Simplification de la production de logiciel par la programmation par aspects

Noury Bouraqadi
bouraqadi@ensm-douai.fr
http ://csl.ensm-douai.fr/noury

Dépt. G.I.P
Ecole des Mines de Douai
941 rue Charles Bourseul - B.P. 838
59508 Douai Cedex - France

Rapport Technique 2003-3-1
Ecole des Mines de Douai
14 Mars 2003

Table des matières

1 Introduction

Comme dans toute industrie, l'informatique n'est pas épargnée par le besoin de développer de plus en plus vite, des produits de plus en plus complexes [Sec00]. En effet, les utilisateurs, dans leur recherche de solutions globales, demandent de plus en plus de fonctions. Cette demande résulte notamment du développement des moyens de télécommunication et d'un matériel toujours plus puissant et financièrement plus abordable.

Face à ce besoin de productivité, les modèles de programmation actuellement utilisés ne sont pas toujours satisfaisants. En particulier, il est difficile de capitaliser les développements passés et de produire de nouveaux logiciels par simple extension/adaptation (réutilisation) de l'existant.

L'objectif des travaux que nous menons à Douai est de répondre à ce besoin de réutilisation en particulier et de contribuer plus généralement à la simplification de la production de logiciel. Notre thème de recherche s'inscrit dans la lignée des travaux amorcés notamment par le consortium OMG (Object Management Group) et le Xerox PARC (Xerox Palo Alto Research Center). Une partie de nos travaux porte sur les problèmes soulevés par les concepts de *composants logiciels* [Szy98][OMG99] et de *programmation par aspects* [KLM+97][EFB01][BL01].

Dans ce papier, nous nous focalisons sur la programmation par aspects. Ce paradigme de programmation vise à séparer les "fonctionnalités" des applications et les propriétés d'infrastructure (i.e. propriétés non-fonctionnelles) telles que la synchronisation, la persistance ou la distribution. Cette séparation introduit une nouvelle dimension dans la structuration des logiciels, simplifiant le processus de développement. En effet, le code métier et les propriétés d'infrastructure se trouvent définis dans des modules découplés les uns des autres. Le code résultant est alors plus simple à écrire et à comprendre en comparaison avec les approches plus conventionnelles et notamment l'approche objet. En effet, un module décrit totalement et exclusivement une unique facette du logiciel. Et la modification d'un module n'affecte pas les autres modules. D'où un gain de productivité dans les différentes étapes du cycle de vie du logiciel (développement, test, maintenance...).

Dans ce qui suit, nous commençons par motiver la programmation par aspects. Puis, nous présentons ce paradigme de programmation et son utilisation pour la construction de logiciels. Nous décrivons alors les différentes possibilités de réutilisation qui en découlent. Enfin, nous terminons par un bref aperçu des travaux de recherche que nous menons à Douai.

2 Pourquoi la programmation par aspects ?

Pour définir et motiver la programmation par aspects, nous nous appuyons sur l'exemple d'une librairie électronique représentée sur la figure 1. Les clients utilisent un réseau pour consulter la liste des ouvrages disponibles et éventuellement les commander. Plusieurs clients peuvent être connectés en même temps et plusieurs commandes peuvent être traitées en parallèle.

