GRASP : conception objet et responsabilités

extrait de UML2 et les Design Patterns, Craig Larman

Mireille Blay-Fornarino, Université Nice Sophia Antipolis, Département Info Iut, Septembre 2018

Bibliographie

- Craig Larman, UML2 et les Design Patterns
- Glenn D. Blank CSE432, Lehigh University
- Sylvain Cherrier, Design Patterns
- Laurent Henocque, Design Patterns

GRASP

General Responsability Assignment Software Patterns*

*Patrons logiciels d'affectation de responsabilités

Patterns ?

"A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
Christopher Alexander, professor of architecture

- En génie Logiciel, un **patron de conception** (**design pattern** en anglais) est un *concept* destiné à résoudre les **problèmes récurrents** suivant le paradigme objet. Les patrons de conception décrivent des **solutions** standards pour **répondre** à des **problèmes** d'architecture et de conception des logiciels (source Wikipédia)

© 2008 WhiteHawk Press

GRASP : General Responsability Assignment Software Patterns

- **Une approche méthodique de la COO**

- Patterns généraux d'affectation des responsabilités pour aider à la conception orientée-objet

  - raisonner objet de façon méthodique, rationnelle, explicable

  - Patterns fondamentaux, simples, basiques...

- Utiles pour l'analyse et la conception

- **Référence : Larman 2004**

Conception dirigée par les responsabilités

Responsabilité des objets (Anthropomorphism): Object-oriented programming works like human organizations. Each object will communicate with another one by sending messages. So the software objects work by just sending those messages.

Ron McFadyen, 2004

- **Métaphore**
  - communauté d'objets responsables qui collaborent (cf. humains) : Anthropomorphism.
  - penser l'organisation des composants (logiciels ou autres) en termes de responsabilités par rapport à des rôles, au sein de collaborations.
- **Responsabilité**
  - abstraction de comportement (contrat, obligation par rapport à un rôle)
  - une responsabilité n'est pas une méthode;
  - les méthodes s'acquittent des responsabilités.

GRASP : Un principe général

- Toujours chercher à réduire le décalage des représentations entre
  - la façon de penser le domaine (humaine)
    - « un échiquier a des cases »
  - les objets logiciels correspondants
    - un objet Echiquier contient des objets Case
    - vs. un objet Echiquier contient 4 objets 16 Cases
    - vs. un objet TYR43 contient des EE25

Conception dirigée par les responsabilités

- Les responsabilités de **Faire** d'un objet peuvent être :
  - faire quelque chose (un calcul, créer un autre objet),
  - déclencher une action sur un autre objet,
  - contrôler et coordonner les activités d'un autre objet
- Les responsabilités de **Savoir** d'un objet peuvent être :
  - connaître les valeurs de ses propres attributs (données privées encapsulées),
  - connaitre les objets qui lui sont rattachés,
  - connaître les éléments qu'il peut calculer ou dériver.

Des Patterns GRASP

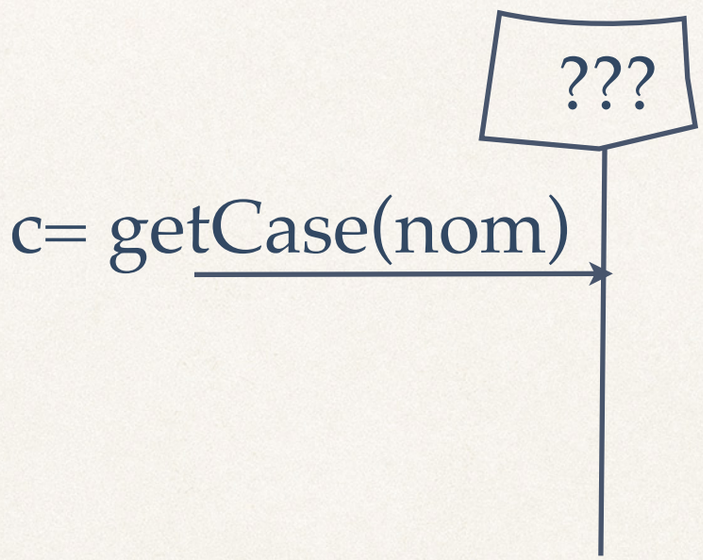
- Information Expert
- Creator
- Low Coupling
- Controller
- High Cohesion
- Polymorphisme
- Fabrication Pure

Expert en Information

- **Problème**
  - Quel est le principe général d'affectation des responsabilités aux objets ?
- **Solution**
  - Affecter la responsabilité à l'expert en information i.e. la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité

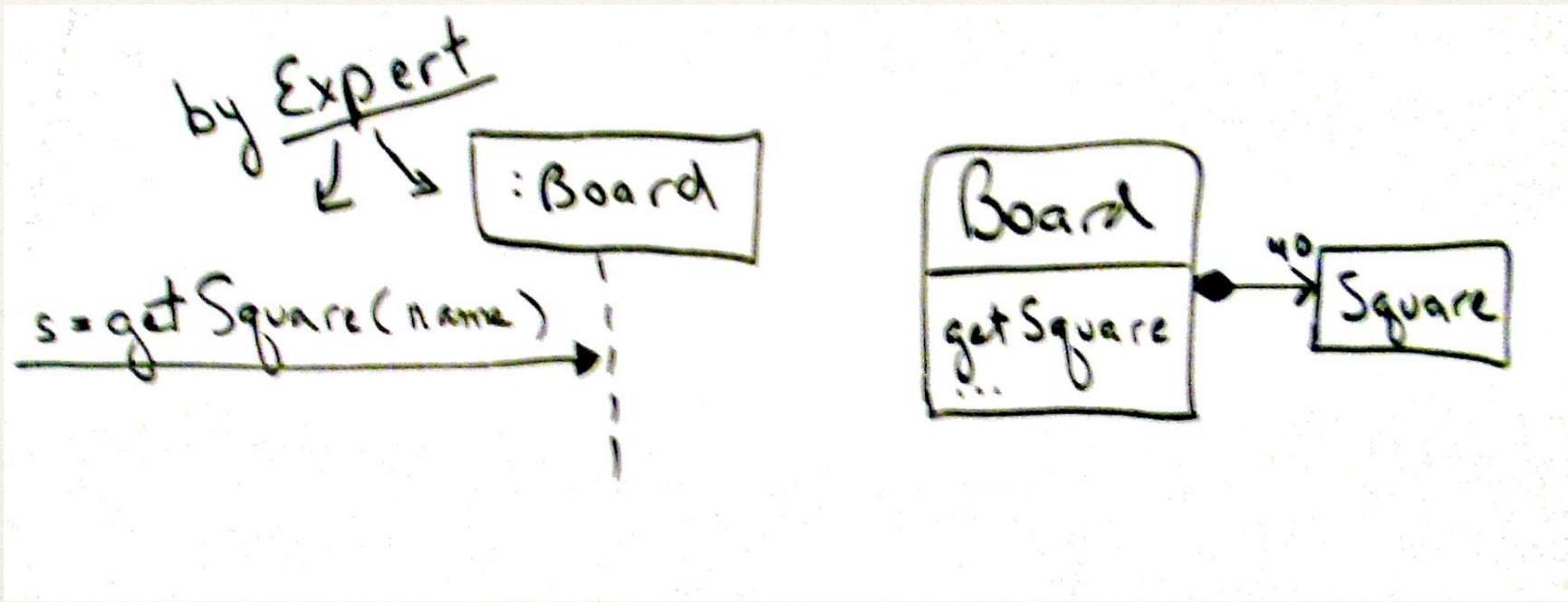
Expert (GRASP) : Exemple

- Monopoly : qui connaît un objet Case étant donné une clé ? Qui est responsable de retrouver une case à partir de son nom ?



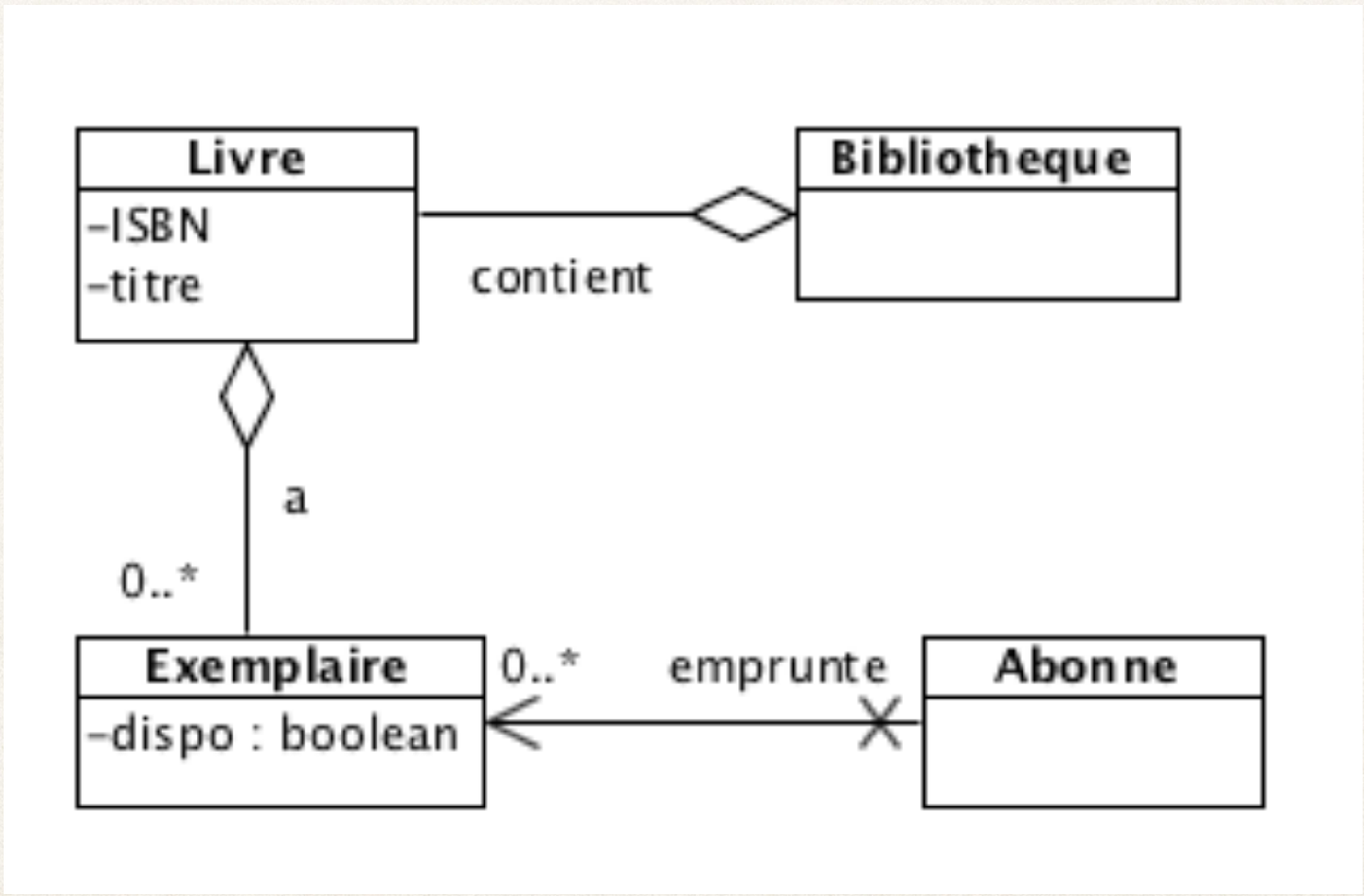
Expert (GRASP) : Exemple

- Monopoly :
  - Qui connaît un objet Case étant donné une clé ?
  - Qui est responsable de retrouver une case à partir de son nom ?



Expert (GRASP) : Exemple

- Bibliothèque : qui doit avoir la responsabilité de connaître le nombre d'exemplaires disponibles d'un livre ?

