LE PATTERN VISITEUR

Sommaire

Introduction aux patterns du GOF

Les design patterns sont des **solutions éprouvées** aux **problèmes récurrents** de conception logicielle, adoptées par la communauté de développement. Ils facilitent la structuration du code pour le rendre plus **maintenable, flexible et évolutif**.

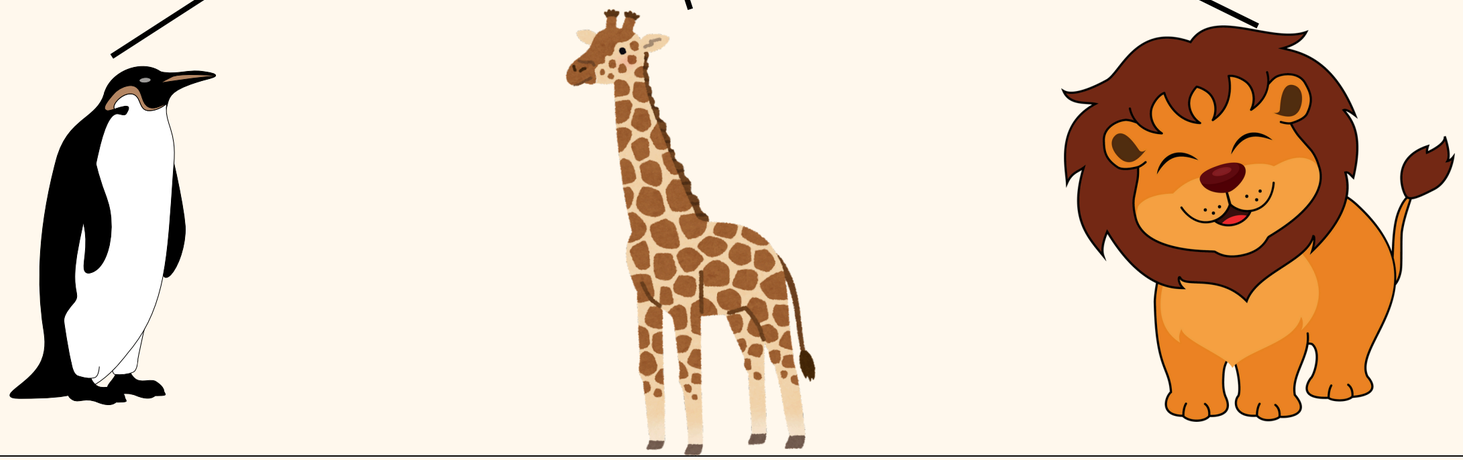
Il existe trois catégories de patterns :

- Créationnels
- Structuraux
- Comportementaux

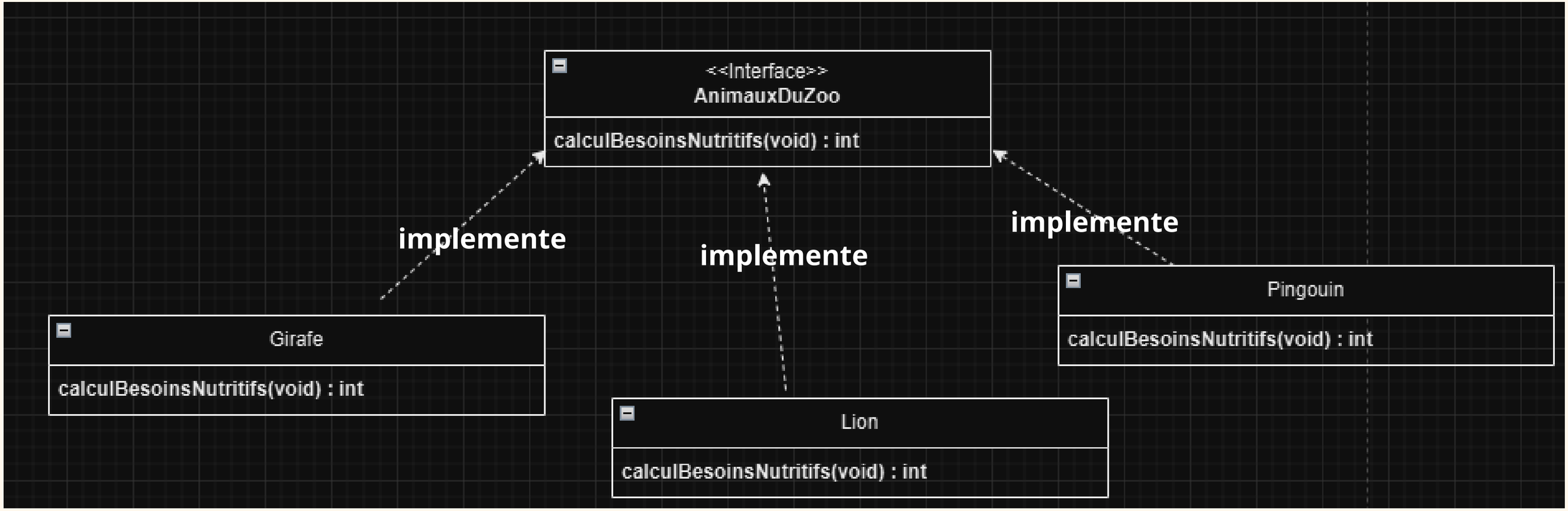
Un exemple avec une solution pas à pas.

