Université Montpellier
UFR des Sciences - Département Informatique
Master Informatique - UE FMIN107 - UE HMIN 102 -

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

Partie No 2 : Réutilisation des Conceptions et des Architectures Logicielles

Notes de cours
Christophe Dony

1 Introduction

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/

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)

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.

Paramétrage, Variabilité

Les schémas offrent des solutions génériques paramétrées selon les schémas par spécialisation et composition présentés dans la première partie de ce cours.

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]*

1.1 Généralisation et Extension : l'ingénierie des "design patterns"

— Succès de l'idée : des schémas sur tous les sujets
Object-Oriented Reengineering Patterns Par Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz, 2002, Morgan Kaufmann, 282 pages, ISBN 1558606394
http://c2.com/ppr/

— Foutse Khomh. Patterns and quality of Object-oriented Software Systems. 2010.

— Guéhéneuc Yann-Gaël, Khashayar Khosravi. A Quality Model for Design Patterns. 2004.
http://www.yann-gael.gueheneuc.net/Work/Research/PatternsBox/Introduction/

— 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.

1.2 Classification des schémas

— **Schémas créateurs** : Décrire des solutions pour la création d'objets,

— **Schémas structuraux** : Décrire des solutions 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` si les booléens sont des objets (*Smalltalk*, *Javascript*, *Python*, ...) .

Solution "tout langage"

```
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, et empêche un ''new'' externe.*/
```

```
6       private Singleton() {}

8       /** ''synchronized'' sur la méthode de création
9        * empêche toute instanciation simultanée par différents threads.
10       * Rend l'unique instance de la classe,
11       * sauvegardée dans un attribut de classe (statique) */

13      public synchronized static Singleton getInstance() {
14          if (INSTANCE == null)
15              INSTANCE = new Singleton();
16          return INSTANCE;}
17      }
```

Listing (1) – *Une classe Singleton en Java.*

Discussions

1. Commentaire : empêcher les instantiations par les clients : visibilité du constructeur,

```
1    private Singleton() {}
```

2. Commentaire : Empêcher les copies
   - (Si C++) - déclarer sans le définir le constructeur par copie)

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

   - (Si Java) - ne pas implanter l'interface Cloneable.

3. Alternative : (si C++) surcharger *new* ,

```
1  class iSingleton{
2    static iSingleton *INSTANCE;
3   public:
4      iSingleton() {
5         INSTANCE=this;
6       }
7
8      void *operator new(size_t s) {
9        if(INSTANCE != NULL)
10          return INSTANCE;
11       else
12          return malloc(s);
13 }
```

Listing (2) *– Surcharge de l'opérateur* `new` *en C++.*

4. Alternative : si ∃ métaclasse `Class` et si `new` est une méthode (exemples : *Pharo*, *Smalltalk*, *Python*, *Ruby*, ...), Redéfinir `new` sur chaque classe de classe Singleton :

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

Listing (3) *– Singleton,via une redéfinition de la méthode* `new`*. INSTANCE est toujours un attribut de classe (static)*

5. Alternative : si le langage permet de définir de nouvelles métaclasses (exemple : *Common-Lisp*), Définir une méta-classe `Singleton-Class`, instance et une sous-classe de `Standard-class`.

```
1  (defclass singleton-class (standard-class)
2    ((UNIQUE-INSTANCE :accessor GET-INSTANCE :initform nil))
3    (:metaclass standard-class))
```

Listing (4) *– Singleton, via une nouvelle méta-classe* `Singleton-class`*, la classe des classes qui ne peuvent avoir qu'une seule instance. Elle possède un attribut de méta-classe* `Unique-Instance`*. Version Common-Lisp*

```
1  (defmethod make-instance ((newC singleton-class) &rest args)
2    (or (GET-INSTANCE newC)
3        (let ((newI (call-next-method)))
4              (setf (get-instance newC) newI)
5              newI)))
```

Listing (5) *– Spécialisation de l'instantiation sur* singleton-class, (call-next-method') *réalise l'appel de la méthode masquée par la redéfinition (envoi de message à* `super`*).*

6. Exemple : Utilisation du schéma `Singleton` pour les classes `True` et `False`, possible discussion sur l'implantation des 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 modifier dynamiquement une fonctionnalité à un objet (individuel), sans modifier ni sa classe, ni donc les autres instances de sa classe.

Application possible : Ajouter des décorations ("barre de scroll", "bordure") à un objet graphique (Figure 2).

3.2 Exemple Type : décoration d'une "textView"

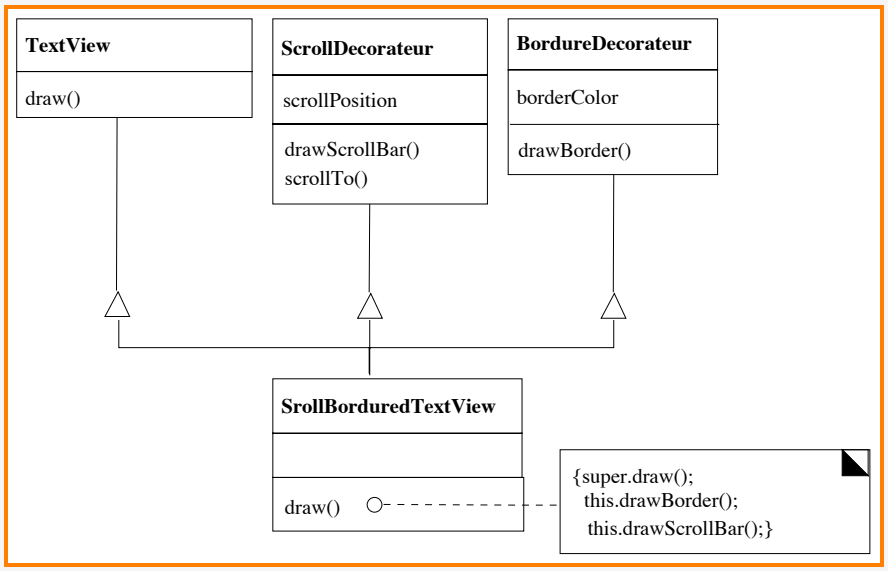


Figure ***(2)*** *– Décoration d'une* `textview`*, solution universelle avec héritage multiple. Limitations : a) statique b) autant de sous-classes que de combinaisons potentielles de décorations.*

