Université Montpellier-II
UFR des Sciences - Département Informatique
Master Informatique - UE FMIN 107 -

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

Partie No 2 : Les Schémas de Conception ou la transmission du savoir logiciel

Notes de cours - 2007-2014
Christophe Dony

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://c2.com/cgi/wiki?DesignPatternsBook
- les schémas du livre en ligne : http://www.oodesign.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 chémas 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 variés pour réaliser des paramétrages que l'on peut classifier de façon plus abstraite.

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

Figure *(1)* – *Classification extraite de [GHJV 94]*

1.3 Gestion automatisée des design patterns

— Foutse Khomh. Patterns and quality of Object-oriented Software Systems. 2010.
— Guéhéneuc Yann-Gaël Khashayar Khosravi. A Quality Model for Design Patterns. 2004.
— Carl G. Davis Jagdish Bansiya. A Hierarchical Model for Object-Oriented Design Quality Assessment. 2002.
— Yann-Gaël Guéhéneuc Hervé Albin-Amiot, Pierre Cointe. Un méta-modèle pour coupler application et détection des design patterns. 2002.
— Tu Peng Jing Dong, Yajing Zhao. Architecture and Design Pattern Discovery Techniques. 2007.
— Jakubik Jaroslav. Extension for Design Pattern Identification Using Similarity Scoring Algorithm. 2009.
— Marcel Birkner. Objected-oriented design pattern detection using static and dynamic analysis in java software. 2007.

1.4 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.

1.5 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;
3
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 sroll”, “bordure”) à un objet graphique (Figure 2).

3.2 Exemple Type : décoration d'une "textView"

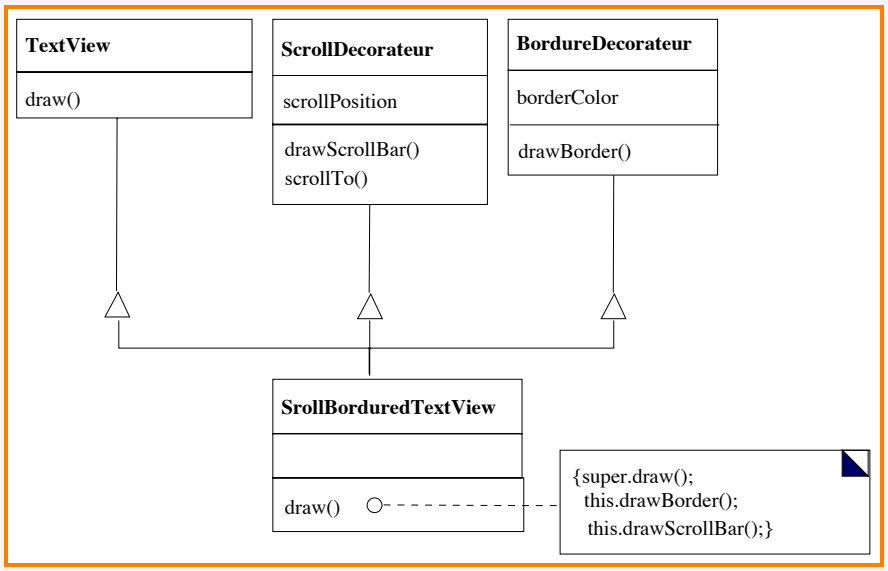

Figure *(2)* – *Décoration d'une* `textview`*, solution universelle avec héritage multiple. Limitations : a) statique b) autant de sous-classes que de combinaisons potentielles de décorations.*

