Amphi 5

Design Pattern

Design pattern = motif patron récurrents d'architecture de programme qui sont orientés objets.

1] Factory

but = créer des objets par des méthodes static.

Peut générer des objets différent selon des critères

ex TD : classe Ensemble, petitEnsemble et grandEnsemble

2) Singleton

but = classe qui à au plus UNE instance

-> son constructeur est privée

-> but : centraliser le code : demande d'accès à une base de données

3) State

but = changer la représentation de l'objet

-> ex : passe d'un vector à table de hachage

-> Pourquoi ? : représentation dynamique de l'objet / améliorer performance

-> sur type pour les différentes représentation en interne

```
public Ensemble() {
  e = new PetitEnsemble();
}

public void inserer(int x) {
  if(!e.contient(x) && e.taille() + 1 > N)
    e = new GrandEnsemble(e);
  e.inserer(x);
}
```

Au début e est un petitEnsemble, mais si on insère alors qu'il est plein, on l'agrandi = on change dynamiquement la représentation de l'objet .

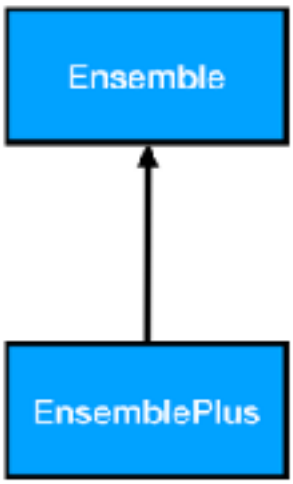
4) Bridge

-> On a deux hiérarchie différentes à intersection non vide

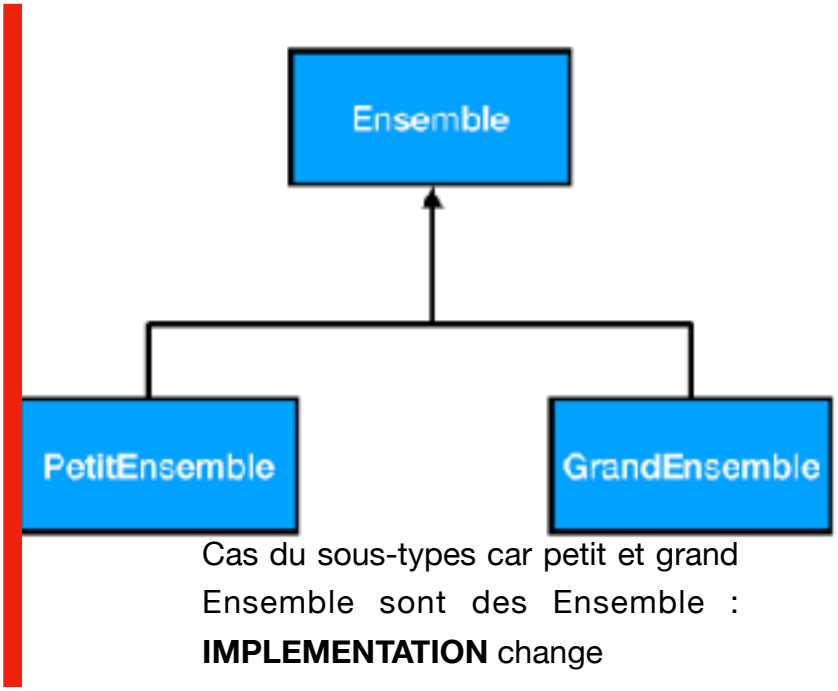
-> ex : une hiérarchie d'extension et une d'implémentations

-> mettre un pont de connexion entre deux hiérarchie parallèles.

Exemples avec Ensemble

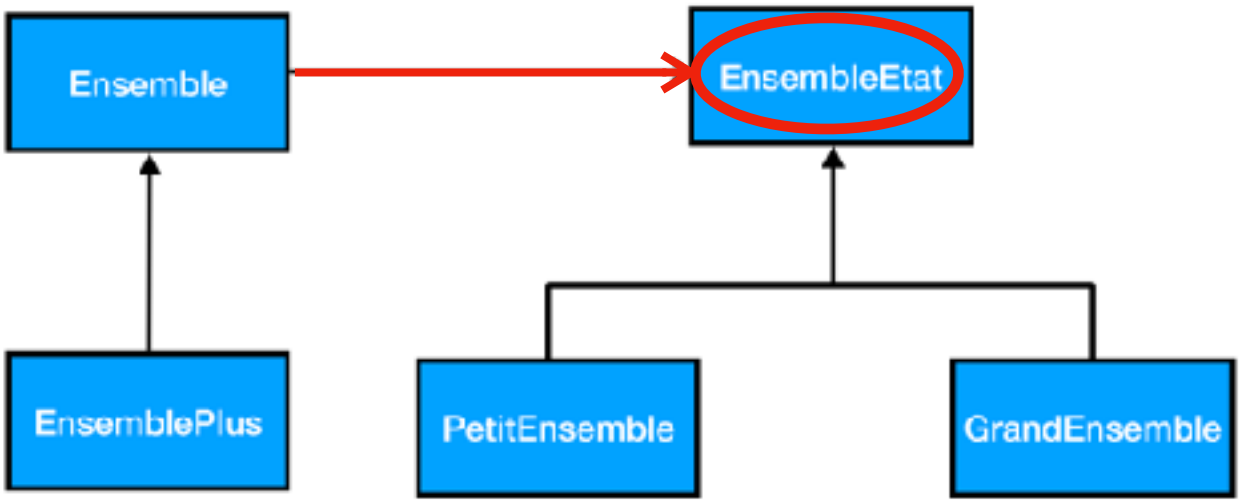


Cas de l'extension, Ensemble Plus est une interface plus spécifique à Ensemble : **OPERATION**



