Plan

- **Rappel bases OO**
- Premiers Principes OO
- Grands principes OO
- Divers
  - Approches de développement
  - Pratiques de programmation
- Design Patterns OO
- Restes

Bases de l'OO

- Abstraction
- Encapsulation
- Polymorphisme
- Héritage
- …

=> revoir cours module M2103 !!

Plan

- Rappel bases OO
- **Premiers Principes OO**
- Grands principes OO
- Divers
  - Approches de développement
  - Pratiques de programmation
- Design Patterns OO
- Restes

Un bon logiciel

- Un bon logiciel doit satisfaire le client
  - Le logiciel **doit faire ce que le client veut qu'il fasse**
- Un bon logiciel est **bien conçu, bien codé** et **facile à maintenir, réutiliser et étendre**

Un bon logiciel en 3 étapes

1. Assurez-vous que votre logiciel fait ce que le client veut qu'il fasse
2. Appliquez les **principes OO** de base pour ajouter de la souplesse
3. Œuvrez pour une conception facile à maintenir et à réutiliser

L'ACOO sert à écrire du bon logiciel

- Les clients sont satisfaits quand leurs applications FONCTIONNENT
- Les clients sont satisfaits quand leurs applications CONTINUENT A FONCTIONNER
- Les clients sont satisfaits quand leurs applications peuvent être AMELIOREES
- Les programmeurs sont satisfaits quand leurs applications peuvent être REUTILISEES
- Les programmeurs sont satisfaits quand leurs applications sont SOUPLES

Principes OO : les prémisses

- Un principe de conception est un outil ou une technique de base qui peut être appliqué à la conception ou l'écriture du code pour le rendre plus souple, plus facile à maintenir ou à étendre

Principe OO : encapsulation

- Elle permet de subdiviser votre application en parties cohérentes
- En séparant les différentes parties, on peut alors changer l'une sans modifier toutes les autres
- En général on encapsule les parties de l'application qui peuvent varier indépendamment des parties qui vont rester stables
  - L'information d'une partie de l'application est ainsi protégée des autres parties

Principe OO : encapsulation (2)

- L'encapsulation des données des classes en les rendant privées est le plus connu

- On peut aussi encapsuler
  - un **ensemble complet de propriétés**
  - ou même des **comportements**
  - Ceci est modélisé par une **agrégation/composition**

- Chaque fois que vous voyez du code dupliqué, cherchez où l'encapsuler

Principe de délégation

- Action d'un objet renvoyant une opération/tâche vers un autre objet pour qu'elle soit réalisée pour le compte du premier objet

- Elle permet au code d'être plus facilement réutilisable
  - Vos objets s'occupent ainsi de leurs propres fonctionnalités plutôt que de disséminer le code gérant leur comportement un peu partout dans le code de l'application

- La méthode « equals() » est l'exemple le plus courant de délégation

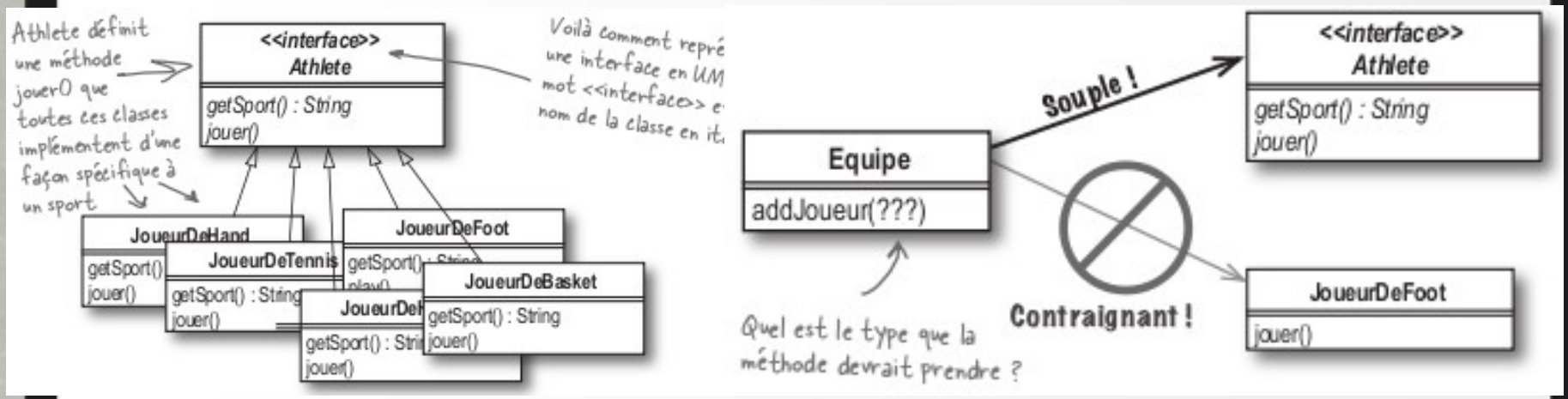
- Avec la délégation les objets sont plus indépendants les uns des autres : ils sont lâchement couplés, chaque objet a une responsabilité unique

Astuces

- L'une des meilleures façons de vérifier qu'un logiciel est bien conçu est d'essayer de le MODIFIER
- Si votre logiciel est difficile à modifier, il y a certainement quelque chose à améliorer dans la conception...

Principe : utiliser les interfaces comme des types

- Coder avec une interface à la place d'une implémentation rendra votre logiciel plus facile à étendre



- En codant avec une interface votre code fonctionnera avec **toutes** les sous-classes de l'interface, même celles qui ne sont **pas encore créées**

Astuce : mort d'une décision de conception

- Codez une fois, vérifiez deux fois
- Continuez à inspecter vos architectures quand vous rencontrez des problème
- Une conception se construit par cycles... et vous devez être capable de changer vos propres conceptions, de la même façon que celles que vous héritez d'autres programmeurs

Astuce : mort d'une décision de conception (2)

- La fierté tue une bonne conception : n’ayez pas peur d’examiner vos propres décisions de conception et de les améliorer, même si cela implique de revenir en arrière

- La plupart des bonnes conceptions viennent de l'analyse des mauvaises conceptions ! N'ayez donc pas peur de faire des erreurs puis de tout restructurer

Principe : cohésion de classe

- La cohésion mesure le degré de connectivité entre les éléments d'un module, d'une classe ou d'un objet donné
- Plus la cohésion de votre logiciel est forte, plus les responsabilités de chaque classe individuelle sont bien définies et reliées dans votre application
- Chaque classe a un ensemble très spécifique d'actions de proximité qu'elle accomplit.
- Une classe **cohésive** fait une chose vraiment bien et n'essaie pas de faire ou d'être quelque chose d'autre
- L'objectif d'une bonne conception est un logiciel **fortement cohésif** et **lâchement couplé**

Principe de non duplication du code

- Évitez le code dupliqué en rendant les éléments communs abstraits et en les plaçant tous au même endroit
- **DRY** = *Don't Repeat Yourself*
- S'applique également aux exigences
  - **UNE** exigence à **UN** endroit
- L'intérêt du principe de non duplication est d'attribuer une **place unique** et **appropriée** pour chaque information ou comportement de votre système

Principe de responsabilité unique

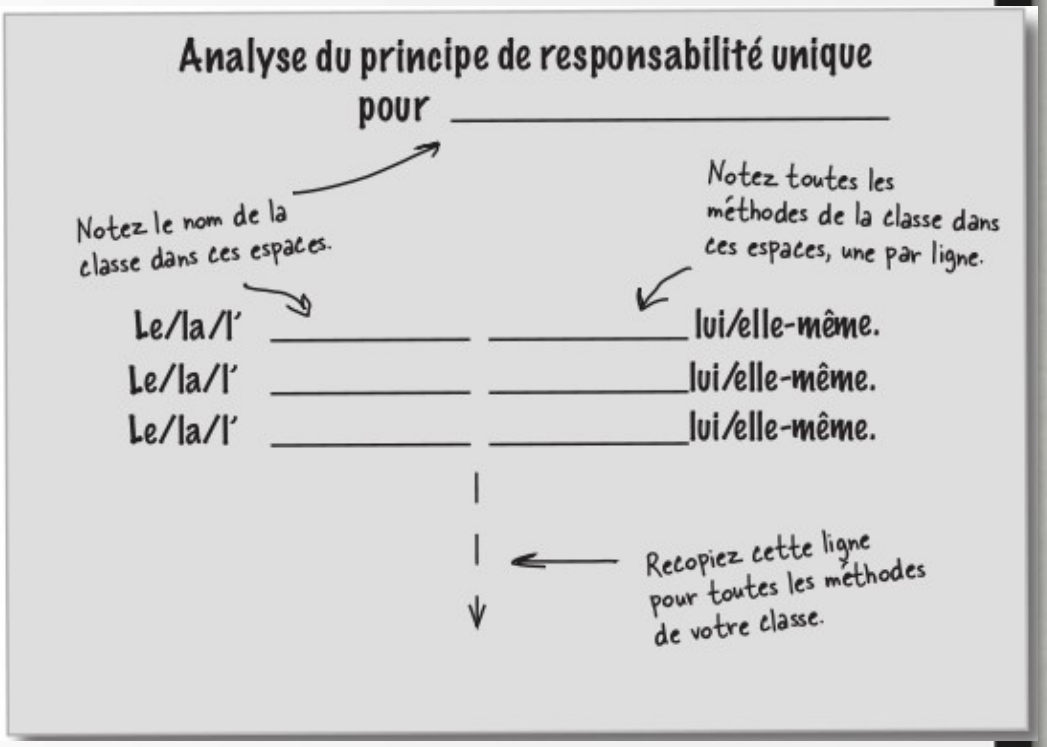
- Chaque objet de votre système ne doit avoir qu'une seule responsabilité et tous les services de cet objet doivent s'efforcer d'assumer cette unique responsabilité

