

POO-IG

# Programmation Orientée Objet et Interfaces Graphiques

---

Cristina Sirangelo  
IRIF, Université de Paris  
[cristina@irif.fr](mailto:cristina@irif.fr)

# Design patterns

---

Exemples, code et matériel empruntés :

\* Cay Horstmann. OO Design & Patterns, 2nd ed.

# Conception et implémentation de programmes

---

- Aller d'une description informelle d'un problème à résoudre, jusqu'à l'implémentation d'un programme qui le résout
- Nécessite plusieurs étapes
  - Analyse
  - "Design" (conception / modélisation)
  - Implémentation

# Conception et implémentation de programmes

## □ Analyse :

- Fournir une description complète du logiciel à produire
  - appelée spécification (fonctionnelle) du logiciel
  - typiquement en format textuel en langage naturel (des formalismes existent mais rarement utilisés)
- décrit **ce** que le logiciel doit faire et **pas comment**
- Informations que la spécification doit fournir :
  - les fonctionnalités du programme
  - leurs interaction
  - les modalités d'interaction avec l'interface graphique
  - en résumé : toutes les infos que l'on trouverait dans un manuel utilisateur du logiciel
- Ne dépend pas du langage de programmation ni du paradigme de programmation (OO, fonctionnel, etc)

# Conception et implémentation de programmes

---

- **Design (modélisation)**
  - Identification des éléments (données) du programme, de leurs interactions et des fonctionnalités qui les manipulent
  - dépend du paradigme de programmation choisi, mais pas du langage
  - Pour le paradigme OO le design revient à l'identification :
    - des **classes**
    - de leurs **responsabilités** (quelle classe fournit quelle fonctionnalité)
    - de leurs **relations** (quelle classe dépend / étend / agrège quelle autre classe)
  - Produit : une spécification formelle des classes, de leurs relations, de leur fonctionnalités et de l'évolution de l'état des objets
  - Des formalismes existent et sont souvent utilisés : UML, diagrammes d'état

# Conception et implémentation de programmes

---

## □ **Implementation**

- Développement, test et mise en oeuvre du logiciel
- Dans le paradigme orienté objet : croissance progressive du programme par l'ajout d'une classe / groupe de classes à la fois (accompagné de tests)
- Souvent prototypage : retarder l'implémentation de certaines fonctionnalités pour monter rapidement une version prototype à faire évoluer

# Design patterns

---

- Nous nous concentrerons dans ce cours sur la phase de design (modélisation)
- Il n'y a pas de recette à suivre, chaque problème à modéliser est différent
- Cependant certaines situations à modéliser sont récurrentes
- Design pattern :
  - Une description abstraite d'un problème de modélisation et d'un arrangement d'éléments (classes et objets dans notre cas) qui le résout.
  - Utilité : à chaque fois que le problème de modélisation décrit par le design pattern se présente on peut utiliser la solution qu'il propose.
- Il en existe plusieurs dans la littérature  
cf.  
*Gamma, Hendl, Johnson, Vlissides - Design Patterns: Elements of Reusable Object-Oriented Software*
- Nous en étudierons quelques uns...

# Design patterns

---

- Un design pattern doit spécifier
  - les conditions dans lesquelles il peut être utilisé
  - la solution, i.e.
    - les classes et instances qui participent à la solution,
    - leur responsabilités et
    - leur relations
  - les conséquences de son utilisation sur la modélisation



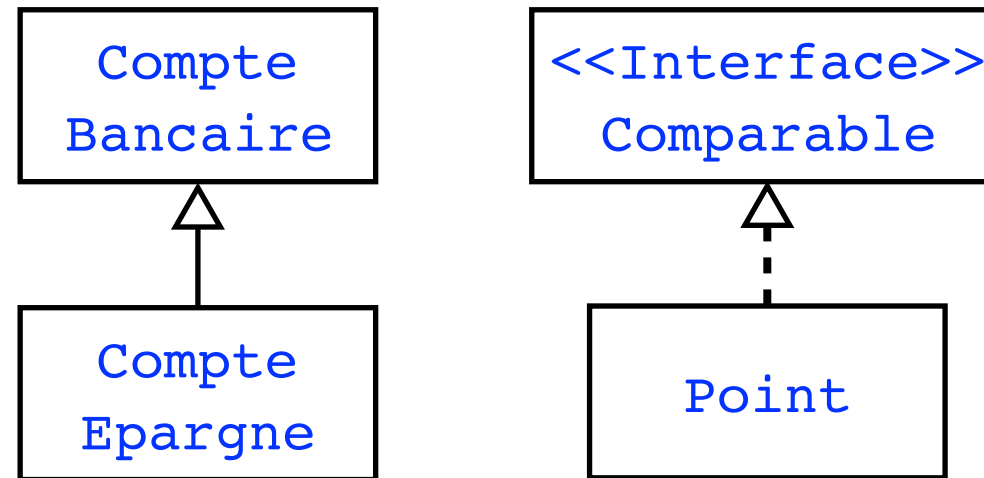
# Relation entre classes

---

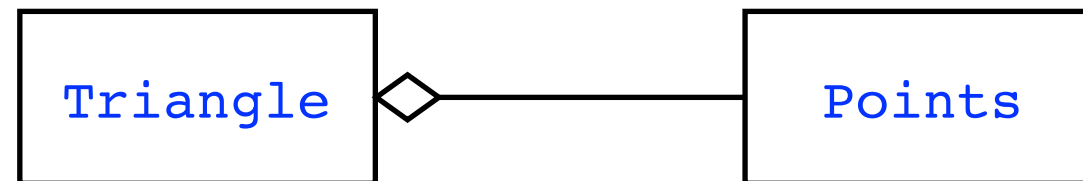
- Trois types récurrents de relations entre classes :
  - **Heritage** ("être")
    - Ex : un CompteEpargne "est" un CompteBancaire
  - **Agrégation** ("avoir")
    - Ex : un Triangle "a" des Points
  - **Dépendance** ("utiliser")
    - Ex : un Achat utilise un CompteBancaire (les méthodes d'Achat manipulent des objets CompteBancaire)
    - Il y a dépendance à chaque fois qu'une classe a besoin de connaître une autre classe
    - L'agrégation est un cas particulier de dépendance

# Notation graphique (UML)

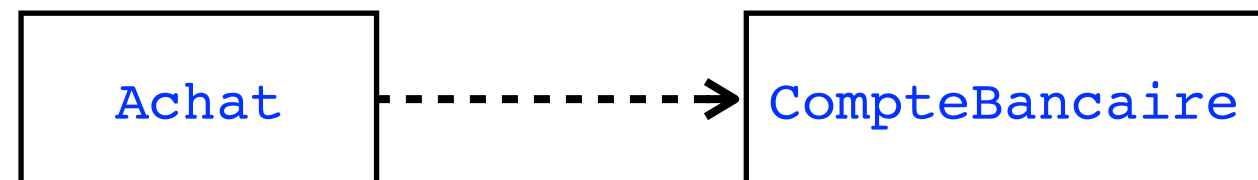
- **Heritage** ("être")



- **Agrégation** ("avoir")

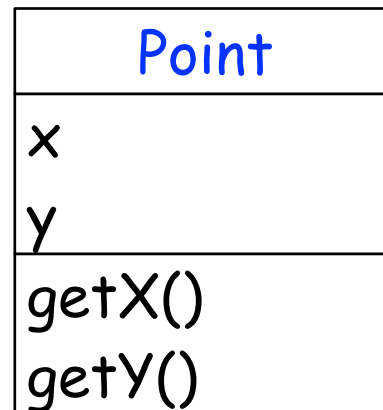


- **Dépendance** ("utiliser")

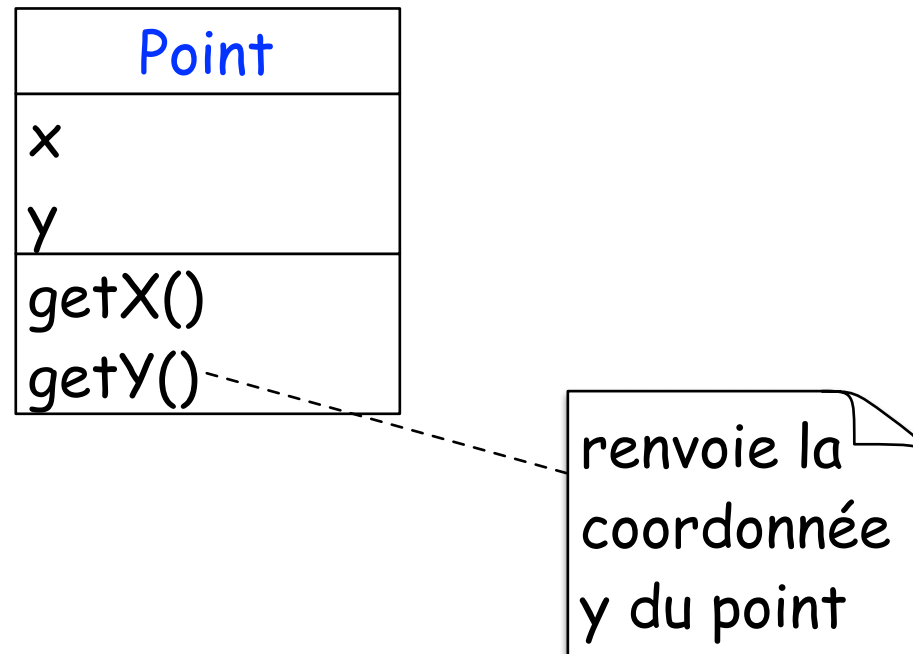


# Notation graphique (UML)

- Pour spécifier qu'une classe a des champs et méthodes (pas forcément tous) :



- Pour ajouter la description d'une méthode :



# Quelques design patterns

---

# Iterator pattern

---

# Exemple de problème : parcourir une liste

---

- Exemple : une classe Liste (e.g. la classe LinkedList de Java)
  - collectionne des objets
  - doit permettre aux classes clientes d'accéder à ces objets (parcourir la liste)
  
- Comment la modéliser?

