

Les fabriques et ponts

Le nom fabrique est un peu utilisé à toutes les sauces, il existe [un article de RefactoringGuru](#) qui liste les différences entre les différentes appellations du mot.

Fabriques

Exemple du problème

On veut créer une classe permettant de créer et entrainer les StormTroopers. On a donc fait ceci :

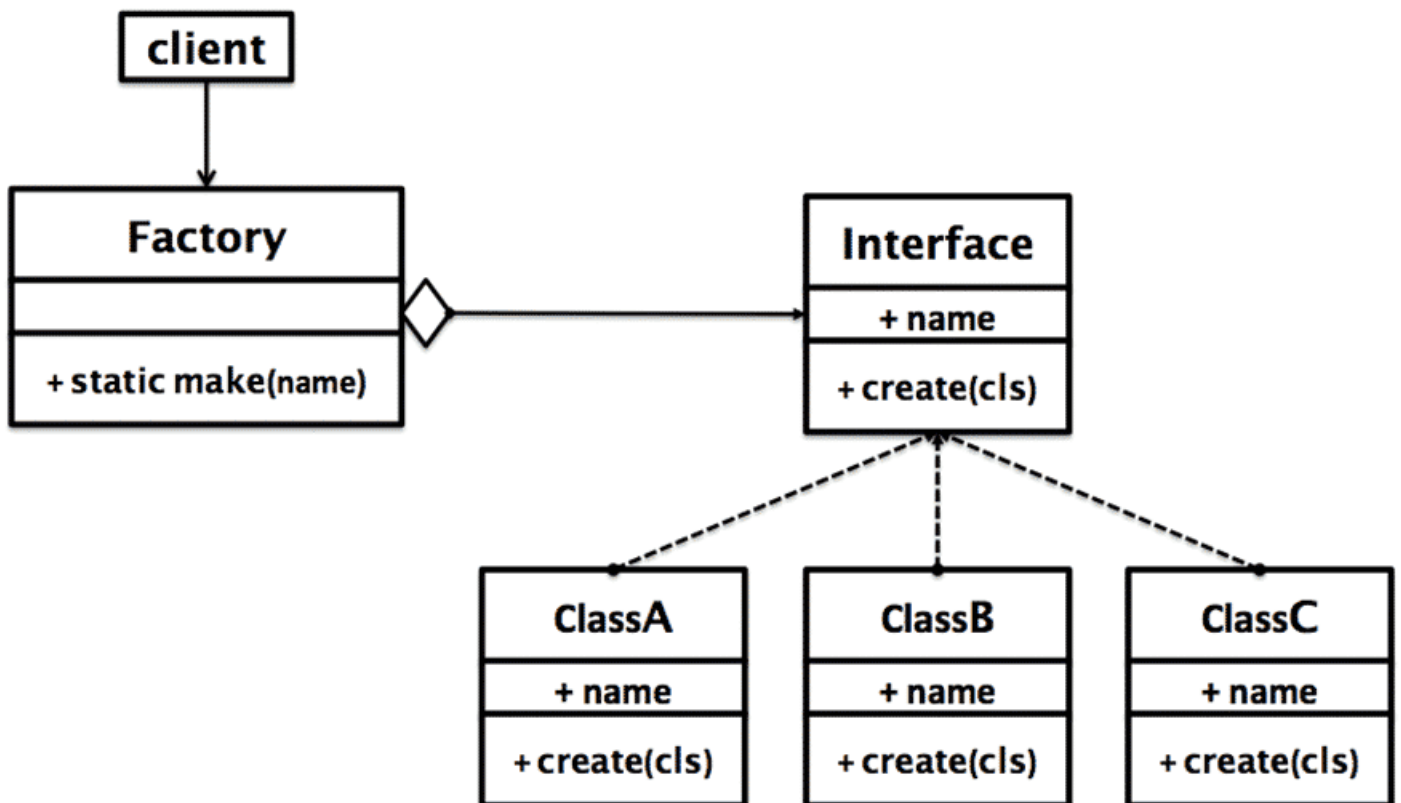
```
public class StormtrooperTrainingFacility {  
    // Cette méthode va créer et entrainer les stormtroopers  
    // MilitarySection est une enum permettant de distinguer les différents types  
    public Stormtrooper createAndPrepare(MilitarySection section) {  
        // Un Stormtrooper du bon type est généré en fonction de la MilitarySection  
        Stormtrooper product = switch(section) {  
            case ASSAULT -> new AssaultStormTrooper();  
            case GRENADIER -> new GrenadierStormTrooper();  
            case PILOT-> new PilotStormTrooper();  
            default-> throw new IllegalArgumentException("section");  
        };  
  
        // Ensuite les stormtroopers vont être entraîné  
        product.equip();  
        product.train();  
        product.passExam();  
  
        // Puis retourné  
        return product;  
    }  
}
```

