Pattern Visitor

- 6 Minutes chacun

Paul
Maeva
Florian
Mathias

● Introduction générale sur les patterns

Le design pattern, ou modèle de conception, est un élément très utilisé et important de nos jours, dans la programmation orientée objet (POO).

Les patterns sont des **solutions classiques** à des **problèmes récurrents de la conception** de logiciels. Ce sont des sortes de plans ou de schémas que l'on peut adapter afin de résoudre un problème récurrent dans notre code. On peut aussi voir cela comme un plan pour un architecte, c'est d'ailleurs de là que nous viennent les patterns. L'utilisation de plan a d'abord été utilisée en architecture puis en anthropologie. C'est en 1987 que Kent Beck et Ward Cunningham réalisent que le principe de modèle de conception est applicable dans le milieu de la programmation.

C'est ainsi qu'en 1994, est née l'oeuvre qui est devenue, et reste aujourd'hui la référence dans le monde des design pattern, le "*Design Patterns : Elements of Reusable Object-Oriented Software* " (un peu la Bible des développeurs) qui à été écrit par 4 auteurs: Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, connu sous le nom de *"gang of four*", le GOF.

Ces patterns se divise en 3 catégories.

- **Créationnel,** ce sont les patterns qui examinent le moyen de résoudre les problèmes de conception découlant de la création d'objets. Pour faire plus simple, leur but est la création de nouveaux objets tout en augmentant la flexibilité et la réutilisabilité du code (singleton, abstract factory…)

- **Structurel,** ce sont les patterns qui facilitent la conception en identifiant un moyen simple de réaliser des relations(heritage,composition) entre les entités. Concrètement, ces patrons vous guident pour assembler des objets et des classes en de plus grandes structures tout en gardant celles-ci flexibles et efficaces.

- **Comportemental**, dans cette dernière catégorie, les patterns identifient les modèles de communication communs entre les objets et améliore le code. Ces patrons s'occupent des algorithmes et de la répartition des responsabilités entre les objets.

Mais pourquoi les patterns sont-ils si utilisés ? Car il permettent beaucoup de choses, comme par exemple d'accélérer le processus de développement d'une application, en appliquant une solution qui a fait ses preuves, mais aussi, comme nous avons pue le voir avec les différentes définitions données plus haut, les patterns permettent surtout de respecter le principe d'un code SOLIDE. Mais la plupart du temps, les design pattern sont appliquées au moment de la phase de conception. avant même que des problèmes apparaissent dans la phase d'implémentation.

- ## Énonciation du besoin

Prenons désormais une situation concrète, supposons que nous sommes un développeur logiciel, et que nous travaillons pour une compagnie d'assurance. Notre compagnie nous informe qu'elle a plusieurs catégories de client différents, comme les banques, les particuliers ou les entreprises par exemple. Il faut donc que nous modélisions cette situation, nous passons donc en phase de conceptualisation.

- ## Première conception

prob car respecte pas SOLID notamment le principe OCP et SRP

Durant la phase de conception, nous décidons de plusieurs choses afin de simplifier et d'optimiser au maximum notre code. Nous savons que nous avons plusieurs catégories de clients, et que ces derniers partagent des éléments communs comme un nom, une adresse et un numéro de téléphone. Nous allons donc jouer sur le polymorphisme et l'héritage afin de simplifier notre conception. En partant de cette idée, nous obtenons le diagramme de classe suivant :

Tout se passe pour le mieux, nous venons juste de finir notre implémentation, notre code est prêt, finalement pourquoi parler des patterns, on se débrouille très bien sans ! Mais au même moment, votre responsable débarque dans notre bureau et nous demande d'implémenter une nouvelle fonctionnalité,une messagerie publicitaire à destination des différentes catégories de clients. Votre responsable vous donne donc plus de détails sur cette fonctionnalité.

En fonction de la catégorie du client, un courrier particulier sera envoyé pour l'informer des nouvelles offres de l'assurance. Par exemple si le client est un résident, une annonce pour une assurance maladie lui sera envoyée ou si le client est une banque une assurance de vol lui sera envoyé.

Il est assez simple de régler ce problèmes, il suffit de définir une méthode abstraite "sendMessage" dans la classe client, et grâce au polymorphisme mis en place dans notre projet pendant la conception, nous aurons juste à redéfinir cette méthode dans les classe qui hérite de la classe client, les différentes classe représentant les catégories de client. Ensuite, il suffira d'appeler pour chaque catégorie de client sa méthode "sendMessage", et le tour est joué !

Nous avons fait des tests pour vérifier le bon fonctionnement de notre méthode après implémentation, nous obtenons les résultats suivants :

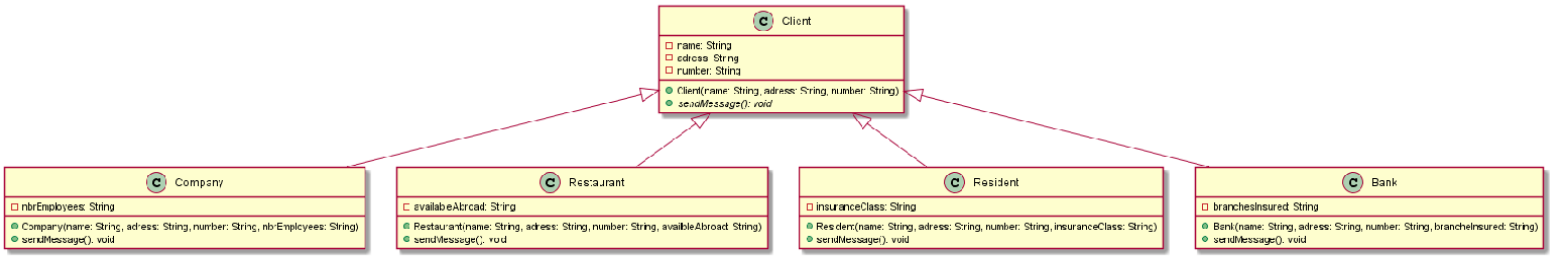
```
Send message about theft insurance
Send message about equipments insurances
Send message about food insurance
Send message about medical insurance
```

Notre méthode fonctionne donc correctement.

