Ingénierie Logicielle

Les Design Patterns

Plan

- Objectifs d'ingénierie logicielle
- Les Designs patterns
- Patterns du GOF

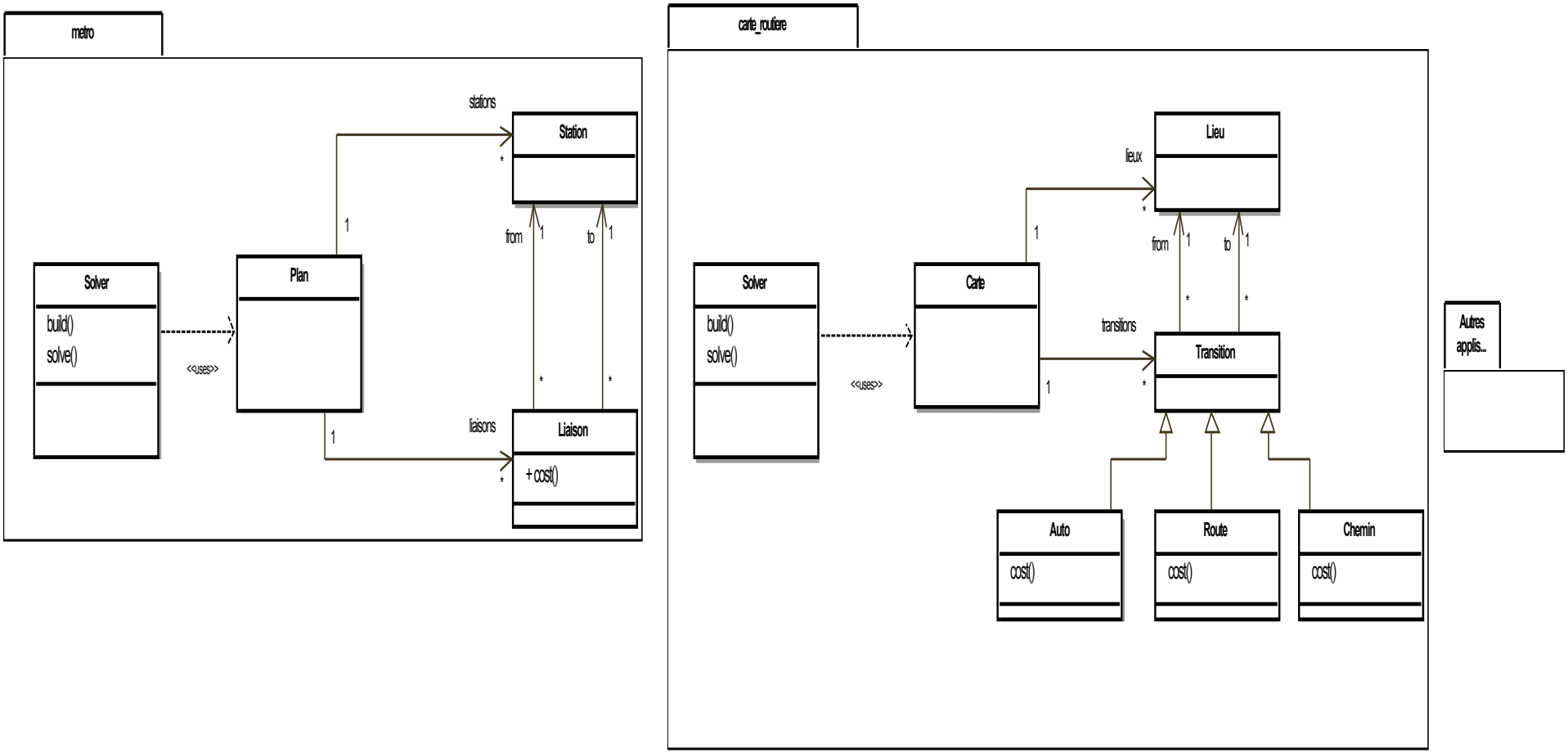
IL

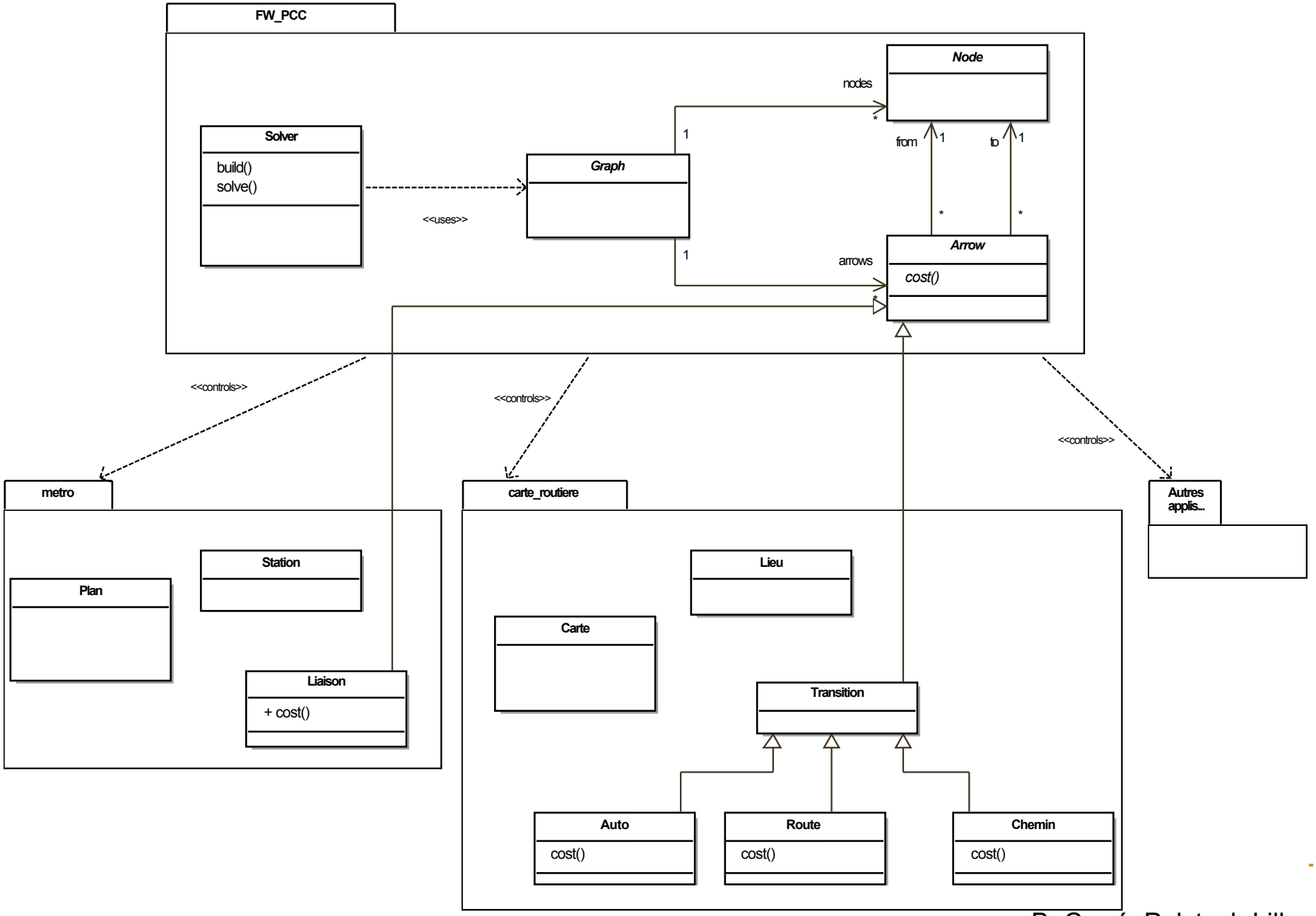
- Concevoir bien
- Rationaliser, fiabiliser la production logicielle
  - détecter les erreurs au plus tôt
  - utiliser des solutions (logicielles mais aussi de conception) éprouvées
- Capitaliser les produits et les pratiques
  - productivité
  - ne pas « refaire » avec le risque d'erreurs de programmation, mais aussi de conception, d'analyse...

IL

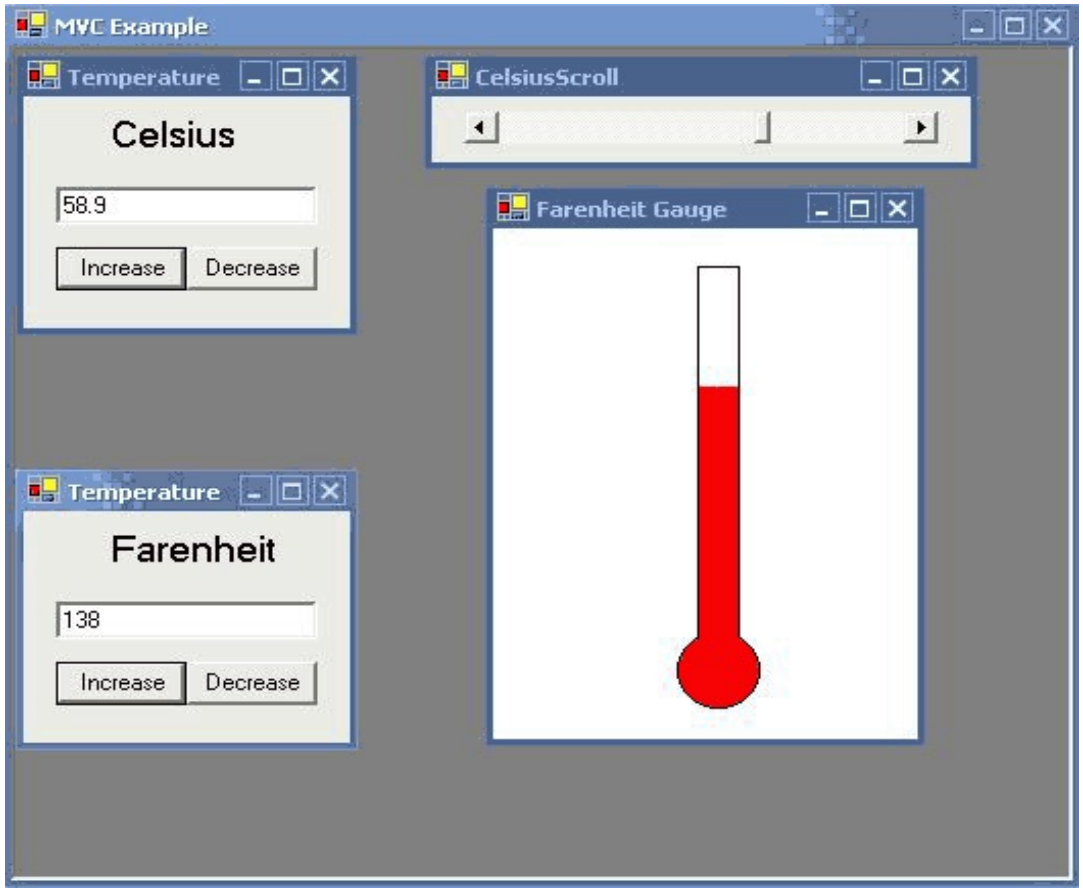
- Réutiliser
  - du logiciel éprouvé
    - bibliothèques, API
    - frameworks extensibles
      - {composants} + une architecture de fonctionnement
      - « moteur générique »
  - mais aussi « plus en amont »
    - des schémas ou patrons de conception
      - ***cours Design patterns***
    - des modèles: de produits (UML) mais aussi de processus
      - ***cours IDM (Ingénierie Dirigée par les Modèles)***

Démarche de conception FW
Inversion Of Control / Dependencies (IOC / IOD)

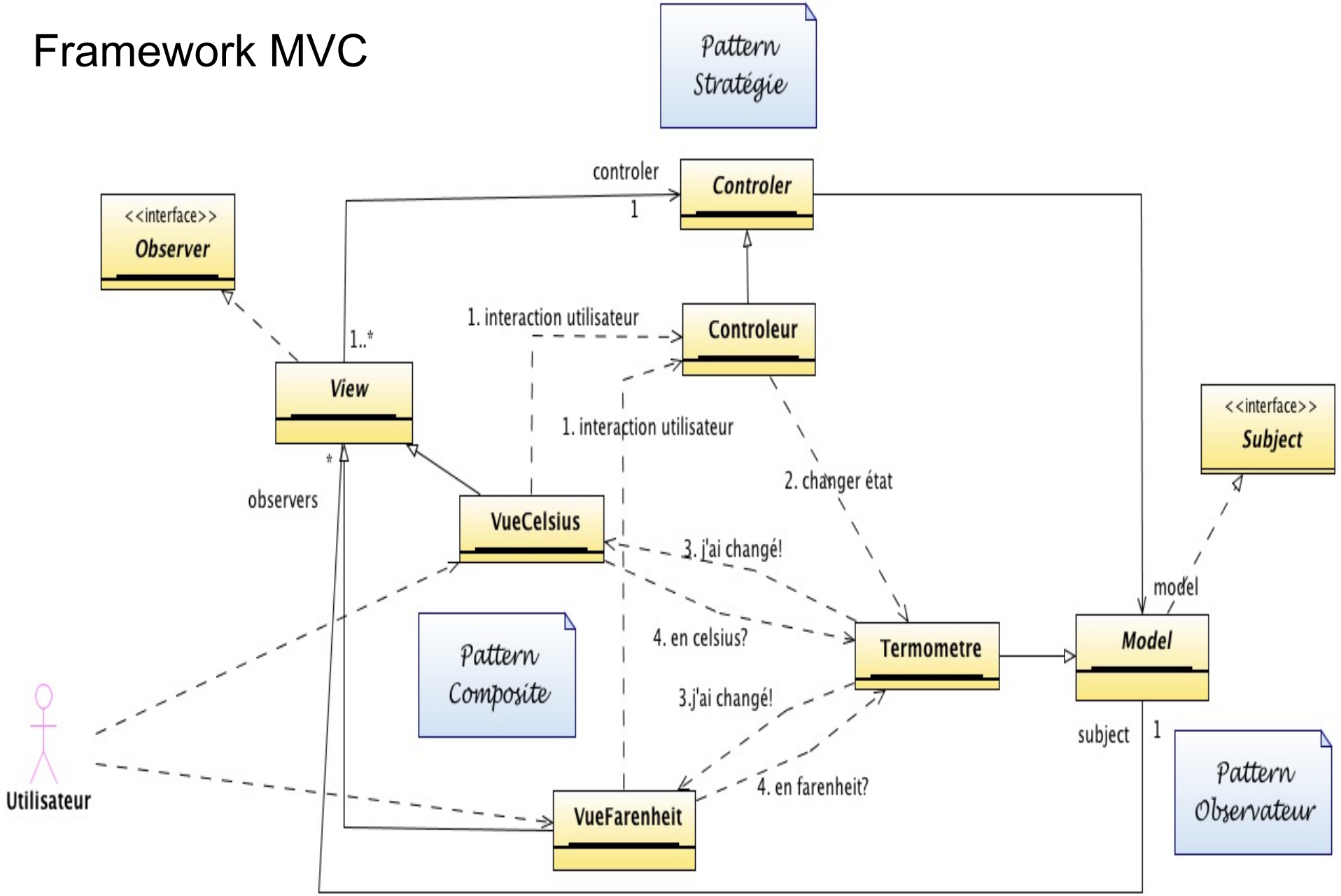



FW_PCC
Node
nodes
*
Solver
build()
solve()
1
Graph
from
1
to
1
<<uses>>
*
*
1
Arrow
arrows
cost()
*
<<controls>>
<<controls>>
<<controls>>
metro
carte_routiere
Autres applis...
Station
Plan
Lieu
Carte
Liaison
+ cost()
Transition
Auto
cost()
Route
cost()
Chemin
cost()

Framework MVC



Framework MVC

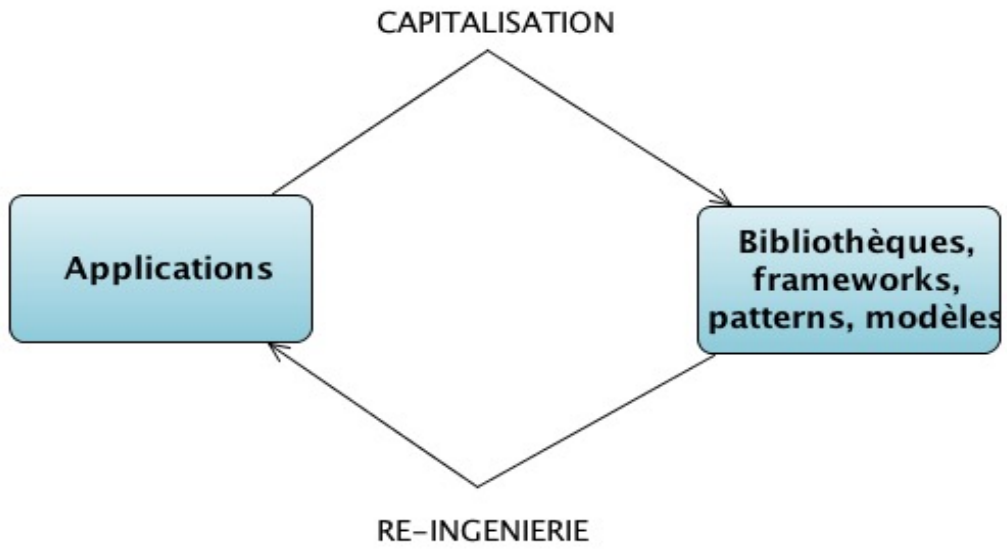


IL

- Catégories logicielles et métiers
  - concepteur d'*applications*
    - savoir utiliser, réutiliser, assembler
  - concepteur d'*outillage (générique)*
    - « concevoir pour réutiliser »
      - bibliothèques de classes, de composants
      - frameworks et architectures ouvertes
      - schémas de conception
      - modèles réutilisables
    - « standards » (éditeur), maison (interne à une société d'ingénierie) ou perso (constituer mon propre outillage récurrent)

IL

- Re-concevoir (mieux)
  - acquisition de nouvelles technologies
  - mais aussi de nouvelles pratiques, maison (formalisation de l'expérience) ou standards
- Montée en abstraction (méta)



Les Designs patterns

Principes généraux

Design Patterns (patrons de conception)

« Chaque moule (pattern) décrit un **problème** qui réapparaît de manière régulière dans notre environnement, puis il décrit le **noyau de la solution** du problème, de telle manière que vous pouvez réutiliser cette solution autant de fois que vous le voulez, sans jamais réaliser la solution finale deux fois de suite de la même manière »

Christopher Alexander, Architecte
(à l'origine de la notion de pattern)

Design Patterns (patrons de conception)

- Abstraction de schémas de conception récurrents identifiés et définis par des experts
  - synthèse de situations réelles (ce ne sont pas des créations de toute pièce)
  - orientés problèmes
    - effet « ah ah !» ou « tiens tiens j'ai déjà rencontré ce problème quelque part !»
  - indépendants des situations
    - contexte (application)
    - langages, technologies et techniques d'implantation
    - même s'ils sont plus ou moins faciles à mettre en oeuvre selon les capacités à disposition

Design Patterns

- capturent une forte expertise de conception
  - solutions reconnues (ne pas réinventer la roue), problèmes déjà résolus
  - efficacité et qualités éprouvées par un grand nombre de cas
- facilitent la réutilisation lors de la conception
  - limitent les alternatives infructueuses
  - mais réclament un certain recul sur l'activité de conception...
- à comparer aux algorithmes...

Design Patterns

- Langage métier (« vocabulaire ») synthétique des concepteurs

  « J'ai créé une classe Releves qui mémorise les relevés météo et qui peut être interrogée par toute sorte d'applications, comme des afficheurs de température, de pression, de cartes, des outils statistiques, des applications de reporting. Plus encore, la mise à jour est automatique: quand un nouveau relevé est enregistré les applications sont informées et peuvent se mettre à jour. On peut même ajouter de nouvelles applications dynamiquement, il suffit qu'elle s'enregistre auprès de l'objet Releves »

  =>

  « J'ai appliqué le pattern Observateur entre une classe de relevés et des applications utilisatrices comme des afficheurs de température, de pression, de cartes, des outils statistiques, des applications de reporting. »

  [Freeman & al. 2005]

Design Patterns

- langage puissant, efficace et précis
  - en un seul terme: tout un implicite de conception, des propriétés, de qualités et aussi de contraintes
- facilite la productivité à la conception
  - s'abstraire des détails d'implémentation et donc consacrer plus de temps à la conception (en réunions...)
- facilite les échanges
  - langage « métier » (qui peut s'élaborer progressivement dans une équipe de conception)
  - côté « transfert de compétences » de conception aux collaborateurs, aux développeurs

Design Patterns

- Pouvoir de documentation
  - critères de catégorisation des applications, bibliothèques, frameworks, API, ...
  - aident aussi à mieux identifier, comprendre et exploiter les capacités de ces produits logiciels
  - commentaires (synthétiques) et dossiers de conception des applications

Design Patterns

- Evolution de la pratique de conception
  - plutôt que concevoir « from scratch », chercher s'il existe un pattern éprouvé
  - méthodologie orientée pattern « Thinking in Patterns »
    - http//www.mindview.net.Books/TIPatterns

- Cribles de ré-ingénierie
  - pour améliorer et en récupérer les propriétés et qualités
  - pour nettoyer, « dépatouiller » un code qui devient inextricable
  - sans changement de fonctionnalités (non concernés par les specs)
    - ex:
      - un code avec trop de conditionnelles peut suggérer le pattern Etat
      - limiter les dépendances directes entre classes grâce à Fabrique
  - http://www.iam.unibe.ch/~scg/OORP

Design Patterns

- Au dessus des bons principes de conception qui restent les fondamentaux
  - qu'ils exploitent et aident à atteindre
  - auxquels il faut revenir s'il n'existe pas de DP pour le problème et concevoir une solution « from scratch »...
  - qu'il faut toujours privilégier

- Techniques
  - encapsulation
  - polymorphisme
  - abstraction
  - héritage (OO)
  - composition, ...

Design Patterns

- Principes fondamentaux et généraux (OO mais pas seulement...)
  - *couplage faible*
    - encapsulation, séparation interfaces-implémentation
  - *forte cohésion*
    - regrouper des ingrédients selon un critère logique (et le respecter) et une bonne granularité
  - *principe d'ouverture-fermeture* (« open-closed » principle ou d'extension contrôlée)
    - encapsulation
    - héritage, composition, principe de substitution (de Liskov)
    - principe d'inversion des dépendances ou IOC « Inversion Of Control"
      - « Hollywood Principle : Don't call us, we'll call you » !
      - pratique systématisée dans les frameworks

Design Patterns

- ce n'est pas du logiciel!
  - même si il existe des bibliothèques ou des frameworks les facilitant dans une technologie particulière (Observer, Prototype (clone) en Java, ...)
  - même si il existe des modèles de conception tout faits et applicables dans certains IDE pour assister leur application...
    - bibliothèques de patterns (en UML) de NetBeans
    - « templates » de modèles de conception
  - ... mais pas leur choix, ce qui reste le problème du concepteur!

- c'est de la conception
  - réclament une interprétation, une adaptation dans le contexte d'application

Design Patterns

- Les Design pattern sont moins rigides que du logiciel
  - même s'ils sont « formalisés » sous la forme de fiches en catalogues
  - ce ne sont pas des lois mais plutôt des guides (préconisations) à suivre et adapter s'ils ne correspondent pas exactement au problème...
  - ...mais au risque de ne pas en récupérer les qualités éprouvées (expliciter les adaptations et en quoi le problème ne correspond pas).

Design Patterns

- Catalogues de DP
  - « le GOF » : la bible (23 DP fondamentaux)
  - catalogues sur le net
  - catalogue « maisons »
  - catalogue par domaine
    - monde J2EE : http://java.sun.com/blueprints/corej2eepatterns/Patterns
    - systèmes concurents
    - systèmes industriels (RT)
    - systèmes mobiles
- A connaître pour parler patterns...

Designs Patterns

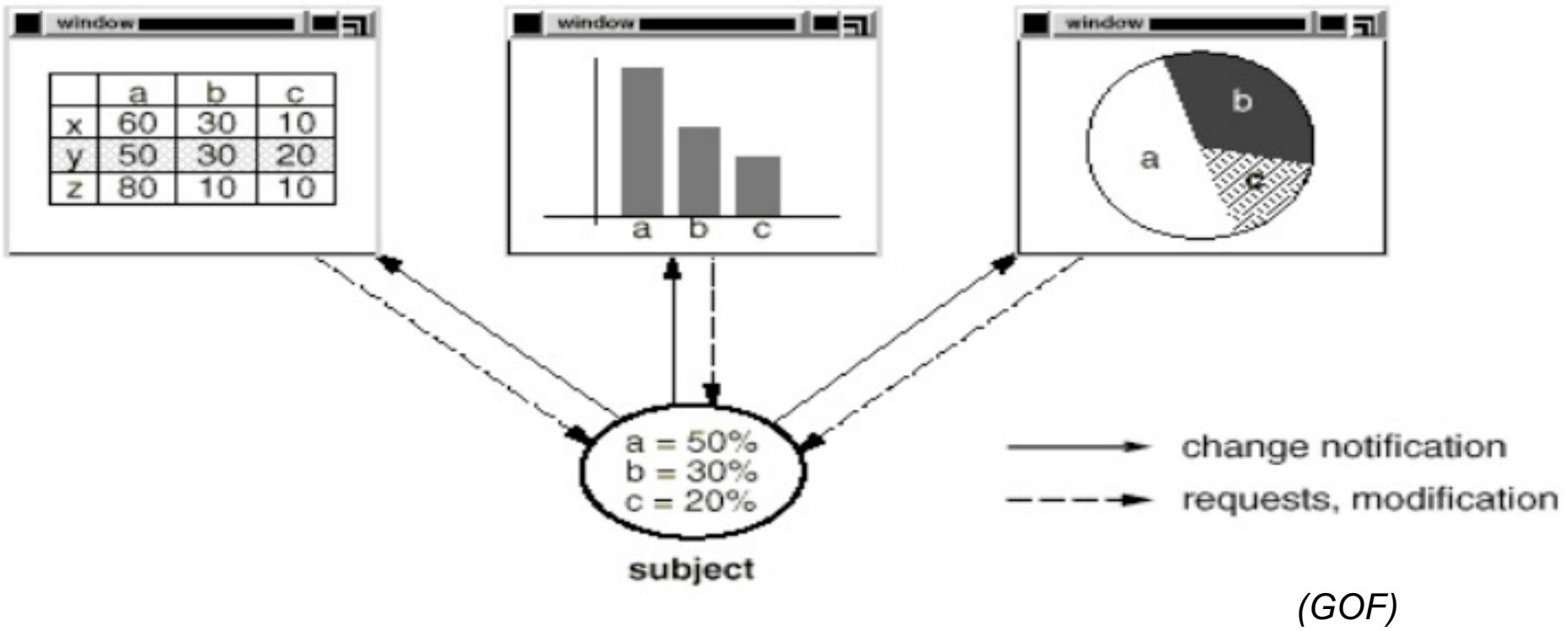
Description et classification du GOF

Description

- Description complète au « standard » GOF
- Nom du DP (+ aliases)
  - important pour parler DP
  - ex: Observateur
  - aliases: Dépendants, Diffusion-Souscription
- Intention
  - objectif, courte description de ce que fait le pattern
  - ex: définit une interdépendance entre un objet et plusieurs autres de façon telle que quand l'objet change d'état, les autres soient notifiés et mis à jour

