Design Patterns

Table des matières

I - Cadre architectural (UML , Prog. Objet)

1. Avant propos (cadrage , problématique , ...)

Un **design pattern** est un **modèle de conception** (récurrent et réutilisable) qui permet de **bien structurer une application informatique**.

Il est très important de distinguer les 2 sortes suivantes de "design patterns":

- **DP d'architecture** (ayant un impact fort sur la structure d'ensemble)
- **DP d'implémentation interne** (éléments structurels cachés à portée limitée)

Les "design patterns" d'architecture visent essentiellement à obtenir une **architecture modulaire** et à **limiter la complexité de l'ensemble** (en *simplifiant* l'accès à certaines fonctionnalités via des "*façades*" par exemple).

Certains "design patterns" tels que le "composite" sont à l'inverse des "***astuces techniques***" (engendrant une petite complexité complémentaire) et qui doivent idéalement être à portée limitée (implémentations internes cachées).

GOF

GRASP

Design Principles

IOC/injections de dépendances

MVC

DAO

DTO/VO

2. Cadre architectural & technique

Un **composant "orienté objet"** peut souvent être vu comme un **ensemble d'éléments qui sont fortement reliés entre eux** et qui **forment un tout cohérent et indissociable**.

Par exemple, un service métier est indissociable des structures des entités ou vues métiers qui sont passées en tant qu'arguments des méthodes ou bien en tant que valeur(s) de retour.
Concrètement, la classe (plutôt orientée traitement) "*GestionComptes*" et la classe (plutôt orientée données) "*CompteDto*" forment globalement un composant de type "Service Métier" .

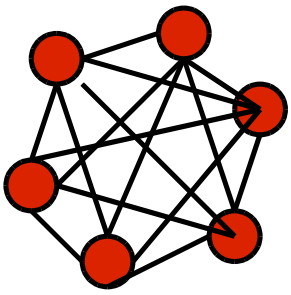
Des sous composants peuvent être imbriqués dans de "gros" composants.
On peut éventuellement considérer qu'un "DAO" (Data Access Object) est un sous composant vis à vis d'un "module de services" .

---> Modéliser en UML les composants (leurs portées , leurs imbrications , ...) est donc extrêmement important pour se repérer dans l'architecture d'un système informatique.

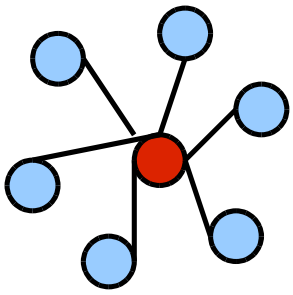
Couplage entre objets (idéalement faible)

Un ***objet élémentaire (tout seul)*** rend souvent des ***services assez limités***.

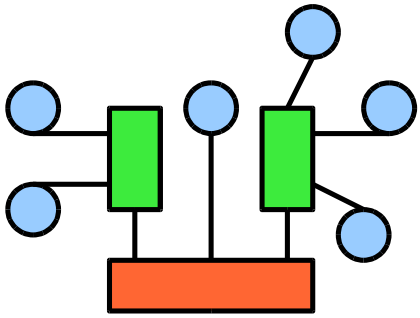
Un ***assemblage d'objets complémentaires*** rend globalement des ***services plus sophistiqués***. Cependant la complexité des liens entre les éléments peut éventuellement mener à un édifice précaire:



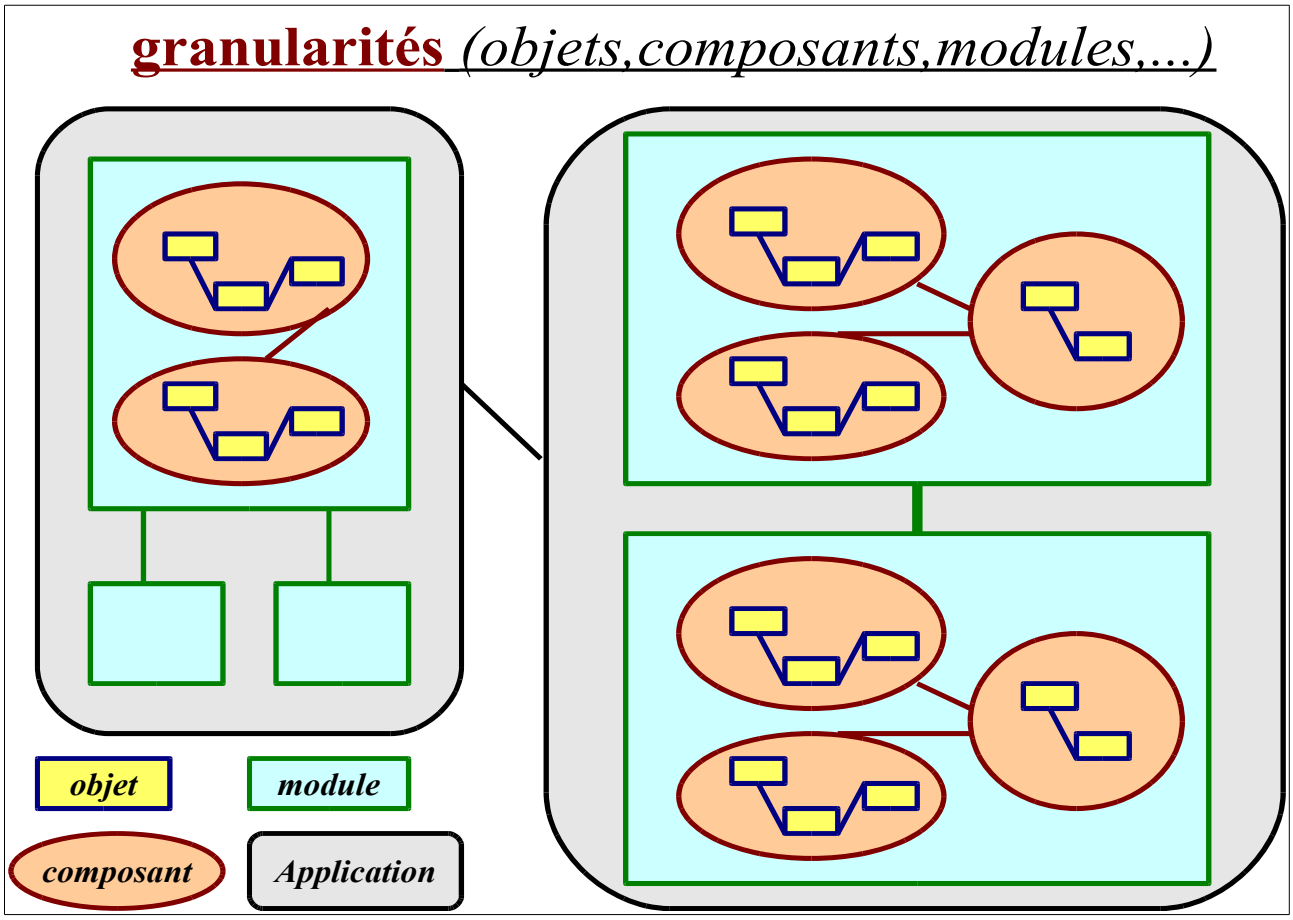
couplage trop fort
==> *complexe* ,
trop d'inter-relations
et de dépendances
==> *inextricable*.

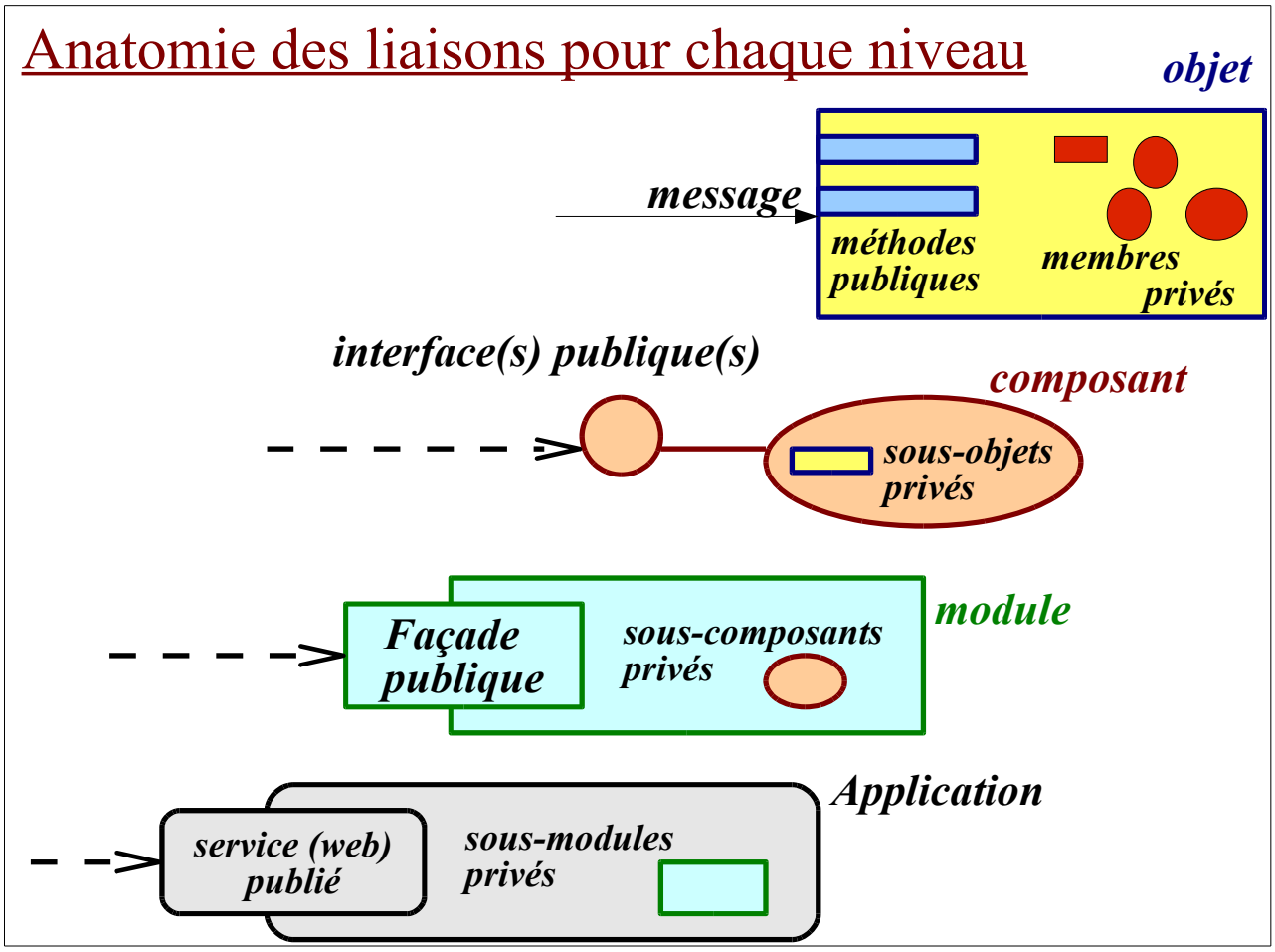


couplage faible
mais ***centralisé***
==> peu flexible
et point central
névralgique



couplage faible
et ***décentralisé***
==> simple ,
relativement flexible
et plus robuste


granularités (objets,composants,modules,...)
objet
module
composant
Application

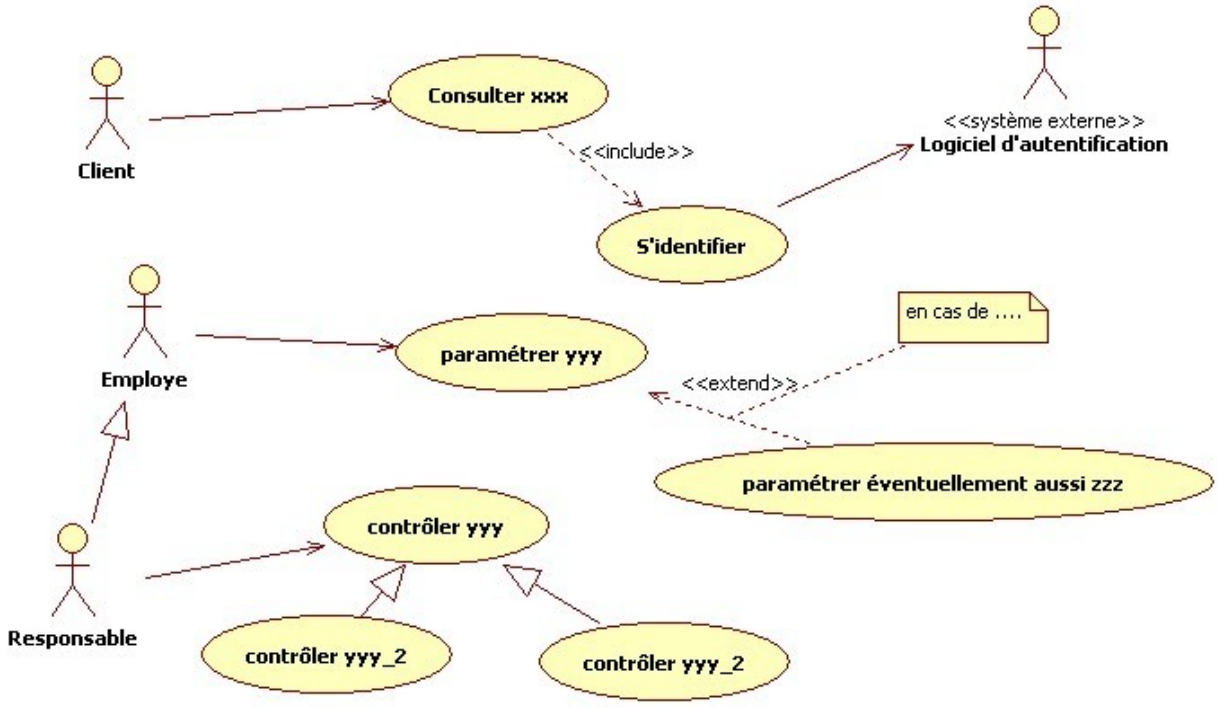

Anatomie des liaisons pour chaque niveau
objet
message
méthodes publiques
membres privés
interface(s) publique(s)
composant
sous-objets privés
Façade publique
sous-composants privés
module
Application
service (web) publié
sous-modules privés

3. Modélisation UML efficace

3.1. Rappels rapides UML

La notion de "**cas d'utilisation**" correspond à la fois à:

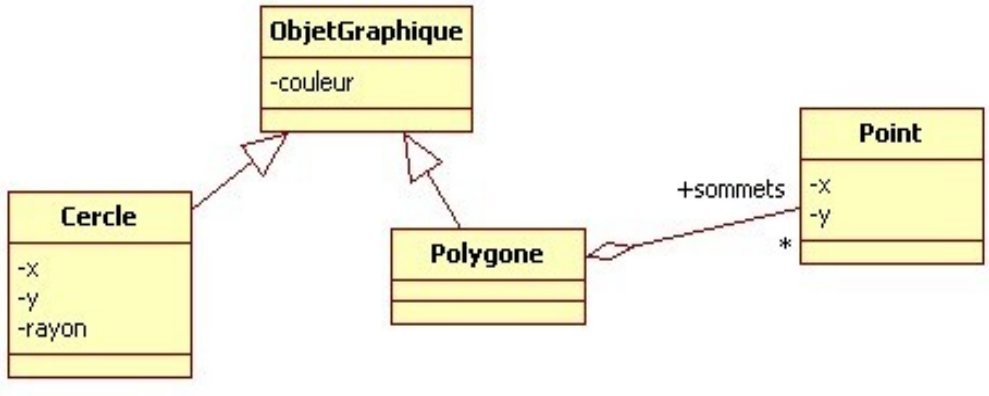
- **une (grande) fonctionnalité du système à développer**
- **un objectif (ou sous objectif) utilisateur**
- **un cas d'utilisation du système se manifestant par au moins une interaction avec un acteur extérieur .**



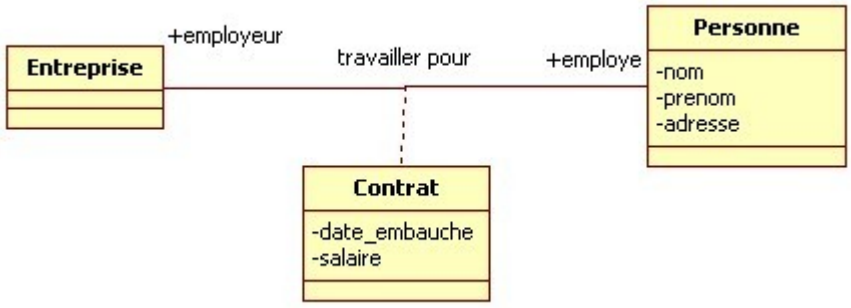
NB:

- Au sein d'un diagramme UML de "uses cases" comme le précédent , il est conseillé de **nommer les cas d'utilisation** par des termes du genre "***verbe_complément_d'objet_direct***". Les ***compléments d'objets*** permettront d'associer ultérieurement les cas d'utilisations aux ***entités métiers*** et aux ***services métiers.***
- NB: **<<*include*>>** pour sous tâche indispensables/obligatoires , **<<*extends*>>** pour tâches d'extensions facultatives.

Diagramme de classes:



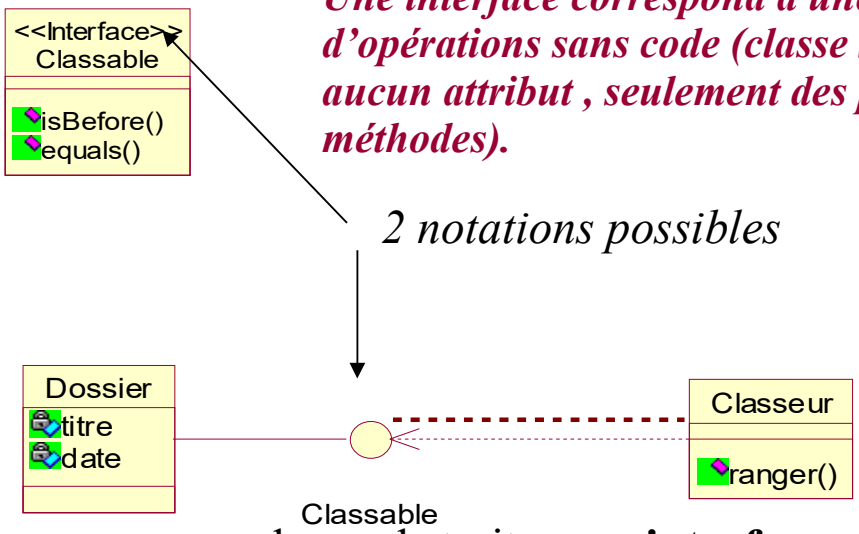
Rappel : le losange est placé du coté "agrégat / conteneur"



3.2. Interfaces

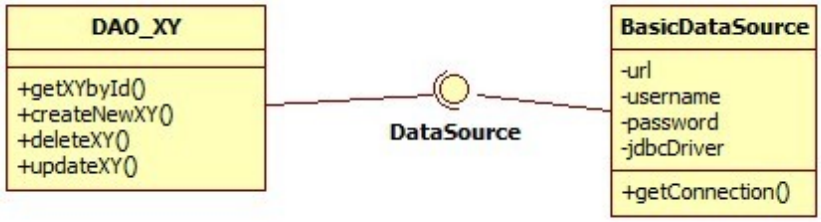
Interface

Une interface correspond à une collection d'opérations sans code (classe spéciale sans aucun attribut , seulement des prototypes de méthodes).

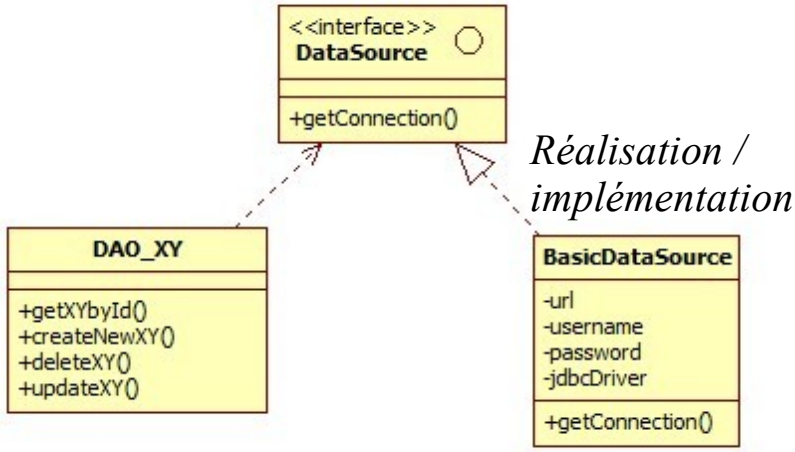


Tout comme une classe abstraite, une **interface** correspond à un **type de données** (permettant de déclarer des références).

Notation Compacte (dépendance)



Notation développée (avec détails)

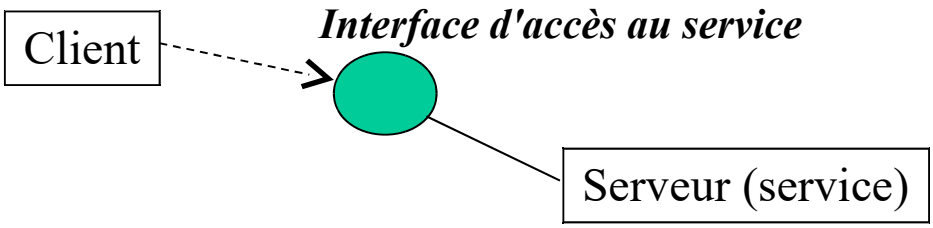


Réalisation / implémentation

Interface (contrat)

• Une **interface** peut être considérée comme un **contrat** car chaque classe qui choisira d'implémenter (réaliser) l'interface sera obligée de programmer à son niveau toutes les opérations décrites dans l'interface.

• Chacune de ces opérations devra être convenablement codée de façon à rendre le **service effectif** qu'un client est en droit d'attendre lorsqu'il appelle une des méthodes de l'interface.



3.3. Composants

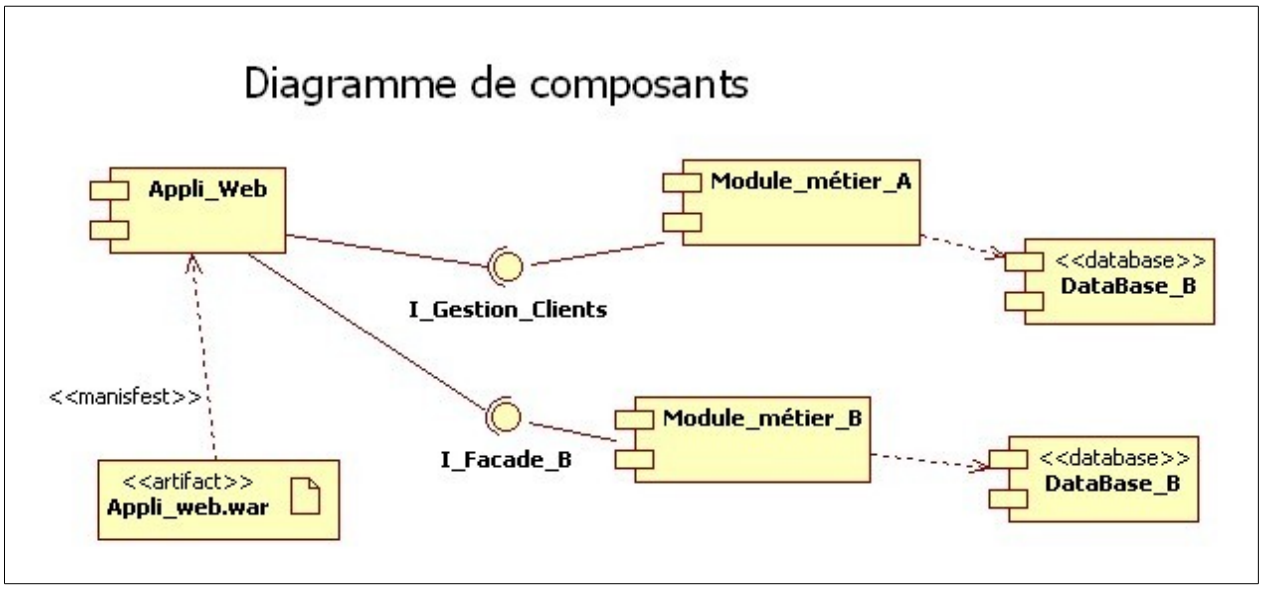
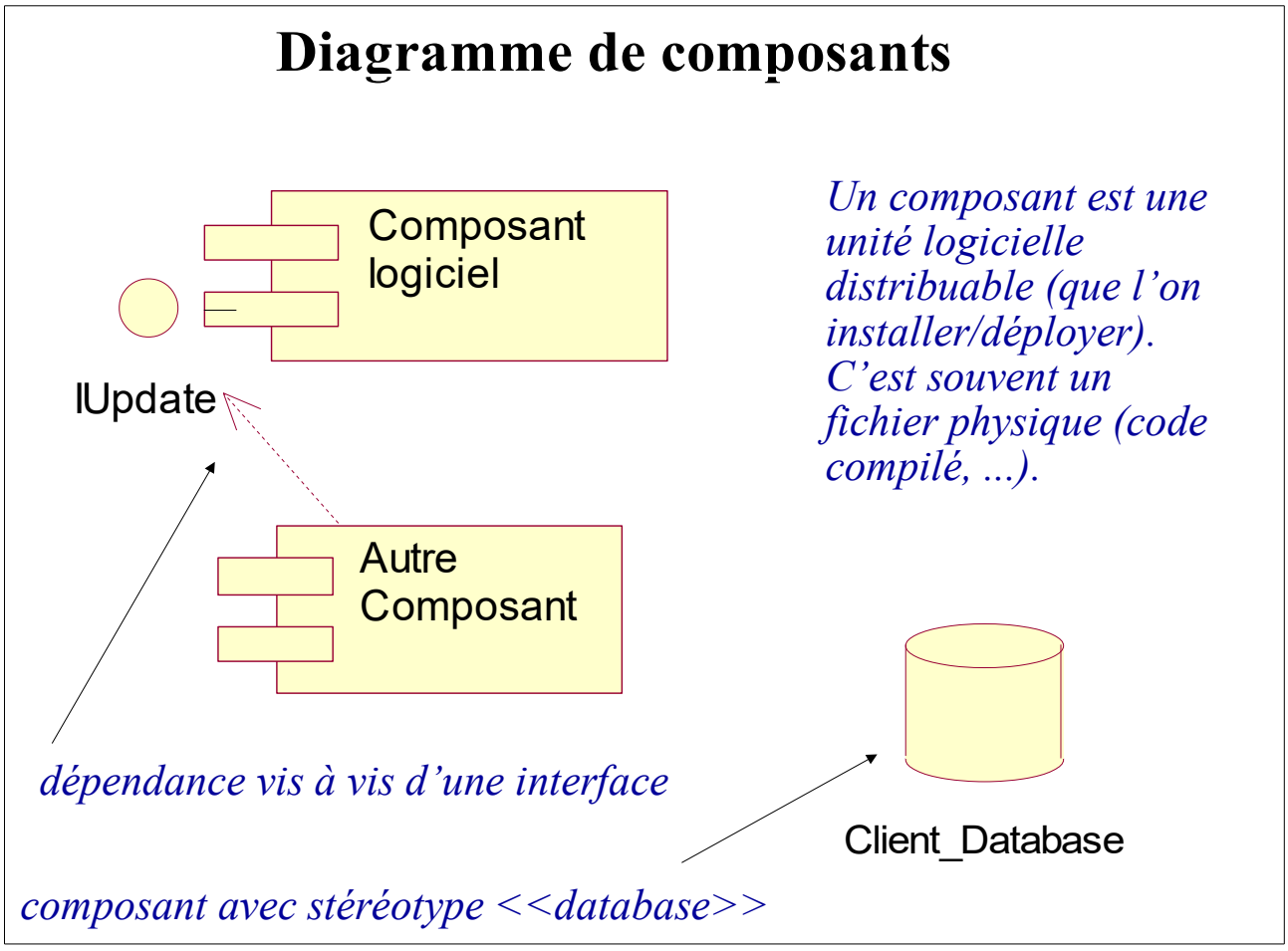
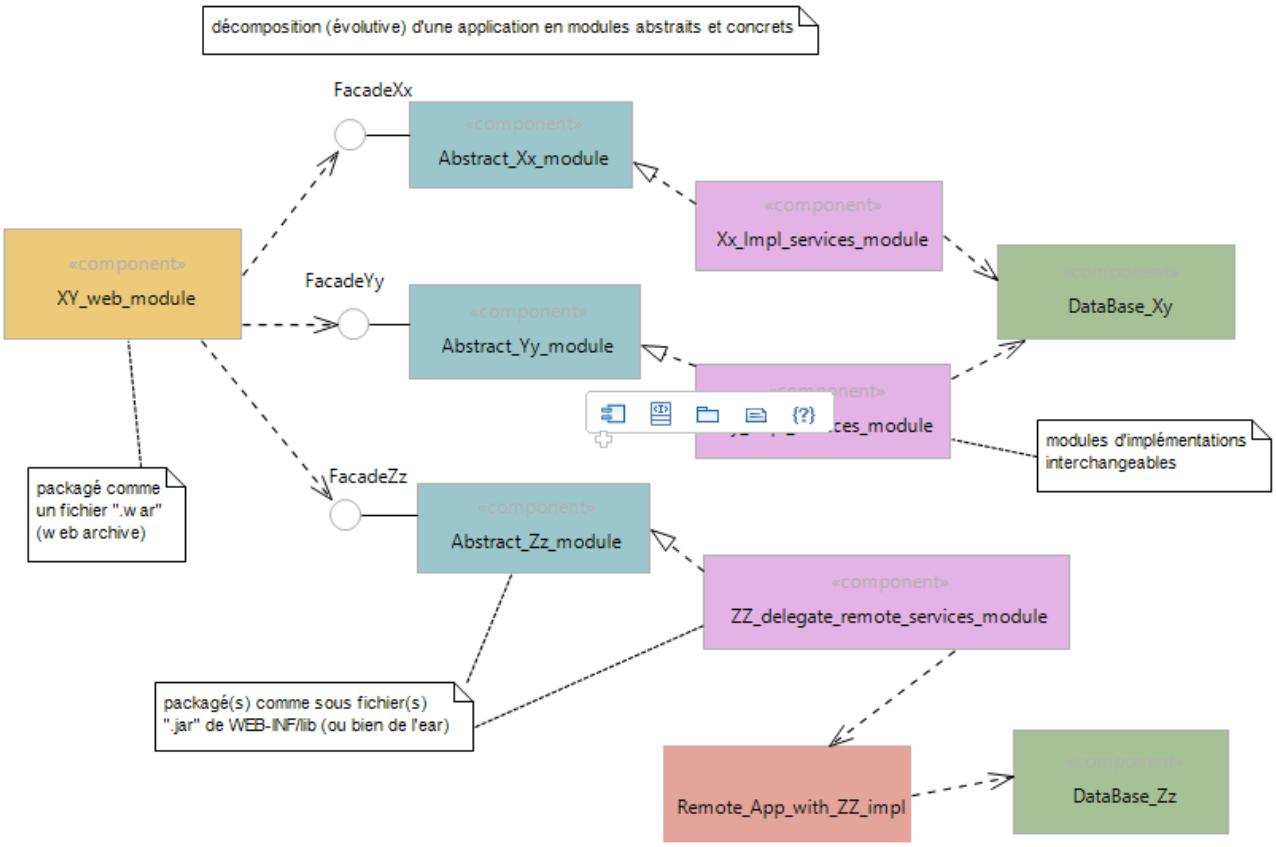


Diagramme de composants

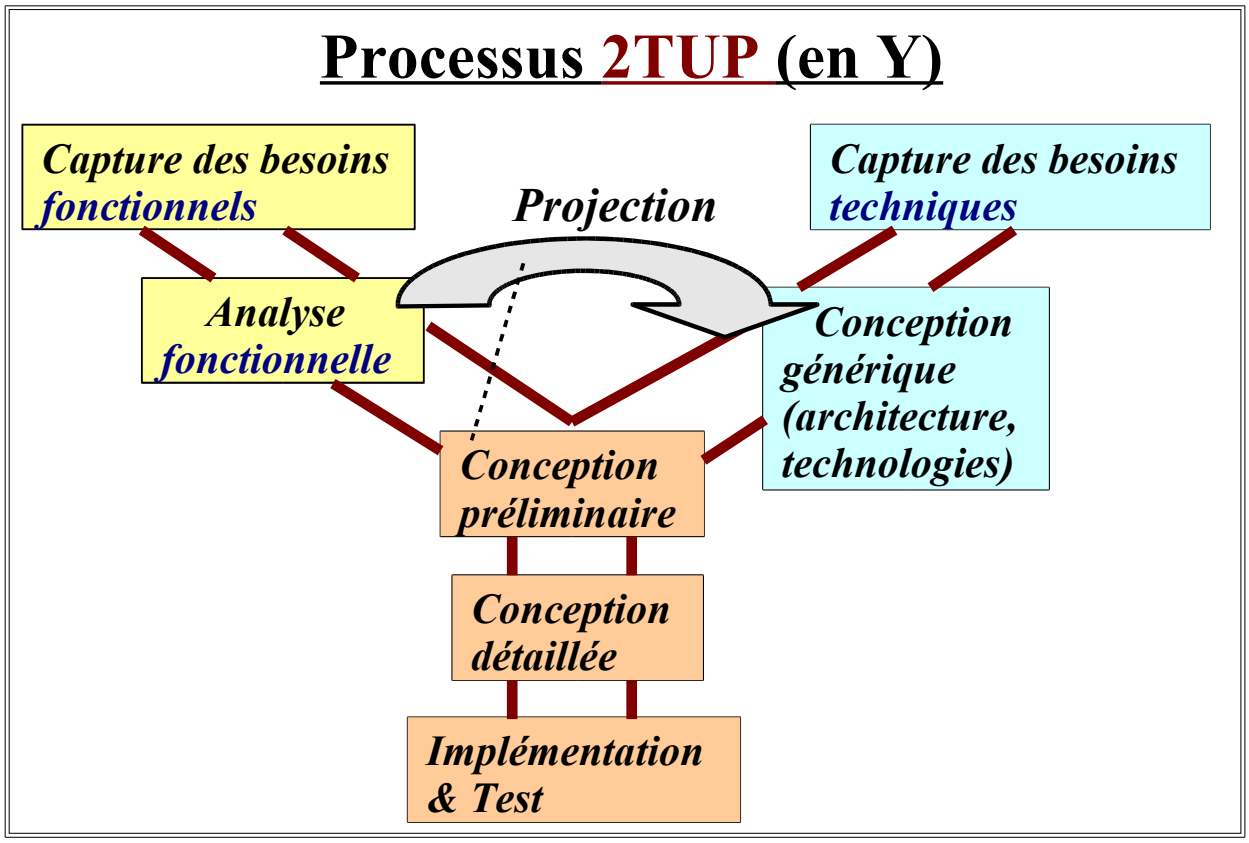


Un composant est une unité logicielle distribuable (que l'on installer/déployer). C'est souvent un fichier physique (code compilé, ...).


décomposition (évolutive) d'une application en modules abstraits et concrets
FacadeXx
«component»
Abstract_Xx_module
«component»
Xx_Impl_services_module
«component»
DataBase_Xy
«component»
XY_web_module
FacadeYy
«component»
Abstract_Yy_module
modules d'implémentations interchangeables
packagé comme un fichier ".war" (web archive)
FacadeZz
«component»
Abstract_Zz_module
«component»
ZZ_delegate_remote_services_module
packagé(s) comme sous fichier(s) ".jar" de WEB-INF/lib (ou bien de l'ear)
Remote_App_with_ZZ_impl
«component»
DataBase_Zz

4. Projection (2TUP / Y)

Il s'agit ici de **projeter** le résultat de l'analyse au sein d'une architecture logique et technique définie durant l'étape "architecture" (ou "conception générique").



