Noursine ABDALLAH
Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

Séance 6

Cas n°1

Explication de la résolution du cas

Pour mettre en place cette solution, nous avons utilisé le patron de conception **Strategy**. Celui-ci nous permet de prendre une classe qui fait quelque chose de spécifique de plusieurs façons et de l'extraire dans différentes classes. Ici, nous avions besoin de simuler une authentification utilisant soit une base de données, soit un système de fichier.

Navigator

La classe **Navigator** permet de définir quel type d'authentification nous souhaitons utiliser. Ensuite, celui-ci utilisera son objet **Authenticator** afin de démarrer le processus d'authentification via la méthode **authenticate**.

```
public class Navigator {
    private Authenticator auth;

    public void setAuthenticator(Authenticator a)
    {
        auth = a;
    }

    public boolean authenticate(String username, String password)
    {
        return auth.authenticate(username, password);
    }
}
```

Authenticator

La classe **Authenticator** est une classe abstraite définissant le squelette des classes représentant les différents moyens de s'authentifier. Chacune de ses méthodes sont elles aussi définies comme étant abstraites. Cela signifie que les classes héritant d'**Authenticator** devront avoir au minimum ces trois méthodes, mais qu'elles devront avoir leur propre version de celles-ci.

```
public abstract class Authenticator {
    public abstract boolean authenticate(String username, String password);
    public abstract boolean isLogin(String username);
    public abstract boolean isPassword(String username, String password);
}
```

Vu la complexité de l'implémentation d'une base de données dans un projet, nous nous sommes contentés d'une **HashMap** afin de stocker les identifiants d'un utilisateur dans les deux classes représentant les deux moyens de s'authentifier.

DatabaseAuthenticator

```
public class DatabaseAuthenticator extends Authenticator {

    private HashMap<String,String> BD = new HashMap<>();

    public DatabaseAuthenticator()
    {
        BD.put("john","password");
    }
```

```
    @Override
    public boolean authenticate(String username, String password) {
        return isLogin(username) && isPassword(username,password);
    }

    @Override
    public boolean isLogin(String username) {
        if(BD.containsKey(username))
            return true;
        else
            return false;
    }

    @Override
    public boolean isPassword(String username, String password) {
        if (BD.get(username).equals(password))
            return true;
        else
            return false;
    }
}
```

FileAuthenticator

```
public class FileAuthenticator extends Authenticator{

    private HashMap<String,String> file = new HashMap<>();
    public FileAuthenticator()
    {
        file.put("johnny","passwordJ");
    }

    @Override
    public boolean authenticate(String username, String password) {
         return isLogin(username) && isPassword(username,password);
    }

    @Override
    public boolean isLogin(String username) {
        if(file.containsKey(username))
            return true;
        else
            return false;
    }

    @Override
    public boolean isPassword(String username, String password) {
        if(file.get(username).equals(password))
            return true;
        else
            return false;
    }
}
```

Comme nous avons fait un exemple simplifié, ces deux classes possèdent du code dupliqué. Une solution afin de pallier ce problème serait de remonter le code dans la classe mère, en retirant le mot clé **abstract** de la/les méthode(s) concernées.

Authenticator V.2

```
public abstract class Authenticator {
    public boolean authenticate(String username, String password)
    {
        return isLogin(username) && isPassword(username,password);
    }
    public abstract boolean isLogin(String username);
    public abstract boolean isPassword(String username, String password);
}
```

DatabaseAuthenticator V.2

```
public class DatabaseAuthenticator extends Authenticator {

    private HashMap<String,String> BD = new HashMap<>();


    //Entrées de la base de données
    public DatabaseAuthenticator()
    {
        BD.put("john","password");
    }

    @Override
    public boolean isLogin(String username) {
        if(BD.containsKey(username))
            return true;
        else
            return false;
    }

    @Override
    public boolean isPassword(String username, String password) {
        if (BD.get(username).equals(password))
            return true;
        else
            return false;
    }
}
```

