

Génie Logiciel - Schémas de Conception

Université Montpellier-II

Master Informatique IFPRU - Parcours GL

Notes de cours

Christophe Dony

1 Schéma de conception

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 : *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns : Elements of Reusable Object-Oriented Software Addison Wesley, 1994.*
<http://freeonlinebookstore.org/B-4205.html>
<http://www.oodesign.com/>
- *Object-Oriented Reengineering Patterns Par Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, 2002, Morgan Kaufmann, 282 pages, ISBN 1558606394*

Définition :

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

Intérêts :

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

1.1 Paramétrage, Variabilité

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

1.2 Les éléments d'un schéma

- **Le problème**
- **Le nom**
- **La solution** : les éléments (diagrammes UML, code) qui traitent le problème ; leurs relations, leurs rôles et leurs collaborations.
- **L'analyse** : avantages et inconvénients de l'utilisation du schéma, considérations sur l'implantation, exemples de code, utilisations connues, schémas de conception connexes, etc.

1.3 Classification des schémas

1. **Schémas créateurs** : Décrire des solutions pour la création d'objets,
2. **Schémas structurels** : Décrire des solutions classiques d'organisation structurelles,
3. **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 :

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

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

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

    public synchronized static Singleton getInstance() {
        if (INSTANCE == null)
            INSTANCE = new Singleton();
        return INSTANCE;
    }
}
```

Discussion :

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

```
class Singleton class
  new
    INSTANCE isNil ifTrue: [INSTANCE := super new].
    return (INSTANCE)
```
- Empêcher la copie (déclarer sans le définir le constructeur par copie)

