Université Claude Bernard Lyon 1
Département Informatique

Licence 3, 2019–2020

LIFAP7

# TD5. Diagramme de classes et polymorphisme (Fil rouge Épisode III) (correction)

---

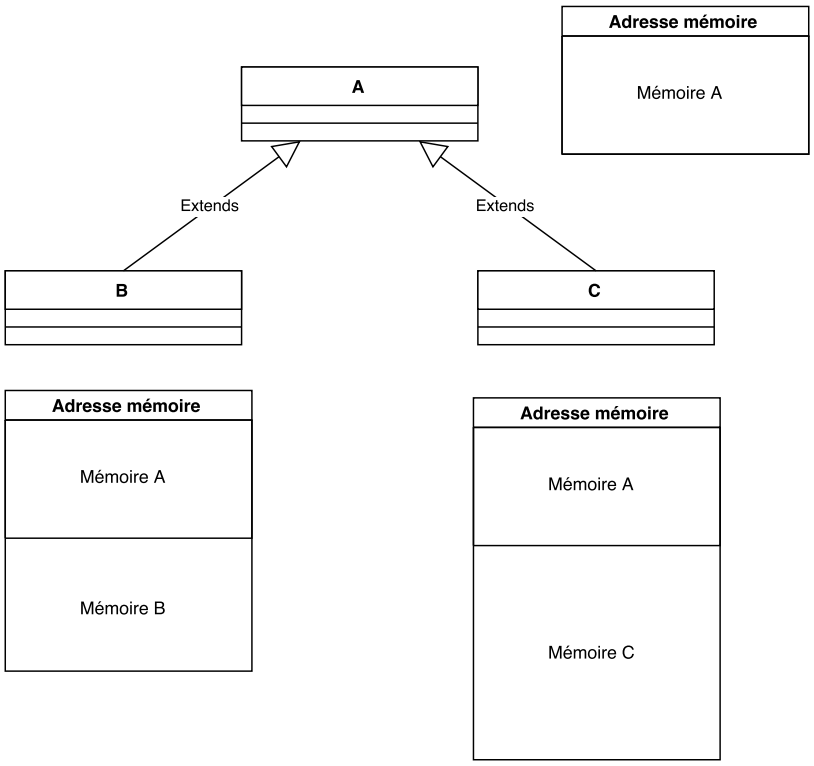
# 1 Rappels

## Polymorphisme

Le polymorphisme est un concept phare de la programmation orientée objet. Il s'agit de pouvoir traiter de manière similaire des objets de classes différentes pour peu qu'ils partagent une classe mère ou une interface. Il repose sur le fait que les classes filles partagent le même type que la classe mère car elles instancient en mémoire cette dernière.

Un exemple à garder en tête :

> Vous avez deux voitures, un vélo et une remorque. Vous les stockez dans un garage : vous venez alors de faire du polymorphisme ! Le garage est assimilable à un conteneur de véhicules qui peut accueillir toute entité étant un véhicule. Par contre, il vous est, en l'état, impossible d'y mettre ne serait-ce que la moindre framboise.



## Listes hétérogènes

Les listes hétérogènes sont une des conséquences directes du polymorphisme. Puisque les classes filles partagent un concept identique (i.e. classe A dans la Fig.), il est en effet possible de regrouper les instances de ces classes sous ce concept fédérateur A. On peut ainsi imaginer un tableau de A qui contient des A, des B et des C, puisque B et C sont des A.

## Masquage/Redéfinition

Le masquage d'une méthode consiste à redéfinir une méthode présente dans la chaîne d'héritage de la classe considérée. Pour parler de masquage, cette méthode doit avoir la même signature (à savoir le même nom, les mêmes arguments et le même type de retour).

```
class A{
    public montype maMethod(float k){
        /*do something*/
    }
}

class B extends A{
    public montype maMethod(float leNomNimportePas){
        /*do something else*/
    }
}
```

