DESIGN PATTERNS

[GoF] Comportementaux

- **Quoi ?**
  Communication entre objets

- **Comment ?**
  Répartition des responsabilités entre objets
  encapsulation des comportements
  délégation des requêtes

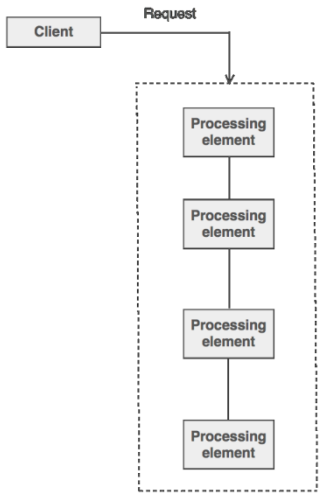
- **Bénéfice ?**
  Meilleure flexibilité (découplage)

[GoF] Chaine de Responsabilité

(chain of responsability)

- **Objectif : émettre une demande pouvant être traitée potentiellement par plusieurs objets (liste)**

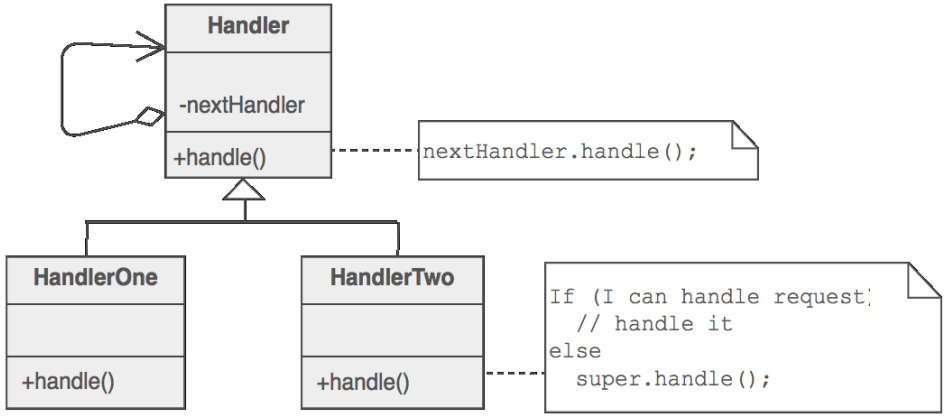
- **Problème**
  l'émetteur ne connait ni le nombre d'objets ni lequel traitera effectivement la demande



- **Solution**
  structure de pipeline

- **Exemple**
  Distributeur de billets
  Crible d'Eratosthène

[GoF] Chaine de Responsabilité
(chain of responsability)



- **Handler**
  définit la chaîne d'objet (référence nextHandler)
  et le transfère de la demande à celui-ci
- **HandlerOne, HandlerTwo,…**
  spécialisent la classe Handler (méthode handle)
  si l'objet ne peut pas traiter la demande, il la relaye à la classe de base
  qui la transfère à l'objet suivant
- **Cas du dernier Handler (null next)**

[GoF] Chaine de Responsabilité
(chain of responsability)

- **Découple émetteur et récepteur**
  comme Commande, Médiateur et Observateur
- **Demande passée à une chaine de récepteurs potentiels**
- **Objets Commandes pour représenter la requête**
- **Associé au DP Composite**
  nœud père = successeur

