Réutilisation dans les SI : patrons de conception

M1 MIAGE - SIMA - 2007-2008
Yannick Prié – Lionel Médini
UFR Informatique
Université Claude Bernard Lyon 1

Introduction : réutilisation

- Constante de la conception d'outils en général
  - Ex. : je dois fabriquer quelque chose qui permette à des passagers d'attendre la fin du voyage sur un bateau.
    - Dois-je tout concevoir depuis zéro ? Que puis-je réutiliser ?
- En informatique
  - réutilisation de code
    - sous la forme de composants
      - à acheter / fabriquer
    - sous la forme de frameworks
      - à utiliser en les spécialisant
  - réutilisation de principes de conception
      - à connaître
- Dès que des principes se révèlent pertinents
  - abstraction / réutilisation

Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks

Généralités sur les patterns

- Pattern
  - solution classique à un problème de conception classique dans un contexte donné
- Pattern de conception
  - structure et comportement d'une société de classes
  - description nommée d'un problème et d'une solution
  - avec conseils d'application

Description d’un pattern

- Nom du pattern : un ou deux mots
  - Permet essentiellement d’en parler
  - N’est pas significatif en soi
- Problème : quand appliquer le pattern ?
  - explication du problème et de son contexte
- Solution : description abstraite
  - Élément de conception, relation, responsabilités, collaboration
- Exemple
- Discussion : conseils sur la façon de l’appliquer
  - Avantages, inconvénients, conseils d’implémentation, variantes…

Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks

Conception pilotée par les responsabilités

- Métaphore
  - communauté d'objets responsables qui collaborent (*cf.* humains) dans un projet (rôles)
- Principe
  - 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

Deux catégories de responsabilités pour les objets

- Savoir
  - connaître les données privées encapsulées
  - connaître les objets connexes
  - connaître les attributs à calculer ou dériver
- Faire
  - faire quelque chose soi-même (ex. créer un autre objet, effectuer un calcul)
  - déclencher une action d'un autre objet
  - contrôler et coordonner les activités d'autres objets

Exemples (bibliothèque)

- Savoir
  - *Livre* est responsable de la connaissance de son numéro *ISBN*
  - *Abonné* est responsable de savoir s'il lui reste la possibilité d'emprunter des livres
- Faire
  - *Abonné* est responsable de la vérification du retard sur les livres prêtés

GRASP

- *General Responsability Assignment Software Patterns*
- Ensemble de patterns généraux d'affectation de responsabilités pour aider à la conception orientée-objet
  - raisonner objet de façon méthodique, rationnelle, explicable
- Utile pour l'analyse et la conception
  - réalisation d'interactions avec des objets
- Référence : Larman 2004

9 patterns GRASP

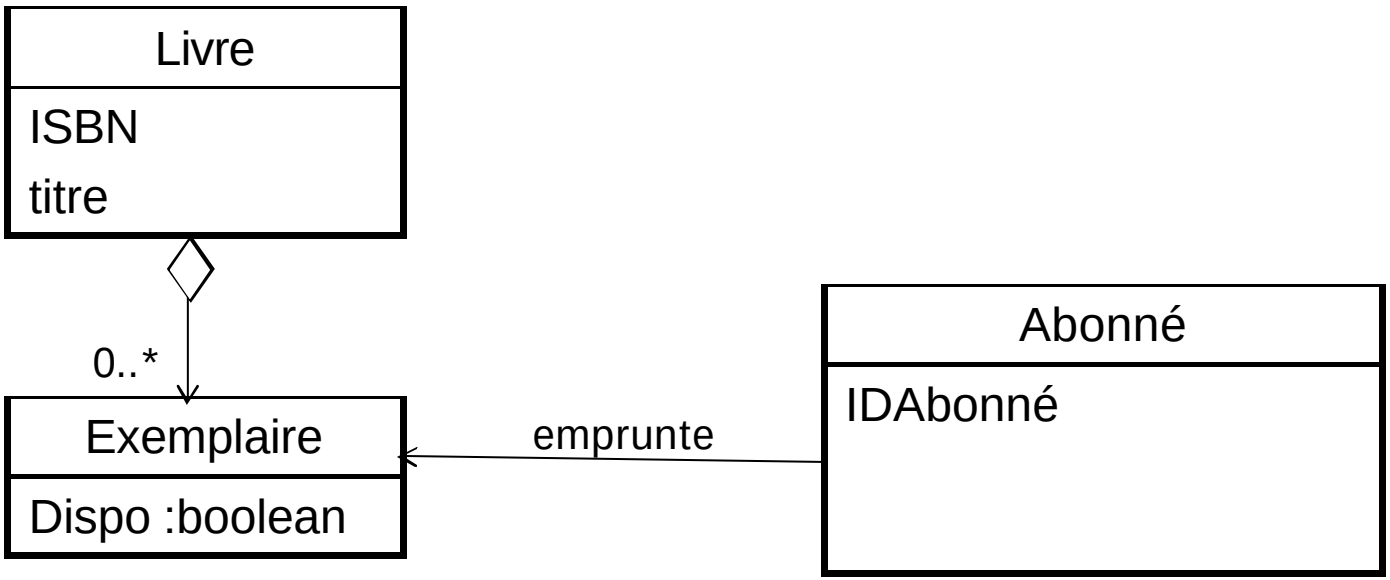
1. Créateur
2. Expert en information
3. Faible couplage
4. Contrôleur
5. Forte cohésion
6. Polymorphisme
7. Fabrication pure
8. Indirection
9. Protection des variations

Expert (GRASP)

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

Expert : exemple

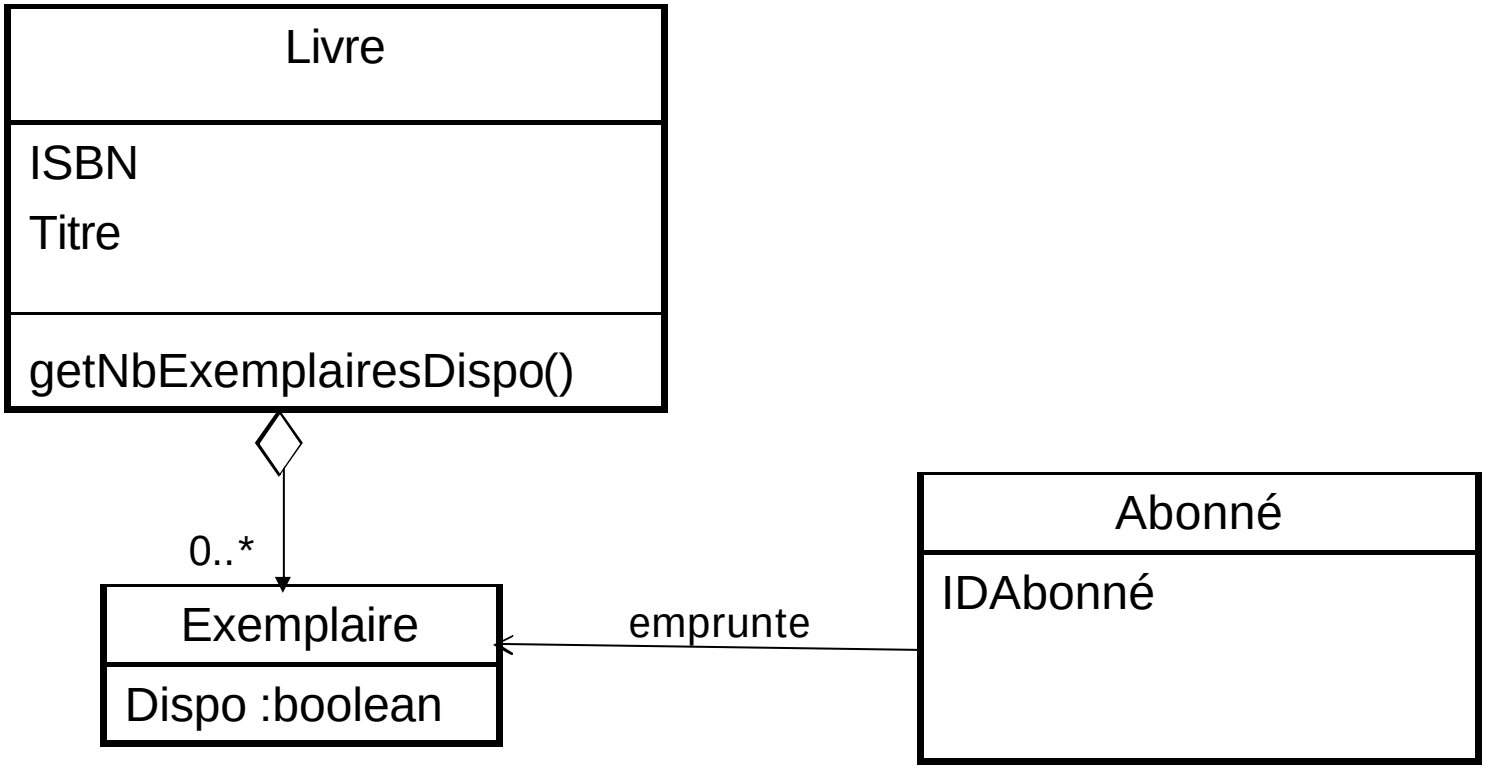
- Bibliothèque : qui doit avoir la responsabilité de connaître le nombre d'exemplaires disponibles ?