Description

- Motivation
  - un scénario ou un exemple montrant l'intérêt du pattern



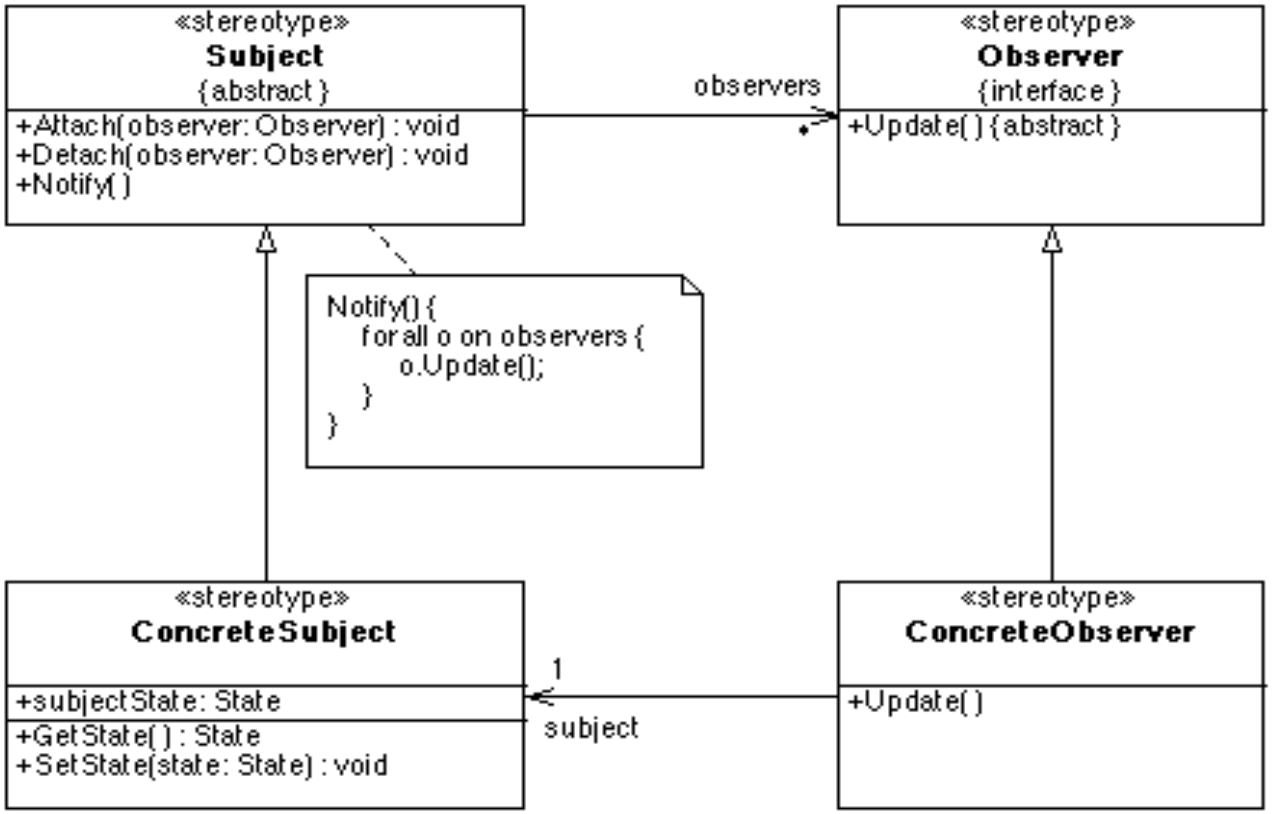
(GOF)

Description

- Indications d'utilisation
  - dans quels cas l'utiliser?
  - conditions à vérifier pour en tirer partie
  - ex.
    - quand un objet a 2 représentations inter-dépendantes mais pouvant être utilisées et évoluer séparément
    - quand la modification d'un objet entraine la notification d'autres dont il ne connait pas le nombre ni la nature (couplage faible)

Description

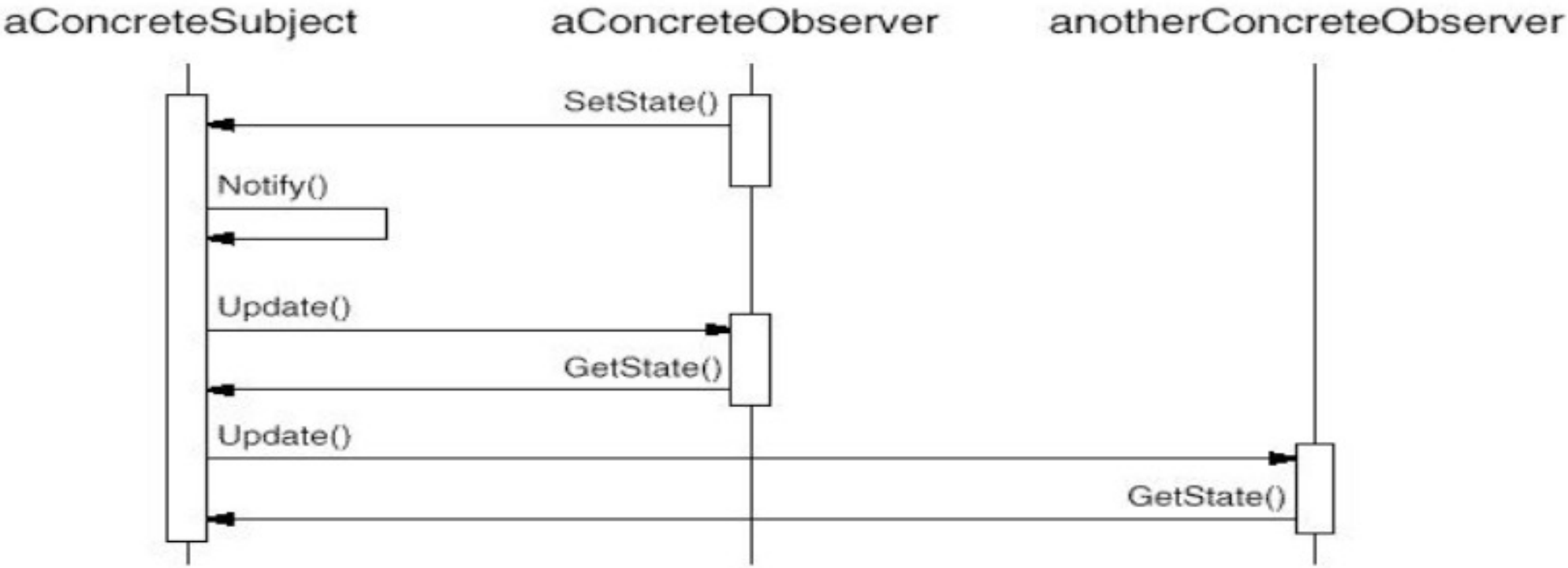
- Solution: structure
  - participants, rôles et responsabilités



(GOF)

Description

- Solution : collaborations
  - comment les participants interagissent (souvent diagramme de séquences)



(GOF)

Description

- Conséquences
  - qualités, avantages, inconvénients
  - effets possibles, positifs ou négatifs, mises en garde
  - ex:

    + couplage faible

    - performances (modifications inopinées et/ou coûteuses dues à la minimalité du protocole de notification)

Description

- Implémentation
  - consignes d'implémentation possibles
  - techniques à employer
  - difficultés, pièges à éviter
  - ex:
    - table sujets/observateurs
    - attention à la disparition d'observateurs
    - alternatives push/pull entre sujets et observateurs, avantages et inconvénients

Description

- Exemples de code
  - fragments de code typiques
  - dans des langages différents
- Utilisations remarquables
  - dans des systèmes ou situations réels
  - ex:
    - IHM : découplage objet (sujet) et ses vues (observateurs)
    - MVC de Smalltalk, Java Swing, Framework Struts
- Patterns apparentés
  - relations entretenues avec d'autres patterns, comparaison
  - ex:
    - *Médiateur* entre sujet et observateurs
    - *Commande* pour créer et gérer dynamiquement des observateurs

Classification des DP

- faciliter l'identification, le repérage
- « groupes de patterns » concernant la même catégorie de problèmes
- efficacité des classifications
  - contribue à la précision, concision du langage métier...
- démarche « scientifique » et systématique
- précision de l'espace des solutions de conception
  - comparer les patterns entre eux, partitionner
  - éviter les redondances (sinon quelle utilité pour le concepteur !?)
  - repérer les vraies solutions récurrentes des variantes

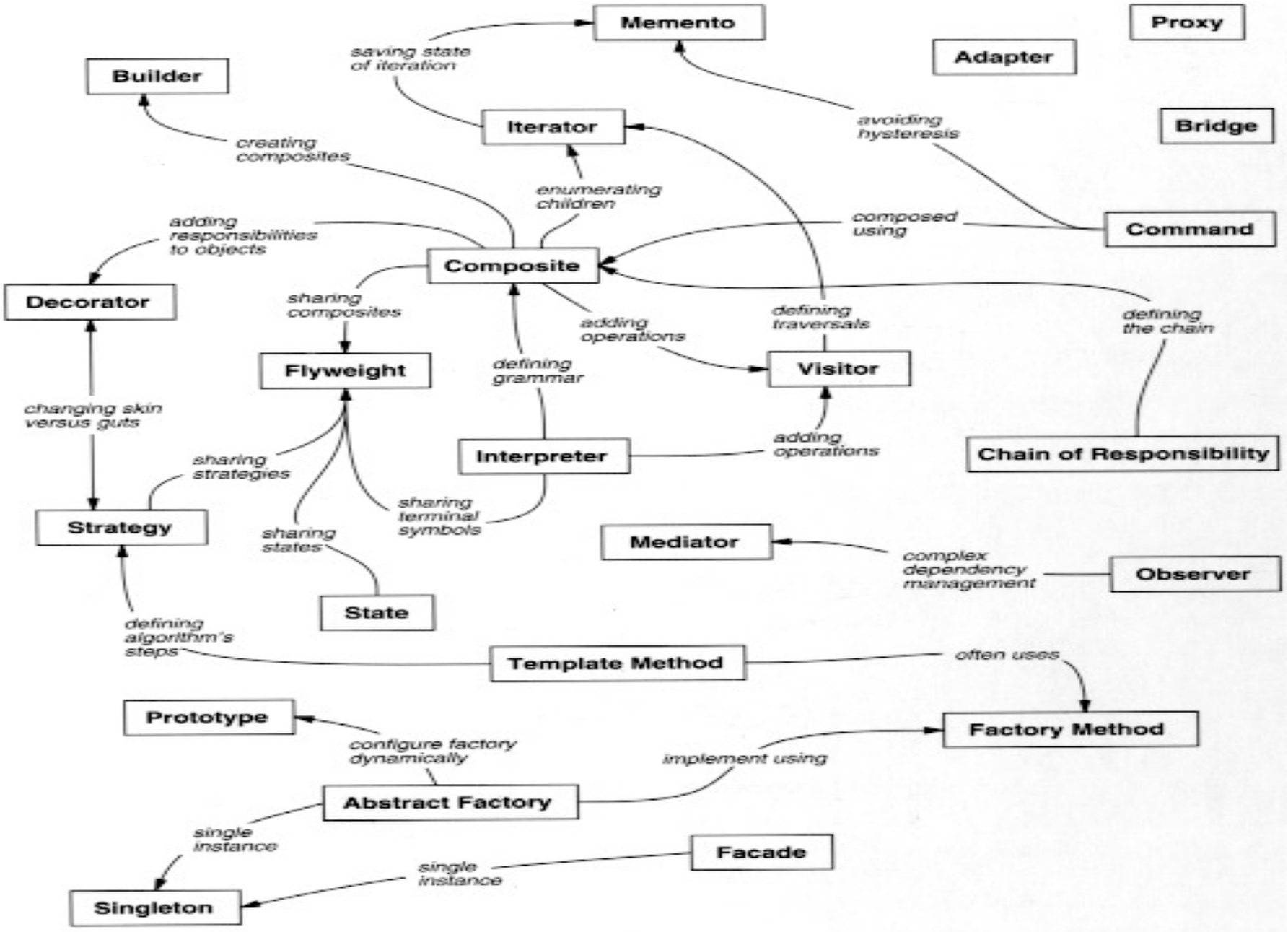
Classification des DP

- Classification du GOF

- Créateurs
  - abstraient et encapsulent les processus de création d'objets
- Structurels
  - composition de classes ou d'objets pour former des structures plus vastes
- Comportementaux
  - interaction (interconnexion entre objets et communication) répartition des responsabilités

Classification des DP

Catégorie GOF	*DP de niveau classes plutôt orientés héritage*	*DP de niveau objets plutôt orientés composition*
Créateurs	Fabrique	Monteur, Prototype, Singleton
Structurels	Adaptateur	Composite, Décorateur, Façade, Pont, Proxy, Poids mouche
Comportementaux	Interprète, Patron de méthode	Itérateur, Observateur, Etat, Stratégie, Visiteur, Chaîne de Responsabilités, Médiateur, Commande, Mémento



Design Pattern Relationships

Classification parfois déroutante...

- un pattern semble souvent appartenir à plusieurs catégories
- mais subtilité dans l'intention des auteurs (ne pas se tromper de problème...)
- exemple
  - Décorateur : structurel
  - Proxy : comportemental
  - pourtant semblables (concernent le comportement)
  - justification:
    - Décorateur = composition par wrapping pour ajouter du protocole à un objet
    - Proxy = interaction entre objets, basée sur leur protocole

Design Pattern

Les patterns fondamentaux du GOF

Patterns Fabriques (Créateurs)

- Fabrique
  - Pattern créateur de niveau classe
  - création d'instance répondant à une hiérarchie de classes
- Fabriques abstraites
  - Pattern créateur de niveau objet
  - délègue la création d'objets ou de groupes d'objets corrélés à un objet créateur

Patterns Fabriques

- Problème commun
  - tous les langages offrent des primitives d'instanciation de classes (« new »)
  - problème : l'opérateur new fait une référence explicite à la classe
    - faciliter l'ajout de nouveaux types (classes) est l'une des principales promesses de la conception OO
    - il suffit de prévoir une sur-classe (abstraite ou interface) commune et de l'étendre
    - idem dans les frameworks extensibles
      - basés sur un ensemble d'interfaces à étendre
    - ce principe tient ses promesses quant à l'utilisation « générique » de ces types...
    - ... mais les problèmes arrivent dès qu'il s'agit de les instancier
      - ajouter des références (statiques) aux nouvelles classes « partout »
      - peu traçable, peu évolutif, inefficace en recompilation

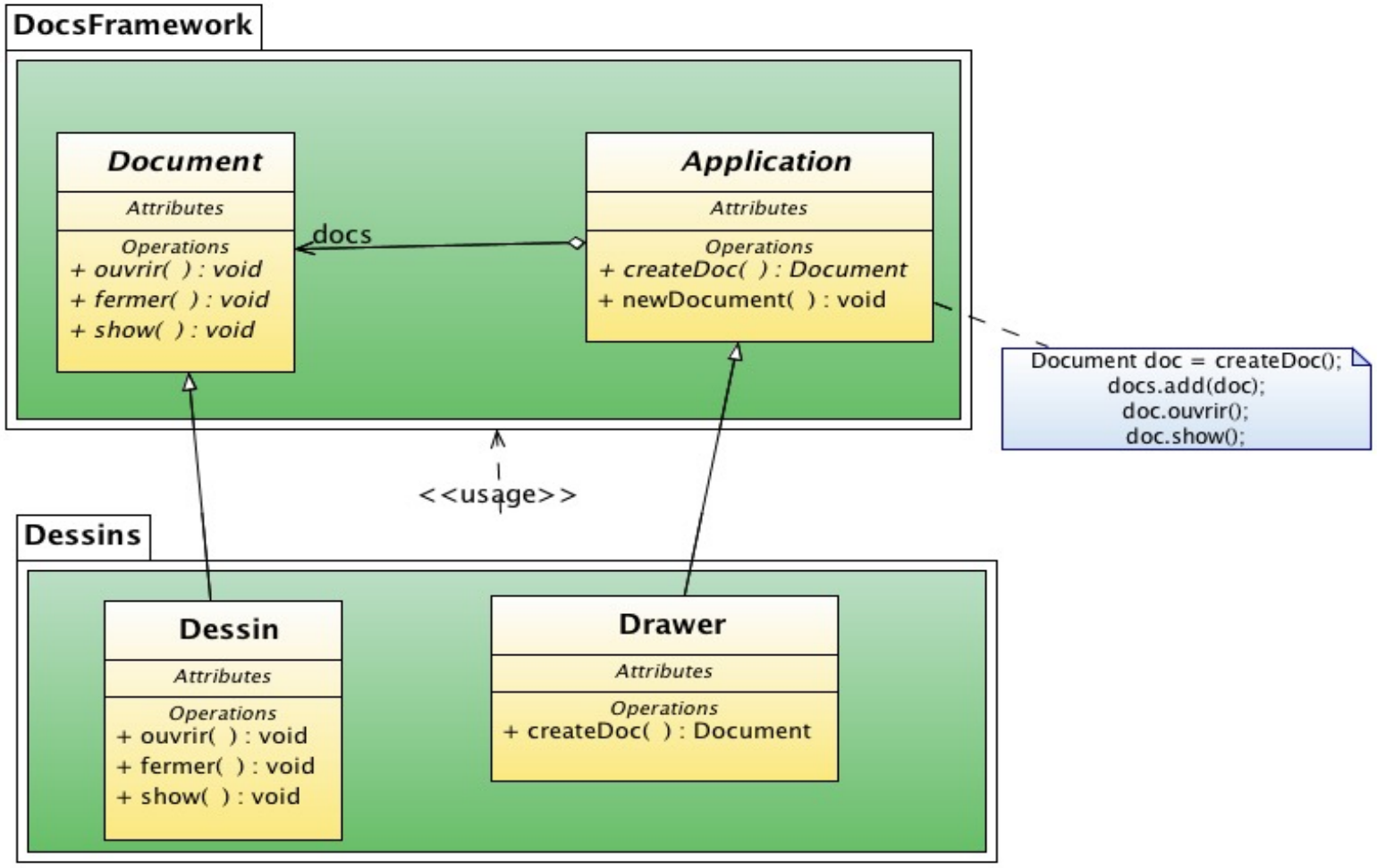
Patterns Fabriques

- Problème commun
  - les patterns Fabrique offrent des solutions pour faciliter la création d'objets (ou de groupes d'objets)
    - sans connaître la classe
    - centralisant la création d'objets (un seul lieu de modification)
    - encapsulant la création d'objets...
      - « calcul » de classe à instancier
- Favoriser le couplage faible
  - découplage code-classes présentes et futures
  - faciliter l'ajout de nouvelles (sous-)classes sans modifier le code

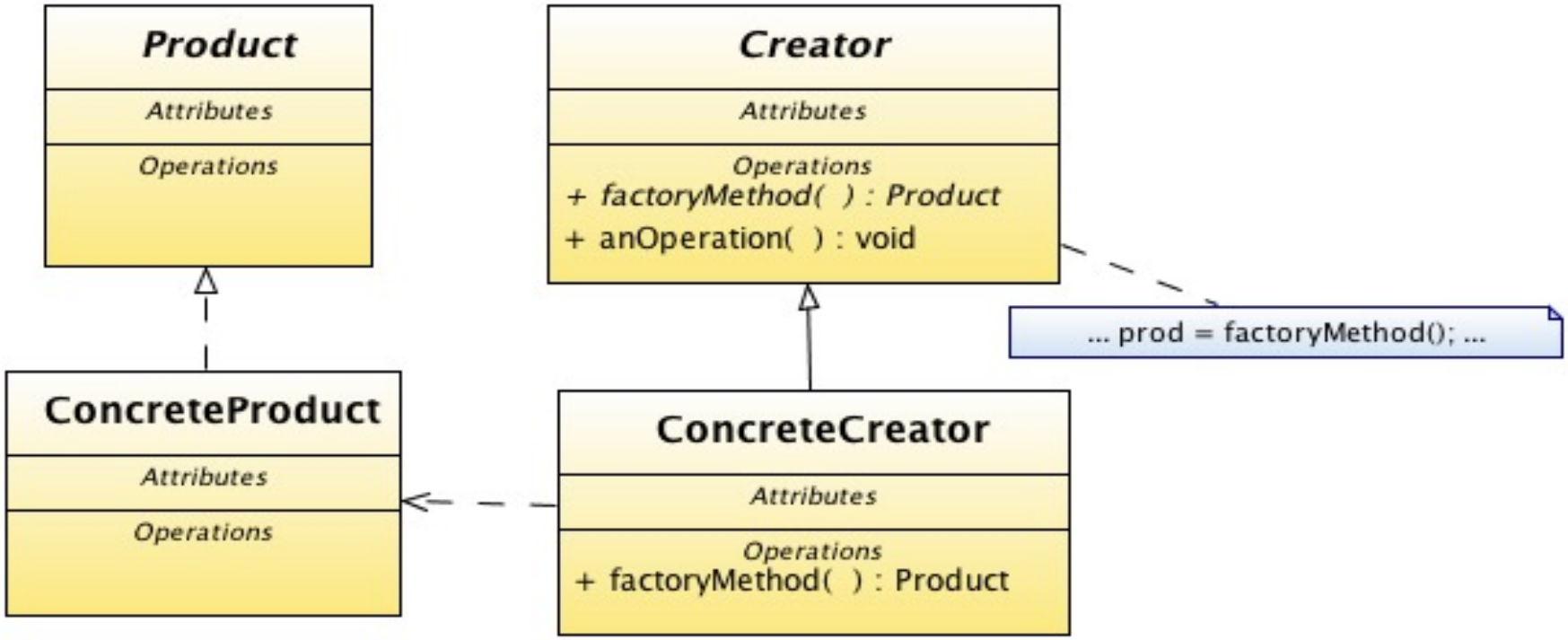
Pattern Fabrique

- aliases
  - Factory Method
  - Usine, Fabrication, Constructeur Virtuel
- intention
  - définit une interface (protocole redéfinissable) pour la création d'objet en laissant aux sous-classes le choix de la classe à instancier.
- indications d'utilisation
  - quand une classe ne peut anticiper la classe des objets qu'elle aura à créer
  - quand une classe attend de ses sous-classes qu'elles déterminent les objets qu'elles créent

Pattern Fabrique : Motivation

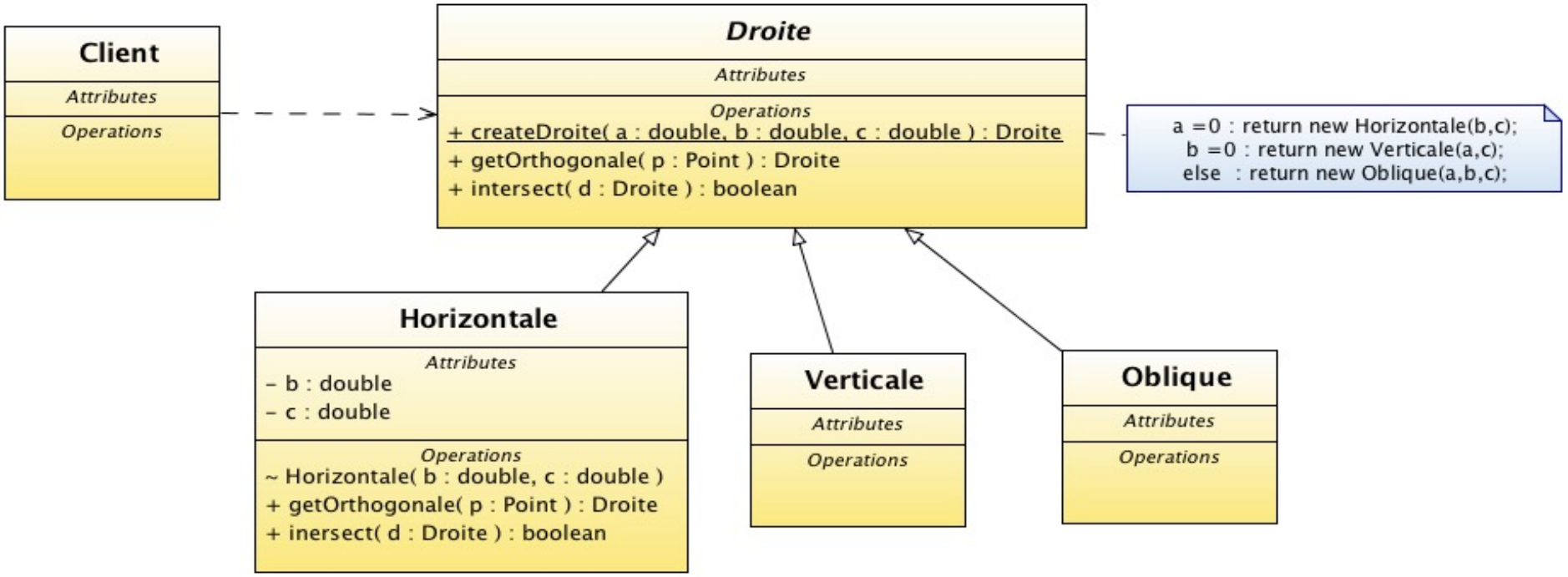


Pattern Fabrique : Structure



Pattern Fabrique

- Utilisation: « calcul » de classe
  - le client ne connaît que le type et non les classes d'implémentation



Pattern Fabrique

- Utilisations remarquables
  - conteneurs (FW) technologiques avec leurs services techniques
  - objets distribués (RMI), J2E, Corba, .Net
  - le client ne connait que le type (interface) des composants (Produits) et leurs fabriques

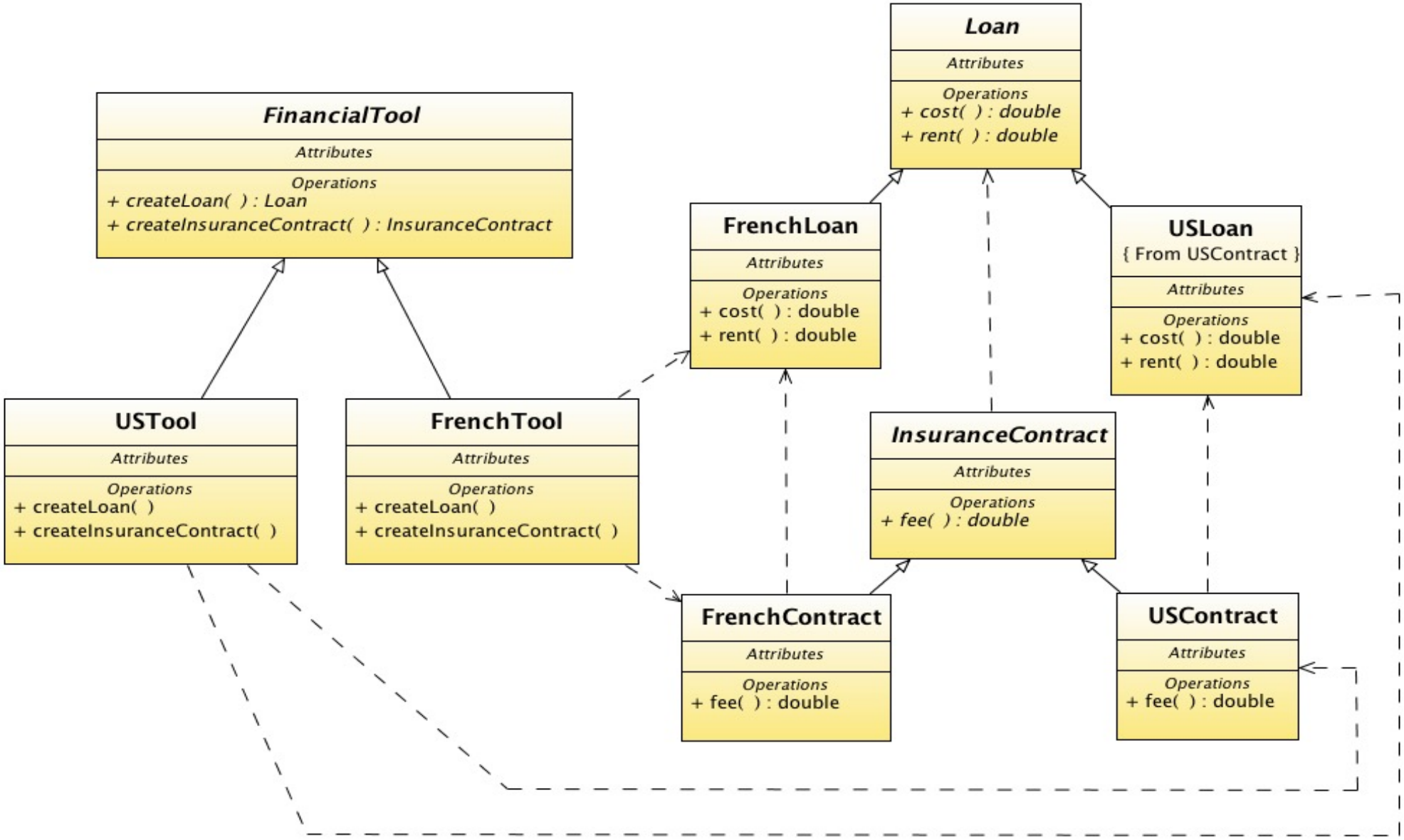
```
import javax.naming.*;
public class Client { //EJB
// acces au contexte de nommage JNDI
  Context initialCtx = new InitialContext ();
  // lookup de l'interface EJBHome (fabrique)
  AccountHome home=(AccountHome)initialCtx.lookup("Account");
  // pour "creer" un objet Session (type Produit)
  Account account = home.create (5082, "Duduche", 545.70);
  // utiliser les operations (du type)
  account.crediter (300.00);
}}
```

Pattern Fabrique

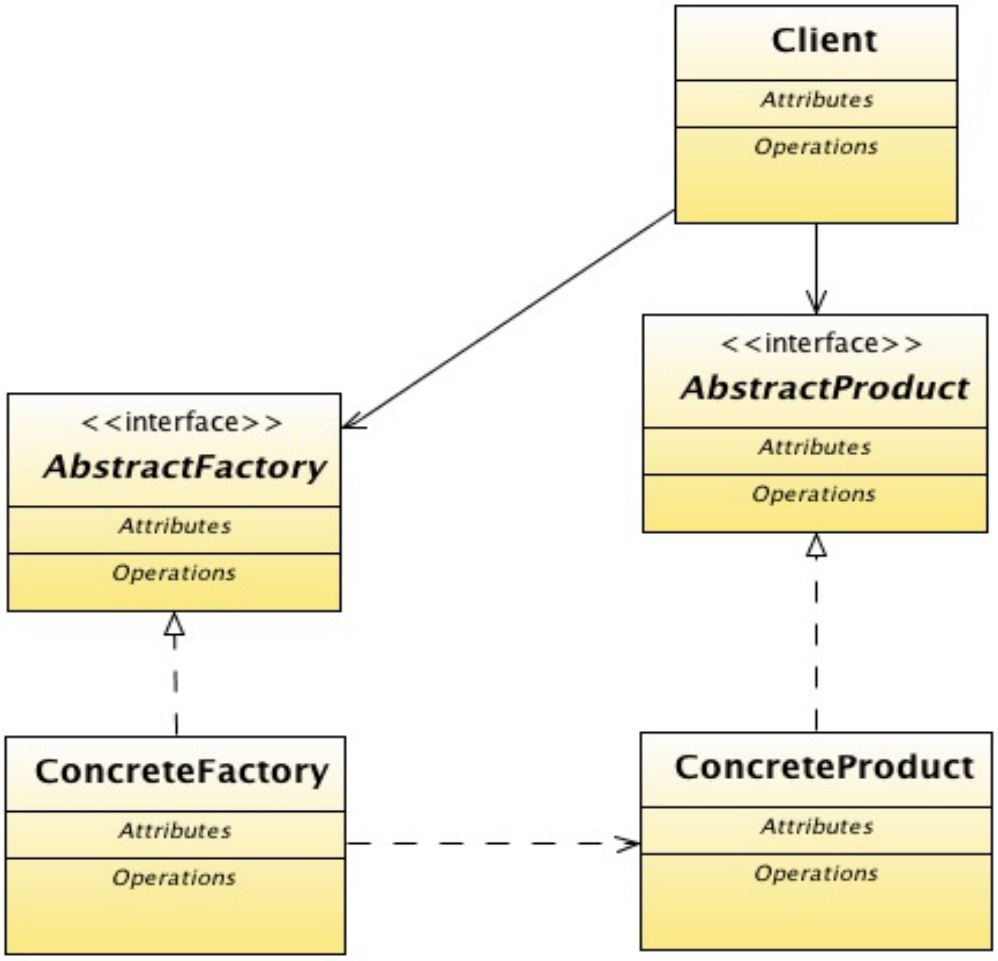
- Patterns apparentés
  - Fabrique abstraite
  - Patron de méthode (comportemental): méthode de fabrication

Pattern Fabrique Abstraite (Créateur)

- Alias
  - Abstract Factory
  - Kit de fabrication
- Intention
  - fournit une interface pour créer des familles d'objets corrélés ou dépendants sans avoir à spécifier leurs classes concrètes.
- Motivation...