Dans le code précédent, le code a plusieurs problèmes :

- Il fait plusieurs choses (le training et la création d'objets), ce qui fait que si quelque chose change, il va falloir créer de nouvelles conditions.
- Il connaît trop de classes concrètes (problème d'injection des dépendances)
- Si la création d'un objet change, le code de préparation devra changer aussi
- De plus la structure des StormTroopers est basée sur l'héritage ce qui rends tout beaucoup plus compliqué

Solution : La fabrique simple

Théorie



Il vaut tout de même utiliser un enum pour définir le nom de la chose à produire

On extrait la partie de la création dans une classe Factory à part avec une méthode statique "create"

La "fabrique simple" permet de séparer la responsabilité d'utiliser un objet, de celle de la créer. La méthode de fabrique simple peut aussi être appelée "méthode de fabrique statique polymorphe".

Exemple

```

public class StormtrooperFactory {
    // Cette méthode va créer les stormtroopers
}
  
```

```
// MilitarySection est une enum permettant de distinguer les différents types
public static Stormtrooper create(MilitarySection section) {
    // Un Stormtrooper du bon type est généré en fonction de la MilitarySection
    return switch(section) {
        case ASSAULT -> new AssaultStormTrooper();
        case GRENADIER -> new GrenadierStormTrooper();
        case PILOT -> new PilotStormTrooper();
        default -> throw new IllegalArgumentException("section");
    };
}
}
```

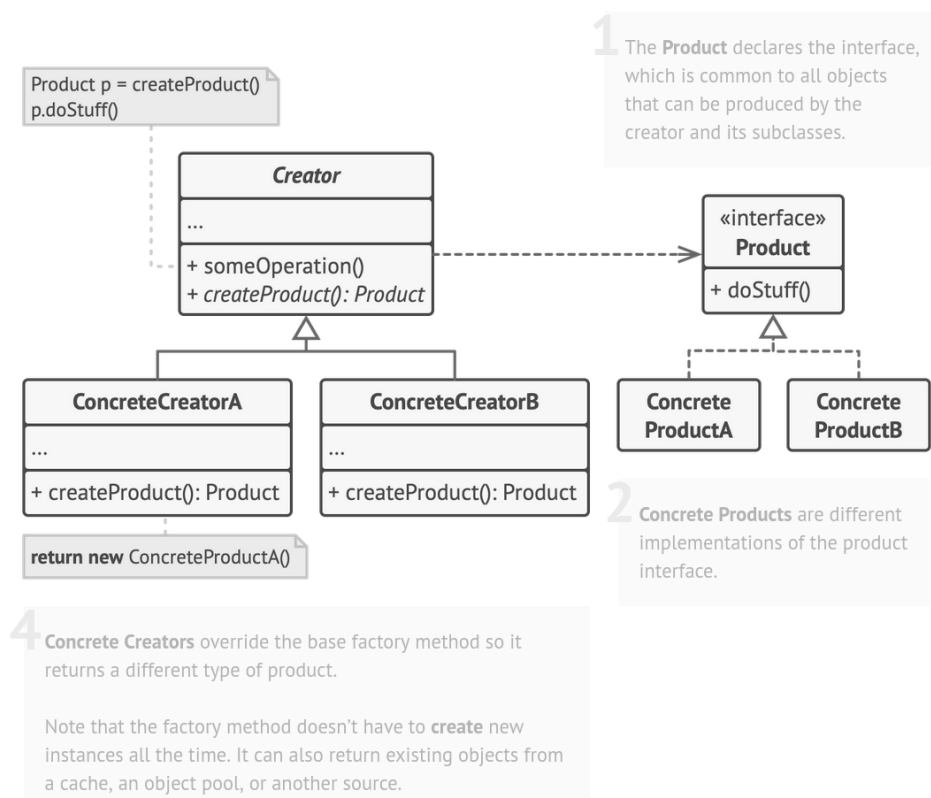
Anti-pattern : La méthode de fabrique

Théorie

3 The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

You can declare the factory method as `abstract` to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.



