

Ingénierie Logicielle - Concepts et Outils de la modélisation et du développement de logiciel
par et pour la réutilisation.

Partie No 2 : Des Schémas de Réutilisation aux Schémas de Conception

Notes de cours - 2007-2012
Christophe Dony - Clémentine Nebut

1 Introduction

1.1 Idée

- Architecture - génie civil (Christopher Alexander) : *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*
- Informatique - génie logiciel
livre de base : [GHJV 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns : Elements of Reusable Object-Oriented Software* Addison Wesley, 1994.
- Le livre en ligne : <http://freeonlinebookstore.org/B-4205.html>
- les schémas du livre en ligne : <http://www.oodeesign.com/>
- Succès de l'idée :
Object-Oriented Reengineering Patterns Par Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, 2002, Morgan Kaufmann, 282 pages, ISBN 1558606394
<http://c2.com/ppr/>

Définition

- Un **Schéma de conception** nomme, décrit, explique et permet d'évaluer une conception d'un système extensible et réutilisable digne d'intérêt pour un problème récurrent.

Intérêts

- vocabulaire commun (communication, documentation et maintenance, éducation)
- gain de temps (explication, réutilisation)

1.2 Paramétrage, Variabilité

Tous les schémas du livre [GHJV 94] utilisent les schémas de paramétrage de base (paramétrage ou adaptation par spécialisation et par composition) présentés dans la première partie du cours.

Ils sont utilisés de façon variées pour réaliser des paramétrages que l'on peut classer de façon plus abstraite.

Ce qui varie	Schéma
Algorithmes	Strategy, Visitor
Actions	Command
Implementations	Bridge
Réponse aux changements	Observer
Interaction entre objets	Mediator
Création des objets	Factory, Prototype
Création des données	Builder
Traversal algorithm	Visitor, Iterator
Object interfaces	Adapter
Object behaviour	Decorator, State

Fig. (1): Classification extraite de [GHJV 94]

1.3 Les éléments d'un schéma

- **Un problème (récurrent)**
- **Un nom**
- **Une solution intéressante** : les éléments (diagrammes UML, code) qui traitent le problème; leurs relations, leurs rôles et leurs collaborations.
- **Des analyses de la solution et ses variantes** : avantages et inconvénients de l'utilisation du schéma, considérations sur l'implantation, exemples de code, utilisations connues, schémas de conception connexes, etc.

1.4 Classification des schémas

- **Schémas créateurs** : Décrire des solutions pour la création d'objets,
- **Schémas structuraux** : Décrire des solutions classiques d'organisation structurelles,
- **Schémas comportementaux** : Décrire diverses formes de collaboration entre objets.

2 Un exemple de schéma créateur : “Singleton”

Problème : Faire en sorte qu'une classe ne puisse avoir qu'une seule instance (ou par extension, un nombre donné d'instances).

Exemple : Les classes `True` et `False` de *Smalltalk*.

Solution de base :

```
1 public class Singleton {
2     private static Singleton INSTANCE = null;

4     /** La présence d'un constructeur privé (ou protected) supprime
5      * le constructeur public par défaut. */
6     private Singleton() {}
```

```

8      /** ''synchronized'' sur la méthode de création
9      * empêche toute instanciation multiple même par différents threads.
10     * Retourne l'instance du singleton. */

12     public synchronized static Singleton getInstance() {
13         if (INSTANCE == null)
14             INSTANCE = new Singleton();
15         return INSTANCE;
16     }

```

Discussion :

- Visibilité du constructeur
- Comparaisons entre la solution *Smalltalk* et la/solution *C++/Java*.

```

1  class Singleton class
2      new
3      (INSTANCE isNil) ifTrue: [INSTANCE := super new].
4      return (INSTANCE)

```

Commentaires : méta-classes, possibilité de spécialiser différentes primitives dont l'instantiation.

- Empêcher la copie (déclarer sans le définir le constructeur par copie)

```

1  Singleton (const Singleton&) ;

```

- Utilisation du schéma Singleton pour les classes **True** et **False** en *Smalltalk*, possible discussion sur l'envoi de message, les structures de contrôle dans le monde objet et l'intérêt des fermetures lexicales.

3 Un exemple de schéma structurel : “Decorateur” ou “Wrapper”

3.1 Problème

Ajouter ou retirer dynamiquement des fonctionnalités à un objet individuel, sans modifier sa classe.

Application typique : Ajouter des décorations (“barre de scroll”, “bordure”) à un objet graphique (Figure 2).

3.2 Exemple Type : décoration d'une "textView"

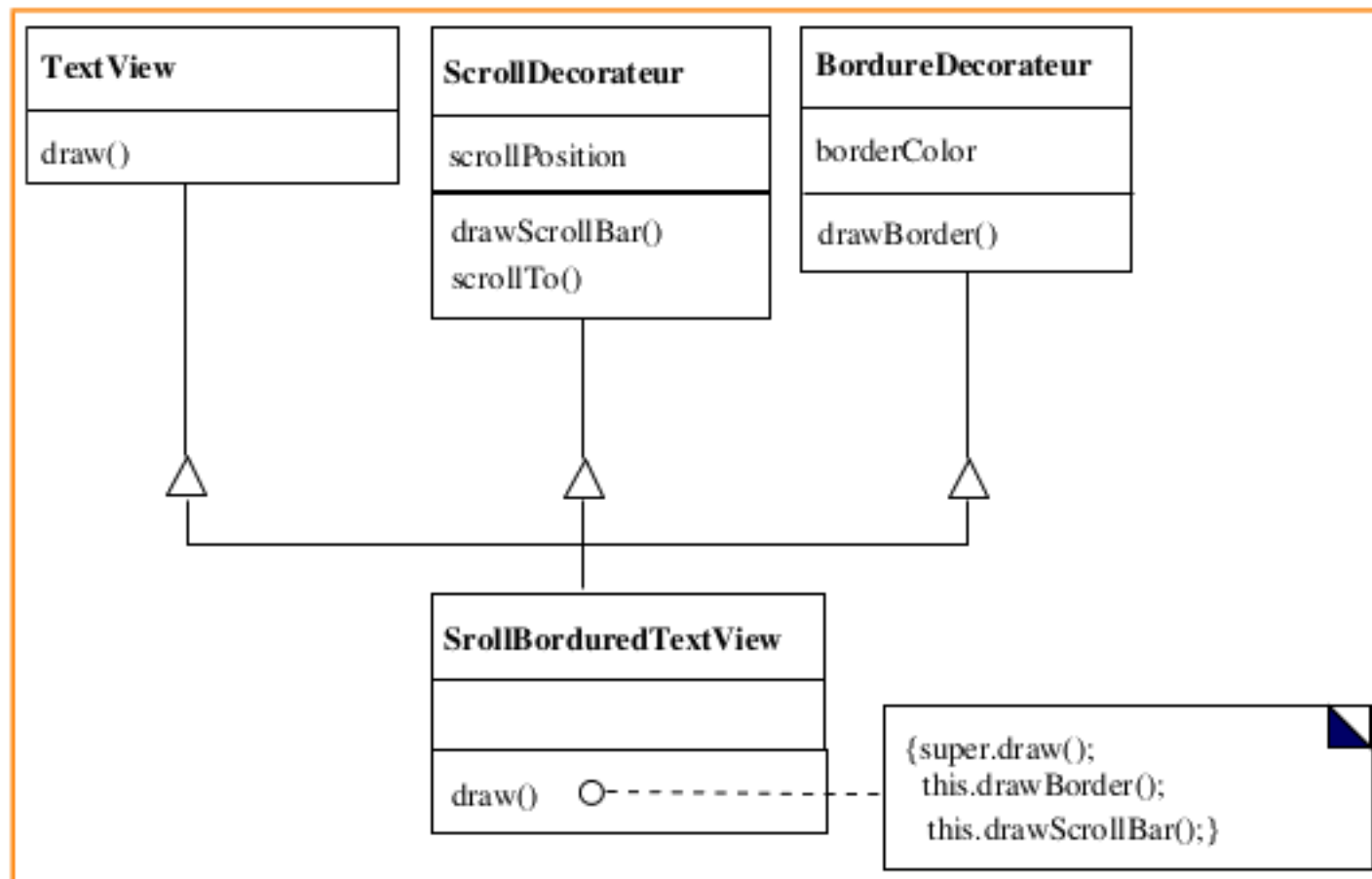


Fig. (2): Décoration d'une `textView`, solution universelle avec héritage multiple. Limitations : a) statique b) autant de sous-classes que de combinaisons potentielles de décorations.

