Design Patterns

Aurélien Tabard

Master 1 – Université Claude Bernard Lyon 1

Inspiré des cours de Y. Prié, O. Aubert, L. Medini et F. Echtler.

Plan

- Introduction
- Patrons architecturaux
- Patrons GRASP
- Design patterns
- Antipatterns
- UI patterns

Plan

- **Introduction**
- Patrons architecturaux
- Patrons GRASP
- Design patterns
- UI patterns
- Antipatterns

Les design patterns (patrons de conception)

Recettes pour des problèmes communs

Disponibles à tout les niveaux (architectural, interface, implémentation)

Souvent décris sans code, au plus de l'UML.

Des diagrammes simples boîtes-lignes spécialement pour les diagrammes architecturaux.

Format d’un design pattern

Problème

- Raison / domaine d’application

Solution

- Structure (diagramme de classe)
- Éléments (nom des classes et opérations)
- Interactions entre objets (diagramme de séquence)

Discussion

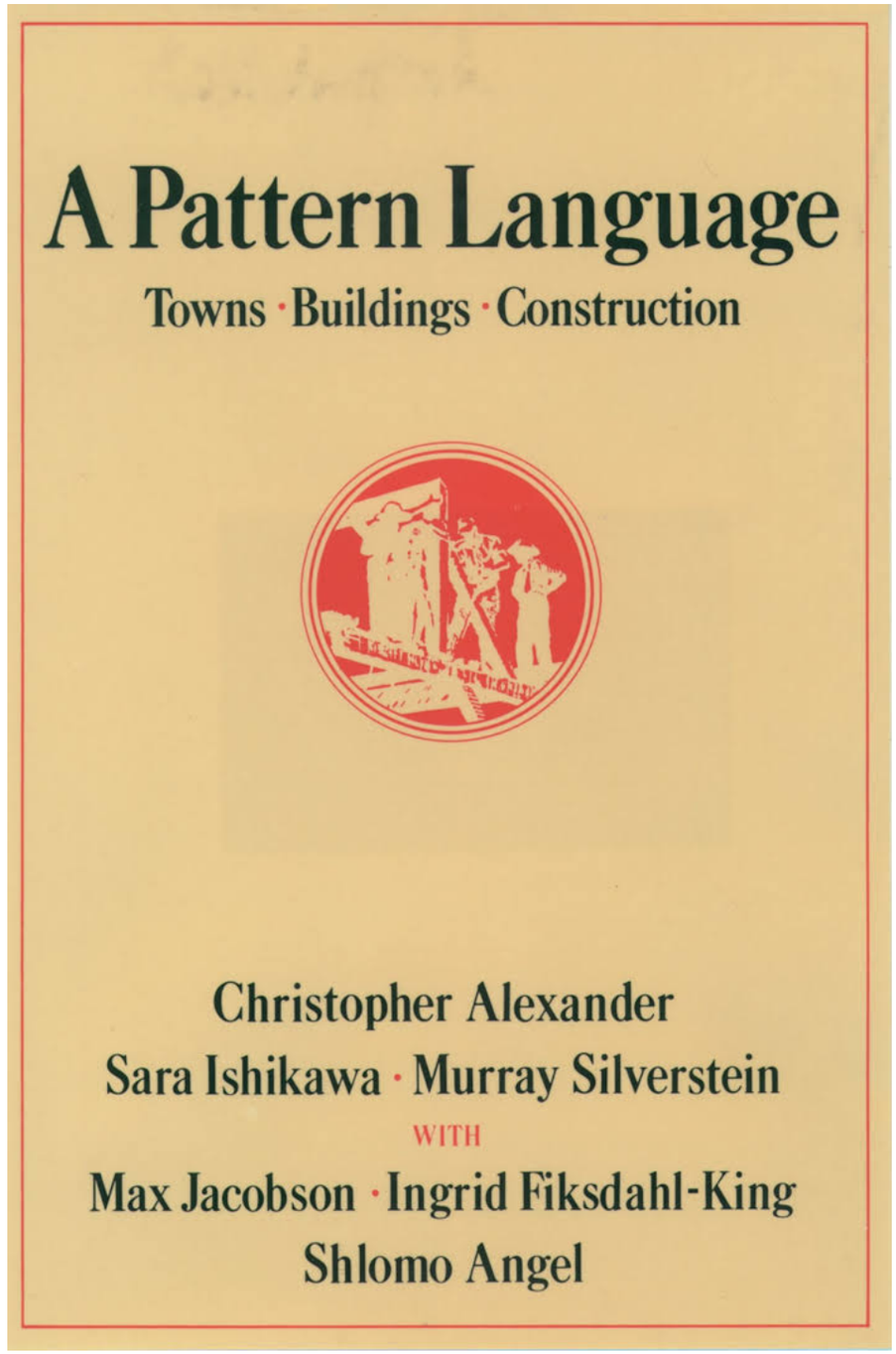
- Avantages, inconvénients, patrons associés
- Contraintes, cas spéciaux, utilisations connues

L’origine des design patterns – 1977

Sources en Architecture

Livre de Alexander, Ishikawa et Silverstein: A Pattern Language – Towns, Buildings, Construction.

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”



Exemples

141 A Room of One's Own

May be part of Intimacy Gradient (127), The Family (75), House for a Small Family (76), Common Areas at the Heart (129).

Conflict

No one can be close to others, without also having frequent opportunities to be alone.

Resolution

Give each member of the family a room of his own, especially adults. A minimum room of one's own is an alcove with desk, shelves, and curtain. The maximum is a cottage- like a Teenagers Cottage(154), or an old age cottage(155). In all cases, especially the adult ones, place these rooms at far ends of the intimacy gradient- far from the common rooms.

22 Nine Percent Parking

May be part of Local Transport Areas (11), Community Neighbourhood (14).

Conflict

Very simply- when the area devoted to parking is too great, it destroys the land.

Resolution

Do not allow more that 9% of the land in any given area to be used for parking. In order to prevent the bunching of parking in huge neglected areas, it is necessary for a town or a community to subdivide its land into parking zones no larger than 10 acres each and to apply the same rule in each zone.

May contain Shielded Parking (97), Small Parking Lots (103).

Image source (FU): “A Pattern Language”, C. Alexander

Propriétés génériques des patterns

Pragmatisme

- solutions existantes et éprouvées

Récurrence

- bonnes manières de faire éprouvées

Généralité

- comment et quand appliquer, indépendance au langage de programmation

Émergence

- la solution globale émerge de l’application d’un ensemble de patrons

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

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)

Orienté architecture

- Martin Fowler (2002) Patterns of Enterprise Application Architecture, Addison Wesley

Sites

- http://www.hillside.net/patterns
- http://java.sun.com/blueprints/corej2eepatterns/

Plan

- Introduction
- **Patrons architecturaux**
- Patrons GRASP
- Design patterns
- UI patterns
- Antipatterns

Patterns architecturaux

Patterns de haut niveau: vue globale du système

Plusieurs vues sont possibles

Exemples:

- Architecture Client-Serveur
- Architecture en couche
- Model-View-Controller
- Repository pattern
- Pipe-and-Filter architecture

Patterns architecturaux

Patterns de haut niveau: vue globale du système

Plusieurs vues sont possibles

Exemples:

- **Architecture Client-Serveur**
- Architecture en couche
- Model-View-Controller
- Repository pattern
- Pipe-and-Filter architecture

Architecture client-serveur

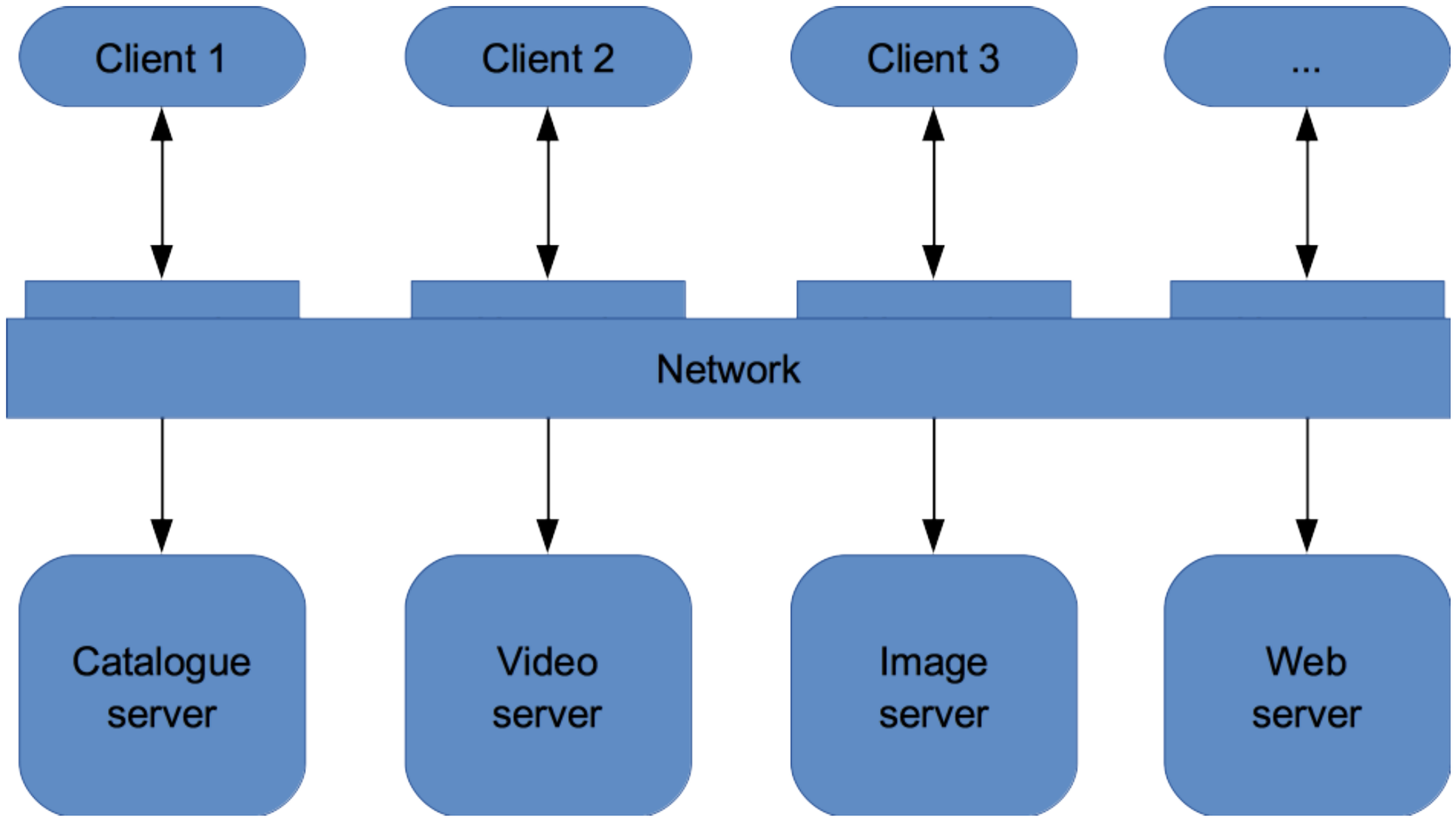


Image source (FU): Sommerville, Software Engineering, Chapter 6

Architecture client-serveur

Fonctionnalités divisée en services

Chaque service est fournit par un serveur (logique)

- Accès à des données partagées en plusieurs lieux
- Équilibrage de charge → serveurs de réplications

Avantages:

- Distribué, architecture transparente d'un point de vue réseau

Désavantages:

- Chaque serveur → single point of failure
- Performance du réseau imprévisible

Patterns architecturaux

Patterns de haut niveau: vue globale du système

Plusieurs vues sont possibles

Exemples:

- Architecture Client-Serveur
- **Architecture en couche**
- Model-View-Controller
- Repository pattern
- Pipe-and-Filter architecture

Architecture en couche, un exemple : Internet

HTTP (HyperText Transfer Protocol)	Application/ Presentation
TLS (Transport Layer Security)	Session
TCP (Transmission Control Protocol)	Transport
IP (Internet Protocol)	Network
802.11* MAC/LLC (WLAN Management)	Data Link
802.11* PHY (Wireless Hardware)	Physical

Architecture en couche

System couche une pile de couches interconnectées

- Les couches ne communiquent qu'avec leurs voisins
- Complexité grandissante

Utilisée souvent pour les protocoles réseaux

Avantages:

- Les couches peuvent être remplacées
- Des spécifications claires sur lesquelles construire

Inconvénients:

- Une séparation propre des coches peut être difficile

Patterns architecturaux

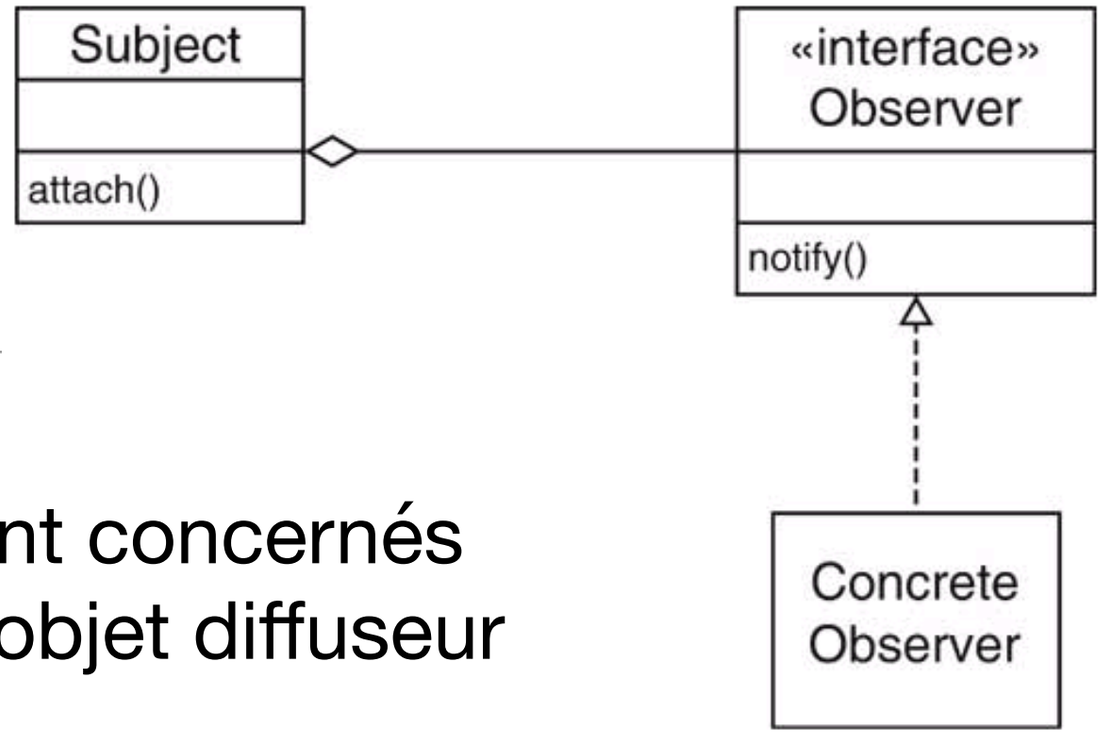
Patterns de haut niveau: vue globale du système

Plusieurs vues sont possibles

Exemples:

- Architecture Client-Serveur
- Architecture en couche
- **Model-View-Controller**
- Repository pattern
- Pipe-and-Filter architecture

Observateur (Observer)



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 (pub-sub),
  Modèle de délégation d'événements

Modèle-Vue-Contrôleur (MVC)

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

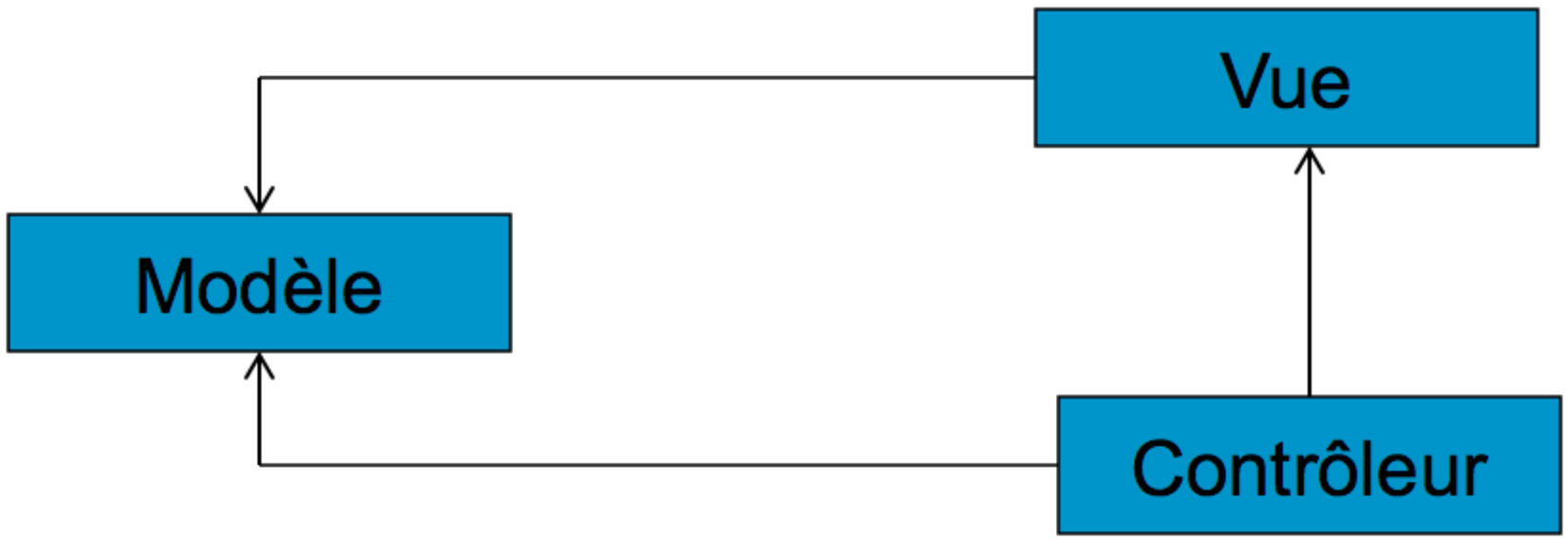
Contrôleur

- Gère les changements d'état du modèle
- Informe le modèle des actions utilisateur
- Sélectionne la vue appropriée

