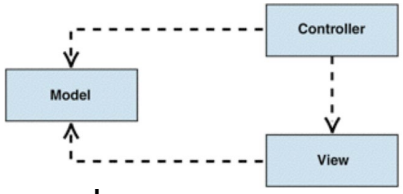
Modalité de l'examen : Projet

1. Introduction Java
2. Introduction IHM: Swing(composants, conteneurs, layouts),[réflexion projet]
3. Les évènements, [Cahier des charges]
4. **MVC, [Projet]**
5. Boites de dialogues, [Projet]
6. Composants complexes, [Projet]
7. Présentations

Internet

- https://www.oracle.com/technical-resources/articles/java/java-se-app-design-with-mvc.html

Modèle – Vue – Contrôleur



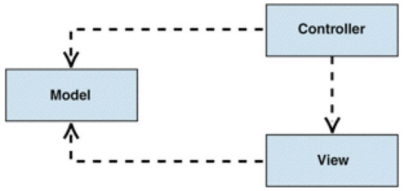
MVC est un design pattern (modèle d'architecture logicielle) qui permet de séparer les responsabilités d'une application en trois couches distinctes :

1. **Modèle (Model):**
   - Gère les données et la logique métier
   - Ne s'occupe ni de l'affichage ni des interactions utilisateur
   - Exemples : base de données, règles de gestion, calculs

2. **Vue (View)**
   - Gère l'interface utilisateur
   - Affiche les données fournies par le modèle
   - Réagit aux actions de l'utilisateur (clics, saisies)

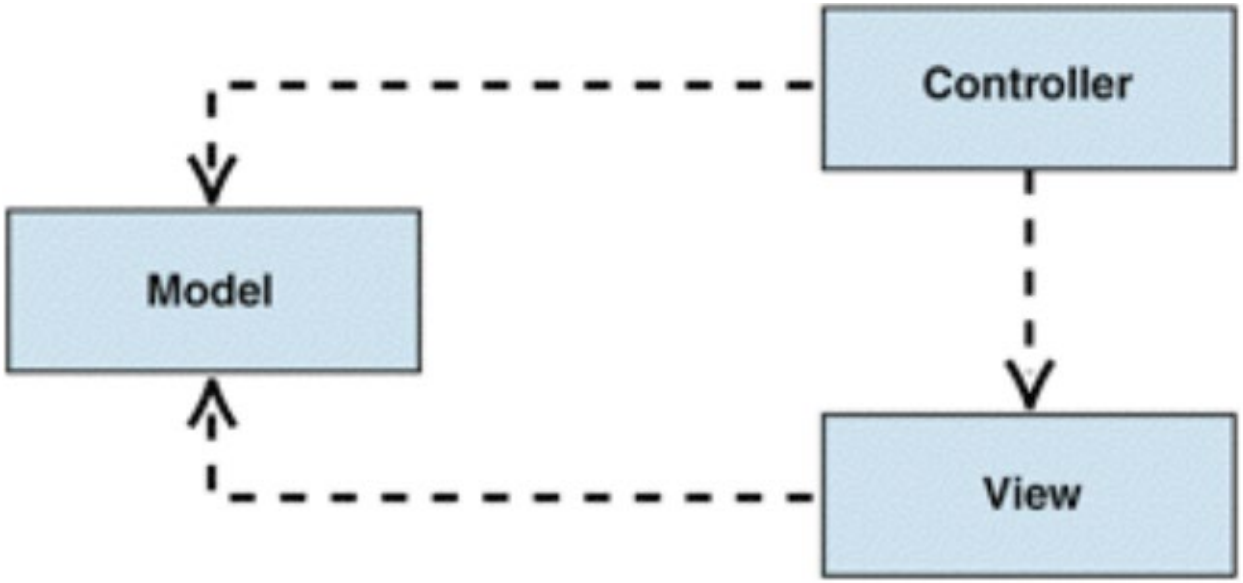
3. **Contrôleur (Controller)**
   - Sert de pont entre la Vue et le Modèle
   - Réagit aux actions de l'utilisateur
   - Met à jour le modèle et la vue en conséquence

