Cour Design Pattern

Principe d'ingénierie:

→ Rigueur : s'assurer de faire ce qui est demander.
→ Abstraction : raisonner sur les concepts généraux et implémenter les cas particulier.
→ décomposition en sous-problème.
→ Modularité : partition en module intergissent.
→ construction incrémentale.

Design Pattern:

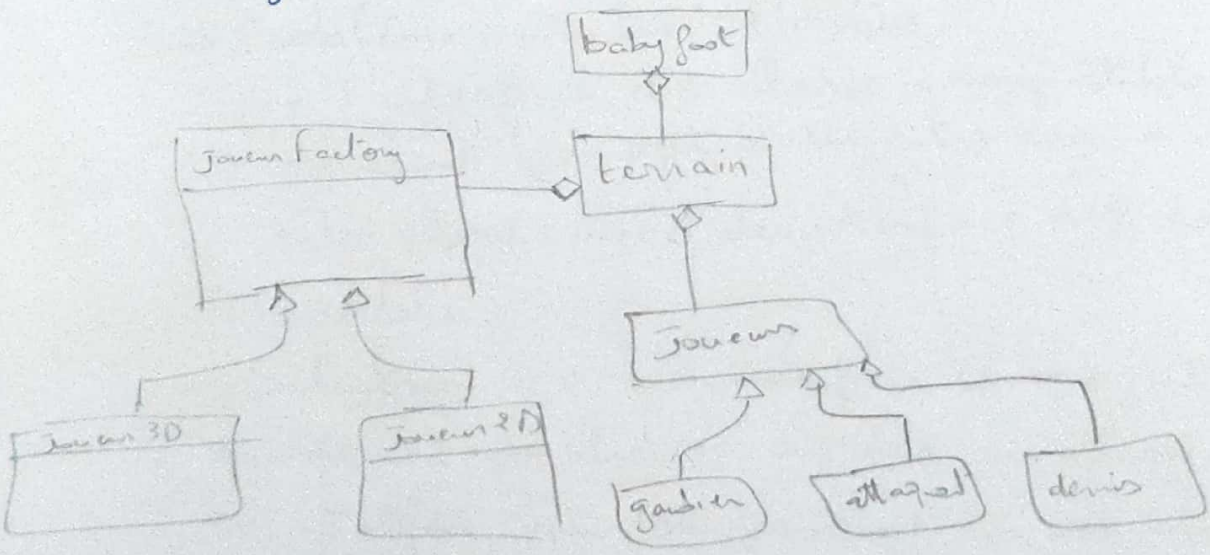
↳ décrit des solution simple pour des problème spécifique en OO.

* il existe plusieurs type de design pattern:

↝ Pattern de création
↝ Pattern de structure.
↝ " de comportement: les interaction entre objet.

⇒ Pattern de création:

• déléguer à d'autre classe la création.

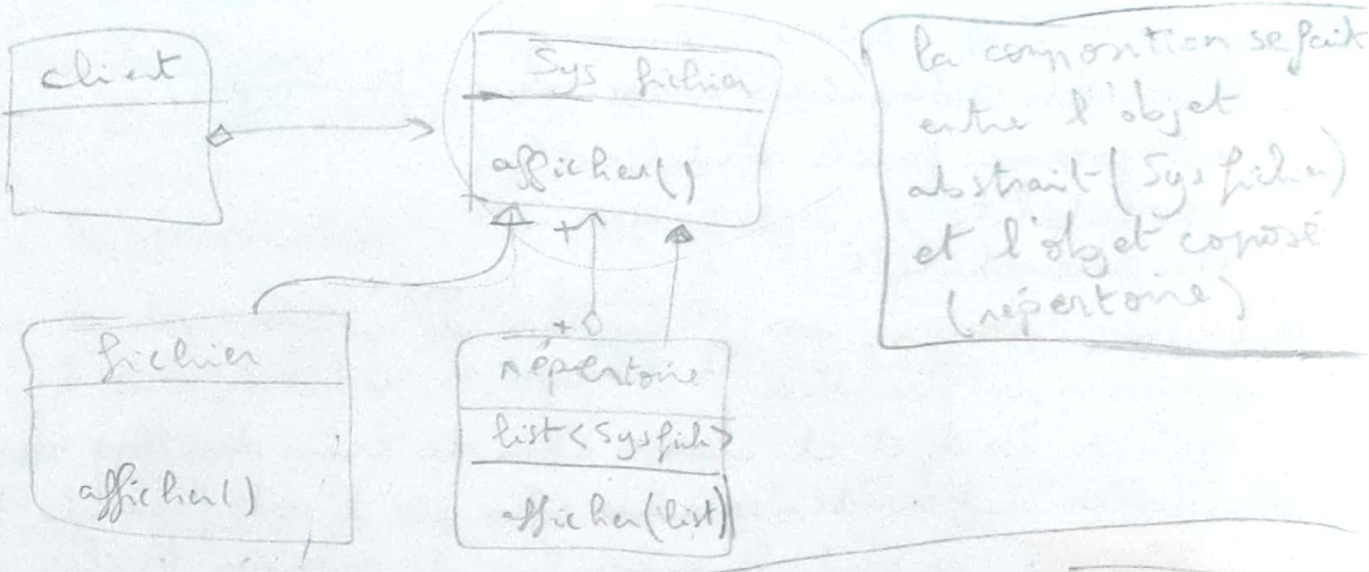


* ADL A *

* Critère de qualité:
  - validité: compatible avec cahier de charge.
  - Ergonomie: facile d'utilisation.
  - portabilité
  - Maintenabilité.

