Programmation orientée objet en Java

Le patron de conception Modèle-Vue-Contrôleur (MVC)

Dominique Blouin
Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

Objectifs d'apprentissage

- **Introduction au MVC**
- **Le patron de conception observateur – observable**
- **Variantes du MVC**
- **Affinement du MVC**

Le Modèle-Vue-Contrôleur (MVC)

- Le MVC est un patron de conception (design pattern) utilisé pour programmer des **interfaces utilisateur**.
- L'un des aspects du MVC est l'**encapsulation des données** dans un objet appelé le **modèle**.
- L'interface utilisateur peut alors proposer une ou plusieurs **vues** des données de ce modèle.
- Nous allons utiliser ce patron pour notre simulateur :
  - Nous allons définir un modèle de notre usine robotisée et une interface graphique qui visualisera ce modèle.
  - Lorsque le modèle sera simulé, grâce au MVC, les modifications du modèle seront automatiquement visibles dans l'interface graphique.

Vue (Interface Homme Machine)

- La **vue** ou Interface Homme Machine (IHM) permet aux utilisateurs d'interagir avec l'application logicielle.
  - Exemple : une application PowerPoint
- Elle prend souvent la forme de fenêtres contenant des contrôles (widgets) tels que boutons, menus, champs d'entrée, zone de dessin, etc.
- Autres noms :
  - Interfaces utilisateur (User Interface, UI).
  - Interfaces graphiques (Graphical User Interface, GUI).
  - Interfaces personne-machine.
- A distinguer de l'expérience utilisateur (User eXperience, UX).
  - Traite de l'ergonomie et du contenu, tandis que l'UI traite des interactions, et de l'aspect visuel.
- Dans ce cours, la vue vous sera **fournie** sous forme d'une bibliothèque.
  - Votre modèle devra implémenter un jeu d'interfaces Java de la bibliothèque afin de permettre sa visualisation.

Le modèle

- C'est là que se trouvent les **données**. Il s'agit en général d'un ou plusieurs objets constituant ce qu'on appelle **la couche métier** de l'application.
  - Il s'agit du cœur du programme.

- Le modèle devra contenir tout ce qui est nécessaire pour déterminer **l'état** de l'interface utilisateur et les **données** affichées par la vue de l'application :
  - Un canevas de taille représentative de la taille de l’usine robotisée.
  - Une liste des composants contenus dans l’usine.

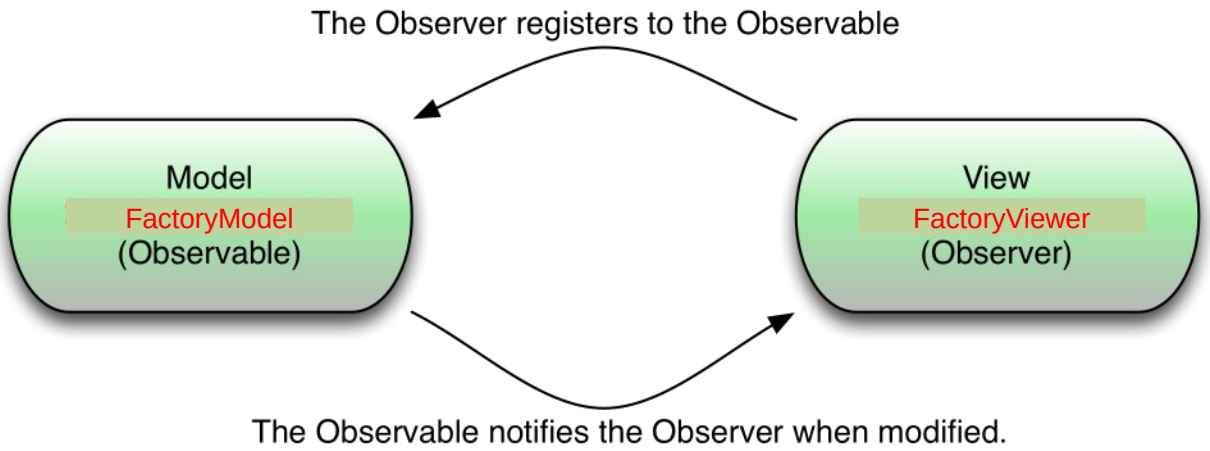
Le contrôleur

- Permet de faire le lien entre la **vue** et le **modèle** lorsqu'une **action utilisateur** intervient sur la vue.

- C'est cet objet qui aura pour rôle de **contrôler** les données du modèle.

Le patron observateur - observable

- Dans le patron de conception MVC, la vue est une représentation graphique du modèle.
  - Toute **modification** du modèle doit être **répercutée** dans la vue.
  - Pour cela, le MVC se base sur le patron de conception (design pattern) observateur - observable.