Modèle – Vue – Contrôleur



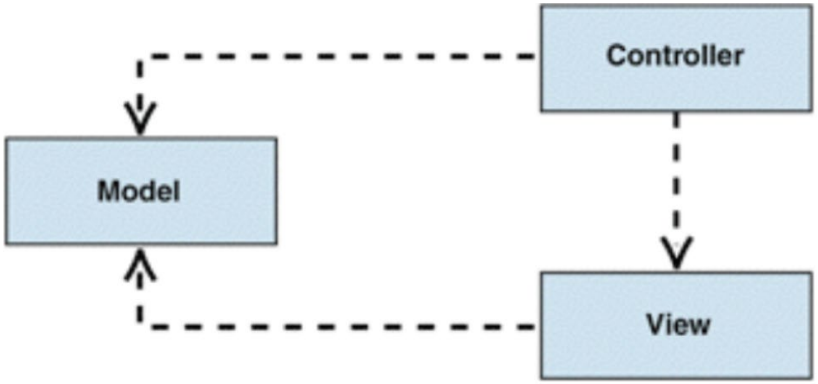
- Séparer les responsabilités pour :
- Faciliter la maintenance du code
- Permettre une réutilisation des composants
- Favoriser les tests unitaires
- Faciliter le changement de l'interface sans toucher à la logique

MVC



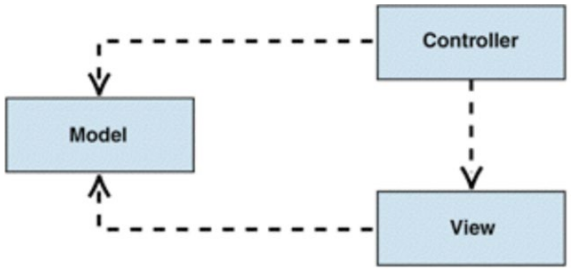
- Comment obtenir un bon découplage entre la partie métier et l'interface
  - le motif conceptuel MVC
    - Modèle / Vue / Contrôleur

MVC passif



- Le modèle est conçu indépendamment de l'interface
- Le contrôleur reçoit des événements et modifie en conséquence le modèle, puis prévient les vues de se remettre à jour
- La vue, sur réquisition du contrôleur, récupère les données intéressantes du modèle et les présentent à l'interface
- **c'est le contrôleur qui la pousse à se mettre à jour.**

MVC passif



Exemple:

```
class Model {
    private int value;
    public Model(int value) { this.value = value; }
    public int getValeur() { return value; }
    public void setValeur(int value) { this.value = value; }
}
```

```
interface View {
    void update(Model m);
}

class View1 extends JLabel implements View {
    public View1() {
        super("Value=0");
    }

    public void update(Model m) {
        setText("Value=" + m.getValeur());
    }
}
```

le update - interface

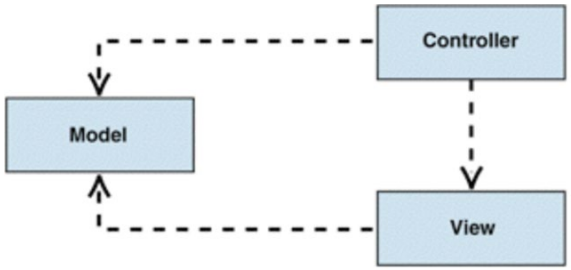
```
class Controller implements ActionListener {
    private Model model;
    private List<View> views = new ArrayList<>();

    public Controller(Model model) {
        this.model = model;
    }

    public void addView(View v) {
        if (!views.contains(v)) {
            views.add(v);
            v.update(model);
        }
    }

    public void actionPerformed(ActionEvent e) {
        model.setValeur(model.getValeur() + 1);
        for (View v : views) {
            v.update(model);
        }
    }
}
```

