

# Chap.1-1 – Paradigmes de Conception

**V. Deslandres** ©

**S1-10 – Qualité de Développement-1**

BUT Informatique - ASPE

IUT de Lyon - Université Lyon 1

# Présentation réciproque

- **V. Deslandres**

- Enseignante-chercheure à l'IUT de Lyon - site Doua depuis 2003
- 5 ans chez Renault - Direction de la Recherche
- Laboratoire LIRIS, équipe IA (modélisation multi-agents)

## Tour de salle

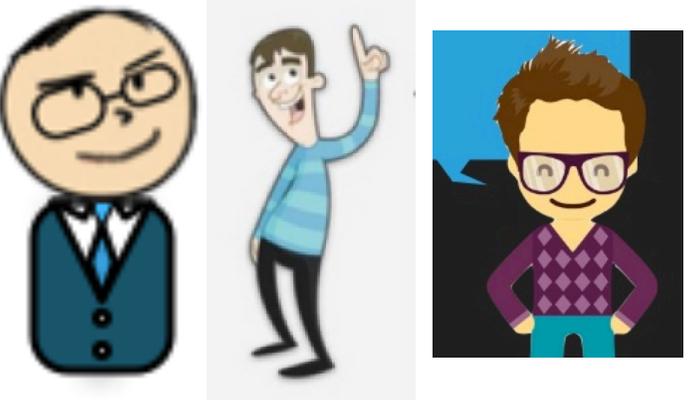
- Quelle formation antérieure ?
- Connaissance d'UML ? Expériences de développement de projets ?



# Présentation du module

- **Qualité de Développement-1 (24h)**
  - Qualité de Conception (18h) : principes SOLID et Design Patterns
  - Qualité de production logicielle : refactoring, gestion de versions et travail collaboratif avec GitLab (6h)
- Sera suivi d'un module **Qualité de Développement-2 (12h)**
  - Tests unitaires, couverture de tests, CI/CD
- **Evaluation**
  - QROC à mi-parcours (40%)
  - DS final (60%)
  - Bonus/malus de 2pts max sur le sérieux de la participation en séance

# Analyse & Conception : 2 phases distinctes



# Conception fonctionnelle vs. Technique

- ▶ **Conception fonctionnelle ou Analyse** : « dissection » du problème

QUOI

- ▶ 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.

- ▶ **Conception technique** : fabrication d'une solution informatique

COMMENT

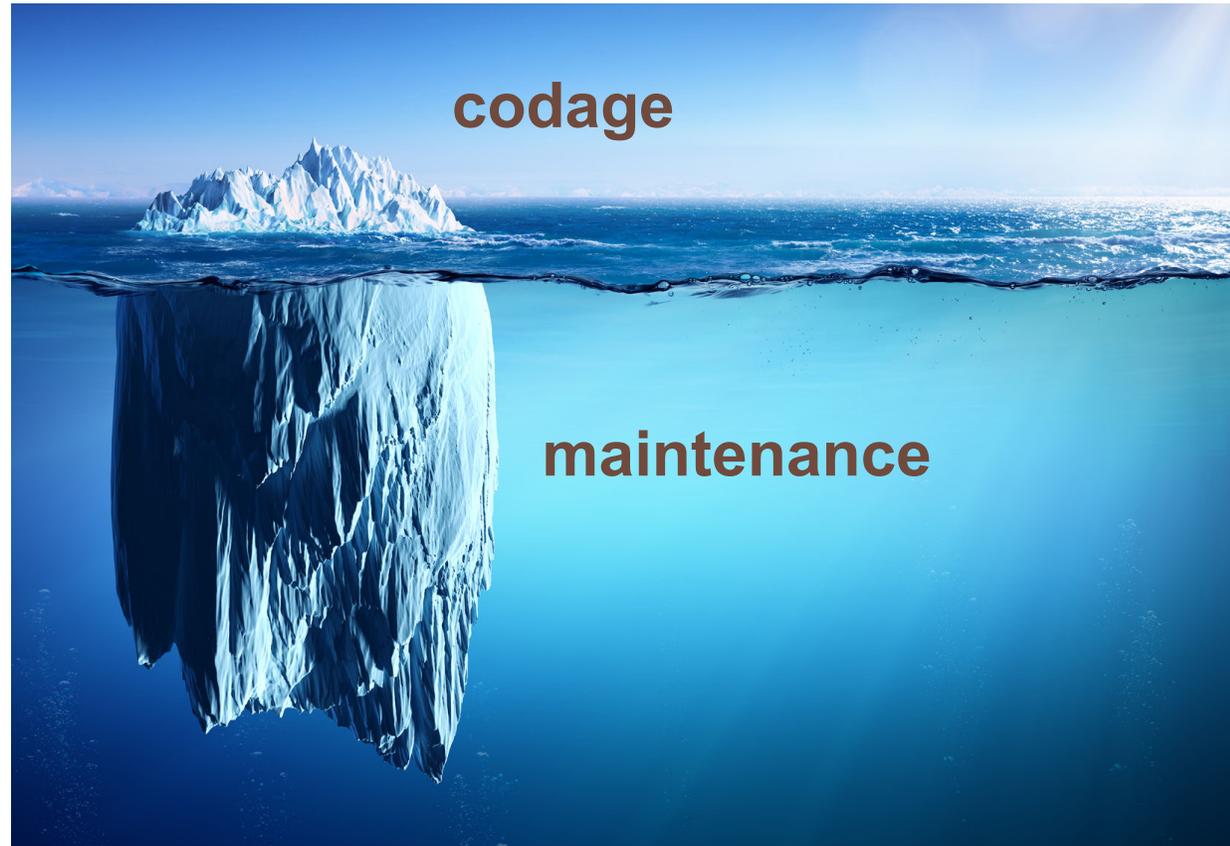
- ▶ Détails d'implémentation réelles (IHM, stockage, accès, sécurité, fréquence/débits, performances)
- ▶ Premiers livrables de code
- ▶ Ex.: Découper le trajet en tronçons ; lister les trajets demandés en moins de 2 sec. ; identifier un Conducteur par son nom/prénom/dateNaissance obtenu de son permis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

# Pourquoi bien concevoir, est-ce important ?

Concevoir un logiciel non pas pour son utilisation IMMEDIATE, mais qui sera MODIFIÉ, qui pourra ETRE ÉTENDU.

Conception = définir un logiciel facile à faire évoluer

Formaliser les choses de façon générique. Toujours anticiper leur possible évolution.



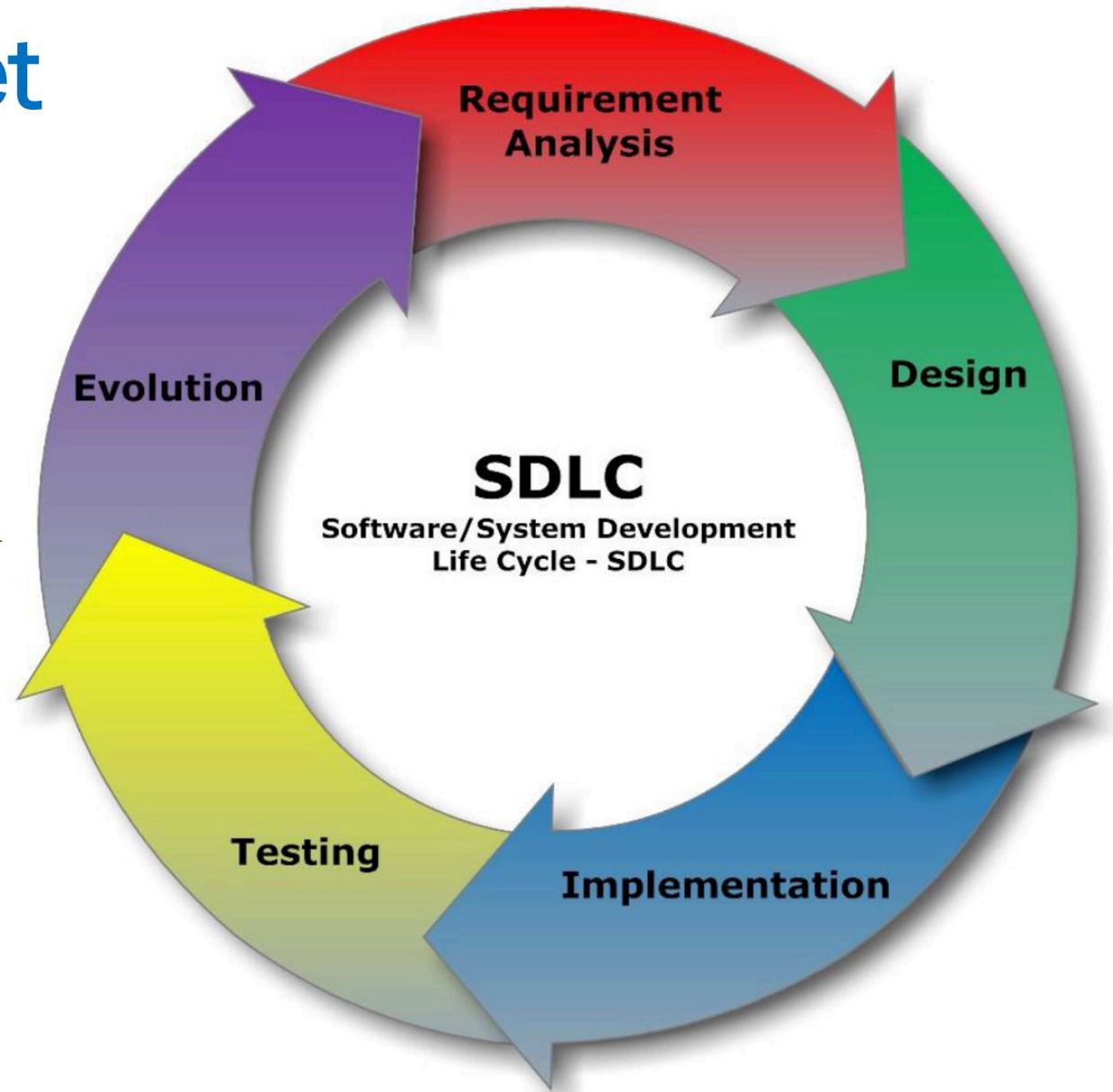
# Cycle de vie d'un projet

Durée de vie moyenne : 7 ans

À votre avis ?

% du temps consacré à la **création**  
d'un logiciel ? (test compris)

% consacré à son **évolution** ?



