TD-TP : Le Design pattern « Observateur »

1. PRESENTATION « Observateur »

La situation d'intérêt pour ce Design Pattern se présente dès lors que…

… les changements d'état d'un objet *`Observable`* (cad. observé) intéresse des objets *`Observateurs`*.

Ce Pattern apporte une solution pour ce genre de situation, et le ***Diagramme.1*** donne une représentation schématique de cette solution.

- Tout *`Observable`* connaît les *`Observateurs`* intéressés par son changement d'état (cf. `mesObservateurs`).
- Lorsqu'un *`Observable`* change d'état il notifie tous ses observateurs (cf. méthode `notifierObservateurs`) afin que chacun d'eux puisse *`réagir`* à sa façon.
- Chaque `ObservateurConcret` connaît l'`ObservableConcret` qui l'intéresse.

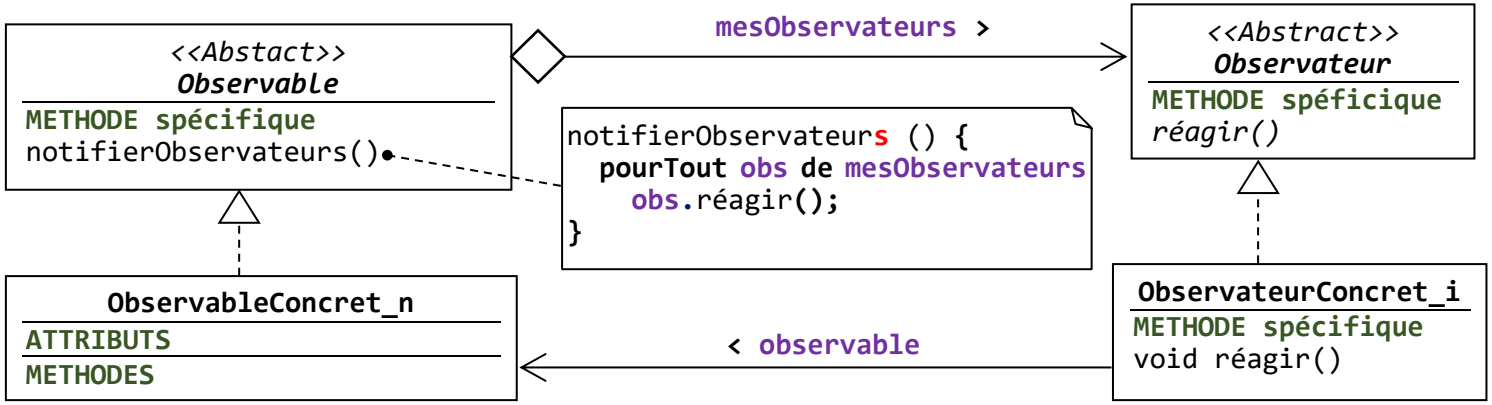

Diagramme.1 *- Représentation Schématique UML du patron Observateur*

2. COMPRENDRE « Observateur »

Si (1) une valeur dans une base de données est un *Observable*, (2a) qu'un *Observateur* affiche à l'écran la valeur de l'objet observé, et (2b) qu'un autre *Observateur* affiche une moyenne de valeurs à laquelle participe la valeur de la base de données, alors, une modification de la valeur dans la base de données (1) devra mettre à jour la valeur affichée par le premier observateur (2a) de même que la moyenne affichée par (2b).

Dans l'esprit de cet exemple, le ***Diagramme.2*** complète le schéma de solution précédent. Celui-ci modélise en plus :

- la mise en œuvre de la relation `mesObservateurs` et de la relation `observable`,
- le changement d'état de tout objet `ObservableConcret` avec la méthode `setValeurObservée()`,
- l'implémentation de `réagir()` pour tout objet `ObservateurConcret`.