## Surcharge

La surcharge d'une méthode ou d'un constructeur est le fait de déclarer plusieurs méthodes avec le même nom mais avec des paramètres différents (et éventuellement un type de retour différent). Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments (à partir du type apparent de l'objet). La surcharge peut s'appliquer sur les méthodes de la chaîne d'héritage ou au sein de la même classe (comme dans l'exemple ci-dessous). **Attention**, vous ne pouvez pas faire de surcharge en changeant uniquement le type de retour d'une fonction et en laissant les paramètres identiques à la méthode surchargée.

```
class B extends A{
    public montype maMethod(float leNomNimportePas){
        /*do something else*/
    }

    public montype maMethod(int leNomNimporteToujoursPas){
        /*do another something else*/
    }
}
```

## Remarque

Il est important d'être capable d'identifier quelle méthode sera choisie par le compilateur et les possibilités que cela offre. Un exemple :

```
public static void main(){

    A[] tab = new A[5];
    tab.add(new A());
    tab.add(new B());
    tab.add(new A());
    tab.add(new B());
    tab.add(new A());

    for (int i = 0 ; i < tab.length; i++) {
        tab[i].maMethod(2.0);
        // Compilation: choix signature sur le type apparent (A) -> maMethod(float)
        // Exécution: appelle la méthode maMethode(float) de la classe du type réel

        tab[i].maMethod(7777);
        // Compilation: maMethod(float) (car un int est un float).
        // Exécution: appelle la méthode maMethode(float) de la classe du type réel

        if(tab[i] instanceof B)
            ((B)tab[i]).maMethod(7);
            // Compilation: type apparent (B) -> maMethod(int)
            // Exécution: on appelle donc maMethode(int) de B

        ((B)tab[i]).maMethod(7);
        // Compilation, type apparent (B) -> maMethod(int)
        // Exécution: erreur puisque certains objets sont de type A (donc pas des B)
     }
}
```

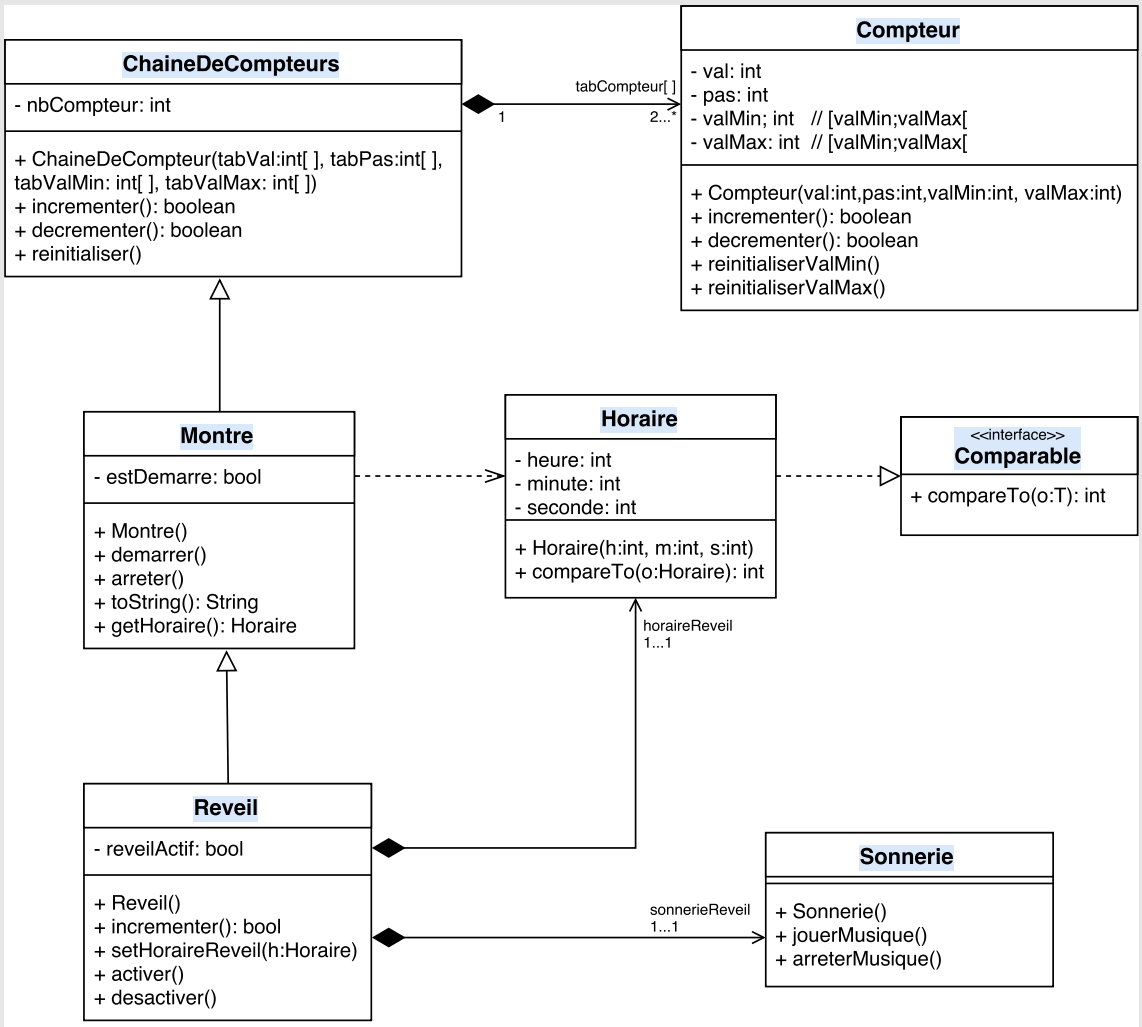
# 2 Introduction aux concepts avancés de la POO

