Programmation objets, web et mobiles (JAVA)

Cours 7 – Design Patterns

Licence 3 Professionnelle - Multimédia

Philippe Esling (esling@ircam.fr)

Maître de conférences – UPMC

Equipe représentations musicales (IRCAM, Paris)

UPMC SORBONNE UNIVERSITÉS

ircam Centre Pompidou

Rappel des fondations de la POO

- **Abstraction**
  - Animal est abstrait. Zoo contient des animaux
- **Encapsulation**
  - Protection des attributs de l'objet
  - Contrôle des accès, isolation de l'implémentation
- **Polymorphisme**
  - Signature polymorphes, résolution des invocation
- **Héritage**
  - Redéfinition de comportement par héritage

Design patterns

- Solutions « **prototypiques** » à des problèmes objets
- **Réutilisable** à des problèmes récurrents
- Peu d'algorithmique, plus des schéma orientés-objet
- Façons d'organiser le code pour augmenter
  - Flexibilité
  - Maintenabilité
  - Extensibilité
  - Configurabilité
  - …
- Le plus souvent basé sur des **interfaces et abstractions**

Design patterns: principes

Principe 1 :

Favoriser la composition (liens dynamiques, flexibles) sur l'héritage (lien statique, peu flexible)

- La délégation est un exemple de composition
- Attention il s'agit juste de **favoriser** car l'héritage est également très utilisé dans les designs patterns

Principe 2:

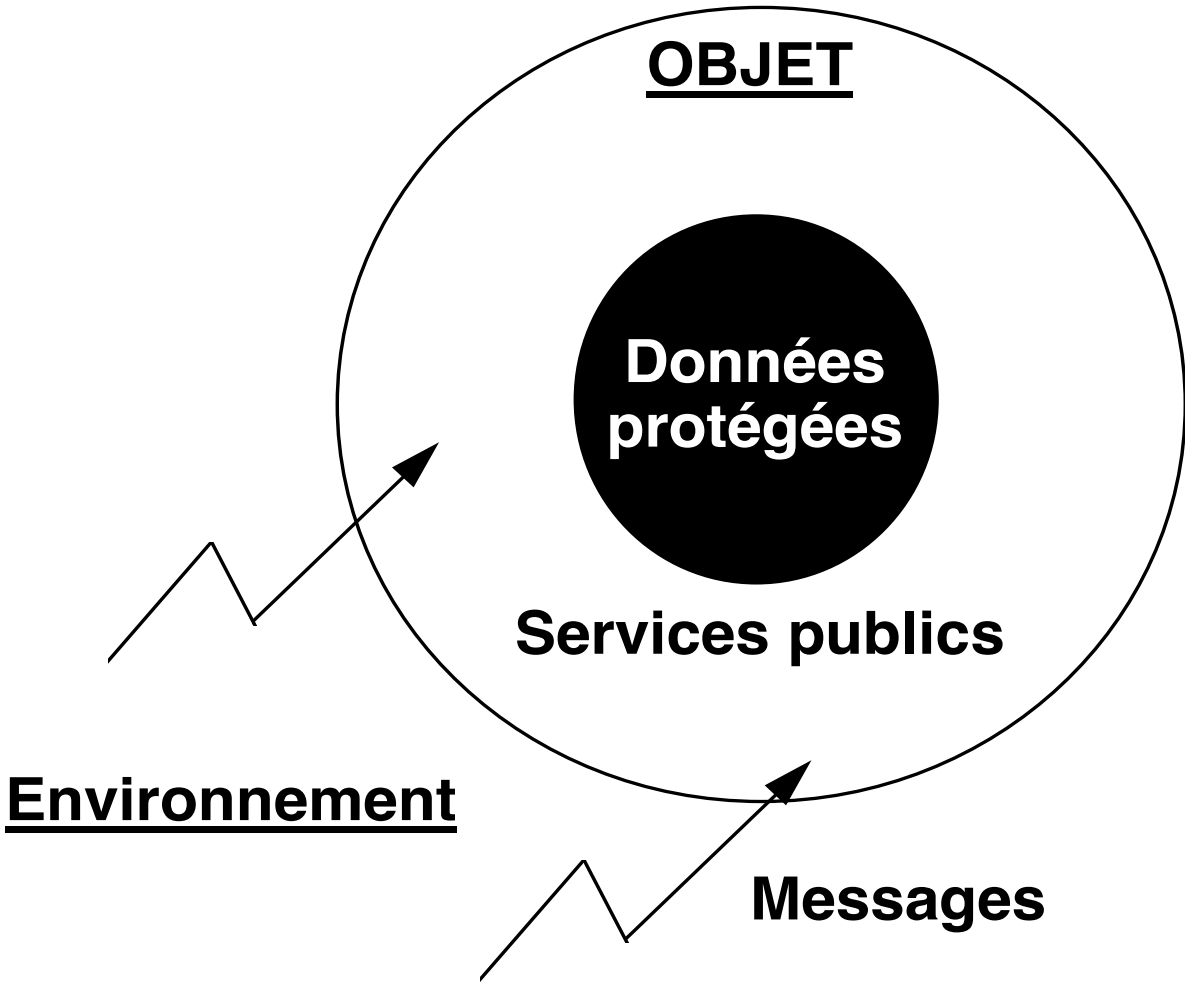
Les clients programment **en priorité pour des abstractions** (interfaces) plutôt qu'en lien avec les implémentations concrètes (classes)

Principe 3: Privilégier l'encapsulation forte

Intérêt des patterns

- Un vocabulaire commun et puissant
- Les patterns aident à concevoir facilement des systèmes
  - Réutilisables: Responsabilités isolées, dépendances maitrisées
  - Extensibles: Ouverts aux enrichissements futurs
  - Limiter la modification de l'existant
  - Maintenables par faible couplage
- Les patterns reflètent l'expérience de développeurs objets
  - Solutions éprouvées et solides
- Les patterns ne sont pas du code mais des cadres de solutions générales à adapter à son problème particulier
- Les patterns aident à maîtriser les changements
  - Les solutions plus triviales sont souvent moins extensibles
- Attention à **l'overkill** ! Utilisez les patterns intelligemment

Encapsulation niveau objet



Design patterns: classification

Patterns créateurs

Ciblent la construction des objets (« aider » **new, clone**)

- Patterns *Factory, AbstractFactory, Singleton …*

Patterns structuraux

Travaillent sur des *aspects statiques*, à « l'extérieur » des classes (notamment **extensibilité**)

- Patterns *Façade, Adapter, Decorator, Proxy, Composite …*

Patterns comportementaux

Travaillent sur des *aspects dynamiques*, à « l'intérieur » des classes (parfois même des instances)

- Patterns *Strategy, Iterator, Observer, Visitor*

Notre référence : **Formes**

- On va réaliser une application de dessin.
- Celle-ci se base sur des formes à utiliser
- Pour cela on définit l'interface *Forme* suivante

```
public interface Forme {
   public void translate (int dx, int dy);

   public void dessine (Graphics g);
}
```

Formes : *Carré*

```
public class Carre implements Forme {
  private int x;
  private int y;
  private int cote;
  public Carre(int x, int y, int cote) {
    this.x = x;
    this.y = y;
    this.cote = cote;
  }
  public void translate(int dx, int dy) {
    x += dx;
    y += dy;
  }
  public void dessine(Graphics g) {
    g.drawRect(x, y, cote, cote);
```

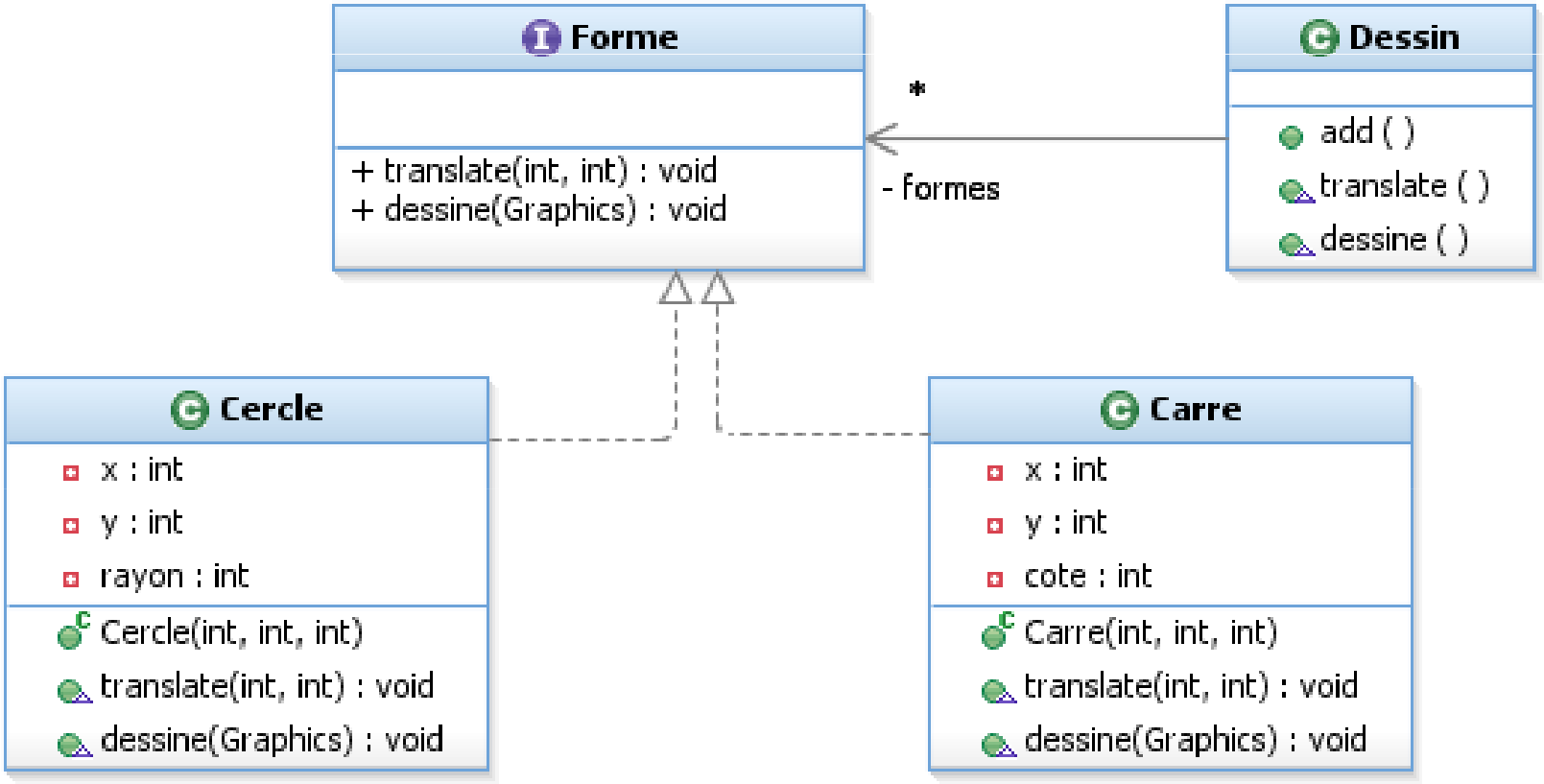
Formes : *Cercle*

```
public class Cercle implements Forme {
  private int x;
  private int y;
  private int rayon;
  public Cercle(int x, int y, int rayon) {
    this.x = x;
    this.y = y;
    this.rayon = rayon;
  }
  public void translate(int dx, int dy) {
    x += dx;
    y += dy;
  }
  public void dessine(Graphics g) {
    g.drawOval(x, y, rayon, rayon);
```

Dessin = Liste de formes

```
public class Dessin {
    List<Forme> formes = new ArrayList<Forme>();

    public void add (Forme f) {
        formes.add(f);
    }
    public void translate(int dx, int dy) {
        for (Forme f : formes)
   f.translate(dx, dy);
    }
    public void dessine(Graphics g) {
        for (Forme f : formes)
   f.dessine(g);
    }
}
```

Diagramme de classes



Mélanger des formes ?

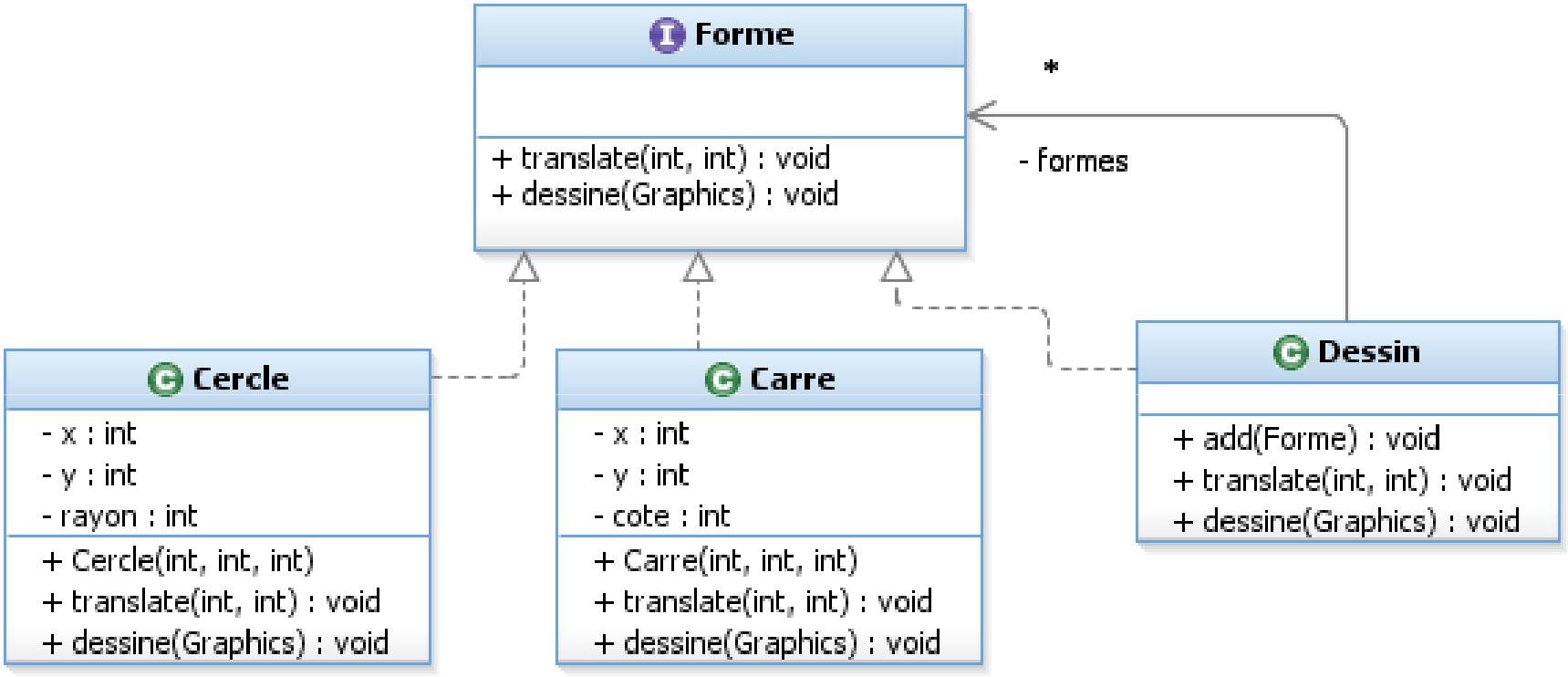
- Pour étendre les possibilités de dessin, on pense aux mélanges
- Comment créer un CarréCercleConcentrique (carré contenant et cercle et contenu dans un autre) ?
- **Option 1 :** Implémentation directe (x,y) et longueur
  - Forte redondance dans le code
- **Option 2 :** Représenter par un *dessin lui-même !*
  - Toutes formes est la composition de formes de base
  - On peut donc représenter par une liste de formes (ie. Dessin)
  - Ici, contiendra un carré et deux cercles …
  - **Mais nécessite que *dessin soit aussi une forme***