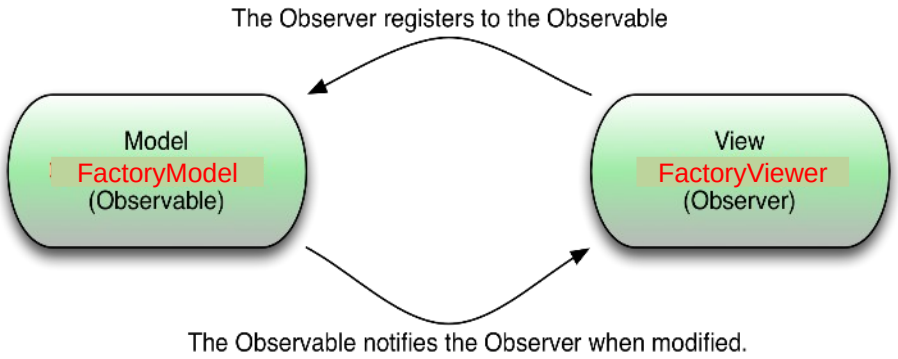


Interfaces observer - observable

- Avant que la version 8 de Java ne les déclare obsolètes (**@Deprecated**), Il existait une classe **Observable** et une interface **Observer**.

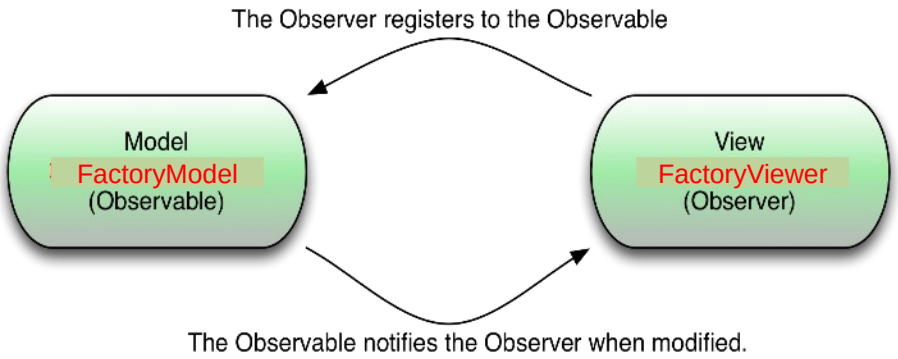
- Nous allons donc définir nos propres interfaces pour ces classes :

```
public interface Observer {

    void modelChanged();
}

public interface Observable {

    boolean addObserver(Observer observer);

    boolean removeObserver(Observer observer);
}
```



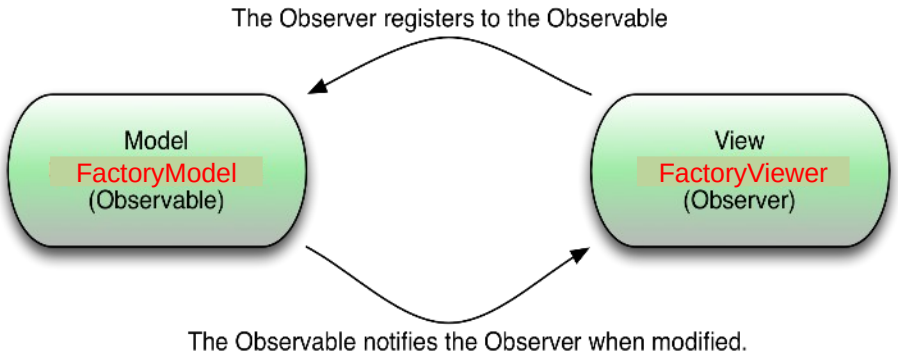
Exemple d'utilisation pour le modèle de l'usine robotisée : Observable

```
public class Factory extends Component implements Observable {

    private final List<Component> components;

    private final Set<Observer> observers;

    public Factory(Dimension dimension,
                   String name) {
        super(0, 0, dimension, name);

        components = new ArrayList<>();
        observers = new HashSet<>();
    }

    @Override
    public boolean addObserver(Observer observer) {
        return observers.add(observer);
    }

    @Override
    public boolean removeObserver(Observer observer) {
        return observers.remove(observer);
    }

    protected void notifyObservers() { // To be called every time model data is modified
        for (final Observer observer : observers) {
            observer.modelChanged();
        }
    }

    public boolean addComponent(Component component ) {
        if (components.add(component)) {
            notifyObservers(); // Notify observers that some data have changed
        }
    }
```



Exemple d'utilisation pour le modèle de l'usine robotisée : Observer

```
public class FactoryViewer implements Observer {

    private final Factory factory;

    public FactoryViewer(Factory factory) {
        factory.addObserver(this);
    }

    @Override
    public void modelChanged() {
        repaint();
    }

    @Override
    public void dispose() {
        super.dispose();

        factory.removeObserver(this);
    }
    ...
}
```



En résumé

- L’observateur (la vue-fenêtre **`FactoryViewer`**) s'enregistre auprès de l’observable (le modèle **`Factory`**) à l'aide de la méthode **`addObserver()`**.

