TP N° 2

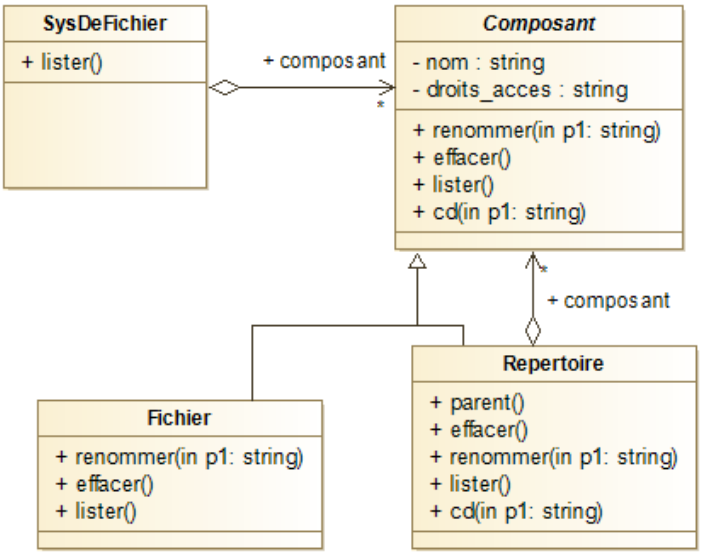
Architecture et Développement Logiciel
Patterns de Structure et de comportement

Préparé par :

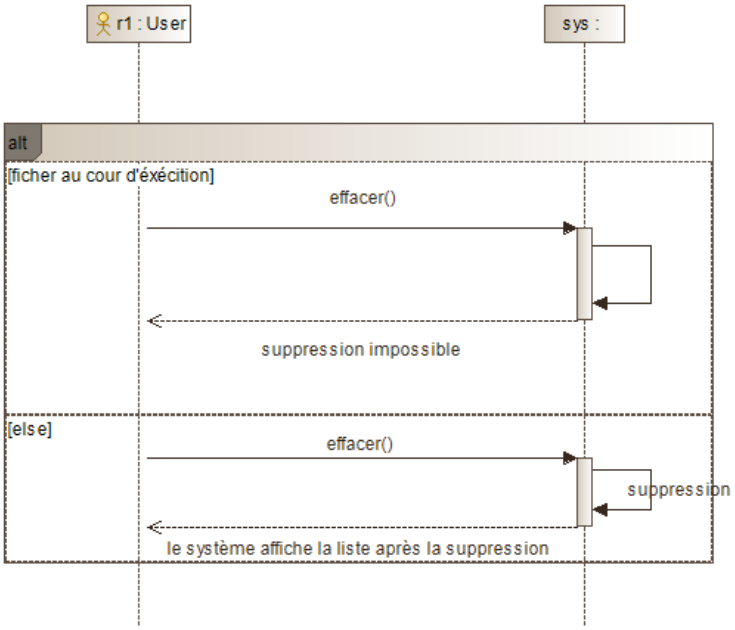
HAMMOU TRARI Bouchra Ahlem
M1 GL

Exercice 1 :

1- conception en utilisant les diagrammes UML et le design pattern composite :



2- le diagramme de séquence de l'effacement d'un fichier et d'un répertoire :



3- L’implémentation :

- Premièrement on crée l'interface qui sera implémentée par toutes les autres classes :

```
public abstract class Composant {

    private String nom;
    private String droits_acces;

    public abstract void renommer(String nom);
    public abstract void effacer();
    public abstract void lister();
    public abstract void cd(String nom);

}
```

- Ensuite on crée les différentes classes que l'on pourra composer :

