Cours - Patron de conception - #6

MVC (Modèle-Vue-Contrôleur)

Guillaume Santini

8 février 2024

Plan

1. Motivations
   - Objectifs
   - Architecture
   - Principes mis en œuvre
2. Mauvaises conceptions
   - Cas d'étude
   - Une mauvaise conception
3. Une meilleur conception
   - Préconisations
   - Une meilleur conception
4. Une bonne conception : le patron MVC
   - Le patron Modèle-Vue-Controleur
   - Principes de conception
5. Credits

Objectifs

Structuration d'une application avec interface utilisateur

- fournir une séparation claire des responsabilités dans une application, pour faciliter la maintenance et l'évolutivité du code.
- Permettre un développement modulaire (les changements dans une partie du système n'affectent pas les autres parties).
- Favoriser la réutilisation du code et permettre une meilleure gestion des complexités liées aux applications logicielles

Objectifs

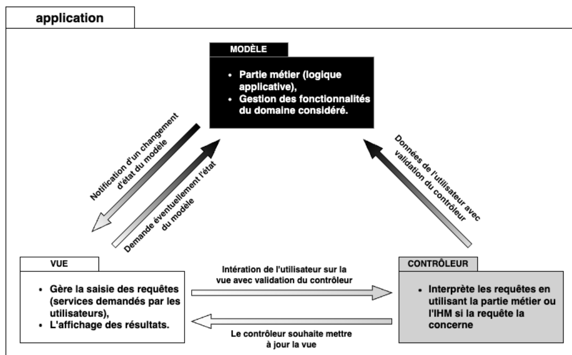
Structuration d'une application avec interface utilisateur

- fournir une séparation claire des responsabilités dans une application, pour faciliter la maintenance et l'évolutivité du code.
- Permettre un développement modulaire (les changements dans une partie du système n'affectent pas les autres parties).
- Favoriser la réutilisation du code et permettre une meilleure gestion des complexités liées aux applications logicielles

MVC propose une architecture logicielle

C'est un méta patron qui intègre l'utilisation des autres patrons (*i.e.* Stratégie, Observateur, . . .).

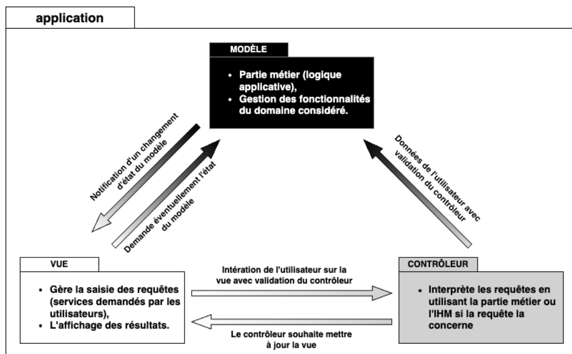
Architecture MVC (Modèle-Vue-Contrôleur)



Modèle : responsable de la gestion des données et de la logique métier

- représente les données de l'application et les règles métier associées,
- gère l'accès aux données, effectue les opérations nécessaires et notifie la vue de tout changement dans les données.

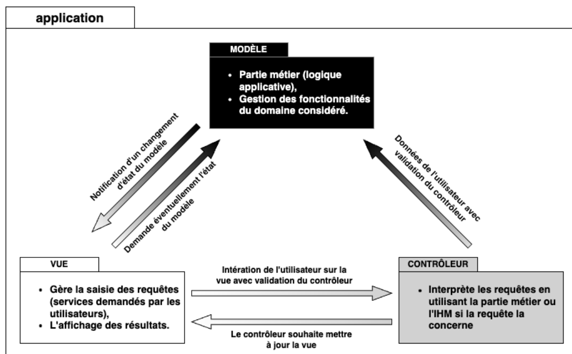
Architecture MVC (Modèle-Vue-Contrôleur)



Vue : responsable de l'affichage des données par l'IHM

- reçoit les données du modèle et les présente à l'utilisateur.,
- ne manipule pas directement les données,
- informée des changements du modèle elle met à jour l'IHM.