# Importance des tests de régression

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

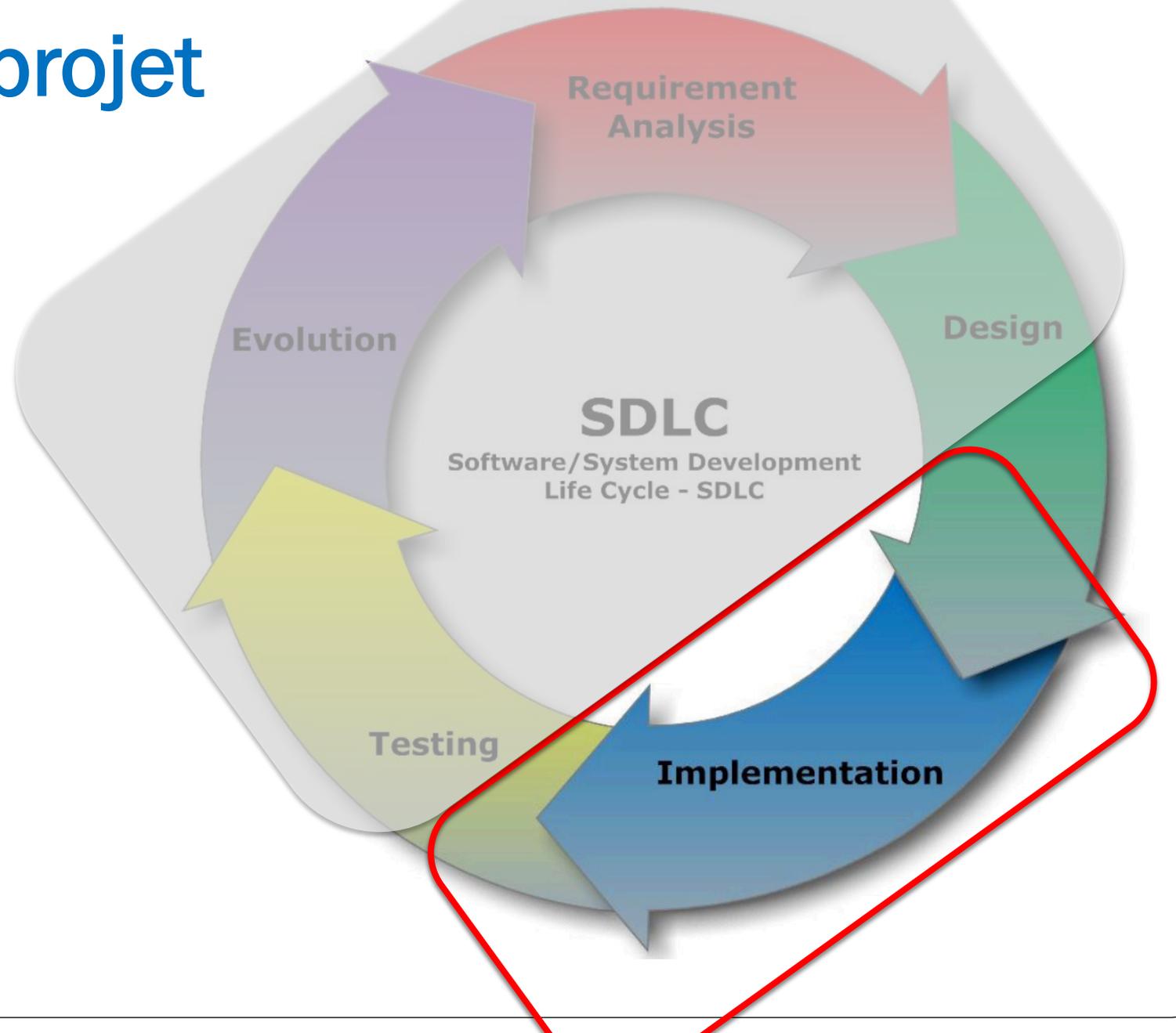
- Doit être utilisé longtemps ;
- Sera utilisé dans des environnements différents : compilateur, processeur, version de langage, langue de l'utilisateur... ;
- Risque d'évoluer.