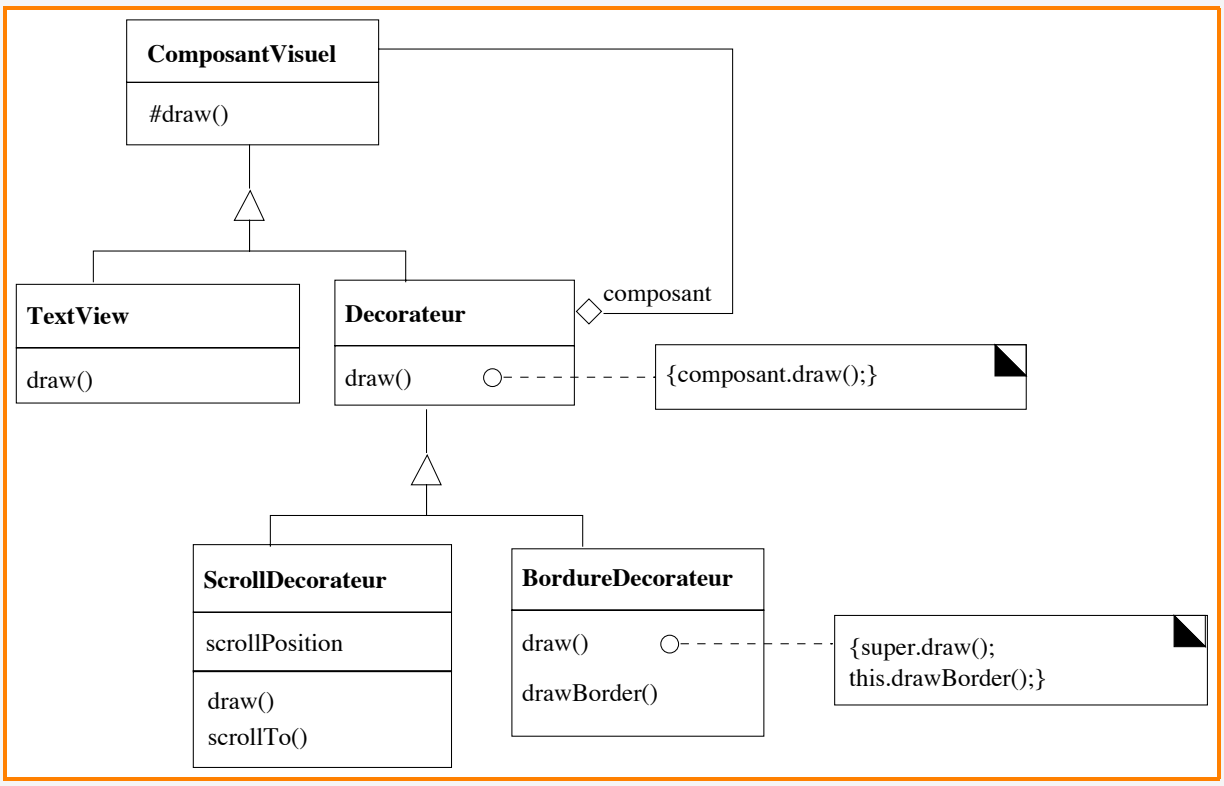


Figure *(3)* – *Décoration d'une* `Textview` *: solution avec le schéma "Decorateur"*

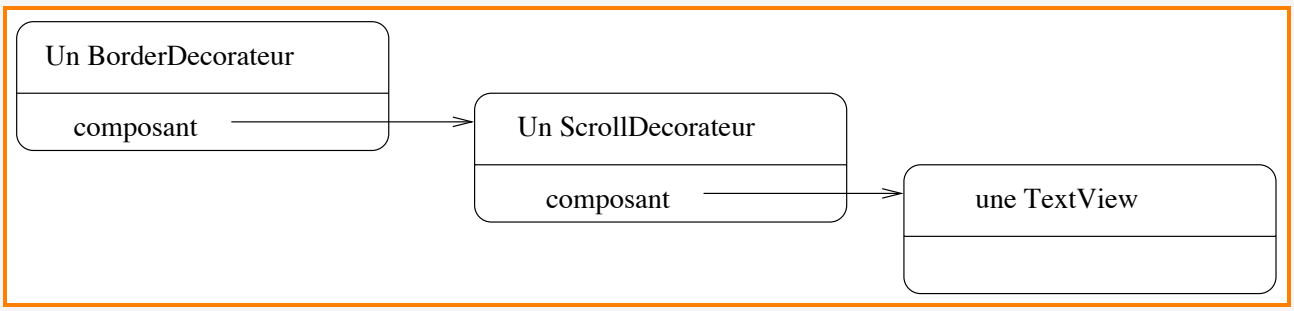


Figure *(4)* – *Objets représentant une* `Textview` *décorée selon le schéma "Decorateur".*

