GÉNIE LOGICIEL ORIENTÉ OBJET (GLO-2004)
ANALYSE ET CONCEPTION DES SYSTÈMES ORIENTÉS OBJETS (IFT-2007)

Automne 2016

Module 11 - Grands principes en conception orientée objet

Martin.Savoie@ift.ulaval.ca

Bachelier Génie logiciel, Chargé de cours,
département d'informatique et de génie logiciel

UNIVERSITÉ LAVAL

Questions?

L’examen arrive vite!

- Semaine prochaine révision
  - Faire un retour complet sur la matière
- Soyez présent pour poser vos questions

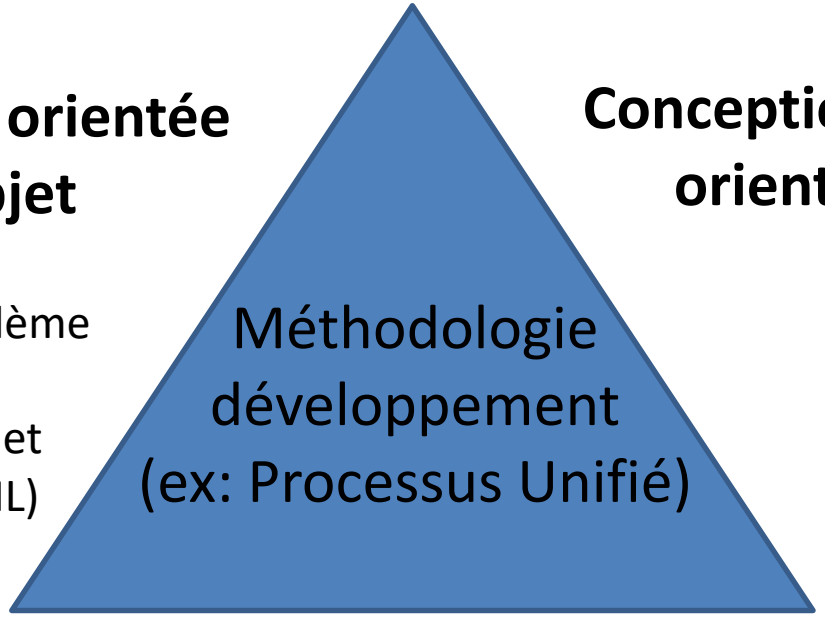
Retour sur certain concept

- Comment allez-vous faire la sauvegarde?
  - Repository?
  - Base de donnée?
  - XML?
  - Json?
  - Toutes d'excellentes réponses, mais...
  - Java.Io.Serialize
- Undo/Redo
  - Command pattern, mais...
  - Java.Io.Serialize

Génie logiciel orienté objet

Analyse orientée objet

- Comprendre le problème
- Décrire la situation à l'aide de documents et diagrammes (ex: UML)



Conception (design) orienté objet

- Concevoir une solution informatique
- Tracer des plans (plus ou moins détaillés) sous la forme de documents et diagrammes (ex: UML)

Programmation orientée objet

Mettre en œuvre la solution à l'aide d'un langage (ex: Java)

UNIVERSITÉ LAVAL

Grand principes OO

L'organisation des classes et de leurs responsabilités est d'une grande importance

Qu’est-ce qui distingue un bon design d’un mauvais design?

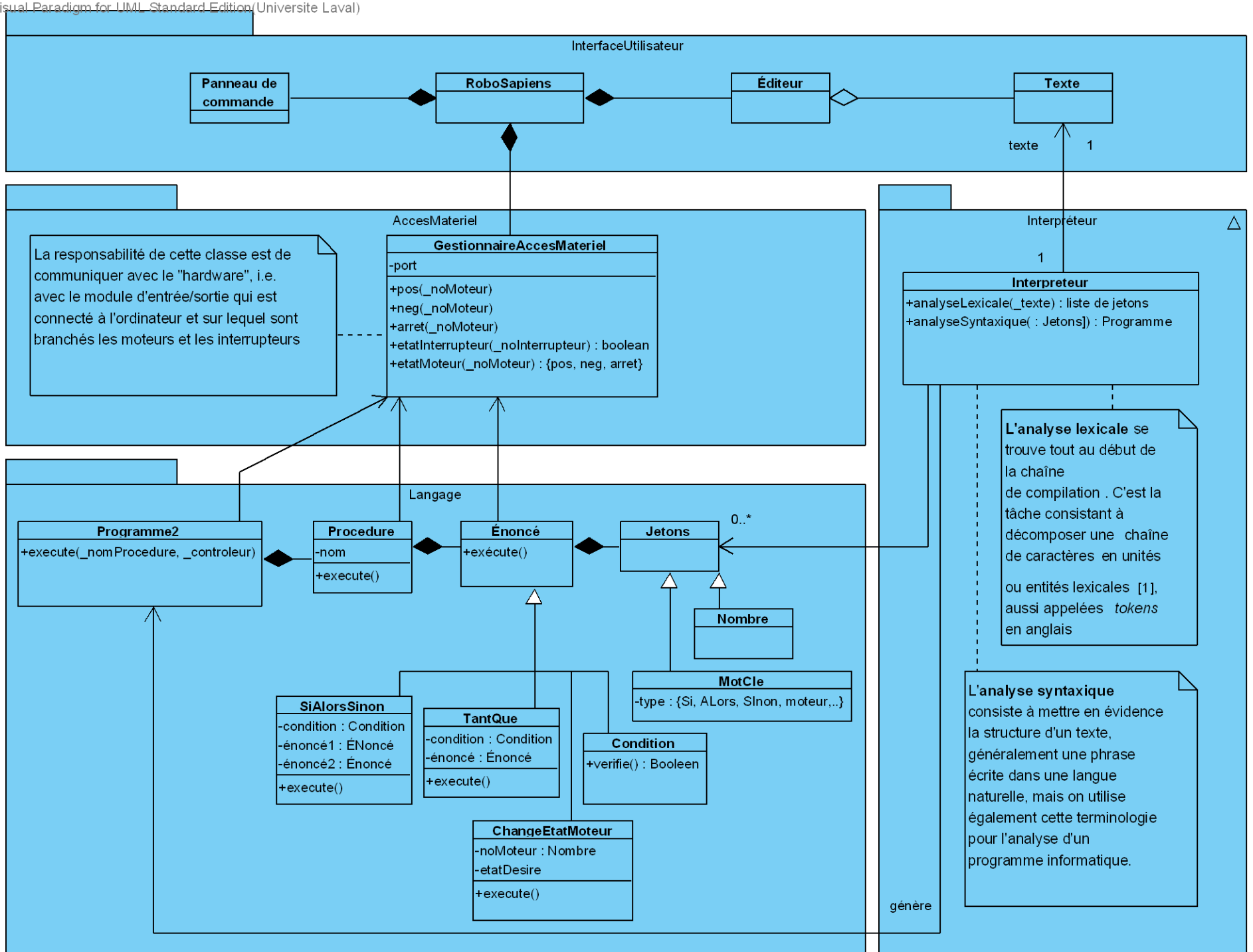
Le développement logiciel ainsi que le génie logiciel c'est de l'art! Pourquoi?

Modularité

- La modularité est la propriété d'un système qui est décomposé en un ensemble de modules **cohésifs** et **faiblement couplés** [Booch94]
  - **Forte cohésion** :
    - Regrouper ensemble dans une même classe des fonctionnalités similaires et apparentées
  - **Faible couplage** :
    - Couplage faible: classes sont relativement autonomes (excellent)
    - Couplage fort: toutes les classes communiquent avec toutes les autres (mauvais)
- Notre objectif pour notre conception/design (notamment pour l'attribution des responsabilités à nos classes)

Architecture logique

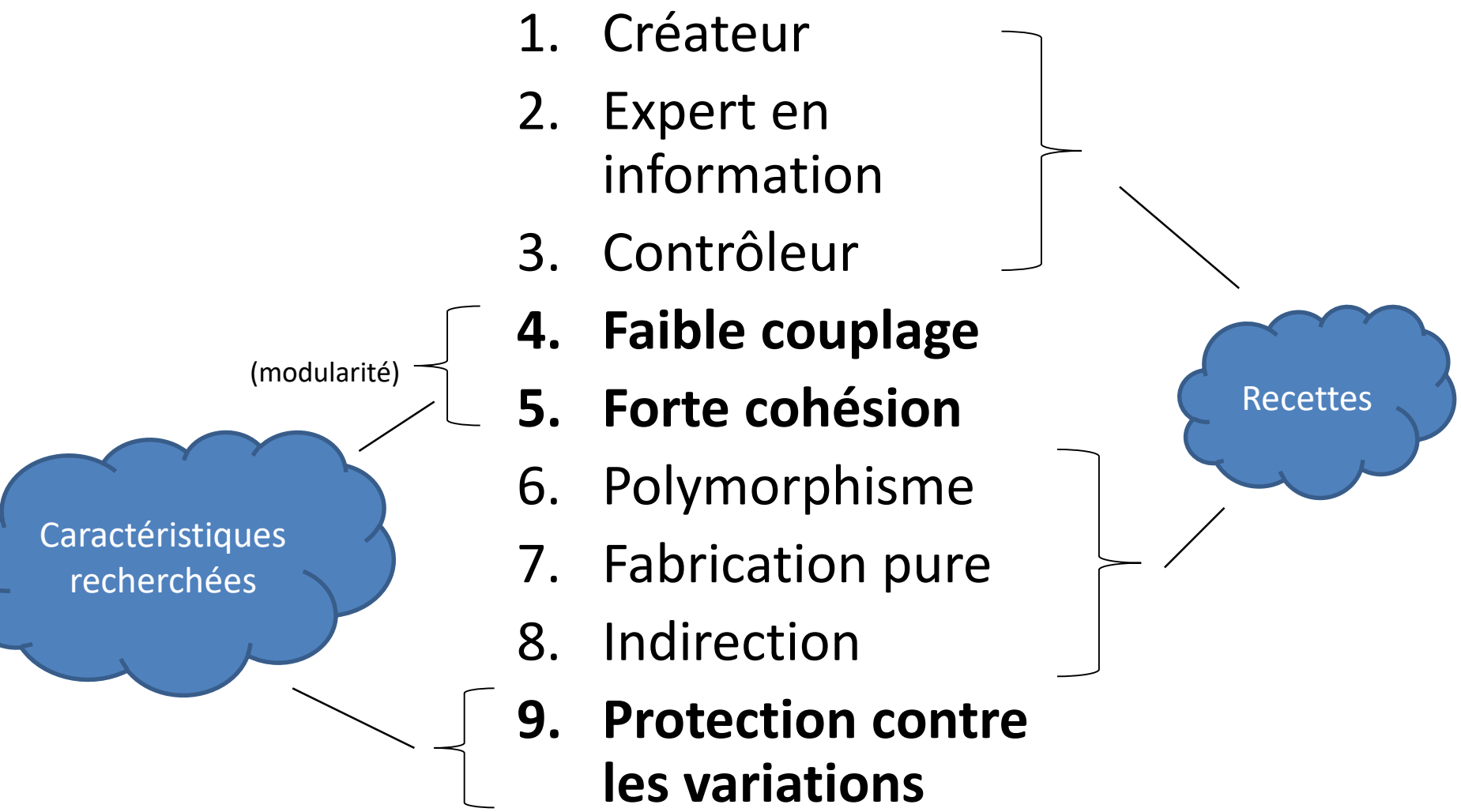
- L'**architecture logique**, c'est l'organisation à grande échelle des classes logicielles en packages, sous-systèmes et couches.
- On la nomme architecture logique, car elle n'implique aucune décision quant à la façon dont ces éléments seront déployés physiquement.