# Parcourir une liste : une solution naïve

- Exposer la structure des liens pour que les classes clientes puissent les parcourir

```
public class ListElem {
    public String data;
    public ListElem suite = null;
    public ListElem (String data) { this.data = data;}
}

public class MaListe {
    public ListElem head = null;
    public ListElem last = null;
    public Liste (){};
    public void append (String data) {
        ListElem e = new ListElem(data);
        if (head == null) {head = e;}
        else {last.suite = e;}
        last = e;
    }
    ...
}
```

# Parcourir une liste : une solution naïve

- Une classe qui a besoin de parcourir une liste

```
public class classeClient {  
    ...  
    public void f( MaListe l) {  
        ListElem currentElem = l.head;  
        while (currentElem != null){  
            String current = currentElem.data;  
            currentElem = currentElem.next;  
        }  
    }  
}
```

Donne accès à l'implémentation - ne préserve pas l'encapsulation.  
Susceptible de favoriser les erreurs



# Parcourir une liste : une solution avec des limites

## Liste avec curseur

- Ajouter un champ curseur, qui maintient un pointeur à l'élément courant du parcours
- le curseur peut être déplacé et l'élément sous le curseur peut être lu

```
public class MaListe {  
    private static class ListElem {  
        String data;  
        ListElem suite = null;  
        ListElem (String data) { this.data = data;}  
    }  
    private ListElem head = null;  
    private ListElem last = null;  
    private ListElem cursor = null;  
    public void reset() { cursor = head;}  
    public boolean hasNext() { return cursor!= null; }  
    public String next() { String n = cursor.data;  
        cursor = cursor.suite; return n;} //precond: hasNext()  
    ...  
}
```

# Parcourir une liste : une solution avec des limites

- Une classe qui a besoin de parcourir une liste avec curseur

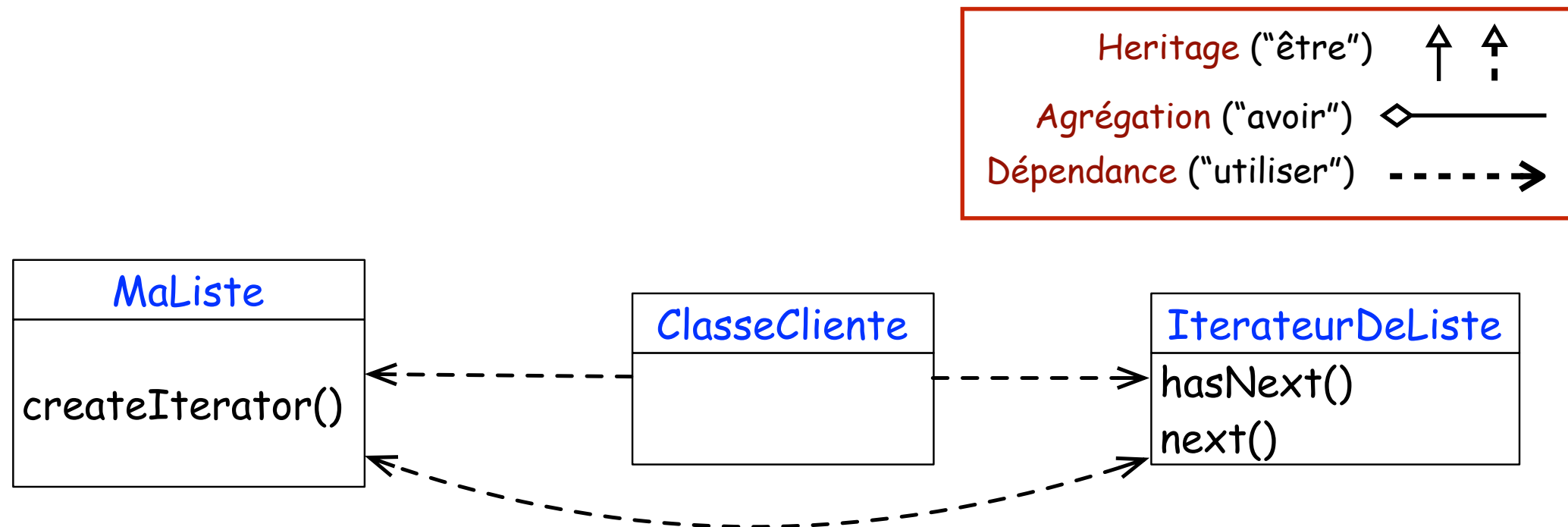
```
public class classeCliente {  
    ...  
    public void f( MaListe l ) {  
        l.reset();  
        while (l.hasNext())  
            String s = l.next();  
    }  
}
```

**OK :** Le client n'a pas accès à la structure interne de la liste (ListElem peut même être une classe interne privée!)

**Inconvénient :** une liste peut avoir un seul curseur, pas possible d'effectuer plusieurs parcours de la même liste en même temps

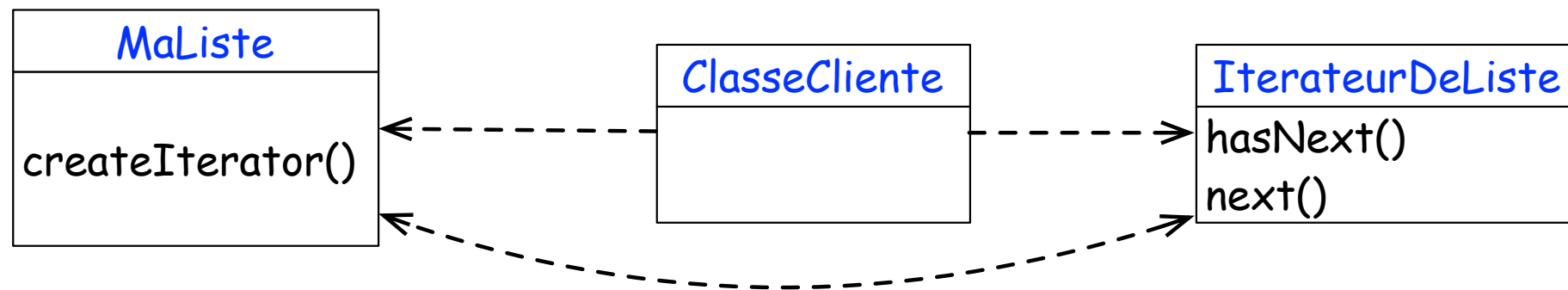
# Parcourir une liste : une bonne solution

- Introduire une classe "itérateur" : la seule qui a accès à la structure interne de la liste
  - offre des méthodes de parcours (next(), hasNext())
- La classe Liste a une méthode qui crée un itérateur et le fournit aux classes clientes



- Plusieurs itérateurs peuvent parcourir la liste en même temps !

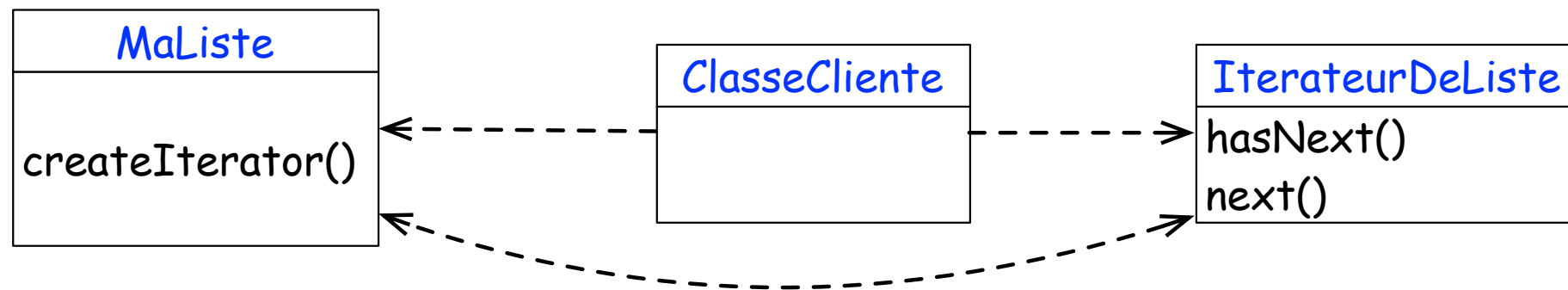
# Parcourir une liste : une bonne solution



- En Java implémentation naturelle par classe interne (membre)

```
public class MaListe {
    private static class ListElem {...}
    private ListElem head = null;
    private ListElem last = null;
    public class IterateurDeListe {
        private ListElem cursor = head;
        public boolean hasNext() { return cursor != null; }
        public String next() { String n = cursor.data; cursor =
            cursor.suite; return n; } //precond: hasNext()
    }
    public IterateurDeListe createIterator() {
        return new IterateurDeListe();
    }
    ...
}
```

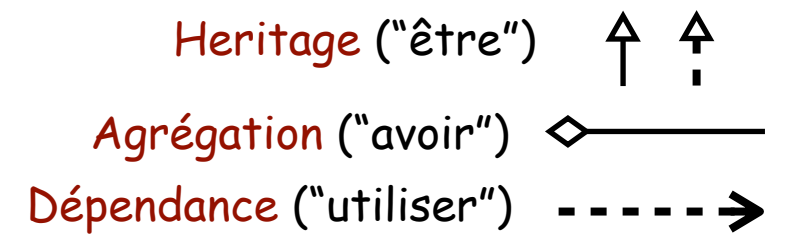
# Parcourir une liste : une bonne solution



