

Patrons de comportement (PARTIE VII - Patrons de conception)

Bruno Bachelet
Christophe Duhamel
Luc Touraille

Patrons de comportement (1/4)

- Abstraction du comportement
 - ❑ Structure algorithmique
 - ❑ Affectation de responsabilités aux objets
 - ❑ Communication entre objets

- Niveau classe
 - ❑ Utilisation de l'héritage
 - ❑ Répartition du comportement

- Niveau objet
 - ❑ Utilisation de la composition
 - ❑ Coopération d'objets pour effectuer une tâche

- Permet l'assemblage de composants
 - Pour obtenir une fonctionnalité plus élaborée
 - Algorithmes vus comme des objets

- Comment les composants communiquent ?
 - Niveau de connaissance des pairs
 - Références explicites les uns envers les autres
 - Perte des références, utilisation d'un intermédiaire
 - Propagation d'un message
 - Délégation
 - Transmission
 - Messages vus comme des objets

Patrons de comportement (3/4)

- Chaîne de responsabilité / *Chain of Responsibility*
 - Transmettre une requête de proche en proche jusqu'à traitement
- Commande / *Command*
 - Encapsuler une action dans un objet
- Interpréteur / *Interpreter*
 - Représenter la grammaire d'un langage
- Itérateur / *Iterator*
 - Fournir un accès séquentiel aux éléments d'un agrégat
- Médiateur / *Mediator*
 - Encapsuler la manière d'interagir d'un groupe d'objets

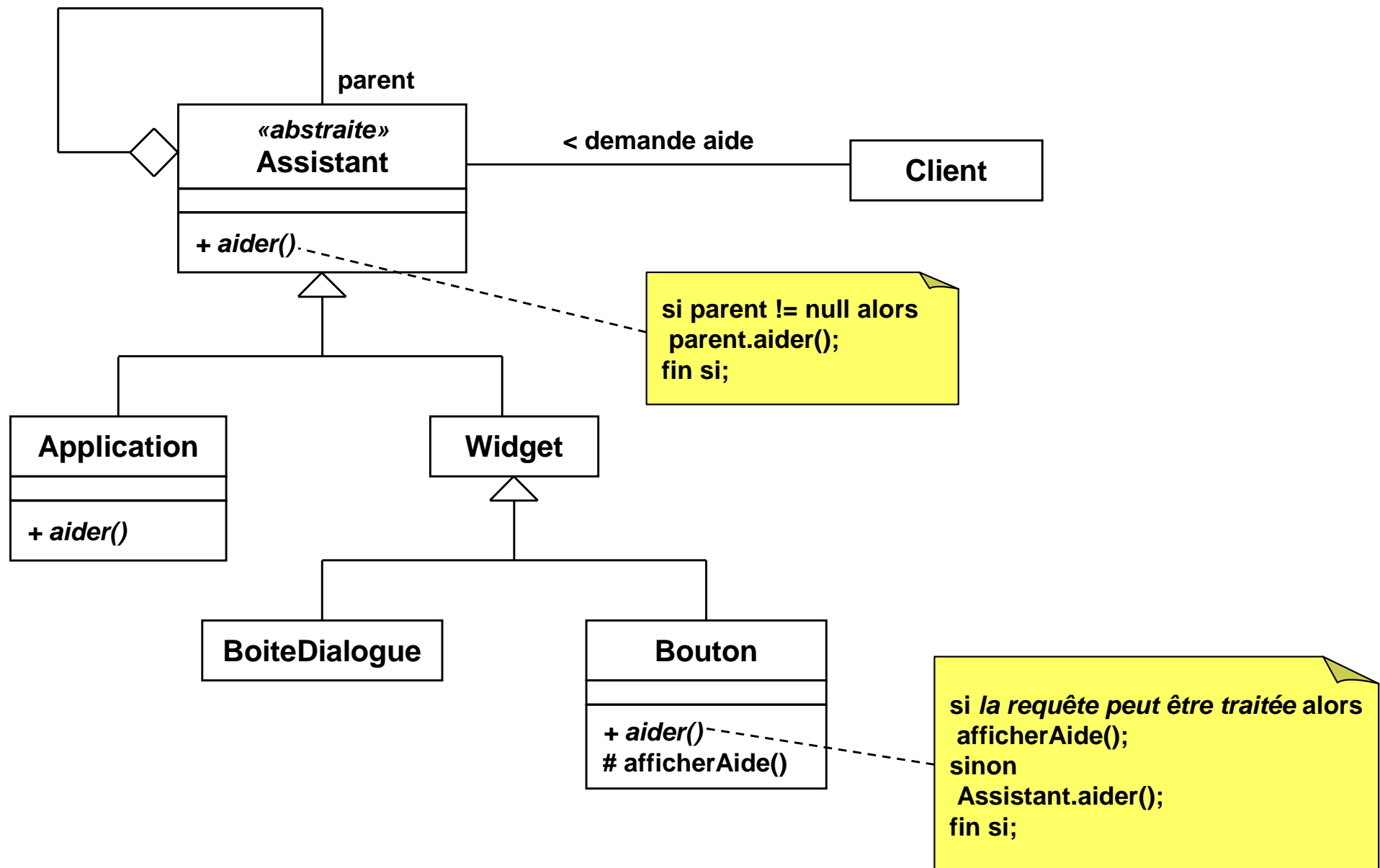
Patrons de comportement (4/4)

- Memento / *Memento*
 - Capturer et externaliser l'état d'un objet
- Observateur / *Observer*
 - Synchroniser plusieurs objets sur l'état d'un autre objet
- Etat / *State*
 - Changer le comportement d'un objet en fonction de son état
- Stratégie / *Strategy*
 - Rendre les algorithmes d'une même famille interchangeables
- Méthode patron (Patron de méthode) / *Template Method*
 - Spécialiser un algorithme sans changer sa structure générale
- Visiteur / *Visitor*
 - Représenter une opération à appliquer sur un ensemble d'objets

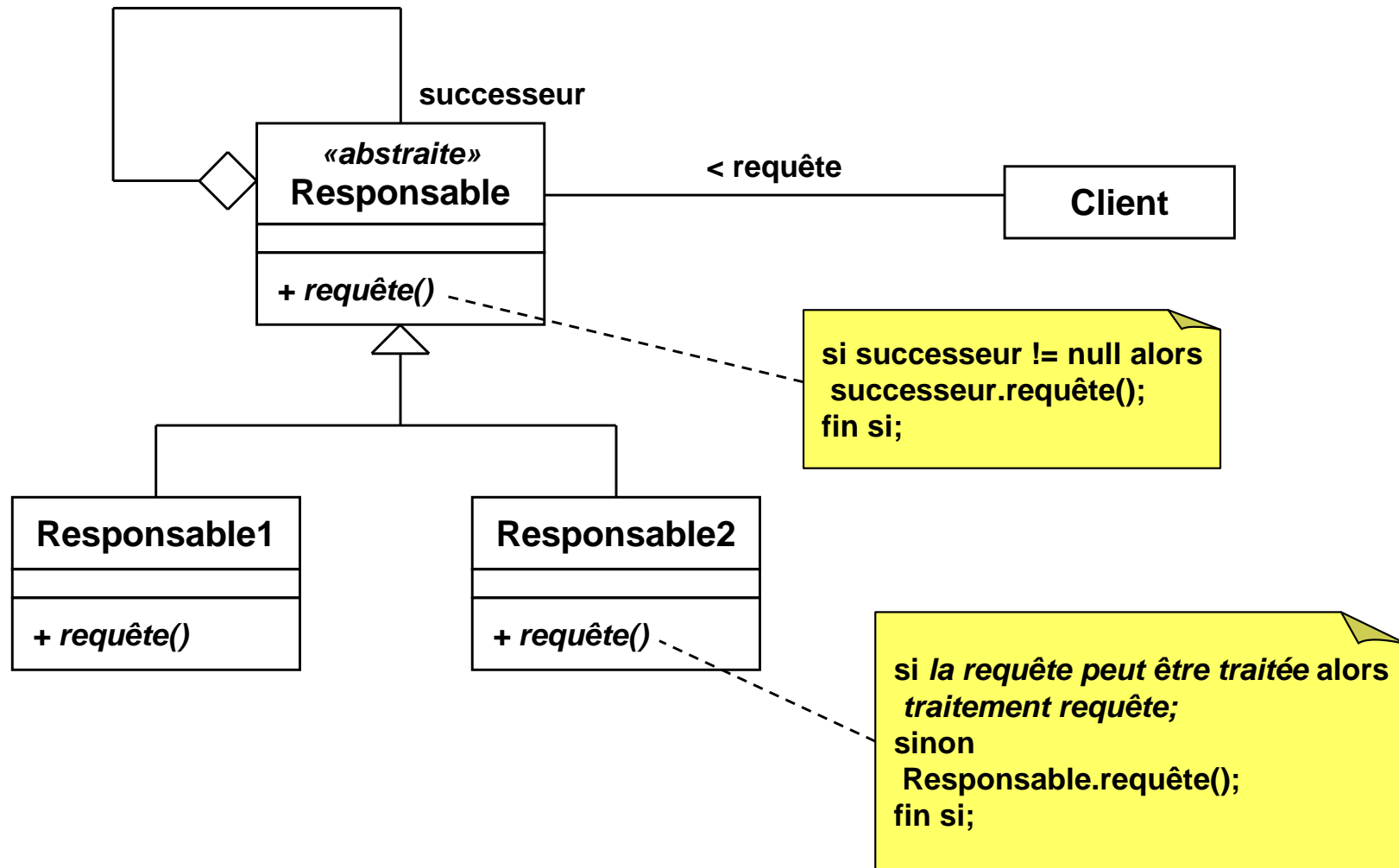
Chaîne de responsabilité (1/4)

- «*Chain of Responsibility*»
- Objectif
 - Transmettre une requête de proche en proche jusqu'à traitement
 - Eviter le couplage entre l'émetteur et les receveurs potentiels
- Principe
 - La requête est transmise de receveur en receveur
 - Jusqu'à ce qu'elle soit traitée
 - Les receveurs ont la même interface pour répondre à la requête
 - Référence sur un successeur pour former une chaîne
 - Le client transmet une requête au premier de la chaîne
- Motivation
 - Action contextuelle: demande d'aide dans une application
 - Requête reçue par le composant le plus bas
 - Requête remontée jusqu'à ce qu'un composant la traite

Chaîne de responsabilité (2/4)



Chaîne de responsabilité (3/4)



Chaîne de responsabilité (4/4)

■ Intérêts

- ❑ Couplage réduit entre l'émetteur et les receveurs potentiels
 - Pas de connaissance explicite de tous les receveurs
 - Evite que l'émetteur maintienne une liste de candidats
- ❑ Modification dynamique de la chaîne des responsables
 - A tout moment, un objet peut être ajouté ou retiré
- ❑ Mais une requête peut ne pas être satisfaite

■ Relations avec d'autres patrons

- ❑ Composite
 - Avec une représentation sous forme arborescente
 - La chaîne de responsabilité peut être induite par la hiérarchie
 - Le parent agit alors comme le successeur dans la chaîne

■ Objectif

- ❑ Encapsuler une action dans un objet
- ❑ Permet l'abstraction de l'action
- ❑ Possibilité de file d'attente, annulation...

