

Design patterns : de la réutilisation dans les SI

Université Claude Bernard Lyon 1 – M1 Informatique
M1IF01 – Gestion de projet et génie logiciel

Lionel Médini – septembre 2020
D'après le cours de Yannick Prié



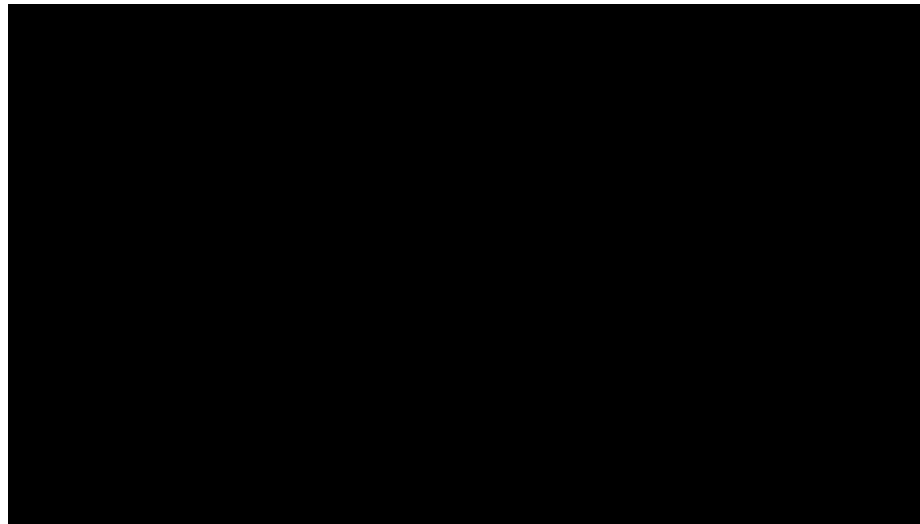


Plan

- Introduction
- Principes GRASP
- Design patterns
- Patterns architecturaux
- Conclusion

Réutilisation

- Constante de la conception d'outils en général
 - Ex. : je dois concevoir un outil pour casser une noix
<https://youtu.be/ZEskkZyuoKw?t=1m10s>



- Dois-je tout concevoir depuis zéro ?
- Que puis-je réutiliser ?

Réutilisation en informatique

- Réutilisation de code métier
 - Sous la forme de bibliothèques / composants
 - À acheter / fabriquer
 - Réutilisation de code générique
 - Sous la forme de frameworks
 - À utiliser en les spécialisant
 - Réutilisation de principes de conception
 - Dès que des principes se révèlent pertinents
 - Abstraction
 - Réutilisation
- } Design patterns**



Généralités sur les patterns

■ Définition

- Modèle de solution à un problème de conception classique, dans un contexte donné

■ Objectifs

- Identifier, catégoriser et décrire un problème et la solution proposée
- Faire émerger la solution globale grâce à l'application d'un ensemble de patterns

■ Spécification de la solution

- Structure et/ou comportement d'une société d'objets

Historique

- Origine en architecture
 - Ouvrage : *A pattern language: Towns, Buildings, Construction*, Christopher Alexander (1977)
- « Récupération » en IHM (Interaction design)
 - Ouvrage : *User Centered System Design*, Donald Norman & Stephen Draper (1986)
- ...Puis en conception informatique
 - 1987 : 1er projet de conception mettant en oeuvre des DP par K. Beck & W. Cunningham, Tektronix
 - 1991 : [Gang Of Four](#) : Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides ; *Design Patterns: Elements of Reusable Object-Oriented Software*
 - 1994 : the [Hillside Group](#) créent une série de conférences : *Pattern Languages of Programs*

Éléments d'un patron (1/3)

- Nom
 - Évocateur
 - Concis (un ou deux mots)
- Problème
 - Points bloquants que le patron cherche à résoudre
- Contexte initial
 - Comment le problème survient
 - Quand la solution fonctionne
- Forces/contraintes
 - Forces et contraintes interagissant au sein du contexte
 - Détermination des compromis

Éléments d'un patron (2/3)

■ Solution

- Comment mettre en œuvre la solution ?
- Point de vue statique (structure) et dynamique (interactions)
- Description abstraite
 - Élément de conception, relation, responsabilités, collaboration
- Variantes de solutions

■ Contexte résultant

- Description du contexte résultant de l'application du patron au contexte initial
- Conséquences positives et négatives

■ Exemples

- Illustrations simples d'application du pattern
- Applications dans des cas réels

Éléments d'un patron (3/3)

■ Justification

- Raisons fondamentales conduisant à l'utilisation du patron
- Réflexions sur la qualité du patron

■ Patrons associés

- Similaires
- Possédant des contextes initial ou résultant proches

■ Discussion

- Avantages, inconvénients
- Conseils d'application / d'implémentation
- Variantes
- Autres...

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

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

Principles Are Not Patterns

- Principes généraux très utiles, mais qui ne s'appliquent pas à un problème concret
- Exemples
 - “Patterns” GRASP
 - Keep it Simple, Stupid (KISS)
 - Don't Repeat Yourself (DRY) / Duplication is Evil (DIE)
 - "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"
 - S'applique à tout le système : code, configuration, modèles, documentation, etc.
 - You aren't gonna use it
 - Issu de l'extreme programming (XP)

Principles Are Not Patterns

■ Exemples

– SOLID

- Single responsibility
 - Voir forte cohésion
- Open/closed
 - les classes doivent être ouvertes à l'extension, mais fermées à la modification
- Liskov substitution
 - “Substituabilité” d'un objet par ses sous-types
 - Voir Design by Contract
- Interface segregation
 - Utiliser l'interface la plus spécifique possible
- Dependency inversion
 - Dépendre d'une abstraction (interface) et non d'une implémentation

Types de patterns

- Idiomes de programmation
 - Techniques, styles spécifiques à un langage
- Patrons de conception
 - conception
 - interaction de composants
 - architecture
 - conception de systèmes
- Patrons d'analyse
- Patrons d'organisation
- ...

Classifications des patterns de conception

■ Par scope

- Classe : statique, interne à une classe ou ses sous-classes
- (Société d') objet(s) : mis en oeuvre par les relations dynamiques entre objets
- Application : permettent de mettre en place l'architecture générale d'une application ou d'un framework

Design patterns

Architectural patterns

■ Par objectif

- Création, structure, comportement, concurrence...

Design patterns

Contenu de ce cours

- Beaucoup (trop ?) de choses
 - Principes / définitions
 - Liste de patterns
 - Ordonnée par catégories
 - Non exhaustive
 - Et pourtant très longue
 - Synthèse / généralisation
- À considérer comme des pointeurs



Plan

- Introduction
- Principes GRASP
- Design patterns
- Patterns architecturaux
- Conclusion

Conception pilotée par les responsabilités

■ Métaphore

Concevoir une société d'objets responsables qui collaborent dans un objectif commun

■ Concrètement

- penser l'organisation des composants en termes de **responsabilités** par rapport à des **rôles**, au sein de **collaborations**

Conception pilotée par les responsabilités

■ Rôle

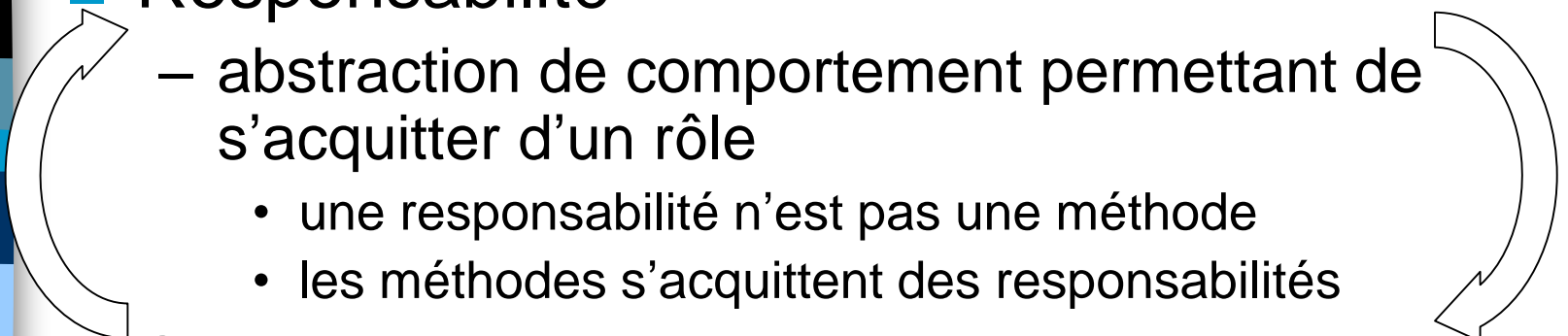
- contrat, obligation vis-à-vis du fonctionnement global
 - remplit une partie de l'objectif
 - déduit de l'expression des besoins en phase de conception

■ Responsabilité

- abstraction de comportement permettant de s'acquitter d'un rôle
 - une responsabilité n'est pas une méthode
 - les méthodes s'acquittent des responsabilités

■ Collaboration

- échanges d'informations entre responsabilités
 - interfaces entre les classes / packages / composants
 - types de communication (appel de méthode, événement.¹⁹)



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

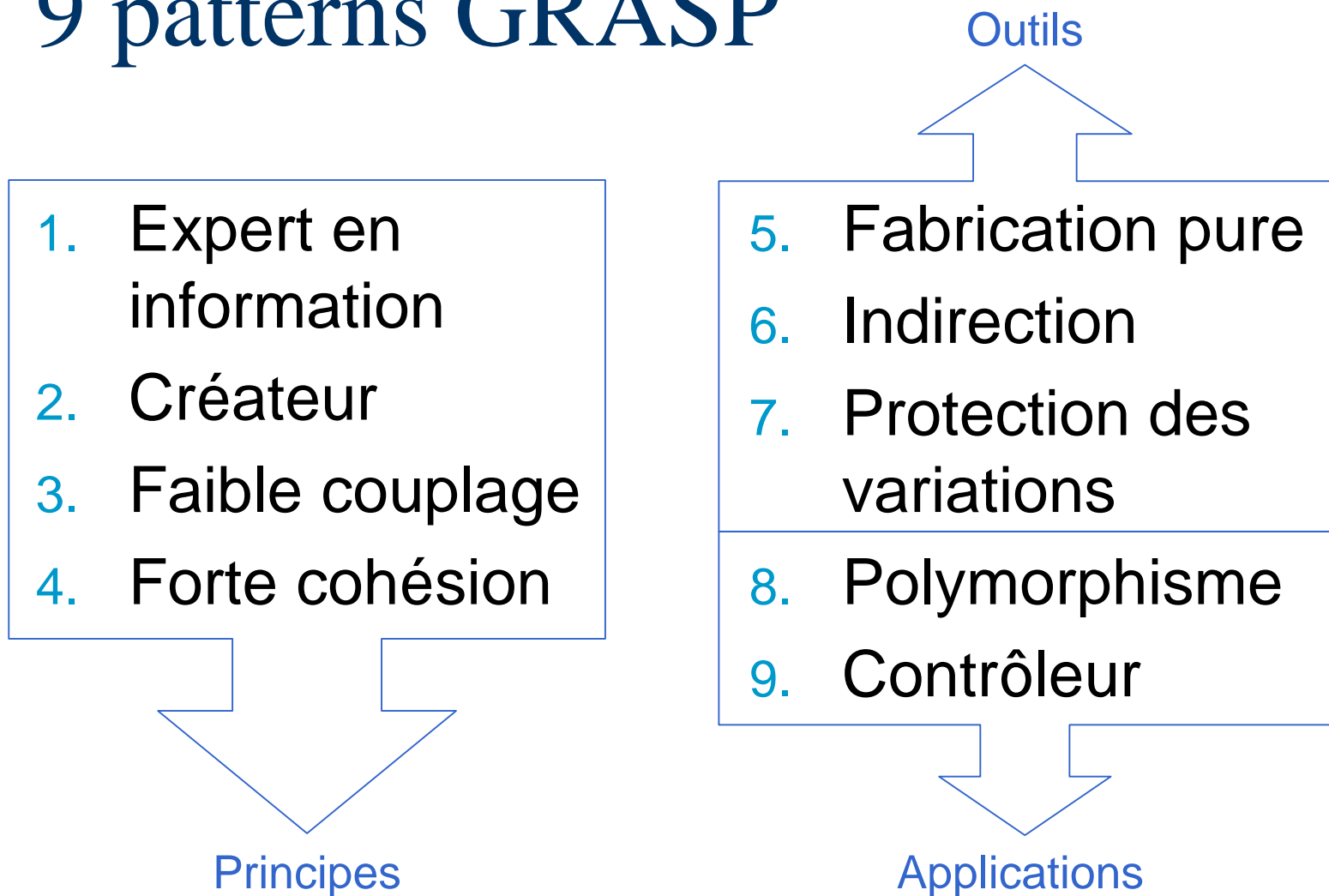
■ Faire

- *Abonné* est responsable de la vérification du retard sur les livres prêtés

General Responsibility Assignment Software Patterns (GRASP)

- Ensemble de principes (plutôt que 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 2005

9 patterns GRASP



Expert (GRASP)

■ Problème

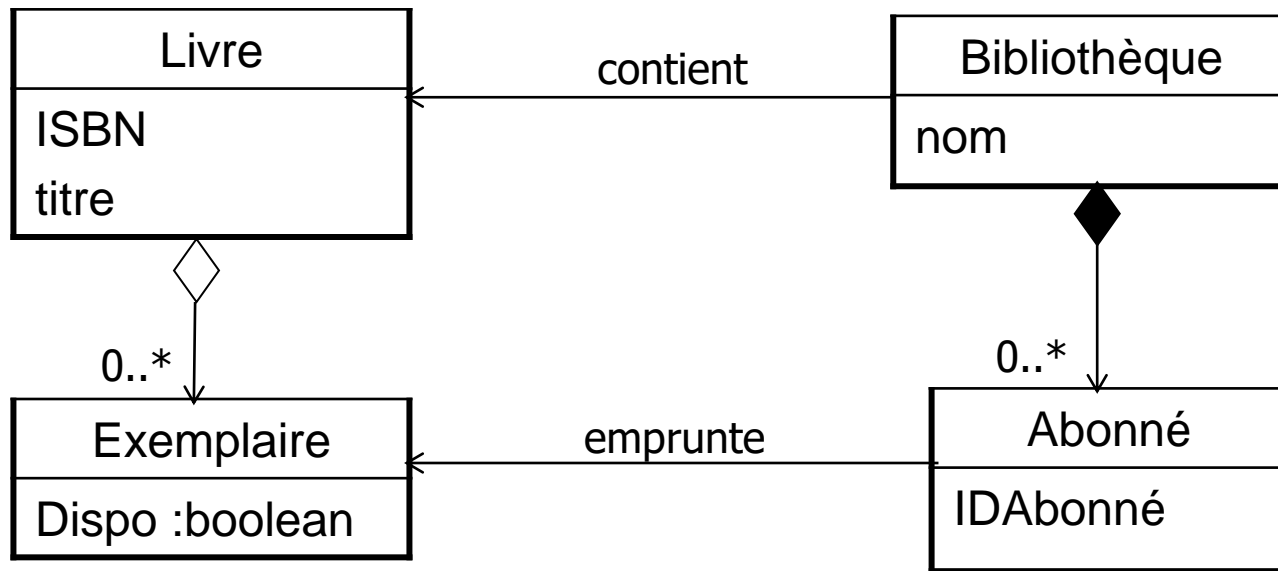
- Quel est le principe général d'affectation des responsabilités aux objets ?

■ Solution

- Affecter la responsabilité à l'*expert* en information
 - la classe qui possède les informations nécessaires pour s'acquitter de la responsabilité

Expert : exemple

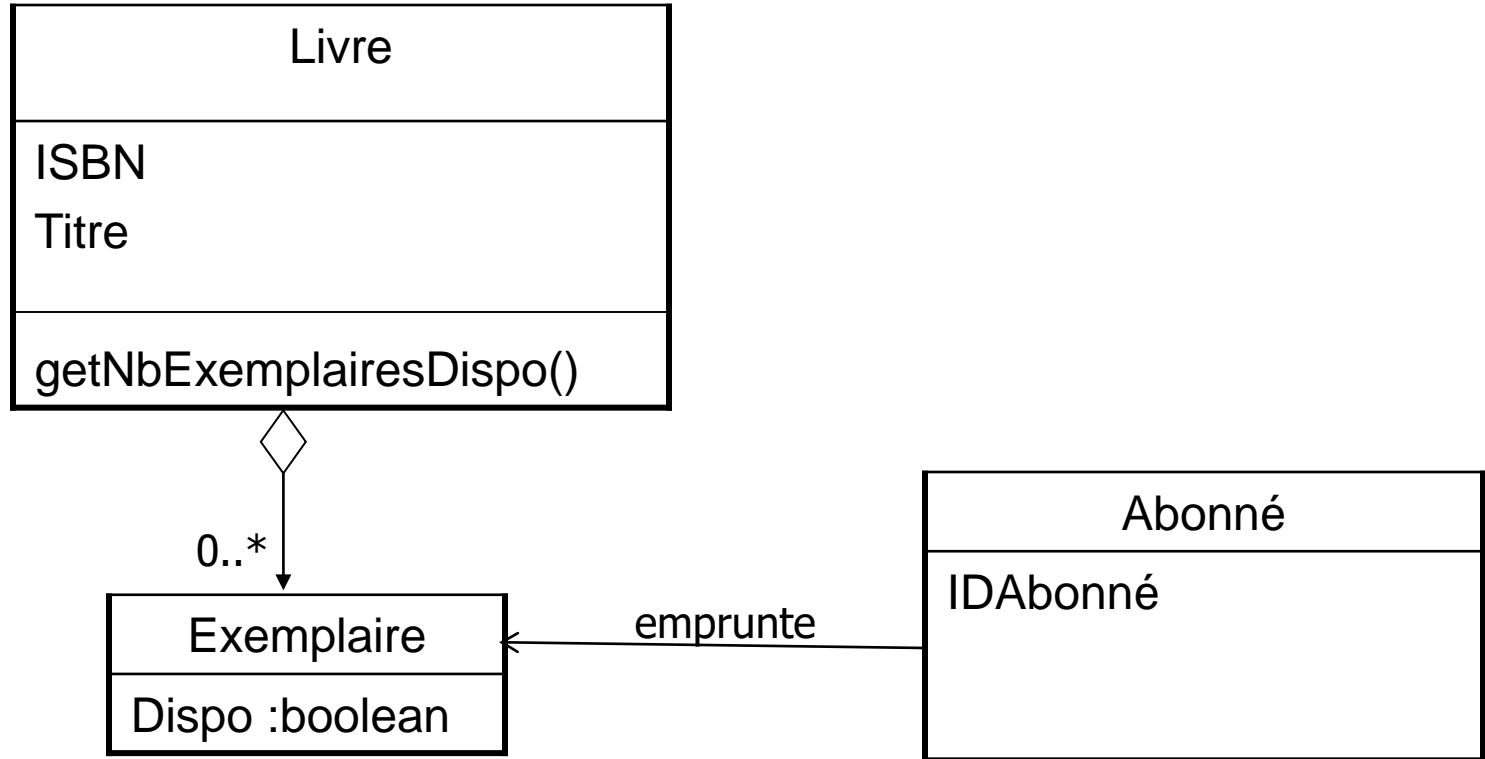
- Bibliothèque : qui doit avoir la responsabilité de connaître le nombre d'exemplaires disponibles ?



