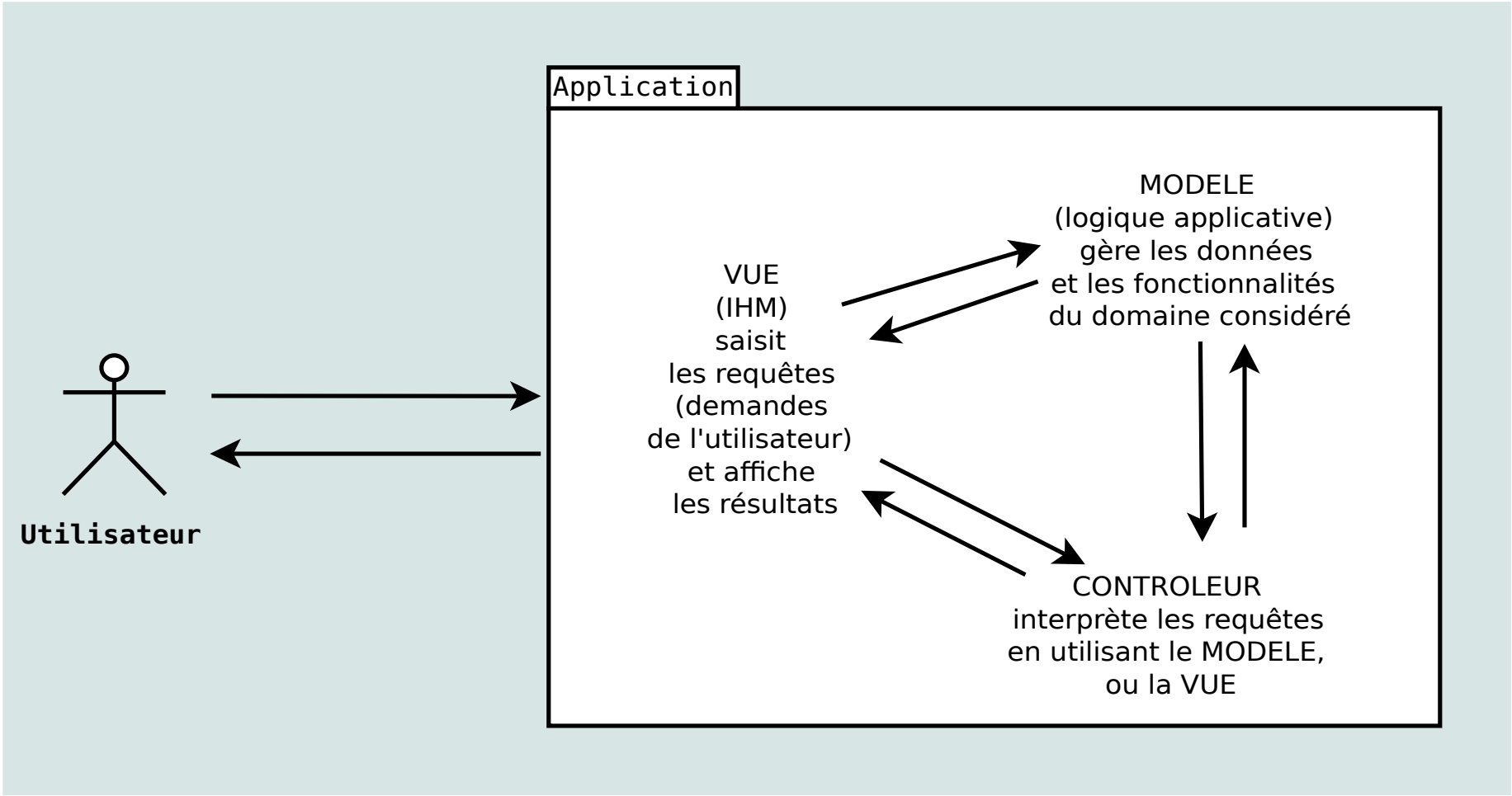
Conception et programmation orientées objet avancées

l'architecture modèle-vue-contrôleur :
une composition de 3 design patterns
(composite + observateur + stratégie)

(ce cours présente l'architecture MVC depuis le point de vue des designs patterns et des principes de conception objet, il fait volontairement peu référence à la programmation d'interfaces graphiques déjà présentée dans le cours IHM)

Motivation

Créer des applications avec interface homme-machine (IHM) facilement maintenables et extensibles



Exemple :
créer un application gérant un point du plan

L'utilisateur peut :

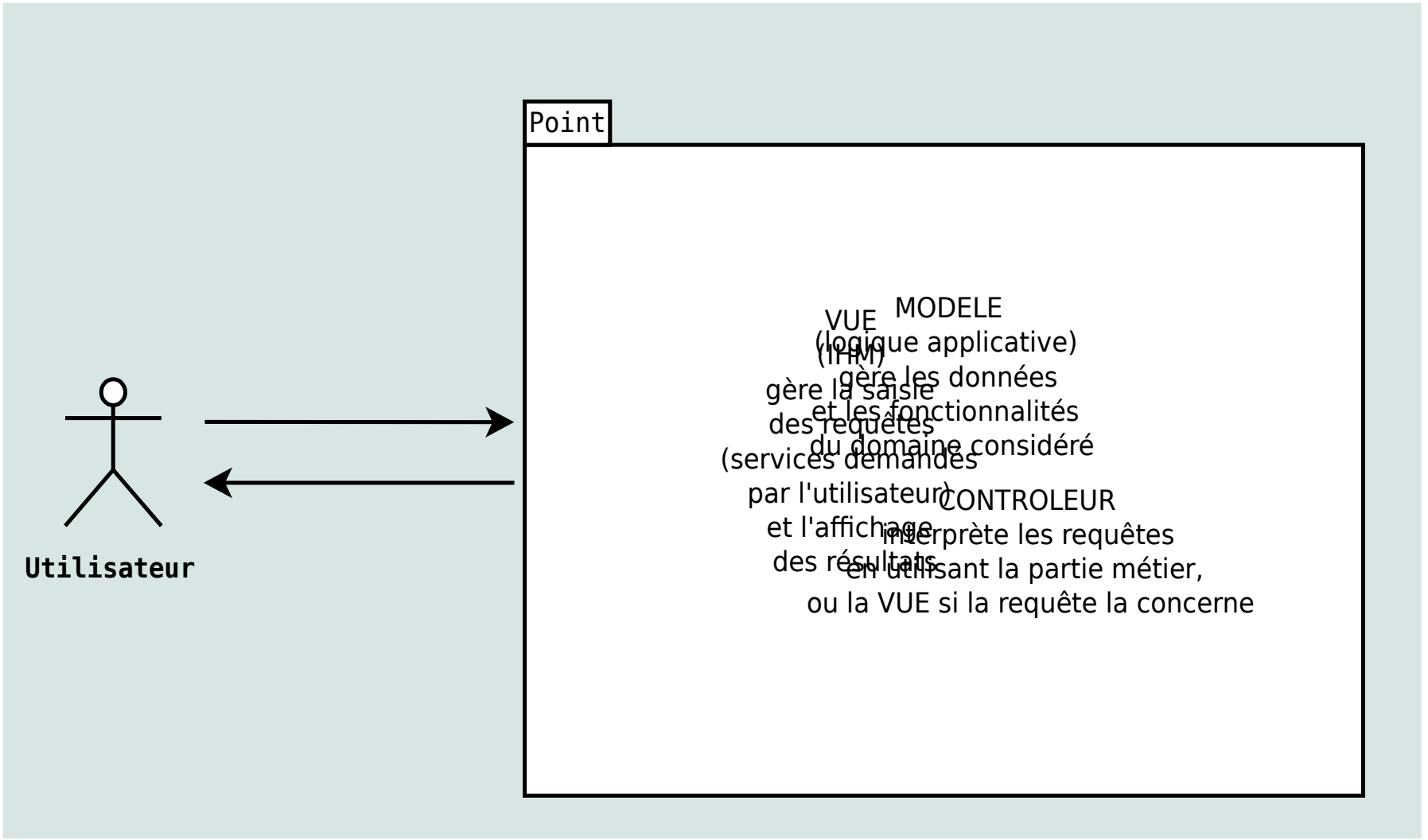
- entrer/modifier les coordonnées du point