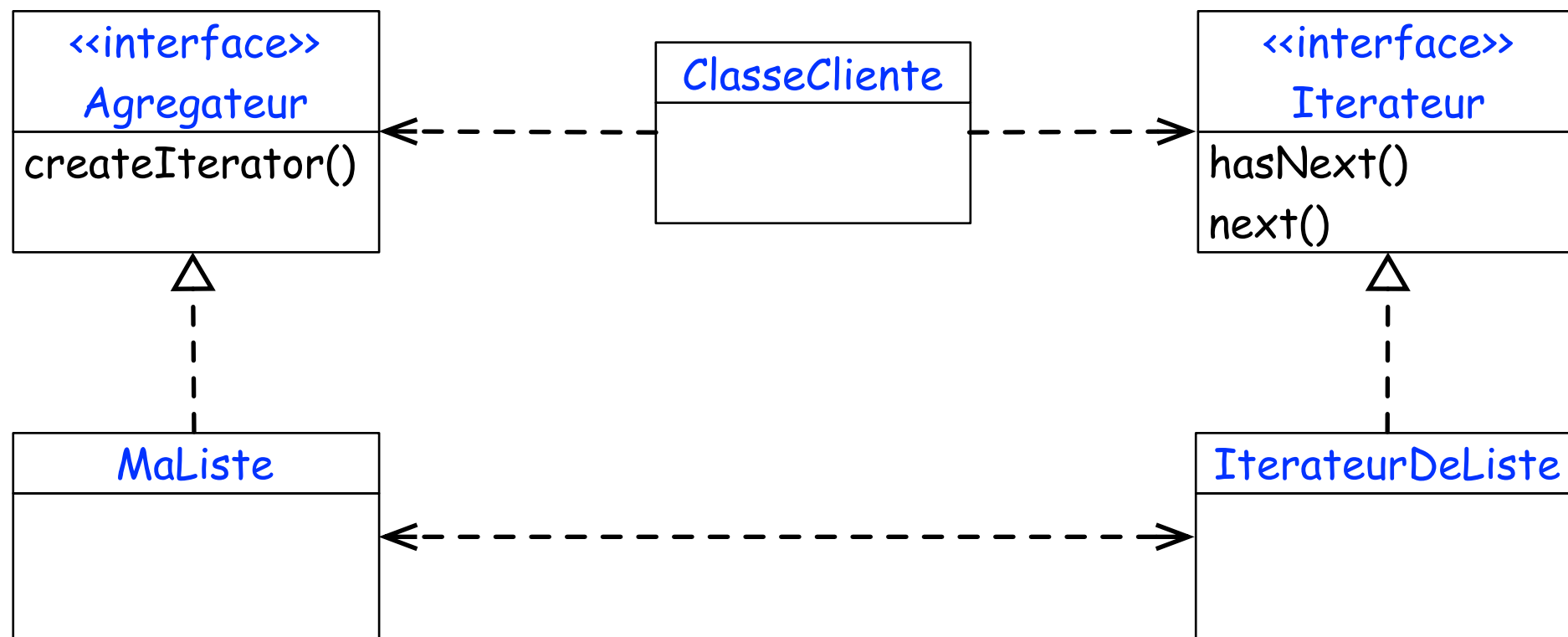
- La classe cliente doit connaître à la fois la classe Liste et sa classe Iterateur

```
public class ClasseClient {
    public static void f( MaListe l ) {
        Liste.IteratorDeListe it1 = l.createIterator();
        Liste.IteratorDeListe it2 = l.createIterator();
        while (it1.hasNext()) { String s = it1.next(); }
        while (it2.hasNext()) {...}
    }
    ...
}
```

# Parcourir une liste : mieux



- Pour pouvoir appliquer ce mécanisme à plusieurs types de listes et plusieurs types d'itérateurs, la solution suivante est plus flexible

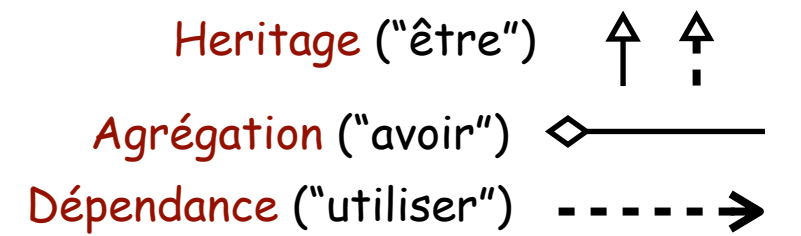


Remarque : des interfaces qui ont le rôle de Agregateur et Iteratur existent déjà dans l'API Java :

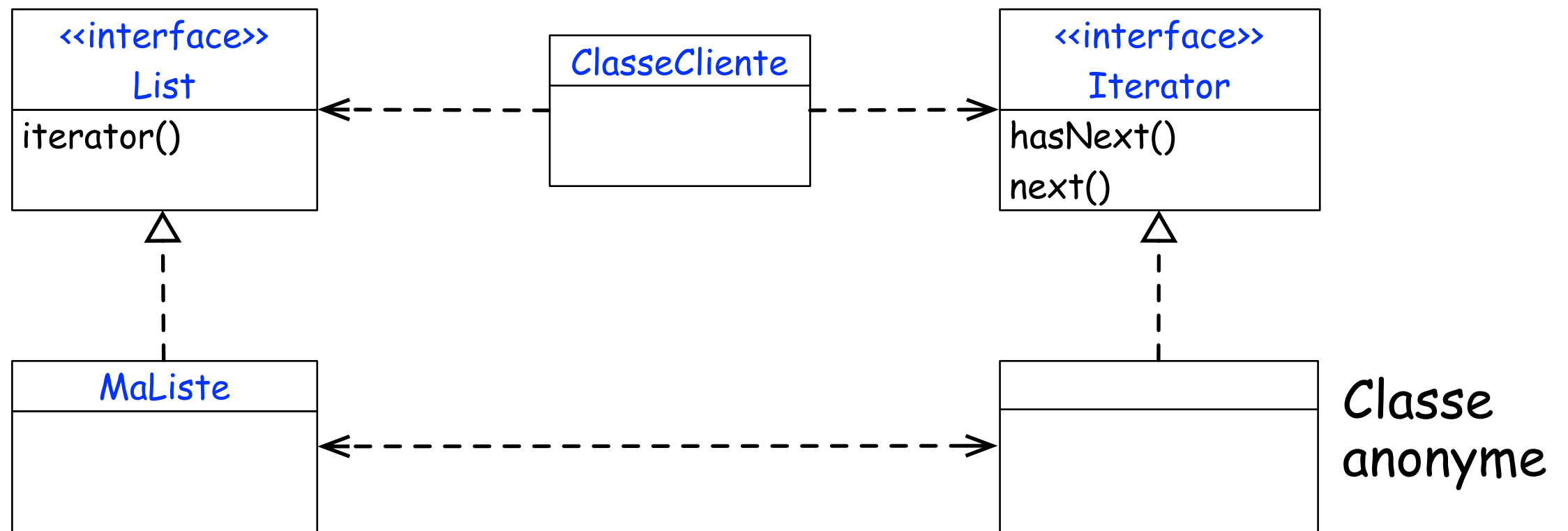
Agregateur : `List<E>` avec méthode `iterator()`

Iterateur: `Iterator<E>` avec méthodes `next()` et `hasNext()`

# Parcourir une liste : mieux



- Implémentation en Java (avec les interfaces de l'API)



# Parcourir une liste : une meilleure solution

## ▫ Implémentation en Java

```
public class MaListe implements List<String>{
    private static class ListElem {...}
    private ListElem head = null;
    private ListElem last = null;
    public Iterator<String> iterator() {
        return new Iterator<String>(){
            private ListElem cursor = head;
            public boolean hasNext() { return cursor != null; }
            public String next() {
                String n = cursor.data; cursor = cursor.suite;
                return n;
            } //precond: hasNext()
        };
    }
    ...
}
```

- La classe Java `LinkedList` est implémentée selon ce même pattern



# Parcourir une liste : une meilleure solution

- Implémentation en Java

```
public class ClasseClient {
    public void f( List<String> l ) {
        Iterator<String> it1 = l.iterator();
        Iterator<String> it2 = l.iterator();
        while (it1.hasNext()) {
            String s = it1.next();
        }
        while (it2.hasNext()) {...}
    }
    ...
}
```

- Avec cet exemple nous sommes parvenus à utiliser un design pattern connu sous le nom de "Iterator pattern" pour concevoir nos classes