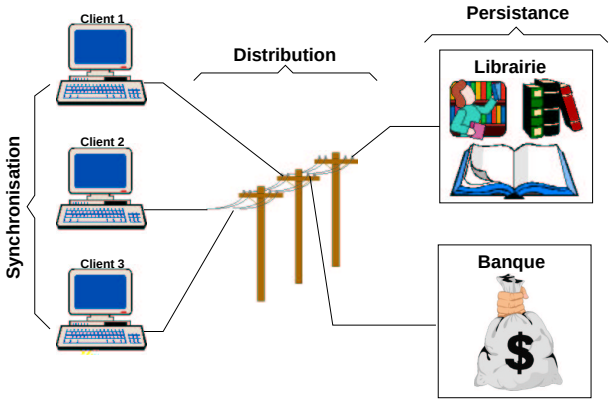

FIG. 1 – Exemple d'une application avec des propriétés d'infrastructure

La conception objet de cette application nous conduit à définir notamment les classes Librairie, Client, Ouvrage, Commande et Banque. Par ailleurs, nous pouvons identifier au moins trois propriétés d'infrastructure de cette application : la distribution, la persistance et la synchronisation. La distribution apparaît du fait que les instances des différentes classes en jeu se trouvent sur des sites distants. Cette propriété regroupe entre autres la gestion des communications distantes et la définition du protocole de communication utilisé. La persistance apparaît parce que des objets comme, par exemple, les ouvrages mis en vente, doivent survivre à un arrêt de l'application. La sauvegarde de tels objets sur un support de masse et la gestion des mises à jour de cette sauvegarde font partie de la définition de cette propriété de persistance. Quant à la synchronisation, elle englobe la gestion des accès concurrents aux structures des objets. C'est le cas, par exemple, lorsqu'un client demande le prix d'un ouvrage pendant que le libraire modifie ce même prix.

L'implémentation de l'application de la librairie électronique dans un langage à objets conventionnel tel que C++, Smalltalk ou Java introduit deux problèmes qui compromettent la réutilisation [HL95] [KLM+97] [EFB01] [BL01]. D'une part, l'approche objet conduit au *mélange du code* source métier de l'application avec celui correspondant aux propriétés d'infrastructure que sont la distribution, la synchronisation et la persistance. Et d'autre part, elle conduit à la *dispersion du code* correspondant à chacune de ces trois propriétés.

2.1 Mélange du code

Nous illustrons le mélange du code par une implémentation possible de la classe Ouvrage. La figure 2 en donne le code correspondant en Java. Le code métier apparaît en police normal, tandis que la distribution, la persistance et la synchronisation apparaissent respectivement en souligné, gras et italique. Comme le montre la figure 2, les lignes de code représentant le code métier ou chacune des propriétés d'infrastructure sont intimement enchevêtrées. De tels enchevêtrements nuisent à la lisibilité du code rendant difficiles sa compréhension, sa maintenance et sa réutilisation. Par exemple, il est difficile de réutiliser la classe Ouvrage dans un contexte utilisant un autre protocole de communication distante ou une autre politique de synchronisation.

La solution polymorphique, qui consiste à utiliser l'héritage pour remplacer une ou plusieurs propriétés d'infrastructure n'est pas satisfaisante.

- D'une part, la définition d'une sous-classe peut s'avérer impossible. C'est le cas dans les langages qui ne supportent que l'héritage simple, lorsque la politique de communication impose une super-classe. Par exemple, Java RMI impose que la classe Ouvrage hérite de java.rmi.UnicastRemoteObject.
- D'autre part, il faut remplacer certaines références à la classe originale par des références la sous-classe. Chose qui n'est pas toujours aisée voir impossible en l'absence de la totalité du code source de l'application.
- Enfin, la définition d'une sous-classe pour remplacer une propriété d'infrastructure intégrée à la super-classe peut conduire au problème *d'anomalie d'héritage* [MY93]. Ce problème se traduit par l'obligation de redéfinir dans les sous-classes plusieurs, voire la totalité, des méthodes de la super-classe, pour introduire la propriété d'infrastructure souhaitée. Le code métier des méthodes de la sous-classe est une copie à l'identique de celui défini dans la super-classe. La réutilisation par héritage est clairement compromise.

En résumé, avec la programmation par objets, la définition des fonctionnalités réalisés par l'application ne peut être facilement réutilisée avec une autre infrastructure. Inversement, il est difficile de réutiliser une propriété d'infrastructure avec un autre code métier.