FileAuthenticator V.2

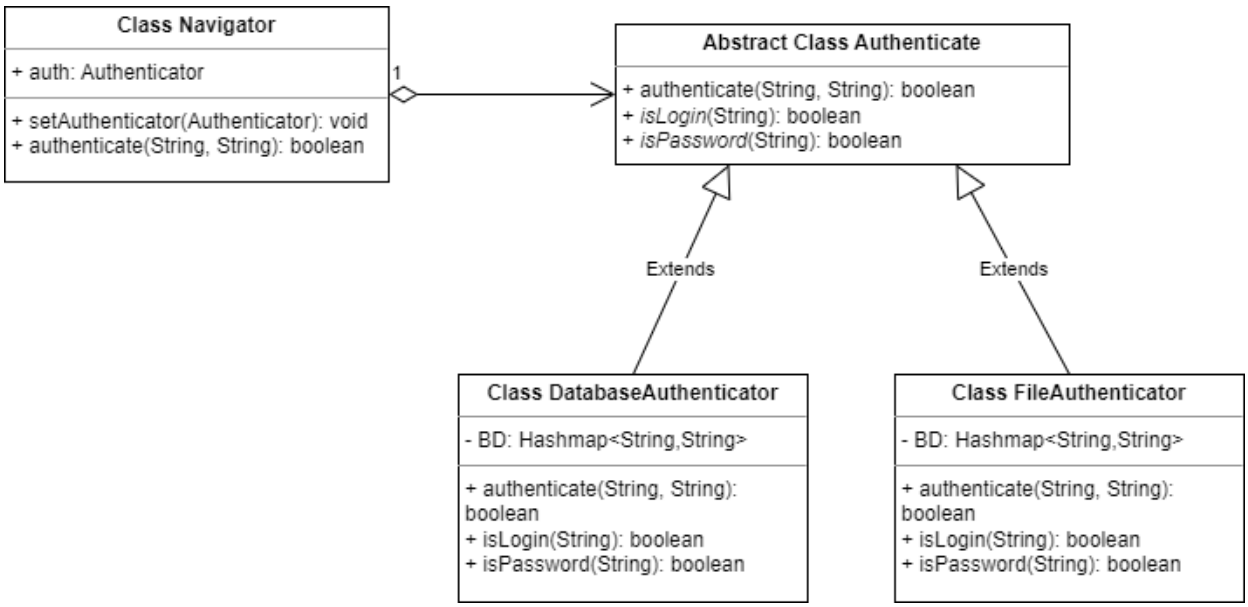
```
public class FileAuthenticator extends Authenticator{

    private HashMap<String,String> file = new HashMap<>();
    public FileAuthenticator()
    {
        file.put("johnny","passwordJ");
    }

    @Override     public boolean isLogin(String username) {
        if(file.containsKey(username))
            return true;
        else             return false;
    }

    @Override     public boolean isPassword(String username, String password) {
        if(file.get(username).equals(password))
            return true;
        else             return false;
    }
}
```

