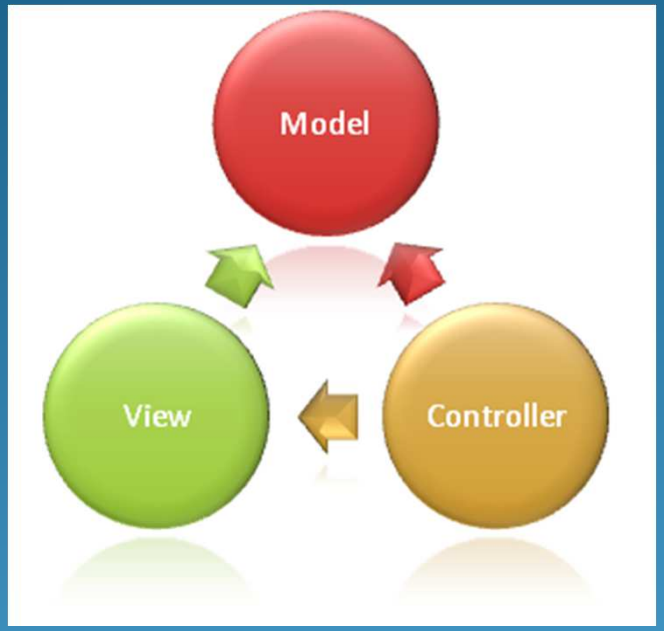


SLAM 4 :

Design pattern MVC Obserser/Observable

Anthony Médassi

Design pattern : c'est quoi?

- Solution à un problème récurrent dans la conception
- Représenté par:
  - Nom
  - Problématique
  - Solution
  - Conséquences

Pourquoi?

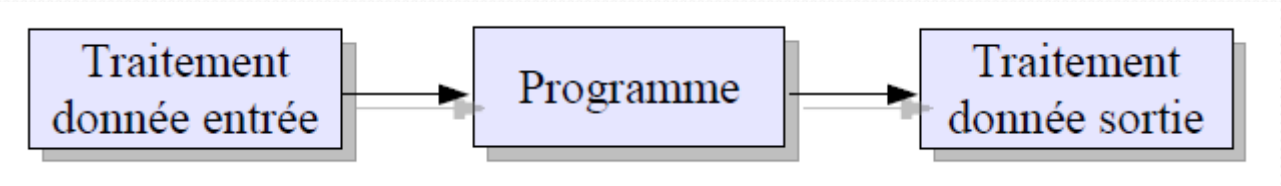
- Pour les applications devenant un peu plus complexes
- Exemple de problème:
  - Pour qu'un objet A utilise la méthode d'un autre objet B:
    - Il faut que A connaisse la référence à B
    - Donc besoin de lui passer avec un setter ou au constructeur
- Devient vite pénible pour beaucoup d'objets
- Idée -> regrouper :
  - Les vues ensembles
  - Les données ensemble

Définition

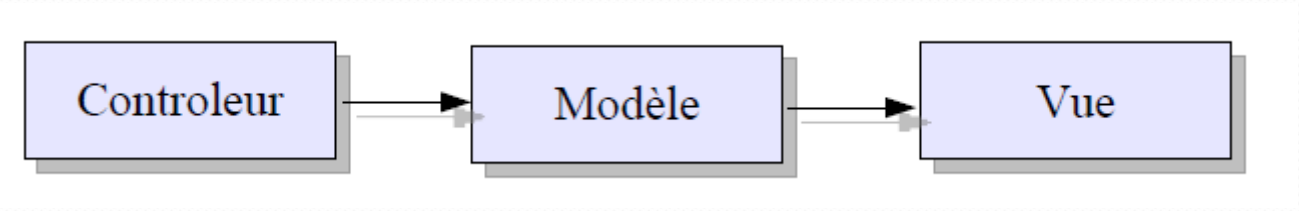
- MVC : Modèle-Vue-Contrôleur
- Répond aux besoins des applications interactives
- Sépare les problématiques des différents composants
- Regroupe les fonctions en 3 :
  - Modèle (modèle de données)
  - Vue (Présentation, interface utilisateur)
  - Contrôleur (logique de contrôle, évènement, synchro)
- Gestion de la communication par le patron Observer/Observable