Mais notre code est-il aussi bien qu'il n'y paraît ?

- **Identification des problèmes et seconde modélisation :**

Jetons un oeil à notre nouveau diagramme de classe, obtenue par retro-engineering, comportant désormais la fonctionnalité demandé par notre responsable :



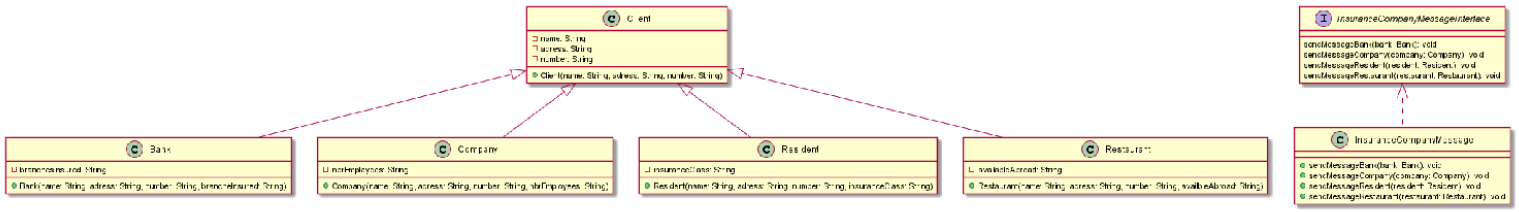
En étudiant notre implémentation, on se rend compte qu'il y à plusieurs problèmes importants dans cette conception, certes notre code est fonctionnel, mais il n'est pas SOLID du tout. Notre projet est en implémentation, il se pourrait par exemple que notre méthode sendMessage soit ensuite dépendante de système comme par exemple gMail, or est ce que c'est à l'objet Bank par exemple, de s'occuper de cela ?

Non ! donc notre code viol le SRP, soit Single Responsability Principle, notre objet a plusieurs responsabilitées, ce qui n'est pas SOLID du tout !

Pour continuer on peut aussi s'intéresser à un autre aspect de la méthode sendMessage, si notre responsable venait à nous demander de faire évoluer notre méthode, que se passerait-il ? Il faudrait ouvrir notre code pour le modifier, nous venons de détruire un autre principe SOLIDE, le OCP, Open Closed Principle. Notre code doit-être ouvert au extensions et fermé au modifications. Or si des modifications venaient à être devoir effectué, il faudrait changer le code dans toutes nos classes une à une.

Il faudrait donc extraire ces comportements de ces classes pour pouvoir les modifier en dehors de cela dans le but de rendre notre code optimal et SOLID. Essayons donc de faire cela, c'est à dire d'isoler notre méthode du reste de notre code :

Pour cela, pourquoi ne pas utiliser une interface qui réunira toutes les différentes méthodes implémentés dans les différentes classes bank, Resident, etc en jouant sur leur signature. Ensuite, nous pourrions faire une classe qui implémente nos méthode "sendMessage", en proposant une implémentation différente pour chaque méthode. Nous nous mettons donc au travail, est obtenons le diagramme suivant :

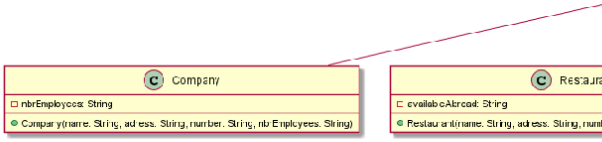


Nous pouvons maintenant voir que nous avons bien séparer d'un côté nos classes métiers, de l'autre il y a notre méthode sendMessage, mais maintenant, comment pouvons-nous faire pour lier ces deux parties ensemble, vérifier la classe a chaque fois pour appeler la méthode ?

Cela nous donnerait le code suivant :

```
if(cl instanceof Bank) {
    insuranceCompany.sendMessageBank((Bank) cl);
} else if (cl instanceof Company) {
    insuranceCompany.sendMessageCompany((Company) cl);
}
```

Celà n'est clairement pas optimal, notre code ne ressemble pas à grand chose, et si l'on essaie d'utiliser la surcharge le code ne compilera même pas, car il faudra caster l'expression avec l'un de nos types de catégorie. Il existe une solution qui peut nous sauver la technique du ***"double dispatch".*** Comme les objets pourquoi ne pas laisser le choix de la méthode directement à



nb double dispatch : une interface qui contient une méthode **visit** par type d'objet visitable (ainsi on se sert du typages dynamique pour lors de l'utilisation appeller la bonne méthode – double dispatch).

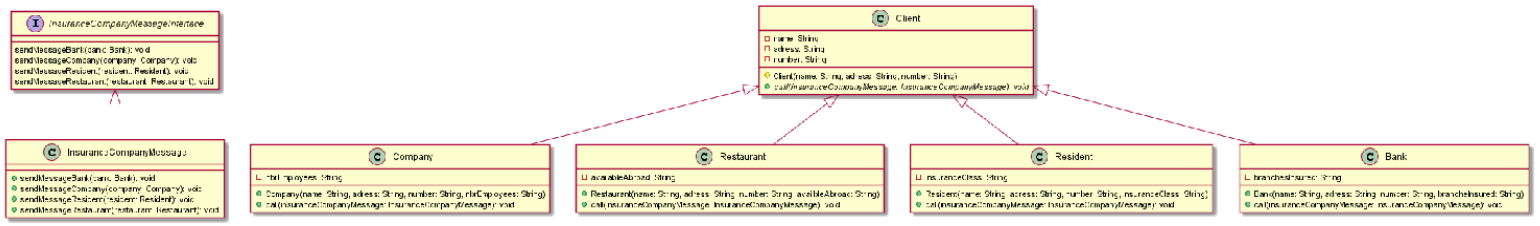
En termes de conception et d'implémentation, l'idée semble être un bon compromis, il suffit seulement de faire en sorte que lorsque l'on appelle la méthode sur notre objet, et bien cette méthode appelle la méthode qui se trouve dans la seconde partie de notre code, la méthode que nous avons implémenté à l'écart de nos classe métier, cela nous donne donc le code suivant :