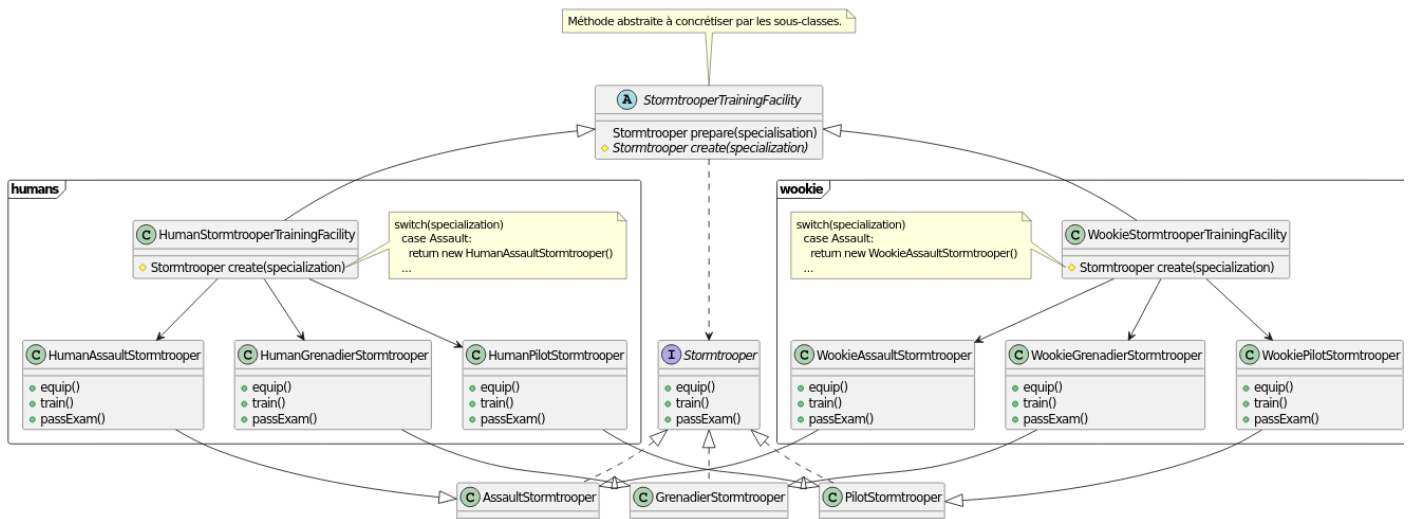
Quand on a vraiment beaucoup de produits et que ça devient un bordel de tout avoir au même endroit, on peut utiliser de patron de conception pour séparer les choses dans plusieurs classes.

Dans ce patron de conception, on a donc plusieurs sous-fabriques qui héritent d'une classe fabrique. La classe va donc créer des Produits et chaque produit concret va implémenter la classe Produit.

Ce patron est également caca, il faut donc éviter de l'utiliser car il y a beaucoup trop de relations d'héritages et cela va créer beaucoup trop de classes.

