**UNIVERSITE DE CORSE**
**Licence ST 3ème année**
**Option INFORMATIQUE**

# UE Qualité Logicielle et Tests
# CH3 – Bonnes pratiques OO
# Patterns GRASP




Evelyne VITTORI
vittori@univ-corse.fr

# Plan du cours

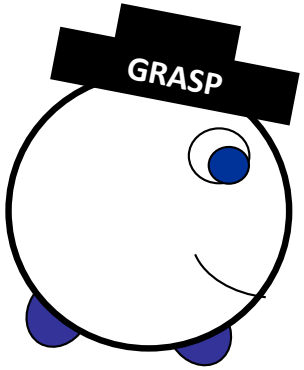

CH1 – Principes de Qualité

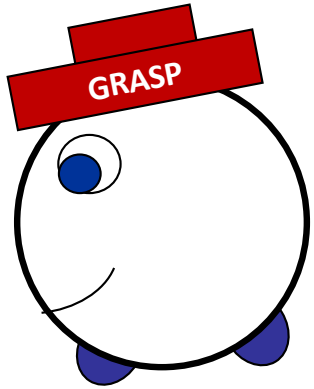
CH2 – Métriques logicielles


CH3 - Bonnes pratiques OO

CH4 – Tests

# Patterns Grasp



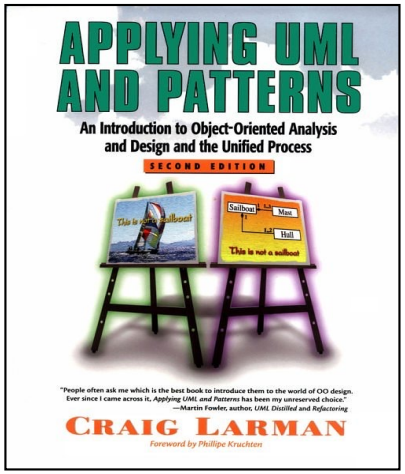




Università di Corsica
Pasquale Paoli

# Patterns Grasp

- Patterns généraux d'affectation des responsabilités (1997)

G.R.A.S.P « General Responsability Assignement Software Patterns »




- Formalisation des principes d'identification des méthodes et de leur répartition entre les classes
- Guide pour la réalisation des cas d'utilisation (analyse) : passage au codage.

**Craig LARMAN**

# Objectifs des  Patterns Grasp

Guide pour implémenter un cas d'utilisation

- Identifier les classes impliquées  ?
- Identifier les méthodes?
- Affecter les méthodes aux classes ?
- Définir l'enchaînement dynamique des invocations?

Comment répartir les responsabilités entre les différentes classes pour assurer la réalisation d'un scénario en assurant la qualité du code?

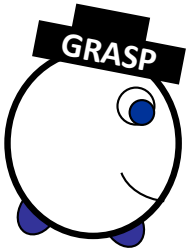
Guide des « bonnes pratiques OO »

# Liste des Patterns GRASP

## 9 principes de conception

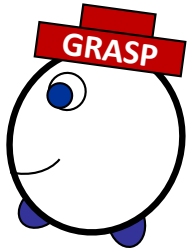
Patterns élémentaires

- Expert
- Créateur
- Faible couplage
- Forte Cohésion




Patterns spécialisés

- Contrôleur
- Polymorphisme
- Indirection
- Fabrication Pure
- Protection des variations

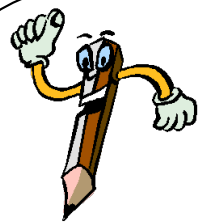


# EXPERT

**Nom** Expert en Information


**Problème** Trouver la classe responsable d'une opération

**Solution** Affecter l'opération à la classe qui possède les données nécessaires pour la réaliser



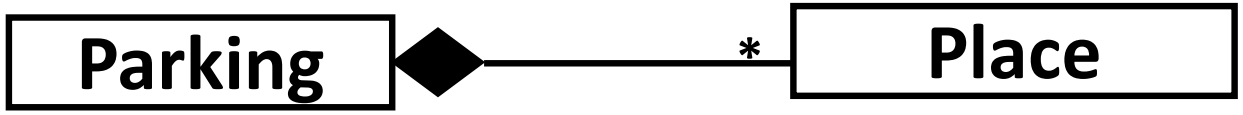
***Principe général d'affectation des responsabilités***

# Exemple : EXPERT

Placement de l'opération « Trouver place libre» dans un parking.

Données nécessaires

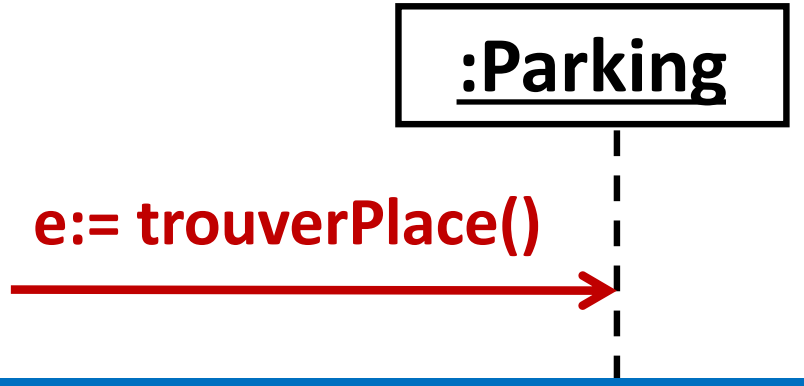
Ensemble des places





**Parking dispose de toutes les informations concernant l'ensemble des places.**

**Parking est « expert » pour «trouver place».**




**Parfois, ce n'est pas si simple: rechercher l'expert dominant**

# Exercice= Expert

On s'intéresse à la réalisation de l'opération getTotalGene() qui effectue le calcul du montant total d'une commande.

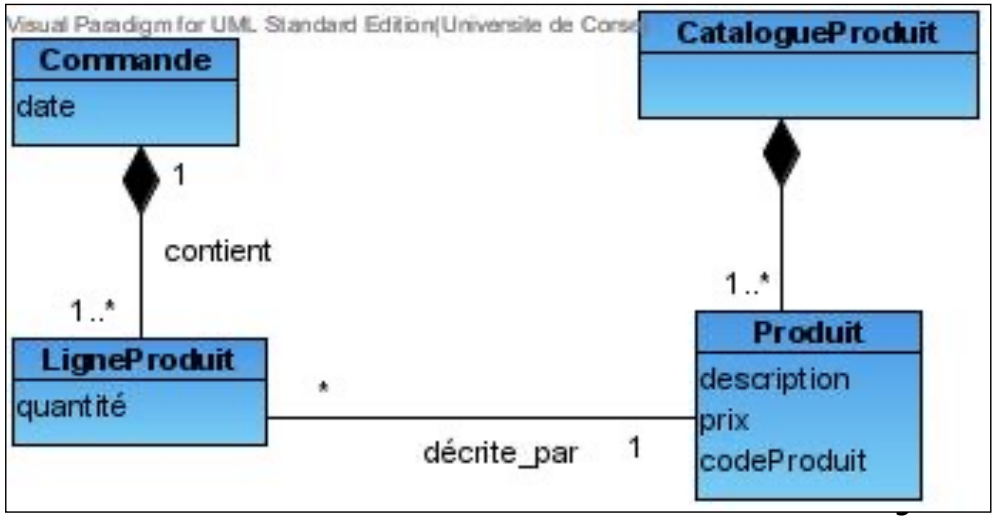
1. Identifiez la classe à laquelle attribuer cette responsabilité.

   Quelles sont les données nécessaires à cette opération? Quelle classe les possède?

2. Identifiez les opérations élémentaires impliquées dans l'implémentation de cette méthode

3. Pour chaque opération identifiée:

   Appliquez le pattern expert pour identifier les classes auxquelles en attribuer la responsabilité.



# Exercice= Expert (Correction)

1. Identifiez la classe à laquelle attribuer cette responsabilité.

   *Données nécessaires: liste des produits de la commande, avec leur quantité et leur prix unitaire*

   *Qui les connait? Seule la classe Commande possède toutes ces informations car elle est liée par une relation de composition avec la classe Ligne Produit.*

   *D'après le Pattern Expert, c'est donc la classe Commande qui doit assumer la responsabilité du calcul du montant total de la commande.*

2. Identifiez les opérations élémentaires impliquées dans l'implémentation de cette méthode

   *Les opérations élémentaires sont :*

   *getTotalLigne : réel* *classe LigneProduit*

   *getPrix : réel* *classe Produit*

3. Pour chaque opération identifiée:

   - Appliquez le pattern expert pour identifier les classes auxquelles en attribuer la responsabilité.

