Design Pattern

En génie Logiciel, un **patron de conception** (**design pattern** en anglais) est un *concept* destiné à résoudre les problèmes récurrents suivant le paradigme objet. En français on utilise aussi le terme motif de conception qui est une mauvaise traduction de «*design pattern*».

Les patrons de conception décrivent des **solutions** standards pour **répondre** à des **problèmes d'architecture** et de **conception** des logiciels.

(source Wikipédia)

Design Pattern

Ce n'est pas un algorithme, mais une **structure générique** qui permet de résoudre le problème référencé. Il est *indépendant* du langage de programmation, il est ”*standardisé*”, suffisamment pour que tous puissent s'y référer. C'est l'expérience qui montre que la *solution est valide.*
On parle parfois de *Best Practices*...
S'y *référer* permet de gagner du temps...
Il y a deux familles de DP, les **GRASP** (DP de base) et les GoF (du **Gang Of Four**). A l'inverse, des **antiPatterns** montrent ce qu'il ne faut pas faire.

les Design Pattern GRASP

GRASP est l'acronyme de *General Responsability Assignment Software Patterns* (schémas généraux d'affectation des responsabilités). ”To GRASP” signifie ”saisir”.

Ce sont des DP présentés par **Craig Larman**, qui sont assez simples, et intuitifs. C'est la **recherche du meilleur responsable** pour une action.

Nos objets collaborent, ont des rôles, sont responsables.

Une **responsabilité n'est pas une méthode**, mais des méthodes s'acquittent de responsabilités.

les Design Pattern GRASP

L'objectif poursuivi par les Design Pattern GRASP de Craig Larman est de programmer objet de façon *claire, méthodique, rationnelle et explicable*.
Les responsabilités :

Faire : faire quelque chose (un calcul, un autre objet), déclencher une action sur un autre objet, contrôler et commander les activités d'un autre objet

Savoir : savoir les valeurs de ses propres attributs, connaitre les objets qui lui sont rattachés, savoir comment obtenir des résultat de ces objets

Les 4 premiers pattern

Expert : Affecter la responsabilité à la classe qui *détient* l'information.

Faible couplage : Affecter les responsabilités de sorte que le *couplage (entre Classe)* reste *faible*.

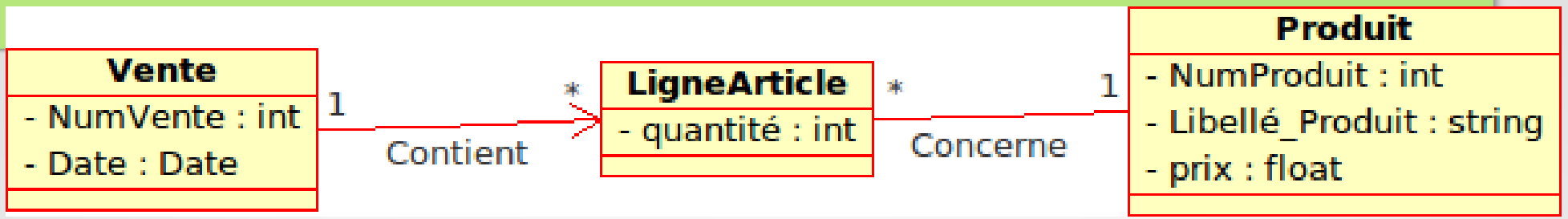
Forte cohésion : Affecter les responsabilités de sorte que la *cohésion (dans la Classe)* demeure *forte*.

Création : La responsabilité de créer une classe incombe à la classe qui *agrège, contient, enregistre, utilise* étroitement ou *dispose* des données d'initialisation de la classe à créer.