- L'option 2 est la définition correspondant au **pattern Composite**

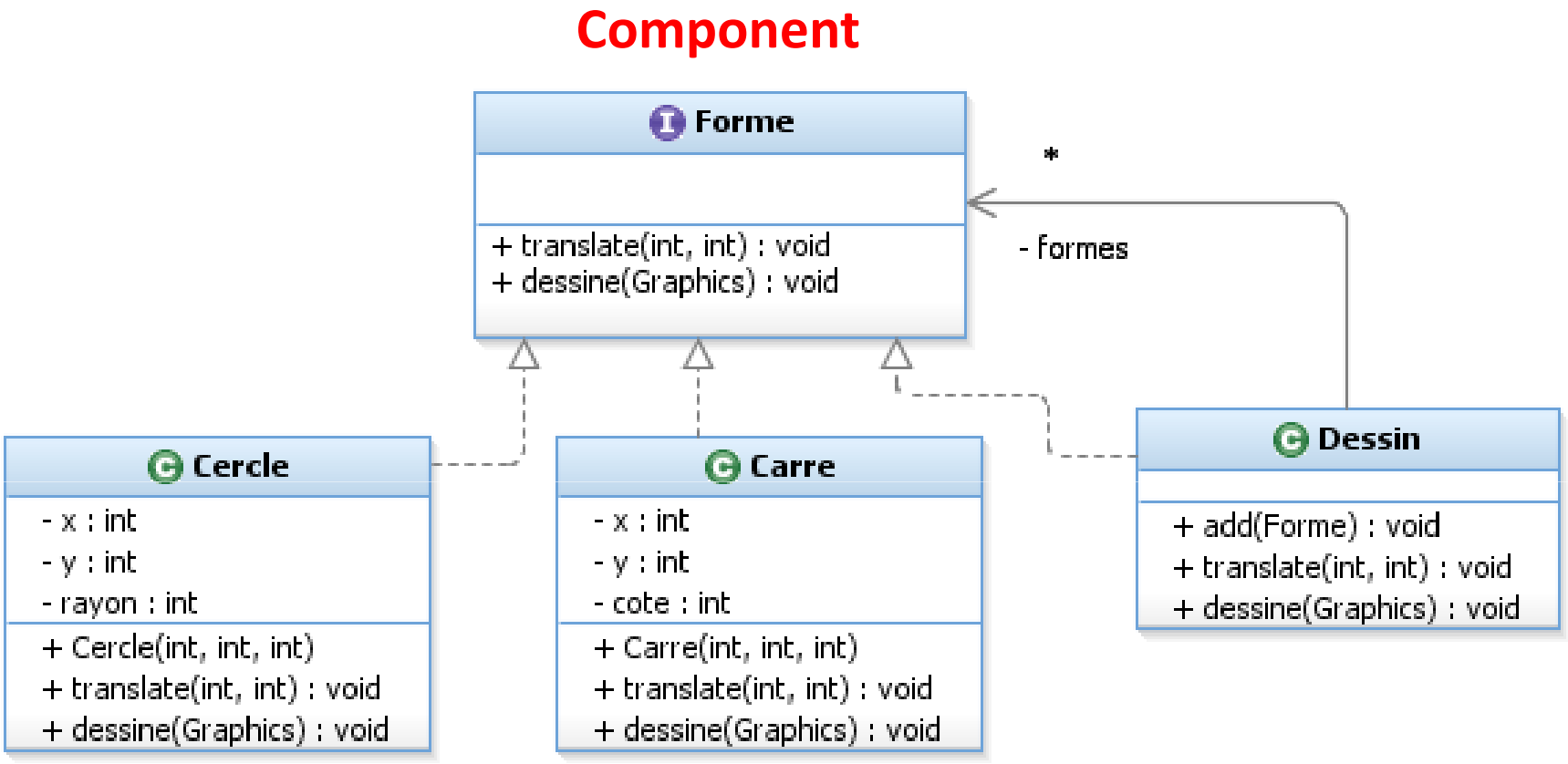
Pattern Composite

- **Objectif** : traiter des structures arborescentes
  - Arbres de syntaxe
  - Expression arithmétiques
  - Arborescence de fichiers
- **Structure**
  - *Component*, un nœud de l'arbre quelconque abstrait
  - *Leaf*, une feuille de l'arbre qui n'a pas de fils
  - *Composite*, un nœud ayant des fils *Component*