Grands principes de conception orientée objet (GRAPS) et Patrons de conception (GoF)

- Grands principes de conception orientée-objet
  - Larman les appelle les GRASP (General Responsibility Assignment Software Patterns)… mais ce ne sont pas vraiment des « patrons » (au sens classique du terme)
- Patrons de conception / patterns
  - Exemples de constructions/assemblages d'objets classiques que l'on retrouve fréquemment dans les design d'analystes expérimentés (constitue un répertoire d'idées)
  - Décrits dans le livre « Design patterns » de Gamma et al. (« Gang of four »)

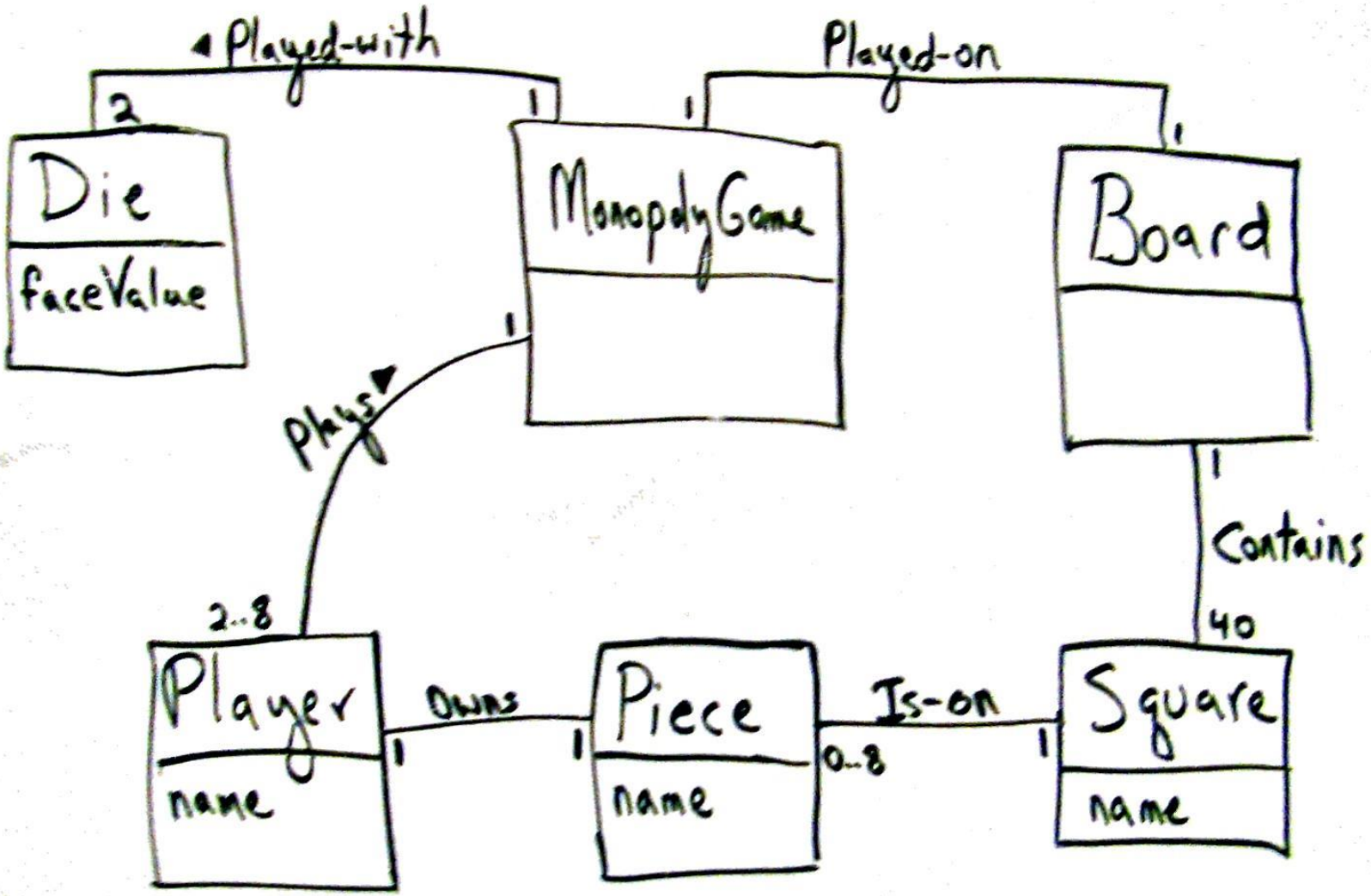
Grands principes (GRASP)



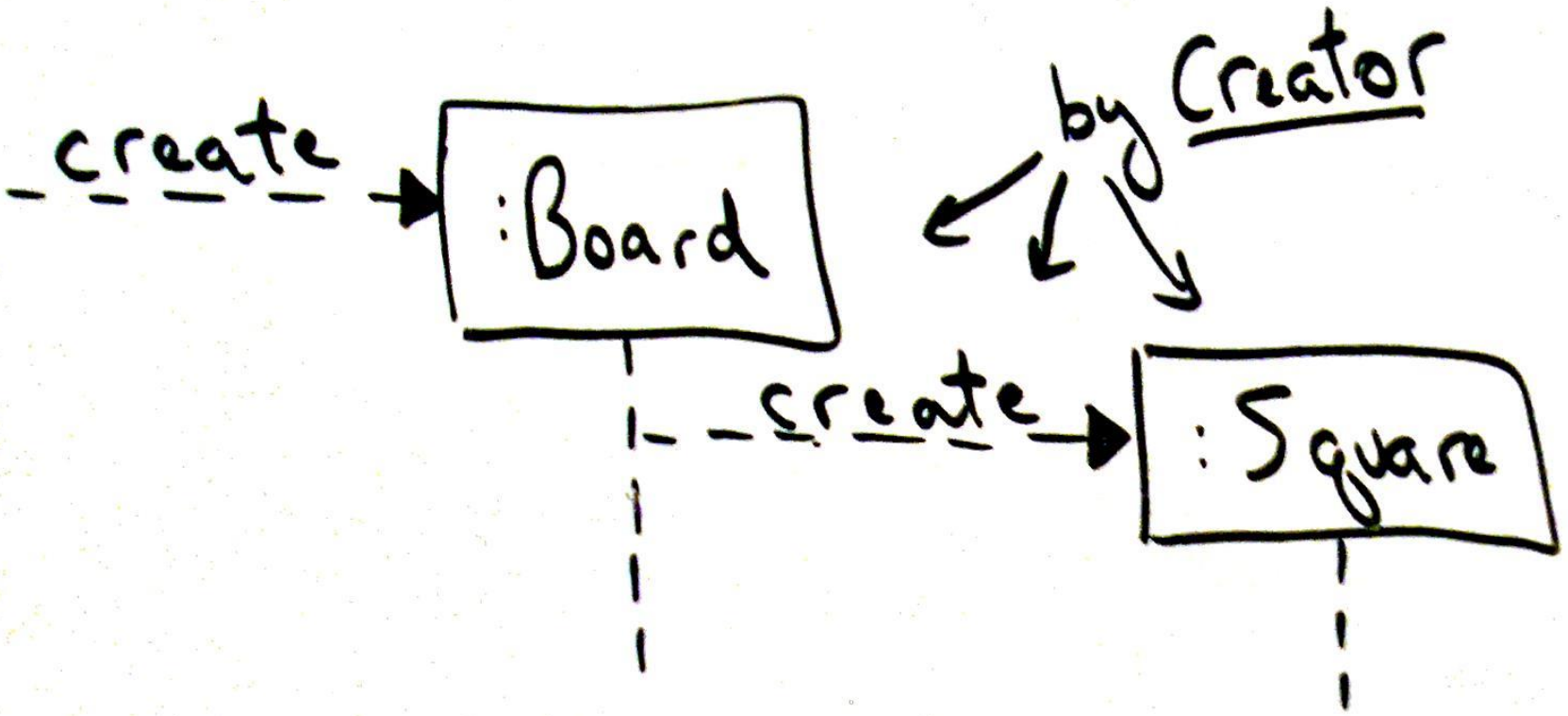
Principe 1 : Créateur

- **Problème**:
  - Qui devrait créer les instances de la classe A?
- **Solution**:
  - La classe qui…
    - Contient ou agrège les A
    - Enregistre les A
    - Utilise étroitement les A
    - Possède les données pour initialiser des objets A

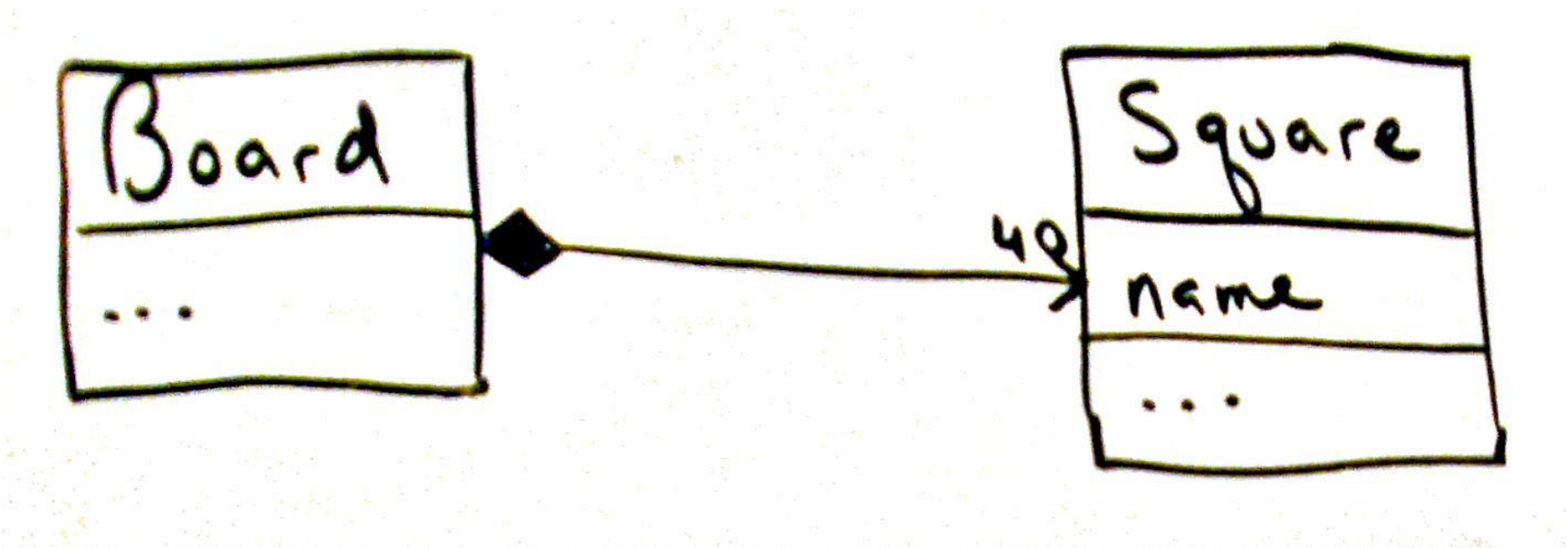
Monopoly – Qui devrait créer les cases?



