

# Introduction à la conception et patron stratégie

## Ressources

- [Head First Design Pattern](#), la bible de la COO, assez simple et ludique à lire, très recommandé et avec beaucoup d'illustrations.
- [Plongée au coeur des patrons de conception](#), créé par le site de Refactoring Guru mais un peu moins accessible que l'autre. En revanche celui ci est disponible en français.
- Le site [Refactoring Guru](#) qui est vraiment bien, avec de bonnes illustrations et des descriptions simples de chaque pattern.

## Conception

### Exemple

Imaginons que dans un cas nous avons un code fonctionnant en hiérarchie de classe, comment faire pour transformer une classe en une autre ? Ce n'est pas possible, c'est donc une erreur de conception.

### Définition

La conception permet de trouver des solutions permettant de structurer le code pour favoriser sa **maintenabilité** (modification du code) et son **extensibilité** (ajouter du code) car une application, quoi qu'il arrive doit absolument grandir et changer, sinon elle va mourir. On veut donc *structurer* le code pour faire émerger des qualités désirables en fonction des usages (performance, stabilité, fiabilité, etc) Structurer un code c'est faire les liens entre les différents éléments/modules d'une application. Le niveau au dessus c'est l'architecture, qui établit les liens entre différentes applications d'un système.

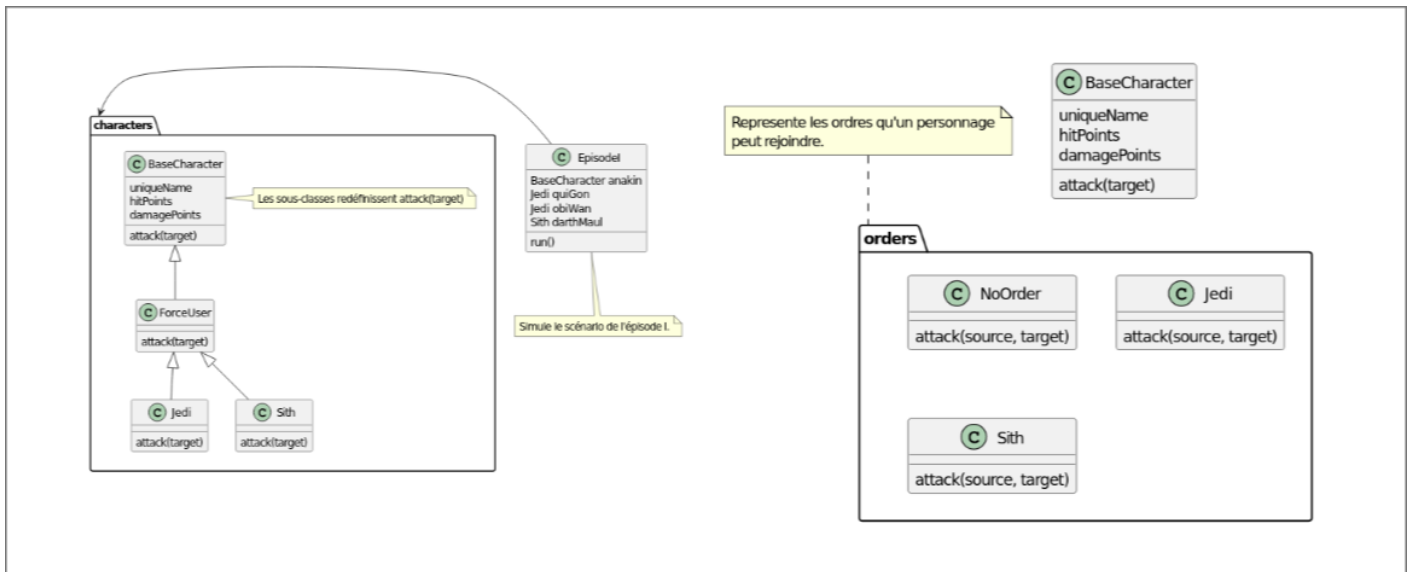
# Principes de la conception

## Etape 1 - Séparer les occupations

Tout d'abord on doit se demander :

- Qu'est ce qui varie ?
- Qu'est ce qui ne change pas ?