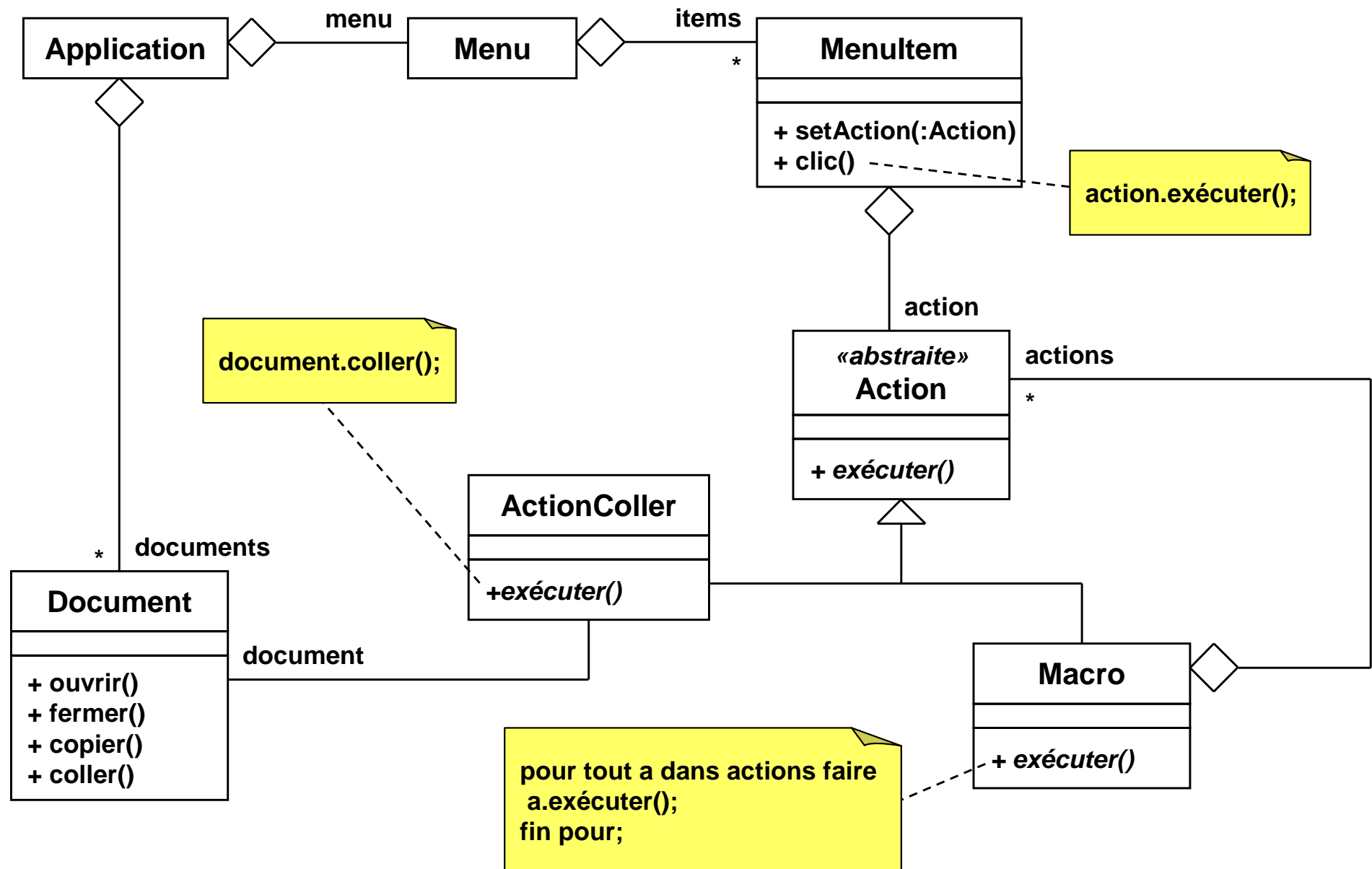
■ Principe

- ❑ Une interface modélise les actions
- ❑ Les objets déclencheurs agrègent une action
 - Déclencheur activé \Rightarrow exécution de l'action
- ❑ L'action connaît toutes les informations pour l'exécution
 - Procédure à exécuter
 - Quels sont les objets concernés

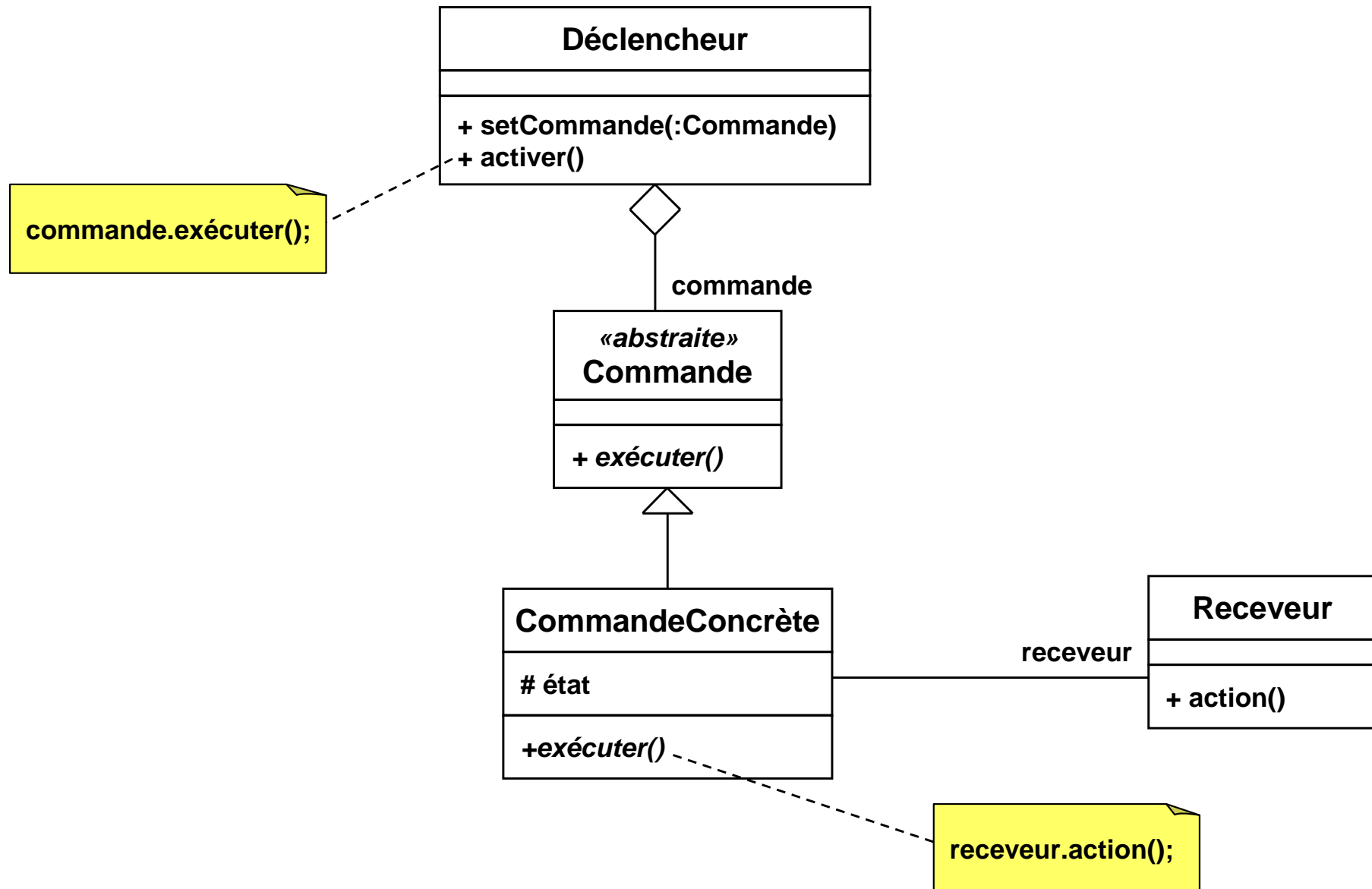
■ Motivation

- ❑ Associer des actions aux boutons d'une interface graphique
- ❑ Les boutons n'ont pas de lien direct avec le code métier

Commande / Command (2/4)



Commande / Command (3/4)



- Appelé aussi «action», «transaction»
- Intérêts
 - Découplage entre déclencheur et receveur
 - Ajout dynamique de nouvelles commandes
 - Possibilité de commandes agrégées (macrocommandes)
- Relations avec d'autres patrons
 - Composite
 - Utilisé pour concevoir une macrocommande
 - Memento
 - Mémorisation d'informations pour un processus d'annulation

- Objectif
 - ❑ Pour un langage simple donné
 - ❑ Définir une représentation pour sa grammaire
 - ❑ Ainsi qu'un interpréteur qui utilise la représentation pour interpréter des phrases du langage

- Motivation
 - ❑ Recherche de chaînes qui correspondent à un format

- Intérêts
 - ❑ Facile de changer et d'étendre une grammaire
 - ❑ Implémenter une grammaire est facile
 - ❑ Mais peu adapté à des grammaires difficiles

- Objectif
 - Fournir un accès séquentiel aux éléments d'un agrégat
 - Sans exposer sa représentation interne
- Principe
 - Itérateur = «pointeur» sur un élément d'un agrégat
 - L'agrégat fournit des itérateurs
 - Tous implémentent la même interface (quelque soit l'agrégat)
 - Le client ne manipule que l'itérateur
 - Il n'a pas forcément connaissance de l'agrégat
 - L'itérateur connaît la structure interne de l'agrégat
 - Il définit la manière de parcourir les éléments
- Motivation
 - Parcourir les éléments d'un conteneur

