ABDALLAH Noursine
BEN MOUSA Reda
LEQUIEN Florent
LUCAS Romain

Séance 2

Cas 1 :

Idée 1 : Injection de dépendance

Création d'une classe par type de connectique qui implémente l'interface *Chargeable.* Par exemple avec USBCSmartphone :

```
1. public class USBCSmartphone implements Chargeable {
2.     public void charger() {
3.         System.out.println("Charging USB-C phone ...");
4.     }
5. }
6.
```

De cette manière on peut placer notre téléphone dans le *StandardCharger* grâce à l'interface *Chargeable*.

```
1. public static void main( String[] args )
{
    Chargeable simplePhone=new SimplePhone();
    StandardCharger standardCharger=new StandardCharger(simplePhone);
    standardCharger.connectPhone();

    Chargeable usbCSmartphone= new USBCSmartphone();
    standardCharger=new StandardCharger(usbCSmartphone);
    standardCharger.connectPhone();

    Chargeable microUSBSmartphone= new MicroUSBSmartphone();
    standardCharger=new StandardCharger(microUSBSmartphone);
    standardCharger.connectPhone();
}
2.
```

Cette solution règle le problème au niveau du *StandardCharger*, mais pose un autre problème : le code de *USBCSmartphone* a du être modifié puisque la méthode « *System.out.println("Charging USB-C phone") »* est supprimée afin de ne pas avoir deux méthodes de charge.

Idée 2 : Adapter USBC

L'idée d'un adapter est de pouvoir adapter un smartphone qui ne possède pas de méthode *charger* mais ici une méthode *chargeWithUSBCCable()*. Pour ce faire une classe implémentant l'interface *chargeable* va être créée. Voici cette classe :

```
1. public class CableAdapter implements Chargeable{
    USBCSmartphone usbcSmartphone;

    public CableAdapter(USBCSmartphone smartphone){
        usbcSmartphone = smartphone;
    }

    @Override     public void charger() {
        usbcSmartphone.chargeWithUSBCCable();
    }
}
2.
```

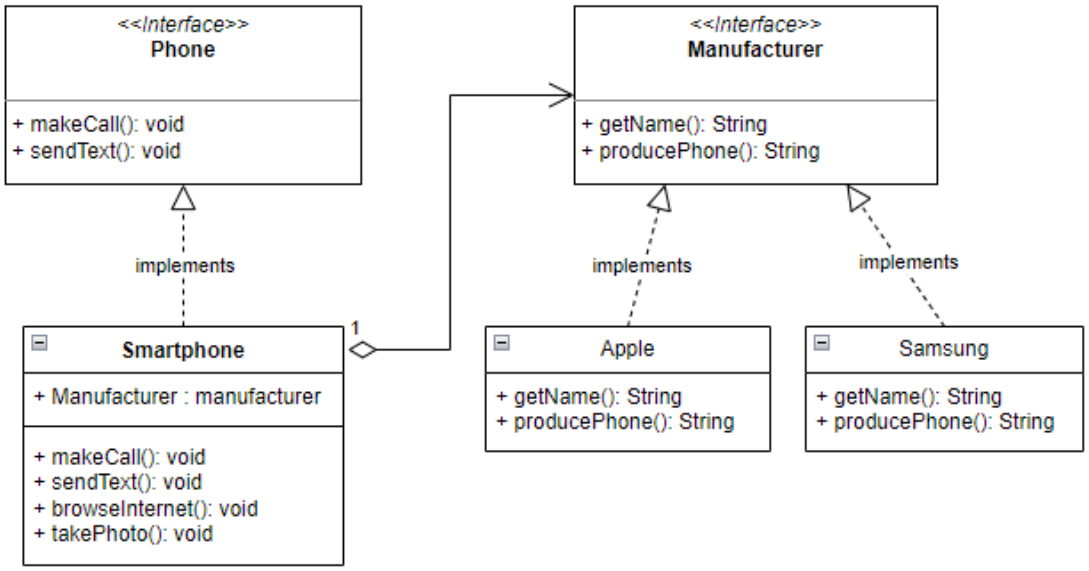
Idée d'amélioration :

Créer un adapteur générique afin de pouvoir traiter tous les smartphones à l'aide d'un seul adapteur.