Expert : 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

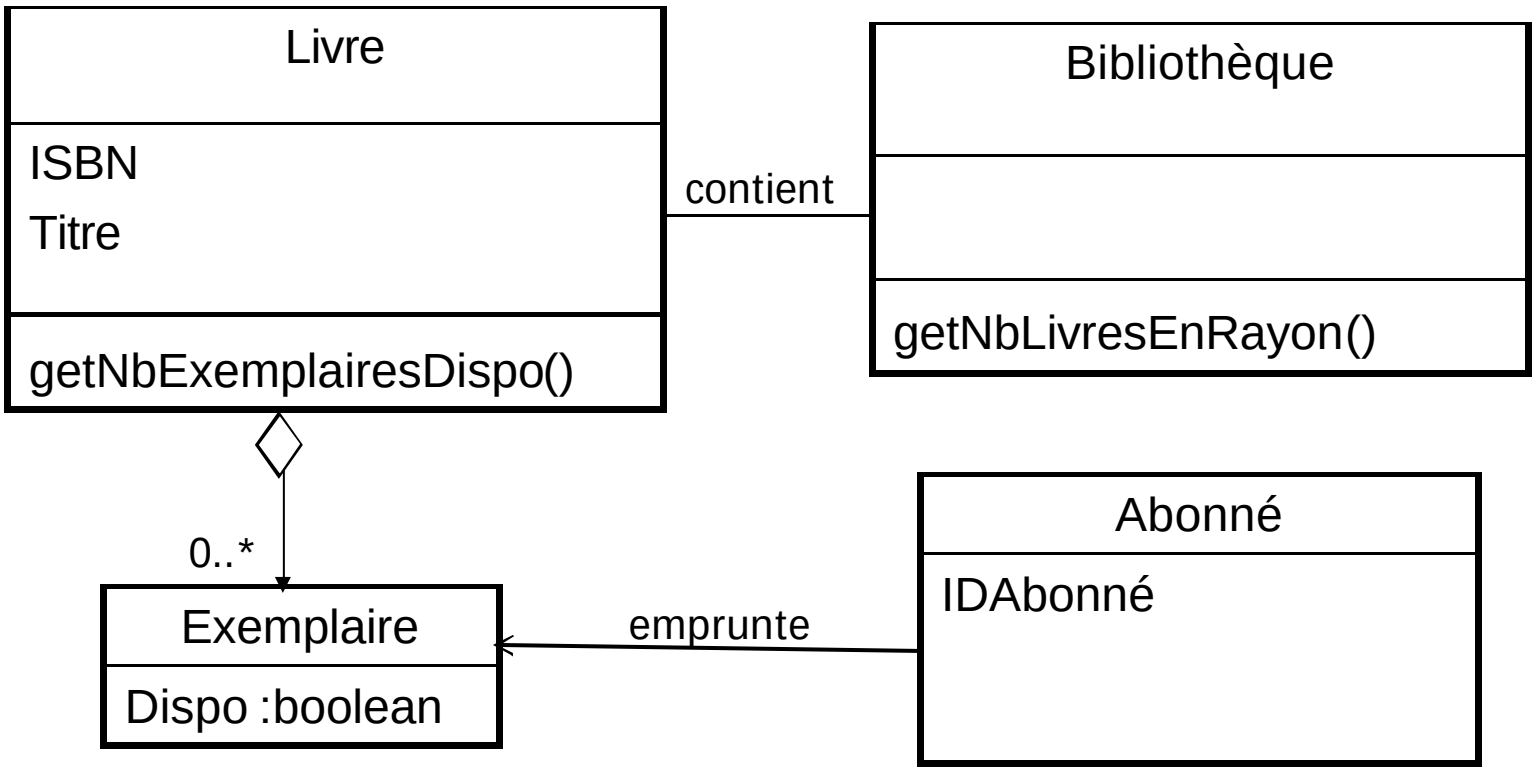
Expert : exemple



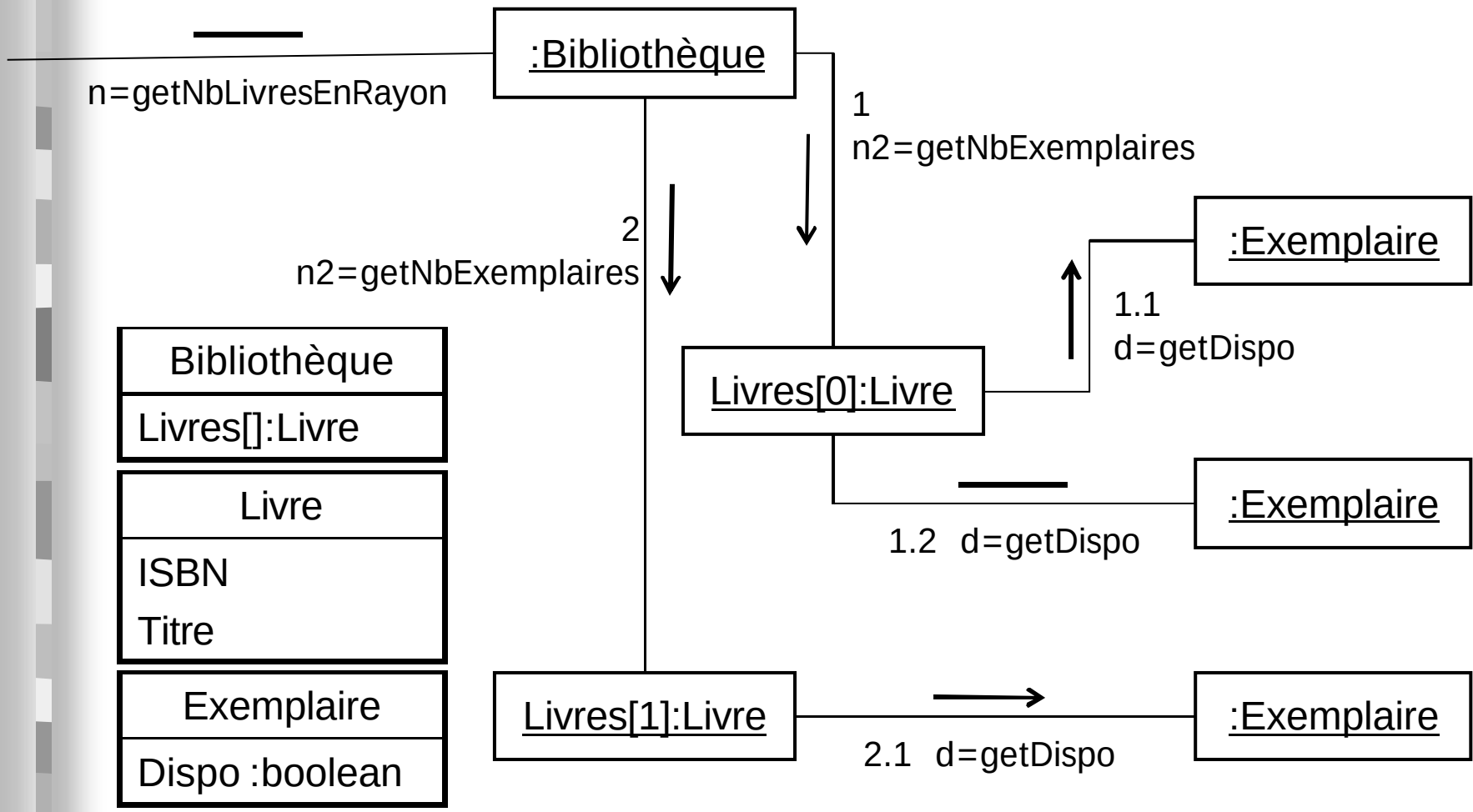
Expert (suite)

- Tâche complexe
  - Que faire quand l'accomplissement d'une responsabilité nécessite de l'information répartie entre différents objets ?
- Solution : décomposer la tâche
  - Déterminer des « experts partiels »
  - Leur attribuer les responsabilités correspondant aux sous-tâches
  - Faire jouer la collaboration pour réaliser la tâche globale

Expert : exemple (suite)



Expert : exemple (suite)



Expert : discussion

- Modèle UML approprié
  - Quel modèle UML utiliser pour cette analyse ?
    - Domaine : classes du monde réel
    - Conception : classes logicielles
  - Solution :
    - Si l'information est dans les classes de conception, les utiliser
    - Sinon essayer d'utiliser le modèle du domaine pour créer des classes de conception et déterminer l'expert en information
- Diagrammes UML utiles
  - Diagrammes de classes : information encapsulée
  - Diagrammes de communication + diagrammes de classes partiel : tâches complexes

Expert : discussion

- Avantages
  - Conforme aux principes de base en OO
    - encapsulation
    - collaboration
  - Définitions de classes légères, faciles à comprendre à maintenir, à réutiliser
  - Comportement distribué entre les classes qui ont l'information nécessaire

  ➔Systèmes robustes et maintenables

Expert : discussion

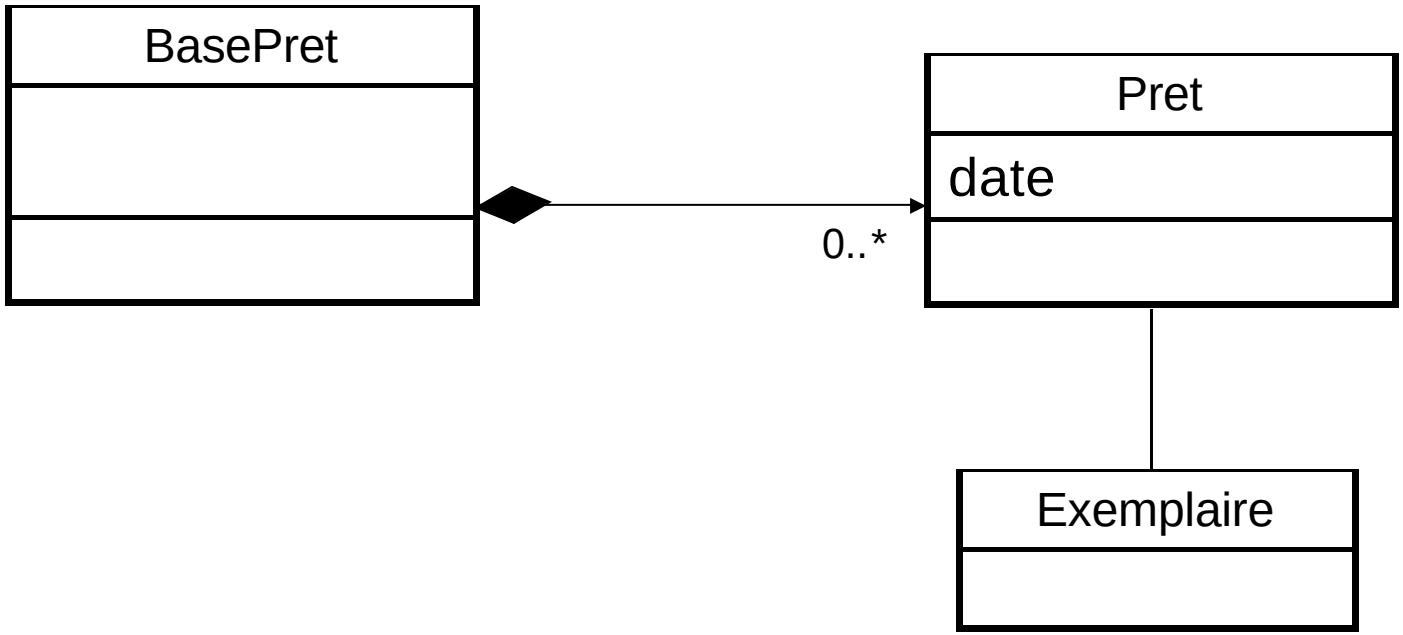
- Le plus utilisé de tous les patterns d'attribution de responsabilités
- Autres noms (AKA - Also Known As)
  - Mettre les responsabilités avec les données
  - Qui sait, fait
  - Faire soi-même
- Patterns liés (voir plus loin)
  - *Faible couplage*
  - *Forte cohésion*

Créateur (GRASP)

- 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 A si une - ou plusieurs - de ces 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 : exemple

- Bibliothèque : qui doit être responsable de la création de *Pret ?*
- *BasePret* contient des *Pret* : elle doit les créer.



Créateur : discussion

- Guide pour attribuer une responsabilité pour la création d'objets
  - 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*

Faible couplage (GRASP)

- Problème
  - Comment minimiser les dépendances ?
  - Comment réduire l'impact des changements ?
  - Comment améliorer la réutilisabilité ?
- Solution
  - Affecter les responsabilités des classes de sorte que le *couplage* reste faible
  - Appliquer ce principe lors de l'évaluation des solutions possibles

Couplage

- Définition
  - Mesure du degré auquel un élément est lié à un autre, en a connaissance ou en dépend
- Exemples classiques de couplage 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

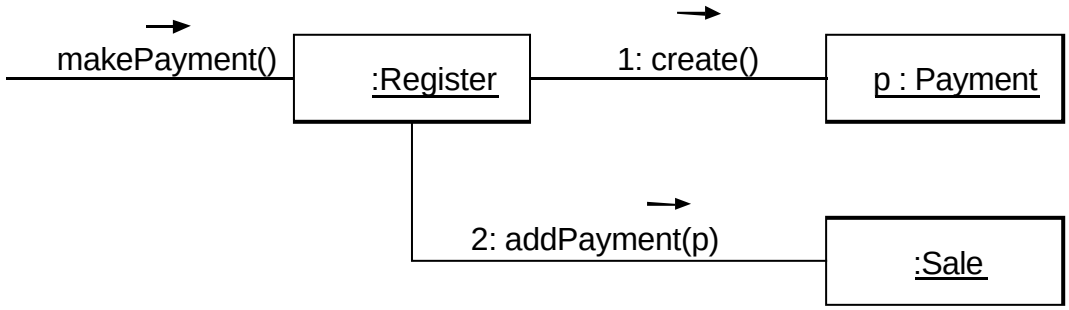
Faible couplage (suite)

- 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

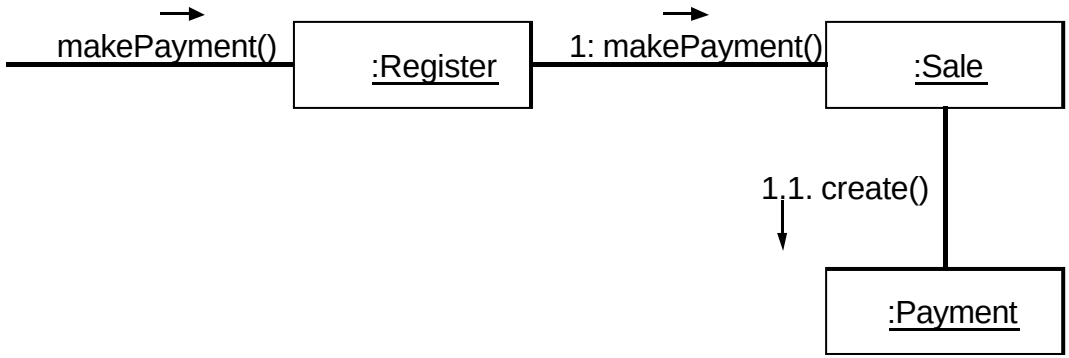
Faible couplage (suite)

- Principe général
  - Les classes, très génériques et très réutilisables par nature, doivent avoir un faible couplage
- Mise en œuvre
  - déterminer plusieurs possibilités pour l'affectation des responsabilités
  - comparer leurs niveaux de couplage en termes de
    - Nombre de relations entre les classes
    - Nombre de paramètres circulant dans l'appel des méthodes
    - Fréquence des messages
    - …

Faible couplage : exemple

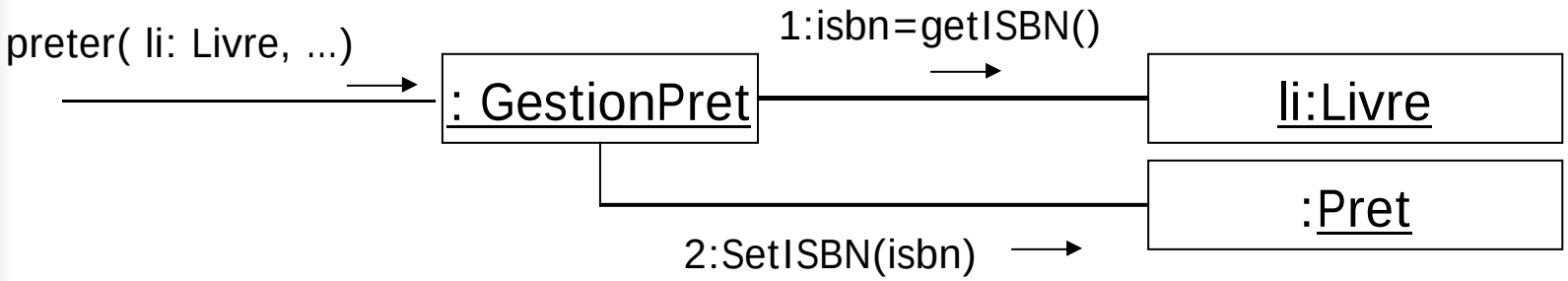


Que choisir ?