Votre chef de projet a décidé d'introduire le concept d' Horloge-Réveil (ci-après, simplement "Réveil") au sein du projet, qui permet non seulement d'afficher l'heure mais aussi d'avoir une fonctionnalité sonnerie. Les spécifications du Réveil sont les suivantes :

- être activable et désactivable
- être réglable à un certain horaire
- avoir une sonnerie déclenchable automatiquement (méthode `jouerMusique`). Le déclenchement de la sonnerie se fait si et seulement si l'horaire du réveil est supérieur à un horaire donné et le réveil est actif.

**Question 1**

*Modéliser le* ***`Reveil`****, ainsi que la fonctionnalité de* ***`sonnerie`****, qui doit être réutilisable dans d'autres projets.*

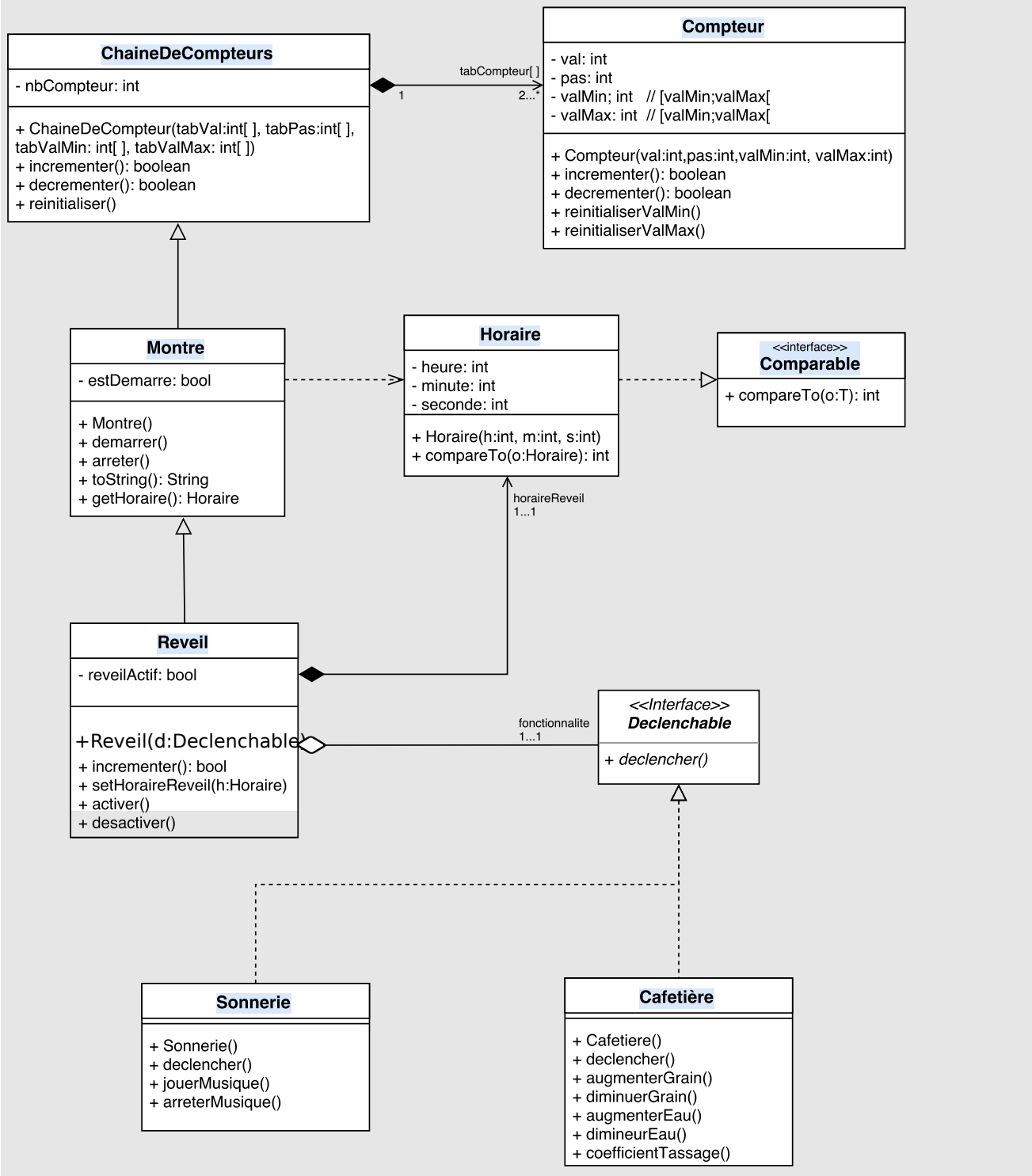



```
public class Reveil extends Montre {
   private boolean reveilActif ;
   private Horaire horaireReveil ;
   private Sonnerie sonnerieReveil ;

    public Reveil(){
        super();//appel du constructeur de la classe mère
     this.reveilActif = false;
     this.horaireReveil = new Horaire(0,0,0);
     this.sonnerieReveil = new Sonnerie();
    }

    //Redefinition de la méthode incrementer()
    // presente dans la chaine d'heritage (classe ChaineDeCompteurs)
    public boolean incrementer() {
        /* On appelle la methode incrementer de la classe mère
          * en utilisant le mot clé super. Si on avait utilisé this, alors
          * on aurait fait un appel récursif à la méthode incrémenter
          * que nous sommes entrain d'implémenter. */
        boolean b = super.incrementer()
        /* Ici,  en faisant this.getHoraire, on utilise la méthode
          * de la classe mère Montre puisque this (ici Reveil) ne
          * réimplemente pas cette méthode. */
        if(reveilActif && this.getHoraire.compareTo(this.horaireReveil) == 0) {
            sonnerieReveil.jouerMusique();
        } return b;
```

Après une étude sur les informaticiens, il s'avère que l'odeur du café est autant, voire plus, efficace pour les réveiller qu'une sonnerie. L'idée ici est d'ajouter une fonctionnalité pour que notre réveil soit capable de faire du café. On peut imaginer la possibilité de régler la force et la quantité de café. Le Reftière ® (réveil faisant cafetière) se `déclenchera` dans les mêmes conditions que pour la sonnerie. En revanche, lors de la création d'un Réveil, il faudra spécifier la fonctionnalité qui se `déclenchera` (sonnerie ou café).

