

LABORATOIRE



INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

VERS UNE MEILLEURE RÉUTILISATION DES PATRONS DE CONCEPTION

Laurent Quintian, Philippe Lahire

Projet OCL

Rapport de recherche
I3S/RR-2002-50-FR

Octobre 2002

RÉSUMÉ :

La mise en œuvre des patrons de conception nécessite des techniques d'implémentation. Celles fondées sur le paradigme objet ne sont pas satisfaisantes, car elles sont fréquemment basées sur les mécanismes d'héritage. Ceux-ci rendent les applications dépendantes des patrons et réduisent la facilité de réutilisation de l'application. Dans cet article, nous présentons des techniques d'implémentation plus flexibles grâce aux concepts de la séparation des préoccupations [LOP 95]. Toutes ces techniques d'implémentation sont comparées par rapport à leur facilité de réutilisation pour un patron. À partir des résultats obtenus lors de cette comparaison, nous proposons les bases d'un nouveau modèle.

MOTS CLÉS :

Patron de conception, Programmation orientée objets, Séparation des préoccupations.

ABSTRACT:

In order to use design patterns we need appropriate implementation techniques for them. Techniques that are based on object paradigm are not suitable, since they are frequently based on inheritance mechanisms which make applications very dependent on patterns and reduce reuse facilities. In this paper we introduce new ways to implement more flexible design patterns by applying the concept of separation of concerns [LOP 95]. All these techniques of implementation are compared with respect to their facility of reuse for a pattern. From the results obtained by this comparison, we propose the basis of a new model.

KEY WORDS :

Design Pattern, Object-Oriented Programming, Separation of Concerns.

Vers une meilleure réutilisation des patrons de conception

Laurent Quintian — Philippe Lahire

*Laboratoire I3S (UNSA/CNRS)
Equipe OCL
Les Algorithmes, bâtiment Euclide B
2000, route des lucioles B.P. 121
F-06903 Sophia Antipolis Cedex
Prénom.Nom@unice.fr*

RÉSUMÉ. La mise en œuvre des patrons de conception nécessite des techniques d'implémentation. Celles fondées sur le paradigme objet ne sont pas satisfaisantes, car elles sont fréquemment basées sur les mécanismes d'héritage. Ceux-ci rendent les applications dépendantes des patrons et réduisent la facilité de réutilisation de l'application. Dans cet article, nous présentons des techniques d'implémentation plus flexibles grâce aux concepts de la séparation des préoccupations [LOP 95]. Toutes ces techniques d'implémentation sont comparées par rapport à leur facilité de réutilisation pour un patron. A partir des résultats obtenus lors cette comparaison, nous proposons les bases d'un nouveau modèle.

ABSTRACT. In order to use design patterns we need appropriate implementation techniques for them. Techniques that are based on object paradigm are not suitable, since they are frequently based on inheritance mechanisms which make applications very dependent on patterns and reduce reuse facilities. In this paper we introduce new ways to implement more flexible design patterns by applying the concept of separation of concerns [LOP 95]. All these techniques of implementation are compared with respect to their facility of reuse for a pattern. From the results obtained by this comparison, we propose the basis of a new model.

MOTS-CLÉS : Patron de conception, Programmation orientée objets, Séparation des préoccupations.

KEYWORDS: Design Pattern, Object-Oriented Programming, Separation of Concerns.

1. Introduction

Les patrons de conception sont fréquemment utilisés dans le processus de construction du logiciel car ils apportent des solutions en matière de réutilisation. Cependant, leur implémentation dans le paradigme objet a une fâcheuse tendance à être transversale à la hiérarchie de classes, « polluant » ainsi cette dernière. Cet article propose une solution pour ce problème grâce à d'autres paradigmes de programmation bâtis sur l'objet.

Cet article met en évidence ce problème général aux patrons par l'étude du patron de conception « observateur » [GAM 99] qui est suffisamment représentatif. En effet, pour être utilisable, ce dernier nécessite l'adaptation de ses clients de manière à la fois fonctionnelle (opère sur les classes) et comportementale (opère sur les méthodes).