on appelle l'update

en enclenchant, on va changer la valeur du model et faire un update sur les vues

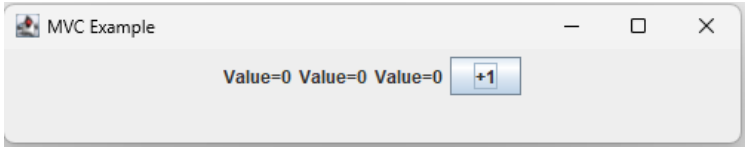
C'est pas le top, ça marche, mais on peut faire mieux. ce n'est pas automatique

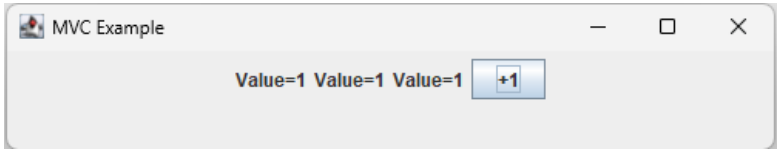
MVC passif



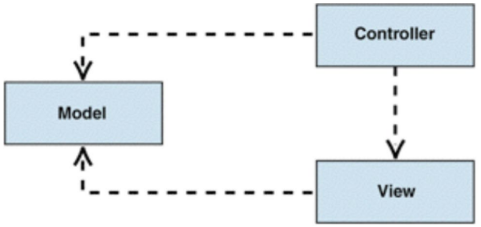
Exemple:

```
public class PassiveMVC implements Runnable {
    public void run() {
        Model model = new Model(0);
        JFrame frame = new JFrame("MVC Example");
        View1 view1 = new View1();
        View1 view2 = new View1();
        View1 view3 = new View1();
        JButton incrementButton = new JButton("+1");

        Controller controller = new Controller(model);
        controller.addView(view1);
        controller.addView(view2);
        controller.addView(view3);
        incrementButton.addActionListener(controller);

        frame.setLayout(new FlowLayout());
        frame.add(view1);
        frame.add(view2);
        frame.add(view3);
        frame.add(incrementButton);

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }
```





MVC passif



On remarque que :

- **c'est le contrôleur qui pousse la vue à se mettre à jour.**
- **La responsabilité de la vue est passive.**

MVC actif

- Le modèle peut être mis à jour de manière autonome (par un timer, un thread, une API, etc.)
- Il doit alors notifier automatiquement toutes les vues concernées.
- On utilise alors le pattern **Observer/Observable**

Pattern Observable/Observer

- **Les rôles:**
  - Sujet (Observable) : source de données, émet des notifications.
  - Observateurs (Observers) : reçoivent les notifications, réagissent.

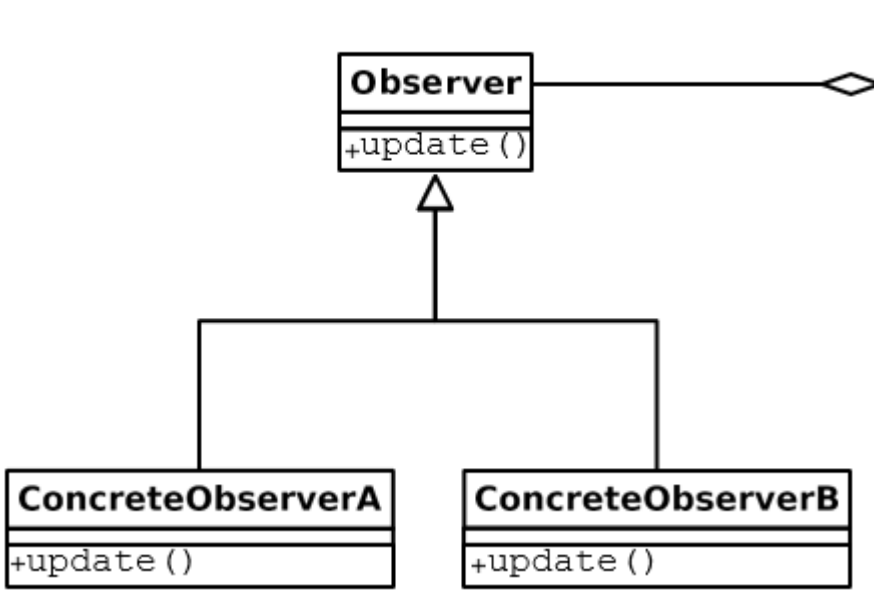
on ne les voit pas, c'est caché, intrinsec