## 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.

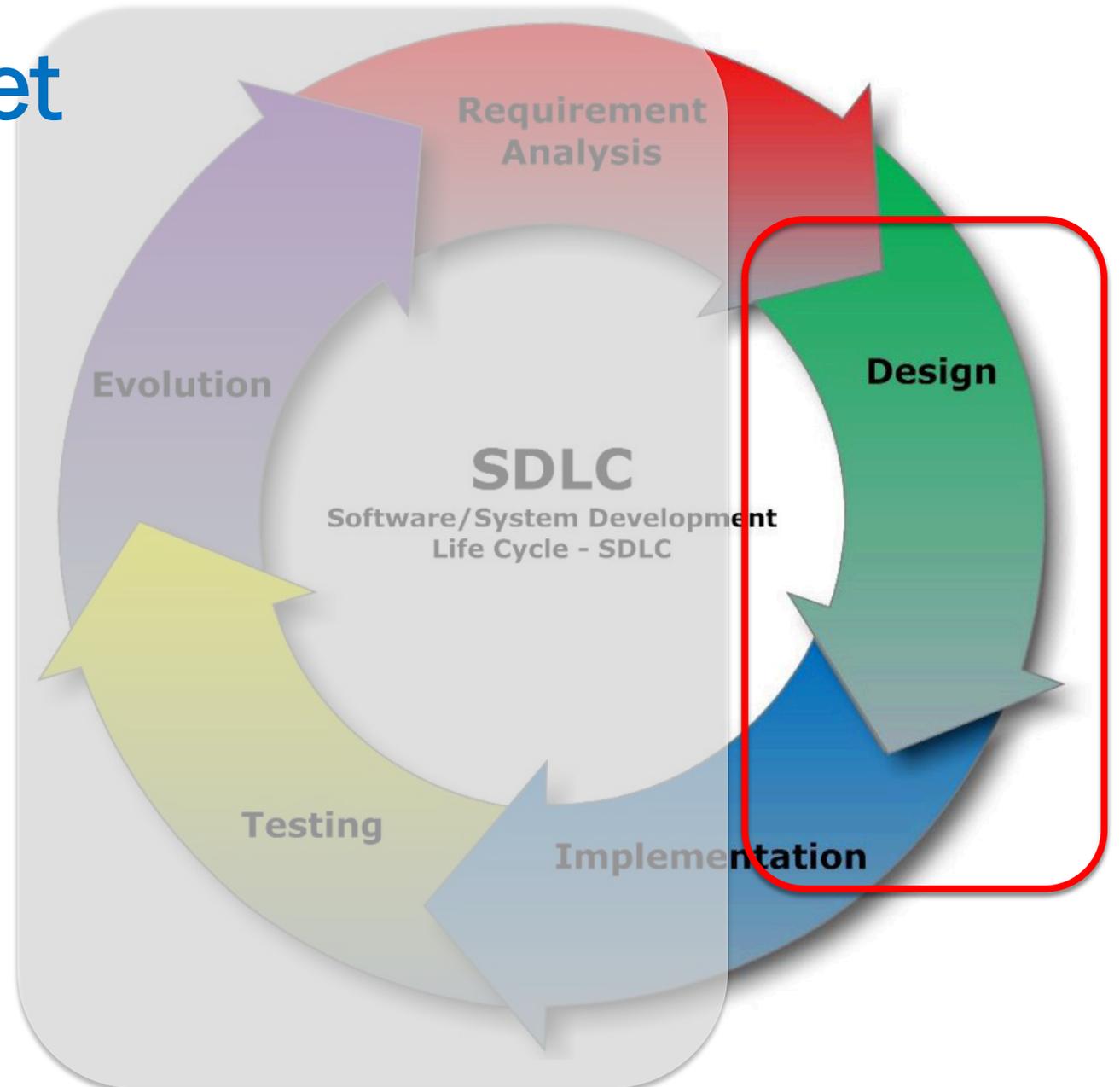
# Cycle de vie d'un projet

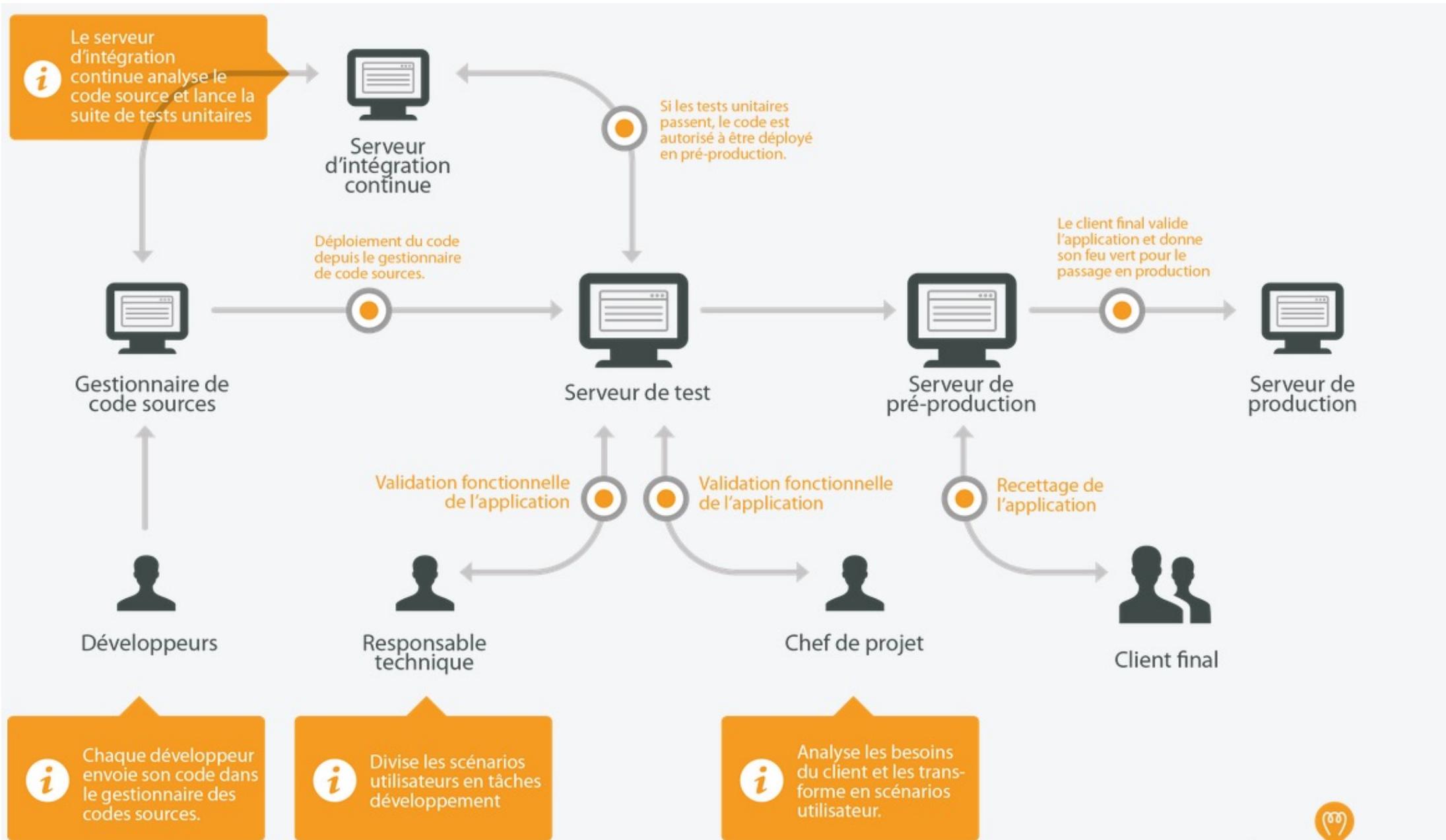
Partie vue à l'IUT



# Cycle de vie d'un projet

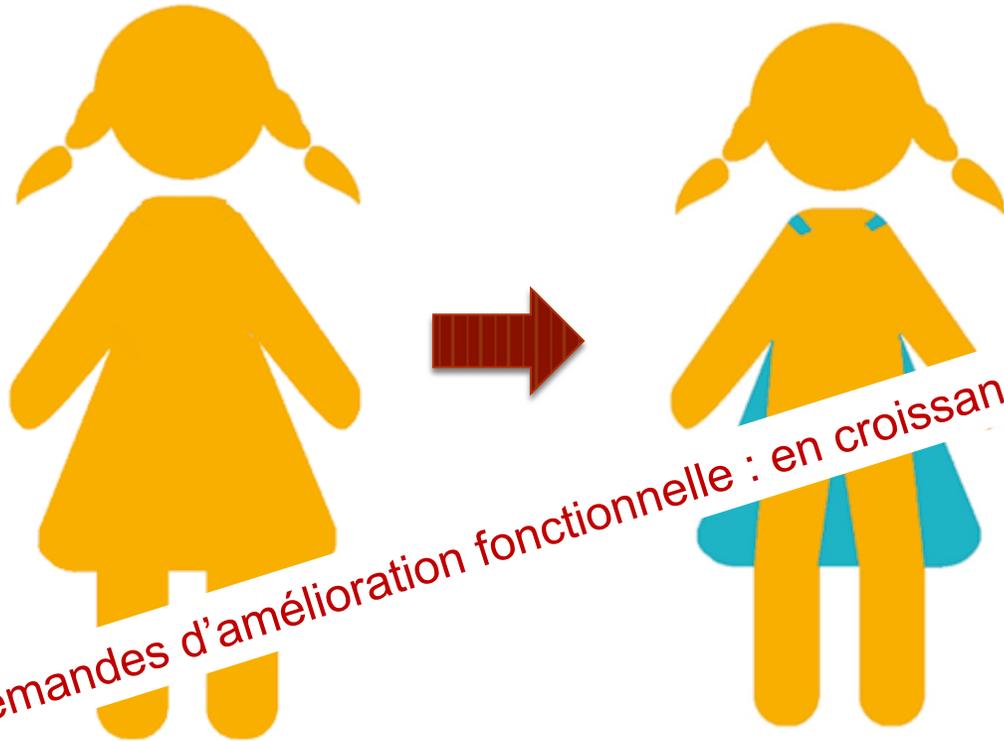
Partie concernée par ce module





# Evolution : 2 types

- Evolutions fonctionnelles



Demands d'amélioration fonctionnelle : en croissance !

- Evolutions technologiques



Les plus fréquentes !



# Ex de besoins technologiques : déploiement sur différents OS, migration d'OS

78% des répondants à leur enquête utilisent de 1 à 4 OS différents

(Enquête PUPPET 2018, 3000 professionnels de l'IT, enquête annuelle internationale)

# of OSes Selected	# of Responses	% of Total
0	25	1%
1	1,035	22%
2	1,038	23%
3	883	19%
4	636	14%
5	420	9%
6	234	5%
7	139	3%
8	86	2%

78%

78% of respondents are widely deployed on 1-4 different operating systems.

# Ex. de fonctionnalités techniques

- Mise à jour automatique du DNS
- Empêcher la duplication des noms de déploiement
- Échec rapide si opération non autorisée
- Mise à jour automatique des variables d'environnement
- Retour à la version précédente (rollback) manuelle
- Auto-rollback en cas d'échec d'une nouvelle livraison
- Améliorer les performances de copie de BD
- Etc.

Structures de données après  $n$  modifications, extensions et évolutions technologiques...

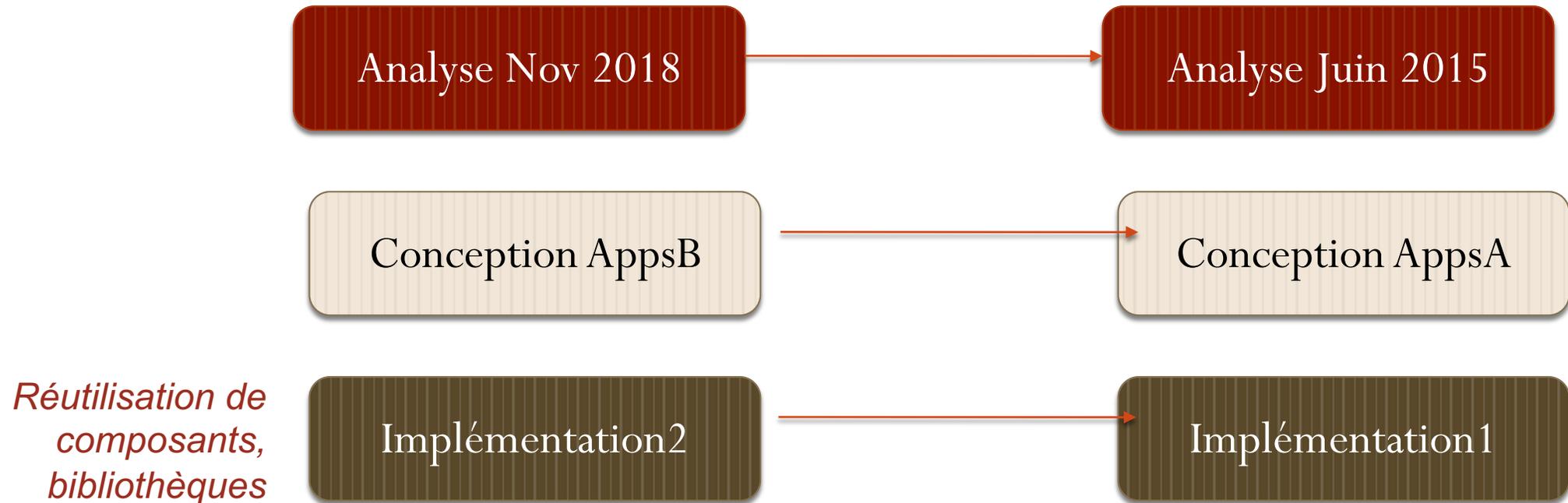


# Bonne conception : découper

- *Segmenter* son application en **modules le plus indépendants possibles**
  - En termes de code, de fonctionnalités, pas forcément en termes de données
- On réduit les problèmes pour arriver à des composants **plus locaux, plus ciblés**
- Les classes qui changent **au même rythme** seront placées dans un même module.
- *Segmenter* diminue aussi le **périmètre** des tests
  - Modifier des lignes de code → génération de bugs
  - En cas d'évolution, comme on a découpé une appli en gros modules, on ne vérifie / teste que le module où a eu lieu la modification
  - (Et éventuellement toutes les applis Clientes qui utilisent ce module)
- Parvenir à **séparer son application en modules** n'est pas une chose facile
  - Les diagrammes des cas d'utilisations constituent une bonne approche

# Autre objectif d'une conception générique : la réutilisation

- On peut « réutiliser » à différents niveaux et de différentes sources :



- A chaque nouvelle expérience, on peut réutiliser.
- Pas d'outil spécifique : capitalisation de l'expérience

# Coder avec les principes LIM, KISS, DRY

Coder avec ces objectifs fondamentaux :

- **LIM – Less is more** : moins il y a de code, le mieux c'est.
  - Code minimaliste → meilleure factorisation → meilleure évolutivité
- **KISS – Keep it simple and stupid** : le plus simple est le mieux.
  - Code simple → facile à appréhender → facile à maintenir dans le temps
- **DRY – Dont Repeat Yourself**
  - Dupliquer le code est source de problèmes à l'avenir (modification partielle, oubli)

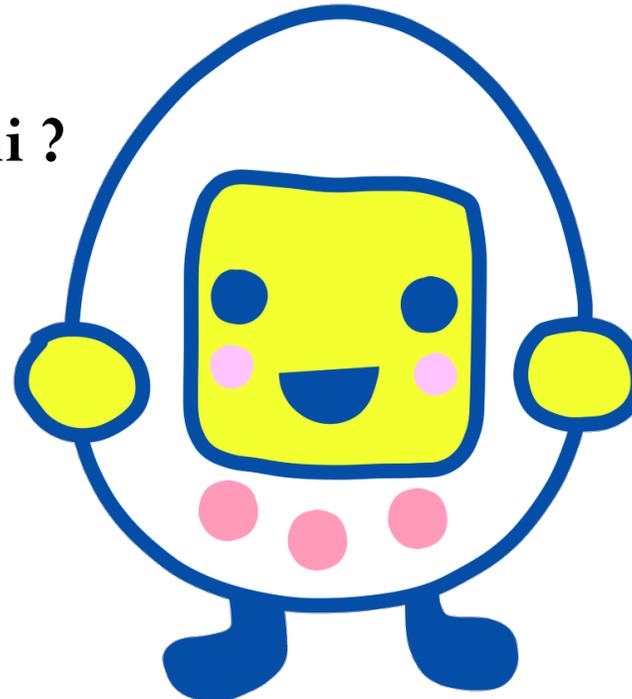
# Sommaire de ce cours

- Notions de base de la Conception :
  - Responsabilité ----- #5
  - Contrat, service ----- #7
  - Abstraction ----- #12
  - Cohésion, couplage ----- #21
  - Exercices Cohésion / couplage ----- #24
  - Exercice d'ancrage ----- #39
- Règles élémentaires de Conception ----- #47