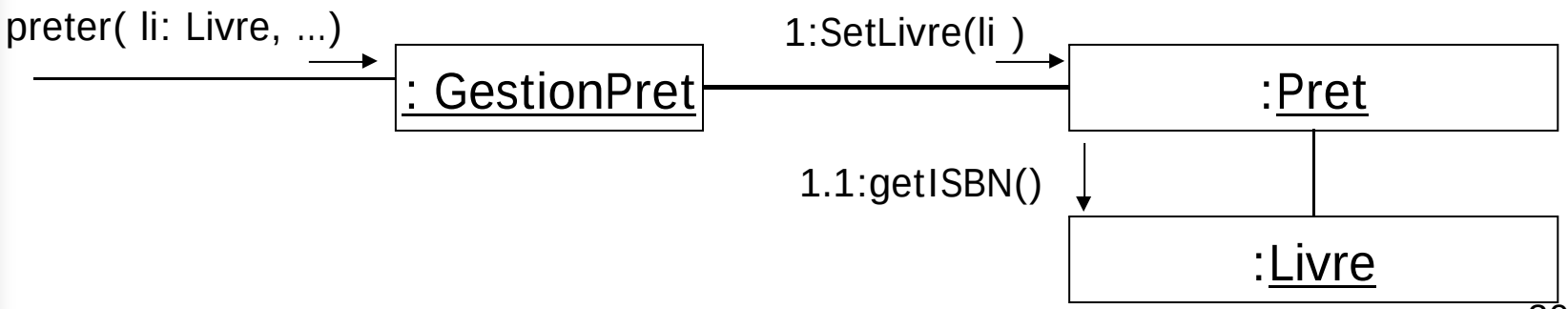


Faible couplage : autre exemple

- Pour l'application de bibliothèque, il faut mettre l'ISBN d'un Exemplaire dans le Prêt.



Que choisir ?



Faible couplage : discussion

- Un principe à garder en tête pour toutes les décisions de conception
- Ne doit pas être considéré indépendamment d'autres patterns comme *Expert* et *Forte cohésion*
  - en général, *Expert* soutient *Faible couplage*
- Pas de mesure absolue de quand un couplage est trop fort
- Un fort couplage n'est pas dramatique avec des éléments très stables
  - java.util par exemple

Faible couplage : discussion (suite)

- 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

  ➔ mauvaise conception qui va à l'encontre des principes OO (collaboration d'objets, forte cohésion)
- Bref
  - un couplage modéré est nécessaire et normal pour créer des systèmes OO

Forte cohésion (GRASP)

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

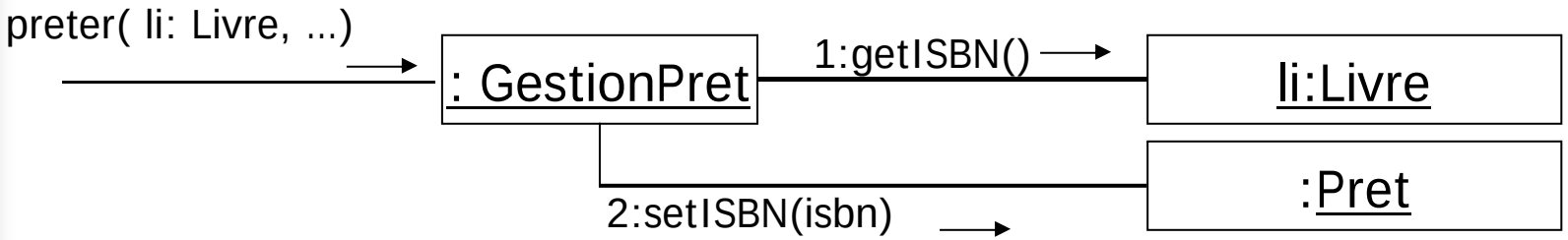
Cohésion
(ou cohésion fonctionnelle)

- Définition
  - mesure informelle de l'étroitesse des liens et de la spécialisation des responsabilités d'un élément (d'une classe)
    - relations fonctionnelles entre les différentes opérations effectuées par un élément
    - volume de travail réalisé par un élément
  - Une classe qui est fortement cohésive
    - a des responsabilités étroitement liées les unes aux autres
    - n'effectue pas un travail gigantesque
- Un test
  - décrire une classe avec une seule phrase

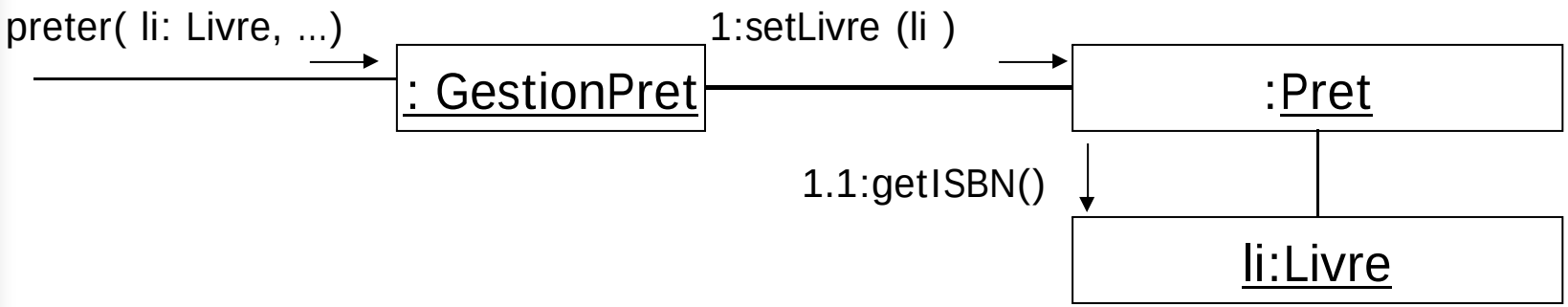
Forte cohésion (suite)

- Problèmes des classes à faible cohésion
  - Elle effectuent
    - trop de tâches
    - des tâches sans lien entre elles
  - Elles sont
    - difficiles à comprendre
    - difficiles à réutiliser
    - difficiles à maintenir
    - fragiles, constamment affectées par le changement
- Bénéfices de la forte cohésion : ...

Forte cohésion : exemple



- On rend GestionPret partiellement responsable de la mise en place des ISBN
- *GestionPret* sera responsable de beaucoup d'autres fonctions



- On délègue la responsabilité de mettre l'ISBN au prêt

Forte cohésion : discussion

- Forte cohésion va en général de paire avec Faible couplage
- C'est un pattern d'évaluation à garder en tête pendant toute la conception
  - Permet l'évaluation élément par élément (contrairement à *Faible couplage*)

Forte cohésion : discussion

- Citations
  - [Booch] : Il existe une cohésion fonctionnelle quand les éléments d'un composant (*eg.* les classes)
    « travaillent tous ensemble pour fournir un comportement bien délimité »
  - [Booch] : « la modularité est la propriété d'un système qui a été décomposé en un ensemble de modules cohésifs et peu couplés »

Contrôleur (GRASP)

- Problème
  - Quel est le premier objet au delà de l'IHM
    qui reçoit et coordonne (contrôle)
    une opération système (événement majeur entrant dans le système) ?
- Solution
  - Affecter cette responsabilité à une classe qui représente
    - Soit le système global, un sous-système majeur ou un équipement sur lequel le logiciel s'exécute
      ➔ *contrôleur Façade* ou variantes
    - Soit un scénario de cas d'utilisation dans lequel l'événement système se produit
      ➔ *contrôleur de CU* ou *contrôleur de session*

Contrôleur (GRASP)

- Principes à bien comprendre : idéalement
  - un contrôleur est un objet qui ne fait rien
    - *reçoit* les événements système
    - *délègue* aux objets dont la responsabilité est de les traiter
  - il se limite aux tâches de contrôle et de coordination
    - vérification de la séquence des événements système
    - appel des méthodes *ad hoc* des autres objets
- Règle d'or
  - Les opérations système des CU sont les messages initiaux qui parviennent au contrôleur dans les diagrammes d'interaction de la couche domaine

Contrôleur (GRASP)

- Mise en œuvre
  - Au cours de la détermination du comportement du système (besoins, CU, DSS), les opérations système sont déterminées et attribuées à une classe générale *Système*
  - À l'analyse/conception, des classes contrôleur sont mises en place pour prendre en charge ces opérations

Contrôleur : exemple

- Pour la gestion d'une bibliothèque, qui doit être contrôleur pour l'opération système emprunter ?
- Deux possibilités
  - Le contrôleur représente le système global

    :ControleurBiblio
  - Le contrôleur ne gère que les opérations système liées au cas d'utilisation emprunter

    :GestionPret

Bibliothèque
preterLivre() enregistrerMembre() …..

- La décision d'utiliser l'une ou l'autre solution dépend d'autres facteurs liés à la cohésion et au couplage

Contrôleur Façade

- Représente tout le système
  - exemples : ProductController, RetailInformationSystem, Switch, Router, NetworkInterfaceCard, SwitchFabric, *etc*.
- À utiliser quand
  - il y a peu d'événements système
  - il n'est pas possible de rediriger les événements systèmes à un contrôleur alternatif

Contrôleur de cas d'utilisation

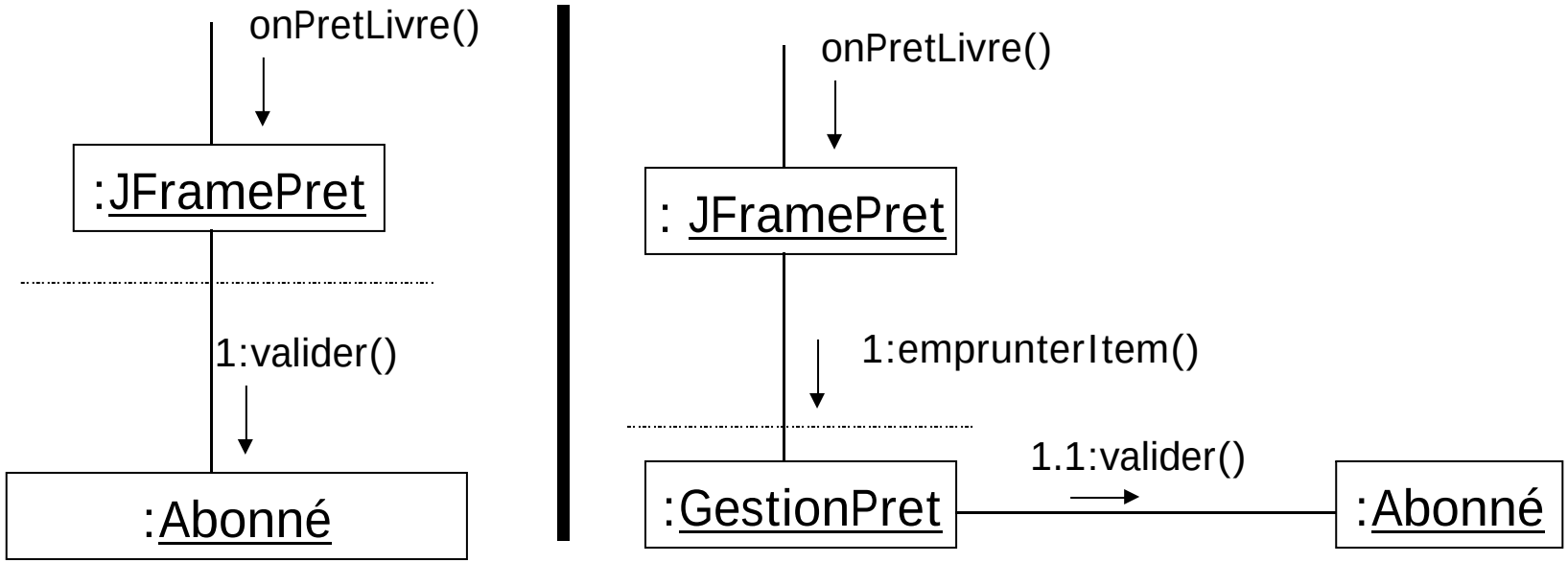
- Un contrôleur différent pour chaque cas d'utilisation
  - Commun à tous les événements d'un cas d'utilisation
  - Permet de connaître et d'analyser la séquence d'événements système et l'état de chaque scénario
- À utiliser quand
  - les autres choix amènent à un fort couplage ou à une cohésion faible (contrôleur *trop chargé - bloated*)
  - il y a de nombreux événements système qui appartiennent à plusieurs processus
    - ➔Permet de répartir la gestion entre des classes distinctes et faciles à gérer
- Contrôleur artificiel : ce n'est pas un objet du domaine

Contrôleur trop chargé (pas bon)

- Pas de focus, prend en charge de nombreux domaines de responsabilité
  - un seul contrôleur reçoit tous les événements système
  - le contrôleur effectue la majorité des tâches nécessaires pour répondre aux événements système
    - 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
- Solution
  - ajouter des contrôleurs
  - concevoir des contrôleurs dont la priorité est de déléguer

Remarque : couche présentation

- Les objets d'interface 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 ou application



Contrôleur : discussion

- Avantages
  - Meilleur potentiel de réutilisation
    - permet de réaliser des composants d'interface *enfichables*
      - « porte d'entrée » des objets de la couche domaine
      - la rend indépendante des types d'interface (Web, client riche simulateur de test…)
    - ➔Niveau d'indirection matérialisant la séparation Modèle-vu
    - ➔Brique de base pour une conception modulaire
  - Meilleure « architecturation » des CU
- Patterns liés
  - Indirection, Couches, Façade, Fabrication pure, Commande

Polymorphisme (GRASP)