Bravo, vous venez de décrocher un stage dans un zoo. On vous a assigné votre première tâche qui consiste à représenter les animaux du zoo et de calculer leurs besoins nutritifs afin que les soigneurs puissent au mieux effectuer leur travail

Besoins nutritifs



Vous vous souvenez bien de vos cours de Java et pensez qu'il serait judicieux de profiter de la puissance du polymorphisme. Vous décidez alors de créer une **interface** appelée **AnimauxDuZoo** avec une méthode **calculBesoinsNutritifs()**.



On voit ici un problème de conception :

L’idée de surcharger une méthode pour que chaque animal définisse ses besoins nutritifs semble bonne, mais elle viole le principe de Responsabilité Unique (SRP) du SOLID. Le modèle de données des animaux ne devrait pas gérer les calculs des besoins nutritifs, car il évolue plus lentement que la logique métier. Cela viole donc aussi le principe Open/Closed, parce que toute modification de la logique métier nécessite de modifier chaque classe animale.

On voit ici un problème de conception :

Peut-être êtes-vous déjà familier avec ces principes, alors vous décidez de prendre les devants et créez une classe **CalculateBesoinsNutritifs**. Vous avez fait une méthode qui contient le code suivant

```
if (animal instanceof Girafe) {
    //TODO: Logique spécifique pour Girafe
} else if (animal instanceof Lion) {
    //TODO: Logique spécifique pour Lion
} else if (animal instanceof Tigre) {
    //TODO: Logique spécifique pour Tigre
}
```

Avec cette solution, non seulement vous perdez les avantages du polymorphisme, mais en plus, vous brisez le principe **Open/Closed du SOLID.** Chaque fois qu'un nouvel animal est ajouté au zoo, vous serez obligé de revenir dans cette classe pour modifier le switch. On peut surement faire mieux 

Une solution pas à pas.

