Améliorer la conception des applications interactives par l'utilisation conjointe du modèle PAC et des patrons de conception

Thierry Duval, Jean-Claude Tarby

Améliorer la conception des applications interactives par l'utilisation conjointe du modèle PAC et des patrons de conception

Thierry Duval

IRISA (Siames) UMR 6074
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
thierry.duval@irisa.fr
tél. : (33)2 99 84 72 56 --- fax : (33)2 99 84 71 71

Jean-Claude Tarby

Laboratoire Trigone, Institut CUEEP
Université Lille 1
59655 Villeneuve d'Ascq Cedex, France
jean-claude.tarby@univ-lille1.fr
tél. : (33)3 20 43 32 62 --- fax : (33)3 20 43 32 01

RESUME
Des modèles d'architecture tels que MVC et PAC donnent lieu à de nombreuses possibilités d'implémentation. Ceci pose des problèmes aux étudiants qui ne savent pas quelle solution choisir lorsqu'ils doivent les coder. Depuis plusieurs années, nous avons mis au point une méthodologie qui implémente le modèle PAC en utilisant principalement les patrons de conception Proxy, Fabrique Abstraite, et Singleton. Grâce à cette méthodologie, les étudiants sont guidés dans la structuration de leur code, et l'IHM est séparée efficacement du noyau fonctionnel, montrant ainsi aux étudiants comment réaliser proprement cette séparation avec des propriétés d'évolution importantes : changement de l'IHM ou du code du noyau fonctionnel, mais aussi ajout d'interactivité à un noyau fonctionnel non interactif.

MOTS CLES : modèle d'architecture logicielle pour les IHM, modèle PAC, patrons de conception, méthodologie de conception des IHM.

ABSTRACT
Software architectural models like MVC and PAC are now well known, and can be implemented in many different ways. This is a problem for students when they have to implement these models, because they do not know how to select an appropriate solution. For now several years we have proposed a methodology to facilitate the implementation of the PAC model. This methodology is mainly based on Design Patterns such as « Proxy », « Abstract Factory », and « Singleton ». Thanks to this methodology, the structure of the source code is imposed and ensures an efficient separation between the application kernel and the GUI. Then, students learn to implement this separation efficiently, and they discover also with this separation that the software can evolve more easily: it is easy to change the GUI or the kernel without affecting the other part, and to add interaction possibilities to a non-interactive kernel.

CATEGORIES AND SUBJECT DESCRIPTORS:
D.2.11 [Software Engineering]: Software Architecture---Domain-specific architectures, Patterns; D.1.5 [Programming Techniques]: Object-Oriented Programming; H.5.2 [Information Interfaces and Presentation]: User Interfaces---Evaluation/methodology, Graphical user interfaces (GUI).

GENERAL TERMS: Design.

KEYWORDS: Software Architectural Models for GUI, PAC Model, Design Patterns, Methodology for GUI Design

INTRODUCTION

L'enseignement des IHM passe inévitablement, à un moment ou un autre, par la mise en application de concepts fondamentaux dont la séparation du noyau fonctionnel et de l'IHM. Sur ce dernier point, de nombreux modèles d'architecture ont été proposés dont MVC et PAC [3]. Ces modèles pouvant être implémentés de nombreuses façons, se pose alors le problème d'expliquer aux étudiants comment coder de façon rigoureuse ce type d'architecture.

Depuis plusieurs années, nous avons testé différentes façons d'enseigner l'implémentation de ces modèles. Au travers de ces essais, nous avons introduit petit à petit les patrons de conception [7][8] (ou « Design Patterns ») afin de faciliter et de structurer le codage. A terme, nous sommes arrivés à une méthodologie qui implémente le modèle PAC en se reposant sur ces patrons de conception. Cette méthodologie permet de mettre en œuvre efficacement le modèle PAC en utilisant principalement les patrons de conception "Proxy" (GoF207), "Fabrique Abstraite" (GoF87), et "Singleton" (GoF127).

La première partie de l'article expose notre constat quant à l'enseignement des IHM et des patrons de conception. Ceci nous amène à la méthodologie, que nous utilisons à présent dans nos enseignements, dont nous présentons la validation ainsi que son instrumentation dans la seconde et la troisième partie. Nous détaillons ensuite chacune des cinq étapes qui composent cette méthodologie en nous appuyant sur l'exemple de la résolution du problème des tours de Hanoï. Enfin, nous concluons par un bilan de l'utilisation de cette méthodologie.

