Pattern Strategy

BATARDIERE Simon - PATERNE Baptiste - TRICARD Aurélien

Sommaire

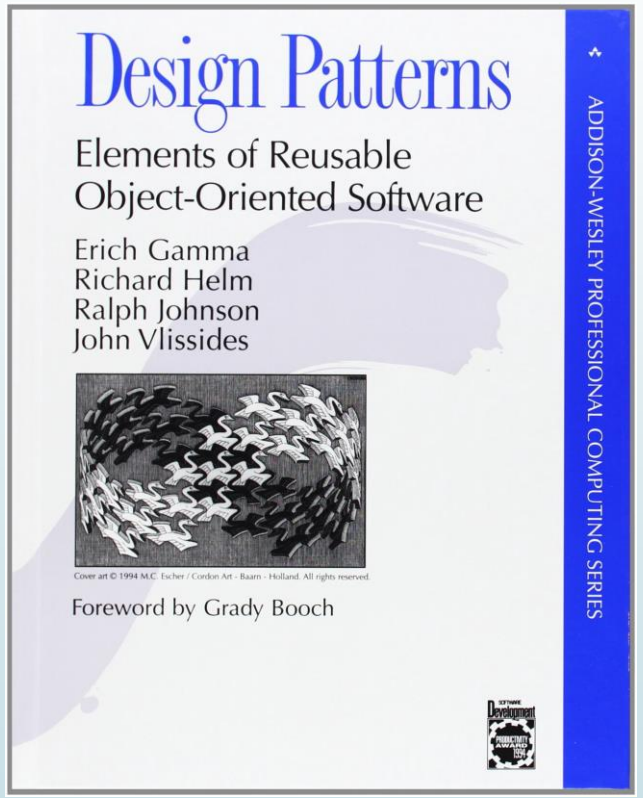
I. Design Patterns

Patron de création

- Solution à un problème de conception récurrent

Dates clés :

- 1970 : Concept issue de l'architecture par *Christopher Alexander*
- *1987 :* Début d'application de ces modèles à la programmation par *Kent Beck et Ward Cunningham*
- *1995 : Design Patterns : Elements of Reusable Object-Oriented Software* par le Gang Of Four



I. Design Patterns

Pourquoi utiliser les patterns ?

- Accélérer le processus de développement
- Anticiper des problématiques
- Améliorer la lisibilité du code

Trois grandes catégories :

- les patrons de création (Creational patterns)
- les patrons de structuration (Structural patterns)
- **les patrons de comportement (Behavioral patterns)**