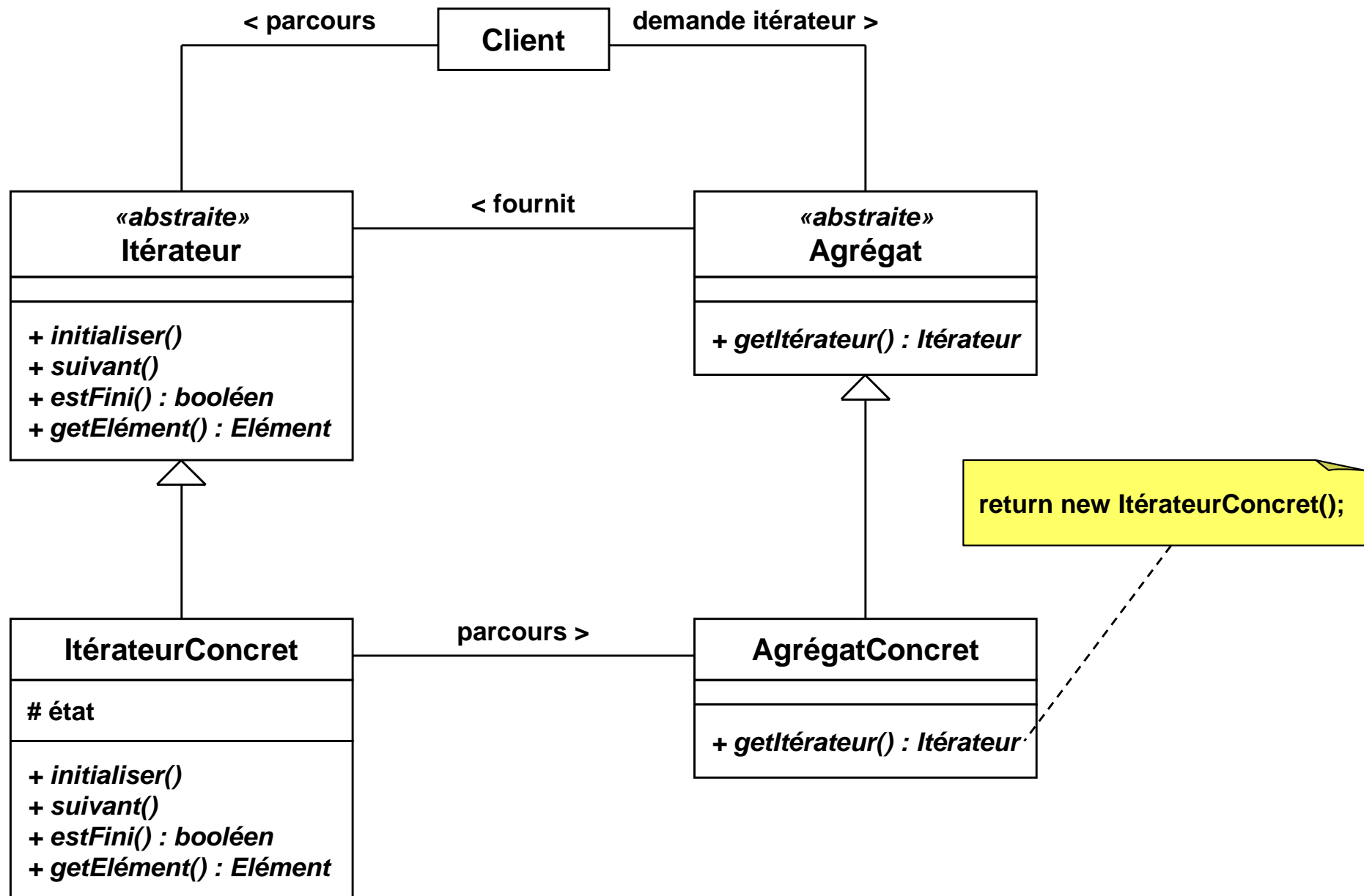
- L'itérateur fournit généralement 4 services
 - ❑ Positionnement sur le 1^{er} élément du parcours
 - ❑ Déplacement à l'élément suivant
 - ❑ Accès à l'élément courant
 - ❑ Test de fin de parcours

- L'accès à l'élément peut être contrôlé
 - ❑ Exemple: itérateur en lecture simple

- L'itérateur possède un état interne
 - ❑ Référence directe à l'élément
 - ❑ Référence à l'agrégat + position

- Besoin d'accéder à l'implémentation de l'agrégat
 - ❑ Classe embarquée
 - ❑ Classe amie

Itérateur / Iterator (3/4)

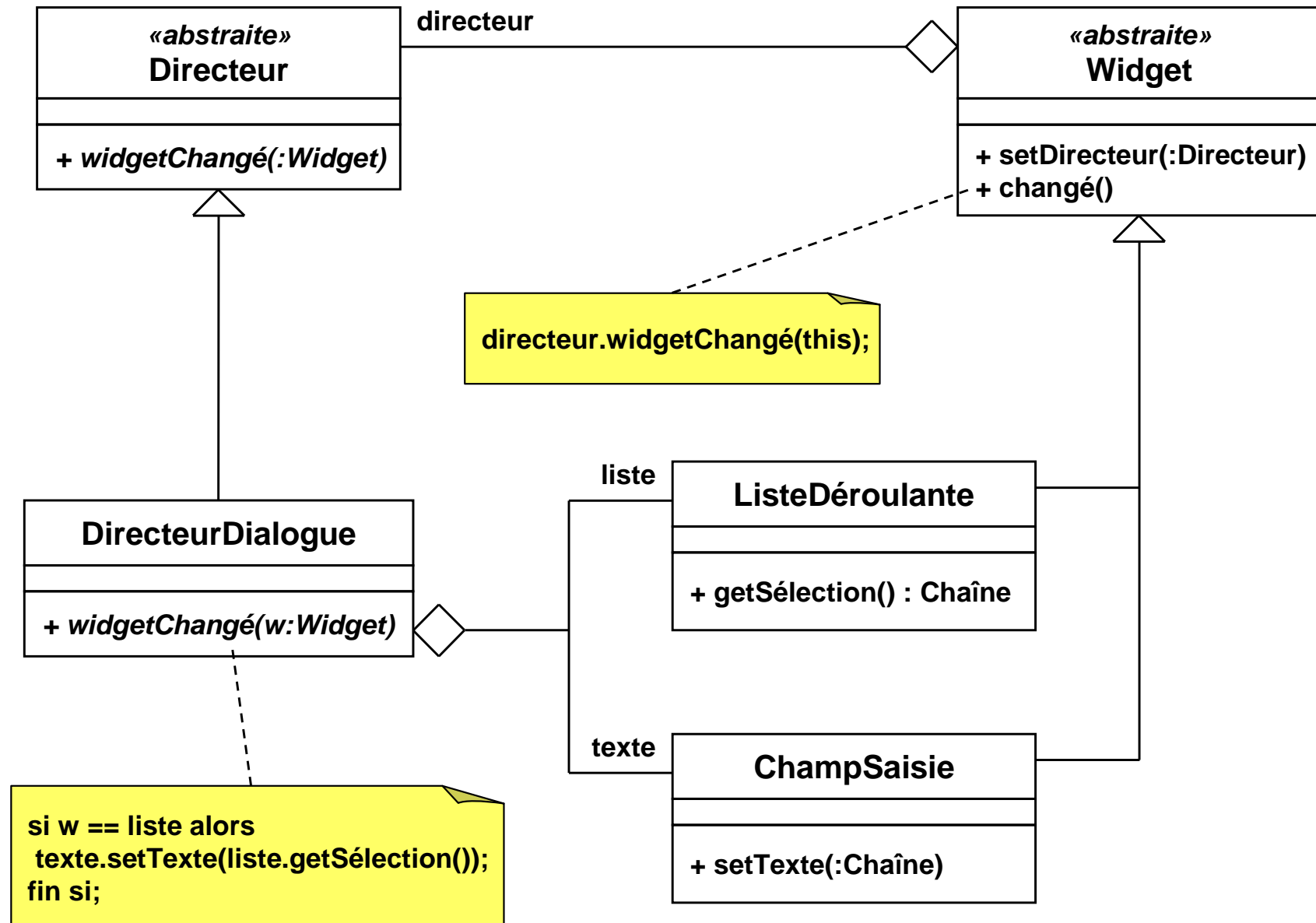


- Appelé aussi «curseur»
- Intérêts
 - Abstraction de la structure de l'agrégat
 - Seul l'itérateur est connu
 - Abstraction de la manière de parcourir
 - Un type d'itérateur par type de parcours
 - Evite de polluer l'interface de l'agrégat
 - Parcours simultanés possibles
- Relations avec d'autres patrons
 - Composite
 - Itérateur souvent utilisé pour parcourir l'arborescence

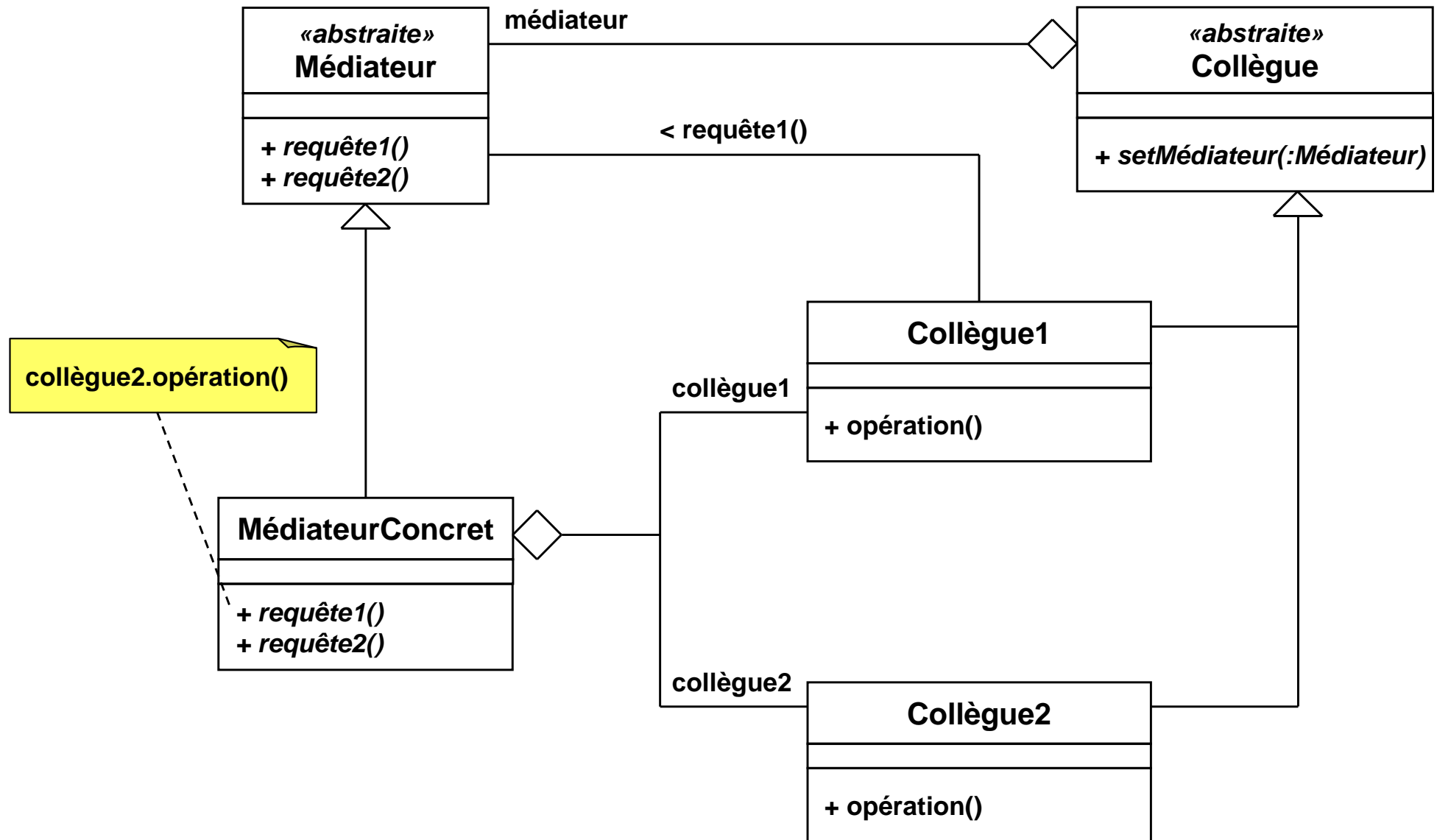
Médiateur / Mediator (1/4)