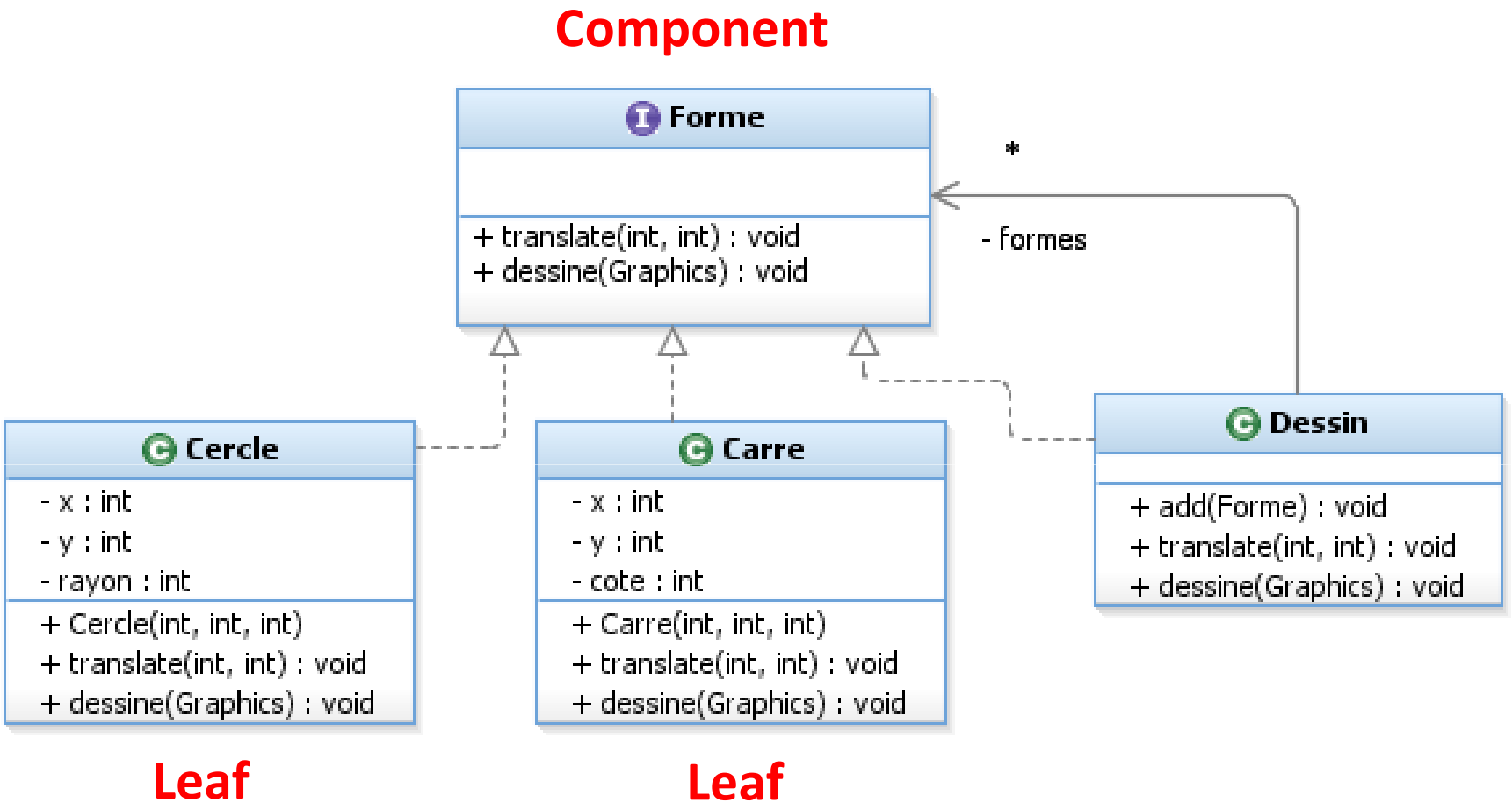
Pattern Composite: Formes



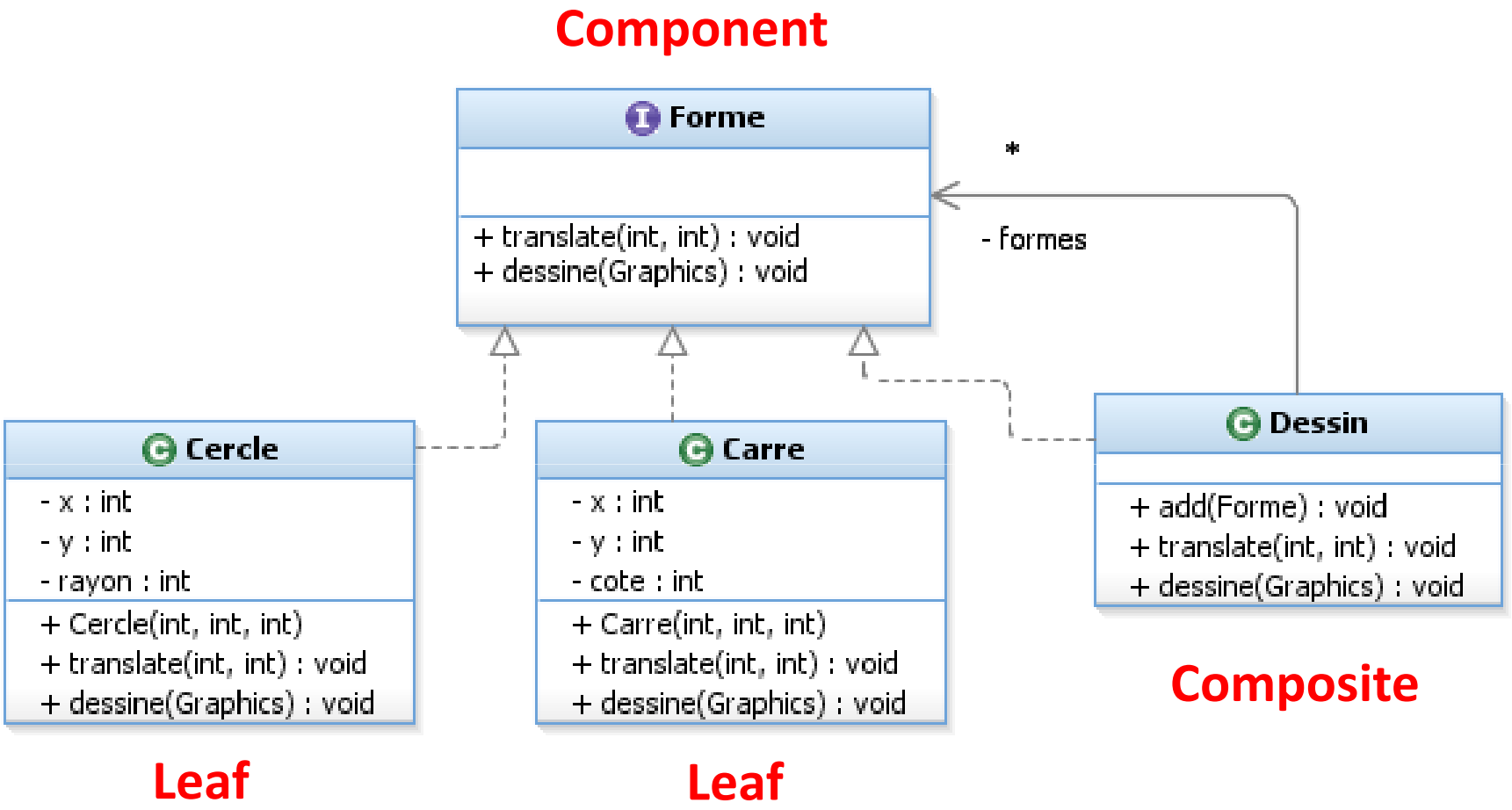
Pattern Composite: Formes



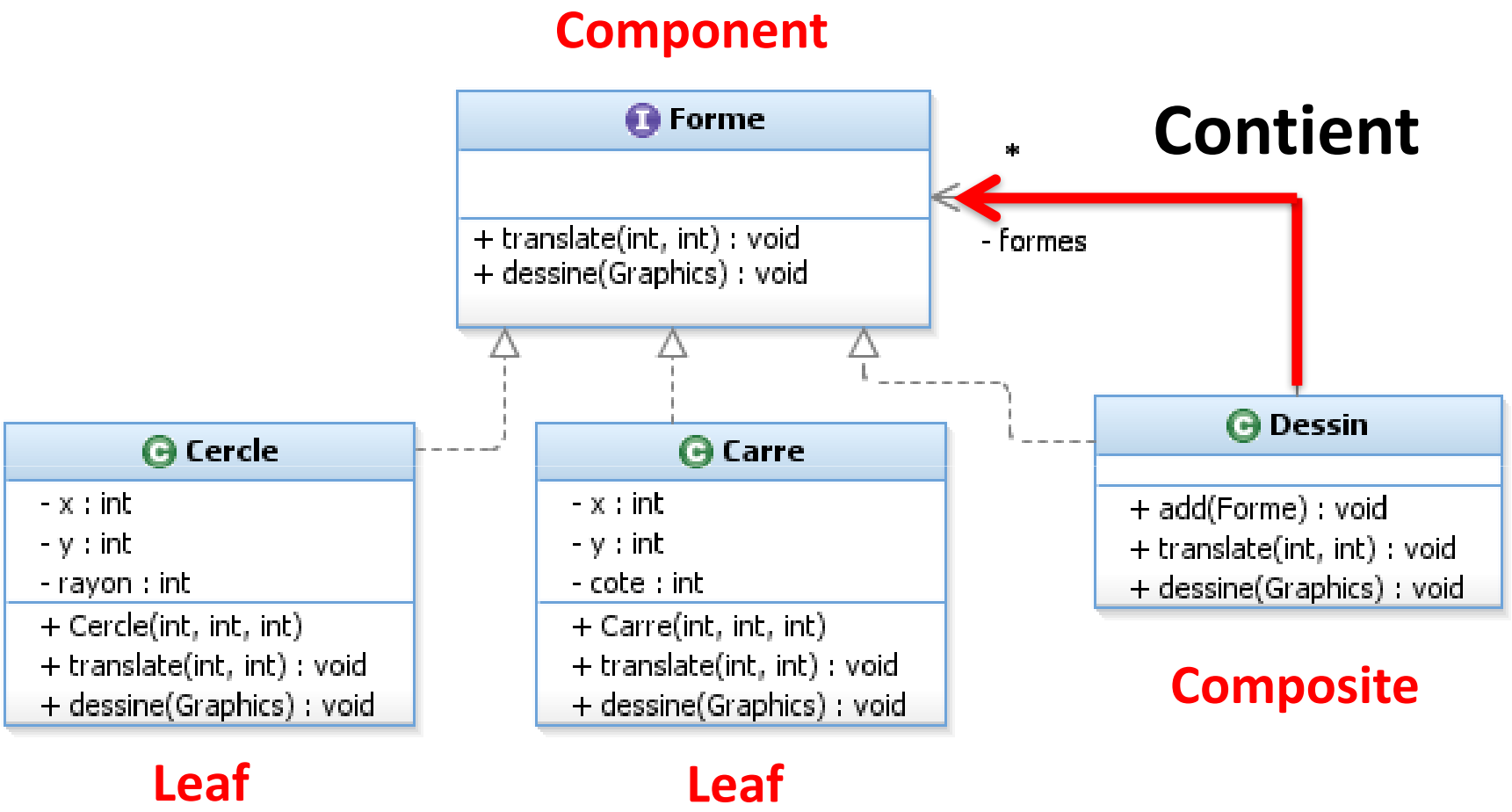
Pattern Composite: Formes



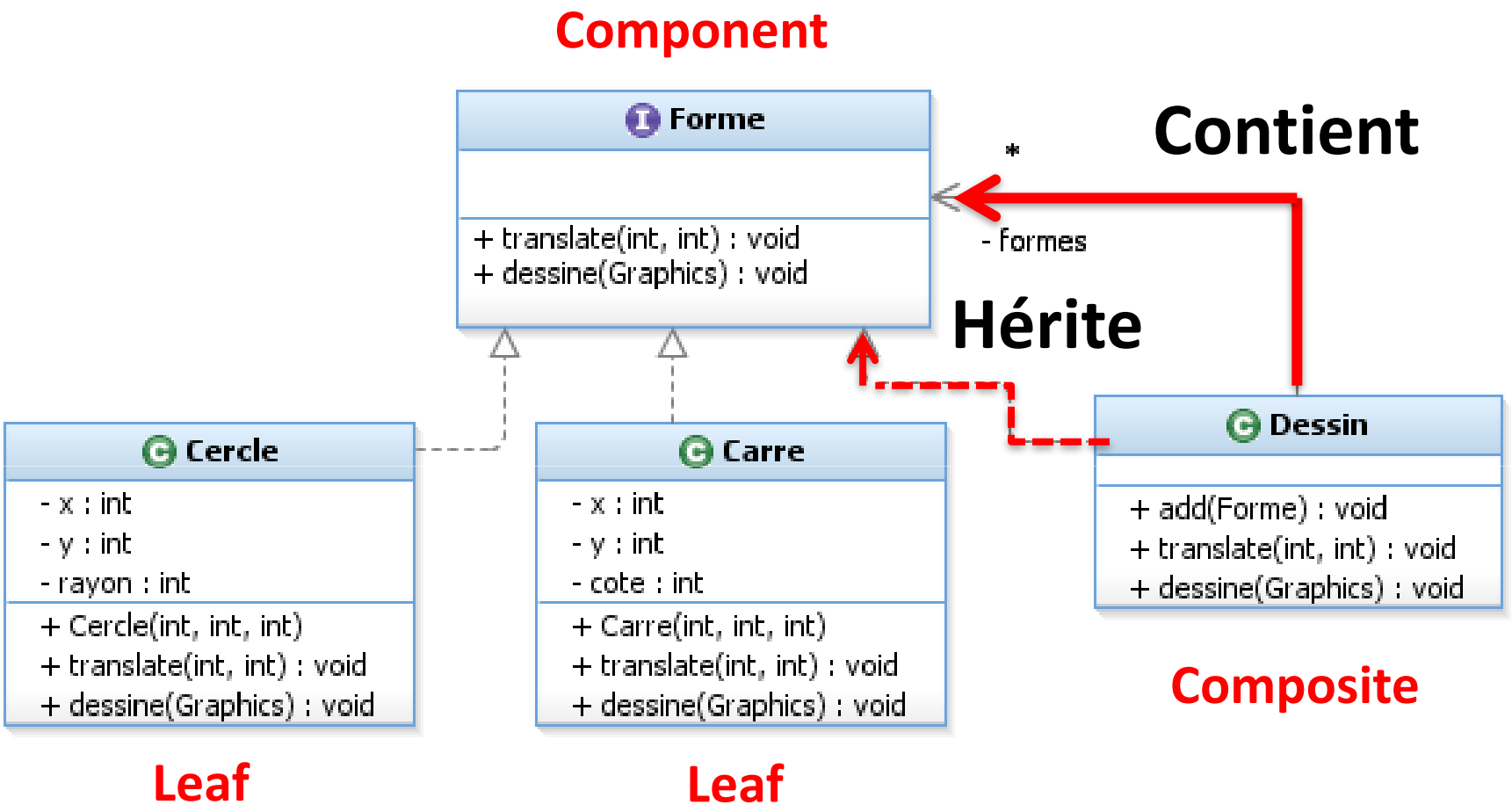
Pattern Composite: Formes



Pattern Composite: Formes



Pattern Composite: Formes



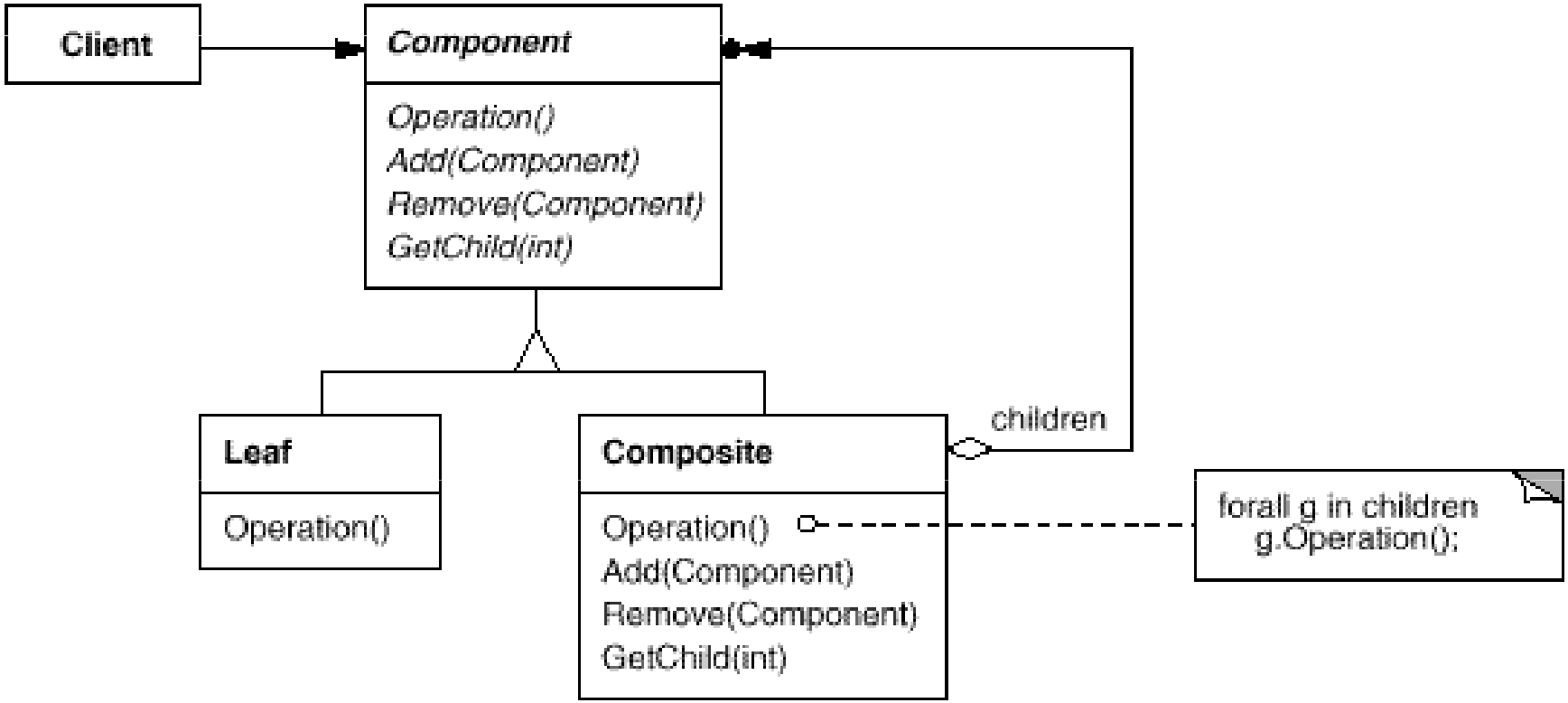
Pattern Composite: Formes

- **public class** Dessin **implements** Forme
- Construire notre CarreCercleConcentrique

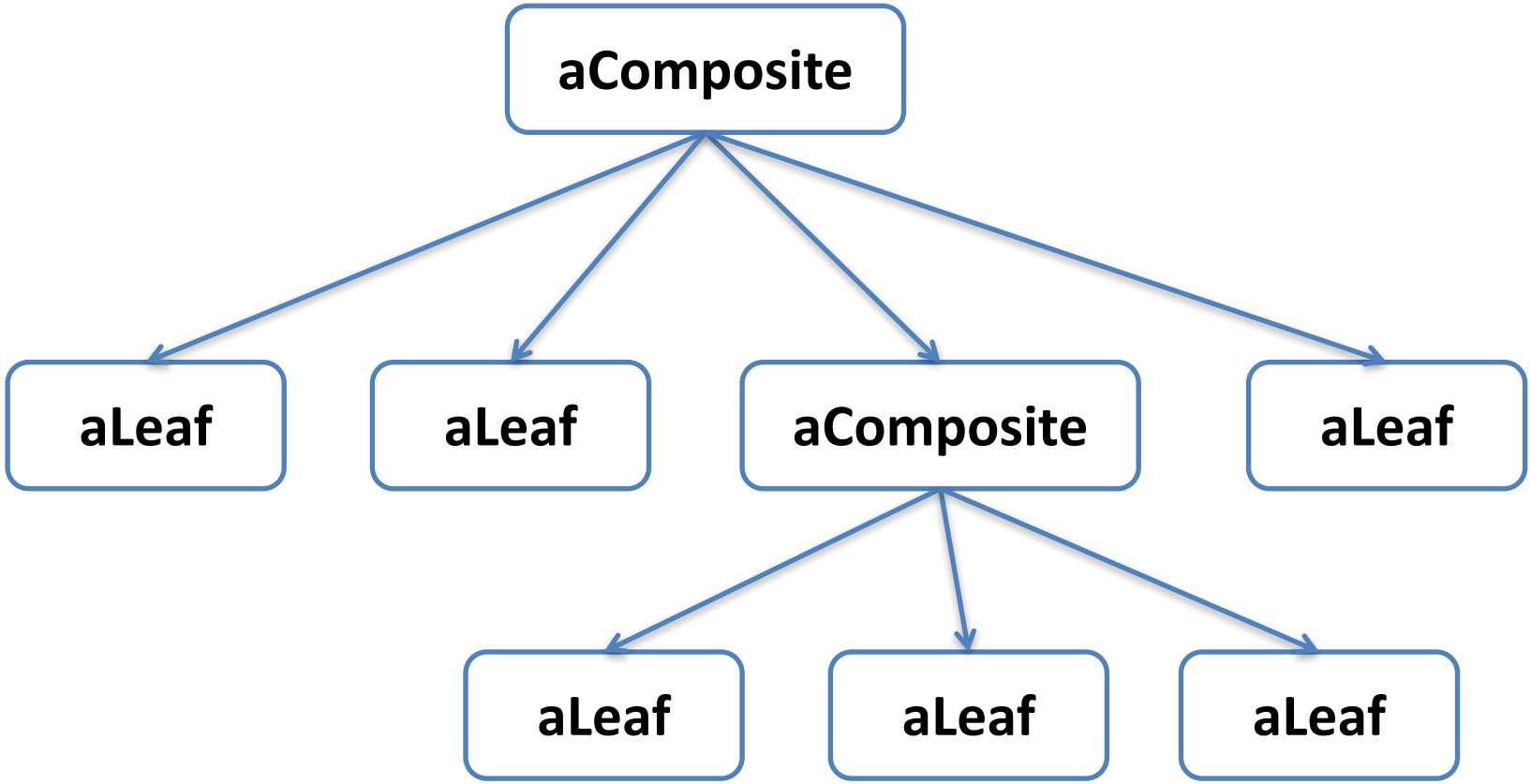
```
Dessin d = new Dessin();
d.add(new Carre(x,y,width));
d.add(new Cercle(x+width/2, y + width/2, width/2));
return d;
```

- **Aucune limite sur la composition ☺**
- Ici un Dessin peut être la composition …
- … d'un ensemble de dessins

Pattern Composite générique



Pattern Composite *instancié*



Intérêt et subtilité

- Permet de manipuler les Composite et les Feuilles de manière homogène et transparente
- Permet de récursivement supporter que les fils d'un Composite soient eux-mêmes Composites
  - Les opérandes d'une Expression sont des Expression
  - (exemple mathématique)
- Double lien Composite – Component. Un composite est un Component **et** référence des Component
- Fonctions de manipulation déclarées sur Component
  - Une feuille n'a pas de fils
  - Evite au client de connaître les classes Composite

Mini-exercice Composite

- Créer un ensemble de classes de formes
- Ajouter un Design Pattern Composite
  - Créer une classe composite
  - Gérer l'architecture des classes
  - Gérer les fonctions de déplacement et dessin
- Tester votre composite en créant des composites imbriqués
- Créer une interface graphique Swing de dessin
- Deux Jpanels: un pour le *dessin* et l'autre pour le *contrôle*
- Le panel de contrôle permet
  - Ajouter des formes simples (à dessiner)
  - Une liste de formes permet de contrôler celles déjà ajoutées
  - Un sous-panel permet de créer un composite (ajout itératif)