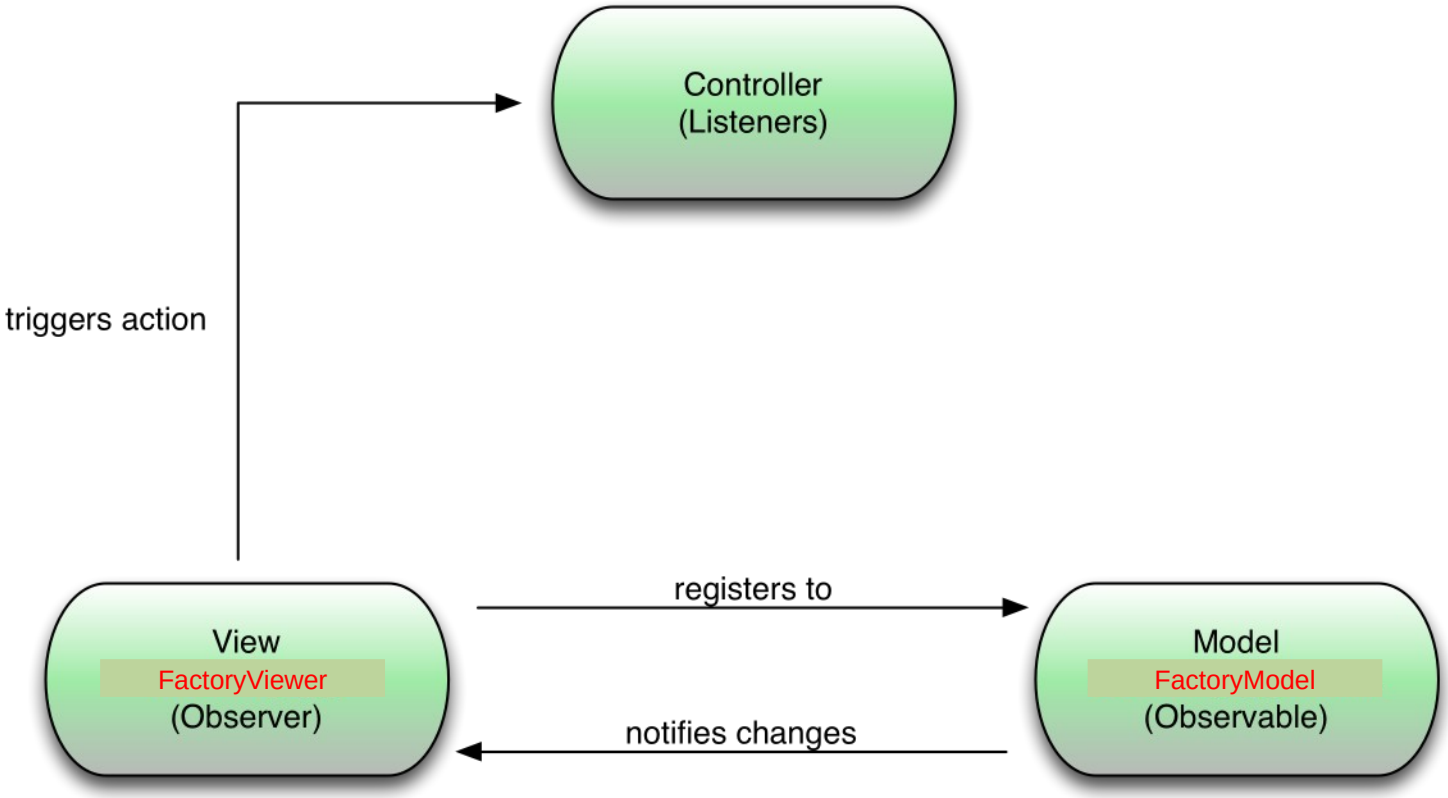
- Toute **méthode de modification** du modèle (par exemple les **setters**) a été modifiée pour qu’elle **notifie** les observateurs (vues) afin qu’elles puissent se rafraîchir.

- L’observateur (la vue-fenêtre **`FactoryViewer`**) propage hiérarchiquement le message de rafraîchissement à tous ses composants (widgets) qui doivent être rafraîchis.

- Le mécanisme est **automatisé**.

