Réification des schémas de conception : une expérience

Stéphane Ducasse

Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, CH-3012 Bern, Switzerland
ducasse@iam.unibe.ch, http://iamwww.unibe.ch/∼ducasse/
Appeared in LMO'97

RÉSUMÉ. *Alors même que les schémas de conception sont devenus un moyen populaire afin de décrire des solutions à des problèmes de conception, la question de savoir si les schémas doivent être intégrés dans les langages (réification, constructeurs) reste posée. Dans cet article après avoir identifié les problèmes liés à l'implémentation des schémas de conception et présenté certaines approches, nous montrons comment les connecteurs du langage* FLO *permettent une représentation et implémentation de certains schémas. Ensuite, l'évaluation de cette approche montre qu'elle offre une meilleure intégration que les solutions basées sur l'introduction de nouvelles instructions dans le langage grâce à la composition d'abstractions élémentaires de l'envoi de messages.*

ABSTRACT. *Although design patterns are becoming increasingly popular as a way to describe solutions to general design problems, question about their integration in language (reification or language support) stays an open question. In this article, after identifying the implementation problems and presenting some approaches, we show how* FLO*'s connectors allows for the representation and implementation of some design patterns. Thus the evaluation of this approach outlines the fact that such an approach by composing elementary message passing abstractions offers a better integration than the approaches based on construct introduction in the language.*

MOTS-CLÉS : *schémas de conception, langage, interactions entre objets*
KEYWORDS: *design pattern reification, language support, interactions between objects*

1. Introduction

Les schémas ou patrons de conception *(design patterns)* sont devenus un moyen populaire afin de décrire des solutions à des problèmes de conception. Les schémas de conception [GHJV94, BJM$^+$96] sont utilisés comme un outil pour décrire de manière *abstraite* et *indépendante* du langage et de l'application, des solutions à des problèmes de conceptions. Ces solutions peuvent alors être spécialisées pour le besoin d'applications plus spécifiques. Diverses classifications ont été proposées suivant leur contexte [BJM$^+$96] ou leur but (création, structure ou comportement) [GHJV94]. Ainsi cer-

tains schémas de conception sont spécifiques à certains domaines et d'autres sont plus généraux.

La plupart des auteurs des schémas de conception considèrent que ceux-ci doivent restés indépendants du langage d'implémentation pour ne pas perdre de leur abstraction. Cependant, dans la communauté la question se pose de l'intérêt de l'introduction d'instructions explicites pour représenter les schémas de conception ou de leur réification [BHK97]. Cette question, qui au premier abord pourrait être jugée comme inutile ou allant à l'encontre du concept même des schémas de conception, est motivée par l'existence de problèmes rencontrés lors de leur implémentation. Bien que peu de travaux aient été menés à ce sujet, différents auteurs ont identifié les problèmes suivants : perte de l'entité conceptuelle créant une différence entre la conception et l'implémentation d'un schéma de conception, prolifération de classes, impossibilité de réutiliser l'implémentation du schéma de conception [Sou95, Bos97].

Notre travail s'inscrit dans une démarche d'analyse et de comparaison des besoins rencontrés lors de l'implémentation de schémas de conception et des différentes solutions apportées. L'expérience servant de base à notre réflexion a été de représenter explicitement certains schémas de conception par des entités représentant l'interactions entre objets. Cette expérience et ses limites nous a montré que des abstractions d'envoi de messages sont un moyen souple pour l'implémentation des schémas de conceptions. Ainsi à l'opposé d'approches basées sur l'introduction systématique d'instructions pour chaque schémas de conception au niveau du langage [Bos97], cette expérience montre que le contrôle des messages seul permet un pouvoir d'expression similaire et une plus grande flexibilité.

Dans cet article, nous précisons les problèmes posés par le manque d'une réelle prise en compte de schémas de conception au niveau des langages d'implémentation traditionnels. Nous présentons rapidement les solutions proposées dans la littérature. Ensuite, nous montrons comment la représentation explicite de certains schémas de conception par des entités représentant l'interaction entre objets apporte une solution dans le cas de schémas de conception et nous terminons par une analyse de notre solution et sa confrontation aux autres approches.

2. Schémas de Conception et Problèmes

Les schémas *(pattern)* prennent leur origine dans le travail de l'architecte C. Alexander [AIS+77]. Il définit un schéma comme la description d'un problème récurrent et d'une solution générique pouvant être adaptée à des situations spécifiques. Un ensemble composable de tels schémas définit alors un langage de schémas *(pattern language)*.

