Cours M3105 : Conception et programmation objet avancées

l'architecture Modèle-Vue-Controleur

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

IUT Villetaneuse

2019-2020

Plan

Motivations

Quelques exemples de mauvaises conceptions

Les principes que la classe `Point` ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

Principe de conception rencontré

Contents

Motivations

Quelques exemples de mauvaises conceptions

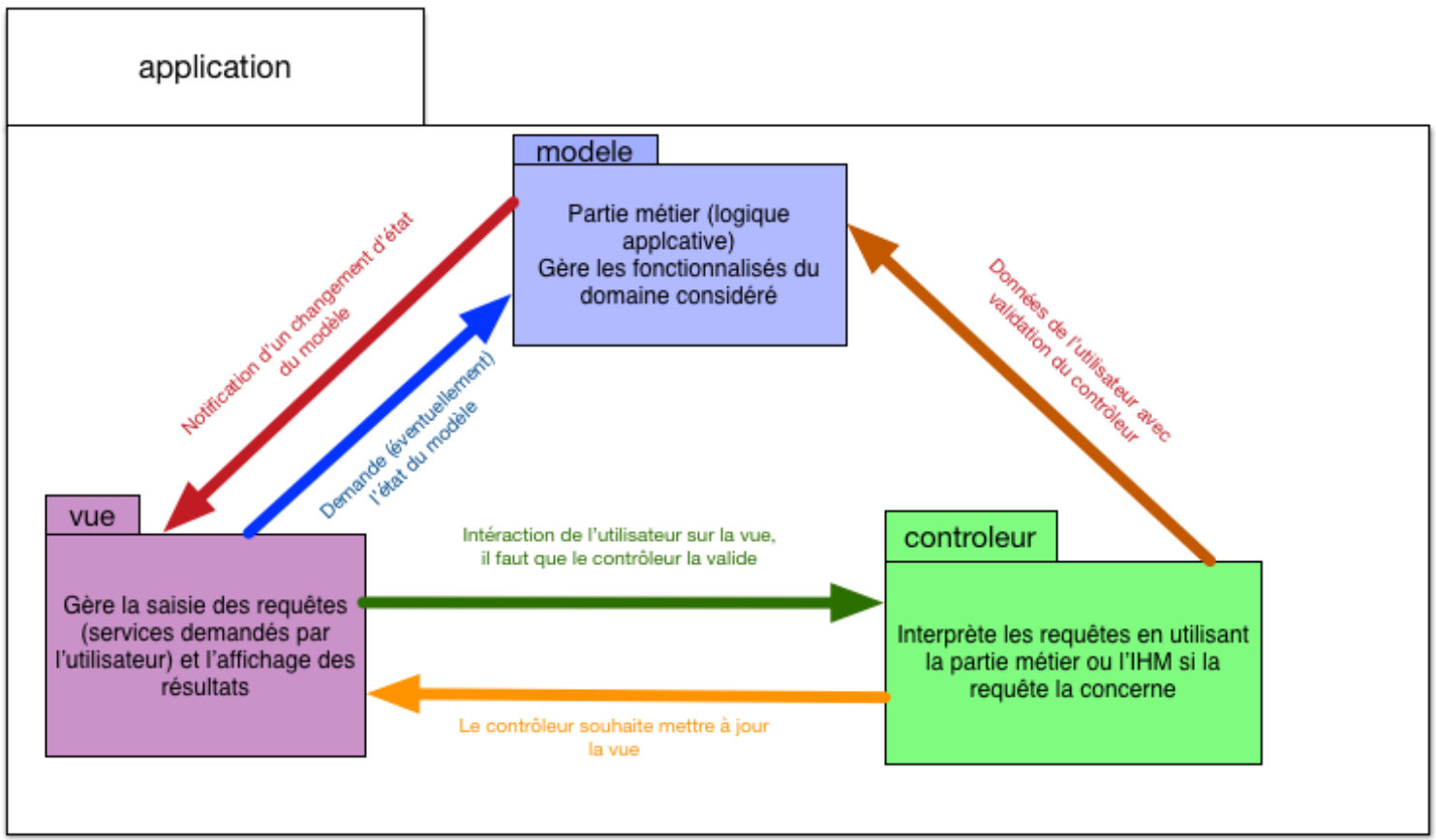
Les principes que la classe Point ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

Principe de conception rencontré

Principe général du design pattern MVC



D'un point de vue logiciel

Le patron de conception **M**odèle **V**ue **C**ontrôleur sépare le code d'une application en 3 composants logiciels :

- le **modèle** : s'occupe du traitement des données, il correspond aux **classes métiers**,
- la (les) **vue(s)** : qui gère(nt) le rendu,
- le **contrôleur** : véritable chef d'orchestre de l'application ; il interagit avec le modèle et les vues.

L'intérêt d'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.

Par exemple, 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.

Contents

Motivations

Quelques exemples de mauvaises conceptions

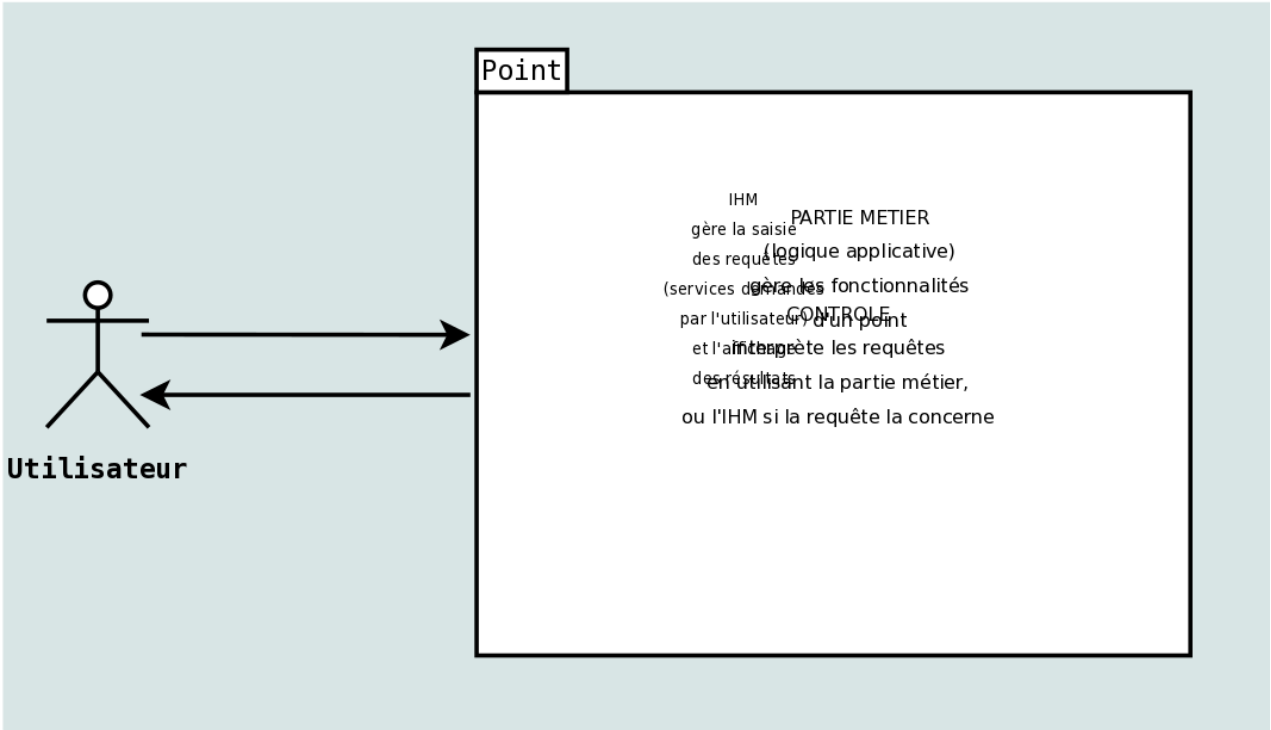
Les principes que la classe Point ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