```
Singleton (const Singleton&);
```

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

3.1 But

Ajouter ou retirer dynamiquement des fonctionnalités à un objet individuel, sans modifier sa classe, sans utiliser d’héritage multiple ni créer de trop nombreuses classes de “Mixin”.

Application typique : Ajouter des décorations (“barre de sroll”, “bordure”) à un objet graphique (Figure 1).

3.2 Exemple Type : décoration d'une "textView"

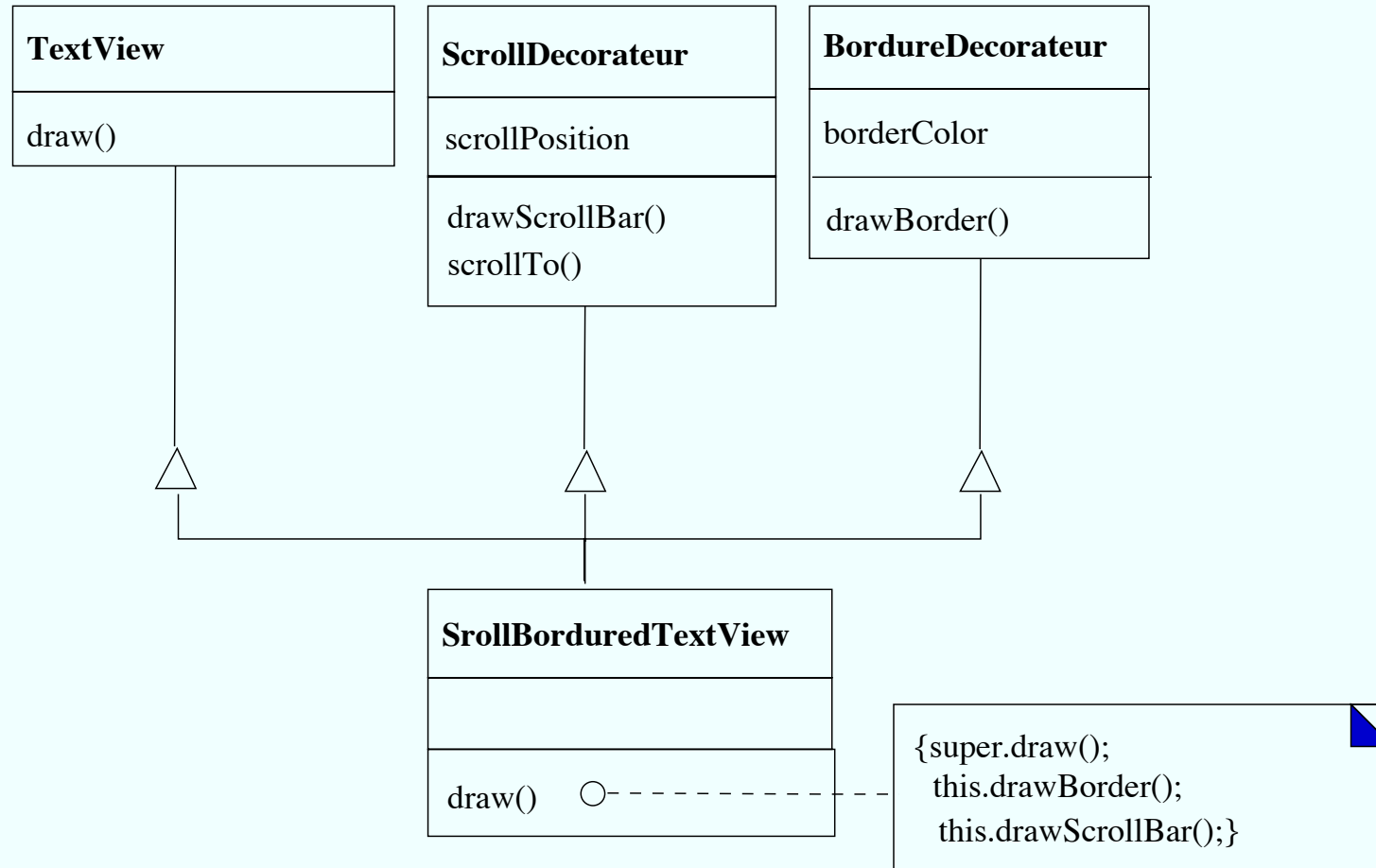


FIG. 1 – Décoration d'une `textView`, solution universelle avec héritage multiple. Problème : autant de sous-classes que de combinaisons potentielles de décorations.

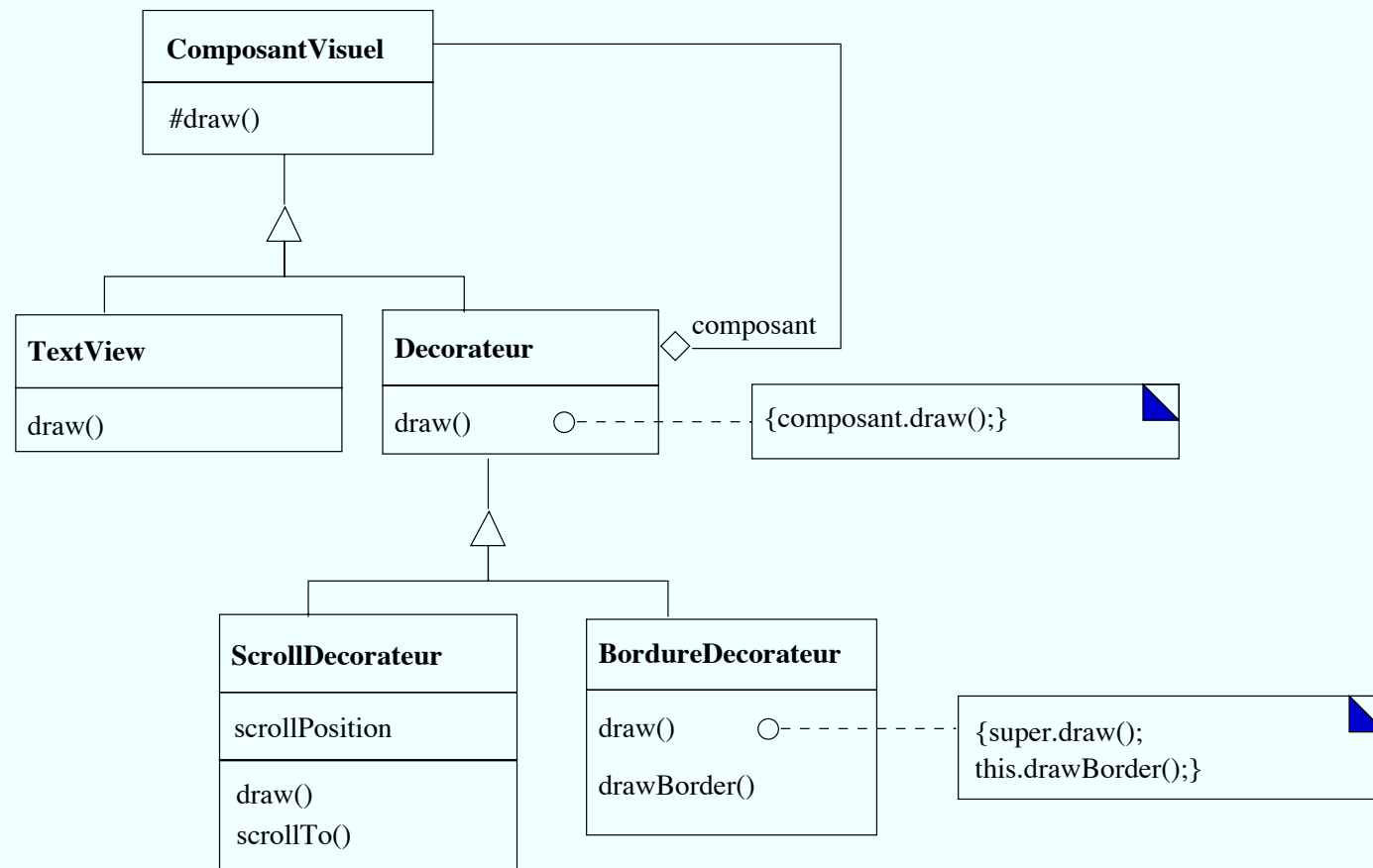


FIG. 2 – Décoration d’une `textView`, solution avec le schéma “Decorateur”

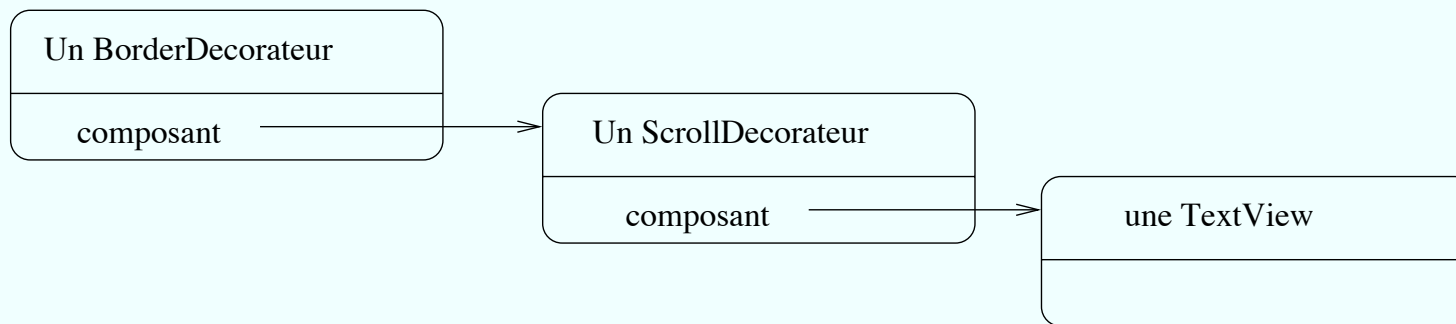


FIG. 3 – Décorateur : Objets représentant une `textView` décorée

3.3 Principe Général de la solution (figure 4)

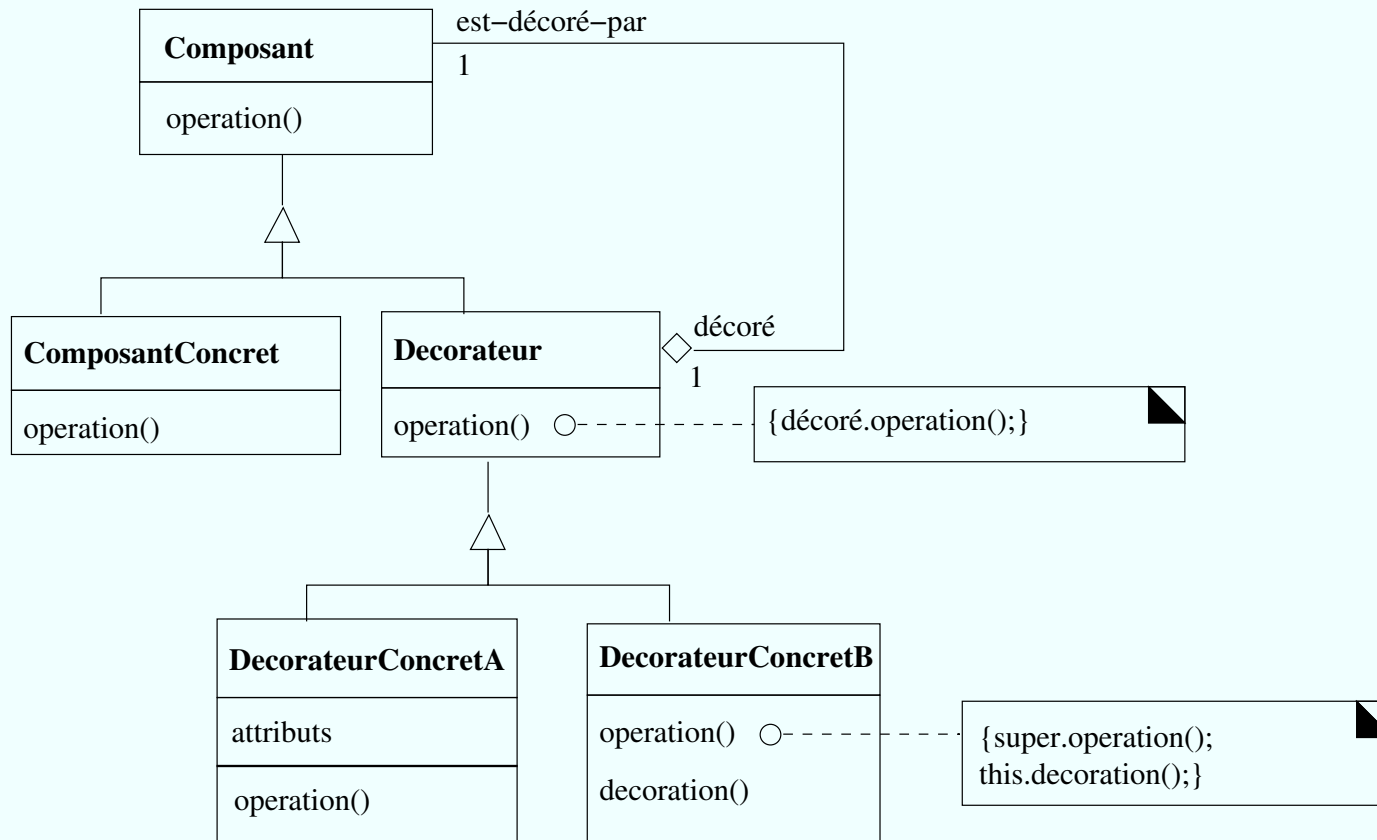


FIG. 4 – **Composant** : objet métier quelconque à décorer, exemple : `textView`. **Décorateur** : objet décorant (donc ajoutant des fonctionnalités) à un objet métier, exemple : `scrollDecorator`).

3.4 Une implantation de “Décorateur”

Une implantation de “décorateur” sur un exemple concret de composants visuels.