Le concept de schémas est apparu récemment dans la communauté objet sous le nom de *schémas de conception (design pattern)* comme étant une description abstraite d'un problème de conception et sa solution. Les auteurs de [GHJV94] définissent un schéma comme : "*description of communicating objects and classes that are customized to solve a general design problem in particular context*". Les schémas de conception sont utilisés durant la conception des applications. Ils identifient un contexte, les classes ou les objets participants, leurs responsabilités et leurs collaborations. De

possibles implémentations du schéma décrit sont souvent proposées dans le cadre de langage comme C++ ou SMALLTALK.

2.1. *Implémentation et Problèmes*

L'implémentation des schémas de conception à l'aide de langages objets traditionnels n'est pas sans poser de problèmes. Différents auteurs ont identifiés les points suivants [Bos97, Sou95] :

Perte de l'entité de conception. Les langages objets en ne proposant pas d'instructions ou d'entités explicites pour la représentation des schémas obligent le programmeur à disperser la sémantique du schéma dans un ensemble de méthodes et d'échanges de messages. Il n'y a plus alors de correspondance directe entre les entités présentes lors de la phase de conception et le code. J.BOSCH évoque ce problème en parlant d'un problème de traçabilité *(traceability problem)*. Ce problème est aussi identifié dans [Sou95].

Prolifération de classes. L'implémentation d'un schéma de conception nécessite l'introduction de nouvelles classes ayant souvent des comportementaux triviaux comme la redirection de messages et de changer le code des classes existantes. Ceci conduit à une augmentation de la complexité des applications et à une profusion de classes [BFVY96].

Réutilisation délicate. Alors que les schémas de conception sont réutilisables au niveau de la conception des applications, le fait qu'ils ne soient pas représentés explicitement empêche la réutilisation de leur implémentation [BFVY96, Bos97]. Chaque fois qu'un schéma de conception est utilisé, le programmeur doit le réimplémenter.

De plus, J.SOUKUP dans [Sou95] précise que la composition de plusieurs schémas de conception peut mener à des groupes de classes mutuellement dépendantes ce qui rend délicat leur réutilisation.

2.2. *Approches pour le support des schémas de conception.*

Divers travaux proposent des solutions pour offrir un support pour les schémas de conception. Ils peuvent être classés en deux catégories :

Environnements. Dans cette approche, les schémas de conception sont pris en compte au niveau des environnements et non au niveau du langage d'implémentation.

Une première voie propose de générer du code à partir de modèles de conception intégrant les schémas de conceptions. Ainsi pour chaque schéma utilisé lors de la conception des classes sont générées. Certaines classes peuvent être impliquées dans plusieurs schémas de conception, ce qui conduit à composer le code de la classe à partir de ces différents schémas. Après la génération du code, du code spécifique à

l'application peut être introduit dans la définition des classes. Un exemple d'une telle approche est présentée dans [BFVY96]. Une limite de cette approche est son unidirectionnalité : il est difficile de retrouver la conception originale à partir du code.

Une seconde voie prend en compte des schémas au niveau de l'environnement de programmation. Ceux-ci sont alors représentés à un méta-niveau [1] et l'environnement est capable de mettre en relation les schémas de conception et le code. Elle s'applique aussi bien durant la phase initiale de conception [EGY97] que dans les phases de rétro-conception [FMvW97].

Par exemple, dans WIZARD [EGY97], les auteurs représentent un schéma de conception comme un ensemble de *tricks*. Un trick est une opération élémentaire de manipulation du source code du langage d'implémentation définie au niveau du méta-langage. Ainsi un schéma de conception est réifié mais au méta-niveau et non au niveau du langage d'implémentation. Seul l'*effet* de l'application du schéma de conception *représente* le schéma à ce niveau. De plus, afin de pouvoir recharger des applications définies à l'aide de Wizard, les tricks sont définis afin que leur action sur le code les authentifie de manière unique.

Extensions des langages. La dernière approche introduit les schémas de conception au sein même des langages de programmation. Cette introduction peut se faire par la définition d'instructions spécifiques représentant les schémas de conception dans un modèle objet basé sur des filtres [ABV92, Bos97], par la définition de macros [Sou95], par l'annotation du code à l'aide d'attributs représentant le schéma [Hed97]. Cette dernière approche permet aussi de détecter de manière statique de possibles erreurs lors de l'évolution des participants. Le travail que nous présentons se situe dans cette dernière catégorie.

3. Le modèle FLO

Nous présentons rapidement le modèle qui est décrit plus en détail dans [Duc97, DR97]. FLO étend les modèles objets traditionnels par la prise en compte de connecteurs des entités responsables de l'interaction entre objets. Trois entités sont distinguées: les objets, les classes et les connecteurs.

