

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 transmission des idées

Notes de cours
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

Les schémas offrent des solutions

- adaptées à un ensemble de problèmes,
- qui utilisent le paramétrage par spécialisation et par composition présentés dans la première partie de ce cours (voir

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

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

1.3 Extension : l'ingénierie 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.
<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.

1.4 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 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 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, donc empêche un "new" externe.*/
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 :

1. Visibilité du constructeur,
2. Empêcher la copie (déclarer sans le définir le constructeur par copie)

```

1 Singleton (const Singleton&) ;

```

3. redéfinition de l'opérateur d'affectation

```

1     class S
2     {
3     public:
4         static S& getInstance()
5         {
6             static S instance; // Guaranteed to be destroyed.
7                                 // Instantiated on first use.
8             return instance;
9         }
10    private:
11        S() {} // Constructor? (the {} brackets) are needed here.

13        // C++ 03
14        // =====
15        S(S const&); // Don't Implement
16        void operator=(S const&); // Don't implement

18        // C++ 11
19        // =====
20        // We can use the better technique of deleting the methods we don't want.
21    public:
22        S(S const&) = delete;
23        void operator=(S const&) = delete;
24    };

```

Listing (1) – Singleton C++, Stack Overflow version

4. Plus simple (conceptuellement) : redéfinir new (Smalltalk, Python, Ruby ...)

```

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

```

Listing (2) – Singleton, via une redéfinition de la méthode new, quand elle est une méthode

5. Plus loin (conceptuellement) : une nouvelle méta-classe explicite Singleton-Class (langage CLOS). Singleton-Class est une instance et une sous-classe de Standard-class, et déclare un nouvel attribut : UNIQUE-INSTANCE.

```

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