Architecture MVC (Modèle-Vue-Contrôleur)



Contrôleur : gère le flux de contrôle de l'application et la coordination modèle-vue

- agit comme un intermédiaire entre le modèle et la vue,
- reçoit les entrées de l'utilisateur et les traduit en actions à effectuer sur le modèle ou la vue.

Principes mis en œuvre

Exemple de modularité

- Éclater une application en 3 composants logiciels permet dans la cas de modification de l'un d'eux, de ne pas remettre en question les autres composants.
- Pour une même application d'accès à une base de données, on peut changer l'interface homme machine (vue) suivant que l'utilisateur possède des privilèges administrateur ou non.

Principes mis en œuvre

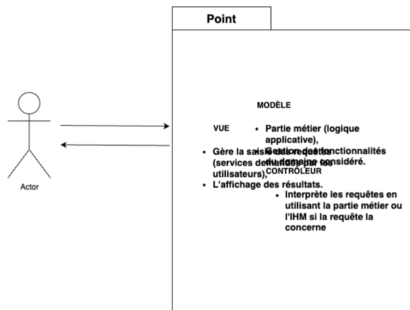
Exemple de modularité

- Éclater une application en 3 composants logiciels permet dans la cas de modification de l'un d'eux, de ne pas remettre en question les autres composants.
- Pour une même application d'accès à une base de données, on peut changer l'interface homme machine (vue) suivant que l'utilisateur possède des privilèges administrateur ou non.

Principes SOLID

- Single responsability,
- Open/Closed,
- Interface segregation,
- Depedency inversion,
- Tout principe utilisé dans les patrons intégrés à l'architecture,

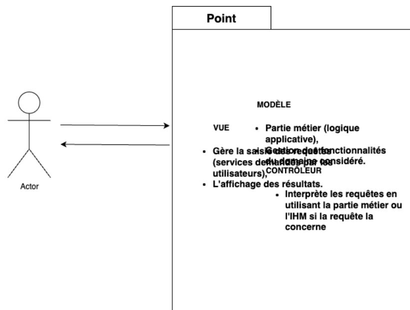
Cas d'étude



Application de gestion de déplacement et de l'affichage d'un `Point`

- Application de gestion d'un point dans un outil de dessin vectoriel,
- Gérer les coordonnées d'un point, son déplacement, son affichage, ...
- A travers des services gérés par une IHM.

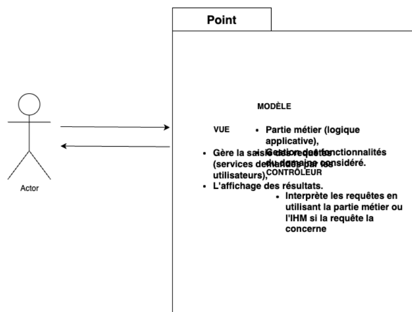
Une mauvaise conception



Principe SOLID Single responsability

- La classe `Point` a trop de responsabilités.
  - gestion de l'interface utilisateur,
  - controle de la validité des requêtes utilisateur,
  - gestion de l'entité point.

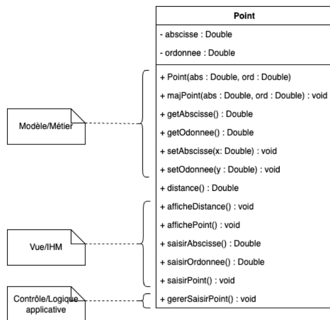
Une mauvaise conception



Principe SOLID Open/Closed

- La classe `Point` manque de souplesse.
  - *i.e.* si on décide de changer d'IHM, il faut modifier la classe déjà testée (*i.e.* IHM texte, graphique, web, ...),
  - si on veut modifier ou adapter d'autre comportements ou responsabilité il là encore modifier la classe.

Une mauvaise conception



Principe SOLID Single responsability

- La classe `Point` porte toutes les responsabilités,
  - modèle métier,
  - gestion des affichages et des saisie (Vue/IHM),
  - contrôle de la logique applicative.

