TP NFP 121 – Programmation avancée

Licence Informatique 2022/2023

Design Pattern Observer / Observable

# Objectifs

Refactoriser une mini-application pour adopter un patron de conception **Observer-Observable**

# Mise en place

* Récupérer l’archive présente sur le réseau (Teams) **TPObserver**.**zip**
* Dézipper l’archive dans votre workspace.

Cela constitue votre point de départ pour le TP.

# Problématique

L’application de départ est découpée en 2 packages :

* views : contient des interfaces graphiques développées avec Swing. Elles joueront le rôle **d’observer**
* observable : contient des classes simulant un émetteur d’évènements (**observable**):
  + la classe DateSimulateur simule un changement de date toutes les 10 secondes
  + la classe WeatherSimulateur simule un changement de conditions météorologiques toutes les secondes.

La classe Main est la classe de départ de d’application. Elle instancie et initialise deux objets : un observable et une vue et les associe.

Dans l’état initial de l’application, les classes sont **fortement dépendantes** : l’observer MeteoView1 contient une propriété de type WeatherSimulateur et inversement, l’observable WeatherSimulateur contient une référence vers MeteoView1.

**En conséquence, il est impossible de** :

* Changer de classe Observer

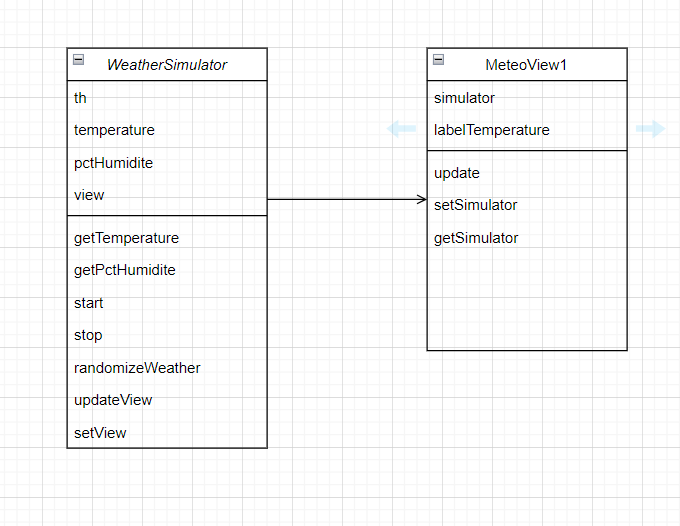
(utiliser une autre vue, implémenter une WebSocket, un service …)

* Changer de classe Observable (écouter un autre type d’évènement)
* écouter plusieurs observables (une fenêtre écoutant les changements de météo **et** de date)
* Ecouter le même observable depuis plusieurs classes Observer.

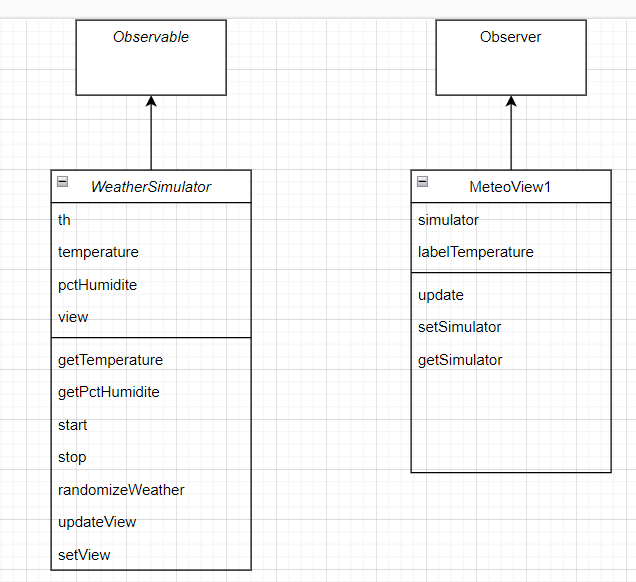
… ce qui va à l’encontre du design pattern Observer/Observable car il n’y a pas une indépendance entre « Observer » et « Observable ».

# 1ère étape : Conception

* Ecrire le diagramme de classes UML des classes WeatherSimulateur et MeteoView1 en l’état actuel.



* Ecrire le diagramme de classes UML de l’application respectant le D.P Observer/Observable (tel que l’on souhaite faire évoluer l’application).



# 2ème étape : Modification des liens d’héritage / interfaces

* **Classes observable.\***
  + Modifier les classes du package observable pour qu’elles étendent la classe Java Observable. (java.util.Observable)
  + La classe Observable contient 2 méthodes : setChanged() et notifyObservers()  
    Modifier les classes de façon à **appeler ces méthodes** (au bon endroit !)
* **Classes views.\***
  + Modifier les classes du package views de façon à ce qu’elles implémentent l’interface java.util.Observer.   
    Implémentez la ou les méthodes nécessaires.

# 3ème étape : Finalisation

Finaliser votre projet (classes Observables, views …) de façon à ce que :

1. **plus aucune** **dépendance** n’existe entre les classes « views » et les classes « observable »  
   (Vous pouvez vérifier cela en regardant les import en début de fichier)
2. La classe Main méthode puisse instancier
   1. Une vue MeteoView1
   2. Une vue MeteoView2
   3. Un WeatherSimulateur

…et faire en sorte que les **2 vues** écoutent le **WeatherSimulateur**.



# Bonus

**Gérer la fermeture des fenêtres Swing**.

Il est possible dans une classe **JFrame** de définir une action à lancer lorsque la fenêtre est fermée (via le bouton « X ») :

this.addWindowListener(new WindowAdapter() {  
 @Override  
 public void windowClosing(WindowEvent e){  
 // A LANCER LORS DE LA FERMETRE DE LA FENETRE  
 }  
});

Faire en sorte qu’à la fermeture d’une fenêtre, celle-ci se désabonne de l’observable.

 Si l’observable (simulateur) n’est écouté plus écouté par aucun observer, la simulation s’arrête (appel de la méthode stop())