Principal résultat de la conception préliminaire (projection):

n "packages fonctionnels" * m "couches/niveaux techniques"

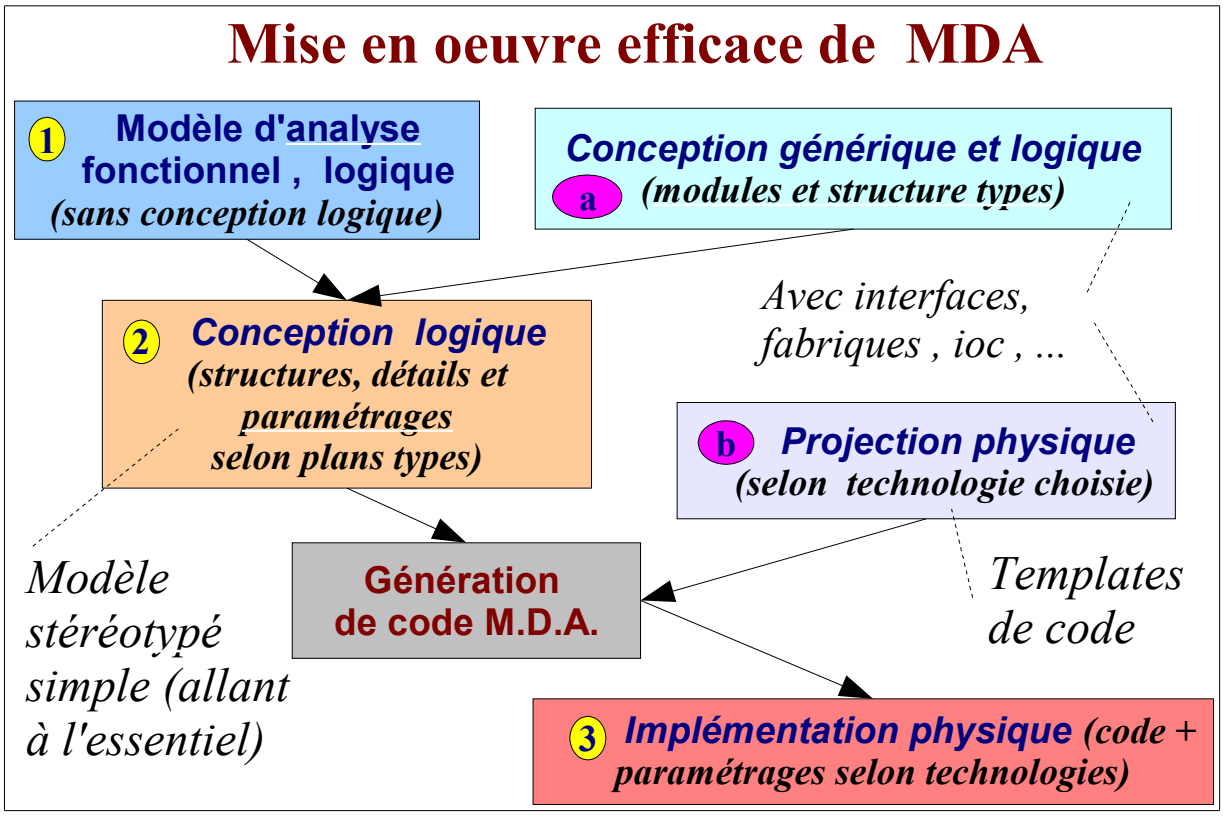
===> n*m packages dans le code à réaliser/produire:

- fr.xxx.yyy.AppliA.partieIHM.**web**
- fr.xxx.yyy.AppliA.*partieFonctionnelleA*.**service**
- fr.xxx.yyy.AppliA.*partieFonctionnelleA*.**entity**
- fr.xxx.yyy.AppliA.*partieFonctionnelleB*.**service**
- fr.xxx.yyy.AppliA.*partieFonctionnelleB*.**entity**
- ...

==> Il vaut mieux appliquer systématiquement certaines règles de projection (via MDA ou ....) pour être efficace/rapide .

4.1. MDA & Stéréotypages

Approche efficace et pragmatique de la conception prenant en compte MDA:



Concrètement :

1) reprise des diagrammes de classes fonctionnels (fruits de l'analyse)
a) Modélisation d'architecture générique en UML (ex : Architecture JEE avec services et DAO)
2) ajout de certains stéréotypes (ex : <<service>> , <<id>> , <<entity>> en vue de paramétrer la future génération de code
b) Templates "accéléo_m2t" (modèles de code java à générer selon stéréotypes UML)
3) amélioration manuelle du code automatiquement généré par le plugin eclipse "accéléo_m2t" .

Point clefs :

- **stéréotype UML** = paramétrage UML interprété (par un développeur ou par un générateur automatique MDA) lors de la génération de code
- **annotations (java, php ou c#)** = paramétrage généralement interprété par un framework au runtime (lors de l'exécution du code)

Certains stéréotypes peuvent quelquefois être retranscris en annotations proches.
Exemples :
<<id>> ---> @Id
<<service>> ---> @WebService ou …
<<entity>> ---> @Entity

II - Design Patterns (présentation / principes)

1. Présentation du concept de Design Pattern

Notion de "Design Pattern"

Un **Design Pattern** [DP] est un **modèle de conception réutilisable** (dont on peut s'inspirer de multiples fois sans pour autant toujours coder les choses de la même façon).

C'est concrètement un document d'une dizaine de pages comprenant:

Description d'un problème récurrent (contexte)

Modélisation (ex: ***UML***) d'une solution de conception générique

Description des avantages/inconvénients

Exemples de code , Variantes

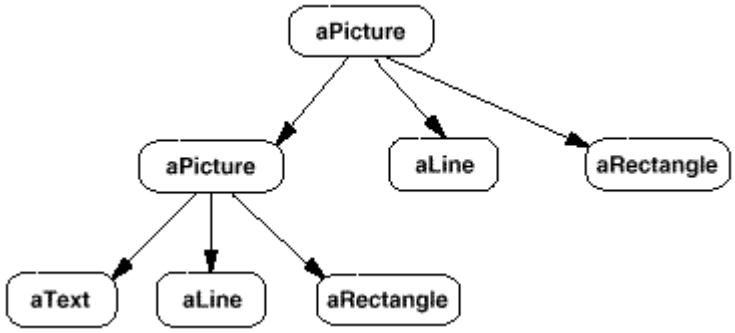
NB:

L'inventeur du concept de "Design Pattern" (**Christopher Alexander**) ne travaillait pas dans le domaine de l'informatique.
Selon lui :

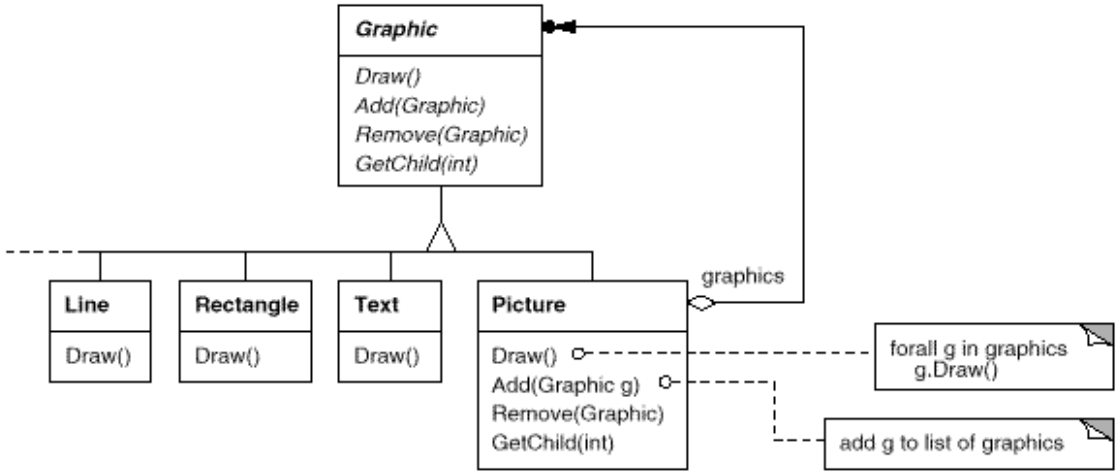
<< ***Chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière*** >> .

1.1. Exemple (modèle composite)

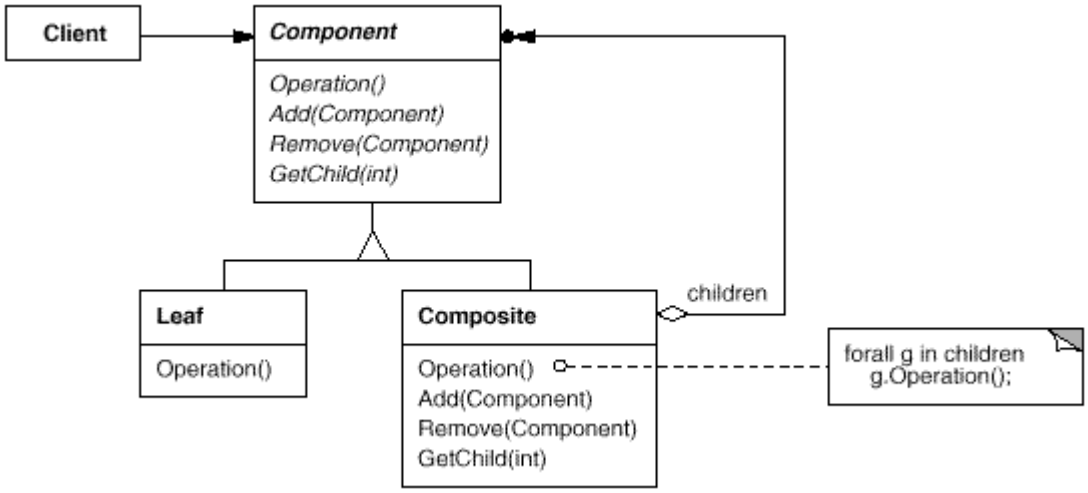
Problématique: Comment géré de façon élégante un niveau quelconque d'imbrication ?



Solution:



structure générale de la solution:



2. Importance du contexte et anti-patterns

2.1. contexte d'un design pattern et objectif visé

Attention: un Design Pattern n'est généralement intéressant qu'au sein d'un contexte particulier:

- Certaines conditions doivent généralement être vérifiées (volume , ...) .
- But à atteindre ?
- ....

L'utilisation inadéquate (non réfléchie ou systématique) d'un design pattern est quelquefois contre-productive ==> ceci constitue un des "anti-pattern" (choses à ne pas faire , modélisation d'un dis-fonctionnement ou problème potentiel, ...).

2.2. anti-pattern

	Design Pattern	**Anti Pattern**
Pb Récurrent / Exemple	Objectif visé revenant souvent	Problème (défaut) apparaissant souvent
Modélisation / Généralisation	Modèle d'une chose à reproduire	Modèle d'une chose à éviter
ex de code / variantes	ex de code dont on peut s'inspirer + variantes	ex de mauvais code (ou mauvaise technique) à prohiber + alternatives conseillées

Quelques exemples d'anti-patterns:

- **trop de "copier/coller" dans le code**
  ==> reproduction/prolifération de choses à faire évoluer en parallèle ou d'éventuels défauts
  ==> maintenance difficile
  ==> une factorisation est souvent possible (via une petite restructuration du code)
- **trop de "if" ou "switch/case" sans polymorphisme**
- **code "mort" (jamais utilisé)**

...

3. Principaux "design patterns"

3.1. Les grandes séries de "Design Patterns" orientés "objet"

Les grandes séries de "design patterns"

• ***GRASP*** *[* ***G****eneral* ***R****esponsability* ***A****ssignment* ***S****oftware* ***P****atterns]*
==> bonnes patiques (peu formalisées mais grandes lignes directrices) , simples à comprendre et intuitifs ==> patterns adaptés dès la fin de l'analyse (non techniques)

• ***<<Design Principles>>*** *(Robert MARTIN , Bertrand MEYER)*
==> grands principes objets assez formalisés (règles à respecter) . Essentiellement liée à la structure générale des modules , cette série de patterns est tout à fait adaptée à la conception préliminaire.

• ***GOF*** *(Gang Of Four)*
==> série assez technique de "design patterns" (proches du code)
==> adapté pour la conception (préliminaire et détaillée).

• ...

3.2. "Design Patterns" du GOF

Principaux "Design Pattern" *(G.O.F.)*

	Rôle		
	Créateur	**Structurel**	**Comportemental**
Classe / Statique	**Fabrication**	**Adaptateur (stat.)**	Interprète
			Patron de Méthode
Objet / Dynamique	**Fabrique Abstraite**	**Adaptateur (dyn.)**	Chaîne de responsabilité
	Monteur	**Composite**	Commande
	Prototype	**Décorateur**	Itérateur
	Singleton	Façade	Médiateur
		Poids Mouche	Mémento
		Pont	Observateur
		Procuration	État
			Stratégie
			Visiteur

G.O.F. ==> **Gang Of For** (le gang des 4 personnes qui ont lancé les "Design Patterns" dans le monde de la conception orientée objet) ==> Thèse & Livre :

Livre (de référence) fortement conseillé :

DESIGN PATTERNS
Catalogue de modèles de conception réutilisables [traduction francaise de Jean-Marie Lasvergères.] / **Vuibert / Addison Wesley**

Auteurs (les 4):
Erich Gamma, ***Richard Helm,*** ***Ralph Johnson,*** ***John Vlissides***

3.3. Autres ensembles de designs patterns

- **GRASP**
- **Object Principles (Meyer & Martin)**

4. Différences entre "Design Pattern" et "Framework"

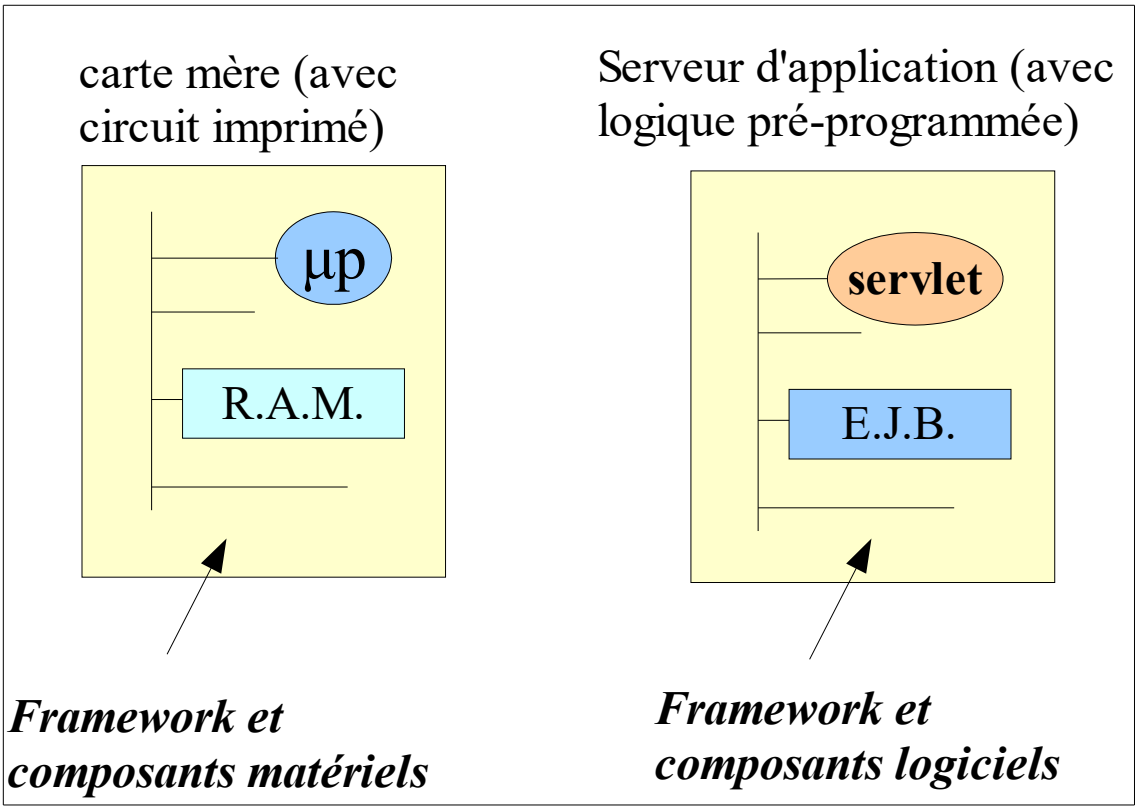
Notion de "Framework"

Un "Design Pattern" est un modèle qui reste à programmer (en fonction du contexte).

Un "**Framework**" est *en grande partie déjà programmé*: il s'agit d'un **schéma applicatif "pré-cablé" prêt à recevoir des composants logiciels**.

Analogie classique:

Carte mère avec circuit pré-cablé pour recevoir des composants matériels (CPU, Ram, carte PCI).
- Idée d'industrialisation.



III - Patterns essentiels - GOF

1. Liste des principaux "design patterns" du GOF

Design Pattern	Description
Abstract factory (fabrique abstraite)	Instanciation indirecte de familles de produits. (ex: Look & Feel "Windows" ou "X11/Motif" Java / SWING). En changeant de fabrique on génère des objets différents (look1,look2, ...) qui ont néanmoins les mêmes fonctionnalités (même interfaces).
Adapter (adaptateur)	Intermédiaire permettant de convertir l'interface (figée) d'une classe existante avec celle attendue par un client.
Bridge (pont)	Correspondance (pont) entre 2 hiérarchies de classes (ex: XXX - XXXImpl de AWT). Séparation de l'abstraction et de la représentation/implémentation.
Builder (monteur)	Fabriquer des parties via des objets monteurs dirigés par un objet directeur. Séparer la construction de la représentation.
Chain of responsability (Chaîne de responsabilité)	Requête avec: 1. 1 émetteur. 2. des récepteurs chaînés entre eux (hop, hop ,hop jusqu'à traitement effectif [+ éventuelle valeur ajoutée sur les intermédiaires aux responsabilités bien définies]). ==> idée principale = déléguer ce qui n'est pas de notre responsabilité .
Command (Commande)	Encapsulation d'une requête dans une méthode (ex: execute()) d'un objet commande. ==> Liste de commandes ==> permet undo/redo et le déclenchement d'une même commande depuis plusieurs parties de l'IHM (menu, toolbar, bouton poussoir).
Composite	Composition récursive à niveau de profondeur quelconque (héritage + composition combinés)
Decorator (Décorateur / Enveloppe transparente)	Enveloppe transparente rajoutant de nouvelles fonctionnalités.
Facade	Interface unifiée pour la totalité d'un sous système / acceuil
Factory method (méthode de fabrication)	Création indirecte d'instance déléguée au niveau d'une méthode de type "create()" [beaucoup de variantes]
Flyweight (poids mouche)	Comment gérer plein de petits objets ? ==> petit objet partagé (avec une partie interne intrinsèque) et avec des méthodes comportant une référence sur un contexte (partie externe à la charge du client).
Interpreter (interpréteur)	Grammaire (phrases / expressions à interpréter , exemple: 2x+y+5*z/3). ==> construction d'un arbre syntaxique dont les nœuds sont des objets. Fonction interpréter() récursive et polymorphe.
Iterator (itérateur)	Traverser (balayer/parcourir) une collection (liste/tableau) sans avoir à connaître la structure interne et de façon à accéder à chacun des éléments.
Mediator (médiateur)	Intermédiaire commun à un paquet d'objet (coordonnant les interactions)==> pour réduire le couplage.
Mémento	Petit objet secondaire (avec interfaces large et fine) permettant de mémoriser l'état d'un objet principal de façon à restaurer les valeurs

	de celui-ci plus tard (Respect de l'encapsulation, permet un undo).
Observer (observateur)	Définit une dépendance de un à plusieurs 1 Mise à jour / des Notifications (callback) 1 Diffusion / des Souscripteurs
Prototype	Créer de nouveaux objets à partir d'un exemplaire déjà instancié (clonage). (ex: Palette d'objets graphiques à cloner).
Proxy (proximité)	Fournir un substitut pour accéder à un objet.
Singleton	Une seule instance pour une certaine classe (Méthode statique pour créer (ou obtenir) cette instance.
State (état)	Différents sous-objets ayant une interface commune codent différents comportements d'un objet englobant ayant plusieurs états (comme s'il changeait de classe).
Strategy (stratégie)	Classe abstraite d'algorithmes interchangeables pour un certain contexte. [externaliser une responsabilité avec variantes]
Template method (Patron de méthode)	Algorithme abstrait basé sur un même squelette (méthode d'une sur-classe) dont différentes sous tâches (parties) sont codées comme des méthodes polymorphes dans diverses sous classes.
Visitor (visiteur)	Sur un objet que l'on peut parcourir (ex : hiérarchie de noeuds d'un arbre ou liste) on va déclencher (via des **visiteurs actifs**) des opérations génériques durant une traversée. ==> objets visitables et visiteurs doivent être pensés et modélisés pour être compatibles.
...	

2. Design Patterns fondamentaux du GOF

2.1. Singleton

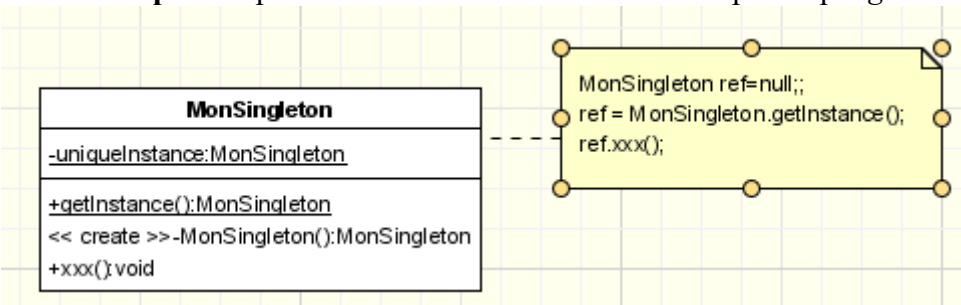
Une et une seule instance pour une classe donnée.

Solution classique:

- variable de classe (static) **uniqueInstance** =null
- méthode de classe (static) **getInstance()**

```
{ if (uniqueInstance == null) {
     uniqueInstance = new Xxx();
    }
  return uniqueInstance; // instance nouvellement ou anciennement créée .
}
```

- **constructeur privé** pour interdire des "***new***" directs sans passer par *getInstance*()



NB : il faudra penser à ajouter **"synchronized"** (ou autre) sur ***getInstance*()** dans un **contexte "multi-thread" .**

Avantages du singleton:

Etre sûr qu'une seule instance sera utilisée à un niveau donné permet:

- d'optimiser la mémoire
- de gérer un contexte (avec données partagées) à un endroit central bien précis (initialisation , lecture/écriture , ...)

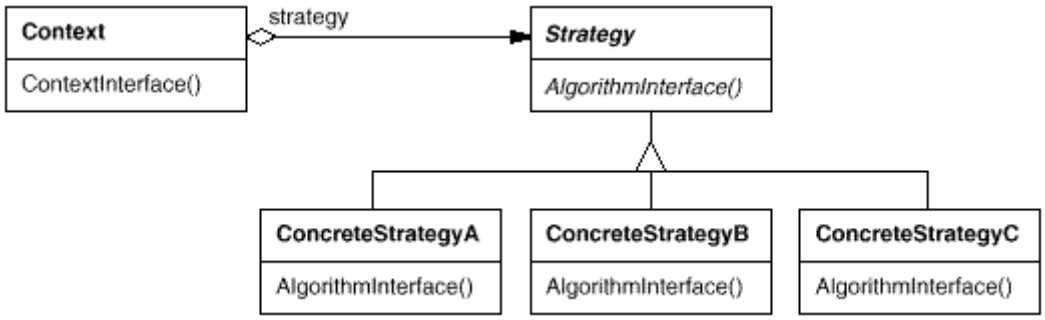
D'autre part, l'appel d'une méthode statique est très pratique pour récupérer l'instance unique depuis un endroit quelconque du programme.

Le Singleton est assez souvent utilisé sur les fabriques et les façades (une seule instance suffit souvent)

Eventuels inconvénients:

Effet "Kitchen Sink" (siphon) ==> Le singleton ramène à lui tous les appels "static" , ce qui peut poser quelques problèmes si les choses doivent évoluées (changement ? changement partiel ? , ...).

2.2. Stratégie

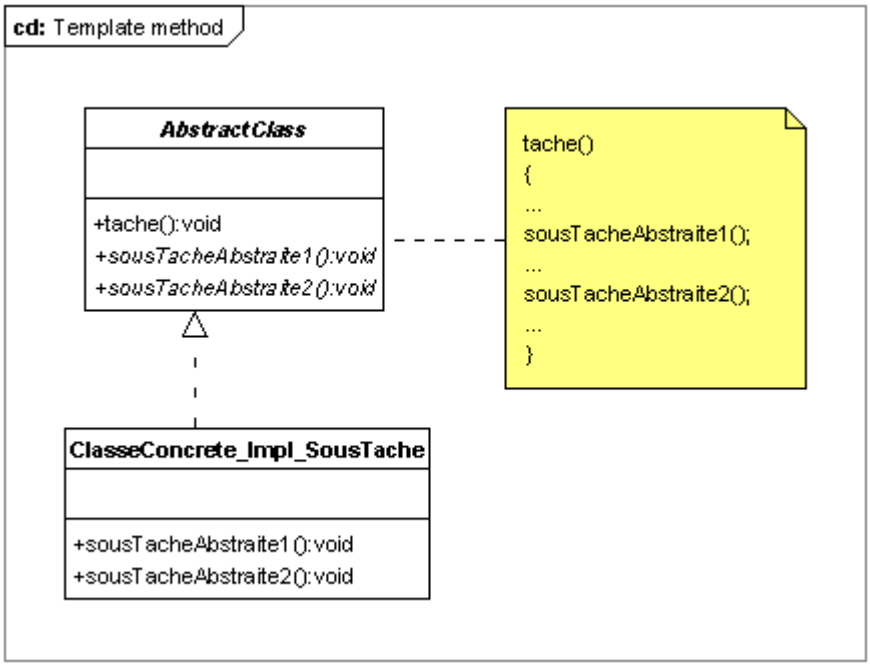


NB: Le design pattern "***stratégie"*** met non seulement l'accent sur le ***polymorphisme*** mais également sur le fait d'***externaliser une certaine responsabilité annexe vis à vis du contexte de départ***.
C'est généralement une très bonne idée favorisant nettement la **modularité** de l'ensemble.

NB: Le design pattern "***D.A.O.***" (Data Access Object) peut être vu comme un cas particulier de stratégie (une stratégie d'accès aux données)

2.3. Patron/modèle de méthode (Template method)

Factoriser ce qui est commun , différentier le spécifique .



Exemple:

// code commun de la tache au niveau de la classe abstraite:

void **dessinerAvecCouleur**(String couleur) {

choisirCouleur(couleur); *// code commun (factorisé au niveau de la classe abstraite)*

dessiner(); *// sous tache abstraite avec code différent pour ligne , rectangle , ...*

}

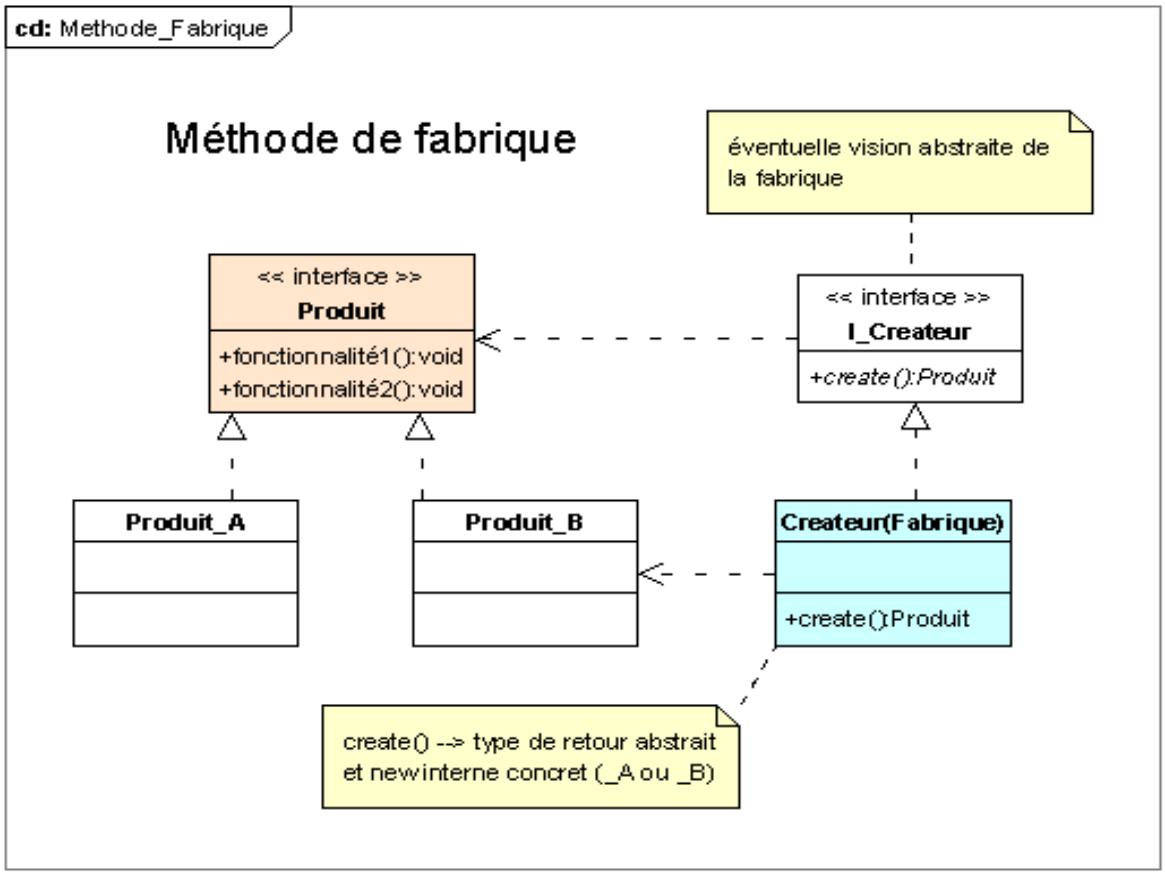
2.4. Méthode de fabrication (factory method)

Problème courant: comment masquer le type exact d'un objet à créer ainsi que les détails de son initialisation ?

Solution courante: Utiliser une méthode de fabrication sur un objet intermédiaire (souvent appelé "fabrique"). On évite ainsi une instanciation directe du genre xxx = new CXxx("v1","v2");

Ce code est alors caché au sein d'une méthode " .create() " éventuellement "static" .

Le code interne de ".create()" peut alors utiliser des mécanismes quelconques (ex: fichier de configuration) pour créer et initialiser un composant CXxxV1 ou CXxxV2 et le retourner ensuite de façon abstraite (type de retour de .create() = interface IXxx)



Variantes: le code interne de la méthode de fabrique (ex: create()) peut :

- instancier une nouvelle instance via new
- retourner un objet pré-construit et rangé dans un pool
- déclencher une instanciation générique via Class.forName("package.NomClasse").newIntance()
- ....

2.5. Facade

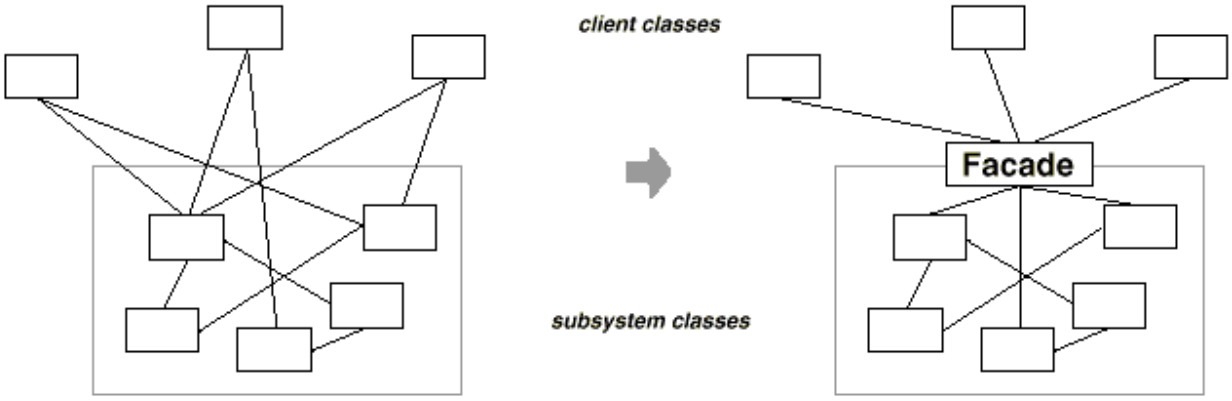
Le design pattern "façade" comporte plein de variantes.

Au sens le plus général, il s'agit de mettre en place une façade (simple à utiliser) de façon à indirectement bénéficier des services rendus par différents éléments d'un module (quelquefois complexe).

- Une façade de type "accueil/redirection" est une facade d'entrée unique pour un module complet ==> on est alors dépendant que d'un seul élément central (qui souvent redirige).
- Une façade "technologiquement agnostique" est un objet (que l'on instancie via un simple new) et qui cache toutes les technologies utilisées en arrière plan (ex: Spring, ...) .

Problème à résoudre (GOF):

Un module sans façade comporte de multiple points d'entrée que l'on est obligé de connaître pour initialiser les appels. D'autre part, ce n'est pas facile de s'y retrouver en cas d'évolution (quelles répercutions suite à tel changement ?).



Solution générale:

Introduire un nouvel élément intermédiaire qui jouera le rôle de "façade / accueil / point d'entrée unique" pour ce module.

Variantes de la solution:

Variantes	***Caractéristiques***	***Astuces (?)***
Nouvelle classe reprenant toutes les méthodes de toutes les classes internes (délégation en interne)	cette classe peut devenir énorme si le module est grand.	solution monolithique pas très évoluée
Simple intermédiaire de type "accueil" redirigeant vers les classes internes du module	ceci permet de ramener à 1 le nombre de point d'entrée à connaître mais le code des appels est, après redirection de l'accueil, de nouveau directement dépendant des éléments internes du module.	solution moyenne : accueil sans abstraction.
Intermédiaire de type "accueil/façade/fabrique" redirigeant vers des éléments concrets internes qui sont retournés comme des éléments abstraits .	Un seul point d'entrée à connaître + vision externe abstraite (pouvant être indépendante de la structure interne du module) ==> évolution interne facilitée.	solution astucieuse (très bien) mais un peu longue à programmer (interfaces , façade , ...)

2.6. Décorateur (enveloppe transparente)

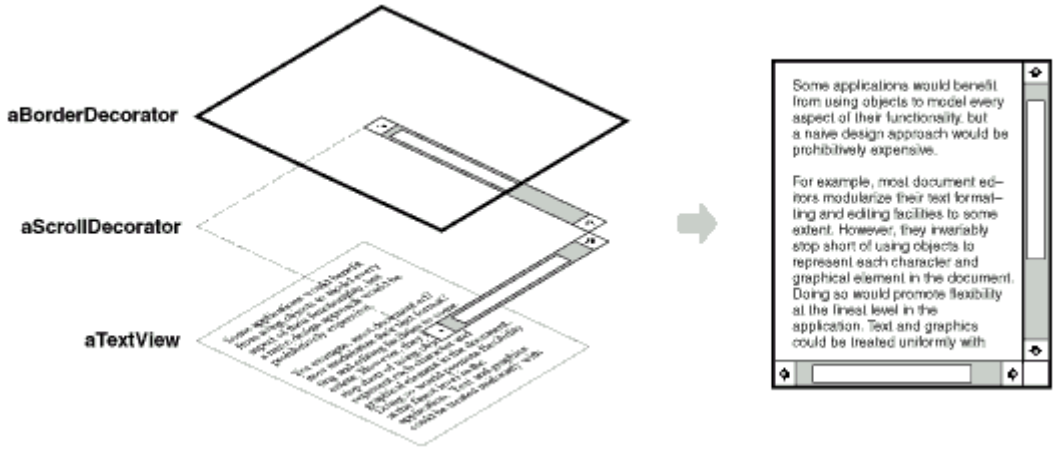
Provenance du nom "décorateur":

Soit "Composant Visuel" une classe correspondant à un composant graphique (sorte de fenêtre) de base.
Cette classe comporte les méthodes classiques "Afficher() , …"

On souhaite maintenant **manipuler de façon uniforme** différents types de composants graphiques avec de nouvelles fonctionnalités:

- Vue (éditeur) de texte (toute simple).
- Vue de texte avec ascenseurs et gestion automatique du scrolling.
- Vue de texte avec bordures de redimensionnement automatique et titre.
- Vue de texte combinant bordures et ascenseurs.

NB: les éventuelles décorations (ascenseurs, bordures) sont toujours gérées automatiquement.



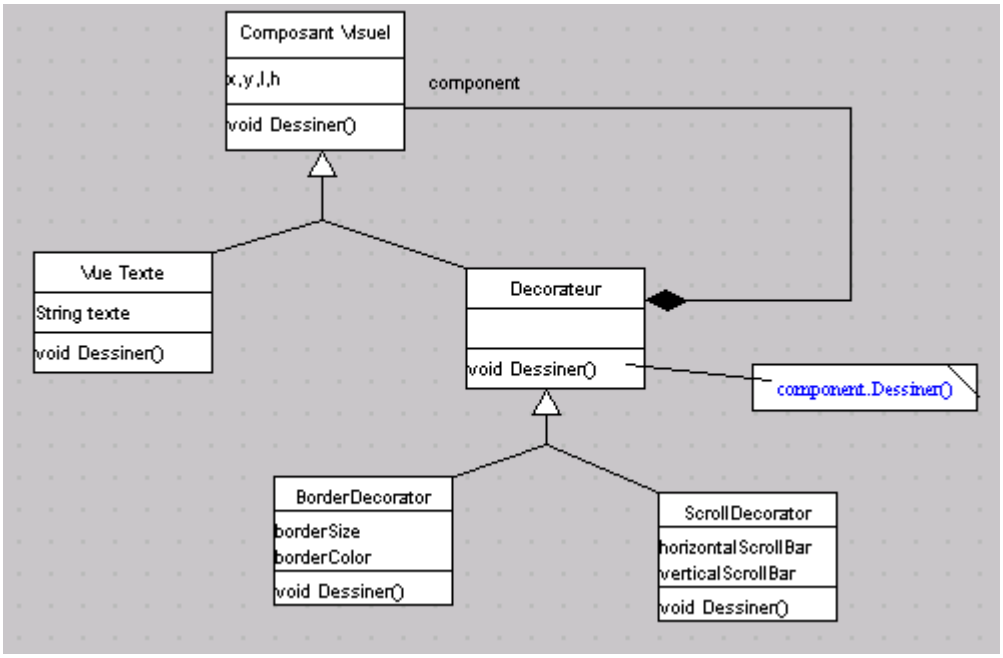
Solution intuitive mais moyenne:

N'utiliser que l'héritage et créer plein de classes telles que "Composant graphique avec bordures" , "Composant graphique avec Ascenseurs" , …

Problème: cette solution est un peu trop statique (figée). Si l'on souhaite une classe combinant bordures et ascenseurs , il faut effectuer un nouvel héritage (éventuellement multiple).

