ISIMA

PARTIE IX

Patrons de conception

(Design Patterns)

Bruno Bachelet

Loïc Yon

Motivations *(1/2)*

- Concevoir un système objet est difficile
- Beaucoup d'aspects à considérer
  - Décomposition du système
  - Factorisation du code
  - Relations entre les composants
    - Héritage, association, agrégation / composition, délégation
- Prévoir et intégrer dès la conception
  - Réutilisation du code
  - Evolutions / extensions possibles

  $\Rightarrow$ introduire de la réutilisabilité

Motivations *(2/2)*

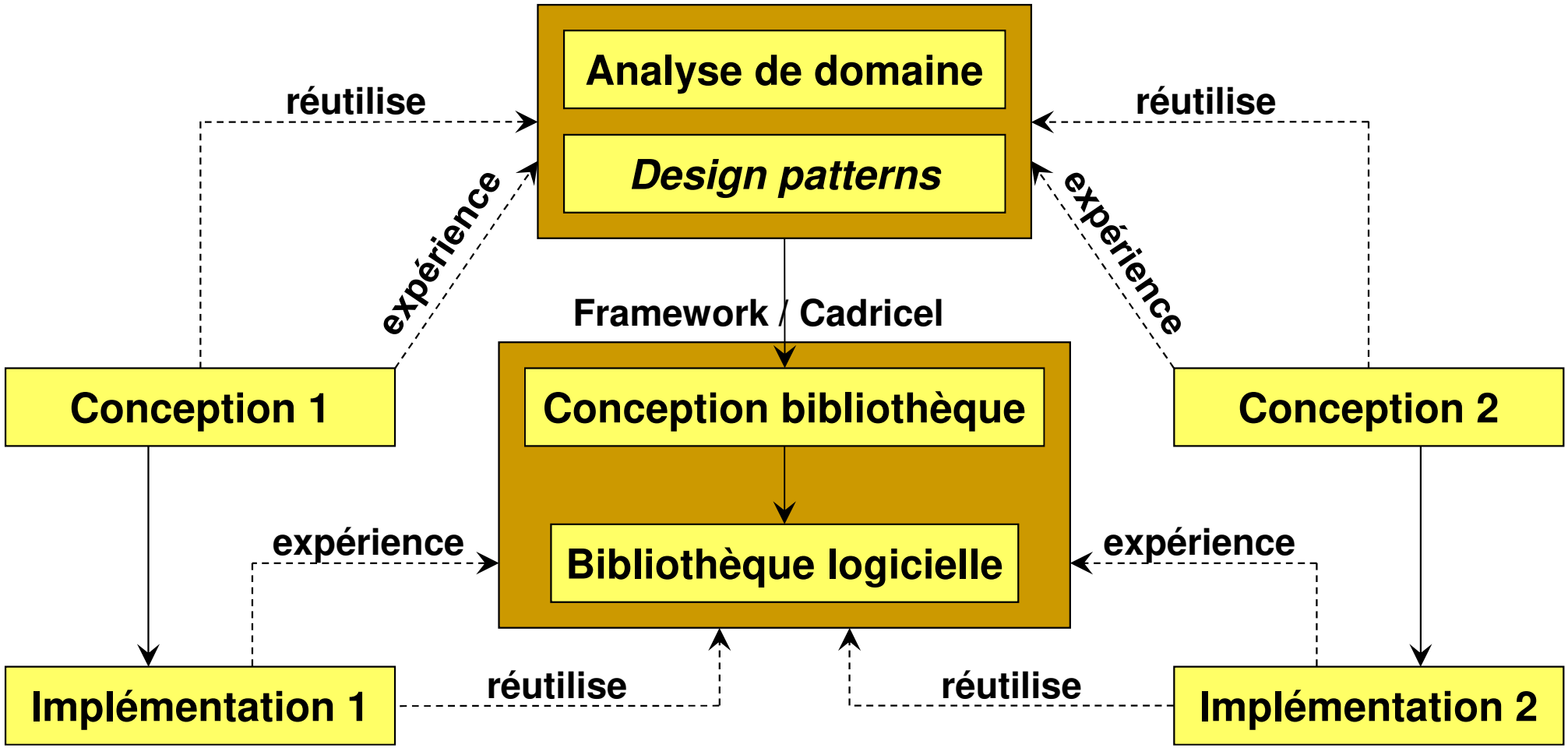
- Bénéficier des bonnes pratiques de l'industrie
  - Minimiser les risques dans la phase de développement
  - Se référer à l'existant
  - Reprendre des solutions éprouvées

- Permettre une réutilisation
  - Au niveau implémentation
    - Mêmes structures de données / algorithmes

    ⇒ bibliothèques logicielles
  - Au niveau conception
    - Mêmes organisations des composants

    ⇒ patrons de conception (ou *«design patterns»*)

Réutilisation à tous les niveaux



Patrons de conception (ou *design patterns*)

- Définition
  - Un *design pattern* traite un problème de conception récurrent
  - Il apporte une solution générale, indépendante du contexte
- En clair
  - Description de l'organisation de classes et d'instances en interaction pour résoudre un problème de conception
- Solution générique de conception
  - Doit être compréhensible et réutilisable
  - Doit être testée et validée dans l'industrie logicielle
  - Doit viser un gain en terme de génie logiciel
  - Doit être indépendante du contexte
- Volonté de référencement des solutions

Patrons de conception du *GoF (1/3)*

- Les patrons présentés ici sont issus du «*GoF*»
  - «*Gang of Four*»: Gamma, Helm, Johnson, Vlissides
  - Livre fondateur: 1ère proposition de référencement (1994)
  - *Design Patterns: Elements of Reusable Object-Oriented Software*
  - 23 patrons de conception

- Mais ce ne sont pas les seuls (cf. Wikipédia version anglaise)
  - Patron MVC (Modèle-Vue-Contrôleur)
  - Patrons GRASP
  - Patrons de concurrence
  - …

- Communauté active
  - De nouveaux *patterns* proposés régulièrement
  - Démocratique: adoptés si utilisés et généraux

Patrons de conception du *GoF (2/3)*

- Classification selon deux critères

- Cible: qui est concerné ?
  - Les classes
    - Relations d'héritage
    - Aspect statique
  - Les instances
    - Relations de composition
    - Aspect dynamique

- Objectif: que veut-on faire ?
  - Création de composants
  - Assemblage de composants
  - Comportement des composants

Patrons de conception du *GoF (3/3)*

Critères		**Objectif**		
		Création	Structure	Comportement
Cible	Classe	*Factory Method*	***Adapter***	*Interpreter* ***Template Method***
	Instance	***Abstract Factory*** *Builder* ***Prototype*** ***Singleton***	***Adapter*** *Bridge* ***Composite*** ***Decorator*** *Facade* *Flyweight* ***Proxy***	*Chain of Responsibility* ***Command*** *Iterator* *Mediator* *Memento* ***Observer*** *State* ***Strategy*** ***Visitor***

Principes de bonnes pratiques

- Favoriser une bonne conception
  - Facile à appréhender
  - Facile à faire évoluer
  - Résistante aux changements

- Quelques principes permettent de tendre à ces buts
  - Principes «SOLID»
    - Responsabilité unique (***S**ingle responsibility*)
    - **O**uvert/fermé
    - Substitution de **L**iskov
    - Ségrégation des **I**nterfaces
    - Inversion des **D**épendances
  - Connaissance minimale (Loi de Déméter)
  - Encapsuler ce qui varie
  - Programmer envers une interface
  - Préférer la composition à l'héritage
  - …

Responsabilité unique

- Chaque classe doit s'occuper d'une seule chose

  $\Rightarrow$ cohésion forte intra-module

  $\Rightarrow$ cohésion faible inter-module

- Une classe devrait avoir une seule raison de changer
  - Facilite la compréhension et la maintenabilité
  - Limite le risque d'introduction de bugs
    - Particulièrement lors d'évolutions
  - Facilite les tests

- Exemple: séparer le calcul de données de leur importation/exportation dans un fichier
  - Besoin de changer la manière de calculer
  - Ou besoin de changer le format d'import/export

  $\Rightarrow$ une seule classe doit être modifiée

Ouvert/fermé

- Un composant doit être ouvert aux extensions...
  - Permettre l'ajout de fonctionnalités
  - Permettre la modification du comportement
- ...mais fermé aux modifications
  - Le code d'un composant ne devrait pas être modifié si les besoins évoluent
- Les besoins changent régulièrement
  $\Rightarrow$ nécessité de pouvoir évoluer
- Tout en évitant de casser du code existant
- Exemple: les «politiques»
  - Prédicat d'un algorithme de copie (e.g. `std::copy_if`)

Encapsuler ce qui varie

- Séparer les aspects susceptibles de changer
  de ce qui ne changera pas
- Protège contre le changement
  - Stabilité du code face aux modifications
- Flexibilité pour les comportements sujets à variation
- Exemple
  - Isoler une partie d'un comportement dans une méthode
  - Encapsulation ⇒ modification sans impact sur le reste du code

Programmer envers une interface

- Programmer envers une interface
  et non envers une implémentation

- Favoriser un code sans dépendance
  avec les détails d'implémentation
  - Modification de l'implémentation sans impact sur le code
  - Changement d'implémentation facilitée

- Exemple
  - En C++: manipuler un vecteur avec des itérateurs plutôt qu'avec des indices
  - En Java: faire référence à un conteneur via une classe abstraite
    - `Collection c = new ArrayList();`
  - Dans les 2 cas, changement de conteneur $\Rightarrow$ aucun changement sur le code

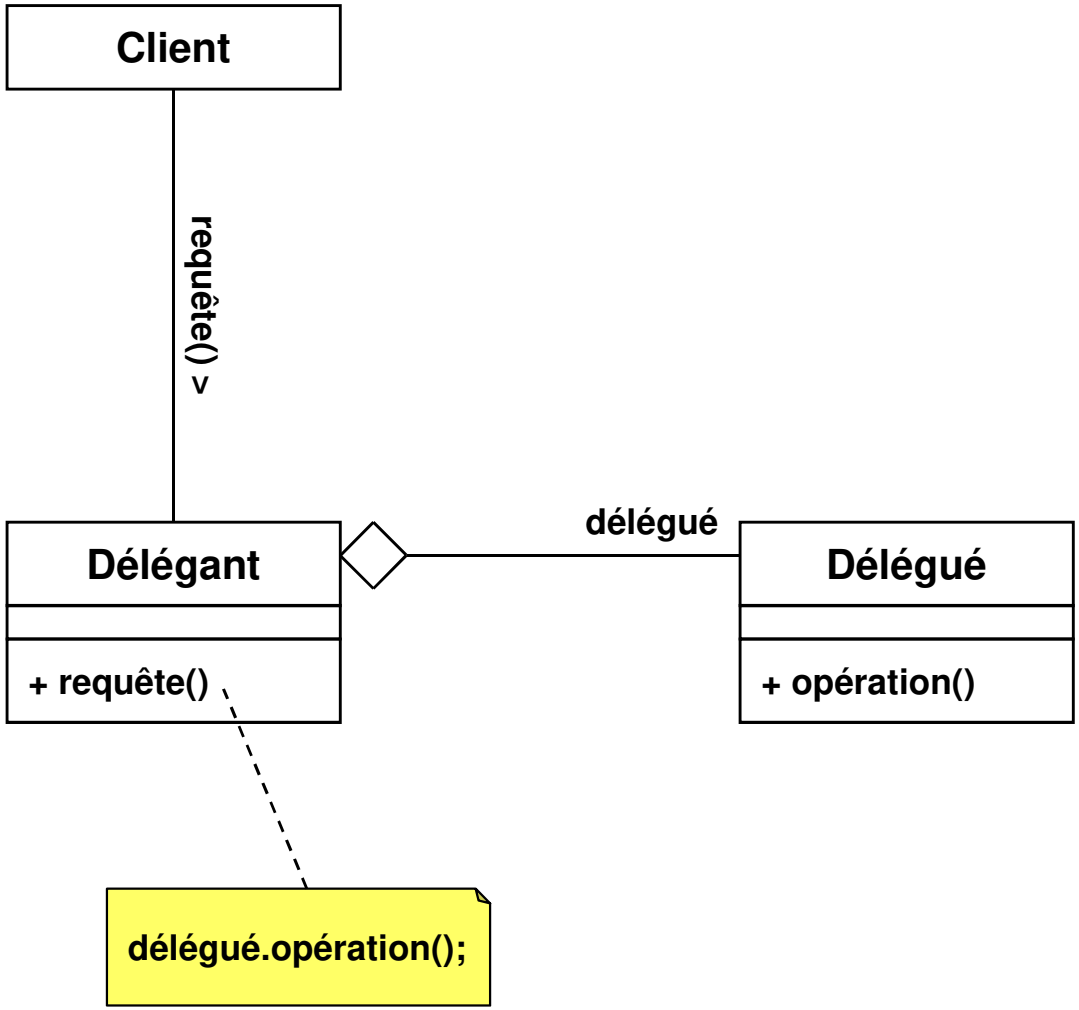
Préférer la composition à l'héritage

- Héritage → statique
- Composition → dynamique
- Utilisation de la délégation
  - Couplage plus faible
  - Changement de l'objet fournissant les services à l'exécution
- Permet d'éviter un héritage conceptuellement bancal
  - Une erreur classique (cf. principe de substitution de Liskov)
- Evite un accès aux données membres
  - Respecte mieux la notion d'interface
  - Respect de l'encapsulation

Mécanisme de délégation *(1/2)*

- Principe
  - Rediriger un message vers un autre objet
  - Utilise la composition: délégant vers délégué
- Intervient dans de nombreux patrons du *GoF*
  - Peut être une alternative à l'héritage
- Plusieurs manières de rediriger
  - Contrôle du message (e.g. *Proxy*)
  - Changement de message (e.g. *Adapter*)
  - Changement de délégué (e.g. *Chain of Responsibility*)
  - Redéfinition du message (e.g. *Decorator*)