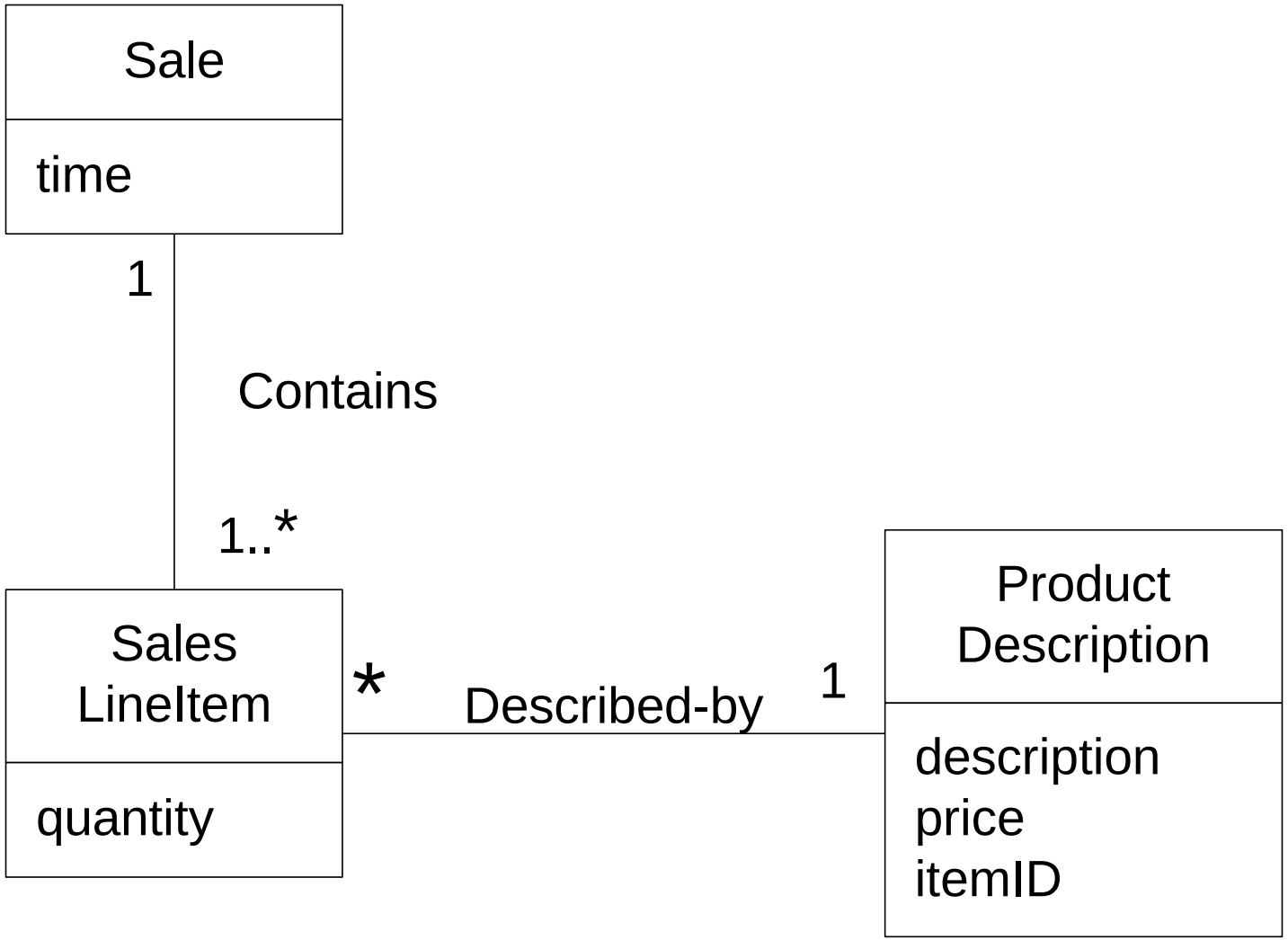
Le tableau de jeu !



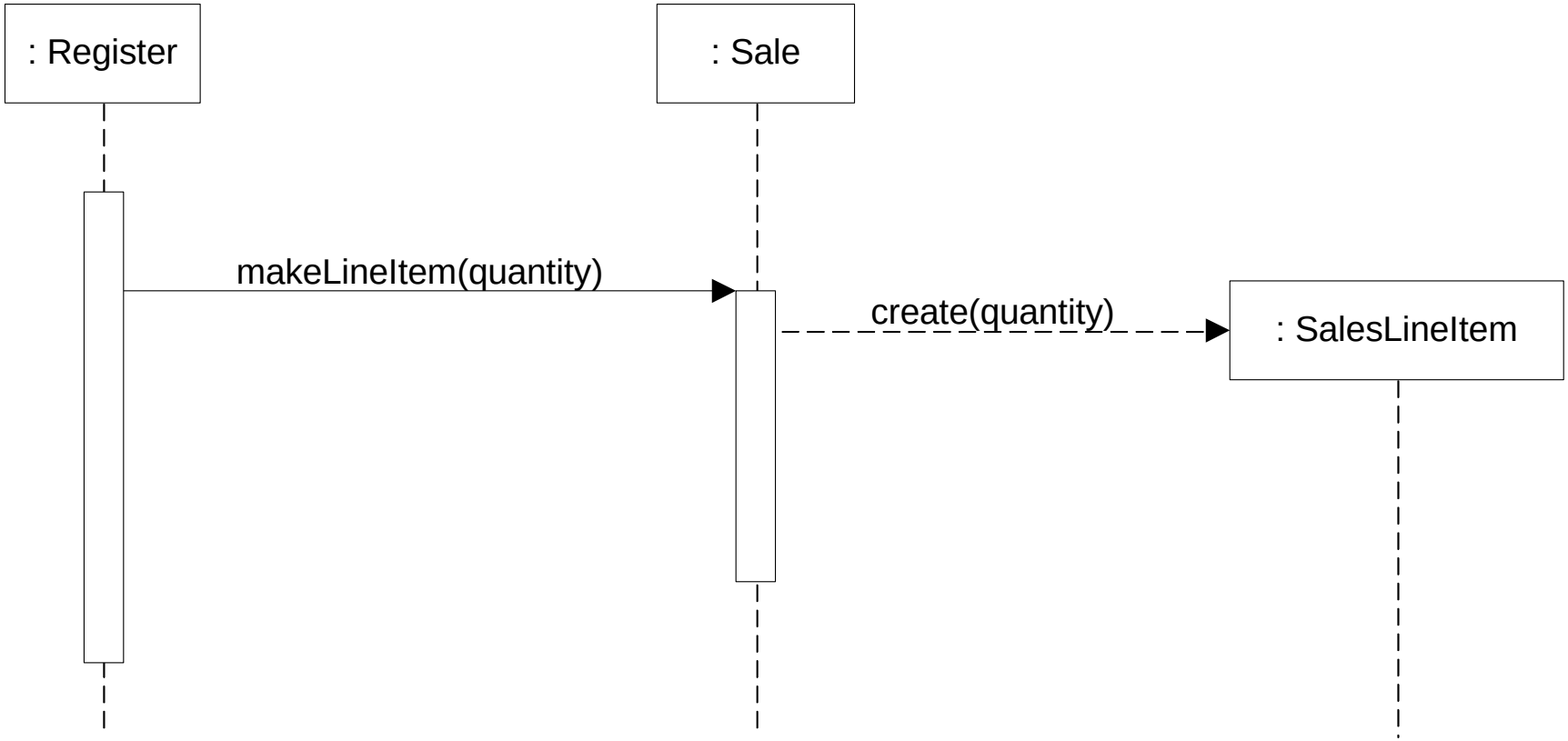
La présence de composition est généralement un bon indice!



Qui crée les “lignes” de la vente?



La vente !

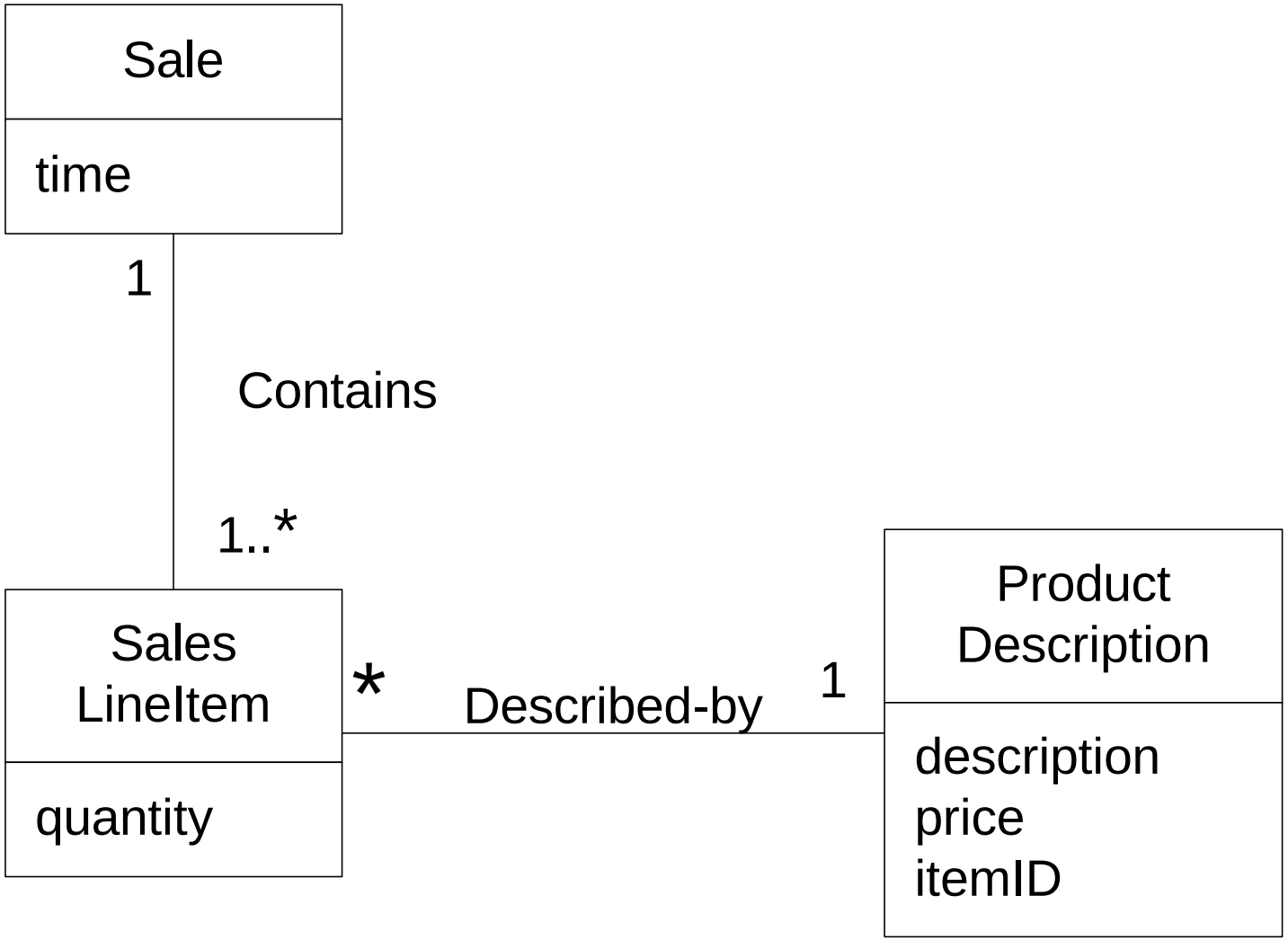


Et dans votre projet de session?

