3TC36: Object-Oriented Programming in Java

The Model-View-Controller (MVC) design pattern

Dominique Blouin
dominique.blouin@telecom-paris.fr
Le March 17, 2026

Learning objectives

- Introduction to MVC
- The observer-observable design pattern
- Variants of MVC
- Refinements of MVC

The Model-View-Controller (MVC)

- MVC (Model-View-Controller) is a design pattern used for programming **user interfaces**.
- One key aspect of MVC is **encapsulating data** in an object called the **model**.
- The user interface can then provide one or more **views** of the data in this model.
- We will use this pattern for our simulator:
  - We will define a model for our robotic factory and a graphical interface that will visualize this model.
  - When the model is simulated, thanks to MVC, changes to the model will automatically be reflected in the graphical interface.

View (Human-Machine Interface)

- The **view**, or Human-Machine Interface (HMI), allows users to interact with the software application.
  - Example: this PDF reader or PowerPoint application.
- It often consists of windows containing controls (widgets) such as buttons, menus, input fields, drawing areas, etc.
- Other names:
  - User Interface (UI).
  - Graphical User Interface (GUI).
  - Human-Machine Interface (HMI).
- In this course, the view will be **provided**, along with Java interfaces that your model will need to **implement** to enable its visualization.

The Model

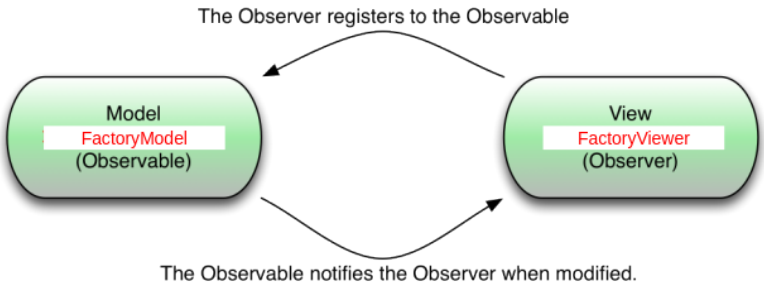
- This is where the **data** resides. It generally consists of one or more objects that form what is called the **business logic layer** of the application.
  - It is the core of the program.
- The model must contain everything necessary to determine the **state** of the user interface and the **data** displayed by the application's view:
  - A canvas sized to represent the dimensions of the production factory.
  - A list of components contained in the factory.

The Controller

- The controller links the **view** and the **model** when a **user action** occurs on the view.
- This object is responsible for **controlling** the data in the model.

The observer-observable pattern

- In the MVC design pattern, the view is a graphical representation of the model.
  - Any **modification** to the model must be **reflected** in the view.
  - To achieve this, MVC relies on the **observer-observable design pattern**.

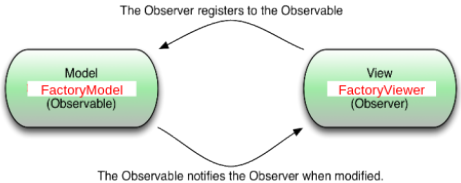


Observer - observable interfaces

- Before Java 8 marked them as obsolete (`@Deprecated`), there was a class called `Observable` and an interface called `Observer`.
- We will therefore define interfaces for these classes:

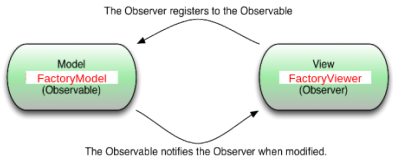
```java
public interface Observer {

    void modelChanged();

}
```

```java
public interface Observable {

    boolean addObserver(Observer observer);
    boolean removeObserver(Observer observer);

}
```



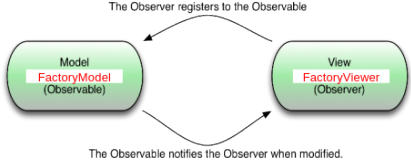
Usage example for the robotic factory model: `Observable`

```
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
        }
    }
```



Usage example for the robotic factory model: `Observer`

```java
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);
    } // ...
}
```