- SRP = *Single Responsability Principle*

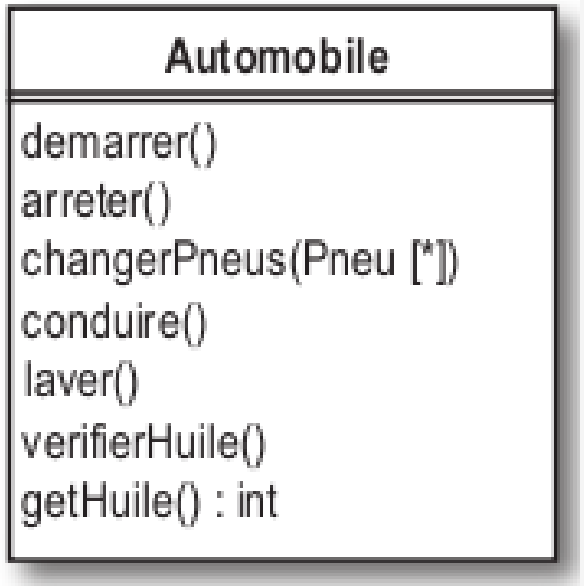
- Vous avez implémenté le principe de responsabilité unique correctement si chacun de vos objets n'a qu'**une seule raison de changer**

Principe de responsabilité unique (2)

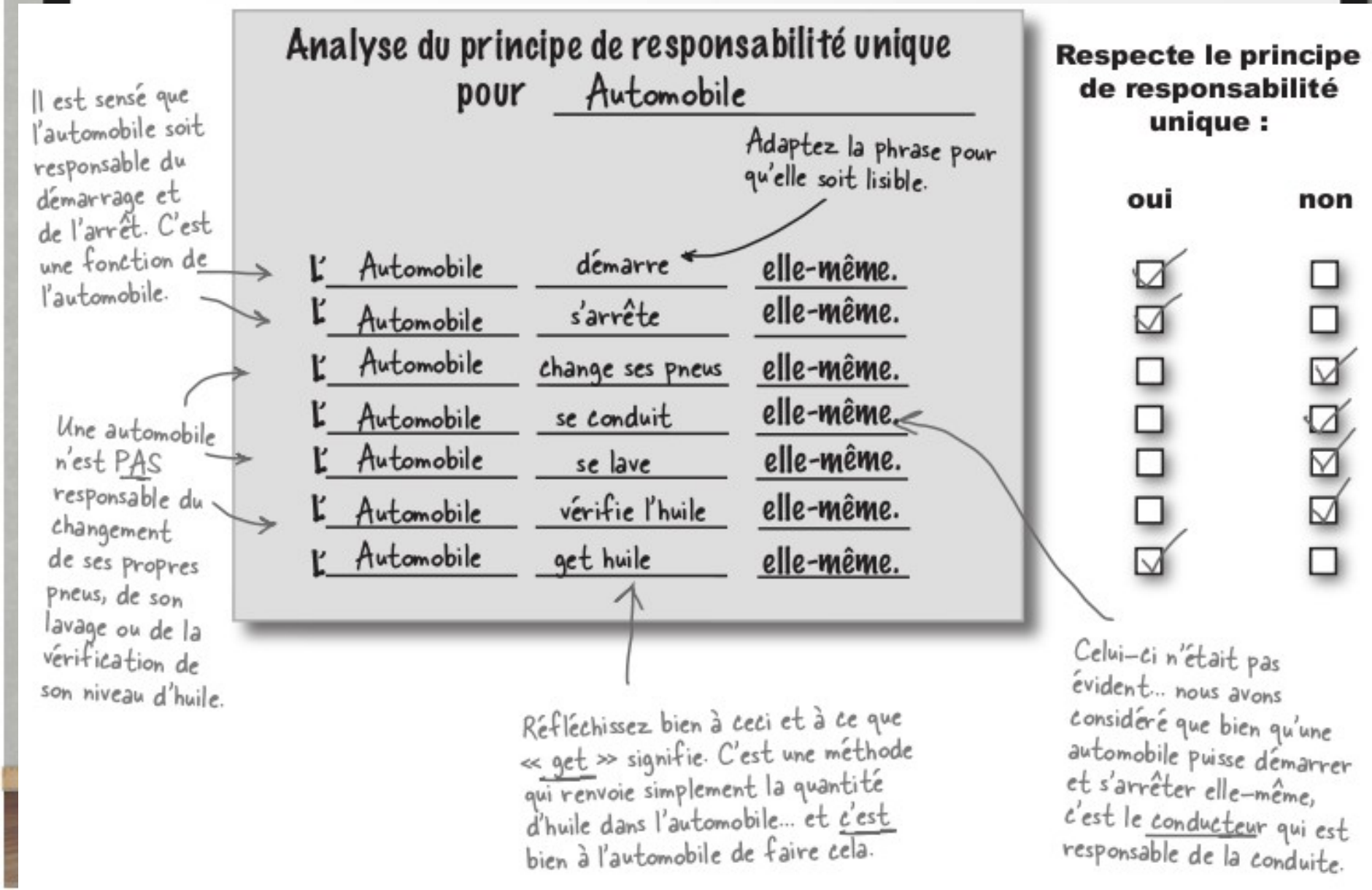
- **Aide pour repérer les responsabilités multiples**
  - Si ce que vous avez dit n'a pas de sens, vous violez probablement le principe de responsabilité unique avec cette méthode
  - La méthode appartient peut-être à une autre classe… voyez si vous pouvez la déplacer



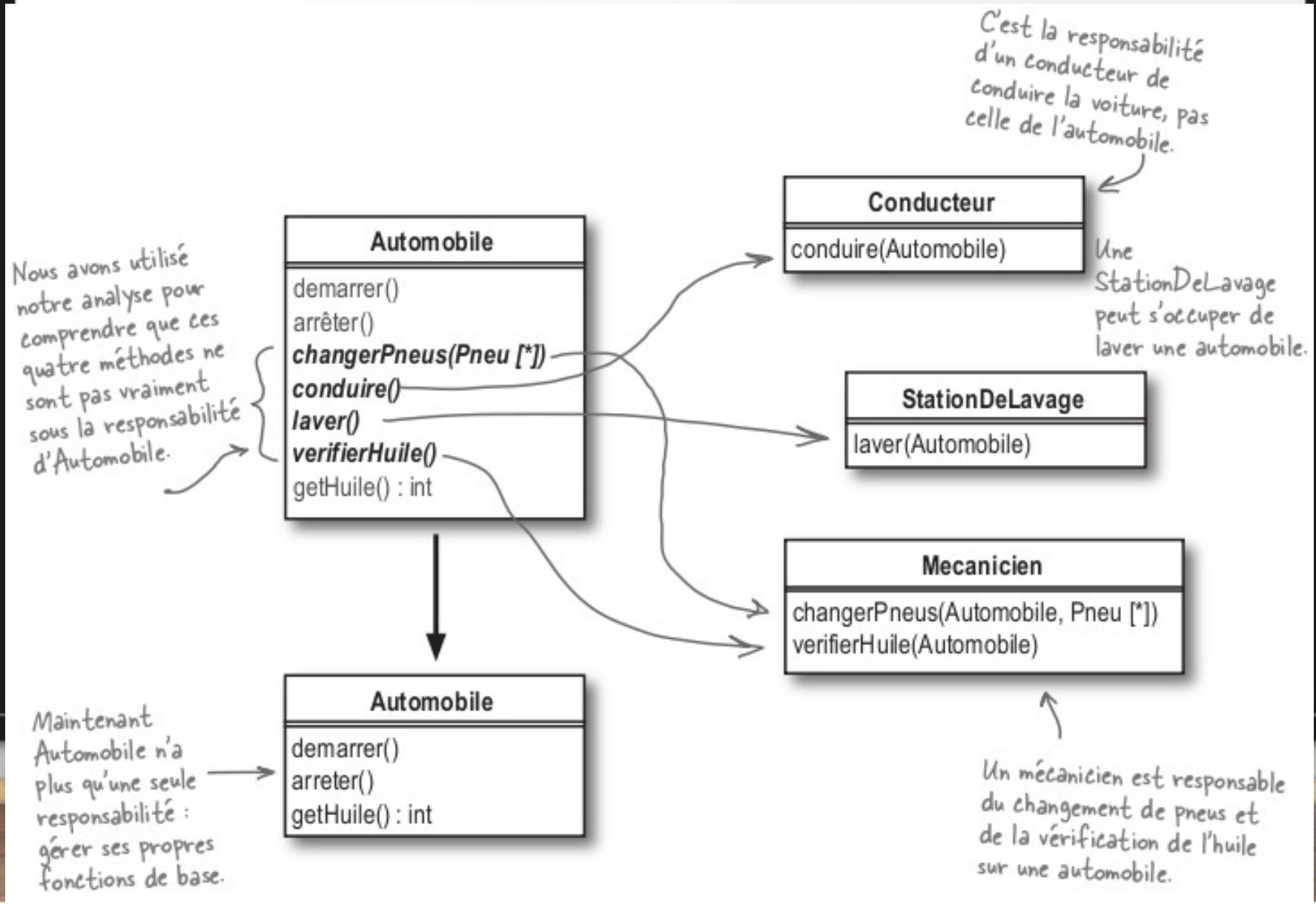
Exemple


Automobile
demarrer()
arreter()
changerPneus(Pneu [*])
conduire()
laver()
verifierHuile()
getHuile() : int

Exemple (suite)



Exemple (suite) : des responsabilités multiples à une responsabilité unique



Autres principe de conception : ne parlez pas aux inconnus – ne parlez qu'à vos amis immédiats

- Empêcher de créer des systèmes constitués d'un grand nombre de classes fortement couplées dans lesquels les modifications d'une composante se propagent en cascade aux autres parties
- À combien de classes ce code est-il couplé?

```
public float getTemp() {
    return station.getThermometre().getTemperature();
}
```

- Lignes directrices : à partir de n'importe quelle méthode d'un objet quelconque, vous ne devez appeler que des méthodes appartenant
  - À l'objet lui-même
  - Aux objets transmis en arguments à la méthode
  - Aux objets que la méthode crée ou instancie
  - Aux composants de l'objet

Autres principe de conception : ne parlez pas aux inconnus – ne parlez qu'à vos amis immédiats

- Exemple
  - Sans le principe

```
public float getTemp() {
  Thermometre thermometre = station.getThermometre();
  return thermometre.getTemperature();
}
```

  - Avec

```
public float getTemp() {
  return station.getTemperature();
}
```

- Contre-exemple courant en Java
  - System.out.printl(...)

