

Design Pattern

Définition : solution apportée à un problème récurrent

Il existe 3 catégories de pattern

- Construction (Singleton, Factory, Abstract Factory ...)
- Structuration (Adapter, Composite, Decorator ...)
- Comportement (Chain of responsibility, Iterator, Strategy)

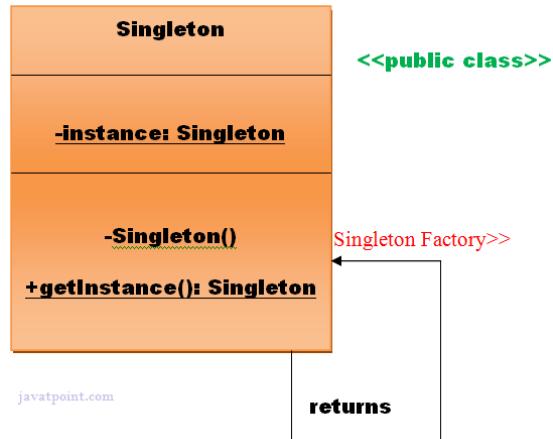
I – Design Patter de Construction

A – Singleton

=> Une classe doit s'assurer qu'une seule instance est créée et qu'un seul objet peut être utilisé par toutes les autres classes.

Utilisation :

- Économise la mémoire car l'objet n'est pas créé à chaque demande.
- Dans les applications multi-thread et les bases de données.
- La journalisation, la mise en cache, les pools de threads, les paramètres de configuration...



```
public class MonSingleton{  
    private static MonSingleton instance = null ;  
    private MonSingleton (){  
        ...  
    }  
    static public MonSingleton getInstance (){  
        if ( instance == null )  
            instance = new MonSingleton() ;  
        return instance ;  
    }  
}
```

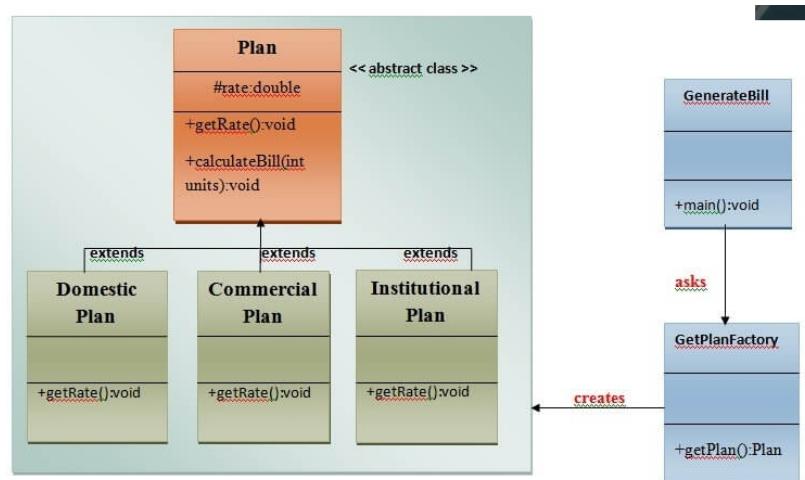
B – Factory

=> Crée une instance d'une classe pour un identifiant donné

Le Factory Pattern indique que les sous-classes sont responsables de la création de l'instance de la classe. Cela signifie que le code interagit uniquement avec l'interface ou la classe abstraite résultante, de sorte qu'il fonctionnera avec toutes les classes qui implémentent cette interface ou qui étendent cette classe abstraite.

Utilisation :