Principe de conception rencontré

Une mauvaise conception

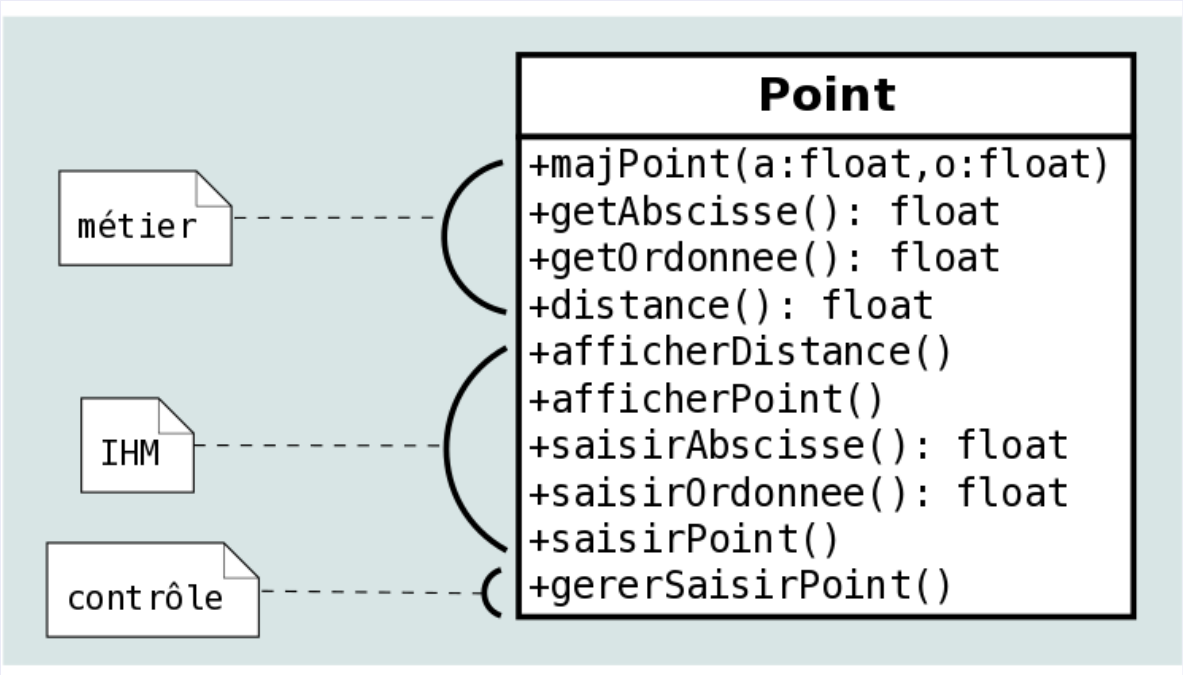


Qu'est ce qui ne va pas dans cette conception ?

- la classe `Point` possède plusieurs responsabilités,
- si on modifie l'IHM, on modifie l'application dans sa globalité. Idem pour les autres responsabilités de la classe `Point`.

Une mauvaise conception

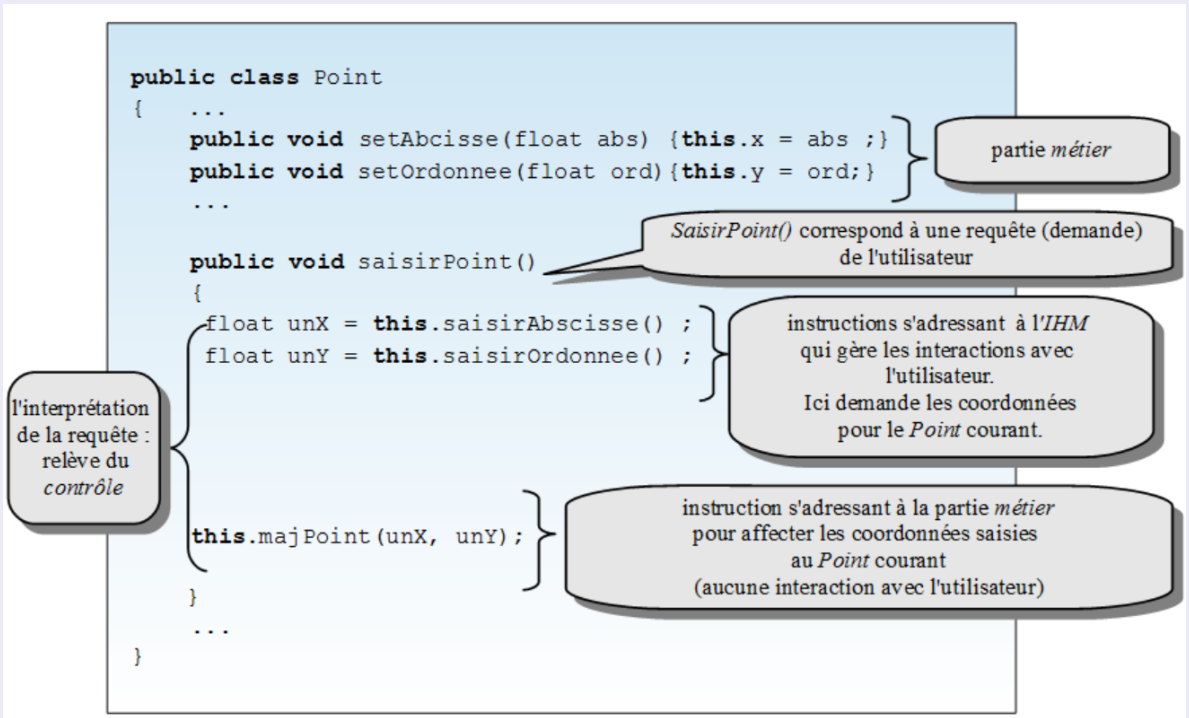
La classe Point possède toutes les responsabilités



- la classe Point englobe toutes les responsabilités : **classes métiers**, gestion du **rendu** IHM), **contrôle** de l'interaction avec l'utilisateur,
- la modification d'une responsabilité entraînera modification de toute l'application.