Plan

- Rappel bases OO
- Premiers Principes OO
- **Grands principes OO**
- Divers
  - Approches de développement
  - Pratiques de programmation
- Design Patterns OO
- Restes

Principes OO : les grands principes

- Gestion des dépendances entre classes
  - **Principe d'ouverture/fermeture**
  - **Principe de substitution de Liskov**
  - **Principe d'inversion des dépendances**
  - **Principe de séparation des interfaces**
- Organisation de l'application en modules
  - Principe d'équivalence livraison/réutilisation
  - Principe de réutilisation commune
  - Principe de fermeture commune

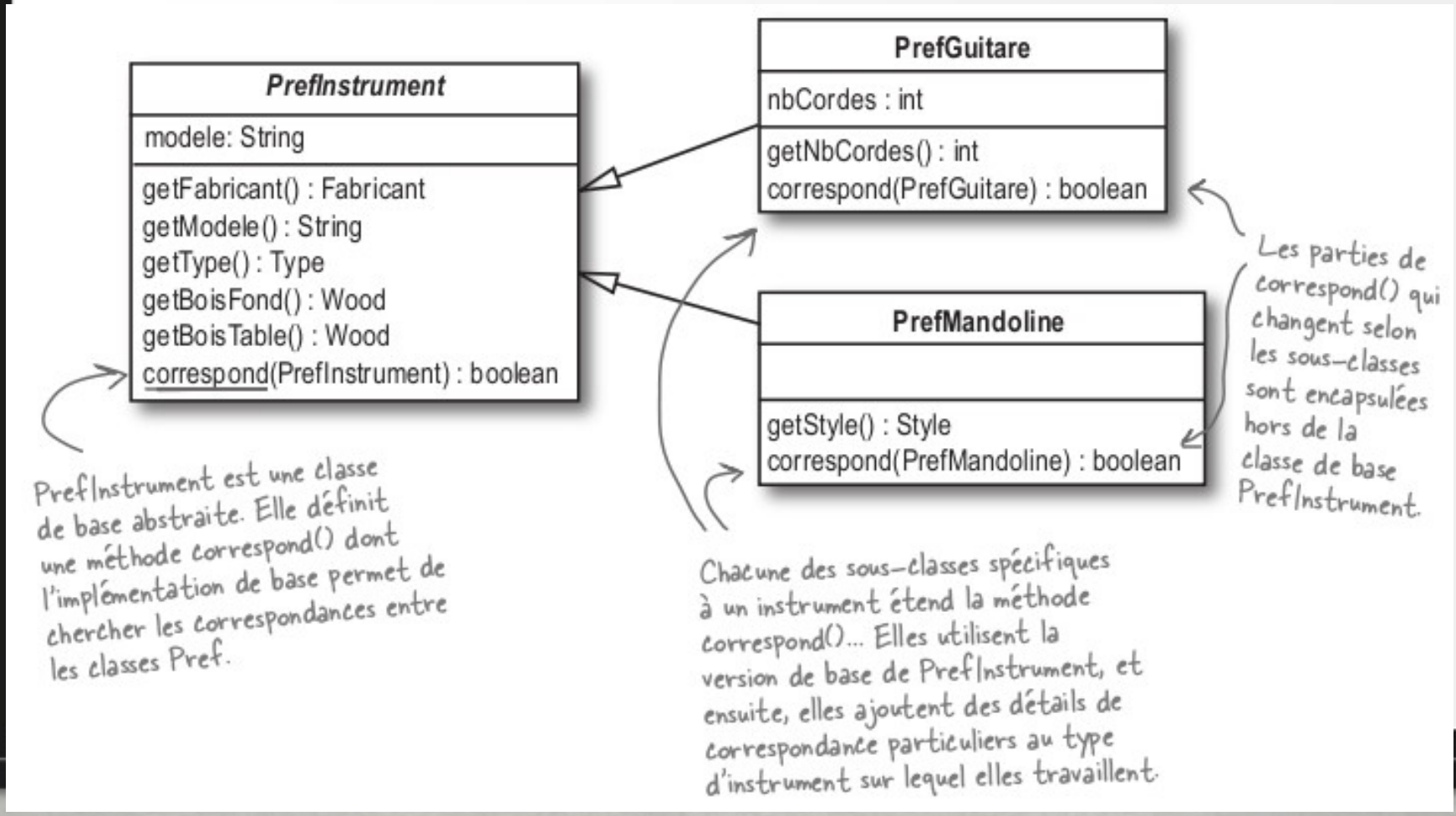
Principes OO : les grands principes (2)

- Gestion de la stabilité de l'application
  - Principe des dépendances acycliques
  - Principe de relation dépendance/stabilité
  - Principe de stabilité des abstractions
- **Principe de responsabilité unique**

Principe d'ouverture-fermeture

- *Open-Closed Principle* (OCP)
- Tout module (paquetage, **classe**, méthode) doit être ouvert aux extensions mais fermé aux modifications
  - Ouvert aux extensions : peut être étendu par des comportements
  - Fermé aux modifications : extensions introduites sans modifier le code
- Utilisation des mécanismes objets : encapsulation, héritage, classes abstraites, interfaces...
- Identifier les points d'ouverture/fermeture en fonction des
  - besoins d'évolutivité (client)
  - besoins de flexibilité (développeurs)
  - changement répétés constatés (développement)

Exemple


PrefInstrument
modele: String
getFabricant() : Fabricant
getModele() : String
getType() : Type
getBoisFond() : Wood
getBoisTable() : Wood
correspond(PrefInstrument) : boolean
PrefGuitare
nbCordes : int
getNbCordes() : int
correspond(PrefGuitare) : boolean
PrefMandoline
getStyle() : Style
correspond(PrefMandoline) : boolean
PrefInstrument est une classe de base abstraite. Elle définit une méthode correspond() dont l'implémentation de base permet de chercher les correspondances entre les classes Pref.
Chacune des sous-classes spécifiques à un instrument étend la méthode correspond()... Elles utilisent la version de base de PrefInstrument, et ensuite, elles ajoutent des détails de correspondance particuliers au type d'instrument sur lequel elles travaillent.
Les parties de correspond() qui changent selon les sous-classes sont encapsulées hors de la classe de base PrefInstrument.

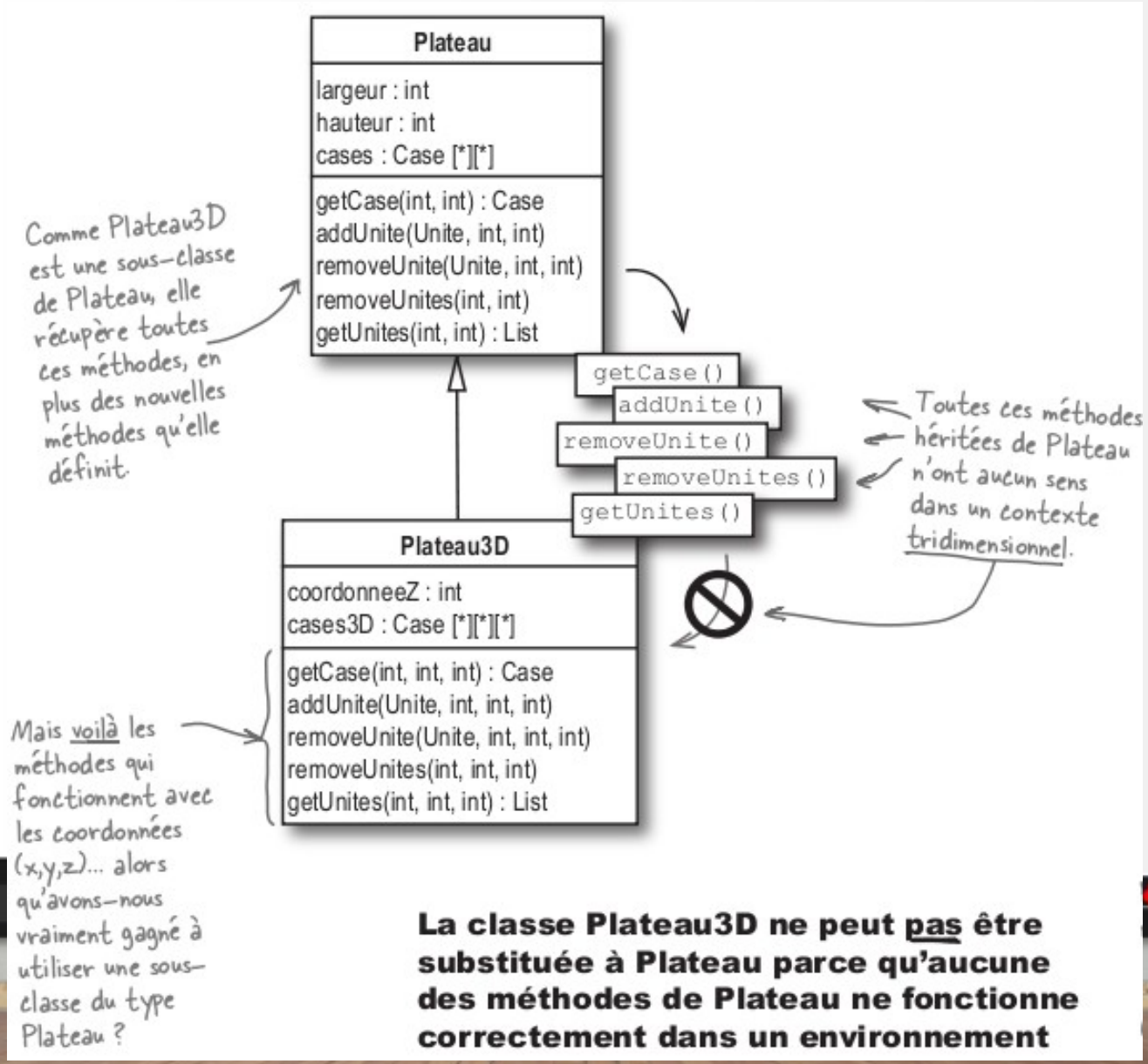
Principe de substitution de Liskov

- *Liskov Substitution Principle* (LSP)
- ***Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir***
- Sous-classe conçue de sorte que ses instances puissent se substituer à des instances de la classe de base
- Peut s'opposer à l'utilisation de l'héritage comme factorisation !
- L'héritage comme offre de service
- *Design by Contract*
  - la redéfinition d'une méthode dans une peut affaiblir les préconditions mais doit garder ou renforcer les postconditions

Exemple de mauvaise utilisation d'héritage

```
class Utilisateur {
   private String nom;
   // post =nom==nouveauNom
   public changeNom(String nouveauNom)
   { nom = nouveauNom; }
}
class Personne extends Utilisateur {
   private String prenom;
   public changeNom(String nouveauNom)
   { prenom = nouveauNom; }
}
```