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)

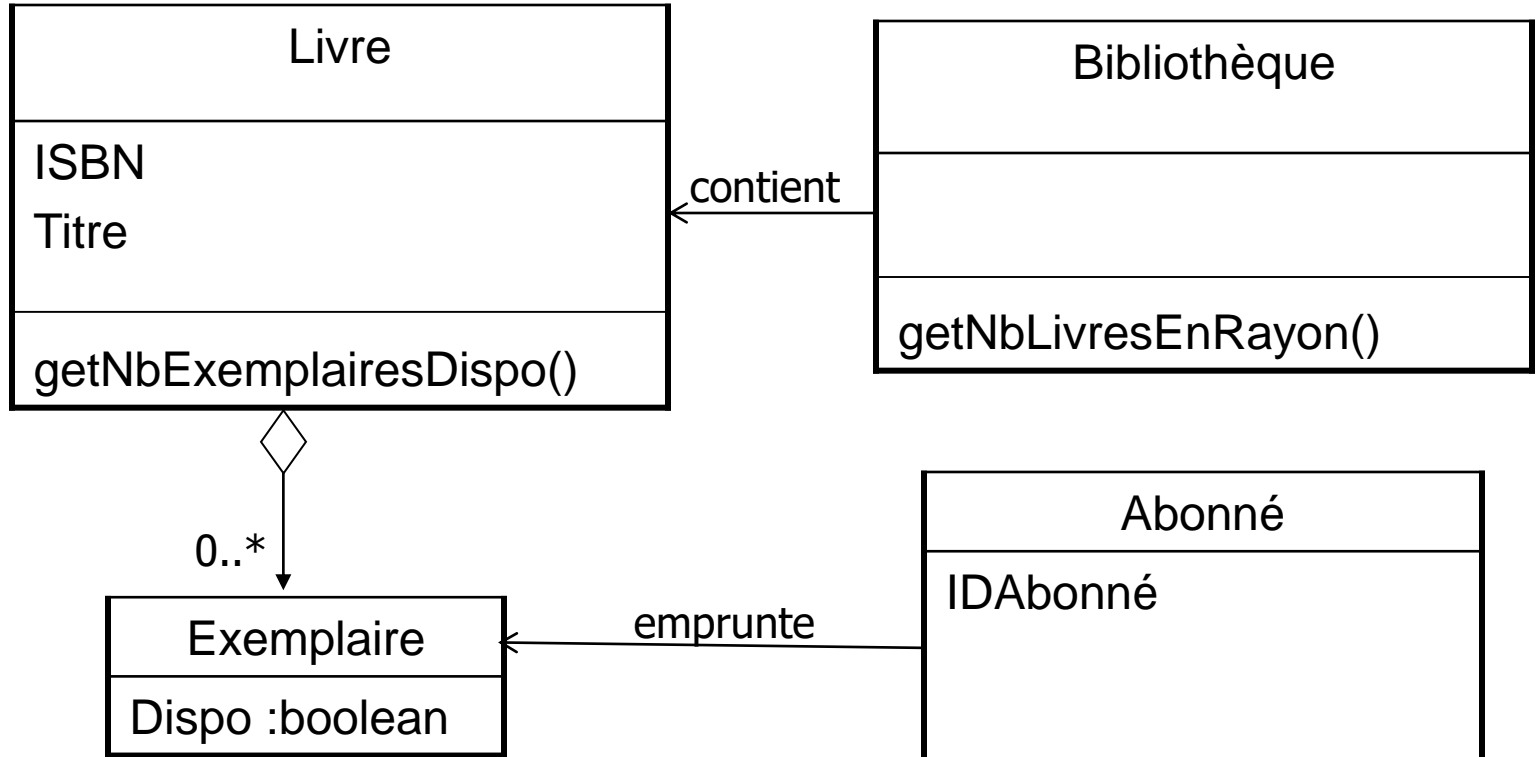
■ Tâche complexe

- Que faire quand l'accomplissement d'une responsabilité nécessite de l'information répartie entre différents objets ?

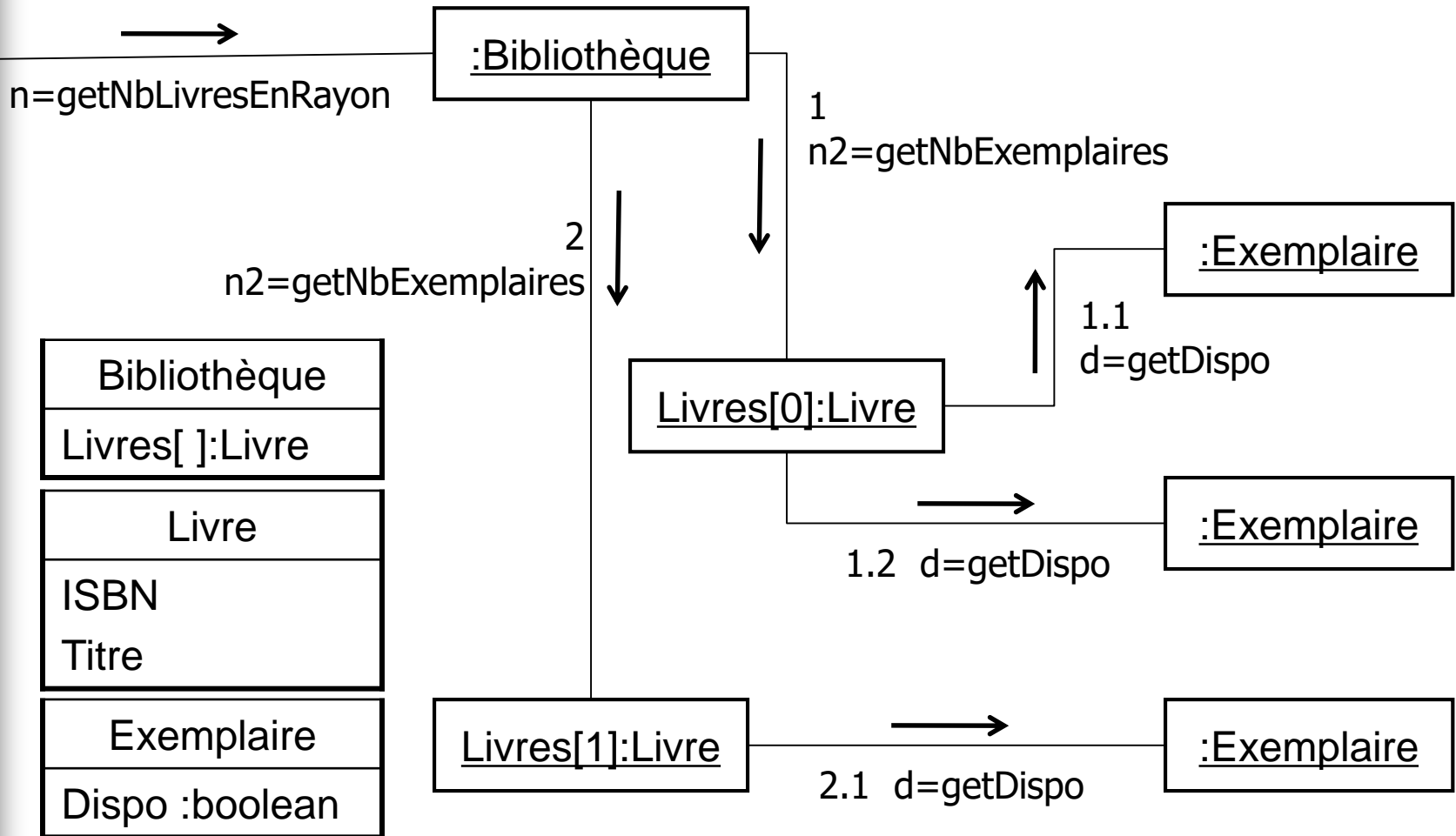
■ Solution : décomposer la tâche

- 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 partiels : 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)

(Creator)

■ Problème

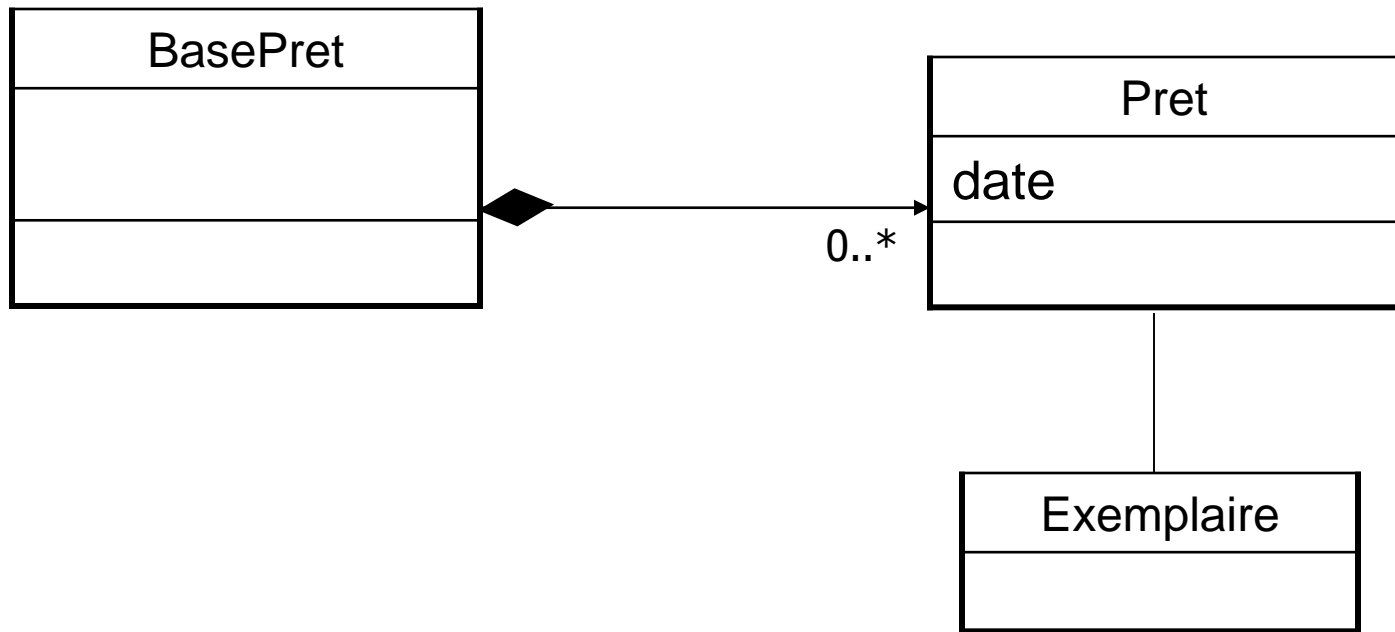
- Qui doit avoir la responsabilité de créer une nouvelle instance d'une classe donnée ?

■ 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*
 - *Fabrique, Monteur...*

Faible couplage (GRASP)

(Low coupling)

■ 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)

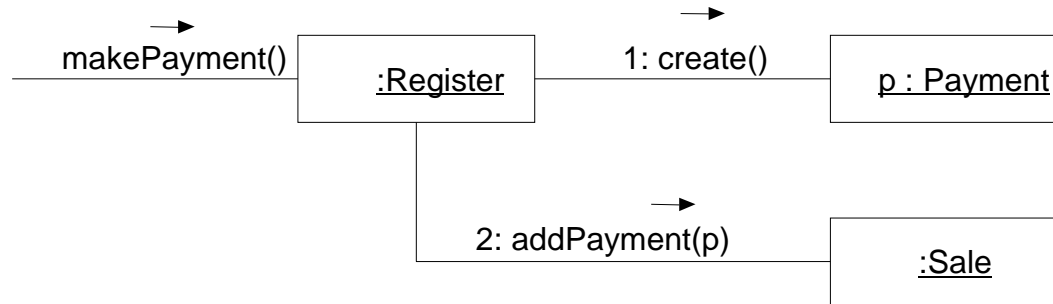
■ Principe général

- Les classes, très génériques et très réutilisables par nature, doivent avoir un faible couplage

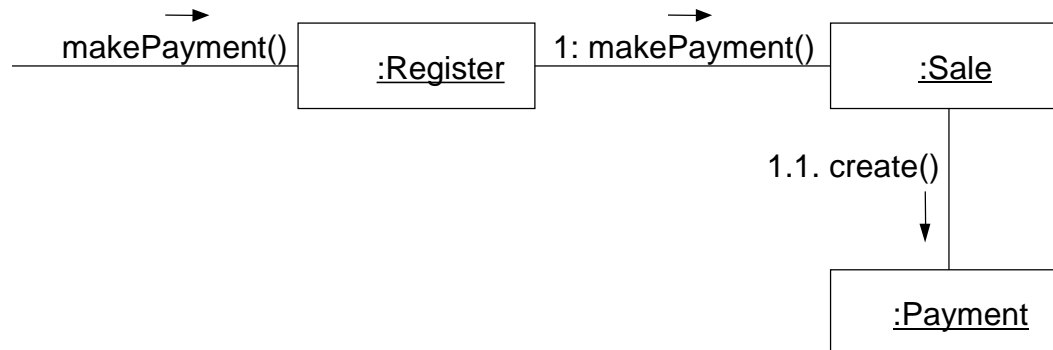
■ Mise en œuvre

- 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

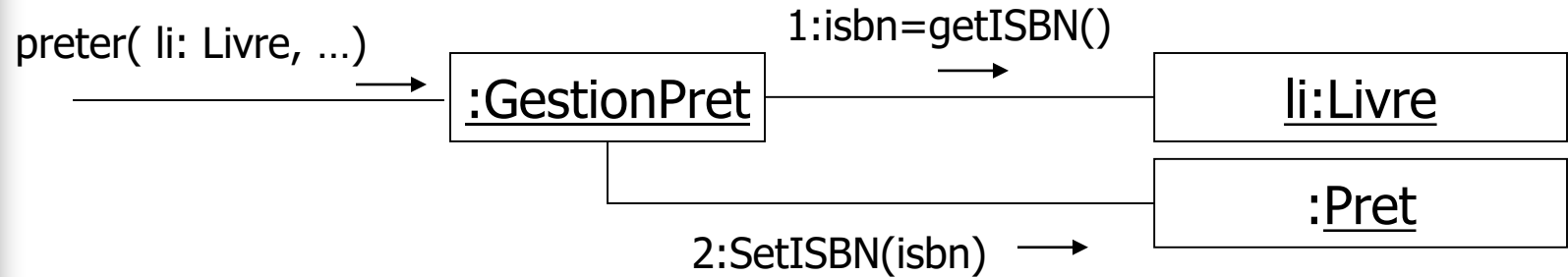


Que choisir ?

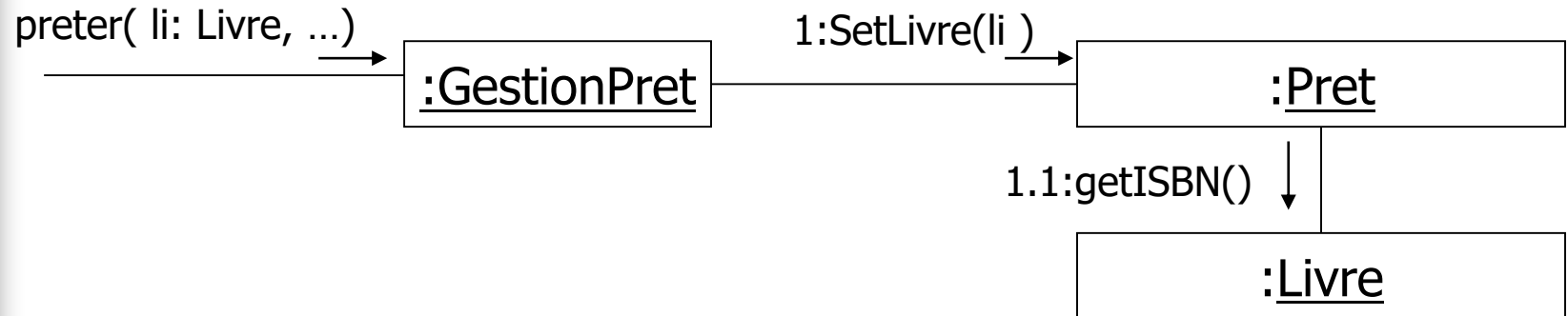


Faible couplage : autre exemple

- Pour l'application de bibliothèque, il faut mettre l'ISBN d'un Exemple 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)

(High cohesion)

- 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

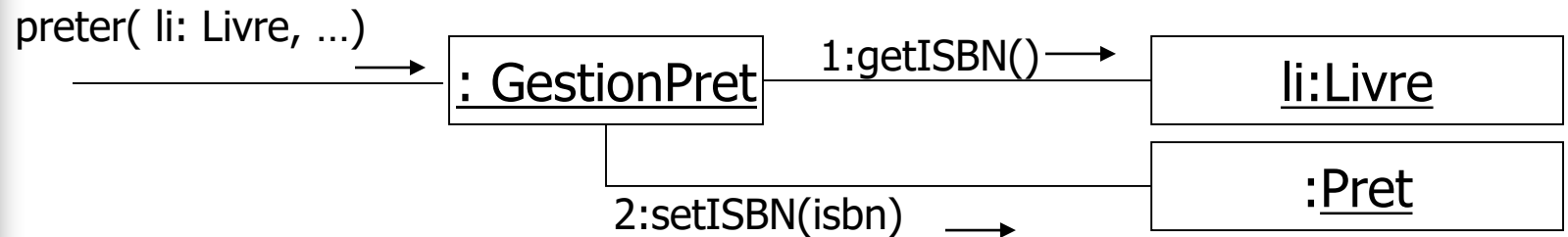
- trop de tâches
- des tâches sans lien entre elles

– Elles sont

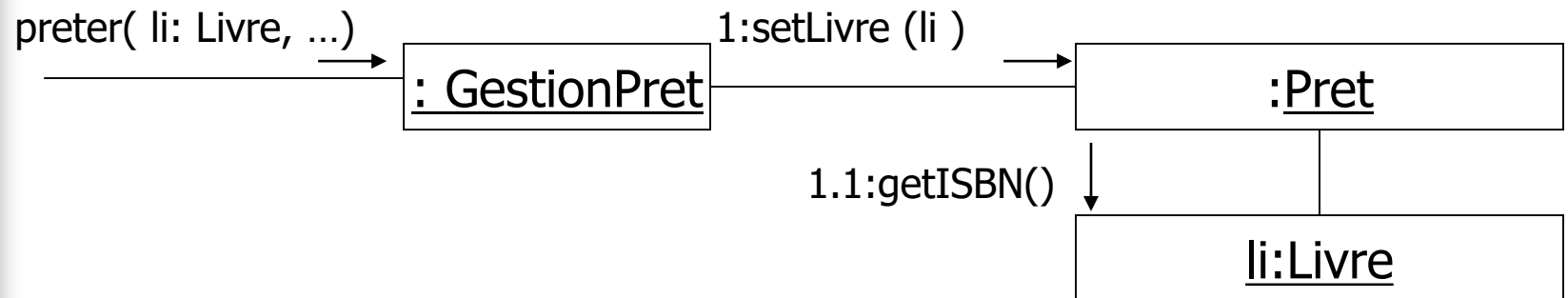
- difficiles à comprendre
- difficiles à réutiliser
- difficiles à maintenir
- fragiles, constamment affectées par le changement

■ 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 (e.g. 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** »

Fabrication pure (GRASP)

(Pure fabrication)

■ Problème

– Que faire

- pour respecter le Faible couplage et la Forte cohésion
- quand aucun concept du monde réel (objet du domaine) n'offre de solution satisfaisante ?