3.1. *Objets, classes et connecteurs*

Dans FLO un objet est défini par son comportement et non par sa structure. Comme dans le modèle OBJVLISP, un objet est une boîte noire qui ne communique que par envoi de messages. Chaque objet est instance d'une classe qui définit la structure et le comportement *intrinsèques* de toutes ses instances.

Dans FLO, le comportement d'un objet dépend de son *environnement*. L'environnement d'un objet est l'ensemble des connexions, interactions ou dépendances avec

1. Ce méta-niveau peut être réflexif ou non, c'est-à-dire le langage de spécification du schéma peut être semblable ou différent du langage d'implémentation.

d'autres objets. Contrairement aux modèles traditionnels dans lequels ces informations sont dispersées dans les classes des objets en connexions, dans FLO un environnement est exprimé au moyen d'entités explicites nommées des *connecteurs*. Les connecteurs expriment localement et déclarativement l'influence de la réception d'un message par un objet participant à une connexion sur les autres objets participants.

3.2. *Connecteurs*

Un connecteur est un objet qui représente la connexion entre d'autres objets nommés *participants*. Un même objet peut participer à plusieurs connecteurs. Un connecteur est spécifié par un constructeur qui décrit toutes les informations nécessaires à la connexion. En particulier il permet de spécifier le *comportement dynamique* de la connexion par la donnée de *règles d'interactions* qui précisent comment les échanges de messages influencent le comportement des participants.

```
(defconnector  connectorAB  (:roleA :roleB)   ; liste de noms de roles
    :inherit ((...))              ; liste de connecteurs
    :var                           ; variables de connecteurs
    :behavior                     ; règles d'interactions
)
```

Suite à sa définition, un connecteur est instancié entre des objets. Durant la connexion, un connecteur intercepte certains messages et contrôle que la spécification décrite par le comportement dynamique est remplie. De plus, un connecteur décrit le contexte de la connexion : ainsi certaines actions doivent être invoquées sur les objets participants avant ou après leurs connexions.

Comportement dynamique. Il est défini par un ensemble de règles d'interactions qui spécifient comment les messages réçus par les objets participants doivent être contrôlés : l'exécution de la méthode est-elle autorisée, sous quelle condition, la réception implique-t-elle l'envoi d'autre messages, le message doit-il être redirigé...? Les règles possèdent la syntaxe simplifiée suivante, leur sémantique différant selon l'opérateur de la règle :

Rule	`::=`	Filter Operator Action
Filter	`::=`	Selector Rolename List-Of-Calling-Args
Context	`::=`	Message^{+}
Message	`::=`	Selector Rolename Args
Operator	`::=`	`implies` | `permitted-if` | `corresponds`

Le *filtre* d'une règle spécifie quel message (sélecteur et arguments d'appels) doit être intercepté lorsqu'il est envoyé à un participant défini par un *nom de rôle*. L'opérateur définit la sémantique de la règle en spécifiant le *l'action* de la règle. `implies`,

`permitted-if` and `corresponds`, les trois opérateurs pré-définis[2] de FLO ont la sémantique suivante :

La propagation est spécifiée à l'aide de l'opérateur `implies`. Après la réception d'un message et l'exécution de la méthode, d'autres messages, nommés *messages compensatoires*, sont envoyés aux participants.

L'inhibition est spécifiée à l'aide de l'opérateur `permitted-if`. Le message reçu n'est exécuté que si une certaine condition, nommée une *garde*, est remplie. L'action est alors une expression booléenne spécifiant la garde.

La délégation est spécifiée à l'aide de l'opérateur `corresponds`. A la place de l'exécution de la méthode associée au message reçu, un nouveau message est émis.

3.3. *Contrôle de l'envoi de messages et filtrage.*

FLO assure que lorsqu'un message est envoyé les règles d'interactions sont appliquées si nécessaires. Ainsi les gardes sont évaluées, les messages compensatoires ou délégués sont automatiquement émis. De plus, seuls les messages, dont le sélecteur est spécifié dans une règle d'interaction, envoyés à un objet participant, sont contrôlés. Les autres messages sont normalement exécutés.

Figure 1. *Une exemple de filtrage :* `val1` *représente l'argument effectif* `12` *du message contrôlé (*`compute-value c1 12`*).*

Filtrage. Lors du contrôle des messages, les arguments d'un filtre sont liés avec les paramètres effectifs de l'appel par un mécanisme de filtrage. Ce mécanisme permet de manipuler ces arguments. Ainsi à la manière des Adaptors [YS94], un connecteur peut changer l'ordre et le nombre des arguments ou convertir des arguments. La figure 1 illustre cela : la règle d'interaction précise que lors du contrôle d'un message de sélecteur `compute-value` envoyé à l'objet de nom de rôle `:calculator` ayant pour argument `val1`, la réaction envoie le message `add-value` à l'objet de nom de rôle `:displayer` avec comme argument le résultat l'exécution de la méthode `convert` sur le connecteur lui même et à l'argument dénommé `val1`.

