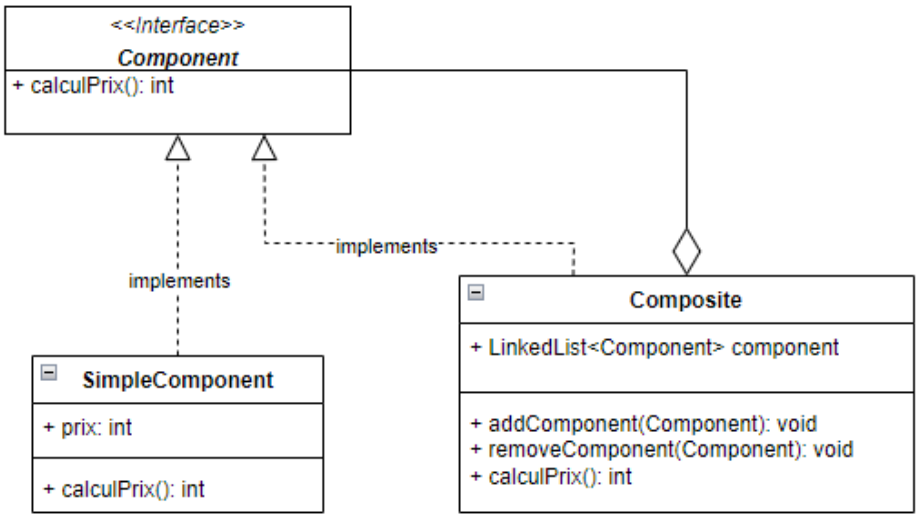
ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Séance 3

Cas n° 1

Pour le cas n°1, nous avons décidé d'utiliser le design pattern appelé **Composite**. Celui-ci est tout à fait compatible car permet de mettre en place des arborescences assez facilement. Ici, nous avons considéré l'objet **Composite** de base comme étant le PC. Ensuite, un ou plusieurs objets **Composite** pourront être ajoutés ou non à sa liste de composants (par exemple une carte réseau qui contiendrait de la RAM et un CPU). Si nous voulons ajouter un composant simple, c'est possible car la classe **SimpleComposite** implémente elle aussi **Component**. En appelant la méthode **calculPrix** du Composite de base (aka l'ordinateur), le prix de tous les composants sera récupéré en une seule fois.

Solution :



A titre d'exemple, voici le **main** utilisé dans ce cas :

```
//Composant principal
Composite c = new Composite("Ordinateur");

//Composants simples
SimpleComponent sc1 = new SimpleComponent(64, "Barrette de RAM");
SimpleComponent sc2 = new SimpleComponent(38, "CPU");

//On ajoute un composite (composant en contenant d'autres).
Composite c1 = new Composite("Carte réseau");
SimpleComponent sc3 = new SimpleComponent(72, "Barrette de RAM");
SimpleComponent sc4 = new SimpleComponent(44, "CPU");
c1.addComponent(sc3);
c1.addComponent(sc4);

//On ajoute tout ça à l'ordinateur
c.addComponent(sc1);
c.addComponent(sc2);
c.addComponent(c1);

System.out.println("Prix total des composants : " + c.calculPrix());
```

Source : https://refactoring.guru/fr/design-patterns/composite

ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Cas n° 2

Situation de départ : un code source quelconque possède déjà des classes et des interfaces. Ces interfaces sont implémentées par ces classes et leur permettent de faire des opérations. Ce code est déjà utilisé en production, mais il est prévu de devoir rajouter des opérations et des fonctionnalités aux classes existantes. Le problème est qu'il a été décidé de ne pas ajouter ces fonctionnalités, ni dans le code de l'interface, ni dans le code des classes, pour ne pas prendre de risques.

Pistes de solutions : il apparait clairement que le nouveau code à produire doit se placer dans une/des nouvelle(s) classe(s). Il convient alors de se demander : ces nouvelles classes vont-elles implémenter une interface ? Comment va-t-on lier les fonctionnalités de ces nouvelles classes aux classes existantes ?

Comment va-t-on lier ces nouvelles classes aux classes existantes ?

- D'une manière ou d'une autre, la classe existante va devoir appeler la nouvelle méthode. On va alors prévoir une nouvelle méthode dans la classe existante, qui appelle la nouvelle méthode grâce à une référence de la nouvelle classe, passée en paramètre. Mais dans ce cas, nous avons quand même dû modifier la classe existante ? Oui, mais c'était obligatoire et nous n'y avons pas placé le code fonctionnel.