2.2 Dispersion du code

Outre le problème de mélange de code, la réutilisation d'une propriété d'infrastructure est d'autant plus difficile que sa définition se trouve dispersée. De ce fait, elle ne peut être isolée pour être réutilisée. En effet, une même propriété d'infrastructure peut toucher plusieurs objets. Par exemple, les ouvrages et les clients de notre librairie électronique doivent être tous persistants. Or, avec l'approche objet, le code définissant la persistance se trouve dispersé et partiellement dupliqué

```
public interface OuvrageInterface extends Remote{
        oat getPrix() throws RemoteException;
       void setPrix( oat nouveauPrix) throws RemoteException;
       int getStock() throws RemoteException;
       void setStock(int nouveauStock) throws RemoteException;
       String getTitre() throws RemoteException;}

public class Ouvrage extends UnicastRemoteObject implements OuvrageInterface, Serializable{
        oat prix;
       int stock = 0; // Nombre d'exemplaires disponibles en stock
       String titre;

       public Ouvrage(String unTitre,  oat unPrix) throws RemoteException{
              super();
              synchronized(this) { titre = unTitre; }
              this.setPrix(unPrix);}

       public synchronized  oat getPrix() throws RemoteException {
              return prix;}

       public synchronized void setPrix( oat nouveauPrix) throws RemoteException {
              prix = nouveauPrix;
              DataBase.update(this);}

       public synchronized int getStock() throws RemoteException {
              return stock;}

       public synchronized void setStock(int nouveauStock) throws RemoteException {
              stock = nouveauStock;
              DataBase.update(this);}

       public synchronized String getTitre() throws RemoteException {
              return titre;}
```

FIG. 2 – Exemple de mélange du code métier avec les propriétés d'infrastructure

dans les classes Ouvrage et Client. Il est donc difficile de modifier ou de réutiliser ce code bien que la persistance de l'application soit conceptuellement considérée comme une unique propriété.

3 Qu'est-ce que la programmation par aspects ?

Une solution aux problèmes de mélange et de dispersion du code des propriétés d'infrastructure rencontrés avec la programmation par objets, consiste à séparer et découpler leurs définitions comme le veut le principe de *separation of concerns* (littéralement séparation des préoccupations) [Dij76] [HL95]. Partant de ce principe, l'équipe dirigée par Gregor Kiczales a formalisé cette séparation et les différentes étapes de réalisation des applications dans ce cadre pour aboutir au paradigme de programmation appelé *programmation par aspects* (*AOP : Aspect Oriented Programming*) [KLM+97]. Ce paradigme de programmation consiste à structurer les applications en modules indépendants qui représentent les définitions de différents aspects.

Il est important de souligner que la programmation par aspects ne vient pas remplacer la programmation par objets. L'approche à base d'aspects permet d'utiliser la technologie objet dans un cadre qui permet d'en conserver les bénéfices tout en palliant les limitations identifiées plus haut.

3.1 Structure d'une application à base d'aspects

L'utilisation de la programmation par aspect conduit à la décomposition d'une application en différents modules :

- **un noyau** : définit le cœur fonctionnel de l'application. Il s'agit des "fonction-alités" de l'application (le "Quoi" de l'application) qui peuvent être décrits sous forme de classes réutilisables. Par exemple, le noyau de la librairie électronique présentée précédemment inclut les fonctionnalités "rechercher un ouvrage", "passer une commande".
- **plusieurs aspects** : définissent les propriétés transversales au noyau. Il s'agit de propriétés le plus généralement d'infrastructure (le "Comment" de l'application) qui affectent le fonctionnement de plusieurs classes du noyau. Par exemple, la librairie électronique comporte les aspects distribution, persistance et synchronisation. Chaque de ces aspects est indépendant des autres et représente une propriété d'infrastructure particulière.

La figure 3 donne le noyau et les aspects de notre exemple d'application de librairie électronique. Une partie du code du noyau est détaillée pour montrer que les propriétés d'infrastructure ont bien été séparées du code métier. Ainsi, ce code métier est indépendant de toute infrastructure et peut être réutilisé dans différents contextes nécessitant des propriétés infrastructure différentes.

3.2 Intégration des aspects

La construction d'une application à partir des différents modules définis précédemment (voir 3.1) nécessite une étape **d'intégration**. En effet, ces modules sont définis séparément les uns des autres, et doivent être assemblés afin de "construire" l'application.