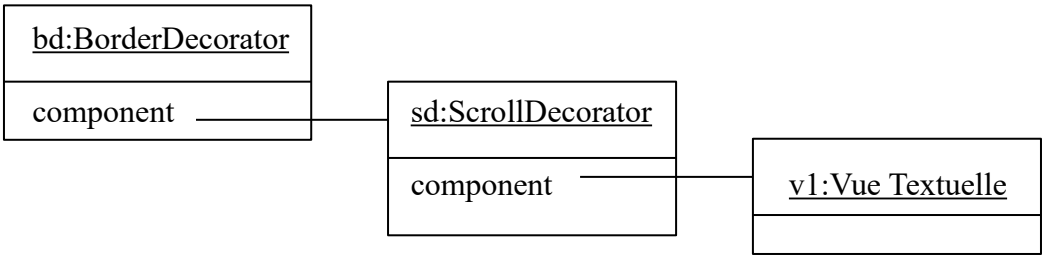
*Solution plus flexible proposée (Design Patter "**Decorator**"):*

.../...



Il suffit d'effectuer une série d'imbrications pour obtenir une vue avec bordures et ascenseurs.

D'autre part, la classe Décorateur hérite de "Composant Visuel" et est donc vue comme un composant visuel ordinaire (bien qu'il ait une composition en interne).



Sémantique "enveloppe transparente":

Le design pattern ci-dessus correspond à la notion de "**enveloppe transparente**".

Le code d'utilisation ne voit l'objet manipulé que comme un composant de base ordinaire.

L'**objet réellement manipulé** peut en fait être une **enveloppe intermédiaire** qui:

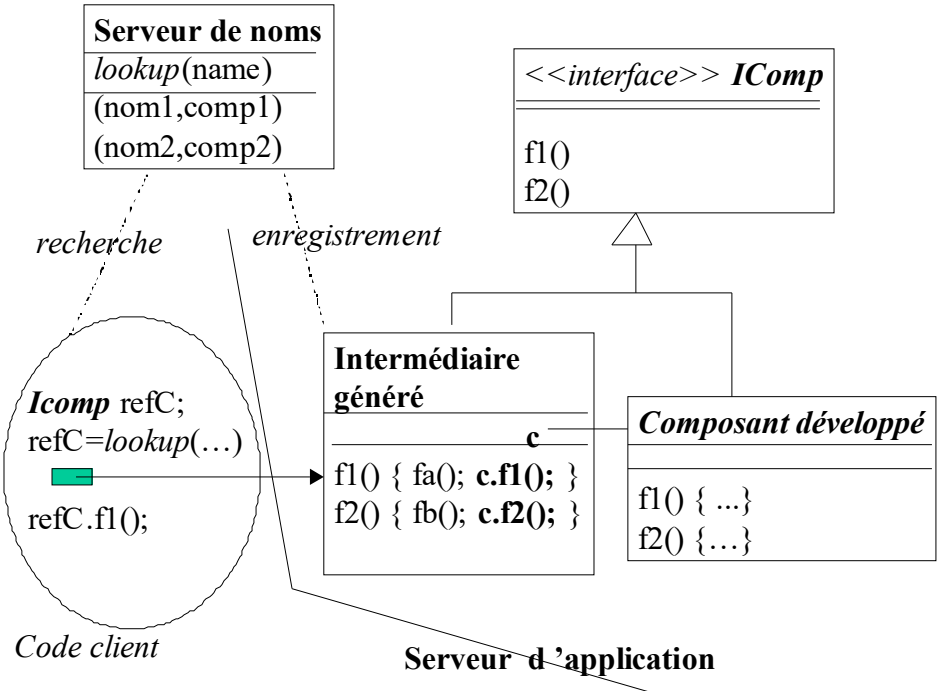
- **ajoute de nouvelles fonctionnalités**
- **délègue les traitements de base au niveau de l'objet imbriqué**.

Quelques exemples concrets du Design pattern "Decorateur":

- classe **JScrollPane** de Java/Swing
- classe ***BufferedReader*** et ***InputStreamReader*** de ***java.io*** ,

Interposition dans les serveurs d'applications:

La plupart des serveurs d'applications (MTS de Microsoft, J2EE , WebLogic, WebSphere, ….) utilisent des astuces dérivées de ce design pattern pour ajouter de nouvelles fonctionnalités aux composants que l'on déploie dedans.



Nouvelles fonctionnalités couramment apportées: gestion des transactions, de la sécurité et de la montée en charge.

Limitations et extension:

Pour être applicable , le design pattern décorateur doit partir d'un élément de base bien défini qu'il faudra envelopper.
Cet élément de base doit comporter un certain nombre de méthodes à conserver dans les décorateurs. **Ce paquet de méthodes doit être en nombre fini et les prototypes de ces méthodes doivent être connus et stables**.

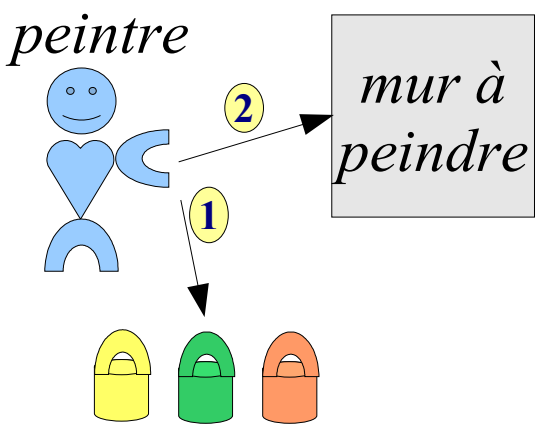
Lorsque les prototypes des méthodes ne sont pas connus et/ou lorsque le nombre de méthode de l'élément de base peut varier on doit alors utiliser la ***programmation par aspects (A.O.P.)***.

IV - IOC (injections de dépendances)

1. Design Pattern "I.O.C." / injection de dépendances

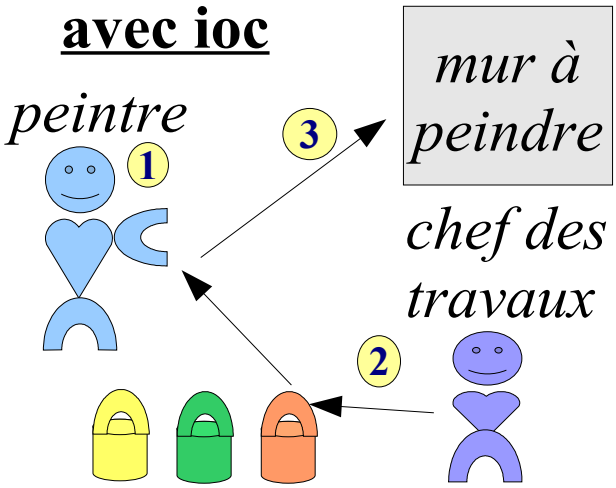
1.1. IOC = inversion of control

sans ioc



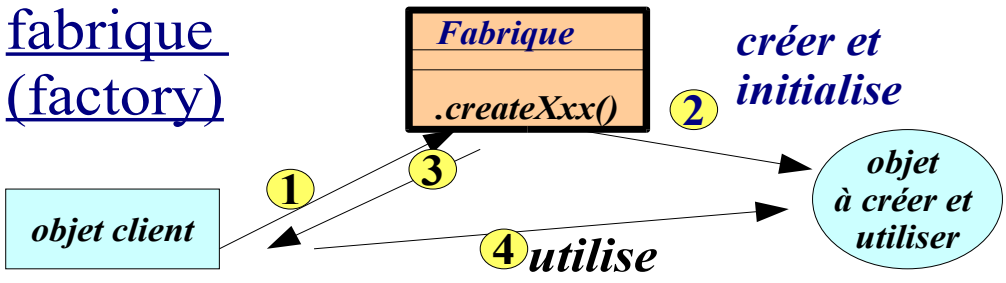
1) le peintre ***choisit*** la ressource "pot de peinture" *(il a le contrôle sur la couleur)*
2) il l'***utilise*** pour peindre

avec ioc

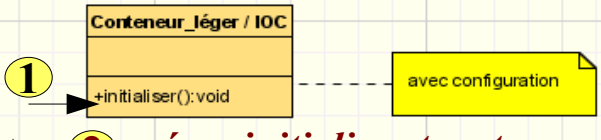


1) le peintre ***attend qu'on lui fournisse*** le pot de peinture
2) le chef des travaux lui donne le pot à utiliser *(c'est maintenant lui qui contrôle la couleur)*
3) ***utilisation*** du pot pour peindre

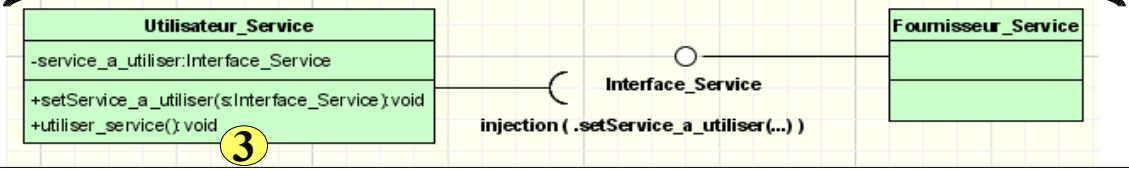
Simple fabrique (factory)



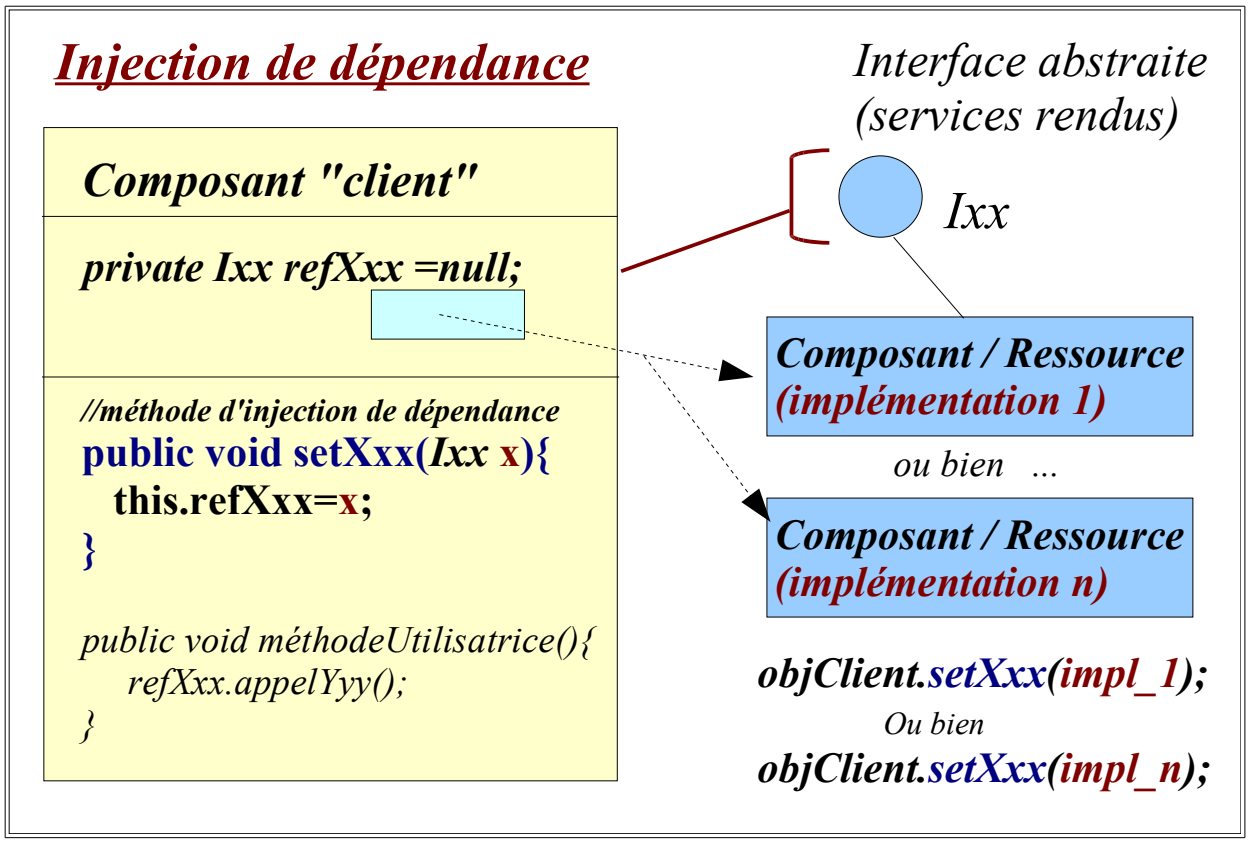
I.O.C.
= Inversion Of Control
(Injection de dépendances)



2 ***créer , initialise et met en relation***



1.2. injection de dépendance



Le ***design pattern "IOC"*** (*Inversion of control*) correspond à la notion d'**injection de dépendances abstraites**.

Concrètement au lieu qu'un composant "client" trouve (ou choisisse) lui même une ressource avant de l'utiliser , cet **objet client exposera une méthode de type:**

public void setRessources(AbstractRessource res)

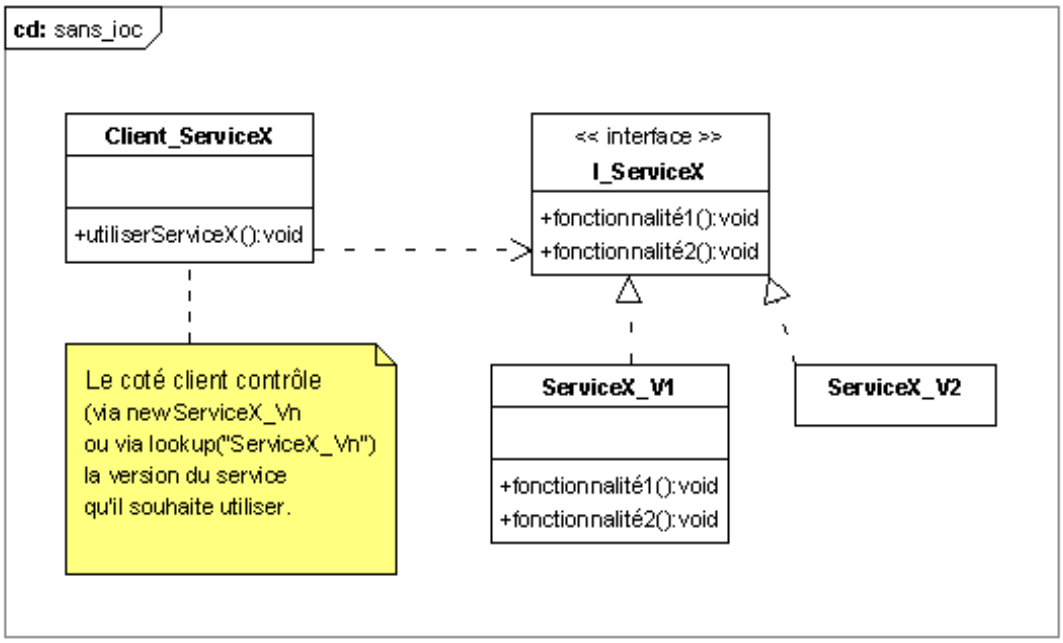
ou bien un constructeur de type:

public CXxx(AbstractRessource res)

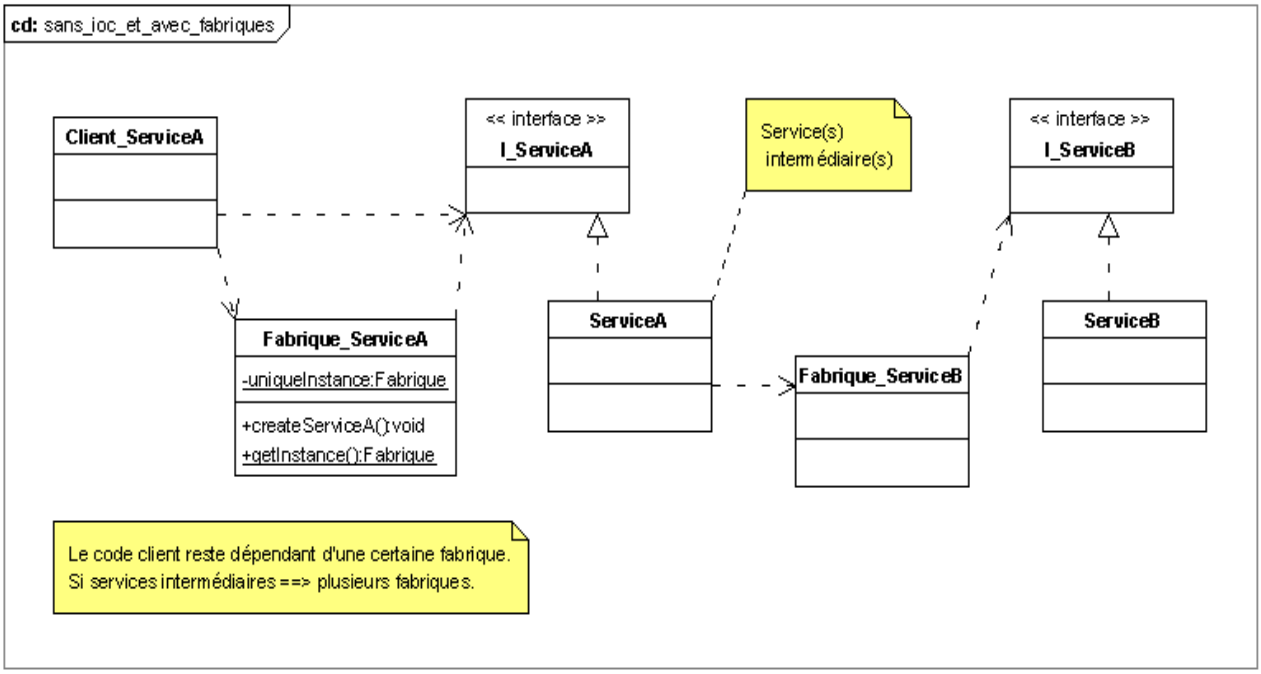
permettant qu'on lui fournisse la ressource à ultérieurement utiliser.

Un tel composant est beaucoup plus réutilisable .

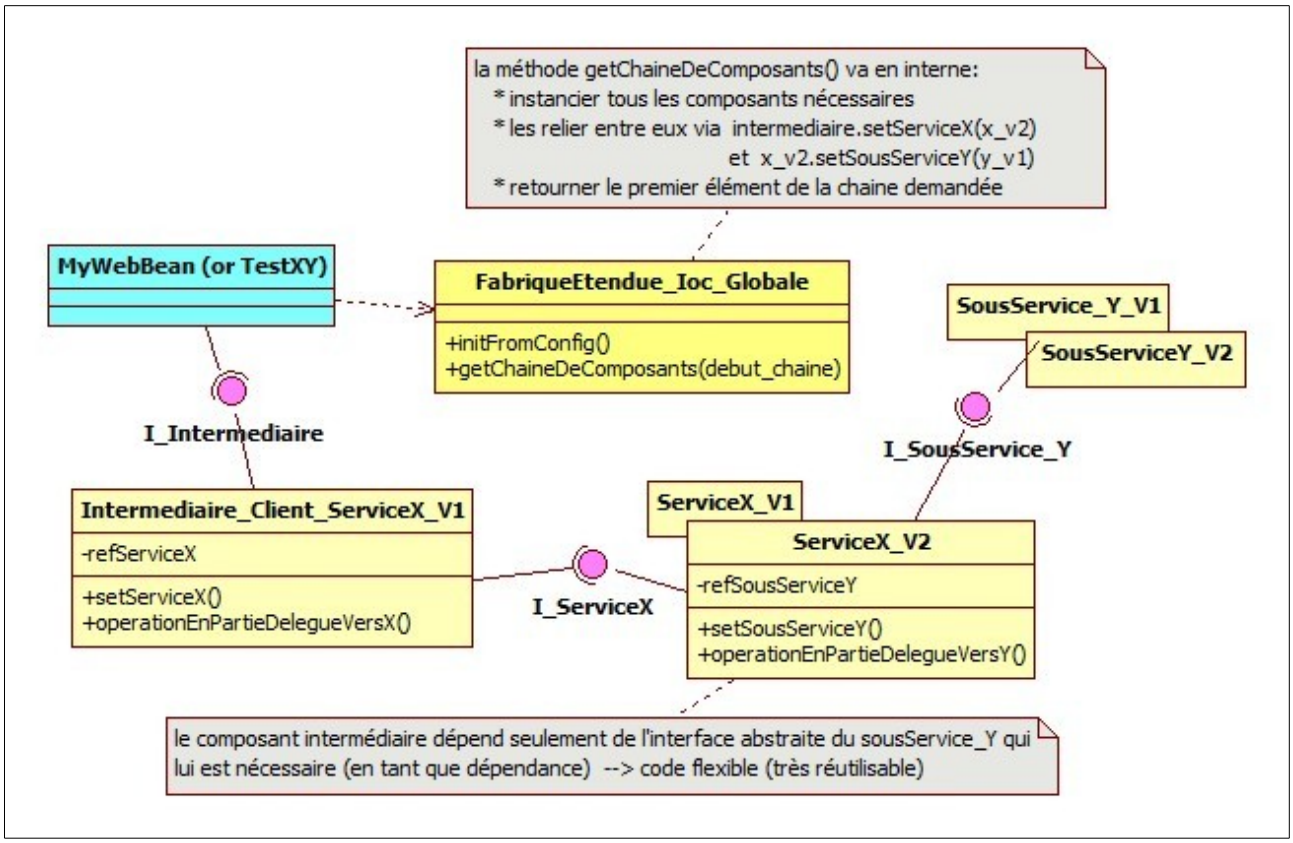
1.3. sans I.O.C. ni fabrique



1.4. sans I.O.C. et avec de multiples petites fabriques



1.5. avec I.O.C. (et donc avec fabrique globale)



1.6. Micro-kernel / conteneur léger

Pour être facilement exploitable , le design pattern IOC nécessite un **petit framework** généralement appelé "**micro-kernel**" ou "**conteneur léger**" prenant à sa charge les fonctionnalités suivantes:

Enregistrement des "ressources" (composants concrets basés sur interfaces abstraites) avec des **identifiants** *(noms logiques)* associés.

Instanciation et/ou initialisation des composants en tenant compte des dépendances à injecter (==> liaisons automatiques avec composants "ressources" nécessaires)

Ceci nécessite **quelques paramétrages** (***fichier de configuration XML*** ou bien ***annotations*** au sein du code).

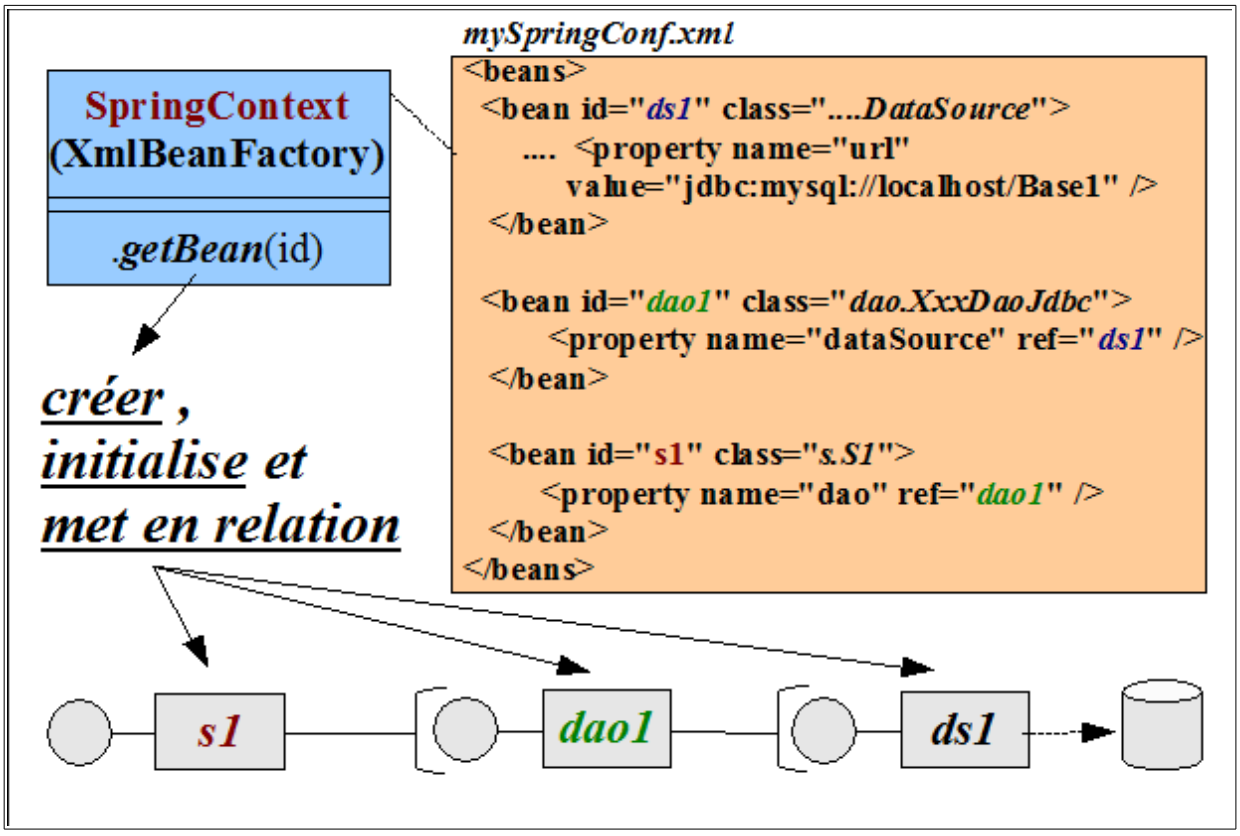
2. injection de dépendances (exemples impl.)

Principales implémentations de l'injection de dépendance

- **Spring** (java – non officiel JEE mais "standard de fait")
- **CDI / JEE6_et_7** (java / JEE officiel)
- **injections avec Angular JS 1.x** (sans typage fort)
- **injections avec Angular 2 (et TypeScript)** (avec typage plus rigoureux)
- …

2.1. Injection de dépendance avec Spring

Époque "2004-2007" : **configuration en XML** (toujours possible) :




```
ApplicationContext context =
        new ClassPathXmlApplicationContext("mySpringConf.xml");
 MyJavaService s1 = (MyJavaService) context.getBean("myService");
 …
```

Epoque "2008-2014" : **configuration par annotations** (toujours possible)

```
@Component("serviceXY")
public class ServiceXYAnot implements IServiceXY {

	private XYDao xyDao;

	//injectera automatiquement l'unique composant Spring
	//dont le type est compatible avec l'interface précisée.
	@Autowired //ici ou bien au dessus du "private ..."
	public void setXyDao(XYDao xyDao) {
		this.xyDao = xyDao;
	}

	public XY getXYByNum(long num){
		return xyDao.getXYByNum(num);
	}
```

avec (dans petite configuration xml) :

```
<context:annotation-config/>

<context:component-scan  base-package="com.xy.tp"/>
```

Epoque "2015-...." : **configuration via "classe java de config"**

```
@Configuration
public class DataSourceConfig {

	@Bean(name="myDataSource") //by default beanName is same of method name
	public DataSource dataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setDriverClassName("com.mysql.jdbc.Driver");
		dataSource.setUrl("jdbc:mysql://localhost:3306/minibank_db_ex1");
		dataSource.setUsername("root");
		dataSource.setPassword("root");//"root" ou "formation" ou "..."
		return dataSource;
	}

	@Bean(name="myEmf")
	public EntityManagerFactory entityManagerFactory(
			JpaVendorAdapter jpaVendorAdapter, DataSource dataSource) {
		LocalContainerEntityManagerFactoryBean factory
				= new LocalContainerEntityManagerFactoryBean();
		factory.setJpaVendorAdapter(jpaVendorAdapter);
		factory.setPackagesToScan("tp.myapp.minibank.impl.persistence.entity");
		factory.setDataSource(dataSource);
		factory.afterPropertiesSet();
		return factory.getObject();
	}
	// ...
```

```
}
```

et

```
ApplicationContext context =
   new JavaConfigApplicationContext(DataSourceConfig.class,
                                    DomainAndPersistenceConfig .class);
   Service service = context.getBean(Service.class);
```

2.2. Injection avec CDI JEE6/7

CDI = "**C**ontext/Container **D**ependencies **I**njections"

CDI (alias JSR-299) utilise en interne DI / JSR-330 (@Inject et @Named) mais va plus loin en apportant bien plus de fonctionnalités (dans un environnement JEE6) .
Les fonctionnalités additionnelles de CDI par rapport à DI sont essentiellement :

- la prise en charge des contextes (associés aux scopes "request", "session" , "application" et "conversation")
- l'association automatique "EJB session stateful" avec session Http .
- des injections transparentes entre EJB et JavaBean web (jsf,...) .
- prise en charge de certains événements
- prise en charge des intercepteurs (décorateurs/aop) .
- une approche fortement typée et avec un très faible couplage

On reconnaît l'utilisation de CDI à la présence d'un fichier "**beans.xml**" (éventuellement vide mais obligatoire) dans les répertoires **WEB-INF** (web) ou **META-INF** (ejb) .

WEB-INF/beans.xml

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
      <!--
        <alternatives>
              <class>....</class> <class>....</class>
        </alternatives>
       -->
</beans>
```

Pour l'instant, les deux principales implémentations des spécifications CDI sont :

- **Weld** (de Jboss)
- **OpenWebBeans** (de Apache)

```
@Named("myXyDao")   //ou bien @Named()
public class XYDaoImplAnot implements XYDao {
…
}
```

et

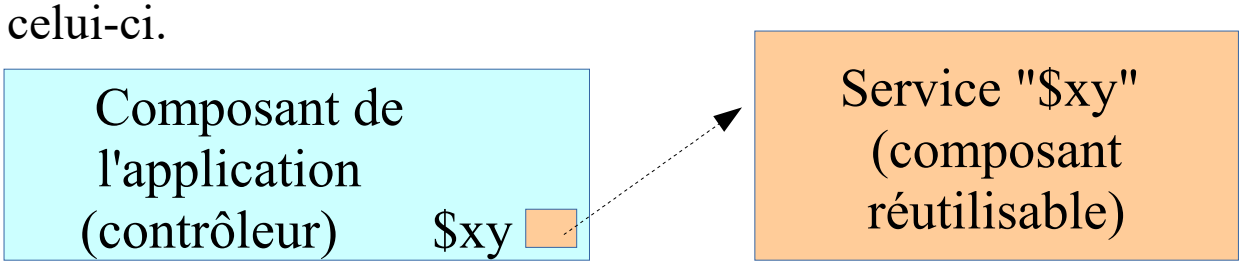
```
@Named("serviceXY")
public class ServiceXYAnot implements IServiceXY {

	private XYDao xyDao;

	//injectera automatiquement l'unique composant Spring
	//dont le type est compatible avec l'interface précisée.
	@Inject //ici ou bien au dessus du "private ..."
	public void setXyDao(XYDao xyDao) {
		this.xyDao = xyDao;
	}

	public XY getXYByNum(long num){
		return xyDao.getXYByNum(num);
	}
```

2.3. Principe d'injection de dépendance sans typage fort

Principe d'injection de dépendance

L'injection de dépendance consiste à relier entre eux deux niveaux de composants (codes orientés objets) compatibles.

Le composant de haut niveau utilise le sous composants (dont il dépend) en arrière plan via une référence variable vers celui-ci.



L'initialisation de cette référence peut éventuellement être automatiquement prise en charge par un framework (ex : spring ou angular-js) selon certaines règles ou certains paramétrages.

Certains langages de programmation (tels que C++ , java , C#) sont fortement typés et les injections de dépendances automatiques peuvent être basées sur des mises en correspondance entre des références et des composants existants qui ont des types de données compatibles (ex : instance d'une classe *ServiceXxImpl* implémentant l'interface *IServiceXx* compatible et une référence à injecter de type *IServiceXx*).
Javascript étant un langage faiblement typé , les injections automatiques ne peuvent simplement être basées que sur des correspondances de noms (ex : nom de paramètre d'une fonction et nom d'un

composant/service existant quelque-part) et des exceptions sont à prévoir en cas d'incompatibilité.

Soit f1 une fonction javascript , on peut récupérer tout son code source via f1**.toString()** ;
Un framework javascript (tel que angular-js) peut donc analyser (via un découpage basé sur des expressions régulières) les noms des paramètres pour ensuite effectuer des mises en correspondances automatiques.

2.4. Injection avec Angular JS 1.x (javascript)

L'injection de dépendance effectuée par angular-js s'effectue en fonction d'**une mise en correspondance automatique entre**
les noms des paramètres d'une fonction javascript prise en charge par le framework (***contrôleur***) **et**
les noms des objets/composants/services instanciés dans le scope courant (en tenant compte des éléments hérités).

→ au final , le nom des paramètres est plus important que l'ordre de ceux ci .

Le mécanisme d'injection d'angular-js , déclenche si besoin l'instanciation d'un composant/service nécessaire via une fabrique spéciale (service factory) .
D'autre part , il peut éventuellement y avoir plusieurs niveaux d'injections (déclenchés automatiquement par transitivité) .

Exemple :