UN CONSTAT SUR L'ENSEIGNEMENT DES IHM ET DES PATRONS DE CONCEPTION

Limites des outils actuels

Contrairement à ce que l'on pourrait croire, l'évolution permanente des outils de conception, qu'ils soient de type ateliers de génie logiciel, environnements de développement intégré ou outils RAD, va à l'encontre d'un enseignement « propre » des IHM. Par enseignement « propre », nous voulons dire permettant de concrétiser de manière significative des concepts fondamentaux tels que la séparation du noyau fonctionnel et de l'IHM, l'évolutivité de l'IHM indépendamment du noyau fonctionnel, la possibilité d'avoir plusieurs IHM branchées sur le même noyau fonctionnel, etc. Les outils de conception actuels (Visual Studio© ou NetBeans© pour ne citer qu'eux) prennent généralement le relais pour le codage de l'IHM, et occultent la phase de production de code, de même que la logique de structuration et de production de ce code. Comment faire alors pour enseigner correctement la conception d'applications interactives si les IHM produites avec ces outils sont produites automatiquement, c'est-à-dire sans aucune implication des étudiants ? Par ailleurs, la prise en charge de la conception des IHM par ces outils a des conséquences directes sur le travail et la méthodologie de travail des étudiants. Ainsi, on constate généralement trois grands problèmes récurrents : ① les étudiants créent l'IHM en 1er, et le noyau fonctionnel en 2nd, ② les étudiants mélangent le code de l'IHM et le code du noyau fonctionnel, ③ les étudiants ne voient pas l'intérêt de séparer l'IHM du noyau fonctionnel. Nous verrons par la suite que notre méthodologie est une solution à ces problèmes.

Les patrons de conception dans l'enseignement des IHM

Les patrons de conception (ou Design Patterns) apparaissent petit à petit dans les outils de conception, par exemple dans Borland© Together© pour la création de diagrammes de classes. L'intérêt d'utiliser les patrons de conception n'est plus à démontrer aujourd'hui, mais il est souvent difficile de faire comprendre cet intérêt aux étudiants. Les patrons de conception demandent en effet une faculté d'abstraction élevée. Généralement, les étudiants n'ont pas été confrontés à ce genre de réflexion avant, et cet « éloignement du clavier » les perturbe beaucoup. Ceci se renforce lorsqu'ils découvrent que les patrons de conception sont des structures de solutions, et non pas du code prêt à l'emploi. Il est du devoir de l'enseignant de faire comprendre aux étudiants qu'utiliser un patron de conception permet de gagner du temps dans la recherche de solution à un problème, et d'échanger plus facilement avec d'autres programmeurs.

Malheureusement, même après avoir compris l'intérêt des patrons de conception, les étudiants restent souvent persuadés qu'il n'est pas intéressant de les appliquer et qu'ils iraient plus vite en programmant eux-mêmes leurs solutions. Pour leur montrer qu'ils ont tort, nous avons conçu et appliqué depuis septembre 1998, en Master 2 Professionnel e-Services (anciennement DESS MICE) de l'Université Lille 1, un ensemble d'exercices visant à leur montrer l'utilité des patrons de conception. Cette suite d'exercices consiste tout d'abord, et avant le cours sur les patrons de conception, à coder en Java Swing un objet interactif simple. Une fois ce travail fait, on demande aux étudiants de réfléchir aux modifications qu'ils devront apporter à leur code dans le cas où l'objet doit changer de comportement ou d'IHM. Suite à cette demande, nous engageons une discussion avec les étudiants afin de leur expliquer que ces problèmes sont classiques et qu'ils ont déjà été résolus souvent, ce qui a donné naissance aux patrons de conception. Cette discussion nous permet de passer en douceur au cours sur les patrons de conception. Après le cours, nous demandons aux étudiants d'appliquer les patrons de conception Etat, Stratégie et Observateur, ce dernier étant tout d'abord codé « à la main », puis grâce aux classes Java présentes dans le JDK. A la fin de cette série d'exercices, les étudiants sont convaincus de l'intérêt des patrons de conception. Ils voient également très bien le lien entre les patrons de conception et les approches type MVC ou PAC. Enfin, ils comprennent l'intérêt de la séparation entre le noyau fonctionnel et l'IHM, ce qui leur permet de mieux évaluer les avantages et les limites des outils qu'ils utilisent habituellement.