- **Le Sujet:**

par exemple la température

  - Contient un état (valeurs, objets, etc.)
  - Peut changer d'état à tout moment
  - Notifie automatiquement tous les observateurs inscrits
  - Fournit le nouvel état aux observateurs (ex : via getValue())
- **Les Observateurs:**
  - Peuvent s'inscrire ou se désinscrire du sujet
  - Sont notifiés automatiquement lorsqu'un changement a lieu
  - Peuvent consulter l'état actuel du sujet pour se mettre à jour

Pattern Observable/Observer



il faut déclarer les observers chez les sujets

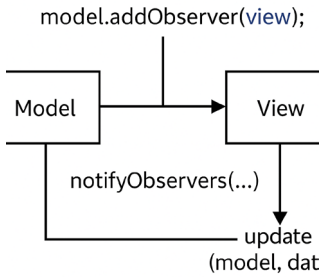
```
for (Observer o : observerCollection) {
      o.update(this);
}
```

un sujet peut être observé par plusieurs observers

Pattern Observable/Observer

```
import java.util.Observable;

public class Model extends Observable {
   private int value = 0;

   public int getValue() {
      return value;
   }

   public void increment() {
      value++;
      setChanged();
      notifyObservers(value);
   }
}
```

il doit envoyer des notifications à ceux qui l'observent



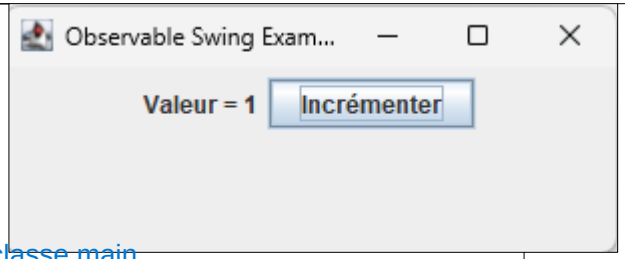

```
import javax.swing.*;
import java.util.Observer;
import java.util.Observable;

public class View extends JLabel implements Observer {
   public View() {
      super("Valeur = 0");
   }

   @Override
   public void update(Observable o, Object arg) {
      setText("Valeur = " + arg);
   }
}
```

C'est là que l'observer reçoit la notife

```
public class MainApp {
   public static void main(String[] args) {
      SwingUtilities.invokeLater(() -> {
         Model model = new Model();
         View view = new View();
         model.addObserver(view);

         JButton button = new JButton("Incrémenter");
         button.addActionListener(e -> model.increment());

         JFrame frame = new JFrame("Observable Swing Example");
         ........
      });
   }
}
```

on doit ajouter l'observer (vue) au modèle dans la classe main

on aimerait que le label (observer) observe le modèle

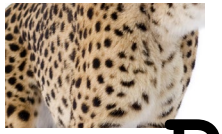


il y a des classes plus puissantes, c'est juste pour comprendre

Pattern Observable/Observer

limité

1. Observable est une classe, pas une interface → empêche l'héritage multiple
2. API rigide et peu flexible :
   1. pas de support pour les noms de propriétés
   2. impossible de filtrer les notifications
3. Notification peu typée : le paramètre Object est générique, pas sécurisé
4. Couplage fort entre modèle et observateurs
5. Déprécié depuis Java 9 → ne doit plus être utilisé dans du code moderne à nos risques et périls
6. Non intégré avec les composants Swing (JComponent, JTable, etc.)