2. Cependant, FLO est un langage ouvert [Duc97] et son protocole de méta-objets permet la définition de nouveaux opérateurs.

4. Implémentation de schémas de conception

Les connecteurs de FLO peuvent servir de support pour l'implémentation des schémas de conception. Cette implémentation apporte une solution aux problèmes énoncés en 2 : la perte du schémas de conception comme une entité explicite et l'accroissement de la complexité du code. Nous illustrons cette approche en montrant comment certains schémas définis dans [GHJV94] peuvent être ainsi implémentés. La présentation des schémas de conception est brève due à des limites de places, plus d'informations peuvent être trouvées dans [GHJV94, BJM+96]. Dans la suite nous nous référons plus précisément à [GHJV94]. De plus, LAYOM étant un des rares langages à prendre en compte les schémas de conception, nous utilisons les exemples définis dans [Bos97] (voir 5.1) afin de faciliter la comparaison des deux approches (voir 5).

4.1. *Schémas structuraux*

Par contraste avec les schémas de conception comportementaux qui décrivent les interactions entre objets, les schémas de conception structuraux décrivent comment les classes et les objets sont composés pour former de nouvelles structures.

4.1.1. *Adaptateur*

But. L'Adaptateur convertit l'interface d'un objet pour qu'il offre une interface compatible avec les attentes des autres objets. Il permet la coopération d'objets ayant des interfaces incompatibles.

Implémentation Traditionnelle. L'Adaptateur est traditionnellement implémenté au moyen d'un objet qui redirige les messages après adaptation à l'objet adapté. La figure 2 extraite de [GHJV94] décrit les relations entre l'objet adapté et l'adaptateur.

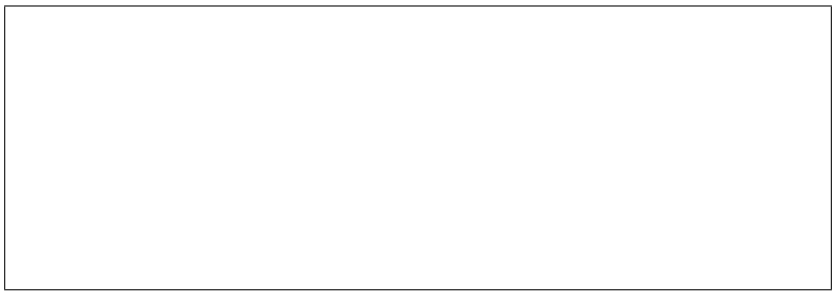

Figure 2. *Un adapateur basé sur la composition et la redirection de messages.*

Bien que cette solution permette la coopération entre objets qui n'auraient autrement pas coopérer, elle possède les désavantages suivants : premièrement, il est néces-

saire d'écrire des méthodes qui redirige les messages pour toutes les méthodes devant être adaptées. De telles méthodes servent alors juste de relais. En second, même les méthodes ne nécessitant pas d'adaptation doivent être adaptées.

Solution. Notre solution est basée sur l'utilisation de l'opérateur de délégation (`corresponds`). Un connecteur, nommé `AdapterForGraphObject`, définit un rôle nommé ici `adaptee` ainsi que les messages devant être adaptés.

```
(defconnector AdapterForGraphObject (:adaptee)
 :behavior
   (((mess1  :adaptee a1 a2) corresponds (newMessA  :adaptee a2  (fct a2) a1))
    ((mess2  :adaptee a1 a2) corresponds (newMessB  :adpatee a2 a1))
    ((mess3  :adaptee a1 a2) corresponds (newMessB  :adaptee a2 a1))))

(make AdapterForGraphObject :adaptee graph)
```

Cette définition spécifie que les messages envoyés (`mess1`, `mess2` and `mess3`) à un objet, nommé `adaptee`, sont contrôlés et donc remplacés par leur message correspondant. Le message `mes1` est remplacé par le message `newMessA` et `mes2` et `mess3` par `newMessB`. De plus, notons que le mécanisme de filtrage que nous avons présenté en 3.3 permet de changer l'ordre des arguments, de transformer les arguments lors de l'adaptation comme illustré par l'application de la fonction *fct* dans la première règle d'interaction.