L'application doit :

- afficher les coordonnées du point
- afficher la distance du point à l'origine

Une mauvaise conception

Le *modèle*, la *vue* et le *contrôleur* sont (mélangés) au sein d'une même classe



La classe *Point* contient le *modèle*, la *vue*...

```
public class Point
{
    private float x, y ;
                                // valeurs par défaut
    public Point() {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 distance() {...}


    public void saisirPoint(){...}
    public void afficherPoint() {...}
    public void afficherDistance() {...}
    public void activerVuePoint() {...}
    public void saisirAbscisse() {...}
    public void saisirOrdonnee() {...}
}
```

modèle :
gestion d'un point

vue :
gestion des interactions avec l'utilisateur (affichages et saisies)

...et le *contrôleur* qui n'est pas clairement localisé au sein de la classe

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

      …                    // VUE

      public void saisirPoint()
      {
       float unX = this.saisirAbscisse() ;
       float unY = this.saisirOrdonnee() ;

      this.majPoint(unX, unY);

      }
      ...
}
```

modèle

SaisirPoint() correspond à une requête (demande) de l'utilisateur, requête proposée par la vue

l'interprétation de la requête : relève du *contrôleur*

instructions s'adressant à la *vue* qui gère les interactions avec l'utilisateur.
Ici demande les coordonnées pour le *Point* courant.

instruction s'adressant au *modèle* pour affecter les coordonnées saisies au *Point* courant
(aucune interaction avec l'utilisateur)

La partie *contrôle* est éclatée dans les méthodes de la *vue*

Une classe cliente active la *vue* (*IHM*)

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

```
abscisse  = 0.0
ordonnee  = 0.0
distance  = 0.0


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


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


entrer votre choix (puis Entree)
```

menu affiché quand on active la *vue* : attend le choix de l'utilisateur (les choix correspondent à des requêtes prédéfinies)

La classe *Client* (et l'*utilisateur* qui en bénéficie) n'interagit qu'avec la *vue*

Une (à peine) meilleure conception

Le *modèle*, la *vue* et le *contrôleur* sont clairement séparés dans la classe *Point*

```
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() {...} ;
}
```

modèle :
gestion d'un point

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

contrôleur :
interprète les requêtes
(ici une seule requête)

Chaque requête-utilisateur de la *vue* délègue son interprétation à une méthode du *contrôleur*

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

                         // CONTROLEUR
     ...
     public void gererSaisirPoint()
     {
          float unX = this.saisirAbscisse() ;
          float unY = this.saisirOrdonnee() ;

          this.majPoint(unX, unY) ;
     }
}
```

modèle

délègue au *contrôleur* l'interprétation de la requête *saisirPoint()*

interprétation de la requête *saisirPoint()*

le *contrôleur* s'adresse à la *vue* pour la saisie des valeurs...

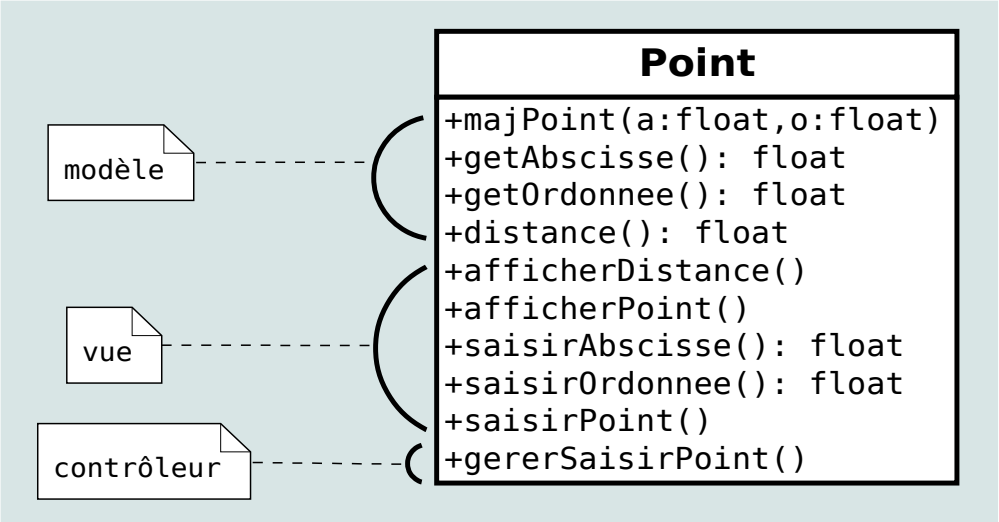
...puis le *contrôl*eur s'adresse *au modèle* pour l'affectation des valeurs

Laisser le *modèle*, la *vue* et le *contrôleur* dans une même classe contrevient à plusieurs principes de conception objet

- Séparation des interfaces (Interface segregation principle) (SOL**I**D)
- Responsabilité unique (Single responsability principle) (**S**OLID)
- Séparer ce qui change du reste
- Dépendre d'interfaces non d'implémentations

Principe de responsabilité unique (SOLID : Single responsability principle)

Une classe doit avoir une responsabilité unique
(une seule raison de changer)



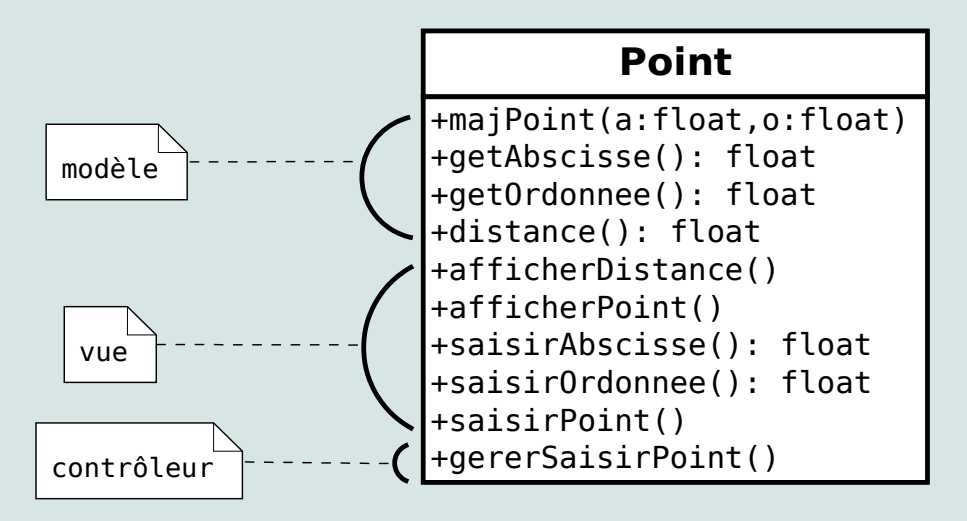
La classe *Point* a trois responsabilités :

- gérer un point (*modèle*)
- gérer l'*IHM* relative à un point (*vue*)
- gérer l'interprétation des requêtes de l'utilisateur (*contrôleur*)

Principe de séparation des interfaces (SOLID : Interface segregation principle)

Un client ne doit jamais être obligé de dépendre d'une interface qu'il n'utilise pas