ABDALLAH Noursine
BEN MOUSA Reda
LEQUIEN Florent
LUCAS Romain

Cas 2 :

L'utilisation du Bridge dans notre cas permet de créer une application flexible et extensible si jamais d'autres téléphones d'autres fabricants venaient à s'ajouter par la suite. Deux concepts sont à prendre en compte ici : l'abstraction (les fonctionnalités des téléphones) et l'implémentation (les détails des fabricants). Cela signifie que pour les nouveaux ajouts, nous n'aurons pas à modifier le code existant.



ABDALLAH Noursine
BEN MOUSA Reda
LEQUIEN Florent
LUCAS Romain

Cas 3 :

Pour résoudre cette problématique, on est parti du postulat qu'un constructeur ne pouvait construire que des voitures familiales ou que des voitures sportives. Les deux types de voitures se distinguant par leurs options.

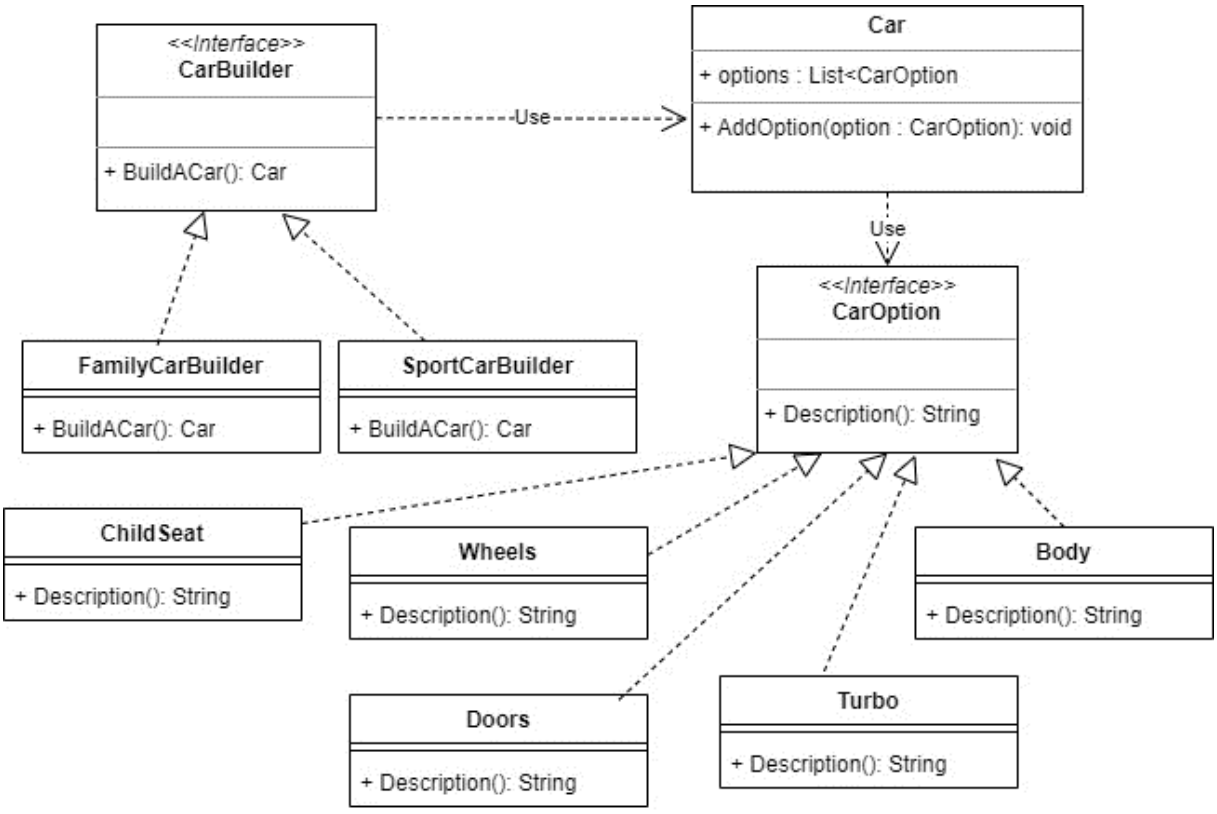
On peut donc considérer qu'une voiture possède une liste d'option. Dans ce cas, une option est représentée par un objet implémentant l'interface *CarOption.*

La classe *Car* contient donc une *List<CarOption>* et une méthode *AddOption()* permettant d'ajouter une option à cette liste.