# Notions de base d'une bonne conception :

## 1 - Responsabilité de classe

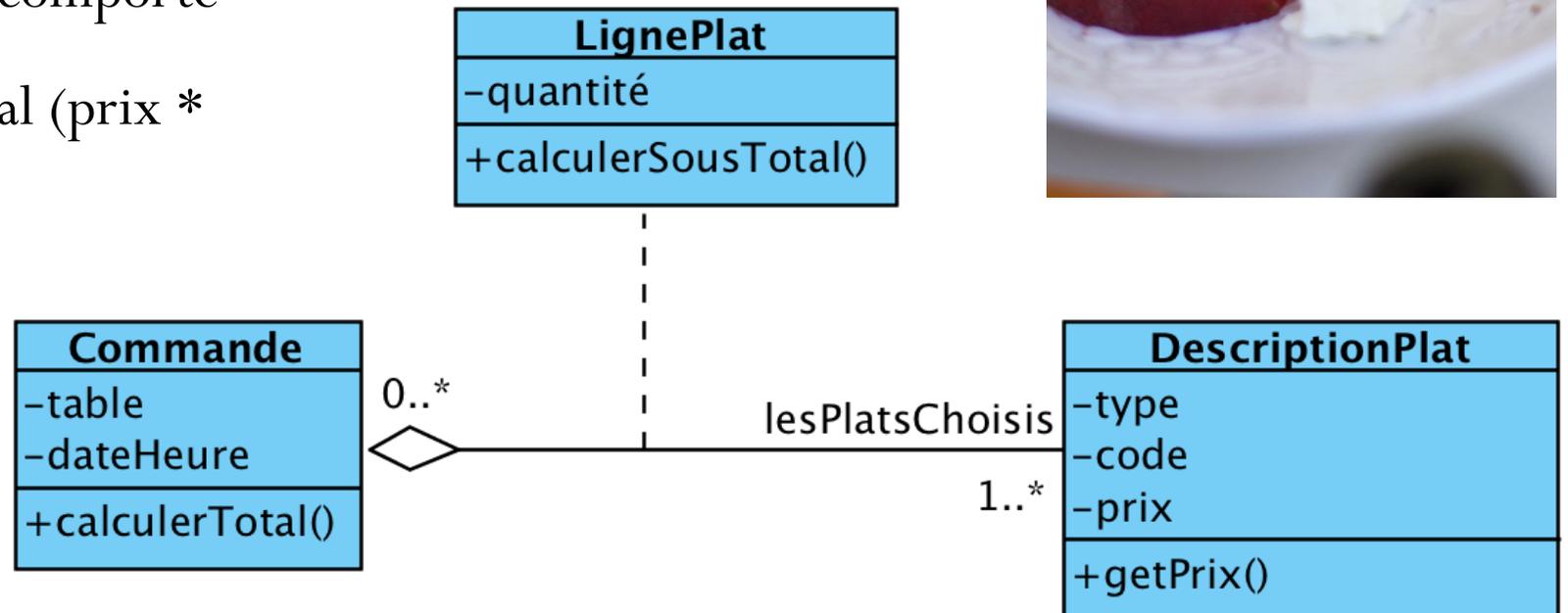
- DEF **Responsabilités** d'une classe :
  - Ce qu'elle **SAIT**
  - Ce qu'elle est capable de **FAIRE**
- Par ex., pour cette classe Tamagoshi ?



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

# Responsabilité de classe

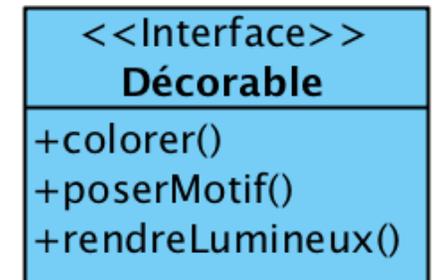
- Pour la **classe LignePlat** suivante ?
  - Elle sait : à quel objet **Commande** elle appartient, et à quel objet **DescriptionPlat** elle correspond
  - Elle sait combien de plats elle comporte
  - Elle peut calculer son sous-total (prix \* quantité de plats)



# 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

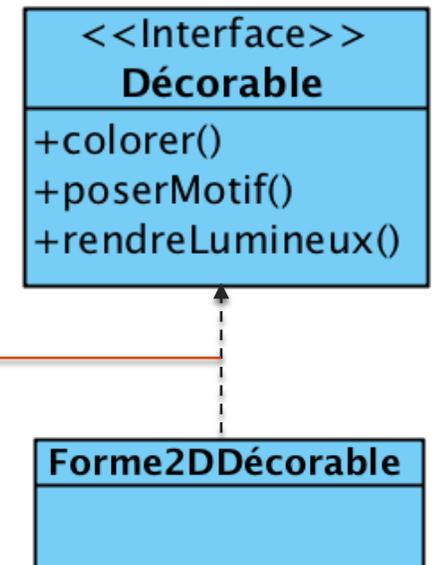
- **L'implémentation**

- 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 ?**

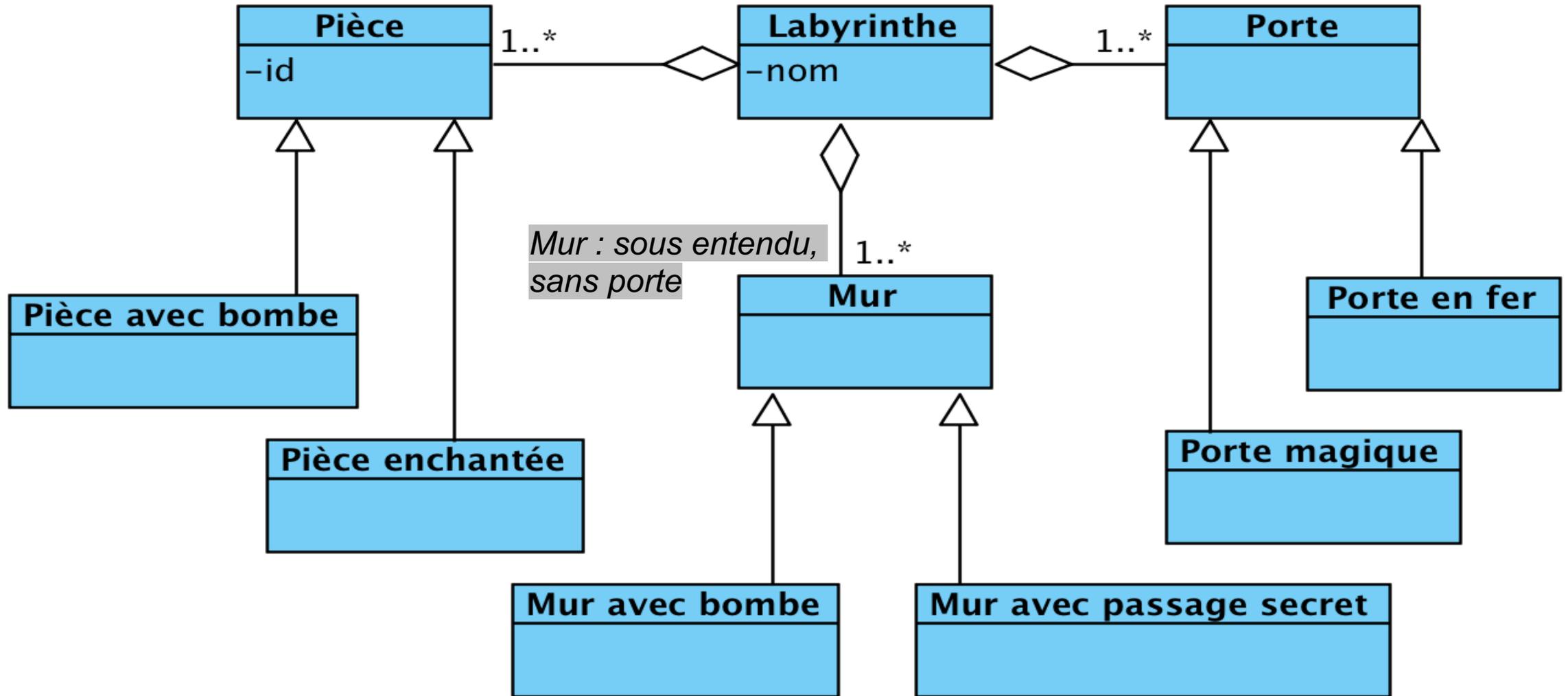
Ainsi si on veut gérer les collisions d'indices par la fonction de hachage à l'aide d'un **sondage linéaire\***, si on aura à stocker 50 clés, il suffit de prendre un tableau 2x plus grand et les performances resteront bonne.

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\*\***)