```

Listing (3) – Singleton-class est la classe des classes qui ne peuvent avoir qu’une seule instance. Elle possède un attribut instance

```

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 (4) – Spécialisation de l’instantiation sur singleton-class, (call-next-method’) réalise l’appel de la méthode masquée par la redéfinition.

- Exemple d’utilisation du schéma Singleton pour les classes True et False, 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 et individuellement des fonctionnalités à un objet, sans modifier 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”

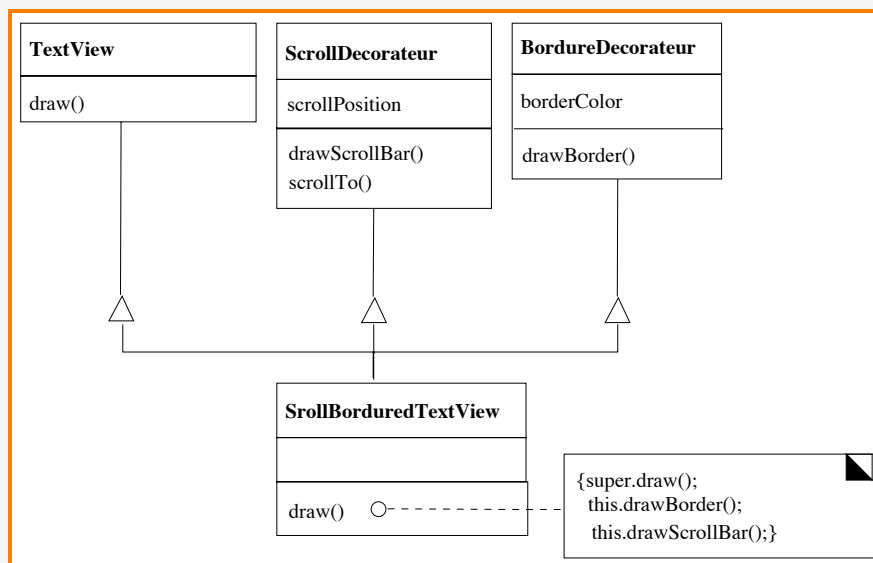


Figure (2) – Décoration d’une textView, solution universelle avec héritage multiple. Limitations : a) statique b) autant de sous-classes que de combinaisons potentielles de décorations.