- Une classe ne sait pas quelles sous-classes elle devra créer
- Une classe veut que ses sous-classes spécifient les objets à créer.
- Les classes parentes choisissent la création d'objets à ses sous-classes.



```
public class nomCouleur {
    private static Hashmap < String , Couleur > map ;
    static{
        map= new Hashmap < > () ;
    }

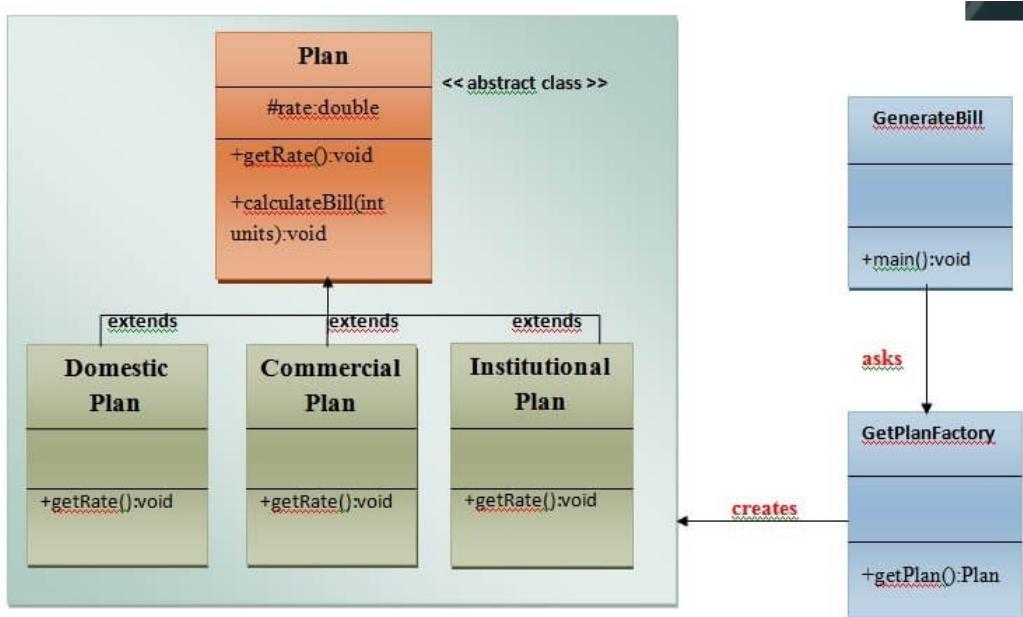
    private Couleur ( String nom ) {
        nomCouleur=nom ;
    }
    public String toString () {
        return « Couleur : » + nomCouleur ;
    }
    public static getInstance ( String nomCouleur ) {
        Couleur couleur = map.get( nomCouleur ) ;
        if ( couleur == null ){
            couleur = new Couleur ( nomCouleur ) ;
            map.put(nomCouleur, couleur) ;
        }
        return couleur ;
    }
}
```

C – Abstract Factory

Il suffit de définir une interface ou une classe abstraite pour créer des familles d'objets liés, mais sans spécifier leurs sous-classes concrètes. C'est la raison pour laquelle le modèle de fabrique abstraite est un niveau plus élevé que le modèle de fabrique.

Utilisation :

- Le système doit être indépendant sur la création, la composition et la représentation
- La famille d'objets apparentés doit être utilisée ensemble, cette contrainte doit être appliquée
- Fournir une bibliothèque d'objets qui ne montre pas les implémentations et ne révèle que les interfaces.
- Lorsque le système doit être configuré avec une famille d'objets parmi plusieurs.



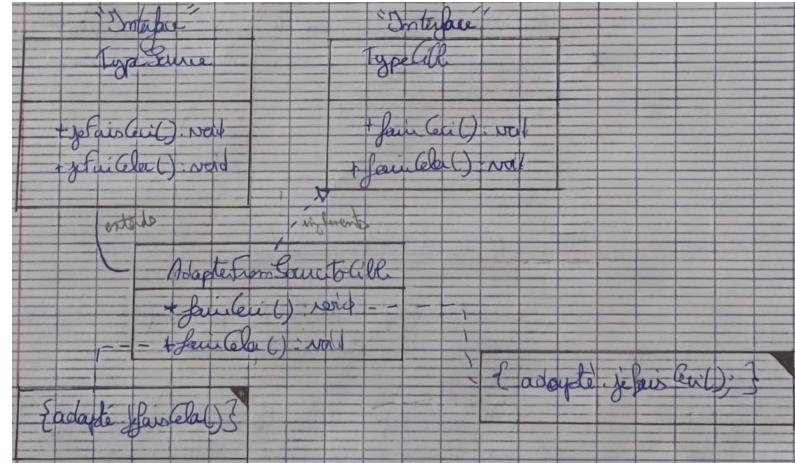
II – Design Patter de Structuration

A – Adapter

=> soumettre à un certain traitement qui prend en entrée un type différent et non-compatible par polymorphisme

Utilisation :

- Lorsqu'un objet doit utiliser une classe existante avec une interface incompatible.
- Lorsque qu'on créer une classe réutilisable qui coopère avec des classes qui n'ont pas d'interfaces compatibles.