```
public class Decorateur extends ComposantVisuel {
    ComposantVisuel décoré;

    public Decorateur(ComposantVisuel c){
        //affectation polymorphique
        décoré = c;}

    public void draw(){décoré.draw();}
}
```

```
public class BorderDecorator extends Decorateur {
    //ajoute une bordure à un composant visuel

    float largeur; //largeur de la bordure

    public BorderDecorator(ComposantVisuel c, float l) {
        super(c);
        largeur = l;
    }

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

    public void drawBorder() {
        // dessin de la bordure
        ...}
}
```

3.5 Décorateur : discussion

- 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
- Ordre des décorations

4 Un exemple de Schéma structurel : Adapteur

But

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

Participants

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 5 et 6

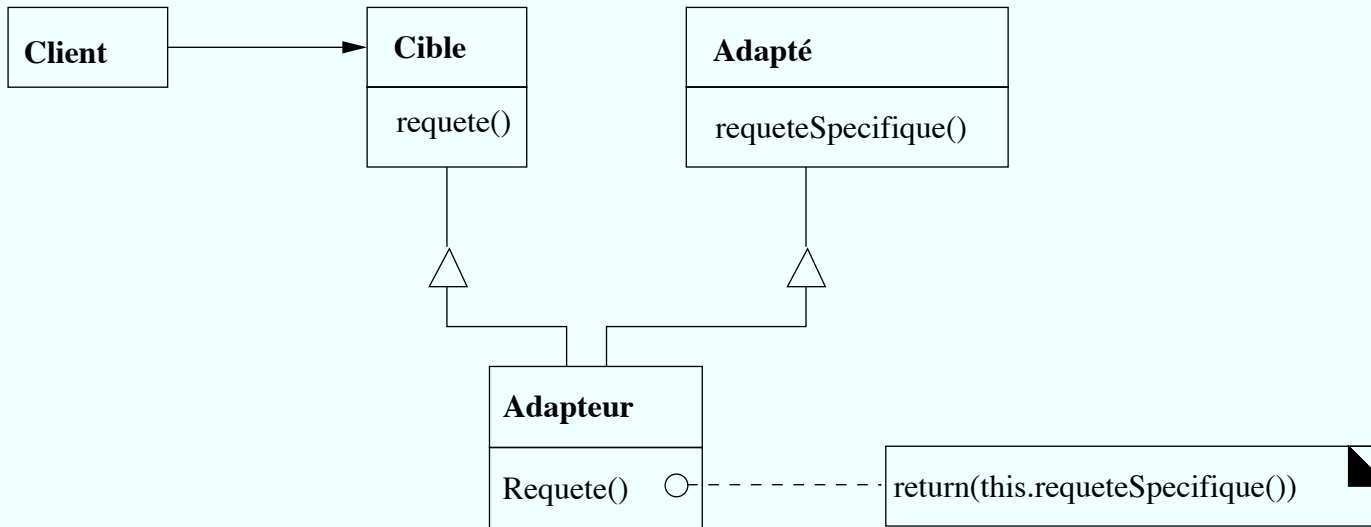


FIG. 5 – Adaptation par spécialisation