```
myAjsApp.controller('ProductListCtrl', function ($scope,$http) {

 $http.get('data/products.json').success( function(data) {
        $scope.products = data;
        });
 $scope.title = "list of smartphones";
});
```

Dans l'exemple ci-dessus , lorsque la fonction **function ($scope,$http) { … }** est prise en charge par angular-js , les paramètres $scope et $http sont automatiquement initialisé en y injectant le scope courant ($scope) et le service $http pré-défini .

V - Patterns pour architecture n-tiers (JEE, ...)

1. D.A.O. (Data Access Object)

Problématique / Motivations:

- Les applications ont besoin à un moment ou à un autre d'accéder à des données persistantes, il peut s'agir par exemple de LDAP, Mainframe, Base de données, Système B2B. Mais l'inclusion du code de connexion et l'accès aux données dans les composants introduit un couplage étroit.

- De telles dépendances dans les composants rendent la migration d'un type de source de données à un autre difficile et fastidieuse, parce qu'il faut alors modifier les composants pour qu'il puissent la prendre en charge.

Solution :

- Utiliser DAO pour abstraire et encapsuler tout ce qui concerne l'accès aux sources de données. DAO gère la connexion, l'obtention des données et leur mémorisation.

Conséquences :

- Facilitation de la migration.
- Simplification du code des objets métier.
- Centralisation de l'accès aux données dans une couche séparée.

NB: le design pattern "DAO" est un cas particulier du design pattern "Stratégie" .
Le DAO correspond à l'implémentation le plus classique de la (sous-) couche de persistance.

Un composant de type **DAO** est généralement structuré en :

- une partie "**interface abstraite**" répertoriant tout un tas de méthodes CRUD (Create, Retreive, Update, Delete) et ayant:

  - des POJOs de données en paramètres d'entrée et/ou en sortie (valeur de retour)
  - des remontées d'exceptions uniformes (quelque soit la technologie utilisée).

- une "**implémentation concrète**" (idéalement interchangeable) basée sur une technologie donnée (ex: JDBC, Hibernate, JPA, ...).

NB1: A une certaine époque (vers 2008-2012) l' hégémonie des bases relationnelles et de JPA en java a peut être pu faire apparaître le composant DAO comme un peu superflu (au sens "une seule variante") .

Aujourd'hui la progression des bases "NoSQL" (MongoDB , Cassandra , …) remet en avant l'utilité du DAO (si l'on souhaite obtenir une architecture modulaire au niveau de la persistance).

NB2 : Un composant DAO est techniquement un "sous service en arrière plan" et peut souvent se mettre en œuvre avec les mêmes technologies qu'un service métier (ex : EJB @Stateless ou composant "spring"avec @Transactional).

NB3 : Attention : il faut penser à la structure globale du composant DAO ("Interface + structure de données manipulées") pour mesurer son niveau d'indépendance vis à vis d'une technologie (ex : classe d'entité persistante java avec @Entity , @Id fortement connotée JPA) .

Selon les types d'entités manipulées par un DAO, on peut ranger les DAO en deux grandes classes:

- les **DAO génériques** (fonctionnant sur n'importe quel type d'entité ["**Object**" en java]).
- les **DAO spécifiques** (spécialisés dans la persistance d'un type précis d'entité --> exemple: CompteDAO avec méthodes "getCompteByNum() , deleteCompte() , ...)

NB: Les objets "session" d'hibernate et l'objet "entity manager" de JPA peuvent être considérés comme des DAO génériques .

Exemple:

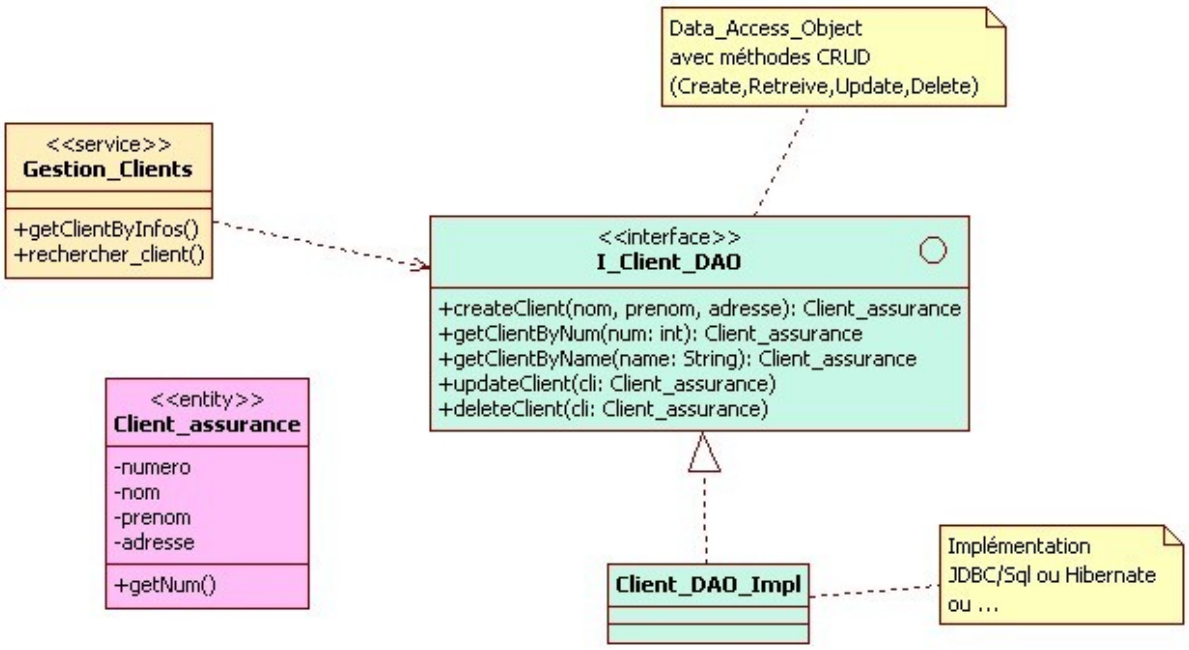
soit ***Compte*** une classe (JavaBean / POJO) représentant une entité "Compte"
avec getNum() / setNum(...) , getSolde() / setSolde(...) , ...

la vision abstraite du DAO sera alors ressemblant à:

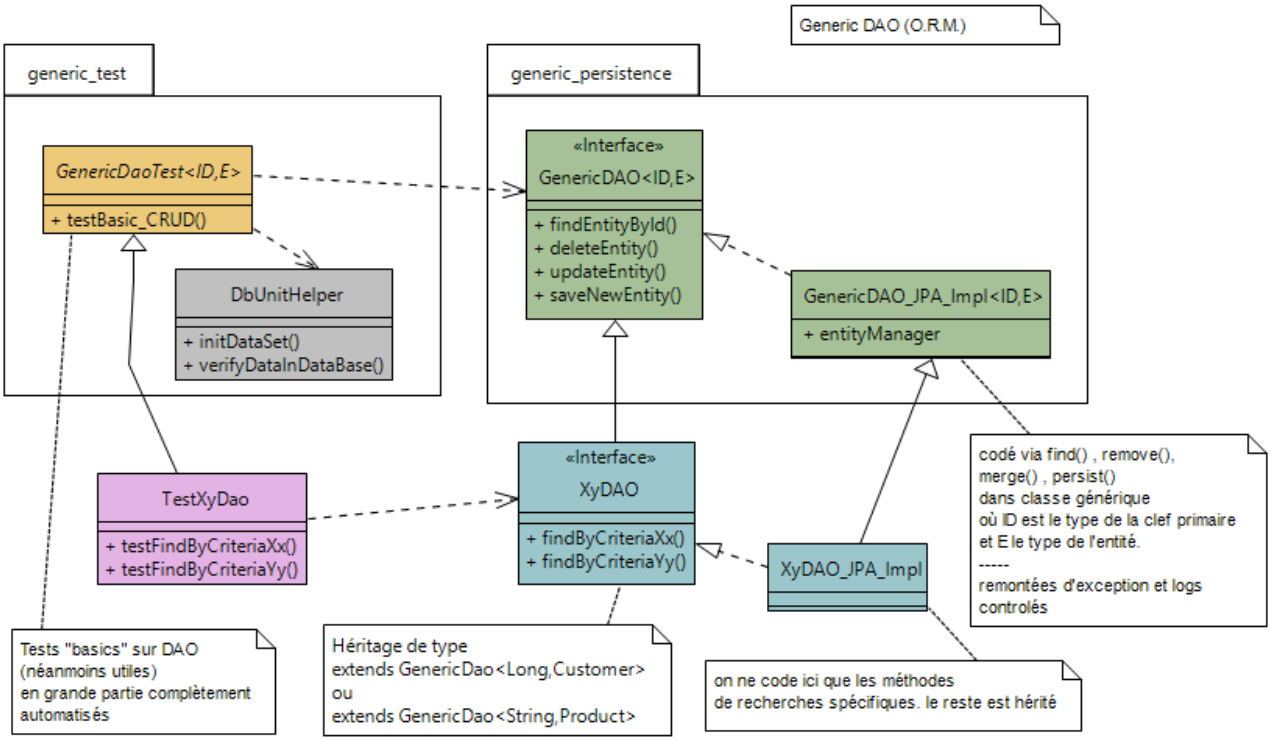
```
public interface CompteDAO {
 public Collection<Compte> getCompteOfClient(long num_client) throws RuntimeException ;
 public Compte getCompteByNum(long numCpt) throws RuntimeException ;
 public void updateCompte(Compte cpt) throws RuntimeException;
 public void deleteCompte(long numCpt) throws RuntimeException;
 public long createCompte(Compte cpt) throws RuntimeException; // return pk;
 ...}
```

(NB: le throws RuntimeException) peut être implicite (non explicité).

Exemple2 (en UML) :



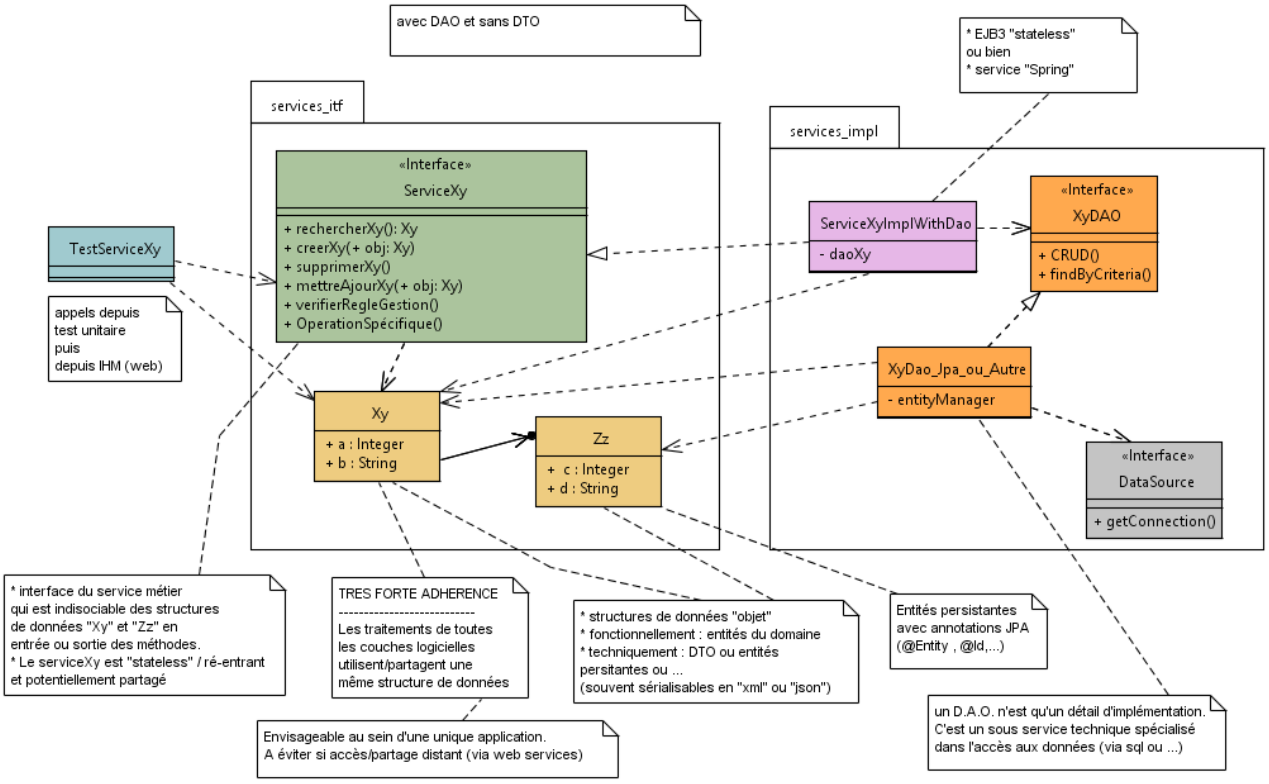
Eventuel Dao générique :



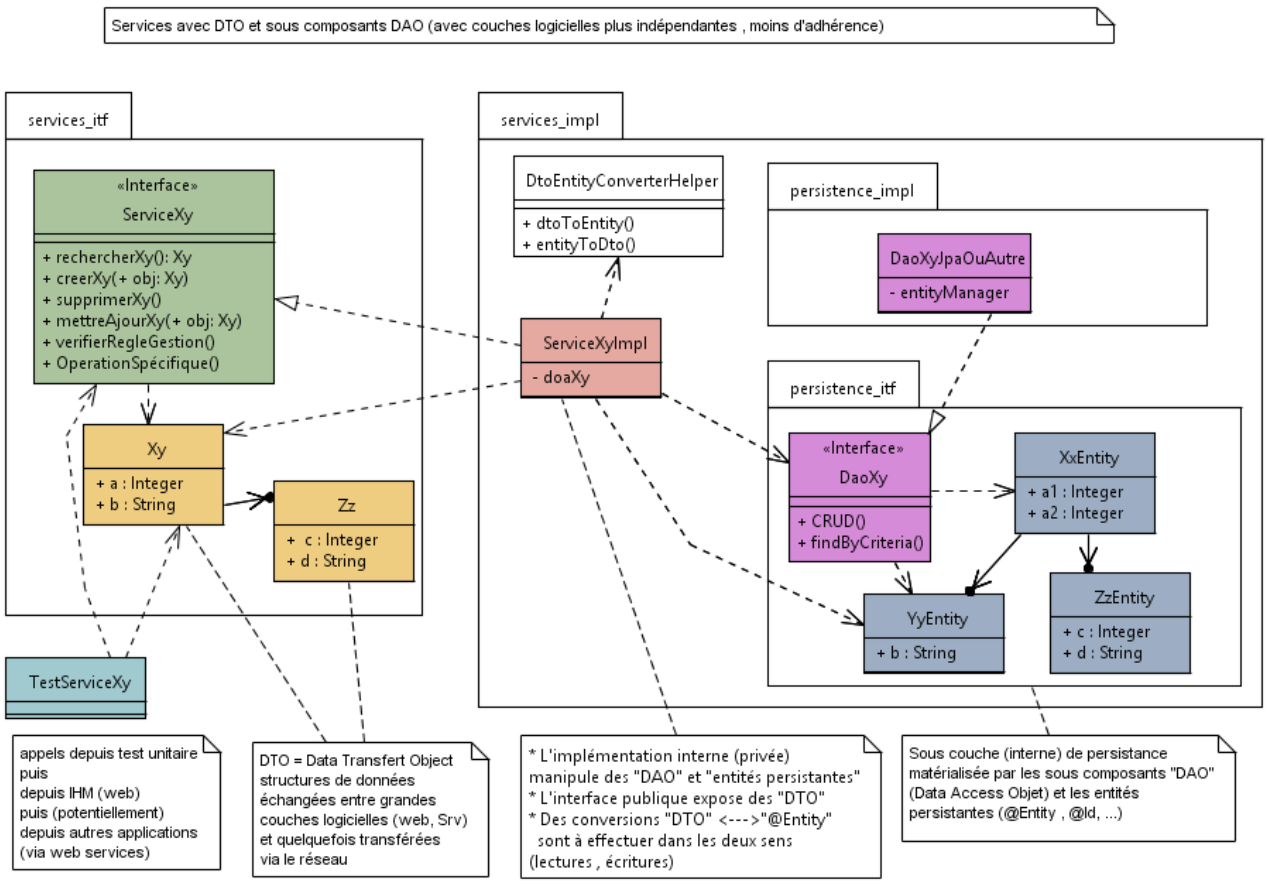
Un DAO générique permet :

- de compacter le code (moins de lignes de code à écrire grâce à l'héritage).
- automatiser les tests les plus simples

Architecture avec DAO (et sans DTO) / FORTE ADHERENCE :



2. DTO / VO



2.1. VO / DTO – pour technologies RPC (EJB,WS,...)

DTO = **D**ata **T**ransfert **O**bject , **VO** = **V**alue **O**bject (proche DTO , quelquefois "immuable")

2.1.a. Optimisation des flots de données en informatique répartie

L'informatique répartie permet de rendre les appels de fonctions à travers le réseau quasiment transparents.
Néanmoins cette fonctionnalité cache des mécanismes généralement coûteux (création d'un paquet "requête" , transfert réseau , analyse du paquet, appel de la fonction, création d'un paquet "résultat" , transfert inverse , …).

Chose à ne surtout pas faire : effectuer pleins de "micro-requêtes" (appeler getNom() , puis getPrenom() , puis getAge() , …).

➔ Pour optimiser les performances, on a généralement recours à de nouvelles classes qui ne font que rassembler dans une bloc global une multitudes de données élémentaires accessibles en lecture seulement:

PersonneDTO persData = serviceDistant.getPersonneData(); *// une seule requête distante*
// persData est la valeur de retour récupérée coté client et donc locale.
nom=persData.**getNom**() ; *// immédiat*
prenom=persData.**getPrenom**(); *// immédiat*

…

==> public class PersonneDTO **implements java.io.Serializable** { ... }

2.2. DTO / VO (avec ou sans RPC)

DTO est un synonyme pour le design pattern VO.

Vues "métier" (alias DTO / VO)

Une "***vue métier***" est un ***objet sérialisable*** et qui servira à faire communiquer un service métier avec un client (Web ou ...) potentiellement distant. [*synonymes classiques*: ***V**alue **O**bject ou **D**ata **T**ransfert **O**bject*].
Outre son aspect technique lié aux appels éventuellement distants, une vue métier est utile pour que:

* La couche N (Appli-Web,...) ne voit pas directement les entités persistantes remontées par la couche N-2 (D.A.O.).
* L'application puisse manipuler des vues souvent simplifiées (sans tous les détails des tables relationnelles ou des objets persistants).

Rappel: Un DTO (Data Transfert Object) [alias VO=Value Object , alias "vue orienté objet"]

est un objet sérialisable qui est une copie (souvent simplifiée) d'une entité persistante.

NB: Les classes des "DTO" ne devraient idéalement pas être reliées aux classes d'entité par un héritage (ni dans un sens , ni dans l'autre) pour garantir une bonne indépendance entre deux formats de données qui seront utilisés dans deux couches logicielles bien distinctes .

Une classe de "DTO" peut être une version "partielle" d'une classe d'entité (ne reprenant que les principales propriétés qui ont besoin d'être affichées ou saisies) ou peut être une structure "ad hoc" résultant d'un assemblage de valeurs puisées dans différentes entités.

Analogie : "Vue relationnelle basée sur une ou plusieurs tables d'une base de données).

Astuce technique:

Lorsque l'on a besoin de recopier beaucoup de valeurs d'un objet vers un autre (qui ont des propriétés communes) on peut s'appuyer sur la méthode statique utilitaire

BeanUtils.copyProperties(obj1,obj2); de Spring (ou de jakarta commons).

Ceci est un équivalent plus compact pour:

obj2.setXxx(obj1.getXxx()); obj2.setYyy(obj1.getYyy()); obj2.setZzz(obj1.getZzz());

NB: copyProperties recopie toutes les propriétés qui ont le même nom (en effectuant si besoin des conversions élémentaires String ---> int , ... et en ignorant les propriétés sans correspondance de nom).

Attention: BeanUtils.copyProperties(source,destination) en version Spring et inversement BeanUtils.copyProperties(destination, source) en version "jakarta-commons" .

NB: On peut également utiliser le framework de conversion "Dozer" (paramétrable en XML) pour convertir des "entity" en "dto" et vice-versa.

2.3. Avec ou sans DTO/VO ???

Avec DTO:

Avantages : couches logicielles bien indépendantes , plus de problème "lazyException" , on ne remonte à la couche présentation que ce qui est nécessaire (et dans un format bien ajusté/contrôlé). On LIMITE les ADHERENCES .

Inconvénients: long à programmer (assez lourd) et un peu pénalisant sur les performances (charges mémoire & CPU supplémentaires).

Sans DTO:

Avantages : architecture simple et légère , performances optimisées , rapide à développer.

Inconvénients: Filtre "Open....InViewFilter" nécessaire pour ne pas trop avoir de "LazyException" (avec tout de même quelques petits effets de bord) , couches logicielles fortement couplées .

Astuce technique: via @XmlTransient (de JAXB2 utilisé en interne par JAX-WS) et via le filtre "Open....InViewFilter" il est éventuellement possible de remonter directement les entités persistantes via des "Services WEB" (avec CXF ou autre) sans pour autant avoir des boucles dans le suivi des relations bidirectionnelles .

SLOGANs des "Anti DTO" :

- DRY = **D**on't **R**epeat **Y**ourself

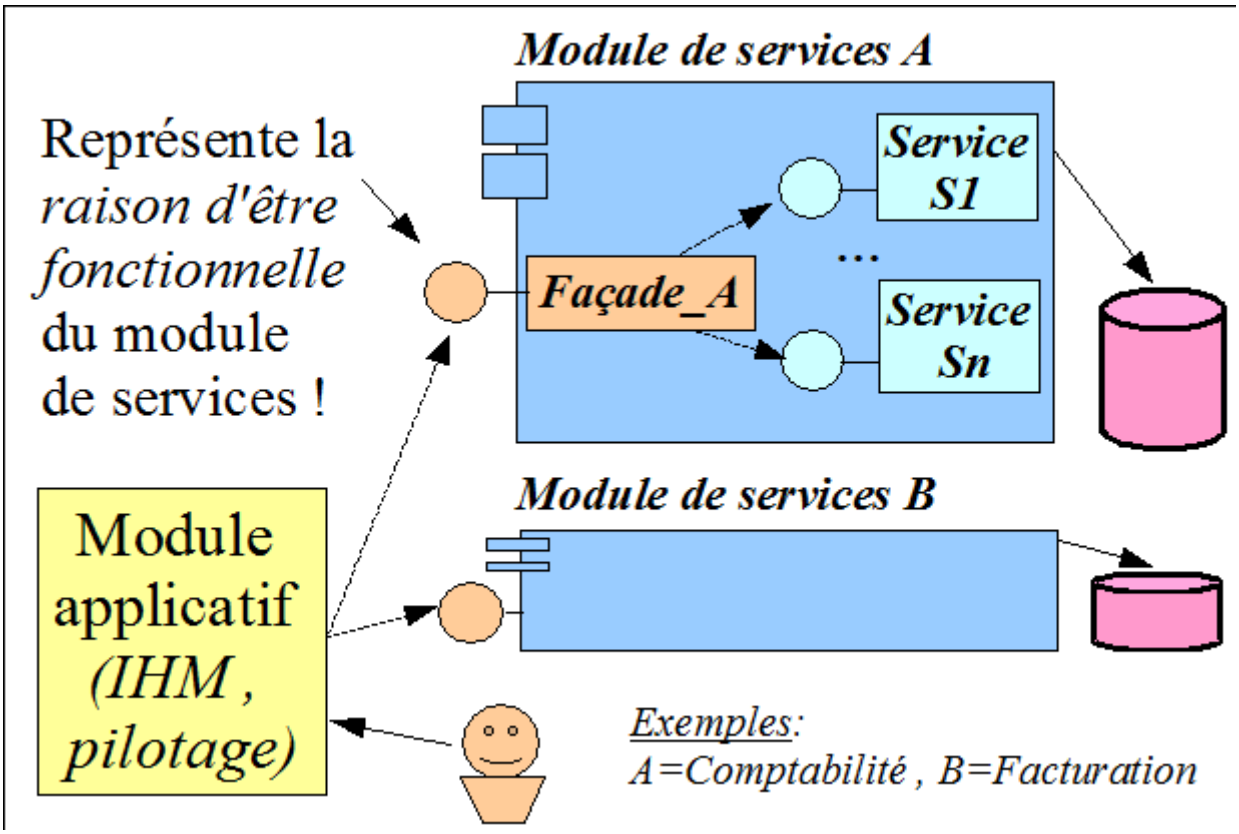
- KISS = **K**eep **I**t **S**imple **S**tupid

- critique "**modèle objet anémique**" sur DTO

Au Final : Tout dépend du contexte (petite ou grosse application , Web Services ou pas , degré de complexité souhaité , …) .

3. Façades , Référentiel et BusinessDelegate

3.1. Façade pour module de services



D'un point de vue fonctionnel , une façade donne du sens à un module de service. Son nom doit donc être bien choisi (pour être évocateur).

D'un point de vue technique, une façade n'est qu'un nouvel élément intermédiaire de type "Façade d'accueil" qui servira à orienter les clients vers les différents services existants.

Exemple:

Façade_Comptabilité
. getServicePostesComptables()
.getServiceBilan()
.getServiceJournal()
.getServiceGrandLivre()
...
.getServiceN()

utilisation:

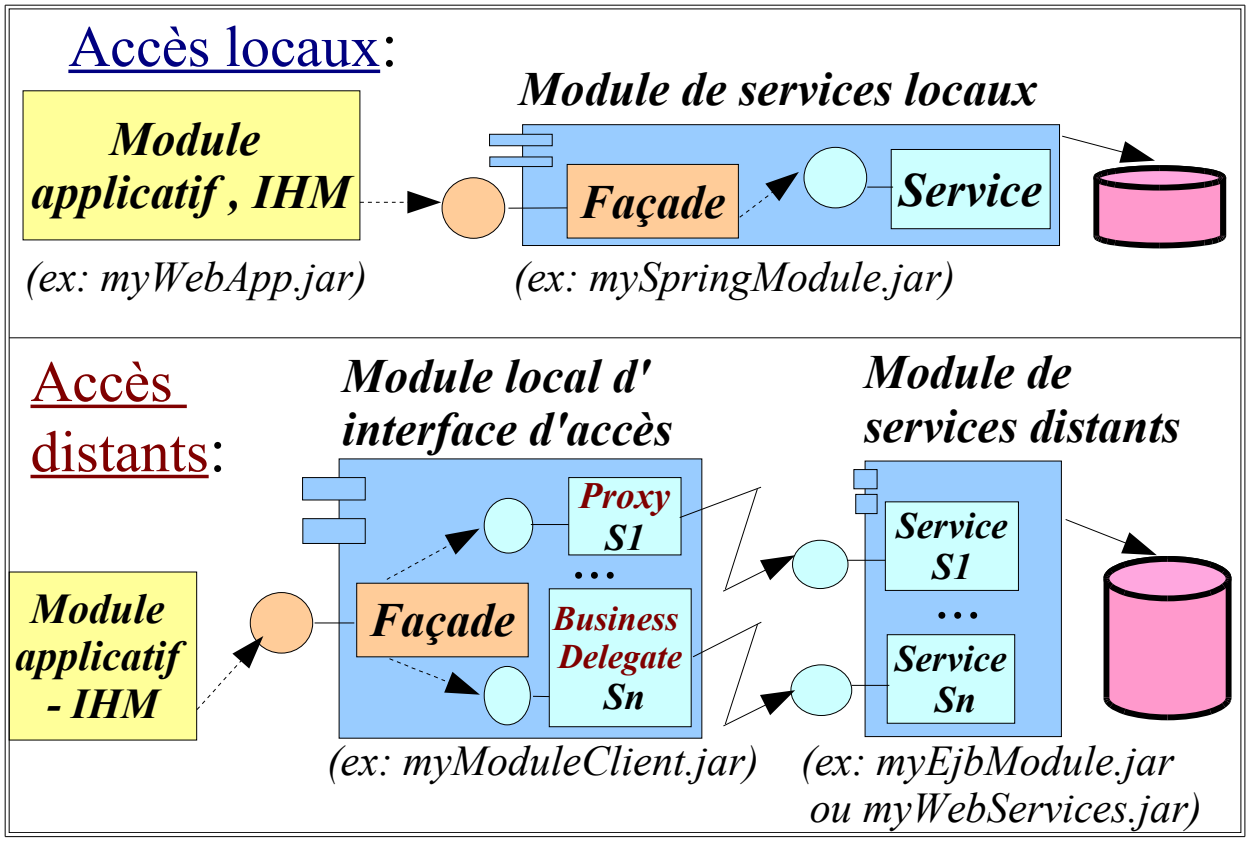
facadeCompta.getServiceBilan().xxx()

facadeCompta.getServiceGrandLivre().yyy();

Une facade simple (sans accès distants) est techniquement un simple niveau intermédiaire avec :

- injection des services (setXxxService())
- méthode d'accès aux services (getServices())
- initialisation via framework IOC (spring ou Jsf ou ....)

En mode distant:



Un "**proxy**" est un représentant local d'un service distant .

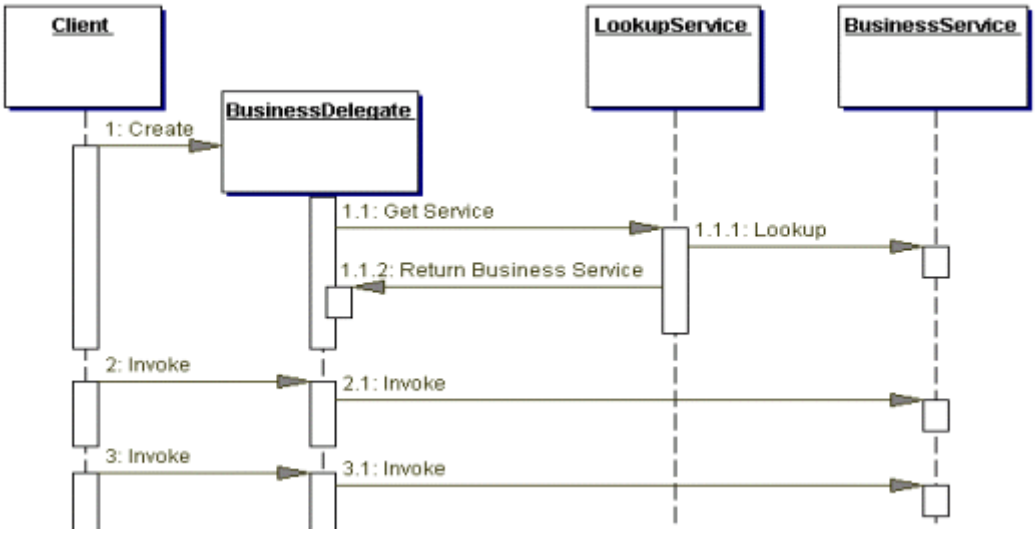
Ce terme (plutôt technique) désigne assez souvent du code généré automatiquement (à partir d'un fichier WSDL par exemple).

Pour bien contrôler l'interface locale d'un service distant (de façon à n'introduire aucune dépendance vis à vis d'une technologie particulière), on met parfois en oeuvre des objets de type "**business delegate**" qui cache dans le code privé d'implémentation tous les aspects techniques liés aux communications réseaux:

- localisation du service , connexion (lookup(EJB) ou ...)
- préparations/interprétations des messages/paramètres .
- ...

Utiliser un Business Delegate pour réduire le couplage entre les clients de la couche présentation et les services métier. Le Business Delegate cache les détails d'implémentation de la couche métier, tels que ceux de la recherche de service métier,...
C'est une sorte de proxy évolué (pas trop technique mais plutôt orienté métier) avec éventuellement

des talents d'adaptateur .



Référentiel ---> Façades (avec ou sans BusinessDelegate) ---> Services

4. MVC (Model – Vue – controller)

MVC (Model – View – Controller)

standard de facto, utilisé depuis 1972 !

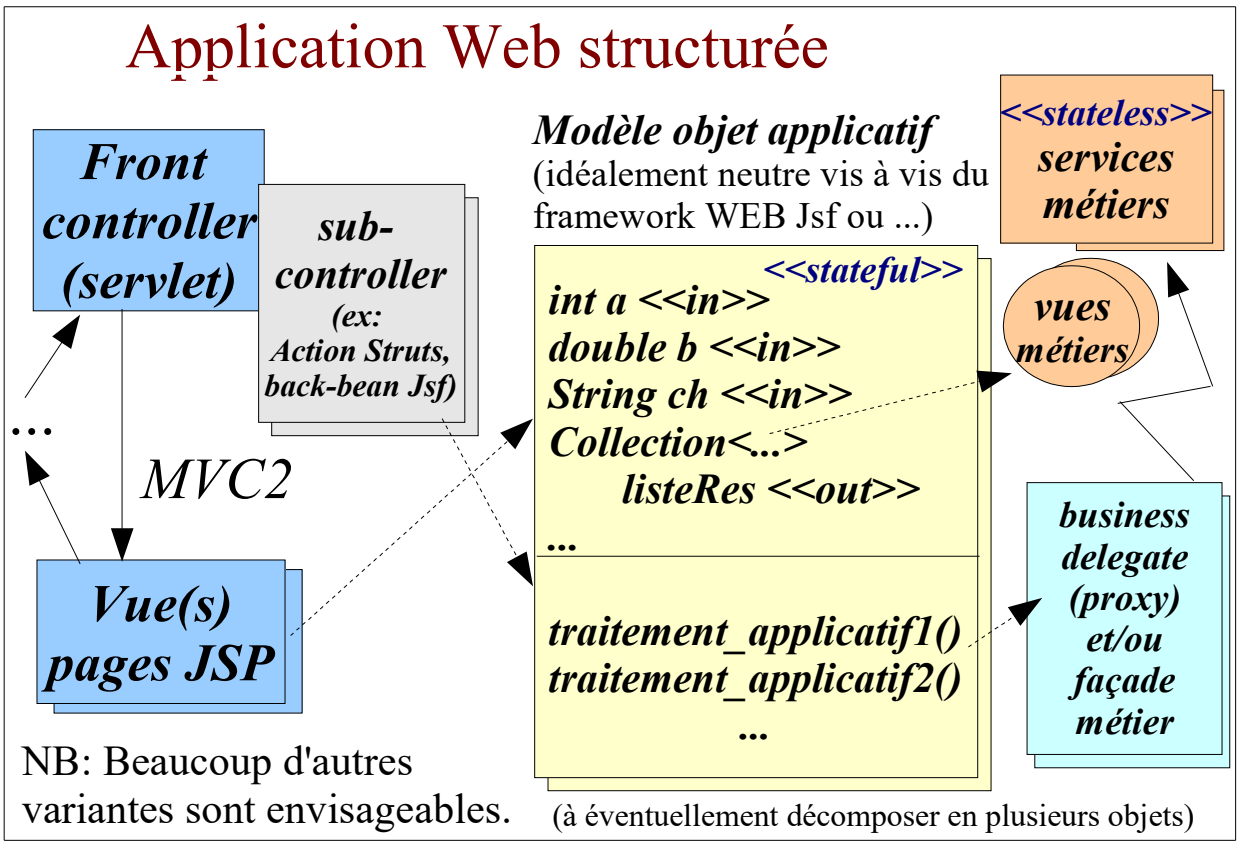
énormément de variantes (ex. MVC 2 pour présentation des applications Web récentes)

Model = modèle objet (structure de données & traitements)

View = IHM (affichage)

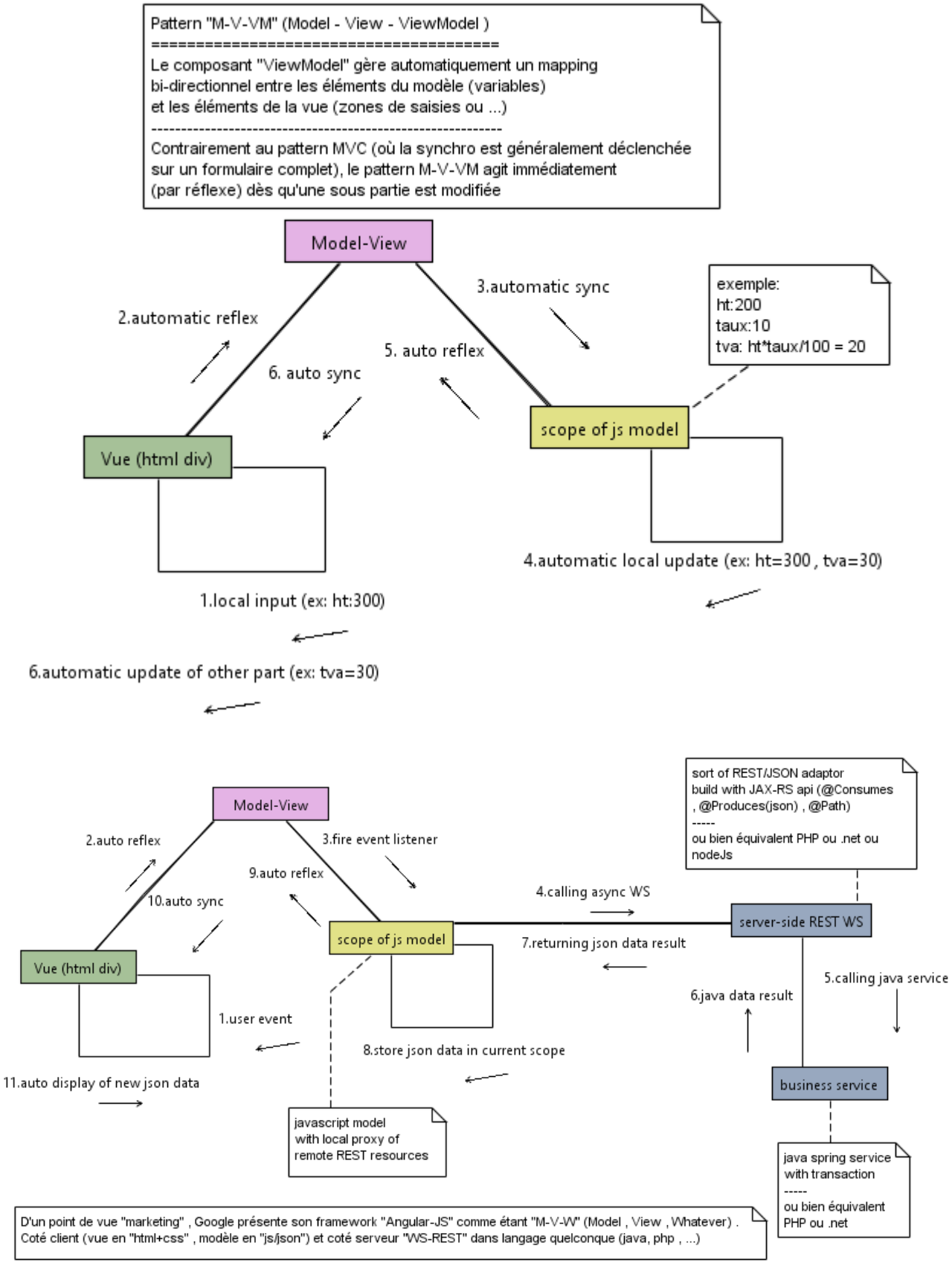
Control = contrôles (navigation, ...)

MVC "Java/web" et distinction "stateless" / "stateful" :



M-V-VM (variante plus atomique du pattern MVC) :

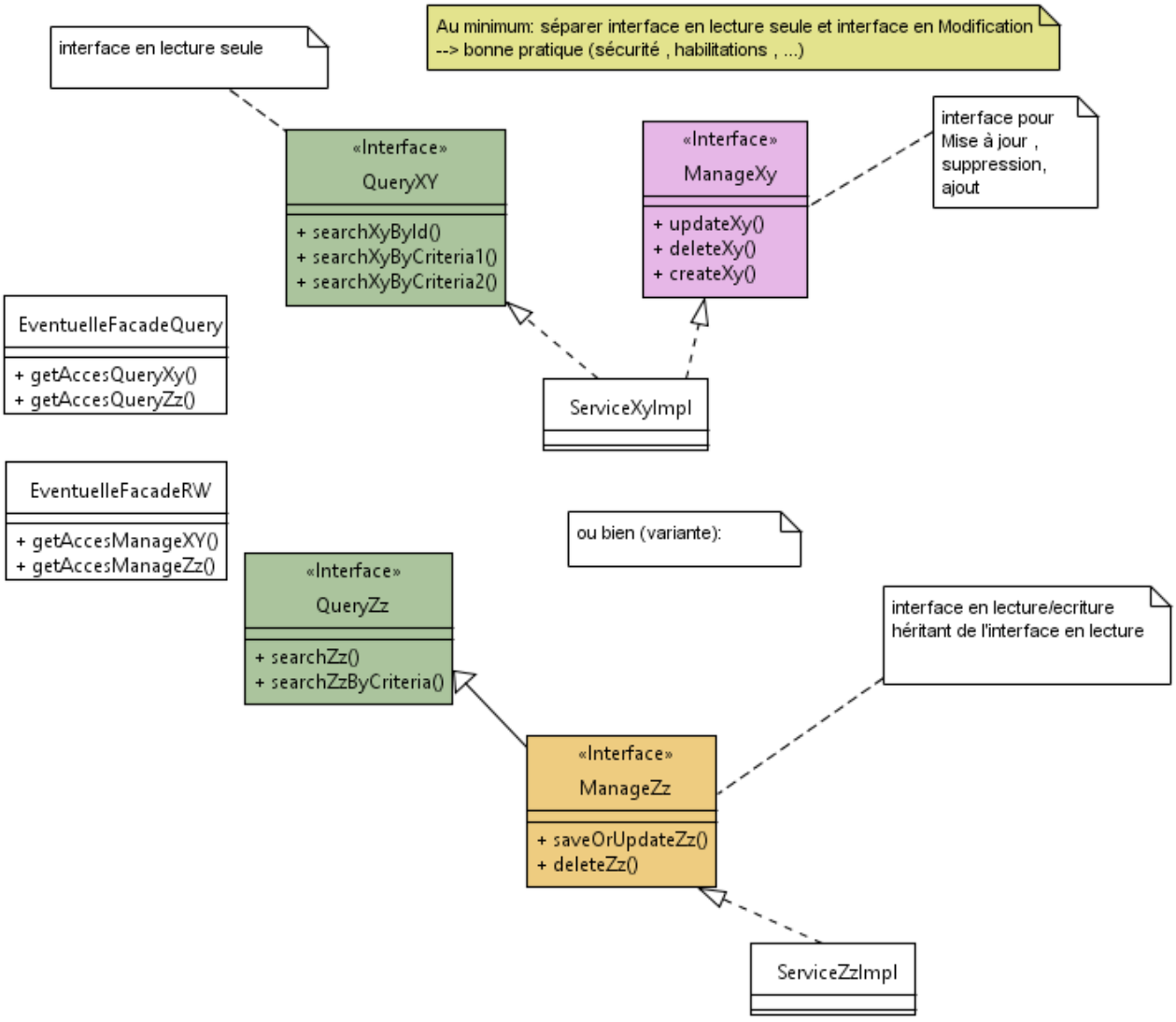
Les **diagrammes de communication UML2** ci dessous montrent le principe de fonctionnement d'un **framework** (ex : ***AngularJs*** , …) **basé sur** le **design pattern "M-V-VM"** :



5. Séparation des interfaces et CQRS

5.1. Séparation des interfaces "Lecture seule" et "Mise à jour"

Bonne pratique générale : **Séparer "interface en lecture seule" et "interface en mise à jour"**



5.2. Architecture CQRS (à adapter)

CQRS = **C**ommand **Q**uery **R**esponsability **S**eggregation/Separation .

Où "***Command***" est à interpréter au sens *design pattern "Command" pour véhiculer des demandes de "mise à jour" (ajout , modification, suppression de "parties")* .

Motivations :

Dans beaucoup de cas (site web , applications) , on effectue :

- un très **grand nombre** d'**opérations en lecture** (gros volume , avec **rapidité souhaitée**)
- seulement quelques opérations en "modifications / mises à jour" avec fiabilité souhaitée (transactions)

d'où **l'idée de** non seulement séparer les interfaces en "lecture" / "mise à jour" mais également de **séparer les implémentations** :

- **programmer les opérations en lecture avec des composants très légers et très rapides** (ex : JDBC direct vers XML ou JSON"
