Les architectures

Adrien CAUBEL

8 septembre 2022

Table des matières

Table des matières 2

1 Les éléments d'une interface graphique **1**
1.1 Introduction . . . 1
1.1.1 La logique métier : le Modèle . . . 1
1.1.2 L'affichage des données : la Vue . . . 1
1.2 Une première application graphique . . . 1
1.2.1 Objectif à réaliser . . . 1
1.2.2 Comment réaliser l'application ? . . . 2
1.2.3 Implémenter la solution . . . 2
1.3 Notifier la Vue d'un changement . . . 3
1.3.1 Notifier la Vue d'un changement . . . 4
1.3.2 Mise à jour de la Vue . . . 4
1.3.3 Avantages de cette solution . . . 5
1.4 Conclusion . . . 5

2 Complexifier notre interface graphique **7**
2.1 Introduction . . . 7
2.2 Réalisation d'une architecture plus complexe . . . 7
2.2.1 Objectif à réaliser . . . 7
2.2.2 Diagramme de classe . . . 8
2.2.3 Implémentation . . . 8
2.2.4 Diagramme de séquence . . . 10
2.2.5 Cette conception est-elle satisfaisante ? . . . 10
2.3 Conclusion . . . 11

3 L'architecture Modèle-Vue-Contrôleur **13**
3.1 Introduction . . . 13
3.2 L'architecture MVC . . . 13
3.2.1 Les composants . . . 13
3.3 Construire l'exemple via l'architecture MVC . . . 13
3.3.1 Conserver la relation Vue/Modèle . . . 13
3.3.2 Le Contrôleur est le point d'entrée de notre application . . . 14
3.3.3 La communication Vue/Contrôleur : détecter les évènements . . . 15
3.3.4 La communication Vue/Contrôleur : une alternative . . . 16
3.3.5 Sortir le choix de la vue du Modèle . . . 16
3.3.6 Diagramme de séquence final . . . 17

3.3.7 Résumé de la mise en place du MVC 18
3.3.8 Questions soulevées 19
3.4 Amélioration de l'architecture MVC 19
3.4.1 Le Modèle doit-il avoir connaissance les Vues ? 19
3.4.2 Une interface pour découpler : patron Observateur 19
3.4.3 Relation transitive Contrôleur-Vue-Modèle / Contrôleur-Modèle 20
3.4.4 Résumé de l'architecture MVC 21
3.4.5 Conclusion 21
3.5 Variantes de l'architecture précédente 22
3.5.1 Relation Contrôleur-Vue inversée 22
3.5.2 Relation bidirectionnelle Contrôleur-Vue 25
3.5.3 Conclusion 26

Bibliographie **29**

1

Les éléments d’une interface graphique

1.1 Introduction

Le développement d’une application graphique nécessite d’une part la logique métier rassemblant les données de l’application et d’autre part l’affichage de ces informations au travers d’une interface graphique.

1.1.1 La logique métier : le Modèle

Dans la suite des architectures nous utiliserons le nom *Modèle* pour définir la logique métier de notre application. À travers le *Modèle* nous évoquons plusieurs aspects de notre application :
- les données de l’application
- la logique métier
- la logique de persistance et la lecture des données en base

Par exemple, si nous créons une application bancaire, nous aurons respectivement :
- les comptes, les clients
- les opérations de crédit et retrait avec leur vérification (solde suffisant, etc.)
- enregistrer et lire le solde en base de données

Les données traitées par le Modèle sont brutes. Il ne prend en compte aucune considération sur leur présentation (en gras, en route, etc.) ou leur transformation (majuscule, etc).

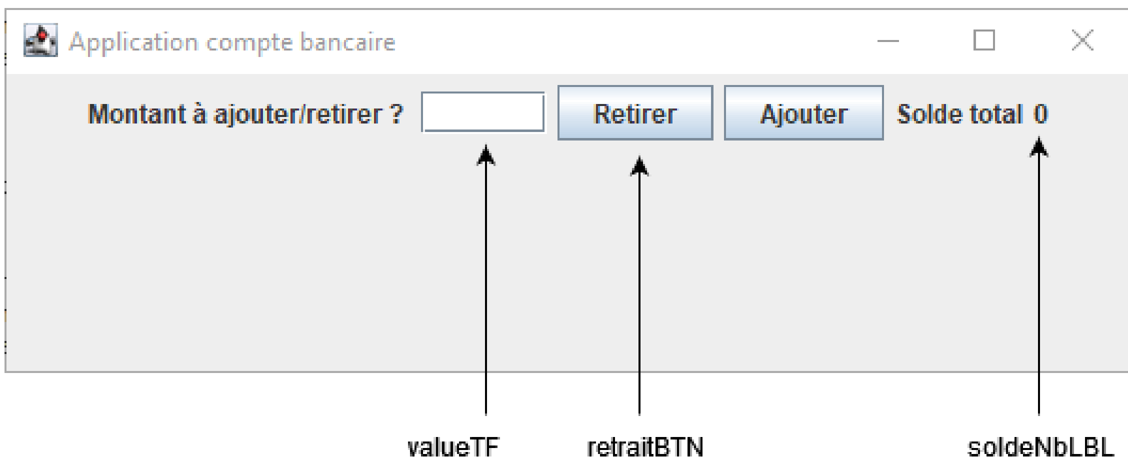
1.1.2 L’affichage des données : la Vue

La Vue est l’élément graphique de notre application. Elle contient des éléments visuels (boutons, images, champs de saisie, etc.) ainsi que la logique nécessaire pour afficher les données en provenance du Modèle (couleur, formatage, mise en évidence, etc.).

1.2 Une première application graphique

1.2.1 Objectif à réaliser

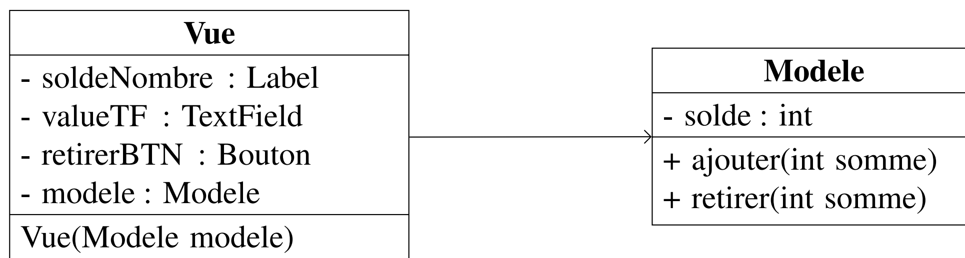
Nous souhaitons réaliser une application permettant d’ajouter ou retirer de l’argent à un compte bancaire.



1.2.2 Comment réaliser l'application ?

Les notions de Modèle et de Vue vont nous permettre de réaliser cette première application graphique. Pour ce faire nous allons définir les relations suivantes :

- Une Vue qui a la connaissance du Modèle. Ainsi lorsque la Vue souhaitera afficher un élément elle enverra une requête au Modèle.
- Un Modèle, qui, quant à lui, est complètement ignorant de l'interface graphique. Le Modèle sera mis à jour par la Vue lorsque l'utilisateur interagira.



1.2.3 Implémenter la solution

Cette solution implique que lorsqu'on clique sur le bouton pour valider un retrait nous réalisons les actions suivantes :

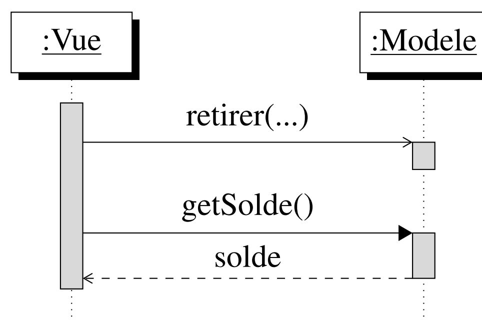
- retirer la somme sur le solde dans le compte bancaire (le Modèle [1]) .
- mettre à jour l'information dans l'interface graphique.

```
public class Modele {
   private int solde;

   public void ajouter(int montant) {
      solde += montant;
   }

   public void retirer(int montant) {
      solde -= montant;
   }
}
```

1. Dans les exemples suivant nous considèrerons le Modele comme étant un compte bancaire

```
public class Vue extends JFrame {
   public Vue(Modele modele) {
      retraitBTN.addActionListener(new ActionListener() {

         @Override
         public void actionPerformed(ActionEvent e) {
            modele.retirer(Integer.parseInt(valueTF.getText()));
            soldeNbLBL.setText("" + modele.getSolde());
         }
      });
   }

   public static void main(String[] args) {
      Modele modele = new Modele();
      Vue vue = new Vue(modele);
   }
}
```

Le diagramme de séquence suivent nous montre l'enchainement des instructions lorsqu'on clique sur le bouton `retraitBTN`.



Remarque :
Nous nous sommes obligés à passer par le Modèle pour mettre à jour le solde. Cette bonne pratique permet de dire que le Modèle est le cur de notre application et que toutes informations à afficher sont issues du Modèle. La Vue ne doit avoir aucune intelligence concernant les données à afficher.