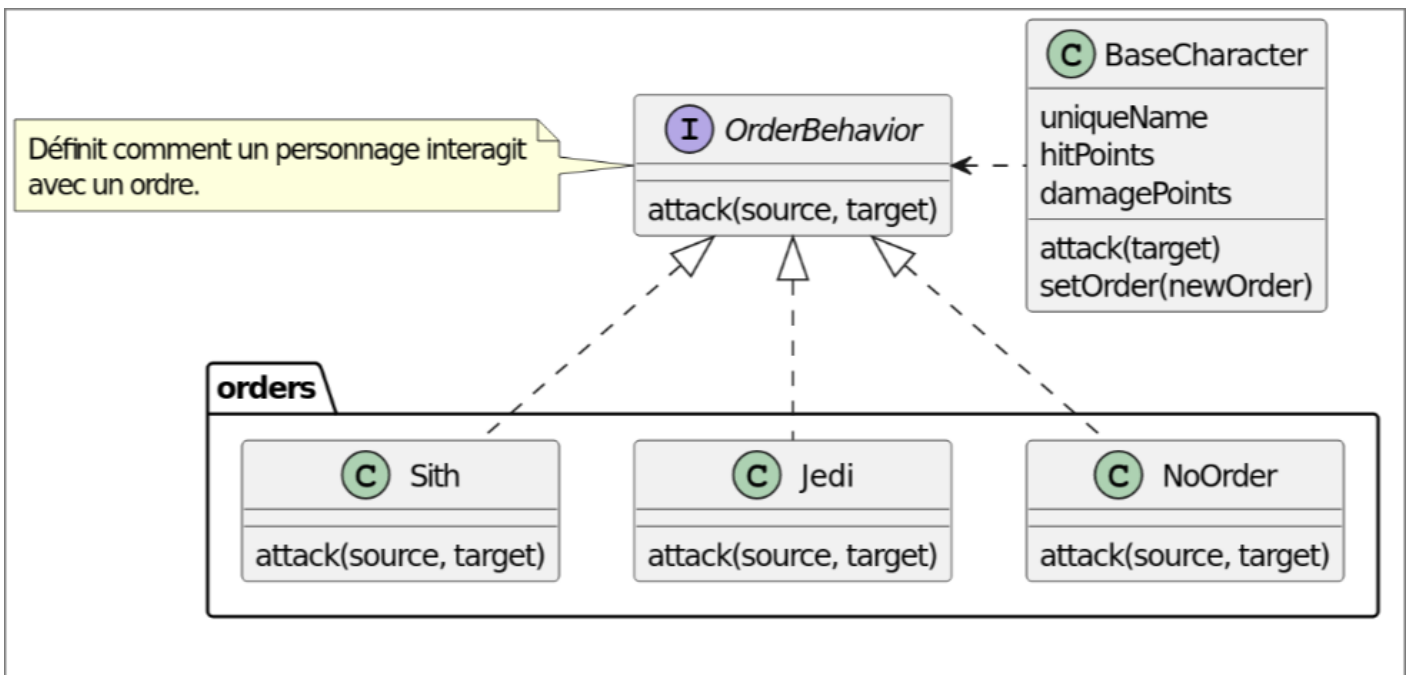
On va donc séparer ce qui change de ce qui ne change pas (très important!)



Par exemple ici, on veut que les personnages puissent changer d'ordre (c'est donc ce qui varie ici), en revanche **BaseCharacter** change pas. On va donc séparer les ordres des personnages.

## Etape 2 - Programmer avec des interfaces et éviter les types concrets

On va donc faire un lien entre ce qui change et ce qui ne change pas. Par exemple ici on peut utiliser une interface pour lier les ordres avec les personnages.