Loan
Attributes
Operations
+ cost( ) : double
+ rent( ) : double
FinancialTool
Attributes
Operations
+ createLoan( ) : Loan
+ createInsuranceContract( ) : InsuranceContract
FrenchLoan
Attributes
Operations
+ cost( ) : double
+ rent( ) : double
USLoan
{ From USContract }
Attributes
Operations
+ cost( ) : double
+ rent( ) : double
USTool
Attributes
Operations
+ createLoan( )
+ createInsuranceContract( )
FrenchTool
Attributes
Operations
+ createLoan( )
+ createInsuranceContract( )
InsuranceContract
Attributes
Operations
+ fee( ) : double
FrenchContract
Attributes
Operations
+ fee( ) : double
USContract
Attributes
Operations
+ fee( ) : double

Pattern Fabrique Abstraite: Structure

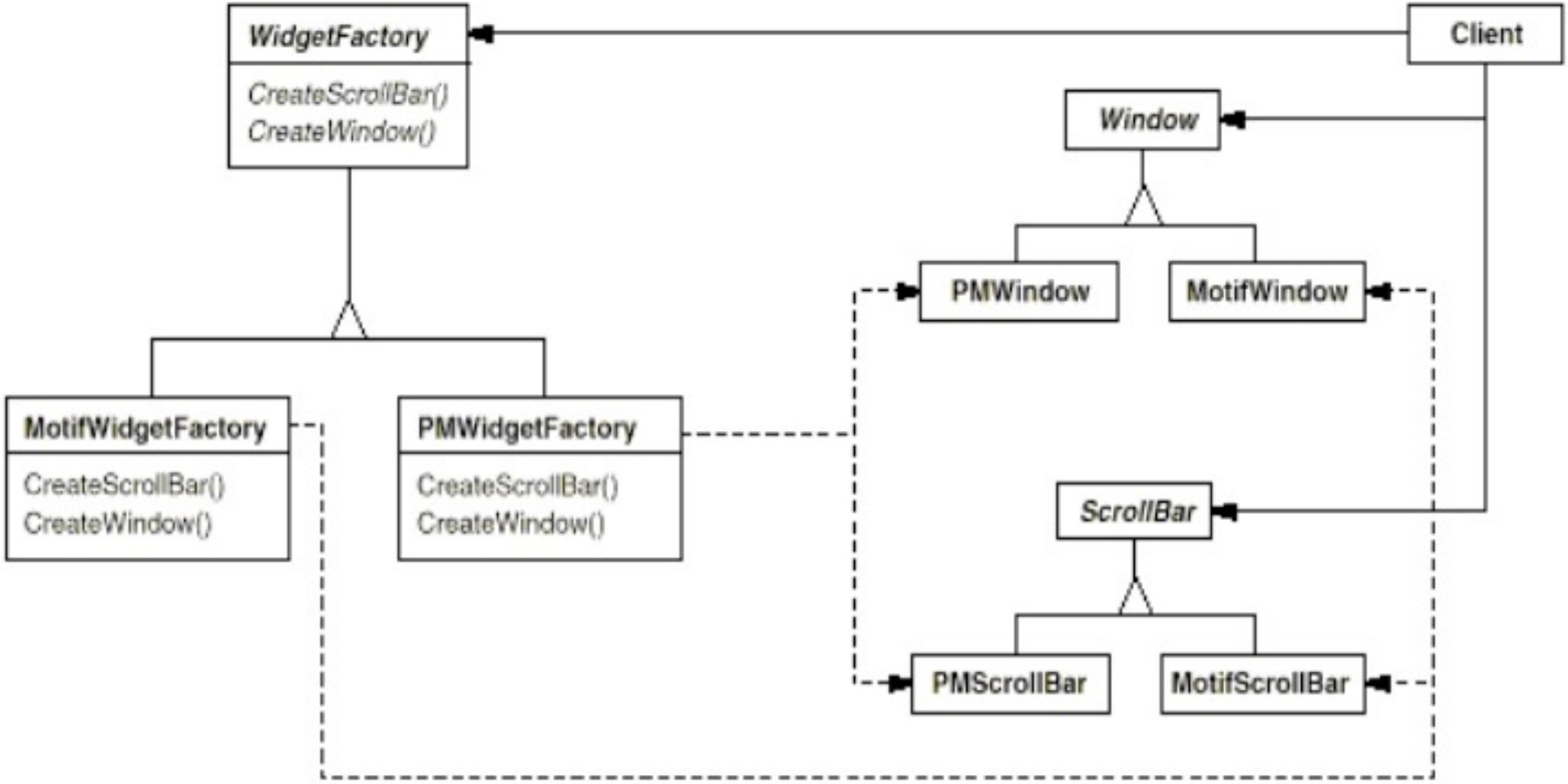


Pattern Fabrique Abstraite

- Contextualisation d'applications complexes
  - internationalisation
  - Contextes fonctionnels sur des mêmes produits (objets)
    - CAO, progiciels (production, gestion, ventes, décisionnels)
- Window Toolkits ou applications graphiques portables
  - les widgets ne doivent pas être codés en dur
  - indépendants du look-and-feel, de la plate-forme (peer mapping)
    - exemple : AWT Java

```
public class Button extends Component {
   public Button() {
     peer = getToolkit().createButton(this);
...}}
```

Fabrique Abstraite (ex. du GOF)



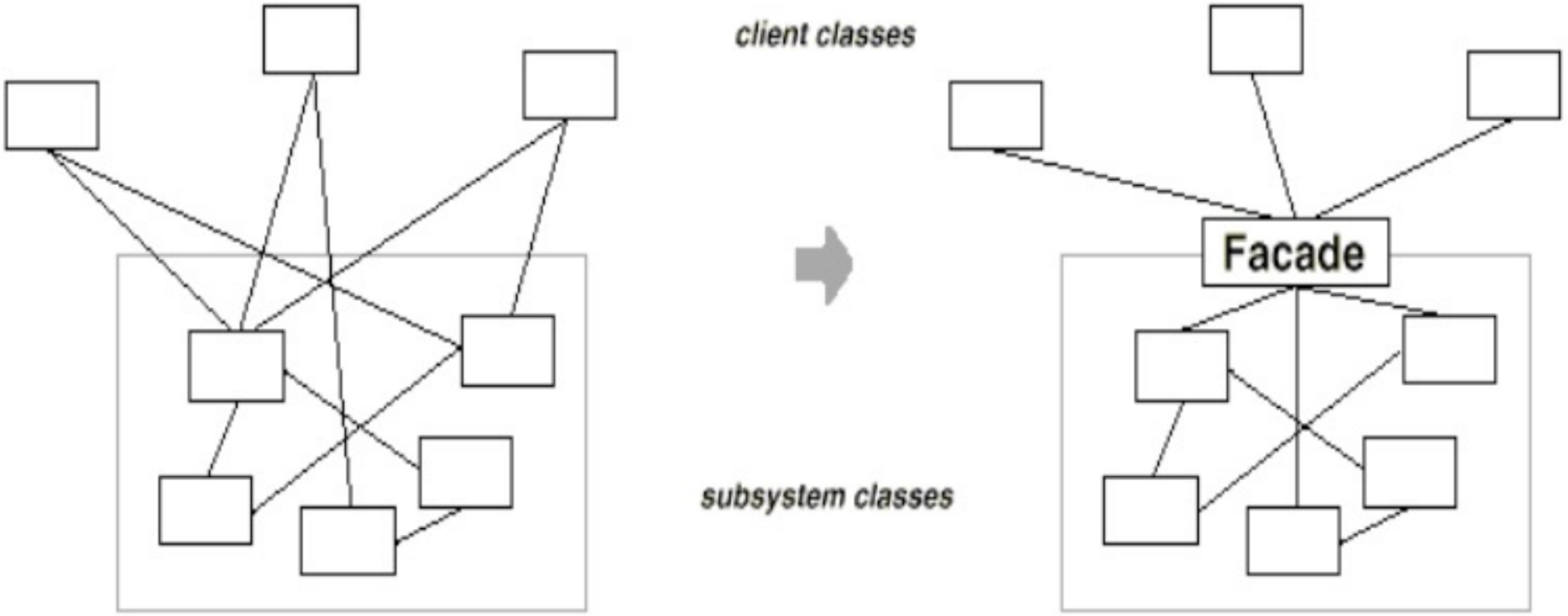
Pattern Fabrique Abstraite

- Patterns apparentés
  - souvent implémentés en utilisant des méthodes de fabrication (pattern Fabrique)
  - Façade : interface unifiée de manipulation d'un système complexe
  - une fabrique abstraite est souvent un Singleton
  - pattern Pont (Bridge)
    - séparation couche d'objets d'abstraction / couche d'objets d'implantation (« physique »)

Pattern Façade (structurel)

- Intention
  - Fournit une interface unifiée à l'ensemble des interfaces des sous-systèmes. La façade fournit une interface de plus haut niveau, qui rend le système plus facile à utiliser.
- Utilisation
  - Souvent associé à Fabrique Abstraite (si ce n'est confondue avec) pour offrir une interface de création des objets des sous-systèmes (sans les exposer).
  - Application typique : composants métiers (Sessions J2EE)

Pattern Façade (structurel)

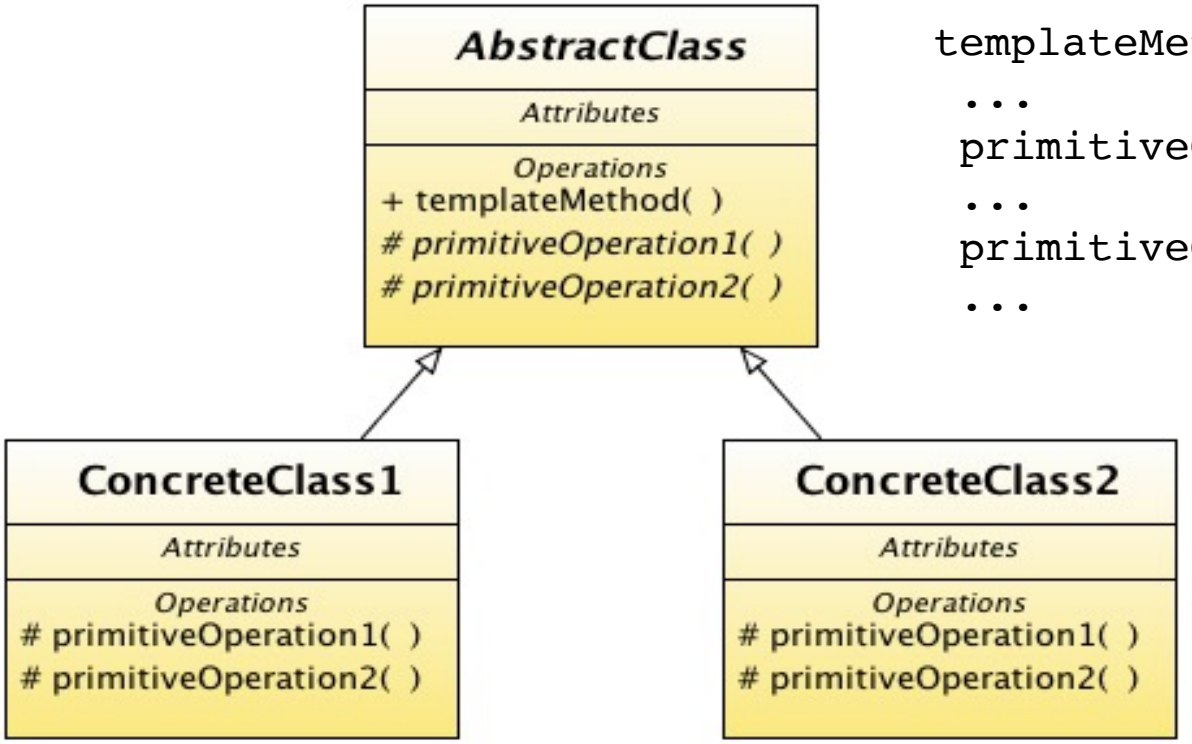


Pattern de méthode (comportemental)

- Alias
  - Method Template
- Intention
  - Définit le squelette d'un algorithme en en déléguant certaines parties aux sous-classes
- Utilisation
  - essentielle et classique en bonne conception OO ;)
  - réutilisation et conception de frameworks: *contrôler l'extensibilité*
  - Inversion des dépendances (IOC)

    "the Hollywood principle: Don't call us, we'll call you »

Pattern de méthode

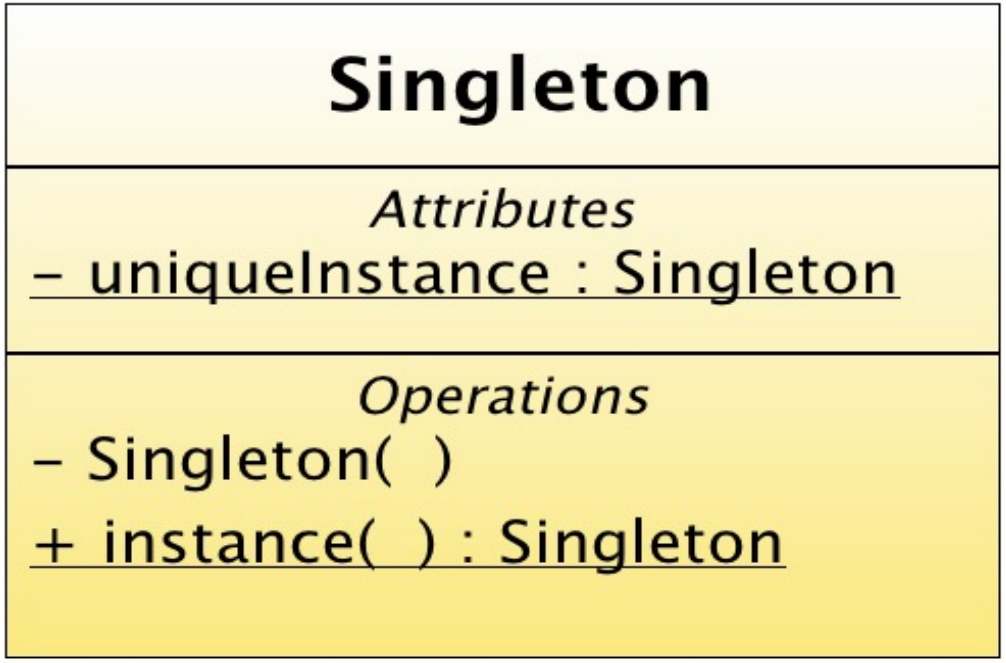



```
templateMethod() {
 ...
 primitiveOperation1()
 ...
 primitiveOperation2()
 ...
```

Pattern Singleton (Créateur)

- Intention
  - garantit qu'une classe n'a qu'une seule instance et en fournit un point d'accès global
- Problème
  - on ne veut qu'une seule instance d'une classe pour tout le système
  - ex.
    - un seul Window Manager, un seul Ressources Manager, un seul File Manager
    - une seule fabrique de produits
    - un seul registre de lookup (ex. JNDI, RMI Registry)
  - assurer que d'autres instances ne peuvent être créées

Pattern Singleton



Pattern Singleton

- Evaluation
  - noter le rôle important de la classe comme meilleur lieu pour contrôler son instanciation et surtout prévenir de la création d'autres instances
  - masque l'instanciation et la réserve à la classe pour créer sa seule instance
  - à comparer avec des ressources globales initialisées (variables globales)
  - création « par nécessité » lazy initialisation
    - coût :) surtout si gourmand en ressources
  - par rapport à: 1 classe avec tout en statique « sans instance »
    - pas de lazy initialisation (si coûteux en ressources à créer)
    - moins OO
    - non extensible
      - Singleton est une « vraie classe »
  - attention à l'abus de « single instance classes » (OO?)

Pattern Singleton

- Précautions
  - si multi-threadé:
    - synchroniser getInstance() (mais chère pour une seule fois)

```
class Singleton {
 public static synchronized Singleton getInstance()
   ...
}
```

    - ou initialisation statique (au chargement)

```
class Singleton {
 private static instance= new Singleton();
 public static Singleton getInstance() {
   return instance;
 }
}
```

Pattern Singleton

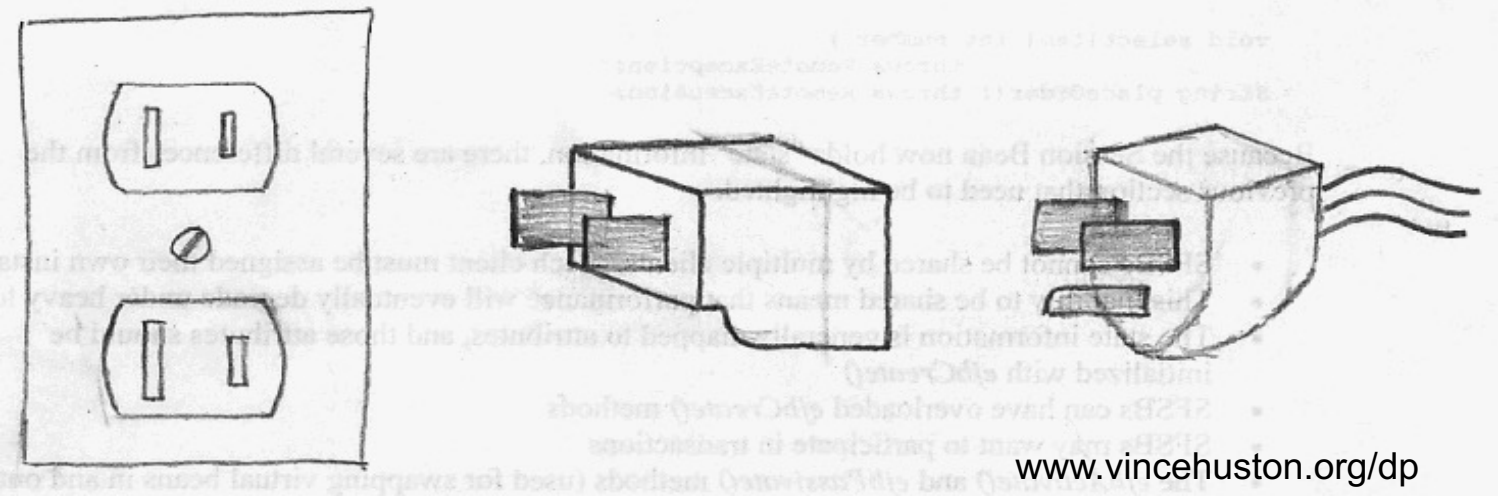
- Patterns apparentés
  - Fabrique Abstraite
  - Monteur
  - Prototype

Pattern Adaptateur (Structurel)

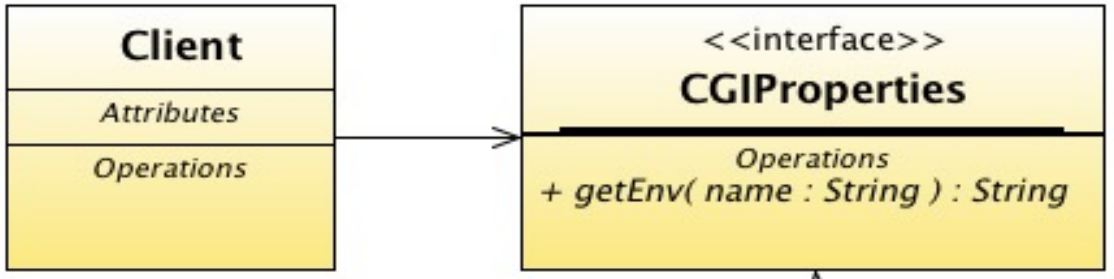
- Alias
  - Adapter
- Intention
  - Convertit l'interface d'une classe en une autre conforme à l'attente d'un client.
  - Permet à des classes de collaborer malgré l'incompatibilité de leurs interfaces.
- Motivation...

Pattern Adaptateur

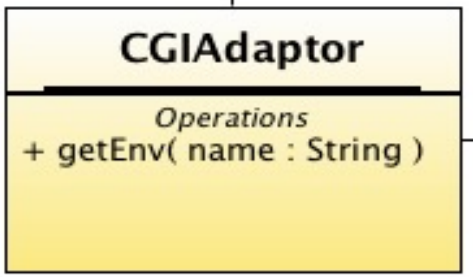
- Motivation

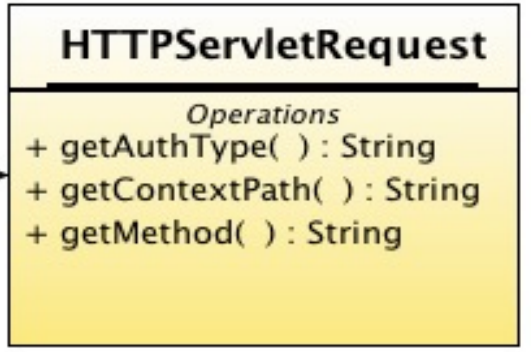


Pattern Adaptateur: motivation

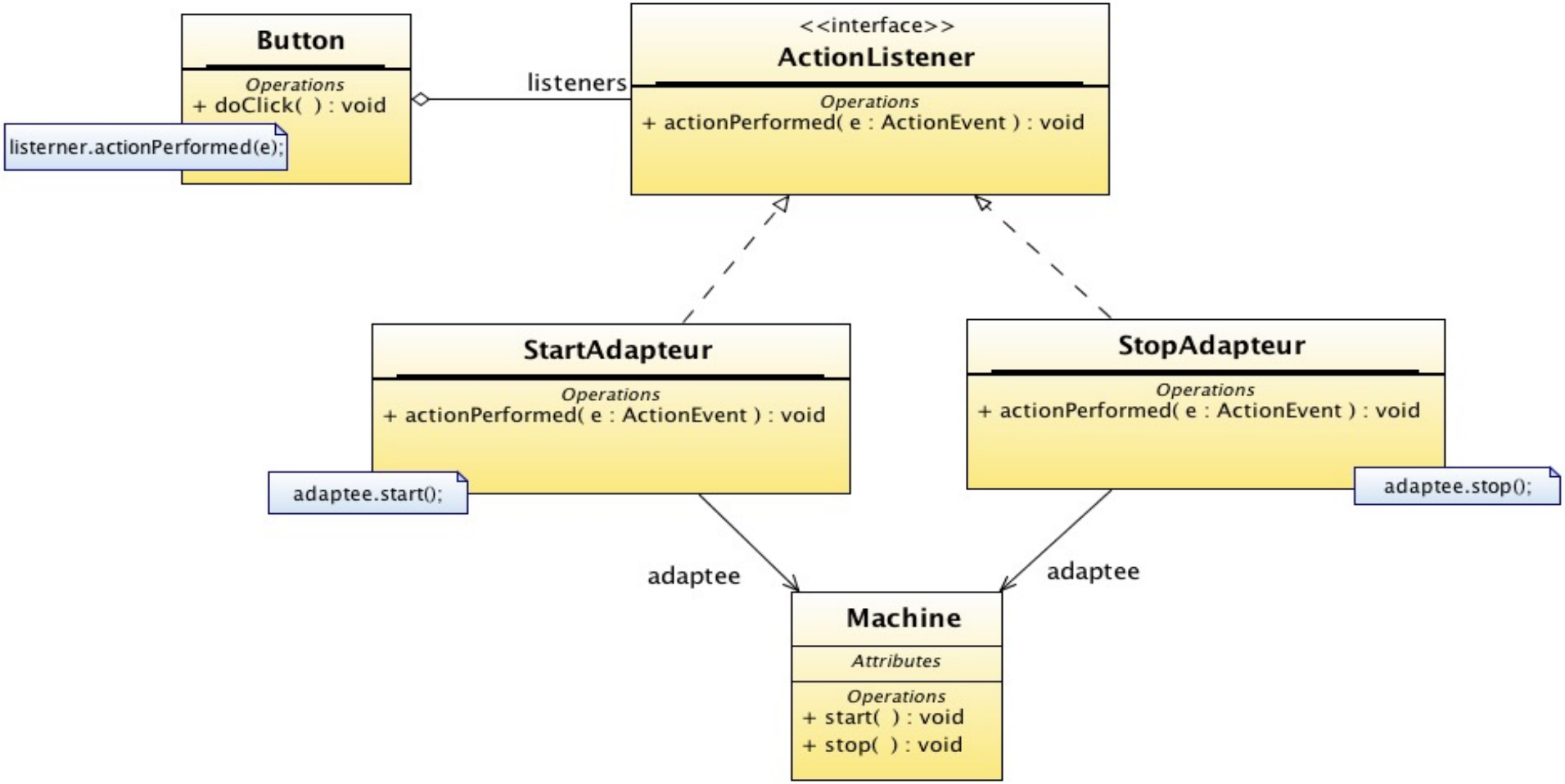


AUTH_TYPE	getAuthType()
CONTEXT_PATH	getContextPath()
REQUEST_METHOD	getMethod()
SCRIPT_NAME	getServletPath()
SERVER_NAME	getServerName()
...	