B – Composite

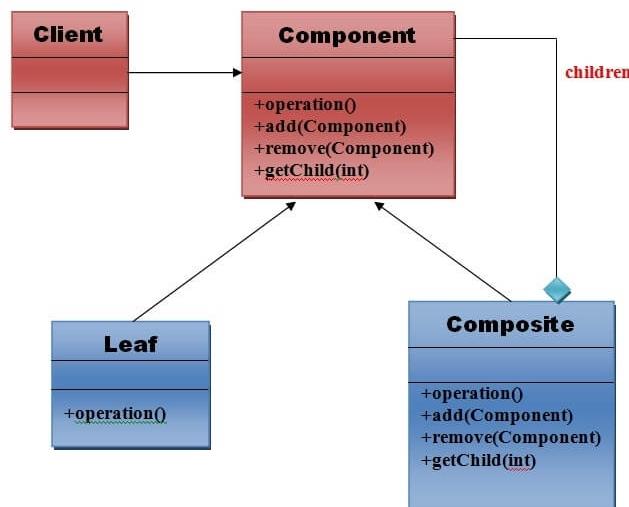
Permet d'opérer de manière générique sur des objets qui peuvent ou non représenter une hiérarchie d'objets (sous forme d'arbre).

- héritage entité simple (leaf) et entité composée (composé)
- création d'une association => Composite puisse contenir des composants

Utilisation :

- Lorsque qu'on veut représenter une hiérarchie complète ou partielle d'objets.
- Lorsque qu'on ajoute les responsabilités dynamiquement aux objets individuels sans affecter les autres objets.
- Lorsque la responsabilité d'un objet peut varier de temps en temps.

Exemple d'utilisation : translation d'une figure dans un groupe de figure

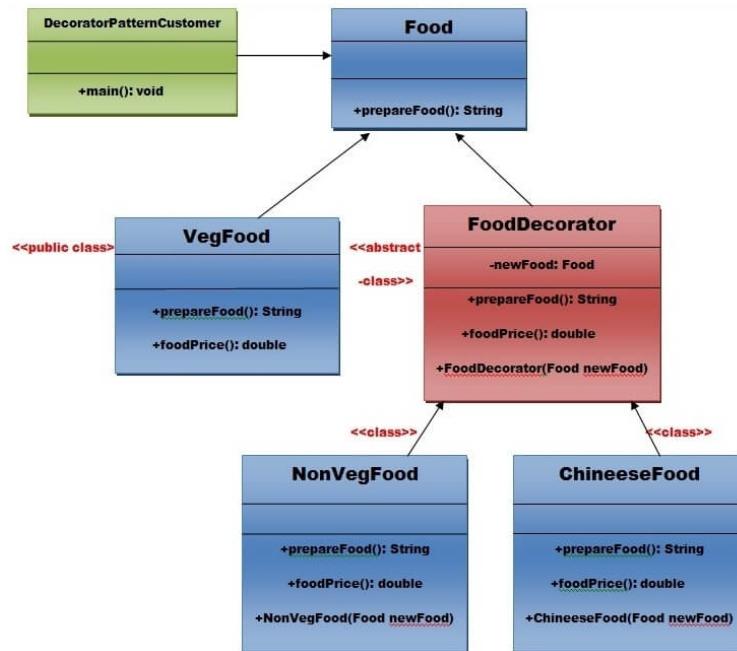


C – Decorator

=> Attacher dynamiquement des responsabilités supplémentaires flexibles à un objet (utilise la composition au lieu de l'héritage.)

Utilisation :

- Lorsqu'on ajoute de manière transparente et dynamique des responsabilités aux objets sans affecter les autres objets.
- Lorsque vous voulez ajouter des responsabilités à un objet que vous pourriez vouloir modifier à l'avenir.



