Programmation orientée objet et temps réelle avec Java

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

Dominique Blouin
Maître de conférence
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 de production et une interface graphique qui visualisera ce modèle.
  - Lorsque le modèle sera simulé, grâce au MVC, les modifications do modèle seront automatiquement visible dans l'interface graphique.

Interfaces Homme Machine

- Les interfaces homme machine (IHM) permettent aux utilisateurs d'interagir avec leurs applications logicielles.
  - Exemple : cette application PowerPoint

- Elles prennent 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.

- Dans ce cours, l'interface graphique sera **fournie**, ainsi que des interfaces Java que votre modèle devra **implémenter** 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 Java. Ces objets s'apparentent généralement à ce qu'on appelle souvent **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 de production.
  - Une liste des composants contenus dans l'usine.

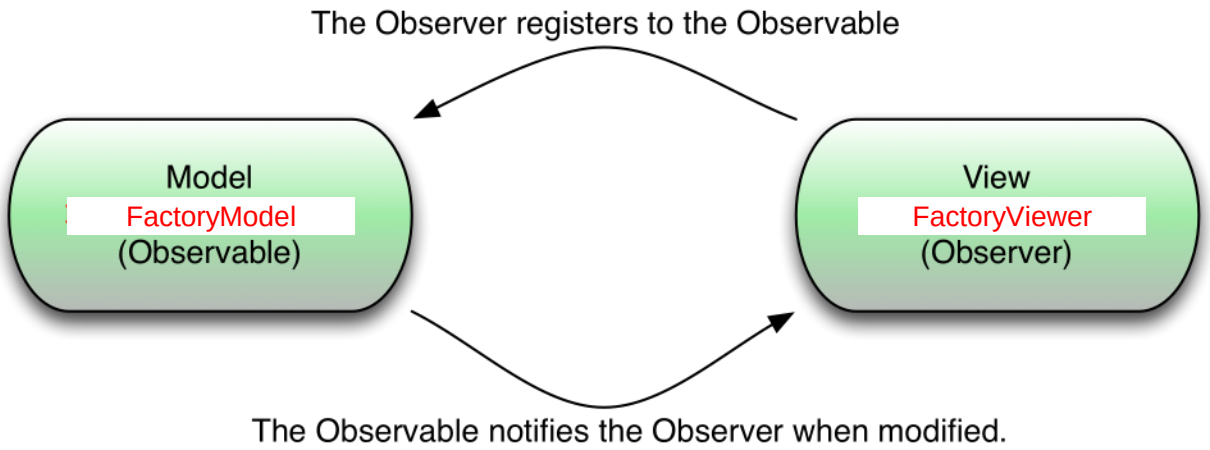
Le contrôleur

- Cet objet permet de faire le lien entre la **vue** et le **modèle** lorsqu'une **action utilisateur** est intervenue sur la vue.

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

Le patron Observateur - Observable

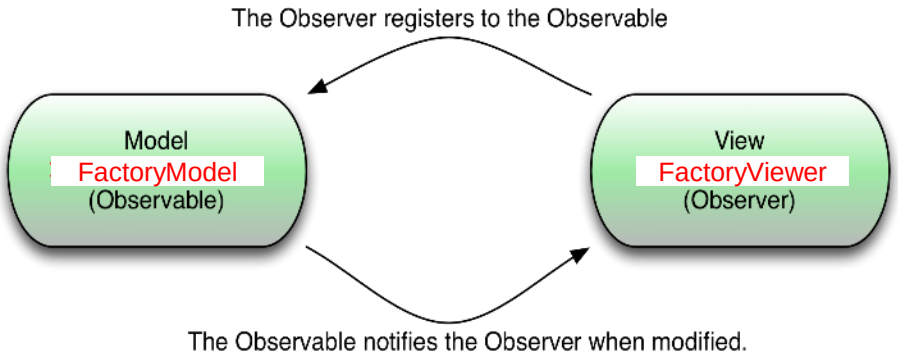
- Dans le patron de conception MVC, la vue est une représentation graphique du modèle.
  - Toute **modification** du modèle doit être **répercutée** dans la vue.
  - Pour cela, le MVC se base sur le patron de conception (design pattern) Observateur-Observable.