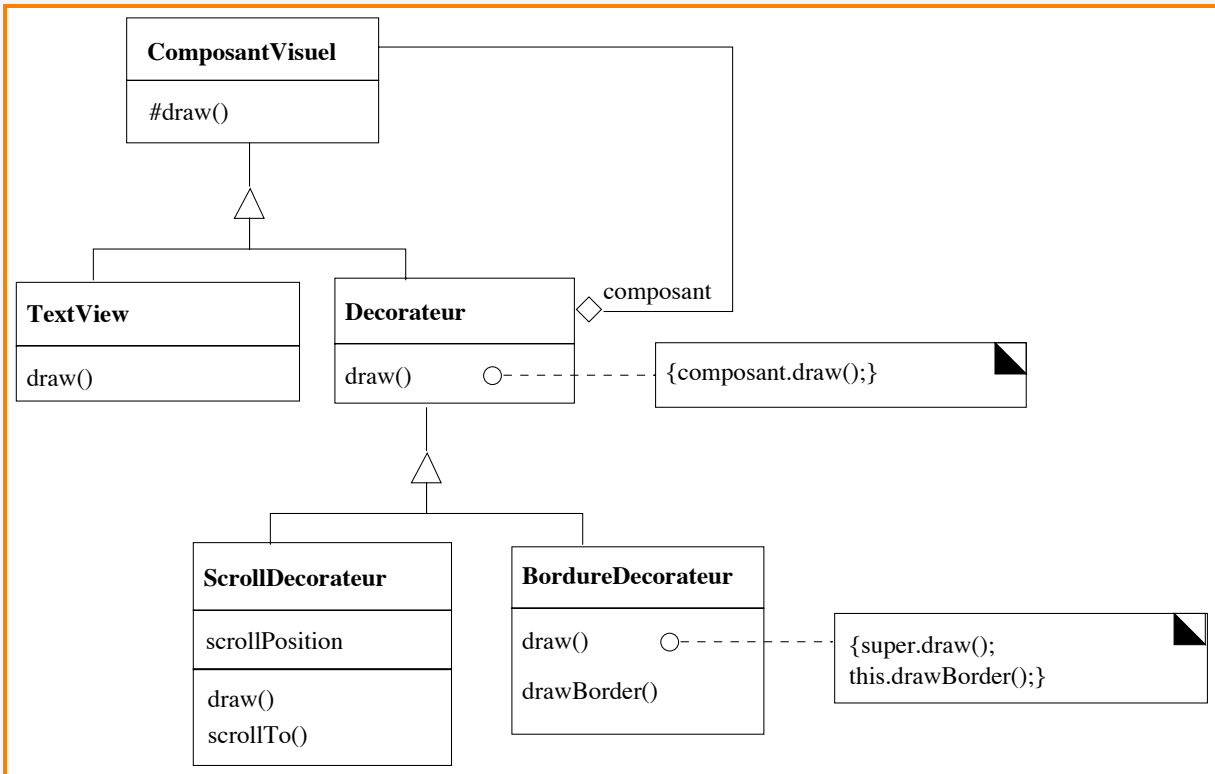


Figure (3) – Décoration d’une Textview : solution avec le schéma “Decorateur”

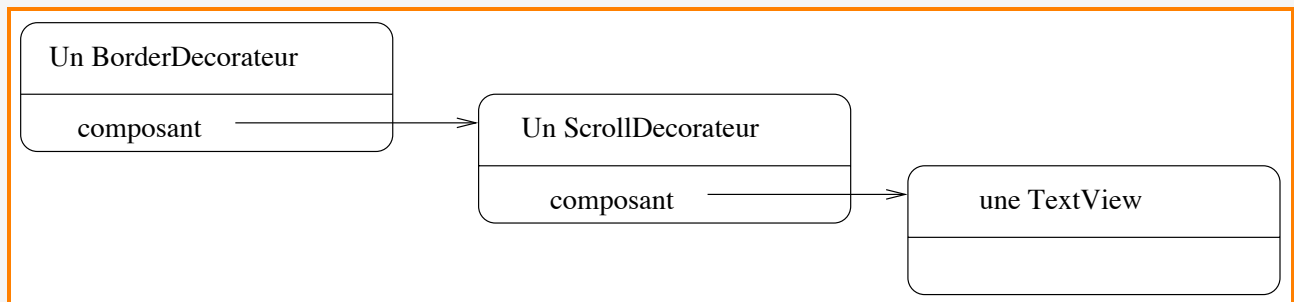


Figure (4) – Objets représentant une Textview décorée selon le schéma “Decorateur”.

3.3 Principe Général de la solution (figure 5)

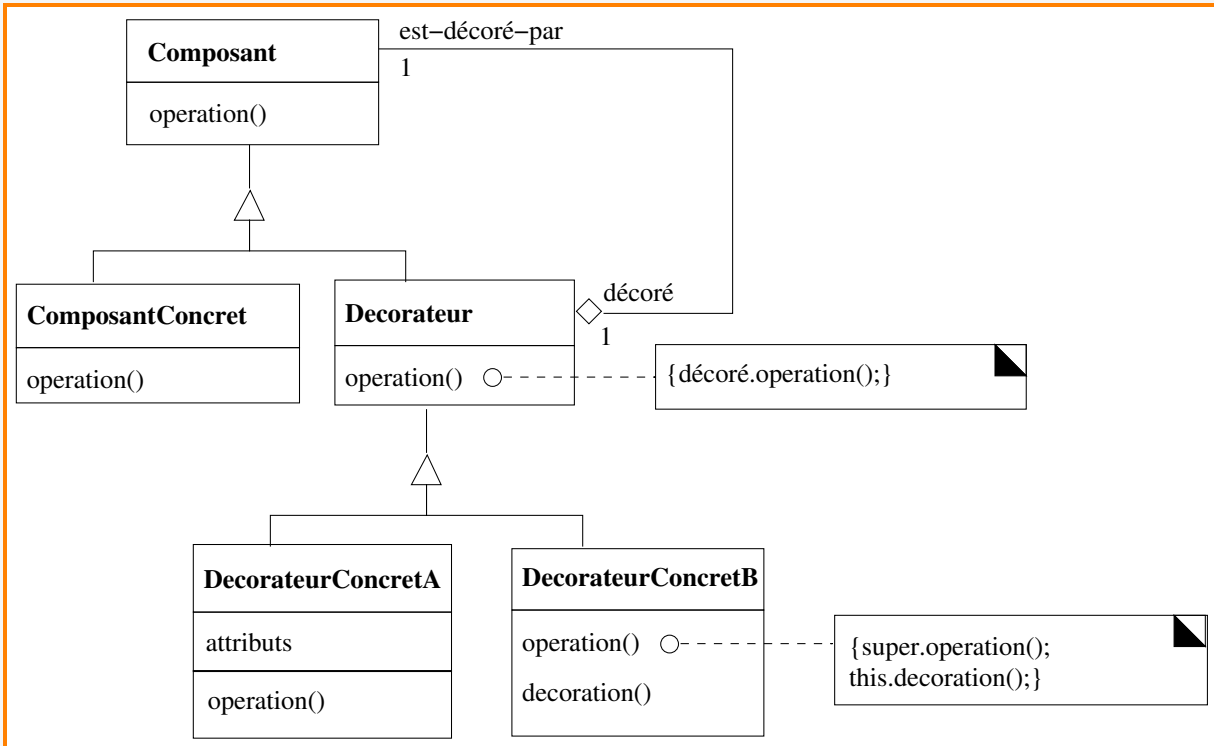


Figure (5) – **Composant** : objet métier quelconque à décorer, exemple : `textView`. **Décorateur** : objet décorant (donc ajoutant des fonctionnalités) à un objet métier, exemple : `scrollDecorator`.

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 }
```

```

1 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 }