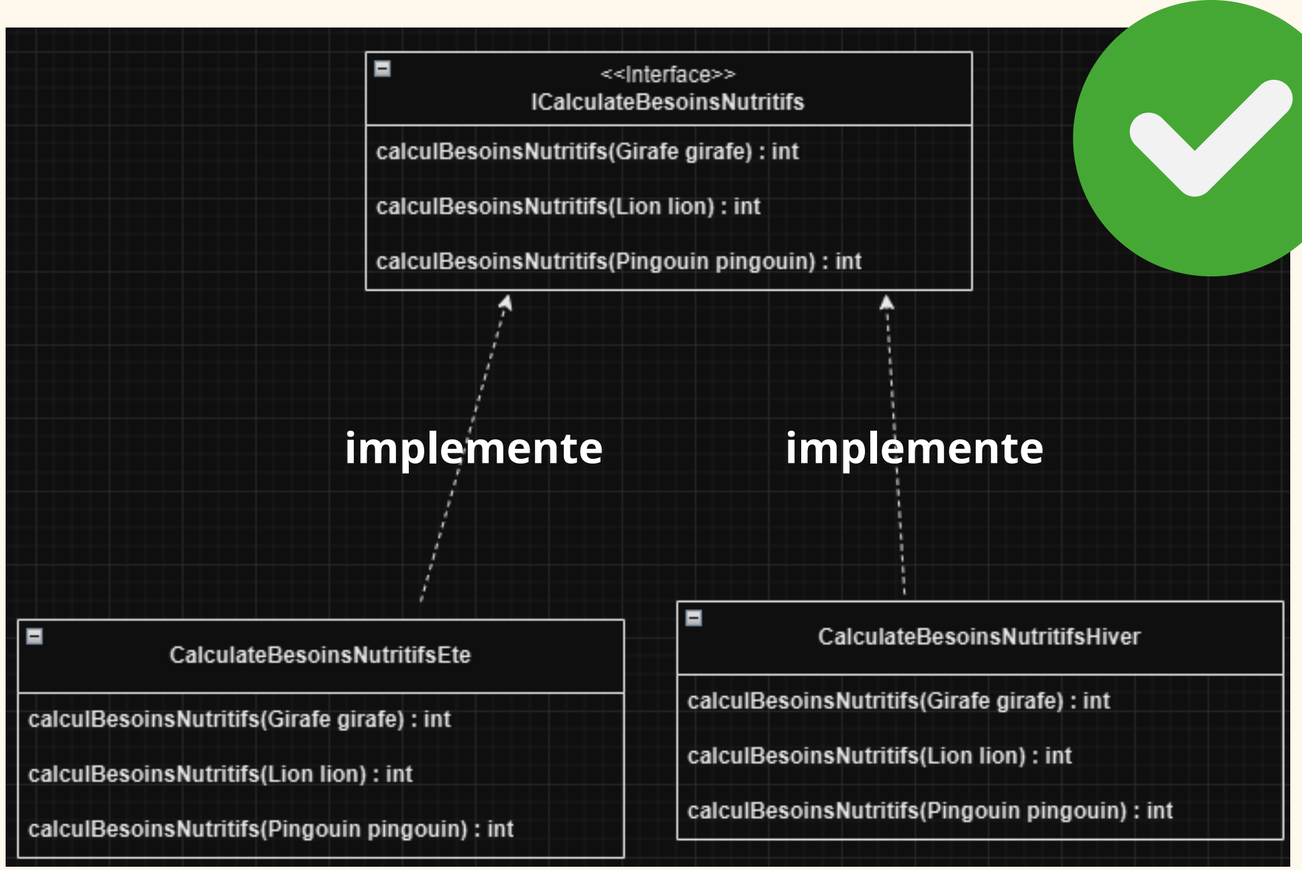
Pour commencer, on pourrait peut-être utiliser la **surcharge de méthode** en JAVA, c'est une approche intéressante, mais on peut peut-être la compléter pour la rendre plus efficace.

```java
public class CalculateBesoinsNutritifs {

    public void gererBesoinsNutritifs(Girafe girafe) {
        //TODO: Logique spécifique pour les besoins nutritifs de la Girafe
    }

    public void gererBesoinsNutritifs(Lion lion) {
        //TODO: Logique spécifique pour les besoins nutritifs du Lion
    }

    public void gererBesoinsNutritifs(Tigre tigre) {
        //TODO: Logique spécifique pour les besoins nutritifs du T
    }
}
```