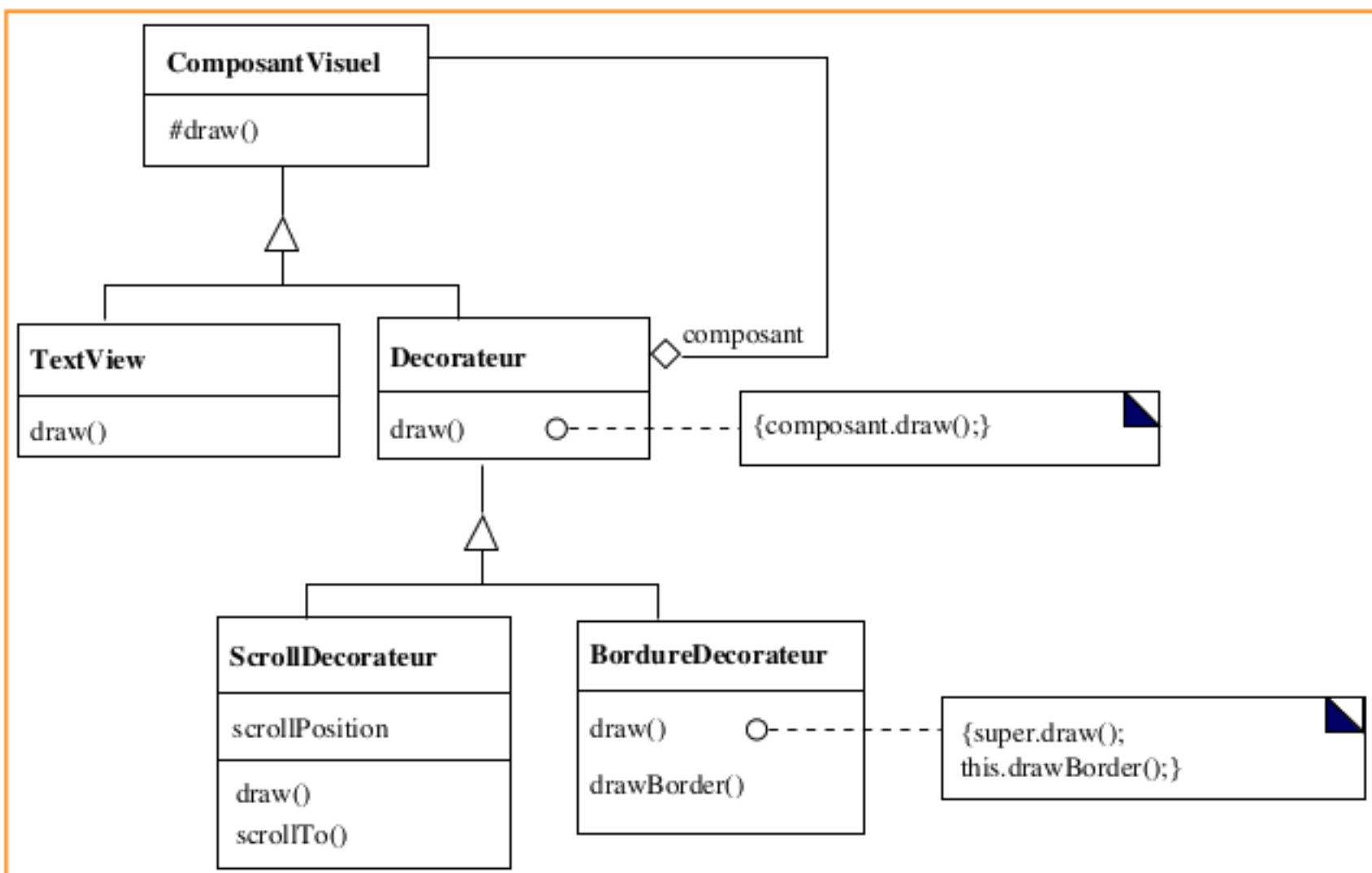


Fig. (3): Décoration d'une `Textview` : solution avec le schéma "Decorateur"

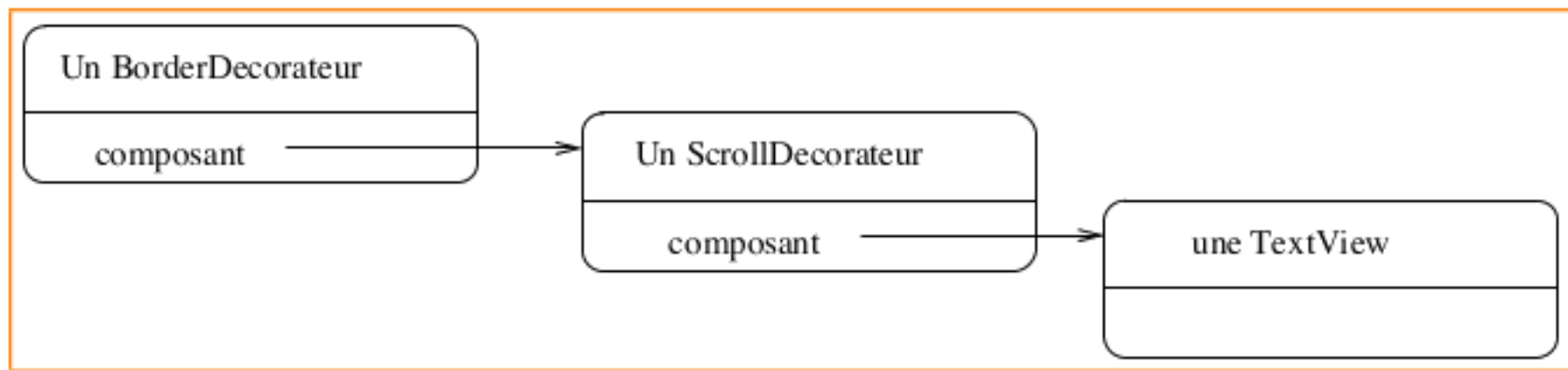


Fig. (4): Objets représentant une Textview décorée selon le schéma “Decorateur”.