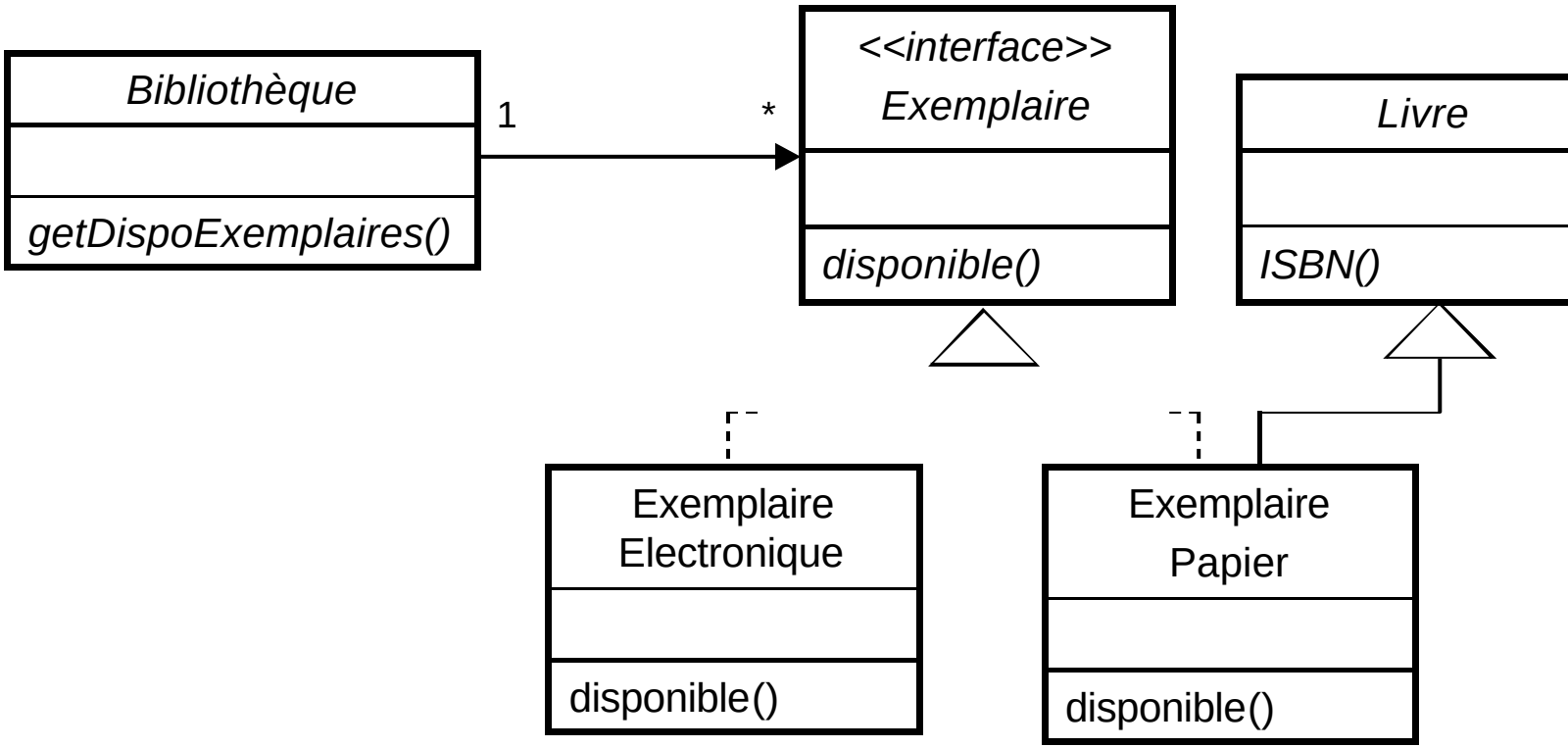
- Problème
  - Comment gérer des alternatives dépendantes des types ?
  - Comment créer des composants logiciels « enfichables » ?
- Solution
  - Affecter les responsabilités aux types (classes) pour lesquels le comportement varie
  - Utiliser des opérations *polymorphes*
- Polymorphisme
  - Donner le même nom à des services dans différents objets
  - Lier le « client » à un supertype commun

Polymorphisme (GRASP)

- Principe
  - Tirer avantage de l'approche OO en sous-classant les opérations dans des types dérivés de l'Expert en information
    - L'opération nécessite à la fois des informations et un comportement particuliers
- Mise en œuvre
  - Utiliser des classes abstraites
    - Pour définir les autres comportements communs
    - S'il n'y a pas de contre-indication (héritage multiple)
  - Utiliser des interfaces
    - Pour spécifier les opérations polymorphes
  - Utiliser les deux (CA implémentant des interfaces)
    - Fournit un point dévolution pour d'éventuels cas particuliers futur

Polymorphisme : exemple

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



Polymorphisme : discussion

- Autre solution (mauvaise)
  - Utiliser une logique conditionnelle (test sur le type d'un objet) au niveau du client
    - Nécessite de connaître toutes les variations de type
    - Augmente le couplage
- Avantages du polymorphisme
  - Évolutivité
    - Points d'extension requis par les nouvelles variantes faciles à ajouter (nouvelle sous-classe)
  - Stabilité du client
    - Introduire de nouvelles implémentations n'affecte pas les clients
- Patterns liés
  - Protection des variations, Faible couplage

Fabrication pure (GRASP)

- Problème
  - Que faire
    - pour respecter le Faible couplage et la Forte cohésion
    - quand aucun concept du monde réel (objet du domaine) n'offre d solution satisfaisante ?
- Solution
  - Affecter un ensemble fortement cohésif à une classe artificiell ou de commodité, qui ne représente pas un concept du domaine
    - entité fabriquée de toutes pièces

Fabrication pure (GRASP)

- Exemple typique : quand utiliser l'Expert en information
  - lui attribuerait trop de responsabilités (contrarie Forte cohésion
  - le lierait à beaucoup d'autres objets (contrarie Faible couplage
- Mise en œuvre
  - Déterminer les fonctionnalités « annexes » de l'Expert en information
  - Les regrouper dans des objets
    - aux responsabilités limitées (fortement cohésifs)
    - aussi génériques que possible (réutilisables)
  - Nommer ces objets
    - pour permettre d'identifier leurs responsabilités fonctionnelles
    - en utilisant si possible la terminologie du domaine

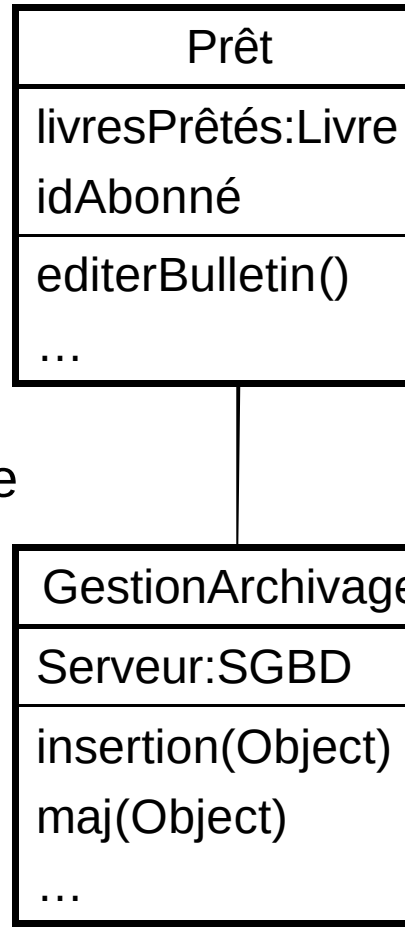
Fabrication pure : exemple

- Problème
  - les instances de *Prêt* doivent être enregistrées dans une BD
- Solution initiale (d'après Expert)
  - Prêt a cette responsabilité
  - cela nécessite
    - un grand nombre d'opérations de BD
      ➔ Prêt devient donc non cohésif
    - de lier Prêt à une BD
      ➔le couplage augmente pour Prêt

Prêt
livresPrêtés:Livre idAbonné Serveur:SGBD
editerBulletin() insertionBD(Object) majBD(Object) …

Fabrication pure : exemple (suite)

- Constat
  - l'enregistrement d'objets dans une BD est une tâche générique utilisable par de nombreux objets
    - pas de réutilisation, beaucoup de duplication
- Solution avec Fabrication pure
  - créer une classe artificielle GestionArchivage
- Avantages
  - Gains pour Prêt
    - Forte cohésion et Faible couplage
  - Conception de GestionArchivage « propre »
    - relativement cohésif, générique et réutilisable



Fabrication pure : discussion

- Choix des objets pour la conception
  - Décomposition représentationnelle (objets du domaine)
    - Conforme au principe de base de l'OO : réduire le décalage des représentations entre le réel et le modèle
  - Décomposition comportementale (Fabrication pure)
    - sorte d'objet « centré-fonction » qui rend des services transverses dans un projet (POA)

➔Ne pas abuser des Fabrications pures

Fabrication pure : discussion

- Règle d'or
  - Concevoir des objets Fabrication pure en pensant à leur réutilisabilité
    - s'assurer qu'ils ont des responsabilités limitées et cohésives
- Avantages
  - Supporte Faible couplage et Forte cohésion
  - Améliore la réutilisabilité
- Patterns liés
  - Faible couplage, Forte cohésion, Adaptateur, Observateur, Visiteur
- Paradigme lié
  - Programmation Orientée Aspects

Indirection (GRASP)

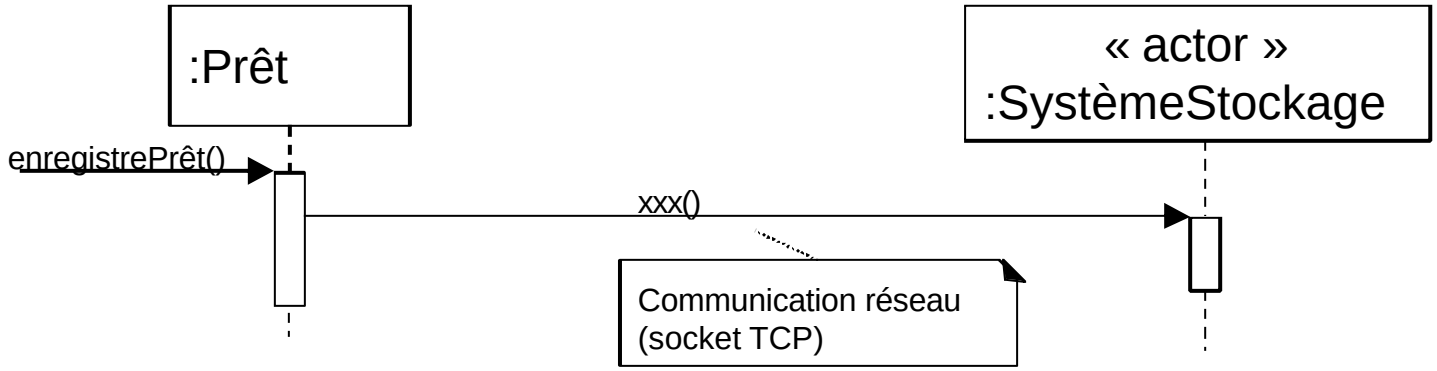
- Problème
  - Où affecter une responsabilité pour éviter le couplage entre deux entités (ou plus)
    - de façon à diminuer le couplage (objets dans deux couches différentes)
    - de façon à favoriser la réutilisabilité (utilisation d'une API externe) ?
- Solution
  - Créer un objet qui sert d'intermédiaire entre d'autres composants ou services
    - l'intermédiaire crée une *indirection* entre les composants
    - l'intermédiaire évite de les coupler directement

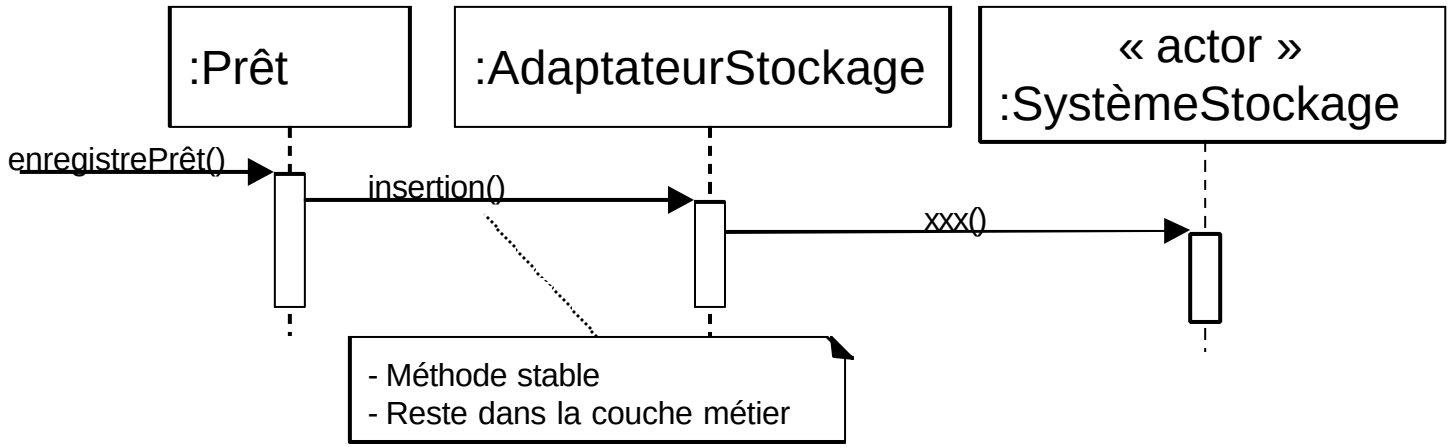
Indirection (GRASP)