```

```

1 class Application {
2     public static void main(String[] args){
3         Composant c = new VueTexte('Hello');
4         //ajouter une décoration
5         c = new BorderDecorator(c, 2.5);
6         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.

```

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

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

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

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

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

```

3.5 Discussion

- La possibilité d'ajouter une fonctionnalité à un objet individuel est une occasion de comparer la programmation par classe à la programmation par prototypes, l'héritage versus la délégation.
- 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.

- 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 significatif.
- Décorateur vu comme un framework : permet de composer tout décorateur avec tout composant.

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

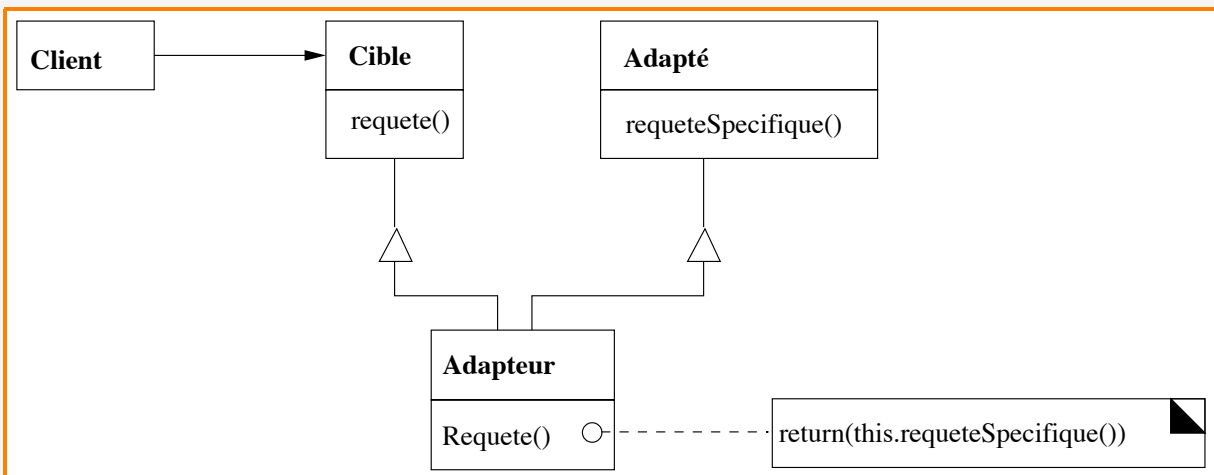


Figure (6) – Adapteur réalisé par spécialisation

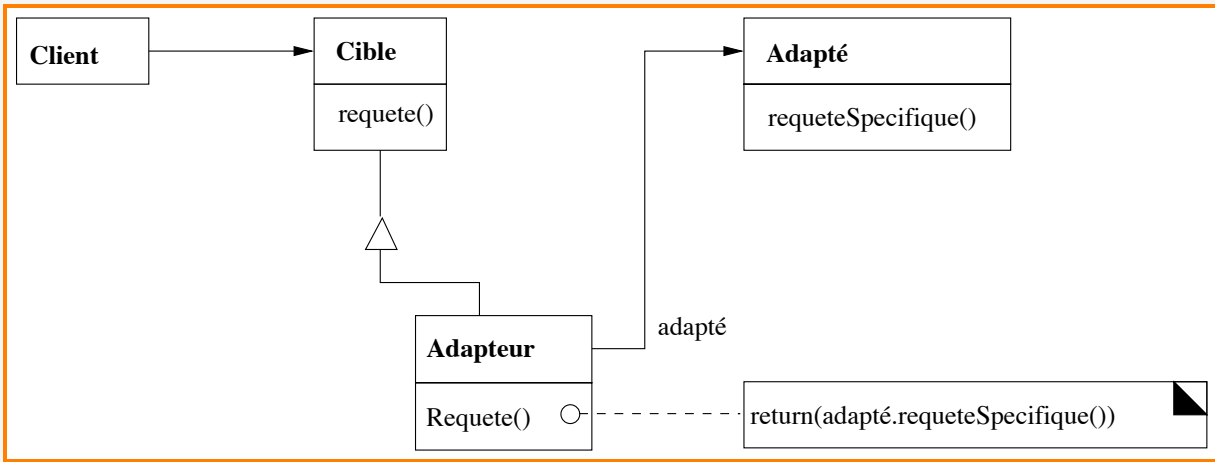


Figure (7) – *Adaptateur réalisé par composition*

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.

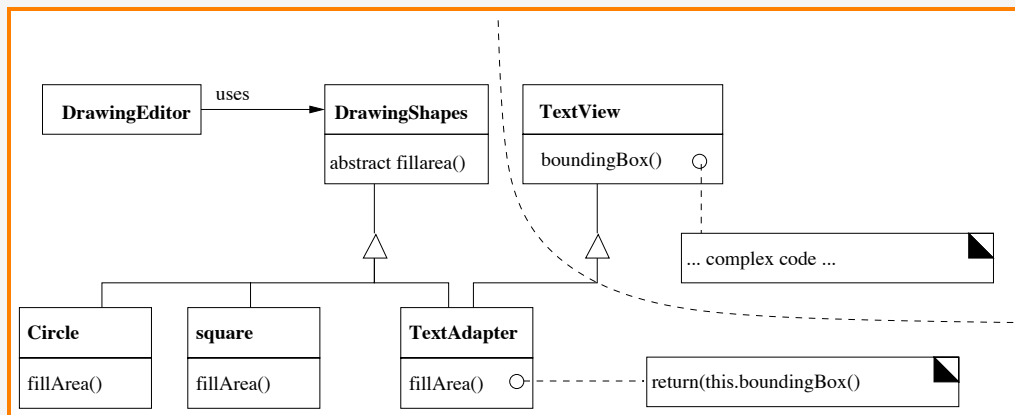


Figure (8) – *Exemple d’adaptation par spécialisation*

4.3 Discussion