# Iterator pattern

---

## □ Contexte

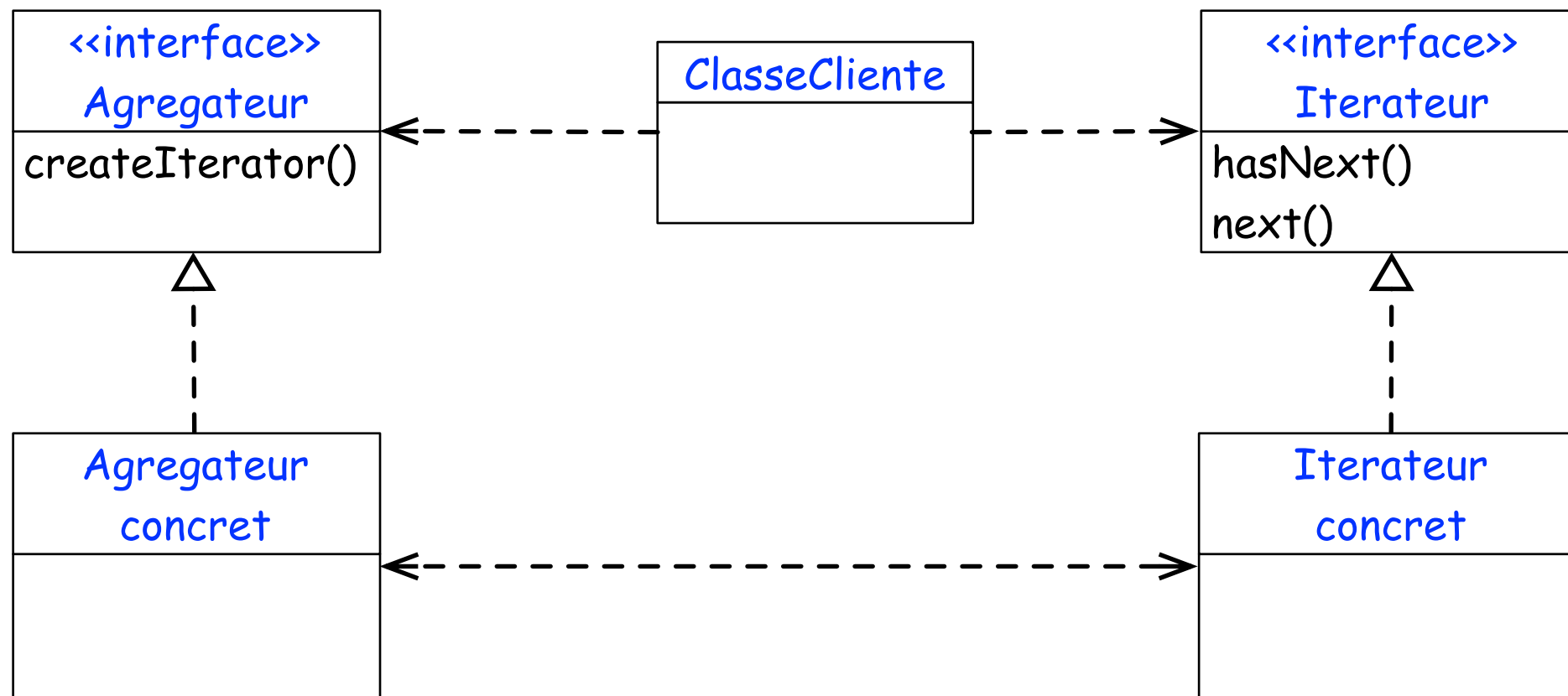
- Un objet agrégateur contient des objets éléments
- Les classes clientes doivent pouvoir accéder aux éléments
- L'objet agrégateur ne peut pas exposer sa structure interne
- Plusieurs accès indépendants aux éléments doivent être possible en même temps

# Iterator pattern



## Solution

- Définir une classe Itérateur qui récupère les éléments un à la fois
- L'agrégateur crée et retourne aux classes clientes des itérateurs sur ses éléments
- S'il y a plusieurs variantes d'agrégateur/itérateur elles implémentent des interfaces communes
  - les classes clientes dépendent uniquement de ces interfaces



# Observer pattern

---

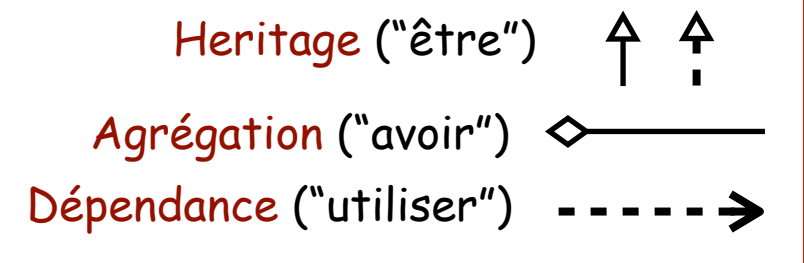
# Observer Pattern

---

## Contexte

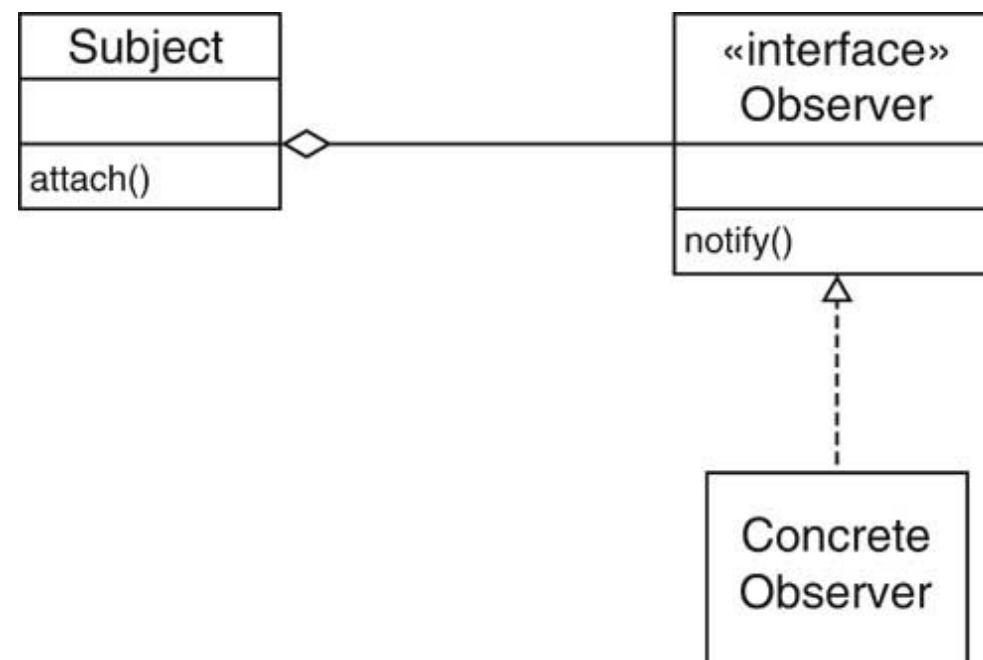
- Un objet, appelé le Sujet, est source d'événements  
(Exemple : un Bouton de l'interface graphique)
- Un ou plusieurs autres objets observateurs veulent être notifiés  
quand ces événements ont lieu  
(Exemple : les objets qui réagissent au bouton pressé)

# Observer Pattern

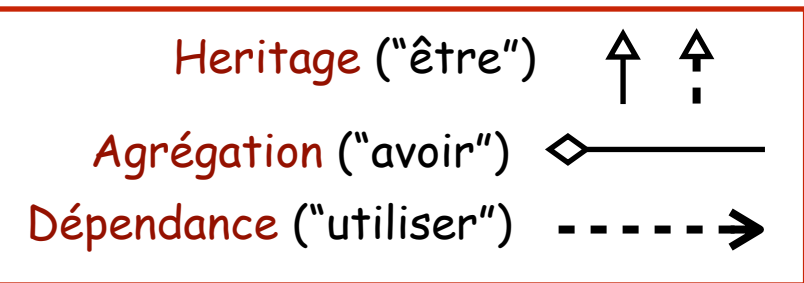


## Solution

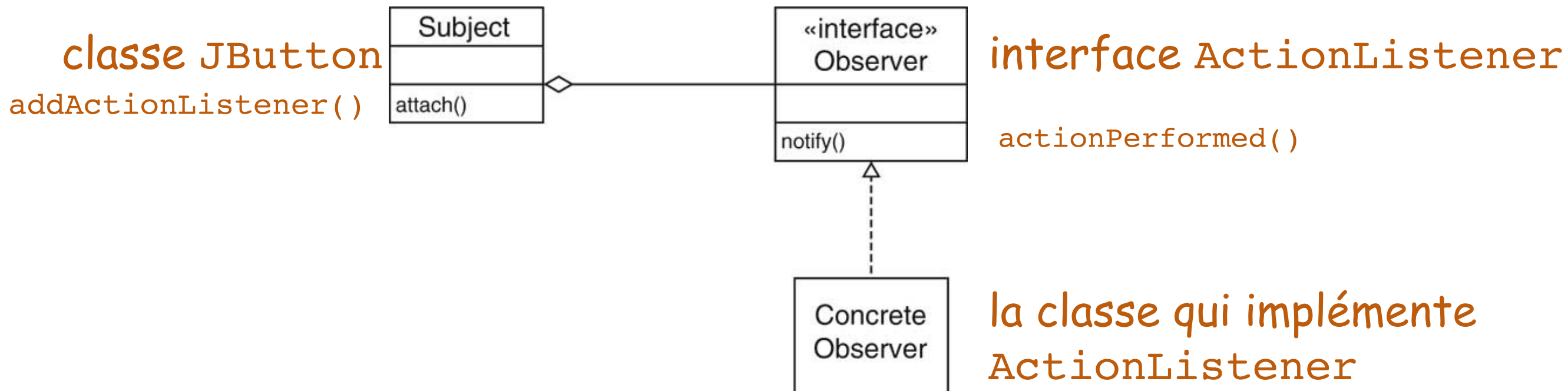
- Définir une interface "Observer". Tous les observateurs concrets l'implémentent.
- Le sujet maintient une collection d'observateurs.
- Le sujet fournit des méthodes pour attacher/ détacher des observateurs.
- Quand un événement a lieu, le sujet alerte tous les observateurs qui lui ont été attachés (invoque notify()).



