ISIMA

PARTIE VII

Patrons de conception

(Design Patterns)

Bruno Bachelet

Christophe Duhamel

Luc Touraille

Plan

- Introduction
  - Motivations
  - Réutilisation au niveau conceptuel
- Description
  - Référencement des patrons
  - Les patrons du GoF
  - Classification des patrons
- Patrons de création
  - Processus de création d'objets
- Patrons de structure
  - Structure d'objets complexes
- Patrons de fonctionnement
  - Organisation des algorithmes

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)*

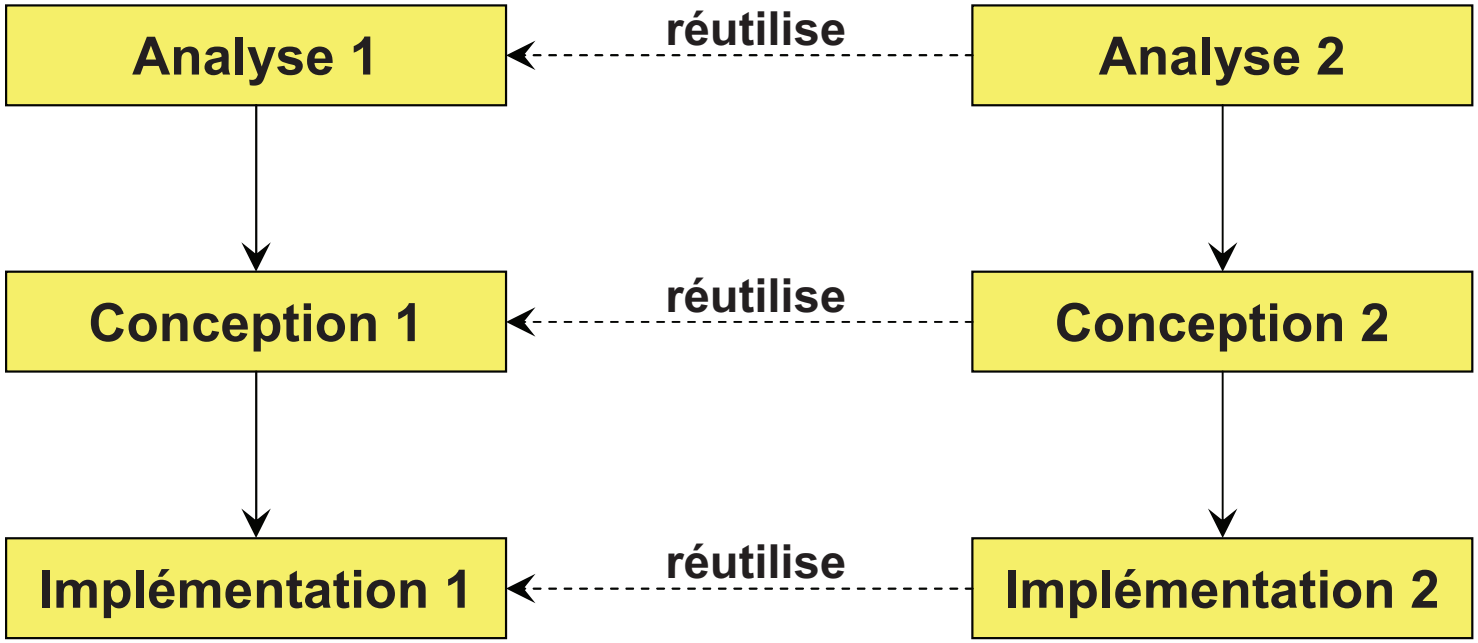
- Bénéficier des bonnes pratiques de l'industrie
  - Minimiser les risques dans la phase de développement
  - Se référer à l'existant
  - Reprendre des solutions éprouvées

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

    $\Rightarrow$ Bibliothèques logicielles
  - Au niveau conception
    - Mêmes organisations des composants

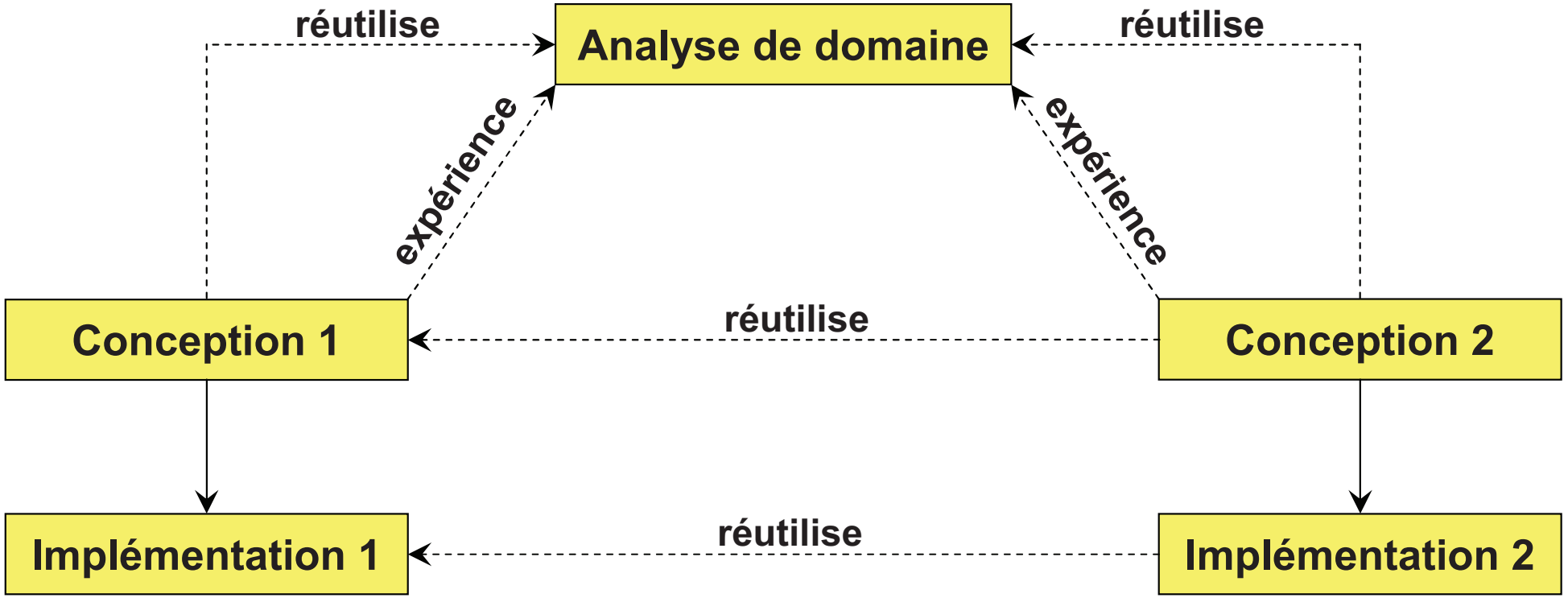
    $\Rightarrow$ Patrons de conception (ou *«design patterns»*)

Réutilisation: niveaux d'abstraction



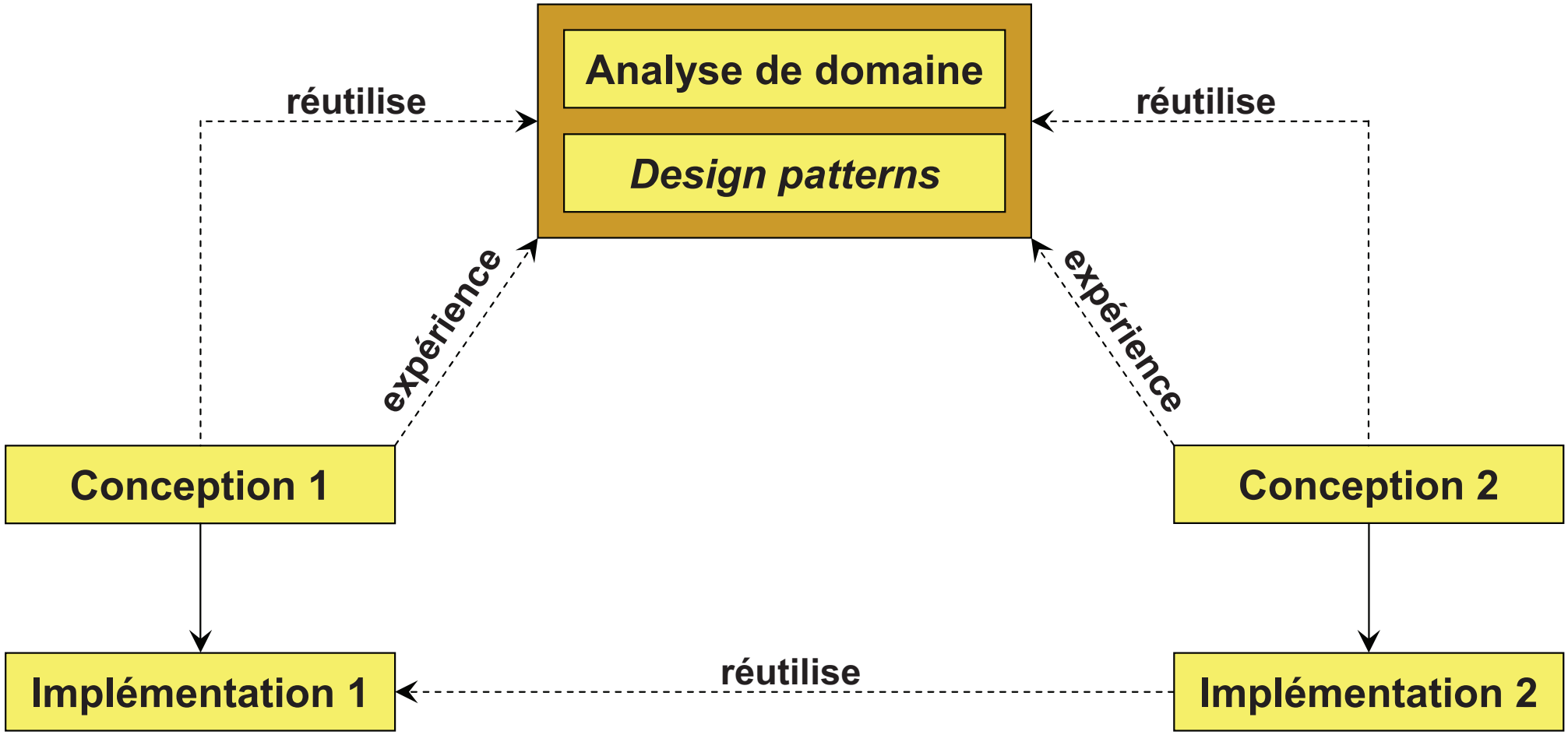
- A chaque nouvelle expérience, on peut réutiliser
- Sans outil particulier: réutilisation niveau par niveau

Réutilisation: analyse de domaine



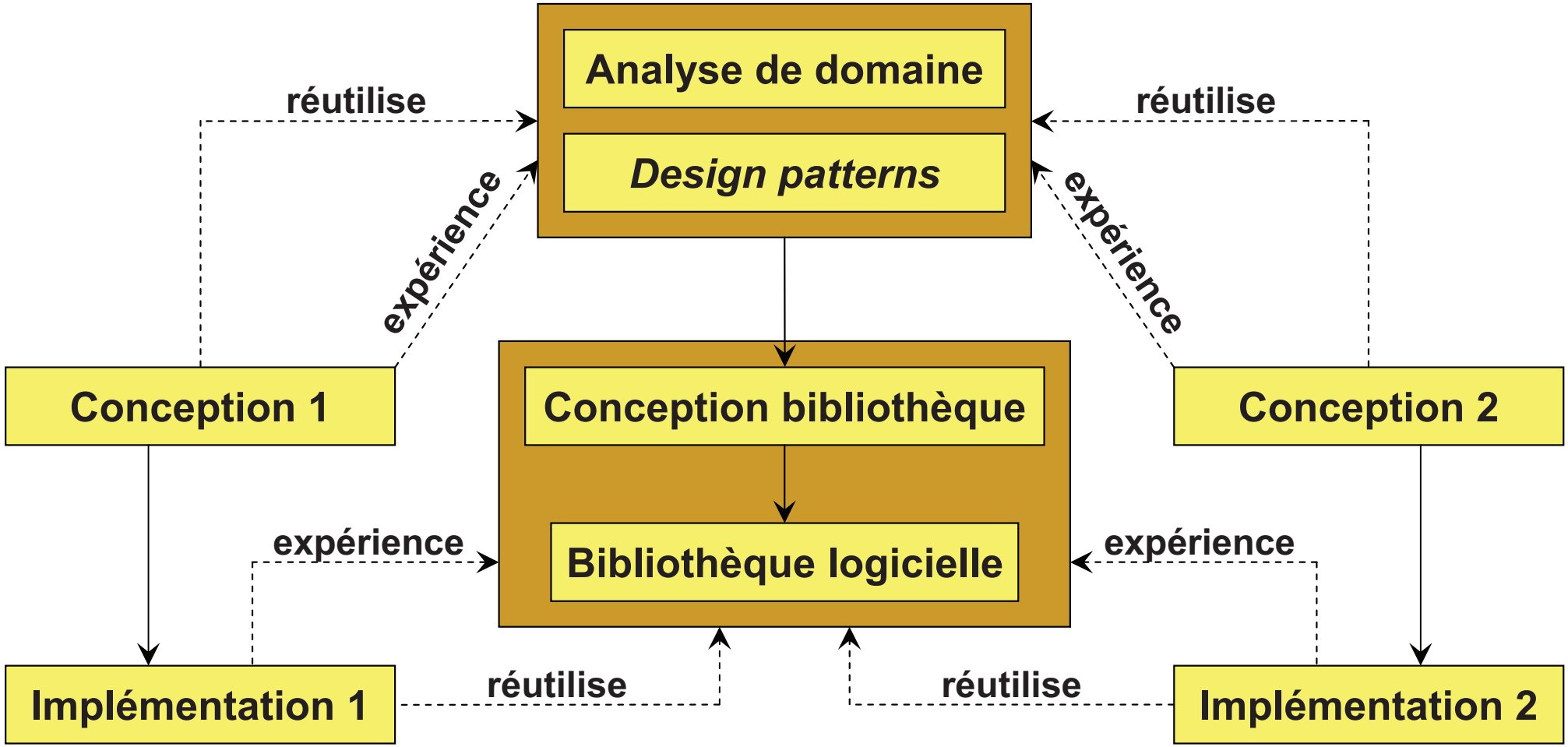
- Profiter de plusieurs expérience du même domaine

Réutilisation: patrons de conception



- Problèmes de conception récurrents ⇒ Patrons de conception
- Solutions génériques réutilisables au niveau conception

Réutilisation: cadriciel (*framework*)



- Réutilisation au niveau implémentation ⇒ Bibliothèques logicielles
- Cadriciel = composants réutilisables (conception + implémentation)

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 «élégante» 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

Référencement des *design patterns (1/2)*

- Tentatives de référencement les patrons de conception

- Livre fondateur
  - «Design Patterns: Elements of Reusable Object-Oriented Software»
  - Gamma, Helm, Johnson, Vlissides
    - Surnommé le «*GoF*» (*Gang of Four*)
  - Addison-Wesley, 1994
  - 23 patrons de conception

- Lecture conseillée
  - «Design patterns tête la première», O'Reilly
  - «Pour mieux développer avec C++», Dunod
  - «Modern C++ Design», Addison-Wesley

Référencement des *design patterns (2/2)*

- Les patrons présentés ici sont issus du *GoF*

- Mais ce ne sont pas les seuls !
  - Patron MVC (Modèle Vue-Contrôleur)
    - Il n'est pas dans la liste du *GoF*
  - Patrons GRASP
    - Proposés par Craig Larman
    - Plus conceptuels

- Communauté active
  - De nouveaux patterns proposés régulièrement
  - Démocratique: adoptés si utilisés et généraux
  - Exemples (cf. Wikipedia)
    - *Reversible command (undo)*
    - *Lazy initialization*
    - Patrons de concurrence

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

- Quatre éléments principaux définissent un patron
- Objectif
  - Description de son utilité
- Problème / Motivation
  - Quand appliquer le patron de conception
  - Relations problématiques entre les classes
- Solution proposée
  - Eléments impliqués
  - Leurs relations
  - Schémas conceptuels (e.g. diagrammes UML)
- Conséquences
  - Compromis éventuels
  - Qualité de la solution

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 $\Rightarrow$ Patrons de création
  - Assemblage de composants $\Rightarrow$ Patrons de structure
  - Comportement des composants $\Rightarrow$ Patrons de comportement

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 généraux des *design patterns*

- Favoriser une bonne conception
- Principes
  - Responsabilité unique
  - Connaissance minimale
  - Ouvert/fermé
  - Encapsuler ce qui varie
  - Programmer envers une interface
  - Favoriser la composition à l'héritage

Qu'est-ce qu'une bonne conception ?

- Facile à appréhender
- Facile à faire évoluer
- Résistant aux changements

$\Rightarrow$ Quelques principes permettent de tendre à ces buts

Responsabilité unique

- Faire une seule chose, et le faire bien
  $\Rightarrow$ cohésion forte des classes et des modules

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

- Exemple: séparer le calcul de données de leur lecture/écriture dans un fichier

Connaissance minimale

- Ne parler qu'à ses «connaissances» proches
- Loi de Déméter
  - Invocation de méthodes sur
    - Soi-même
    - Paramètres de méthode
    - Objets créés (variables locales)
    - Attributs
  - Éviter d'appeler les méthodes d'un objet retourné par une autre méthode
- Limite les couplages
- Facilite la compréhension

Ouvert/fermé

- Etre ouvert aux extensions...
  - Permettre l'ajout de fonctionnalités
  - Permettre la modification du comportement
- ...mais fermé aux modifications
  - Le code d'un module ne devrait pas devoir être modifié si les besoins changent
- Besoins changent régulièrement
  $\Rightarrow$ nécessité de pouvoir évoluer
- Tout en évitant de casser du code existant

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
- Exemples
  - Comparateur dans un algorithme de tri
  - Thèmes d'affichage

Programmer envers une interface

- Programmer envers une interface
  et pas une implémentation

- Module sans dépendance
  avec les détails d'implémentation
  - Modification de l'implémentation sans impact sur le module
  - Changement d'implémentation facilitée

- Exemple (Java)
  - Dépendre de `Collection` plutôt que de `ArrayList`

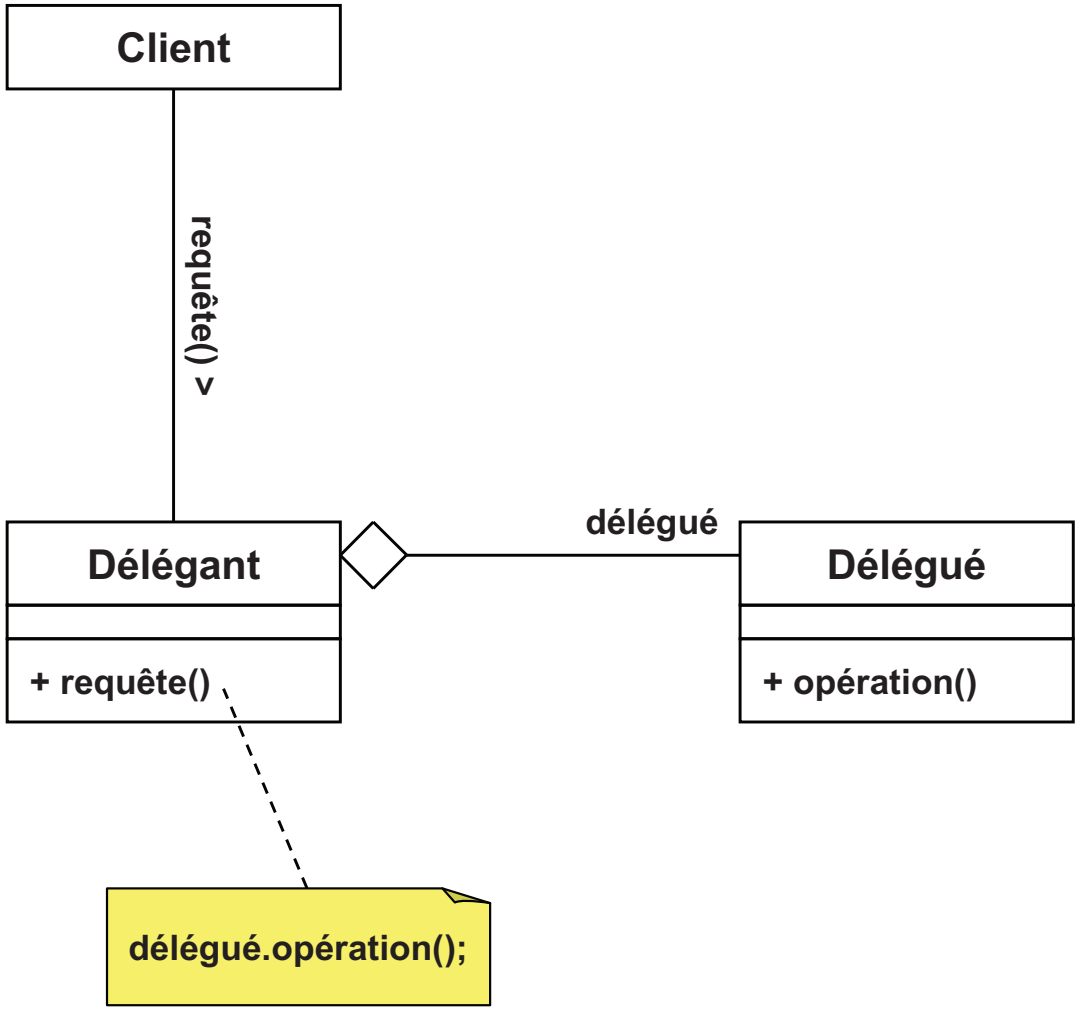
Favoriser la composition à l'héritage

- Héritage → statique
- Composition → dynamique
- Utilisation de la délégation
  - Couplage plus faible
  - Changement de fournisseur de services à l'exécution
- Évite souvent des héritages sans réelle relation «est un»
  - Principe de substitution de Liskov
- Exemple
  - `EnregistreurResultats` agrège `Writer`
  - Plutôt que de spécialiser `XMLWriter`

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
  - Pas de changement de message (e.g. *Proxy*)
  - Changement de message (e.g. *Adapter*)
  - Changement de délégué (e.g. *Chain of Responsibility*)
  - Surcharge du message (e.g. *Decorator*)

Mécanisme de délégation *(2/2)*



Injection de dépendance

- Injecter une implémentation dans un objet
  - Soit simplement (constructeur, *setter*)
  - Soit avec un *framework*
- Découple de l'implémentation
- Facilite le changement d'implémentation
- Facilite les tests
  - Injection d'objets *mocks*/*stubs*
  - Exemple: émulation d'un réseau, d'une BdD... pour contrôler l'environnement de test

Inversion de contrôle

- Principe d'Hollywood
  - «Ne nous appelez pas, nous vous appellerons»
- Flot d'exécution contrôlé par bibliothèque/*framework*
- Focalisation du développeur sur les aspects métiers
- Exemples
  - Patron de méthode (algos STL)
  - Serveurs d'application (Java EE)

ISIMA

Patrons de création

(PARTIE VII - Patrons de conception)

Bruno Bachelet

Christophe Duhamel

Luc Touraille

Patrons de création *(1/2)*

- 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
  - Une tendance actuelle des systèmes logiciels
  - *Component-Based Development*
  - Introduit de la flexibilité dans l'assemblage

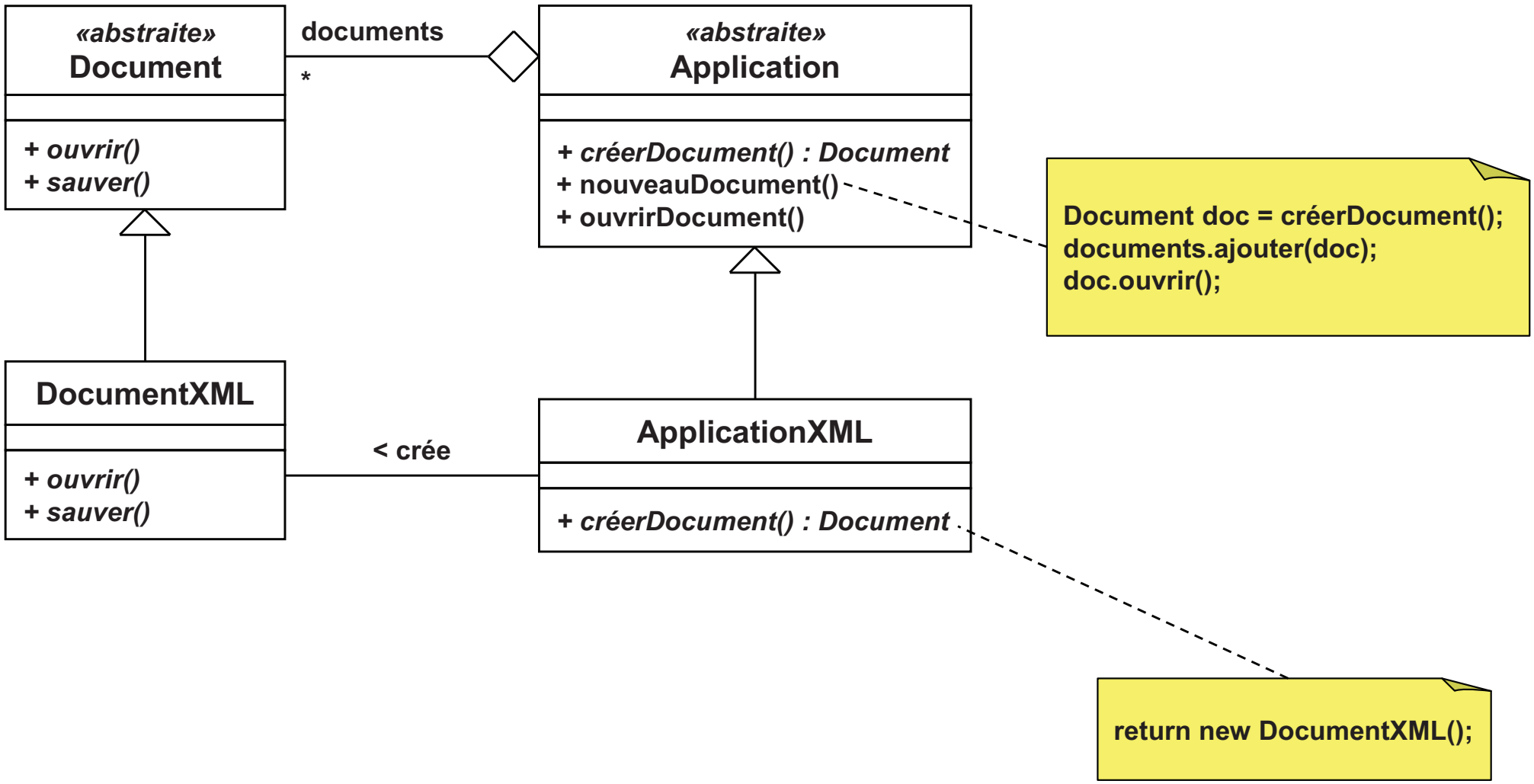
Patrons de création *(2/2)*

- (Méthode) Fabrique / *Factory Method*
  - Déléguer la création d'un objet
- Fabrique abstraite / *Abstract Factory*
  - Créer une famille d'objets cohérents
- Monteur / *Builder*
  - Séparer la construction d'un objet complexe de sa représentation
- Prototype / *Prototype*
  - Créer un objet par clonage d'une instance modèle
- Singleton / *Singleton*
  - Garantir une seule instance pour une classe