Pattern Proxy

- Proxy : objet faisant semblant d'être un autre objet
- Par exemple le *proxy réseau* de votre browser
  - Se comporte comme un gateway internet (box)
  - Mais rajoute des traitements (filtres, cache, …)
- Pour la Pattern, le proxy est une classe qui **implémente les même opérations que l'objet** qu'elle protège/contrôle.
- Plusieurs variantes de Proxy suivant l'usage
  - Proxy Virtuel: retarde les allocations/calculs couteux
  - Proxy de Sécurité: filtre/contrôle les accès à un objet
  - Proxy Distant: objet local se comportant comme le distant (et donc masque le réseau)
  - Smart Reference: proxy qui compte les références (GC)

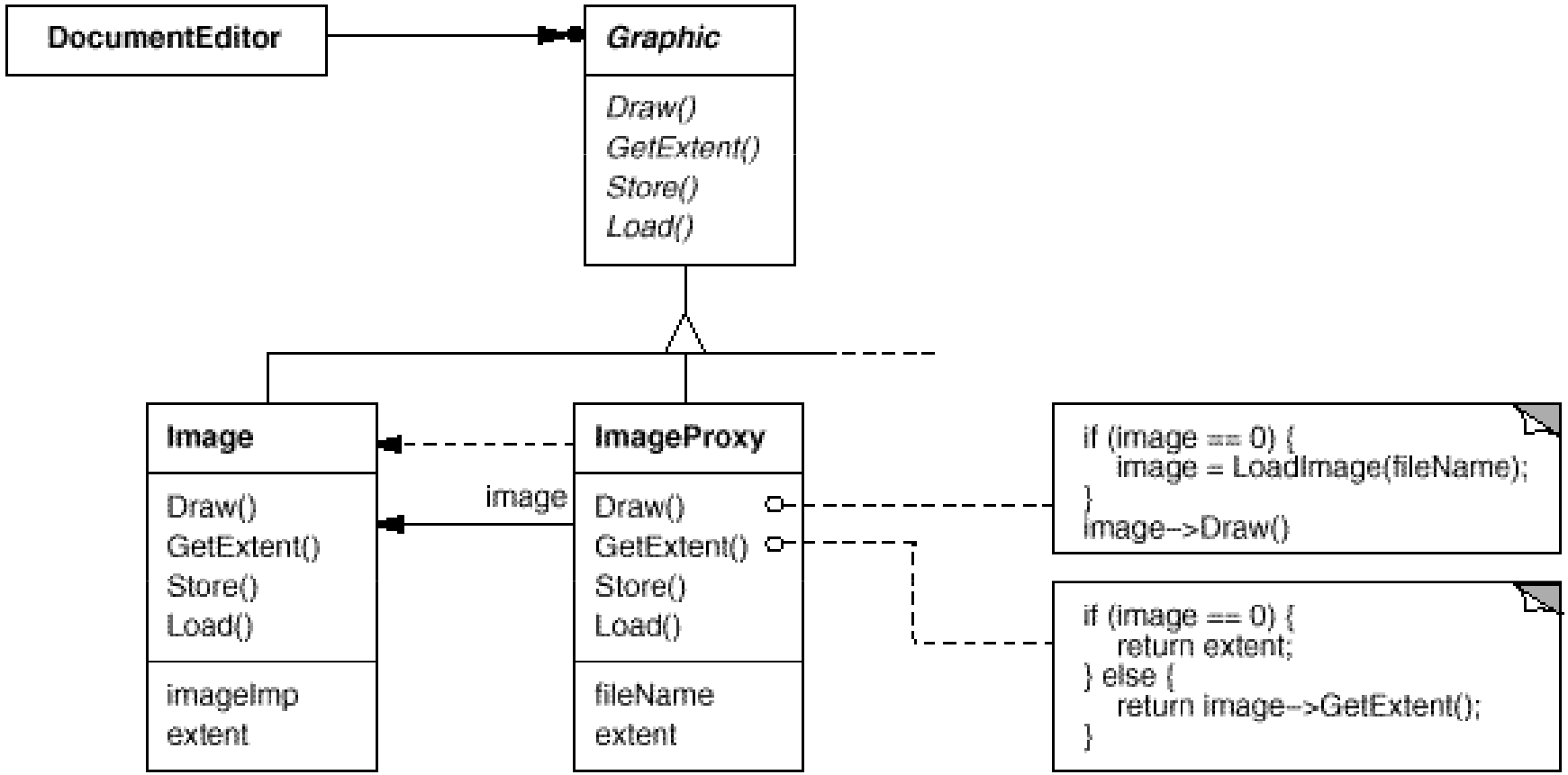
Pattern Proxy Virtuel

- Permet de **retarder les opérations coûteuses**
- Par exemple dans un éditeur de texte type Word
- Le document est rempli d'images « lourdes »
- Celles-ci sont stockées dans des fichiers séparés

- Quand on ouvre un document, il faut calculer la mise en page
- … Et donc la taille des images
- … Et donc faire le rendu des images présentes partout
- … Quelle lenteur !

- Comment retarder le chargement des images?

Proxy virtuel : Example

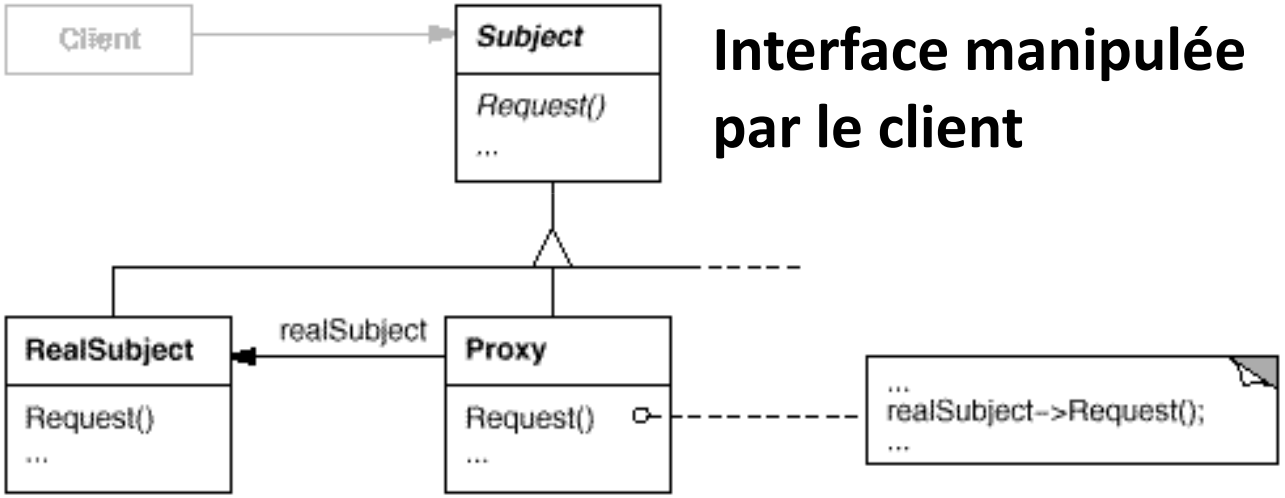


Proxy Virtuel : Principes

- On **construit initialement des ImageProxy pour chaque image**
  - Par exemple le document le fait via une ImageFactory

- Ces objets stockent et connaissent la taille de l'image
- Ce n'est que lorsqu'on affiche la page avec l'image (correspond à **la prèmière invocation de draw** sur le proxy) que l'image va être chargée (à la volée).
- Conclusion: le document s'ouvre rapidement et en plus le **mécanisme est transparent** pour l'utilisateur !
- **Impossible de distinguer le Proxy de l'objet réel** !

Pattern Proxy



Interface manipulée par le client

Objet lourd à instancier

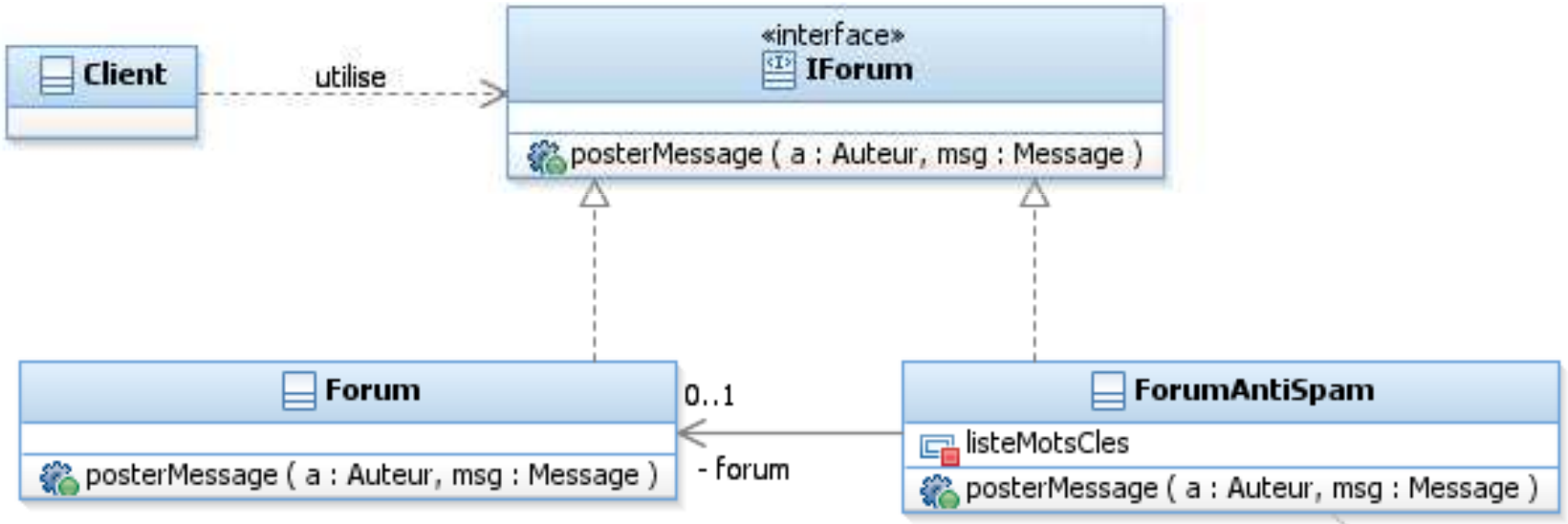
Retarde la création du sujet réel

Délégation particulière ou délégat (Proxy) et délégué (RealSubject) réalisent la même interface

Proxy de sécurité

- Permet de protéger ou contrôles les accès à un objet
- Exemple Forum de discussion
  - Classe Forum: munie d'une opération de post
    - posterUnMessage(Auteur a, Message m)
  - La classe Forum existe, il s'agit de ne pas la modifier
- Comment bloquer des messages indésirables
- Contenant des mots clés interdits (langage SMS, Bieber)
- Cette fois un **proxy de sécurité**
- Ceci permet d'être **orthogonal au traitement protégé**
  - La sécurité est une couche supplémentaire
  - Distincte du traitement de base …
  - Mais transparente pour l'utilisateur !

Proxy de sécurité forum



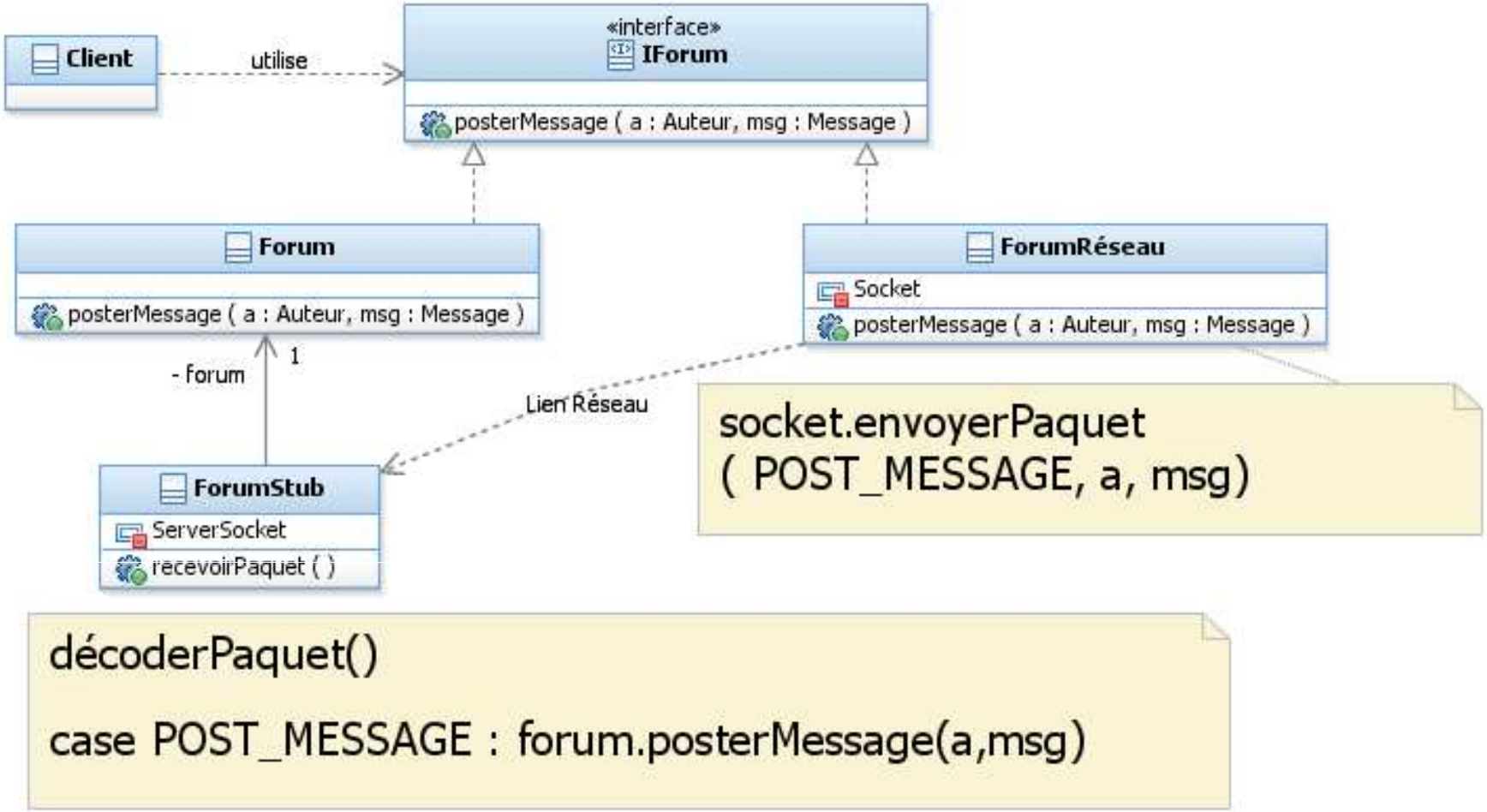

```
for (String mot : listeMotsCles)

    if (msg.contains(mot))
    return;

forum.posterMessage(a,msg);
```

