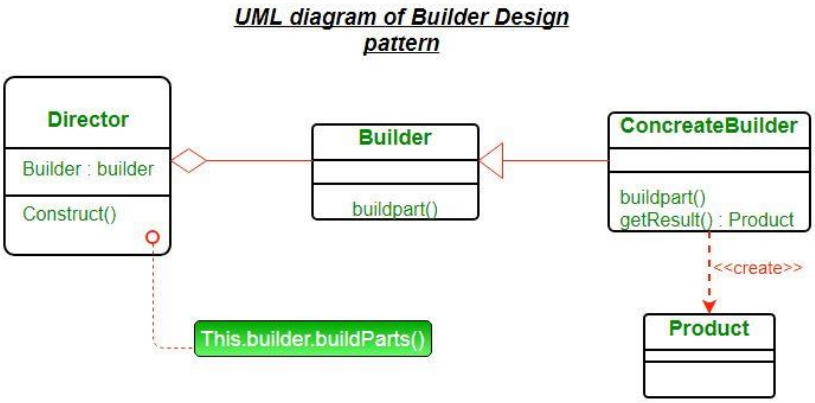
Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

Séance 5

Cas n°1

URIBuilder :

Identifier les « collaborateurs » :



- URIBuilder est le ConcreteBuilder du schéma. Il crée une URI grâce aux informations que lui donne le director.
- Il n'y a pas d'interface abstraite Builder dans notre cas, la classe URIBuilder n'implémente aucune interface et n'hérite d'aucune classe (sauf Object, comme toute classe en Java).
- Le director, dans notre cas, est le code que nous écrivons pour constituer l'URI. Soit dans le constructeur de URIBuilder, soit en utilisant les setters disponibles. C'est ici notre *main*.
- Le product peut se présenter sous plusieurs formes :
    - Un objet URI -> *build()*
    - Un objet URL -> *build().toURL()*
    - Un objet ASCIIString -> *build().toASCIIString()*
    - Un objet String classique -> *build().ToString()*

On remarque alors que le URIBuilder nous permet de créer uniquement un objet URI en interne, mais il nous permet de le transformer, au dernier moment, en objet URL, String ou ASCIIString. En interne, le processus de building est fixe, il crée une URI. Ensuite, on peut décider de transformer cette URI en une autre classe par après.

Théoriquement, nous devrions avoir un nouveau ConcreteBuilder pour chaque représentation : URI, URL, ASCIIString. Mais il est facilement compréhensible que cela aurait été inutile, lorsqu'une simple conversion entre format suffit.

Quel sont les avantages liés à l'utilisation d'un Builder par rapport à une autre méthode de création ?

- Puisque le Builder est une interface abstraite, l'addition de nouvelle méthode de création est complètement indépendante du code. Il « suffit » d'ajouter un nouveau ConcreteBuilder implémentant le Builder à sa manière.

Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

- L’assemblage du Product final est complètement caché à l’utilisateur, qui n’a pas besoin de savoir ni de se soucier de l’implémentation interne du ConcreteBuilder. Il récupère juste le Product comme il l’a demandé après l’appel de *getResult()*.

- L’assemblage du Product final est modulable, dans notre exemple, nous pouvons passer par le constructeur URIBuilder directement ou bien passer par les nombreux setters disponibles, ou bien les deux. La construction du Product se fait morceaux par morceaux de façon modulable.

Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

Cas n°2

Explication

L'utilisation d'un décorateur pourrait être comparée au concept de module. Nous aurions une application, à laquelle nous ajouterions des modules selon notre besoin. Dans notre cas, il fallait gérer un système permettant de filtrer un fichier contenant du texte, selon certains caractères. Cela fonctionne comme suit :

1. Une interface de base qui va définir les méthodes permettant de lire ou écrire dans un fichier

```
1. public interface DataSource {
2.     void writeData(String data);
3.     String readData();
4. }
5.
```

2. Une classe qui contiendra la logique de base pour lire ou écrire dans un fichier. Celle-ci implémente directement l'interface citée précédemment

```
 1.  public class FileDataSource implements DataSource{
 2.     private String name;
 3.
 4.     public FileDataSource(String name) {
 5.         this.name = name;
 6.     }
 7.
 8.     @Override     public void writeData(String data) {
 9.         File file = new File(name);
10.         try (FileWriter writer = new FileWriter(file)) {
11.             writer.write(data);
12.         } catch (IOException ex) {
13.             System.out.println(ex.getMessage());
14.         }
15.     }
16.
17.     @Override     public String readData() {
18.         char[] buffer = null;
19.         File file = new File(name);
20.         try (FileReader reader = new FileReader(file)) {
21.             buffer = new char[(int) file.length()];
22.             reader.read(buffer);
23.         } catch (IOException ex) {
24.             System.out.println(ex.getMessage());
25.         }
26.         return new String(buffer);
27.     }
28. }
29.
```

3. Un décorateur (ou module) de base, qui définit le comportement que tous les modules ajoutés par la suite adopteront

```
 1. public class DataSourceDecorator implements DataSource{
 2.     private DataSource wrappee;
 3.
 4.     DataSourceDecorator(DataSource source) {
 5.         this.wrappee = source;
 6.     }
 7.
 8.     @Override     public void writeData(String data) {
 9.         wrappee.writeData(data);
10.     }
```

```
11.
12.     @Override     public String readData() {
13.         return wrappee.readData();
14.     }
15. }
16.
17. 4.    Enfin, un ou des décorateurs spécifiques, selon les nécessités du problème. Pour
l'exemple, voici un décorateur permettant de retirer les points et de les remplacer par des
retours à la ligne
18. public class DataSourceDecoratorDot extends DataSourceDecorator{
19.     public DataSourceDecoratorDot(DataSource source) {
20.         super(source);
21.     }
22.
23.     @Override     public void writeData(String data) {
24.         super.writeData(adaptedTreatment(data));
25.     }
26.
27.     @Override     public String readData() {
28.         return adaptedTreatment(super.readData());
29.     }
30.
31.     private String adaptedTreatment(String data) {
32.         data = data.replace(".", "\n");
33.         return new String(data);
34.     }
35.
36. }
37.
```