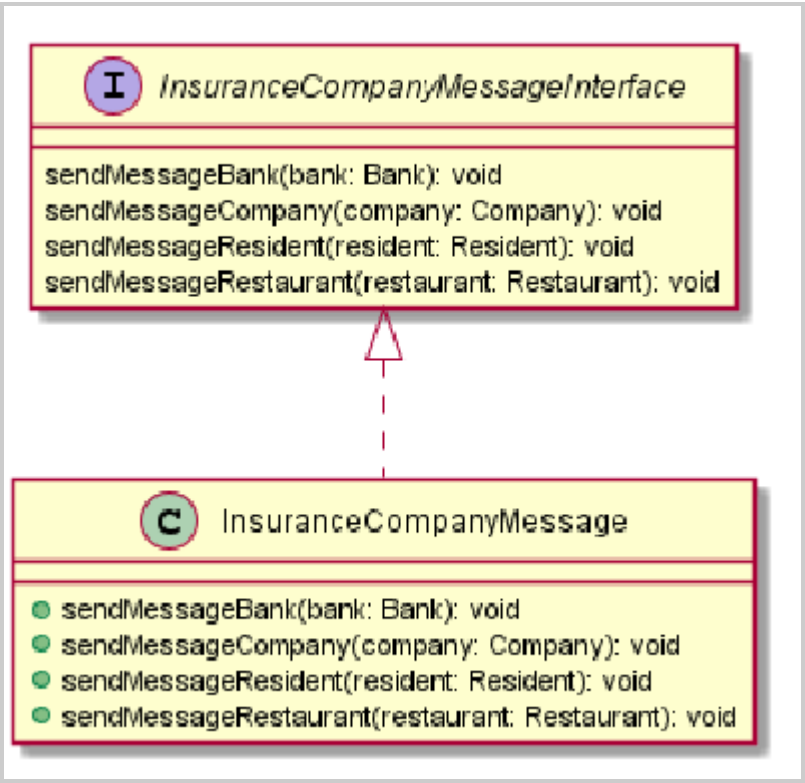
```
@Override
public void call(InsuranceCompanyMessage insuranceCompanyMessage) {
    insuranceCompanyMessage.sendMessageCompany(this);

}
```

Grâce à cela, notre code est fonctionnelle est nous pouvons désormais comme avant, par le simple biai des instructions suivantes exécuter la méthode sendMessage sur tous les objets.

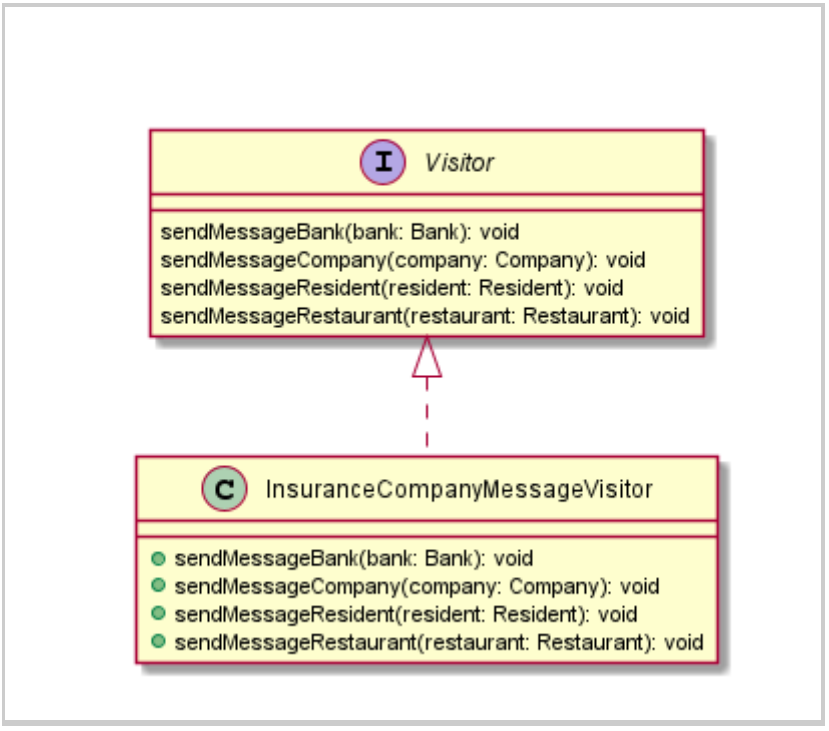
```
for(Client cl : client) {
    cl.call(insuranceCompany);

}
```

Notre programme est maintenant un peu plus propre et surtout bien plus SOLID. Mais regardons maintenant le diagramme de classe :