3.3 Principe Général de la solution (figure 5)

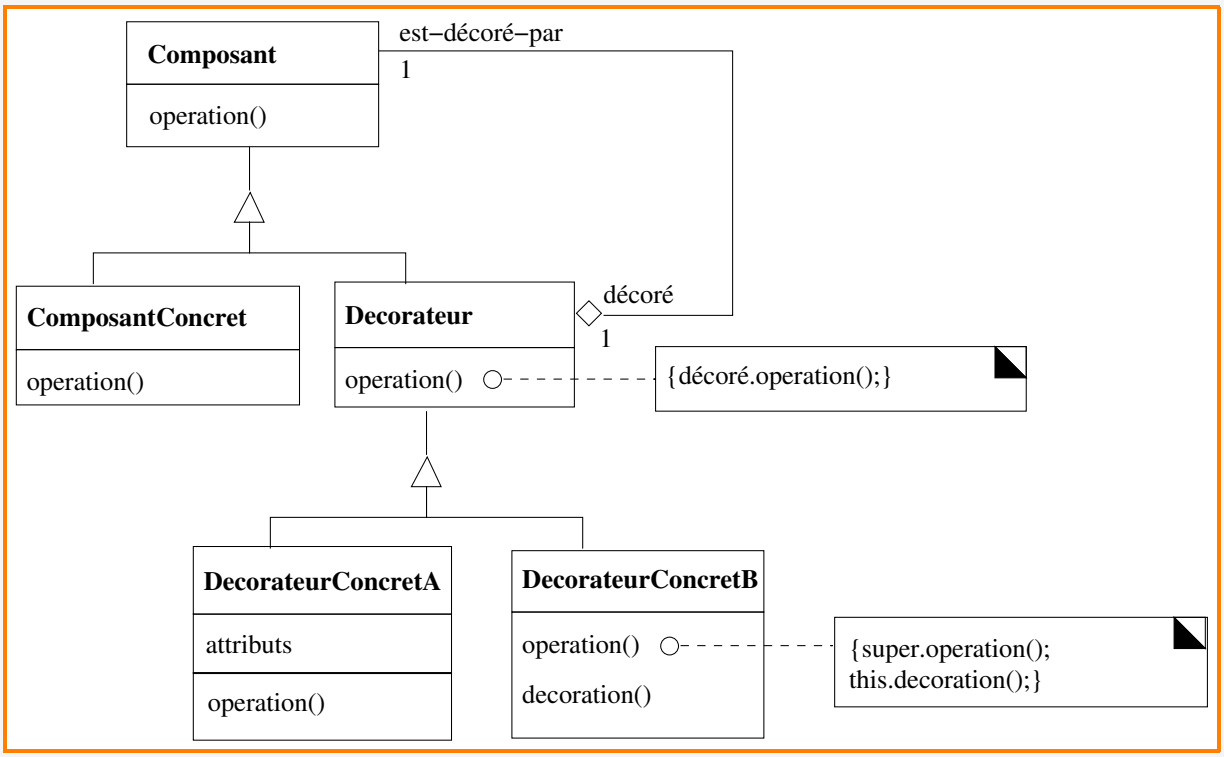


Figure *(5)* – Composant *: objet métier quelconque à décorer, exemple :* `textView`. **Décorateur** *: objet décorant (donc ajoutant des fonctionnalités) à un objet métier, exemple :* `scrollDecorator`*).*

3.4 Une mise en oeuvre concrète

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

```
1  public class Composant {
2      abstract public void draw();
3  }

5  public class VueTexte extends Composant{
6      public void draw() { ... } //affiche un texte
7  }
```

```
1  public class Decorateur extends Composant {
2      Composant décoré;

4      public Decorateur(Composant c){
5          //injection de dépendance
6          //affectation polymorphique
7          décoré = c;}

9      public void draw(){
10         //redirection de message
11         décoré.draw();} //''dispatch 2, sélection sur le composant décoré''
12 }
```

```java
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
        ... }
}
```

```java
class Application {
  public static void main(String[] args){
    Composant c = new VueTexte(''Hello'');
    //ajoûter une décoration
    c = new BorderDecorator(c, 2.5);
    c.draw(); //dispatch 1, sélection sur la décoration
```

Application - Les *streams* en Java (http://stackoverflow.com/questions/6366385/decorator-pattern-for-io)

> *InputStream is an abstract class. Most concrete implementations like BufferedInputStream, GzipInputStream, ObjectInputStream, etc. have a constructor that takes an instance of the same abstract class. That's the recognition key of the decorator pattern (this also applies to constructors taking an instance of the same interface). ... Let's say that we have a bunch of serialized Java objects in a Gzipped file and that we want to read them quickly.*

```java
//First open an inputstream of it:
FileInputStream fis = new FileInputStream("/objects.gz");

//We want speed, so let's buffer it in memory:
BufferedInputStream bis = new BufferedInputStream(fis);

//The file is gzipped, so we need to ungzip it:
GzipInputStream gis = new GzipInputStream(bis);

//We need to unserialize those Java objects:
ObjectInputStream ois = new ObjectInputStream(gis);

//Now we can finally use it:
SomeObject someObject = (SomeObject) ois.readObject();
```

3.5 Discussions

1. Peut-on réaliser un décorateur en *Javascript* avec le lien `__proto__` : oui.
   L'héritage entre objets, via le lien `__proto__` ouvre même de nouvelles possibilités de décoration ; mais quelles sont les propriétés de cet héritage ?

```
var personne1 = {
   prenom:"Jean",
   nom:"Dupont",
   getPrenom:function(){return this.prenom},
   getNom:function(){return this.nom},
   setNom:function(n){this.nom=n},
};

var personne2 = {
   __proto__: personne1, //héritage entre objets
   prenom: "Paul",
 };

personne1.getPrenom(); // Jean
personne2.getPrenom(); // Paul ... héritage de getPrenom() et liaison dynamique
personne2.setNom("Martin");
personne2.getNom(); // Martin
personne1.getNom(); // ????
```

2. 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.
3. 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*.
4. Problème : incompatibilité potentielle de différentes décorations.
5. Problème : L'ordre d'ajout des décorations est signifiant.
6. commentaire : un décorateur peut être vu comme un composite avec un seul composant, mais le décorateur a des fonctionnalités propres que n'a pas son composant ...

4 Un exemple de Schéma structurel : Adapteur

Problème

Intégrer dans une application client une instance d'une classe existante dont l'interface ne correspond pas à la façon dont le 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 Application Typique

Intégrer dans une application en cours de réalisation une instance d'une classe définie par ailleurs (cf. fig. 6).

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.

