Redécouvrir les Solutions des Design Patterns

Mikal Ziane [*,**]

[*] *Laboratoire d'Informatique de l'Université Paris 6*
Pole IA 8, rue du Capitaine Scott 75015 Paris, France
Mikal.Ziane@lip6.fr
http://www-poleia.lip6.fr/~ziane/
[**] *Université René Descartes (Paris 5) Paris, France*

ABSTRACT. A design pattern can be seen as a <problem, solution> couple. Most formal descriptions of design patterns focus on the solution part and do not pay attention to the problem nor to the links between problems and solutions. In this paper we propose to express these problems using meta-variables which encapsulate a code fragment (awkwardly) expressing the intention of the developer. A generalization of the classical fold transformation will refactor this awkward term by displacing it to the proper functional or data abstraction. This generalized folding seems to be able to represent mechanisms pervasive in design patterns and to reproduce their solutions. This gives hope that much better tool support can be achieved than with today's approaches

RÉSUMÉ. Un design pattern peut être vu comme un couple <problème, solution>. La plupart des descriptions formelles des design patterns se concentrent sur la solution et néglige le problème ainsi que les liens qui les unissent. Dans ce papier nous proposons d'exprimer ces problèmes en utilisant des méta-variables qui encapsulent un fragment de code exprimant (maladroitement) l'intention du développeur. Une généralisation d'une classique transformation dite de pliage permet de corriger le terme intentionnel en le déplaçant vers une nouvelle abstraction fonctionnelle ou de données. Ce pliage généralisé semble en mesure de représenter des mécanismes constamment utilisés dans les design patterns et de reproduire leurs solutions. Ceci permet d'espérer que les design patterns pourront à l'avenir être beaucoup mieux outillés.

KEY WORDS: *design pattern, program transformation*

MOTS-CLÉS : *design pattern, transformation de programme*

1. Introduction

Le concept de design pattern[1] a eu un grand impact sur la programmation à objets. L'objectif était d'exprimer « l'expertise des concepteurs sous une forme que les gens puissent effectivement utiliser » [GAM 94]. De fait, les design patterns ont contribué à la diffusion d'un certain nombre de solutions éprouvées à des problèmes récurrents de conception à objets.

Malheureusement, on peut penser comme [AGE 98] que « le nombre de design patterns va croître à un point tel, qu'il deviendra impossible de savoir quels patterns existent et encore moins de savoir quels problèmes ces patterns résolvent vraiment ». Certaines personnes doutent encore de la validité de cette prédiction alors que les auteurs de [GAM 94] disaient eux même déjà : « Avec plus de 20 design patterns dans le catalogue parmi lesquels choisir, il peut être difficile de trouver celui qui répond à un problème de conception particulier » ! Si le problème était déjà présent avec 23 patterns, qu'en est-il aujourd'hui ou plus de 1000 patterns on été publiés [RIS 00] ?

Pour tirer pleinement profit de l'expérience capitalisée dans les design patterns nous pensons qu'un support par des outils est nécessaire. Or les design patterns n'ont pas été conçus pour être traités ou analysés par des ordinateurs [COP 94]. Il est donc nécessaire de traduire sous une forme exploitable automatiquement, l'expertise que les design patterns visent à disséminer.

Malheureusement, comme le dit [EDE 99b], la plupart des travaux sur la formalisation ou le support des patterns par des outils, s'est concentrée sur un seul aspect : la micro-architecture qui représente la solution du pattern. [BUD 96] décrit un outil qui crée des définitions et des déclarations de classes à partir de quelques informations telles que des choix pour le nom des classes ou pour décider entre un certain nombre de variantes. [LAU 98] représente un effort significatif pour décrire les patterns de façon plus abstraite que dans [GAM 94] de sorte que des solutions plus générales puissent être décrites de manière visuelle et néanmoins rigoureuse. [EDE 99a] décrit la micro-architecture générique des design patterns en utilisant une notation visuelle qui est équivalente à un sous-ensemble de la logique d'ordre supérieure monadique.

Notons qu'aucune de ces approches ne s'intéresse aux mécanismes fondamentaux qui sont sous-jacents aux design patterns. Tout concepteur qui connaît un certain nombre de patterns se rend pourtant vite compte que les même mécanismes reviennent constamment. Il est ainsi très fréquent qu'il s'agisse d'introduire une indirection pour désolidariser deux aspects qui doivent pouvoir varier indépendamment. La façon d'introduire cette indirection varie mais les idées fondamentales semblent, au moins intuitivement, assez peu nombreuses.

---

[1] Nous utilisons le terme anglais « design pattern » a défaut d'un terme français consensuel, le terme « modèle de conception » n'étant pas à notre avis compris par tout le monde.

[BOS 96] propose un langage de programmation qui étend le modèle à objets classique. En particulier ce langage supporte la notion de couche, ces dernières encapsulant les objets et interceptant les messages. Plusieurs patterns peuvent ainsi être élégamment définis à partir de quelques primitives de gestion de messages. [EDE 97] propose une approche par méta-programmation dans laquelle l'application d'un pattern est exprimée par un algorithme. [ROB 97] propose une démarche fort intéressante qui s'appuie sur des primitives de « refactoring ». Cette démarche a été notamment utilisée pour modifier un «design » en y introduisant le pattern Visiteur.

Malgré leur intérêt, ces propositions se concentrent toutes sur les solutions des patterns et négligent complètement les problèmes que ces patterns résolvent et à fortiori le lien entre ces problèmes et ces solutions. Or un design pattern est un couple <problème, solution> et la prise en compte du problème est littéralement essentielle (sans son problème un pattern perd son essence même). Ne serait-ce que pour sélectionner ou présélectionner des patterns en fonction des besoins des concepteurs, il paraît difficile de s'appuyer sur une description des patterns qui occulterait les problèmes qu'ils résolvent.

[BOR 99] propose une classification des outils de mise en œuvre des design patterns. En ce qui concerne le type de support [BOR 99] distingue la génération de code à partie de la sélection de paramètres, la génération de code à partir de l'instanciation d'une méta-description de la micro-architecture des patterns et enfin le support linguistique. Cette classification, qui nous paraît pertinente, illustre bien le manque total de support concernant la partie « problème » des design patterns.