Modèle-Vue-Contrôleur (suite)

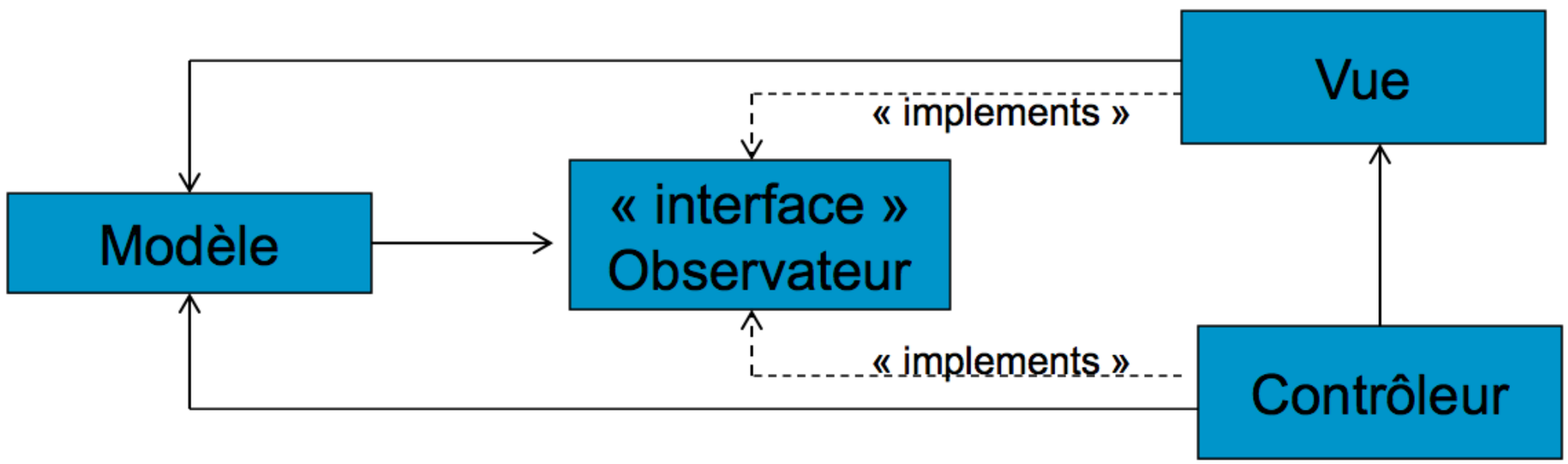
Version modèle passif

- 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



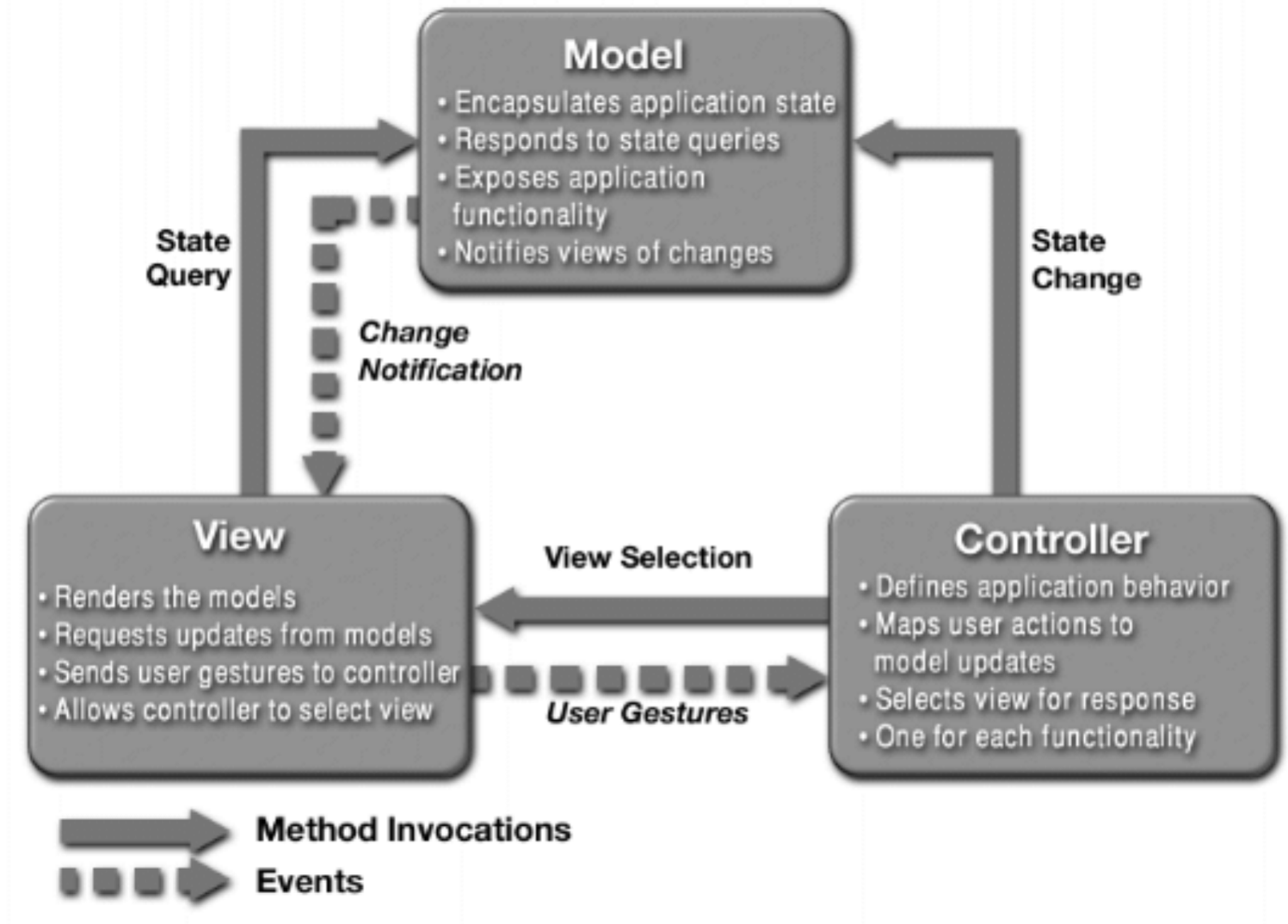
Version modèle actif

- 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)



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

Patterns architecturaux

Patterns de haut niveau: vue globale du système

Plusieurs vues sont possibles

Exemples:

- Model-View-Controller
- Architecture Client-Serveur
- Architecture en couche
- Repository pattern
- Pipe-and-Filter architecture

Les points à considérer

Performance

- minimiser la communication entre composants

Sécurité

Sureté

- Identifier les fonctionnalités critiques de chaque sous-système

Disponibilité

- Inclure de la redondance

Maintenance

- Utiliser de petits composants, remplaçables

Plan

- Introduction
- Patrons architecturaux
- **Patrons GRASP**
- Design patterns
- UI patterns
- Antipatterns

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

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 identifiant unique 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

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

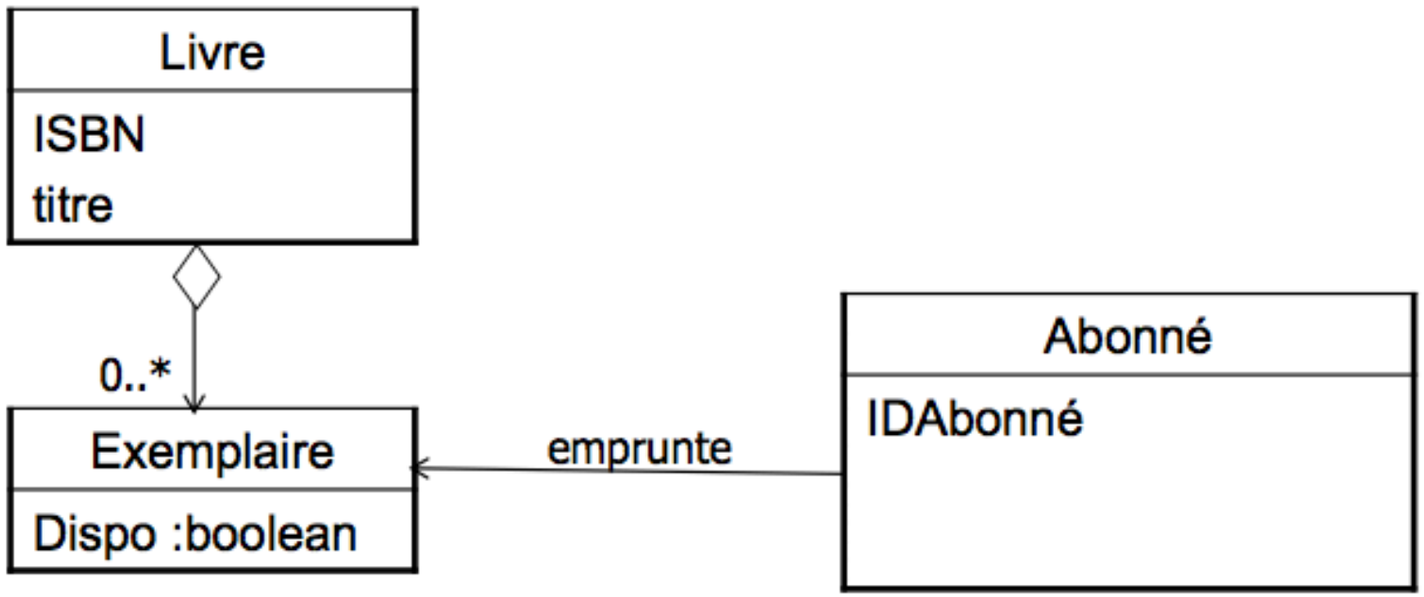
- Quel est le principe général d'affectation des responsabilités aux objets ?

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

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

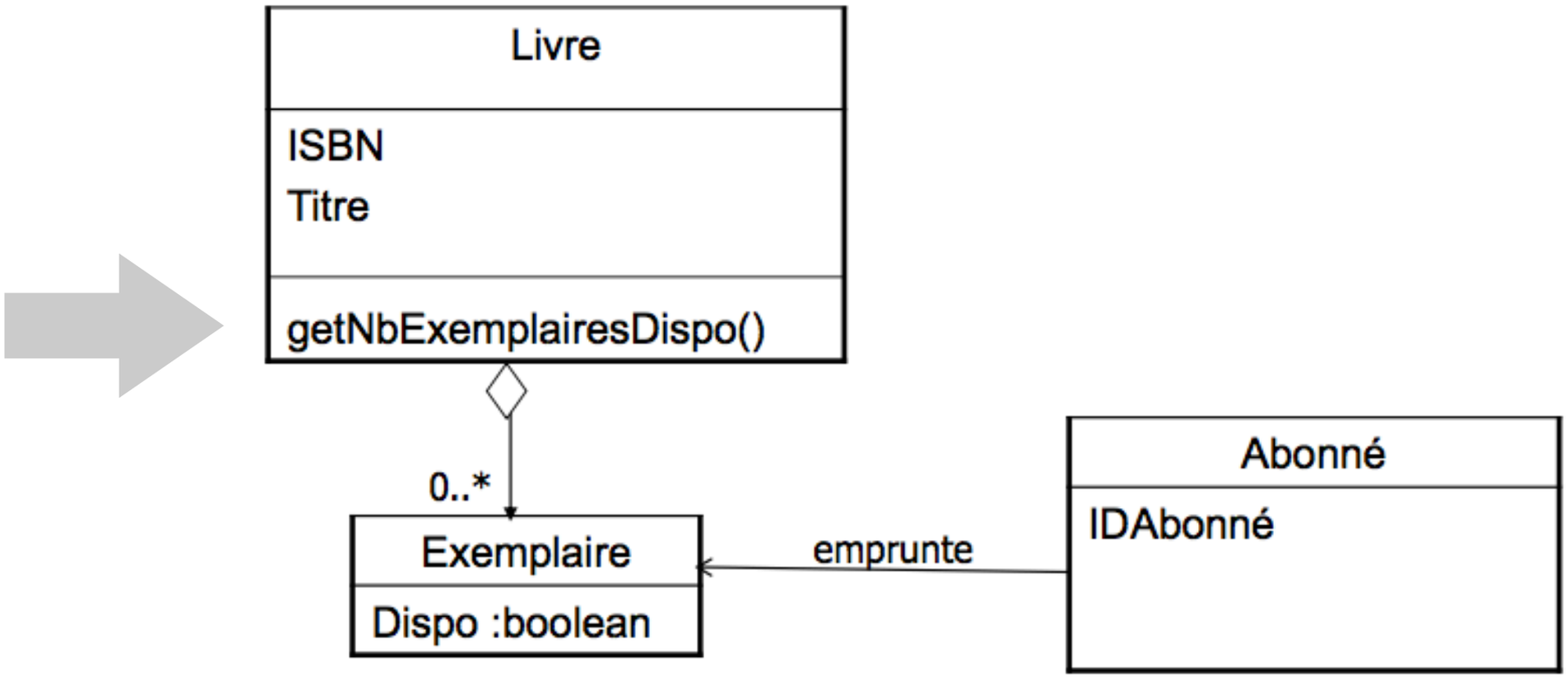
Commencer avec la question :

- 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)

