Les modèles de comportements

- Ce rôle désigne tous les patterns dont le rôle est de spécifier :
  - la façon dont des classes ou des objets interagissent,
  - la répartition des responsabilités de différents objets.

Les modèles comportementaux

- Dans cette famille, nous étudierons les patterns suivants :

- **Chaîne de responsabilités**
- **Commande**
- **Observateur**
- **Modèle Vue Contrôleur**
- **Etat**
- **Visiteur**
- **Médiateur**
- **Itérateur**
- **Interpréteur**

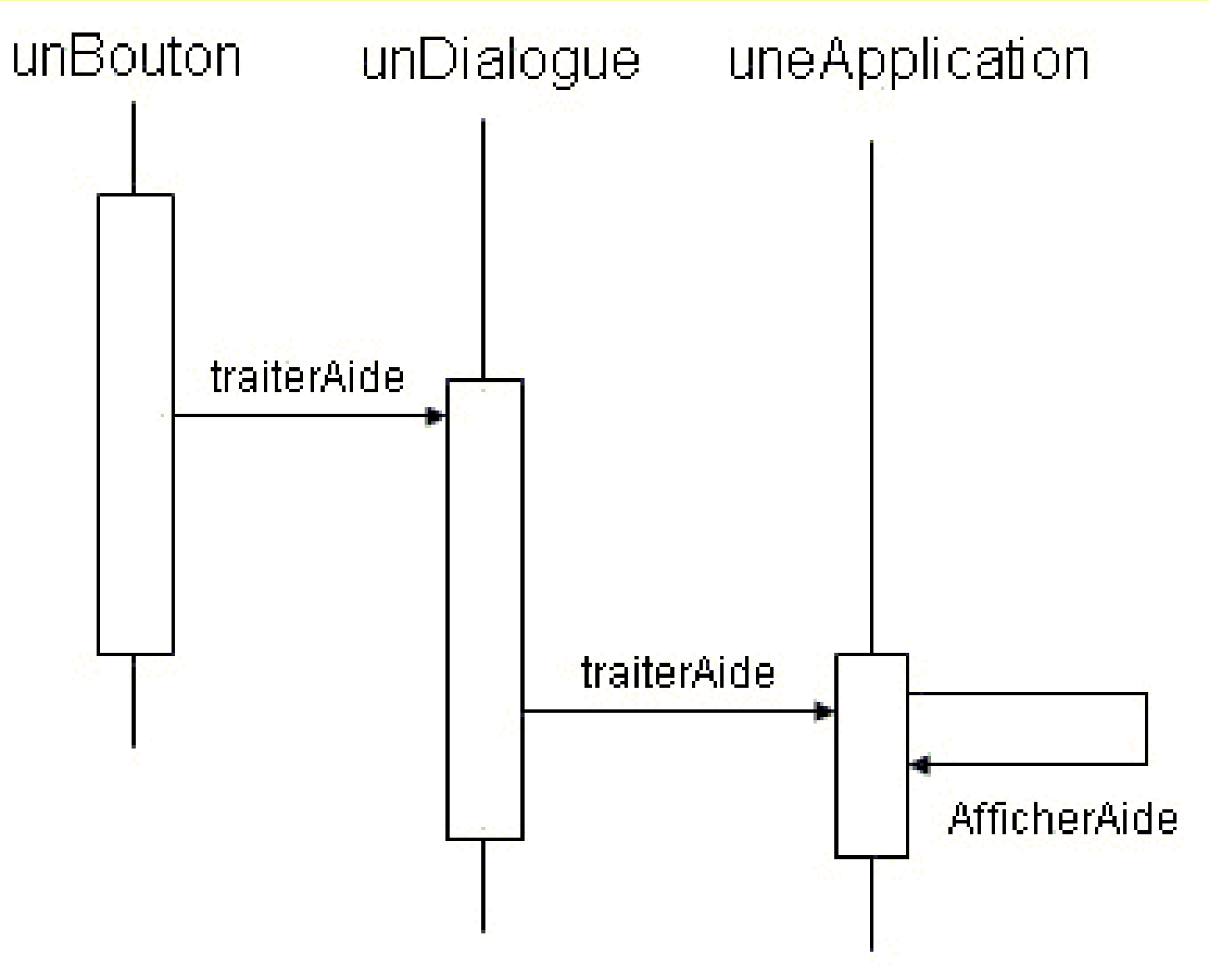
Le pattern Chaîne de responsabilités

- **Motivation**
  - Dans une IHM, on veut gérer une aide contextuelle.
  - Quand l'utilisateur clique dans un composant avec le bouton droit, il désire une aide : de deux choses l'une
    - Une aide est prévue sur le composant, et on l'affiche
    - On demande au composant immédiatement englobant de traiter la requête

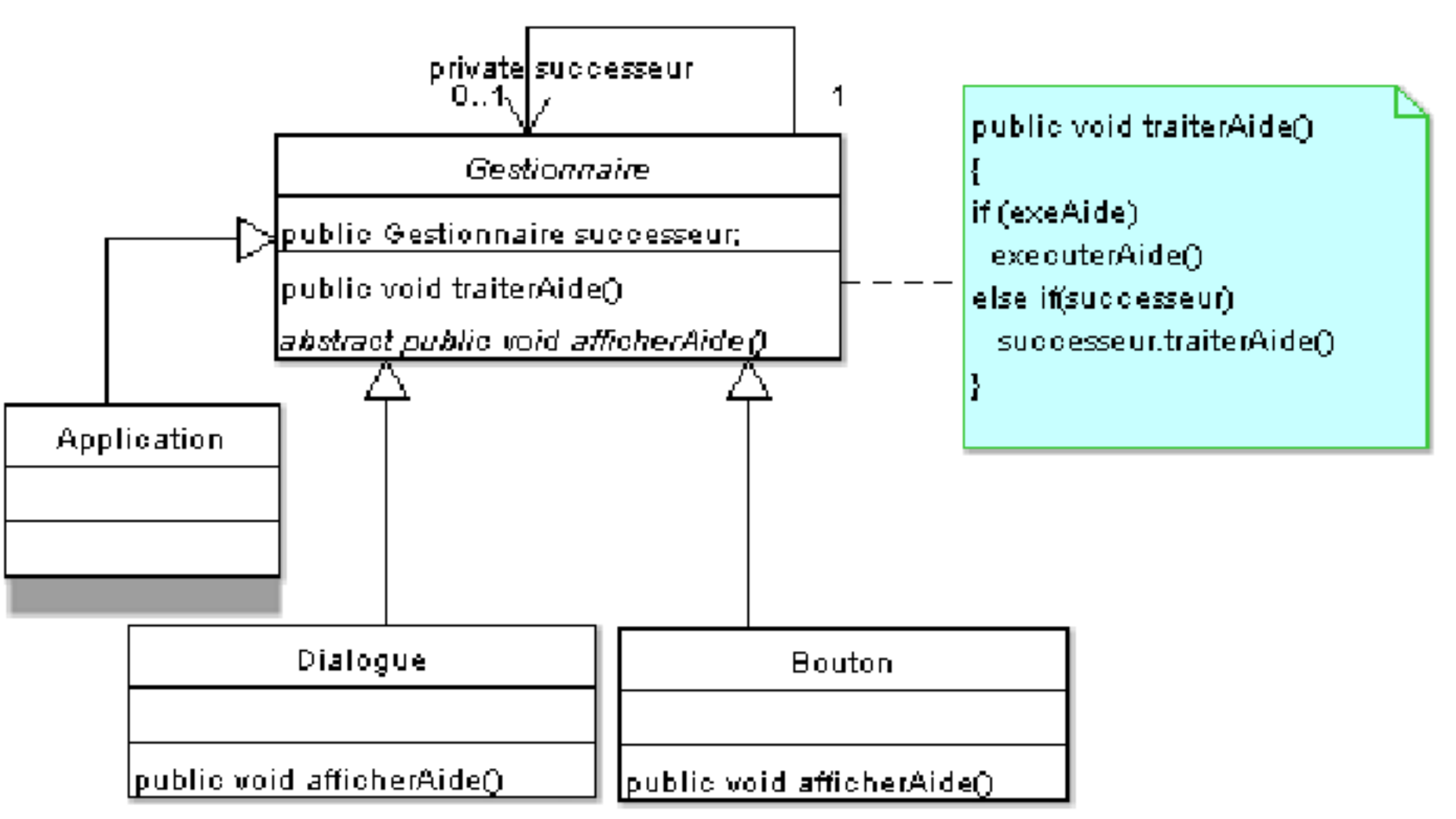
Le pattern Chaîne de responsabilités

- **Question ?**
  - Faire le diagramme de séquences relatif au scénario :
    « l'utilisateur a cliqué sur un widget bouton marqué "imprimé" pour demander une aide »
  - On suppose que ce bouton est dans une boite de dialogue elle même dans la fenêtre d'application et que l'aide est concentrée dans cette fenêtre.