Le question de savoir ce qui concerne plutôt les langages de programmation et sur ce qui doit plutôt être traité par des outils, s'est naturellement posé pour les design patterns [CHA 00]. Peu importe pour nous ce qui peut ou doit être exprimé dans un langage, car ce langage devra bien être compilé par un outil. L'utilisation des méta-variables que nous proposons pourrait sans doute être intégrée à un langage de programmation et les transformations que nous proposons à un compilateur. A cause de cet aspect méta toutefois, le concepteur peut-être amené à dire son mot sur les solutions produites puisqu'elles sont dans le langage qu'il manipule habituellement. Un tel « compilateur » devrait donc sans doute être plus interactif qu'un compilateur C++ par exemple.

A notre connaissance, [MAR 00a,b] est la seule autre proposition qui concerne cette partie « problème ». Leur démarche concerne en fait plutôt les patterns architecturaux (architectural patterns) et s'appuie sur la méthode B [ABR 96]. Le point commun de cette démarche et de la notre est de viser un véritable support sémantique pour les design patterns. Etant donné les efforts importants qu'une telle ambition requière nos deux équipes ont donc unis leur forces et se sont associées à la société SOFTEAM dans le projet RNTL LUTIN[2] qui a été labellisé en avril 2001 et a démarré en janvier 2002.

[2] http://www-poleia.lip6.fr/~ziane/lutin.html

Au delà même de l'outillage des design patterns, pour maîtriser, dans tous les sens du terme, un nombre sans cesse croissant de patterns il est nécessaire de comprendre les mécanismes sous-jacents qui les sous-tendent. Il s'agit d'un simple principe d'économie et de rationalité : remplacer une nombre sans cesse croissant de patterns par un nombre limité de mécanismes permettant d'engendrer leurs solutions selon le problème à résoudre et le contexte dans lequel il se manifeste.

Dans [ZIA 00] et [ZIA 01] nous avons proposé d'exprimer ces mécanismes par des règles de transformations. Nous avons aussi proposé d'exprimer les problèmes de conception (susceptibles d'être résolus par un ou plusieurs patterns) via l'introduction de méta-variables. Nous avons appliqué ces propositions au problème de la création virtuelle auquel s'attaquent notamment les patterns **Prototype** et **Factory Method**. Nous avons montré comment quelques transformations, principalement un pliage sur lequel nous revenons ici, permettent de retrouver les solutions de ces patterns et donc de produire le code correspondant.

Dans le présent document nous développons et généralisons ces résultats préliminaires. Nous montrons que d'autres sortes de patterns que les patterns créateurs sont accessibles à notre démarche. Nous l'illustrons sur un pattern structurel, **Bridge**, et un pattern comportemental, **Stratégie**. Pour cela nous raffinons la notion de méta-variable introduite dans [ZIA 00] et distinguons les méta-variables statiques des méta-variables dynamiques et de façon orthogonale les méta-variables qui dénotent des termes « comportementaux » (des instructions) de celles qui dénotent des définitions de données. La classique transformation appelée pliage (fold en anglais) est généralisée[3] pour introduire des définitions de données et non plus seulement des définitions de routines.

2. Exprimer le problème du concepteur

Un concepteur à objets qui fait face à un problème et qui souhaite de l'aide doit tout d'abord pouvoir exprimer son problème. Nous supposons toutefois que ce concepteur n'est pas près à changer radicalement ses habitudes pour embrasser une démarche lui demandant une spécification formelle complète de son application. La description de son problème doit donc être aussi simple que possible. Or il nous semble que les problèmes auxquels s'adressent les design patterns sont peut-être moins difficile à exprimer qu'on pourrait le croire.

En fait ces problèmes nous semblent souvent être l'application à des contextes différents de préoccupations relativement récurrentes qui consistent typiquement à garantir que certaines règles facilitant la maintenance des applications sont respectées.

Notre proposition consiste à utiliser des méta-variables c'est à dire des variables représentant un terme du langage de programmation lui même. Le concepteur utilisera une méta-variable à la place d'un **terme voulu** qu'il ne peut pas ou ne sait pas exprimer dans ce contexte.

---

[3] Nous ne prétendons pas que cette généralisation est nouvelle, sans doute pas tant elle est naturelle. La contribution consiste à monter en quoi elle peut décrire le mécanisme sous-jacent de certains patterns.

Plusieurs raisons possibles peuvent pousser le concepteur à ne pas exprimer directement le terme voulu :

- le terme voulu peut être illégal (violation d'encapsulation...),
- le terme voulu peut violer une règle de génie logiciel (introduire une dépendance non voulue...),
- le terme voulu peut dépendre d'une condition statique ou dynamique.

Le problème pour un outil d'assistance est de remplacer la méta-variable par un terme de substitution qui doit être équivalent[4] avec le terme voulu mais qui doit aussi être légal et acceptable dans son contexte. Par « acceptable » nous entendons « qui respecte les règles de qualité en vigueur » ces règles devant être explicitées pour éviter que l'outil inonde le concepteur de suggestions inutiles. Le minimum bien sûr est que l'outil respecte les règles du langage de programmation concernant notamment les droits d'accès ou la visibilité des identificateurs. Ces règles de qualité sont celles qui justifient de nombreux design patterns.

Par exemple la règle « programmez selon une interface par selon une implémentation » citée par [GAM 94] justifie en partie un pattern comme Prototype dans la mesure où on s'interdira de mentionner le nom d'une sous-classe d'une classe abstraite en dehors de la hiérarchie commençant à la classe abstraite (voire à la sous-classe elle-même). Ceci permet d'éviter une dépendance du code client de la classe abstraite envers l'ensemble de ses sous-classes.

Le lecteur peut nous rétorquer qu'un concepteur, comme nous l'avions nous même souligné, n'est pas forcément un expert en spécification et qu'il pourrait avoir du mal à exprimer ce genre de contrainte. En fait ceci ne devrait pas poser de problème en pratique pour plusieurs raisons :

- des règles par défaut peuvent fort bien (et doivent) être connues par l'outil d'assistance,
- les règles spécifiques à une équipe de développement n'ont pas à être modifiées fréquemment,
- seules les exceptions concernant telle ou telle classe ou entité de l'application devront être fournie au coup par coup.