Chaque patron de conception correspond à un ensemble de classes qui décrit son protocole ; il représente une préoccupation de l'application au sens de la séparation des préoccupations (« Separation of Concerns » [LOP 95]). Typiquement un patron a vocation à être utilisé plusieurs fois dans l'application ; cette préoccupation doit donc dans un premier temps être séparée de ses clients, puis composée avec eux. La description de cette composition, c'est-à-dire les adaptations des clients, doit être réalisée séparément des clients et du patron.

L'étude menée par [NOD 01] sur les avantages de l'implémentation des patrons de conception avec AspectJ [KIC 97] ou Hyper/J [HAR 93] ne répond pas à nos attentes en matière de facilité de réutilisation, car les patrons de conception et leur composition sont encapsulés dans une même entité, ce qui diminue sa facilité de réutilisation. Bien que nous partagions l'opinion de [HAC 02] sur la méthode employée dans l'étude précédente, leur solution n'est toujours pas selon nous satisfaisante, car il n'est pas tenu compte du fait qu'AspectJ ne puisse pas encapsuler tout le protocole de composition d'un patron dans un aspect abstrait.

Cet article compare différentes approches : objet, aspect et sujet, avec comme critère la facilité de réutilisation du patron observateur. Cette étude montre que les trois approches prises individuellement ne sont pas pleinement satisfaisantes. Il est nécessaire d'aller plus loin dans la collaboration entre ces approches ; la proposition que nous faisons après la comparaison va dans ce sens.

L'article est organisé en 5 parties. La section 2 présente l'exemple utilisé comme support de la comparaison. La section 3 compare différentes approches pour améliorer la réutilisation du patron de conception. La section 4 dresse un premier bilan sur les avantages et les défauts des approches étudiées. La section 5 présente notre approche. Enfin, nous concluons en dégagant des perspectives.

2. Un exemple d'utilisation de l'observateur

Le patron de conception observateur « définit une interdépendance de type *1 à plusieurs*, de façon telle que, quand un objet change d'état, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour » [GAM 99]. Sa mise en œuvre est décomposée en deux parties :

- Une partie composée des trois classes (figure 1) qui réifie les éléments du patron. Elle forme son protocole : une entité *observable* permet à d'autres entités, les *observateurs*, de s'abonner pour recevoir des informations sur les modifications de son état. Cette partie est donc indépendante des classes qui vont réutiliser le patron.

- Une partie dépendante de l'utilisation du patron formée de quatre adaptations. Trois sont fonctionnelles¹ : l'état de l'objet observé, les liens d'héritage à modifier, et la méthode *miseAJour()* à implémenter dans les objets observateurs. Une est comportementale² : l'ajout des appels à la méthode *notifieObservateurs()* qui prévient des changements d'état de l'objet observable.

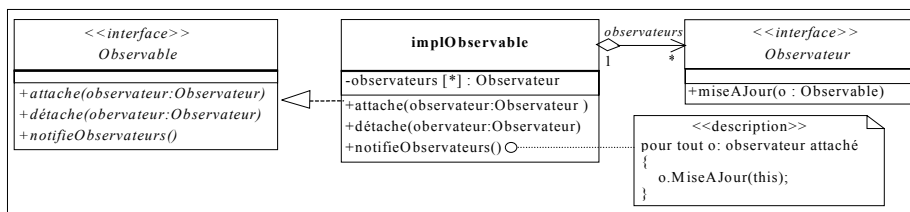


Figure 1. Le patron de conception de l'observateur

Tout au long de l'article nous garderons le même patron et un même exemple dans le cadre d'une interface homme-machine qui consiste à l'utiliser pour encapsuler une interaction entre un bouton (représenté par la classe *Button*) et un label (classe *Label*) qui l'observe. Ainsi quand un bouton est cliqué (appel à la méthode *click()*) le label qui l'observe doit être modifié (appel à la méthode *colorCycle()*).

¹ L'adaptation fonctionnelle modifie les fonctionnalités d'une classe : ses méthodes, ses variables, et son graphe d'héritage.

² L'adaptation comportementale modifie les fonctionnalités existantes d'une classe en modifiant ses méthodes en ajoutant du code avant, après, ou autour d'elle.

