Faculté SCIENCES ET TECHNIQUES Le Mans

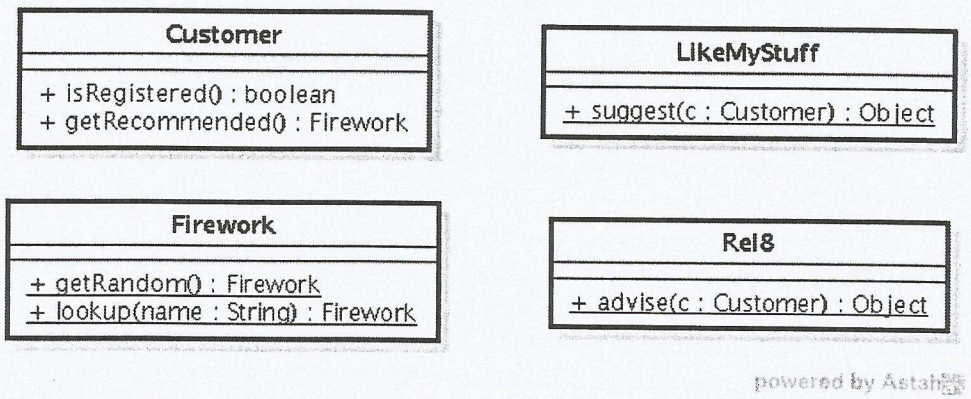
Master STS Mention Informatique Parcours ISI
Unité 178EN004
Génie Logiciel et Conception Avancée

TD1

Exercice n°1 : recommandations de feux d'artifices

Examinons les principes de publicité suivis par la société Oozinoz pour suggérer un feu d'artifice lorsque le client visite le site Web de la société ou lorsqu'il contacte la société par téléphone.

Oozinoz utilise deux moteurs de recommandations commerciales pour aider à choisir le feu d'artifice à un client. La classe Customer choisit et applique un de ces moteurs afin de décider sur quel feu d'artifice la recommandation au client va se faire.



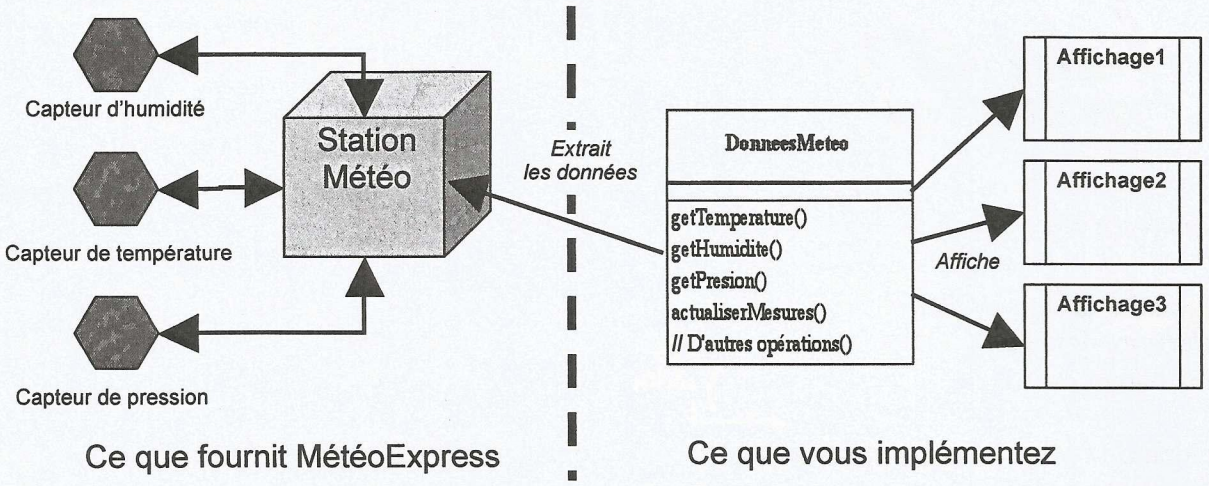
L'un des moteurs de recommandation, le moteur Rel8, suggère un achat basé sur la similarité du client par rapport à d'autres clients. Pour cette recommandation, le client doit être inscrit et avoir donné des informations sur ses goûts concernant les feux d'artifice et autres divertissements. Si le client ne s'est pas déjà inscrit, Oozinoz utilise LikeMyStuff, un autre moteur qui suggère un achat en fonction des achats récents du client. Si les données sont insuffisantes pour activer l'un des deux moteurs précédents, on prend un feu d'artifice au hasard. Cependant, une promotion spéciale peut remplacer toutes ces considérations pour promouvoir un feu d'artifice spécifique que Oozinoz veut vendre.