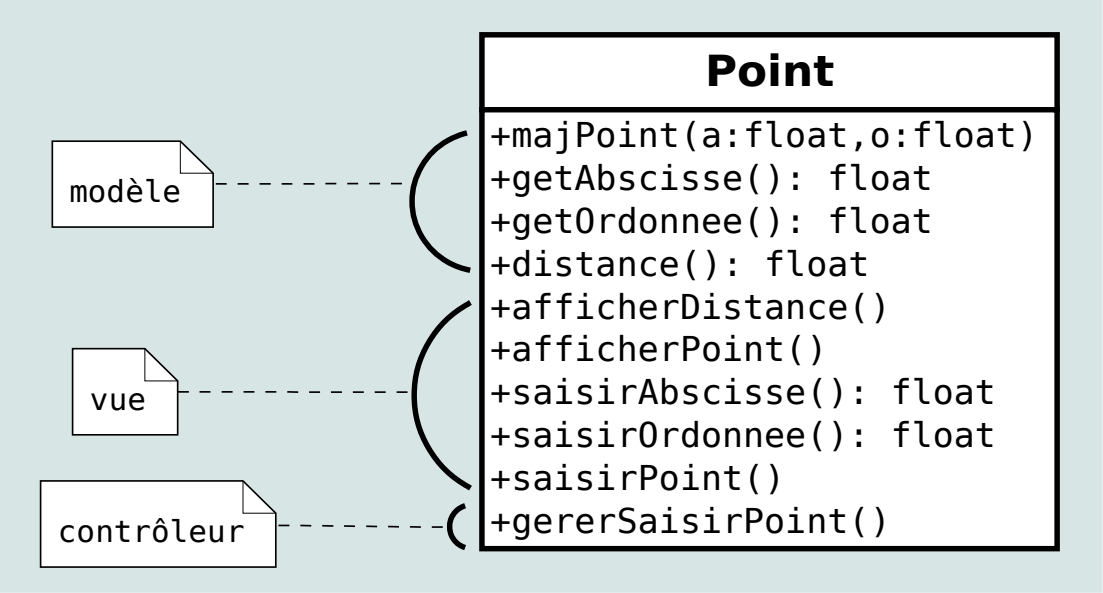
(*interface* peut se comprendre comme *interface* au sens java, ou comme la partie publique d'une classe)



Tout client de la classe *Point* qui souhaite gérer un point sans se soucier de la *vue* (IHM) ou du *contrôle* reste dépendant des méthodes relatives à ces parties qu'il n'utilisera pas.

Séparer ce qui change du reste

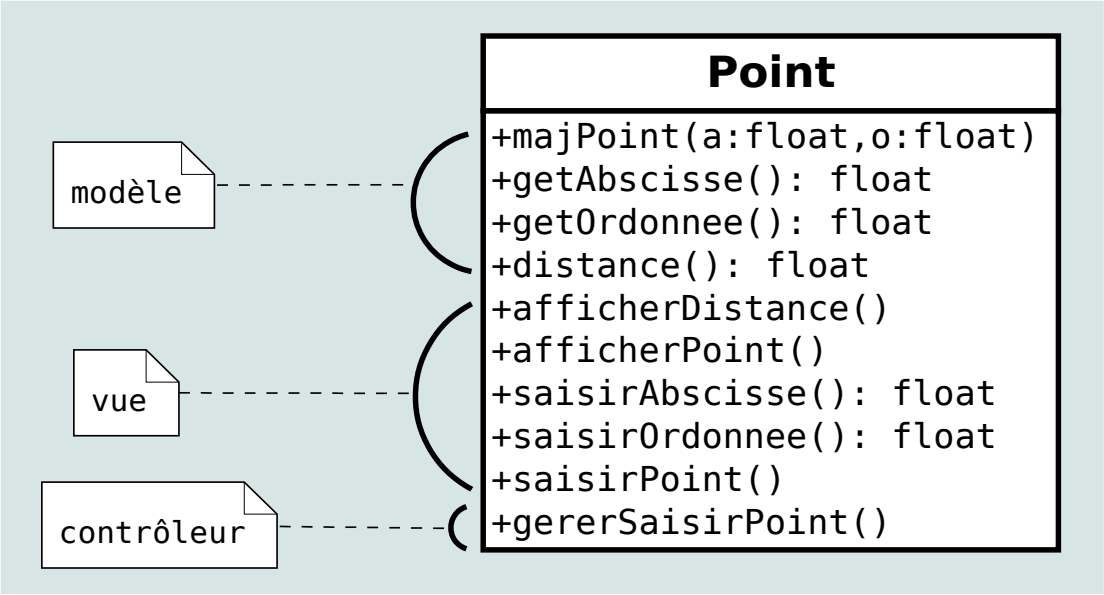
Les éléments d'une classe susceptibles de changer doivent être placés dans d'autres classes liées à la première par composition



La *vue* et le *contrôleur* 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 du *modèle*, de la *vue* et du *contrôleur*.



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

Conséquences du non respect des principes de conception objet

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

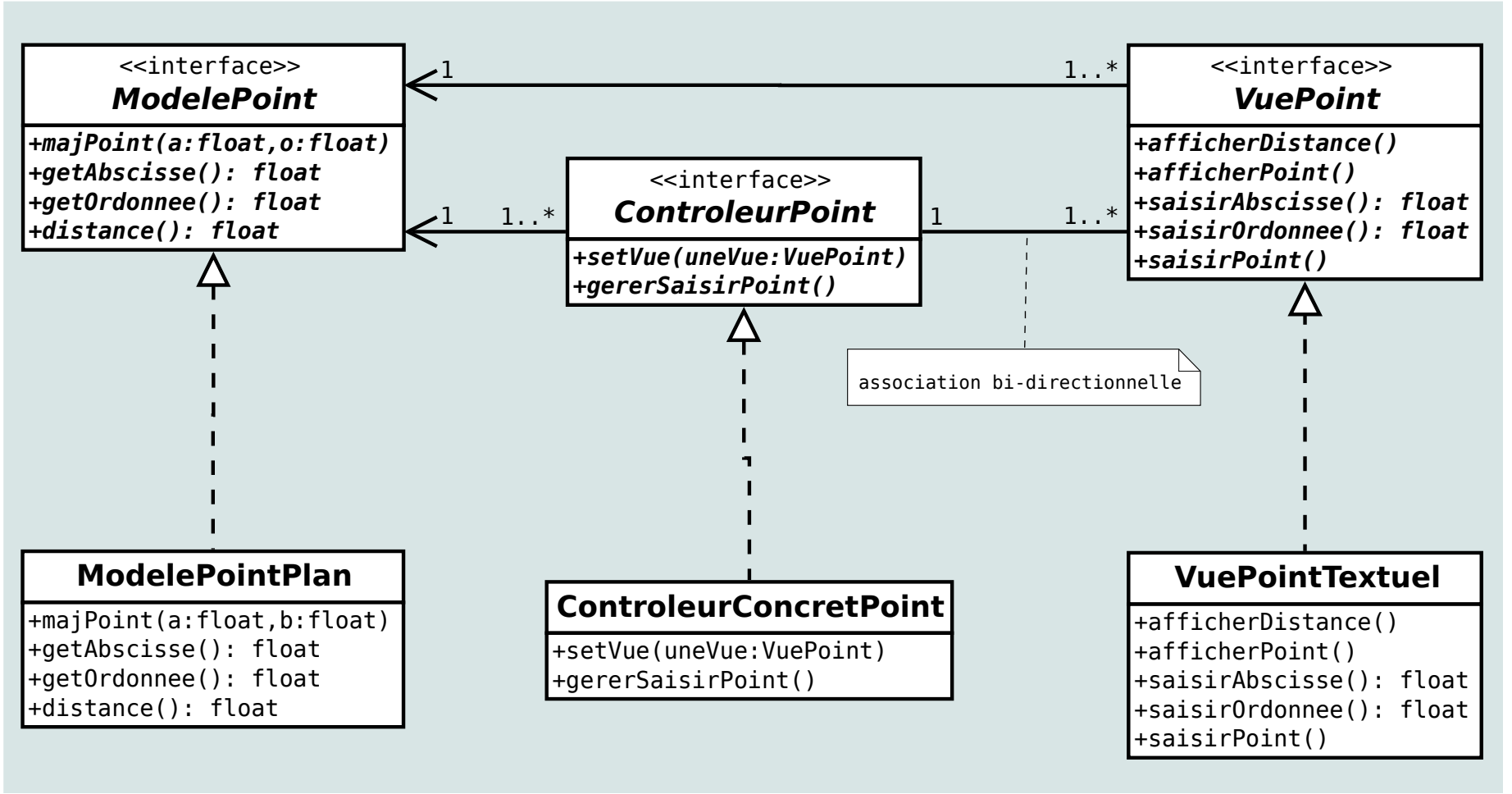
- Manque d'indépendance
  - Le *modèle*, la *vue* et le *contrôleur* risquent de dépendre structurellement les uns des autres
  - on ne peut plus confier le *modèle*, la *vue* et le *contrôleur* à des concepteurs/développeurs distincts