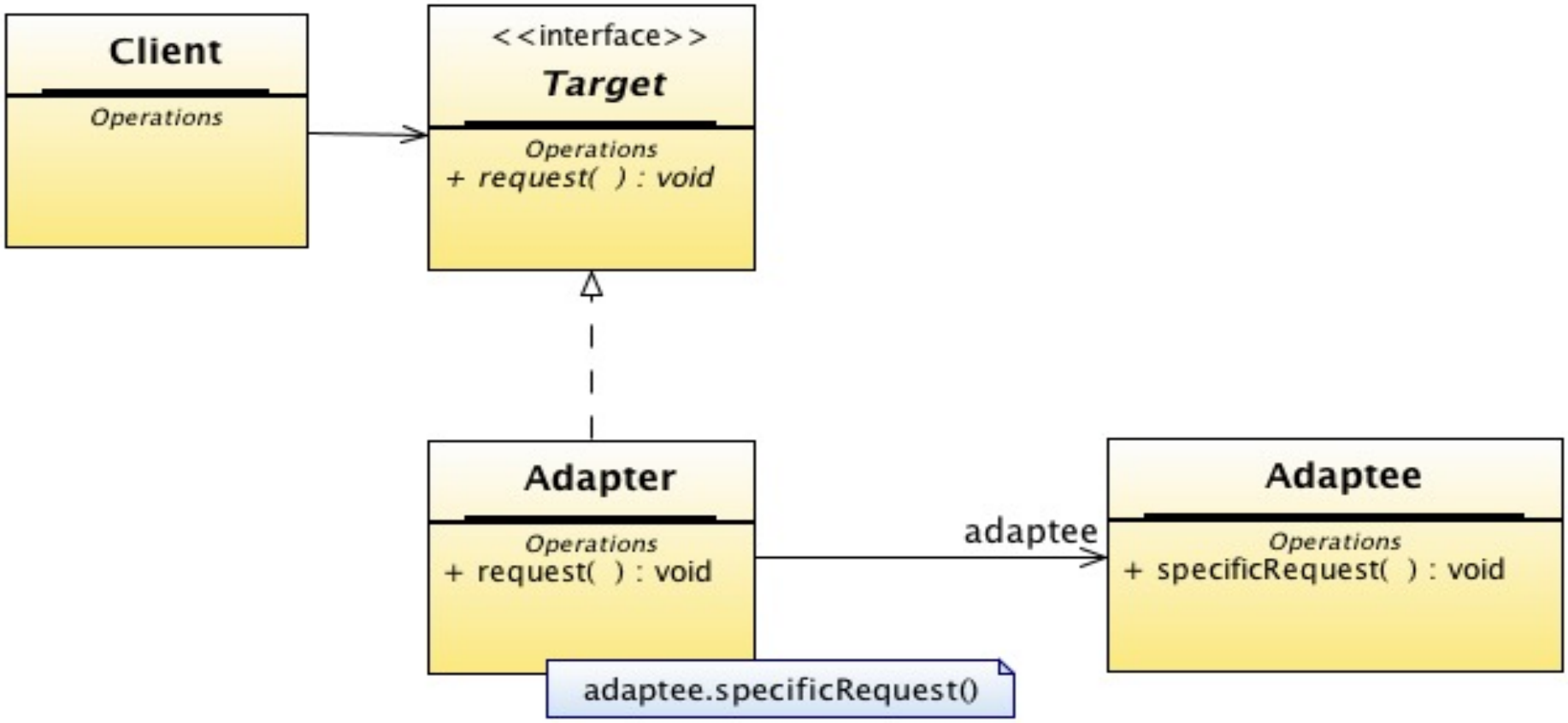




Pattern Adaptateur



Pattern Adaptateur: structure



Pattern Adaptateur

- Indications d'utilisation
  - quand on veut utiliser une classe existante mais dont l'interface ne coincide pas avec celle attendue
    - nom des opérations
    - nombre de paramètres, types (transformation de données)
  - découplage : création de classes réutilisables qui doivent collaborer - interface attendue - avec des classes encore inconnues et qui n'auront pas nécessairement cette interface.
    - programmation par composants avec interfaces requises
- Utilisations remarquables
  - la plupart des bibliothèques d'IHM
  - programmation par composants JavaBeans

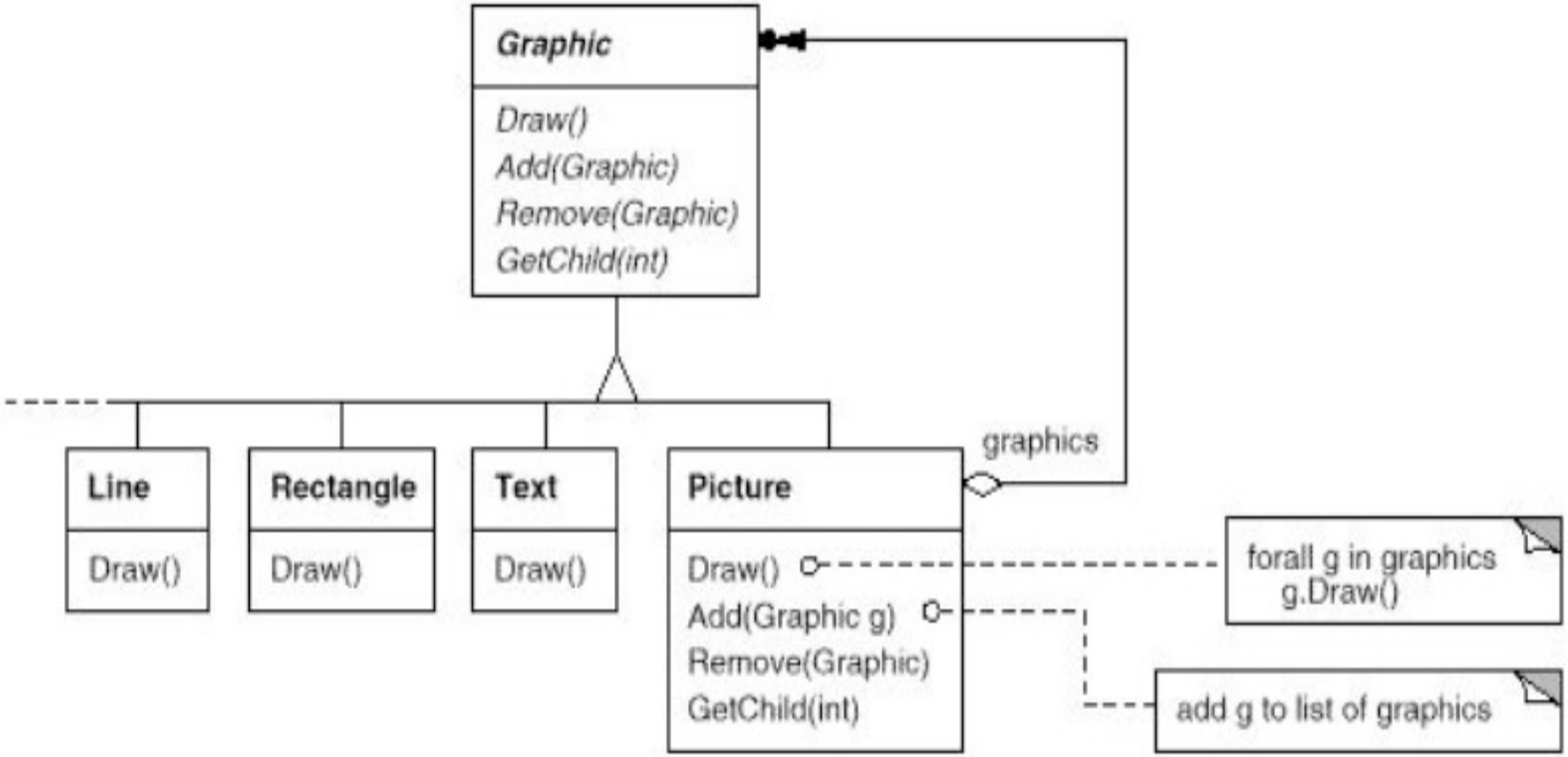
Pattern Adaptateur

- Patterns apparentés
  - Adaptateur **convertit** un protocole dans un autre alors que Decorateur **ajoute** du protocole
  - Bridge est dédié à la séparation entre **protocole et implémentation** et s'utilise en phase de **conception a priori**, alors que Adaptateur s'applique à l'**assemblage** de classes existantes pour les connecter **a posteriori**.
  - Commande, de niveau objet, concerne **l'exécution et son contrôle** : différée, historisée, redirigée, do/undo/redo alors que Adaptateur est de niveau classe et n'est pas concerné pas l'état de l'exécution

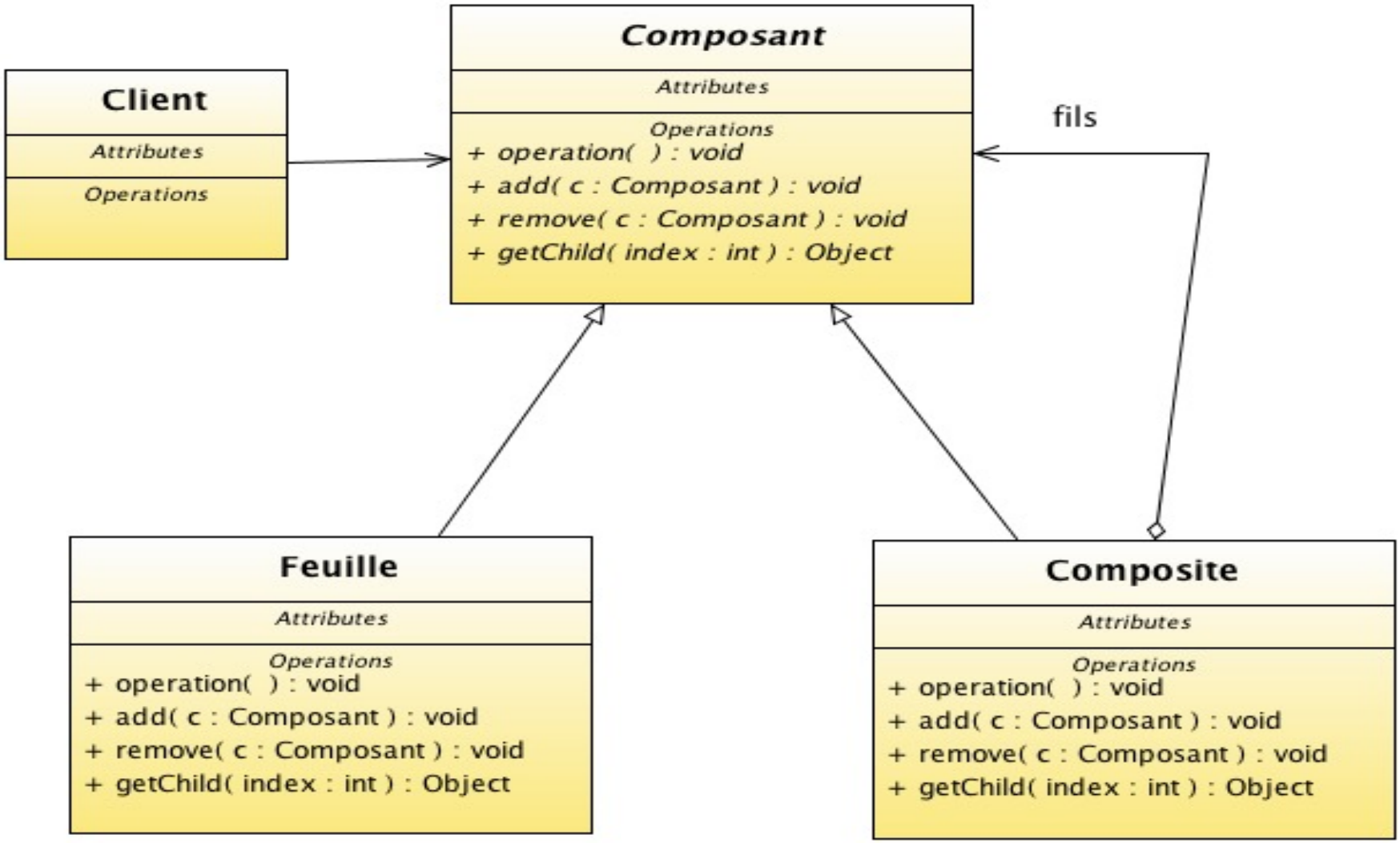
Pattern Composite (structurel)

- Aliases
  - hiérarchie partie-tout, part-whole hierarchy
- Intention
  - Compose des objets en structures arborescentes pour représenter des hiérarchies récursives composant/composé.
  - Il permet au client de traiter de la même façon les objets individuels et les combinaisons de ceux-ci.

Pattern Composite: motivation (GOF)



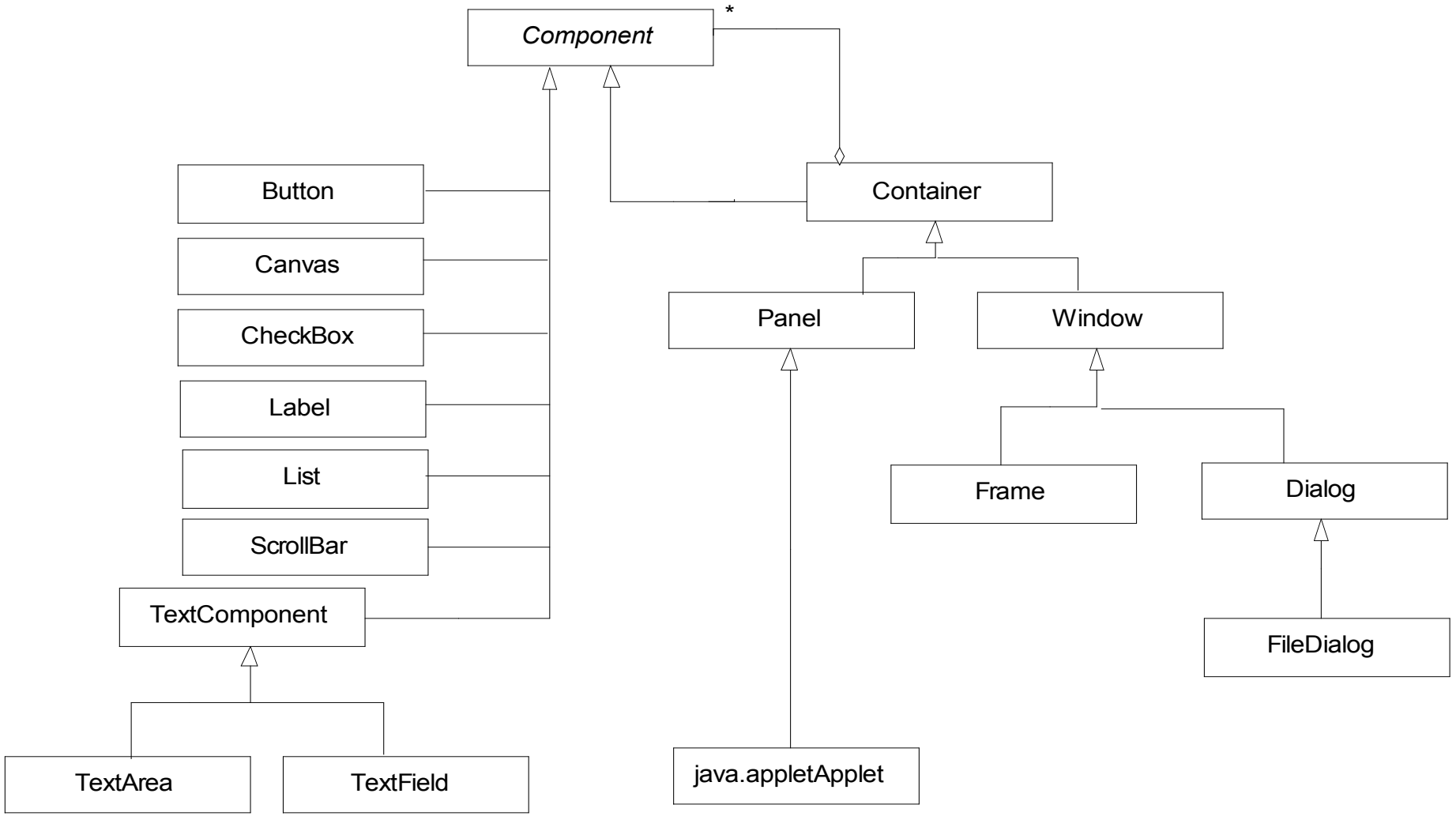
Pattern Composite : Structure



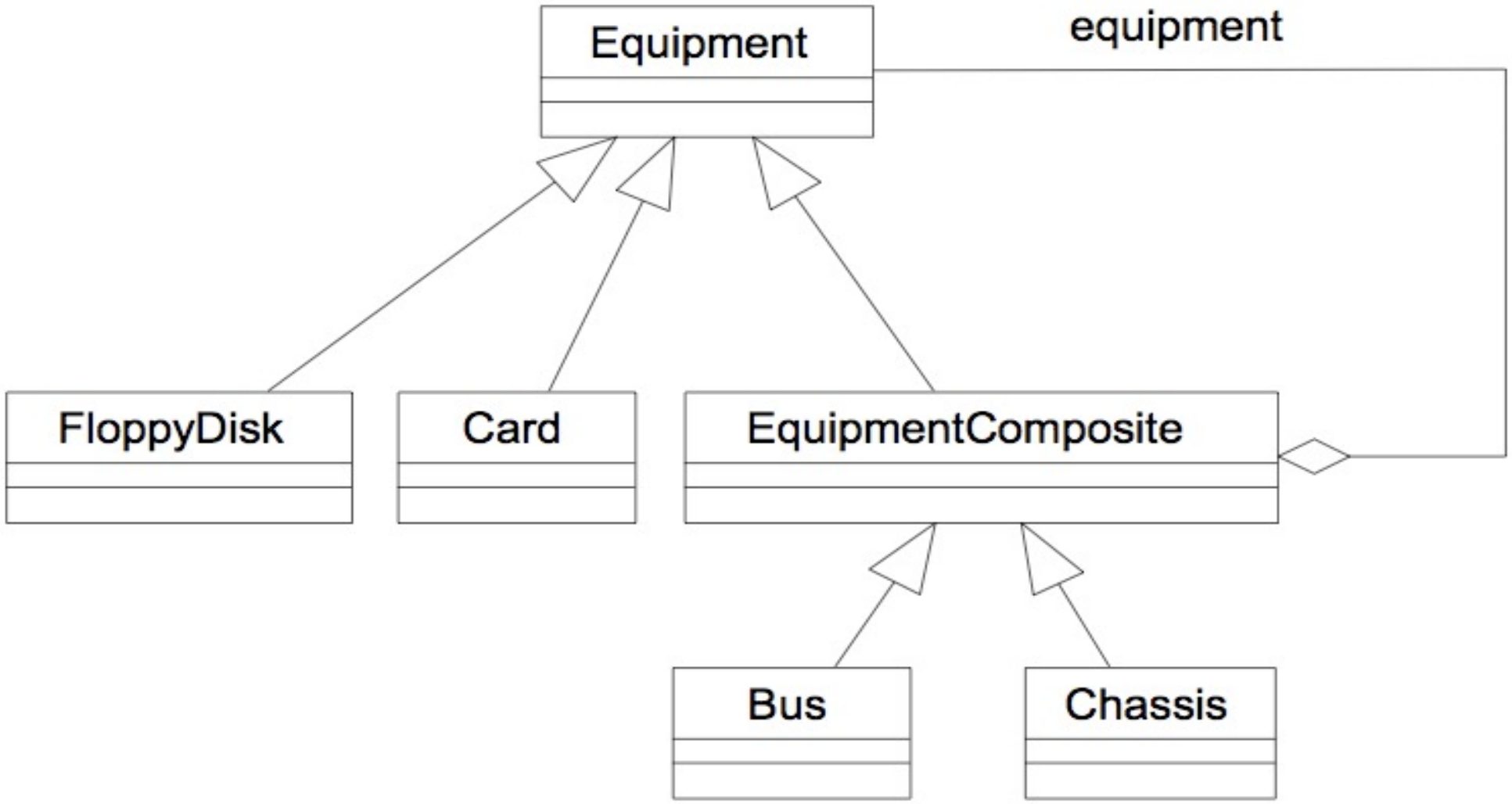
Pattern Composite

- Conséquences
  - facilite d'ajout de nouveaux types de composants
  - simplifie les applications clientes qui n'ont pas à se soucier du statut du composant (feuille ou composite)
    - pour cela il est conseillé de maximiser l'interface de Composant, même si certaines opérations ne feront rien dans Feuille
      - dont les opérations de gestion (add/remove)
    - mais compromis:
      - transparence
        - alourdit l'exécution
      - typage
        - ranger les opérations « composites » dans Composite
        - vérification statique

Utilisation: Window Toolkits



Utilisation: « monde réel »



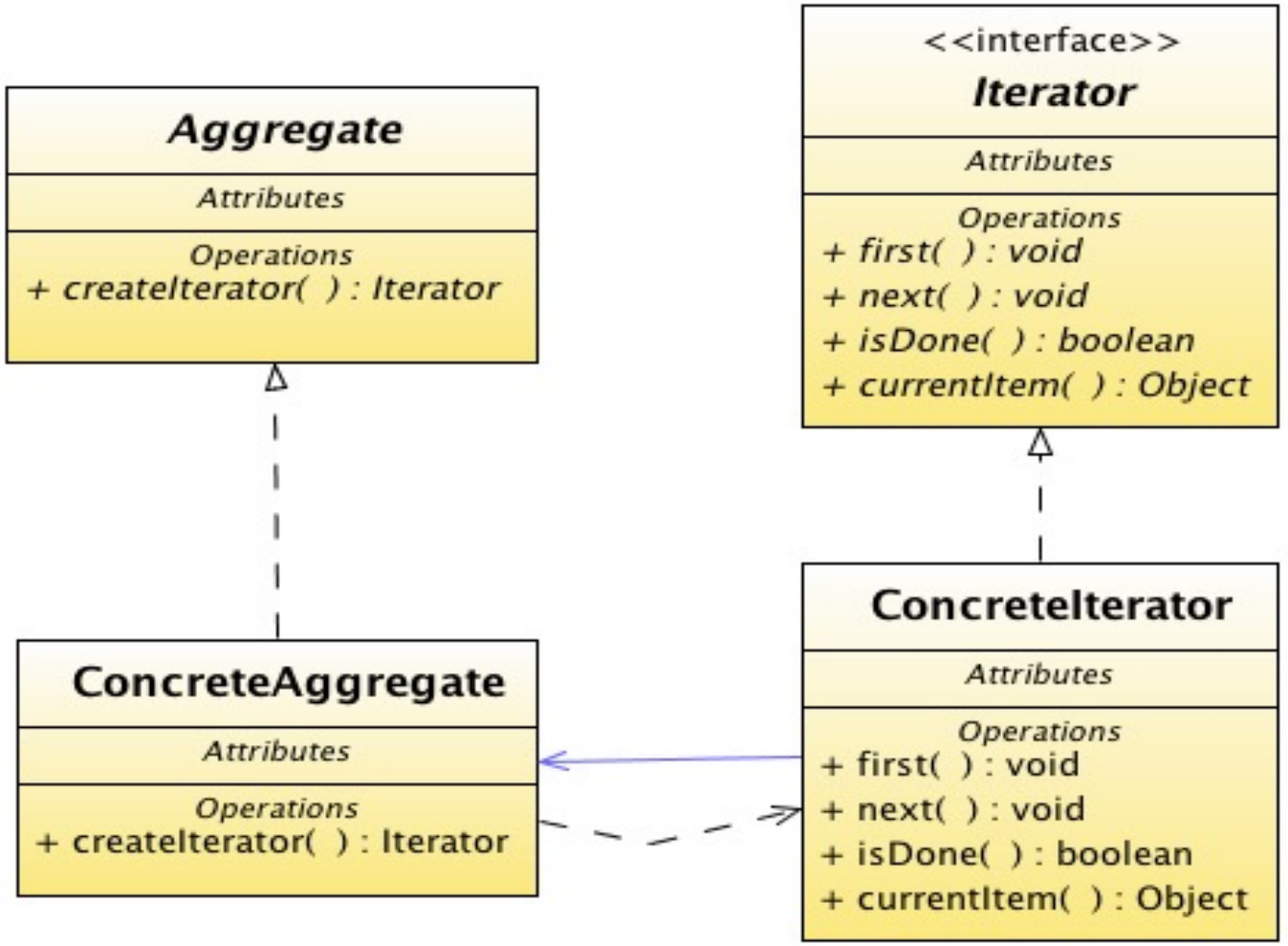
Pattern Composite

- Patterns apparentés
  - Chaine de Responsabilité
  - Décorateur (avec classe parente commune)
  - Visiteur : pour traverser le composite et appliquer des traitements transversaux, « externalisés » pour ne pas les charger.
  - Itérateur: pour parcourir les composants sans se soucier de la structure

Pattern Iterateur (Comportemental)

- Alias
  - Iterator
  - Curseur
- Intention
  - fournit un moyen d'accéder aux élèments d'un agrégat sans exposer sa structure interne.
- Motivation
  - Collections
  - Structures complexes
  - Composites

Pattern Itérateur



Pattern Itérateur

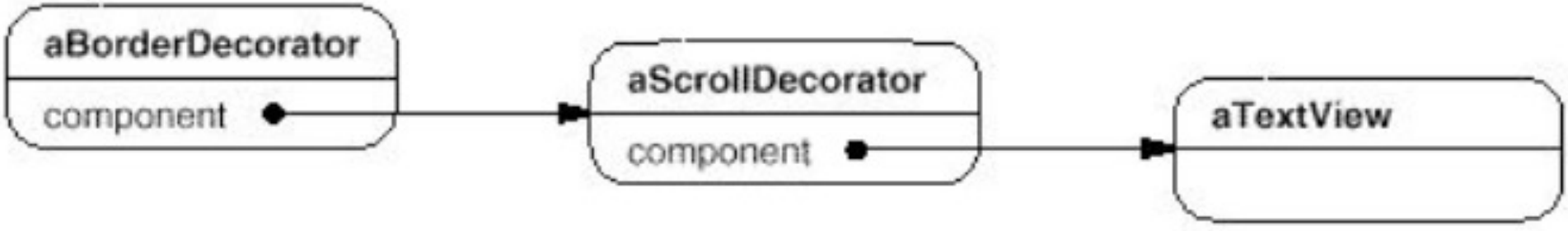
- Indications d'utilisation
  - accéder aux éléments d'un contenant sans en révéler la représentation interne
  - on ne veut (peut) pas charger une représentation avec des opérations d'accès
  - permettre plusieurs parcours simultanés (et dissociés)
  - offrir une interface commune d'itération sur diverses représentations (polymorphes) et en conséquence une algorithmique indépendante (abstraite)
- Utilisations remarquables
  - Itérateurs de Java, C++, ... sur toute séquence Itérable

Pattern Itérateur

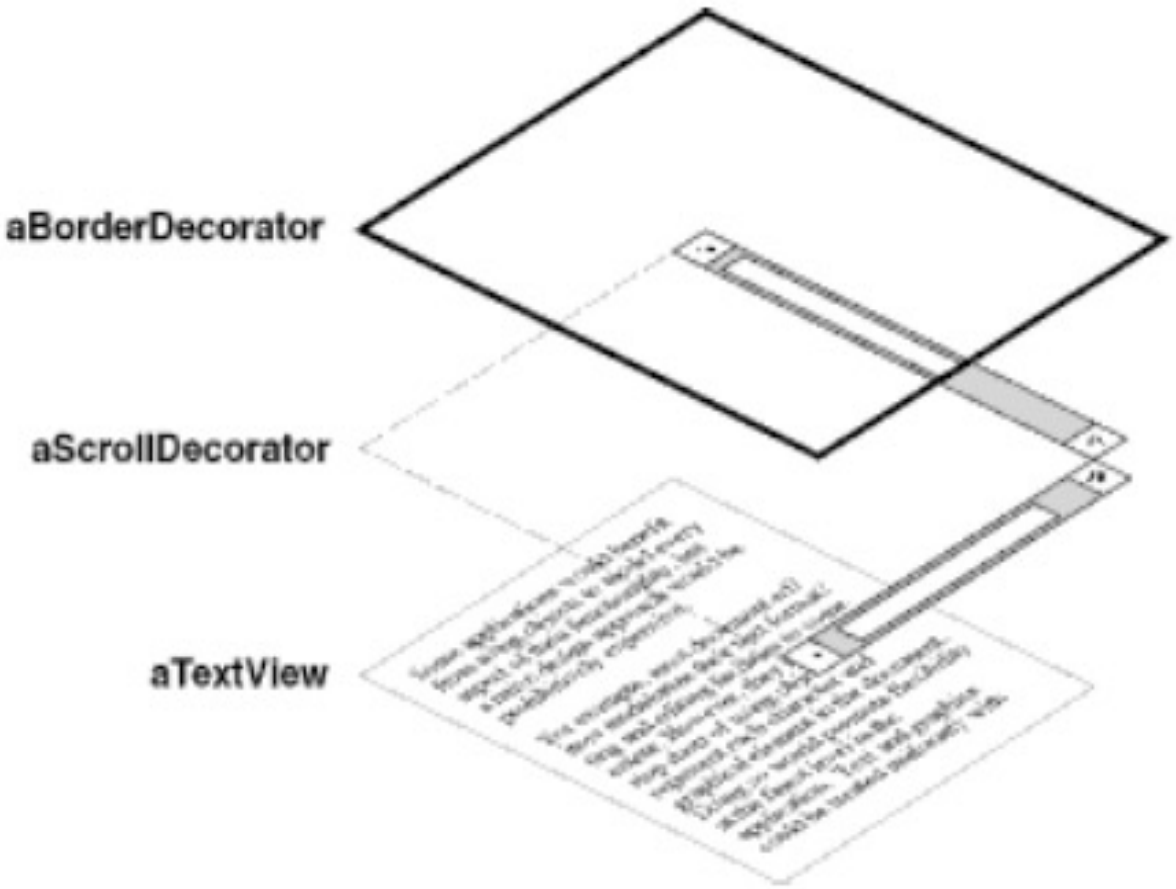
- Précautions d'implémentation
  - « aliasing » : modification de l'agrégat
    - copie ou enregistrer l'itérateur dans l'agrégat (qui gère l'ajustement)
  - Itérateur sur Composite
    - homogénéité
      - Composite:createIterator() => Iterator sur ses fils
      - Feuille: createIterator() => NullIterator
- Patterns apparentés
  - Composite : itérateurs récursifs
  - Fabrique: abstrait la création des itérateurs
  - Memento : un itérateur peut utiliser un Memento pour conserver en interne l'état de l'itération.