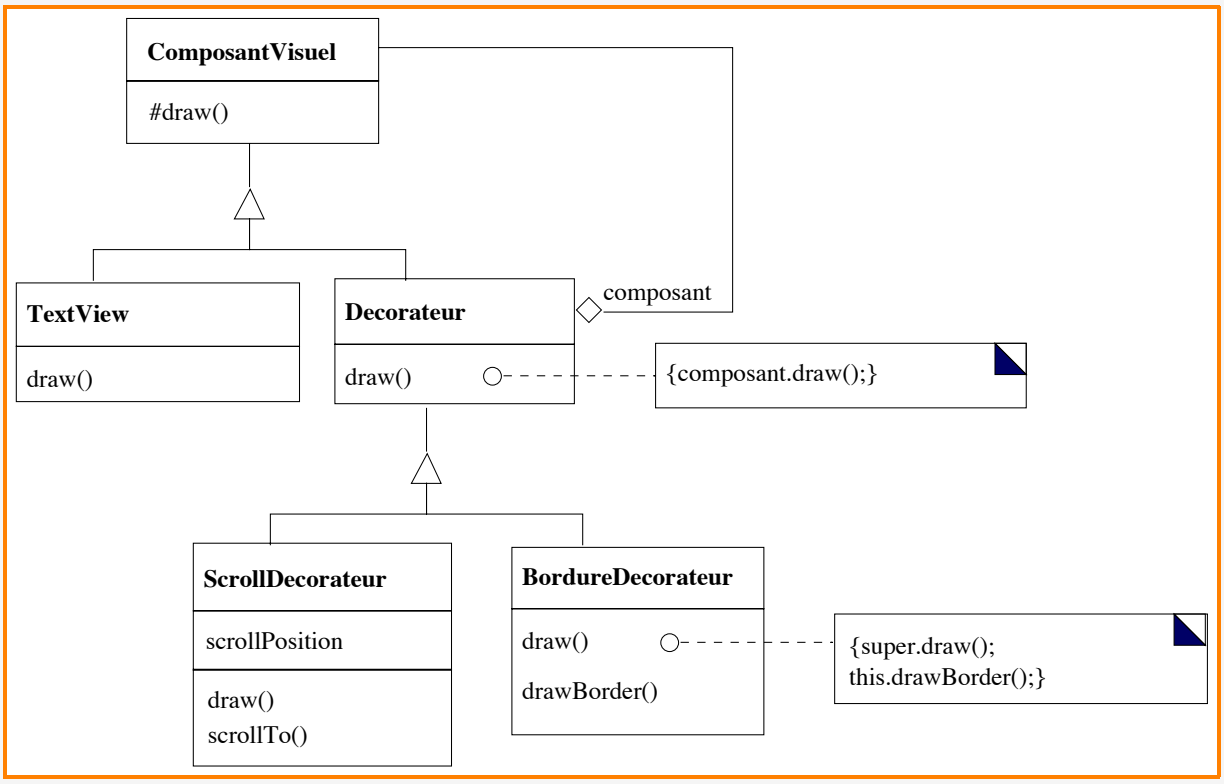

Figure *(3)* – *Décoration d'une* `Textview` *: solution avec le schéma "Decorateur"*

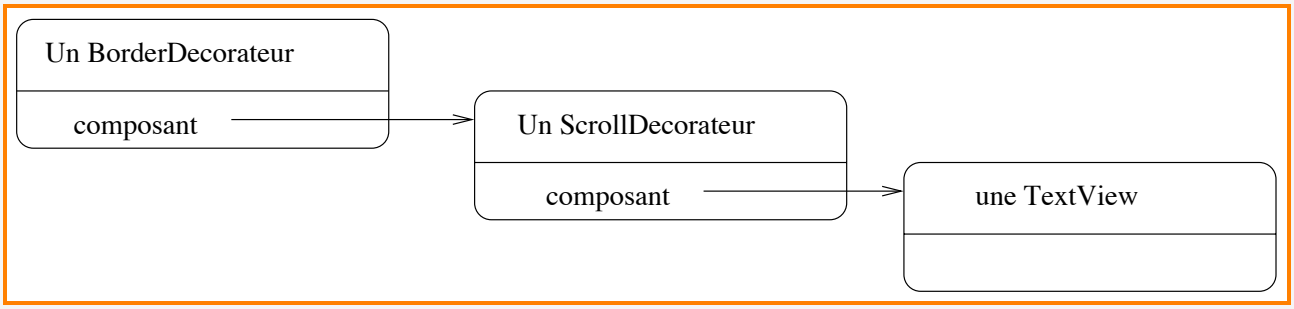

Figure *(4)* – *Objets représentant une* `Textview` *décorée selon le schéma "Decorateur".*