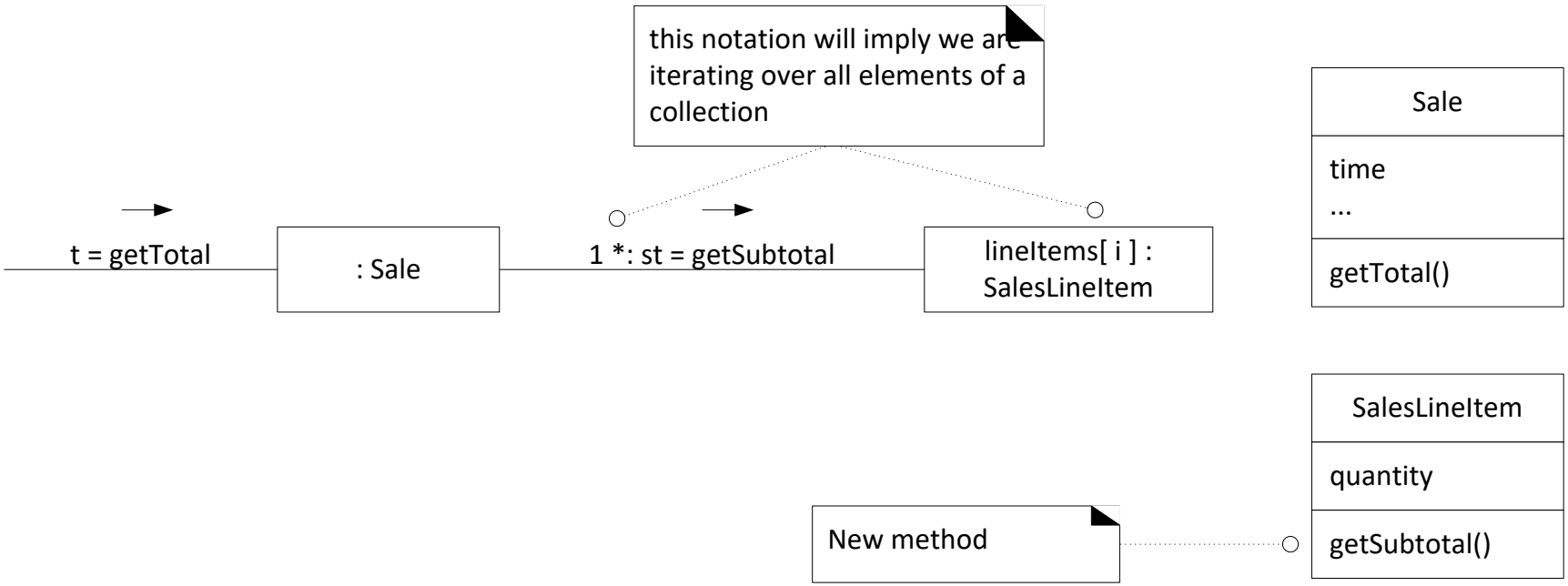
Principe 2: Expert en information

- **Problème**:
  - À quel classe affecter une certaine responsabilité (méthode) ?
- **Solution**:
  - À la classe qui possède les informations nécessaires pour s'en acquitter

Qui calcule le montant total à exiger au client?



Qui calcule le montant total à exiger au client?



Analogie avec la vraie vie vraie

- En entreprise, à qui demande-t-on tel ou tel rapport?
- À celui qui a l'information!

En orienté objet, chaque objet est considéré comme une « personne » !

- Ils accomplissent des choses en fonction de l'information qu'ils possèdent
- C'est un peu comme se retrouver dans un dessin animé où tous les objets sont vivants

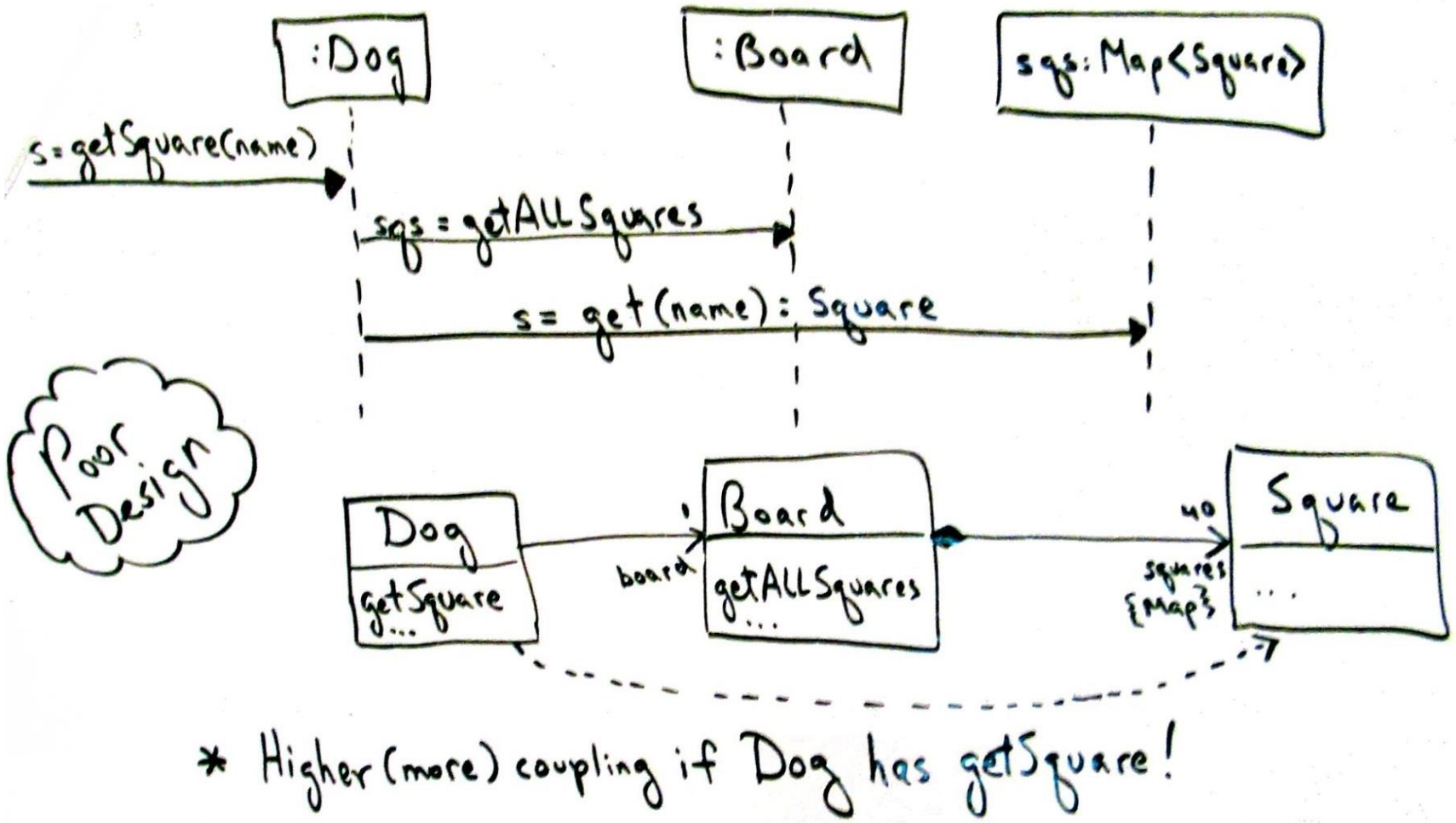
Sale
time ...
getTotal()

SalesLineItem
quantity
getSubtotal()

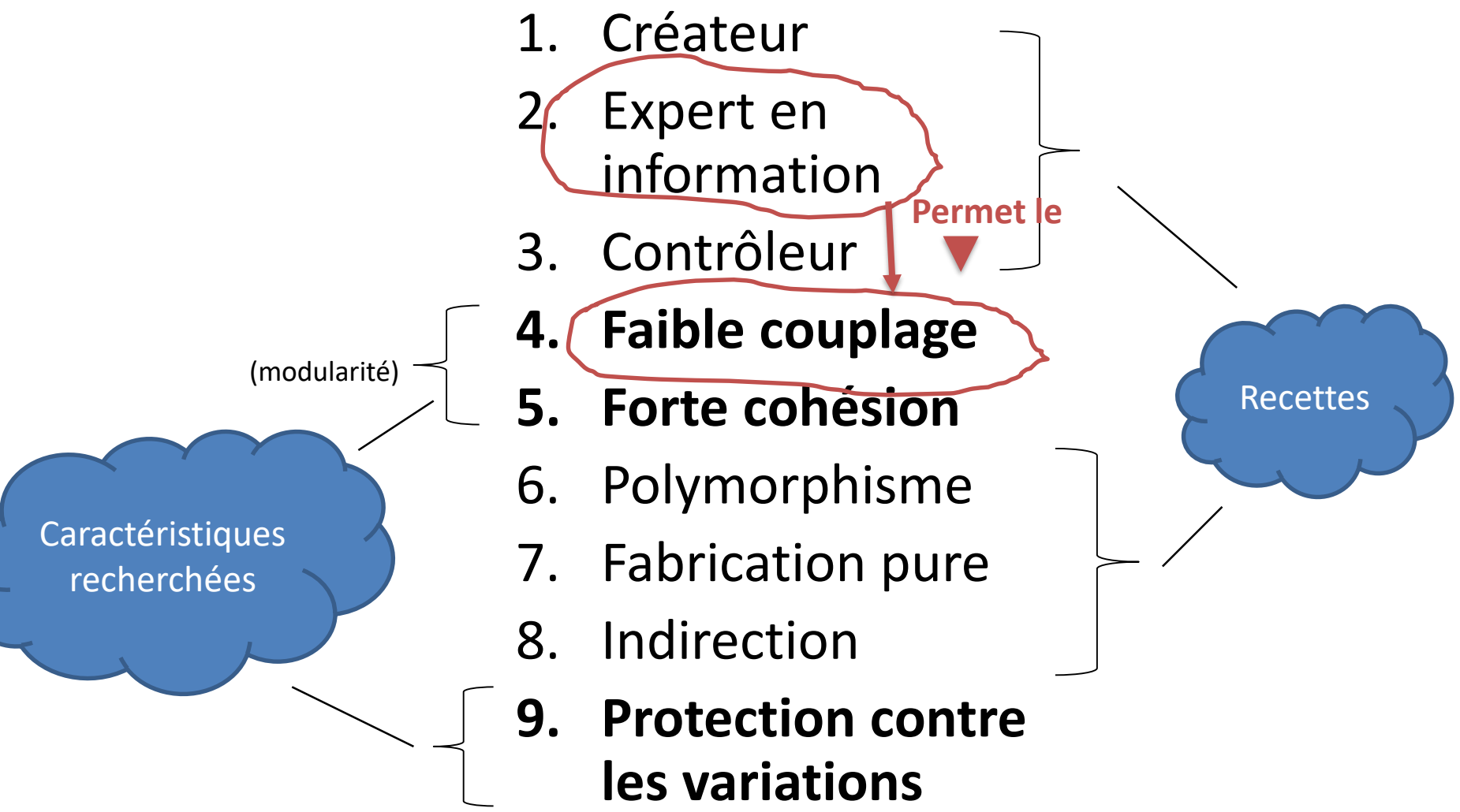
Principe 4: Faible couplage

- **Problème**:
  - Comment réduire l'impact des modifications futures?
- **Solution**:
  - Affecter les responsabilités aux classes de manière à éviter tout couplage inutile entre les classes

Où placer une méthode permettant de retrouver une case à partir de son nom?