Exemple

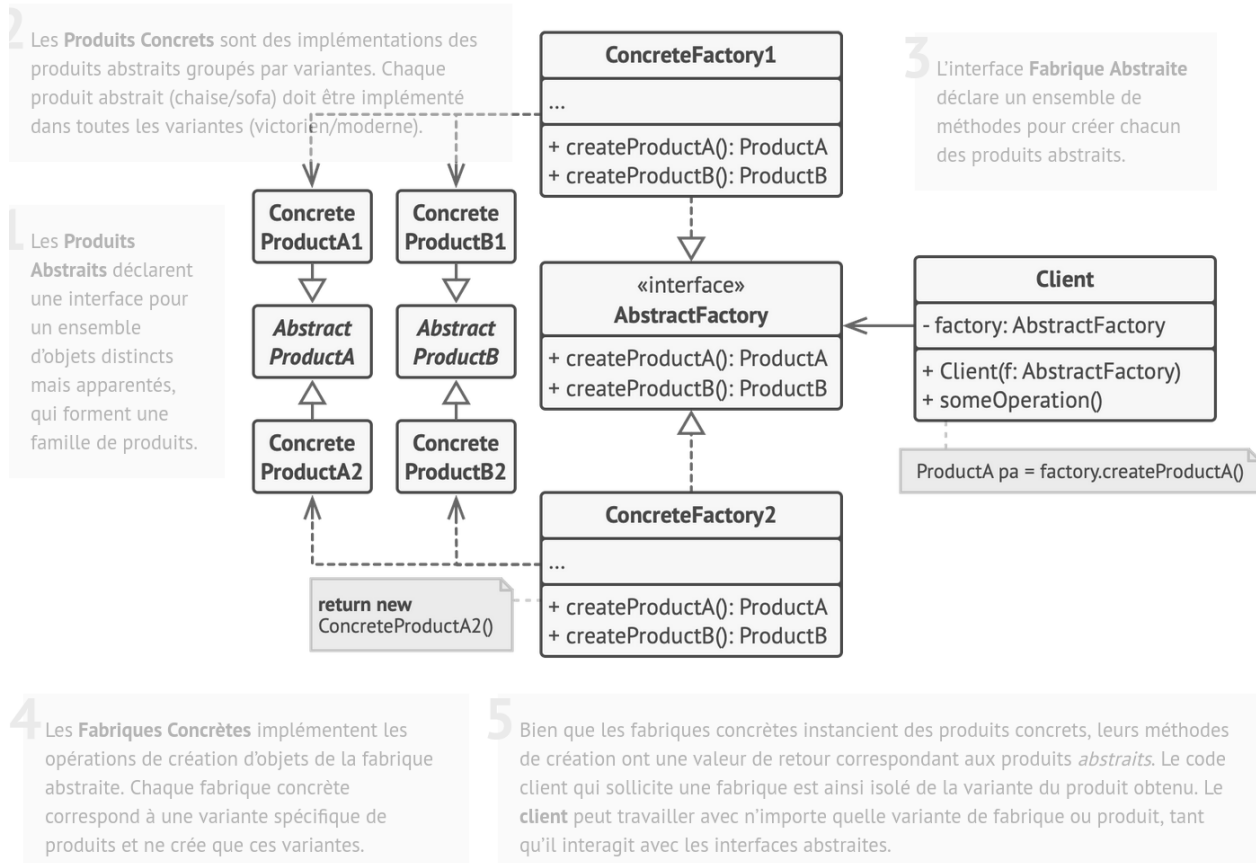
Pour donner un exemple de son utilisation et du bordel qu'il cause, imaginons que l'on aie des StormTroopers Wookiee en plus des humains. Selon ce pattern on devrait créer des sous fabriques héritant d'une fabrique principale abstraite. Ce qui donnerait ceci :



- Le **Creator** ici est la classe abstraite "StormtrooperTrainingFacility"
- Le **Product** ici est l'interface Stormtrooper
- Les **Concrete Products** sont les classes AssaultStormtrooper, GrenadierStormtrooper et PilotStormtrooper
 - Les classes WookieeAssaultStormTrooper, WookieeGrenadierStormTrooper et WookieePilotStormtrooper (et leurs équivalents humains) héritent de Concrete Products et sont donc aussi des Concrete Products
- Les **Concrete creators** sont les classes HumanStormtrooperTrainingFacility et WookieeStormtrooperTrainingFacility

Anti-pattern : Fabrique abstraite

Théorie



La fabrique abstraite est utilisée lorsqu'on a besoin de créer des familles d'objets sans préciser leurs classes concrete. Dans cette fabrique... tout est trop compliqué avec beaucoup trop de relations d'héritages.

Cette fabrique a tous les désavantages du précédent, en pire. Ne surtout pas l'utiliser. Encore une fois, ce patron favorise l'héritage à la composition.

Exemple

Vous ne voulez pas savoir...