Décomposition appli

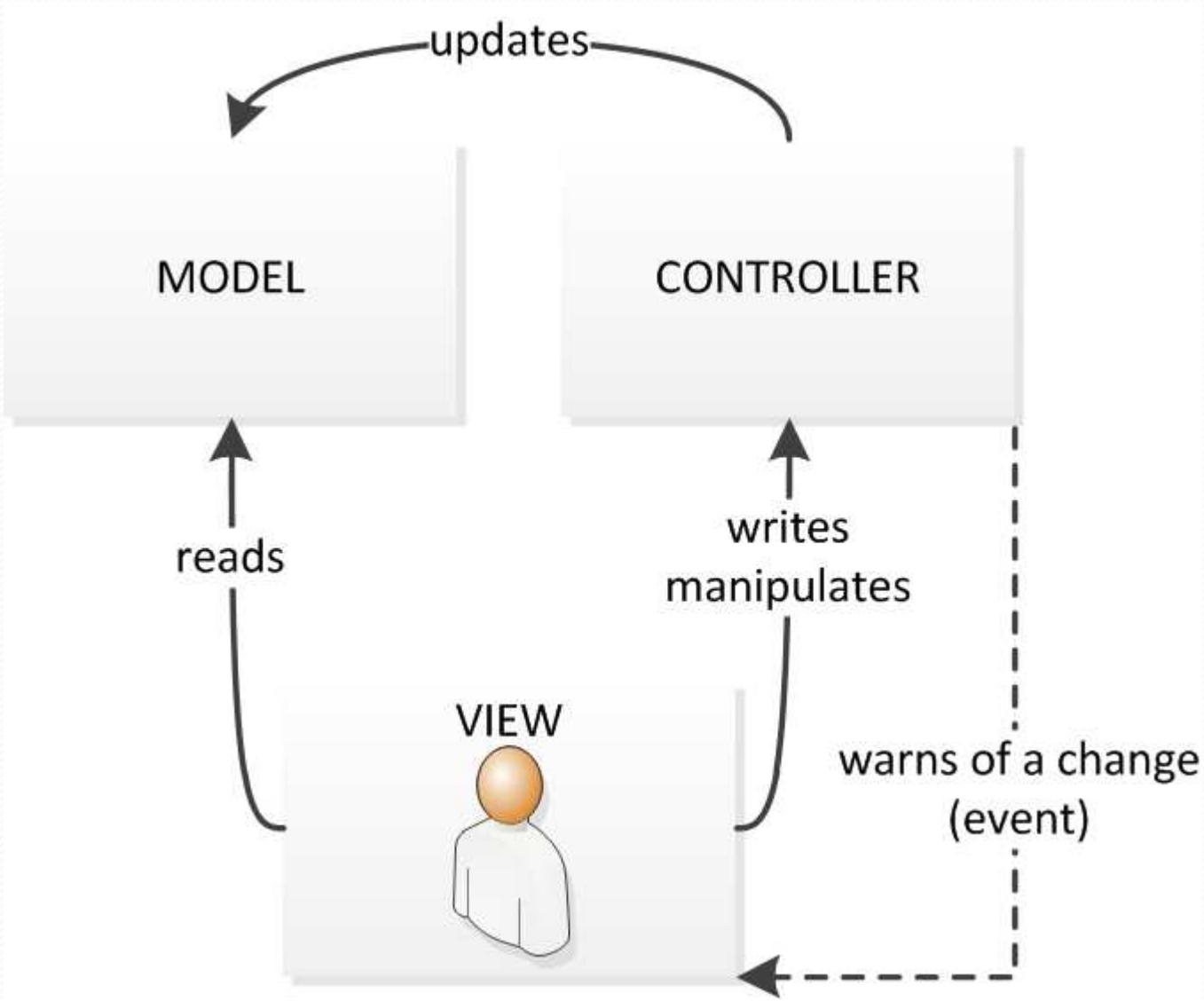
- Le plus souvent, on a:



- Ce qui peut correspondre à:



Schéma


updates
MODEL
CONTROLLER
reads
writes
manipulates
VIEW
warns of a change
(event)

Le Modèle 1/2

- Représente le cœur algorithmique de l'appli
- Il traite les données:
  - BD
  - Fichier
  - Données de saisies…
- Contrôle l'intégrité des données
- Comporte souvent des méthodes afin :
  - D'ajouter
  - De supprimer
  - De modifier des données
  - De récupérer

Le Modèle 2/2

- Résultats renvoyés par le modèle non mis en forme
- Le modèle ne connait :
  - Ni de vues
  - Ni de contrôleur
- Doit implémenter un mécanisme de Listener pour avertir les vues des changements

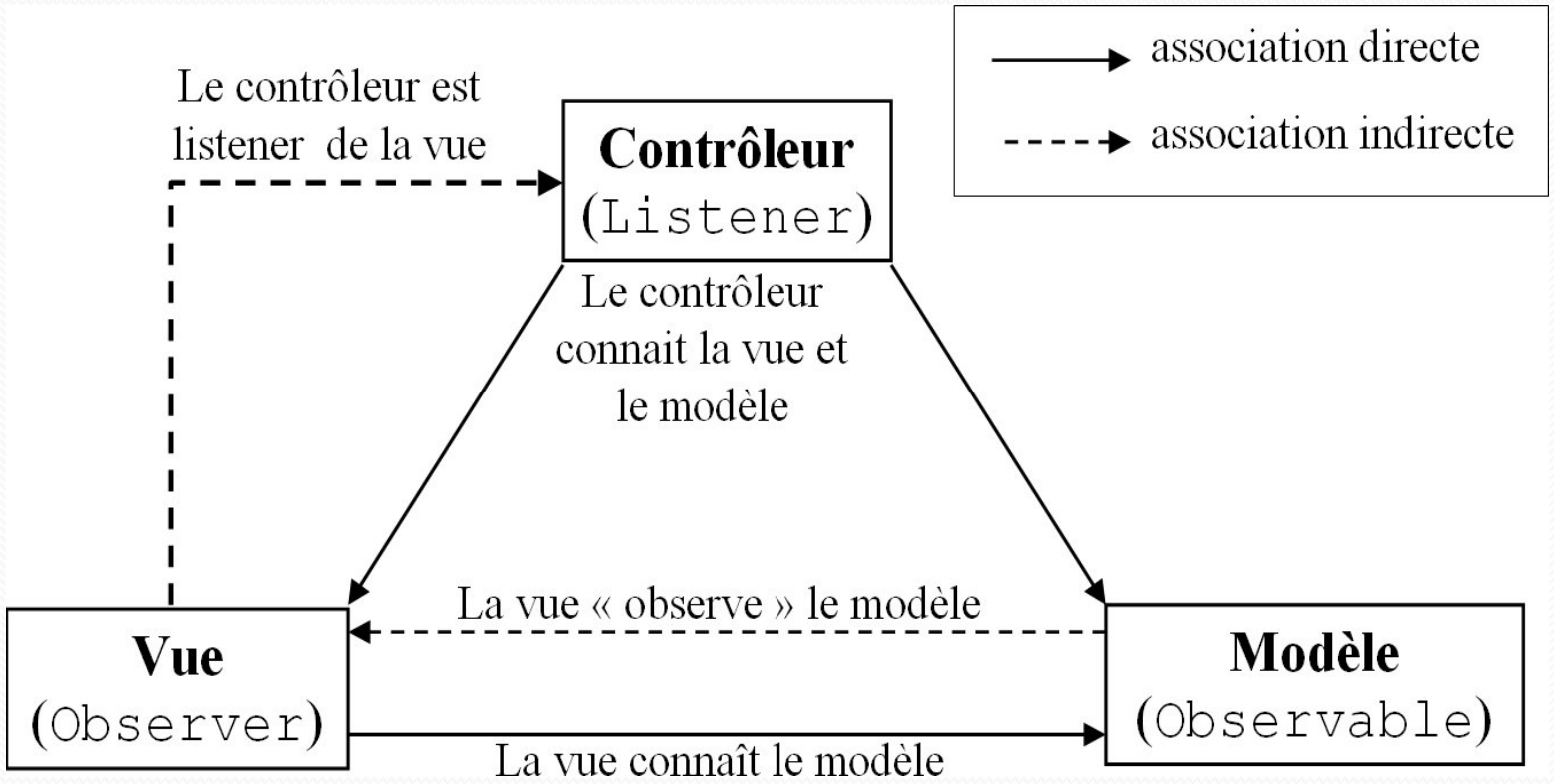
La Vue

- Interface avec l'utilisateur:
  - En entrée
  - En sortie
- Présente les résultats envoyés par le modèle
- Reçoit les actions utilisateurs