Ici on peut créer une interface "Ordre" pour lier les différents ordres. Ainsi si on veut en créer un nouveau, il suffit de créer une nouvelle classe implémentant l'interface.

## Etape 3 - Favoriser la composition à l'héritage

Enfin il faut pouvoir mémoriser le lien, pour cela dans notre exemple on peut simplement faire un attribut de type Order (notre interface). Il vaut mieux utiliser l'attribut (qui est plus flexible) plus tôt que l'héritage qui est beaucoup plus rigide.

Maintenant il faut pouvoir lier les ordres avec les personnages. Pour cela on peut créer un attribut "ordre" demandant un type de notre interface "Ordre" dans la classe BaseCharacter.

```

public class BaseCharacter {
    private final String uniqueName;
    private final int[] points;
    private OrderBehavior order = new NoOrderBehavior(); // ← composition de ordre dans basecharacter
    public static final int HIT_POINTS = 0;
    public static final int DAMAGE_POINTS = 1;

    public BaseCharacter(String uniqueName, int hitPoints, int damagePoints) {
        this.uniqueName = StringExtensions.requireNotBlank(uniqueName, "uniqueName").strip();
        this.points = new int[]{
            NumberUtils.requireGrEq(hitPoints, 0, "hitPoints"),
            NumberUtils.requireGrEq(damagePoints, 0, "damagePoints")
        };
    }
}
  
```

```

/*Reste du code*/

public String attack(BaseCharacter target) {
    return order.attack(this, target);
}

public void setOrder(OrderBehavior order) {
    this.order = order;
}
}

```

## Etape 4 - Injecter les dépendences

Il est important de ne pas avoir de valeurs null pour un attribut, cependant il faut aussi éviter d'avoir un type concret. Alors pour définir la classe sans utiliser de classe concretes dedans, on peut simplement la demander (et l'obliger) dans le constructeur (soit, faire une **injection de dépendence**).

Ainsi ce code (qui ne respecte pas l'étape 2, car elle utilise un type concret NoOrderBehavior) :

```

public class BaseCharacter {
    private final String uniqueName;
    private OrderBehavior order = new NoOrderBehavior(); // berk
}