3. Le patron de conception observateur face aux approches disponibles

Les sections suivantes étudient l'implémentation du patron de conception observateur dans différentes approches : objets avec Java, aspects avec AspectJ et sujets avec Hyper/J. Java est conservé jusqu'à la fin de cet article y compris pour présenter notre modèle car AspectJ et Hyper/J s'appuient sur ce langage.

3.1. Implémentation en programmation orientée objet

Pour rendre la classe `Button` observable dans un langage à héritage simple en évitant de la modifier, l'unique solution consiste à la sous-classer en une classe `ButtonObservable` qui implémente l'interface `Observable`. La classe `implObservable`, décrite dans la figure 1, doit être dupliquée dans la classe `ButtonObservable` (lignes 6 à 8). Ceci devrait être exprimé (si c'était possible) par un lien de réutilisation d'implémentation.

```

1 class ButtonObservable extends Button implements Observable {
2     public void click() {
3         super.click();
4         notifieObservateurs();
5     }
6     public void attache (Observateur o) { ... } } Duplication de code depuis
7     public void détache (Observateur o) { ... } } implObservable
8     public void notifieObservateur () { ... }
9 }
```

Nous ne détaillerons ni la classe `LabelObservateur` qui s'implémente sur le même modèle, ni la mise en œuvre des deux classes pour l'exemple.

L'implémentation ci-dessus possède plusieurs inconvénients :

(1) *augmentation du nombre de classes et duplication de code* : l'obligation de créer une nouvelle classe `ButtonObservable` et d'y dupliquer une partie importante du patron de l'observateur (lignes 6 à 8),

(2) *modifications sur le reste de l'application* : les clients de la classe `Button` doivent dorénavant créer des objets de type `ButtonObservable`,

(3) *ajout difficile de nouvelles préoccupations* : pour ajouter de nouvelles préoccupations comme un autre patron, soit la classe `Button` doit être à nouveau sous-classée, soit c'est la classe `ButtonObservable` qui doit l'être. Le choix dépend du contexte d'utilisation des préoccupations ce qui oblige le programmeur à se reposer la question à chaque implémentation d'une préoccupation.

Quelle que soit l'approche choisie [OST 01] (métaprogrammation, héritage multiple, etc.), le paradigme de l'objet ne répond pas à nos attentes [NOD 01] [HAC 02] : les classes qui utilisent des patrons de conceptions sont plus difficilement réutilisables, car le code correspondant au(x) patron(s) de conception est dispersé à travers toutes ces classes clientes *polluant* ainsi ces dernières. La séparation des préoccupations permet cependant d'envisager des solutions à ces

problèmes à travers l'extension du paradigme de l'objet, pour prendre en compte la notion de préoccupation dans de nouvelles entités comme les aspects ou les sujets.

3.2. Implémentation avec des aspects

La programmation par aspects [KIC 97] est une extension du paradigme objet avec une double structuration : du code orienté objets et des aspects pour composer les préoccupations qui s'entrelacent avec le reste de l'application de manière modulaire. Un aspect grâce à ses points de jointures¹ définit le tissage pour que la préoccupation qu'il représente soit fusionnée avec le reste du programme.

Pour implémenter l'exemple, deux aspects sont utilisés, comme le préconise [VAN 01]. Un premier abstrait représente le protocole de composition de la préoccupation. Le protocole de composition est le tissage abstrait et générique pour composer la préoccupation. Un deuxième spécialise le premier par héritage [HAN 01] pour adapter la composition de la préoccupation à un contexte d'utilisation. Le patron de conception est implémenté en Java par les classes de la figure 1.

Conformément à cette approche l'aspect `ProtocoleObservateur` (version modifiée du tutorial d'AspectJ [ASP 02]) contient le protocole de composition :

```

1 import patrondeconceptions;
2 abstract aspect ProtocoleObservateur {
3     abstract pointcut changementEtat(Observable s);
4     after(Observable s): changementEtat(s) {
5         s.notifieObservateurs();
6     }
7
8     private Vector Observable.observateurs=new Vector();
9     public void Observable.attache(Observateur obs) {
10        Observateurs.addElement(obs);
11    }
12    public void Observable.detache(Observateur obs) {
13        Observateurs.removeElement(obs);
14    }
15    public void Observable.notifieObservateurs() {
16        for (int i = 0; i < Observateurs.size(); i++)
17            ((Observateur)Observateurs[i]).miseAJour(this);
18    }
19 }

```

} Adaptation
comportementale

} Adaptation
fonctionnelle

La première partie de l'aspect (lignes 3 à 6) définit un point de jointure abstrait (ligne 3 : `changementEtat`) qui représente les changements d'état de l'objet jouant le rôle de l'observé. La méthode `notifieObservateurs` est exécutée après le point de jointure `changementEtat` (lignes 4 à 6) pour mettre à jour les observateurs.