- Application à la connexion non anticipée de composants. Voir cours “cbeans.pdf”.
- Mise en évidence du problème posé par la **redirection de message** ici liée à l’utilisation de la composition en palliatif de l’héritage multiple :
 - Nécessité de redéfinir toutes les méthodes publiques de l’adapté.
 - **Perte du receveur initial** (cf. fig. 9). Lors de la redirection vers l’adapté, le receveur initial du message est perdu, ceci rend certains schémas de réutilisation difficiles à appliquer.

4.4 Discussion sur la perte du receveur

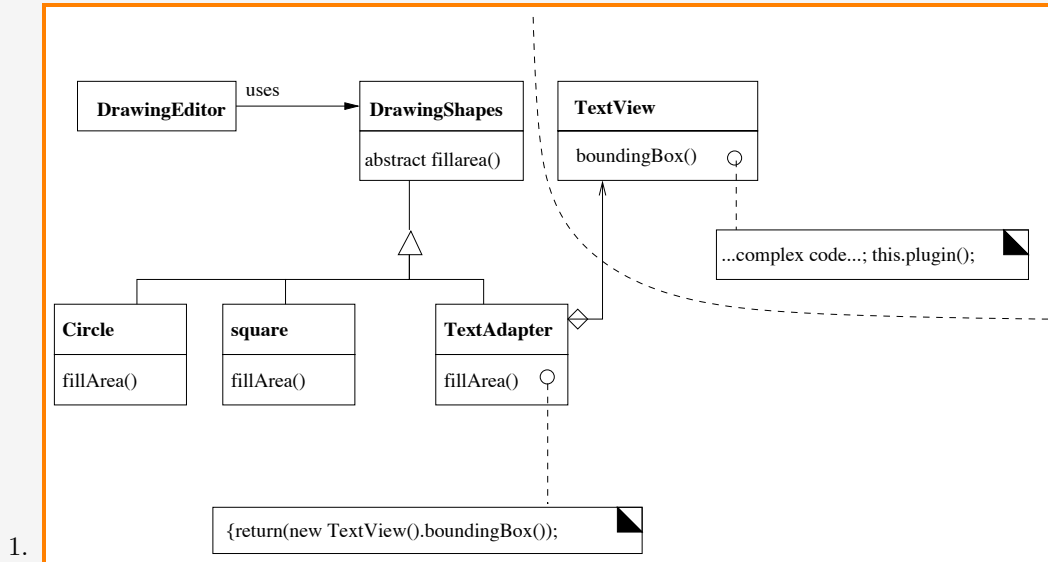


Figure (9) – Adaptation par composition dans l'exemple de l'éditeur de dessin. Perte du receveur initial dans la méthode `boundingBox()`. Ceci rend en l'état impossible la prise en compte d'une spécialisation dans l'application de la méthode `plugin` de la classe adaptée **TextView**. La figure 10 propose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

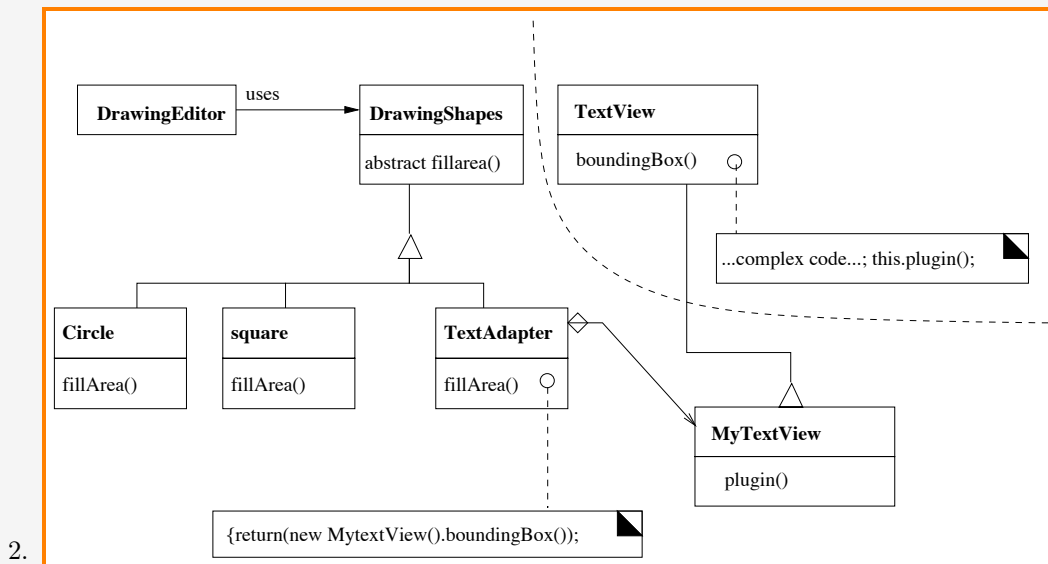


Figure (10) – Adaptation par composition dans l'exemple de l'éditeur de dessin selon un modèle offrant une solution partielle au problème de perte du receveur initial.

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 *Adapteur* 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. Comparer, pour ne pas confondre, la perte du receveur initial précédente, à l'héritage en 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++`.