3.3 Principe Général de la solution (figure 5)

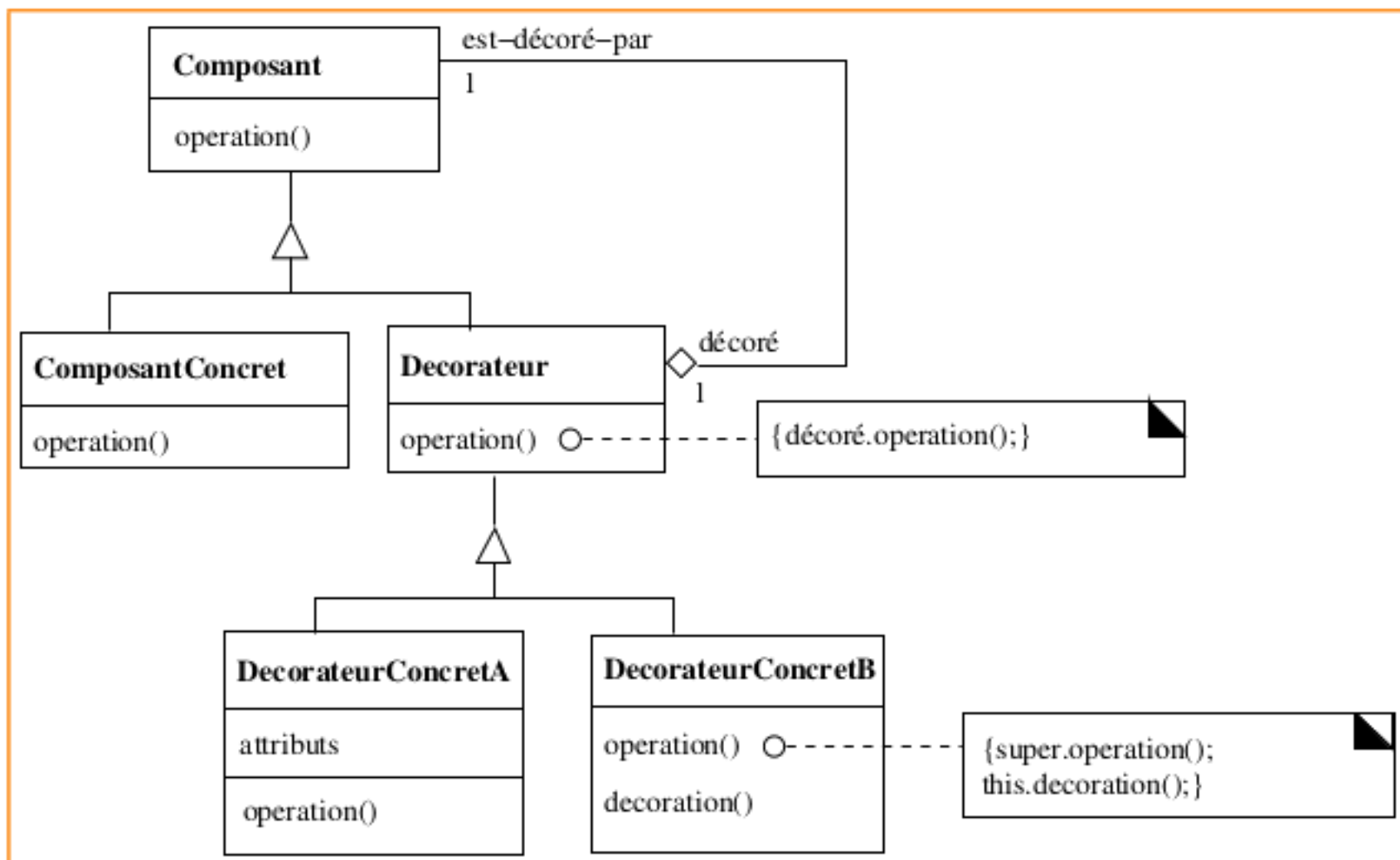


Fig. (5): **Composant** : objet métier quelconque à décorer, exemple : **textView**. **Décorateur** : objet décorant (donc ajoutant des fonctionnalités) à un objet métier, exemple : **scrollDecorator**).

3.4 Une mise en oeuvre concrète

Une implantation de “décorateur” sur un exemple jouet de composants graphiques.

```

1 public class Decorateur extends Composit {
2     Composit décoré;

4     public Decorateur(Composit c){
5         //affectation polymorphique
6         décoré = c;}
  
```



```

8     public void draw(){décoré.draw();}
9 }

```

```

1 public class public class BorderDecorator extends Decorateur {
2     // ajoute une bordure à un composant graphique

4     float largeur; //largeur de la bordure

6     public BorderDecorator(Composant c, float l) {
7         super(c);
8         largeur = l; }

10    public void draw(){
11        super.draw();
12        this.drawBorder(); }

14    public void drawBorder() {
15        // dessin de la bordure
16        ... }
17 }

```

3.5 Discussion

- Lien avec *Composite*.
- Nécessité pour un décorateur d'hériter de la classe abstraite **Composant** et donc de redéfinir toutes les méthodes publiques pour réaliser une redirection de message
- Poids des objets : il est recommandé de ne pas définir (trop) d'attributs dans la classe abstraite **composant** afin que les décorateurs restent des objets "légers".
- Incompatibilité potentielle de différentes décorations.
- L'ordre d'ajout des décorations est significatif.
-

4 Un exemple de Schéma structurel : Adapteur

Problème

Adapter une classe dont l'interface ne correspond pas à la façon dont un client doit l'utiliser.

Fait intervenir les Participants suivants :

Cible : objet définissant le protocole commun à tous les objets manipulés par le client, (dans l'exemple : shape)

Client : objet utilisant les cibles (l'éditeur de dessins)

Adapté : l'objet que l'on souhaite intégrer à l'application

Adapteur : objet réalisant l'intégration.