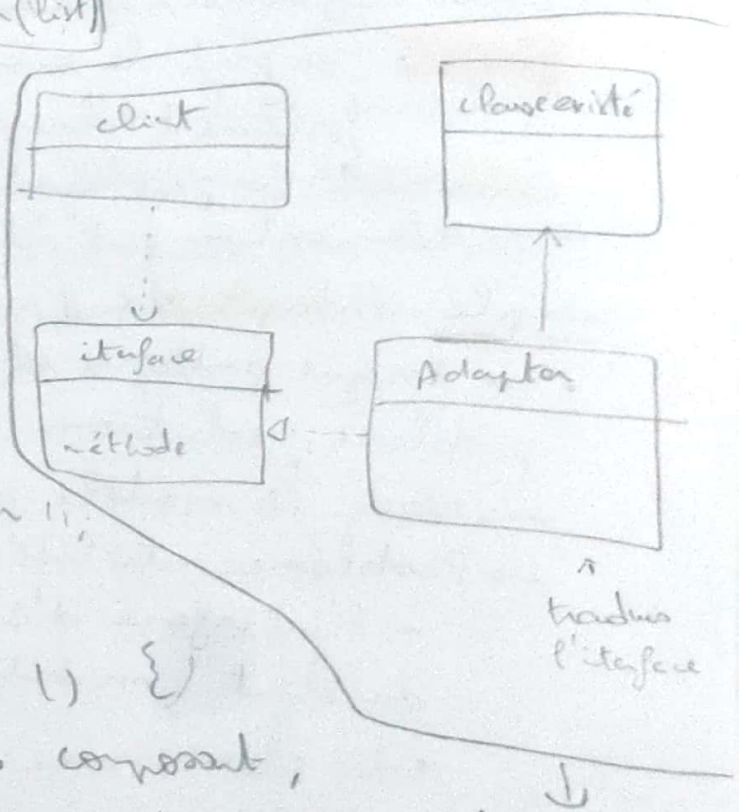
» Design Pattern: ⇒ nous permet de créer un logiciel de qualité
→ Pattern de structure:
relier les objet et classe afin de créer des sys important.
→ coller différent morceaux afin qu'il soit flexible et extensible
flexible: on peut le modifier de manière facile, pas forcément changer le code.
extensible: on peut ajouter des fonctionnalité.
cette composition peut être changer à l'éxecution
exemple ->Adapter < // réutilisation de code
* un designe pattern est representer par: Nom, type, problème, solution.
  → Nom: le modèle adaptateur.
  → Problème: utilisé lorsque:
    * l'interface d'1 classe incompatible avec l'interface du client (on doit placer entre eux une classe traductrice)
    * des classes avec des interface diff doivent fonctionner ensemble.
  → Solution: on a interface serveur, interface client qui doivent fonctionner ensemble on introduit une classe ou interface qui va traduire la communication entre ces 2 composants // exemple de pile et liste.
>Composite<
  → Nom: le modèle composition
  → Problème: on l'utilise pour créer des objet composée (arborescence) et on doit les traiter uniformément (de la même manière).

exemple composite




```
lister(composant c) {
  { c.lister();
  }
ArrayList<SysFichier> L
Composant b = new repertoire();
  L.add(b);
```

Sol:
```
public static void main () {
  composant [] sys = new composant,
  composant Nrep = new repertoire(),
  sys.add (Nrep);
  composant Nfichier = new fichier();
  sys.add (Nfichier);
  lister ( ){
    sys.lister();
  }
}
```



Pattern de
Comportements: traite la partie algorithmique.

deux type de pattern:

↝ Pattern de classe: utilisation d'héritage pour répartir le comportement

↝ Pattern d'objet: utilisation de l'association entre objets.

Pour décrire:

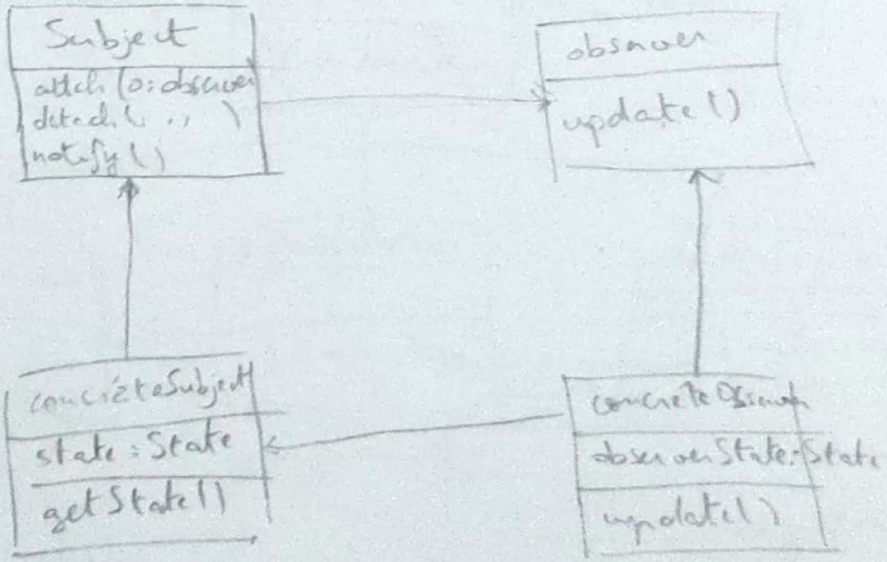
- la coopération entre groupe d'objet
- la dépendance entre objets
- l'encapsulation de comportement, déléguer les requêtes à d'autre objet.

exemple de pattern:

Observer: // ça se peut que un seul objet soit observé par plusieurs observateur.

on l'utilise pour:

* garder la cohérence entre des classe coopérant entre elles, en maintenant leurs indépendance.
* définir la dépendance one-to-many, en changeant l'état de l'objet, automatiquement et le mettant à jour.