- **programmer les opérations en "modifications / mises à jour" avec une succession de traitements plus élaborés** pour bien gérer les aspects "intégrité , transaction, …) .

En lecture → "***thin query layer***"

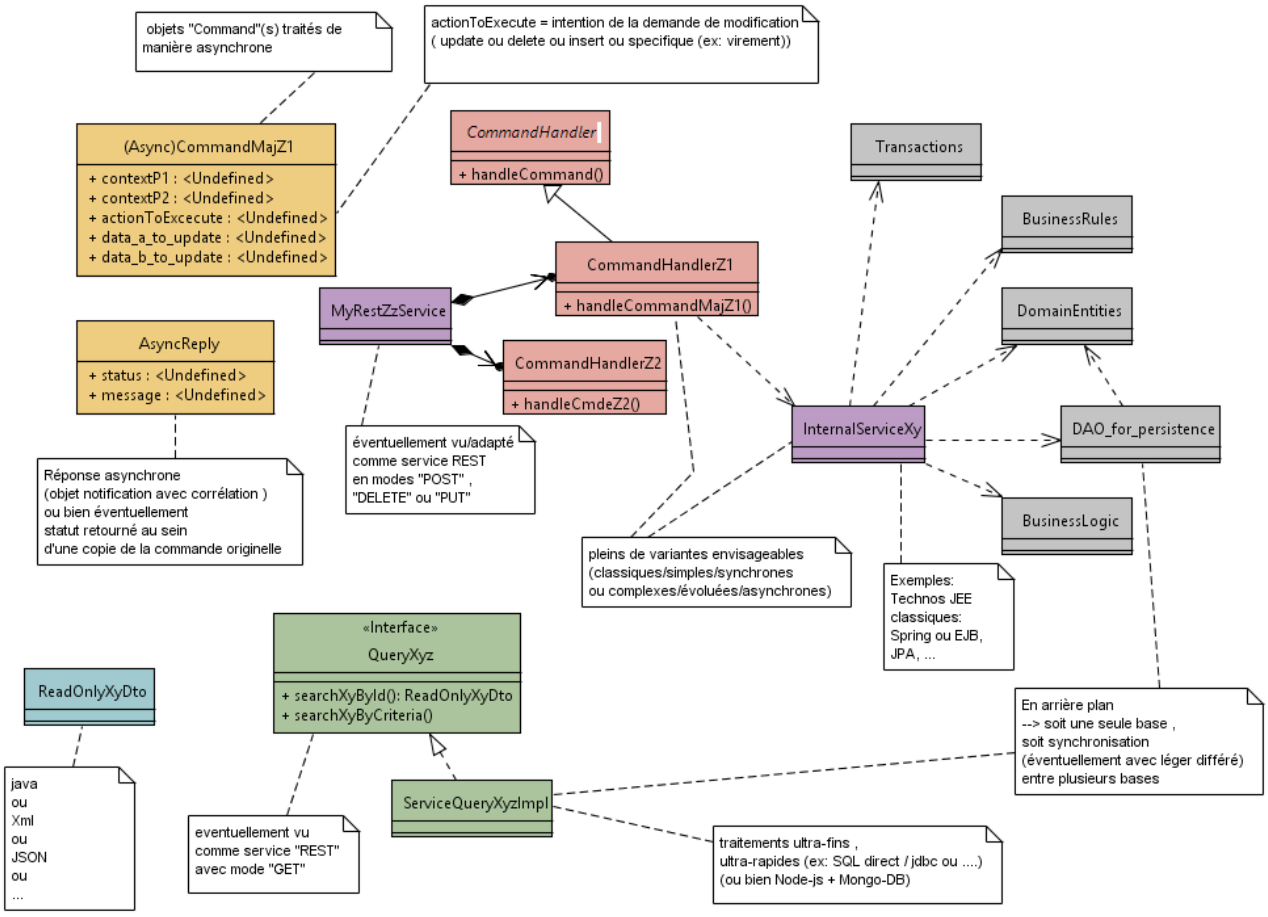
En écriture/modification → **envoi/réception d'un objet "Command" véhiculant** :

- **l'intention de la mise à jour** (ex : "suppression de l'entité" ou "ajout d'un sous élément à rattacher")
- **le contexte** (ex : données existantes pour préciser "où effectuer un attachement de nouvelles parties"
- **les données à mettre à jour** (ou insérer ou supprimer) .

Mise en œuvre :

Telle que souvent documentée sur internet , l'architecture CQRS est assez lourde/complexe .
Il semble préférable d'implémenter CQRS de manière libre et en prenant soin de bien s'adapter au contexte technologique .

Le schéma ci-après propose quelques idées (à débattre , à adapter , …) .


objets "Command"(s) traités de manière asynchrone
actionToExecute = intention de la demande de modification (update ou delete ou insert ou specifique (ex: virement))
(Async)CommandMajZ1
+ contextP1 : <Undefined>
+ contextP2 : <Undefined>
+ actionToExecute : <Undefined>
+ data_a_to_update : <Undefined>
+ data_b_to_update : <Undefined>
CommandHandler
+ handleCommand()
Transactions
BusinessRules
CommandHandlerZ1
+ handleCommandMajZ1()
MyRestZzService
DomainEntities
AsyncReply
+ status : <Undefined>
+ message : <Undefined>
CommandHandlerZ2
+ handleCmdeZ2()
InternalServiceXy
DAO_for_persistence
Réponse asynchrone (objet notification avec corrélation) ou bien éventuellement statut retourné au sein d'une copie de la commande originelle
éventuellement vu/adapté comme service REST en modes "POST" , "DELETE" ou "PUT"
pleins de variantes envisageables (classiques/simples/synchrones ou complexes/évoluées/asynchrones)
Exemples: Technos JEE classiques: Spring ou EJB, JPA, ...
BusinessLogic
«Interface»
QueryXyz
+ searchXyById(): ReadOnlyXyDto
+ searchXyByCriteria()
ReadOnlyXyDto
En arrière plan
--> soit une seule base ,
soit synchronisation
(éventuellement avec léger différé)
entre plusieurs bases
java
ou
Xml
ou
JSON
ou
...
eventuellement vu comme service "REST" avec mode "GET"
ServiceQueryXyzImpl
traitements ultra-fins , ultra-rapides (ex: SQL direct / jdbc ou ....) (ou bien Node-js + Mongo-DB)

VI - Design Principles

1. Présentation des "*design principles*"

Grands principes de conception orientée objet

Une autre série de "design patterns" appelés <<***design principles***>> et élaborés par *Bertrand MEYER* et *Robert MARTIN* permettent (si besoin) d'approfondir certains points et de formaliser un peu plus quelques principes objets fondamendaux.
On parle alors en terme de *principes* ou de *règles* là où jusqu'içi *GRASP* s'exprimait essentiellement en terme de *bonnes pratiques*.

Essentiellement liée à la structure générale des modules , cette série de patterns est tout à fait adaptée à la conception préliminaire.

Préambule (sur le vocabulaire employé dans ce chapitre)

La série des « design principles » utilise intensément le terme « package » à interpréter ici au sens « module » ou « artifact » plutôt que « namespace » .

Autrement dit « package » est ici à comprendre comme « packaging » (ex : archive « .jar ») .

2. Gestion des évolutions et dépendances

Gestion des évolutions et dépendances (1)

OCP (Open-Close P.) / Principe d'ouverture-fermeture:

Un module doit être ouvert aux extensions
mais fermés aux modifications.

LSP (Liskov Substitution Principle) / Principe de substitution de Liskov:

Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir (*substitution parfaite par une sous classe*).

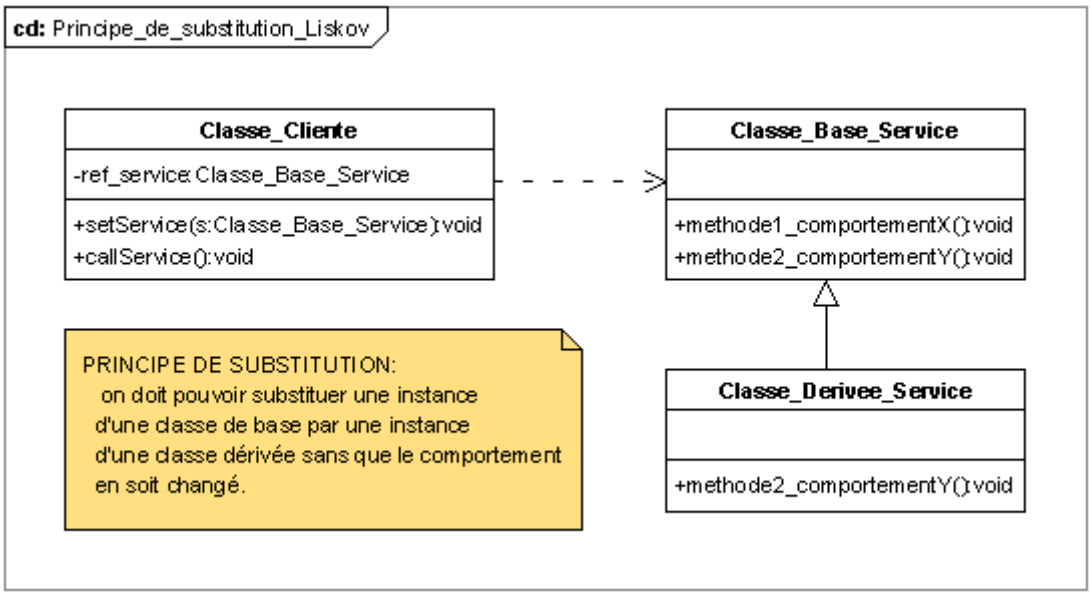
.../...

2.1. OCP et Principe de substitution (de Liskov)

Exemple (ouverture/fermeture):

* Un bon schéma d'héritage (avec niveau abstrait , sous classes concrètes et polymorphisme approprié) est naturellement ouvert à des ***extensions*** potentielles qui prendront la forme de ***nouvelles sous classes***.
* A l'inverse un code basé sur de multiples *if (xxx instanceof Cxx) ...else if(...instanceof Cyy)* est assez fermé car il *doit malheureusement être modifié pour évoluer.*
* D'un point de vue technique le LSP favorise l'OCP.
* D'un point de vue sémantique, *un LSP trop artificiel peut conduire un des implémentations non applicables (avec éventuelles exceptions à gérer)* , ce qui n'est pas mieux qu'un unique test "instanceof" (idéalement lié à toute une sous branche de l'arbre d'héritage).

Un objet d'une classe CX doit normalement pouvoir être substitué par n'importe quel objet d'une sous classe CY sans que le comportement en soit changé.



2.2. Principe d'inversion des dépendances

Gestion des évolutions et dépendances (2)

DIP (Dependency Inversion P.) / Principe d'inversion des dépendances:

A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions

B. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.

ISP (Interface Segregation P.) / Principe de séparation des interfaces:

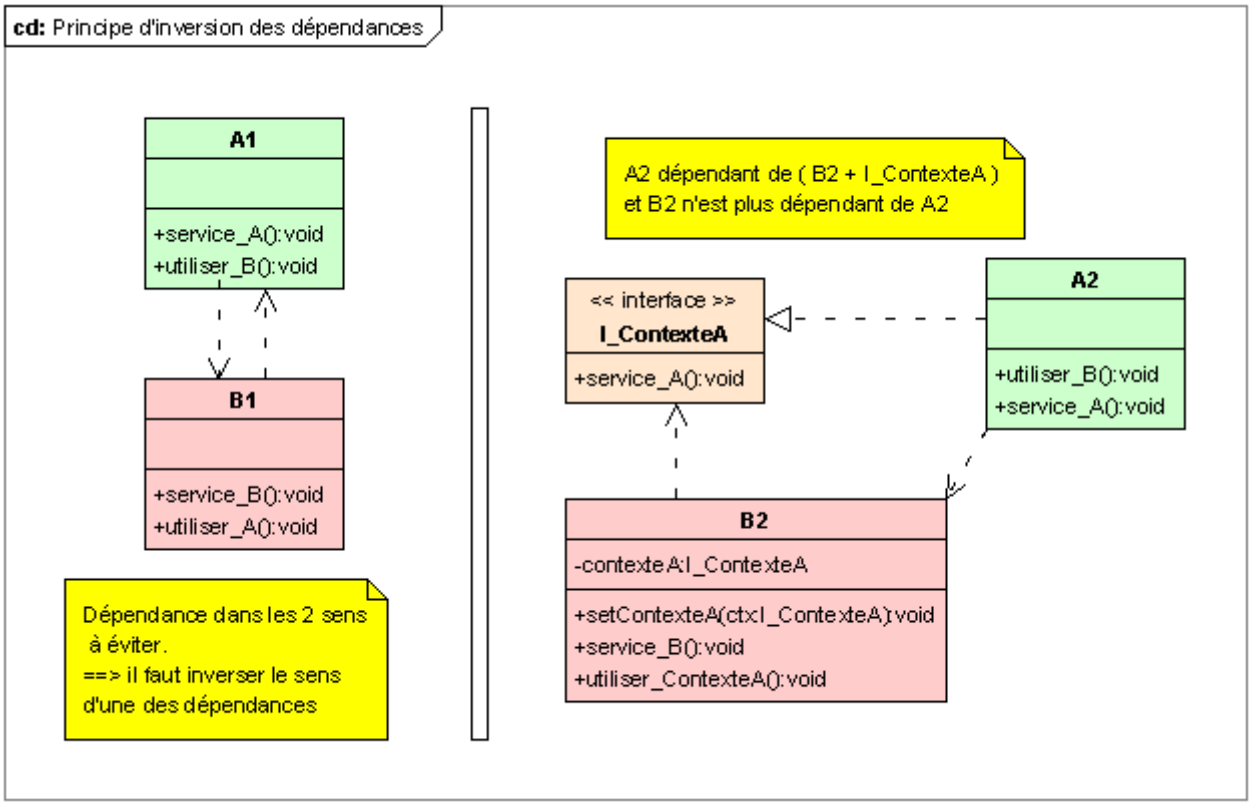
Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.

Autrement dit:

DIP_A : Un module de haut niveau doit toujours voir un service de plus bas niveau de façon abstraite (interface).

Pour qu'un module de bas niveau soit réutilisable par plusieurs modules de haut niveau, il doit interagir avec ceux-ci via des "callback" déclarées sur des contextes abstraits *(ex: méthodes événementielles et "listener")*

ISP : Ne pas hésiter à faire en sorte qu'une classe implémente plusieurs interfaces complémentaires. Les futurs clients ne seront alors dépendants que d'un seul point d'entrée abstrait.



Il vaut mieux éviter des dépendances dans les 2 sens entre deux classes ou packages (ou modules) :

A-->B et B-->A

On introduit alors une interface (I_ContexteA) de type "interface sortante coté B" devant être ultérieurement implémentée par A et on met en place une méthode d'enregistrement de contexte I_ContexteA coté B: ".setContexteA(IA ...) ou constructeur ..." .

Les nouvelles dépendances sont alors les suivantes:

A-->I_ContexteA (en l'implémentant)

B-->I_ContexteA (en l'utilisant)

A-->B

NB:

- L'interface I_ContexeA est censée être packagée coté B (dans le même package)
- Ce principe est très utilisé pour le traitement des événements.

3. Organisation d'une application en modules

Organisation de l'application en modules

REP (Reuse/Release Equivalence P.) / Principe d'équivalence Réutilisation/Livraison:
La granularité en termes de réutilisation est le package. Seuls des packages livrés sont susceptibles d'être réutilisés.

CRP (Common Reuse P.) / Principe de réutilisation commune:
Réutiliser une classe d'un package, c'est réutiliser le package entier.

CCP (Common Closure P.) / Principe de fermeture commune:
Les classes impactées par les mêmes changements doivent être placées dans un même package.

Autrement dit:

Ne pas hésiter à modifier un élément interne (et caché/privé) d'un package car ceci n'a pas d'impact négatif sur la réutilisation du package.

Décomposer s'il le faut un gros package en un ensemble de petits packages pour affiner les dépendances *(pour ne redéployer que ce qui a changé et pour favoriser de futures évolutions)*.

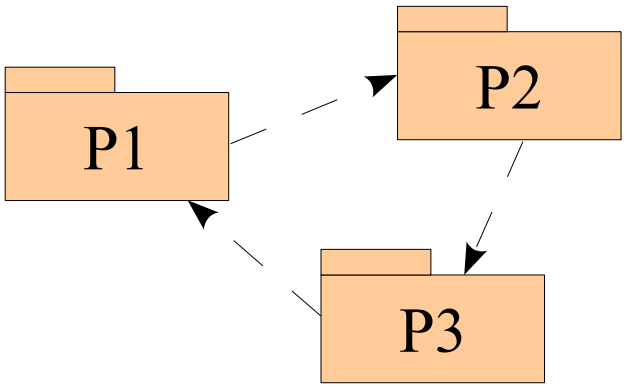
4. Gestion de la stabilité de l'application

Gestion de la stabilité de l'application

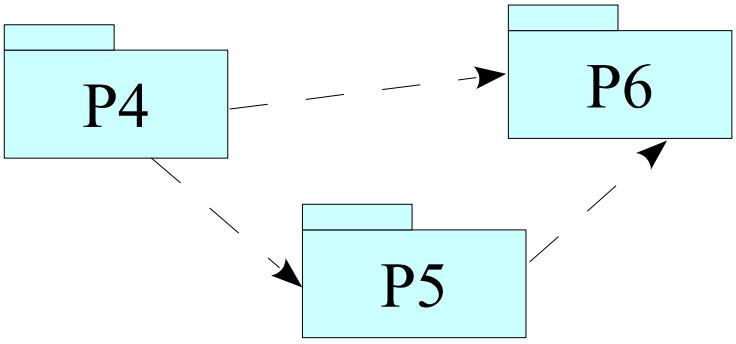
ADP (Acyclic-Depedencies P.) / Principe des dépendances acycliques: Les dépendances entre packages doivent former un graphe acyclique (sans dépendance(s) circulaire(s)).

SDP (Stable Depedencies P.) / Principe de relation dépendance/stabilité: Un package doit dépendre uniquement de packages plus stables que lui.

SAP (Stable Abstractions P.) / Principe de stabilité des abstractions: Les packages les plus stables doivent être les plus abstraits. Les packages instables doivent être concrets. Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.

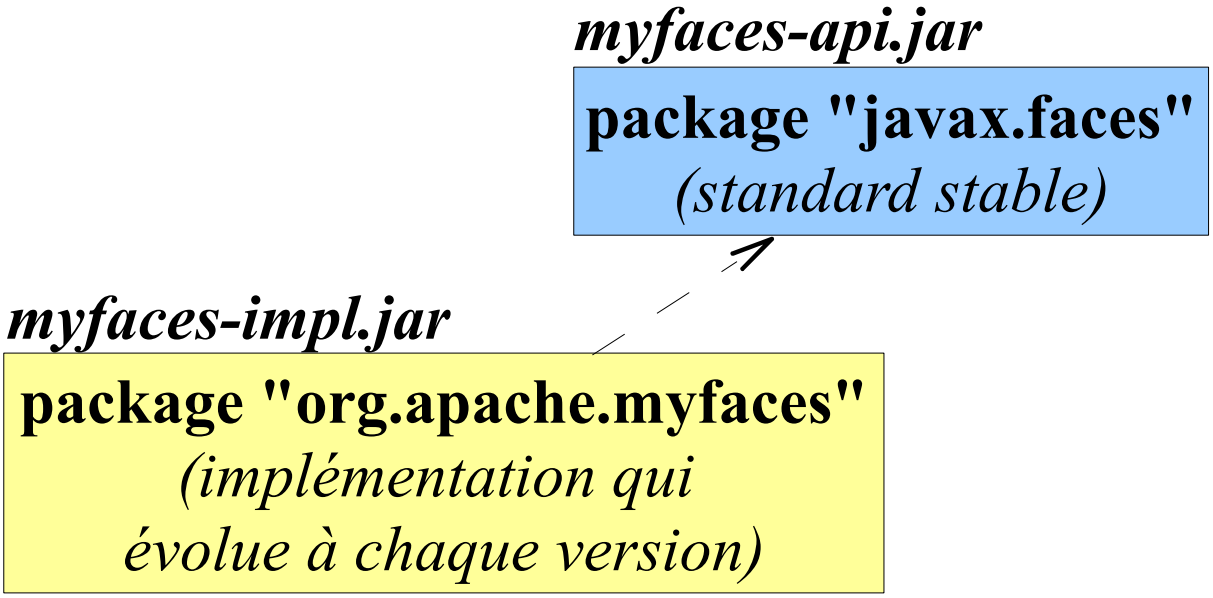


pas bien !



bien !

exemple: implémentation "*myfaces / Apache*" du framework WEB "*JSF*"



Application courante :

client_module -----> **moduleXY_itf**

(implémenté via)

mod_XY_local_impl *ou bien* **mod_XY_delegate_impl** ----> services_distant

(selon ".jar" présent dans le classpath) .

NB : Il existe des "métriques de packages" qui permettent de mesurer le respect des principales règles de "bonne conception orientée objet" .

VII - Design patterns GOF (suite)

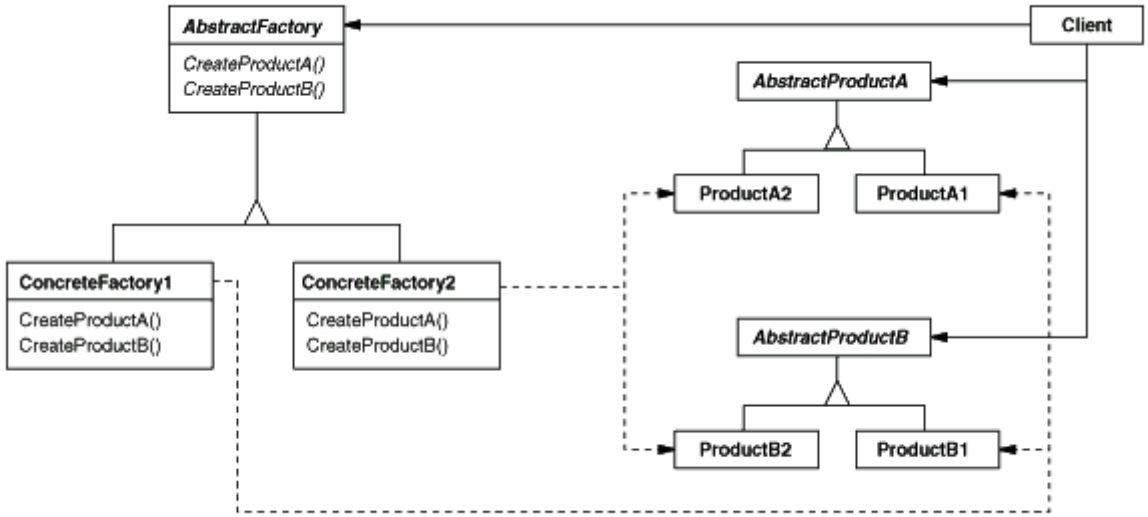
1. Quelques Design Pattern plus spécifiques en détails

1.1. Fabrique abstraite (abstract factory)

Frabrique = objet en fabriquant d'autres . Une classe de fabrique peut elle même implémenter une interface abstraite .

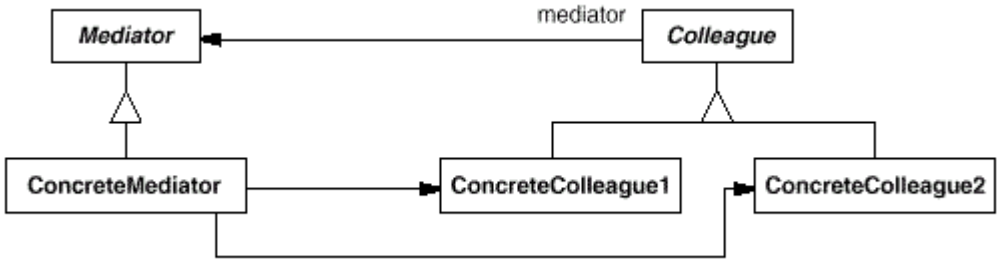
On peut alors envisager une hiérarchie de fabriques (abstraite et concrètes) associée à une hiérarchie de produits (abstrait et concrets).

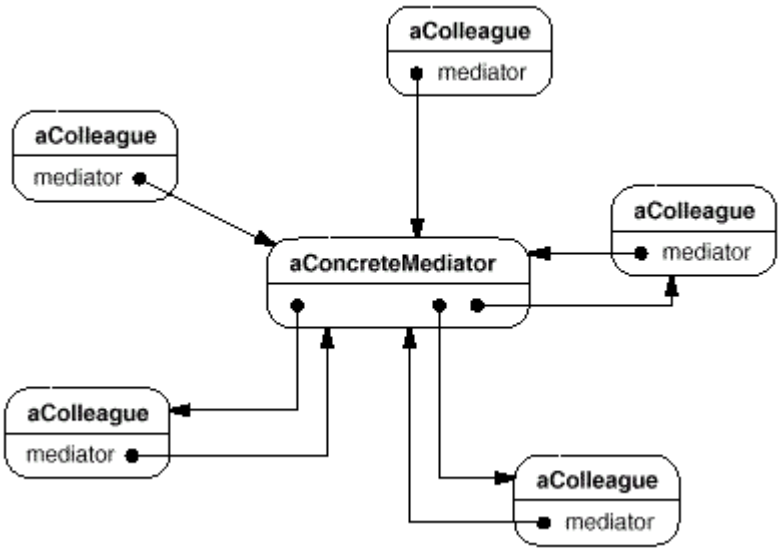
==> en basculant de type de fabrique et en redéclenchant le processus de fabrication , on recréer alors automatiquement toute une famille de produits dans une autre version (ex: autre "look & feel" , autre implémentation , ...).



...

1.2. Médiateur (à peaufiner via le principe d'inversion de dépendances)





==> Evite une multitude de dépendances directes entre collègues.

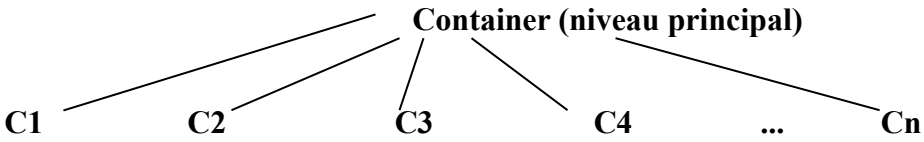
==> Le médiateur (concret) devrait idéalement être injecté de façon abstraite au sein des collègues [principe d'inversion des dépendances / logique événementielle / ...]

1.3. Introduction au D.P. "Observateur" (vues cohérentes sur mêmes données, ...) et logique événementielle

Problème à résoudre: comment rendre facilement cohérentes n vues sur un même document (paquet commun de données) ? ==> une mise à jour partielle d'une vue et du document associé doit être répercutée sur toutes les autres vues.

Autrement dit,considérons que"composant" soit içi un synonyme de "vue" --->

De quelle manière doit procéder un composant C1 pour déclencher des mises à jour cohérentes dans des composants frères C2, C3, ..., Cn ?



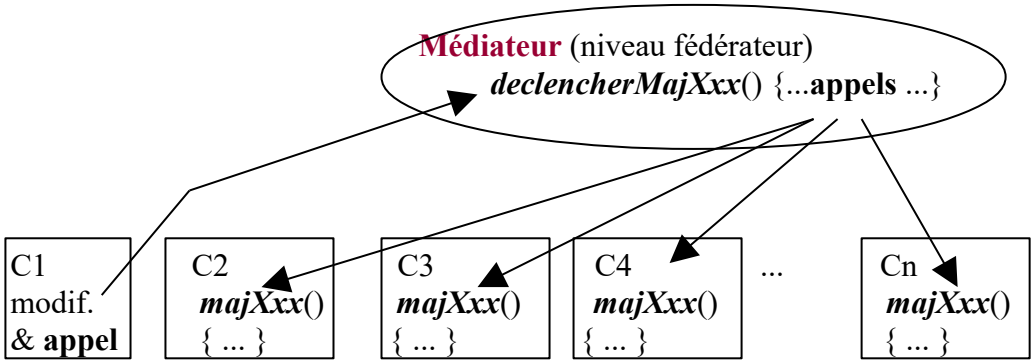
Solution 1 (pas bien !!!)

Suite à un modification interne , le composant C1 appel directement une fonction du genre ***majXxx*** sur chacun des autres composants C2, C3, ... Cn .

Cette solution est assez mauvaise car le composant C1 devient ainsi dépendant des composants C2, C3 et Cn. Si chaque composant fonctionne de cette façon , on se retrouve avec une multitude de dépendances dans tous les sens => le logiciel sera très difficilement maintenable (évolutions laborieuses) . Cette mauvaise architecture est souvent appelée plat de nouilles ou usine à gaz.

Solution 2 (déjà mieux)

Suite à un modification interne , le composant C1 appel une fonction du genre *declencherMajXxx*() sur son conteneur parent (ex: fenêtre mère). Cet élément principal (quelquefois appelé ***médiateur***) va alors réagir en appelant une fonction du genre *majXxx*() sur chacun des autres composants C2, C3, ... Cn .



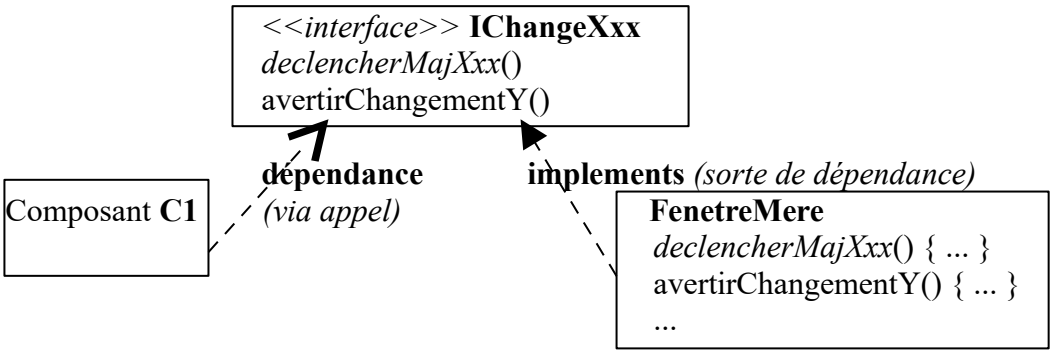
Cette solution est déjà beaucoup mieux structurée car les sous composants C1, C2, ... Cn ne se voient pas directement et sont donc complètement indépendants.

Reste tout de même un problème non négligeable: le composant C1 devient fortement dépendant du niveau principal (conteneur parent) et peut donc difficilement être réutilisable dans un autre contexte.

Solution 3 [composants avec logique événementielle] (bien !!!)

La meilleur des solutions au problème consiste à ***peaufiner la solution 2*** en y inversant la dépendance entre C1 et son conteneur parent (niveau fédérateur) . **L'inversion d'une dépendance s'effectue en introduisant une interface**:

Au lieu d'appeler explicitement la fonction ***declencherMajXxx()*** sur un type précis de conteneur parent , **le composant C1 se contente d'appeler cette fonction sur une chose vague dont le type n'est qu'une simple interface événementielle**.



Pour que ce mécanisme fonctionne, il faut que l'objet précis qui implémente l'interface *(et donc la fonction declencherMajXxx())* ***passe à C1 une référence sur lui même de façon à ce que C1 puisse plus tard y accéder***.
Par exemple une méthode de type ***add/setChangeXxxListener(IChangeXxx obj)*** permet d'enregistrer au sein du composant le (ou les) objet(s) sur le(s)quel(s) il faut déclencher des mises à jour.
On se retrouve donc dans le schéma événementiel classique du langage java (listener, ...).
....

==========

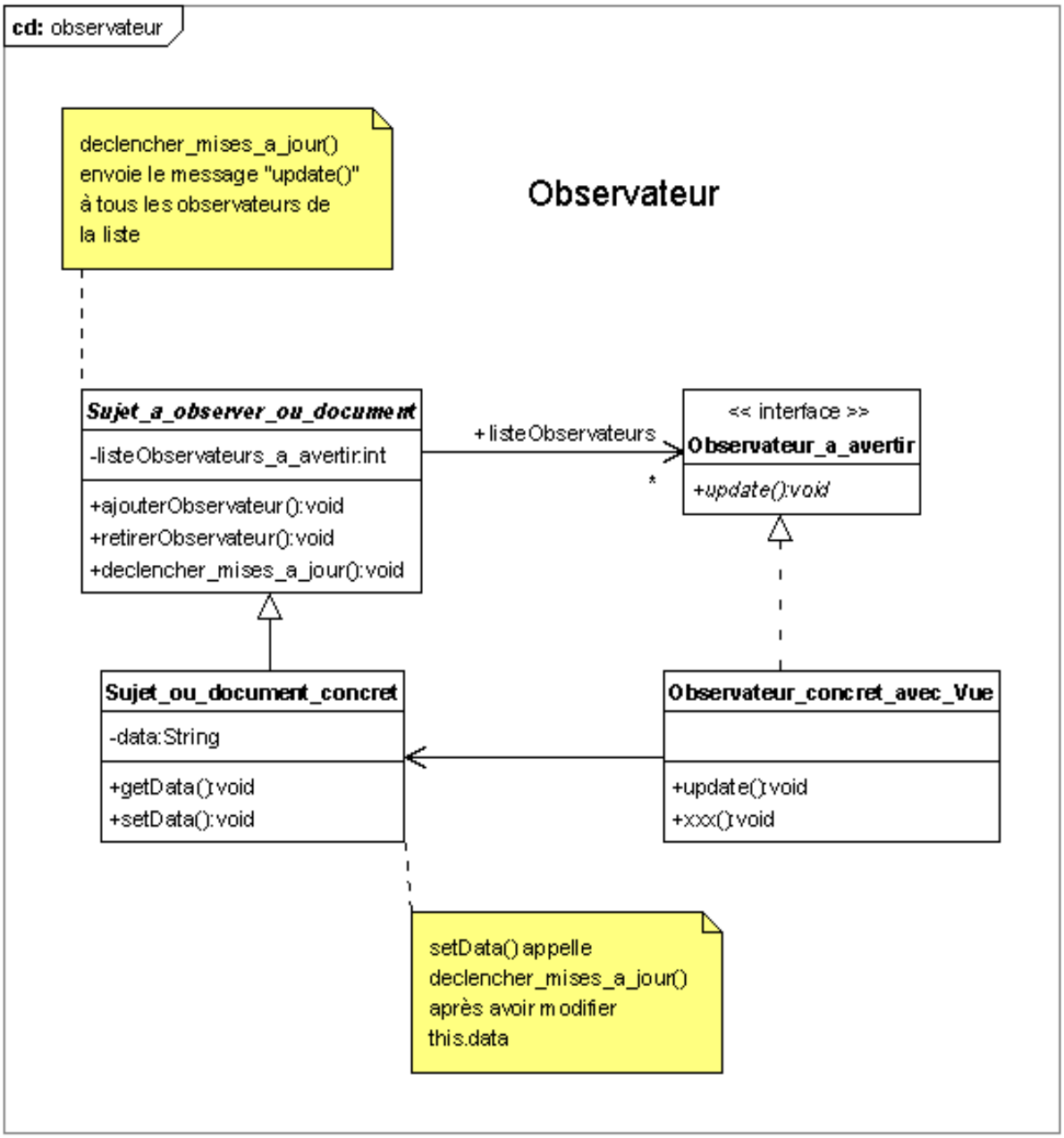
Dans la variante officielle du D.P. "observateur" (G.O.F.) , on ne parle pas en terme de composants devant indirectement déclencher des mises à jours sur d'autres composants mais on parle en terme de :

- **Observateurs** (ou *Vues*) modifiant quelquefois une partie d'un sujet (document) commun .
- **Sujet** (*document*) **observable** avertissant spontanément tous les observateurs qui se sont préalablement enregistrés comme tels qu'une mise à jour est à effectuer (du coté de leurs "vues").

.../...

1.4. design pattern "Observateur":

Variante classique sur les termes : ***Sujet Observable /Observateurs*** ou ***Document/Vues***



Quelques implémentations concrètes :

- le langage **java** comporte la classe java.util.**Observable**
  et l'interface java.util.**Observer**
- le langage **C#** (.net ≥ v4) comporte des **IObservable<T>** et **IObserver<T>** complexes
- Framework **RX-JS** (utilisé dans Angular2)
- …..

Exemple de code (partiel) **java** :