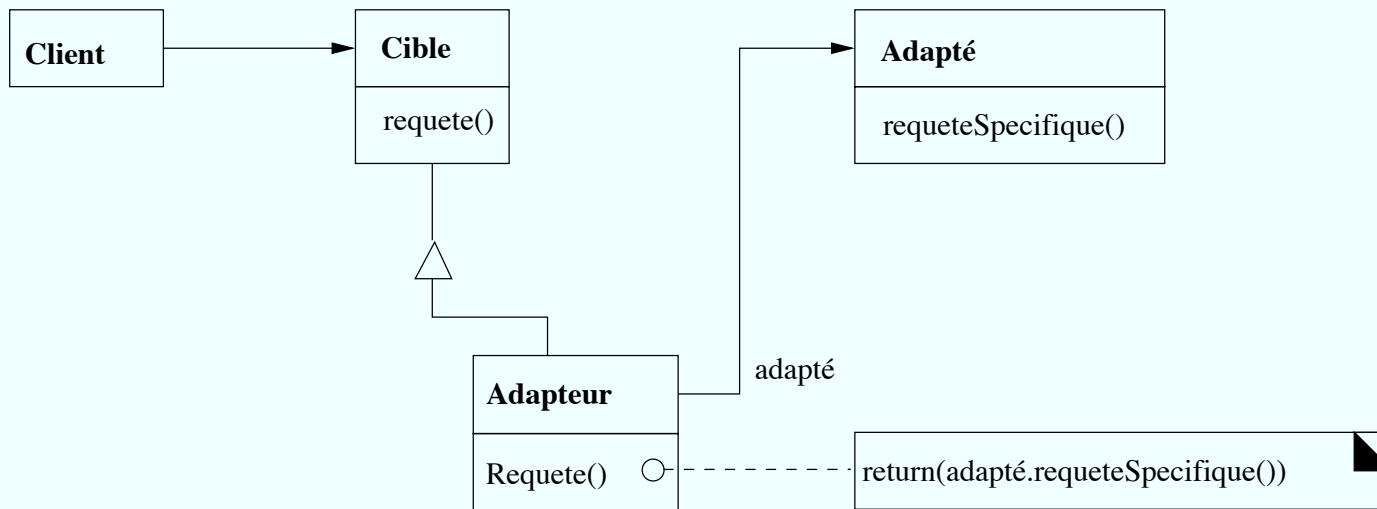


FIG. 6 – Adaptation par composition

4.2 Application

Application typique : Intégrer dans une application en cours de réalisation une classe définie par ailleurs (cf. fig. 7).

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 souhaite réutiliser une classe `textView` (l'objet à adapter) définie par ailleurs.

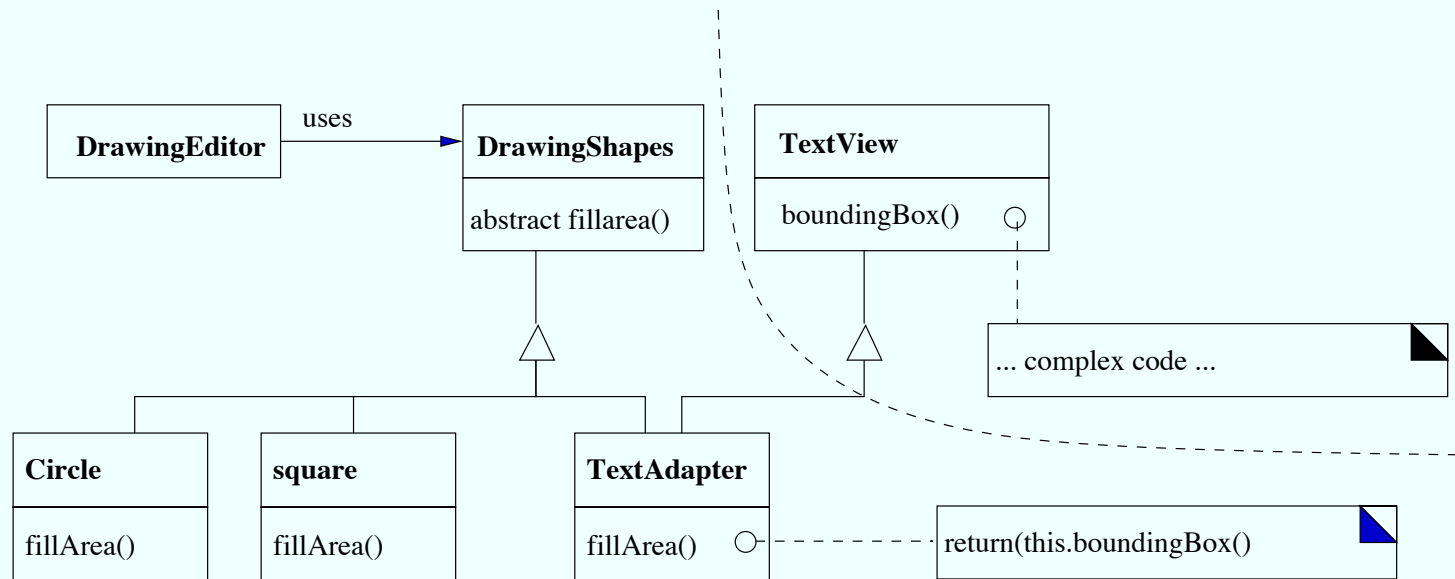


FIG. 7 – Exemple d'adaptation par spécialisation

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 pour simuler l’héritage multiple :
 1. Nécessité de redéfinir toutes les méthodes publiques de la classe du composite.
 2. **Perte du receveur initial** (cf. fig. 8). Lors de la redirection vers le composite, le receveur est perdu, ceci rend certains schémas de réutilisation difficiles à appliquer.

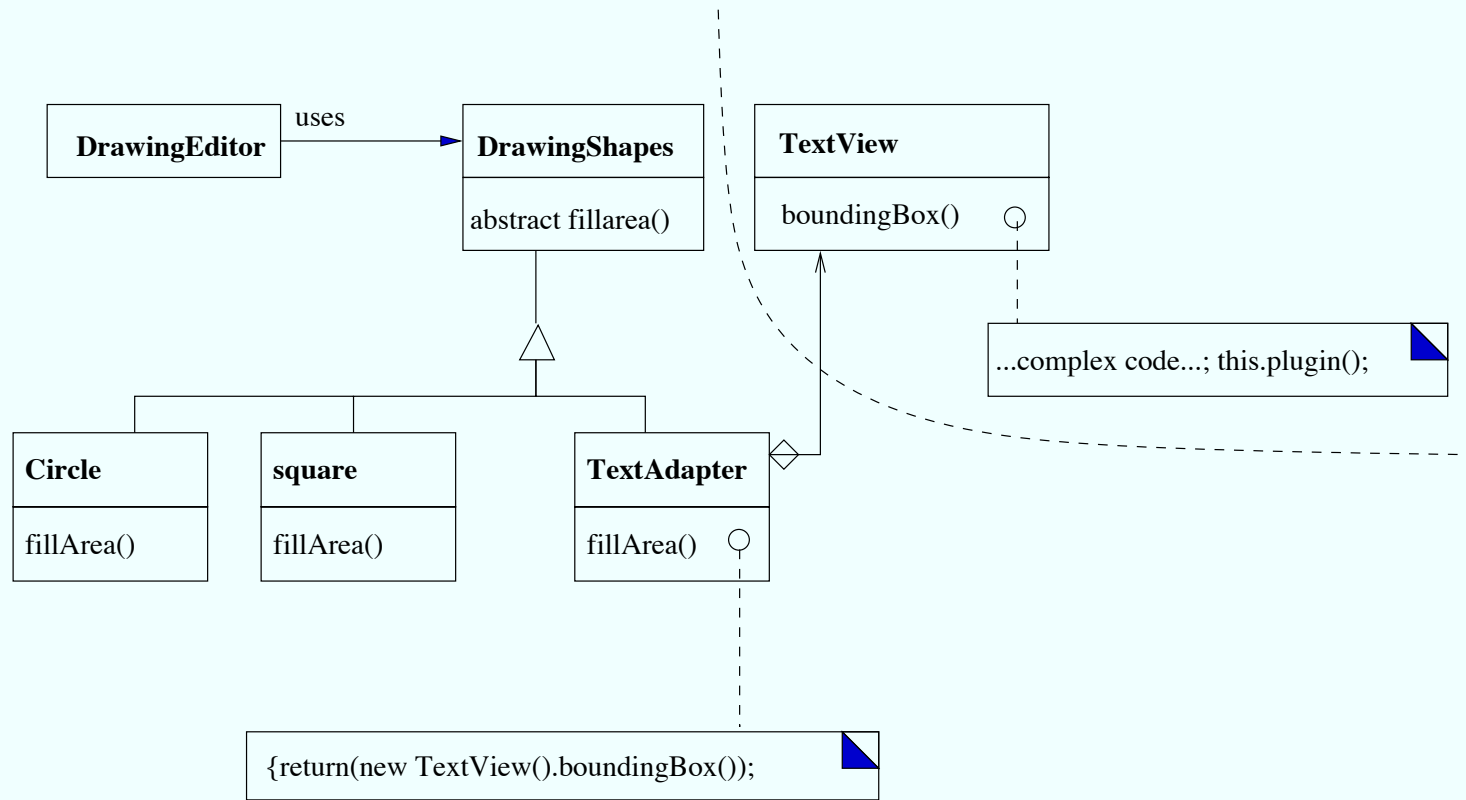