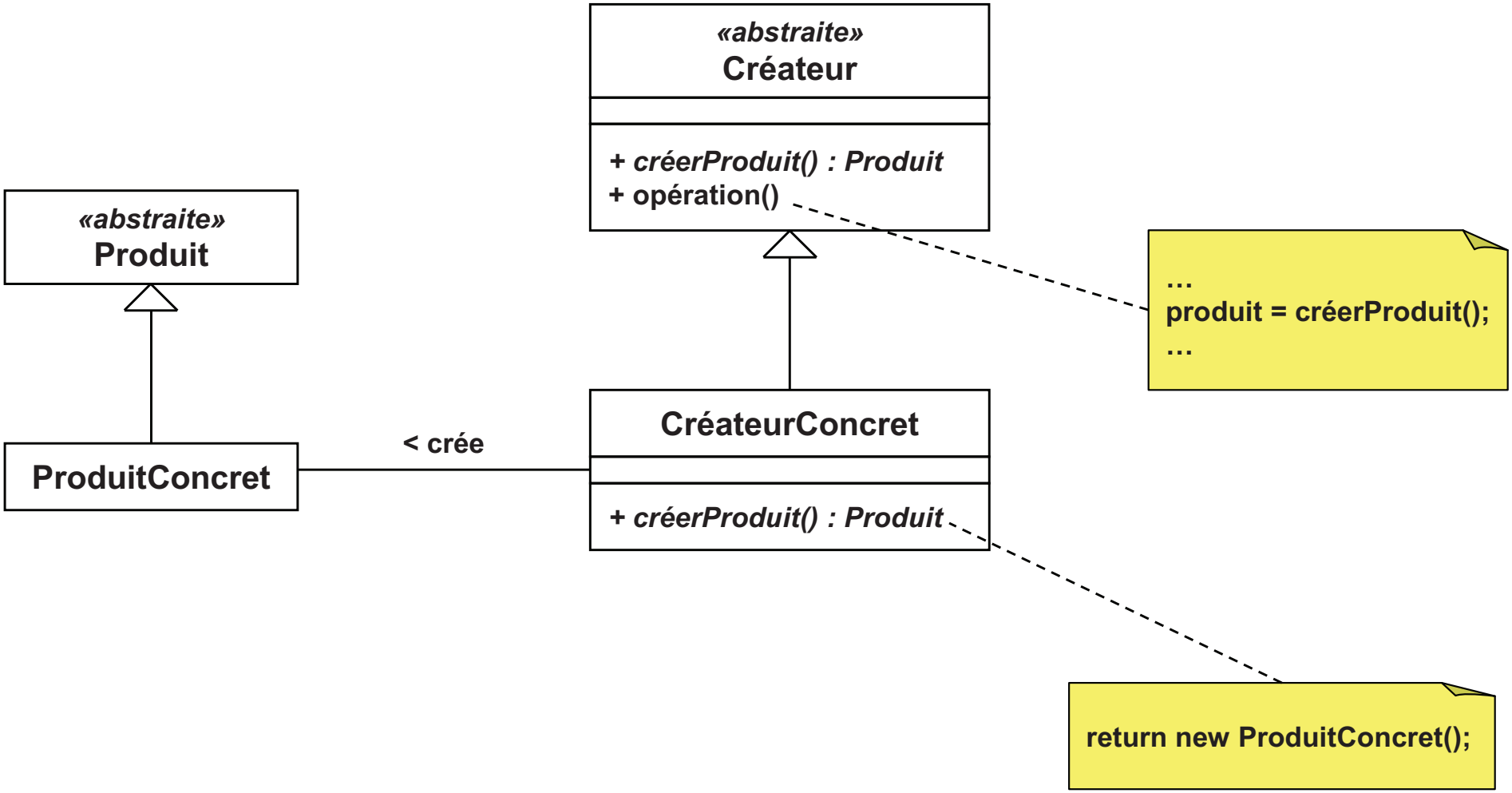
(Méthode) Fabrique / *Factory Method (1/4)*

- Objectif
  - Déléguer la création d'un objet
- Principe
  - Définir une interface pour créer un objet ⇒ le «créateur»
    - Le processus de création peut être changé par héritage
  - Le client demande au créateur de lui fournir une instance
    - Le client ne connaît que l'interface de l'objet
    - Seul le créateur connaît la classe réelle de l'objet
- Motivation
  - Application qui manipule des documents de types différents

(Méthode) Fabrique / *Factory Method (2/4)*



(Méthode) Fabrique / *Factory Method (3/4)*



(Méthode) Fabrique / *Factory Method (4/4)*

- Méthode «`créerProduit`» = méthode «fabrique»
- Appelé aussi «constructeur virtuel»
- Intérêt
  - Isolation de la classe concrète
    - Créateur responsable de la création
  - La méthode fabrique peut choisir le type d'objet à créer
    - Exemple: `créerProduit(type:Chaîne)`
- Relations avec d'autres patrons
  - Fabrique abstraite
    - Utilise la fabrique dans son implémentation
  - Méthode patron
    - La méthode fabrique est une méthode patron

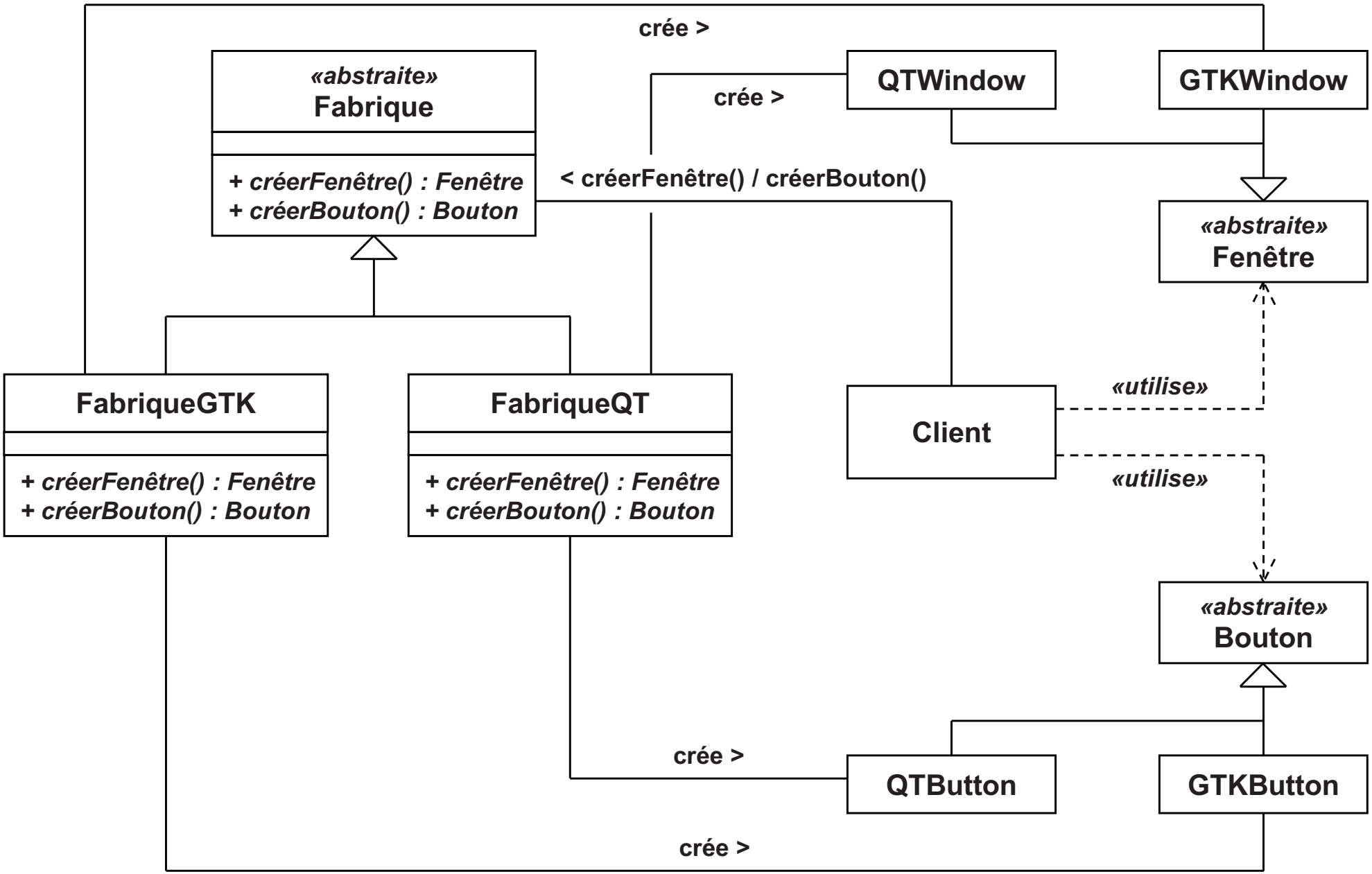
Fabrique abstraite / *Abstract Factory (1/4)*

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

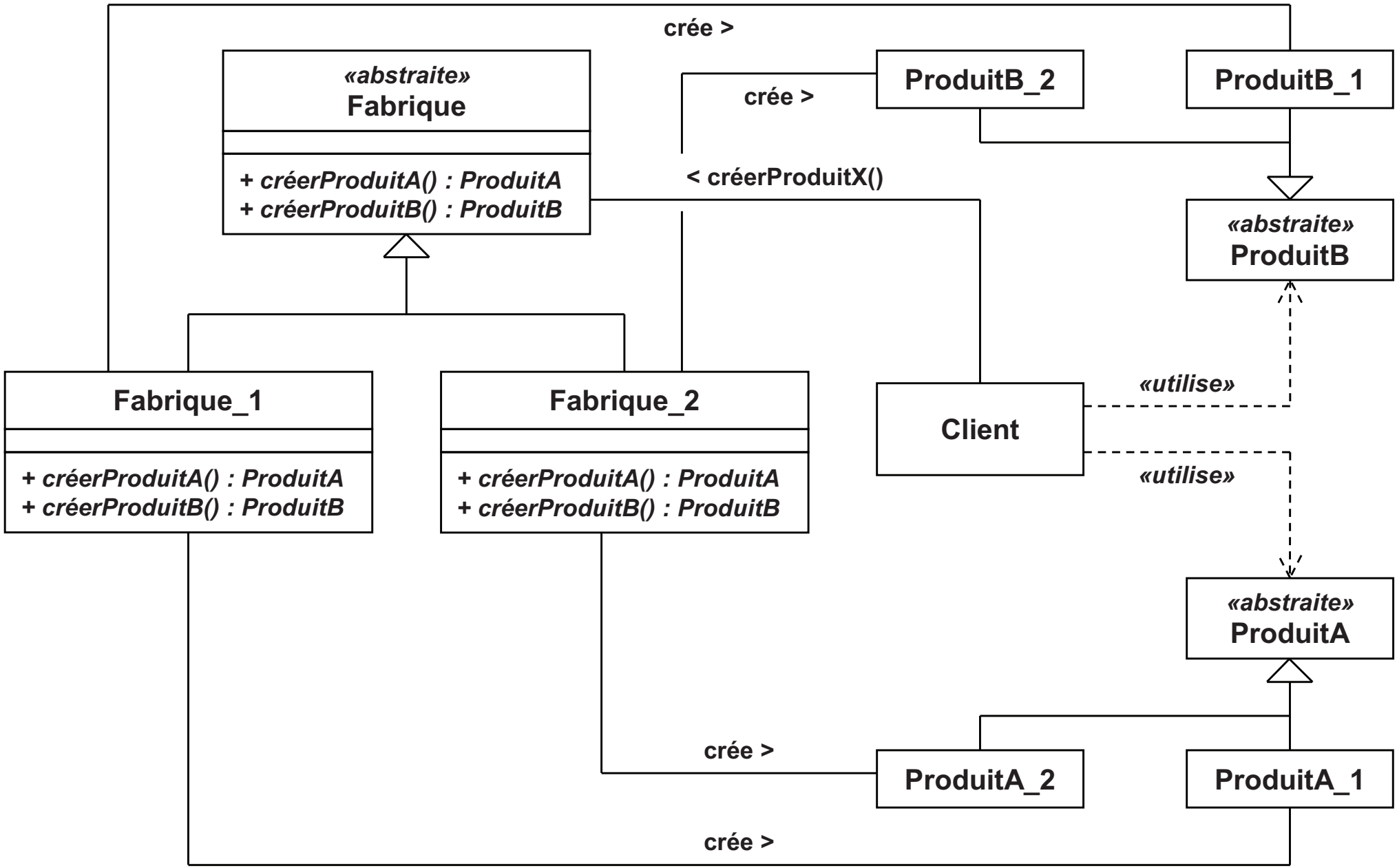
- Principe
  - Fournir une interface pour créer une famille d'objets $\Rightarrow$ la «fabrique»
  - Le client demande à la fabrique de lui fournir des instances
    - Le client ne connaît que les interfaces des objets
    - Seule la fabrique connaît les classes réelles des objets

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

Fabrique abstraite / *Abstract Factory (2/4)*



Fabrique abstraite / *Abstract Factory (3/4)*



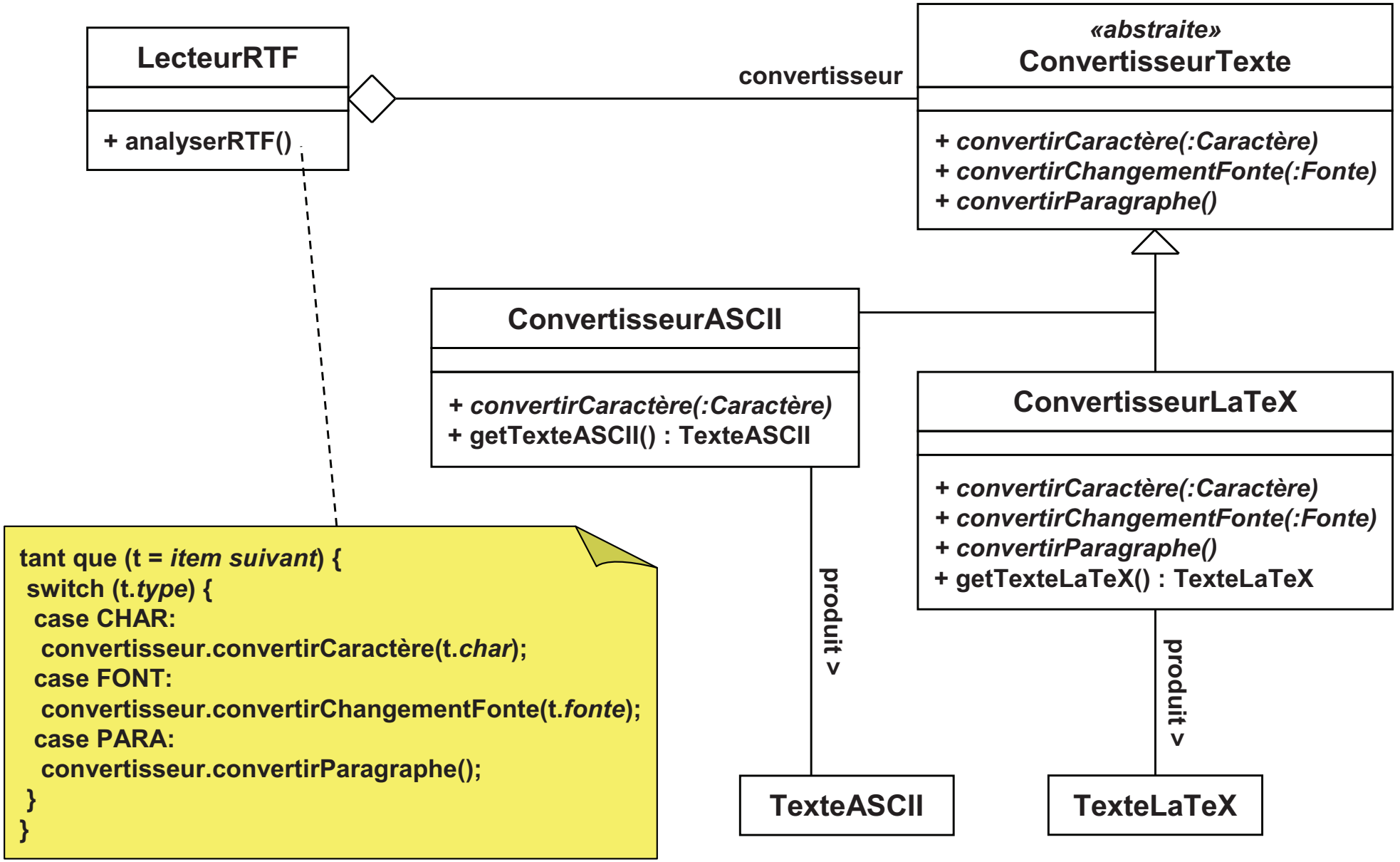
Fabrique abstraite / *Abstract Factory (4/4)*

- Appelé aussi «kit»
- Intérêts
  - Isolation des classes concrètes
    - Fabrique responsable de la création
  - Echange de famille de produits très facile
    - Remplacer la fabrique concrète par une autre
- Relations avec d'autres patrons
  - Singleton
    - Souvent, une seule instance de chaque fabrique
  - (Méthode) Fabrique
    - Une méthode fabrique par type de produit

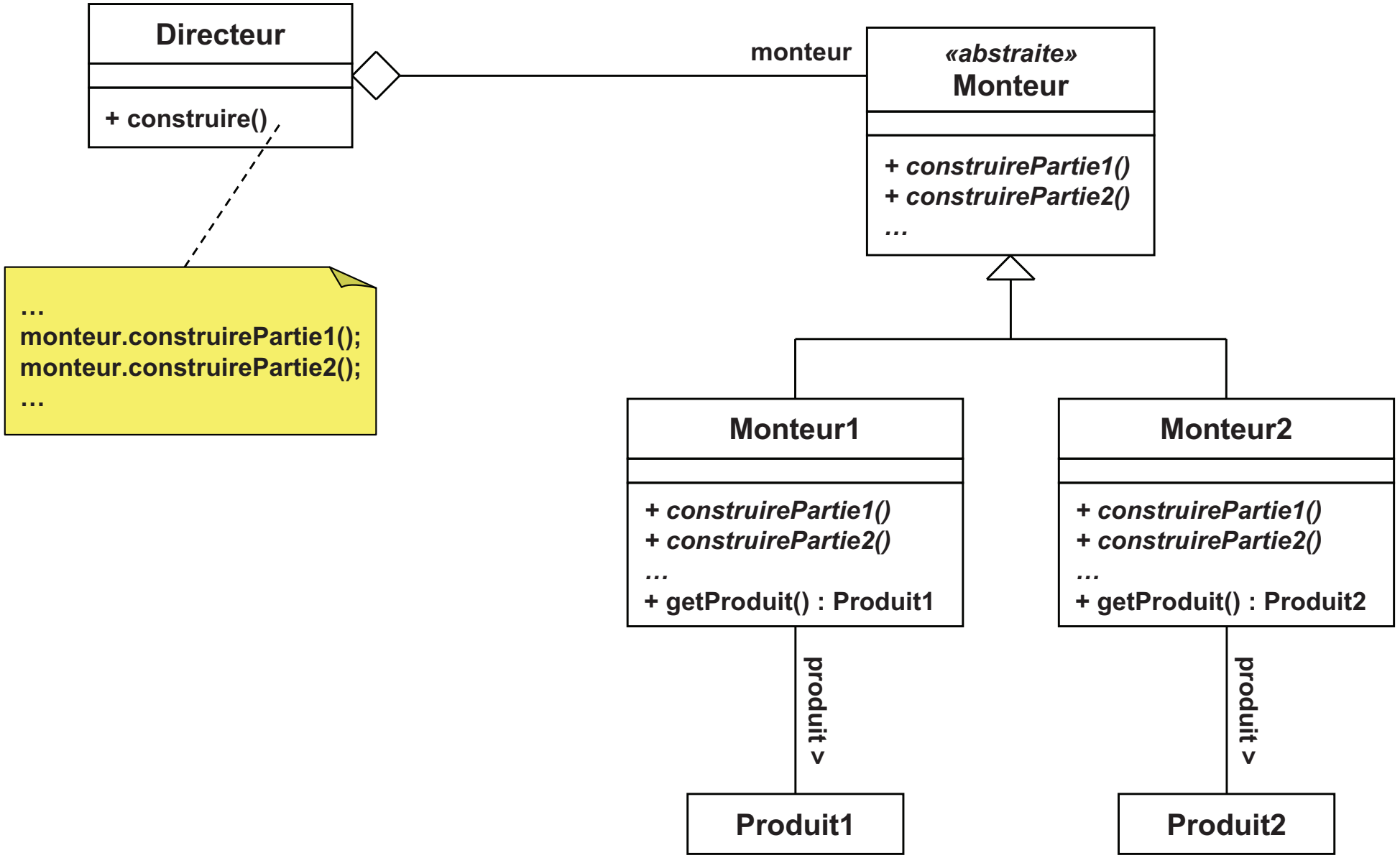
Monteur / *Builder (1/4)*

- Objectif
  - Séparer la construction d'un objet complexe de sa représentation
  - Même processus de construction mais représentations différentes
- Principe
  - Un «directeur» construit une structure complexe
  - Il délègue la création des parties à un «monteur»
  - Le directeur ne connaît que l'interface des parties
  - Seul le monteur connaît la classe réelle des parties
- Motivation
  - Conversion d'un format de fichier vers un format cible
  - Le format cible peut changer

Monteur / *Builder (2/4)*



Monteur / *Builder (3/4)*



Monteur / *Builder (4/4)*

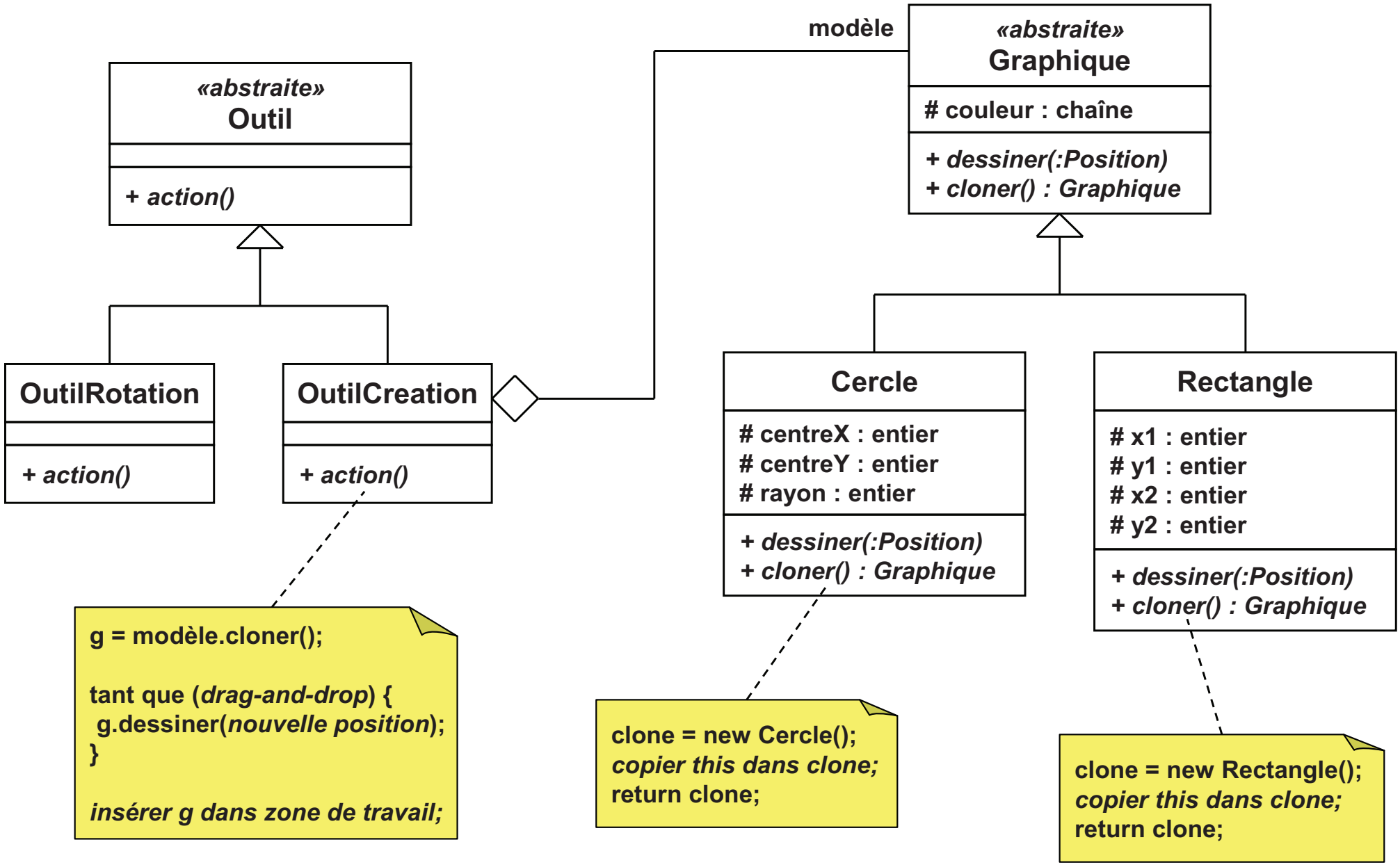
- Intérêts
  - Isolation de la construction et de la représentation
    - Le processus de création des parties est masqué
    - Le processus d'assemblage des parties est masqué
  - Echange de représentation d'une structure complexe facile
    - Remplacer le monteur concret par un autre
  - Malgré l'abstraction, contrôle précis du processus de création
    - Produit construit pas à pas
    - Sous la supervision du directeur
    - Accès au produit une fois le processus terminé

- Relations avec d'autres patrons
  - Fabrique abstraite
    - Fabrique abstraite = construction d'une famille de produits
    - Monteur = construction pas à pas d'un produit complexe

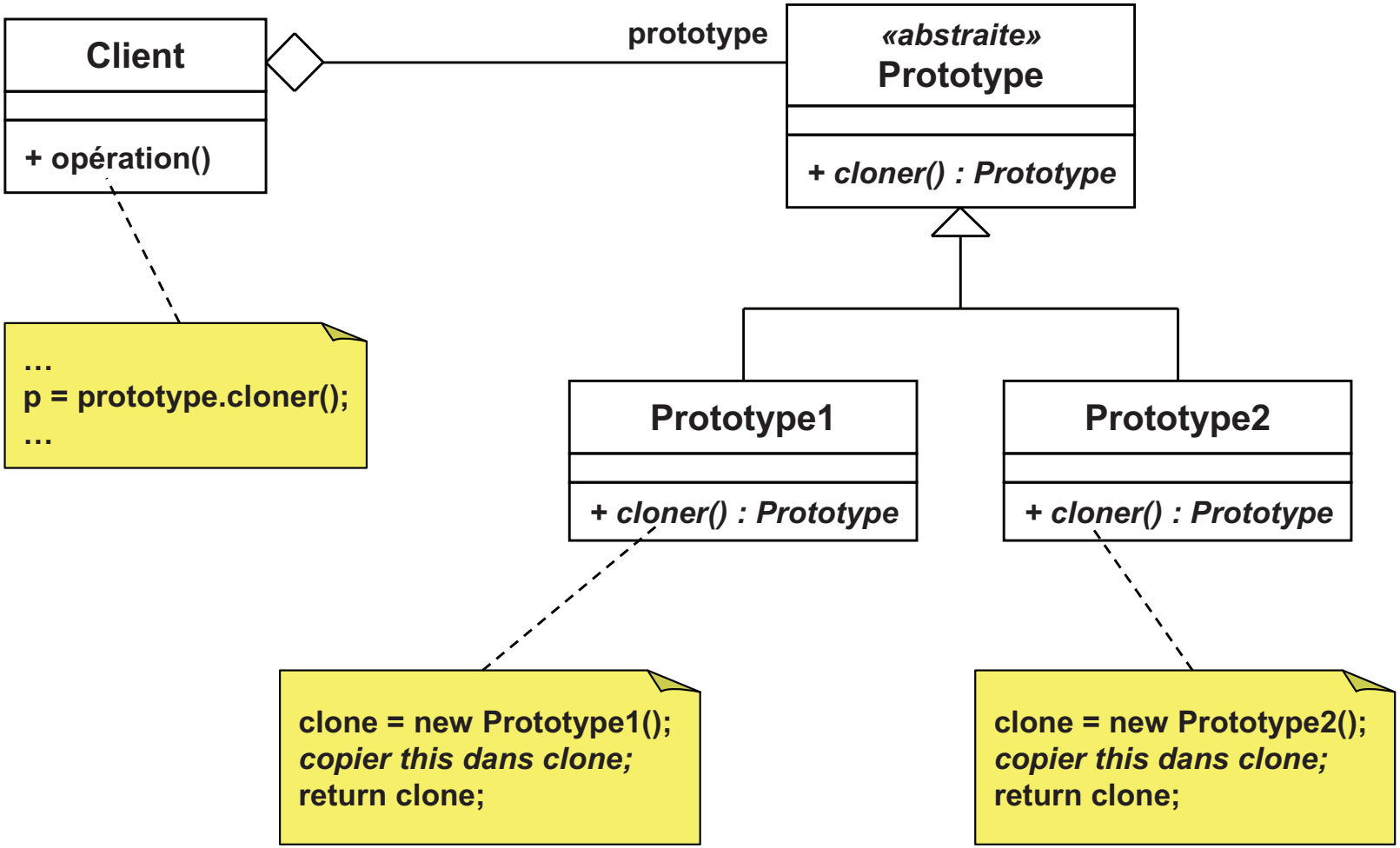
Prototype / *Prototype (1/4)*

- 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 (2/4)*



Prototype / *Prototype (3/4)*



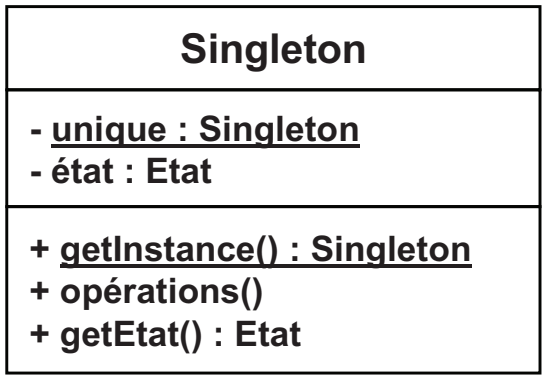
Prototype / *Prototype (4/4)*

- Intérêts
  - Abstraction de la construction
    - 2 instances différentes $\Rightarrow$ 2 initialisations différentes
  - Utile lorsque la phase d'initialisation est coûteuse
    - Plus rapide de recopier une instance
- Relations avec d'autres patrons
  - Fabrique abstraite
    - Peut utiliser des prototypes pour créer les objets
- En Java
  - Tous les objets appartiennent à la classe «`Object`»
  - Cette classe fournit une méthode «`clone`»