Proxy distant

- On a une application répartie sur plusieurs machines
- On voudrait développer l'application sans trop se soucier de l'endroit ou sont stockés physiquement les objets
- Proxy réseau
  - objet local à la machine
  - Se comporte comme l'objet distant
  - Répercute ses opérations sur l'objet distant via réseau

- **Comportement par délégation …** mais avec le réseau interposé ☺

Proxy distant



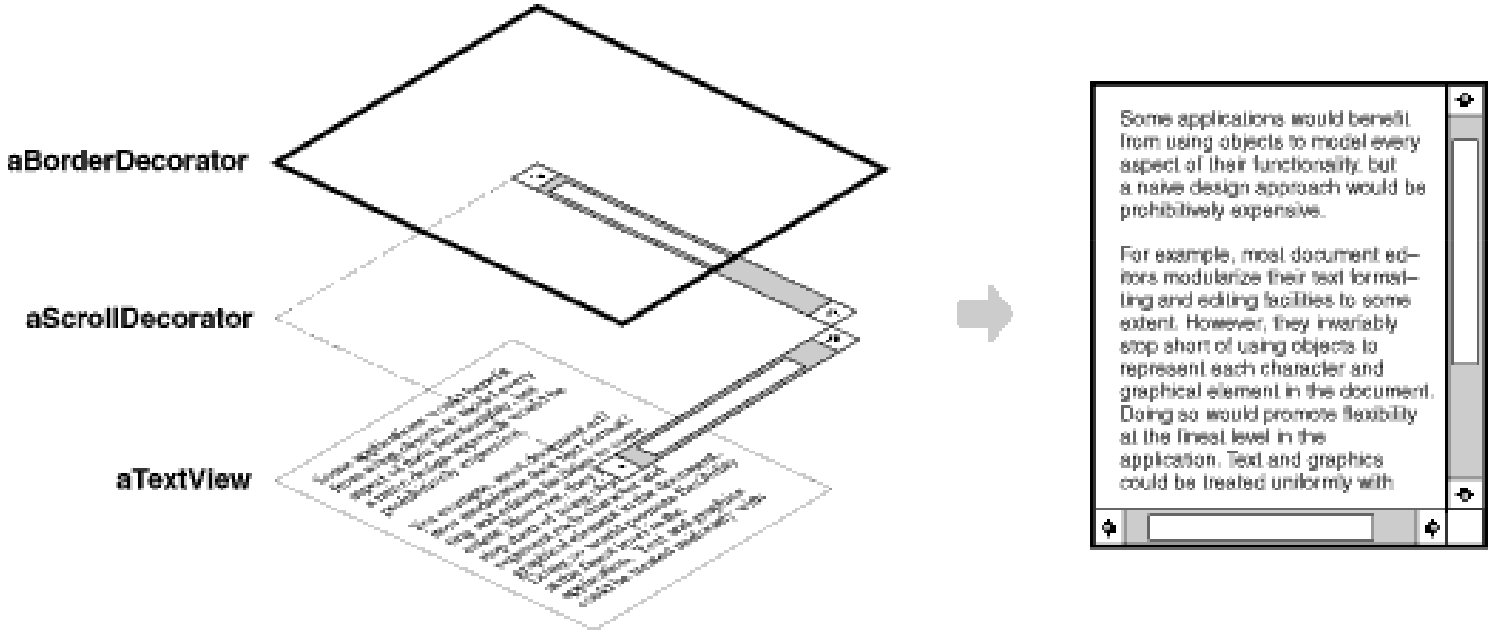
Proxy distant

- Généralise la notion de *Remote Procedure Call* (RPC)
- Rends transparent la localisation des objets
- Réalisation du Proxy réseau et du stub suit une ligne standard
- De nombreux frameworks offrent de générer cette glue
- Permettent également de la cacher ou non
- Eg. Java RMI (Remote Method Invocation)

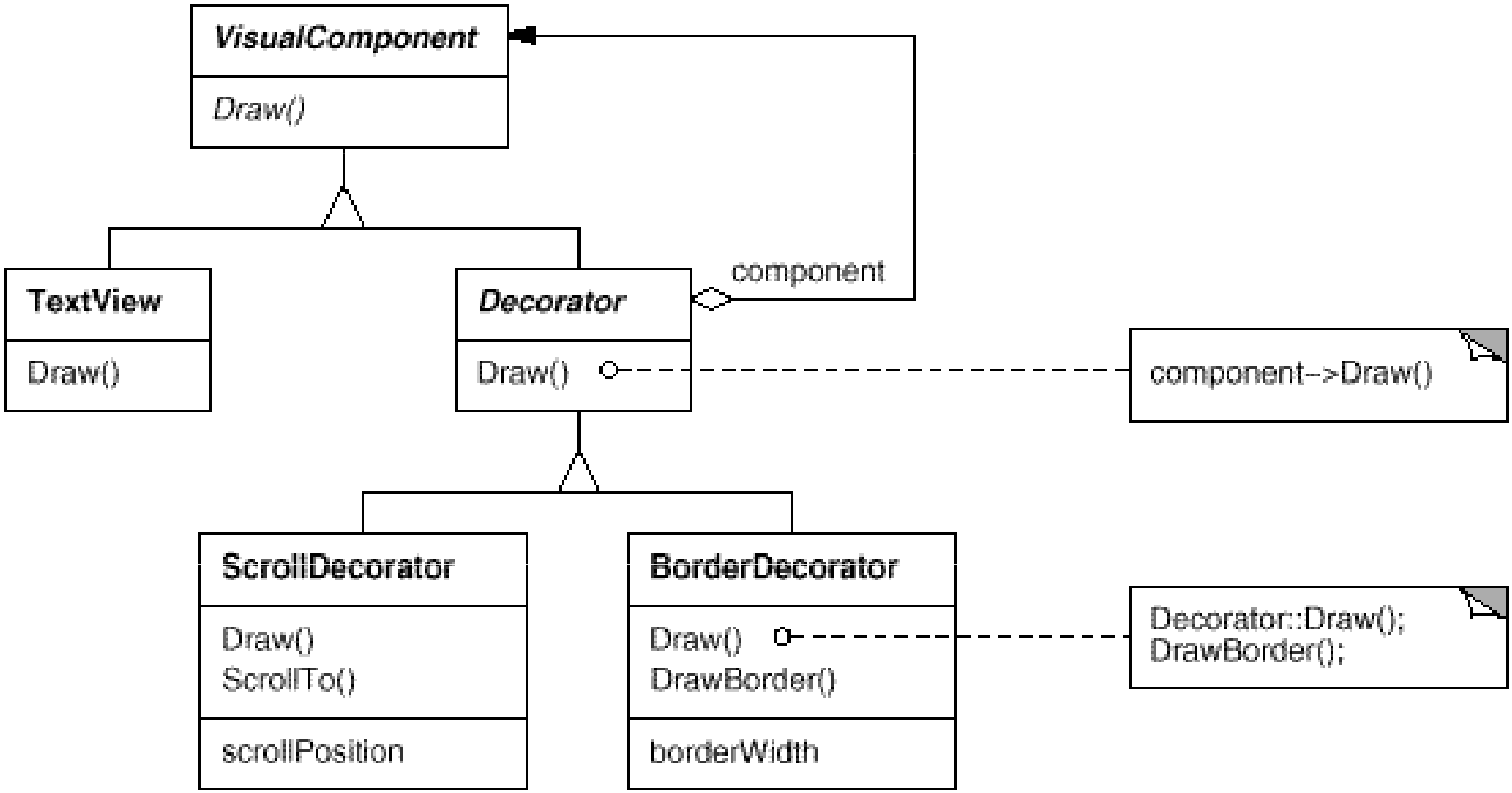
Pattern Decorator

- Ajouter dynamiquement des caractéristiques ou des responsabilités à un objet
- L'objet décoré se manipule comme l'objet de base
- Les décorateurs doivent pouvoir s'empiler

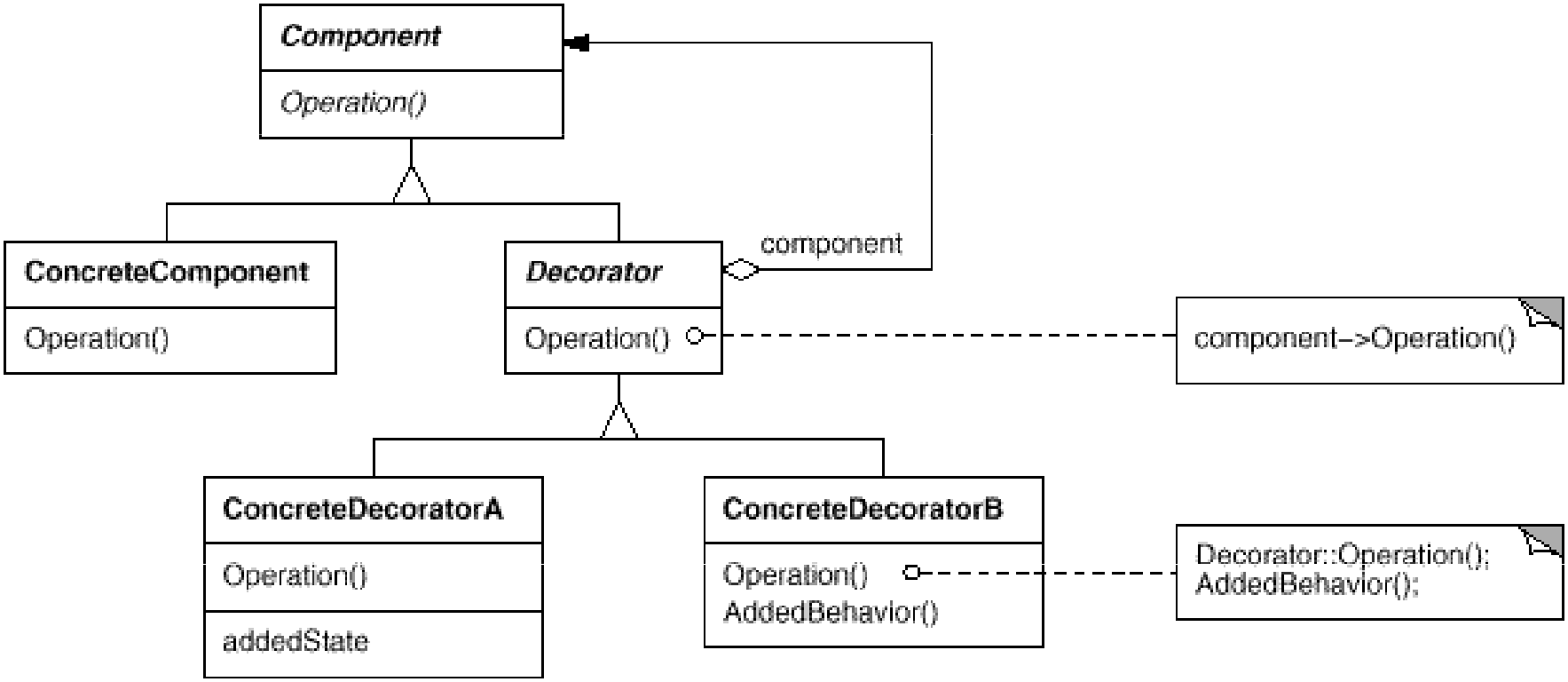
Eg. Border(Scroll(Texte))



Pattern Decorator: exemple



Pattern Decorator: structure



Mini-exercice Decorator

- Créer un Design Pattern Decorator à deux variantes
- Commencer par implémenter le schéma global
- On veut ajouter des couleurs aux formes
  - Decorator de forme avec une couleur
- On veut positionner la taille et le style du trait
  - Decorator de Forme avec lineWidth, lineStyle
  - Implémentation similaire à la précédente
- On peut également combiner ces Decorator

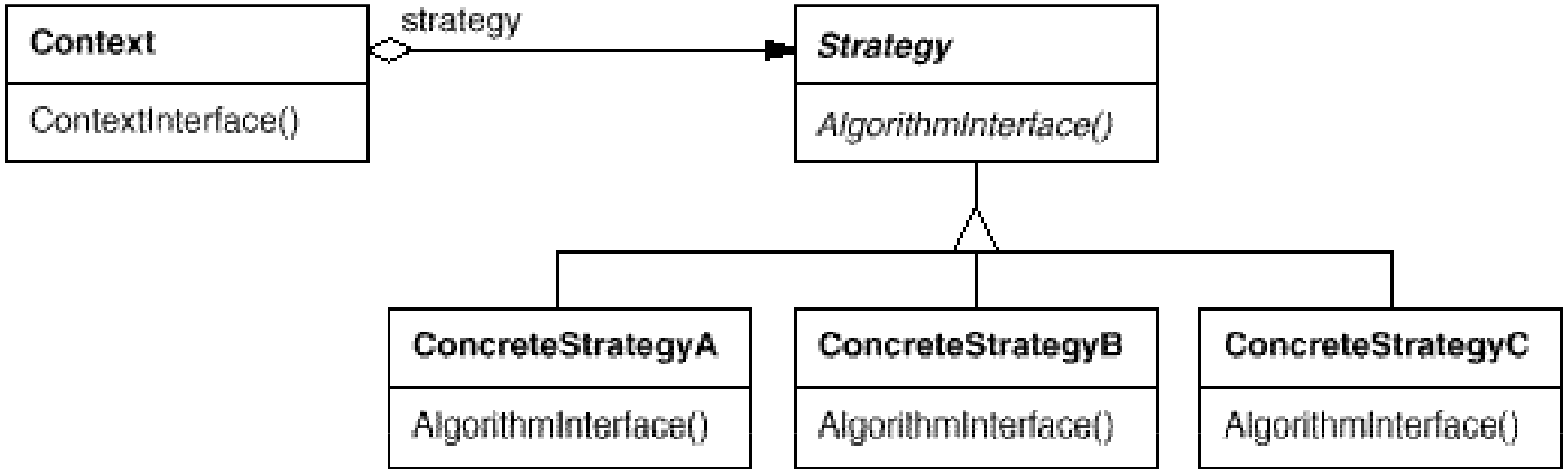
Decorator: conclusions

- Permet de combiner les traitement ajoutés
  - Evite l'explosion du nombre de classes
  - Même si on combine les traitements par héritage
- Proche de Proxy mais …
  - Proxy connaît la réalisation particulière
  - Decorator s'appuie uniquement sur l'abstraction
- Proche de Composite, mais
  - Composite s'intéresse aux fils du nœud composite
  - Ici on rend transparent l'ajout de fonctionnalités

Pattern Strategy

- Exemple de robots dans un monde = matrice de cases
  - Robot.move()
  - Robot.action()
  - Des robots paresseux
  - Des robots pollueur/nettoyeur
  - Des robots avec différents types de comportements
- Héritage
  - RobotPollueurParesseuxRandom, RobotNettoyeurBosseur, etc…
  - Même en considérant l'héritage multiple, reste difficile
  - Impossible de modifier le comportement d'un robot au runtime
  - Choix de comportement au moment du new !