# 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

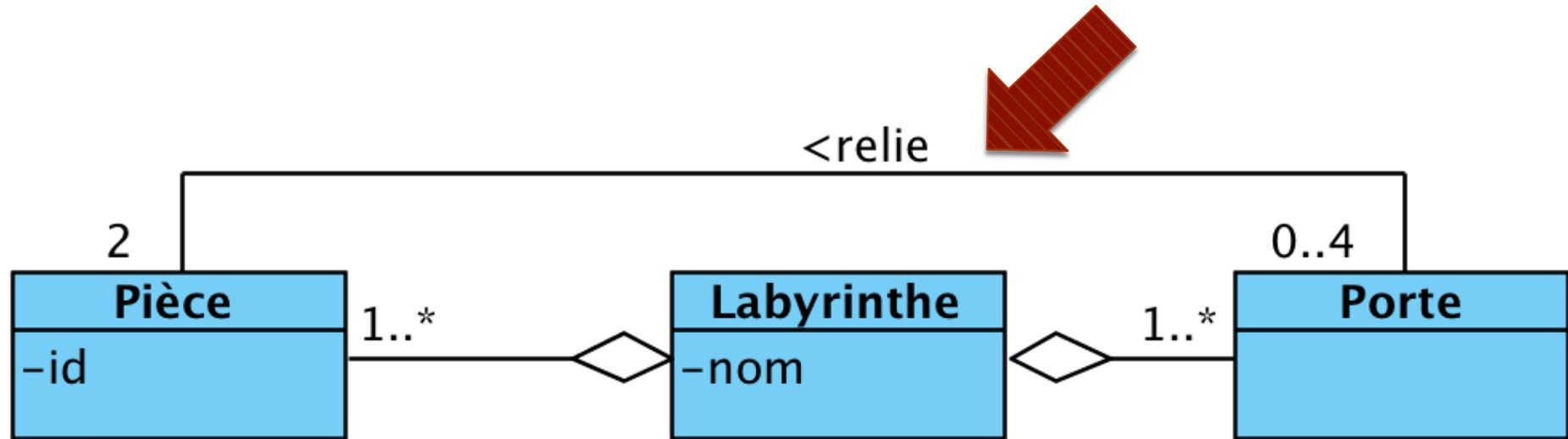




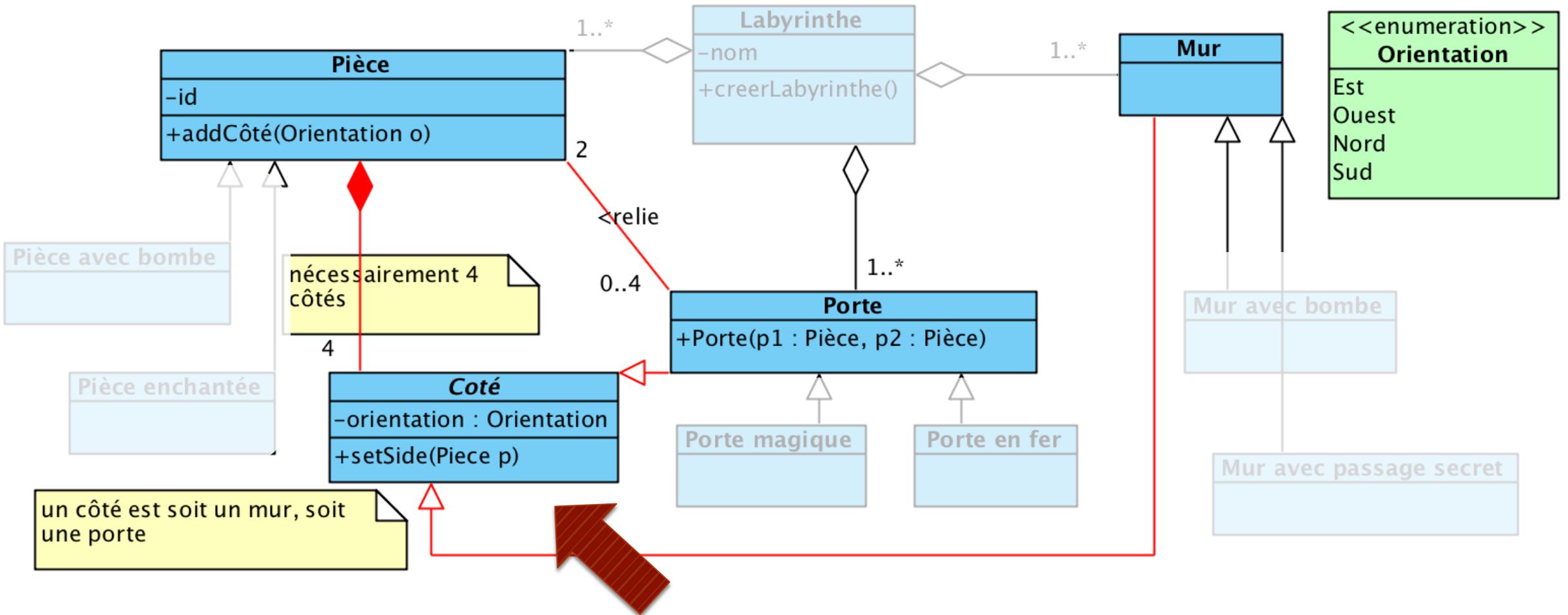
1- Que manque-t-il ici ?

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

# 1- Une porte relie nécessairement 2 pièces :



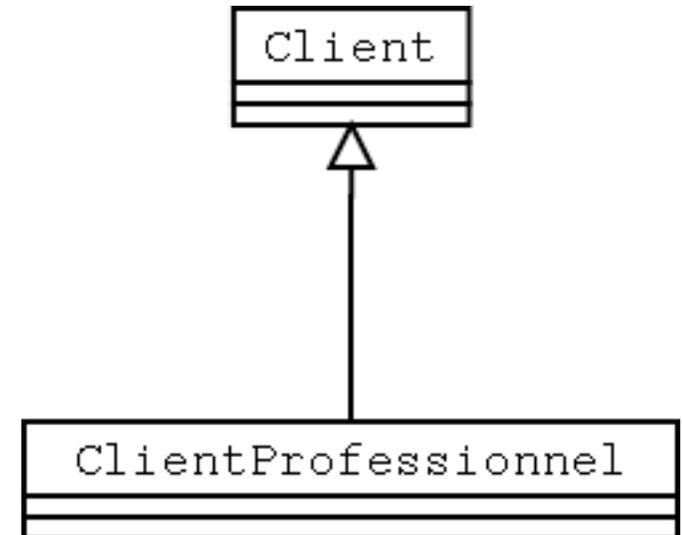
## 2- Une pièce possède 4 côtés, chaque côté pouvant être un mur ou une porte



# Rappel : Interface vs. classe abstraite

Similaires (méthodes abstraites) mais différentes (objectifs)

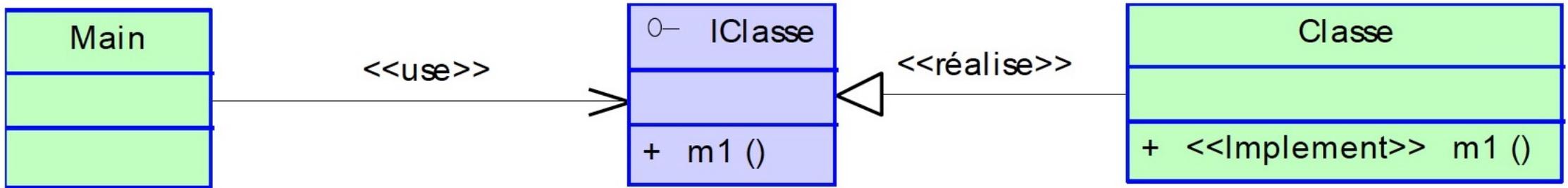
- **Avantages d'une interface** : spécifie un contrat, abstraction quasi\* complète, héritage multiple d'interfaces, nouvelle implémentation possible à tout moment
  - Analogie : prise de courant, son contrat = fournir du courant alternatif; comment ? pour faire quoi ? Peu importe.
- **Avantages d'une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