Autre exemple



Exemple suite

- *« Les sous-types doivent pouvoir être substitués à leurs types de base »*

```
Plateau plateau = new Plateau3D();
```

Du point de vue du compilateur, Plateau3D peut être utilisé à la place de Plateau ici.

```
Unite unite = plateau.getUnites(8, 4);
```

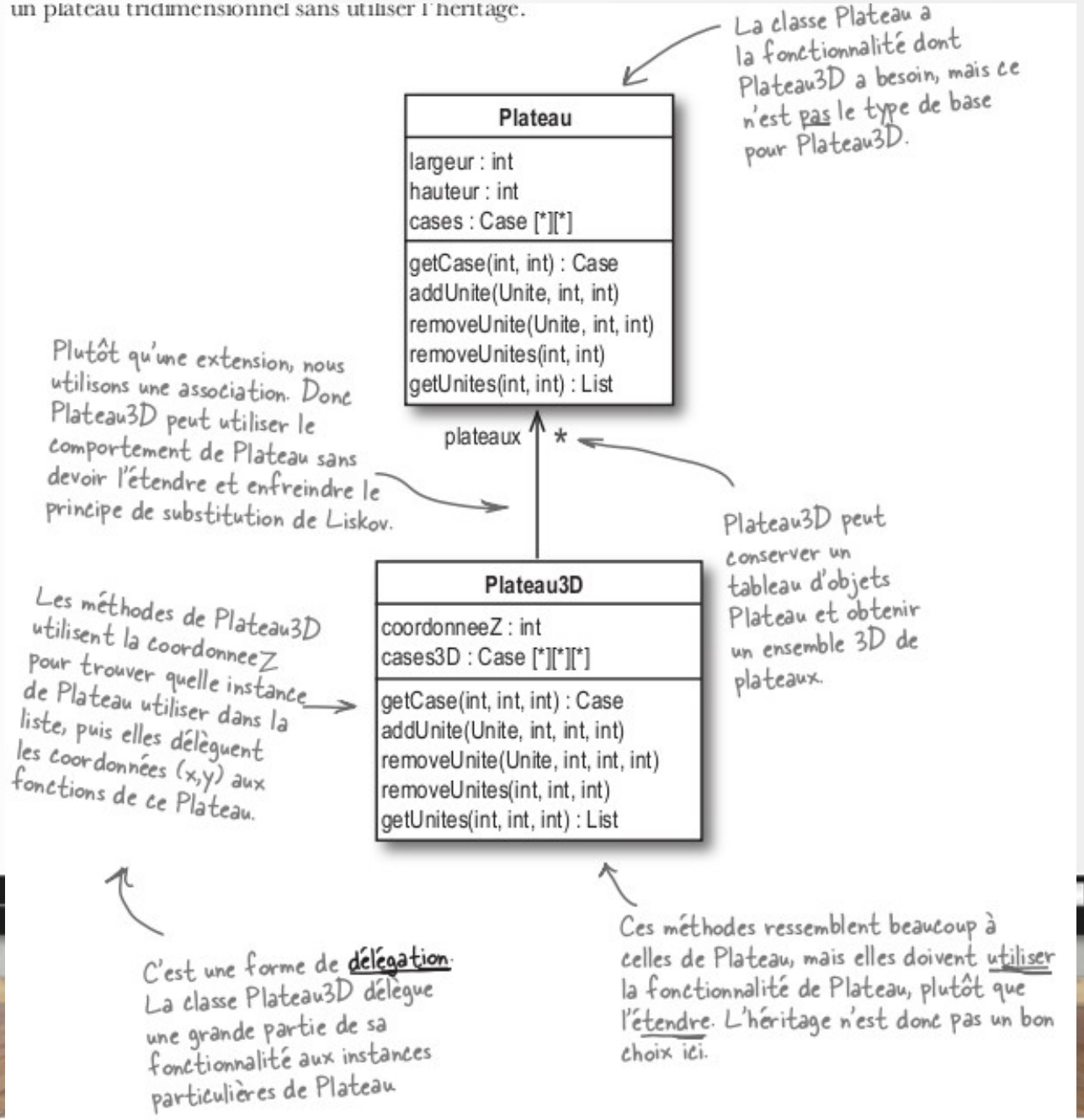
Rappelez-vous : plateau est ici une instance du sous-type, Plateau3D.

Mais que signifie cette méthode dans Plateau3D ?

Comment résoudre le pb du Plateau3D sans héritage ?

Délégation !

un plateau tridimensionnel sans utiliser l'heritage.

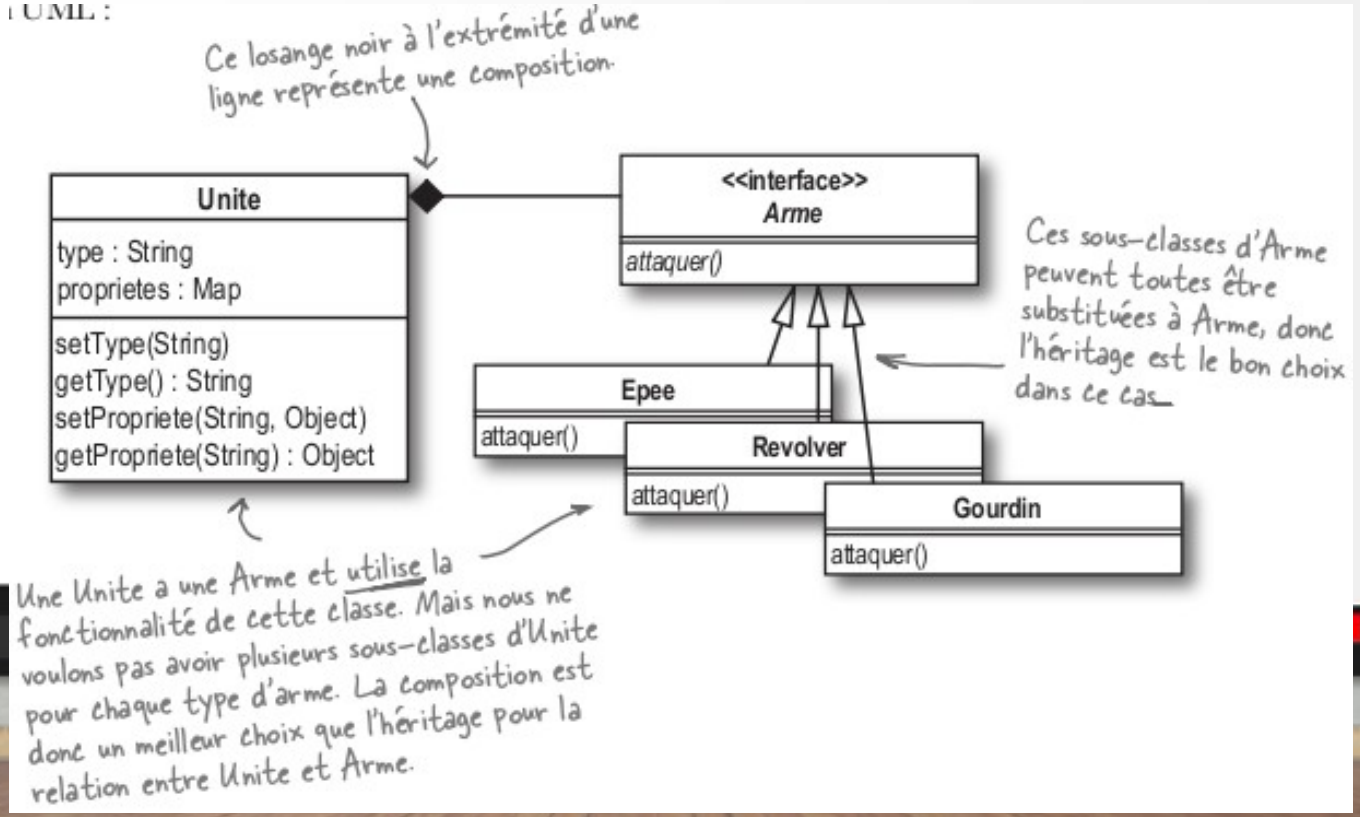


Délégation au lieu d'héritage

- Si vous voulez utiliser la fonctionnalité d'une autre classe, mais sans changer cette fonctionnalité, pensez à utiliser la délégation plutôt que l'héritage
- La **délégation** se modélise par une **association** UML
- La délégation ne convient que si le comportement de l'objet à qui vous déléguez le comportement **ne change jamais**

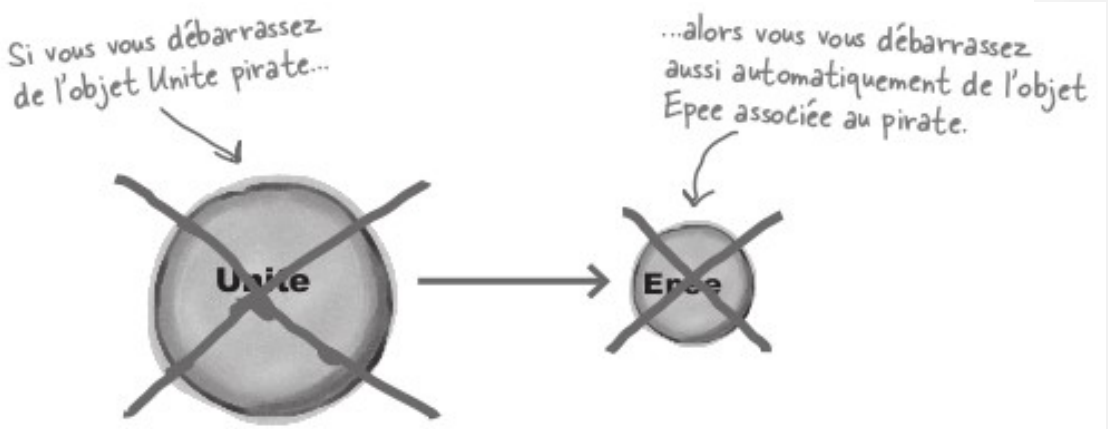
Composition au lieu d'héritage

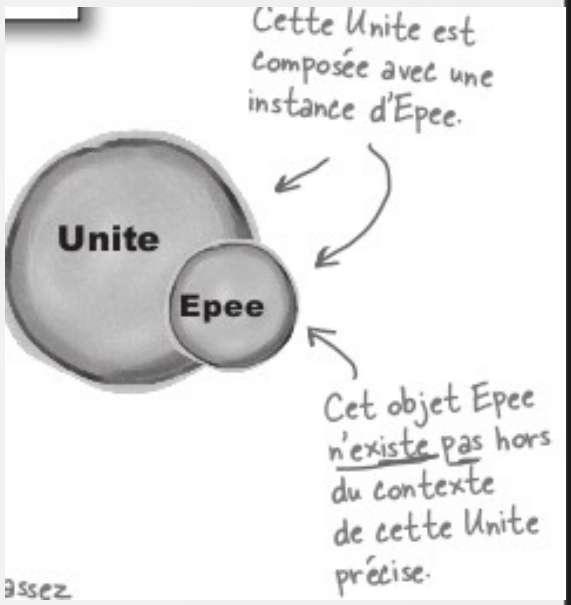
- Utilisez la composition pour assembler **des** comportements d'autres classes
- La composition vous permet d'utiliser le comportement d'une famille d'autres classes et de **changer** ce comportement **pendant** le fonctionnement de l'application