# CREATEUR


**Nom** Créateur

**Problème** Trouver la classe B responsable de la création des instances d'une classe A

**Solution** Affecter à B la responsabilité de créer une instance de A si :

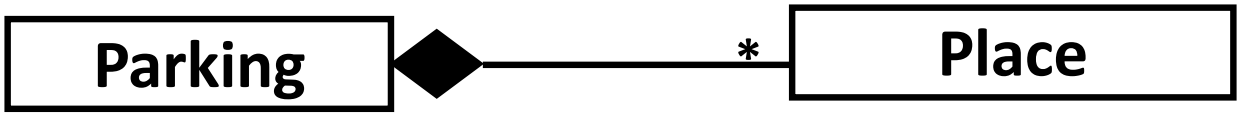
et/ou

1. B contient ou agrège des objets de A
2. B utilise étroitement des objets de A
3. B possède les données d'initialisation de A

# CREATEUR

**Exemple**

**Création d'une nouvelle place dans un parking.**





**Parking**

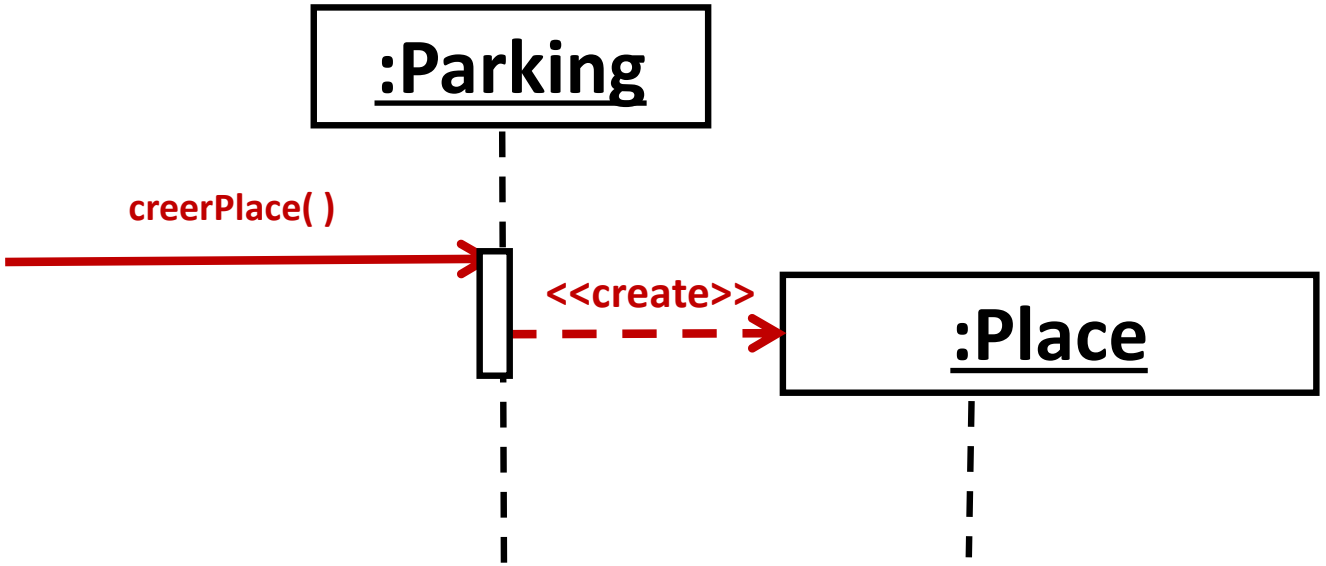
- **contient des places (1)**
- **utilise étroitement des places (2)**

**Parking peut-être «créateur» pour Place.**
**Parking est aussi «expert» pour la création d'une Place.**

# CREATEUR

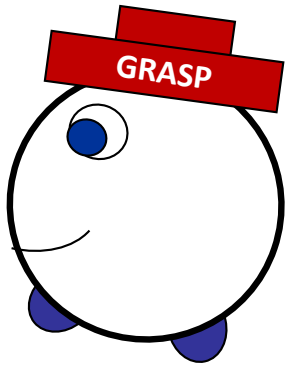
**Exemple**

**Création d'une nouvelle place dans un parking.**



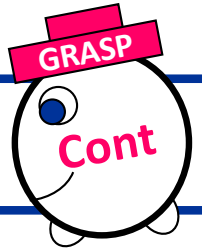
# Patterns GRASP spécialisés

- Contrôleur
- Polymorphisme
- Indirection
- Fabrication Pure
- Protection des variations



# Contrôleur

| | |
|---|---|
| **Nom** | **Contrôleur** |

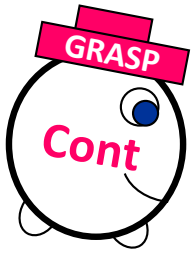



| | |
|---|---|
| **Problème** | **Trouver la classe *(n'appartenant pas à l'interface utilisateur)* qui reçoit et coordonne une opération « système ».** |

| | |
|---|---|
| **Solution** | **Créer une classe Contrôleur qui correspond à l'un des cas suivants :**<br>▪ **Classe représentant le système global : contrôleur de façade.**<br>▪ **Classe représentant un scénario de cas d'utilisation : contrôleur de session (ou de cas d'utilisation) .** |

# Contrôleur

**Conseils**




- **Contrôleur de façade**

**Souhaitable si les opérations systèmes sont peu nombreuses**

- **Conventions de nommage :**
  - **nom d'un objet physique global**
  - **nom du système :**
  - **System*<NomGlobal>***
  - **nom d'un équipement physique**

# Contrôleur

**Conseils**

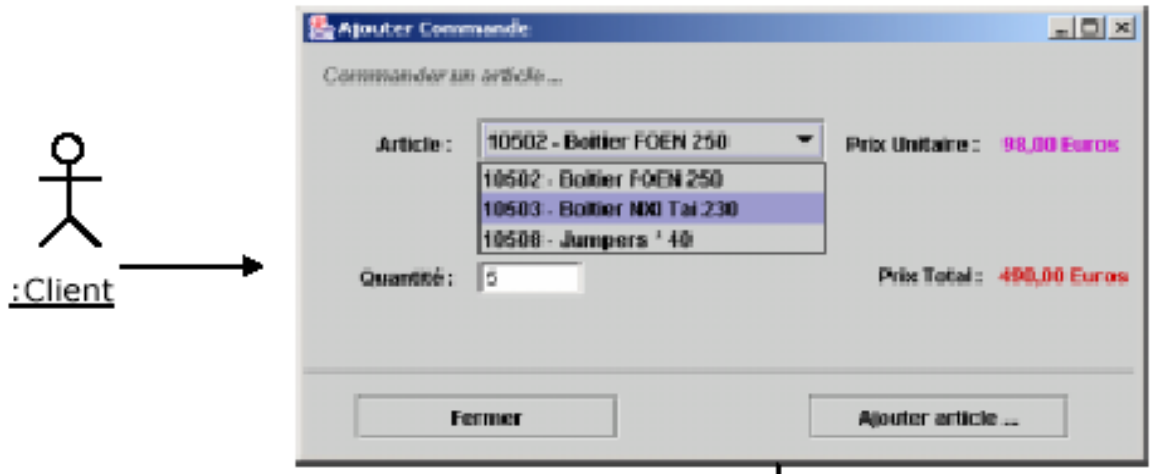
- **Utiliser le même contrôleur de UC pour tous les événements systèmes d'un scénario**

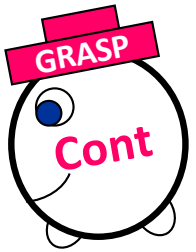
- **Conventions de nommage :**

  **Gestionnaire<NomUC>**
  **Coordinateur<NomUC>**
  **Session<NomUC>**
  **Controle<NomUC>**
  **Ctl<NomUC>**

# Contrôleur

***EXEMPLE***




actionPerformed()

**Couche Présentation**

:JCommandeFrame

ajouterArticle( article , quantité )

