ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Séance 4

Cas n°1

Situation de départ

L'objectif de ce cas est de définir une méthode permettant la création de voiture correspondant à trois packs :

- Le pack standard : présent de base dans l'ensemble des véhicules. Il contient le kit radio et téléphone,
- Le pack familial : présent dans les voitures familial et comprend un siège enfant et des miroirs supplémentaires,
- Le pack sportif : présent dans les voitures sportives et comprend des suspensions et jantes spécifiques.

Les voitures ne pourront donc être seulement d'un des trois types cités ci-dessus. On pourrait imaginer les modifier par la suite mais dans premier temps on souhaite créer des véhicules qui seront des copies d'un véhicule « de base ».

Solution en javascript

Le javascript propose un mécanisme de création des objets assez particulier :

- La notion d'objet littéral : On crée l'objet en donnant les valeurs souhaités à chaque propriétés :

```
1. let personne = {
2.   nom: "Nomdefamille",
3.   prenom: "Kenny",
4.   age: 20
5. };
6.
```

- Par un constructeur : De manière plus classique, l'objet est créé via une fonction qui a le rôle de constructeur :

```
1. function Personne(nom, prenom, age) {
2.   this.nom = nom;
3.   this.prenom = prenom;
4.   this.age = age;
5. }
6. let personne1 = new Personne("Nomdefamille", "Kenny", 20);
7.
```

- En utilisant des objets prototypes : Un objet littéral est créé et l'on copiera cette objet pour l'utiliser à l'aide de la méthode « create » de la classe « Object »

```
1. let personneProto = {
2. nom: "Nomdefamille",
3. prenom: "Kenny",
4. age: 20
5. };
6. let personne2 = Object.create(personneProto);
7.
```

Si on le souhaite on peut modifier l'objet personne deux pour modifier les noms mais pour l'instant ce sera une copie de l'objet personneProto.

Pour revenir sur le cas de départ, on aura 3 classes :

```
class KitStandard {
    constructor() {
        this.equipements = "Kit Radio + Téléphone";
    }
}

class KitFamille extends KitStandard {
    constructor()  {
        super();
        this.equipements = this.equipements + " + Siège Enfant + Miroirs";
    }
}

class KitSport extends KitStandard  {
    constructor() {
        super();
        this.equipements = this.equipements +" + Suspension + Jantes Spécifiques";
    }
}
```

Celles-ci permettent de représenter les 3 voitures disponibles on va donc pouvoir créer 3 objets prototype :

```
let VoitureDeBaseProto = new KitStandard();
let VoitureDeSportProto = new KitSport();
let VoitureFamilialleProto = new KitFamille();
```

Une fois les 3 classes prototypes créées, il est facile de créer des objets à l'aide du moyen expliqué ci-dessus :

```
let NouvelleVoiture =  Object.create(VoitureDeBaseProto);
console.log(NouvelleVoiture.equipements);
NouvelleVoiture = Object.create(VoitureFamilialleProto);
console.log(NouvelleVoiture.equipements);
NouvelleVoiture = Object.create(VoitureDeSportProto);
console.log(NouvelleVoiture.equipements);
```

Cette solution ressemble fortement à un design pattern assez utilisé : le Prototype, aussi appelé Clone.

Ce design pattern fonctionne avec le même mécanisme, il est possible de créer des clones d'objets à partie de l'objet en lui-même. Afin d'avoir une meilleure idée de l'implémentation de ce pattern, nous avons aussi souhaité l'implémenter en java.

Solution en Java

En Java, nous retrouvons une classe mère « StandardPackCar », qui contient bien les spécificités du pack standard. Ensuite, les classes « SportsPackCar » et « FamilyPackCar » hérite de la classe standard. De cette manière, elles possèdent la radio, le téléphone et leurs spécificités individuelles.

Ce qui nous intéresse le plus est la méthode clone, disponible dans chacune des trois classes. Cette méthode ne prend aucun paramètre et contient une seule ligne de code. En fait, elle retourne un objet de la même classe, identique, en appellant un constructeur spécial qui prend un objet de la même classe en paramètre. Dans ce constructeur, tous les champs de l'objet passé en paramètre sont alors copiés dans le nouvel objet, qui est retourné par la méthode clone. On se retrouve alors avec un nouvel objet identique, sans avoir eu à le (re)configurer individuellement.

```
Dans la classe StandardPackCar, identique à la classe SportsPackCar

 1. public StandardPackCar clone() {
 2.     return new StandardPackCar(this);
 3. }
 4.
 5. public StandardPackCar(StandardPackCar standardPackCar) {
 6.     this.radioModel = standardPackCar.getRadioModel();
 7.     this.phoneSoftVersion = standardPackCar.getPhoneSoftVersion();
 8. }

Dans le main..

 9.
10. SportsPackCar sportsCar = new SportsPackCar("ClioRadioV3", "v2012.10.23.A",
"SportsSuspensionV6", "SportsWheelsV8");
11. SportsPackCar sportsCarClone = sportsCar.clone();
12. System.out.println(sportsCar == sportsCarClone); // toujours false
```

Conclusion

Ce design pattern peut être extrèmenent utile lorsque l'on a besoin d'une grande quantité d'objet se ressemblant mais que l'on souhaite pouvoir parfois modifier un attribut de cet objet.