Le pont

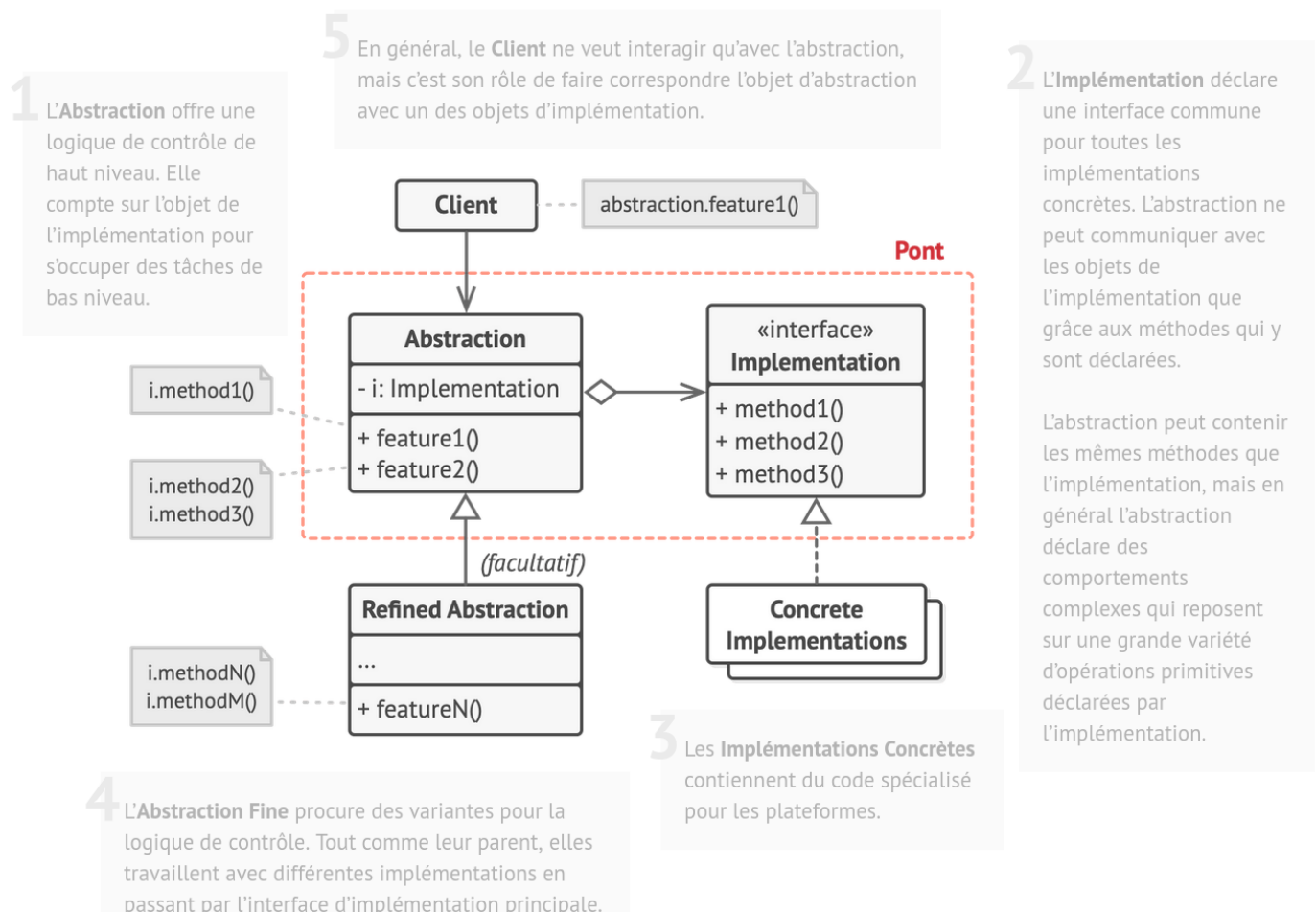
Théorie

Bien que les méthodes de fabrique (en français simplement appelée "Fabrique") et la fabrique abstraite sont listé comme des patterns dans le site Refactoring Guru, ils sont en vérité plus des anti-patterns (comme toute chose qui utilise l'héritage d'ailleurs).

Le problème de l'exemple avec la méthode de fabrique, est que l'espèce et la spécialisation du Stormtrooper étaient mélangées (par exemple dans une seule classe

"WookieAssaultStormTrooper", "HumanAssaultStormTrooper" ou encore "WookieGrenadierStormTrooper") et ce alors que leur spécialisation et leur espèce sont deux choses complètement distinctes.