Renforcement de la méthode d'enseignement

Il est donc possible d'enseigner conjointement la conception des IHM et les patrons de conception. L'approche présentée précédemment a fait ses preuves pendant plusieurs années, mais elle a également montré ses limites. En effet, chaque mise en pratique implique de repartir de zéro, et même si certains patrons de conception sont souvent utilisés, les étudiants sont obligés de coder chaque fois entièrement l'application.

Pour remédier à ce problème, nous avons renforcé notre méthodologie en ajoutant une nouvelle étape. Celle-ci montre qu'il est possible d'appliquer une méthodologie de conception basée sur les patrons de conception et PAC, permettant d'obtenir du code encore plus structuré et plus évolutif. C'est cette méthodologie que nous détaillons par la suite.

VALIDATION DE LA METHODOLOGIE

La méthodologie présentée dans cet article est utilisée

depuis septembre 2000 en Master 2 Professionnel spécialité Génie Logiciel (anciennement DESS ISA filière Génie Logiciel) à l'Université de Rennes 1, dans le cadre du mini projet d'IHM, afin de rendre visible, puis interactif, un noyau applicatif, correspondant à un jeu de carte de type solitaire, dont les étudiants ne possèdent pas le code source. La totalité des étudiants parvient chaque année à fournir une visualisation du jeu, et plus des deux tiers parviennent à fournir une version interactive du jeu à base de « glisser-déposer ». Depuis 2005, notre méthodologie est également utilisée dans le cadre du projet de fin d'étude où les étudiants doivent produire un logiciel interactif de synthèse sonore. Ici encore, presque tous les groupes d'étudiants parviennent à la mettre en œuvre. En 2005, elle a été mise en pratique avec succès dans le Master e-Services à l'Université Lille 1 dans le cadre de deux projets d'IHM, un en Java avec Swing et un autre en Flash Action Script 2. Enfin, elle a également été utilisée par d'anciens élèves pour le développement de plusieurs versions de logiciels (constitués de plusieurs milliers de classes) dans un contexte industriel [4].

INSTRUMENTATION DE LA METHODOLOGIE

Sans le support d'un outil informatique lui correspondant, la mise en pratique de cette méthodologie (par exemple avec des étudiants) peut provoquer un nombre important d'erreurs (oubli d'importation de classes, incohérence entre les noms de classes ou de méthodes, etc.) et cause souvent une lassitude de la part des étudiants puisque le code s'écrit de façon assez répétitive. Par ailleurs, il faut remarquer que la méthodologie n'est pas facile à appréhender pour des étudiants ne possédant pas des bases suffisantes, surtout en patrons de conception. C'est pourquoi nous avons instrumenté notre méthodologie au travers de l'outil ModX [10]. Cette instrumentation [12] est basée sur une approche dirigée par les modèles [9], ce qui nous donne une grande souplesse quant à de potentielles évolutions de notre méthodologie. Il nous est possible maintenant d'utiliser celle-ci avec des publics plus variés (des Licences par exemple). Notre outil dégage en effet les concepteurs des contraintes liées à la méthodologie, et ces derniers peuvent alors se concentrer sur leur conception en se « contentant de remplir des boîtes » (cf. figure 1). L'outil est capable de générer la majeure partie du code Java (cf. figure 2), et sait s'adapter aux modifications de la méthode. Avec notre outil, le concepteur est déchargé de l'écriture de tous les enchaînements basiques entre composants (les composants de contrôle et leurs abstraction et présentation associées), et il peut alors se focaliser sur l'expression de la dynamique du dialogue, à l'intérieur des composants de contrôle.

METHODOLOGIE DE CONCEPTION BASEE SUR PAC ET LES PATRONS DE CONCEPTION

Notre méthodologie repose sur l'utilisation de patrons de conception pour concevoir des applications interactives basées sur les modèles PAC [3] et PAC-Amodeus [11], et systématise la proposition d'implémentation de ces modèles qui a été faite dans [5][6]. Nous proposons de découper une application en trois parties principales : le noyau fonctionnel qui correspond à l'Abstraction, l'IHM qui correspond à la Présentation, et le contrôleur de dialogue représenté par le Contrôle. Nous verrons plus loin que ce découpage est plus complexe dans chacune de ces parties, mais qu'il respecte le modèle PAC. Dans un souci de clarté, nous prendrons l'exemple des tours de Hanoï pour illustrer notre méthodologie.