■ 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 (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

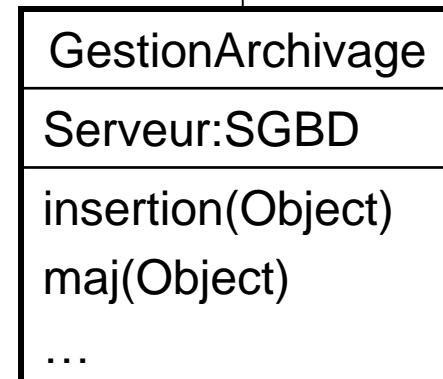
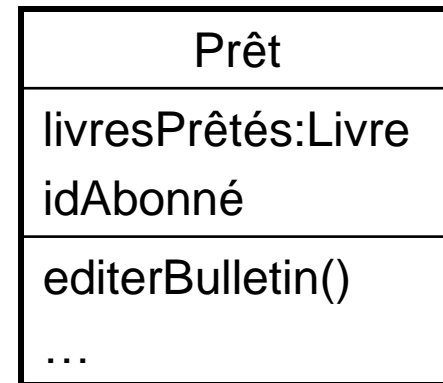
- l'enregistrement d'objets dans une BD est une tâche générique utilisable par de nombreux objets
 - pas de réutilisation, beaucoup de duplication

■ 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, Contrôleur, 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

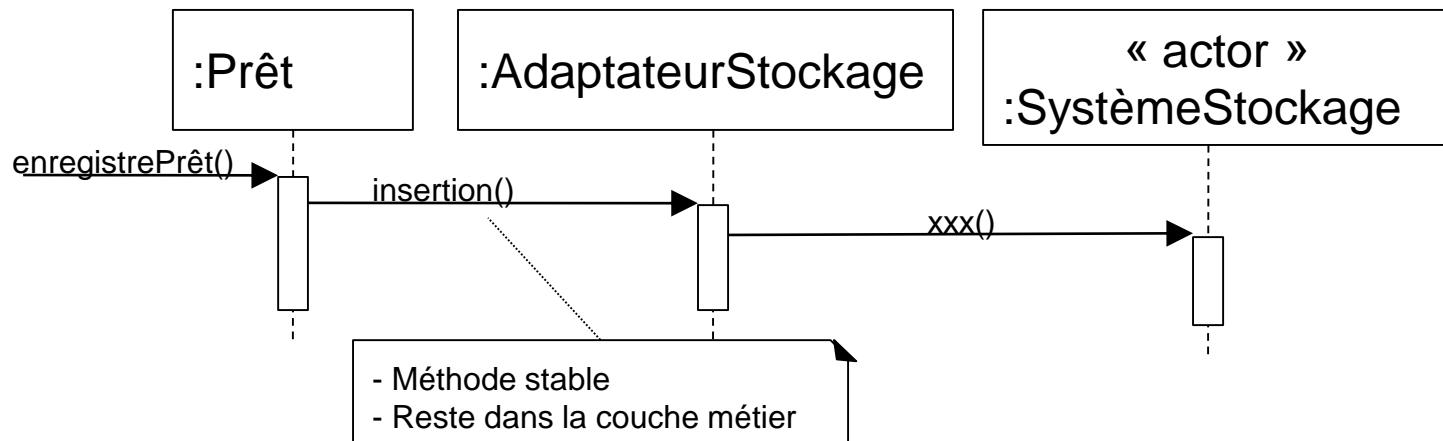
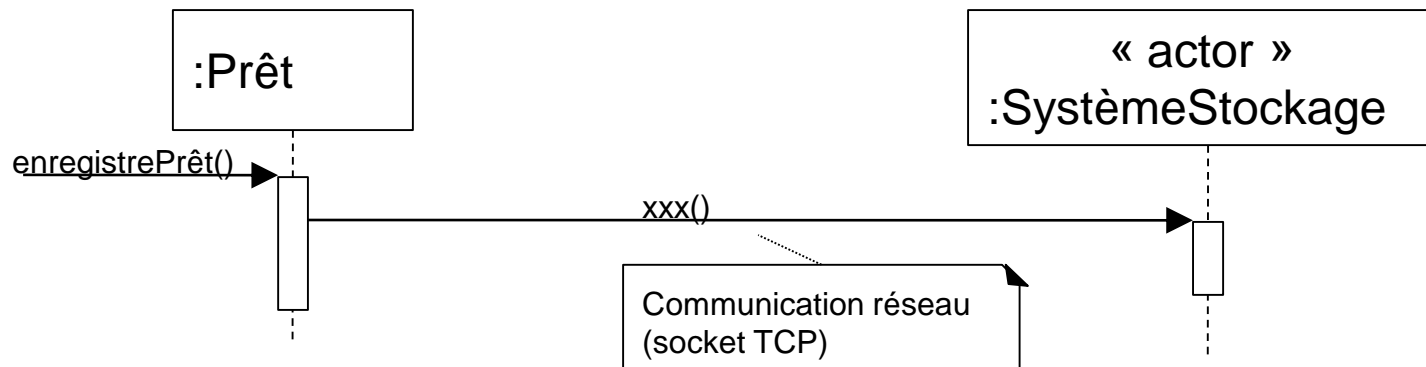
- Réaliser des adaptateurs, façades, etc. (pattern Protection des variations) qui s'interfacent avec des systèmes extérieurs
 - Exemples : proxys, DAO, ORB
- Réaliser des *inversions de dépendances* entre packages

■ 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 raisons 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)

(Protected variations)

■ Problème

Comment concevoir des objets, systèmes, sous-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 identifiés 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

(Don't talk to strangers)

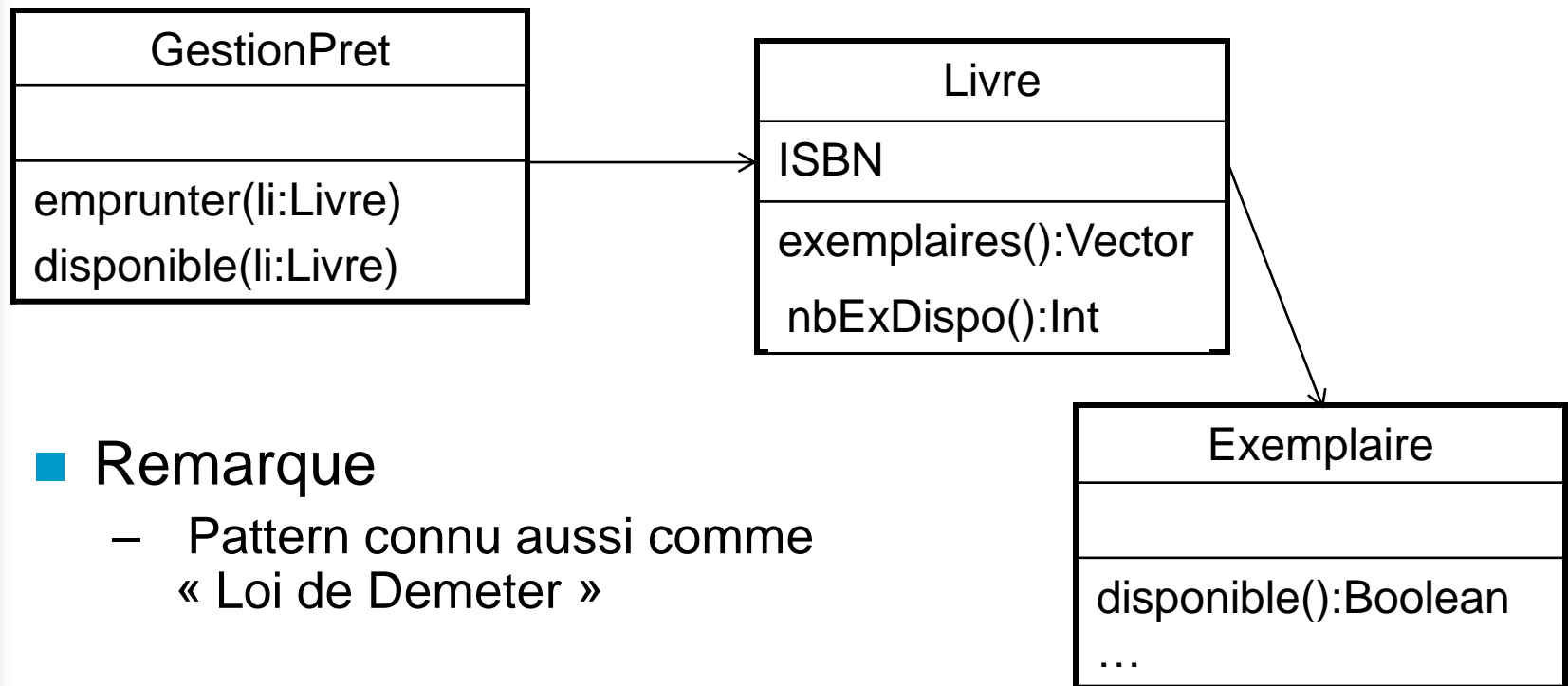
- Cas particulier de Protection des variations
 - protection contre les variations liées aux évolutions de structure des objets
- Problème
 - Si un client utilise un service ou obtient de l'information d'un *objet indirect* (inconnu) *via* un *objet direct* (familier du client), comment le faire sans couplage ?
- 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 »

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

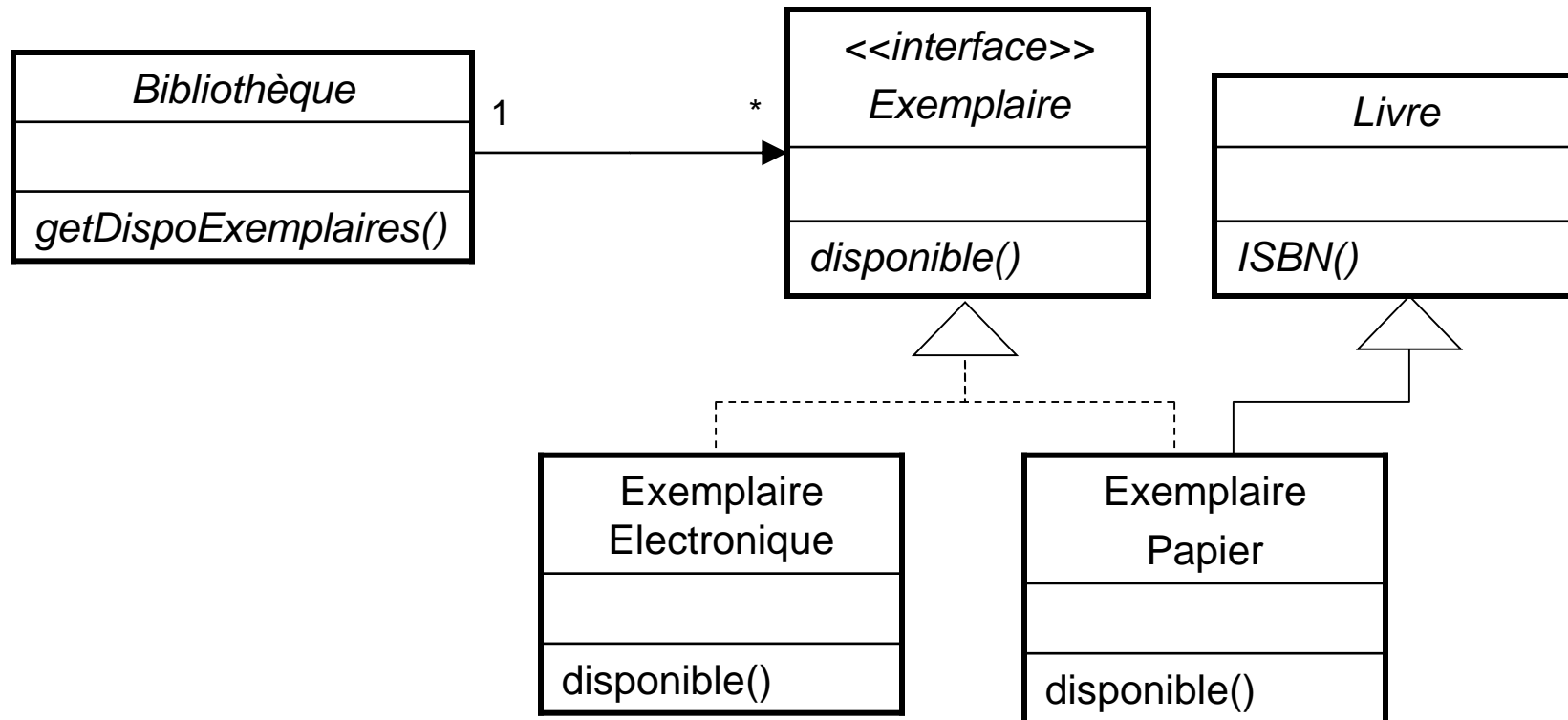
- Tirer avantage de l'approche OO en sous-classant les opérations dans des types dérivés de l'Expert en information
 - L'opération nécessite à la fois des informations et un comportement particuliers

■ 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 d'évolution pour d'éventuels cas particuliers futurs

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

■ Principe lié

- Design by Contract (SOLID)

Contrôleur (GRASP)

(Controller)

■ 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

- 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
- il n'est donc pas modélisé en tant qu'objet du domaine → Fabrication pure

■ Règle d'or

- Les opérations système des CU sont les messages initiaux qui parviennent au contrôleur dans les diagrammes d'interaction entre objets du 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

1. Le contrôleur représente le système global
:ControleurBiblio
2. 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 Façade trop chargé (pas bon)

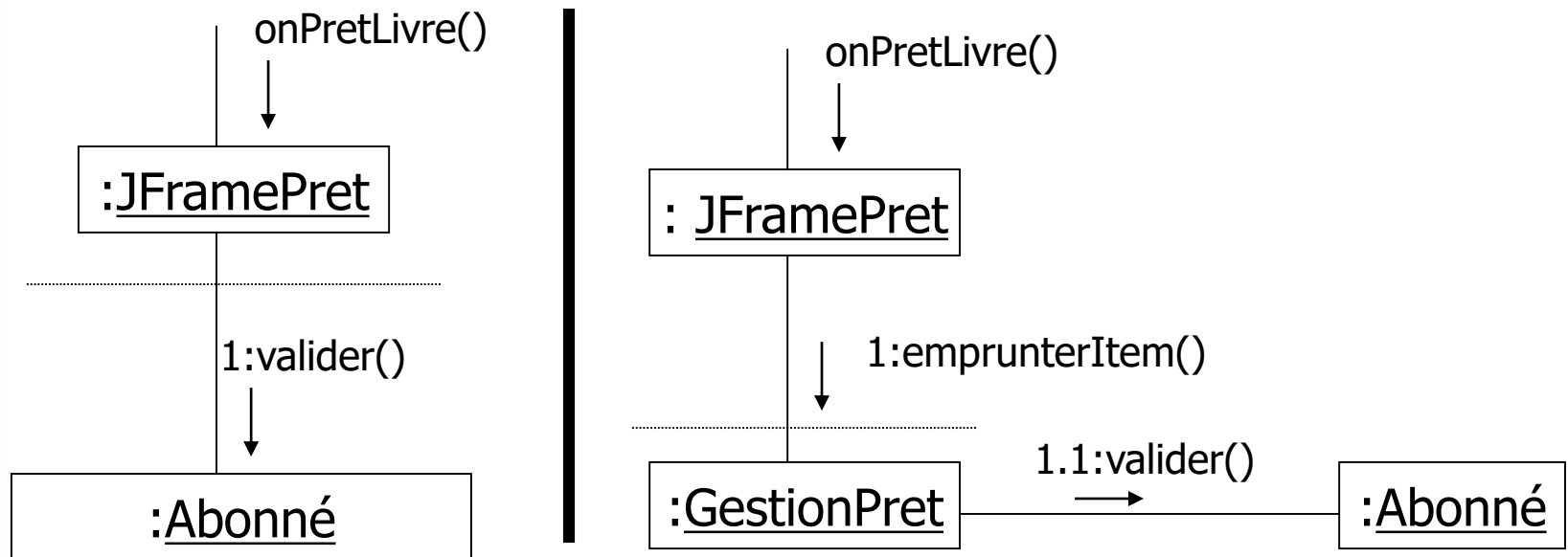
- Pas de focus, prend en charge de nombreux domaines de responsabilité
 - un seul contrôleur reçoit tous les événements système
 - le contrôleur effectue la majorité des tâches nécessaires pour répondre aux événements système
 - un contrôleur doit déléguer à d'autres objets les tâches à effectuer
 - 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

Contrôleur de cas d'utilisation (contrôleur délégué)

- 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
- Élément artificiel : ce n'est pas un objet du domaine

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 métier et 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-Vue
 - ➔ Brique de base pour une conception modulaire
- Meilleure « architecturation » des CU

■ Patterns liés

- Indirection, Couches, Façade, Fabrication pure, Commande

Les patterns GRASP et les autres

- D'une certaine manière, tous les autres patterns sont
 - des applications,
 - des spécialisations,
 - des utilisations conjointesdes 9 patterns GRASP, qui sont les plus généraux.