4.1 Principe général de la solution : figures 6 et 7

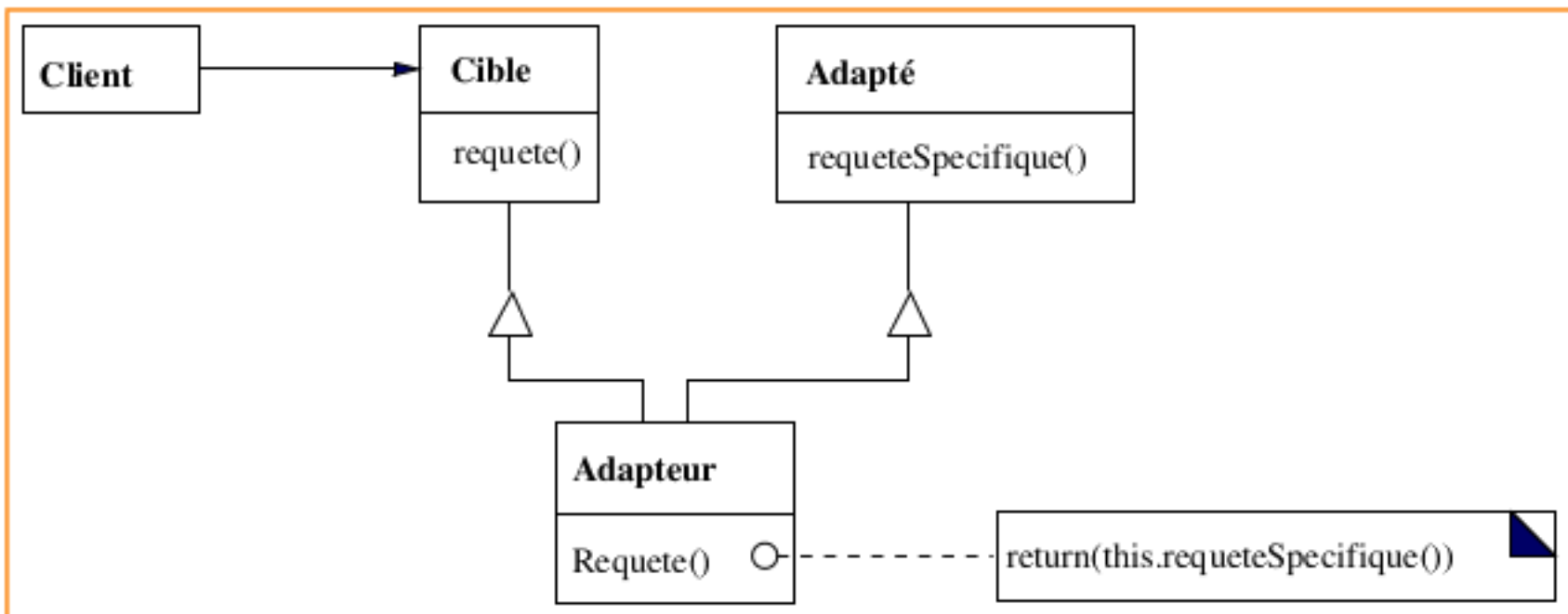


Fig. (6): Adapteur réalisé par spécialisation

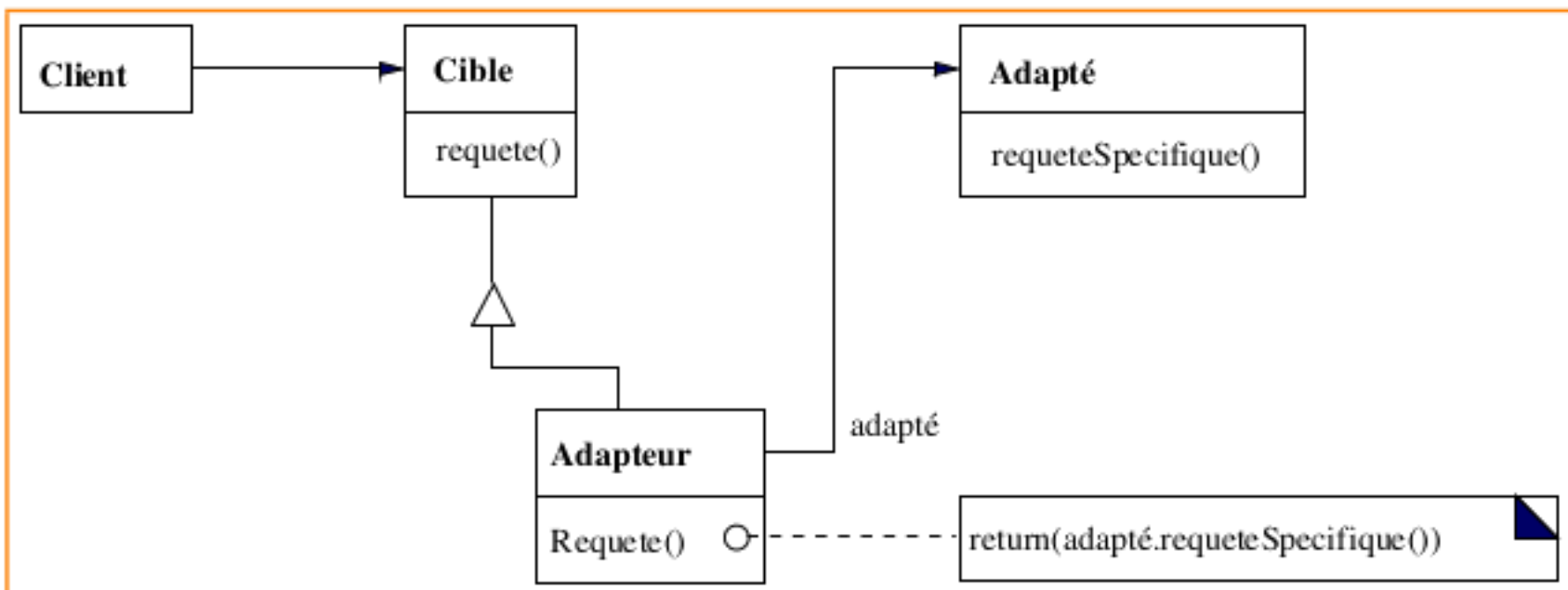


Fig. (7): Adaptateur réalisé par composition

4.2 Application Typique

Intégrer dans une application en cours de réalisation une instance d'une classe définie par ailleurs (cf. fig. 8). Soit à réaliser un éditeur de dessins (le client) utilisant des objets graphiques (les cibles) qui peuvent être des lignes, cercles, quadrilatères mais aussi des textes.

On pose comme contrainte de réutiliser une instance (l'objet à adapter) d'une classe `textview` définie par ailleurs.

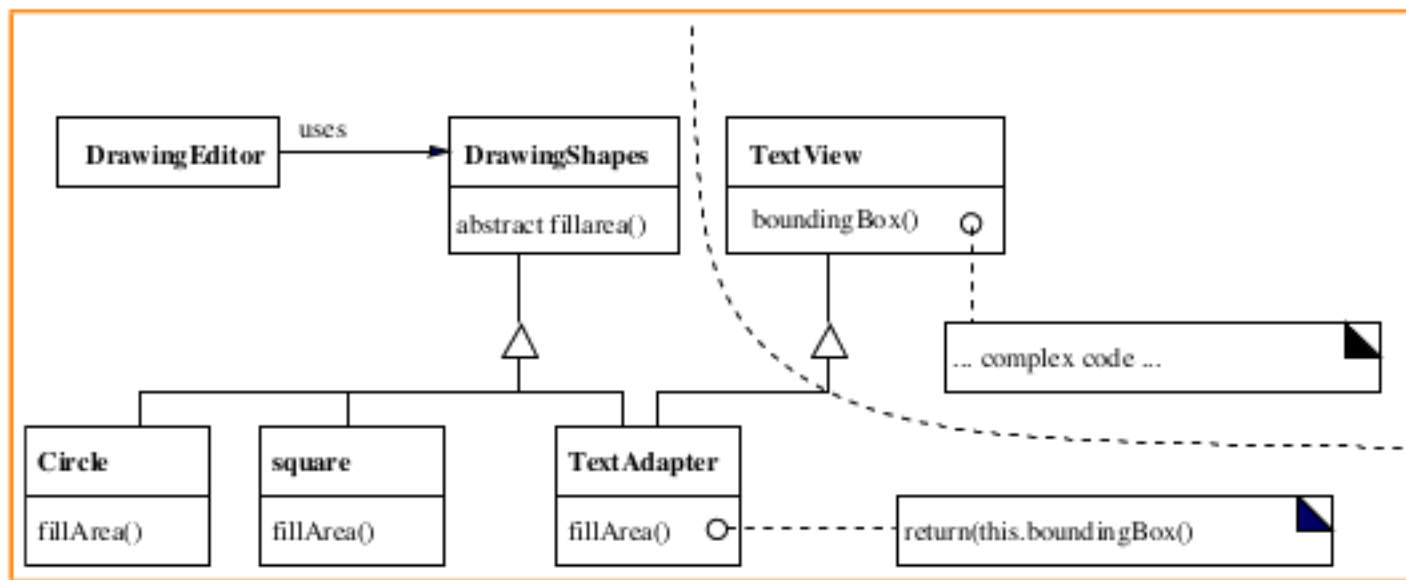


Fig. (8): Exemple d'adaptation par spécialisation

4.3 Discussion

- Application à la connexion non anticipée de composants. Voir cours “cbeans.pdf”.
- Mise en évidence du problème posés par la composition quand elle est utilisée en paliatif de l’héritage multiple :
 1. Nécessité de redéfinir toutes les méthodes publiques de la classe du composite.
 2. **Perte du receveur initial** (cf. fig. 9). Lors de la redirection vers le composite, le receveur est perdu, ceci rend certains schémas de réutilisation difficiles à appliquer.