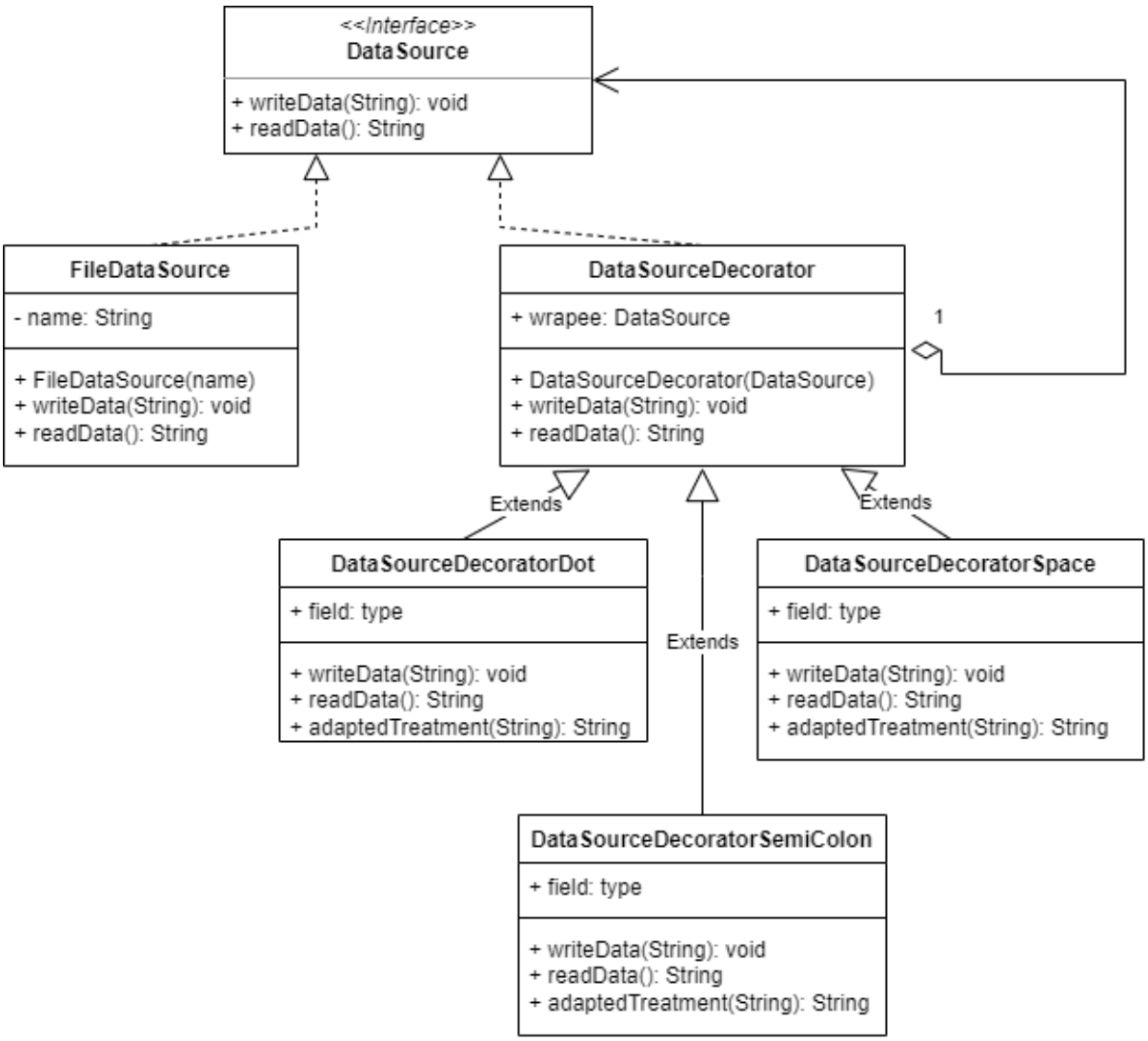
De cette façon, il suffit, après avoir chargé le fichier une première fois grâce à la classe FileDataSource, d'appeler les modules dont nous avons besoin. Cela effectuera des traitements les uns à la suite de l'autre, sans aucune interaction entre chaque classe. Ainsi, il est possible de réaliser toutes les combinaisons possibles et donc avoir un système ayant une très grande flexibilité.

```
 1. public class Main {
 2.      public static void main(String[] args) {
 3.         String text = "hello.ceci. est. un. ;message;.avec. une. ponctuation. spécia le";
 4.
 5.         System.out.println("- Input ----------------");
 6.         System.out.println(text);
 7.         System.out.println("- Post Treatment --------------");
 8.         FileDataSource fd = new FileDataSource("D:\\Ecole\\MASI\\1\\Génie Logiciel\\Séance
4\\cas_2\\src\\main\\resources\\deco.txt");
 9.         DataSourceDecorator dsd = new DataSourceDecoratorSpace(
10.                                     new DataSourceDecoratorSemiColon(
11.                                             new DataSourceDecoratorDot(fd)));
12.         System.out.println(dsd.readData());
13.
14.         dsd.writeData(fd.readData());
15.     }
16. }
17.
```

Dans cet exemple, trois décorateurs sont utilisés afin de traiter une chaine de caractère contenue dans un fichier, contenant des points, points virgules et espaces.

Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

Diagramme



Source :
https://refactoring.guru/fr/design-patterns/decorator

Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

Cas n°3

Problématique de départ :

L'objectif de cas est de développer une application simulant le fonctionnement d'un réveil ayant 3 états possible :

- Eteint : le réveil ne fait aucune action ;
- Allumé : le réveil attend le moment désiré pour faire sonner l'alarme ;
- En sonnerie : le réveil sonne jusqu'à ce qu'on le désactive.