Plan

- Introduction
- Principes GRASP
- Design patterns
- Patterns architecturaux
- Conclusion

Définition

- Bonnes pratiques de combinaison d'un ensemble de modules, d'objets ou de classes
 - Réutilisabilité
 - Maintenabilité
 - Vocabulaire commun
- Portée
 - Met en scène plusieurs éléments (différence GRASP)
 - Résout un problème localisé à un contexte restreint (différence architecture)
- Vocabulaire
 - Instances, rôles, collaboration

Catégories de design patterns

■ Création

- Processus d'instanciation / initialisation des objets

■ Structure

- Organisation d'un ensemble de classes à travers un module (statique)

■ Comportement

- Organisation des rôles pour la collaboration d'objets (dynamique)

Source : http://fr.wikipedia.org/wiki/Patron_de_conception



Patterns de création

- Singleton (Singleton)
- Fabrique (Factory Method)
- Fabrique abstraite (Abstract Factory)
- Monteur (Builder)
- Prototype (Prototype)

Singleton

■ Objectif

- S'assurer d'avoir une instance unique d'une classe
 - Point d'accès unique et global pour les autres objets
 - Exemple : Factory

■ Fonctionnement

- Le constructeur de la classe est privé (seules les méthodes de la classe peuvent y accéder)
- l'instance unique de la classe est stockée dans une variable statique privée
- Une méthode publique statique de la classe
 - Crée l'instance au premier appel
 - Retourne cette instance

Singleton

Singleton	
-	<u>singleton : Singleton</u>
-	Singleton()
+	<u>getInstance() : Singleton</u>

Source :

[http://fr.wikipedia.org/wiki/Singleton_\(patron_de_conception\)](http://fr.wikipedia.org/wiki/Singleton_(patron_de_conception))

Notion de Fabrique (Factory)

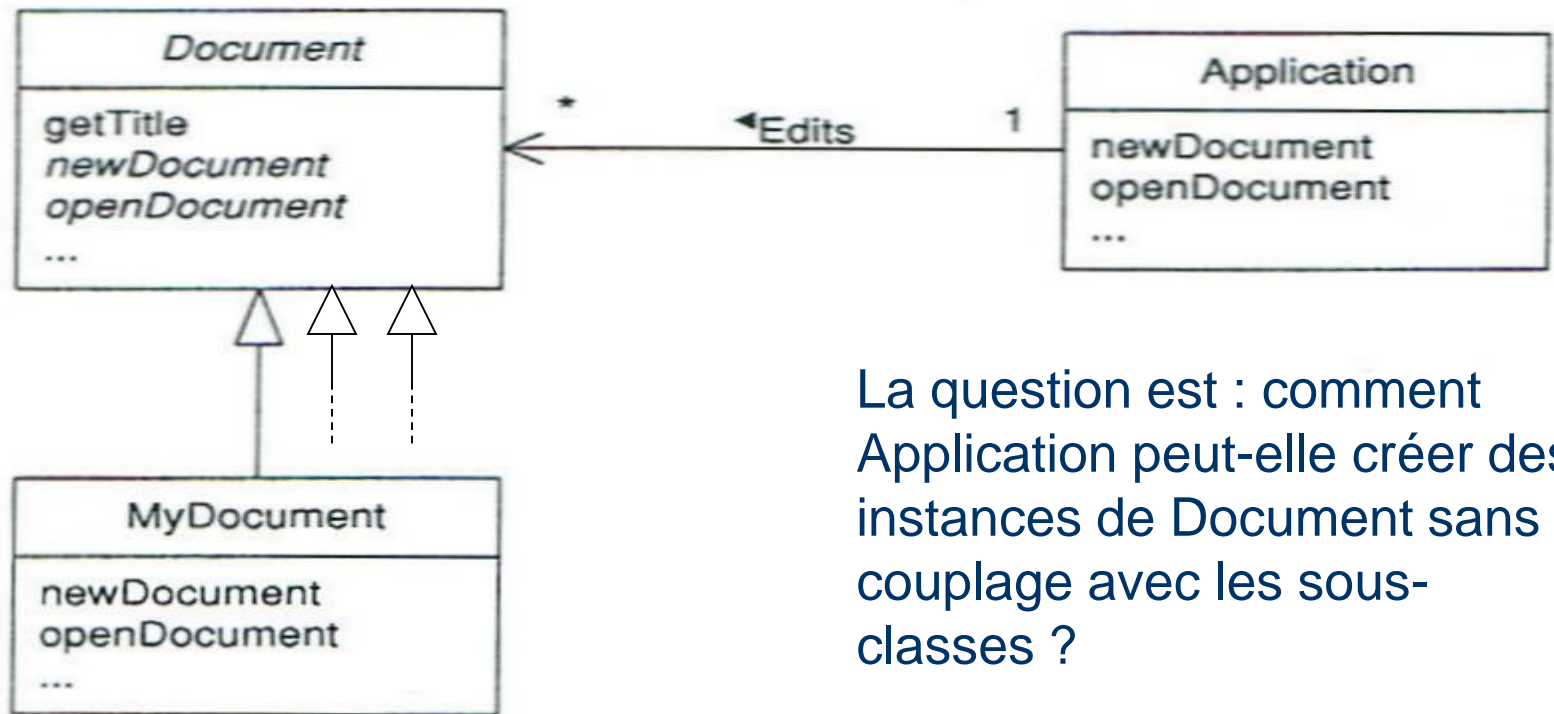
- Classe responsable de la création d'objets
 - lorsque la logique de création est complexe
 - lorsqu'il convient de séparer les responsabilités de création
- Fabrique concrète = objet qui fabrique des instances
- Avantages par rapport à un constructeur
 - la classe a un nom
 - permet de gérer facilement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

Factory method

■ 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



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

FIGURE 5.1 Application framework.

(From Grand's book.)

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

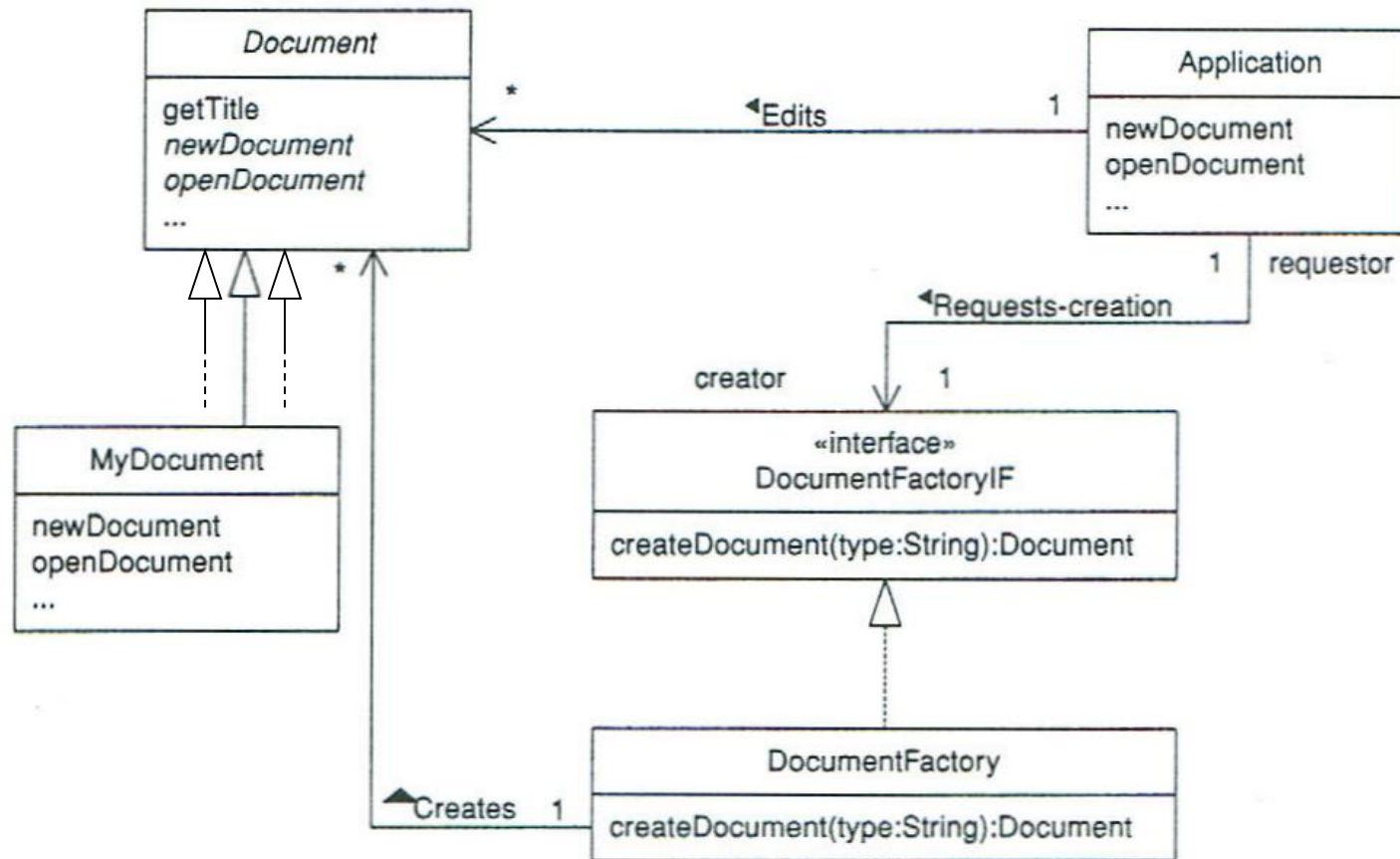


FIGURE 5.2 Application framework with document factory.

(From Grand's book.)

Factory Method Pattern : structure générale

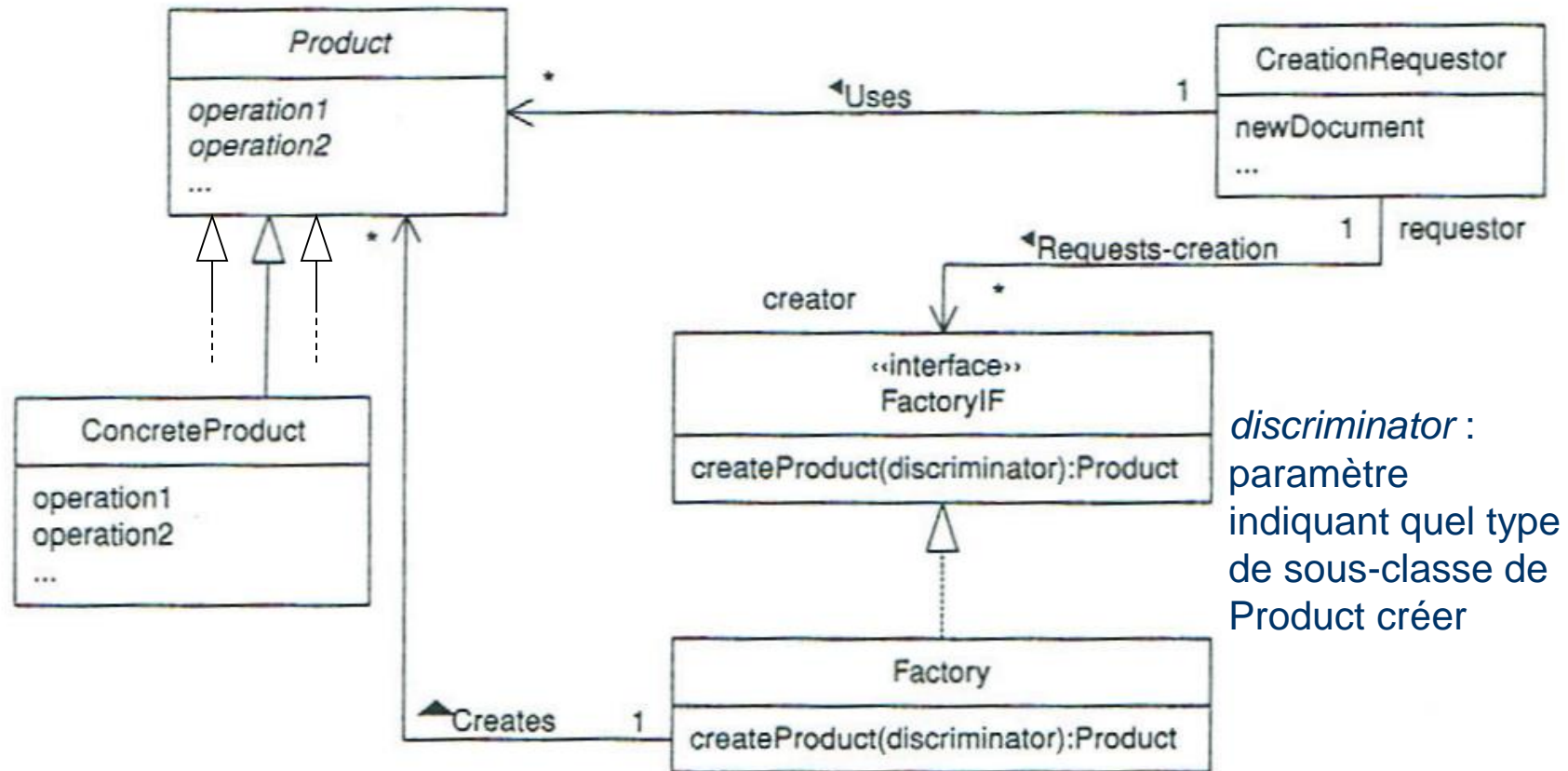


FIGURE 5.3 Factory method pattern.

(From Grand's book.)

Abstract Factory

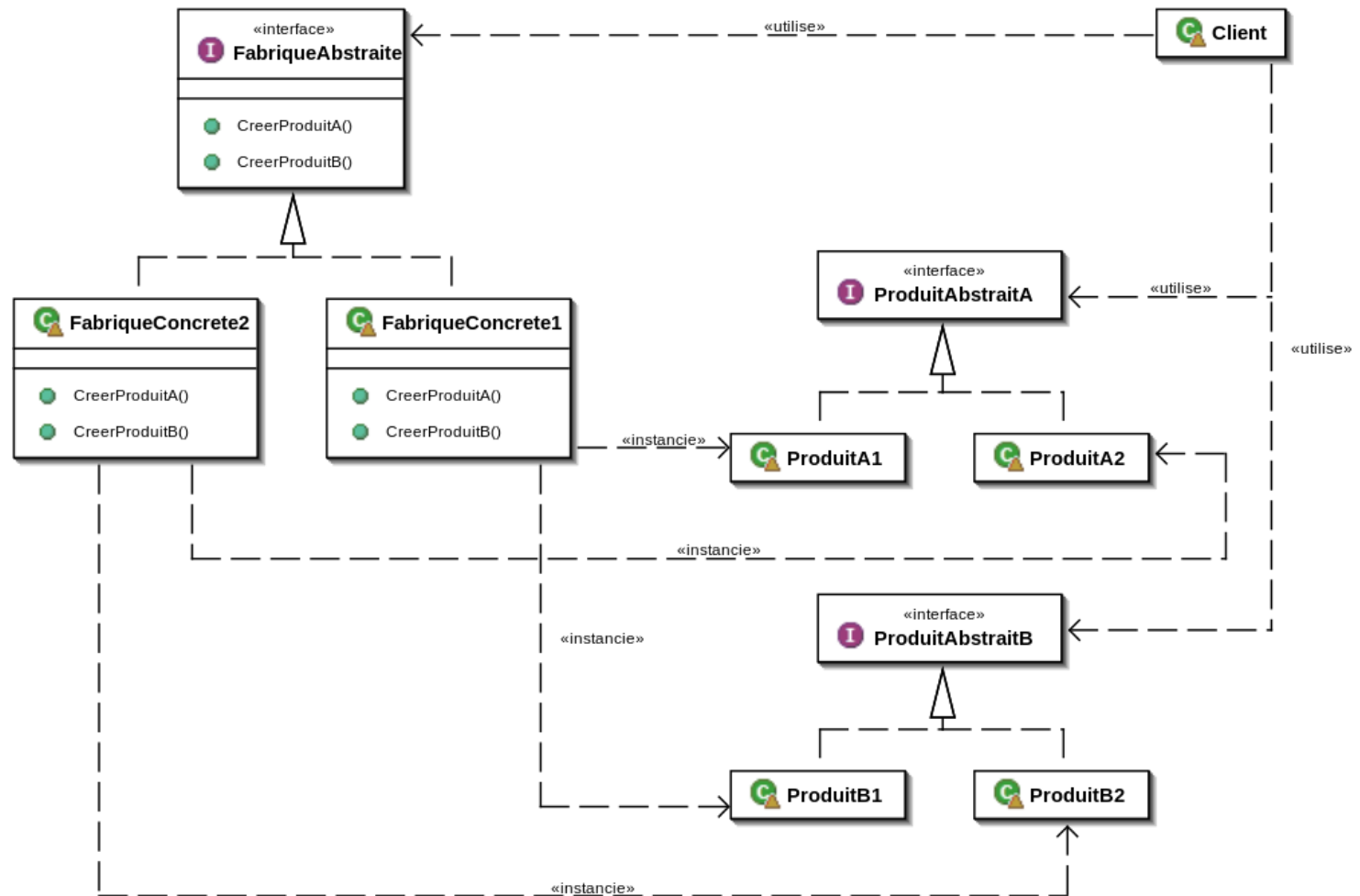
■ Objectif

- Création de familles d'objets
- Généralisation du pattern Factory Method

■ Fonctionnement : « fabrication de fabriques »

- Regroupe plusieurs Factories en une fabrique abstraite
- Le client ne connaît que l'interface de la fabrique abstraite
- Il invoque différentes méthodes qui sont déléguées à différentes fabriques concrètes

Abstract Factory



Source : [http://fr.wikipedia.org/wiki/Fabrique_abstraite_\(patron_de_conception\)](http://fr.wikipedia.org/wiki/Fabrique_abstraite_(patron_de_conception))

Monteur (Builder)

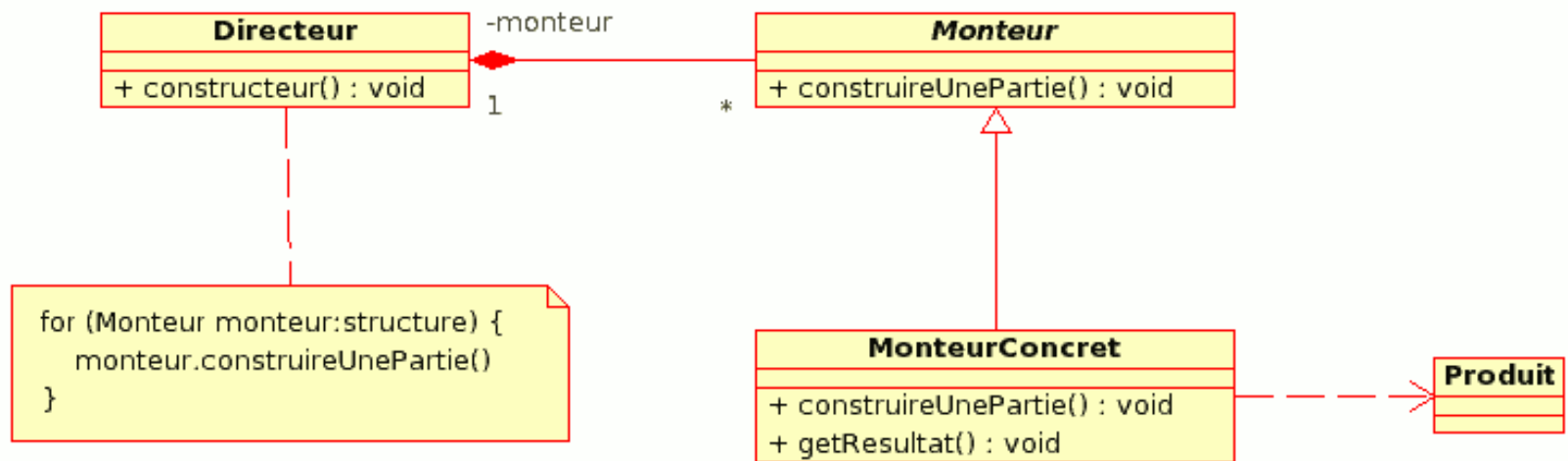
■ Objectif

- Instancier et réaliser la configuration initiale d'un objet en s'abstrayant de l'interface de l'objet
- Fournir une instance à un client

■ Remarques

- S'applique en général à des objets complexes
- Différence avec le pattern [Abstract] Factory
 - Plutôt utilisé pour la configuration que pour la gestion du polymorphisme