Cette solution se situe au niveau des instances car l'instanciation du connecteur ne s'applique que sur les instances spécifiées. Pour adapter toutes les instances d'une classe, il suffit de redéfinir la méthode de création de la classe et d'instancier le connecteur sur chacune des instances créées.

Outre une meilleure abstraction du schéma de conception au niveau de l'implémentation, cette approche nécessite seulement de définir les méthodes devant être adaptées, les autres méthodes restent inchangées et ne sont donc pas contrôlées.

4.1.2. *Façade*

But. Le schéma de conception Façade est utilisé pour donner une interface unifiée à un ensemble d'interfaces d'un sous-système simplifiant ainsi son utilisation.

Implémentation Traditionnelle. Une implémentation traditionnelle de la Façade est de définir une classe ayant pour composants les différentes parties du sous-système. Le but de ce schéma est double : il aide à la coordination des classes du sous-systeme et offre une interface uniforme. Le premier but est bien rempli par les implémentations traditionnelles. Par contre pour le second but, un problème similaire à celui présenté dans l'Adaptateur arrive : il est nécessaire d'introduire des méthodes redirigeant les messages.

Solution. Nous définissons un connecteur qui représente le schéma et nous utilisons une nouvelle fois l'opérateur de délégation.

La définition suivante précise que les messages `mess1` et `mess2` envoyés au connecteur lui-même correspondent après une possible adaptation à des messages envoyés à l'objet représenté par la variable `:Part1`. De même, `mes3` est redirigé à l'objet `:Part02`.

```
(defconnector   FacadebetweenPt1Pt2 (:Part01 :Part02)
  :behavior
  (((mess1 connector a1 an) corresponds (mess1  :Part01 a1 an))
   ((mess2 connector a2)    corresponds (mess1  :Part01 a2))
   ((mess3 connector a3)    corresponds (mess3  :Part02 a3))))
(make FacadebetweenPt1Pt2 :Part01 composant1 :Part02 composant2)
```

Notons que le mot-clé `connector` fait référence à l'instance de connecteur. Ainsi, les messages qui lui sont envoyés sont redirigés sur les objets composants le sous-système.

4.2. *Schémas de conception comportementaux*

Les schémas comportementaux ne décrivent pas seulement des schémas structurels entre objets mais ils spécifient des schémas de communication entre ces objets.

4.2.1. *Observateur*

But. Ce schéma de conception définit une dépendance entre un et plusieurs objets. Ainsi lorsqu'un objet, nommé *sujet*, change d'état tous ses dépendants, nommé *observateurs*, sont notifiés et mis à jour.

Implémentation Traditionnelle. L'implémentation traditionnelle est basée sur l'utilisation d'un mécanisme de notification. L'objet observé émet une notification à ses observateurs lors de changements pouvant rendre inconsistante sa relation avec les objets observateurs. La figure 3 illustre le schéma de communication entre l'observé et les observateurs.

A côté du fait que le schéma est perdu lors de l'implémentation dans les classes de l'observé et des observateurs, cette approche n'est pas exempte de problèmes. Tout d'abord la séparation souhaitée entre observé et observateurs n'est pas réelle. Les observateurs sont liés à l'observé par l'implémentation de leur méthode de remise à jour (update). Il existe un couplage fort entre ces classes. D'autre part, la notification est souvent trop imprécise : lorsqu'un objet observé émet une notification tous les observateurs sont avertis et les méthodes de mises à jour sont invoquées, comme c'est le cas dans l'utilisation des dépendances en SMALLTALK (Notons sur ce dernier point que ce mécanisme peut être amélioré comme c'est le cas dans VISUALWORKS2.0 qui permet de définir l'événement et l'objet observateur lors de la notification). Notons que [KB95] propose différentes implémentations de ce schéma. Troisièmement, la notification automatique des changements d'états nécessite la modification de toutes les méthodes de l'observé étant impliquées dans l'observation ainsi que la définition des

Figure 3. *Relations structurelles entre les classes des objets participants à un schéma Observateur et collaborations entre le sujet et les observateurs.*

réactions dans chacun des observateurs. La sémantique de l'observation est dispersée dans les classes que l'on doit parcourir afin de suivre le flot de propagation.

Une première solution. Si l'on choisit de suivre à la lettre l'implémentation proposée dans [GHJV94] qui illustrée par le protocole de la figure 3, on peut définir le connecteur suivant :

```
(defconnector Observer (:subject :observers)
  :behavior
  (((setState :subject val) implies (map update :observers))))

(make Observer :subject suj1 :observers (list obs1 obs2 obs3))
```

Lorsque le sujet reçoit le message `setState`, après l'exécution de la méthode associée, le message `update` est invoqué automatiquement sur chacun des observateurs. Cependant comme FLO est basé sur le contrôle des messages, la méthode réalisant la notification (i.e. `changed` en SMALLTALK) et le mécanisme même de notification sont inutiles. La notification est implicite dans les règles d'interactions.