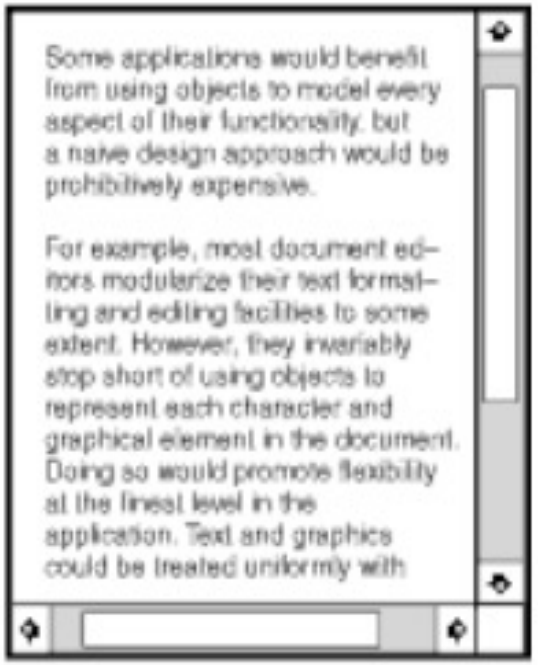
Pattern Décorateur (Structurel)

- Alias
  - Wrapper, Empaqueteur
- Intention
  - Ajouter dynamiquement des nouvelles fonctionnalités à un objet.
- Motivation (GOF)
  - on veut ajouter des propriétés à un composant d'IHM, une bordure, un scrollbar...

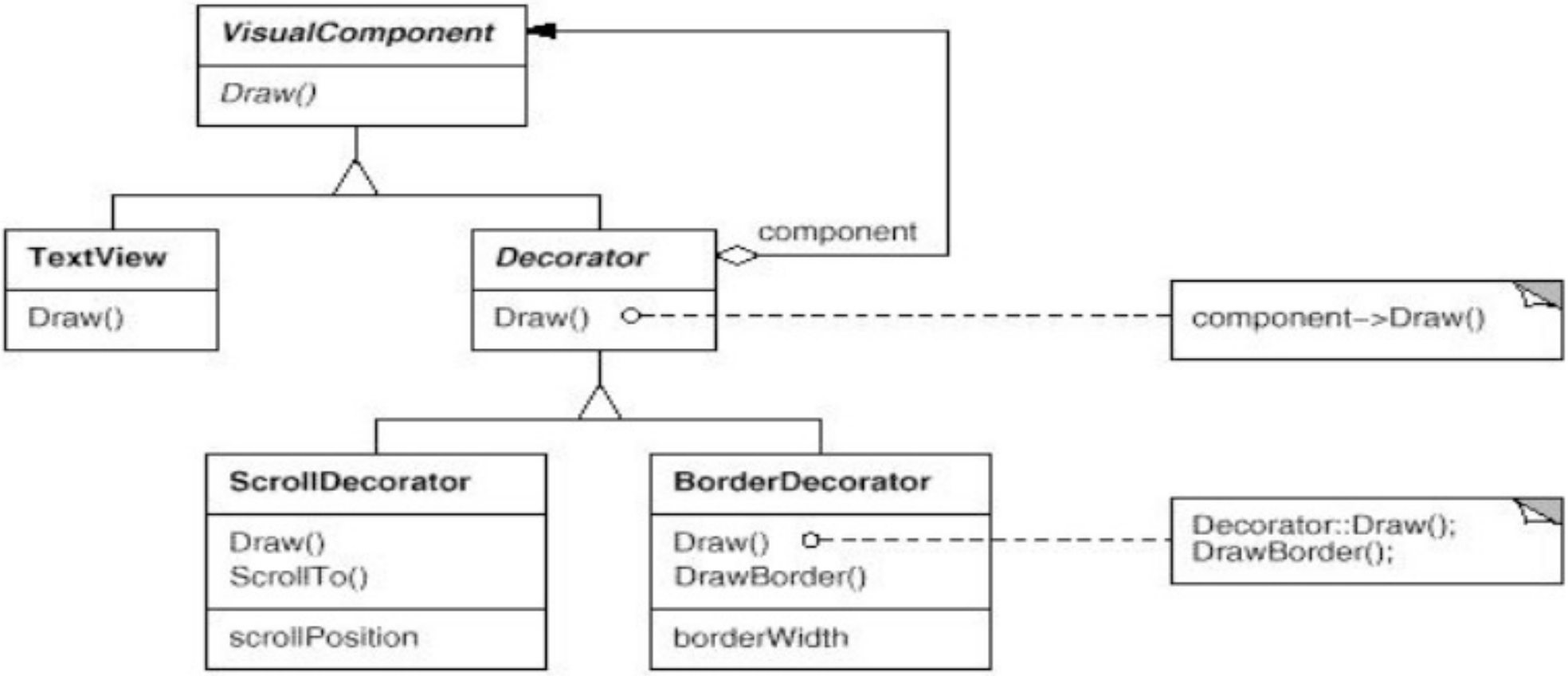


Décorateur: Motivation (GOF)

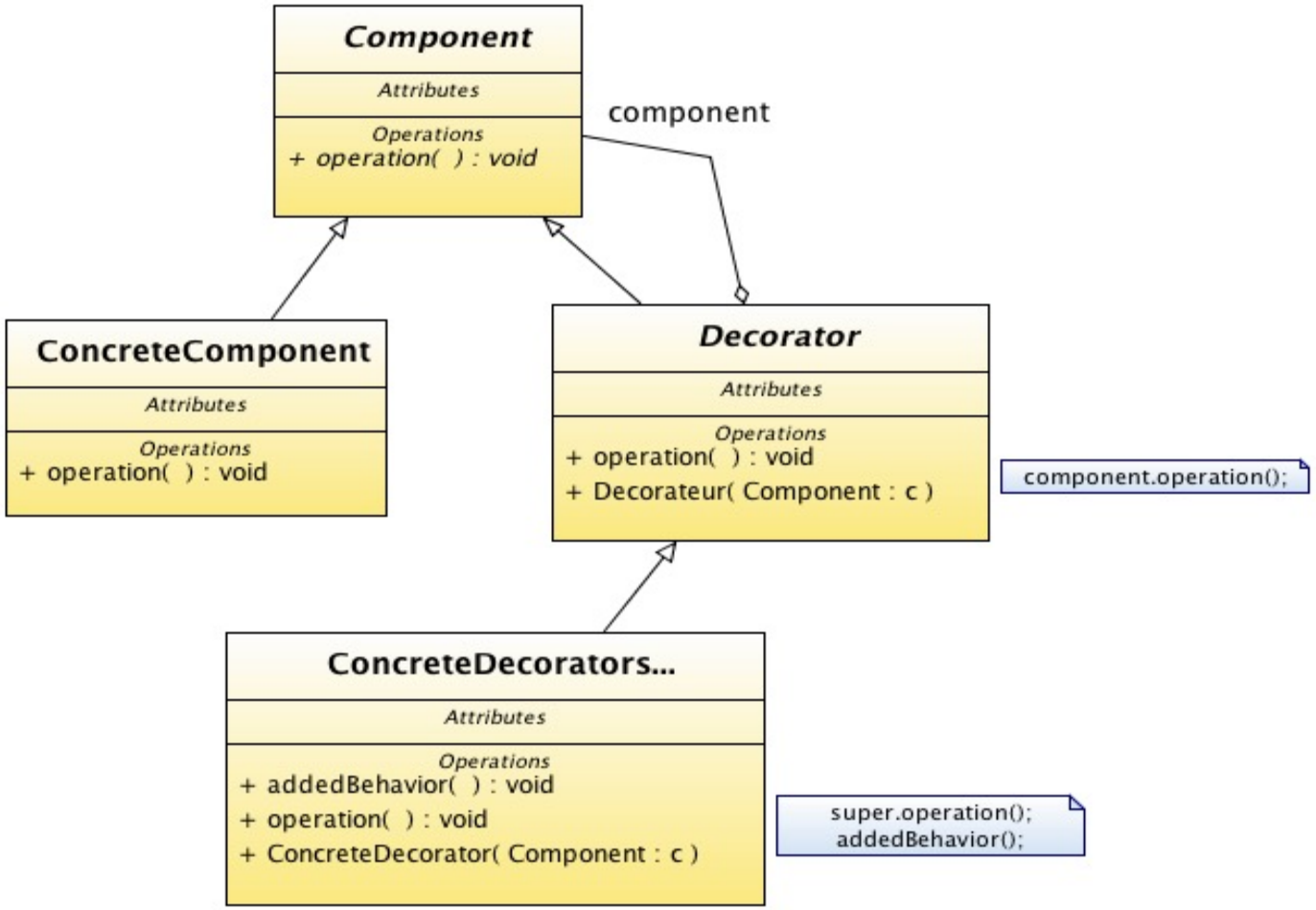




Décorateur: Motivation (GOF)



Pattern Décorateur: structure



Pattern Décorateur

- Indications d'utilisation
  - quand on veut ajouter (retirer) dynamiquement des fonctionnalités à un objet
  - ajouter des fonctionnalités à un objet sans affecter les autres (instances de la même classe)
  - c'est aussi une alternative de composition à l'héritage
    - pour éviter l'explosion des combinaisons de sous-classes
    - quand l'héritage multiple n'est pas possible
    - quand une classe ne peut être sous-classée (cachée ou fermée à l'héritage).

Pattern Décorateur

- Utilisations remarquables
  - Streams Java
  - Basics
    - InputStream, OutputStream: Byte
    - Reader, Writer: texte
  - Décorateurs (wrappers ou « filtres » en Java)
    - Buffered Stream - ajoute la bufferisation
    - Data Stream - lecture de données (int, double, ...)
    - Pushback Stream - permet les « undo »

Pattern Décorateur

```
// Ouvrir un fichier binaire
FileInputStream in = new FileInputStream("test.dat");
x=in.read(); // byte

// le bufferiser par le decorateur BufferedInputStream
BufferedInputStream bin = new BufferedInputStream(in);
x=bin.read(); // byte dans buffer

// lire des donnees
DataInputStream dbin = new DataInputStream(bin);
x=in.read(); // byte
double d = dbin.readDouble();

//
```

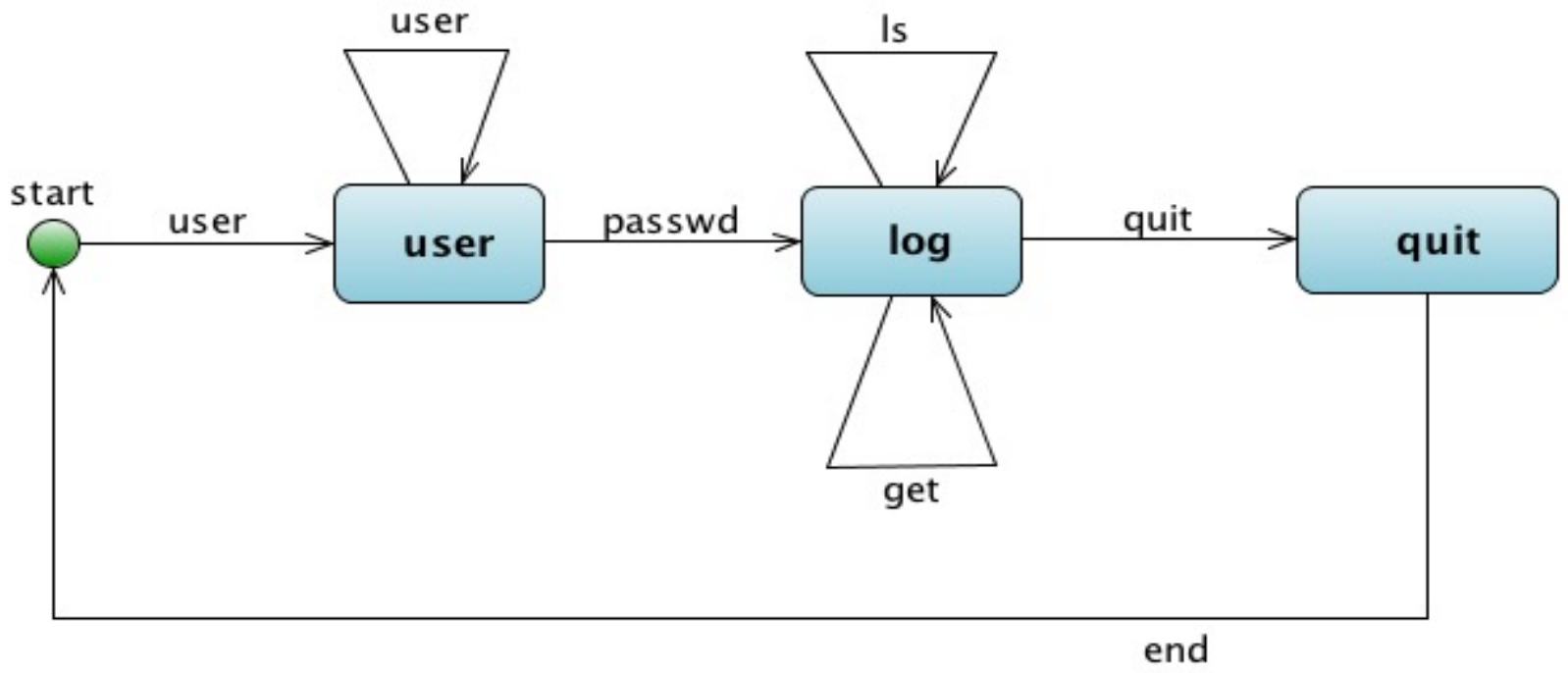
Pattern Décorateur

- Patterns Apparentés
  - différent de Adaptateur
    - Adaptateur est de niveau Classe (statique) alors que Decorateur s'applique aux objets (dynamique).
    - Adaptateur permet de **convertir** un protocole dans un autre (protocole d'implémentation) alors que Decorateur implémente le protocole et **l'enrichit**
  - Composite : Decorator utilise la composition **mais à un seul niveau** et d'un seul composant. En plus Décorateur **ajoute** des responsabilités, alors que Composite les factorise.
  - Strategy : Decorator permet d'ajouter une enveloppe **externe** (nouvelle « peau ») à un objet alors que Strategy permet **de changer la structure interne** ou de déporter une implémentation de Composant trop complexe.
  - Chaines de Responsabilités

Pattern Etat (Comportemental)

- Alias : State
- Intention
  - permet à un objet de modifier son comportement quand son état change
  - tout se passe comme si l'objet changeait de classe en fonction de son état.
- Motivation: connexion ftp
  - `user username` : première commande
  - `passwd` : valide la connection et permet les autres commandes
  - `ls` : liste les fichiers
  - `get(file)` : récupère le fichier file
  - `quit` : délogue l'utilisateur

Pattern Etat: motivation



Pattern Etat: motivation

- les commandes n'ont pas le même fonctionnement selon l'état de la connexion

```
public class FTP {
     // etats possibles
     static final int QUIT = 1;
     static final int USER_KNOWN= 2;
     static final int START = 3;
     static final int LOGGED = 4;

     // etat
     private int state = START;
     private String userName;
     private String password;
```

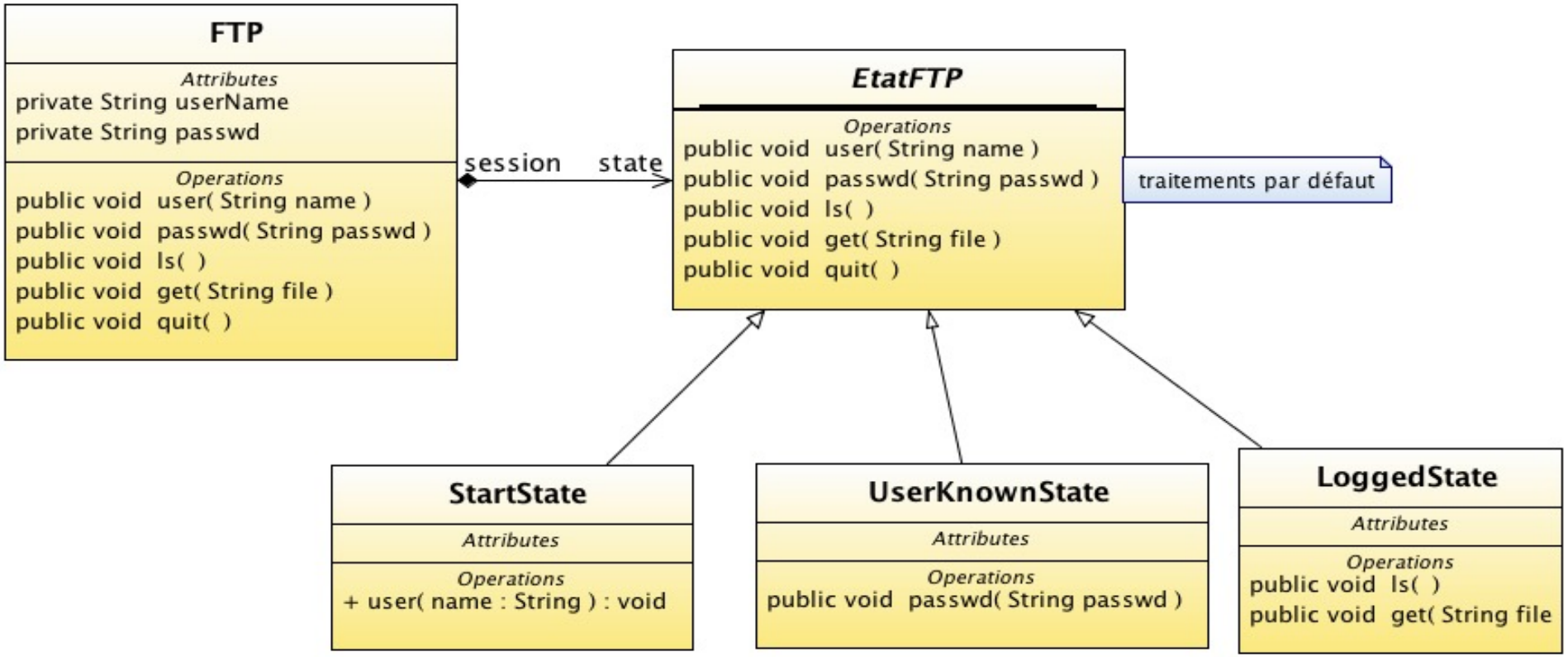
Pattern Etat: motivation

```
public void user(String userName) {
        switch (state) {
          case START: {
            this.userName = userName;
            state = USER_KNOWN;
            break;
          }
          default: { // invalid command
            error();
            userName = null;
            password = null;
            state = START;
          }
        }
      }
```

Pattern Etat: motivation

```
public void passwd(String password) {
        switch (state) {
          case USER_KNOWN: {
            this.password = password;
            if (validateUser())
              state = LOGGED;
            else {
              error ();
              userName = null;
              password = null;
              state = START;
            }}
          default: {  // commande invalide
            error();
            end();
            state = START;
          }}
...}
```

Pattern Etat: motivation



Pattern Etat: motivation

```
package ftp;
public class FTP {
 // etats possibles
 private EtatFTP startState, userKnownState, loggedState,
   quitState;
 // etat
 private EtatFTP state;
 private String userName;
 private String password;

 public FTP(){
  startState= new StartState(this);
  userKnownState = new UserKnownState(this);
  loggedState = new LoggedState(this);
  quitState = new QuitState(this);
  setStartState(); }
```

Pattern Etat: motivation

```
 public void user(String name) {state.user(name);}
 public void passwd(String paswd){state.passwd(paswd);}
 public void ls() {state.ls();}
 public void get(String file) {state.get(file);}
 public void quit() {state.quit();}

 void setStartState() {
   userName = password = "";
   state = startState;
 }
 void setUserKnownState() {
   state = userKnownState;
 }
 ...
}
```

Pattern Etat: motivation

```
package ftp;
class EtatFTP {
 protected FTP session;

 void user(String userName) {par defaut ...}
 void pass(String passwd) {...}
 void ls() {...}
 void get(String file) {...}
 void quit() {...}

 EtatFTP(FTP session) {
  this.session = session;
 }
}
```

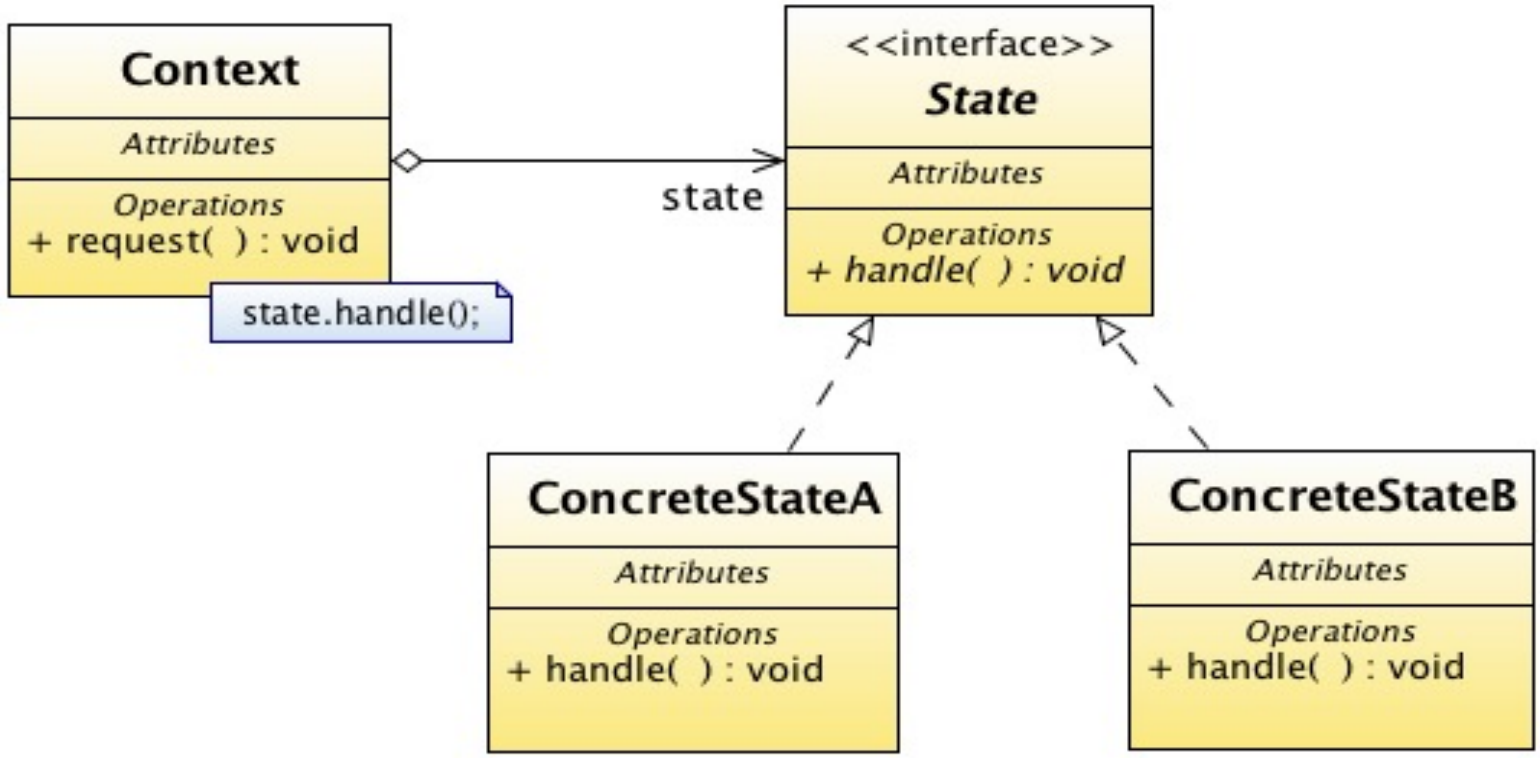
Pattern Etat: motivation

```
package ftp;
class StartState extends EtatFTP {
 void user(String userName) {
   if(validate(userName)) {
        session.setName(userName);
        session.setUserKnownState();
  }
  else message("unknown user name");
 }
 void passwd(String passwd) {
  message("user before");
 }
 boolean validate(String userName) {...}
 ...
}
```

Pattern Etat: motivation

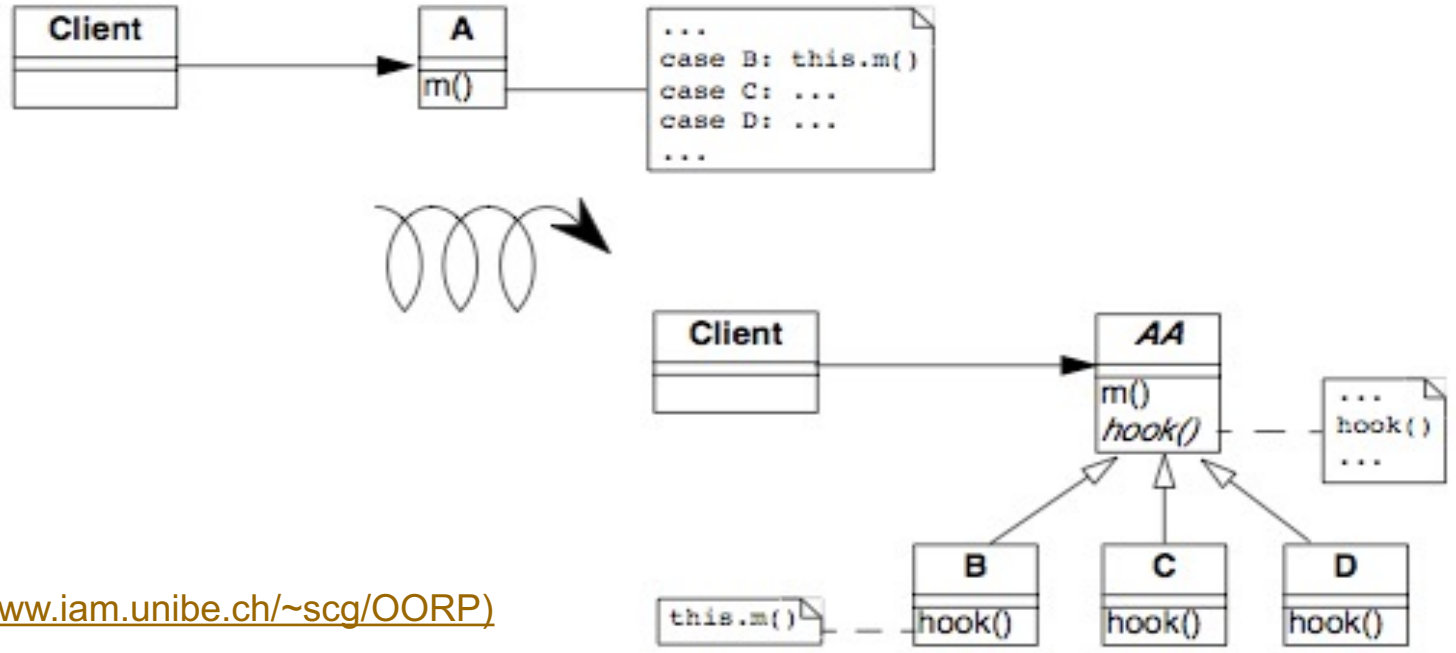
```
package ftp;
class UserKnownState extends EtatFTP {
 void user(String userName) {
   session.setStartState();
   session.user(userName);
 }
 void passwd(String passwd) {
   if (validate(passwd)) {
        session.setPassword(passwd);
        session.setLoggedState()
   } else {
        message("unknown user name");
        session.setStartState();
   }}
 boolean validate(String passwd) {...}
 ...
}
```

Pattern Etat: structure



Pattern Etat: indications d'utilisation

- quand le comportement d'un objet dépend de son état et ce changement est dynamique (polymorphisme d'état).
- quand l'algorithme d'une méthode est trop complexe en switch, fonction de l'état des variables d'instance.