Ces nouvelles classes vont-elles implémenter une interface ?

- Si nous partons du principe que chaque nouvelle classe représente une nouvelle fonctionnalité pour les classes existantes, alors oui ces nouvelles classes vont implémenter une interface permettant de les généraliser auprès de la classe existante, et de sa méthode décrite dans la dernière question.

Une seule nouvelle classe pour toutes les nouvelles opérations ?

- Doit-on créer une seule nouvelle classe à laquelle nous rajoutons toutes les opérations au fil du temps ? Non car cela finirait par saturer la classe et la rendre illisible ou incompréhensible.

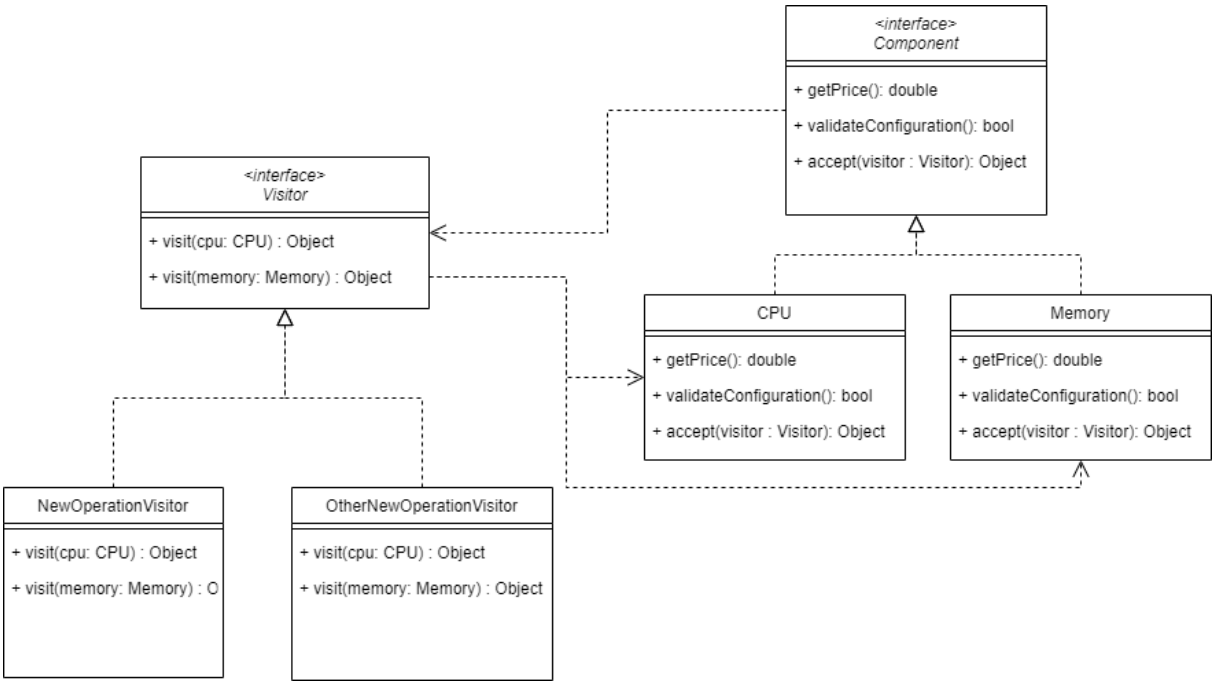
Si les fonctionnalités sont déclarées dans Component, quels principes ne sont pas respectés ?

- Comme nous l'avons dit, nos classes existantes ne peuvent pas être modifiées (dans la mesure du possible). Ainsi, en rajoutant une nouvelle méthode à l'interface Component, il nous faudra l'implémenter en cascade dans chaque classe existante implémentant Component. C'est dans ce sens qu'il nous est interdit de modifier les classes existantes. Le principe qui nous concerne ici est le principe **Open-Closed**, qui établit qu'une classe doit être extensible mais pas modifiable. Et c'est ce que nous avons réussi à faire ici, nous avons étendu les fonctionnalités de la classe, sans en modifier le code existant !

Démarche : recherchons les patrons de conceptions qui ressemblent à notre situation problème. Nous parcourons les catalogues en ligne et un des livres disponibles durant le cours (voir sources). Une fois que nous avons repéré le patron **Visitor**, nous tentons de comprendre son fonctionnement et nous assurons qu'il résout efficacement notre problème. Nous produisons ensuite un exemple concret en Java ainsi qu'un diagramme des classes UML.

ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Solution :



Code : voir Github.

Sources

- *Visiteur / visitor*. (s. d.). https://refactoring.guru/fr/design-patterns/visitor, visité le 03/10/2023
- Allen, P. R., & Bambara, J. J. (2007). *SCEA Sun Certified Enterprise Architect for Java EE Study Guide (Exam 310-051)*. (Ouvrage mis à disposition en classe)

ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Cas n° 3

Situation de départ :

L'objectif de ce cas est de pouvoir réaliser des opérations simples (additions et soustractions) à l'aide de classe telle que *Plus, Moins* ou *Nombre.* Ces classes doivent permettre de réaliser des opérations composées et dans l'idéal être le moins modifiées que possible.

Idée 1 :

Dans un premier temps, c'est le pattern de l'interpréteur qui a semblé le plus logique à implémenter. En effet ce pattern permet de décomposer une opération complexe en plusieurs opérations plus simples. A l'aide de ce pattern, on va pouvoir décomposer une suite de soustraction et addition en opération distincte à réaliser de manière récursive.

Pour le cas de la calculatrice, une interface *Expression* est mise en place, celle-ci permet de définir une méthode retournant un int du nom de *Résultat*.

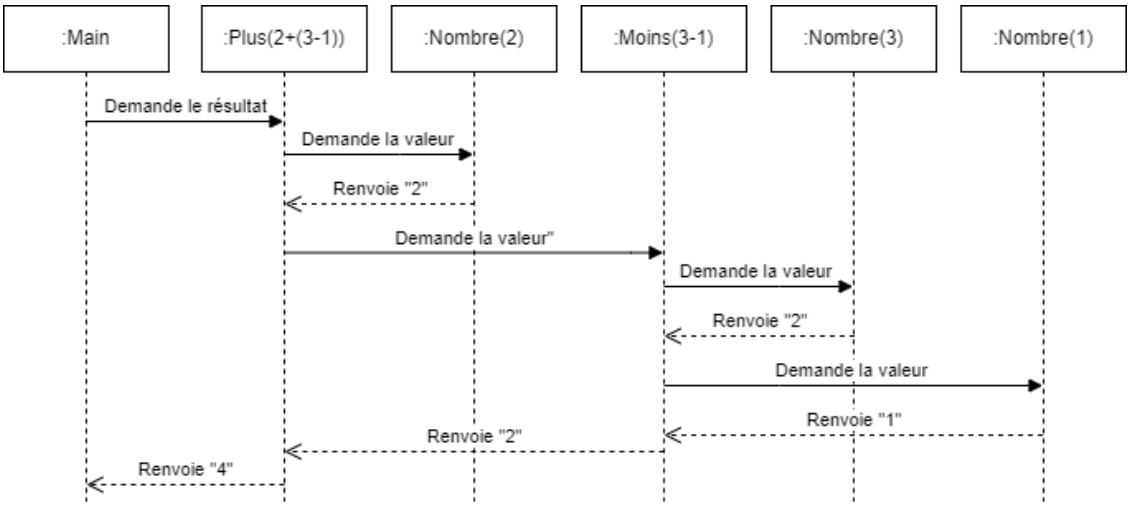
L'idée étant d'avoir trois classe implémentant cette interface :

- Nombre : une classe qui retourne comme résultat sa valeur.
- Plus : une classe permettant de recevoir deux objets implémentant l'interface *Expression* et qui retourne la valeur de la somme de ces deux objets.
- Moins : une classe permettant de recevoir deux objets implémentant l'interface Expression et qui retourne la valeur de la soustraction de ces deux objets.

A l'aide de ces classes, il est possible d'imaginer des calculs plus « complexes ». Par exemple :

```
//2 + (3-1)
Expression expression = new Plus(
        new Nombre(2),
        new Moins(
                new Nombre(3),
                new Nombre(1)
        )
);
int result = expression.Resultat() ;
```

⇨ result vaut 4 selon cette opération.



ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Le problème de ce pattern est qu'il faut ajouter de la logique métier dans les classes existantes, ce qui peut parfois être problématique. Afin d'en ajouter le moins possible, on pourrait utiliser le pattern du visiteur.