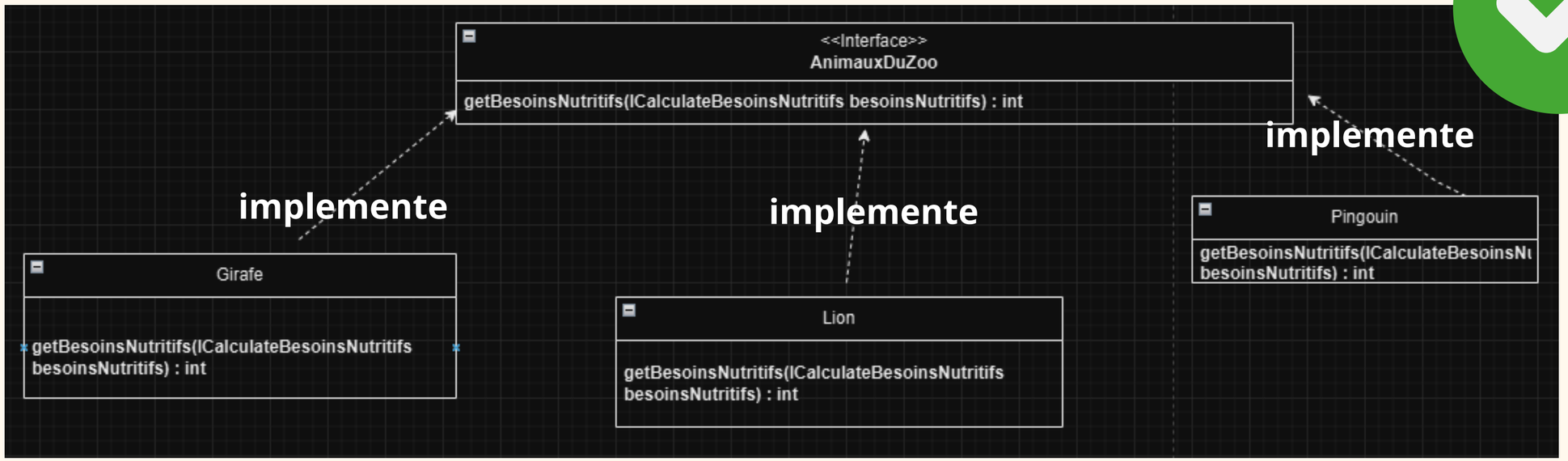
Les limites de cette approche :

Premièrement on est limité pour le nom de la méthode meme si ce n'est pas tres grave, de plus si on veut d'autre comportements comme le calcul des besoins nutritif en hiver cela risquerais à la fin de créer un problème de couplage élevé.

Pour régler ce problème nous allons créer **une interface** qui permettra de regrouper tout **ces comportements**



Maintenant que nous avons évité le **couplage fort**, nous allons exploiter le polymorphisme pour appeler efficacement la méthode appropriée. Pour cela, nous allons créer une méthode dans le modèle qui déterminera et invoquera la bonne méthode en fonction du type de l'élément. Ce mécanisme est connu sous le nom de **double dispatch** (on y reviendra plus tard). La methode se trouvera d’ailleurs directement dans l’interface, avec cela on s’assure que chaque animal a bien sa propre méthode de calcule définit.



Une application de la méthode dans la classe Lion

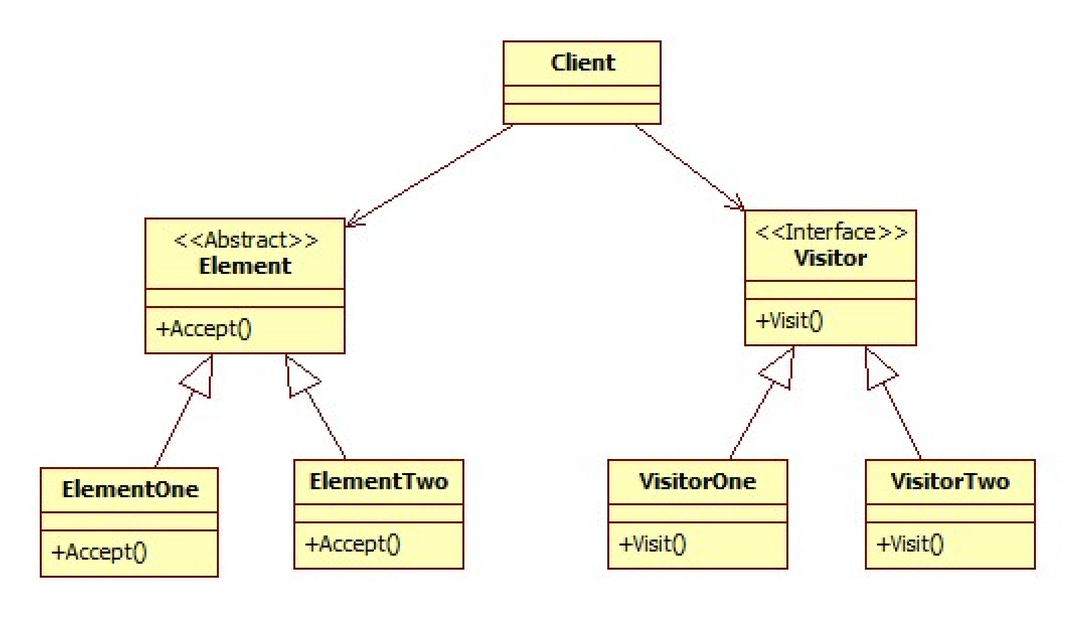
```java
public class Lion {
    public void getBesoinsNutritifs(ICalculateBesoinsNutritifs calculateBesoinsNutritifs) {
        calculateBesoinsNutritifs.calculBesoinsNutritifs(this);
    }
}
```