# Observer Pattern



## Exemple



# Strategy Pattern

---



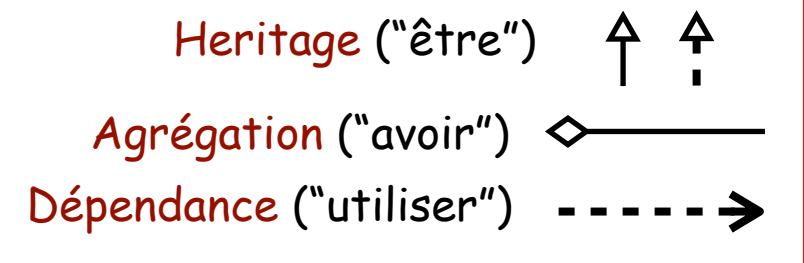
# Strategy Pattern

---

## □ Contexte

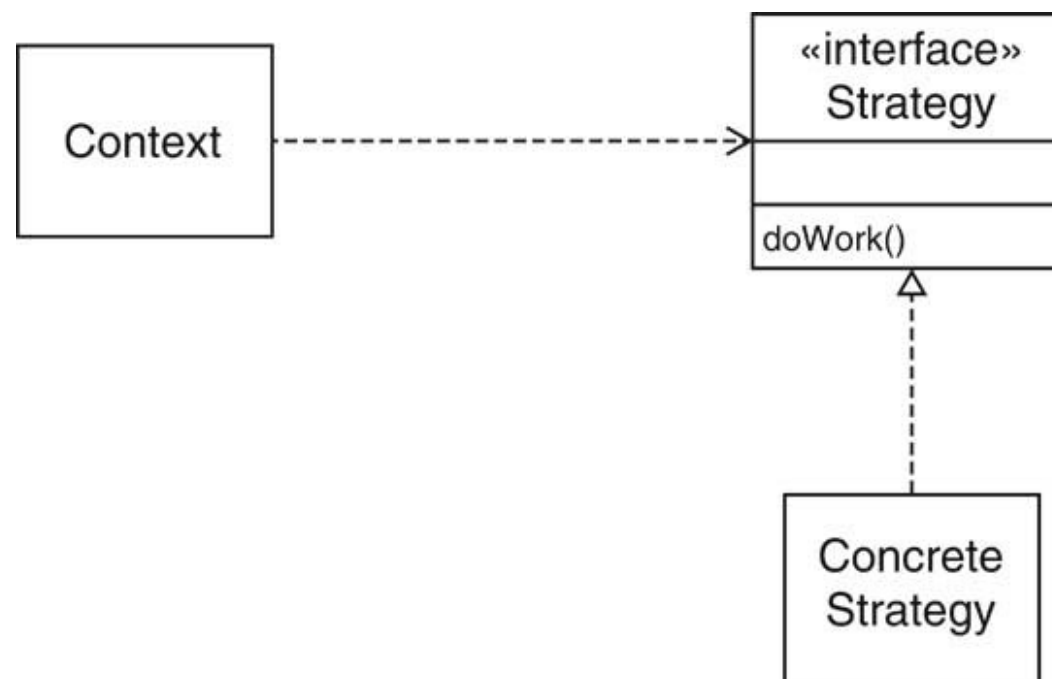
- Une classe Contexte peut exécuter une tâche en utilisant plusieurs variantes d'un algorithme  
(Exemple : Une classe SEL peut utiliser plusieurs algorithmes pour résoudre un système d'équations linéaires)
- Les clients de cette classe veulent pouvoir avoir le contrôle sur quelle variante de l'algorithme (i.e. quelle stratégie) est utilisée

# Strategy Pattern

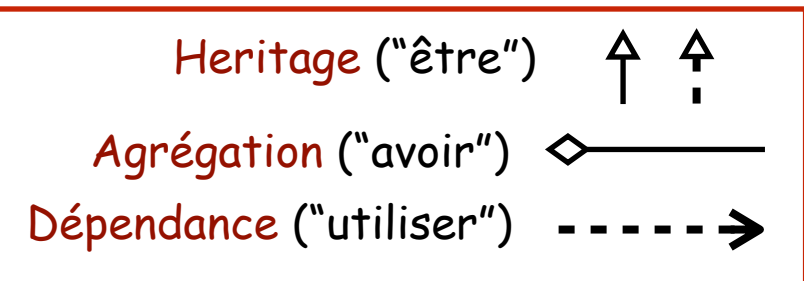


## □ Solution

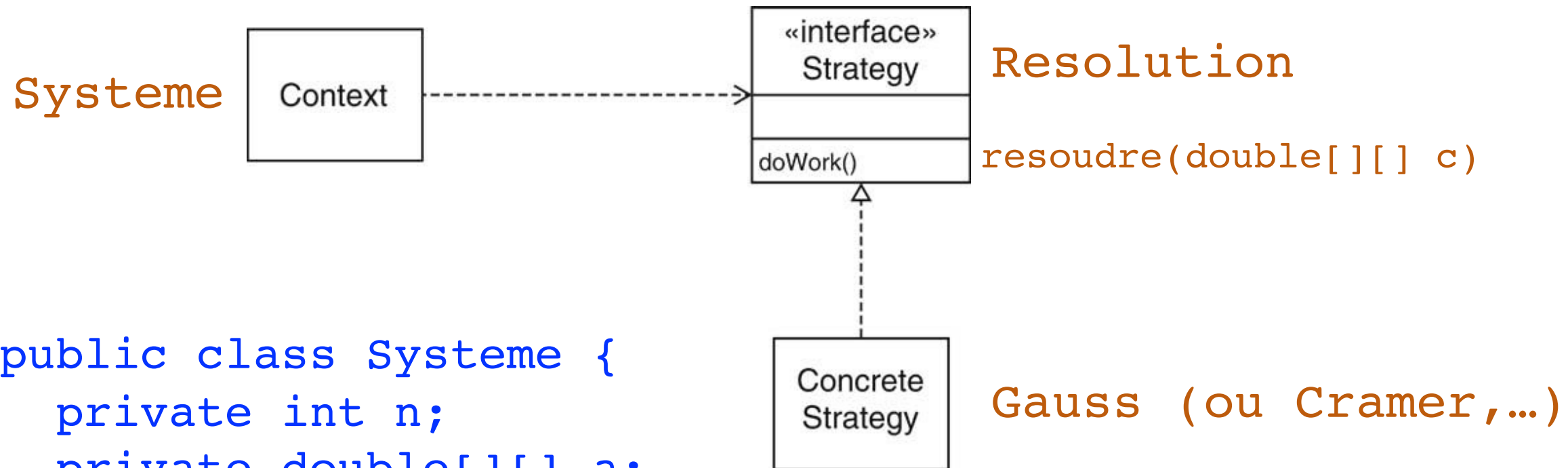
- Définir une interface Stratégie qui est une abstraction pour l'algorithme
- Pour définir une stratégie concrete, définir une classe qui implémente cette interface  
(Exemple : une classe Gauss, une classe Cramer, ...)
- Les clients de la classe Contexte peuvent lui fournir la stratégie en passant un objet d'une de ces classes Stratégie
- Quand la classe Contexte doit executer la tâche, elle invoque la méthode `doWork()` de l'objet stratégie (méthode disponible par l'interface Stratégie)



# Strategy Pattern

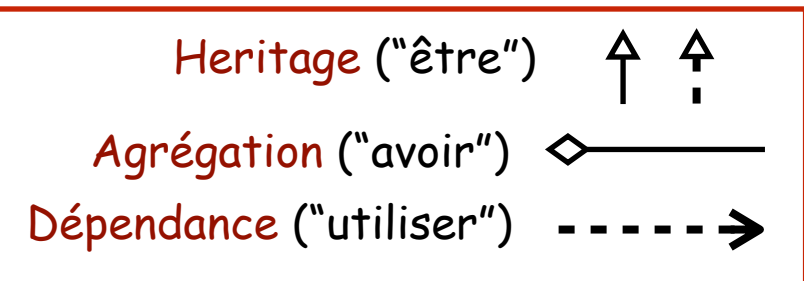


## Exemples

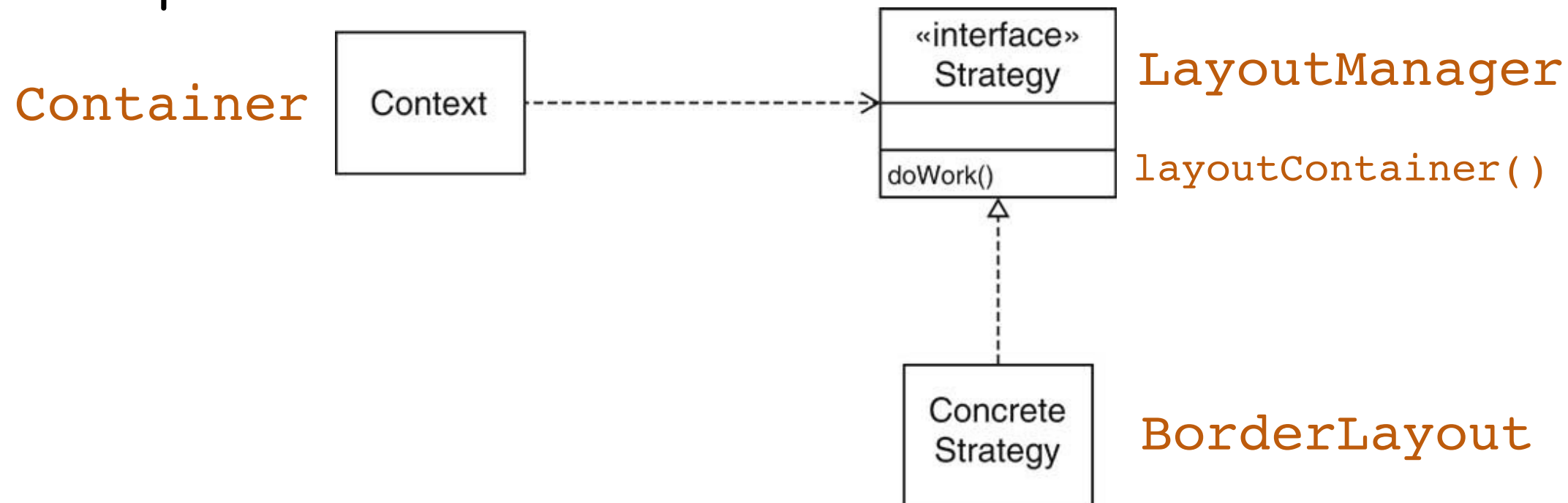


```
public class Systeme {
    private int n;
    private double[][] a;
    private double[] b;
    private Resolution strategie;
    ...
    public int getN() { return n; }
    public double[] getSolution() throws SystemeSingulier {
        ...//crée c en ajoutant la colonne b à la matrice a
        return strategie.resoudre(c);
    }
}
```