Le pattern Chaîne de responsabilités



Le pattern Chaîne de responsabilités

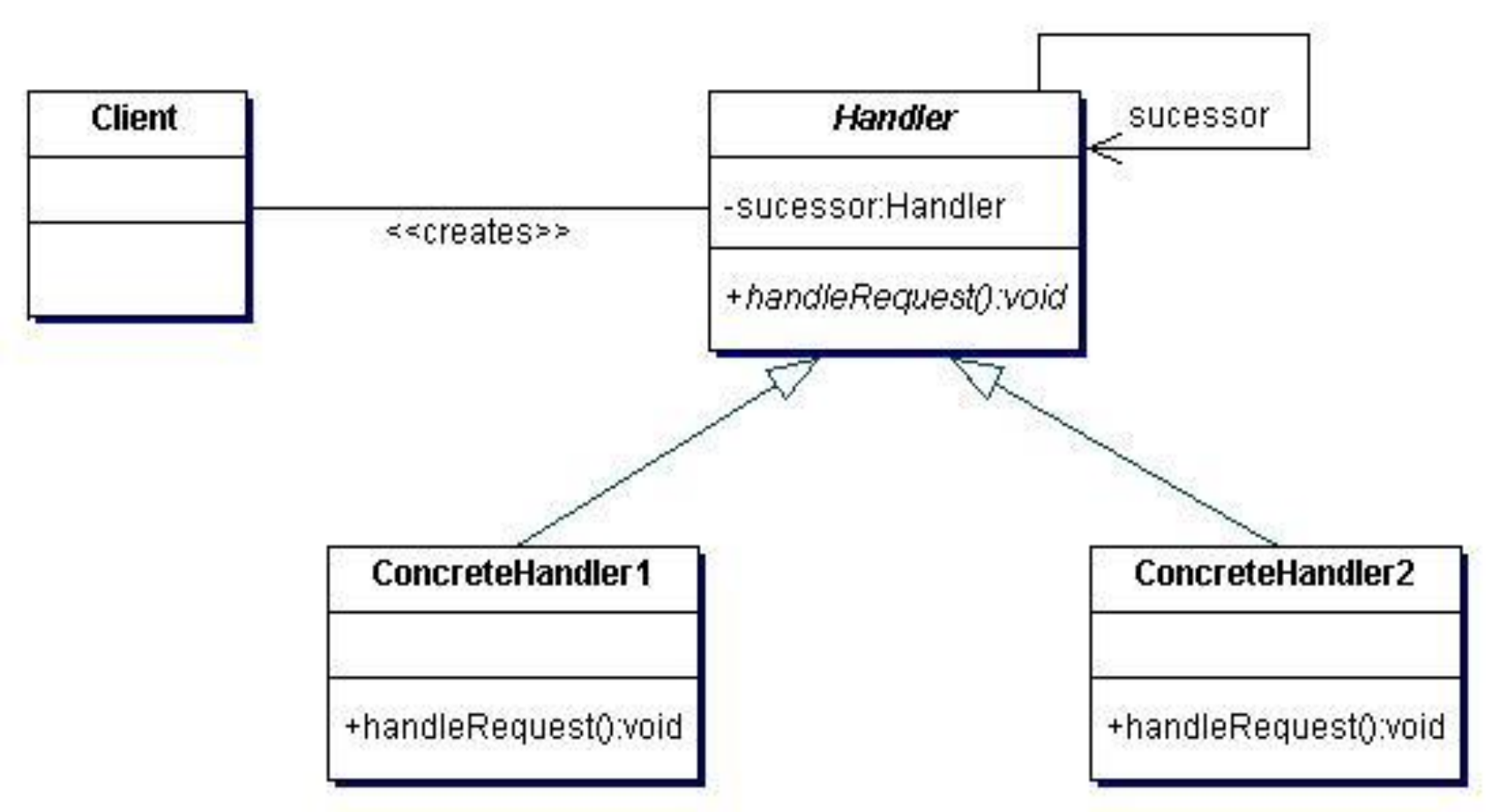


Le pattern Chaîne de responsabilités

- **Intention**
  - Ce pattern permet d'éviter le couplage entre l'émetteur d'une requête et les récepteurs.
  - L'émétteur ne doit pas connaître le premier récepteur et plus largement toute la chaîne de récepteurs qui traite la requête.

Le pattern Chaîne de responsabilités

- **Structure**



Le pattern Chaîne de responsabilités

- **Constituants**
  - **Une interface nommée Handler qui définit les prototypages des requêtes du client**
  - **Des classes gestionnaires concrètes (composants graphiques dans notre exemple) qui implémentent Handler**
  - **Une classe Client qui propose la requête à un objet gestionnaire du début de la chaîne**

Le pattern Chaîne de responsabilités

- **Implémentation**

```
public void handleRequest() {
   // Je fais ce que je sais faire
   ....
   // Si j'ai un successeur, il traite aussi
   if (successor != null)
      successor.handleRequest();
}
```

Le pattern Commande

- **Motivation**
  - On considère une application qui permet de faire des dessins.
  - L'item de menu "nouveau" permet de créer et d'ajouter à l'application un nouveau dessin
  - L'item de menu "coller" permet d'insérer dans un dessin le contenu du presse-papier

Le pattern Commande

- On remarque que les deux objets "Item" ont à transmettre une requête à des objets totalement différents :
  - Un objet Application pour le premier
  - Un objet Dessin pour le second.

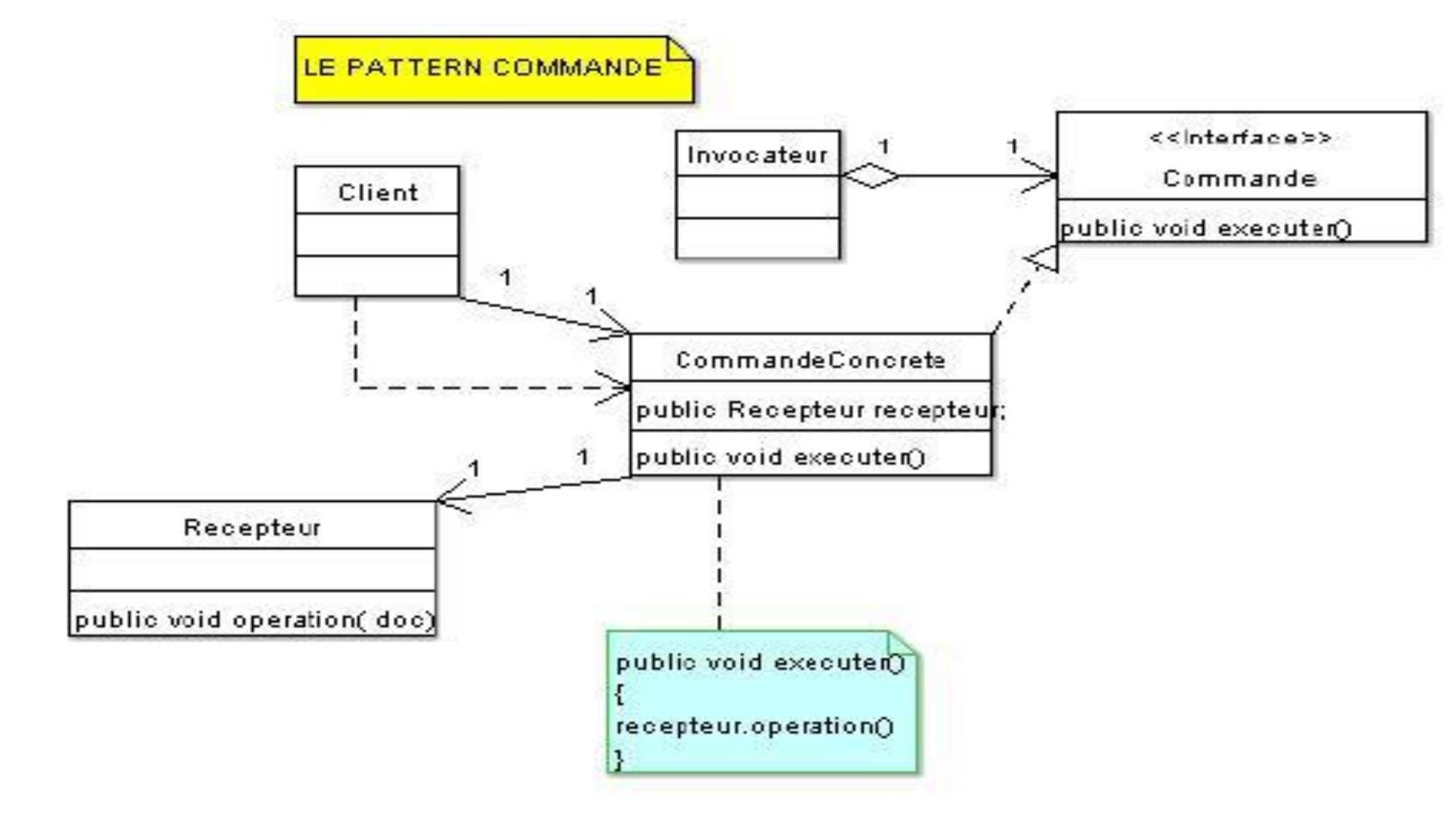
- Pourtant ces deux objets ont en commun d'être des items de menu donc issus de la même classe.

Le pattern Commande

- Intention
  - d'encapsuler une requête comme un objet pour découpler complétement l'objet qui émet la requête de celui qui va la traiter. L’objet emetteur n’aucune information sur la nature du traitement de la requête.
  - Le traitement de la requête n'est plus exécutée directement par l'objet concerné.

Le pattern Commande

- **Structure**



Le pattern Commande

- **Constituants**
  - **Une interface Commande pour exécuter une opération**
  - **Une ou plusieurs classes commandes concrètes (CmdColler, CmdNouveau) qui implémentent Commande.**
  - **Une ou plusieurs classes Invocateur (les items de menus)**

Le pattern Commande

- **Constituants**
  - **Des classes Récepteur (Application, Document) dont les objets sont impactés par l'exécution de la commande. Ses objets exécutent la commande.**

Le pattern Commande

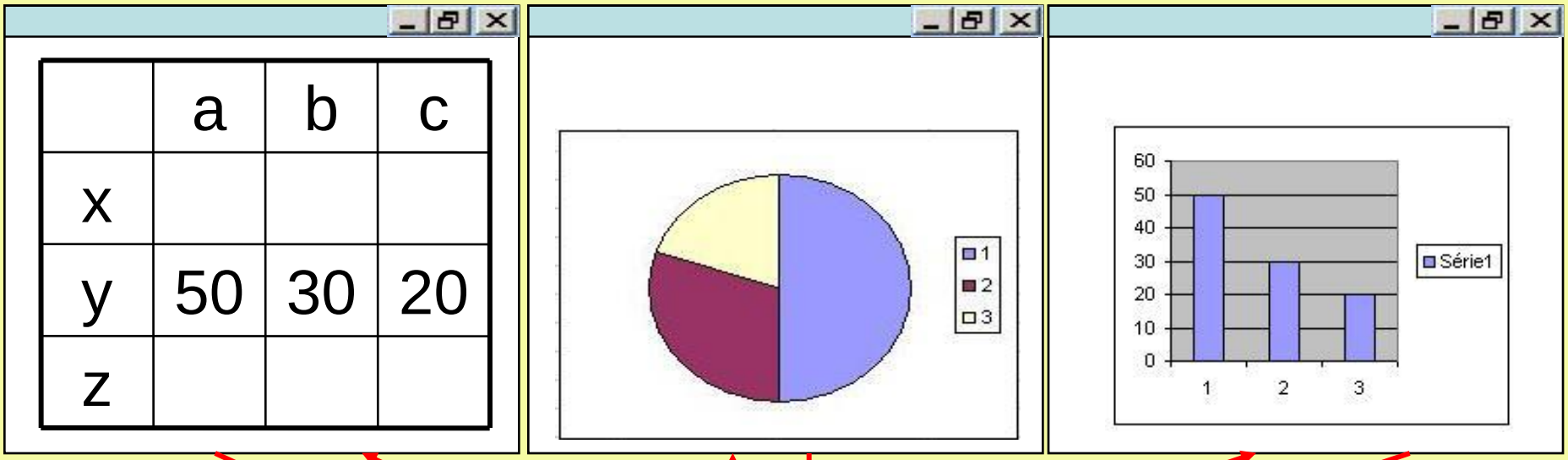
- **Constituants**
  - **Une classe Client (Application) qui crée la commande concrète, le récepteur et le positionnement du récepteur dans la commande concrète.**

Le pattern Observateur (1)

- **Motivation**
  - Dans de nombreuses IHM, les données peuvent être présentées à l'utilisateur de plusieurs façons.
  - Comment rendre les données et leurs représentations indépendantes.