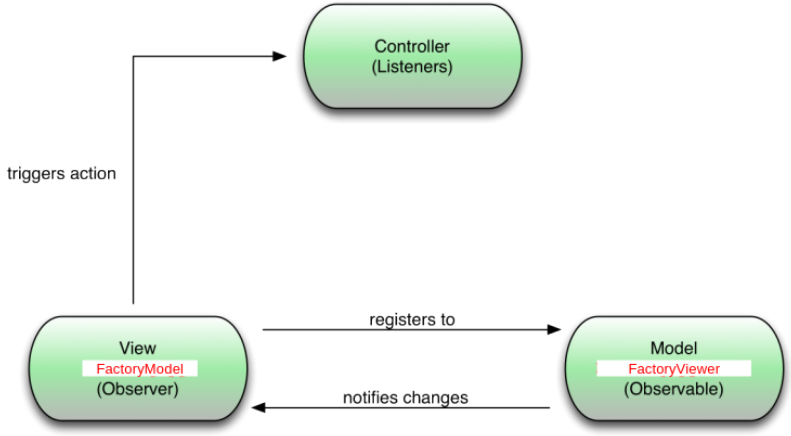


In summary

- The observer (the view-window `FactoryViewer`) registers with the observable (the model `Factory`) using the method `addObserver()`.
- Any **method that modifies** the model (for example, **setters**) has been modified to **notify** the observers (views) so that they can refresh themselves.
- The observer (the view-window `FactoryViewer`) propagates the refresh message hierarchically to all its components (widgets) that need to be refreshed.
- The mechanism is **automated**.

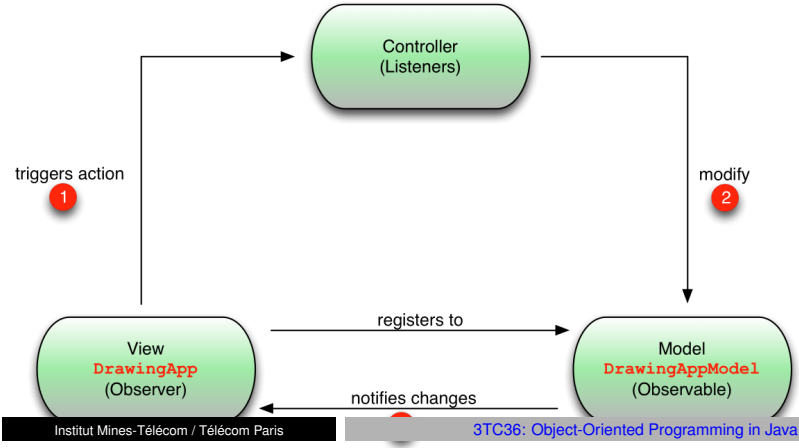
And the Controller?

- The view **modifies** (interacts with) the model through the **controller**.



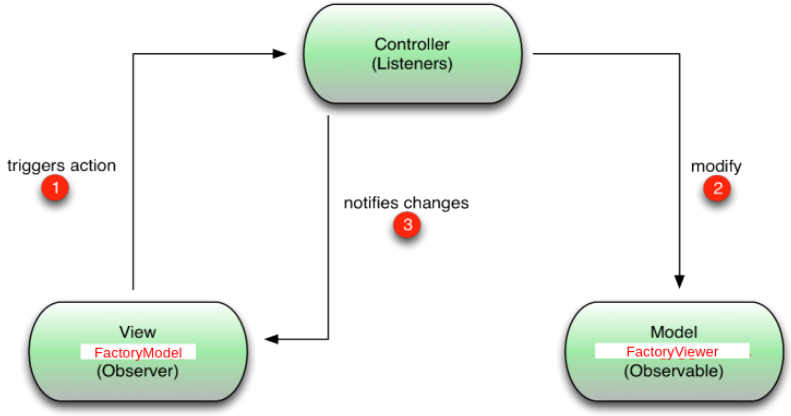
Controller variant with observer-observable

- The controller modifies the data in the model, which then asks the view to update itself.

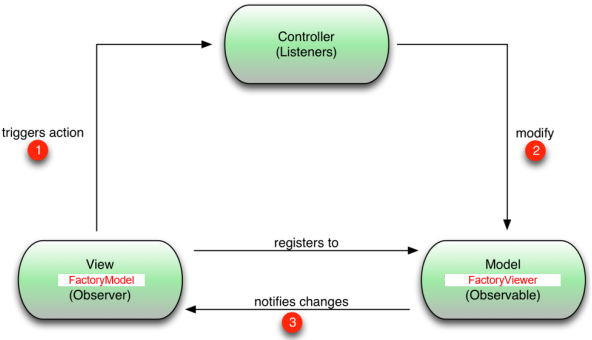


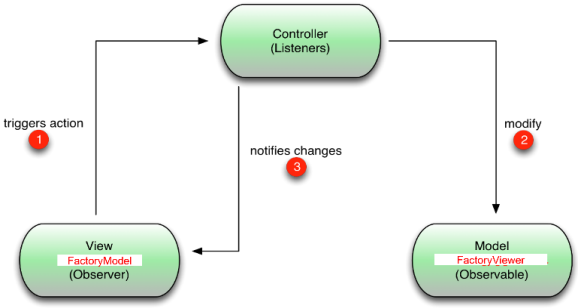
Another variant of Controller

- The controller can modify the data in the model and notify the view itself.