Une mauvaise conception

Phénomène de dilution des responsabilités dans la classe Point



- les responsabilités sont mélangées dans les différents méthodes de la classe Point,
- la maintenance de l'application devient complexe !

Une mauvaise conception

Exemple de classe cliente de Point

```
public class Client
{
    public static void main(String[] args)
    {
        Point p = new Point() ;
        p.activerIHMPoint() ;
    }
}
```

```
abscisse  = 0.0
ordonnée  = 0.0
distance  = 0.0

********** MENU POINT **********

1 - entrer/modifier le point
4 - sortir du programme

entrer votre choix (puis Entrée)
```

menu affiché quand on active l'*IHM* : attend le choix de l'utilisateur, les choix correspondent à des requêtes prédéfinies.

Le classe `Client` (et l'utilisateur humain) n'interagit qu'avec l'IHM.

Une (à peine) meilleure conception

```
public class Point
{
    ...
    public void majPoint(float a, float o)
                        {this.x = a ; this.y = o ;}
    public void distance() {...}

    public void afficherPoint() {...}
    public void afficherDistance() {...}
    public void activerIHMPoint() {...}
    public float saisirAbscisse() {...}
    public float saisirOrdonnee() {...}
    public void saisirPoint(){...}

    public void gererSaisirPoint() {...} ;
}
```

métier : gestion d'un point

IHM : ne s'occupe que des interactions avec l'utilisateur

contrôle : interprète les requêtes (ici une seule requête)

- on "sépare bien" les parties métier, contrôle et IHM dans la classe `Point`,
- mais, la classe `Point` possède plusieurs responsabilités qui induisent les inconvénients déjà évoqués...

Contents

Motivations

Quelques exemples de mauvaises conceptions

Les principes que la classe `Point` ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

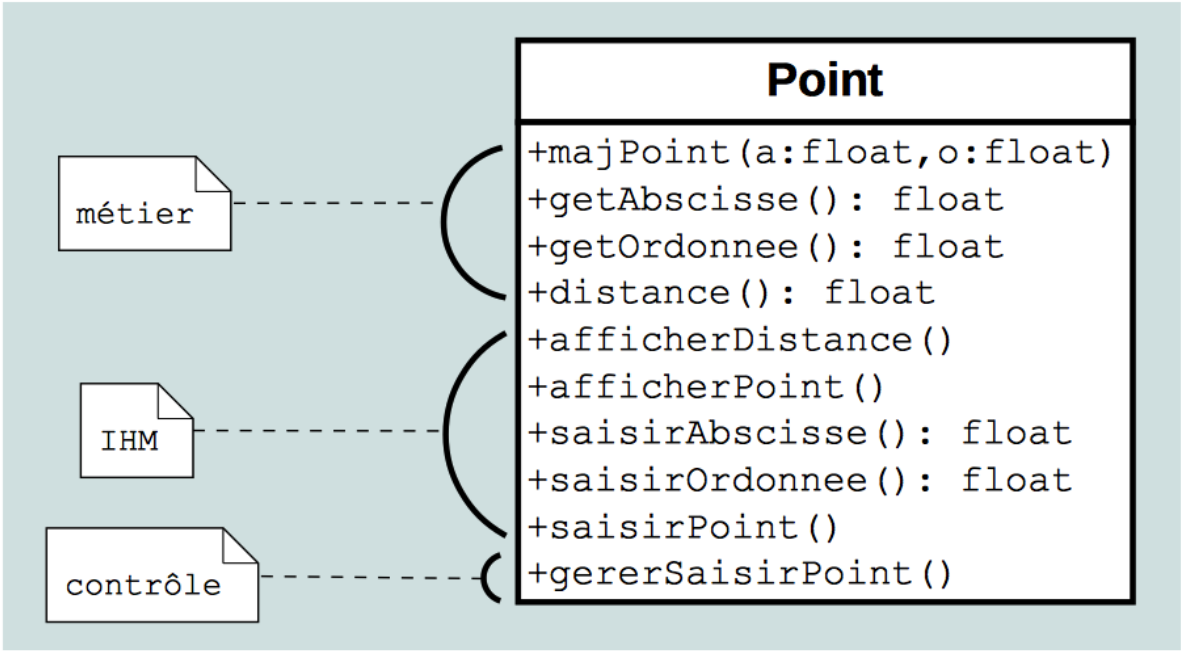
Principe de conception rencontré

La classe `Point` enfreint plusieurs règles des principes SOLID

La classe `Point` ne sépare pas la **vue** (IHM), la **partie métier** et la partie **contrôle** ; elle contrevient à plusieurs principes de conception objet :

- séparation des interfaces (*Interface segregation principle*) (SOLID),
- responsabilité unique (*Single responsability principle*) (SOLID),
- séparer ce qui change du reste,
- dépendre d'interfaces non d'implémentations.

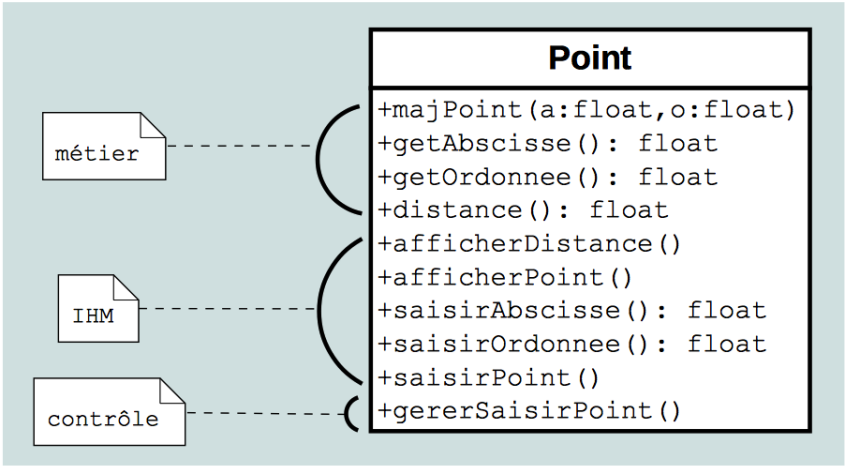
Le principe de responsabilité unique est enfreint par Point



La classe Point a trois responsabilités :

- gérer un point qui correspond à la **partie métier**,
- gérer l'IHM relative à un point qui correspond à la **partie vue**,
- gérer l'interprétation des requêtes soumises par l'utilisateur qui correspond à la **partie contrôle**.

Principe de séparation des interfaces est enfreint par Point



- Un client ne doit jamais être obligé de dépendre d'une interface [1] qu'il n'utilise pas :
  - 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.