Grands principes (GRASP)

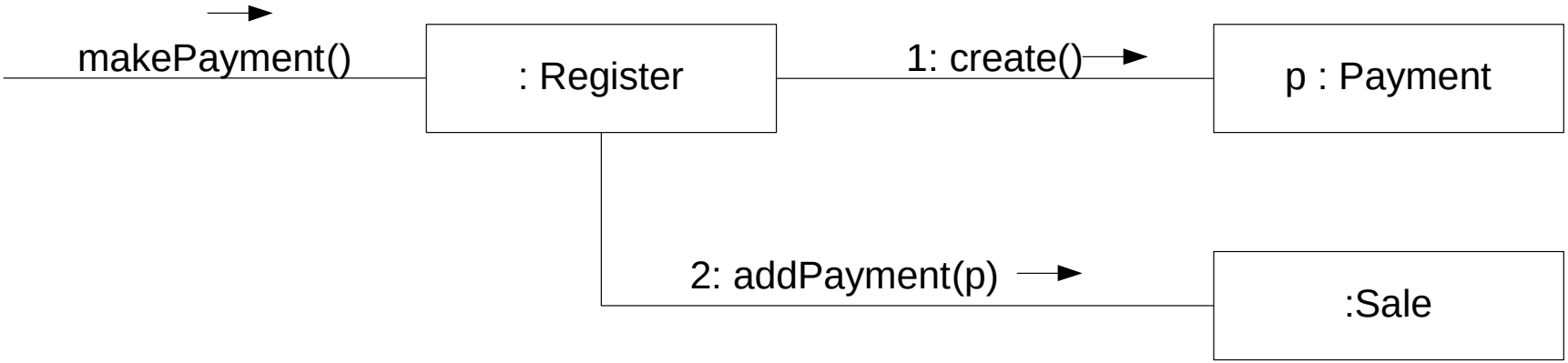


Principe 5: Forte cohésion

- **Problème**:
  - Comment s'assurer que les objets restent compréhensibles et faciles à gérer, et qu'ils contribuent au faible couplage?
- **Solution**:
  - Affecter les responsabilités de manière à ce que la cohésion demeure élevée
- **Attention**:
  - Va parfois à l'encontre d'autres principes

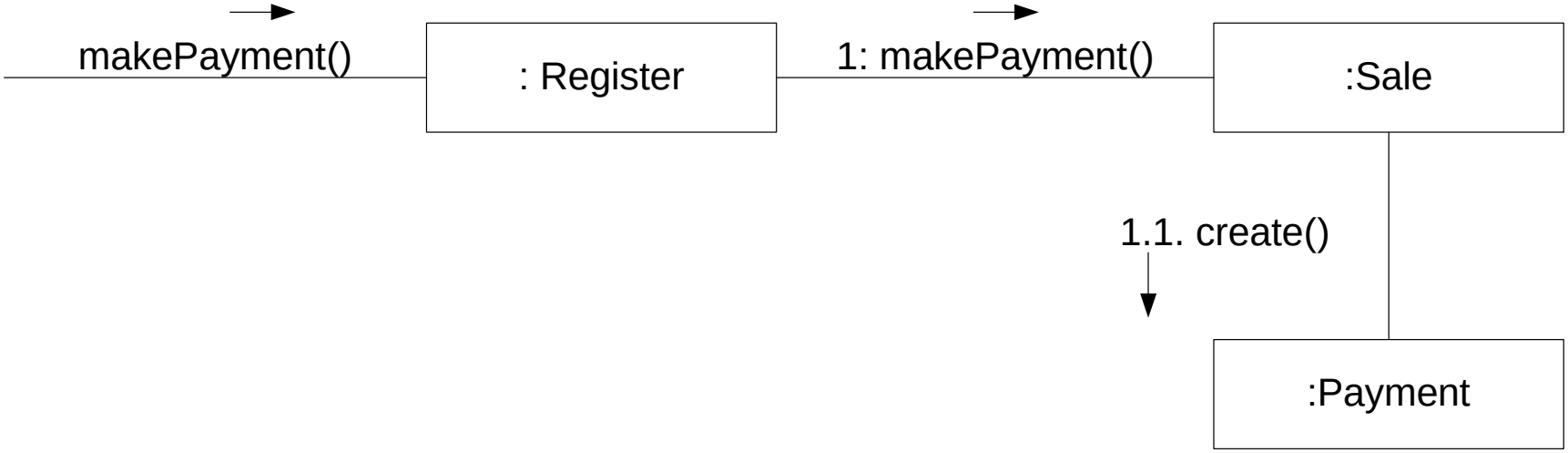
Qui devrait créer l'objet paiement?

- Selon le principe **Créateur**:



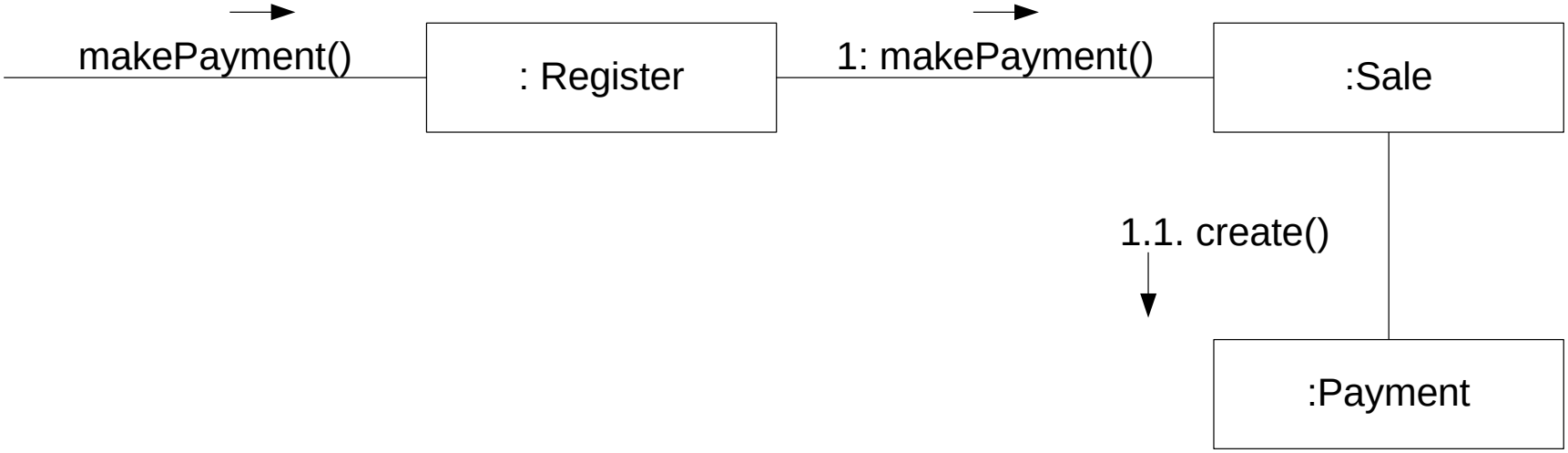
Qui devrait créer l'objet paiement?

- Selon le principe **Faible couplage**:



Qui devrait créer l'objet paiement?

- Selon le principe **Forte cohésion**:



Que faire lorsque des grands principes se contredisent?

- Dans cet exemple:
  - **Couplage** et **cohésion** vont dans le même sens, on aurait tendance à les favoriser (par rapport à **Créateur**)
- En général, garder en tête que ce sont des principes pour nous guider et évaluer nos différentes options. Ce ne sont pas des lois!

Choisir ses batailles

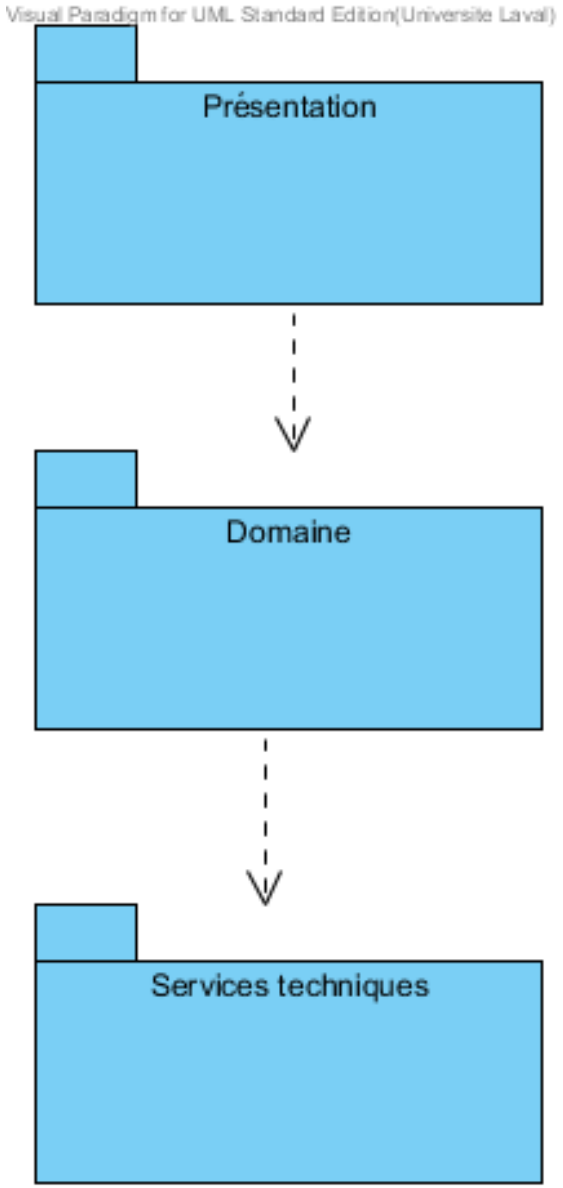
- On ne veut pas nécessairement bannir tout couplage
- On veut bannir le couplage à des éléments qui risquent de d'évoluer, de changer fréquemment.

Principe 3: Contrôleur

- **Problème**:
  - Quel est le premier objet au-delà de la couche présentation qui reçoit les « messages » de l'utilisateur et contrôle l'accès aux objets de la couche du domaine?