Le pattern Observateur (2)

Les observateurs (présentation des données)



a = 50%
b = 30%
c = 20%

Le sujet (données)

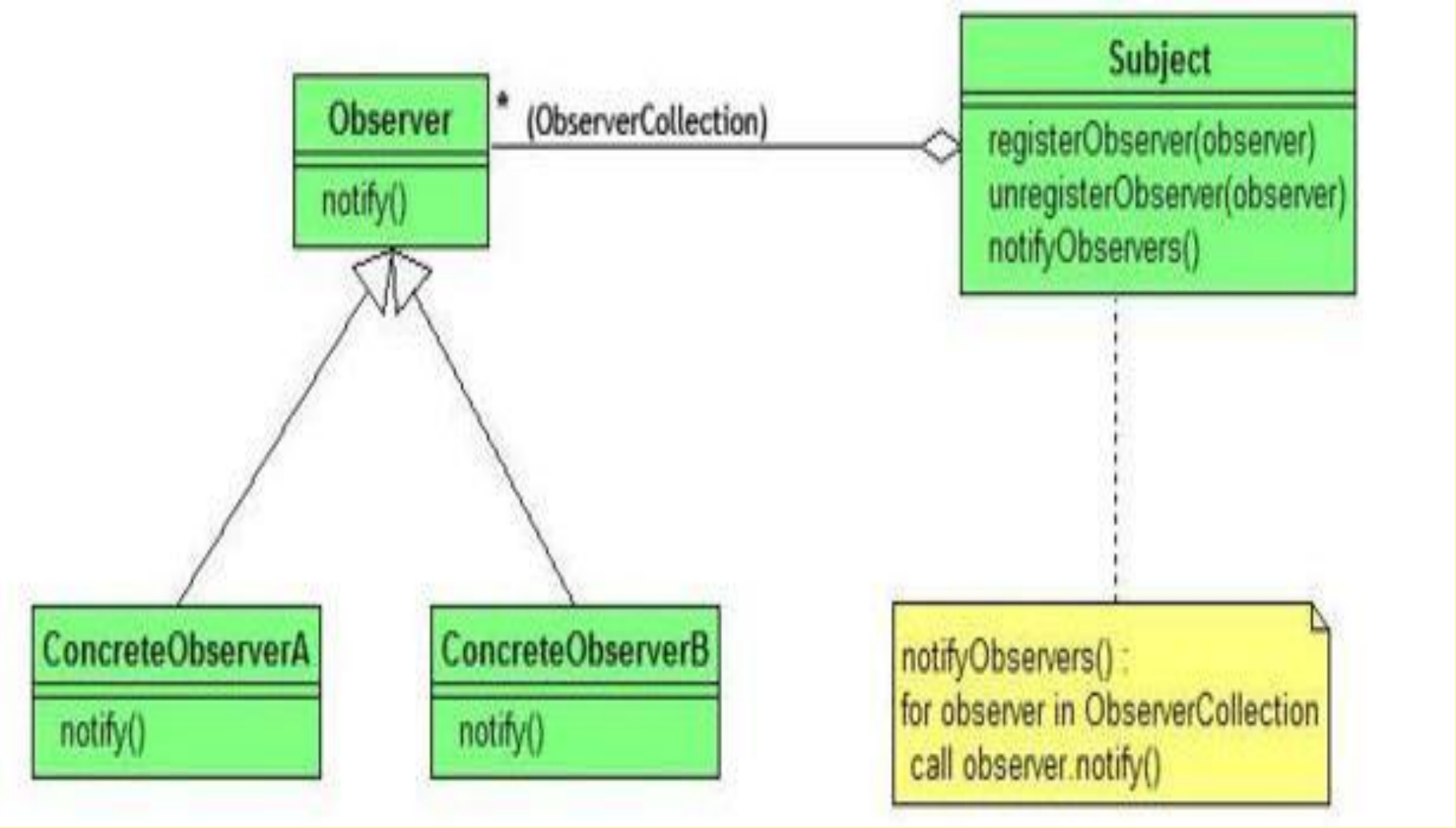
notification de changement

requêtes, modifications

Le pattern Observateur (3)

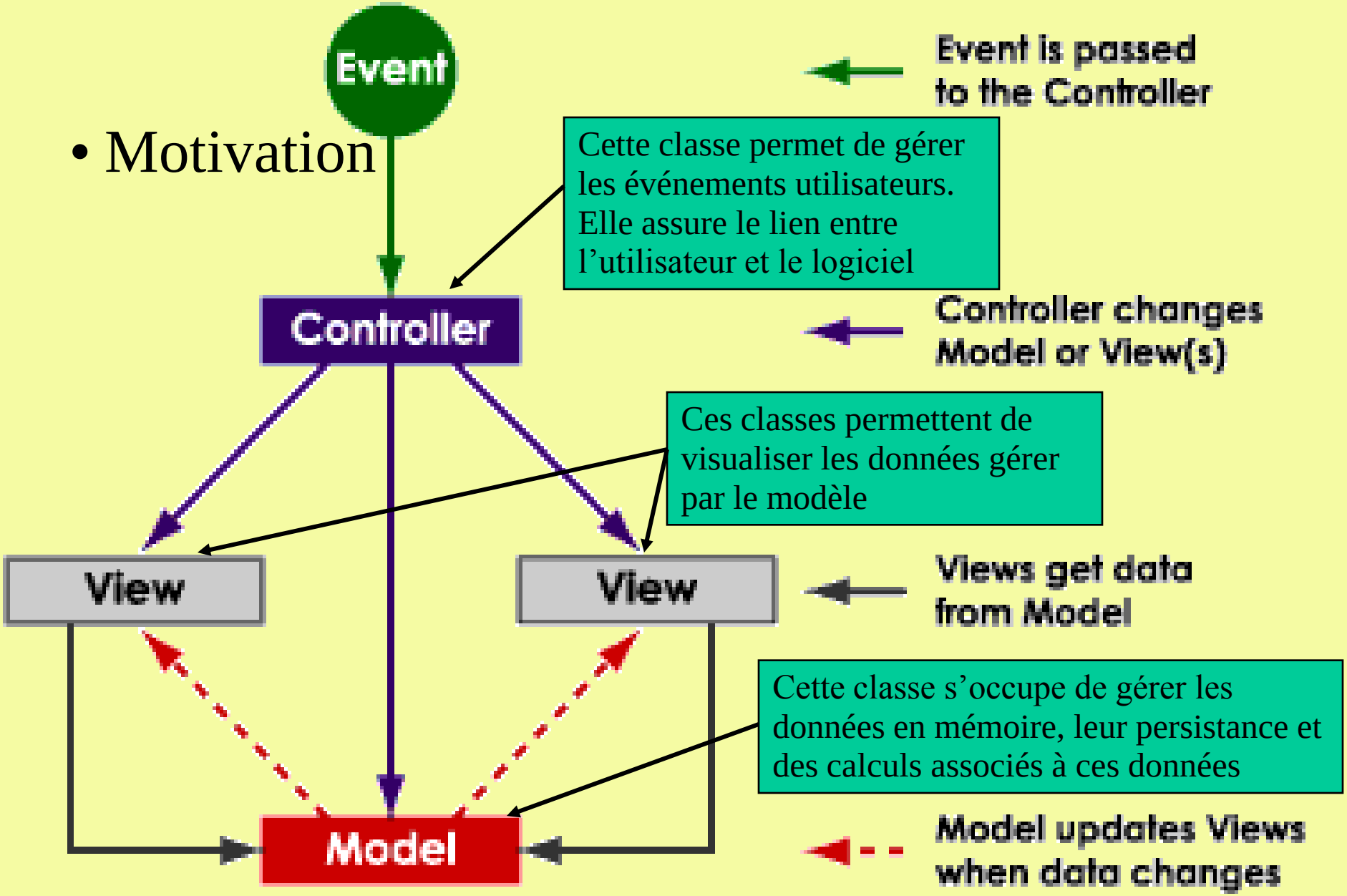
- **Intention**
  - Créer des dépendances entre un objet X et ses objets liés de telle façon que dès que l'objet X change d'état tous les objets liés en soient notifiés automatiquement.

La structure du pattern Observateur



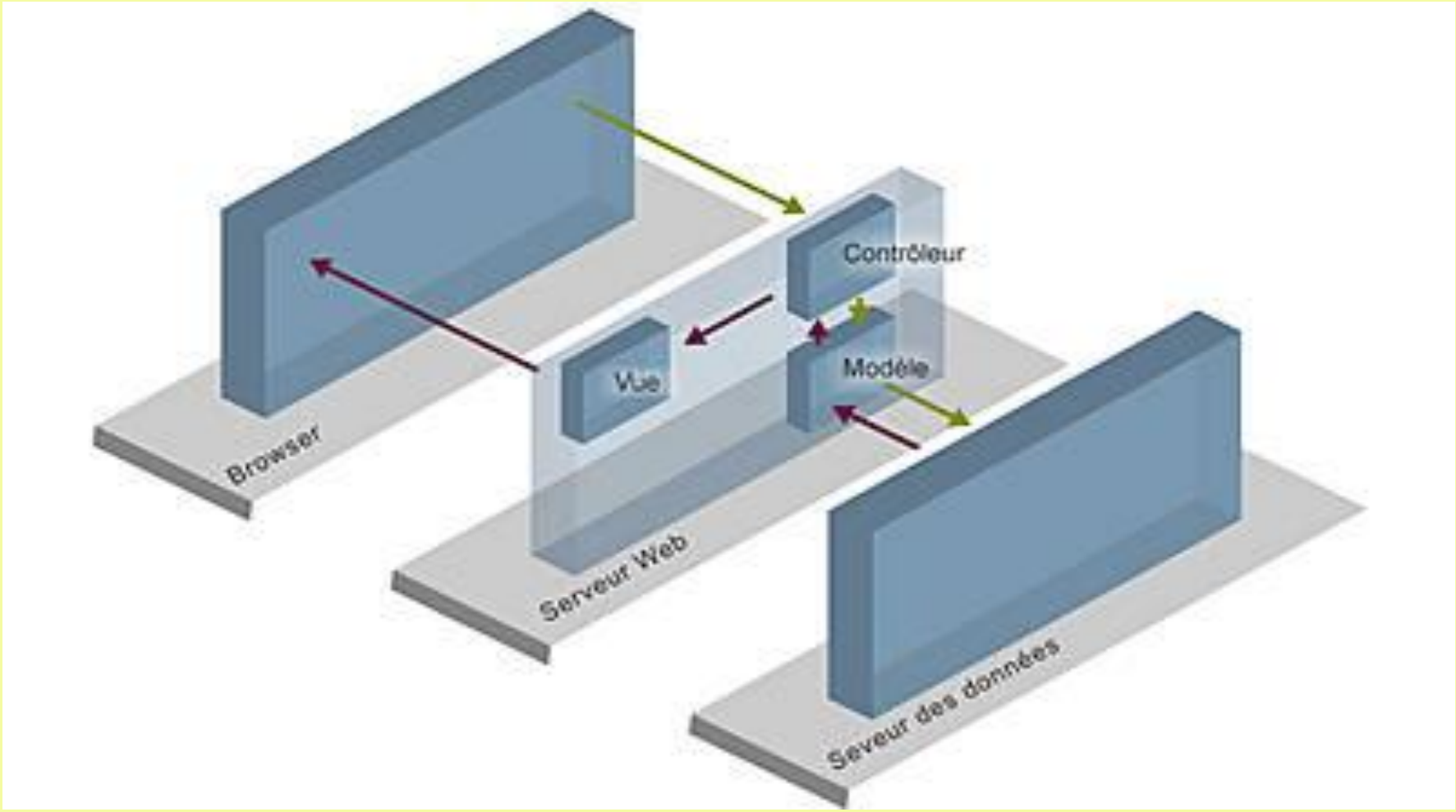
Ce schéma a été obtenu à l'adresse http://en.wikipedia.org/wiki/Observer_pattern