Une seconde solution. La précédente solution est une stricte application de la description du protocole décrit par la figure 3. Les connecteurs permettent une implémentation plus souple de ce schéma. Tout d'abord, plusieurs sortes d'observateurs peuvent être distingués recevant alors des messages de notification différents. Ensuite, l'utilisation d'opérateurs comme l'opérateur `implies-before` qui permet d'émettre la réaction avant l'exécution de la méthode[3] peut permettre d'avoir une sémantique différente. La présence d'une méthode `setState` n'est pas nécessaire car les règles d'interactions peuvent être définies sur n'importe quelle méthode de l'observé. En

3. Un tel opérateur est nécessaire pour la mise en œuvre de certaines contraintes d'exclusion comme n'avoir jamais deux boutons d'une liste de boutons sélectionnés en même temps. Ainsi, la connexion ne sera pas dans un état incohérent, même pendant la phase de réaction.

troisième, la définition de la méthode `update` dans la classe des observateurs peut être évitée en définissant la *réaction* au niveau du connecteur. Notons cependant que pour cela les classes des observateurs doivent rendre publiques les méthodes ou l'accès aux variables d'instances permettant de définir de telles actions.

Évaluation. Le schéma de conception Observateur en étant clairement représenté par le connecteur n'est plus dispersé dans les classes des participants. Les flots de messages et les implications sont logiquement regroupés dans une entité unique. De plus les classes des participants n'ont pas à être modifiées et les classes représentant le sujet et les observateurs ne sont plus liées par le mécanisme de notification et de maintient de la cohérence.

4.2.2. *Médiateur*

But. Le but de ce schéma de conception est de définir un objet, nommé *médiateur*, qui encapsule l'interaction d'un groupe d'objets, nommé *collègues*. Il permet un couplage souple entre les objet en évitant qu'ils ne se référencent explicitement mutuellement. Chaque objet peut alors adapter son interaction de manière indépendante.

Approche traditionnelle. Une implémentation de ce schéma de conception est basée sur la création d'une classe représentant le médiateur. Les objets ayant besoin de coopérer font uniquement référence à cette classe qui est responsable de la coordination des objets. Ce schéma peut être perçu comme une généralisation du schéma Observateur, aussi les mêmes problèmes sont rencontrés lors de l'implémentation : le schéma n'est pas explicite, de nombreuses méthodes doivent être modifiées pour mettre en place des messages de notification.

Solution. Selon les besoins, notre solution peut faire participer ou pas le médiateur. Ainsi supposons qu'un collègue A doive exécuter une action après une action exécutée par un collègue B, le schéma de conception peut s'écrire comme suit:

```
(defconnector Mediator1betweenC1C2 (:CColleague1 :CColleague2)
 :behavior
   (((B :CColleague1 a1 an) implies (A :CColleague2 a1))))

(make Mediator1betweenC1C2 :CColleague1 obj1 :CColleague2 obj2)
```

Cependant, si le médiateur participe comme dans la solution proposée par J.BOSCH, il est possible de spécifier le connecteur comme suit :

```
(defconnector Mediator2betweenC1C2 (:CColleague1 :CColleague2)
   :behavior
   (((messA connector b1 b2) corresponds (messA :CColleague1 b1 b2))
    ((messB connector a1 an) corresponds (messB :CColleague2 a1 an))))

(make mediator :CColleague1 obj1 :CColleague2 obj2)
```

5. Travaux relatifs et évaluation de notre approche

J.SOUKUP dans [Sou95] propose une implémentation de schémas structuraux à l'aide de macros et du pré-processeur de C++. Il propose une bibliothèque de schémas. La solution proposée peut s'apparenter à des templates C++[4]. Il découple ainsi l'implémentation des classes des participants de celles des schémas de conception. J.SOUKUP explique que cette approche n'est utilisable que pour les schémas structuraux faisant intervenir au moins deux participants.

La comparaison entre notre approche est celle de LAYOM[5]) [Bos97] est plus appropriée car les deux approches sont basées sur des langages extensions des modèles traditionnels ayant la particularité d'introduire un contrôle de l'envoi de messages. Dasn ce but, nous présentons rapidement deux *couches* extraites de [Bos97] permettant d'implémenter des schémas de conception en LAYOM. Ensuite, à la lumière de cette comparaison nous évaluons notre approche.