- Utilité
  - Réaliser des adaptateurs, façades, etc. (pattern Protection des variations) qui s'interfacent avec des systèmes extérieurs
    - Exemples : proxys, DAO, ORB
  - Réaliser des *inversions de dépendances* entre packages
    - *Cf.* TD sur les compagnies aériennes
- Mise en œuvre
  - Utilisation d'objets du domaine
  - Création d'objets
    - Classes : *cf.* Fabrication pure
    - Interfaces : *cf.* Fabrication pure + Polymorphisme

Indirection : exemple

- Bibliothèque : accès à un système de stockage propriétaire





Indirection : discussion

- Remarques
  - Beaucoup de Fabrications pures sont créées pour des raison d'indirection
  - Objectif principal de l'indirection : faible couplage
- Adage (et contre adage)
  - « En informatique, on peut résoudre la plupart des problèmes en ajoutant un niveau d'indirection » (David Wheeler)
  - « En informatique, on peut résoudre la plupart des problèmes de performance en supprimant un niveau d'indirection »
- Patterns liés
  - GRASP : Fabrication pure, Faible couplage, Protection des variations
  - GoF : Adaptateur, Façade, Observateur…

Protection des variations (GRASP)

- Problème
  - Comment concevoir des objets, sous-systèmes, systèmes pour que les variations ou l'instabilité de certains éléments n'aient pas d'impact indésirable sur d'autres éléments ?
- Solution
  - Identifier les points de variation ou d'instabilité prévisibles
  - Affecter les responsabilités pour créer une interface (au sens large) stable autour d'eux (indirection)

Protection des variations (GRASP)

- Mise en œuvre
  - *Cf.* patterns précédents (Polymorphisme, Fabrication pure, Indirection)
- Exemples de mécanismes de PV
  - Encapsulation des données, brokers, machines virtuelles…
- Exercice
  - Stockage de Prêt dans plusieurs systèmes différents
  - Utiliser Indirection + Polymorphisme

Protection des variations : discussion

- Ne pas se tromper de combat
  - Prendre en compte les *points de variation*
    - Nécessaires car dentifiés dans le système existant ou dans les besoins
  - Gérer sagement les *points d'évolution*
    - Points de variation futurs, « spéculatifs » : à identifier (ne figurent pas dans les besoins)
    - Pas obligatoirement à implémenter
    - Le coût de prévision et de protection des points d'évolution peut dépasser celui d'une reconception

  ➔Ne pas passer trop de temps à préparer des protections qui ne serviront jamais

Protection des variations : discussion

- Différents niveaux de sagesse
  - le novice conçoit fragile
  - le meilleur programmeur conçoit tout de façon souple et en généralisant systématiquement
  - l'expert sait évaluer les combats à mener
- Avantages
  - Masquage de l'information
  - Diminution du couplage
  - Diminution de l'impact ou du coût du changement

Ne pas parler aux inconnus

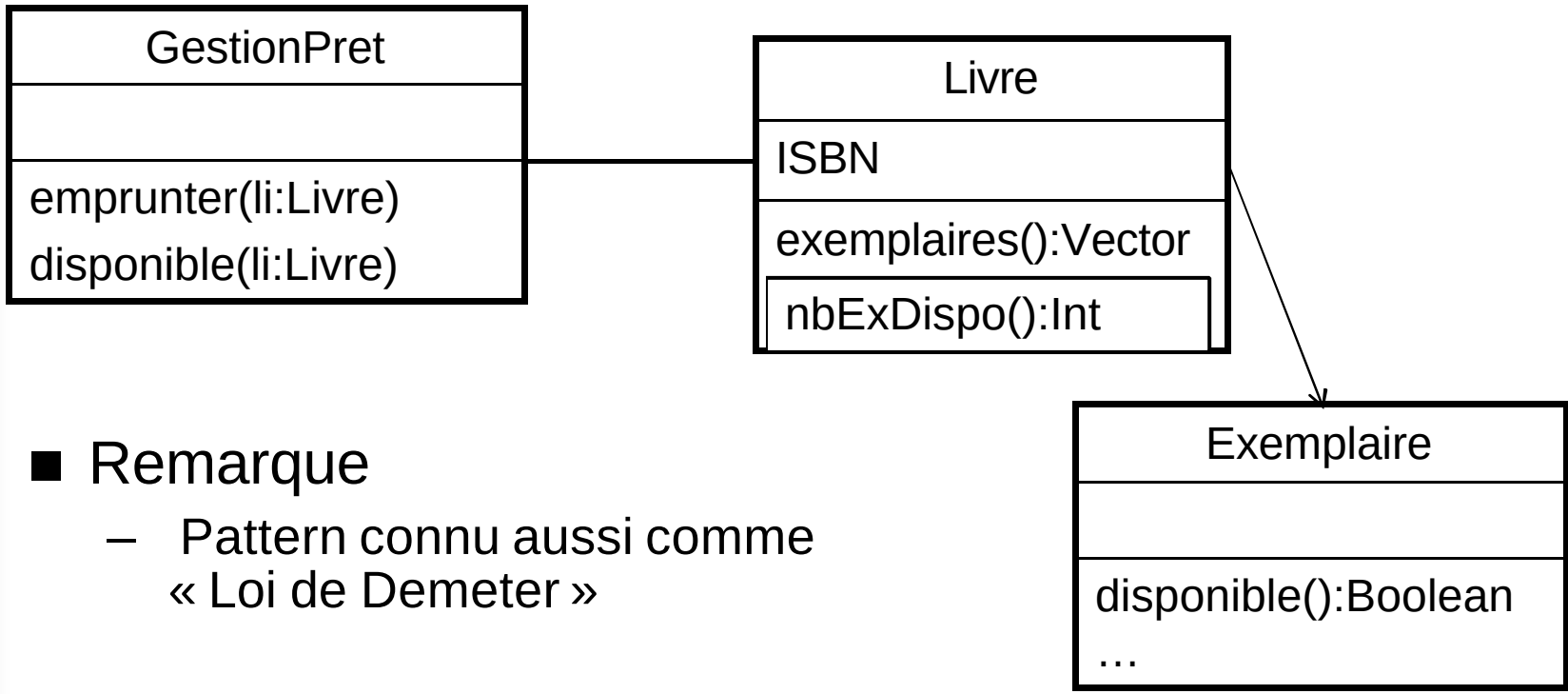
- Cas particulier de Protection des variations
  - protection contre les variations liées aux évolutions de structure des objets
- Problème
  - Si un client utilise un service ou obtient de l'information d'un *objet indirect* (inconnu)
    via un *objet direct* (familier du client),
    comment le faire sans couplage ?
  - Cas général à éviter : a.getB().getC().getD().methodeDeD();
- Solution
  - Éviter de connaître la structure d'autres objets indirectement
  - Affecter la responsabilité de collaborer avec un objet indirect à un objet que le client connaît directement
    pour que le client n'ait pas besoin de connaître ce dernier.

Ne pas parler aux inconnus (suite)

- Cas général à éviter a.getB().getC().getD().methodeDeD();
  - Si l'une des méthodes de la chaîne disparaît, A devient inutilisable
- Préconisation
  - Depuis une méthode, n'envoyer des messages qu'aux objets suivants
    - l'objet *this* (self)
    - un paramètre de la méthode courante
    - un attribut de *this*
    - un élément d'une collection qui est un attribut de *this*
    - un objet créé à l'intérieur de la méthode
- Implication
  - ajout d'opérations dans les objets directs pour servir d'opérations intermédiaires

Ne pas parler aux inconnus : exemple

- Comment implémenter *disponible()* dans GestionPret ?



- Remarque
  - Pattern connu aussi comme « Loi de Demeter »

Les patterns GRASP et les autres

- D’une certaine manière, tous les autres patterns sont
  - des applications,
  - des spécialisations,
  - des utilisations conjointes

  des 9 patterns GRASP, qui sont les plus généraux.

Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks

Généralités

- Origine dans l'architecture
  - ouvrages de Christopher Alexander (77)
- Propriétés
  - pragmatisme
    - solutions existantes et éprouvées
  - récurrence
    - bonnes manières de faire éprouvées
  - générativité
    - comment et quand appliquer, indépendance au langage de programmation
  - émergence
    - la solution globale émerge de l'application d'un ensemble de patrons

Types de patrons informatiques

- Patrons de conception
  - architecture
    - conception de systèmes
  - conception
    - interaction de composants
      - comportement
      - structure
      - création
  - idiomes de programmation
    - Techniques, styles spécifiques à un langage
- Patrons d'analyse
- Patrons d'organisation
- …

Références

- Ouvrage du « Gang of Four »
  - Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994), *Design patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 395 p. (trad. française : *Design patterns. Catalogue des modèles de conception réutilisables*, Vuibert 1999)
- Plus orienté architecture
  - Martin Fowler (2002) *Patterns of Enterprise Application Architecture*, Addison Wesley
- Sites
  - http://www.hillside.net/patterns
  - http://java.sun.com/blueprints/corej2eepatterns/
  - …

Éléments d'un patron

- Nom
  - Évocateur, référence concise
- Problème
  - Objectifs que le patron cherche à atteindre
- Contexte initial
  - Comment le problème survient
  - Quand la solution fonctionne
- Forces/contraintes
  - Forces et contraintes interagissant au sein du contexte
  - Détermination des compromis
- Solution
  - Comment mettre en œuvre la solution ?
  - Point de vue statique (structure) et dynamique (interactions)
  - Variantes de solutions

Éléments d'un patron (suite)

- Exemples
  - Exemples d'applications
- Contexte résultant
  - Description du contexte résultant de l'application du patron au contexte initial
  - Conséquences positives et négatives
- Justification
  - Raisons fondamentales conduisant à l'utilisation du patron
  - Réflexions sur la qualité du patron
- Patrons associés
  - Similaires ou possédant des contextes initial ou résultant proches
- Utilisations connues
  - Exemples d'applications réels

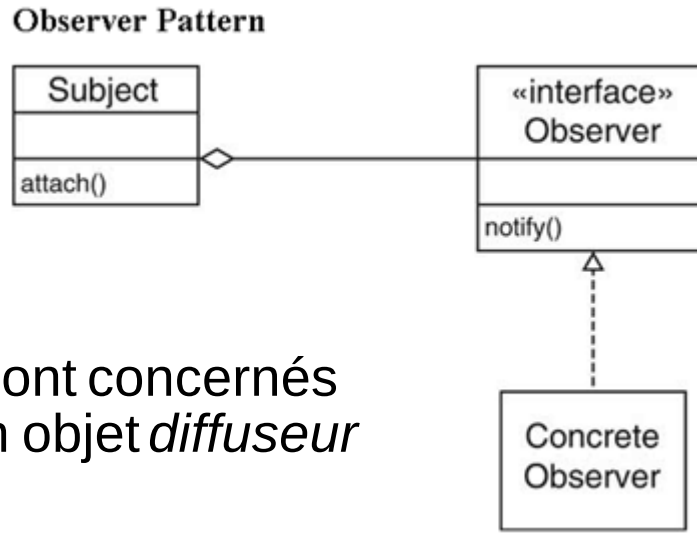
Les patrons ne sont pas

- Limités au domaine informatique
- Des idées nouvelles
- Des solutions qui n’ont fonctionné qu’une seule fois
- Des principes abstraits ou des heuristiques
- Une panacée

Les patrons sont

- Des solutions éprouvées à des problèmes récurrents
- Spécifiques au domaine d'utilisation
- Rien d'exceptionnel pour les experts d'un domaine
- Une forme littéraire pour documenter des pratiques
- Un vocabulaire partagé pour discuter de problèmes
- Un moyen efficace de réutiliser et partager de l'expérience

Observateur



- Contexte
  - Plusieurs objets *souscripteurs* sont concernés par les changements d'état d'un objet *diffuseur*
- Problème
  - Comment faire pour que chacun d'eux soit informé de ces changements ?
  - Comment maintenir un faible couplage entre diffuseur et souscripteurs ?
- Solution (théorique)
  - Définir une interface « Souscripteur » ou « Observer »
  - Faire implémenter cette interface à chaque souscripteur
  - Le diffuseur peut enregistrer dynamiquement les souscripteurs intéressés par un événement et le leur signaler

Observateur (suite)

- Fonctionnement
  - Un *Observateur* s'attache à un *Sujet*
  - Le sujet *notifie* ses observateurs en cas de changement d'état
- En pratique
  - Subject : classe abstraite
  - ConcreteSubject : classe héritant de Subject
  - Observer : classe (utilisée comme classe abstraite)
  - ConcreteObserver : classe héritant d'Observer
- Autres noms
  - Diffusion-souscription
  - Modèle de délégation d'événements

Modèle-Vue-Contrôleur

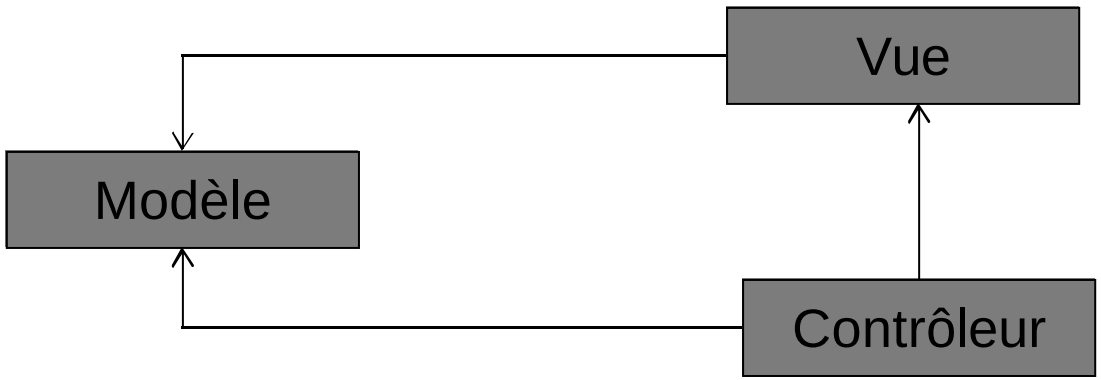
- Problème
  - Comment rendre le modèle (domaine métier) indépendant des vues (interface utilisateur) qui en dépendent ?
  - Réduire le couplage entre modèle et vue
- Solution
  - Insérer une couche supplémentaire (contrôleur) pour la gestion des événements et la synchronisation entre modèle et vue

