La Rochelle Université

La Rochelle Université
Licence Informatique
UE Génie logiciel 2

TP 2 - Cas d'étude "*Magasin*"
Application des design patterns Observer et Decorateur

1 Dépôt

Votre travail, individuel, devra être déposé sur Moodle. Votre dépôt contiendra un fichier PDF contenant votre diagramme de classes. Votre dépôt contiendra également un sous répertoire *src* contenant votre code Java.

2 Retour sur le TP1

Le diagramme de classes de la figure 1 représente une utilisation des design patterns Fabrique et Visiteur sur le cas d'étude Magasin vu la semaine dernière. Vous remarquerez par ailleurs l'utilisation de différents packages pour chacun d'eux.

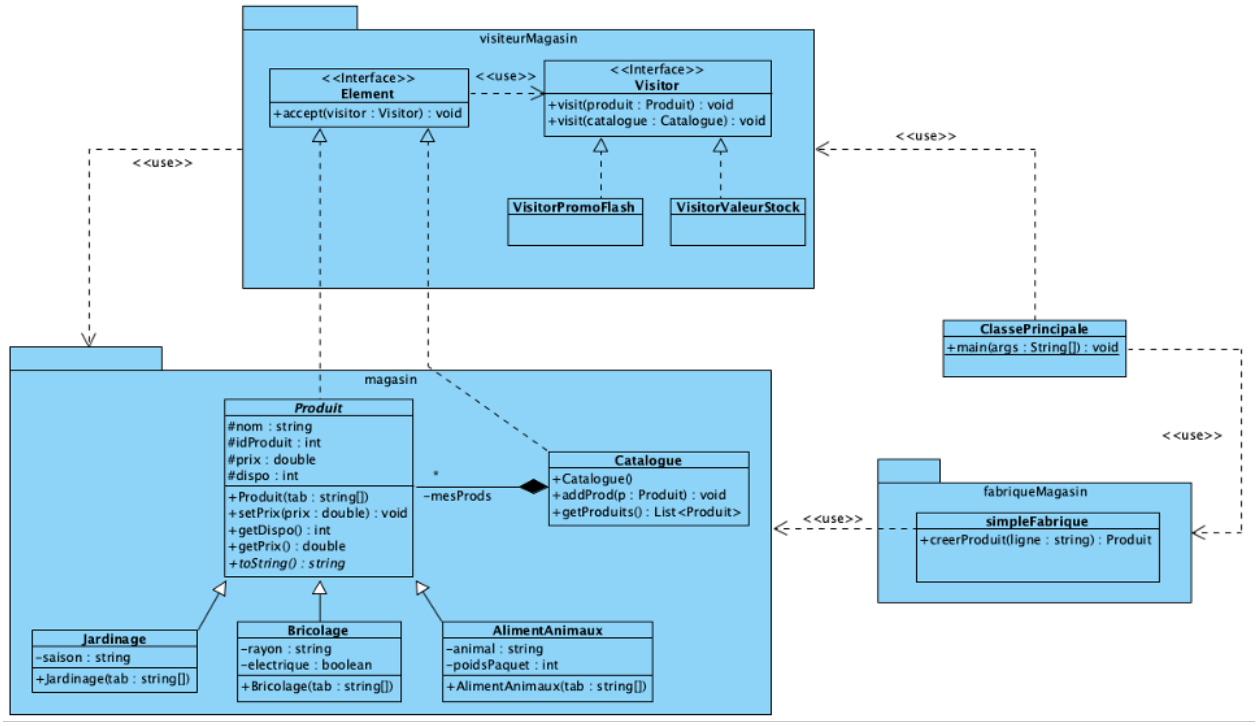


FIGURE 1 – Diagramme de classes intégrant les design patterns Fabrique et Visiteur pour le cas d'étude Magasin

3 Design pattern Observer et sauvegarde automatique

On souhaite déclencher une sauvegarde du fichier de catalogue dès que l'un des produits est modifié (changement du prix, de quantité en stock ou de rayon par exemple). Le pattern le plus proche de ce besoin est le pattern *observer*.