Pour construire une voiture, un garage à besoin d'un *CarBuilder,* ce-dernier étant une interface implémentée par *FamilyCarBuilder* ou *SportCarBuilder*. L'interface force l'implémentation d'une méthode : *public Car buildACar()*. Cette méthode a pour but de retourner un objet *Car* avec les options propres à chaque *CarBuilder.*

En procédant de cette manière, on pourrait facilement créer d'autre *CarBuilder* si c'était nécessaire et aussi très facilement rajouter des options en rajoutant des objets implémentant l'interface *CarOptions*

Ce pattern est appelé abstract factory, son implémentation permet la création d'objet complexe tout en s'assurant un type commun (ici l'objet *Car*).



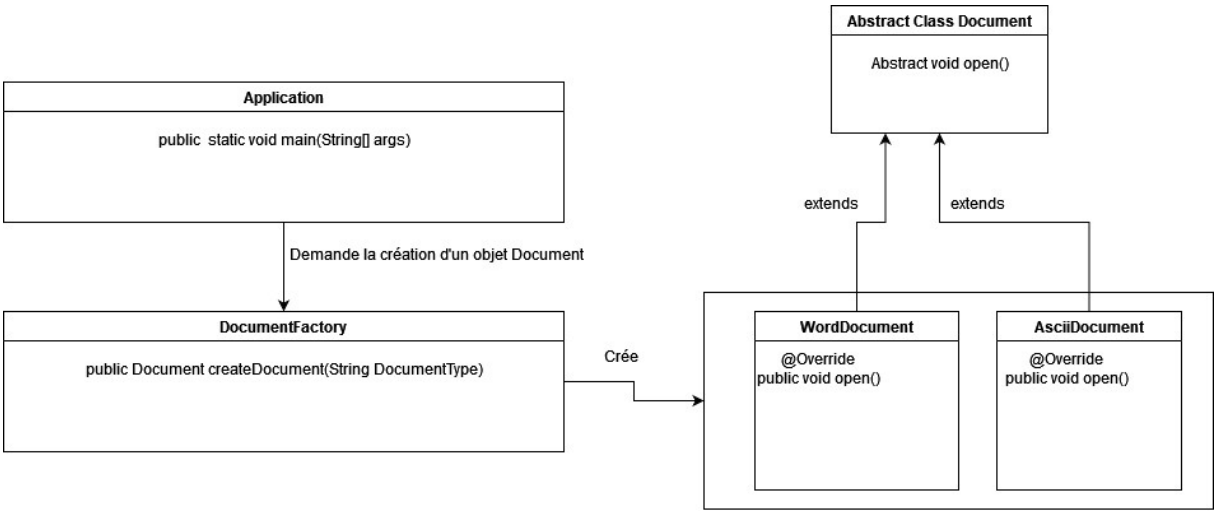
Pour implémenter ce pattern, nous nous somme aidés de cette source :

https://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

ABDALLAH Noursine
BEN MOUSA Reda
LEQUIEN Florent
LUCAS Romain

Cas 4 :

Grâce à l'implémentation de la FactoryMethod (DocumentFactoryMethod.java dans notre code), on délègue la responsabilité de la création d'un document à cette classe FactoryMethod. Cela signifie que la classe Application peut créer des objets Document sans avoir à connaitre l'implémentation concrète du document qui sera créé après.
C'est très pratique de mettre en place ce design pattern dans un cas comme celui présenté au cas n°4 parce que cela permet de simplifier l'ajout de nouveaux types de document. Si une application a besoin d'un autre type de document, il suffira de créer l'implémentation de ce document qui extend la classe Document, et d'ajouter dans le switch case de la FactoryMethod la nouvelle implémentation de document. Ce schéma illustre la manière dont est structurée le code du projet :



Nous nous sommes aidés de ces 2 sites pour implémenter le pattern :
https://www.javatpoint.com/factory-method-design-pattern

https://refactoring.guru/design-patterns/factory-method