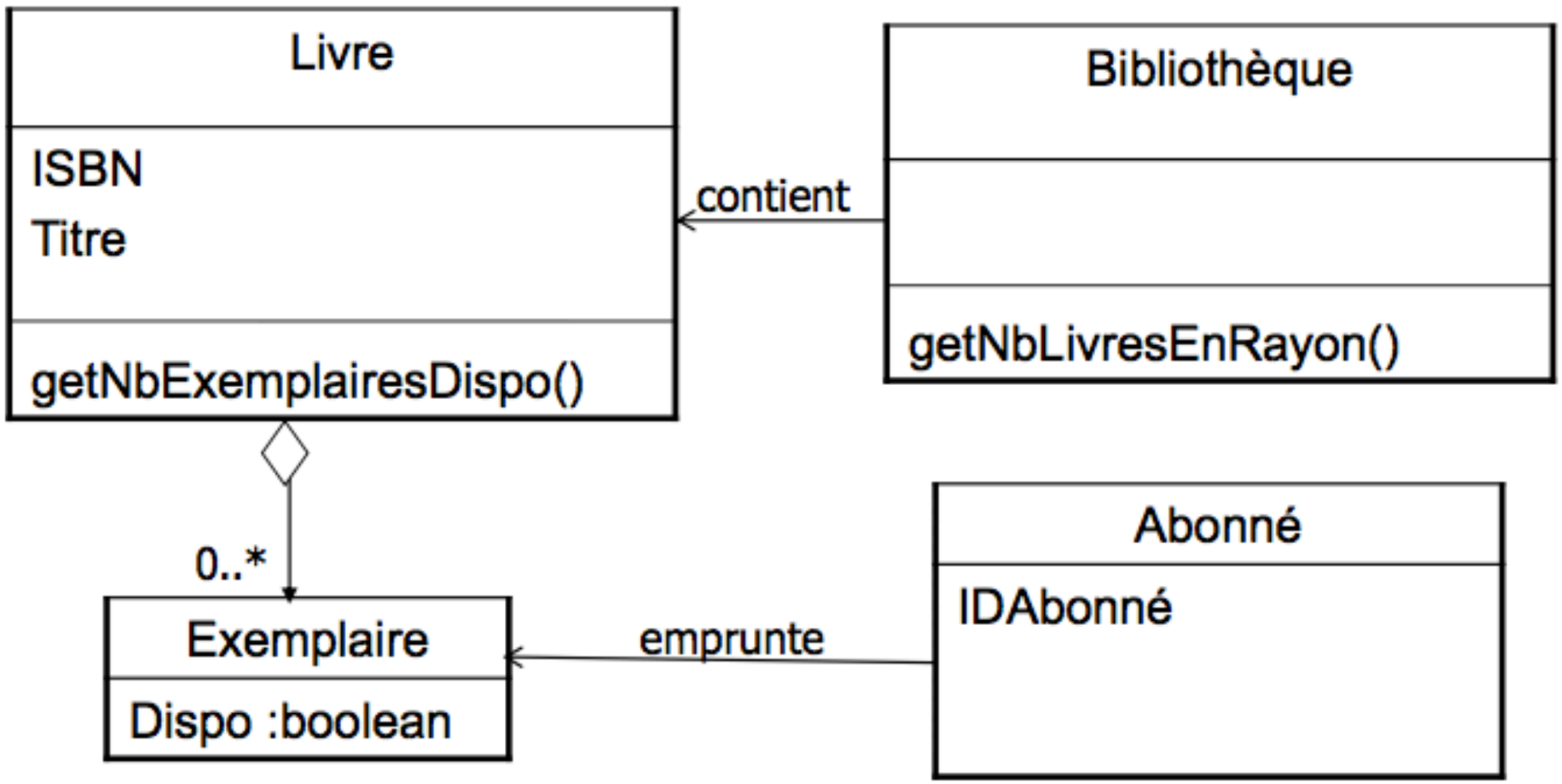
Tâche complexe

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

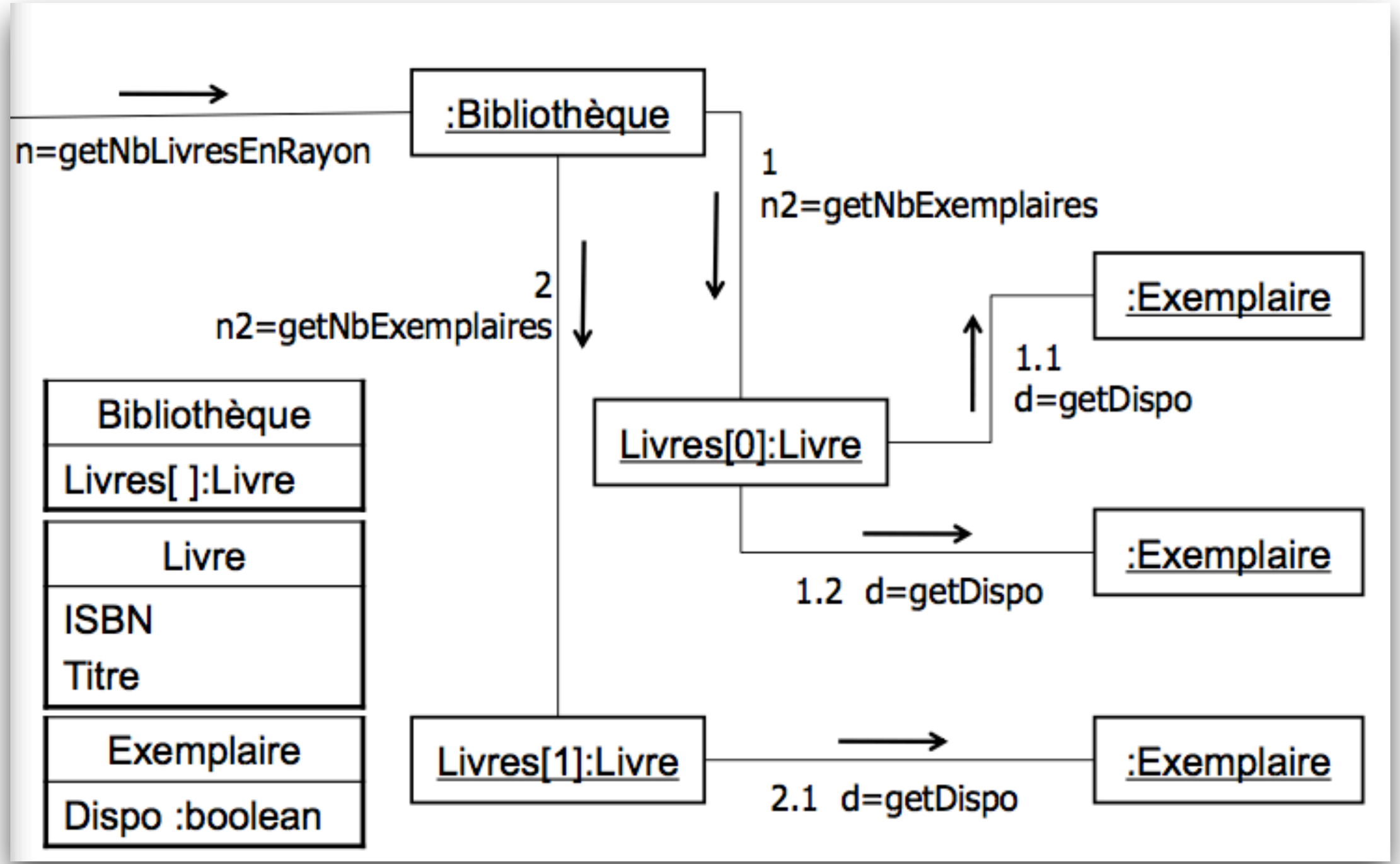
Solution : décomposer la tâche

- 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 : 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

➡ Le plus utilisés de tous les patterns d'attribution de responsabilités

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

Créateur

Problème

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

Solution – Affecter à la classe B la responsabilité de créer une instance de la classe A si une (ou +) de ces conditions est vraie :

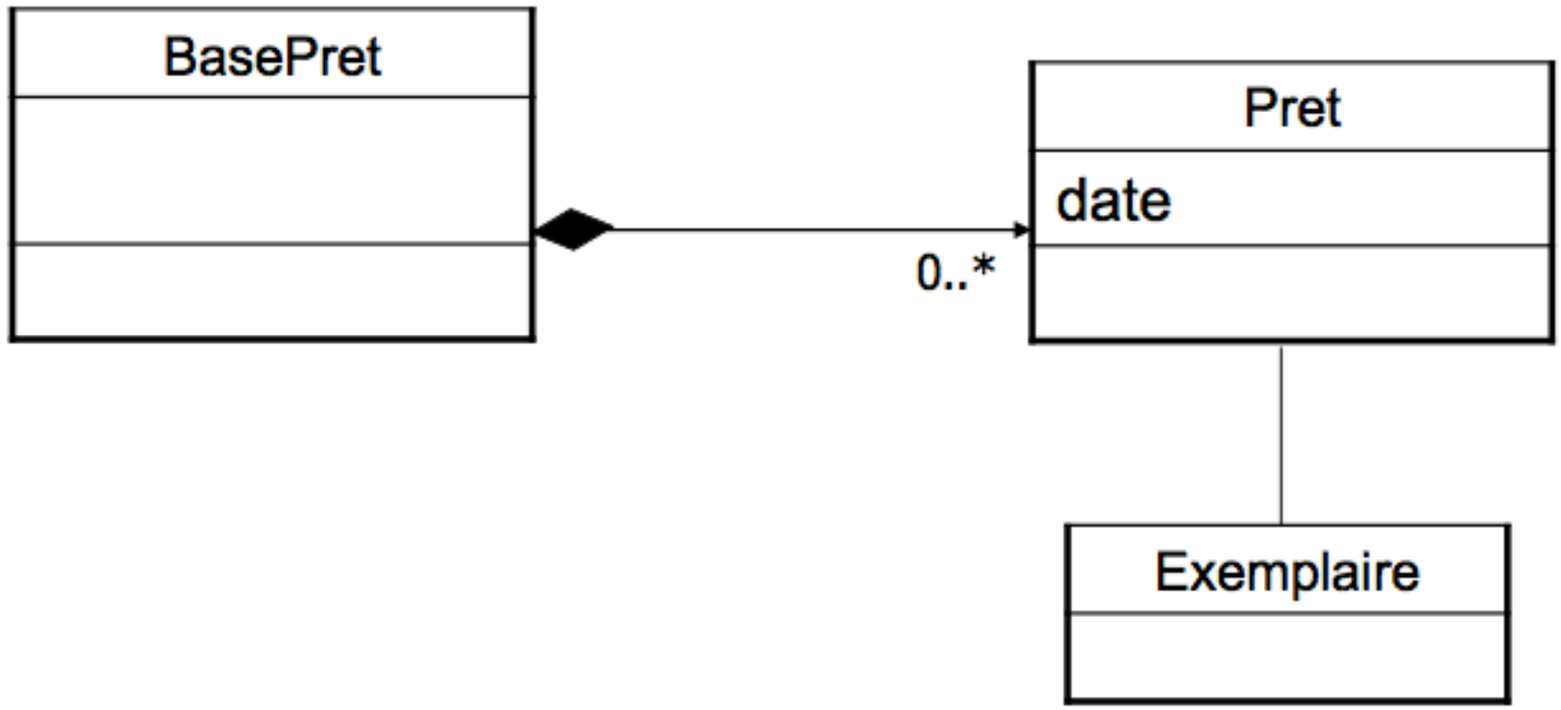
- B contient ou agrège des objets A
- B enregistre des objets A
- B utilise étroitement des objets A
- B a les données d'initialisation qui seront transmises aux objets A lors de leur création

➡ 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 de réutilisation

Pattern liés

- Faible couplage
- Composite
- Fabricant

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

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)

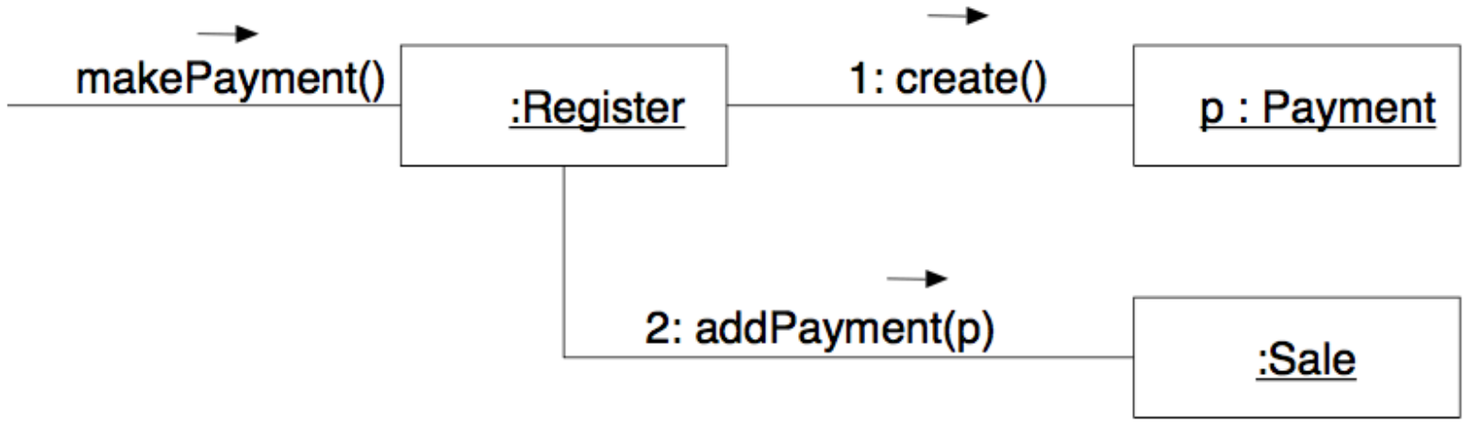
Principe général

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

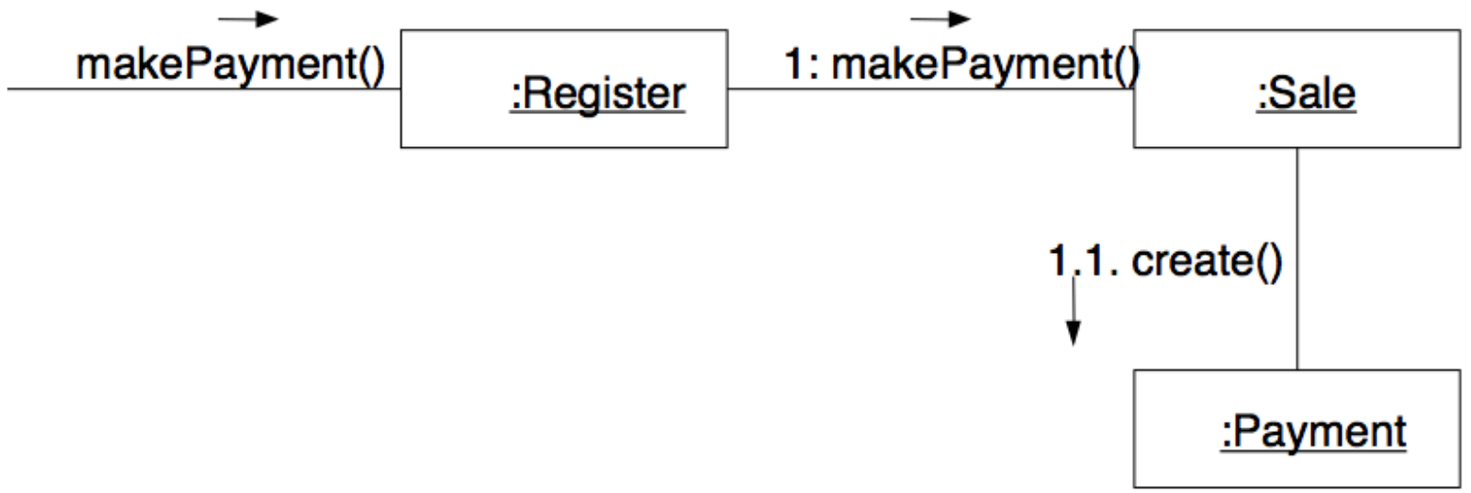
Mise en œuvre

- 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

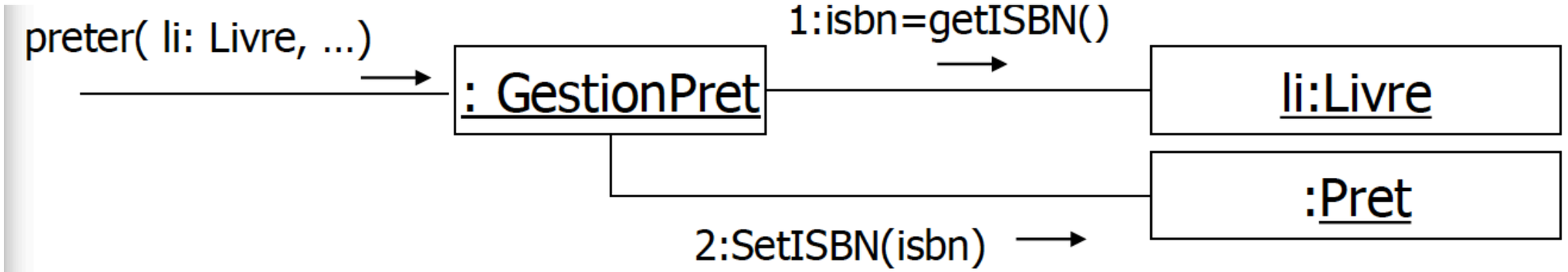


Que choisir ?