```
...
import java.util.Observable;

public class SubjectWithCommonData extends Observable {
        private String commonData;

        public String getCommonData() {    return commonData;          }

        public void setCommonData(String commonData) {
                this.commonData = commonData;
                this.setChanged();
                this.notifyObservers(); //avertir tous les observateurs qu'un changement
                 // a été effectué et INDIRECTEMENT DECLENCHER DES mises à jour
        }
}
```

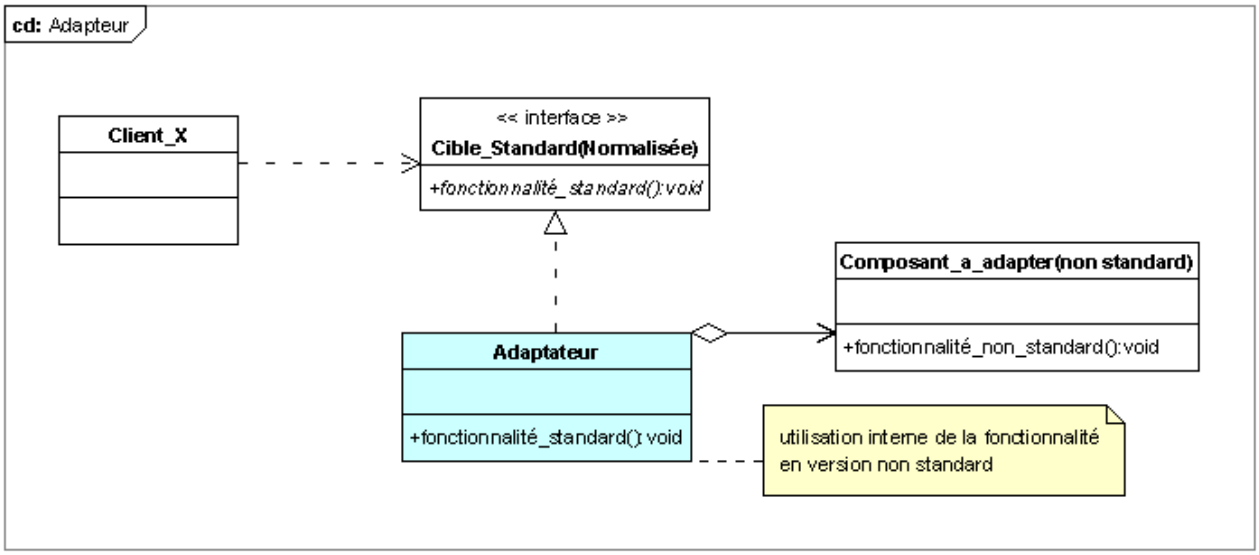
```
import java.util.Observer;  //version prédéfinie "java"
public class MyObserver1 implements Observer{
//...
private SubjectWithCommonData subjectWithCommonData;
        @Override
        public void update(Observable o, Object arg) {
         //arg est une éventuelle indication et o peut être "casté" en "subjectWithCommonData"
                String couleur = subjectWithCommonData.getCommonData();
                //mise à jour locales (au cas par cas)
        }
        public void setSubjectWithCommonData(SubjectWithCommonData subject) {
                this.subjectWithCommonData = subject;
                subject.addObserver(this);
                //POINT CLEF (enregistrement d'un observateur à ultérieurement avertir)
        }
}
```

1.5. Adaptateur

Problème courant: un composant "Ca" doit utiliser "Cx" mais l'interface de "Cx" ne convient pas . D'autre part, les composants "Ca" et "Cx" ont des interfaces figées que l'on ne peut pas modifier.

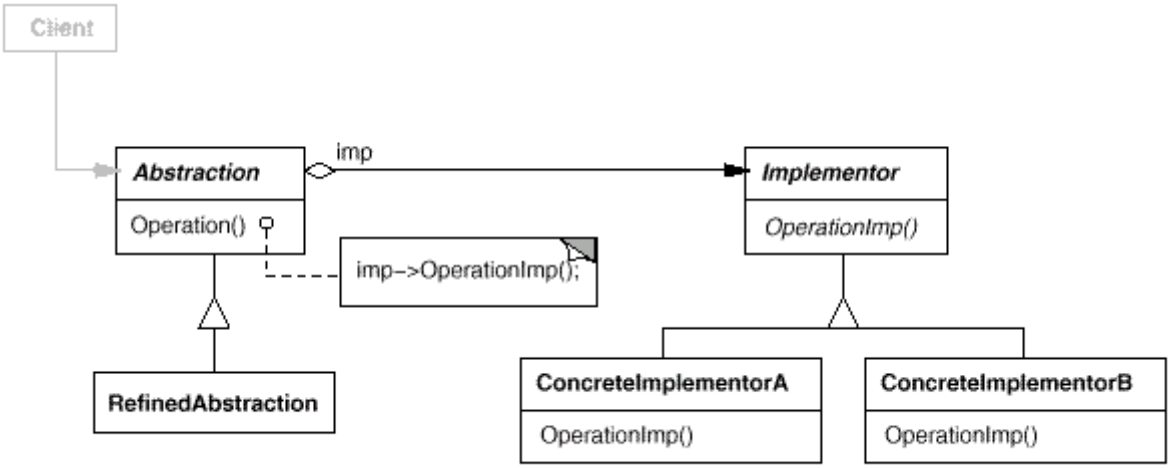
Solution ==> introduire un composant intermédiaire Ci qui va :

- implémenter l'interface attendue par l'appelant "Ca"
- re-déléguer en interne à Cx la plupart des appels/fonctionnalités



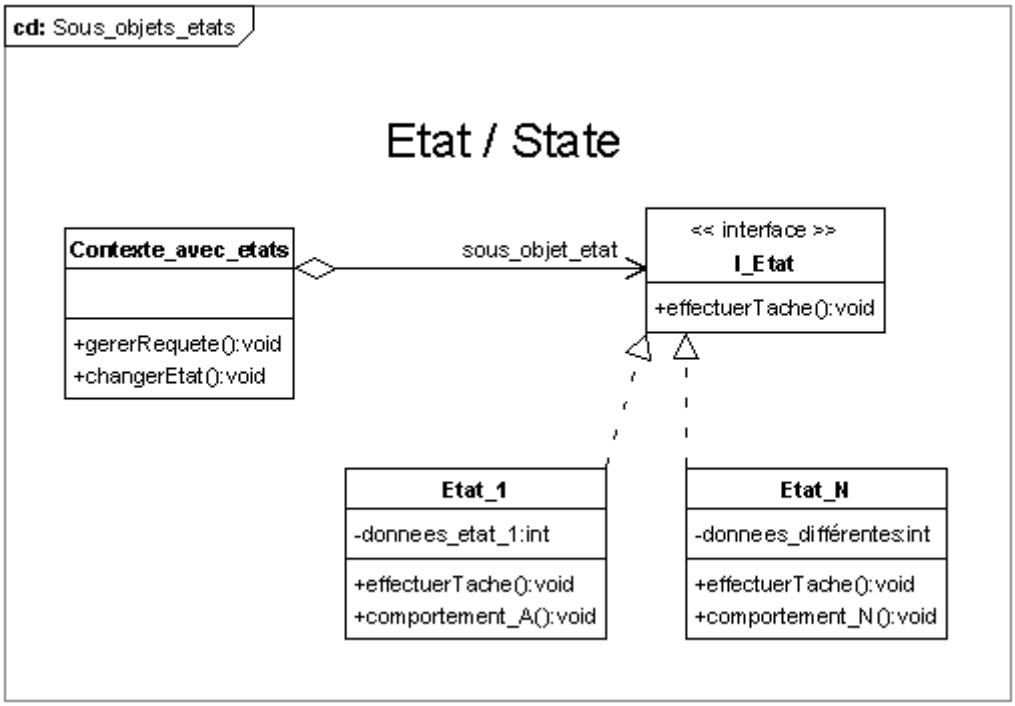
NB : L'adaptateur peut également être implémenté (lorsque c'est possible) via un héritage vis à vis du composant à adapter

1.6. Pont (bridge)



1.7. (sous objet) Etat (State)

Problème courant: un objet doit gérer en interne un certain nombre d'états (avec pour chaque état des comportements attendus différents)



Solution classique: l'objet en question peut matérialiser chaque état comme un sous objet interne spécifique (un changement d'état se résume alors à changer de sous objet interne /switch).

Exemple classique : objet IHM avec différents états (à chaque état peut correspondre un sous objet de type "écran" ou "fenêtre" ou "page HTML" ou ...).

ex1: CardLayout (java.awt)

ex2: diagramme d'états UML pour modéliser les différents états d'une IHM complexe (chaque état : écran ou action[traitement_ou_présentation] , transitions : navigation / conditions à respecter)

1.8. Prototype (à cloner)

Au lieu de créer de nouvelles instances via un code inélégant du genre :

```
switch(type_xy_to_create){
case TYPE_CERCLE :
   obj = new Cercle() ; break ;
case  TYPE_LIGNE :
   obj = new Ligne() ; break ;
case  TYPE_RECTANGLE :
   obj = new Rectangle() ; break ;
....
}
```

On peut créer un ***pool d'instances prototypes*** dès l'initialisation de l'application.

On peut ensuite ranger ces instances dans une ***table d'association*** (pour que chaque type de prototype soit associé à un icône adéquat dans une "**palette**").

L'instanciation d'une nouvelle instance (en fonction du choix courant dans la palette) peut alors se faire via un code simplifié ressemblant à :

```
current_icone = evenement.getSource() ;
obj = mapPrototypes.get(current_icone).clone() ;
```

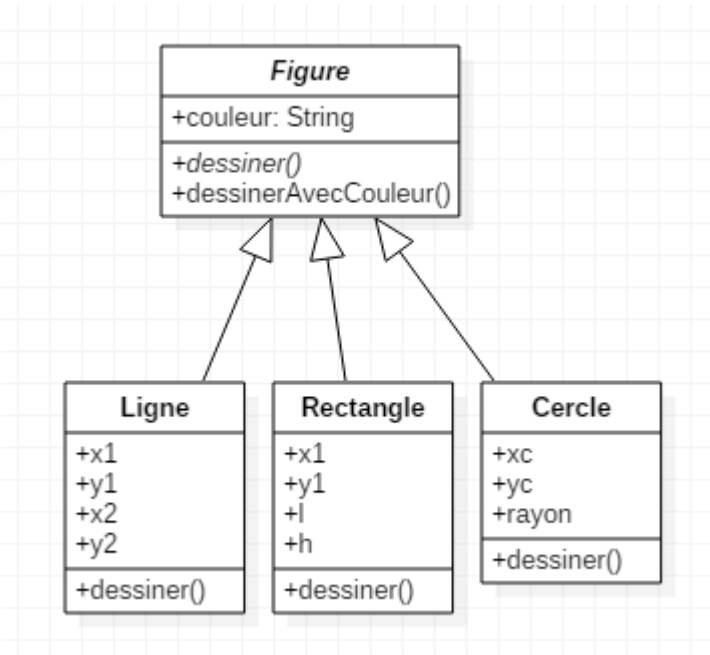
NB : La méthode .**clone**() est prédéfinie sur la classe racine "**Object**" du langage java.

Cependant, le clonage automatique n'aura pas toujours le niveau de profondeur souhaité (si l'on ne reprogramme rien). On sera quelquefois amené à redéfinir la méthode ".clone()" sur certaines classes.

1.9. Visiteur (actif)

Le "visiteur" est un design pattern assez complexe qui est cependant quelquefois très intéressant si l'on souhaite décomposer un objet (ou une hiérarchie d'objets) en au moins deux parties complémentaires bien distinctes de façon à bien séparer certaines responsabilités (découplage) .

Prenons l'exemple d'un programme de dessin vectoriel. La hiérarchie de classe intuitive qui vient au premier abord à l'esprit est souvent du type suivant :



Bien qu'opérationnelle , cette hiérarchie unique à un potentiel inconvénient :

Elle n'est réutilisable que dans le cadre d'une technologie d'affichage bien déterminée (ex : affichage AWT/SWING en java) mais ne pourra pas être facilement réutilisée dans un contexte d'affichage différent (ex : SVG , ….).

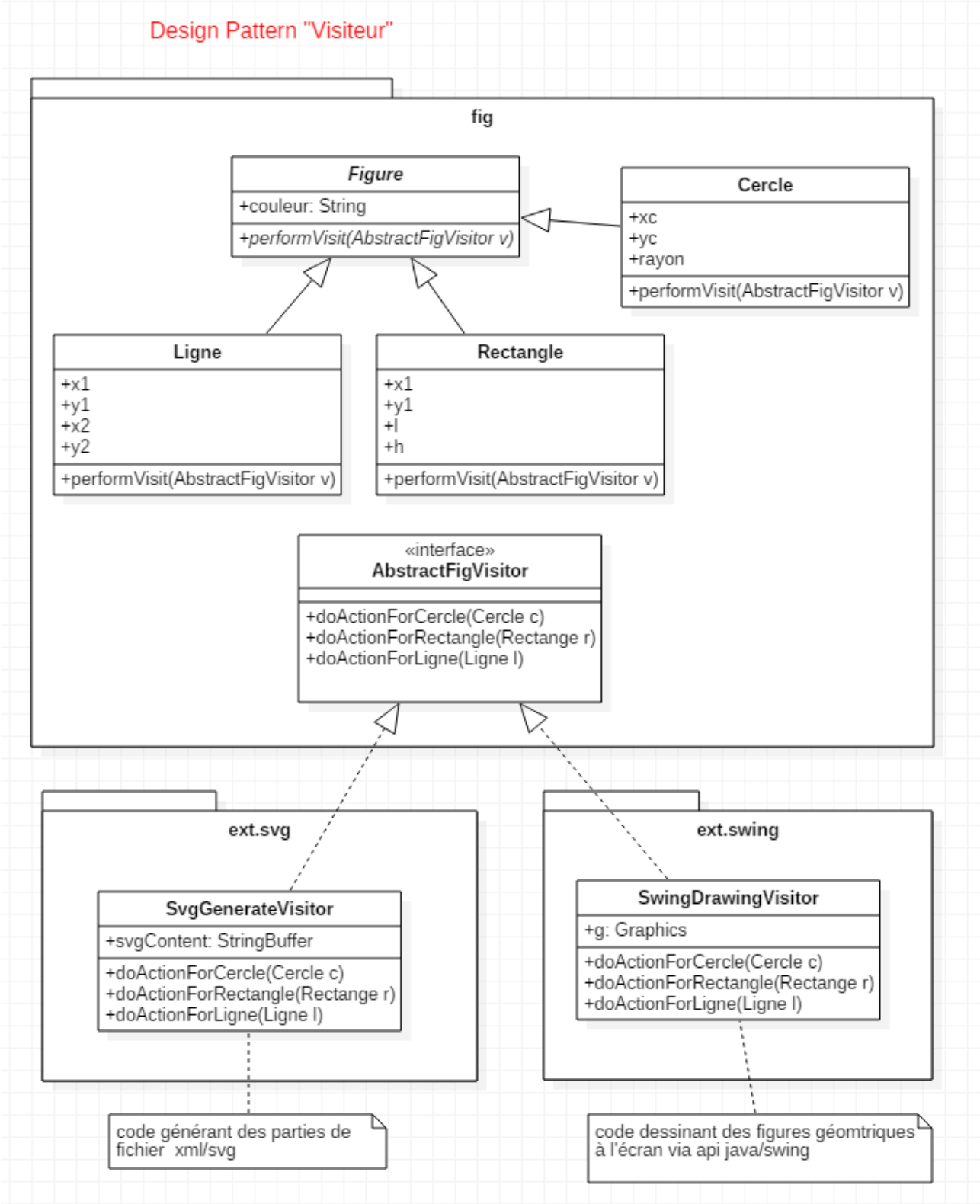
Pour obtenir une meilleur ré-utilisabilité, il est fortement conseillé de découpler les deux responsabilités suivantes :

- gestion des coordonnées (hiérarchie principale)
- gestion des affichages/rendus (avec éventuellement différentes variantes).

Pour atteindre cet objectif, le design pattern "Visiteur" consiste à :

1) Définir une classe ou interface de "***Visiteur abstrait***" permettant d'***effectuer de futurs actions (quelconques)*** **sur tel ou tel type d'élément concret** de la structure (hiérarchie) principale.
[ex : interface "***AbstractVisitor***" avec méthodes "***doActionForConcreteElementA***(a)" , "***doActionForCercle***(c)" , "***doActionForRectangle***(r)" , …]
2) Ajouter une méthode polymorphe de type "***performVisit***(visitor)" ou "***accept***(visitor)" que l'on redéfinira sur chaque classe d'élément concret avec un code type ressemblant à :
performVisit(AbstractVisitor visitor){
visisitor.doActionForCercle /*ou Rectangle ou ….*/ (this) ;
}
3) Définir par exemple une classe "***Renderer2D***" héritant de "***AbstractVisitor***" et implémentant chacune des méthodes abstraites "***doActionForXxxx***(x)" avec le code d'affichage adéquat *(ex : g.drawLine(ligne.getX1() , ligne.getY1() , ligne.getX2(), ligne.getY2()) ;)*
4) Parcourir en boucle tous les éléments *(ligne, rectangle ou cercle avec coordonnées)* de la structure *(ex : dessin)* en invoquant sur chaque élément la méthode ***performVisit***() à laquelle on passera en argument une instance du visiteur concret "Renderer2D". Ceci devrait normalement déclencher tous les affichages nécessaires.

Point technique clef du design pattern "Visiteur" : re-propager automatiquement le comportement "polymorphe" intrinsèques de certains objets sur des éléments externes (les visiteurs) . Le visiteur (en tant qu'extension externe ou "plugin") pourra ainsi effectuer une action appropriée (vis à vis du type exact d'objet visité) sans avoir besoin d'effectuer le moindre "if(... instanceof ….) ".



1.10. Mémento (mémorisation d'état)

Un objet "**Mémento**" est un **objet secondaire qui sert à mémoriser l'état d'un objet** de façon à le ***restaurer*** (si besoin) par la suite ou bien de façon à pourvoir effectuer des ***comparaisons*** entre l'état courant et un ancien état .

1.11. Commande (déclenchement uniforme d'actions)

Un objet "Command_XY" est un petit objet intermédiaire qui encapsule de façon uniforme le déclenchement d'une certaine action spécifique .

Une classe ou interface "Command" abstraite pourra par exemple comporter une méthode qui s'appellera toujours ".execute()" ou bien un couple de méthodes ".do() et .undo()" .

Chaque sous classe concrète (ex : CommandInsert , CommandDelete , ….) redéfinira la méthode ".execute()" en déclenchant une action bien précise .

On pourra éventuellement envisager une méthode ".undo()" symétrique de ".do() ou .execute()" permettant de déclencher une action contraire pour défaire ce qui a été fait lorsque c'est encore possible.

---> en stockant dans une pile les "commandes/actions" lancées par l'utilisateur on pourra alors effectuer une série de "do" ou "undo" (de types exacts éventuellement différents).

On pourra souvent relier (de façon événementielle) une même instance d'un objet "Command" aux différents éléments d'une interface graphique qui sont prévus pour déclencher une même action (ex : "bouton poussoir" , "menuItem" , "icône" d'une "toolbar" , …).

Comme pour tout design pattern, on peut envisager différentes façons de coder une "Commande".

Le critère important (à idéalement prendre en compte) est un "contexte applicatif" que l'on peut souvent associer à une commande (ex1 : référence sur objet courant à insérer ou à supprimer , ex2 : coordonnées d'un élément à créer/dessiner , …).

De ceci, peut éventuellement découler deux types de "Commandes" :
* des commandes brutes (ex : New , Save , Close , Quit , Delete , Insert , ….) directement associées aux éléments de l'IHM (sans contexte) *[une instance de chaque type suffit]*.

- des commandes élaborées (de second niveau) "avec contexte" (ex : DeleteObject avec refObj en tant que propriété interne) *[plusieurs instances sont souvent nécessaires pour encapsuler les différents états contextuels]*.
-

1.12. Chaîne de responsabilités (responsable(s) auxiliaire(s) en "backup")

Client appelant "opXy()" sur A1 avec A1 chaîné avec A2 lui même chaîné avec An .

A1() peut déléguer l'opération "opXy()" à A2 et ainsi de suite sans que :

* le Client se rendre compte de "quoi que ce soit"

* le Client connaisse la longueur de la chaîne (aucun parcours en boucle n'est nécessaire)

Exemple concret (présenté par le "gof") :
Une information d' aide contextuelle ….

Autre exemple concret et sémantiquement proche: **ACL** (Access Control List)

VIII - Autres "design patterns"

1. Initialisation tardive (LAZY) & "Proxy-ing"

Au lieu d'initialiser un composant (avec toutes ses valeurs) dès le départ, il est quelquefois pratique de pouvoir reporter l'initialisation complète des valeurs jusqu'au moment où l'on souhaite y accéder ou bien l'utiliser .

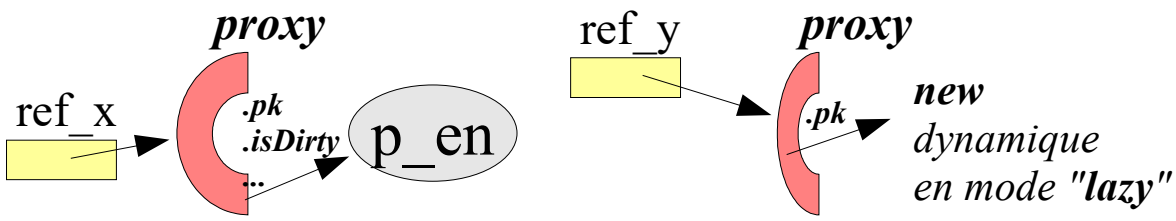
Ceci permet d'éviter des initialisations complètes inutiles (dans le cas où un objet/composant n'est jamais utilisé par la suite).

Ce design pattern est beaucoup utilisé au sein des frameworks "ORM" (Hibernate , JPA , ...).

Attention tout de même aux effets de bords:

> L'initialisation (trop) tardive est quelquefois rendue impossible (suite à une fermeture de connexion) !!!!

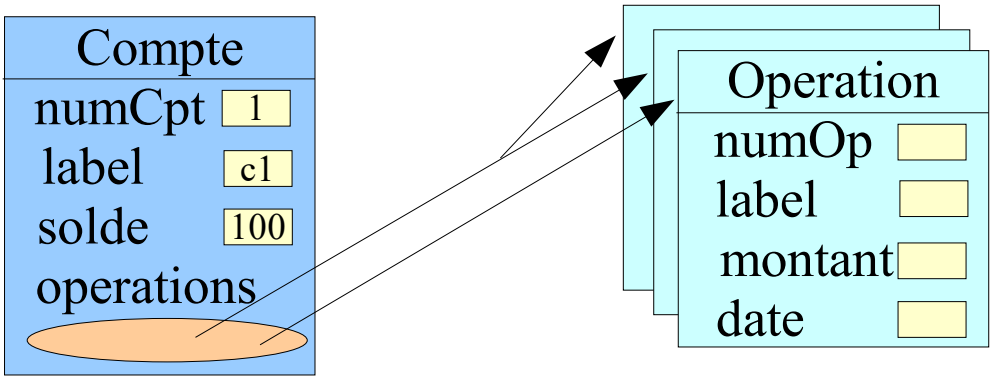
Proxy-ing et persistance



Le "**proxy**" *(généré par cglib ou autre)* sert entre autres à
- ***marquer les entités comme devant être mises à jour*** en base *(isDirty=true)* lors du prochain "***.flush()***"
- remonter les entités en mémoire de façon différée (***lazy***).

ref_persistant = entityManager.***merge***(ref_détaché);
ref_proxy_sans_entité = entityManager.***getReference***(class,pk);

FetchType (Lazy & Eager)

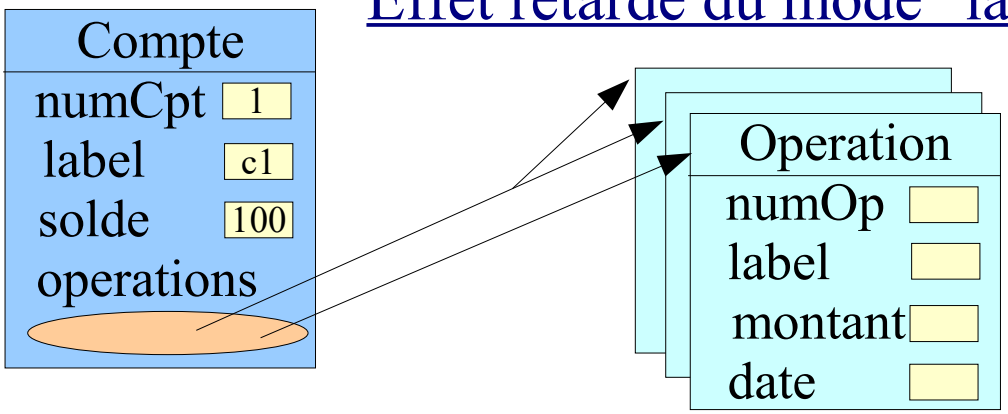


Suite à "***select c from Compte as c where c.numCpt=1***" , remonte(nt) alors immédiatement en mémoire:

- ***uniquement le compte1 en mode "lazy"***
- ***le compte1 et toutes ses opérations en mode "eager"***

(via un 2 ème "select" SQL implicitement déclenché sur l'ensemble des opérations associées au compte1)

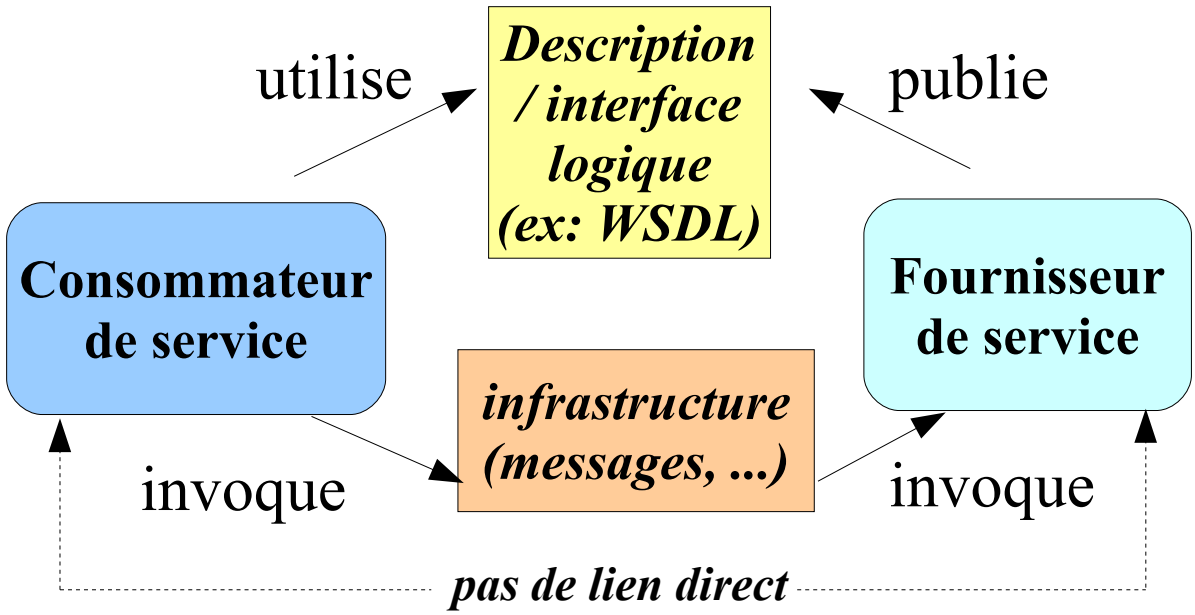
Effet retardé du mode "lazy"



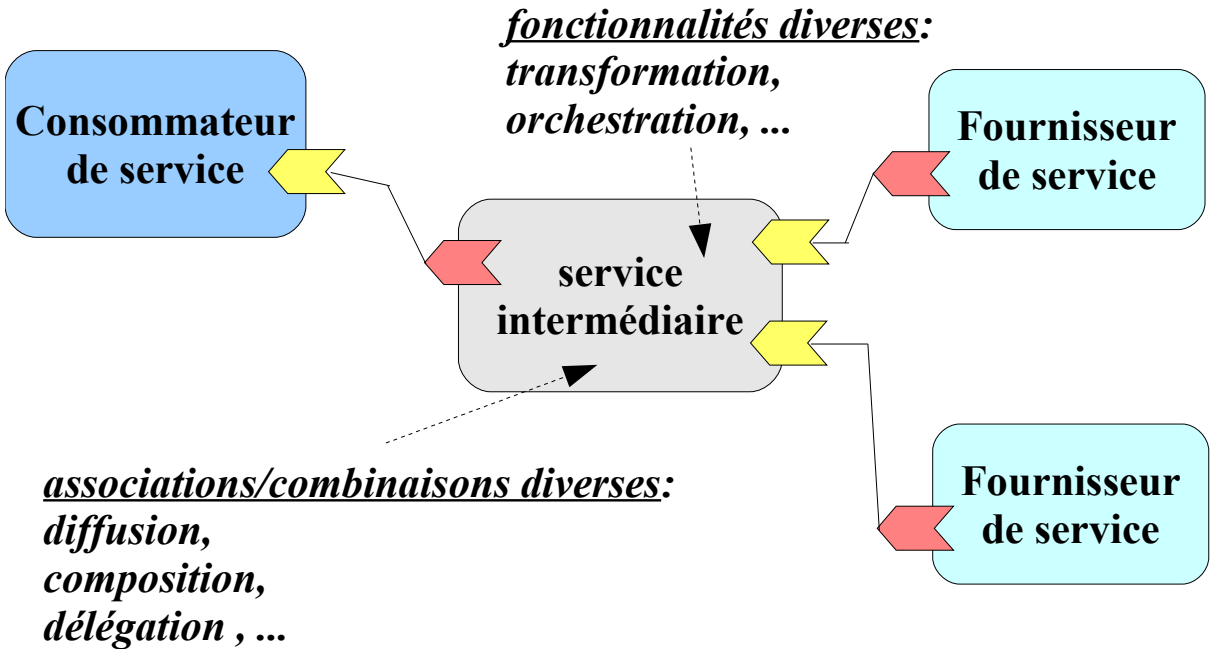
- Si l'on accède qu'à "c1.getLabel() et c1.getSolde() les opérations ne sont alors jamais remontées en mémoire.
- Par contre, ***dès le parcours de la collection des opérations*** *(via un itérateur ou une boucle for)*, ***Les entités "Opération" sont alors automatiquement remontées en mémoire*** par JPA ***via une série de petits "select"*** SQL (jusqu'à "***1+N***" en tout").

2. Producteur/Consommateur (découplés)

SOA : Combinaison de services sur le mode "fournisseurs/consommateurs faiblement couplés"



SOA : Services intermédiaires = fournisseurs et consommateurs.



3. Patterns eip

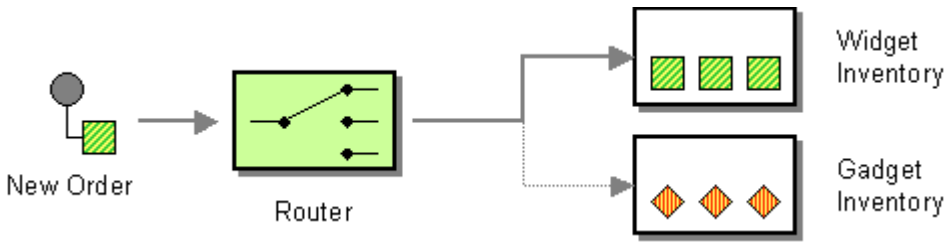
EIP signifie "***enterprise integration pattern***".

Cette série de design patterns concerne essentiellement SOA et le paramétrage des ESB (Petals , ServiceMix , OpenESB, ...)

patterns	*caractéristiques*
Content-based router	Router les messages selon leurs contenus (selon tests "xpath")
Message filter	Filtrer les messages entrant (en mode "InOnly")
Pipeline	Pont entre ["In-Only" Request , "In-Only" response] et "InOut"
Static recipient list	Diffuser des messages "in-only" vers une liste de destinaires
Static routing slip	Router les messages entrants (en mode in-only) vers une série ordonnée de services.
Wire Tap	Intercepteur (en mode "proxy") envoyant une copie en "InOnly" des message "input" ou "output" ou "fault" vers un "listener" . Tous les MEP (in-only, in-out, ...) sont supportés.
Xpath Splitter	Découper un grand message en plusieurs petits messages (xpath)
Aggregator	Recomposer un grand message à partir de plusieurs parties
Content enricher	Enrichir le contenu d'un message via un service annexe d'extraction de données supplémentaires
Resequencer	Réordonne (re-séquence) une série de messages en in-only .
AsyncBridge	Pont inverse vis à vis de "Pipeline" (de "InOut" vers 2 "InOnly") avec erreur si dépassement timeout et id_correlation .

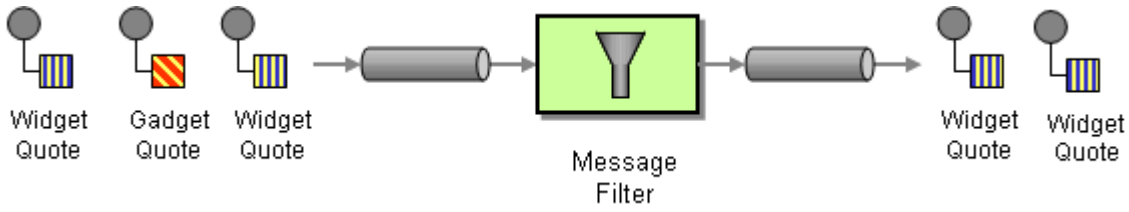
3.1. Content-Based Router

ContentBasedRouter : Router les messages selon une partie de leur contenu .
Sorte de switch/case avec conditions exprimées avec xpath .



3.2. Message Filter

MessageFilter permet de filtrer les messages entrants (en mode in-only) . Certains messages seront supprimés et jamais acheminés vers le service cible. InOut Mep incompatible .

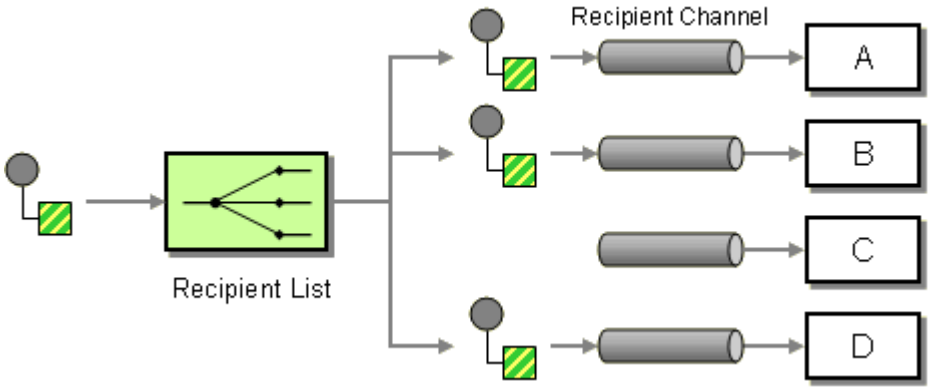


3.3. Pipeline

Le composant "Pipeline"constitue un pont entre "In-Only (or Robust-In-Only) MEP" et "In-Out MEP". Lorsque le "Pipeline" reçoit un message en "In-Only MEP", il renvoie le message entrant en "In-Out MEP" vers une destination de transformation/traitement ("transformer") et relaie ensuite la réponse en "In-Only MEP" vers une destination cible

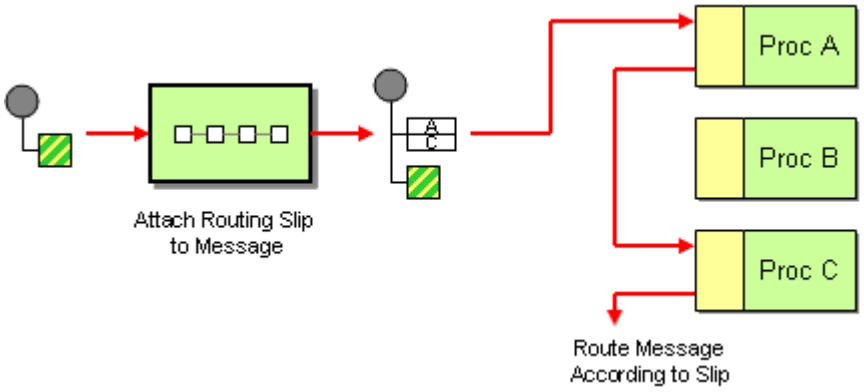
3.4. Static Recipient List

Le composant "StaticRecipientList" sert à relayer des messages entrants (en mode "In-Only" ou "Robust-In-Only") vers une liste statique de destinataires ("recipients").



3.5. Static Routing Slip

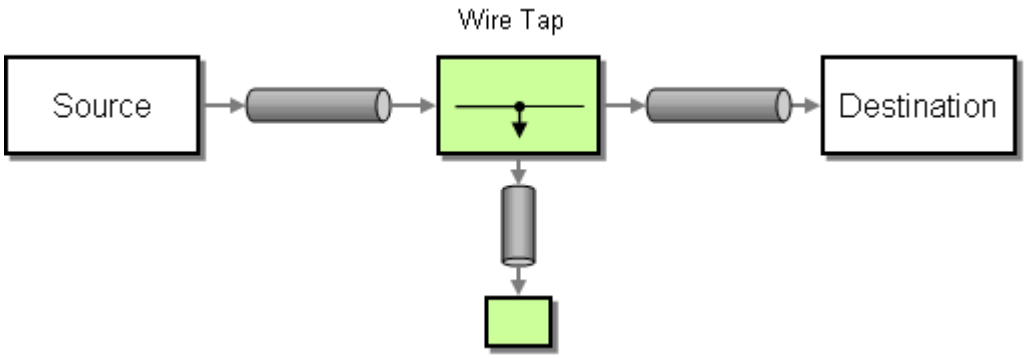
Le composant "RoutingSlip" peut être utilisé pour router des messages entrants (en mode "InOut") vers une série ordonnées de services cibles .En cas d'erreur (ou "faults") à un niveau quelconque de la chaine le processus de routage est interrompu et le message "faults" est renvoyé à l'origine.



3.6. Wire Tap

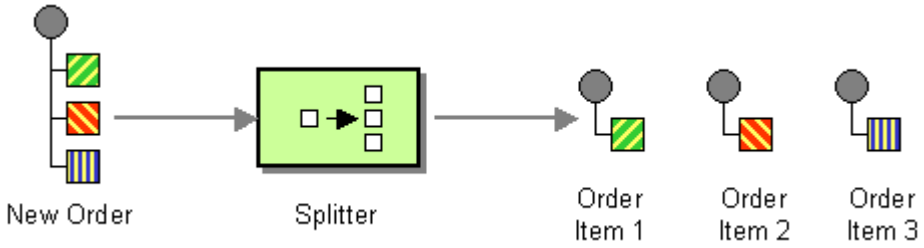
Le composant WireTap peut être utilisé pour relayer une copie d'un message entrant vers un autre composant intéressé ("listener") et ceci de façon transparente vis à vis du composant cible (en mode "proxy").

Entre source et destination cible, tous les "MEP" sont supportés et les copies seront envoyées en "In-only" vers le "listener".



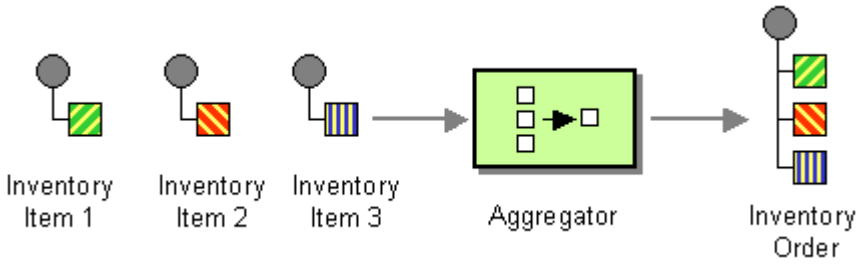
3.7. XPath Splitter

Le composant "XpathSplitter" découpe un message xml entrant en plusieurs messages xml sortants (et acheminés vers une cible "target") selon l'expression d'un chemin xpath .



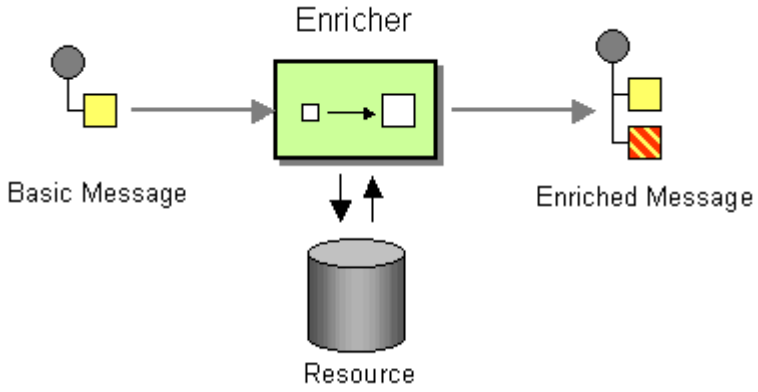
3.8. Aggregator

Le composant "SplitAggregator" sert à recomposer un grand message en rassemblant une liste de petits messages . Cette recomposition est effectuée selon plusieurs propriétés des échanges (count, index, correlationId).



3.9. Content Enricher

"Content Enricher" permet d'extraire des informations complémentaires via un service annexe d'extraction de données supplémentaires. Ceci permet d'enrichir le message retourné par le service cible (ex: ajouter "userName' en fonction de 'userId').



3.10. AsyncBridge

Le composant "AsyncBridge" assure une correspondance (pont) inverse vis à vis du "Pipeline": il permet de décomposer un échange "InOut" en 2 échanges "InOnly" (Requête et réponse attendue) . Une erreur sera générée et retournée si dépassement d'un timeout.
Un id de corrélation est automatiquement attribué , véhiculé sous forme de propriété du message JBI et fait l'objet d'une comparaison pour associer les réponses aux requêtes asynchrones.

Il est également possible de personnaliser l'id de corrélation en précisant les choses à extraire et comparer

ANNEXES

IX - GRASP

1. Affectation des responsabilités (GRASP)

Affectation/répartition des responsabilités (patterns ***GRASP*** de *Craig Larman*)

GRASP = ***G****eneral* ***R****esponsability* ***A****ssignment* ***S****oftware* ***P****atterns*

Les patterns "*GRASP*" vise l'ultime objectif suivant:

- Comment répartir au mieux les ***responsabilités*** (*services rendus aux travers d'un sous-ensemble cohérent de méthodes publiques*) au niveau d'un ensemble de classes plus ou moins inter-connectées ?

- Quelles sont les *affectations* qui garantissent le mieux la ***modularité*** et **l'*extensibilité*** de l'ensemble ?

Les 4 patterns "GRASP" fondamentaux

Expert: affecter la responsabilité à la classe qui détient l'information.

Faible couplage: la répartion des responsabilités doit conduire à un faible couplage (relative indépendance)

Forte cohésion : la répartion des responsabilités doit conduire à une forte cohésion (pas de dispersion , ...)

Création: La responsabilité de créer une instance incombe à la classe qui agrège, contient, enregistre, utilise étroitement ou dispose des données d'initialisation de la chose à créer.

5 patterns "GRASP" plus spécifiques

Contrôleur: classe supervisant des interactions élémentaires , stéréotype <<*control*>> , en liaison avec scénario de U.C.
Polymorphisme: pour petites variations au niveau des sous classes tout en gardant une homogénéïté et une bonne extensibilité.
Fabrication pure: affecter un ensemble de responsabilités fortement cohésif à une *classe artificielle* ou de commodité qui ne représente pas un concept du modèle du domaine .
Indirection: ajouter un *intermédiaire* entre 2 éléments pour éviter de les coupler de façon trop rigide.
Protection des variations: *anticiper* de futures *variations* et les placer derrières des *interfaces stables*.

2. Les 4 patterns GRASP fondamentaux

2.1. Expert

Expert (GRASP)

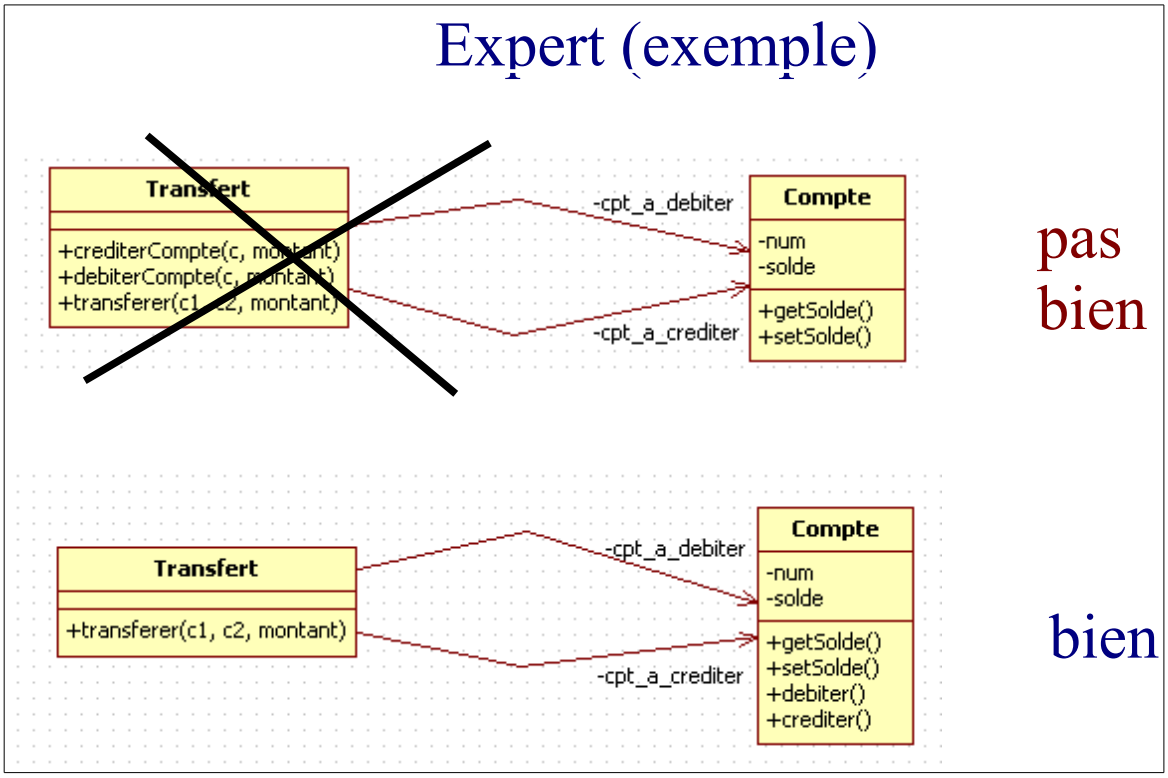
L'*expert*, c'est celui qui sait (qui détient quelques informations clefs) et qui peut donc prétendre pouvoir assumer une certaine responsabilité.

--> Placer de préférence les services/méthodes (traitements internes) au plus près des données utiles en passant par le moins d'intermédiaires possibles *(l'autonomie est à rechercher au niveau des objets)*.

--> Déléguer le plus possible
(sous services / sous responsabilités / subsidiarité , ...)

--> Ne prendre du recul vis à vis des informations que s'il faut agir à un niveau relativement global.

Expert (exemple)



2.2. Faible couplage

Faible couplage (GRASP)

Le couplage désigne la densité des liens/relations existants entre les différents objets d'un système.

--> Trop de couplage (beaucoup d'inter-dépendances,...) amène généralement à une grande complexité, une certaine fragilité de l'édifice et à l'impossibilité de réutiliser un seul élément sans avoir à comprendre et réutiliser aussi tout ce qu'il y a autour.

--> Inversement un couplage trop faible est quelquefois la marque d'une chose monolythique (pas assez décomposée) ou bien une entité assez isolée rendant peu de services utiles.

--> Le bon niveau de couplage est une affaire de compromis et de jugement (*idéalement faible pour minimiser les dépendances) (avec quelques liaisons tout de même pour ne pas conduire à trop de parties totalement isolées/déconnectées*).

Faible couplage (exemple)

faible couplage non respecté !!!

S'il faut choisir entre 2 cablages, choisir celui qui utilise le moins de fils

2.3. Forte cohésion

Forte cohésion (GRASP)

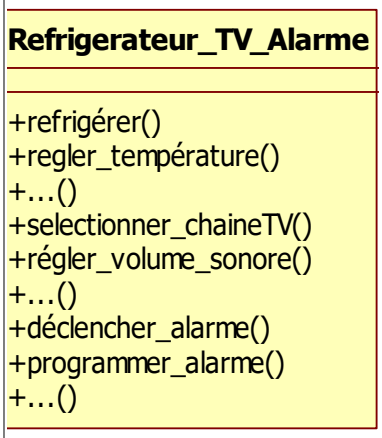
La forte cohésion d'une classe ou d'un système désigne la cohérence fonctionnelle de l'ensemble (la non dispersion des responsabilités)

--> Une faible cohésion est très souvent la marque d'une mauvaise conception (pas assez de réflexion , d'organisation) et menant à un édifice difficile à comprendre.

--> Une entité fortement cohésive doit normalement faire peu de choses mais le faire bien (à fond ou presque).

--> "Forte cohésion" va souvent dans le même sens que "spécialisation fonctionnelle" dans l'élaboration d'un édifice modulaire.

Forte cohésion (contre exemple)



tout et n'importe-quoi !

2.4. Création

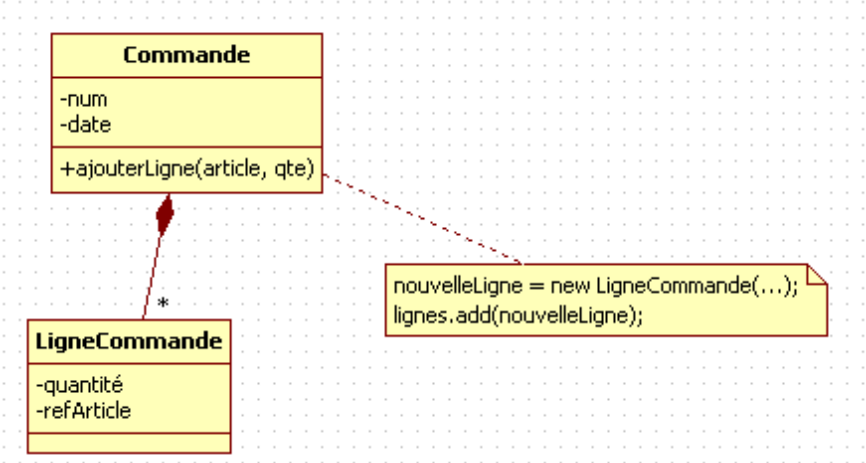
Créateur (GRASP)

Le design pattern *Créateur* indique que la *responsabilité de créer une nouvelle instance* est bien *souvent à affecter à l'élément qui contient, enregistre, initialise ou utilise le nouvel objet qui sera créé*.

--> Ceci est assez logique et intuitif car dans l'écriture "*refNewObj = new Cxx(a,b);*" les valeurs *a* et *b* servent à initialiser la nouvelle instance et sont connues par celui qui enregistre ou initialise (ou ...) et la référence retournée permet d'accéder à la nouvelle instance à manipuler ou contenir (ou ..)

--> NB: Appliquer le pattern GRASP "*Créateur*" en fin d'analyse ne dispense pas d'appliquer ultérieurement les patterns "*Factory*" ou "*IOC / injection de dépendances*" en phase de conception de façon à anticiper des variations technologiques ou des extensions.

Créateur (exemple)



c'est à la commande que revient la responsabilité de créer une nouvelle ligne de commande.

Exemple2: C'est normalement la boîte de dialogue qui doit créer ses propres sous éléments (boutons "ok" , "cancel").

3. Les 5 paterns GRASP spécifiques

3.1. Contrôleur

Contrôleur (GRASP)

Un élément "***contrôleur***" est censé superviser ou coordonner un ensemble cohérent d'opérations/interactions sur des objets quelquefois éparpillés.

--> On peut citer 3 types très classiques de contrôleurs:
- le *contrôleur d'IHM* dans le modèle *MVC* (gestion des événements , déclenchement des actions & affichages)
- le *contrôleur de façade* comme point d'entrée unique d'un module métier complet (rôle d'aiguillage)
- le *contrôleur de séquence applicative ou métier* (très souvent associé à la réalisation d'un *scénario* attaché à un *cas d'utilisation*): classe "*GestionnaireXxx*" , "*CoordonnateurXxx*" ou "*SessionXxx*" et ayant le stéréotype d'analyse ***<<control>>*** *(Jacobson)*.

3.2. Polymorphisme

Polymorphisme (GRASP)

Une même opération abstraite peut être implémentée plusieurs fois avec des variantes liées aux types exacts des objets.

==> un même fond (service rendu) mais plusieurs formes possibles (au cas par cas) devant quelquefois cohabitées.

==> automatisme des langages objets dans le choix de la variante (sans if/else ...à coder).

3.3. Fabrication pure

Fabrication pure (GRASP)

* Une ***fabrication pure*** est une ***réalisation/construction totalement artificielle*** qui ne correspond pas directement à un des éléments du domaine *(de la réalité)* mais qui est nécessaire pour obtenir une bonne conception (faible couplage, ...).
* Une fabrication pure est généralement une *entité assez abstraite* (ex: fabrique , D.A.O. , intermédiaireXY, ...) qui est obtenue par *décomposition comportementale ou représentationnelle/structurelle*. Autrement dit certaines responsabilités sont extraites d'un élément du domaine pour être déplacées dans une *entité annexe artificielle*.

3.4. Indirection

Indirection (GRASP)

En ajoutant (de façon avisée) un ***nouvel élément intermédiaire*** (et par conséquence une ***nouvelle indirection*** dans une séquence d'interactions), on répartit généralement mieux les responsabilités et l'on aboutit à un semble plus modulaire.
Exemples:
* "Source de données" pour indirectement établir une connection à une base de données.
* "Décorateur (gof)" pour indirectement ajouter quelques fonctionnalités à un élément de base.

3.5. Protection des variations

Protection des variations (GRASP)

Certaines évolutions/modifications du système sont quelquefois prévisibles et l'on peut anticiper leurs mises en oeuvre en plaçant l'instable derrière une interface stable.
==> La notion d'encapsulation va dans ce sens (public / privé).
==> Les interfaces abstraites constituent le moyen le plus classique et le plus efficace de se protéger contre les variations. Etant néanmoins vue comme un contrat idéalement inaltérable, l'interface doit absolument être bien pensée pour demeurer stable et utilisable (non jetable) *[sinon c'est quelquefois pire que mieux]*.

X - Technologies pour framework

1. Nouveau framework (généralités)

1.1. Intérêts récurrents liés à un nouveau framework

- Automatiser le répétitif (contrôles, liaisons , collaborations, ...)
- Gagner en productivité (si réutilisation fréquente du framework)
- Gagner en robustesse (si framework éprouvé bien au point)
- Structurant (beaucoup de composants partagent la même structure et la même logique) ---> maintenance à priori plus aisée.
- Intellectuellement enrichissant pour le concepteur qui met au point le framework.

1.2. Inconvénients fréquents d'un nouveau framework

- Yet Another Framework (risque de réinventer la roue si framework existant proche)
- Tout framework tend à imposer certaines structures (héritage ou ....) . Ceci peut éventuellement s'avérer bloquant (manque d'ouverture)
- Un framework mal documenté (qui n'explique pas exactement ce qu'il automatise) est un mauvais framework (incompréhensible , ...)
- Pour l'utilisateur d'un framework : soit intellectuellement enrichissant si l'on cherche à étudier le fonctionnement interne du framework , soit intellectuellement appauvrissant si on se contente de coder les sous composants et les paramétrages attendus par mimétisme (copier/coller d'autres exemples) ; manque de liberté ; initiative limitée .

2. Technologies pour Framework

2.1. Analyse d'un fichier de configuration xml

La plupart des frameworks puisent leurs configuations dans des fichiers xml (ex: springConf.xml , struts-config.xml, faces-config.xml , ...) .
Un nouveau framework peut procéder de la même façon et il aura donc besoin de lire et de convenablement interpréter l'arborescence des balises d'un fichier xml.

Technologies classiques pour le parsing XML (en java):

- SAX
- ***DOM*** (et variantes jdom , dom4j)
- ***JAXB2***
- ***stax*** (stream api for xml)

Configuration xml ===> configuration orienté objet en mémoire ===> application/exécution.

Alternatives à la configuration xml :

- **fichiers ".properties"** (plus simples à analyser mais pas arborescents)
- **annotations** (ex: *@Table(name="xxx")*) présentes dans certaines parties du code java des composants (à analyser par ***introspection*** lors de l'exécution ou bien via ***apt*** [annotation processing tool])

2.2. Introspection et méta classe java.lang.Class

De nombreux frameworks automatisent certains traitements en analysant la structure des classes java de l'application (mécanisme d'introspection).

Rappels sur l'introspection

Introspection

Le package **java.lang.reflect** permet d'effectuer une **introspection** des classes java. L'introspection consiste à ***demander aux classes de dresser la liste de leurs attributs et méthodes***.

Cette **faculté d'auto-analyse** qu'offre les mécanismes internes du langage Java est pour l'instant inexistante en C++ .
➔L'introspection est un **gros point fort de Java** .
➔Ceci ***permet d'automatiser entièrement des opérations de bas niveaux (sauvegarde / restauration de valeurs, ...).***