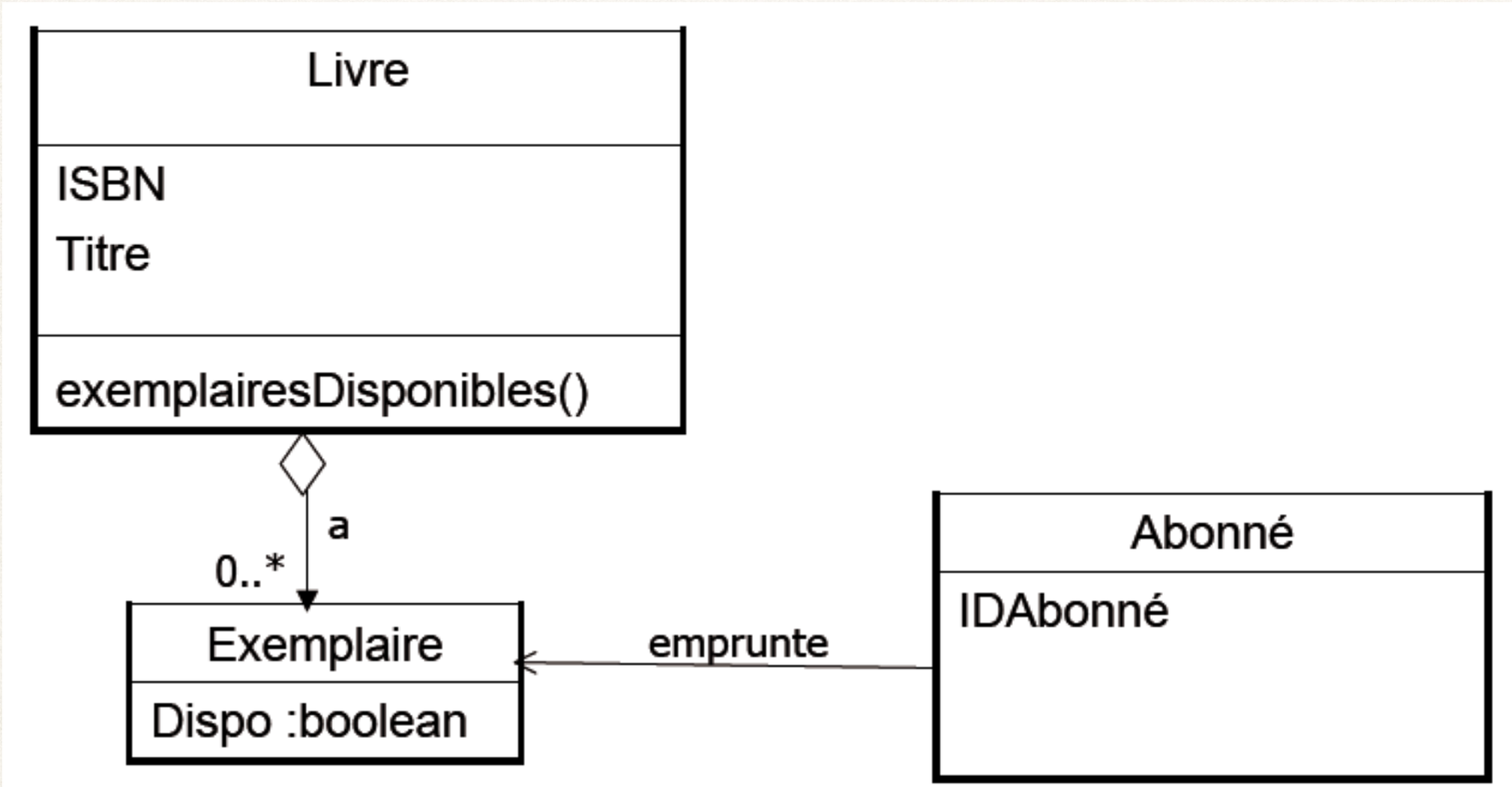


- d'après Sylvain Cherrier, Design Patterns

Expert (GRASP) : Exemple

- Commencer avec la question
  - De quelle information a-t-on besoin pour déterminer le nombre d'exemplaires disponibles ?
    - *Disponibilité de toutes les instances d'exemplaires*
- Puis
  - Qui en est responsable ?
    - Livre est l'Expert pour cette information

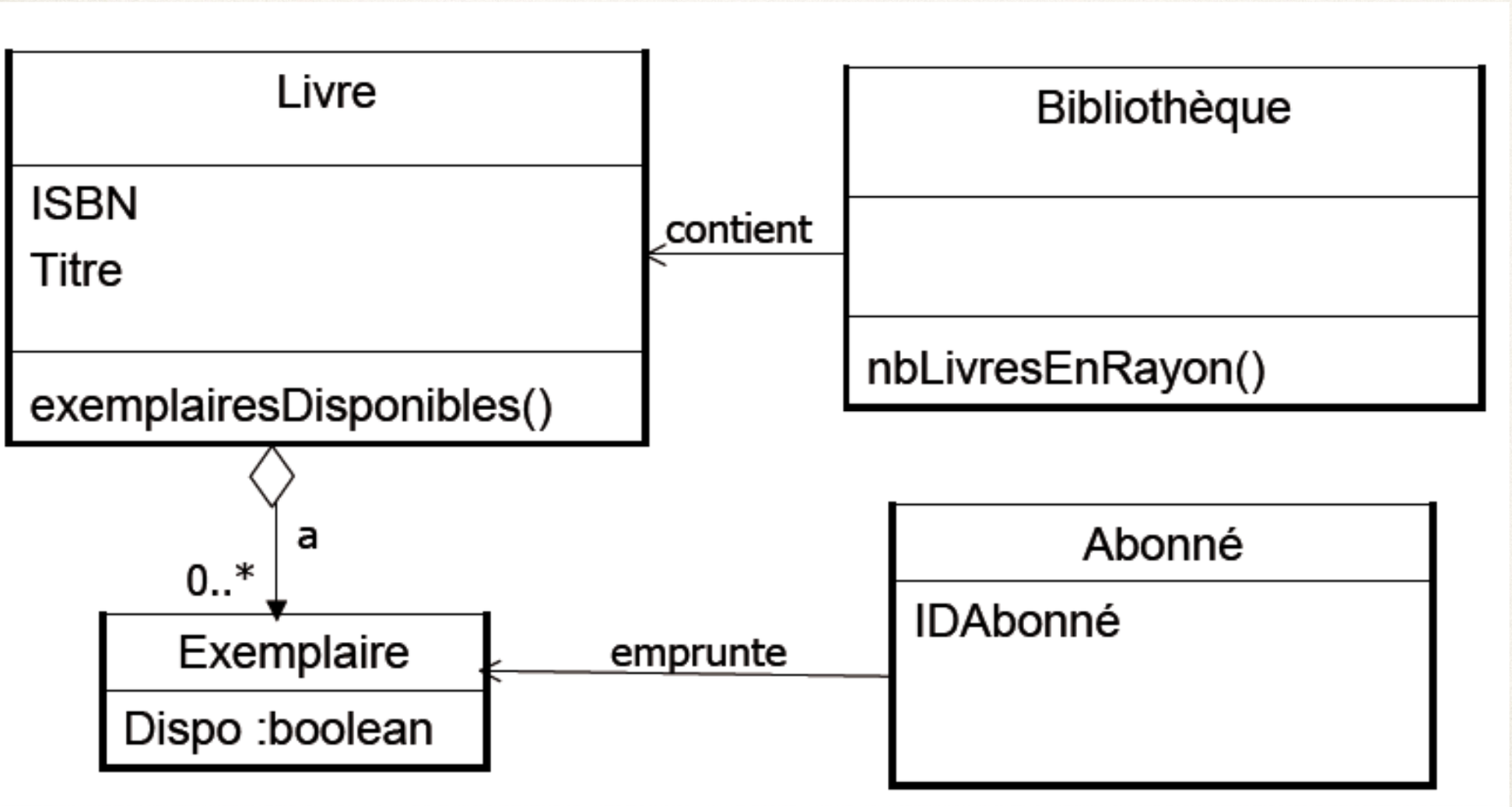
Expert (GRASP) : Exemple



Qui est responsable de connaitre le nombre de livres en rayon?

- Sylvain Cherrier, Design Patterns

Expert (GRASP) : Exemple



- Sylvain Cherrier, Design Patterns

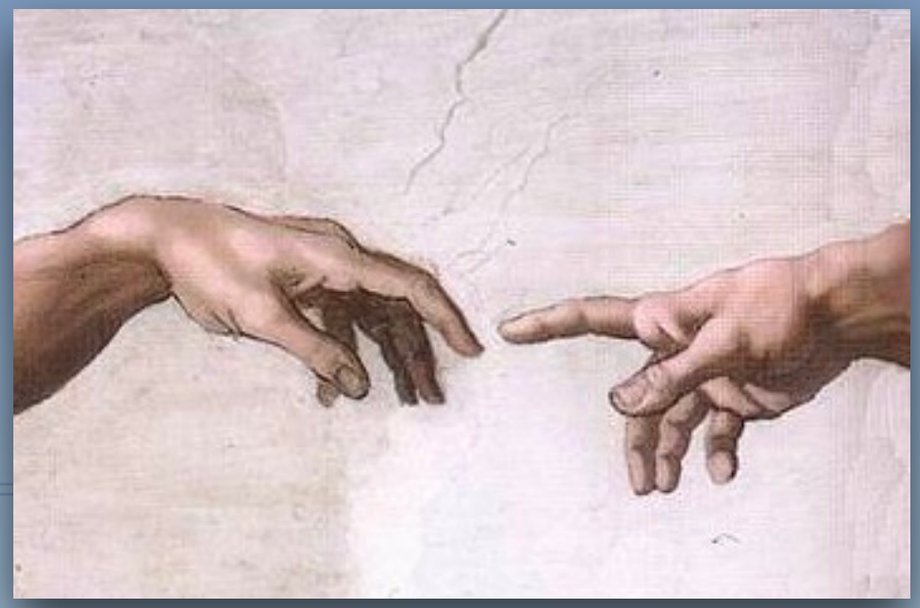
Expert (GRASP) : Discussion

- Le plus utilisé de tous les patterns d'attribution de responsabilité
- Un principe de base en OO
- L'accomplissement d'une responsabilité nécessite souvent que l'information nécessaire soit répartie entre différents objets.

Expert (GRASP) : Discussions

- Facilite l'encapsulation et la cohésion
  - les objets utilisent leur propre information pour mener à bien leurs tâches
- Le comportement est distribué à travers différentes classes qui ont l'information nécessaire
  - encourage des définitions de classes plus légères, plus cohésives, plus faciles à comprendre et à maintenir
- Autres noms
  - Mettre les responsabilités avec les données
  - Qui sait, fait
  - Faire soi-même

Créateur



- **Problème**
  - Qui doit avoir la responsabilité de créer une nouvelle instance d'une classe donnée ?

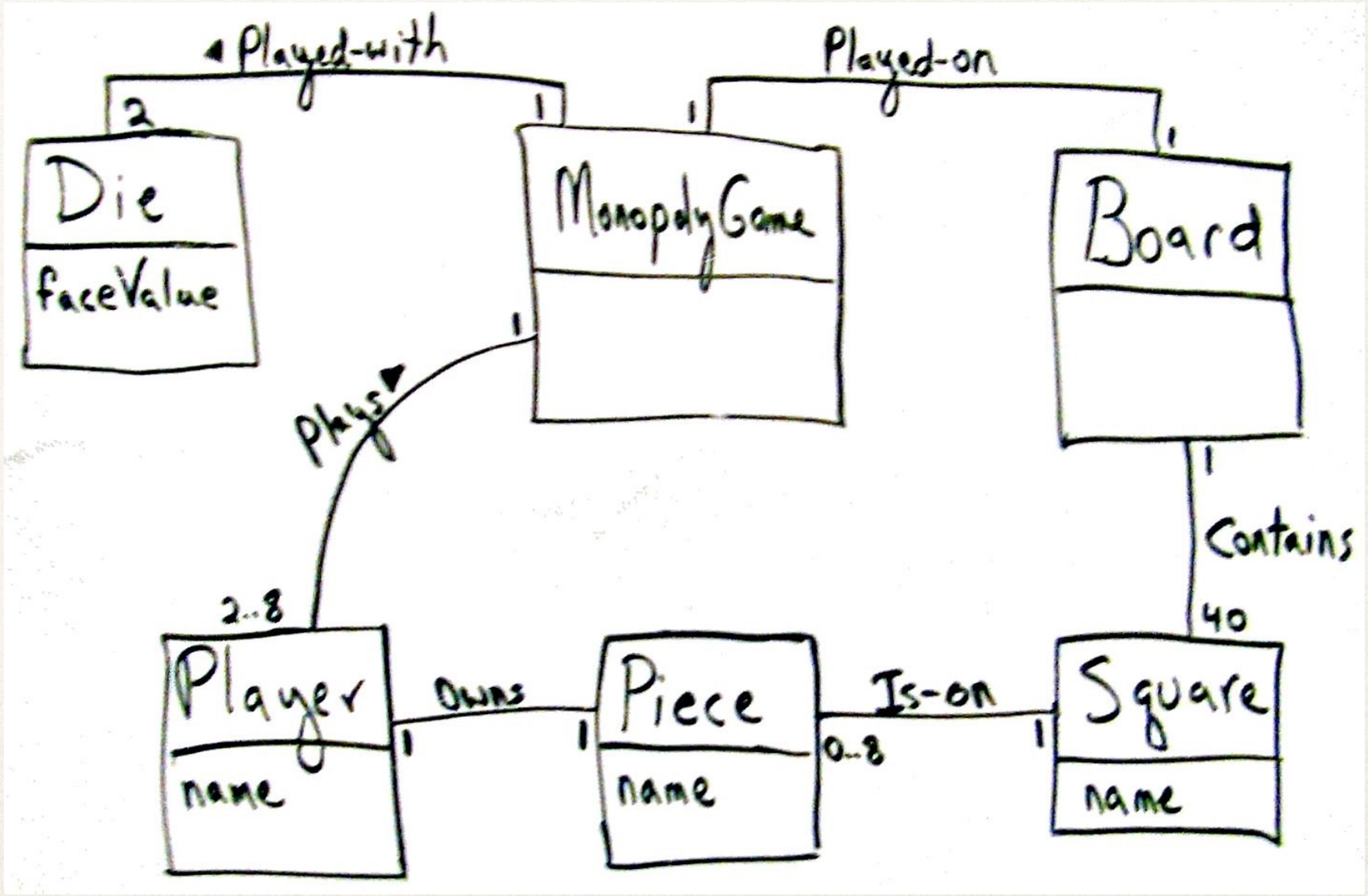
- **Solution**

  Affecter à la **classe B la responsabilité de créer une instance de la classe** si une - ou plusieurs - des conditions est vraie :
  - B contient ou agrège des objets A
  - B enregistre des objets A
  - B utilise étroitement des objets A
  - B a les données d'initialisation qui seront transmises aux objets A lors de leur création