3.3 Principe Général de la solution (figure 5)

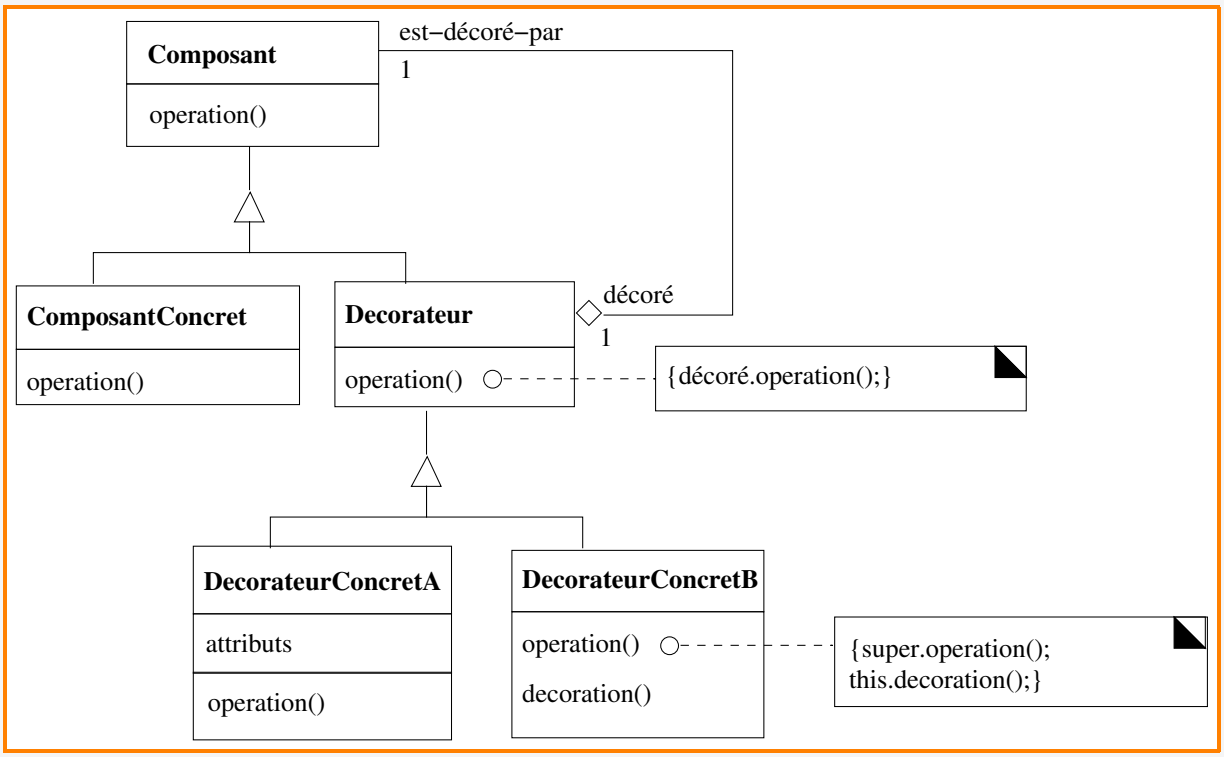

Figure *(5)* – **Composant** *: objet métier quelconque à décorer, exemple :* `textView`*.* **Décorateur** *: objet décorant (donc ajoutant des fonctionnalités) à un objet métier, exemple :* `scrollDecorator`*).*

Intuition :

```
Stream filteredMemoryStream = new StreamReader(new StreamFilter(new MemoryStream()));
```