# Strategy Pattern



## Exemples



- La classe `Container` de AWT possède une référence à son `LayoutManager`
- Le client du `Container` peut lui passer un `Layout Manager` de son choix :

```
JPanel p = new JPanel(); p.setLayout(new BorderLayout());
```

- La méthode `doLayout()` de la classe `Container` invoque la méthode `layoutContainer()` de son `LayoutManager` pour disposer son contenu

# Composite Pattern

---

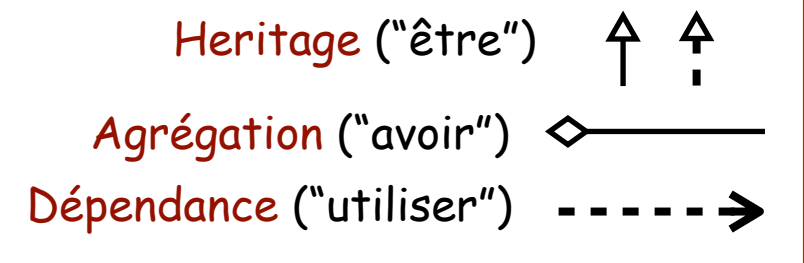
# Composite Pattern

---

## □ Context

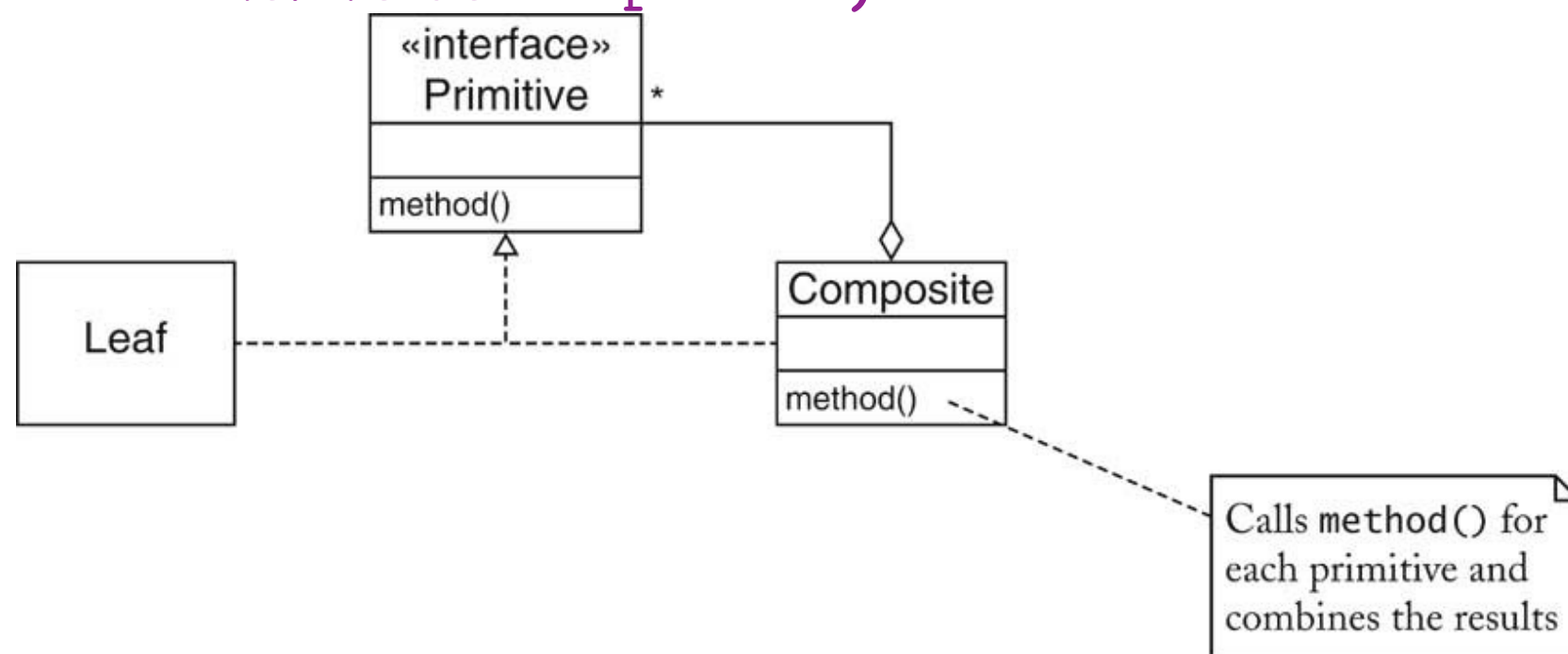
- Des objets primitifs peuvent être combinés pour former des objets composites (Ex. : plusieurs composants graphiques peuvent être ajoutés à un Container)
- En même temps un objet composite doit pouvoir être traité aussi comme un objet primitif (Ex : un Container peut être ajouté comme composant d' un autre Container)

# Composite Pattern



## □ Solution

- Définir une interface ou classe abstraite Primitive qui est une abstraction pour les objets primitifs (Ex. la classes Component de AWT)
- Les classes décrivant les objets primitifs implémentent cette interface (classes Leaf)
- Les objets composites contiennent une collection d'objets primitifs (Ex : les Container contiennent une collection de Component)
- La classe Composite implémente également l'interface Primitive (Ex. Container hérite de Component)

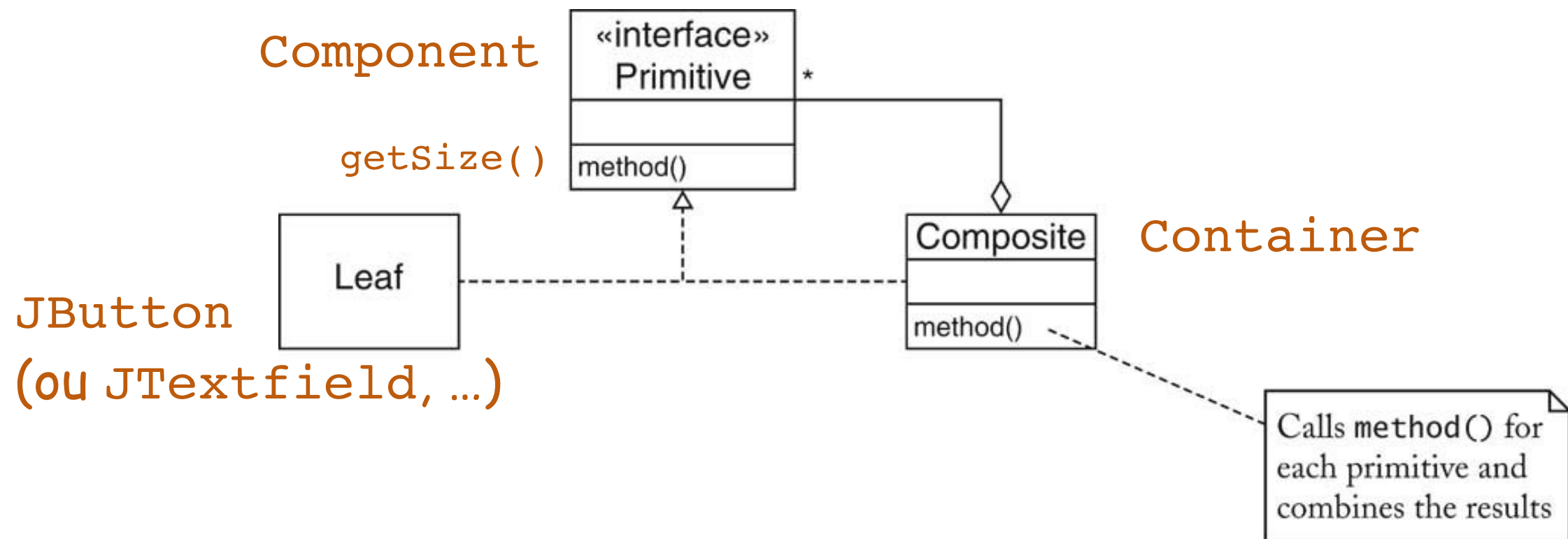


- Pour implémenter une méthode de l'interface primitive (Ex. `getSize()`) la classe Composite invoque la même méthode sur ses objets primitifs et combine les résultats

# Composite Pattern



## Exemple





# Un exemple complet de modélisation par design patterns

---

# Un exemple complet de modélisation par design patterns

---

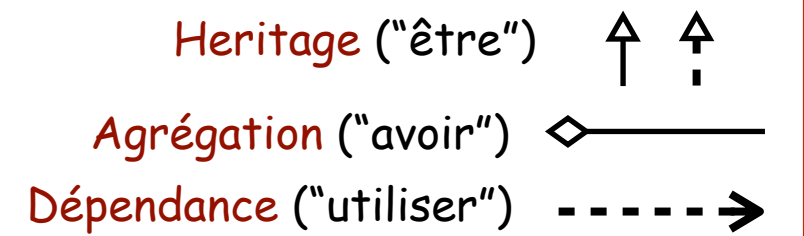
- But : modéliser les classes nécessaires pour définir et opérer avec des factures
  - Une facture a un ensemble d'items (ses lignes)
  - Chaque item a un prix
  - Chaque item est la description d'un produit facturé
  
- Modélisation naturelle par :
  - Une interface LineItem: **LineItem.java**
  - Une classe Product qui implémente cette interface : **Product.java**