Modèle-Vue-Contrôleur (suite)

- Modèle (logique métier)
  - Implémente le fonctionnement du système
  - Gère les accès aux données métier
- Vue (interface)
  - Présente les données en cohérence avec l'état du modèle
  - Capture et transmet les actions de l'utilisateur
- Contrôleur
  - Gère les changements d'état du modèle
  - Informe le modèle des actions utilisateur
  - Sélectionne la vue appropriée

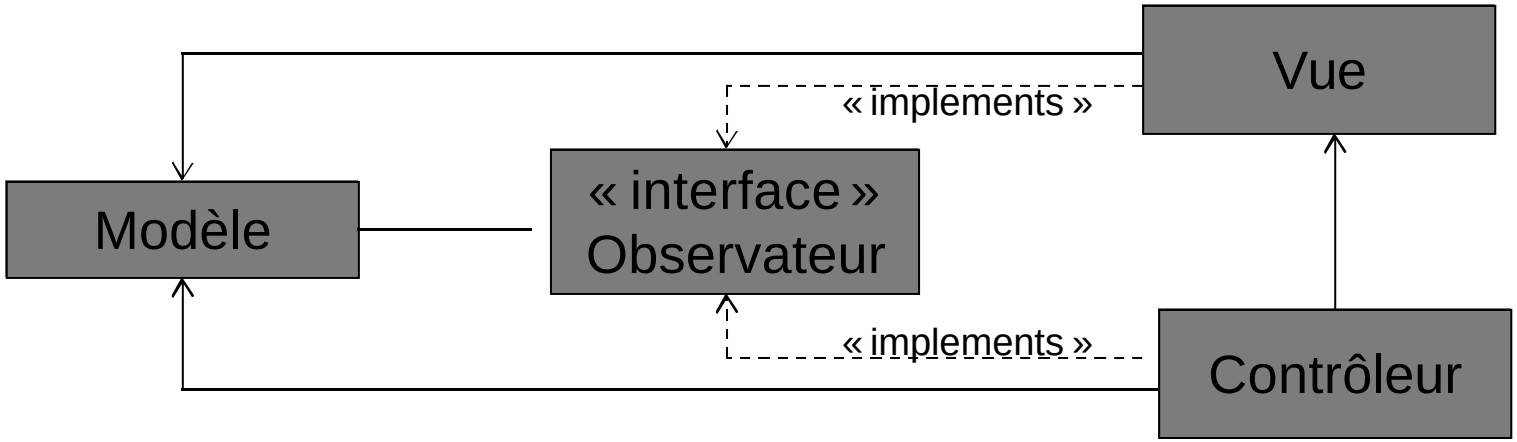
Modèle-Vue-Contrôleur (suite)

- Version modèle passif
  - la vue se construit à partir du modèle
  - le contrôleur notifie le modèle des changements que l'utilisateur spécifie dans la vue
  - le contrôleur informe la vue que le modèle a changé et qu'elle doit se reconstruire



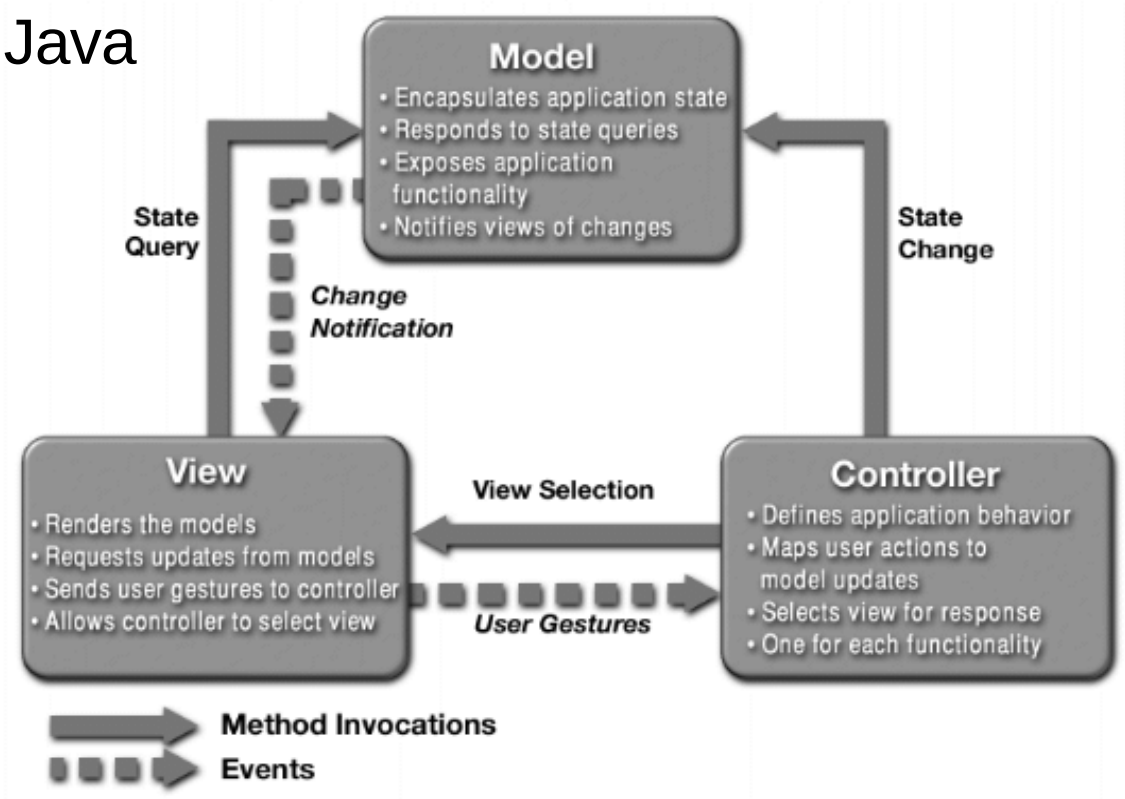
Modèle-Vue-Contrôleur (suite)

- Version modèle actif
  - quand le modèle peut changer indépendamment du contrôleu
  - le modèle informe les abonnés à l'observateur qu'il s'est modifié
  - ceux-ci prennent l'information en compte (contrôleur et vues)



Modèle-Vue-Contrôleur (suite)

- Version Java



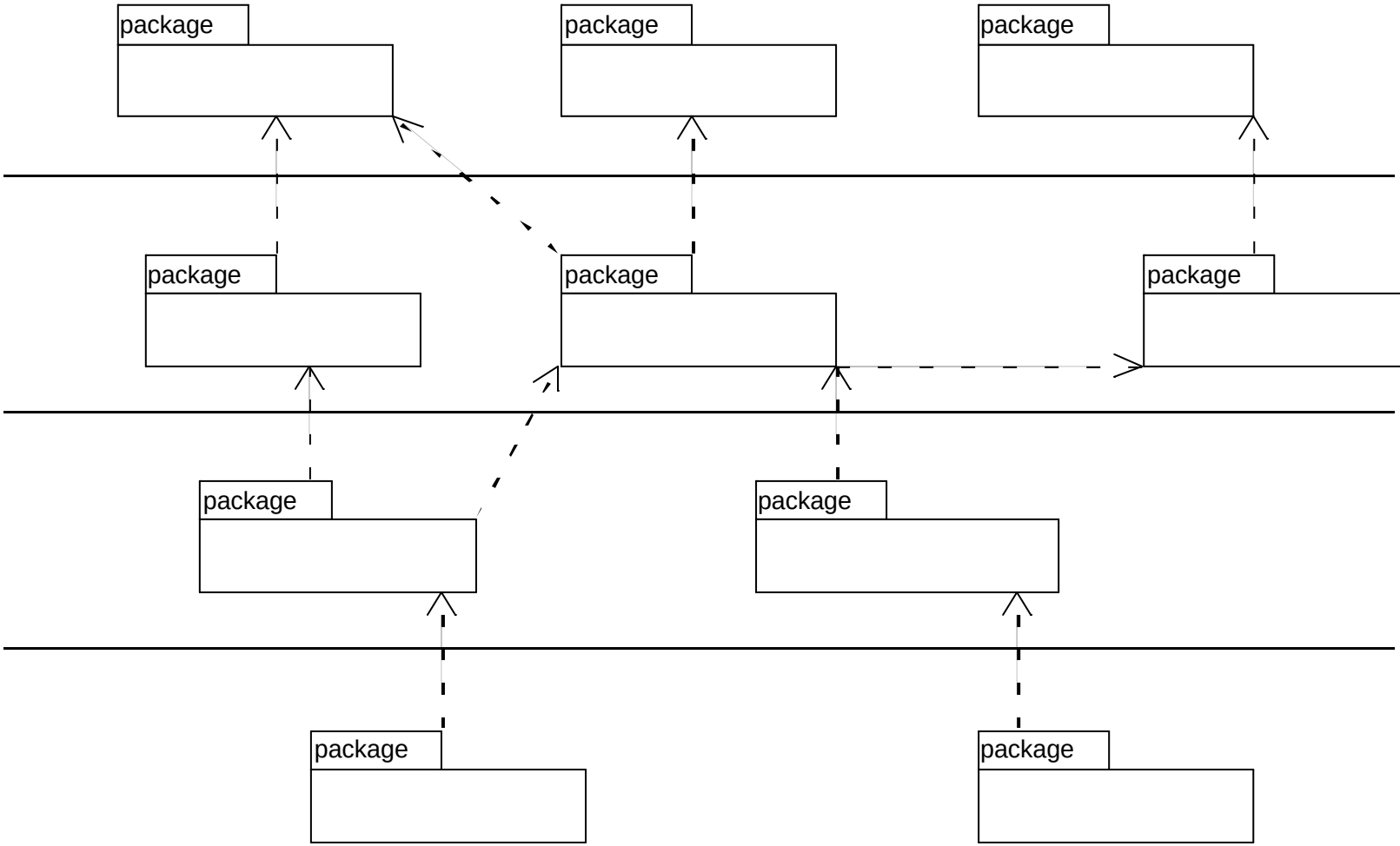
Source : http://java.sun.com/blueprints/patterns/MVC-detailed.html

Modèle-Vue-Contrôleur (suite)

- Différentes versions
  - la vue connaît le modèle ou non
  - le contrôleur connaît la vue ou non
  - le vue connaît le contrôleur ou non
  - « Mélange » avec le pattern Observer
  - Un ou plusieurs contrôleurs
- Choix d'une solution
  - dépend des caractéristiques de l'application
  - dépend des autres responsabilités du contrôleur

Pattern Couches

Présentation
(Application)
Domaine
Service
Middleware
Fondation



Fabrique concrète

- Classe responsable de la création d'objets
  - lorsque la logique de création est complexe
  - lorsqu'il convient de séparer les responsabilité de création
- Fabrique concrète = objet qui fabrique des instances
- Avantages par rapport à un constructeur
  - la classe a un nom
  - permet de gérer facilement plusieurs méthodes de construction avec des signatures similaires
  - peut retourner plusieurs types d'objets

Factory method (GoF / Gang Of Four)

- Factory
  - un objet qui fabrique des instances conformes à une interface ou une classe abstraite
  - par exemple, une *Application* veut manipuler des documents, qui répondent à une interface *Document*
    - ou une *Equipe* veut gérer des *Tactique…*

Factory - Fabrique (GoF)