3.4 Une mise en oeuvre concrète

Une implantation de "décorateur" sur un exemple jouet de composants graphiques (classe `Composant`).

```
1 public class Decorateur extends Composant {
2     Composant décoré;
3
4     public Decorateur(Composant c){
```

```
        //injection de dépendance
        //affectation polymorphique
        décoré = c;}

    public void draw(){
        //redirection de message
        //''double dispatch''
        décoré.draw();}
}
```

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

    float largeur; //largeur de la bordure

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

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

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

3.5 Discussion

— Commentaire, un décorateur peut être vu comme un composite avec un seul composant. Avec la différence que le décorateur a des fonctionnalités propres que n'a pas son composant.
— Problème, 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
— Problème, 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".
Une solution à ce problème est de modifier le pattern en remplaçant la classe abstraite `Composant` par une interface donc le lien *sous-classe-de* entre `Décorateur` et `Composant` par un lien *implémente*.
— Problème, incompatibilité potentielle de différentes décorations.
— L'ordre d'ajout des décorations est signifiant.
— Localisation du *double dispatch*

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

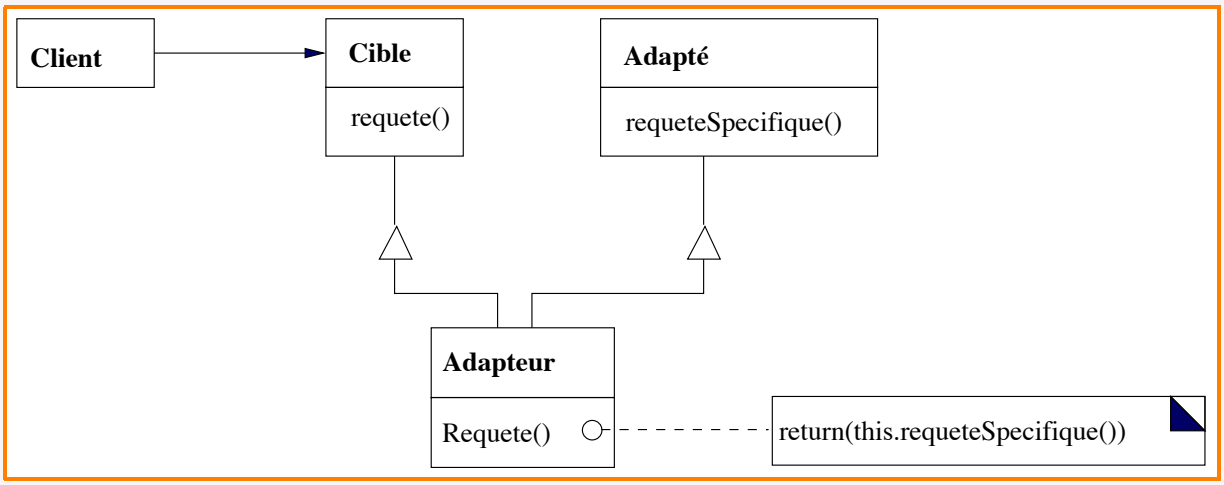

Figure ***(6)*** – *Adapteur réalisé par spécialisation*

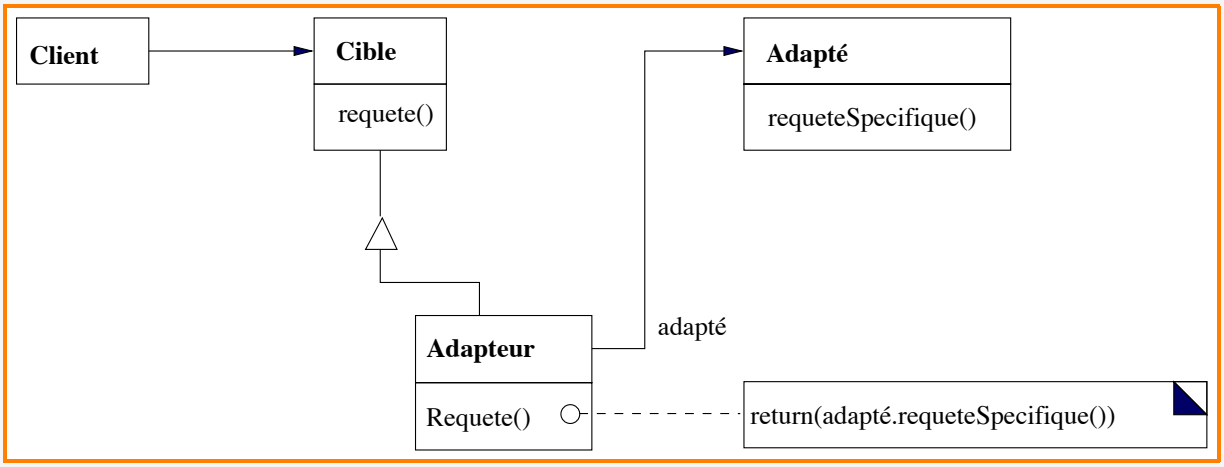

Figure ***(7)*** – *Adaptateur réalisé par composition*

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.

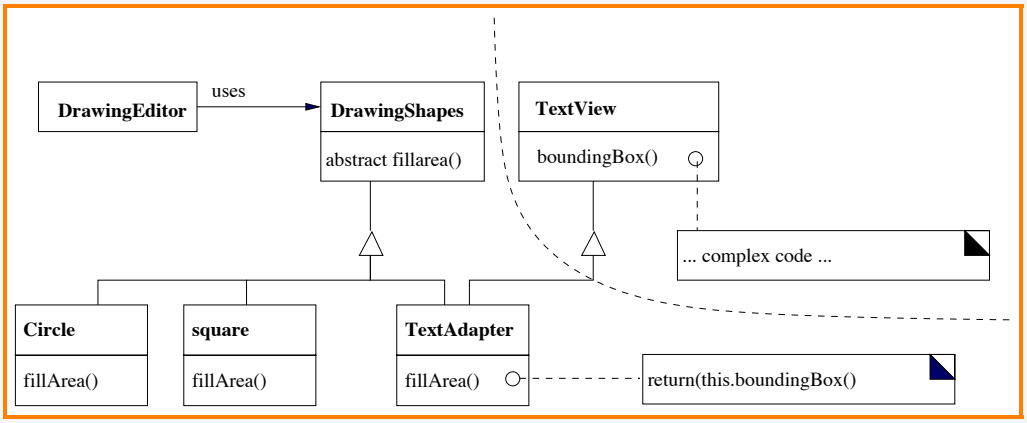

Figure ***(8)*** – *Exemple d'adaptation par spécialisation*

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.