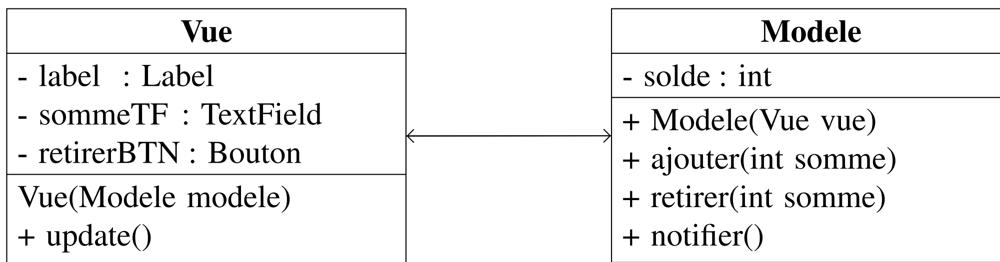
1.2.3.1 Problème de cette solution

Malgré sa facilité à être implémenté, cette solution n'est pas envisageable pour la création d'une application complète. En effet, le Modèle peut être modifié par d'autres acteurs que la Vue. Dans le code précédent, lorsqu'un acteur externe modifie le Modèle la Vue n'est pas mise au courant de ce changement. Par conséquent, l'interface graphique n'est pas mise à jour.

1.3 Notifier la Vue d'un changement

Pour pallier à ce problème, la solution est de notifier la Vue dès que le Modèle est modifié. Cela implique une relation bidirectionnelle entre la Vue et le Modèle :

— la Vue a connaissance du Modèle : pour mettre à jour le Modèle (solution précédente).
— le Modèle a connaissance de la Vue : pour notifier que le solde a été modifié (par un acteur externe).



1.3.1 Notifier la Vue d'un changement

Lorsqu'une méthode métier dans le Modèle est appelée en plus de réaliser de modifier la valeur du solde le Modèle informe la Vue du changement.

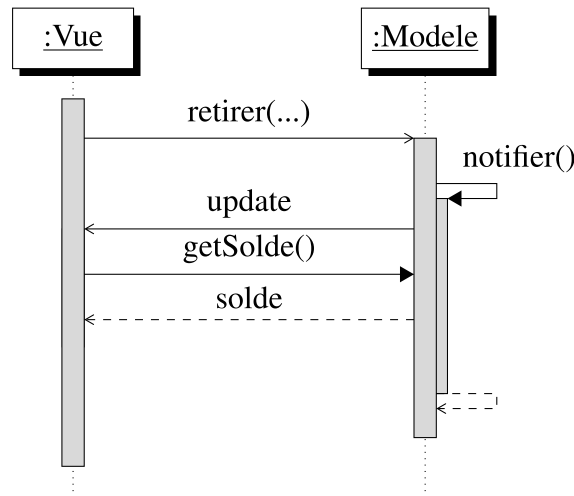
```
public class Modele {
   public void retirer(int montant) {
      solde -= montant;
      notifier();
   }

   public void notifier() {
      vue.update();
   }
}
```

1.3.2 Mise à jour de la Vue

Dans la Vue nous avons rajouter une méthode publique `notifier()` qui doit être appelée par le Modèle pour mettre à jour la Vue. Ainsi, dans `actionPerform()` nous n'avons plus besoin de récupérer les informations auprès du Modèle.

```
public class Vue extends JFrame {
   public Vue(Modele modele) {
      retraitBTN.addActionListener(new ActionListener() {

         @Override
         public void actionPerformed(ActionEvent e) {
            modele.retirer(Integer.parseInt(valueTF.getText()));
         }
      });
   }

   public void update() {
      // On récupère les modifications auprès du Modele (le compte bancaire)
      soldeNbLBL.setText("" + modele.getSolde());
   }

   public static void main(String[] args) {
      Modele modele = new Modele();
      Vue vue = new Vue(modele);
      modele.setVue(vue);
   }
}
```

Comme précédemment nous pouvons représenter l'action d'un retrait grâce au diagramme de séquence. Avec cette nouvelle architecture, la récupération du solde auprès du Modèle ce fait uniquement lorsque la méthode `update` est déclenchée par le Modèle.



1.3.3 Avantages de cette solution

Le principal avantage de cette solution est la répartition des actions dans différentes méthodes. En effet, dans la première solution la méthode `actionPerformed()` été en charge de modifier de solde sur le compte bancaire et également de mettre à jour la Vue.
Dans cette seconde solution, nous avons séparé ces deux concepts. C'est seulement lorsque le Modèle est modifié que nous intervenons sur la Vue. De plus, si un autre acteur modifie notre Modèle nous serons capables d'en notifier la Vue pour quelle se mette à jour.

En contrepartie, nous avons une architecture un peu plus complexe qui fait intervenir une relation bidirectionnelle entre notre Modèle et notre Vue.

1.4 Conclusion

Dans ce premier chapitre nous venons de créer notre première application graphique grâce aux notions de Modèle et de Vue.

- Le Modèle est le cur de notre application. Il informe la Vue de se mettre à jour à chaque fois qu'il se voit modifié.
- La Vue est le point d'entrée de notre application. Elle met à jour le Modèle en fonction des interactions de l'utilisateur.

Nous avons également vu qu'une relation bidirectionnelle est nécessaire entre le Modèle et la Vue. Elle permet de s'assurer que la modification du Modèle par n'importe quel acteur est suivie d'une mise à jour de la Vue.

2

Complexifier notre interface graphique

2.1 Introduction

Dans ce chapitre nous allons complexifier l'architecture présentée dans le chapitre précédent. Ce chapitre est un chapitre de transition afin que vous compreniez pourquoi les architecture MVC, MVP, etc ... ont été créé et comment permettent-elles de réduire la complexité de notre application.

2.2 Réalisation d'une architecture plus complexe

2.2.1 Objectif à réaliser

Nous souhaitons améliorer l'application précédente en rajoutant une seconde Vue affichant le solde total. Lorsque l'utilisateur aura cliqué sur le bouton *retirer* ou *ajouter* la première Vue devra disparaitre et seulement la seconde sera affichée à l'écran.

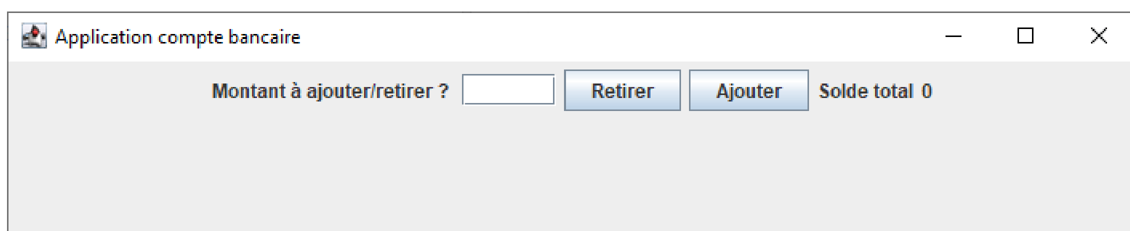


FIGURE 2.1 – Vue nř1 qui doit disparaitre lors d'un clic sur un bouton

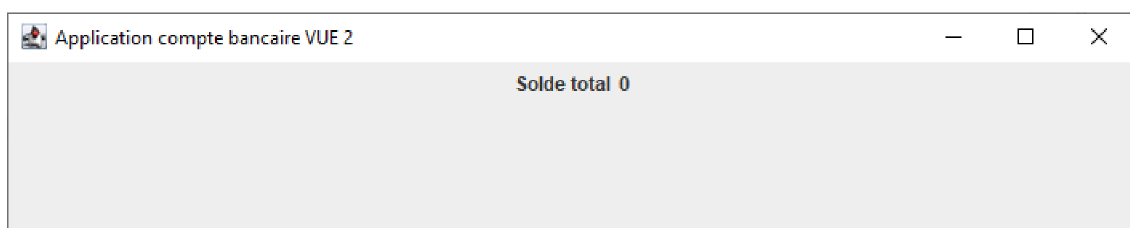
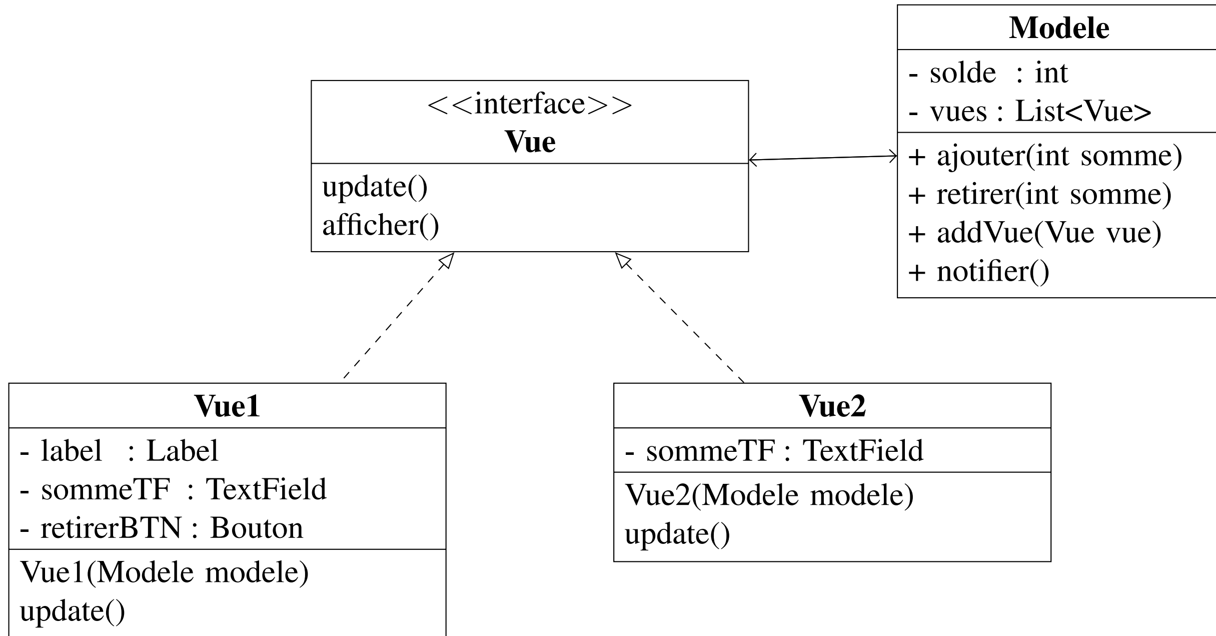


FIGURE 2.2 – Vue nř2 à afficher une fois un clic réalisé

2.2.2 Diagramme de classe

Pour réaliser cette application, nous allons faire les choix suivants :
- Définir une interface `Vue` qui sera implémentée par les classes `Vue1` et `Vue2`.
- Définir une méthode `afficher` qui permettra de dire si la Vue doit être affichée ou non.
- Définir une liste de `Vue` dans le `Modèle`
- Lorsque sur la `Vue1` un bouton est cliqué alors le Modèle notifiera les deux Vues.