Mécanisme de délégation *(2/2)*



Patrons de création

- Abstraction du processus de création
  - Indépendance du type réel
  - Indépendance de l'initialisation
  - Indépendance de la composition
- Niveau classe
  - Utilisation de l'héritage
- Niveau objet
  - Délégation de l'instanciation
- Utiles pour la création d'objets par composition
  - Introduit de la flexibilité dans l'assemblage

Singleton / *Singleton (1/2)*

- Objectif
  - Garantir une seule instance pour une classe
  - Fournir un point d'accès global à cette instance
- Principe
  - Empêche toute construction ou copie de cette classe
    - Impossibilité de créer un objet en dehors de la classe elle-même
  - Fournir une méthode de classe qui retourne l'objet unique
- Motivation
  - Représentation de ressources physiques uniques
  - Exemple: flux d'entrée et sortie standards

Singleton / *Singleton (2/2)*

- Exemple C++

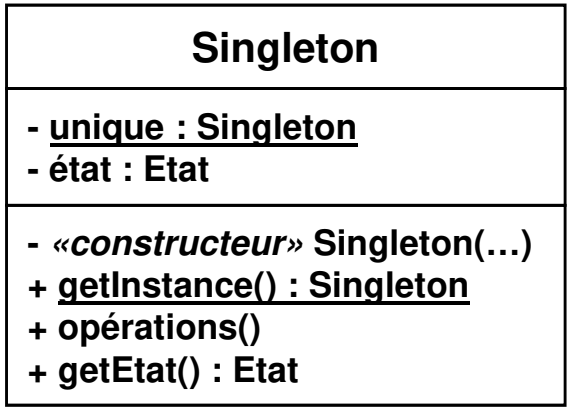
```
class Singleton {
 private:
  static Singleton unique_;
  // Attributs du singleton

 private:
  Singleton(…) {…}

  Singleton(const Singleton &) = delete;
  Singleton & operator=(const Singleton &) = delete;

 public:
  static Singleton & getInstance() { return unique_; }
  // Méthodes du singleton
};

Singleton Singleton::unique_(…);
```



- Constructions et copies d'un objet interdites ⇒ opérateurs privés ou supprimés

- Seule possibilité: utiliser l'instance unique via «`getInstance`»

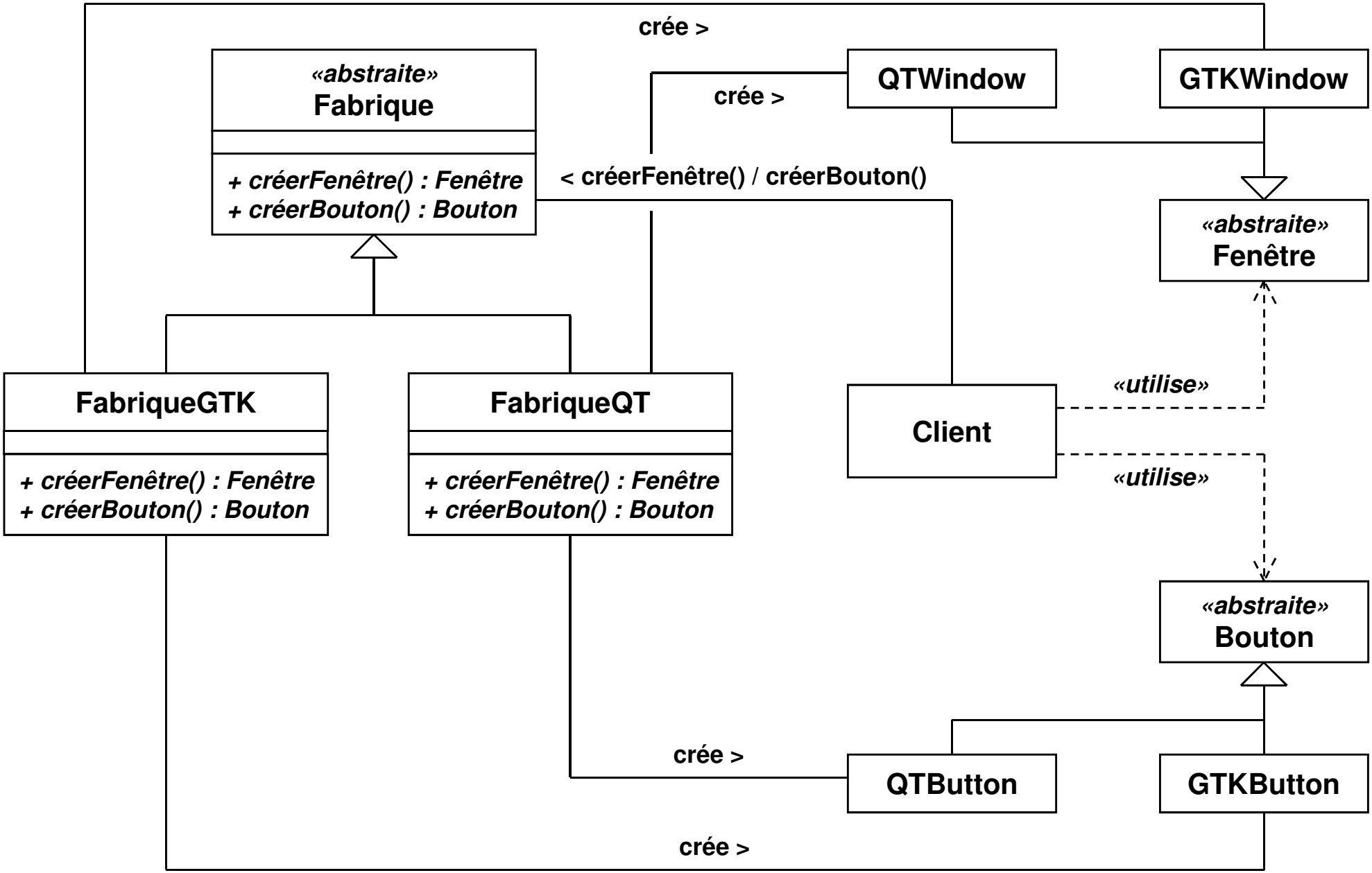
Fabrique abstraite / *Abstract Factory (1/3)*

- Objectif
  - Créer une famille d'objets cohérents
  - Des objets de classes différentes sont à créer
  - Mais les classes doivent être cohérentes entre elles

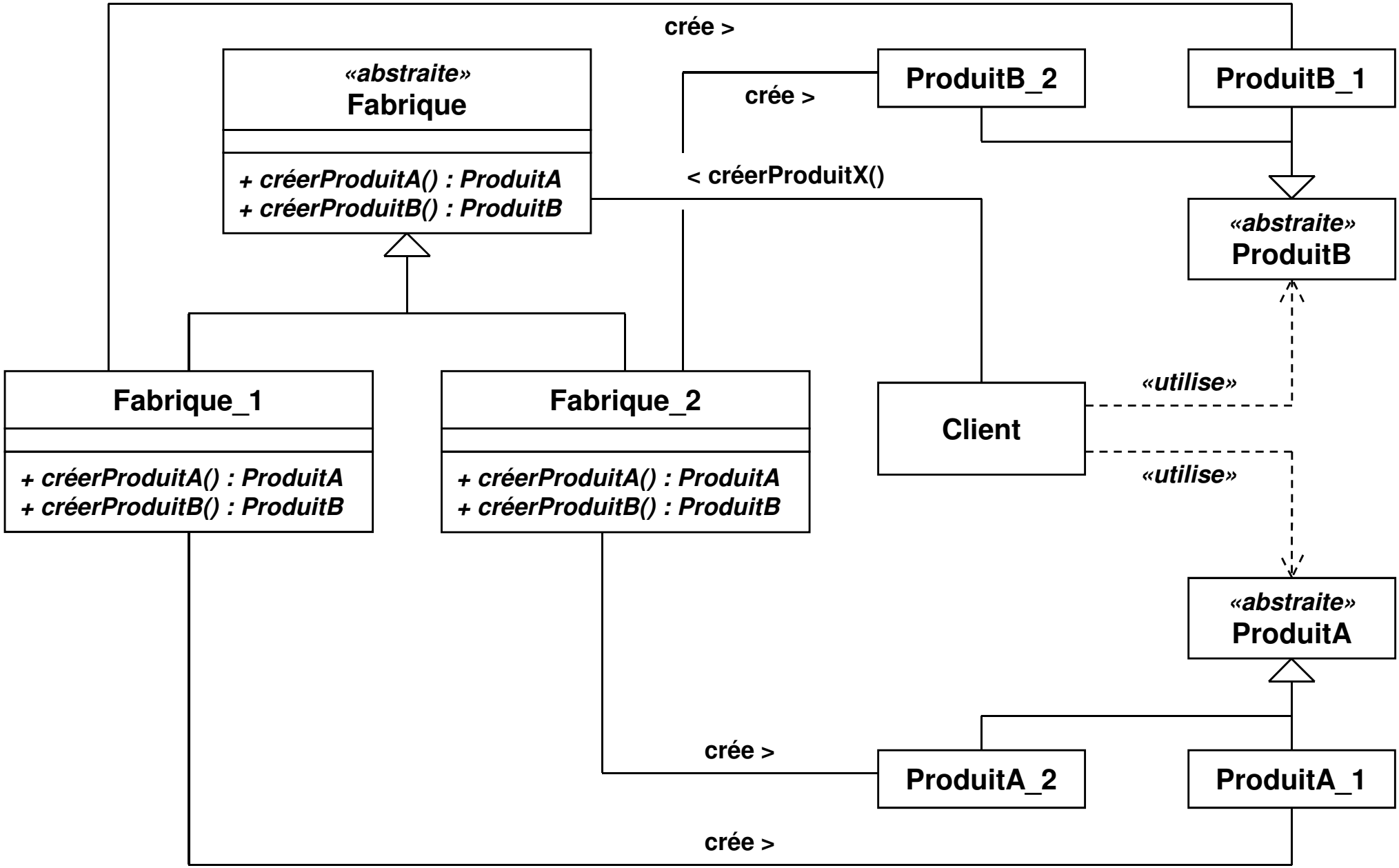
- Principe
  - Fournir une interface pour créer une famille d'objets ⇒ la «fabrique»
  - Le client demande à la fabrique de lui fournir des instances
    - Le client ne connaît que les interfaces des objets
    - Seule la fabrique connaît les classes réelles des objets

- Motivation
  - Système indépendant de l'interface graphique
  - Créer des composants graphiques cohérents selon la plateforme

Fabrique abstraite / *Abstract Factory (2/3)*



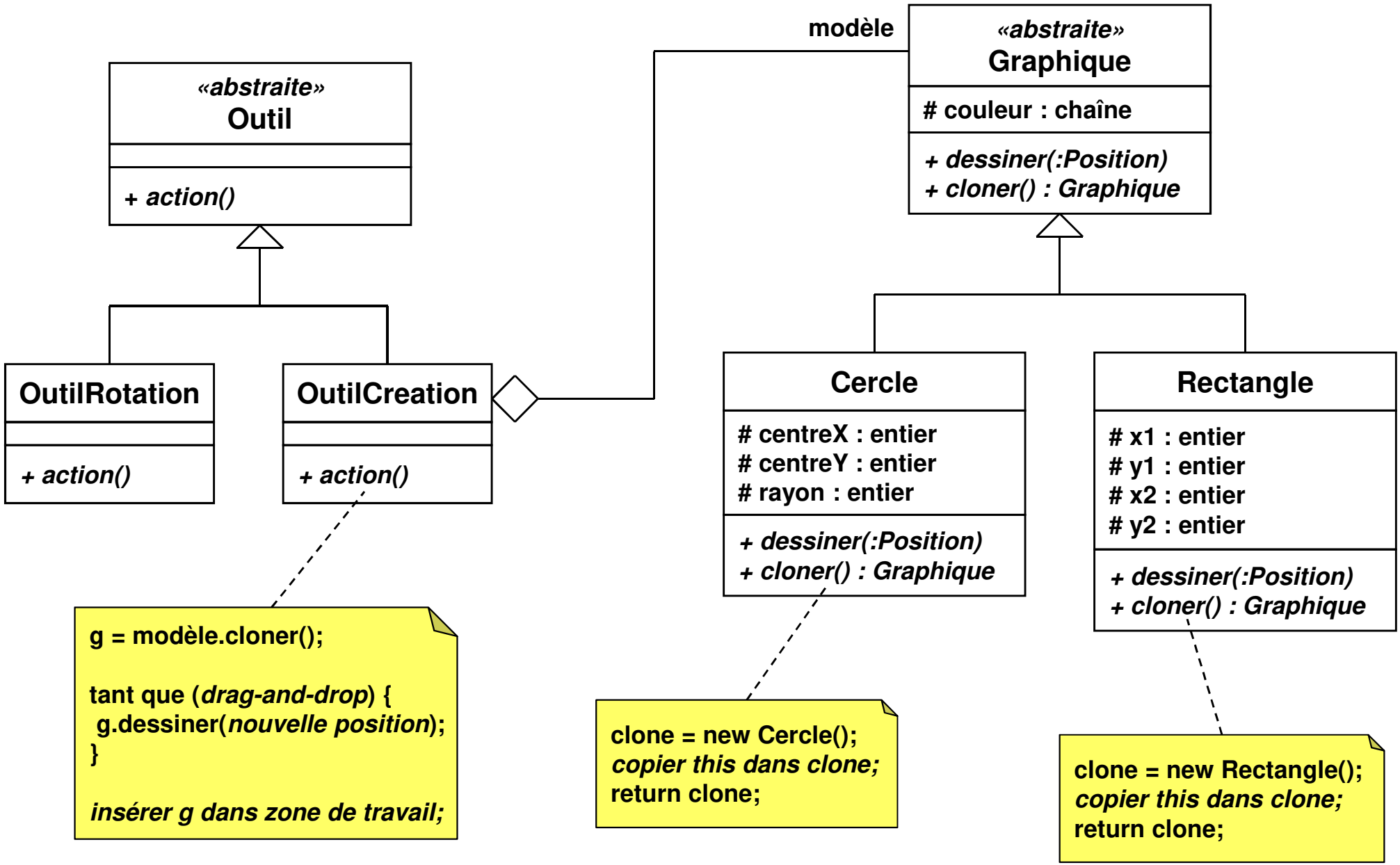
Fabrique abstraite / *Abstract Factory (3/3)*