Pattern Strategy: structure



Strategy: exemples d'utilisation

- Plusieurs classes ne **différent que par leur comportement**
  - Ex. robots pollueurs
  - Les stratégies peuvent former leur propre arbre d'héritage
- **Variante d'algorithmes**, mémoire vs. Temps
  - Ex. Chercher un objet dans une collection
- L'algorithme nécessite des **structures complexes** qui polluent le code principal de la classe hôte
- Votre classe peut avoir plusieurs comportements exprimés comme des branchements (switch, case, etc…)
  - Strategy capture chaque branche dans une unité
- Le **comportement d'un objet évolue** au cours de sa vie

Strategy: combinaisons

- Le pattern Strategy peut très bien s'employer en complément d'autres design patterns
- Factory/Strategy: Une Factory dont on définit le comportement en leur donnant une stratégie
- Decorator/Strategy: les stratégies sont elles-même décorée
- Proxy Sécurité/Strategy: On définit la politique de sécurité appliquée par le Proxy au cours du temps

Mini-exercice Strategy

- Création d'une Intelligence Artificielle **très** basique
- Robots dans un monde = matrice de cases
- Mise en place d'un Design Pattern strategy
- Les robots ont des points de vie, look(), move() et action()
- Trois comportements (d'action) sont implémentés
  - Aggressif : Cherche à attaquer les robots proches
  - Neutre : Se déplace de manière aléatoire
  - Peureux : Cherche à éviter les robots proches
- + Trois comportements (de déplacements)
- Création aléatoire de robots puis boucle de jeu
  - A chaque boucle, les robots se déplacent ou attaque (comportement)
  - Le comportement change en fonction des points de vie.
  - Si PV < 50: Tout robot devient peureux
  - Si PV < 20: **Rage mode**, tout robot devient agressif
- **Bonus:** Afficher l'évolution grâce à une interface Swing

Séance **Kick-starter Projet**

- Etablissement de l'architecture orientée objet
  - Packages ?
  - Héritage de classes ?
  - Diagramme UML ?
- Réflexion sur les **designs patterns** éventuels
- Etablissement des **milestones** du projet
- Fonctionnalités des différentes classes
- Estimation temporelle correspondante
- Répartition des tâches

Programmation objets, web et mobiles (JAVA)

Cours 8 – Design Patterns II

Licence 3 Professionnelle - Multimédia

Philippe Esling (esling@ircam.fr)

Maître de conférences – UPMC

Equipe représentations musicales (IRCAM, Paris)

Design patterns créationnels

- **Objectif**: créer les objets de façon configurable

- *SimpleFactory (Factory* statique)
  - Classe responsable de créer des occurrences d'abstractions
- *Abstract Factory*
  - Permet de positionner une famille d'objets pour configurer un système
- *Factory Method*
  - Isole des traitements communs pour un type de produit abstrait
- *Singleton*
  - Permet d'assurer qu'une classe n'est instanciée qu'une fois
- *Prototype*
  - Factory configurable via une instance qui sera clonée

Pattern Factory

Le pattern Factory permet d'**isoler les créations d'objets**

Par exemple, un client doit pour créer une Forme appeler

- Forme f = **new** Carre(10, 12, 20)

Le problème est qu'il doit donc connaître Carré, Cercle,…

- Evolution difficile: impossible de supprimer Carré pour mettre Rectangle
- Comment gérer le passage à un Composite ?

- Donc le client dépend des classes concrètes **car il est forcé d'appeler leur construction par *new***
- On peut retirer ces dépendances, en codant une classe qui **se charge de construire les objets**

Pattern Factory: Formes

```
public class FormeFactory {

  public static Forme createCarreCercleConcentrique
(int x, int y, int width) {
    Dessin d = new Dessin();
    d.add(new Carre(x,y,width));
    d.add(new Cercle(x+width/2, y + width/2,
width/2));
    return d;
  }
  public static Forme createCarre (int x, int y, int
width) {
    return new Carre(x,y,width);
  }
  …
}
```

Pattern Factory

Le client avant:

- Forme f = **new** Carre(10, 12, 20)

A présent on a:

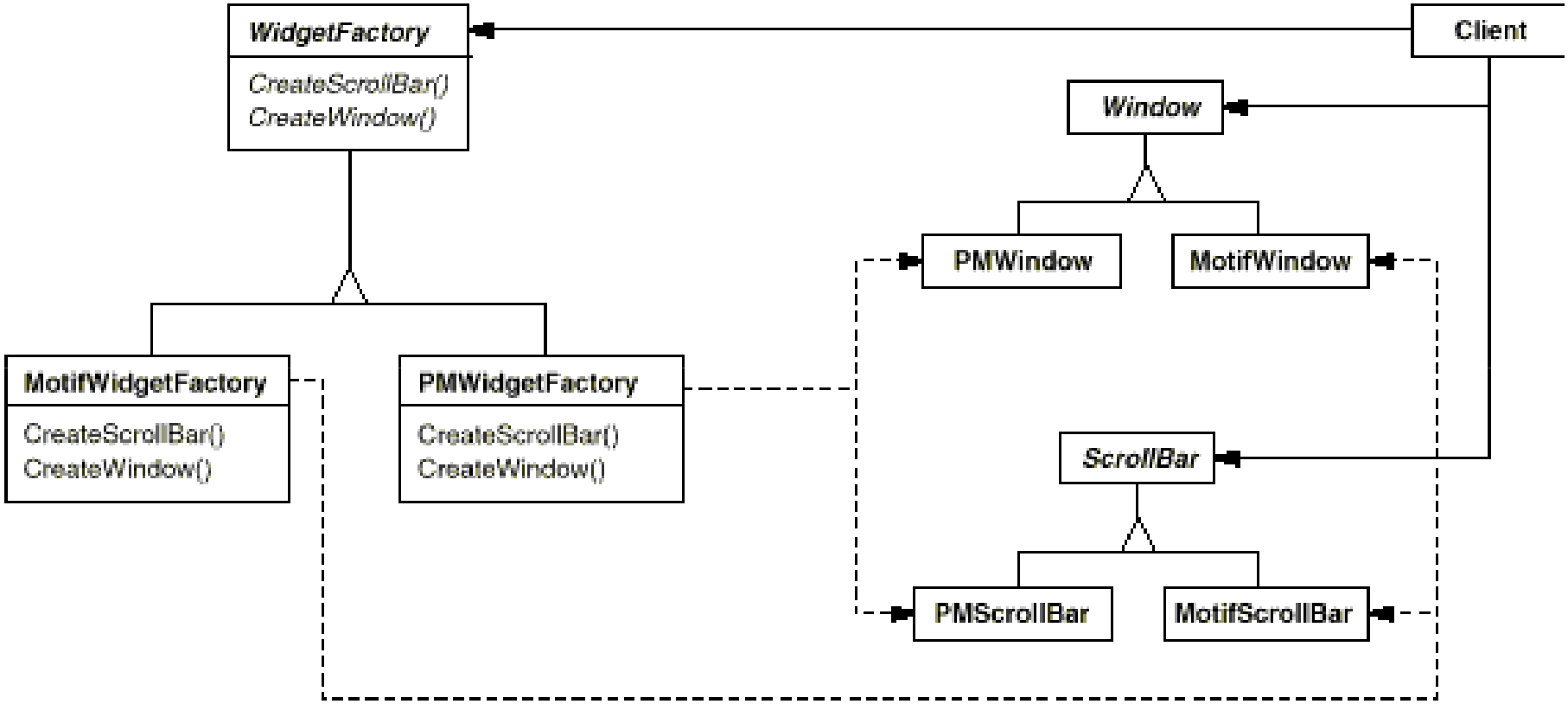
- Forme f = FormeFactory.createCarre(10, 12, 20)

- Faire évoluer l'implémentation sans modifier le client
  - Par exemple retirer la classe Carré pour un Rectangle
  - Idem pour les classes composites etc…
- Bien sûr il faut mettre à jour le code de la Factory …
- Mais on **maîtrise la portée des modifications**