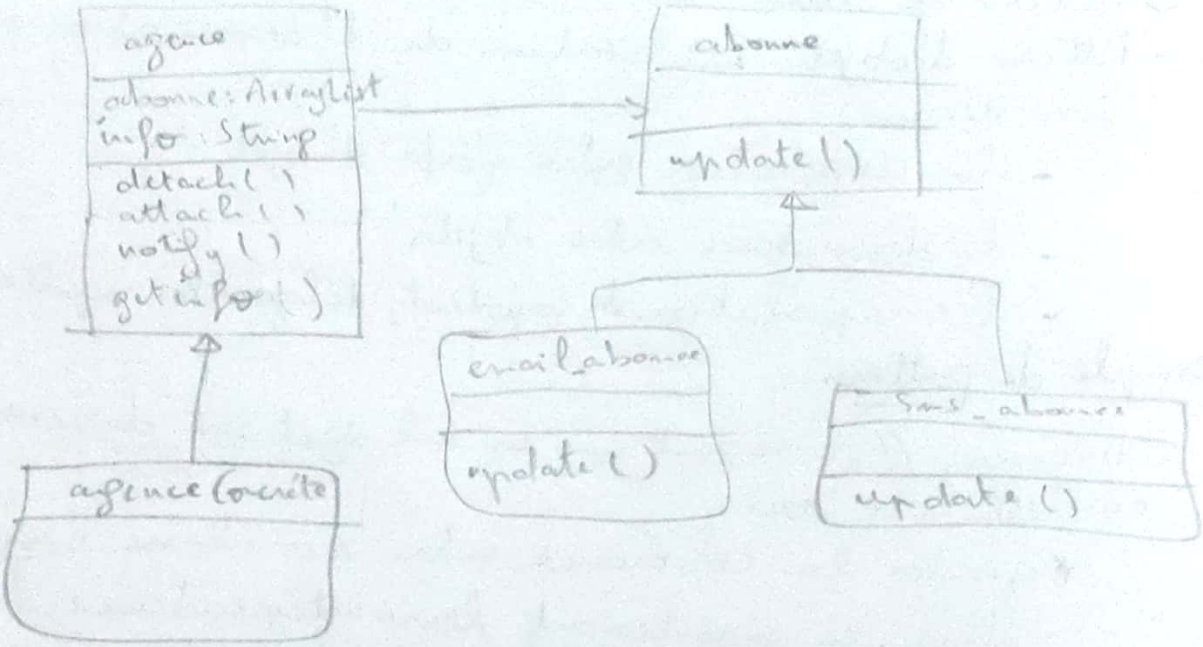


conséquence:

→ Avantage: Couplage abstrait entre un sujet et un observateur
support pour la communication par diffusion.

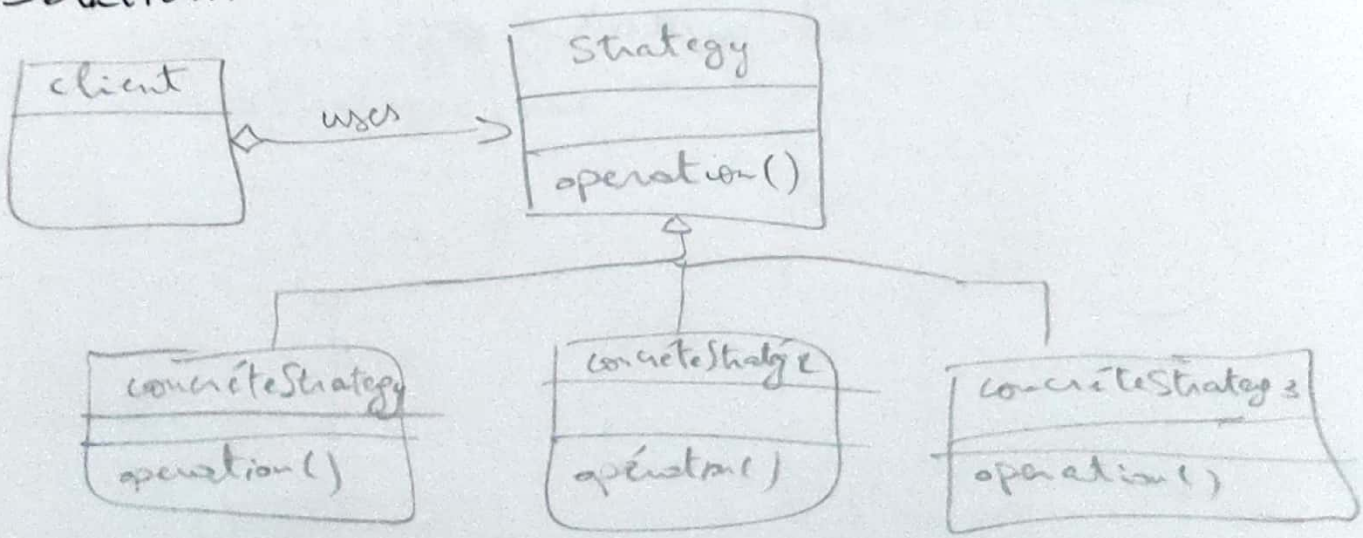
→ Limitation: Des mises à jour inattendues peuvent survenir avec des coûts importants.

exemple agence d'information :



⇒ Stratégie : // un objet a plusieurs comportement

Solution :

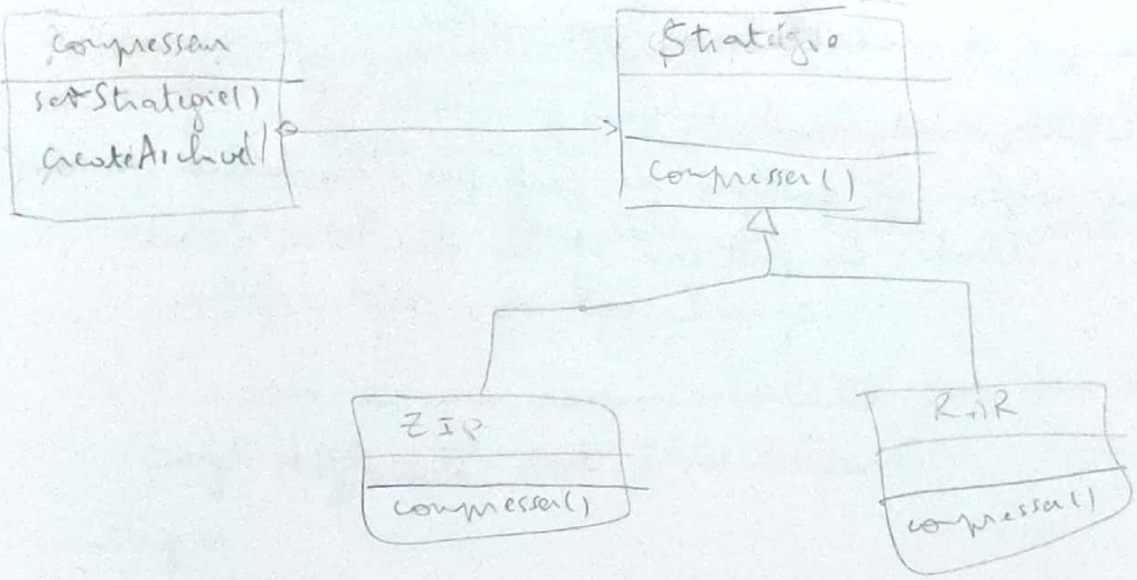


Conséquence :