**Couche Domaine**

:GestionnaireCommande

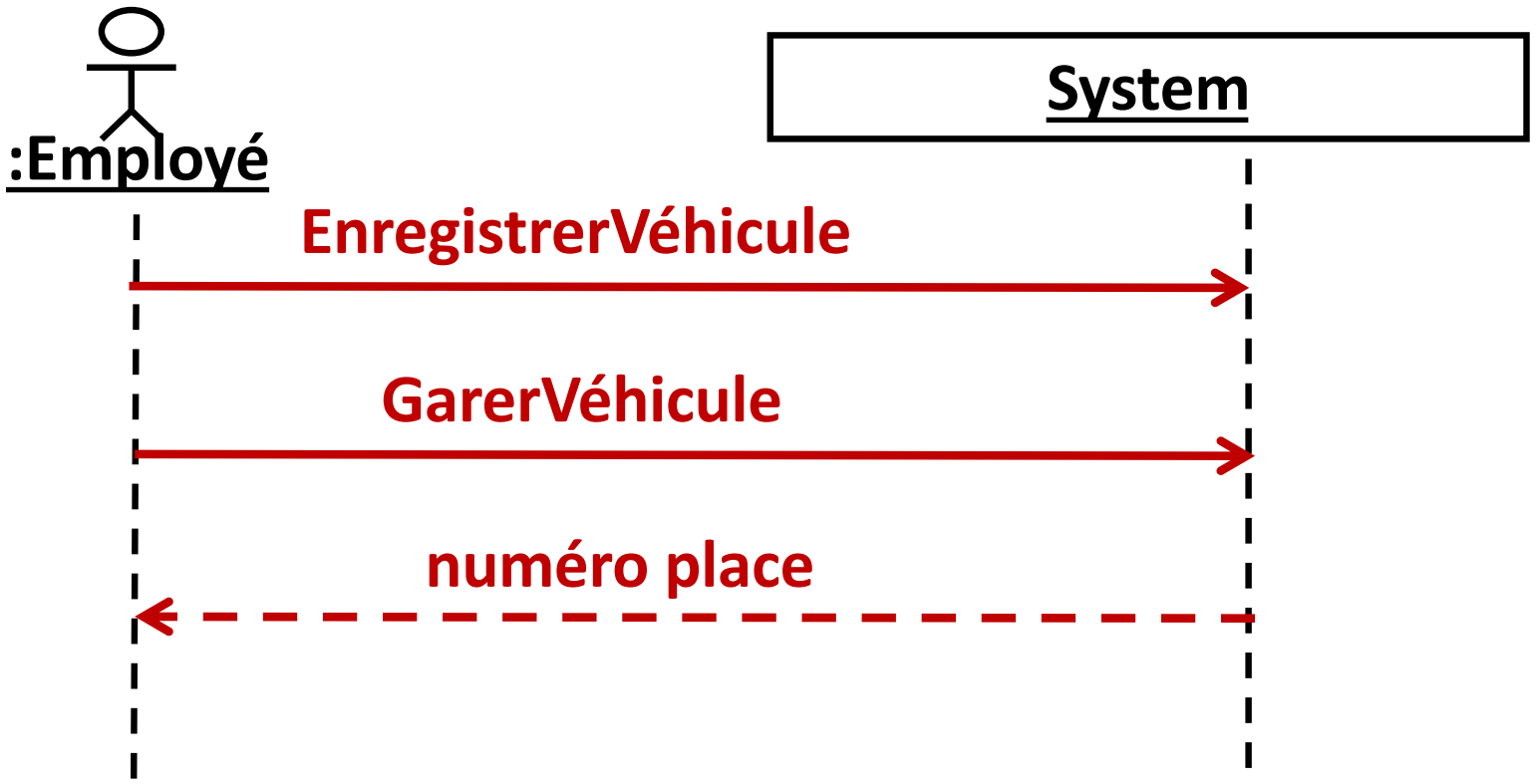
**Classe contrôleur**

# Contrôleur

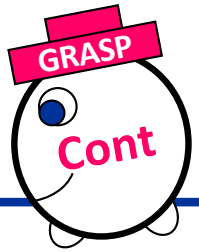
**Exemple**

**Qui est le contrôleur de l'opération système «EnregistrerVéhicule »?**

***DSS Arrivée Véhicule***



# Contrôleur

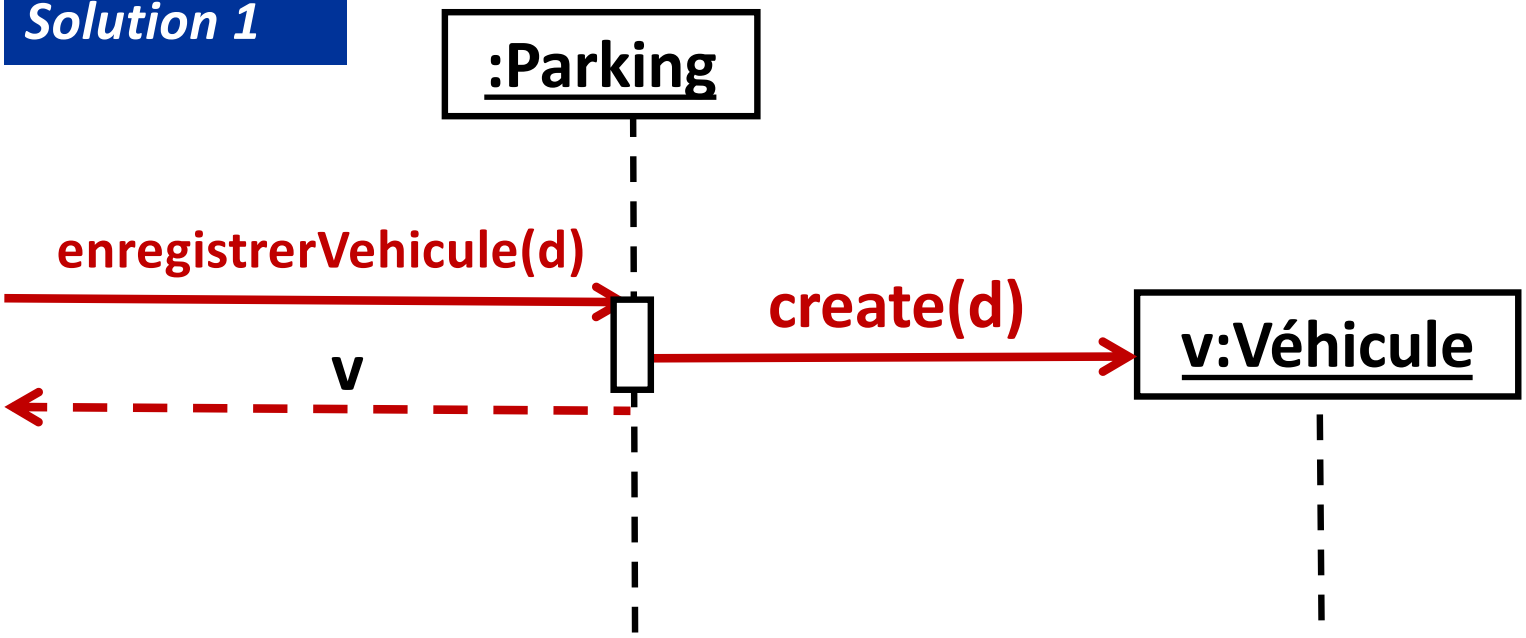



**Exemple**

**Qui est le contrôleur de l'opération système «enregistrerVéhicule »?**

**Parking joue le rôle de Controleur**

***Solution 1***

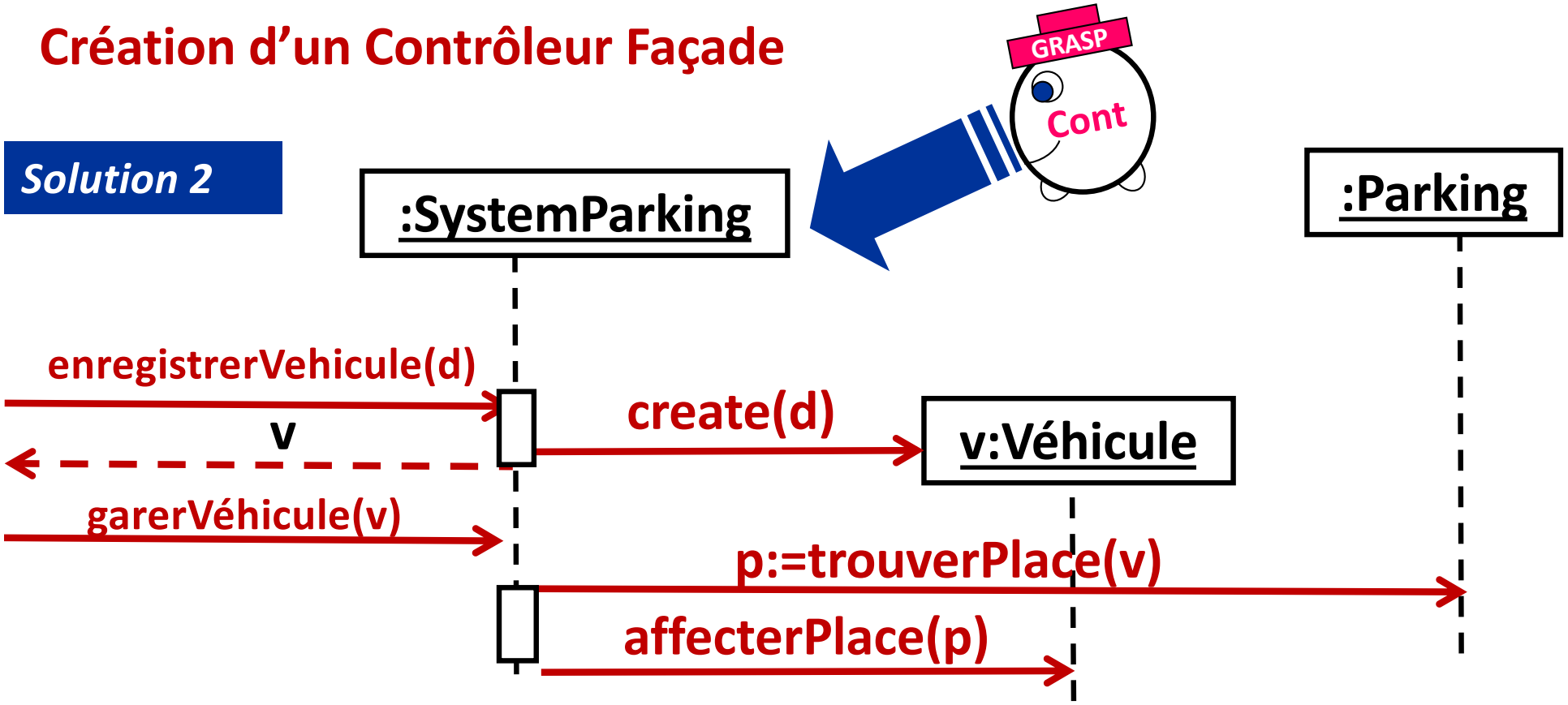


# Contrôleur

**Exemple**

**Qui est le contrôleur de l'opération système «enregistrerVéhicule »?**

## Création d'un Contrôleur Façade



# Contrôleur

**Exemple** **Qui est le contrôleur de l'opération système «enregistrerVéhicule »?**

***Solution 1***

**Couplage fort Parking-Véhicule**
**Diminution de la cohésion de Parking**

***Solution 2***

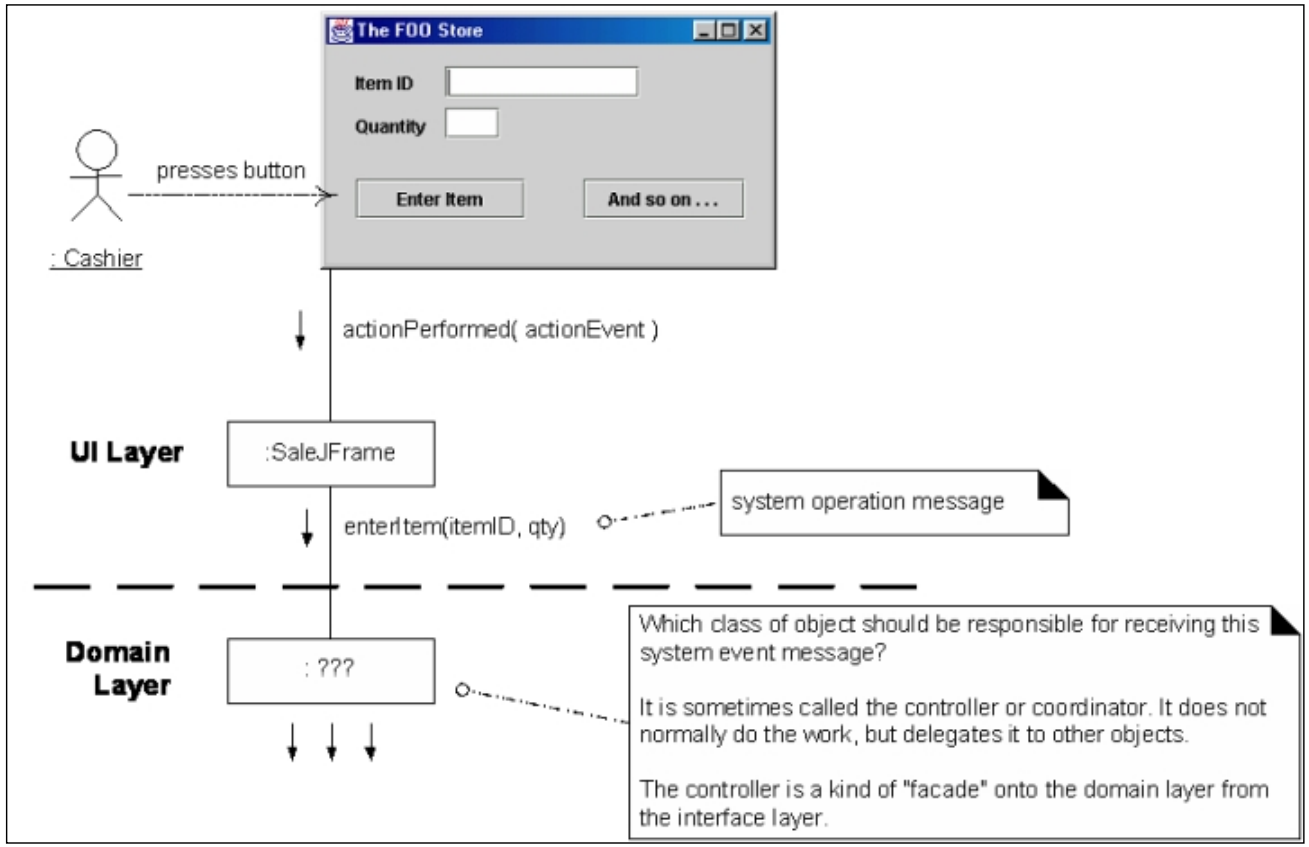
**« découplage » Parking-Véhicule**
**Amélioration de la cohésion de Parking**
**Meilleure répartition des responsabilités**

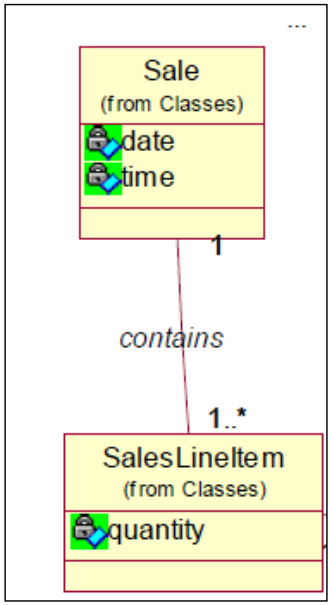
**Choix**

# Exercice= Controleur

Appliquez le pattern Controleur afin d'identifier ou de créer la classe responsable de la réalisation de l'opération système « enterItem » décrite dans l'illustration ci-dessous.

Expliquer les avantages de votre solution.



# Exercice= Controleur (correction)

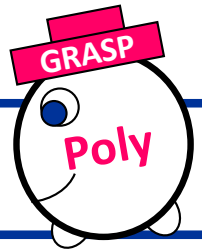
