

Le Strategy Pattern



Plan :

- I – Le Strategy Pattern
 - 1 – Les Design Pattern
 - 2 – Problématique liée au Strategy Pattern
 - 3 – Principe Du Strategy Pattern
- II – Explication par l'exemple
 - 1– La vue d'ensemble du Strategy Pattern
 - 2– L'implémentation du Strategy Pattern

I- Le Strategy Pattern :

1- Design Pattern ?

- Solution à un problème récurrent dans la conception d'applications orientées objet.
- Indépendamment des langages de programmation utilisés.



I- Le Strategy Pattern :

- 3 catégories :
 - **Création** : ils permettent d'instancier et de configurer des classes et des objets.
 - **Structure** : ils permettent d'organiser les classes d'une application.
 - **Comportement** : ils permettent d'organiser les objets pour qu'ils collaborent entre eux.

I- Le Strategy Pattern :

2- Problématique liée au Strategy Pattern :

Réalisation de différentes opérations sur des entiers avec le même objet



I- Le Strategy Pattern :

Solutions :

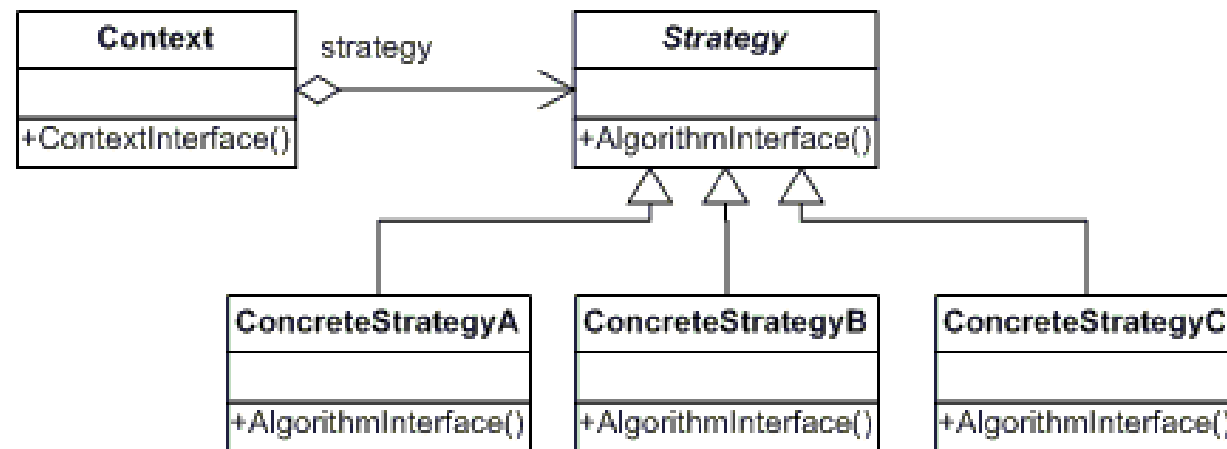
- Une classe avec toutes les opérations ?

 - ➡ Non respect du "Single Responsibility Principle"


 - ➡ Utilisation du Strategy Pattern !

I- Le Strategy Pattern :