Expert en information

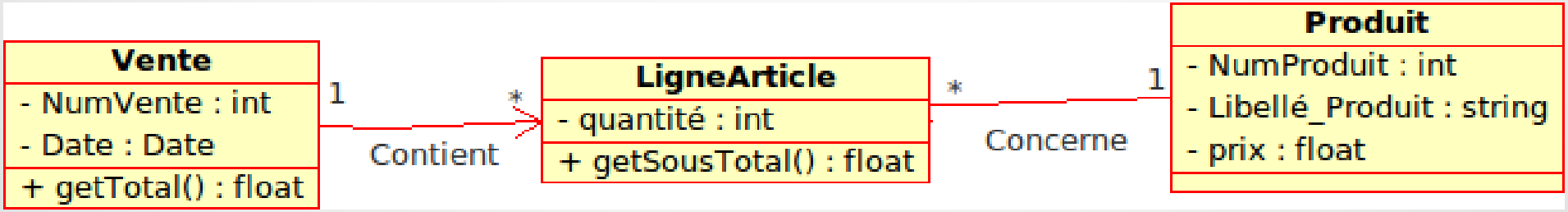
L'objet chargé de la responsabilité d'une action est **celui qui détient l'information concernée** !
Dans l'exemple des vols d'avions, nous avons dit que les compagnies aériennes pouvaient ouvrir ou fermer les réservations pour un vol... Or, qui **sait** si la réservation est ouverte ou fermé, et qui sera chargé de stocker l'information ? Le vol. C'est donc **lui** qui **stocke** l'information, et qui offre à la compagnie la possibilité d'ouvrir ou de fermer la résa.
(Vol dispose donc de deux méthodes : *ouvrirResa*() et *fermerResa*())

Vente : Comment obtenir le total ?



Si on veut obtenir le total général concernant une vente, comment s'y prendre ? Qui est responsable de ce calcul ?

C'est la classe Vente, qui offrira une méthode **getTotal**().. Mais comment cette méthode pourra-t-elle faire le calcul ?



Créateur

Qui **doit créer** un objet A ? Qui en a la **responsabilité** ?
Un objet de la classe B aura la responsabilité de créer des objets de la classe A si :
Il contient ou agrége des objets A
Il possède les informations pour initialiser A
Il utilise étroitement des objets A

la Classe B est une **bonne candidate**, selon le pattern **Créateur**.
Dans l'exemple précédent, **Vente** est une **bonne candidate** pour instancier les **LignesArticles**...

Faible Couplage

Objectif : Réduire l'impact des modifications, limiter les interactions.

Il faut essayer d'affecter les responsabilités **au plus près**. Il faut avoir **accès le plus directement** possible à la bonne méthode, sans passer par trop d'instances. Il y a **fort couplage** si il y a beaucoup d'objets en jeu pour obtenir une information. Ce pattern est lié à **Expert** qui nous pousse à trouver l'objet qui dispose des informations, afin de lui affecter une responsabilité.

Forte cohésion

La **cohésion** est une notion assez vague dont l'objectif est de déterminer **si l'objet n'en fait pas ”trop”.**
Ainsi, un objet avec *100 méthodes, des milliers de lignes de codes, couvrant divers domaines n'est plus cohérent*, il a **perdu sa cohésion**. Il vaut mieux *déléguer*, instituer une *collaboration* avec de nouveaux objets : de plus, un objet trop ”gros” (**faible cohésion**) risque d'introduire un **fort couplage** (car trop d'objets auront besoin de lui, trop d'interactions...)

Les autres pattern GRASP

Les cinq patterns suivants sont plus spécifiques.

Contrôleur : Affecter la responsabilité à une classe « façade » représentant l'ensemble d'un système ou un scénario de cas d'utilisation.

Polymorphisme : Utiliser le polymorphisme pour implémenter les variation de comportement en fonction de la classe.

Fabrication pure : Affecter un ensemble de responsabilités fortement cohésif à une classe artificielle ou de commodité qui ne représente pas un concept du modèle de domaine.

Indirection : Affecter des responsabilités à un objet qui sert d'intermédiaire entre d'autres composants ou services pour éviter de les coupler directement.

Protection des variations : Identifier les points de variations ou d'instabilité prévisibles. Affecter les responsabilités pour créer une interface stable autour d'eux.

Contrôleur

Comment **représenter** le programme principal, le système lui-même ?

Il faut gérer l'**ensemble des actions**, **coordonner** les différents éléments. Il faut une classe **Contrôleur**, qui *déléguera* ensuite aux divers objets spécialisés, et *récupérera* les résultats de leurs actions.