¹ Les points de jointure sont des éléments de la sémantique du langage sur lesquels les aspects peuvent agir par adaptation comportementale ou fonctionnelle (limitée).

L'aspect `ProtocoleObservateur` devrait ensuite déclarer deux autres adaptations fonctionnelles. Une première représentant la classe qui doit devenir observable, cette adaptation est effectuée en deux étapes : (1.a) La classe à rendre observable implémente l'interface `Observable`, (1.b) Celle-ci doit donc inclure l'implémentation des méthodes de la classe `implObservable` afin de l'empêcher de devenir abstraite. Une deuxième (2) qui doit adapter la classe observatrice pour qu'elle implémente l'interface `Observateur`.

Ces deux adaptations ne peuvent pas être déclarées dans l'aspect `ProtocoleObservateur` car AspectJ n'offre pas la possibilité d'utiliser des points de jointure abstraits pour réaliser des adaptations fonctionnelles. Ces adaptations sont réalisées dans l'aspect concret : (1.a) à la ligne 23, (2) à la ligne 27. (1.b) est réalisable dans l'aspect abstrait en concrétisant l'interface `Observable` (lignes 8 à 18).

L'exemple est composé grâce à l'aspect `Application_IHM` (lignes 20 à 30) qui hérite de `ProtocoleObservateur`. Il permet à un `Button` d'être observé (ligne 22) par un `Label` (ligne 26) qui change sa couleur (lignes 27 à 28, introduction de la méthode `miseAJour` dans la classe `Label`) quand le bouton est cliqué (lignes 23 à 24, concrétisation du point de jointure `changementEtat`).

```

20 aspect Application_IHM extends ProtocoleObservateur {
21     declare parents: Button implements Observable;
22     pointcut changementEtat(Observable s):
23         target(s) && call(void Button.click());
24
25     declare parents: Label implements Observateur;
26     public void Label.miseAJour() {
27         colorCycle();
28     }
29 }

```

} Rôle
Observable

} Rôle
Observateur

Les quelques instructions qui suivent mettent en œuvre une association observateur - observable entre un `Button` (observable) et un `Label` (observateur) :

```

30 Button button= new Button (...)
31 Label label = new Label (...)
32 button.attache(label);
33 button.click();

```

La dernière ligne déclenche indirectement l'exécution de `notifieObservateur()` qui entraîne `label.MiseAJour()` et donc `label.ColorCycle()` grâce aux déclarations se trouvant dans l'aspect `Application_IHM`.

Le patron de conception est implémenté de manière modulaire sous forme de classes Java non couplées avec le reste de l'application. AspectJ a permis de le composer de manière non intrusive avec les classes qui doivent le réutiliser.

L'adaptabilité du patron de conception est restreinte par les possibilités d'adaptation fonctionnelle d'AspectJ : un aspect ne peut pas définir un lien de réutilisation d'implémentation qui permettrait d'introduire dans une classe

l'implémentation de méthodes fournies par d'autres classes. Cette limitation a un impact direct sur la facilité de réutilisation.

Malgré la possibilité offerte par AspectJ d'utiliser un couple aspect abstrait / aspect spécialisé, la facilité de réutilisation est amoindrie [DEV 01] par l'impossibilité d'encapsuler tout le protocole de composition dans l'aspect abstrait. L'utilisateur doit donc compenser dans l'aspect spécialisé l'absence de ce service, sans contrôle sur la composition de la part d'AspectJ.

Dans la suite, nous abordons le même exemple en utilisant la programmation orientée sujets.