- Maintenance/extensibilité difficiles
  modifier une des parties aura des conséquences sur les autres et sera difficile à organiser

- Manque de souplesse
  impossible de changer facilement la *vue* et/ou le *contrôleur*

Une meilleure conception

On place le *modèle*, les *vues* et les *contrôleurs* dans des interfaces (et classes) distinctes



il peut y avoir plusieurs *vues* et plusieurs *contrôleurs* associés au même *modèle*
(chaque *vue* est en générale associée à un *contrôleur*)

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

Le *modèle* ne connaît ni les *contrôleurs* ni les *vues*

Une *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 données

Un *contrôleur*

```
public class ControleurConcretPoint implements ControleurPoint
{

    private ModelePoint modele ;
    private VuePoint    vue    ;

    public ControleurConcretPoint(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 le *modèle* et la *vue* contrôlée (déclarer un *ArrayList<VuePoint>* si plusieurs *vues* sont contrôlées)

pour changer la *vue* devant être contrôlée

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 client

```
public class Client
{

    public static void main(String[] args)
    {

        ModelePoint m = new ModelePointPlan() ;


        ControleurPoint c = new ControleurConcretPoint(m) ;


        VuePoint v = new VuePointTextuel(c, m) ;


        c.setVue(v) ;


        v.activerVue() ;
    }
}
```

1) on créé un *modèle*

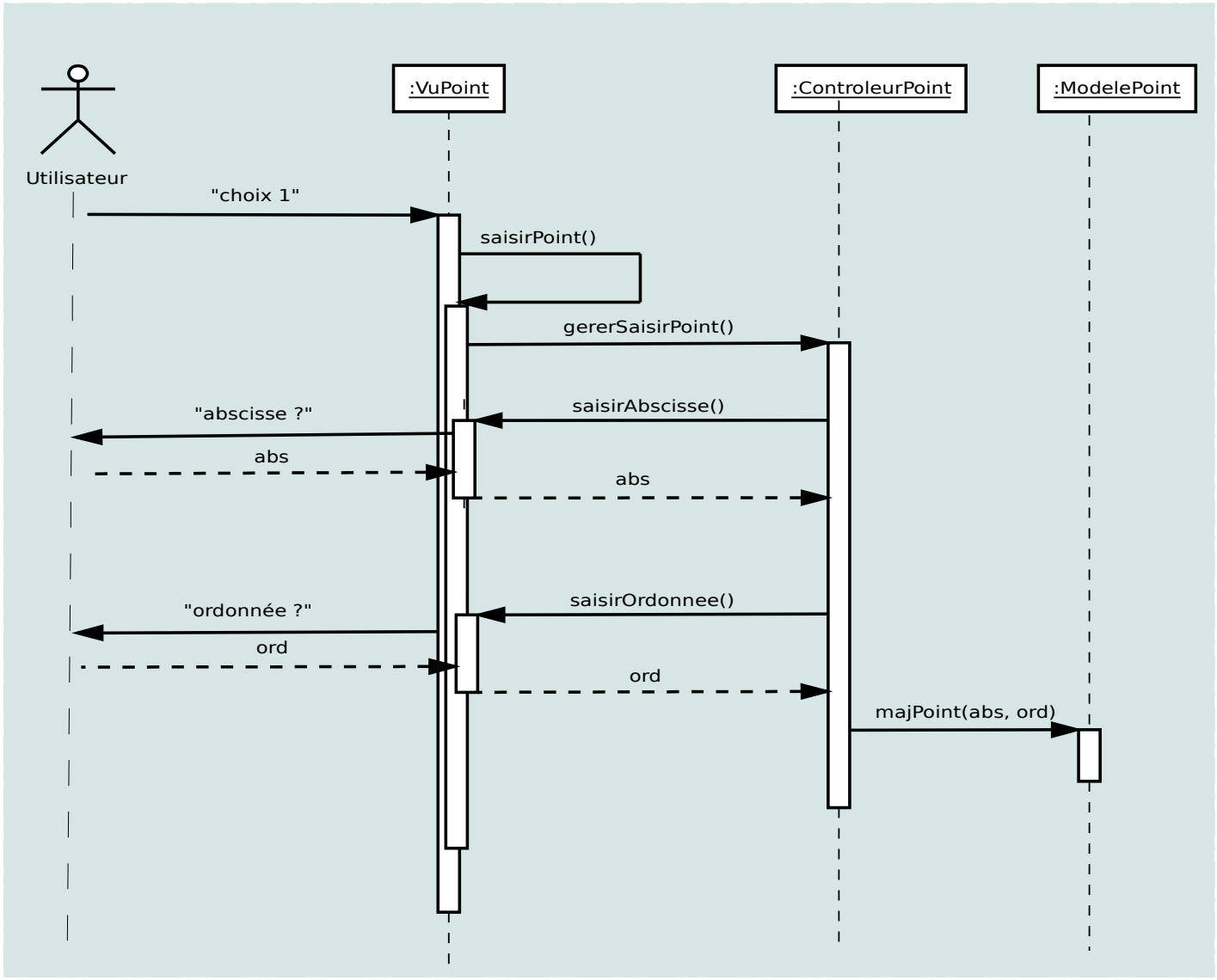
2) on créé un *contrôleur* et on lui associe le *modèle*

3) on créé une *vue* et on lui associe le *contrôleur* et le *modèle*

4) on associe la *vue* au *contrôleur* (l'association est bi-directionnelle)

4) on active la *vue* (*IHM*) qui attend les requêtes de l'utilisateur

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***

La meilleure conception : l'architecture modèle-vue-contrôleur

