

MCDC - Design Pattern

TD

hmmm

Elana Courtines
courtines.e@gmail.com
<https://github.com/irinacake>

Séance 1 - 16 novembre 2022

Christine Regis - name@domain

Exercice 1

Question a : Quel design pattern convient pour éviter la duplication du code commun ?

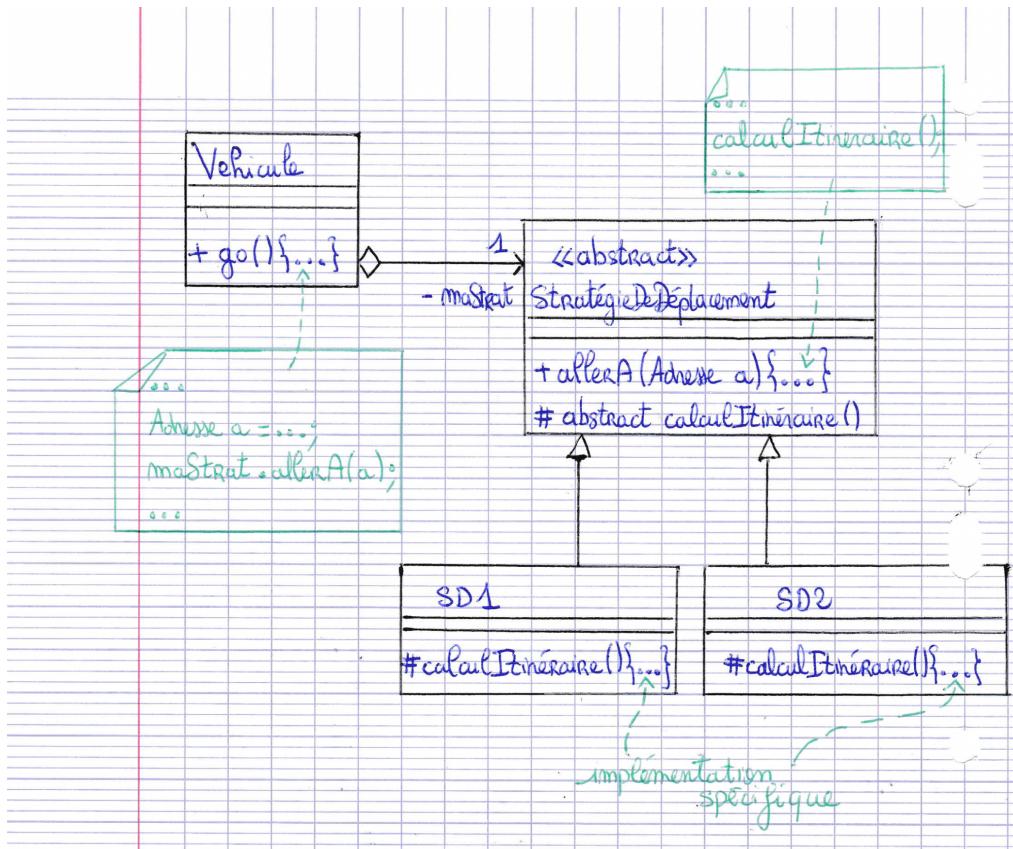
Patron de Méthode

Question b : Critiquez les mises en œuvre (au moyen de ce pattern) proposées par Alice, Bob, Carole et David (cf. Annexe 1).

- Alice : elle a juste c/c la proposition... ;
- Bob : la voiture lance "allerA()". Cette méthode doit rester la seule méthode appelée, sinon on ne respecte plus la spécification ;
- Carole :
 - elle a bien réalisé l'encapsulation
 - une implémentation d'une classe abstraite doit implémenter TOUTES les méthodes abstrates
 - la déclaration d'une méthode abstraite ne doit PAS contenir de code (elle a mis des accolades sur calculer11() et calculer12())
 - la "note" du contenu de allerA() semble indiquer une redéfinition du contenu (avec les accolades), c'est non
- David : non... ceci ne correspond pas à un patron de méthode.

Note : il est préférable d'ajouter des notes / explications sur les schémas.

Question c : Concevoir la solution en modifiant le diagramme de classes ci-dessus. Donner les éléments de code utiles.