Une mauvaise conception

```
public class Point {
  private Double abs, ord ;
  public void setAbscisse(Double x) this.abs = x; }
  public void setOrdonnee(Double y) this.ord = y; }

  public void saisirPoint(){
    Double x = this.saisirAbscisse() ;
    Double y = this.saisirOrdonnee() ;
    \dots
    this.majPoint( x, y);
  }
}
```

Dilution des responsabilités dans la classe Point

- Les setters relèvent de la partie métier,
- saisirPoint() : à une requête (demande) de l'utilisateur,
- des méthodes de saisie interactives relevant de l'IHM (Vue) et gérant les les interactions avec un utilisateur humain (ici, la demande des cordonnées du Point) ,
- des instructions s'adressant à la partie métier (Modèle) pour affecter les coordonnées saisies au Point (sans interaction avec l'utilisateur).

Une mauvaise conception

```
public class Point {
  private Double abs, ord ;
  public void setAbscisse(Double x){ this.abs = x; }
  public void setOrdonnee(Double y){ this.ord = y; }

  public void saisirPoint(){
    Double x = this.saisirAbscisse() ;
    Double y = this.saisirOrdonnee() ;
    ...
    this.majPoint( x, y);
  }
}
```

Dilution des responsabilités dans la classe `Point`

- ...et la séquence d'instructions prenant en charge l'interprétation de la requête qui relève de la couche de contrôle !

Principes non respectés

Principes SOLID single responsability

La classe `Point` à trois responsabilités

- **Modèle** gérer un point,
- **Vue** gérer l'IHM relative à un point,
- **Contrôle** gérer l'interprétation des requêtes soumises par l'utilisateur.

Principes SOLID Ségrégation des interfaces

La classe `Point` offre une interface unique pour la Vue, le Modèle et la couche de contrôle applicative.

Principes non respectés

Principes SOLID Inversion des dépendances

- Tout client de la classe `Point` qui souhaite gérer un point sans se soucier de l'IHM ou du contrôle reste dépendant des méthodes relatives à ces parties qu'il n'utilisera pas, or
- **Un client ne doit jamais être obligé de dépendre d'une interface qu'il n'utilise pas**[1].
- Ici, la classe `Point` dépend évidemment de sa propre implémentation (d'où la tendance à la dilution des responsabilités) et pas de classes abstraites ou d'interfaces.

Principes SOLID Open/closed

La classe `Point` n'offre aucune souplesse,
(ajout d'une nouvelle IHM $\Rightarrow$ la modification de la classe).

1. interface peut se comprendre comme interface au sens java, ou comme la partie publique d'une classe.

Principes non respectés

Conséquences

- Manque de lisibilité :
  - le code la classe `Point` mélange des méthodes et instructions qui n'ont rien à faire ensemble,
  - elle devient trop importante.
- Manque d'indépendance :
  - les parties métier, contrôle et l'IHM risquent de dépendre structurellement les unes des autres,
  - on ne peut plus confier les parties métier, contrôle et IHM à des développeurs distincts.
- Maintenance/extensibilité difficiles :
  - modifier une des parties aura des conséquences sur les autres et sera difficile à organiser.

Préconisations

Séparer ce qui change du reste

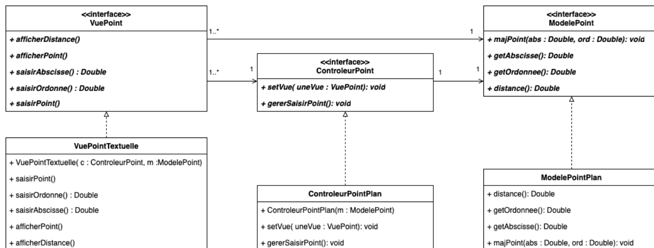
- Les éléments d'une classe susceptibles de changer doivent être placés dans d'autres classes liées à la première par composition :
  $\Rightarrow$ L'IHM[2] et le contrôle sont susceptibles de changer. Il faut donc les placer hors de la classe Point.