### Question 2

*Proposer une modification du diagramme de classes précédent afin d'intégrer au mieux cette nouvelle fonctionnalité* `Cafetière` *(penser aux interfaces).*



Voici à quoi pourrait ressembler une partie du code de la classe `Reveil`

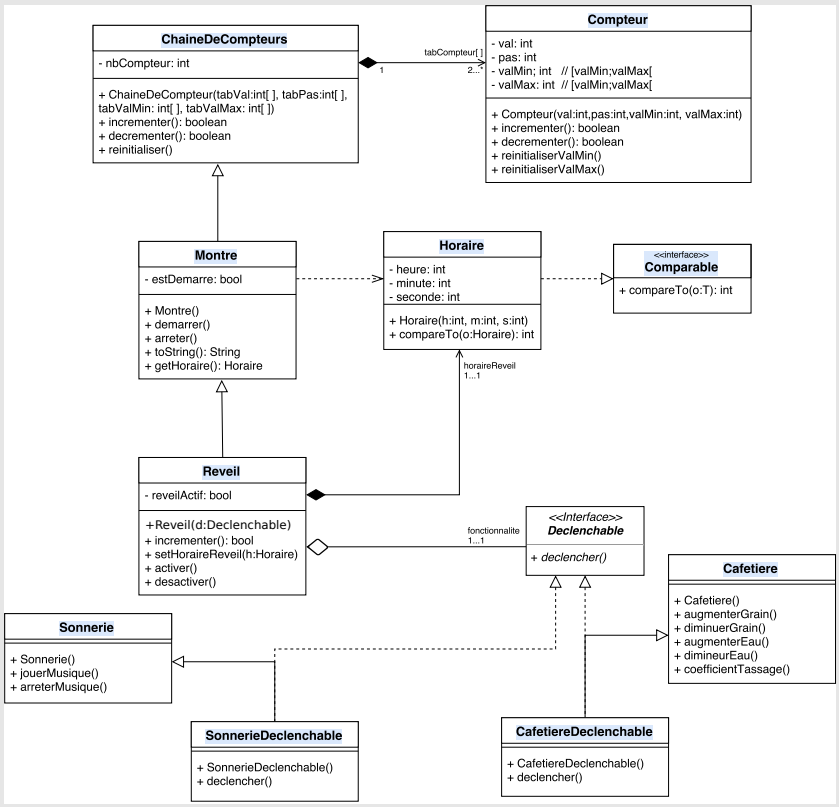
```
public class Reveil{
   private boolean reveilActif ;
   private Horaire horaireReveil ;
       /* Ce Declenchage est initialisé dans le constructeur
        * avec un paramètre de ce type (objet de type Sonnerie
        * ou Cafetière ) */
    private Declenchable fonctionnalite;

     public Reveil(Declenchable d){
     super();//appel du constructeur de la classe mère
     this.reveilActif = false;
     this.horaireReveil = new Horaire(0,0,0);
     this.fonctionnalite = d;
    }

    //Surcharge de la méthode incrementer()
    public boolean incrementer() {
        super.incrementer()
        if(reveilActif && this.getHoraire.compareTo(this.horaireReveil) == 0) {
            fonctionnalite.declencher();
        }
    }
}
```

Pour instancier un Réveil, un upcast sera nécessaire : le type apparent de la fonctionnalité est `Declenchable` mais le typé réel est `Sonnerie` ou `Cafetiere`.

```
Reveil r = new Reveil ( new  Sonnerie() );
```

À la vue de notre récent succès, nous comptons réutiliser les classes correspondantes aux différentes fonctionnalités développées dans différents projets, tels que nos cafetières-PC (la chaleur des composants permet de chauffer l'eau) et nos meubles d'ascenseur (Qui sonnent ou joue de la musique en permanence). Il faut donc réussir à isoler le concept de Sonnerie et de Cafetière de l'aspect déclenchable.