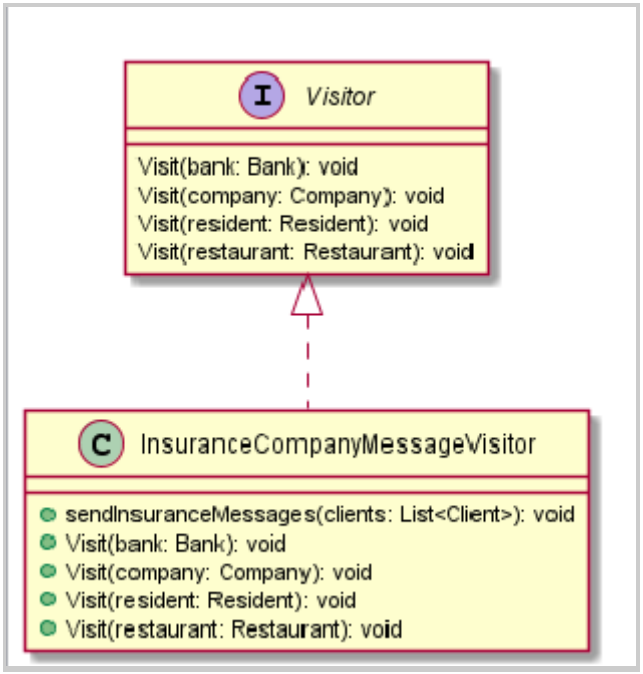


Le nom de notre interface n'est pas terrible, car si ensuite l'on veut rajouter d'autres comportements, ce nom ne sera plus valable. Actuellement que fait notre interface, elle définit des méthodes spécifiques pour appliquer des traitements sur nos objets, on peut dire que ces méthodes "visitent" notre classe, elle n'apporte pas de modifications sur ces dernières, nous pourrions donc faire une interface nommée "visiteurs", qui réunit ces méthodes qui "visitent" chacune de nos classes.

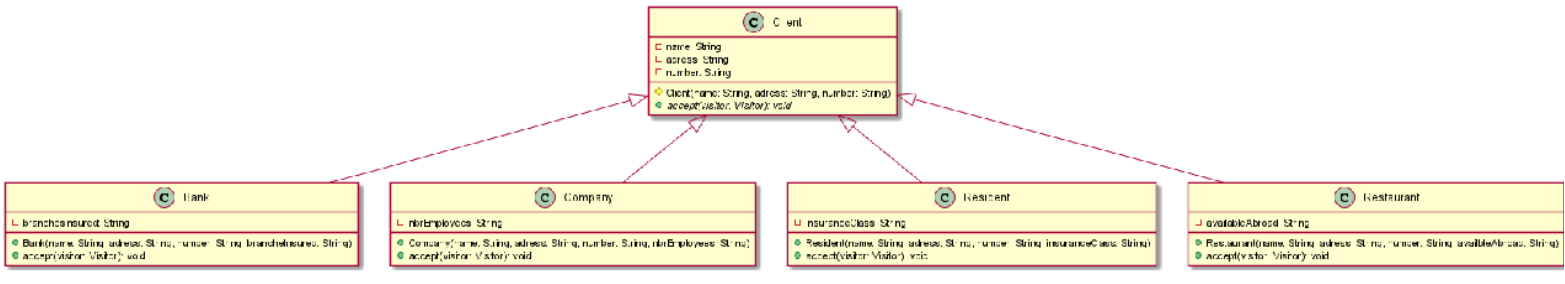


Après un solide refracting de nos classes, nous obtenons les éléments suivants :

Nos visiteurs

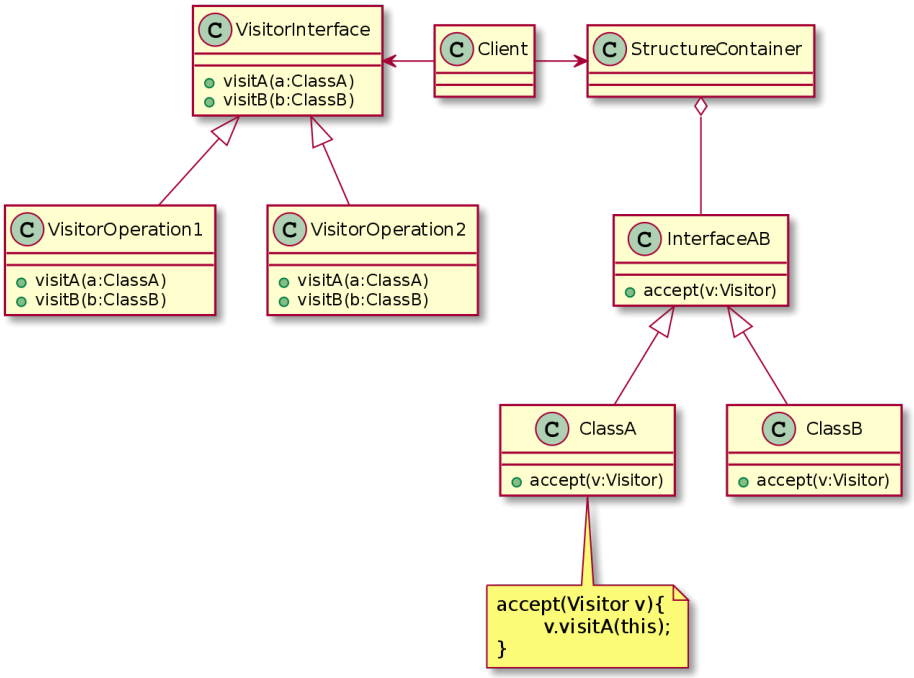


Classes métiers



- Généralisation du diagramme

Pourrions nous utiliser ce que nous venons de mettre en place dans d'autre projet futur, si l'on essaie de généraliser cela par un diagramme, nous obtiendrons le diagramme suivant :



Au premier abord ce diagramme peut sembler complexe mais regardons plus en détails. Nos classe *“ClassA”* et *“ClassB”*, possèdent une méthode "accept()” qui nous permet d'injecter notre visiteur. On peut donc adopter cette conception à chaque fois qu’il faudra que l’on sépare nos classes des traitements qui doivent être effectués dessus.

explication du diagramme en détail à l’oral

En même temps les visiteurs concret “VisitorOperation1”, “VisiteurOperation2” implémente une interface qui contient une méthode visit par type d'objet visitable.