Dépendre d'interfaces non d'implémentations

- Tout client de la classe Point dépend des implémentations des méthodes de gestion du point, de l'IHM et du contrôle.
  $\Rightarrow$ Il faut créer 3 interfaces, tout client pouvant choisir les implémentations qui lui conviennent parmi les implémentations proposées (ou en concevoir de nouvelles)

2. il peut y avoir plusieurs vues du même modèle.

Une meilleur conception



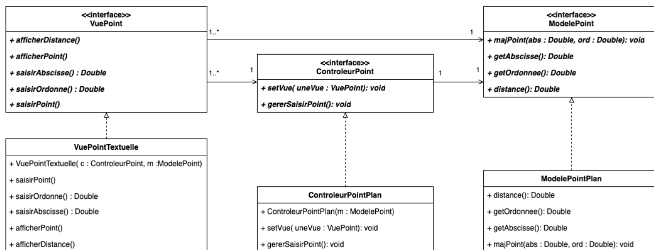
Principe SOLID Depedency inversion

Les classes concrètes dépendent d'interfaces et non de classes concrètes.

Principe SOLID Interface segregation

La vue, le modèle et la couche contrôle ont des interfaces disctinctes.

Une meilleur conception



Principe SOLID Single responsability

Les classes concrètes sont en charge chacune d'une seule responsabilité.

Principe SOLID Open/Closed

Une nouvelle vue peut être ajoutée ou se substituer facilement.

Le contrôleur

```
public class ControleurPointPlan implements ControleurPoint {

  private ModelePoint modele ;
  private VuePoint vue ;

  public ControleurPointPlan( ModelePoint unModelePoint ) {
    this.modele = unModelePoint ;
  }
  public void setVue( VuePoint uneVuePoint) {
    this.vue = uneVuePoint ;
  }
  public void gererSaisirPoint() {
    Double abs = this.vue.saisirAbscisse();
    Double ord = this.vue.saisirOrdonnee();

    this.modele.majPoint( abs , ord );
  }
}
```

Le chef d'orchestre du flux applicatif

- se charge de l'interaction avec l'utilisateur,
- connait et accède au modèle pour lui envoyer des messages de m.a.j.,
- connait la vue pour lui envoyer les messages de m.a.j.et solliciter l'utilisateur.

Le contrôleur

```
public class ControleurPointPlan implements ControleurPoint {

  private ModelePoint modele ;
  private VuePoint vue ;

   public ControleurPointPlan( ModelePoint unModelePoint ) {
     this.modele = unModelePoint ;
  }
   public void setVue( VuePoint uneVuePoint) {
     this.vue = uneVuePoint ;
  }
   public void gererSaisirPoint() {
    Double abs = this.vue.saisirAbscisse();
    Double ord = this.vue.saisirOrdonnee();

    this.modele.majPoint( abs, ord);
  }
}
```

Implémentation

- Association avec la ou les *vues* et le *modèle*.
- Interprétation de la requête saisirPoint() : de la *vue*.

Le contrôleur

```
public class ControleurPointPlan implements ControleurPoint {

  private ModelePoint modele ;
  private VuePoint vue ;

  public ControleurPointPlan( ModelePoint unModelePoint ) {
    this.modele = unModelePoint ;
  }
  public void setVue( VuePoint uneVuePoint) {
    this.vue = uneVuePoint ;
  }
  public void gererSaisirPoint() {
    Double abs = this.vue.saisirAbscisse();
    Double ord = this.vue.saisirOrdonnee();

    this.modele.majPoint( abs, ord);
  }
}
```

L'implémentation sollicite :

- **la vue** pour que l'utilisateur saisisse l'abscisse et l'ordonnée.
- **le modèle** pour qu'il mette à jour son état en enregistrant les nouvelles coordonnées.