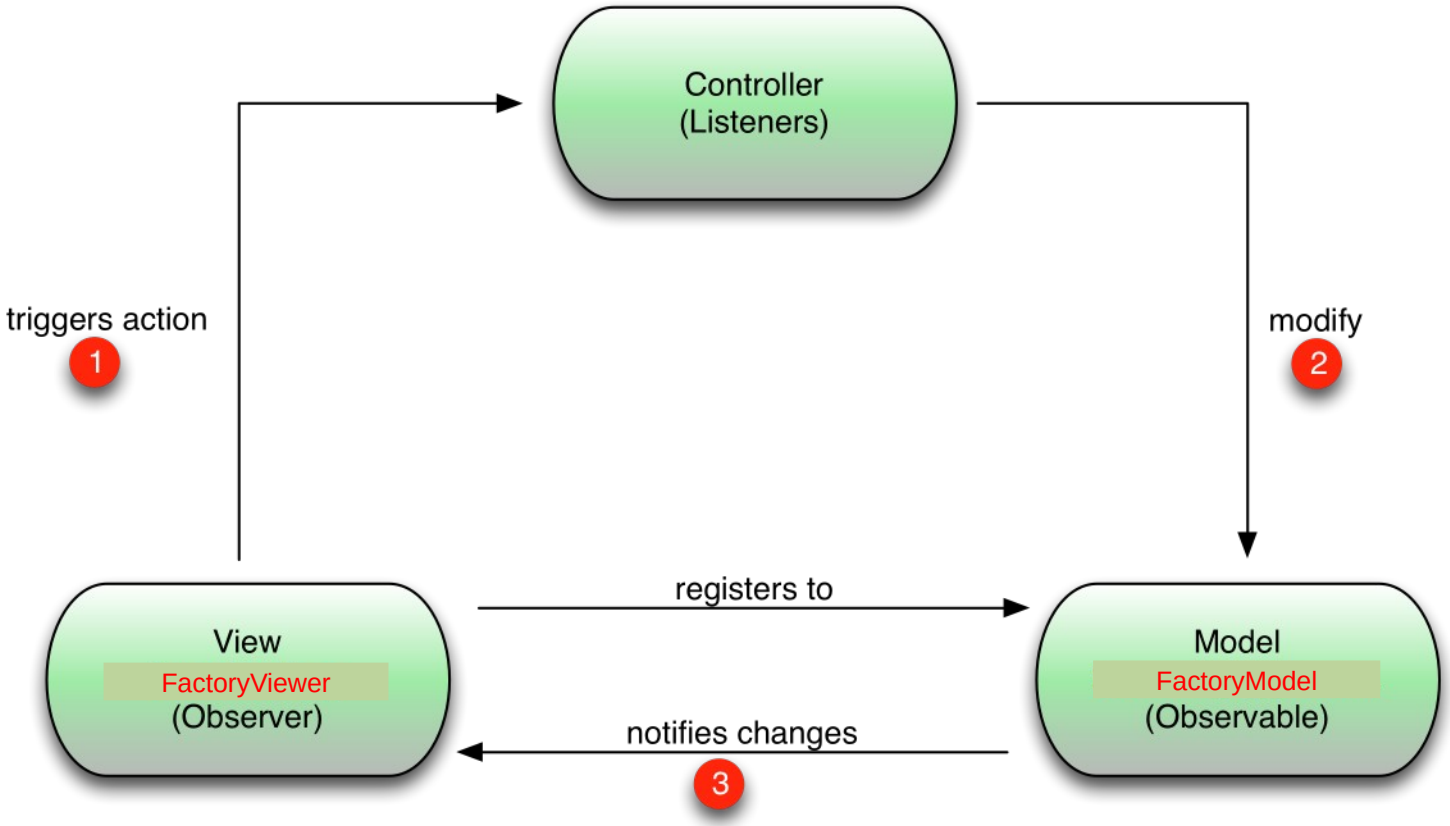
Et le contrôleur?

- La vue **modifie** (actionne) le modèle à travers le **contrôleur**.