Diagramme



Source :
https://refactoring.guru/design-patterns/strategy

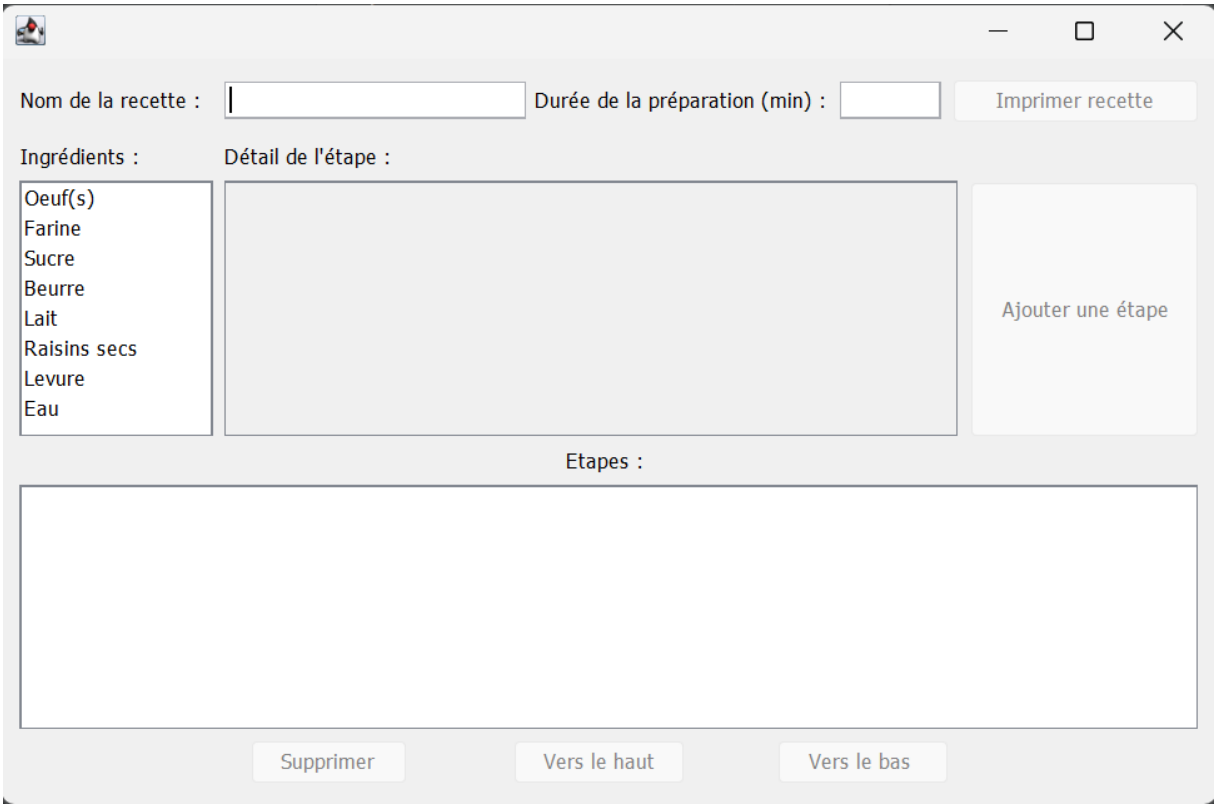
https://refactoring.guru/design-patterns/strategy/java/example

Cas n°2

Explication du cas

L'objectif de ce cas est de réaliser une application permettant la gestion d'une recette, cette application doit pouvoir interdire certaine action à l'utilisateur si une opération n'a pas été effectuée auparavant.

Voici l'interface réalisée pour cette application :



Comme on peut le constater certain boutons sont bloqués ;

- **Imprimer recette** : ce bouton est débloqué lorsqu'un nom de recette, une durée et au moins une étape est ajoutée.
- **Ajouter une étape** : ce bouton est débloqué lorsqu'un texte est entré dans le zone de texte « Détail de l'étape ».
- **Supprimer** : ce bouton est débloqué lorsqu'au moins une étape est ajoutée dans l'application.
- **Vers le haut** et **Vers le bas** : ces boutons sont débloqués lorsqu'au moins deux étapes sont ajoutées dans l'application.

Cette application possède donc une multitude de contrôle à réaliser pendant toutes son utilisation par l'utilisateur. Il a donc fallu trouver un moyen de gérer ça de la manière la plus simple possible.

Résolution du cas

Pour gérer le plus simplement possible cette interface l'utilisation d'un pattern a semblé particulièrement adapté : le pattern du médiateur.

Celui-ci va permettre à chaque modification de réaliser les contrôles nécessaires pour les modifications dans l'interface.

Un médiateur peut prendre plusieurs formes, soit une classe soit une méthode. Dans notre cas, il a semblé plus cohérent de réaliser une méthode dans la classe de la fenêtre afin d'avoir directement accès à toutes les références des éléments de l'application. Une classe pourrait être plus cohérente si les éléments pouvaient être débloqués à partir d'une autre fenêtre ou d'événement extérieur.