Objectifs

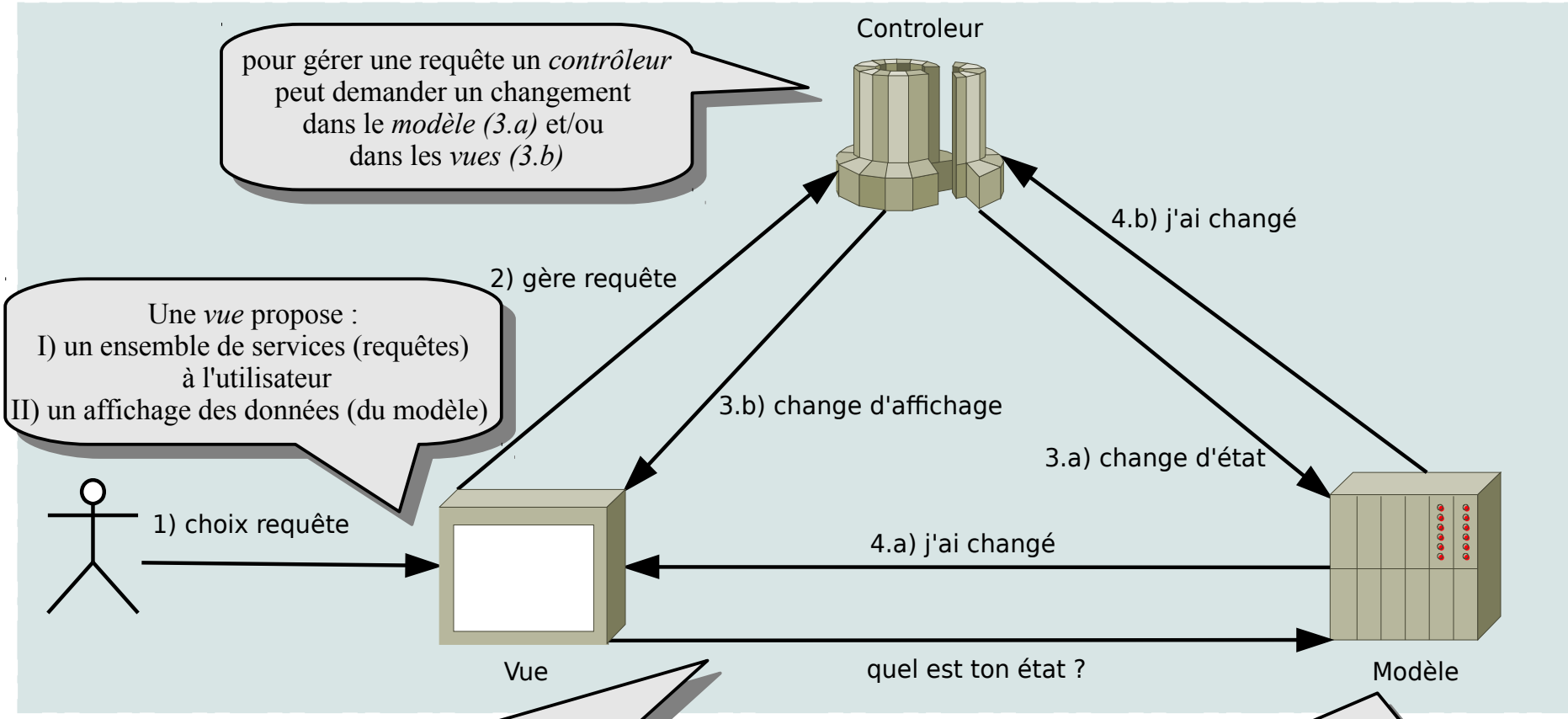
- Mise à jour automatique des vues (design pattern *observateur*)
  (les vues changent automatiquement quand le modèle change)

- Mise à jour automatique des contrôleurs (design pattern *observateur*)
  (*les contrôleurs changent automatiquement quand le modèle change*)

- Séparation des vues et des contrôleurs (design pattern *stratégie*)
  (on peut dynamiquement changer de contrôleur et/ou de vue)

- Imbrication des vues (design pattern *composite*)

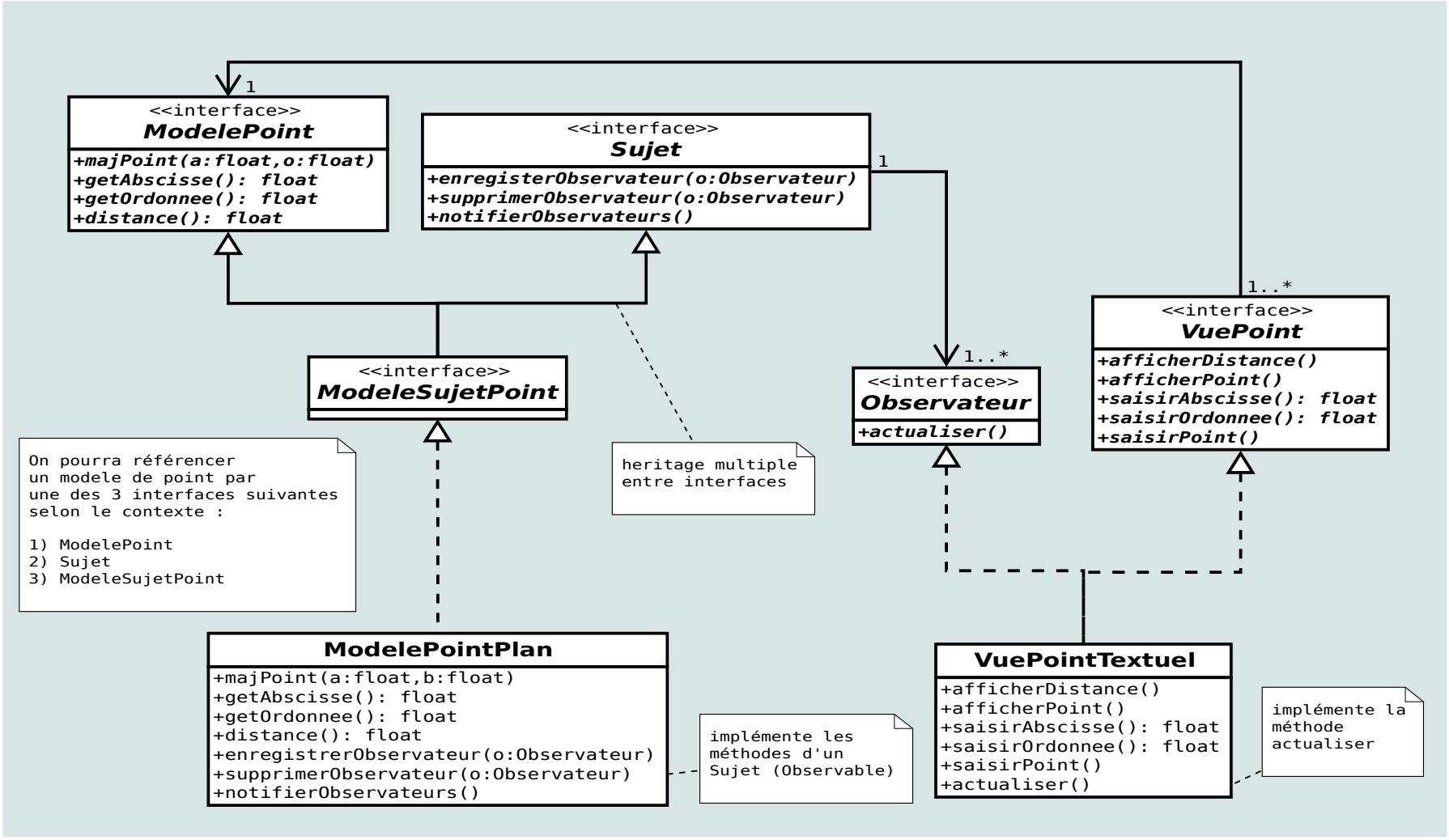
Dynamique générale de l'architecture MVC



Les *vues* et les *contrôleurs* sont automatiquement avertis des changements du *modèle*

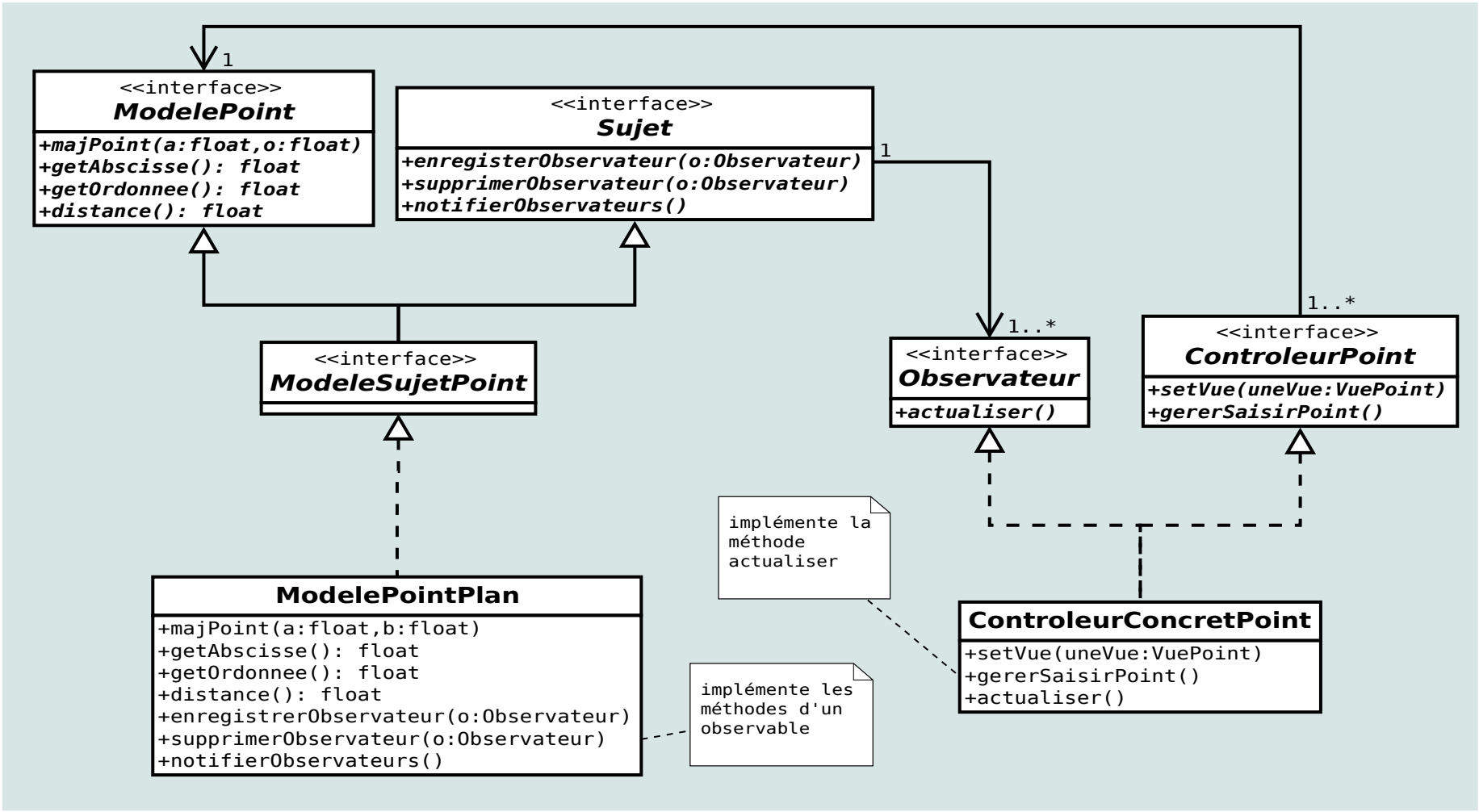
Mise à jour automatique des vues (design pattern *Observateur*)

Les *vues* sont des *observateurs* du *modèle* (qui est l'*observable*)



Mise à jour automatique du contrôleur (design pattern *Observateur*)

Les *contrôleurs* sont des *observateurs* du *modèle* (qui est l'*observable*)



Le *modèle*

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

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

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

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

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

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

liste des *Observateur(s)*
(contiendra les *vues* et les *contrôleurs*)

les *vues* et les *contrôleurs* sont actualisés

le *modèle* notifie les *vues* et
les *contrôleurs*
quand l'état du point change

Une *vue*

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

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

une *vue* est (aussi) un *Observateur*

la *vue* s'enregistre comme O*bservateur* du *modèle*

quand 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

Un *contrôleur*