```

Deviendra à ce code (avec l'injection de dépendence)

```

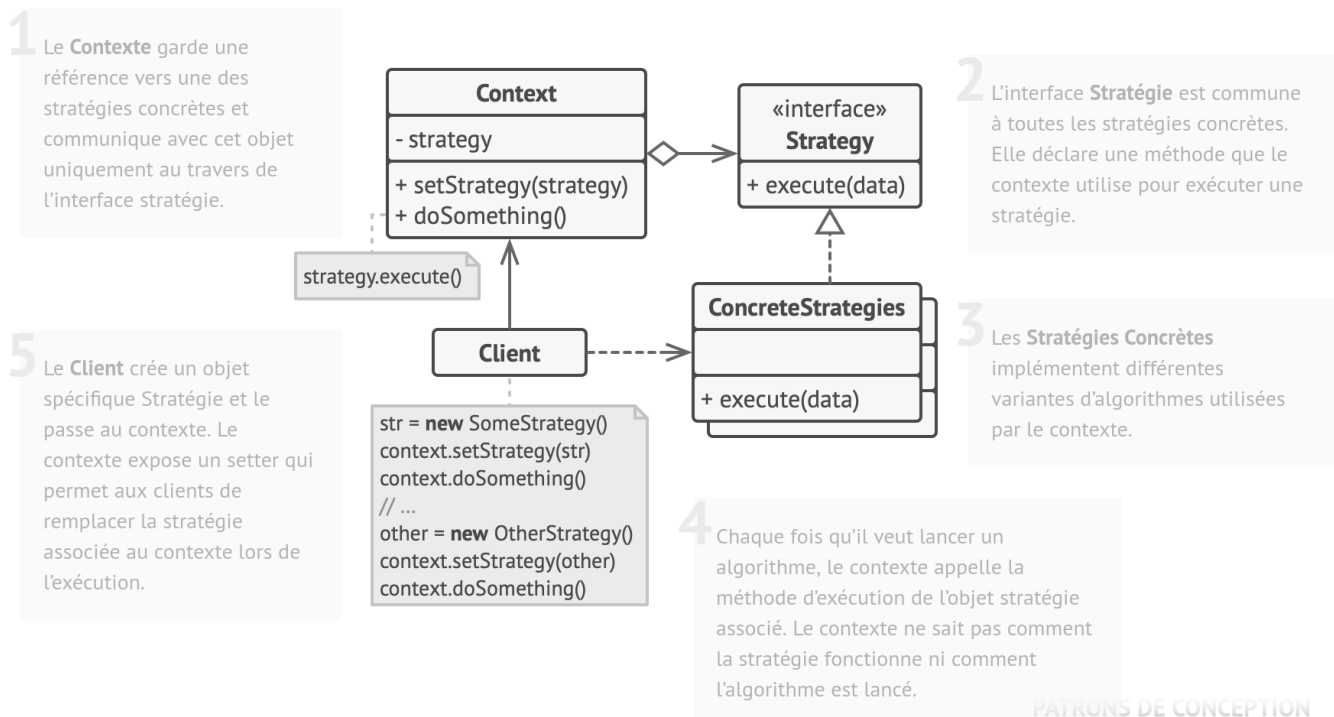
public class BaseCharacter {
    private final String uniqueName;
    private OrderBehavior order;

    public BaseCharacter(String uniqueName, OrderBehavior order) {
        this.uniqueName = uniqueName;
        this.order = Objects.requireNonNull(order);
    }
}

```

## Les patrons

La solution que l'on a trouvé dans l'exemple précédent, nous avons utilisé le patron "stratégie" (qui est l'une des plus basique). Les patrons de conceptions sont des solutions éprouvées à des problèmes réccurents et qui peuvent être facilement adapté à des problèmes spécifiques.



## Catégories

Il existe plusieurs catégories de patrons de conceptions.

- Les patrons **comportementaux** qui proposent des solutions aux problèmes de collaboration et d'affectation des responsabilités entre objets (c'est le cas du patron stratégie)
- Les patrons **créationnels** (créer des objets de façon flexible)
- Les patrons **structurels** (assemblent des objets en structures flexibles)

En vérité on s'en fout des catégories, ce qui compte c'est *l'intention* de chaque pattern (patron).

## Description d'un patron

Chaque patron compte 4 sections:

- L'**intention**, qui décrit brièvement le problème et la solution
- La **motivation**, qui explique en détail la problématique et la solution offerte par le patron
- La **structure**, montre les différentes parties du patron et leurs relations
- L'**exemple de code**, écrit dans un des langages de programmation les plus populaire

Pour voir cette description en action, on peut simplement aller voir le site [Refactoring Guru](https://refactoring.guru/).

## Avantages

- **Réutilisabilité** : Les design patterns offrent des solutions génériques à des problèmes communs
- **Modularité** : ils favorisent la séparation des préoccupations, ce qui facilite la maintenance et l'évolutivité de l'application.
- **Compréhensibilité**, ils sont bien documentés et largement reconnus, ce qui rends le code plus compréhensible pour les développeur·euse qui connaissent ces modèles.
- **Interopérabilité**, ils permettent de standardiser les solutions à des problèmes courants. Cela favorise l'interopérabilité entre les différentes parties d'une application ou entre différentes applications.

# Eviter les généralisations spéculatives

Il faut vraiment attendre qu'un design pattern soit vraiment nécessaire pour le mettre en place pour ne pas intégrer une complexité inutile au code.

---

Revision #6

Created 19 September 2023 18:00:21 by SnowCode

Updated 4 October 2023 10:18:12 by SnowCode