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

Conséquences du non respect des ces principes

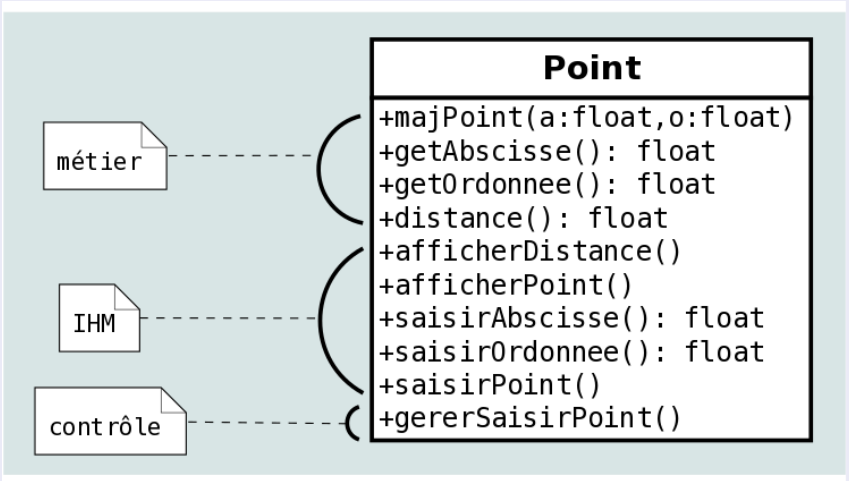
- **Manque de lisibilité** :
  - le code la classe `Point` mélange des méthodes et instructions qui n'ont rien à faire ensemble. Elle devient trop importante.

- **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

Que faut-il faire ?

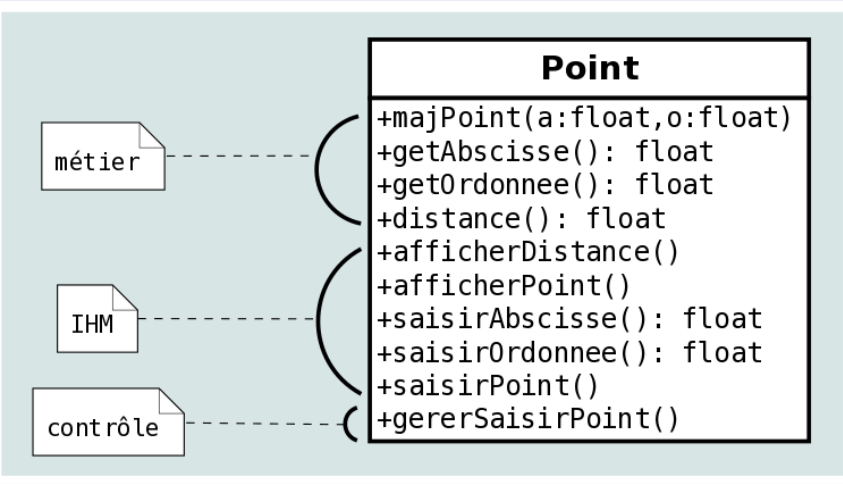
Séparer ce qui change du reste



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

Que faut-il faire ?

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.
  - Il aurait fallu créé 3 **interfaces**, tout client pouvant choisir les implémentations qui lui conviennent parmi les implémentations proposées (ou en concevoir de nouvelles)

Contents

Motivations

Quelques exemples de mauvaises conceptions

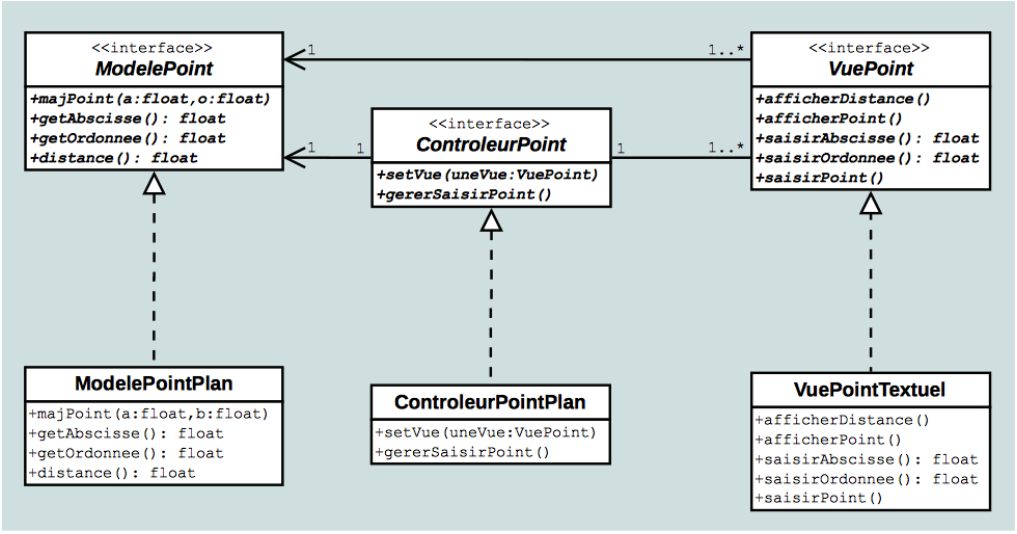
Les principes que la classe Point ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

Principe de conception rencontré

Vers une meilleur conception



- **Dépendre d'abstraction et non d'implémentation** : Créer des interfaces !
- **Séparer les responsabilités** : On isole les parties métier, contrôle et l'IHM dans des interfaces (et classes) distinctes

Par convention la partie métier se nomme **modèle**, la partie contrôle se nomme **contrôleur**, et l'IHM se nomme **vue** [2]

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

Vers une meilleur conception : 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()
    {
        float abs = this.vue.saisirAbscisse() ;
        float ord = this.vue.saisirOrdonnee() ;

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

associations avec la *vue* (ou les *vues*) et le *modèle*

interprétation de la requête *saisirPoint()* de la *vue*

s'adresse à la *vue* pour que l'utilisateur entre l'abscisse et l'ordonnée

s'adresse au *modèle* pour enregistrer les nouvelles coordonnées

- Le contrôleur joue le rôle de chef d'orchestre ; il se charge lui-même de l'interaction avec l'utilisateur mais il connaît :
  - le modèle pour lui envoyer des messages de mise à jour,
  - la vue pour lui envoyer les messages de mise à jour.