La vue

```
public class VueTextuellePoint implements VuePoint {
  private ControleurPoint controleur ;
  private ModelePoint modele ;
  public VuePointTextuelle( ControleurPoint co, ModelePoint mo) {
    this.controleur = co; this.modele = mo;
  }

  public void activerVue(){ ... }

  public saisiePoint(){                this.controleur.gererSaisirPoint(); }
  public saisirAbscisse(){ ... }
  public saisirOrdonnee(){ ... }

  public void afficherPoint(){
    Sustem.out.println(" x : " +       this.modele.getAbscisse());
    Sustem.out.println(" y : " +       this.modele.getOrdonnee());
  }
  public vois afficherDistance(){
    System.out.println("dist : " +     this.modele.distance());
  }
}
```

Assure le rendu des informations

- connait le *modèle* pour l'interroger sur son (nouvel) état,
- connait le *controleur* pour lui déléguer l'analyse et la validation des requêtes de l'utilisateur.

La vue

```
public class VueTextuellePoint implements VuePoint {
  private ControleurPoint controleur ;
  private ModelePoint modele ;
  public VuePointTextuelle( ControleurPoint co, ModelePoint mo) {
    this.controleur = co; this.modele = mo;
  }

  public void activerVue(){ ... }

  public saisiePoint(){                    this.controleur.gererSaisirPoint(); }
  public saisirAbscisse(){ ... }
  public saisirOrdonnee(){ ... }

  public void afficherPoint(){
    Sustem.out.println(" x : " +           this.modele.getAbscisse());
    Sustem.out.println(" y : " +           this.modele.getOrdonnee());
  }
  public vois afficherDistance(){
    System.out.println("dist : " +         this.modele.distance());
  }
}
```

Implémentation

- gère l'association avec le *modèle* et le *contrôleur*,
- délégue au *controleur* analyses et validations des requêtes utilisateur,
- sollicite le *modèle* pour afficher les données de celui-ci.

Le modèle

```
public class ModelePointPlan implements ModelePoint {
  private Double abs ;
  private Double ord ;

  private ModelePointPlan( Double x, Double y){
    this.majPoint( x, y);
  }
  public Double getAbscisse(){       return this.abs;  }
  public Double getOrdonne(){        return this.ord ; }

  public void setAbscisse(Double x){ this.abs = x ;    }
  public void setOrdonne(Double y){  this.ord = y ;    }

  public void majPoint( Double x, Double y) {
    this.setAbscisse(x);
    this.setOrdonnee(y);
  }
  public Double distance(){ ... }
}
```

Le modèle est indépendant

- Le *modèle* ne connaît ni le *contrôleur* ni la *vue*,
- C'est à la charge des autres composants logiciels de lui envoyer des messages.

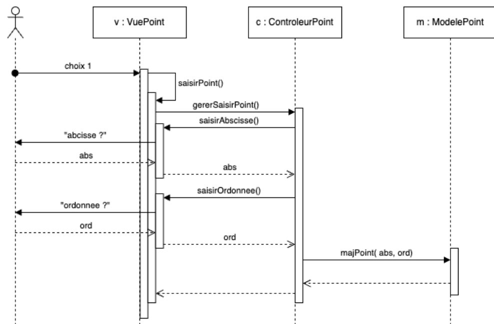
Le Client

```
public class Client {
  public static void main( String [] args ) {

    ModelePoint m = new ModelePointPlan() ;

    ControleurPoint c = new ControleurPointPlan( m );

    VuePoint v = new VuePointTextuelle( c, m);

    c.setVue(v) ;

    v.activerVue();
  }
}
```

Responsabilités

- Instancier le *modèle* le *contrôleur* et la *vue*,
- Assurer les *bonnes* liaisons entre ces composants pour qu'ils puissent communiquer entre eux.

Le flux applicatif



Pas encore dynamique

- Le *modèle* a changé (nouvelles coordonnées du point) mais la *vue* affiche encore les anciennes coordonnées.
- Il n'y a pas de mise à jour automatique de la *vue*.