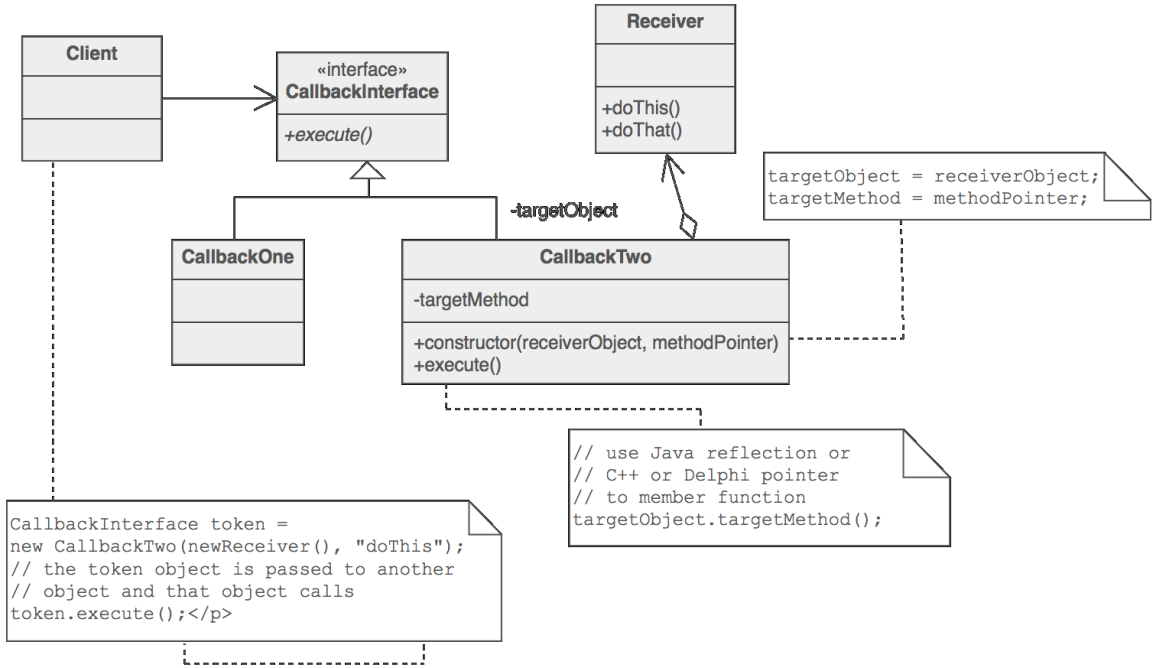
[GoF] Commande

- **Objectif :**
  - encapsuler la requête dans un objet (différentes requêtes, liste de requêtes, …)
  - appel de fonction (callback) orienté objet

- **Problème**
  émettre une requête sans connaitre ni la nature de la requête ni l'objet la traitant

- **Solution**
  classe abstraite
  constructeur : objet + action
  méthode execute() : action sur l'objet

[GoF] Commande



- **Client**
  utilise la commande (≠ client créant la commande)
- **CallbackInterface**
  interface proposant la méthode execute()
- **CallbackOne, CallbackTwo, …**
  implémente la méthode execute à partir d'un objet receveur et d'une méthode de cet objet)

[GoF] Commande, exemple

- **Classe Livre**
  titre, auteur, prix, cadeau(O/N)
  constructeur
  méthode *Acheter() // cadeau=false;*
  Offrir () // cadeau = true;
  Afficher () // titre, auteur, prix

- **Interface LivreCommande**
  méthode execute()

- **Classe LivreCommandeAcheter**
  constructeur (unLivre)
  execute() // unLivre.Acheter()

- **Classe LivreCommandeOffrir**
  constructeur (unLivre)
  execute() // unLivre.Offre()

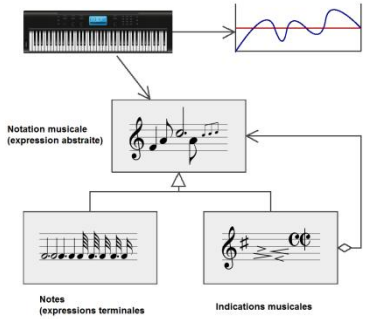
- **Classe Client**
  instantie un livre,
  instantie un LivreCommandeOffrir, appelle execute
  instantie un LivreCommandeAcheter, appelle execute

[GoF] Interprète (interpreter)