```
public class ControleurConcretPoint implements ControleurPoint,
                                                Observateur
{

    private ModelePoint modele ;
    private VuePoint    vue    ;

    public ControleurConcretPoint(ModeleSujetPoint unModelePoint)
    {
        this.modele = unModelePoint ;
        unModelePoint.enregistrerObservateur(this) ;
    }


    public void actualiser()
    {
        ...
    }

    ...
}
```

le modèle est typé par l'interface *ModelePoint* et non par la classe concrète *ModelePointPlan*

le *contrôleur* s'enregistre comme *Observateur* du *modèle*

instructions exécutées quand le *modèle* demande au *contrôleur* de s'actualiser.

Le client ne change pas

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

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

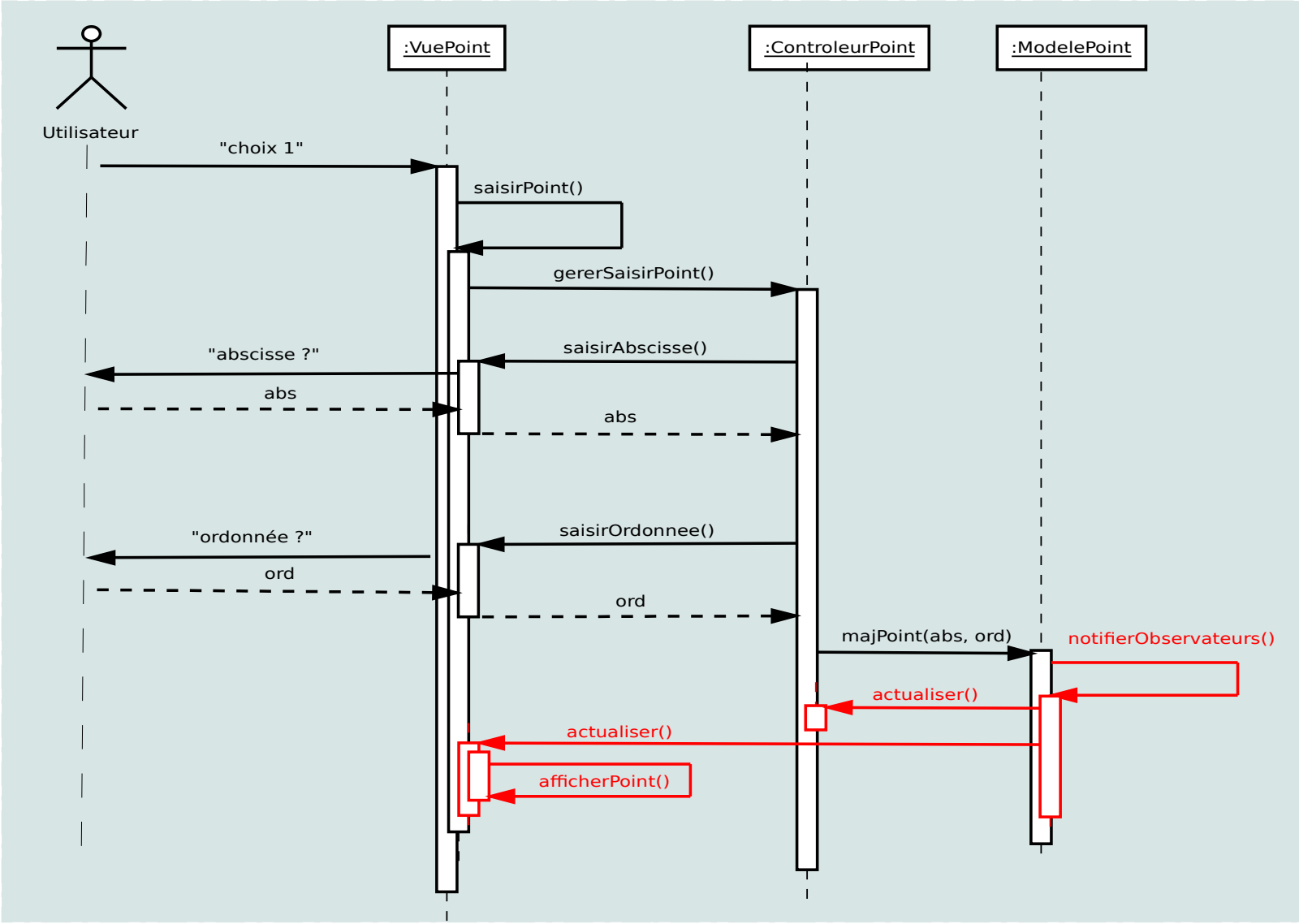
```
abscisse  = 0.0
ordonnée  = 0.0
distance  = 0.0