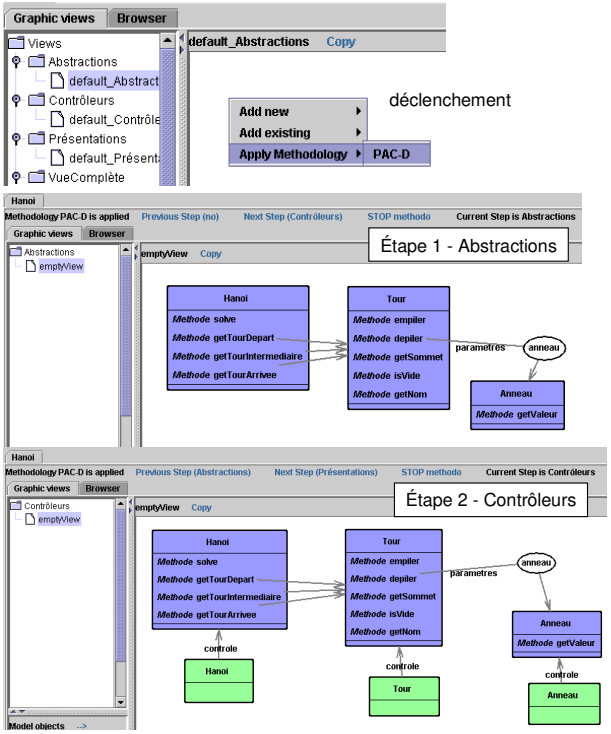

Figure 1 : Application de la méthodologie dans ModX

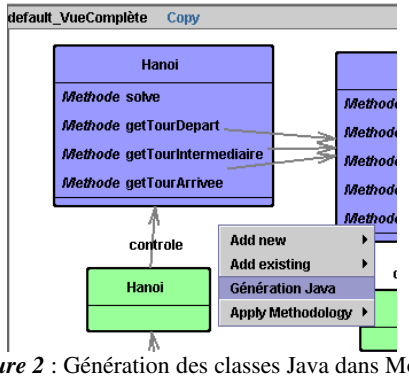

Figure 2 : Génération des classes Java dans ModX

Cadre de travail

Pour que notre méthodologie soit facilement applicable, plusieurs conditions doivent être réunies :

- Le noyau fonctionnel doit pouvoir être séparé de l'IHM.
- L'application doit être codée à l'aide d'un langage objet équivalent à Java (Action Script 2 de Macromedia Flash© convient très bien également).

- Les étudiants doivent avoir un niveau suffisant pour assimiler les niveaux d'abstraction de la méthodologie. Les publics de Master 2 ou équivalent sont un bon exemple. Rappelons que cette dernière condition peut être levée grâce à notre outil.

Détails de la méthodologie

Notre méthodologie se découpe en 5 étapes successives. La succession présentée ici est un exemple de parcours et correspond au cas le plus fréquent.

Principe de base. Le noyau fonctionnel contient toutes les abstractions qui sont gérées par l'application. L'ensemble de ces abstractions représente l'Abstraction de PAC. Chaque abstraction X du noyau fonctionnel, ainsi que chaque contrôle et chaque présentation associés à ces abstractions, sont représentés par leur interface (au sens UML), respectivement IA_X, IC_X et IP_X. Par ailleurs, l'interface IC de chaque « contrôle » hérite de l'interface IA de l'« abstraction » associée (cf. figure 3), ce qui permettra à un composant contrôle d'être un *proxy* de son composant abstraction associé. Cette caractéristique facilite grandement la structure du code et « soude » chaque abstraction à son contrôle, la présentation venant alors se « greffer » sur cette association abstraction-contrôle. Notons qu'il est interdit de rajouter des méthodes directement dans les classes sans les avoir ajoutées au préalable dans les interfaces associées.

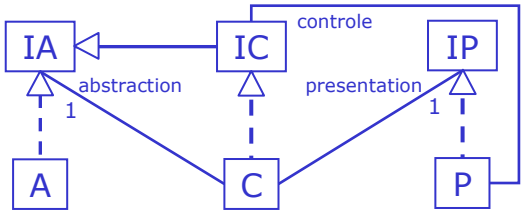

Figure 3 : Notre modèle d'implémentation de PAC.