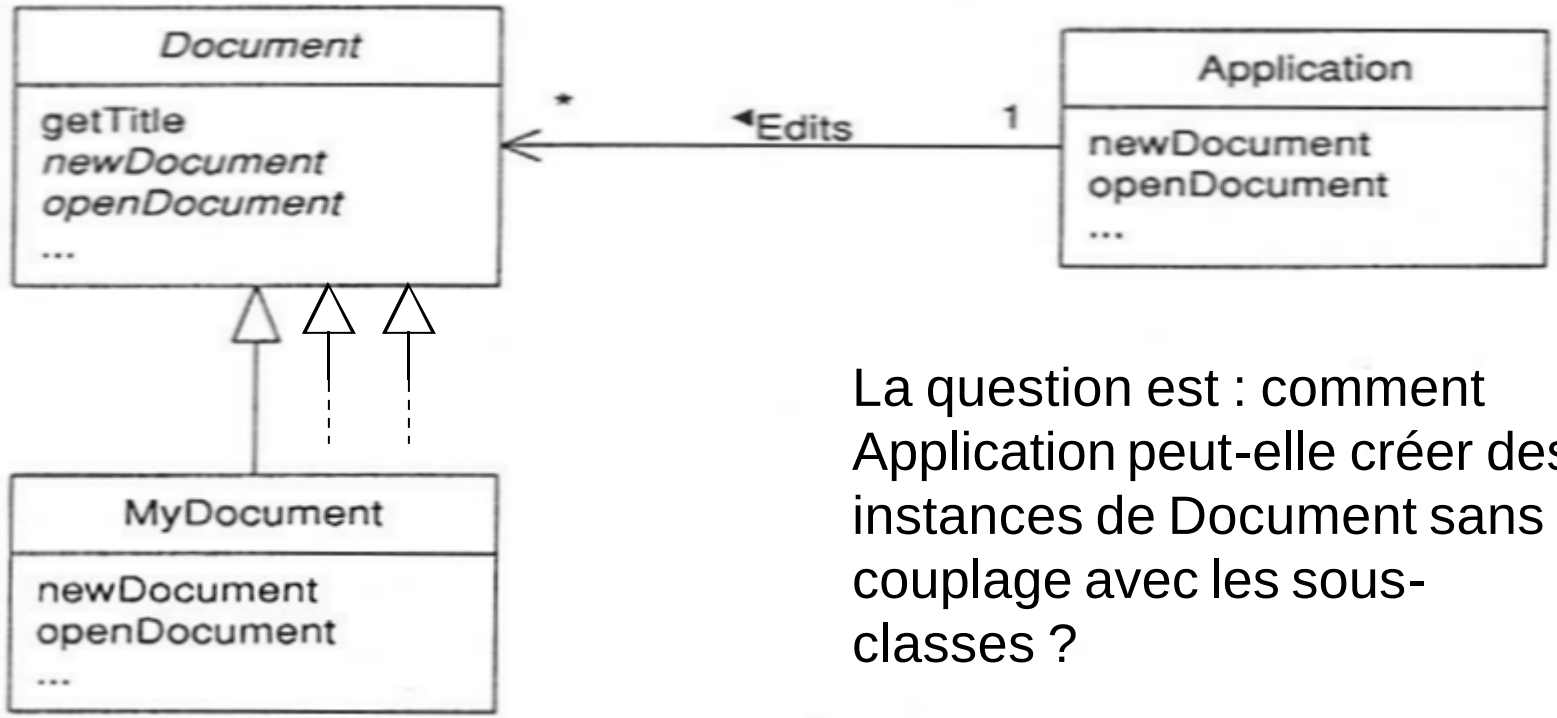
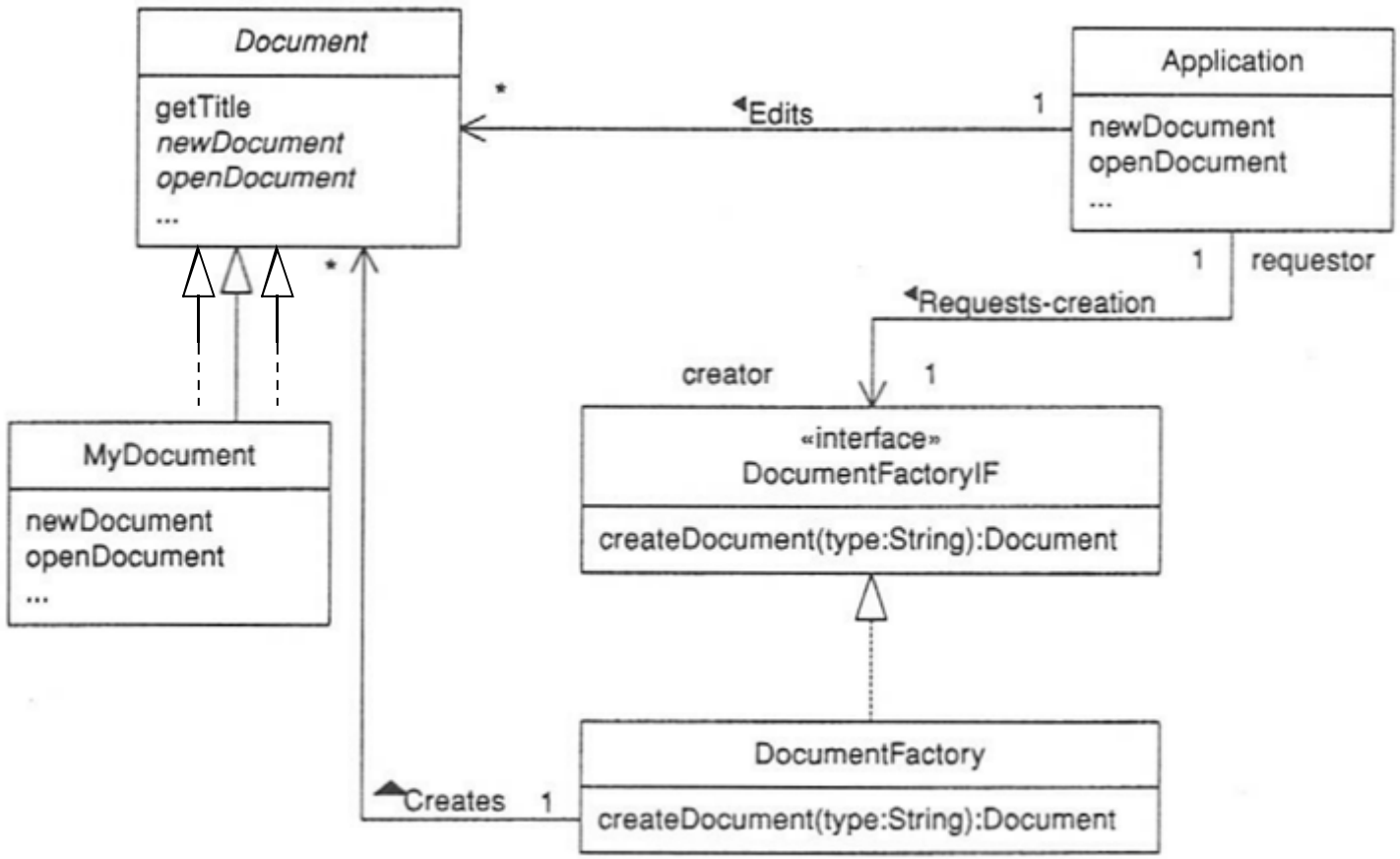


FIGURE 5.1 Application framework.

La question est : comment Application peut-elle créer des instances de Document sans couplage avec les sous-classes ?

(From Grand's book.)

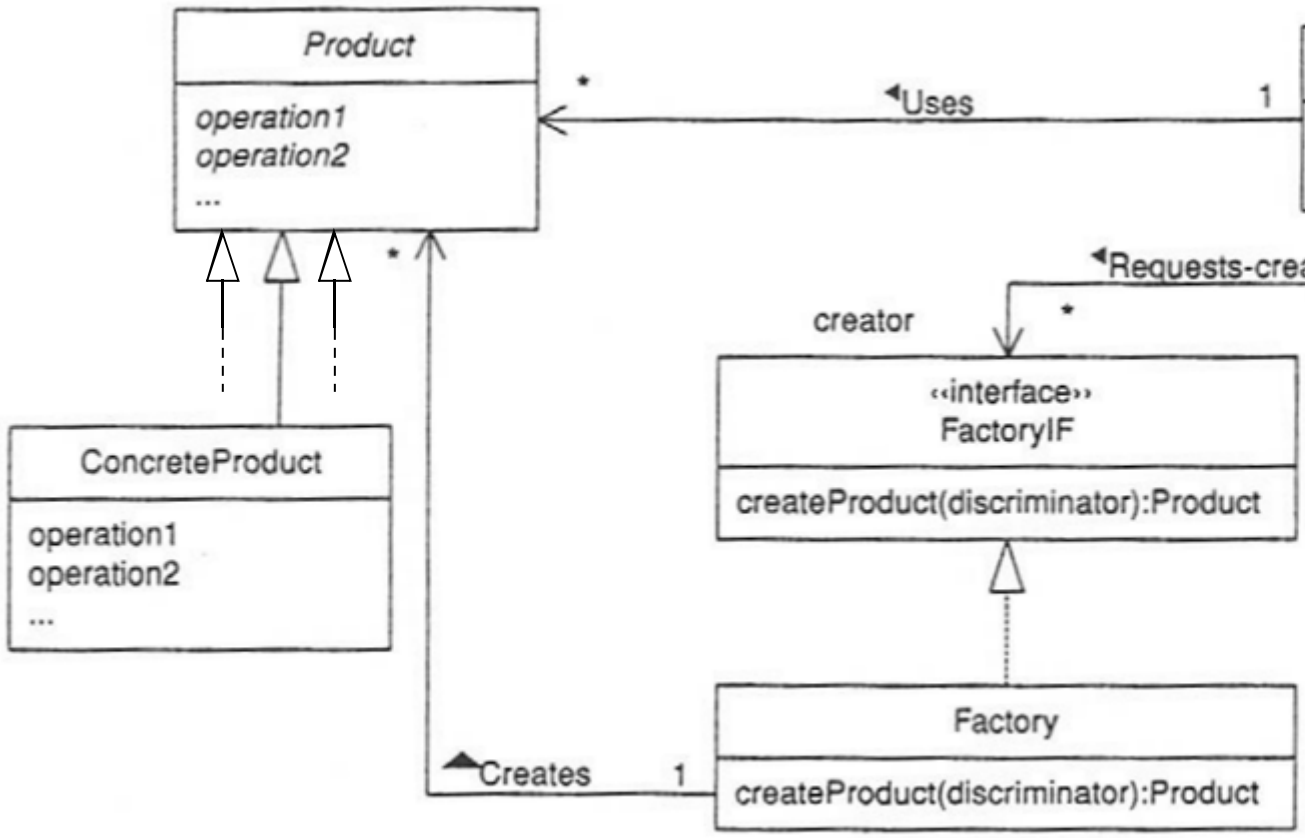
Solution : utiliser une classe DocumentFactory pour créer différents types de documents



(From Grand's book.)

FIGURE 5.2 Application framework with document factory.