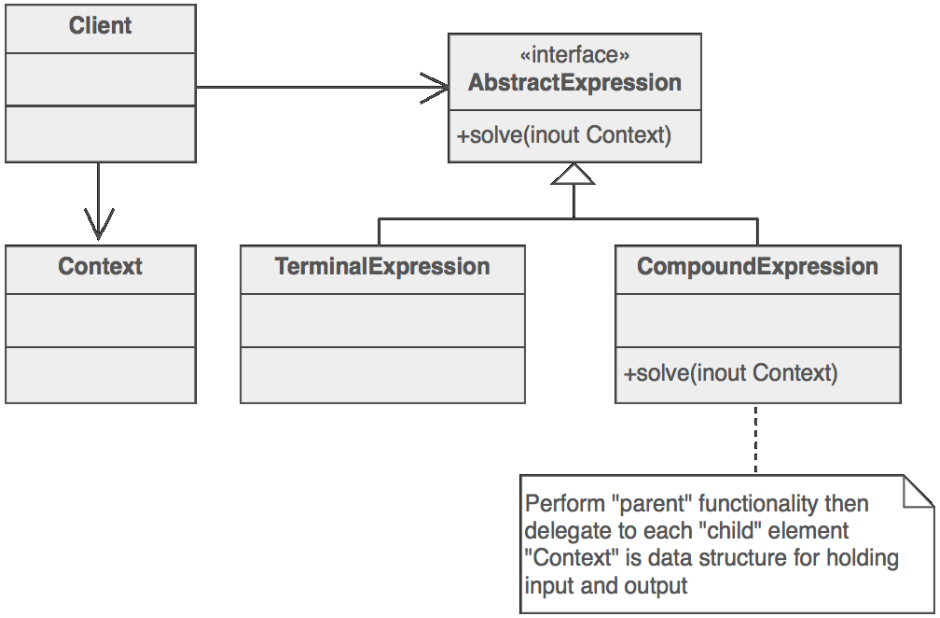
- **Objectif : interprétation de phrases d'un langage à partir de sa grammaire (analyse lexicale)**

- **Problème**
  grammaire bien définie analysable par un moteur

- **Solution**
  class abstraite – méthode interprete()
  sous-classes implémentant interprete()
  fonction de l'état courant et du texte lu
  construction d'un arbre
- **Exemple**
  lecture d'une partition musicale
  *lecture (x – 2) * (x + 5) → expression*



[GoF] Interprète (interpreter)



- **Modélisation du domaine par une grammaire récursive**
  cf DP Composite

[GoF] Interprète (interpreter)

- **Composite implique Interprète**
  cf exercice Expressions mathématiques
- **Utilisation conjointe du DP Etat**
  pour représenter le contexte
- **L'arbre produit par l'analyse est un Composite**
  Itérateur et Visiteur utilisable également
- **Si la grammaire est complexe, d'autres outils à considérer**
  analyseur syntaxique

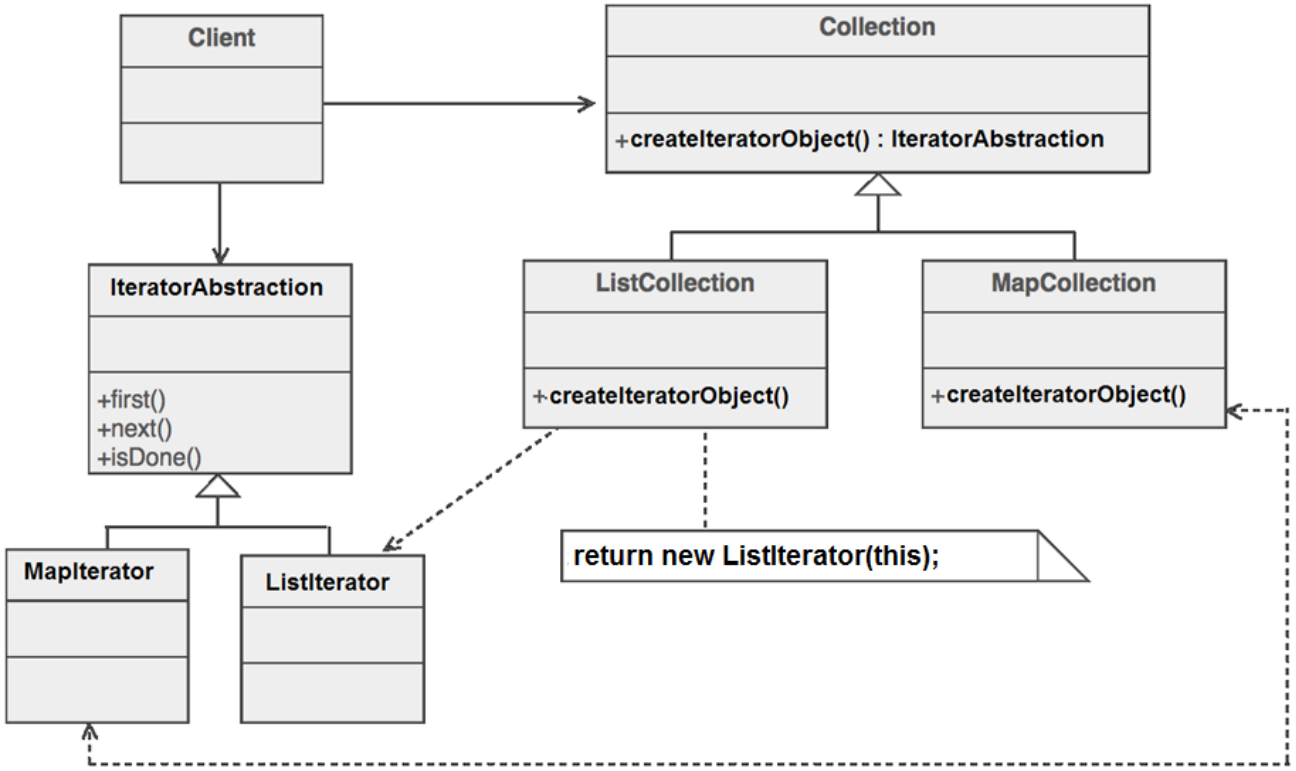
[GoF] Iterateur (iterator)