- Objectif
 - Encapsuler la manière d'interagir d'un groupe d'objets
 - Pas de référence explicite entre les objets
- Principe
 - Lorsqu'un objet a besoin d'un service
 - Il s'adresse à un objet central: le «médiateur»
 - Le médiateur comprend les requêtes
 - Il connaît les objets du système
 - Il sait à qui déléguer les requêtes
- Motivation
 - Communication d'objets dans une interface graphique
 - Beaucoup de messages liés aux événements
 - Il n'est pas nécessaire que chacun connaisse tous les autres

Médiateur / Mediator (2/4)



Médiateur / Mediator (3/4)



■ Intérêts

- ❑ Abstraction du mécanisme d'interaction
 - Découplage des objets
 - Possibilité de modifier le mécanisme indépendamment
- ❑ Contrôle centralisé
 - Evite la répartition sur plusieurs objets
 - Facilite l'extension: une classe à étendre

■ Relations avec d'autres patrons

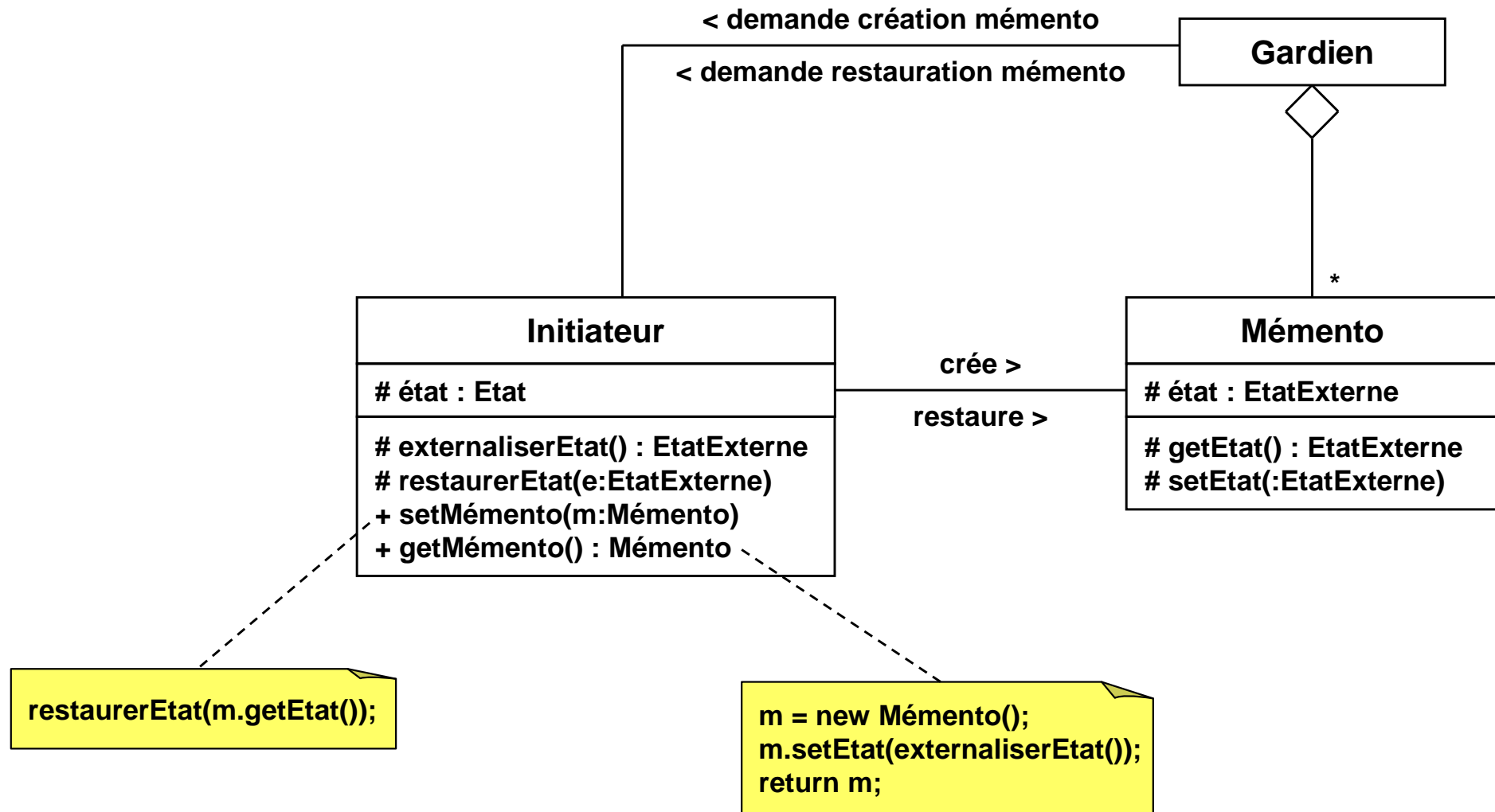
- ❑ Observateur
 - Moyen de communication avec le médiateur

- Objectif
 - Capturer et externaliser l'état d'un objet
 - Sans violer l'encapsulation
 - Permettre sa restauration ultérieure

- Principe
 - «Memento» = objet représentant l'état interne d'un autre objet
 - Un objet est le seul «initiateur» de ses mementos
 - Lui seul peut accéder aux données du memento par la suite

- Motivation
 - Mécanisme d'annulation («undo») parfois complexe
 - Nécessité de mémoriser l'état d'un objet pour le restituer plus tard

Memento / Memento (2/3)



- Appelé aussi «*token*»
- Intérêts
 - Préserve l'encapsulation de l'initiateur
 - Aucun objet extérieur n'accède à son état
 - Peu d'impact sur le code de l'initiateur
 - Seules les méthodes de création et restauration sont rajoutées
 - La gestion des mementos est faite à l'extérieur
 - Difficile de garantir l'accès exclusif de l'initiateur
 - Dépend du langage
 - C++: méthodes protégées et relation d'amitié
- Relations avec d'autres patrons
 - Commande
 - Utilisation conjointe pour le mécanisme d'annulation
 - Itérateur
 - Memento = état d'une itération

■ Objectif

- ❑ Synchroniser plusieurs objets sur l'état d'un autre objet
- ❑ Quand l'état de l'objet change
 - Les objets dépendants sont informés
 - Ils se mettent à jour

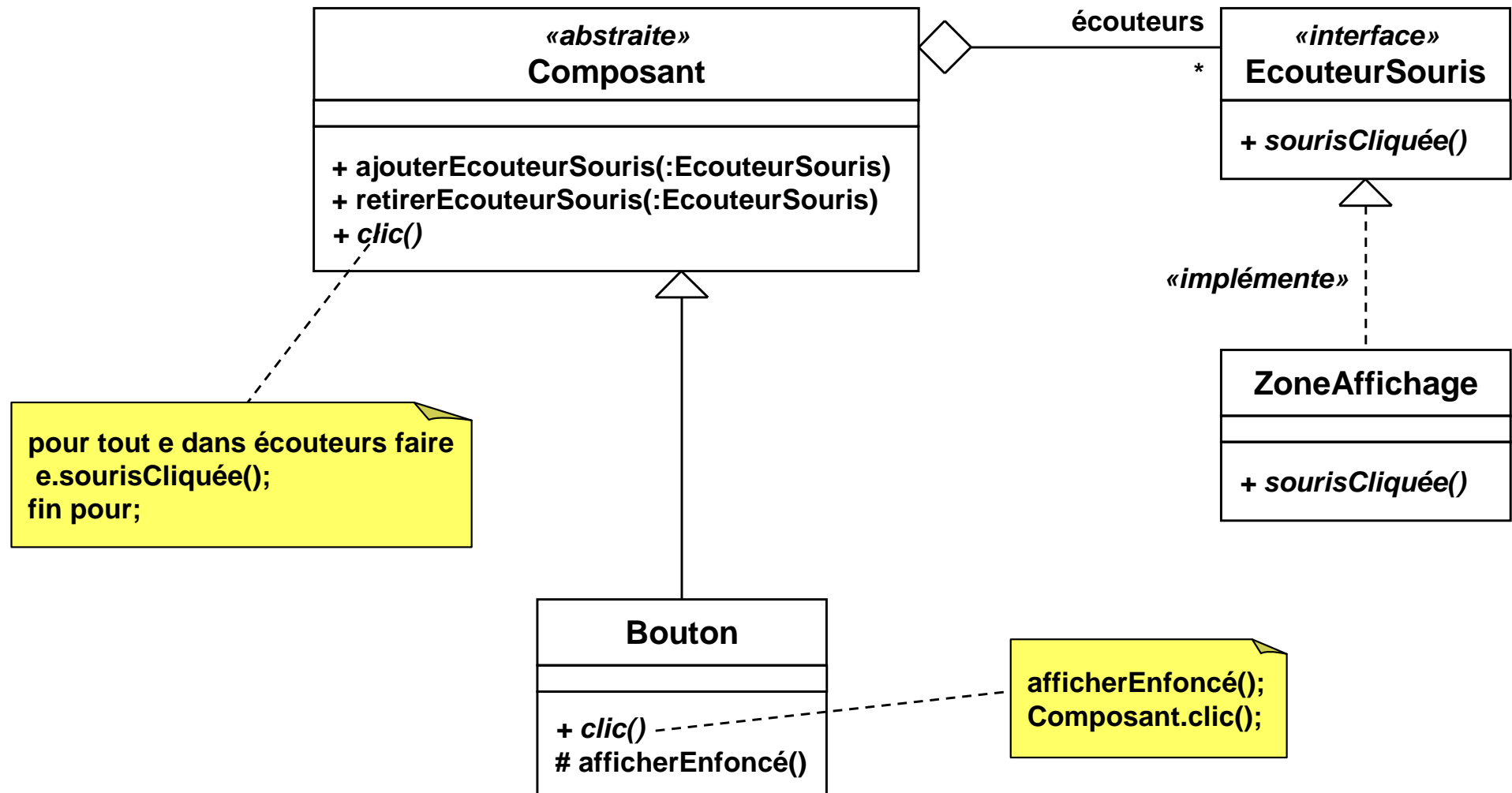
■ Principe

- ❑ Des objets «observateurs» s'enregistrent auprès d'un «sujet»
- ❑ Le sujet maintient donc une liste de ses observateurs
- ❑ Changement de l'état du sujet ⇒ Notification aux observateurs
 - Une méthode spécifique des observateurs est invoquée
 - Tous les observateurs doivent donc implémenter la même interface