Singleton / *Singleton (1/3)*

- Objectif
  - Garantir une seule instance pour une classe
  - Fournir un point d'accès global à cette instance
- Principe
  - Masquer les constructeurs de cette classe
    - Impossibilité de créer un objet en dehors de la classe
  - 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 (2/3)*



- Exemple C++

```
class Singleton {
 private: static Singleton unique;

 // Attributs du singleton

 private:
  Singleton(...) { ... }
  Singleton(const Singleton &);
  Singleton & operator=(const Singleton &);

 public: static Singleton & getInstance()
 { return unique; }

 // Méthodes du singleton
};

Singleton Singleton::unique(...);
```

- Création et copie d'un objet interdites
  - Opérateurs privés
- Seule possibilité: utiliser l'instance unique
  - Autorisé: `Singleton::getInstance()`

Singleton / *Singleton (3/3)*

- Intérêts
  - Contrôler la création des objets d'une classe
    - Permet notamment d'imposer le nombre d'instances
  - Contrôler l'accès aux instances
    - Exemple: accès protégé pour le *multithreading*
  - Fournit un espace de nommage
    - Alternative aux variables globales
    - Alternative aux fonctions
  - Extension possible par héritage

- Relations avec d'autres patrons
  - Fabrique abstraite / Monteur / Prototype
    - Ils peuvent utiliser le singleton dans leur implémentation

ISIMA

Patrons de structure

(PARTIE VII - Patrons de conception)

Bruno Bachelet

Christophe Duhamel

Luc Touraille

Patrons de structure *(1/3)*

- 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

Patrons de structure *(2/3)*

- Adaptateur / *Adapter*
  - Adapter l'interface d'une classe à ses besoins
- Pont / *Bridge*
  - Découpler l'interface d'un composant de son l'implémentation
- Composite / *Composite*
  - Composer des objets sous forme arborescente

Patrons de structure *(3/3)*

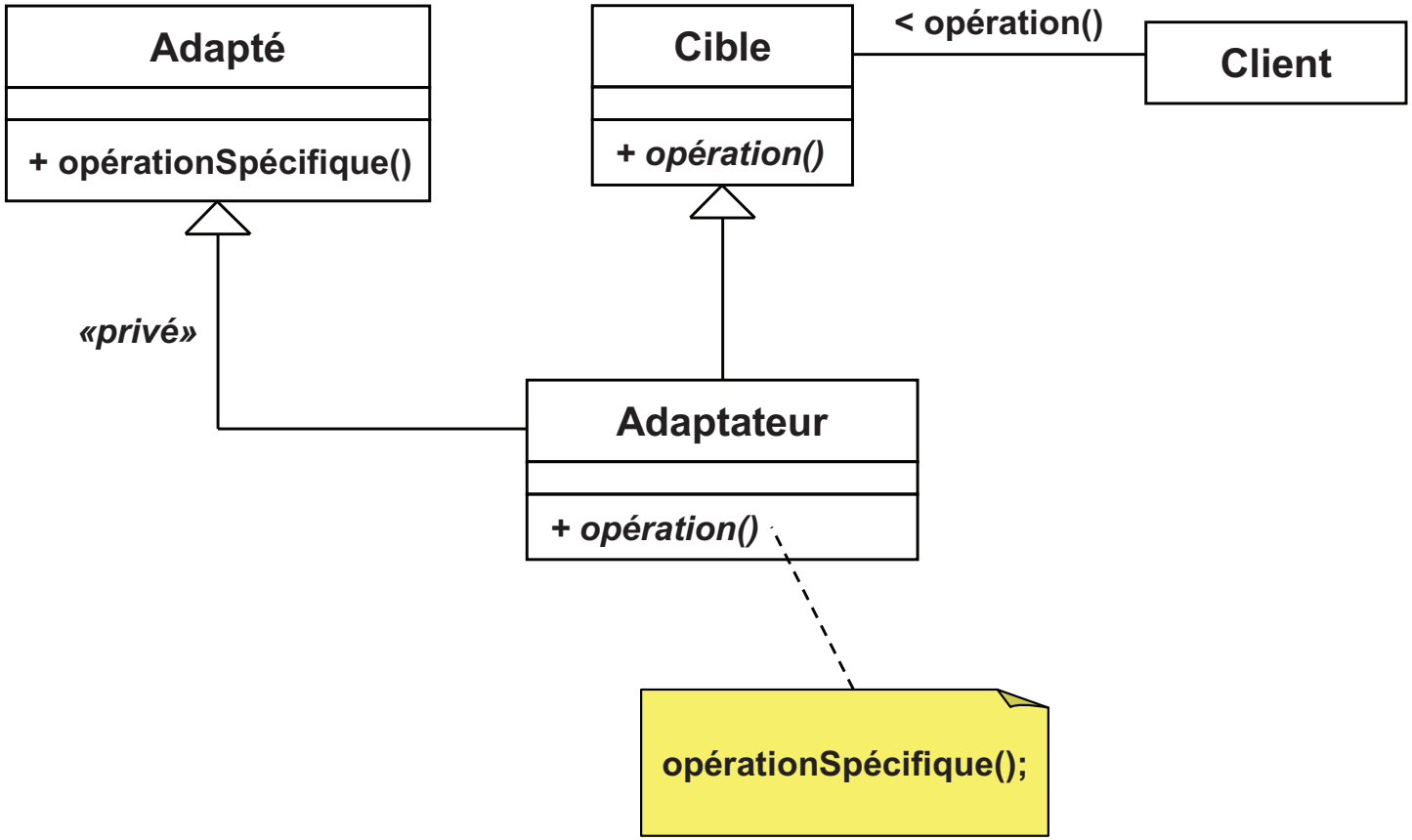
- Décorateur / *Decorator*
  - Ajouter dynamiquement des fonctionnalités à un objet
- Façade / *Facade*
  - Découpler un sous-système de ses clients
- Poids-mouche / *Flyweight*
  - Partager des instances pour éviter un nombre trop important
- Proxy / *Proxy*
  - Fournir un substitut pour accéder à un objet

Adaptateur / *Adapter (1/5)*

- Objectif
  - Adapter l'interface d'une classe à ses besoins
  - Permettre le dialogue entre classes incompatibles
- Principe
  - Deux approches
  - Classe «adaptateur»
    - Héritage de la nouvelle interface
    - Héritage de l'implémentation de l'ancienne interface
  - Objet «adaptateur»
    - Héritage de la nouvelle interface
    - Agrégation d'un objet de l'ancienne interface, et délégation
- Motivation
  - Utiliser une fonctionnalité d'une bibliothèque tierce
  - Mais l'interface n'est pas adaptée

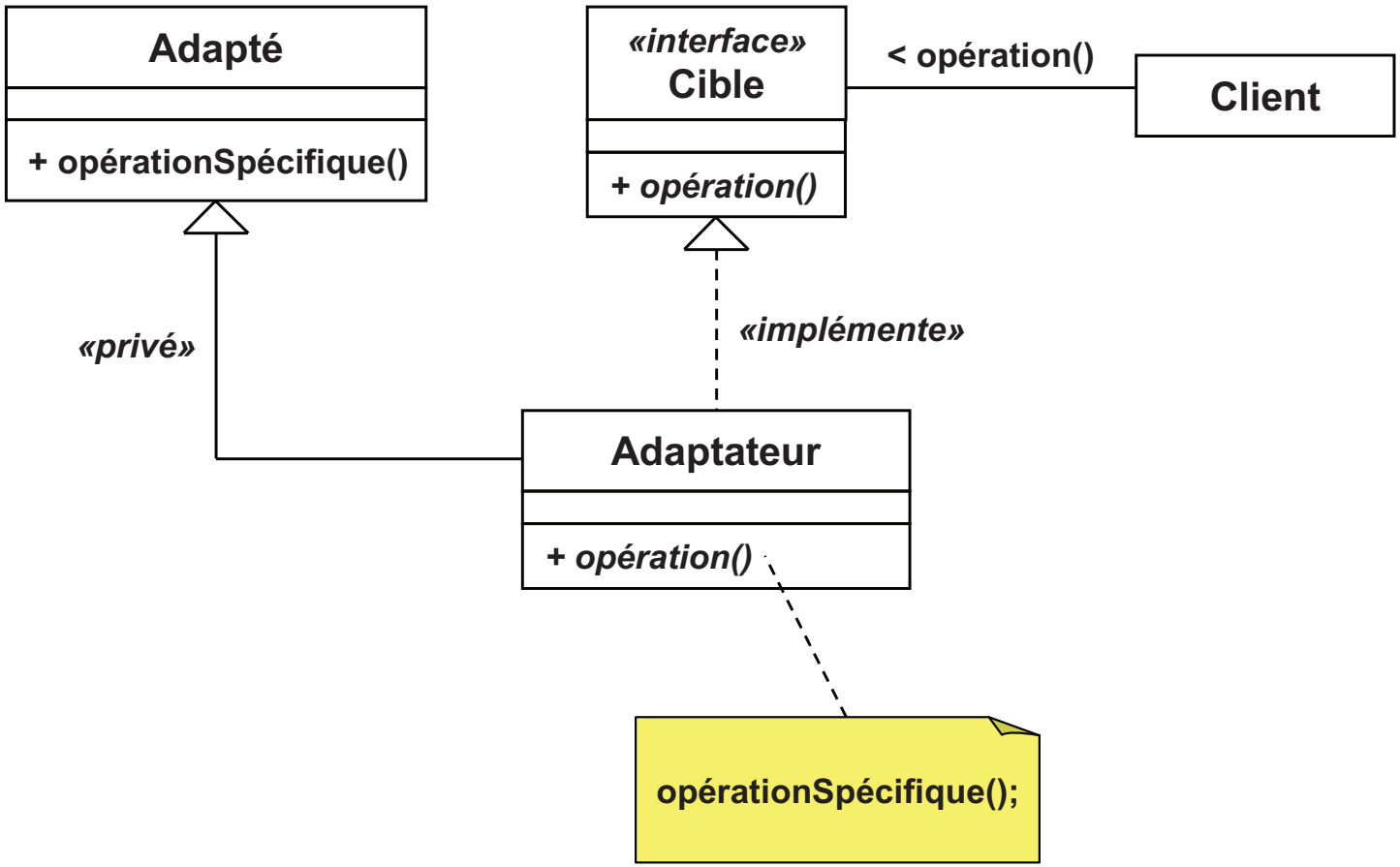
Adaptateur / *Adapter (2/5)*

- Classe adaptateur, version 1
  - Héritage de l'implémentation de l'adapté = héritage privé
  - Inconvénient: héritage multiple



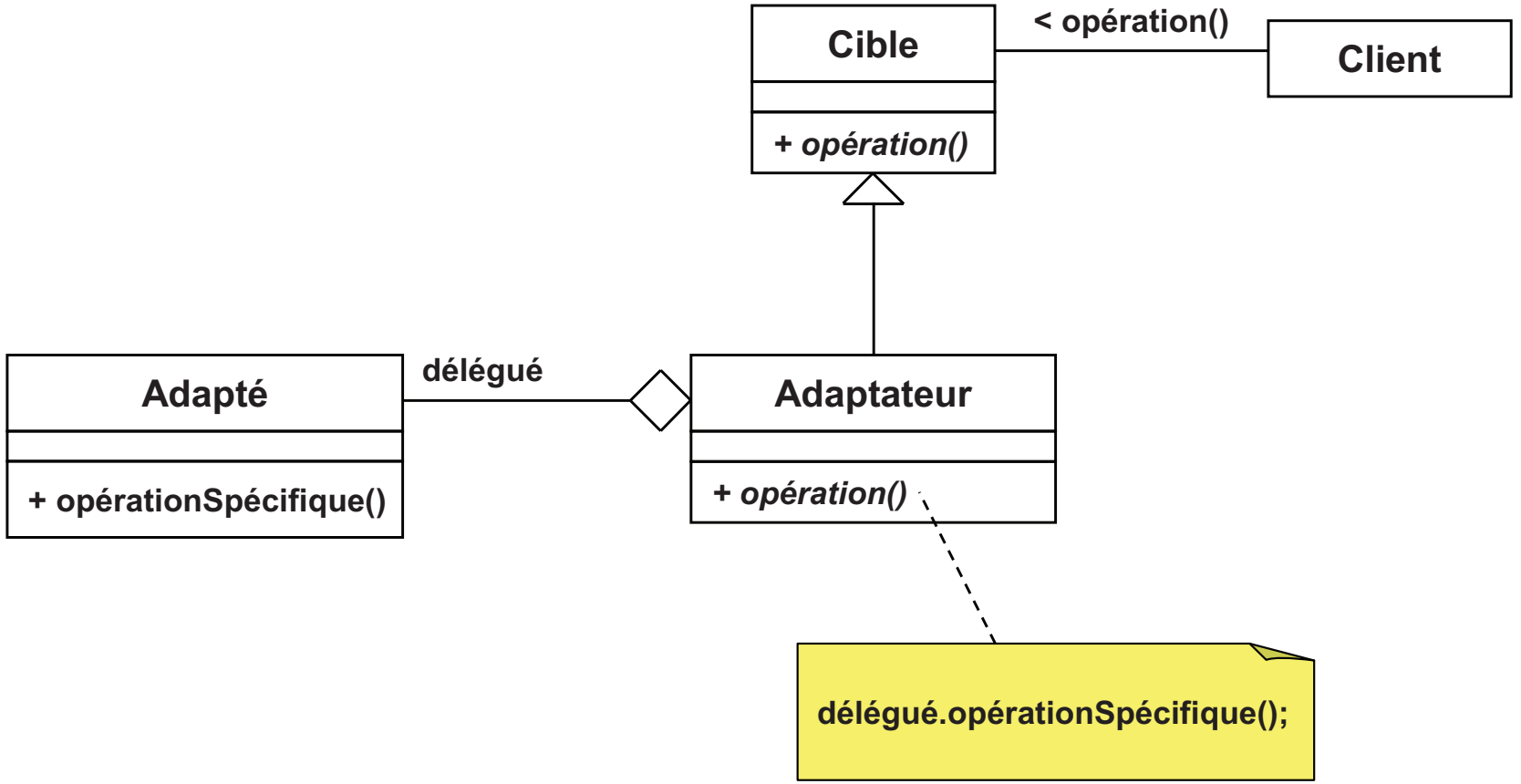
Adaptateur / *Adapter (3/5)*

- Classe adaptateur, version 2
  - La cible est une interface, et non une classe
  - Plus de problème d'héritage multiple



Adaptateur / *Adapter (4/5)*

- Objet adaptateur, délégation
  - Un objet de la classe adaptée est agrégé dans l'adaptateur
  - Plus d'héritage privé, ni multiple



Adaptateur / *Adapter (5/5)*

- Appelé aussi *«wrapper»*

- Intérêts
  - Classe adaptateur
    - Permet une surcharge simple des fonctionnalités de l'adapté
    - L'adaptation ne crée qu'un seul objet
  - Objet adaptateur
    - Plus d'héritage multiple
    - L'adaptation se fait sur une classe et ses sous-classes

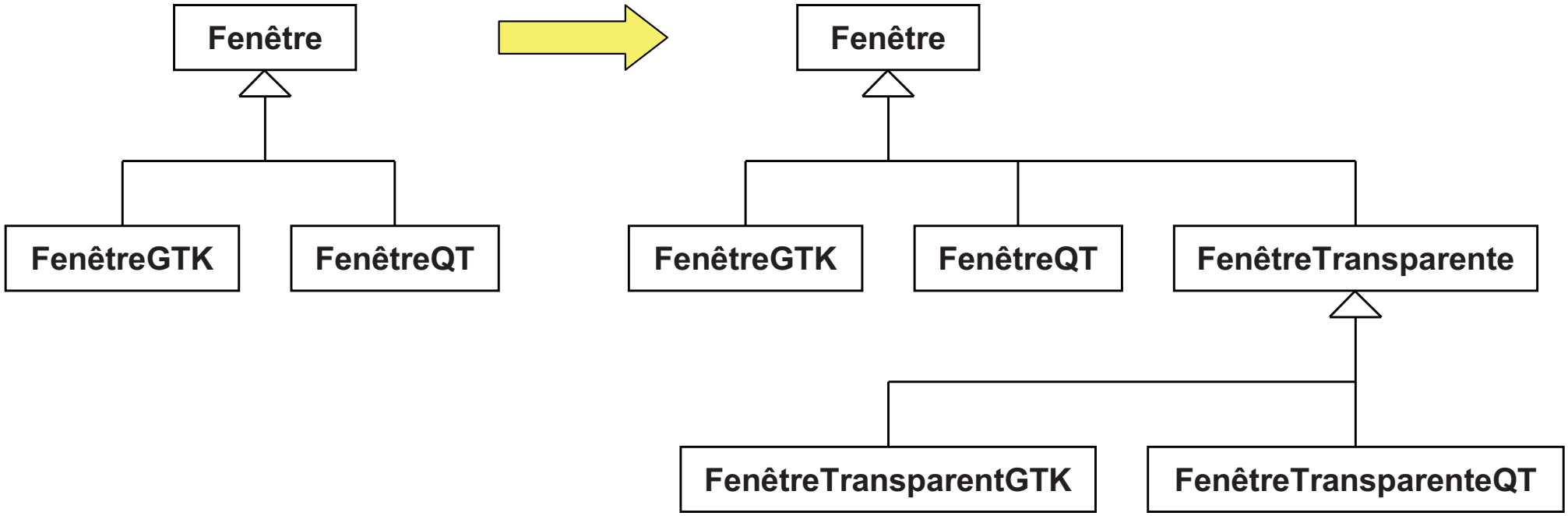
- Patrons similaires
  - Pont: séparation interface / implémentation
  - Décorateur: ajout dynamique de fonctionnalités à un objet
  - Proxy: accès à un objet à travers un intermédiaire

Pont / *Bridge (1/5)*

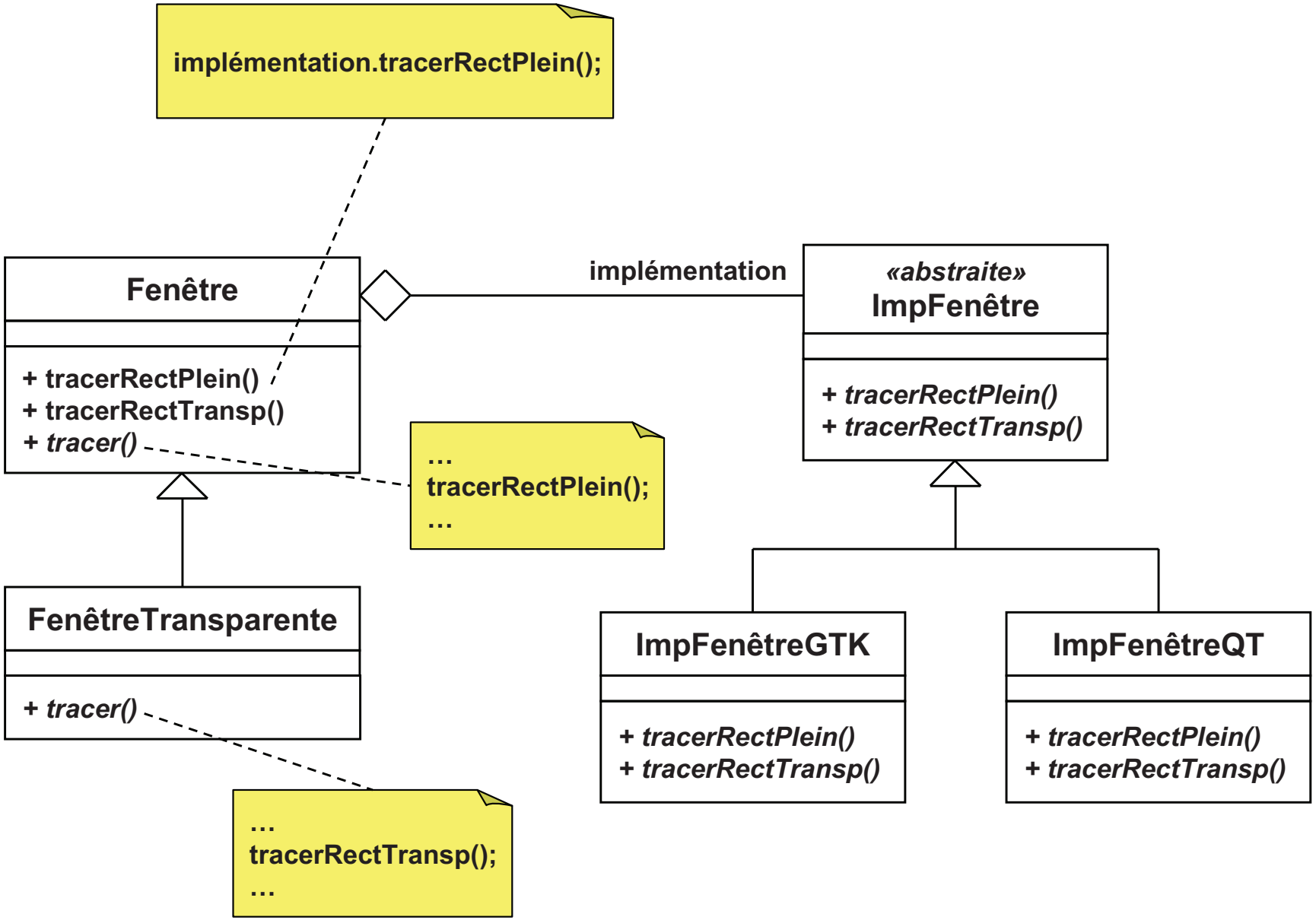
- Objectif
  - Découpler l'interface d'un composant de son implémentation
- Principe
  - Séparation de la classe en deux classes
    - L'une représente l'interface du composant
    - L'autre l'implémentation du composant
  - L'interface agrège une implémentation à laquelle elle délègue les appels aux méthodes
- Motivation
  - Plusieurs logiques d'héritage peuvent s'entremêler

Pont / *Bridge (2/5)*

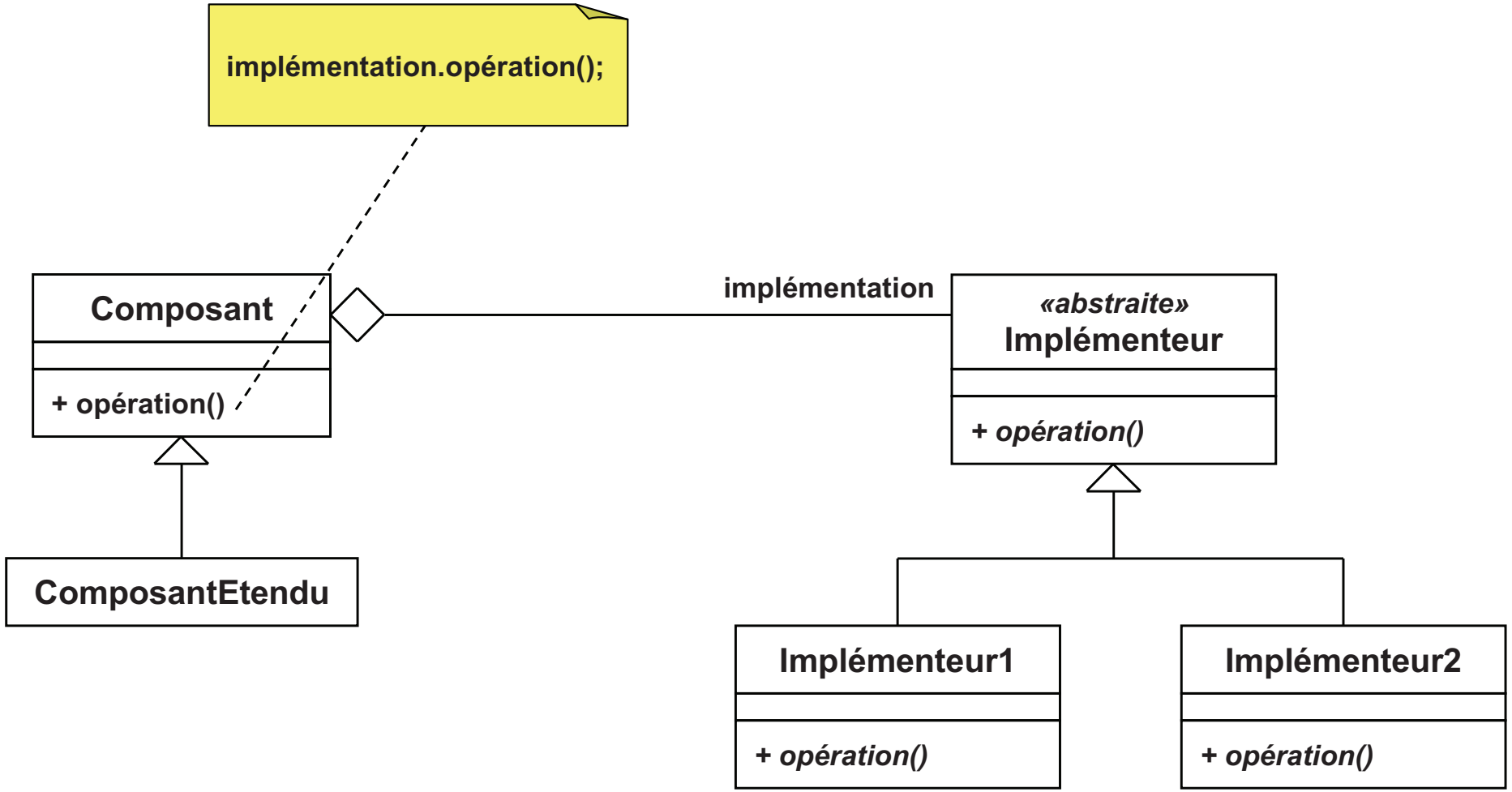
- Exemple d'entremêlement
  - Classe «`Fenêtre`» spécialisée pour 2 bibliothèques graphiques
  - Sous-classe «`FenêtreTransparente`» ⇒ 2 spécialisations



Pont / *Bridge (3/5)*



Pont / *Bridge (4/5)*



Pont / *Bridge (5/5)*

- Appelé aussi *«handle»* ou *«body»*
- Intérêts
  - Découplage interface / implémentation
    - Pas de lien permanent entre les deux
  - Augmentation de l'extensibilité
    - Deux hiérarchies séparées: composant et implémenteur
  - Masquer totalement l'implémentation
    - Plus d'attributs déclarés dans le composant
- Relations avec d'autres patrons
  - Fabrique abstraite
    - Peut être utilisée pour construire un pont
  - Adaptateur
    - Utilisation *a posteriori* (contrairement au pont)