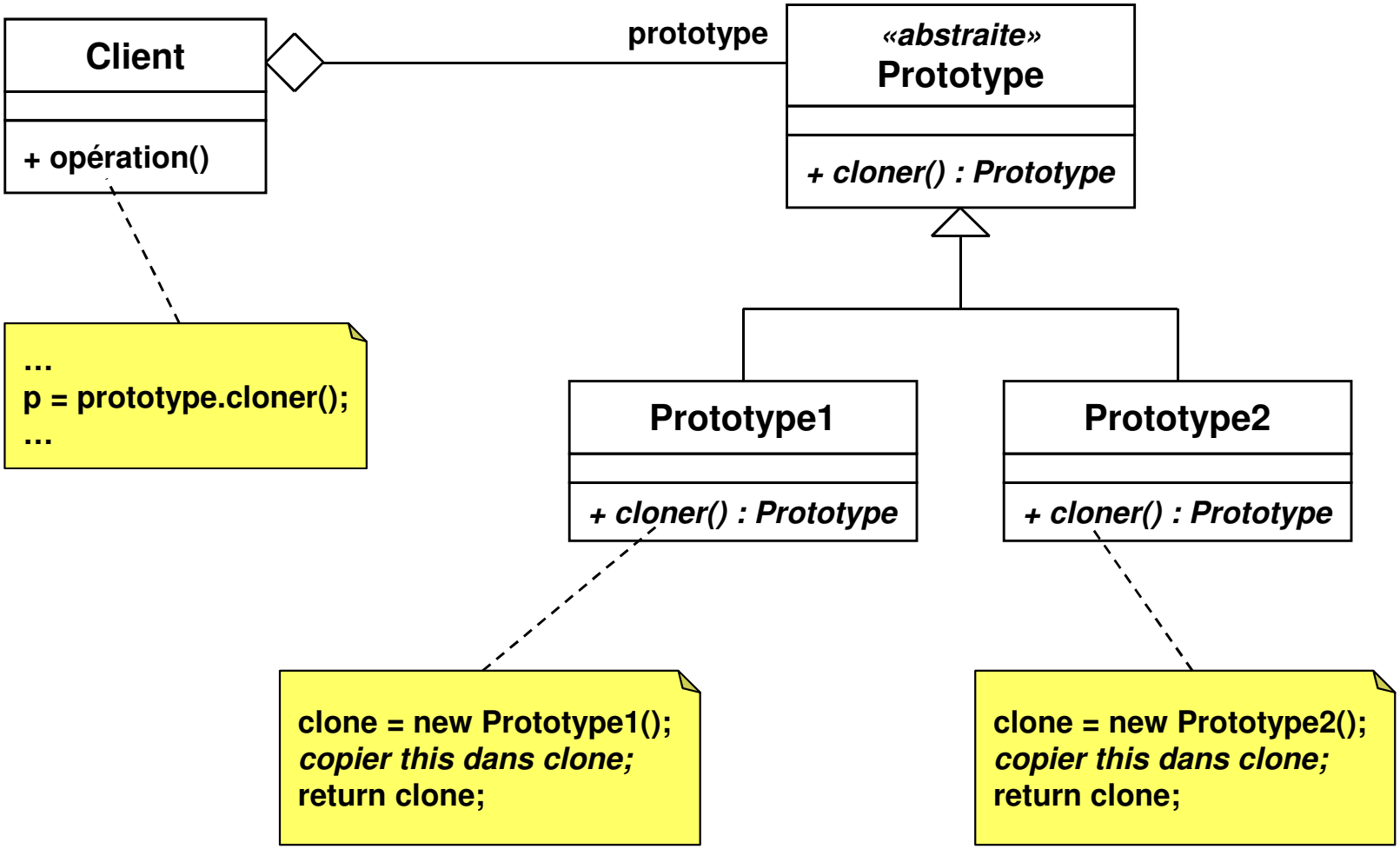
Prototype / *Prototype (1/3)*

- Objectif
  - Créer un objet par clonage d'une instance modèle
  - Le type de l'objet est déterminé par celui de l'instance modèle
- Principe
  - Un objet «prototype» est fourni
  - Il possède une méthode de clonage
  - Le client utilise cette méthode pour obtenir une copie de l'objet
- Motivation
  - Boîte à outils: déposer des objets par *drag-and-drop*
  - Une copie du modèle est déposée sur la zone de travail

Prototype / *Prototype (2/3)*



Prototype / *Prototype (3/3)*



Patrons de structure

- Concevoir de nouveaux composants par assemblage
  - Pour former des structures plus vastes
  - Avec un comportement plus complexe
- Objectif: exploiter les capacités d'un composant et les adapter à de nouveaux besoins
- Niveau classe
  - Utilisation de l'héritage

  $\Rightarrow$ composition d'interfaces ou d'implémentations
- Niveau objet
  - Utilisation de la composition

Adaptateur / *Adapter (1/3)*

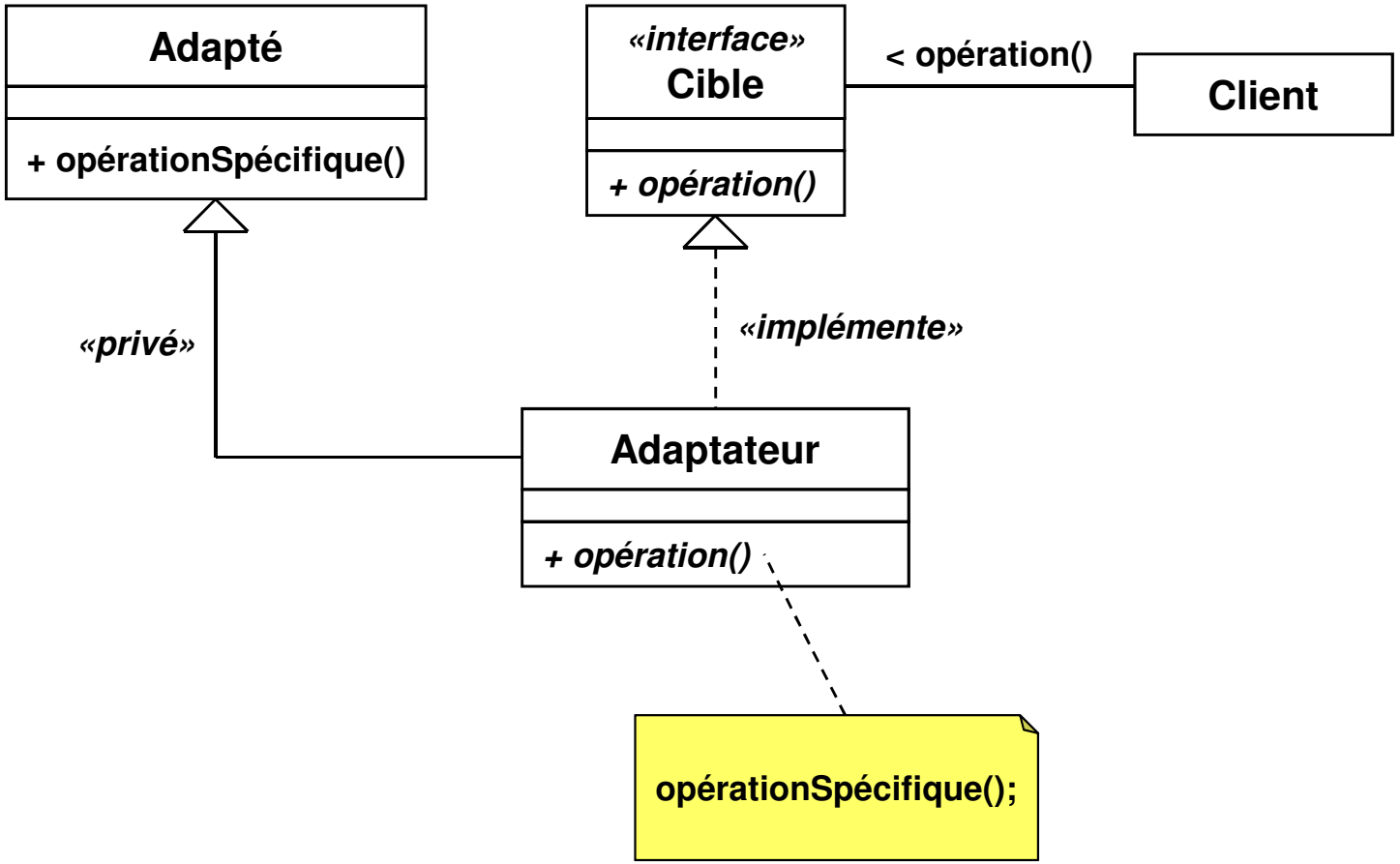
- Objectif
  - Adapter l'interface d'une classe à ses besoins
  - Permettre le dialogue entre classes incompatibles

- Principe
  - Deux approches
  - Classe «adaptateur» $\Rightarrow$ héritage «multiple»
    - Héritage de la nouvelle interface
    - Héritage de l'implémentation de l'ancienne interface
  - Objet «adaptateur» $\Rightarrow$ délégation
    - Héritage de la nouvelle interface
    - Agrégation d'un objet de l'ancienne interface

- Motivation
  - Utiliser une fonctionnalité d'une bibliothèque tierce
  - Mais l'interface n'est pas adaptée

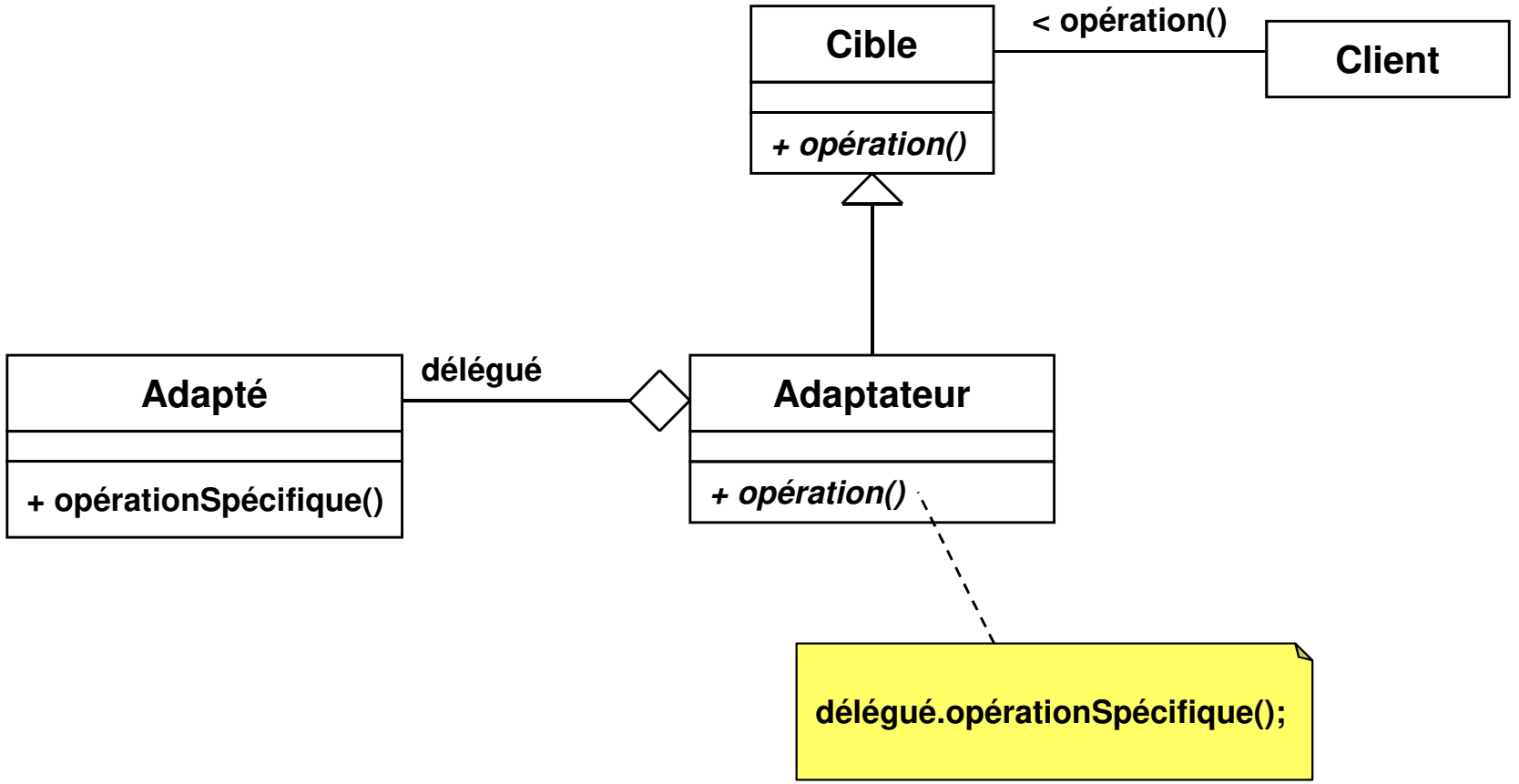
Adaptateur / *Adapter (2/3)*

- Classe adaptateur
  - Héritage de la nouvelle interface
  - Héritage de l'implémentation de l'ancienne interface
  - Un seul objet (adapté + adaptateur) est créé



Adaptateur / *Adapter (3/3)*

- Objet adaptateur
  - Héritage de la nouvelle interface
  - Agrégation d'un objet de l'ancienne interface, et délégation
  - Permet l'adaptation d'une classe et de ses sous-classes