Vers une meilleur conception : La vue

```
public class VuePointTextuel implements VuePoint
{
    private ControleurPoint controleur ;
    private ModelePoint modele ;

    public VuePointTextuel(ControleurPoint co,
                           ModelePoint mo)
    { this.controleur = co ; this.modele = mo ; }


    public void activerVue(){...}

    public void saisirPoint() {this.controleur.gererSaisirPoint();}

    public float saisirAbscisse() {...}

    public float saisirOrdonnee() {...}

    public void afficherPoint()
    { System.out.println("abscisse  = " + this.modele.getAbscisse()) ;
      System.out.println("ordonnee  = " + this.modele.getOrdonnee()) ;}

    public void afficherDistance()
    { System.out.println("distance  = " + this.modele.distance()) ; }
}
```

association avec le *contrôleur*

association avec le *modèle* pour pouvoir afficher les données du modèle

la *vue* délègue au *contrôleur* l'interprétation d'une requête

la *vue* interroge le *modèle* pour afficher certaines de ses données

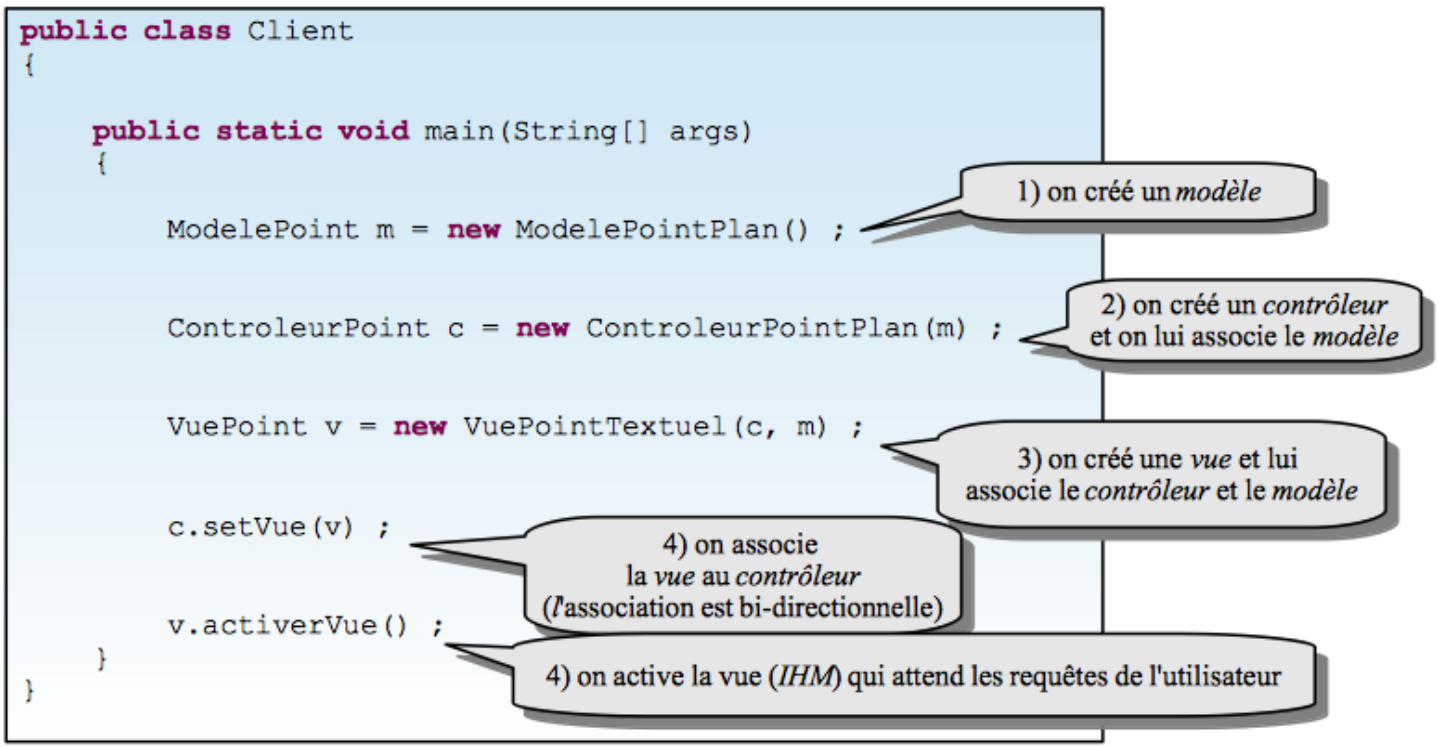
- La vue permet d'assurer le rendu des informations, elle connaît :
  - le modèle ; pour l'interroger sur son (nouvel) état,
  - le contrôleur ; pour lui déléguer l'analyse et la validation des requêtes de l'utilisateur.

Vers une meilleur conception : Le modèle

```
public class ModelePointPlan implements ModelePoint
{

    private float x ;
    private float y ;

    public ModelePointPlan()
    { this.x = 0 ; this.y = 0 ;}

    public float getAbscisse()
    {return this.x ;}

    public float getOrdonnee()
    {return this.y ;}

    public void majPoint(float a, float o)
    {this.x = a ;
     this.y = o ;}

    public void setOrdonnee(float ord)
    {this.y = ord ;}

    public float distance()
    {...}
}
```

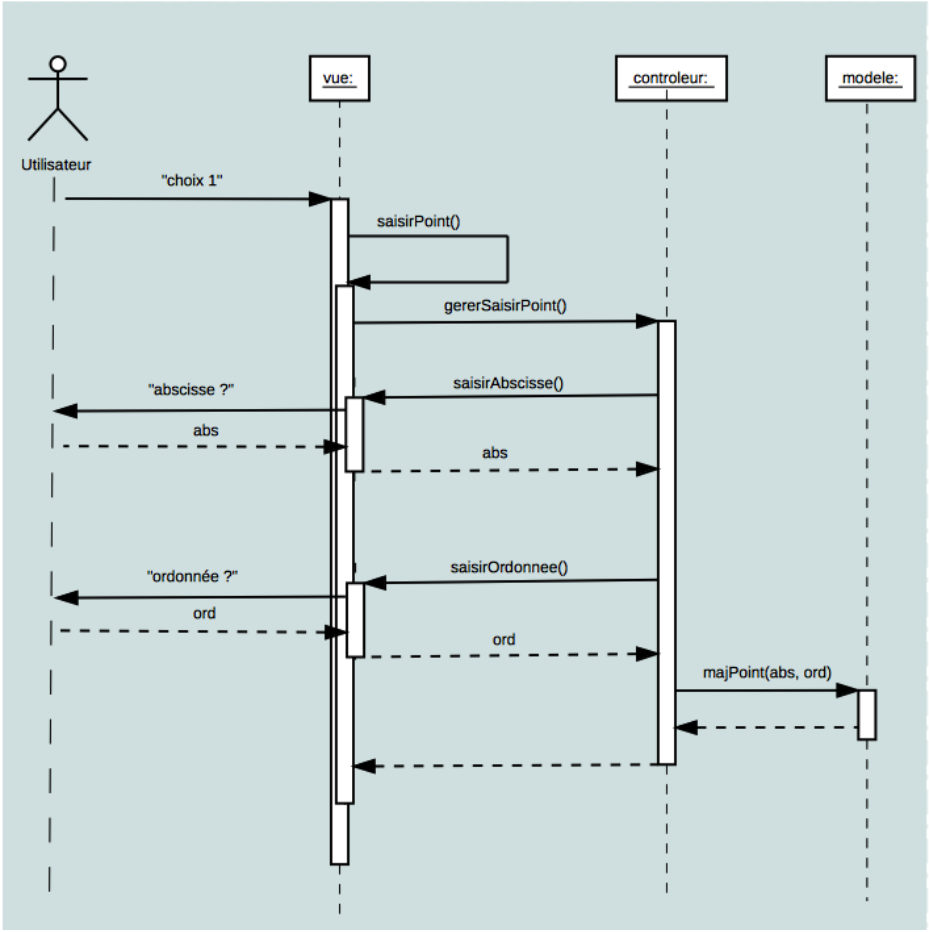
- Le modèle ne connaît ni le contrôleur ni la vue. C'est à la charge des autres composants logiciels de lui envoyer des messages.