3.3. Implémentation avec des sujets

La programmation orientée sujets [HAR 93] et son évolution, la programmation orientée HyperSpaces [OSS 00], permettent deux nouvelles formes d'encapsulation : les sujets et les modules de composition. Un sujet (HyperSlice) est un ensemble de classes qui encapsule une préoccupation, éventuellement transversale à la hiérarchie de classes de l'application. Un module de composition (HyperModule) rassemble les préoccupations à composer et les opérateurs de composition pour produire une nouvelle préoccupation composite.

Un sujet possède une propriété de complétude : aucune classe appartenant à un sujet ne peut référencer des entités (des méthodes, des classes, etc.) qui lui sont extérieures. Toutefois, un sujet ne requiert pas une définition complète de ces déclarations : par exemple une fonction peut être spécifiée sans pour autant être implémentée. La complétude des sujets permet d'éliminer le couplage entre les différents sujets. Les morceaux manquants d'un sujet sont obtenus lors de la phase de composition.

Le problème contient deux préoccupations qui sont représentées par deux sujets. Le patron de conception de l'observateur est encapsulé par identification des classes qui le compose dans un premier sujet (HyperSlice) nommé `PatronDeConceptions.Observateur` (lignes 2 à 3). Ce sujet contient les trois classes implémentées en Java de la figure 1 : `Observable`, `Observateur` et `implObservable`. Un deuxième sujet `Application.IHM` (ligne 4) contient les classes `Button` et `Label` qui vont être composées avec les classes du patron.

Hyper/J [HYP 02] ne permet pas d'encapsuler des protocoles de composition, car les modules de composition ne possèdent ni l'héritage, ni l'agrégation. Les deux sujets précédents sont donc directement composées pour produire un nouveau sujet : `Application.IHMetendue`. Cette composition utilise deux opérateurs de composition :

- **equate class** *classe1* *classe2* : est un opérateur de composition fonctionnel qui permet de fusionner *classe1* et *classe2* à l'intérieur d'une nouvelle classe, qui est

produite par le module de composition et qui a le même nom que *classe1*. La fusion de classe est une opération récursive : elle s'applique aussi aux méthodes et aux variables ; ces dernières sont fusionnées si elles ont le même nom (clause **mergebyname**, ligne 12).

– **Bracket méthode1 with after méthode2** : est un opérateur de composition comportemental qui provoque l'exécution de *méthode2* après un appel à *méthode1* sans transporter de contexte (paramètres, valeurs de retour, etc.) de *méthode1* vers *méthode2*.

Tous les opérateurs de composition d'Hyper/J sont *ex-situ* [OST 00] : la composition qu'ils définissent est produite, par le module de composition qui les contient, dans un nouveau sujet. Ce dernier possède le même nom que le module de composition qui l'a produit.

Le module de composition `Application.IHMetendue` (lignes 6 à 19) contient la composition du patron de conception (ligne 8) et des classes qui doivent l'utiliser (ligne 7). Le sujet composite `Application.IHMetendue` (lignes 5 à 20) représente la réutilisation du patron de conception : la classe `Button` devient `Observable` (ligne 12), la classe `Label` devient un `Observateur` (ligne 16), la méthode `click` de la classe `Button` provoque une mise à jour des observateurs (lignes 13 à 14) et l'appel de la méthode `MiseAJour()` sur un `Label` déclenche l'exécution de la méthode `colorCycle()` (lignes 17 à 18).

```

1  -concerns
2    package patrondeconceptions : PatronDeConceptions.Observateur;
3    package application.ihm      : Application.IHM ;
4  -hypermodules
5    hypermodule Application.IHMetendue
6      hyperslices:
7        Application.IHM,
8        PatronDeConceptions.Observateur;
9      relationships:
10     mergebyname;
11
12     equate class Button , Observateur.implObservable;
13     bracket Button.Click with after
14       Observateur.implObservable.notifieObservateurs();
15
16     equate class Label , Observateur.Observateur;
17     bracket Observateur.MiseAJour with after
18       Label.colorCycle();
19   end hypermodule;

```

Hyper/J a permis d'implémenter de manière modulaire le patron de conception sous forme de classes Java non couplées avec le reste de l'application puis de le composer de manière non intrusive avec les classes qui doivent l'utiliser grâce à un module de composition.

Pourtant, La réutilisation du patron de conception de l'observateur est difficile car Hyper/J ne permet pas d'encapsuler un protocole de composition. L'utilisateur du patron de conception doit respecter le protocole de composition de l'observateur sans contrôle de la part d'Hyper/J.