  « B est un Expert en ce qui concerne la création de A»

Créateur (GRASP) : exemple

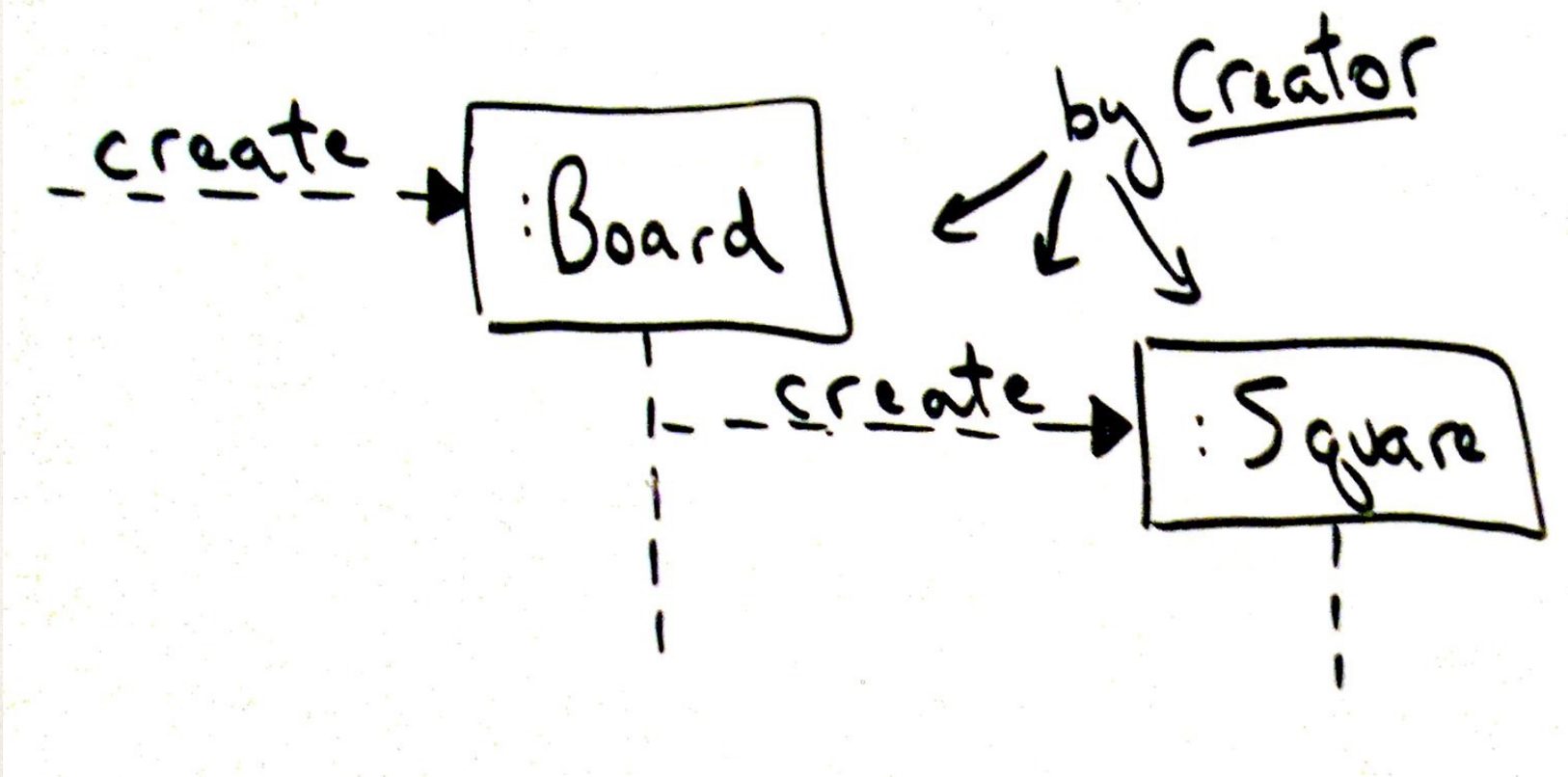
- Monopoly : qui doit être responsable de la création des cases du jeu?
- Intuitivement ?



Créateur (GRASP) : exemple

- Monopoly : qui doit être responsable de la création des cases du jeu?

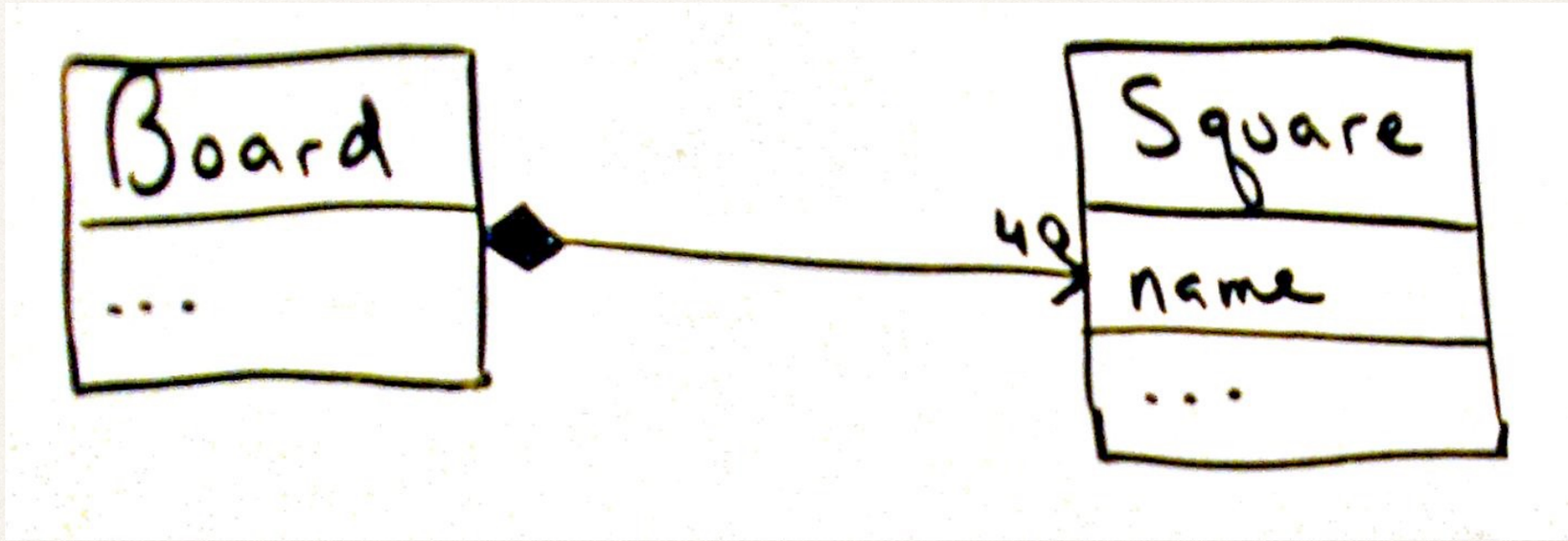
- On conforte notre intuition par un diagramme de séquence.



Créateur (GRASP) : exemple

- Monopoly : qui doit être responsable de la création de la case du jeu?

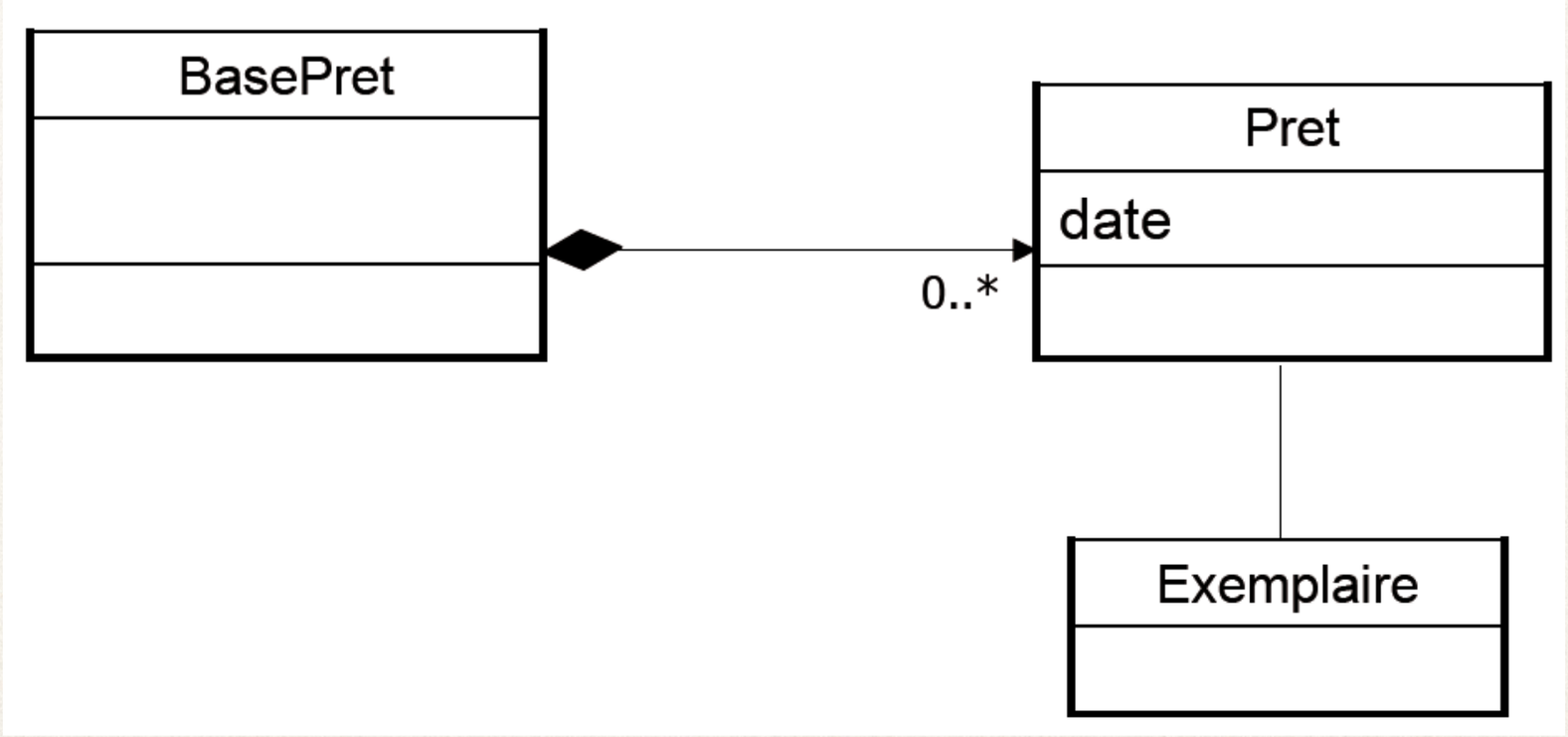
- Cela nous amène au diagramme en conception.



Créateur (GRASP) : exemple

- Bibliothèque : qui doit être responsable de la création de Pret ?

- BasePret contient des Prêts : elle doit les créer.