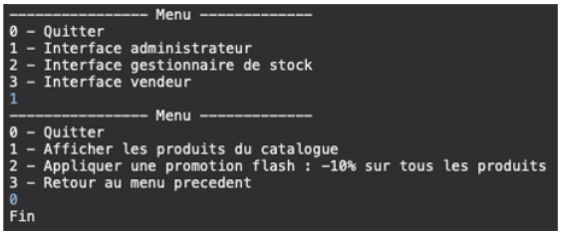
— Dans l'exemple du magasin, quelle classe doit être l'observable et quelle classe doit être l'observer ?
— Ajoutez les éléments de ce design pattern au diagramme de classes du magasin ;
— Implémentez ce pattern en java sur le code développé au TP précédent.

4 Design pattern Decorateur et IHM textuelle

On souhaite proposer une IHM (un menu principal textuel) permettant aux utilisateurs d'accéder à des fonctionnalités spécifiques en fonction de leur statut :

— Tous les utilisateurs peuvent afficher la liste des produits du catalogue ;
— L'administrateur peut appliquer une promotion flash ;
— Le gestionnaire de stock peut calculer la valeur globale du stock, générer un fichier HTML du catalogue.

L'exécution peut donc se dérouler selon les deux scénarios suivants :



— En vous inspirant de la description du design pattern décorateur vu en cours dont un résumé est donné en annexe, proposez une intégration de ce design pattern dans le diagramme de classes précédent pour traiter cette interface textuelle. Vous ajouterez pour cela un nouveau package ;
— Implémentez ce pattern en java sur le code développé ;
— On souhaite ajouter un nouveau type d'utilisateur : le vendeur. Ce dernier peut vendre un produit (ce qui diminuera la quantité d'une unité en stock). Ajoutez au diagramme de classes les éléments nécessaires et faites la modification sur le code.

```
---------------- Menu -------------
0 - Quitter
1 - Interface administrateur
2 - Interface gestionnaire de stock
3 - Interface vendeur
3
---------------- Menu -------------
0 - Quitter
1 - Afficher les produits du catalogue
2 - Vendre un produit
3 - Retour a l'ecran precedent
2
        0- Retour
        1- Vendre Rateau
        2- Vendre Boite clous
        3- Vendre Croquettes au poulet
1
Changement pris en compte pour  Produit de jardinage -> Identifiant: Rat345; Prix: 50.0; Quantite: 11; Saison: Ete

Catalogue sauvegarde
        0- Retour
        1- Vendre Rateau
        2- Vendre Boite clous
        3- Vendre Croquettes au poulet
0
---------------- Menu -------------
0 - Quitter
1 - Afficher les produits du catalogue
2 - Vendre un produit
3 - Retour a l'ecran precedent

0
Fin
```