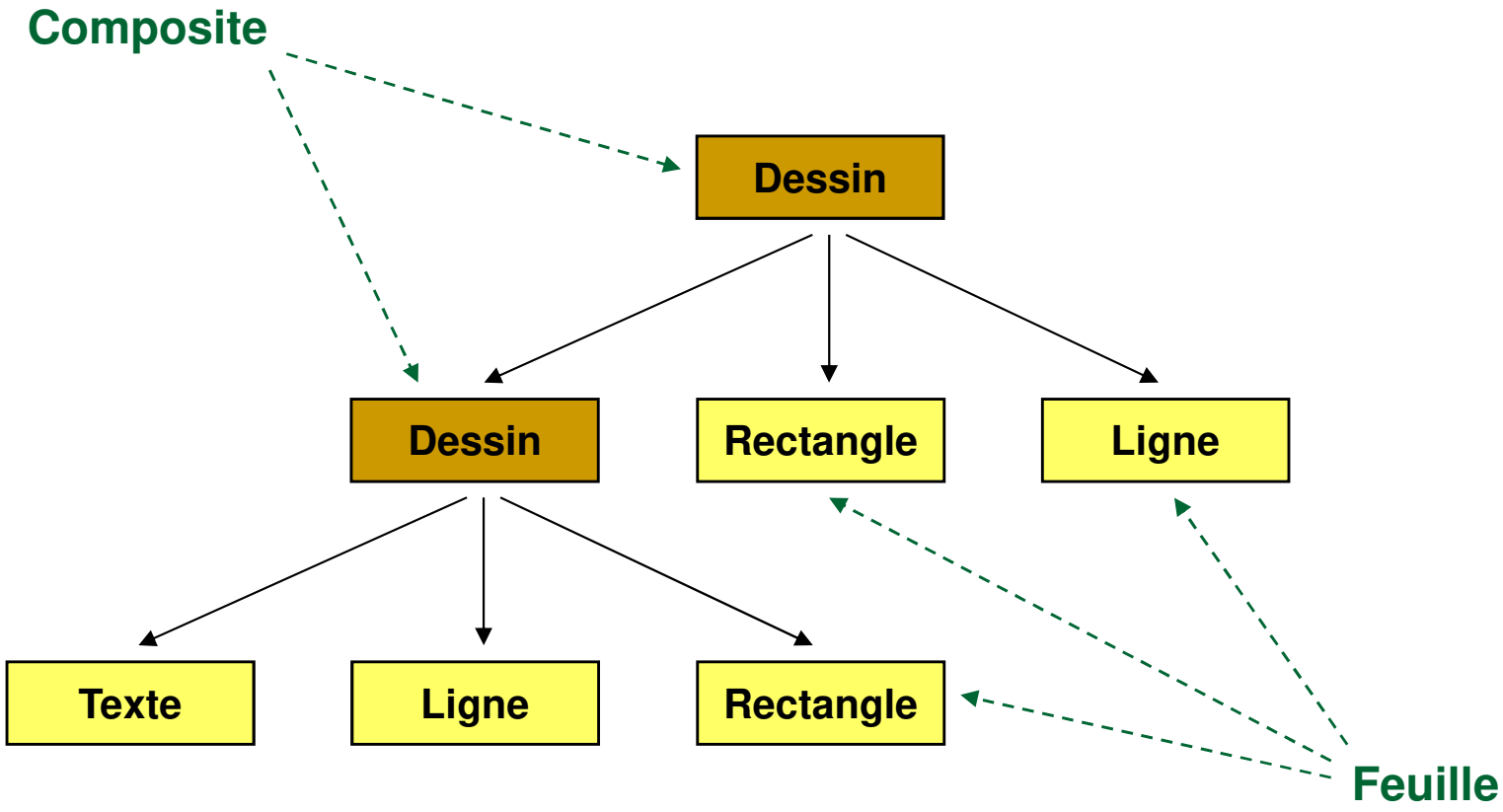


Composite / *Composite (1/5)*

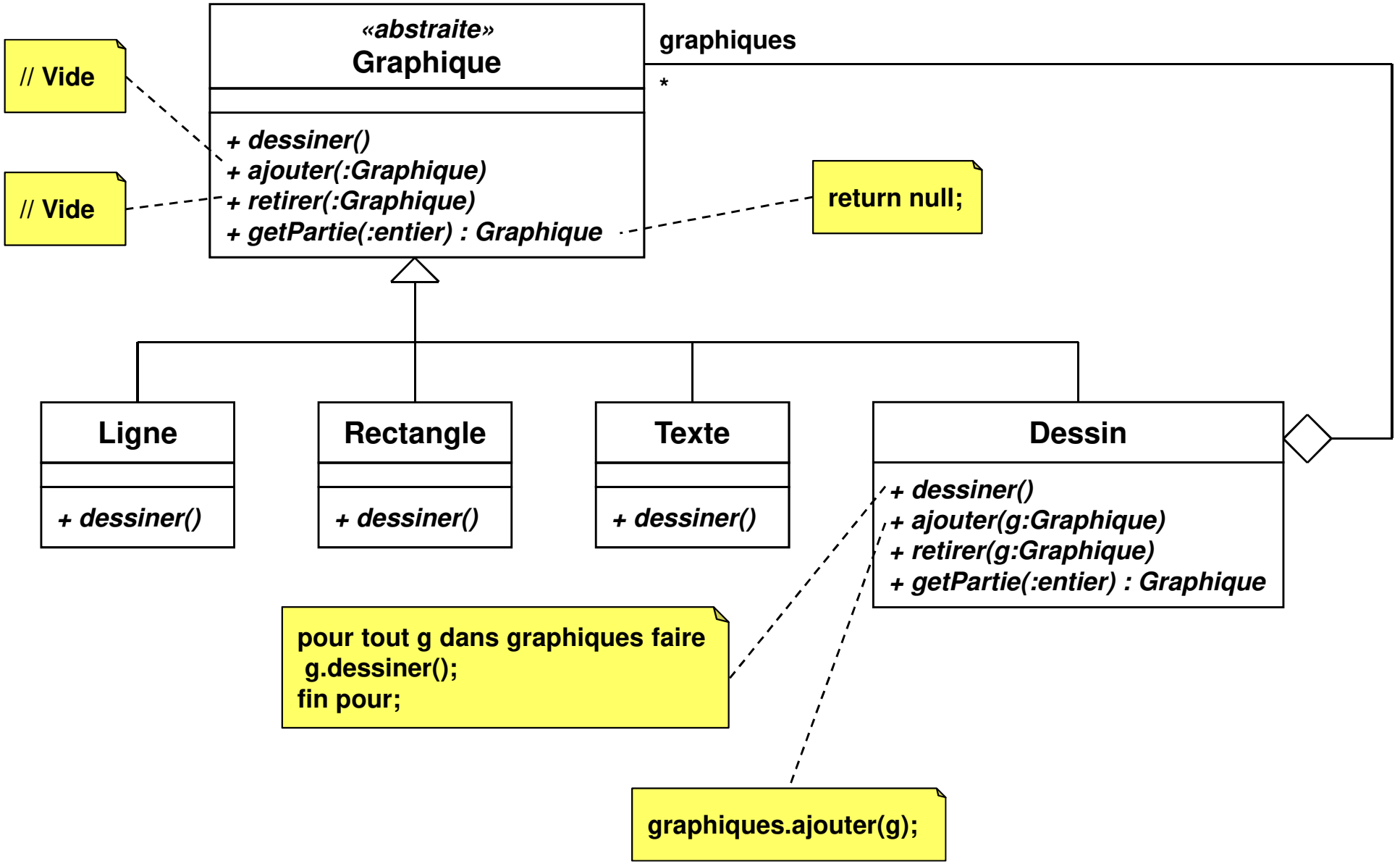
- Objectif
  - Composer des objets sous forme arborescente
  - Objet individuel ou composition traités de la même manière
- Principe
  - Un objet est composé d'autres objets
  - Ces objets peuvent également être des agrégats d'objets

  $\Rightarrow$ récursivité dans la composition
- Motivation
  - Schéma/dessin composé d'objets graphiques
  - Hiérarchie d'héritage des objets graphiques
  - Un objet graphique peut être un groupement d'objets graphiques

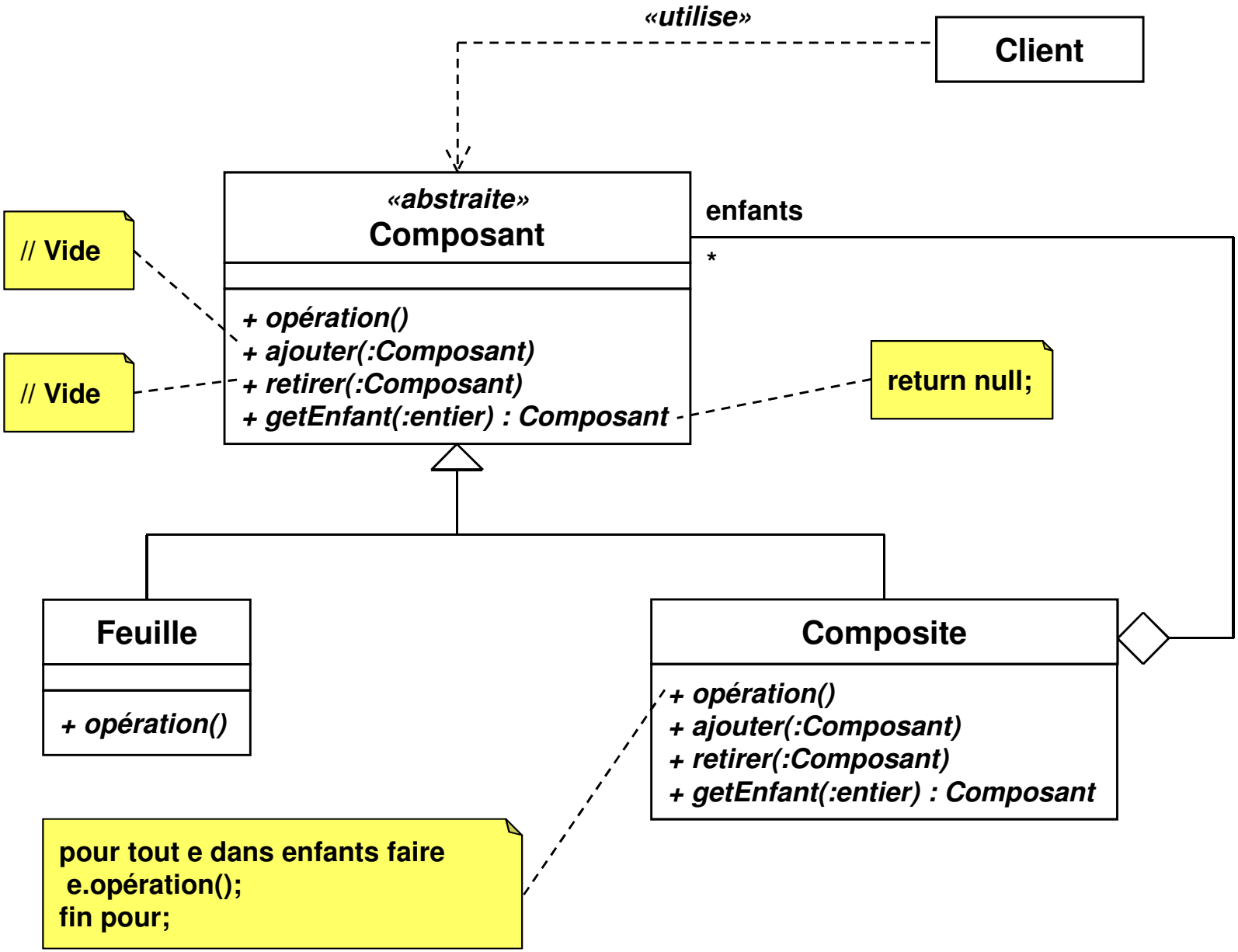
Composite / *Composite (2/5)*



Composite / *Composite (3/5)*



Composite / *Composite (4/5)*



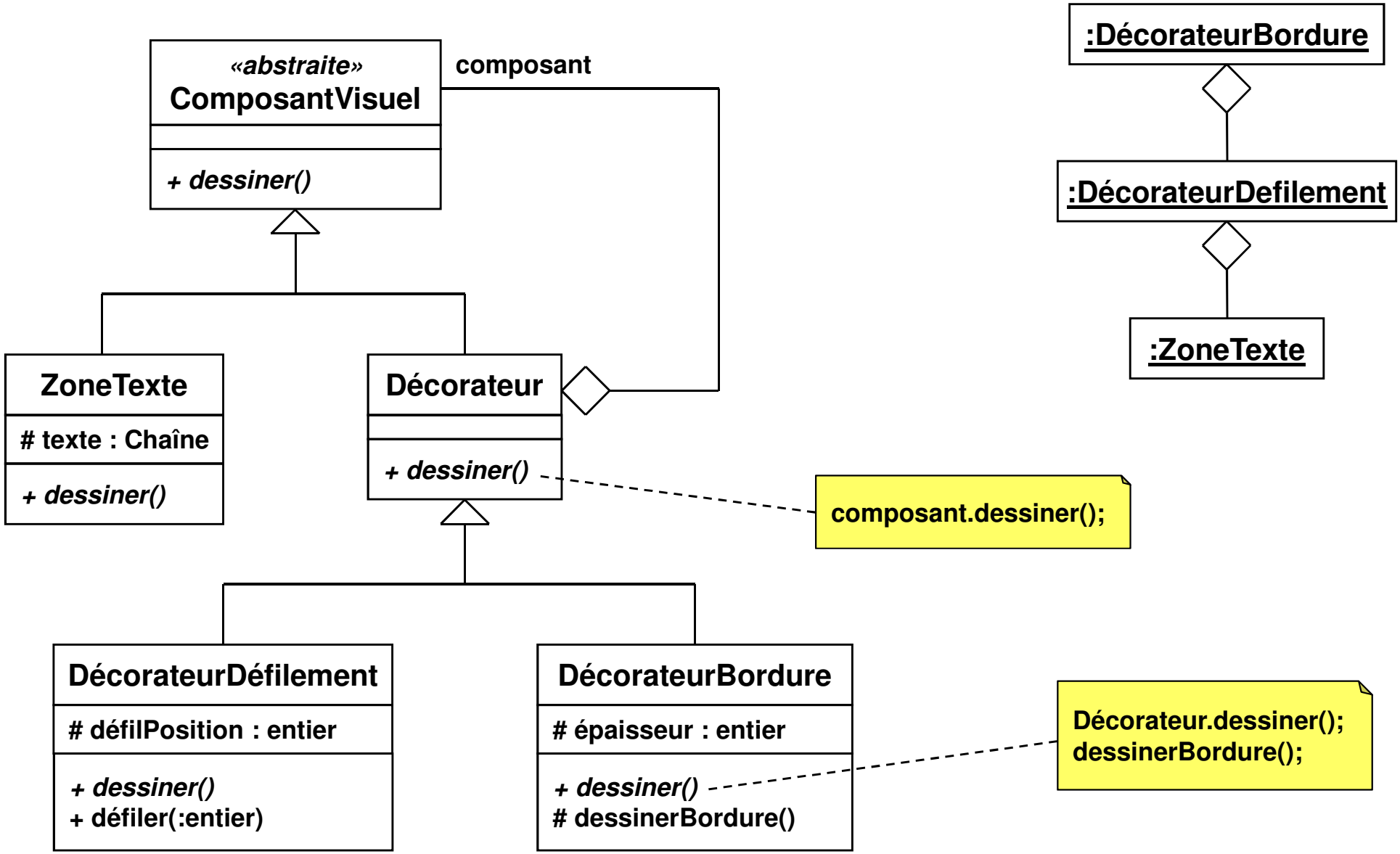
Composite / *Composite (5/5)*

- Le client fait abstraction de la classe réelle des composants
- S'il peut manipuler un objet simple,
  il peut manipuler un agrégat
- L'ajout d'un nouveau type de composant est très simple
  - Sans modification, le client pourra le manipuler
  - Sans modification, il pourra être ajouté dans un composite