Pour simplifier l'écriture des ces règles il est parfois possible de s'appuyer sur le langage de programmation lui même. Ainsi on peut imaginer une règle par défaut (pour C++) qui dise « tous les constructeurs des sous-classes d'une classe abstraite doivent être déclarés *protected* ».

Le concepteur n'aura donc pas à manipuler d'expression logiques complexes, même si une personne de l'équipe devrait pouvoir exprimer les règles spécifiques à l'équipe.

2.1 Exemple 1 : violation d'encapsulation

Le problème suivant est extrêmement simple et seul un débutant pourrait avoir besoin d'aide pour le résoudre. Supposons donc qu'un programmeur C++ débutant

[4] Sémantiquement équivalent. Etant donné que c'est le concepteur qui va assurer en partie cette équivalence il ne nous semble pas utile d'en donner une définition plus précise.

écrive le fragment de code suivant dans une méthode *methodeIndiscrete* de la classe *MaClasse* :

```
unePersonne.age
```

où *age* est un attribut privé de la classe *Personne* et où *unePersonne* est une variable de type *Personne*.

A moins qu'une permission spéciale (*friend* en C++) soit donnée à *methodeIndiscrete* pour accéder un champ privé de la classe *Personne*, le terme *unePersonne.age* est illégal dans *methodeIndiscrete*.

Si le programmeur débutant souhaite de l'aide pour résoudre son problème, il ou elle introduira une méta-variable pour masquer le terme voulu *unePersonne.age*. Ici comme le programmeur est capable d'exprimer le terme voulu (ce n'est pas toujours le cas) il est souhaitable de l'indiquer à l'outil d'assistance et de ne pas le remplacer simplement par une méta-variable. Il s'agira donc dans ce cas d'**encadrer le terme voulu** pour constituer une méta-variable anonyme. Peu importe bien sûr la syntaxe concrète et on pourrait aussi bien sélectionner le terme voulu à la souris.

```
[[unePersonne.age ]]
```

Un outil d'assistance peut alors essayer de remplacer la méta-variable par un terme correct équivalent mais légal et « acceptable ». Ceci sera explicité plus loin mais en bref il s'agit ici simplement de déplacer le terme voulu par un pliage qui le déplace dans une routine (fonction, méthode...) qui évalue elle même ce terme.

Ici la version actuelle de notre prototype en prolog[5] essaye naïvement une fonction mais se rend compte qu'elle ne peut pas avoir le droit d'évaluer ce terme non plus et propose donc une méthode de la classe *MaClasse*, ce qui revient bien sûr à introduire et utiliser une méthode d'accès. Le lecteur est invité à voir en annexe le terme prolog produit par notre prototype. Il serait bien sûr possible d'optimiser le raisonnement suivi par ce petit prototype mais ce n'est pas une priorité.

[PAR 90] distingue le cas où un pliage introduit une nouvelle routine de celui où il utilise une routine déjà existante. Malheureusement il ne nous paraît pas possible dans le cas général de reconnaître si parmi les routines déjà existante il en existe une qui pourrait convenir. Il nous faudrait pour cela analyser le code des routines candidates. Même si une tentative de reconnaissance pourrait marcher en pratique dans de nombreux cas nous nous limitons pour le moment à produire une nouvelle routine qui devra encore être acceptée en dernier ressort par le concepteur comme résolvant vraiment son problème convenablement.

Le but de cet exemple très simple est de montrer concrètement comment une (parmi d'autres sortes de) méta-variable peut-être introduite et quel mécanisme, ici un pliage classique, peut résoudre le problème sous contrôle de règles qui évitent les solutions illégales ou non acceptables.

Malgré la grande simplicité de cet exemple, il faut tout de même noter qu'un simple pliage résout un problème somme toute fondamental de la programmation et dont la solution est une des bases de la programmation à objets. Ce n'est qu'une

[5] Une version partielle contenant certains exemples de ce papier est disponible ici : http://www-poleia.lip6.fr/~ziane/gnome

demie surprise dans la mesure où le simple couple pliage/dépliage[6] permet des transformation d'un très grande généralité [DAR 75] [PAR 90].

2.2 Exemple 2 : choix de configuration

A nouveau nous prenons un exemple très simple pour illustrer une autre sorte de méta-variable et de pliage. Il s'agit d'utiliser le mécanisme de liaison dynamique pour effectuer une configuration dynamique d'un aspect de l'application. Cet exemple est trop simple pour constituer à lui seul un design pattern mais son principe revient très souvent dans les patterns. Cette technique est sans doute même utilisée un peu trop systématiquement dans les patterns, y compris pour effectuer dynamiquement des configurations qui pourraient être faites statiquement. Il est toutefois vrai qu'il est ensuite possible de spécialiser semi-automatiquement les programmes résultant [SCH 90].
Supposons qu'on veuille configurer une application en fonction de la langue des utilisateurs (anglais, français...), de façon assez naïve il est vrai puisqu'il existe bien sûr en fait des techniques sophistiquées d'internationalisation pour le faire. Simplifions encore notre problème et concentrons nous sur le fragment de code suivant :

```
print("hello")
```

Si un concepteur souhaite afficher un message qui dépend de la langue choisie par l'utilisateur de son application, plusieurs solutions sont possibles. La solution canonique en programmation à objets consiste à définir une hiérarchie de classes telle que celle de la *Figure 1* sur laquelle sera définie une fonction virtuelle (au sens de C++) qui retournera "hello" pour la classe *Anglais* et "salut" pour la classe *Francais*.

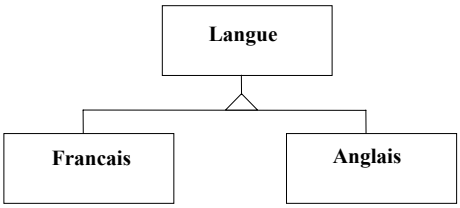

Figure 1. *Hiérarchie des langues*

Dans cet exemple le terme voulu n'est pas simplement "hello" sinon il n'y aurait sans doute pas de problème. Le terme voulu est la traduction de "hello" dans la langue de l'utilisateur. Nous proposons dans ce genre de cas où le terme voulu dépend d'un choix, c'est à dire lorsqu'il y a plusieurs possibilités, qu'une méta-

[6] Un dépliage est la tansformation inverse du pliage. Un compilateur C++ essaye ainsi de déplier les routines déclarées « inline », comme un pré-processeur « macro-expanse » les termes qui correspondent à des macro-définitions.

variable d'un genre particulier soit utilisée, que nous appellerons **méta-variable dynamique**.