- **Objectif :**
  - parcours d'un object composé (conteneur) en respectant l'encapsulation
  - possibilité de différents parcours (polymorphisme)

- **Problème**
  description abstraite du parcours (adaptable à plusieurs structures de données)

- **Solution**
  - ajout d'une méthode createIterator dans la classe composée
  - création d'une classe iterator
  first() / next() / isDone() / currentItem()
  - dans le client, appel de la createIterator() et parcours

[GoF] Iterateur (iterator)



[GoF] Iterateur (iterator), exemple

```
abstract class Aggregate {
  public abstract Iterator CreateIterator();
 }

 class ConcreteAggregate : Aggregate
 {
  private ArrayList items = new ArrayList();

  public override Iterator CreateIterator()                    {     return new
        ConcreteIterator(this);    }

  public int Count                          { get{ return items.Count; } }

  public object this[int index] {              get{ return items[index]; }
                                               set{ items.Insert(index, value); }
  }
 }

//-------------------------------------------------------------------------------------------------------
// Main
        ConcreteAggregate a = new ConcreteAggregate();
        a[0] = "Item A";
        a[1] = "Item B";
        a[2] = "Item C";
        a[3] = "Item D";

        ConcreteIterator i = new ConcreteIterator(a);
        Console.WriteLine("Iterating over collection:");
        object item = i.First();

        while (item != null)      {
                    Console.WriteLine(item);
                    item = i.Next();
        }
```

```
abstract class Iterator{
  public abstract object First();
  public abstract object Next();
  public abstract bool IsDone();
  public abstract object CurrentItem();
}

class ConcreteIterator : Iterator{
  private ConcreteAggregate aggregate;
  private int current = 0;

  public ConcreteIterator(ConcreteAggregate aggregate)
        { this.aggregate = aggregate; }

  public override object First()
        { return aggregate[0]; }

  public override object Next() {
    object ret = null;
    if (current < aggregate.Count - 1)
      ret = aggregate[++current];

    return ret;
  }

  public override object CurrentItem()
        { return aggregate[current];}

  public override bool IsDone()
        {              return (current >= aggregate.Count);}
}
```