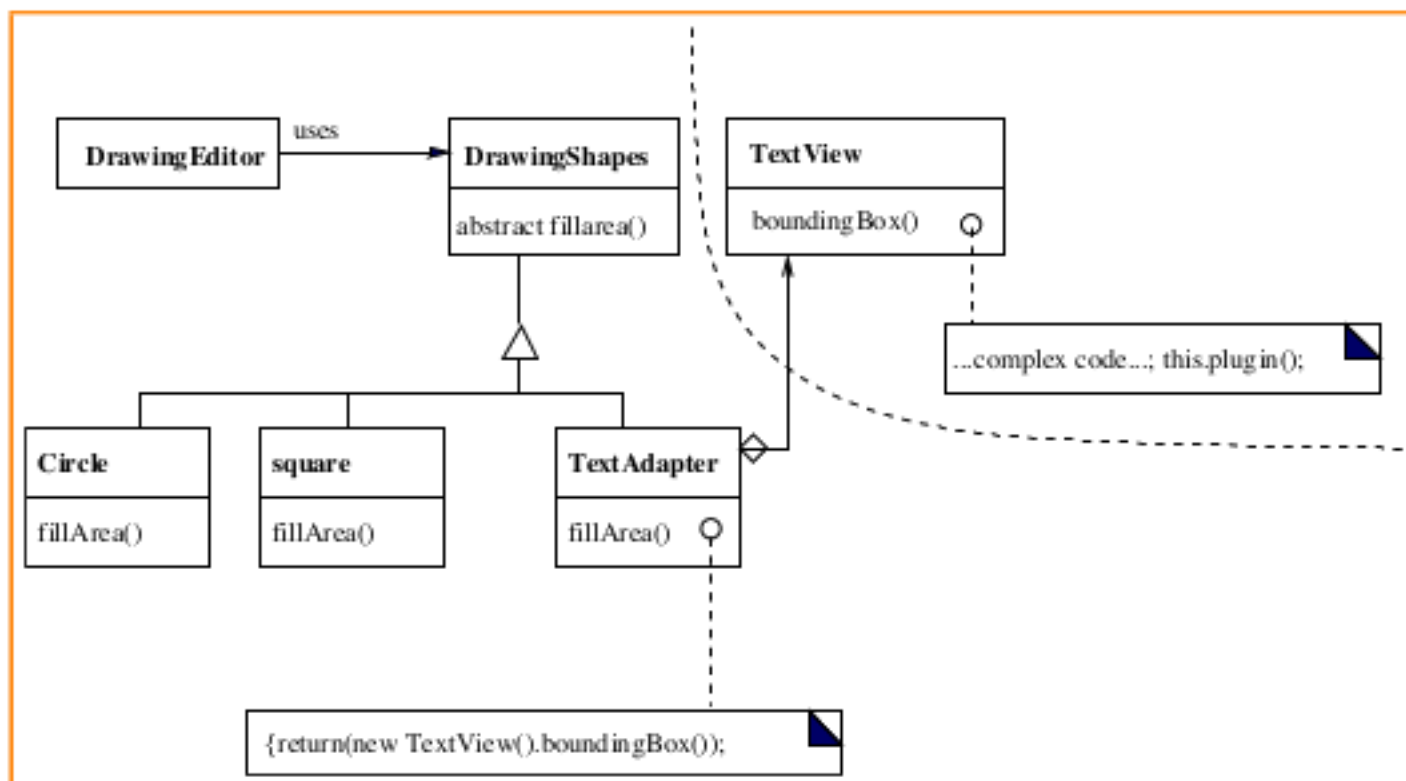


Fig. (9): Adaptation par composition dans l'exemple de l'éditeur de dessin. Perte du receveur initial dans la méthode boundingBox(). Ceci rend en l'état impossible la prise en compte d'une spécialisation dans l'application de la méthode plugin de la classe adaptée TextView. La figure (cf. fig. 10) propose un schéma global de solutionnement de ce problème.

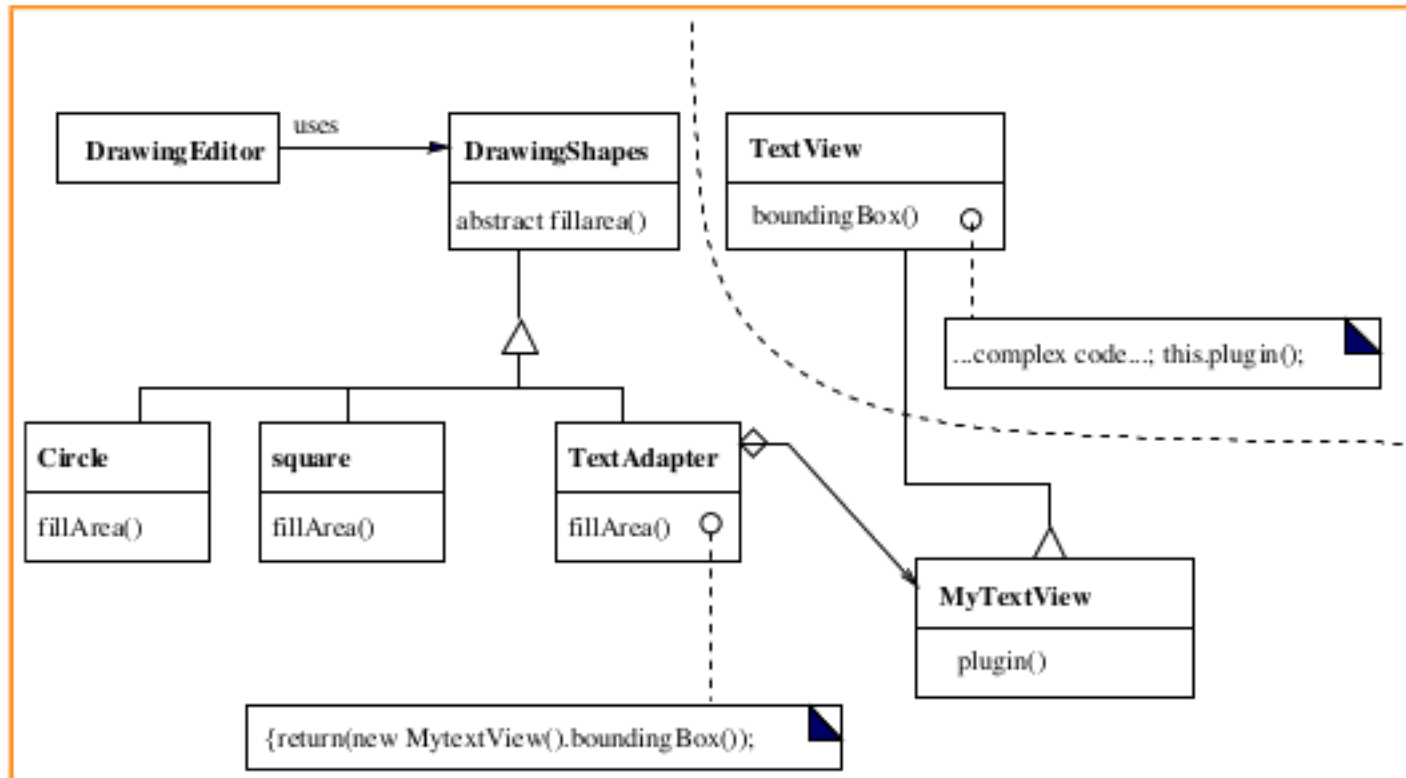


Fig. (10): Adaptation par composition dans l'exemple de l'éditeur de dessin selon un modèle offrant une solution au problème de perte du receveur initial.

5 Bridge : Séparation des interfaces et des implantations

5.1 Problème et Principe

Problème : Découpler une hiérarchie de concept des hiérarchies réalisant ses différentes implantations.

Principe (cf. fig. 11) : **Bridge** utilise l'adaptation par composition pour séparer une hiérarchie de concepts de différentes hiérarchies représentant différentes implantations de ces concepts.