Le nom de ce genre de méta-variable indique que la solution qui sera introduite sera à priori la solution canonique s'appuyant sur la liaison dynamique même si le choix pourrait être fait statiquement. Comme nous l'avons dit plus haut nous ne faisons ici que reproduire un biais présent dans les design pattern et rien n'empêcherait d'introduire d'autres transformations si ce n'est une complexité accrue. En ce qui concerne la notation que le concepteur doit utiliser pour introduire une méta-variable dynamique, là encore peu importe à ce stade de nos travaux[7] mais on peut proposer par exemple :

```
print(%%salutation("hello",salut"))
```

Il n'est pas indispensable ici non plus de donner un nom à la méta-variable (ici *salutation*) mais cela pourrait servir pour donner un nom parlant à la fonction virtuelle. Il n'est pas non plus indispensable de donner les valeurs possibles "hello" ou "salut" et dans ce cas seule la classe abstraite et la déclaration de la fonction virtuelle seront définies mais pas les sous classes.

La transformation qui permet d'introduire une fonction virtuelle à la place de la méta-variable dynamique *%%salutation* est ce que nous appellerons plus bas un **pliage dynamique** qui est un cas particulier du pliage classique.

Il faut toutefois signaler un problème que nous n'avons pas vu traité dans la littérature sur les transformations de programme, à savoir le choix du contexte (la classe par exemple) où placer la routine (ici la fonction virtuelle) introduite par le pliage. Ce choix est crucial comme nous l'avons vu dans notre premier exemple. S'il s'agit d'utiliser une classe déjà existante, des règles sur ce qui est légal ou acceptable peuvent limiter les choix. Mais il est aussi possible de créer une nouvelle classe pour recevoir une méthode, voire une nouvelle hiérarchie pour recevoir une fonction virtuelle.

Ce choix revient au concepteur car il nous paraît là aussi difficile de reconnaître dans le cas général si une hiérarchie existante peut convenir. Notons par exemple que la différence entre les patterns Prototype et Factory Method se résume selon nous à ce choix[8]. Techniquement, tant qu'elle a suffisamment de classes toute hiérarchie pourrait convenir mais bien évidemment il n'est pas question d'attacher de fonction virtuelle n'importe où et de polluer le code existant.

Le terme résultant de la transformation de la méta-variable %%salutation("hello",salut") est un appel à la fonction virtuelle introduite par le pliage. Il est toutefois nécessaire pour réaliser cet appel de disposer d'une variable, mettons *x*, dont le type dynamique correspond à la langue préalablement choisie. Si la fonction virtuelle est appelée *salutation* le terme résultant sera *x.salutation()* et le fragment de code devient donc :

---

[7] A l'heure actuelle toute le code existant doit être mimé (méta-décrit) en prolog à la main car aucun système de traduction n'a été écrit.

[8] Il est toujours possible d'arguer que la différence est plus subtile étant donné que ces patterns ne sont pas formalisés.

```
print(x.salutation())
```

avec bien sûr en sus la création de la fonction virtuelle salutation et éventuellement de la hiérarchie correspondante. Le lecteur est invité à consulter l'annexe 3 pour le détail de la session prolog correspondante dans notre prototype.
De façon générale l'introduction d'une variable (où comme toujours l'identification d'une variable déjà existante ce qui est plus complexe et demande l'aval du concepteur) est nécessaire pour permettre le choix de configuration, l'initialisation de cette variable étant laissée à la responsabilité du concepteur. En fait ce dernier peut ici déléguer le problème en déclarant plutôt *x* comme un paramètre de la méthode qui contient le fragment à réécrire ou comme un attribut de sa classe.

2.3 Différentes sortes de méta-variables

Il est temps de présenter de façon plus complète les différentes sortes de méta-variables que nous distinguons et de récapituler ce que nous avons dit sur l'expression du problème du concepteur.

Nous distinguons donc deux sortes de méta-variables. Les transformations correspondantes seront présentées plus loin. Les **méta-variables dynamiques** représentent un terme qui doit être choisi parmi un ensemble de possibilités. L'ensemble des termes possibles et le prédicat de choix peuvent ou non être connus par le concepteur au moment ou le contexte de (i.e. la méthode contenant) la méta-variable est compilé. Ainsi dans notre exemple sur le choix de la langue de l'application, le concepteur ne sait pas forcément quelles seront toutes les langues possibles ni comment précisément l'utilisateur choisira telle ou telle langue.

Les méta-variables dynamiques donnent lieu à l'introduction d'une fonction virtuelle et éventuellement d'une hiérarchie de classe où l'attacher. Comme nous l'avons souligné plus haut cette façon de faire est typique des design patterns même pour certains choix qui pourraient être fait statiquement. La transformation correspondant est un pliage dynamique (voir plus bas).

Les **méta-variables statiques** indiquent un terme voulu qui est unique. Comme il n'y a pas de choix à effecteur, le terme voulu sera simplement déplacé dans un autre contexte. La transformation correspondante est un pliage simple ou ce que nous appelons un pliage de données dans le cas ou le terme voulu est une définition de donnée (voir plus bas, notamment le pattern Bridge). Bien entendu dans le cas d'une méta-variable dynamique où l'ensemble des termes voulus contient des définitions de données la transformation correspondante sera un pliage dynamique de données.

Il est possible comme pour les méta-variables dynamiques que le terme voulu ne soit pas explicité. Par exemple ce terme voulu peut être un algorithme pour effectuer tel ou tel calcul. Le concepteur peut très bien savoir ce doit faire l'algorithme (par exemple calculer la date du lendemain) sans pour autant vouloir l'expliciter à ce moment précis. Dans ce cas ce qui est déplacé c'est la méta-variable elle-même qui devient alors une **méta-variable dégradée** qui doit être remplacée par le concepteur lui-même (ou un autre développeur) dans le nouveau contexte.