Interfaces Observer - Observable

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

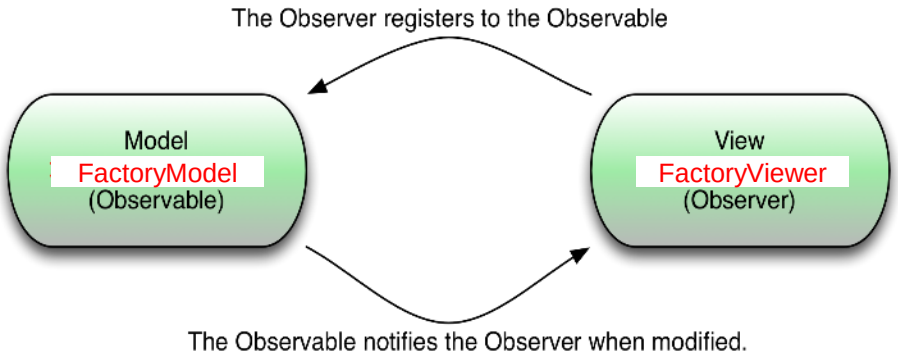
- Nous allons donc définir des interfaces pour ces classes:




```
public interface Observer {
    void modelChanged();
}

public interface Observable {
    boolean addObserver(Observer observer);
    boolean removeObserver(Observer observer);
}
```

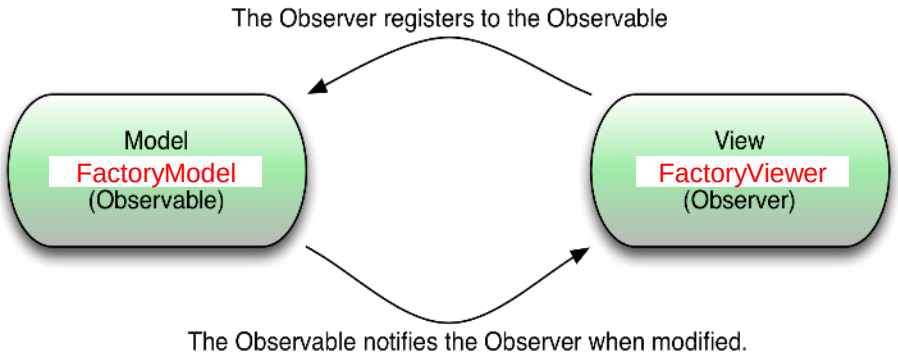
Exemple d'utilisation pour le modèle de l'usine de produits: `Observable`

```
public class Factory extends Component implements Observable {
    private final List<Component> components;
    private final List<Observer> observers;

    public Factory( final Dimension dimension,
                    final String name) {
        super(0, 0, dimension, name);
        components = new ArrayList<>();
        observers = new ArrayList<>();
    }

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

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

    protected void notifyObservers() {
        for (final Observer observer : observers) {
            observer.modelChanged();
        }
    }

    public boolean addComponent( final Component component ) {
        if (components.add( component )) {
            notifyObservers();
        }
    }
```