- Motivation


Event
Event is passed to the Controller
Cette classe permet de gérer les événements utilisateurs. Elle assure le lien entre l'utilisateur et le logiciel
Controller
Controller changes Model or View(s)
Ces classes permettent de visualiser les données gérer par le modèle
View
View
Views get data from Model
Cette classe s'occupe de gérer les données en mémoire, leur persistance et des calculs associés à ces données
Model
Model updates Views when data changes

Le pattern MVC

- Motivation dans le cadre d'Internet



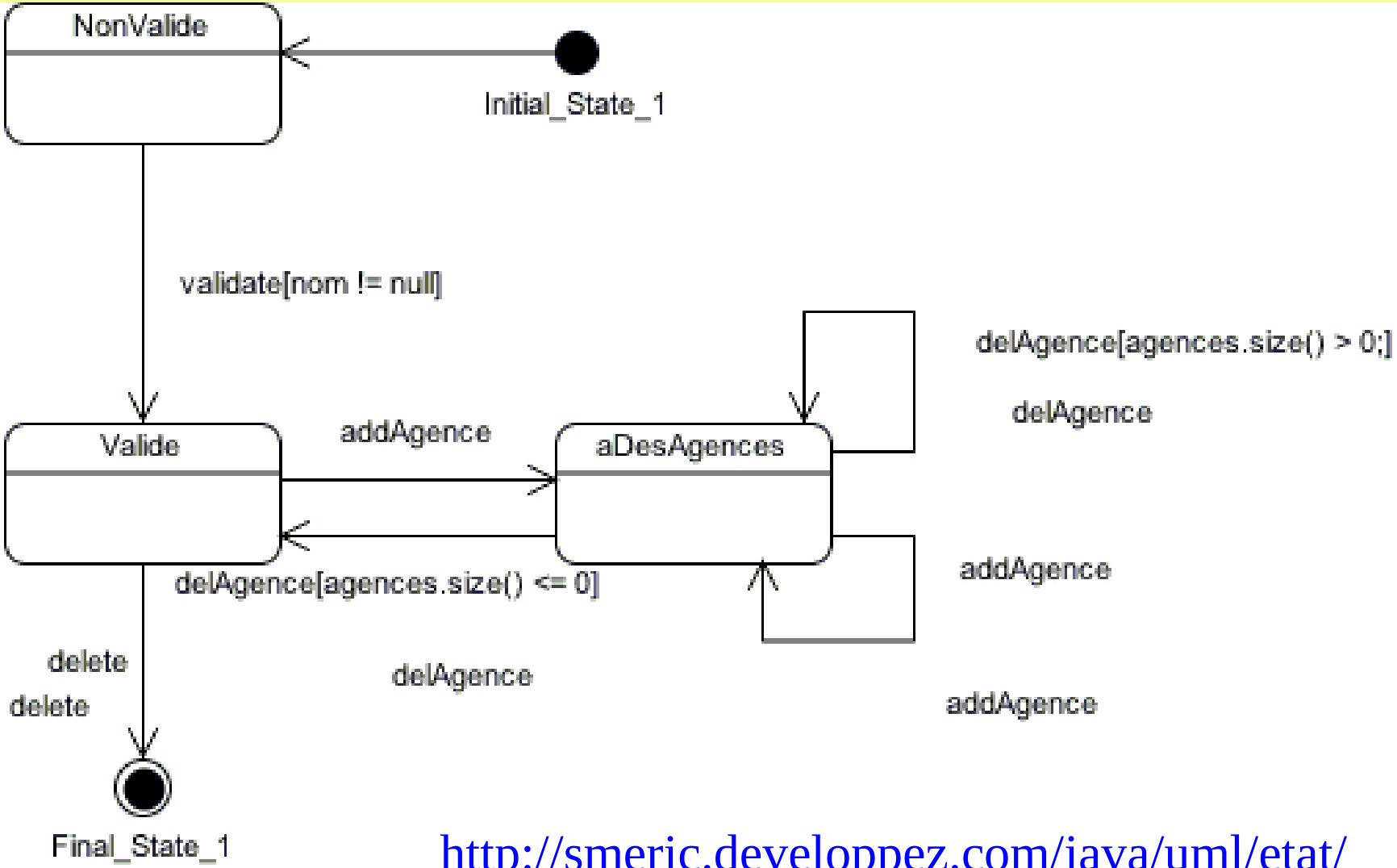
Le pattern Etat

- **Motivation**
  - On gére des banques et leurs agences associées.
  - Un objet Banque doit :
    - avoir un nom
    - ajouter des agences
    - retirer des agences
    - se détruire

Le pattern Etat

- **Motivation**
  - Tant que la banque n'a pas de nom, elle n'est pas valide
  - Elle peut ajouter des agences que si elle est valide
  - Elle peut retirer des agences que si elle en a
  - Elle peut se détruire que si elle est valide et si elle n'a pas d'agences

le pattern Etat



http://smeric.developpez.com/java/uml/etat/

Le pattern Etat

- **Intention**
  - Autoriser un objet X à changer son comportement quand son état interne change
  - Un objet orienté machine d'états
  - Mise en place d'un mécanisme pour donner l'impression que l'objet a été modifié. En fait d'autres objets dits d'états que X doivent gérer :
    - les changements d'états
    - les nouveaux comportements

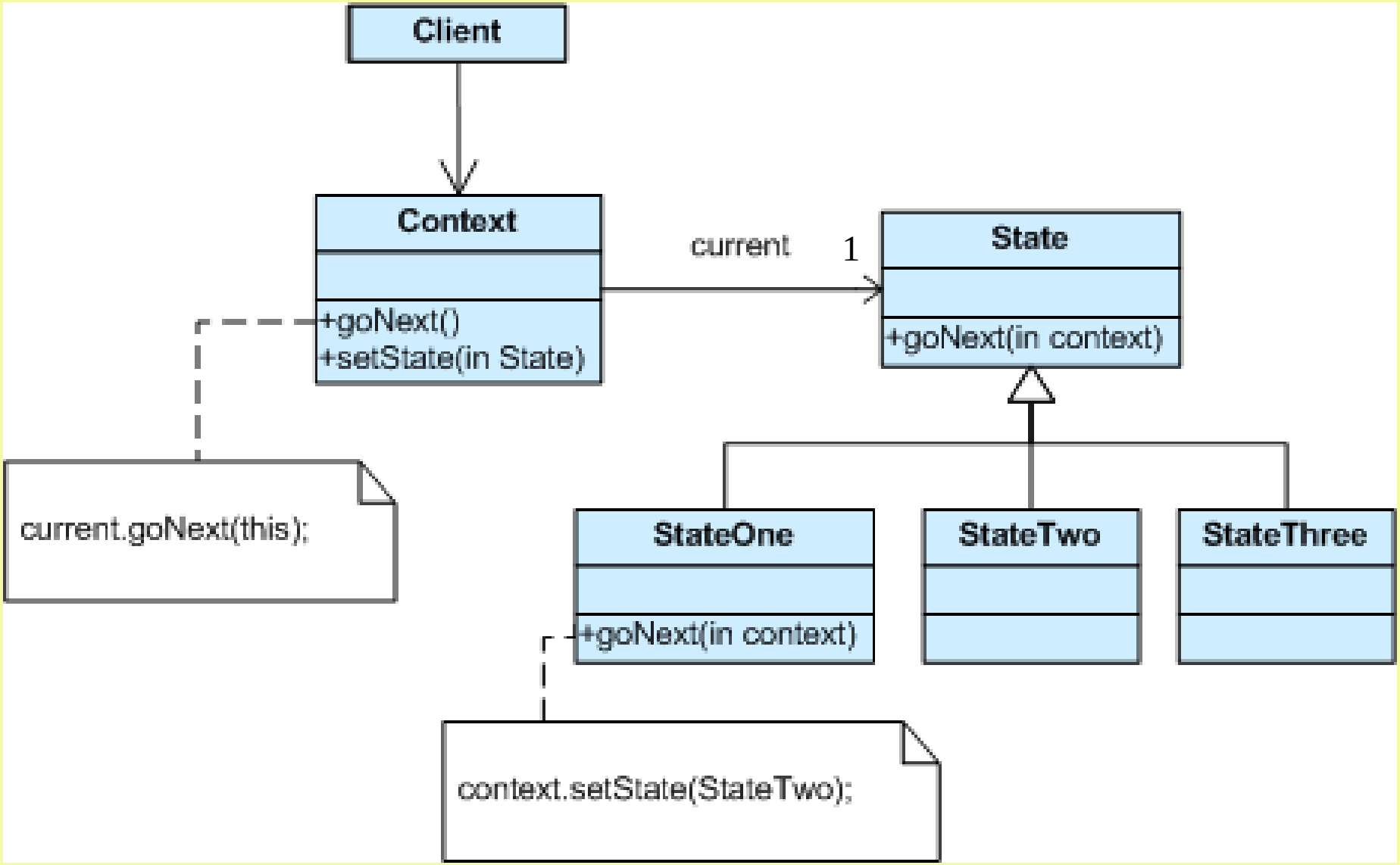
Le pattern Etat

- **Les constituants et leur rôle**
  - L'objet qui change de comportement : le contexte
  - Autant d'objets qu'il y a d'états différents : les états
  - Le contexte connaît à tout moment un unique objet d'état : son état courant
  - A chaque message du client, le contexte demande à son état de traiter le message.

