2. Patrons de conception
Design Patterns

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1

Plan

Patrons (*Patterns*)

Rappel : CM1

- Maitrise des dépendances pour faciliter la réutilisation et ajouter de la souplesse à une application.
- Respect des concepts fondamentaux de COO (couplage faible, modularité, encapsulation, polymorphisme)
- Suivi des principes avancés de COO (principes SOLID)

- Ces principes constituent des préceptes globaux et assez abstraits
- Ces principes se retrouvent dans les différents types de Pattern

Patrons (*Patterns*)

Qu'est-ce qu'un patron (*pattern*) ?

- Un moyen d'atteindre un objectif
- Offre une **solution à un problème récurrent**, dans un contexte donné.
- Est fondamentalement destiné à être imité.

Origine

- Inspiré des travaux de l'architecte en bâtiments Christopher Alexander dans les années 70 (livre *A Pattern Language*)
- Premier auteur à **compiler les meilleures pratiques d'un métier en documentant ses modèles**.
- Forte influence sur la communauté logicielle

Catégories de Patrons (*Patterns*)

Catégories de patrons

- Patrons de conception (*Design Patterns*) : arrangement caractéristique de modules ou de classes.
- Patrons d'architecture (*Architectural Patterns*) : schémas d'organisation structurelle (en packages) de logiciels (Client-Serveur, MVC, ...)
- Anti-patrons (de conception, de test, de projet) : solutions évidentes mais **fausses** à des problèmes récurrents ; ou comment sortir d'une mauvaise solution
- Patrons d'organisation : Schémas d'organisation de tout ce qui entoure le développement d'un logiciel (humains)

Patrons de conception : Définition

- Arrangement caractéristique de modules ou de classes
- Solution reconnue comme **bonne pratique en réponse à un problème récurrent de conception orientée objet**
- Formulé de manière abstraite et indépendante du contexte ou de l'application :
  - Structuration détaillée en classes
  - Détaille attributs et opérations des classes
  - Détaille relations entre classes : délégation, héritage, réalisation, ...
- *Un groupe d'objets coopérants liés par des relations et des règles qui expriment les liens entre un contexte, un problème de conception et sa solution* [Aarsten96]

Patrons de conception : Objectifs

- **Capitaliser** l'expérience : réutilisation de solutions qui ont prouvé leur efficacité
- **Concevoir** de bons modèles orientés objet : faciles à comprendre, à mettre en oeuvre, à maintenir et à réutiliser
  - Meilleure qualité grâce à un niveau d'abstraction plus élevé
  - Réduction du nombre d'erreurs, d'autant que les patrons sont examinés avec attention avant d'être publiés
- Définir **un vocabulaire commun** entre l'architecte et le programmeur
  - Communication plus aisée
  - Ecrire du code facilement compréhensible par les autres
- Accélérer l'**apprentissage** des concepts OO en suivant des exemples respectant les principes de COO
- Moyen de **documentation** de logiciels

Patrons de conception

Patrons GRASP

- *General Responsability Assignment Software Patterns*, Larman (2005)
- 9 principes fondamentaux de conception OO plutôt que des patrons
- Concernent **l'affectation des responsabilités aux classes** :
  - Faible Couplage : Le but est de minimiser les dépendances, réduire l'impact des modifications et faciliter la réutilisation. La solution est d'affecter les responsabilités de manière à minimiser le couplage.
  - Créateur : Déterminer quelle classe a la responsabilité de créer des instances d'une autre classe.
  - Forte cohésion : Avoir des sous-classes terminales très spécialisées.
  - Indirection : Découpler des classes en utilisant une classe intermédiaire.
  - ...
- Non détaillés dans ce cours ...

Patrons de conception

≪ patrons GoF ≫

- GoF = *Gang of Four* : Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- Ouvrage de référence intitulé *Design Patterns – Elements of Reusable Object-Oriented Software* en 1995
- Description de 23 patrons de conception : les plus connus.
- **Recueil** (identification et formalisation) des meilleures solutions de la communauté du développement OO pour résoudre des problèmes récurrents
- Notions plus avancées et détaillées que GRASP

Patrons de conception GoF

Chaque patron de conception résout un problème dans un certain contexte.

Formalisme d'un patron de conception

- Nom
- But : Description du problème à résoudre (sujet à traiter et contexte)
- Moyen : Description de la solution (le patron de conception)
  - Détaille les différents éléments de la solution
    $\rightarrow$ Utilisation de diagrammes UML
  - Détaille les relations entre composants et leurs **rôles** dans la résolution du problème.
    $\rightarrow$ par ex. le patron *Observer* implique deux rôles qui sont le sujet et l'observateur
- Conséquences : effets résultants de la mise en oeuvre du patron (complexité en temps/mémoire, impact sur flexibilité, portabilité, ...)

Patrons de conception GoF

Trois familles

- structuraux : Concevoir des structures de classes séparant l'interface de l'implémentation :
  - Adaptateur (*Adapter*), Composite, Façade, Décorateur (*Decorator*), Pont (*Bridge*), Proxy.
- de construction/création : Décrivent des processus de création d'objets :
  - Singleton, Fabrique (*Factory*), Abstract Factory, Prototype, Builder
- comportementaux : Spécifient comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités).
  - Itérateur (*Iterator*), Observateur/Observé (*Observer*), Stratégie (*Strategy*), Chaine de responsabilité, Patron de méthode (*Template method*), Etat, Interpréteur, Commande, Médiateur, Mémento, Visiteur

Plan

Patterns structuraux

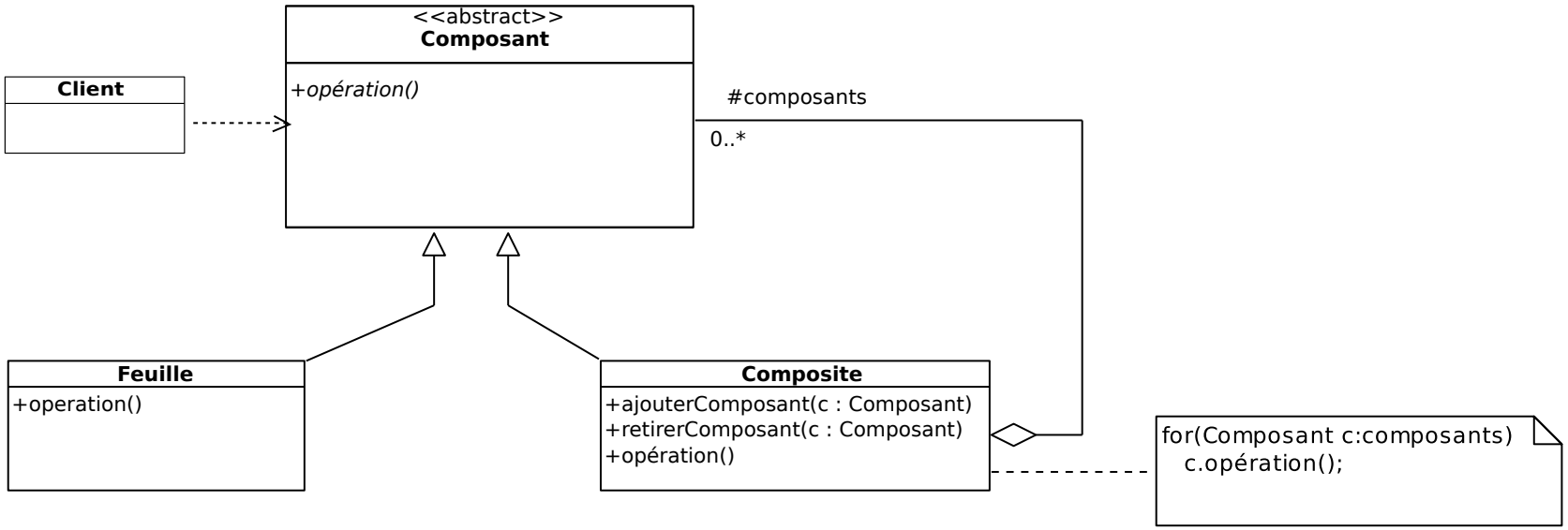
Comment organiser les classes d'un programme pour former une structure plus large (séparant l'interface de l'implémentation) ?

- **Encapsulation** de la composition des objets
- Augmentation du niveau d'**abstraction** du système
- Mise en avant des **interfaces**

Plan

Composite (*Composite*)

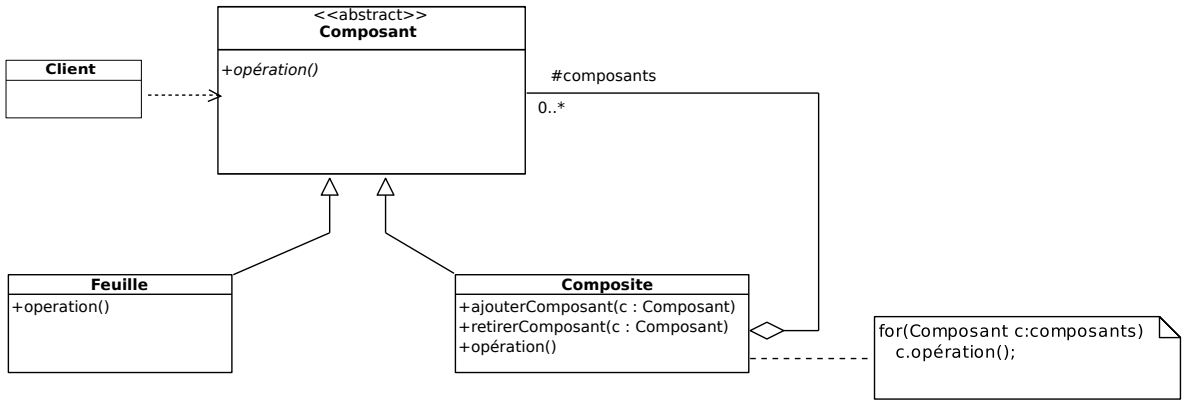
- Contexte : Structure arborescente d'objets
- Buts :
  - Hiérarchies composant/composite avec ajout simple de nouveaux composants
  - Le client traite de manière indifférente des objets atomiques ou des ensembles d'objets
- Moyen : Interface commune pour éléments individuels et composites (`Composant`)



Composite (*Composite*)

Conclusion

- S'il est nécessaire de représenter au sein d'un système des hiérarchies de composition, vous pouvez appliquer le pattern Composite.
- Si les clients d'une composition doivent **ignorer s'ils communiquent avec des objets composés ou non**, à travers le pattern Composite, ils vont traiter tous les objets uniformément.
- Cadre générique de représentation d'une composition d'objets de profondeur variable qui peut-être vue comme un arbre.



Composite (*Composite*)

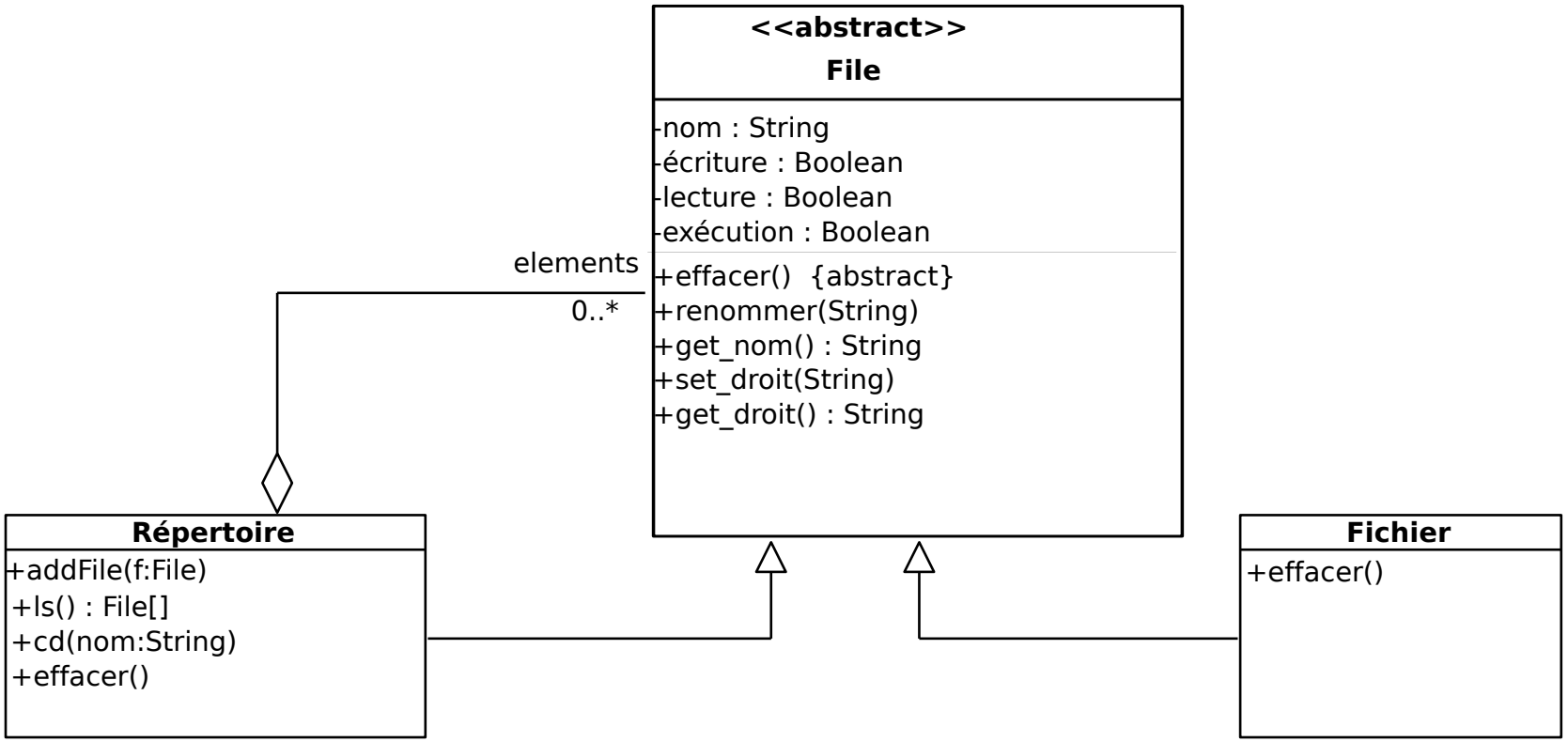
Exercice : Arborescence de répertoires

- Un répertoire et un fichier possèdent tous deux un nom et des droits de lecture, d'exécution et d'écriture.
- Un répertoire peut contenir des répertoires et des fichiers.
- Les fichiers et les répertoires possèdent une opération pour l'effacement et le renommage. Un répertoire possède une opération permettant de lister son contenu (*ls()*) et une opération permettant de se rendre dans l'un de ses sous-répertoires en précisant son nom (*cd()*).

Modéliser à l'aide d'un diagramme de classes cette arborescence de répertoires. Donner les diagrammes de séquence lorsqu'un client efface un fichier et efface un répertoire.

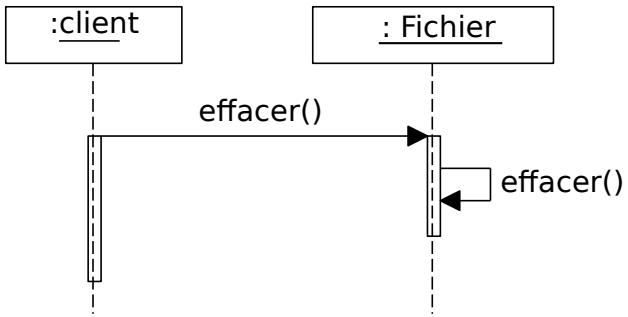
Composite (*Composite*)

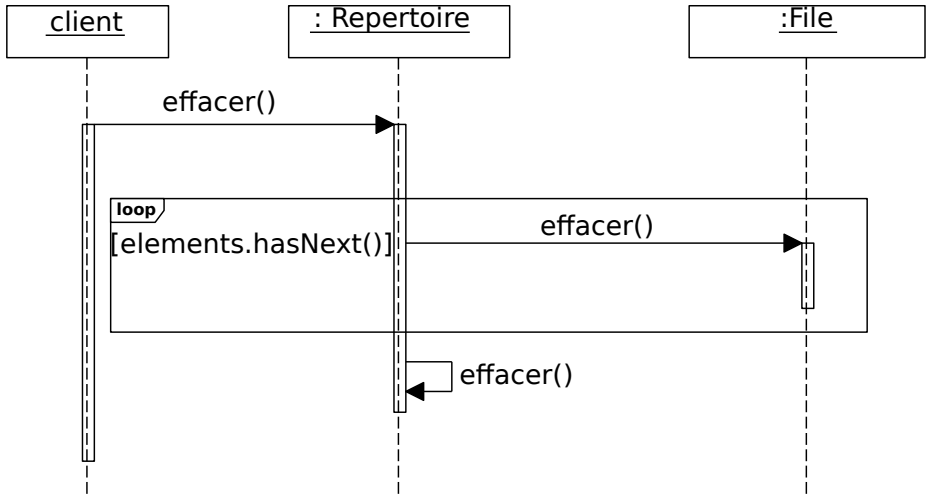
Exercice : Arborescence de répertoires



Composite (*Composite*)

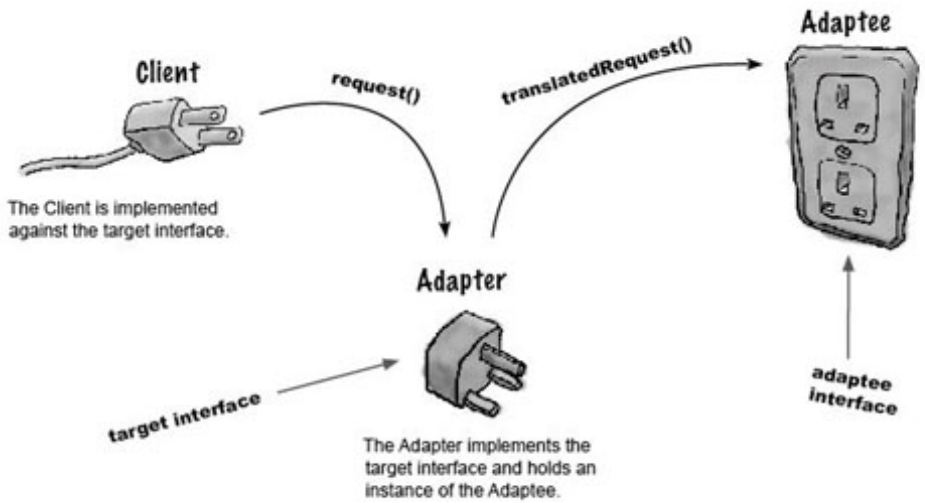
Exercice : Arborescence de répertoires





Plan

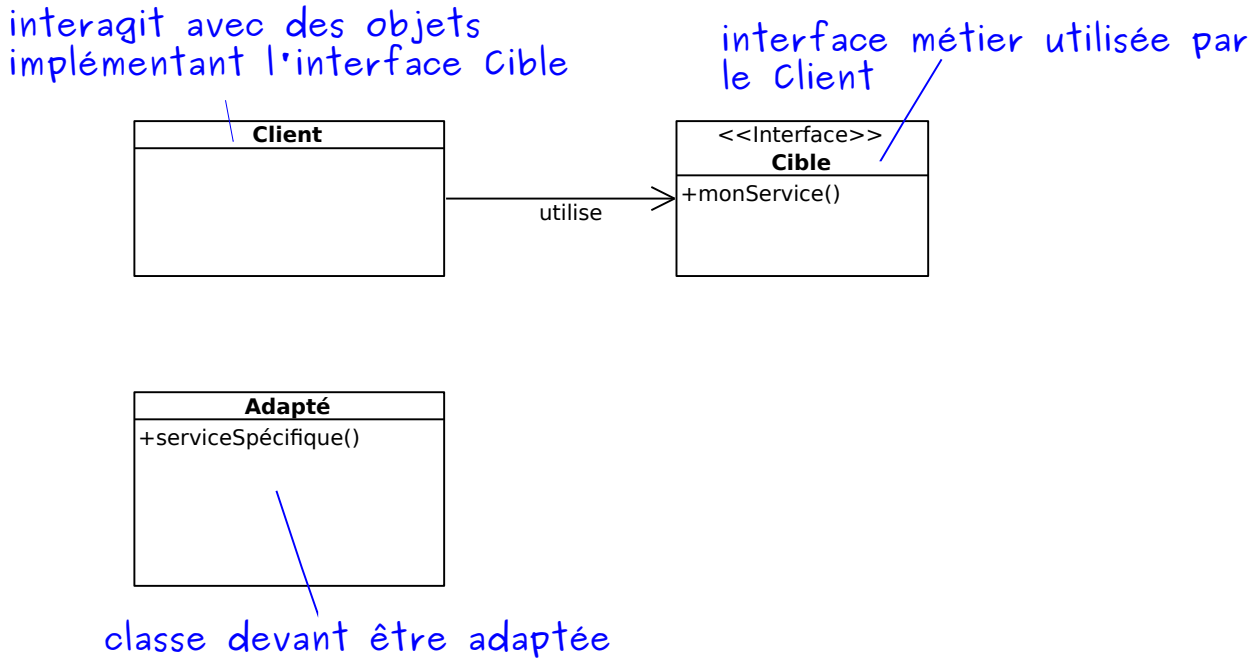
Adaptateur (*Adapter*)



- Contexte : Correspondance d'interfaces
- But :
  - Faire collaborer des classes dont les interfaces sont incompatibles.
  - Unifier une interface requise par un "client" et une classe fournissant les services attendus.
  - **Fournir une interface stable** à un composant dont l'interface peut varier.

Adaptateur (*Adapter*)

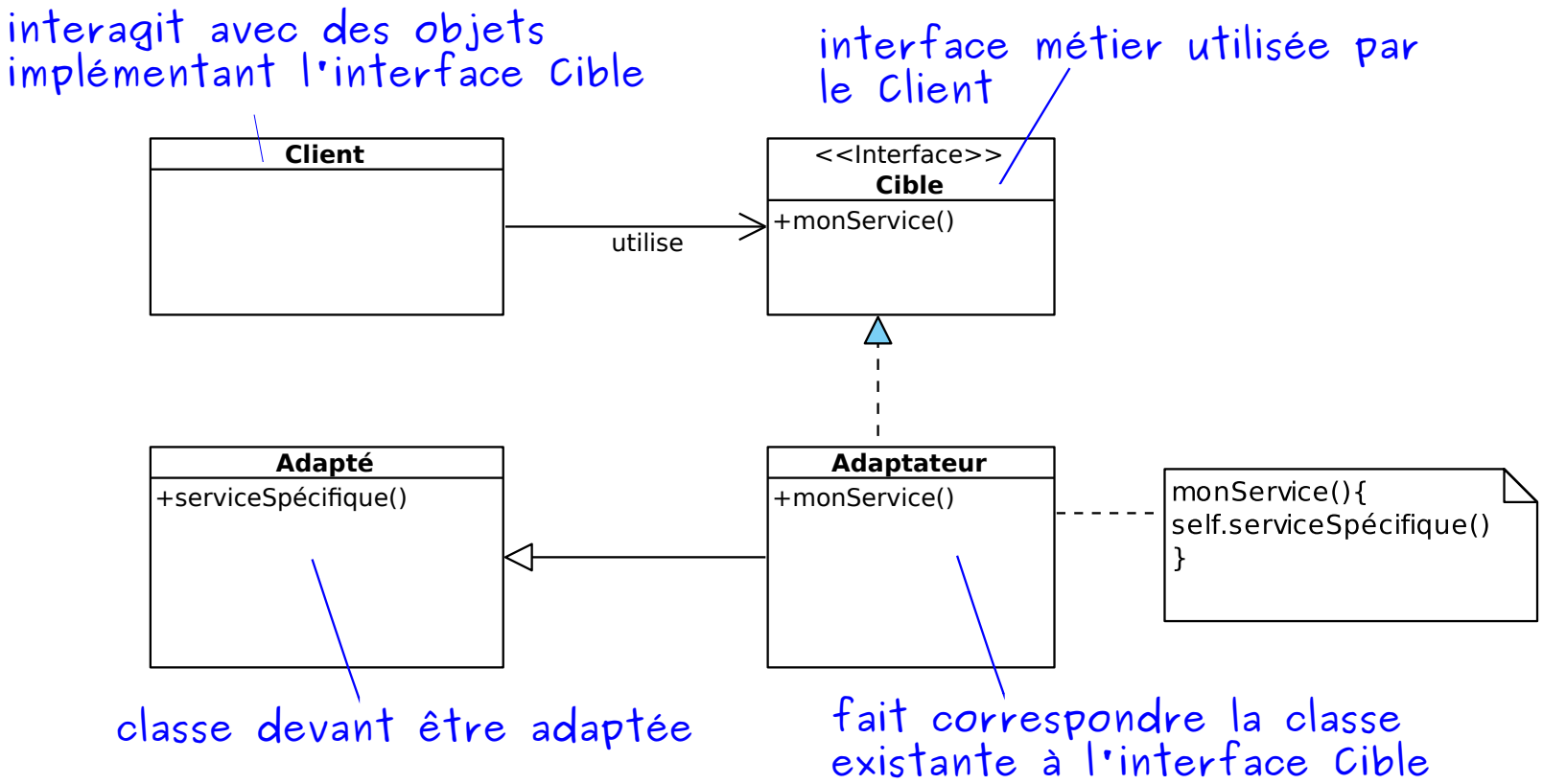
- Le client appelle `monService()`
- La méthode `serviceSpecifique()` de `Adapté` satisfait la requête.
- 2 types d'**Adaptateur** pour adapter `Adapté` à l'interface `Cible` requise par le client



Adaptateur (*Adapter*)

Moyen : Adaptateur de classe avec héritage

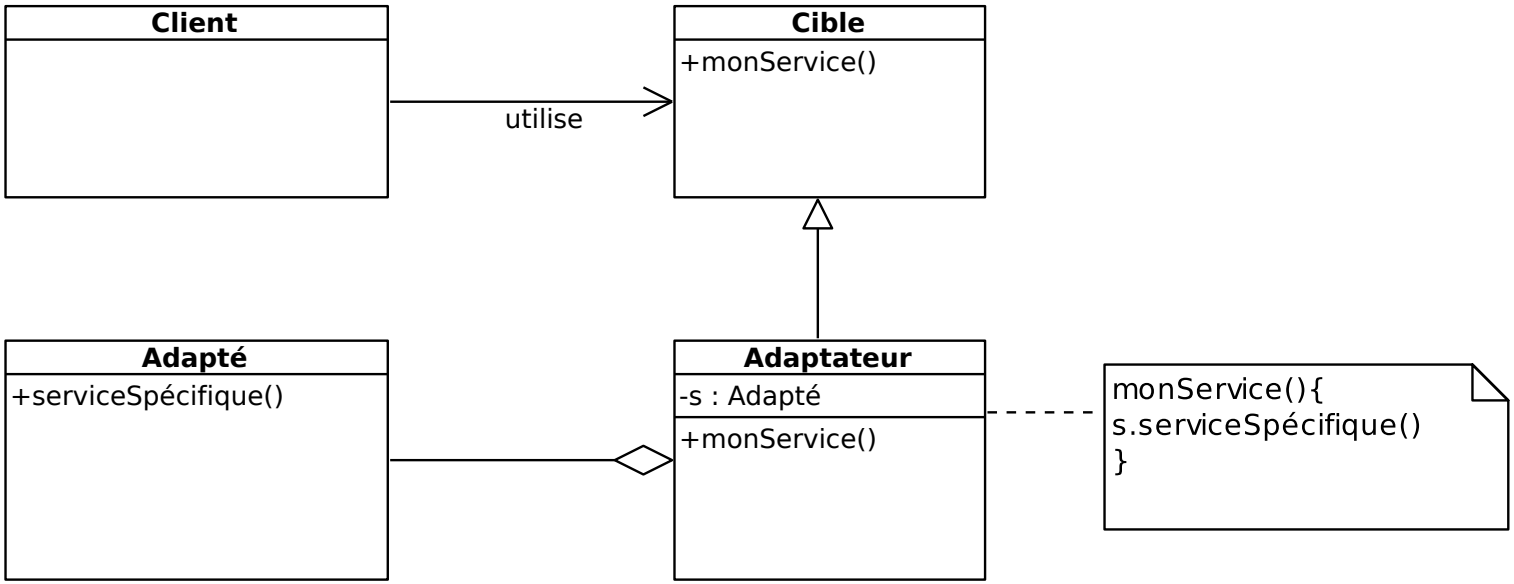
Fournir une nouvelle classe qui hérite de la classe existante (`Adapté`) et qui implémente l'interface requise (`Cible`) tout en utilisant les méthodes de la super-classe. L'adaptateur de classe traduit les appels du client en appels des méthodes de la classe existante.