Le meta-patron MVC (Modèle-Vue-Contrôleur)

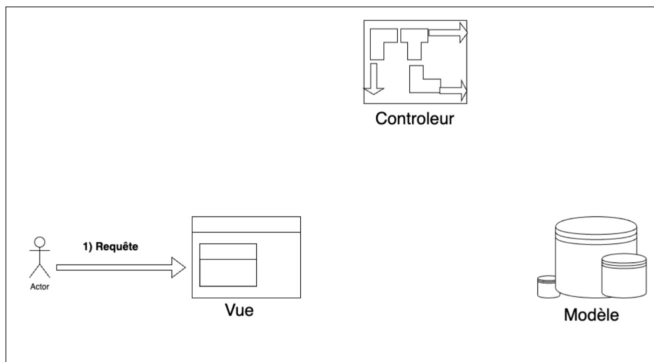
Objectifs

Le patron MVC permet :

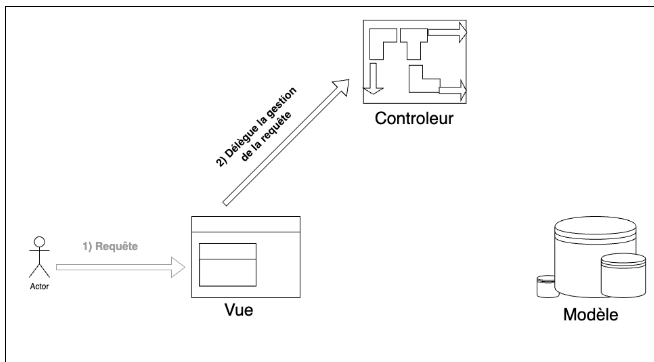
- la mise à jour automatique des vues[3] :
  ⇒ **patron de conception observateur**,
- la séparation des implémentations des vues et du contrôleur[4] :
  ⇒ **patron de conception stratégie**,
- la vue peut gérer des composants imbriqués :
  ⇒ **patron de conception composite**.

3. Un modèle peut avoir plusieurs vues différentes par exemple une vue textuelle et une vue graphique.
4. on peut changer de contrôleur.