Appliquez le pattern Controleur afin d'identifier ou de créer la classe responsable de la réalisation de l'opération système « enterItem » décrite dans l'illustration ci-dessous.

Expliquer les avantages de votre solution.

- *Le pattern Controleur préconise l'utilisation d'une classe spécifique différente de la classe SaleJFrame pour jouer le rôle de controleur et assurer la réalisation de l'opération système « enterItem »*
- *Le pattern Controleur nous conduit donc à créer une classe CTRLVente chargée d'implémenter l'opération enterItem.*
- *La classe Spécifique Controleur a pour rôle de « découpler » la classe SaleJFrame des classes du modèle du domaine (Sale ici). Il s'agit de rendre les classes de l'interface indépendantes des classes du domaine ainsi si l'on change la nature de l'interface, le graphisme par exemple, les classes du domaine ne seront pas affectées. Seule la classe Controleur sera modifiée.*

# Polymorphisme

**Nom** Polymorphisme




**Problème**

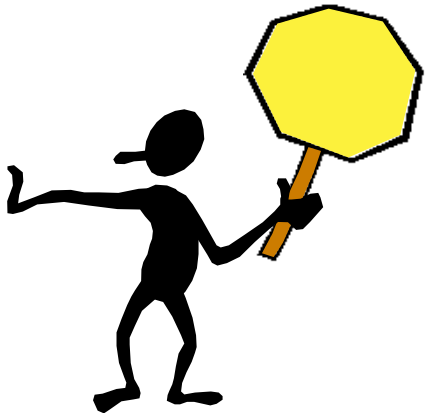
- Gérer des **alternatives** de comportement dépendantes des types (sans utiliser de schémas conditionnels).
- Créer des composants logiciels «enfichables » (modifiables sans effets de bord sur les composants clients)

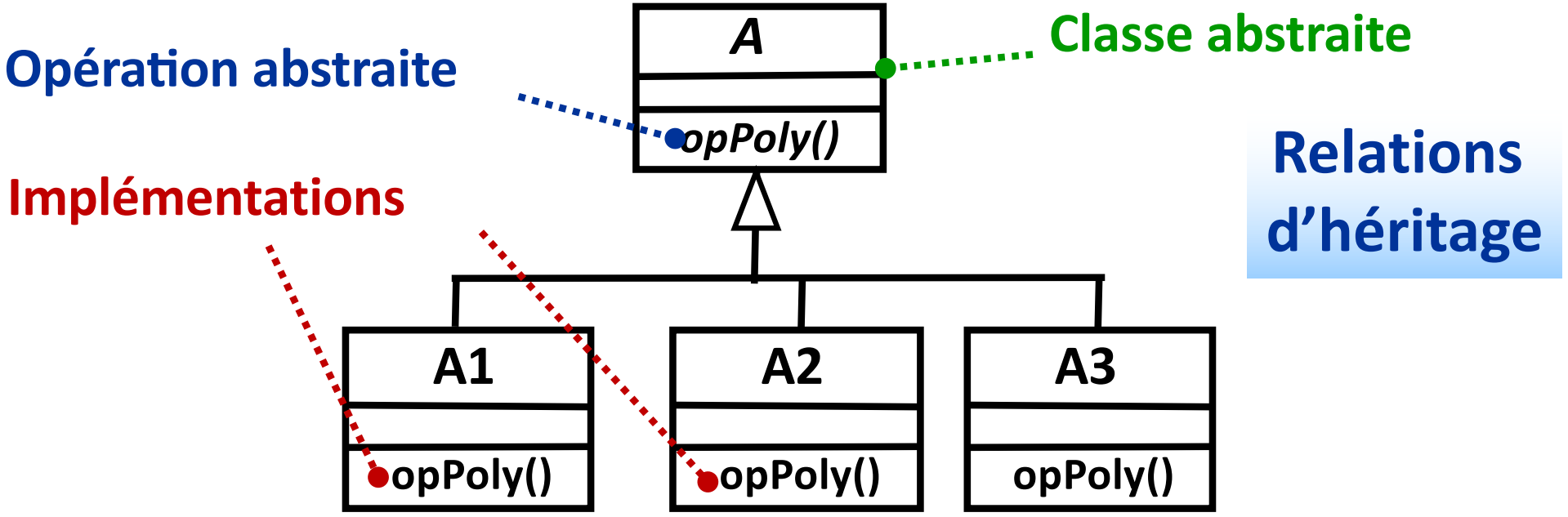
**Solution** Affecter les responsabilités aux types à comportement variable en utilisant des **opérations polymorphes**