L'intégration du noyau avec les aspects se traduit par l'établissement d'une jonction. Cette jonction se situe au niveau d'un ensemble de points du flot d'exécution du noyau, appelés **points de jonction** [KLM+97]. Tout point de ce flot d'exécution constitue un point de jonction potentiel, susceptible d'être utilisé pour l'intégration. Ainsi, une invocation de méthode, une boucle ou une affectation sont autant de points de jonction potentiels. Notons que la variété de points de jonction potentiels qui peuvent être désignés dans la configuration dépend fortement de la mise en œuvre de la programmation par aspects.

Pour illustrer le concept de point de jonction, considérons l'aspect persistance de notre exemple de la librairie électronique. Cet aspect se traduit par la mise en place d'une base de donnée qui stocke notamment les ouvrages, les clients et les commandes. Les points de jonctions utilisés par cet aspect correspondent à la création d'instances de ces classes et aux modifications de leurs champs. Par exemple, la définition de l'aspect persistance doit stipuler que chaque fois qu'une

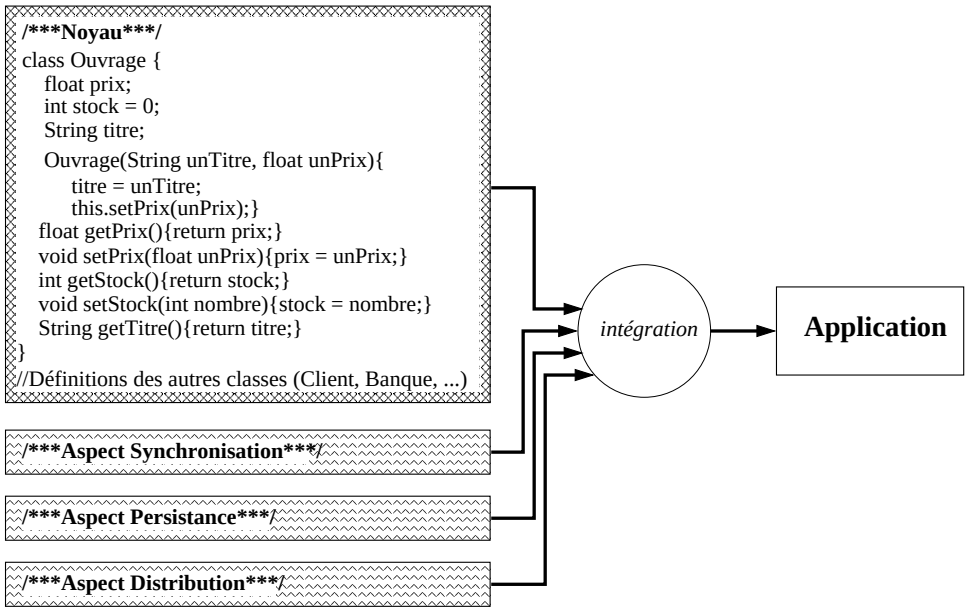

FIG. 3 – Construction d'une application avec la programmation par aspects

instance d'une des classes ouvrage, client ou commande est créée, la nouvelle instance doit être ajoutée à la base de données. Ainsi, l'objet résultat d'un "new Ouvrage()", doit être ajouté à la base de donnée. De plus, l'aspect persistance doit également indiquer que la base de données doit être mise à jour chaque fois qu'une instance de l'une des trois classes ouvrage, client et commande est modifiée. C'est le cas lors des accès en écriture aux champs prix et stock de la classe Ouvrage. En résumé, l'aspect persistance s'appuie sur les points de jonction que représentent les créations d'instances des classes ouvrage, client et commande, ainsi que les accès aux champs définis par ces trois classes.

L'assemblage d'un aspect avec le noyau peut utiliser plusieurs points de jonctions. Inversement, un même point de jonction peut servir pour l'assemblage de différents aspects avec le noyau. Ce partage de points de jonction nécessite de disposer de mécanismes de composition qui permettent une "superposition" harmonieuse des différents aspects. Il s'agit de modifier le flot d'exécution du noyau de manière à prendre en compte tous les aspects tout en préservant leurs sémantiques respectives.