→ Avantage :

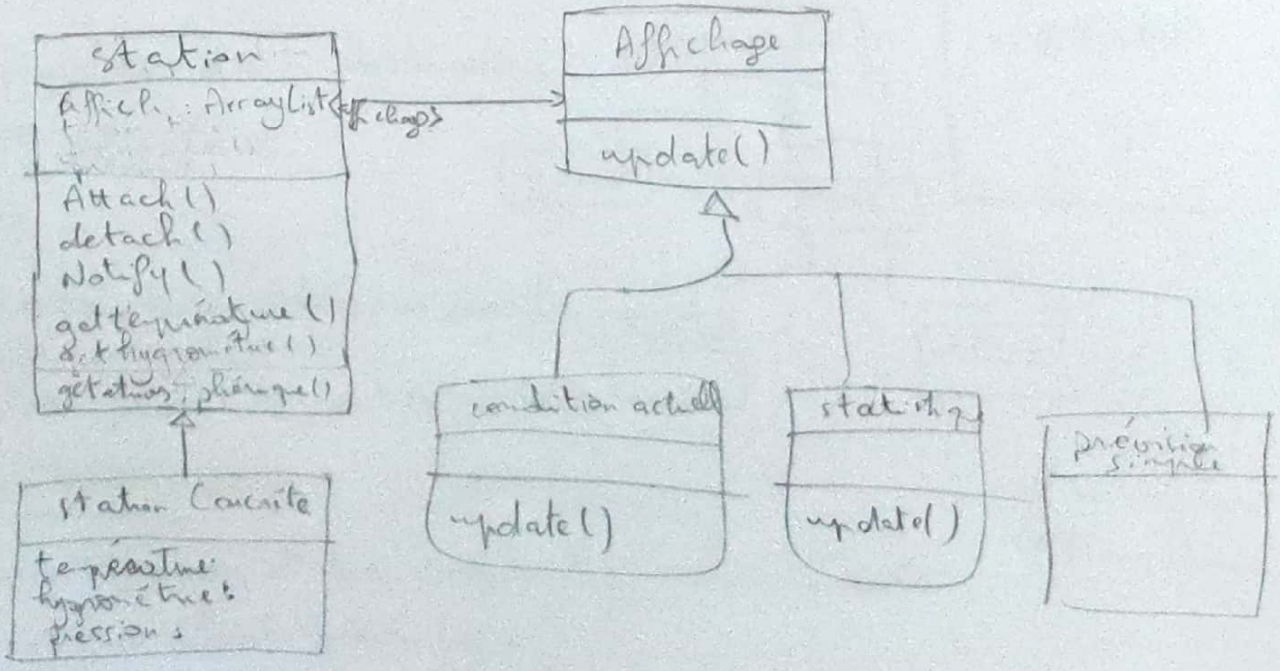
* expression hiérarchique de famille d'algo
* Elimination des tests.
* Selection dynamique d'algo.

Exercice 1 :




```
Public Main {
    public static void main(    ) {
        Compresseur c = new Compresseur();
        c.setStratégie (new Strategie ("ZIP"));
        c. SetStrategie (new Strategie ("RAR"));
    }
}
```

Exercice 2 :



Conclusion :

- le design pattern c'est une description d'une solution classique à un problème récurrent.
- c'est une technique d'architecture logicielle.
- ce n'est pas une brique : dépend de son environnement.
  - → une règle : un pattern ne peut pas s'appliquer mécaniquement.
  - → une méthode : un pattern est la décision prise.

Les principe ~~solid~~ SOLID:

- **Single responsability.**

"a class should have only one single responsability"

- Seulement si le rôle de cette classe a besoin d'une modification, on la change.
- l'accès à une responsabilité par des utilisateurs avec profil differents doit être séparé.

Avantages:

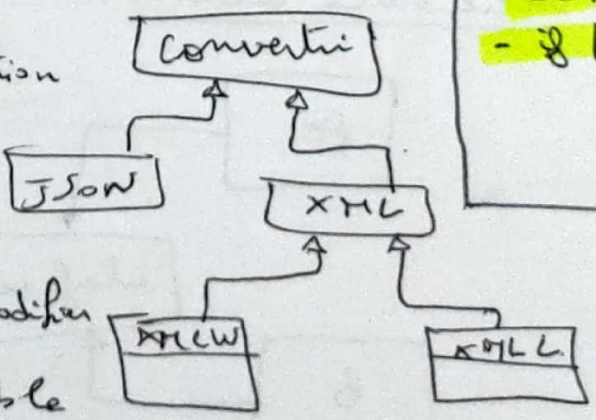
- conception faiblement couplé.
- des dependences moins nombreuses et plus légères.

- **O Open / close.**

"Software entities should be open for extension and closed for modification".

- modification d'une classe existante n'est pas permise.
- Surchage des méthodes concernés.
- ~~ex~~ ajout d'un niveau d'abstraction.
- Sol: ajouter une abstraction via des interface.

à éviter:
- switch
- cont
- if / else



Avantages:

- ajouter des fonctionnalités sans modifier le code existant.
- conception flexible, extensible

- **L Liskov substitution**

"Derived type must be completely substitutable for their base type"

// substituer = Remplacer

le fils ne doit pas casser l'abstraction de son parent (comporte comme son parent et rajoute d'autre chose).