Le pattern visiteur dans la théorie ça donne quoi

- Le pattern visiteur permet de séparer les opérations des objets sur lesquels elles s'appliquent, en déléguant ces opérations à des classes visiteuses spécialisées

- Le pattern Visiteur est un pattern comportemental, car il permet de définir de nouvelles opérations sur des objets sans modifier leur structure. Il sépare les opérations des objets visités, facilitant l'ajout de comportements sans changer les classes d'objets

Comment ça marche



- **Une interface visiteur *:*** Qui contient les opérations et implémente une méthode **visit()** pour chaque classe visitable. Cette méthode prend en paramètre la classe visée.

- **Des classes visitables :** Implémentent une méthode ***accept()*** unique. Cette méthode prend en paramètre les visiteurs et fait appel à leurs méthodes ***visit()***, passant son objet comme paramètre.

- enfin et surtout le **double dispatch (diapositive suivante)**

Le Double Dispatch

Pour bien comprendre le Pattern **Visiteur**, il est nécessaire de comprendre le concept de **Simple** et de **Double Dispatch** en programmation.

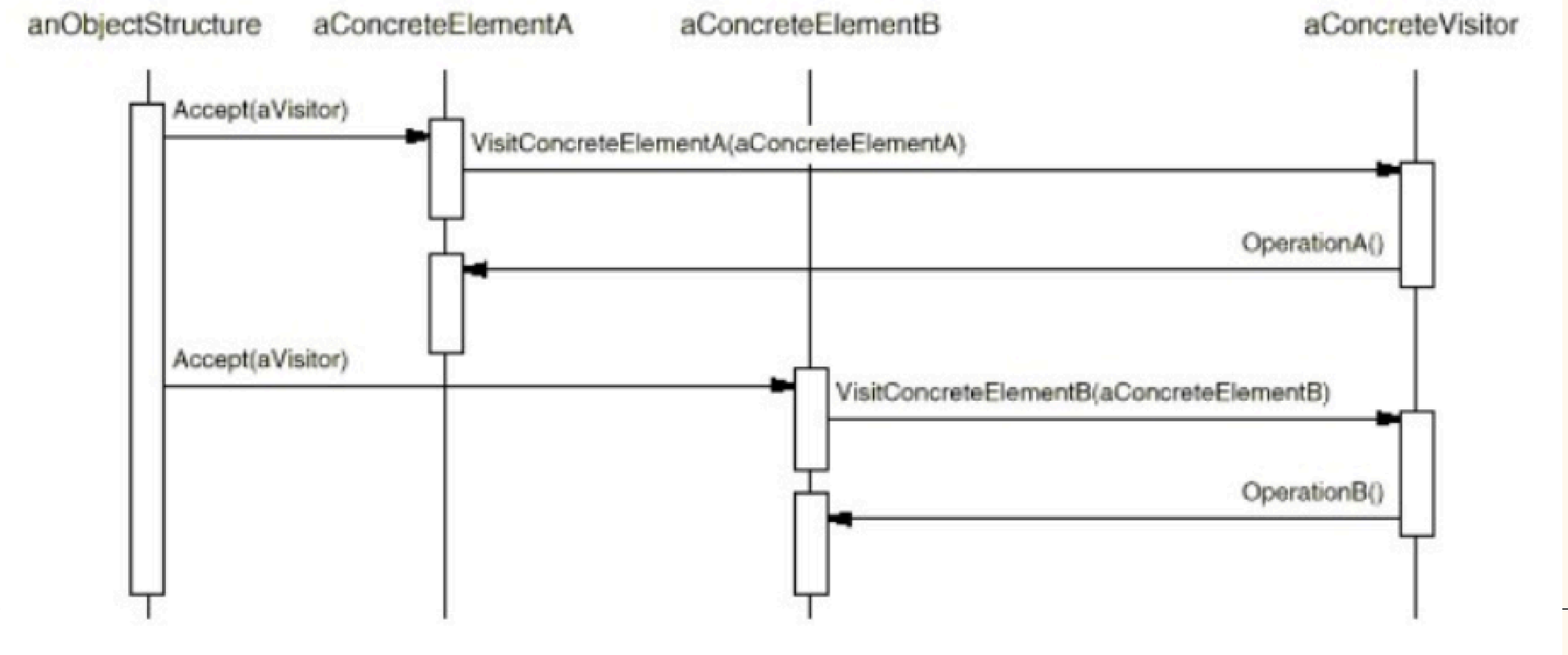
D’une manière assez simple, le concept de “Dispatch” influe sur la manière dont sont appelées les méthodes.