FIG. 8 – Adaptation par composition dans l'exemple de l'éditeur de dessin. Perte du receveur initial dans la méthode `boundingBox()`. Ceci rend en l'état impossible une spécialisation dans l'application de la méthode `plugin` de la classe adaptée `TextView`. La figure (cf. fig. 9) propose un schéma global de solutionnement de ce problème.

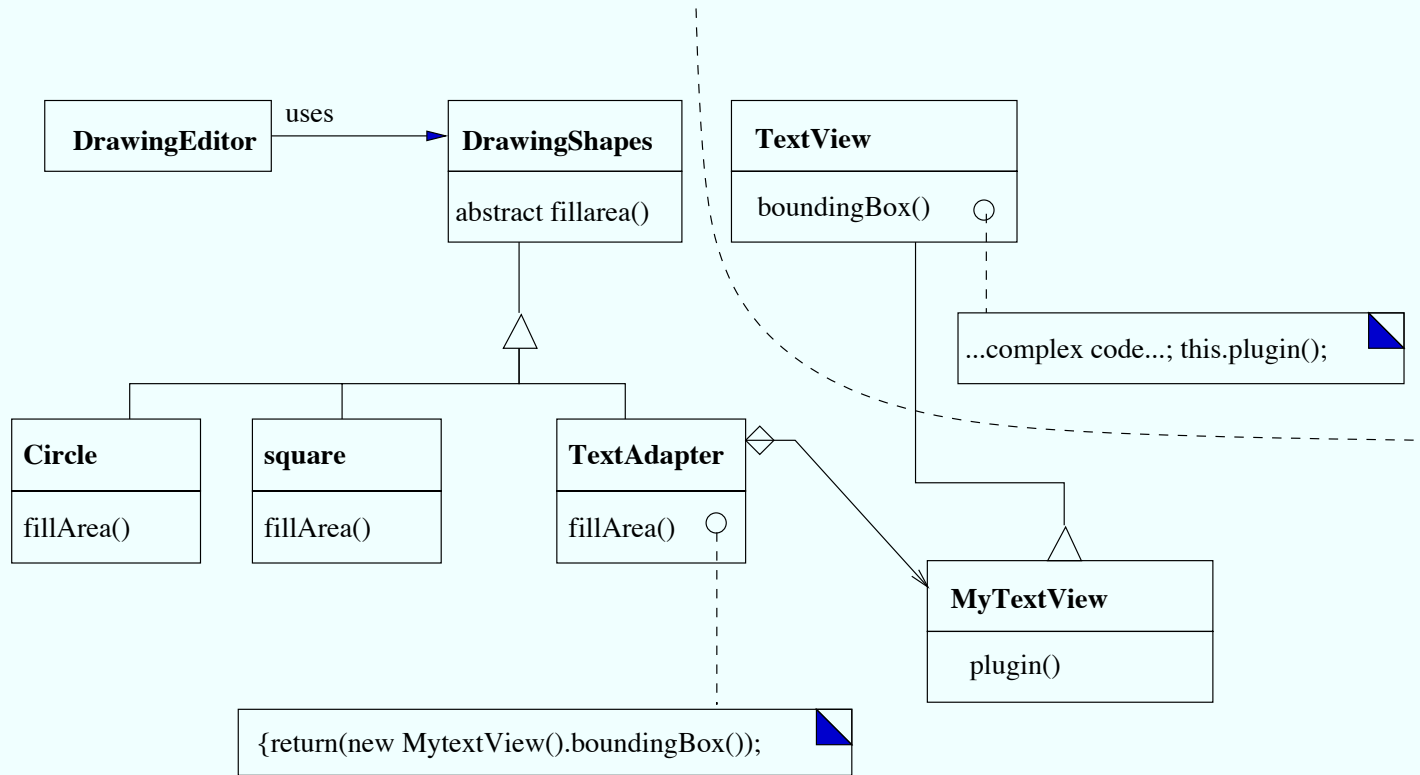


FIG. 9 – Adaptation par composition dans l'exemple de l'éditeur de dessin selon un modèle offrant une solution au problème de perte du receveur initial.

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. 10) : **Bridge** utilise la composition pour séparer une hiérarchie de concepts de différentes hiérarchies représentant différentes implantations de ces concepts.

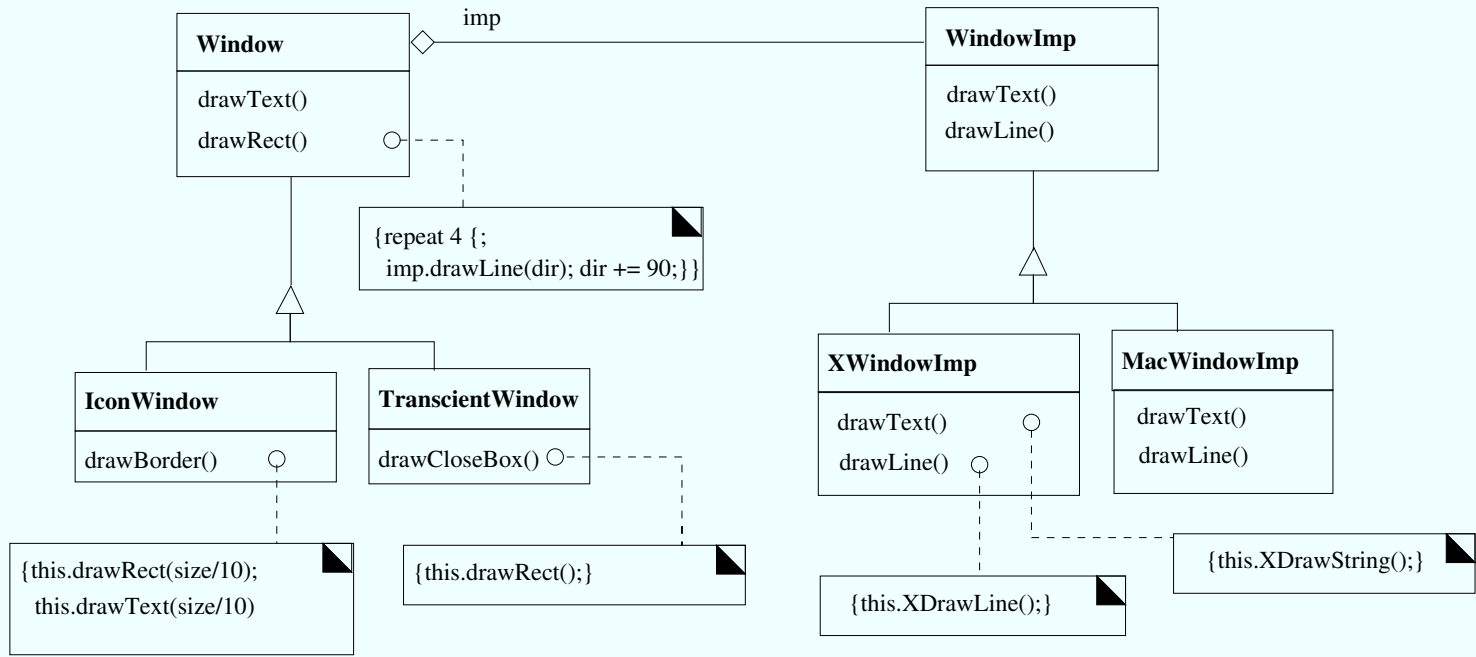


FIG. 10 – Application du pattern “Bridge”

5.2 Discussion

- **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.
- Il est possible de changer une implantation (recompilation) sans que les clients n'en soient affectés.