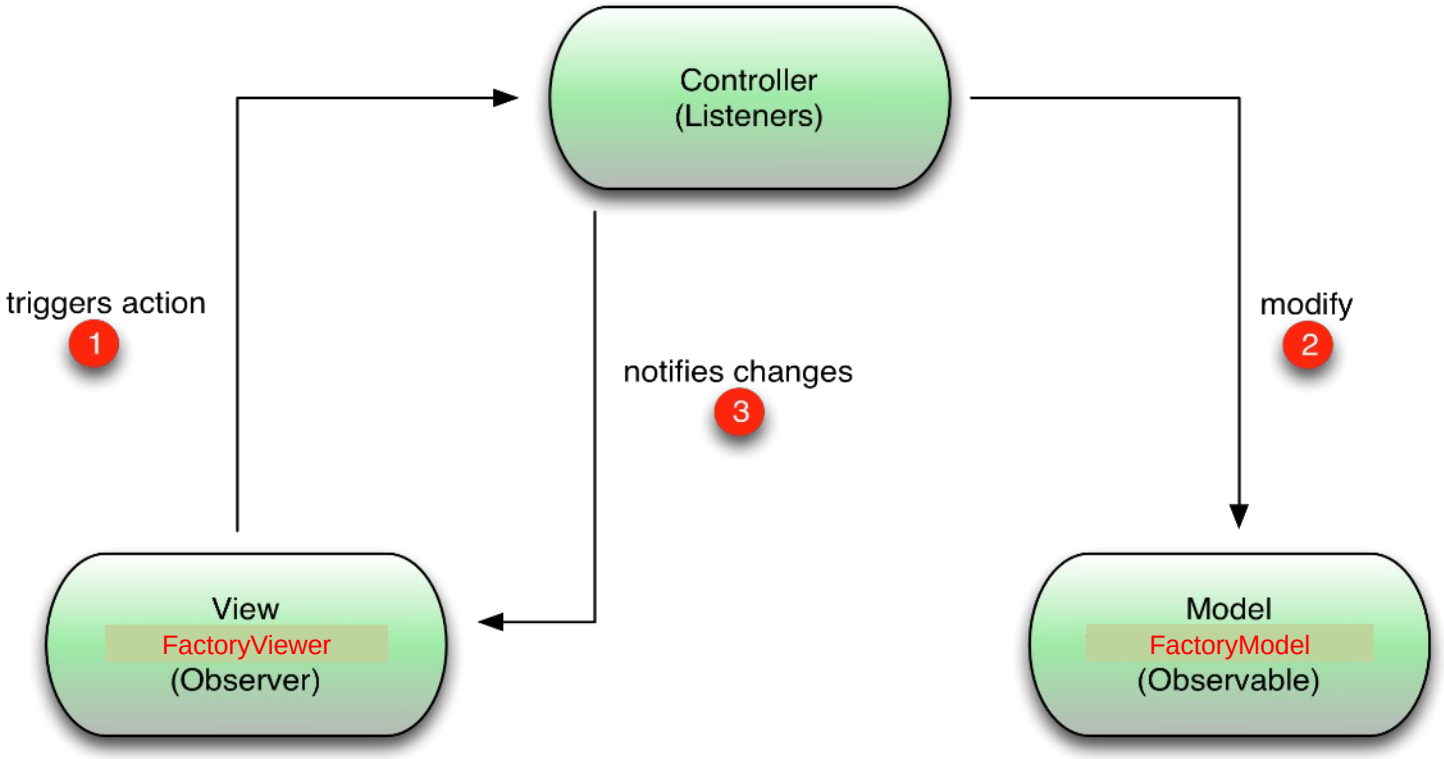
Contrôleur variante observateur - observable

- Le contrôleur modifie les données dans le modèle qui demande ensuite à la vue de se mettre à jour.