2.2.3 Implémentation

2.2.3.1 Le Modèle

C'est dans la méthode `notifier` où nous spécifions la Vue qui doit être affichée (`get(1)`) et nous faisons disparaitre la vue principal de l'écran (`get(0)`).

```
public class Modele {
   private List<Vue> vues = new ArrayList<>();
   public void notifier() {
      for (Vue vue : vues) {
         vue.update();
      }
      vues.get(0).afficher(false);
      vues.get(1).afficher(true);
   }
   public void addVue(Vue vue) {
      this.vues.add(vue);
   }
}
```

2.2.3.2 Les Vues

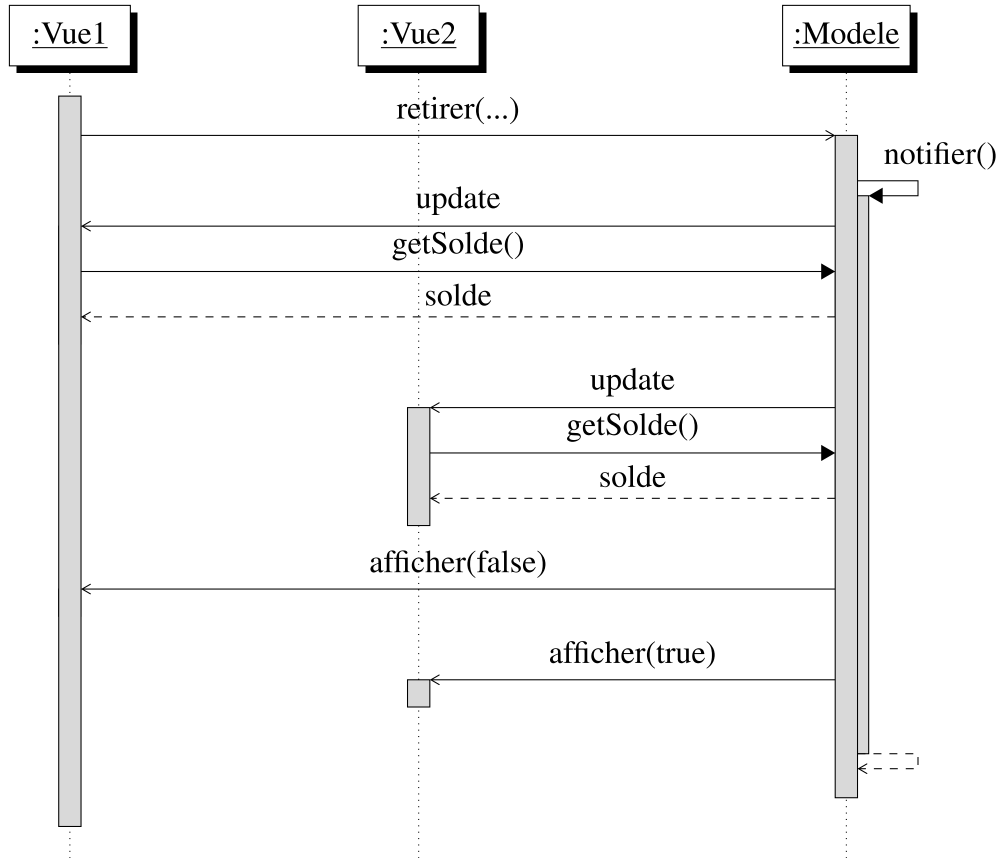
La `Vue1` est notre vue principale. Elle sera donc en charge de créer la `Vue2` est d'initialiser le Modèle. La `Vue2` quant à elle, implémente simplement l'interface `Vue`.

```
public class Vue1 extends JFrame implements Vue {
   public Vue1(Modele modele) {
      retraitBTN.addActionListener(new ActionListener() {

         @Override
         public void actionPerformed(ActionEvent e) {
            modele.retirer(Integer.parseInt(valueTF.getText()));
            soldeNbLBL.setText("" + modele.getSolde());
         }
      });
   }

   public void update() {
      soldeNbLBL.setText("" + modele.getSolde());
   }

   @Override
   public void afficher(boolean aAfficher) {
      setVisible(aAfficher);
   }

   public static void main(String[] args) {
      Modele modele = new Modele();
      Vue vue = new Vue1(modele);
      Vue vue2 = new Vue2(modele);
      modele.addVue(vue);
      modele.addVue(vue2);

   }
}
```

```
public class Vue2 extends JFrame implements Vue {
   public Vue2(Modele modele2) {
   @Override
   public void afficher(boolean aAfficher) {
      setVisible(aAfficher);
   }

   @Override
   public void update() {
      soldeNbLBL.setText("" + modele.getSolde());
   }
}
```

2.2.4 Diagramme de séquence

1. Lorsque l'utilisateur clique sur le bouton `retraitBTN` la méthode `retirer()` est déclenchée.
2. La méthode `retirer()` déclenche la méthode `notifier()` qui va avertir les vues (`update`) qu'elles doivent se mettre à jour.
3. À la fin de la méthode `notifier()` nous éteignons la `vue1` et faisons apparaitre la `vue2`.



2.2.5 Cette conception est-elle satisfaisante ?

2.2.5.1 Un problème de scalabilité

Est-ce que cette solution est réalisable avec des dizaines ou centaines de Vues ?

Le Modèle arriverait à afficher une certaine Vue parmi des dizaines. Cependant une telle opération augmente la complexité algorithmique de notre modèle et plus particulièrement de la méthode `notifier`. Donc même si c'est réalisable ce n'est pas évident à mettre en application.

2.2.5.2 Un problème d'organisation

Est-ce que le Modèle doit manipuler les Vues ?

C'est le problème majeur de notre application. Nous avons défini le Modèle comme un élément neutre qui ne doit dépendre de rien d'autre que de lui/même. En effet, un Modèle peut être exploité sur une interface graphique (comme ici), mais également sur d'autres supports : terminal, led, ...
Avec notre approche notre Modèle est dépendant de l'interface graphique nous perdons donc la capacité à pouvoir être utilisé par n'importe quel "outil" externe.

La réponse à la question est donc *Non*. Un Modèle ne doit ni connaitre ni manipuler les Vues.

2.2.5.3 Qui doit donc choisir la Vue à afficher ?

Les deux questions précédentes nous confirment que le Modèle n'a pas pour rôle de choisir la vue à afficher. Nous devrons donc définir une nouvelle entité (classe) dont son rôle sera spécifiquement de choisir la bonne Vue à afficher.
Dans l'architecture Modèle-Vue-Contrôleur ce sera le rôle du Contrôleur et dans l'architecture Modèle-Vue-Présentateur le rôle du Présentateur. Nous allons bien sûr détailler ces architectures dans les chapitres suivants.

2.3 Conclusion

Ce chapitre nous a permis de comprendre pourquoi l'architecture précédente n'est pas adaptée pour le développement d'interface graphique.
Premièrement, il devient difficile de définir des règles pour manipuler les Vues. Secondement, un Modèle ne doit pas dépendre des Vues. Le Modèle est un objet autonome qui doit pouvoir être réutilisé sur différentes applications possédant chacune un système d'affichage différent.

Ces deux premiers chapitres nous ont permis de définir deux éléments importants pour la réalisation d'une application graphique que sont le Modèle et la Vue. Les chapitres suivants vont présenter différentes architectures permettant de lier le Modèle et la Vue en gardant notre Modèle indépendant.

3

L'architecture Modèle-Vue-Contrôleur

3.1 Introduction

Introduite à la fin des années 1970 l'architecture Modèle-Vue-Contrôleur s'est vite retrouvée populaire pour les applications web. En effet, il met l'accent sur la séparation entre la logique métier et l'affichage du logiciel.

3.2 L'architecture MVC

3.2.1 Les composants

Les deux chapitres précédents ont introduit la notion de Modèle et de Vue où nous avons conclu que le Modèle est un objet indépendant qui n'a pas pour rôle de choisir qu'elle Vue doit être affiché.

Le troisième composant de notre architecture MVC est donc le Contrôleur. Ce dernier élément doit être considéré comme le relai entre l'action utilisateur, son affichage sur la Vue et les modifications à apporter dans le Modèle. En effet, il aura pour rôle de modifier le Modèle suivant l'action utilisateur et de choisir la Vue adéquate en retour. En d'autres termes, à chaque action il doit déterminer quoi faire (modifier le Modèle, mettre à jour la Vue, changer de Vue, etc ...).

Si nous reprenons l'exemple précédent, nous devons sortir de méthode `notifier` les actions de changement de vues.