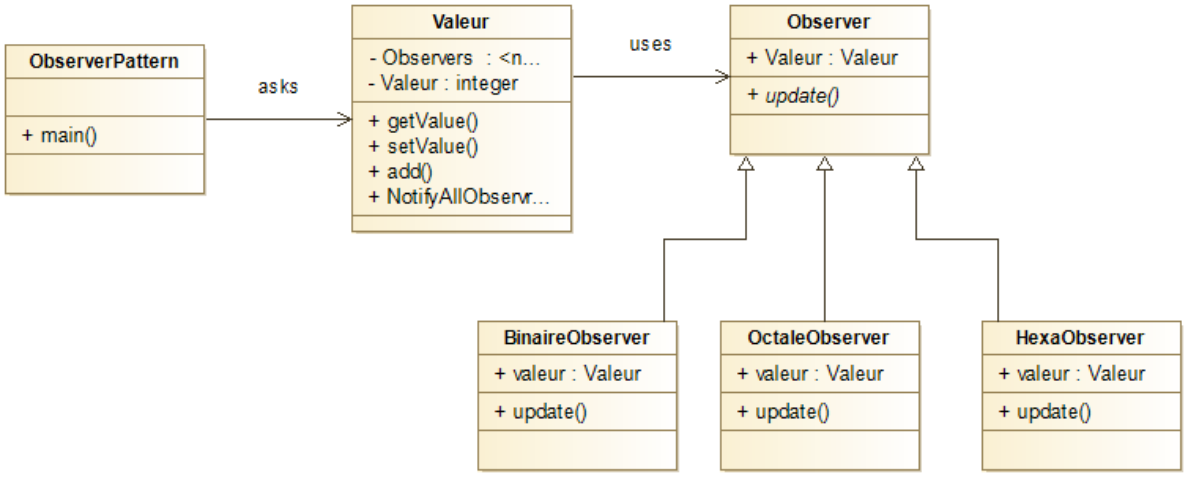
```
public class Fichier extends Composant{

    private String nom;
    private String droits_acces;

    public Fichier(String nom, String droits_acces) {
        this.nom = nom;
        this.droits_acces = droits_acces;
    }

    @Override
    public void renommer(String nom) {
        this.nom = nom;
    }

}
```

- Puis, on implémente la classe composite RepertoireComposite, avec les méthodes add et remove qui permettront d'ajouter ou de supprimer plusieurs éléments à une composition :

```
public class RepertoireComposite extends Composant {
    private String nom;
    private String droits_acces;
    private Collection list;
    public RepertoireComposite(String nom, String da) {
       this.nom = nom;
       this.droits_acces = da;
       this.list = new ArrayList();
    }
     public void add(Composant composant){
        list.add(composant);
    }
    public void effacer(Composant composant){
        list.remove(composant);
    }
    @Override
    public void renommer(String nom) {
        this.nom = nom;
    }
    public void lister() {
        Iterator iterator = list.iterator();
        System.out.println("List elements : ");
        while (iterator.hasNext())
            System.out.print(iterator.next() + " ");
        System.out.println();
    }
```

Exercice 2 :

1- conception en utilisant le design pattern observateur :



2- L'implémentation :

Le modèle Observateur utilise trois classes d'acteurs. Valeur, Observer et le client.

Sujet est un objet ayant des méthodes pour attacher et détacher des observateurs à un objet client. J'ai créé une classe abstraite Observer et une classe concrète Valeur qui étend la classe Observer.

ObserverPattern, ou ce trouve le main, utilisera Valeur et un objet de classe concret pour afficher l'observateur en action.

- Création de la classe Valeur :

Les observateur seront notifier si la valeur est modifier donc quand la méthode setValeue(int v) est déclencher.

Un observateur est attaché à la liste des observateurs par la méthode add(Observer o) .

La méthode notifyAllObserves déclenche la modification des obsevateurs.

```java
public class Valeur{
    private List<Observer> observers = new ArrayList<Observer>();
    private int value;

    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
        notifyAllObservers();
    }
    public void add(Observer observer){
        observers.add(observer);
    }
    public void notifyAllObservers(){
        for (Observer observer : observers) {
            observer.update();
        }
    }
}
```

- Création de la classe abstraite Observer :

```java
public abstract class Observer {
    protected Valeur valeur;
    public abstract void update();
}
```

La méthode update() sera implémenter dans les sous classe de Observer.

- Création des sous classes de Observer :

```java
public class BinaireObserver extends Observer{
    public BinaireObserver(Valeur valeur){
        this.valeur = valeur;
        this.valeur.add(this);
    }

    @Override
    public void update() {
        System.out.println( "Binaire: " + Integer.toBinaryString( valeur.getValue() ) );
    }
}
```

```java
public class OctaleObserver extends Observer{
    public OctaleObserver(Valeur valeur){
        this.valeur = valeur;
        this.valeur.add(this);
    }

    @Override
    public void update() {
        System.out.println( "Octale: " + Integer.toOctalString( valeur.getValue() ) );
    }
}
```

```java
public class HexaObserver extends Observer{
    public HexaObserver(Valeur valeur){
      this.valeur = valeur;
      this.valeur.add(this);
    }

    @Override
    public void update() {
      System.out.println( "Hexa: " + Integer.toHexString( valeur.getValue() ).toUpperCase() );
    }
}
```