Builder



Source :

http://commons.wikimedia.org/wiki/File:Monteur_classes.png

Prototype

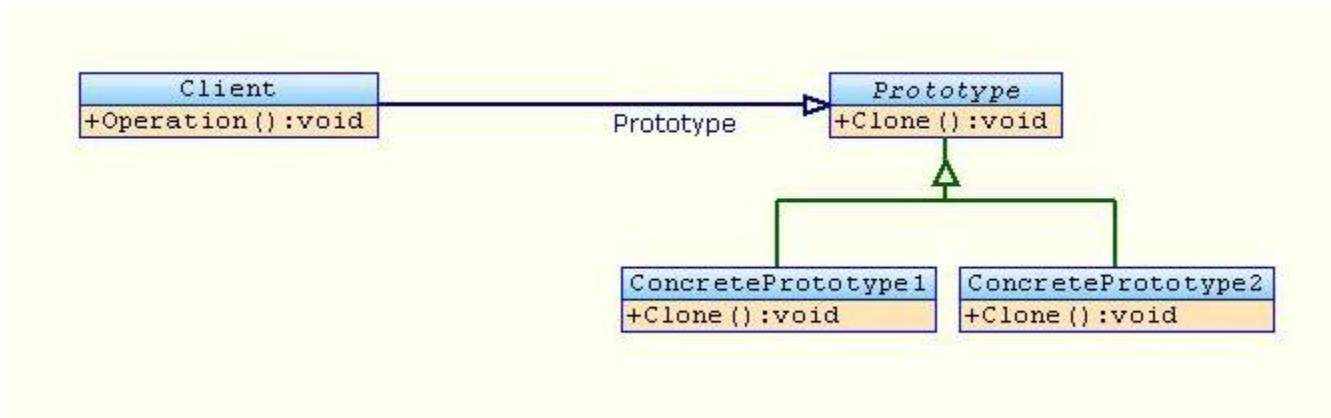
■ Objectifs

- Réutiliser un comportement sans recréer une instance
 - Économie de ressources

■ Fonctionnement

- Recopie d'une instance existante (méthode `clone()`)
- Ajout de comportements spécifiques :
« polymorphisme à pas cher »

Prototype



Source :

[http://fr.wikipedia.org/wiki/Prototype_\(patron_de_conception\)](http://fr.wikipedia.org/wiki/Prototype_(patron_de_conception))

■ Remarque

- Implémentation choisie pour l'héritage en JavaScript (pas de classes)

Patterns de structure

- Objet composite (Composite)
- Adaptateur (Adapter)
- Façade (Facade)
- Proxy (Proxy)
- Décorateur (Decorator)

Composite

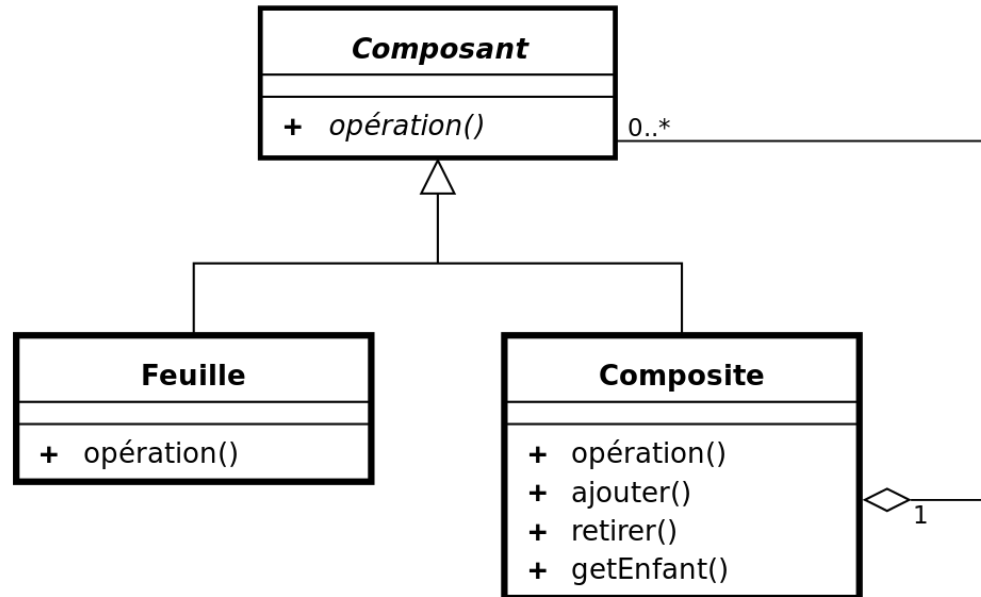
■ Objectif

- Représenter une structure arborescente d'objets
- Rendre générique les mécanismes de positionnement / déplacement dans un arbre
 - Exemple : DOM Node

■ Fonctionnement

- Une classe abstraite (Composant) qui possède deux sous-classes
 - Feuille
 - Composite : contient d'autres composants

Composite



Source : http://fr.wikipedia.org/wiki/Objet_composite

■ Remarque

- Pourquoi une relation d'agrégation et non de composition ?

Adaptateur (Adapter, Wrapper)

■ Objectif

- Résoudre un problème d'incompatibilité d'interfaces (API)
 - Un client attend un objet dans un format donné
 - Les données sont encapsulées dans un objet qui possède une autre interface

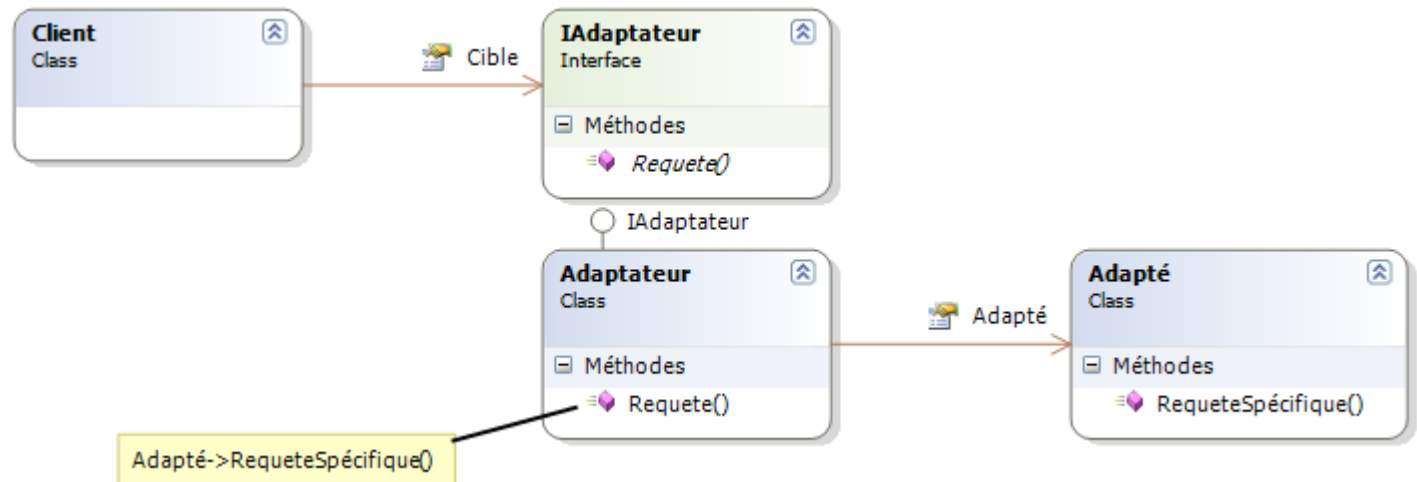
■ Fonctionnement

- Insérer un niveau d'indirection qui réalise la conversion

■ Patterns liés

- Indirection, Proxy

Adaptateur (Adapter, Wrapper)



Source :

[http://fr.wikipedia.org/wiki/Adaptateur_\(patron_de_conception\)](http://fr.wikipedia.org/wiki/Adaptateur_(patron_de_conception))

Façade

■ Objectif

- Cacher une interface / implémentation complexe
 - rendre une bibliothèque plus facile à utiliser, comprendre et tester;
 - rendre une bibliothèque plus lisible;
 - réduire les dépendances entre les clients de la bibliothèque

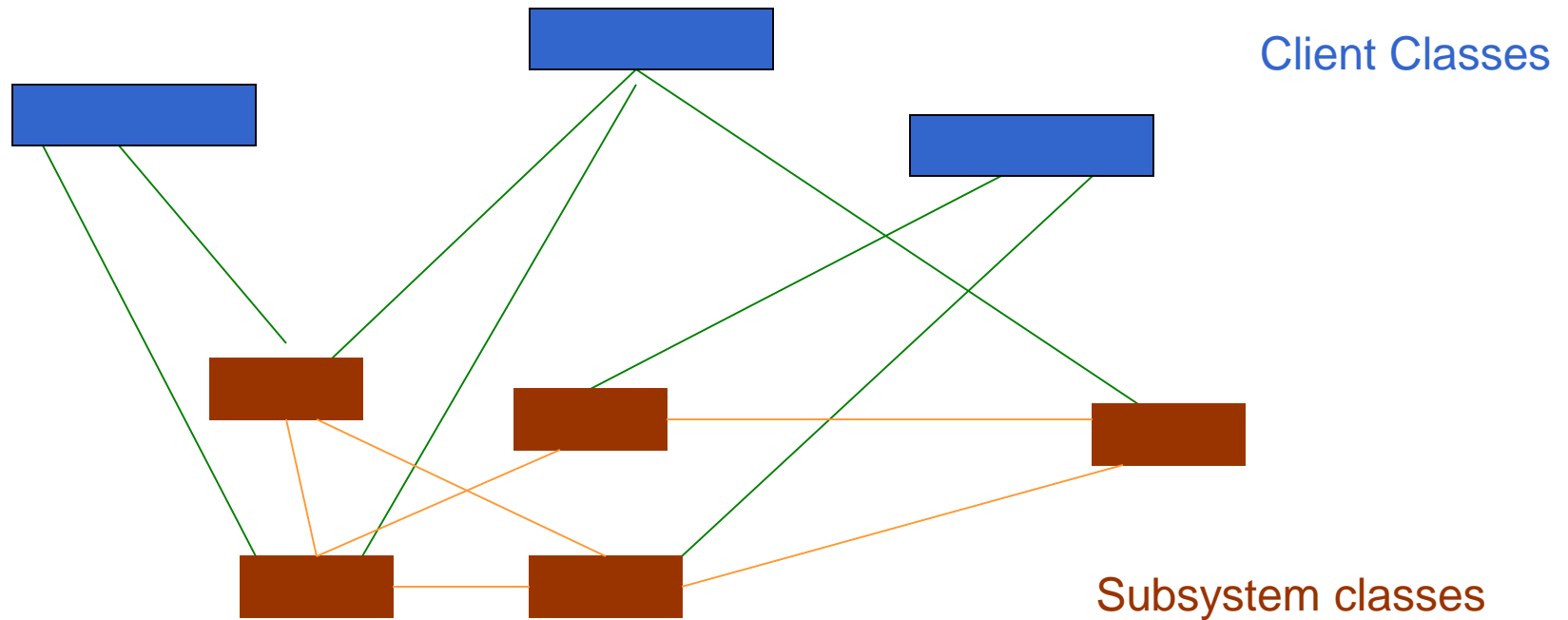
■ Fonctionnement

- Fournir une interface simple regroupant toutes les fonctionnalités utiles aux clients

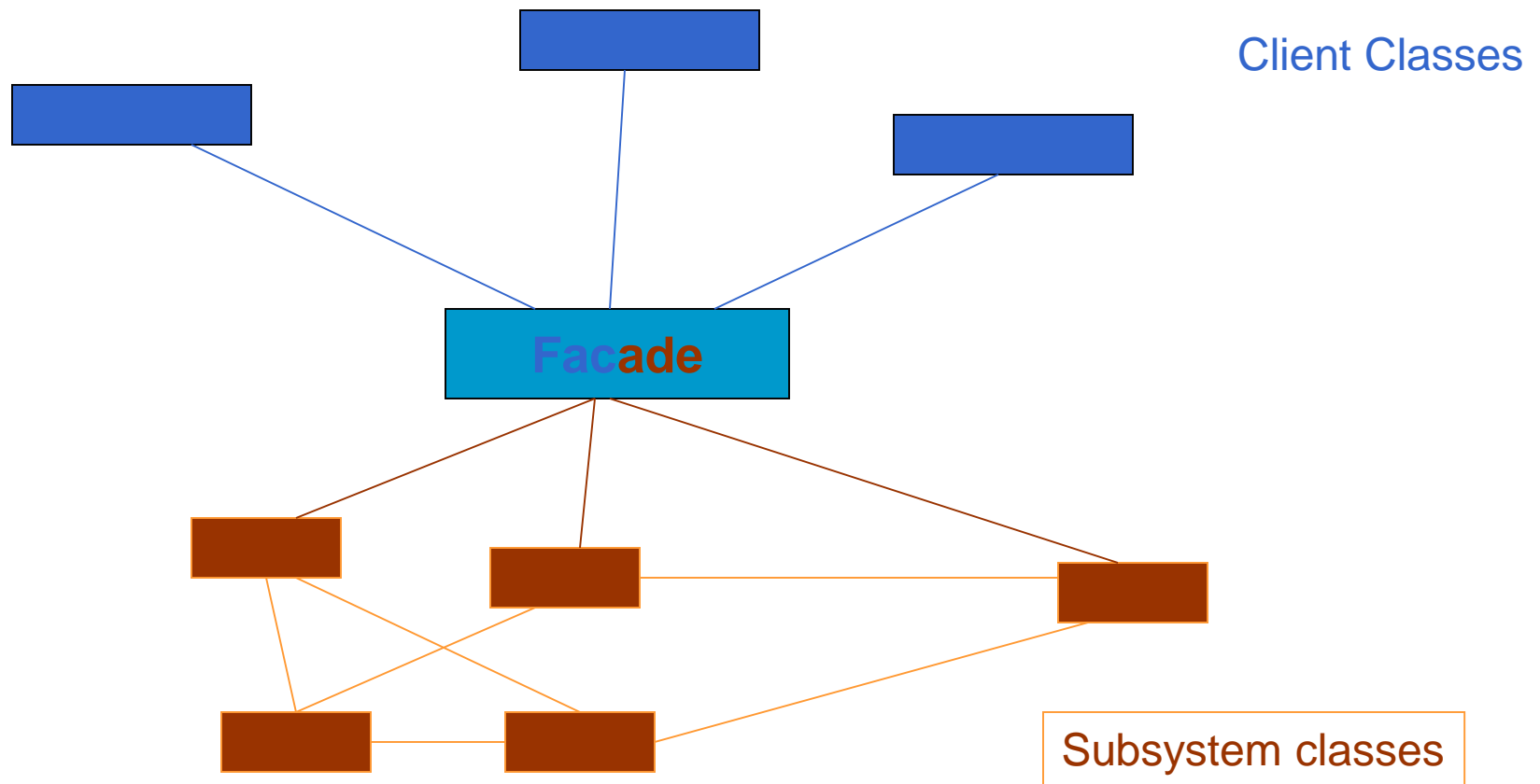
■ Patterns liés

- Indirection, Adaptateur

Façade



Façade : solution





Proxy

■ Objectif

- Résoudre un problème d'accès à un objet
 - À travers un réseau
 - Qui consomme trop de ressources...

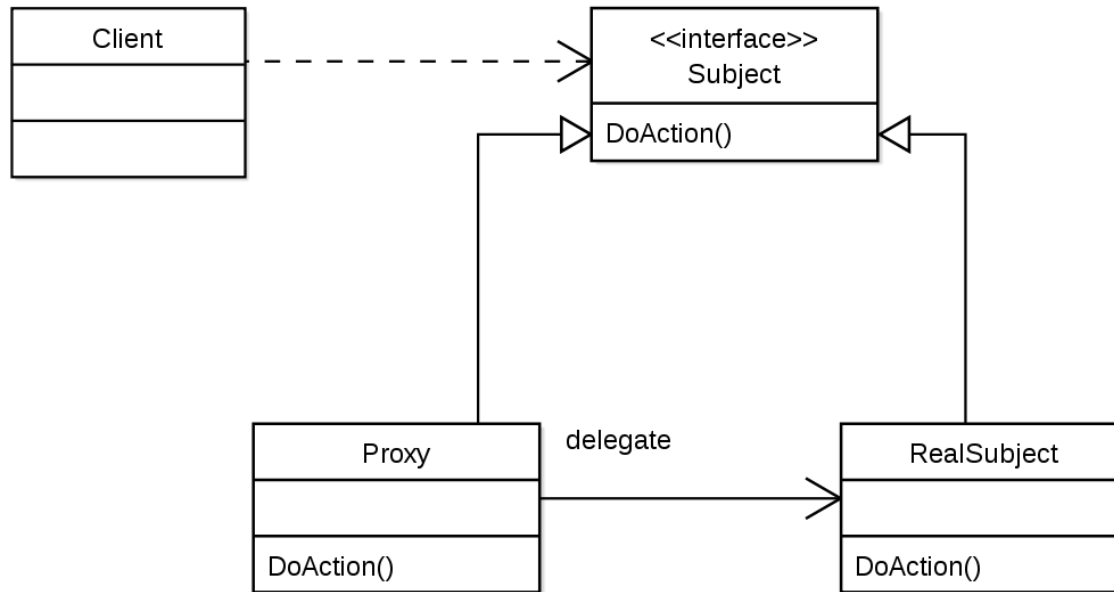
■ Fonctionnement

- Créer une classe qui implémente la même interface
- La substituer à la classe recherchée auprès du client

■ Patterns liés

- Indirection, État, Décorateur

Proxy



Source : http://en.wikipedia.org/wiki/Proxy_pattern

Décorateur

■ Objectif

- Résister au changement
 - Principe général :

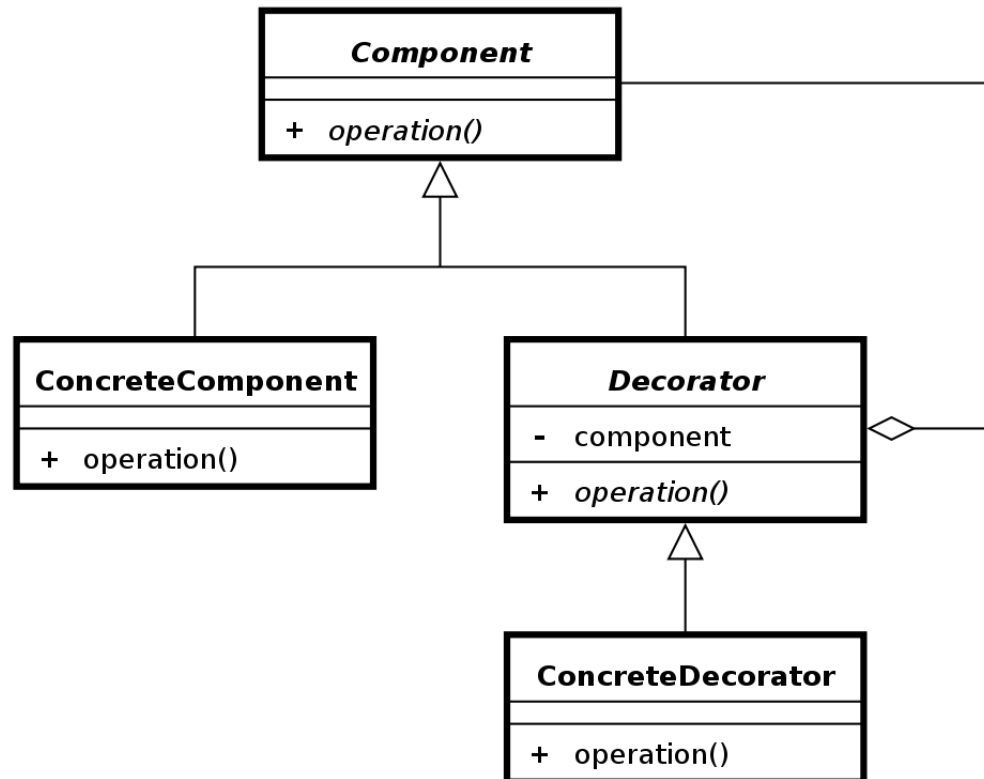
Les classes doivent être ouvertes à l'extension,
mais fermées à la modification

- Permettre l'extension des fonctionnalités d'une application sans tout reconcevoir