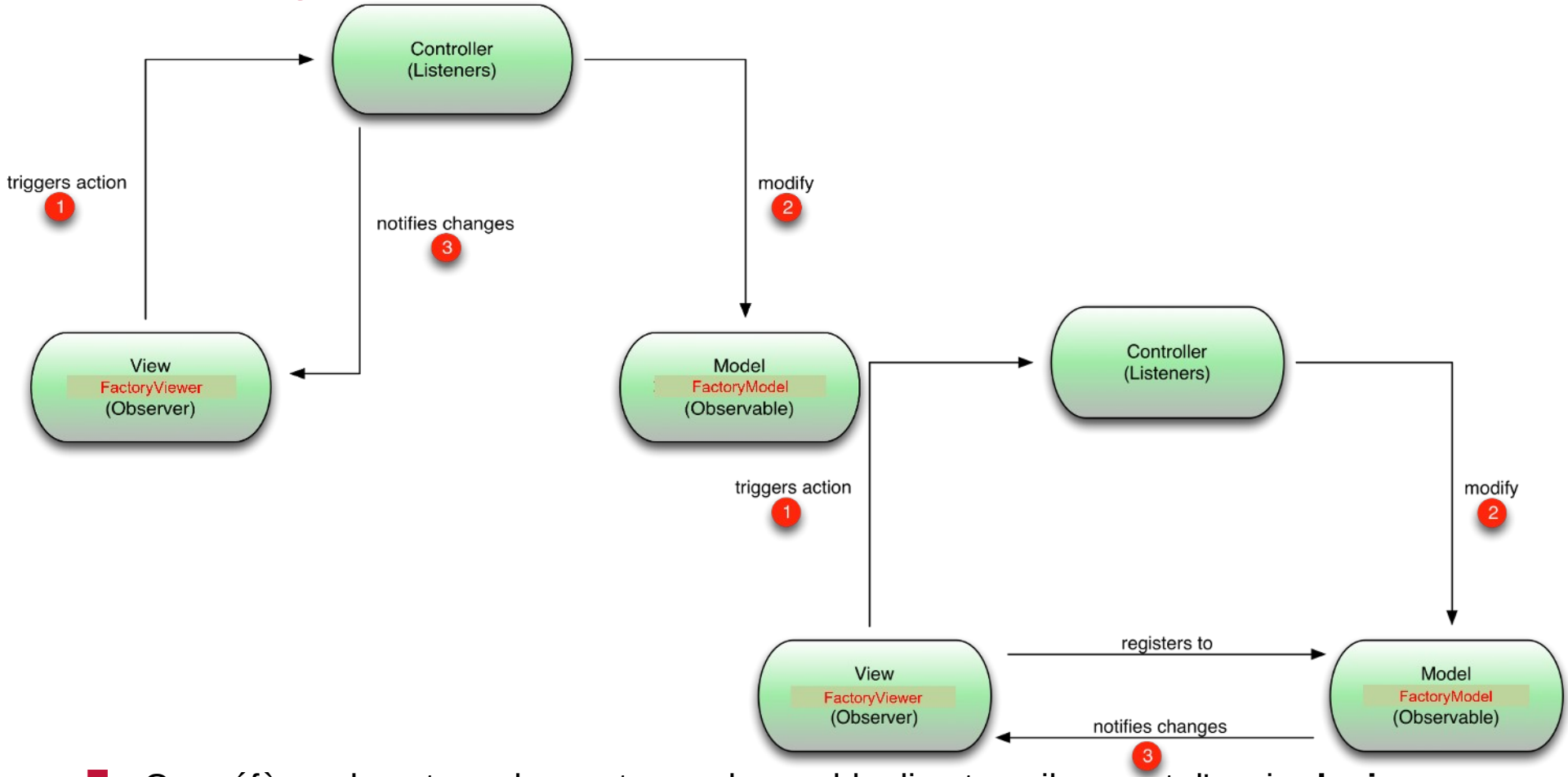


Autre variante du contrôleur

- Le contrôleur peut à la fois modifier les données dans le modèle et notifier la vue elle-même.



Quelle variante utiliser?



- On préfèrera le patron observateur - observable direct car il permet d'avoir **plusieurs vues** sur les **mêmes données**.
  - Par exemple, **Google Docs**

Interface contrôleur pour l'interface graphique de visualisation de canevas

```
/**
 * Represents the controller used by the canvas viewer to obtain the canvas model
data
 * and to control the figures animation.
 */
public interface CanvasViewerController extends Observable {

    /**
     * Returns the canvas model associated with this controller.
     * @return A non {@code null} {@code Canvas} object.
     */
    Canvas getCanvas();

    /**
     * Starts the animation of the canvas model associated with this controller.
For
     * example, moving, resizing, adding or removing figures.
     */
        void startAnimation();

        /**
         * Stops the animation of the canvas model associated with this
controller.
         */
        voi
}
```

Exemple de contrôleur pour le simulateur

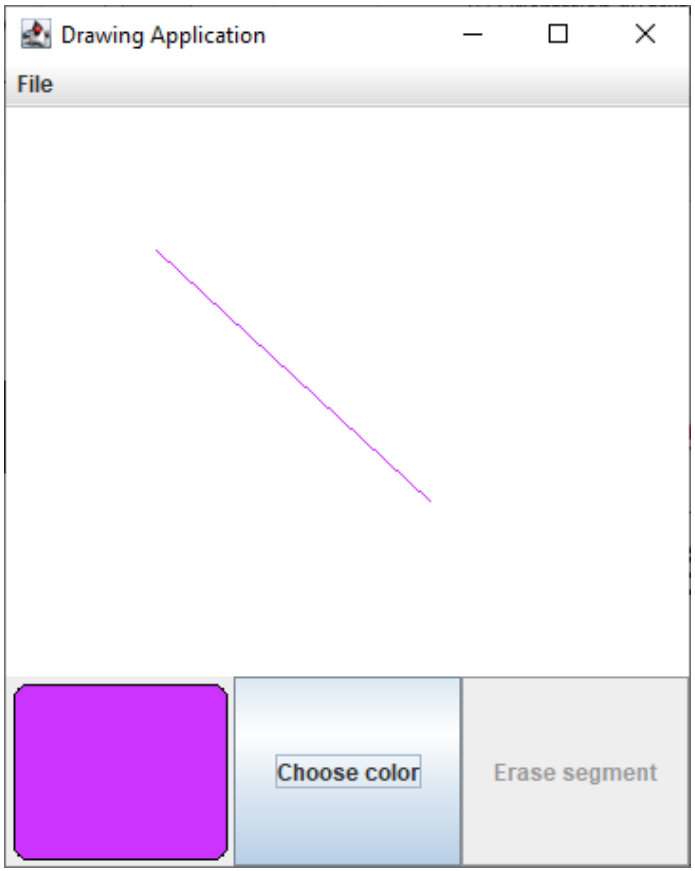
```
public class SimulatorController implements CanvasViewerController {

    private final Factory factoryModel;

    public SimulatorController(Factory factoryModel) {
        this.factoryModel = factoryModel;
    }

    @Override
    public boolean addObserver(Observer observer) {
        return factoryModel.addObserver(observer);
    }

    @Override
    public boolean removeObserver(Observer observer) {
        return factoryModel.removeObserver(observer);
    }

    @Override
    public void startAnimation() {
        factoryModel.startSimulation()
    }

    @Override
    public void stopAnimation() {
        factoryModel.stopSimulation()
    }
    ...
}
```