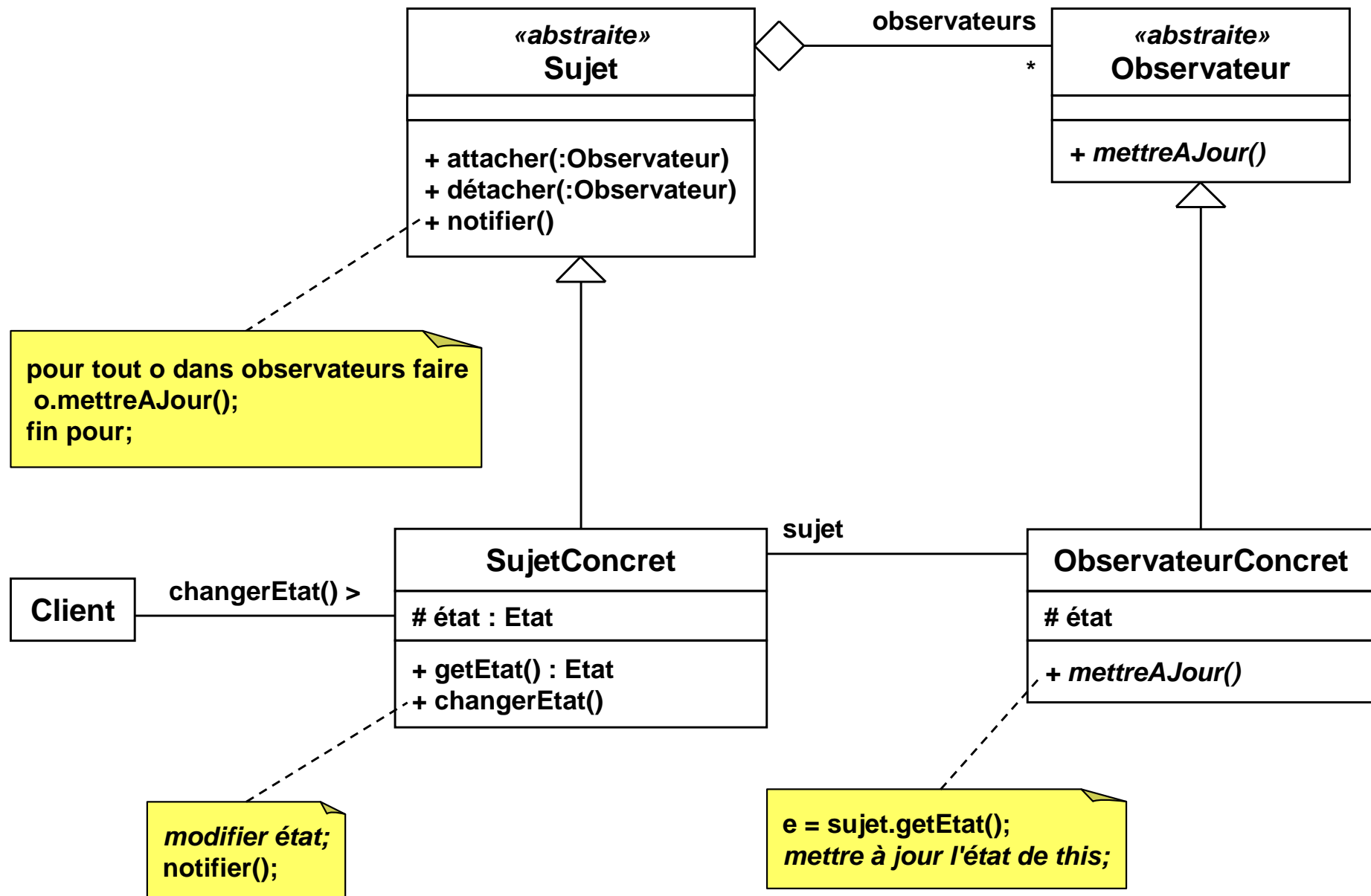
■ Motivation

- ❑ Capter des événements dans une interface utilisateur

Observateur / Observer (2/4)



Observateur / Observer (3/4)



Observateur / Observer (4/4)

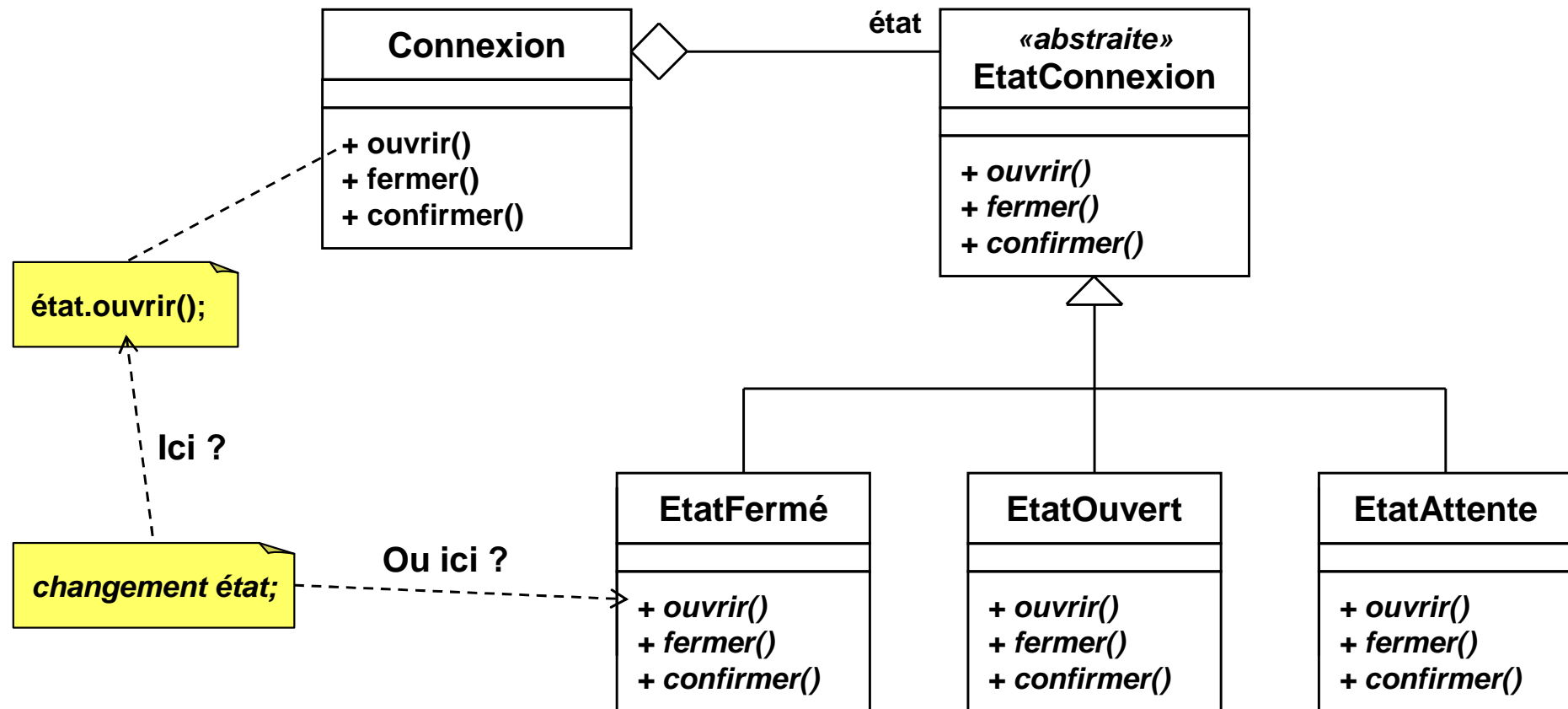
- Appelé aussi «*dependents*», «*publish-subscribe*» ou «*listener*»
- Intérêts
 - Evite un couplage fort entre le sujet et les observateurs
 - Le type concret des observateurs n'est pas connu du sujet
 - Les observateurs sont passifs
 - Pas besoin d'interroger le sujet en permanence
 - Informés quand le sujet change d'état
 - Mais attention au coût de modification de l'état du sujet
- Implémentation
 - Stockage de l'association observateur-sujet
 - Interne, chaque sujet maintient une liste d'observateurs
 - Externe, dans un conteneur associatif
 - Un observateur peut avoir plusieurs sujets
 - La méthode «**mettreAJour**» doit recevoir le sujet
- Relations avec d'autres patrons
 - Modèle-Vue-Contrôleur (MVC)
 - Modèle = sujet
 - Vue = observateur

- Objectif
 - ❑ Changer le comportement d'un objet en fonction de son état
 - ❑ Changement d'état équivalent à un changement de classe