```
import java.lang.reflect.*;

Class c = Class.forName(nomClasse);
Field[] tabChamps = c.getDeclaredFields();
Method[] tabMethods = c.getDeclaredMethods();
 + boucle "for(i=0; i < tabXxxx.length; i++) ...
```

NB: depuis le jdk1.5 , il est également possible de récupérer dynamiquement à l'exécution du programme la liste de toutes les annotations (de rétention adéquate) (ex: @Id , ...).

Analyse d'une structure pour recopier des données

....

// La Classe **Field** (représentant un attribut d'une classe) comporte tous les éléments nécessaires
// pour récupérer la visibilité , le type , le nom et la valeur d'un attribut.

// On peut donc assez facilement balayer par boucle tous les attributs de façon à automatiser certains
// traitements (recopie de valeur , affectation de valeur , ...)

....

Pour simplifier la manipulation des propriétés d'une instance java par introspection, on pourra éventuellement s'appuyer sur les classes du package "***org.apache.commons.beanutils***"(de ***commons-beanutils.jar***) .

Analyse et/ou sélection d'une opération pour l'invoquer

```
Class implClass = Class.forName(className);
  // ou bien implClass = objXxx.getClass();
String nomMethode="validate"; // méthode à invoquer
Method[] tabMeth = implClass.getMethods();
for(i=0;i<tabMeth.length;i++)
        if(tabMeth[i].getName().equals(nomMethode))
                { index=i; break; }
if(index>0)
  tabMeth[index].invoke(objXx, yyyValue);  //approfondir si besoin java.lang.Method et invoke
                                           // via javadoc du jdk
...
```

Prise en compte d'une annotation à l'exécution (runtime):

ex: code de la nouvelle annotation @A_valider

```
....
@Documented
@Retention(RUNTIME)
public @interface A_valider {
      /** Message si invalide */
      String value();
     ...
     /** type à valider (défaut : STRING). */
      TypeData  data_type() default  TypeData.STRING;
      /** Enumération des différents niveaux de criticités. */
      public static enum  TypeData { INT, DOUBLE, STRING };
}
```

(Remarque: l'information data_type est simplement içi montrée pour la syntaxe (avec enum) . Cette information n'est pas très utile dans un cas réel car l'introspection classique est déjà capable de récupérer l'information de type . Dans un cas de validation réel , on pourra plutôt placer des informations de type "min" , "max" , ...).

Utilisation (déclarative):

```
import xxx.A_valider;
class Xxx {
@A_valider(value="doit être un entier" , data_type=INT)
int age;

@A_valider(value="doit être une chaine")
String nom;
...
}
```

Prise en compte (test effectué par le framework):

```
// Instanciation de l'objet:
Xxx objet = new Xxx();

// On récupère la classe de l'objet :
Class<Xxx> classInstance = objet.getClass();

// On regarde si la classe possède une annotation :
A_valider annotation = classInstance.getAnnotation(A_valider.class);

if (annotation!=null) {
      System.out.println ("MonAnnotation : " + annotation.value() );
      }
```

Instanciation dynamique (sans new explicite):

```
String className = .....; // nom complet avec package (ex: "java.util.Date")
Class implClass = Class.forName(className);
Object objRes = implClass.newInstance();
....
```

2.3. Autres technologies diverses pour les frameworks

Décorateur / Proxy / Intercepteur

Beaucoup de frameworks automatisent certains éléments en:

- introduisant/interposant un nouvel objet intermédiaire entre le client appelant et le service appelé.
- cet intermédiaire (appelé "intercepteur" , "décorateur" , "enveloppe" ou "proxy") reprend la même interface que le service d'origine (mêmes méthodes publiques exposées) et ces nouvelles versions des méthodes (ainsi directement appelées par le client) vont en général :
  - introduire (avant et/ou après) de nouvelles fonctionnalités (ex: logs , tests , automatisme xy, ...)
  - rappeler en interne les méthodes de mêmes noms sur le service d'origine
- placer/relier le nouveau composant intermédiaire de façon idéalement transparente entre le client et le service d'origine (via une fabrique à Proxy ou AOP ou ....).

Programmation par aspects

Certains éléments de la programmation par aspects peuvent être utiles à la mise en oeuvre d'un framework (ex: ajout de fonctionnalités --> décorateur/proxy/intercepteur, ...).

Concrètement , un aspect supplémentaire nécessite une technologie adéquate (ex: Spring AOP , aspectJ, ...) . En d'autres termes le nouveau framework doit assez souvent être construit à partir d'un premier framework (ex: Spring ou ...) pour bénéficier de AOP.

Initialisation coté java/web via Listener(s)

Dans beaucoup de frameworks "java/web" , on a assez souvent besoin de déclencher certaines initialisations dès le chargement de l'application dans le conteneur Web (Tomcat ou ....).

Un **Listener** (de l'api des ***Servlet***) est tout à fait approprié pour remplir cette tâche.

Rappels:

Certains ***événements liés à une application WEB*** (ou plus exactement à l'objet central "ServletContext") peuvent être gérés au sein d'objets appelés "**Listener**" .
Ceci permet essentiellement de ***déclencher automatiquement certains traitements*** aux moments des ***chargement/initialisation*** et ***arrêt/déchargement*** d'une ***application WEB***.

Interfaces événementielles (javax.servlet)	***Descriptions***
ServletContextListener	objet "application" (ServletContext) tout juste créé ou bien sur le point d'être supprimé.
ServletContextAttributeListener	Attribut ajouté,supprimé ou modifié sur l'objet application (ServletContext)

NB:

Une classe d'objet "***Listener***" doit ***implémenter l'interface événementielle adéquate*** et son code compilé doit être placé dans **WEB-INF/classes** ou bien dans un des **"...jar"** de **WEB_INF/lib**.

D'autre part, un "***Listener***" (gestionnaire d'événements) doit être déclaré au sein du fichier ***WEB-INF/web.xml*** pour qu'il soit pris en compte:

```
<web-app ...>
<display-name>MyListeningApplication</display-name>
<listener>
    <listener-class>mypackage.MyListenerClass</listenerclass>
</listener>
<listener>
      <listener-class>mypackage.MyOtherListenerClass</listener-class>
</listener>
<servlet>...</servlet>
...
</web-app>
```

Exemple:

```
package mypackage;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class MyListenerClass implements ServletContextListener {

        public void contextInitialized(ServletContextEvent e) {
     // initialisation au chargement/démarrage de l'application WEB
                ServletContext application = e.getServletContext();
```

```
            Integer objCompteur = new Integer(1);
            application.setAttribute("compteur",objCompteur);
    }

    public void contextDestroyed(ServletContextEvent e) {
  //  terminaison lors de l'arrêt de l'application WEB
            ServletContext application = e.getServletContext();
            Integer objCompteur = (Integer) application.getAttribute("compteur");
            System.out.println("compteur:" + objCompteur.intValue());
    }  }
```

Enrichissement de la partie html/javascript via des Filtres

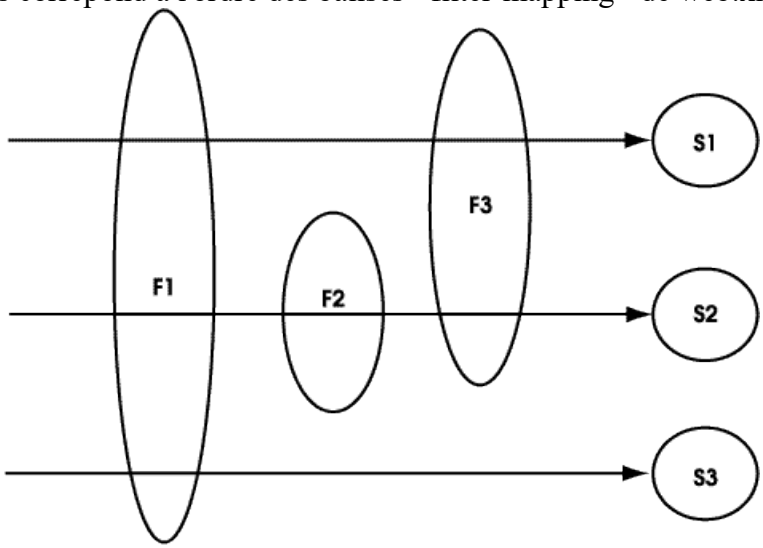
Notion de filtre:
Un filtre est un élément supplémentaire qui s'insère en tant qu'enveloppe au niveau de la chaîne d'exécution des servlets et qui peut modifier la requête et/ou la réponse Http.

Ceci peut être utile pour effectuer l'une des tâches suivantes:

- Effectuer des conversions (images, données, ...)
- Gérer le cryptage des données
- Effectuer des transformations XSLT.
- Effectuer des compression & décompression (gzip)
- Rajouter automatiquement des entêtes , des compteurs , des stats, ...
- Ajouter des éléments javascripts (menus , validations, ...)

Insertion d'un filtre (Filter Mapping de web.xml):
Un filtre n'est jamais directement mentionné dans une url , il est simplement associé à certaines url en tant qu'élément supplémentaire (s'activant avant en tant qu'enveloppe) :

Nb: l'ordre des filtres correpond à l'ordre des balises <filter-mapping> de web.xml.




```
<web-app>
<filter>
    <filter-name>XSLTFilter</filter-name>
    <filter-class>XSLTFilter</filter-class>
  </filter>
```

```
<filter>...</filter>
  <filter-mapping>
    <filter-name>XSLTFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <filter-mapping> ... </filter-mapping>
 <servlet> ... </servlet>  <servlet-mapping> .... </servlet-mapping>...<web-app>
```

Programmation d'une classe de Filtre:

```
public final class XXXFilter implements Filter {
   private FilterConfig filterConfig = null;

   public void init(FilterConfig filterConfig)
      throws ServletException { this.filterConfig = filterConfig; }

   public void destroy() { this.filterConfig = null; }

   public void doFilter(ServletRequest request,
      ServletResponse response, FilterChain chain)
      throws IOException, ServletException {
           //... code du filtre ....
        chain.doFilter(request,response); // déclenche l'élément
                         // suivant (autre filtre ou servlet , ...)
        }
}
```

De façon à pouvoir modifier la réponse générée par le code du servlet (élément suivant de la chaine) , il faut invoquer **chain.doFilter(request,wrapper);** où **wrapper** est une enveloppe autour de response dont la méthode **getWriter**() retourne un flux sur un paquet d'octets qui ne sera pas directement renvoyé mais qui sera retraité par la fin du filtre actuel:

```
public class CharRespWrapper extends   HttpServletResponseWrapper {
  private CharArrayWriter output;
  public String toString() {  return output.toString(); }
  public CharResponseWrapper(HttpServletResponse response)
          { super(response); output = new CharArrayWriter(); }
 public PrintWriter getWriter() {return new PrintWriter(output);}
}
```

```
...
PrintWriter out = response.getWriter();
CharRespWrapper wrapper = new
CharRespWrapper( (HttpServletResponse)response);
chain.doFilter(request, wrapper);
out.write("...." + wrapper.toString() + "...");
```

2.4. Autres techniques pour "framework"

TLS (Thread Local Storage) → accès global vis à vis du code (comme singleton) mais chaque thread à sa propre version .

XI - Eléments AOP (Prog. Orientée Aspects)

1. A.O.P. = complément nécessaire de la P.O.O.

Lacune de la P.O.O. utilisée seule : Tyrannie de la hiérarchie dominante
Apports de la programmation orientée aspect : Tisser les aspects , généricité , ...

2. Les concepts de AOP (vocabulaire)

Aspect:
Un **aspect** correspond à une **fonctionnalité transverse** qui **concerne une multitude d'objets** (ex: gestion des logs , gestion des transactions , ...).

JointPoint:
Un **point de jonction** (***JoinPoint***) correspond à un **point précis de l'exécution d'un programme** (*ex: appel d'une méthode , remontée d'une exception*) . C'est à ce genre d'endroit que pourront être insérés de nouvelles fonctionnalités (traitements transverses/génériques).

Advice:
Un "**advice**" correspond à une **action** (**ajout de traitements supplémentaires**) qui sera déclenchée au niveau de certain(s) point(s) de jonction.
Il en existe plusieurs catégories : "around" , "before" , "throws" , ...

PointCut:
Un "***PointCut***" correspond à un ensemble de points de jonction qui doivent être associés à un certain "advice" (ajout de traitements supplémentaires) . Autrement dit un "*pointcut*" correspond au paramétrage global du tissage des aspects: [*PointCut* = liste des points d'intersection entre le code principal et un certain aspect]

Target (advised) object:
Objet contenant un (ou plusieurs) point(s) de jonction.

Introduction:
Action particulière consistant à ajouter un nouveau membre (méthode ou champ) à une classe d'objet cible (advised object).

AOP_Proxy:
Nouvel objet créé par les mécanismes d'AOP et comportant des ajouts ("advices").

Weaving:
Assembler (tisser les aspects) pour former de nouveaux objets plus complets.
Ceci peut être effectuer lors de la compilation ou bien à l'exécution du programme (selon la technologie employée).

3. Les grands axes de la mise en oeuvre d' A.O.P.

Il existe actuellement tout un tas d'implémentation de AOP qui diffèrent essentiellement selon les grands axes suivants:

- le **langage de programmation** utilisé (Java , C++ , C# , ...).
- l'**encodage des points de jonctions** (fichiers xml , annotations , composants spécifiques , ...)
- la **manière utilisée et le moment pour tisser les aspects** (pré-compilation , dynamiquement à l'exécution , ...)

Chaque solution a ses avantages et ses inconvénients qui sont à évaluer au cas par cas (en fonction du contexte) avant de choisir une technologie précise.

3.1. avantages et inconvénients selon les choix technologiques

Technologies utilisées	*avantages (points forts)*	*inconvénients (points faibles)*
pré-compilation (pré-processeur)	solution efficace et robuste (bonnes performances, ...) . beaucoup de possibilités (ajout d'attributs , ...). méthode assez statique (vis à vis du code compilé)	nécessite un environnement de développement particulier (IDE avec pré-processeur) ==> petite dépendance vis à vis de la plate-forme de développement (avant qu'apparaisse UN standard).
dynamique – à l'exécution	solution plus simple et plus souple (éventuellement re-paramétrable sans recompilation). pas de dépendances vis à vis du compilateur (pas de pré-processeur)	performances et possibilités moins évoluées.
mécanisme double (pré-compilation + complément dynamique)	compromis intéressant	complexité. ensemble peu homogène (si détails non masqués)
encodage xml des "pointCut"	rien à ajouter dans le code de base. possibilités évoluées (ex: dans toutes les classes du package , ...).	fichier de paramétrage (.xml) basé sur une syntaxe spécifique à la technologie employée (en attendant une éventuelle standardisation).
"pointCut" sous forme d'annotations (java 5, ...)	très pratique pour fournir des paramétrages fins . les informations sont très proches du code concerné.	Si les annotations utilisées ne sont pas liées à un standard , on introduit alors une dépendance assez forte vis à vis de la technologie employée.
"pointCut" sous forme de composant (bean + ioc , ...)	très pratique pour injecter de façon modulaire des mécanismes (AOP) au sein d'un framework IOC (conteneur léger de type HiveMind	Le paramétrage de cette variante utilise généralement des fichiers xml ou des annotations ou un mixte des

Technologies utilisées	*avantages (points forts)*	*inconvénients (points faibles)*
	/ Spring)	deux.
encodage mixte (xml + annotations) des "pointCut"	souple – compromis. Elements génériques ==> xml Paramétrages fins ==> annotations	double dépendance (syntaxe des annotations + syntaxe du fichier xml)

==> Après une phase de recherche en grande partie déjà effectuée, A.O.P. est aujourd'hui entré dans une phase d'ingénierie (beaucoup de projets concurrents).

==> Une future phase de standardisation est très attendue .

==> Les valeurs ajoutées de AOP sont suffisamment importantes pour se lancer dès aujourd'hui dans une technologie pilote (quitte à restructurer le code lorsqu' apparaîtra une standardisation).

3.2. solutions basées sur des pré-compilations (pré-processeur)

AspectJ

==> Cette technologie "AOP" dédiée au langage Java est assez avancée et est adoptée par une grande communauté de développeur.

==> AspectJ est une technologie basée sur une pré-compilation .

==> Le projet "Ecipse-AspectJ" permet d'intégrer la technologie "AspectJ" dans l'I.D.E. "Eclipse".

3.3. solutions basées sur des mécanismes dynamiques lors de l'exécution

Spring AOP , ...

Les méanismes de Spring AOP sont entièrement dynamiques. ils sont déclenchés lors de l'exécution du programme (et n'influent en rien la compilation).

4. Spring AOP (essentiel)

4.1. Technologies AOP et "Spring AOP"

- AOP (Aspect Oriented Programming) est un complément à la programmation orientée objet.

- *AOP* consiste à programmer une bonne fois pour toute certains aspects techniques (logs , sécurité , transaction, ...) au sein de classes spéciales.
- Une configuration (xml ou ...) permettra ensuite à un framework AOP (ex: AspectJ ou Spring-AOP) d'appliquer (par ajout automatique de code) ces aspects à certaines méthodes de certaines classes "fonctionnelles" du code de l'application.

- *Vocabulaire AOP*:
  PointCut : endroit du code (fonctionnel) où seront ajoutés des aspects
  Advice : ajout de code/aspect (avant, après ou bien autour de l'exécution d'une méthode)
- On parle de tissage ("weaver") du code :
  Le code complet est obtenu en tissant les fils/aspects techniques avec les fils/méthodes fonctionnel(le)s .

Les mécanismes de Spring AOP (en version >= 2.x) sont toujours dynamiques (déclenchés lors de l'exécution du programme) . Spring AOP 2 utilise néanmoins des syntaxes de paramétrage (annotations) volontairement proches du standard de fait java "AspectJ-weaver" .

4.2. Mise en oeuvre rapide de Spring aop via des annotations