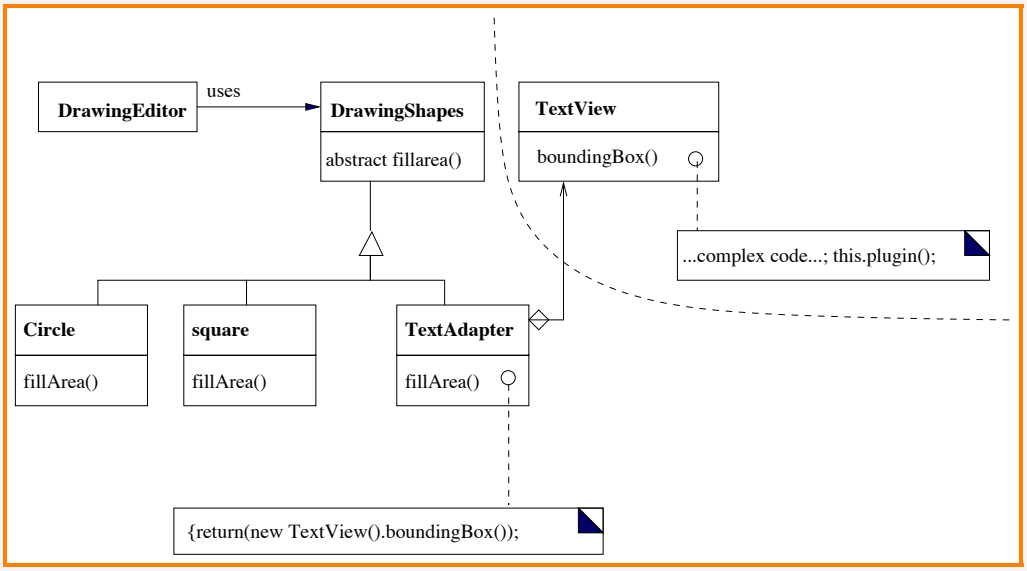

Figure ***(9)*** – *Adaptation par composition dans l'exemple de l'éditeur de dessin. Perte du receveur initial dans la méthode* `boundingBox()`*. Ceci rend en l'état impossible la prise en compte d'une spécialisation dans l'application de la méthode* `plugin` *de la classe adaptée* `TextView`*. La figure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

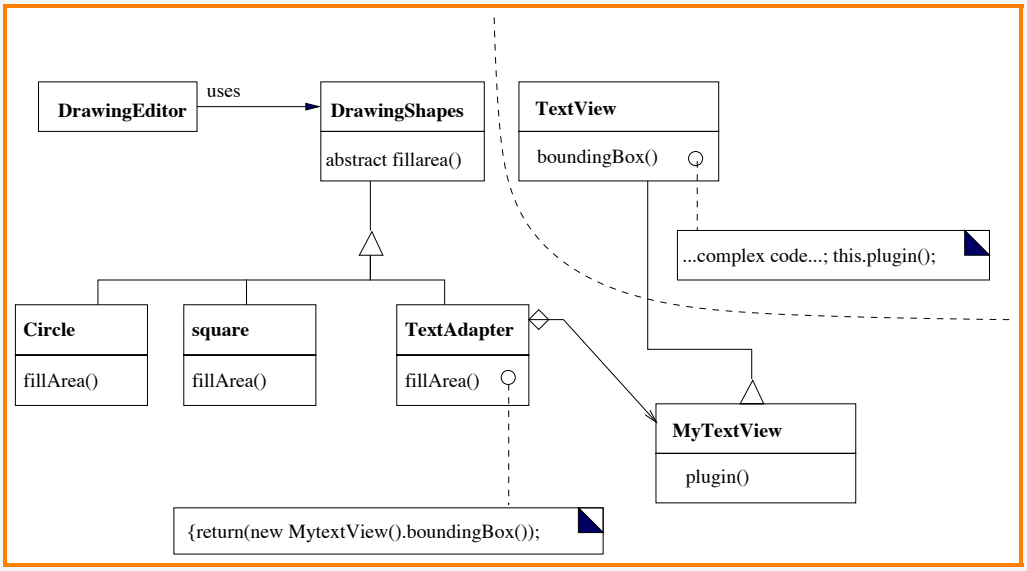

Figure ***(10)*** – *Adaptation par composition dans l'exemple de l'éditeur de dessin selon un modèle offrant une solution au problème de perte du receveur initial.*

4.4 Généralisation du problème de perte du receveur initial

Le problème de perte du receveur initial apparait plus globalement dans toute mise en oeuvre d'une forme d'héritage multiple utilisant la composition.

Il n'est pas inintéressant de comparer la perte du receveur en composition et en héritage sans liaison dynamique (héritage non virtuel de C++) ou à l'"embarquement de type" (`embedded` type) du langage `GO`.

```
1  package main
2  import "fmt"
3
4  type Widget struct {
5      X, Y int
6  }
7  type Label struct {
8      Widget // Embedding (matérialisé par un attribut sans nom)
9      Text string // Aggregation
10 }
11 func (label Label) KindOf() string{
12      return "Label"
13 }
14 func (label Label) Paint() {
15     //fmt.Printf("Un label:%p\n", &label)
16     fmt.Printf("un %q : %q %d %d \n", label.KindOf(), label.Text, label.X, label.Y)
17 }
```