■ Fonctionnement

- Rajouter des comportements dans une classe qui possède la même interface que celle d'origine
- Appeler la classe d'origine depuis le décorateur
- Effectuer des traitements « autour » de cet appel

Décorateur



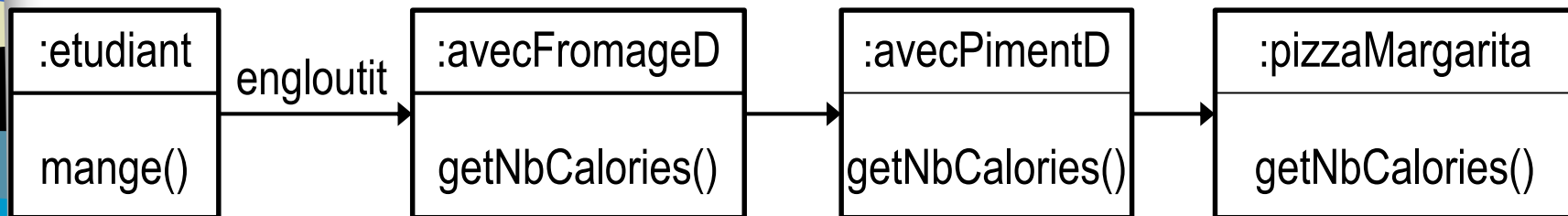
Source : http://en.wikipedia.org/wiki/Decorator_pattern

Décorateur

- Utilisation courante

- Rajouter un comportement à un comportement existant

- Exemple



- Pattern lié

- Proxy

- Pattern antagoniste

- Polymorphisme

Patterns de comportement

- Interpréteur (Interpreter)
- Memento (Memento)
- État (State)
- Stratégie (Strategy)
- Visiteur (Visitor)
- Commande (Command)
- Chaîne de responsabilité (Chain of responsibility)
- Observateur (Observer)
- Fonction de rappel (Callback)
- Promesse (Promise)

Interpréteur

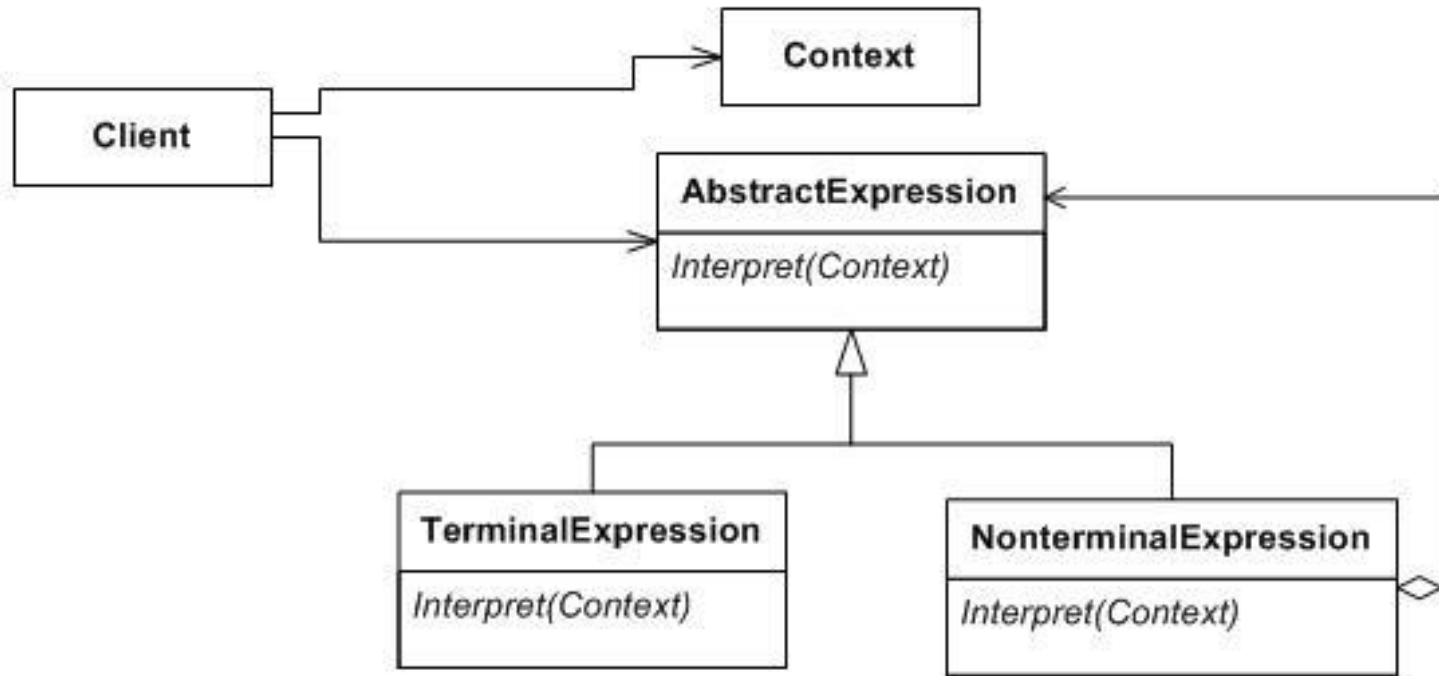
■ Objectif

- Évaluer une expression dans un langage particulier
 - Exemples : expressions mathématiques, SQL...

■ Fonctionnement

- Stocker l'expression dans un « contexte » (pile)
- Définir les classes de traitement terminales et non terminales, à l'aide de la même interface

Interpréteur



Source : http://en.wikipedia.org/wiki/Interpreter_pattern

Memento

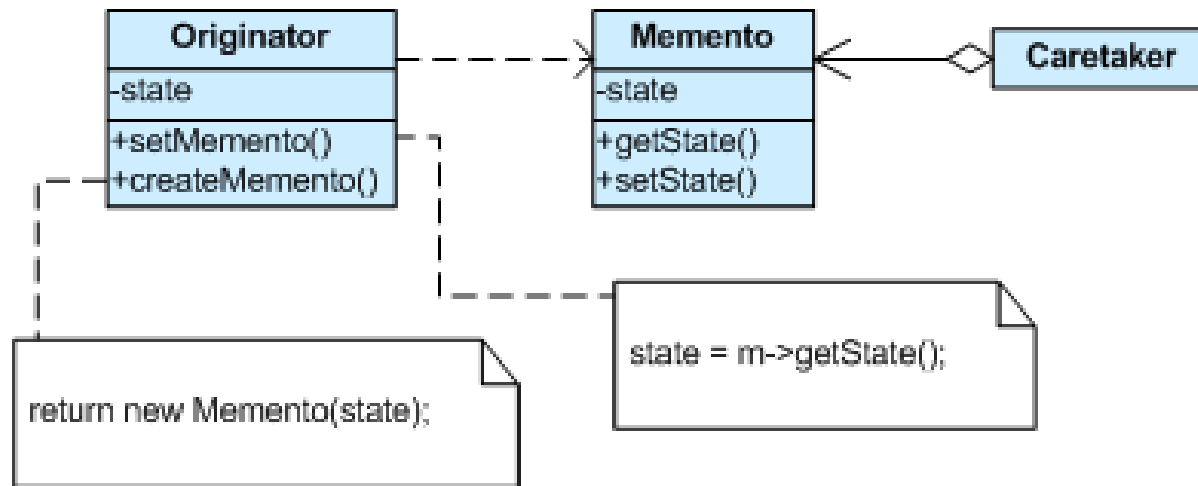
■ Objectif

- Restaurer un état précédent d'un objet sans violer le principe d'encapsulation (pas d'attributs publics)

■ Fonctionnement

- Sauvegarder les états de l'objet d'origine (Originator) dans un autre objet : Memento
- Transmettre ce Memento à un « gardien » (CareTaker) pour son stockage
 - Memento doit être opaque pour le CareTaker, qui ne doit pas pouvoir le modifier
- Ajouter à l'Originator des méthodes de sauvegarde et de restauration

Memento



Source :

http://sourcemaking.com/design_patterns/memento

État (State)

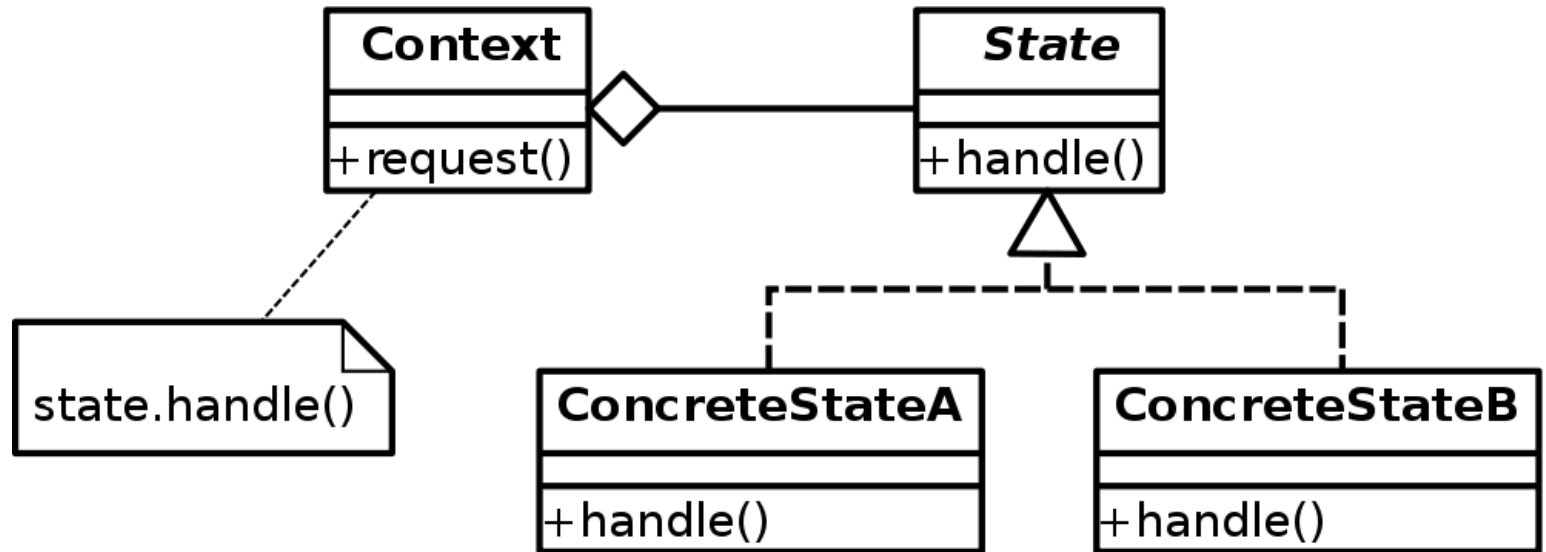
■ Objectif

- Changer le comportement apparent d'un objet en fonction de son état
 - Généralisation des automates à états (IA)

■ Fonctionnement

- Une interface (*State*) définit le comportement
- Des *ConcreteState* implémentent les comportements
- Un *Context* stocke l'état courant et appelle les comportements correspondants
- Les *ConcreteState* peuvent changer l'état courant dans le contexte

État (State)



- Pattern lié
 - Stratégie

Source : [http://fr.wikipedia.org/wiki/État_\(patron_de_conception\)](http://fr.wikipedia.org/wiki/État_(patron_de_conception))

Stratégie

■ Objectif

- Permettre (et orchestrer) le changement dynamique de comportement d'un objet

■ Fonctionnement

- Désencapsuler les comportements de la classe mère de l'objet
- Les déporter dans des classes liées, à l'aide d'une interface commune
- Permettre au client d'utiliser une implémentation quelconque de cette interface
- Utiliser un contexte qui gère les changements d'implémentation

Stratégie

■ Principes généraux de conception

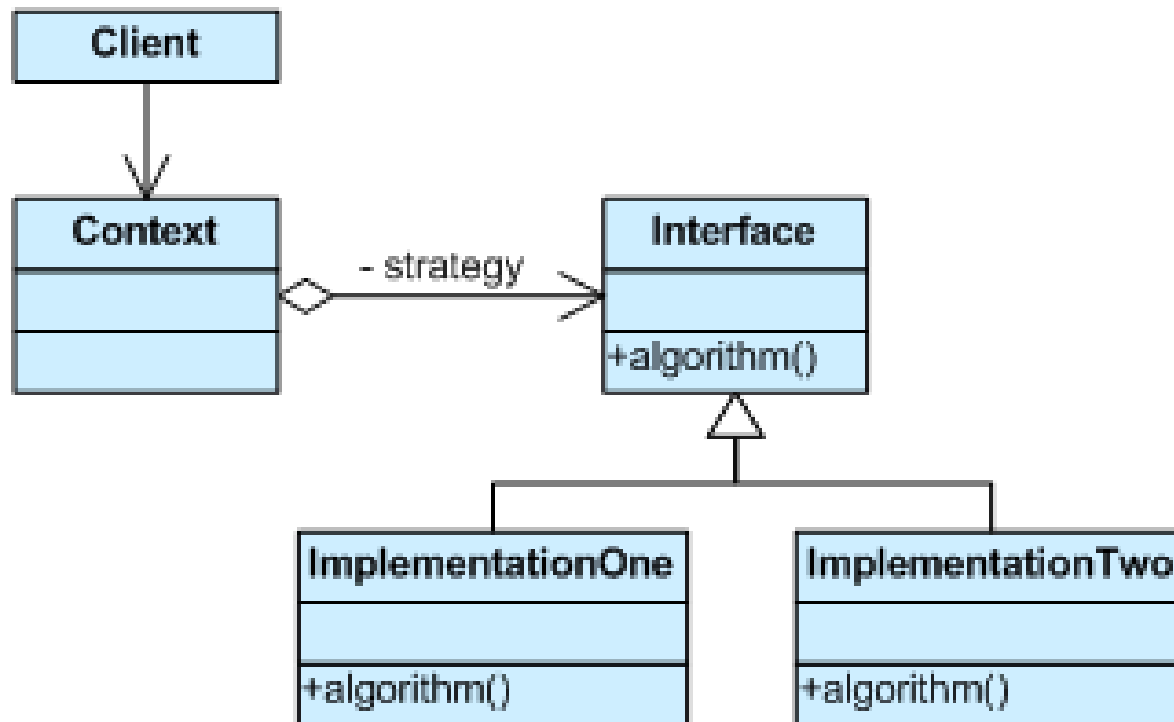
– Ouvert-fermé :

- les modules doivent être
 - Ouverts pour l'extension
 - ➔ prévoir dans l'architecture des points d'extensions
 - Fermés pour la modification
 - ➔ Le code testé n'est pas modifié
- Privilégier la relation « a un »
à la relation « est un »

■ Pattern lié

- État, Décorateur

Stratégie



Source : http://en.wikipedia.org/wiki/Strategy_pattern
http://sourcemaking.com/design_patterns/strategy

Visiteur

■ Objectif

- Séparer un comportement de la structure d'objets à laquelle il s'applique
- Ajouter de nouvelles opérations sans modifier cette structure

■ Fonctionnement

- Ajout aux classes de fonctions « virtuelles » génériques qui redirigent les opérations vers une classe spécifique « Visiteur » (Fabrication pure)
- Cette classe redirige les opérations vers les bonnes implémentations « double dispatch »

Visiteur

- Principe général de conception
 - Ouvert-fermé

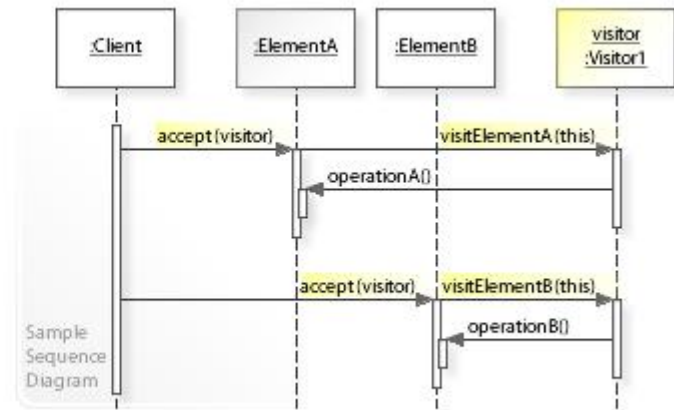
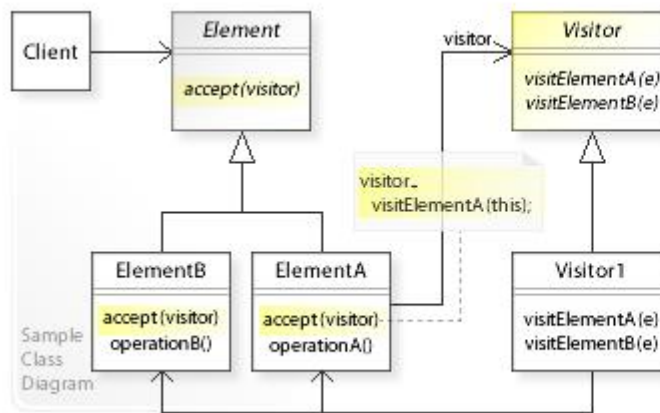


Image : By Vanderjoe - Own work, CC BY-SA 4.0,
<https://commons.wikimedia.org/w/index.php?curid=63201110>

Source : https://en.wikipedia.org/wiki/Visitor_pattern

Commande

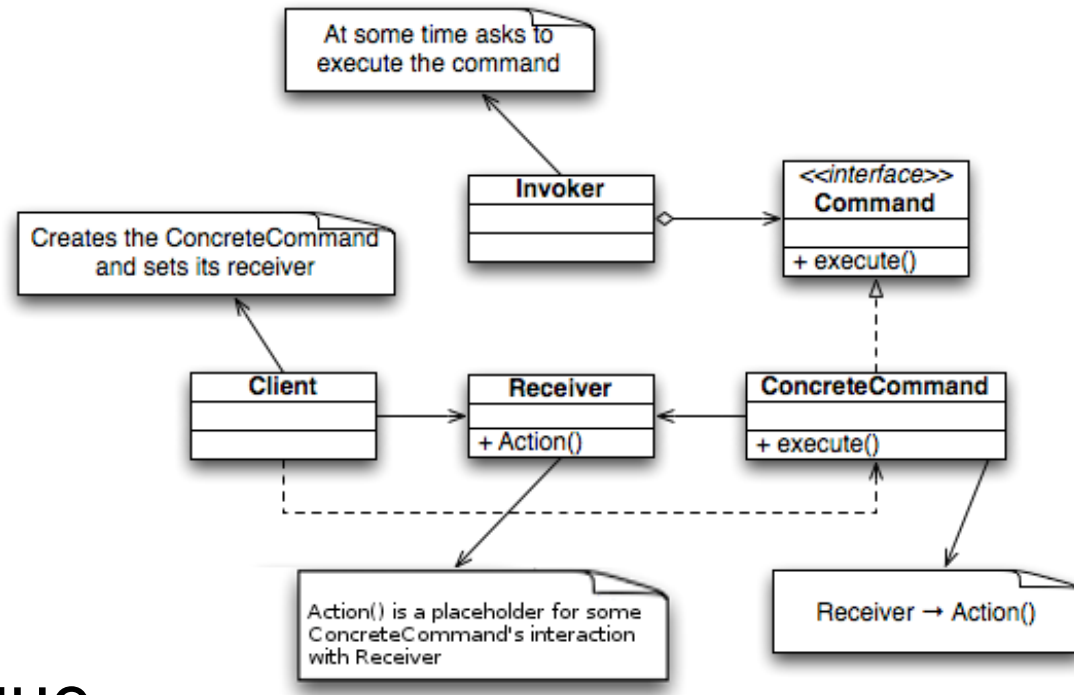
■ Objectif

- Encapsuler la logique métier d'un objet derrière une interface standardisée

■ Fonctionnement

- Un Receiver exécute les commandes
- Des ConcreteCommand appellent chaque méthode métier du Receiver
- Une Command décrit l'interface des ConcreteCommand
- Un Invoker stocke les instances de ConcreteCommand pour pouvoir les appeler de manière standardisée