```
1 package main
2 import "fmt"

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

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

```

1 type Button struct {
2     Label // Embedding (matérialisé par un attribut sans nom)
3 }
4 func (b Button) Click() {
5     fmt.Printf("%p:Button.Click\n", &b)
6 }
7 func (b Button) KindOf() string {
8     return "Button"
9 }
10 func main() {
11     label := Label{Widget{10, 10}, "Joe"}
12     label.Paint() //un "Label" : "Joe" 10 10
13     fmt.Printf("SUITE\n")
14     button1 := Button{Label{Widget{10, 70}, "Bill"}}
15     button1.Click() //0x2081c2020:Button.Click
16     //... appel de la méthode héritée Paint mais en liaison statique
17     //... donc appel de KindOf de Label ... perte du receveur initial
18     button1.Paint() //un "Label" : "Bill" 10 70
19     //... la preuve, on obtient le même résultat en envoyant le message au label
20     button1.Label.Paint() //un "Label" : "Bill" 10 70
21     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.

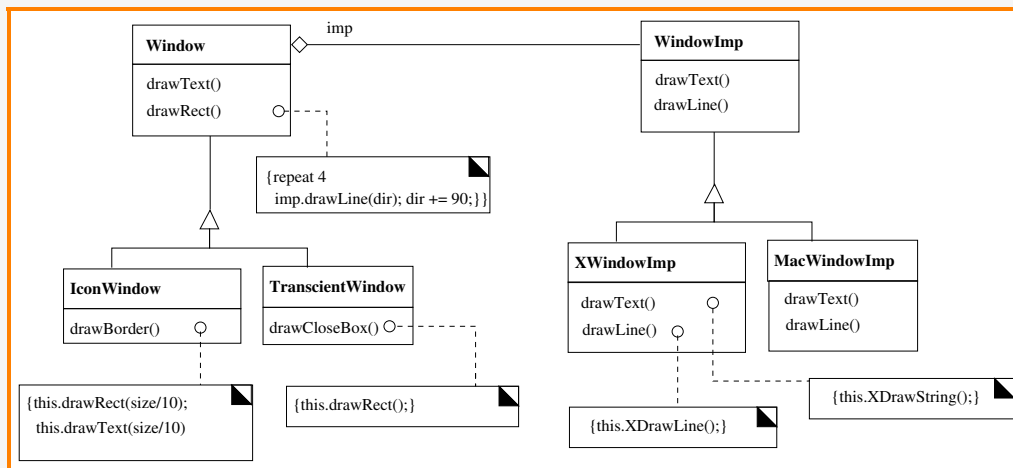


Figure (11) – Exemple d’application du schéma “Bridge”

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 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 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. 12).

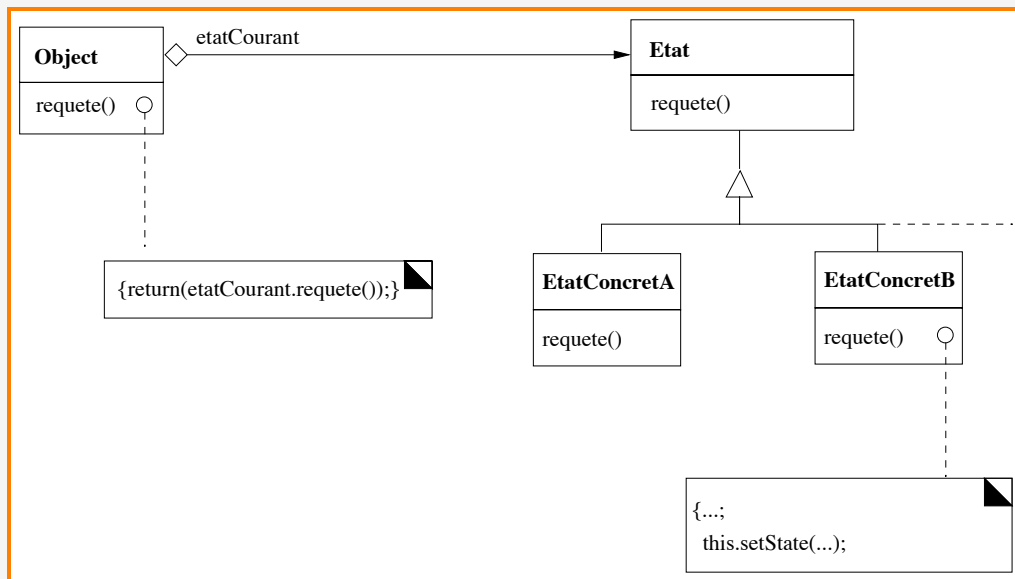


Figure (12) – Principe général du schéma “State”

6.2 Exemple d'application : implantation d'une calculatrice

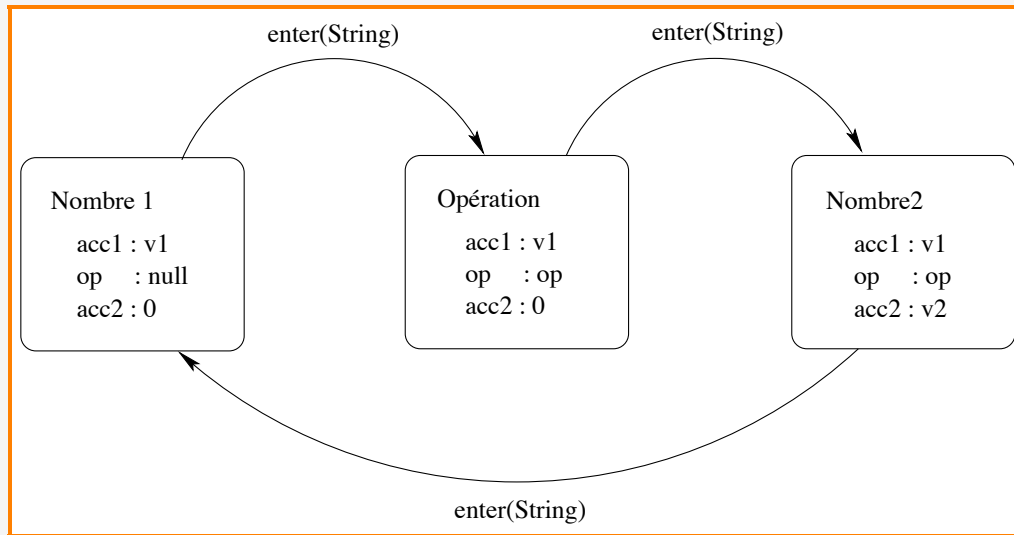


Figure (13) – Les différents états et transitions d’une “calculatrice” basique

Gestion des états

```
1 public class Calculette {
2     protected EtatCalculette etatCourant;
3     protected EtatCalculette[] etats = new EtatCalculette[3];
4     double accumulateur;
5     String operateur;
6
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     ....
3
4     public void enter(String s) throws CalculetteException{
5         //toute requête 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

```
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'operation a 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;
5
6     EtatCalculette(Calculette c){ calc = c; }
7 }
```

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

```
1 public class ENombre1 extends EtatCalculette{
2
3     ENombre1(Calculette c) { super(c); }
4
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         //l'état suivant est le 2 (entrée opérateur)
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 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 ENombre2 extends EtatCalculette {
2     ENombre2(Calculette c){super(c);}
3
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);}}

```

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).

```

1 [Mineur isa select
2   from Person //ensemble réel de référence
3   such-that [[oself age] <= 18] //prédicat de sélection
4   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.*

```

1 Object subclass: #Personne
2   instanceVariableNames: 'nom age'

4 Personne subclass: #Majeur
5   instanceVariableNames: ''

7 Personne subclass: #Mineur
8   instanceVariableNames: ''

10 !Personne methodsFor: 'accessing'!
11 age
12     ^age!!