Vers une meilleur conception : Le client



- le client a la responsabilité de l'instanciation du modèle, du contrôleur et de la vue et d'assurer les "bonnes" liaisons pour que ces différents composants puissent communiquer entres-eux.

Vers une meilleur conception : Dynamique d'une requête utilisateur



- 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.**

Contents

Motivations

Quelques exemples de mauvaises conceptions

Les principes que la classe Point ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

Principe de conception rencontré

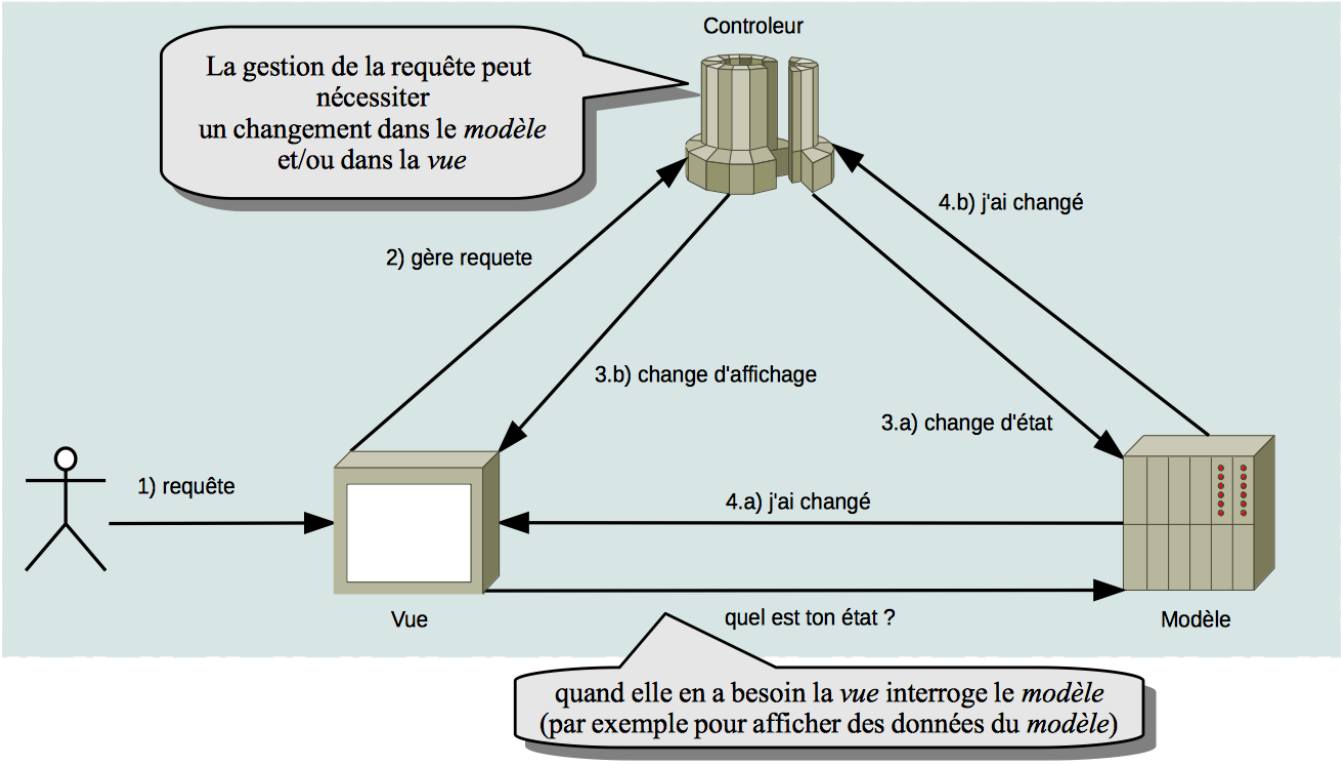
Le design pattern Modèle Vue Contrôleur

Le design pattern MVC permet de pallier aux inconvénients vus précédemment, il permet :

- la mise à jour **automatique** des vues[3] : **design pattern observateur**,
- la séparation des vues et du contrôleur[4] : **design pattern stratégie**,
- La vue peut gérer des composants imbriqués : **design pattern composite**.

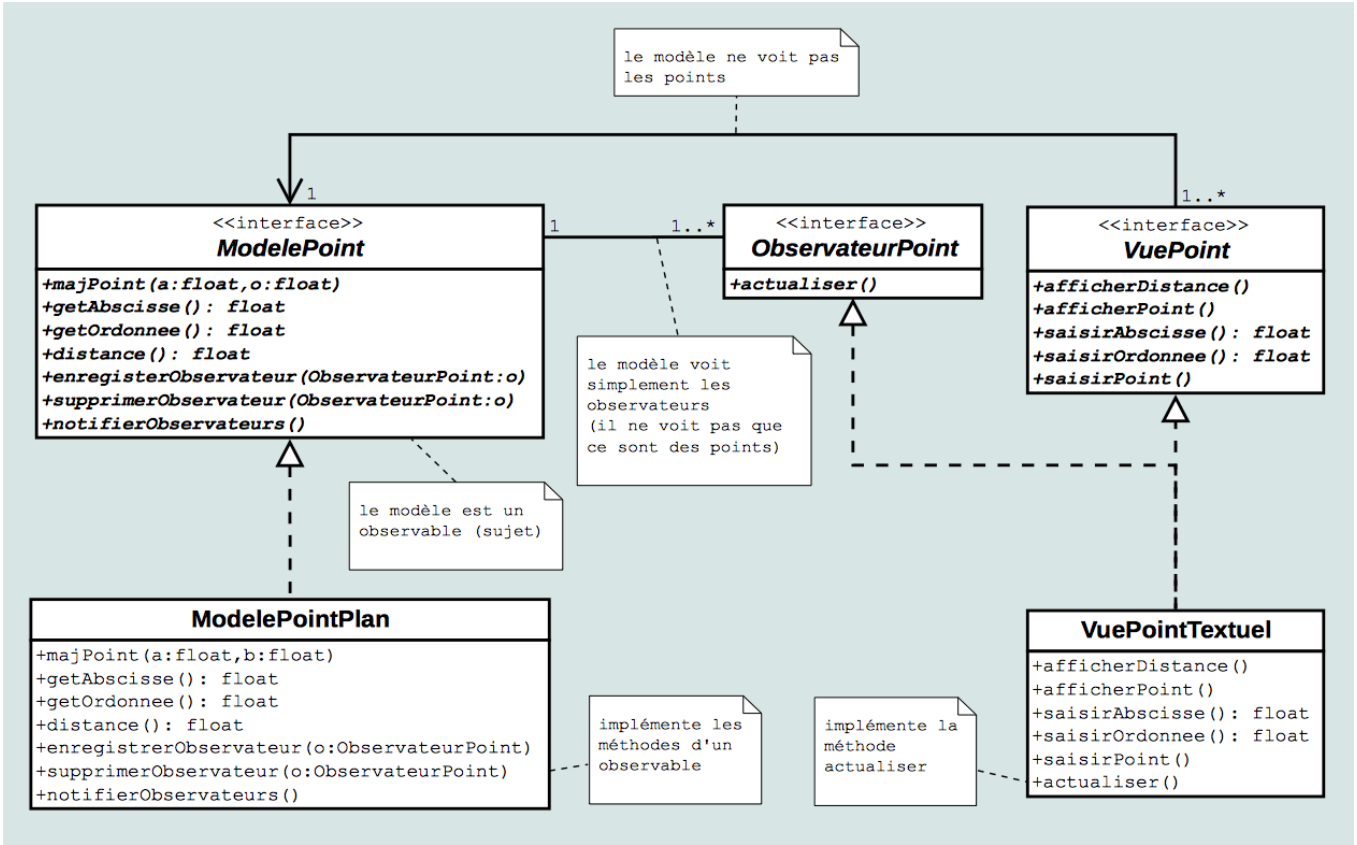
3. **Un modèle peut avoir plusieurs vues différentes par exemple une vue textuelle et une vue graphique**
4. **on peut changer de contrôleur**