Le pattern Etat

- **Les constituants et leur rôle**
  - Chaque objet d'état connaît son contexte
  - A chaque traitement d'un message, un objet d'état doit s'il y a changement d'état du contexte :
    - allouer un nouvel objet d'état
    - l'affecter à son contexte comme nouvel objet d'état courant

La structure du pattern Etat



Le pattern Iterateur

- **Motivation**
  - On peut vouloir parcourir un conteneur de plusieurs façons différentes
    - Parcourir un arbre en profondeur ou parcourir un arbre en largeur
    - Parcourir une liste du premier élément au dernier élément ou parcourir du dernier élément au premier

Le pattern Iterateur

- **Motivation**
  - Au même instant, on peut avoir en cours, plusieurs parcours d'un même conteneur
  - On peut enfin vouloir parcourir un conteneur sans risque de modification intempestive de la structure interne du conteneur

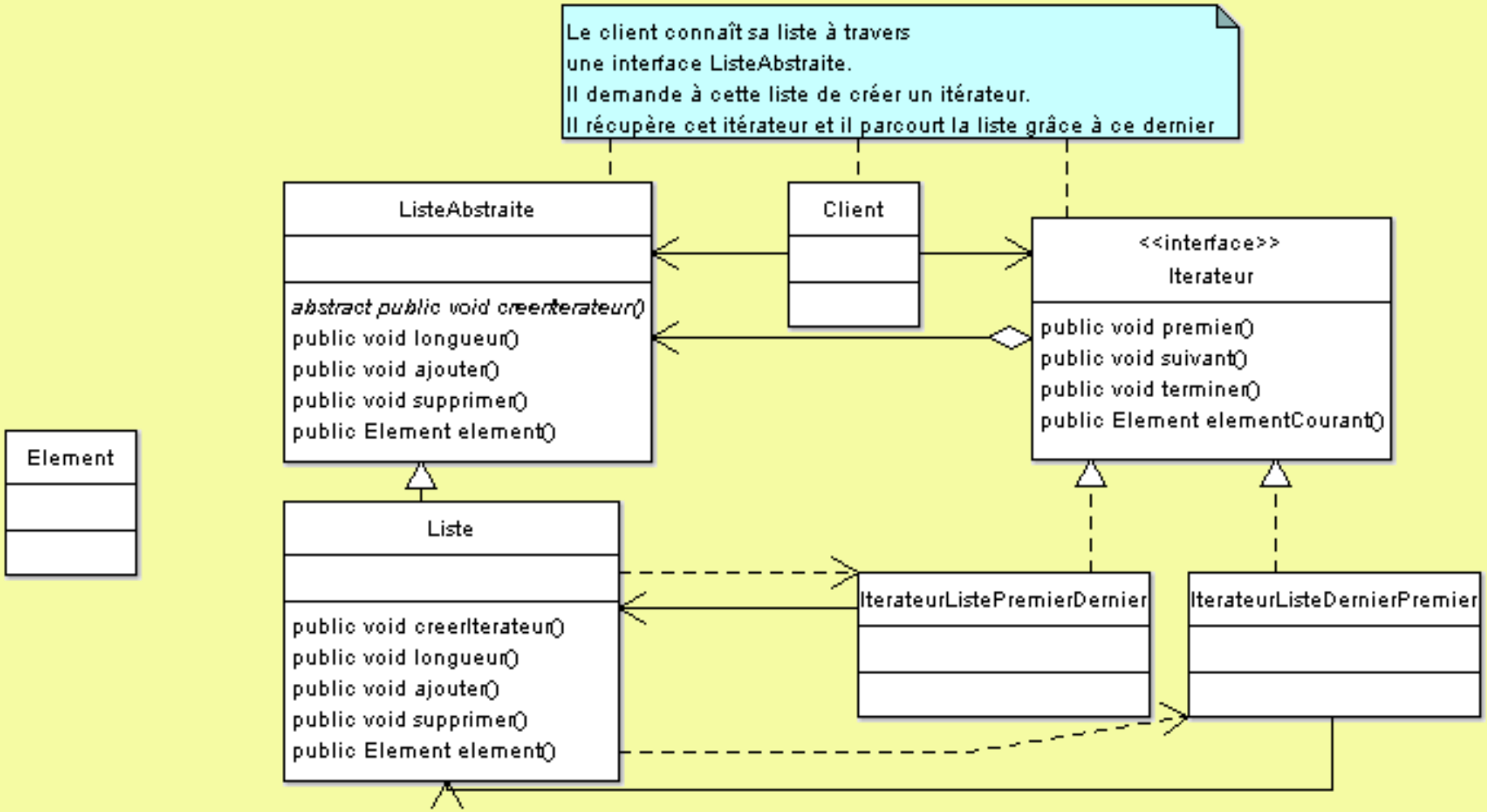
Le pattern Iterateur

- **Intention**

  Fournir un moyen d'accès séquentiel à un agrégat d'objets sans mettre à découvert la représentation interne de ce dernier

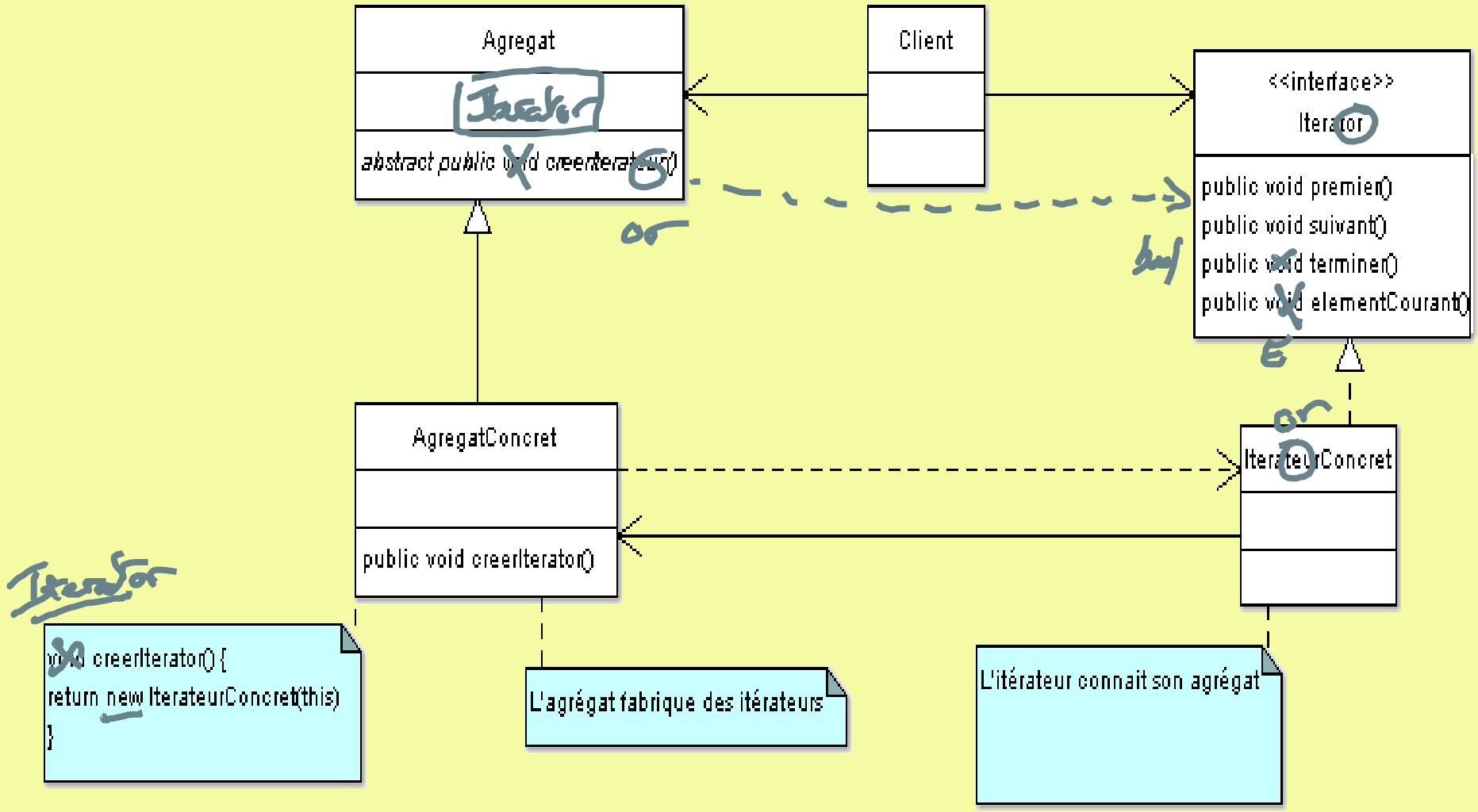
Le pattern Iterateur

La structure du Pattern et ses composants avec une liste



Le pattern Iterateur

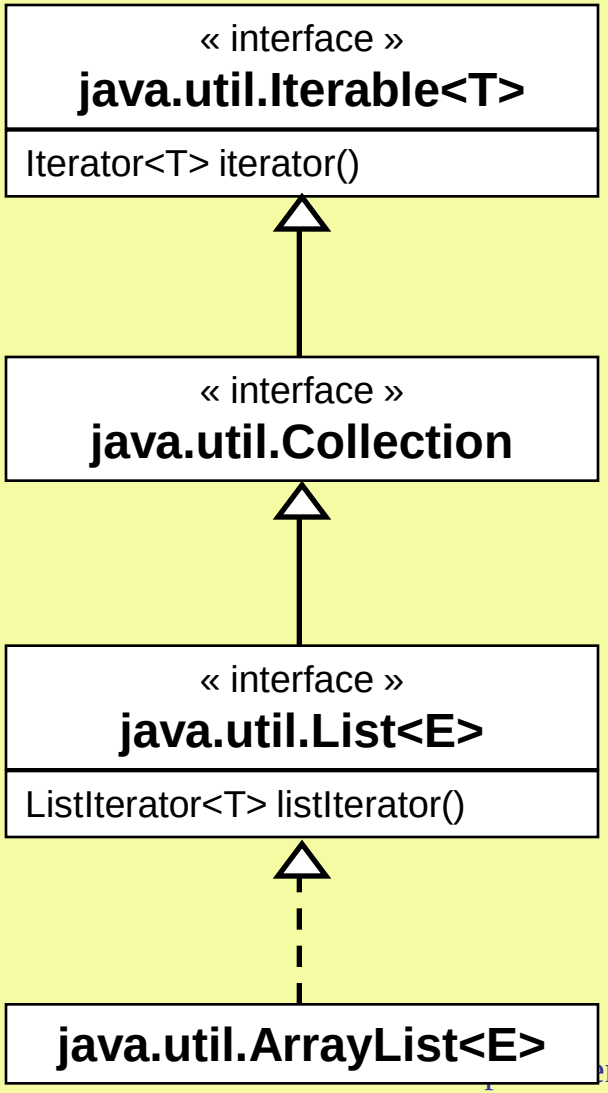
La structure générique du Pattern et ses composants