Créateur (GRASP) : discussion

- Guide pour attribuer une responsabilité pour la création d'objet
  - une tâche très commune en OO
- Finalité : trouver un créateur pour qui il est nécessaire d'être connecté aux objets créés
  - favorise le *Faible couplage*
    - Moins de dépendances de maintenance, plus d'opportunités de réutilisation
- Pattern liés
  - Faible couplage
  - Composite
  - Fabricant

- Sylvain Cherrier, Design Patterns

Créateur (GRASP) : contre-indication

- Creation may require significant complexity:
  - recycling instances for performance reasons
  - conditionally creating instances from a family of similar classes
- In these instances, other patterns are available…
- We'll learn about Factory and other patterns later…

Faible couplage

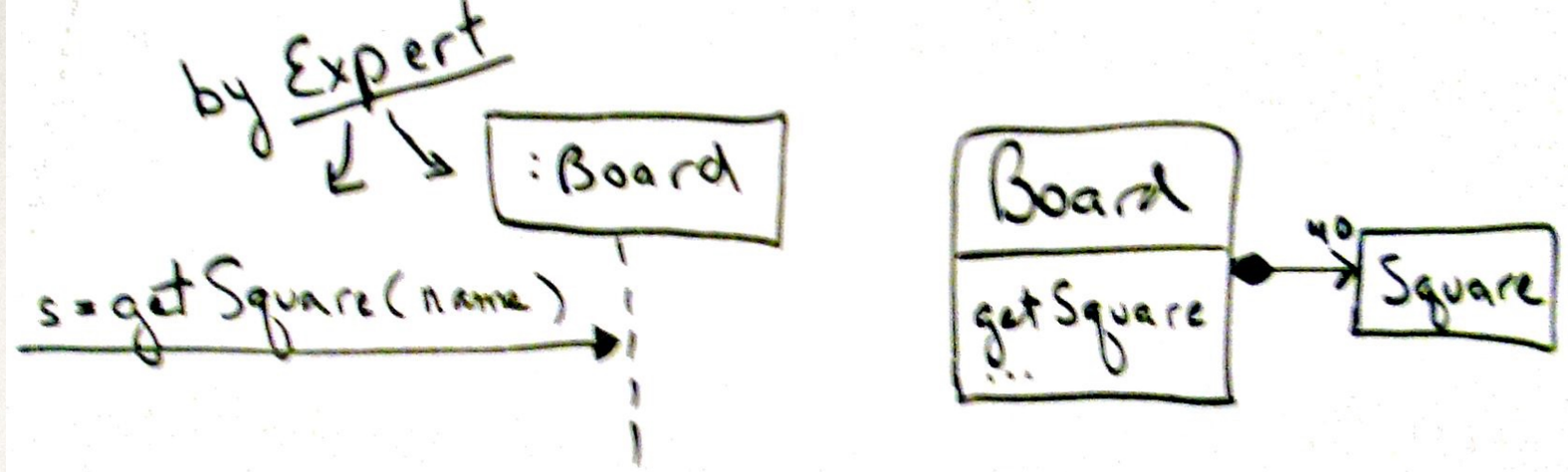
- **Problème**
  - Comment minimiser les dépendances, réduire l'impact des changements, et augmenter la réutilisation ?
- **Solution**
  - Affecter une responsabilité de sorte que le couplage reste faible. Appliquer ce principe pour évaluer les solution possibles.
- **Couplage**

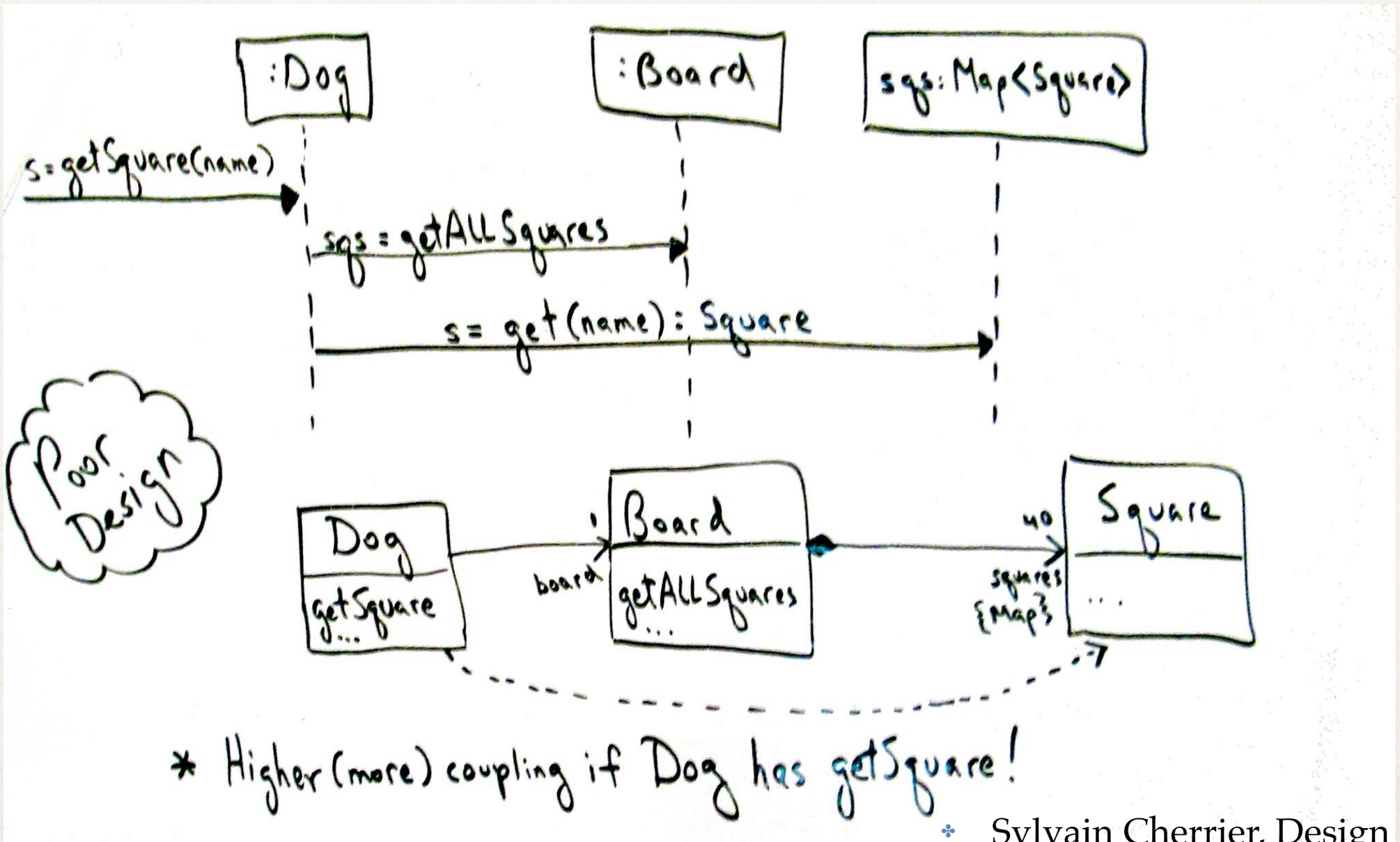
Mesure du degré auquel un élément est lié à un autre

Couplage

- Exemples classiques de couplages de *TypeX* vers *TypeY* dans un langage OO
  - *TypeX* a un attribut qui réfère à *TypeY*
  - TypeX a une méthode qui référence TypeY
  - TypeX est une sous-classe directe ou indirecte de TypeY
  - TypeY est une interface et TypeX l'implémente

Couplage faible ?





Couplage faible (suite) !

- Avoir un couplage faible signifie une faible dépendance aux autres classes

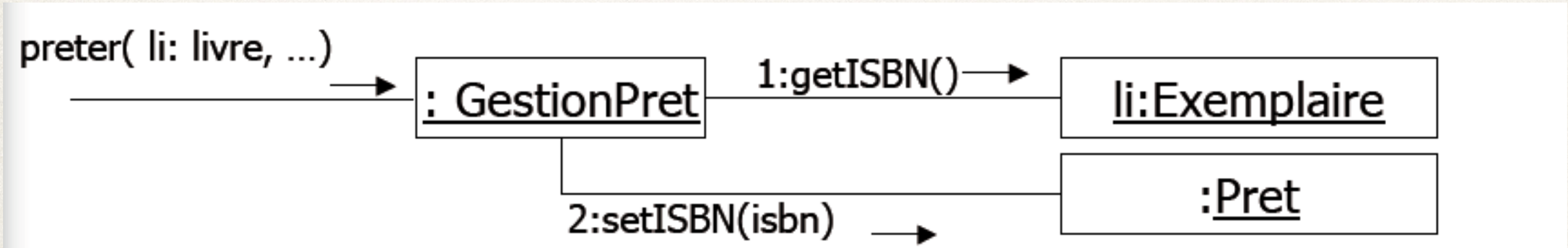
- Problèmes du couplage fort
  - Un changement dans une classe force à changer toutes ou la plupart des classes liées.
  - Les classes prises isolément sont difficiles à comprendre.
  - Réutilisation difficile : l'emploi d'une classe nécessite celui des classes dont elle dépend.