Dans notre objet visitable on appellera donc la méthode visit de notre visiteur en lui passant l’objet lui-même.

- Généralisation sur le pattern Visiteur

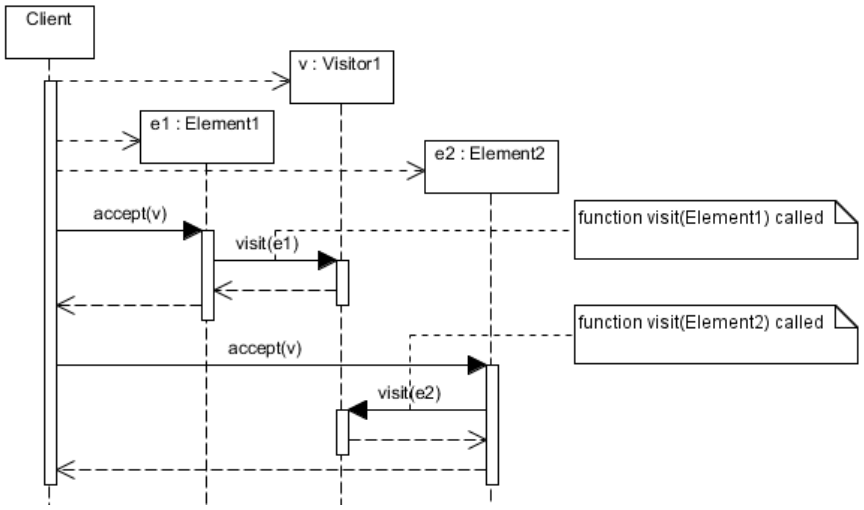
Bien que nous soyons des génie de l’informatique, cette méthode ne vient pas de notre imagination mais d’un pattern comportemental : le visiteur pattern. Le Visiteur pattern permet donc de séparer de manière claire un algorithme d’une structure de données. Les nouvelles fonctionnalités ne sont pas implémentées dans la classe mais dans des classes externes.

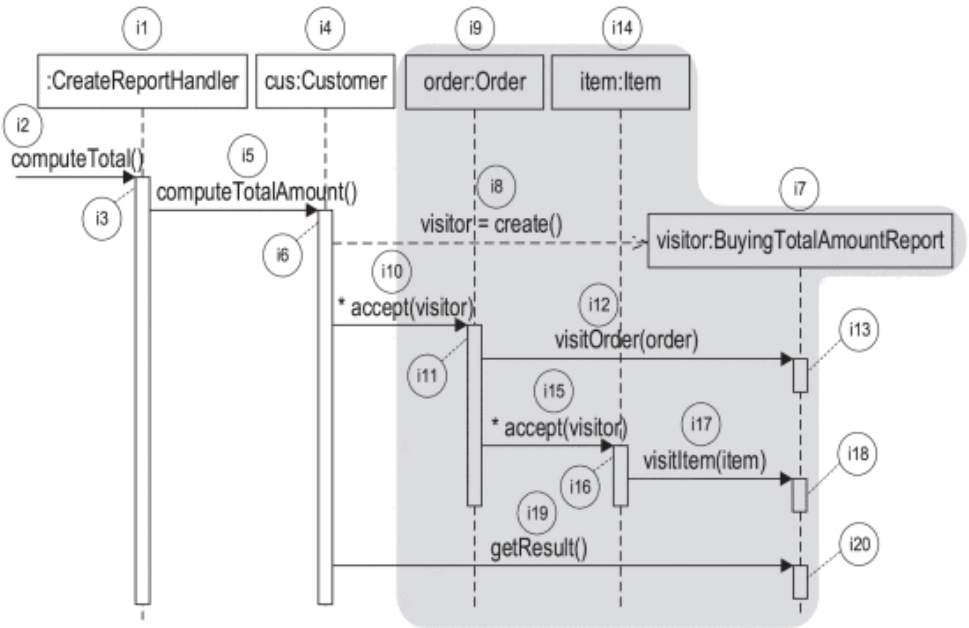
On limite ainsi tout couplage entre les données et leurs traitements ce qui permet d’ajouter des fonctionnalités à nos structures sans avoir à les modifier.

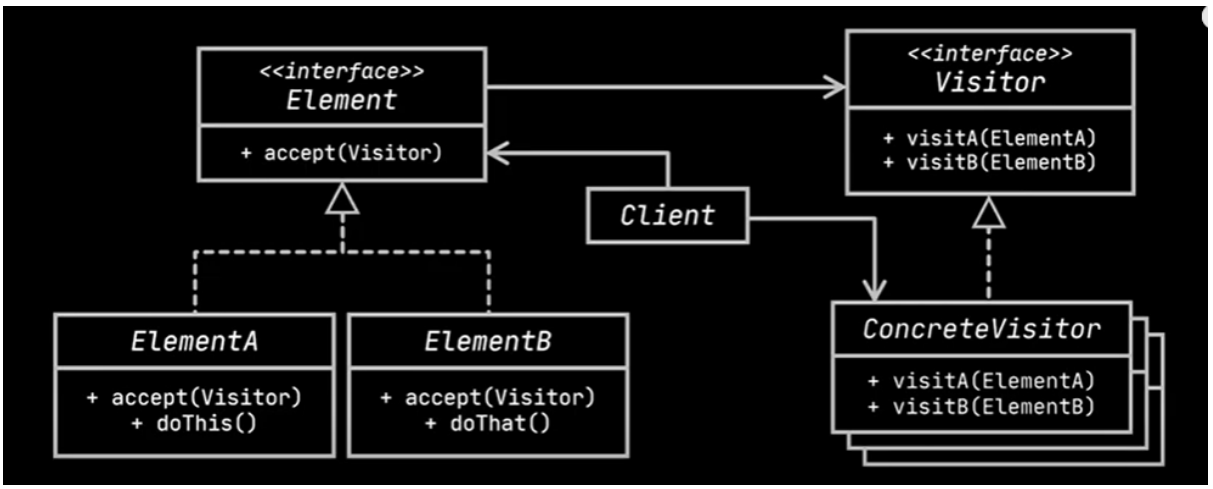
- Énoncer clairement la problématique de ce pattern