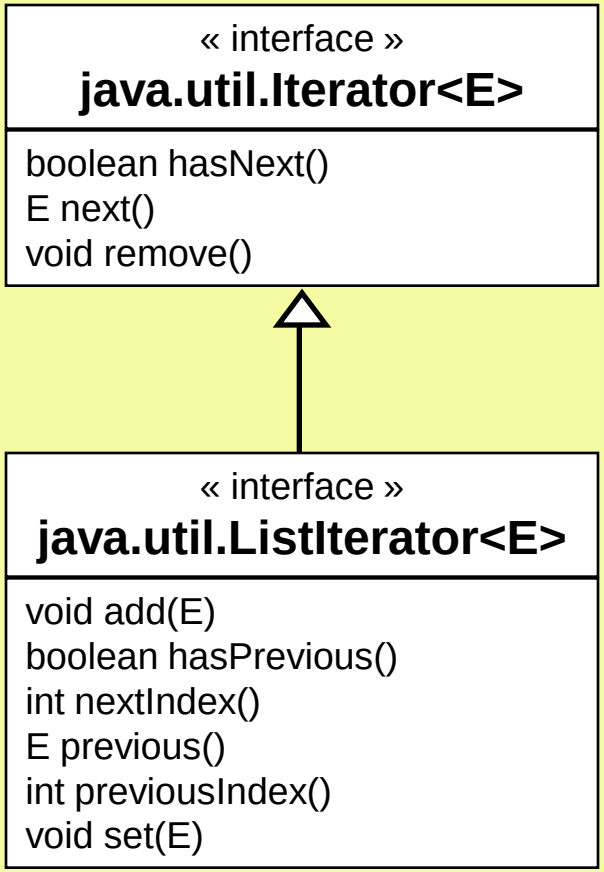


Le pattern Iterateur

- **Composant**
  - La classe IterateurNul est un itérateur dégénéré qui est toujours terminé
  - Les itérateurs en Java

Le pattern Iterateur





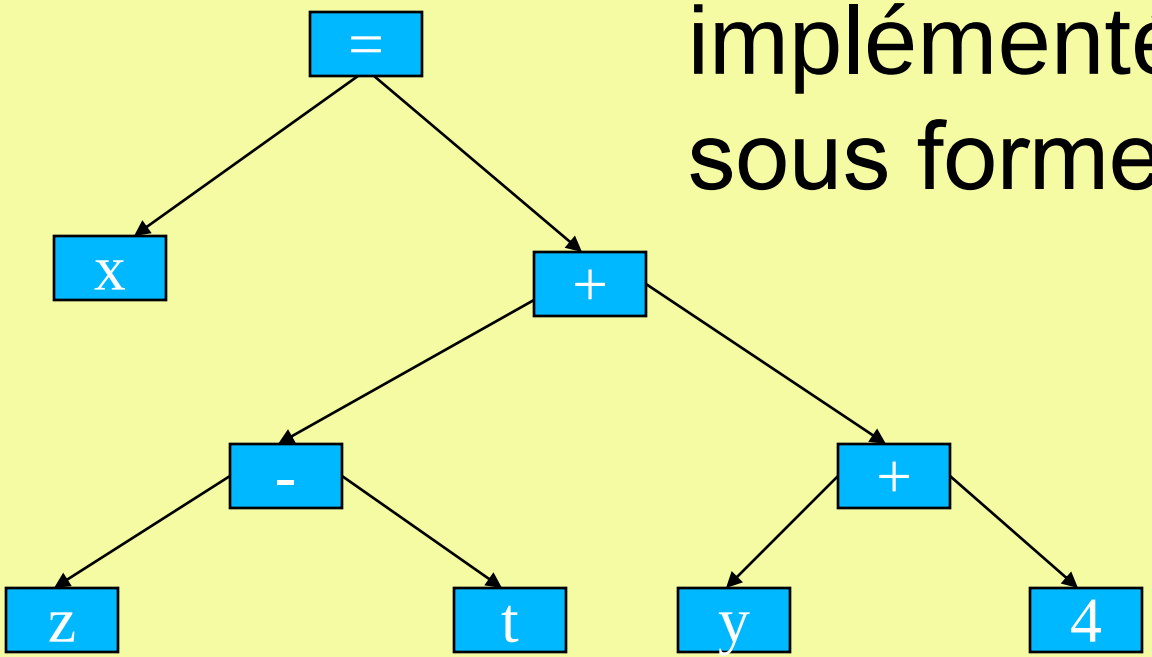
Le pattern Iterateur

```
ArrayList<String> al = new ArrayList<String>();
al.add("C");al.add("A");al.add("E");
al.add("B");al.add("D");al.add("F");

    System.out.print("Contenu initial de al : ");
    Iterator<String> itr = al.iterator();
    while (itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    ListIterator<String> litr = al.listIterator();
    while (litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }

    System.out.print("Liste inversée modifiée : ");
    while (litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
```

Le pattern Visiteur

Motivation

$x = (z - t) + y + 4$



On considère une expression numérique implémentée en mémoire sous forme d'arbre

Le pattern Visiteur

Motivation

On peut parcourir une expression pour

- l'afficher en postfixée
- l'afficher en infixée
- l'évaluer
- vérifier que les variables sont déclarées
- d'autres traitements à venir

Le pattern Visiteur

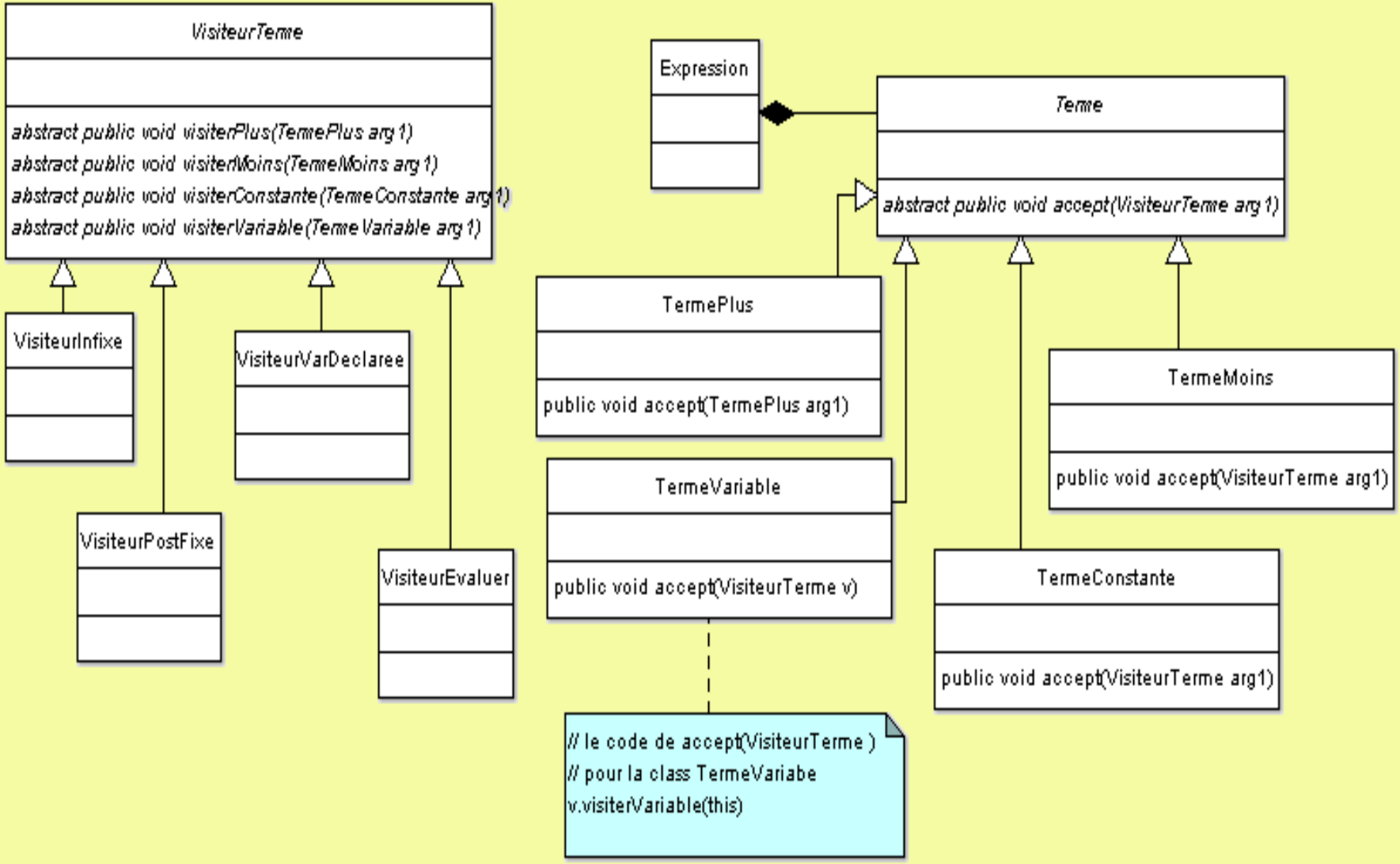
Motivation

Comment gérer cette multiplicité de traitements avec le même objet qui sait naviguer dans sa structure mais qui n'a pas nécessairement à savoir ce qu'il faut faire sur chaque élément (+, -, variable, constante) rencontré.

Le pattern Visiteur

- **Intention**
  - Ce pattern s'intéresse à une opération qui doit être effectuée sur les éléments (+, -, variable, constante) de la structure d'un objet (expression).
  - La pattern Visiteur vous permet de définir une nouvelle version de cette opération sans changer les classes des éléments sur lesquels il opère.

le pattern Visiteur



Le pattern Visiteur

- **Les constituants et leur rôle**
  - **Le client connaît l'objet qui est une structure d'objets (expression)**
  - **Le client connaît via l'interface Visiteur l'objet qui sait faire les traitements dans un contexte particulier (afficher, évaluer, …) pour chaque élément de la structure**