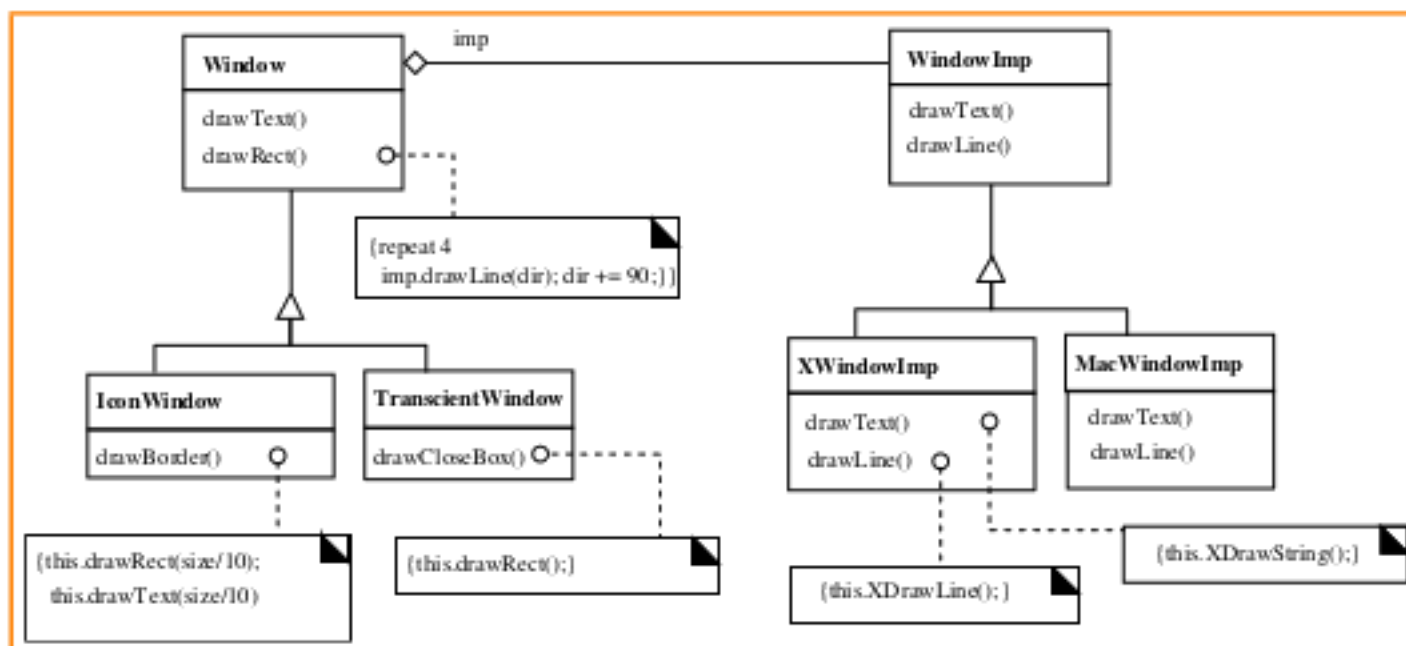


Fig. (11): Exemple d'application du schéma "Bridge"

5.2 Discussion

- Implantation : Qui décide des types de classes d'implantation créer ? Constructeur avec argument (lien à faire avec l'utilisation d'un framework paramétré par composition). Test dynamique selon critère. Délégation du choix à un objet externe, lien avec une fabrique (schéma *Factory*).

- **Bridge** évite la création de hiérarchies multi-critères mêlant les classes conceptuelles et les classes d'implémentation.
- Les concepts et les implantations sont extensibles par spécialisation de façon indépendantes.
- Les clients sont indépendants de l'implantation (Il est possible de changer une implantation (recompilation) sans que les clients n'en soient affectés).
- L'idée de **Bridge** peut être mise en relation avec le concept d'interface à la *Java*. Il en diffère selon divers points dont celui d'autoriser la définition de méthodes dans les hiérarchies de concept.

6 Schéma comportemental : “State”

6.1 Problème et Principe

Le schéma “State” propose une architecture permettant à un objet de changer de comportement quand son état interne change (cf. fig. 12).

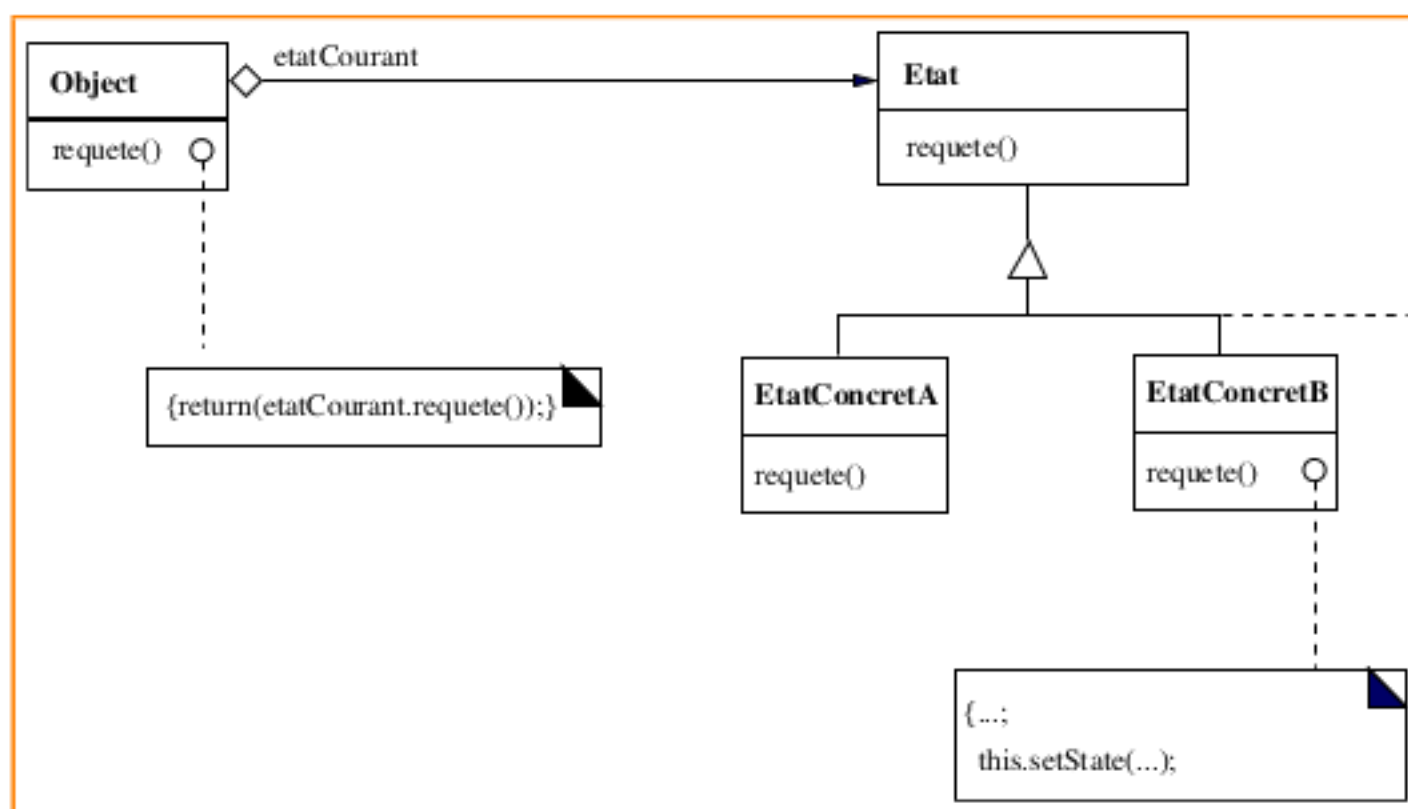


Fig. (12): Principe général du schéma “State”

6.2 Discussion

- Implémentation : comment représenter l'état courant ?
- Ce schéma rend explicite dans le code les changements d'état des objets en représentant chacun d'eux via une instance de classe : applications pour la sécurité (contrôle du bon état interne des objets) et la réutilisation (exercice : passer d'une calculatrice infixée à une postfixée).
- Amélioration possible des implantations en utilisant les énumérations.
- Langage intégrant les états comme concept de base (La définition par sélection de **Lore**).
- Solution liée, le *Become* : de Smalltalk. Si a et b sont des variables de type référence, après “a.become(b)”, toutes les références vers l'objet référencé par a sont devenues des références vers l'objet référencé par b.

```

1 A := Point x:2 y:3.
2 B := Point x:5 y:7.
  