Avantages LSP:

- Bien comprendre la logique métier.
- Renforce le principe de l'héritage en POO qui se base sur les fonctionnalités et non la définition de l'objet.

* I Interface Segregation : // quand on parle de I on parle de user

  * implémenter l'interface selon le besoin du client (~~client~~ user)

  → permet d'avoir :

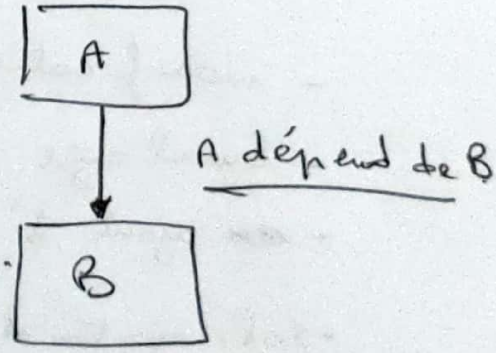
  * conception faiblement couplé.
  * " flexible et extensible.
  * donner au utilisateurs seulement ce qu'il a besoin

* D Dependency inversion :

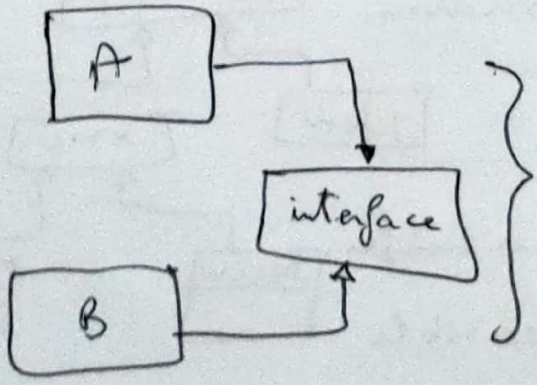
  ~~Bread~~ " Depends on abstraction, not on concretions ".

→ la dépendance :

Faut rendre A independant de B



→ dependance inversé :



A ne depends plus de B, nous avons inversé la dependance.

Avantages :

→ A ne voit plus B

→ B peut changer ou remplacer sans toucher A

→ possibilité de tester A en utilisant le principe des Mocks.

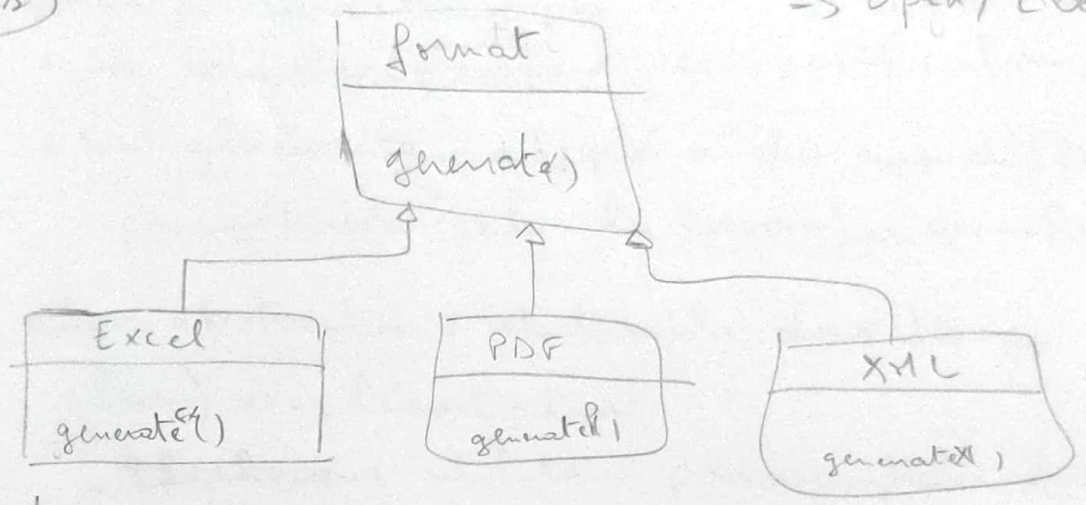
Les inconvénients:

* trop d'abstraction rend une partie du code inutile.

* guide de création d'interface: ---- diapo 35.

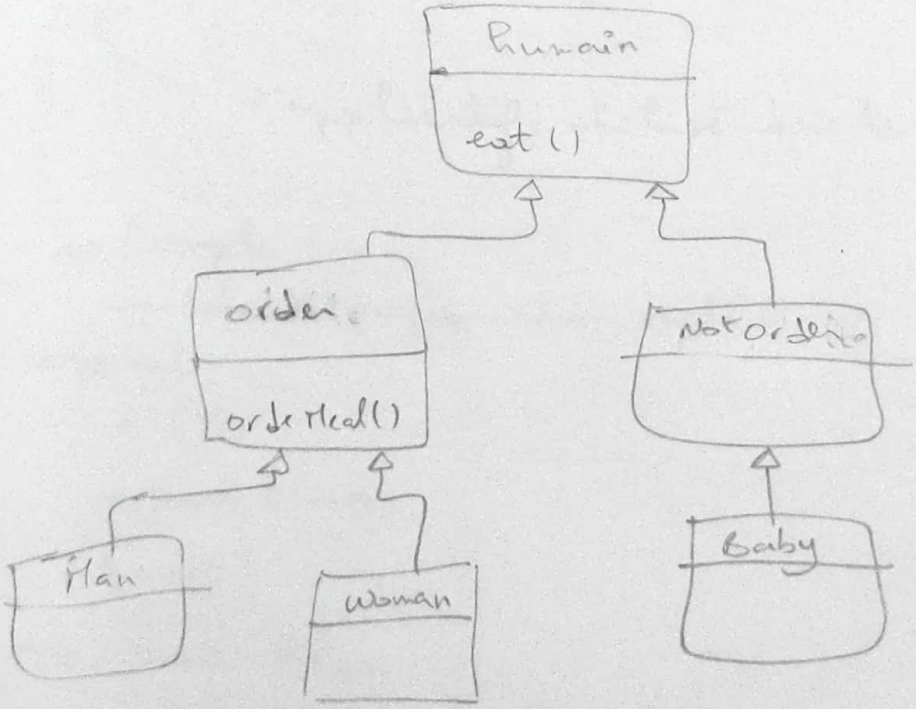
Exo 1)

-> Open / close.