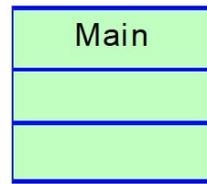
# Corrigé exercice Interface

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

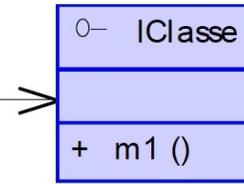
*Signature uniquement*

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

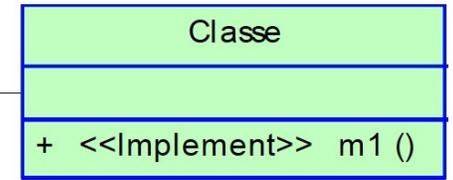
*Implémentation de m1()  
pour Classe*



<<use>>



<<réalise>>



```
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,
- ▶ Éléments 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 ?

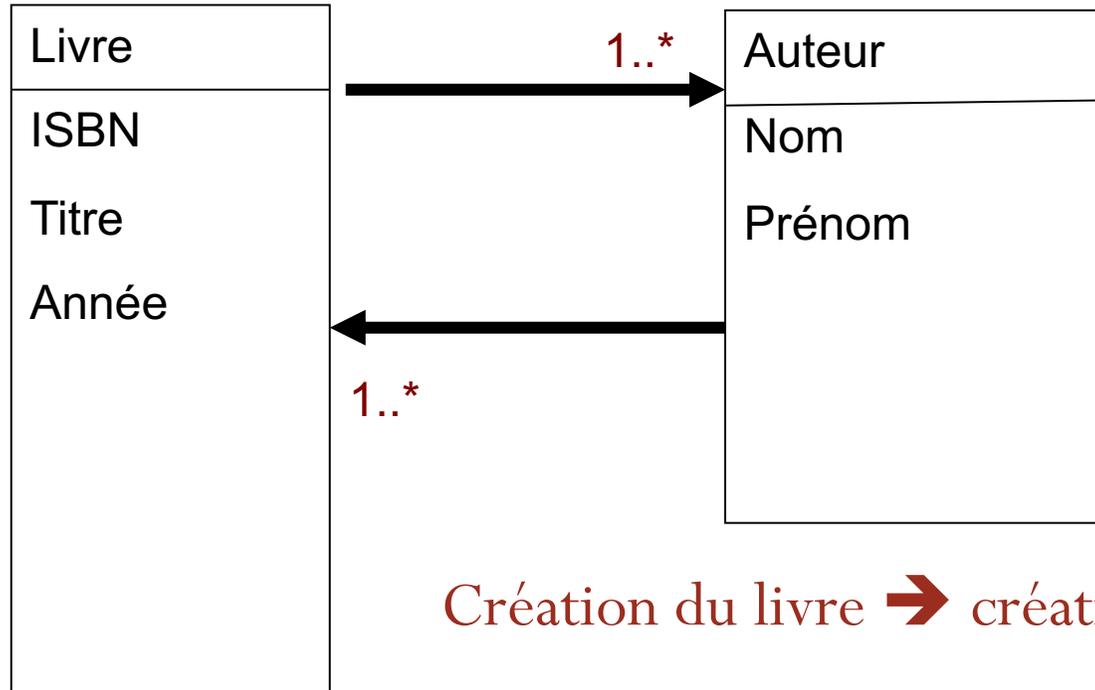
**Module** en COO = une classe, un package, une méthode, un composant physique

- 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



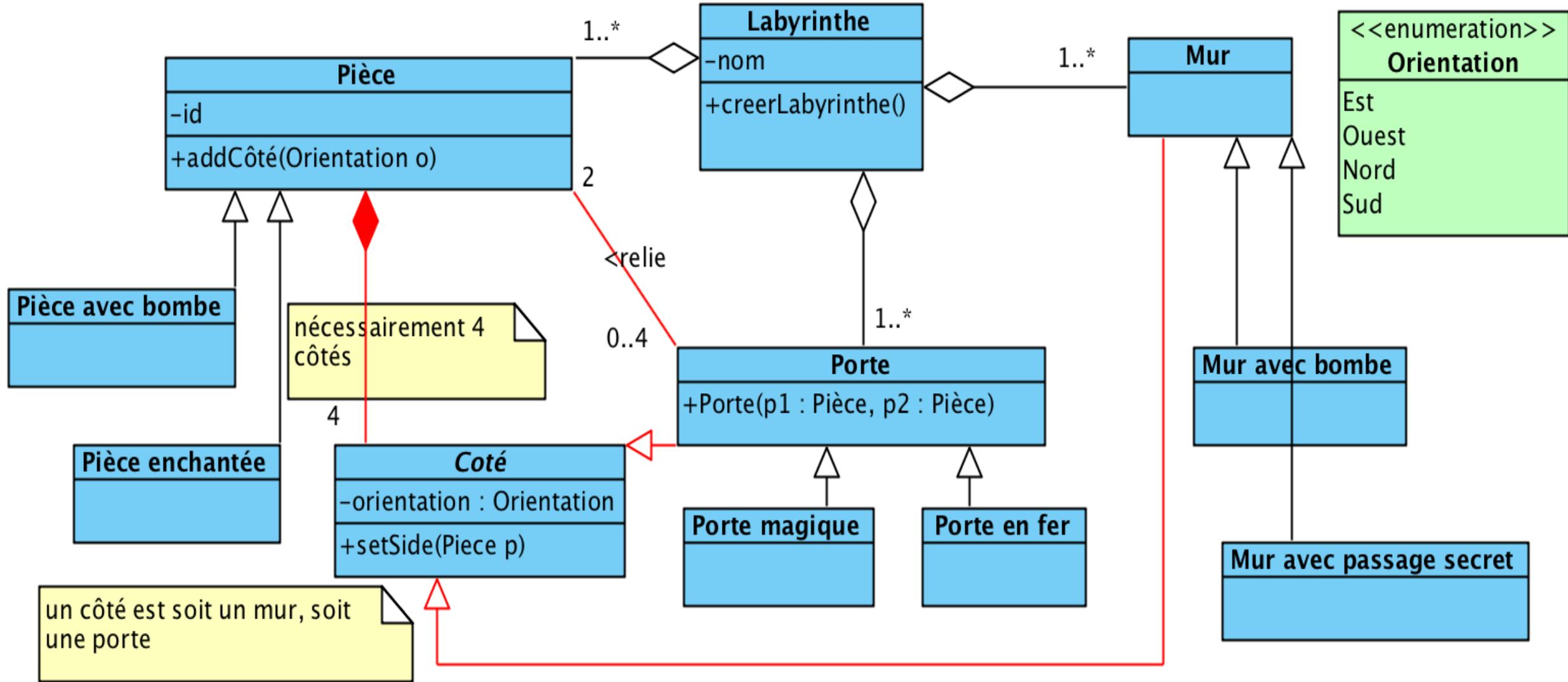
Un fort couplage n'est pas dramatique avec des composants **stables** (ex. `java.util`)

# Que penser de la cohésion de ces composants ?

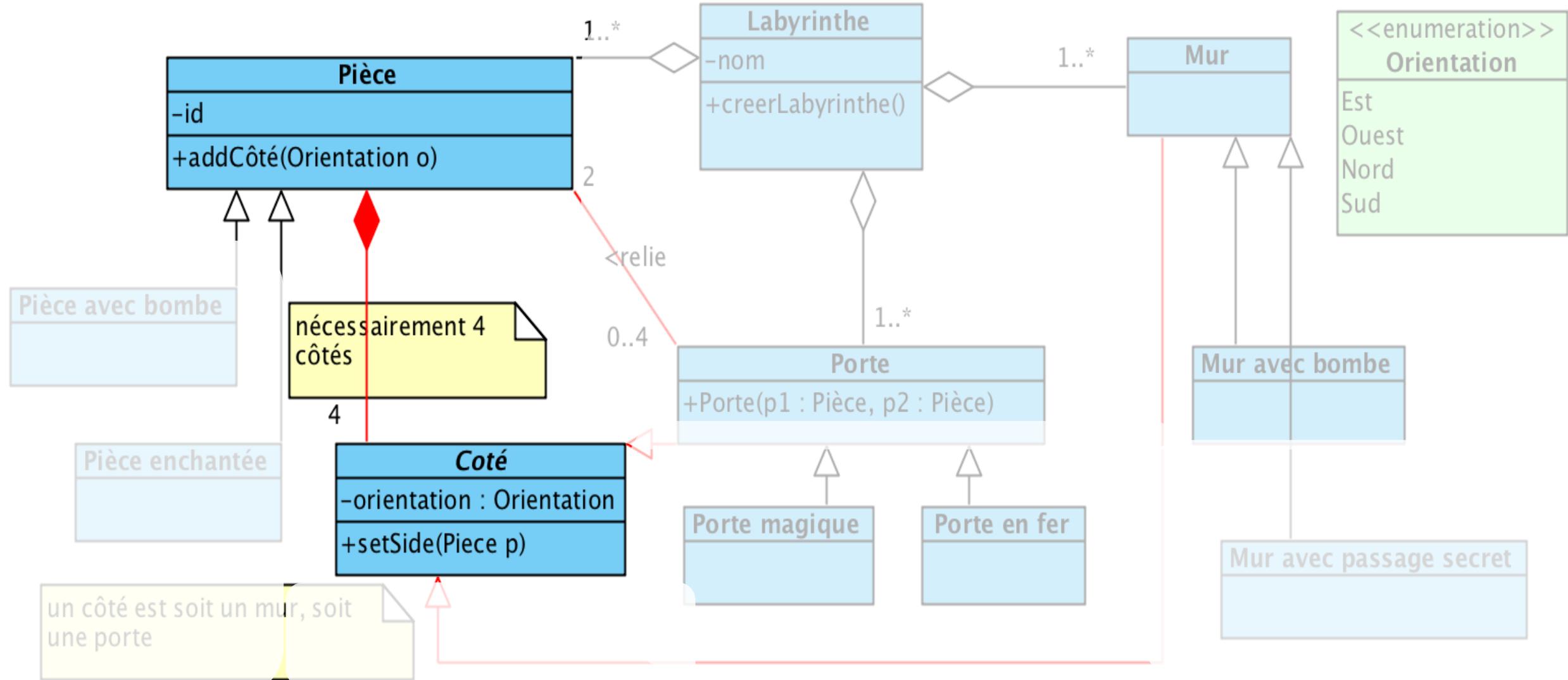
1. Un **composant A** avec une fonction qui lit les entrées d'un système à bande magnétique, une autre qui lit un disque, une dernière fonction qui lit les données du réseau
2. Un **composant B** avec un élément qui lit des entrées et génère des données qui viennent alimenter un autre élément du composant,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



# Labyrinthe : quels couplages voyez-vous ?



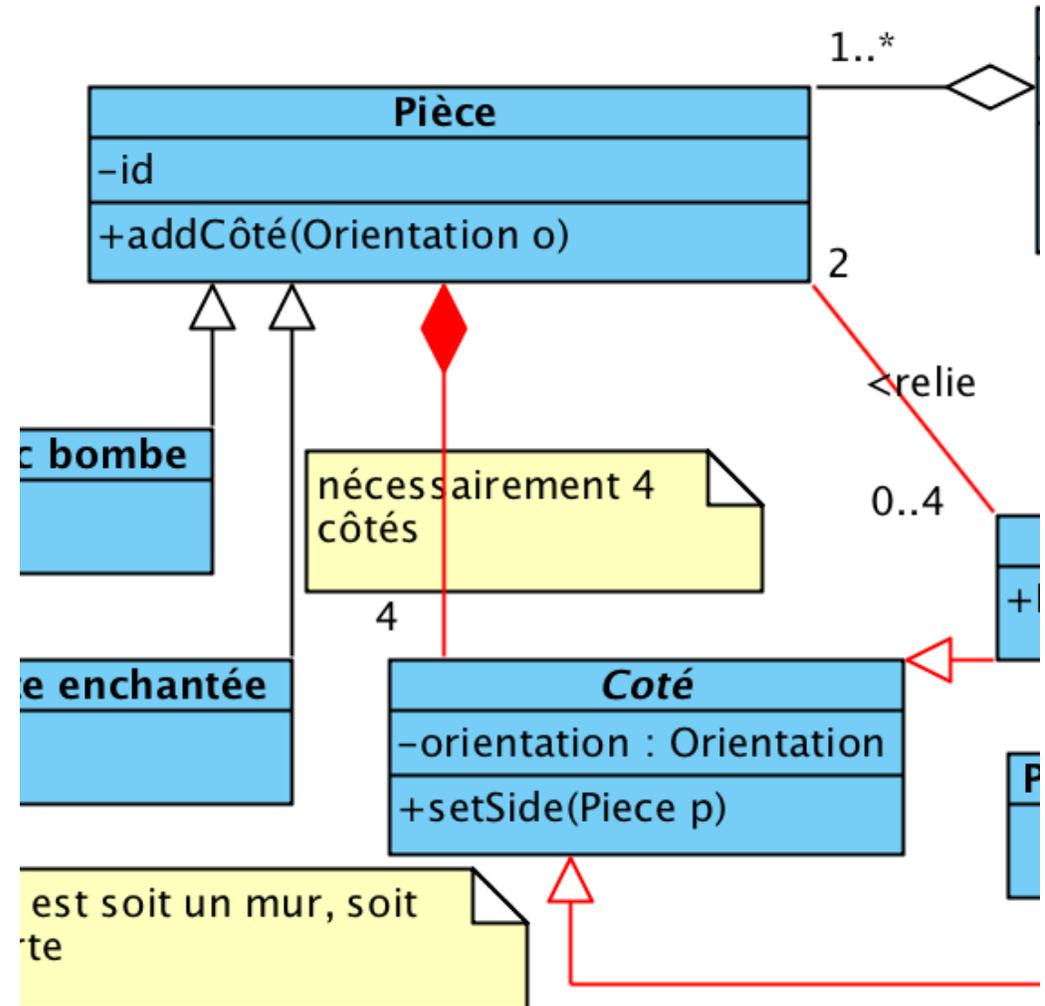