Les classes et les interfaces sont groupées dans des paquetages pour plus de lisibilité. Généralement, nous créons les paquetages abstraction, interfaceAbstraction, presentation, interfacePresentation, controle, interfaceControle, dont les contenus sont directement en rapport avec leurs noms (par exemple interfaceAbstraction contient toutes les interfaces d'abstraction). Ce découpage en six paquetages se fait de façon transversale afin de pouvoir proposer des paquetages d'implémentation différents pour les parties Abstraction, Contrôle, et Présentation. Cependant, lorsqu'on a affaire à un très grand nombre de classes, on a intérêt à procéder en plus à un découpage hiérarchique, comme présenté dans [4], et donc il est intéressant de proposer, dans ce cas, des fabriques de fabriques.

Par ailleurs, pour des raisons de simplicité de mise en œuvre, mais aussi d'évolution et de maintenance, nous créons un paquetage concreteFactory qui contient une classe et son interface, chargées de définir et de renvoyer des fabriques associées aux abstractions, contrôles et présentations et réalisant la création toutes les instances de ces classes. Il est à noter que, grâce à notre méthodologie, ce paquetage est invariant d'une application à une autre. En effet, le code de la concreteFactory reste le même quelle que soit l'application, hormis l'importation des paquetages relatifs à l'application. Ceci peut être généré automatiquement très facilement.

Etape 1 : Elaboration du Noyau Fonctionnel (IA et A). Cette première étape consiste à définir les interfaces des abstractions (IA). Pour chaque abstraction X, on définit dans son interface IA_X les méthodes qui permettent d'accéder aux données associées à l'abstraction. Lorsque l'interface IA_X est définie pour une abstraction X, on crée ensuite la classe abstraction A_X associée.

Quand toutes les interfaces d'abstraction ont été produites, on crée une fabrique A_Factory, en utilisant le patron de conception *Fabrique Abstraite*, qui se charge de créer toutes les abstractions. Cette fabrique, qui possède également une interface IA_Factory, ne devant être créée qu'à un seul exemplaire, nous utilisons le patron de conception *Singleton* pour écrire le code qui lui correspond.

Cette étape oblige les étudiants à réfléchir aux abstractions à manipuler (lesquelles ? pourquoi ? quelles méthodes associées ?...) et surtout en restant totalement indépendant de toute IHM. De plus, les étudiants doivent immédiatement intégrer les patrons de conception Singleton et Fabrique Abstraite dans leur code.

Dans le cas des tours de Hanoï, les abstractions sont A_Anneau, A_Tour, et A_Hanoi (qui permet la résolution du problème). Par conséquent, notre méthodologie donnera naissance aux interfaces IA_Anneau, IA_Tour et IA_Hanoi. Parmi les méthodes, on trouve, par exemple pour IA_Tour, void depiler () et void empiler (IA_Anneau a).

L'interface de fabrique IA_Factory contient quant à elle des méthodes telles que IA_Tour newA_Tour (String nom, int n) ou IA_Anneau newA_Anneau (int v), que la fabrique A_Factory implémente[1] avec :

```
public IA_Tour newA_Tour (String nom, int n) {
    return new A_Tour (nom, n) ;
}
public IA_Anneau newA_Anneau (int v) {
    return new A_Anneau (v) ;
}
```

Etape 2 : Création des Interfaces de Présentation (IP) et de Contrôle (IC). Pour cette seconde étape, on peut commencer indifféremment par les interfaces de présen-

[1] Pour ne pas surcharger le code présenté dans l'article, nous avons volontairement omis des portions de code dont les « import » pour lesquels la quasi-totalité peut être déduite de la structure présentée sur la figure 3. Pour le code complet de tous les exemples présentés dans ce document, se reporter à : http://perso.univ-rennes1.fr/thierry.duval/IHM2006.

tation ou de contrôle. L'important est de maintenir la cohérence entre les deux.

Les Interfaces de Présentation.
Par défaut, une interface de présentation est supposée contenir des signatures de méthodes similaires aux signatures des méthodes de l'interface d'abstraction associée afin de pouvoir répercuter visuellement les effets des méthodes de l'abstraction. Cela signifie que si IA_X contient une méthode f(IA_Y y) (resp. f()[2]), alors IP_X contiendra une méthode f(IP_Y y) (resp. f()). Pour éviter des appels en boucle infinie sur la présentation suivant que l'appel provient du contrôle ou directement de l'interaction de l'utilisateur, nous avons choisi de renommer les méthodes f(IP_Y y) (resp. f()) en f_out(IP_Y y) (resp. f_out()). Il est bien évident que ces méthodes f_out sont optionnelles. Le concepteur peut donc les supprimer avant de passer à l'étape 3.