Le médiateur

```
1. private void Mediator(String source) {
2.     switch (source)
3.     {
4.         case "jButton1":
5.             for(int i=0; i<listModel2.getSize();i++)
6.             {
7.                 System.out.println(i +". "+listModel2.getElementAt(i));
8.             }
9.             System.out.println("La recette est imprimée !");
10.
11.            break;
12.        case "jButton2":
13.            String newLine = new String();
14.            List<String> alimentsList= jList1.getSelectedValuesList();
15.            if(!alimentsList.isEmpty())
16.            {
17.                String aliments = "(";
18.                for (String aliment:alimentsList) {
19.                    if(aliments.equals("("))
20.                        aliments = aliments + aliment;
21.                    else
22.                        aliments = aliments + "," + aliment;
23.                }
24.                newLine = aliments + ") ";
25.            }
26.            newLine = newLine + jTextPane1.getText();
27.            jTextPane1.setText("");
28.            jList1.clearSelection();
29.            listModel2.addElement(newLine);
30.            break;
31.        case "jButton3":
32.            int selectedIndex = jList2.getSelectedIndex();
33.            if(selectedIndex != -1)
34.                listModel2.remove(selectedIndex);
35.            jList2.clearSelection();
36.            break;
37.        case "jButton4":
38.            int index = jList2.getSelectedIndex();
39.            if(index==0)
40.                return;
41.            int newIndex = index-1;
42.            String temp = listModel2.getElementAt(index);
43.            listModel2.setElementAt(listModel2.getElementAt(newIndex), index);
44.            listModel2.setElementAt(temp, newIndex);
45.            jList2.clearSelection();
46.            break;
47.        case "jButton5":
48.            index = jList2.getSelectedIndex();
49.            if(index==listModel2.getSize()-1)
50.                return;
51.            newIndex = index+1;
52.            temp = listModel2.getElementAt(index);
53.            listModel2.setElementAt(listModel2.getElementAt(newIndex), index);
54.            listModel2.setElementAt(temp, newIndex);
55.            jList2.clearSelection();
56.            break;
57.        case "jList1":
58.            break;
59.        case "listModel2":
60.            if(listModel2.isEmpty())
61.            {
```

```
62.                 jButton1.setEnabled(false);
63.                 jButton3.setEnabled(false);
64.                 jButton4.setEnabled(false);
65.                 jButton5.setEnabled(false);
66.             }
67.             else
68.             {
69.                 jButton1.setEnabled(true);
70.                 jButton3.setEnabled(true);
71.                 if(listModel2.getSize()>1)
72.                 {
73.                     jButton4.setEnabled(true);
74.                     jButton5.setEnabled(true);
75.                 }
76.                 else
77.                 {
78.                     jButton4.setEnabled(false);
79.                     jButton5.setEnabled(false);
80.                 }
81.             }
82.             break;
83.         case "jList2":
84.             break;
85.         case "jTextField2":
86.         case "jTextField1":
87.             if(jTextField1.getText().length() > 0 && jTextField2.getText().length() > 0)
88.                 jTextPane1.setEnabled(true);
89.             else{
90.                 jTextPane1.setEnabled(false);
91.                 jTextPane1.setText("");
92.             }
93.             break;
94.         case "jTextPane1":
95.             if(jTextPane1.getText().length() > 0)
96.                 jButton2.setEnabled(true);
97.             else
98.                 jButton2.setEnabled(false);
99.             break;
100.     }
101. }
102.
```

Son fonctionnement est assez simple à comprendre, chaque modification, action ou événement produit dans l'application qui pourrait susciter un changement dans l'interface de l'application. Afin de rendre le travail du médiateur plus efficace, en appelant la méthode « Mediator », l'objet spécifie son nom en paramètre.

Avantage de ce pattern

En utilisant ce pattern, on rend l'application très facile à maintenir puisque la logique est centralisée en une seul classe/méthode. Il est donc très facile de changer un comportement ou d'en rajouter des nouveaux.

Source :

https://refactoring.guru/design-patterns/mediator