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

```

Listing (5) – Application de AdoptInstance

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

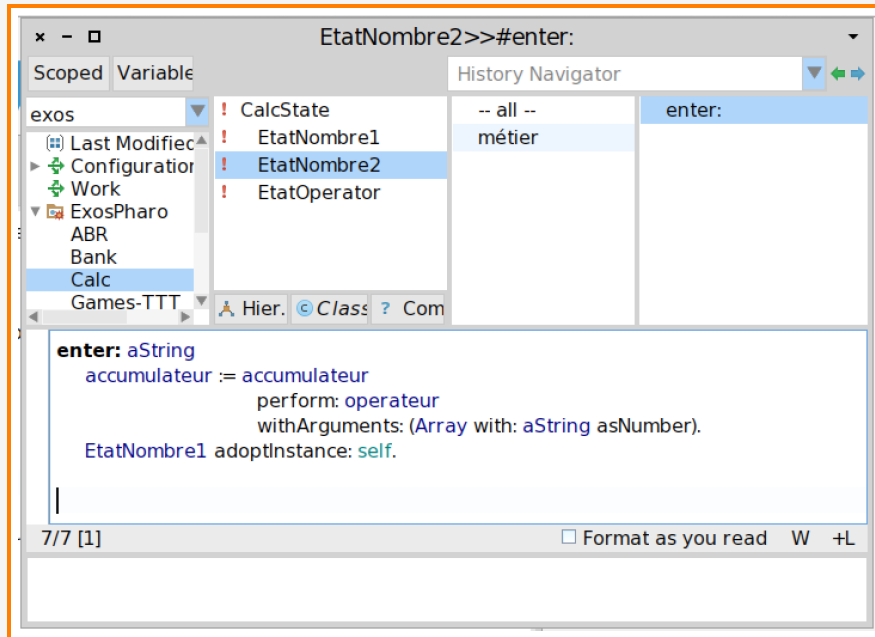


Figure (14) – Une implantation du schéma “State”, utilisant la possibilité pour une instance de changer dynamiquement de classe.

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

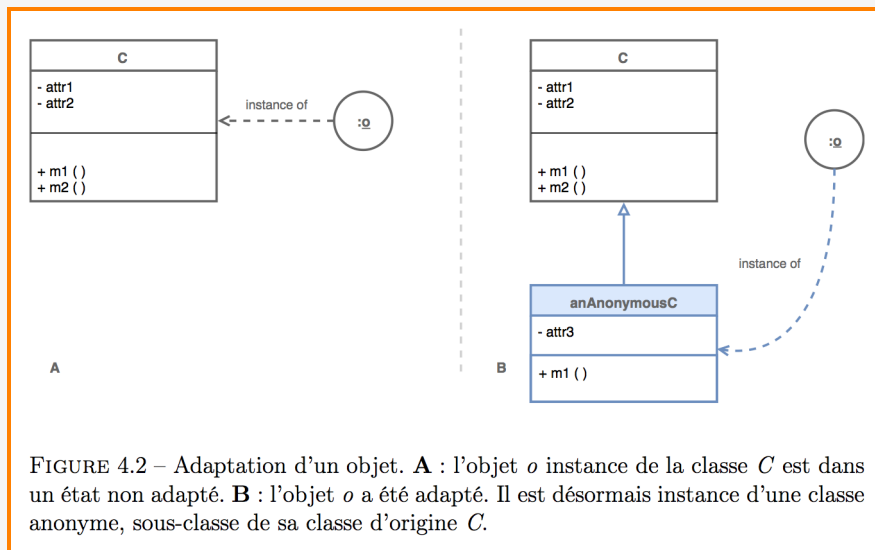


Figure (15) – Adaptation non-anticipée de comportement : application au déverminage de programmes en cours d’exécution - thèse Steven Costiou (à paraître).

- *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.

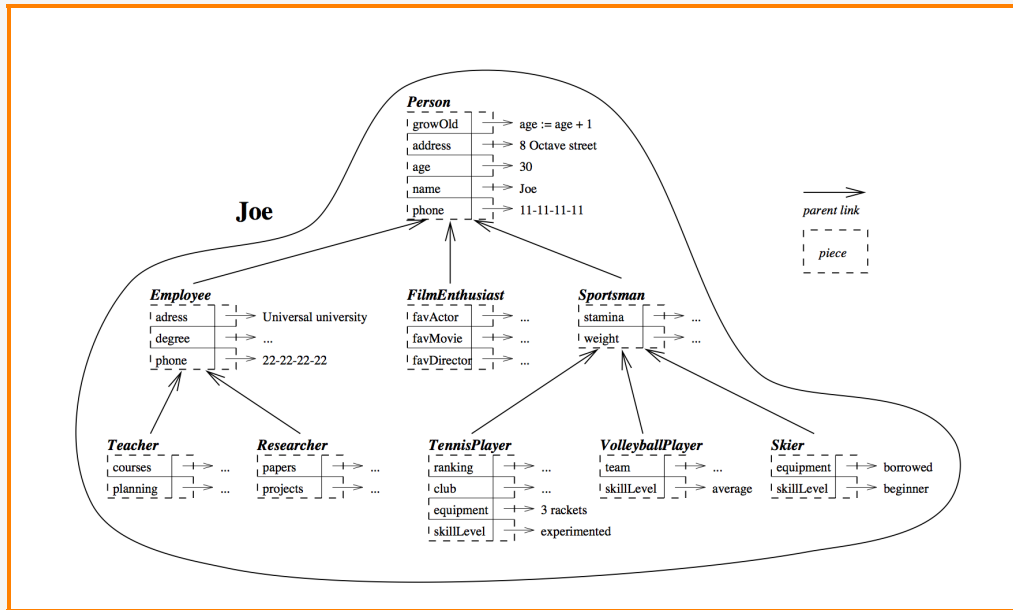


Figure (16) – *Partage de valeur entre objets en programmation par prototypes, application aux points de vues.*
- Daniel Bardou and Christophe Dony. Split Objects : a Disciplined Use of Delegation within Objects. October 1996.

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 17)

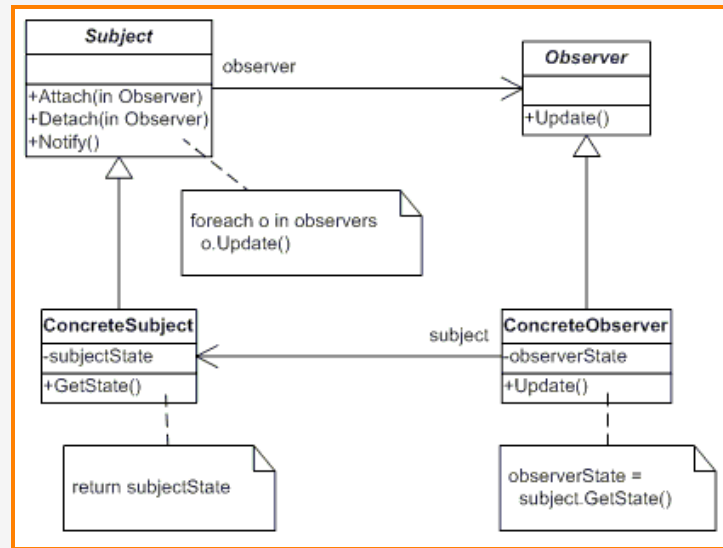


Figure (17) – Le schéma **Observer** permet d’établir une collaboration entre un observateur et un observé (ou “sujet” selon ce diagramme).