Adaptateur (*Adapter*)

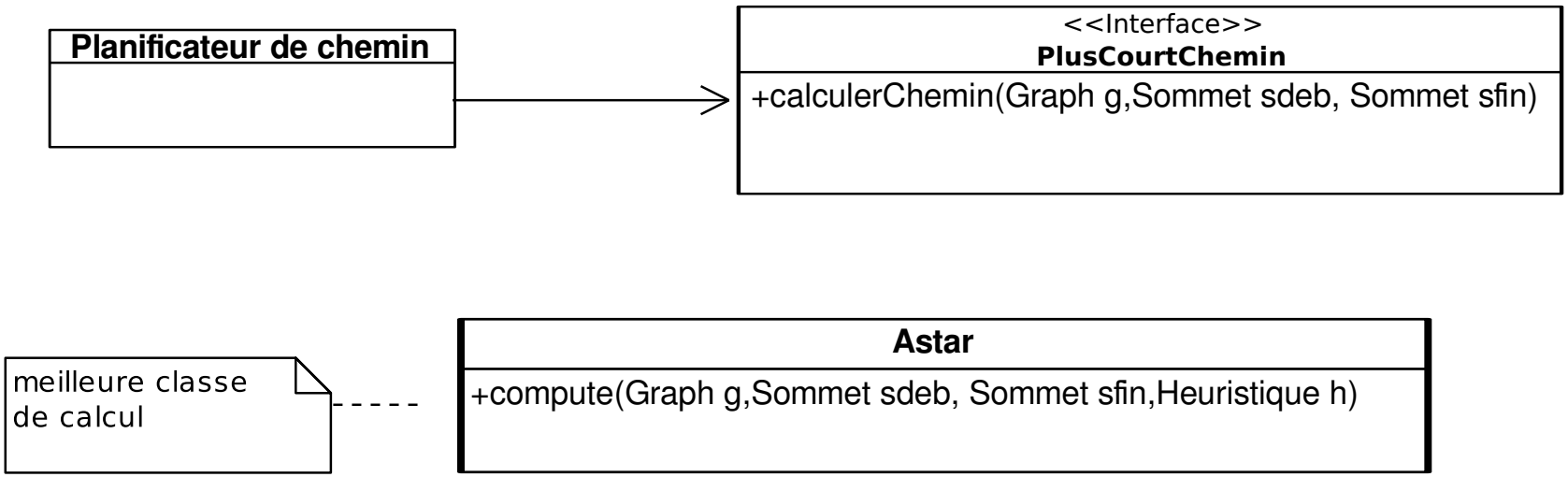
Moyen : Adaptateur d'objet avec délégation

Créer une sous-classe cliente qui hérite de la classe cible et qui utilise une instance de la classe existante par délégation.



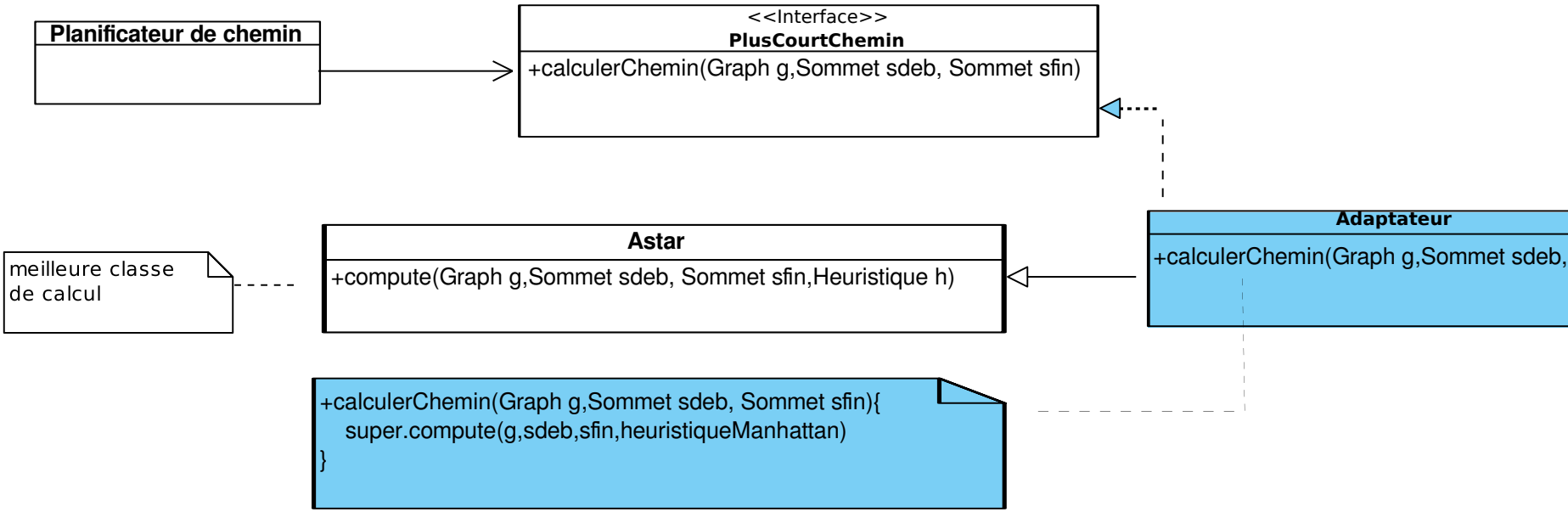
Adaptateur (*Adapter*)

Exemple



Adaptateur (*Adapter*)

Exemple



Adaptateur de classe

Adaptateur (*Adapter*)

Conclusion

Retenez la différence :

- un adaptateur de classe étend une classe existante et implémente une interface cible ;
- un adaptateur d'objet étend une classe cible et délègue à une classe existante.

Application des principes :

- d'inversion des dépendances : Limiter le couplage et protéger des variations en ajoutant des objets intermédiaires
- de protection des variations : Identifier les points de variation et d'évolution, et séparer ces aspects de ceux qui demeurent constants

Adaptateur (*Adapter*)

Exercice

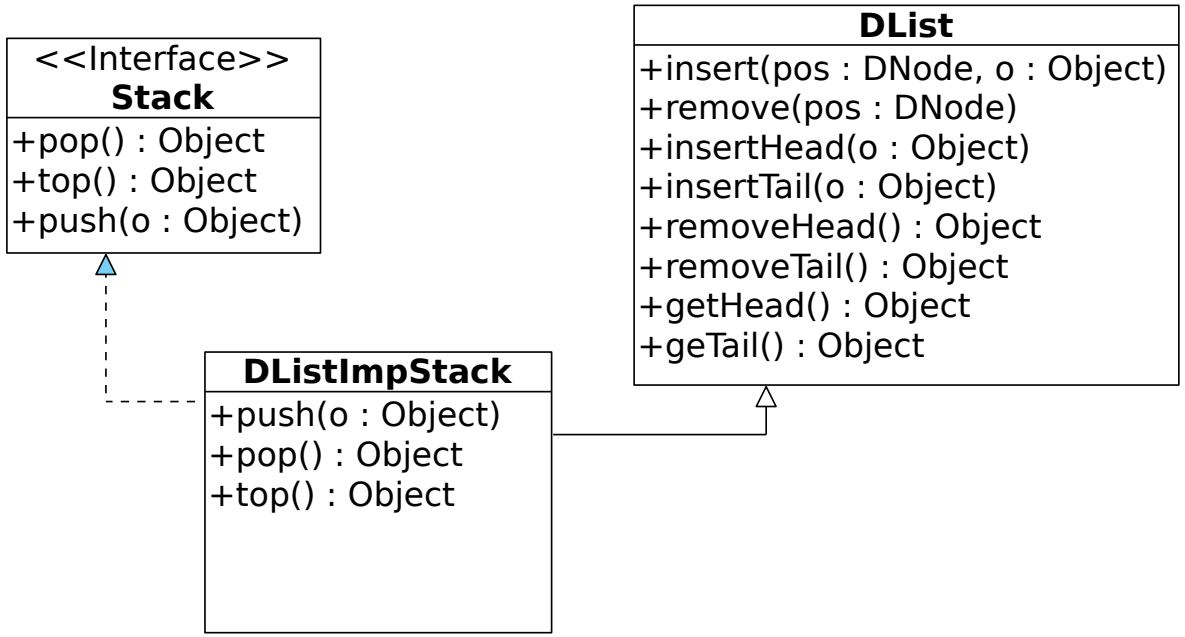
Comment adapter une liste doublement chaînée `DList` en une pile `Stack` ?

```
//interface cible utilise par le Client
interface Stack {
 void push(Object o);
 Object pop();//enleve et renvoie l'element en haut de la pile
 Object top();// renvoie l'element en haut de la pile
}
//classe a adapter: Liste doublement chainee
class DList {
   public void insert (DNode pos, Object o) { ... }
   public void remove (DNode pos) { ... }
   public void insertHead (Object o) { ... }
   public void insertTail (Object o) { ... }
   public Object removeHead () { ... }
   public Object removeTail () { ... }
   public Object getHead () { ... }
   public Object getTail () { ... }
}
```

Utiliser le pattern Adaptateur pour adapter la classe `DList` à l'interface `Stack`. Donner le diagramme UML et spécifier l'implémentation de la nouvelle classe.

Adaptateur (*Adapter*)

Exercice : Adaptateur de classe

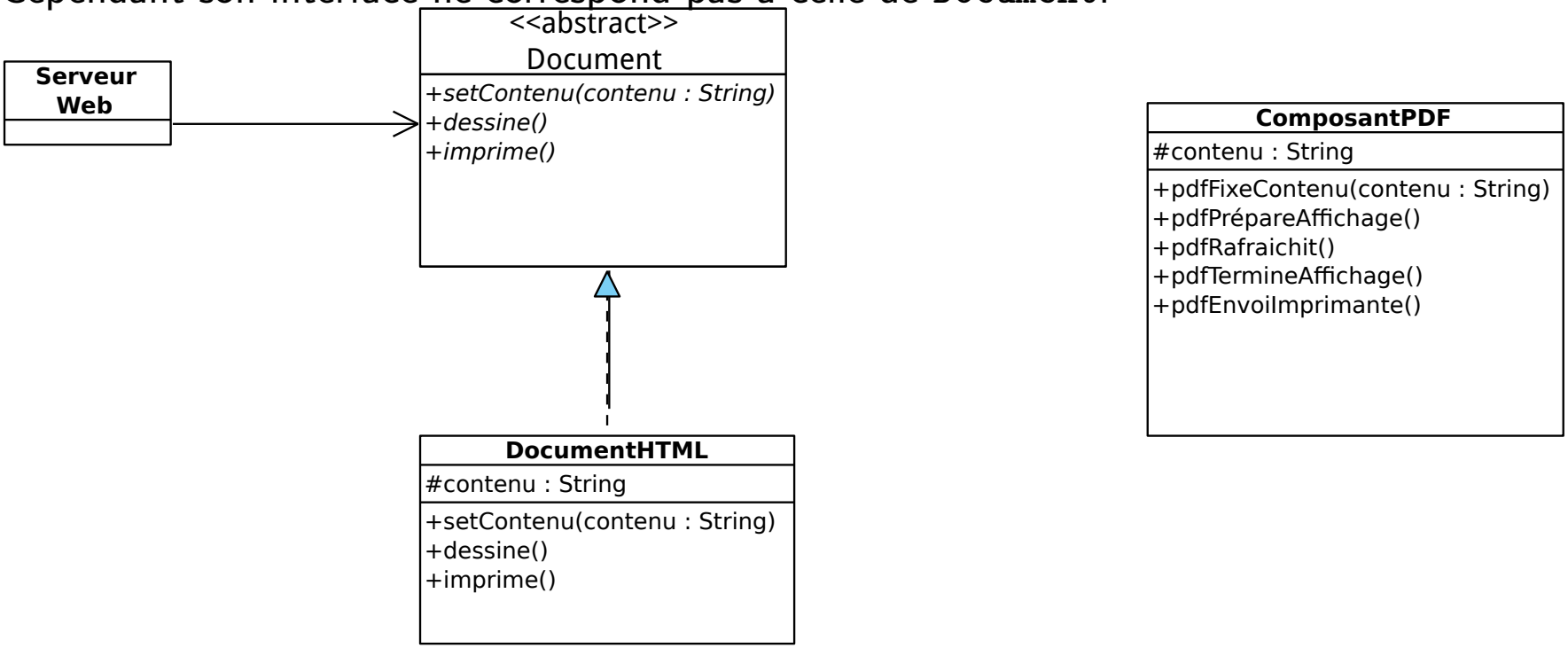



```
/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack {
 public void push(Object o) {
  insertTail(o);
 }
 public Object pop() {
  return removeTail();
 }
 public Object top() {
  return getTail();
 }
```

Adaptateur (*Adapter*)

Exercice

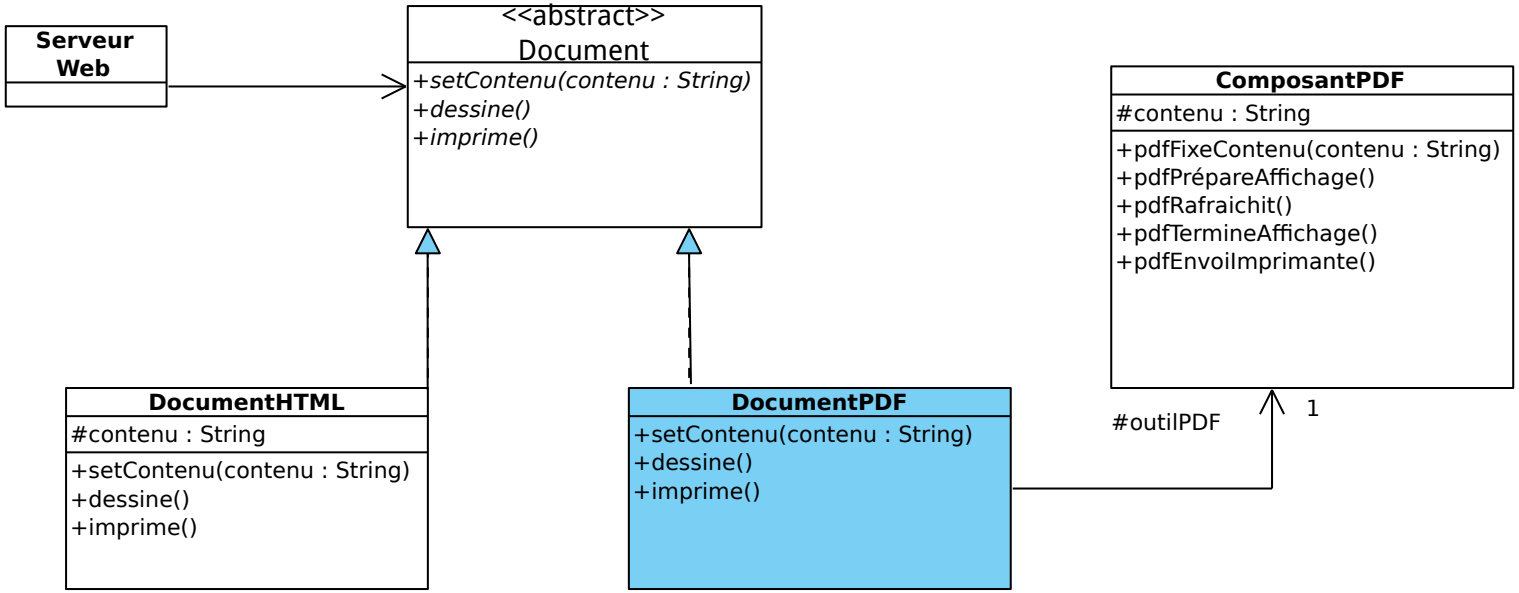
Un serveur web gère des documents destinés aux clients en utilisant la classe abstraite `Document`. Une première implémentation de cette interface a été réalisée : `DocumentHTML`. Pour l'ajout de documents PDF, un composant du marché **non-modifiable** a été choisi et doit être intégré à l'application (classe `ComposantPDF`). Cependant son interface ne correspond pas à celle de `Document`.



Utiliser le pattern Adaptateur pour que le serveur Web puisse gérer des documents HTML et PDF. Spécifier l'implémentation de la nouvelle classe.

Adaptateur (*Adapter*)

Exercice



Adaptateur d'objet

```
public Class DocumentPdf extends Document {
   protected ComposantPdf outilPdf = new ComposantPdf();
   public void setContenu(String contenu){
     outilPdf.pdfFixeContenu(contenu);}
   public void dessine(){
     outilPdf.pdfPrepareAffichage();
     outilPdf.pdfRafraichit();
     outilPdf.pdfTermineAffichage();}
   public void Imprime(){
     outilPdf.pdfEnvoieImprimante();}
    }
```

Plan

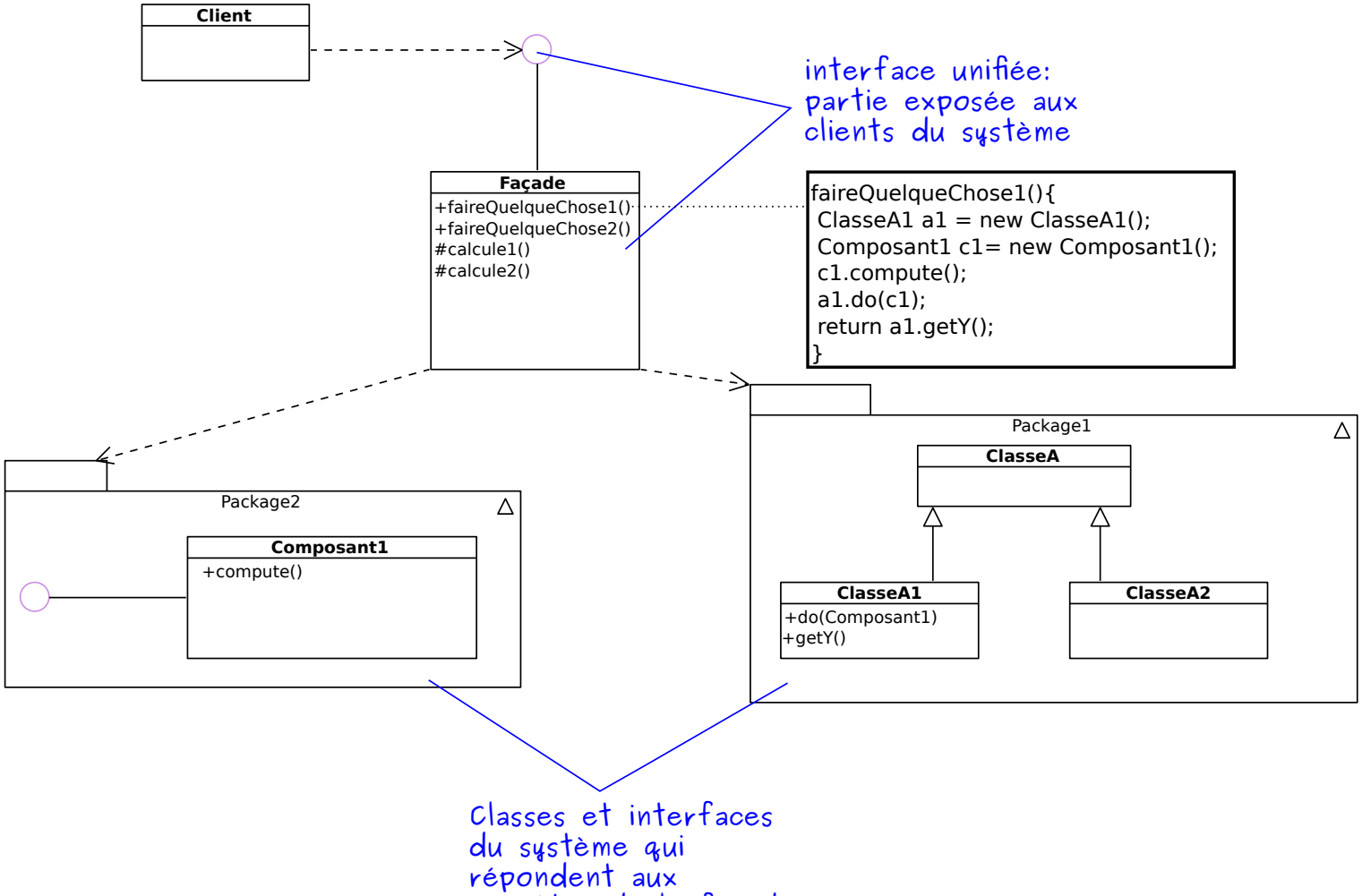
Façade (Facade)

- Contexte : Un sous-système est fait de plusieurs interfaces différentes compliquant son utilisation
- Buts :
  - **Fournir une interface simple** à un sous-système complexe
  - Cacher une conception et une interface complexe, difficile à comprendre
  - Permettre à un client d'utiliser de façon simple et transparente un ensemble de classes/packages qui collaborent pour réaliser une tâche courante.

Façade (Facade)

Moyen

Une classe fournit une façade en proposant des méthodes qui réalisent les tâches usuelles en utilisant les autres classes.

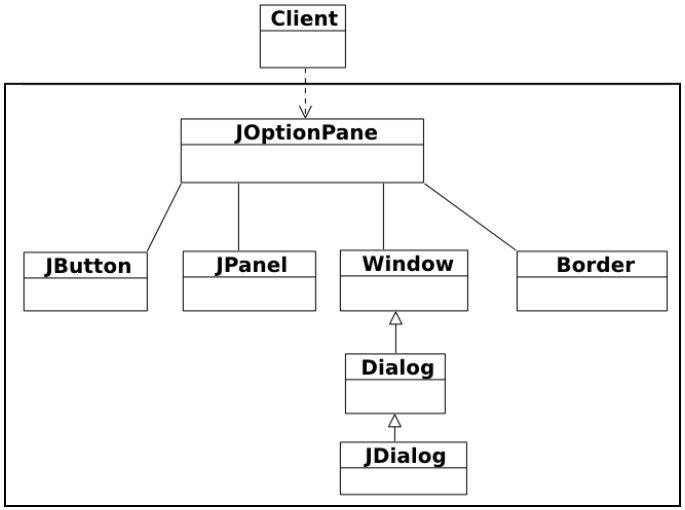


Façade (Facade)

Exemple tiré de Java

La classe `javax.swing.JOptionPane` est une façade qui masque l'ensemble des composants constituant une fenêtre de dialogue et fournit une interface simple au client pour manipuler l'ensemble.

Structure et exemple de `JOptionPane`



Façade (Facade)
Conclusion

- **Regroupe dans une abstraction unique (simplifiée)**, des fonctionnalités spécifiques à plusieurs sous-systèmes, sans empêcher de les utiliser directement (*(make simple tasks simple, complex tasks possible)*)
- Rend un ensemble de classes/packages plus lisible, facile à utiliser, comprendre et tester ;
- Assainit/Clarifie une API que l'on ne peut pas modifier

Adaptateur vs. Façade

- Un adaptateur est utilisé lorsque l'on doit respecter une interface bien définie : **convertit** une interface en une autre
- La façade est utilisée pour **simplifier** l'utilisation d'un système : fournit une interface simplifiée

Plan

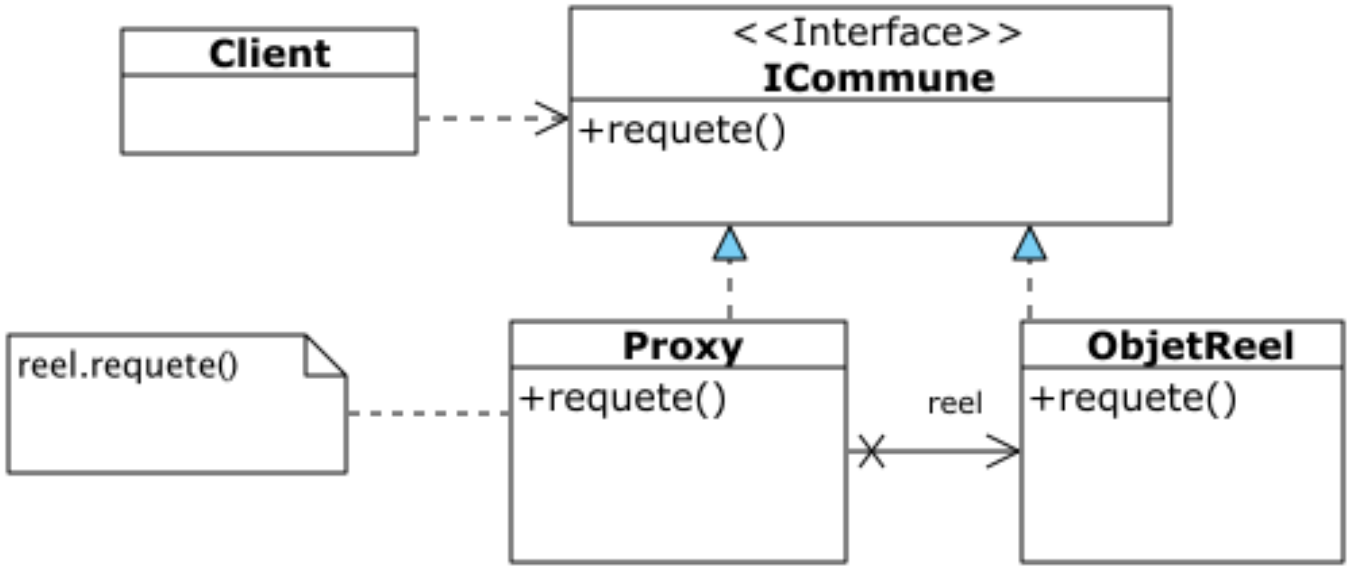
Pattern Proxy

- But : Accès indirect à un objet en **fournissant un remplaçant** qui **contrôle l'accès** à un autre objet, qui peut être distant, coûteux à créer ou qui doit être sécurisé
  - proxy distant : le proxy se charge de l'accès distant,
  - proxy virtuel : le proxy se charge de créer l'objet au plus tard (*lazy instantiation*), sert de substitut à l'objet avant sa création
  - proxy sécurisé : le proxy se charge de contrôler l'accès

Pattern Proxy

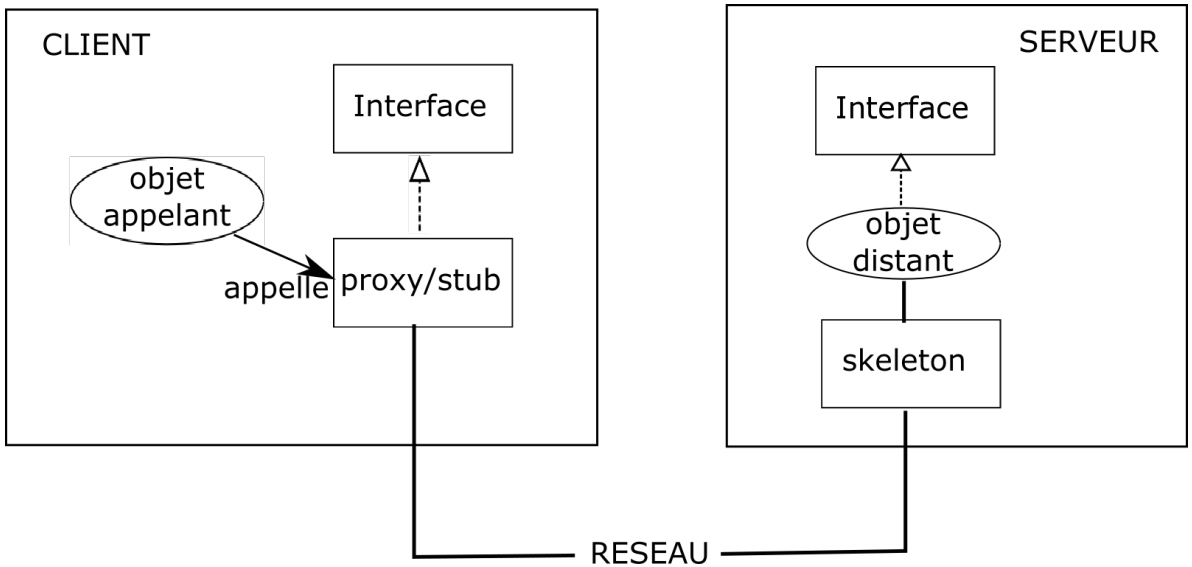
Moyen

- Interface commune entre proxy et objet réel : traitement identique par le client
- Le proxy transmet les requêtes à l'objet réel lorsque c'est nécessaire, gère sa création, ...