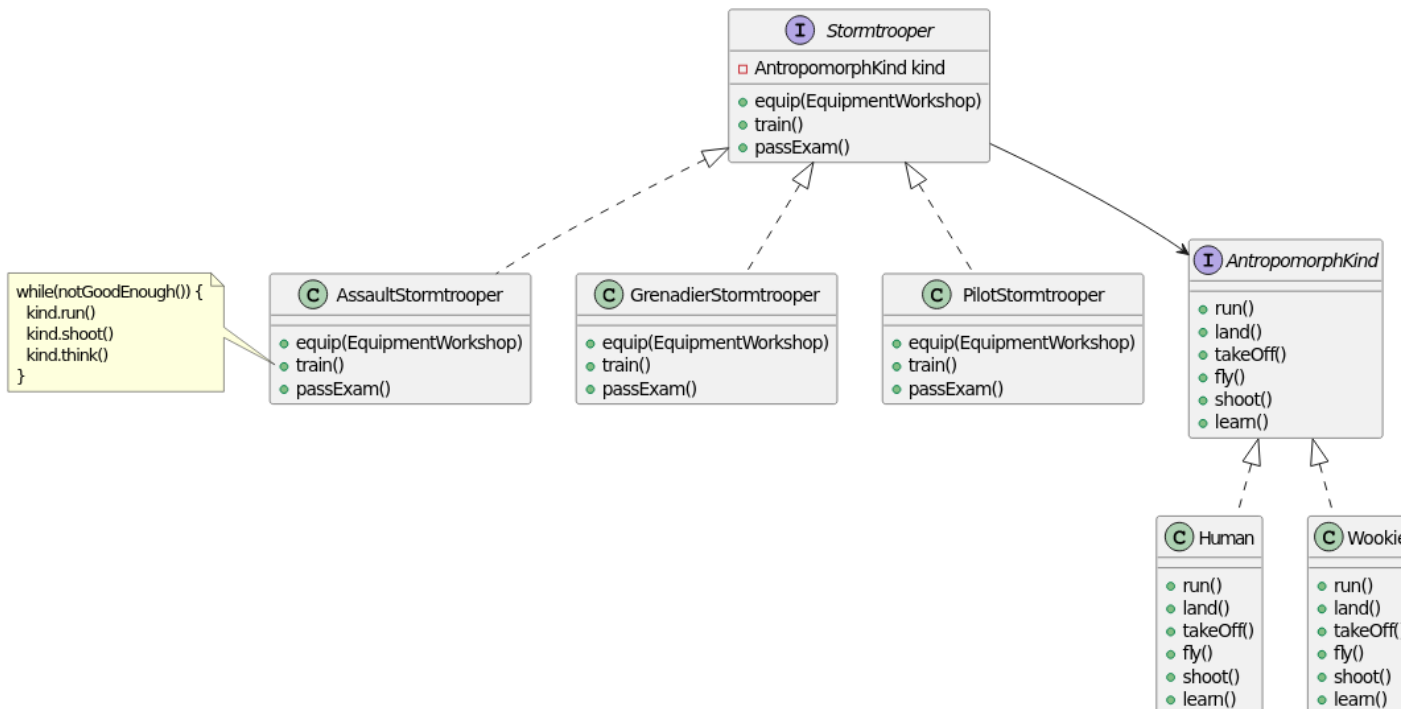
C'est là que le patron de conception du Pont entre en jeu, il sert à diviser ces grosses classes, permettant donc les espèces et les spécialisations d'évoluer indépendamment.



Ce diagramme donne met *Refined Abstraction* comme héritant de la classe *Abstraction* mais c'est une meilleure idée d'éviter l'héritage en transformant l'*Abstraction* en interface à implémenter

Exemple

Pour reprendre notre exemple de départ, à la place d'avoir un *WookieAssaultStormTrooper* on va avoir un *AssaultStormtrooper* qui a comme attribut "espèce" "Wookie". On a donc séparé les espèces et les spécialisations en deux ensembles de classes distinctes (on a donc séparé les préoccupations et favorisé la composition à l'héritage).



- L'**abstraction** ici est l'interface Stormtrooper
- L'**abstraction fine** ici sont les classes AssaultStormtrooper, GrenadierStormtrooper et PilotStormtrooper
- L'interface **implémentation** ici est l'interface AntropomorphKind (ou Espèce)
- Les classes **implémentations concrètes** sont ici Human et Wookie

Résumé

En résumé les deux patrons de conceptions à retenir ici sont ceux de la fabrique simple et du Pont.

- La fabrique simple permet de simplement créer des objets de différents types (de cette façon se concentrer uniquement sur la création des objets)
- Le pont sert à simplifier l'architecture du code de façon à séparer les préoccupations (par exemple préoccupation "Spécialisation" et préoccupation "Espèce")

Si on utilise correctement le pont on ne devrait jamais avoir besoin d'autre chose que de la fabrique simple pour créer les objets.

Revision #1

Created 28 September 2023 08:41:42 by SnowCode

Updated 3 October 2023 21:07:06 by SnowCode