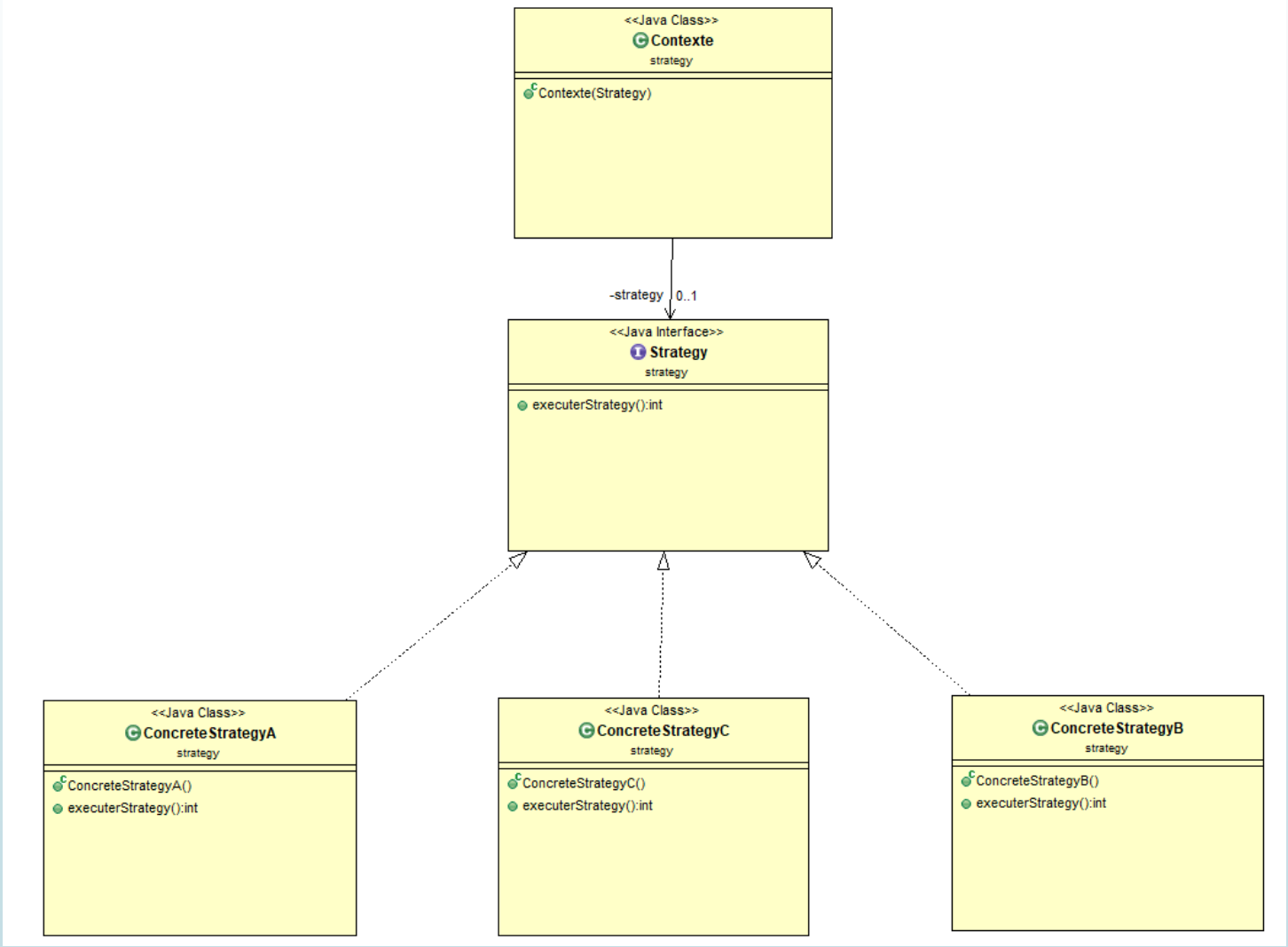
II. Pattern Strategy

1. Principe du Pattern Strategy

- Catégorie : Comportement
- Problématique : réaliser différentes opérations sur un même objet
- Avantages :
  - Code plus lisible
  - Évite la violation de principes SOLID
  - Permet de définir des algorithmes interchangeables
- Inconvénient :
  - Rajoute une classe

II. Pattern Strategy

1. Principe du Pattern Strategy



II. Pattern Strategy

1. Principe du Pattern Strategy

Principes SOLID :

- OCP : Open/Closed Principle
- SRP : Single Responsibility Principle

2. Exemple

- 1ére étape : Création de l'interface

```
package kebab;

public interface Strategy {
    String choixSauce();
}
```

2. Exemple (Suite)

- 2éme étape : création des classes qui implémentent la classe interface

```java
package kebab;

public class SauceSamourai implements Strategy {

    public String choixSauce() {
        return "Samourai";
    }

}
```

```java
public class SauceKetchup implements Strategy {

    public String choixSauce() {
        return "Ketchup";
    }

}
```

2. Exemple (Suite)

- 3éme étape : la contextualisation

```
package kebab;

public class ChoixSauceKebab {
    private Strategy strategy;

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

    public String executeStrategy() {
        return "Vous avez choisi la sauce " + strategy.choixSauce();
    }

}
```