- Bénéfices du couplage faible
  - Exactement l'inverse

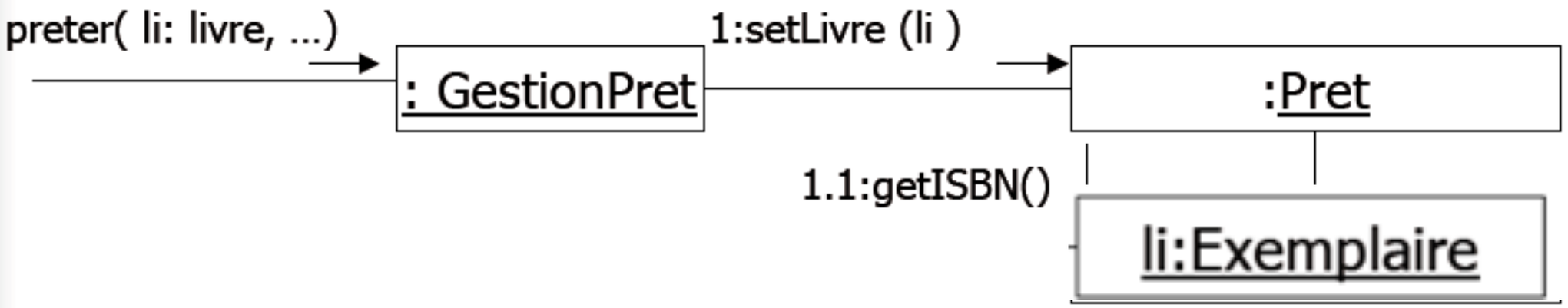
- Sylvain Cherrier, Design Patterns

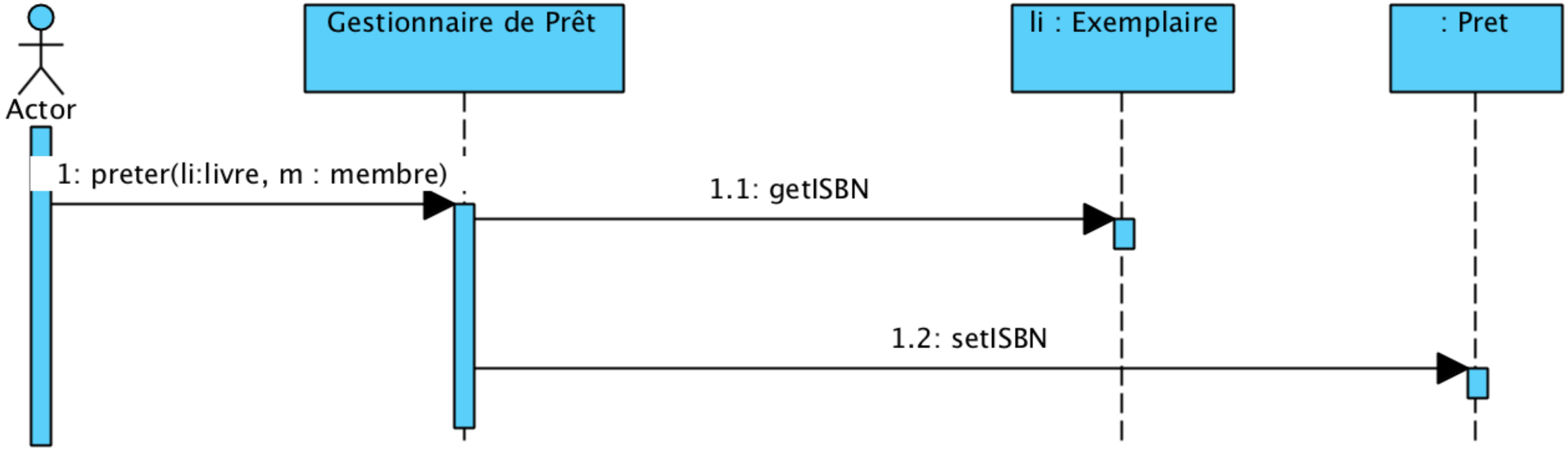
Couplage faible : Exemple

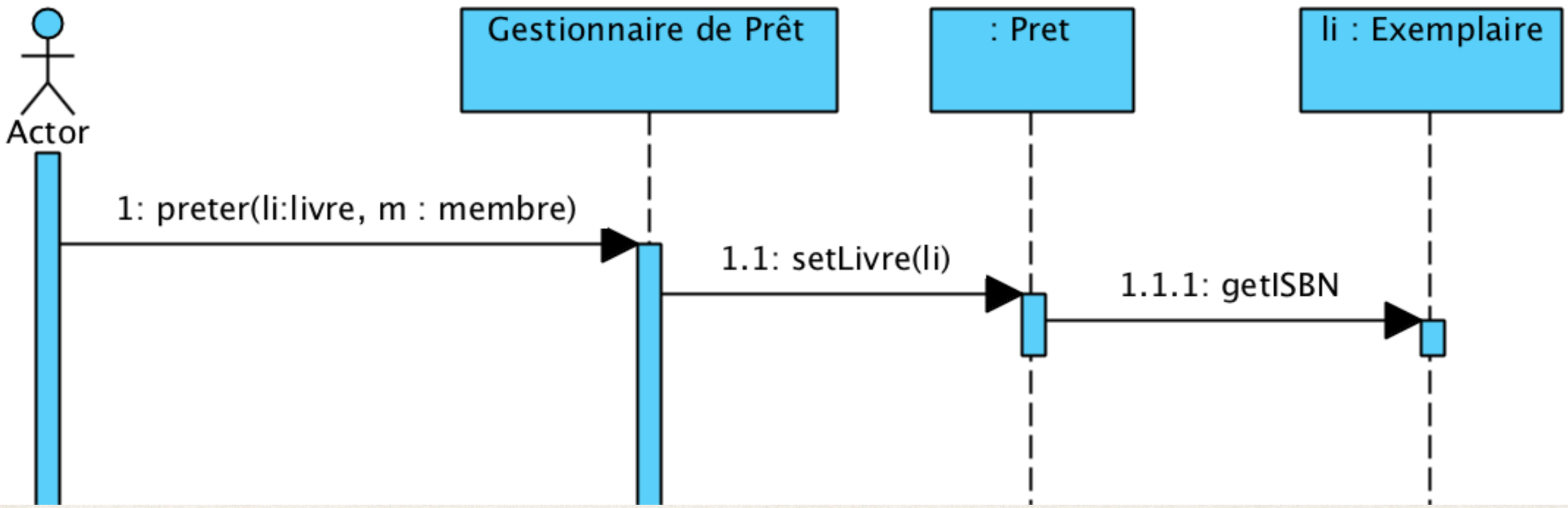
- Pour l' application de bibliothèque, il faut mettre l'ISBN d'un Exemplaire dans le Prêt (...). Quelle classe en sera responsable ?



Que choisir ?




Actor
Gestionnaire de Prêt
li : Exemplaire
: Pret
1: preter(li:livre, m : membre)
1.1: getISBN
1.2: setISBN


Actor
Gestionnaire de Prêt
: Pret
li : Exemplaire
1: preter(li:livre, m : membre)
1.1: setLivre(li)
1.1.1: getISBN

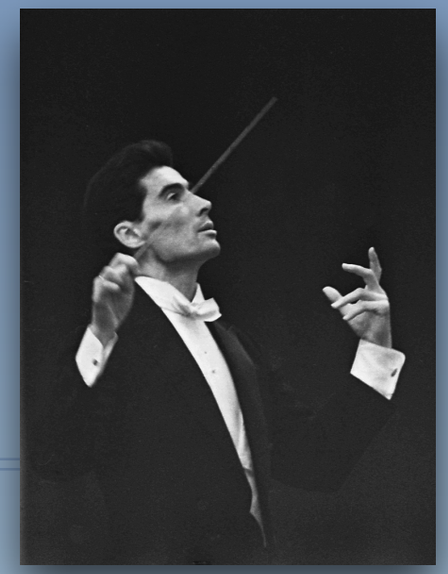
Couplage faible : discussion

- Pas de mesure absolue de «quand un couplage est trop fort»
- De façon générale, les classes qui sont très génériques par nature, et très réutilisables doivent avoir un faible couplage.
- **Un fort couplage n'est pas dramatique avec des éléments très stables**
  - Java.util par exemple

Couplage faible : discussion

- Cas extrême de faible couplage
  - des objets incohérents, complexes, qui font tout le travail
  - des objets isolés, non couplés, qui servent à stocker les données
  - peu ou pas de communication entre objets
  - une mauvaise conception qui va à l'encontre des principes OO (collaboration d'objets)
- Bref
  - un couplage modéré est nécessaire et normal pour créer des systèmes OO

Contrôleur



- **Problème**
  - Quel est le premier objet au delà de l'IHM qui reçoit et coordonne (contrôle) une opération système ?
    - opération système : événement majeur entrant dans le système
- **Solution**
  -

Choisir ou Inventer un objet dans la couche application pour gérer le «contrôle»

- Sylvain Cherrier, Design Patterns

Contrôleur (GRASP)

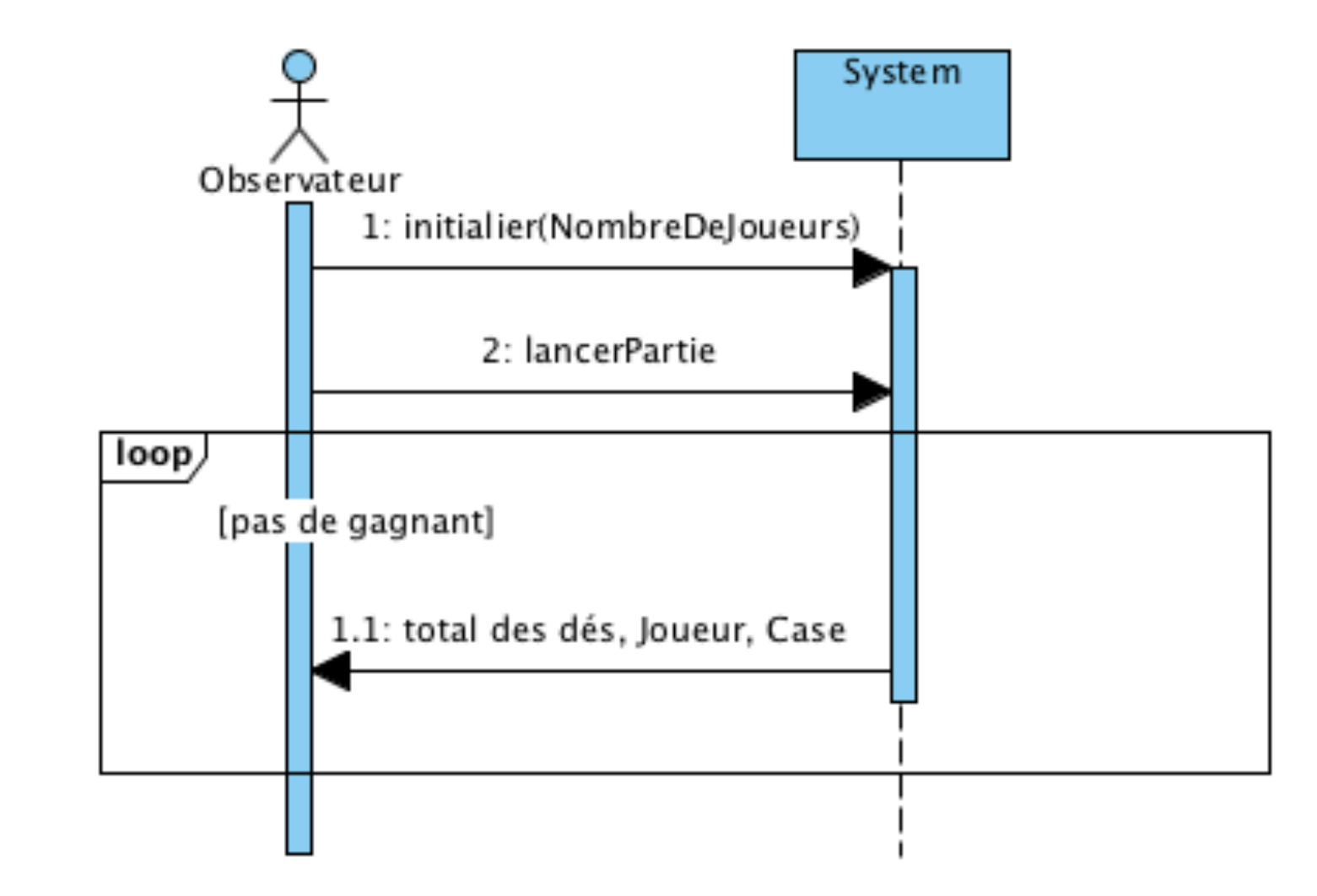
- **Problème**
  - Quel est le premier objet au delà de l'IHM qui reçoit et coordonne (contrôle) une opération système ?

> - **contrôleur** : objet n'appartenant pas à l'IHM ayant la responsabilité de recevoir ou de gérer un événement système

- **Solution**
  - Affecter cette responsabilité à une classe qui correspond à l'un des cas suivants
    - elle représente le système global, un sous-système majeur, un équipement sur lequel le logiciel s'exécute (eq. à des variantes d'un contrôleur Façade)
    - elle représente un scénario de cas d'utilisation dans lequel l'événement système se produit

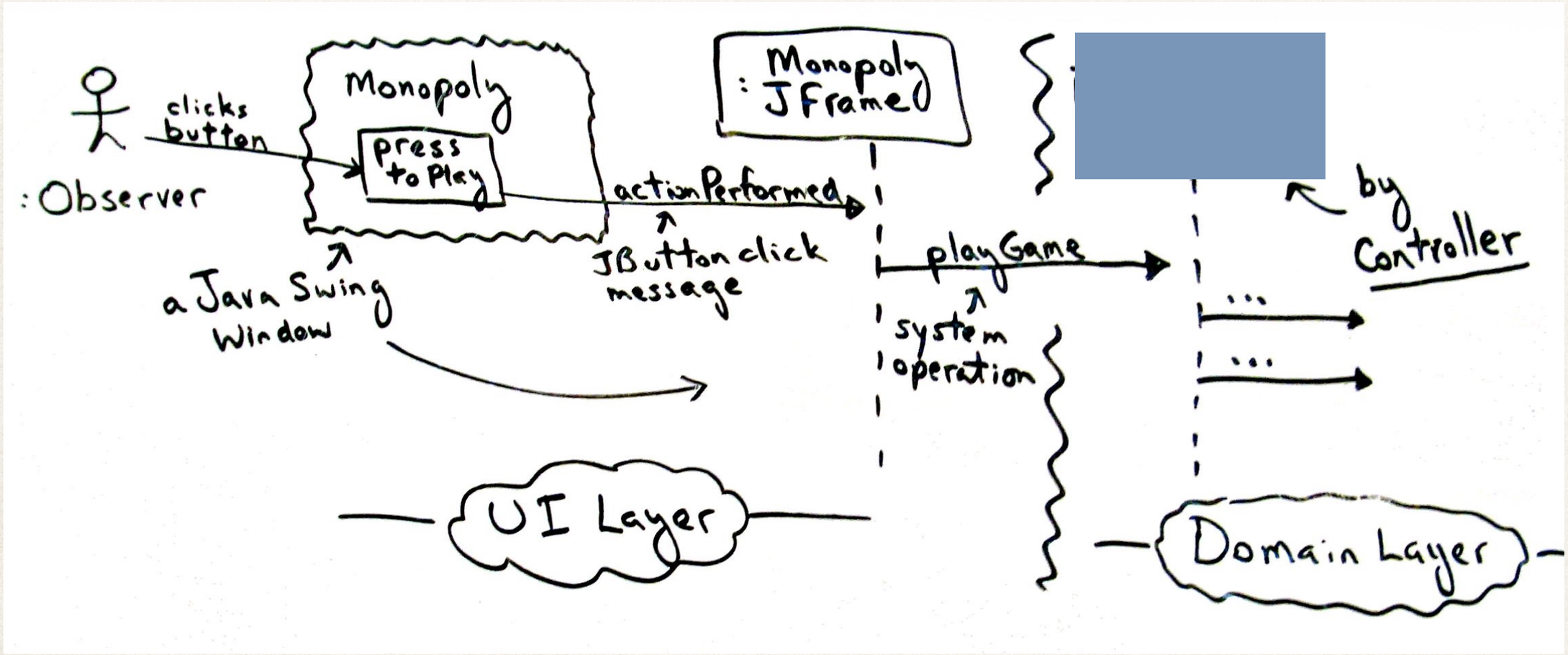
Contrôleur : exemple

- Qui est responsable de l'opération *lancerPartie* ? (vue système)



Contrôleur : exemple (suite)

- Qui est responsable de l'opération playGame ? Quelle classe représente le système ou le scénario?