```

```

3  C := A.
4  A become: B.
5  A x. "5"
6  B x. "2"
7  C x. "5"

```

6.3 Exemple d'application : implantation d'une calculatrice

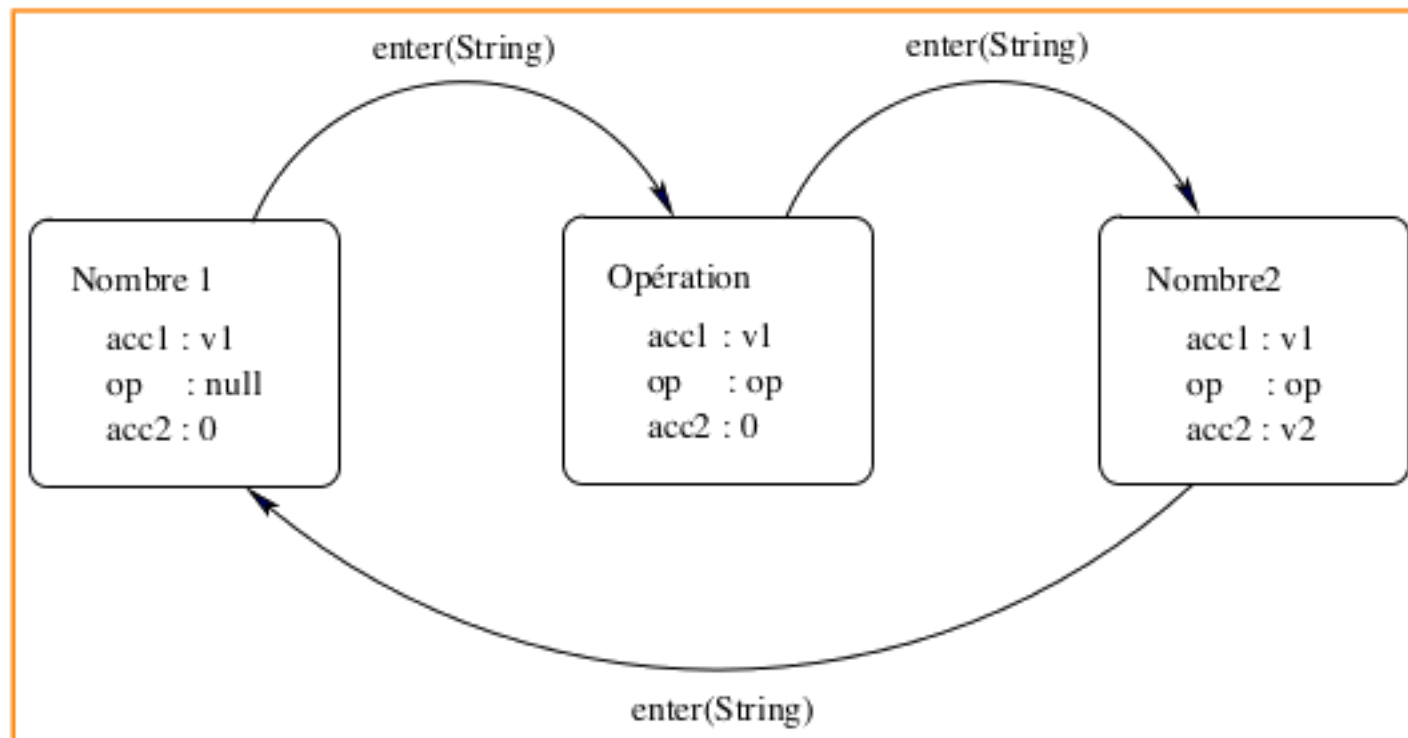


Fig. (13): Les différents états et transitions d'une "calculatrice" basique

Gestion des états

```

1  public class Calculette {
2      protected EtatCalculette etatCourant;
3      protected EtatCalculette[] etats = new EtatCalculette[3];
4      double accumulateur; // accumulateur
5      String operateur;

7      public Calculette(){
8          etats[0] = new ENombre1(this);
9          etats[1] = new EOperateur(this);
10         etats[2] = new ENombre2(this);
11         etatCourant = etats[0];
12         accumulateur = 0; }

14     ...
15     // accesseurs lecture/écriture pour "acc" et "operateur"
16     public double getResult() { return accumulateur; }

```

Distribution des calculs

```

1 public class Calculette {
2     ....

4     public void enter(String s) throws CalculetteException{
5         etatCourant = etats[etatCourant.enter(s) - 1]; }

```

Les états sont invisibles aux clients

```

1 public static void main(String[] args){
2     Calculette c = new Calculette();
3     c.enter("123"); //etat 1 : stocke le nombre 123 dans accumulateur
4     c.enter("plus"); //etat 2 : stocke l'opération à effectuer dans un registre
5     c.enter("234"); //etat 3 : stocke le résultat de l'opération dans accumulateur
6     System.out.println(c.getResult());}

```

Classe abstraite de factorisation

```

1 abstract class EtatCalculette {
2     static protected enum operations {plus, moins, mult, div};
3     abstract int enter(String s) throws CalculetteException;
4     Calculette calc;

6     EtatCalculette(Calculette c){ calc = c; }
7 }

```

Calculette dans état initial, dans l'attente de l'entrée d'un premier opérande

```

1 public class public class ENombre1 extends EtatCalculette{

3     ENombre1(Calculette c) { super(c); }

5     public int enter(String s) throws CalculetteException {
6         try{calc.setAccumulateur(Float.parseFloat(s));}
7         catch (NumberFormatException e)
8             {throw new CalculetteNumberException(s);}
9         //rend le nouvel état courant
10        return(2);} }

```

Calculette dans l'attente de saisie de l'opération à effectuer

(une gestion des exception plus fine serait nécessaire) :

```

1 public class public class EOperateur extends EtatCalculette{
2     EOperateur(Calculette c){ super(c); }

```

```
3 public int enter(String s) throws CalculetteException {  
4     calc.setOp(s);  
5     return(3);} }
```

Calculette dans l'attente de saisie d'un second opérande

L'application de l'opération aux opérandes peut y être réalisée :

```
1 public class public class ENombre2 extends EtatCalculette {  
2     ENombre2(Calculette c){super(c);}  
  
4     int enter(String s) throws CalculetteException {  
5         float temp = 0;  
6         try {temp = Float.parseFloat(s);}  
7         catch (NumberFormatException e) {  
8             throw new CalculetteNumberException(s);}  
  
10        switch (operations.valueOf(calc.getOp())) {  
11            case plus: calc.setAccumulateur(calc.getAccumulateur() + temp); break;  
12            case mult: calc.setAccumulateur(calc.getAccumulateur() * temp); break;  
13            default:  
14                throw new CalculetteUnknownOperator(calc.getOp());}  
15            return (1);} }
```

7 Le schéma comportemental : “Observateur”

7.1 Problème

Faire qu'un objet devienne un observateur d'un autre afin qu'à chaque fois que l'observé est modifié, l'observateur soit prévenu.

Exemple d'application :

- Abonnements, toutes formes de “publish/subscribe”,
- connexion non anticipée de composants
- IHM (MVC).