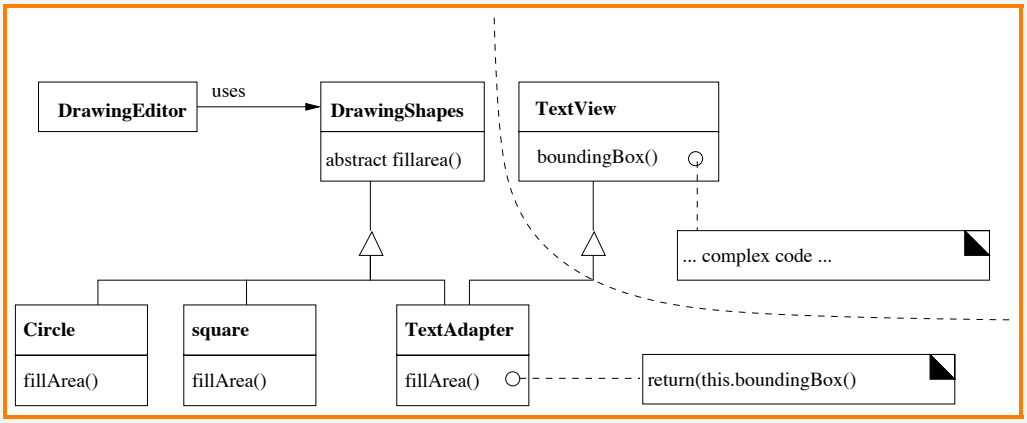


Figure ***(6)*** – *Exemple d'adaptation par spécialisation*

4.2 Principe général de la solution : figures 7 et 8

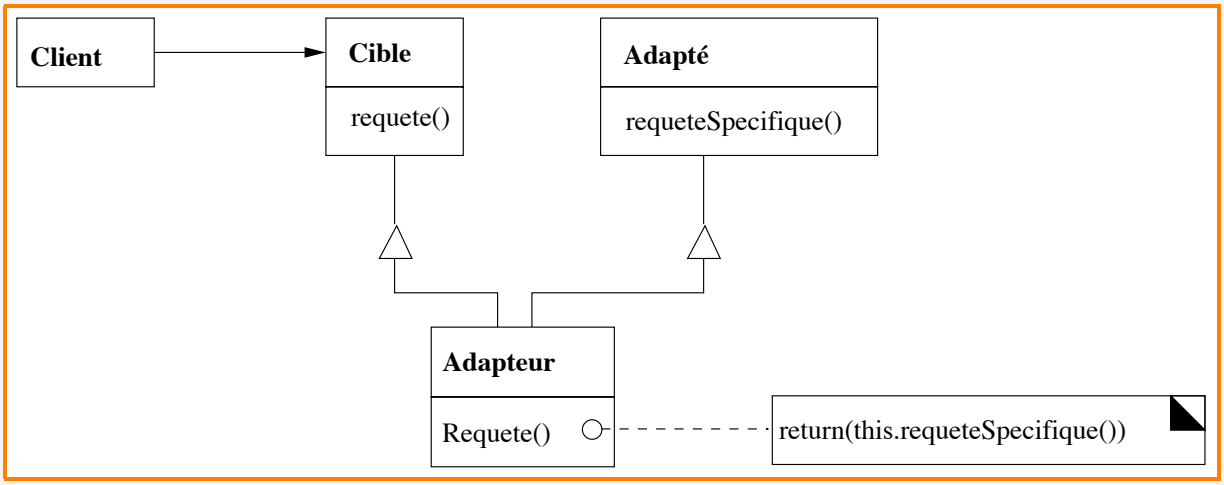


Figure ***(7)*** – *Adapteur réalisé par spécialisation*

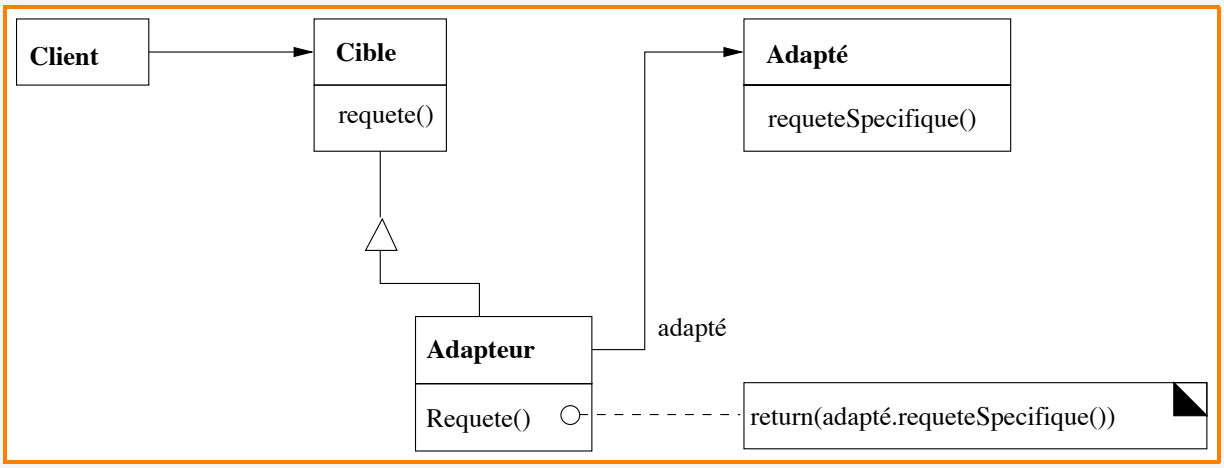


Figure ***(8)*** – *Adaptateur réalisé par composition*

4.3 Discussion

1. Application à la connexion non anticipée de composants. Voir cours “cbeans.pdf”.
   Exemple, intégrer un compteur (`unCompteur` instance de `Compteur` définie dans une bibliothèque) dans une application graphique existante.
   Problème : la classe `Compteur` n'implante pas l'interface `MouseListener`.

```
1 jButton.addMouseListener(new java.awt.event.MouseAdapter() {
2     public void mouseClicked(java.awt.event.MouseEvent evt) {
3         unCompteur.incr();
4         }
5     });
```

Listing (6) *– Solution Java : adaptation d'un objet externe via une classe anonyme dotée d'une méthode conforme au client,réalisant la redirection de message*

2. Mise en évidence du problème posé par la **redirection de message** ici liée à l'utilisation de la composition en paliatif de l'héritage multiple :
   - Nécessité de redéfinir sur l'adapteur toutes les méthodes publiques de l'adapté.
   - Redirection de message : perte du receveur initial (cf. fig. 9). initial du message est perdu, ceci rend certains schémas de réutilisation difficiles à appliquer.

4.4 Discussion sur redirection et perte du receveur

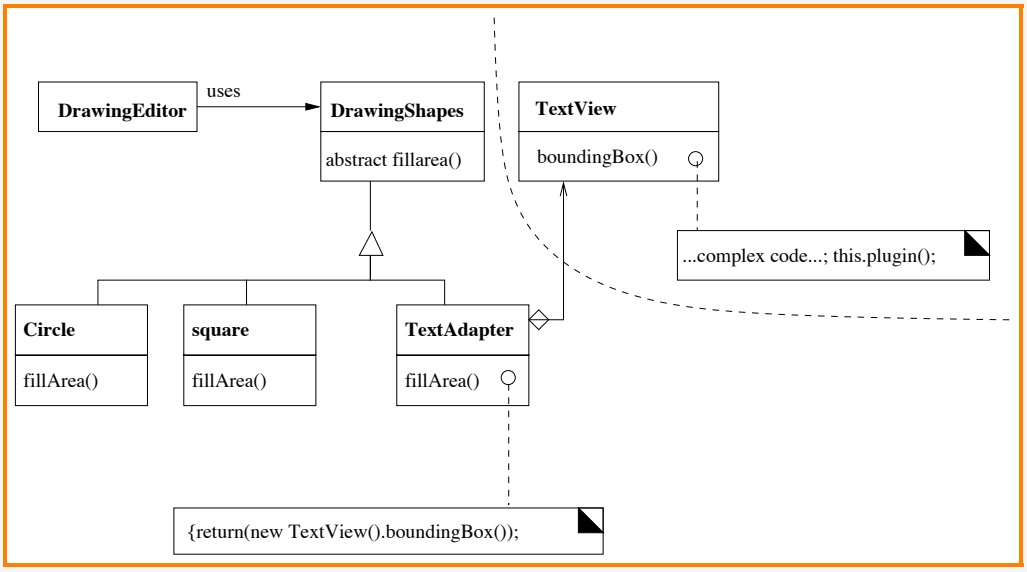
1. 