(http://www.iam.unibe.ch/~scg/OORP)

Pattern Etat

- Avantages
  - modularise le comportement de l'objet, explicite sa structuration en états isolés (« modules internes » regroupant variables et sous-méthodes spécifiques)
  - facilite l'ajout de nouvelles modalités par l'introduction de nouvelles sous-classes d'Etat.
- Inconvénients
  - multiplie le nombre de classes et d'objets mais cette complexité est à comparer avec la complexité du code des méthodes (switch).
    - cet inconvénient peut être limité si l'on dispose de modules ou de classes internes (« inner classes » en Java)

Pattern Etat

- Utilisations remarquables
  - implémentation des machines à états (Diagrammes états-transitions d'UML)
  - processeurs de commandes et serveurs (FTP, mail, EJB)
  - IHM contextuels (menus contextuels, actions dépendantes de l'état).
  - programmation par modes ou modalités
- Patterns apparentés
  - Stratégie
    - même structure apparente mais intentions différentes, Stratégie ne gère pas d'état (changement, ...)
    - encapsulation vs. exportation: changement d'état automatique vs. stratégie choisie par le client
  - Singleton : souvent utilisé pour garantir l'unicité de chaque état
  - Memento : mémorise l'état (historique) d'un objet. N'est pas concerné par le polymorphisme de comportement.

Pattern Stratégie (Comportemental)

- Alias
  - Strategy
- Intention
  - Définit une famille d'algorithmes interchangeables pour résoudre un même problème, et qui peuvent évoluer indépendamment des applications qui l'utilisent.
- Motivation...

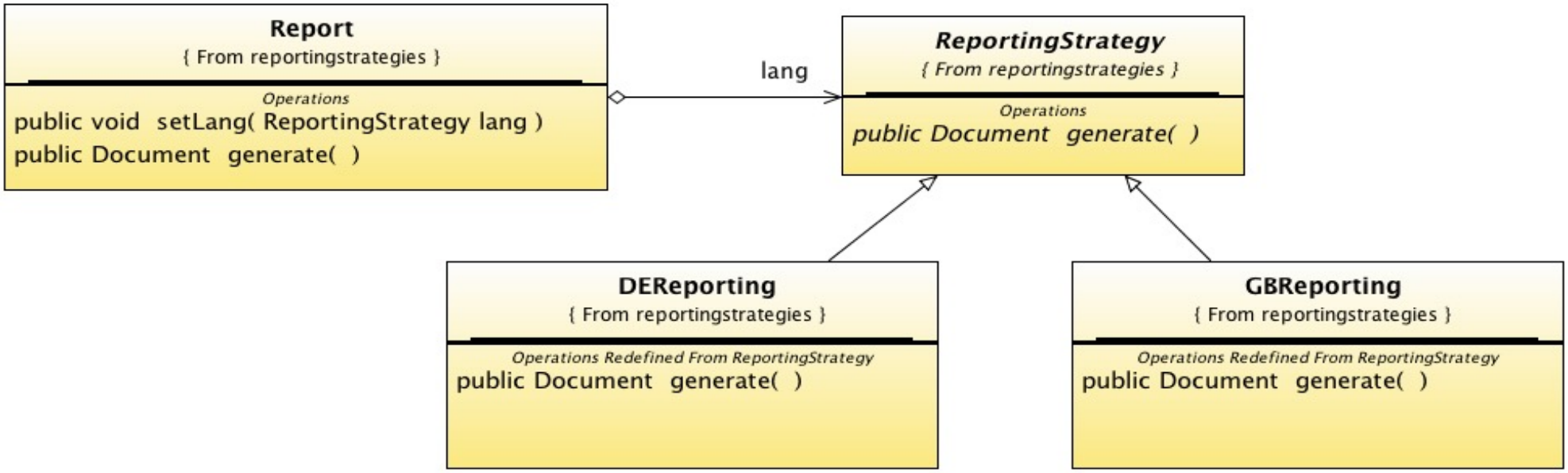
Pattern Stratégie: motivation

```
public class Report {
    static final int gb = 1, fr =2, de = 3, sp = 4;
    int lang;
    public Document generate() {
        Document doc;
        switch (lang) {
            case gb: doc = generateGB(); break;
            case fr: doc = generateFR(); break;
            case de: doc = generateDE(); break;
            case sp: doc = generateSP(); break;
            default: doc = new Document(); break;
        }
        return doc;
    }
    public void setLang(int lang){this.lang = lang;}
    private Document generateGB() {...}
    private Document generateFR() {...} ...}
```

Pattern Stratégie: motivation

```
// avec Strategie

public class Report {
    ReportingStrategy lang;
    public Document generate() {
        return lang.generate();
    }
    public void setLang(ReportingStrategy lang) {
        this.lang = lang;
}}
```

Pattern Stratégie



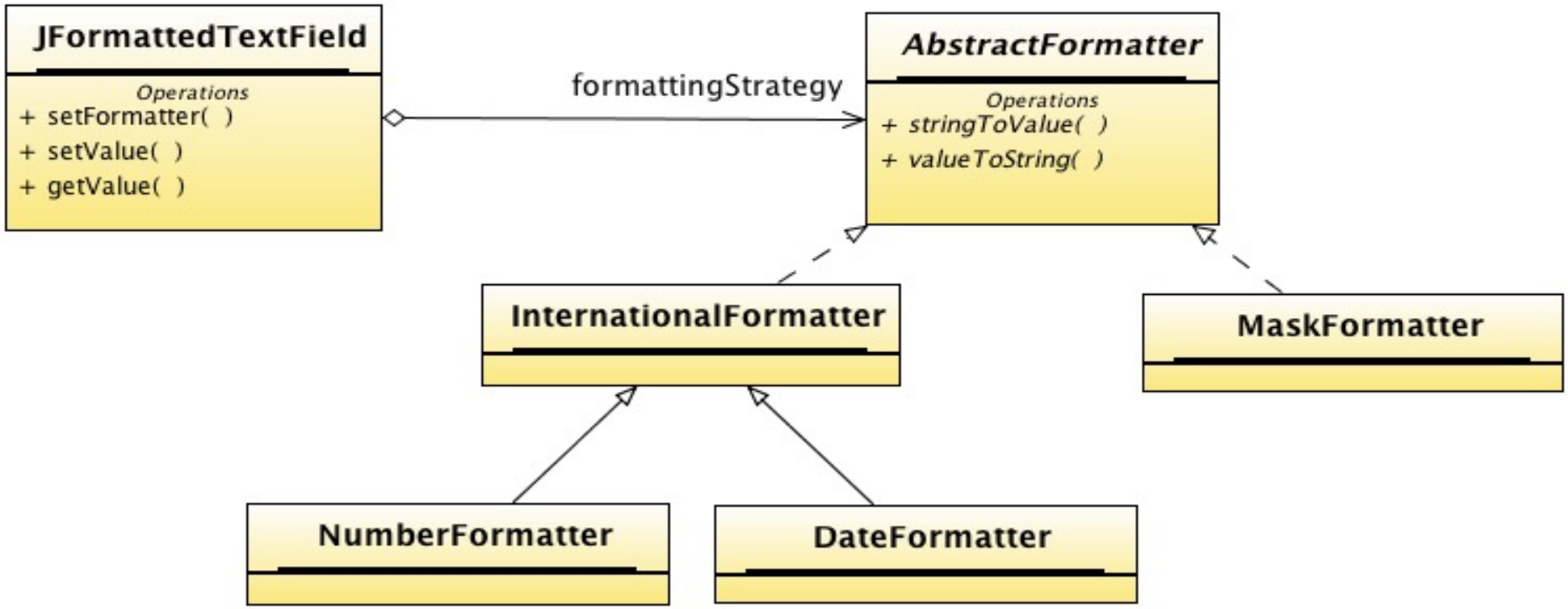
Pattern Stratégie

- Indications d'utilisation
  - plusieurs classes ne différent que dans leur comportement face au même protocole
  - les comportements sont interchangeables et déterminés par le client
  - vous avez besoin de différentes variantes d'un même algorithme
  - un algorithme utilise des SD que le client n'a pas à connaître.
  - une classe (Context) définit des choix alternatifs au même problème, qui figurent sous la forme de conditionnelles (switch).

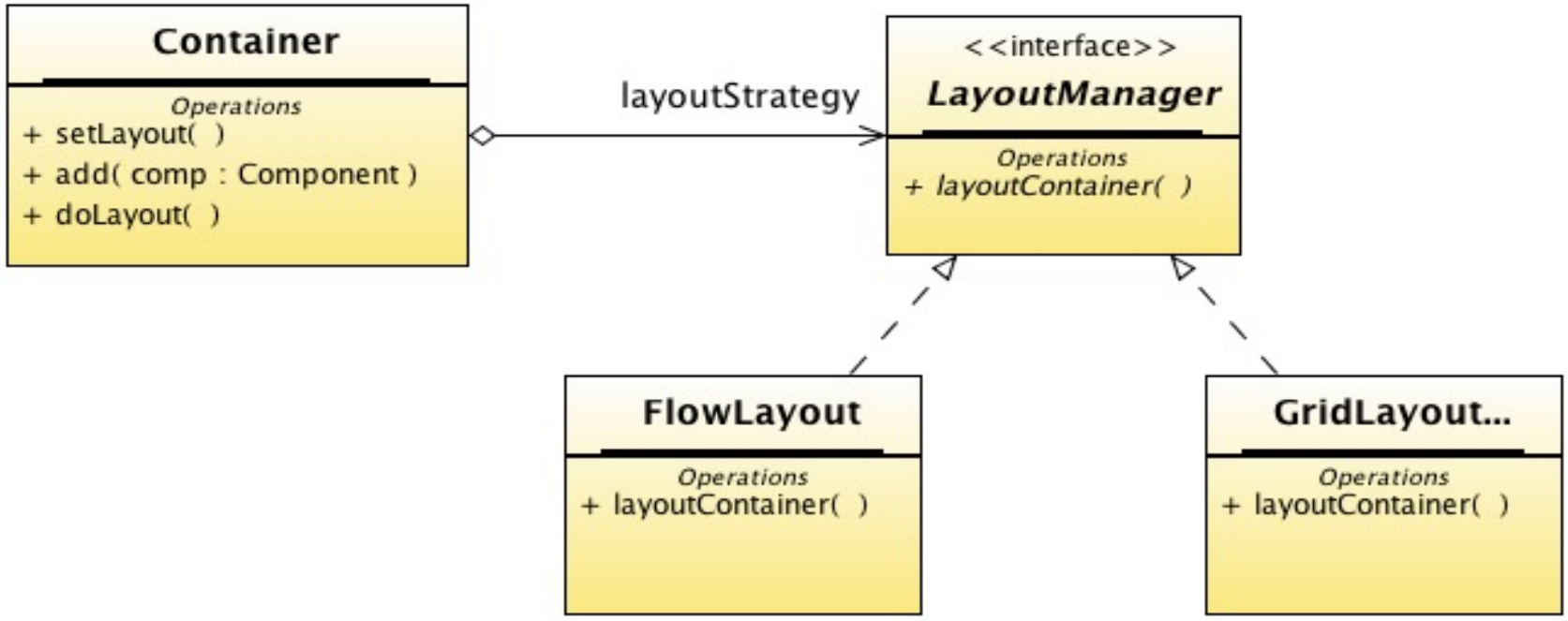
Pattern Stratégie

- Avantages
  - alternative au sous-classement de Context pour obtenir des variantes de comportement.
  - supprime les switch
  - offre un choix d'implémentations pour un même traitement
  - il est facile d'ajouter une nouvelle stratégie.
- Inconvénients
  - multiplie les objets
  - tous les algorithmes doivent répondre au même protocole.

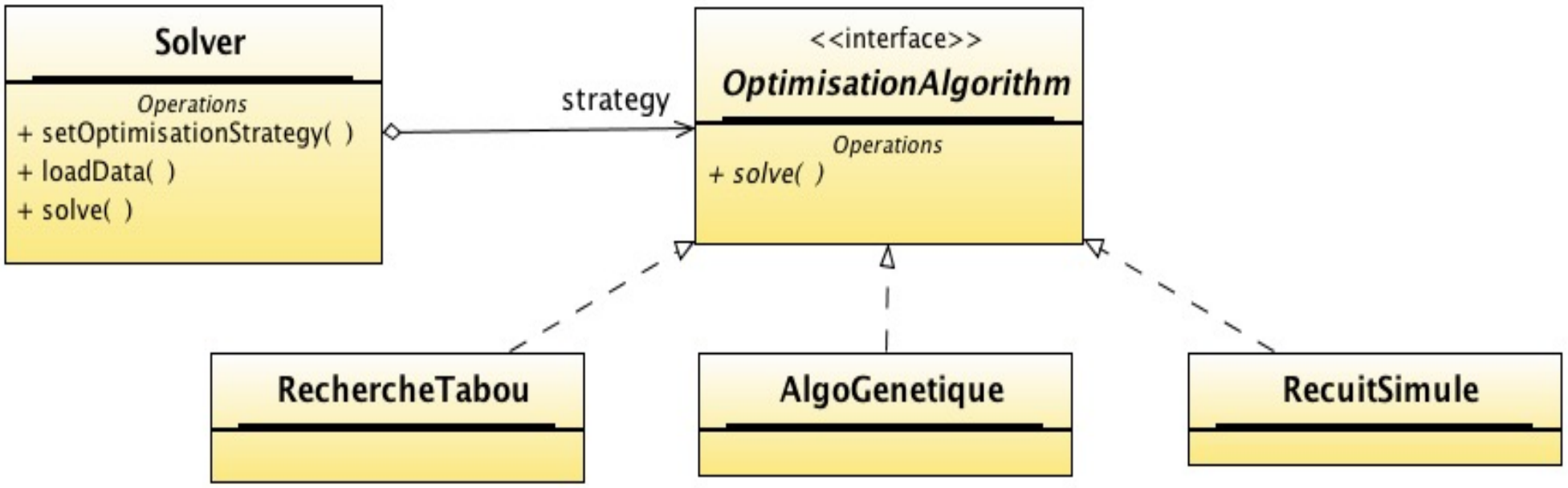
Stratégie: utilisations remarquables



Stratégie: utilisations remarquables



Stratégie: utilisations remarquables



Stratégie

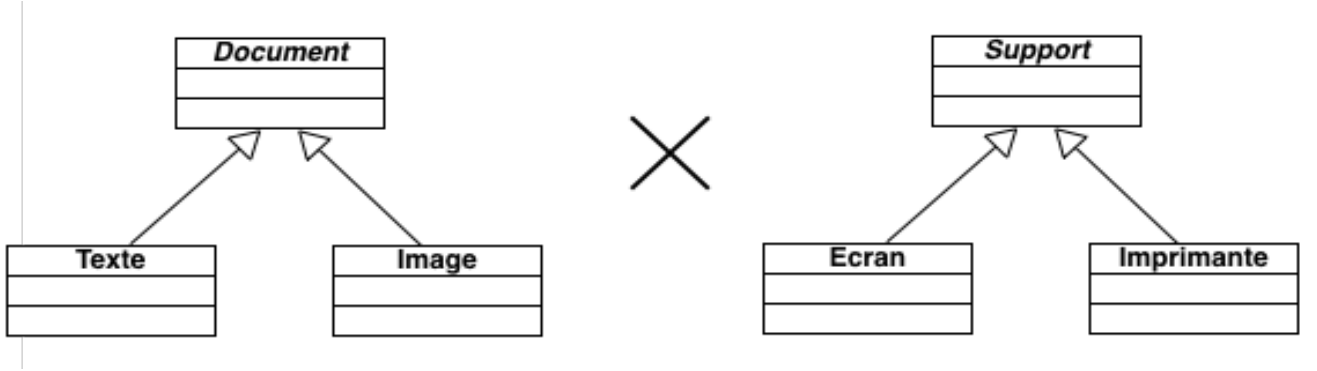
- Patterns apparentés
  - Etat
    - même structure apparente mais intentions différentes:
      - les stratégies sont indépendantes et interchangeables
      - la stratégie est choisie par le client (exportation) alors que le changement d'état est « automatique »
  - Décorateur
    - est aussi une alternative au sous-classement mais
      - cumulatif sur les fonctionnalités (ajout), alors que Stratégie est alternatif sur une fonctionnalité
      - publique alors que Stratégie (et Etat) est géré par le Contexte (voire encapsulé).

Pattern Visiteur (Comportemental)

- Alias : Visitor
- Intention
  - Explicite des fonctionnalités (visiteurs) applicables transversalement aux objets d'une structure (graphe, agrégat, composite, ...).
  - Greffer des nouvelles fonctionnalités sans modifier leurs classes
  - Produits de hiérarchies de classes « double polymorphisme »
- Motivations
  - GOF : exploitations d'AST
    - analyse sémantique (type checking)
    - génération(s), optimisation, transformation de code
    - pretty-printing
    - mesure...
  - Gestion de parcs matériels

Problème du « double polymorphisme »

On veut pouvoir produire différents types de ***documents*** sur différents types de ***supports***



à démultiplier sur :

- la hiérarchie de types de ***Document***:
  - Texte: .txt, .ps, .pdf, .doc
  - Image: .jpg, .png, .svg
  - autres: diagrammes formatés (UML/XMI), docs XML, …
- la hiérarchie de types de ***Support***:
  - File
  - Smartphone (Android, IOS, …)
  - Tablettes ...

Problème

print	Texte	Image
Ecran	*implem_ET*	*implem_EI*
Imprimante	*implem_IT*	*implem_II*

- ce traitement (*print*) dépend de 2 types d'objets: *Support* (Ecran, Imprimante) × *Document (*Texte, Image)
- d'où le problème du **« double polymorphisme »**

Problème

- Comment profiter du polymorphisme pour simplifier le code des applications devant gérer la diversité de ces 2 types d'objets (documents × supports)
  - utilitaires système : file manager, browsers, …
  - éditeurs, IDE, GED, tableurs, …

- Typiquement

```
Document doc; // Texte, Image, …
Support support; // Ecran, Imprimante, …
doc.print(support); // par polymorphisme …
```

Solution naïve…

- Tout d'abord où ranger les `implem_XX` de `print`? Dans les classes de supports ou de documents ?
- Partons de leur implantation dans les classes de supports
  - plus conforme à la réalité technologique:
    prise en charge des divers types de document par les supports
  - sachant que le raisonnement est symétrique et vaudrait pour l'autre choix

```
abstract class Support {
   abstract void print(Texte doc);
   abstract void print(Image doc);
}
class Ecran extends Support {
   void print(Texte doc) { implemET }
   void print(Image doc) { implemEI }
}
class Imprimante extends Support {
   void print(Texte doc) { implemIT }
   void print(Image doc) { implemII }
}
```

Solution naïve…

- on aurait alors dans la classe Document :

```
abstract class Document {
   void print(Support support) {
       support.print(this);
} … }
```

- en espérant obtenir par polymorphisme (objectif initial):

```
Document doc;
Support support;
doc.print(support); // par polymorphisme
```

- mais ça ne marche pas …

Solution naïve…

- ça ne compile même pas!

- en effet
  - à ce niveau (compilation) le type (statique) de `this` est `Document`
  - et il n'y a pas de méthode `print(Document)` dans la classe `Support` …

- Idée!

ajouter cette méthode à la classe `Support` …

Solution naïve…

```
abstract class Support {
  abstract void print(Texte doc);
  abstract void print(Image doc);
  void print(Document doc) {
        if (doc instanceof Texte)
                this.print((Texte)doc);
        else if (doc instanceof Image)
                this.print((Image)doc);
        else if …
        …
}…}
```

☹ peu satisfaisant, peu modulaire, peu évolutif…

- en comprenant bien que dans la ***liaison dynamique*** de méthode (à l'exécution) seul le *type dynamique* du ***destinataire*** est pris en compte, ceux des paramètres n'intervenant pas
- le type (statique) des ***paramètres*** n'intervenant qu'à la ***compilation*** : liaison statique, polymorphisme statique ou « de surcharge » (overloading).

Solution

Remplacer ce « switch » de type programmé à la main (dynamiquement) dans la classe Support par la **copie du code**:

```
void print(Support support) {
   support.print(this);
 }
```

dans chaque classe de Document, ce qui permet la sélection (statique) de la bonne méthode `print` par le **type statique de `this`**. C'est la solution du « double-dispatch ».

```
Document doc;
Support support;
doc.print(support);
```

avec par exple doc = unTexte et support = unEcran on a bien :

`unTexte.print(unEcran)`

⇒`Texte : unEcran.print(this)` avec type statique de this = Texte

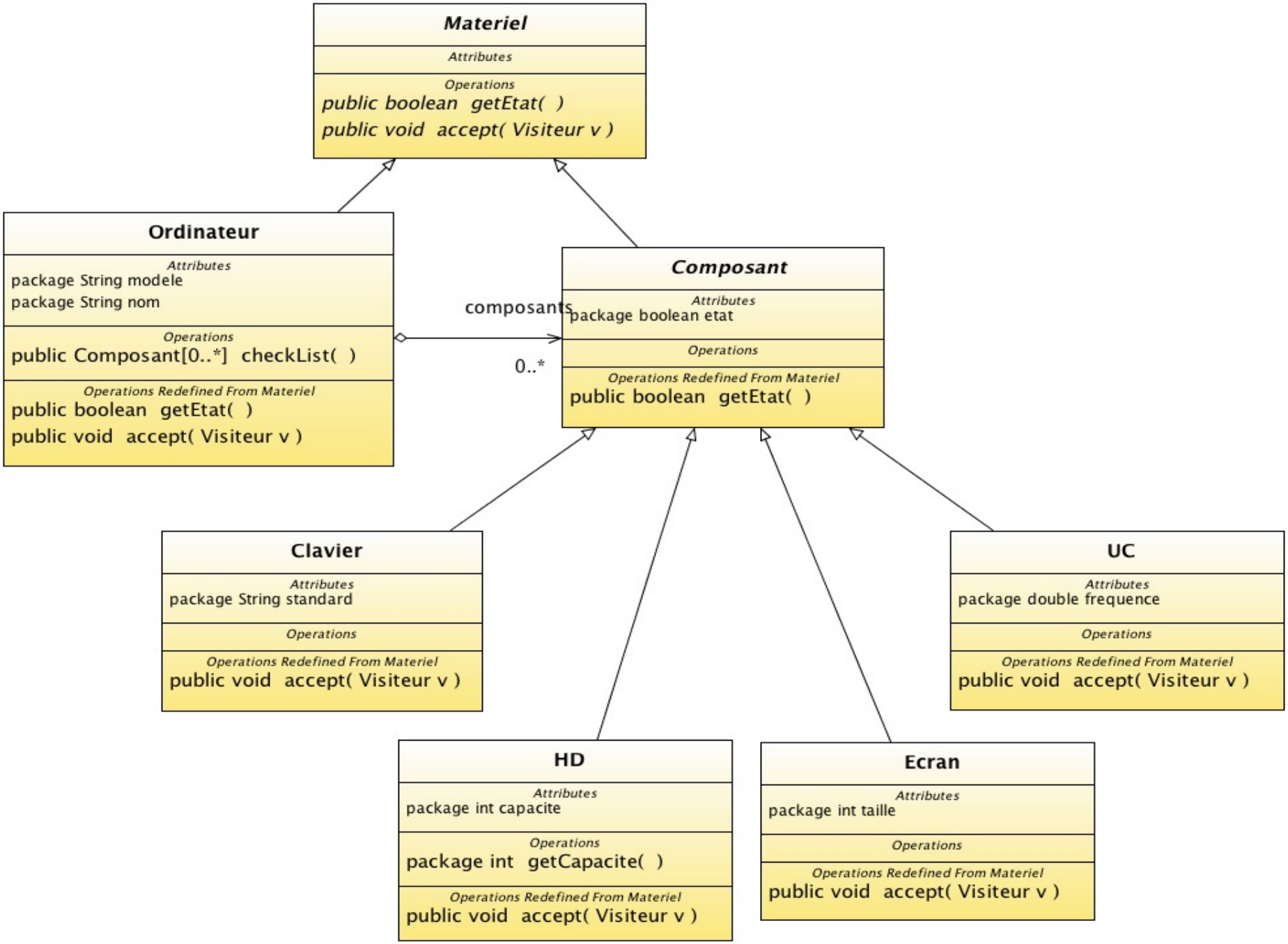
⇒`Ecran : print(Texte)` ☺

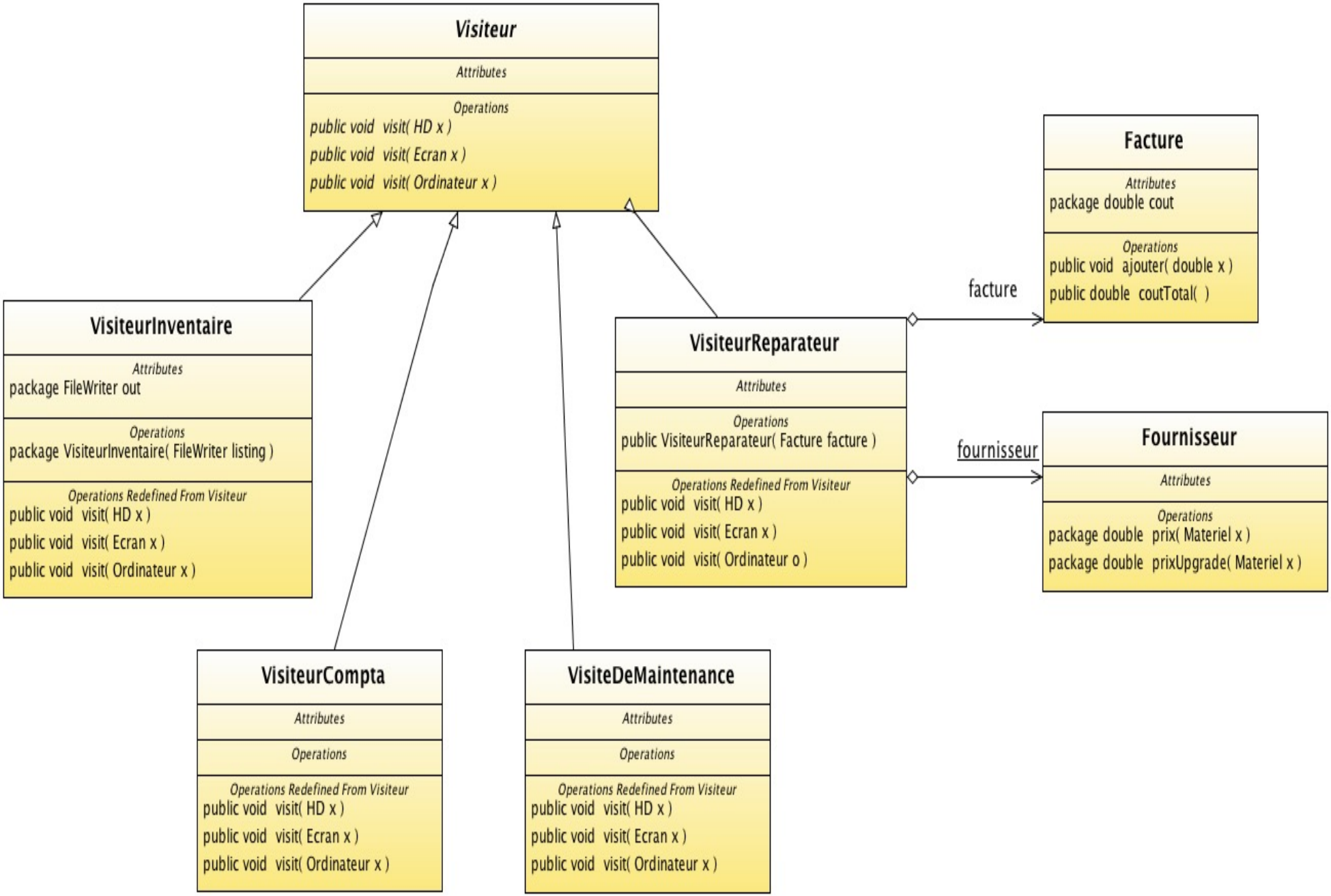
A savoir …

- La même démarche s'appliquerait symétriquement si on était parti du rangement des `implemXX` dans les classes de document.
- Noter que les langages entièrement dynamiques ne font pas mieux (Smalltalk, Pharo, Javascript, Python…), reportant le polymorphisme statique de surcharge sur le nommage (statique) des méthodes.
- Ce n'est pas un problème de langages statiquement ou dynamiquement typés mais un problème inhérent à la POO (« Programmation Orientée UN Objet »)

```
class Ecran extends Support {
   printText(doc) { implemET }
   printImage(doc) { implemEI }
} …
class Texte extends Document {
   void print(Support support) {
            support.printText(this);
} …
```

- Le « double polymorphisme » est sous-jacent au **Design Pattern Visiteur** qui est plus général. Exemple : application à la gestion d'un parc informatique...