```
package util;
//Nécessite quelquefois aspectjrt.jar , aspectjweaver.jar
//(de spring.../lib/aspectj)
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
```

```
@Aspect
public class MyLogAspect {

     @Around("execution(* xxx.services.*.*(..))")
     public Object doXxxLog(ProceedingJoinPoint pjp)
     throws Throwable {
          System.out.println("<< trace == debut == "
                    + pjp.getSignature().toLongString() + " <<");
          long td=System.nanoTime();
          Object objRes =  pjp.proceed();
          long tf=System.nanoTime();
          System.out.println(">> trace == fin == "
                    + pjp.getSignature().toShortString() +
                       " [" + (tf-td)/1000000.0 + " ms] >>");
          return objRes;
     }
}
```

avec `@Around("execution(typeRetour package.Classes.methode(..))")`

et dans ***dans myspringConf.xml***

```
....
 <aop:aspectj-autoproxy/>
     <bean id="myLogAspect"
          class="util.MyLogAspect">
     </bean>
....
```

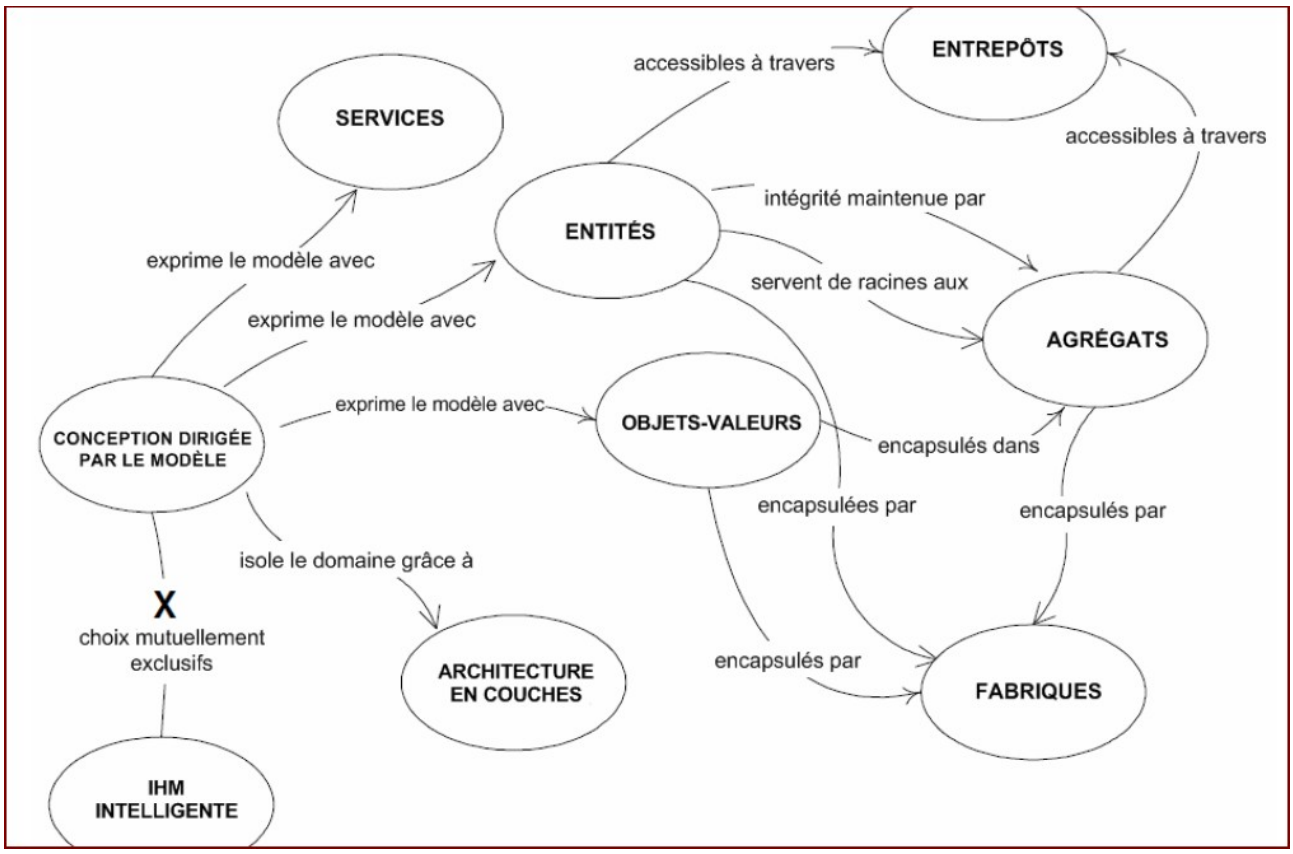
Configuration aop en xml (sans annotations dans la classe java de l'aspect)

```
...
<bean id="myLogAspectBean"  class="tp.util.MyLogAspect"></bean>
  <aop:config>
     <aop:pointcut id="execution_methodes_package_livre"
     expression="execution(* tp.bibliotheque.livres.*.*.*(..))" />

     <aop:pointcut id="execution_methodes_package_ab_emp"
     expression="execution(* tp.bibliotheque.ab_emp.*.*.*(..))" />

     <aop:aspect id="myLogAspect" ref="myLogAspectBean" >
         <aop:around  method="doXxxLog"
               pointcut-ref="execution_methodes_package_livre" />
         <aop:around  method="doXxxLog"
              pointcut-ref="execution_methodes_package_ab_emp" />
     </aop:aspect>

 </aop:config>
```

XII - Domain D. Design ,métriques, aspects divers

1. Domain Driven Design

Référence: "***Domain Driven Design Quickly***" de ***Eric EVANS*** *(le fondateur du courant de pensée)* .



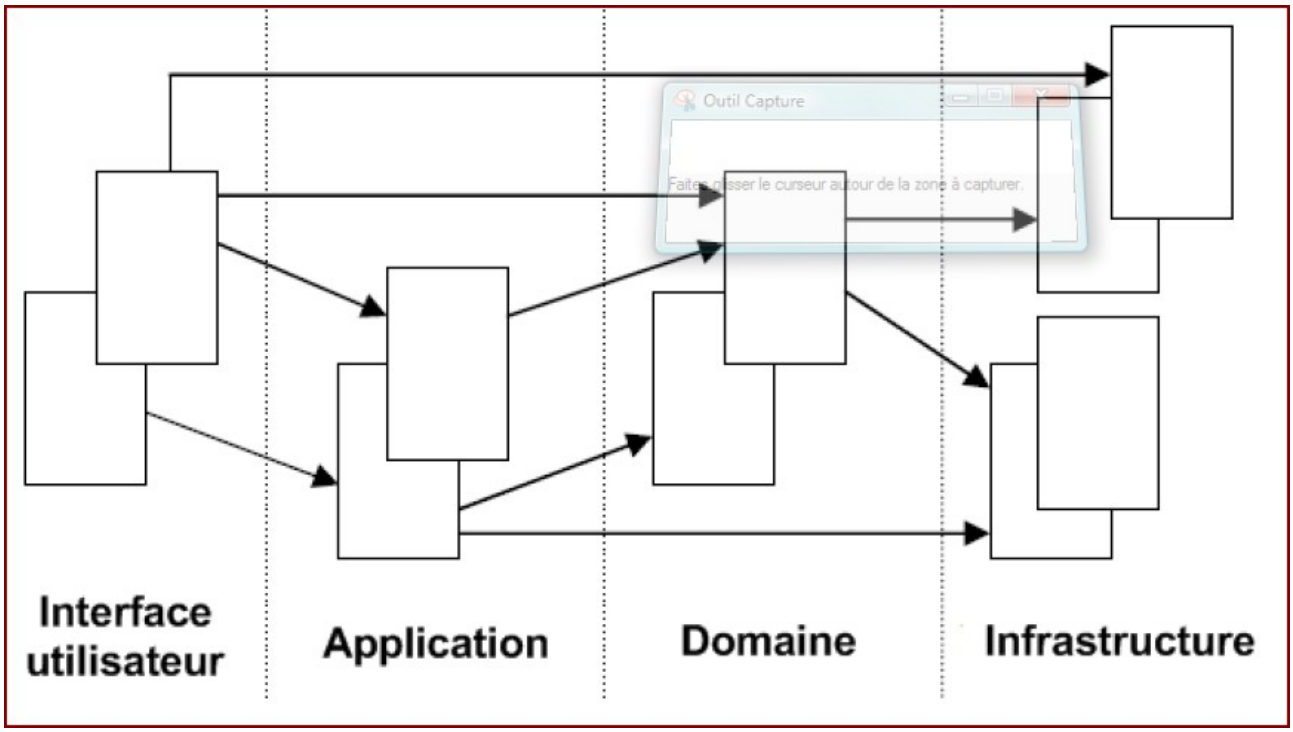
1.1. Principales idées (structure modèle & méthodologie)

DDD= plutôt adapté pour moyenne/grosse application (potentiellement complexe).

- **Centrer la modélisation et le développement du logiciel autour du modèle du domaine métier (proche du modèle d'analyse).**
- **Pas de séparation forte entre modèle "métier/analyse" et modèle "technique/conception" mais plutôt faire l'effort de construire un modèle convenant à tout le monde .**
- **Faire communiquer (experts métiers + analystes) et (concepteurs/développeurs).**
- **Utiliser un langage commun (dit "omniprésent" / "ubiquitous" en anglais) plutôt que d'utiliser des jargons différents.**
- Utiliser des diagrammes "UML" comme "coeur du modèle" (+textes/commentaires en +).
- **Pas que des entités mais aussi des "VO" avec idéalement des "*VO immuables*" pour partages faciles .**
- Privilégier des petits "VO" simples et d'éventuelles compositions ....
- Services métiers (réellement nécessaires dès l'analyse si opération(s) pas directement lié à 1

(seule) entité ou 1 (seul) VO bien précis mais plutôt liée(s) à plusieurs objet(s) et si "sans état")

- **Agrégats = grappe d'objets avec racine = entité fondamentale** . Tout accès ou toute modification d'un objet lié secondaire passe par une opération déclenchée sur la racine fondamentale .
- C'est la racine qui (au sein de ses méthodes/opérations) assure les "invariants/règles d'intégrités" .
- Opérations en cascade sur l'agrégat .
- Idée à débattre/expérimenter : mixer "entités" , "VO" , "agrégat" , ... au sein d'un même modèle (avec fabrique ?) ????
- **Entrepôt** = objet spécialisé dans opérations de persistance "*CRUD*" (style ***DAO***) : avec peut être en plus la notion d'entité élargie (avec agrégats et "vo") .
- Refactoring
- **Couches logicielles** (IHM+application_coordination , domaine , infrastructure) .

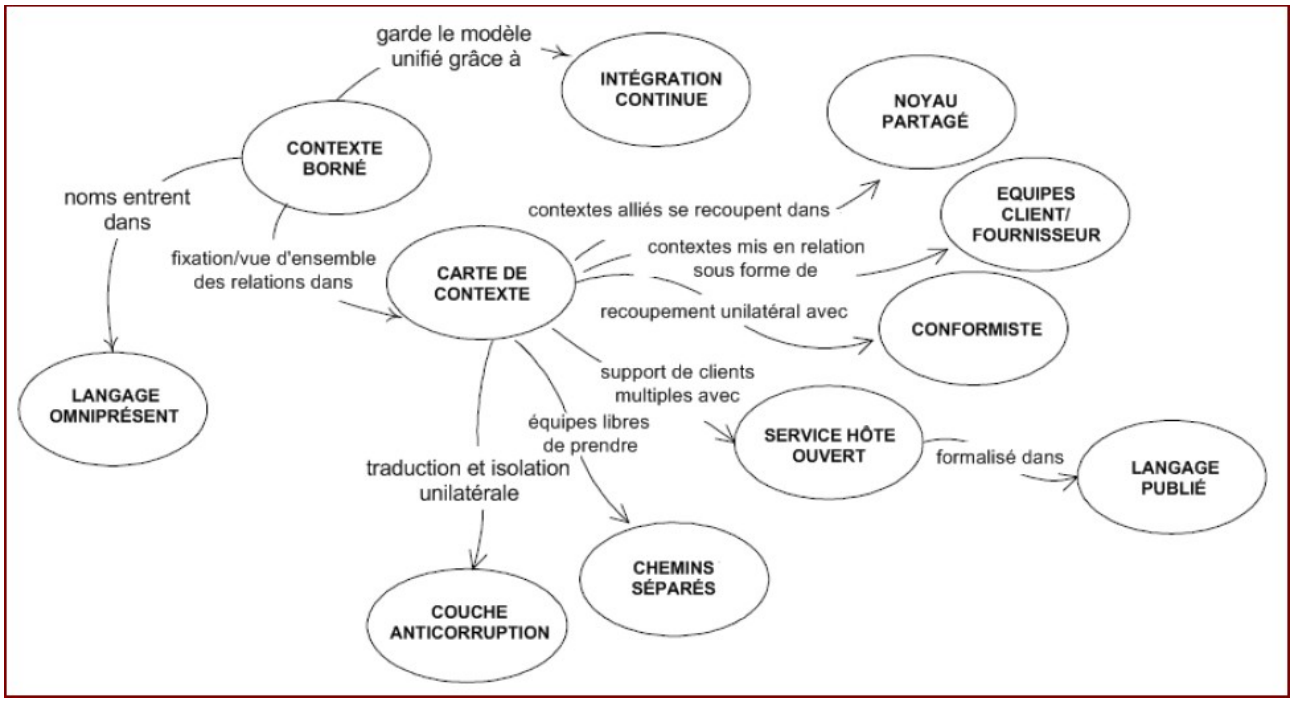


2. Organisation proposée par DDD

Idée principale: **Préserver l'intégrité (stabilité) du modèle (dans le cadre d'un développement collaboratif / à plusieurs).**

- si gros/énorme modèle ---> le diviser en plusieurs sous modèles .
- **Périmètre applicatif (contexte borné)** englobant plusieurs modules/packages .
- Ce périmètre sera peut être amené à évoluer un petit peu .
- **Intégration continue** de "l'ensemble des modules d'un contexte borné" .
- Au sein d'un (sous-)modèle" fin, **"la conception peut être affinée et distillée afin d'obtenir le maximum de pureté"** .

- **Carte de contextes** : diagramme montrant plusieurs contextes/modèles et les relations entre différents éléments de différents modèles (ex: traductions/mappings).
- ---> **éventuel noyau partagé** entre 2 modèles/contextes
- ---> **éventuelle relation "client/fournisseur"** entre 2 systèmes *(interfaces , timing / livraison ? , ...)*
- **approche "conformiste"** : le client s'adapte à la structure du fournisseur (qui de son coté doit garantir une certaine stabilité) .
- **Approche "adaptée"** (appelée "***anti corruption***" dans DDD) : traduction de modèle client/interne vers modèle externe avec idéalement façade vers itf/adaptateur , ...
- **Chemins séparés** : décomposer une grosse applications en plusieurs petites applications (2 contextes bornés) et IHM fine (du genre portail) donnant l'illusion d'un tout homogène .
- **Service hôte ouvert** : développer une ***couche d'accès distant de référence*** (qui sera utilisée par les futurs clients (directement ou via transformations) . Mais ne pas développer n liens sur mesure avec n clients différents .
- **Distillation** : ***faire ressortir l'essentiel d'un domaine (la substance essentiel) et transformer les sous-produits auxiliaires en "sous-domaines" .***
- En déplaçant l'*auxiliaire* , l'essentiel est mieux mis en valeur dans la modélisation.
- Bien identifier le "**cœur de domaine**" !! (par affinage / refactoring successifs)
- éventuels *sous domaines génériques* (compta, ...) sous forme d'applications existantes à intégrer .



3. Métriques de packages

Quelques critères de qualité:

Abstractness (A) ou degré d'abstraction.	Pourcentage de classes concrètes par rapport aux classes abstraites. Si proche de 0 : package concret . Si proche de 1 : package abstrait.	Le degré d'abstraction d'un package doit tendre vers l'une ou l'autre des deux borne : 0 ou 1. Une valeur proche de 0.5 montrerait une mauvaise écriture du code.
Afferent coupling (Ca) où couplage par dépendance descendante.	Le nombre de packages tiers qui utilisent les classes du package analysé/courant.	Si ce nombre est très grand, il est peut être nécessaire de fragmenter ce package "trop central" .
Efferent coupling (Ce) où couplage par dépendance ascendante.	Le nombre de packages utilisés par les classes du package analysé/courant	C'est un indicateur d'indépendance du code. Plus ce nombre est faible, mieux c'est.
Instability (I) où degré de stabilité	Indicateur de résilience du package : propriété de stabilité par rapport à la mise à jour d'autres packages. (cet indicateur est lié à Ca et Ce) .	Proche de 0 : package stable (cas idéal d'un pacakge abstrait), si proche de 1 : package instable (normal pour un package conret d'implémentations)
Distance from the Main Sequence (D)	Distance normale (perpendiculaire) à la droite A + I = 1. Proche de 0 : le package coïncide avec la « Main sequence », proche de 1 : très éloigné de la « Main séquence ».	Dans le cas idéal (D =0), un package est soit complètement abtrait et stable (A=1, I = 0) ou complètement concret et instable (A=0, I=0). Si D est proche de 1, c'est mauvais et est le signe d'un package mal géré.
Dépendance cyclique (Package Dependency Cycles)	Ce critère indique s'il existe des dépendances cycliques	Pas bien si il y en a!
...		

Plugins pour Eclipse:

- **JDepends**
- **Metrics**
- **...**

Liste des métriques proposées par "Metrics":

- **Lines of Code (LOC)**: Le nombre total de lignes de code. Les lignes blanches et les commentaires ne sont pas comptabilisés
- **Number of Static Methods (NSM)**: Le nombre de méthodes statiques dans l'élément sélectionné.
- **Afferent Coupling (CA)**: Le nombre de classes hors d'une package qui dépendent d'une classe dans le package
- **Normalized Distance (RMD)**: RMA + RMI - 1: Ce nombre devrait être petit, proche de zéro pour indiquer une bonne conception des parquets.

- **Number of Classes (NOC)**: Le nombre de classes dans l'élément sélectionné.
- **Specialization Index (SIX)**: NORM * DIT / NOM: Moyenne de l'index de spécialisation.
- **Instability (RMI)**: CE / (CA + CE) : Ce nombre vous donnera l'instabilité de votre projet. C'est-à-dire les dépendances entre les paquets.
- **Number of Attributes (NOF)**: Le nombre de variables dans l'élément sélectionné.
- **Number of Packages (NOP)**: Le nombre de packages dans l'élément sélectionné.
- **Method Lines of Code (MLOC)**: Le nombre total de lignes de codes dans les méthodes. Les lignes blanches et les commentaires ne sont pas comptabilisés
- **Weighted Methods per Class (WMC)**: La somme de la complexité cyclomatique de McCabe pour toutes les méthodes de la classe.
- **Number of Overridden Methods (NORM)**: Le nombre de méthodes redéfinies dans l'élément sélectionné.
- **Number of Static Attributes (NSF)**: Le nombre de variables statiques dans l'élément sélectionné.
- **Nested Block Depth (NBD)**: La profondeur du code
- **Number of Methods (NOM)**: Le nombre de méthodes dans l'élément sélectionné.
- **Lack of Cohesion of Methods (LCOM)**: Une mesure de la cohésion d'une classe. Plus le nombre est petit est plus la classe est cohérente, un nombre proche de un indique que la classe pourrait être découpée en sous-classe. Néanmoins, dans le cas de Javabean, cette métrique n'est pas très correcte, car les getteurs et les settteurs sont utilisés comme seules méthodes d'accès aux attributs. Le résultat est calculé avec la méthode d' Henderson-Sellers : on prend m(A), le nombre de méthodes accédant à un attribut A, on calcule la moyenne de m(A) pour tous les attributs, on soustrait le nombre de méthodes m et on divise par (1-m).
- **McCabe Cyclomatic Complexity (VG)**: La complexité cyclomatique d'une méthode. C'est-à-dire le nombre de chemins possibles à l'intérieur d'une méthode, le nombre de chemin est incrémenté par chaque boucle, condition, opérateur ternaire, … Il ne faut pas que ce nombre soit trop grand pour ne pas compliquer les tests et la compréhensibilité de la méthode.
- **Number of Parameters (PAR)**: Le nombre de paramètres dans l'élément sélectionné.
- **Abstractness (RMA)**: Le nombre de classes abstraites et d'interfaces divisés par le nombre total de classes dans un package. Cela vous donne donc le pourcentage de classes abstraites par package
- **Number of Interfaces (NOI)**: Le nombre d'interfaces dans l'élément sélectionné.
- **Efferent Coupling (CE)**: Le nombre de classes dans un packages qui dépendent d'une classe d'un autre package.
- **Number of Children (NSC)**: Le nombre total de sous-classes directes d'une classe
- **Depth of Inheritance Tree (DIT)**: Distance jusqu'à la classe Object dans la hiérarchie d'héritage.

4. Stéréotypes --> annotations / framework

Un modèle fonctionnel est souvent assez clair et compréhensible car il n'est pas surchargé par tout un tas d'éléments techniques (fabriques , DAO , ...) .

Pour effectuer une bonne conception qui ne dénature pas le modèle fonctionnel on pourra adopter une démarche proche de la suivante:

- **Modéliser de façon générique des architectures techniques types** (frameworks et sous frameworks existants, design patterns récurrents , ...)
- **Se créer une collection de stéréotypes UML** permettant de **spécifier des fonctionnalités techniques attendues** (ex: tx: transactionnel , stateless/stateful , sessionScoped , ...) .
  NB: Ces stéréotypes doivent idéalement ne pas être liés à une technologie précise pour permettre une éventuelle re-génération de code MDA avec une future technologie .
  [ex: ejb , servlet , ... ne sont pas des stéréotypes conseillés] .
- **Programmer s'il le faut de nouveaux petits frameworks** (construits au dessus de frameworks existants et "valeurs sures") pour répondre à des besoins spécifiques ou tout simplement pour automatiser le "répétitif" et ainsi gagner en productivité .
- Utiliser si possible une **technologie MDA** (Model Driven Architecture) de type "accéléo M2T" **pour générer du code spécifique** (avec utilisation de frameworks et paramétrages par annotations ou bien en xml) **en analysant la structure du modèle UML et en tenant compte des stéréotypes** .

Avantages:

- Le modèle fonctionnel d'analyse n'est que très peu surchargé par la conception fondamentale (type de données , stéréotypes , quelques façades structurantes , ...) .
  On peut donc utiliser et faire évoluer un même et unique modèle pour l'analyse fonctionnelle et la conception principale. [ceci est un gage de stabilité et de bonne communication].
- Modéliser des architectures types et en tirer la quintessence dans un nouveau framework constitue un bon exercice technique (souvent rentable sur le long terme si le projet "framework" n'est pas figé mais si il évolue et s'améliore petit à petit) .
- La génération de code (en partie automatique) via MDA n'est à voir que comme un accélérateur de développement . Ce n'est pas essentiel et c'est généralement rentable sur un gros projet (ou sur plusieurs petits projets de même type) .
  ---> *A ne pas faire (déconseillé)* : générer plein de lignes répétitives (sans framework / sans automatisme pré-programmé) car cela pénaliserait grandement l'évolution et la maintenance.
  ---> *A faire (conseillé)*: **générer une petite quantité de lignes de code qui utilisent des annotations interprétées par des frameworks** . Gros avantage: si future évolution technologique: on fait évoluer le framework , on le teste bien , et hop l'application en bénéficie sans chambouler sa structure.

Stéréotype UML (besoin en fonctionnalité technique) ---> **annotations** (java ou ...) **interprétée** (au runtime) par un **framework** .

XIII - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie

Titre	*Auteur / [editeur]*	*Commentaire(s)*
DESIGN PATTERNS Catalogue de modèles de conception réutilisables	*Auteurs:* ***Erich Gamma, Richard Helm,Ralph Johnson, John Vlissides*** traduction francaise de Jean-Marie Lasvergères. **Vuibert / Addison Wesley**	Livre de référence. Très bonne introduction. Diagramme OMT (avant UML) Exemples de code en C++
La tête la première dans les design patterns		apprendre en s'amusant (humour)

2. Liens WEB

URL	*Contenu / Commentaire(s)*
www.yahoo.fr	*rechercher "GRASP" "GOF" ou "design principles"*
www.google.fr	*idem*

3. Préliminaire

Les TP proposés ci-après ne sont pas du tout obligatoires . Il ne s'agit que d'une proposition de TP.
==> ne pas hésiter à effectuer d'autres TP .
==> ne pas hésiter à réaliser des variantes (en adaptant les énoncés).
==> les "design patterns" devraient idéalement l'occasion de s'exprimer (liberté dans l'implémentation , créativité , ...).

4. TP: Stratégies + Fabrique + Singleton

- Dans un projet java (par exemple sous eclipse) , créer les packages suivants pour structurer un peu le code: tp.dao , tp.entity, tp.test , ...
- Développer une première stratégie (***stratégie de type Data.Access.Object***) sur des entités de type Produit (avec reference, label et prix):
  interface abstraite ***ProduitDao*** et 2 implémentations:
  - ***ProduitDaoSimu*** (simulation avec HashMap interne)
  - ***ProduitDaoJdbc*** (accès jdbc vers une base de données mysql)
- Développer une Fabrique "***ProduitDaoFactory***" avec:
  - une méthode ***createProduitDao***() s'appuyant sur un fichier "***myDao.properties***"
    [#produitDao=tp.dao.ProduitDaoJdbc
    produitDao=tp.dao.ProduitDaoSimu] pour décider quelle version instancier.
  - le code classique du design pattern "***singleton***"
- Tester toute cette première partie (tp.test.MyTestApp avec main(...) et/ou testDao JUnit).
- Développer une seconde stratégie (***statégie de connexion à la base de donnés***):
  - **interface** "***ds.MyDataSource***" ou "***javax.sql.DataSource***" avec méthode **getConnection**() retournant une connection jdbc
  - **implémentation** "***ds.MyParamDataSource***" s'appuyant sur un fichier ***myDB.properties***
    [*jdbcDriver*=com.mysql.jdbc.Driver
    url=jdbc:mysql://localhost/prod_db
    userName=root
    password=root] et basée sur *Class.forName(...)* et *DriverManager.getConnection(...)*
  - **implémention** "***ds.MyJndiDataSource***" s'appuyant sur un ***lookup jndi*** permettant théoriquement d'accéder à un pool de connexion mis en oeuvre au niveau d'un serveur d'application (Tomcat , JBoss , WebSphere ou ...) .
    ==> il est tout à fait normal que cette implémentation parte en exception lorsque l'on effectue des tests sans serveur d'application (directement sous eclipse).
  - fabrique "***MyDataSourceFactory***" (s'appuyant sur *myDataSource.properties*) plus utilisation depuis ***ProduitDaoJdbc*** plus ***tests.***

5. TP: Amélioration via le Design pattern IOC

Dupliquer "***ProduitDaoJdbc***" en "***ProduitDaoJdbcIoc***" et adapter le code interne de la copie "***ProduitDaoJdbcIoc***" en introduisant une injection de la dépendance "*dataSource*" ==> **setDataSource**(...) .

Pour éviter d'avoir à programmer de multiple petites fabriques telles que "***ProduitDaoFactory***" "***MyDataSourceFactory" ,*** développer une fabrique globale "***tp.ioc.FabriqueEtendueGlobale***" avec une méthode (paramétrée ou non) permettant de :

- créer les différents éléments de la chaîne (Dao + DataSource)
- mettre ceux-ci en relation (gérer les injections de dépendances)
- retourne le premier de la chaîne construite

NB: cette fabrique globale étendue pourra s'appuyer sur les fichiers "myDao.properties" et "myDataSource.properties" , sachant qu'une version idéale (mais un peu trop longue à développer pour un simple TP) devrait s'appuyer sur un fichier de configuration xml (avec encodage des injections de dépendances).

Tester le tout via une nouvelle version de la classe de test.

Seconde partie (facultative) du TP:

- intégrer "**spring.jar**" dans le projet
- écrire un fichier ***mySpringConf.xml*** reprenant (en version xml) la configuration jusqu'içi dispersée dans "*myDao.properties*" , "*myDataSource.properties*" et le code interne de la fabrique "*tp.ioc.FabriqueEtendueGlobale**"***
- écrire une nouvelle version de la classe de test s'appyant sur la fabrique étendue globale prédéfinie de Spring "***XmlBeanFactory***" ou sur la variante "***ApplicationContext***" .

6. TP: Couches logicielles, services et vues "métiers"

Modéliser au sein d'un diagramme de classes UML les différentes couches logicielles classiques (*Présentation , Services métiers , accès aux données , ...*) [Remarque: on pourra utiliser des packages avec le stéréotype <<*layer*>> et des dépendances entre les packages]

Modéliser en UML de façon générique la structure de la couche "services métiers" en explicitant clairement les types d'objets "view" retournés et les types d'entités utilisées en interne .
Expliciter aussi les différents niveaux d'injections (Services métiers , Dao , DataSources) en utilisant les notations UML adéquates (interface , réalisation/implémentation , dépendance)

[On pourra facultativement tenir compte des notions de "Value Object" alias "Data Transfert Object" alias "View" et du principe d'inversion des dépendances appliqué aux DAOs].

Reprendre le Tp précédant et introduire un service métier "***GestionProduits***" comportant une méthode de type ***getProduitByRef***() déléguant le gros des traitements à ***ProduitDao*** (vue comme une dépedance injectée) et effectuant une conversion entre les types "*tp.entity.Produit*" et "*tp.view.ProduitView*" .
NB: cette conversion pourra éventuellement s'appuyer sur ***BeanUtil.copyProperties(...,...)*** de *common-beans.jar* ou de *spring.jar* .

7. TP: Design pattern "Façade"

- Développer plusieurs services métiers très simples:
 - *GestionConversion* avec *euroToFranc*(...) et *francToEuro*(...)
 - *GestionTva* avec *getTva*(...taux , ... montantHt)
- Développer ensuite une **façade** redirigeant vers ces différens services
  [Remarque: cette façade fera l'objet d'un singleton éventuellement basé sur une configuration Spring]
- + classe de test

8. TP (facultatif): Design patterns pour accès distants

- Développer une version distante des services métiers précédants (avec une technologie quelconque : services WEB , RMI , EJB , ...)
- coté serveur sur "*GestionTva*" , ajouter une méthode *calculTva*() retournant d'un coup un objet sérialisable "*TvaVO*" alias "*TvaDTO*" alias "*TvaView*" .
- Appliquer coté client les designs patterns suivants:
  - ***Proxy*** *(si nécessaire selon technologie utilisée)*
  - ***ServiceLocator***
  - ***BusinessDelegate***
- Développer éventuellement une *façade locale* coté client (si nécessaire)

9. TP: D.Pattern "décorateur" appliqué aux "caddy"

1. Modéliser au sein d'un diagramme de classe ***UML*** l'application du design pattern ***décorateur*** sur un ***panier électronique***.
2. Implémenter ensuite ceci en ***java*** .

NB: on pourra par exemple se baser sur les idées suivantes:

- caddy de base (***BasicCaddy***) avec simple liste de produits sélectionnés et méthodes *addInCaddy*() , *listerContenuCaddy*() .
- **AbstractCaddy** , **CaddyDeco** (niveau intermédiaire – classe abstraite)
- **StatCaddyDeco** avec redéfinition de *listerContenuCaddy() --> décoration = calculer et afficher le nombre d'éléments du caddy.*
- **LogCaddyDeco** avec redéfinition de *addInCaddy() --> décoration = ligne de log ou de trace à chaque appel*
- Classe de test avec différents décorateurs imbriqués les uns dans les autres .
- Sur un vrai projet (plus qu'un simple TP) , le D.P. "décorateur" peut éventuellement servir de base à une extension "CRM" .

10. TP(facultatif): Design pattern "composite" appliqué aux catégories et sous-sous catégories

- Modéliser au sein d'un diagramme de classe ***UML*** l'application du design pattern ***composite*** sur des entités de type "catégories" , "sous catégories" et "produits".

  Ex: "Livres" , "CD" ,"BD" , "BD enfants" , "Romans" – "roman1", "BD1" , ...

- Implémenter ensuite ceci en ***java*** .

Partie facultative du TP facultatif (pour le sport):

- élaborer un mapping objet/relationnel compatible avec le design pattern composite étudié ci-dessus (sachant qu'il y a un héritage à gérer).
- structurer le code via un "DAO" lié à la structure composite et via un service métier "GestionProduits" masquant cette structure composite mais offrant des méthodes du type "getProduitsByCategorie()" et "getSousCategories" .

11. TP: Design pattern "Observateur" appliqué au sujet "planning" et observateurs "vues sur planning"

Soit un sujet "planning_fournisseur" simple basé sur les éléments suivants:
- liste des semaines réservées (pour prépa, vacances , ...) [num_semaine , raison]
- liste options_clients [client_name , num_semaine]
  ===> sur les 52 semaines d'une année , toutes les semaines nons répertoriées dans l'une des 2 listes précédantes sera considérée comme "libre/disponible" pour d'éventuels clients intéressés.

==> 2 types d'observateurs:
- Vue de planning depuis client existant/effectif ==> on voit les semaines dispos (sans options placées par d'autres clients) , les semaines réservées (mais sans raison) et non disponibles (options placées par d'autres clients) , les semaines en option pour le client considéré.
- Vue de planning depuis nouveau client potentiel ==> on voit les semaines dispos (non réservées et sans option placée par un quelconque client existant)

==> Dès qu'un client place une nouvelle (ou supprime une ancienne) option , le planning fournisseur est évidemment mis à jour et toutes les vues "clientes" doivent être raffraîchies .

==> si une techno web (ex: servlet/jsp) est choisie pour mettre en oeuvre ce TP, on pourra éventuellement utiliser un raffraîchissement html automatique (ex: toutes les 5s) .
Lorsque le servlet sera ainsi régulièrement ré-appelé en mode "get", il pourra se contenter de:
- récupérer en scope = session le nom du client (si celui ic est connu / pas nouveau).
- récupérer dans une "map" une "vue cliente" prête à être rendue en html

Suite à une modification récupérée par le servlet en mode "post" , on déclenche toutes les mises à jour qui sont liées au pattern "observateur" ==> mise à jour sur sujet "planning_fournisseur" déclenchant indirectement des mise à jours des différentes "vues clientes" (observateurs enregistrés)

12. TP : Modélisation du modèle applicatif (IHM)

- Modéliser de façon générique au sein d'un diagramme de classe ***UML*** les éléments récurrents qui interviennent classiquement au niveau du modèle applicatif d'une IHM.
  Bien montrer:
  - le lien avec les services ou la façade
  - le lien avec les DTO/VO/View retournés par les services métiers
  - les détails jugés importants: <<in>> , <<out>> , <<stateless>> , <<stateful>> , ...
  - l'utilisation d'un éventuel référentiel de scope="application" avec éventuels caches, ...
- Entrevoir une éventuelle génération de code partielle (page jsp , config JSP ou ..., ...)

13. TP: découpages en différents packages

- Reprendre le diagramme UML modélisant les différentes couches logicielles (techniques) et peaufiner si besoin certaines dépendances en tenant compte du "D.I.P."
- Etudier sur une toute petite étude de cas UML les découpages fonctionnels d'une application en différents packages.
  Exemple: Commande/lignes_de_commande -----> Produits/....
  avec packages initiaux "commandes" et "produits"
  --> quelles sont les dépendances , qels sont les éléments les plus stables ?
- Tenir compte des "design principles" (OCP , DIP , CCP, ...)
  --> introduire de nouveaux packages intermédiaires
  --> réadapter les dépendances