Figure ***(9)*** *– Adaptation par composition dans l'exemple de l'éditeur de dessin ; receveur initial perdu 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.*

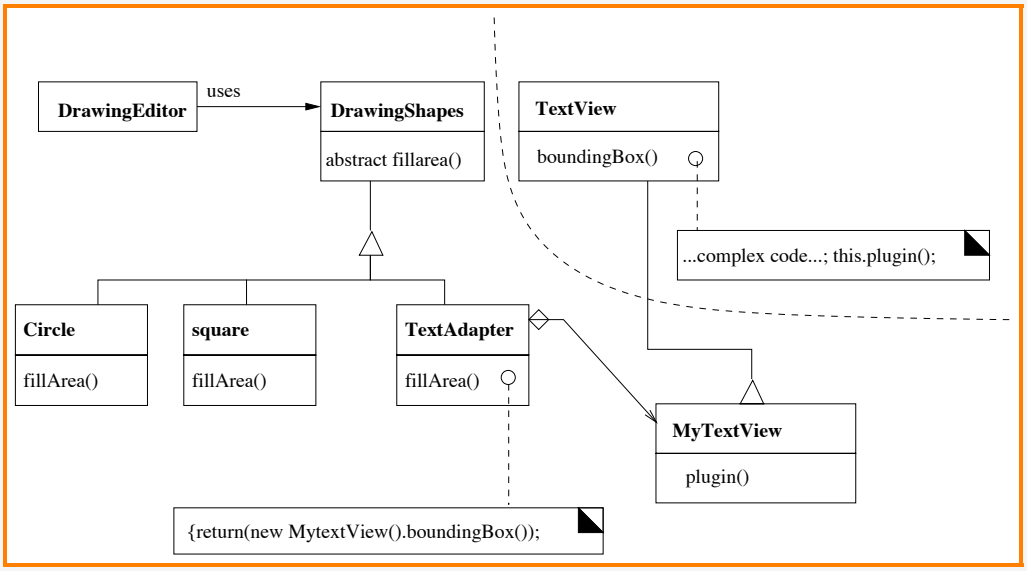
2. 

Figure *(10)* – *Adaptation par composition et solution partielle au problème de perte du receveur initial.*

3. **Généralisation : Adapter, Proxy, State**
   La "redirection de message" et la "perte du receveur initial" associée apparaissent dans de nombreux schémas utilisant la composition ...
   dont ... dont *Adapter*, *Proxy*, *State* .

   Proxy versus **Adapter** :
   Un *Adapter* adapte l'interface (de l'adapté) , en préservant le comportement
   Un *Proxy* modifie (ou simplifie) le comportement (de l'adapté) tout en préservant l'interface. (exemple, le talon côté client en programmation distribuée).
4. Etude (pour ne pas confondre avec le précédent) de la perte du receveur initial en héritage+liaison statique.
   On teste ici l'héritage en liaison statique nommé ("embarquement de type" (`embedded` type du langage `GO`. On obtiendrait le même comportement avec des méthodes non virtuelles en C++.

```go
package main

type Widget struct {
    X, Y int
}
type Label struct {
    Widget // Embarquement de type (matérialisé par un attribut sans nom)
    Text string // Composition explicite
}
func (l Label) KindOf() string{
  return "Label"}

func (l Label) Paint() {
    fmt.Printf("un %q : %q %d %d \n", l.KindOf(), l.Text, l.X, l.Y)}
```

Listing (7) *– Go - Embarquement de type.*
(Adapted from http ://www.drdobbs.com/open-source/go-introduction-how-go-handles-objects/240005949)

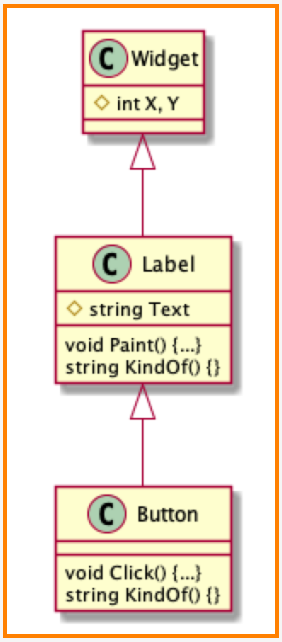


Figure *(11)*

```go
type Button struct {
    Label // Embarquement (héritage matérialisé par un attribut sans nom)
}
func (b Button) Click() {
    fmt.Printf("%p:Button.Click\n", &b) }
func (b Button) KindOf() string {
  return "Button" }

func main() {
     label := Label{Widget{10, 10}, "Joe"}
     label.Paint() //un "Label" : "Joe" 10 10

     button1 := Button{Label{Widget{10, 70}, "Bill"}}
     button1.Click() //ok, fonctionne

     button1.Paint() //un "Label" : "Bill" 10 70
     //... appel de la méthode héritée Paint mais en liaison statique
     //... donc appel de KindOf de Label ... perte du receveur initial

     button1.Label.Paint() //un "Label" : "Bill" 10 70
     //... la preuve, on obtient le même résultat en envoyant le message au label
```

Listing (8) *– Point clé : appeler "button1.Paint" donne le même résultat que "button1.Label.Paint()" ...* **le receveur passé à la méthode Paint est le `Label`, pas le `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. 12) : `Bridge` utilise l'adaptation par composition pour séparer une hiérarchie de concepts de différentes hiérarchies représentant différentes implantations de ces concepts.