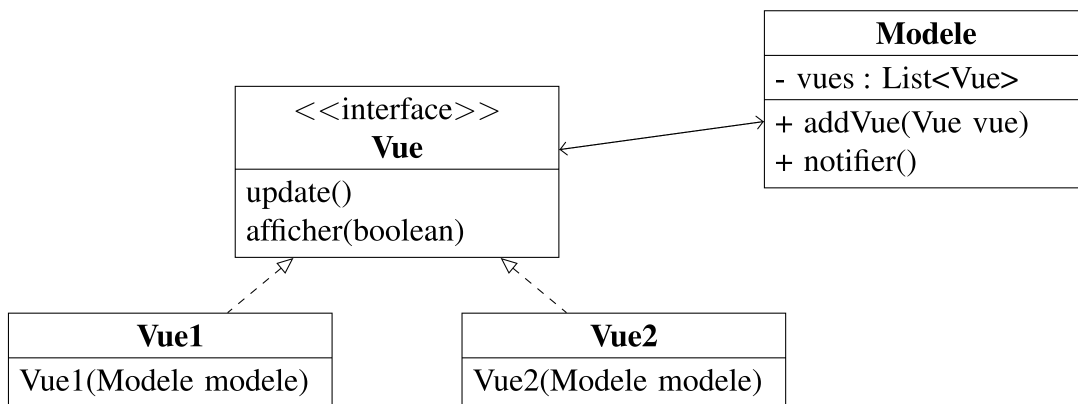
- Les Vues sont les éléments graphiques
- Le Modèle est notre objet indépendant qui se voit modifier par l'utilisateur
- Le Contrôleur est en charge de choisir la Vue à afficher. Il devient le point d'entrée de notre application.

3.3 Construire l'exemple via l'architecture MVC

Pour mieux comprendre comment l'architecture MVC va améliorer notre architecture nous allons nous appuyer sur l'exemple développé jusqu'à présent.

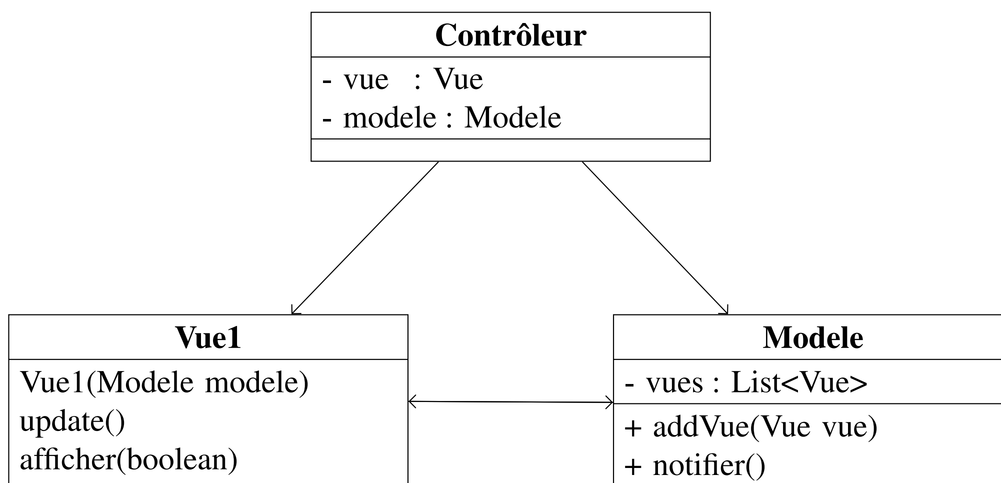
3.3.1 Conserver la relation Vue/Modèle

Pour accomplir cet exercice, nous devons conserver la relation Vue/Modèle. Elle permet à une vue de s'abonner à un modèle. Ainsi quand le modèle sera modifié (par la vue au travers du contrôleur) nous continuerons à interroger la modèle depuis notre vue pour récupérer les informations.



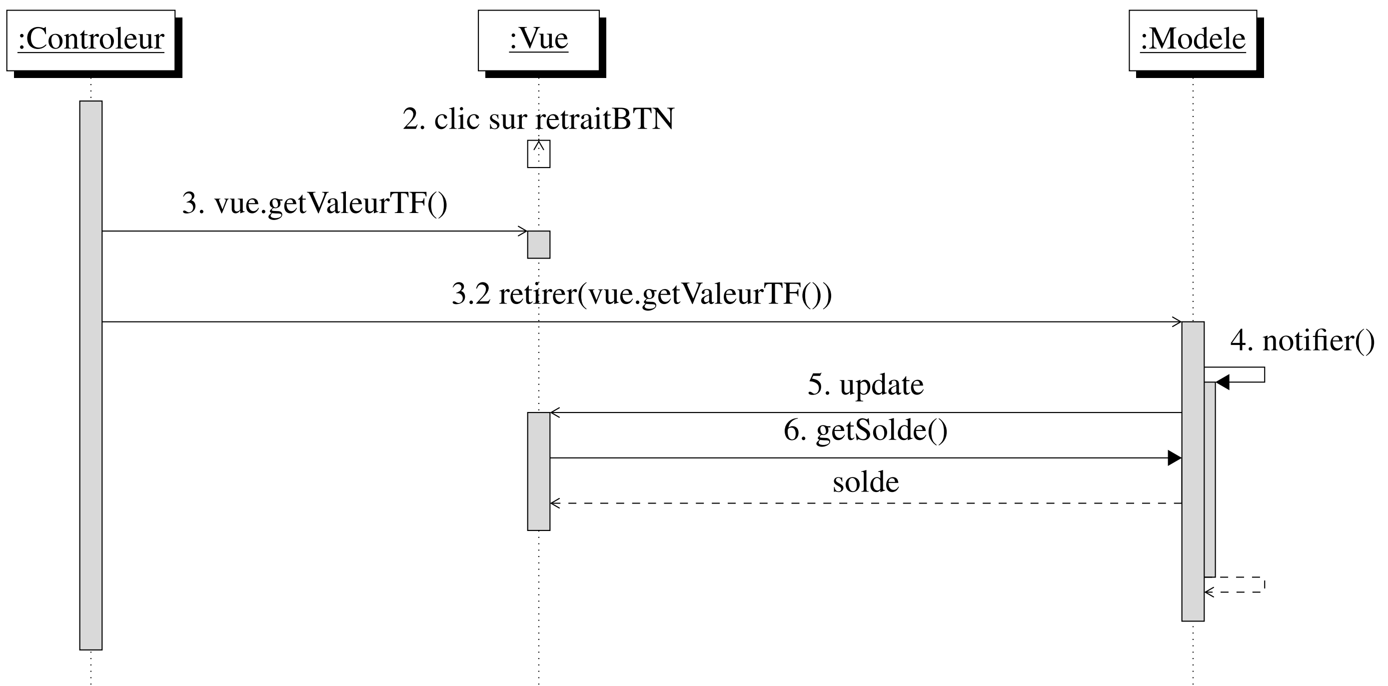
3.3.2 Le Contrôleur est le point d'entrée de notre application

Là ou avant le point d'entrée de notre application est la `Vue1` (méthode `main`) maintenant le point d'entrée devient le Contrôleur. Cela implique donc une dépendance de la classe `Contrôleur` vers la classe `Vue`.
De plus il va être en charge de modifier le Modèle suivant les actions de l'utilisateur. Nous avons donc en plus une relation entre le Contrôleur de la classe `Contrôleur` vers la classe `Modèle`.



Les enchainements de méthodes/actions avec cette nouvelle architecture peuvent sembler difficiles. Nous allons tenter de les détailler avec le diagramme de séquence ci-dessous.

1. L'utilisateur exécute la méthode `main`
   — Un objet `modèle` est crée
   — Les objets `vue` sont créés en passant en paramètre le `modèle`
   — A l'objet `modèle` nous ajoutons les `vue` comme étant abonnées du `modèle`
   — Nous créons un objet `controleur` avec les `vue` et le `modèle`
2. L'utilisateur appuie sur le bouton `retirerBTN`
3. Le `contrôleur` détecte un évènement sur ce bouton
4. Le `contrôleur` récupère les valeurs auprès de la `vue` et les envoies au `modèle`
5. Le `modèle` met à jour ces attributs et déclenche la méthode `notifier()`
6. La méthode `notifier()` appelle la méthode `update()` sur chaque `vue`
7. Les `vue` récupère les nouvelles valeurs auprès du `modèle`




```
public class Controleur {

   Modele modele;
   Vue vue1;
   Vue vue2;

   public Controleur(Modele modele, Vue vue1, Vue vue2) {
      this.modele = modele;
      this.vue1 = vue1;
      this.vue2 = vue2;

      this.vue1.afficher(true);
      this.vue2.afficher(false);
   }

   public static void main(String[] args) {
      Modele modele = new Modele();
      Vue vue1 = new Vue1(modele);
      Vue vue2 = new Vue2(modele);
      modele.addVue(vue1);
      modele.addVue(vue2);

      Controleur c = new Controleur(modele, vue1, vue2);
   }
}
```

Il nous reste à réaliser les étapes 3) et 4) de notre diagramme de séquence.