4. Synthèse de l'étude

AspectJ et Hyper/J ont permis d'isoler le protocole du patron de conception de l'observateur dans un ensemble de classes (voir figure 1). Ce dernier est ensuite réutilisé pour être composé de manière non intrusive avec les classes qui l'utilisent.

Dans chacun des langages la composition d'une préoccupation est encapsulée dans une nouvelle entité : les aspects pour la programmation orientée aspects, et les modules de composition pour la programmation orientée sujets.

La composition peut [OST 00] : soit synthétiser de nouvelles classes comme le fait Hyper/J, c'est une composition *ex situ* ; ou soit modifier des classes existantes comme le fait AspectJ, c'est une composition *in situ*.

La composition adapte la préoccupation à son contexte d'utilisation. Cette adaptation est fréquemment un mélange d'adaptations structurelles et comportementales. Cependant, AspectJ privilégie l'adaptation comportementale tandis qu'Hyper/J privilégie l'adaptation fonctionnelle. L'adaptabilité d'une préoccupation est donc fonction des possibilités du langage utilisé.

Pour être facilement réutilisable, une préoccupation doit avoir un protocole de composition. Ce dernier définit la composition générique à utiliser pour adapter la préoccupation à son contexte d'utilisation. Ce protocole est ensuite spécialisé par héritage pour s'adapter au contexte d'utilisation. Cependant AspectJ n'a permis d'implémenter qu'une partie du protocole de composition de l'observateur et Hyper/J ne permet pas l'implémentation de protocoles de composition.

	AspectJ	Hyper/J	Java
Paradigme	Programmation orientée aspects	Programmation orientée sujets	Programmation orientée objets
Entité qui représente une préoccupation	Aucune	Un sujet	Aucune
Entité qui encapsule la composition	Aspect	Module de composition	Classe(s)
Opération sur les entités qui encapsulent la composition	Héritage (limité) [HAN 01]	Agrégation	Héritage et Agrégation
Nature de la composition	In situ	Ex situ	In situ
Type d'adaptation prédominant	Comportementale	Fonctionnelle	Les deux
Protocole de composition	Encapsulation partielle	Aucune encapsulation	Aucune encapsulation

Tableau 1. Caractéristiques d'AspectJ, d'Hyper/J et de Java.

Ces constatations (résumées dans le tableau 1) ainsi que [DEV 01] [HAN 01] [OST 00] [VAN 01] montrent que les modèles actuels les plus aboutis (AspectJ et Hyper/J) ne sont pas pleinement satisfaisants. Nous pensons que ceci est dû aux orientations des approches par sujets et par aspects qui repose sur deux modèles

orthogonaux. Ces derniers possèdent *ensemble* les qualités nécessaires pour réutiliser et adapter convenablement le patron de conception. Nous souhaitons donc les réconcilier à travers un nouveau modèle.

5. Notre approche : une réconciliation entre objet, aspect et sujet

Notre approche est une réconciliation entre objet, aspect et sujet pour bénéficier des avantages cumulés des trois approches.

Nous avons choisi, comme AspectJ, d'encapsuler les préoccupations par des ensembles de classes représentées par des packages en Java.

La composition de préoccupation(s) est encapsulée par une entité nommée *adaptateur* : la composition d'une préoccupation *adapte* la préoccupation au contexte de ses clients. L'adaptateur possède comme une classe, l'abstraction qui permet d'encapsuler un protocole de composition dans un adaptateur abstrait et l'héritage pour concrétiser ces derniers. Nos adaptateurs sont une amélioration des aspects d'AspectJ et des modules de composition d'Hyper/J avec une relation d'héritage plus complète comme le préconise [HAN 01] et [DEV 01].

La composition produite par un adaptateur [OST 00] est au choix *in situ* (comme AspectJ) ou *ex situ* (comme Hyper/J). Les adaptateurs supportent les deux types d'adaptation : fonctionnelle (inspiré d'Hyper/J) et comportementale (inspiré d'AspectJ).

L'exemple est implémenté par deux adaptateurs (comme avec AspectJ) : un adaptateur abstrait [VAN 01] et un adaptateur qui spécialise le premier pour réutiliser le patron de conception.