Quelle(s) classes connectent la couche «Présentation» à la couche chargée de la logique applicative ?

Contrôleur **Facade**

- Représente tout le système
  - exemples : ApplicationController, etc.
- **A utiliser quand**
  - il y a peu d'événements système

(Il peut lui même déléguer à d'autres contrôleurs)

- d'après Sylvain Cherrier, Design Patterns

Contrôleur Cas d'utilisation

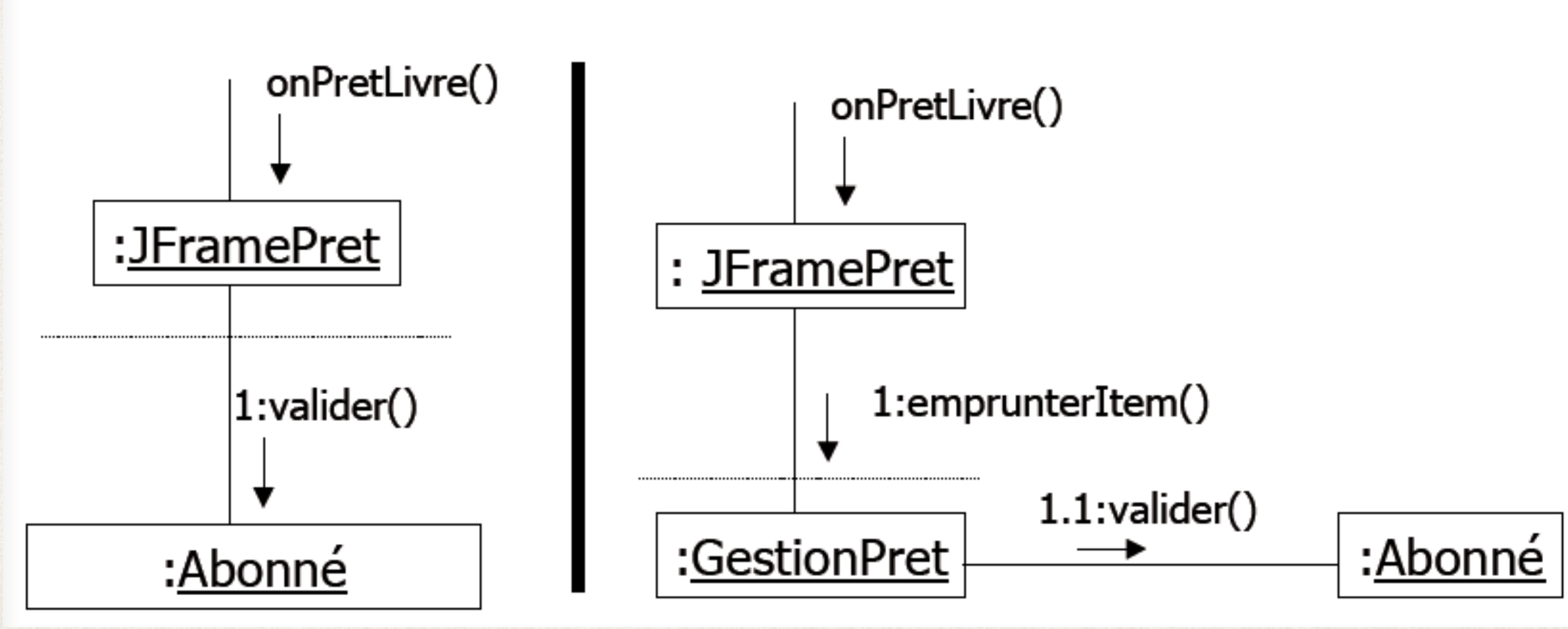
- Utiliser le même contrôleur pour tous les événements d'un cas d'utilisation
- Un contrôleur différent pour chaque cas d'utilisation
  - contrôleur artificiel («Handler»), pas un objet du domaine
- **A utiliser quand**
  - les autres choix amènent à un fort couplage ou à une cohésion faible (contrôleur trop chargé)
  - il y a de nombreux événements systèmes qui appartiennent à plusieurs processus
    - répartit la gestion entre des classes distinctes et faciles à gérer
    - permet de connaître et d'analyser l'état du scénario en cours

Contrôleur trop chargé (**mauvais**)

- Pas de focus; prend en charge trop de responsabilités
  - un seul contrôleur reçoit tous les événements système
    - **ajouter des contrôleurs**
  - le contrôleur effectue la majorité des tâches nécessaires pour répondre aux événements systèmes
    - **un contrôleur doit déléguer à d'autres objets les tâches à effectuer**
  - il a beaucoup d'attributs et gère des informations importantes du système ou du domaine
    - **ces informations doivent être distribuées dans les autres objets**
    - **ou doivent être des duplications d'informations trouvées dans d'autres objets**

Remarque : couche présentation

- Les objets d'interfaces graphique (fenêtres, applets) et la couche de présentation ne doivent pas prendre en charge les événements système
  - c'est la responsabilité de la couche domaine



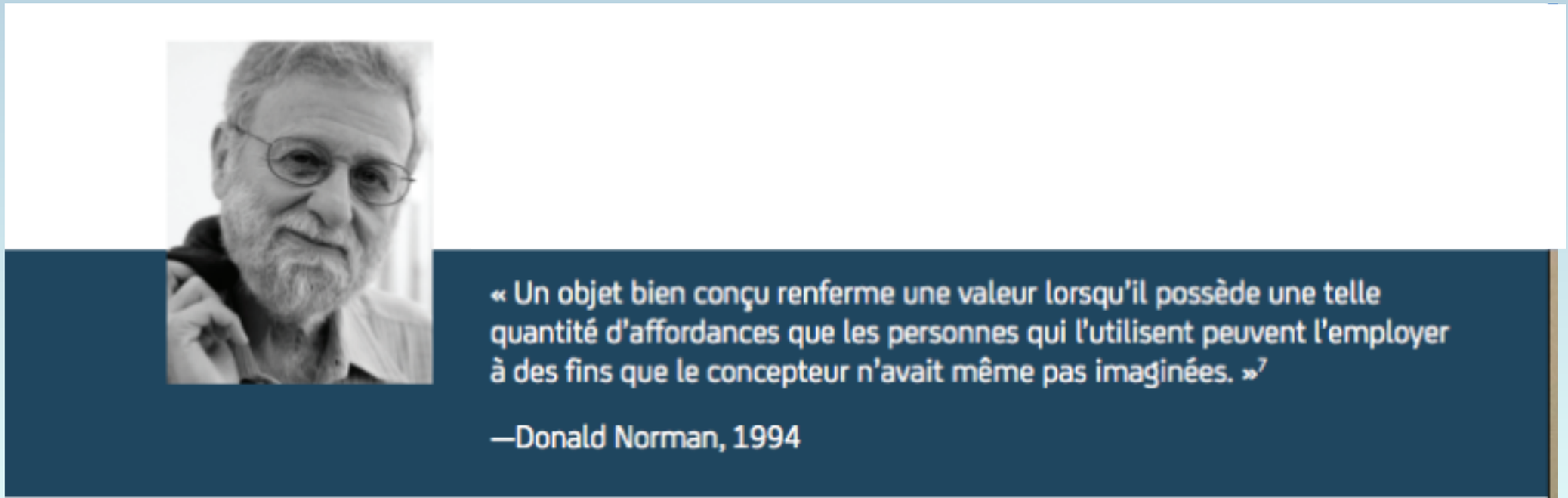
Contrôleur : Avantages

- Meilleur potentiel de réutilisation
  - moyen de séparer les connaissances du domaine de la couche présentation/IHM
- Capture l'état d'un Cas d'Utilisation
  - Permet de s'assurer que les opérations système se produisent dans le bon ordre

Forte Cohésion

- **Cohésion**
  - La cohésion (la cohésion fonctionnelle) est une mesure de l'étroitesse des liens et de la spécialisation des responsabilités d'un élément (d'une classe)
  - Une classe qui a des responsabilités étroitement liées et n'effectue pas un travail gigantesque est fortement cohésive

- Sylvain Cherrier, Design Patterns



Forte Cohésion (GRASP)

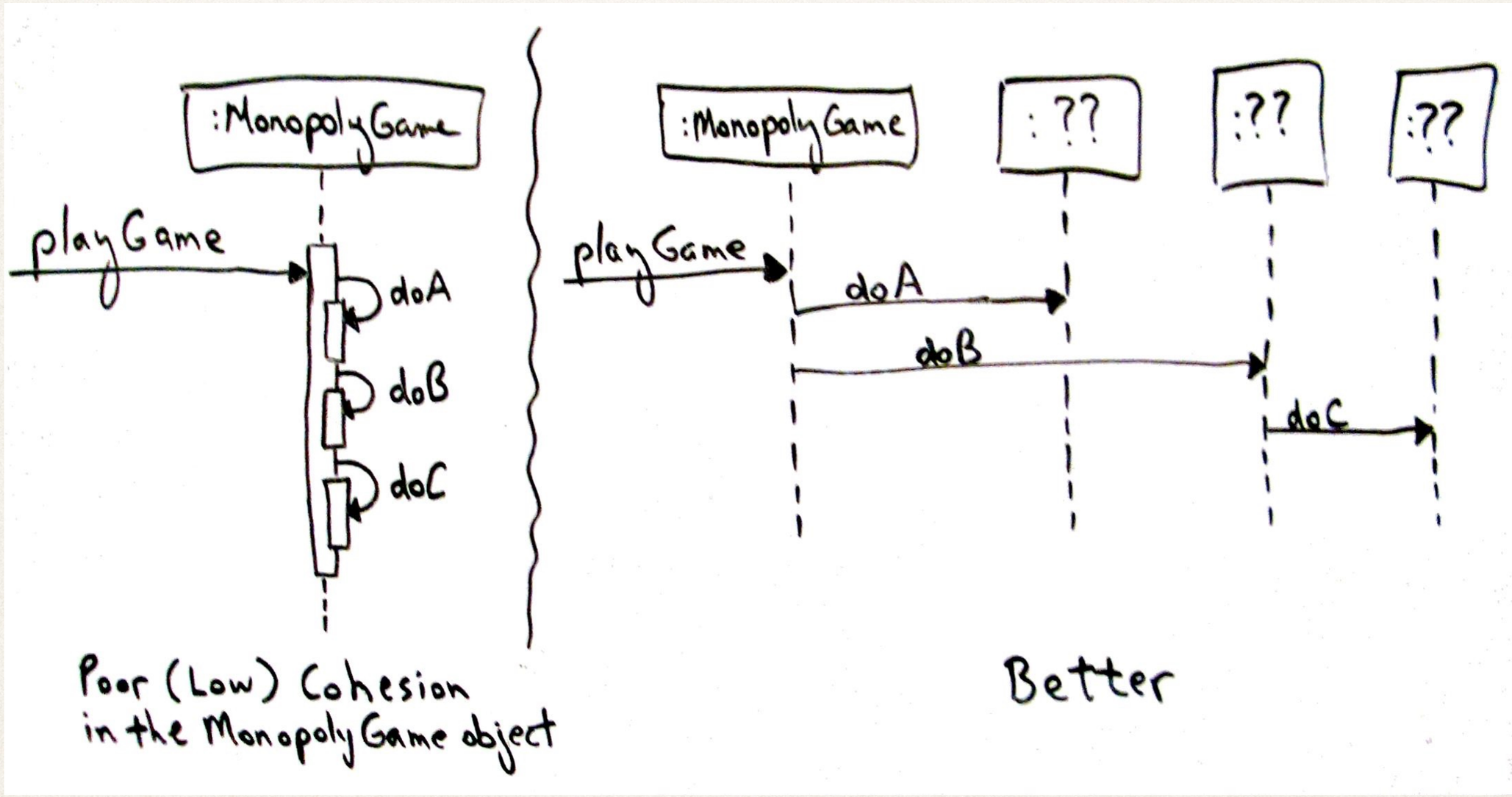
- **Problème**
  - Comment parvenir à maintenir la complexité gérable ? Comment s'assurer que les objets
    - restent compréhensibles et faciles à gérer?
    - qu'ils contribuent au faible couplage ?
- **Solution**
  - Attribuer une responsabilité de telle sorte que la **cohésion** reste forte.
  - Appliquer ce principe pour évaluer les solutions possibles.