PropertyChangeListener

interface

The “Observable”

Model

- PropertyChangeSupport p

+ addPropertyChangeListener(...)
+ removePropertyChangeListener(...)
...

PropertyChangeSupport

+ addPropertyChangeListener(...)
+ removePropertyChangeListener(...)
+ fireIndexedPropertyChange(...)
...

EventObject

source: Object

+ Object getSource()

PropertyChangeEvent

+ PropertyChangeEvent(Object source,
String propertyName,
Object oldValue, Object
newValue)
+ Object getNewValue()
+ Object getOldValue()
+ String getPropertyName()

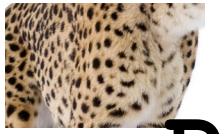
Notify changes

c'est plus clair et sophistiqué, car il donne plus d'information

The “Observer”

PropertyChangeListener

+propertyChange(PropertyChangeEvent)



PropertyChangeListener

The “Observable”

Model

- PropertyChangeSupport p

+ addPropertyChangeListener(...)
+ removePropertyChangeListener(...)
...

PropertyChangeSupport

+ addPropertyChangeListener(...)
+ removePropertyChangeListener(...)
+ fireIndexedPropertyChange(...)
...

Gère la liste des écouteurs et envoie les notifications

EventObject

source: Object

+ Object getSource()

Notify changes

PropertyChangeEvent

+ PropertyChangeEvent(Object source,
String propertyName,
Object oldValue, Object
newValue)

+ Object getNewValue()
+ Object getOldValue()
+ String getPropertyName()

Objet envoyé à chaque listener, contenant :le nom de la propriété,l’ancienne et la nouvelle valeur la source

The “Observer”

Interface que les vues implémentent pour recevoir les événements

PropertyChangeListener

+propertyChange(PropertyChangeEvent)

PropertyChangeListener

Flux logique

PropertyChangeSupport crée un **PropertyChangeEvent**

1. Une propriété change dans le modèle :

```
support.firePropertyChange("valeur", 4, 5);
```

cette firePropertyChange valeur, old, new value

2. PropertyChangeSupport crée un PropertyChangeEvent

3. Il envoie cet événement à chaque objet qui a fait :

```
model.addPropertyChangeListener(listener);
```

il va envoyer l'évènement à chaque objet qu'on a ajouté

4. La vue (ou autre composant) reçoit l'événement via :

```
propertyChange(PropertyChangeEvent evt);
```

mettre à jour la nouvelle valeur

5. La vue peut faire

```
label.setText("Valeur = " + evt.getNewValue());
```

Exemple

(via model.increment())

Model

```
public class Model {
   private int value = 0;
   private PropertyChangeSupport support = new PropertyChangeSupport(this);
   il a créé une classe de support
   public int getValue() {
      return value;
   }

   public void increment() {
      int old = value;
      value++;
      support.firePropertyChange("value", old, value);
   }
   Il donne les infos dans l'ancienne valeur et de la nouvelle

   public void addPropertyChangeListener(PropertyChangeListener pcl) {
      support.addPropertyChangeListener(pcl);
   }
}
   ce modèle est capable d'ajouter des listener sur lui
```

Notification via PropertyChangeListener

Controller

```
public class Controller implements ActionListener {
   private Model model;
                                   controller a actionListener

   public Controller(Model model) {
      this.model = model;
   }

   @Override
   public void actionPerformed(ActionEvent e) {
      model.increment();
   }
}
```

Action de l'utilisateur

View

```
public class View extends JLabel implements PropertyChangeListener {
                                   il implémente PropertyChange, c'est la seule.
   public View() {
      super("Valeur = 0");
   }

   @Override
   public void propertyChange(PropertyChangeEvent evt) {
      setText("Valeur = " + evt.getNewValue());
   }
            il prend simplement la nouvelle valeur
}
```

MVC actif
Fonctionnement général

1. Le contrôleur reçoit un événement utilisateur (ex. : clic sur un bouton).

2. Il appelle une méthode du modèle pour modifier son état (ex. : setValeur()).

3. Le modèle met à jour ses données et notifie automatiquement les vues enregistrées via PropertyChangeListener.

4. Chaque vue, lorsqu'elle est notifiée (propertyChange()), récupère les données auprès du modèle (getValeur()) et met à jour son affichage.