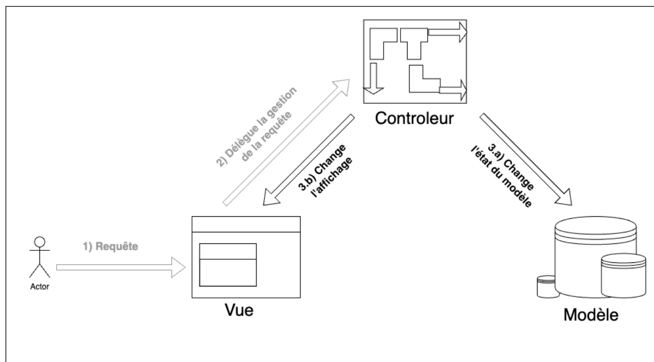
Flux applicatif dans MVC



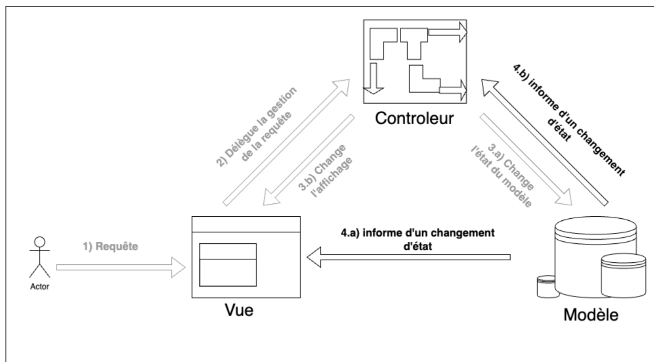
Flux applicatif dans MVC



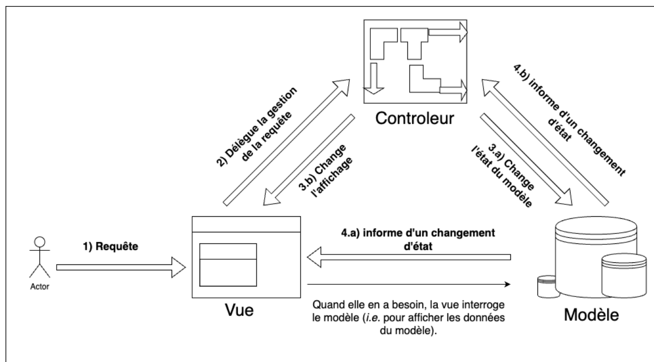
Flux applicatif dans MVC



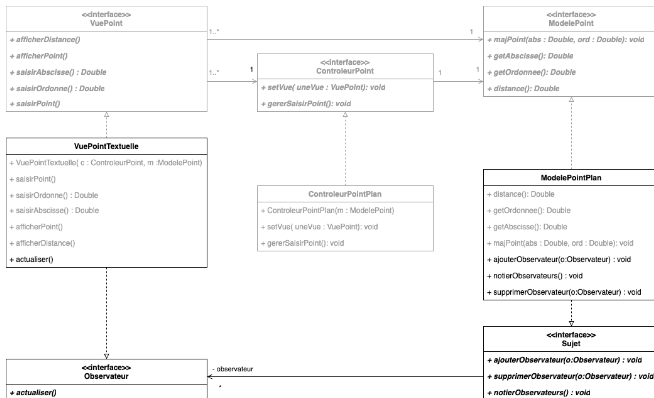
Flux applicatif dans MVC



Flux applicatif dans MVC



Mise à jour des vues



Patron Observateur

Le *modèle* connaît ses `Observateurs` pour pouvoir leur notifier tout changement dans son état.

Mise à jour des vues

```
public class ModelePointPlan implements ModelePoint, Sujet {
   private Double abs;
   private Double ord;
  private ArrayList<Observateur> observateurs ;

   public ModelePointPlan( Double x, Double y) {
     this.abs = x; this.ord = y;
     this.observateur = new ArrayList<Observateur>();
  }
  public void ajouterObservateur(Observateur o){ this.observateurs.add(o); }
  public void supprimerObservateur(Observateur o){ this.observateurs.remove(o); }

   public void majPoint(Double x, Double y){
     this.abs = x; this.ord = y;
    this.notifierObservateurs();
  }
  public void notifierObservateurs(){
    tfor( Observateur o : this.observateurs )
      to.actualiser();
  }
}
```

Patron Observateur

- Implémentation du patron,
- Le *modèle* notifie les vues quand le *modèle* change,
- Les vues sont actualisées.

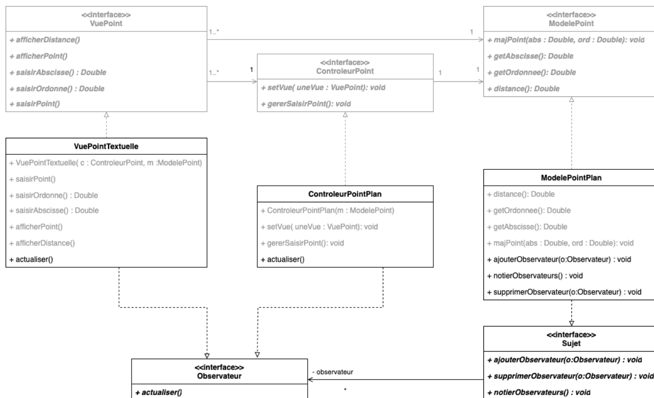
Mise à jour des vues

```
public class VueTextuellePoint implements VuePoint, Observateur {
  private ControleurPoint controleur ;
  private ModelePoint modele ;
  public VuePointTextuelle( ControleurPoint co, ModelePoint mo) {
    this.controleur = co; this.modele = mo;
    this.modele.ajouterObservateur(this);
  }
  public void actualiser(){
    this.afficherPoint();
    this.afficherDistance();
  }
  public void afficherPoint(){
    Sustem.out.println(" x : " +        this.modele.getAbscisse());
    Sustem.out.println(" y : " +        this.modele.getOrdonnee());
  }
  public vois afficherDistance(){
    System.out.println("dist : " +      this.modele.distance());
  }
}
```

Patron Observateur

- La *vue* est un Observateur,
- La *vue* s'enregistre comme Observateur du *modèle*,
- L'actualisation provoque un rafraichissement des affichages,
- La*vue* ne fait aucun traitement, seulement des rendus.