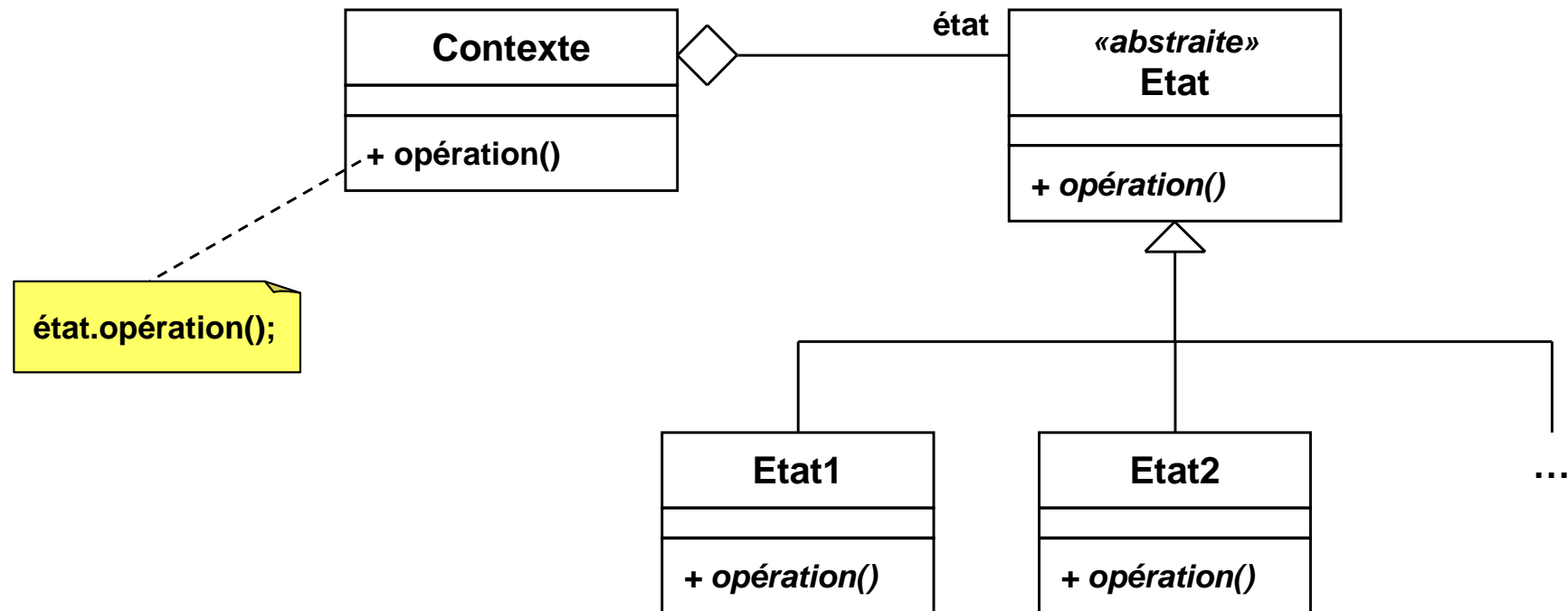
- Principe
 - ❑ Etats représentés sous forme d'objets
 - Ils possèdent la même interface
 - Chaque implémentation \Rightarrow un état différent
 - ❑ Un objet agrège un état
 - Changement de comportement \Rightarrow changement d'objet état

- Motivation
 - ❑ Réponses différentes suivant l'état d'une connexion réseau
 - ❑ Etats: établie, en attente, fermée...

Etat / State (2/4)



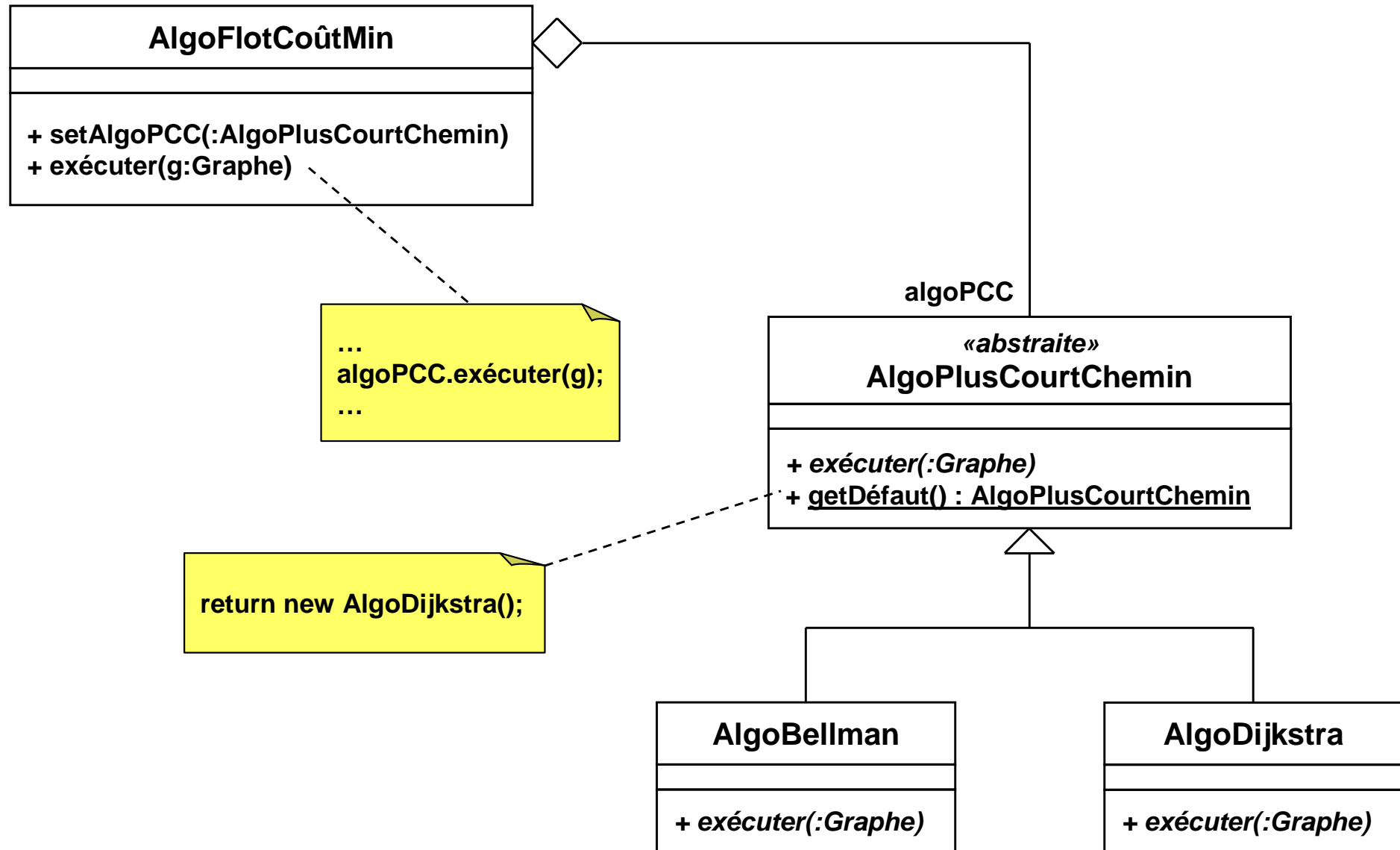
Etat / State (3/4)



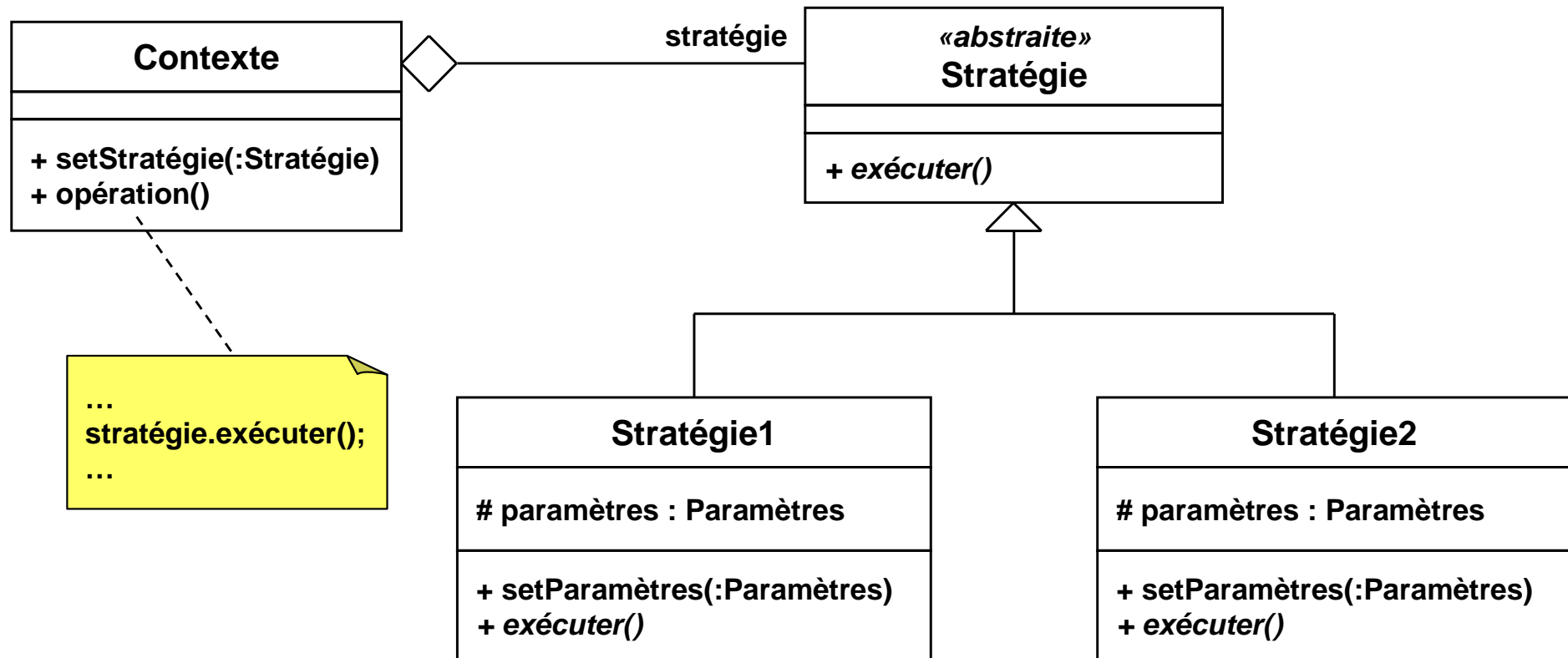
- Appelé aussi «*objects for states*»
- Intérêts
 - Evite des tests de comportement suivant l'état
 - Les comportements spécifiques sont localisés
 - Donc faciles à maintenir
 - Les transitions d'état sont explicites
 - Mais qui change l'état de l'objet ?
 - L'objet lui-même ou son objet état ?
- Relations avec d'autres patrons
 - Poids-mouche
 - Les objets états peuvent être partagés entre objets
 - Singleton
 - Les objets états peuvent être des singletons