L'adaptateur abstrait `ProtocoleObservateur` encapsule le protocole de composition du patron observateur ; il fait donc aussi partie du même package que les classes de la figure 1.

```

1 package patrondeconception;
2 abstract adaptor ProtocoleObservateur {
3     abstract Class classeObservable;
4     abstract Method methodesDeclanchantNotification[];
5     introduce ImplObservable in classeObservable;
6     after (classeObservable.methodesDeclanchantNotification)
7         do { notifieObservateurs(); }
8
9     abstract Class classeObservateur;
10    implements Observateur in classeObservateur;
11 }

```

} Partie
Observable

} Partie
Observateur

La partie observable déclare deux variables : `classeObservable` (ligne 3) jouant le rôle de l'entité observable et `methodesDeclanchantNotification` (ligne 4) qui est un ensemble de méthodes déclenchant un appel à `notifieObservateurs` grâce à l'opérateur d'adaptation comportementale **after** (lignes 6 à 7). La variable

`classeObservable` est ensuite utilisée par l'opérateur fonctionnel **introduce** (ligne 5) afin de fusionner la classe `ImplObservable` dans `classeObservable`.

La partie de l'observateur (lignes 9 à 10) déclare une variable `classeObservateur` qui est la classe tenant le rôle d'observatrice. Cette classe est ensuite adaptée fonctionnellement par l'opérateur **implements** qui permet d'ajouter de nouvelles interfaces à une classe.

L'adaptateur `Application_IHM` décrit ci-dessous concrétise l'adaptateur `ProtocoleObservateur` et spécialise la composition pour notre exemple. Cet adaptateur est l'équivalent de l'aspect `Application_IHM` ou du module de composition `Application_IHMetendue`. Il adapte la classe `Button` pour la rendre observable et adapte la classe `Label` pour qu'elle soit observatrice.

```

12 package application.ihm;
13 import patrondeconception;
14 adaptator Application_IHM extend ProtocoleObservateur {
15     classeObservable = Button;
16     methodesDeclanchantNotification = Click; } Partie Observable
17
18     classeObservateur = Label;
19     introduce classeObservateur.miseAJour(Observable o) { } Partie
20         colorCycle(); } Observateur
21 }
22 }

```

L'adaptateur `Application_IHM` est concret. Puisqu'il fait partie du package `application.ihm` (ligne 12) il modifie à l'intérieur (*in situ*) du package `application.ihm` les classes `Button` et `Label`.

La ligne 15 attribue à la classe `Button` le rôle de l'observable et la ligne 18 la classe qui observe, c'est à dire `Label`. La ligne 16 indique que la méthode `Click` de `Button` doit déclencher une mise à jour des observateurs. Une nouvelle méthode est ajoutée à `Label` (lignes 19 à 21, grâce à l'adaptation fonctionnelle fournie par le mot clef **introduce**) pour que l'observateur réagisse aux changements des éléments qu'il observe.

Cet exemple montre que notre approche a permis de composer le patron de manière non intrusive avec ses classes clientes.

Notre approche par rapport à `AspectJ` ou à `Hyper/J` a permis d'encapsuler complètement le protocole de composition de l'observateur par un adaptateur abstrait. Malgré cela, comme notre approche est constituée des avantages respectifs d'`AspectJ` et d'`Hyper/J`, nous envisageons d'utiliser des techniques d'implémentation basées sur la transformation de programme, pour projeter notre approche vers ces langages.

La raison qui rend le patron de conception de l'observateur plus facilement réutilisable par rapport à `AspectJ` et `Hyper/J` est que l'utilisateur ne doit s'appuyer que sur quatre éléments quand il réutilise l'observateur. Ces quatre éléments

représentent le fonctionnement synthétique du patron : la classe jouant le rôle d'entité observée, les méthodes à observer de cette classe, la classe représentant l'observateur et la réaction de l'observateur lorsque qu'une des méthodes à observer est déclenchée.

Ces quatre éléments sont les éléments indispensables que doit connaître et comprendre l'utilisateur du patron de conception observateur et notre approche a permis de les expliciter à la fois dans le protocole de composition (par l'adaptateur abstrait) et lors de sa composition avec le client à travers l'adaptateur concret. Ceci permet d'envisager des contrôles sur la validité de la composition.