Si l'IHM doit piloter le noyau fonctionnel[3], cela signifie que des présentations P_X envoient des messages aux abstractions A_X associées en passant par les contrôles C_X. Dans ce cas, d'une part on ajoute aux interfaces de présentation IP_X une méthode IC_X getControle() qui renvoie le contrôle C_X associé, et d'autre part, pour éviter à nouveau des appels en boucle infinie, les méthodes que l'on ajoute à la présentation (IP_X et plus tard P_X), s'appelleront f_in(IP_Y y) ou f_in(), suivant que le concepteur décide ou non d'utiliser des paramètres. Ces méthodes f_in sont également optionnelles. A ce pilotage direct, via des appels à des méthodes de l'abstraction, peut s'ajouter un pilotage indirect qui nécessite des étapes intermédiaires d'échanges entre la présentation et le contrôle avant de déclencher une méthode de l'abstraction. On peut donc être amené dans ce dernier cas à ajouter de nouvelles méthodes, de type g_in et g_out, qu'il n'est pas possible de déduire automatiquement des interfaces d'abstraction, dans les interfaces de contrôle et de présentation.

Dans notre exemple des tours de Hanoï, parmi les méthodes d'interfaces de présentation, on trouve ainsi dans IP_Anneau la méhode IC_Anneau getControle(), et dans IP_Tour la méthode void empiler_out (IP_Anneau pa).

Dans tous les cas, les paramètres des méthodes de l'interface de présentation ne peuvent être que de type « interface de présentation », de type « interface de contrôle » et des types standards (booléen, etc.).

Ces propriétés des interfaces de présentation garantissent que l'IHM est fortement découplée du reste de l'application et peut ainsi être facilement remplacée par une autre IHM respectant ces mêmes contraintes.

Les Interfaces de Contrôle.
Les composants de contrôle seront les proxys de leurs abstractions associées, c'est pourquoi ils devront implémenter la même interface que leur abstraction. Les interfaces de contrôle IC_X héritent donc des interfaces IA_X et contiennent en plus des méthodes IP_X getPresentation() qui renvoient les présentations qui leur sont associées. Ainsi, dans IC_Anneau et IC_Tour, on trouve des méthodes getPresentation() qui renvoient respectivement un IP_Anneau et un IP_Tour. De plus, comme indiqué précédemment, si des méthodes g_in() ou g_in(IP_Y y) (et éventuellement g_out() ou g_out (IP_Y y)) ont été ajoutées au niveau des interfaces de présentation pour piloter le noyau fonctionnel, alors on doit trouver de nouvelles méthodes g() ou g(IC_Y y) au niveau des interfaces de contrôle afin de faire la jonction entre la partie présentation et la partie abstraction. Par exemple, IC_Tour contient des méthodes, appelées par son composant de présentation, dont le but est de traiter les événements graphiques d'interaction de l'utilisateur. C'est le cas de entreeDnD(IC_Anneau a), qui réagit au début du survol d'une tour par un anneau manipulé par l'utilisateur, et qui est appelée par la méthode entreeDnD_in de la présentation.

A la fin de cette étape, on ajoute le code de l'interface de la fabrique, IP_Factory, chargée de créer les interfaces de présentation. Par défaut, les signatures des méthodes de IP_Factory sont directement issues de IA_Factory, même si tout n'est pas nécessaire pour la version finale ; le concepteur pourra ensuite éliminer les méthodes ou les paramètres inutiles. Notons également qu'un paramètre IC_X est automatiquement ajouté dans les paramètres ; à charge du concepteur de l'utiliser ou non.