Source :

https://refactoring.guru/fr/design-patterns/prototype

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

```
public interface DataSource {
    void writeData(String data);
    String readData();
}
```

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

```
 public class FileDataSource implements DataSource{
    private String name;

    public FileDataSource(String name) {
        this.name = name;
    }

    @Override     public void writeData(String data) {
        File file = new File(name);
        try (FileWriter writer = new FileWriter(file)) {
            writer.write(data);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }

    @Override     public String readData() {
        char[] buffer = null;
        File file = new File(name);
        try (FileReader reader = new FileReader(file)) {
            buffer = new char[(int) file.length()];
            reader.read(buffer);
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
        return new String(buffer);
    }
}
```

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

```
public class DataSourceDecorator implements DataSource{
    private DataSource wrappee;

    DataSourceDecorator(DataSource source) {
        this.wrappee = source;
    }

    @Override     public void writeData(String data) {
        wrappee.writeData(data);
    }

    @Override     public String readData() {
        return wrappee.readData();
    }
}
```

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

```
public class DataSourceDecoratorDot extends DataSourceDecorator{
    public DataSourceDecoratorDot(DataSource source) {
        super(source);
    }

    @Override     public void writeData(String data) {
        super.writeData(adaptedTreatment(data));
    }

    @Override     public String readData() {
        return adaptedTreatment(super.readData());
    }

    private String adaptedTreatment(String data) {
        data = data.replace(".", "\n");
        return new String(data);
    }

}
```

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.

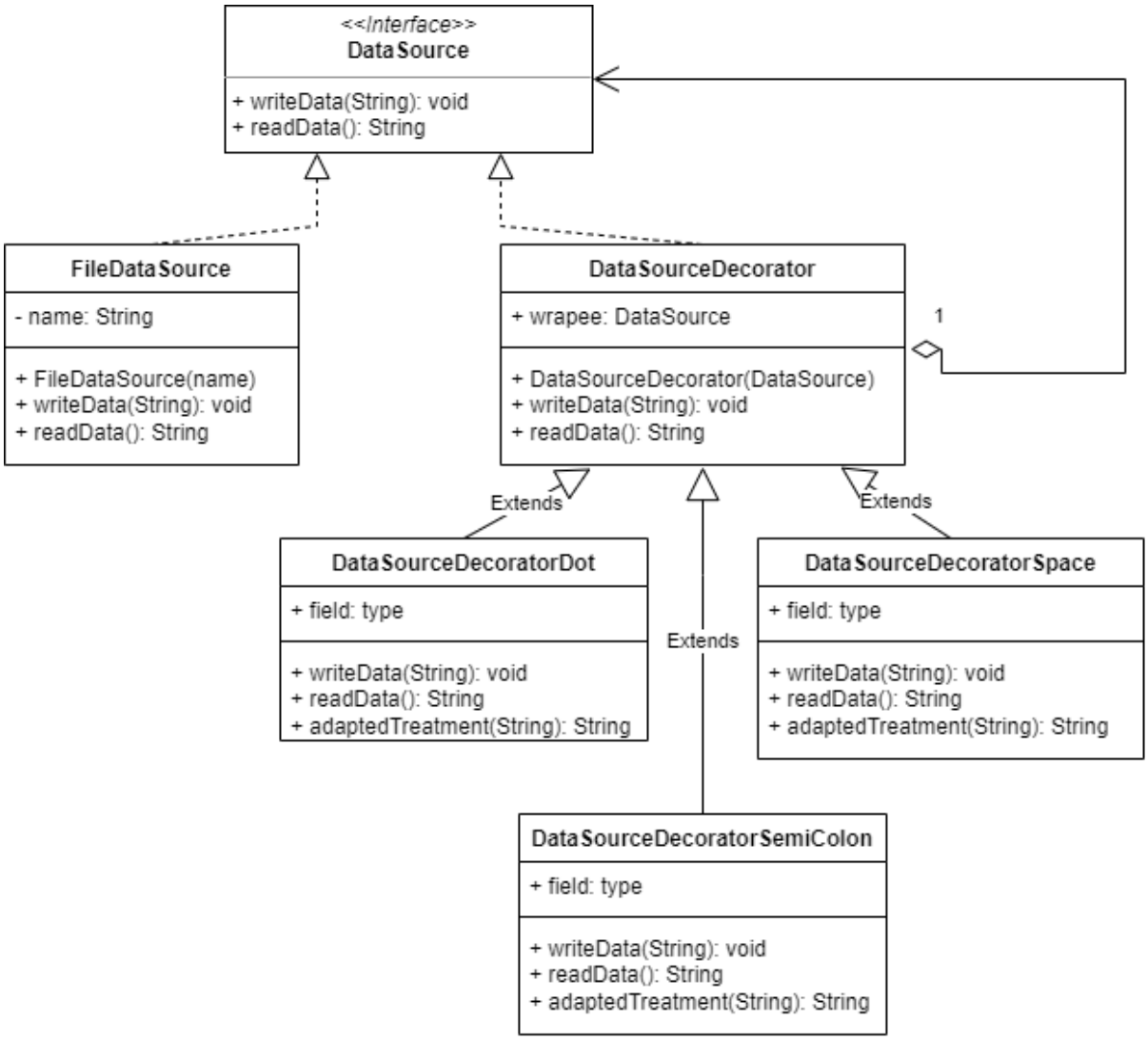
```
public class Main {
     public static void main(String[] args) {
        String text = "hello.ceci. est. un. ;message;.avec. une. ponctuation. spécia le";

        System.out.println("- Input ----------------");
        System.out.println(text);
        System.out.println("- Post Treatment --------------");
        FileDataSource fd = new FileDataSource("D:\\Ecole\\MASI\\1\\Génie Logiciel\\Séance
4\\cas_2\\src\\main\\resources\\deco.txt");
        DataSourceDecorator dsd = new DataSourceDecoratorSpace(
                                    new DataSourceDecoratorSemiColon(
                                            new DataSourceDecoratorDot(fd)));
        System.out.println(dsd.readData());

        dsd.writeData(fd.readData());
    }
}
```

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.

Diagramme



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