6. Conclusion et perspectives

L'étude pragmatique de la facilité de réutilisation d'un patron de conception a permis de mettre en évidence les lacunes des trois approches évaluées (objet, aspect et sujet). L'approche qui a été présentée ensuite apporte une solution à ces manques ; elle nécessite néanmoins d'être appliquée à d'autres exemples comme les canevas, les composants, les objets métiers pour être pleinement validée. Nous espérons ainsi généraliser l'approche de façon à élaborer un modèle plus général à l'image de celui d'AspectJ ou d'Hyper/J, tout en accordant, comparativement, une plus grande importance à la facilité de réutilisation.

L'étude a aussi montré les limites des approches orientées aspects et sujets en ce qui concerne la réutilisation des entités qui encapsulent la composition : carence partielle pour la première et absence totale pour la seconde. Ces limites proviennent de la faiblesse des mécanismes d'héritage entre ces entités. Notre approche ne souffre pas de ce problème, car elle bénéficie d'une relation d'héritage plus complète. Nous pensons approfondir cette voie par une étude de la sémantique des liens (héritage, agrégation, etc.) qu'il est possible d'exprimer entre les entités qui encapsulent la composition.

D'autre part, aspects et sujets se distinguent par un manque évident de collaboration entre eux, car ils définissent deux types de composition qui sont orthogonales. Une composition plutôt orientée adaptation comportementale pour les aspects et adaptation fonctionnelle pour les sujets. Ces deux types d'adaptation sont nécessaires pour le patron observateur ; notre approche tire donc ses avantages de l'hybridation entre les aspects et les sujets, pour intégrer les deux types de compositions.

La symbiose entre le monde des objets et les nouveaux paradigmes comme la séparation des préoccupations est encore trop jeune pour montrer tous les bénéfices de telles unions. Ceci est principalement dû, à la fois au manque de prise en compte de certains apports de l'objet, et à la trop faible diversité des applications de ces nouvelles méthodes de développement. C'est néanmoins une direction prometteuse qui devrait contribuer à améliorer de la réutilisation des applications.

7. Bibliographie

- [ASP 02] Aspect J, <http://www.aspectj.org>
- [DEV 01] De Volder D., « Code Reuse, an Essential Concern in the Design of Aspect Languages ? », *Workshop ASoC ECOOP 01*, 2001.
- [GAM 99] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Catalogue de modèles de conception réutilisables*, Addison-Wesley Publishing Co., 1999.
- [HAC 02] Hachani O., Bardou D., « Using Aspect-Oriented Programming for Design Patterns Implementation », *Workshop Reuse in OOIS 02*, 2002.
- [HAN 01] Hanenberg S., Unland R., « Using and Reusing Aspects in AspectJ », *Workshop ASoC OOSPLA 01*, 2001.
- [HAR 93] Harrison W., Ossher H., « Subject-Oriented Programming (A Critique of Pure Objects) », *OOPSLA 93*, 1993.
- [HYP 02] The HyperSpace HomePage, <http://www.research.ibm.com/hyperspace/>.
- [KIC 97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C.V., Loingtier J.M., Irwin J., « Aspect Oriented Programming », *ECOOP 97*, 1997.
- [LOP 95] Lopes C.V., Hirsch W., « Separation of Concerns », TR NU-CCS-95-03, Northeastern University, Boston, February 1995.
- [NOD 01] Noda N., Kishi T., « Implementing design patterns using advanced separation of concerns », *Workshop ASoC OOSPLA 01*, 2001.
- [OSS 00] Ossher H., Tarr P., « Hyper/J: Multi-Dimensional Separation of Concern for Java », *ICSE 00*, 2000.
- [OST 00] Ostermann K., Kniesel G., « Independent Extensibility – an open challenge for AspectJ and Hyper/J », *Workshop Aspect and Dimension of Concern ECOOP 00*, 2000.
- [OST 01] Ostermann K., Mezini M., « Object-Oriented Composition is Tangled », *Workshop ASoC ECOOP 01*, 2001.
- [VAN 01] Vanhaute B., De Win B., De Decker B., « Building Frameworks in AspectJ », *Workshop ASoC ECOOP 01*, 2001.