Composition au lieu d'héritage (2)

```
Unite pirate = new Unite();
pirate.setPropriete("arme", new Epee());
```

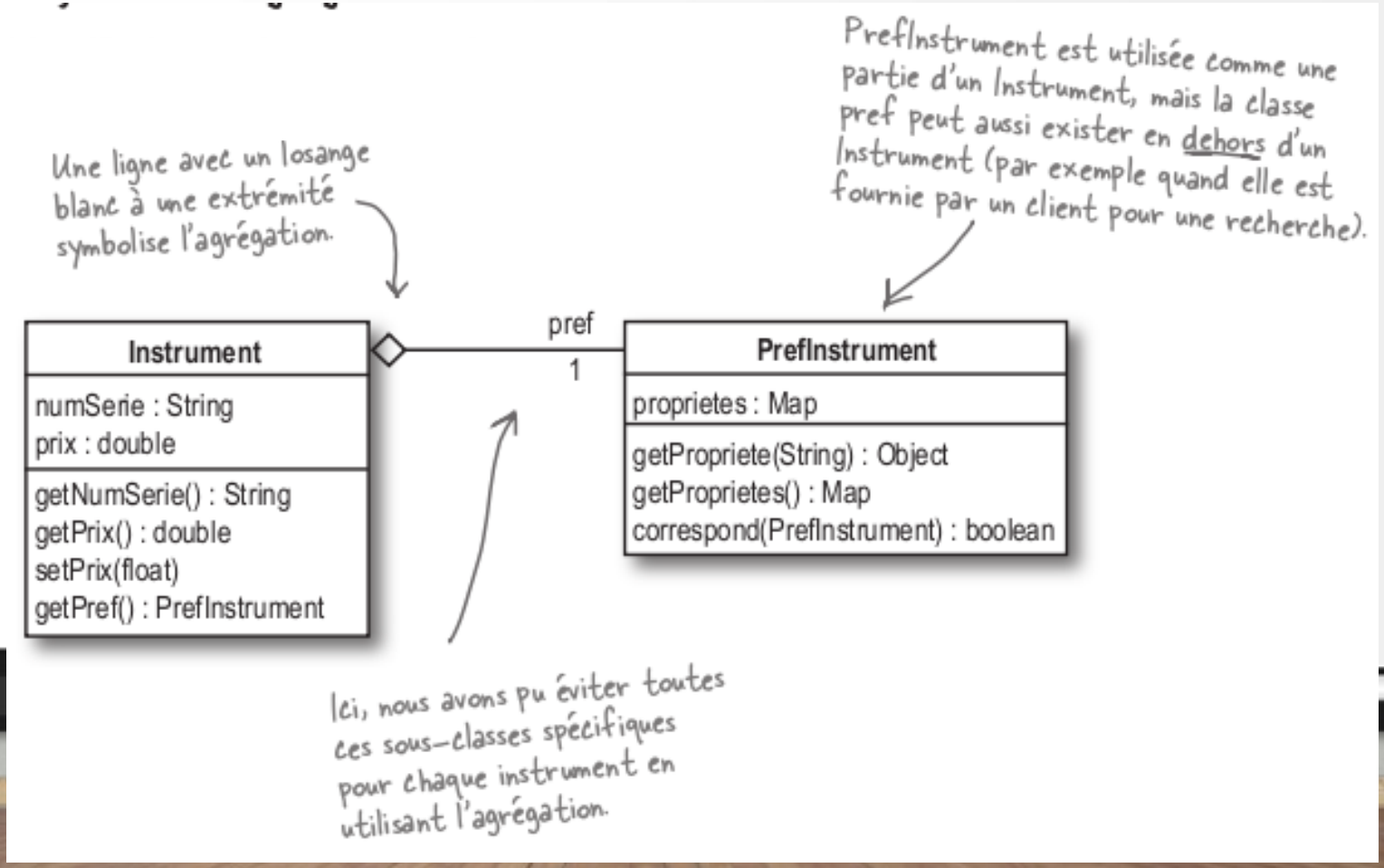




- Avec la composition, l'objet composé des autres comportements possède ces comportements
- Quand l'objet est détruit, tous ses comportements le sont aussi
- Les comportements, dans une composition, n'existent pas en dehors de la composition elle-même

L'agrégation : composition sans fin brutale

- L'agrégation désigne le fait qu'une classe est utilisée comme une partie d'une autre classe, mais continue à exister en dehors de cette autre classe



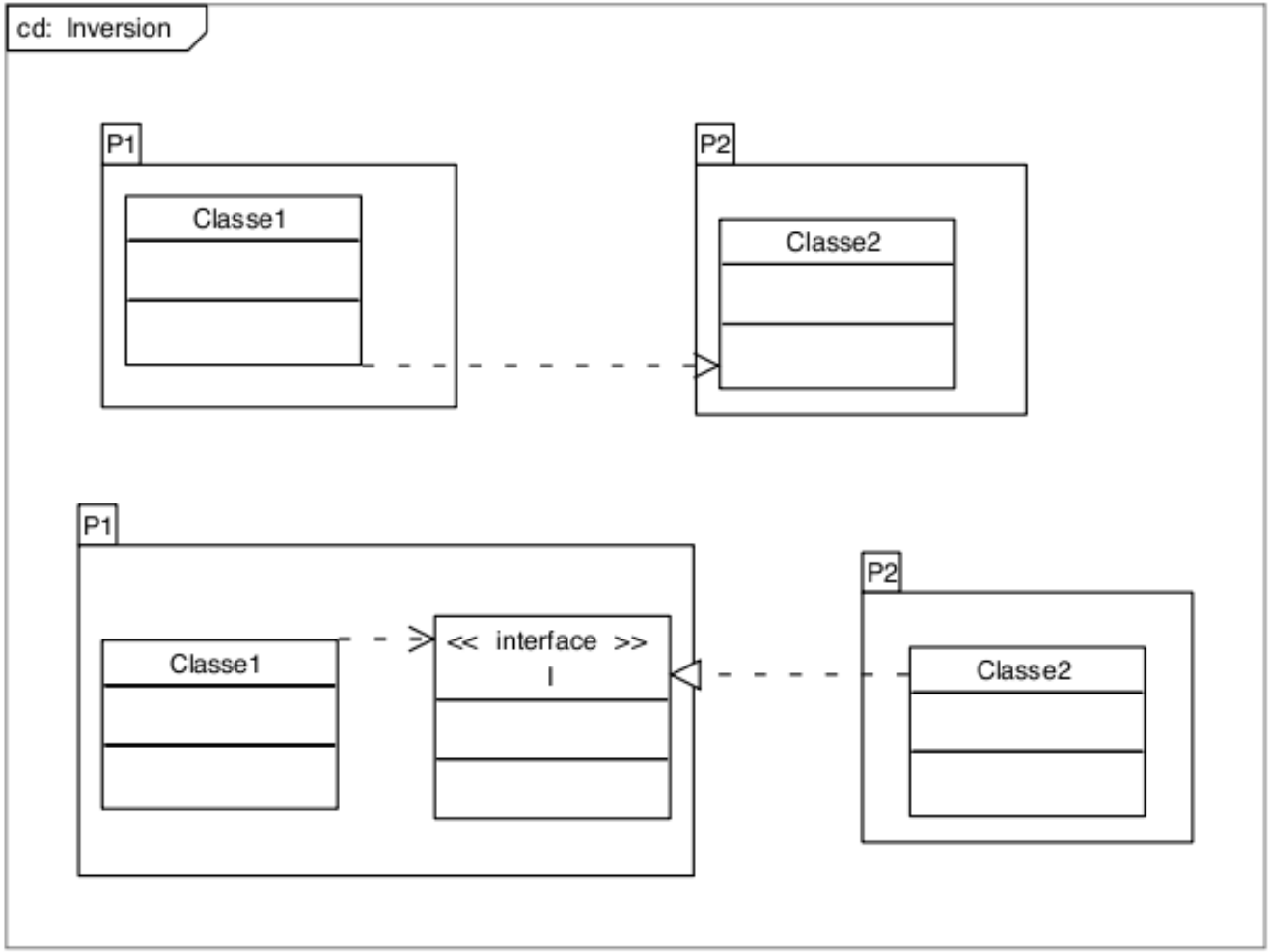
L'héritage n'est qu'une option !