Nous allons à présent illustrer l'utilisation de ces méta-variables sur plusieurs sortes de design patterns. Est-ce que nous prétendons que ces quelques catégories de

méta-variables et les pliages associés suffisent à représenter tous les design patterns ? Très probablement pas, ne serait-ce que parce qu'il est même difficile de définir parmi les patterns ceux qui sont des « design » pattern de ceux qui concernent d'autres aspects.

En revanche, nous prévoyons que l'utilisation de ce genre de méta-variables, accompagnées des transformations correspondantes, sous contrôle de règles de qualité et des règles imposées par la syntaxe du langage cible, permettent en quelque sorte de redécouvrir les solutions d'un nombre important de design patterns.

3. Transformations et application à quelques patterns

La transformation appelée pliage [DAR 75] introduit une nouvelle définition de fonction ou utilise une définition existante et remplace un terme donné par un appel à cette fonction. Dans le cas de la programmation à objets, cette définition est bien sûr étendue aux méthodes et plus généralement aux différentes sortes de routines du langage de programmation visé.

pliage classique

Term $\rightarrow$ f (Args)
à condition que
f soit une routine définie par
f (FormalArgs) = Term <Args, FormalArgs>

où *Term <Args, FormalArgs>* est la substitution de *Args* par *FormalArgs* dans *Term*.

Cette transformation n'est pas si simple à implémenter dans la mesure où le choix des arguments Args n'est pas trivial. Comme nous l'avons déjà signalé, l'espace de nom où f est définie (notamment les classes des méthodes) est important car la possibilité d'accéder à certains identificateurs en dépend.

Enfin cette transformation doit être appliquée avec parcimonie, notamment lorsqu'une nouvelle définition de fonction est introduite, sous peine d'introduire des définitions arbitraires. A l'heure actuelle nous restreignons donc son application à la transformation de méta-variables (le genre de pliage dépendant du genre de méta-variable).

L'introduction d'une méta-variable par le concepteur est un demande d'aide parce qu'un terme, tout en représentant bien l'intention du concepteur, ne convient pas pour des raisons qu'on peut qualifier de techniques. Un pliage va donc déplacer ce terme (en l'adaptant si besoin est, par substitution des paramètres adéquats) vers une routine où il sera légal et où il respectera les règles de qualité en vigueur (et explicitées au préalable).

Pour traiter les méta-variables dynamiques, nous avons utilisons un cas particulier du pliage que nous appelons pliage dynamique dans lequel f est une fonction virtuelle.

pliage dynamique

Term → f ({Target} U Args)
à condition que
f soit une fonction virtuelle sur un type T définie par
f_S ([this] + FormalArgs)
= Term <[Target] + Args, [this] + FormalArgs>
pour tout S dans les sous-types de T

où f_S est la version de f pour le sous-type *S*, où [ ] délimite des listes et où + est la concaténation de listes. On voit apparaître ici un paramètre obligatoire *this* qui correspond à la cible de la liaison dynamique. La définition de la fonction virtuelle *f* est ici générique, c'est à dire la même pour chaque version de *f* ! Toutefois il faut considérer ici que le terme voulu *Term* contient implicitement une condition sur une donnée correspondant au type dynamique de la cible. Ainsi dans l'exemple 2 plus haut, le terme voulu complet serait *si langue= francais alors "salut" sinon ...*

Pour traiter certains patterns comme *Bridge* qui déplacent des définitions de données d'un classe à l'autre, et non pas seulement du code, nous proposons de généraliser la notion de pliage aux définitions de données. Au lieu d'introduire une abstraction fonctionnelle, un pliage de données introduit une classe.

pliage de données

class C1 (Data, Code)
→ class C1 (x : C2, NewCode)
à condition que
class C2 (Data, code(Code, Data) <C1,C2>)
NewCode = delegate [Code, Data, x]

Cette transformation remplace un ensemble de définitions de données *Data* dans une classe *C1* par un simple attribut *x* de type *C2*. La classe *C2* a pour données *Data* et pour code le code de *C1* qui concerne *Data*, noté *code(Code,Data)* modifié pour remplacer *C1* par *C2*.

Les méthodes de la classe *C1* doivent être modifiées pour déléguer le traitement concernant *Data* vers une méthode de *C2*. Pour chaque méthode de *C1* mentionnant une donnée de *Data* cette délégation est un pliage classique de tout son code vers une méthode de *C2*, où *Args* = [*x*].

Un **pliage dynamique de données** est un cas particulier dans lequel C2 est une classe abstraite ou une interface dont les méthodes sont des fonctions virtuelles. Dans ce cas les délégations sont bien sûr des pliages dynamiques.

Les patterns Prototype et Factory Method

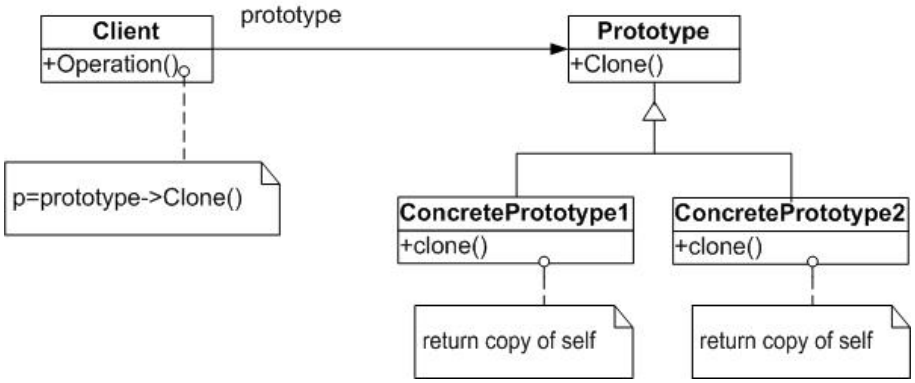

***Figure 2**. Structure du pattern Prototype*

Ces deux patterns [GAM 94] visent à résoudre le problème de la création virtuelle dans lequel de nouvelles instances d'une classe abstraite (ou d'un interface) P doivent être créées alors que les sous-classes (ou les implémentations) de P sont inconnues et n'existent d'ailleurs peut-être pas encore. Il est possible aussi que ces sous-classes soient connues mais qu'on ne veuille pas les mentionner explicitement pour ne pas introduire de dépendance potentiellement nuisible à la maintenance de l'application (ajout ou suppression d'une sous-classe).