# Polymorphisme

**Opération polymorphe** =

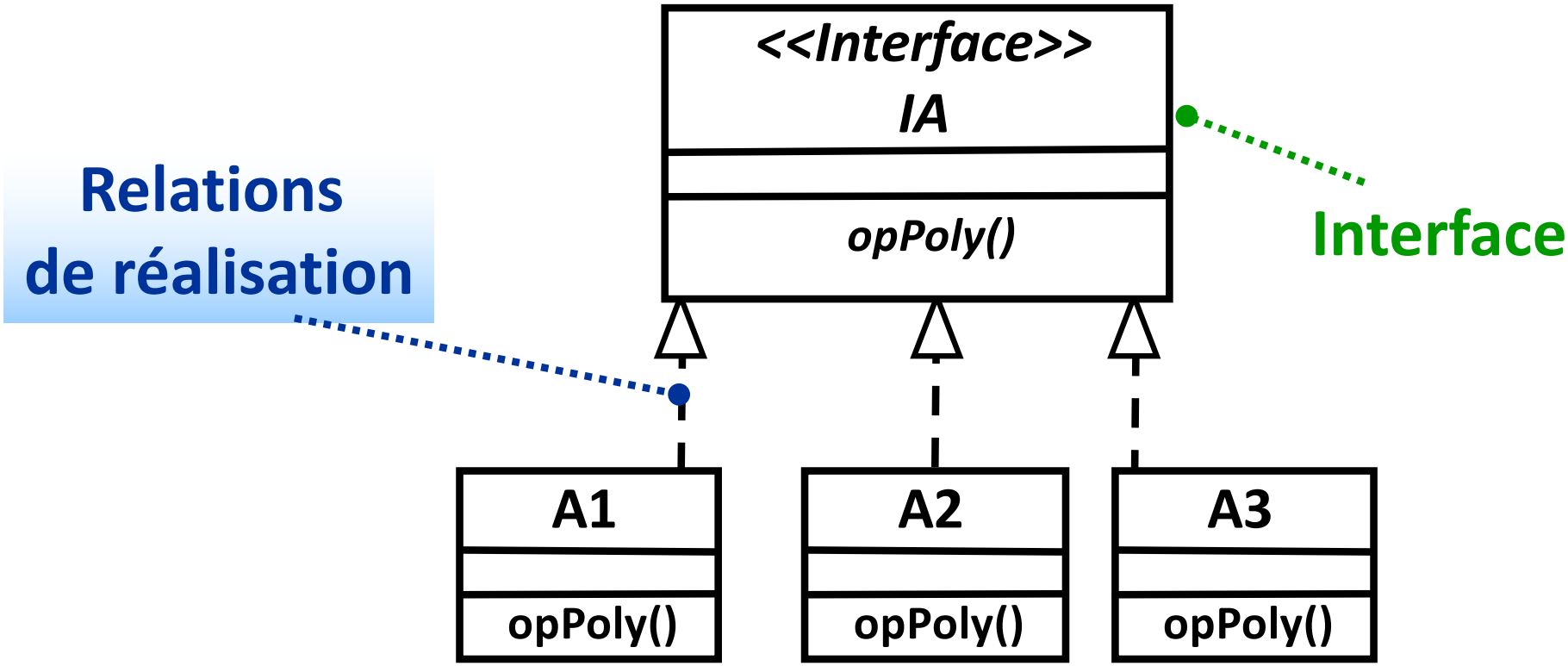
un même profil d'opération
avec plusieurs implémentations
différentes en fonction du type





# Polymorphisme

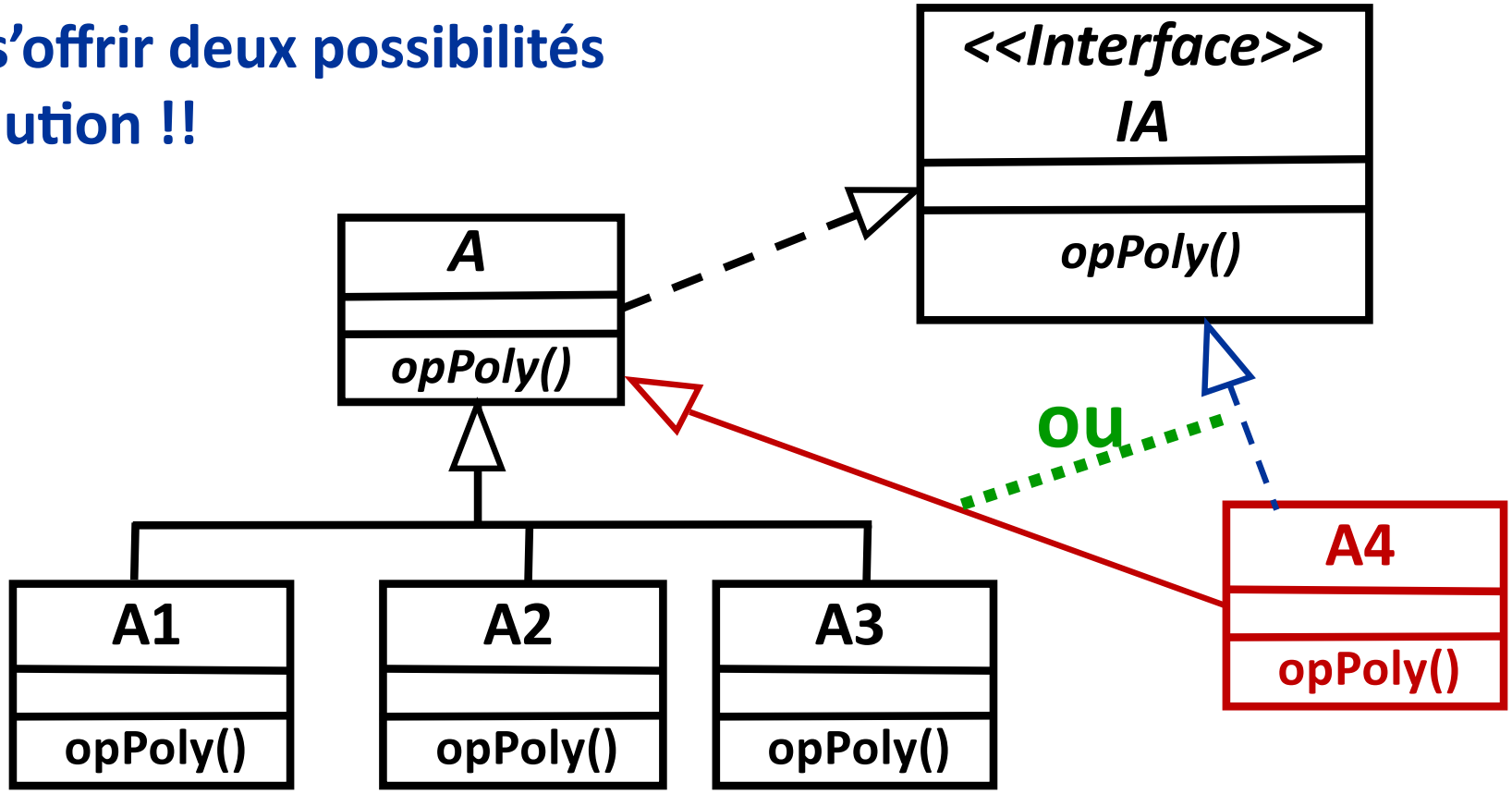
## Solution avec Interface



# Polymorphisme

## Solution mixte Interface/Hiérarchie de Classes

**Pour s'offrir deux possibilités d'évolution !!**



# Polymorphisme

**Exemple**

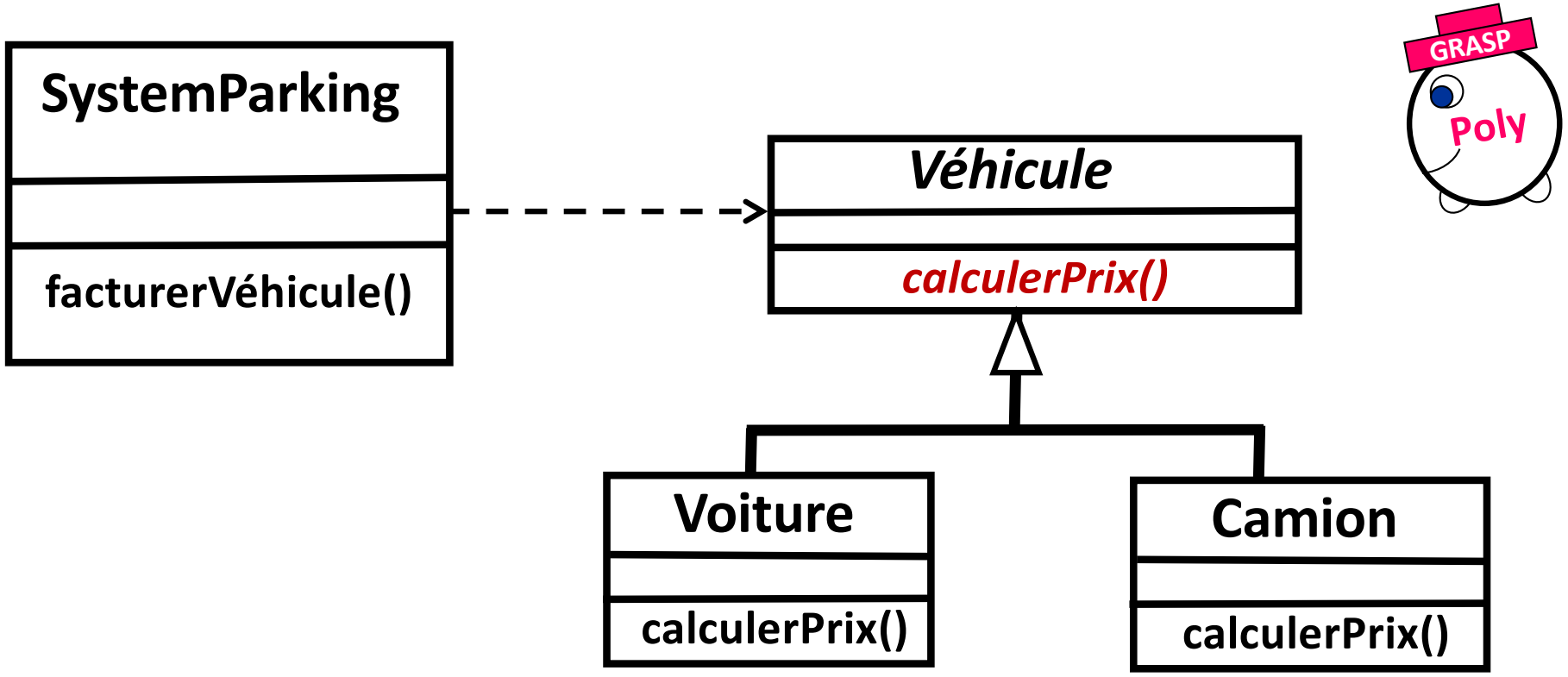
**Le calcul du prix d'une journée de parking dépend de la nature du véhicule qui y stationne:**