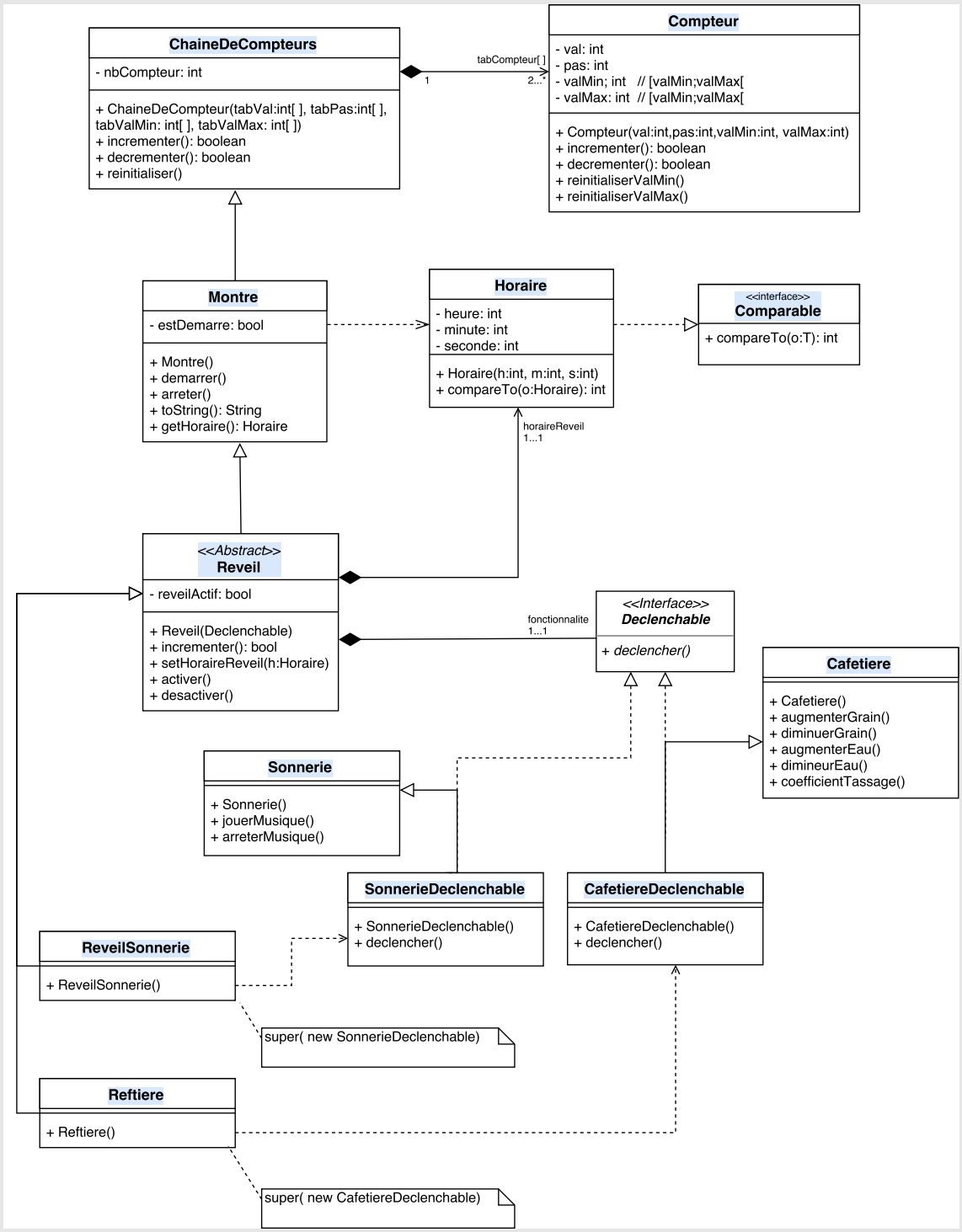
**Question 3** *Proposer une modification du diagramme de classes afin de faciliter ces réutilisations futures.*




Nous avons optimisé nos lignes de productions et ne laissons désormais plus le choix de la fonctionnalité des réveils. Soit ce sont des Reftières®, soit des réveils à sonneries tristes et banales.

## Question 4

*Proposer une modification du diagramme de classes permettant de limiter la création de réveils à ces deux types, puis proposer une implémentation en pseudo-code Java des nouveaux constructeurs.*




On a ici du surclassement (upcast) dans le constructeur de `ReveilSonnerie` : une instance de type réel `SonnerieDeclenchable` est mise dans une variable de type déclaré `Declenchable`. De même pour le constructeur de `Reftiere`.