- Si vous préférez la **délégation**, la **composition** et **l'agrégation** plutôt que **l'héritage**, votre logiciel sera généralement plus **souple** et plus facile à **maintenir**, **étendre** et **réutiliser**

Principe d'inversion des dépendances

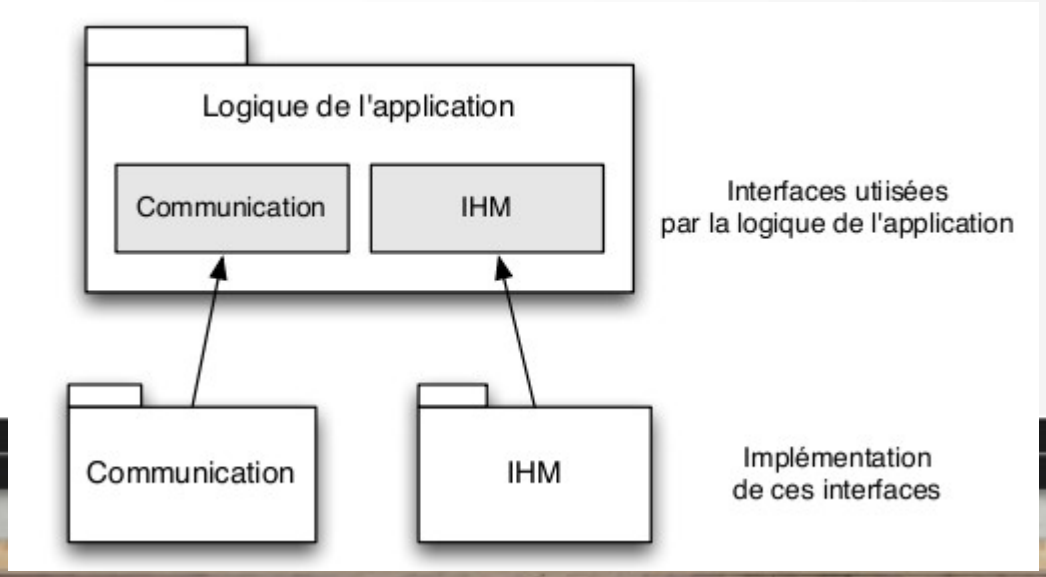
- *Dependency Inversion Principle* (DIP)
- Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions
- Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions
- Modules de bas niveau plus susceptibles d'être modifiés en cas de changement d'environnement d'exécution
- Modules de haut niveau (métier) doivent dépendre d'abstractions (interfaces) modélisant les services nécessaires

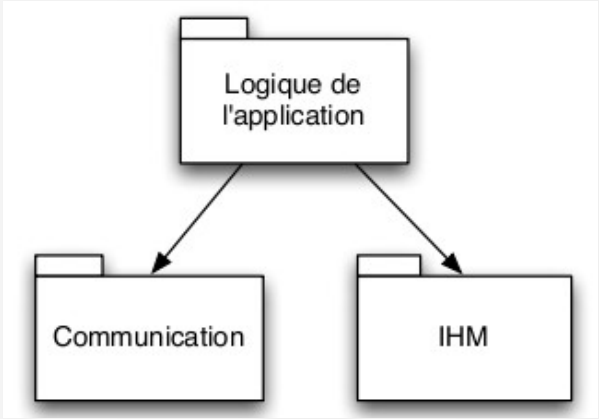
Exemple


cd: Inversion
P1
Classe1
P2
Classe2
P1
Classe1
<< interface >>
I
P2
Classe2

Autre exemple

- Pas bon
  - Les modules de haut niveau doivent être modifiés lorsque les modules de bas niveau sont modifiés
  - Il n'est pas possible de réutiliser les modules de haut niveau indépendamment de ceux de bas niveau. En d'autres termes, il n'est pas possible de réutiliser la logique d'une application en dehors du contexte technique dans lequel elle a été développée
- Bon !



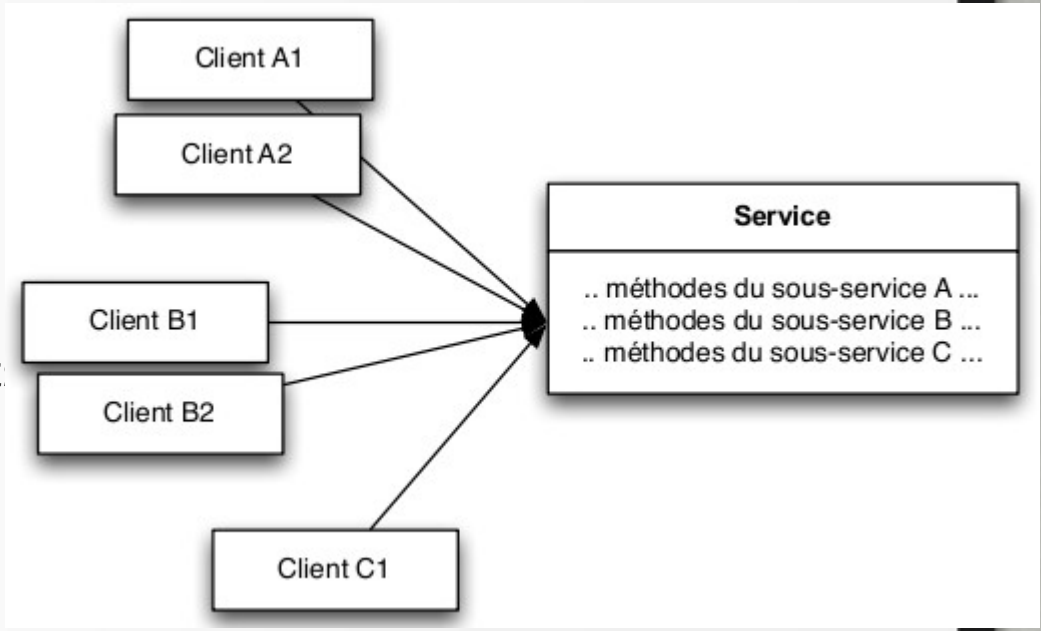


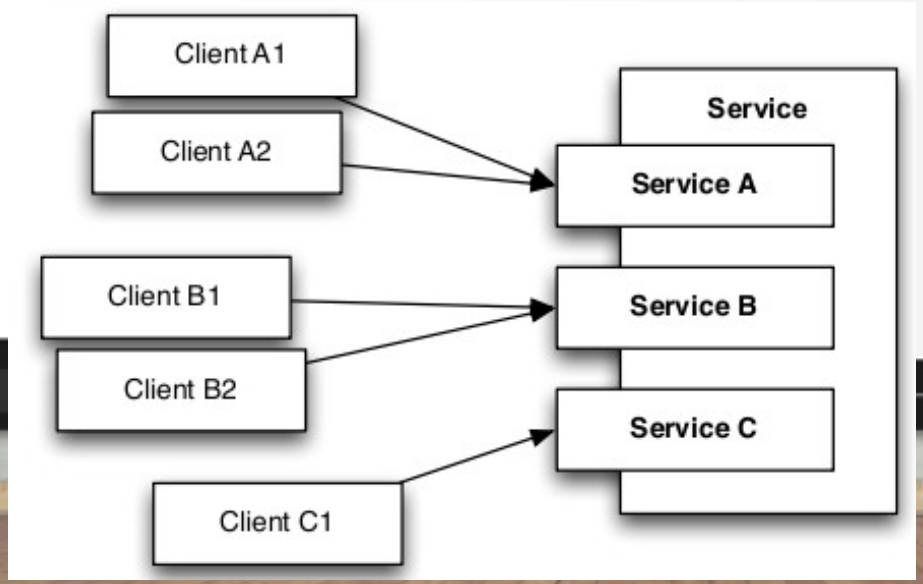
Principe de séparation des interfaces

- *Interface Segregation Principle* (ISP)
- Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas
- Chaque client ne doit «voir» que les services qu'il utilise réellement
- Evite une tentation courante : accumuler dans une classe un ensemble de services sous prétexte que la classe contient les informations nécessaires
- Solution : utiliser des interfaces différentes par type de client différent

Exemple

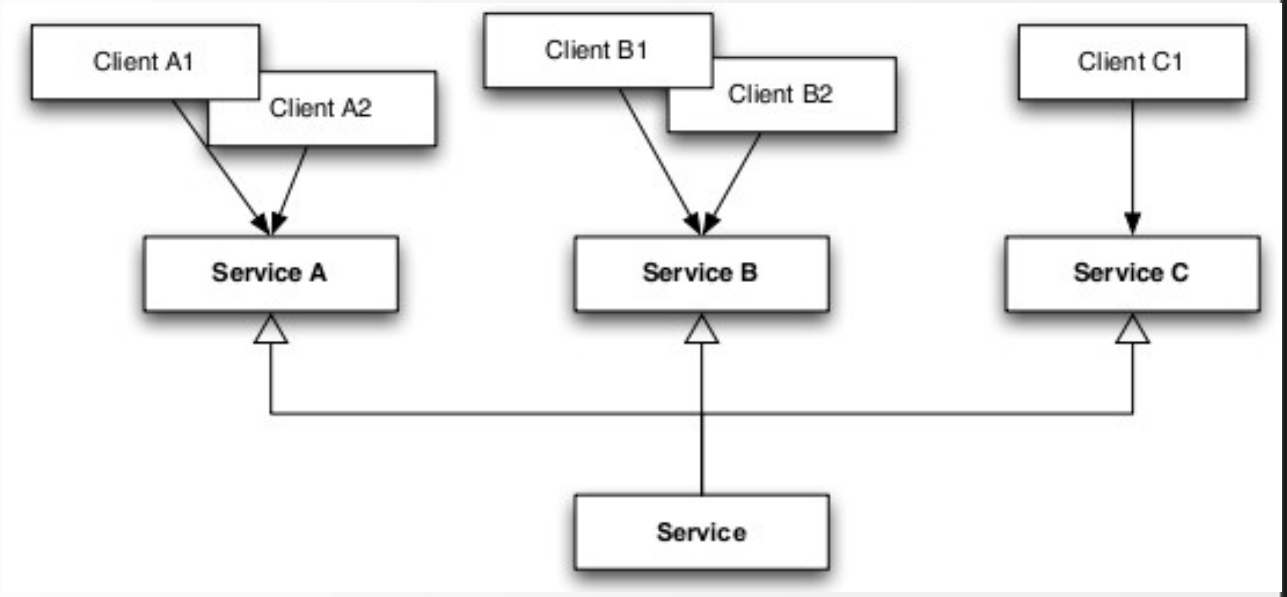
- Pollution !
  - Chaque client voit une interface trop riche dont une partie ne l'intéresse pas
  - Chaque client peut être impacté par des changement d'une interface qu'il n'utilise pas
- Solution