Dynamique générale de l'architecture MVC



- Les vues sont automatiquement averties des changements du modèle (si nécessaire le contrôleur l'est aussi).

Mise à jour automatique des vues (design pattern Observateur)



- Le modèle connaît ses observateurs pour pouvoir **leur notifier** tout changement dans son état.

Le modèle

```
public class ModelePointPlan implements ModelePoint
{
    private float x, y;
    private ArrayList<ObservateurPoint> observateurs ;

    public ModelePointPlan()
    {
        this.x = 0 ; this.y = 0 ;
        this.observateurs = new ArrayList<ObservateurPoint>() ;
    }

    public void enregistrerObservateur(ObservateurPoint o)
    {this.observateurs.add(o);}

    public void supprimerObservateur(ObservateurPoint o)
    {this.observateurs.remove(o);}

    public void majPoint(float abs, float ord)
    { this.x = abs ; this.y = ord ;
      this.notifierObservateurs() ; }

    public void notifierObservateurs()
    {
        for (int i = 0 ; i < this.observateurs.size() ; i++)
        {this.observateurs.get(i).actualiser() ;} }
    }
    ...
}
```

le *modèle* notifie les *vues* quand le point change

les vues sont actualisées

- le modèle connaît ses observateurs : `ArrayList<ObservateurPoint>` (par exemple) qui contiendra les références de tous les observateurs. La méthode `enregistrerObservateur()` permet l'enregistrement d'un observateur, et la méthode `enregistrerObservateur()` sa suppression,
- le modèle notifie tout changement à ses vues : `notifierObservateurs()` envoie une notification à tous les observateurs.

La vue

```
public class VuePointTextuel implements VuePoint, ObservateurPoint
{
    private ControleurPoint controleur ;
    private ModelePoint modele ;

    public VuePointTextuel(ControleurPoint unControleur,
                           ModelePoint unModelePoint)
    {
        this.controleur = unControleur ;
        this.modele     = unModelePoint ;
        this.modele.enregistrerObservateur(this) ;
    }

    public void actualiser()
    {
        this.afficherPoint() ;
        this.afficherDistance() ;
    }

    public void afficherPoint()
    {
        System.out.println("abscisse  = " + this.modele.getAbscisse()) ;
        System.out.println("ordonnee  = " + this.modele.getOrdonnee()) ;
    }

    public void afficherDistance()
    { System.out.println("distance  = " + this.modele.distance());}
}
```

la *vue* est (aussi) un *ObservateurPoint*

la *vue* s'enregistre comme *observateur* du *modèle*

Le *modèle* demande à la vue de s'actualiser : elle appelle ses fonctions d'affichage...

chaque fonction d'affichage interroge le modèle pour lui demander son état actuel

- la vue ne fait **aucun traitement**, elle se charge seulement du rendu.

La 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()
    {
        float abs = this.vue.saisirAbscisse() ;
        float ord = this.vue.saisirOrdonnee() ;

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

associations avec la *vue* (ou les *vues*) et le *modèle*

interprétation de la requête *saisirPoint()* de la *vue*

s'adresse à la *vue* pour que l'utilisateur entre l'abscisse et l'ordonnée

s'adresse au *modèle* pour enregistrer les nouvelles coordonnées

- Le contrôleur ne change pas.

Le client

```
public class Client
{
     public static void main(String[] args)
     {
          ModelePoint m = new ModelePointPlan() ;
          ControleurPoint c = new ControleurPointPlan(m) ;
          VuePoint v = new VuePointTextuel(c, m) ;

          c.setVue(v) ;
          v.activerVue() ;
     }
}
```

```
abscisse  = 0.0
ordonnée  = 0.0
distance  = 0.0


*************** MENU POINT ********************

1 - entrer/modifier le point
4 - sortir du programme


entrer votre choix (puis Entree)
1
abcisse ?
5
ordonnee ?
7

abscisse  = 5.0
ordonnée  = 7.0
distance  = 8.6
```