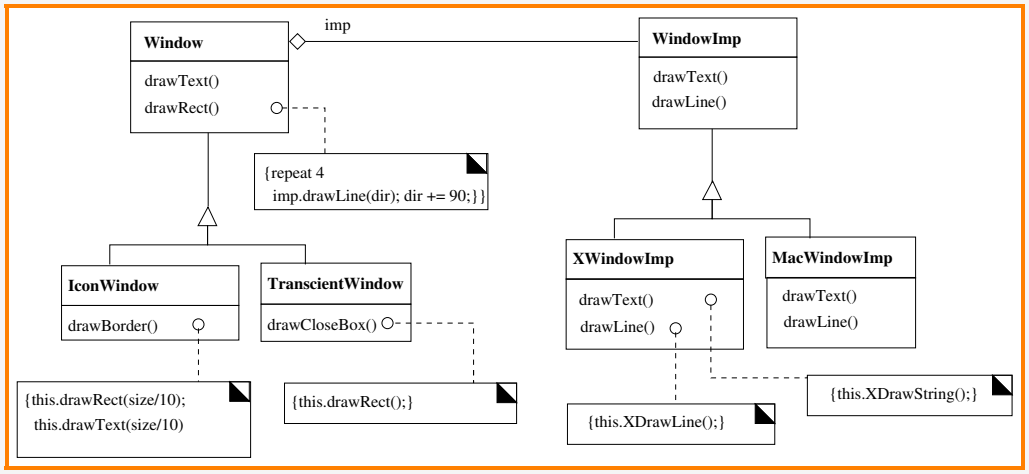


Figure *(12)* *– Exemple d'application du schéma "Bridge"*

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 liée à celle d'interface (à la *Java*), alternative autorisant la définition de méthodes dans la hiérarchies des 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. 13).

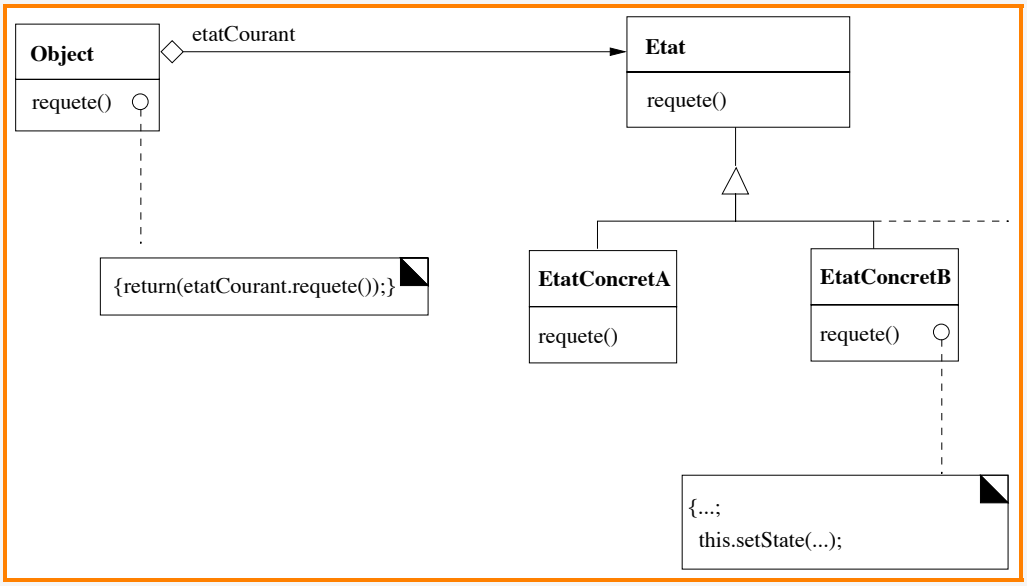


Figure *(13)* – *Pricipe général du schéma "State"*

6.2 Exemple d'application : implantation d'une calculatrice

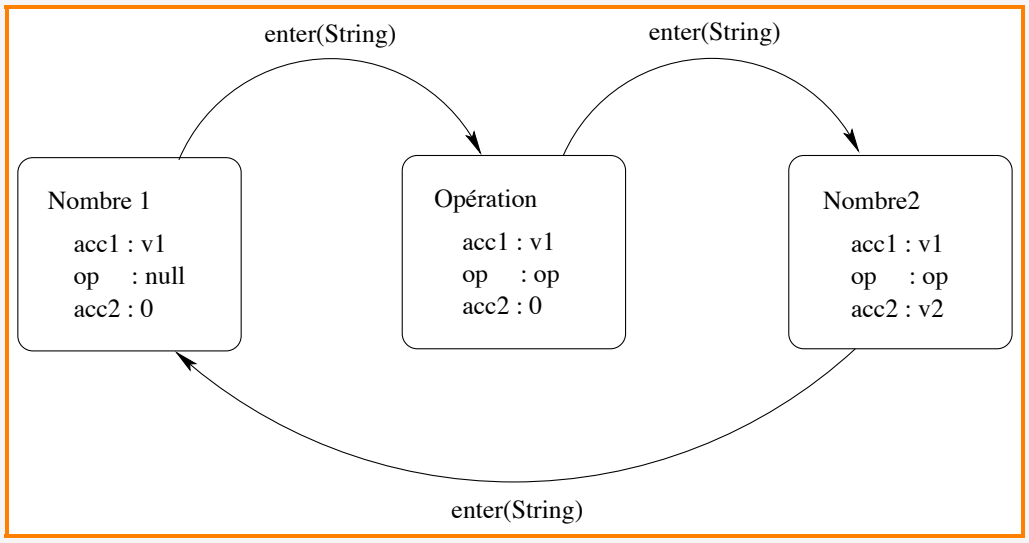


Figure ***(14)*** – *Les différents états et transitions d'une "calculatrice" basique*

Gestion des états

```
1  public class Calculette {
2     protected EtatCalculette etatCourant;
3     protected EtatCalculette[] etats = new EtatCalculette[3];
4     double accumulateur;
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       //qui, dans cette implantation, 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);}
      //l'état suivant est le 2 (entrée opérateur)
      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 {
```

```
    float temp = 0;
    try {temp = Float.parseFloat(s);}
    catch (NumberFormatException e) {
       throw new CalculetteNumberException(s);}

    switch (operations.valueOf(calc.getOp())) {
    case plus: calc.setAccumulateur(calc.getAccumulateur() + temp); break;
    case mult: calc.setAccumulateur(calc.getAccumulateur() * temp); break;
    default:
        throw new CalculetteUnknownOperator(calc.getOp());}
    return (1);}}
```