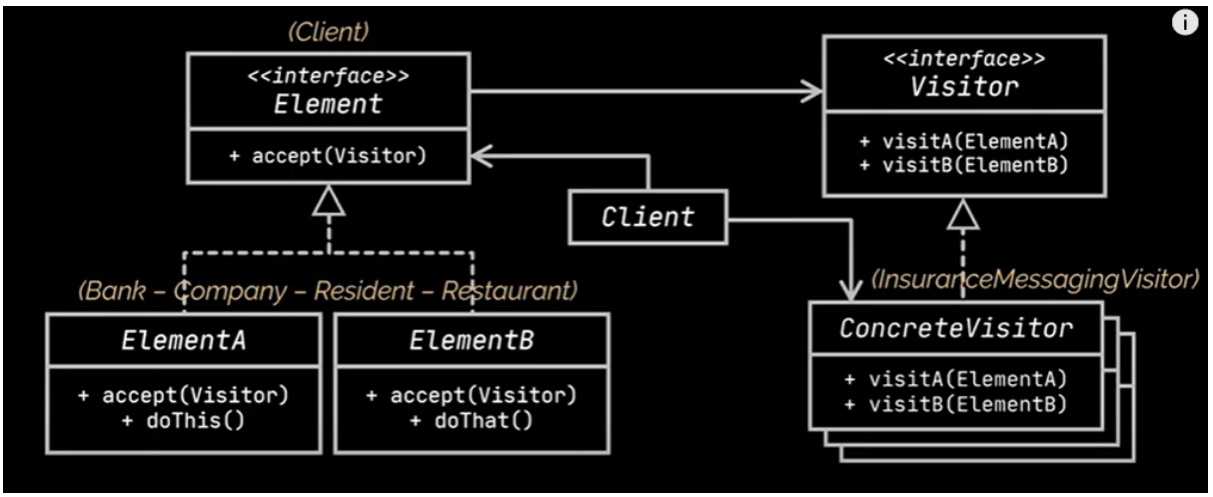
Le visitor pattern est fait pour répondre à plusieurs problématiques : effectuer un traitement sur plusieurs objets mais de manière différentes pour chaque objet (notre méthode sendMessage qui devait effectuer une tâche différentes en fonction de la catégories du client), mais aussi respecter les principes SOLID. Le **visiteur (Visitor)**, permet d'**ajouter de nouvelles fonctionnalités virtuelles** à une famille de classes sans avoir à les modifier. Il est donc bien adapté pour réussir à respecter des principes comme SRP ou OCP. Enfin, ce pattern évite de modifier nos classes, ce qui peut se révéler très utile si par exemple, il y a un nombre de livraison limité de notre application par mois.

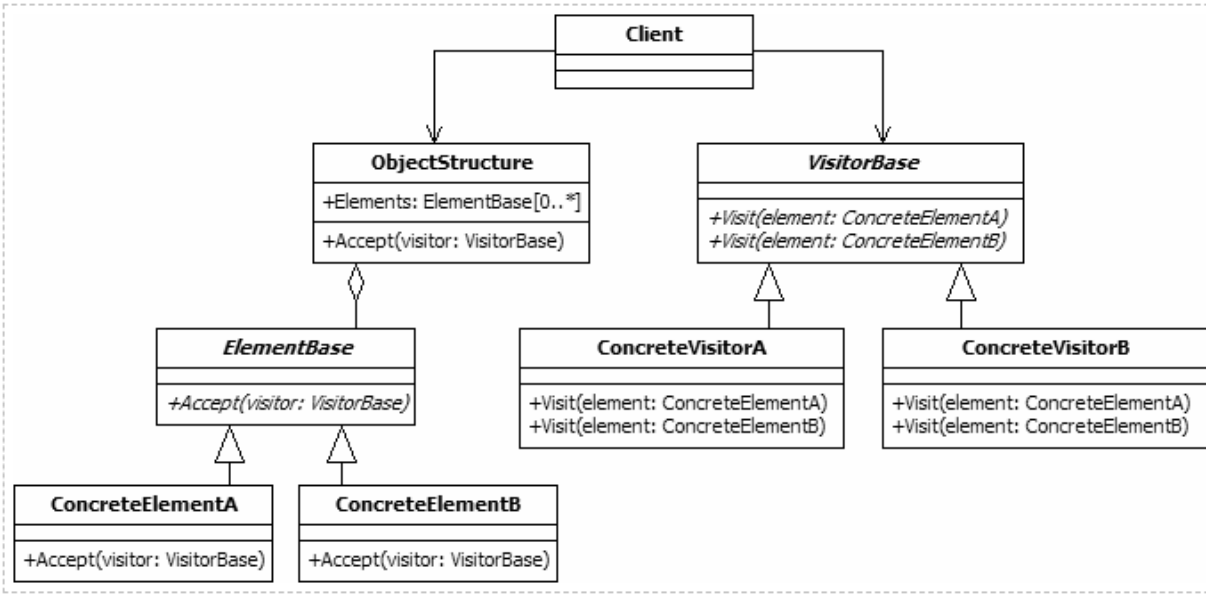
- Solution du pattern donné par la littérature (pour le Gof : diagrammes de classe/diagrammes de séquences)











"Détaillement des classes participantes faite à l'oral"

regarder la vidéo suivante, il explique bien le rôle de chaque élément
https://www.youtube.com/watch?v=UQP5XqMqtqQ

- Lien avec les principes SOLID, en quoi ce pattern permet d'obtenir un code plus maintenable.

  diapo

- Limites du pattern

  Malgré l'efficacité de ce pattern, il comporte quelque défauts. En adoptant les principes de ce pattern, les développeurs doivent être conscients des éléments suivants.