3.3.3 La communication Vue/Contrôleur : détecter les évènements

Dans les deux premiers chapitres la Vue envoyée directement ses requêtes au modèle. Par exemple, lorsqu'on clique sur le bouton *retirer* ont déclenché l'action `modele.retirer(n)`. Le Contrô-

leur va écouter les évènements qui se déroulent sur les vues. Ainsi, nous déplaçons les *listeners* des boutons dans le Contrôleur.

```
public Controleur(Modele modele, Vue vue1, Vue vue2) {
   ...

   ((Vue1) vue1).retraitBTN.addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
         modele.retirer(Integer.parseInt(((Vue1)
            vue1).valueTF.getText()));
      }
   });
}
```

3.3.4 La communication Vue/Contrôleur : une alternative

Nous pouvons implémenter la communication Vue/Contrôleur en implémentent l'interface `ActionListener` dans notre `Contrôleur` et spécifiant dans le `main` quel où se trouve le *listener* du bouton.

```
public class Controleur implements ActionListener {
   public Controleur(Modele modele, Vue vue1, Vue vue2) {
      /* on supprime le addActionListener qu'on déplace dans
         actionPerformed */
   }

   @Override
   public void actionPerformed(ActionEvent e) {
      System.out.println(e.getSource());
      if (e.getSource() == ((Vue1) vue1).retraitBTN) {
         modele.retirer(Integer.parseInt(((Vue1)
            vue1).valueTF.getText()));

      } else if( ... ) {

      }
   }

   public static void main(String[] args) {
      ...
      Controleur c = new Controleur(modele, vue1, vue2);
      ((Vue1) vue1).retraitBTN.addActionListener(c);
   }
```

Quelle que soit la solution retenue, notre architecture ainsi que l'enchainement de nos actions ne sont pas modifiés.

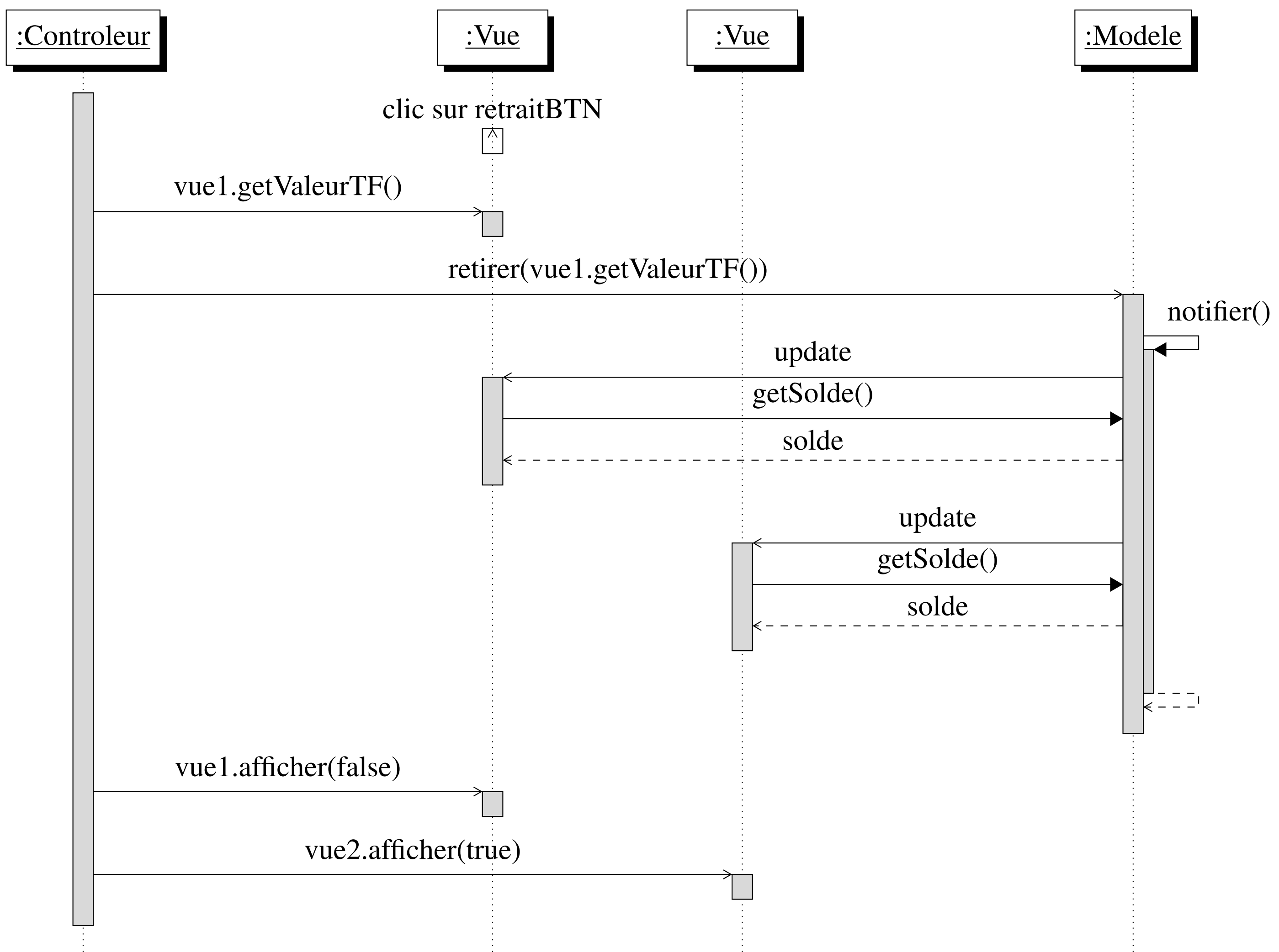
3.3.5 Sortir le choix de la vue du Modèle

Le choix de notre `vue` à afficher après l'action de l'utilisateur est toujours dans le Modèle. Or, cette action doit maintenant être réalisée par le Contrôleur.
Nous supprimons les actions `vues.get(n).afficher(b)` de la méthode `notifier` et les mettons dans le *listener*.

```
public class Controleur implements ActionListener {
   ...
   @Override
   public void actionPerformed(ActionEvent e) {
      System.out.println(e.getSource());
      if (e.getSource() == ((Vue1) vue1).retraitBTN) {
         modele.retirer(Integer.parseInt(((Vue1)
            vue1).valueTF.getText()));

         vue1.afficher(false);
         vue2.afficher(true);
      }
   }
}
```

3.3.6 Diagramme de séquence final

Le diagramme de séquence final est donc le suivant. Nous y avons ajouté la `vue2` ainsi que leur affichage par le `controleur`.



3.3.7 Résumé de la mise en place du MVC

- Nous sommes partis de l'architecture Vue/Modèle qui permet à chaque vue de s'abonner à un modèle et au modèle de notifier les vues à chaque changement.
- Cependant, le rôle du Modèle n'est pas de choisir la vue à afficher.
- Pour ce faire, nous avons rajouté une entité Contrôleur qui
  - devient le point d'entrée de notre application.
  - écoute les évènements sur la vue (clic sur un bouton, etc) pour en informer le modèle
  - une fois l'évènement pris en compte elle choisie la vue à afficher en retour.

- Le Modèle ne connaît ni la vue, ni le Contrôleur. Il est donc indépendant.
- La Vue connaît le Modèle.
- Le Contrôleur connaît la vue et le Modèle.

Notre application peut se résumer par le diagramme suivant. Nous reviendrons sur la notion *Observer-Observable* dans les prochaines sections du chapitre.

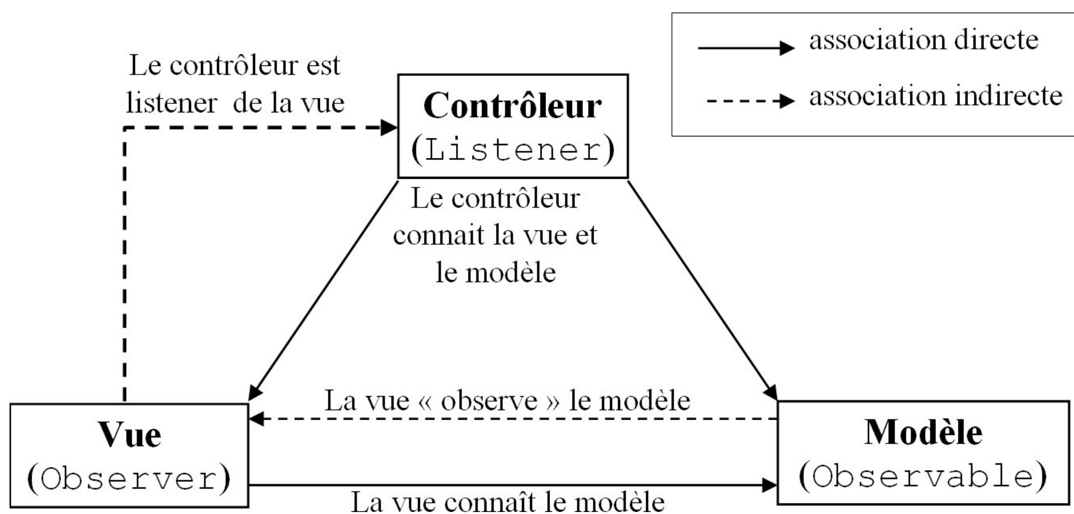


FIGURE 3.1 – Schéma récapitulatif du patron MVC [1]

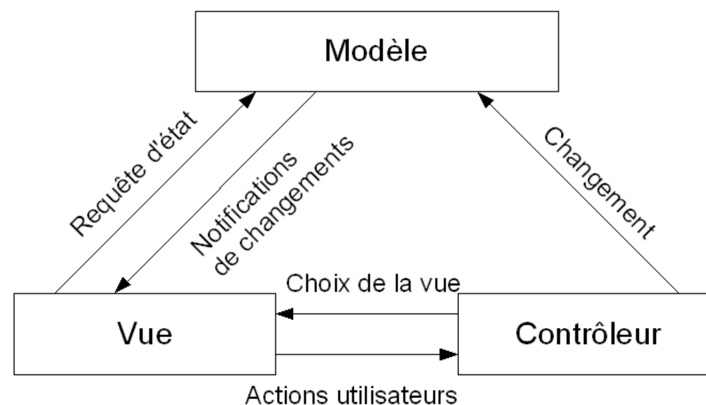


FIGURE 3.2 – Schéma récapitulatif du patron MVC termes différents

3.3.8 Questions soulevées

— Est-ce que le Modèle doit connaitre directement les Vues ?
— Étant donné que le Contrôleur connait la Vue qui connait le Modèle. La relation Contrôleur/-Modèle est-elle nécessaire ?