Question d : Donner un exemple de code de déploiement (création et configuration d'un véhicule avec une stratégie au choix).

```
1 package sourceExo1;
2 public class Adresse {
3     public Adresse() {
4     }
5 }
```

```
1 package sourceExo1;
2 public abstract class StrategieDeDeplacement {
3     public StrategieDeDeplacement() {
4     }
5     public void allerA(Adresse a){
6         calculerItineraire();
7     }
8     protected abstract void calculerItineraire();
9 }
```

```
1 package sourceExo1;
2 public class SD1 extends StrategieDeDeplacement {
3     @Override
4     protected void calculerItineraire() {
5     }
6 }
```

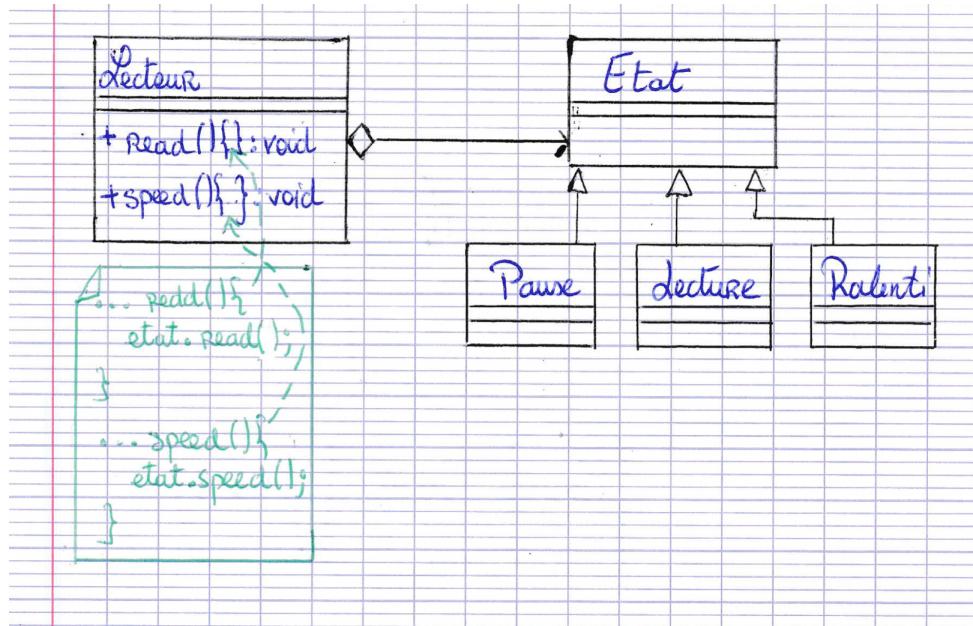
```
1 package sourceExo1;
2 public class SD2 extends StrategieDeDeplacement {
3     @Override
4     protected void calculerItineraire() {
5     }
6 }
```

```
1 package sourceExo1;
2 public class Vehicule {
3     private StrategieDeDeplacement maStrat;
4     public Vehicule(StrategieDeDeplacement maStrat) {
5         this.maStrat = maStrat;
6     }
7     public void go(){
8         Adresse a = new Adresse();
9         maStrat.allerA(a);
10    }
11 }
```

```
1 package sourceExo1;
2 public class Main{
3     public static void main(String [] args) {
4         StrategieDeDeplacement SDChoisie = new SD1();
5         Vehicule v = new Vehicule(SDChoisie);
6         v.go();
7     }
8 }
```

Exercice 2

Question a : Concevoir l'application et décrire la solution (diagramme de classes, etc.).



Question b : La solution met en œuvre un design pattern : lequel ? De quel autre pattern est-il proche et en quoi diffère-t-il ?

Il s'agit ici du design pattern *State*, à ne pas confondre avec le design pattern *Strategy*. Cf. ce topic sur [StackOverflow](#) :

Honestly, the two patterns are pretty similar in practice, and the defining difference between them tends to vary depending on who you ask. Some popular choices are:

- States store a reference to the context object that contains them. Strategies do not.
- States are allowed to replace themselves (IE: to change the state of the context object to something else), while Strategies are not.
- Strategies are passed to the context object as parameters, while States are created by the context object itself.
- Strategies only handle a single, specific task, while States provide the underlying implementation for everything (or most everything) the context object does.

A "classic" implementation would match either State or Strategy for every item on the list, but you do run across hybrids that have mixes of both. Whether a particular one is more State-y or Strategy-y is ultimately a subjective question.