Il ne s'agit pas de l'interface homme machine. Le pattern Contrôleur consiste en l'affectation de la responsabilité de la réception et/ou du traitement d'un message système à une classe.

Il *reçoit les demandes*, et le *redirige vers la bonne classe*, celle qui en a la **responsabilité**.

Contrôleur

Une classe Contrôleur doit être créée si elle répond à l'un des cas suivants :

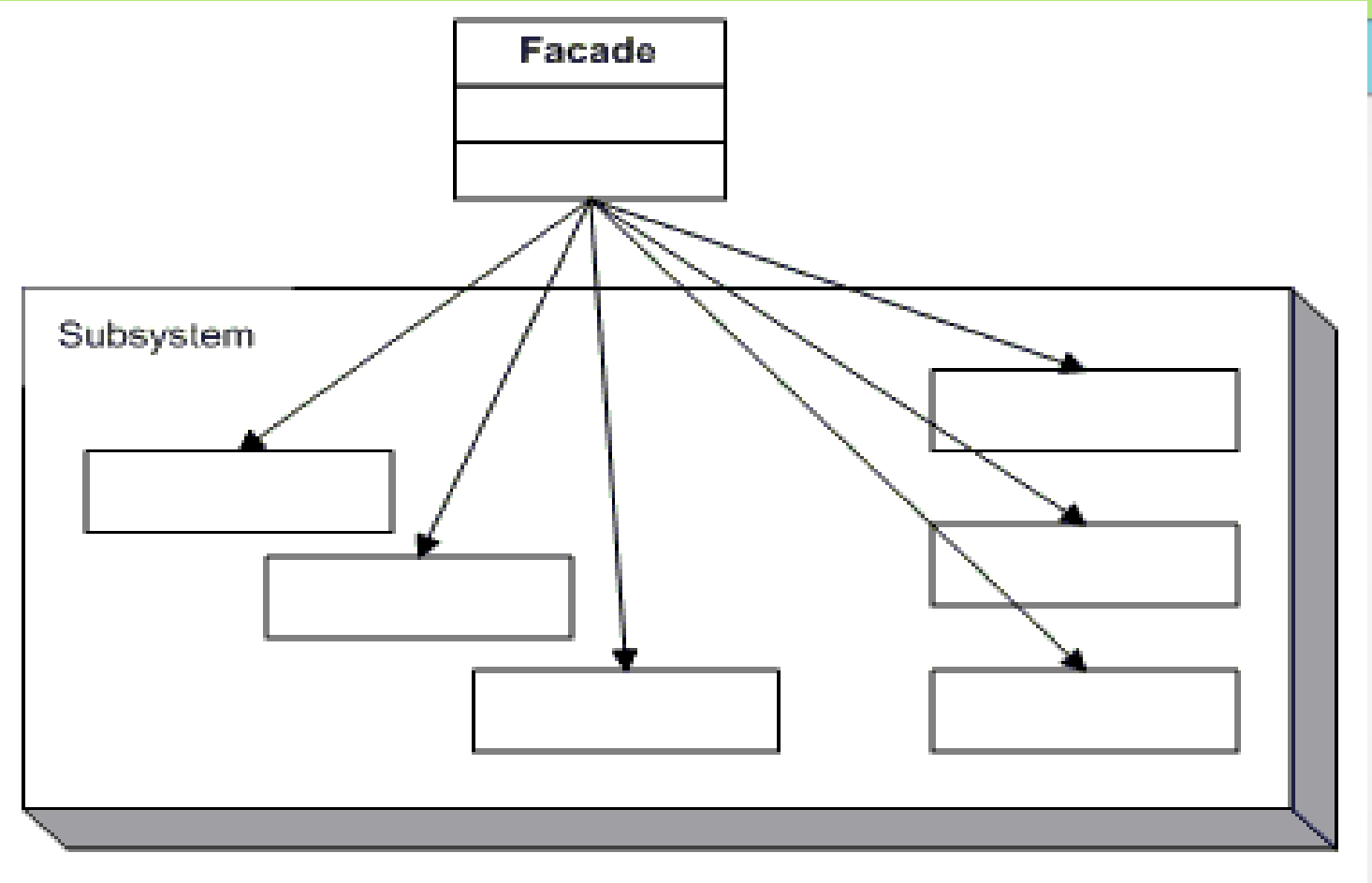
La classe **représente un contrôleur de façade,** c'est à dire l'interface d'accès à l'ensemble d'un système (pas graphique, mais de gestion).