Idée 2 :

Le visiteur, va permettre de réaliser les opérations en dehors des classes *Plus, Moins* et *Nombre.*
D'une manière semblable au cas 2.

Source :

https://www.tutorialspoint.com/design_pattern/interpreter_pattern.htm

ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Cas N°4

Situation de départ

L'objectif de ce cas pratique est de chercher une solution afin de charger les images d'un site web uniquement au moment où l'image est à l'écran et non lors du chargement de la page. En procédant de cette manière, on imagine pouvoir accélérer l'ouverture de cette page en question.

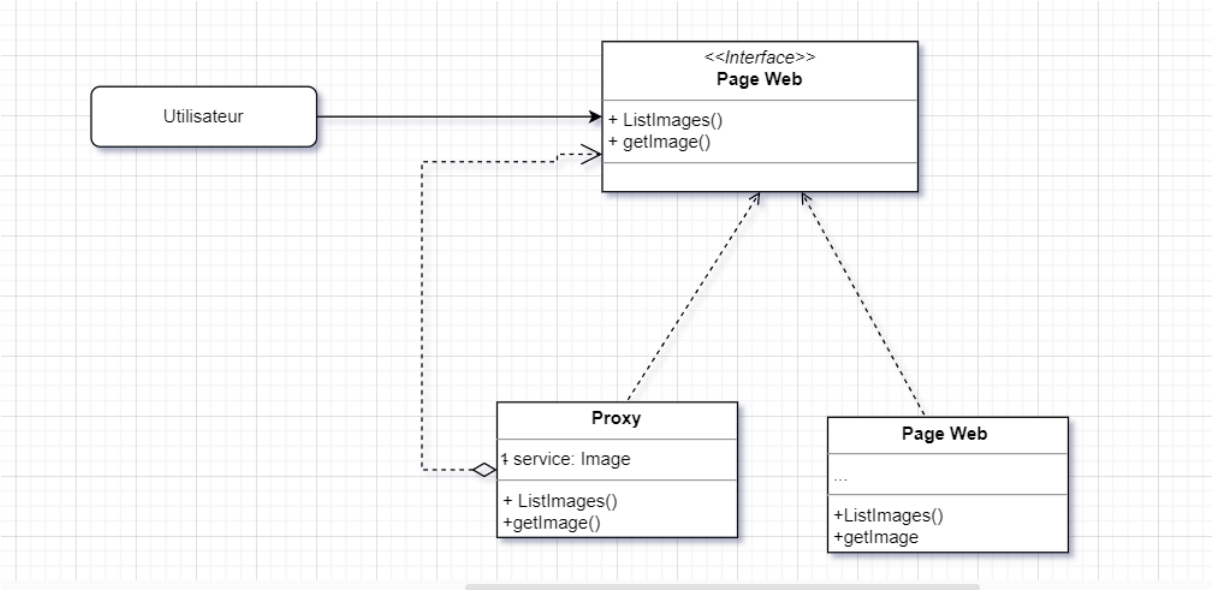
Solution :

Pour résoudre cette problématique, nous avons utilisé une balise HTML permettant de définir le chargement d'une image comme étant *lazy*. Cela va permettre de charger l'image lorsqu'elle sera dans la zone affichée à l'écran.

Comme on peut le constater cette méthode n'est pas un pattern de conception cependant il a un fonctionnement qui pourrait s'apparenter à un pattern existant : le proxy.

Le Proxy permet de donner le control d'un objet à un autre objet, ce qui va permet au nouvel objet d'effectuer des manipulations sur le premier.

Dans ce cas précis, on donne le control de l'image à la page web, c'est cette-dernière qui va s'occuper de lancer le chargement de l'image au moment souhaité.



ABDALLAH Noursine
BEN MOUSSA Reda
LEFRANCOIS Enzo
LUCAS Romain

Voici notre code HTML :

```
<!DOCTYPE html>
<html>
<head>
<style>
img {
  width: 100%;
}
</style>
</head>
<body>

    <!-- off-screen images -->
    <img src="images/img3.jpg" alt="deve"  style="width:100%"  loading="lazy">
    <img src="images/img4.jpg" alt="dev" style="width:100%" loading="lazy">
    <img src="images/img5.jpg" alt="oui" style="width:100%" loading="lazy">
    <img src="images/img6.jpg" alt="Oc" style="width:100%" loading="lazy">

</body>
</html>

```

Sources :

https://refactoring.guru/fr/design-patterns/proxy

W3Schools Online Web Tutorials