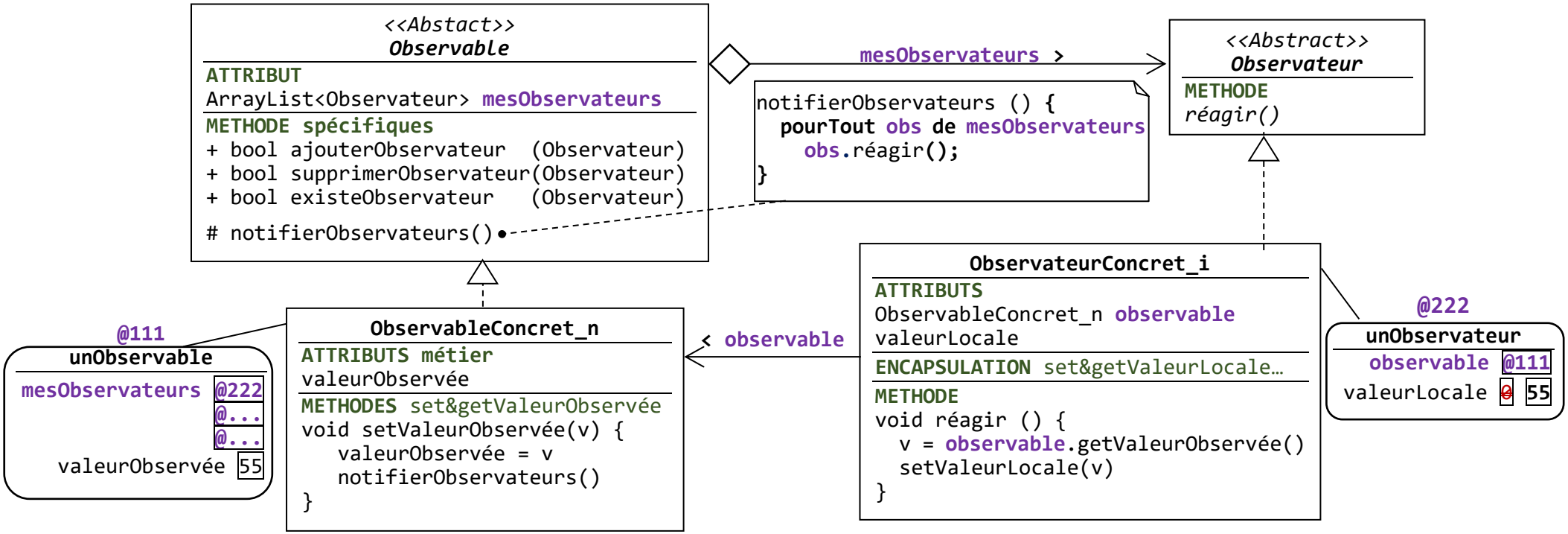

Diagramme.2 *- Représentation Schématique UML d'une 'solution' utilisant le patron Observateur*

Travail à faire : comprendre « Observateur »

1. Suivre le code induit par le ***Diagramme.2***, dès lors qu'un client invoque la méthode `unObservable.setValeurObservée(55)`. Constater que l'objet `unObservateur` mettra à jour sa `valeurLocale` (à chaque fois) que l'objet `unObservable` modifiera sa `valeurObservée`.

3. PROGRAMMER « ObservateurMétéo »

Une sonde météo positionnée sur un bâtiment réalise régulièrement 3 mesures numériques : l'`heure`, la `température` et la `pression` atmosphérique. La `sondeMétéo` est observée par un `afficheur` dans le hall du bâtiment, qui informe en temps réel sur la température et la pression atmosphérique du moment. De façon similaire, du côté d'un serveur web, l'observateur `baseDeDonnées` trace/stocke les mesures réalisées, à savoir `heure`, `température`, et `pression`.

En vous basant sur le Design Pattern « Observateur », vous allez écrire une application qui simulera un tel fonctionnement.

Avant de vous lancer dans le développement, il s'agit de cadrer l'architecture et le codage de l'application pour cela il est donc fondamental d'imaginer des exemples de comportement.

Par exemple ci-dessous, avec en **`noir`** ce qu'affichera votre application et en **`rouge`** ce que vous saisirez. Pour la saisie, vous utiliserez des éléments de code fournis en Annexe.

```
### SAISIR LES VALEURS POUR LA SONDE (on simule) ###
Date et heure (aaaa/mm/jj hh:mm) ? 2023/11/15 12:30
Température en °C ? 17
Pression en hPa ? 1024

# Le programme principal met à jour l'objet observé et l'affiche (cf. toString)
SONDE METEO : Date et heure (2023/11/15 12:30), Température °C (17.0), Pression hPa (1024.0)

... les observateurs sont mis à jour par le design pattern ...

# Le programme principal affiche (cf. toString) l'observateur : AFFICHEUR
AFFICHEUR : Date et heure (2023/11/15 12:30), Température °C (17.0), Pression hPa (1024.0)

# Le programme principal affiche (cf. toString) l'observateur : BASE DE DONNEES
BASE DE DONNEES : Date et heure (2023/11/15 12:30), Température °C (17.0), Pression hPa (1024.0)

### SAISIR LES VALEURS POUR LA SONDE (on simule) ###
... etc etc
```

Travail à faire : utiliser le *design pattern* « Observateur » pour une sonde météo

2. Sur la base du **Diagramme 2** ci-dessus, proposez une représentation UML de classes et d'objets que votre application devra supporter. Pour vous aider, il vous faut bien identifier les objets qui sont à l'origine des différents messages affichés.

3. Créer un projet Eclipse `6.ObservateurMétéo` avec les classes et comportements modélisés selon votre schéma UML.

Annexe

Extrait de code pour saisir au clavier en java (cf. https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html)

```
String uneChaine;
Double unDouble;

java.util.Scanner saisie;                    // Déclare saisie comme un Scanner
saisie =  new java.util.Scanner(System.in); // Associe saisie au flux d'entrée

unDouble  = saisie.nextDouble(); // Saisie d'un double
saisie.nextLine();               // Attend fin de ligne
uneChaine = saisie.nextLine();   // Affecte à uneChaine la saisie jusqu'à fin de ligne
```