3 – Principe Du Strategy Pattern



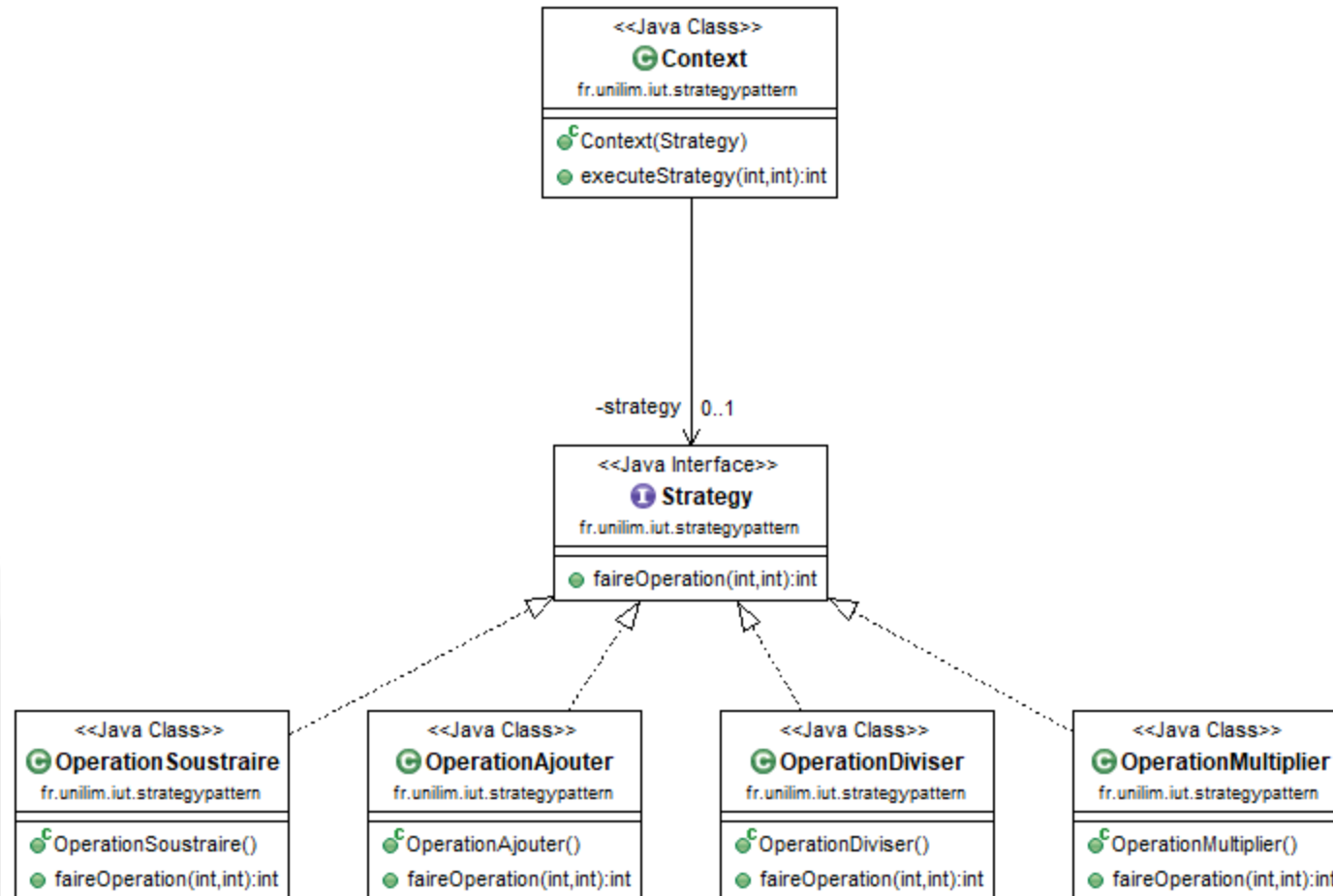
I- Le Strategy Pattern :

- Respect du "Single Responsibility Principle"
 - Meilleure lisibilité du code
 - Moins de risque de bug lors de l'ajout de fonctionnalités
- 

II – Explication par l'exemple

1- La vue d'ensemble du Strategy Pattern

Diagramme:



II – Explication par l'exemple

2- L'implémentation du Strategy Pattern

1^{ère} étape :

Création d'une interface "Strategy"

```
package fr.unilim.iut.strategypattern;  
  
public interface Strategy {  
    public int faireOperation(int num1, int num2);  
}
```

II – Explication par l'exemple

2ème étape :

Création des opérations qui implémentent l'interface

```
package fr.unilim.iut.strategypattern;  
  
public class OperationDiviser implements Strategy{  
    @Override  
    public int faireOperation(int num1, int num2) {  
        return num1 / num2;  
    }  
}
```

```
package fr.unilim.iut.strategypattern;  
  
public class OperationMultiplier implements Strategy{  
    @Override  
    public int faireOperation(int num1, int num2) {  
        return num1 * num2;  
    }  
}
```

```
package fr.unilim.iut.strategypattern;  
  
public class OperationSoustraire implements Strategy {  
    @Override  
    public int faireOperation(int num1, int num2) {  
        return num1 - num2;  
    }  
}
```

```
package fr.unilim.iut.strategypattern;  
  
public class OperationAjouter implements Strategy{  
    @Override  
    public int faireOperation(int num1, int num2) {  
        return num1 + num2;  
    }  
}
```

II – Explication par l'exemple

3ème étape:

Création de la classe qui va contextualiser l'interface

```
package fr.unilim.iut.strategypattern;

public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.faireOperation(num1, num2);
    }
}
```

II – Explication par l'exemple

4ème étape:

Test du pattern :

```
package fr.unilim.iut.strategypattern;

public class StrategyPatternDemo {

    public static void main(String[] args) {

        Context context = new Context(new OperationAjouter());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationSoustraire());
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiplier());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationDiviser());
        System.out.println("10 / 5 = " + context.executeStrategy(10, 5));

    }
}
```

Sortie :

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
|
```

Conclusion :

- Nombreux points positifs
- Facile à comprendre et implémenter