[GoF] Memento

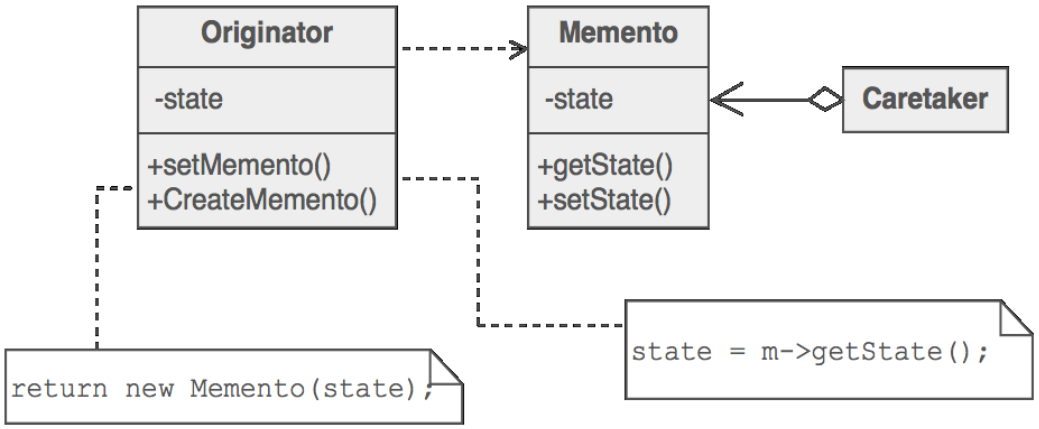
- **Objectifs**
  - capture et externalise l'etat d'un objet (point de sauvegarde)
  - fonctionnalités de undo/rollback

- **Problème**
  restauration d'un objet à un état antérieur

- **Solution**
  - classe Memento mémorisant l'état (get/set)
  - client : décide de la sauvegarde/restauration
  n'accède pas directement à l'état
  - seul l'objet peut se sauvegarder/restaurer
  - plusieurs undo (pile de mémento)

- **Exemple**

[GoF] Memento



- **Originator**
  l'objet à sauvegarder/restaurer (lui seul sait le faire)
  méthodes CreateMemento (sauvegarde) / SetMemento (restauration)
- **Caretaker**
  l'objet décidant quand et pourquoi sauvegarder/restaurer l'objet Originator
- **Memento**
  l'objet stockant l'état (getState / setState)

[GoF] Memento

- **Memento vs Command**
  Commande stocke la méthode exécutée
  Memento stocke l'état

- **Memento et Command**
  dans le cas d'une commande undo

- **Memento et Iterateur**
  stocke l'état après une itération

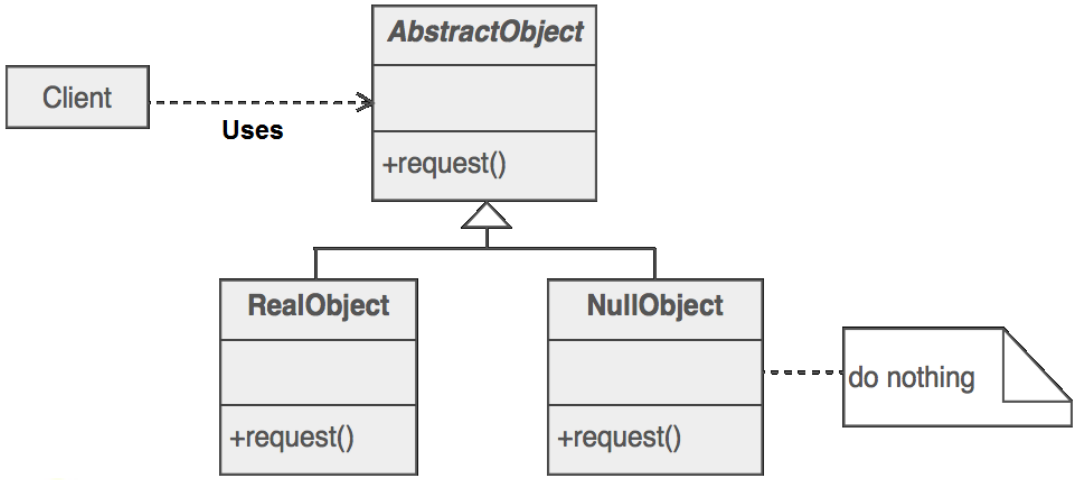
[GoF] Memento, exemple

- **Classe PetitPoucet *(Originator)***
  données X, Y
  Constructeur
  Deplacer (dX, dY)
  PoserCaillou / RetourAuCaillou
  OuSuisJe ()
- **Classe Caillou *(Memento)***
  stocke X et Y (propriété get)
- **Classe AngeGardien *(CareTaker)***
  propriété Caillou (get/set)
- **Main**
  instancie lePetitPoucet (0,0), lAngeGardien
  déplace lePetitPoucet, pose un caillou, déplace à nouveau
  retourne au caillou
- ***Quid de plusieurs cailloux ?***