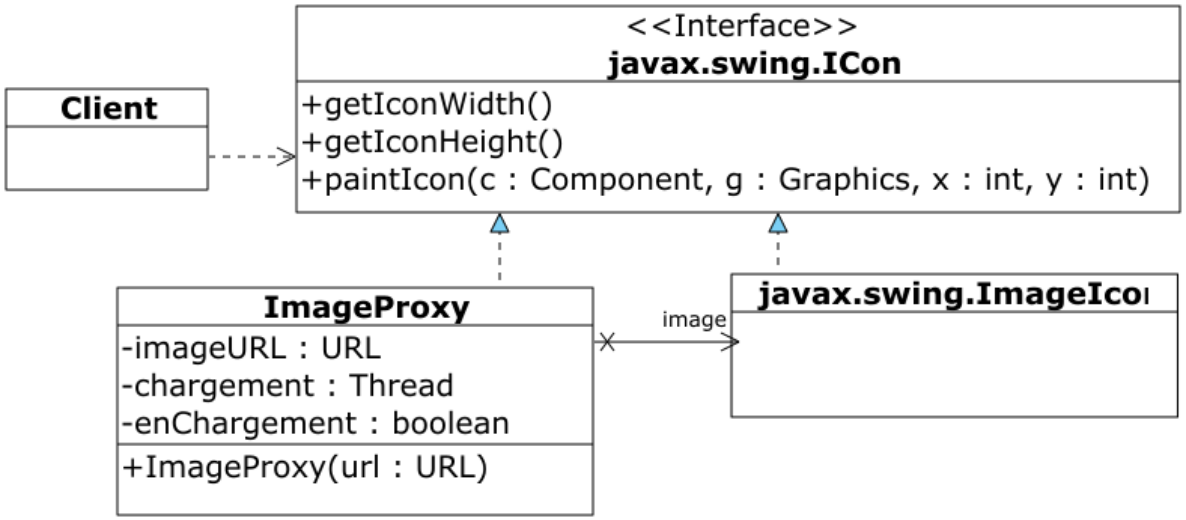
Pattern Proxy : Exemple de proxy distant

- API RMI (*Remote Method Invocation*) pour l'appel de méthode d'un objet distant
- le client possède un proxy distant = représentant local d'un objet distant, avec la même interface
- lors de l'invocation d'une méthode sur le proxy, la requête est sérialisée, transmise au serveur, dé-sérialisée par l'objet skeleton, transmise à l'objet distant, puis la valeur retournée fait le sens inverse



Pattern Proxy : Exemple de proxy virtuel

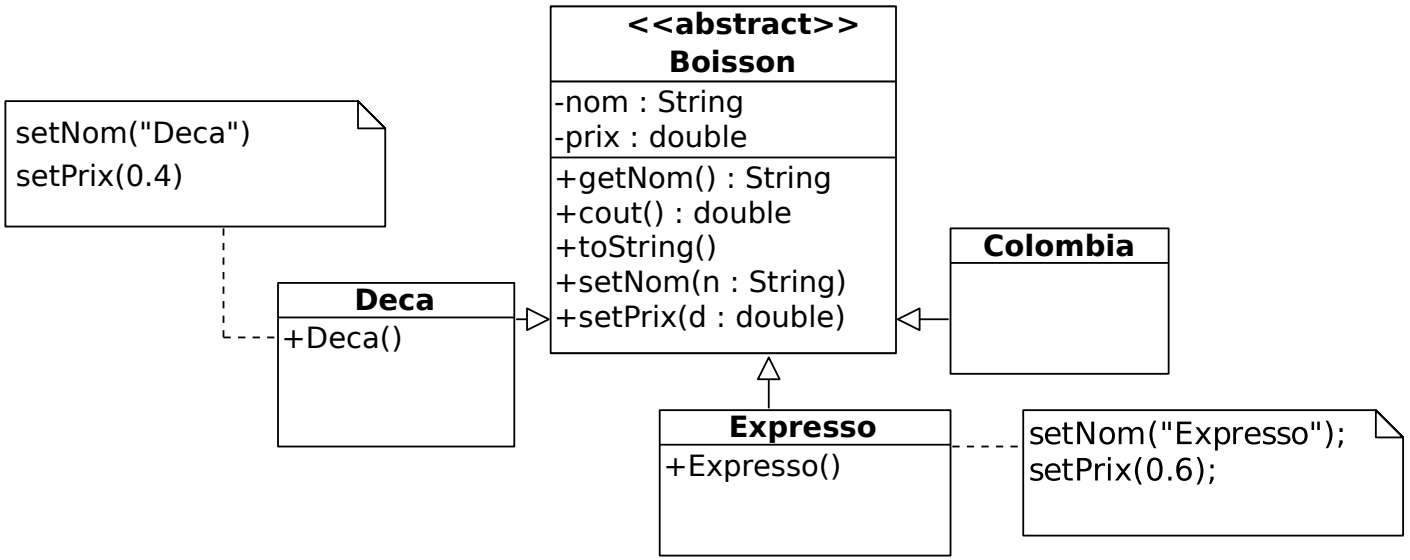
- Proxy virtuel pour éviter de geler l'application pendant le chargement d'une image obtenue à partir d'une adresse web




```
public int getIconWidth(){
   if (image!=null) return image.getIconWidth();    else return 800;}
public void paintIcon(...){
    if (image!=null) image.paintIcon(...);
    else{ if (!enChargement){
                enChargement = true;
                chargement = new Thread(new Runnable() {
                            public void run() {
                                        image = new ImageIcon(imageURL,
                                              "image");...}}
                chargement.start(); }     }}
```

Plan

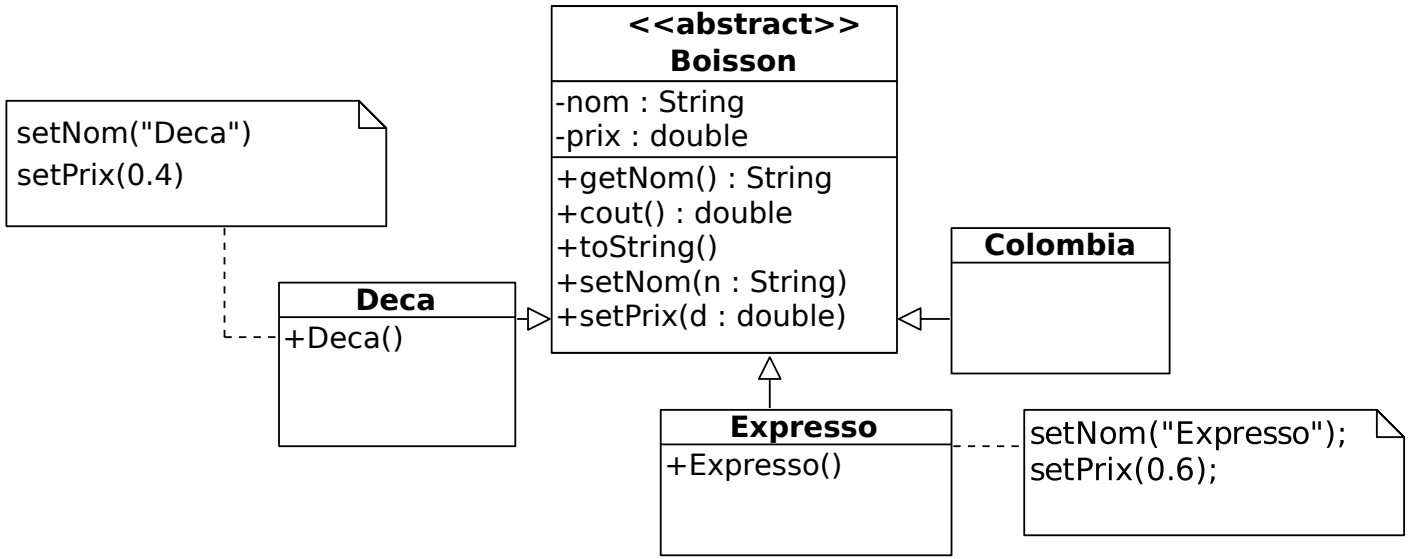
Pattern Décorateur (*Decorator*)



Exemple : Distributeur de boisson

A un distributeur, les clients ont le choix entre 3 types de café : Colombia (0.5 €), Expresso (0.6 €) et Deca (0.4 €). On veut afficher dans la console le nom et le prix du cafe choisi.

Pattern Décorateur (*Decorator*)



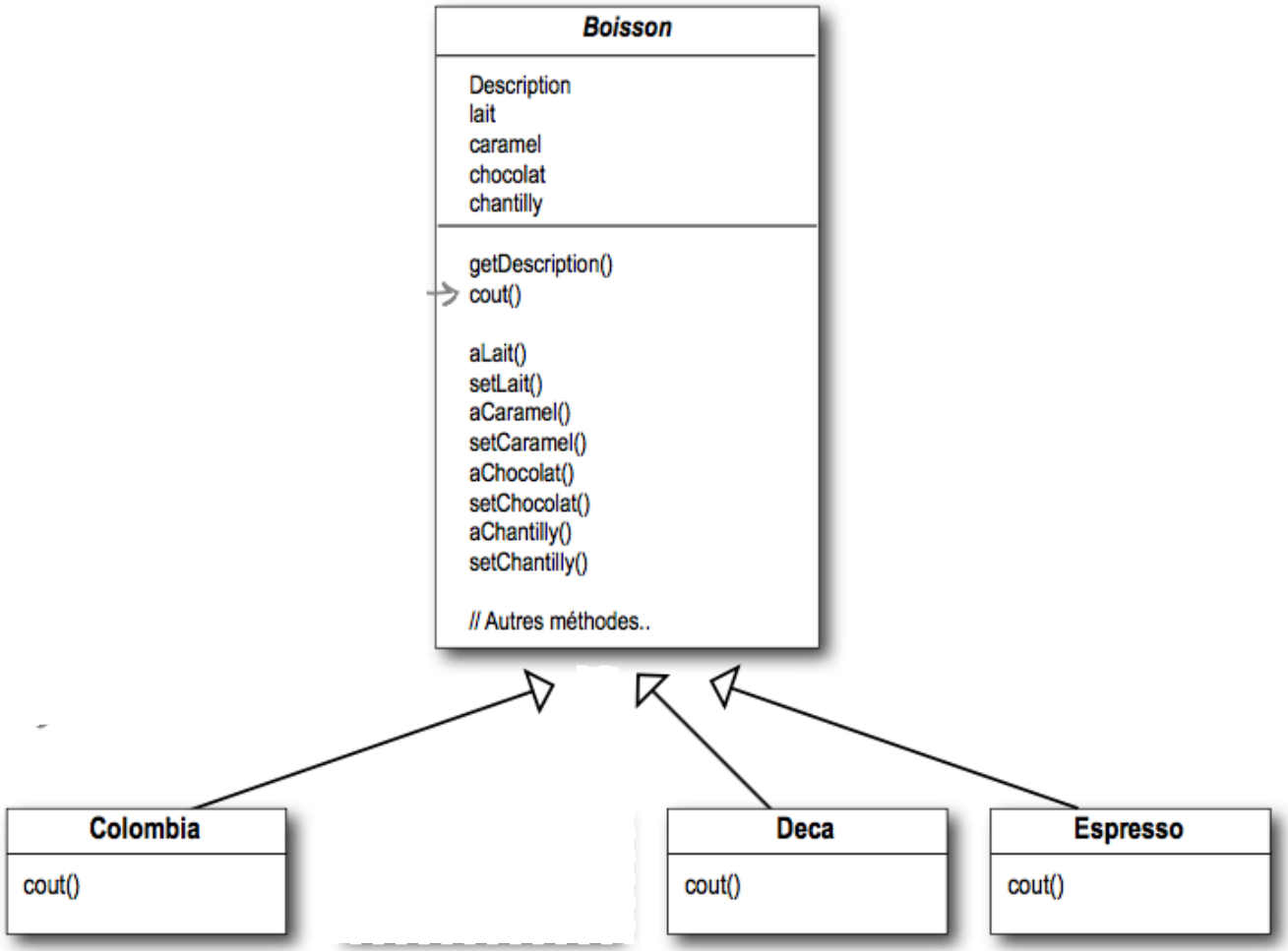
Exemple : Distributeur de boisson avec suppléments

On souhaite maintenant pouvoir **ajouter des suppléments aux boissons** : Lait, Sucre, Caramel, Chantilly. L'ajout de sucre ou de lait est facturé 0.1 €, de caramel 0.2 €et de chantilly 0.4 €. Il faut garder à l'esprit que l'ajout futur de nouveaux éléments (boissons ou suppléments) doit être simplifié.

Pattern Décorateur (*Decorator*)

Différentes solutions possibles pour ajouter des suppléments :

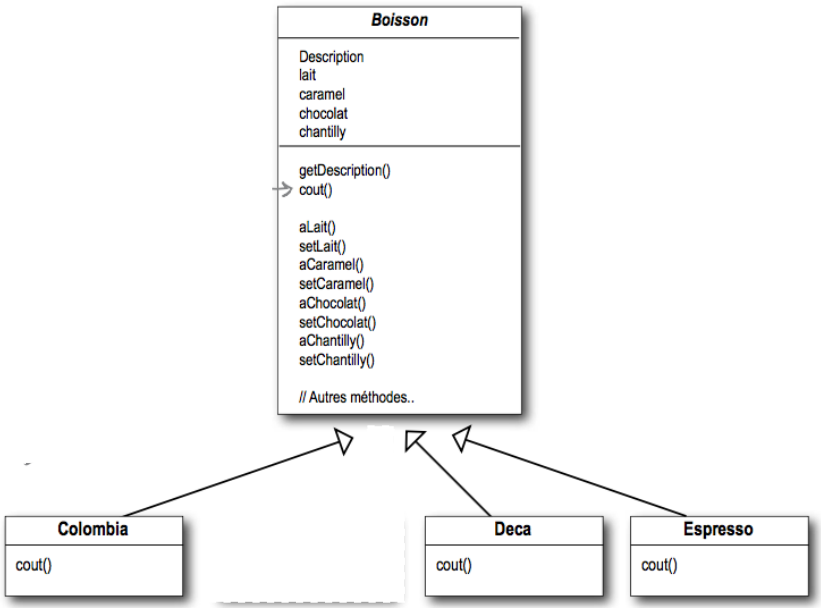
- Attributs booléens dans la classe `Boisson` pour mémoriser les suppléments



Pattern Décorateur (*Decorator*)

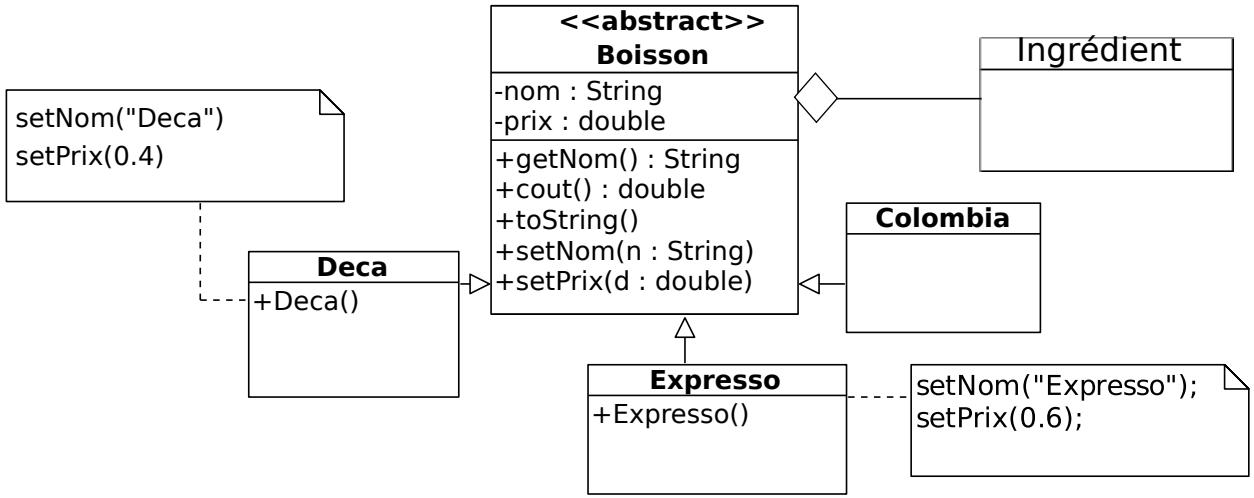
Différentes solutions possibles pour ajouter des suppléments :

- Attributs booléens dans la classe `Boisson` pour mémoriser les suppléments



- Modification du code existant si ajout de nouveaux ingrédients : OCP non respecté
- Ajout de fonctionnalités non appropriées pour certaines sous-classes de `Boisson`

Pattern Décorateur (*Decorator*)



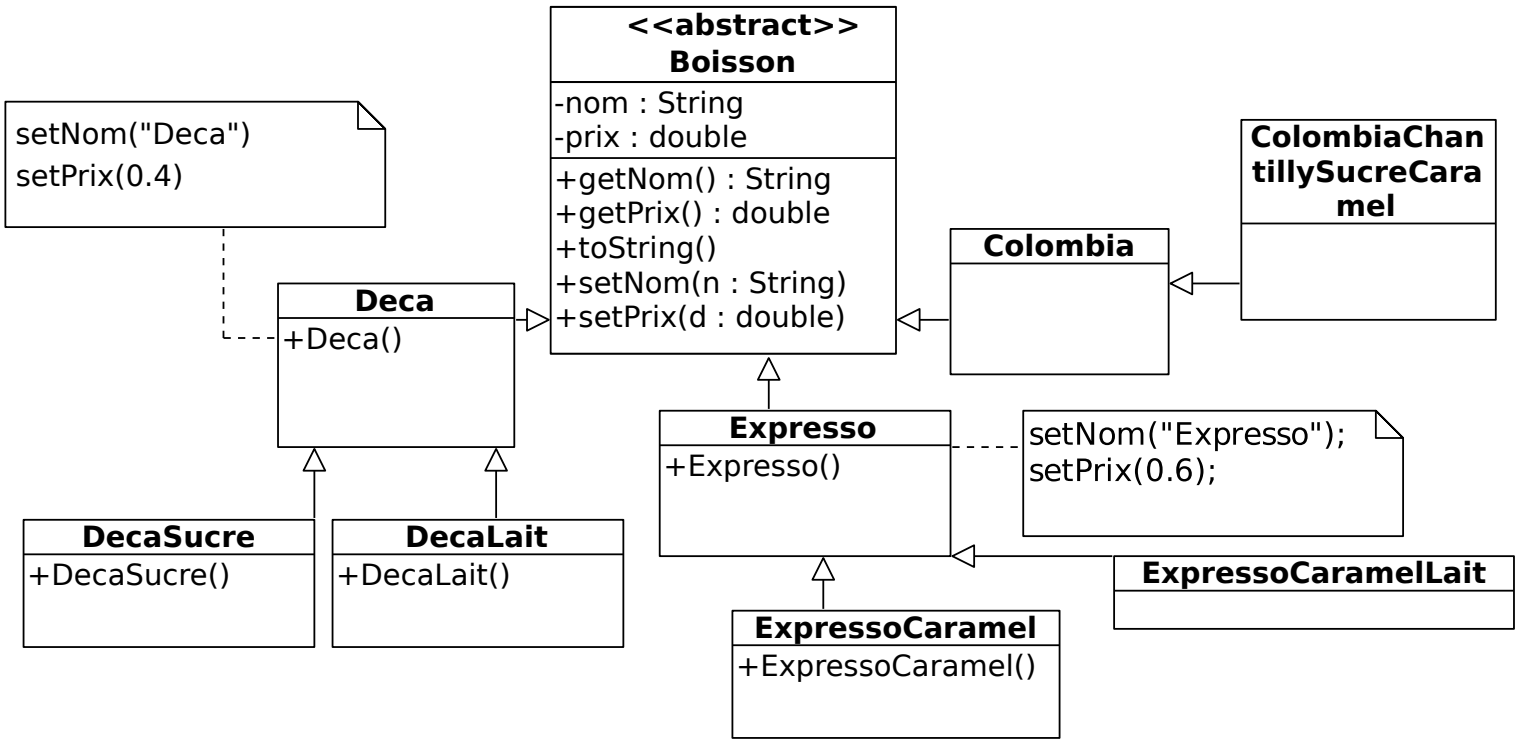
Différentes solutions possibles pour ajouter des suppléments :

- Attribut liste d'ingrédients dans la classe `Boisson`.

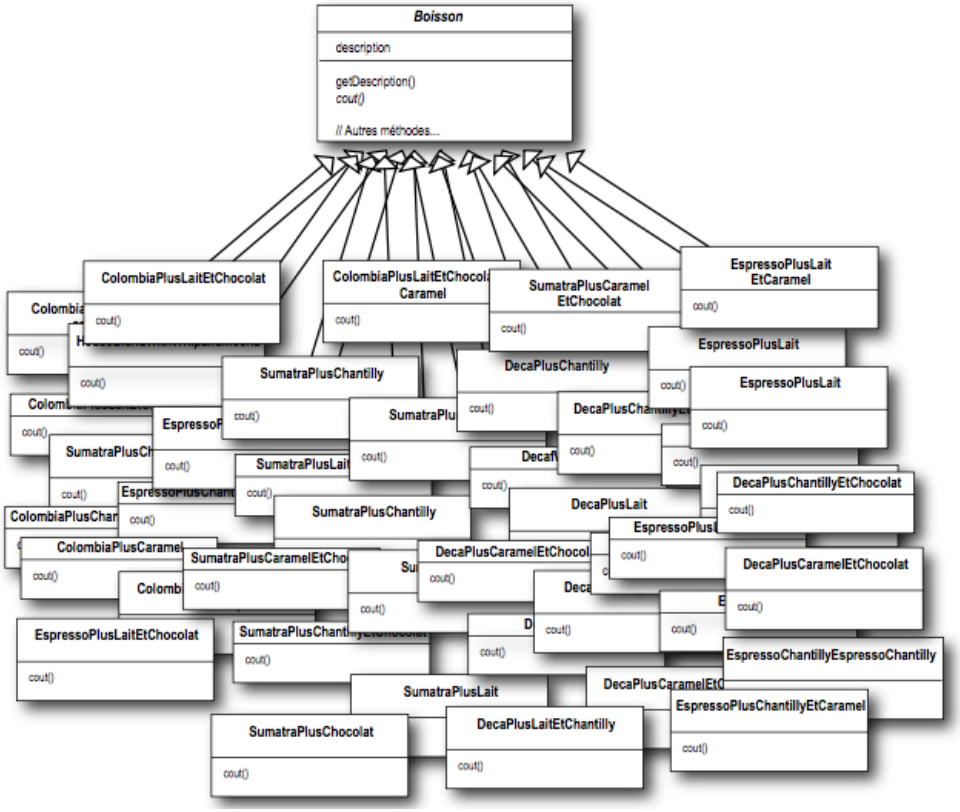
- Classe Boisson dépend de l'interface Ingrédient
- Classe Boisson contient la gestion de tous les Ingrédients, même s'il n'y a aucun ingrédient.

Pattern Décorateur (*Decorator*)

Une solution possible pour ajouter des suppléments **sans modifier le code existant** : Utiliser l'héritage.



Pattern Décorateur (*Decorator*)



- Modification de plusieurs classes si modification du prix d'un seul ingrédient : OCP non respecté
- **Explosion combinatoire** du nombre de classes filles si ajout d'un nouvel ingrédient ou d'une nouvelle boisson.

Pattern Décorateur (*Decorator*)

Héritage vs. composition pour la réutilisation

- Héritage : extension d'un comportement défini **statiquement** à la compilation
- Composition + Délégation : extension d'un comportement défini **dynamiquement** à l'exécution.
- La composition dynamique permet d'ajouter de nouvelles fonctionnalités sans toucher au code existant.

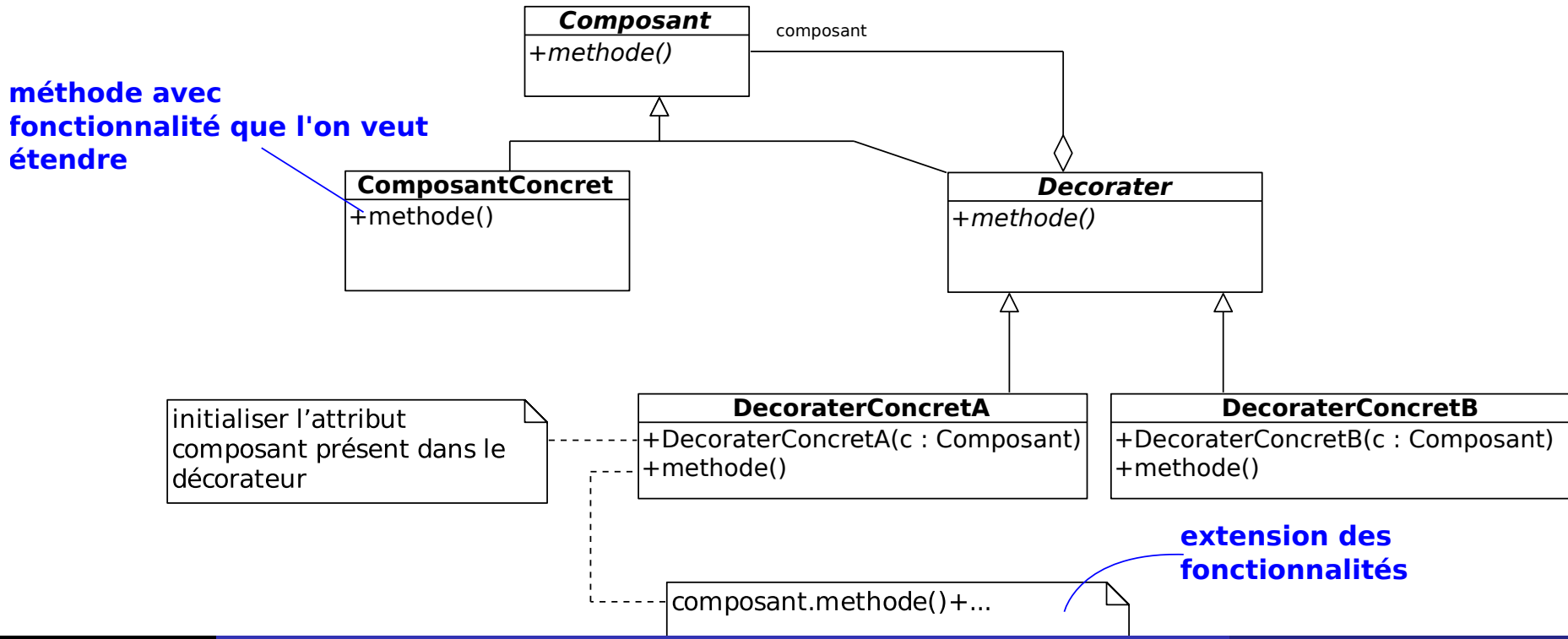
Pattern Décorateur

- But : Ajout dynamique de propriétés à des objets, sans modifier leur code (respect du principe OCP) et sans écrire une sous-classe.
- Alternative souple et dynamique à l'héritage.