La classe **représente le scénario** issu d'un cas d'utilisation. On le nomme en général « Session », « gestionnaire » ou « coordonnateur ». Elle est chargée de **traiter tous les événements** systèmes contenus dans un scénario de cas d'utilisation.

Contrôleur et 'Façade'

Parfois, si il y a **beaucoup de cas d'utilisation**, ou **beaucoup d'actions** à coordonner, pour éviter de perdre le Pattern '**cohésion forte**', on peut utiliser une classe **Façade**, qui **utilisera des Contrôleurs** qui géreront ensuite des cas spécialisés.
Ainsi, sur une grosse application, si l'ensemble des actions possibles devenait trop important, on pourrait imaginer une **façade générale** qui utiliserait les **contrôleurs** '*GestionClients*', '*GestionCommandes*' et '*GestionProduits*'.

Un contrôleur Façades et ses controleurs.



Le **Contrôleur** (ici, la version Façade, qui appelle d'autres sous contrôleurs...)

Polymorphisme

Le **polymorphisme** est le concept idéal pour réaliser une **variation du comportement** des objets en fonction de leur type. Il permet également de réaliser des **composants logiciels enfichables** grâce aux patterns **adaptateurs**. Il est utile dans le cadre de prévisions de remplacement ou d'ajout de composants.

Les programmes utilisant les instructions *if elseif* ou *switch case* nécessitent une intervention au niveau du client lorsqu'on souhaite ajouter un cas. Cela rend le programme plus difficile à maintenir et le module difficile à réutiliser.

Fabrication Pure

La **fabrication pure** est une réalisation qui ne ne se **déduit pas** du domaine. Elle est totalement **artificielle**. Ce pattern est utilisé lorsque les patterns ne parviennent pas à assurer les principes de **forte cohésion** et de **faible couplage**.
Le concepteur doit **inventer** de nouveaux concepts abstraits sans relation directe avec les domaine. Ils sont plutôt **génériques**.. Par exemple, une classe chargée de la **persistance**. Ce n'est pas un objet métier, c'est un choix du concepteur pour résoudre SA problèmatique de stockage.

Indirection

Lorsque certaines classes ont **trop de responsabilités**, la cohésion tend à baisser (pattern **forte cohésion**) et la classe **devient plus difficile** à comprendre et à maintenir. Une solution à ce type de problème est de **déléguer** une partie des traitements et des données à d'autres classes en ajoutant un niveau **d'indirection**.

Il convient néanmoins de rester attentif aux **baisses de performances** liées à un trop grand nombre de niveaux d'indirection.

Un exemple d'indirection est le découplage des services *métier* et de la *persistance*.

Indirection...

L'un des problèmes évoqués pour **Fabrication Pure** est la nécessité de **découpler** l'objet de sa persistance... On délégue à une autre classe la responsabilité du dialogue avec la base de données. On a ici une **Indirection**, c'est à dire qu'on a crée une **Classe intermédiaire** afin d'éviter de coupler directement l'objet de la BD.

« *En informatique, on peut résoudre la plupart des problèmes en ajoutant un Objet niveau d'indirection* ».

Par contre, on dit aussi ”*En informatique, on peut accélerer la réactivité d'un logiciel en retirant des niveaux d'indirection*”

Protection des Variations

L'application sera sujette à **évolutions**. Il faut la protéger des conséquences de ces modifications prévisibles.

Le principe consiste à utiliser les mécanismes **d'abstraction** pour créer une interface **stable** autour des objet du système.

Il faut empécher qu'une **variation** d'un sous-système **n'impacte** trop son environnement. On retrouve l'idée qui nous guide depuis le début : *comment permettre l'évolution sans compromettre l'ensemble du logiciel ?*

L'accès aux attributs via des méthodes participe de cette démarche.