3.4 Amélioration de l'architecture MVC

La section précédente nous a montré comment construire une application grâce à l'architecture MVC. Nous avons vu que le Contrôleur joue un rôle important pour la maintenabilité et l'évolution de notre application.

À la fin de la section précédente, nous avons également évoqué deux questions auxquelles nous allons maintenant répondre.

3.4.1 Le Modèle doit-il avoir connaissance les Vues ?

La première question à se poser est de savoir si la dépendance entre le Modèle et la Vue doit exister. L'avantage de cette dépendance est de pouvoir notifier la Vue dès que le Modèle est modifié et ce par n'importe quel acteur (Contrôleur ou non).
Mais cette dépendance impose une contrainte majeure. Lorsque nous développons notre Modèle devons/nous être conscients de la Vue qui va être utilisée ? La réponse est non, le modèle n'a pas à se soucier de savoir comment il va être utilité (IHM, console, etc). Le Modèle est une entité indépendante. Ainsi, en supprimant cette dépendance, nous serons libres :

— de pouvoir changer la Vue sans altérer le Modèle
— d'avoir plusieurs Vues pour un seul Modèle

Alors, je vous mens un peu. Nous avons déjà mis en place ces principes dans notre architecture et évoqué les termes de *observable-observer* mais sans nous y attarder.

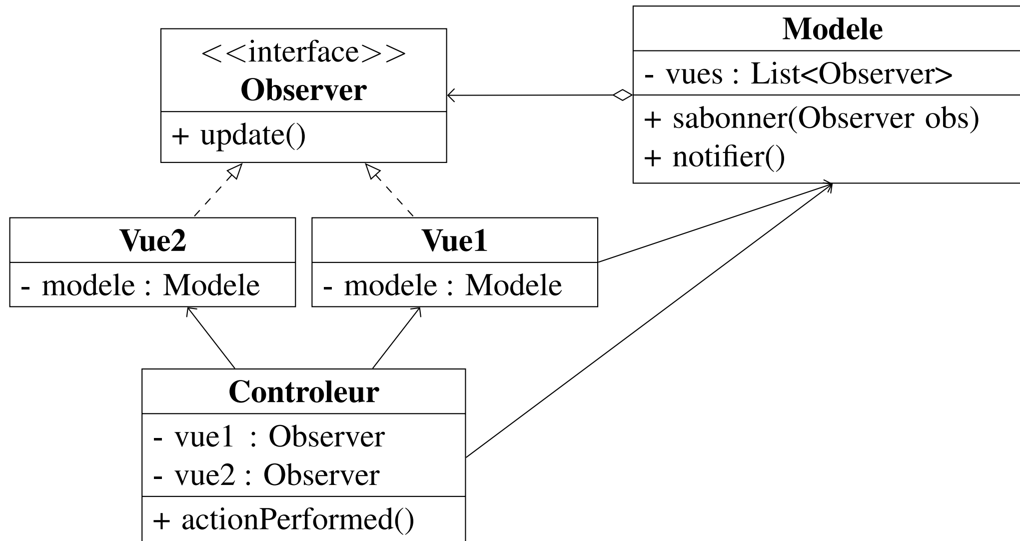
3.4.2 Une interface pour découpler : patron Observateur

Dans une application composée de plusieurs interfaces graphiques qui utiliserait sur certaines un même modèle. Si l'utilisateur fait une modification sur l'une des interfaces, les autres interfaces graphiques doivent être mises au courant. Pour faire cela sans créer de dépendances, nous avons besoin d'implémenter le patron *Observateur* [2].

Grâce à l'interface `Observer` :

— Le Modèle ne dépend plus des Vues, il dépend d'une interface `Observer`. L'interface `Vue` des exemples précédents font office d'observateur.
— Les Vues dépendent toujours du Modèle
— Donc nous avons une relation unidirectionnelle de Vue vers Modèle

3.4.2.1 Implémentation



Ainsi, ce n'est lorsqu'on ensemble les briques de notre application où nous spécifierons pour chaque Vue à quel(s) modèle(s) elle se réfère.

```
main() {
   Modele modele = new Modele
   Vue vue1 = new Vue1();
   Vue vue2 = new Vue2();

   modele.subscribe(vue1); // anciennement .addVue(vue1)
   modele.subscribe(vue2); // anciennement .addVue(vue2)
}
```

Lorsqu'une action modifie le Modèle informe directement les Vues concernées.

```
public void notifier() {
   for(Vue vue : vues) {
      vue.update();
   }
}
```

3.4.3 Relation transitive Contrôleur-Vue-Modèle / Contrôleur-Modèle

La seconde question est sur l'utilité d'avoir une relation entre Contrôleur vers Modèle étant donné que nous pouvons accéder au Modèle grâce à la relation Contrôleur vers Vue vers Modèle.

```
public class Controleur implements ActionListener {
   ...
   @Override
   public void actionPerformed(ActionEvent e) {
      System.out.println(e.getSource());
      if (e.getSource() == ((Vue1) vue1).retraitBTN) {
         modele.retirer(Integer.parseInt(((Vue1)
            vue1).valueTF.getText()));

      }
   }
```

Remplacer le code de l'`actionListener` par le code suivant est-il une bonne idée.

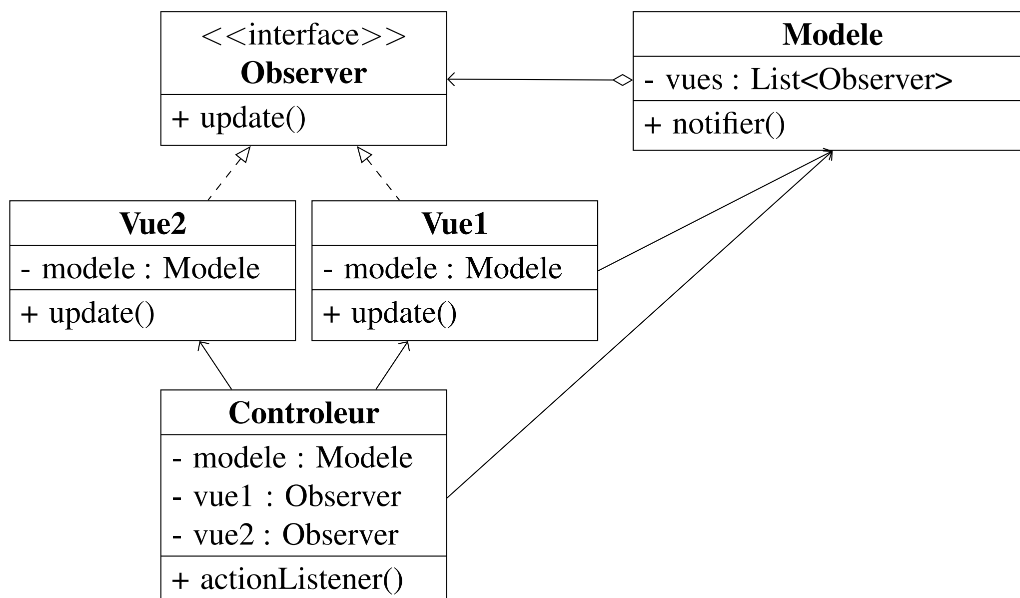
```
vue1.modele.retirer(Integer.parseInt(((Vue1)
   vue1).valueTF.getText()));
```

3.4.3.1 La *Loi de Déméter* et ses impacts

Remplacer le l'`actionListener` par le nouveau code ne respecte pas la *Loi de Déméter*. Une des conséquences de cette transgression est la difficulté à tester notre application. En effet, si nous souhaitons vérifier que l'appel à `retirer()` est correctement réalisé nous devons créer un bouchon de notre Vue et dans ce bouchon créer un autre bouchon pour le Modèle.

En gardant la conception actuelle, seul un bouchon sur notre Modèle est nécessaire lors de la création de tests unitaires.

3.4.4 Résumé de l'architecture MVC



Nous arrivons donc à une architecture satisfaisante où

1. Le Contrôleur joue l'action (`actionPerform()`), le Modèle est donc modifier
2. le Modèle notifie ses changements aux Vues abonnées (patron Observateur)
3. les Vues vont récupérer les informations auprès du Modèle

3.4.5 Conclusion

Nous venons de reprendre les deux architectures présentées dans la section précédente. Nous y avons rajouté un patron `Observateur` afin de ne garder qu'une seule relation unidirectionnelle entre la Vue et le Modèle. Puis nous avons décidé de relier directement notre Contrôleur à notre Modèle.

