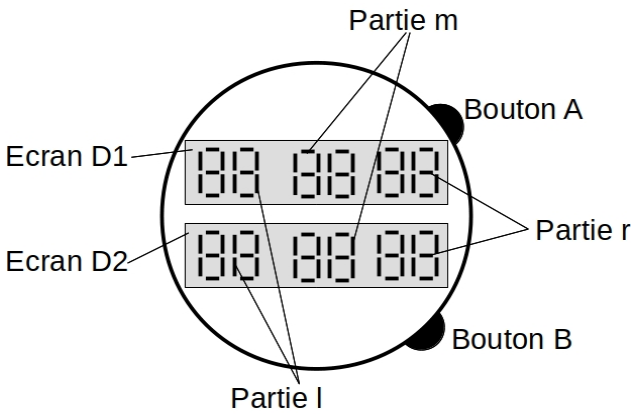
TD 8 : Programmation d'une montre digitale

Objectifs pédagogiques :

- ***Design Pattern Strategy***
- ***Design Pattern Decorator***
- ***Combinaisons de Design Pattern***

L'objectif est de programmer le contrôleur d'une montre digitale. La montre possède deux boutons, A et B. Un appui sur un de ces boutons générera un appel de méthode sur le contrôleur. La montre possède également deux écrans, D1 et D2, que le contrôleur peut mettre à jour. Chaque écran affiche trois nombres : `l` à gauche (*left*), `m` au milieu (*middle*) et `r` à droite (*right*). Le mécanisme a une horloge externe qui génère des ticks réguliers, c'est à dire appelle une méthode de notre contrôleur, toutes les 0.01 secondes.

Voici le schéma de la montre :



Le contrôleur de la montre obéit à l'interface `Watch` suivante :

```
package pobj.watch;

public interface Watch {
   public void tick(); // appelé à chaque interruption horloge
   public void pushA(); // ce qu'on doit faire quand A est appuyé
   public void pushB(); // ce qu'on doit faire quand B est appuyé
   public Display d1(); // obtenir la référence sur l'écran D1
   public Display d2(); // obtenir la référence sur l'écran D2
}
```

La classe `Display` permet de contrôler l'affichage d'un écran et est donnée en annexe. La classe `CsTimeCounter`, également donnée en annexe, est un utilitaire permettant de compter des ticks d'horloge et de convertir ce nombre de centièmes de secondes en un temps exprimé en heures, minutes et secondes.

Question 1. Écrivez une classe abstraite `AbstractWatch` qui implante partiellement l'interface `Watch` avec les méthodes `d1()` et `d2()`. Cette classe contient un constructeur sans argument et deux attributs de type `Display` (pour les écrans D1 et D2) instanciés lors de la construction.

Nous proposons ci-dessous une classe concrète de contrôleur, `SimpleWatch`, qui hérite d'`AbstractWatch` de la question précédente :

```
package pobj.watch;

import java.util.Calendar;
import java.util.TimeZone;
```

```java

public class SimpleWatch extends AbstractWatch {

   private boolean showDate = false;
   private CsTimeCounter clock = new CsTimeCounter(); // compteur de ticks de l'horloge de la montre

   public SimpleWatch() {
      refreshDisplay();
   }

   public void refreshDisplay() {
      d1().set(clock.getHour(), clock.getMinute(), clock.getSecond());
      if (showDate) {
         Calendar c = Calendar.getInstance(TimeZone.getDefault());
         d2().set(c.get(Calendar.DAY_OF_MONTH), c.get(Calendar.MONTH) + 1, c.get(Calendar.YEAR));
      } else {
         d2().turnOff();
      }
   }

   public void tick() {
      clock.incr();
      refreshDisplay();
   }

   public void pushA() {
      showDate = !showDate;
   }

   public void pushB() {
   }

}
```

Question 2. Que fait le contrôleur `SimpleWatch` ?

Question 3. Nous souhaitons produire une montre permettant d'avoir un mode chronomètre. Pour cela, nous souhaitons instancier le Design Pattern Strategy sur cet exemple, de façon à ce que le mode montre (de `SimpleWatch`) soit une stratégie particulière, à laquelle on puisse substituer une autre stratégie.

On propose l'interface `WatchStrategy` suivante :

```java
package pobj.watch.strategy;

import pobj.watch.Watch;

public interface WatchStrategy {
   public void tick();
   public void refreshDisplay(Watch w);
   public void pushA(Watch w);
   public void pushB(Watch w);
}
```

Appliquez le Design Pattern Strategy en introduisant une nouvelle classe concrète `WatchWithStrategy` qui a en attribut une `WatchStrategy`. Donnez également le code de la stratégie concrète `ClockStrategy`. La configuration de `WatchWithStrategy` avec une `ClockStrategy` doit produire exactement le même effet que `SimpleWatch`.