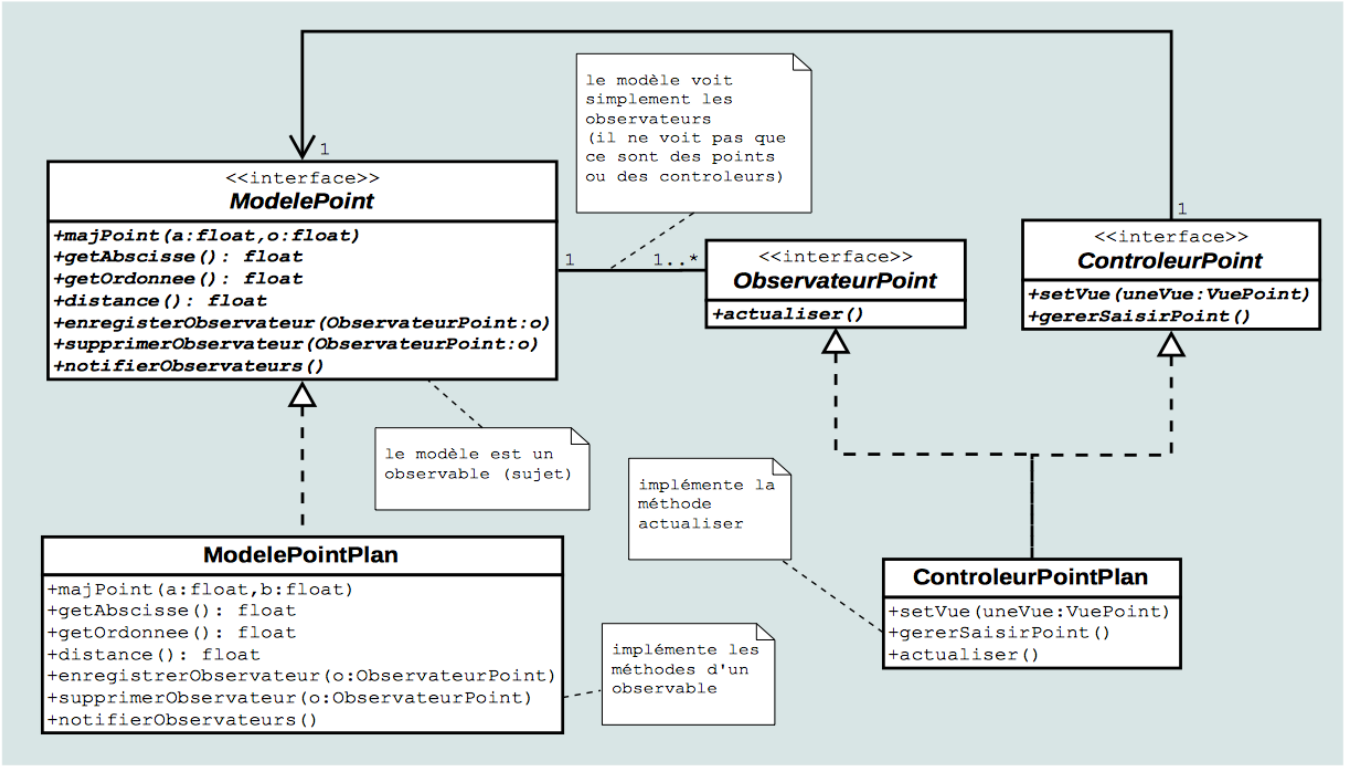
affichage initiale de la *vue*

l'utilisateur entre de nouvelles coordonnées

la *vue* est automatiquement mise à jour

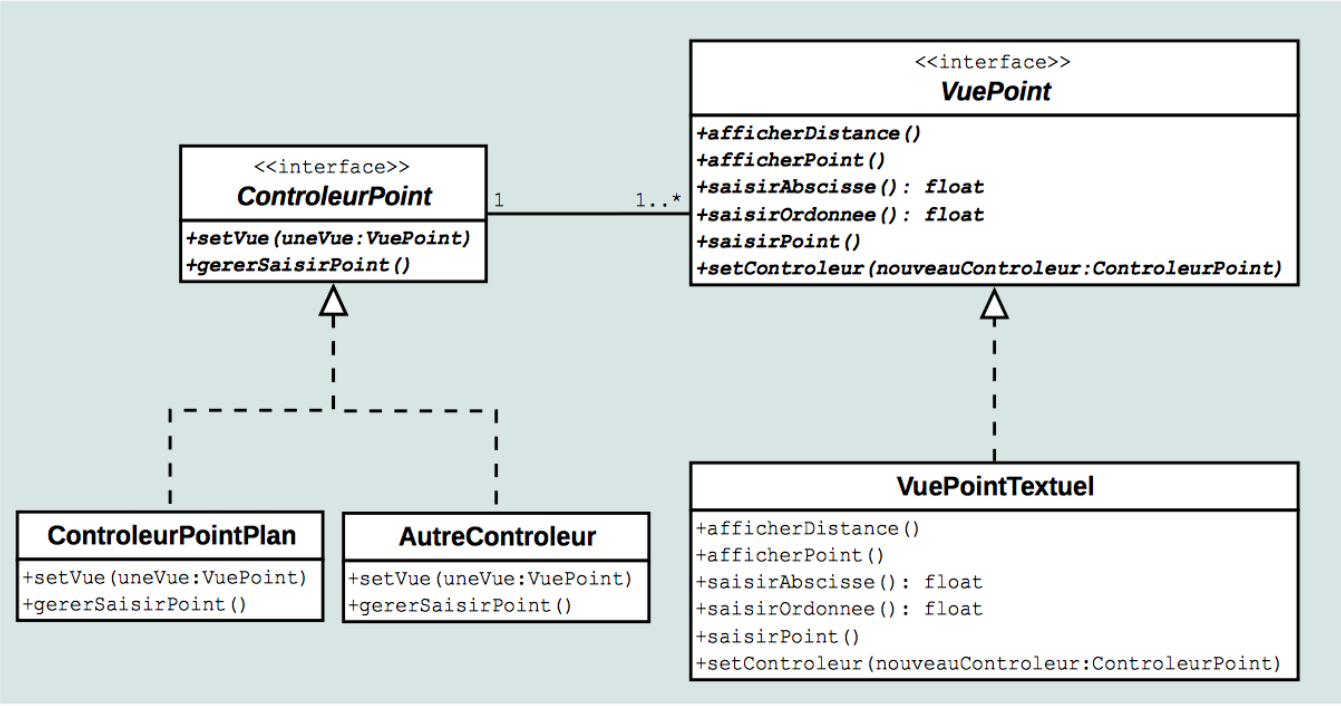
- Le client ne change pas.

Mise à jour automatique du contrôleur (design pattern 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*.

Le contrôleur est la stratégie associée aux vues (design pattern Strategie)



- Le contrôleur sait comment gérer les actions demandées par l'utilisateur [5].

5. **On peut changer de contrôleur sans affecter ni les vues ni le modèle**

L'architecture MVC est flexible

```
public class Client
{
    public static void main(String[] args)
    {
        ModelePoint     m  = new ModelePointPlan() ;
        ControleurPoint c1 = new ControleurPointPlan(m) ;
        VuePoint        v1 = new VuePointTextuel(c1, m) ;

        c1.setVue(v1) ;
        v1.activerVue() ;

        Controleur c2 = new AutreControleur(m) ;
        VuePoint   v2 = new Autrevue(c2, m) ;

        c2.setVue(v2) ;
        v2.activerVue() ;
    }
}
```

l'utilisateur utilise la nouvelle *vue* pour entrer ses *requêtes* qui seront interprétées par le nouveau *contrôleur*

- L'architecture MVC permet de changer de contrôleur et/ou de(s) vue(s) **sans aucune modification**.

Contents

Motivations

Quelques exemples de mauvaises conceptions

Les principes que la classe Point ne respecte pas

Vers une meilleur conception

Le design pattern Modèle Vue Contrôleur

Principe de conception rencontré

Principe de conception rencontré

- les classes doivent être le plus faiblement couplées :
  - le design pattern Stratégie permet de découpler la vue du modèle,
  - le contrôleur a la responsabilité d'interagir avec le modèle.