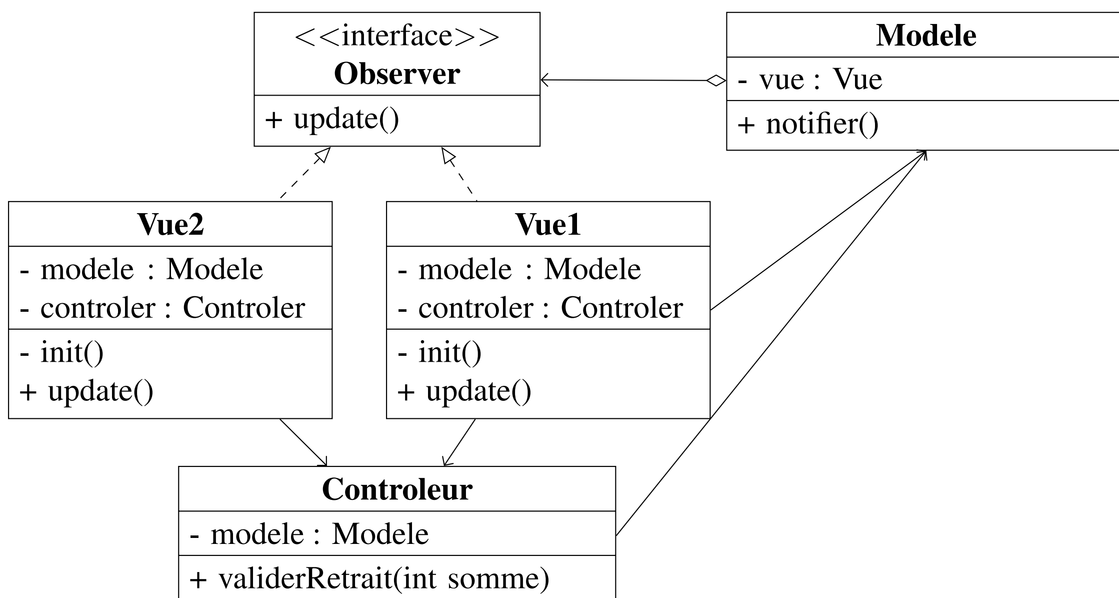
3.5 Variantes de l'architecture précédente

3.5.1 Relation Contrôleur-Vue inversée

Nous terminons notre étude de l'architecture MVC avec une variante de l'architecture développée jusqu'à présent. Dans la partie 3.3.3 nous avions imposé d'avoir l'action de l'utilisateur dans le Contrôleur, mais nous revenons ici sur cette décision. L'argument avancé est qu'une interface graphique (la Vue) ne devrait pas exposer ses composants. Ainsi, le Contrôleur ne devrait pas se soucier de la façon dont certaines actions se produisent, mais juste être au courant qu'elles existent.

Cela implique donc que
- Les évènements `ActionListener` soient maintenus par la Vue.
- La Vue avertit le Contrôleur lorsqu'une action est réalisée.
- Le Modèle reste inchangé. Il notifiera la Vue à chaque changement.

Par conséquent, nous n'avons plus une relation de la classe `Contrôleur` vers la classe `Vue` mais de la `Vue` vers le `Contrôleur`.




```
public Vue {
   private Controler controler;
   private Modele modele;

   public Vue(Modele m, Controler c) {
      ...
      retirerBTN.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            // Appel du Contrôleur lors du clic sur le bouton
            controler.validerRetrait(Integer.parseInt(valueTF.getText()));
         }
      });
   }
}
```

```
public Controler {
   Modele modele;
   ...

   void validerRetrait(int somme) {
      modele.retirer(somme)
   }
}
```

3.5.1.1 Avantage : avoir un Contrôleur réutilisable

Le fait de mettre les évènements dans la Vue permet de pouvoir avoir un Contrôleur réutilisable. Si nous souhaitons développer notre produit pour différentes plateformes, nous serons amenés à utiliser différentes librairies d'interface graphique. Chaque librairie à sa propre façon de gérer les évènements. Nous serons donc amenés à développer la même Vue avec une librairie différente. Si la gestion des évènements (*actionListener*) se fait dans le Contrôleur alors en plus des Vues nous devons développer nos Contrôleurs pour chaque plateforme.

Grâce à cette approche, le Contrôleur est indépendant de la Vue. Il ne se soucie pas de savoir comment les évènements sont déclenchés. Donc nous pouvons réutiliser notre Contrôleur sur toutes les plateformes.

3.5.1.2 Inconvénient : un Contrôleur qui ne choisit pas les Vue à afficher

Un des rôles majeur du Contrôleur est de choisir qu'elle est la Vue à afficher suite à un évènement. Avec cette approche où nous n'avons qu'une relation `Vue` vers `Contrôleur` il nous est impossible de modifier la Vue à afficher. La Vue qui déclenche l'évènement est responsable de cela.

Est-ce un problème ?

La réponse à cette question n'est pas évidente. Nous allons cependant apporter quelques précisions. On va retrouver un problème de scalabilité. Si une Vue *vue1* à plusieurs boutons redirigeant vers d'autres vues, alors la *vue1* va avoir des dépendances vers un nombre important de vues. Chaque vue aurait donc une liste de `vues` vers lesquels les boutons peuvent rediriger. Ceci peut être complexe à maintenir.

On autre problème lié au précédent est la répartition de la gestion d'affichage des vues dans chaque vue. En effet, grâce au Contrôleur notre gestion était centralisée en un point. Maintenant chaque vue ayant une liste de vue nous avons une architecture distribuée.

3.5.1.3 Question : quelle est donc l'utilité du Contrôleur ?

Cette nouvelle approche remet également en question l'utilité du Contrôleur. En effet, pourquoi la Vue n'appellerait tout simplement pas directement le Modèle.

```
public void actionPerformed(ActionEvent e) {
   // Appel du Contrôleur lors du clic sur le bouton
   controler.validerRetrait(Integer.parseInt(valueTF.getText()));
}
```

```
public void actionPerformed(ActionEvent e) {
    // Remplacer par
    modele.retirer(Integer.parseInt(valueTF.getText()));
}
```

Pour comprendre l'utilité du Contrôleur nous devons complexifier notre application. Nous affirmons que pour les retraits de plus de 1000 euros nous ne pouvons pas utiliser la Vue développée jusqu'à présent, mais utiliser une *Vue Sécurisée*.

```
public void actionPerformed(ActionEvent e) {
    // Appel du Contrôleur lors du clic sur le bouton
    controler.validerRetrait(Integer.parseInt(valueTF.getText()));
}
```

```
public void validerRetrait(int somme) {
    if(somme < 1000) {
        modele.retirer(somme);
    } else {
        // afficher la Vue Sécuriée
    }
}
```

Mais vous pouvez me poser la question suivante (surtout que `Vue` connait `VueSecurisee`)

Pourquoi ne pas coder le bloc conditionnelle dans la vue ?

```
public void actionPerformed(ActionEvent e) {
    int valeur = Integer.parseInt(valueTF.getText());
    if(valeur < 1000) {
        controler.validerRetrait(Integer.parseInt(valueTF.getText()));
    } else {
        vueSecurisee.afficher(true); // on affiche la vue sécuriée
        this.afficher(false);
    }
}
```

Cette approche n'est pas bonne car une Vue ne doit pas avoir d'intelligence. Une vue sert juste à afficher les informations qui lui sont envoyées ou qu'elle récupère auprès du Modèle.

- Nous ne pouvons pas reporter cette intelligence dans le Modèle car il n'a pas à choisir qu'elle vue doit être affichée 2.2.5.2.
- Il ne reste que le Contrôleur pour gérer cette intelligence. La solution qui doit être retenue est donc d'avoir le bloc conditionnel dans le Contrôleur. Donc le Contrôleur est utile !

```
public void validerRetrait(int somme) {
    if(somme < 1000) {
        modele.retirer(somme);
    } else {
        // afficher la Vue Sécurisée, mais comment ?
    }
}
```

3.5.1.4 Résumé

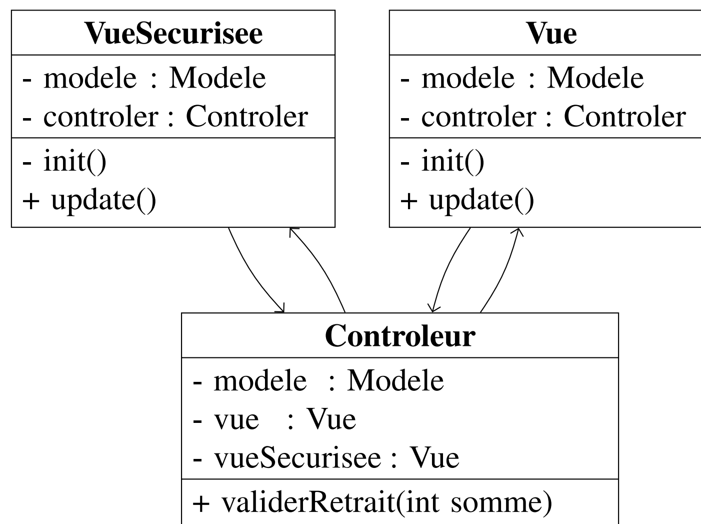
Avant déterminer comment afficher *Vue Sécurisée* depuis le Contrôleur nous allons faire un bref résumé sur l'approche étudiée.

- Nous avons voulu modifier la dépendance Contrôleur/Vue pour l'avoir de la Vue vers le Contrôleur.
- L'avantage de cette approche est d'avoir un Contrôleur réutilisable.
- Mais en contrepartie, le Contrôleur ne choisit plus les Vues à afficher puisqu'il ne les connait pas
- Cependant, une Vue ne peut pas avoir d'intelligence qui doit être déléguée au Contrôleur.
- Donc le Contrôleur doit connaitre les Vues, mais comment ?