Factory Method Pattern : structure générale

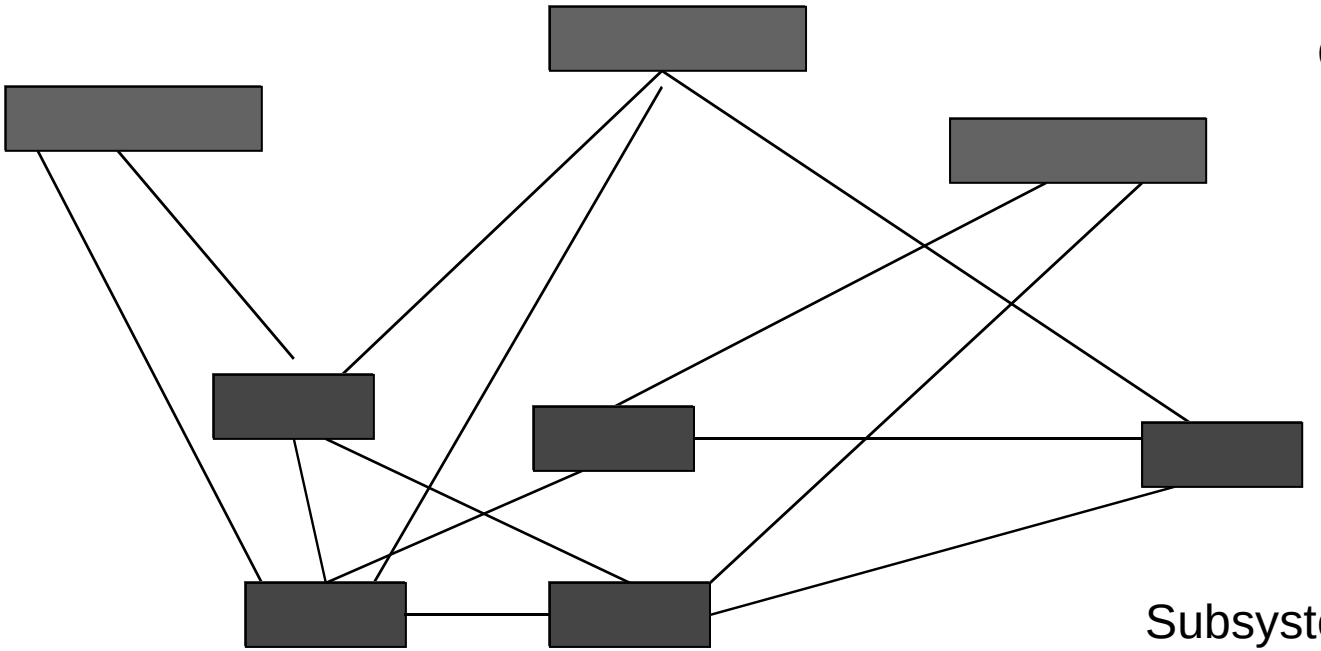


discriminator : paramètre indiquant quel typ de sous-classe d Product créer

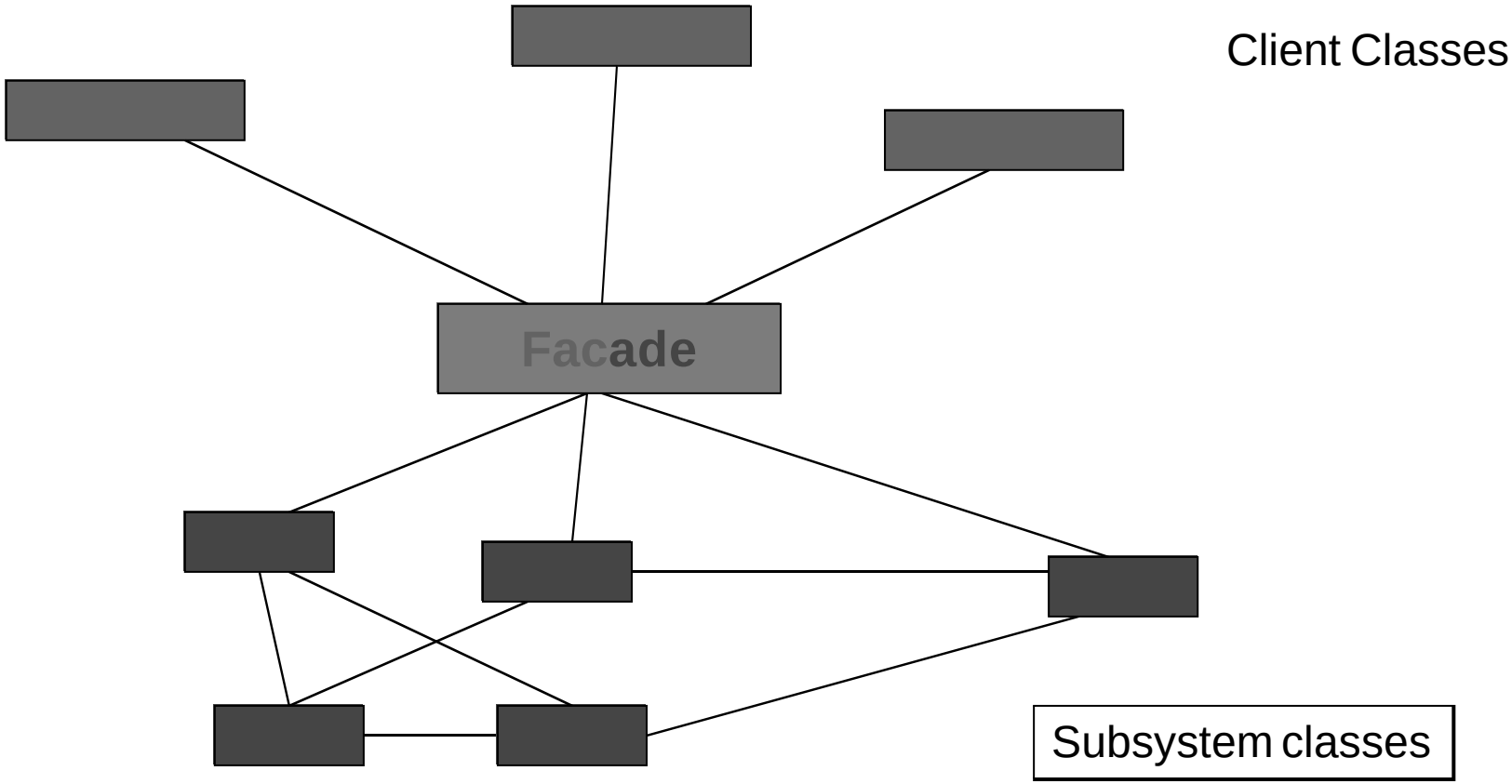
FIGURE 5.3 Factory method pattern.

(From Grand's book.)

Façade (GoF) : problème

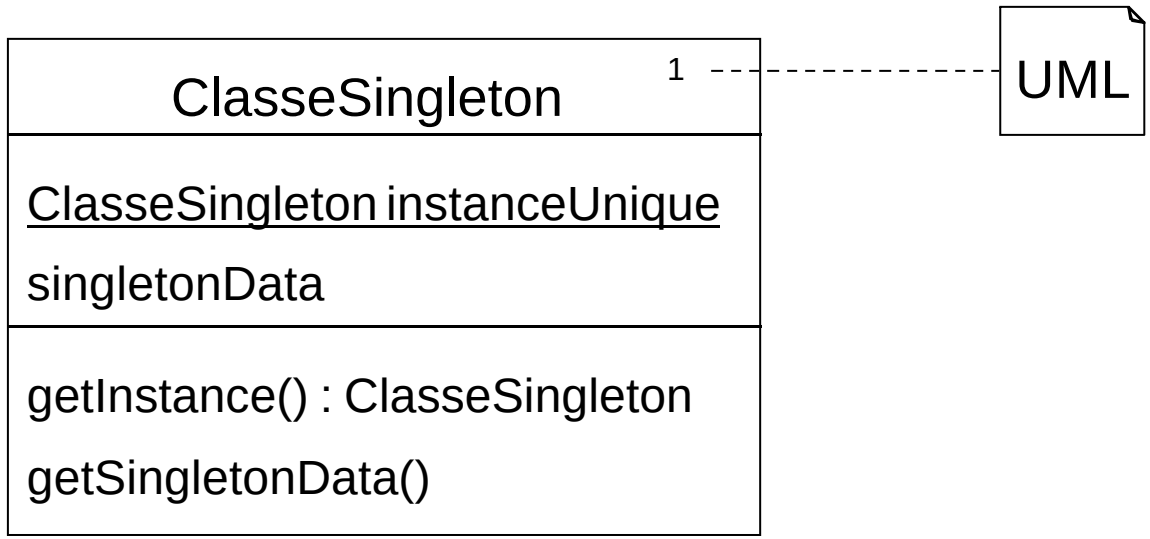


Façade (GoF) : solution



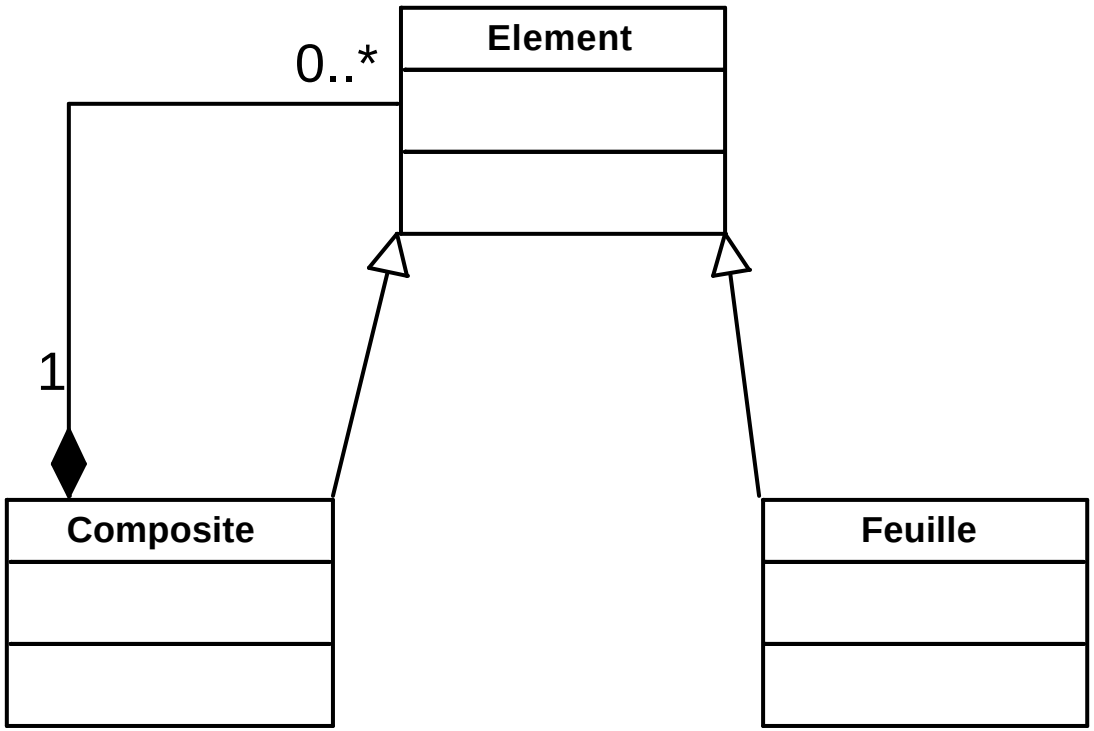
Singleton (GoF)

- Quand on a besoin d'une instance unique d'une classe (ex. Factory), avec un point d'accès unique et global pour les autres objets
- Singleton
  - utiliser une méthode statique de la classe qui retourne l'instance



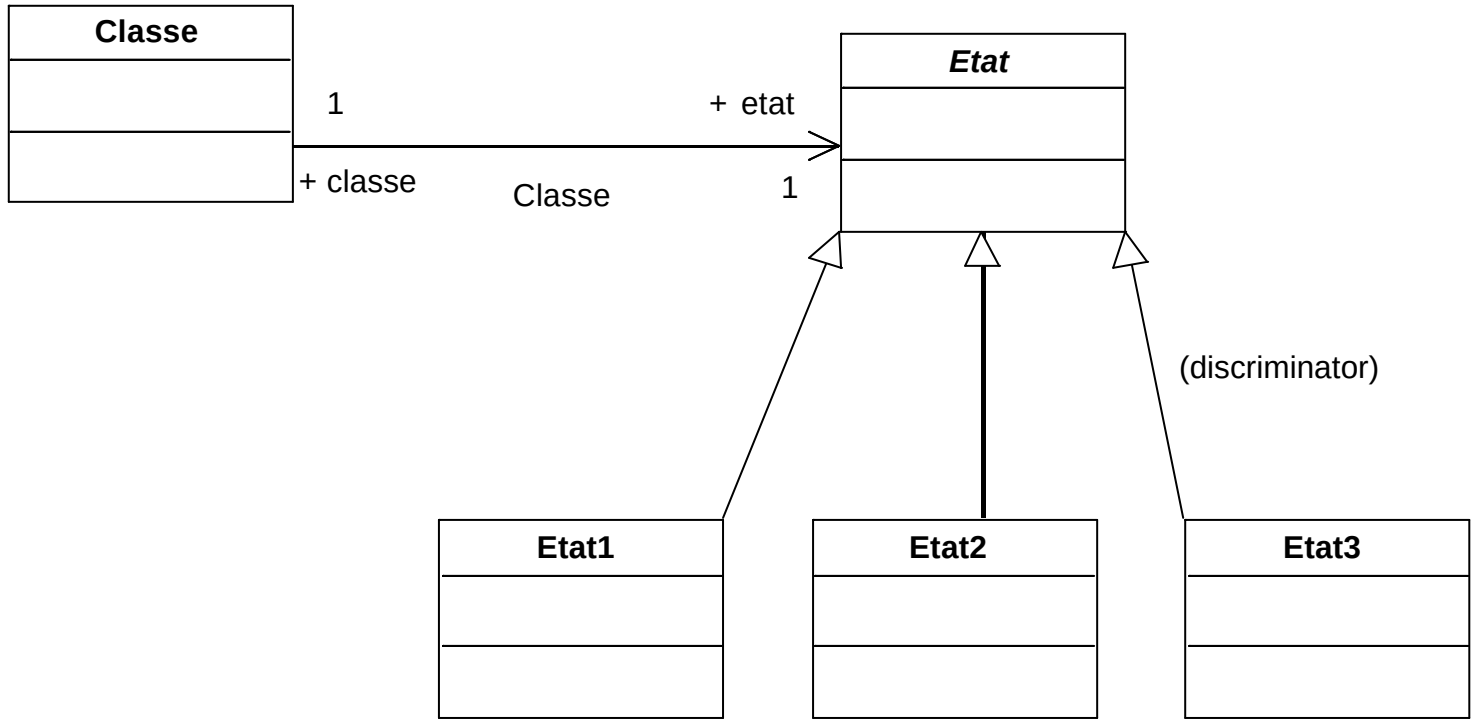
Composite (GoF)

- Pour représenter une hiérarchie de composition



Etat (GoF)

- Pour déléguer un comportement complexe qui varie en fonction de l'état



Autres patterns

- Bridge (GoF)
- Chain of responsability (GoF)
- Proxy (Gof)
- Adaptateur (GoF)
- DAO (J2EE)
- …

(*wikipedi*

Anti-patrons

- Erreurs courantes de conception documentées
- Caractérisés par
  - lenteur du logiciel
  - Coûts de réalisation ou de maintenance élevés
  - Comportements anormaux
  - Présence de bogues.
- Exemples
  - Action à distance
    - Emploi massif de variables globales, fort couplage
  - Coulée de lave
    - Partie de code encore immature mise en production, forçant la lave à se solidifier en empêchant sa modification
  - …

IDE « orientés » Design patterns

- Fournir une aide à l'instanciation ou au repérage de patterns
  - nécessite une représentation graphique (au minimum collaboration UML) et le codage de certaines contraintes
- Instanciation
  - choix d'un pattern, création automatique des classes correspondantes
- Repérage
  - assister l'utilisateur pour repérer
    - des patterns utilisés (pour les documenter)
    - des « presque patterns » (pour les faire tendre vers des patterns)
- Exemples d'outils
  - Describe + Jbuilder
  - Objecteering
  - Struts
  - …

Plan

- Introduction sur les patterns
- Patrons GRASP
- Design patterns
- Frameworks

Framework

- Définition
  - ensemble de classes qui collaborent à la réalisation d'une responsabilité qui dépasse celle de chacune
  - conception générale réutilisable pour une classe d'application donnée
- Un framework définit
  - architecture, classes, responsabilités, flot de contrôle, *etc*.
- Un framework doit être spécialisé
  - reprise de code de haut niveau (beaucoup de classes abstraites), ajout de code de spécialisation

Exemple de framework : STRUTS

- Couche IHM d'une application web
  - gérer des sessions utilisateur, gérer des actions en fonction des requêtes HTTP
- Basé sur MVC
- Ensemble de classes à spécialiser
  - Fabriques
  - Contrôleurs
  - …

Conclusion

- On a vu assez précisément les patterns les plus généraux (GRASP)
- On a survolé les autres
  - un bon programmeur doit les étudier et en connaître une cinquantaine

Remerciements

- Olivier Aubert
- Yohan Welikala (Sri Lanka)