[GoF] Objet NULL

- **Objectif : offrir un objet remplaçant l'absence d'objet *(null)***
- **Problème**
  appel de méthode sur un référence null ??
- **Solution**
  sous classe NullObject de la classe de base substituable aux objets « non nuls » (mêmes méthodes)

[GoF] Objet NULL



[GoF] Objet NULL, exemple

```
class NullOutputStream extends OutputStream {
    public void println(int b) {
        // rien…
    }
}

class NullPrintStream extends PrintStream {
    public NullPrintStream() {        super(new NullOutputStream());    }
}

class Application {
    private PrintStream sortieDebug;
    public Application(PrintStream sortieDebug) {        this. sortieDebug = sortieDebug; }

    public void  Calculer() {
        int somme = 0;
        for (int i = 0; i < 10; i++) {
            somme += i;
          sortieDebug.println("i = " + i);
        }
        System.out.println("somme = " + somme);
    }
}
```

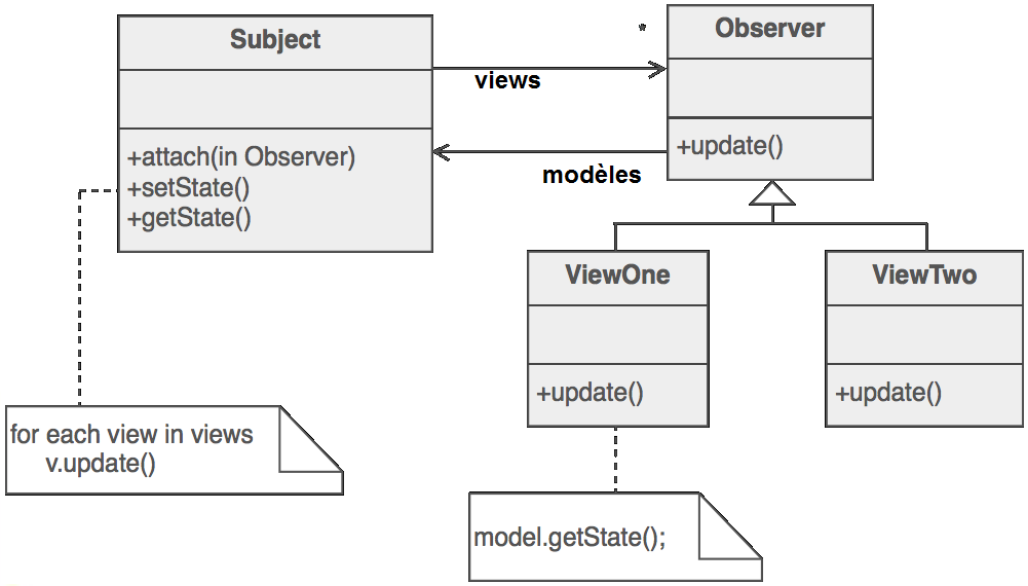
[GoF] Observateur (observer)

- **Objectif**
  mise en place d'une relation 1-N (objet observé, objets observateurs) permettant de notifier automatiquement les changement de l'objet observé *(partie Vue du modèle MVC)*

- **Problème**
  éviter une conception monolithique difficile à maintenir

- **Solution**
  - Classe **Sujet** : responsable des données *(logique métier)*
  - Classes **Observateurs** : responsables des représentations *(vues)*
  - à chaque changement d'état
    - notification des observateurs
    - mise à jour par ceux-ci des représentations gérées
  - Protocoles entre Sujet et Observateurs
    - pull : les observateurs récupèrent les données qui les intéressent
    - push : le sujet envoie aux observateurs ce qui a changé

[GoF] Observateur (observer)