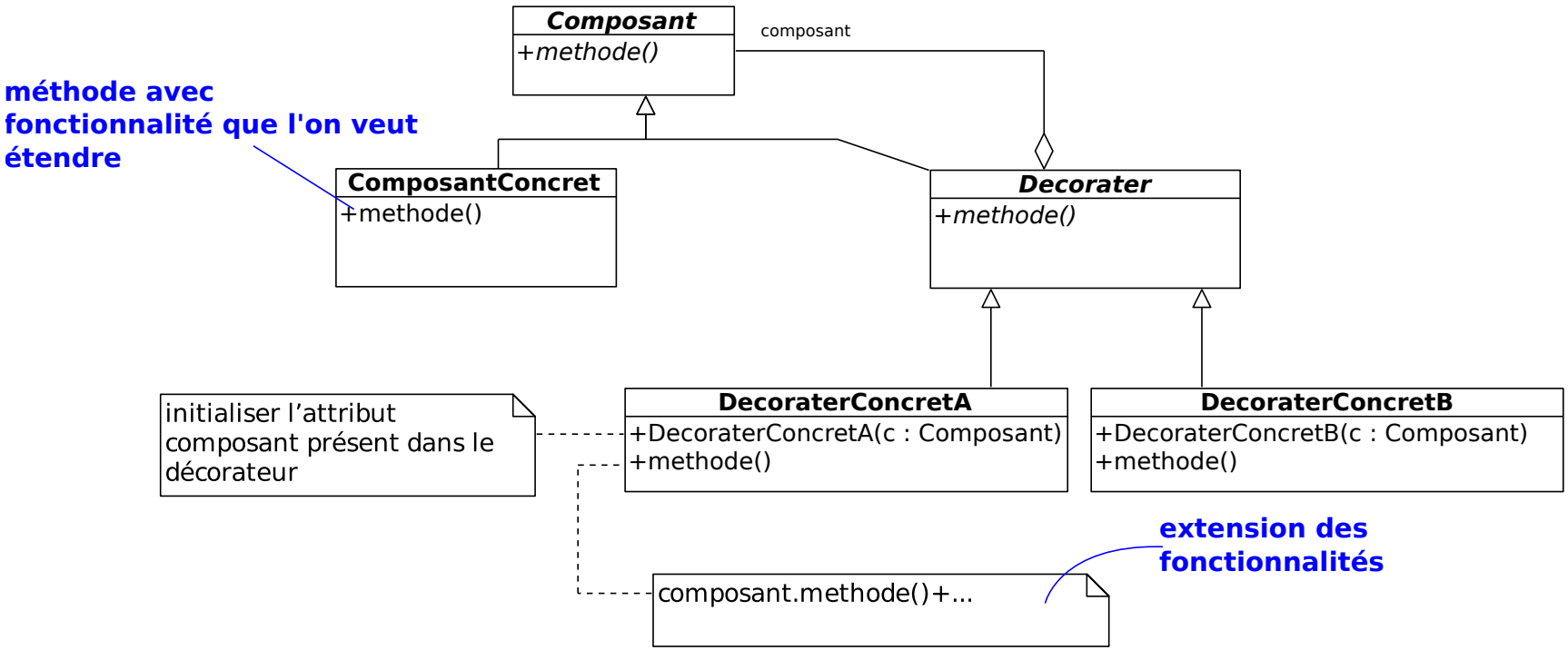
Pattern Décorateur (*Decorator*)

Moyen

- Combiner héritage et composition : une classe `Decorateur` est créée pour chaque caractéristique, chacune pouvant contenir de manière récursive un objet et ses caractéristiques.
- Ajout dynamique de responsabilités à `ComposantConcret` sans le modifier
- Peut conduire à créer de nombreux objets



Pattern Décorateur (*Decorator*)



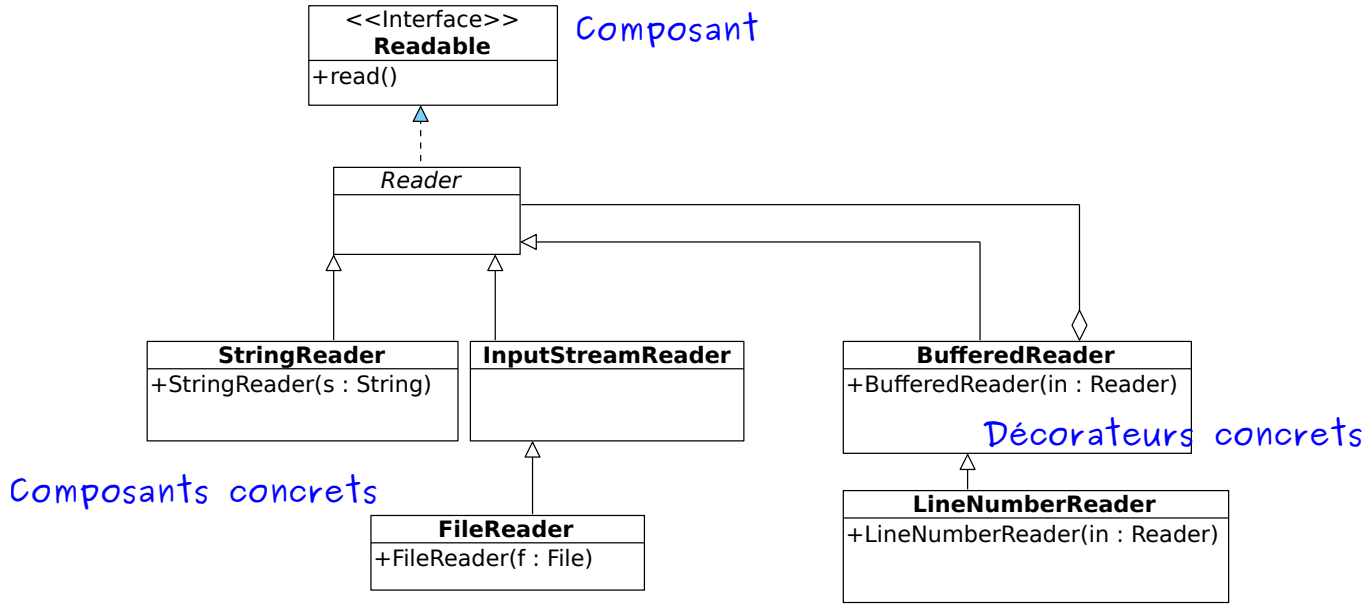
On obtient un objet `ComposantConcret` qui est emballé dans un `DecorateurConcret` (*wrapper*).

- Si on appelle la méthode sur l'objet `DecoraterConcret`, celle-ci va appeler la méthode du composant concret, ajouter ses propres traitements et retourner le résultat.
- Si on appelle la méthode de l'objet `ComposantConcret` (sans passer par le décorateur) on utilise alors l'ancienne version de la méthode.

Pattern Décorateur (*Decorator*)

Example : `java.io` pour les flux d'entrées/sorties

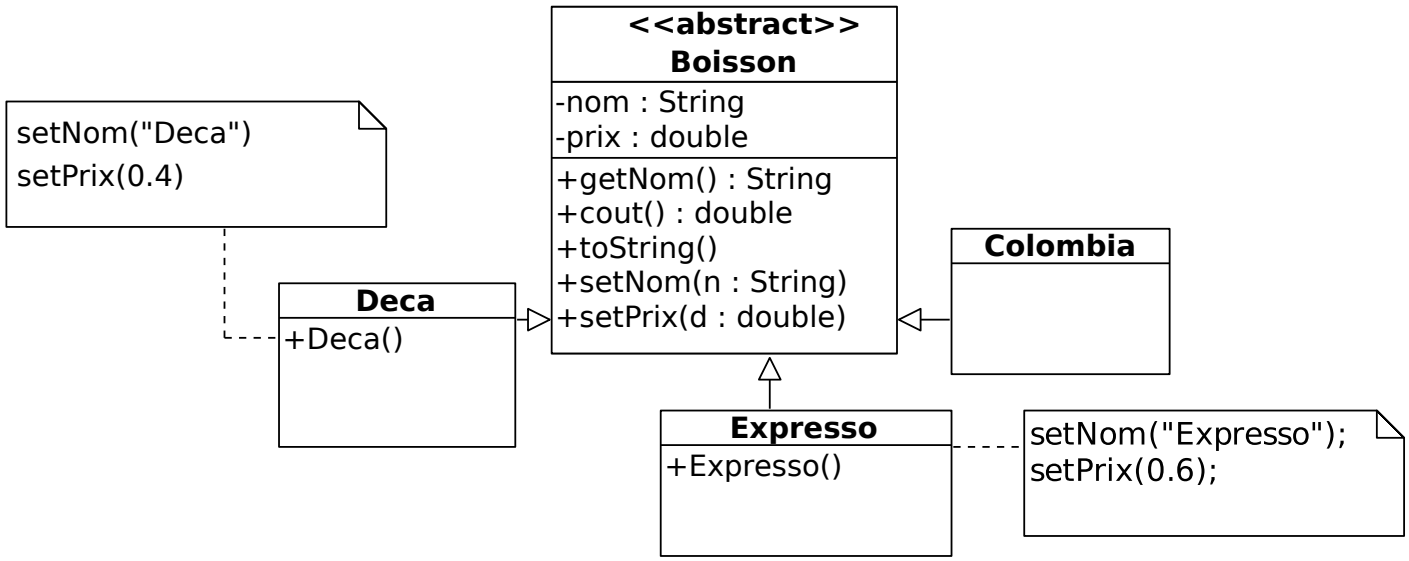
- classes abstraites et classes concrètes préfixées par flux ou filtre :
- pattern `Decorator` permet d'assembler toutes ces classes :



Pour lire les lignes d'un fichier texte, en comptant les lignes

```
FileReader fr = new FileReader("/path/to/file");
BufferedReader reader = new BufferedReader(fr);
LineNumberReader counter = new LineNumberReader(reader);
```

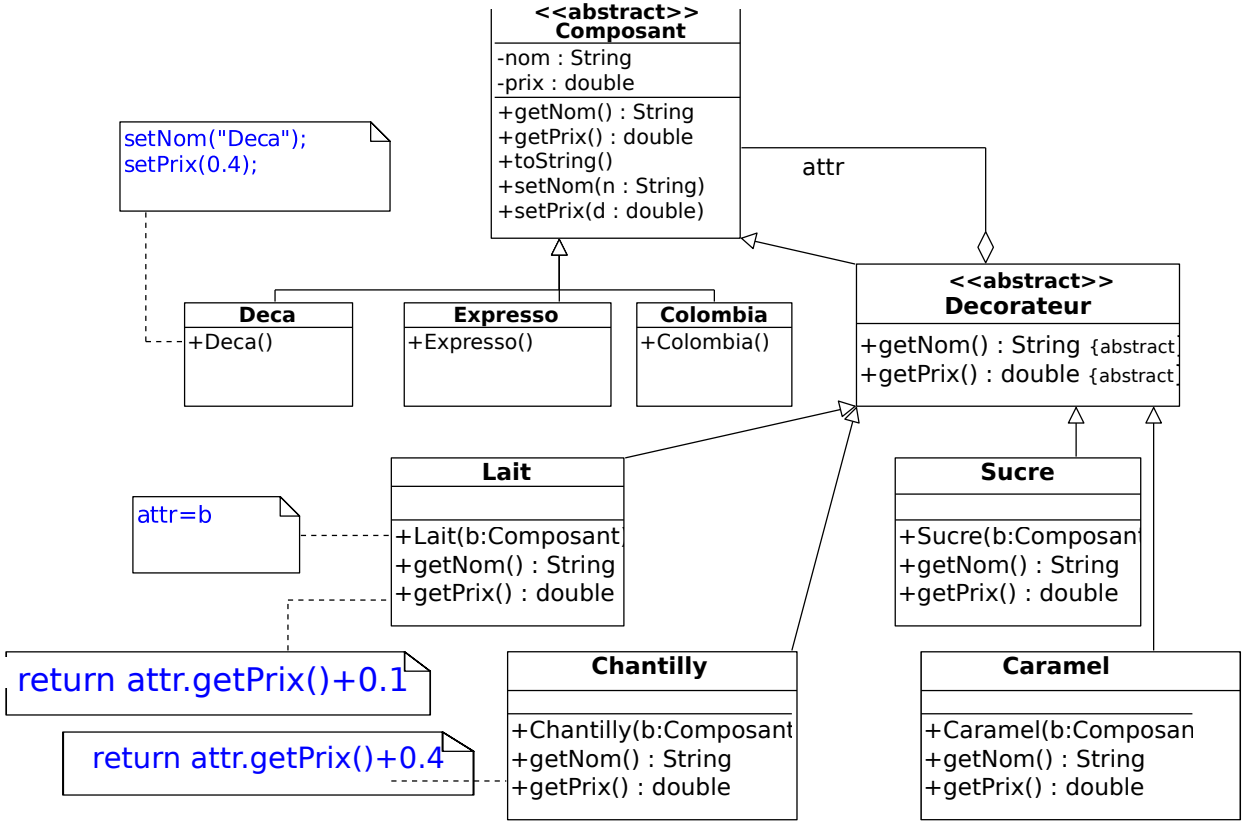
Exercice : Gestion d'un distributeur de café



A un distributeur, les clients ont le choix entre 3 types de café : Colombia (0.5 €),a Expresso (0.6 €) et Deca (0.4 €). Ils peuvent ajouter les suppléments suivants : Lait (+0.1€), Sucre (+0.1€), Caramel (+0.2€), Chantilly (+0.4€). L'ajout futur de nouveaux éléments (café ou supplément) doit être simplifié. On veut afficher dans la console le nom du café choisi avec ses suppléments et son prix.

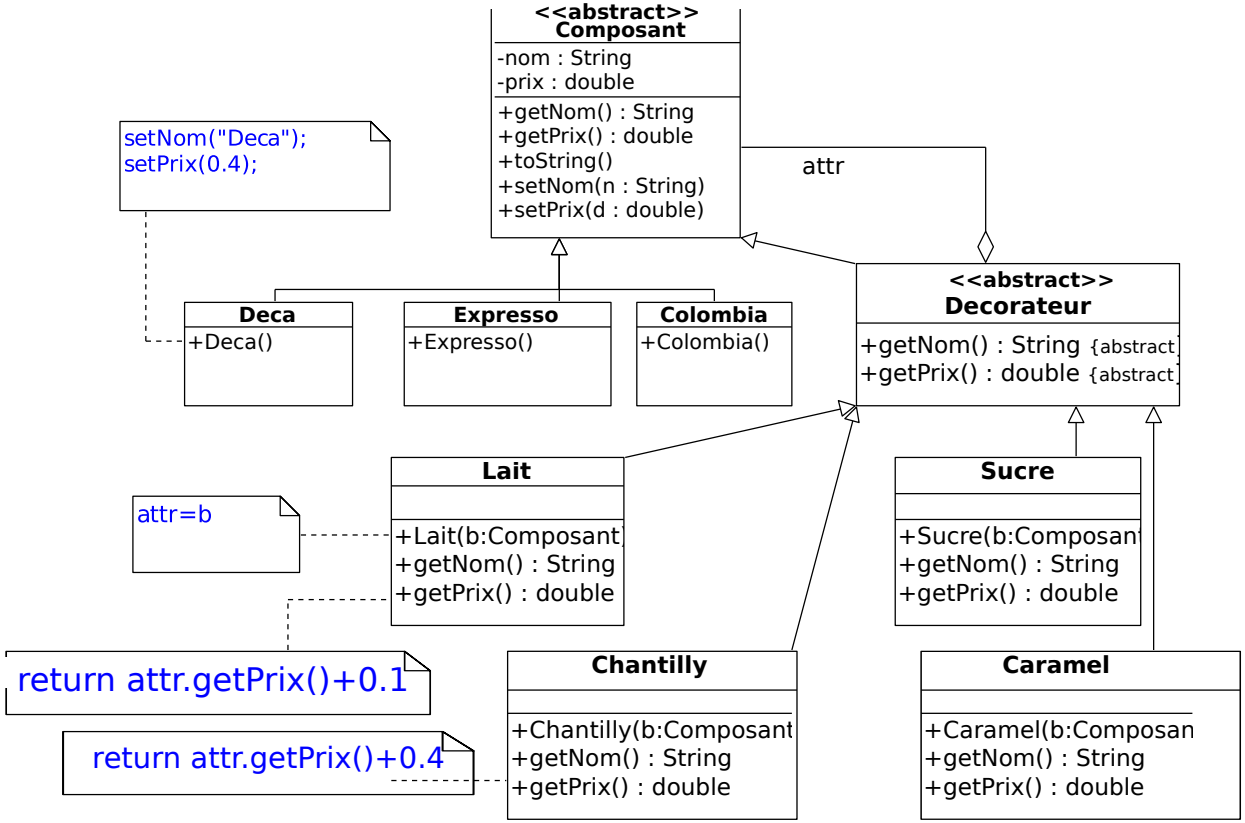
Proposer un refactoring en utilisant le patron Decorateur. Donner le diagramme de classes, préciser l'implémentation des méthodes, et donner le code pour afficher le prix d'un Expresso avec sucre, et d'un Colombia Chantilly Caramel.

Exercice : Gestion d'un distributeur de café



Donner le code pour afficher le prix d'un Expresso avec sucre, et d'un Colombia Chantilly Caramel.

Exercice : Gestion d'un distributeur de café



Donner le code pour afficher le prix d'un Expresso avec sucre, et d'un Colombia Chantilly Caramel.

```
// Creation et affichage du prix d'un  Expresso avec sucre
Composant d1 = new Expresso(); d1 = new Sucre(d1);System.out.println(d1.getPrix());
// Creation et affichage du prix d'un  Colombia Chantilly Caramel
Composant d1 = new Colombia(); d1 = new Chantilly(d1);d1 = new Caramel(d1);System.out.println(d1
      .getPrix());
```

Plan

Patterns de comportement

- Définir comment organiser les objets pour que ceux-ci **collaborent** (distribution des responsabilités).
- Fournir des solutions pour **distribuer** les traitements et les algorithmes entre les objets.
- Utilisation de l'association entre objets :
  - pour décrire comment des groupes d'objets coopèrent (ex : Mediator)
  - pour définir et maintenir des dépendances entre objets (ex : Observer)
  - pour encapsuler un comportement dans un objet et déléguer les requêtes à d'autres objets (ex : Stratégie, Etat, Commande)
  - pour parcourir des structures en appliquant des comportements (ex : Visiteur, Iterateur)

Plan

Pattern Iterateur (*Iterator*)

Contexte

Le système doit parcourir les éléments d'un objet complexe. La classe de l'objet complexe peut varier.

But

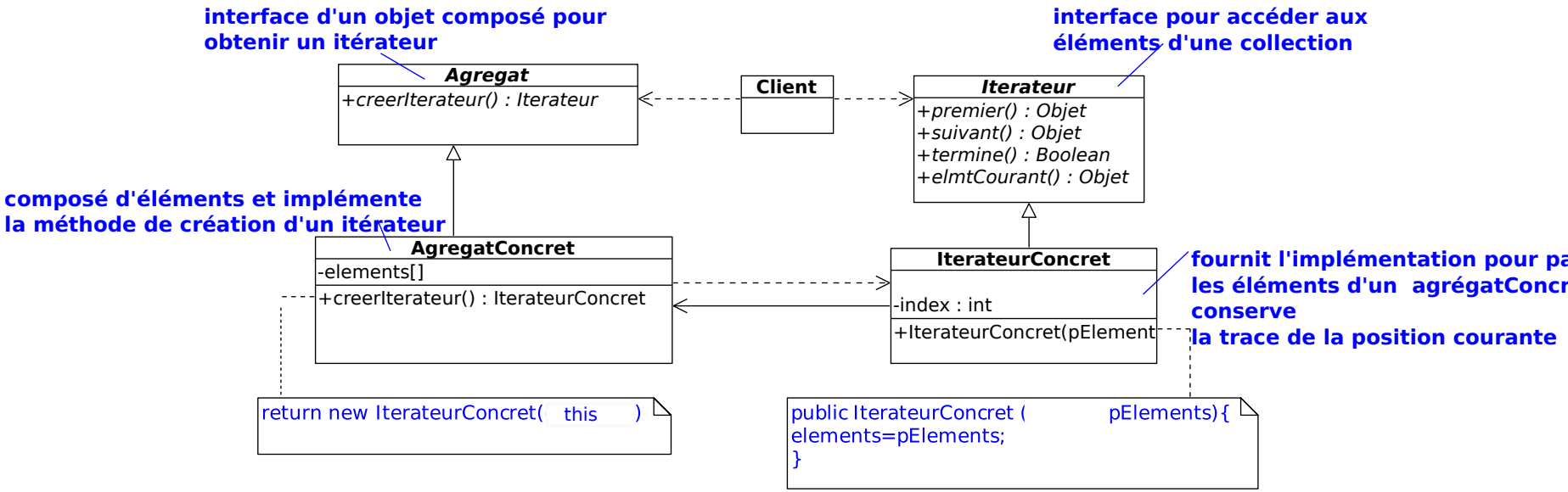
- Permettre à un client de parcourir une collection d'objets sans connaître l'implémentation de cette collection (liste, tableau, ...).
- **Encapsulation de l'itération** (ce qui varie) ; le client est protégé des variations possibles de l'implémentation de la collection

Exemple : Une collection contient un ensemble d'objets stockés par différentes structures (tableau, vecteur, liste chainée ...). Le client qui accède au contenu de la collection ne souhaite pas être concerné par cette manière de gérer les objets. La collection offre donc un point d'accès unique sous la forme d'une interface.

Pattern Iterateur (*Iterator*)

But

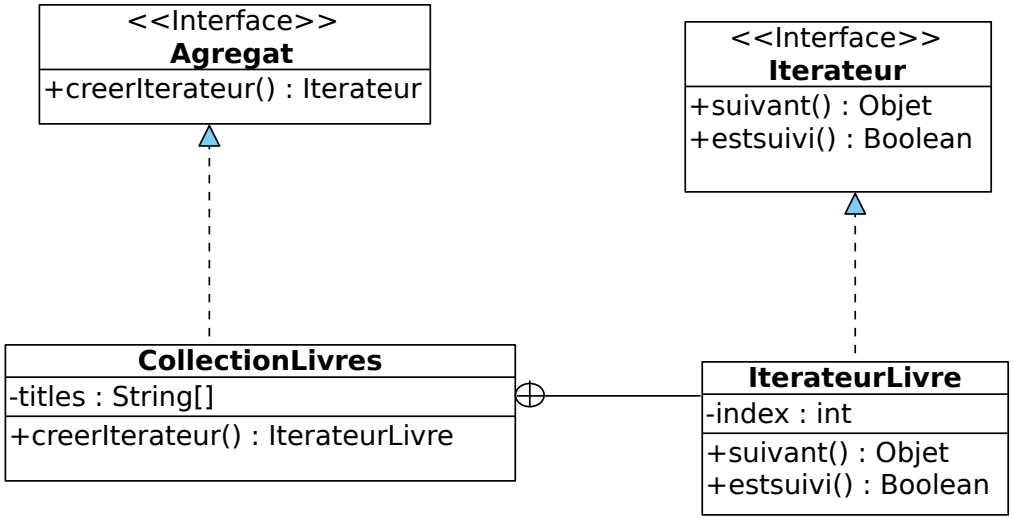
- Permettre à un client de **parcourir une collection d'objets sans connaître l'implémentation** de cette collection (liste, tableau, ...).
- Encapsulation de l'itération (ce qui varie) avec l'interface `Iterateur` : le client est protégé des variations possibles de l'implémentation de la collection




```
Agregat lAgregat = new AgregatConcret();
Iterateur lIterateur = lAgregat.creerIterateur();
// Parcours les elements de l'objet lAgregat grace a Iterateur
while(!lIterateur.fini()) {
System.out.println(lIterateur.suivant()); }
```

Pattern Iterateur (*Iterator*)

Exemple : Pattern Iterateur sur une collection de livres.




```
class CollectionLivre implements Agregat
{ private String titles[] = {"Design Patterns","1","2","3","4"};
  public IterateurLivre creerIterateur(){
        return new IterateurLivre();     }

    private class IterateurLivre implements Iterateur{
        private int index;
        public boolean estsuivi(){
          if (index < titles.length) return true;
          else return false;}
        public Object suivant(){
          if (this.estsuivi()) return titles[index++];
          else return null;}}}
```

Pattern Iterateur (*Iterator*)

- Le client n'a pas à savoir comment est implémenté l'agrégat : le client est **protégé des variations** possibles de l'agrégat, découplage du client de l'implémentation réelle de la collection
- Une boucle peut gérer de façon polymorphe toute collection d'éléments tant qu'elle implémente les interfaces.
- Parcours (itération) séparé de l'agrégation : **encapsulation de l'itération** ; l'`Agregat` a une **unique responsabilité** : gérer la collection d'objets.
- Possibilité d'avoir plusieurs itérateurs en même temps sur la même structure
- Les itérateurs différents peuvent implémenter des politiques de visite différentes (p. ex. *depth-first* vs *breadth-first*)

Pattern Iterateur (*Iterator*)

- Java offre de nombreuses classes implémentant l'interface `Collection` pour stocker des groupes d'objets et y accéder (`Vector`, `LinkedList`, ...). Ces classes disposent d'un moyen d'obtenir un itérateur : `iterator(): Iterator`.
- l'interface `Iterator` (dans java.util), ajoutée dans le JDK 1.2 pour gérer les collections ;

```
interface Iterator<E> {
hasNext():Boolean
next():E
remove()
}
```

```
List l = new ArrayList();
Iterator it = l.iterator();
while (it.hasNext()){
...traitement sur it.next()...}
```

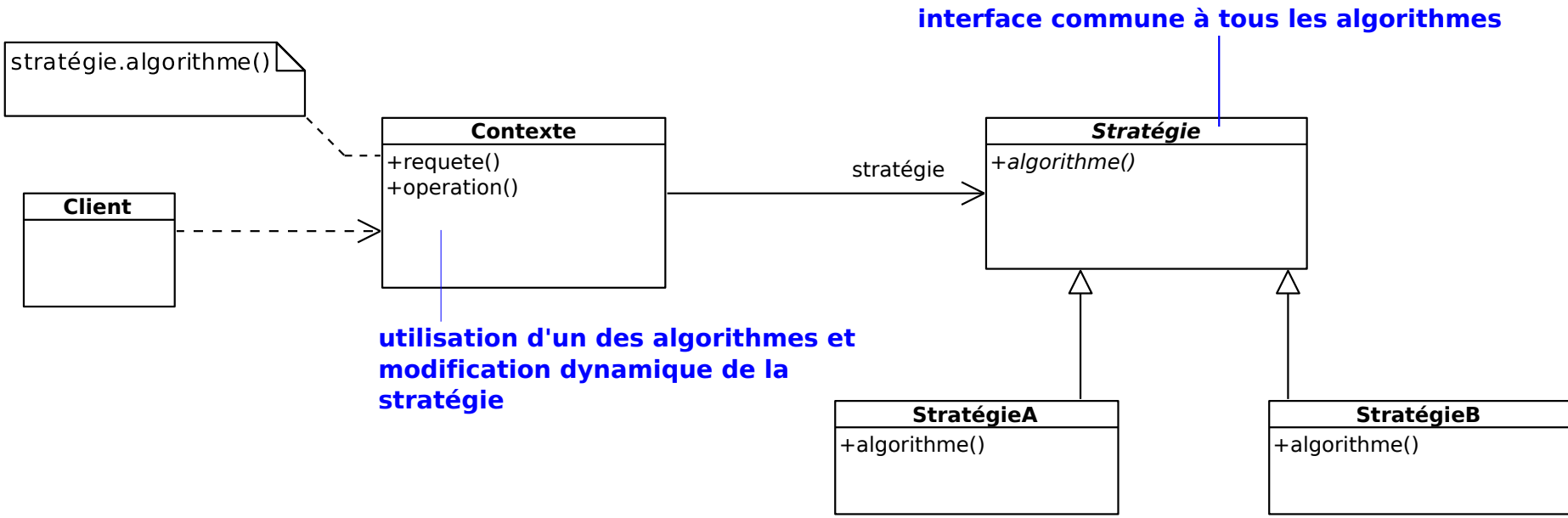
- les boucles `for` étendues (*foreach*), ajoutées dans le JDK 1.5 avec l'interface `Iterable`.

```
interface Iterable<T> {
Iterator<T> iterator();
}
```