Dans le cas du Dispatch Simple : C’est le principe de base du polymorphisme et le Dispatch le plus courant. L’appel se fait sur un objet et l’implémentation dépend de ce dernier.

Dans le cas du Double Dispatch : La méthode invoqué dépend de deux objets, le receveur de l’appel (tout comme dans le simple dispatch) mais aussi l’objet en paramètre qui déterminera plus précisément la méthode !

Diagramme de séquence

Le **double dispatch** au niveau du **diagramme de séquence**



Les limites du pattern Visiteur

Bien que le pattern **visiteur** soit puissant pour **séparer** les opérations des structures d’objets, il présente plusieurs **limites :**

- Difficulté à ajouter de nouveaux types d'éléments
- Complexité accrue
- Couplage fort
- Problemes au niveau de l’encapsulation

- **Difficulté à ajouter de nouveaux types d'éléments :**
  - Chaque nouvel élément nécessite des modifications dans tous les visiteurs existants.

- **Complexité accrue :**
  - Gérer plusieurs visiteurs devient rapidement complexe avec une grande hiérarchie de classes.

- **Couplage fort :**
  - Le visiteur dépend fortement des classes visitées, augmentant la dépendance entre les composants

- **Problèmes au niveau de l’encapsulation :**
  - Le visiteur peut briser l'encapsulation en accédant aux détails internes des classes visitées (notamment déclaration de **méthodes publiques** qui n’auraient pas besoin de l’être sans le pattern).

Recentralisation avec le principe SOLID

- **S**ingle Responsibility Principle
- **O**pen/Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

- **SRP**
  - Le pattern Visiteur permet de respecter ce principe en séparant la logique métier (les calculs, traitements, etc.) des objets eux-mêmes. Chaque classe a une responsabilité claire
- **OCP**
  - Une fois la structure des éléments définie, vous pouvez ajouter de nouveaux comportements (visiteurs) sans modifier les classes des éléments
- **LSP**
  - Ce principe est respecté car chaque visiteur concret peut être substitué par un autre sans affecter le fonctionnement du code
- **ISP**
  - Le visiteur définit des interfaces spécifiques pour chaque type d’élément, évitant ainsi d’imposer des méthodes inutiles aux classes
- **DIP**
  - Le pattern Visiteur permet de respecter ce principe en introduisant une dépendance abstraite (l'interface Visiteur) plutôt qu'une dépendance directe à des implémentations concrètes

Parallèle avec le Pattern Décorateur :
Bien qu'on puisse penser que ces deux patterns ont des objectifs similaires, ce n'est pas exactement le cas !

Le pattern Décorateur permet d'ajouter des responsabilités ou des comportements à une classe par l'intermédiaire d'une autre classe qui la référence dans une structure d'enrobage. Il est principalement utilisé pour des ajouts combinatoires de comportements.

À l'inverse, le pattern Visiteur permet de déléguer des opérations à une classe tierce, permettant ainsi d'ajouter ou de modifier ces opérations sans pour autant modifier l'ensemble de la structure d'objets, qui est souvent complexe.

Un exemple d’association :
Imaginez un système utilisant des formes géométriques telles que des cercles, des rectangles, des triangles et d'autres.