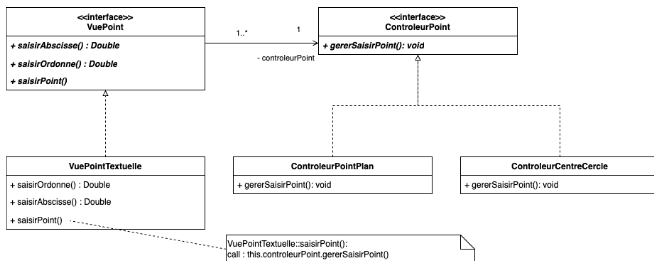
Mise à jour du contrôleur



Patron Observateur

Lorsque le *contrôleur* a besoin de savoir que le *modèle* a changé, on adopte le même schéma qu'entre le *modèle* et les *vues*.

Choix du contrôleur



Patron Stratégie

- Le *contrôleur* est la stratégie associée aux *vues*,
- La *vue* délègue à la stratégie à laquelle elle est associé de gérer son interaction avec le modèle,
- Plusieurs stratégies de contrôle peuvent être mises en place et inter-changées dynamiquement sans affecter le fonctionnement du *modèle* ou de la *vue*.

MVC : une architecture flexible

```
public class Client {
  public static void main( String [] args){

    ModelePoint pt = new ModelePointPlan( 1., 3.7) ;
    ControleurPoint ctrlPt = new ControleurPointPlan( pt);
    VuePoint vuePoint = new VuePointTextuelle( ctrlPt , pt) ;

    ctrPt.setVue(vuePoint);
    vuePoint.activerVue();

    ControleurPoint ctrCenter = new ControleurCentreCercle(  pt);
    VuePoint vueCercle = new VuePointTextuelle( ctrCenter , pt)

    ctrCenter.setVue(vueCercle);
    vueCercle.activerVue();
  }
}
```

Principe Open/Closed

- Plusieurs stratégies de contrôle (et plusieurs vues) peuvent être associée à un point,
- Les différentes stratégies sont interchangeables sans impact sur le reste du code.

MVC : une architecture flexible

Principe Interface segregation

- Les différentes couches de l'architecture sont découplée et définies de façon abstraites dans des interfaces différentes.

Principe Single responsability

- Chaque couche de l'architecture est en charge d'un aspect de l'application :
  - Modèle : représentation des données métiers et de implémentation des services associés à ces données,
  - Vue : interaction avec l'utilisateur (capture des requêtes),
  - Contrôle : gestion du flux applicatif, contrôle des saisies, mise à jour du *modèle*.

Principe Dependency inversion

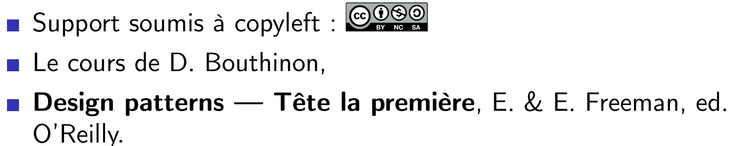
- Le code dépend d'abstractions et non d'implémentations,

MVC : une architecture flexible

Autres principes mis en œuvre

- Faible couplage,
- Gestion dynamique,
- Préfère la composition à l'héritage,
- Préserve la factorisation du code,
- ...

Credits

- Support soumis à copyleft : 

- Le cours de D. Bouthinon,
- **Design patterns — Tête la première**, E. & E. Freeman, ed. O'Reilly.