Cas du sous-types car petit et grand Ensemble sont des Ensemble : **IMPLEMENTATION** change

-> Etablir la communication = Ajout d'un champs (pont) qui sert de communication dans la classe Ensemble



5] Strategy and command

But : passer en paramètre une fonction pb : on peut pas écrire :
“ int somme = iterer(new int[] {1, 2, 3, 4, 5}, 0 , +); “ (PAS EN JAVA)

Solution -> closure

```
interface Fonction<R>{    // R = type de renvoie
     <R> laFonction();    // Et les paramètres
}
```

Ajout des paramètres :

```
interface Fonction<R,P,R,……>{    P,R…. type des  paramètre
     <R> laFonction(P param1, R param2, ….);
}
```

Problème => long et lourd et éviter d’avoir une classe pour un nombre différent de paramètres.

Solution : les paramètres sont des attributs dans la classe (exemples avec somme). On a donc une classe pour chaque opération (produit, somme etc…)

```
class Somme implements Fonction<Integer> {
     private Integer x, y;
     Somme(Integer x, Integer y) {
          this.x = x;
          this.y = y;
     }
     @Override
     Integer laFonction() {
     return x+y;  }
}
```

```
exemple 3+5 : (new Somme(3,5)).laFonction();
```

6) Observer

-> déclencher une opération lorsqu'un objet change d'état.

-> exemples : rafraichir l'écran

-> L'objet intéressé va générer une liste de personnes à contacté, qui sont intéressé par son changement via l'appel de la méthode `update()`

```
interface Observer {
     void update(State newState); // prévenir les personnes
intéressé
}

interface Observed {
     void subscribe(Observer o);  //s'inscrire à une observation
     void unsubscribe(Observer o); //se désinscrire à une
observation
}
```

Collection

Il existe plusieurs façon de groupe et de stocker des objets existants. Le plus basique sont les tableaux

=> Problème : la structure peut ne pas être adapté à la situation ou à l'utilisation.

Tableau

Point fort : Accès et insertion **rapide** O(1) -> `t[i]=4`

Point faible : la recherche et le tri peuvent vite être long O(n)

Vecteur

Objet qui représente un tableau, représentation caché à l'utilisateur.

Point fort : Accès et insertion rapide + taille DYNAMIQUE -> `t[i]=4`

Point faible : la recherche et le tri peuvent vite être long O(n). Lorsqu'on agrandit le tableau, il y a **recopies** des valeurs d'indices inférieurs.

LinkedList<T>

Chaque élément est chainé à son prédécesseurs et son successeur.

Point fort : insertion O(1) on ajoute un élément = créer un nord et changer les pointeurs.

Point faible : accès, recherche en O(n)

Pile

Deux opération : push = ajouter / pop = retirer

Point fort : opérations rapide O(1)

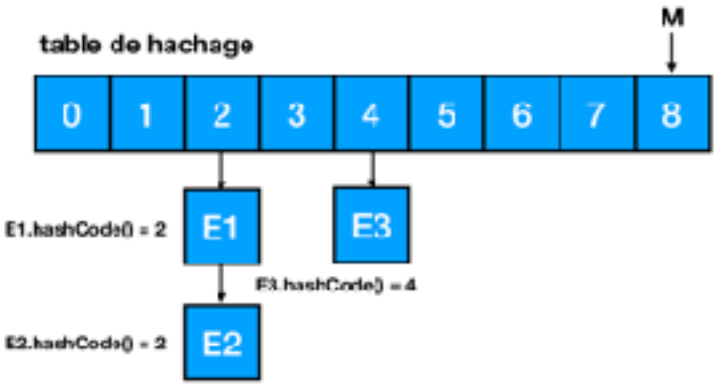
Point faible : seulement 2 opérations

File

principe d'une file d'attente : le premier arrivé = le premier servit

Point fort : doublement liée (pointeur vers l'avant et pointeur vers l'arrière de la queue)

Point faible : comme pour la pile



HashSet<E>

Chaque élément à une code (code de hachage) qui est calculé par `hashCode()`

Point fort : insertion /recherche /suppression / accès = O(1)

Point faible : gérer très mal le parcours régulier.

Itérateur

parcours d'une collection : objet abstrait

exemples d'utilisation : les boucles "Pour chaque" = `for(E e : collection)`

A chaque utilisation sa collection :

- Ordre important = Listes

- Ordre pas important = Ensemble

- Accès rapide important = tableau ou vecteur