Quelle solution apportée ? Identifiez quel GoF pattern on pourrait utilement utiliser et appliquez-le.

Exercice n°2 : la société *MétéoExpress*

Votre équipe vient de remporter le marché de la construction de la station météorologique de dernière génération, consultable en ligne de MétéoExpress. La station sera basée sur l'objet DonneesMeteo (brevet en cours) qui enregistre les conditions météorologiques à un moment donné (température, hygrométrie et pression atmosphérique).

La société aimerait que vous créiez une application qui fournira d'abord trois affichages : conditions actuelles, statistiques et prévisions simples, tout trois mise à jour en temps réel au fur et à mesure que l'objet DonneesMeteo acquiert les données les plus récentes.



De plus, cette station météo doit être *extensible*. MétéoExpress veut commercialiser une API pour que les autres *développeurs puissent réaliser leurs propres affichages* et les insérer directement.

Votre tâche consiste à implémenter l'opération *actualiserMesures* pour qu'elle mette à jour les trois affichages. Cette méthode est appelée (vous ne savez pas comment mais ce n'est pas votre problème) à chaque fois qu'une nouvelle mesure est disponible. Il existe également trois méthodes d'accès aux dernières valeurs mesurées.

1ère solution envisagée :

- Ajouter simplement notre code à l'opération *actualiserMesures*.
- Défaut : pour ajouter de nouveaux types d'affichage, nous sommes obligés de modifier notre API pour chaque nouveau type d'affichage, ce qui est problématique puisque les utilisateurs de l'API non pas accès aux codes sources de l'API. Ils ne peuvent donc pas ajouter de nouveaux affichages par eux-mêmes.

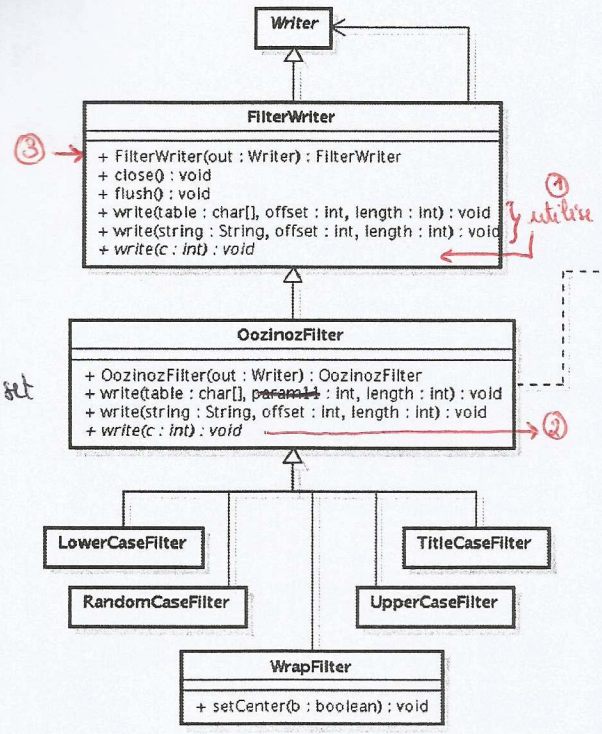
```
public class DonneesMeteo {
        ...
        public void actualiserMesures() {
                float temp = getTemperature();
                float humidite = getHumidite();
                float pression = getPression();

                affichageConditions.actualiser(temp, humidite, pression);
                affichageStats.actualiser(temp, humidite, pression) ;
                affichePrevisions.actualiser(temp, humidite, pression) ;
        }
        ...
}
```

