

# Lab MVC

Antoine Pigeau\*

Last update: February 14, 2024

## MVC modelling

### Goals

The goal is to implement an application that manages a constraint integer with a MVC architecture. The implementation will be carried out in JS.

The screenshot shows a web application interface titled "TP IHM". It contains two rows of input fields. Each row has a "+" button, a "-" button, a text input field, a range slider, and a status indicator. The first row has a value of 2 and a status of "On / Off" with a checked checkbox. The second row has a value of 8 and a status of "On / Off" with a checked checkbox. At the bottom, there is a "Change count" label followed by a text input field containing the number 96.

Few rules to respect:

- the code is divided in different files, one for the view, one for the model and one for the controller (as presented in the lecture)
- classes, variables, methods are correctly named
- Java convention is followed, check here for the Java convention

## 1 Model and observer

Here is the requirements of our application: the goal is to manage the value of an integer comprised in the interval  $[0, 10]$ . The model object is in charge of its value and must not accept any values outside this interval.

The first goal is to understand and implement the observer pattern, base of the MVC architecture. Read all the questions first.

---

\*from a subject by Nicolas Normand

1. create an abstract class `Observable` that contains a list of `Observer` objects, a boolean `state`, a method `boolean addObserver(Observer o)` and a method `void notifyObservers(object)` (these two methods must be implemented). An `Observer` object is notified by calling its `update(observable, object)` method. Observers are notified only if the state of the `Observable` object is changed;
2. implement an `Observer` interface with a `public void update(Observable o, object)` method;
3. implement two classes `ModelInteger` and `PrintConsole` that inherit respectively from `Observable` and `Observer`.

The class `ModelInteger` contains an integer `i` et two methods `void plus()` et `void minus()` to increment/decrement `i`. For an instance of `PrintConsole`, if `i` is changed, its state is changed and the observers are notified.

An instance of `PrintConsole` observes the `ModelInteger` instance. It only displays on the console the value of the integer observed;

4. implement a test of your observer pattern

## 2 Code organization

Retrieve the resource for the lab on Madoc, and be sure to organize your code as follow:

- `utils.js` : it contains the classes `Observable` and `Observer`
- `model.js` : it contains the class `ModelInteger`
- `controler.js`: it contains the classes `PrintConsole` and `Controller`
- `main.js` : it instanciates the model and the controller. The controller takes the model as parameter

To execute your code :

1. open the `index.html` file with your browser
2. right click on the page, inspect it and open the console tab

## 3 GUI

The interface is composed:

1. a "plus" button to increment the integer;
2. a "minus" button to decrement the integer;
3. a text field to display and change the integer.

### Question(s):

1. build your html page (without style for the moment)
2. follow the tutorial on Flexbox:  
<http://flexboxfroggy.com/>
3. create a "css" directory and implement your `style.css` to position your widgets

4. create a view class in the file `js/view.js`
5. in the class `View`, create an attribute for each component (the ones that interact with the controller), and initialize them
6. create a class `Controller` in the file `js/controller.js`, that takes your model as parameter and that creates an instance of your class `View`
7. in the class `Controller`, add the actions on the buttons
8. in the class `Controller`, add an observer on the model to update the text field
9. in the class `Controller`, add an action to change the integer directly in the text field (by taping the value - a value outside of the interval  $[0, 10]$  will the integer with the respective max or min value)
10. each button is activated / deactivated when the integer value reaches a boundary of the interval  $[0, 10]$ . Add or change an observer **on your model** to manage this task (actions of the view must not update the view - this task is the responsibility of the model observers)

For the following:

- the class `Controller` contains all the instance of observers/actions and their attachments
- the class `View` contains all the access to the widgets (all the attributes associated to the widgets)

## 4 Extension

Here, we add a second model that will be synchronized with the first one: if the first is incremented, the second is decremented (and inversely). Another extension is the possibility to deactivate the GUI of a view. Finally, extra widgets are added to manage an integer.

### 4.1 Second model with synchronization

Our app will now use a second integer, synchronized with the first one: the sum of both integer is always equal to 10. Each model will have its own view.

#### Question(s):

1. create a `superController.js` file
2. in this file, create a `SuperController` class that instantiates your previous controller two times
3. in this file, create your `UpdateSynchronize` observer to synchronize your models
4. instantiate your `UpdateSynchronize` observer in the `SuperController` constructor and make your links between your observer and the integer model(s)
5. update your main file to now create a `SuperController` instance
6. add your `superController.js` in the `html` file
7. did you carry out a copy/paste of your previous code ? of your model ? your view ? of your observer-s/actions ? (if you have at least one "yes", call a teacher to review your code)

## 4.2 Activation/deactivation

Our app must now contain a [checkbox](#) to activate/deactivate the possibility to change an integer (one [checkbox](#) for each view, and so for each model). When the activation is set to [False](#), all the widgets that provide a way to change the associated integer are deactivated.

### Question(s):

1. add a label [Activation](#) and a [checkbox](#) in your [View](#) class
2. add a new model [ModelActivation](#)

**Warning: a Javascript class is different from a Java class - in inheritance, if the same name is used in the parent class and the child class, then only one attribute is created - do not use the same variable name between [Observable](#) and [ModelActivation](#)**

3. add actions / observers in your architecture
4. if your activation / deactivation of widgets are dispersed in different classes, use the [Mediator](#) pattern to centralize the update of the view

## 4.3 GUI extension

### Question(s):

1. add new widgets:
  - a menu that contains plus and minus actions on each model (two possibilities here, one menu on each view, or a main menu for the window)
  - a slider to change the value of the integer ([Slider](#))
  - a contextual menu (a right click on your panel displays a menu with plus and minus button)
2. add tool tips on your actions

## 5 I18n

All the cultural properties of your app must be set at the loading phase of your app.

The i18n aspect will be managed with the [jQuery.i18n](#) library.

### Question(s):

1. identify all the strings to localize
2. define a key for each string (key used in our dictionaries to save the different values (different translations) of our strings)
3. provide the [json](#) dictionaries key → translation. If you use a server, it is possible to write theses dictionaries in [json](#) files
4. load the strings in your [main.js](#) file
5. add the today date on your html page
6. use a [Format](#) object to display the date
7. test your app in French and English

## 6 Component

In this part, the goal is to update our controller to create a component in the React / Angular way.

### Question(s):

1. add a new parameter to your controller: the `parentView`. This parameter is the parent element of your component
2. update your code to use this `parentView` parameter
3. update your view class to use now *Embedding javascript* (String surrounded with backticks `'Hello, ${name}'` - use the `innerHTML` to set your content - no need to use the `document.createElement` function anymore)

The skeleton of your view should look like this:

---

```
class View {
  constructor(parentView){
    this.div = document.createElement('div');
    this.div.id = '...';
    this.div.classList.add('...');

    let innerHtml = `
      ... create here your html content
      <button id="buttonPlus" title="increase of 1" class="button">+</button>
    `;

    this.div.innerHTML = innerHtml;

    this.buttonPlus = this.div.querySelector('#buttonPlus${View.count}');

    parentView.appendChild(this.div);
  }
}
```

---

And now, with this updated architecture, we can create new component. For instance, a component to count the number of change applied on an integer model.

### Question(s):

1. create the files `componentCount.js`, `viewComponentCount.js` and import them in the `index.html` file
2. implement your view with a class in `viewComponentCount.js`
3. implement your component in `componentCount.js`, with a controller class, with a constructor (`model`, `parentView`)
4. instantiate your component in the main file