Composite / *Composite (1/6)*

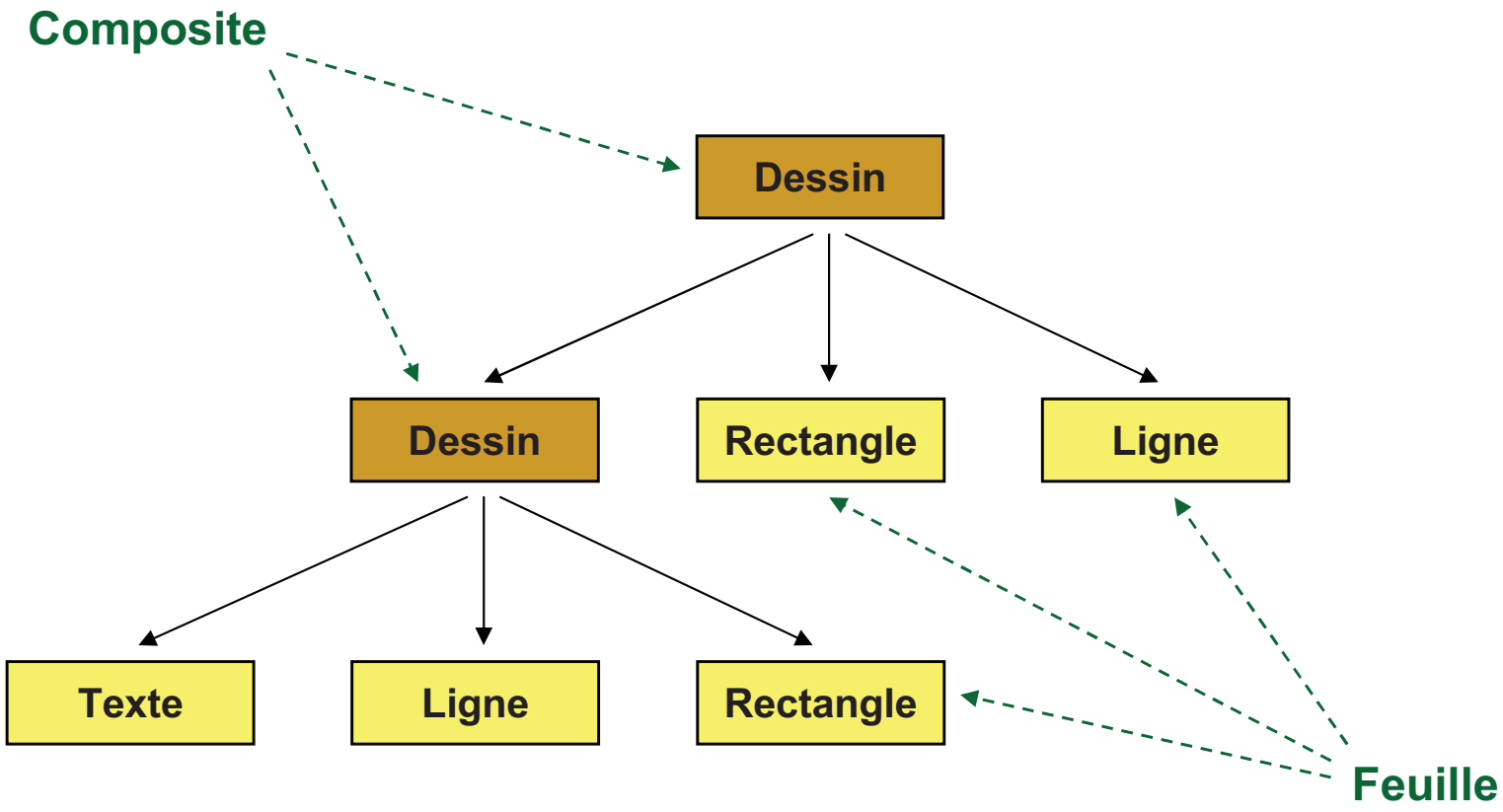
- Objectif
  - Composer des objets sous forme arborescente
  - Objet individuel ou composition traités de la même manière

- Principe
  - Un objet est composé d'autres objets
  - Ces objets peuvent également être des agrégats d'objets

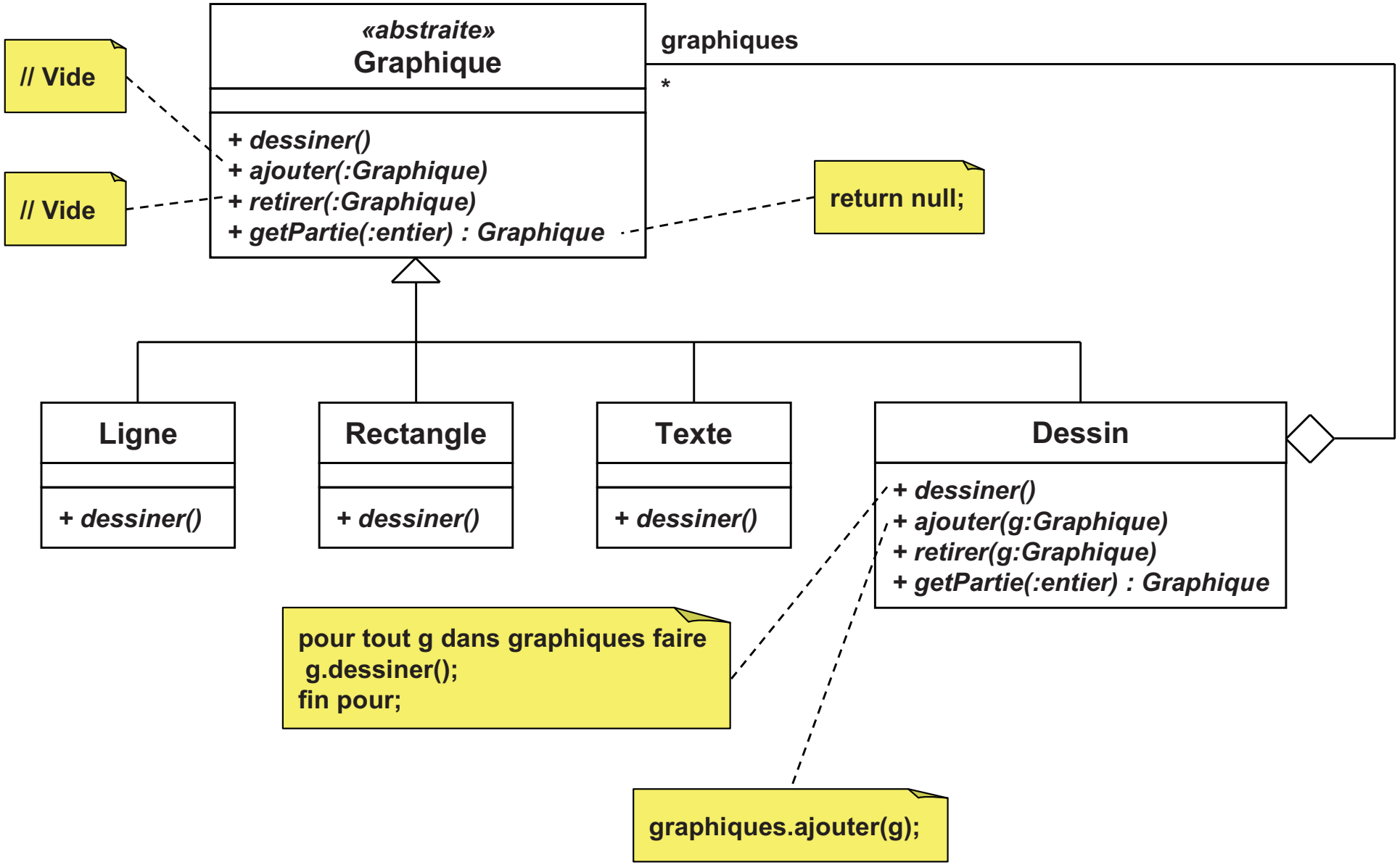
  $\Rightarrow$ Récursivité dans la composition

- Motivation
  - Schéma/dessin composé d'objets graphiques
  - Hiérarchie d'héritage des objets graphiques
  - Un objet graphique peut être un groupement d'objets graphiques

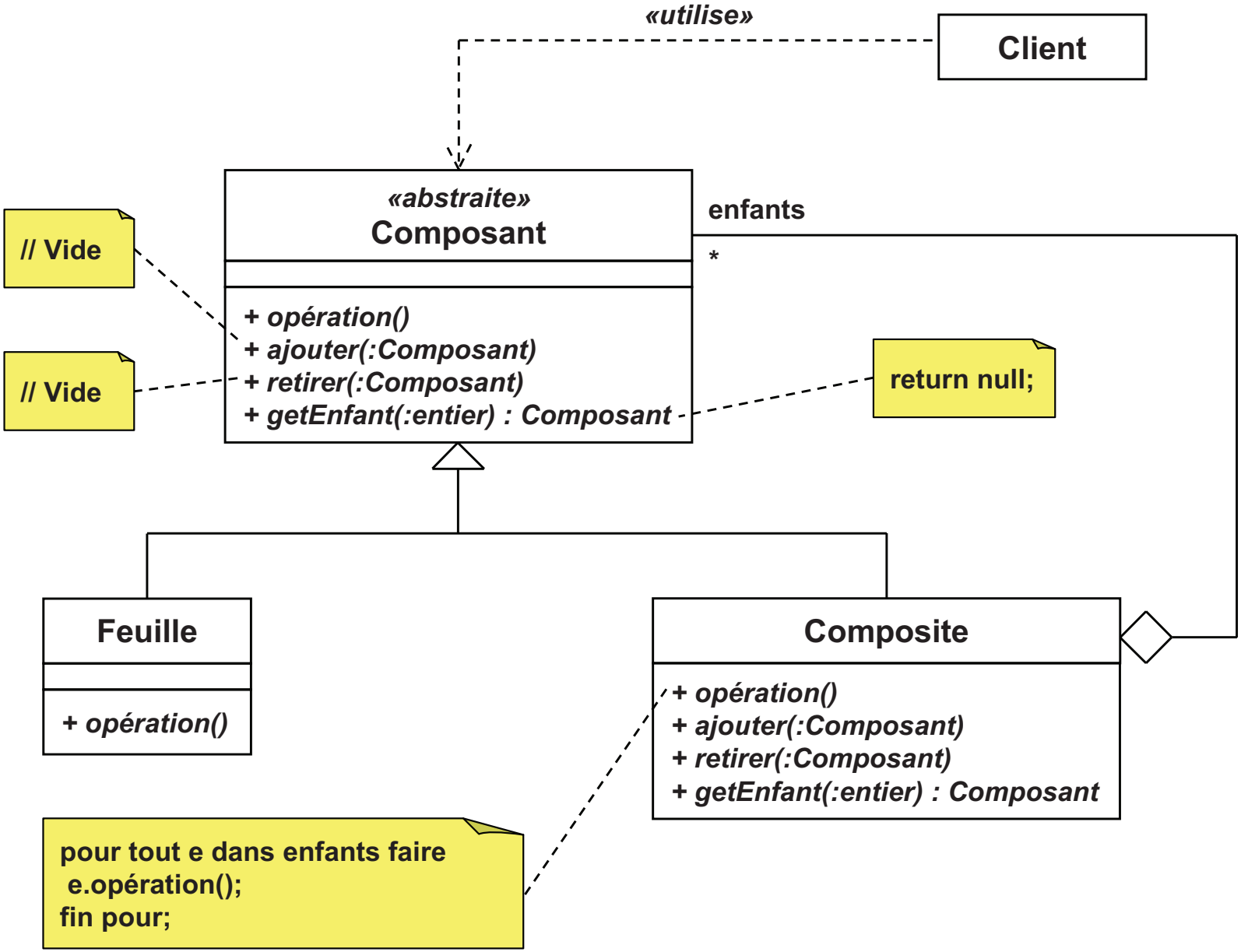
Composite / *Composite (2/6)*



Composite / *Composite (3/6)*



Composite / Composite (4/6)



Composite / *Composite (5/6)*

- Intérêts
  - 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 saura le manipuler
    - Sans modification, il pourra être ajouté dans un composite
- Implémentation
  - Référence au parent ?
    - Pour faciliter certaines manipulations, l'enfant peut connaître son parent
    - Mais, plus délicat si l'enfant fait partie de plusieurs composites

Composite / *Composite (6/6)*

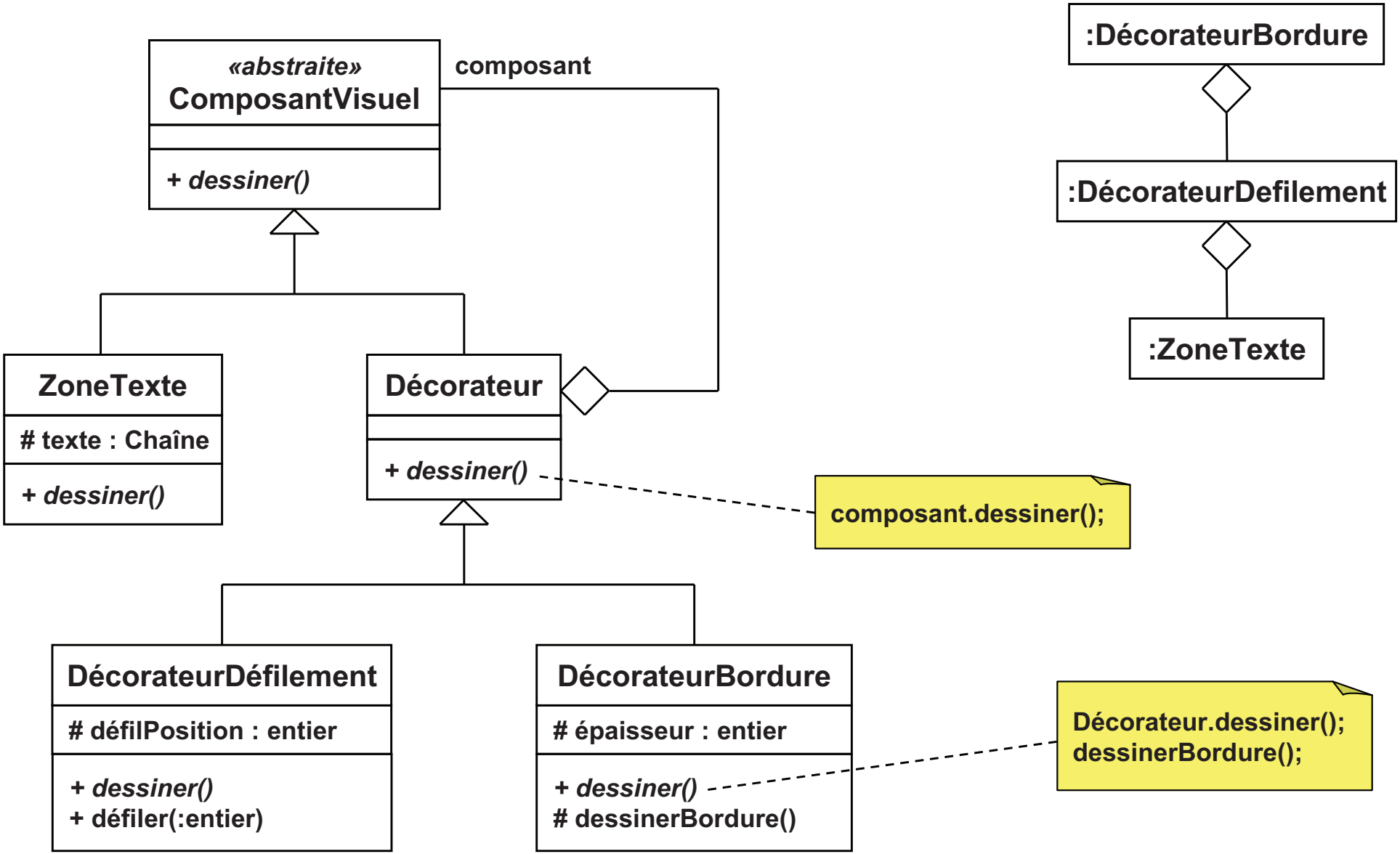
- Implémentation
  - L'interface du composant peut avoir tendance à «gonfler»
    - Tendance à tout faire passer par la classe «`Composant`»
    - Pour gérer les méthodes spécifiques
      - Solution 1: Méthodes abstraites dans la classe «`Composant`»
      - Solution 2: Reconnaissance dynamique de type et conversion
  - Suppression d'un composite
    - Enfants supprimés, détachés ou rattachés au parent ?

- Relations avec d'autres patrons
  - Décorateur
    - Implémentation sous forme de composite
  - Itérateur
    - Utilisé pour parcourir les composants
  - Visiteur
    - Utilisé pour appliquer une opération à tous les composants

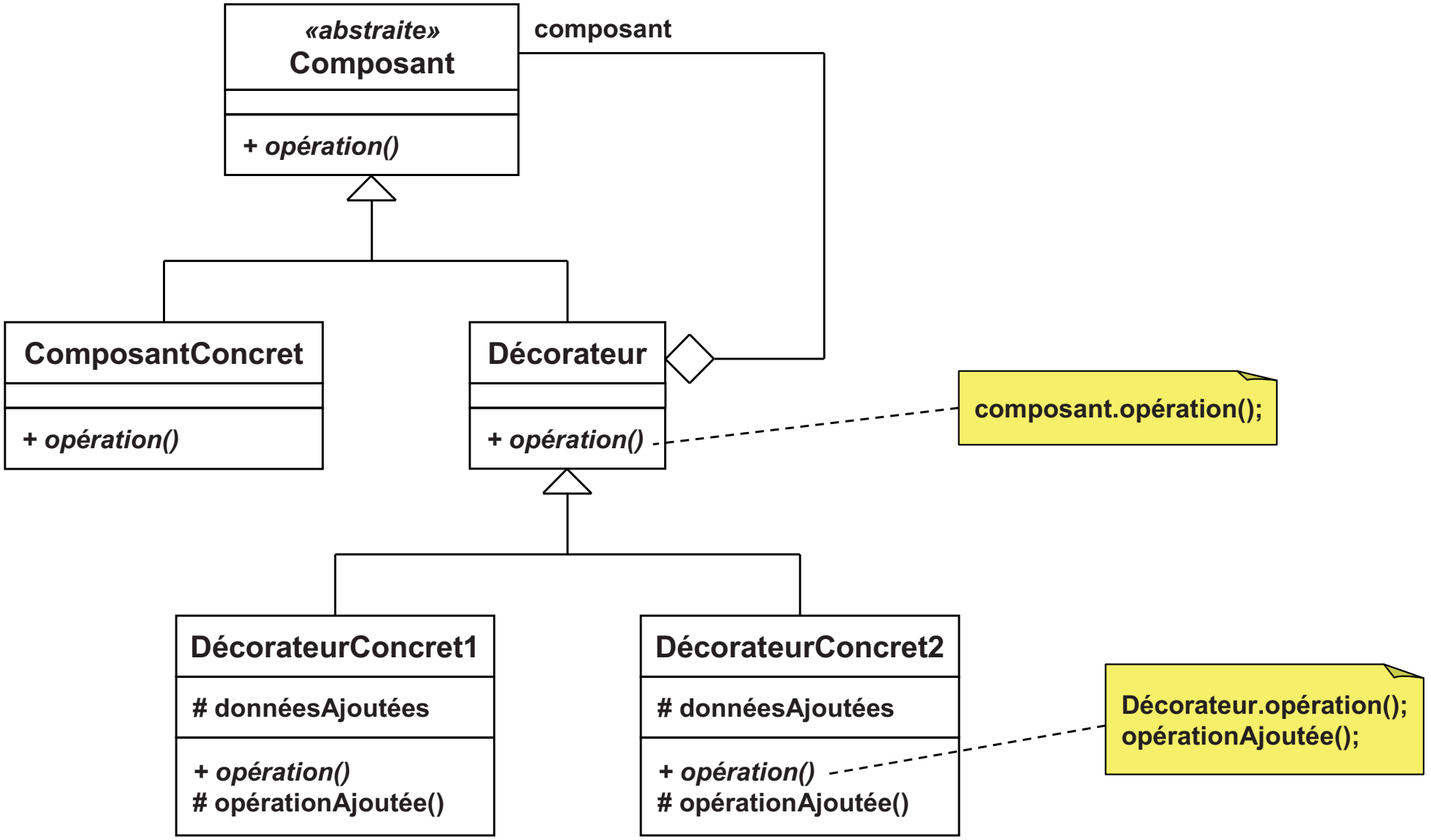
Décorateur / *Decorator (1/5)*

- 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 (2/5)*



Décorateur / *Decorator (3/5)*



Décorateur / *Decorator (4/5)*

- Appelé aussi *«wrapper»*

- Intérêts
  - 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 ⇒ Extension des 3 classes
  - Mais le décorateur ajoute un objet à chaque décoration

Décorateur / *Decorator (5/5)*

- Relations avec d'autres patrons
  - Adaptateur
    - Similaires, mais l'adaptateur modifie l'interface
  - Composite
    - Utilisation «dégénérée» (1 seul enfant) du patron composite
  - Stratégie
    - Similaires: ils changent les fonctionnalités
    - Décorateur: ajoute des fonctionnalités par agrégation
    - Stratégie: change l'implémentation de fonctionnalités par héritage

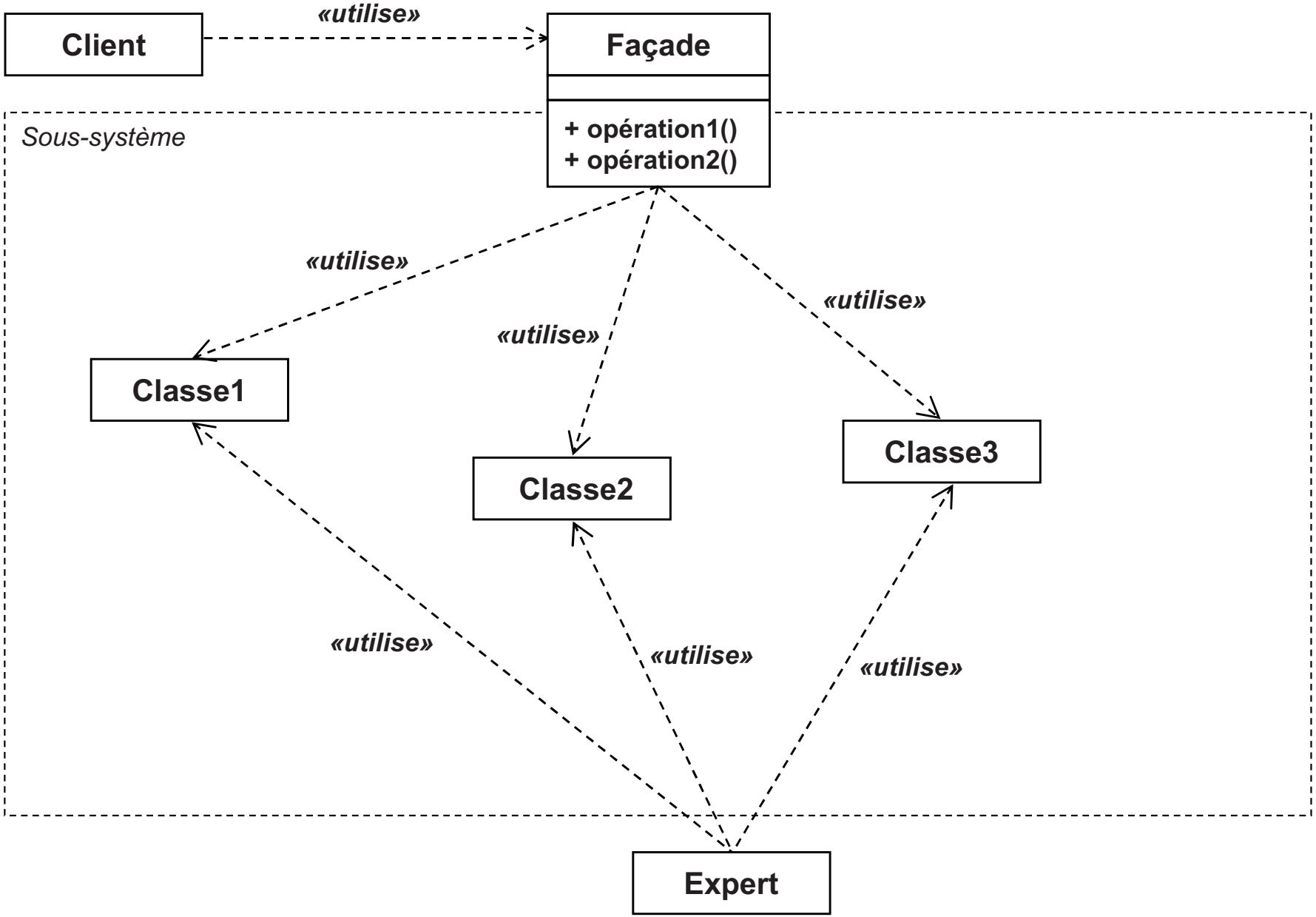
Façade / *Facade (1/3)*

- Objectif
  - Découpler un sous-système de ses clients
  - Fournir une interface unifiée pour l'ensemble des composants
  - Fournir une interface de plus haut niveau pour faciliter son utilisation

- Principe
  - Une interface «simplifiée» est proposée: la «façade»
    - Elle connaît les détails du sous-système
  - Le client envoie ses requêtes à la façade
    - La façade délègue les requêtes aux composants du sous-système

- Motivation
  - Bibliothèque complexe, avec beaucoup d'interfaces
    - Complexité nécessaire pour des clients experts
    - Mais inutile pour une majorité de clients
  - Objectif
    - Garder la puissance de la bibliothèque
    - Tout en fournissant une interface simplifiée

Façade / *Facade (2/3)*



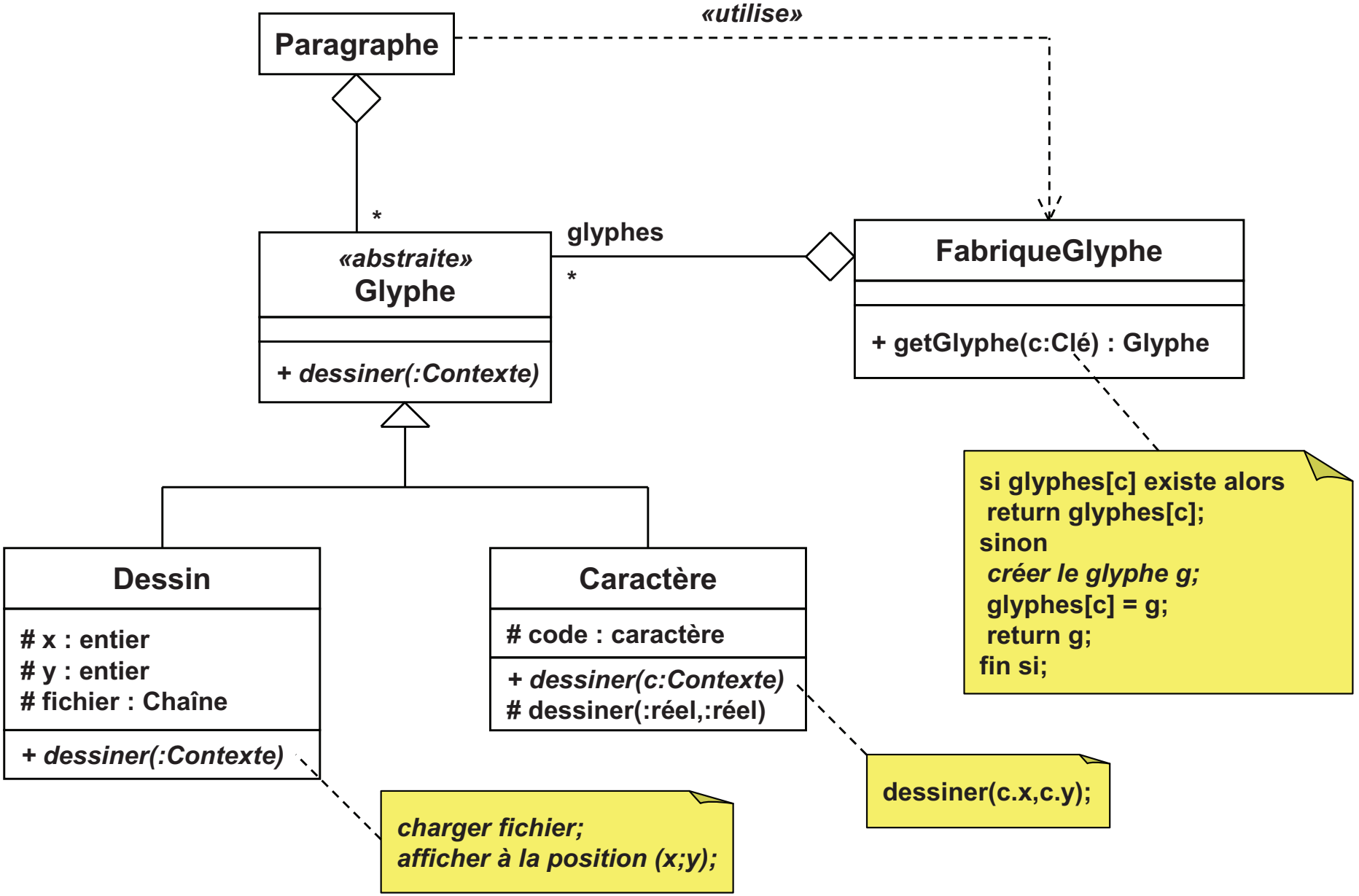
Façade / *Facade (3/3)*

- Intérêts
  - Découple le sous-système de ses clients
    - Un seul point d'entrée
  - Laisse la liberté au client d'utiliser le jeu d'interfaces bas niveau
    - Nécessaire pour des utilisateurs experts
    - Permet d'utiliser toute la puissance du sous-système
- Relations avec d'autres patrons
  - Fabrique abstraite
    - Permet d'assurer une construction cohérente d'objets du sous-système
  - Médiateur
    - Similaires, intermédiaires qui masquent des composants
    - Mais le but du médiateur est de centralisé / abstraire des communications
  - Singleton
    - Souvent, un seul objet façade par programme