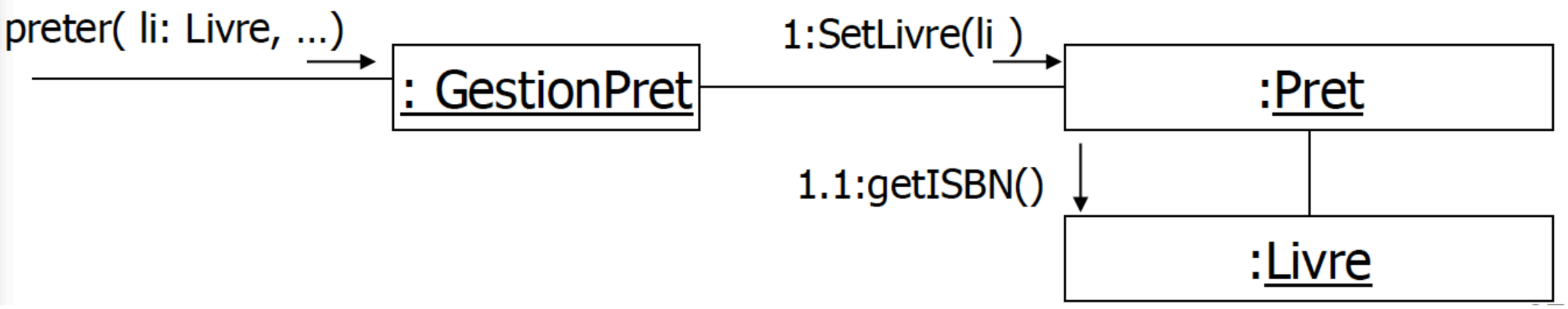


Faible couplage : exemple

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

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

Forte cohésion (GRASP)

Problème : maintenir une complexité gérable

- Comment s'assurer que les objets restent :
  - compréhensibles ?
  - faciles à gérer ?
  - Comment s'assurer 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

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

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

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*)

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

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

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

Contrôleur : discussion

Avantages

- Niveau d'indirection matérialisant la séparation Modèle-vue
- Brique de base pour une conception modulaire
- 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…)

Patterns liés

- Indirection, Couches, Façade, Fabrication pure, Commande

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

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

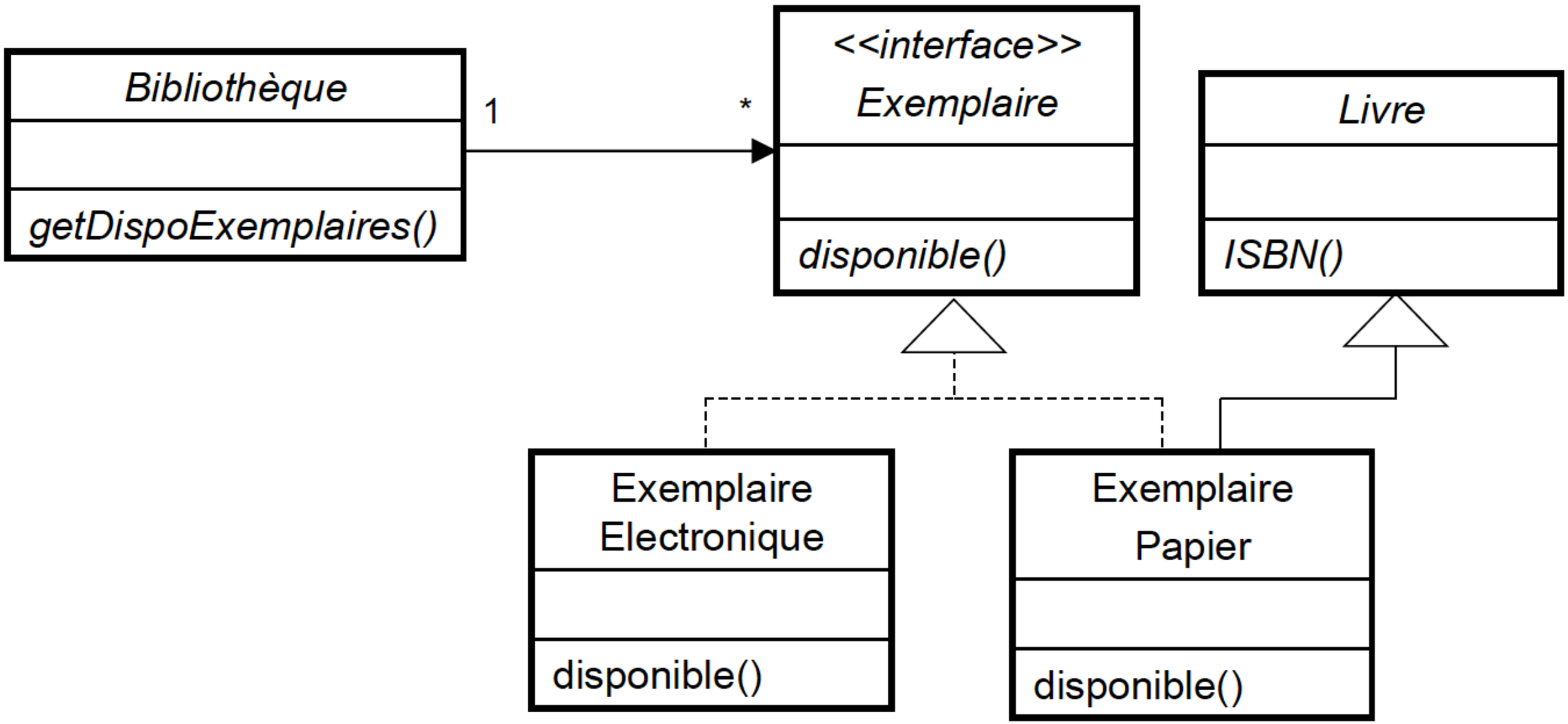
- Tirer avantage de l'approche OO en sous-classant les opérations dans des types dérivés de l'Expert en information
  L'opération nécessite à la fois des informations et un comportement particuliers

Mise en oeuvre

- 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

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

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

Fabrication pure (GRASP)

Problème

- Que faire pour respecter le Faible couplage et la Forte cohésion
- Que faire 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 : 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 oeuvre

- 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

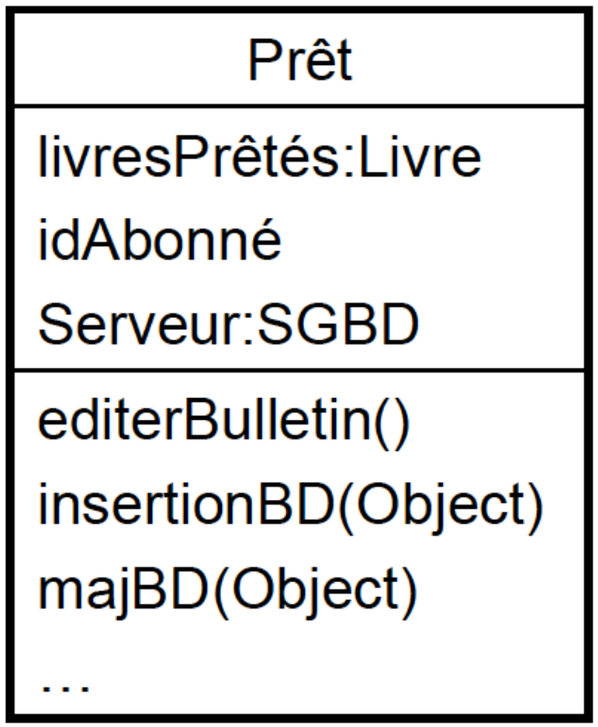
Fabrication pure : exemple

Problème

- les instances de Prêt doivent être enregistrées dans une BD

Solution initiale (d'après Expert)

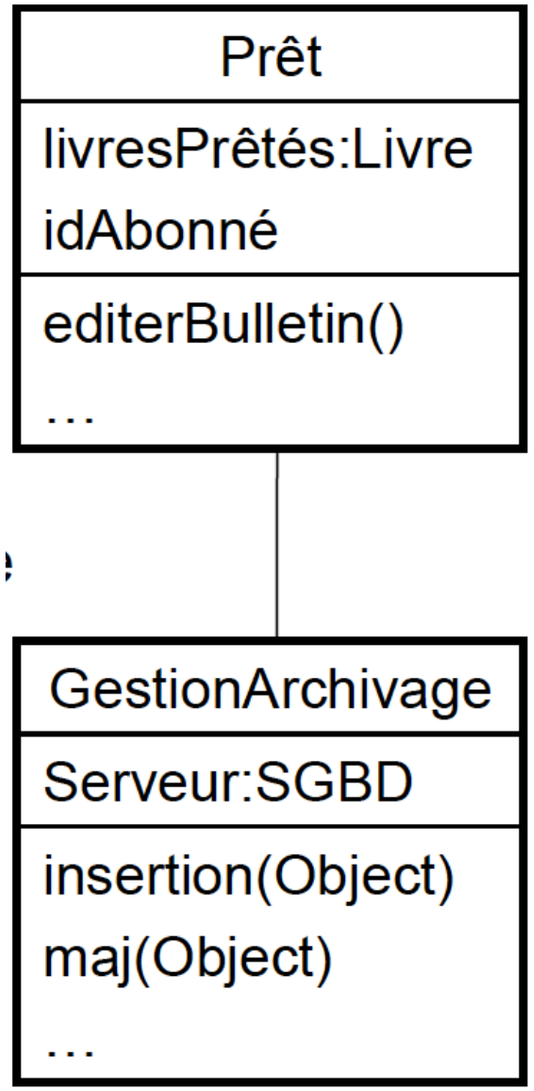
- Prêt a cette responsabilité
- cela nécessite
  - un grand nombre d'opérations de BD
    -> Prêt devient donc non cohésif
  - de lier Prêt à une BD
    -> le couplage augmente pour Prêt



Fabrication pure

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

Règle d'or

- Concevoir des objets Fabrication pure en pensant à leur ré-utilisabilité
- 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

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

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

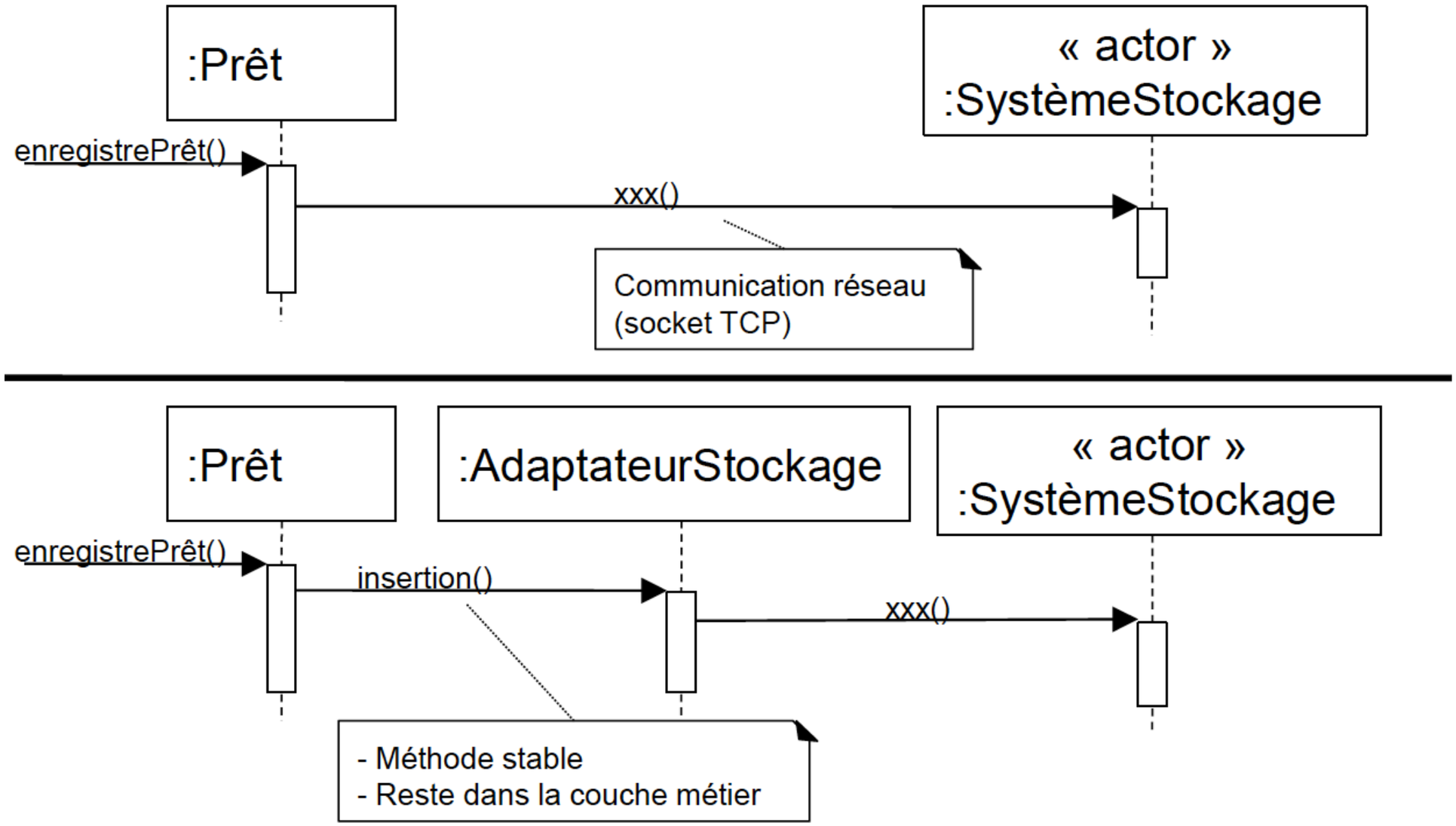
- Réaliser des adaptateurs, façades, etc. (pattern Protection des variations) qui s'interfacent avec des systèmes extérieurs
- Exemples : proxys, DAO, ORB
- Réaliser des inversions de dépendances entre packages

Mise en oeuvre

- 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

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…

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**

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 oeuvre

- 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

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

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
- Patrons architecturaux
- Patrons GRASP
- **Design patterns**
- UI patterns
- Antipatterns

Design patterns(niveau code)

Design Patterns – Elements of Reusable Object-Oriented Software

La référence par Gamma, Helm, Johnson and Vlissides aussi appelé **GangOfFour** ou GoF-Book

3 grandes catégories: Création, Structure, Comportement

23 patterns génériques

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)

1. Patrons de création

Utilisé pour créer des objets

Quelques bonnes pratiques :
http://www.vincehuston.org/dp/creational_rules.html

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

Notion de Fabrique (Factory)

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

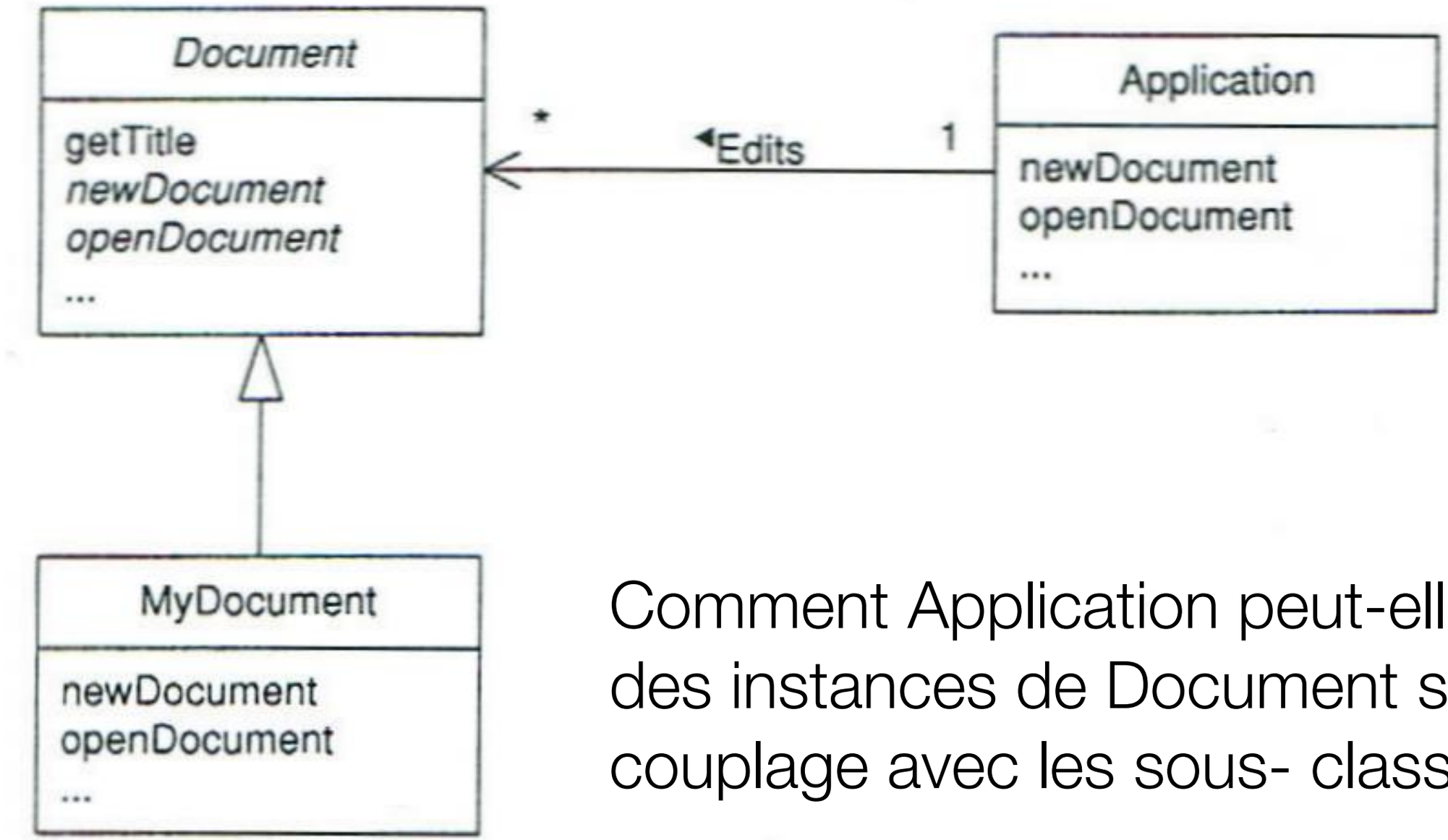
- la classe a un nom
- permet de gérer facilement plusieurs méthodes de construction avec des signatures similaires
- peut retourner plusieurs types d'objets (polymorphisme)

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 Tactiques...