Materiel
Attributes
Operations
public boolean getEtat( )
public void accept(Visiteur v)
Ordinateur
Attributes
package String modele
package String nom
Operations
public Composant[0..*] checkList( )
Operations Redefined From Materiel
public boolean getEtat( )
public void accept(Visiteur v)
composants
0..*
Composant
Attributes
package boolean etat
Operations
Operations Redefined From Materiel
public boolean getEtat( )
Clavier
Attributes
package String standard
Operations
Operations Redefined From Materiel
public void accept(Visiteur v)
HD
Attributes
package int capacite
Operations
package int getCapacite( )
Operations Redefined From Materiel
public void accept(Visiteur v)
Ecran
Attributes
package int taille
Operations
Operations Redefined From Materiel
public void accept(Visiteur v)
UC
Attributes
package double frequence
Operations
Operations Redefined From Materiel
public void accept(Visiteur v)


Visiteur
Attributes
Operations
public void visit(HD x)
public void visit(Ecran x)
public void visit(Ordinateur x)
Facture
Attributes
package double cout
Operations
public void ajouter(double x)
public double coutTotal( )
facture
VisiteurInventaire
Attributes
package FileWriter out
Operations
package VisiteurInventaire(FileWriter listing)
Operations Redefined From Visiteur
public void visit(HD x)
public void visit(Ecran x)
public void visit(Ordinateur x)
VisiteurReparateur
Attributes
Operations
public VisiteurReparateur(Facture facture)
Operations Redefined From Visiteur
public void visit(HD x)
public void visit(Ecran x)
public void visit(Ordinateur o)
fournisseur
Fournisseur
Attributes
Operations
package double prix(Materiel x)
package double prixUpgrade(Materiel x)
VisiteurCompta
Attributes
Operations
Operations Redefined From Visiteur
public void visit(HD x)
public void visit(Ecran x)
public void visit(Ordinateur x)
VisiteDeMaintenance
Attributes
Operations
Operations Redefined From Visiteur
public void visit(HD x)
public void visit(Ecran x)
public void visit(Ordinateur x)

Pattern Visiteur: motivation

```
public class VisiteDeMaintenance extends Visiteur {
 public void visit(HD x) {
  if(x.getEtat())
    System.out.println("HD "+x+ "fonctionne");
  else
    System.out.println("HD HS:"+x+x.getCapacite());
    }
 public void visit(Ecran x) {...}
 public void visit(Ordinateur x) {
   if(x.getEtat())
     System.out.println(x.getNom()+" fonctionne");
   else {
     System.out.println("Ordinateur HS: "+x.getNom());
     System.out.println("\t modele:"+x.getModele());
     System.out.println("\t checkup: "+x.checkList());
}}}
```

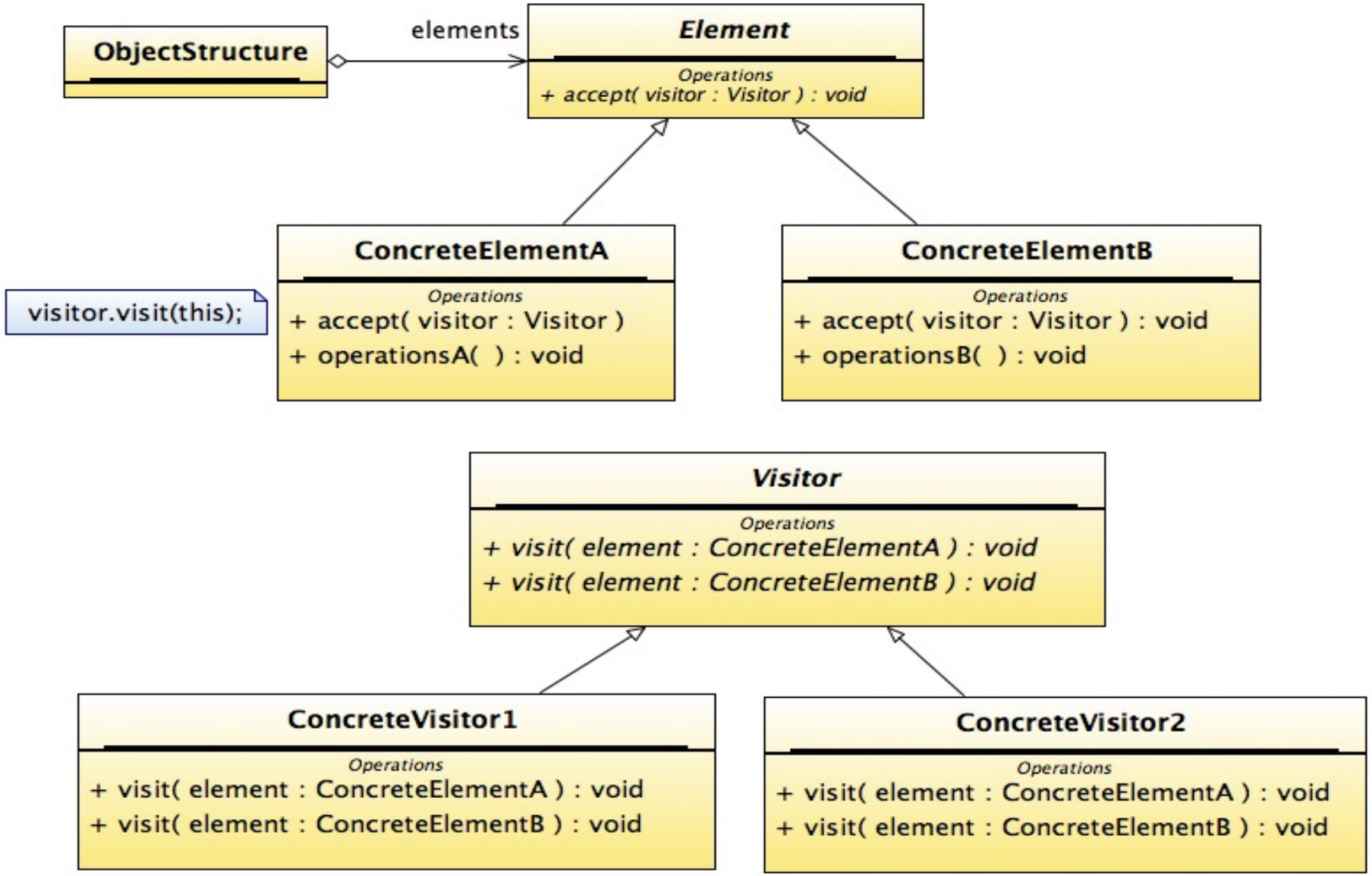
Pattern Visiteur: motivation

```
public class VisiteurReparateur extends Visiteur {
 Facture facture;
 public void visit(HD x) {
   if(!x.getEtat())
     facture.ajouter(getFournisseur.prixUpgrade(x));}
 public void visit(Ecran x) {...}
 public void visit(Ordinateur o) {
  if(!o.getEtat()) {
   double prixEchangeStandard = getFournisseur.prix(o);
   double prixReparation=0.0;
   for (Composant c : o.checkList())
    prixReparation+=fournisseur.prix(c);
   if(prixReparation<prixEchangeStandard)
    facture.ajouter(prixReparation);
   else
    facture.ajouter(prixEchangeStandard);
}}}
```

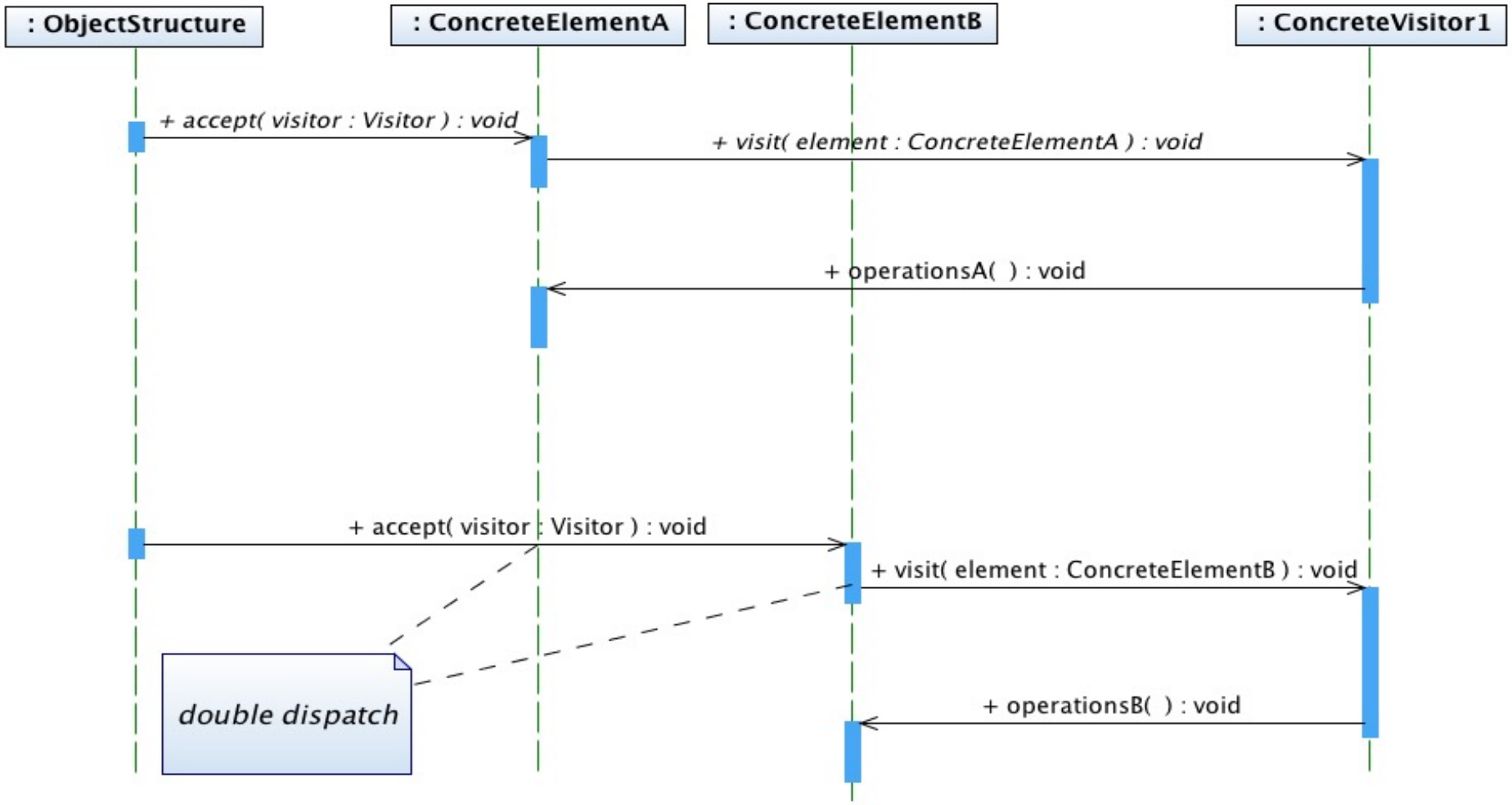
Pattern Visiteur: motivation

```
public class Gestion {
 Collection<Materiel> materiels;
 Ordinateur serveur1, serveur2;
 Collection<HD> disksServeurs;

 public void maintenanceServeurs() {
  VisiteDeMaintenance main = new VisiteDeMaintenance();
  serveur1.accept(main);
  serveur2.accept(main);
  for (HD disk : disksServeurs) disk.accept(main);
 }

 public void reparation() {
  Facture facture = new Facture();
  VisiteurReparateur rep=new VisiteurReparateur(facture);
  for (Materiel matos : materiels) matos.accept(rep);
}}
```

Pattern Visiteur: structure



Pattern Visiteur : collaborations



Pattern Visiteur : collaborations

- Double dispatch
  - OO : simple dispatch
    - c'est l'objet destinataire qui détermine le traitement à appliquer
  - Visiteur
    - {Objets} X {Visiteurs} -> traitement
      - x.accept(v)
        - x détermine le type d'objet à traiter par...
        - ... v qui détermine le traitement effectif
- permet de définir des traitements qui dépendent à la fois
  - de l'objet (ConcreteElement)
  - et de l'acteur (Visiteur)

Pattern Visiteur

- Indications d'utilisation
  - quand on veut appliquer et expliciter des traitements annexes et transversaux (fonctionnalités transverses) à une structure d'objets
  - quand la structure d'objets contient beaucoup de classes d'interfaces distinctes pour réaliser ces traitements
  - ces traitements dépendent des types d'objets mais on ne veut (peut) pas « polluer » leurs classes avec de nouvelles opérations
  - ces traitements sont indépendants
  - quand on doit souvent ajouter de tels traitements mais que la structuration en classes est relativement stable
    - sinon il faut modifier l'interface de tous les Visiteurs!

Pattern Visiteur

- Avantages
  - il est facile d'ajouter un traitement : ajouter une classe de Visiteur
  - localisation du traitement dans le Visiteur (et non éparpillement dans les classes)
  - les visiteurs peuvent collecter de l'information dans leur visite (sinon ceci oblige à la passer en paramètre dans toutes les opérations de parcours!).

- Inconvénient
  - il est difficile d'ajouter une nouvelle classe de ConcreteElement.
  - l'interface des éléments visités doit être suffisante pour effectuer les traitements de visite (ce qui peut mener à trop « ouvrir » les objets).
  - orienté objet ou fonction? ne pas en abuser...

Pattern Visiteur

- Utilisations remarquables
  - Compilateurs
  - Traitements de repositories d'objets
    - DOM, métamodèles (IDM) ...
  - Composants métiers
    - un visiteur = une fonction du système
- Patterns apparentés
  - Composite
    - traitements transversaux sur une structure hiérarchique hétérogène
  - Décorateur
    - mais unitaire et ne gère pas le double dispatch.

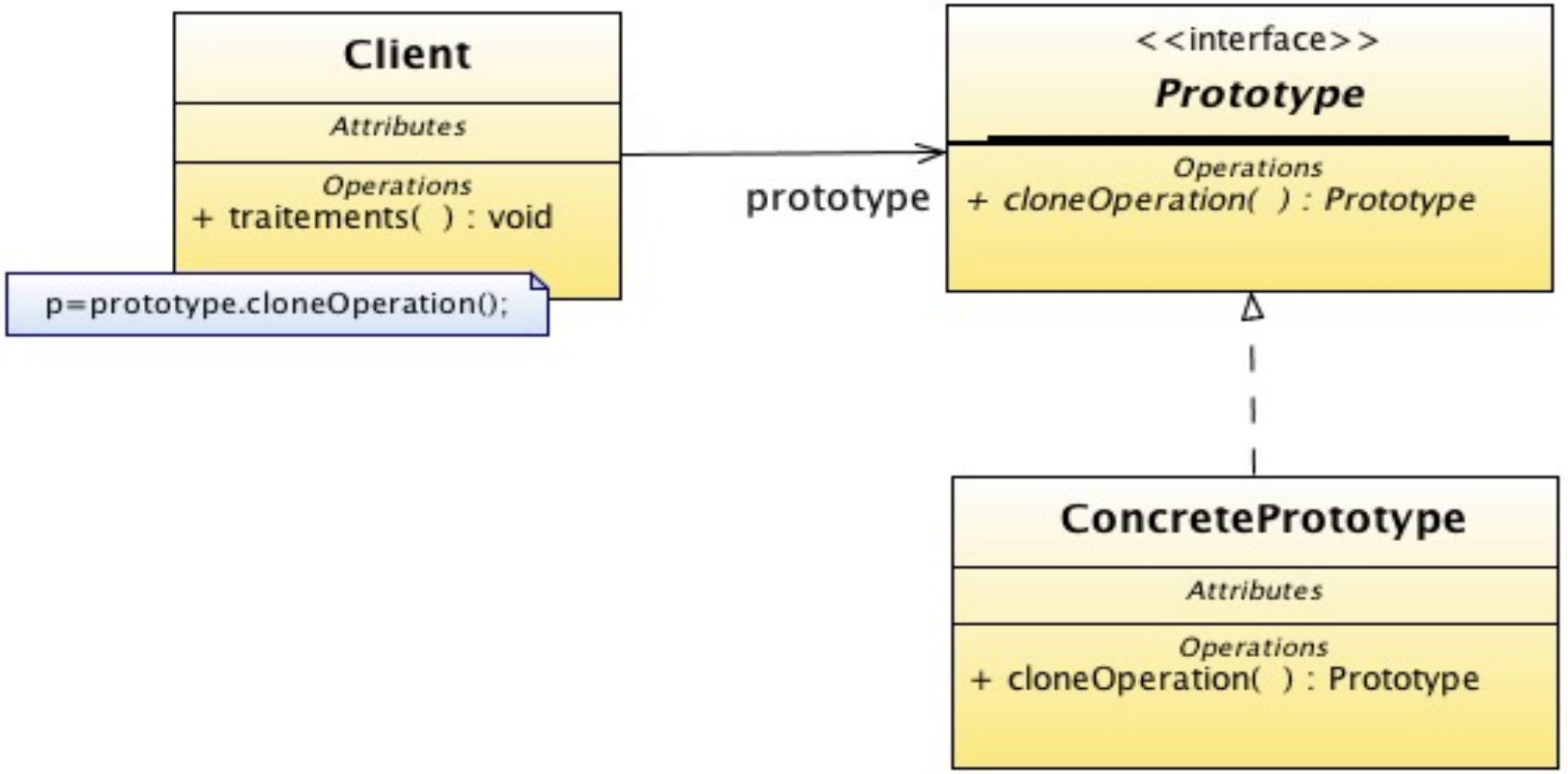
Pattern Prototype (Créateur)

- Alias
  - clone, objet type
- Intention
  - Spécifie le type des objets à créer à partir d'une instance de prototype, et crée de nouveaux objets en copiant ce prototype.
- Motivation
  - prototypes de matériels informatiques...

Pattern Prototype

```
 Ordinateur o1 = new Ordinateur("Dell", "clodion01");
        o1.addComposant(new UC(2.2, 100));
        o1.addComposant(new Ecran(19));
        o1.addComposant(new Clavier("azerty"));
        o1.addComposant(new HD(100));
Ordinateur o2 = new Ordinateur("Dell", "clodion02");
        o2.addComposant(new UC(2.2, 100));
        o2.addComposant(new Ecran(19));
        o2.addComposant(new Clavier("azerty"));
        o2.addComposant(new HD(100));
//prototype
Ordinateur proto = new Ordinateur("Dell", "proto");
        proto.addComposant(new UC(2.2, 100));
        proto.addComposant(new Ecran(19));
        proto.addComposant(new Clavier("azerty"));
        proto.addComposant(new HD(100));
o1=proto.clone("clodion01");
o2=proto.clone("clodion02");
```

Pattern Prototype : structure



Prototype: utilisations

- si les instances d'une classe peuvent prendre un état parmi un ensemble de configurations (=prototypes). C'est souvent le cas pour des composites.
- pour limiter le nombre de sous-classes, une par configuration type, ou de fabriques (alternative)
- quand la classe (de configuration) n'est connue qu'à l'exécution
  - en effet Prototype remplace l'instanciation avec référence *statique* à la classe par la copie avec référence *dynamique* à un objet type.
  - il simplifie le code de construction (switch)

Prototype: utilisations

- quand des types de produits (configurations) doivent pouvoir être créées (ou ajoutées) par l'utilisateur
  - Prototype offre d'une certaine façon des capacités de programmation à l'utilisateur lui-même (« programmation dynamique de classes »)
- pour offrir des « catalogues de classes »

```
switch type
   "imprimante": new Imprimante()
   "ordinateur": new Ordinateur() ...
=> catalogue.get(type).clone()
```

"imprimante"	Imprimante
"ordinateur"	Ordinateur
"hd"	HD

Prototype: utilisations

- Généralisation aux configurations: catalogue fournisseur

  ordinateur 19', 2.2GHz, RAM 2 GO, 100 GO HD

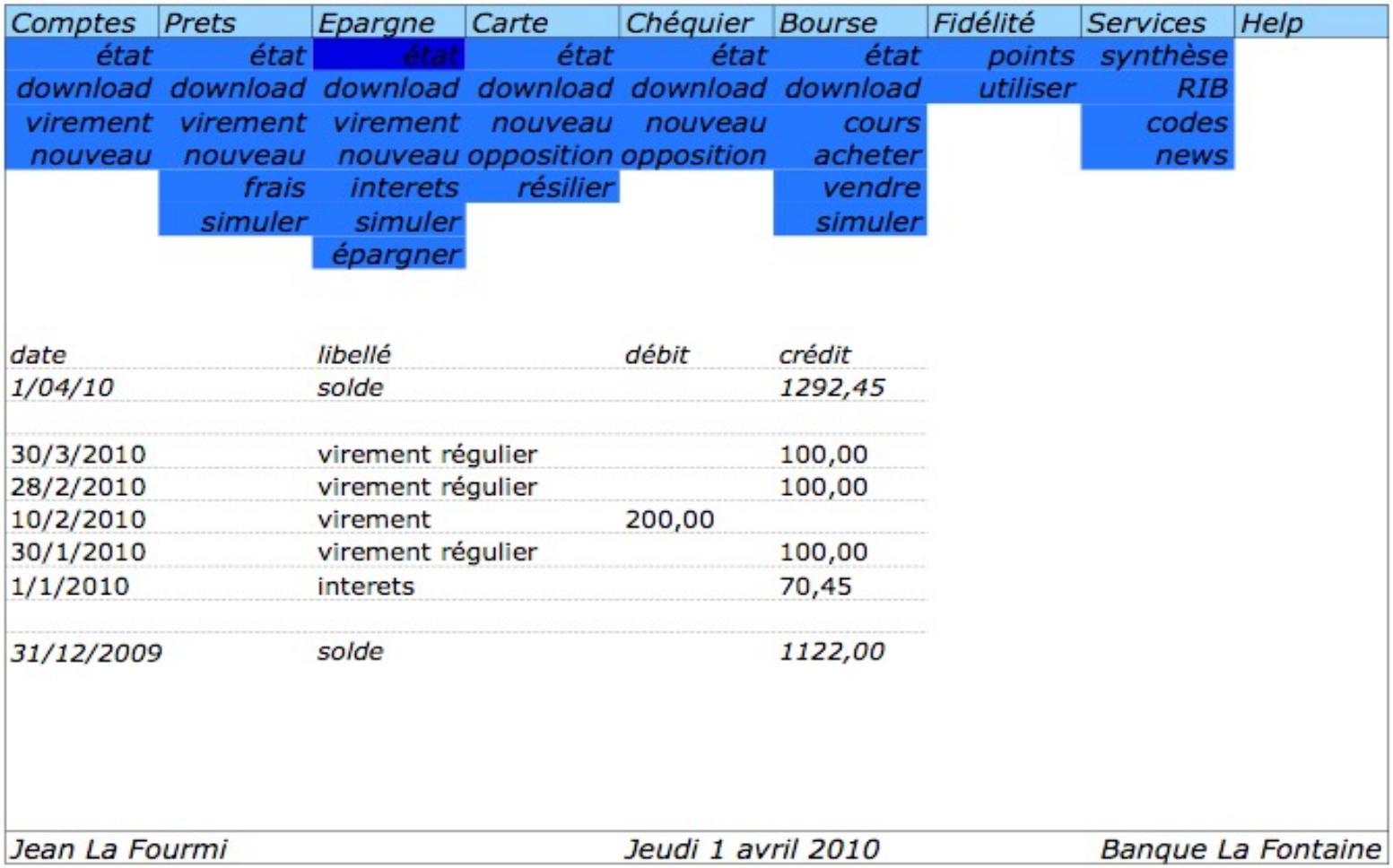
  ordinateur 15', 2GHz, RAM 500 MO, 20 GO HD ...
- Prototype est une solution pour les langages qui n'exposent pas les classes à l'exécution comme C++ et contrairement à Java ou PHP.
  - dans ce cas la solution est simplifiée :

```
(Ordinateur)Class.forName("Ordinateur").newInstance();
```

  - mais ceci ne résout pas le problème des configurations (dynamique)
  - noter aussi que contrairement au prototype, l'objet Class n'est pas une instance de produit! Le prototype est un produit de plein droit, co-opté comme représentant type, qui peut être exploité comme tel : traitements, comparaison (recherche dans le catalogue par instance matching), simulation, ...

Pattern Commande (Comportemental)

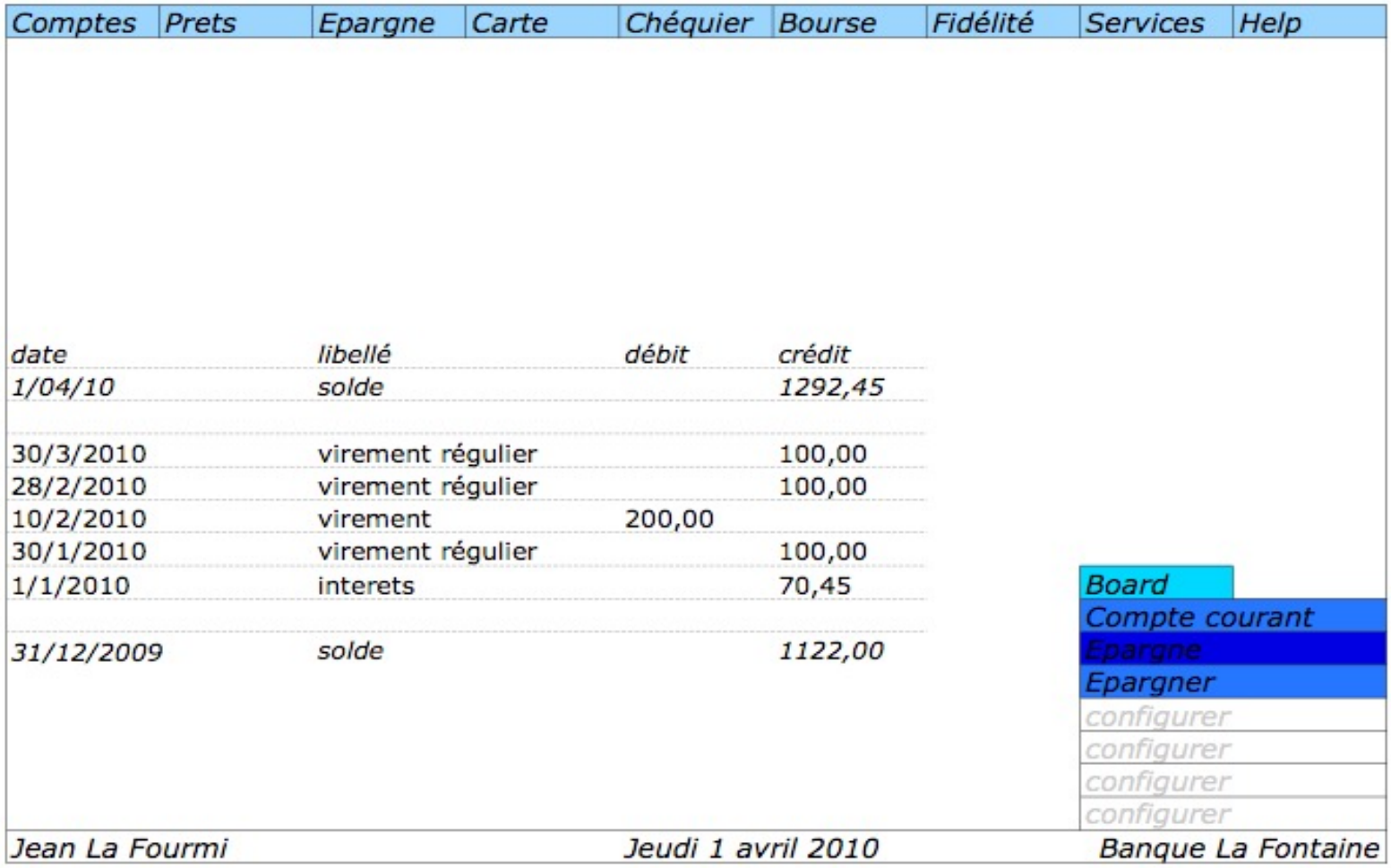
- Intention
  - Encapsuler les requêtes comme des objets (« réifier ») pour permettre le paramétrage dynamique des applications.
  - Commande permet aussi le contrôle de l'exécution de requêtes: différer, mise en attente, annuler, traces (log).
- Motivation


Comptes
Prets
Epargne
Carte
Chéquier
Bourse
Fidélité
Services
Help
état
download
virement
nouveau
état
download
nouveau
frais
simuler
état
download
virement
nouveau
interets
simuler
épargner
état
download
nouveau
opposition
résilier
état
download
nouveau
opposition
état
download
cours
acheter
vendre
simuler
points
utiliser
synthèse
RIB
codes
news
date
libellé
débit
crédit
1/04/10
solde
1292,45
30/3/2010
virement régulier
100,00
28/2/2010
virement régulier
100,00
10/2/2010
virement
200,00
30/1/2010
virement régulier
100,00
1/1/2010
interets
70,45
31/12/2009
solde
1122,00
Jean La Fourmi
Jeudi 1 avril 2010
Banque La Fontaine