  Tout d'abord toutes les modifications apportées à la classe d'un élément, même les plus petites, entraînent généralement un ajustement des classes visiteurs affectées. Par ailleurs, ce pattern n'évite pas le travail supplémentaire nécessaire pour introduire de nouveaux éléments a posteriori puisqu'il faudra également implémenter pour ces éléments des méthodes visit() qui devront à leur tour être ajoutées dans les classes ConcreteVisitor. Enfin les visiteurs n'ont parfois pas les accès nécessaires aux attributs ou méthode privés des éléments qu'ils sont censés manipuler. La remarquable modularité des unités logicielles n'est donc pas obtenue sans efforts.

- En quoi ce pattern peut être rapproché d'autres patterns (lesquels et pourquoi ?
- Vous pouvez traiter le Visiteur comme une version plus puissante du patron de conception Commande. Ses objets peuvent lancer des traitements sur divers objets dans différentes classes.
- Vous pouvez utiliser le Visiteur pour lancer une opération sur un arbre Composite entier.
- Vous pouvez utiliser le Visiteur avec l'Itérateur pour parcourir une structure de données complexe et lancer un traitement sur ses éléments, même s'ils ont des classes différentes.
- Pattern décorateur: comme le pattern visiteur il permet d'ajouter des fonctionnalités sans altérer le code existant car chaque nouvelle fonctionnalité va être isolée dans une nouvelle classe afin d'appliquer le SRP et le OCP.
- indiquer des classes de la javadoc qui mettent en œuvre ce pattern

- javax.lang.model.element.AnnotationValue and AnnotationValueVisitor
- javax.lang.model.element.Element and ElementVisitor
- javax.lang.model.type.TypeMirror and TypeVisitor
- java.nio.file.FileVisitor and SimpleFileVisitor
- javax.faces.component.visit.VisitContext and VisitCallback

Ces éléments reprennent et mettent en œuvre le pattern visiteur, par exemple, ElementVisitor est une interface qui nous fournit la méthode “visit” utilisée plus tôt.
-> détailler rapidement les autres classes à l’oral.

- ## Utilisation du pattern dans le cadre de la programmation d’un jeu

On peut faire le lien entre le Visitor pattern et la programmation d’un jeu vidéo. Pour illustrer cela, imaginons que nous faisons un jeu dont le principe est de se combattre avec des personnages. Ces personnage ont des points de vie, un nom, un sexe, et une action, cette action c’est comabttre. mais chaque personnage à une façon de combattre différente, propre à chacun(attaquer à l’épé, attaquer avec une hache, attaquer avec ses griffes…). De plus, au départ les personnages se combattront seulement avec une attaque, mais on peut imaginer qu’après on puisse rajouter d'autres actions comme par exemple lancer un sort. Pour réussir à développer cela, le pattern visiteur peut nous aider, en isolant la méthode combattre à part, qui pourra ainsi avoir un comportement propre à chacun des personnages. Et, encore grâce au pattern, on pourra aussi rajouter le fait de pouvoir lancer un sort plus tard.

- ## Nouveau contexte, diagrammes et code.

Sujet :

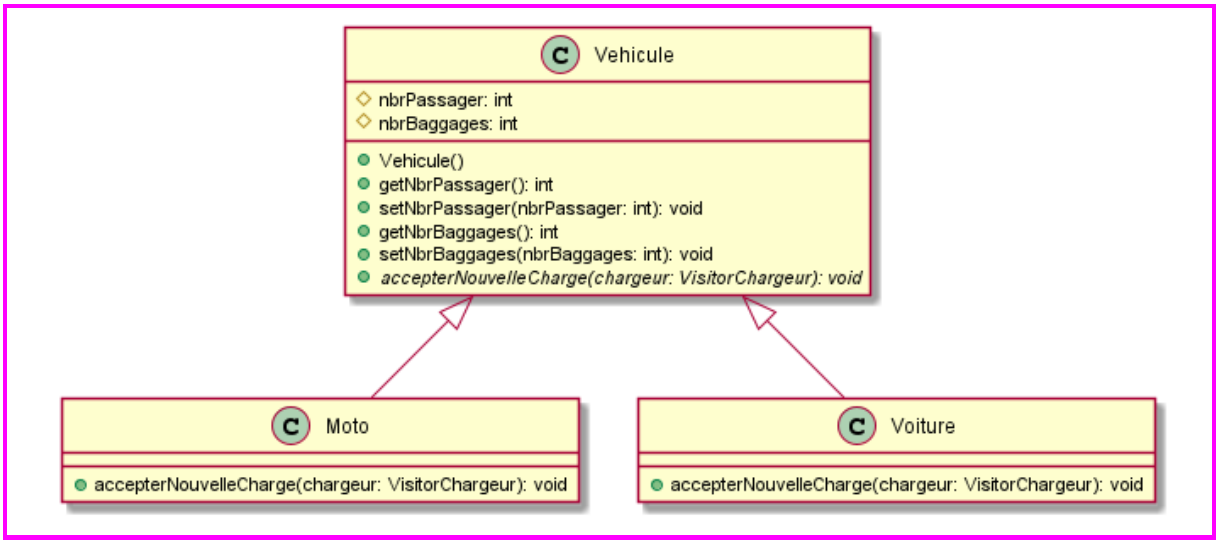
Depuis un ensemble d’objets de véhicules, nous devons pouvoir définir une certaine charge de bagages et un certain nombre de passagers par véhicule.