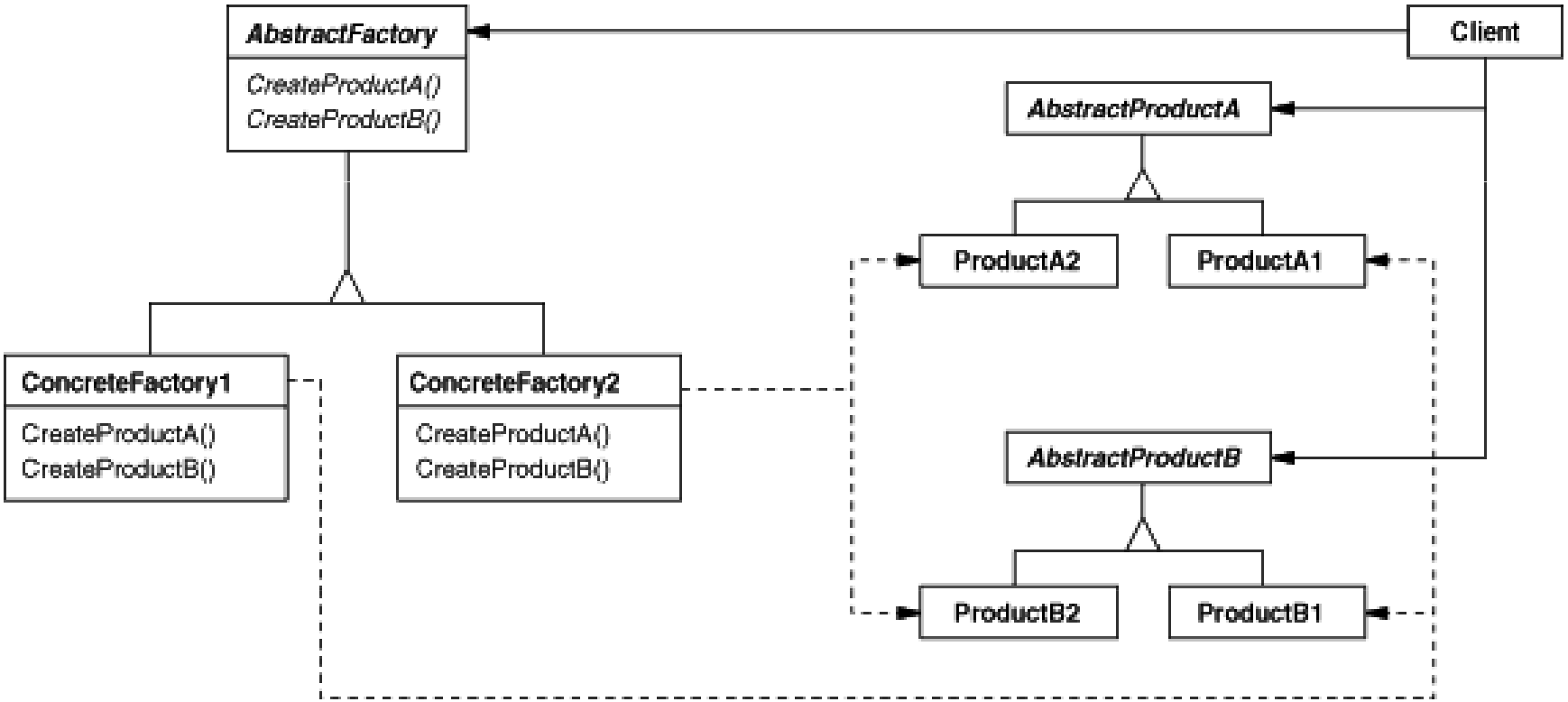
Pattern Abstract Factory

Plus flexible que la Factory static

La factory est elle-même une interface abstraite



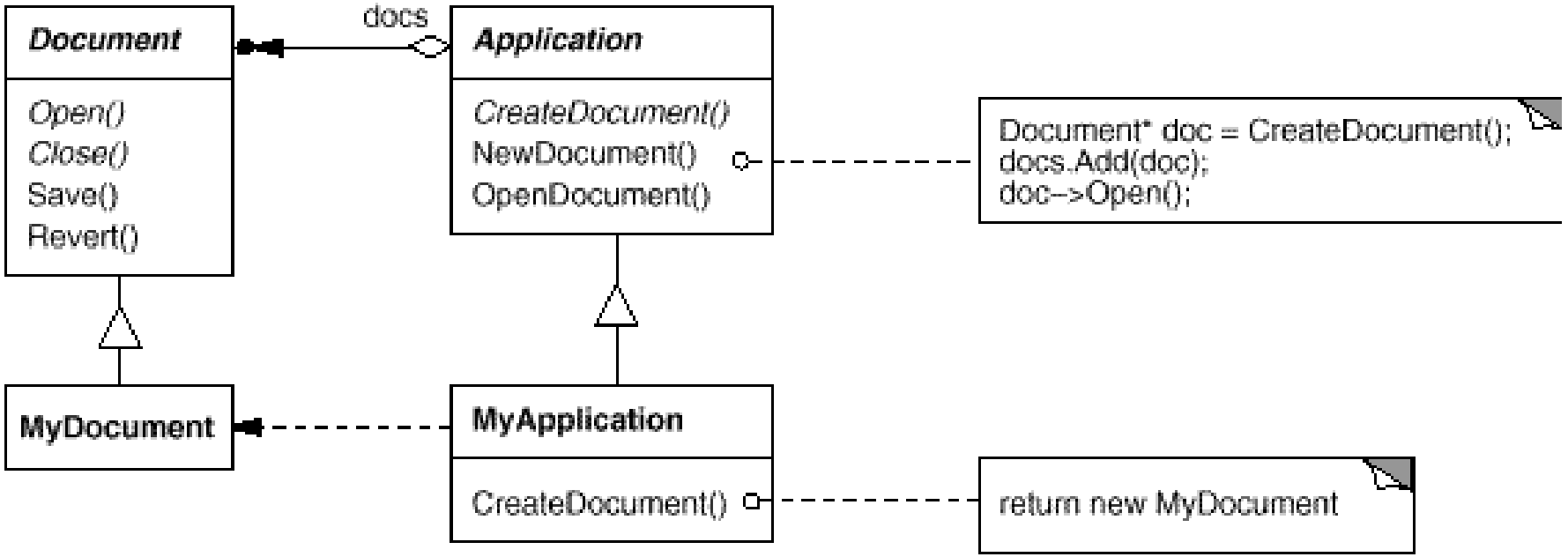
Pattern Abstract Factory



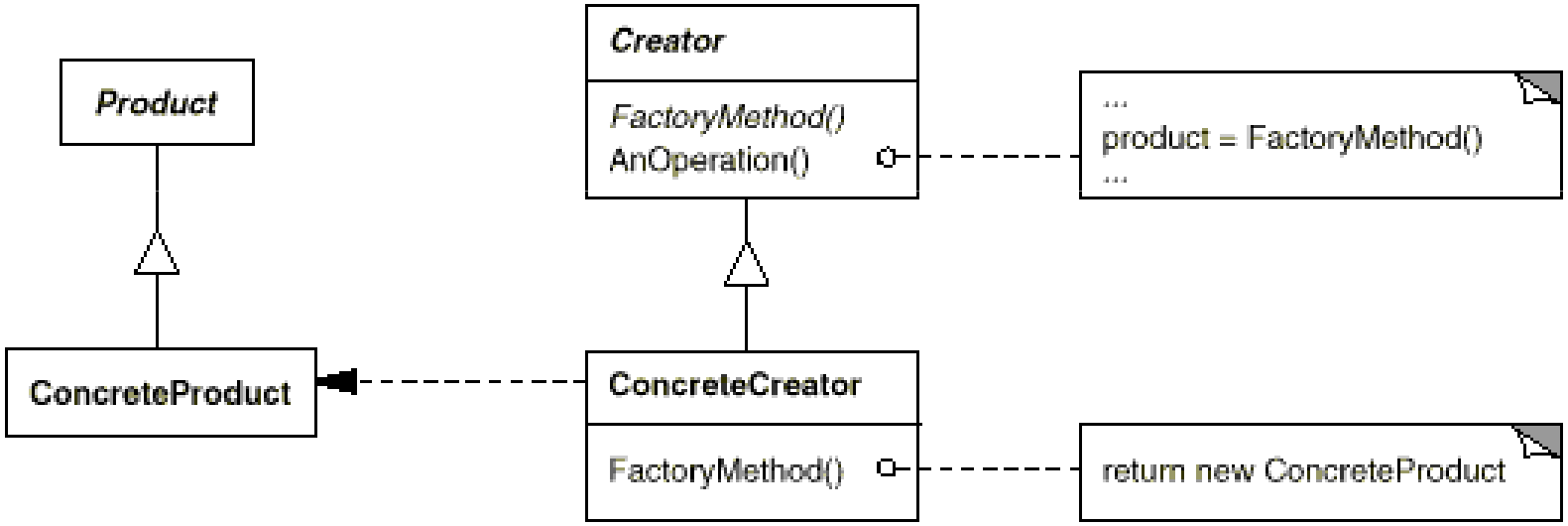
- Permet de créer des *familles d'objets liés entre eux*
- Configurer le client = lui passer une factory

Pattern Factory Method

- Variante de Factory travaillant sur les **fonctions**
- Notion de classe « à trou » qu'on rempli par comportements
- Permet même de moduler le comportement des classes !



Factory Method : structure



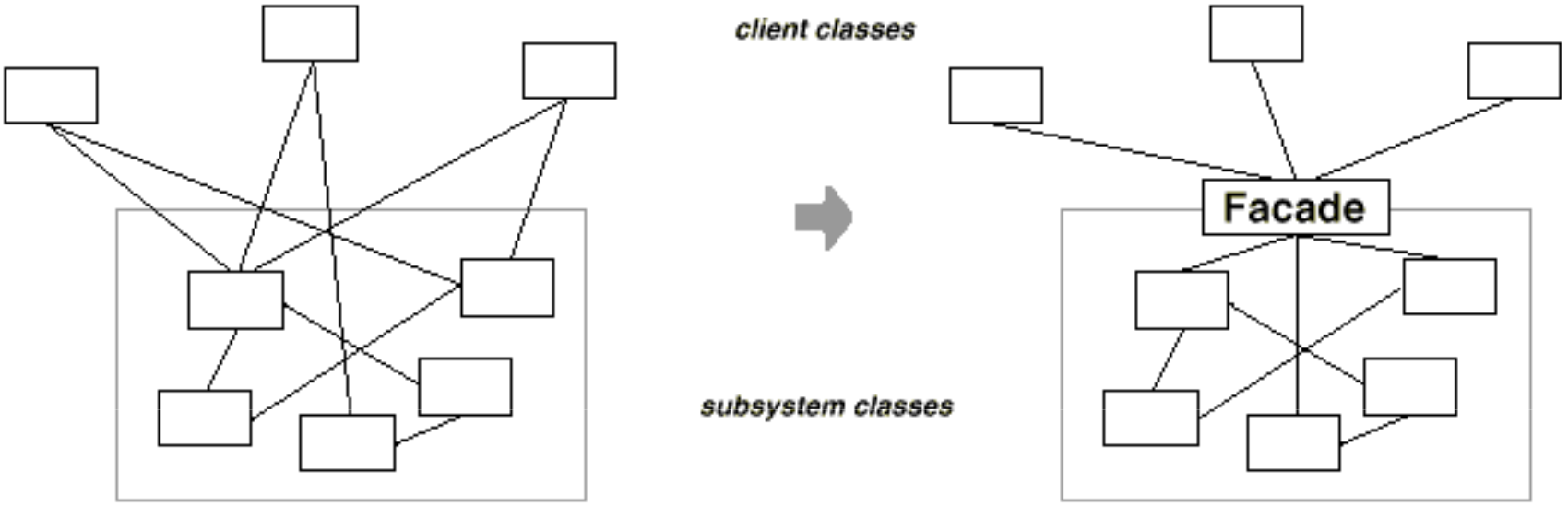
Intérêt

- Découple la classe (abstraite) des instances qu'elle manipule
- L'abstraite contient une grande majorité du code
- Pas de notion de « familles » d'objets comme AbstractFactory

Mini-exercice Factory

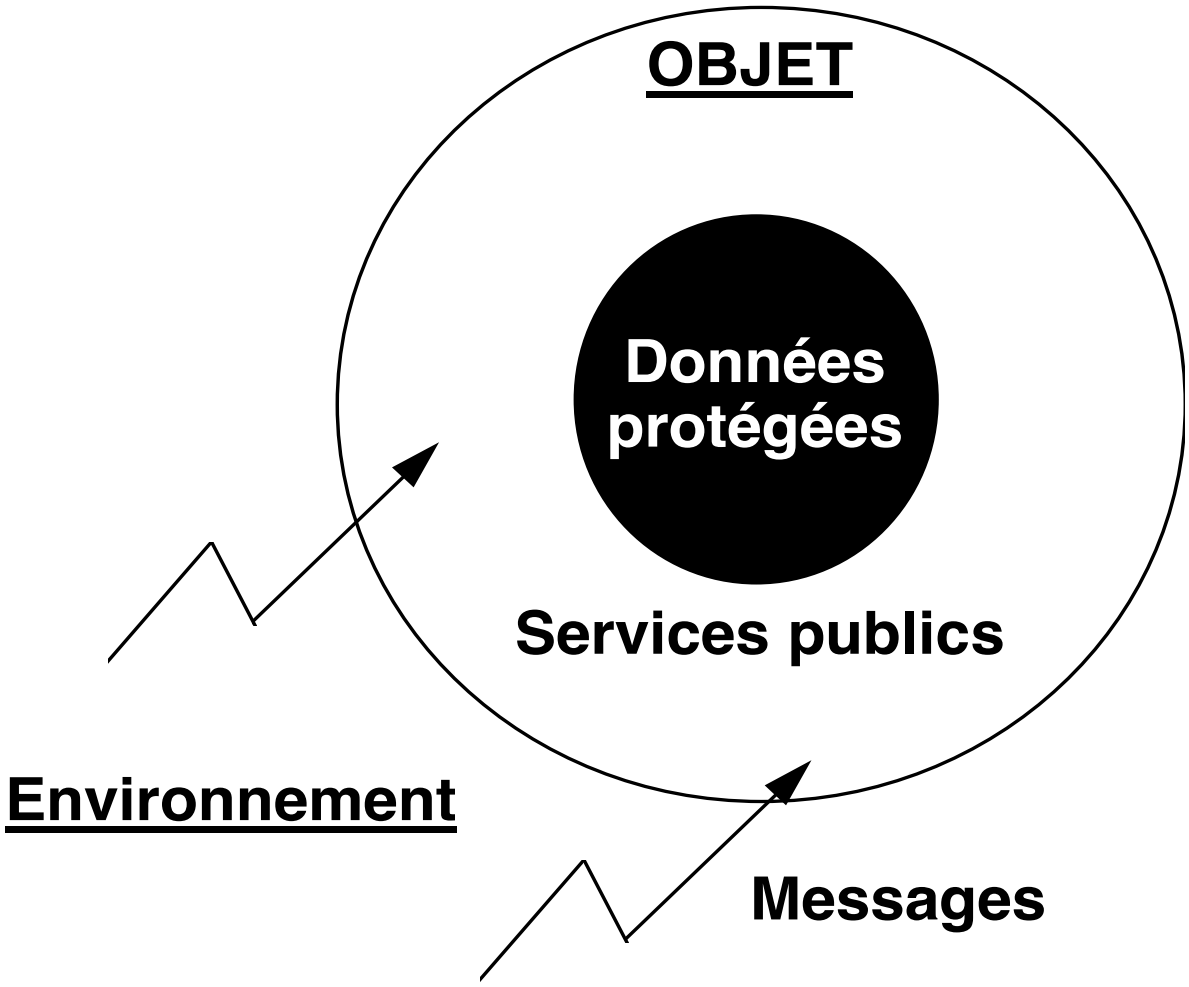
Pattern Façade

- Permet **d'isoler les dépendances entre sous-systèmes**
  - Maîtrise des modifications



- Principe très important, permet d'effectuer une forme de package depuis une classe (très pratique pour les distribs)

Encapsulation niveau objet



Pattern Façade : exemples

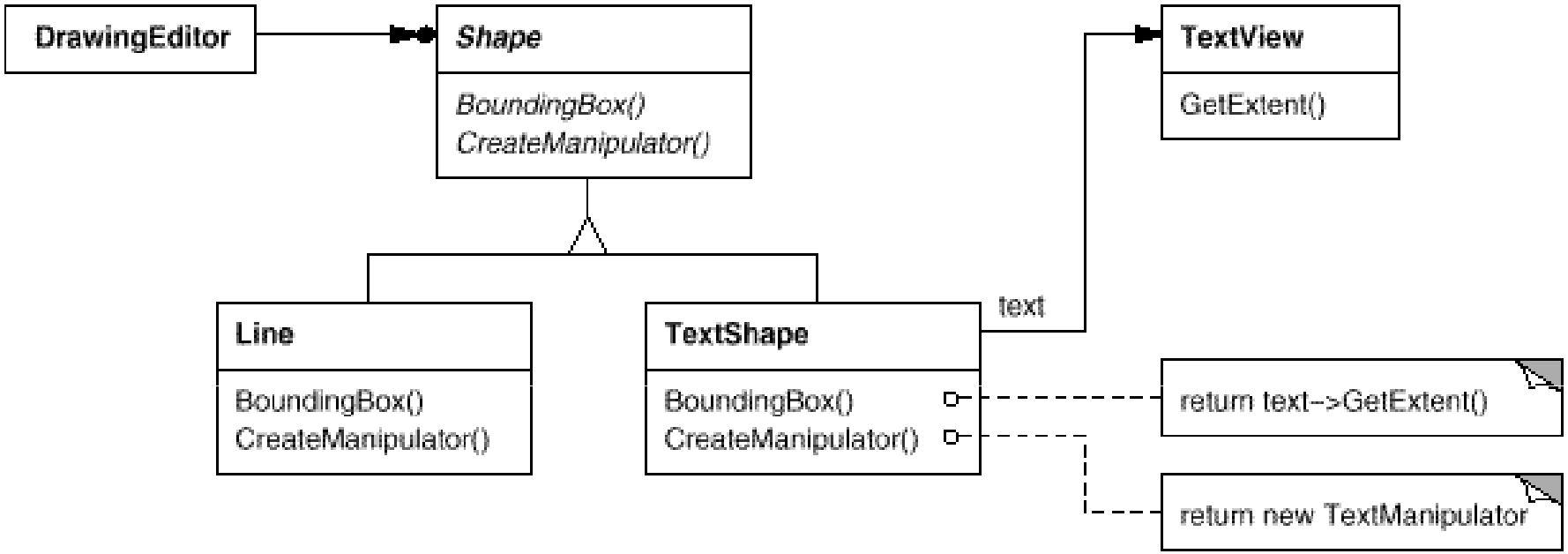
- La Façade de Formes est composée de
  - Forme
  - FormeFactory
- Les classes internes (Carré, Cercle) restent cachées

- Exemple ExpressionArithmétique
  - Expression, EnvironnementEvaluation
  - ExpressionFactory
- Classe concrètes (Add, Mul, …) non exposées.

Mini-exercice Façade

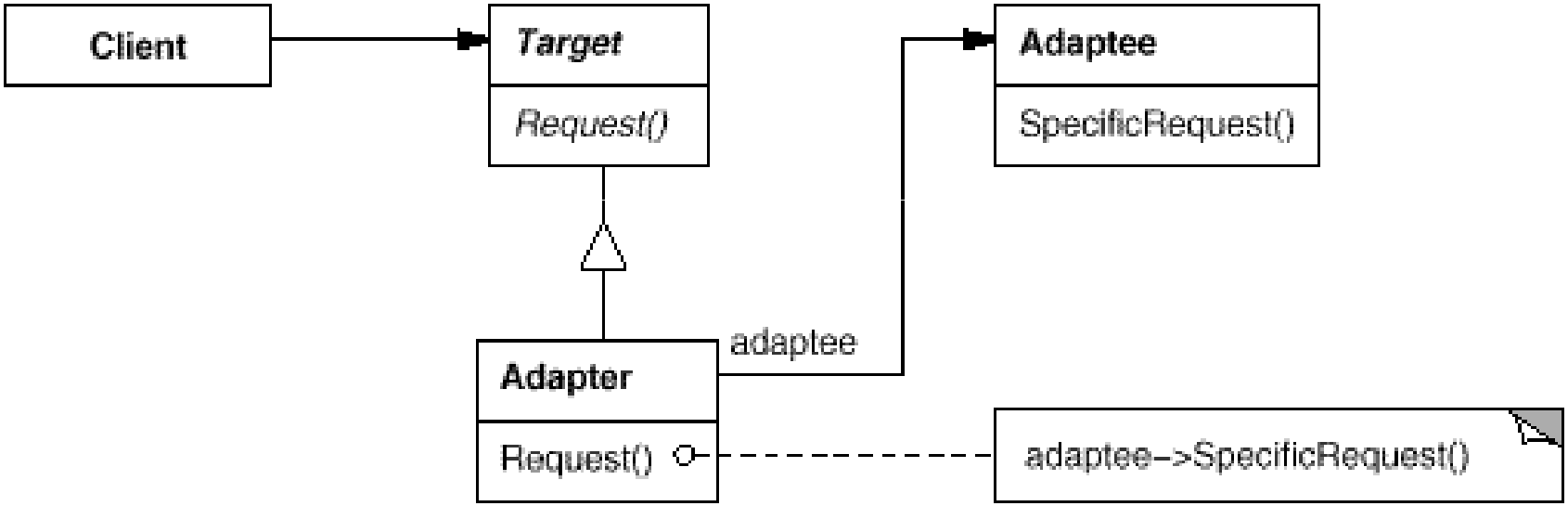
Pattern Adapter

- Réutiliser l'existant en **adaptant signature et opérations**



Pattern Adapter

- Adaptation plus ou moins complexe
- Parfois plus que de la simple délégation
- Base de la réutilisation en POO: délégation + typage interface