- **Bridge** est utilisable dans tous les langages, par exemple c++ ou Smalltalk ne proposant pas la notion d'interface.
- En autorisant la création de méthode dans les hiérarchies de concept, **Bridge** va plus loin que ce qu'autorisent les interfaces Java.

6 Schéma comportemental : “State”

6.1 Principe

Le schéma “State” propose une architecture permettant à un objet de changer de comportement quand son état interne change (cf. fig. 11).

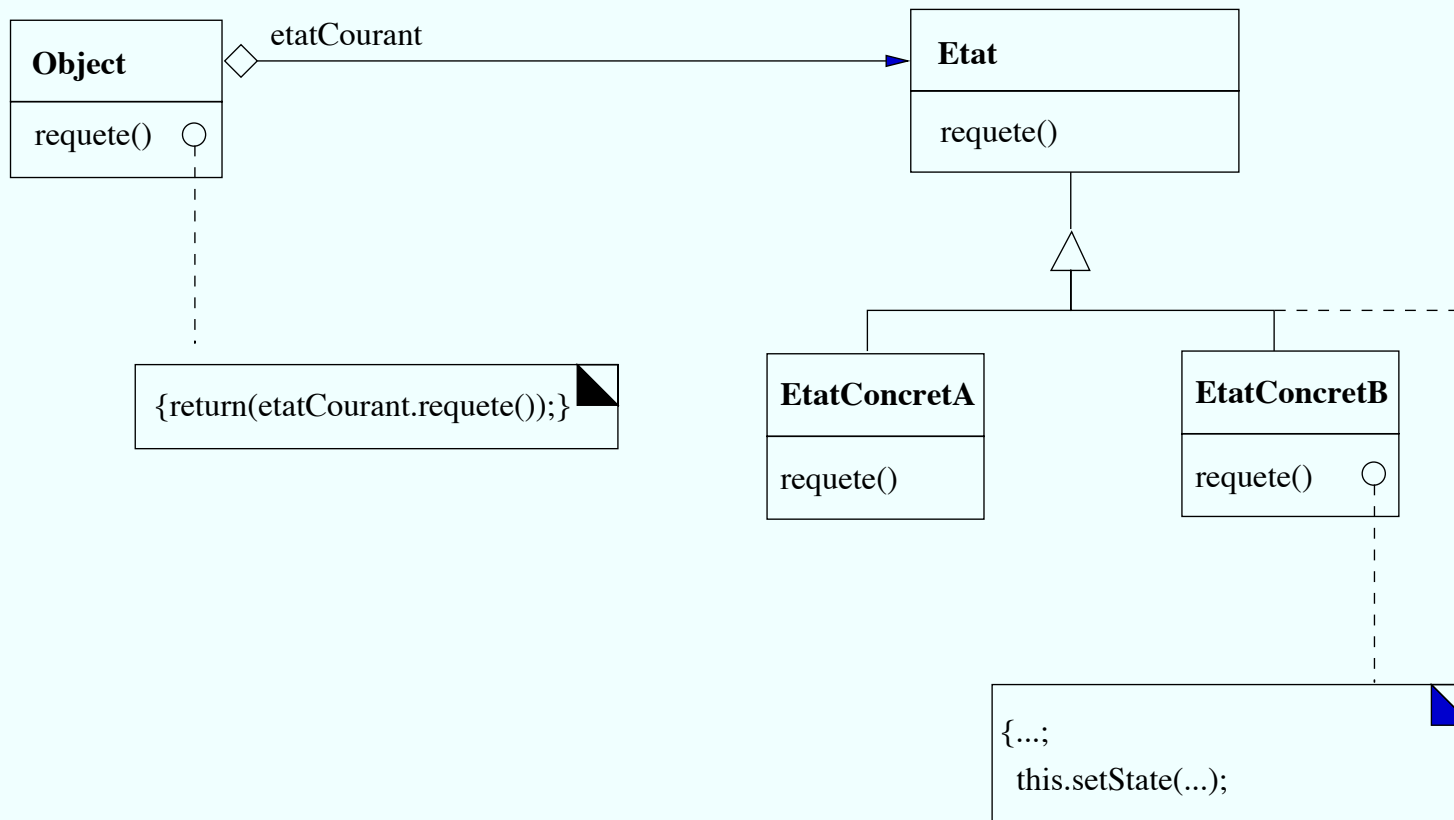


FIG. 11 – Principe général du pattern “State”

6.2 Application typique : implantation d'une calculatrice

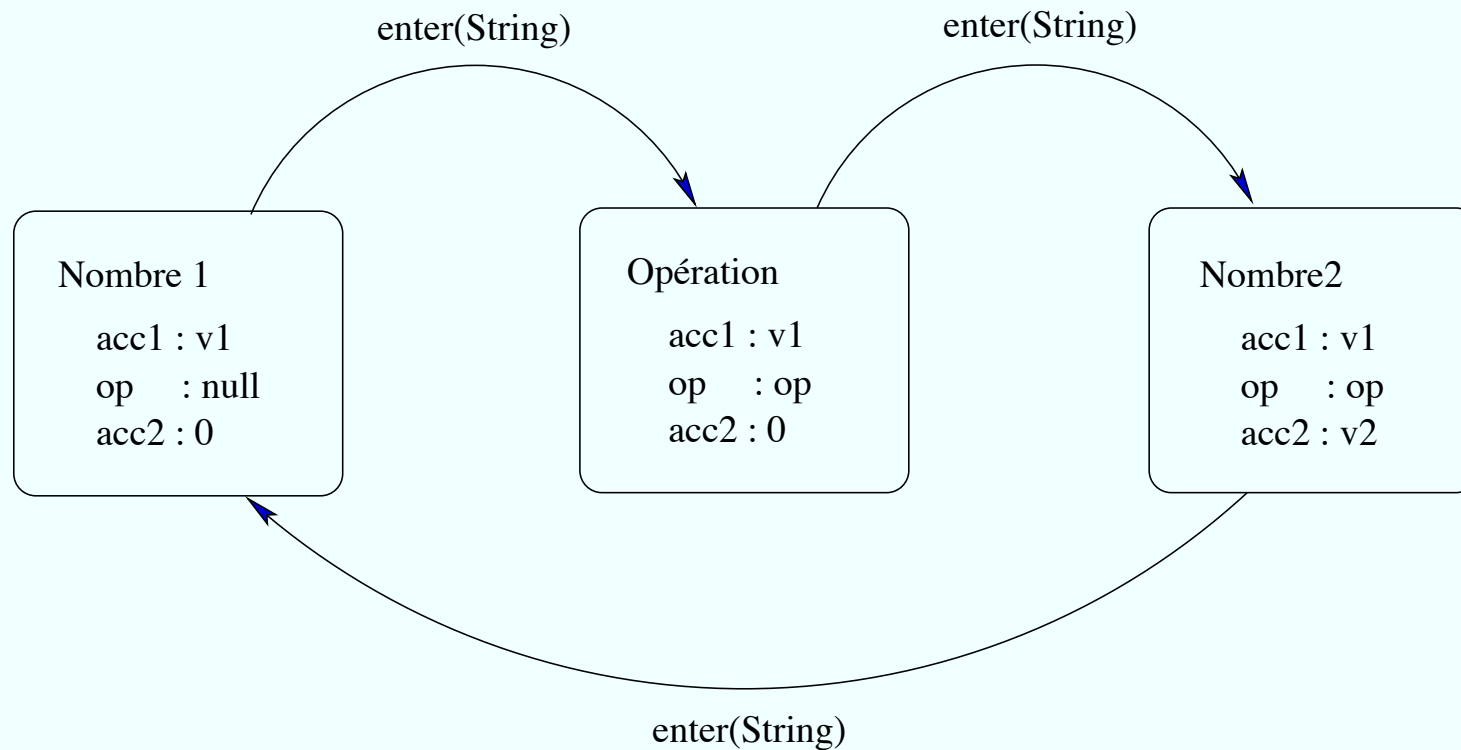


FIG. 12 – Cas d'applicabilité du pattern "State"

6.3 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 classe : applications pour la sécurité (contrôle du bon état interne des objets) et la réutilisation (exercice : passer d'une calculette infixée à une postfixée).
- Amélioration possible des implantations en utilisant les énumérations.

Gestion des états et distribution des calculs :