Comme nous l'expliquions dans [ZIA 01], ce problème de création virtuelle peut-être exprimé par l'introduction d'une méta-variable dynamique puisqu'il est possible, voire probable, qu'il y ait plusieurs sous-classes parmi lesquelles un choix devra être fait. Le concepteur peut donc écrire quelque chose de la forme suivante :

```
P* p = new %%C();
```

où *p* est un pointeur sur *P* et *%%C* une méta-variable dynamique.

Comme le montre la**,** la solution introduite par le pattern **Prototype** consiste à utiliser une instance de la classe voulue (un prototype), créée au préalable, et à la cloner. Sur la, la classe *P* est appelée *Prototype*. Un pliage dynamique de *new %%C()* reproduit cette solution, la nouvelle fonction virtuelle étant *clone*.

Nous devons toutefois préciser ici qu'un certain nombre de difficultés rendent l'application de ce pliage moins facile qu'on pourrait le croire. Tout d'abord, il n'est pas possible de considérer que la méta-variable dynamique est exactement le terme *%%C* car cela conduirait à remplacer ce être par l'invocation d'une méthode virtuelle ce qui n'a pas de sens. Il faut donc que l'outil d'assistance vérifie si la méta-variable est légale telle qu'elle est déclarée ou si elle doit inclure un terme plus englobant. Dans notre exemple le premier terme englobant qui peut légalement (en C++) être remplacé par une invocation de routine est *new %%C()*. C'est donc ce terme qui sera remplacé. Nous n'avons pas implémenté ce genre de déduction pour le moment

puisqu'elle se situe en amont de la représentation en prolog du code de l'application et que nous nous sommes concentrés sur le moteur de réécriture en prolog.

Une autre difficulté, que nous avons déjà discutée, concerne le choix de la hiérarchie où attacher la fonction virtuelle qui est introduite, par la transformation d'une méta-variable dynamique. Il faut d'ailleurs signaler que le pattern **Factory Method** est une variante du pattern Prototype où la fonction virtuelle est attachée à une autre hiérarchie, parallèle à celle dont la racine est la classe *P* (ou Prototype sur la). C'est au concepteur de choisir où attacher la fonction virtuelle même si des heuristiques raisonnables pourraient lui suggérer de considérer Prototype en priorité.

Concrètement le code résultant a la forme suivante :

```
P* x; // initialisez avec le choix approprié
P* p = x.vf();
```

avec par ailleurs la définition de la fonction virtuelle *vf* et éventuellement d'une hiérarchie correspondante. Une contrainte qui sert de commentaire indique la forme du code que doit avoir une version de *vf* dans une sous-classe de la hiérarchie. Si des sous-classes existent déjà, le code de la version de vf lui correspondant est généré.

Le pattern Bridge

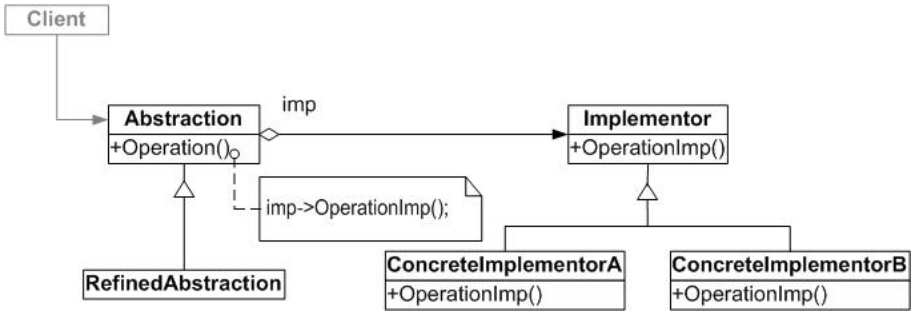

Figure 3**. Structure du pattern Bridge**

Le problème que ce pattern structurel [GAM 94] vise à résoudre consiste à découpler une abstraction (une interface) de son implémentation. Plus précisément, ce pattern concerne plusieurs problèmes similaires dont le besoin de cacher une implémentation, de ne pas recompiler le code client quand une implémentation change et la possibilité de sélectionner ou de changer une implémentation pendant l'exécution.

Nous proposons d'exprimer ces différents besoins par une méta-variable qui sera dynamique s'il s'agit de changer ou de sélectionner l'implémentation pendant l'exécution et statique sinon. Le lecteur notera que nous n'exprimons que le minimum absolu pour pouvoir proposer une solution, faisant abstraction d'un certain nombre de différences dans les motivations du concepteur, qui conduiraient de toute façon à la même solution.

La méta-variable que le concepteur doit introduire, encadre une ou plusieurs définitions de données dans une classe Abstraction ou les remplace si ces définitions

n'ont pas encore été écrites. A nouveau, peu importe la syntaxe concrète de l'introduction de ces variables, qu'elle se fasse de façon textuelle ou par sélection à la souris mais des exemples pourraient être :

```
class Abstraction
{ private : [[int x,y;]] public : ... };
```

pour une méta-variable statique encadrant la définition des attributs x et y, ou

```
class Abstraction
{ private : %%X public : ... };
```

pour une méta-variable dynamique sans définition de données préalable.

Suivant le genre de méta-variable, un pliage de données statique ou (plus souvent) dynamique sera appliqué. Ces pliages comme nous l'avons vu introduisent une classe ou une hiérarchie correspondant à *Implementor* dans la Figure 3. Par ailleurs, la méta-variable, y compris les définitions éventuelles qu'elle encadre, sont remplacées par un seul attribut (*imp* dans la Figure 3) qui est un pointeur sur la classe *Implementor*. Le pliage de données applique aussi un pliage (fonctionnel) aux méthodes de la classe *Abstraction* (du moins celles qui mentionnent les définitions encadrées s'il y en a) de façon à déléguer les traitements à l'objet *imp*.