Exo 2)

-> Liskov Substitution



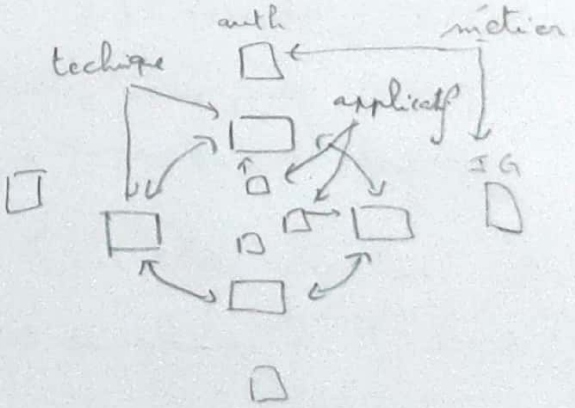
Cour Bean

* Programation orienté composants :
  → décomposer un problème en composants.
  → utilisé pour le logiciel par agrégation des briques déjà existé. développé

~> composant est caractérisé par :
  * sa robustesse : respect les spécification, sans bug.
  * sa généricité : adapté à des app différents (module paramétrable selon le besoin) => spécification
  * Son abstraction : utilisable dans des app diff (interface clair) => implémentation.

Composant peut être :
- technique : utilitaire pour manipuler des données.
- métier : définit les entités du domaine.
- applicatif : utilisé dans le traitement interne d'une app.

=> Exemple :
modèle de programation orienté composants :
  * EJB
  * Java Beans
  * .NET

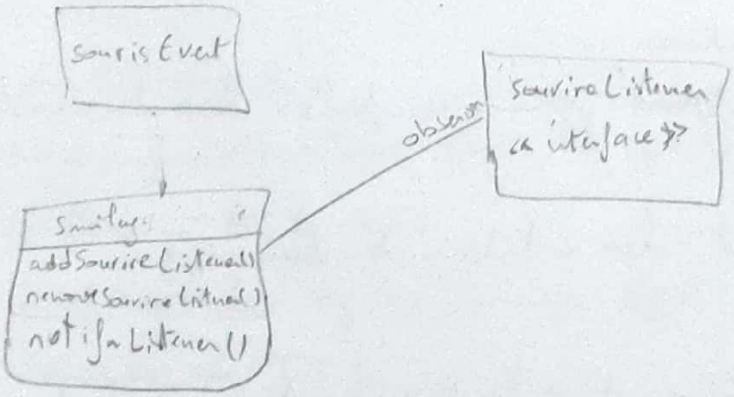


~> Java Bean :
caractéristique Java-Bean.
→ constructeur son paramétre.
→ implémenter l'interface Serializable.
→ attribut toujours privés et ont des getters et setters.
→ méthode utilisé par composants exterieurs doivent être public et gérer les accés concurrents.

↳ les Beans communiquent via le modèle événementiel, en utilisant (Listener / observer).

Java Bean : propriétés liées :

↳ appartient à un Java Bean et qui observé par un autre Java Bean, implement le design pattern Observer.




```
class abonnée () {
  private nom, intérêt;
  private String prénom, mail;

public abonnée () {

}

  public getnom () {
     return this.nom;
  }
  public getprenom () {
  } return this.prenom
```

```
setnom (String name) {
    this.nom = name;
}
setprénom (String firstname) {
   this.prenom = firstname;
}
```

Cours EJB 1-2

» la programmation par composant est basé sur le fait d'intégrer des composants entre eux pour composer un programme.

il existe 2 catégories :

- composant généraux : clients / IHM (Java Beans)
- " : les EJBs.

Couche présentation :

- client lourd (app Java pures)
- client léger (navigateur web)

Couche application :

- EJB utilisé dans la couche métier
- Coté serveur dans l'architecture N-tiers

=> le Serveur EJB :

- Service infrastructure gère la partie technique.
- le conteneur EJB gère :
  - gestion du cycle de vie EJB.
  - accès au EJB
  - sécurité d'accès.
  - accès concurrents
  - transactions (communication avec BDD)

→ Avantages EJB :

- concentrer sur la logique métier (car elle gère tous autre)
- l'environnement d'exécution prend en charge le traitement technique.
- séparer le code métier du code technique.

EJB composé de :

- Session Beans : (couche présentation - couche application)
  - extention du client sur le serveur.
  - pont entre " et data
- entité Beans : (couche application - BDD)
  - representation des data de la BDD, sert à accéder aux data.
- Beans orienté message : gestion asynchrone de events.

=> Session Bean:

- fournit un service à diff app clients.
- contient les service métier de l'app.
- le Bean est supprimé lorsque le client n'en a plus besoin.

↳ Statful (avec état)

* Session privé que pour cet utilisateur (client) => quand il demande un service qui offert juste pour ce client.

↳ Statless (sans état)

* quand le client demande un service qui est accéder en public
* ne conserve pas l'état entre deux invocations de méthode.
* consulter en lecture seule des données persistat.

Bean en fait un contructeur sans param pour faire une ligne dans BDD pour faire des setter.

Statful consomme de la mémoire car c'est des instance à créer.

=> Les sessions Beans forme un pont entre le client et la logique métier.

~> Entité Bean : est une classe Bean

↳ Entité Bean = persistance des données.

↳ simplifier la gestion de donnée.

↳ faciliter la sauvegarde en bdd.

↳ établir les cnx avec bdd.

* chaque objet métier est représenter par une entité Bean.

objet dont l'interaction avec un autre objet nous donne une fonctionnalité.

↳ représente les objets dont on a besoin.

-> Les EJBs utilise

-> le mapping objet/relationnel (récupérer les donnée Bdd et les stocké dans les attribut d'objet qui convient)

-> chaque entité Bean est mappé à une table en bdd.

entité Bean représente une entité de l'application.

session Bean représente une fonctionnalité de l'application.

exemple: app bancaire:

- un compte: objet persistant => entité Bean | - ajouter un compte: un service => session Bean.