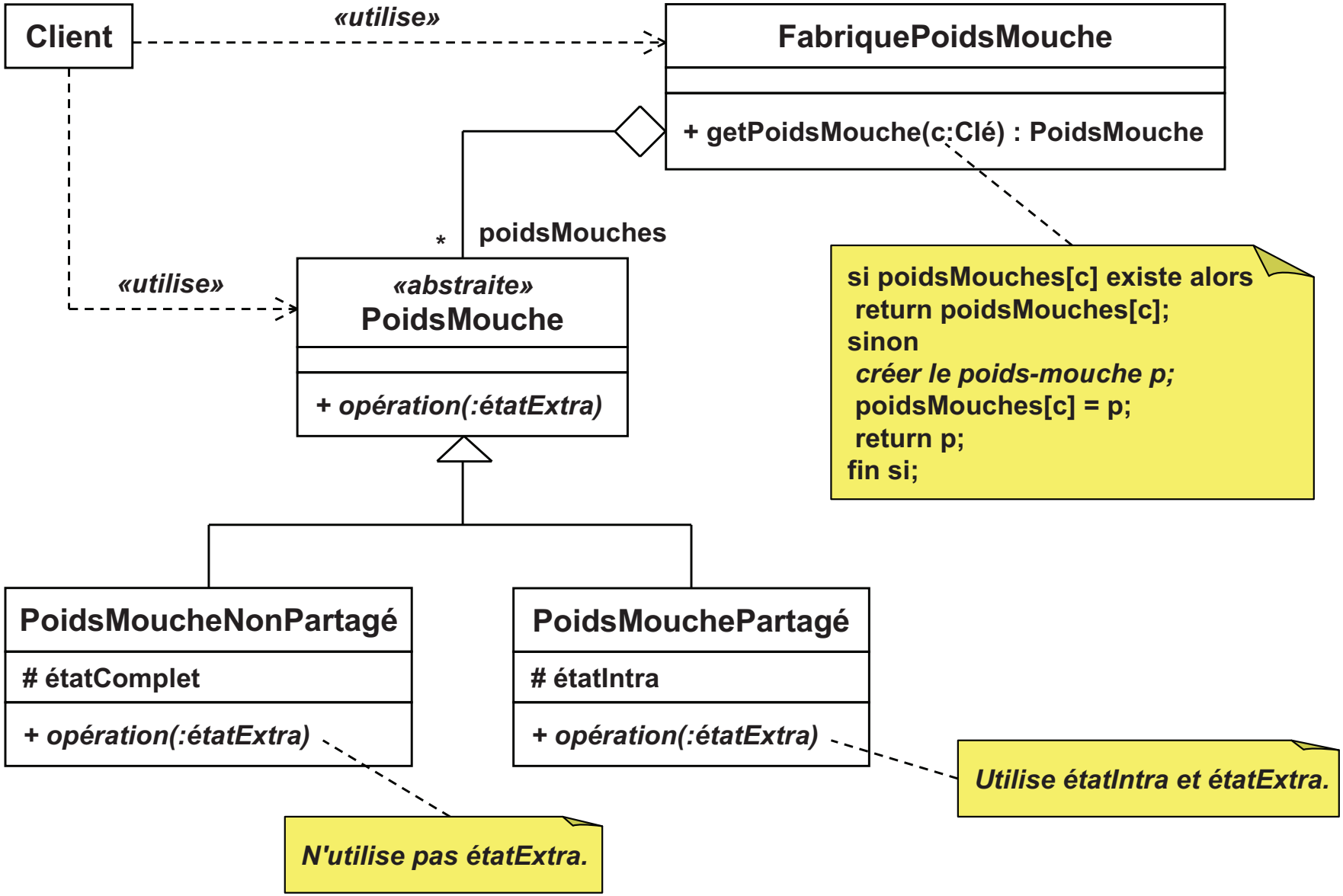
Poids-mouche / *Flyweight (1/4)*

- Objectif
  - Partager des instances pour éviter un nombre trop important
- Principe
  - Séparation de l'état d'un objet en deux parties
    - Etat intrinsèque: indépendant du contexte
    - Etat extrinsèque: dépendant du contexte
  - Etat intrinsèque stocké dans l'objet
  - Etat extrinsèque fourni en paramètre par le client
- Motivation
  - Représentation des caractères dans un traitement de texte
  - Modéliser chaque caractère comme un objet
  - Mais éviter d'avoir effectivement un objet par caractère

Poids-mouche / *Flyweight (2/4)*



Poids-mouche / *Flyweight (3/4)*



Poids-mouche / *Flyweight (4/4)*

- Intérêts
  - Evite la duplication inutile de données
    - Etat intrinsèque jamais dupliqué
    - Etat extrinsèque calculé ou mémorisé
  - Mais génère un surcoût à l'exécution
    - Lié à la transmission de l'état extrinsèque
- Relations avec d'autres patrons
  - Composite
    - Peuvent être combinés
    - Pour obtenir une arborescence avec feuilles partagées

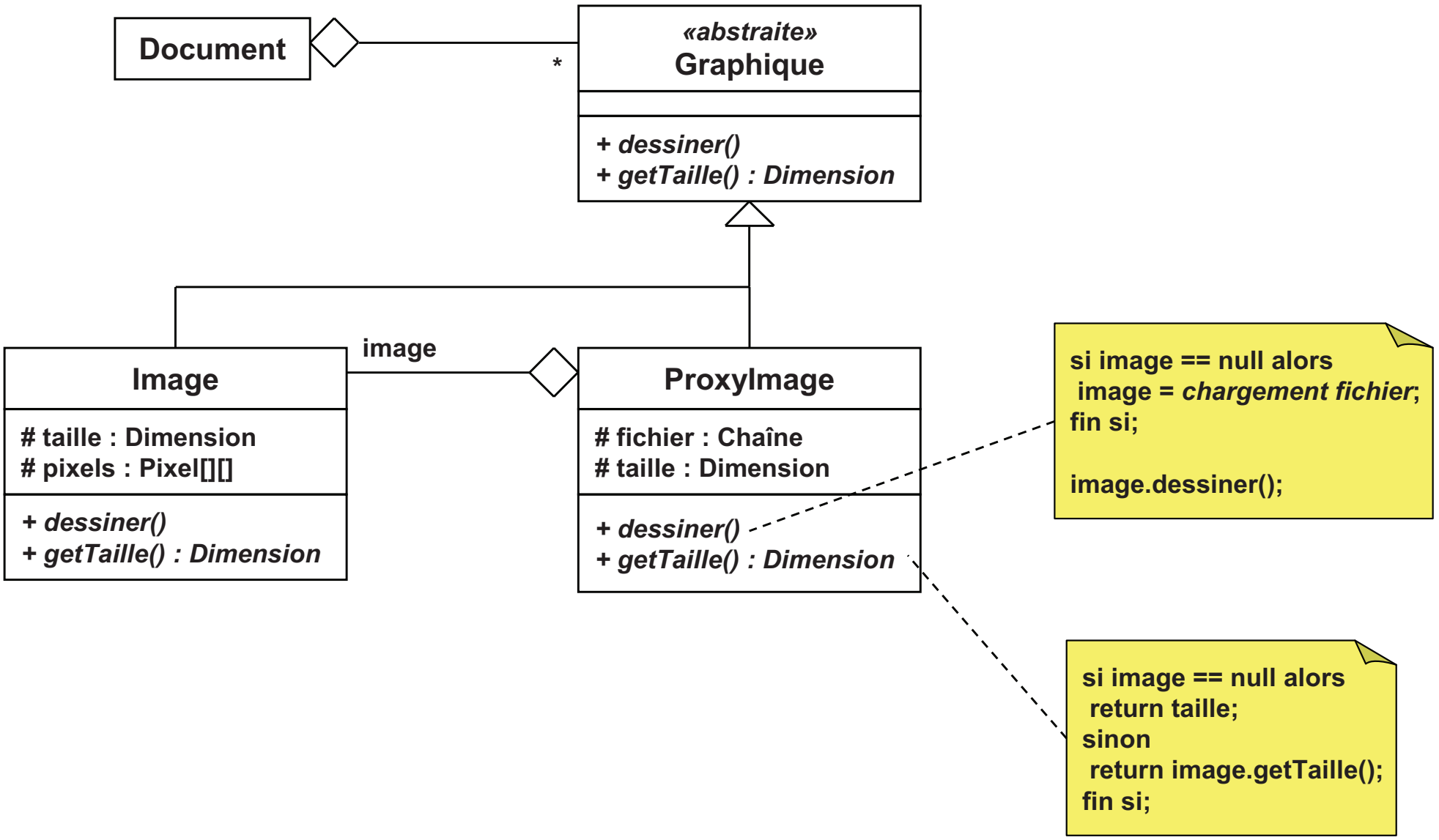
Poids-mouche / *Flyweight (4/4)*

- Intérêts
  - Evite la duplication inutile de données
    - Etat intrinsèque jamais dupliqué
    - Etat extrinsèque calculé ou mémorisé
  - Mais génère un surcoût à l'exécution
    - Lié à la transmission de l'état extrinsèque
- Relations avec d'autres patrons
  - Composite
    - Peuvent être combinés
    - Pour obtenir une arborescence avec feuilles partagées

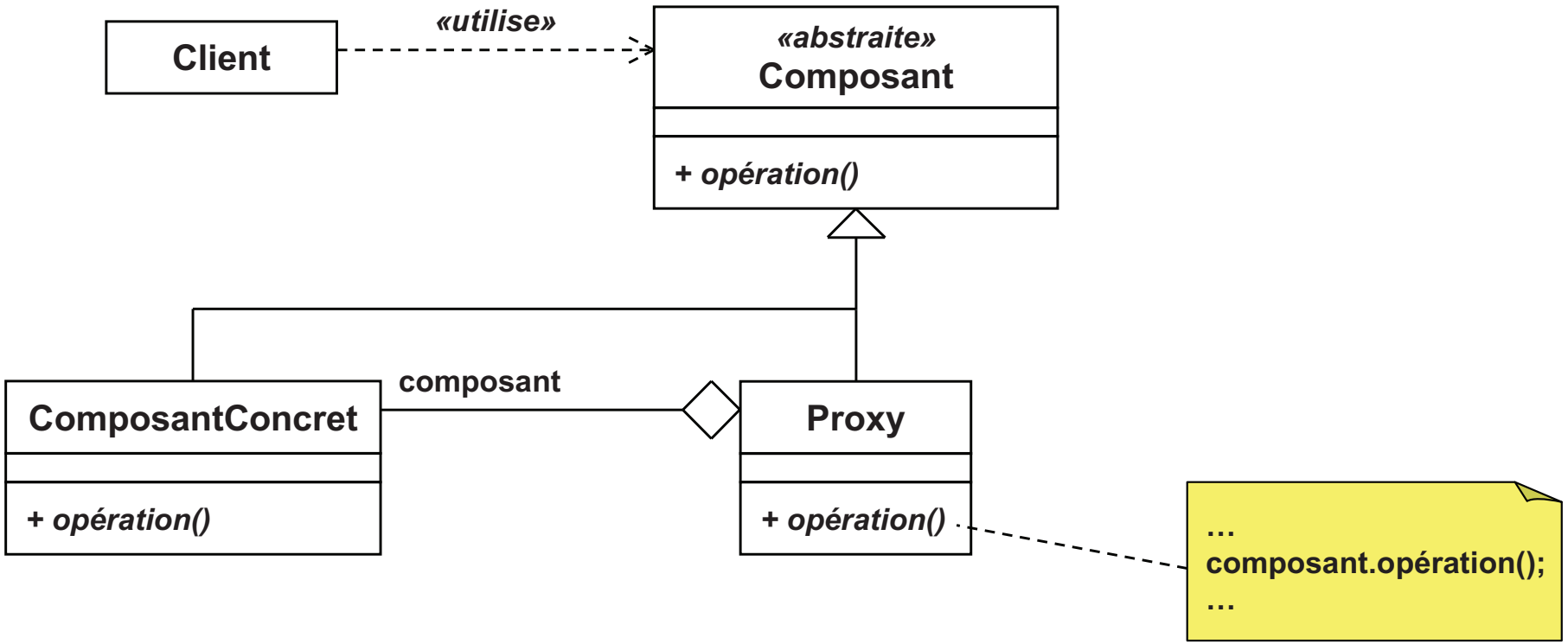
Proxy / Proxy (1/4)

- 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)*

- Appelé aussi *«surrogate»* (substitut)
- Relations avec d'autres patrons
  - Adaptateur
    - Similaires, mais l'adaptateur change l'interface de l'objet
  - Décorateur
    - Similaires, mais des buts différents
    - Décorateur: ajouter des fonctionnalités
    - Proxy: contrôler les accès
- Intérêts
  - 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»*)

ISIMA

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

Patrons de comportement *(2/4)*

- 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)*

- Mémento / *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)*

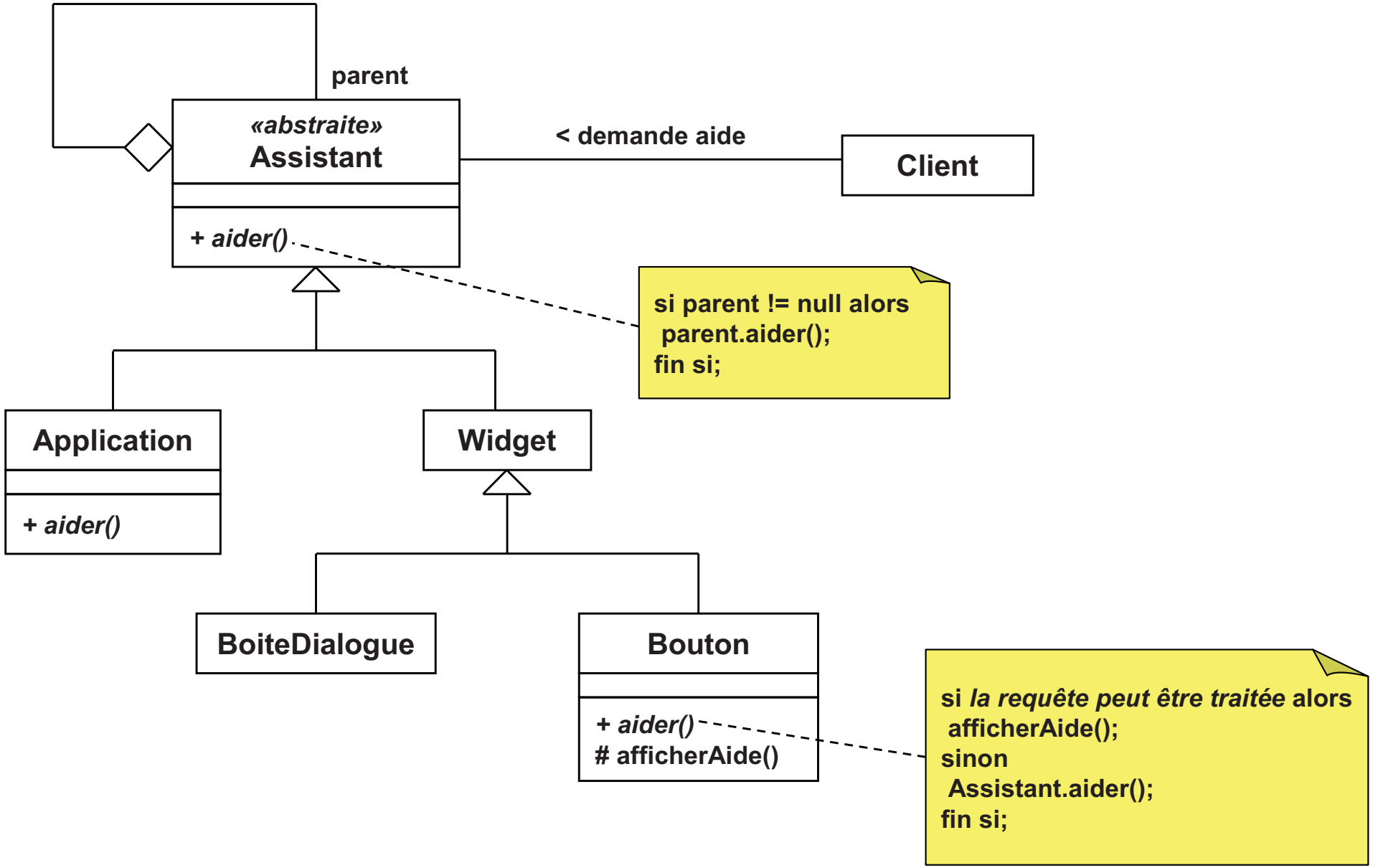
- *«Chain of Responsibility»*

- Objectif
  - Transmettre une requête de proche en proche jusqu'à traitement
  - Eviter le couplage entre l'émetteur et les receveurs potentiels

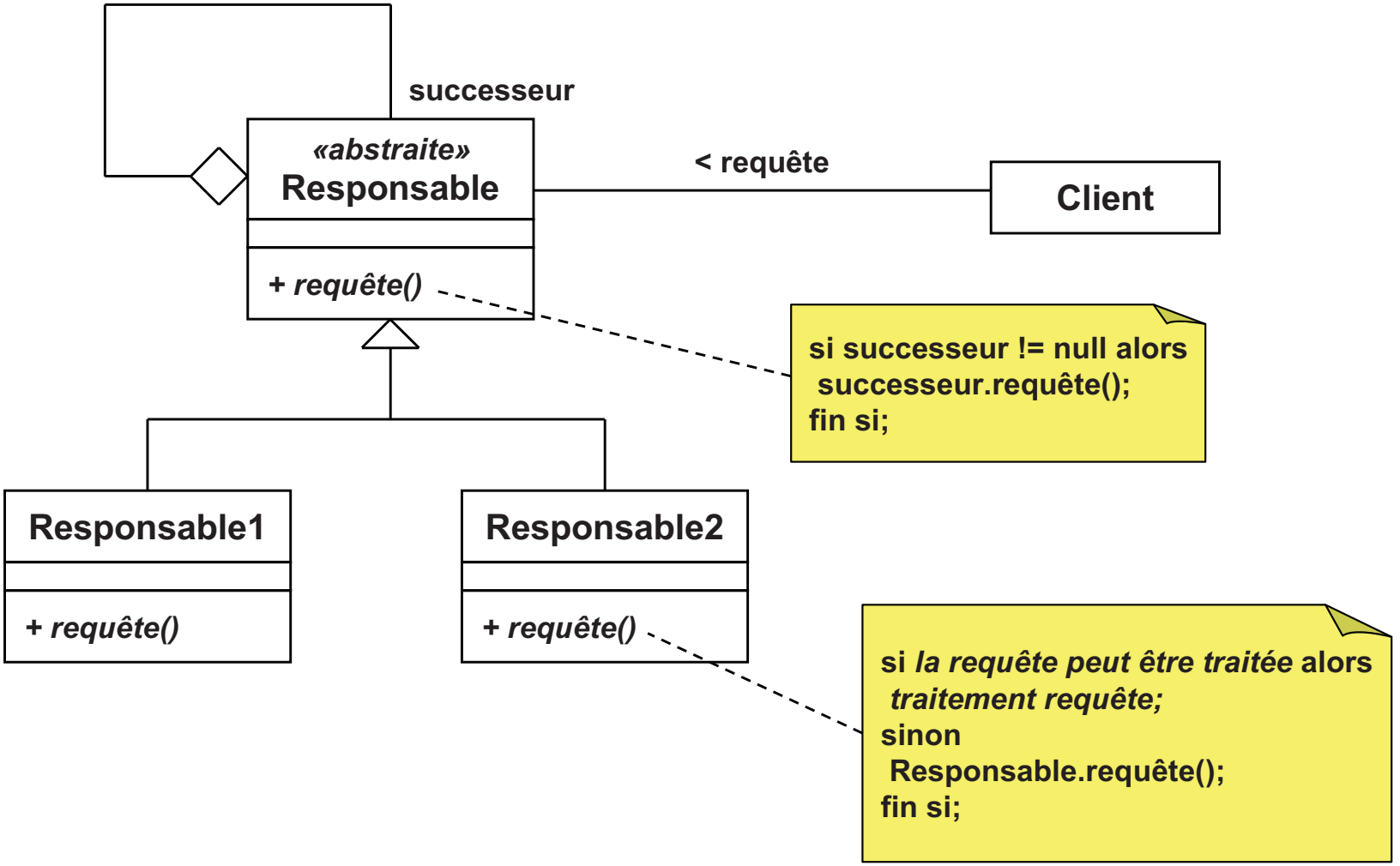
- Principe
  - La requête est transmise de receveur en receveur
    - Jusqu'à ce qu'elle soit traitée
  - Les receveurs ont la même interface pour répondre à la requête
    - Référence sur un successeur pour former une chaîne
  - Le client transmet une requête au premier de la chaîne

- 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)*

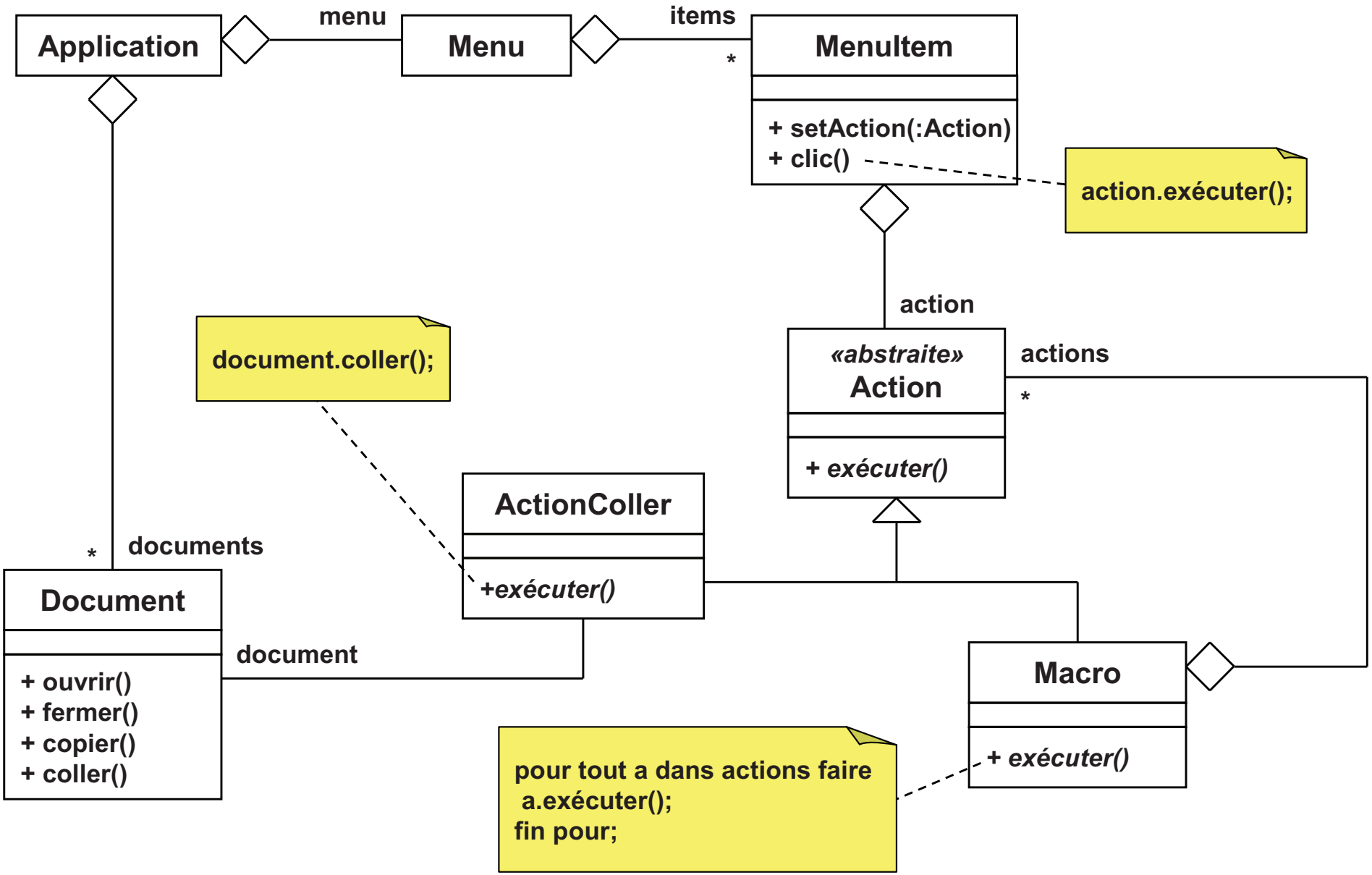
- Intérêts
  - Couplage réduit entre l'émetteur et les receveurs potentiels
    - Pas de connaissance explicite de tous les receveurs
    - Evite que l'émetteur maintienne une liste de candidats
  - Modification dynamique de la chaîne des responsables
    - A tout moment, un objet peut être ajouté ou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