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

```
public class FactoryViewer implements Observer {
    private final Factory factory;

    public FactoryViewer( final 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()`**.

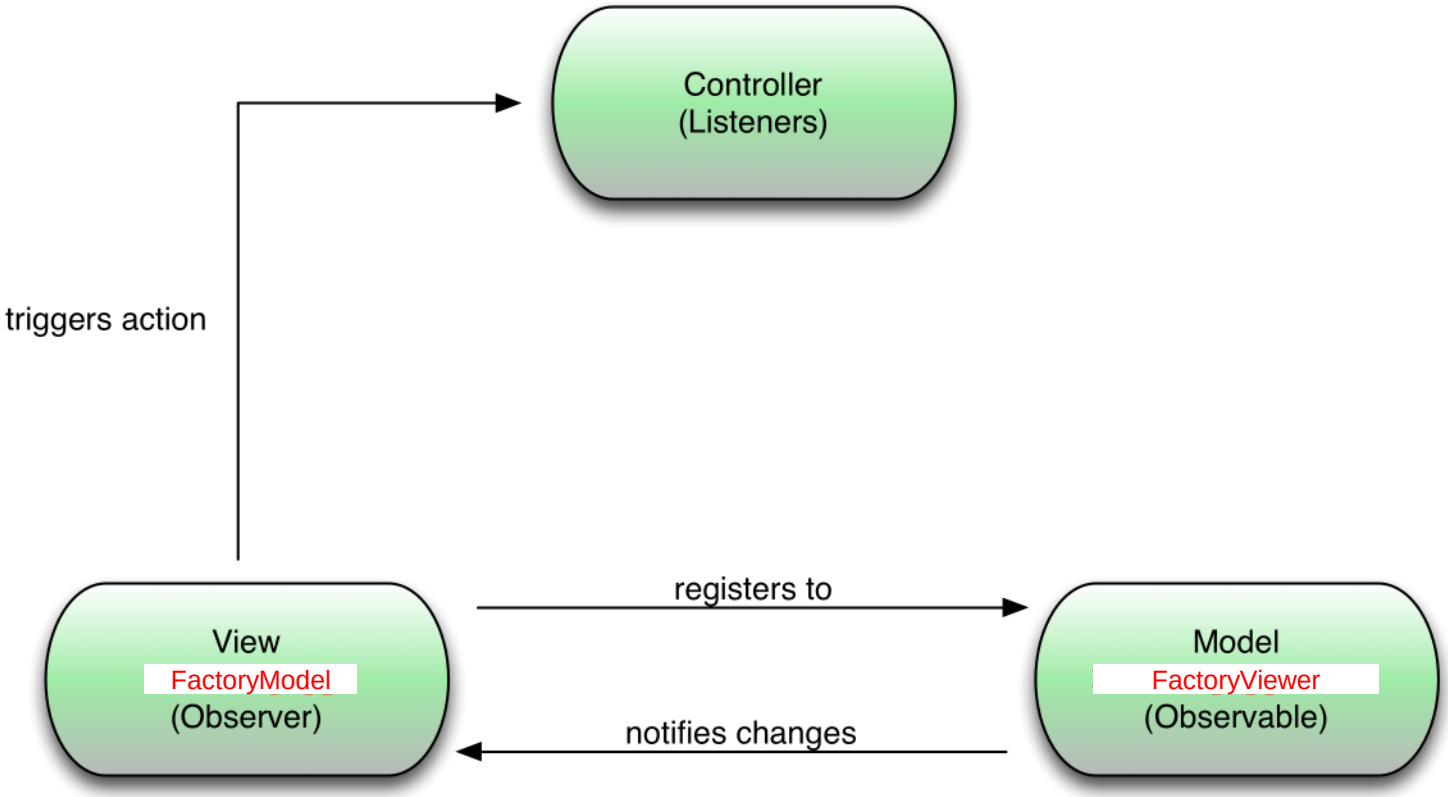
- Toute méthode de **modification** du modèle (par exemple les setters) a été modifié pour qu'elle **notifie** les observateurs (vues) afin qu'elles puissent se rafraîchir.

- 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**.

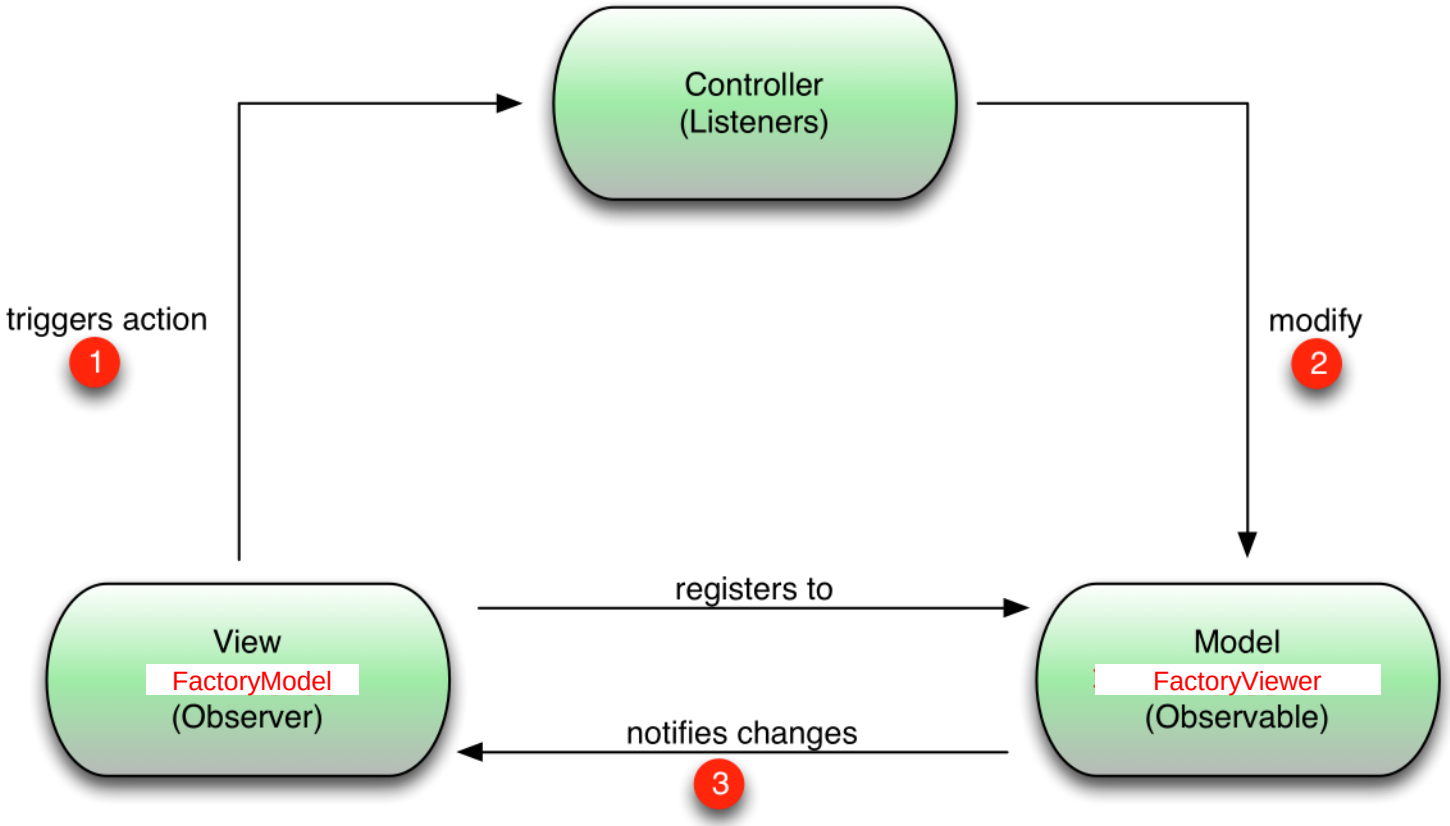
Et le contrôleur?

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



Contrôleur variante observateur - observable

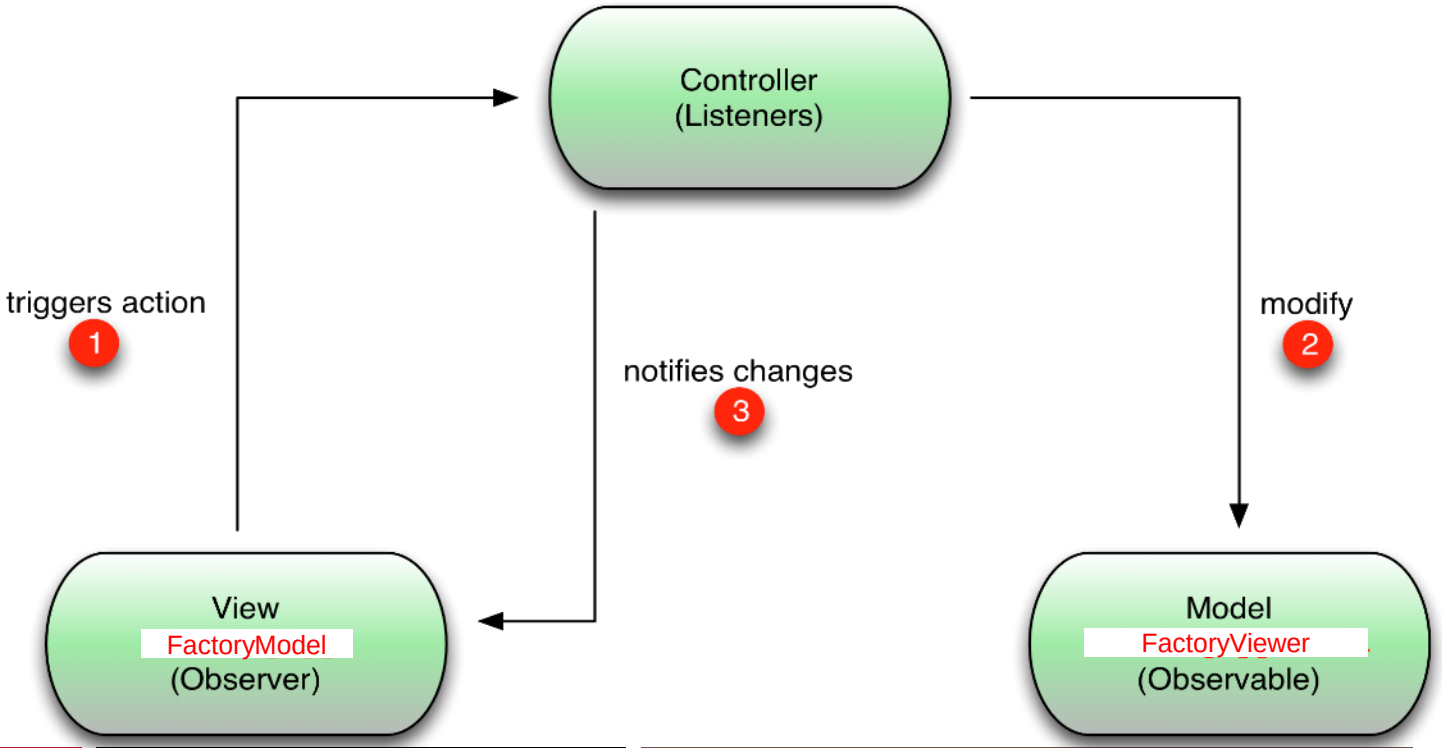
- Le contrôleur modifie les données dans le modèle qui demande ensuite à la vue de se mettre à jour.



Autre variante du contrôleur

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

- On préfèrera le patron Observateur - Observable car il est automatisé. Il permet aussi d'avoir **plusieurs vues** sur les mêmes données.
  - Par exemple, **Google Docs**



Exemple de contrôleur pour le simulateur