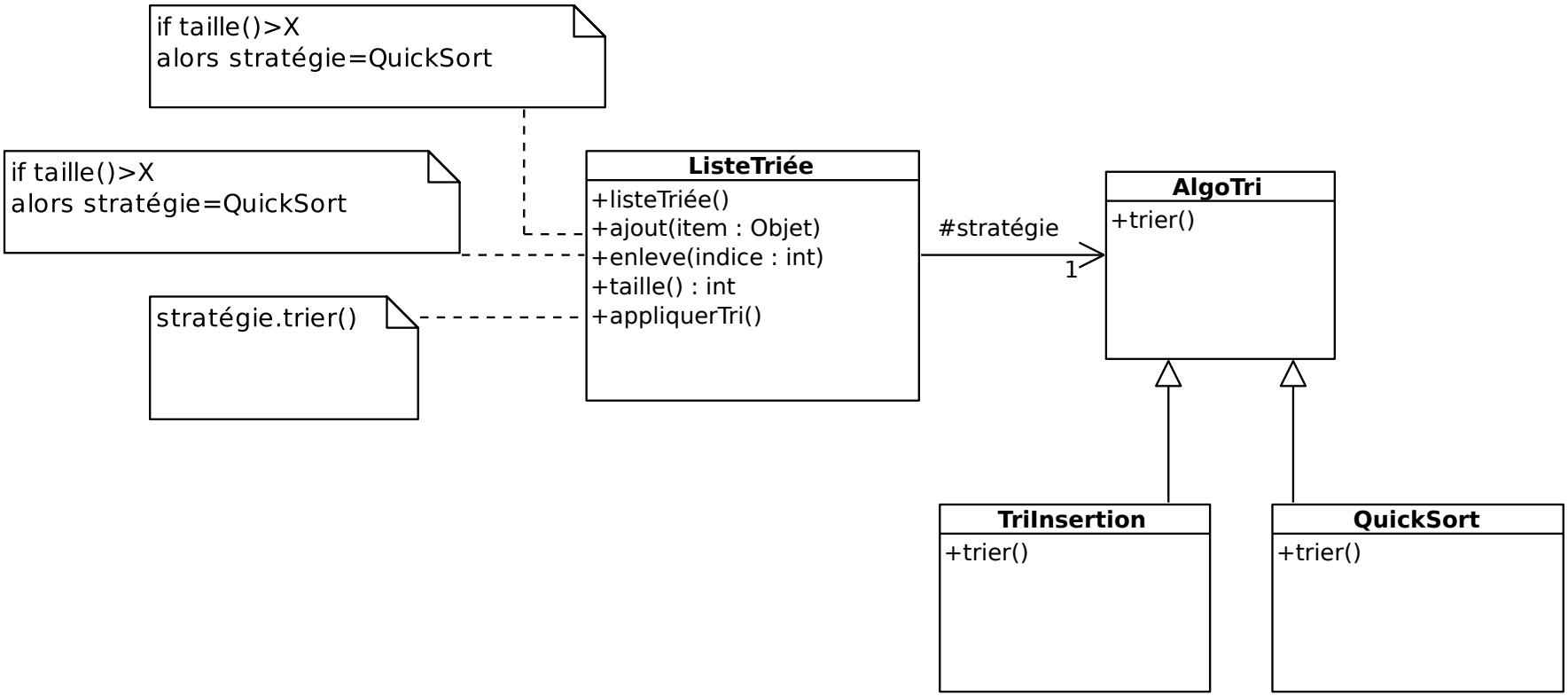
Plan

Pattern Stratégie (*Strategy*)

- But : **Modifier dynamiquement les algorithmes** utilisés par une classe.
- Moyen : Encapsulation des algorithmes (qui peuvent varier) en dehors de la classe qui utilise un de ces algorithmes (stable)
- Interface `Strategie` implémentée par les différents algorithmes
- Le client utilise une famille d'algorithmes encapsulés

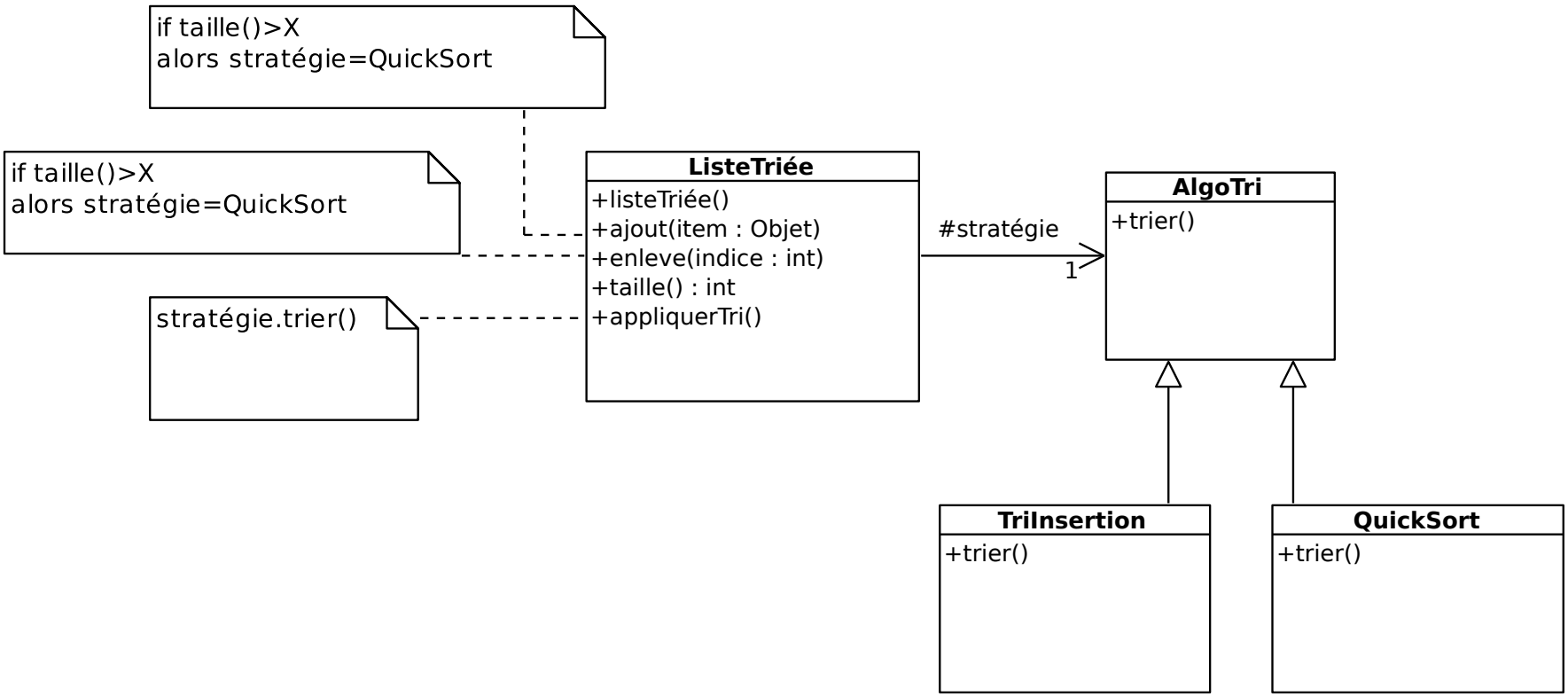


Exemple 1



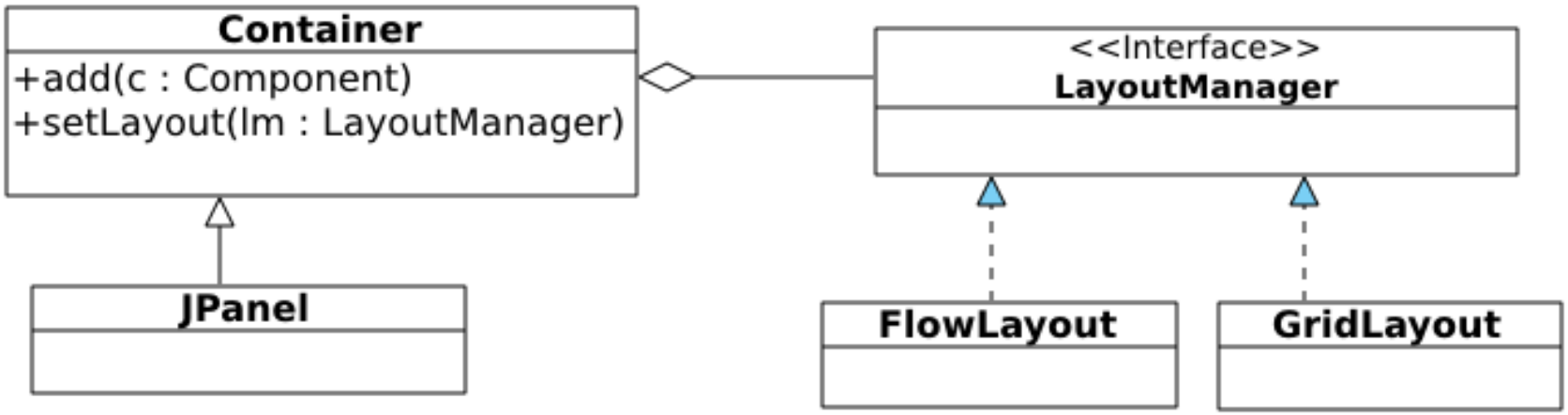
- Associer une classe de base abstraite à la catégorie d'algorithmes de tri.
  $\rightarrow$ interface commune à tous les algorithmes de tri.
- Associer une classe concrète à chaque algorithme.
- La classe `ListeTriée` agrège un objet de la classe `AlgoTri`.

Exemple 1



- possibilité de modifier un algorithme indépendamment des autres
- facilité d'ajout d'un nouvel algorithme (OCP)
- `Stratégie` : modification dynamique par le client de l'algorithme utilisé (setter dans la classe contexte)

Exemple 2



- algorithmes de mise en page encapsulés dans des classes implémentant `LayoutManager`
- associer un algorithme de mise en page à un conteneur graphique via le *pattern* Strategie.

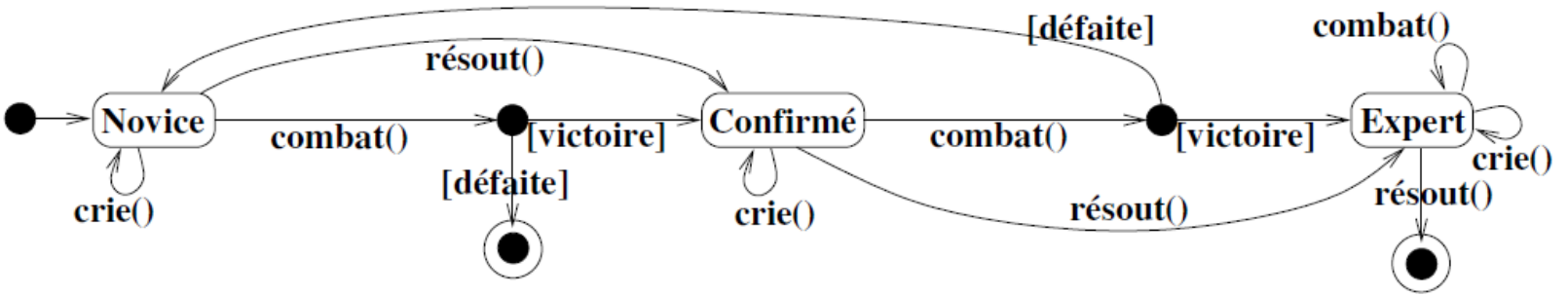
Plan

Etat (*State*)

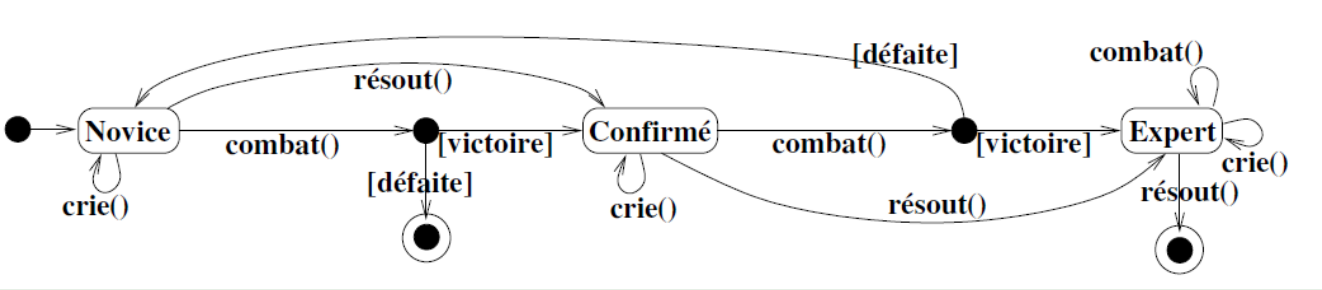
Contexte

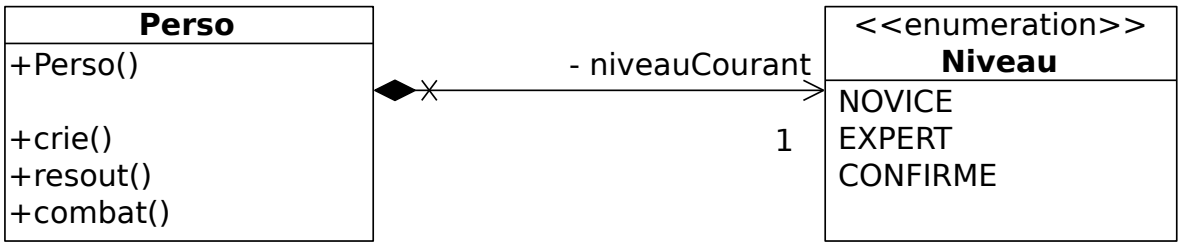
Une classe a de nombreux états finis, et des transitions entre eux.

Les personnages d'un jeu peuvent avoir 3 niveaux (états) et peuvent, **dans chaque état**, combattre, résoudre des énigmes et crier :



Etat (*State*)



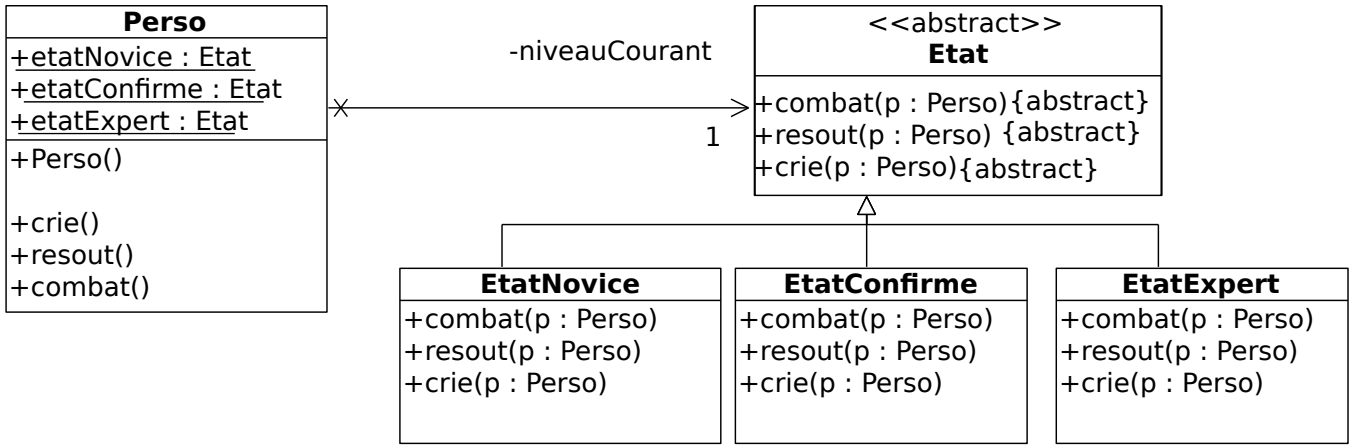


- Sol. 1 : Dans `Perso`, une méthode par action, un cas par état dans chaque méthode :

```
public void resout(){
  if (niveauCourant == NOVICE){ ... ; niveauCourant = CONFIRME
      ;}
  else if (niveauCourant == CONFIRME){ ... ; niveauCourant =
      EXPERT;}
  else { ... ; System.exit(0);} }
```

→ Coûteux d'ajouter un nouvel état

Etat (*State*)



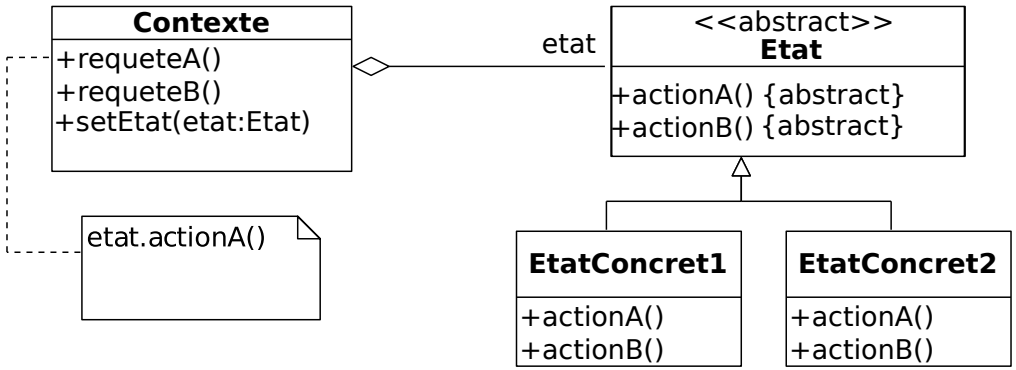
- Sol. 2 : Encapsuler ce qui varie (le comportement de chaque état) dans des classes séparées

```
  public class Perso {
   public static final Etat etatNovice = new EtatNovice(); ...
   public Perso(){niveauCourant = etatNovice;}
   //requete sur perso (action) deleguee a etat courant
   public void resout(){niveauCourant.resout(this);}
 }
  public class EtatNovice extends Etat {...
//Implementation du comportement de l'action pour cet etat +
    decision de l'etat successeur
   public void resout(Perso p){ ... ; p.setNiveau(Perso.etatConfirme
       ); } }
```

Etat (*State*)

But

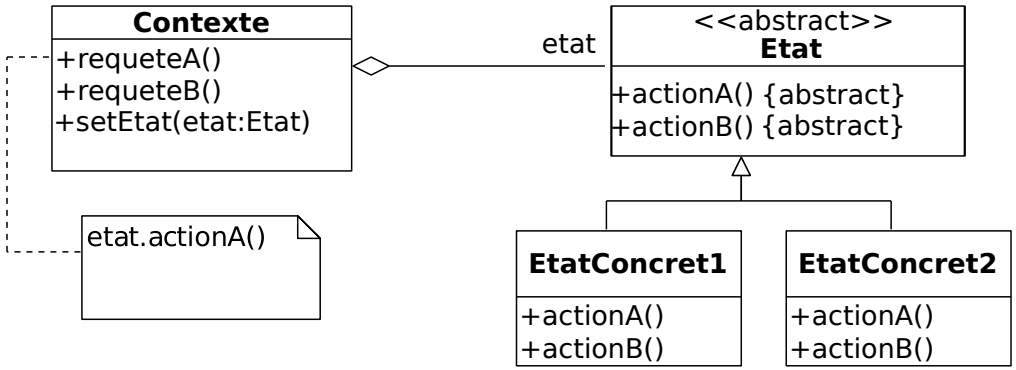
- Modifier le comportement d'un objet en fonction de son état interne (comme si l'objet changeait de classe).
- **Le changement d'état est transparent au client.**



Moyen

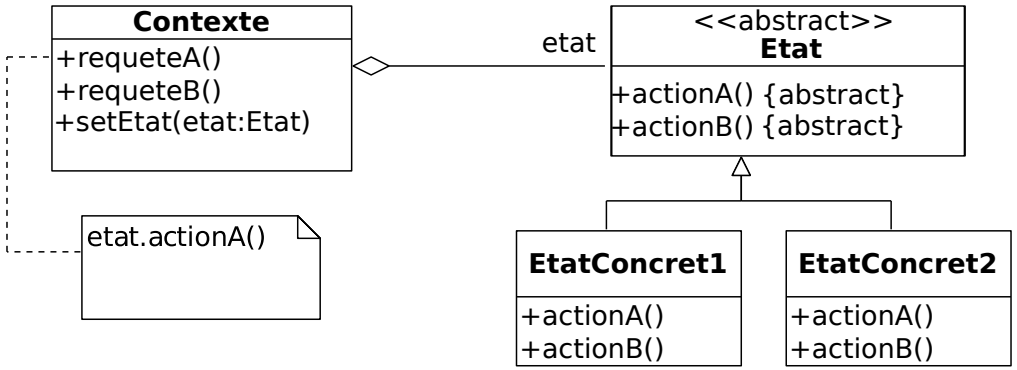
- `Contexte` : objet dont le comportement change en fonction de son état ; présentation des opérations au Client, référence vers l'état actuel, délégation des requêtes à l'état courant
- `Etat` : interface commune à tous les états (états interchangeables)
- `EtatConcret` : sous-classe pour un état interne. Elle fournit sa propre implémentation de la requête (*action_X()*) comme un comportement spécifique à cet état.

Etat (*State*)



- Les états sont fermés aux modifications mais ouverts aux extensions : ajout et la suppression d'états simplifiés
- Suppression de traitements conditionnels de grande taille

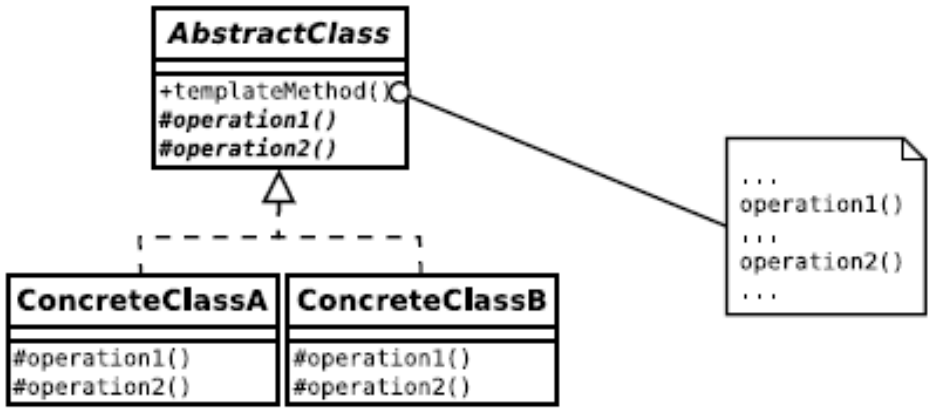
Etat et Strategie



- Même pattern : utilisent le polymorphisme, la délégation, l'encapsulation de comportement dans des objets séparés
- Mais intention différente
- Stratégie : changement dynamique, visible / voulu par le client
- Etat : le client ne sait rien des objets Etat, changement d'état statique / fixé par le diagramme d'états et réalisé dans les états concrets
- Dans les deux cas, l'encapsulation des comportements implique plus de classes : prix à payer pour plus de souplesse

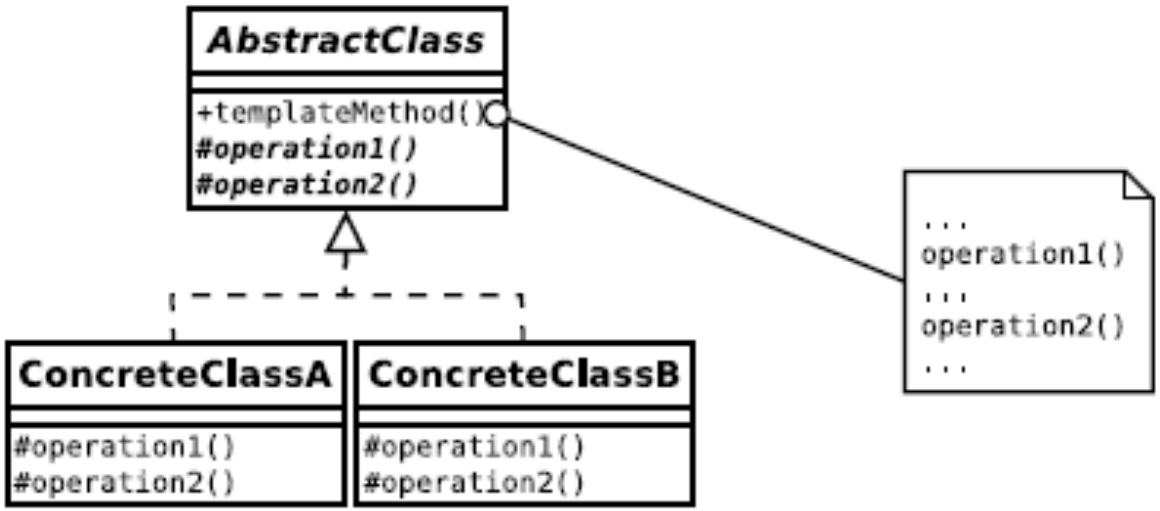
Plan

Pattern Patron de méthode (*Template Method*)



- Contexte :
  - Création d'un **patron d'algorithme** = méthode qui définit un algorithme sous la forme d'une suite d'étapes
  - La structure de l'algorithme est figée, mais l'implémentation de certaines étapes peut varier
- But :
  - Factoriser un comportement générique lorsque le détail de certaines fonctionnalités de ce comportement peut varier.
  - Définir le squelette d'un algorithme dans une méthode,en déléguant certaines étapes aux sous classes.
- Encapsuler ce qui peut varier : des étapes de l'algorithme

Pattern Patron de méthode (*Template Method*)

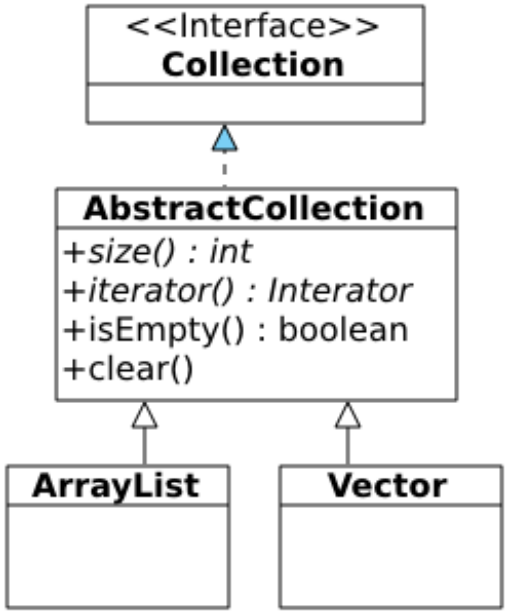


Moyen

Définir une classe abstraite qui :

- déclare dans une opération `templateMethode` (*final* si partie commune non modifiable) le squelette d'un algorithme, chaque étape de l'algorithme correspond à une méthode
- les étapes invariantes sont implémentées dans la classe abstraite
- les étapes pouvant varier sont déclarées abstraites et sont déléguées à des sous-classes

Pattern Patron de méthode (*Template Method*)



Méthodes templates :

```
public boolean isEmpty() {
  return size() == 0;
}
public void clear() {
  Iterator<E> it = iterator();
  while (it.hasNext()) {
    it.next();
    it.remove(); }}
```

Pattern Patron de méthode (*Template Method*)

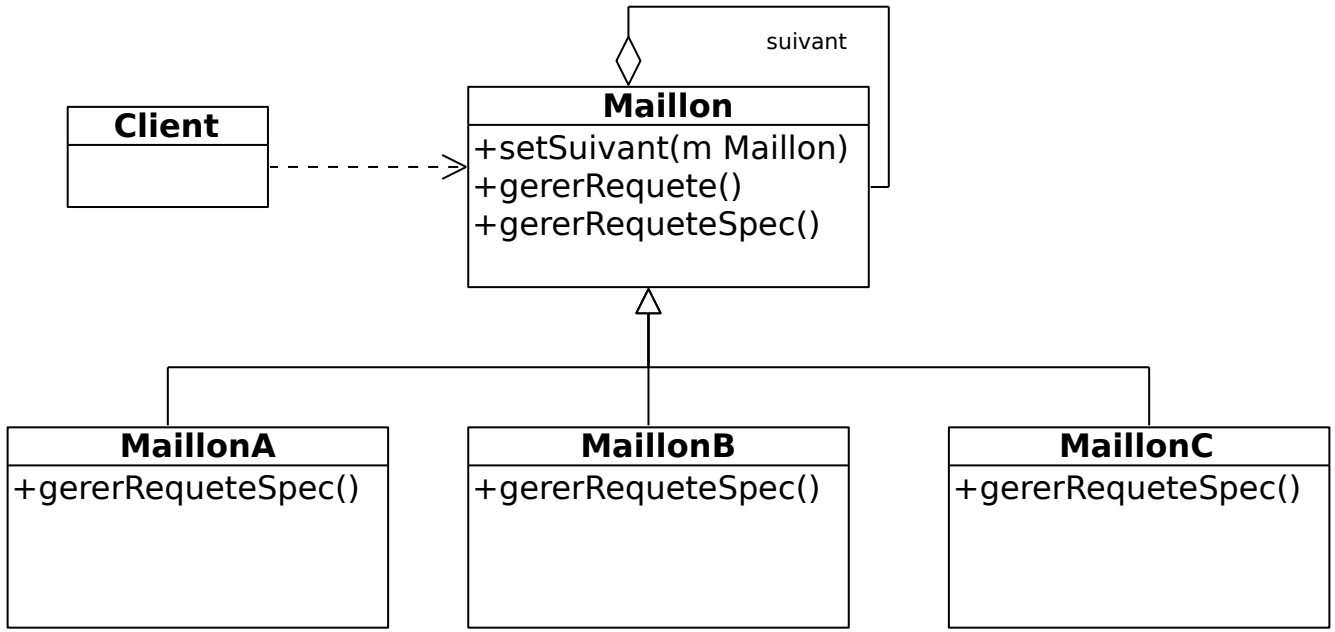
- Supprimer la duplication du code, protection des variations
- Méthodes adaptateurs (*hooks*) : implémentations vides ou par défaut ; points spécifiques dans l'algorithme où l'utilisateur peux ajouter des fonctionnalités
- Structure de contrôle inversée : ce ne sont pas les sous-classes qui appellent les méthodes de la superclasse, mais **la superclasse qui contrôle le flux d'exécution**

Différence entre patrons `Stratégie` et `Template Method` :

- Le patron `Stratégie` **encapsule une famille d'algorithmes** dans des objets, permettant à l'utilisateur d'utiliser et d'ajouter des algorithmes facilement.
- Le patron `Template method` **définit le squelette d'un algorithme** dont certaines parties sont implémentées différemment par les sous-classes. La méthode `templateMethode` contrôle la structure de l'algorithme.