Forte Cohésion (suite)

- Les classes ayant une **faible cohésion** effectuent des tâches sans liens entre elles ou effectuent trop de tâches

- Problèmes des classes à faible cohésion :
  - Difficiles à comprendre
  - Difficiles à réutiliser
  - Difficiles à maintenir
  - Fragiles, constamment affectées par le changement

Forte Cohésion : Exemple



Délégation des responsabilités et coordination des tâches

Absence de cohésion

Class GameBoard {

- **public GamePiece[ ][ ] getState()** - Méthode copiant la grille dans un tableau temporaire, résultat de l ’appel de la méthode.

- **public Player isWinner() -** vérifie l’état du jeu pour savoir s ’il existe un gagnant, dont la référence est retournée. Null est retourné si aucun gagnant.

- **public boolean isTie() -** retourne true si aucun déplacement ne peut être effectué, false sinon.

- **public void display ()** - affichage du contenu du jeu. Espaces blanc affichés pour chacune des références nulles.

https://www.emse.fr/~boissier/enseignement/aco/pdf/1.DesignPattern.MSGL.4pp.pdf

Cohésion

Class GameBoard {

public GamePiece[ ][ ] getState()

public Player isWinner()

public boolean isTie() - -

}

Class BoardDisplay {

public void displayBoard (GameBoard gb) - affichage du contenu du jeu
Espaces blanc affichés pour chacune des références nulles. }

https://www.emse.fr/~boissier/enseignement/aco/pdf/1.DesignPattern.MSGL.4pp.pdf

Forte Cohésion : discussion

- Comme le couplage faible, un **critère d'évaluation** à garder en tête pendant toute la conception

- *[Booch] : Il existe une cohésion fonctionnelle quand les éléments d'un composant (eg. une classe) travaillent tous ensemble pour fournir un comportement bien délimité »*

Forte Cohésion

- Une classe de forte cohésion a un petit nombre de méthodes, avec des fonctionnalités hautement liées entre elles, et ne fait pas trop de travail

- Un test

  - **Décrire une classe avec une seule phrase.**

- *[Booch] : la **modularité** est la propriété d'un système qui a été décomposé en un ensemble de modules cohésifs et peu couplés*

Forte Cohésion : discussion

- Bénéfices de la forte cohésion
  - Augmentation de la clarté et de la compréhension de la conception
  - Maintenance et amélioration simplifiées
  - Signifie en général couplage faible
  - Meilleur potentiel de réutilisation

Polymorphisme

- **Problème**
  - Comment gérer des alternatives dépendantes des types ?
  - Comment créer des composants logiciels « enfichables » ?

- **Solution**
  - Quand des fonctions ou des comportements connexes varient en fonction du type (classe), affectez les responsabilités - en utilisant des opérations polymorphes - aux types pour lesquels le comportement varie
    - ne pas utiliser de test sur le type d'un objet ou une logique conditionnelle (if/then/else) pour apporter des alternatives basées sur le type

Polymorphisme (GRASP)

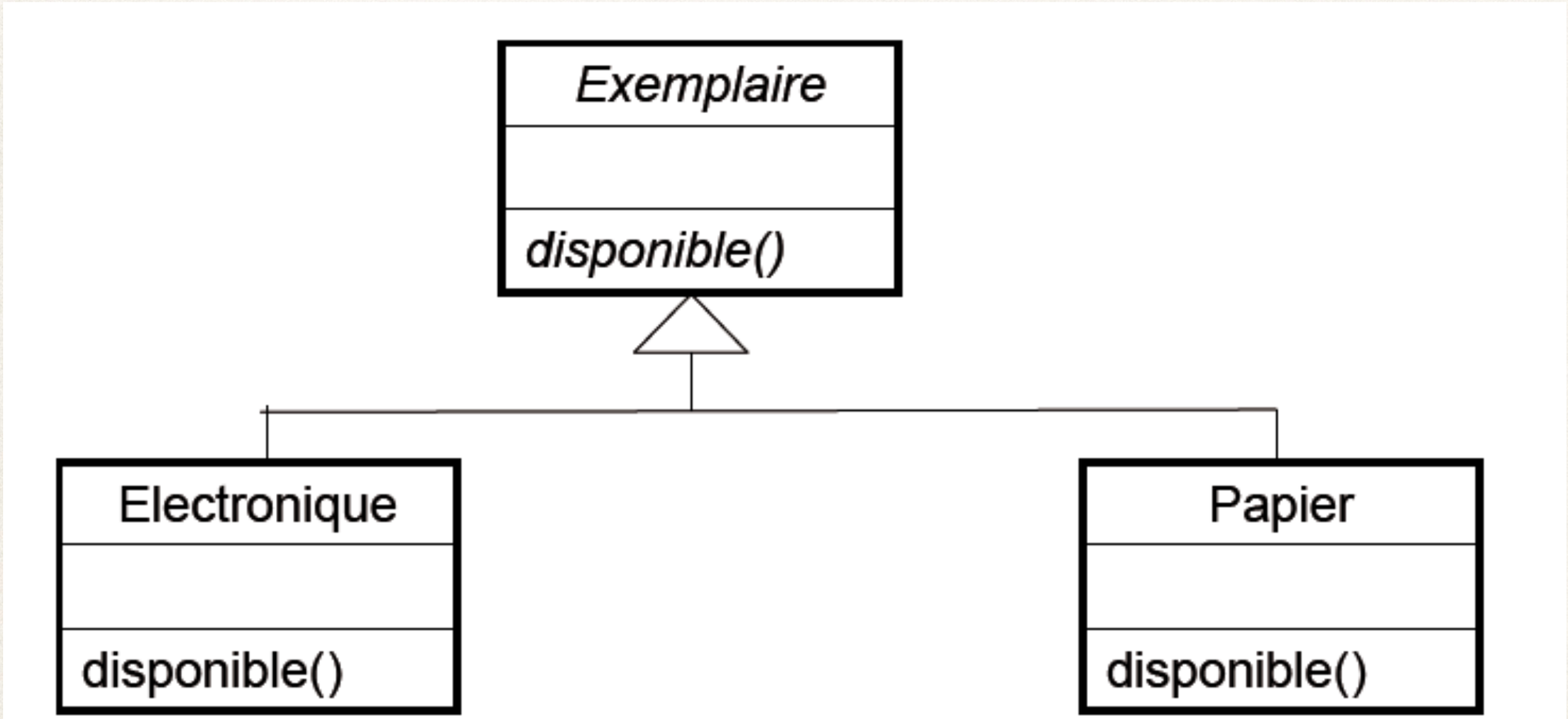
- **Polymorphisme**
  - donner le même nom à des «services» dans différents objets

Cercle vertueux « d'ignorance »

- Je ne peux pas coder une logique qui dépend de « qui fait quoi » !
- anti-spaghetti

Polymorphisme : exemple

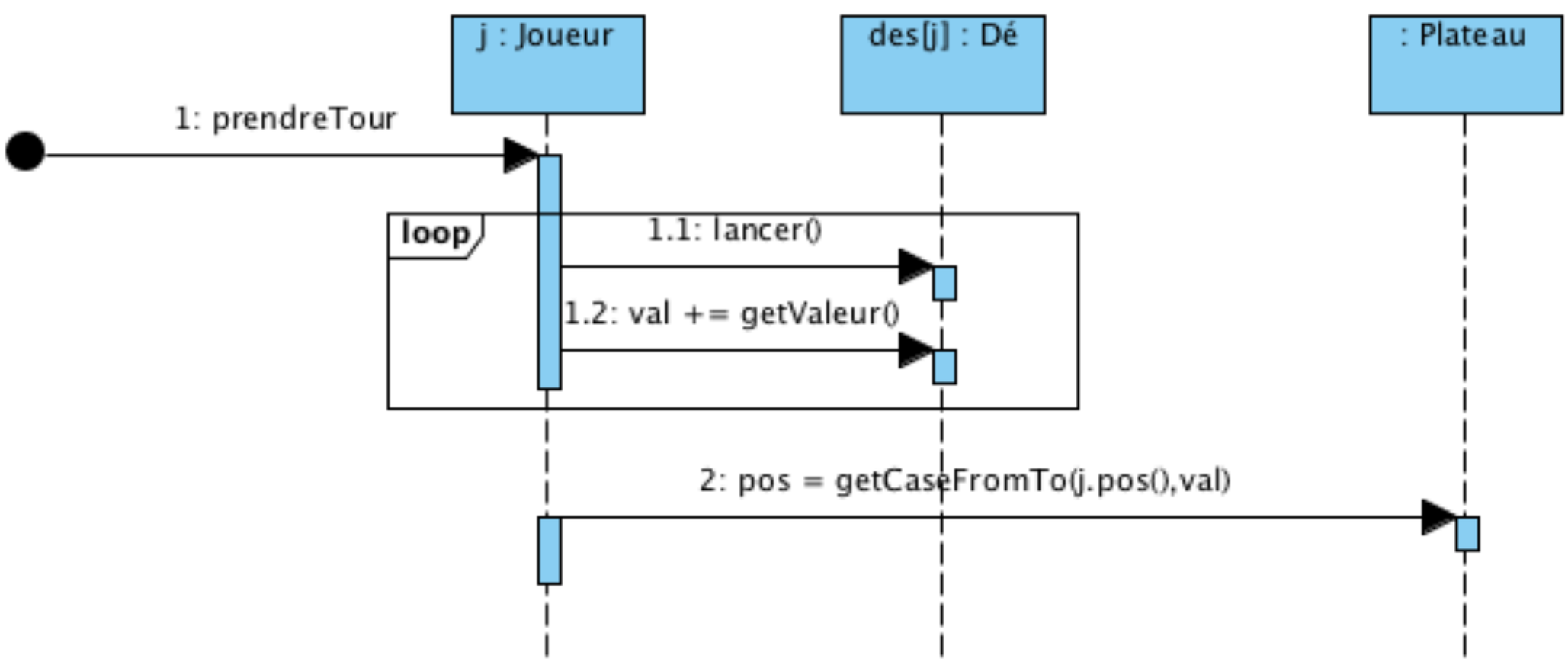
- Bibliothèque : qui doit être responsable de savoir si un exemplaire est disponible ?



- Sylvain Cherrier, Design Patterns

Polymorphisme : exemple

- Monopoly : En fonction de la case sur laquelle atterrit le joueur le comportement est différent : Sur la case départ, le joueur reçoit 200 $, Sur la case impôt, le joueur paie 10% de son cash, sur la case AllezEnPrison, le joueur va sur la case Prison.



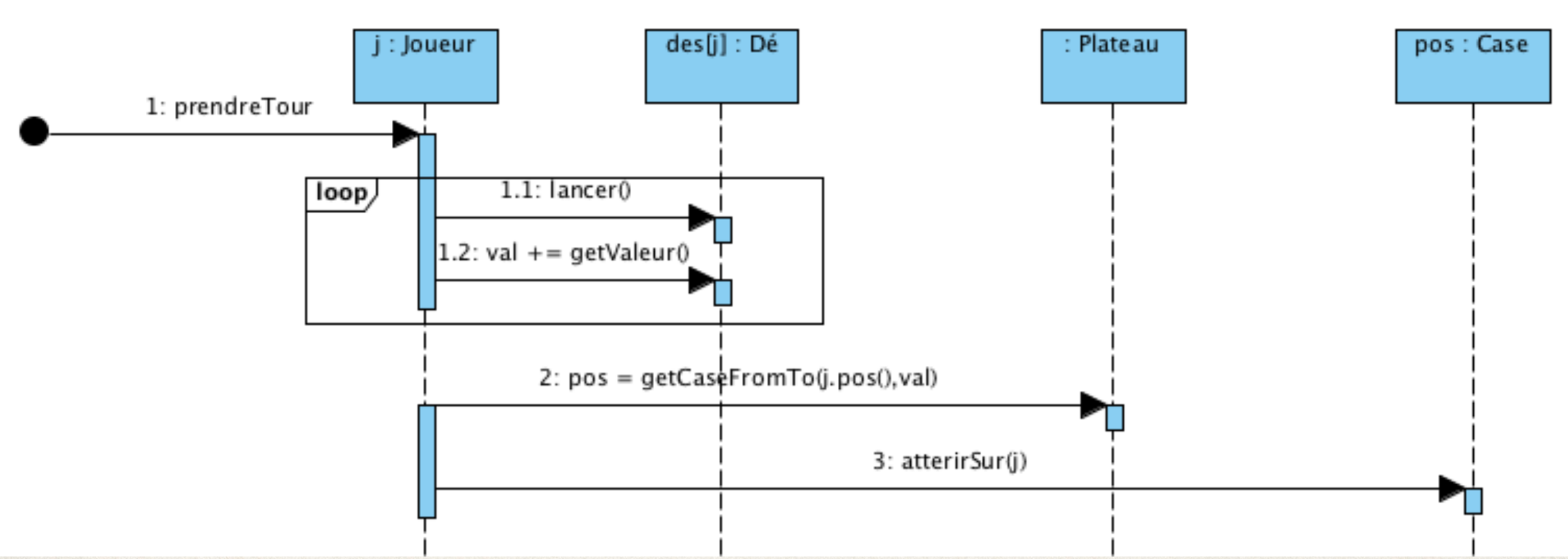
Monopoly

- 28 cartes de propriété
  - **Achat de propriétés :** Si vous vous arrêtez sur une propriété qui n'appartient pas déjà à un autre joueur, ... **Etre propriétaire; Arrêt sur une propriété déjà achetée; Arrêt sur les services publics; gares**

- 16 cartes chance & 16 cartes caisse de communauté
  - Lorsque vous tombez sur une case « Chance » ou « Caisse de communauté » vous devez piocher une carte sur le haut du tas de cartes correspondant. Vous devez ensuite suivre les instructions de celle-ci. Vous pouvez recevoir de l'argent ou au contraire payer une taxe. La carte doit être remise ensuite sous le tas de carte face cachée.