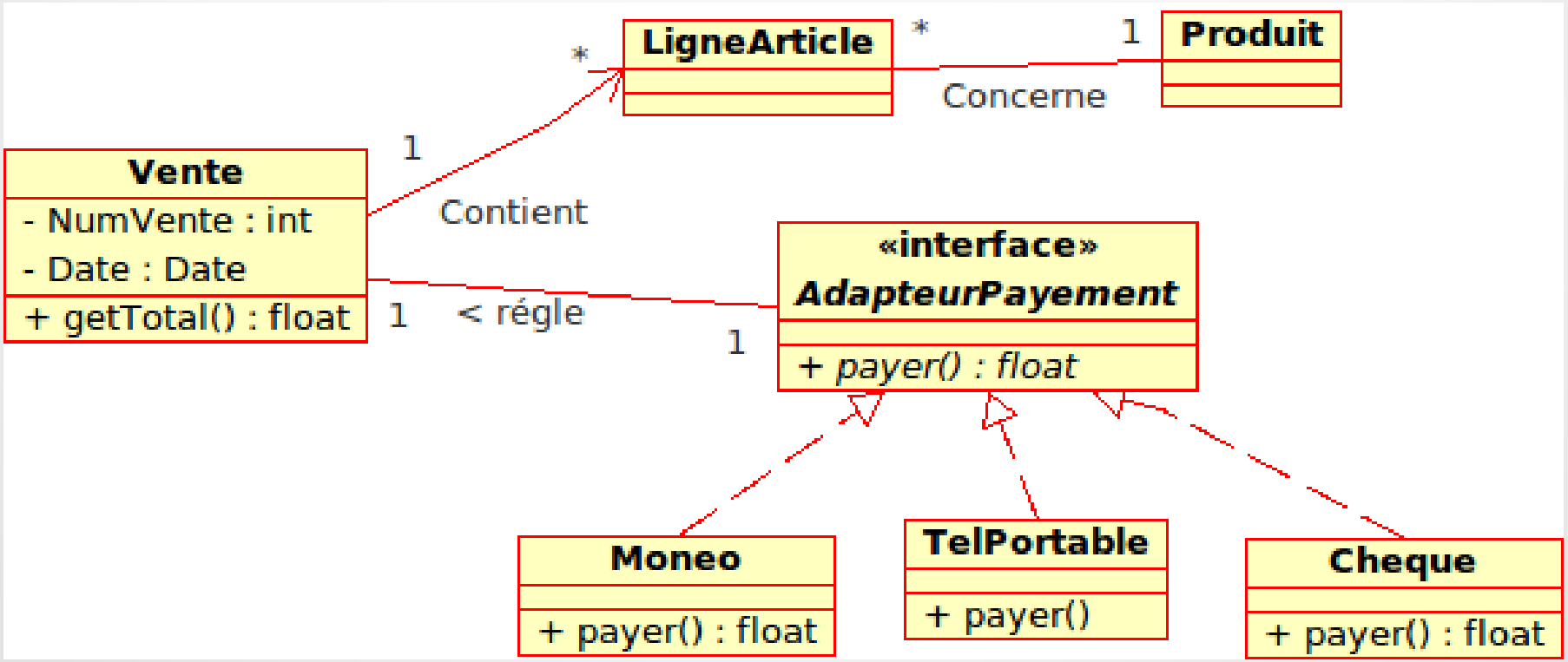
Exemple de PV

En reprenant l'exemple de notre vente, imaginons maintenant que nous voulons **gérer le paiement :** Il existe de nombreux **modes de paiement**, et de nouveaux apparaissent : *monéo*, *paiement par téléphone portable*...

Comment concevoir une solution logiciel pérènne ? La solution peut être de prévoir une *indirection* et une *interface* : L'interface *impose* des *méthodes* (PV), et on a une classe qui dialoguera avec les futurs nouveaux paiements implémentant cette interface (**Indirection**)...

Exemple de PV...

Protégons nous des différentes façons de régler une Vente...