Plan

Pattern Chaîne de responsabilité (*Chain of Responsibility*)

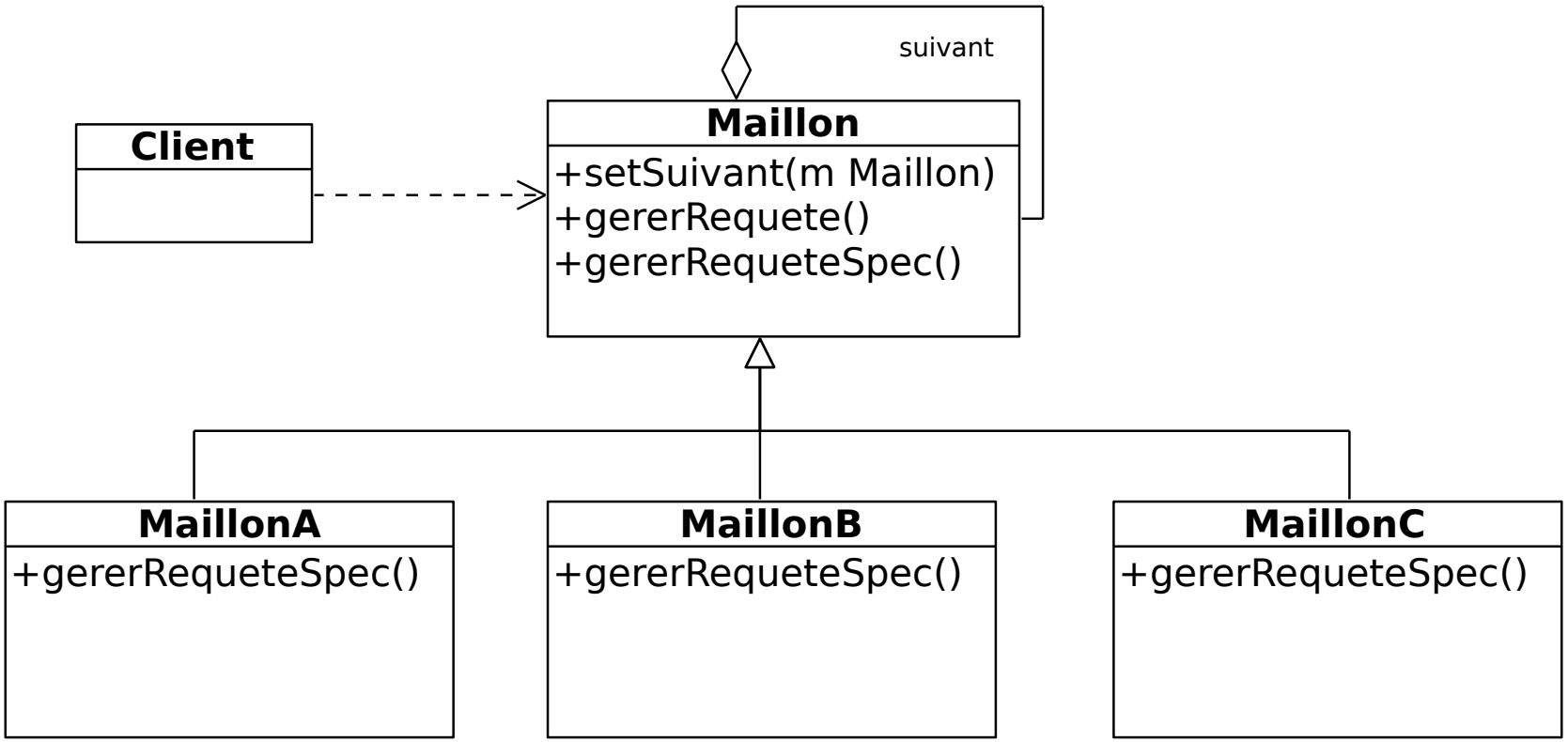


- Contexte : plus d'un objet est capable de répondre à une même requête (et il n'est pas connu *a priori* par le client).
- But : Diminuer le couplage entre la source d'une requête et son destinataire en permettant à plusieurs classes de traiter la requête.

Moyen

Construire une chaine d'objets traitants (*maillons*) et faire passer la requête à travers ces objets jusqu'à ce qu'elle soit traitée.

Pattern Chaîne de responsabilité (*Chain of Responsibility*)



- `Maillon` définit l'interface de gestion des requêtes ; implémente la gestion de la chaîne
- `MaillonConcret` répond à la requête pour lequel il est responsable et a accès à son successeur dans la chaine
- `Client` initie la requête auprès d'un `MaillonConcret`

Pattern Chaîne de responsabilité (*Chain of Responsibility*)

- **Diminuer le couplage** entre les éléments de la chaîne : un objet est couplé à son successeur et pas à tous les maillons
- Plus grande flexibilité dans l'affectation des responsabilités aux objet : on peut changer la chaîne à l'exécution et sans changements significatifs pour le client qui ne communique qu'avec une abstraction.
- Pas de garantie qu'une requête particulière soit traitée (il faut donc prévoir un destinataire *catch all*).
- Pattern souvent appliqué en conjonction avec Composite.
- Utilisé dans les gestionnaires de courrier électronique (spam, etc), interfaces graphiques pour gérer les événements (clic, entrée clavier) (cf. *Design Patterns : Elements of Reusable Object-Oriented Software*), ...

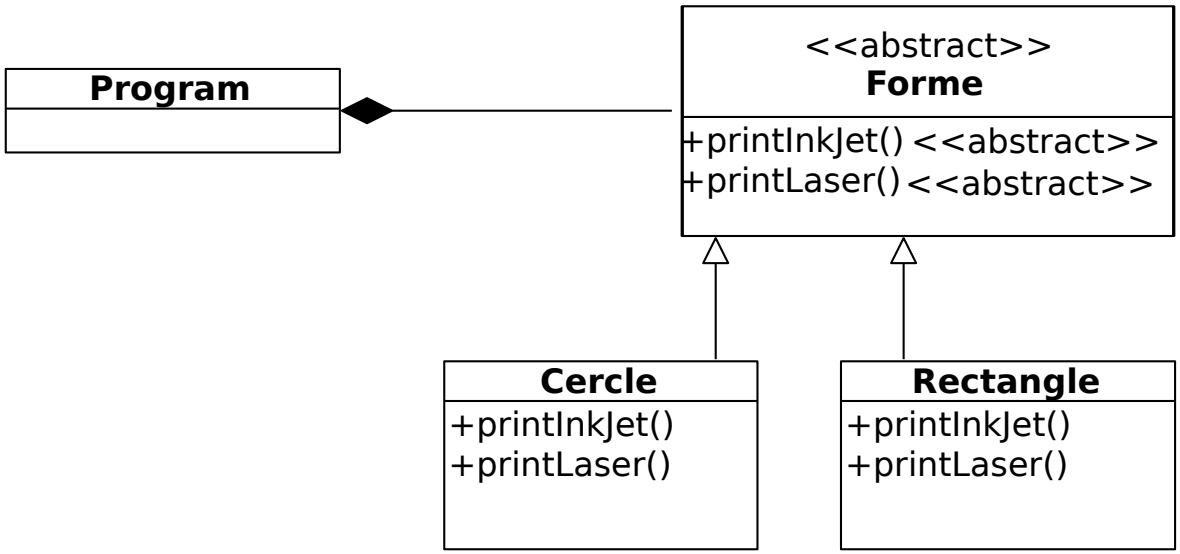
Plan

Pattern Visiteur (*Visitor*)

- Contexte : Des opérations, spécifiques à chaque élément, doivent être réalisées sur tous les éléments d'une structure
- But : Définir une nouvelle opération sur les éléments d'une structure sans changer les classes des éléments sur lesquels on opère
- Moyen : Séparer les éléments des opérations sur ces éléments (encapsulation de ce qui varie)

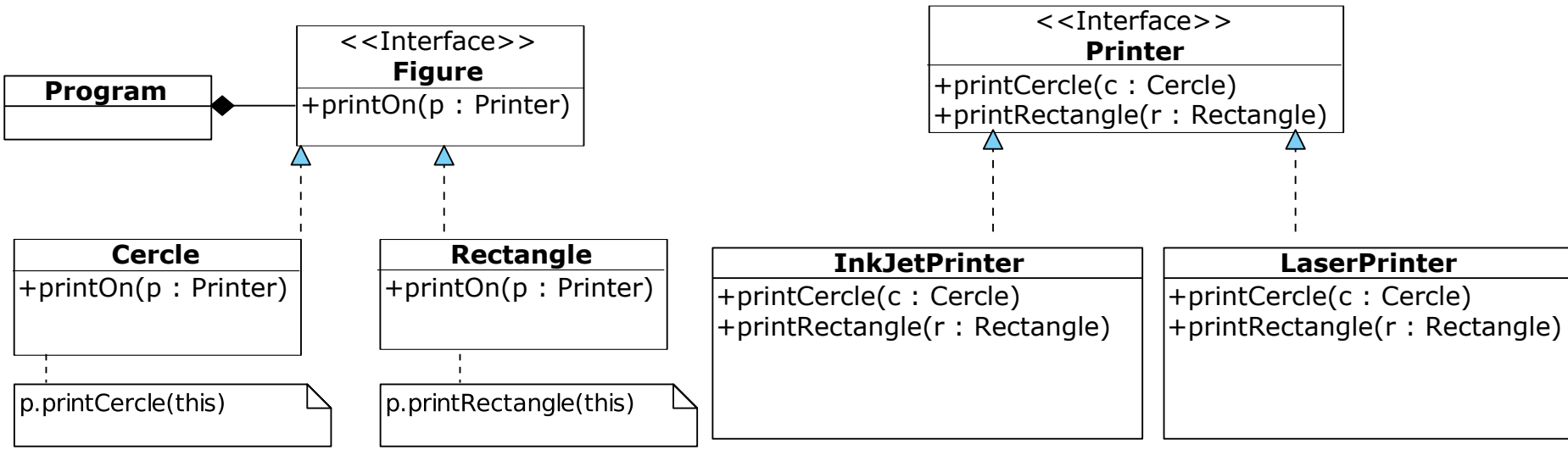
Exemple : les opérations sur les éléments (`print*()`) sont embarquées dans chaque élément.

- si on veut rajouter une opération (`printPS`), il faut modifier toutes les classes de la hiérarchie.

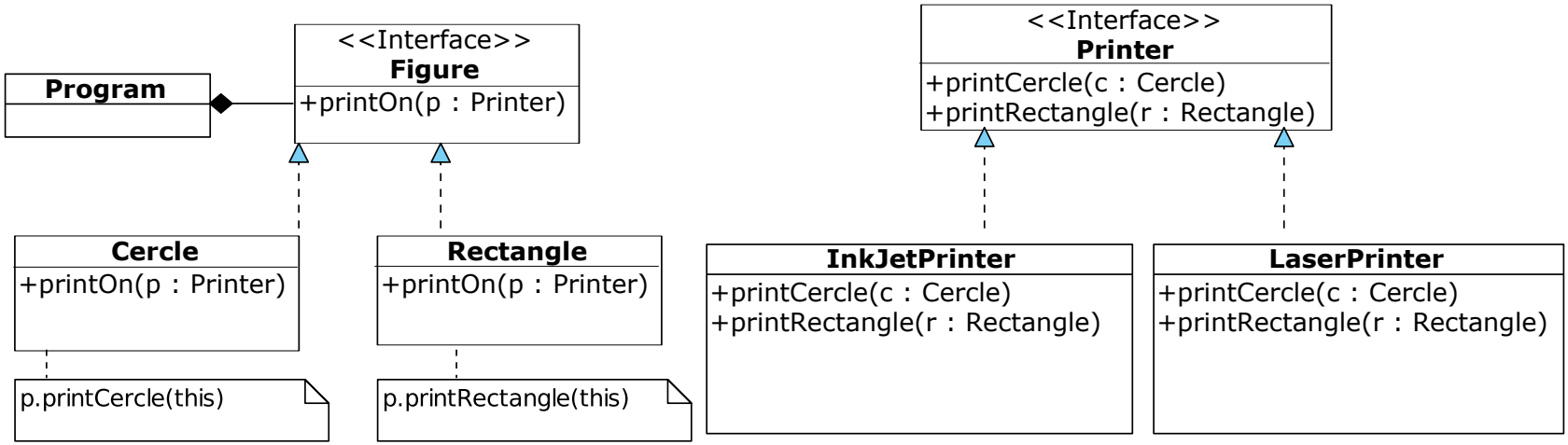


Visiteur (*Visitor*)

- Moyen : Séparer les éléments des opérations sur ces éléments
- **Encapsuler chaque opération dans un objet** (type `Printer`). Cet objet regroupe les implémentations d'une opération (`printInkJet`, ...) pour chaque élément de la structure (Cercle, Rectangle).

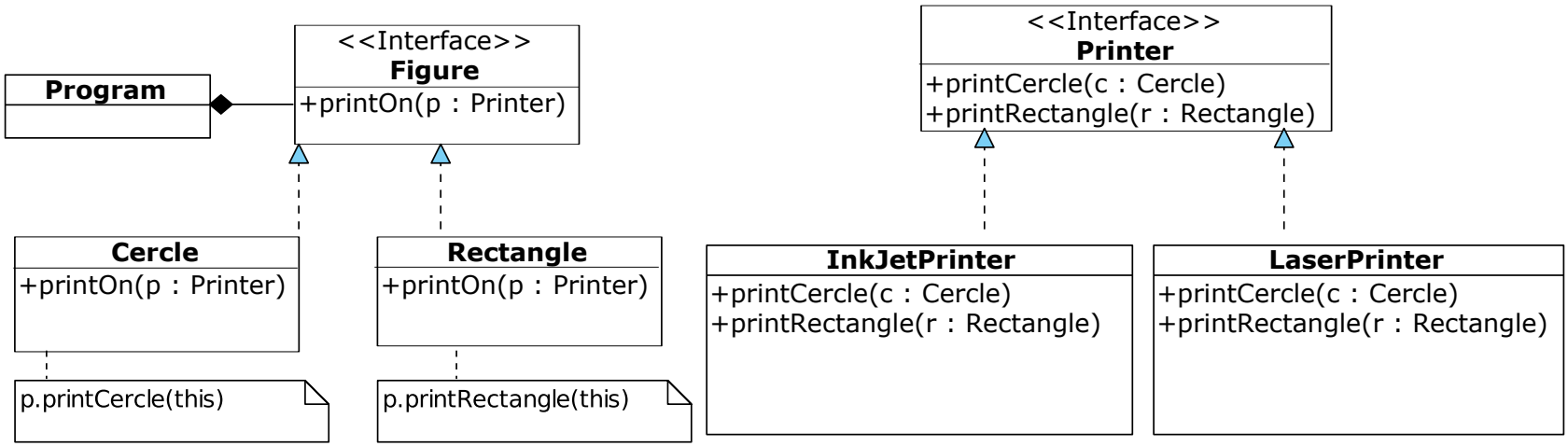


Visiteur (*Visitor*)



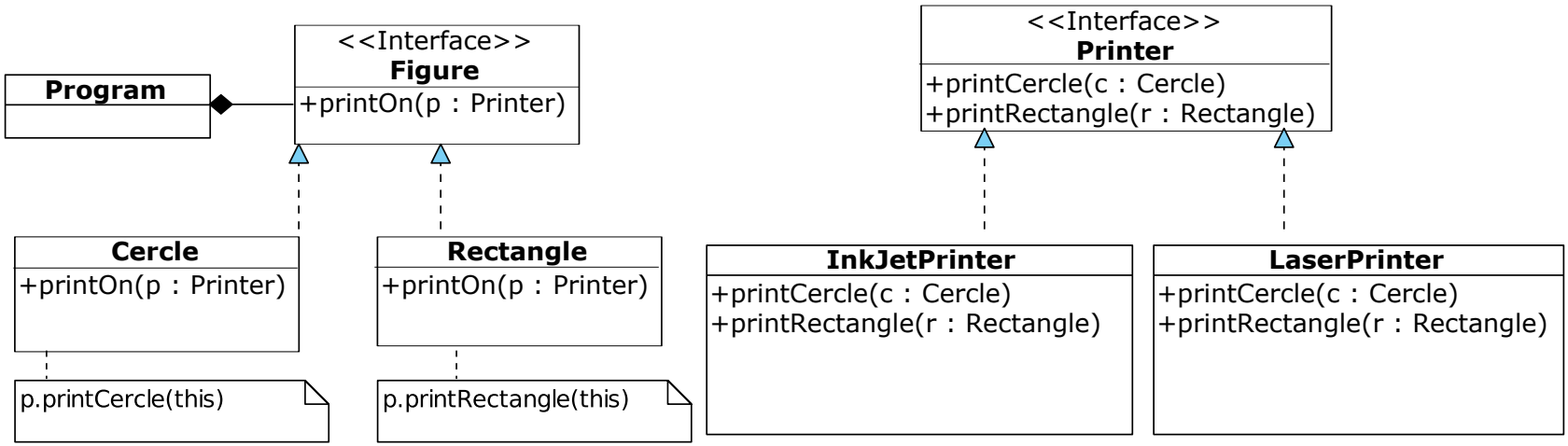

```
Figure fig= new Cercle();
Printer pr = new LaserPrinter();
fig.printOn(pr);
//appelle methode printCercle de la classe LaserPrinter avec fig en
      parametre.
```

Visiteur (*Visitor*)



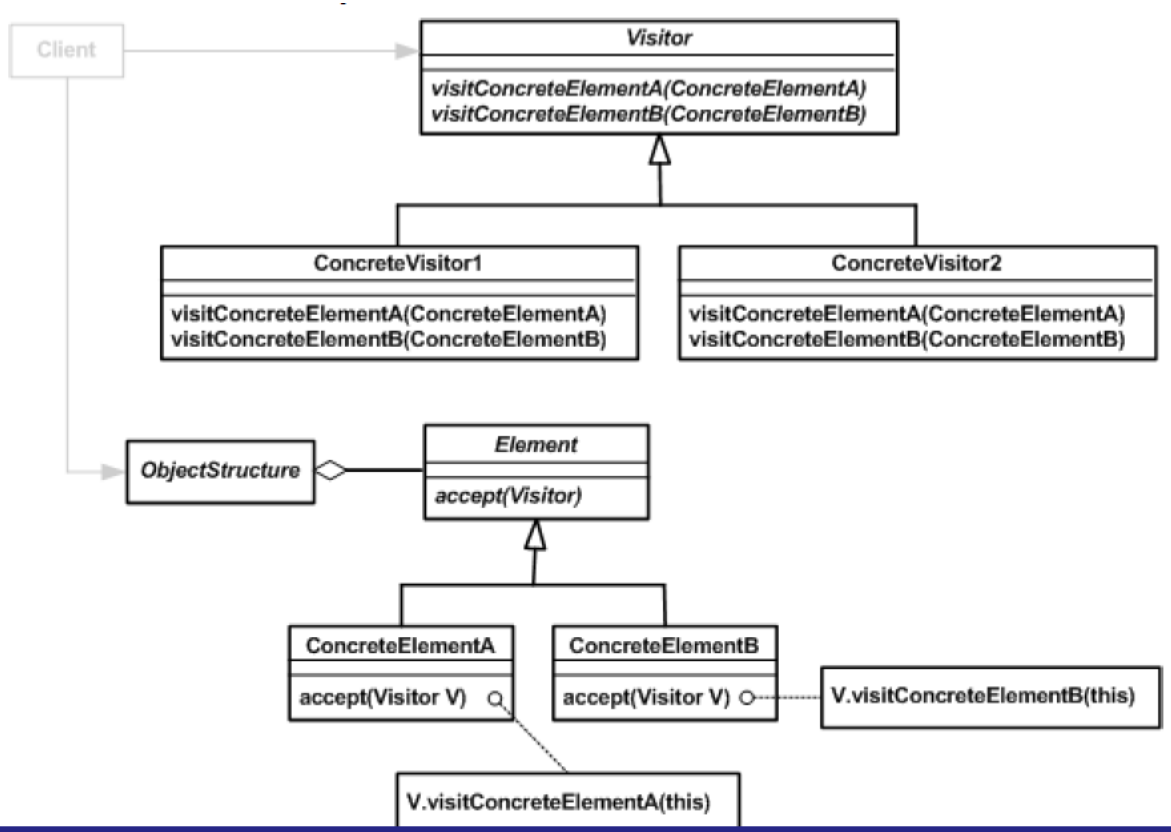
- Comment ajouter de nouvelles méthodes telles que `printPS` ?
- Comment ajouter de nouveaux éléments comme un `Losange` ?

Visiteur (*Visitor*)



- → L'ajout de nouvelles méthodes (`printPS`) est aisé et ne modifie pas les classes de la structure ; les opérations sont regroupées dans une classe (`PostScriptPrinter`).
- → Mais l'ajout de nouveaux éléments (`Losange`) est difficile

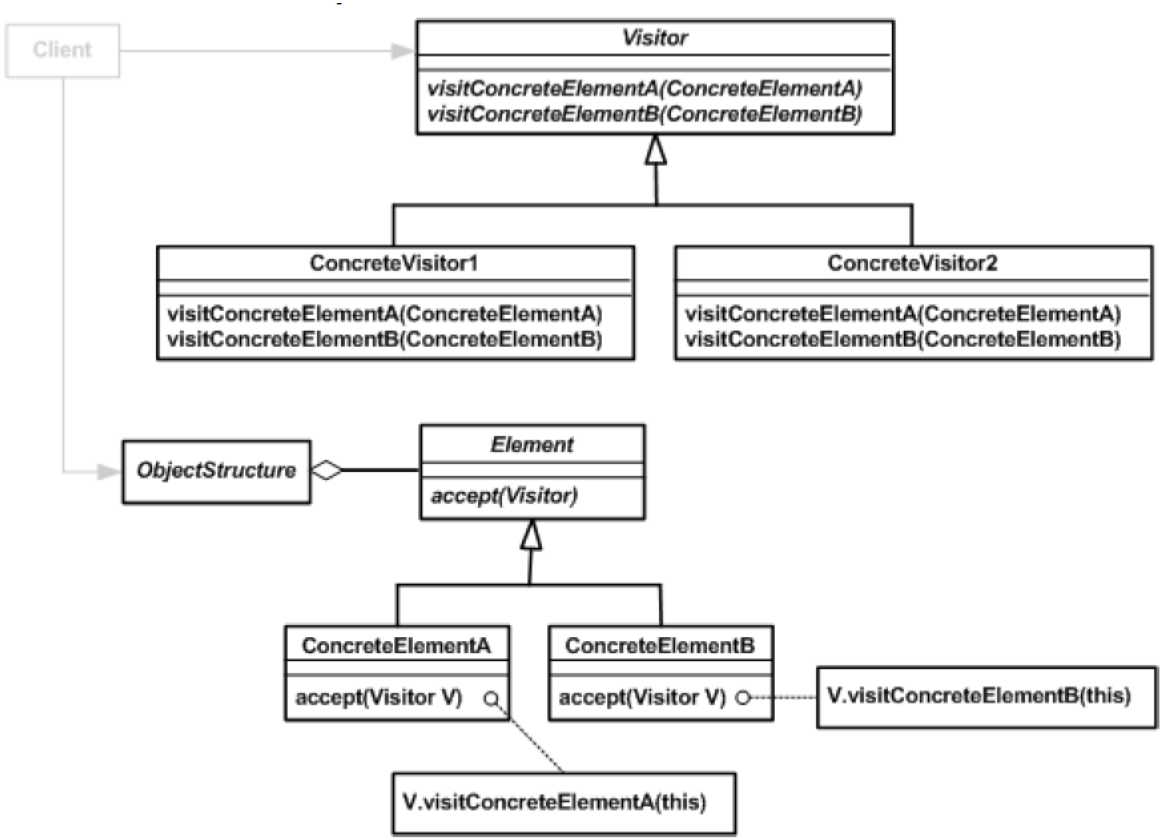
Visiteur (*Visitor*)



Moyen

- `Visitor` : déclare une méthode pour visiter chaque classe concrete (`ConcreteElement`) de la structure d'objet
- `ConcreteVisitor` : implémente chaque opération déclarée par le `Visitor`
- `Element`: défini une méthode *accept* qui prend un visiteur en paramètre
- `ConcreteElement`: implémente la méthode *accept* en transmettant la bonne requête au visiteur

Visiteur (*Visitor*)



Moyen

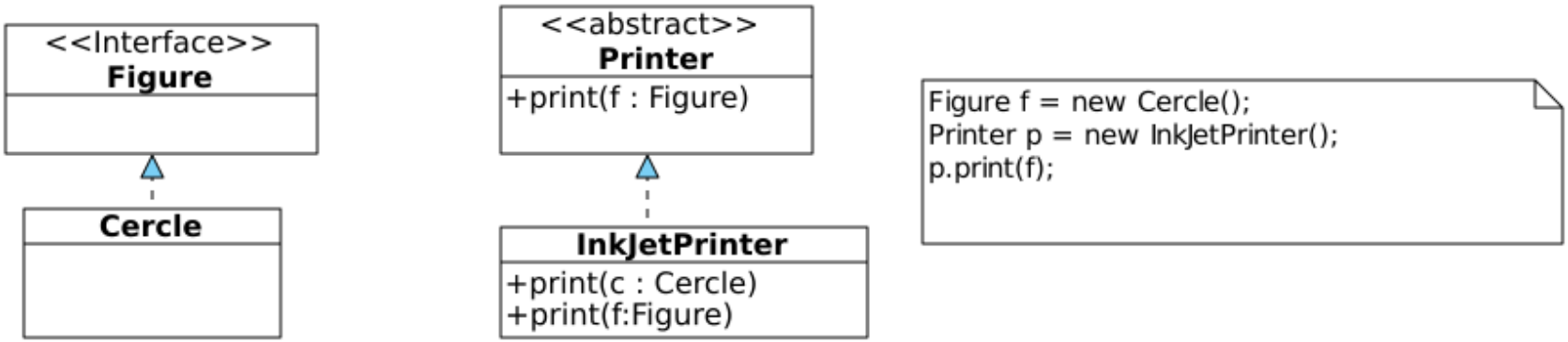
- Pour implémenter une opération, définir une classe qui implémente `Visitor` et qui regroupe les différentes actions de cette opération pour chaque type concret d'élément (`ConcreteElement`).