Annexe 1 : Le pattern Observer

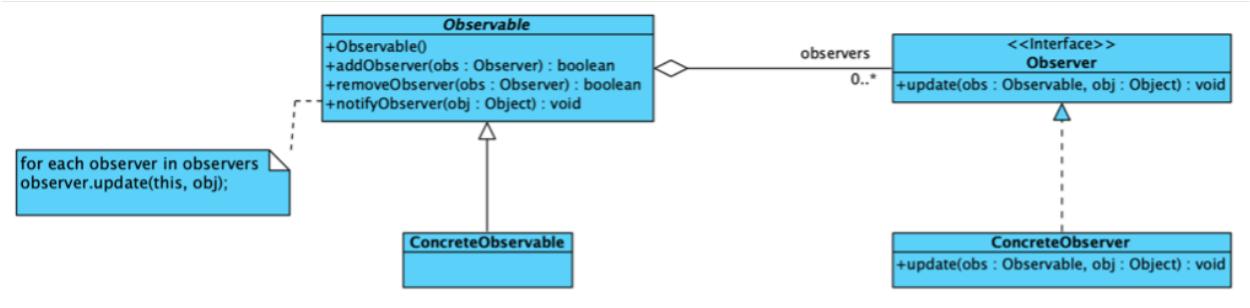


FIGURE 2 – Design pattern Observer

```java
public class Entreprise extends Observable {
   private int id;
   private boolean etat;

   public void setEtat(boolean etat) {
      this.etat = etat;
      notifyObservers(this.etat);
   }

   public boolean isEtat() {
      return etat;
   }

   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String toString() {
      return "Entreprise (id=" + id + " etat=" + etat + ")";
   }
}
```

```java
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Date;
import java.util.Locale;

public class Etablissement implements Observer {
   private int id;
   private String infos;

   public void update(Observable obs, Object obj) {
      if (obs instanceof Entreprise) {
         Calendar calendar = new GregorianCalendar(Locale.FRANCE);
         Date date = calendar.getTime();
```

```
            setInfos("Notification recue le " + new SimpleDateFormat().format(date));
        }
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getInfos() {
        return infos;
    }

    public void setInfos(String infos) {
        this.infos = infos;
    }

    public String toString() {
        return "Etablissement (id=" + id + "; infos : " + infos + ")";
    }
}
```

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args) {
        Logger log = Logger.getLogger("log");

        Etablissement etab = new Etablissement();
        etab.setId(1);
        etab.setInfos("RAS");

        Entreprise ent = new Entreprise();
        ent.addObserver(etab);
        log.log(Level.INFO, "Avant mise à jour de l''etat de l''entreprise: etab: {0}", etab);
        ent.setEtat(true);
        log.log(Level.INFO, "Après mise à jour de l''etat de l''entreprise:etab: {0}", etab);
    }
}
```

Sortie console :
INFO : Avant mise à jour de l'etat de l'entreprise : etab : Etablissement (id=1 ; infos : RAS)
INFO : Après mise à jour de l'etat de l'entreprise :etab : Etablissement (id=1 ; infos : Notification recue le 02/11/2021 16 :58)

Annexe 2 : Le pattern Decorateur

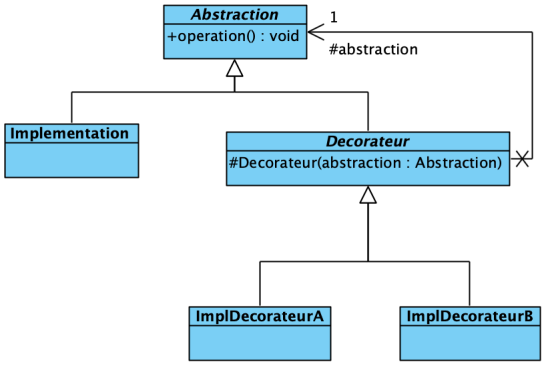


FIGURE 3 – Design pattern Decorateur

```
public abstract class Abstraction {
   public void operation() {
   }
}
```

```
public class Implementation extends Abstraction {
   @Override
   public void operation() {
      Logger log = Logger.getLogger("log");
      log.log(Level.INFO, "Implementation");
   }
}
```

```
public abstract class Decorateur extends Abstraction{
   protected Abstraction abstraction;

   protected Decorateur(final Abstraction abstraction) {
      this.abstraction = abstraction;
   }
}
```

```
public class ImplDecorateurA extends Decorateur{
   public ImplDecorateurA(Abstraction abstraction) {
      super(abstraction);
   }

   @Override
   public void operation() {
      Logger log = Logger.getLogger("log");
      log.log(Level.INFO, "ImplDecorateurA avant");
      this.abstraction.operation();
      log.log(Level.INFO, "ImplDecorateurA apres");
   }
}
```

```
public class ImplDecorateurB extends Decorateur{
   public ImplDecorateurB(Abstraction abstraction) {
```

```
      super(abstraction);
   }

   @Override
   public void operation() {
      Logger log = Logger.getLogger("log");
      log.log(Level.INFO, "ImplDecorateurB avant");
      this.abstraction.operation();
      log.log(Level.INFO, "ImplDecorateurB apres");
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      final Implementation impl = new Implementation();
      final ImplDecorateurA implDecA = new ImplDecorateurA(impl);
      final ImplDecorateurB implDecB = new ImplDecorateurB(implDecA);

      implDecB.operation();
   }
}
```

Sortie console :
INFO : ImplDecorateurB avant
INFO : ImplDecorateurA avant
INFO : Implementation
INFO : ImplDecorateurA apres
INFO : ImplDecorateurB apres