Le pattern Stratégie

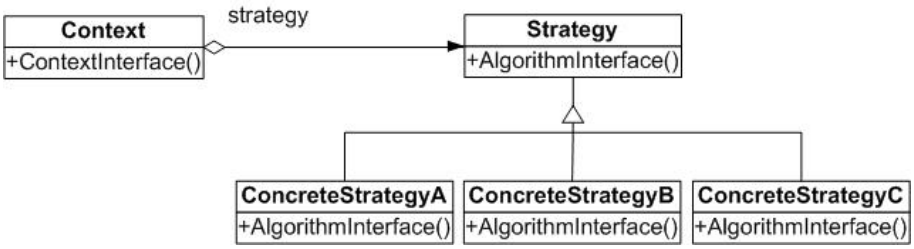

Figure 4. Structure du pattern Stratégie

Le but du pattern Stratégie est de "définir une famille d'algorithmes, d'encapsuler chacun et de les rendre interchangeables. Stratégie permet de faire varier l'algorithme indépendamment des clients qui l'utilisent" [GAM 94].

La section *Applicability* du pattern précise cet objectif général. Un objectif possible est de ne pas dévoiler des structures de données complexes dépendant d'un algorithme spécifique. Bien que ce ne soit pas mentionné dans [GAM 94] cet objectif semble similaire à celui du pattern Bridge.

Nous proposons donc que le concepteur qui fait face à ce problème introduise une méta-variable dynamique pour encadrer les données qui doivent être cachées. Un pliage de données va alors déplacer ces données vers une nouvelle hiérarchie (la classe *Strategy* sur la Figure 5) et déléguer le code des méthodes qui mentionnent ces données.

Un autre objectif possible est de réorganiser une série de test conditionnels (conditional statements). Ce code peut être encadré par une méta-variable dynamique pour application d'un pliage (fonctionnel) dynamique. Bien que nous ne l'ayons pas encore implémenté nous pensons que la hiérarchie de classes correspondante peut être définie automatiquement dans certains cas. Dans les autres cas le concepteur devra effectuer un certain travail d'adaptation et définir lui même les sous classes de *Strategy*. Il aura toutefois été mis sur la voie de la solution.

Si le code conditionnel qui doit être réorganisé utilise des attributs de la classe *Context*, leur définition devrait au préalable être encadrée par une méta-variable dynamique. Sinon ils resteraient dans cette classe et des passages de paramètres inutiles vers les méthodes de *Strategy* seraient générés.

5. Conclusion

Dans ce document nous avons généralisé notre proposition de [ZIA 01] de façon à ce qu'elle s'applique aussi aux patterns structurels et comportementaux et non plus seulement aux patterns créateurs. Notre objectif reste d'améliorer le "support" des design pattern par des outils et notre démarche s'appuie toujours sur l'application de transformations de programmes.

Ces nouvelles sortes de patterns nous on conduit à distinguer plusieurs sortes de méta-variables et à généraliser la classique transformation de pliage pour introduire des abstractions de données et non plus seulement fonctionnelles.

Notre prototype prolog[9] a été amélioré et en fait réécrit pour plus de généralité mais il est encore loin de pouvoir être intégré dans un environnement de développement. C'est la raison pour laquelle nous avons joint nos forces avec celles de [MAR 00a,b] et de la société SOFTEAM pour proposer le projet RNTL LUTIN[10] qui a démarré en janvier 2002.

Bibliographie

[ABR 96] Abrial J.R., *The B Book - Assigning Programs to Meanings* Cambridge University Press, August 1996.

[AGE 98] AGERBO E., CORNIS A., "How to preserve the benefits of Design Patterns", OOPSLA 1998.

[BOR 99] BORNE I., REVAULT N., "Comparaison d'outils de mise en oeuvre de design patterns", revue l'Objet, éditions Hermes, Volume 5, numéro 2, 1999.

[BOS 96] BOSCH J., "Language Support for Design Patterns", TOOLS Europe '96.

[BUD 96] BUDINSKY F. J., FINNIE M. A., VLISSIDES J. M., YU P. S., "Automatic code generation from design patterns", IBM Systems Journal, Vol. 35, No 2, 1996.

[9] Une version partielle contenant certains exemples de ce papier est disponible ici : http://www-poleia.lip6.fr/~ziane/gnome

[10] http://www-poleia.lip6.fr/~ziane/lutin.html

[CHA 00] CHAMBERS C., HARRISSON B., VLISSIDES J., "A Debate on Language and Tool Support for Design Patterns", ACM POPL 2000, pp. 277-289.

[COP 92] COPLIEN J.O., *Advanced C++ - programming styles and idioms*, Addison Wesley 1992.

[COP 94] COPLIEN J.O, "Software Design Patterns: Common Questions and Answers", Proceedings of Object Expo New York, SIGS Publications, pp 39-42, 1994. Also appears in [RIS 98] pp 311-320 and available at http://hillside.net/patterns/papers/

[DAR 75] DARLINGTON J., "Applications of program transformation to program synthesis", *International Symposium on Proving and Improving Programs*, Arc-et-Senans, France, 1975.

[EDE 97] A. H. EDEN, J. GIL, A. YEHUDAI, "Precise Specification and Automatic Application of Design Patterns", Proceedings of the 12th IEEE International Automated Software Engineering Conference, ASE '97, Lake Tahoe, Nevada, November 1997, pp. 143-152. Los Alamos: IEEE Computer Society Press.

[EDE 99a] EDEN A. H., Y. HIRSHFELD, K. LUNDQVIST. "LePUS – Symbolic Logic Modeling of Object Oriented Architectures: A Case Study". *Second Nordic Workshop on Software Architecture – NOSA* 1999.

[EDE 99b] EDEN A. H., Precise Specification of Design Patterns and Tool Support in their Application, Ph.D. dissertation, available at http://www.math.tau.ac.il/~eden/

[GAM 94] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns: Elements of Reusable Object-oriented Software*, Addison Wesley, Reading, 1994.

[LAU 98] LAUDER A., KENT S., "Precise visual specification of design patterns", ECOOP 1998.

[MAR 00a] R.MARCANO-KAMENOFF, N. LEVY, F. LOSAVIO, " Specification formelle de patterns d'architectures distribuees en UML et B ", *Langages et Modèles à Objets*, LMO 2000.

[MAR 00b] MARCANO-KAMENOFF R., "Formalizing Pattern Applicability - An Approach based on UML and B", Doctorial Symposium of the IEEE International Conference on Automated Software Engineering, ASE 2000.

[PAR 90] PARTSCH H. A., *Specification and Transformation of Programs*, Springer-Verlag 1990.