**************** 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

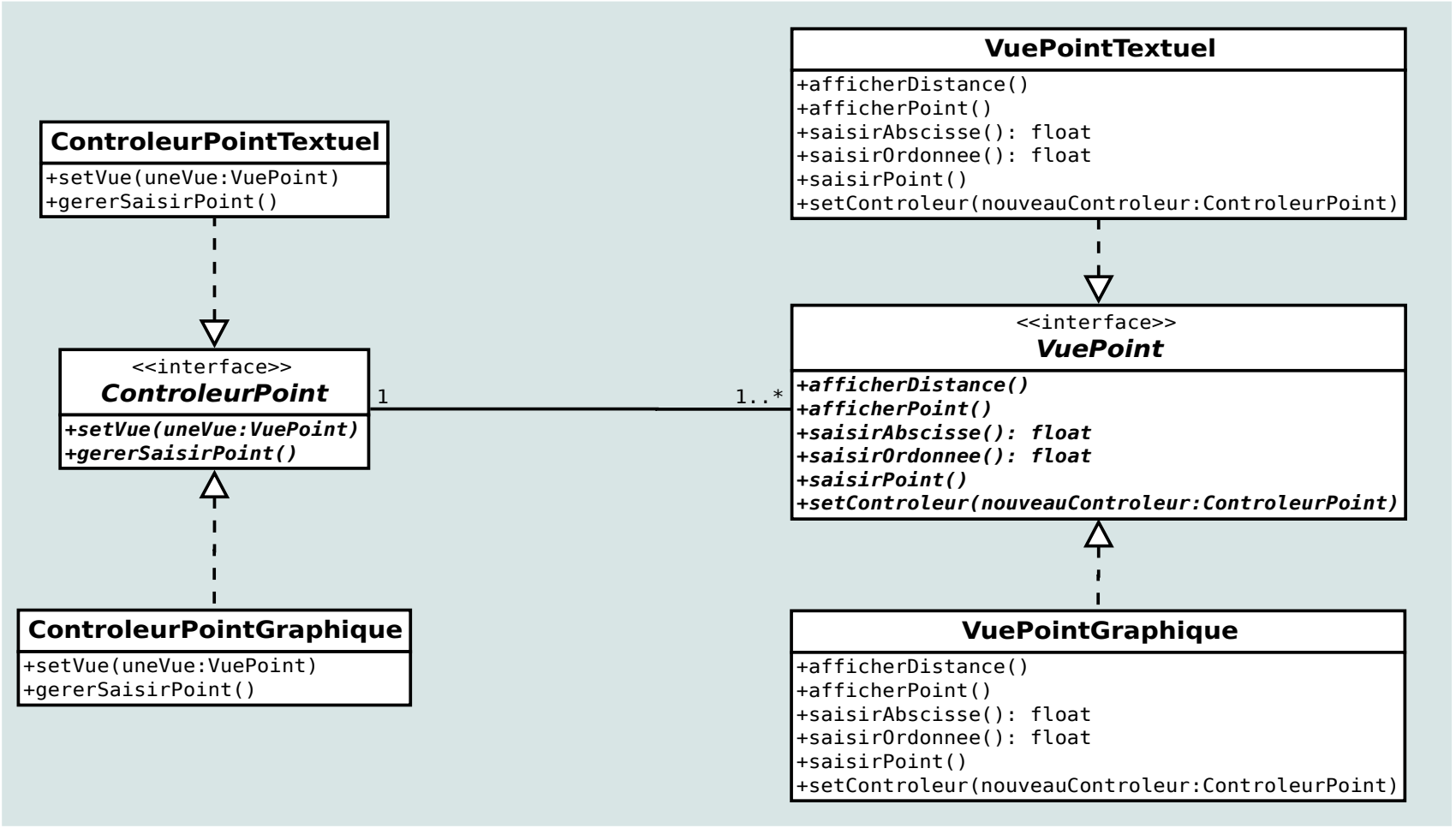
la *vue* est automatiquement mise à jour

Dynamique d'une requête utilisateur



Le *modèle* a changé : la *vue* est actualisée et affiche les nouvelles coordonnées
(si nécessaire le *contrôleur* est aussi actualisé)

Un *contrôleur* définit le comportement d'une ou plusieurs *vues* (design pattern *stratégie*)



Le *contrôleur* encapsule le comportement d'une ou plusieurs *vues*
(il définit les actions à faire pour satisfaire les services que la vue propose à l'utilisateur)

Changement dynamique de *contrôleur*

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

        c1.setVue(v) ;
        v.activerVue() ;

        m.supprimerObservateur(c1) ;

        ControleurPoint c2 = new AutreControleurPoint(m) ;
        c2.setVue(v) ;
    }
}
```

on supprime l'association
entre le *modèle* et le *contrôleur* courant

on créé un autre type de *contrôleur*
(qui sera associé au *modèle* par le constructeur)

On associe la *vue v* au nouveau contrôleur. La vue *v* sera interprétée différemment
(pratique courante pour utiliser une *vue* avec une autre application,
ou une autre version d'une application)

L'architecture MVC permet de changer/ajouter dynamiquement des *contrôleurs* et des *vues*

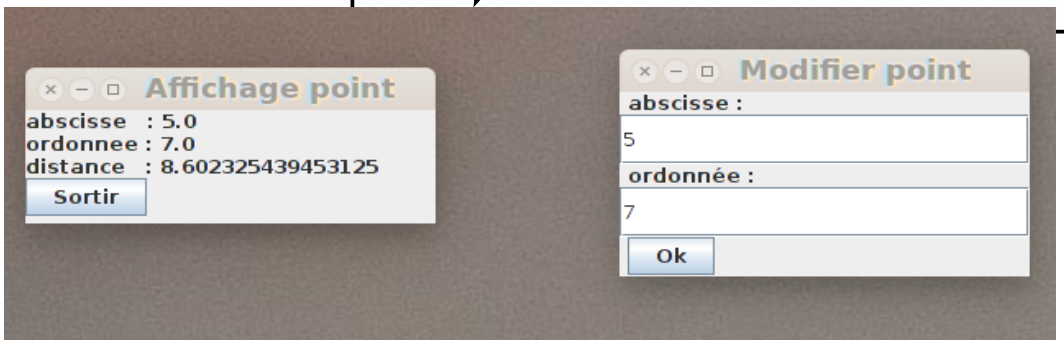
ici 2 *vues* différentes et 2 *contrôleurs* identiques : l'utilisateur peut interagir via les deux *vues*

autre *contrôleur* (identique au premier) associé à une autre *vue* du même modèle

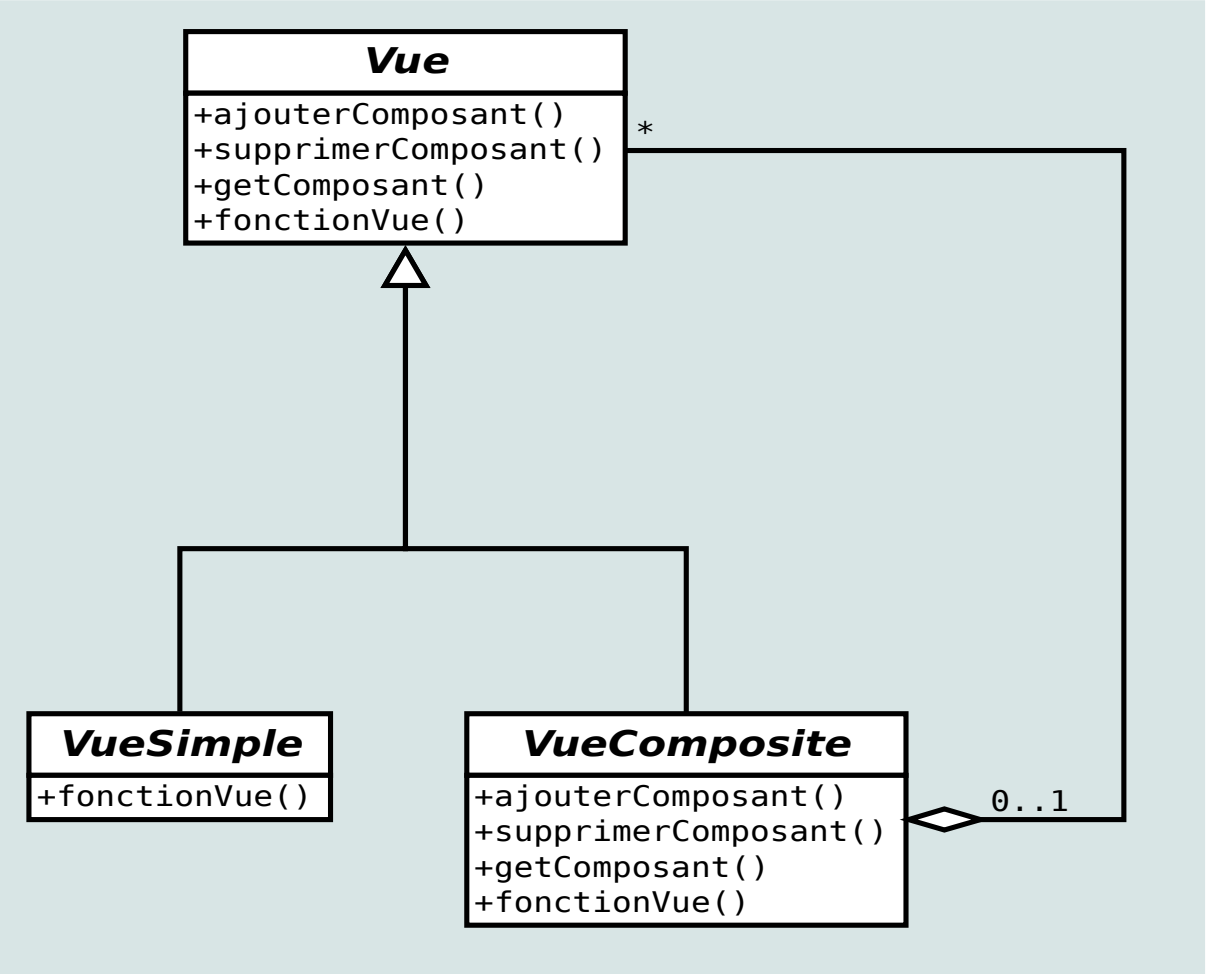
(si on avait mis un seul contrôleur, il aurait demandé à chaque vue la saisie des abscisse et ordonnée)

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

        ControleurPoint c2 = new ControleurConcretPoint(m) ;
        VuePoint        v2 = new VuePointGraphique(c2, m) ;
        c2.setVue(v2) ;

        v2.activerVue() ;
        v1.activerVue() ;
    }
}
```

```
abscisse  = 0.0
ordonnee  = 0.0
distance  = 0.0

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

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

entrer votre choix (puis Entree)

abscisse  = 5.0
ordonnee  = 7.0
distance  = 8.602325439453125
```