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 (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 (simultanées) sur un même objet ou changer facilement de vue.
 - 1995 : schéma “observateur” généralisation de l’idée de séparation entre observateur et observé.
 - 1995 le schéma MVC est un “framework” proposant une architecture logicielle et un ensemble de paramètres (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 défini par une sous-classe de **Controller**, chaque contrôleur connaît 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 connaît son contrôleur et son modèle.

7.6 Modèle d'interaction (figure 18)

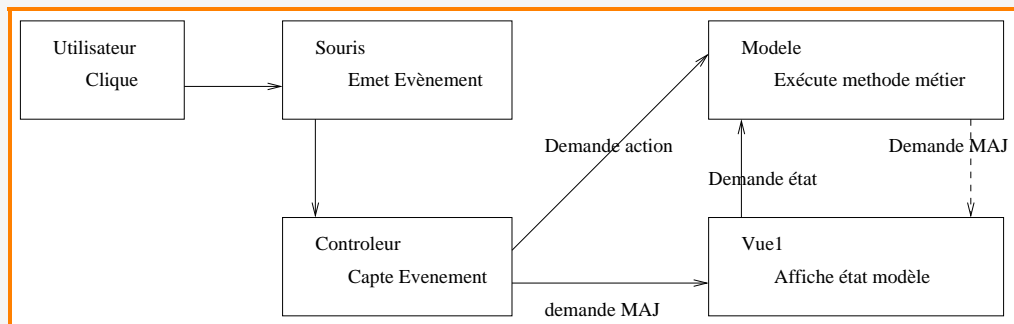


Figure (18) – MVC : Interactions entre modèles, vues et contrôleurs

7.7 Implantation (squelette) inspirée de l'implantation originale (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 succincte 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

```
1 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 d'une application

```
1 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 d'une application

```
1 public class CompteurController
2     extends controller
3     implements ActionListener{
4
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();
```

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*,
- *∴*.