Commande



■ Remarque

- Ce pattern introduit un couplage fort entre ses éléments

Source : http://en.wikipedia.org/wiki/Command_pattern

Chaîne de responsabilité

■ Objectif

- Effectuer plusieurs traitements non liés pour une même requête
(séparer les responsabilités)
 - Selon la même interface
 - En évitant le couplage entre les objets qui réalisent les traitements

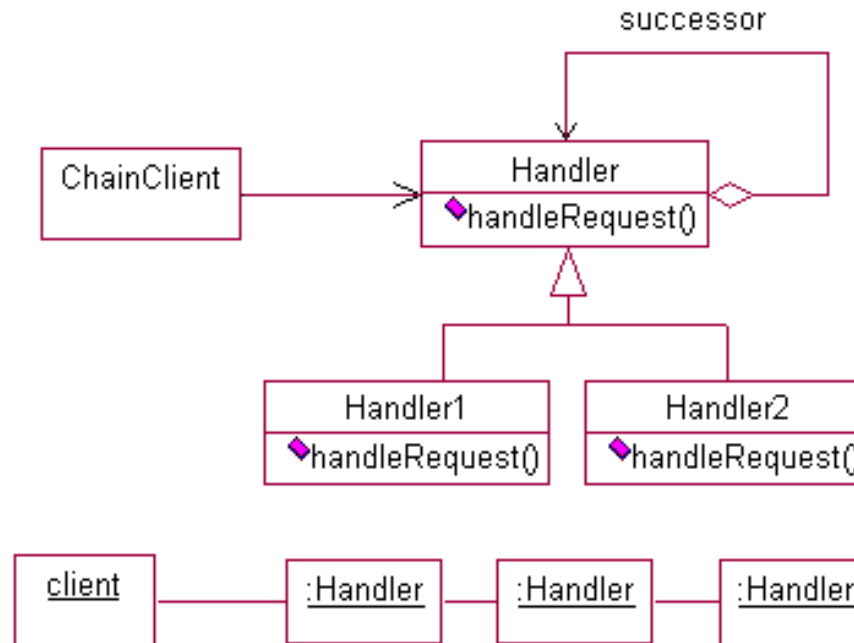
■ Fonctionnement

- Interface commune à tous les handlers
- Chaînage des handlers

■ Pattern lié

- Faible couplage, Décorateur

Chaîne de responsabilité



Source :

<http://www-sop.inria.fr/axis/cbrtools/usermanual-eng/Patterns/Chain.html>

■ Variante :

- Arbre de responsabilités (dispatcher)

Observateur (Observer)

■ Contexte

- Plusieurs objets *souscripteurs* sont concernés par les changements d'état d'un objet *diffuseur*

■ Objectifs

- Comment faire pour que chacun d'eux soit informé de ces changements ?
- Comment maintenir un faible couplage entre diffuseur et souscripteurs ?

■ Fonctionnement (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 (Observer)

■ Fonctionnement

- Un *Observateur* s'attache à un *Sujet*
- Le sujet *notifie* ses observateurs en cas de changement d'état

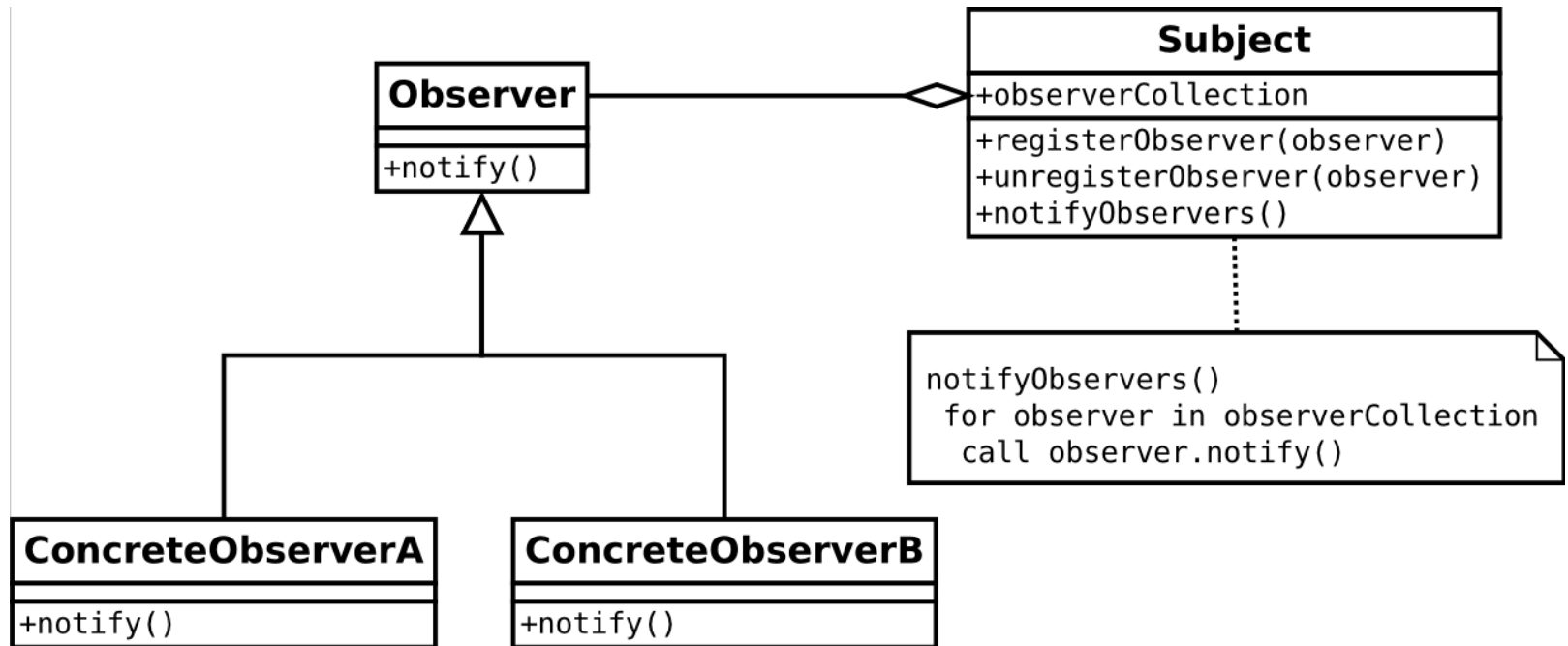
■ En pratique

- Subject : classe abstraite
- ConcreteSubject : classe héritant de Subject
- Observer : classe (utilisée comme classe abstraite)
- ConcreteObserver : classe héritant d'Observer

■ Autres noms

- Publish-subscribe, ou « Pub/Sub » (Diffusion-souscription)
- Modèle de délégation d'événements

Observateur (Observer)



Source : http://en.wikipedia.org/wiki/Observer_pattern

Observateur (Observer)

■ Utilisation en Java :

- Les classes *java.util.Observer* et *java.util.Observable* sont dépréciées depuis avril 2016
 - Modèle événementiel pas assez précis
 - Ordre des notifications non spécifié
 - Implémentation non thread-safe

➔ Mais le pattern en lui-même reste valable

- ➔ Utiliser le modèle événementiel de *java.beans* (*PropertyChangeEvent*, *PropertyChangeListener*)
- ➔ Utiliser les files et sémaphores (*java.util.concurrent*) avec des threads
- ➔ Utiliser votre propre implémentation si besoin

Fonction de rappel (Callback)

■ Objectif

- Définir un comportement sans savoir à quel moment il sera déclenché

■ Exemples d'utilisation

- Synchrones : déclenchement par une bibliothèque externe
- Asynchrone : modèle événementiel

■ Autre nom

- Principe d'Hollywood
« N'appellez pas, on vous rappellera. »

Fonction de rappel (Callback)

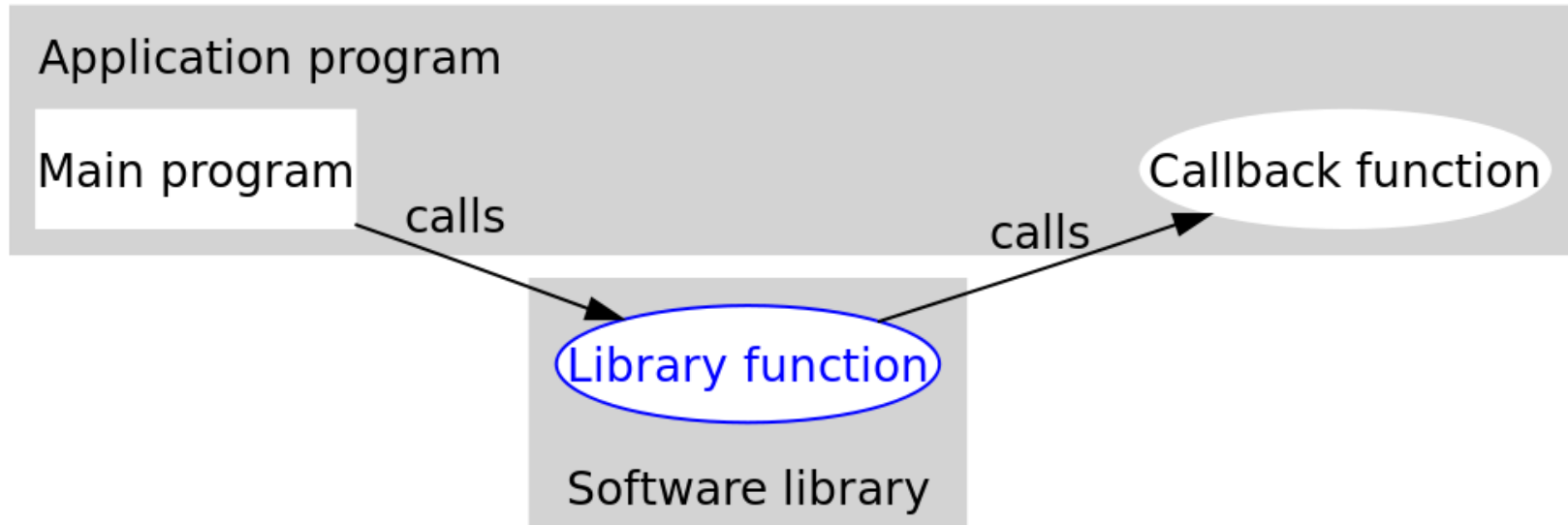
■ Fonctionnement

- Langages fonctionnels : passer une fonction en paramètre d'une autre fonction (fonctions d'ordre supérieur)
- Langages objet : passer un objet (qui encapsule un comportement) en paramètre d'une méthode d'un autre objet

■ Patterns liés

- Inversion de Contrôle (IoC), Observer

Fonction de rappel (Callback)



Source : [http://en.wikipedia.org/wiki/Callback_\(computer_science\)](http://en.wikipedia.org/wiki/Callback_(computer_science))

Promesse (Promise)

■ Problème

- Certains objets ont des comportements déclenchés par des événements extérieurs

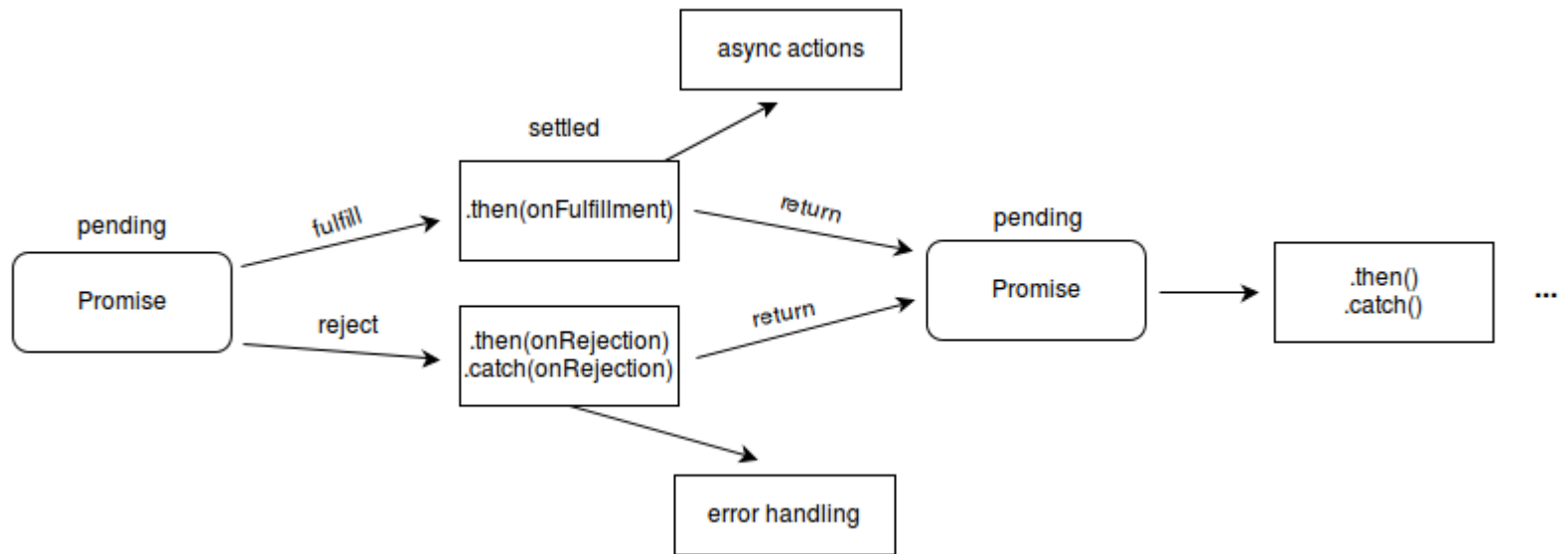
■ Objectif

- Spécifier le comportement d'un objet
 - Sans savoir comment il sera déclenché (asynchrone)
 - Sans en connaître le résultat

■ Fonctionnement

- Créer un objet *Promise* qui encapsule ce comportement et possède trois états
 - Pending : la promesse n'a pas été appelée
 - Fulfilled : elle a été appelée et s'est correctement déroulée
 - Rejected : elle a été appelée et a échoué

Promesse (Promise)



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Promesse (Promise)

■ Discussion

- Pattern beaucoup utilisé en JS (intégré) à ES6
- N'est qu'une simplification de l'encapsulation de callbacks

■ Variantes / extensions

- `Promise.all()`
- `Promise.race()`

■ Patterns liés

- Callback



Plan

- Introduction
- Principes GRASP
- Design patterns
- Patterns architecturaux
- Conclusion

Patterns architecturaux

■ Objectif

- Conception de systèmes d'information

■ Principes

- Organisation d'une société de classes / d'objets
 - Répartition / structuration des rôles
- Souvent présents dans les frameworks

■ Exemples de problèmes abordés

- Performance matérielle
- Disponibilité
- Réutilisation
- ...

Patterns architecturaux

■ Niveau de granularité

- Au-dessus des patterns précédents
 - Peuvent réutiliser d'autres design patterns

■ Références

- Conception de systèmes d'information pour l'entreprise (« Enterprise Architecture »)
 - Modélisation de l'entreprise par les processus :
http://en.wikipedia.org/wiki/Enterprise_modelling
 - Enterprise Architecture (en général) :
http://en.wikipedia.org/wiki/Enterprise_architecture

■ Autre nom

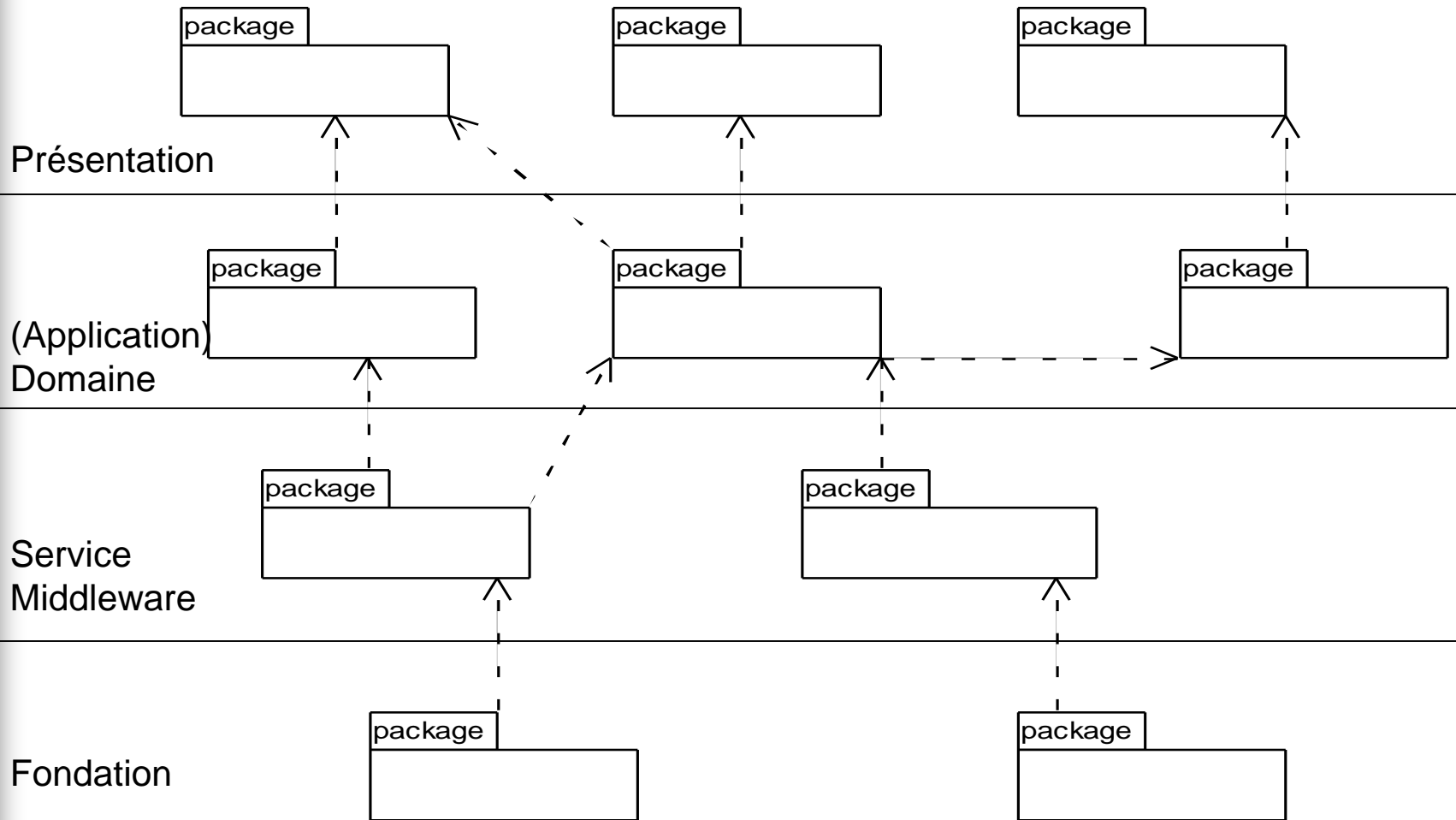
- Patterns applicatifs

Patterns architecturaux

■ Exemples

- Architecture en couches
- Architecture multi-tiers
- MV*
- IoC
- Contexte
- Observer ?
- DAO, DTO

Pattern Couches



Architecture multi-tiers

■ Objectif

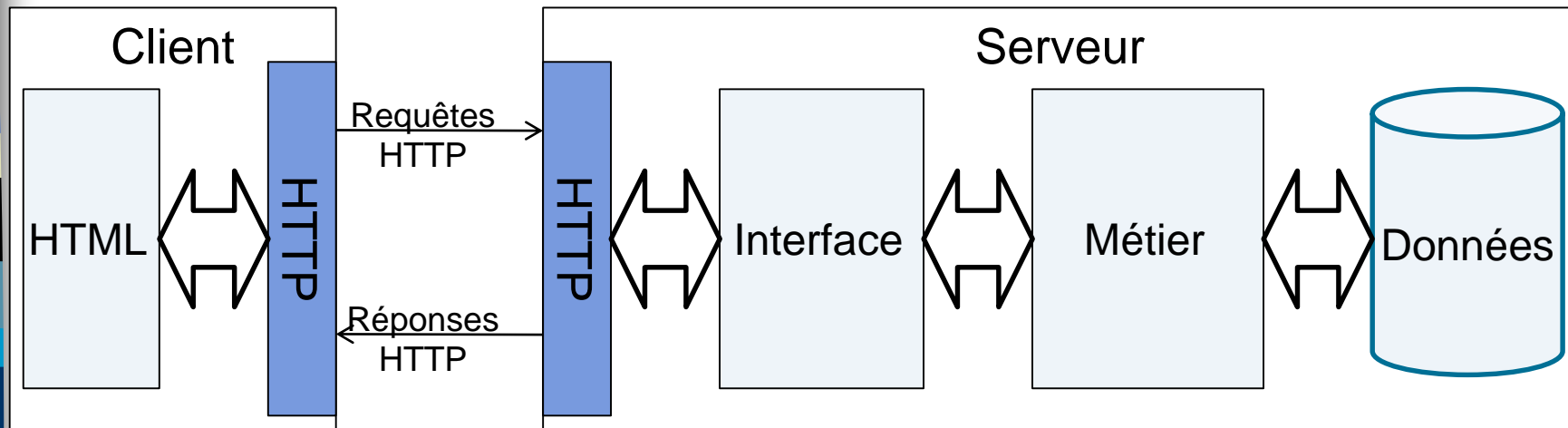
- Découpler les différentes fonctionnalités d'un programme (séparation des préoccupations)
 - Gestion des données, algorithmes métier, présentation...