4 Réutilisation des aspects

Nous avons montré dans la partie précédente que la programmation par aspects permet la production d'un code métier "pur" découplé du code non-fonctionnel. Il en résulte que le code métier ainsi obtenu est réutilisable avec différentes infrastructures (par exemple différents protocoles de communication distante). Le changement d'infrastructure implique uniquement le changement des aspects.

Dans cette partie, nous nous intéressons à la problématique complémentaire que représente la réutilisation des aspects. Il s'agit ici de produire des aspects génériques qui sont découplés de toute application. De tels aspects pourraient être intégrés dans différentes applications. Par exemple, un aspect communication distante basé sur CORBA [GGM99], s'il est générique, devrait pouvoir s'intégrer aussi bien dans une application de commerce électronique que dans un système d'agendas partagés.

Pour arriver à cette généricité, nous proposons d'externaliser les liens entre les aspects et le noyau d'une application. Cela se traduit par l'introduction d'un module supplémentaire, que nous

appelons **configuration**. Il s'agit d'un module qui définit les points de jonction entre les aspects et le noyau. Ainsi, comme le montre la figure 4, une application serait composée de trois types de modules. Il y a d'une part les modules réutilisables que sont le noyau et les aspects génériques. Et d'autre part, nous disposons de la configuration qui est spécifique à une application, puisqu'elle définit la manière d'intégrer les aspects et le noyau.

Afin d'illustrer la réutilisation d'aspects, reprenons l'exemple de l'aspect persistance de notre exemple de librairie électronique. Pour que l'aspect soit générique, il faut que sa définition se limite à définir la base de données à utiliser et les outils pour la manipuler. L'identification des objets à ajouter à la base et le moment de mettre à jour la base sont spécifiques à l'application. Dès lors, ils doivent être spécifiés dans le module de configuration. Nous obtenons ainsi un aspect persistance générique, réutilisable dans une autre application.

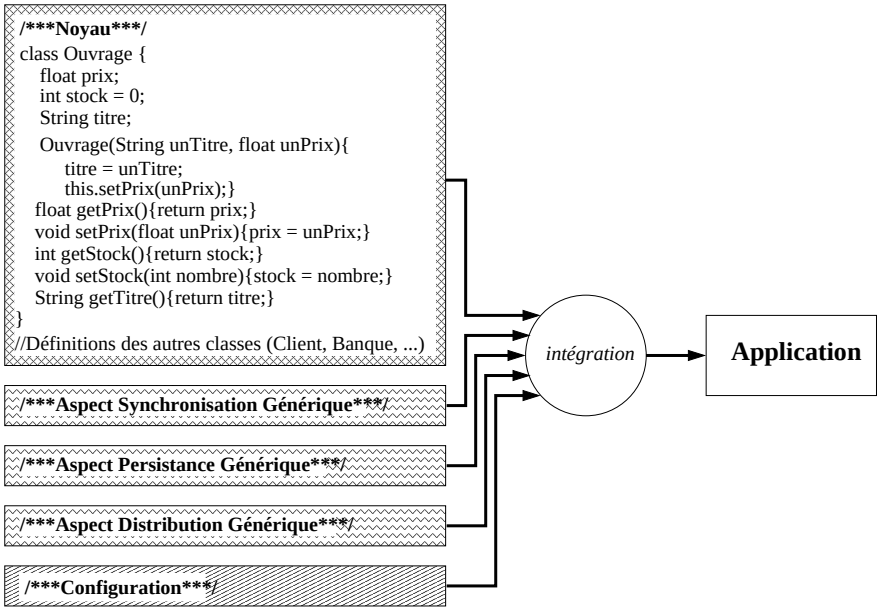

FIG. 4 – Programmer avec des aspects réutilisables

Outre la réutilisation des aspects, le recours à un module de configuration présente l'intérêt supplémentaire de permettre d'expliciter les problèmes de superposition/conflit d'aspects. En effet, l'ensemble des points de jonctions utilisés pour l'intégration de l'application apparaissent dans un unique module. Il devient alors plus aisé de traquer les points de jonction utilisés pour assembler plusieurs aspects et identifier les conflits potentiels qui peuvent en découler.