- …

Architecture en couche « classique »

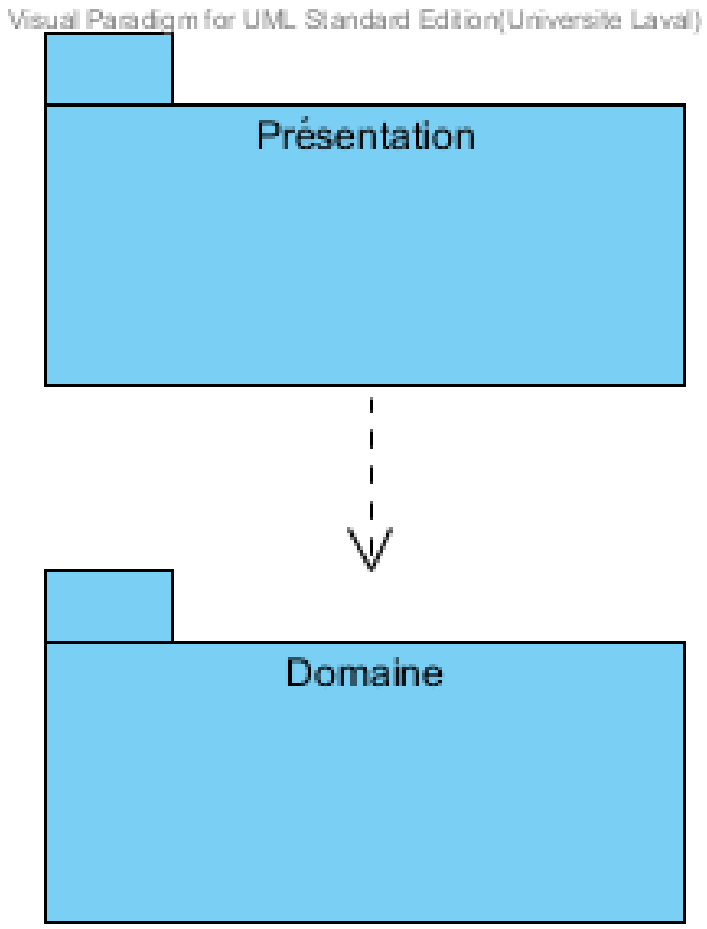


Interface utilisateur

Logique applicative et objets du domaine

Objets à usage général (ex: classes accès à une BD) – généralement indépendant de l'application