Pour résoudre le problème, il faut de nouveau encapsuler le point de variation. Il s'agit ici de l'actualisation des différents affichages. Quelle solution apportée ? Identifiez quel GoF pattern on pourrait utilement utiliser et appliquez-le.

Exercice n°3 : filtres textuels

A partir du diagramme de classe donné ci-après ainsi que des extraits de code Java, expliquez quels designs patterns ont été mis en œuvre et quels problèmes de conception cela permet de résoudre.




```
import java.io.*;

public abstract class OozinozFilter extends FilterWriter {
   protected OozinozFilter(Writer out) {
      super(out);
   }

   public void write(char table[], int offset, int length) throws IOException {
      for (int i = 0; i < length; i++) write(table[i]);
   }

   public void write(String string, int offset, int length) throws IOException {
      write(string.toCharArray(), offset, length);
   }

   public abstract void write(int c) throws IOException;
}
```

```
import java.io.*;

public class ShowFilters {

   public static void main(String args[]) {
      BufferedReader in = new BufferedReader(new FileReader(args[0]));

      Writer out = new FileWriter(args[1]);
      out = new WrapFilter(new BufferedWriter(out), 40);
      out = new TitleCaseFilter(out);

      while (true) {
         String s = in.readLine();
         if (s == null) break;
         out.write(s + "\n");
      }
      out.close();
      in.close();
   }

}
```

demo.txt

```
The "SPACESHOT" shell      hovers
          at 100 meters for 2 or 3
minutes,          erupting star bursts    every 10 seconds that
generate                abundant reading-level light for a
typical     stadium
```

```
> java -classpath ./oozinoz/classes ShowFilters demo.txt demo-out.txt
> cat demo-out.txt

The "Spaceshot" Shell Hovers At 100
Meters For 2 Or 3 Minutes, Erupting Star
Burts Every 10 Seconds That Generate
Abundant Readling-level Ligth For A
Typical Stadium
```

Donnez le code source de la classe RandomCaseFilter.

Exercice n°4 : l'usine à chocolat

Chacun sait que toutes les usines à chocolat modernes ont des bouilleurs assistés par ordinateur. La tâche du bouilleur consiste à contenir un mélange de chocolat et de lait, à le porter à ébullition puis à le transmettre à la phase suivante où il est transformé en plaquettes.

Le code suivant du contrôleur du bouilleur montre bien qu'il faut impérativement éviter toutes catastrophes du genre remplir le bouilleur alors qu'il n'est pas vide, vider alors que le mélange n'a pas bouilli, etc.

```
public class BouilleurChocolat {
        private boolean vide ;
        private boolean bouilli ;

        public BouilleurChocolat() {
                vide = true ;
                bouilli = false ;
        }

        public void remplir() {
                if (estVide()) {
                        vide = false ;
                        bouilli = false ;
                }
                // Remplir avec le mélange...

        }

        public void vider() {
            if ( !estVide() && estBouilli()) {
                    // Vider le mélange bouili...
                    vide = true ;
                    bouilli = false ;
            }

        public void bouillir() {
            if ( !estVide() & !estBouilli()) {
                    // Chauffe Marcel...
                    bouilli = true ;
            }
        }
        ...
}
```

Mettez vous à la place du patron de la société AbonChocoCoco et imaginez ce qu'il pourrait se passer si deux ou plusieurs instances de *BouilleurChocolat* étaient lancées en même temps.

Quelle solution apportée ? Identifiez quel GoF pattern on pourrait utilement utiliser et appliquez-le en modifiant le code de la classe *BouilleurChocolat*.