- **Camion = le prix est calculé en fonction des dimensions du camion**
- **Voiture = le prix est forfaitaire.**

**Quelles sont les classes responsables de ce calcul?**

# Polymorphisme

**Exemple** **Calcul du prix d'une journée de parking**



# Fabrication Pure

**Nom** **Fabrication Pure**


**Problème**

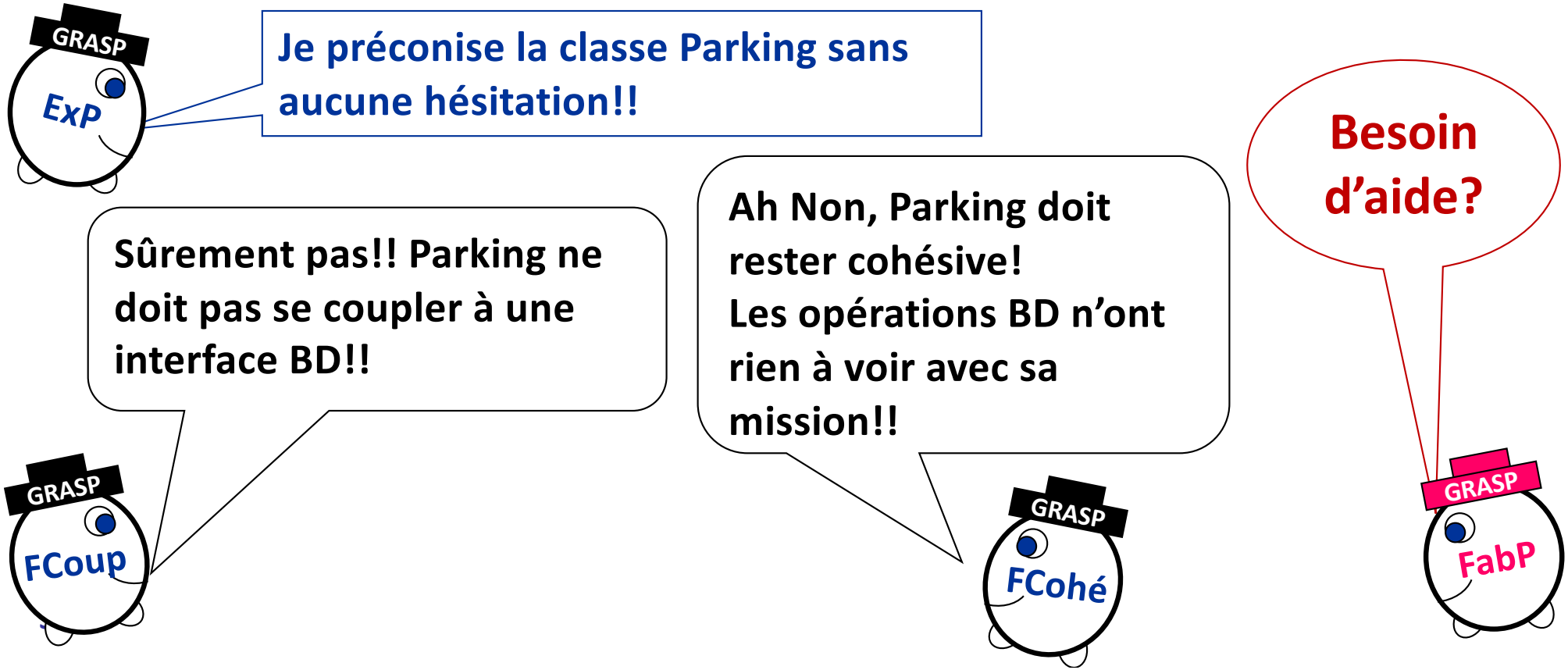
**Trouver une classe responsable lorsque l'on ne veut pas transgresser les principes de FaibleCouplage et de ForteCohésion mais que les solutions offertes par les DP de base ne sont pas appropriées.**

**Solution**

**Créer une classe artificielle pour prendre en compte FaibleCouplage et ForteCohésion.**

# Fabrication Pure

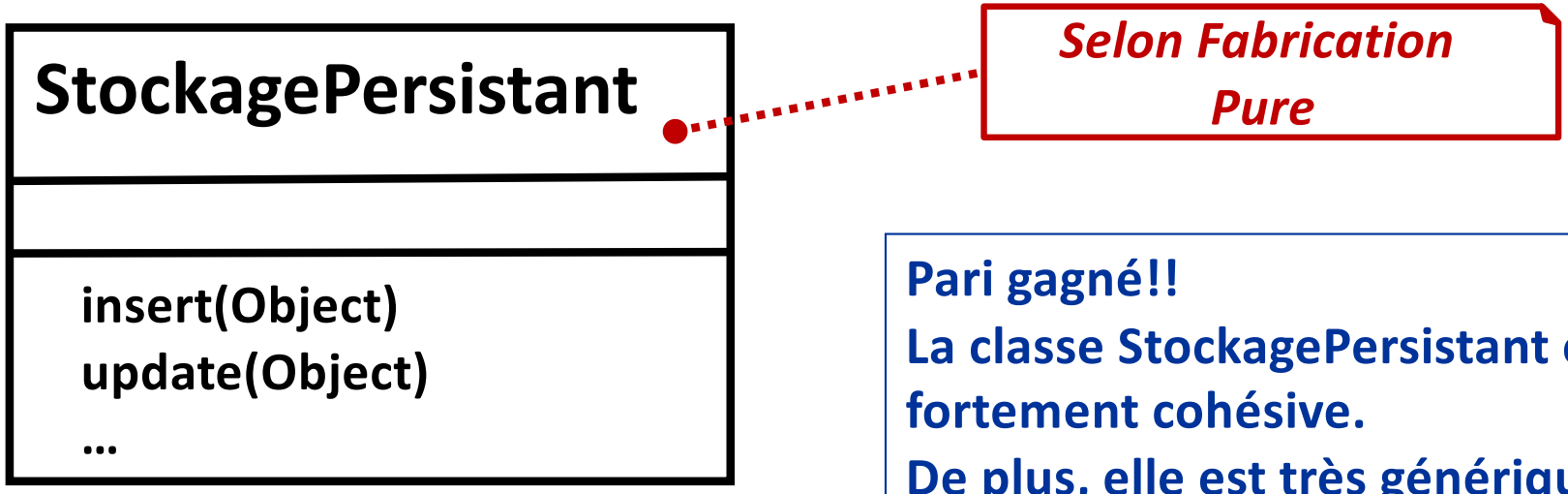
**Exemple** **Sauvegarde des caractéristiques du parking dans une BD relationnelle.**



# Fabrication Pure

**Exemple**

**Sauvegarde des caractéristiques du parking dans une BD relationnelle.**

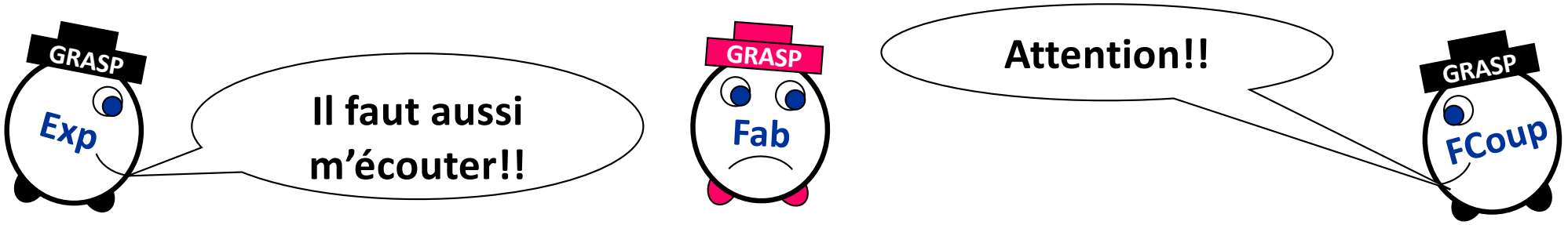


# Fabrication Pure

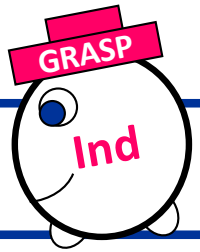
**Certaines classes logicielles sont inspirées directement des classes du domaine par décomposition représentationnelle.**
**D'autres (par ex: fabrication pure) sont « inventées » au cours de la conception par décomposition comportementale.**