■ Fonctionnement

- Concevoir séparément chacune de ces fonctionnalités
- Les isoler les unes des autres (autant que possible)

Architecture multi-tiers

■ Exemple (M1IF03)



■ Pattern lié – Couches

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

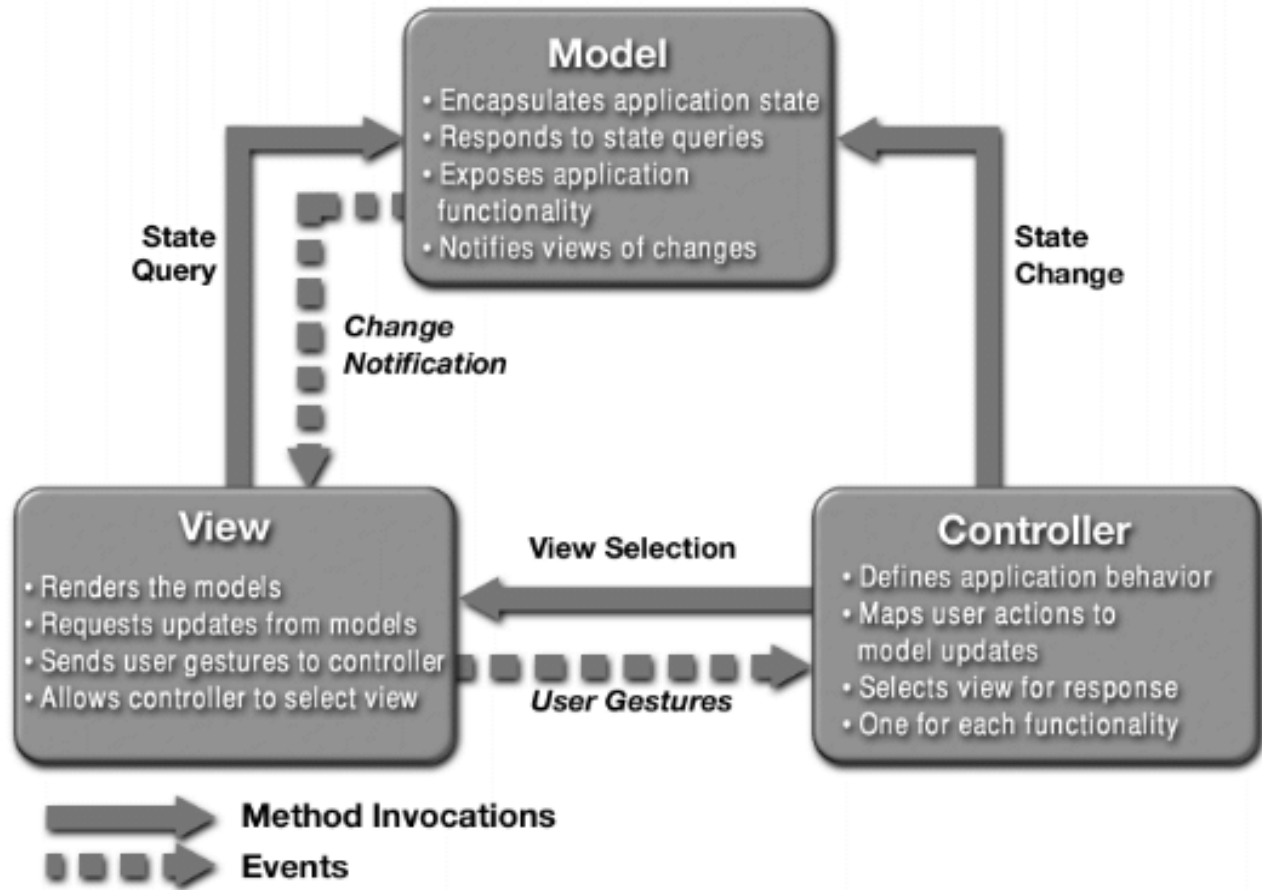
■ Solution

- 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)



Source originale : BluePrint Java (Sun), non maintenue par Oracle.

Modèle-Vue-Contrôleur (suite)

■ Différentes versions

- la vue connaît le modèle ou non
- le contrôleur connaît la vue ou non
- la vue connaît le contrôleur ou non
- « Mélange » avec le pattern Observer
- Un ou plusieurs contrôleurs (« type 1 » / « type 2 »)
- Push-based vs. pull-based

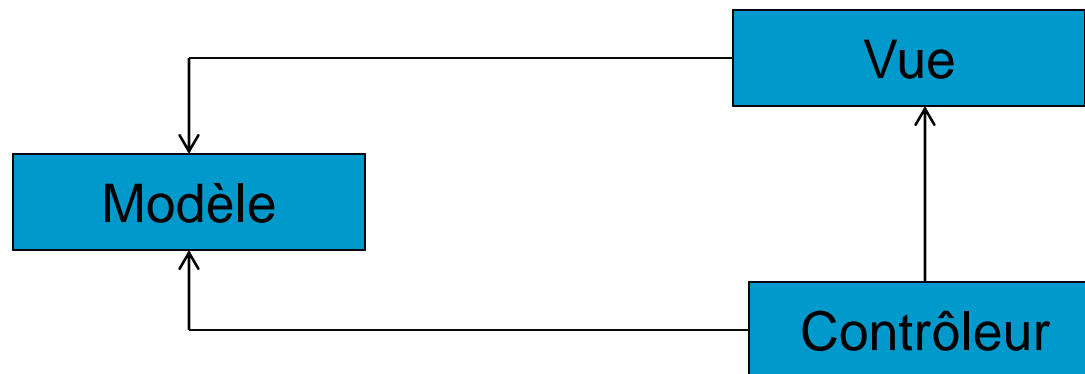
■ Choix d'une solution

- dépend des caractéristiques de l'application
- dépend des autres responsabilités du contrôleur

Modèle-Vue-Contrôleur (suite)

■ Version modèle passif

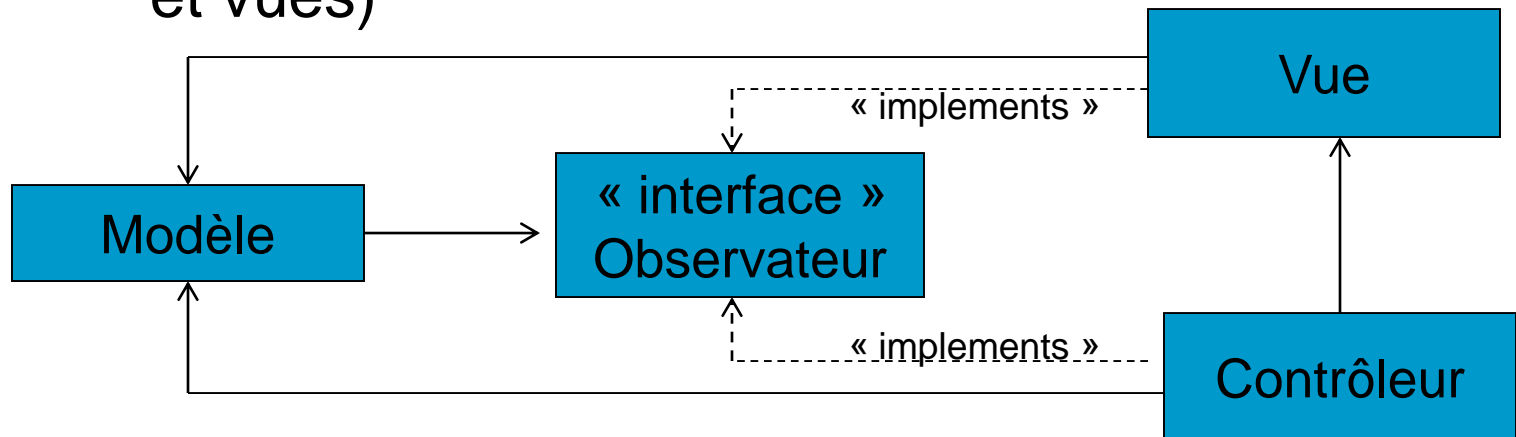
- la vue se construit à partir du modèle
- le contrôleur notifie le modèle des changements que l'utilisateur spécifie dans la vue
- le contrôleur informe la vue que le modèle a changé et qu'elle doit se reconstruire



Modèle-Vue-Contrôleur (suite)

■ Version modèle actif

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



Autres patterns MV*

■ Model-View-Adapter (MVA)

- Pas de communication directe entre modèle et vue
 - Un pattern Adapteur (Médiateur) prend en charge les communications
 - Le modèle est intentionnellement opaque à la vue
- Il peut y avoir plusieurs adapteurs entre le modèle et la vue

■ Model-View-Presenter (MVP)

- La vue est une interface statique (templates)
- La vue renvoie (route) les commandes au Presenter
- Le Presenter encapsule la logique de présentation et l'appel au modèle

■ Model-View-View Model (MVVM)

- Mélange des deux précédents : le composant View Model
 - Sert de médiateur pour convertir les données du modèle
 - Encapsule la logique de présentation
- Autre nom : Model-View-Binder (MVB)

Inversion de Contrôle (IoC)

■ Objectif

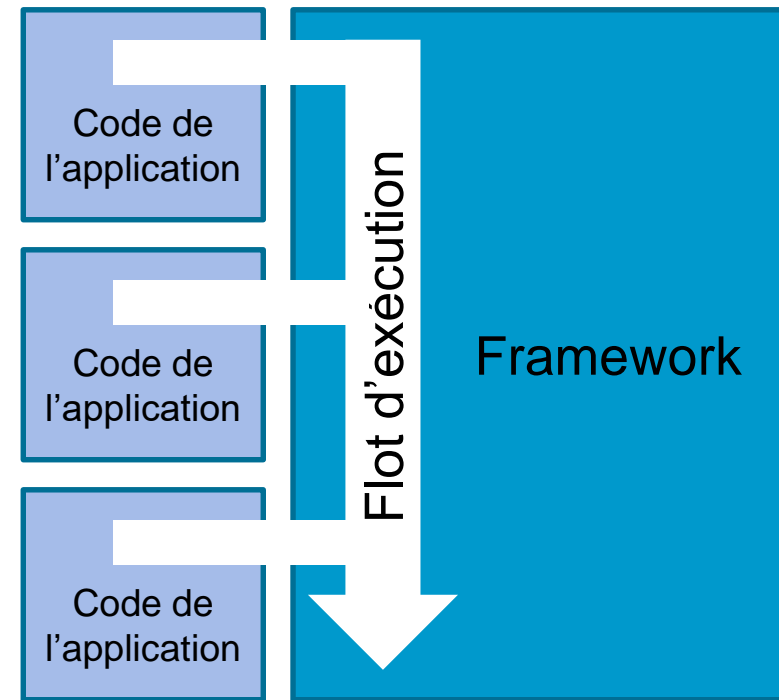
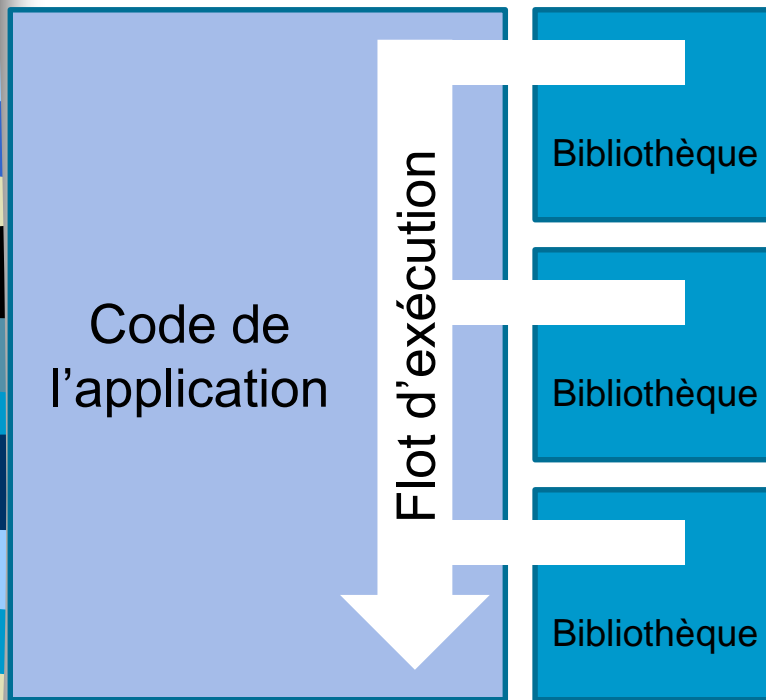
- Ne pas réimplémenter le code « générique » d'une application
- Permettre l'adjonction simple
 - De composants spécifiques métier
 - De services annexes disponibles par ailleurs

■ Fonctionnement

- Utiliser un *Conteneur* capable de
 - Gérer le flot de contrôle de l'application
 - Instancier des *composants*
 - Résoudre les dépendances entre ces composants
 - Fournir des services annexes (sécurité, accès aux données...)

Inversion de Contrôle (IoC)

■ Exemple



■ Autre nom

- Injection de dépendances

Patrons architecturaux

■ Patrons applicatifs (suite)

– Patrons d'authentification

- Directe, à l'aide d'une plateforme
- Single Sign On (CAS)

– Patrons d'autorisation

- Rôles, attributs, activité, utilisateur, heure...

– Patrons de sécurité

- Checkpoint, standby, détection/correction d'erreurs

Patrons architecturaux

■ Patrons de données

- Architecture des données
 - Transactions, opérations, magasins, entrepôts
- Modélisation de données
 - Relationnelle, dimensionnelle
- Gouvernance des données (Master Data Management)
 - Réplication, services d'accès, synchronisation

Patrons architecturaux

- Types d'architectures et d'outils
 - Plateformes de composants (frameworks)
 - Architectures orientées services (SOA)
 - Extract Transform Load
 - Enterprise Application Infrastructure / Enterprise Service Bus



Plan

- Introduction
- Principes GRASP
- Design patterns
- Patterns architecturaux
- Conclusion

Pour aller plus loin...

■ Patterns of Enterprise Application Architecture

– Origine

- Livre de Martin Fowler, Dave Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford, 2002

– Contenu

- Formalisation de l'expérience de développement d'« Enterprise Applications »
- Généralisation d'idiomes de plusieurs langages
- Une quarantaines de patterns souvent assez techniques

– Exemples

- Service Layer, Foreign Key Mapping, MVC, Front Controller, DTO, Registry, Service Stub...

– Référence

- <http://martinfowler.com/eaCatalog/>

Anti-patterns

- 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
 - ...
- Référence
 - <http://en.wikipedia.org/wiki/Anti-pattern>

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 refactorer en patterns)
- Exemples d'outils
 - Eclipse + plugin UML
 - Describe + Jbuilder
 - IntelliJ
 - ...

Conclusion

- On a vu assez précisément les principes les plus généraux (GRASP)
- On a survolé quelques design patterns
 - un bon programmeur doit les étudier et en connaître une cinquantaine
- On a évoqué les patterns architecturaux
 - Ils permettent de comprendre le fonctionnement des outils (frameworks)
- On a à peine abordé les anti-patterns
 - Les connaître est le meilleur moyen de détecter que votre projet est en train de ...



Remerciements

- Yannick Prié
- Laëtitia Matignon
- Olivier Aubert

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

■ En Français

- Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates, *Design Patterns – Tête la première*, O'Reilly Eds., 640 p., 2005.

Références

■ Sur le Web

– Généralités sur les Design patterns

- http://fr.wikipedia.org/wiki/Patron_de_conception
- http://en.wikipedia.org/wiki/Category:Software_design_patterns
- http://en.wikipedia.org/wiki/Architectural_pattern
- <http://stackoverflow.com/questions/4243187/difference-between-design-pattern-and-architecture>
- <http://martinfowler.com/eaCatalog/>
- <http://www.hillside.net/patterns>
- <http://java.sun.com/blueprints/corej2eepatterns/>

– Historique, classification

- <https://www.javaworld.com/article/2078665/core-java/design-patterns--the-big-picture--part-1--design-pattern-history-and-classification.html>
- <http://people.cs.umu.se/jubo/ExJobbs/MK/patterns.htm>
- <http://wiki.c2.com/?HistoryOfPatterns>

Références

■ Sur le Web

- Promesse
 - <https://www.promisejs.org/>
- Design by Contract
 - https://en.wikipedia.org/wiki/Design_by_contract
 - <https://hillside.net/plop/plop97/Proceedings/dechamplain.pdf>
- Enterprise Integration Patterns
 - https://en.wikipedia.org/wiki/Enterprise_Integration_Patterns
 - <https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html>
 - <https://www.enterpriseintegrationpatterns.com/>
- Antipatterns
 - <https://sourcemaking.com/antipatterns/software-development-antipatterns>
 - <http://c2.com/cgi/wiki?AntiPatternsCatalog>