7.2 Principe général de la solution (Figure 14)

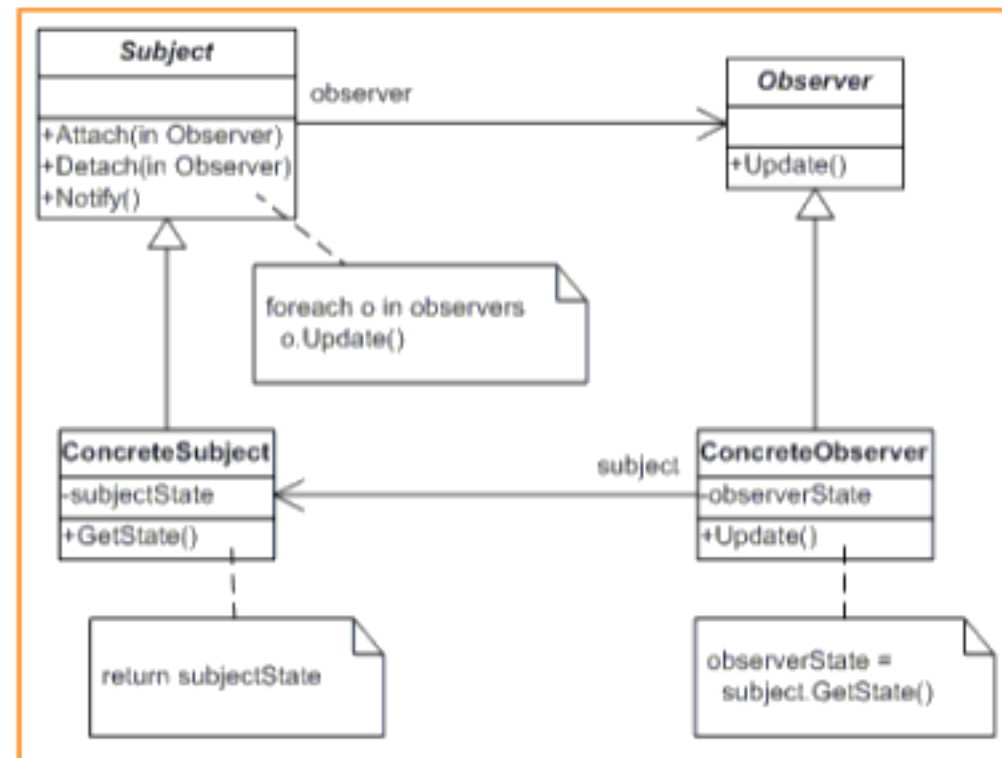


Fig. (14): Le schéma `Observer` permet d'établir une collaboration entre un observateur et un observé (ou "sujet" selon ce diagramme).

7.3 Points clés de l'Implantation

- Gestion de liste d'écouteurs par l'observé.
- Spécification par l'observé des points d'écoute.
- Information des observateurs par envoi de message où d'évènement (parfois une émission d'évènement est implantée comme un envoi de message).

7.4 Une mise en oeuvre typique avec le framework MVC

Historique :

- Le schéma MVC (article¹ [Krasner, Pope 1977]), un mécanisme permettant :
 - découpler les traitements métier (le modèle) de l'affichage (la vue)
 - découpler le contrôle (souris, clavier) de l'affichage
 - avoir plusieurs vues pour un même objet.
- 1995 : schéma "observateur" généralisation de l'idée de séparation entre observateur et observé.
- 1995 le schéma MVC est parfois qualifié de "framework" car il propose une architecture logicielle. Une implantation ainsi qu'une application qui en dérive sont données dans les sections suivantes.

7.5 Architecture

Trois hiérarchies de classes définissent les modèles, les vues et les contrôleurs.

Toute classe métier est sous-classe de la classe `Model`.

¹Krasner Pope tutorial on MVC : <http://wiki.squeak.org/squeak/800>

Chaque contrôleur connaît son modèle (et si on le souhaite sa vue), le contrôleur observe l'utilisateur.
 Chaque vue connaît son contrôleur et son modèle.
 Le modèle est observé par la vue (et si on le souhaite par le contrôleur).

7.6 Modèle d'interaction (figure 15)

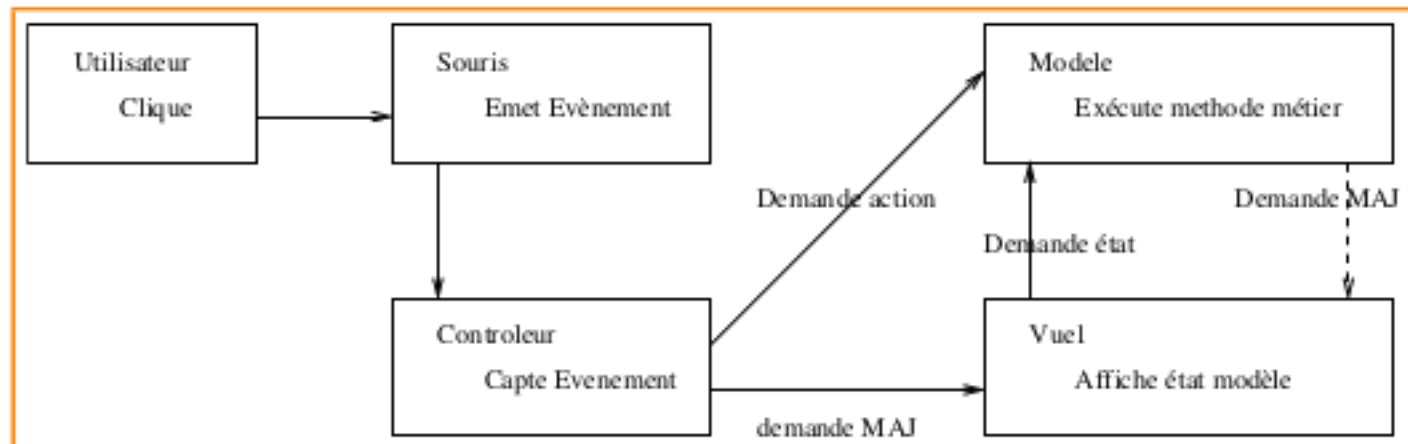


Fig. (15): Diagramme d'interaction entre modèles, vues et contrôleurs du MVC

7.7 Implantation (squelette) inspirée de l'implantation originelle (Smalltalk-80)

7.7.1 Gérer les dépendances “observateur-observé”

Classe du framework

```

1 public class MV-Association{
2     //utilise un dictionnaire pour stocker les couples modèle-vue
3     static Dictionary MVDictionary = new Hashtable();

4
5     //permet d'associer un modèle à une vue
6     public static void add(Model m, View v) { ... }

7
8     // rend la collection de vues associées à un modèle
9     public static Collection<View> getViews(Model m) { ... }
    
```

7.7.2 Les modèles

Classe du framework

```

1 public class Model{

2
3     public void changed(Object how){
4         Iterator i = MV-Association.getViews(this).iterator();
5         while (i.hasNext())
6             (i.next()).update(how);}
7     }
    
```

Classe de l'application

```
1 public class public class Compteur extends Model{
2     protected int valeur = 0;

4     protected changerValeur(i){
5         valeur = valeur + i;
6         this.changed("valeur");}

8     public int getValeur(){return valeur;}
9     public void incrémenter(){this.changerValeur(1);}
10    public void décrémenter(){this.changerValeur(-1);}
```

7.7.3 Les vues

Classe du framework

```
1 public class View{
2     Controller cont;
3     Model model;

5     public View(Model m, Controller c){
6         model = m;
7         cont = c;
8         MVAssociation.add(this, m);

10    public abstract void update (Object how);
11    public void open(){...}
12    public void redisplay(){...}
13 }
```

Classe de l'application

```
1 public class public class CompteurView extends View{
2     ...
3     private JLabel l;
4     ...

6     public void update(Object how){
7         l = new JLabel(String.valueOf(m.getValeur()), JLabel.CENTER);
8         this.redisplay();
```

7.7.4 Les contrôleurs

Classe du framework

```
1 public abstract class Controller implements ActionListener{
2     Model m;

4     public Controller(Model m){
```

```
5     model = m;
6 }
7 }
```

Classe de l'application

```
1 public class CompteurController
2 public class extends controller
3         implements ActionListener{

5     public void actionPerformed(ActionEvent e) {
6         if (e.getActionCommand() == "incr") {m.incrémenter();}
7         if (e.getActionCommand() == "decr") {m.decrémenter();}
8     }
9 }
```

7.7.5 Lancement de l'application

```
1 Compteur m = new Compteur();
2 CompteurView v = new CompteurView(m, new CompteurController(m));
3 v.open();
```