## Conseil

**Il ne faut pas abuser de Fabrications Pures!!**



# Indirection



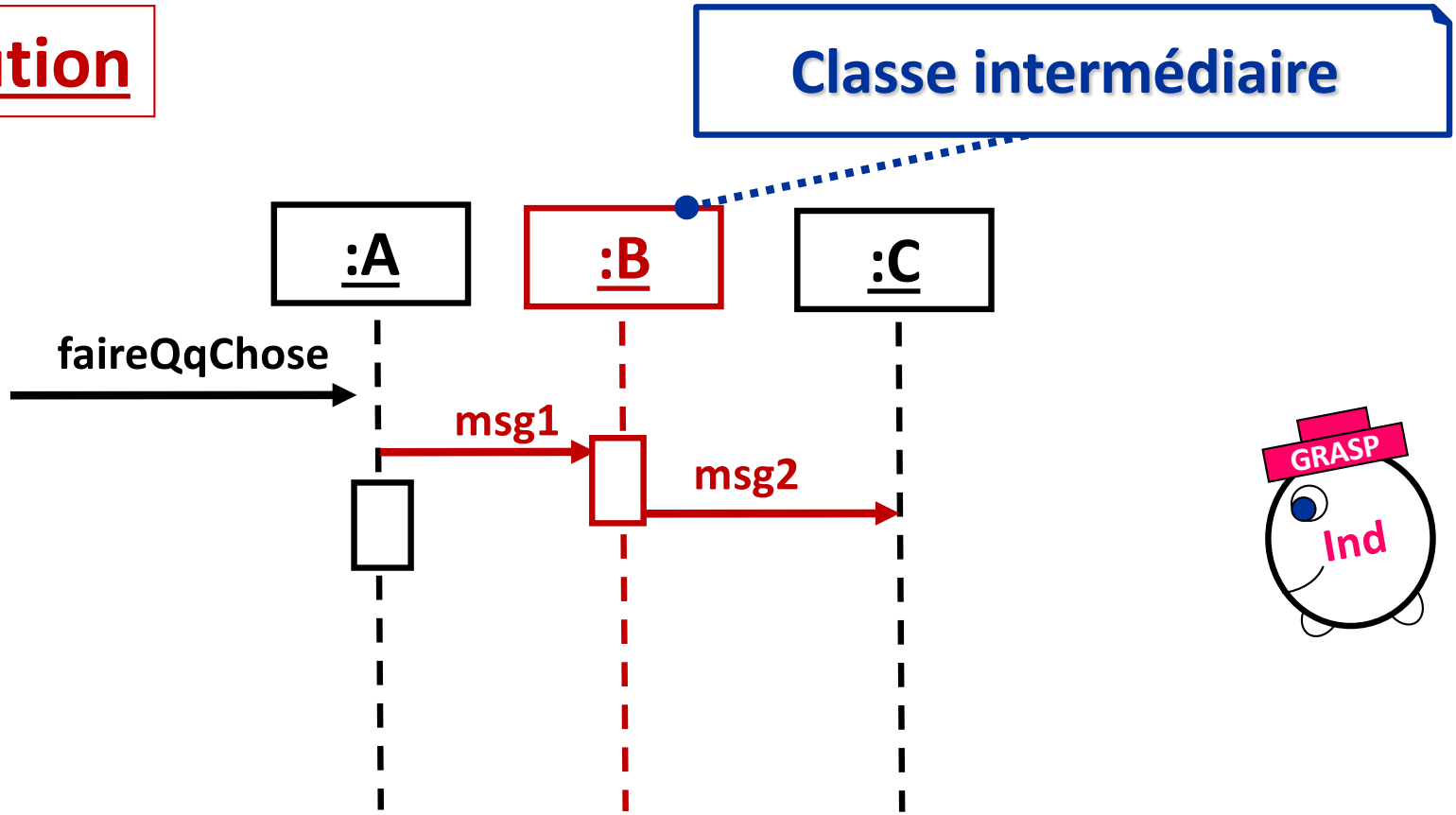

**Nom** — **Indirection**

**Problème** — **Où affecter une responsabilité pour éviter le couplage fort?**
**Comment « découpler » les classes pour maintenir le potentiel de réutilisabilité?**

**Solution** — **Affecter les responsabilités à une classe qui sert d'intermédiaire pour éviter le couplage direct entre composants ou services.**

# Indirection

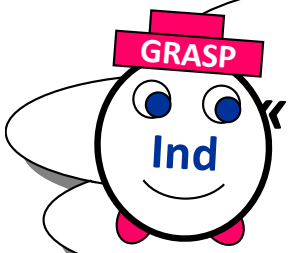
**Solution**



# Indirection

**Exemples**

- **Interfaces masquant différentes API externes**
- **Stockage Persistant : intermédiaire entre Parking et BD.**

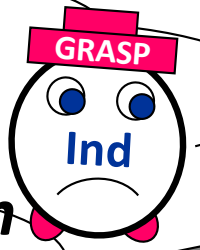



***Vieil Adage Informatique*** **(David Wheeler)**

***« On peut résoudre la plupart des problèmes en ajoutant un niveau d'indirection »***

***Contre- Adage***

***« On peut résoudre la plupart des problèmes de performance en supprimant un niveau d'indirection »***



# Indirection

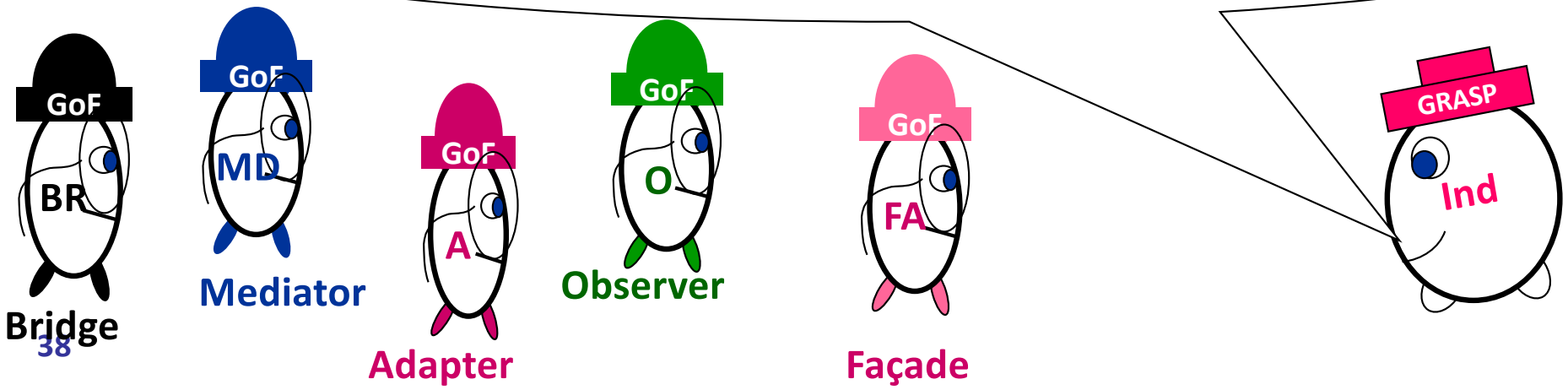
**Patterns connexes**

**Faible couplage (FCoup)**

**Fabrication Pure (Fab)**

**Protection des variations (PV)**

**De nombreux patterns Gof sont des spécialisations de mes idées!!**



# Protection des Variations

**Nom** — **Protection des variations**


**Problème** — **Comment concevoir des éléments de façon à ce que leurs variations n'aient pas d'impacts indésirables sur d'autres éléments?**

**Solution**

1. **Identifier les points de variation ou d'instabilité prévisibles.**
2. **Affecter les responsabilités pour créer une interface (ou classe abstraite) stable autour d'eux.**

# Protection des Variations

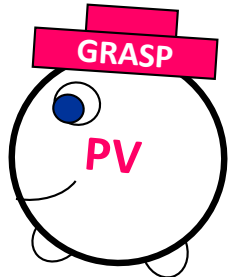
**Autres appelations**

**Masquage des informations**

**Patterns connexes**

**La majorité des patterns (Grasp, GoF et autres) sont des mécanismes de protection des variations.**

**C'est simple, je suis à l'origine de presque tous les patterns. Le Dieu des DP en somme!!**

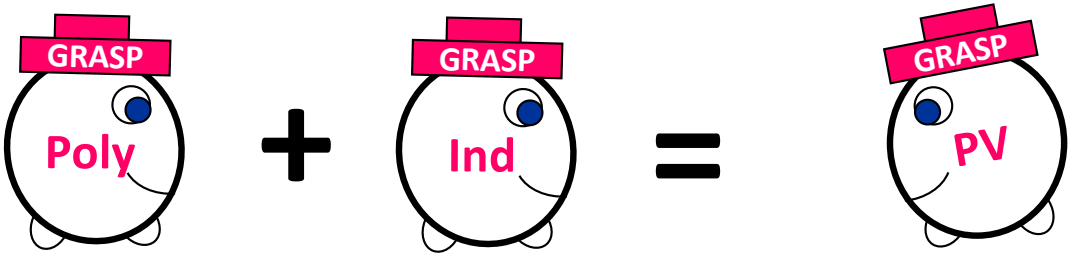


# Protection des Variations

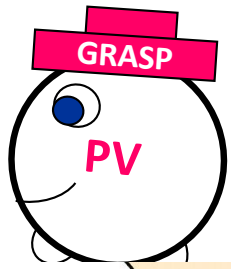
**Exemple**

- **Accès à des API de calculateurs externes : ces API constituent le point d'instabilité ou de variation.**
- **Le système doit pouvoir s'intégrer facilement avec de nouveaux calculateurs externes.**

***Les solutions proposées par les DP Indirection et par Polymorphisme protègent le système contre les variations.***



# Protection des Variations




**Principes voisins**

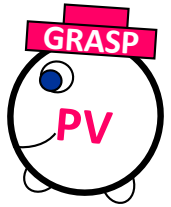
**Loi de Déméter (1988)**

***« Ne pas parler aux inconnus ».***


Dans une méthode, un objet ne peut envoyer de messages qu'aux objets suivants:

