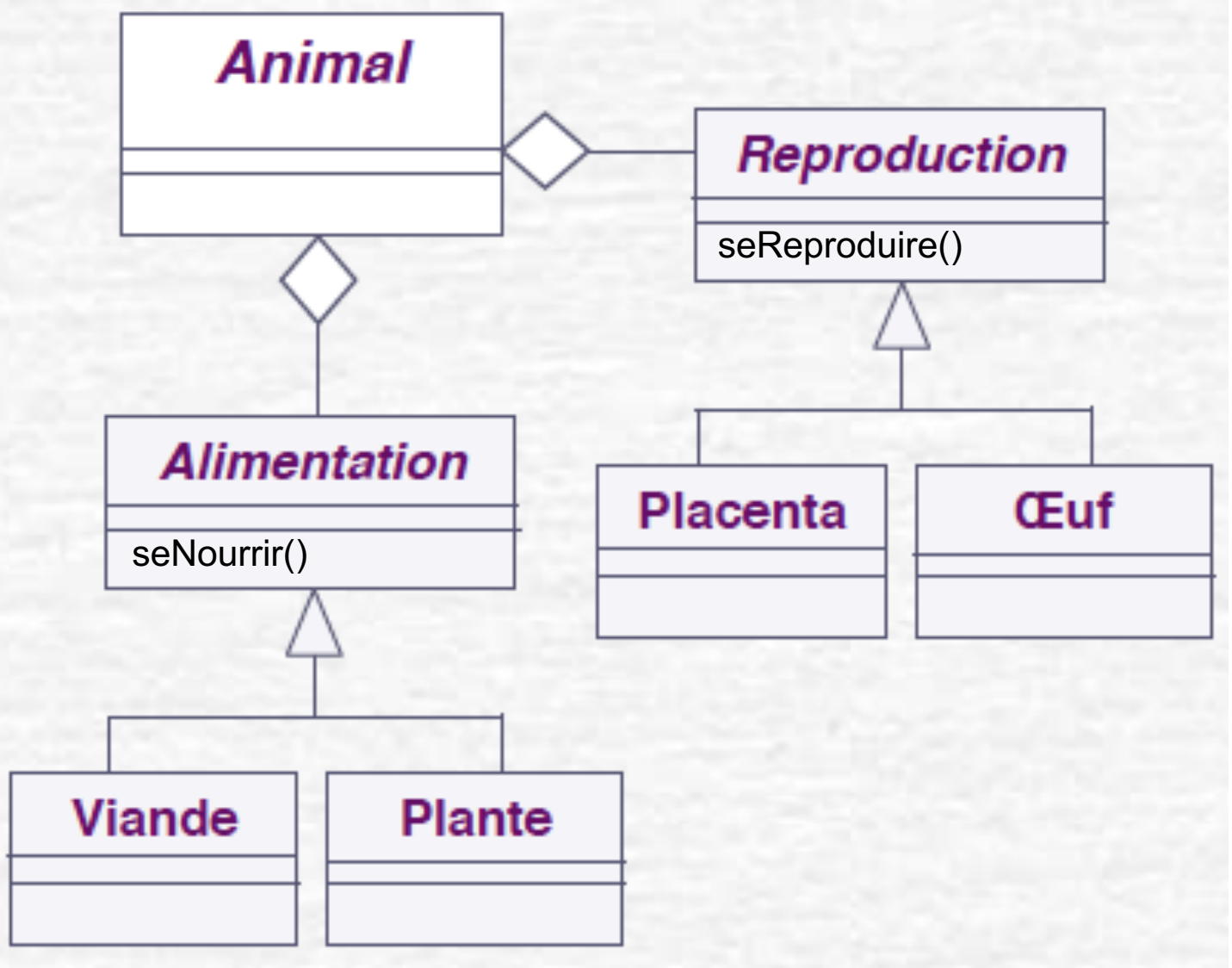
Règle#4 :« Préférer la composition à l'héritage »

On considère qu'un Animal possède 2 caractéristiques intrinsèques :

- un mode d'Alimentation
- un mode de Reproduction

On choisit lequel à chaque instanciation d'Animal :