```
public class SimulatorController implements Controller {
    private final Factory factoryModel;

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

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

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

    public void startSimulation() {
        factoryModel.startSimulation()
    }

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

Exemple de vue pour le simulateur

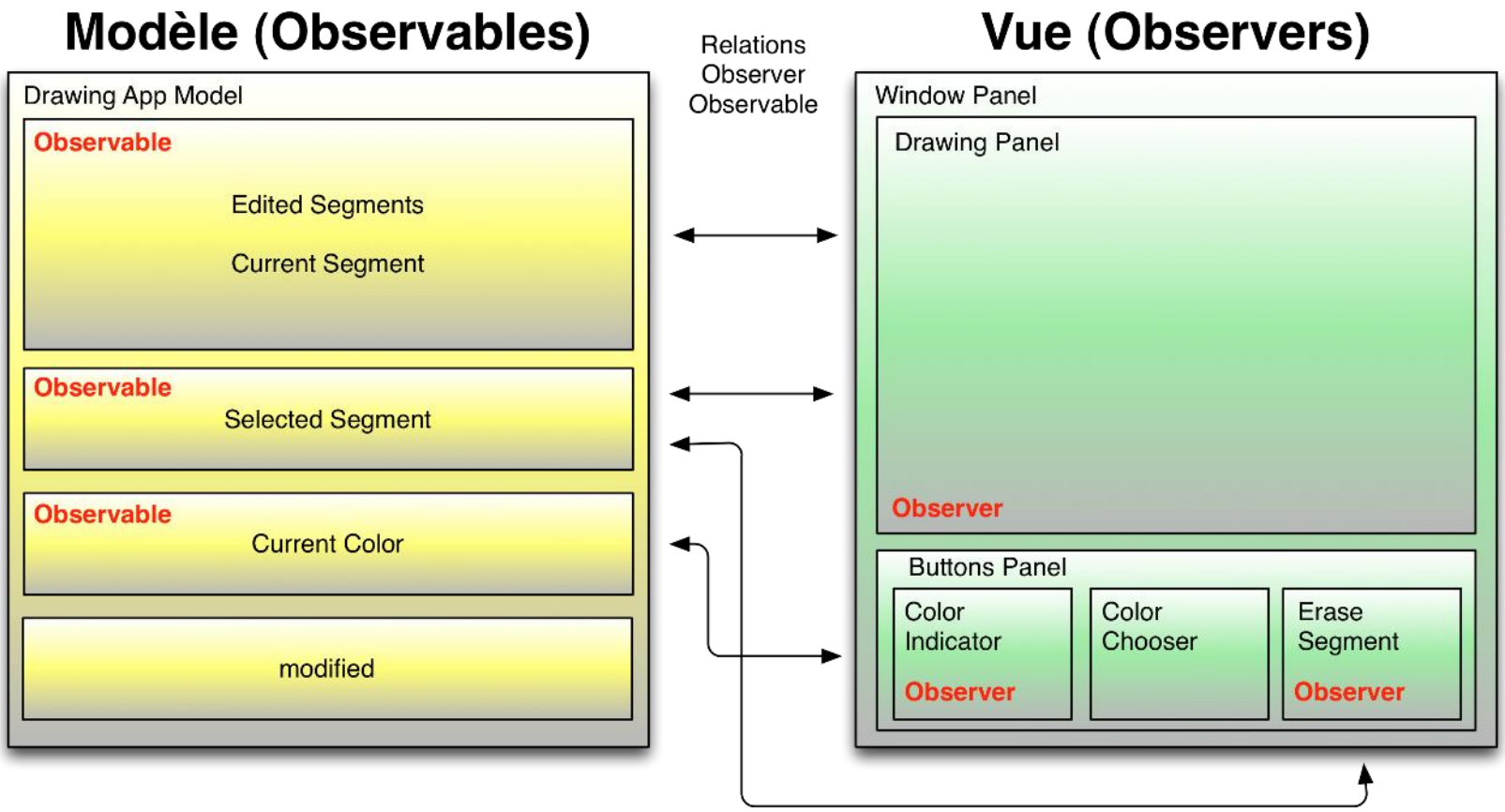
```
public class FactoryViewer implements Observer {
    private final SimulatorController controller;

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

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

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

        controller.removeObserver(this);
    }
    …
}
```

Affiner les relations du MVC

- Dans cette implémentation du MVC, 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'un seul composant, faut-il redessiner tous les composants ?
- C'est pourquoi on peut proposer un découpage **plus fin** du modèle avec des relations Observateur - Observable **plus fines**.

Affiner les relations du MVC

- Démo d'un éditeur de lignes..