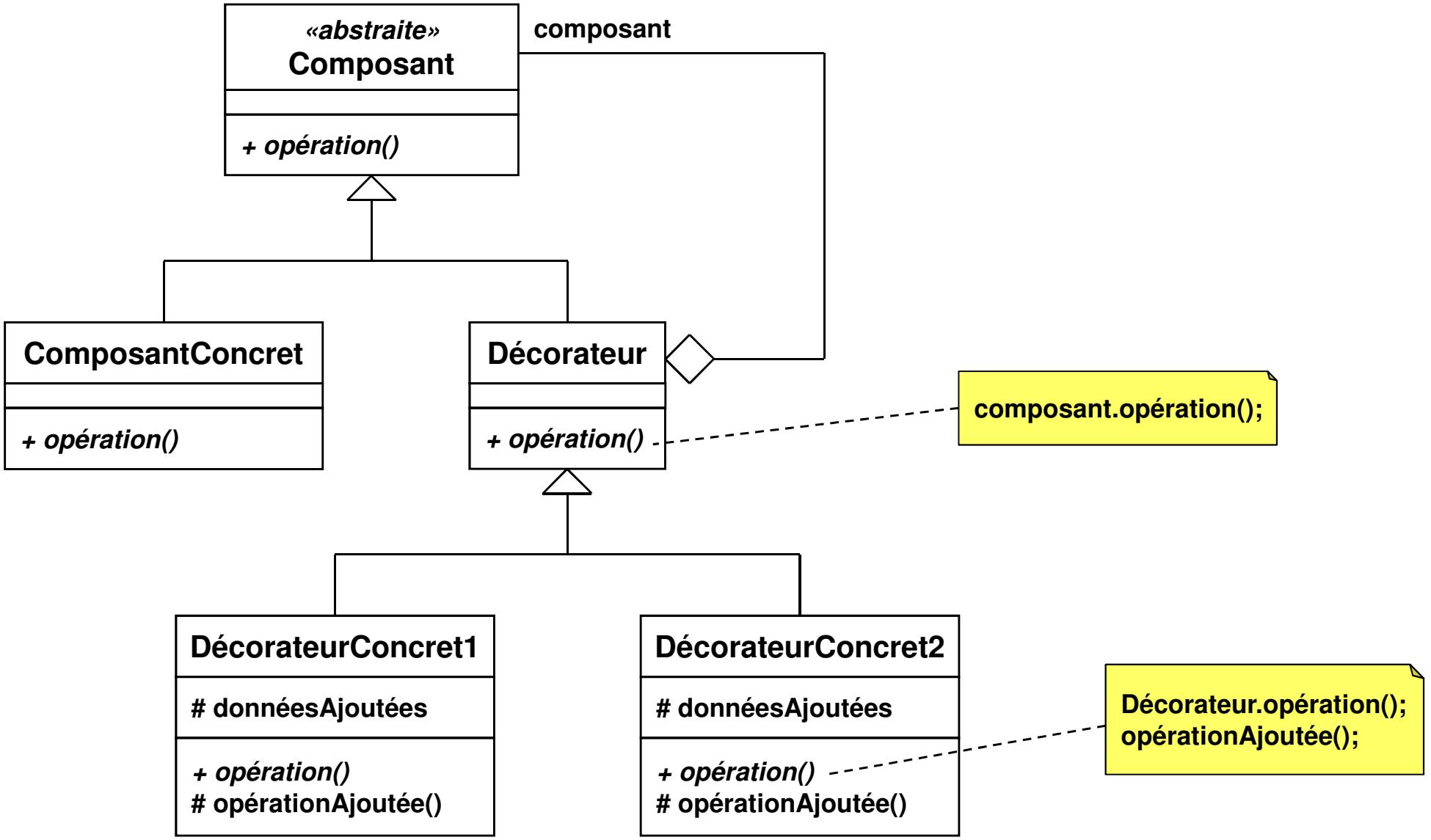
Décorateur / *Decorator (1/4)*

- Objectif
  - Ajouter dynamiquement des fonctionnalités à un objet
  - Alternative à l'héritage pour étendre les fonctionnalités
- Principe
  - Le «décorateur» agrège le composant qu'il adapte
  - Fournit la même interface de base que le composant
  - Il est donc manipulé comme le composant
- Motivation
  - Ajout de fonctionnalités à un composant graphique
  - Eviter l'héritage (car hiérarchie trop complexe)
  - Exemple: zone de texte avec bordure et barre de défilement

Décorateur / *Decorator (2/4)*



Décorateur / *Decorator (3/4)*



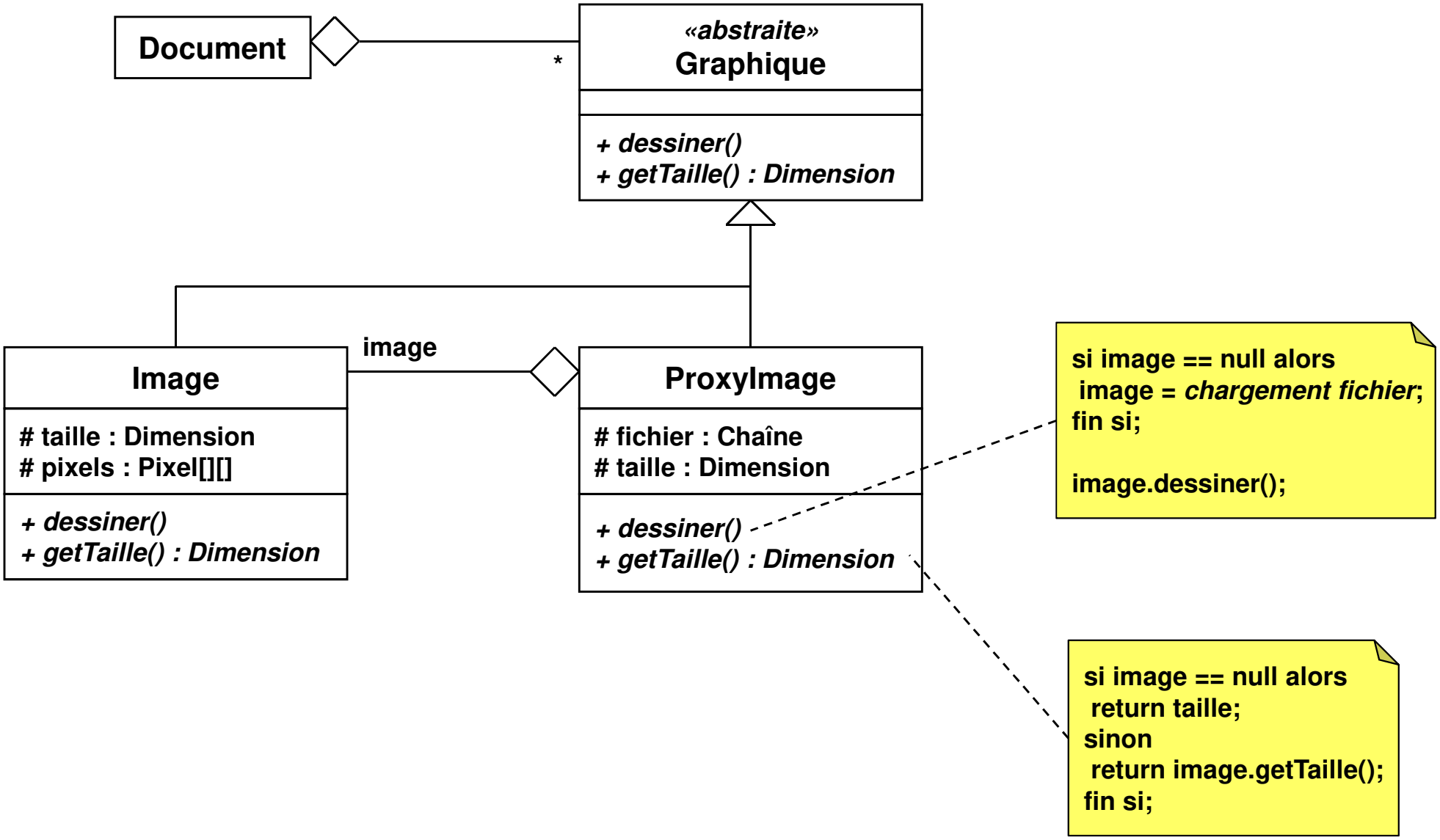
Décorateur / *Decorator (4/4)*

- Evite l'extension par héritage
  - Ajout dynamique de fonctionnalités
  - Ajout individualisé (un seul objet est touché)
- L'héritage pourrait conduire à une hiérarchie lourde
  - Exemple de la zone de texte
  - 3 héritages sont nécessaires (bordure, défilement, les deux)
  - Extension de la zone de texte $\Rightarrow$ extension des 3 classes
- Mais le décorateur ajoute un objet à chaque décoration

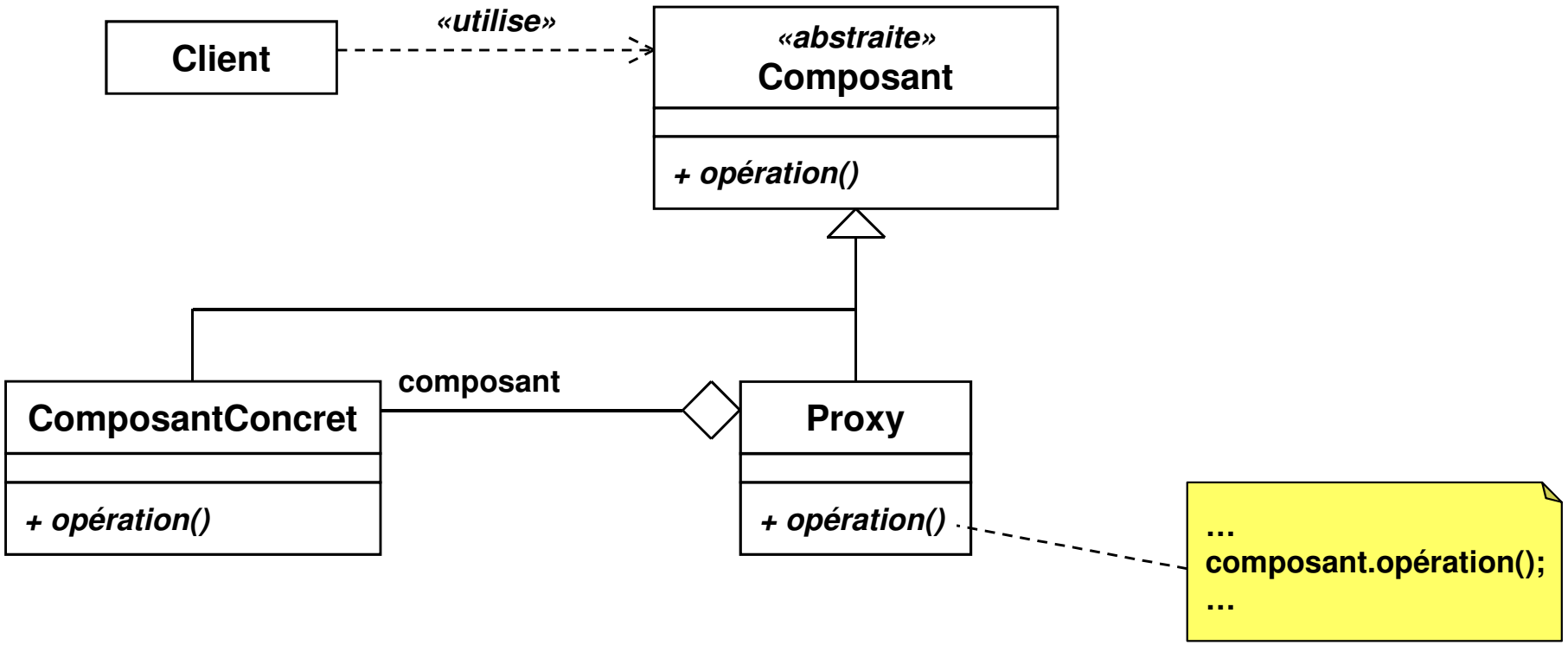
Proxy / *Proxy (1/4)*

- Objectif
  - Fournir un substitut, un intermédiaire, pour accéder à un objet
  - Permettre ainsi de contrôler l'accès
- Principe
  - Le substitut, le «proxy», possède la même interface que l'objet
  - Lorsqu'il reçoit un message, il le transmet à l'objet
  - Il peut effectuer un contrôle sur le message
    - Refuser de le retransmettre
    - Différer la retransmission
    - Altérer le message
- Motivation
  - Différer la création d'un objet car elle est coûteuse
  - Exemple: chargement d'un document avec des images
    - Différer la lecture des images au moment où celles-ci sont visibles