2. Exemple (Suite)

- La dernière étape : l'utilisation du pattern

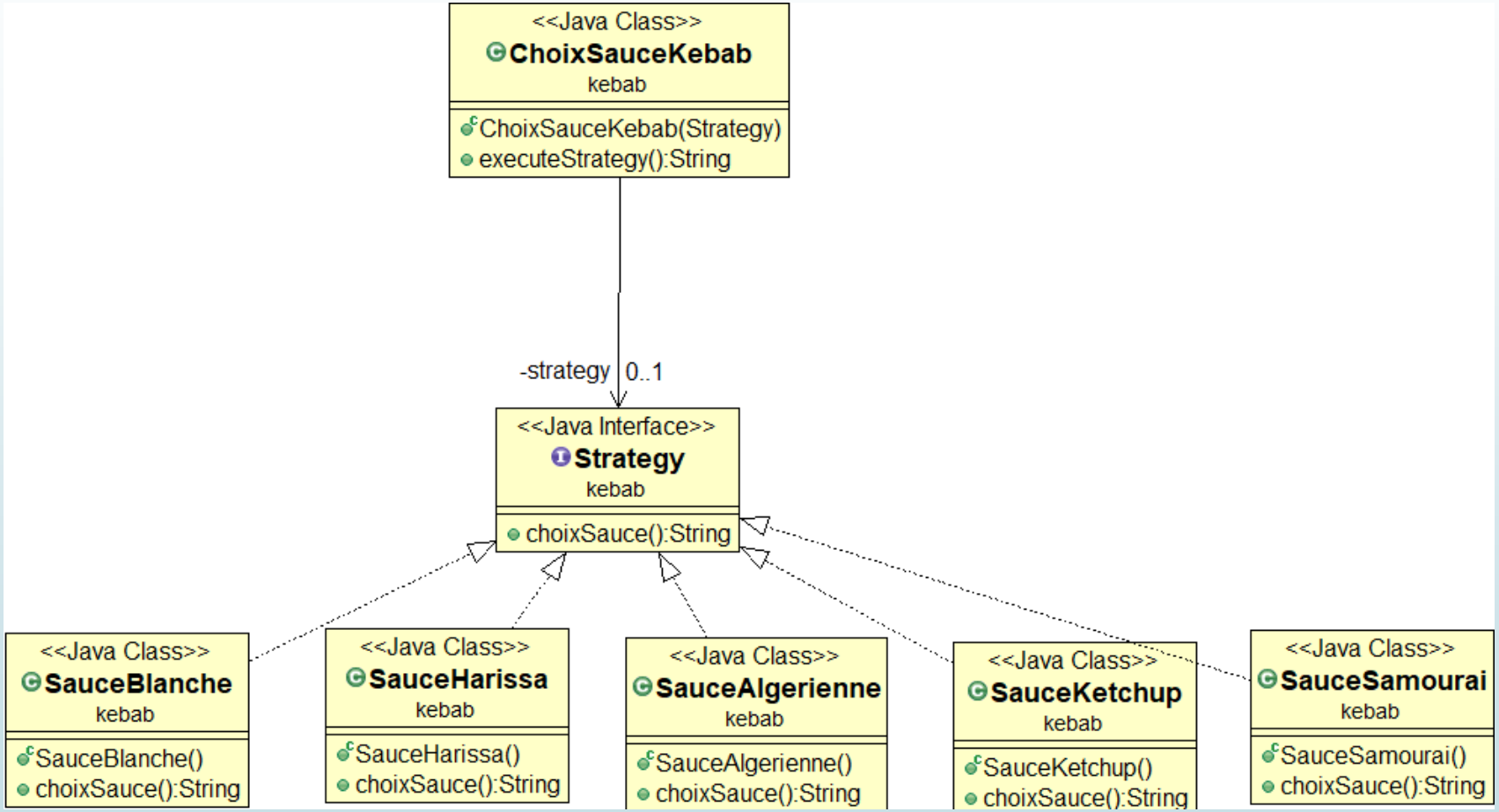
```
public class StategyPatternDemo {

    public static void main(String[] args) {
        ChoixSauceKebab ketchup = new ChoixSauceKebab(new SauceKetchup());
        ChoixSauceKebab Blanche = new ChoixSauceKebab(new SauceBlanche());
        ChoixSauceKebab Algerienne = new ChoixSauceKebab(new SauceAlgerienne());
        ChoixSauceKebab Harissa = new ChoixSauceKebab(new SauceHarissa());
        ChoixSauceKebab Samourai = new ChoixSauceKebab(new SauceSamourai());
        ChoixSauceKebab[] context = {ketchup, Blanche, Algerienne, Harissa, Samourai};

        for (ChoixSauceKebab sauce : context) {
            System .out.println(sauce.toString());
        }
    }
}
```

```
vous avez choisi la sauce Ketchup
vous avez choisi la sauce Blanche
vous avez choisi la sauce Algérienne
vous avez choisi la sauce Ketchup
vous avez choisi la sauce Samourai
```

2. Exemple (Suite & Fin)



III. Conclusion

- Facile à comprendre et à utiliser
- Permet de respecter deux principes SOLID