PropertyChangeSupport

- Classe utilitaire pour la gestion des notifications de changement de propriété
- Permet à un objet (souvent un modèle) :
  - d'enregistrer des observateurs (écouteurs de propriétés)
  - de notifier automatiquement les changements de ses attributs
- Le modèle doit contenir une instance de PropertyChangeSupport

Enregistrement / désenregistrement d'un observateur :

```
addPropertyChangeListener(String propertyName, PropertyChangeListener listener)
removePropertyChangeListener(PropertyChangeListener listener)
```

Notification d'un changement de propriété :

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
firePropertyChange(PropertyChangeEvent evt)
```

PropertyChangeSupport

- Classe utilitaire pour la gestion des notifications de changement de propriété
- Permet à un objet (souvent un modèle) :
  - d'enregistrer des observateurs (écouteurs de propriétés)
  - de notifier automatiquement les changements de ses attributs
- Le modèle doit contenir une instance de PropertyChangeSupport

Enregistrement / désenregistrement d'un observateur :

```
addPropertyChangeListener(String propertyName, PropertyChangeListener listener)
removePropertyChangeListener(PropertyChangeListener listener)
```

Notification d'un changement de propriété :

```
firePropertyChange(String propertyName, Object oldValue, Object newValue)
firePropertyChange(PropertyChangeEvent evt)
```

PropertyChangeListener

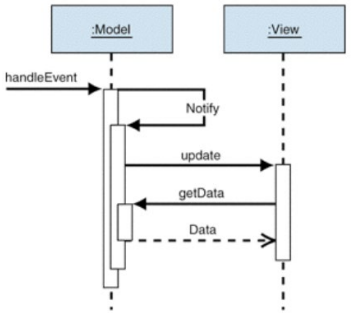
- Interface à implémenter pour écouter les changements de propriétés (l'« observateur »)
- Cette interface permet de recevoir les notifications de changement d'état sur un objet observé.
- Elle ne contient qu'une méthode unique obligatoire à implémenter :