# Labyrinthe : quels couplages voyez-vous ?



# 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)

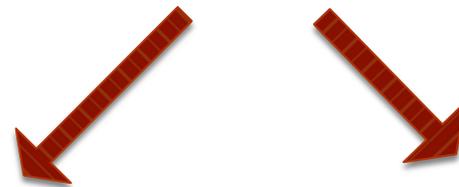
- **Les avantages d'un faible couplage :**
  - les modifications apportées à une partie vont moins se propager dans le reste du système
  - les composants sont faciles à remplacer et à réutiliser.
- Les classes abstraites et l'héritage permettent de minimiser les interdépendances en **contrôlant le niveau 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

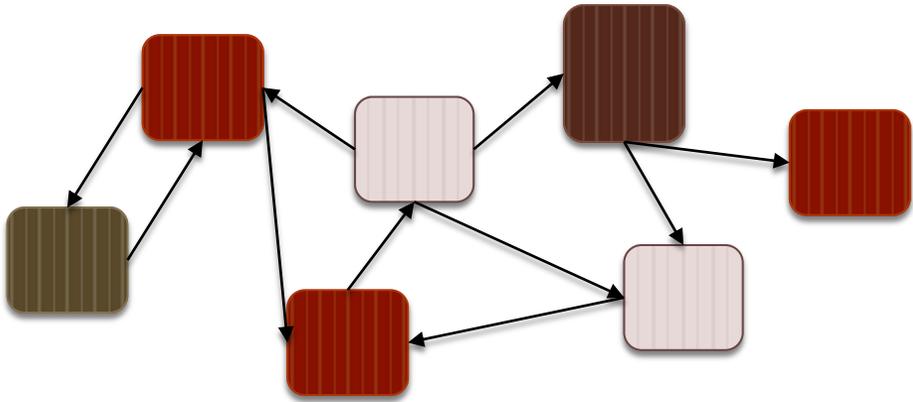
- Un couplage trop élevé ? Une **assiette de spaghettis**
  - Maintenance difficile, voire impossible
  - Lisibilité faible
- C'est parfois volontaire : principe d'**obfuscation**
  - Code rendu illisible pour protéger ses sources de rétro-ingénierie



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

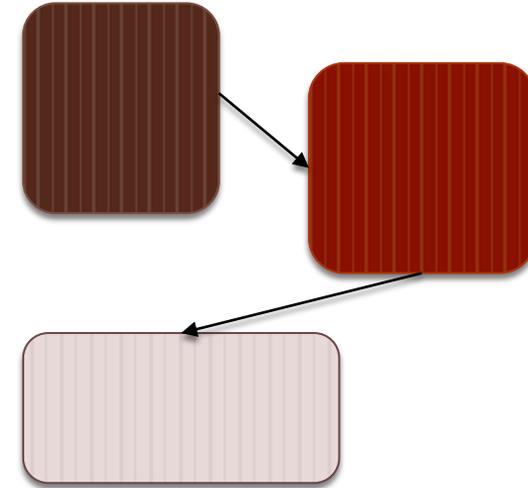
Sinon le développeur va passer du temps à tester tous les modules pour déterminer si chacun d'entre eux est affecté ou non par un changement.

## 2 notions antinomiques ?

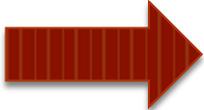


- Chaque composant fait une seule chose
- Faible niveau d'abstraction
- Relations complexes
- Ultra réutilisable

 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**

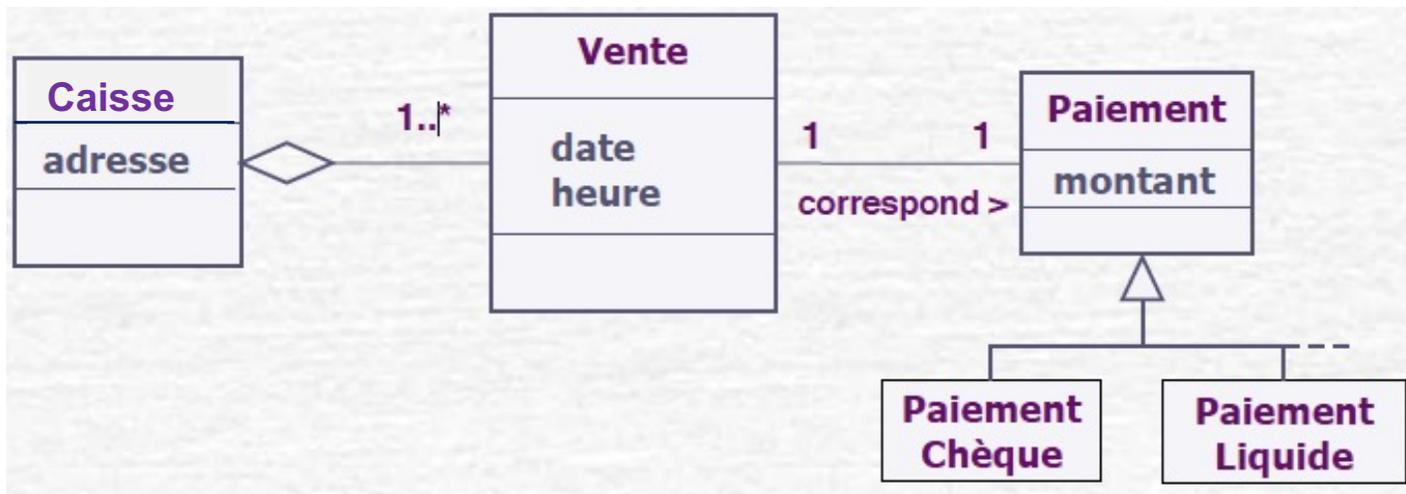


*Un module hautement cohésif exécute une tâche ou atteint un objectif unique  