# Une application de gestion de factures

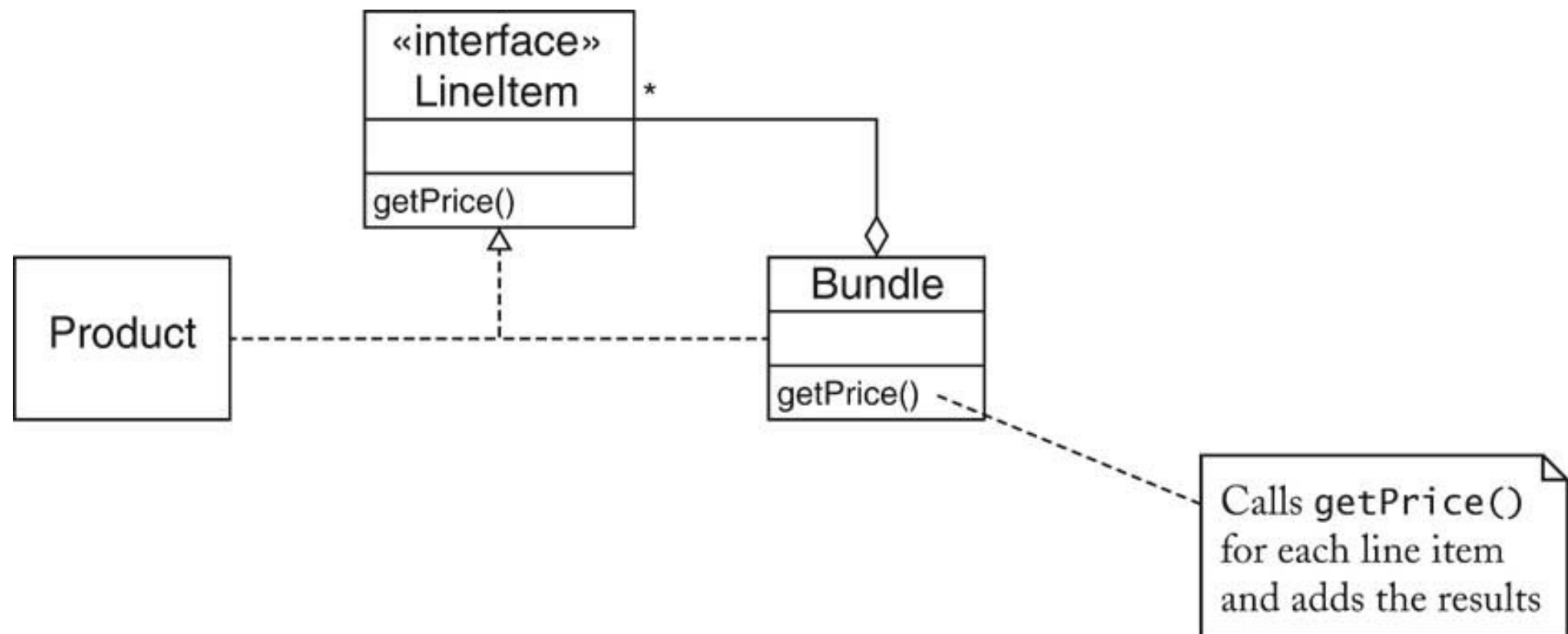
---

- Certains items sont des "packs" (qu'on appelle Bundle)
  - Ex : un pack : chaine hi-fi avec amplificateur + lecteur CD + enceintes
- Bundle = un ensemble de produits reliés avec une description et un prix
- Un bundle contient des items
- Un bundle aussi est un item
- On peut appliquer le COMPOSITE pattern

# Une application de gestion de factures



- **Modélisation des Bundles** par COMPOSITE pattern



- **Bundle.java** (regarder en particulier `getPrice()`)

# Une application de gestion de factures

---

## □ Modélisation des factures

- Une facture consiste en un ensemble de LineItem
- Elle doit être modifiable
- Classe Invoice :

```
public class Invoice {  
    private ArrayList<LineItem> items;  
    ...  
    public void addItem(LineItem item) {  
        items.add(item);  
    }  
    ...  
}
```

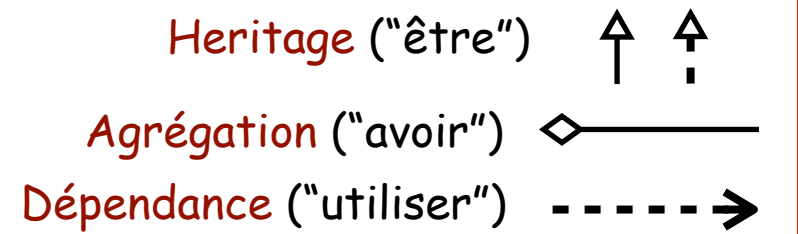
# Une application de gestion de factures

---

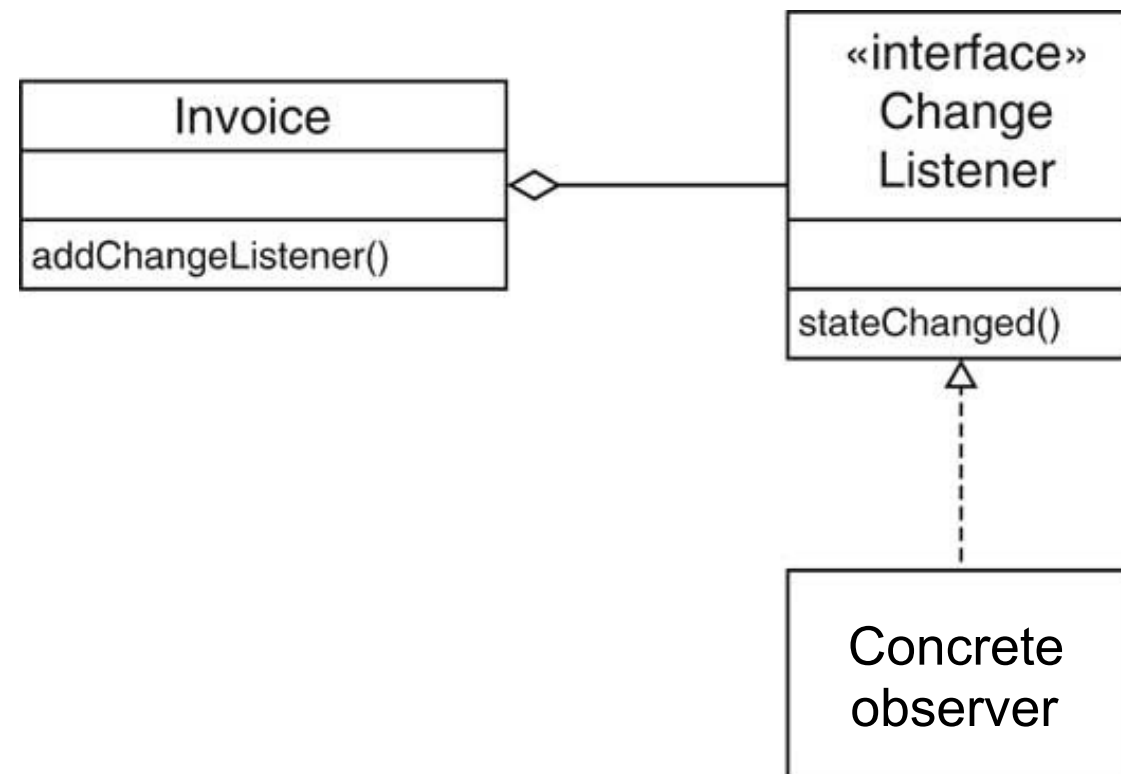
## □ Separation vue / modèle

- La GUI de notre programme affichera la facture dans un composant graphique (p.ex. un JTextArea)
- Nous voulons rendre le **modèle** (classe Invoice) indépendante des éléments de la GUI qui l'affichent (la **vue**)
- Cependant chaque changement dans la facture (ajout d'un item) doit être répercuté sur l'interface graphique
- OBSERVER pattern
  - La classe Invoice maintient une liste de "observateurs" (listeners) de ses changements
  - Quand un changement a lieu dans la facture, elle notifie ses observateurs

# Une application de gestion de factures



- **Separation vue / modèle**
  - **OBSERVER pattern**



(fournie  
par Swing)

Concrete Observer :

- Classe typiquement anonyme (ou expression lambda)
- Fait partie de l'interface graphique : accès au composant (JTextArea) qui affiche la facture
- Implementation de stateChaged() : fait un "refresh" de l'affichage de la facture dans le JTextArea

# Une application de gestion de factures

---

- **Separation vue / modèle : le code**
  - Invoice maintient une liste de ChangeListener

```
public class Invoice {  
    private ArrayList<LineItem> items;  
    private ArrayList<ChangeListener> listeners;  
    public void addChangeListener(ChangeListener listener){  
        listeners.add(listener);  
    }  
    ...  
}
```



# Une application de gestion de factures

- **Separation vue / modèle : le code**
  - Quand la facture change (ajout d'un item) elle notifie tous ses listeners :

```
public class Invoice {  
    ...  
    public void addItem(LineItem item) {  
        items.add(item);  
        // Notifier tous les listeners du changement  
        ChangeEvent event = new ChangeEvent(this);  
        for (ChangeListener listener : listeners)  
            listener.stateChanged(event);  
    }  
    ...  
}
```

# Une application de gestion de factures

- **Separation vue / modèle : le code**

- Dans l'interface graphique (InvoiceTester.java) :

```
// la facture a gérer
final Invoice invoice = new Invoice();
// Ce champ de texte affichera la facture
final JTextArea textArea = new JTextArea(20, 40);

// On enregistre un ChangeListener concret auprès de
la facture
invoice.addChangeListener(event ->
    textArea.setText(invoice.format(...)) );
```

- Remarque : le ChangeListener concret est donné par l'expression lambda

# Une application de gestion de factures

---

- **Itérer sur les items de la facture**
  - Le clients de la classe Invoice ont besoin de parcourir ses items (i.e. pour calculer le prix total, les afficher dans un format personnalisé, les stocker dans un fichier / BD...)
  - Pas souhaitable de donner accès à l'ArrayList<LineItem> (violation de l'encapsulation)
  - **ITERATOR pattern**
    - la classe Invoice fournit un itérateur sur ses LineItem (Iterator<LineItem>)
    - Les classes clientes l'utilisent pour parcourir les items de la facture