III – Design Patter de Comportement

A – Iterator

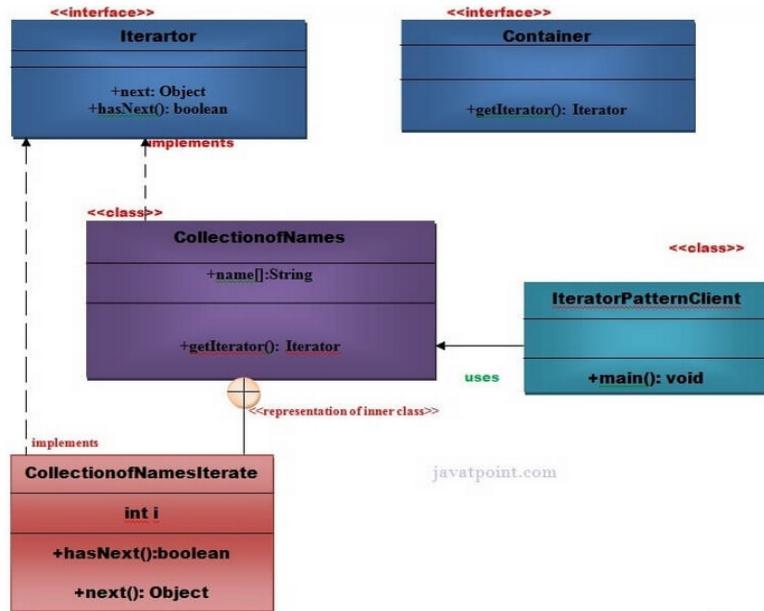
=> Rendre uniforme le parcours d'un élément, structure quelconque

Utilisation :

- Lorsqu'on accède à une collection d'objets sans exposer sa représentation interne.
- Lorsqu'il est nécessaire de prendre en charge plusieurs traversées d'objets dans la collection.

Exemple : Itération des nombre premier

```
private int taille,dernier_premier,currentIndex=1 ;  
  
private static Array <Integer> cache = new ArrayList <> () ;  
  
public IteratorPremier ( int taille){  
    this.taille=taille ;  
    current=0 ;  
}  
  
public boolean hasNext(){  
    return current<taille ;  
}  
  
public static boolean isPremier ( Integer x ) {  
    int l = (int) Math.sqrt(x) ;  
    if ( x%2==0) return false ;  
    for ( int i=0 ; i ≤ l/2 ; i++ ) {  
        if ( x % ( i * 2 + 1 ) == 0 ){  
            return false ;  
        }  
    }  
    return True ;  
}  
  
public Integer next() {  
    if(current<=cache.size()){  
        return cache.get(currentIndex++) ;  
    }  
    int x = dernier_premier + 1 ;  
    while(!isPremier(x)){  
        x ++ ;  
    }  
    dernier_premier = x ;  
    currentIndex ++ ;  
    cache.add(x) ;  
    return x ;  
}  
}
```



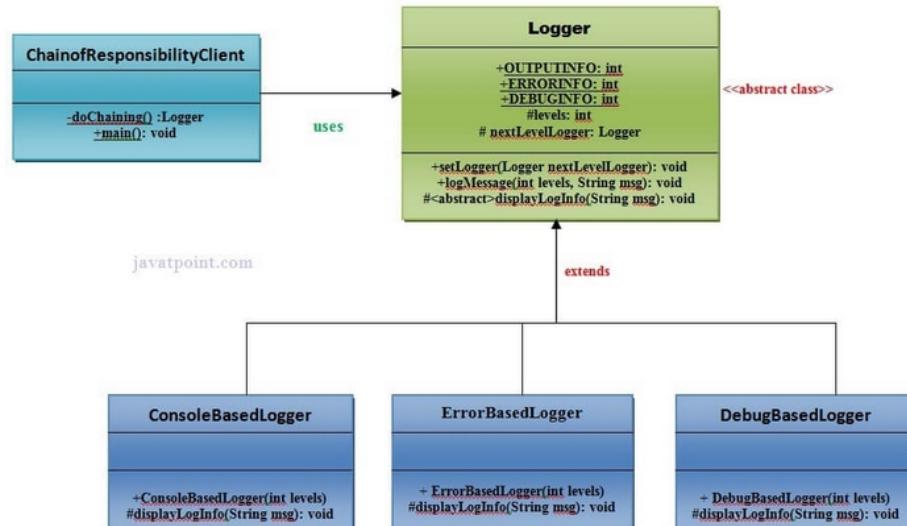
B – Chain of responsibility

Dans la chaîne de responsabilité, l'expéditeur envoie une demande à une chaîne d'objets. La demande peut être traitée par n'importe quel objet de la chaîne. Si un objet ne peut pas traiter la demande, il la transmet au récepteur suivant et ainsi de suite.

Utilisation :

- Lorsque plusieurs objets traitent une requête et que le gestionnaire est inconnu.
- Lorsque le groupe d'objets qui peuvent traiter la demande doit être spécifié de manière dynamique.

Par exemple, un distributeur automatique de billets utilise le modèle de conception Chaîne de responsabilité dans le processus de distribution d'argent.