- Elle transfert chaque évènement au contrôleur
- La vue n'effectue aucun traitement
- La vue peut offrir à l'utilisateur la possibilité de changer de vue
- Elle s'enregistre en tant qu'écouteur du Modèle

Le contrôleur

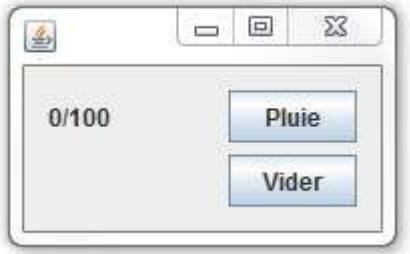
- Il gère les évènements de synchro entre la vue et le modèle
- Il reçoit tous les évènements de l'utilisateur par la vue
- Si une modification des données doit avoir lieu:
  - Il demande la modification au modèle
  - Le modèle notifie alors la vue pour màj

Exemple avec Observer/Observable



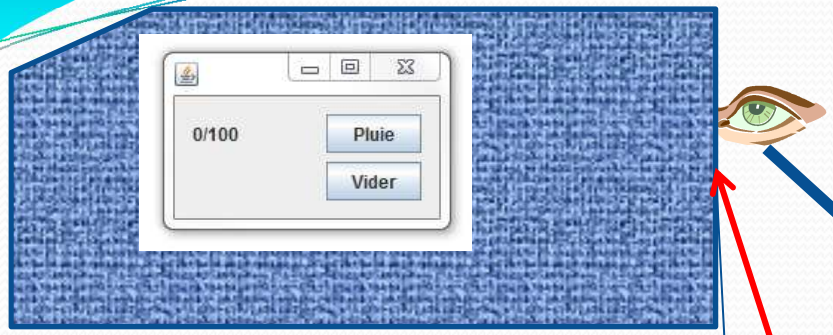
Pluviomètre (exemple simple)

- On souhaite avoir une fenêtre avec:
  - Un label indiquant la hauteur d'eau
  - Un bouton pluie qui incrémente cette hauteur
  - Un bouton vider pour vider le pluviomètre