5.1. *Schémas de conception en* LAYOM

Pour chaque schéma de conception, J.BOSCH définit une nouvelle *couche* qui encapsule l'état interne de l'objet et contrôle les messages reçus ou émis.

Adaptateur. Une nouvelle couche est introduite qui spécifie parmi les messages recus ceux devant être redirigés. Suit la syntaxe et une utilisation de cette couche dans une classe nommée `AdapterForGraphObject`.

```
<i> : Adapter ( accept <mess-sel>+ as <new-mess-sel> , ...);

class AdapterForGraphObject
    layers
      adapt : Adapter( accept mess1 as newMess1,
                       accept mess2, mess3 as newMessB);
      inh : Inherit(Adaptee);
```

Médiateur. Une nouvelle couche est définie, nommée `Mediator`, qui spécifie que certains messages envoyé par un `client` alors sont redirigés sur une autre objet.

4. En fait, les templates ne permettent pas de manipuler les noms des variables d'instance ou des méthodes, contrairement aux macros.

5. Layered Object Model qui est inspiré des Filtres de Composition [ABV92, Bos95] est basé sur la définition de couches qui encapsulent un objet et contrôlent les messages recu et émis

```
<id> : Mediator (forward <mess-sel>+ from <client> to <object> ...);

class ConcreteMediator
  layers
    med : Mediator(forward messA from Any to ConcColleague1,
                   forward messB from ConcColleague1 to ConcColleague2);
```

Façade. J.BOSCH définit une nouvelle couche nommée `Facade` dont la syntaxe et l'utilisation sont les suivantes :

```
<id> : Facade( forward <mess-sel>+  to <object>, ...);

class facade
 layers
     face : Facade ( forward mess1, mess2  to PartO1
                     forward mess3  to PartO2);
     partO1 : PartOf(ClassofO1);
     partO2 : PartOf(ClassofO2);
```

5.2. *Comparaison entre* FLO *et* LAYOM

Contrôle de l'envoi de messages. FLO et LAYOM sont deux extensions des modèles objets traditionnels basées sur le contrôle de l'envoi de messages[Bos95, Duc97]. LAYOM contrairement à FLO contrôle aussi les messages émis par un objet. Par contre, il n'offre pas de mécanisme général de manipulation des arguments des méthodes. Ainsi, les schémas proposés n'offre pas la possibilité de manipuler les arguments (voir les schémas Adaptateur, Façade... en 5.1) ce qui peut être un problème pour l'utilisation du schéma de conception.

Situation. En LAYOM, une couche est défini dans la classe de l'objet qu'elle contrôle. En FLO un connecteur est défini indépendamment des classes des objets sur lesquelles il va s'appliquer, ceci évite de devoir modifier ces classes.

Champ sémantique différent. En LAYOM, l'introduction d'un nouveau schéma de conception nécessite la définition d'une nouvelle couche. Pour cela, il faut décrire la nouvelle couche et définir un parseur et compilateur associé [Bos95]. Or ceci n'est théoriquement pas du ressort du programmeur mais du méta-programmeur. Ainsi la définition d'un schéma de conception ne se trouve pas dans le même champ sémantique que celui du langage utilisé par le programmeur.

En FLO, la définition d'un nouveau schéma est assurée par le programmeur qui en utilisant les opérateurs mis à sa disposition définit librement son schéma.

Flexibilité et minimalité. L'approche proposée par LAYOM est contraignante pour un programmeur. En effet, il n'est pas sûr que le programmeur ait besoin d'un schéma de conception qui corresponde exactement à la sémantique proposée par la couche. Par exemple, le fait que l'adaptateur en LAYOM ne permette pas une manipulation des arguments peut être restrictive. Cette trop grande précision de la syntaxe et surtout de la sémantique du schéma peut nuire à son utilisation et mener à une prolifération de schémas.

L'approche choisie en FLO a été au contraire de définir un nombre minimal d'abstractions d'envoi de message comme la délégation, la propagation et l'interdiction et de permettre au programmeur de définir ses propres schémas. Le programmeur spécifie et implémente exactement la sémantique du schéma de conception par la composition d'abstractions de messages.

5.3. *Réflexions sur notre approche.*

Notre solution, tout comme celle proposée par J.BOSCH, offre une meilleure abstraction des schémas de conception lors de leur implémentation. Les schémas sont des entités à part entière dont l'implémentation n'est plus dispersée dans les classes des objets participants. Il existe alors une équivalence entre les entités conceptuelles et leur implémentation. De plus, le contrôle des envois de messages permet d'éviter la définition de classes ne servant qu'à rediriger des messages.