Pour les tours de Hanoï, IP_Factory contient par défaut :
```
IP_Tour newP_Tour (String nom, int n, IC_Tour controle) ;
IP_Anneau newP_Anneau (int v, IC_Anneau controle) ;
IP_Hanoi newP_Hanoi (int n, IC_Hanoi controle) ;
```
car IA_Factory contient :
```
IA_Tour newA_Tour (String nom, int n) ;
IA_Anneau newA_Anneau (int v) ;
IA_Hanoi newA_Hanoi (int n) ;
```

Etape 3 : Création des Présentations (P). Les interfaces de présentation ayant été définies à l'étape 2, il reste à écrire le code de chaque méthode présente dans ces interfaces. Pour cela, nous pouvons édicter plusieurs règles dont :

- Par défaut, le constructeur de chaque présentation est du type :
  ```
  public P_X (IC_X controle) { this.controle = controle ; }
  ```

[2] Pour des raisons de lisibilité, nous utiliserons le nom « f » pour les méthodes qui ont d'abord été créées dans l'abstraction, et le nom « g » pour les méthodes que l'on crée d'abord dans la présentation (ou le contrôle) et pour lesquelles il n'y a pas forcément de fonction g correspondante dans l'abstraction. Ces méthodes « g » peuvent donc être paramétrées par des IC_Y et non par des IA_Y, contrairement aux méthodes « f » qui implémentent des méthodes déclarées dans l'interface de l'abstraction.

[3] Dans certains cas, l'IHM ne sert qu'à visualiser les actions qui se déroulent dans le noyau fonctionnel, et donc n'envoie aucune donnée au noyau fonctionnel.

ce qui donne en version finale pour P_Anneau :

```
public P_Anneau(int v, IC_Anneau controle) {
    setSize (unite * v, unite) ;
    setPreferredSize (getSize ()) ;
    setBackground (Color.blue) ;
    this.controle = controle ;
}
```

- Si, à la suite de l'étape 2, les interfaces de présentation IP_X contiennent des méthodes f_out(IP_Y y) ou f_out(), alors les classes de présentation P_X implémentent ces méthodes. Leurs contenus sont vides par défaut ; on ne peut pas en effet deviner quel sera le comportement de la présentation. Cependant, notre méthodologie pourrait être couplée avec des approches liées à la plasticité des IHM [1] pour pouvoir générer le code du comportement de l'IHM.
  L'abstraction Tour (IA_Tour et A_Tour) contient par exemple les méthodes void depiler() et void empiler(IA_Anneau a). La présentation associée (IP_Tour et P_Tour) contient donc par défaut les méthodes void depiler_out() et void empiler_out(IP_Anneau a).

- Si, à la suite de l'étape 2, les interfaces de présentation IP_X contiennent des méthodes f_in(IP_Y y) (resp. f()), ou g_in(IP_Y y) (resp. g()), alors les classes de présentation P_X implémentent ces méthodes. Le contenu de ces méthodes est par défaut :

```
f_in(IP_Y y) { controle.f(y.getControle()); }
(resp. f_in() { controle.f(); }
```

  sachant qu'on peut ajouter d'autres lignes de code. La méthode f(IA_Y y) (resp. f()) aura été ajoutée préalablement dans le contrôle associé (IC_X et C_X).
  Par exemple, si la présentation détecte une action de l'utilisateur qui tend à explicitement dépiler un anneau d'une tour, cela se fera via la méthode :

```
void empiler_in (IP_Anneau a) {
    controle.empiler (a.getControle ()) ;
}
```

  à laquelle il n'y aura ici rien à ajouter. De même, si on a besoin d'une méthode qui sert à détecter l'arrivée d'un anneau sur une tour, qui correspond à un dialogue intermédiaire entre présentation et contrôle en vue d'un empilement à terme, on aura une méthode :

```
void entreeDnD_in (IP_Anneau a) {
    controle.entreeDnD (a.getControle ()) ;
}
```

A la fin de cette étape, en utilisant le patron de conception Singleton, on ajoute le code de la fabrique chargée de créer les présentations, c'est-à-dire P_Factory. Le code de cette fabrique est entièrement déduit, y compris les import, d'une part de son abstraction IP_Factory, et d'autre part des constructeurs des présentations P_X. Par exemple, pour le cas de Hanoï, la P_Factory contient :

```
package presentation;

import interfaceControle.IC_Anneau;
(... autres « import »...)

public class P_Factory implements IP_Factory {
    public IP_Tour newP_Tour (String nom, int n, IC_Tour controle) {
      return (new P_Tour (n, controle)) ;
    }
    public IP_Anneau newP_Anneau (int v, IC_Anneau controle) {
      return (new P_Anneau (v, controle)) ;
    }
    public IP_Hanoi newP_Hanoi (int n) {
      return (new P_Hanoi (n)) ;
    }
    protected P_Factory () { }
    private static IP_Factory instance = new P_Factory () ;
    public static IP_Factory getInstance () { return instance ; }
}
```