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

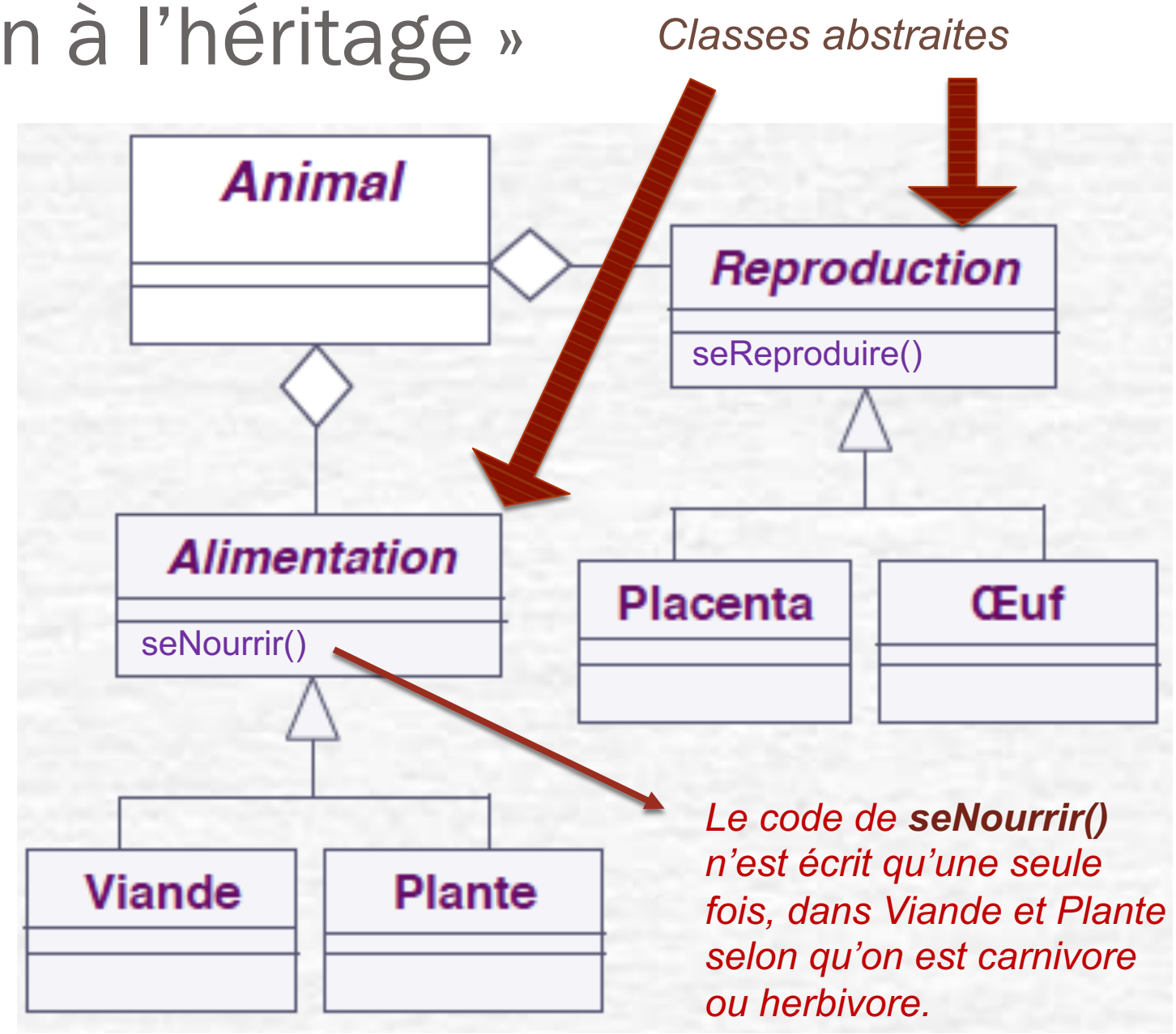


Préférer la composition à l'héritage »

Encore appelé principe d'indirection ou de délégation :

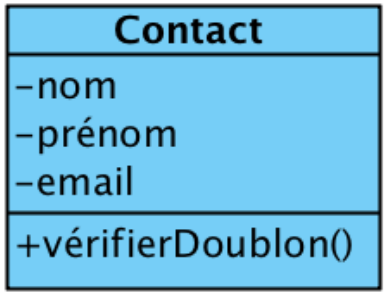
on délègue à la classe **Alimentation** le comportement seNourrir() de l'Animal

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



Conception / Codage : importance des structures de données

- Certaines structures de données peuvent traduire et **simplifier les méthodes imaginées en phase d'analyse**
- Ex. : une classe d'analyse **Contact** avec une méthode **vérifierDoublon()**
  - si l'ID est défini par nom+prénom : il suffit de placer les objets Contact dans un conteneur *set* de Java (*set* ne tolère pas les doublons)



- Ex.: pour un **contrôle d'accès de salles**, on choisira une HashTable avec les numéros de salle (clef) et le code d'accès (valeur: true/false).

- Il est donc important de **bien connaître** les structures de données évoluées : conteneurs Java, arbres bicolores, skip-list, etc.

Importance des tests de régression

Les tests de régression sont **obligatoires** si votre code :

- Doit être utilisé longtemps ;
- Sera utilisé dans des environnements différents : compilateur, processeur, version de langage, langue de l'utilisateur… ;
- Risque d'évoluer.

CONSEILS

- Faire des tests **courts en temps d'exécution** ;
- **Indépendants** les uns des autres, et donc **exécutables dans n'importe quel ordre** ;
- Vérifiant chacun le **minimum** de choses ;
- Documentant ce qu'ils font et pourquoi c'est le résultat attendu ;
- Stockés chacun **dans un fichier séparé** afin de pouvoir facilement utiliser *git bisect** pour trouver automatiquement quand un bug a été introduit.

*https://runebook.dev/fr/docs/git/git-bisect-lk2009