À la lumière des exemples, une objection à notre approche peut être que le langage n'offre pas une claire distinction en terme d'instructions entre les différents schémas de conception. Ainsi les schémas Façade, Adaptateur et Médiateur sont proches. Cette remarque est vraie, cependant nous pensons que c'est le prix à payer pour une plus grande flexibilité. D'autre part, l'apport primordial des schémas de conception qui est d'après Doug Lea de proposer un vocabulaire uniforme formant une base de référence, est conservé par notre approche : le nom du schéma et des participants font partie intégrante du schéma et de son implémentation.

Notre approche nous a permis de modéliser certains schémas de conception. Cependant, ces schémas sont liés à des limites des langages traditionnels. Et nous sommes certains que cette approche n'est pas généralisable à tous les schémas.

6. Conclusion

Dans cet article, nous avons identifié les principaux problèmes liés à l'implémentation des schémas de conception dans des langages traditionnels : perte de l'entité de conception, prolifération de classes, réutilisation délicate.

Nous avons montré comment le langage FLO qui prend en compte les interactions entre objets sous la forme d'entités explicites, offre une solution à ces problèmes. Les schémas de conception sont explicitement représentés. De plus les classes de objets participants ne sont plus modifiés lors de leur participation à des schémas de concep-

tion. Les classes auparavant nécessaires pour la redirection des messages ne le sont plus.

La comparaison avec le travail de J.BOSCH autour du langage LAYOM nous a permis de mettre en avant la spécificité de notre approche. Tout d'abord, la définition d'un nouveau schéma de conception se fait au niveau même du langage et non par l'utilisation d'un méta-niveau, le programmeur est donc libre de spécifier exactement la sémantique du schéma dont il a besoin et n'a pas à composer avec la sémantique prédéfinie des schémas proposés. En second, au lieu d'introduire de nouvelles instructions pour chaque nouveaux schémas, l'approche privilégie la minimalité. En effet, dans le même esprit que les tricks de [EGY97], un ensemble d'abstractions élémentaires d'envoi de messages sont à la disposition du programmeur qui peut les composer à souhait pour définir la sémantique exacte de son schéma depuis le langage lui-même.

Comme un grand nombre des schémas proposés dans [GHJV94] sont des réponses aux lacunes des langages traditionnels en matière de prise en compte d'interactions entre objets, il n'est pas surprenant que les connecteurs permettent d'implémenter des schémas de conception comme ceux que nous avons présentés. Pour notre part, nous considérons seulement cette expérience que comme la preuve que le contrôle de l'envoi de messages apporte une aide pour l'intégration des schémas de conception. Cependant, la comparaison avec LAYOM pour sa part tend à prouver que l'introduction d'instructions n'est pas suffisante pour prendre en compte la flexibilité inhérente des schémas de conception.

Références

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of ECOOP'92, LNCS 615*, pages 372–395, 1992.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[BFVY96] F. Budinsky, M. Finnie, J. Vlissides, and P. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.

[BHK97] J. Bosch, G. Hedin, and K. Koeskimies, editors. *Ecoop'97 Workshop on Languages Support for Design Patterns and Object-Oriented Frameworks*, 1997.

[BJM+96] F. Buschmann, C. Jakel, R. Meunier, H. Rohnert, and M. Stahl. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley, 1996.

[Bos95] J. Bosch. *Layered Object Model: investigating paradigm extensibility*. PhD thesis, Lund University, Oct. 1995. Department of Computer Science.

[Bos97] J. Bosch. Design patterns as language constructs. *Accepted to JOOP*, 1997.

[DR97] S. Ducasse and T. Richner. Executable connectors: Towards reusable design elements. 1997. Accepted to ESEC'97.

[Duc97] S. Ducasse. *Intégration réflexive de dépendances dans un modèle à classes*. PhD thesis, Université de Nice-Sophia Antipolis, 1997.

[EGY97] A. Eden, J. Gil, and A. Yehudai. Precise specification and automatic application of design patterns. *JOOP*, may 1997.

[FMvW97] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proc. of ECOOP'97, LNCS 1241*, jun 1997.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Hed97] G. Hedin. Language support for design patterns using attribute extension. In Bosch et al. [BHK97].

[KB95] J. Kim and K. Benner. Implementation patterns for the observer pattern. In *Pattern Languages of Program Design 2*. Addison-Wesley, 1995.

[Sou95] J. Soukup. Implementing patterns. In *Patterns Languages of Program Design*, pages 395–412. Addison-Wesley, 1995.

[YS94] D. M. Yellin and R. E. Strom. Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors. In *Proc. of OOPSLA'94*, pages 176–190, 1994.