Visiteur (*Visitor*)

- L'ajout de nouvelles méthodes est aisé et ne change pas la structure d'objets
- Regroupement des différentes opérations
- `Visiteur` peut traverser des structures où les éléments sont de types complètement différents (différence importante avec le patron `iterateur`)
- MAIS l'ajout de nouvelles classes concrètes est difficile
- On peut utiliser le `Visiteur` pour appliquer des opérations sur une structure d'objets définie par le patron `Composite`
- Utilisation du **design pattern double dispatch**

Visiteur (*Visitor*)

Double Dispatch

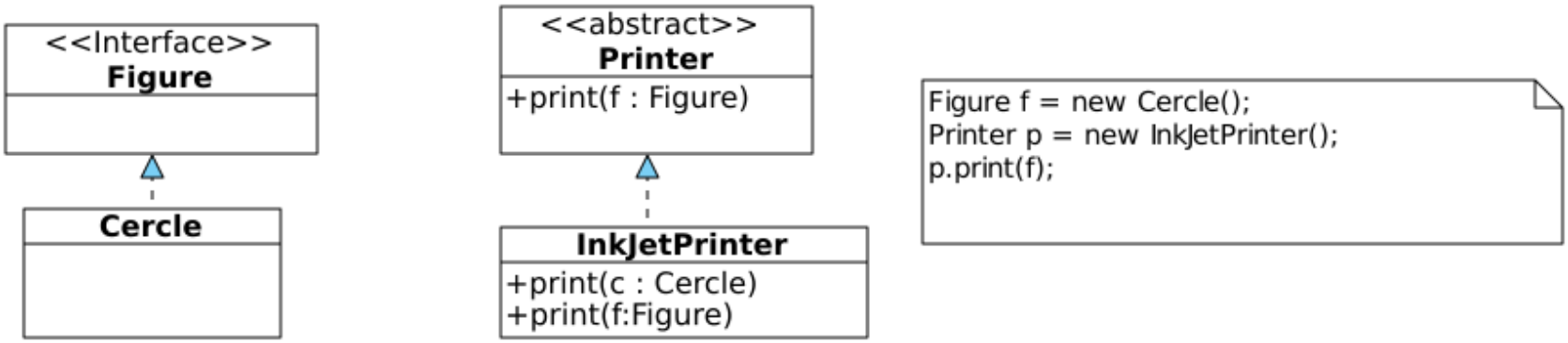


Simple Dispatch

- Déduire dynamiquement le type de l'objet appelant une méthode
- Java et C++(avec méthode *virtual*) implémentent le simple dispatch
- Exemple : La variable p a pour type statique `Printer` et pour type dynamique `InkJetPrinter`. Le compilateur pense que p est de type `Printer`. Le vrai type est déduit à l'exécution.

Visiteur (*Visitor*)

Double Dispatch



Double Dispatch

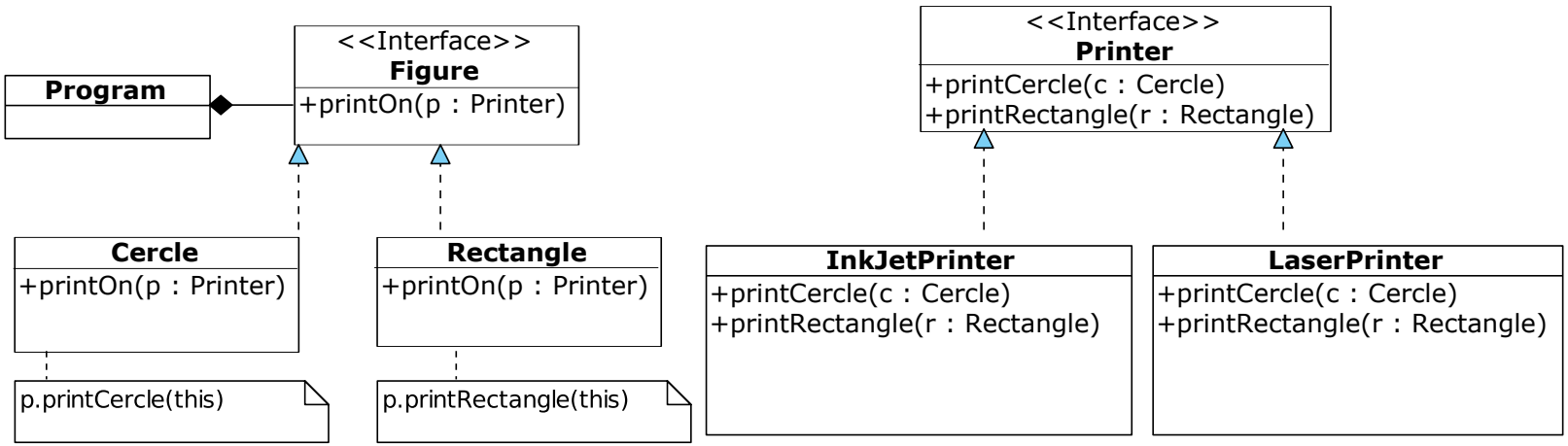
- Déduire dynamiquement le type des arguments de la méthode.
- Java et C++ n'implémentent pas le double dispatch.
- Exemple : la variable f a pour type statique `Figure` et pour type dynamique `Cercle`. Au moment du passage en argument, la méthode appelée est celle dont la signature est `print(f:Figure)`
- Solution proposée par `Visiteur` : inverser l'appelant et l'argument (`printOn(p:Printer)` dans `Cercle` appelle `p.print(this)`).

Visiteur (*Visitor*)

```
Figure fig= new Cercle();
Printer pr = new LaserPrinter();
fig.printOn(pr);
//appelle methode printCercle de la classe LaserPrinter avec fig en
      parametre.
```

Utilise deux appels polymorphiques :

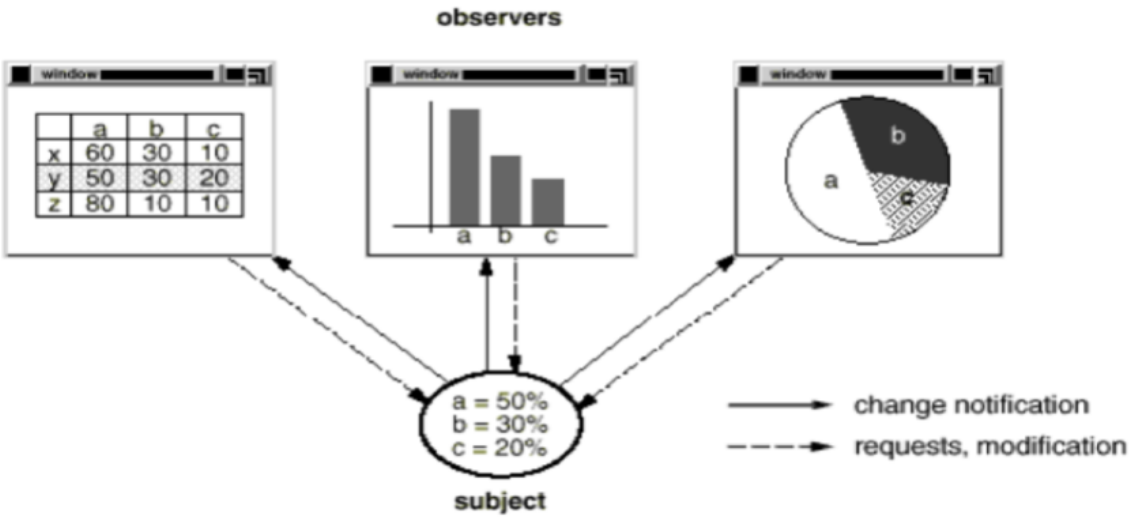
- `figure.printOn` : simple dispatch
- la méthode `printOn` appelle `printer.printCercle` : double dispatch



Plan

Pattern Observateur/Observé (*Observer*)

Problème : Comment faire savoir à un ensemble d'objets (observateurs ou *observer*) qu'un autre objet (observé/sujet/*observable*) dont ils dépendent a été modifié ?



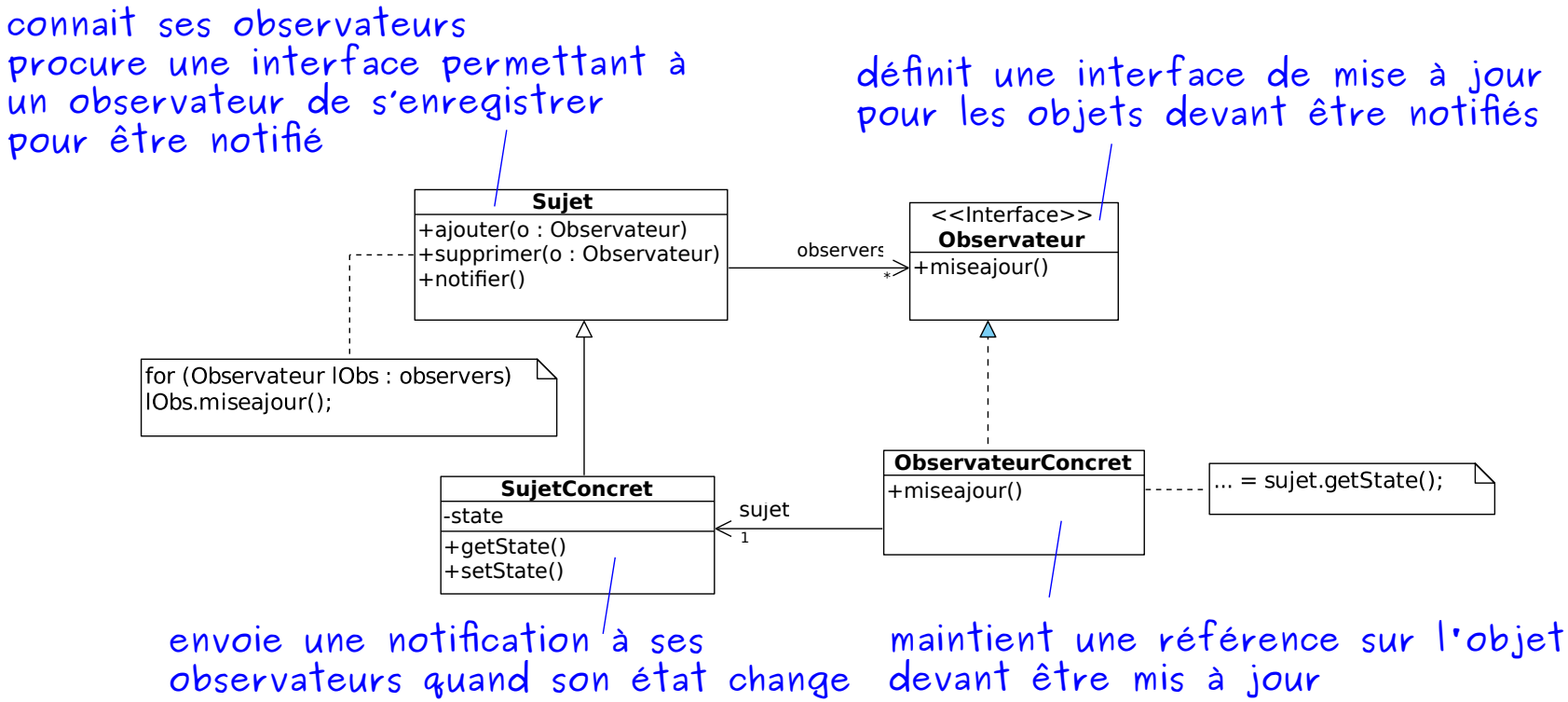
Buts

- Définir une **dépendance de un à plusieurs**.
- Assurer la cohérence entre des classes coopérant entre elles tout en maintenant leur **indépendance**
- Permettre à des objets de prendre en compte les changements d'autres objets à chaque fois qu'ils se produisent. La liste des objets surveillant ainsi les changements peut changer **dynamiquement**.

Pattern Observateur/Observé (*Observer*)

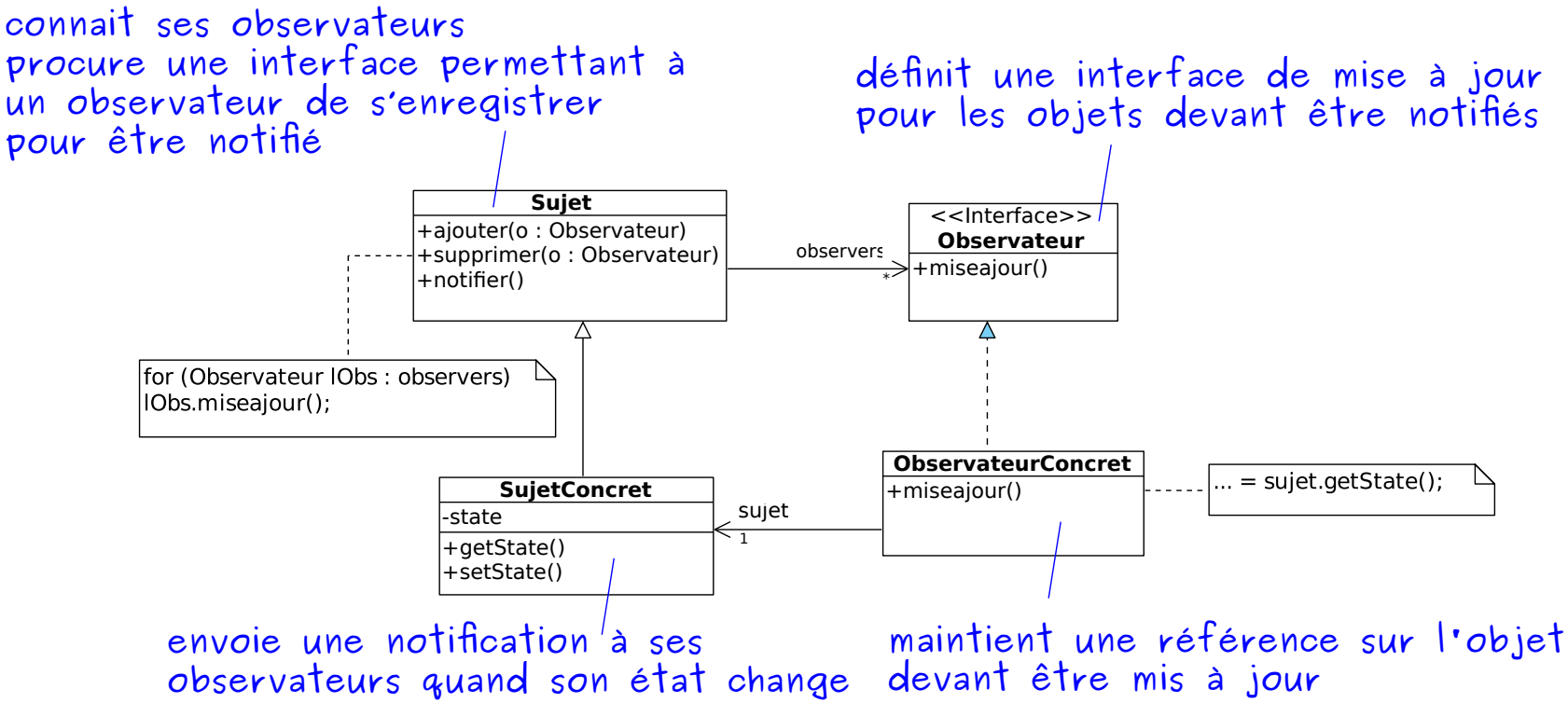
Moyen

Les objets observés (`Sujet`) enregistrent dynamiquement des observateurs (`Observateur`) et les notifient des changements lorsque cela est utile (`notifier()`).



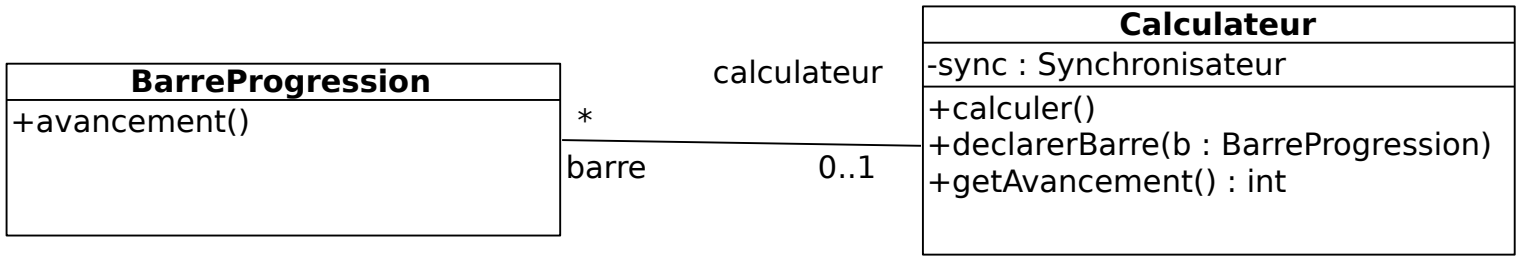
Pattern Observateur/Observé (*Observer*)

- Faible couplage entre `ObservateurConcret` et `SujetConcret`
- Les données de `Sujet` peuvent être "poussées" (dans *notifier*) ou "tirées" (avec des getters)
- Se retrouve dans de nombreuses API Java



Pattern Observateur/Observé (*Observer*)

La classe `BarreProgression` affiche l'état d'avancement du calcul (en pourcentage) d'un calculateur. Elle est associée à au plus un `Calculateur`. Le `Calculateur` offre une opération permettant de connaître le pourcentage d'avancement du calcul.



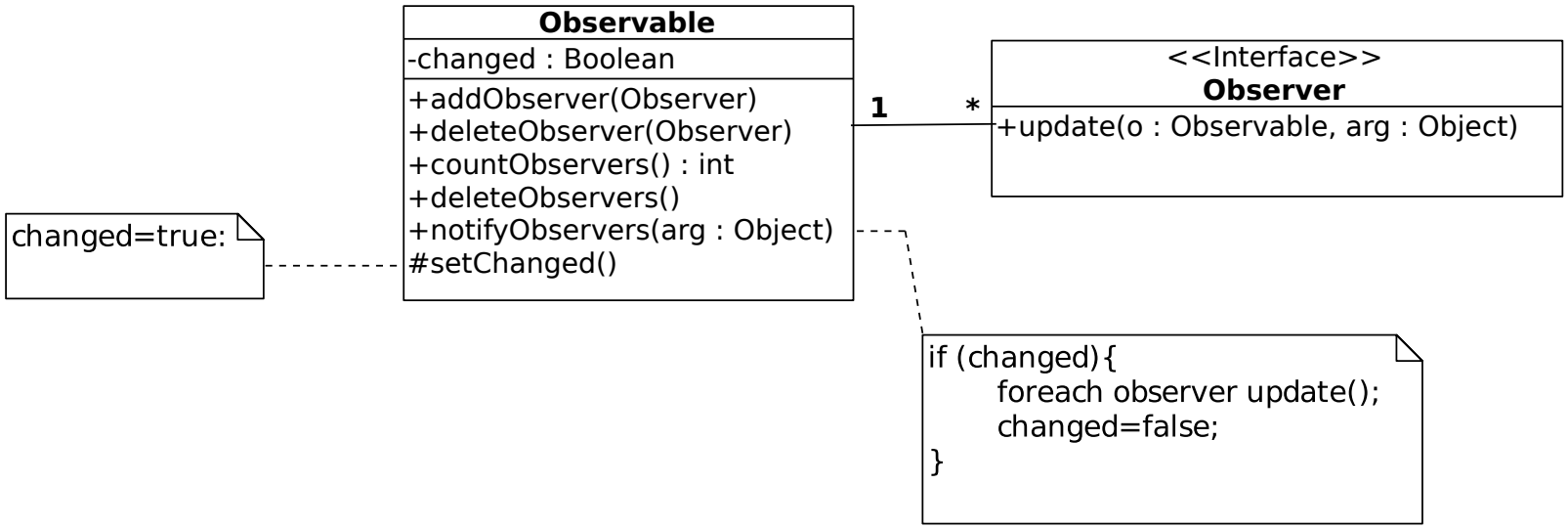
Fin TD2 SOLID

Nous souhaitons que ces deux classes soient dans deux packages différents. Proposez une solution pour casser le cycle de dépendances entre ces deux classes en appliquant le patron de conception `Observer`.

Pattern Observateur/Observé (*Observer*)

Implémentation Java

paquetage java.util

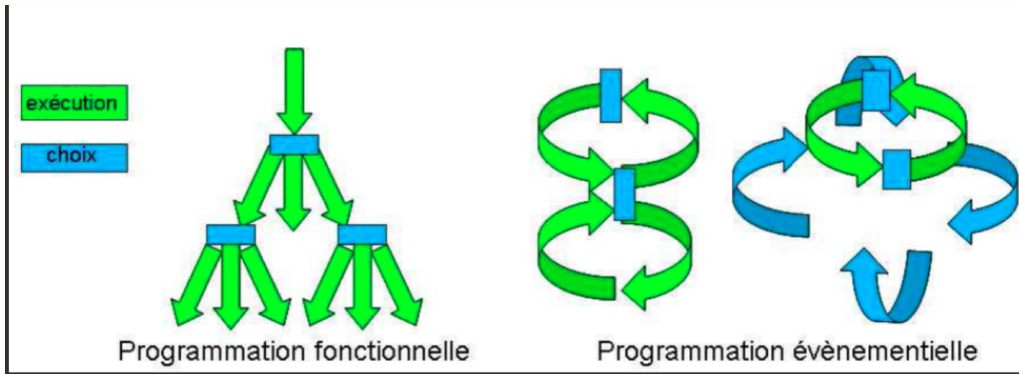


Toute classe qui veut être observé étend `Observable`.
Tout observateur doit implémenter l'interface `Observer`.
`public void addObserver (Observer o)` → pour inscrire un Observer
`protected void setChanged ( )` → sans appel de cette méthode, les 2 suivantes ne font rien !
`public void notifyObservers ( Object arg )` → notifie TOUS les Observer inscrits
`public void update (Observable o, Object arg)`
→ o le modèle qui a envoyé ce message. Cela permet de retrouver le modèle dont l'état a changé de sorte à le questionner pour connaître son état (tirer les informations sur l'état).
→ arg, un paramètre objet fournit par le modèle (par notifyObservers(Object arg)) pour renseigner sur la nature du changement (pousser les informations sur l'état).
Après qu'un `observable` change d'état, il doit invoquer `SetChanged()` puis `notifyObservers(...)` pour que tous ses `observer` soit notifiés du changement.

Programmation évènementielle avec Java

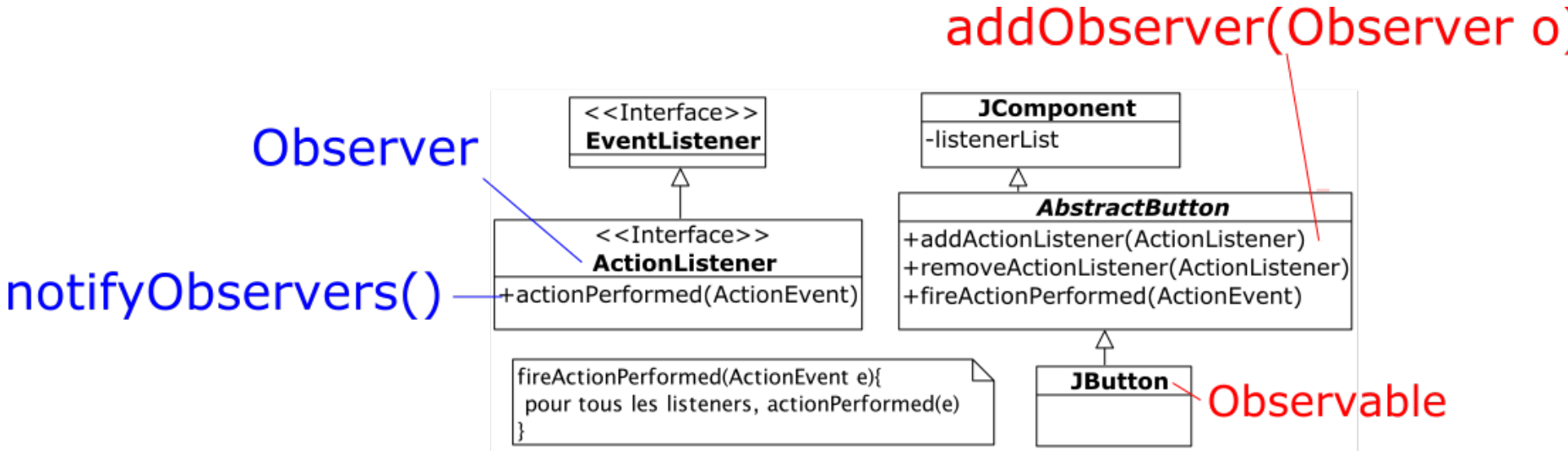
Paradigme de programmation dans lequel le programme est principalement défini par ses réactions face à différents événements

- Un évènement est un signal envoyé dans l'environnement
  - Composants Swing sources d'évènements
  - Classes d'évènements spécifiques à chaque type d'interaction (*MouseEvent*, *KeyEvent*, *ActionEvent*, *ListSelectionEvent*, ...)
- Un événement est capté par les objets abonnés à ce type d'événement
  - Un écouteur d'évènement (*listener*) doit s'enregistrer auprès de la source d'évènements pour être notifié.



Programmation évènementielle avec Java

- Un écouteur d'évènement doit implémenter une interface qui procure la méthode qui sera appelé par la source d'évènements
- Swing propose plusieurs interfaces pour les *listener*, selon le type d'évènement (*ActionListener*, *KeyListener*, *MouseListener*, ...)
- Forme spécialisé du pattern *Observer* : l'observateur est notifié (méthode `actionPerformed()` appelée) lorsque l'évènement observé a lieu et pas lorsqu'un observé change d'état.



Plan

Patterns de création

- Ils définissent comment faire l'instanciation et la configuration des classes et des objets.
- Ils rendent le système indépendant de la manière dont les objets sont créés, composés et représentés :
  - Encapsulation de la connaissance des classes concrètes à utiliser
  - Cacher la manière dont les instances sont créées et combinées

Plan

Pattern Singleton

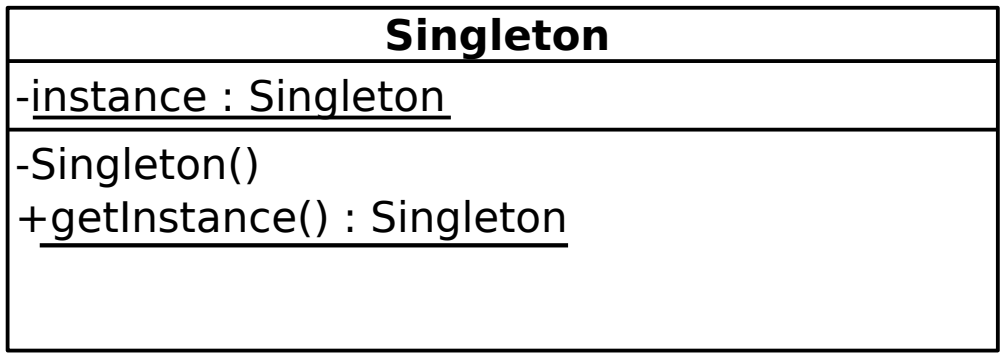
- Buts : Garantir qu'une classe ne possède qu'une seule instance et fournir un accès global à cette instance.

- Unicité de l'instance :
- Accès global à l'instance :

Pattern Singleton

- Buts : Garantir qu'une classe ne possède qu'une seule instance et fournir un accès global à cette instance.

- Unicité de l'instance :
  - déclarer le constructeur privé : impossible de créer une instance de la classe à l'extérieur de la classe elle-même et pas de classes filles.
- Accès global à l'instance :
  - une instance est stockée dans la classe elle-même et accessible par une méthode statique.

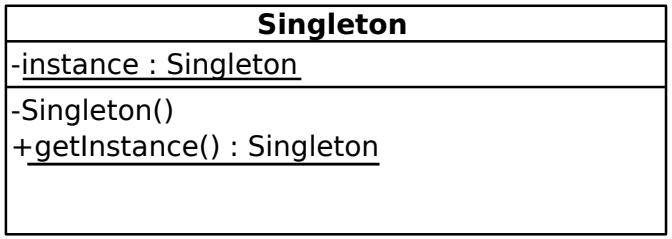


Plusieurs implémentations possibles

Pattern Singleton : Version 1

- Créer cette instance en tant que champ statique dans la classe
- Instance créée au chargement de la classe même si jamais utilisée

```
class Singleton {
  private static Singleton instance = new Singleton();
  private Singleton() { //... }
  public static Singleton getInstance() {
   return instance;}
  }
```

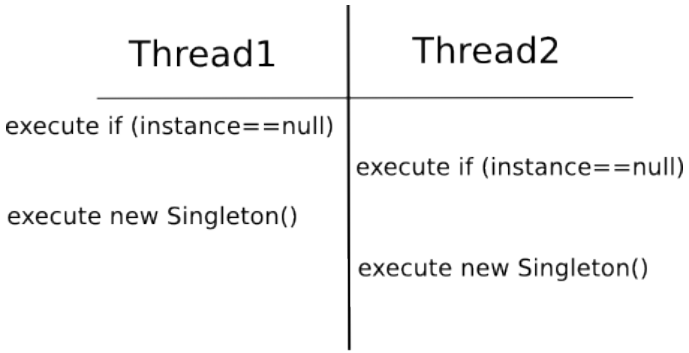


Pattern Singleton : Version 2

- Créer cette instance en tant que champ statique dans la classe :
- *lazy-initialization* : instance créée lors de sa première utilisation

```
class Singleton {
  private static Singleton instance ;
  private Singleton() { //... }
  public static Singleton getInstance() {
   if (instance == null) instance = new Singleton();
   return instance;}
  }
```

Si plusieurs threads accèdent au Singleton, l'unicité n'est pas garantie :



Pattern Singleton : Version 3

- Créer cette instance en tant que champ statique dans la classe :
- *lazy-initialization* : instance créée lors de sa première utilisation
- méthode d'accès synchronisée

```
class Singleton {
  private static Singleton instance ;
  private Singleton() { //... }
  public static synchronized Singleton getInstance() {
   if (instance == null) instance = new Singleton();
   return instance;}
  }
```

Accès limité pour la création mais aussi pour la lecture ...