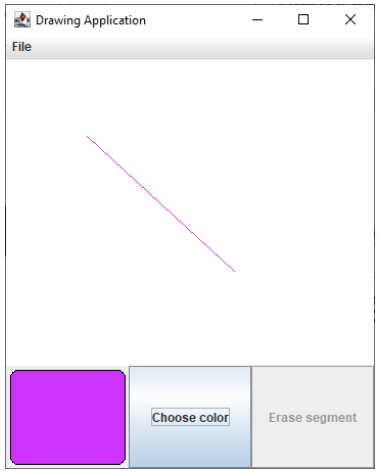
Exemple de vue pour le simulateur

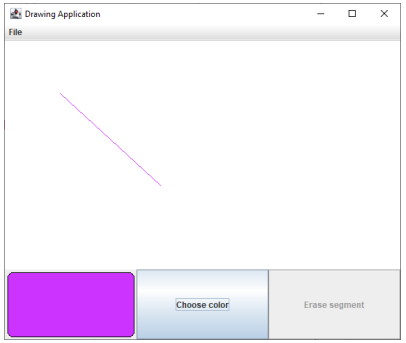
```
public class FactoryViewer implements Observer {

    private final SimulatorController controller;

    public FactoryViewer(SimulatorController controller) {
        controller.addObserver(this);
    }

    @Override
    public void modelChanged() {
        repaint();
    }

    @Override
    public void dispose() {
        controller.removeObserver(this);

        super.dispose();
    }
    ...
}
```

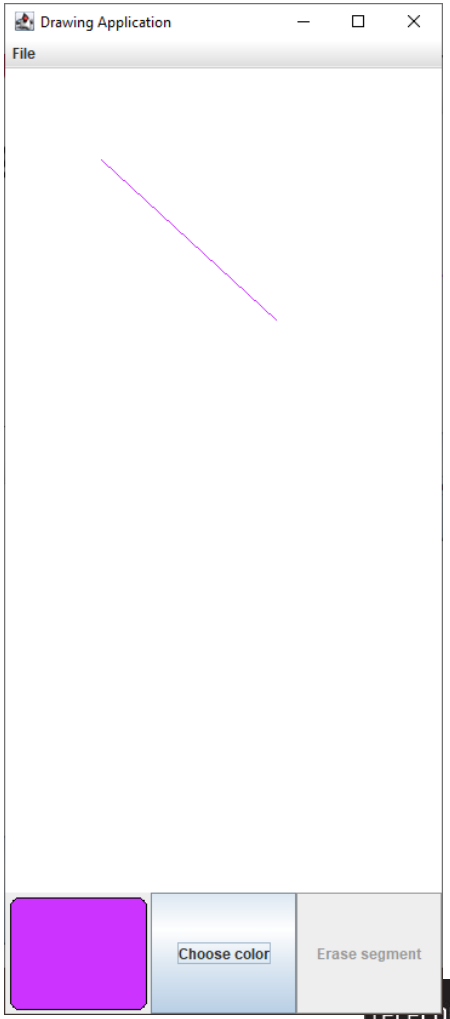
Démo d'un éditeur de lignes basé sur le MVC

- Télécharger et lancer l'éditeur de lignes déposé ici.