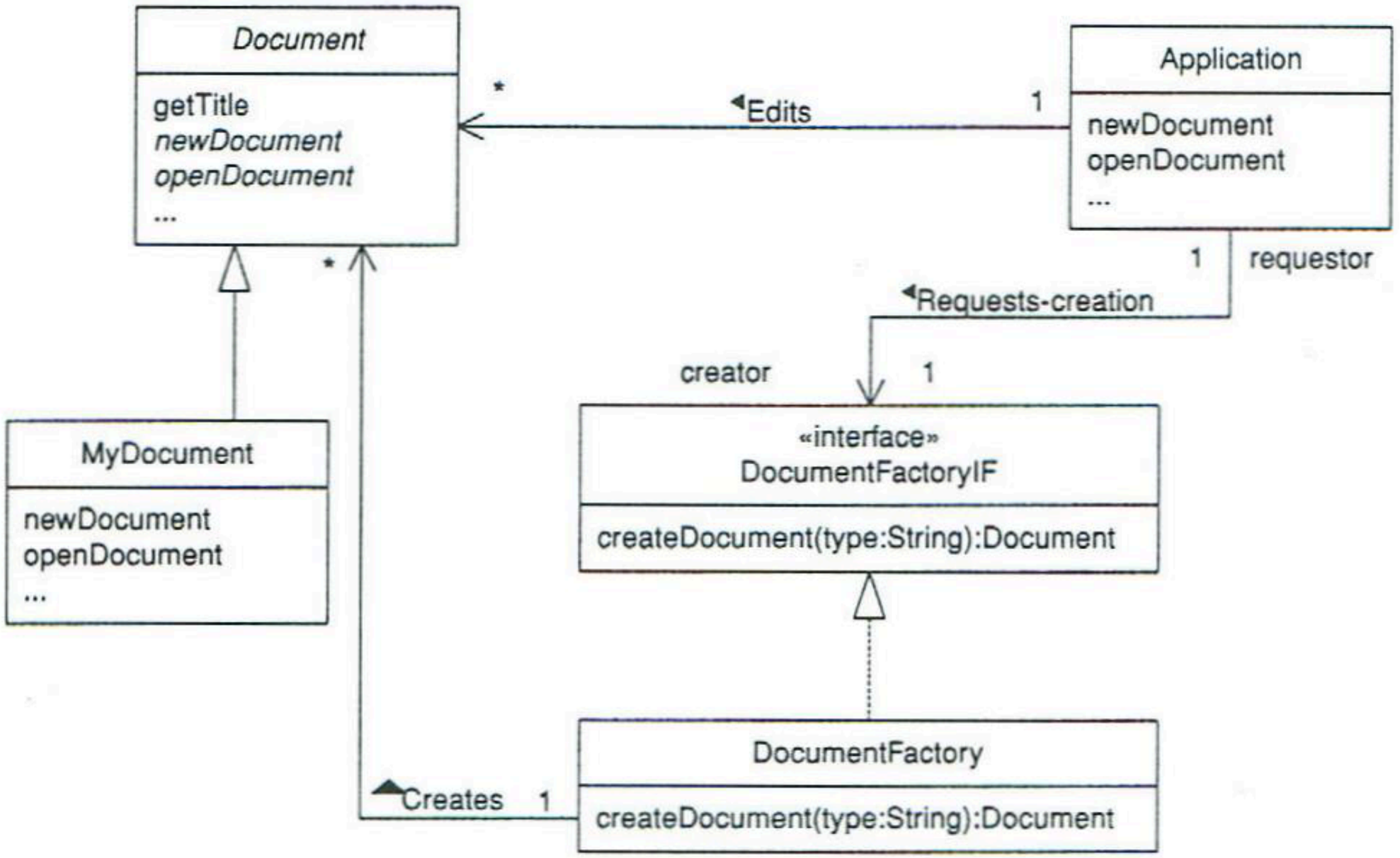
Factory - Fabrique



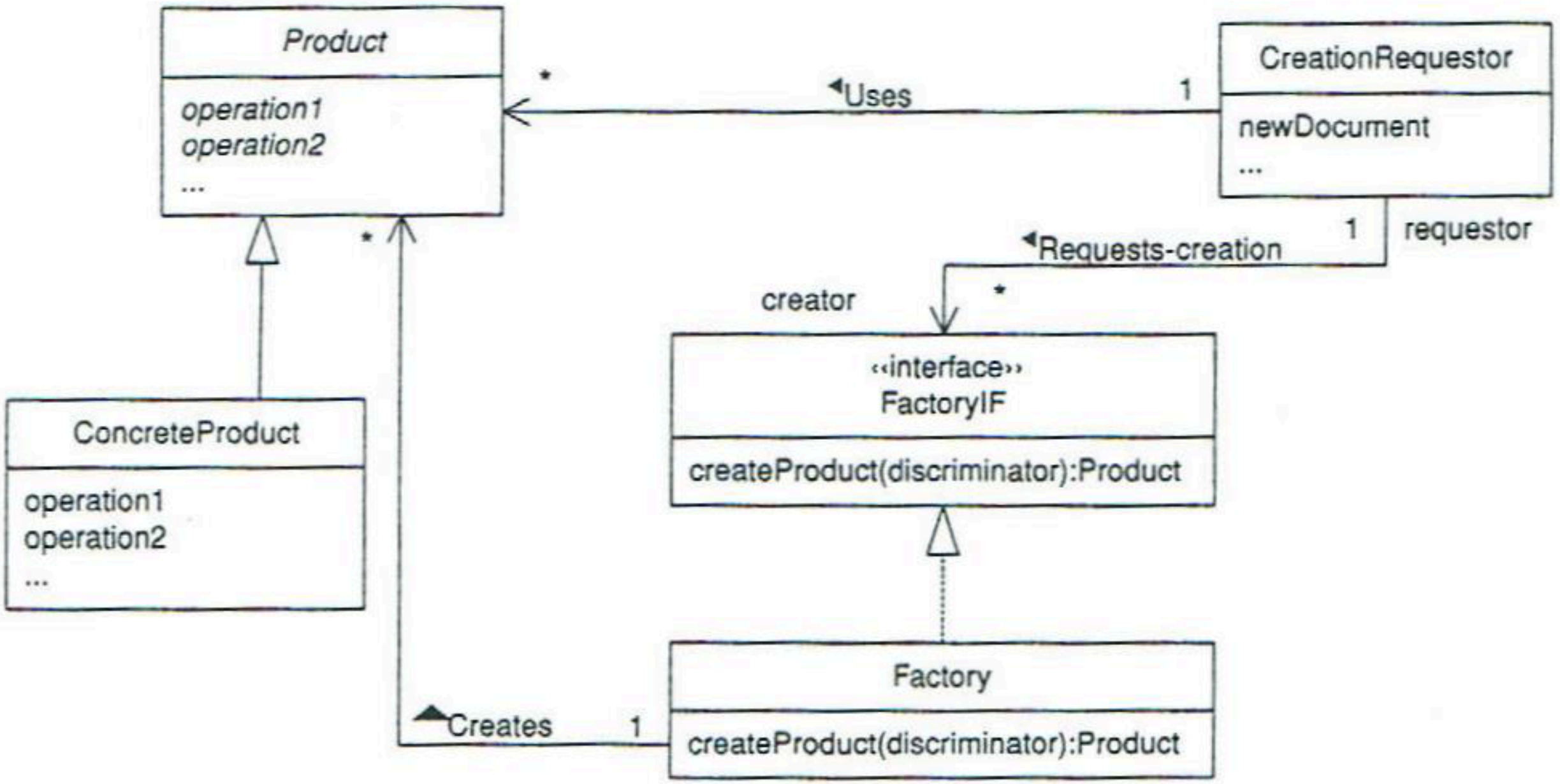
Comment Application peut-elle créer des instances de Document sans couplage avec les sous- classes ?

(T. Horton, CS494)

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



Factory Method Pattern : structure générale



Abstract Factory

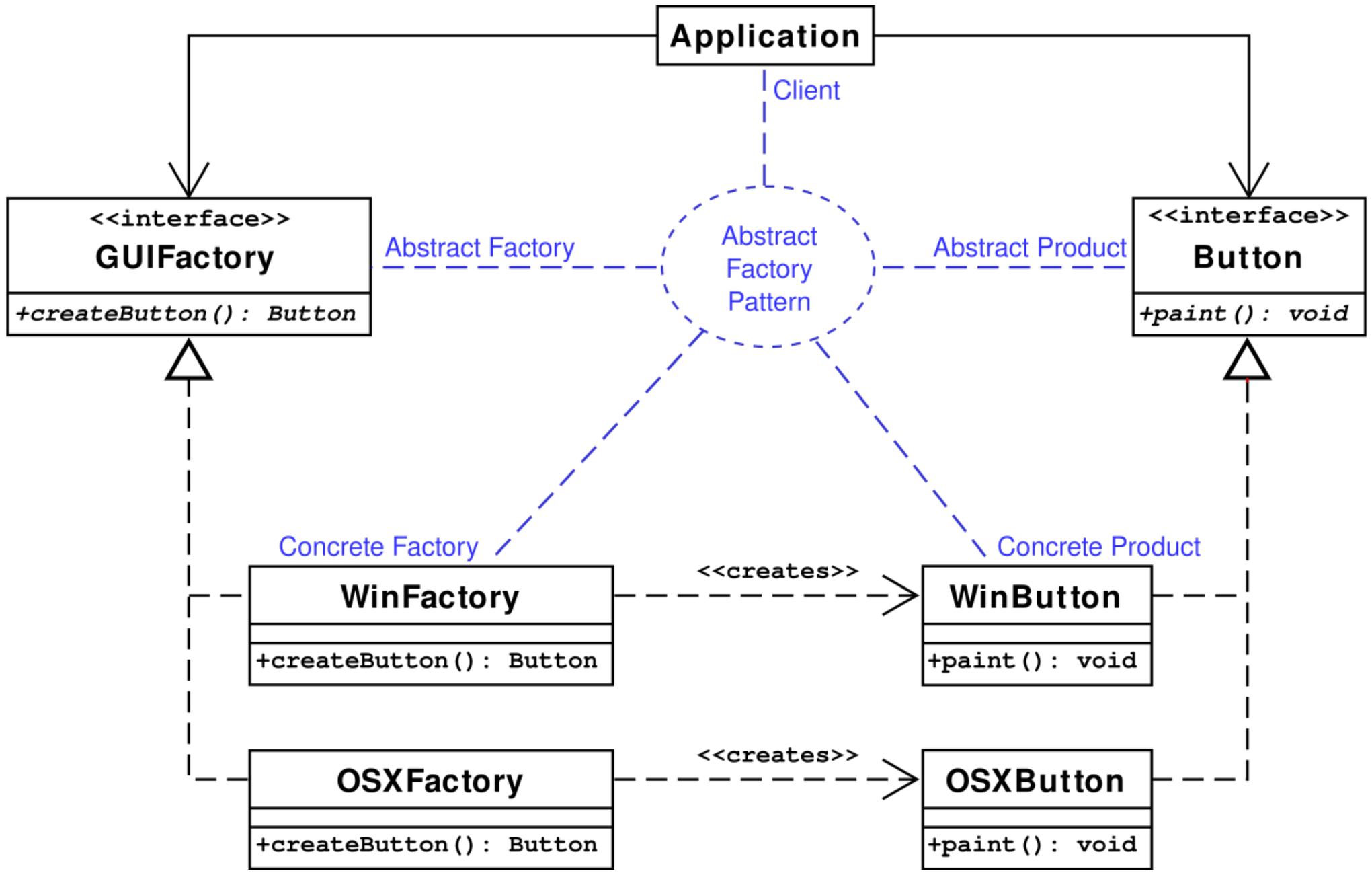
Objectif

- 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 (CC): https://en.wikipedia.org/wiki/Abstract_factory_pattern

Builder (Monteur)

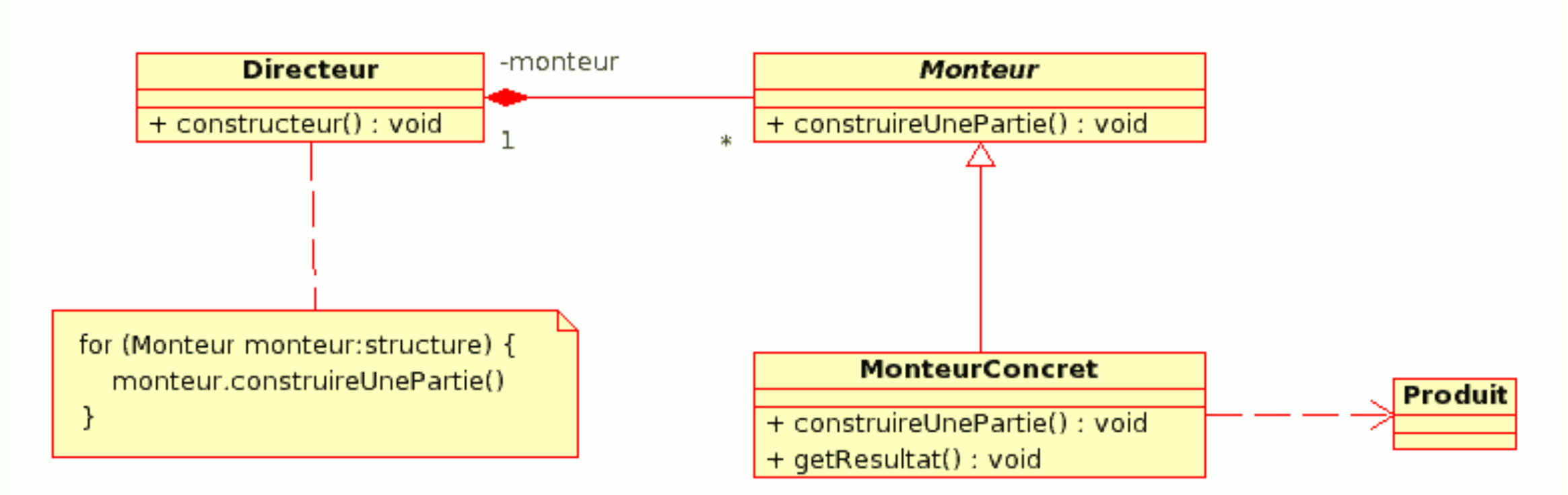
Objectif

- 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 (Monteur)



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

Prototype

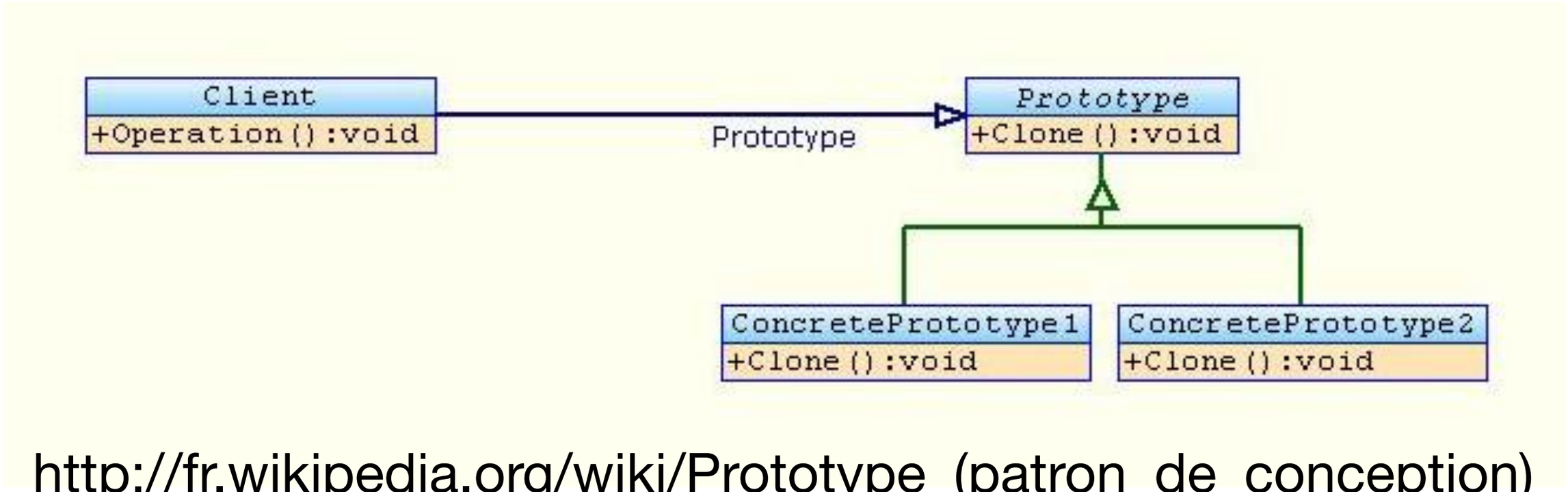
Objectifs

- 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



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

Remarque

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

Singleton

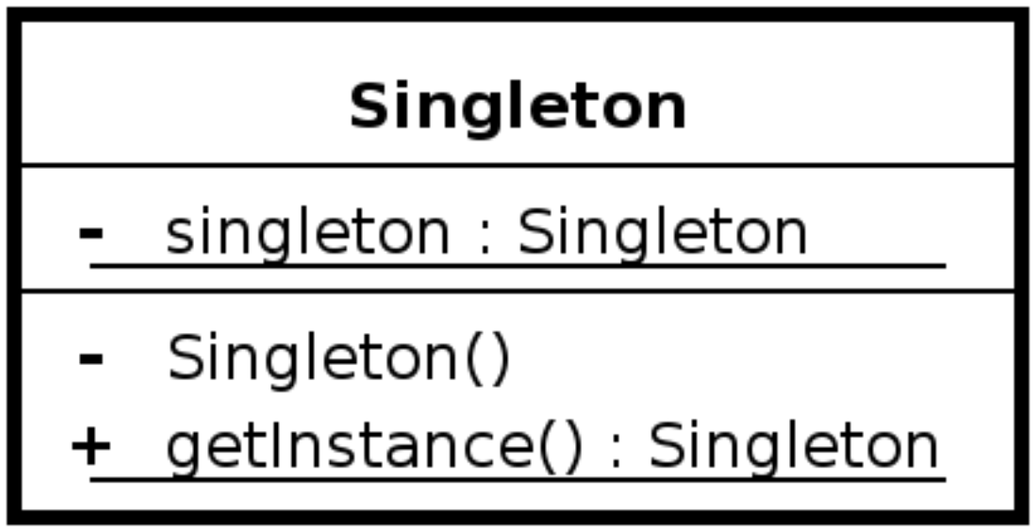
Objectif

- S’assurer d’avoir une instance unique d’une classe, par exemple une seule Factory
  - Point d’accès unique et global pour les autres objets