Proxy / *Proxy (2/4)*



Proxy / *Proxy (3/4)*



Proxy / *Proxy (4/4)*

- Abstraction de l'accès à un objet
  - Niveau d'indirection supplémentaire
- Permet une représentation locale d'un objet distant
  - Autre zone mémoire, sur disque ou réseau
- Permet des optimisations d'exécution des méthodes
  - Technique de cache
  - Création différée («*lazy*»)

Patrons de comportement *(1/2)*

- 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

Patrons de comportement *(2/2)*

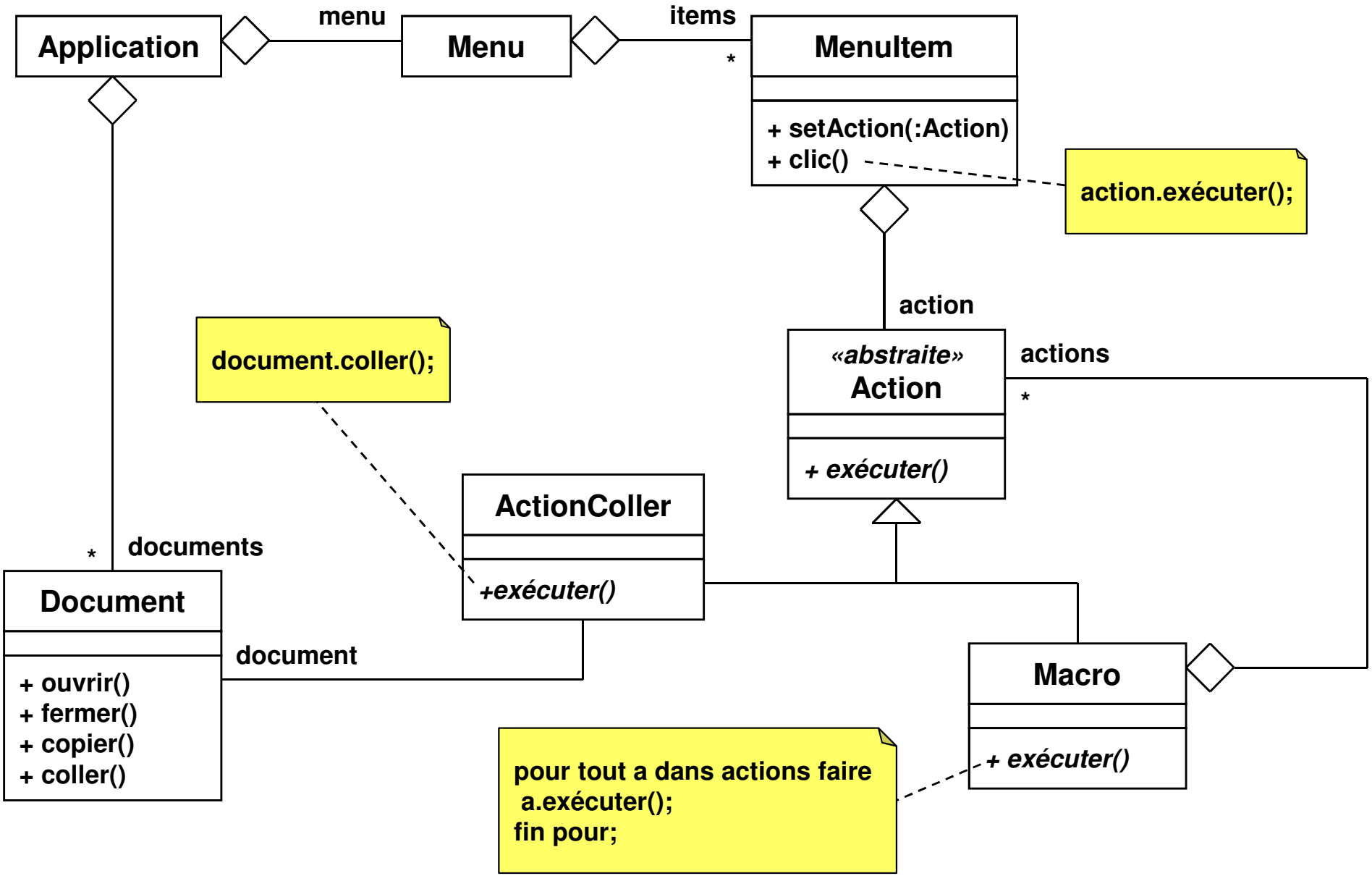
- Permet l'assemblage de composants
  - Pour obtenir une fonctionnalité plus élaborée
  - Algorithmes vus comme des objets

- 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

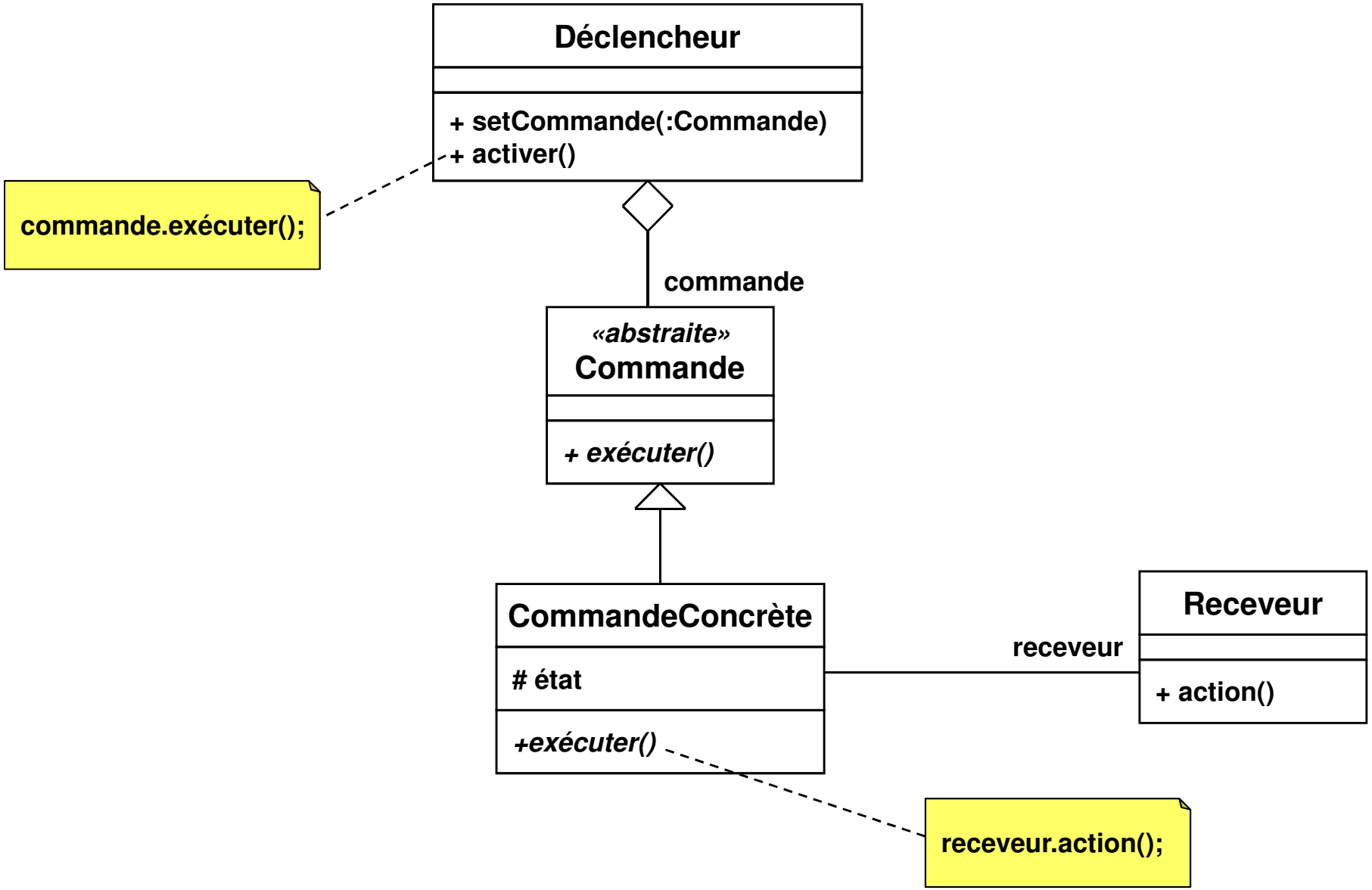
Commande / *Command (1/3)*

- Objectif
  - Encapsuler une action dans un objet
  - Permet l'abstraction de l'action (découplage déclencheur/receveur)
  - 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 (2/3)*