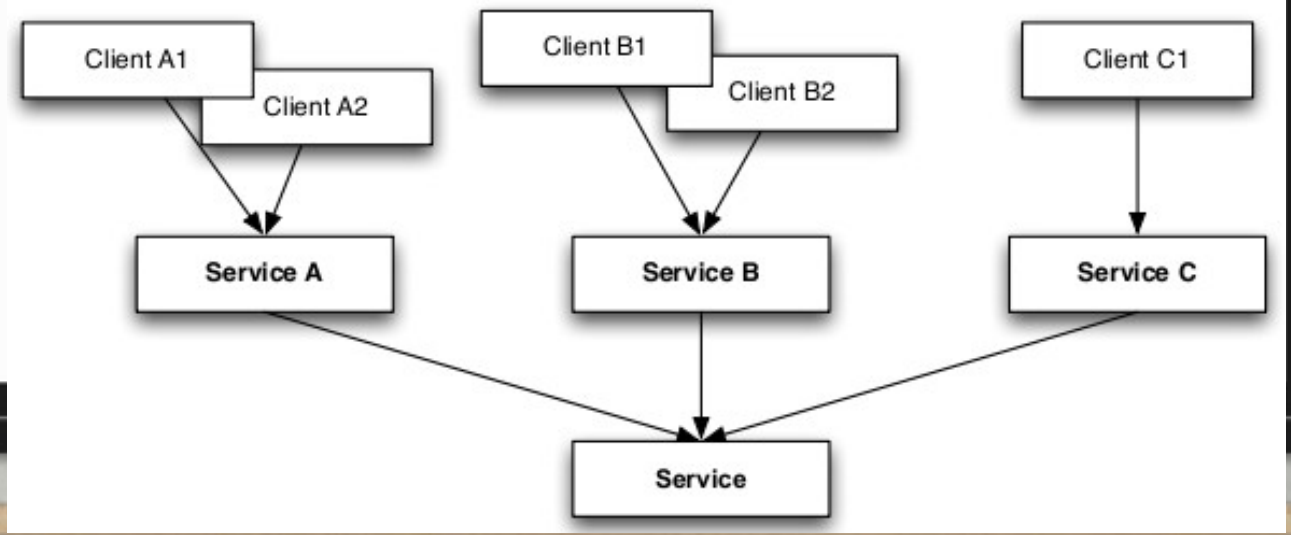


Mise en oeuvre

- Par héritage multiple (qd permis)



- Par classe d'adaptation



Principe d'équivalence livraison/réutilisation

- *Reuse/Release Equivalence Principle* (REP)
- La granularité en termes de réutilisation est le paquetage
- Seuls des paquetages livrés sont susceptibles d'être réutilisés
- Client doit réutiliser le paquetage avec une interface minimale
- Réutilisation efficace si
  - code réutilisé reste la propriété de son auteur, qui garde la charge de le corriger et de le faire évoluer
  - code réutilisé tel quel. Celui qui le réutilise doit se contenter de passer par des interfaces minimales, sans avoir à comprendre le fonctionnement interne du code utilisé

Principe de réutilisation commune

- *Common Reuse Principle* (CRP)
- Réutiliser une classe d'un paquetage, c'est réutiliser le paquetage entier
- Classe est rarement réutilisée seule
- Si plusieurs classes doivent être réutilisées ensemble : les intégrer au même paquetage (librairie)
- Ne pas placer deux classes totalement indépendantes dans un même paquetage
- Principe voisin du précédent

Principe de fermeture commune

- *Common Closure Principle* (CCP)
- Les classes impactées par les mêmes changements doivent être placées dans un même paquetage
- Fermeture complète d'une application impossible
- Reste toujours dans une application des endroits influencés par des changements
- Réduction de l'impact de ces changements et des coûts d'évolution et de maintenance : regroupement dans un même paquetage des classes concernées par un même changement

Principe des dépendances acycliques

- *Acyclic Dependencies Principle* (ADP)
- Les dépendances entre paquetages doivent former un graphe acyclique
- Peut être encore amélioré en préconisant une architecture en couches (paquetage d'une couche N ne peut utiliser que les services de paquetages de la couche N-1)
- Paquetages introduisent des points de synchronisation des changements dans l'application, propagation des changements guidée par les dépendances entre paquetages

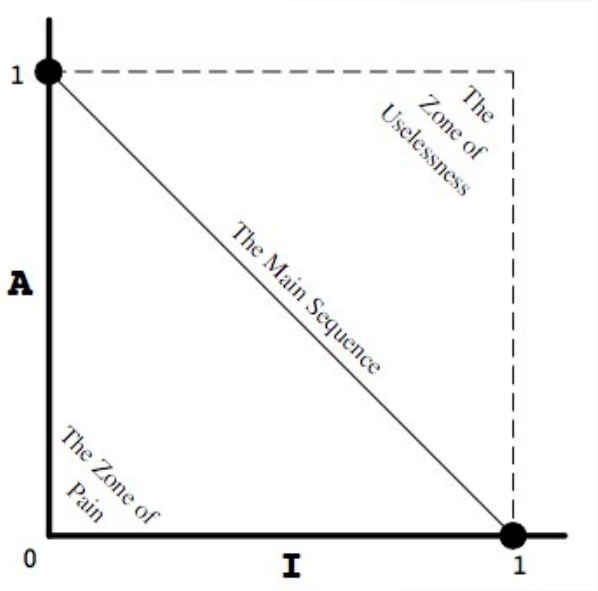
Principe de relation dépendance/stabilité

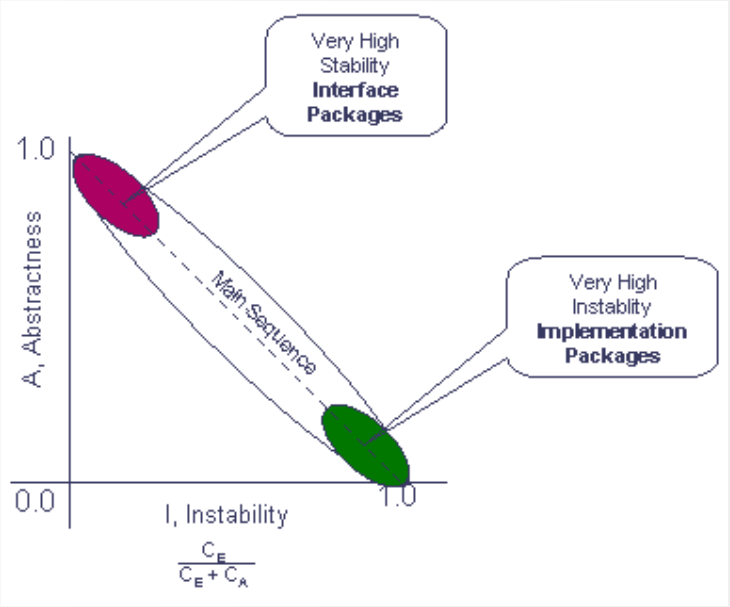
- *Stable Dependencies Principle* (SDP)
- Un paquetage dépend uniquement de paquetages plus stables que lui
- Stabilité (pour un module donné)
  - Plus le nombre de modules dont il dépend est grand, moins il est stable
  - Plus le nombre de modules dépendant de ce module est grand, plus les modifications de ce module sont coûteuses, et donc plus il est stable
- Stabilité du module est
  - Maximale si le module n'utilise aucun autre module et se trouve lui-même utilisé par un grand nombre de modules
  - Minimale si le module utilise de nombreux autres modules alors qu'il n'est utilisé lui-même par aucun autre module

Principe de stabilité des abstractions

- *Stable Abstractions Principle* (SAP)
- Les paquetages les plus stables doivent être les plus abstraits
- Les paquetages instables doivent être concrets
- Le degré d'abstraction d'un paquetage doit correspondre à son degré de stabilité
- Degré d'abstraction
  - A = (nb de classes abstraites)/(nb de classes)
  - entre 0 (concret) et 1 (abstrait)

Principe de stabilité des abstractions





Principe de responsabilité unique

Principe de responsabilité unique

- *The Single Responsability Principle* (SRP)
- Il ne doit jamais y avoir plus d'une raison pour une classe de changer
- Chaque responsabilité d'une classe peut induire des changements (modification des spécifications)
- Exemple : une classe rectangle responsable à la fois des opérations mathématiques et de l'affichage (deux raisons de changer)
- Dans ce contexte, responsabilité = raison de changer

Accronymes des principes de conception

- Ce qu'il faut éviter : **STUPID**
  - ***S****ingleton*
  - ***T****ight Coupling*
  - ***U****ntestability*
  - ***P****remature Optimization*
  - ***I****ndescriptive Naming*
  - ***D****uplication*
- Ce qu'il faut faire : **SOLID** !
  - ***S****ingle responsibility principle*
  - ***O****pen close principle*
  - ***L****iskov principle*
  - ***I****nterface segregation principle*
  - ***D****ependency inversion principle*

A friend of mine often says that there are two rules to optimize an application:
- don't do it;
- (for experts only!) don't do it yet.

Plan

- Rappel bases OO
- Premiers Principes OO
- Grands principes OO
- **Divers**
  - Approches de développement
  - Pratiques de programmation
- Design Patterns OO
- Restes

Approches de développement

- Développement itératif
- Développement orienté cas d'utilisation
- Développement orienté fonctionnalités
- Développement orienté tests

Développement itératif

- Un bon logiciel s'écrit par **itération**
- Travaillez sur la **vision d'ensemble**, puis **itérez** sur les **parties** de l'application jusqu'à ce qu'elle soit **terminée**
- 2 approches :
  - Orienté cas d'utilisation
  - Orienté fonctionnalités
- Les 2 approches sont portées par de **bonnes exigences**
- Les 2 ont pour but de livrer ce que **veut le client**
- À chaque fois que vous itérez, réévaluez vos décisions de conception et n'ayez pas peur de CHANGER quelque chose si cela améliore votre conception

Développement orienté cas d'utilisation

- Consiste à sélectionner un scénario dans un cas d'utilisation et à écrire du code pour réaliser ce scénario complet à travers le cas d'utilisation
- Puis vous prenez un autre scénario et vous y travaillez, etc., jusqu'à ce que tous les scénarios du cas d'utilisation soient réalisés
- Puis vous répétez l'opération avec le cas d'utilisation suivant, jusqu'à ce que tous vos cas d'utilisation fonctionnent

Développement orienté fonctionnalités

- Consiste à sélectionner une caractéristique spécifique dans votre application et à planifier, analyser et développer cette caractéristique jusqu'au bout
- Intéressant si il y a beaucoup de caractéristiques qui ne sont pas trop entrelacées

Développement orienté tests

- Vous devez tester votre logiciel pour **toute utilisation possible** et imaginable. Soyez créatif !
- N'oubliez pas de tester la **mauvaise utilisation** du logiciel également. Vous **verrez les erreurs plus tôt** et vos clients vous en seront reconnaissants
- Philosophie
  - On écrit d'abord les cas de test
  - Puis le code qui va réussir ces tests
- Le développement orienté tests (TDD) a pour but d'élaborer un comportement correct pour vos classes
  - *Approfondi dans la partie de cours dédiée*

Pratiques de programmation

- Programmation par contrat
- Programmation défensive

Programmation par contrat

- Quand vous écrivez un logiciel, vous créez aussi un **contrat** entre ce logiciel et les gens qui l'utilisent
- Le contrat précise la façon dont le logiciel va répondre/se comporter à certaines actions (exemple lorsqu'on demande une propriété inexistante à un objet)
- Si le client veut qu'une action donne un résultat différent alors vous changez le contrat

Programmation par contrat (2)

Il est possible que cette liste ne soit pas initialisée, donc cette méthode peut renvoyer null s'il n'y a pas d'armes pour cette unité.