- Objectif
 - Rendre les algorithmes d'une même famille interchangeables
- Principe
 - Les algorithmes («stratégies») sont modélisés par des classes
 - Une méthode représente le point d'entrée
 - Une classe abstraite définit une famille d'algorithmes
 - Nouvel algorithme = héritage et surcharge du point d'entrée
 - Un objet «contexte» agrège un algorithme
 - Sans connaître sa classe concrète
 - Le polymorphisme rend les algorithmes interchangeables
- Motivation
 - Proposer une variété d'algorithmes pour un même objectif
 - Possibilité de changer dynamiquement l'algorithme

Stratégie / Strategy (2/4)



Stratégie / Strategy (3/4)

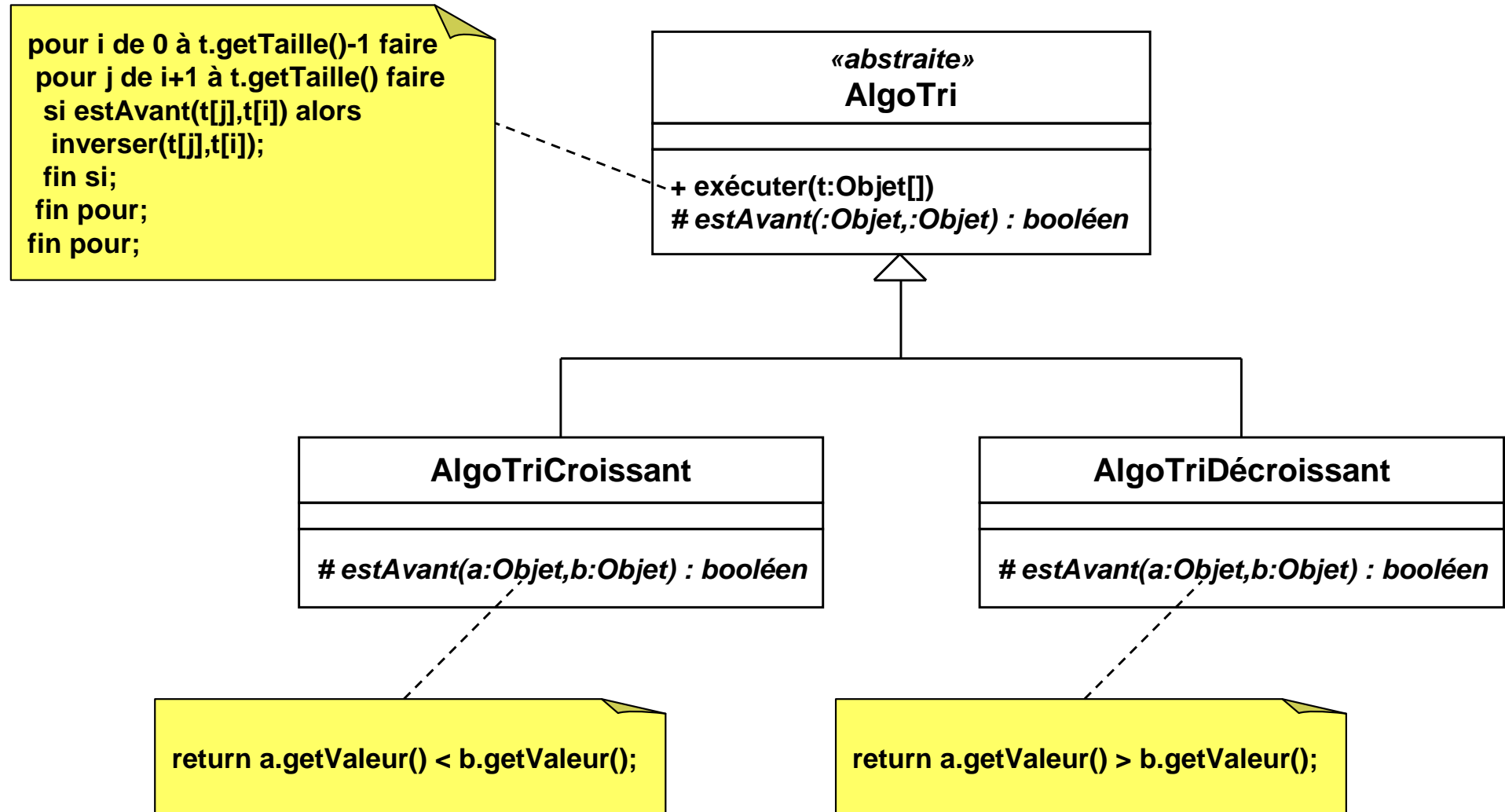


- Appelé aussi «*policy*»
- Intérêts
 - Abstraction de la stratégie
 - Interchangeable dynamiquement
 - Nouvelle stratégie \Rightarrow Aucun impact sur le contexte
 - La stratégie est dissociée du contexte
 - Evite des tests dans le contexte pour sélectionner la stratégie
- Contenu classique d'une classe stratégie/algorithme
 - Un point d'entrée
 - Méthode publique appelée pour exécuter l'algorithme
 - Des sous-algorithmes
 - Méthodes protégées utilisées par le point d'entrée
 - Des paramètres
 - Mémorisés dans des attributs
 - Constructeurs et accesseurs nécessaires pour l'initialisation
- Relations avec d'autres patrons
 - Singleton
 - Les stratégies peuvent être des objets uniques

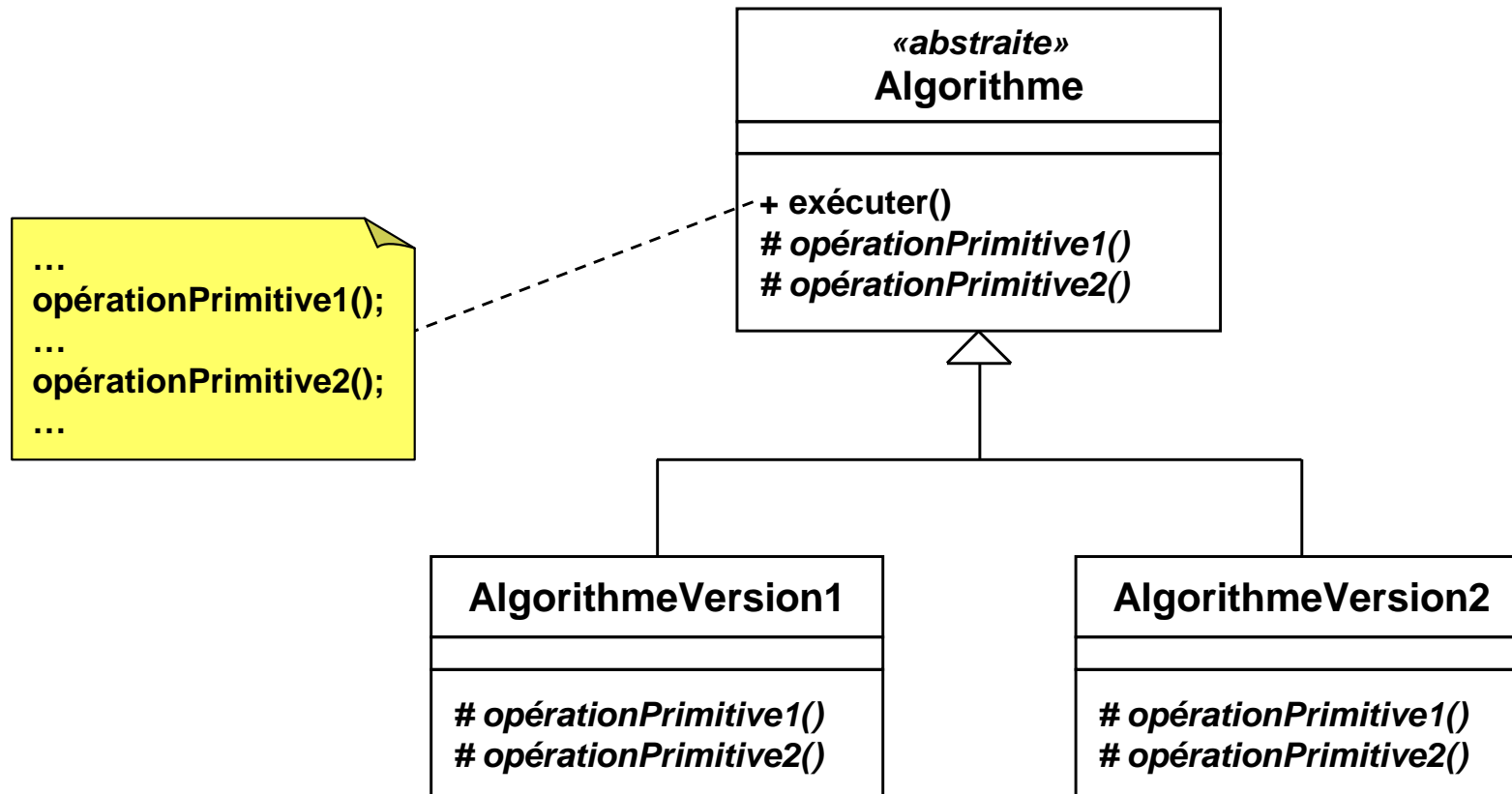
Méthode patron / *Template Method* (1/6)

- Très souvent appelé «patron de méthode»
 - Mauvaise traduction ?
- Objectif
 - Spécialiser un algorithme sans changer sa structure générale
- Principe
 - Définir le squelette d'un algorithme dans une classe
 - De la même manière que la stratégie
 - Délocaliser des parties dans des méthodes virtuelles
 - Par héritage, ces parties pourront être surchargées
- Motivation
 - Proposer plusieurs variantes d'un algorithme
 - Où la structure générale de l'algorithme est inchangée

Méthode patron / *Template Method* (2/6)



Méthode patron / *Template Method* (3/6)



Méthode patron / *Template Method* (4/6)

- Intérêts
 - Abstraction de parties d'un algorithme
 - Conserve la structure générale de l'algorithme
 - Rôle très important dans la réutilisabilité
 - Evite un détournement du rôle d'une classe
 - Guide / facilite la spécialisation de la classe
 - Mais éviter trop d'opérations primitives
 - Appelées trop souvent \Rightarrow Surcoût lié à la virtualité
 - Trop de méthodes \Rightarrow Surcharge fastidieuse pour l'utilisateur
- Utilisé pour la surcharge «par complément»
 - Objectif: surcharger pour compléter une méthode
 - Problème: il ne faut pas oublier d'appeler la version mère
 - Solution: utiliser une méthode patron

Méthode patron / *Template Method* (5/6)

- Surcharge par complément

- Approche classique

```
class Mere {  
    public: virtual void m(void) { /* Quelque chose */ }  
};  
  
class Fille : public Mere {  
    public: void m(void) { Mere::m(); /* Autre chose */ }  
};
```