« faire une chose et la faire bien »*

# 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

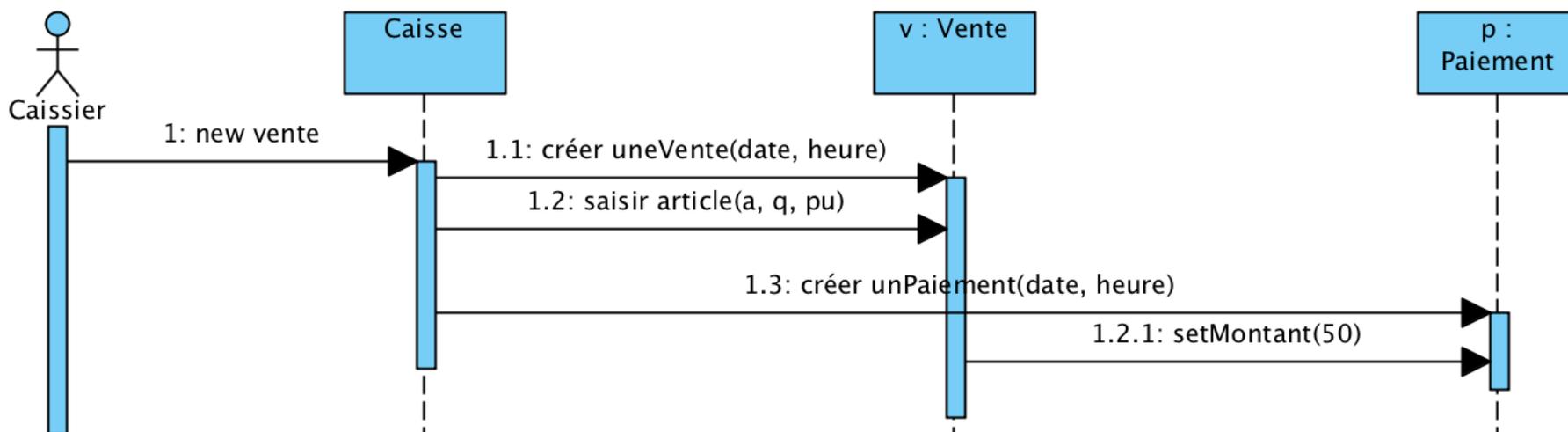
# Illustration : ne mettre que les dépendances nécessaires



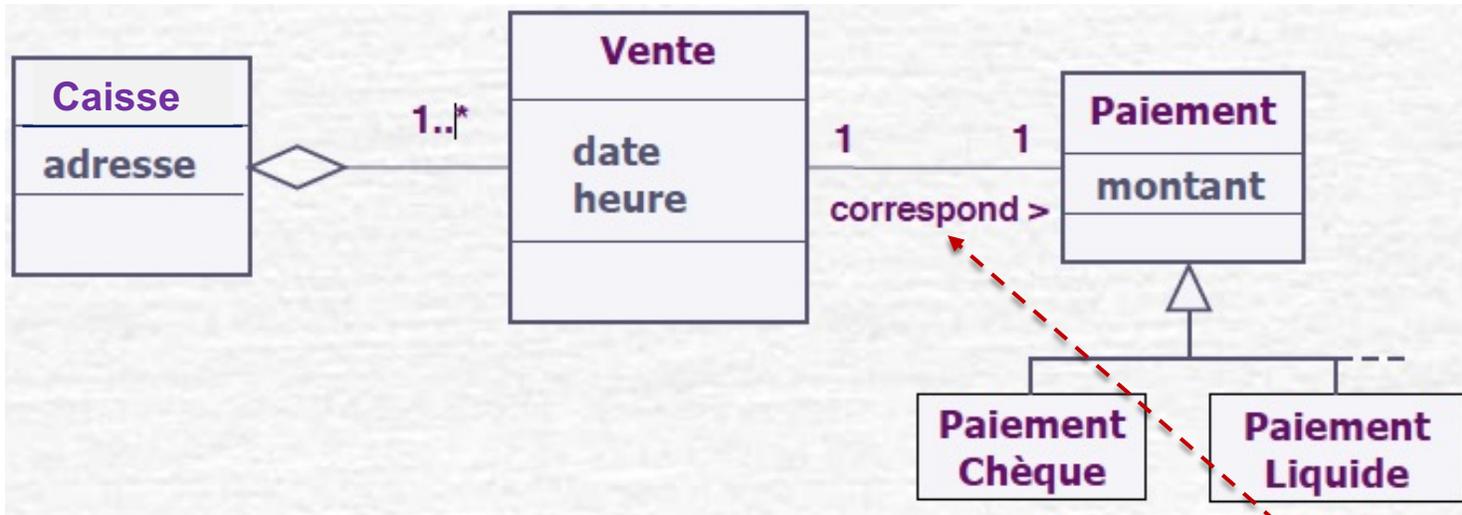
*Si c'est la Caisse qui crée le paiement, on **ajoute** un couplage de Caisse à Paiement, qui n'existait pas dans le DCL*

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

**solution 1 :**

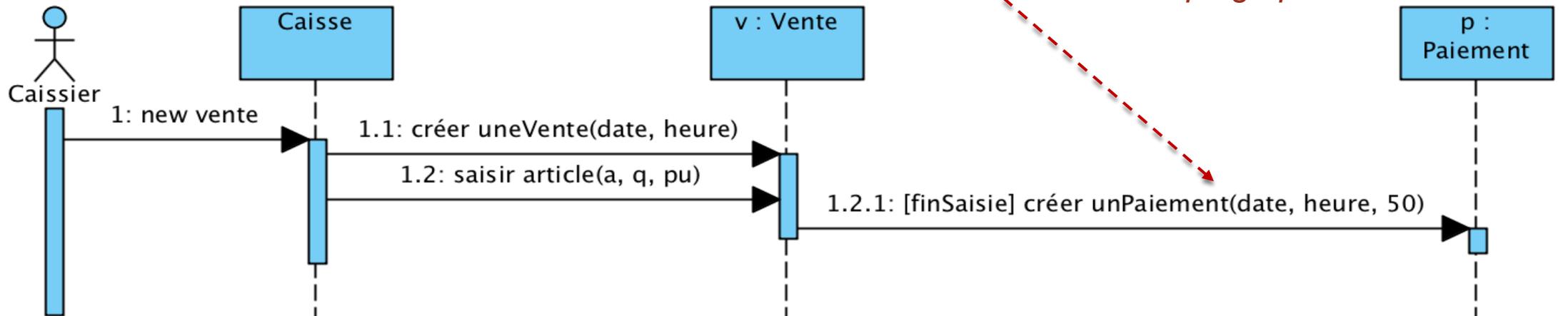


# Faible couplage : règles (2)



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

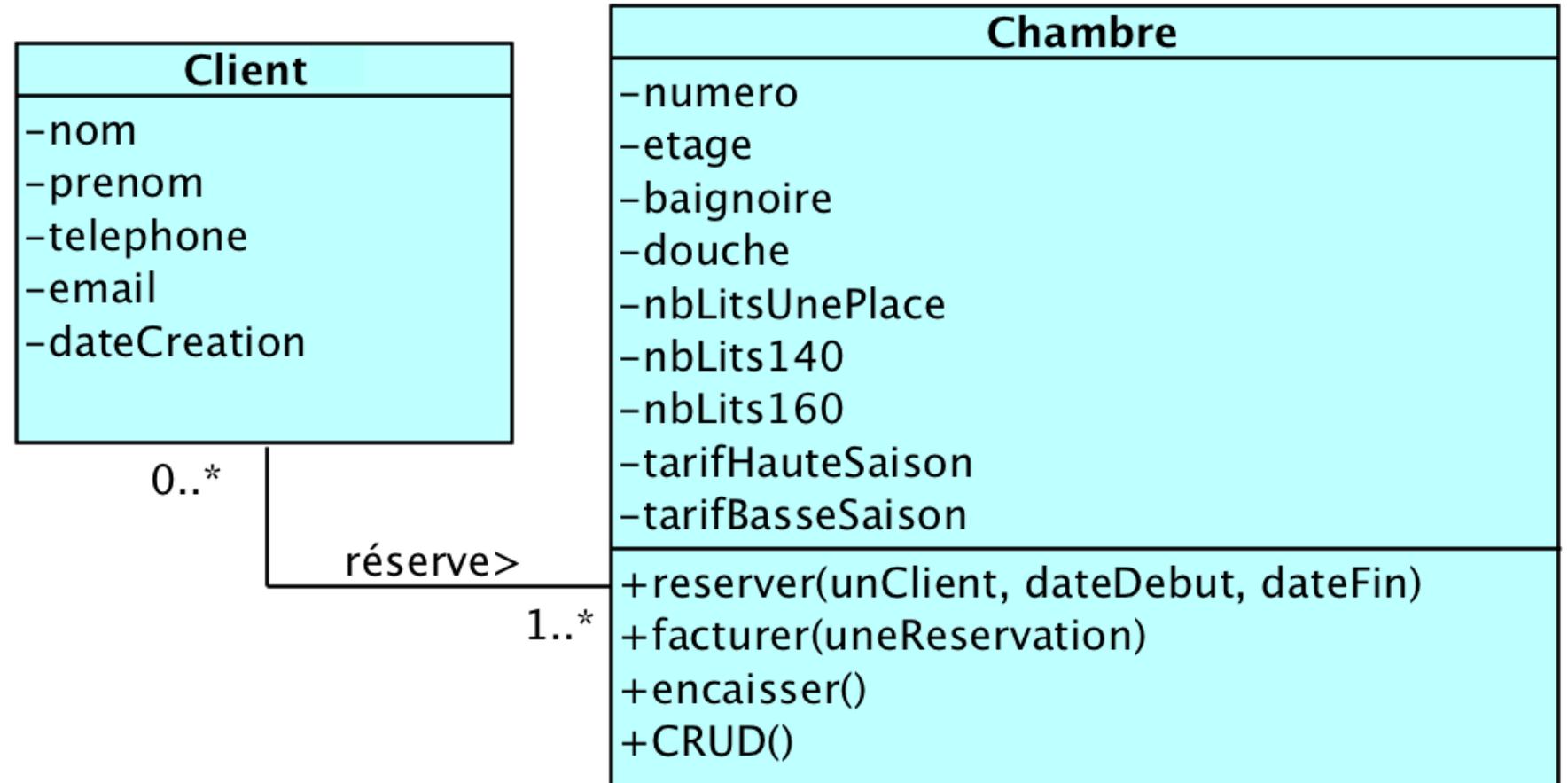
- Des *xxxxxxxxxxx xxxxxx* :
  - Favorisent la lisibilité et la compréhension du code
  - Rendent le code plus facile à maintenir et à tester
  - Permettent le contrôle des impacts lors de changements dans le code
- Une classe a une forte *yyyy* si tous ses attributs sont employés par ses méthodes.
- 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**

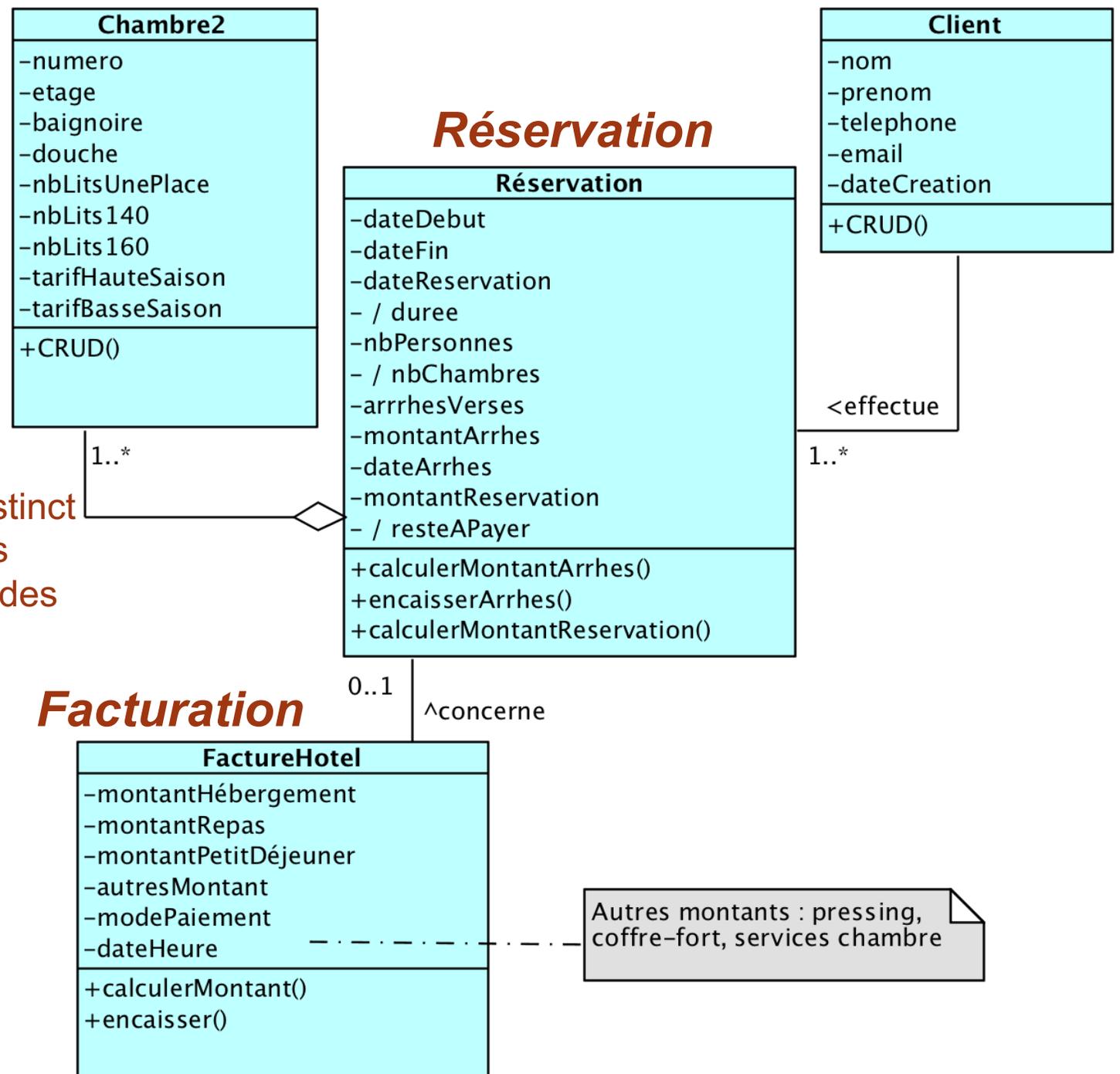
*Différents domaines sont couverts  
dans la classe Chambre !*

- **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.

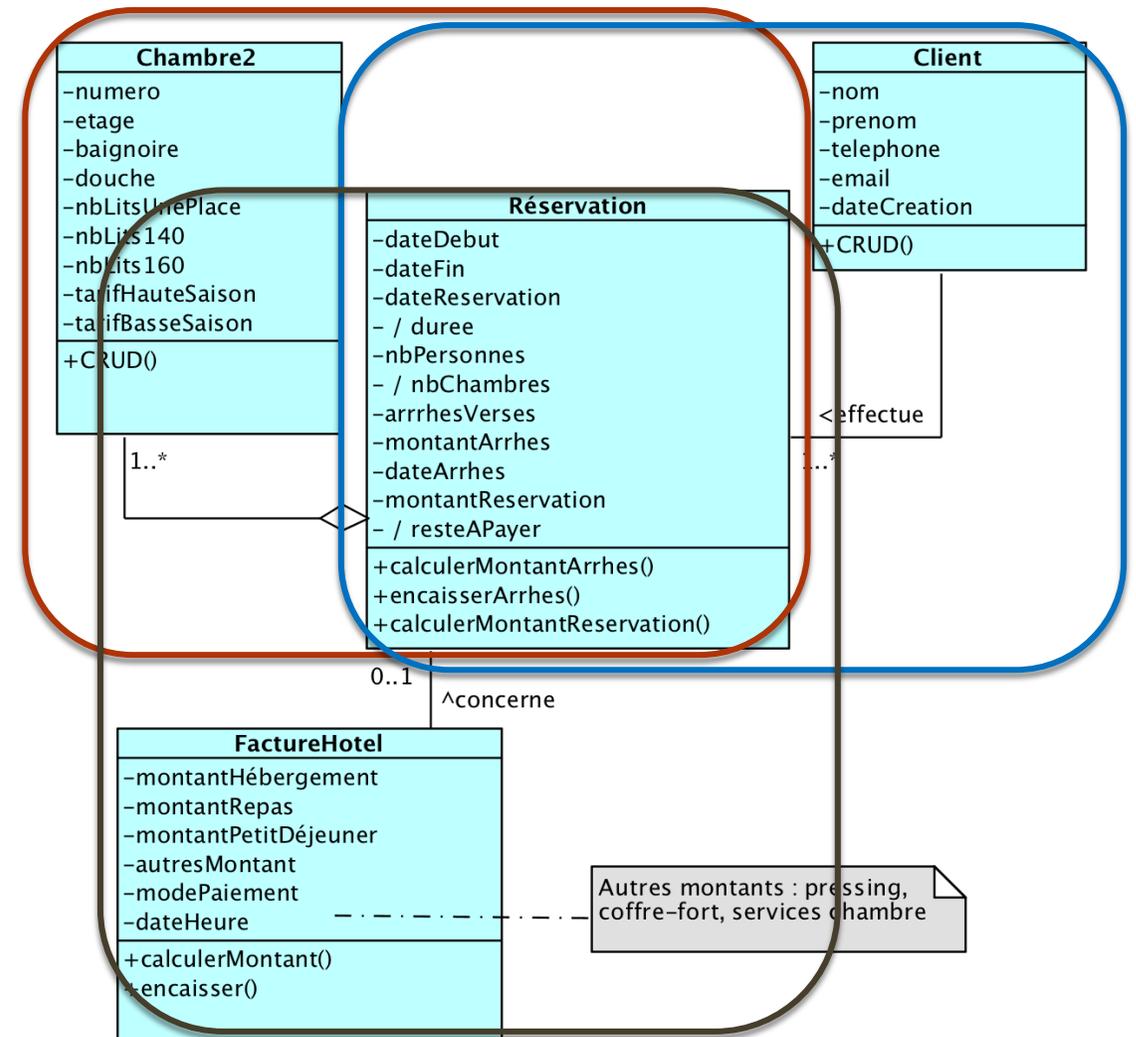
→ **Meilleure cohésion des classes**



# 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 *calculerMontantReservation()* 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.

# Fiche TD#1

---

Souffler un peu....