Adapted from http ://www.drdobbs.com/open-source/go-introduction-how-go-handles-objects/240005949

```go
type Button struct {
    Label // Embedding (matérialisé par un attribut sans nom)
}
func (button Button) Click() {
    fmt.Printf("%p:Button.Click\n", &button)
}
func (button Button) KindOf() string {
     return "Button"
}
func main() {
     label := Label{Widget{10, 10}, "Test:"}
     label.Paint()
     fmt.Printf("SUITE\n")
     button1 := Button{Label{Widget{10, 70}, "OK"}}
     button1.Click() //0x2081c2020:Button.Click
     button1.Paint() //un "Label" : "OK" 10 70 ... liaison statique ... perte du receveur initial
     button1.Label.Paint() //un "Label" : "OK" 10 70 //ressemble à de la composition
     fmt.Printf("FIN\n")}
```

A button is defined as a clickable Label. Key point : calling "button1.Paint" gives same result that calling "button1.Label.Paint()" ... the receiver passed to the method is the Label field, not the whole Button

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.

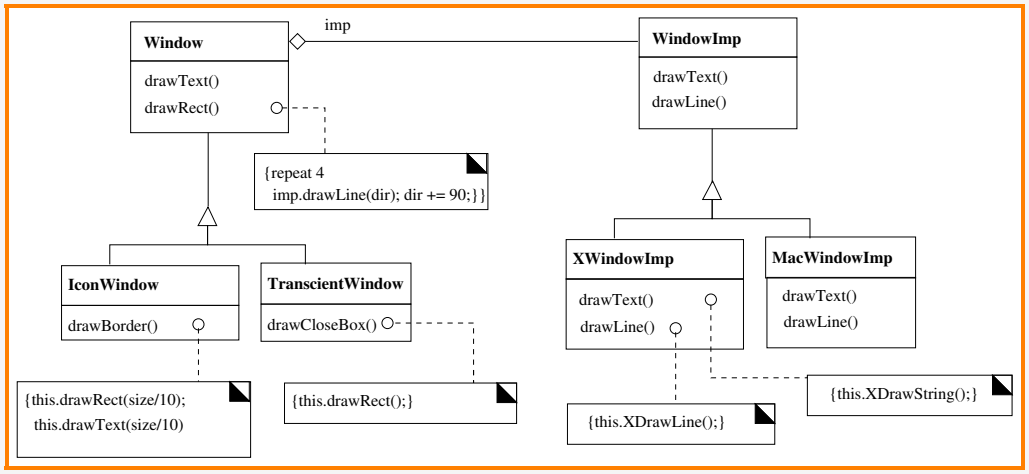

Figure *(11)* – *Exemple d'application du schéma "Bridge"*

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 mélant 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 difère 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).