3.5.2 Relation bidirectionnelle Contrôleur-Vue

Nous venons de démontrer une nouvelle fois l'utilité du Contrôleur. Cependant, nous devons régler un dernier problème : *Comment le Contrôleur connait les Vues ?*.

La solution est d'avoir en plus de la relation `Vue` vers `Contrôleur` une relation `Contrôleur` vers `Vue`.



3.5.2.1 Inconvénients : une relation bidirectionnelle et non/respect du DI

Cette approche soulève donc le problème des relations cyclique dans un programme qui complexifie premièrement son écriture, mais montre également une mauvaise structure de l'application. Une façon de régler ce problème est de construire les Vues dans le constructeur du `Contrôleur` et de passer le `contrôleur` en paramètre.

```
public Vue(Controleur controleur) { this.controleur = controleur }
```

```
public Controleur() {
   this.vue1 = new Vue(this); // vue normale
   this.vue2 = new Vue(this); // vue sécurisée
}
```

Mais c'est une mauvaise façon de créer nos Vues car nous ne respectons pas le principe d'injection de dépendances [4] (le constructeur de la classe Contrôleur devrait prendre en paramètre les deux Vues). Par conséquent, l'écriture des tests pour notre application est plus difficile.

3.5.2.2 Avantage : un Contrôleur qui connait les Vues

Avec cette nouvelle solution, nous réglons les inconvénients soulevés précédemment 3.5.1.2. Une vue n'aura pas une liste de vue dont elle dépend puisque c'est notre Contrôleur qui gère toute l'intelligence.

3.5.2.3 Le serpent qui se mort la queue

Nous venons de perdre l'objectif initial d'avoir un Contrôleur réutilisable. En effet, maintenant que notre Contrôleur connait les Vues il n'est plus indépendant.

La relation Vue vers Contrôleur est-elle donc utile ? Non

3.5.3 Conclusion

Dans cette partie, nous avons étudié une alternative au patron MVC en deux temps :
— Premièrement, nous avions un Contrôleur indépendant. Le but étant d'avoir un Contrôleur réutilisable, qui ne soucie pas de savoir comment les évènements sont déclenchés. Le gros point faible de cette approche est d'avoir le Contrôleur qui ne peut pas choisir quelle Vue afficher en retour. Donc l'intelligence était dans les Vues.
— Pour pallier à ce problème, nous devons avoir une relation bidirectionnelle pour redonner l'intelligence au Contrôleur.
— Mais dans ce cas, nous ne répondons plus à l'objectif d'avoir un Contrôleur réutilisable.

Nous avons essayé de développer une alternative, mais nous retombons sur l'architecture initiale. Cependant, grâce à cette section nous avons *compris* le rôle de chaque dépendance et nous sommes encore plus rentrés en détail dans la compréhension du patron MVC.

Conclusion

Ce document nous a permis d'étudier une première architecture pour la conception l'application graphique. L'architecture MVC se décompose en trois blocs :

- la Vue qui contient seulement nos éléments graphiques
- le Modèle qui représente la logique métier de notre application
- le Contrôleur qui permet de faire la liaison entre les actions de l'utilisateur et la logique métier

Une fois les éléments de l'architecture définis, nous avons abordé les relations entre ces trois blocs

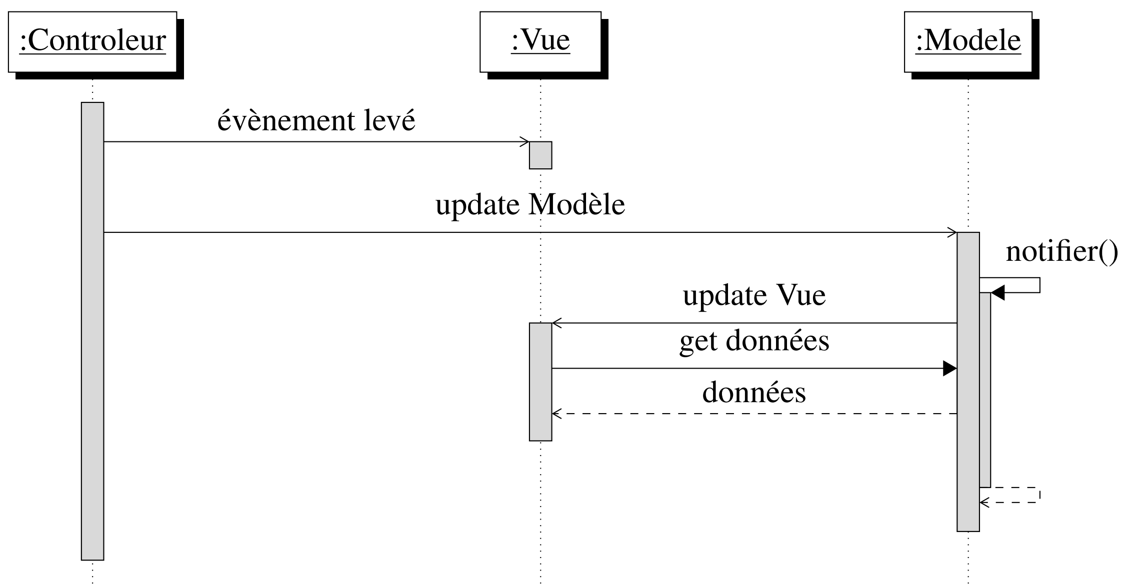
- le Modèle notifie la Vue de ses changements grâce au patron *Observateur* [2]
- la Vue récupère les changements auprès du Modèle
- ⇒ la Vue est donc dépendante du Modèle, mais le Modèle n'a de pas dépendance vers la Vue

Enfin, pour finir l'explication du patron MVC nous avons essayé de développer une alternative. La relation `Vue` vers `Contrôleur` nécessite également la relation `Contrôleur` vers `Vue`. Donc nous n'avons pas besoin de la relation `Vue` vers `Contrôleur`.

Compléments

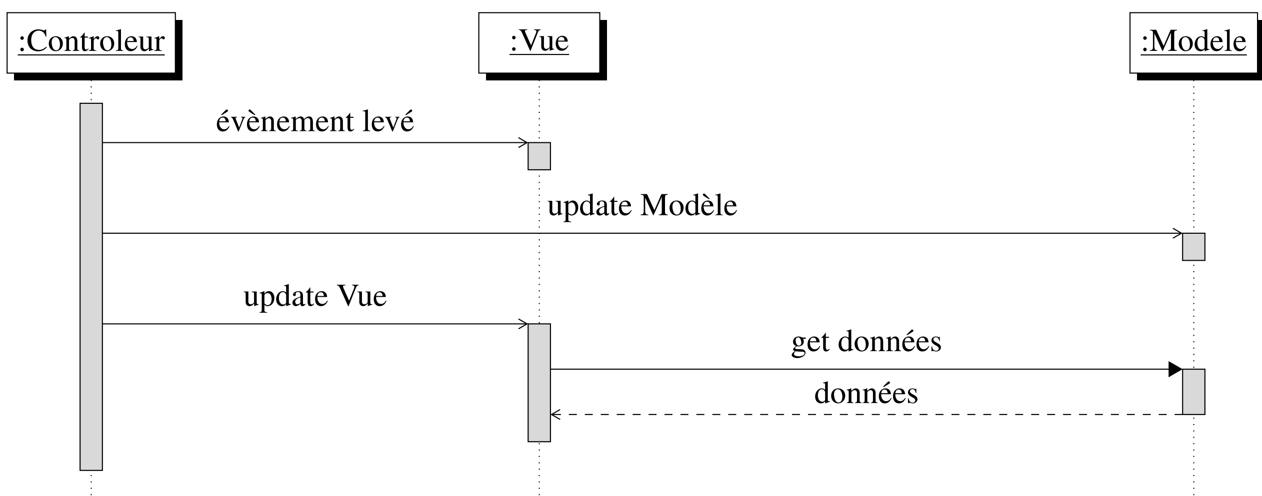
MVC avec Modèle Actif

C'est l'architecture à laquelle nous venons d'aboutir à la fin de ce chapitre Le Contrôleur n'est pas la seule classe qui modifie le Modèle, le Modèle a besoin d'un moyen de notifier la Vue et les autres classes des mises à jour. Ceci est réalisé avec l'aide du pattern Observateur. Le Modèle contient une collection d'Observateurs qui sont intéressés par les mises à jour. La vue implémente l'interface de l'observateur et s'enregistre comme observateur du Modèle.



MVC avec Modèle Passif

Une autre version sur patron MVC existe. Dans la version Modèle Passif, le Contrôleur est la seule classe qui manipule le Modèle. En fonction des actions de l'utilisateur, le Contrôleur doit modifier le modèle. Une fois le Modèle mis à jour, le Contrôleur notifie à la Vue qu'elle doit également être mise à jour. À ce moment-là, la vue demandera les données du modèle.



Bibliographie

[1] Télécom ParisTech 2012 IRÈNE CHARON. *Une conception sur le moèle MVC (Model-View-Controller)*. 2012. URL : https://perso.telecom-paristech.fr/hudry/coursJava/interSwing/boutons5.html#addObs.

[2] Erich GAMMA et al. *Design patterns elements of Reusable Object Oriented Software*. Addison Wesley, 1998.

[3] Robert C. MARTIN. *Clean architecture : A craftsman's guide to software structure and Design*. Prentice Hall, 2018.

[4] WIKIPÉDIA. *Injection de dépendances*. 2014. URL : https://fr.wikipedia.org/w/index.php?title=Injection_de_d%C3%A9pendances&oldid=104738700.