Le pattern Visiteur

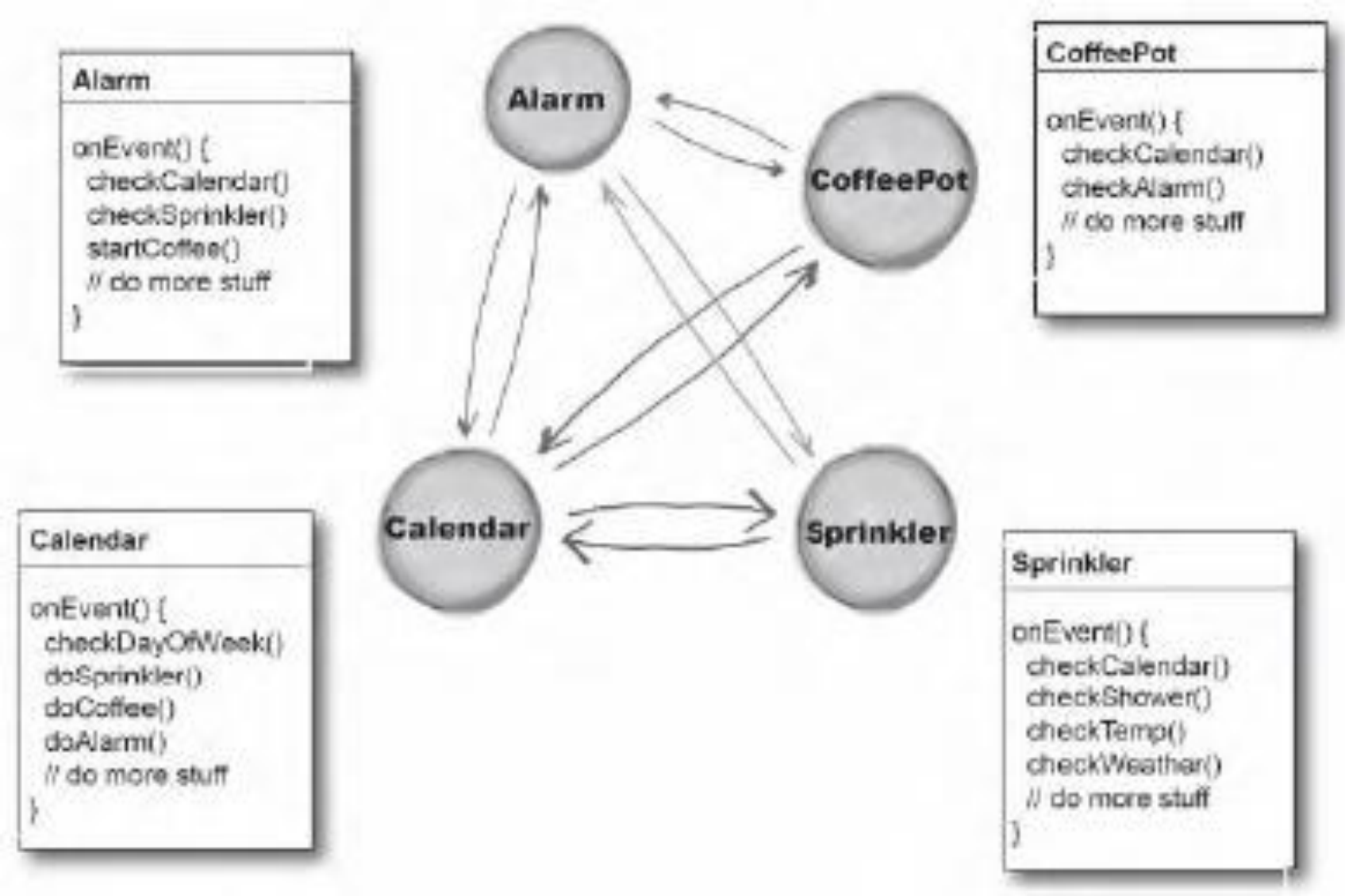
- **Les constituants et leur rôle**
  - **Chaque objet particulier de la structure accepte d'être visité par n'importe quel visiteur en implémentant l'interface Objet**

```
accept(Visitor v)  {
    v.visiteObjet(this);
};
```

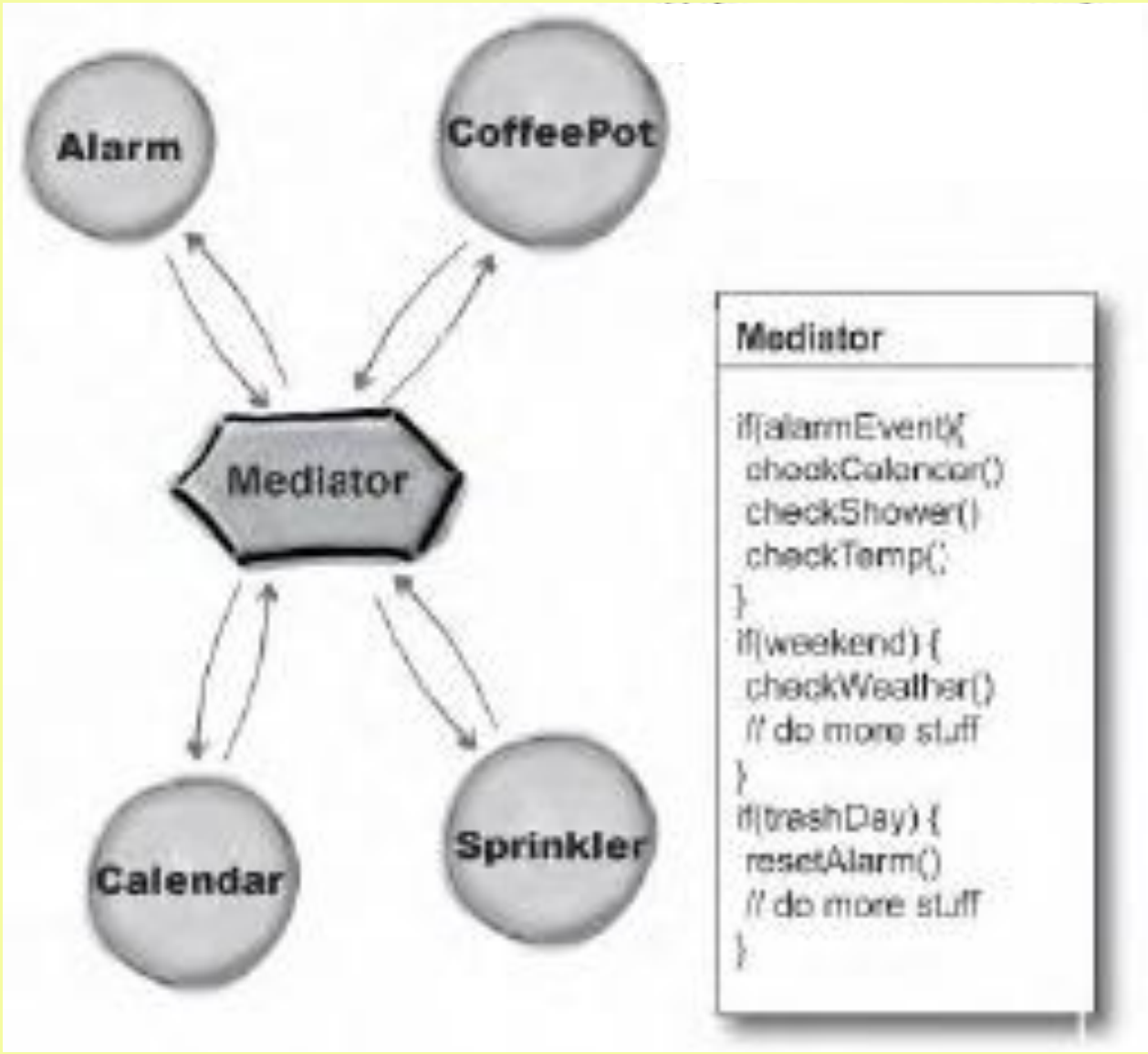
Le pattern Médiateur

- Motivation
  - Un système domotique du futur
    - L'arrêt du réveil déclenche la machine à café
    - Cela doit fonctionner tous les jours sauf le week-end
    - L'arrosage automatique doit s'arrêter 15 min avant la programmation d'une douche
    - Mettre le réveil plus tôt le jour des poubelles
    - …

Le pattern Médiateur



Le pattern Médiateur



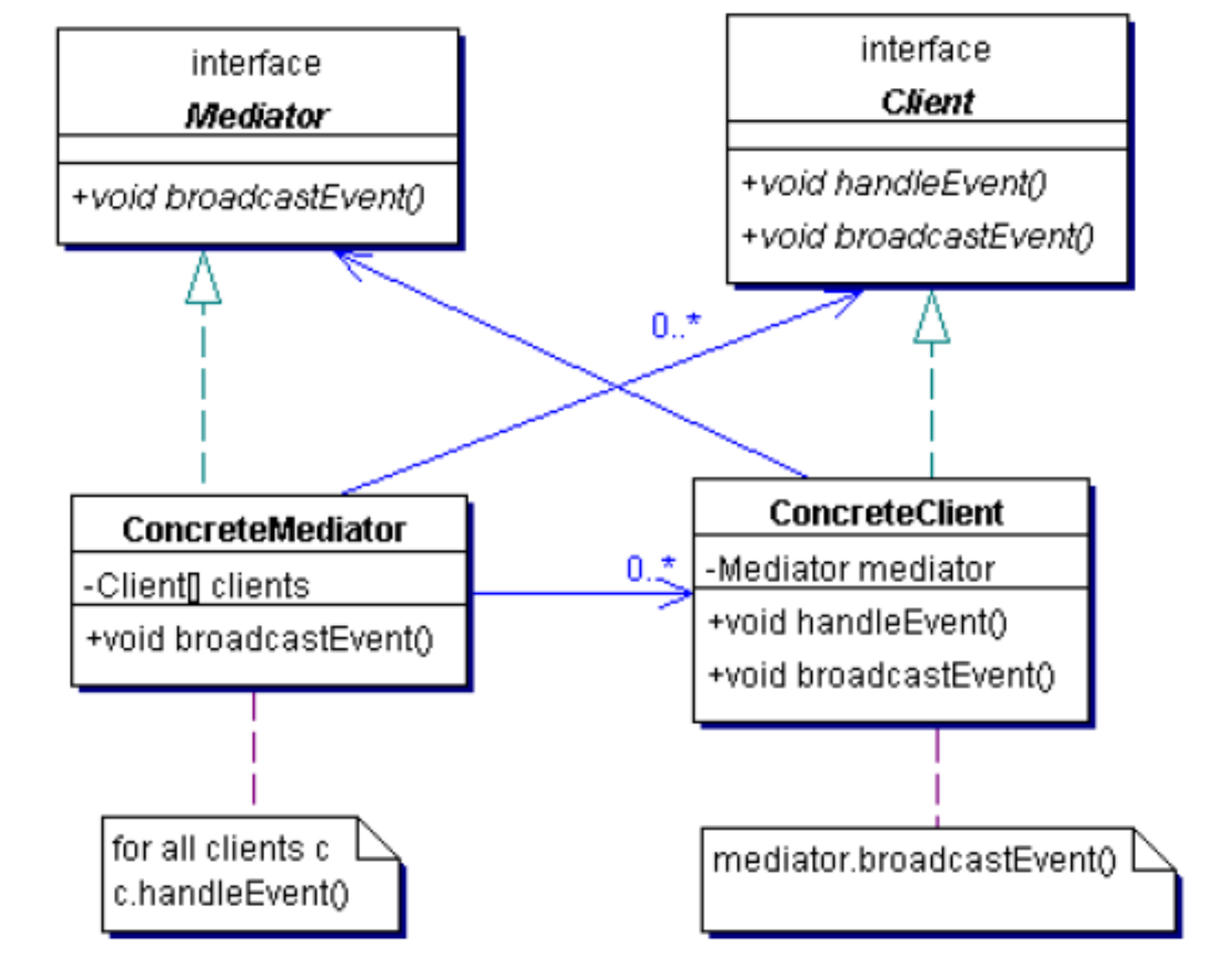
Le pattern Médiateur

- Intention
  - Encapsuler dans un objet m les modalités d'interaction d'un grand ensemble d'objets E
  - Cet objet m est le médiateur
  - Il permet donc un couplage faible en dispensant les objets de E de se faire explicitement référence