Un visualisateur de simulation basé sur le MVC

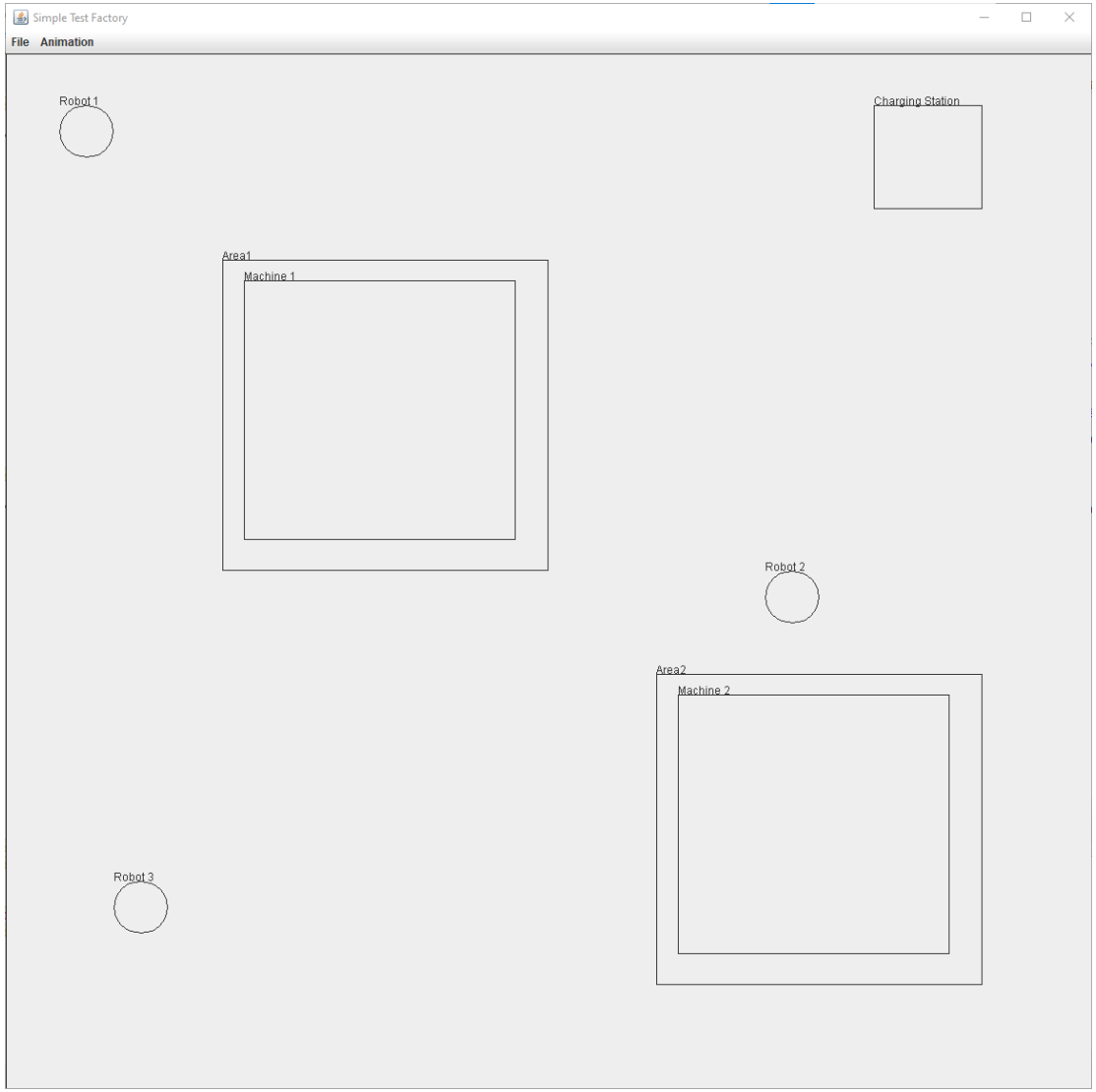
- Pour le projet de ce cours, il sera possible de visualiser vos simulations de l'usine de production de biens grâce à une interface graphique qui vous sera fournie.
- Cette interface graphique permet d'afficher différentes figures géométriques à deux dimensions dans un canevas.
- Avec cette interface graphique, un certain nombre d'**interfaces Java** modélisant un canevas contenant des figures 2D vous seront fournies.
- Pour visualiser votre modèle avec cette interface graphique, il faudra faire **implémenter** les interfaces Java fournies par votre modèle d'usine.
- Votre implémentation de ces interfaces Java constituera donc un **point de vue** de type **figures géométriques** de votre modèle de l'usine.

Interfaces du visualisateur de figures géométriques

```
public interface Figure {
    int getxCoordinate();

    int getyCoordinate();

    String getName();
}

public interface Canvas {
    int getWidth();

    int getHeigth();

    Collection<Figure> getFigures();

    String getName();
}
```

```
public interface OvalFigure extends Figure {
    int getWidth();

    int getHeigth();
}

public interface RectangleFigure extends Figure {
    int getWidth();

    int getHeigth();
}

public interface Controller extends Observable {
    String getName();

    int getCanvasWidth();

    int getCanvasHeigth();

    Collection<Figure> getFigures();

    void startAnimation();

    void stopAnimation();

    boolean isAnimationRunning();
}
```

Démonstration



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 simulation qui vous sera fourni afin de visualiser vos simulations.