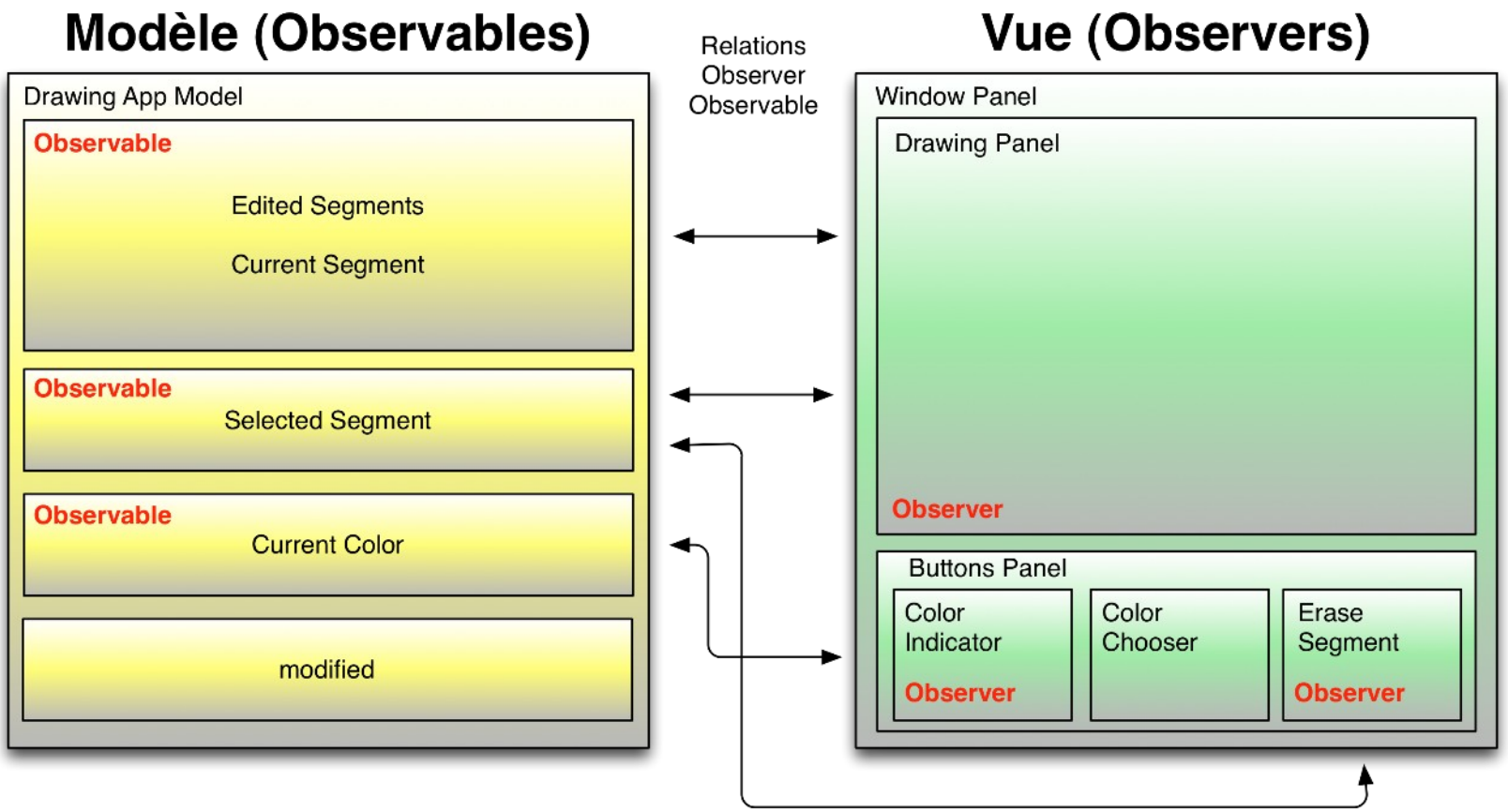




Affiner les relations du MVC

- Dans cette implémentation du MVC pour l'éditeur de lignes, nous avons utilisé une approche très **gros grain**.
- Toute modification du modèle entraîne systématiquement un rafraîchissement de **toute la vue**.
- Est-ce optimal ?
  - Quand on ne change que la position d'une seule ligne, faut-il redessiner toutes les lignes ?
- C'est pourquoi on peut proposer un découpage **plus fin** du modèle avec des relations observateur - observable **plus fines**.

Affiner les relations du MVC



Conclusion

- Le MVC est un patron simple et puissant pour l'implémentation des interfaces graphiques.
- Il s'appuie sur le patron observateur – observable pour **automatiser** la mise à jour de la vue lorsque le modèle est modifié.
- Dans cette variante, plusieurs vues peuvent être définies pour un seul modèle.
  - Très utile pour les applications collaboratives telles que Google Docs.
- Le MVC est au cœur de l'interface graphique de visualisation de canevas qui vous est fournie afin de visualiser la simulation de votre modèle d'usine robotisée.