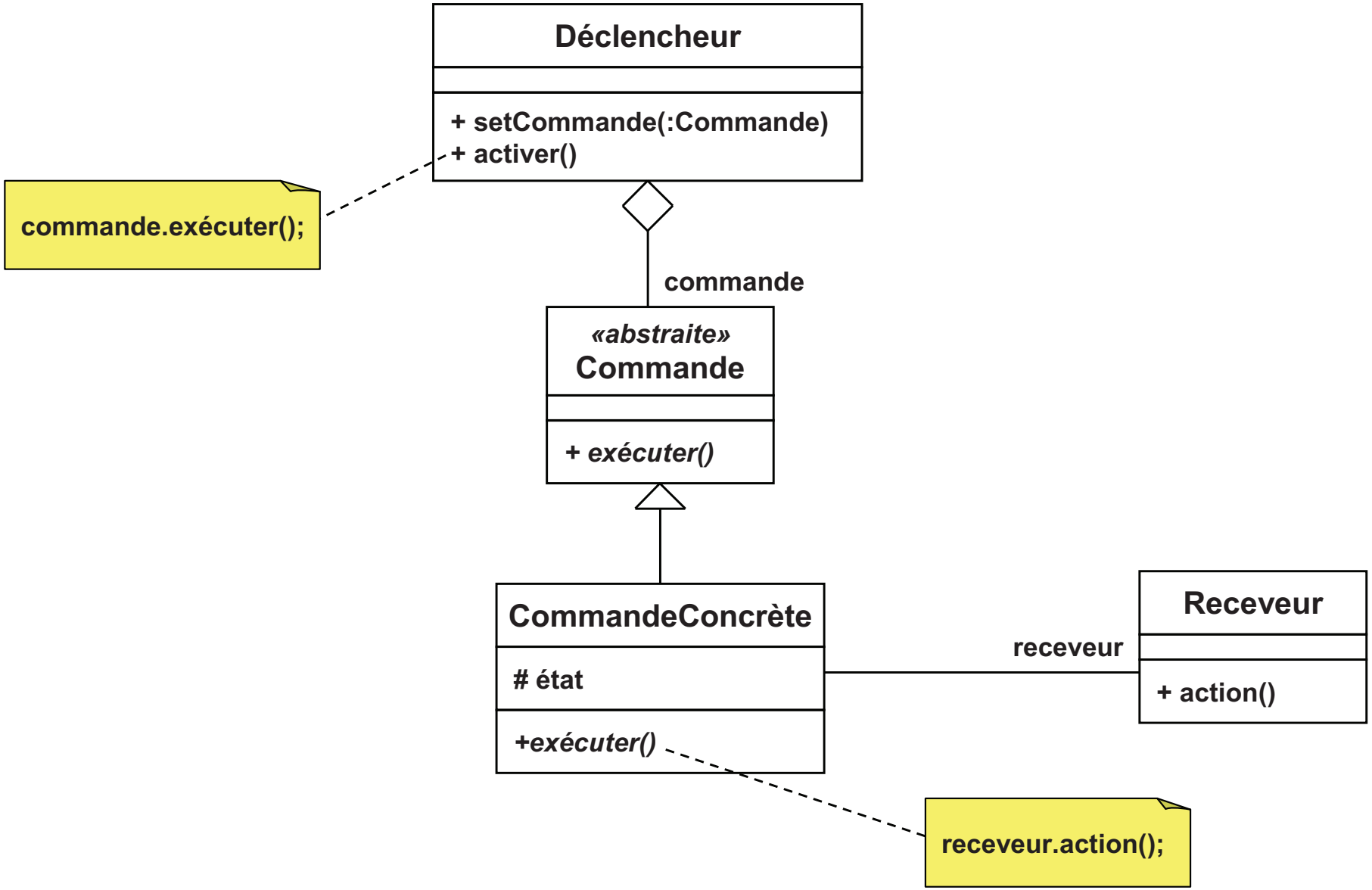
Commande / *Command (1/4)*

- 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 ⇒ 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)*



Commande / *Command (4/4)*

- 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
  - Mémento
    - Mémorisation d'informations pour un processus d'annulation

Interpréteur / *Interpreter*

- 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

Itérateur / *Iterator (1/4)*

- Objectif
  - Fournir un accès séquentiel aux éléments d'un agrégat
  - Sans exposer sa représentation interne

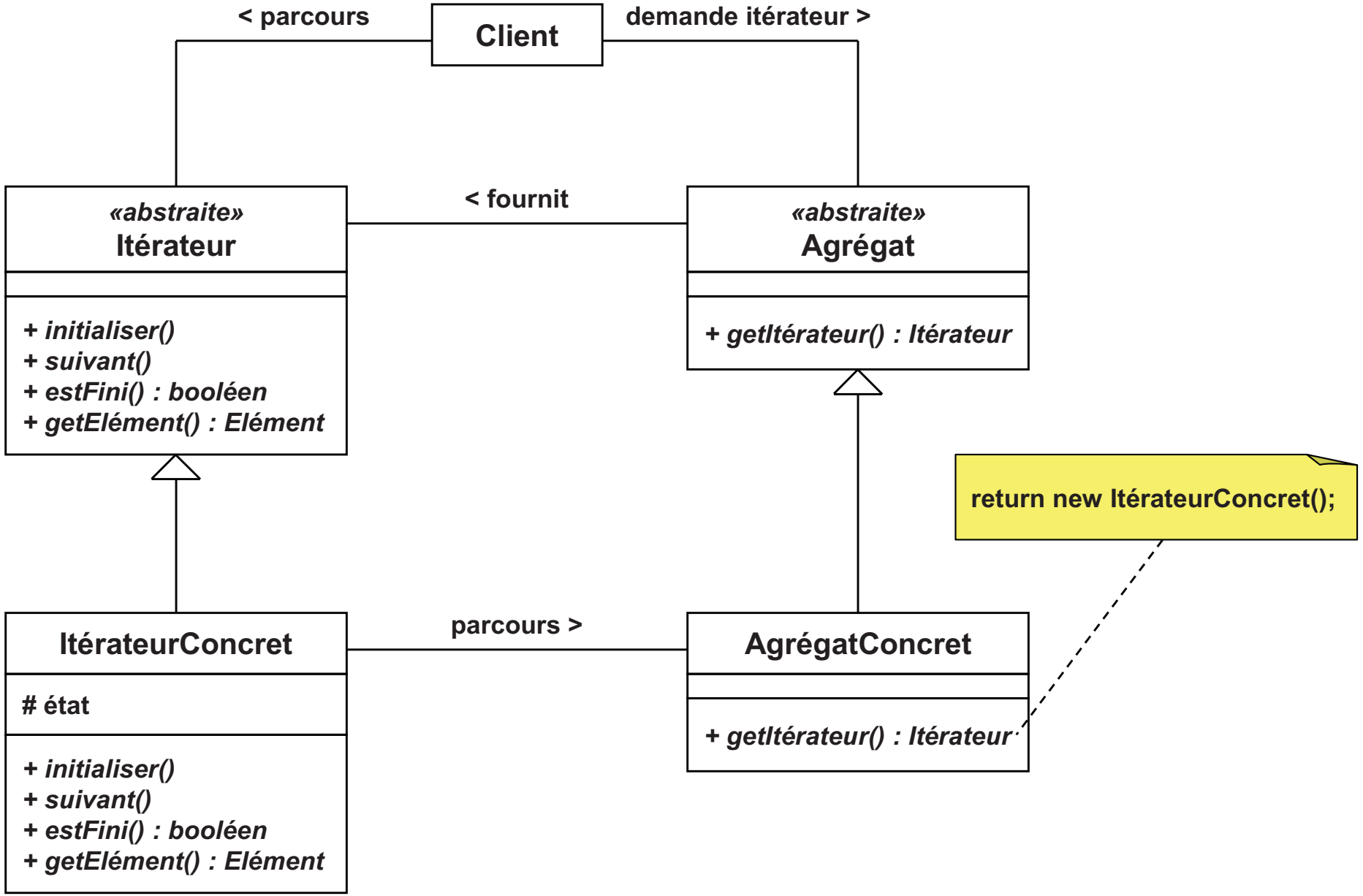
- Principe
  - Itérateur = «pointeur» sur un élément d'un agrégat
    - L'agrégat fournit des itérateurs
    - Tous implémentent la même interface (quelque soit l'agrégat)
  - Le client ne manipule que l'itérateur
    - Il n'a pas forcément connaissance de l'agrégat
  - L'itérateur connaît la structure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

- Motivation
  - Parcourir les éléments d'un conteneur

Itérateur / *Iterator (2/4)*

- L'itérateur fournit généralement 4 services
  - Positionnement sur le 1er 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)*



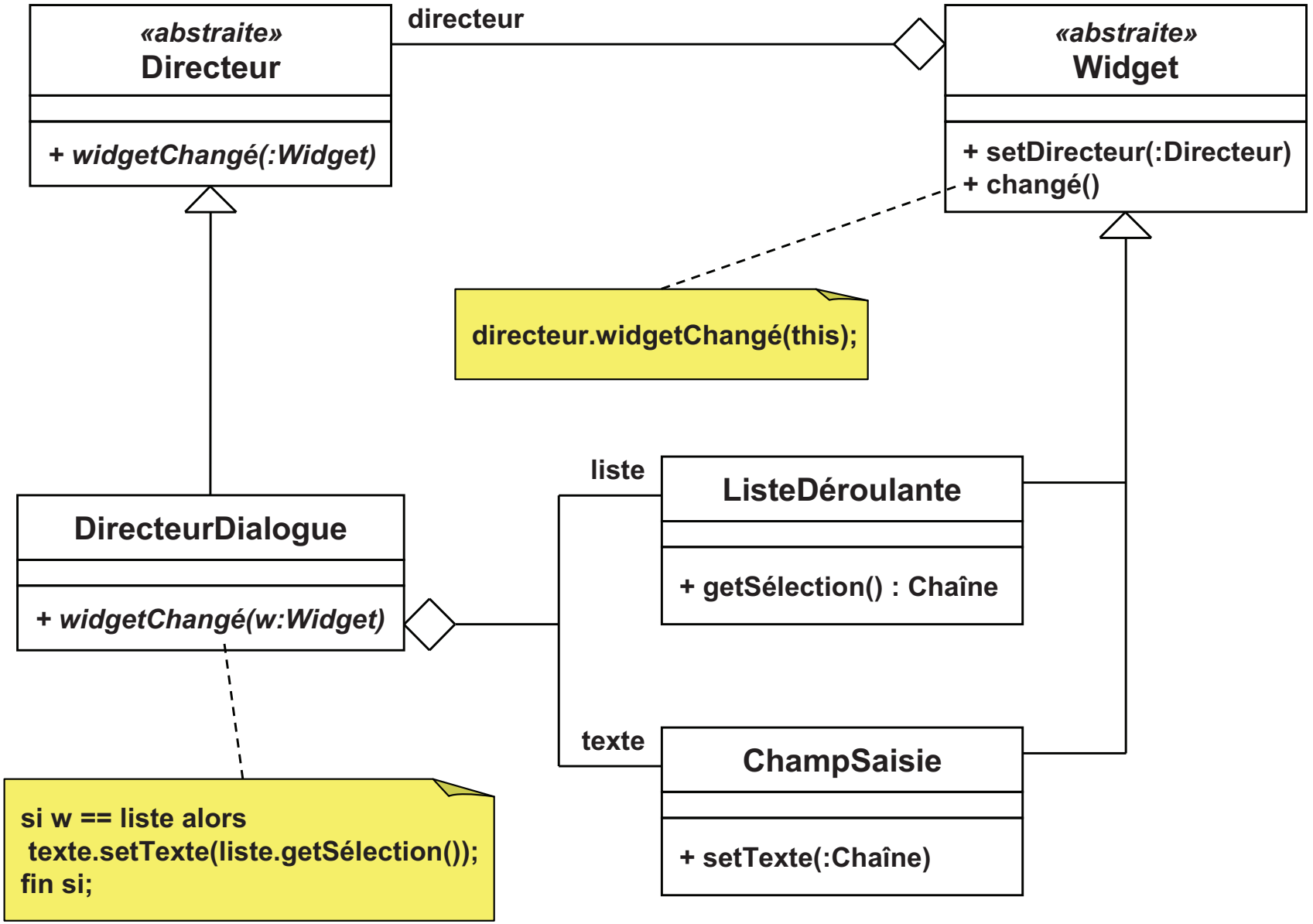
Itérateur / *Iterator (4/4)*

- 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

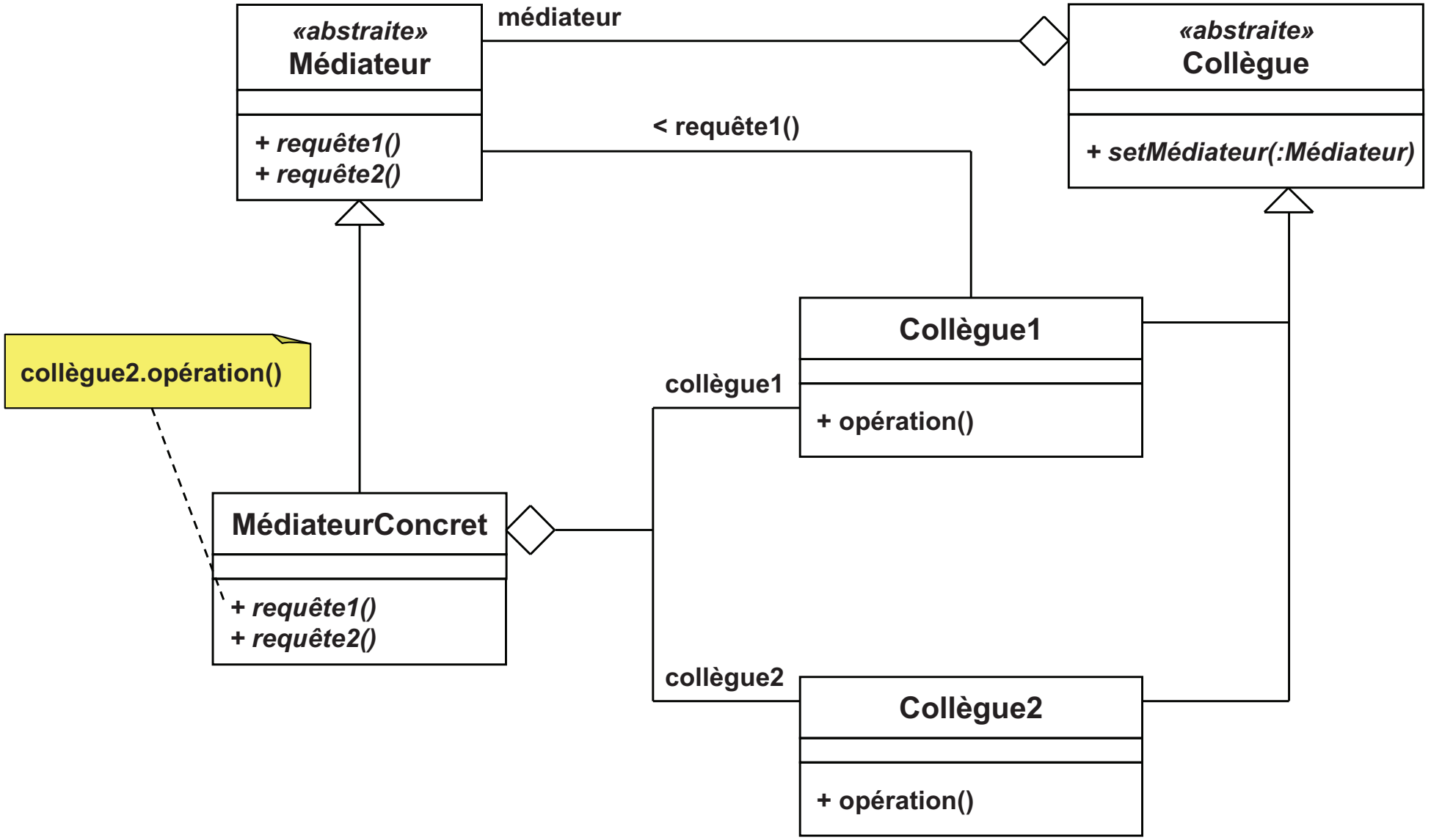
Médiateur / *Mediator (1/4)*

- 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)*



Médiateur / *Mediator (4/4)*

- 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

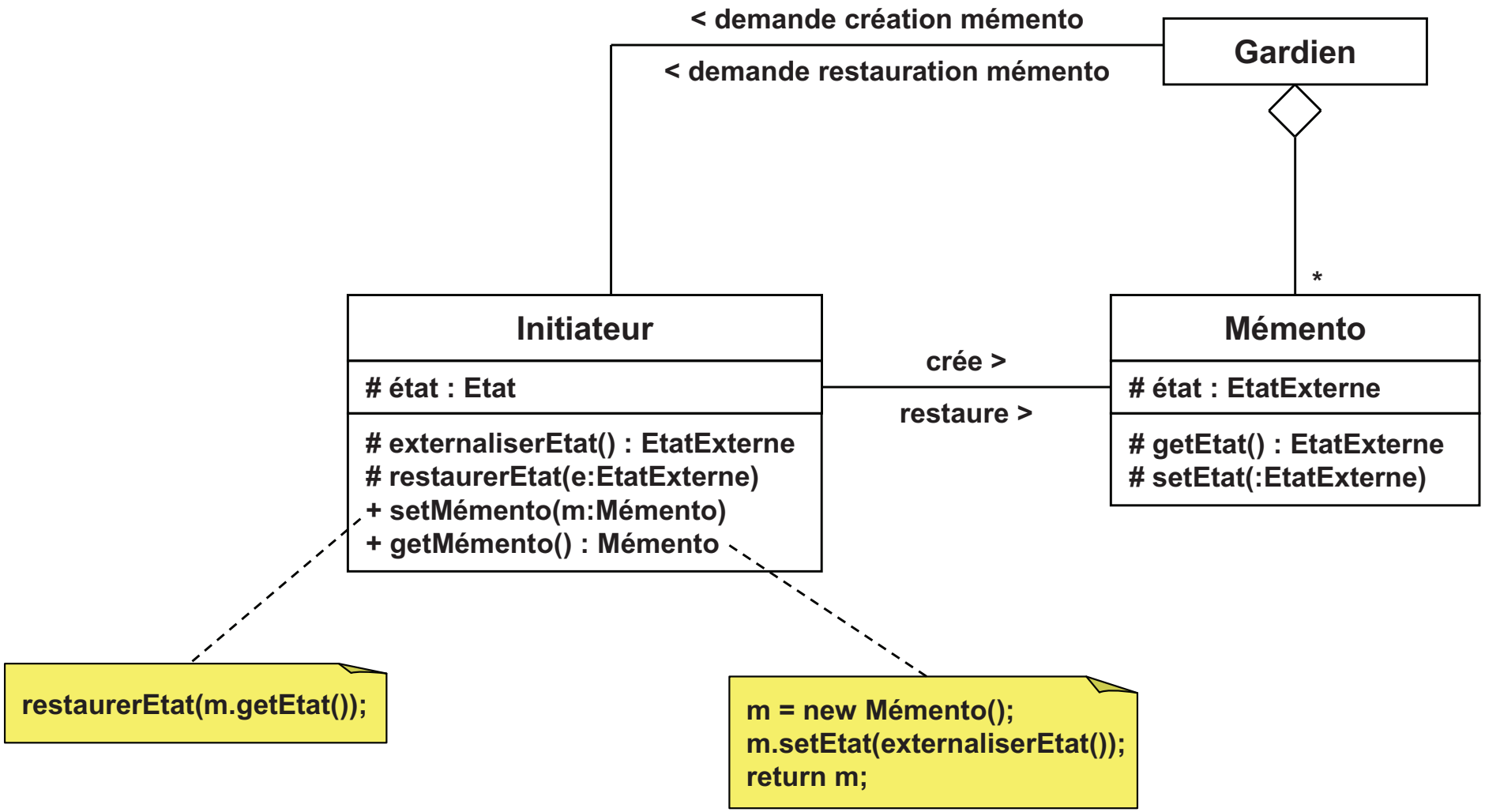
Mémento / *Memento (1/3)*

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

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

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

Mémento / *Memento (2/3)*



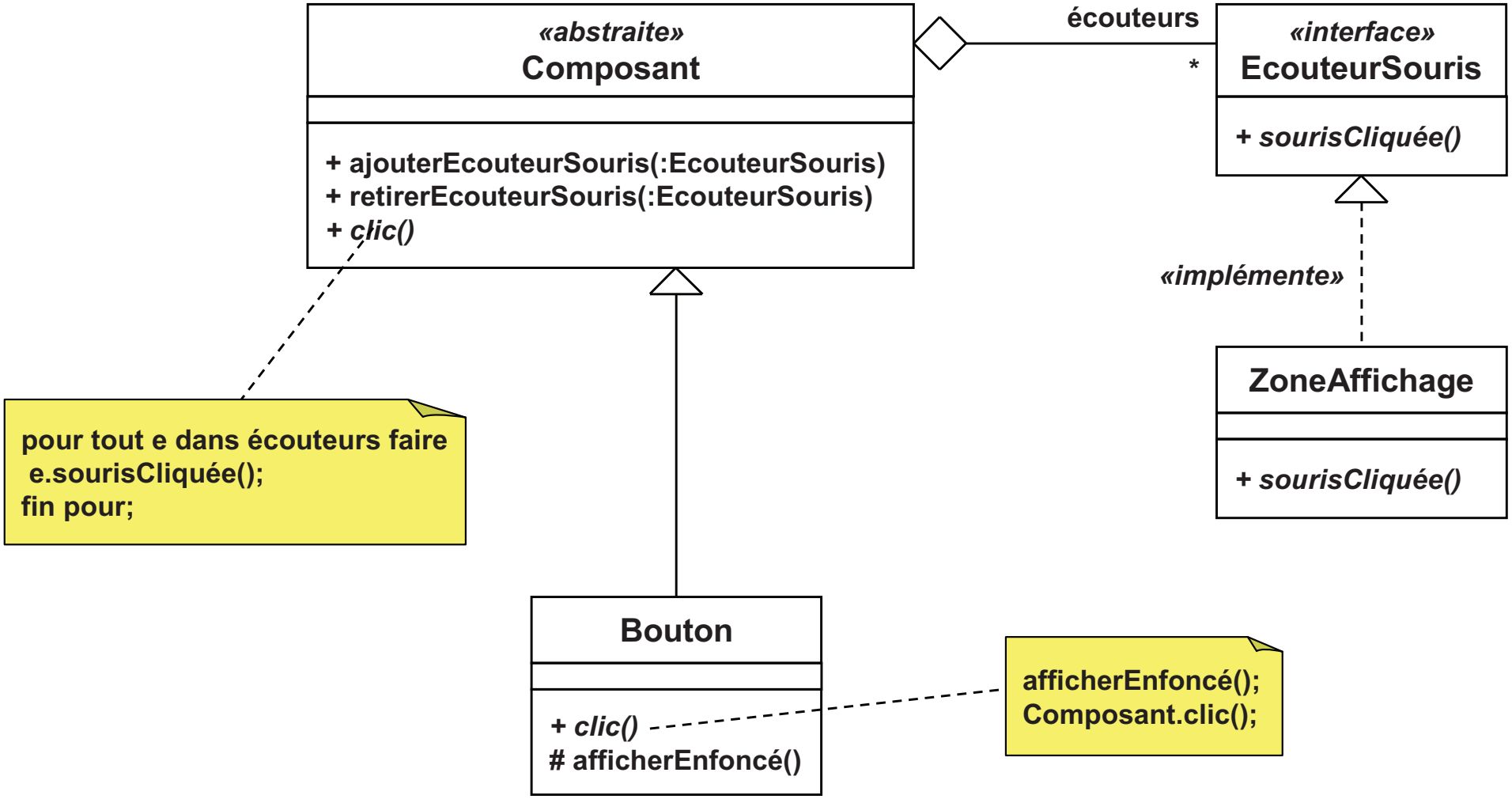
Mémento / *Memento (3/3)*

- 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 mémentos 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
    - Mémento = état d'une itération

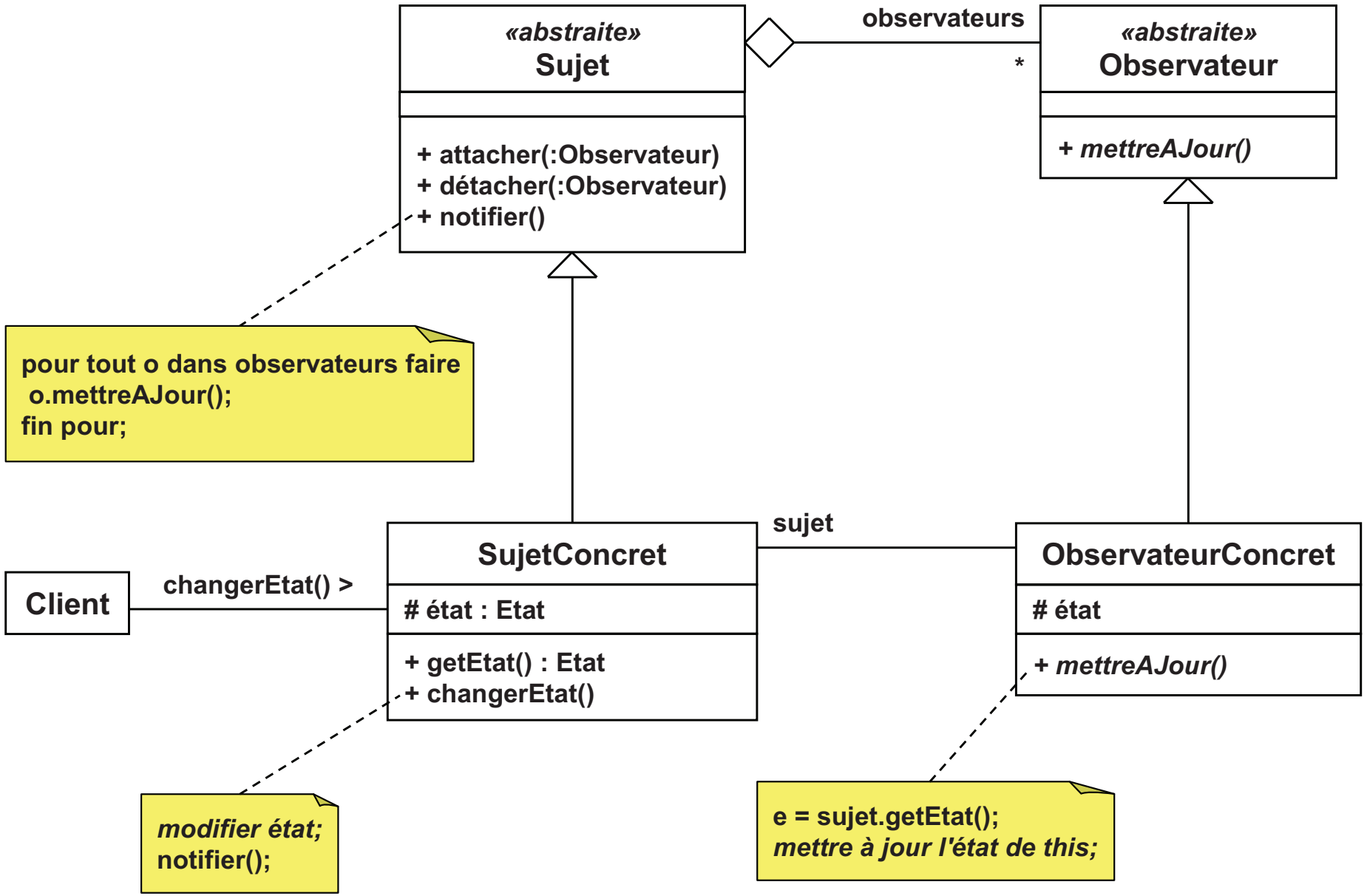
Observateur / *Observer (1/4)*

- 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 $\Rightarrow$ 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