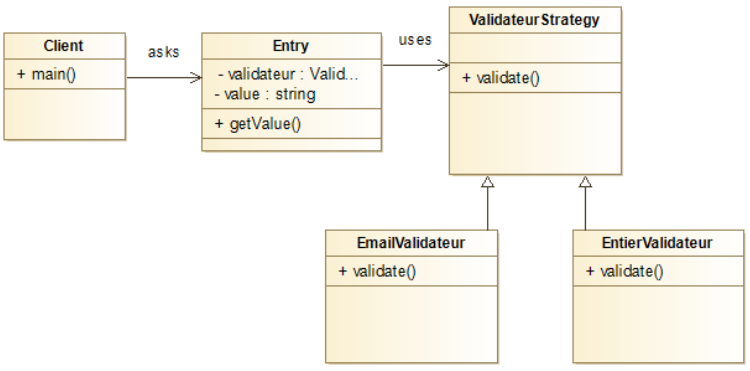
- Le main et l'exécution :

```java
public class ObserverPattern {

    public static void main(String[] args) {
      Valeur valeur = new Valeur();

        HexaObserver hexaObserver = new HexaObserver(valeur);
        OctaleObserver octaleObserver = new OctaleObserver(valeur);
        BinaireObserver binaireObserver = new BinaireObserver(valeur);

      System.out.println("Etat 1 : 15");
      valeur.setValue(15);
      System.out.println("Etat 2 : 8");
      valeur.setValue(10);
    }

}
```

```
Output - DP Compsite (run) | Test Results
Etat 1 : 15
Hexa: F
Octale: 17
Binaire: 1111
Etat 2 : 8
Hexa: A
Octale: 12
Binaire: 1010
BUILD SUCCESSFUL (total time: 2 seconds)
```

Exercice 3 :

1- La conception:



2- L’implémentation :

- On commence par l’interface Strategy, qui implémente la/les méthodes qui changeront. Ici, c'est la méthode validate() qu'on déclare.

```
public abstract class ValidateurStrategy {
    public abstract boolean validate(Entry e);
}
```

- On implémente ensuite les deux classes qui héritent de Strategy (EntierValidateur et EmailValidateur) :

```
public class EntierValidateur extends ValidateurStrategy{

    @Override
    public boolean validate(Entry e) {
        try
        {
            Integer.parseInt(e.getValue());
            return true ;
        }
        catch(NumberFormatException ex)
        {
            return false ;
        }
    }
```

```
import java.util.regex.Pattern;

public class EmailValidateur extends ValidateurStrategy {

    @Override
    public boolean validate(Entry e) {

        String emailRegex = "^[a-zA-Z0-9_+&*-]+(?:\\."+
                            "[a-zA-Z0-9_+&*-]+)*@" +
                            "(?:[a-zA-Z0-9-]+\\.)+[a-z" +
                            "A-Z]{2,7}$";
        Pattern pat = Pattern.compile(emailRegex);
         if (e.getValue() == null)
             return false;
         return pat.matcher(e.getValue()).matches();
    }
}
```

- Enfin, on implémente la classe Entry qui se servira du pattern :

```java
public class Entry {
    private String value;

    public Entry(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

}
```

- Le main et l’exécution :

```java
public class Client {
    public static void main(String[] args) {
        Entry entre1 = new Entry("contribute@geeksforgeeks.org");
        Entry entre2 = new Entry("contribute@geeksforgeeks");
        ValidateurStrategy validator = new EmailValidateur();
        if (validator.validate(entre1))
            System.out.println(entre1.getValue()+ " is : Valide");
        else
            System.out.println(entre1.getValue()+ " is : Invalide");

        if (validator.validate(entre2))
            System.out.println(entre2.getValue()+ " is : Valide");
        else
            System.out.println(entre2.getValue()+ " is : Invalide");

        Entry entre3 = new Entry("12");
        Entry entre4 = new Entry("3.33");
        ValidateurStrategy validator2 = new EntierValidateur();
        if (validator2.validate(entre4))
            System.out.println(entre4.getValue()+ " is : Valide");
        else
            System.out.println(entre4.getValue()+ " is : Invalide");
```

```
- DP Compsite (run)  ×  Test Results
run:
contribute@geeksforgeeks.org is : Valide
contribute@geeksforgeeks is : Invalide
3.33 is : Invalide
12 is : Valide
BUILD SUCCESSFUL (total time: 0 seconds)
```