Problème de PV

Le principal problème est de se protéger à bon escient des **Variations**.
Ainsi, souvent, au début, **la protection est insuffisante.**
Des évolutions de la solution logicielle entraînent de fortes modifications de structures. Ensuite, le programmeur trop méticuleux *risque de trop anticiper* les Variations. Et introduire trop *d'Indirections* dans son programme.
L'équilibre n'est pas aisé à trouver..

Exemple de PV

L'un des atouts les plus spéctaculaires de la POO est l'appel des méthodes sur les résultats de méthodes. Avec les IDEs, on peut rapidement prendre du plaisir à écrire **instance.méthodeA().methodeB().methodeC()**
(rappelons qu'ici la méthodeC est appelée sur l'objet renvoyé par la méthodeB qui s'appliquait à l'objet renvoyé par la methodeA demandé à l'instance)
Cependant, c'est MAL !!
Enfin, disons plutôt que cela **couple** trop tous ces objets. Si jamais la méthodeB est modifiée, il faudra penser à intervenir ici ! (Merci Eclipse)

Ne parlez pas aux inconnus !!

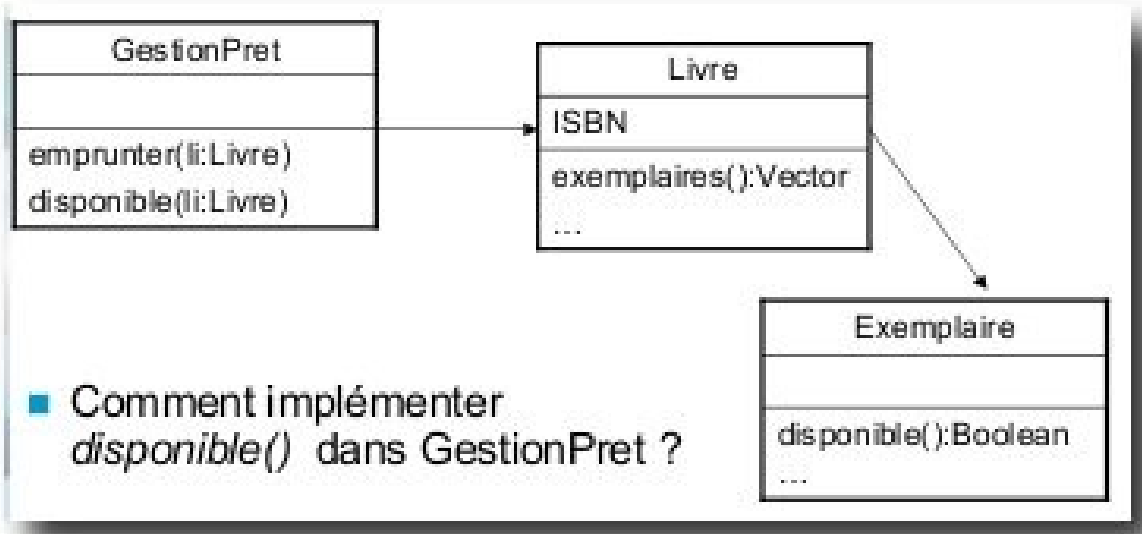
C'est le nom de cette démarche.. **instance.méthodeA().methodeB().methodeC()** : *On parle à des inconnus*...

Pour se protéger d'eux et de leurs hypothétiques variations, il faudrait :

écrire une méthodeA' pour instance, qui elle invoquerait une méthodeB' qui récupérerait le résultat de méthodeC.

Ainsi, une variation de **methodeB** ici n'impacterait que **methodeA'** (et pas notre code ici)...

Exemple de ”ne parlez pas...”



- Comment implémenter *disponible()* dans GestionPret ?

Un exercice...

... et un exemple

Sans le Principe

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

Ici, nous obtenons l'objet thermomètre de la station, puis nous appelons la méthode getTemperature() nous-mêmes.

Avec le Principe

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

Quand nous appliquons le principe, nous ajoutons à la classe Station une méthode qui demande le thermomètre à notre place. Cette technique réduit le nombre de classes dont nous dépendons.