Which variant to use?





- The direct observer-observable pattern is preferred because it allows having **multiple views** on the **same data**.
  - For example, **Google Docs**.

Controller interface for the graphical visualization of the canvas

```
/**
 * 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.
   */
  void stopAnimation();
}
```

Example of a Controller for the simulator

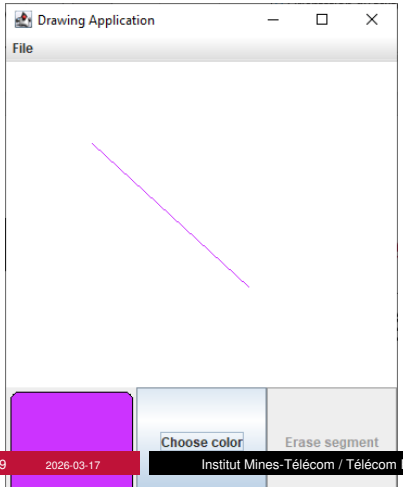
```java
public class SimulatorController implements CanvasViewerController {

    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()
    } // ...
}
```

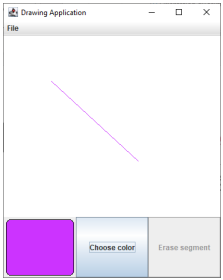
Example of a View for the simulator

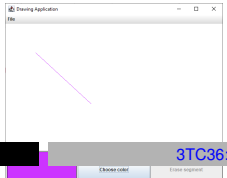
```
public class FactoryViewer implements Observer {

    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();
    } // ...
}
```

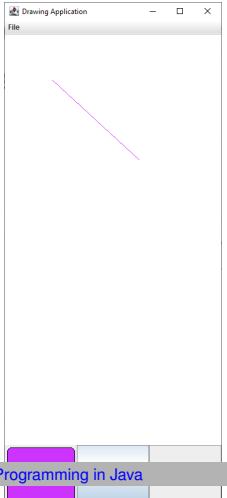
Demo of a `Line Editor` based on MVC

- Download and run the `Line Editor` provided here.







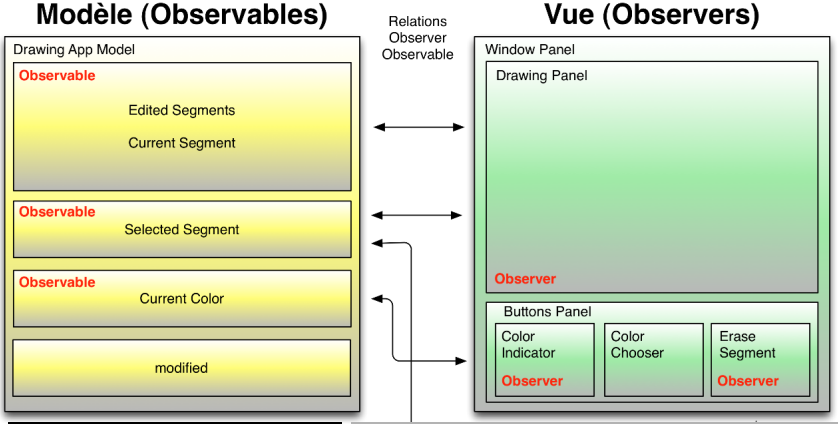


Refining the relationships in MVC

- In this implementation of MVC for the `Line Editor`, we used a very **coarse-grained** approach.
- Any modification to the model systematically triggers a refresh of the **entire view**.
- Is this optimal?
  - When only the position of a single line changes, should all the lines be redrawn?
- This is why we can propose a **finer segmentation** of the model, with **more refined** observer-observable relationships.

Refining the relationships in MVC

- Improving the way the components of the MVC architecture interact with each other.



Conclusion

- MVC is a simple and powerful pattern for implementing graphical interfaces.
- It relies on the observer-observable pattern to **automate** the update of the view when the model is modified.
- In this variant, multiple views can be defined for a single model.
  - Very useful for collaborative applications like Google Docs.
- MVC is at the core of the graphical canvas visualization interface provided to you for visualizing the simulation of your robotic factory model.