```
public class Calculette {
    protected EtatCalculette etatCourant;
    protected EtatCalculette[] etats = new EtatCalculette[3];
    double acc;
    String operateur;

    public Calculette(){
        etats[0] = new ENombre1(this);
        etats[1] = new EOperateur(this);
        etats[2] = new ENombre2(this);
        etatCourant = etats[0];
        acc = 0;
    }

    double getAcc(){return acc;}
    void setAcc(double v){acc = v;}
    String getOp(){return operateur;}
    void setOp(String v){operateur = v;}

    public void enter(String s) throws CalculetteException{
        etatCourant = etats[etatCourant33.enter(s) - 1];
    }
}
```

Les états sont invisibles aux clients de la classe `Calcullette` :

```
public static void main(String[] args){
    Calcullette c = new Calcullette();
    c.enter("123"); //etat 1 : stocke le nombre 123 dans acc
    c.enter("plus"); //etat 2 : stocke l'operation a effectuer dans un registre
    c.enter("234"); //etat 3 : stocke le résultat de l'opération dans acc1
    System.out.println(c.getResult());}
```

Classe abstraite de factorisation pour les différents états :

```
abstract class EtatCalculette {  
    static protected enum operations {plus, moins, mult, div};  
    abstract int enter(String s) throws CalculetteException;  
    Calculette calc;  
  
    EtatCalculette(Calculette c){ calc = c; }  
}
```

Classe concrète, état initial :

```
public class ENombre1 extends EtatCalcullette{

    ENombre1(Calcullette c) { super(c); }

    public int enter(String s) throws CalculletteException {
        try{calc.setAcc(Float.parseFloat(s));}
        catch (NumberFormatException e)
            {throw new CalculletteException("Entrez un nombre svp!");}

        return(2);}
}
```

Classe concrète, état de saisie de l'opération :

```
public class EOperateur extends EtatCalcullette{
    EOperateur(Calcullette c){ super(c); }
    public int enter(String s) throws CalculletteException {
        calc.setOp(s);
        return(3);}
}
```

Classe concrète, état de l'application de l'opération aux opérandes :

```
public class ENombre2 extends EtatCalcullette {
    ENombre2(Calcullette c){super(c);}

    int enter(String s) throws CalculletteException {
        float temp = 0;
        try {temp = Float.parseFloat(s);}
        catch (NumberFormatException e) {
            throw new CalculletteException("Entrez un nombre svp!");}