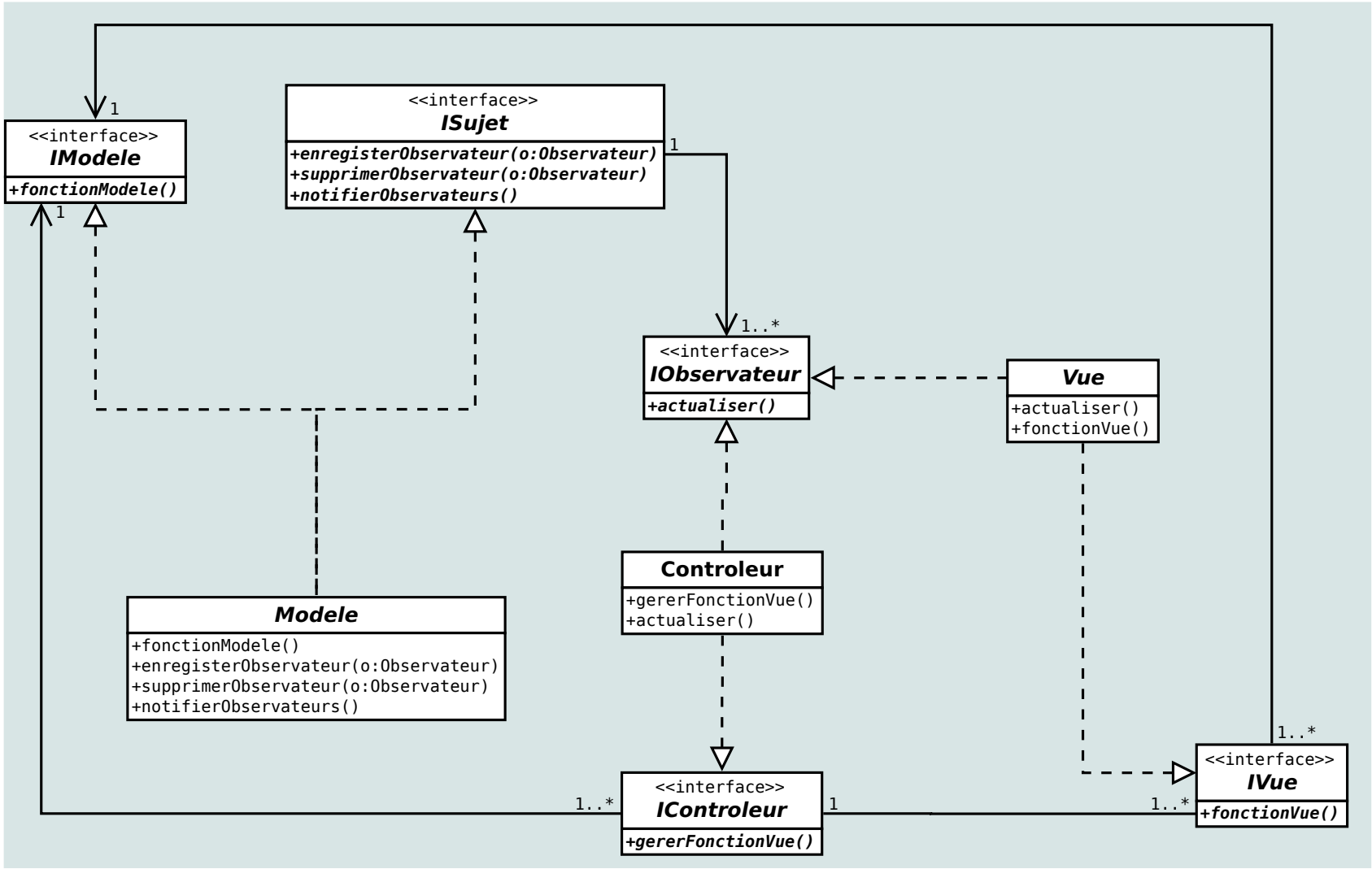
Les *vues* mettent (généralement) en œuvre le design pattern *Composite*



Une *vue* contient d'autre vues…
(les composants graphiques contiennent d'autres composants graphiques...)

Le pattern composé modèle-vue-contrôleur

Architecture MVC



- le *modèle* gère (exclusivement) les données et fonctions du domaine considéré
- la *vue* gère (exclusivement) les interactions avec l'utilisateur
- le *contrôleur* gère l'interprétation des requêtes que la *vue* propose à l'utilisateur
  (et éventuellement vient modifier la présentation de la vue)

L'architecture MVC en java (cours IHM)

- Les composants graphiques *swing* s'appuient sur une architecture MVC

- La mise en œuvre de MVC dans *swing* n'est pas totalement correcte : la *vue* inclut le *contrôleur* (ce qui contrevient a presque tous les principes de conception objet)

- Le programmeur n'a pas a gérer explicitement MVC qui est encapsulé dans les composants *swing* qu'il utilise pour créé la *vue*
(une bonne conception consiste à créer son propre contrôleur, appelé par les traitements des événements captés par les listeners)

Principes de conception rencontrés

Principes mis en œuvre

Principe de responsabilité unique

MVC attribue une seule responsabilité au *modèle*, à la *vue* et au *contrôleur*

Séparation des interfaces

Grâce à MVC le client ne dépend que de ce qu'il utilise
(ex : utiliser le modèle n'implique pas de connaître la *vue ou* le *contrôleur*).

Séparer ce qui change du reste

MVC permet à des concepteurs/développeurs distincts de concevoir/développer/maintenir
le *modèle*, la *vue* et le *contrôleur*

Dépendre d'interfaces et non d'implémentations

MVC permet de facilement changer de *vue* et/ou de *contrôleur*
car le client ne dépend que d'interfaces