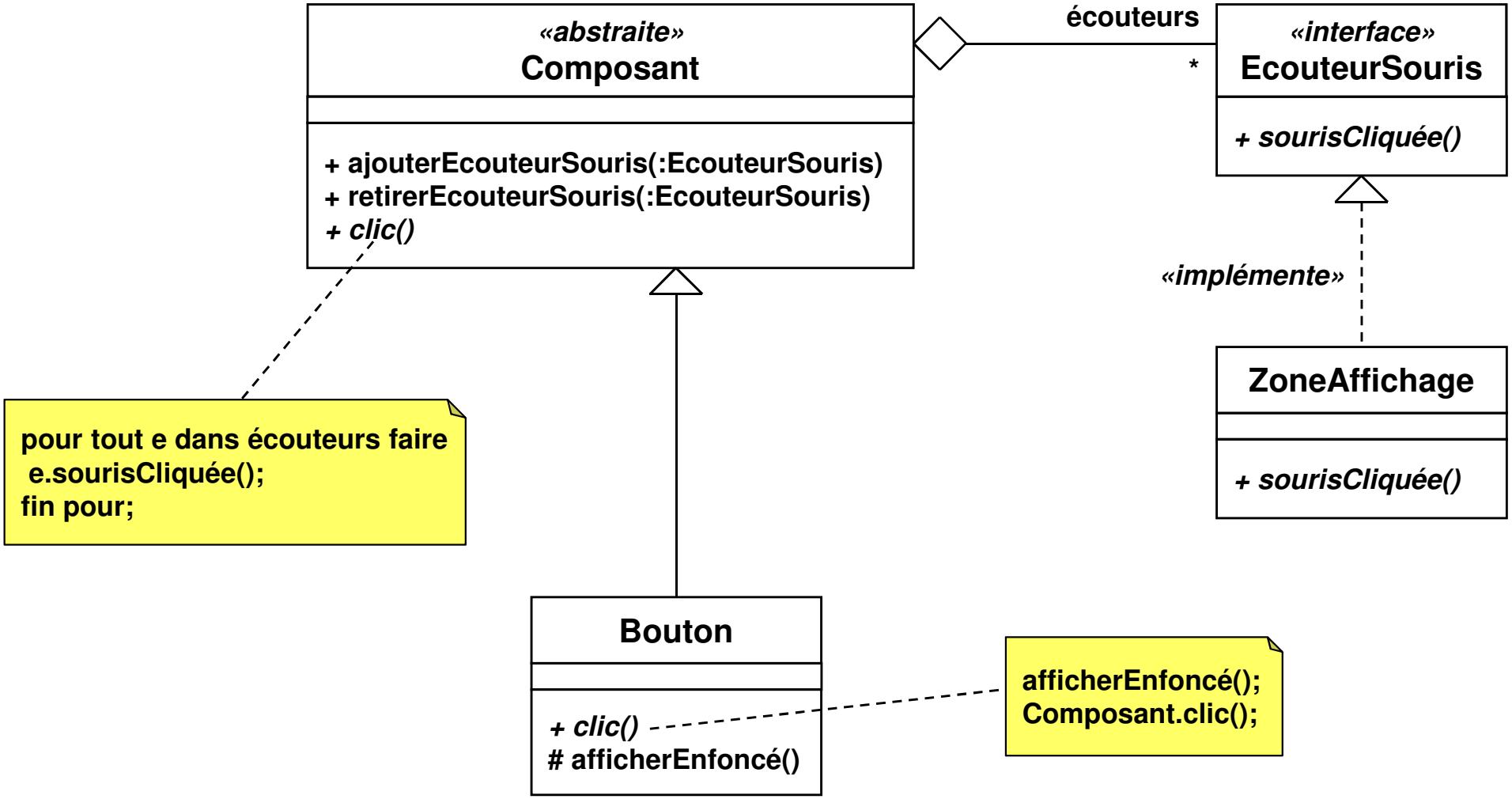
Commande / *Command (3/3)*



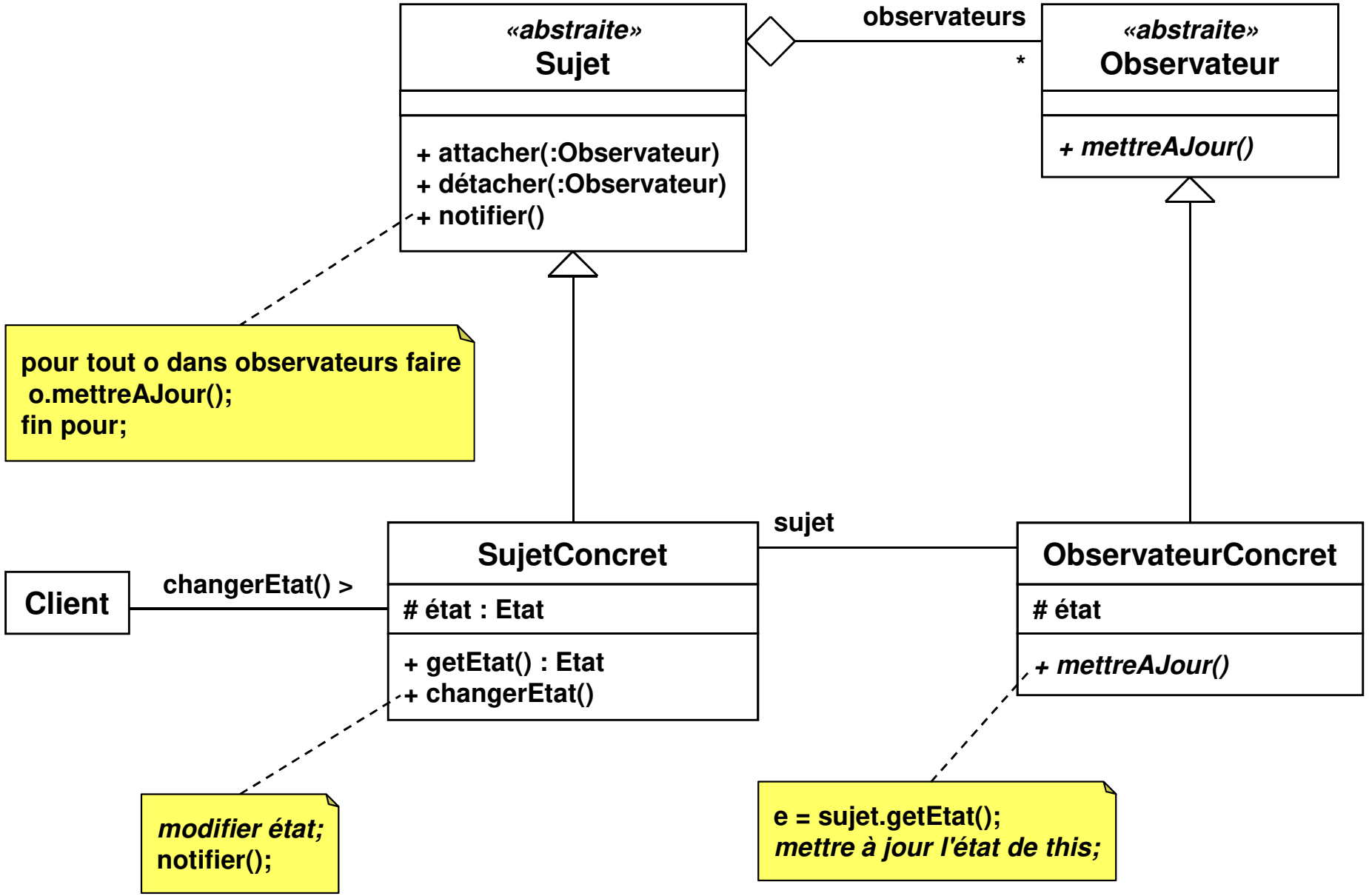
Observateur / *Observer (1/4)*

- 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)*



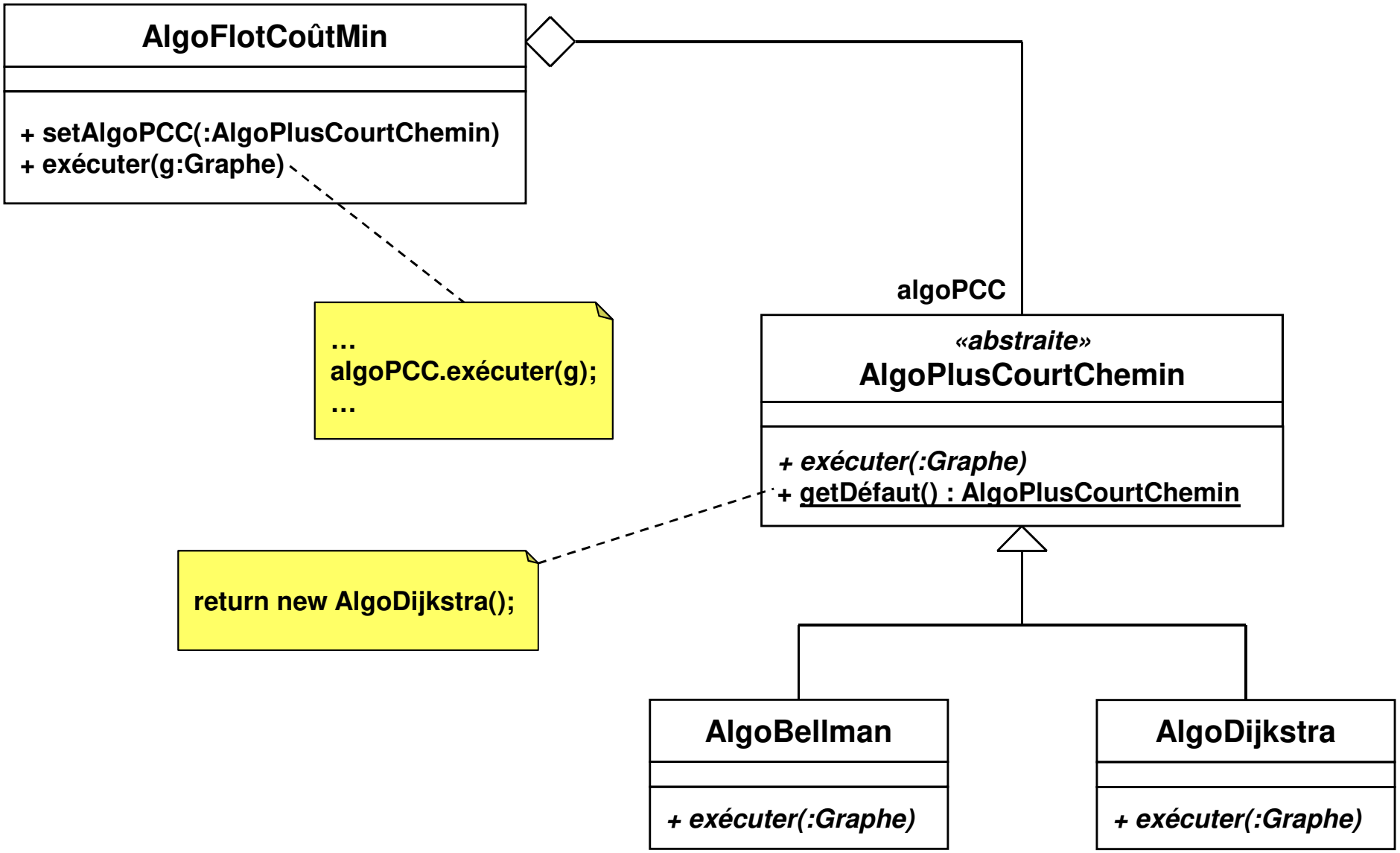
Observateur / *Observer (4/4)*

- 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
  - En cas de chaînage des observations

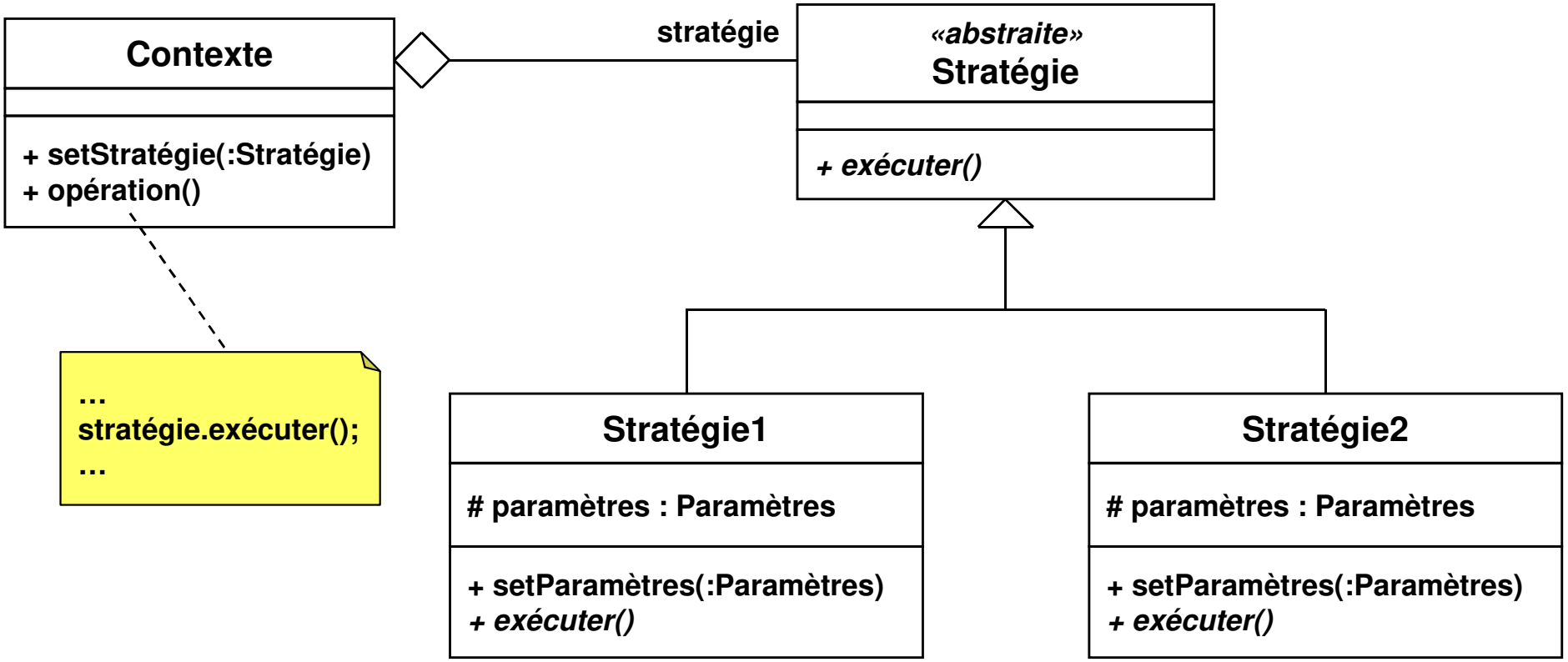
Stratégie / *Strategy (1/4)*

- 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 redéfinition 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)*



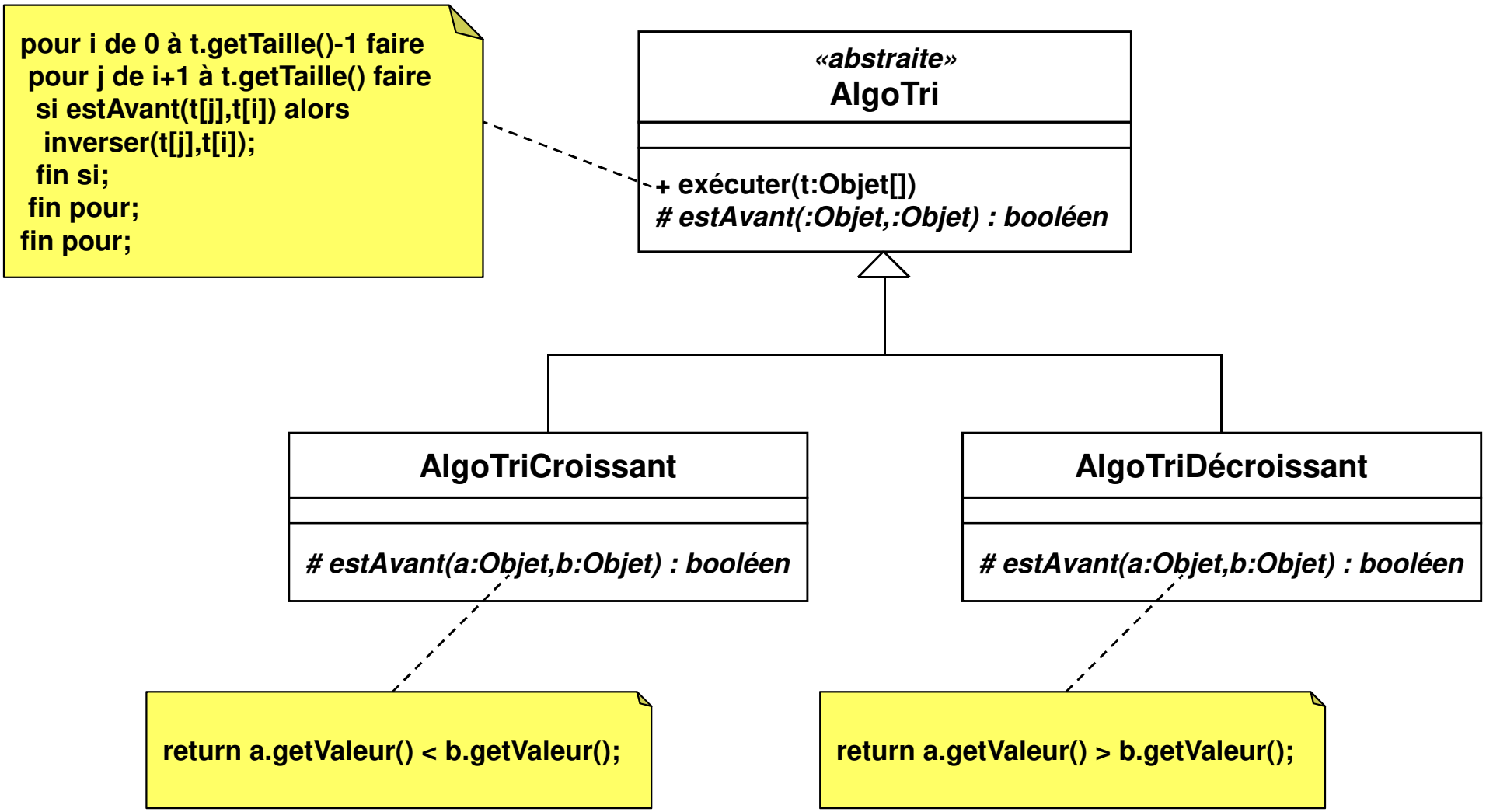
Stratégie / *Strategy (4/4)*

- Abstraction de la stratégie
  - Interchangeable dynamiquement
  - Nouvelle stratégie $\Rightarrow$ aucun impact sur le contexte
- La stratégie est un paramètre du contexte
- Contenu classique d'une classe stratégie/algorithme
  - Un point d'entrée
    - Méthode publique appelée pour exécuter l'algorithme
  - Des sous-algorithmes
    - Méthodes protégées ou privées utilisées par le point d'entrée
  - Des paramètres
    - Mémorisés dans des attributs
    - Constructeurs et accesseurs nécessaires pour l'initialisation