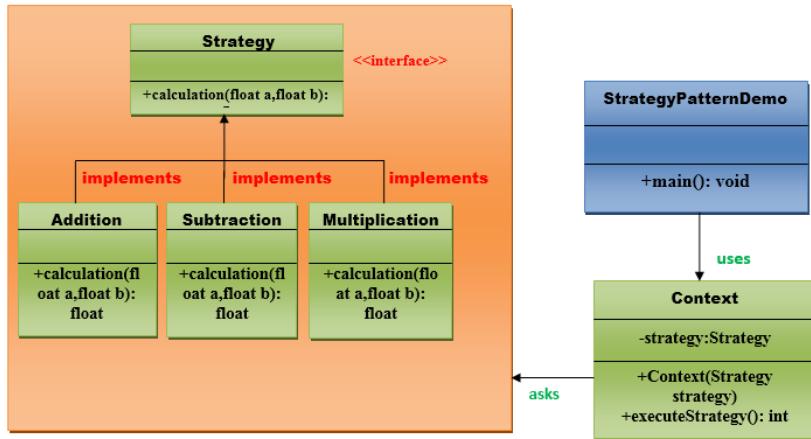


C – Strategy

Le pattern Strategy encapsule chaque fonctionnalité les rendant ainsi interchangeables sans modifier l'application. Il prévoit que le constructeur ou un setter permette d'attribuer ou d'échanger la stratégie

Utilisation :

- Lorsque les classes multiples ne diffèrent que par leurs comportements
- Lorsque vous avez besoin de différentes variations d'un algorithme.



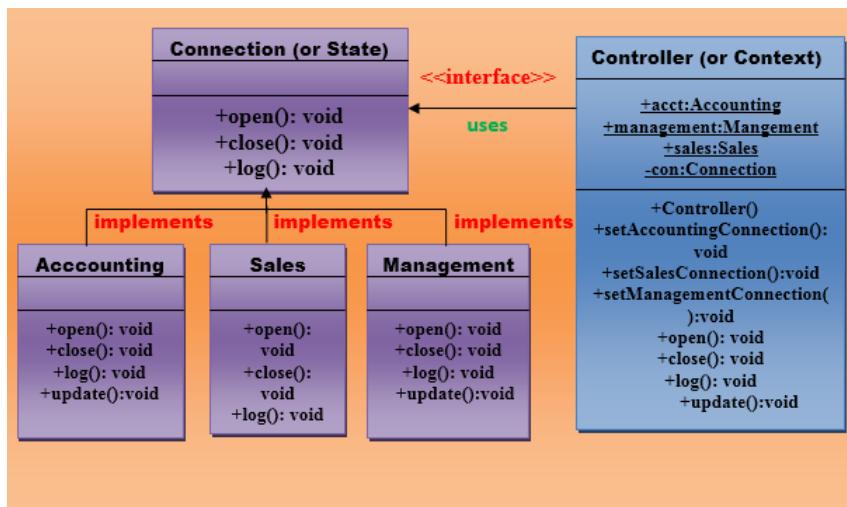
D – State

=> Cas particulier du pattern Strategy

On crée des objets qui représentent différents « états » et un objet de contexte dont le comportement varie dynamiquement en fonction des changements de son objet d'état.

Utilisation :

- Lorsque le comportement d'un objet dépend de son état et qu'il doit être changé dynamiquement



E - Observer

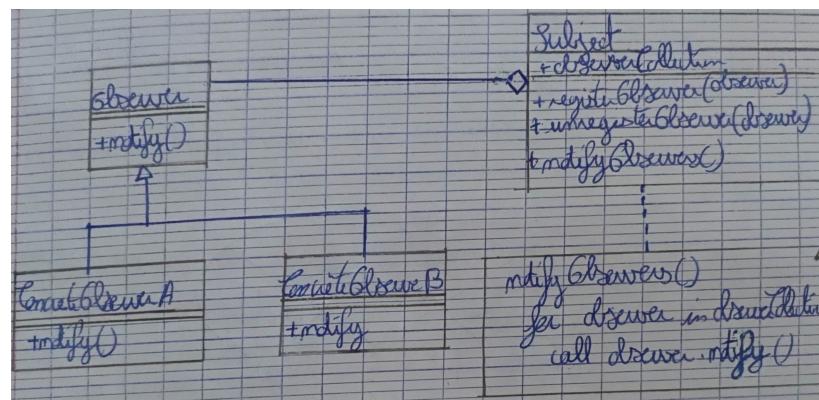
Lorsqu'un objet change d'état, tous ses dépendants soient notifiés et mis à jour automatiquement".