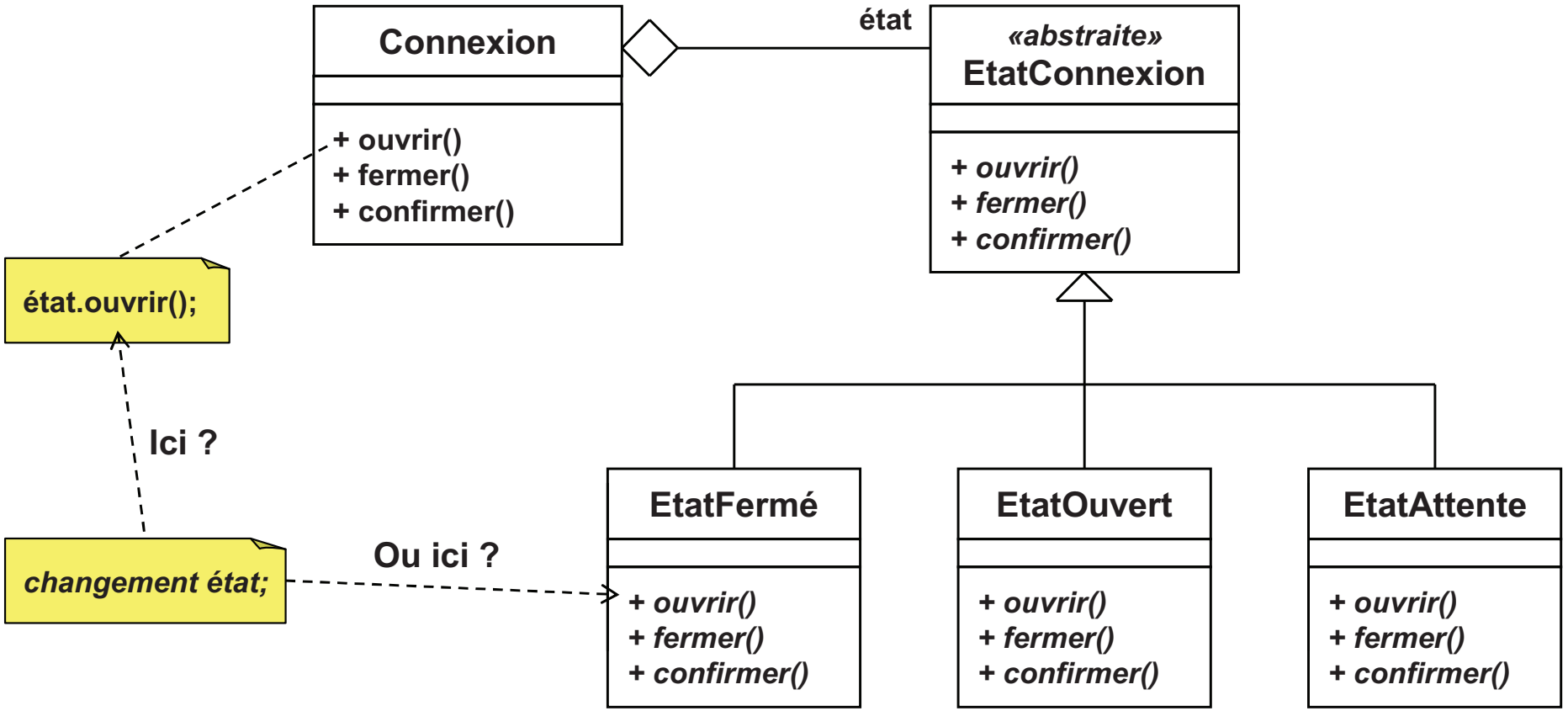
- Implémentation
  - Stockage de l'association observateur-sujet
    - Interne, chaque sujet maintient une liste d'observateurs
    - Externe, dans un conteneur associatif
  - Un observateur peut avoir plusieurs sujets
    - La méthode «`mettreAJour`» doit recevoir le sujet

- Relations avec d'autres patrons
  - Modèle-Vue-Contrôleur (MVC)
    - Modèle = sujet
    - Vue = observateur

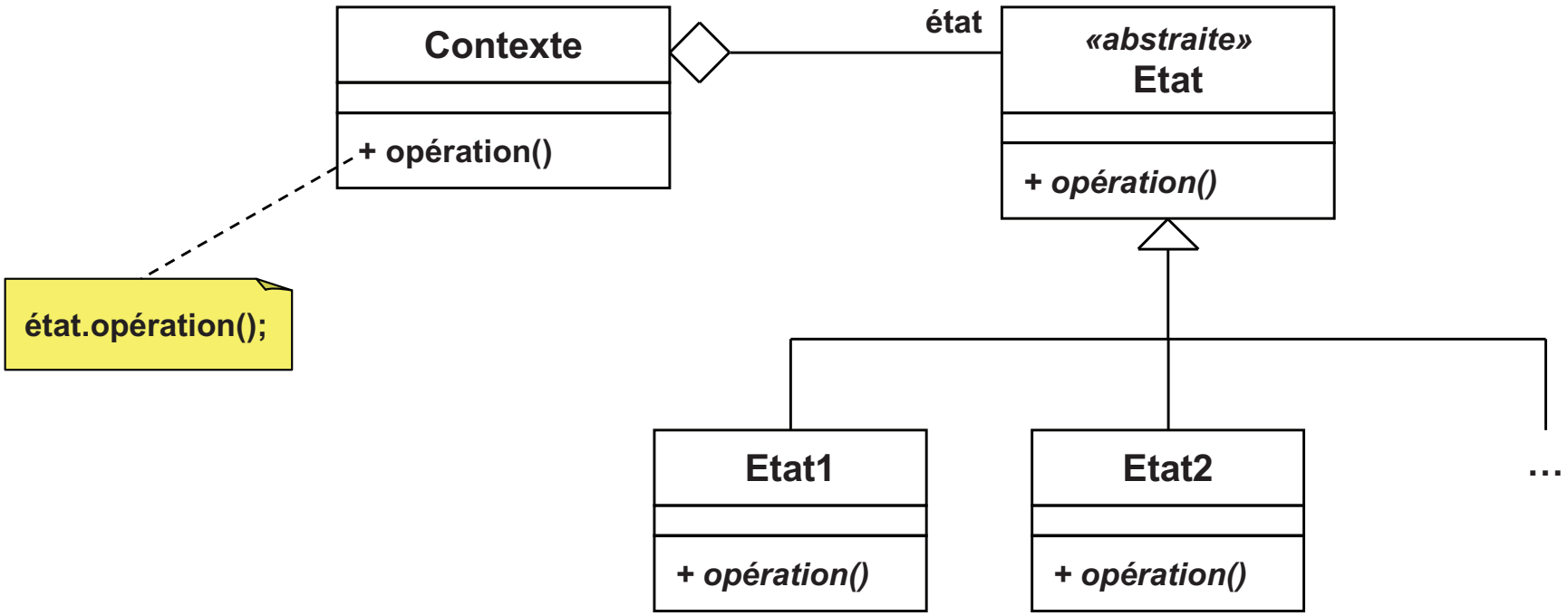
Etat / *State (1/4)*

- 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 ⇒ un état différent
  - Un objet agrège un état
    - Changement de comportement ⇒ 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)*



Etat / *State (4/4)*

- Appelé aussi *«objects for states»*

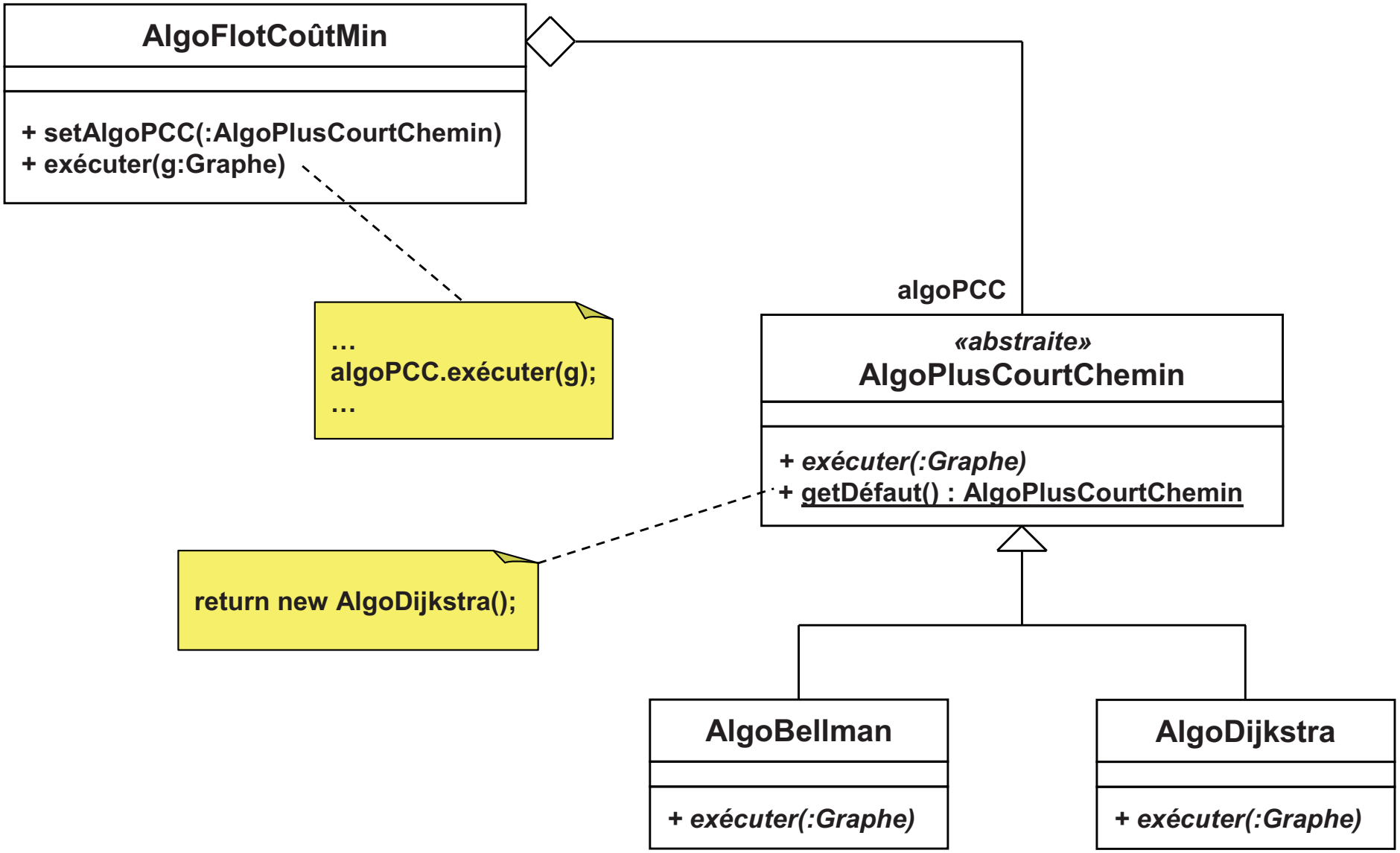
- Intérêts
  - Evite des tests de comportement suivant l'état
  - Les comportements spécifiques sont localisés
    - Donc faciles à maintenir
  - Les transitions d'état sont explicites
    - Mais qui change l'état de l'objet ?
    - L'objet lui-même ou son objet état ?

- Relations avec d'autres patrons
  - Poids-mouche
    - Les objets états peuvent être partagés entre objets
  - Singleton
    - Les objets états peuvent être des singletons

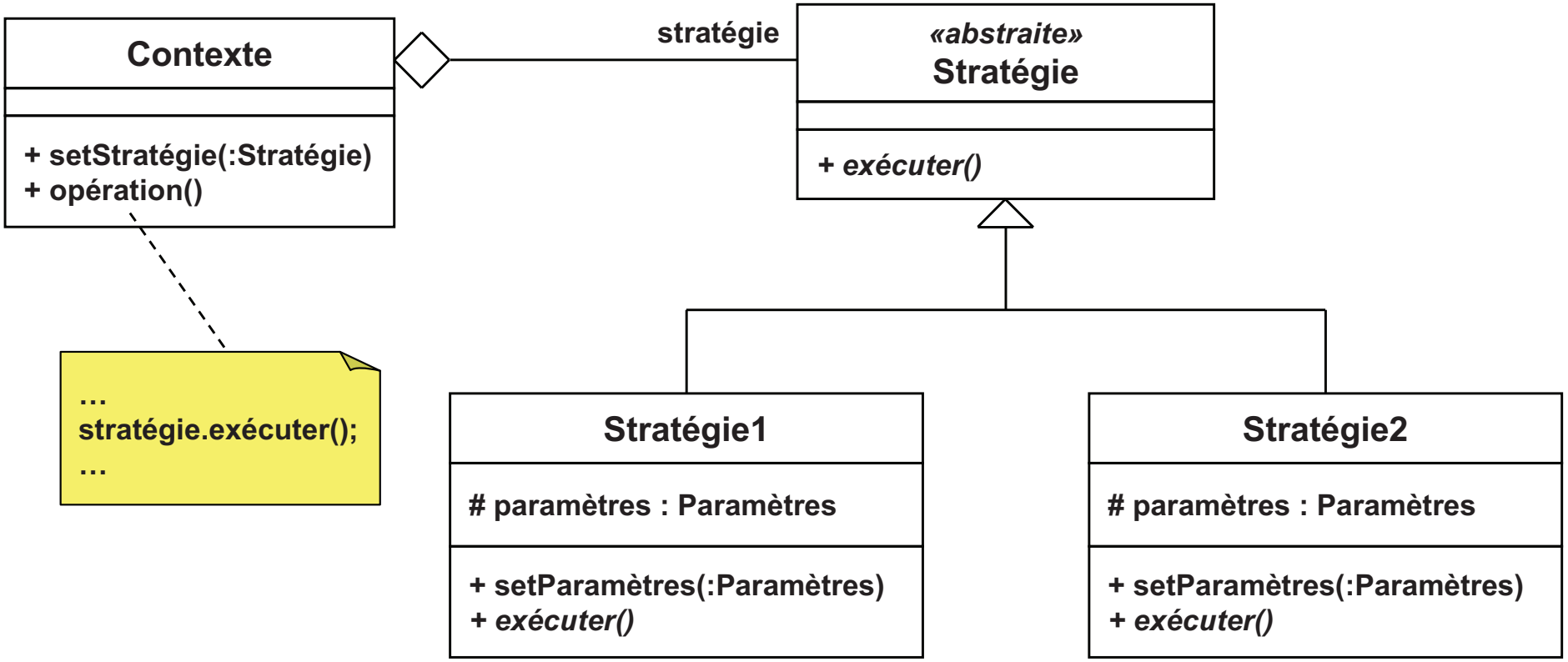
Stratégie / *Strategy (1/4)*

- 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 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)*



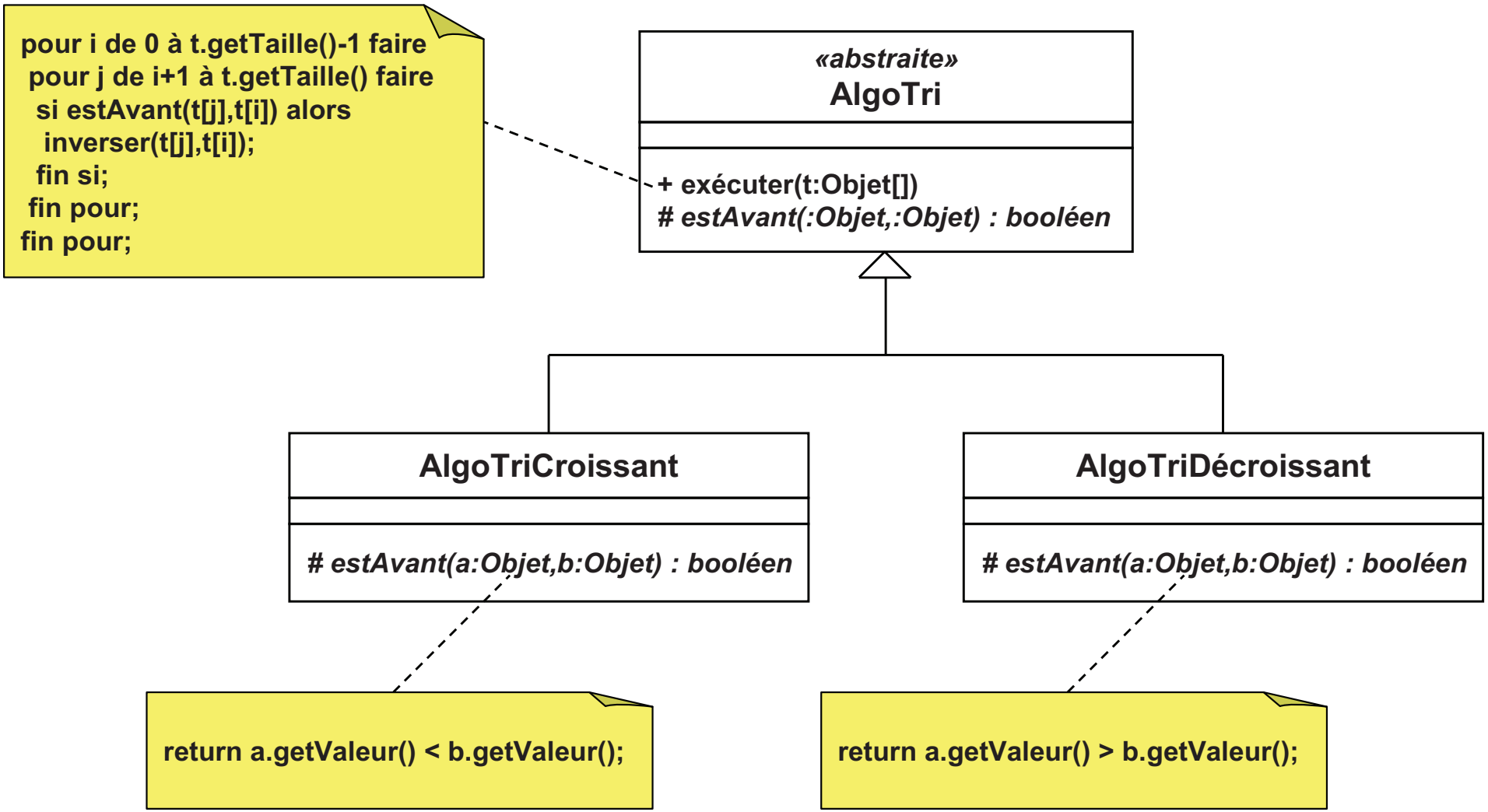
Stratégie / *Strategy (4/4)*

- Appelé aussi *«policy»*
- Intérêts
  - Abstraction de la stratégie
    - Interchangeable dynamiquement
    - Nouvelle stratégie ⇒ 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

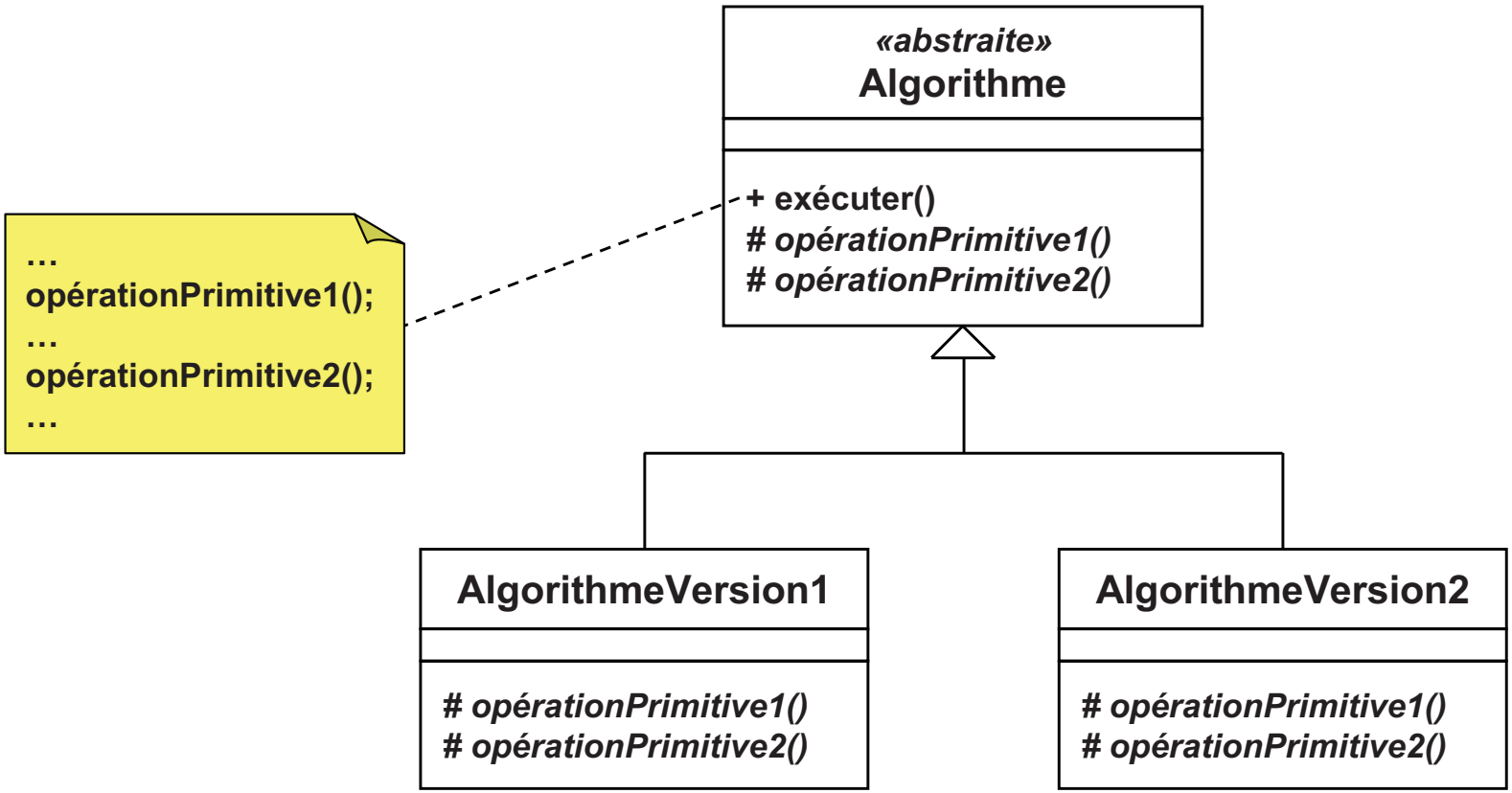
Méthode patron / *Template Method (1/6)*

- 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 ⇒ Surcoût lié à la virtualité
    - Trop de méthodes ⇒ 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

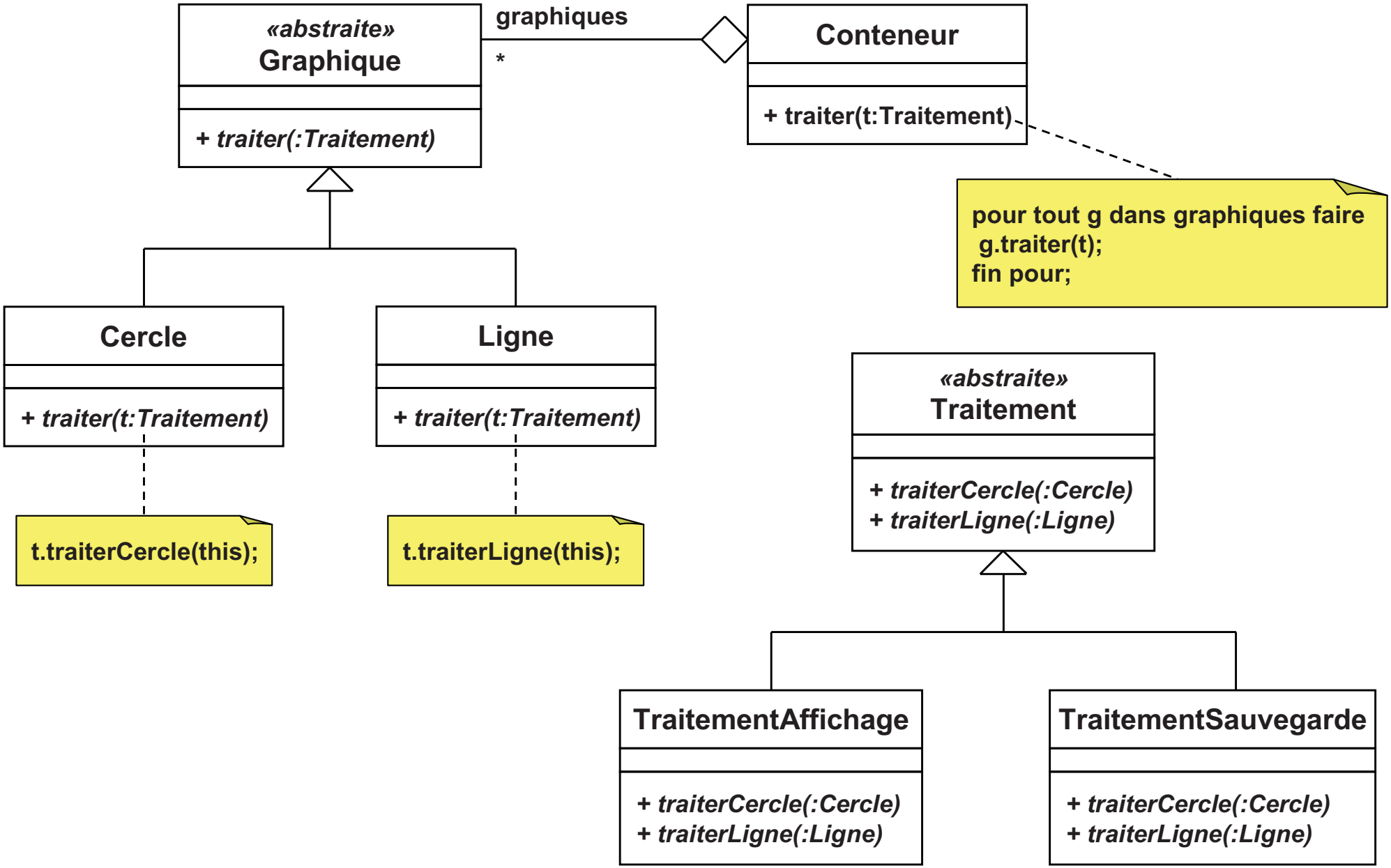
Visiteur / *Visitor (1/4)*

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

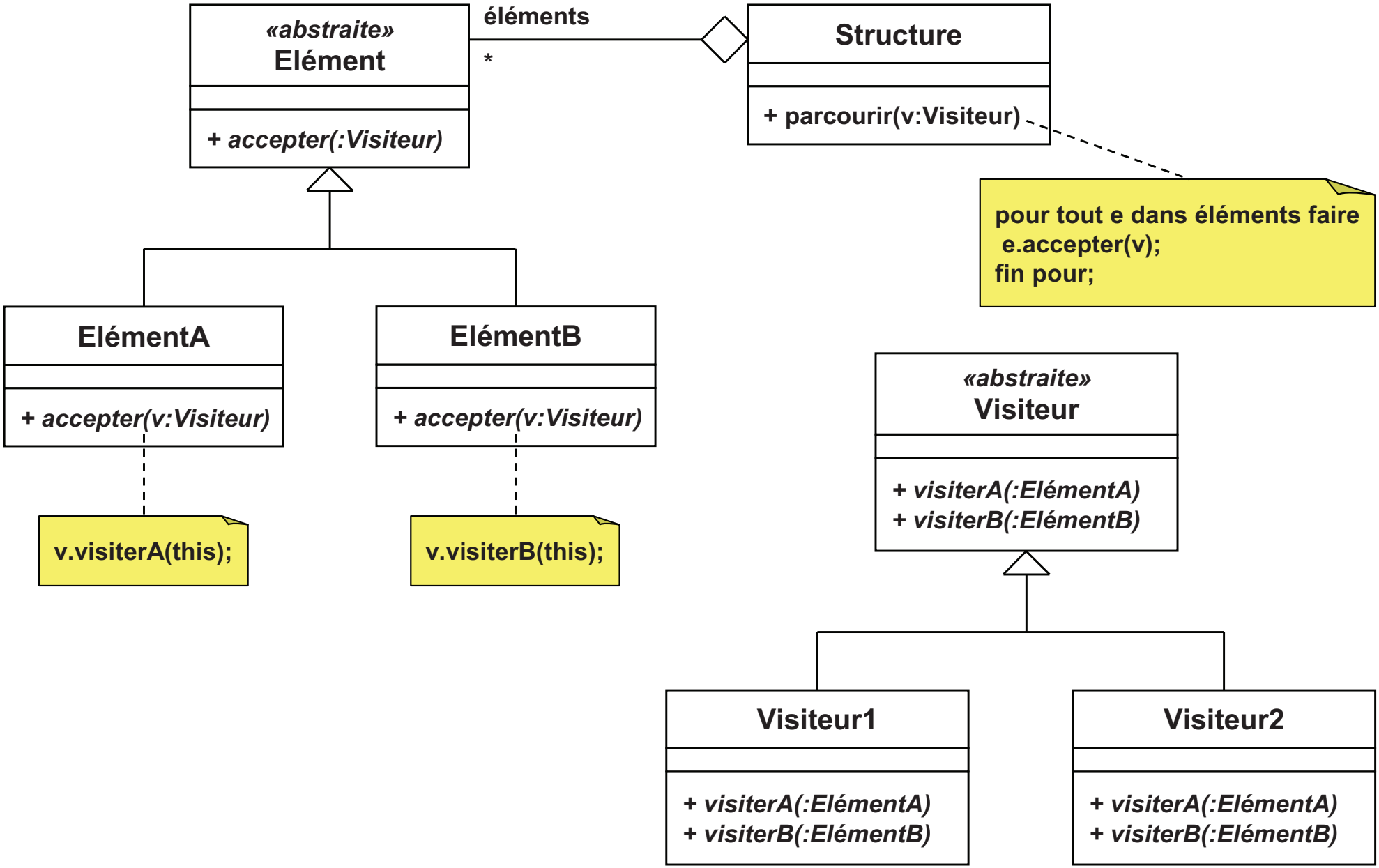
- Principe
  - L'opération est modélisée par un objet, le «visiteur»
    - Une classe, extensible, représente l'opération
  - Les éléments doivent «accepter» un visiteur
    - Une méthode doit recevoir le visiteur
    - Et appliquer l'opération associée sur l'élément
  - Une procédure de parcours applique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