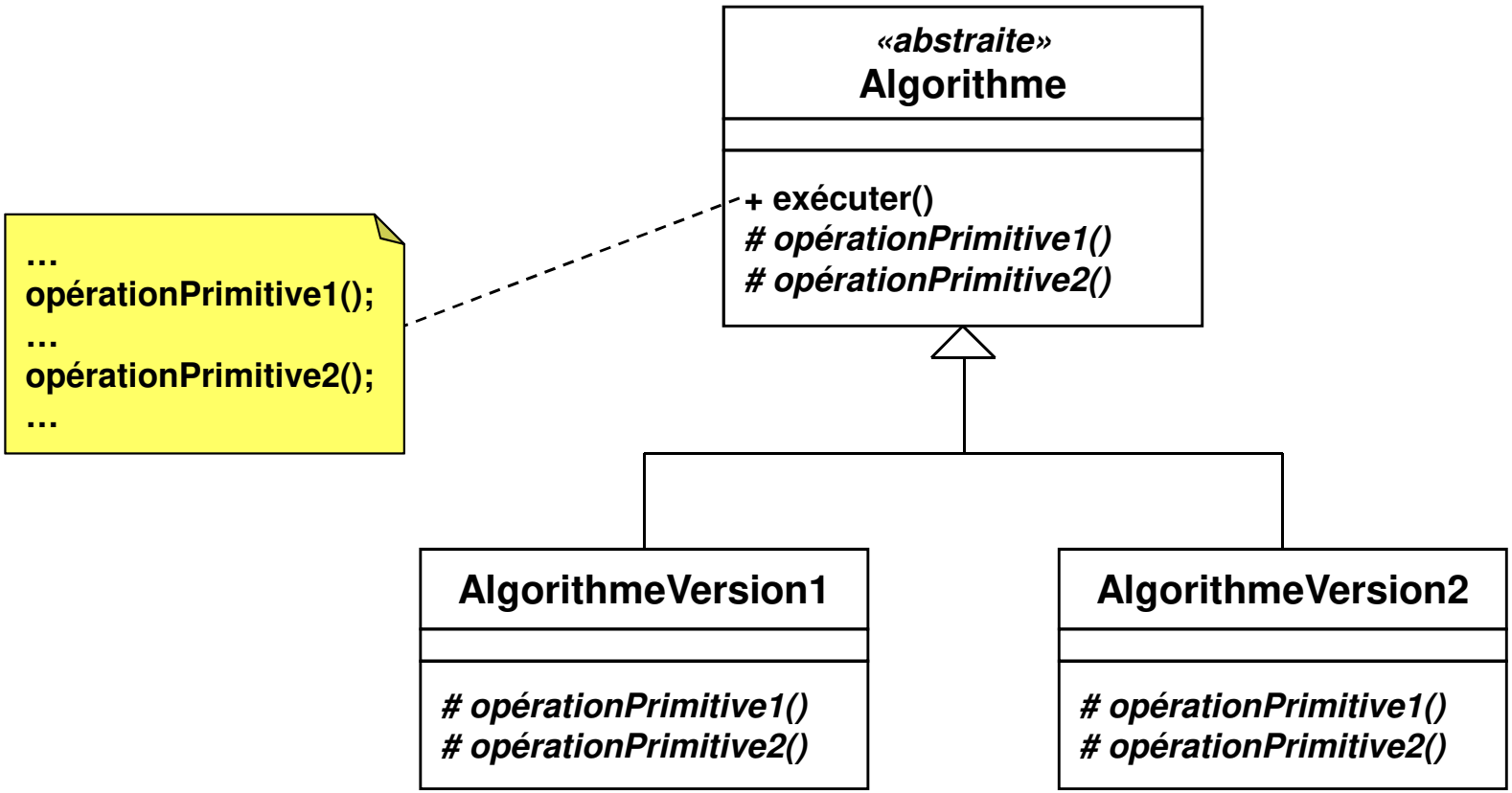
Méthode patron / *Template Method (1/5)*

- 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 redéfinies
- Motivation
  - Proposer plusieurs variantes d'un algorithme
  - Où la structure générale de l'algorithme est inchangée

Méthode patron / *Template Method (2/5)*



Méthode patron / *Template Method (3/5)*



Méthode patron / *Template Method (4/5)*

- 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 ⇒ surcoût lié à la virtualité
  - Trop de méthodes ⇒ redéfinition fastidieuse pour l'utilisateur
- Utilisé pour la redéfinition «par complément»
  - Objectif: redéfinir pour compléter une méthode
  - Problème: il ne faut pas oublier d'appeler la version mère
  - Solution: utiliser une méthode patron

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

- Redéfinition par complément
  - Approche classique

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

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

  - Approche avec méthode patron

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

class Fille : public Mere {
 protected: void autreChose() override { /* Autre chose */ }
};
```

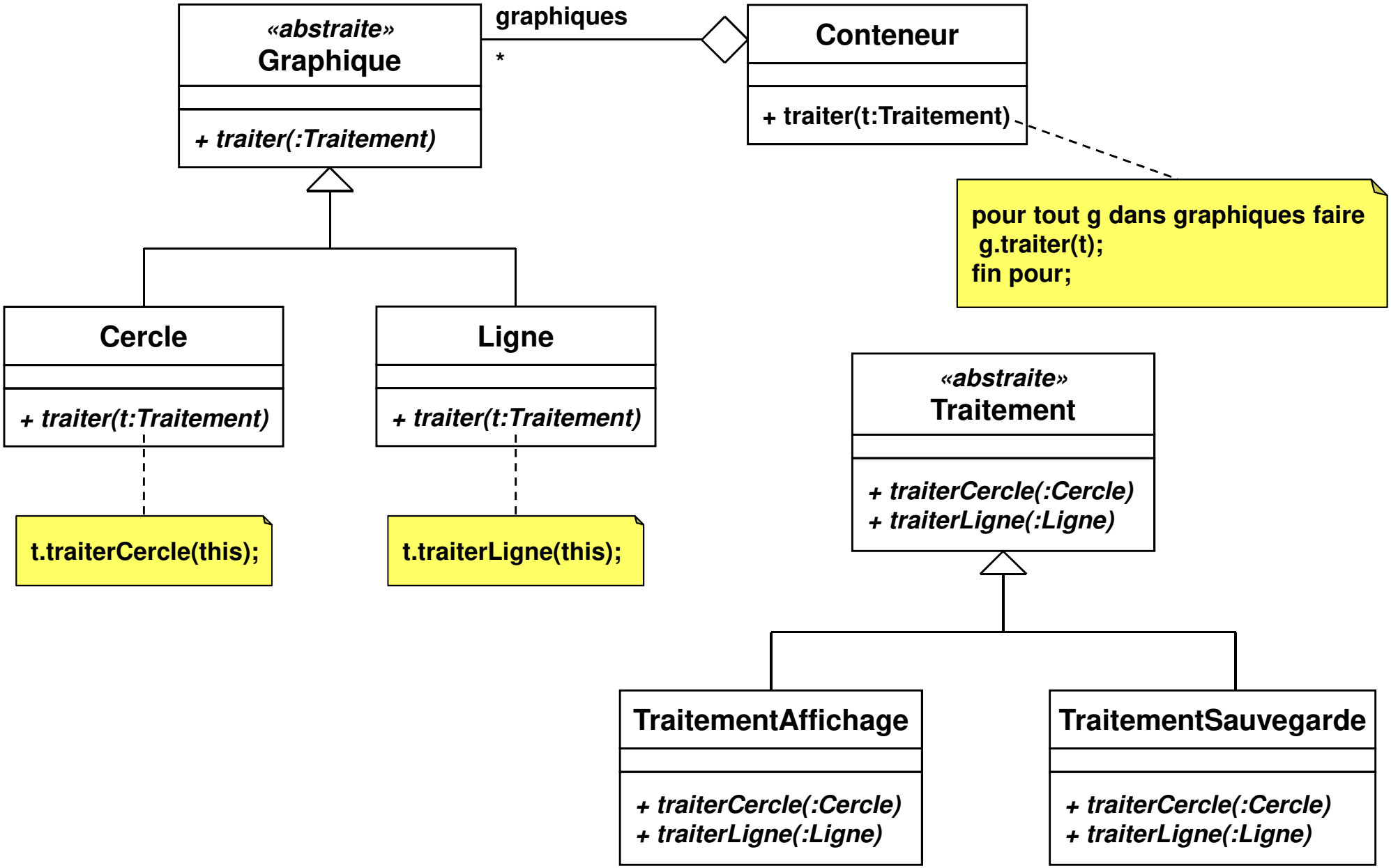
Visiteur / *Visitor (1/4)*

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

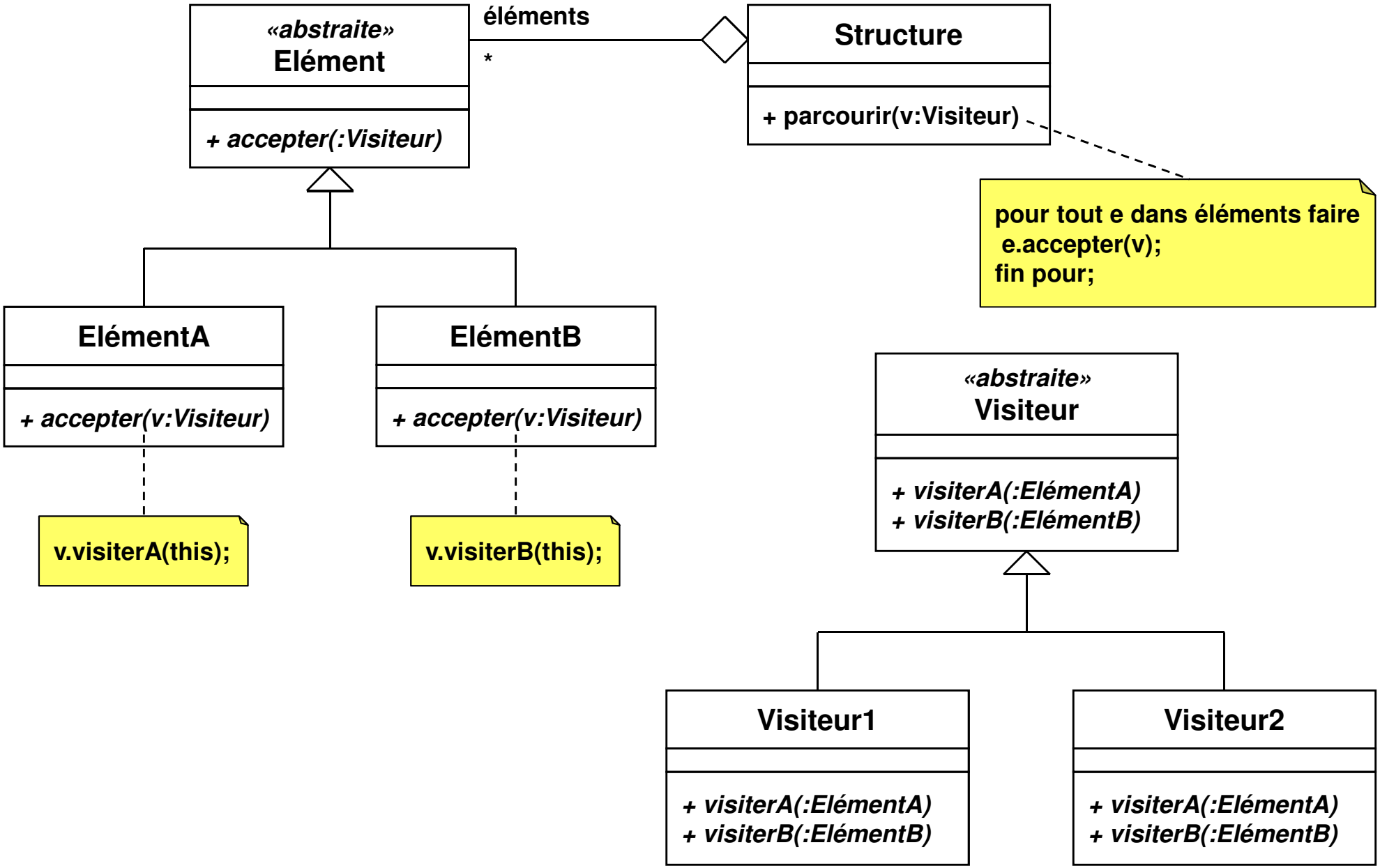
- Principe
  - L'opération est modélisée par un objet, le «visiteur»
    - Une classe, extensible, représente l'opération
  - Les éléments doivent «accepter» un visiteur
    - Une méthode doit recevoir le visiteur
    - Et appliquer l'opération associée sur l'élément
  - Une procédure de parcours applique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



Visiteur / *Visitor (4/4)*

- Propose plusieurs traitements sur les éléments
  - Sans alourdir l'interface de la structure
  - Sans alourdir l'interface des éléments
- Facilite l'ajout d'un nouveau traitement
  - Il suffit de créer un nouveau visiteur
  - Et définir le traitement pour chaque type d'éléments
- Mais plus laborieux d'ajouter un nouveau type d'éléments
  - Il faut ajouter une méthode dans chaque visiteur
  - Pour définir chaque traitement pour le nouveau type d'éléments