Plan

Méthode Fabrique & Fabrique Abstraite

L'instanciation peut souvent entraîner des problèmes de couplage.

Objectifs généraux

Les fabriques gèrent les détails de la **création des objets** :

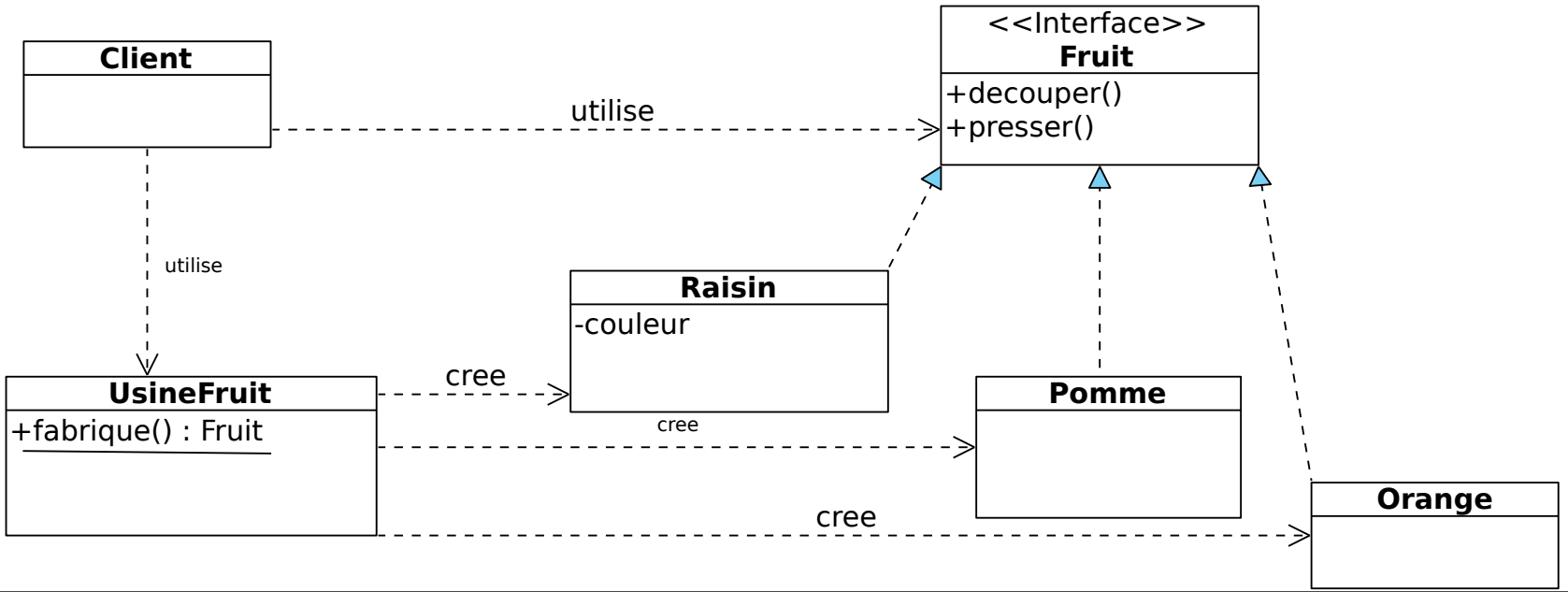
- Séparer l'instanciation en l'encapsulant dans des classes dédiées
- Déléguer l'instanciation des objets d'un certain type à des classes dédiées
- Cacher la classe concrète des objets fabriqués au profit de leur classe abstraite.
- Permettre à un client de créer des objets sans savoir leur type précis.

On rend ainsi **tout le code excepté celui de la fabrique indépendant des implémentations** en recourant aux classes abstraites et interfaces.

Méthode Fabrique

Fabrique Simple

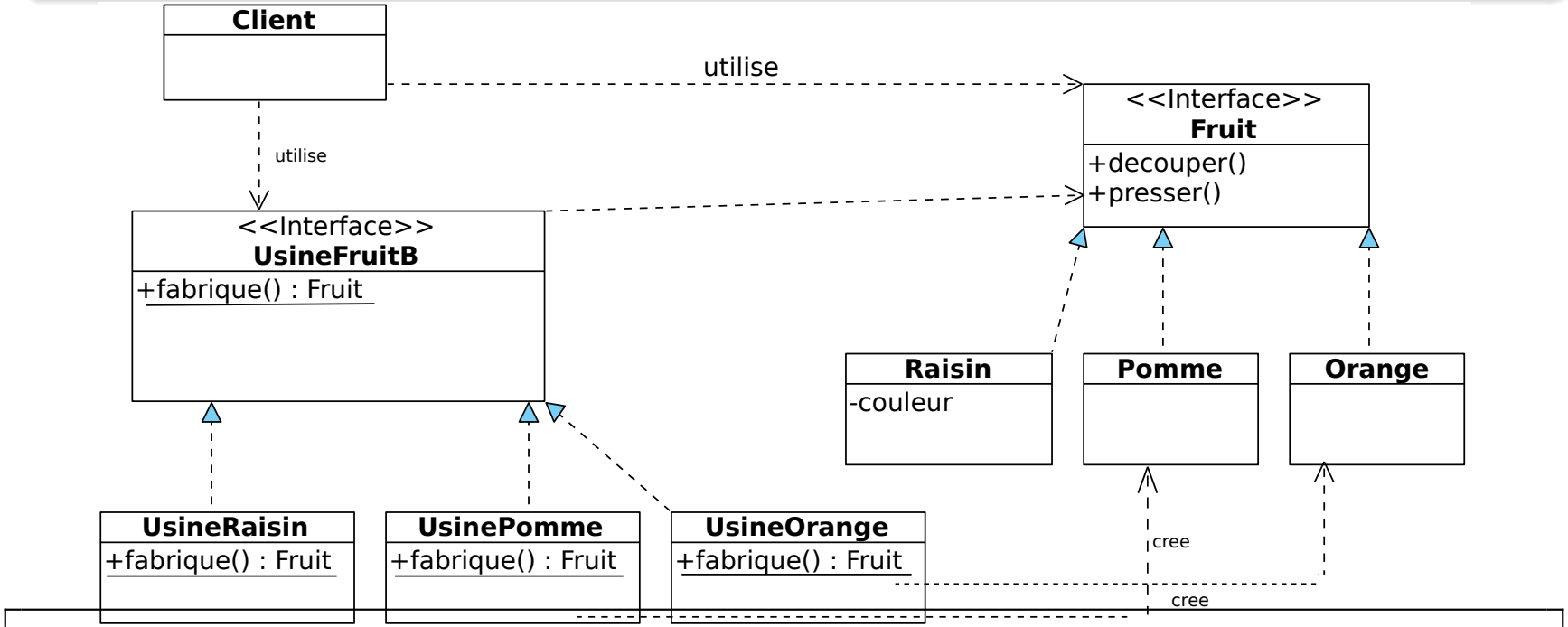
- Utiliser une seule classe pour **créer des instances de différents types**.
- En général, la méthode *fabrique* est paramétrée pour choisir l'instance en fonction des paramètres.




```
Fruit f = UsineFruit.fabrique(...);
f.decouper();
f.presser();
```

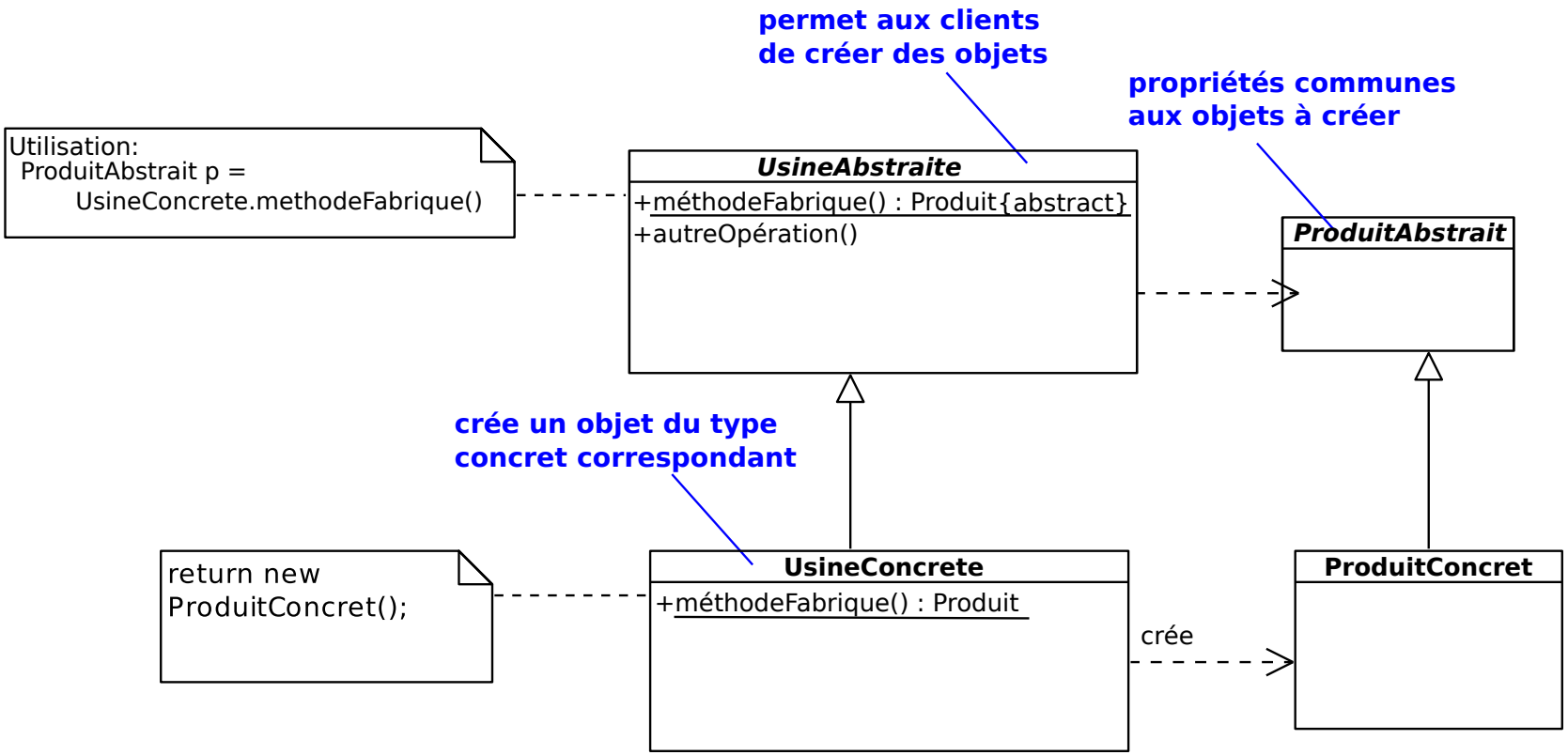
Méthode Fabrique

- Abstraire la solution *Fabrique Simple*
- Un produit → une classe concrète pour le fabriquer




```
Fruit f = UsineRaisin.fabrique();
f.decouper();f.presser();
Fruit ff = UsinePomme.fabrique();
ff.decouper();
ff.presser();
```

Méthode Fabrique (*Factory Method*)

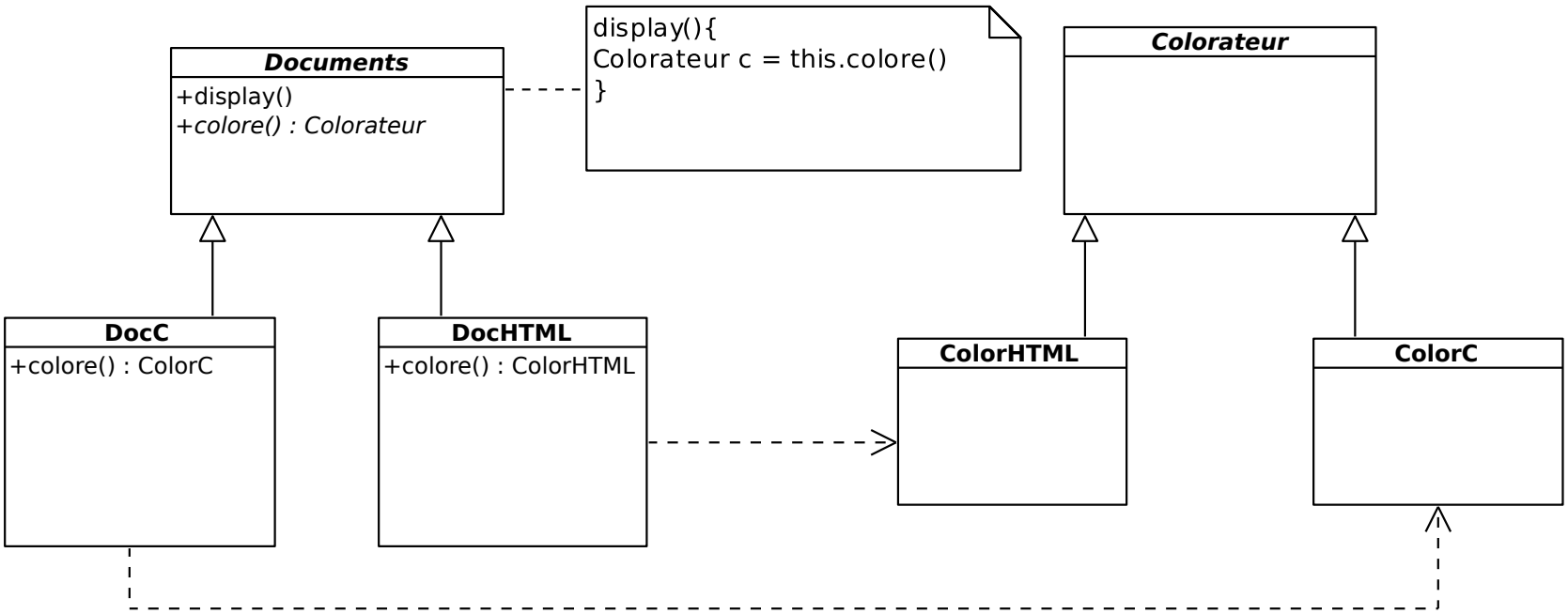


- Classes qui fabriquent (usines) :
  - `UsineAbstraite` : regroupe les parties communes à toutes les `UsineConcrete` et définit la *méthode abstraite fabrique*
  - `UsineConcrete` : chaque usine concrète crée un produit du type concret correspondant (implémente la *méthode fabrique*)
- Objets à créer (produits) : `ProduitAbstrait` : regroupe les parties communes à tous les `ProduitConcret`

Méthode Fabrique (*Factory Method*)

Exemple

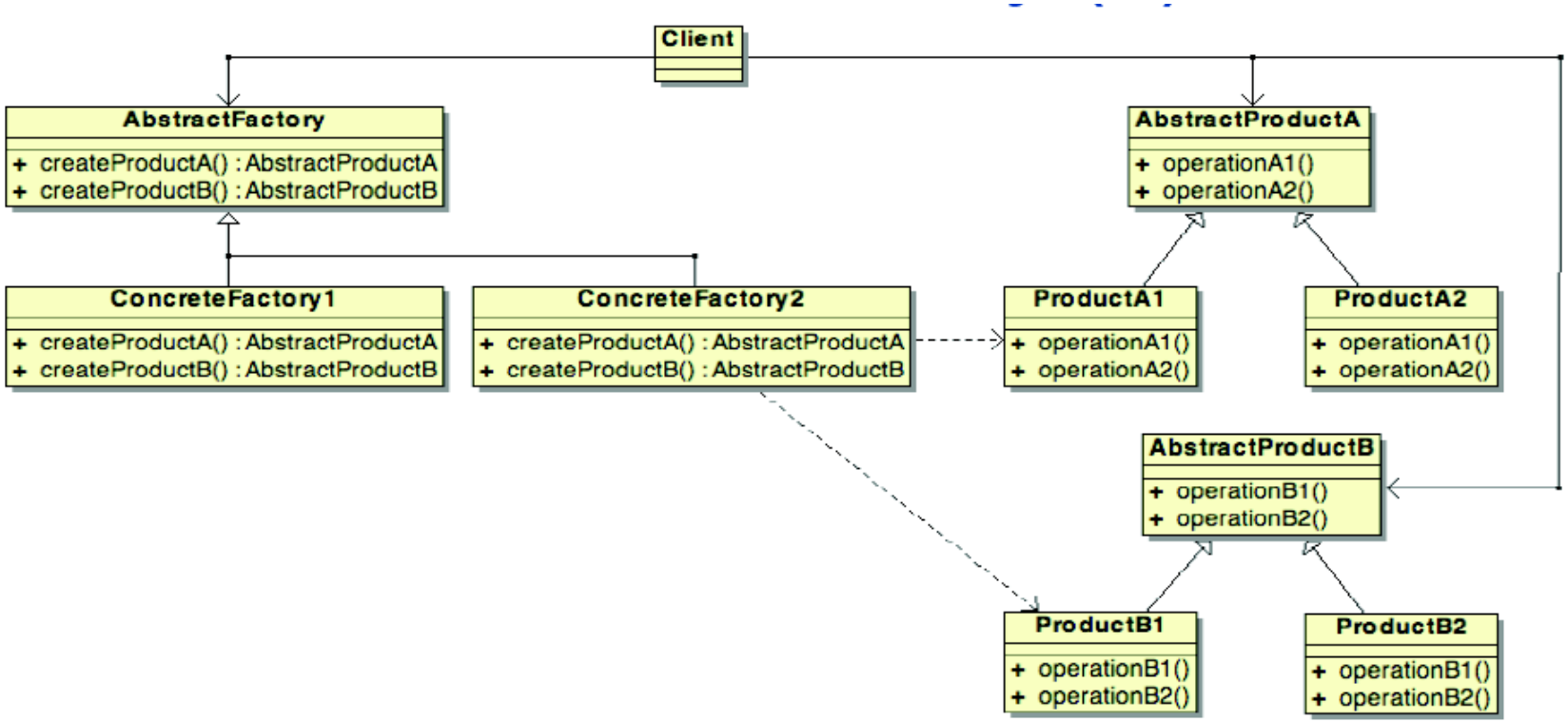
Un éditeur de texte a une fonction de coloration syntaxique selon le type de document affiché (HTML, C, Python, ...). Ainsi en fonction de la nature du document, la commande `Display()` doit utiliser le bon colorateur syntaxique.




```
Documents doc = new DocC(...);
doc.display();
```

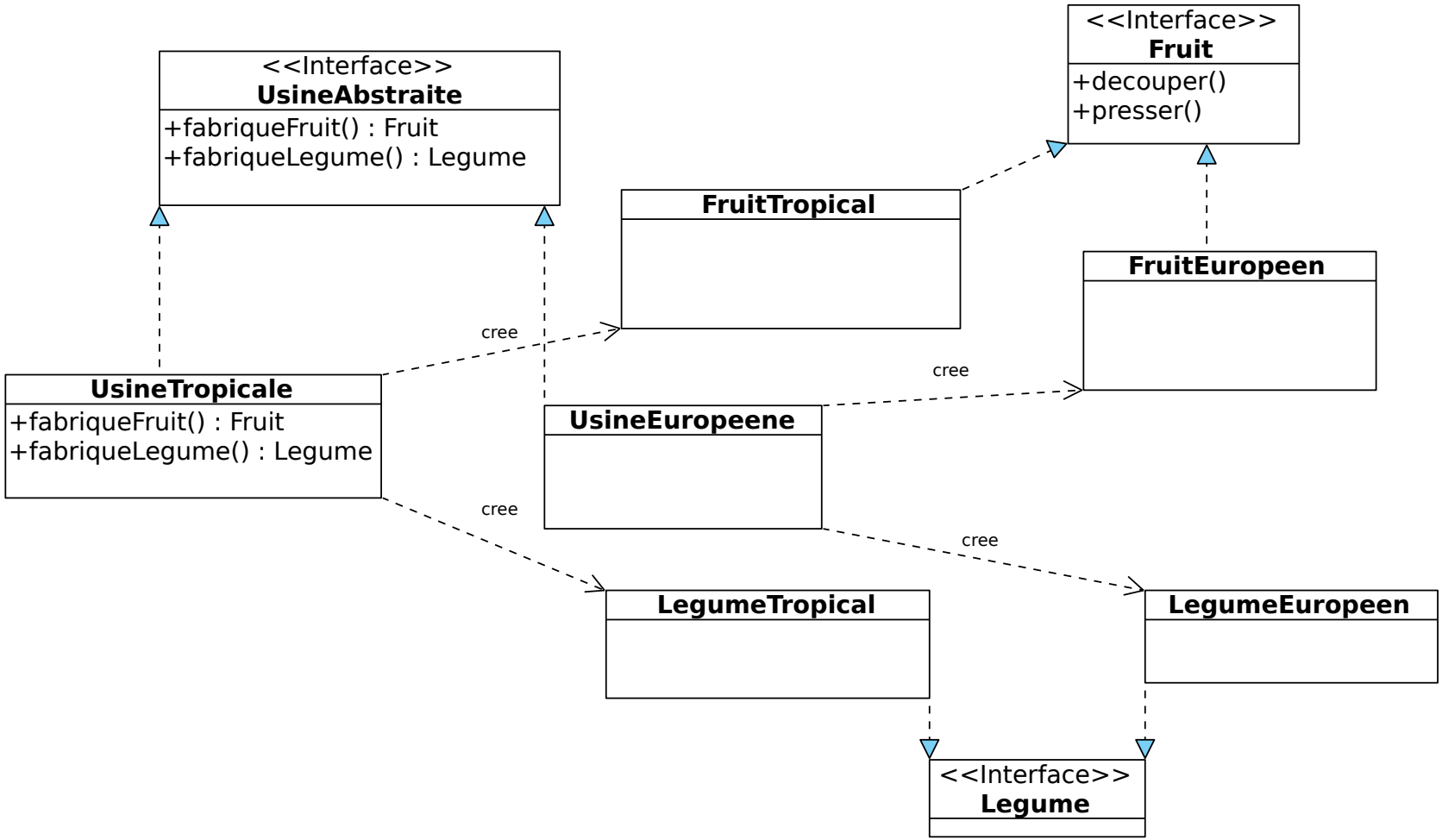
Fabrique Abstraite (*Abstract Factory*)

- Modèle de `fabrique` complètement général
- Fabriquer des fabriques : plusieurs fabriques peuvent être regroupées en une fabrique abstraite permettant d'instancier des objets dérivant de plusieurs types abstraits différents.
- Deux dimensions de classifications



Fabrique Abstraite (*Abstract Factory*)

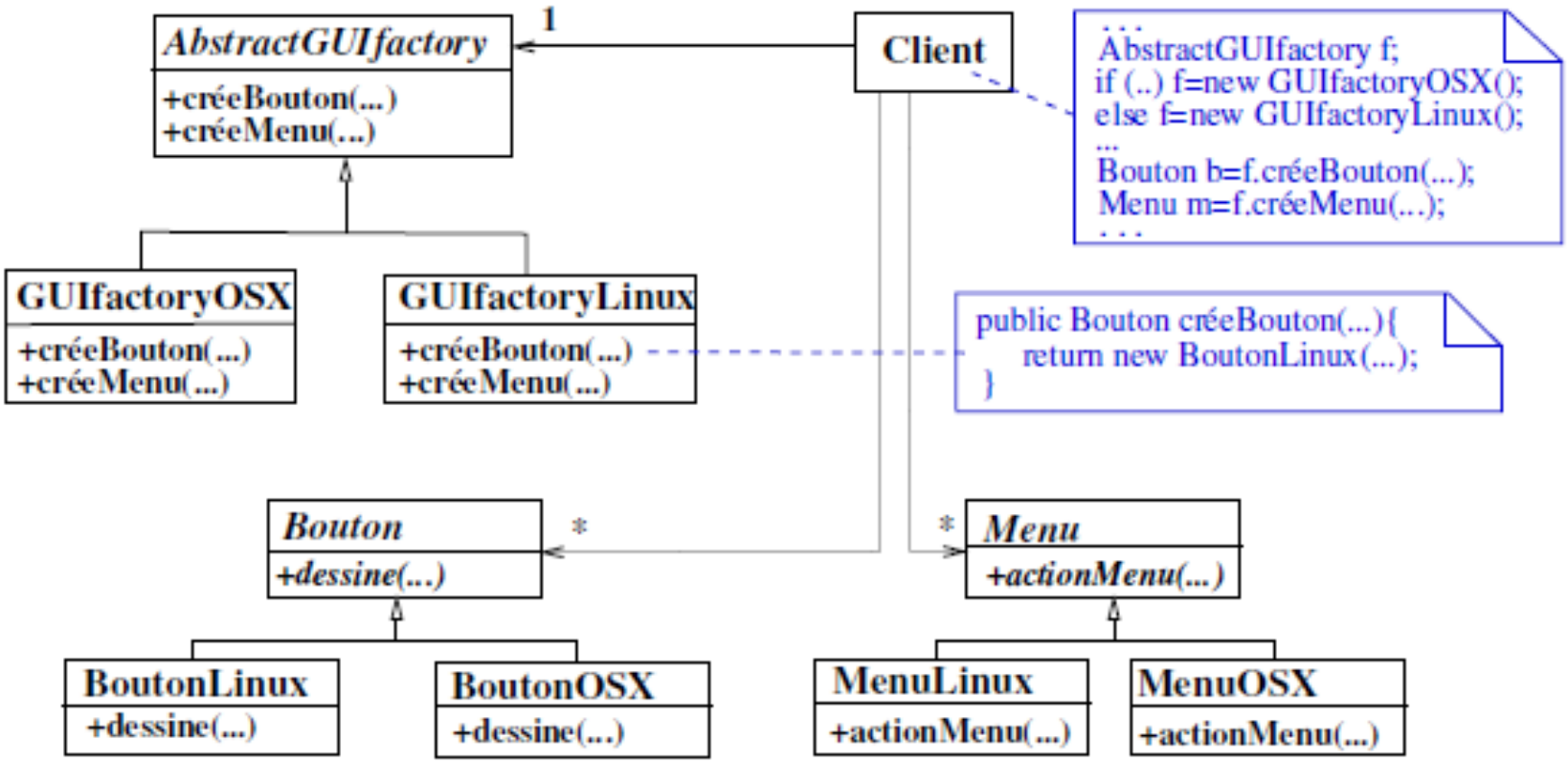
Exemple



Fabrique Abstraite (*Abstract Factory*)

Exemple

- Créer une interface graphique avec widgets (boutons, menus, ...)
- Point de variation : OS (Linux, OSX, Windows)



Méthode Fabrique & Fabrique Abstraite

- Déléguer l'instanciation des types concrets à des classes
  - Introduit une **indirection** pour la construction des données : isole le Client des implémentations des produits
  - Facilite l'échange de familles de produits
  - La mise en place de nouveaux produits dans `l'AbstractFactory` n'est pas aisé
- Une méthode fabrique est plus flexible qu'un constructeur : on peut construire des objets des sous-classes
- **Les fabriques sont souvent uniques dans un programme : on utilise alors le patron de conception Singleton pour les implémenter.**
- Exemple : `http://design-patterns.fr/fabrique-en-java`

Plan

Références

- *Design Patterns. Elements of Reusable ObjectOriented Software.*, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994.
- Craig Larman, *Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*
- *Design Patterns pour Java*, Laurent Debrauwer, 2009
- *Les Design Patterns en Java - Les 23 modèles de conception fondamentaux*, Steven John Metsker, William C. Wake, 2006.
- http ://c2.com/cgi/wiki ?CategoryPattern
- http ://sourcemaking.com/design_patterns
- ...