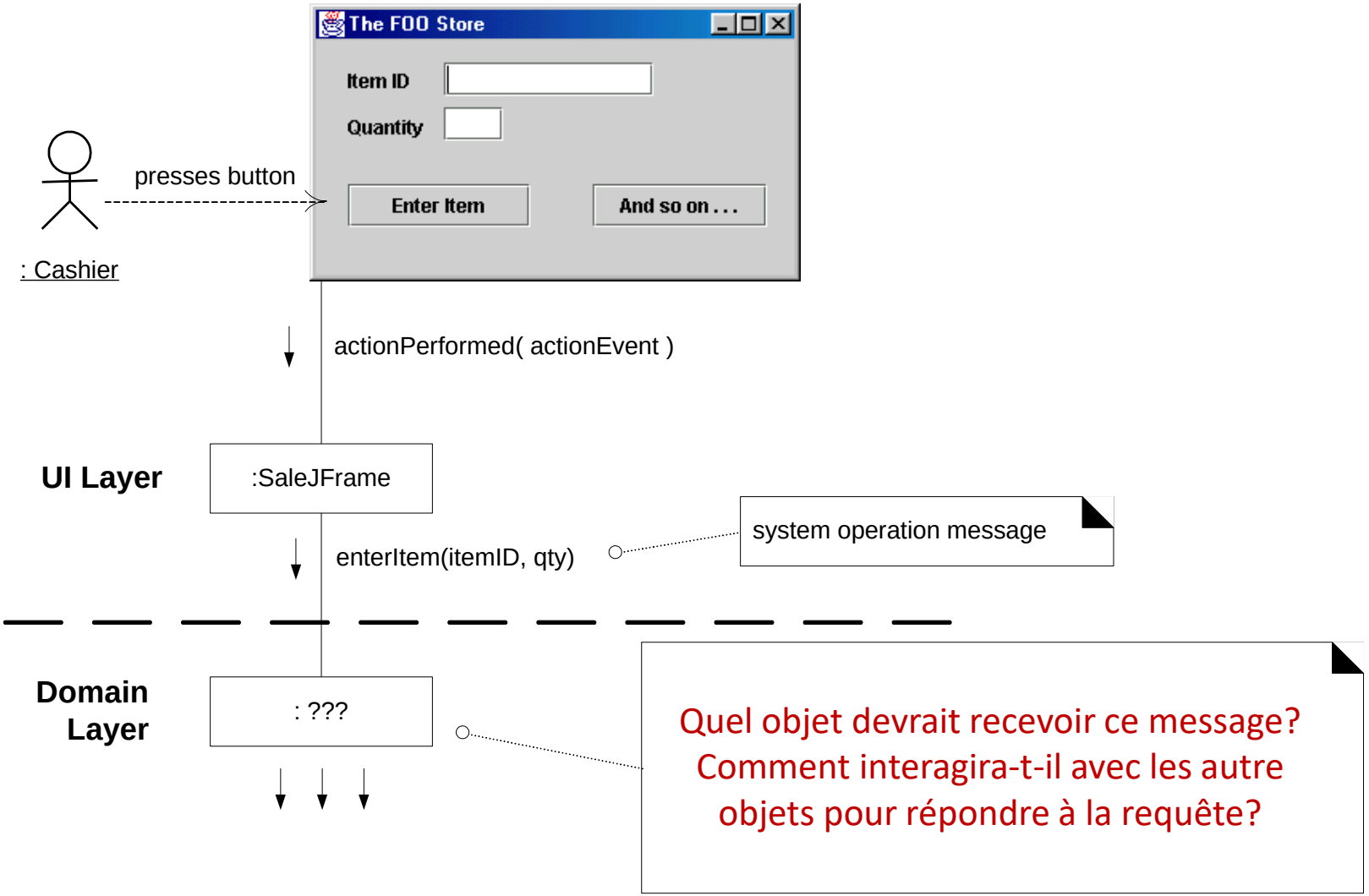
Séparation modèle-vue



Vue

Modèle

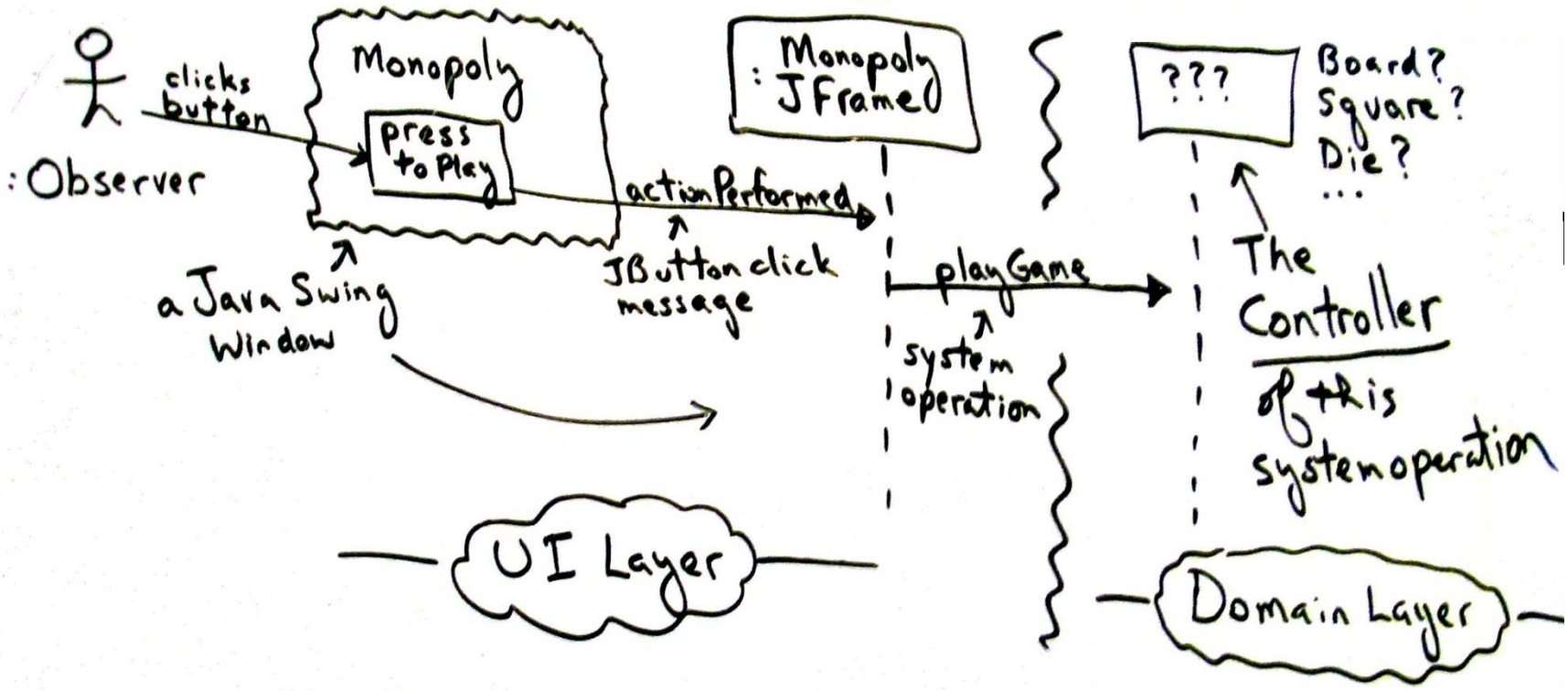
Le problème



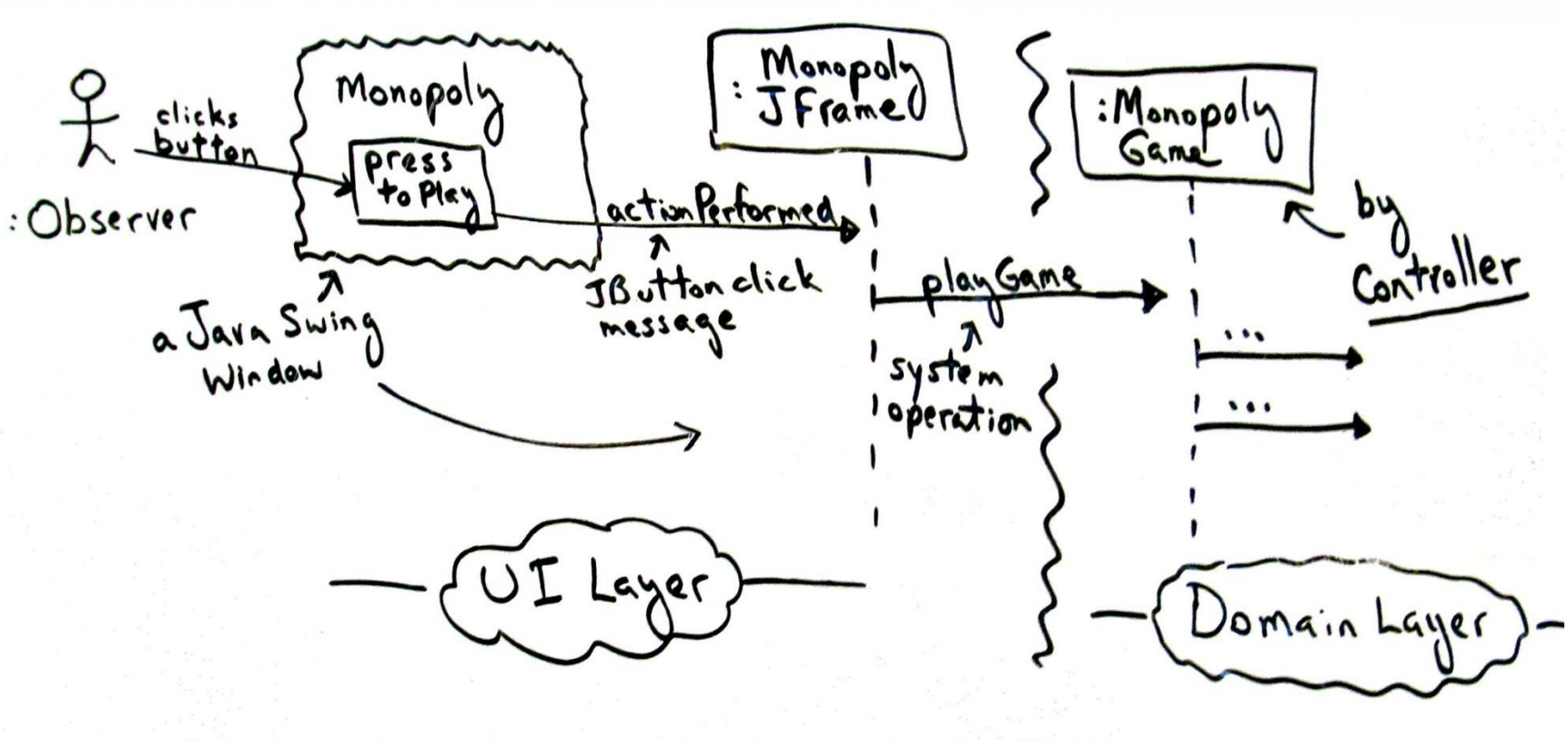
Principe 3: Contrôleur

- **Problème**:
  - Quel est le premier objet au-delà de la couche présentation qui reçoit les « messages » de l'utilisateur et contrôle l'accès aux objets de la couche du domaine?
- **Solution**:
  - Généralement: un objet qui représente le « système global » ou un « objet racine »
  - Plus rarement: un objet qui représente un cas d'utilisation