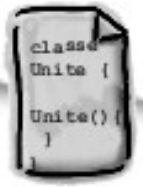
S'il n'y a aucune propriété, nous renvoyons null...

...et s'il n'y a pas de valeur pour la propriété de la requête, cela renverra null.

```
public List getArmes() {
  return armes;
}

// autres méthodes

public Object getPropriete(String propriete) {
  if (proprietes == null) {
    return null;
  }
  return proprietes.get(propriete);
}
```



Unite.java

Unite
type : String proprietes : Map
setType(String) getType() : String setPropriete(String, Object) getPropriete(String) : Object

le saviez pas, ce code définit un contrat pour ce qui se passe quand une propriété n'existe pas.

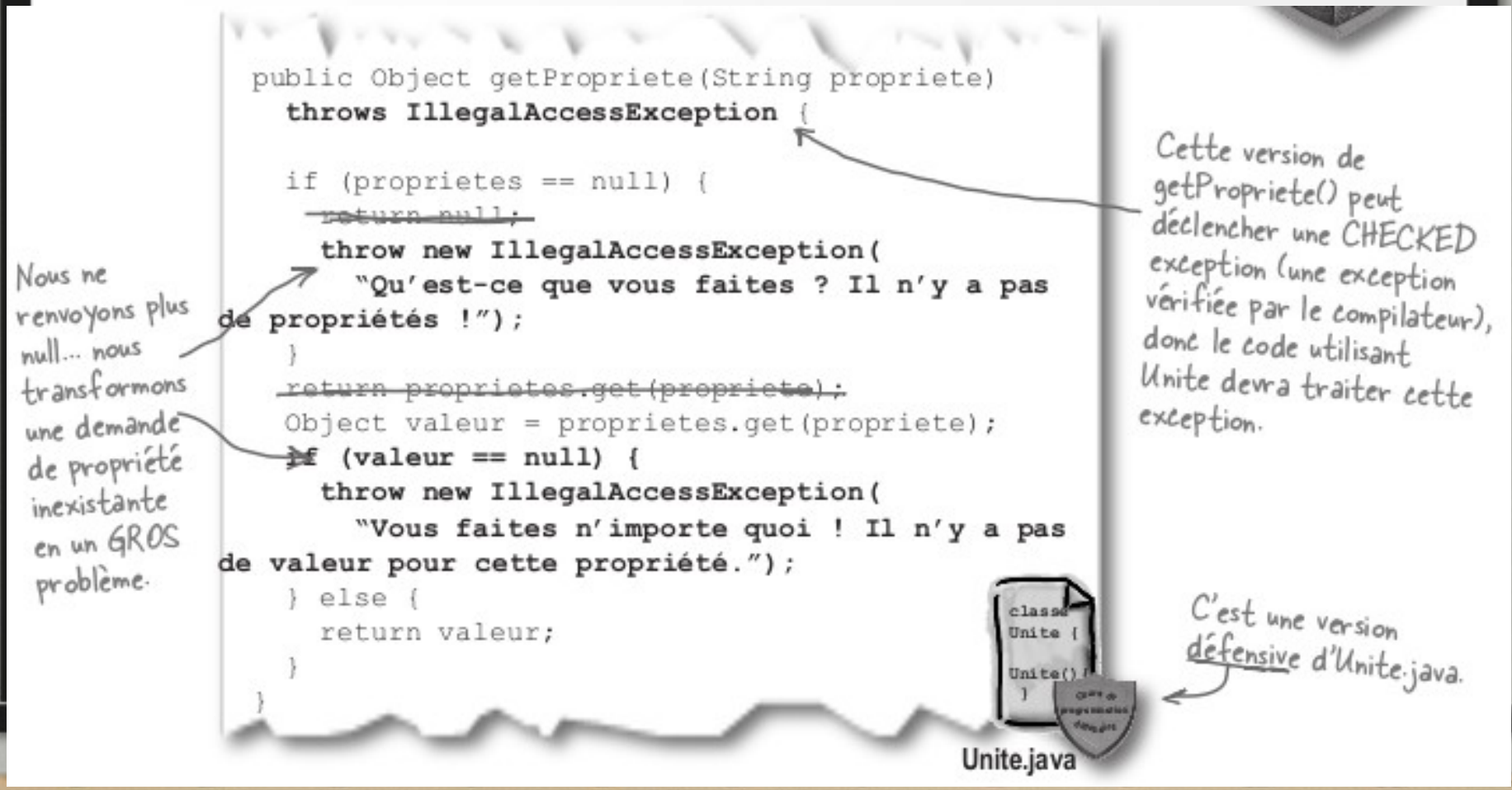
- La classe *Unite* assume que les gens qui l'utilisent sont des programmeurs compétents et qu'il peuvent gérer des valeurs de retour nulles

Programmation par contrat (3)

- Programmer par contrat est une question de **confiance**
  - Quand vous renvoyez *null*, vous faites confiance aux programmeurs pour savoir gérer les valeurs de retour nulles. Les programmeurs disent en substance qu'ils ont bien codé et qu'ils ne vont pas demander des propriétés ou des armes inexistantes, donc leur code ne s'occupe tout simplement pas des valeurs nulles que peut renvoyer la classe Unite
- Et nous pouvons toujours **changer** le contrat si besoin est…
  - Ex. : on nous a demandé d'arrêter de renvoyer null et de lever une exception à la place. Ce n'est vraiment pas un changement majeur dans le contrat, cela signifie seulement que les concepteurs de jeux vont maintenant avoir de gros problèmes s'ils demandent des propriétés ou des armes inexistantes

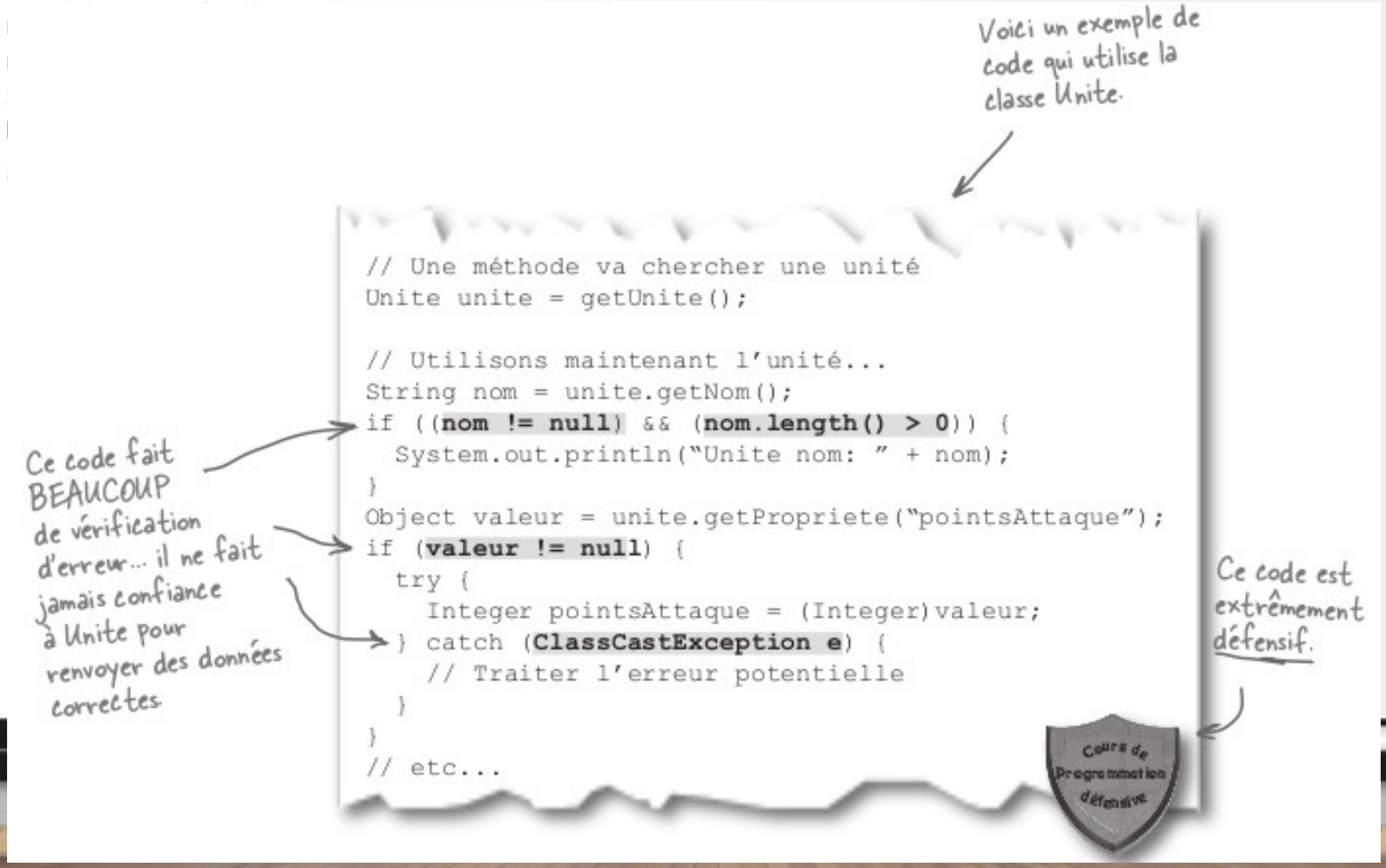
Programmation défensive

- Exemple : pas de confiance aux utilisateurs…



Programmation défensive

- Exemple : pas de confiance aux programmeurs…



Différences

- En programmation par contrat, ce qui compte, c'est la façon dont vos clients sont impliqués dans cette décision. Vous travaillez avec le client pour vous mettre d'accord sur la façon dont vous allez gérer les problèmes
- En programmation défensive, vous prenez cette décision de façon à ce que votre code ne plante pas, peu importe ce que le client en pense. Vous vous assurez de ne pas être responsable du plantage d'un programme et vous faites le maximum pour essayer d'empêcher les clients de planter leur programme