5 Conclusion

Dans cet article, nous avons identifié certaines limites des approches de programmation traditionnelles. Ainsi, nous avons mis en évidence le double problème de mélange et de dispersion du code représentant des propriétés transversales des logiciels. Les propriétés d'infrastructure (par exemple la communication distante, les transactions ou la tolérances aux pannes) représentent l'exemple type de propriétés transversales. En effet, dans un programme à base d'objets, le code de différentes propriétés d'infrastructure se trouve mélangé et dispersé dans différentes classes métier. Ce mélange et cette dispersion sont des freins à la production rapide de logiciels fiables.

Nous avons montré que ce frein est levé avec la programmation par aspects. Cette approche découple le code métier des différentes propriétés d'infrastructure. Chacune de ces propriétés transversales, appelées aspects, est définie dans un module spécifique. Ainsi, il devient possible de réutiliser le code métier dans différentes infrastructures. Cette réutilisation se fait en intégrant le code

métier avec les aspects spécifiques à l'infrastructure cible (par exemple protocole de communication distante, ou SGBD).

Nous avons également abordé la problématique complémentaire relative à la réutilisation des aspects. Il s'agit de définir des aspects génériques réutilisables avec différents codes métier. Nous avons esquissé une solution issue des travaux que nous menons sur la programmation par aspects.

Ces travaux s'inscrivent dans le cadre général de notre activité de recherche sur la simplification de la production de logiciel. Un second volet de cette activité concerne les composants logiciels. Ainsi, un de nos objectifs est de marier l'usage des composants et des aspects pour cumuler leurs avantages respectifs.

Pour valider nos résultats, nous avons développé un prototype appelé MetaclassTalk[1]. Basé sur les techniques de réflexion et de méta-programmation [Smi84][Mae87], MetaclassTalk constitue un laboratoire pour expérimenter différents paradigmes de programmation [Bou02]. Nous l'avons notamment exploité pour la mise en œuvre de la programmation par aspects [BL02].

[1] Disponible en téléchargement sur le site http ://csl.ensm-douai.fr/MetaclassTalk

Références

[BL01] N. Bouraqadi and T. Ledoux. Le point sur la programmation par aspects (aspect-oriented programming - in french). *Technique et Science Informatique*, 20(4) :505–528, 2001.

[BL02] N. Bouraqadi and T. Ledoux. Aspect-oriented programming using reflection. Technical Report 2002-10-3, Ecole des Mines de Douai, October 2002.

[Bou02] N. Bouraqadi. Metaclasstalk - a testbed for exploring programming pradigms. Talk given at the Europeen Smalltalk Users Group (ESUG) 10th Smalltalk Conference, Douai - France, August 2002.

[Dij76] Edgar W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1976.

[EFB01] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10) :29–32, October 2001.

[GGM99] J. M. Geib, C. Gransart, and Ph. Merle. *Corab. Des concepts à la pratique*. Dunod, 2nd edition, 1999.

[HL95] Walter L. Hursch and Cristina Videira Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, Février 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of ECOOP'97*, number 1241 in LNCS, pages 220–242. Springer-Verlag, Juin 1997.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155, Orlando, Florida, 1987. ACM.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. *Research Directions in Concurrent Object-Oriented Programming*, chapter Analysis of inheritance anomaly in object-oriented concurrent programming languages. MIT Press, 1993.

[OMG99] OMG. Corba components : Joint revised submission. Technical Report OMG TC Document obros/99-07-01, Object Management Group, August 1999.

[Sec00] Secrétariat d'Etat à l'Industrie. Technologies-clés 2005. http ://www.industrie.gouv.fr/observat/innov/carrefour/f2o_exer.htm, October 2000.

[Smi84] Brian C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages, POPL'84*, pages 23–35, Janvier 1984.

[Szy98] C. Szyperski. *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998.