Exemple Monopoly

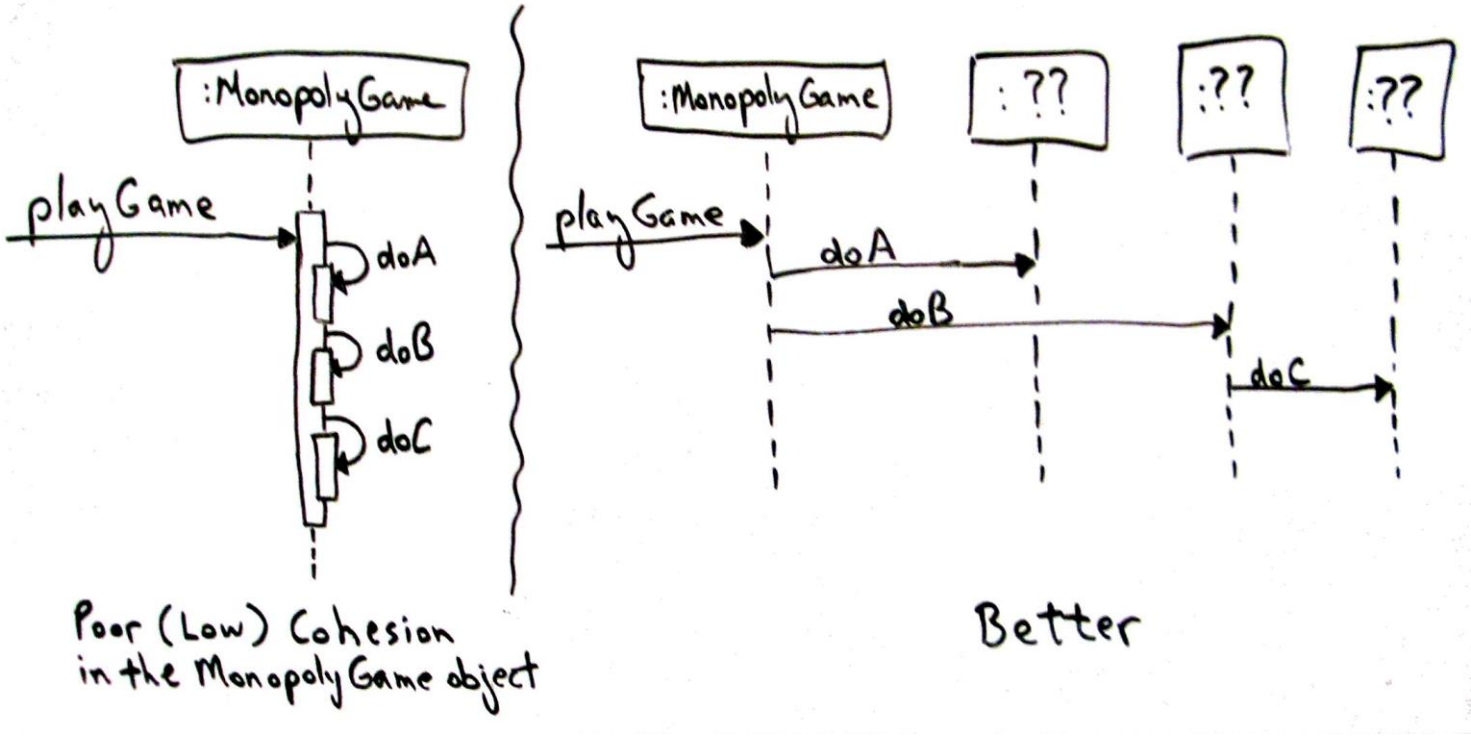


Contrôleur pour Monopoly

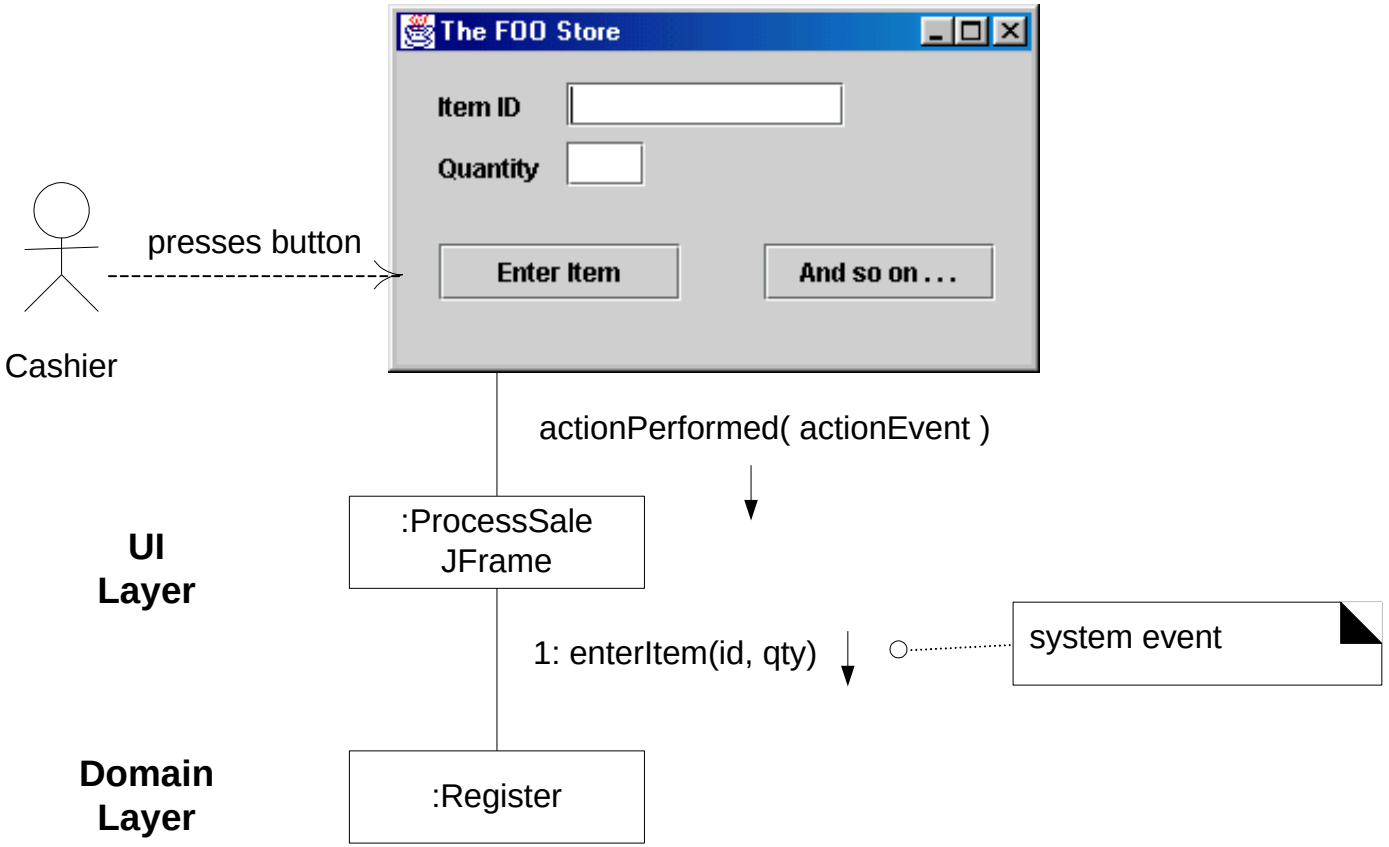


Le rôle du contrôleur

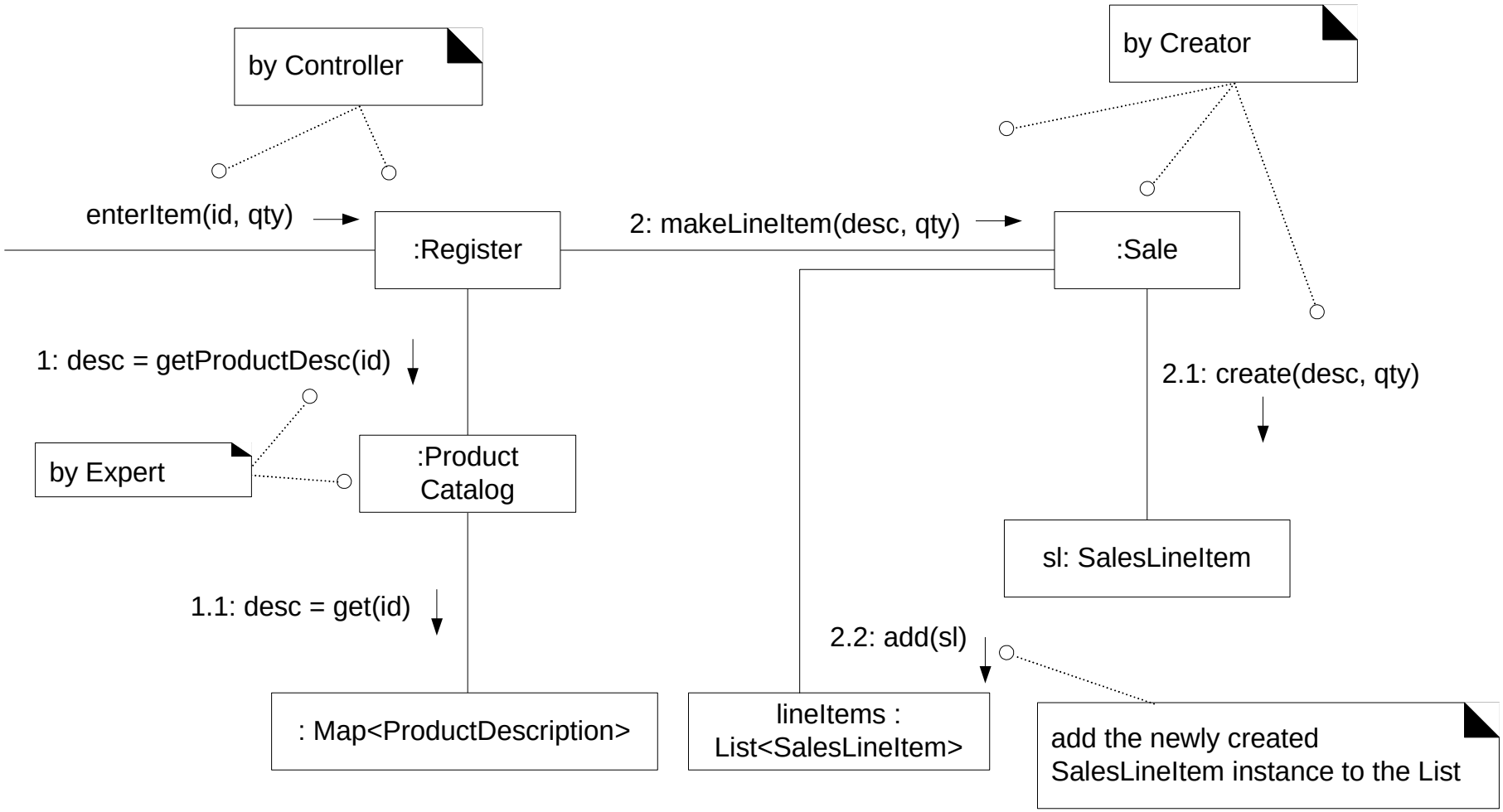
- Le contrôleur délègue les tâches aux autres objets. Il ne fait normalement pas faire grand-chose par lui-même



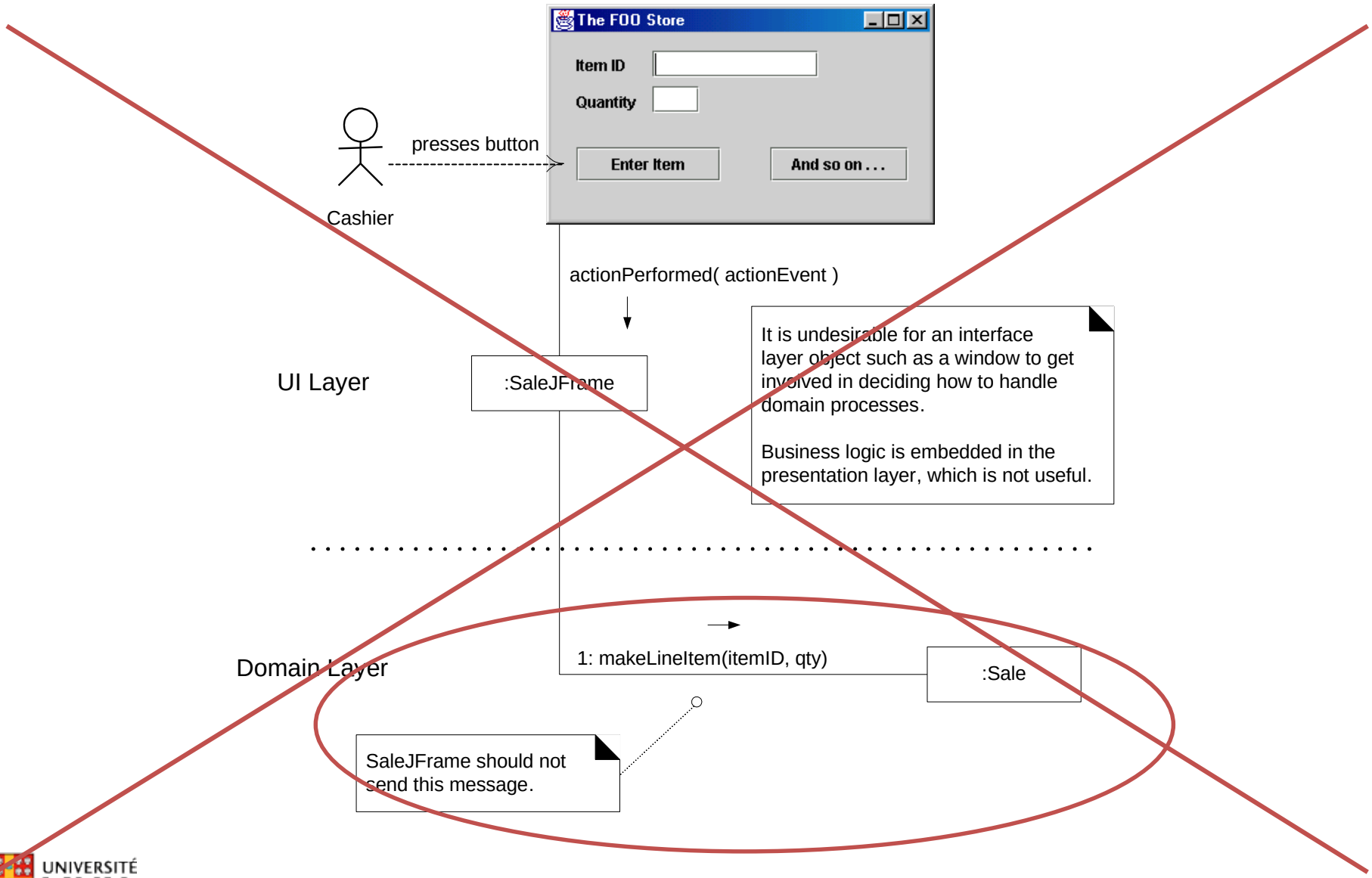
Contrôleur pour NexGenPOS



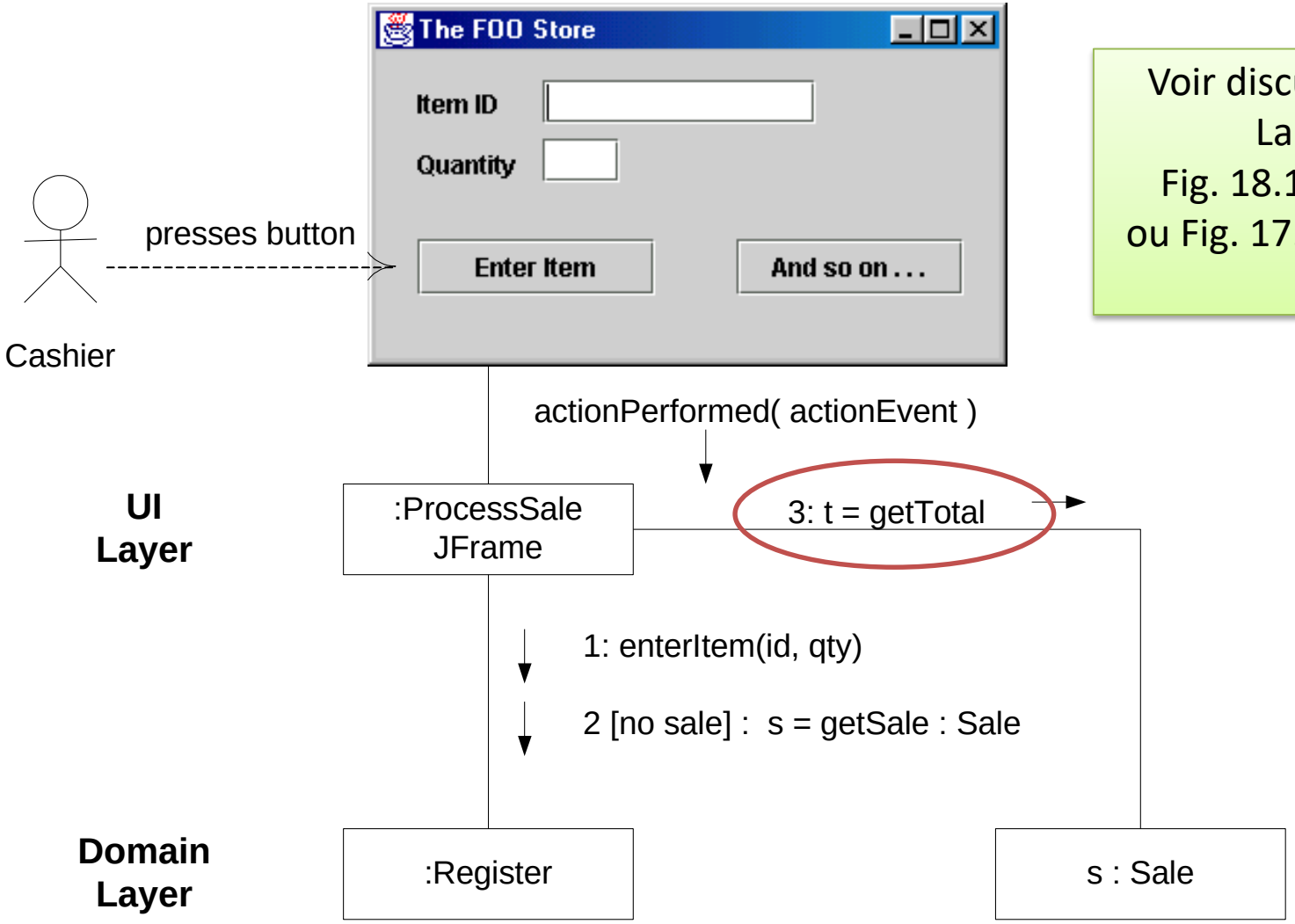
Contrôleur pour NexGenPOS



NexGen: Design alternatif qui engendre une “interface-moteur”



Il est par contre acceptable de « lire » directement les informations de la couche application pour rafraichir l'affichage



Voir discussion dans Larman
Fig. 18.19 (anglais)
ou Fig. 17.18 (français)

Et dans votre projet de session?

- Utilisation d'un contrôleur obligatoire selon l'énoncé du livrable #2 !

- Création d'un objet appelé « contrôleur » ou bien portera-t-il un nom significatif dans le contexte de notre application?
  Il n'y a pas de mauvaise réponse à cette question, mais assurez-vous d'y répondre, d'avoir un contrôleur et de l'identifier clairement!