Pour chaque état, on distingue une seule action possible :

- Lorsqu'il est éteint, on peut l'activer ;
- Lorsqu'il est allumé, on peut l'éteindre ;
- En sonnerie : on peut couper la sonnerie et le réveil se remet en marche.

Dans le cas que l'on a choisi, afin de s'alléger le travail, on considère que lorsque le réveil est allumé il attend 5 secondes avant de s'arrêter.

Explication de la solution :

Pour réaliser cette solution, on utilise le pattern de conception « State », c'est un pattern qui utilise des classes pour représenter l'état d'un objet. En travaillant de cette manière, un objet peut avoir un fonctionnement entièrement différent dépendamment de son état.

Dans le cas de notre problématique, l'objet est une classe appelée **AlarmClock** celle-ci utilise le pattern du singleton. Le pattern singleton est un pattern qui permet d'avoir un objet unique au sein de l'application et de pouvoir récupérer une instance de cette objet de manière très simple. Ce singleton contient deux attributs :

- Thread alarmThread : qui représente le thread permettant de faire sonner le réveil en tout voulu ;
- State state : l'objet représentant l'état du réveil.

State est une interface qui permet de forcer la classe l'implémentant de la méthode « click » qui simule l'appui sur le réveil. De state en découle donc trois classe :

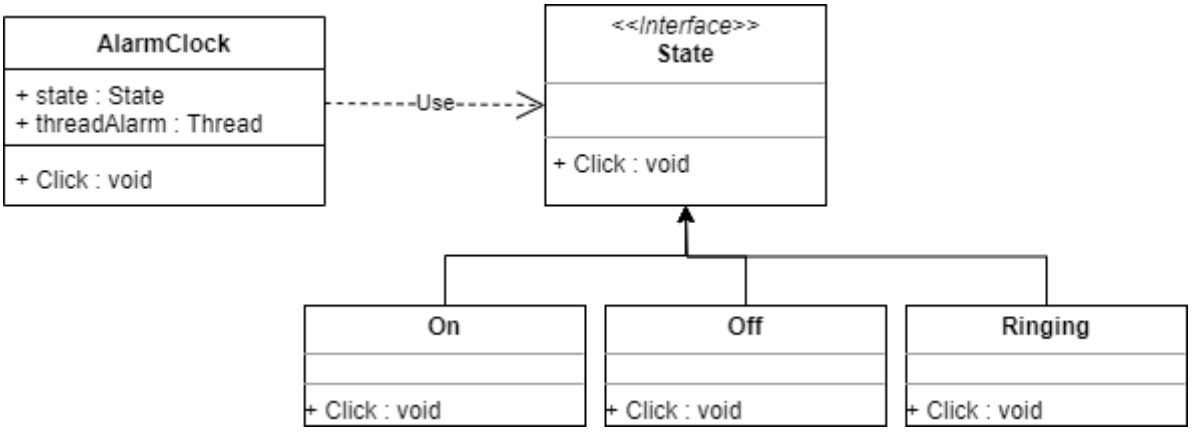
- Off : lors de l'appel de la méthode « click », le réveil passe au mode on ;
- On : le réveil se lance dans le constructeur de cette classe. Lors de l'appel de la méthode « click », le réveil passe à off et le thread de l'alarme est interrompu ;
- Ringing : lors de l'appel de la méthode « click », le thread de l'alarme est interrompu et le thread passe à on.

Dans le main de l'application, le réveil est instancié et une boucle permet de réaliser un click sur le réveil dés que souhaité.

En travaillant avec des états, il est très facile de séparer le travail entre plusieurs développeurs puisque chacun peut travailler sur la classe qui est propre à un état.

Reda BEN MOUSSA
Enzo LEFRANCOIS
Romain LUCAS

Diagramme



Source

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