Propriété entité Bean:

- chaque entité Bean représente une table
- les entités Bean doivent tous posséder un id unique (clé primaire)
- les attributs sont mappé aux champs d'entité Bean.

exemple:

aggrégation en POO.

Avantage Entité Bean:

-> code clair et plus facilement réutilisable.

-> entité Bean hérite des services. (grâce au conteneur EJB).

L'écriture d'Entité Bean:

-> peut être abstrait ou concrète.

-> hériter d'une classe entité que d'une classe non entité et inversement.

-> les méthodes, les attributs de la classe et à la classe ne doivent pas être final

-> rajouter l'annotation Entity

↳ mappé l'entité à une table

@Entity (name = "MyUser") : exemple. doit être unique.

~> l'annotation @Table:

- mappé une entité à une table ne possède pas le à non et déjà existante.
- l'attribut name précise le nom de la table.

- les champs persistants :

les attributs d'entité Bean sont par défaut persistant.

=> les annotations liée aux propriété simple:

@Basic : par défaut pour les attributs persistants.

@Lob (Large Binary Object) : préciser que l'attribut peut avoir une grande taille.

@Temporal : définir des propriétés dites temporelles

prend en paramètre un TemporalType:

* Date (la date) (Java.sql.Date)
* Time (l'heure) (Java.sql.TIME)
* Timestamp (temp précis) (Java.sql.TimeStamp).

@Ennumerated : spécifier un ensemble de valeur possible pour un attribut.

prend en paramètre un EnumType:

* EnumType.STRING (pour chaine de caractère)
* EnumType.ORDINAL (pour entier)

=> Annotation liée aux colonne simple:

@Column : précise le paramétrage des colonnes dans la table relationnelle.

> qlq attribut,

* name : le nom de la colonne.
* nullable : si le champs peut être nulle.
* updatable : si la valeur doit être mis à jour lors de l'exécution d'une requête update.
* precision : nombre de chiffre max.
* scale : nbr fixe de chiffre après la virgule.

=> Identifiant unique : l'annotation avant get ou la déf de l'attribut.

utiliser l'annotation @Id : pour dire qu'un attribut représente un id

@GeneratedValue : définit la stratégie de génération d'id.

→ les champs relationnelles :

@OneToOne : lier deux entités uniques indissociable.

@ManyToOne :

@OneToMany.

@ManyToMany.

⇒ L'unité de persistance :

* permet d'intégré les entité Beans dans l'app Java.

→ une unité persistance est caractérisé par :

- ensemble d'entité Beans.
- fournisseur de persistance (provider)
- une source de données (Datasource)

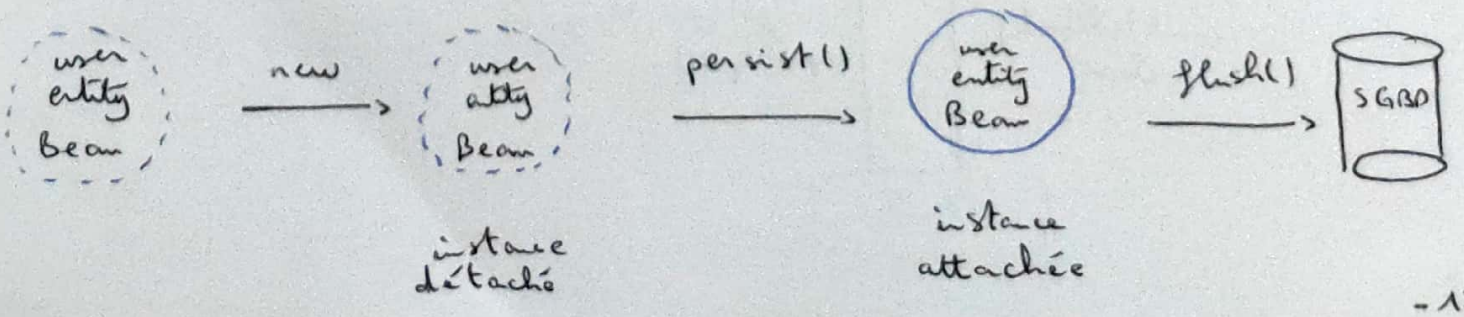
→ le rôle de l'unité persistance est :

- de savoir où et comment stocker les info.
- d'assurer l'unicité des instances de chaque entité persistate.
- de gérer les instances et leurs cycle de vie.

⇒ Gestionnaire d'entité :

définit les méthodes qui gère le cycle de vie des entité :

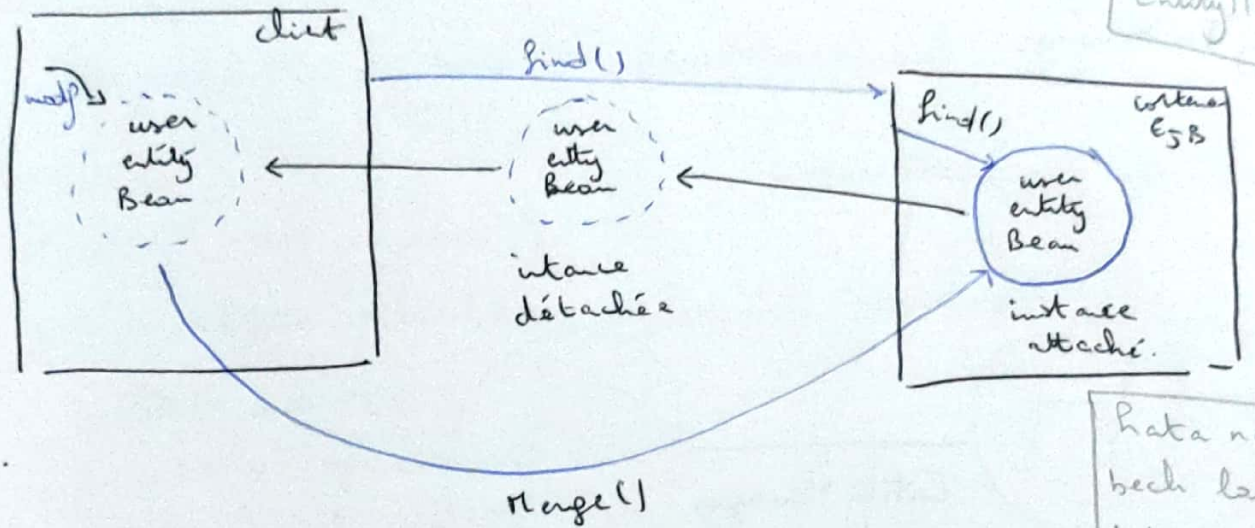
• persist() : enregistrer une entité : insérer dans bdd.