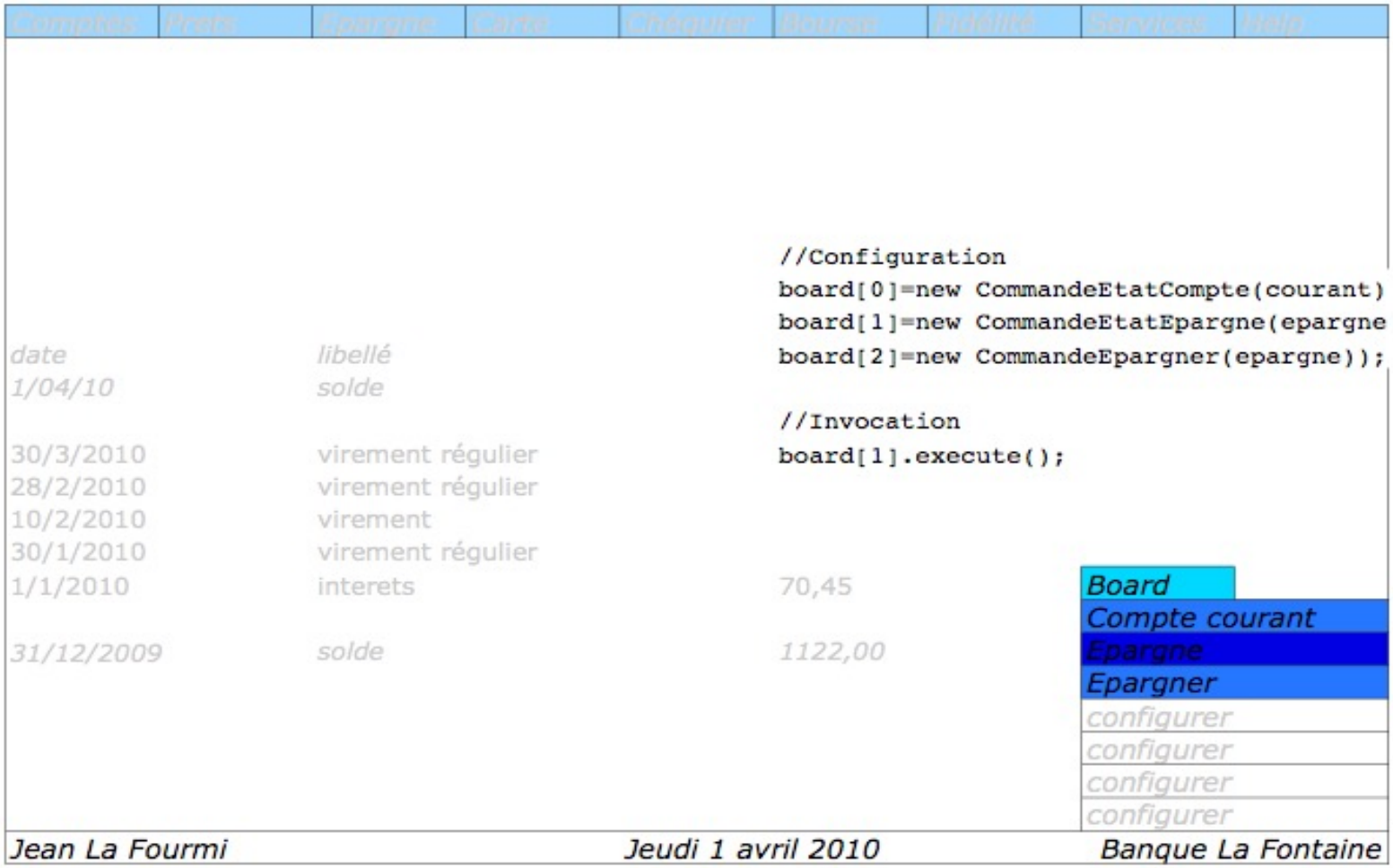
Comptes | Prets | Epargne | Carte | Chéquier | Bourse | Fidélité | Services | Help

- état
- download
- cours
- acheter
- vendre
- simuler

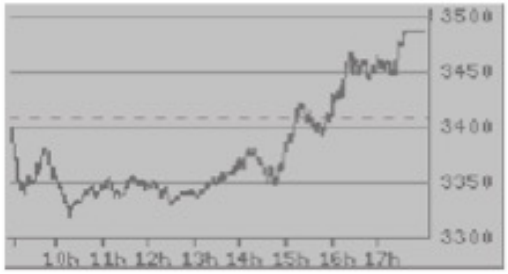
Jean La Cigale | Jeudi 1 avril 2010 | Banque La Fontaine


Comptes
Prets
Epargne
Carte
Chéquier
Bourse
Fidélité
Services
Help
date
libellé
débit
crédit
1/04/10
solde
1292,45
30/3/2010
virement régulier
100,00
28/2/2010
virement régulier
100,00
10/2/2010
virement
200,00
30/1/2010
virement régulier
100,00
1/1/2010
interets
70,45
31/12/2009
solde
1122,00
Board
Compte courant
Epargne
Epargner
configurer
configurer
configurer
configurer
Jean La Fourmi
Jeudi 1 avril 2010
Banque La Fontaine


Comptes
Prets
Epargne
Carte
Chéquier
Bourse
Fidélité
Services
Help
3500
3450
3400
3350
3300
10h 11h 12h 13h 14h 15h 16h 17h
Board
Compte courant
Bourse
Acheter
Vendre
configurer
configurer
configurer
Jean La Cigale
Jeudi 1 avril 2010
Banque La Fontaine


Comptes
Prets
Epargne
Carte
Chéquier
Bourse
Fidélité
Services
Help
//Configuration
board[0]=new CommandeEtatCompte(courant)
board[1]=new CommandeEtatEpargne(epargne
board[2]=new CommandeEpargner(epargne));
//Invocation
board[1].execute();
date
libellé
1/04/10
solde
30/3/2010
virement régulier
28/2/2010
virement régulier
10/2/2010
virement
30/1/2010
virement régulier
1/1/2010
interets
70,45
31/12/2009
solde
1122,00
Board
Compte courant
Epargne
Epargner
configurer
configurer
configurer
configurer
Jean La Fourmi
Jeudi 1 avril 2010
Banque La Fontaine

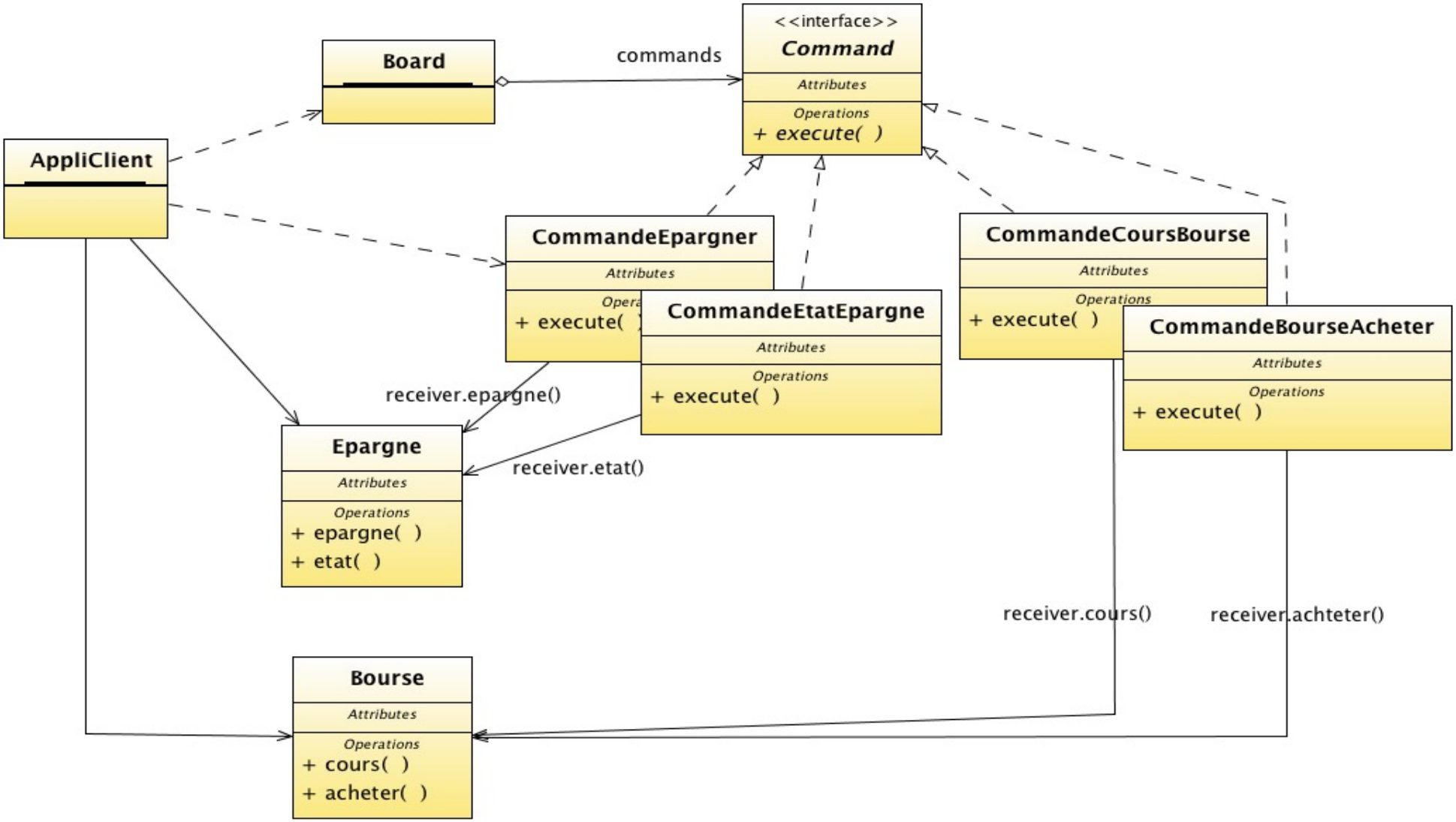
Comptes | Prets | Epargne | Carte | Chéquier | Bourse | Fidélité | Services | Help



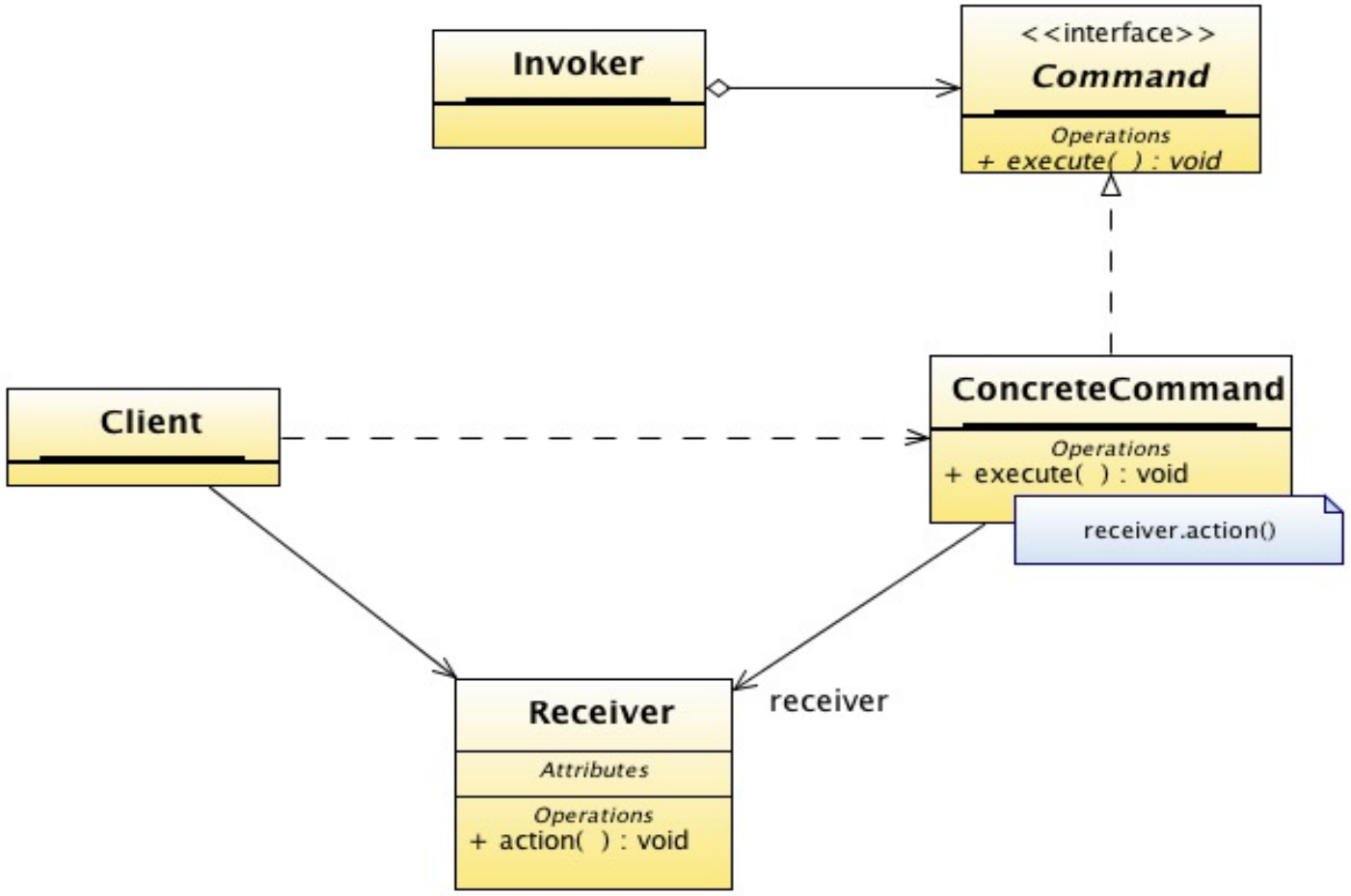

```
//Configuration
board[0]=new CommandeEtat(courant));
board[1]=new CommandeCours(bourse));
board[2]=new CommandeAcheter(bourse));
board[2]=new CommandeVendre(bourse));

//Invocation
board[1].execute();
```

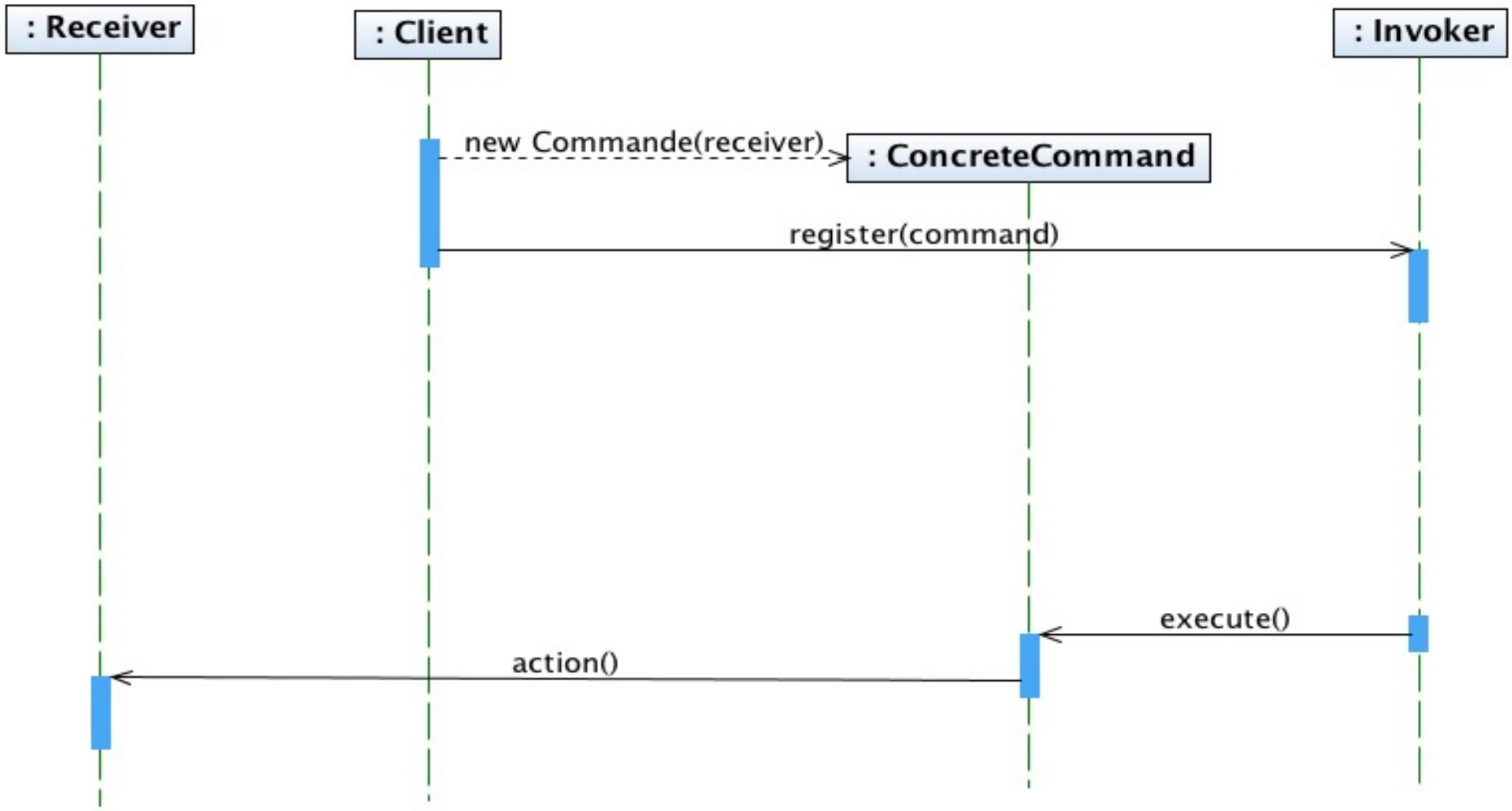
Board
Compte courant
Bourse
Acheter
Vendre
configurer
configurer
configurer

Jean La Cigale | Jeudi 1 avril 2010 | Banque La Fontaine


Board
commands
<<interface>>
Command
Attributes
Operations
+ execute( )
AppliClient
CommandeEpargner
Attributes
+ execute( )
CommandeEtatEpargne
Attributes
Operations
+ execute( )
CommandeCoursBourse
Attributes
Operations
+ execute( )
CommandeBourseAcheter
Attributes
Operations
+ execute( )
receiver.epargne()
receiver.etat()
Epargne
Attributes
Operations
+ epargne( )
+ etat( )
receiver.cours()
receiver.achteter()
Bourse
Attributes
Operations
+ cours( )
+ acheter( )

Pattern Commande: structure



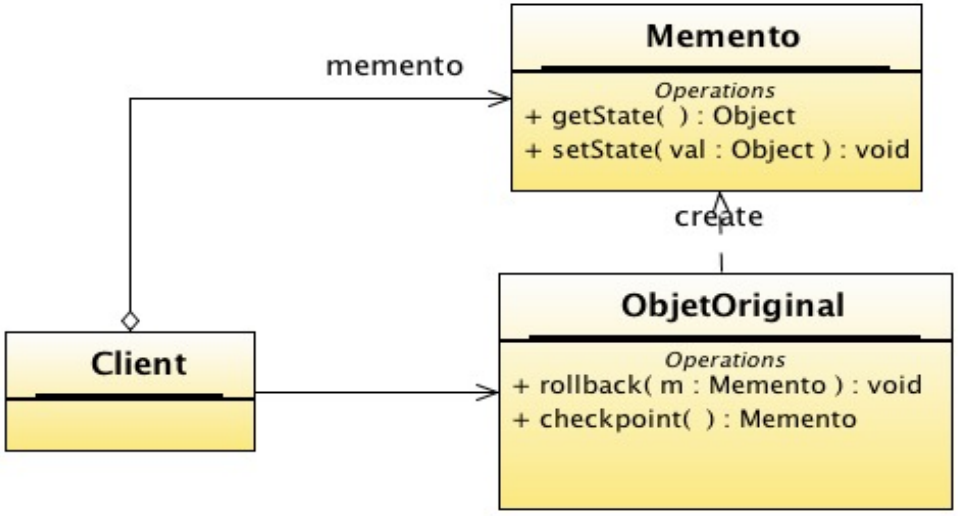

: Receiver
: Client
: Invoker
new Commande(receiver)
: ConcreteCommand
register(command)
execute()
action()

Pattern Commande: utilisations

- Commande découple l'objet qui invoque l'opération de l'objet qui l'exécute(ra).
- Il rend les applications plus malléables et configurables à l'exécution: « scripting »
- Il offre certaines capacités de programmation à l'utilisateur
- Il offre une certaine forme d'intervention sur l'exécution.
- Les commandes sont des objets et sont manipulables comme tels:
  - peuvent être rangées dans des structures, mis en attente (« logger » les changements), ...

Pattern Commande: utilisations

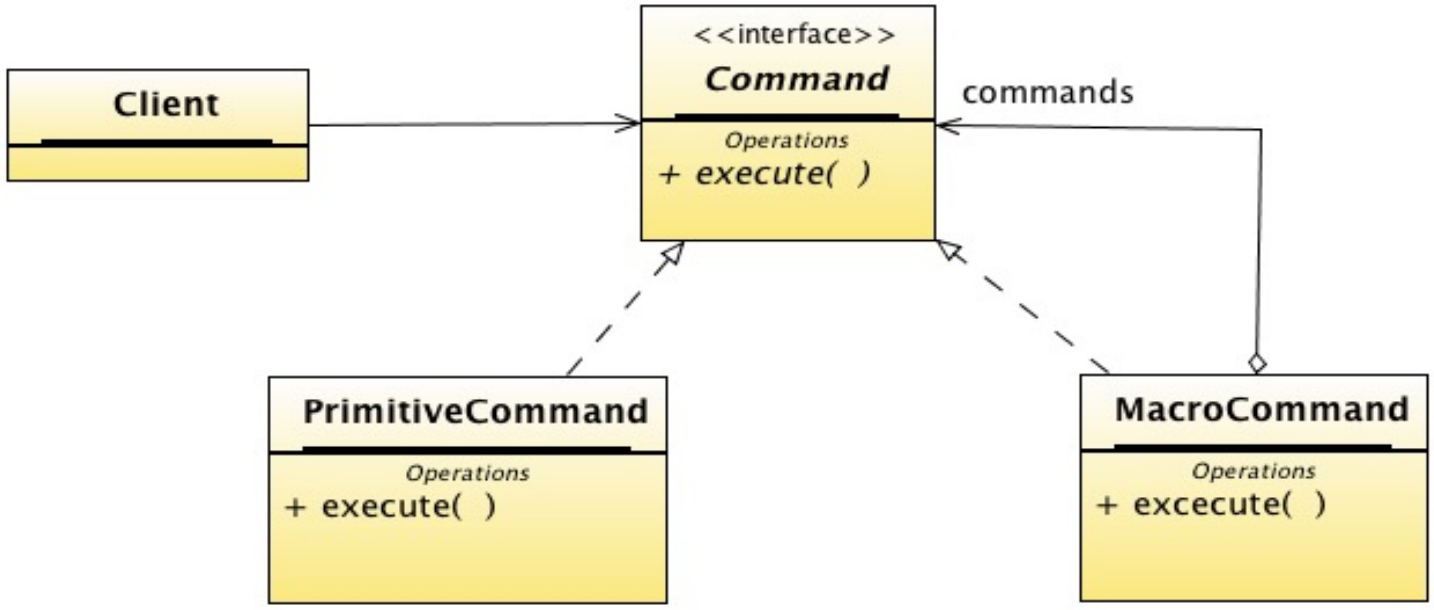
- annulées ou défaites
  - undo dans les interfaces graphiques
  - il peut être nécessaire d'ajouter à l'interface Command l'opération `undo()` inverse de `execute()`
  - il peut être nécessaire également de mémoriser l'état de l'objet: Pattern Memento



Pattern Commande: utilisations

- macro-commandes (« scripts »): Composite

```
MacroCommand:execute()
    for(Command c : commands) c.execute()
```



Commande: patterns apparentés

- Adaptateur
  - est un pattern de structure, il résout des problèmes d'ajustement de code de classes (il est de niveau Classe)
  - alors que Commande est un pattern comportemental, dédié aux questions de communication entre objets et leur contrôle à l'exécution (il est de niveau Objet)

- Décorateur
  - est aussi un pattern de structure mais de niveau Objet, il concerne l'ajout de nouvelles fonctionalités et non le contrôle de leur exécution (pas de gestion d'état).

Commande

- « call-backs »
  - pointeurs de fonctions C/C++
  - « clotures » (blocs Smalltalk, Lambda Lisp, Groovy, Ruby)
  - d'où objets « foncteurs »
    - Comparators, ActionListeners,...

- commande vs. « call-backs » :
  - l'intention de Commande est d'encapsuler (« réifier ») un appel de méthode sur un objet (avec état)
  - alors qu'un foncteur est la réification d'une fonction (sans état)

Commande: utilisations

- Utilisations remarquables
  - « scripting » d'applications
  - environnement de programmation visuelle (beans)
  - configurateurs dynamiques d'applications (plugins et widgets):
    - dashboard, tableaux de bord
    - espaces clients applicatifs, CMS, espaces de travail dynamiques (portails applicatifs et collaboratifs).
    - configurables par l'utilisateur

Autres Design Patterns

- Médiateur
  - centralise les collaborations
  - cf. Fabrique et Observateur
- Poids Mouche (Flyweight)
  - réduire la prolifération de petits objets en les partageant (quand c'est possible)
  - cf. Composite
- Chaine de responsabilités
  - enchainements de traitements d'une même requête par délégations successives
  - cf. Adaptateur
- Pont (Bridge)
  - découpler objets d'abstraction et objets d'implémentation
  - cf. Fabrique Abstraite
- Proxy
  - gérer l'accès à un objet par un mandataire (distant ou de contrôle)

Design Patterns

En guise de conclusion...

En guise de conclusion...

- On n'est pas « obligé » d'utiliser des DP!
  - privilégier toujours la simplicité et les principes fondamentaux
  - se concentrer sur la conception, les patterns ne doivent que se « révéler » naturellement (effet « ah ah !»)
  - ne pas tordre ou introduire un pattern de force
    - s'assurer que les conditions d'application du pattern sont bien respectées
    - voir si le problème n'est pas la combinaison de 2 (n) patterns en isolant les problèmes (répondant à leurs intentions)
  - les patterns ne se justifient souvent que quand il existe un problème d'extensibilité, ou quand ce problème apparaît (ré-ingénierie)
    - un pattern peut introduire de la complexité parasite (code exagérément élaboré par rapport au problème réel)
    - souvent, introduction de classes, objets, couches supplémentaires
    - problème de complexité induite, efficacité

En guise de conclusion...

- Les DP ne se conçoivent pas!
- Rappel
  - les DP ne sont pas des créations de toute pièce
  - sont la synthèse de solutions, de longues expériences, réelles et *récurrentes*
  - investissement lourd
- la plupart des concepteurs utilisent des patterns et n'en écrivent pas...

En guise de conclusion...

- ... si toutefois
  - domaine particulier
  - solution récurrente
  - réellement originale
  - non redondante avec d'autres patterns
    - d'où l'importance de bien connaître les DP existants
    - souvent que des variantes d'un DP existant
  - effectivement utilisé et éprouvé plusieurs fois (par d'autres!)
- méthodologie
  - « remplir » le format GOF
  - itérer...

En guise de conclusion...

- Les patterns se généralisent à d'autres niveaux que la conception
  - patterns d'architecture
    - vocabulaire: composants, flux, sites, infrastructures physiques, topologie
  - patterns de programmation (plutôt appelés idiomes)
    - vocabulaire : construction programmatique, mécanismes
    - souvent propres à un langage
  - patterns d'analyse et de process
    - vocabulaire: concepts du domaine
    - problèmes récurrents d'un domaine
    - gestion, « Business Patterns », ...
  - patterns d'organisation
    - problèmes de management, GRH, schémas d'évènement, patterns de réunions, ...
  - patterns de modèles en IDM...

Références

- « Design Patterns », E.Gamma, R.Helm, R.Johnson, J.Vlissides (GOF, Gang Of Four), 1995, Addison-Wesley Ed.
- « Thinking in Patterns with Java», Bruce Eckel, 2003, ebook: http//www.mindview.net/Books/TIPatterns
- « Head First Design Patterns », E.Freeman & al., 2005, O'Reilly Ed.
- « Object-Oriented Reengineering Patterns », S.Demeyer, S.Ducasse, O.Nierstrasz, 2008, Square Bracket Associates Ed. (Suisse) http://www.iam.unibe.ch/~scg/OORP
- « Design Patterns, les 23 modèles de conception » (version UML, Java ou C#). L.Debrauwer, 2009, ENI Editions
- Repositories de patterns

  http://hillside.net/patterns

  http://www.vincehuston.org/dp : C++

  http://www.javacamp.org/designPattern : en Java

  http://java.sun.com/blueprints/corej2eepatterns/Patterns : J2EE