- lui-même (objet this)
- objet paramètre de la méthode
- Attribut de this
- Élément d'une collection qui est un attribut de this
- Objet créé à l'intérieur de la méthode

# Protection des Variations

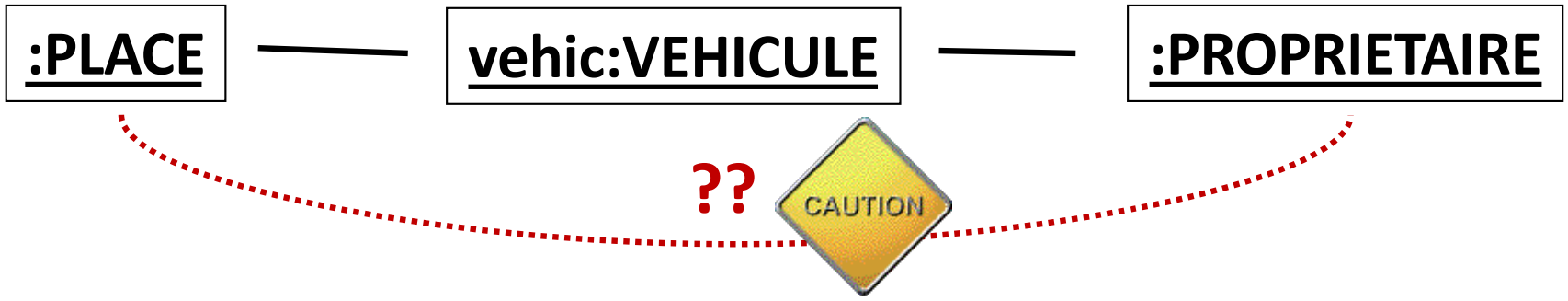



**« Ne pas parler aux inconnus »**

**Éviter les invocations indirectes**

```
Class  PLACE {
   VEHICULE vehic;
   public void methodeUnpeuFragile()
   {
        STRING monProp= vehic.getProp().getNom();
    }
```

**Communication avec un Objet inconnu**



# Protection des Variations

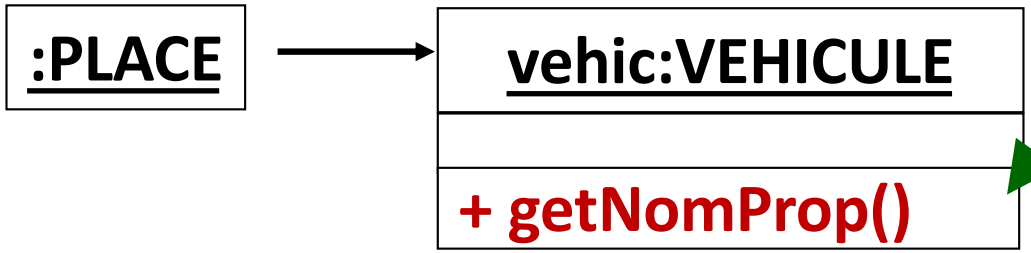

**« Ne pas parler aux inconnus »**

**Éviter les invocations indirectes**

**Solution**

```
Class  PLACE {
   VEHICULE vehic;
   public void methodeUnpeuFragile()
   {
        STRING monProp= vehic.getNomProp().;
    }
```

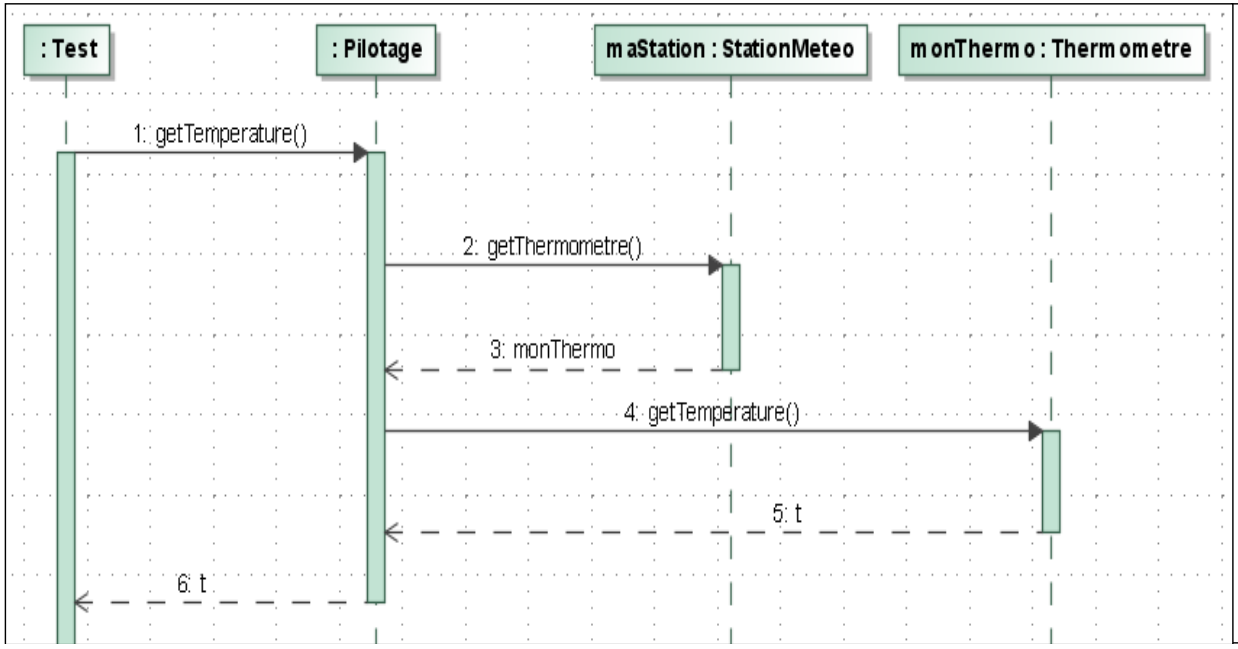
Ajout d'une méthode spécifique

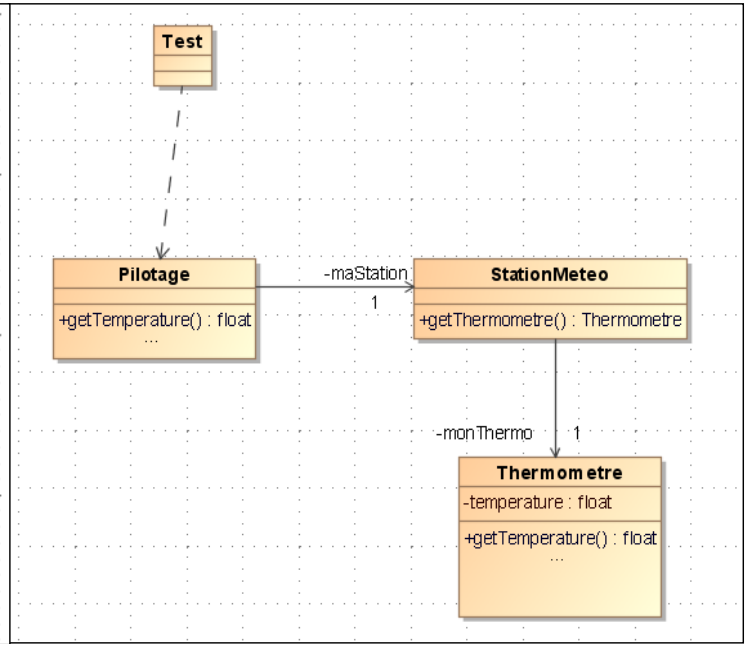


# Exercice= Loi de Demeter

On s'intéresse à la réalisation de la méthode *getTemperature* de la classe Pilotage (cf. diagrammes de classe et de séquence).

1. Définissez le code java de la méthode getTemperature.
2. Expliquez pourquoi cette réalisation ne respecte pas la loi de Demeter et identifiez ses inconvénients.
3. Proposez une solution pour que ce principe soit respecté. Définissez le code java correspondant.
4. Expliquez pourquoi cette solution respecte le principe de faible couplage.



# Exercice= Loi de Demeter (correction)

On s'intéresse à la réalisation de la méthode *getTemperature* de la classe Pilotage (cf. diagrammes de classe et de séquence).

1. Définissez le code java de la méthode getTemperature.

*double getTemperature(){*

*return maStation.getThermometre.getTemperature() ;}*

2. Expliquez pourquoi cette réalisation ne respecte pas la loi de Demeter et identifiez ses inconvénients.

- *L'accés à la méthode getTemperature dans la classe Pilotage constitue un couplage caché. Ainsi si l'entête de cette méthode est modifiée, il faudra modifier la méthode getTemperature de la classe Pilotage.*
- *Cela augmente le couplage car il existe une dépendance entre la classe Pilotage et la classe thermometre et de plus cela pénalise la cohésion car on peut considérer que la classe Pilotage s'occupe de responsabilités qui ne la regarde pas.*

# Exercice= Loi de Demeter (correction)

3. Proposez une solution pour que ce principe soit respecté. Définissez le code java correspondant.

*La solution consiste à ajouter une méthode getTemperature dans la classe StationMéteo. Ainsi, seule cette méthode sera invoquée dans la classe Pilotage.*

4. Expliquez pourquoi cette solution respecte le principe de faible couplage.

*Grace à l'application de ce pattern, la classe Pilotage n'est plus couplée à la classe Thermomètre. Ce pattern contribue ainsi à diminuer le niveau de couplage.*