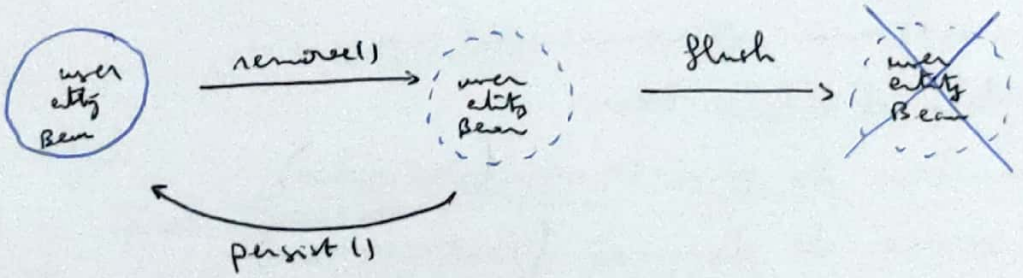
• find() :
récupérer les entité sauvegardé.

ex: entityManager.find(.....class, id);

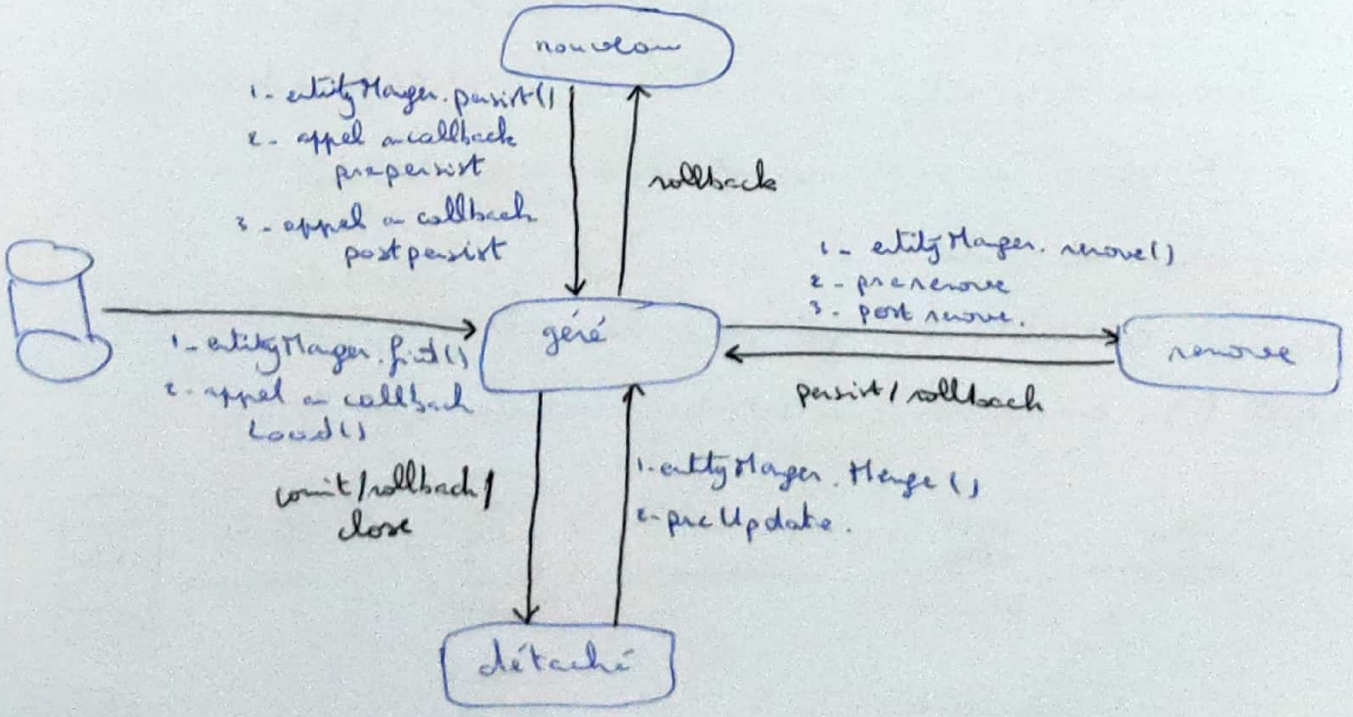
• Merge() : modifier les entité sauvegardé.



• remove() : demande la suppression.



Cycle de vie d'un entité Bean:



qd on crée plusieurs user par exemple, on fait entityManager.persist() pour chaque user créer et à la fin on met EntityManager.flush().

Pata na3 anlou flush() bech la bdd ta3ml bik mise à jour de data sinon on peut annulé normalement flush().

les annotation :

@PrePersist ou @PreRemove.
@PostPersist " @PostRemove.
@PreUpdate.

Entity Manager objet singleton c'est le contenu qui fait le new.

→ La persistance :

les propriété de base de persistance. xml.

Exercice :

① @Entity (name = "video")
public class videoBean implements Serializable {

private int ID;
private string nom;
private double taille;
private string emplacmt;

@ID
public int getID(){ return ID; }

② @Statful
public class Session-videoBean implemt ... {
@PersistanceContext (name = "video")
EntityManager em;

public video creation (string nom, int taille, string emplacmt){

création {
video v = new video ();
v.setNom (nom);
v.set taille (taille);
v.set emplacmt (emplacmt);

ajout {
em.persist(v);
em.flush();
return v;
}
}