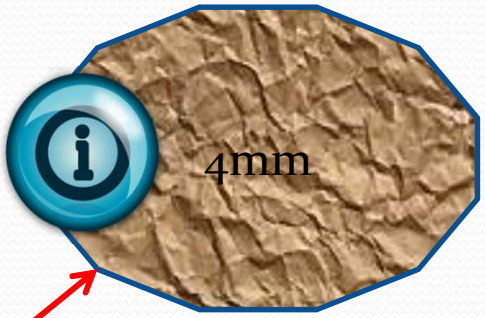


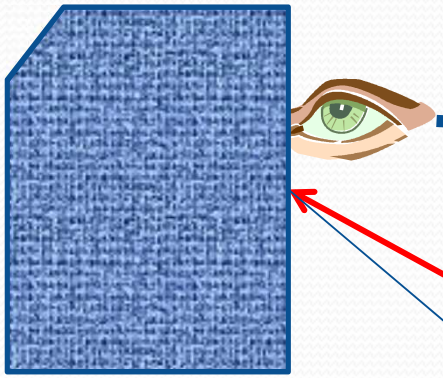
- Plus une autre fenêtre affichant la hauteur d'eau dans une ProgressBar

Les classes



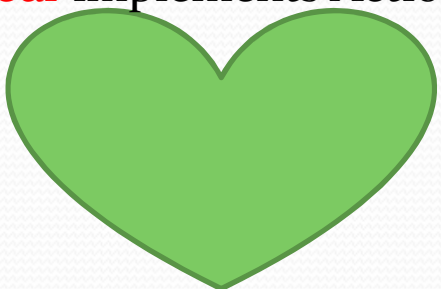
Modele implements Observable





Vue2 implements Observer

Controleur implements ActionListener



La vue1

```
public class Vue1 extends javax.swing.JFrame implements Observer {
    private Modele modele;
    public Vue1(Modele modele) {
        this.modele = modele;
        this.modele.addObserver(this);
        initComponents();
        jButtonPluie.setActionCommand("pluie");
        jButtonVider.setActionCommand("vider");
    }
    void setListener(Controleur controleur) {
        jButtonPluie.addActionListener(controleur);
        jButtonVider.addActionListener(controleur);
    }
    @Override
    public void update(Observable o, Object arg) {
        jLabelHauteurEau.setText(modele.getHauteurEau()+"/100");
    }
    @SuppressWarnings("unchecked")
    Generated Code

    // Variables declaration - do not modify
    private javax.swing.JButton jButtonPluie;
    private javax.swing.JButton jButtonVider;
    private javax.swing.JLabel jLabelHauteurEau;
    // End of variables declaration
}
```

La vue2

```
public class Vue2 extends javax.swing.JFrame implements Observer {
    private Modele modele;
    public Vue2(Modele modele) {
        this.modele = modele;
        this.modele.addObserver(this);
        initComponents();
    }
    @Override
    public void update(Observable o, Object o1) {
        jProgressBarHaureurEau.setValue(modele.getHauteurEau());
    }
    @SuppressWarnings("unchecked")
    Generated Code
    // Variables declaration - do not modify
    private javax.swing.JProgressBar jProgressBarHaureurEau;
    // End of variables declaration
}
```

Le modèle

```
public class Modele extends Observable {
    private int hauteurEau;
    public Modele() {
        this.vider();
    }
    public int getHauteurEau() {
        return hauteurEau;
    }
    public void setHauteurEau(int hauteurEau) {
        this.hauteurEau = hauteurEau;
        setChanged();
        notifyObservers();
    }
    public void inc() {
        hauteurEau++;
        setChanged();
        notifyObservers();
    }
    public void vider() {
        hauteurEau = 0;
        setChanged();
        notifyObservers();
    }
}
```

Le contrôleur

```
public class Controleur implements ActionListener {
    private Modele modele;
    public Controleur(Modele modele, Vue1 vue1, Vue2 vue2) {
        this.modele = modele;
        vue1.setListener(this);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        switch (e.getActionCommand()) {
            case "pluie":
                modele.inc();
                break;
            case "vider":
                modele.vider();
        }
    }
}
```

Exercice

- En utilisant le Pattern Observer/Observable, créer un chronomètre avec affichage en secondes dans une vue, et en minute:seconde dans une autre vue.


The End