  - void propertyChange(PropertyChangeEvent evt);

Objet transmis à l'observateur pour décrire ce qui a changé. Hérite de EventObject

Donne accès à :

getSource() : l'objet qui a déclenché le changement
getPropertyName() : le nom de la propriété modifiée
getOldValue() et getNewValue()

MVC actif
Fonctionnement général



1. Le modèle peut évoluer sans action directe de l'utilisateur (minuterie, capteur, API...).
2. Pour que les vues restent à jour automatiquement, on applique le pattern d'observation :
3. la vue s'enregistre comme observateur du modèle (via PropertyChangeListener), et reçoit une notification dès qu'un changement a lieu.

PropertyChangeListener

1. PropertyChangeListener est une interface Java qui permet à un objet d'écouter les changements d'état (propriétés) d'un autre objet.
2. C'est une implémentation moderne du pattern Observer, très utilisée en Swing et avec les JavaBeans..

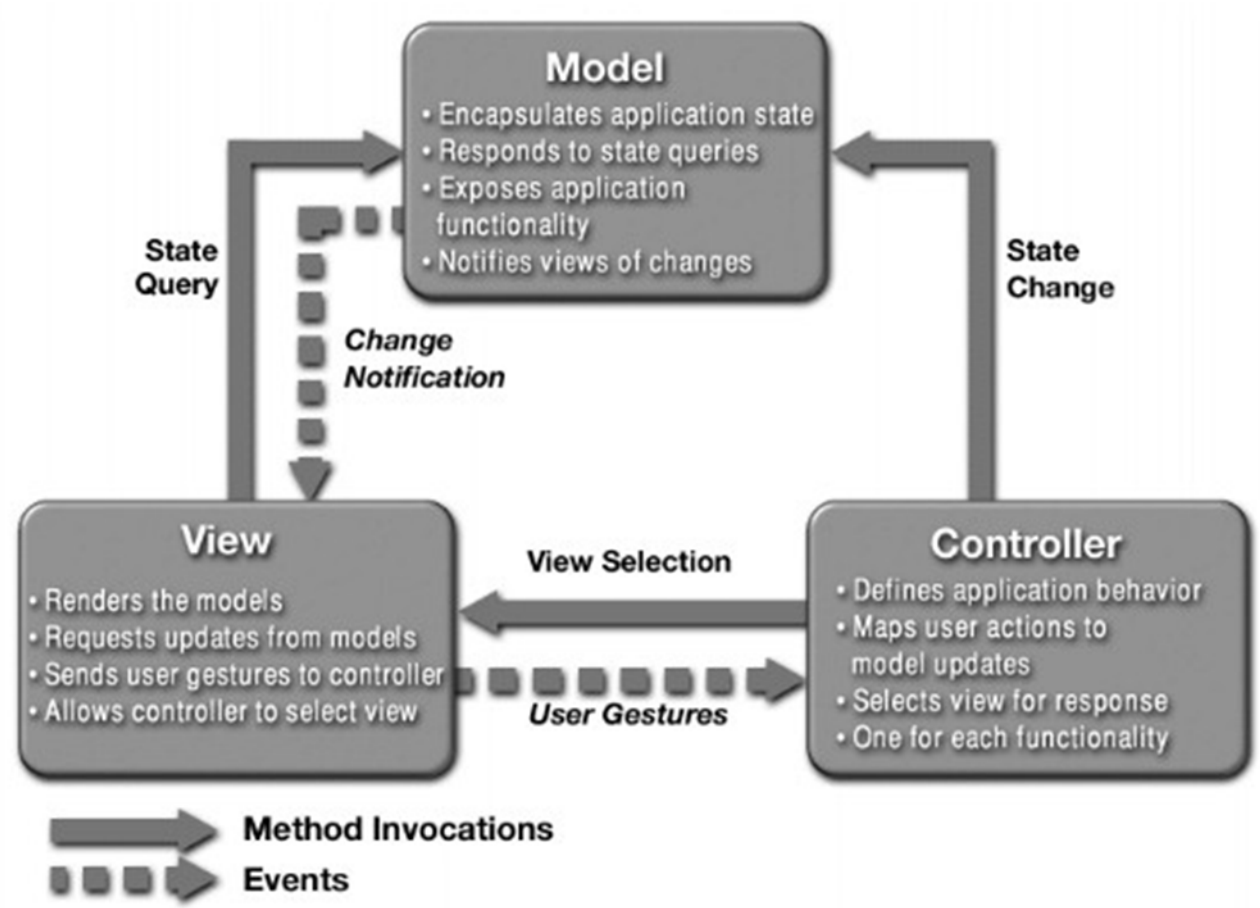
PropertyChangeSupport

1. C'est une classe utilitaire de Java qui gère une liste d'observateurs (PropertyChangeListener) et les notifie automatiquement lorsqu'une propriété change dans un objet.
2. Elle permet à un objet Java (ex. un modèle) de :

    1. Ajouter et retirer des observateurs (vues, logs, etc.)

    2. Déclencher une notification lorsque ses données changent

MVC Resumé


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

MVC exemple (JSlider)

Quelles sont les données associées à un *slider* ?

- *Modèle* :
  - valeur minimale = 0
  - valeur courante = 15
  - valeur maximale = 100

- *Vue* :

- *Contrôleur* :
  - Traiter les clics de souris sur les boutons terminaux
  - Gérer les *drags* de souris sur l'ascenseur