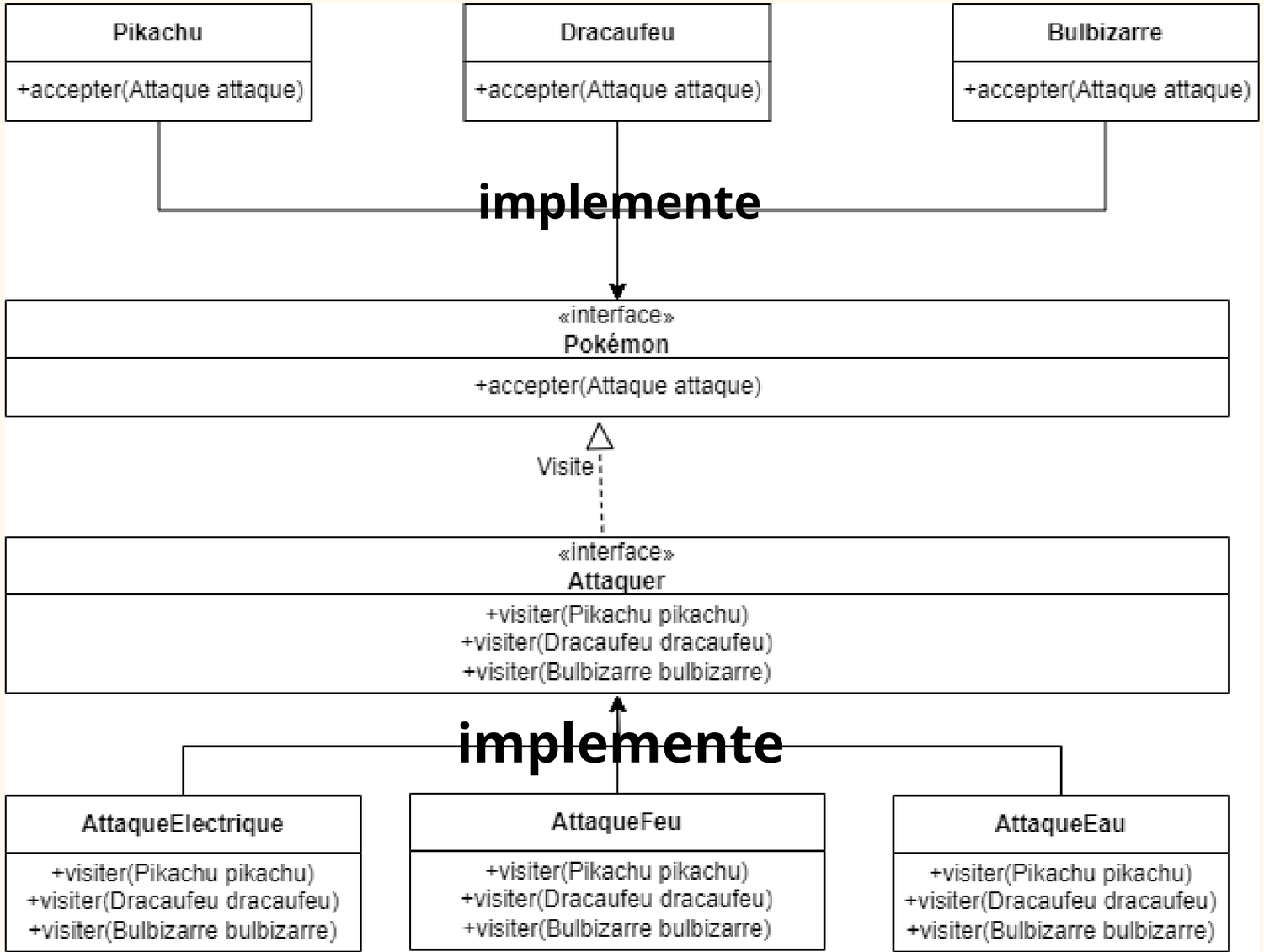
On pourrait décorer les différentes forment avec des comportements comme des bordures, des couleurs et d'autres éléments visuels n'impactant pas les opérations fonctionnelles. Tout ceci serait rajouté grâce au Pattern Décorateur.

On pourrait aussi implémenté des opérations fonctionnelles telles que le calcul de l'aire, qui serait délégué à un Visitor afin de ne pas surcharger les classes de formes et de ne pas devoir modifier chacune des classes en cas de modification ou d’ajout de nouvelle opération !

Vidéo live coding

[Vidéo]

Diagramme UML



Un petit questionnaire pour bien comprendre ?

[KAHOOT]

Bibliographie

- https://www.sfeir.dev/back/les-design-patterns-comportementaux-visiteur/
- https://www.youtube.com/watch?v=UQP5XqMqtqQ
- https://www.youtube.com/watch?v=wzfe7DbzgtI
- https://refactoring.guru/fr/design-patterns

MERCI ! 😉