Avantages :

- Il décrit le couplage entre les objets et l'observateur.
- Il fournit le support pour la communication de type diffusion.

Utilisation :

- Lorsque le changement d'état d'un objet doit être reflété dans un autre objet sans que les objets soient étroitement couplés.
- Lorsque le cadre que nous écrivons et qui doit être amélioré dans le futur avec de nouveaux observateurs avec un minimum de changes.



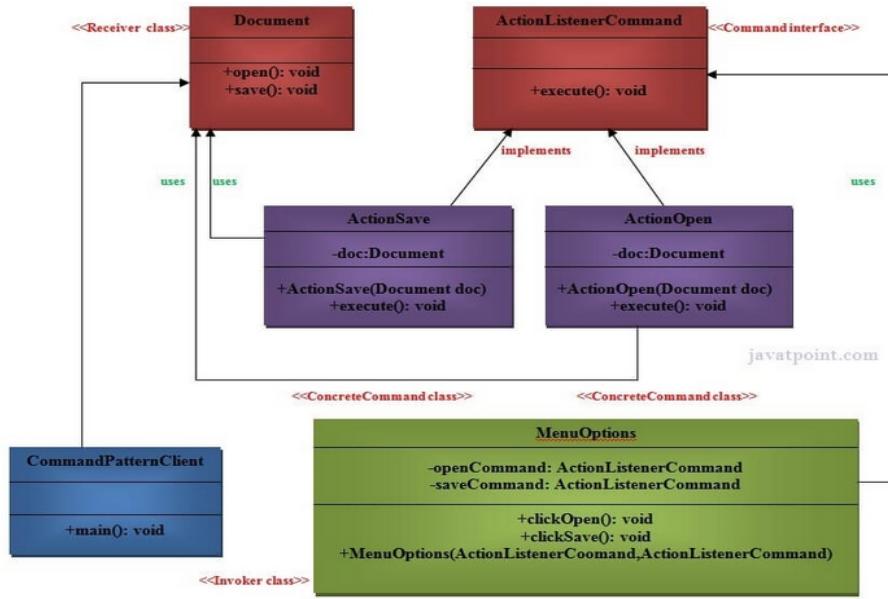
F – Command

Le pattern Command encapsule une requête dans un objet « commande » et la passer à l'objet invoker. L'objet invoker recherche l'objet approprié qui peut traiter cette commande et passe la commande à l'objet correspondant et cet objet exécute la commande.

- Il sépare l'objet qui invoque l'opération de l'objet qui exécute réellement l'opération.
- Il facilite l'ajout de nouvelles commandes, car les classes existantes restent inchangées.

Utilisation :

- Lorsque vous devez créer et exécuter des requêtes à des moments différents.
- Lorsque vous avez besoin de supporter des fonctionnalités de retour en arrière, de journalisation ou de transaction.



IV – MVC

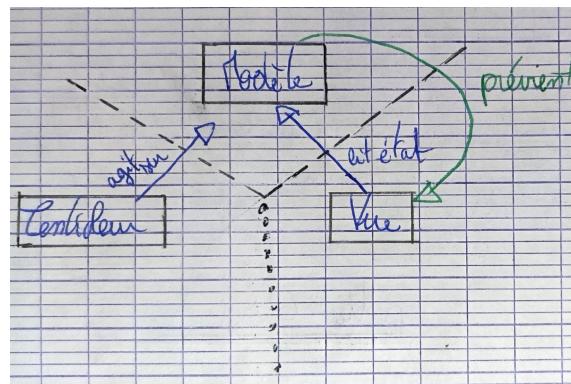
Modèle : état et comportement

Contrôleur : Listeners (clavier, souris) et composant graphique qui génère les events

Vue : différents éléments de Swig

Principe :

- Le modèle doit tout ignorer de la Vue et du Contrôleur.
- Le contrôleur possède une référence sur le Modèle pour agir dessus.
- Vue possède une référence sur le modèle seulement en lecture



Classe abstraite

- classe qui sert de classe mère, pas d'instance (pas de new)
- ex : Personne p = new Homme()
- mettre « super » dans le constructeur de la classe

Méthode abstraite

Si une classe contient une méthode abstraite, la classe devient une classe abstraite.