- Approche avec méthode patron

```
class Mere {  
    protected: virtual void autreChose(void) = 0;  
    public: void m(void) { /* Quelque chose */ autreChose(); }  
};  
  
class Fille : public Mere {  
    protected: void autreChose(void) { /* Autre chose */ }  
};
```

Méthode patron / *Template Method* (6/6)

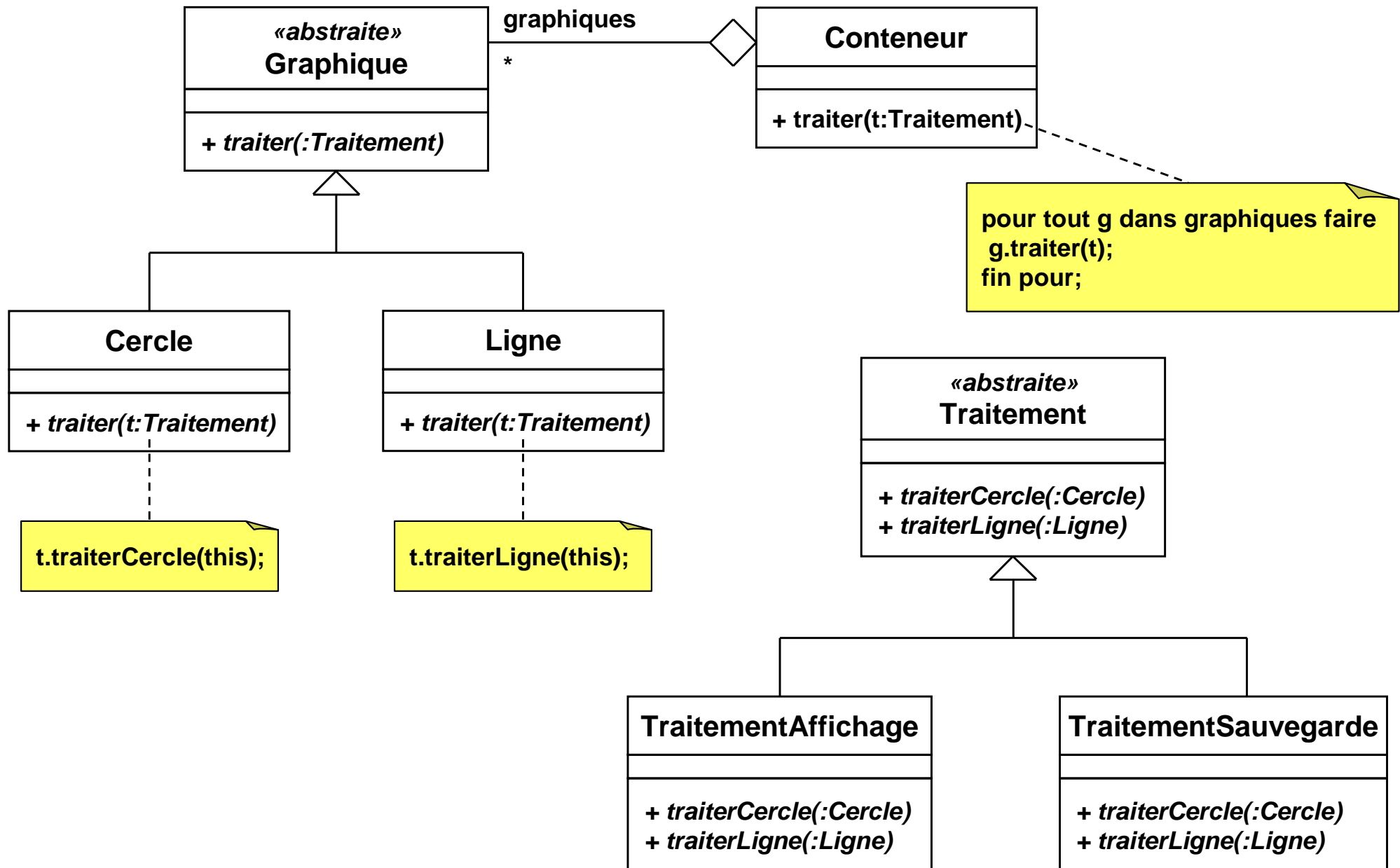
- Implémentation
 - Opérations primitives déclarées en protégé
 - Impossible de les appeler directement
 - Accessibles par les sous-classes, donc surchargeables
 - Pour imposer la surcharge, déclarer ces méthodes abstraites
- Relations avec d'autres patrons
 - Stratégie
 - Peuvent être conjoints
 - Stratégie: composition pour faire varier l'algorithme entier
 - Méthode patron: héritage pour faire varier des parties
 - (Méthode) Fabrique
 - Utilise une méthode patron pour la création de produits
 - Visiteur
 - Similaires, mais le visiteur utilise la composition
 - Les parties sont remplacées dynamiquement

- Objectif
 - Représenter une opération à appliquer sur un ensemble d'éléments
 - Définir une nouvelle opération sans modifier la classe des éléments

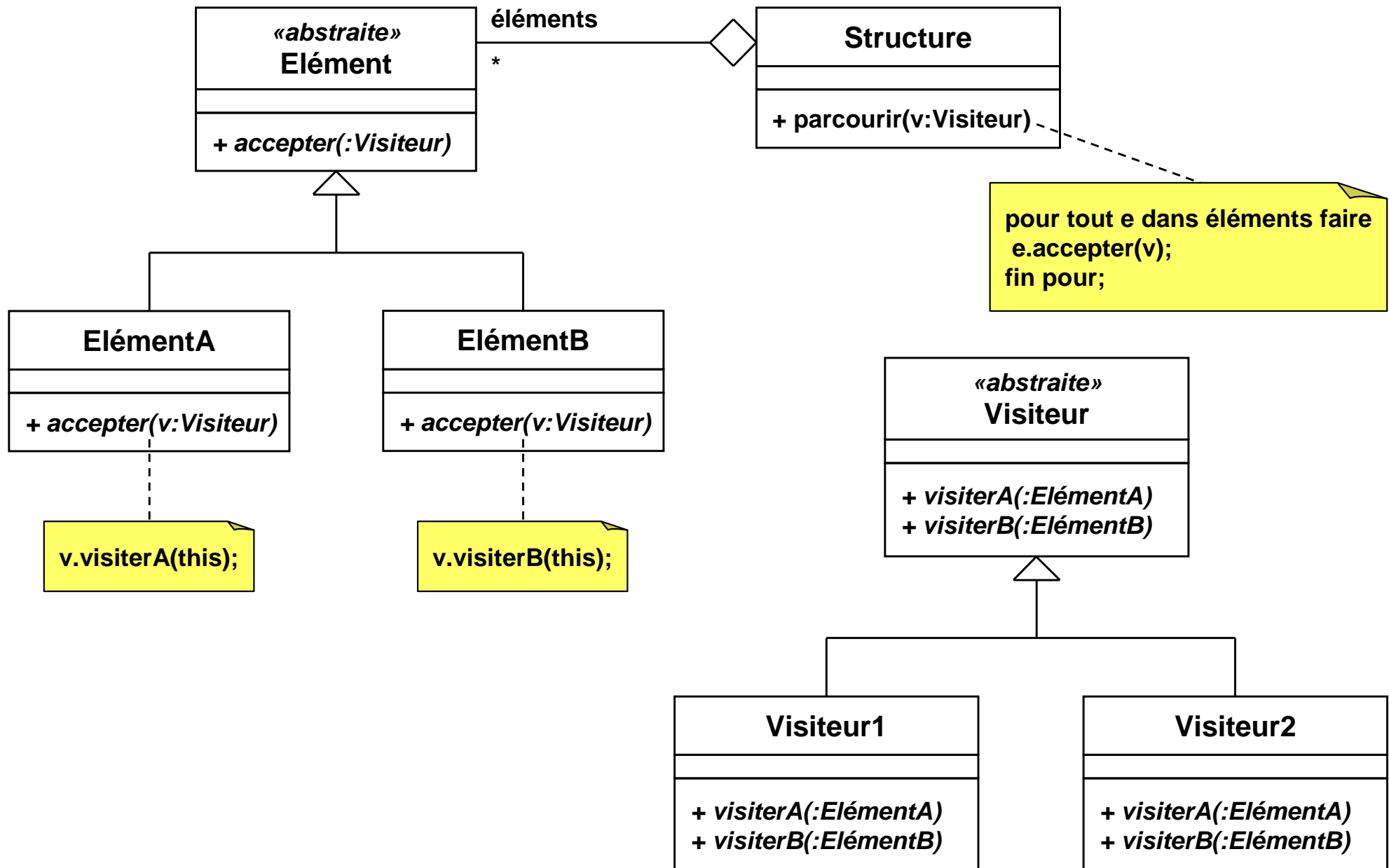
- Principe
 - L'opération est modélisée par un objet, le «visiteur»
 - Une classe, extensible, représente l'opération
 - Les éléments doivent «accepter» un visiteur
 - Une méthode doit recevoir le visiteur
 - Et appliquer l'opération associée sur l'élément
 - Une procédure de parcours applique l'opération aux éléments
 - Il reçoit le visiteur
 - Et le transmet à chacun des éléments

- Motivation
 - Appliquer des opérations différentes sur un ensemble d'objets
 - Mais le processus de parcours est toujours le même

Visiteur / Visitor (2/4)



Visiteur / Visitor (3/4)



■ Intérêts

- ❑ Propose plusieurs traitements sur les éléments
 - Sans alourdir l'interface de la structure
 - Sans alourdir l'interface des éléments
- ❑ Facilite l'ajout de nouveaux traitements
 - Il suffit de créer un nouveau visiteur
- ❑ Mais difficile d'ajouter un nouvel élément
 - Il faut ajouter une méthode dans chaque visiteur

■ Relations avec d'autres patrons

- ❑ Composite
 - Visiteur utilisé pour appliquer une opération sur l'arborescence
- ❑ Méthode patron
 - Visiteur peut être utilisé pour spécialiser des parties d'un algorithme
 - Similaires, mais la méthode patron utilise l'héritage

- Design Patterns
 - ❑ Interviennent au niveau conception
 - ❑ Fournissent des solutions générales et éprouvées
 - ❑ Favorisent la réutilisabilité

- Nous n'avons vu que les design patterns du GoF
 - ❑ Il en existe d'autres !
 - ❑ Exemples
 - MVC (Modèle-Vue-Contrôleur)
 - Patrons de concurrence
 - ...

- Anti-patterns
 - ❑ Erreurs de conception, réduisant la réutilisabilité
 - ❑ Pas vraiment de classification
 - ❑ Exemples
 - Objet divin: trop de fonctionnalités dans une classe
 - Patternite: utilisation abusive de design patterns
 - Lasagne: trop de niveaux d'héritage
 - ...