Mini-exercice Adapter

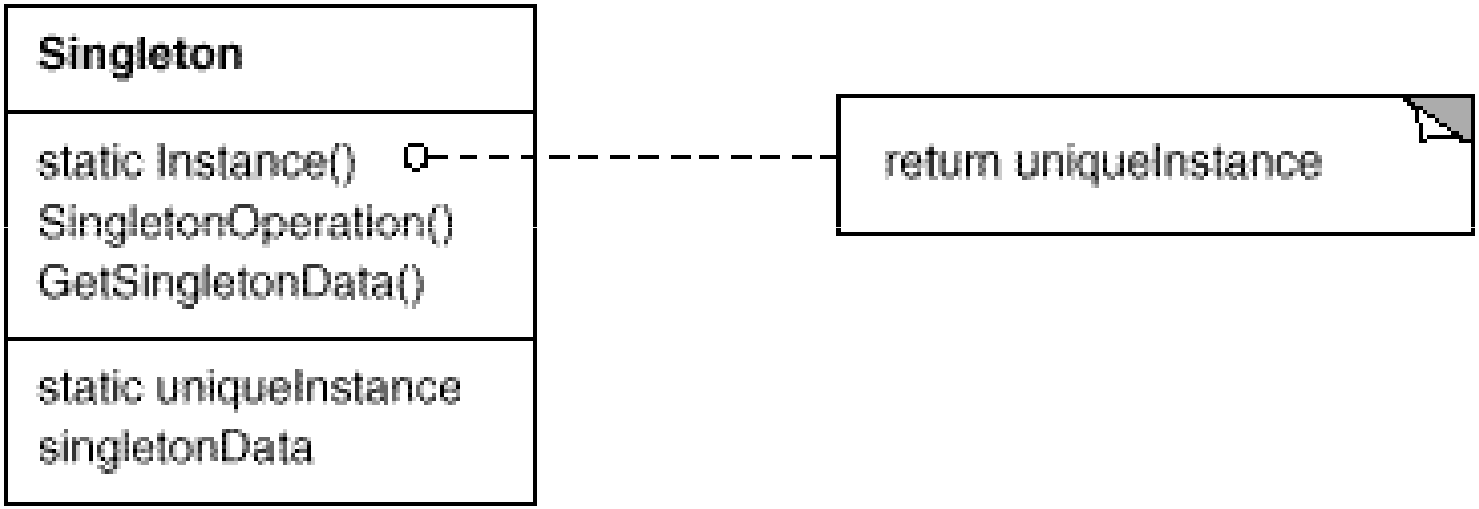
Pattern Singleton

- On souhaite pouvoir régler l'aspect graphique d'une app
  - Bordures, boutons, menus, transparence, etc…
  - Paramètres positionnés par l'utilisateur
- **Solution 1**: On crée une classe de configuration spéciale avec le main, celle-ci contient tous les paramètres et doit être passée à chaque création d'instance
  - **Intrusif, lourd … inacceptable**
- **Solution 2**: On ne définit que des opérations static dans la même classe spéciale
  - **Interdit l'héritage / redéfinition**
  - **Limite les évolutions futures**

Pattern Singleton

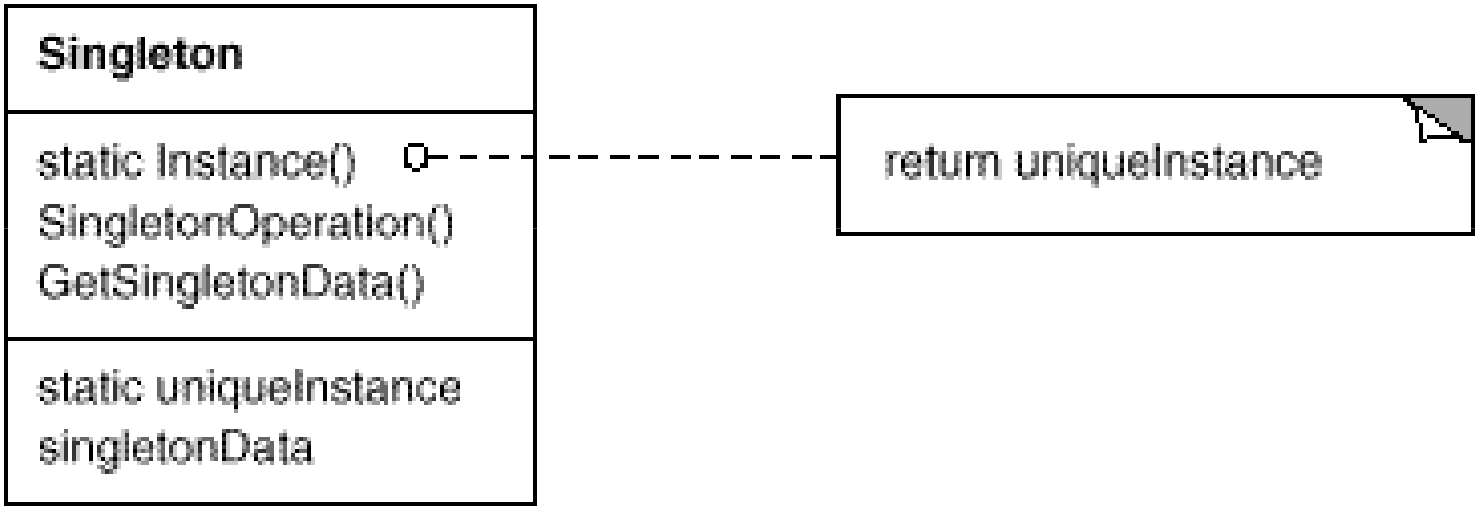
- Le singleton **assure qu'une seule instance est créée**
- L'idée est de n'autoriser qu'un constructeur static
- La classe contient un objet vers lui-même !
- A l'appel du constructeur soit l'instance existe, sinon on la crée

  If (uniqueInstance == null) uniqueInstance = new Singleton()
  return uniqueInstance;



Pattern Singleton: Avantages

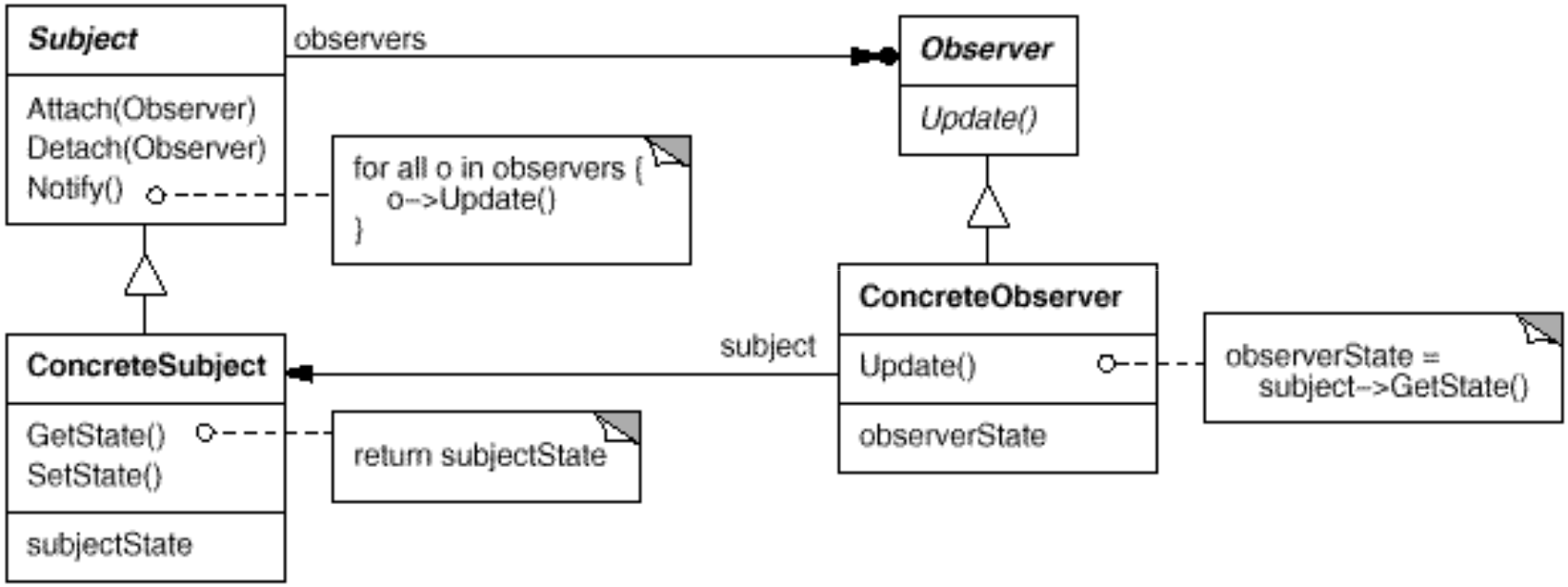
- Permet de contrôler les accès à l'instance
- Permet de configurer l'instance unique au runtime
- Permet de conserver l'extension par héritage
- Variantes possible pour contrôler un nombre N d'instances



Mini-exercice Singleton

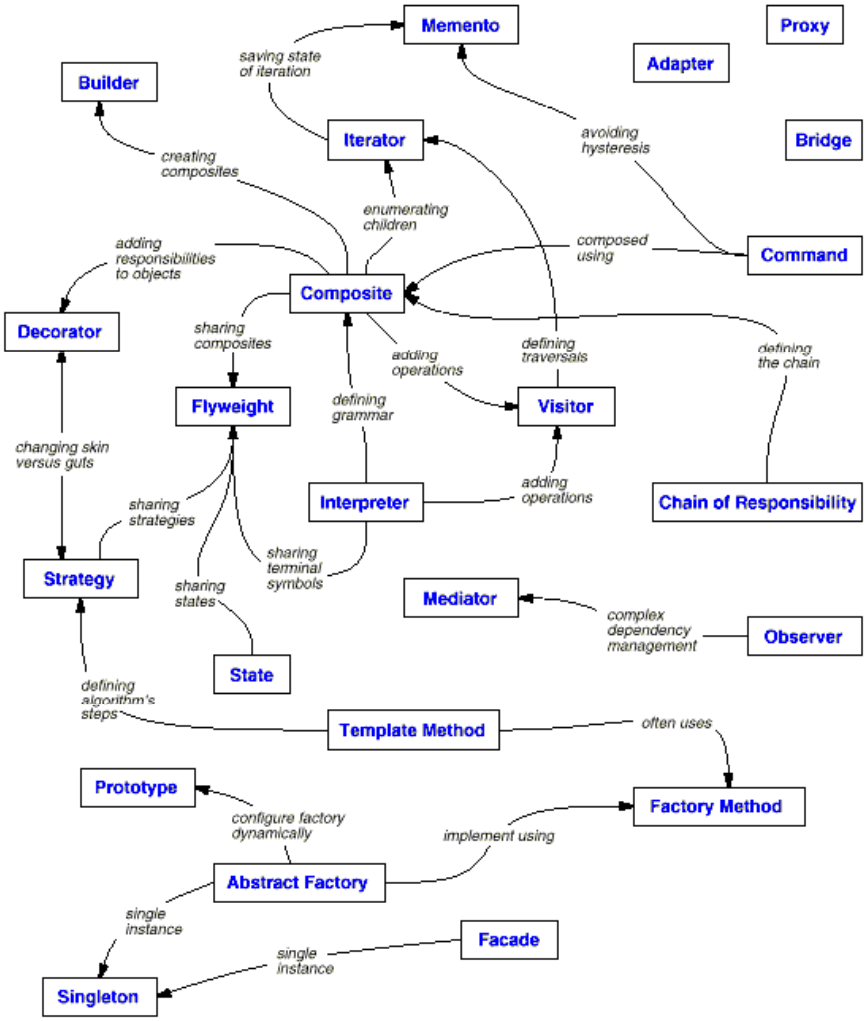
Pattern Observer

Objectif: Définir une dépendance entre un objet et ses observateurs telle que si l'objet est mis à jour, tous ses dépendants sont notifiés automatiquement.



Combinaison de Patterns

- Les Design Patterns sont très flexibles
- On peut facilement imaginer toutes combinaisons de ces Patterns.



Rappel des fondations de la POO

- **Abstraction**
  - Animal est abstrait. Zoo contient des animaux
- **Encapsulation**
  - Protection des attributs de l'objet
  - Contrôle des accès, isolation de l'implémentation
- **Polymorphisme**
  - Signature polymorphes, résolution des invocation
- **Héritage**
  - Redéfinition de comportement par héritage

Design patterns: Grands principes

- Isoler et encapsuler la partie variable
  - Algorithme dans Strategy
  - Façon de créer les objets dans Factory
  - Types de nœuds dans Composite
- Favoriser composition, délégation par rapport à l'héritage
  - Decorator vs. Redéfinition
- Utiliser des interfaces plutôt que des classes concrètes
  - Réutilisation algorithmique (ex. Collections.sort)
- Toujours chercher le couplage le plus faible possible entre des parties indépendantes qui interagissent
  - Dépendances fonctionnelles = interfaces (eg. Observer)
  - Evolutions facile, maîtrise des répercussions de changement

Design patterns: Grands principes

- Classes ouvertes en extension, fermées en modification
  - Réutilisation et modification se fait sans changer l'existant
  - Attention à héritage + redéfinition comme méthode d'extension !
- Toujours dépendre d'abstractions, jamais de classes concrètes
  - Toujours déclarer des interfaces
  - Bien réfléchir à ce que l'on souhaite exposer
- Ne parlez qu'à vos amis
  - Limiter le nombre d'objets connus par un autre objet
  - Réfléchir aux dépendances induites
- Ne m'appelez pas, je vous appellerai
  - Communication asymétriques, gros composants dépendent des petits
- Une classe ne devrait avoir qu'une seule responsabilité

Intérêt des patterns

- Un vocabulaire commun et puissant
- Les patterns aident à concevoir facilement des systèmes
  - Réutilisables: Responsabilités isolées, dépendances maitrisées
  - Extensibles: Ouverts aux enrichissements futurs
  - Limiter la modification de l'existant
  - Maintenables par faible couplage
- Les patterns reflètent l'expérience de développeurs objets
  - Solutions éprouvées et solides
- Les patterns ne sont pas du code mais des cadres de solutions générales à adapter à son problème particulier
- Les patterns aident à maîtriser les changements
  - Les solutions plus triviales sont souvent moins extensibles

Kick-starter Projet v.II