Problème :

Chaque type de véhicule (voiture, moto,...) possède un nombre maximum de passagers et une charge de bagages différents en fonction de son type.

Comment pouvons nous faire pour respecter les principes SOLID ?

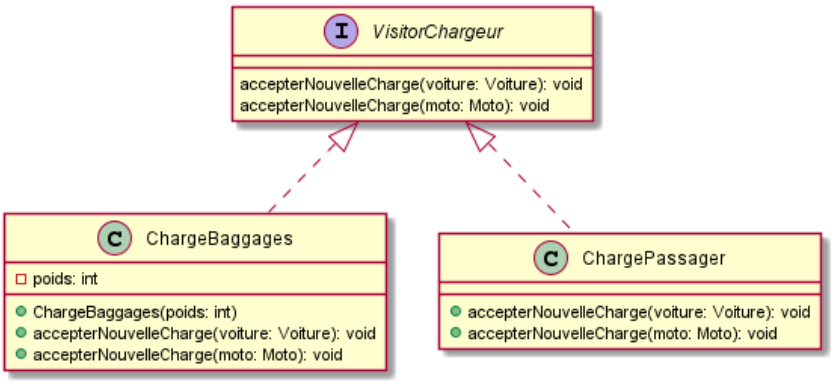
Passage par le pattern visiteur pour pouvoir vérifier si l’ajout de charge peut se passer.



```java
package patternVisiteur;

public abstract class Vehicule {

    protected int nbrPassager;
    protected int nbrBaggages;


    public Vehicule() {
        super();
        this.nbrPassager = 0;
        this.nbrBaggages = 0;
    }


    public int getNbrPassager() {
        return nbrPassager;
    }


    public void setNbrPassager(int nbrPassager) {
        this.nbrPassager = nbrPassager;
    }


    public int getNbrBaggages() {
        return nbrBaggages;
    }


    public void setNbrBaggages(int nbrBaggages) {
        this.nbrBaggages = nbrBaggages;
    }

    public abstract void accepterNouvelleCharge(VisitorChargeur chargeur);
}
```

```java
public class Moto extends Vehicule {
    @Override
    public void accepterNouvelleCharge(VisitorChargeur chargeur) {
        chargeur.accepterNouvelleCharge(this);
    }
}
```

```java
public class Voiture extends Vehicule {
    @Override
    public void accepterNouvelleCharge(VisitorChargeur chargeur) {
        chargeur.accepterNouvelleCharge(this);
    }
}
```




```java
package patternVisiteur;

public interface VisitorChargeur {

    public void accepterNouvelleCharge(Voiture voiture);
    public void accepterNouvelleCharge(Moto moto);

}
```

```java
package patternVisiteur;

public class ChargePassager implements VisitorChargeur {

    public void accepterNouvelleCharge(Voiture voiture) {
        if (voiture.getNbrPassager() < 4) {
            voiture.nbrPassager++;
        }
    }

    public void accepterNouvelleCharge(Moto moto) {
        if (moto.getNbrPassager() < 2) {
            moto.nbrPassager++;
        }
    }

}
```

```java
package patternVisiteur;

public class ChargeBaggages implements VisitorChargeur {

    private int poids;

    public ChargeBaggages(int poids) {
        super();
        this.poids = poids;
    }

    public void accepterNouvelleCharge(Voiture voiture) {
        if (poids < 50 && voiture.getNbrBaggages() < 200) {
            voiture.nbrBaggages += poids;
        }

    }

    public void accepterNouvelleCharge(Moto moto) {
        if (poids < 5 &&  moto.getNbrBaggages() < 15) {
            moto.nbrBaggages += poids;
        }
    }

}
```

- Live coding

Commencer par Création_Métier.mp4 :

Comme pour tout projet java, nous commençons donc par la création les classes métier qui comportent les véhicules définis comme ceci (passage de la vidéo ou la classe abstraite est faite) puis les héritages de cette classe, c'est-à-dire ici les classes voiture et moto. Ainsi, nous avons un contexte métier complet.

Ensuite la vidéo Création_Classes_Visitor.mp4 :

Maintenant que le code métier est fait nous pouvons voir ici (pointe la ligne du bas du code vehicule.java) qu'il est nécessaire d'avoir un ensemble de classes qui serviront à rajouter des bagages et des passagers dans les véhicules.

Ainsi, nous allons commencer par la création de l'interface visitorChargeur qui permettra de définir par la suite 2 classes distinctes afin de respecter le principe de Singleton (du truc SOLID) qui indique qu'une classe doit avoir un seul rôle spécifique.

En conséquence, nous allons créer 1 classe gérant l'augmentation du nombre de passagers puis une classe allant gérer l'augmentation de la charge des véhicules.

Et enfin nous allons tout tester dans une classe SandBox pour s'assurer du bon comportement de notre métier.

bibliographie/webographie :

https://www.javabrahman.com/design-patterns/gof-gang-four-design-patterns/
https://howtodoinjava.com/gang-of-four-java-design-patterns/
https://www.youtube.com/watch?v=UQP5XqMqtqQ
https://cdiese.fr/design-pattern-visiteur/#:~:text=Certains%20probl%C3%A8mes%20se%20posent%3A,4%20passagers%20%C3%A0%20une%20voiture.
https://ryax.tech/fr/design-pattern-cest-quoi-et-pourquoi-lutiliser/
http://www.blackwasp.co.uk/Visitor.aspx
https://kandran.fr/design-pattern-visiteur/
https://tech.gojob.com/design-patterns-typescript-1/
https://refactoring.guru/fr/design-patterns/visitor
https://github.com/geekific-official/geekific-youtube/tree/main/pattern-behavioral-visitor/src/main/java/com/youtube/geekific

oeuvre littéraire : Rule-Based Sequence Diagram Generation with Application of the Visitor Pattern