        switch (operations.valueOf(calc.getOp())) {
        case plus:
            calc.setAcc(calc.getAcc() + temp);
            break;
        case mult:
            calc.setAcc(calc.getAcc() * temp);
            break;
        default:
            System.out.println("Opérateur inconnu");
            break;}
        return (1);}
}
```

7 Le schéma comportemental : “Observateur”

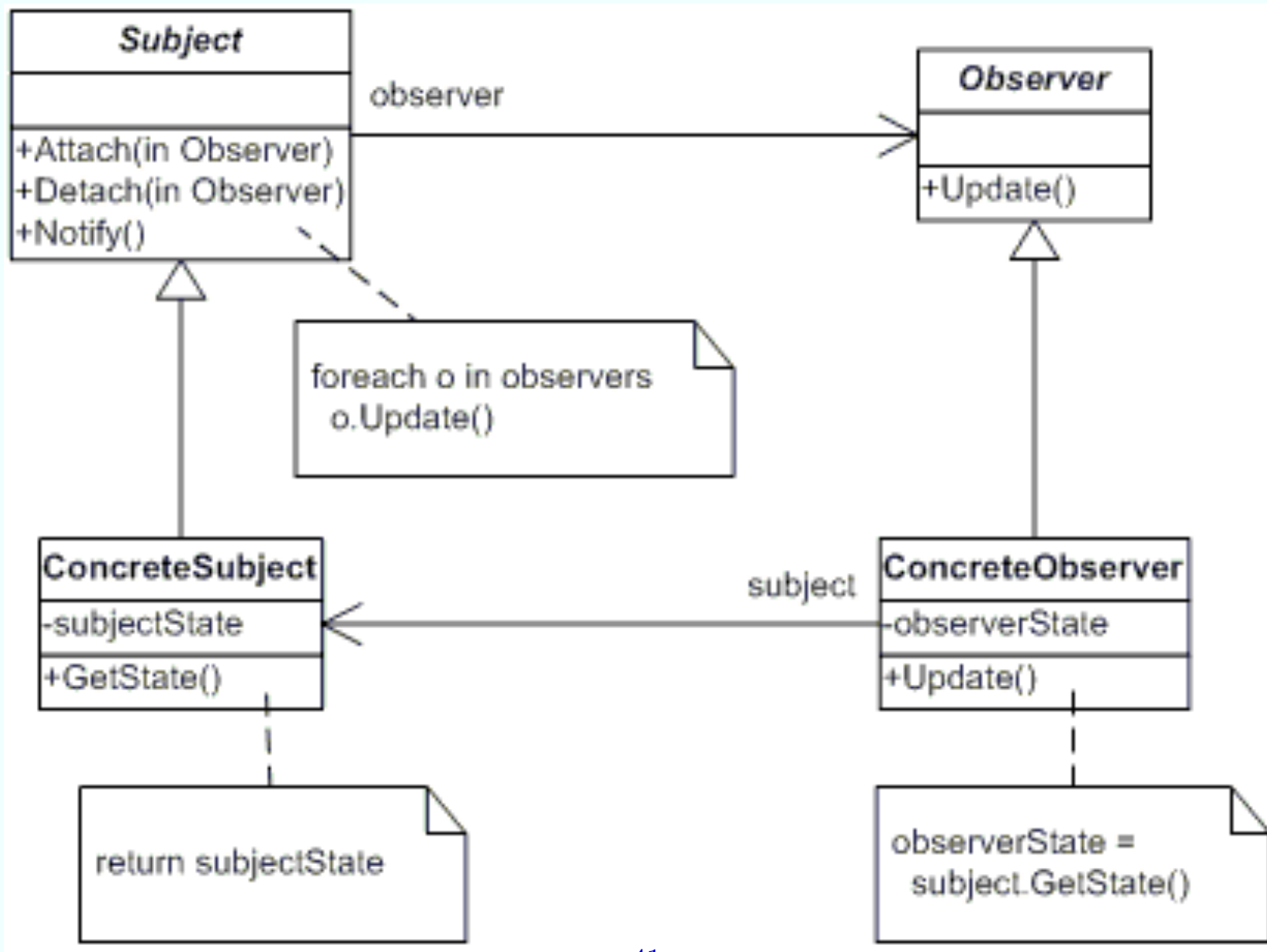
7.1 But

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 13)



41

FIG. 13 – Observer : Collaboration “Observateurs-Observé”

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 implantation typique dans le framework MVC

Le schéma MVC (Krasner, Pope 1977), implanté initialement dans le langage Smalltalk a été une des premières applications de ce qui ne s'appelait pas encore le schéma de conception "observateur". Il sert, dans les applications dotées d'interfaces graphiques, à séparer le code des modèles de celui de leur visualisation.

Plus précisément, il permet de :

- Découpler un objet de son interfaçage graphique (sa vue)
- Découpler le contrôle (souris, clavier) de l'affichage
- d'avoir plusieurs vues pour un même objet.

Son architecture est également celle d'un framework. Une implantation ainsi qu'une application qui en dérive sont données dans la section suivante.

7.5 Architecture

Trois hiérarchies de classes : Les modèles, les vues et les contrôleurs.

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

Chaque contrôleur connaît son modèle.

Chaque vue connaît son contrôleur et son modèle.

Le modèle ne connaît personne, il est observé par la vue et par le contrôleur.

7.6 Modèle d'interaction (figure 14)

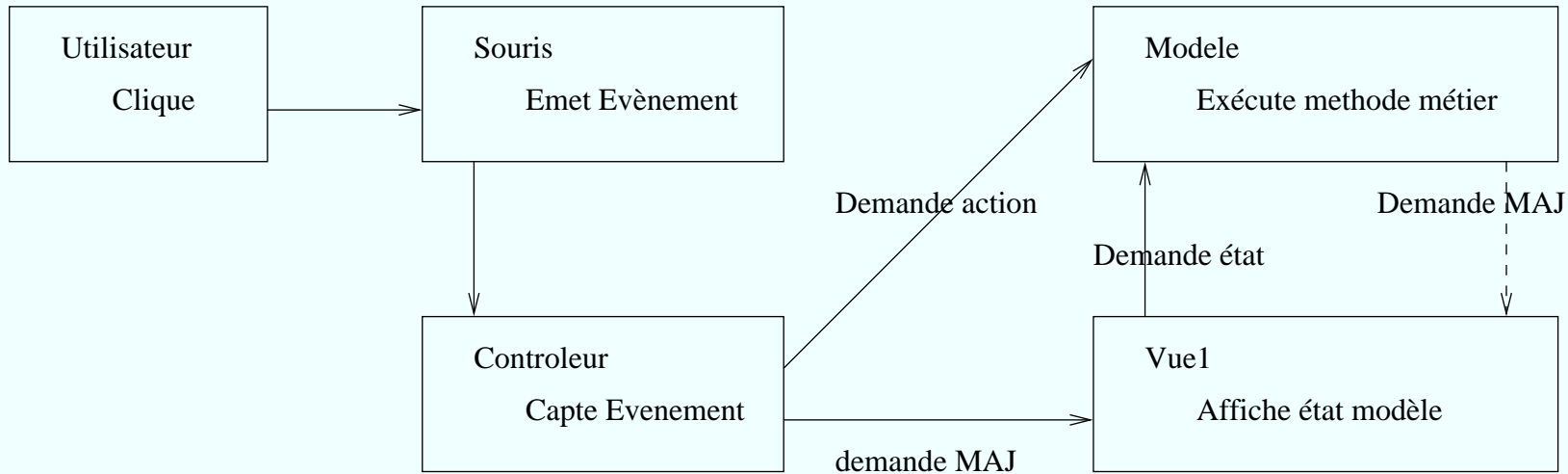


FIG. 14 – Diagramme d'interaction entre modèles, vues et contrôleurs du MVC

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

7.7.1 Gérer les dépendances “observateur-observé”

Classe du framework

```
public class MV-Association{
    //utilise un dictionnaire pour stocker les couples model-controller
    static Dictionary MVDictionary = new Hashtable();

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

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

7.7.2 Les modèles

Classe du framework

```
public class Model{  
  
    public void changed(Object how){  
        Iterator i = MV-Association.getViews(this).iterator();  
        while (i.hasNext())  
            (i.next()).update(how);}  
}
```

Classe de l'application

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

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

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

7.7.3 Les vues

Classe du framework

```
public class View{
    Controller cont;
    Model model;

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

    public abstract void update (Object how);
    public void open(){...}
    public void redisplay(){...}

}
```

Classe de l'application

```
public class CompteurView extends View{  
    ...  
    private JLabel l;  
    ...  
  
    public void update(Object how){  
        l = new JLabel(String.valueOf(m.getValeur()), JLabel.CENTER);  
        this.redisplay();  
    }  
}
```

7.7.4 Les contrôleurs

Classe du framework

```
public abstract class controller implements ActionListener{
    Model m;

    public Controller(Model m){
        model = m;
    }
}
```

Classe de l'application

```
public class CompteurController extends controller implements ActionListener{

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

7.7.5 Lancement de l'application

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