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



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

Catégories de design patterns

1. **Création**
   - Processus d'instanciation / initialisation des objets
2. **Structure**
   - Organisation d'un ensemble de classes à travers un module (statique)
3. **Comportement**
   - Organisation des rôles pour la collaboration d'objets (dynamique)

Patterns de structure

Utilisés pour composer / cacher / modifier des objects

http://www.vincehuston.org/dp/structural_rules.html

Exemples les plus connus :

- Adapter/Facade/Proxy
- Composite
- Decorator

Patterns de structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

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

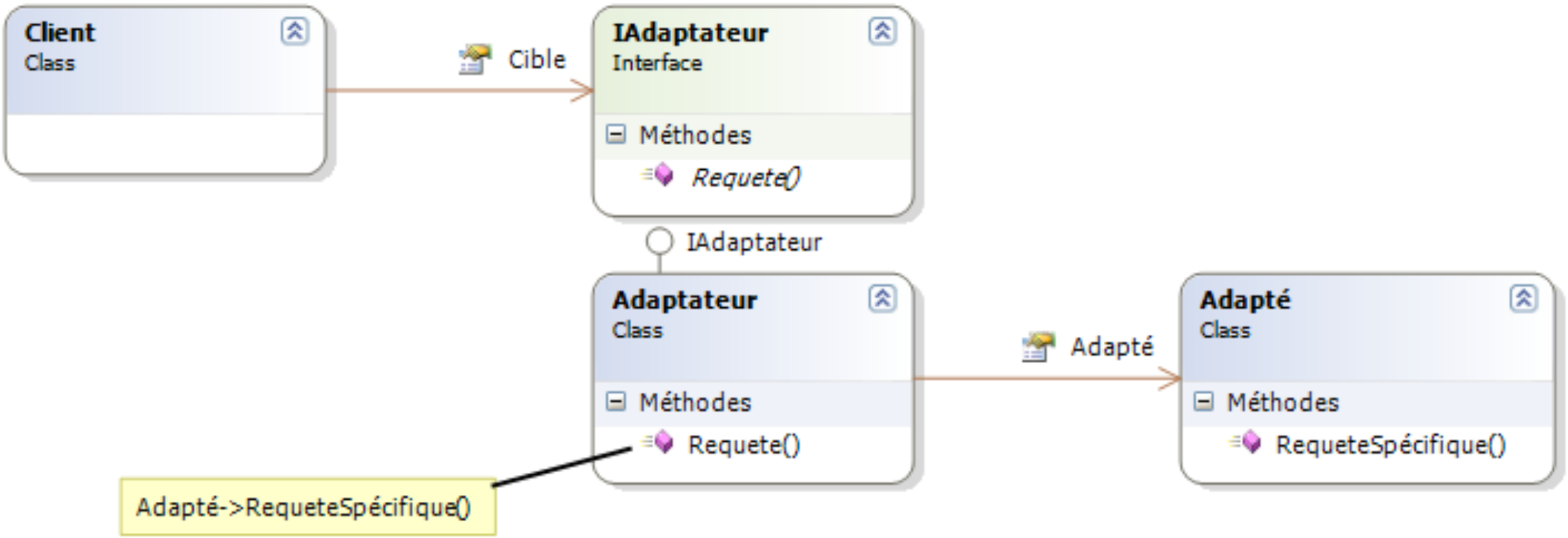
Fonctionnement

- Insérer un niveau d'indirection qui réalise la conversion

Patterns liés

- Indirection, Facade, Proxy

Adaptateur (Adapter, Wrapper)



Source :

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

Patterns de structure

- Adaptateur (Adapter)
- Pont (Bridge)
- **Objet composite (Composite)**
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

Composite

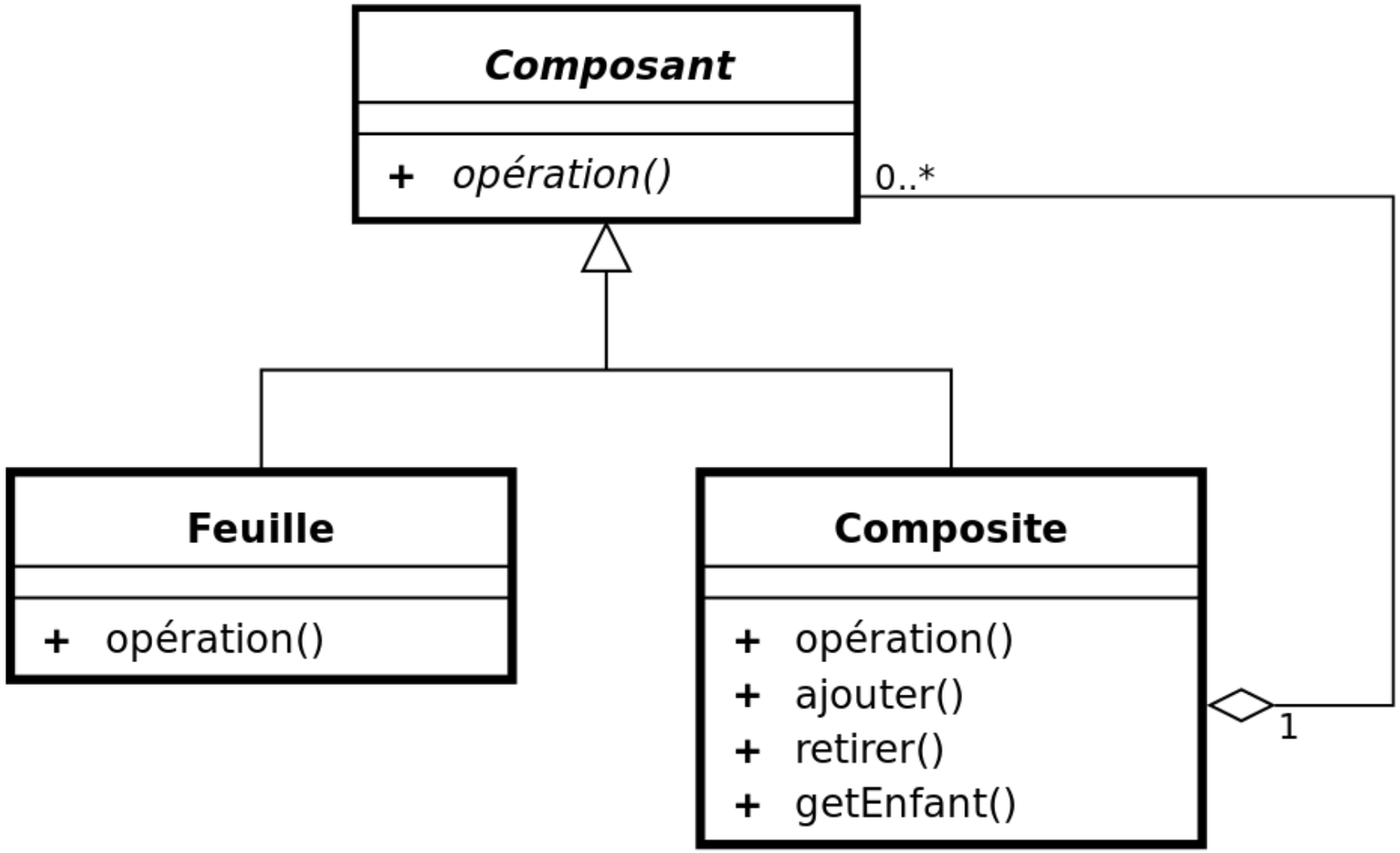
Objectif

- Représenter une structure arborescente d'objets
- Rendre générique les mécanismes de positionnement / déplacement dans un arbre
- Exemple : DOM Node, Graphe de Scène, Widgets, Dossiers…

Fonctionnement

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

Composite



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

Pourquoi une relation d'agrégation et non de composition ?

Patterns de structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- **Décorateur (Decorator)**
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

Décorateur

Objectif

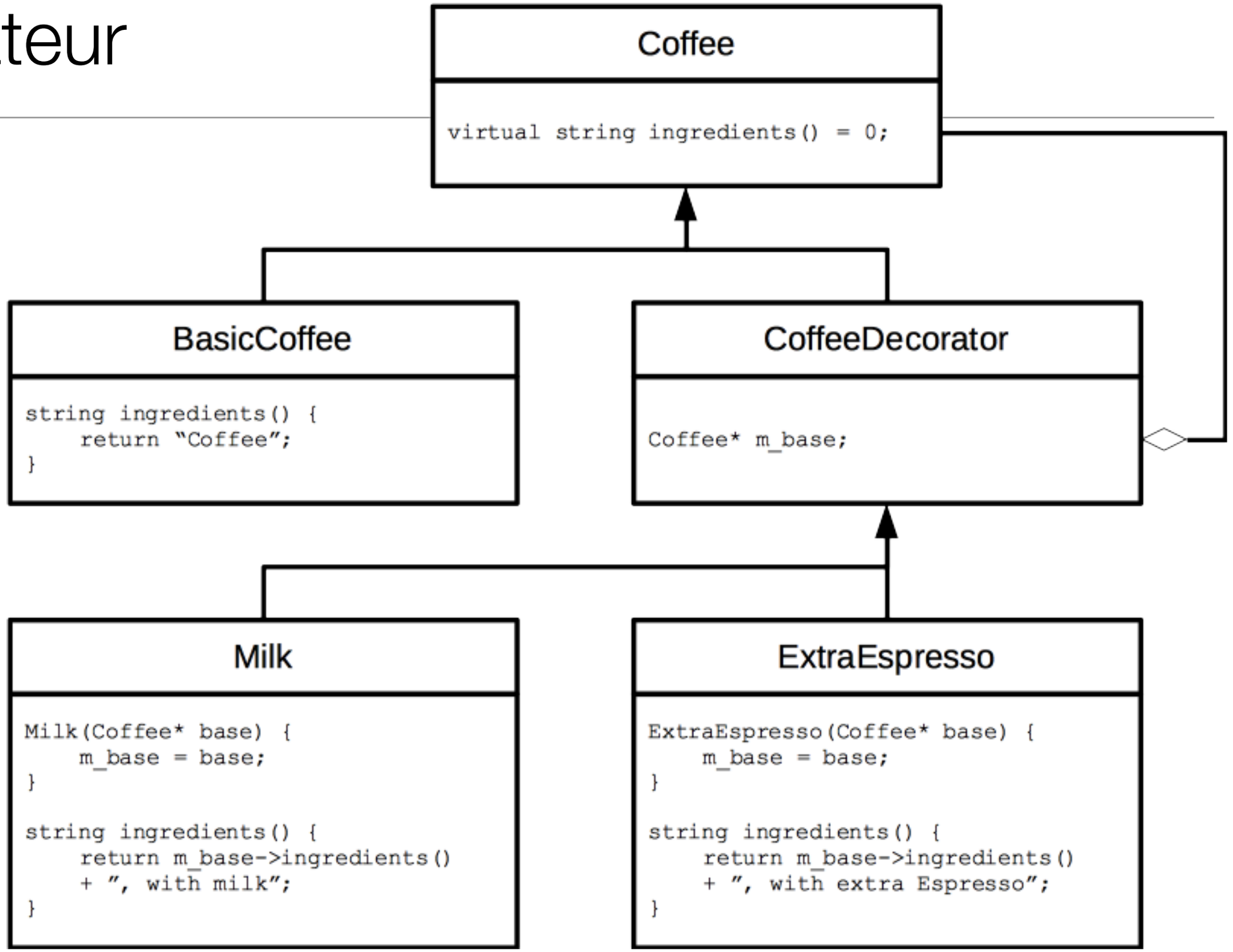
- Résister au changement
- Permettre l'extension des fonctionnalités d'une application sans tout reconcevoir
  - Rappel : les classes doivent être ouvertes à l'extension, mais fermées à la modification

Fonctionnement

- Rajouter des comportements dans une classe qui possède la même interface que celle d'origine
- Appeler la classe d'origine depuis le décorateur

Pattern lié : Proxy

Décorateur


Coffee
virtual string ingredients() = 0;
BasicCoffee
string ingredients() {
return "Coffee";
}
CoffeeDecorator
Coffee* m_base;
Milk
Milk(Coffee* base) {
m_base = base;
}
string ingredients() {
return m_base->ingredients()
+ ", with milk";
}
ExtraEspresso
ExtraEspresso(Coffee* base) {
m_base = base;
}
string ingredients() {
return m_base->ingredients()
+ ", with extra Espresso";
}

Patterns de structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- **Façade (Facade)**
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

Façade

Objectif

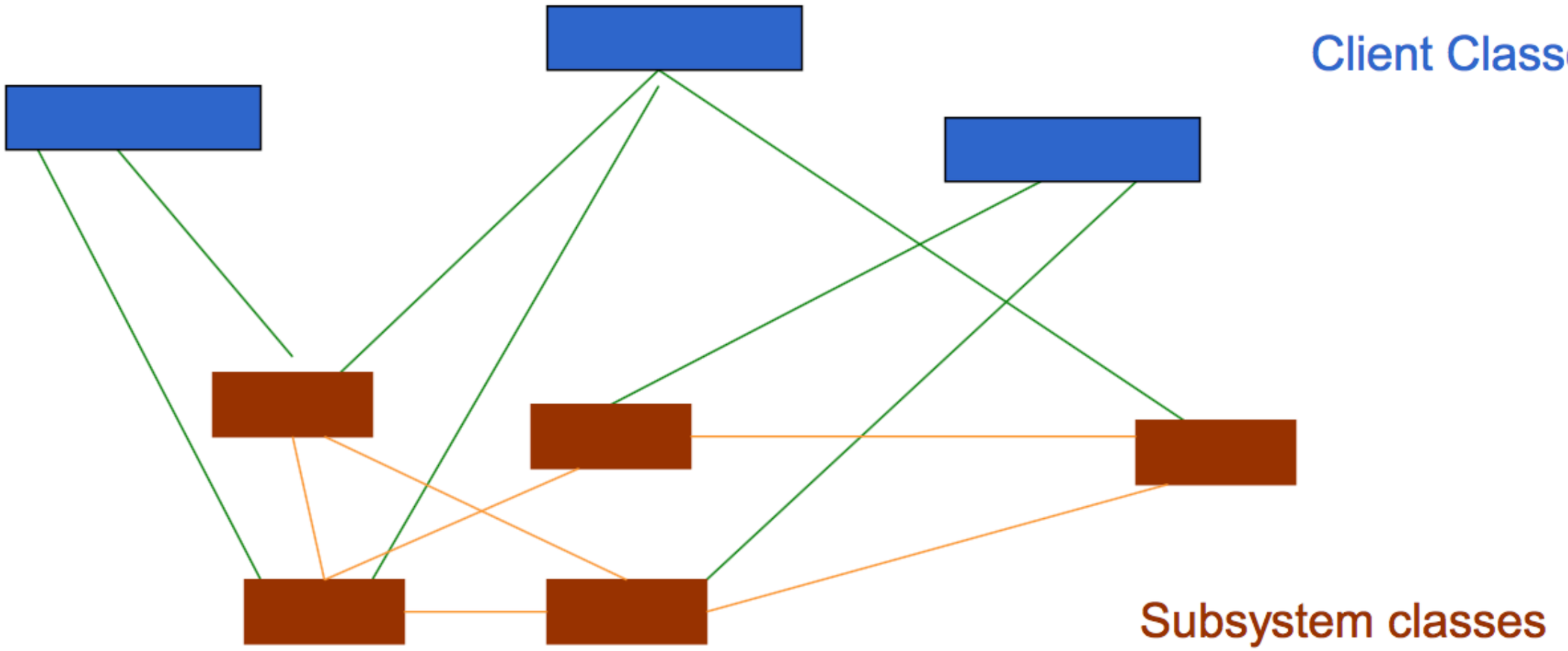
- Cacher une interface / implémentation complexe
- rendre une bibliothèque plus facile à utiliser, comprendre et tester;
- rendre une bibliothèque plus lisible;
- réduire les dépendances entre les clients de la bibliothèque