- 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

Conclusion

- 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
    - …

ISIMA

Patrons de services

(PARTIE VII - Patrons de conception)

Bruno Bachelet

Christophe Duhamel

Luc Touraille

Patrons de service *(1/2)*

- Accéder et configurer des services
- Quelque soit le mécanisme de communication entre composants
- Application *«standalone»*
  - Simple espace d'adressage
    - Fonctions avec paramètres
    - Variables globales
- Application en réseau
  - Communication interprocessus (IPC)
    - Mémoire partagée, tubes, sockets (TCP, UDP)…
  - Protocoles de communication
    - Telnet, FTP, SSH, HTTP…
  - Opérations distantes via services
    - CORBA, COM+…

Patrons de service *(2/2)*

- Façade d'adaptation / *Wrapper Facade*
  - Fournir une interface objet pour une API non objet existante
- Configurateur (de composant) / (*Component) Configurator*
  - Lier dynamiquement des implémentations (sans recompilation)
- Intercepteur / *Interceptor*
  - Ajouter de manière transparente des services à un *framework*
- Interface d'extension / *Extension Interface*
  - Permettre à un composant d'exporter plusieurs interfaces

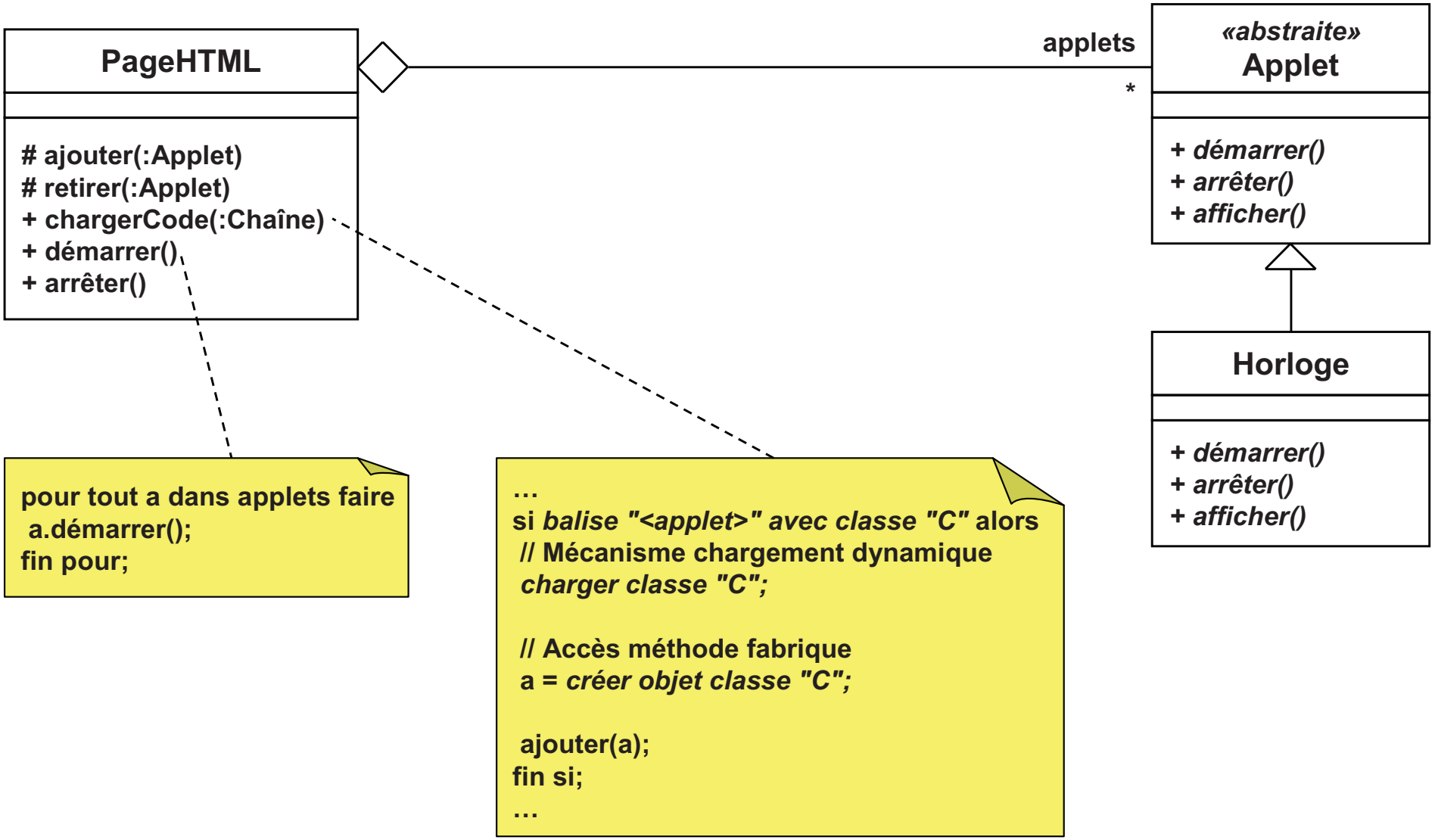
Configurateur / *Configurator (1/4)*

- Objectif
  - Lier et délier l'implémentation de composants à l'exécution
  - Sans recompiler, ni éditer les liens

- Principe
  - Une classe abstraite représente les composants
    - Nouvelle implémentation = définition d'une sous-classe
    - Sous-classe stockée dans une unité chargeable dynamiquement
      - Exemples: bibliothèque dynamique (DLL), classe Java
  - Un «configurateur» manipule les composants
    - S'occupe de la création, du démarrage et de l'arrêt des composants

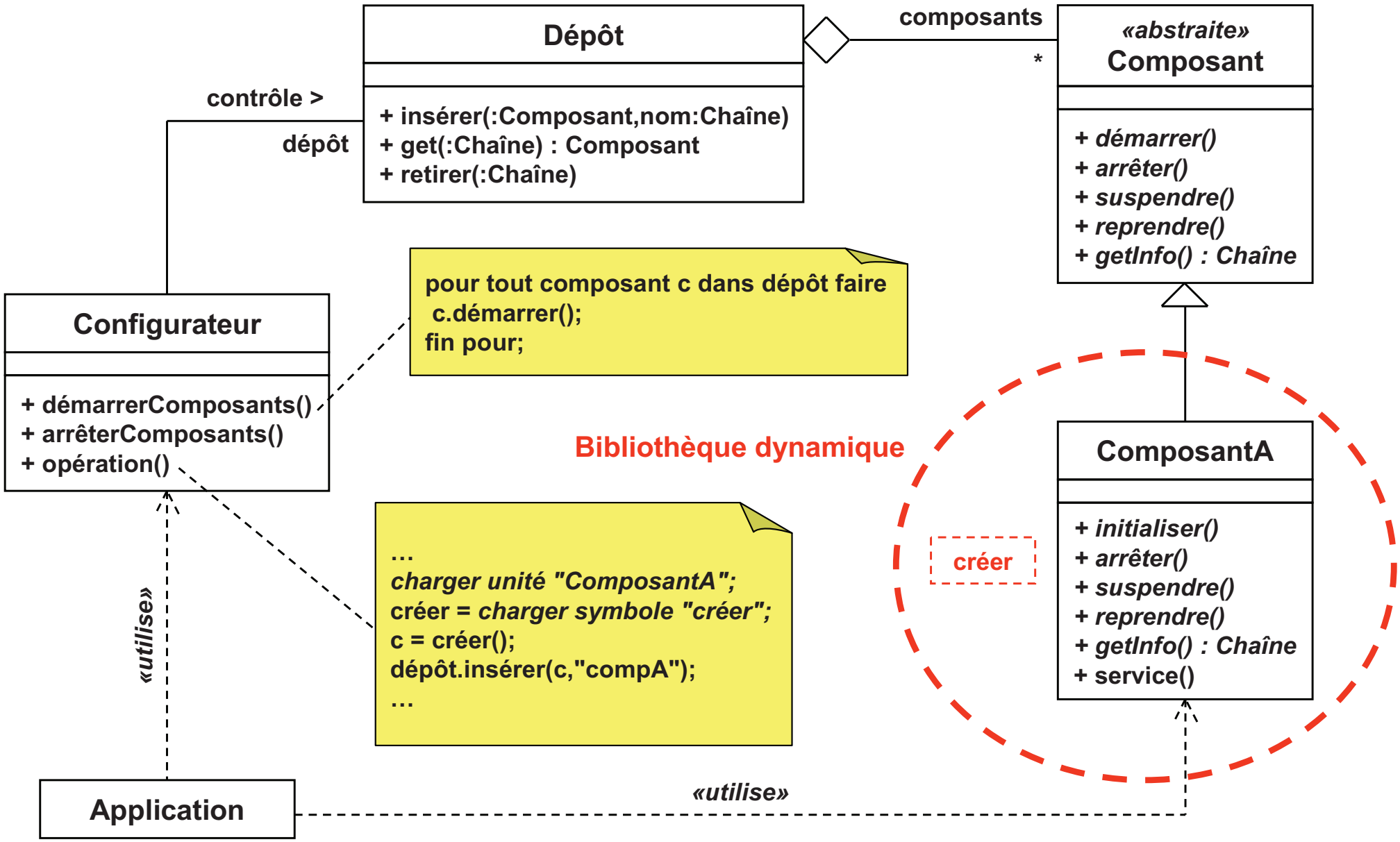
- Motivation
  - Page HTML où les applets sont chargées dynamiquement

Configurateur / *Configurator (2/4)*



Exemple code HTML: `<applet code="Horloge.class">...</applet>`

Configurateur / *Configurator (3/4)*



Configurateur / *Configurator (4/4)*

- Appelé aussi *«service configurator»*

- Intérêts
  - Uniformité des composants
    - Tous les composants respectent la même interface
  - Administration centralisée
    - Facilite le démarrage/arrêt de tous les composants
  - Remplacement de composants «à chaud»
    - Chargement/déchargement dynamique de composants
  - Permet une adaptation dynamique
    - Mécanisme d'analyse / Apprentissage

    ⇒ Réglage des paramètres du composant

    ⇒ Chargement d'un composant mieux adapté

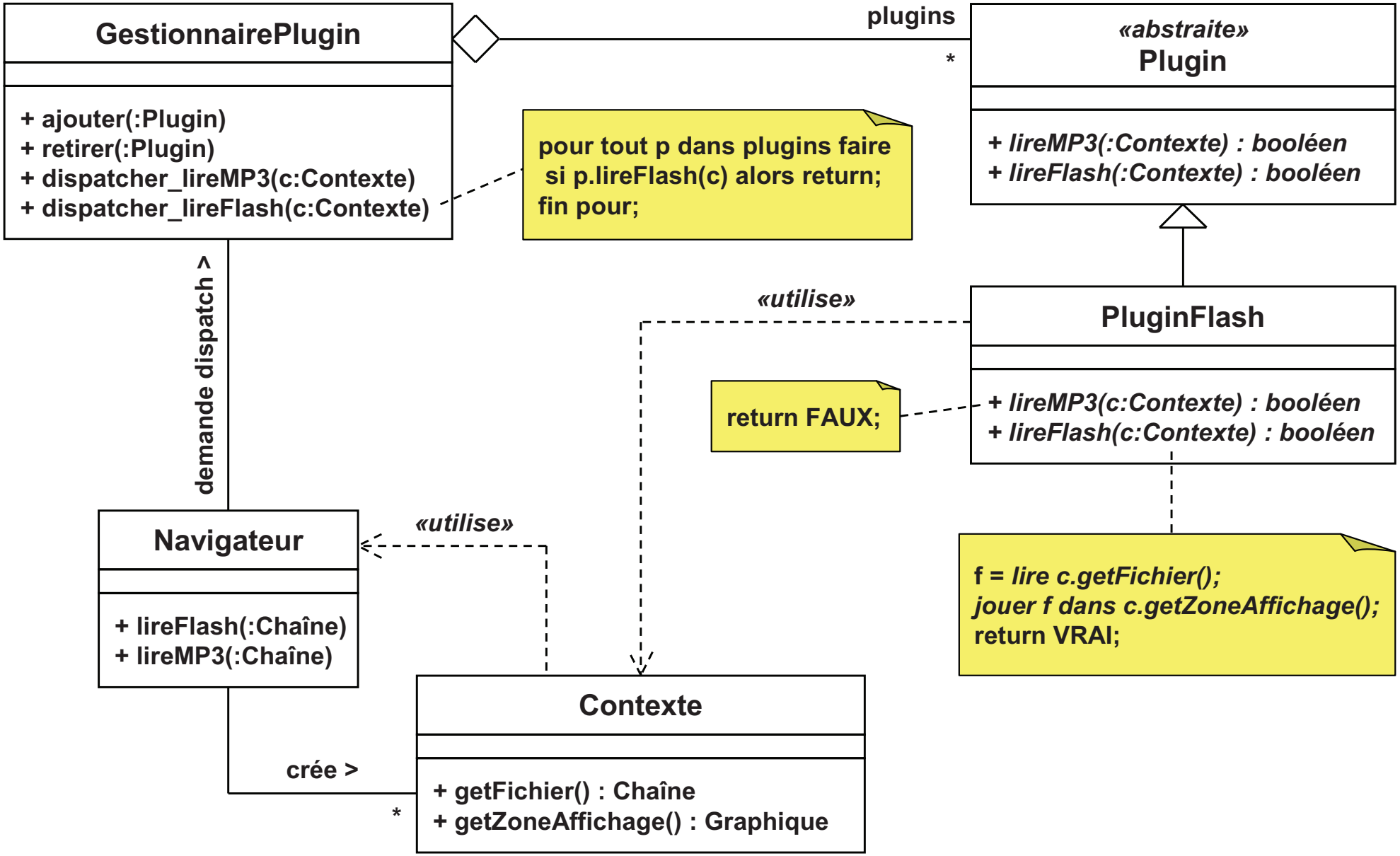
Intercepteur / *Interceptor (1/4)*

- Objectif
  - Ajouter de manière transparente des services à un *framework*
  - Les activer automatiquement suite à certains événements

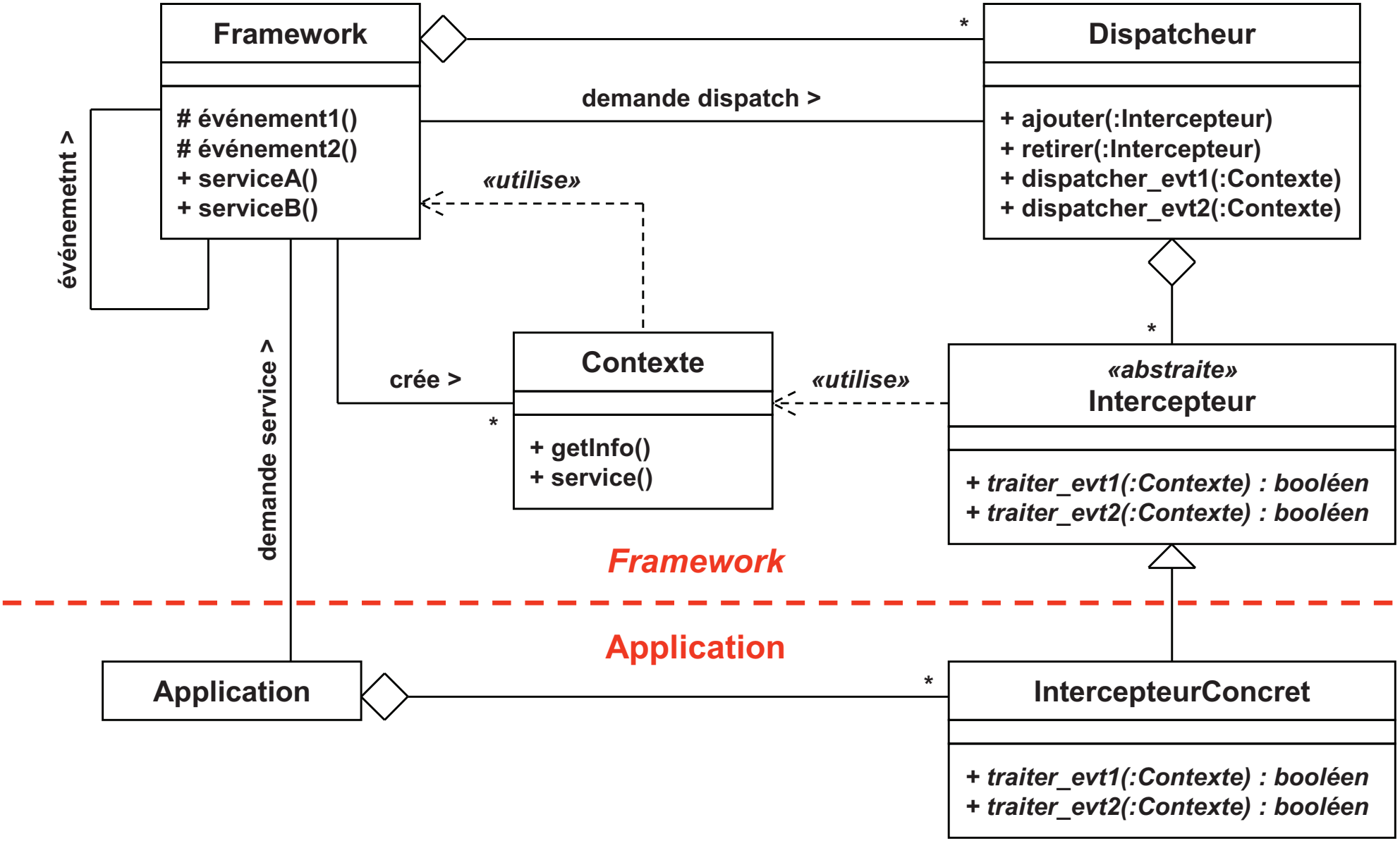
- Principe
  - Un «dispatcheur» est chargé de diffuser les événements
  - Des «intercepteurs» sont capables de recevoir les événements
    - Classe abstraite possédant une méthode par événement
    - Héritage $\Rightarrow$ Proposition de nouveaux services
  - Les intercepteurs n'ont pas un accès direct au *framework*
    - Communication par l'intermédiaire d'un «contexte»
    - Le contexte accède aux données et services du *framework*

- Motivation
  - Mécanisme de *plugins* dans un navigateur Web

Intercepteur / *Interceptor (2/4)*



Intercepteur / *Interceptor (3/4)*



Intercepteur / *Interceptor (4/4)*

- Intérêts
  - Extensibilité du *framework*
    - Services liés aux intercepteurs intégrés au *framework*
    - Nouveaux services ⇒ Héritage de «`Intercepteur`»
      - Aucun impact sur le *framework*
  - Séparation des intérêts
    - Infrastructure du *framework* d'un côté
    - Les services de l'autre
      - Inutile de connaître toute l'infrastructure pour coder un service
  - Mécanisme de contrôle et de surveillance
    - Intercepteur + contexte ⇒ Moyen de tracer l'application

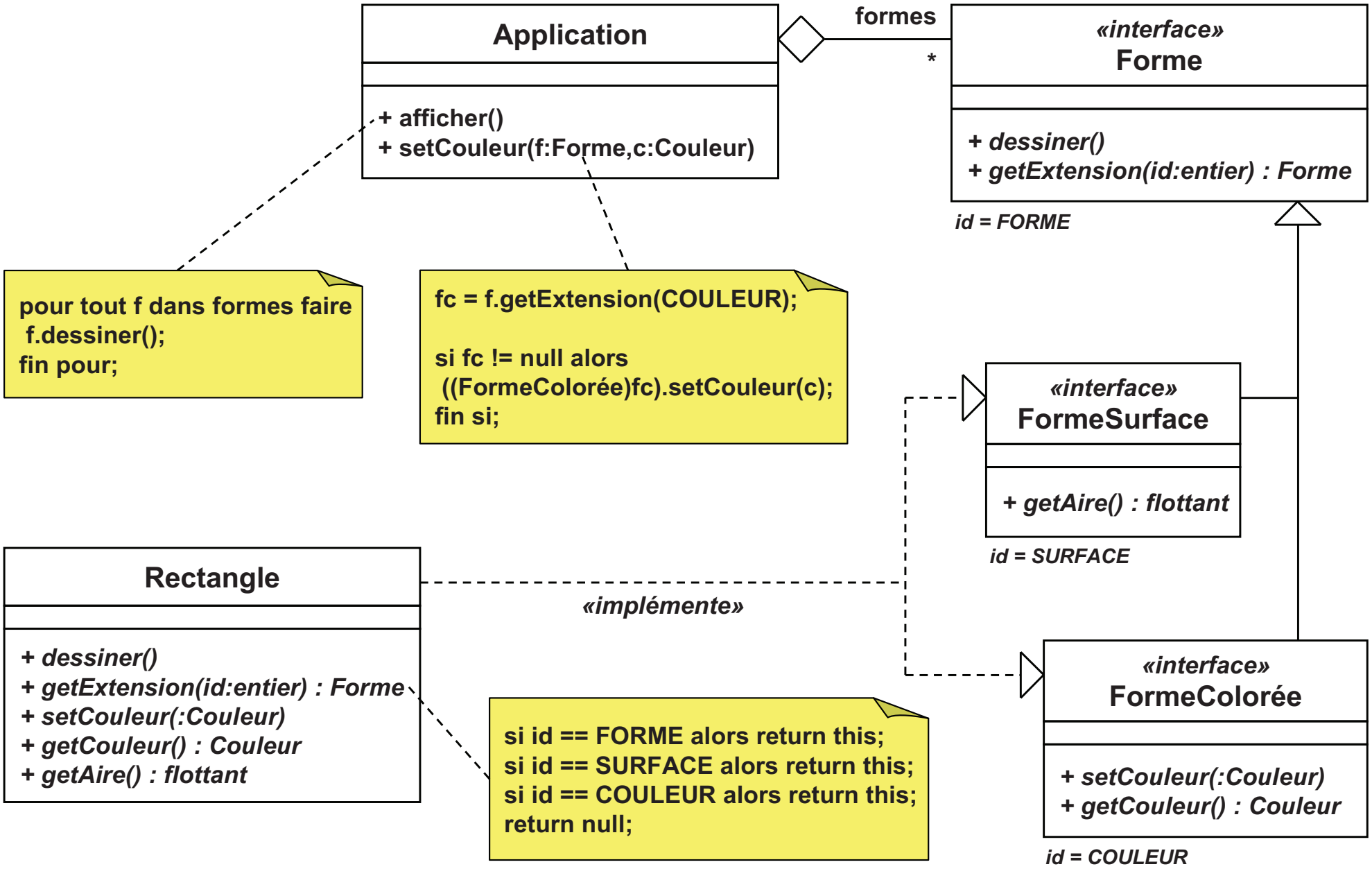
Interface d'extension / *Extension Interface (1/4)*

- Objectif
  - Permettre à un composant d'exporter plusieurs interfaces
  - Eviter le «gonflement» de son interface
    - Lors de l'ajout de nouvelles fonctionnalités
- Principe
  - Décomposer l'interface par intérêt
    - Une interface mère + des interfaces filles (une par intérêt)
  - Un composant agrège plusieurs interfaces
    - Plusieurs possibilités

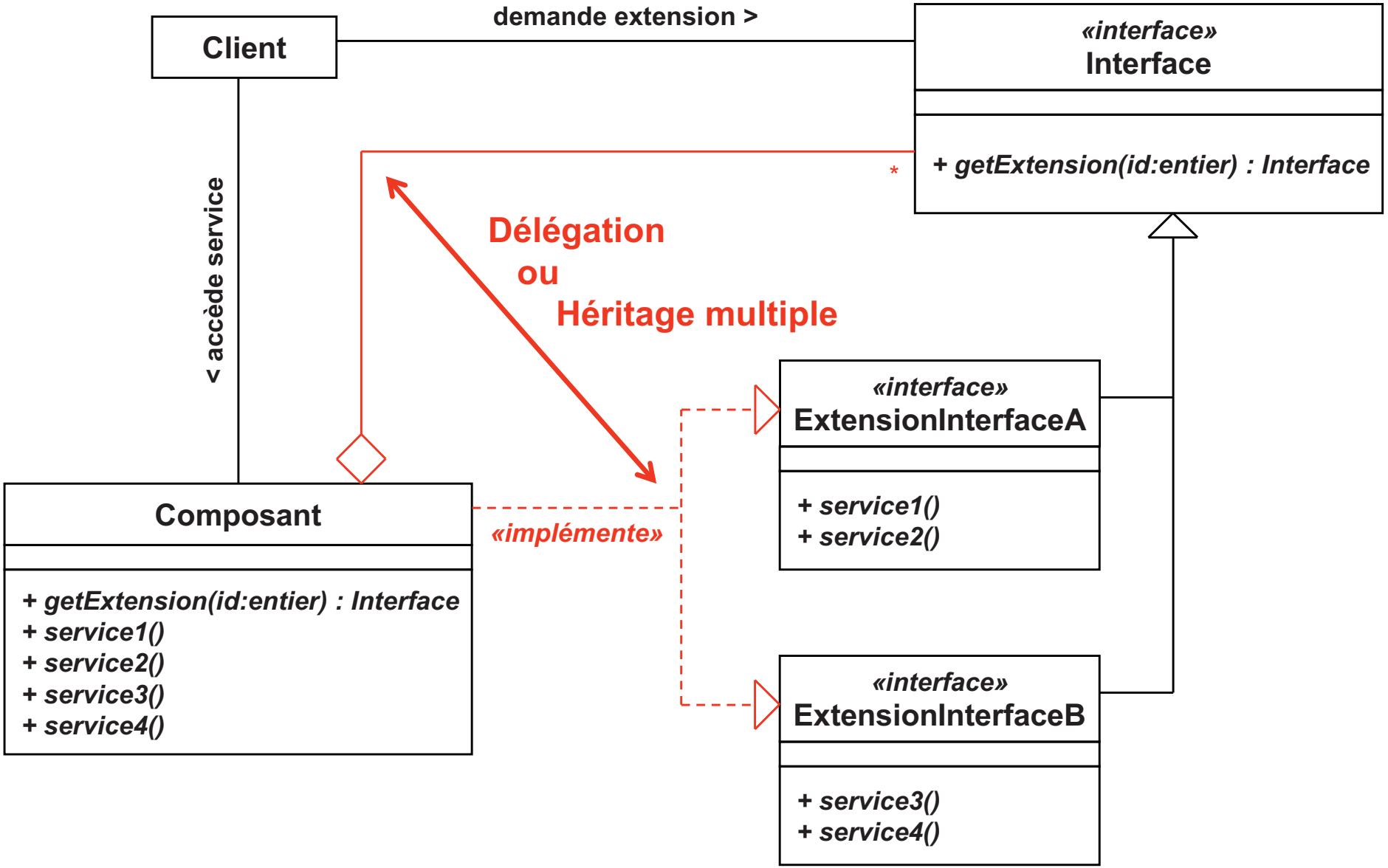
    ⇒ Implémentation multiple des interfaces

    ⇒ Agrégation d'interfaces et délégation
- Motivation
  - Proposer de nombreuses fonctionnalités sur des composants
  - Mais en évitant d'alourdir l'interface de tous les composants

Interface d'extension / *Extension Interface (2/4)*



Interface d'extension / *Extension Interface (3/4)*



Interface d'extension / *Extension Interface (4/4)*

- Intérêts
  - Extensibilité
    - Ajout de nouvelles fonctionnalités ⇒ Nouvelle interface
  - Séparation des intérêts
    - Une interface par thème
  - Attention au surcoût
    - Accès indirect au composant
- Relations avec d'autres patrons
  - Pont
    - Implémentation de la composition d'interfaces
  - Fabrique abstraite
    - Peut être utilisée pour la création de composants