[RIS 00] RISING L., *The Pattern Almanac 2000*, Addisson Wesley 2000.

[ROB 97] ROBERTS D., BRANT J., JOHNSON R., "A Refactoring Tool for Smalltalk", TAPOS 3(4): 253-263, 1997.

[SCH 00] Schultz U.P., Lawall J.L., Consel C., "Specialization Patterns", IEEE int. conf. on Automated Software Engineering, ASE 2000.

[ZIA 00] ZIANE M., "A Transformational Viewpoint on Design Patterns*", IEEE int. conf. on Automated Software Engineering*, ASE 2000.

Annexe 1 Représentation en prolog

Certains exemples de ce papier peuvent être essayés sur notre prototype à cette adresse : http://www-poleia.lip6.fr/~ziane/gnome

Voici un extrait de la représentation prolog du code correspondant aux exemples 1 et 2 du papier. Le format général de représentation de la définition d'une entité (classe, routine, variable...) est le suivant :

```
entity(référence, contexte, nom, représentation, niveau d'accès).
```

La référence est un nom interne unique de l'entité, le contexte est l'espace de noms où est définie l'entité (classe, fichier...), le nom est celui de l'entité dans le programme, la représentation varie en fonction de la sorte d'entité, et le niveau d'accès est public, protected, private...

```
% Class Myclass
entity(myClassFile, none, myClassFile, fileScope, none).
import(myClassFile, personFile, public,_).
entity(myClass, myClassFile, 'MyClass', class, public).
entity(politeMethod, myClass, politeMethod, method(void,[]), public).
entity(indiscreteMethod, myClass, indiscreteMethod, method(void,[]),
public).
entity(var(aPerson),indiscreteMethod , aPerson, variable('Person'),
private).
```

```
% class Person
entity(personFile, none, personFile, fileScope, none).
entity(personClass, personFile, 'Person', class, public).
entity(ageOfPerson, personClass, age, attribute(int), private).
```

Les termes qui vont être manipulés par les règles de transformation sont ce que nous appelons des **tic** (Terms In Context). Un tic a la forme suivante

```
tic(contexte, représentation, contraintes, environnement)
```

Le contexte est la routine ou le terme apparaît, la représentation dépend du terme, les contraintes sont des expressions logiques qui sont accumulées pendant le processus de réécriture dont certaines peuvent être évaluées par l'outil et l'environnement est l'ensemble des entités qui sont introduites au fur et à mesure de la réécriture. Les contraintes qui ne peuvent pas être évaluées par l'outil servent de directives pour la génération de code ou en dernier recours pour le concepteur.

Annexe 2 : Exécution en prolog de l'exemple 1

Le traitement de l'exemple 1 du papier conduit à l'exécution suivante (à l'indentation près) :

```
?- tic2(T), transformTic(T,X).

T = tic(
  indiscreteMethod,
  metavariable(dot(aPerson, age)),
  [],
  []   )

X = tic(
  indiscreteMethod,
  invoke(groutine4, int, [aPerson]),
  [computes(groutine4, [gArg8], dot(gArg8, age))
   ],
  [entity(groutine4, personClass, groutine4, method(int, []), public),
   entity(gArg8, groutine4, gArg8, argument('Person'), private)
   ]   )
```

Ainsi le terme *[[aPerson.age]]* est transformé en l'appel *aPerson.groutine4()* où *groutine4* est une nouvelle méthode publique de la classe *Person* (*PersonClass* est le nom interne). Une contrainte précise que cette méthode évalue et renvoie *gArg8.age* ou *gArg8* est le seul paramètre formel de la méthode (donc ici correspond au paramètre implicite *this* en C++). On voit apparaître *groutine4* et *gArg8* parmi les entités déclarées dans l'environnement avec leur type et leur niveau d'accès (par convention un paramètre formel d'une routine est privé).

Annexe 3 : Exécution en prolog de l'exemple 2

```
tic3(T), transformTic(T,X).

T = tic(
  politeMethod,
  dmv(string, ["hello", "salut"]),
  [],
  []
       )

X = tic(
  politeMethod,
  invoke(gVF2, string, [gVar2]),
  [computes(gVFV4, [], "hello"),
   computes(gVFV3, [], "salut")
  ],
 [entity(gVar2, politeMethod, gVar2, variable(pointer(gClass4)),none),
 entity(gVFV4, gClass6, gVF2, vfVersion(string, []), public),
 subtypeDecl(gClass6, gClass4),
 entity(gClass6, gFile2, gClass6, class, none),
```

```
 entity(gVFV3, gClass5, gVF2, vfVersion(string, []), public),
 subtypeDecl(gClass5, gClass4),
 entity(gClass5, gFile2, gClass5, class, none),
 importDecl(politeMethod, gFile2, public),
 entity(gVF2, gClass4, gVF2, virtualFunction(string, []), public),
 entity(gClass4, gFile2, gClass4, class, none),
 entity(gFile2, none, gFile2, fileScope, none)
 ]
)
```

Le terme à transformer est une méta-variable dynamique (ici anonyme) *dmv(string,"hello","salut")*. Le résultat est l'appel *gVar2.gVF2()* où *gVar2* est une nouvelle variable qui est introduite pour contenir l'information représentant la langue courante.

De façon plus générale une telle variable est nécessaire pour permettre le choix de configuration, l'initialisation de cette variable étant laissée à la responsabilité du concepteur. En fait ce dernier peut ici simplement se défausser du problème en déclarant plutôt *gVar2* comme un paramètre de *politeMethod* ou comme un attribut de *MaClasse*.

Notez toutes les entités qui sont définies : *gClass4* la classe abstraite et ses sous classes *gClass5* et *gClass6*, *gVF2* la fonction virtuelle, et ses versions *gVFV3* et *gVFV4* ainsi que le fichier *gFile2* où définir ces classes. Notez que *GVFV3* et *GVFV4* évaluent bien "hello" et "salut". Enfin, les déclaration de sous-typage sont incluses ainsi qu'une nécessaire déclaration d'importation des entités du fichier *gFile2* par *politeMethod*. En pratique cette dernière déclaration ne pourra pas être traduite directement en C++ et un certain nombre d'inférences sont nécessaire pour déduire à quel niveau *gFile2* doit être importé.