Le pattern Médiateur

- Constituants
  - une interface Médiateur pour déclarer les opérations qui permettront aux objets dit collègues de communiquer entre eux
  - une classe MédiateursConcrets qui implémente l'interface Médiateur. Un objet de cette classe a des références sur les différents collègues à gérer
  - les classes Collègues qui vérifient les propriétés suivantes :
    - Un objet Collègue connait son médiateur
    - Un objet Collègue communique avec un autre objet Collègue par envoi de message à son médiateur

Le pattern Médiateur



Le pattern Interpréteur

- Motivation
  - On désire pour un langage donné
    - définir une représentation de sa grammaire
    - définir un interpréteur utilisant cette représentation pour interpréter les phrases du langage

Le pattern Interpréteur

- Motivation
  - Soit le langage

- **expression ::= litteral | alternation | sequence | repetition | ‘(’ expression ‘)’**
- **alternative ::= expression ‘|’ expression**
- **sequence ::= expression ‘&’ expression**
- **repetition ::= expression ‘*’**
- **literal ::= ‘a’ | … | ‘z’ {‘a’ | … | ‘z’}***

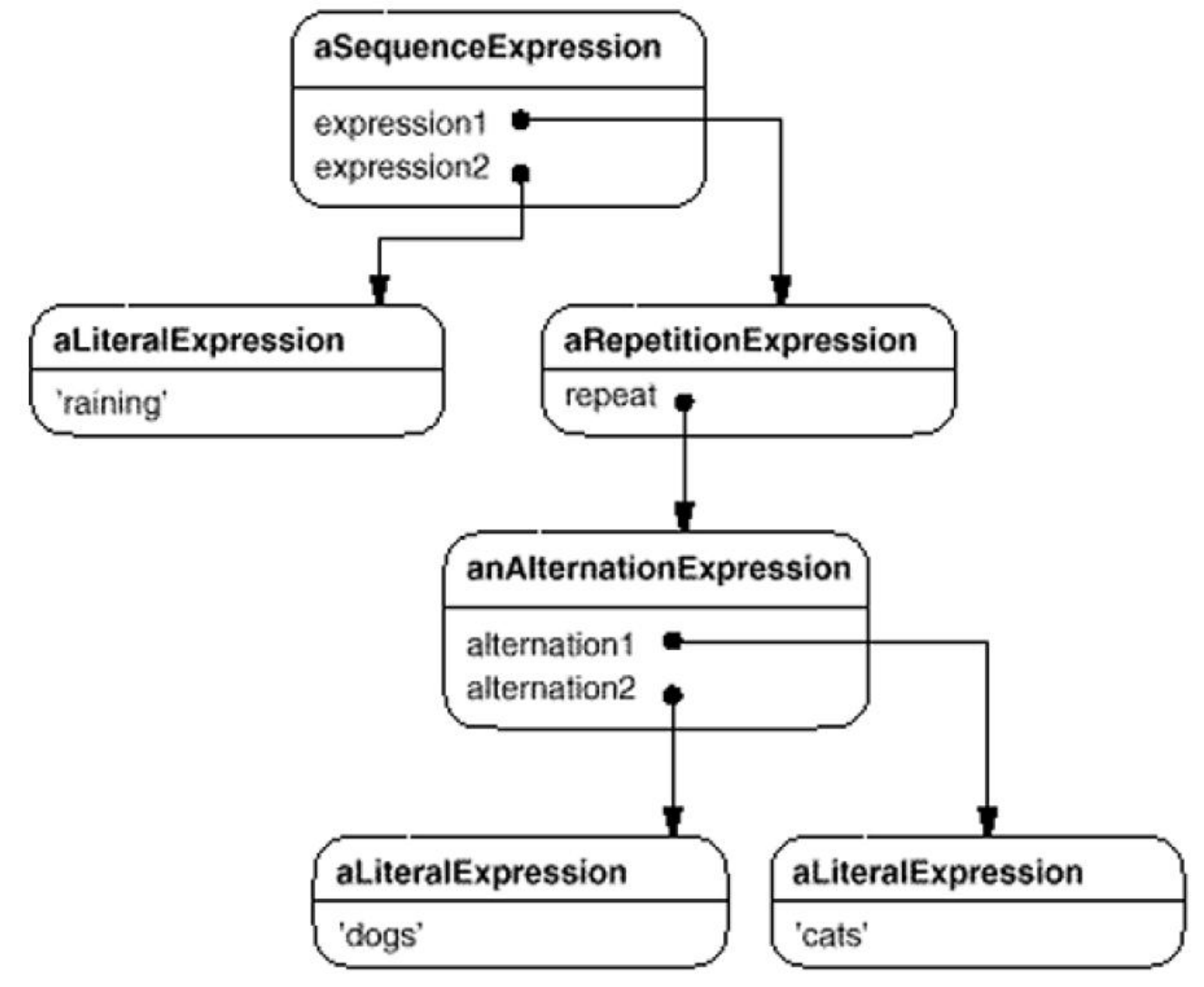
Le pattern Interpréteur

- Motivation
  - Le point de départ de l'analyse est l'expression non terminale (le point de départ constitue une classe particulière)
  - Chaque règle définit une classe
  - Les symboles de la partie de droite d'une règle qui définissent une forme sont des variables d'instance de ces classes

Le pattern Interpréteur

- Question ?
  - Exprimer l'expression régulière `raining & (dogs | cats)` sous forme d'un arbre syntaxique dont chaque nœud est une instance d'une des classes définies ci-dessus

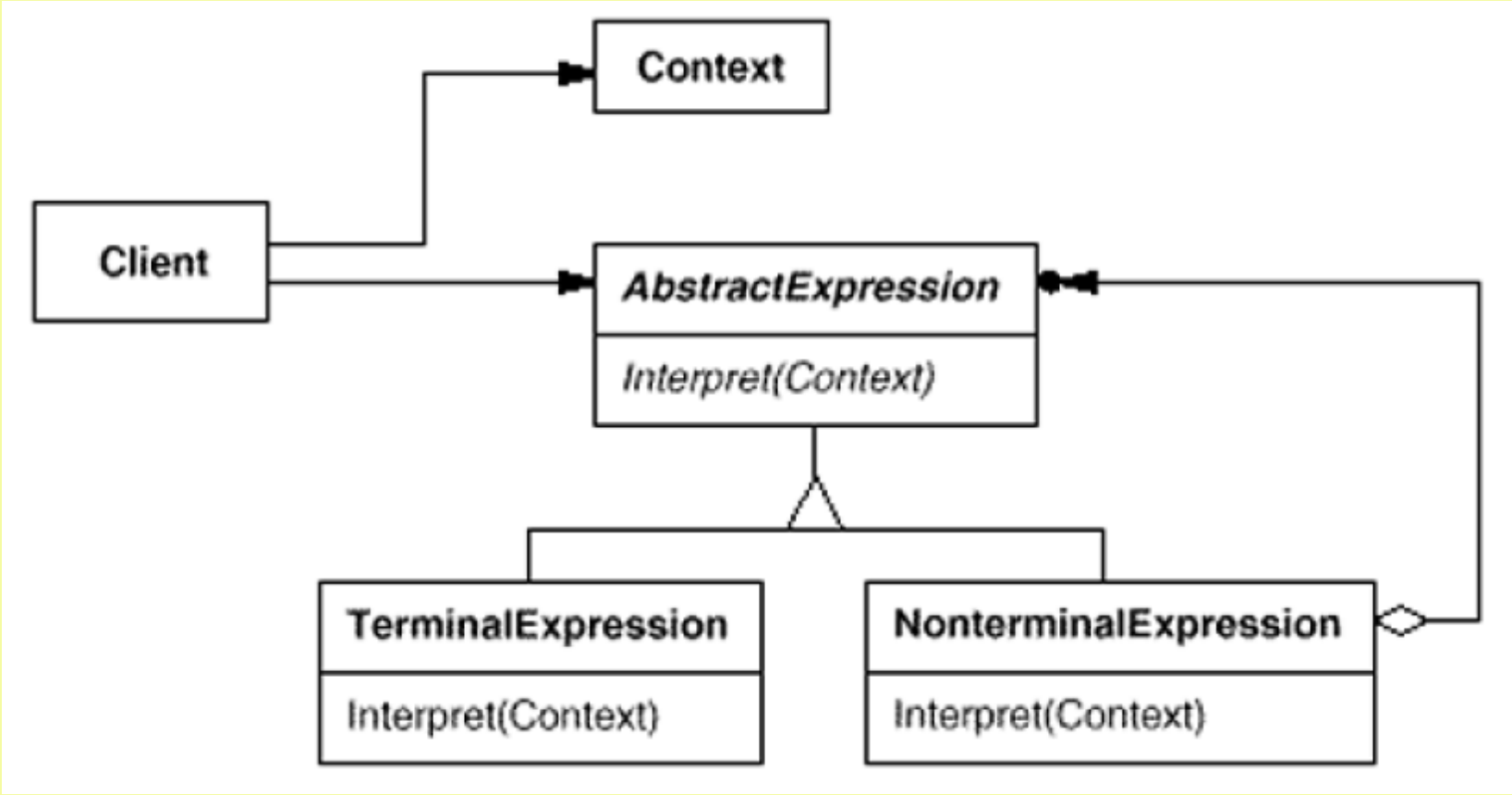
Le pattern Interpréteur



Le pattern Interpréteur

- Intention
  - Définir un interpréteur pour un langage

Le pattern Interpréteur



Le pattern Interpréteur

- Constituants
  - AbstractExpression
    - déclare une opération abstraite interpret() communes à tous les nœuds dans l'arbre de la syntaxe abstraite
  - TerminalExpression
    - implémente une opération interpret() associée à un symbole terminal dans la grammaire
    - une instance est requise pour chaque symbole terminal dans une phrase

Le pattern Interpréteur

- Constituants
  - NonterminalExpression
    - une telle classe est requise pour chaque règle $R ::= R_1 R_2 \ldots R_n$ dans la grammaire
    - maintient des variables d'instance de type AbstractExpression pour chaque symbole de R_1 à R_n
    - implémente une opération interpret() les symboles non terminaux dans la grammaire. Typiquement, interpret() s'appelle récursivement sur les variables représentant $R_1, \ldots, R_n$

Le pattern Interpréteur

- Constituants
  - Context
    - contient l'information globale à l'interpréteur
  - Client
    - construit (ou on lui donne) un arbre syntaxique abstrait représentant une phrase particulière du langage définie par la grammaire. L'arbre syntaxique abstrait est assemblé à partir des instances des classes NonterminalExpression et TerminalExpression
    - invoque l'opération interpret()

Le pattern Interpréteur