6.3 Discussion

1. Implémentation : comment représenter l'état courant ?
2. Ce schéma rend explicite dans le code les changements d'état des objets : applications pour la sécurité (contrôle du bon état interne des objets) et la réutilisation (exercice : passer d'une calculette infixée à une postfixée).
3. Evolution des Langages : constructions prenant en compte la catégorisation selon l'état (exemple : la définition par sélection de `Lore`).

```
[Mineur isa select
   from Person   //ensemble réel de référence
   such-that [[oself age] <= 18] //prédicat de sélection
   with (slot droit-vote init-value nil)]
```

4. Evolution des Langages : constructions pour le changement de classe.

   — *Smalltalk* : primitive `Class>>adoptInstance:` *changes the class of an object, and thus its behavior.*

```
Object subclass: #Personne
       instanceVariableNames: 'nom age'

Personne subclass: #Majeur
       instanceVariableNames: ''

Personne subclass: #Mineur
       instanceVariableNames: ''

!Personne methodsFor: 'accessing'!
age
       ^age ! !

vieillir
       age := age + 1.
       (age = 18) ifTrue: [ Majeur adoptInstance: self ]! !
```

Listing (9) *– Application de AdoptInstance*

   — *Smalltalk* : programmer la calculette avec `Class>>adoptInstance:`

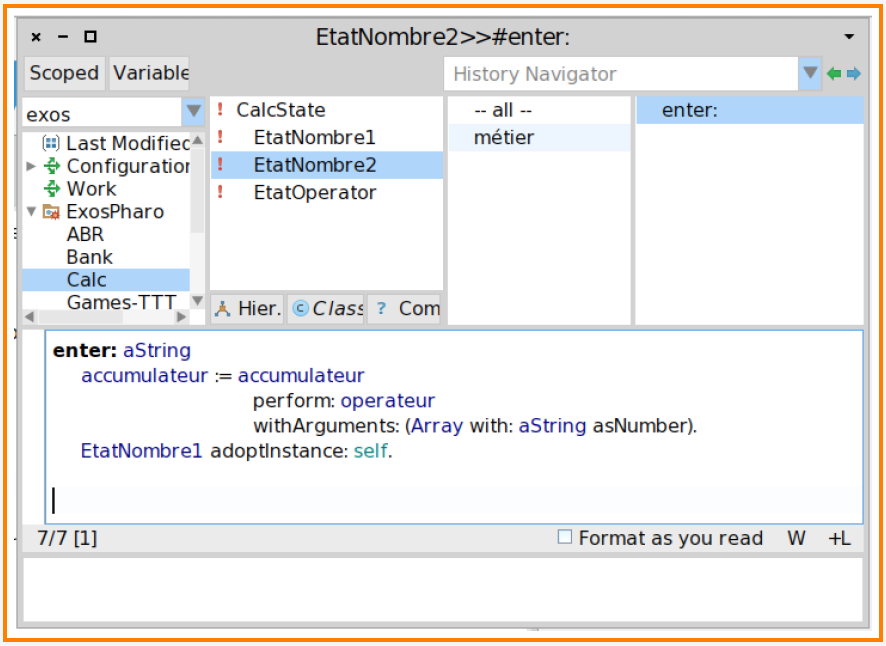


Figure *(15)* – *Une implantation du schéma "State", utilisant la possibilité pour une instance de changer dynamiquement de classe.*

—

— Apparté : application du changement dynamique au (*Dynamic Software Update*) ... le cas de la mise au point de programmes à exécution continue.

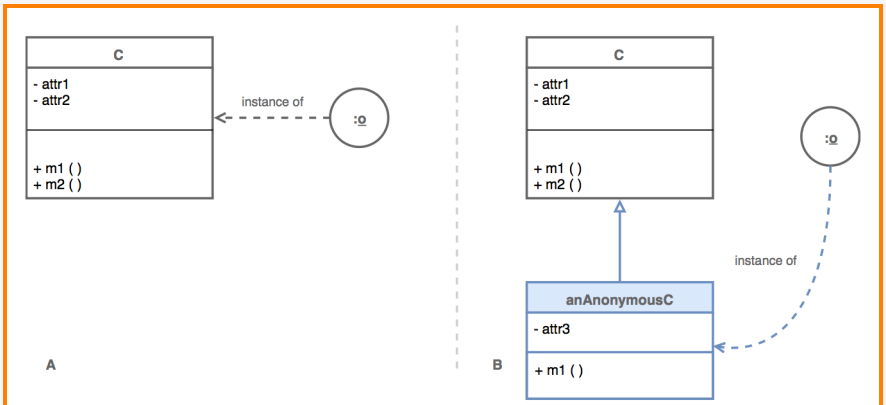


FIGURE 4.2 – Adaptation d'un objet. **A** : l'objet *o* instance de la classe *C* est dans un état non adapté. **B** : l'objet *o* a été adapté. Il est désormais instance d'une classe anonyme, sous-classe de sa classe d'origine *C*.

Figure *(16)* – *Adaptation non-anticipée de comportement : application au déverminage de programmes en cours d'exécution - thèse Steven Costiou (à paraître).*

— *Objective-C* (implantation de MacOS et de IOS) : primitive `object_setClass`

5. Evolution des Langages : utiliser la délégation pour représenter des points de vues.
   Passer d'un état à un autre, c'est aussi changer de point de vue : comment représenter différents points de vues **sur le même objet**, de façon à ce que le même message soit traité différemment selon l'état courant de l'objet.