[GoF] Observateur (observer)

- **Définir la classe abstraite Sujet**
  logique métier
  + attach() / detach() / notify()
- **Définir les classes Observateurs**
  encapsule le sujet
  le sujet n'est lié qu'à la classe de base (collection d'objets)
  update()
- **Le client définit le nombre et les types d'observateurs**
  s'enregistrent auprès de l'objet sujet
- **L'objet sujet notifie les observateurs**
  changements d'états
- **Information demandée**
  envoyée par le sujet (push) ou demandée par l'observateur (pull)

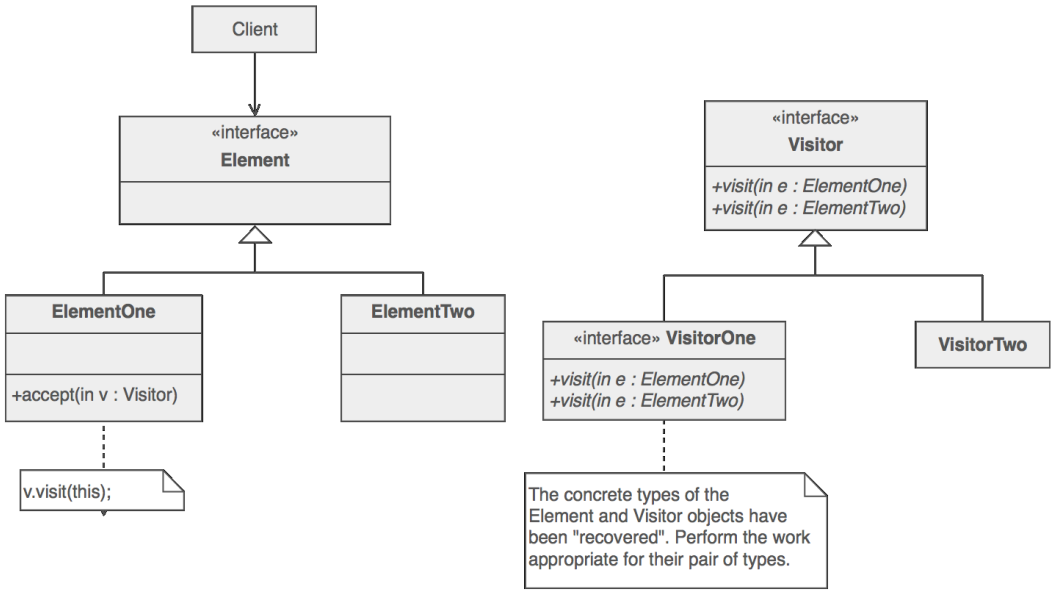
[GoF] Visiteur (visitor)

- **Objectif : opérations à effectuer sur les éléments d'un objet composé sans changer la structure des classes des éléments en question**

- **Problème**
  ne pas surcharger les classes nœud par ces opérations

- **Solution**
  extraire des classes des elements la logique de ces operations et l'attribuer à une classe (hierarchie) Visitor

[GoF] Visiteur (visitor)



[GoF] Visiteur (visitor)

- **Créer une classe de base Visitor**
  pour chaque type d'élement visité :
  méthode visit(Element E)
- **Ajouter une méthode accept(Visitor v)**
  implémentation unique :
  v.visit (this);
- **Créer une sous-classe Visitor pour chaque opération à effectuer**
- **La classe Client crée les objets Visitor et les passe aux objets Elements**
  accept()

[GoF] Résumé (1)

- **Chaîne de responsabilité**
  permet de passer une requête au travers d'une liste d'objets
- **Commande**
  encapsule une commande en tant qu'objet
- **Interprète**
  intègre des éléments de langage
- **Itérateur**
  accède séquentiellement aux éléments d'une collection

[GoF] Résumé (2)

- **Memento**
  Conserve l'état d'un objet (sans violer l'encapsulation) permettant sa restauration (undo)
- **Objet Null**
  fonctionne comme une valeur par défaut pour un objet
- **Observateur**
  Notifie des changements à un ensemble de classes
- **Visiteur**
  définit une nouvelle operation sur une classe sans la changer