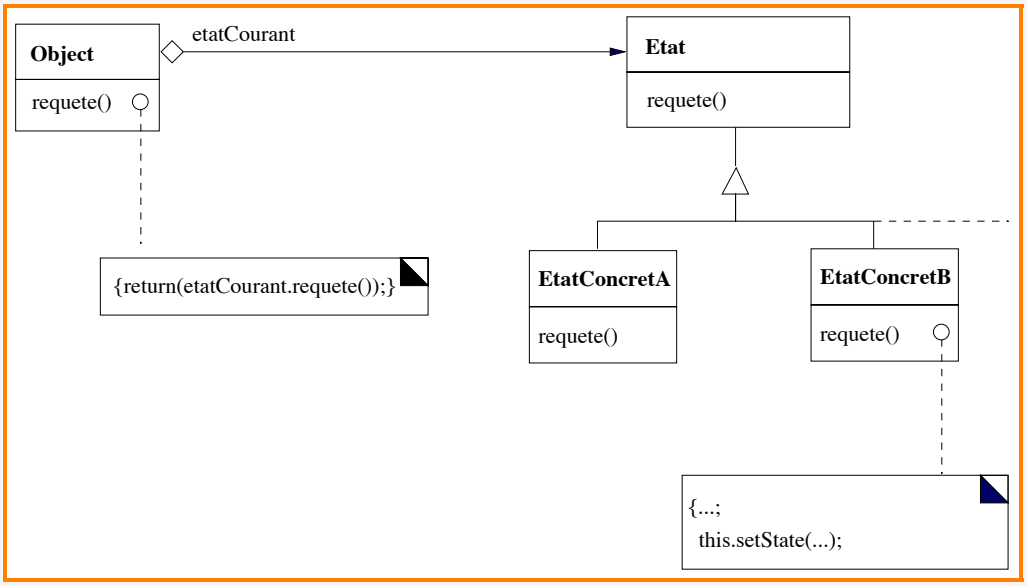

Figure *(12)* – *Pricipe général du schéma "State"*

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 calculette 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 referencé 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

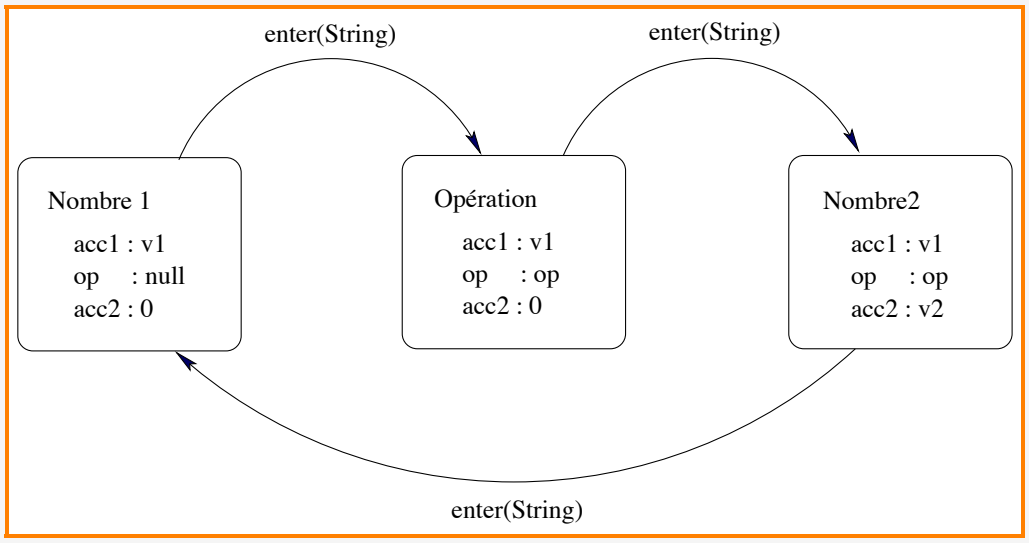

Figure ***(13)*** – *Les différents états et transitions d'une "calculatrice" basique*

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; }
13    ...
14    // accesseurs lecture/écriture pour ''accumulateur'' et pour ''operateur''
15    ...
16    //obtention du résultat
17    public double getResult() { return accumulateur; }
```

Distribution des calculs

```
1  public class Calculette {
2    ....

4     public void enter(String s) throws CalculetteException{
5       //toute requète est redirigée vers l'état courant
6       //ici c'est l'état courant qui décide quel est l'état suivant, ce n'est pas une règle générale
7        etatCourant = etats[etatCourant.enter(s) - 1]; }
```

Les états sont invisibles aux clients

```
public static void main(String[] args){
      Calculette c = new Calculette();
      c.enter("123"); //etat 1 : stocke le nombre 123 dans accumulateur
      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 accumulateur
      System.out.println(c.getResult());}
```

Classe abtraite de factorisation

```
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; }
}
```

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

```
public class ENombre1 extends EtatCalculette{

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

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

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

(une gestion des exception plus fine serait nécessaire) :

```
public class EOperateur extends EtatCalculette{
   EOperateur(Calculette c){ super(c); }
   public int enter(String s) throws CalculetteException {
      calc.setOp(s);
      return(3);} }
```

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

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

```
public class ENombre2 extends EtatCalculette {
   ENombre2(Calculette c){super(c);}

   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 : “Prototype” et les langages à Prototypes