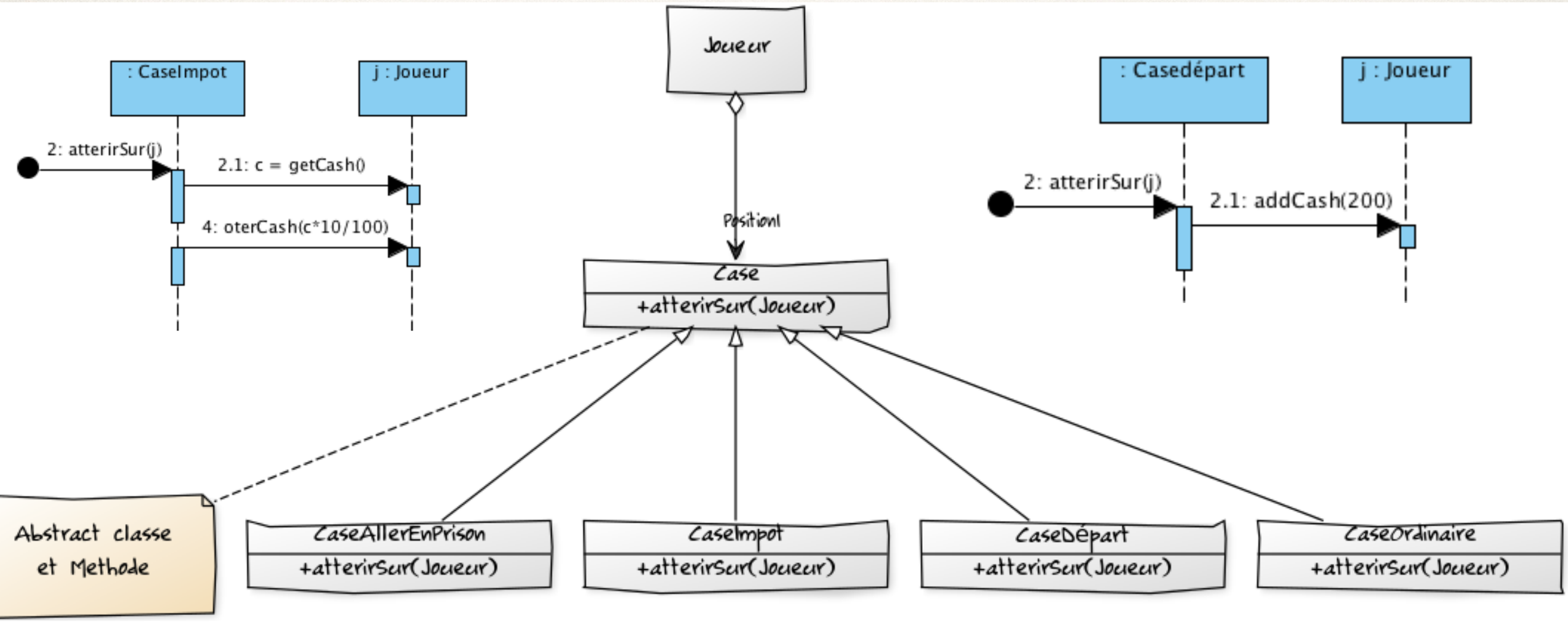
Polymorphisme : exemple

- Monopoly : En fonction de la case sur laquelle atterrit le joueur le comportement est différent : Sur la case départ, le joueur reçoit 200 $, Sur la case impôt, le joueur paie 10% de son cash, sur la case AllezEnPrison, le joueur va sur la case Prison.



Polymorphisme : exemple

- Monopoly : En fonction de la case sur laquelle atterrit le joueur le comportement est différent : Sur la case départ, le joueur reçoit 200 $, Sur la case impôt, le joueur paie 10% de son cash, sur la case AllezEnPrison, le joueur va sur la case Prison.



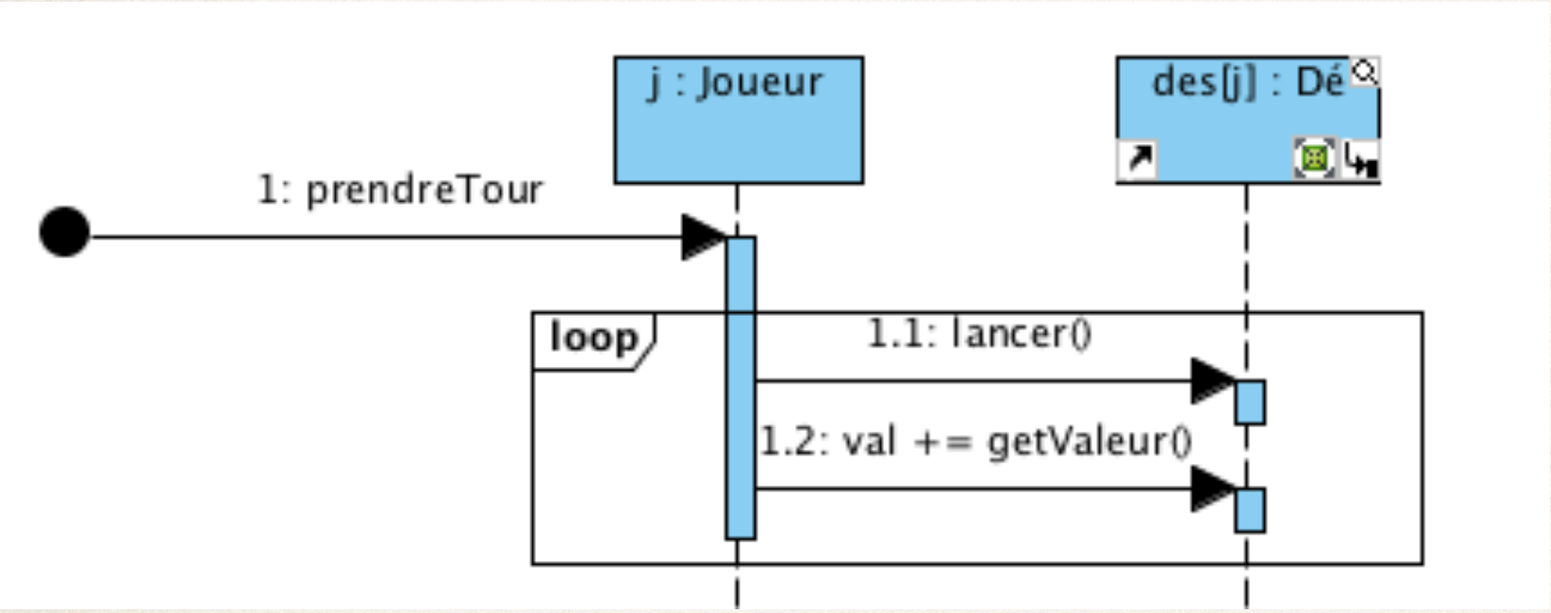
Polymorphisme

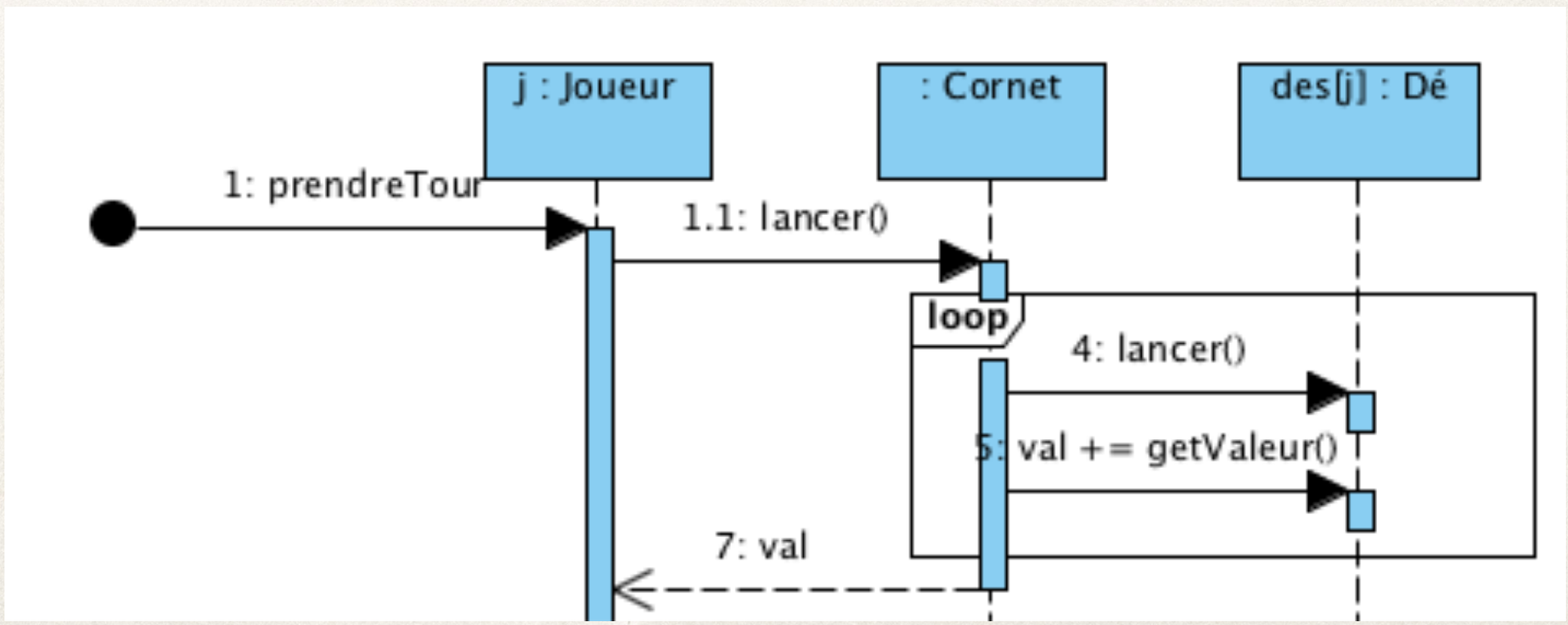
- Implique en général l'utilisation de classes abstraites et d'interfaces
- Avantages
  - Les points d'extension requis par les nouvelles variantes sont faciles à ajouter
  - On peut introduire de nouvelles implémentations sans affecter les clients

Fabrication pure

- **Problème**
  - Que faire quand les concepts du monde réel (objets du domaine) ne sont pas utilisables en respectant le Faible couplage et la Forte cohésion ?
- **Solution**
  - Affecter un ensemble fortement cohésif à une classe artificielle ou de commodité, qui ne représente pas un concept du domaine
    - entité fabriquée de toutes pièces

Fabrication pure : exemple





Fabrication pure : discussion

- Avantages
  - Supporte Faible couplage et Forte cohésion
  - Amélioration de la réutilisabilité
- Attention
  - L'esprit de la conception OO design est centré sur les objets, pas sur les fonctions
  - Ne pas abuser des Fabrications pures

- Sylvain Cherrier, Design Patterns

Conclusion

- Ne pas être dogmatique MAIS «ne pas programmer par coïncidence»
- Sachez évaluer et expliquer votre solution !

"Si vous avez l'impression que vous êtes trop petit pour pouvoir changer quelque chose, essayez donc de dormir avec un moustique... et vous verrez lequel des deux empêche l'autre de dormir." Le Dalaï Lama