Question 4. Construisez à présent une stratégie concrète `ChronoStrategy` qui correspond au comportement d'un chronomètre.

En mode chronomètre, l'écran D1 affiche les heures, les minutes et les secondes écoulées depuis que l'on a lancé le chronomètre. L'écran D2 affiche sur la partie `r` les centièmes de seconde (les parties `l` et `m` ne servent à rien et affichent 0).

Le bouton A permet de démarrer (ou reprendre) le chronomètre. S'il est déjà démarré celà le stoppe.

Le bouton B permet de réinitialiser à 0 le chronomètre lorsque celui-ci est arrêté. Lorsque le chronomètre est démarré, il permet alternativement de figer la valeur du chronomètre sur l'écran (sans l'arrêter) et de reprendre son affichage réel.

Expliquez comment instancier une montre qui se comporte comme un chronomètre.

Question 5. On souhaite à présent étendre la fonctionnalité pour permettre de construire une montre qui intègre à la fois le mode heure et le mode chronomètre (et peut-être d'autres modes). Pour cela, nous commençons par ajouter un bouton de changement de mode. L'interface `Watch` est enrichie d'une méthode `pushMode()`, qui sera appelée quand ce bouton est appuyé. Un appui sur le bouton indiquera au contrôleur de passer du mode heure classique au mode chronomètre. Un nouvel appui permettra de passer à nouveau en mode heure.

Proposez une classe `WatchMultiStrategy` qui porte une liste de stratégies, et bascule de l'une sur la suivante (circulairement) quand on appuie sur le bouton mode. Le « tick » doit maintenant être répercuté sur toutes les stratégies ; cependant, seule la stratégie active doit rafraîchir l'affichage.

Expliquez comment instancier une montre qui possède à la fois une horloge et un chrono. Est-ce facile ou difficile d'ajouter un deuxième fuseau horaire à la montre ?

Question 6. On souhaite à présent permettre de modifier certaines stratégies à l'aide du Design Pattern Decorator.

Par exemple, dans une montre « cassée », le bouton A est inopérant. Dans une montre « lente », un tick sur 10 est perdu. Dans une montre « clignotante », l'affichage est éteint un tick sur deux.

Décrivez les élements permettant la mise en place du Design Pattern et le code de la classe `WatchStrategyDecorator` abstraite.

Donnez ensuite le code des trois décorateurs concrets proposés : `BrokenAButton`, `SlowTick` et `Blinker`.

Question 7. A-t-on besoin de modifier la classe `WatchMultiStrategy` ou `WatchWithStrategy` pour utiliser ces nouvelles décorations ?

Question 8. Peut-on combiner ces décorations ? Comment construire une montre qui ne rafraichit l'affichage qu'une fois sur 4 ? Peut-on dynamiquement (au cours de l'exécution du programme) décider que le bouton A est maintenant cassé ?

Annexe

```
package pobj.watch;

public class Display {

   private int l;
   private int m;
   private int r;

   private boolean on = false;

   public boolean isOn() {
      return on;
   }

   public void turnOff() {
      this.on = false;
   }

   public void set(int l, int m, int r) {
```

```
      this.on = true;
      this.l = l;
      this.m = m;
      this.r = r;
   }

   public int getL() {
      return l;
   }

   public int getM() {
      return m;
   }

   public int getR() {
      return r;
   }

}
```

```
package pobj.watch;

public class CsTimeCounter {

   private int cs = 0;
   private int s = 0;
   private int m = 0;
   private int h = 0;

   public void setHour(int h) {
      this.h = (h % 24);
   }

   public void setMinute(int m) {
      this.m = (m % 60);
   }

   public void reset() {
      cs = 0;
      s = 0;
      m = 0;
      h = 0;
   }

   public void incr() {
      cs++;
      if (cs == 100) {
         cs = 0;
         s++;
         if (s == 60) {
            s = 0;
            m++;
            if (m == 60) {
               m = 0;
               h = (h + 1) % 24;
            }
         }
      }
   }

   public boolean isReset() {
```

```
      return h == 0 && m == 0 && s == 0 && cs == 0;
   }

   public int getHour() {
      return h;
   }

   public int getMinute() {
      return m;
   }

   public int getSecond() {
      return s;
   }

   public int getCentiSecond() {
      return cs;
   }
}
```