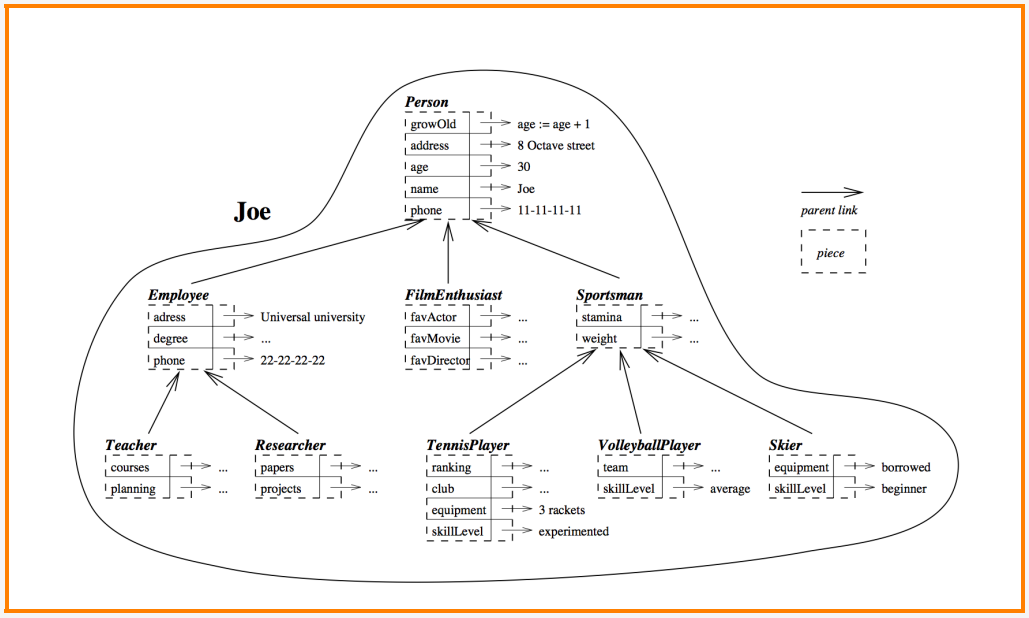


Figure ***(17)*** – *Partage de valeur entre objets en programmation par prototypes, application aux points de vues.* - Daniel Bardou and Christophe Dony. Split Objects : a Disciplined Use of Delegation within Objects. October 1996.

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 18)

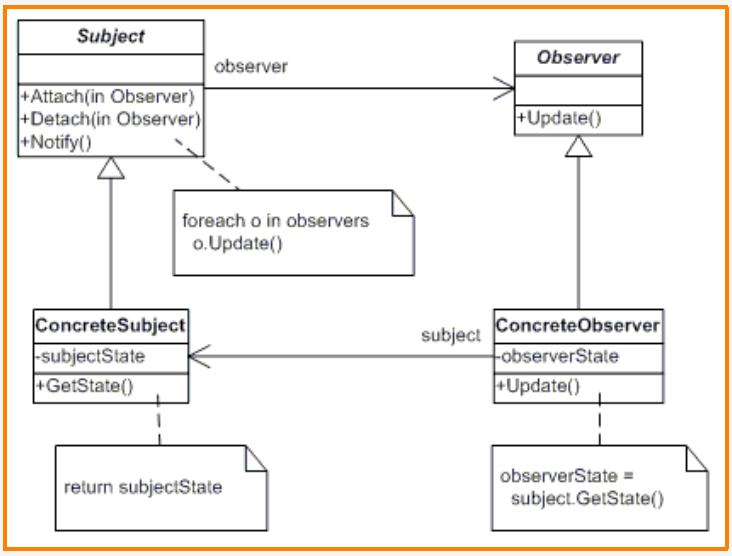


Figure *(18)* – *Le schéma* `Observer` *permet d'établir une collaboration entre un observateur et un observé (ou "sujet" selon ce diagramme).*

7.3 Discussion

- MVC, Event-Based, ...
- Analyser la conception du découplage : `this.notify()`
- Mémorisation des écouteurs et "garbage collector".
- Les limites de la séparation des préoccupations (*separation of concerns*) selon le schema Observateur : comment enlever les `this.notify()` du code métier.
- `Observer` : Classe abstraite versus interface.

7.4 Une mise en oeuvre typique avec l'exemple du framework MVC

Historique :

- Le schéma MVC (article[1] [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 (simultanées) sur un même objet ou changer facilement de vue.
- 1995 : schéma "observateur" généralisation de l'idée de séparation entre observeur et observé.
- 1995 le schéma MVC est un "framework" proposant une architecture logicielle et un ensemble de paramétres (paramétrage par spécialisation) pour la réutilisation.
  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.

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

– Tout modèle (objet métier) est défini par une sous-classe de la classe `Model`. Un modèle peut avoir des observateurs (en premier lieu des vues).

– Tout contrôleur est définie par une sous-classe de `Controller`, chaque contrôleur connait son modèle (et si on le souhaite sa vue), le contrôleur observe l'utilisateur.

– Toute vue est définie par une sous-classe de `View`. Chaque vue connait son contrôleur et son modèle.

7.6 Modèle d'interaction (figure 19)

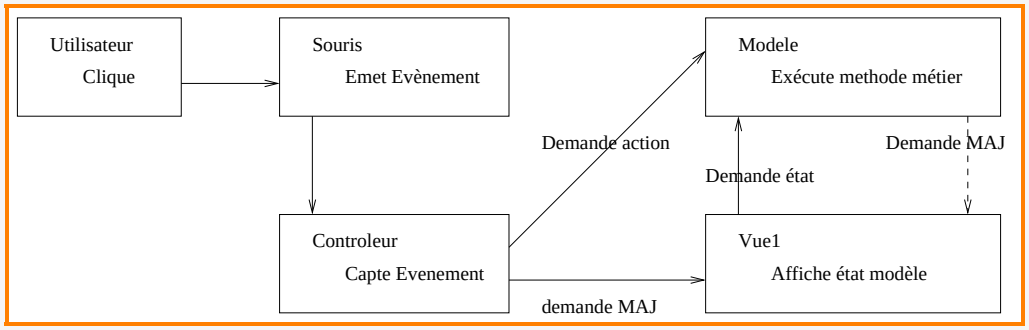


Figure *(19)* – *MVC : Interactions entre modèles, vues et contrôleurs*

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

Parce que le dessin du *pattern* ne dit pas tout, ceci est une tentative de décrire le Framework, ses mécanismes, et une application (exécutable) de la façon la plus succinte possible.

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 d'une 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 d'une 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 d'une application

```
public class CompteurController
                      extends controller
                      implements ActionListerner{

  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();
```

8 Généralisation, Spécialisation

L'utilisation des schémas de conception s'est généralisée, comme en témoigne par exemple la bibliothèque en ligne "Portland Pattern Repository" ().

Les langages de schémas (ensemble de schémas traitant d'une question globale et faisant référence les uns aux autres) se sont spécialisés :

- *reengineering patterns*
- *Architectural pattern in computer science*
- *Interaction patterns*
- :.

et encore plus spécifiquement :

- *exception handling patterns*
- *user interface patterns*,
- :.