Fonctionnement

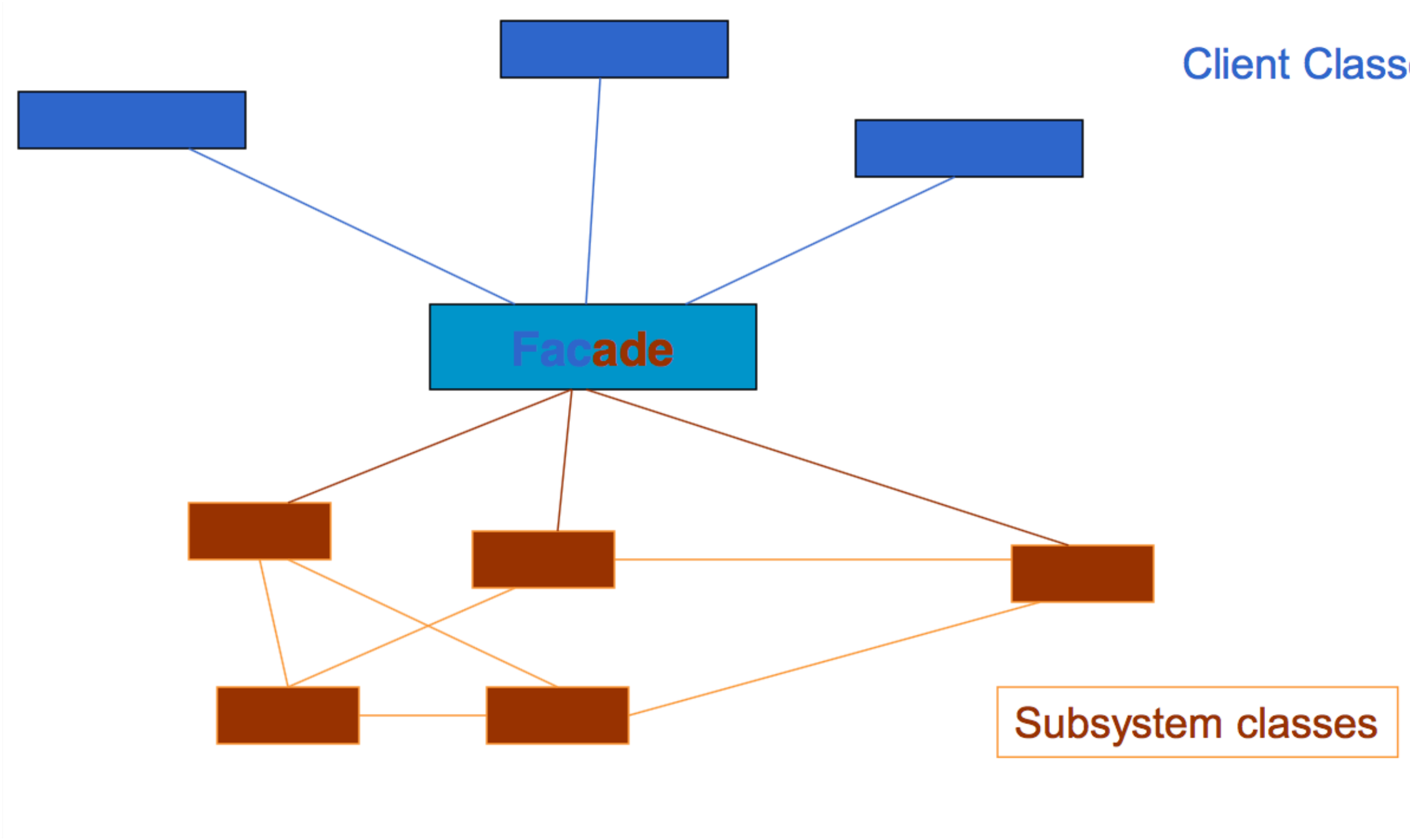
- Fournir une interface simple regroupant toutes les fonctionnalités utiles aux clients

Patterns liés: Indirection, Adapteur

Façade



Façade



Patterns de structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- **Proxy** (Proxy)

Proxy

Objectif

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

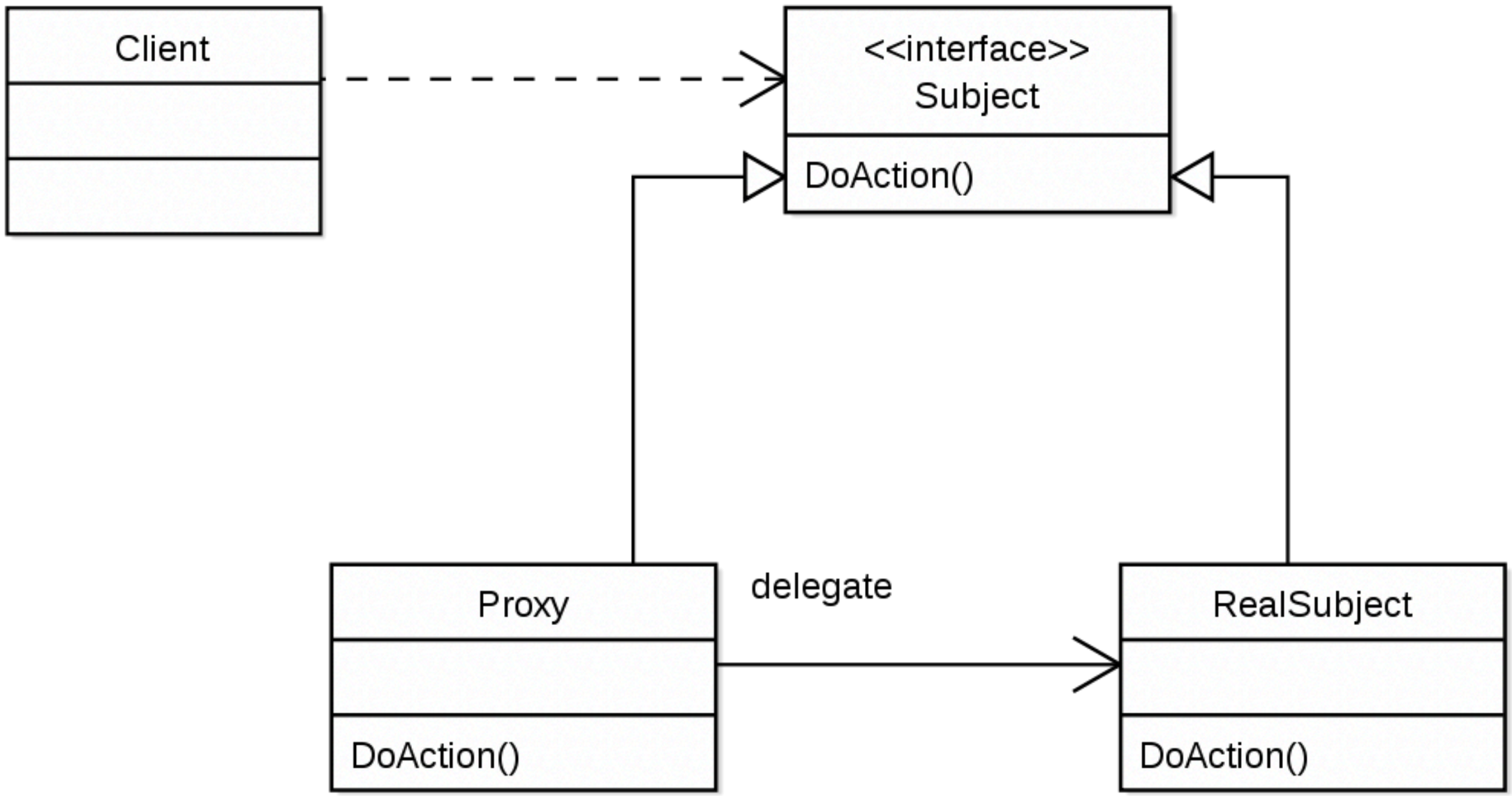
Fonctionnement

- 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

Patterns de structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

Catégories de design patterns

1. **Création**
   - Processus d'instanciation / initialisation des objets
2. **Structure**
   - Organisation d'un ensemble de classes à travers un module (statique)
3. **Comportement**
   - Organisation des rôles pour la collaboration d'objets (dynamique)

Patterns de comportement

Utilisé pour contrôler / interagir avec des objets

http://www.vincehuston.org/dp/behavioral_rules.html

Exemples notables:

- Itérateur (Iterator)
- Commande (Command)
- Visiteur (Visitor)
- Observeur (Observer)

Patterns de comportement

- Chaîne de responsabilité (Chain of responsibility)
- Commande (Command)
- Interpréteur (Interpreter)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Mémento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)
- Fonction de rappel (Callback)

Itérateur

Objectif :

- traverser une collection
- Découpler la structure de donnée des algorithmes

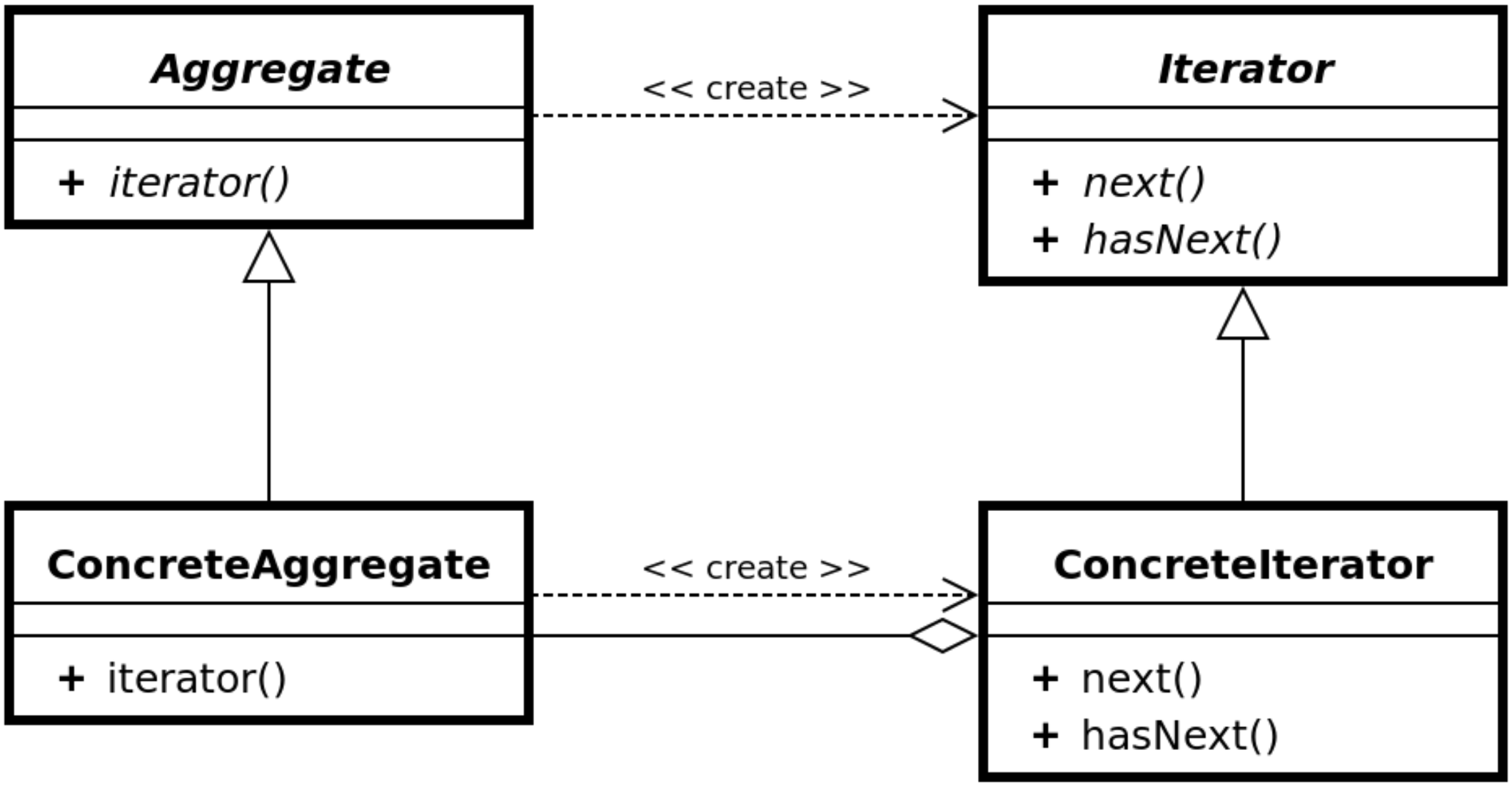
Fonctionnement

- Une collection est gérée par un objet conteneur
- Un objet Iterator est return par la structure de donnée

Intégré dans de nombreux langages.

Itérateur

https://en.wikipedia.org/wiki/Iterator_pattern

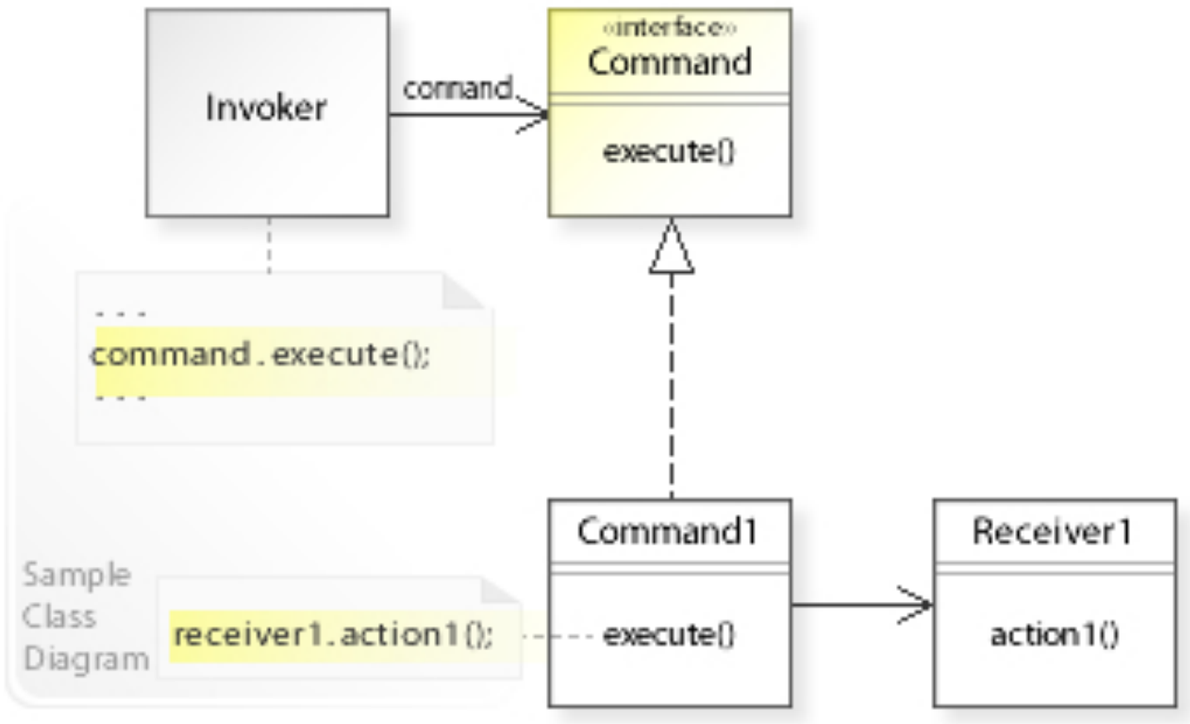


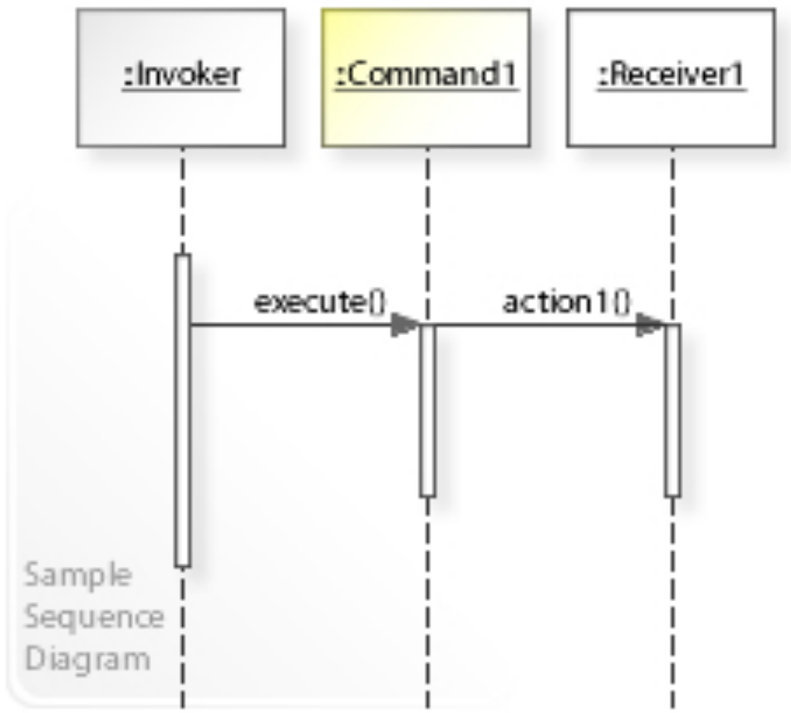
Commande

Objectif:

- Découpler l'appel d'une méthode de son exécution
- Tracer les appels

Fonctionnement



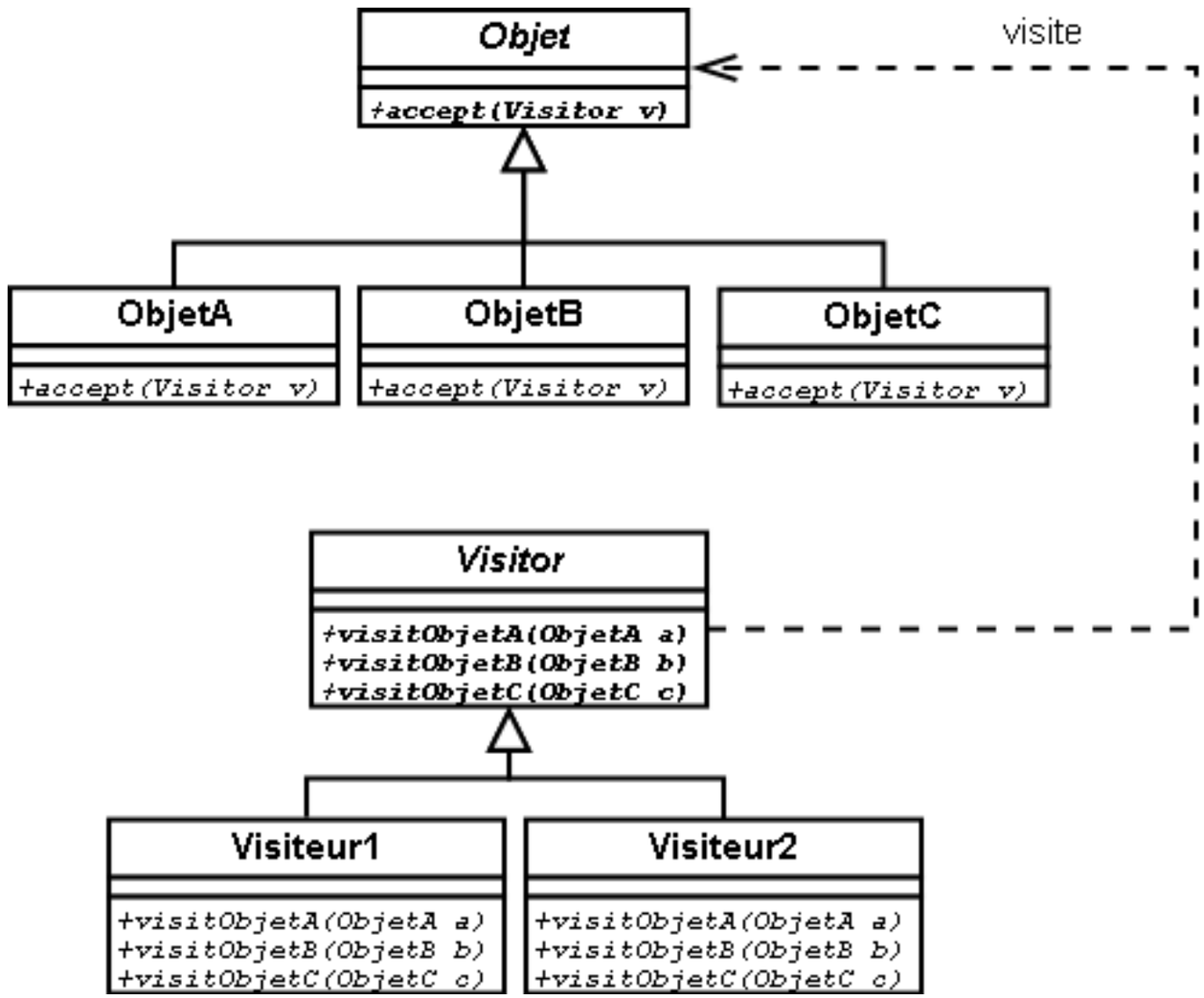


https://en.wikipedia.org/wiki/Command_pattern

Visiteur

Objectif :

- Appliquer une opération à tout les objets d'une hiérarchie
- Ajout de fonctionnalités aux objets sans changer les modifier directement
- Complémente le pattern Composite



Observateur / Observable

Voir le MVC et le début du cours.

Patterns de comportement

- Chaîne de responsabilité (Chain of responsibility)
- Commande (Command)
- Interpréteur (Interpreter)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Mémento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)
- Fonction de rappel (Callback)

Plan

- Introduction
- Patrons GRASP
- Patrons architecturaux
- Design patterns
- **UI patterns**
- Antipatterns

UI Patterns

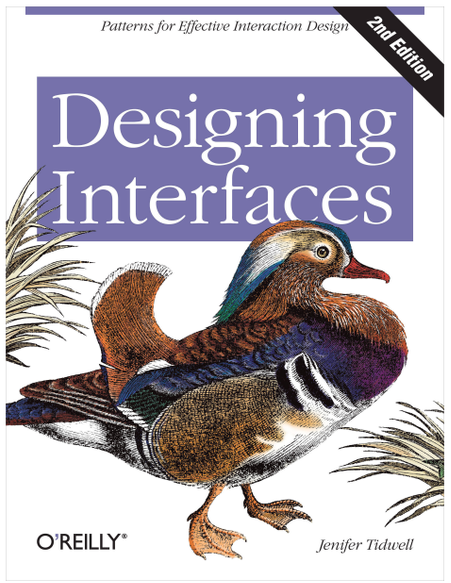
S’applique aussi à l’IHM

Pas forcément niveau code mais

- architecture de l’application : user flow, navigation
- interface : mise en page, couleur, organisation des widgets
- interaction : vocabulaire de geste, commandes vocales, etc.

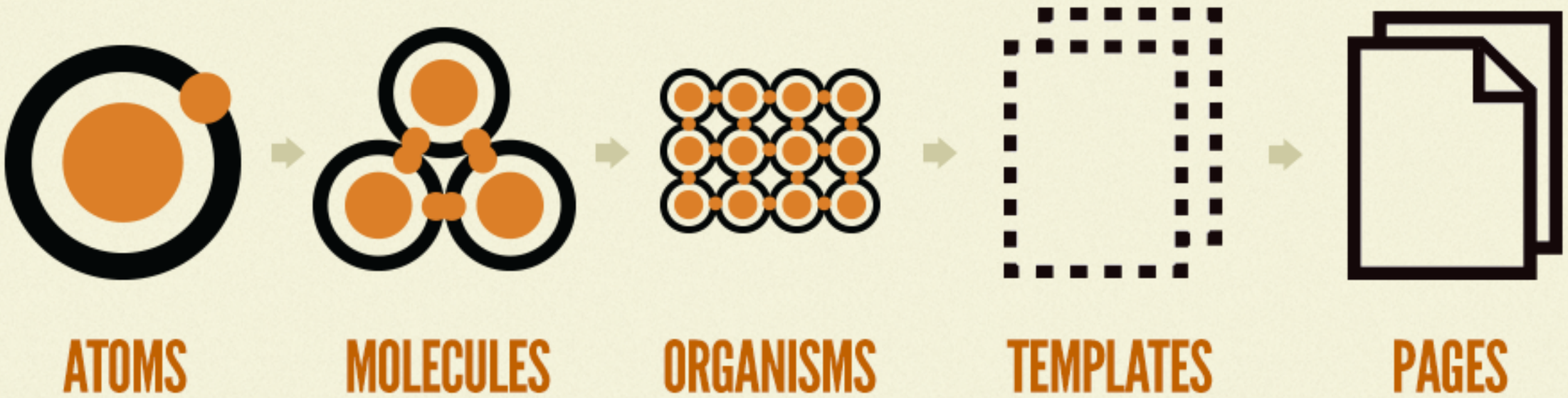
Références:

- “Designing Interfaces” de Jenifer Tidwell, O'Reilly Books



Atomic Web Design

http://atomicdesign.bradfrost.com/



http://bradfrost.com/blog/post/atomic-web-design/

Material Design de Google

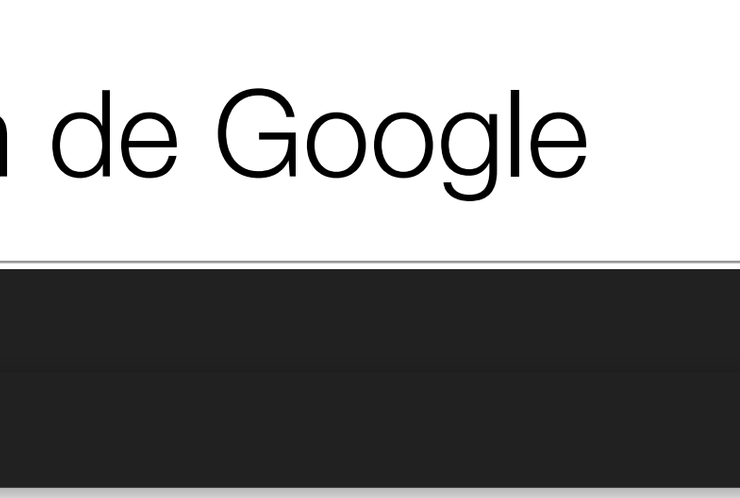
https://material.io/

MATERIAL DESIGN

DESIGN GUIDELINES

The Material Design guidelines are a living document of visual, interactive, and motion guidance.

*LAST UPDATED SEPT 2017



ICONS

Visit our library of over 900 material icons.

*41 NEW ICONS

MATERIAL COMPONENTS

Create beautiful apps with modular and customizable UI components.

*NEW WEBSITE

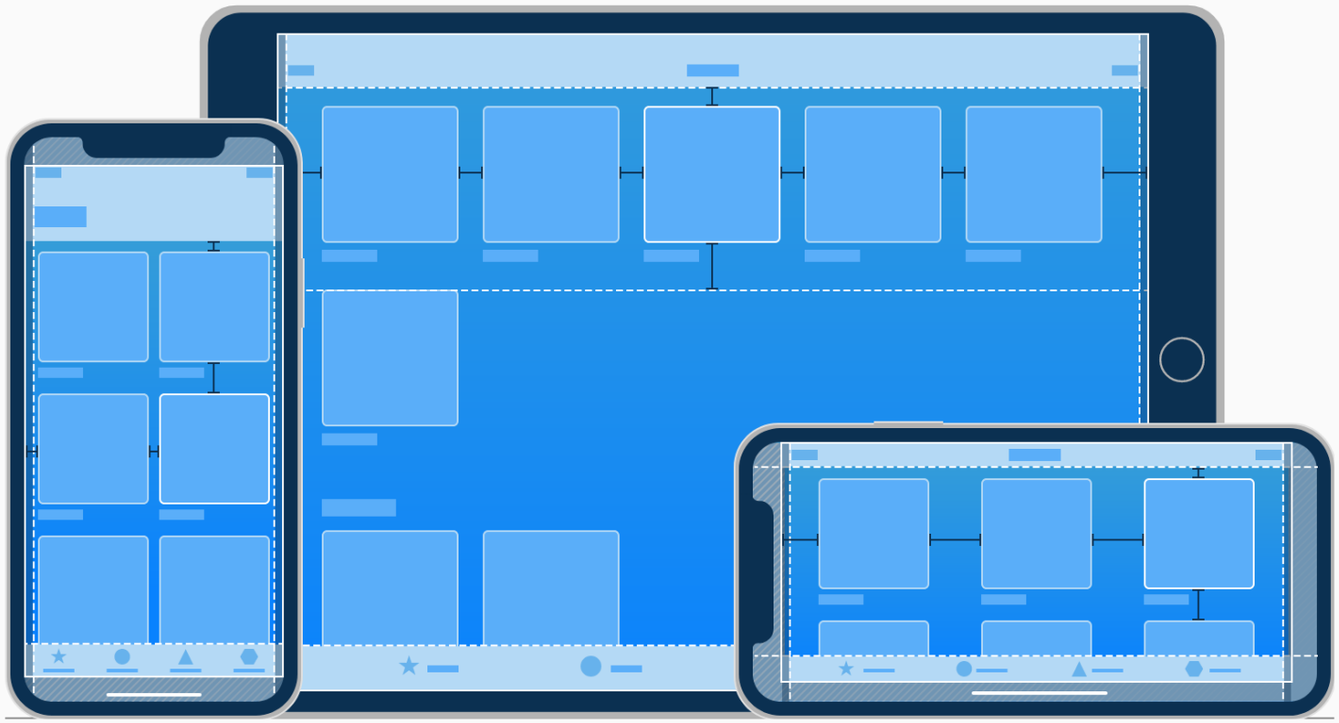
TOOLS

Human Interface Guidelines d'Apple

https://developer.apple.com/ios/human-interface-guidelines/overview/themes/

Human Interface Guidelines

iOS



iOS Design Themes

As an app designer, you have the opportunity to deliver an extraordinary product that rises to the top of the App Store charts. To do so, you'll need to meet high expectations for quality and functionality.

Three primary themes differentiate iOS from other platforms:

Sur mobile

http://www.mobile-patterns.com/

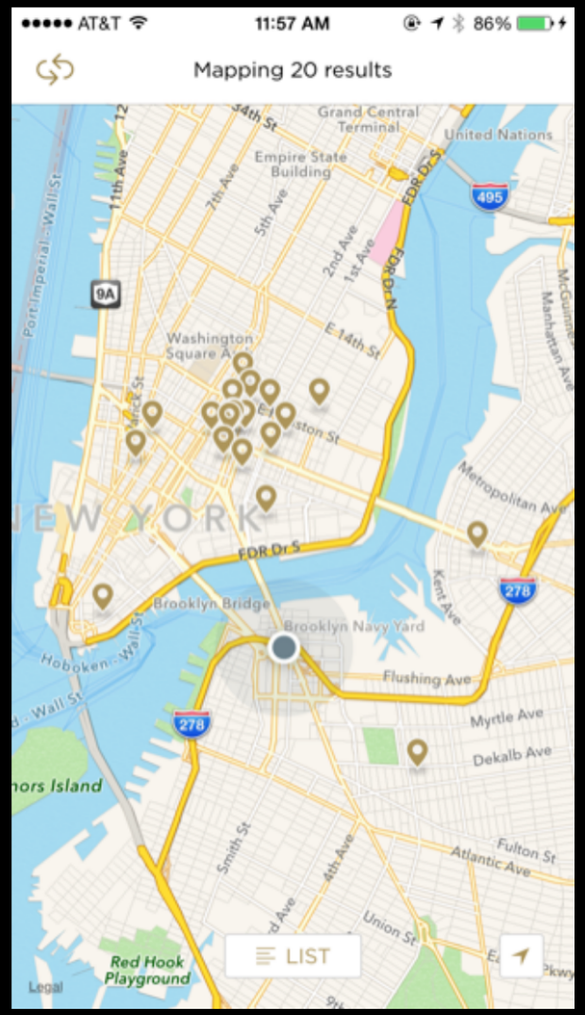
MOBILE PATTERNS

ALL

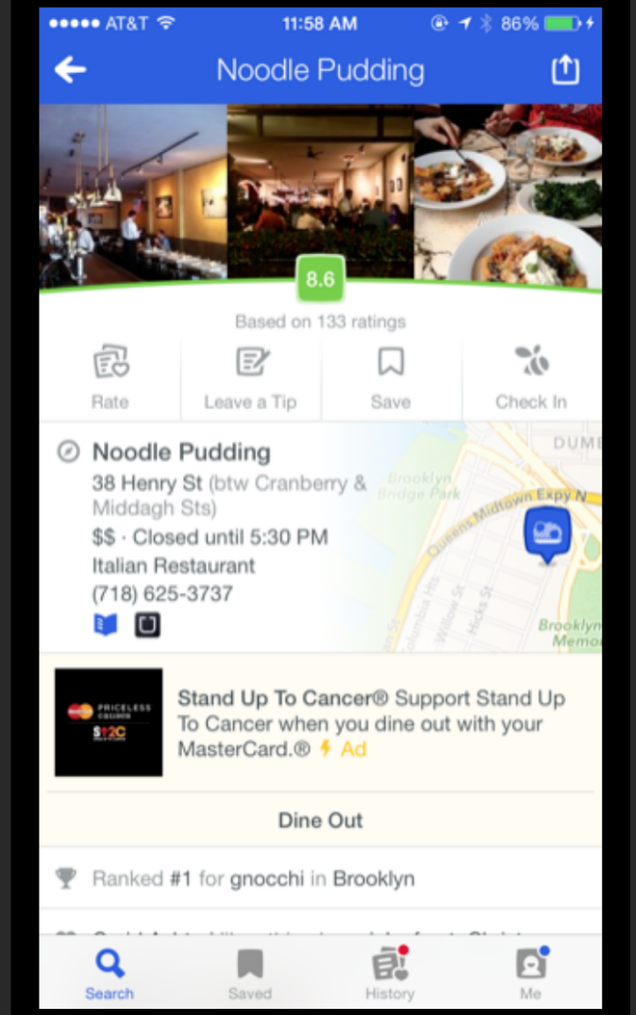
Search

RECENTLY ADDED
CALENDAR
CAMERA CONTROLLER
COACH MARKS
COMMENT COMPOSE
COMPOSE SCREENS
CUSTOM NAVIGATION
DETAIL VIEWS
EMPTY DATA SETS
FEEDS
LISTS
LOG IN
MAPS
GALLERIES
POPOVERS
SEARCH
SETTINGS
SIGN UP FLOWS
SPLASH SCREENS
STATS
TIMELINES
USER PROFILES
WALKTHROUGHS

MAPS



Reserve (iPhone) maps



Foursquare (iPhone) detail views, maps

Instagram (iPhone) maps

Plan

- Introduction
- Patrons GRASP
- Patrons architecturaux
- Design patterns
- UI patterns
- **Antipatterns**

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…

Conclusion

On a vu assez précisément les patterns les plus généraux (GRASP)

On a survolé les autres
-> les étudier et en connaître une cinquantaine