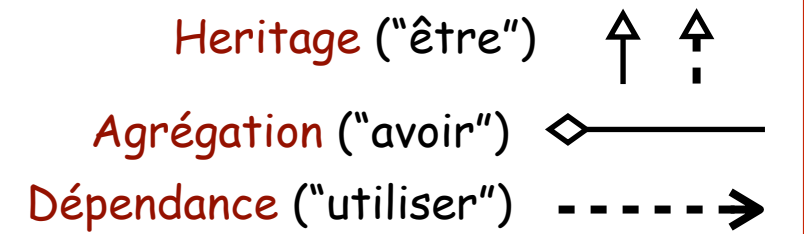
# Une application de gestion de factures

---

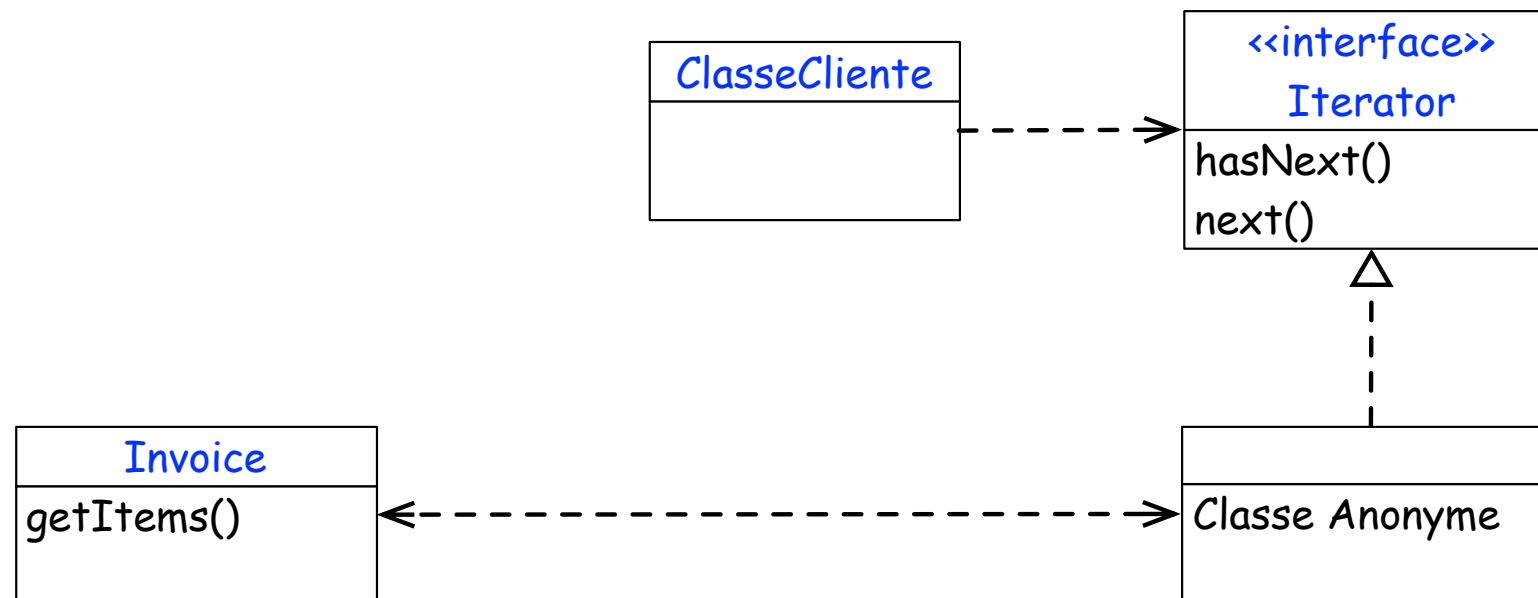
```
class Invoice {
    ...
    public Iterator<LineItem> getItems() {

        return new Iterator<LineItem>() {
            private int current = 0;
            public boolean hasNext(){
                return current < items.size();
            }
            public LineItem next() {
                return items.get(current++);
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
    ...
}
```

# Une application de gestion de factures



- Itérer sur les items de la facture avec un Iterator pattern

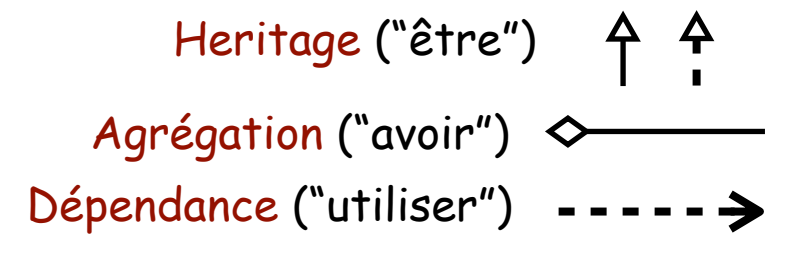


# Une application de gestion de factures

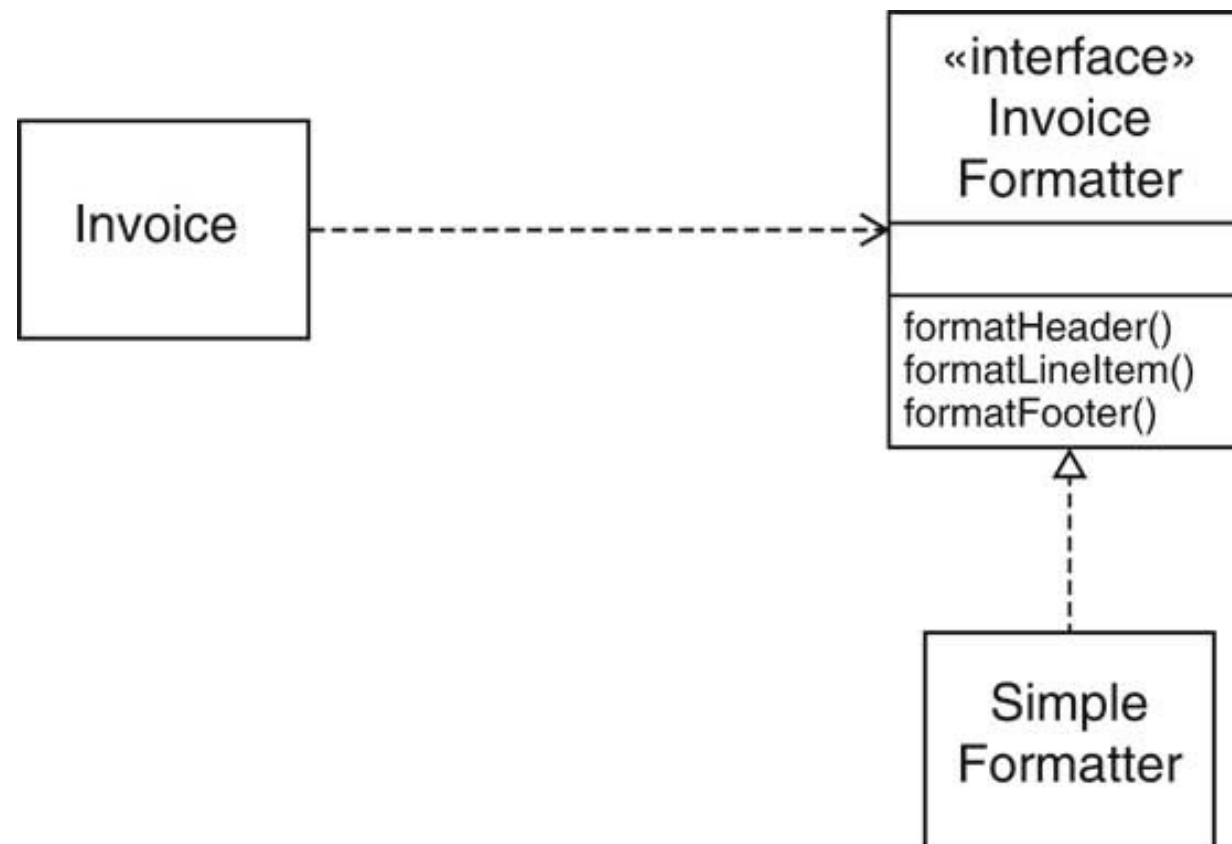
---

- **Formatter les factures**
  - La solution la plus simple : convertir en texte (méthode `toString()` de la classe `Invoice`), et afficher le texte dans un `TextArea`
  - Fige le formatage
    - Et si par exemple on voulait afficher la même facture dans une page Web (format HTML) ?
  - Nous voulons permettre different algorithmes de formatage
  - **STRATEGY** pattern

# Une application de gestion de factures



- Formater les factures avec le STRATEGY pattern



- InvoiceFormatter est une abstraction pour la stratégie de formatage des factures
- SimpleFormatter est une stratégie concrète (il peut y en avoir plusieurs)
- Invoice reçoit un InvoiceFormatter pour produire un formatage de son contenu
- Invoice n'a pas besoin de connaître la stratégie concrète de formatage

# Une application de gestion de factures

- **Formater les factures : le code**

- InvoiceFormatter.java

- SimpleFormatter.java

- //Invoice.java

```
class Invoice {  
    ...  
    public String format(InvoiceFormatter formatter) {  
        String r = formatter.formatHeader();  
        Iterator<LineItem> iter = getItems();  
        while (iter.hasNext())  
            r += formatter.formatLineItem(iter.next());  
        return r + formatter.formatFooter();  
    }  
    ...  
}
```



# Une application de gestion de factures

---

- Test de l'application
  - InvoiceTester.java
  - java InvoiceTester

# Autres design patterns

---

- Un grand nombre de design patterns ont été définis
- En plus de ceux discutés dans ce cours :
  - Decorator
  - MVC
  - Adapter
  - Command
  - Factory method
  - Proxy
  - Singletons
  - Visitor
  - ...et bien d'autres
- Ouvrage de reference :
  - Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, **Design Patterns. Elements of Reusable Object-Oriented Software** - Addison-Wesley Professional (1994)