Etape 4 : Création des Contrôles (C). De même que pour les présentations à l'étape 3, il reste à écrire ici le code de chaque méthode présente dans les interfaces de contrôle. Plusieurs règles existent là aussi, dont :

- Par défaut, un constructeur de contrôle est du type :

```
public C_X (UnTypeY y) {
    abstraction = ConcreteFactory.getAFactory().newA_X(y) ;
    presentation = ConcreteFactory.getPFactory().newP_X(y, this) ;
}
```

  ce qui donne pour C_Anneau :

```
public C_Anneau (int v) {
    abstraction = ConcreteFactory.getAFactory().newA_Anneau(v) ;
    presentation = ConcreteFactory.getPFactory().
                        newP_Anneau(v, this) ;
}
```

  La fabrique concrète (ConcreteFactory) sera abordée à l'étape 5. Etant donné le lien d'héritage entre IA et IC (cf. figure 3), les paramètres du constructeur du contrôle sont compatibles (c'est-à-dire identiques, dérivés ou encore implémentant la même interface) avec ceux du constructeur de l'abstraction associée, par exemple int v dans le cas ci-dessus.

- Si une méthode f(IA_Y y) (resp. f()) a été créée pour une abstraction X dans le noyau fonctionnel (IA_X et A_X), alors on a une méthode de même nom f(IA_Y y) (resp. f()) dans le contrôle associé (IC_X et C_X) et le code de cette méthode contient par défaut :

```
f(IA_Y y) {
    abstraction.f(y) ;
    presentation.f _out(((IC_Y)y).getPresentation() ) ;
}
( resp. f() { abstraction.f() ; presentation.f_out() ; } )
```

  sachant que l'appel à la présentation peut éventuellement être enlevé suivant l'application, et qu'on peut ajouter d'autres lignes de code. « ((IC_Y)y).getPresentation() » signifie qu'on transtype y en IC, et qu'ensuite on demande la présentation associée. Comme précédemment, abstraction et presentation sont deux variables pointant respectivement sur l'abstraction et la présentation associées au contrôle, et sont initialisées dans le constructeur de la classe contrôle.
  Nous avons vu à l'étape 1 que l'abstraction A_Tour contient la méthode empiler(IA_Anneau). Afin de pouvoir offrir des possibilités d'interaction par les anneaux présents dans les tours, sa version finale sera :

```
public void empiler (IA_Anneau aa) {
    if ( ! isVide ()) {
      ((IC_Anneau)getSommet ()).setSelectionnable (false) ;
    }
    abstraction.empiler (aa) ;
    IP_Anneau pa = ((IC_Anneau)aa).getPresentation();
    presentation.empiler_out(pa);
    ((IC_Anneau)aa).setSelected (false) ;
    ((IC_Anneau)aa).setSelectionnable(true);
}
```

Ces méthodes f peuvent être éventuellement appelées par des méthodes f_in de la présentation.

- Si, à la suite de l'étape 2, les interfaces de présentation IP_X contiennent des méthodes g_in(IP_Y y) (resp. g()) qui ne correspondent pas directement à des méthodes de l'abstraction, alors les classes de contrôle C_X implémentent des méthodes g(IC_Y y) (resp. g()) dont le contenu proposé par défaut est :

```
g(IC_Y y) {
    presentation.g_out (y.getPresentation()) ;
}
( resp. g() { presentation.g_out() ; } )
```

sachant qu'on peut modifier ce contenu et lui ajouter d'autres lignes de code. « presentation.g_out (y.getPresentation()) » est là uniquement pour indiquer au concepteur qu'il doit éventuellement penser à une rétroaction sur la présentation, et que celle-ci peut se faire par l'intermédiaire d'un paramètre de type IP ou autre, par exemple un booléen. C'est ainsi que la méthode entreeDnD, qui assure la reconnaissance de l'arrivée d'un anneau sur une tour en mode Drag'n Drop, sera modifiée et complétée pour obtenir un retour visuel adapté à la nature du survol, selon que l'anneau est empilable ou non sur la tour :

```
void entreeDnD(IC_Anneau a) {
    presentation.entreeDnD_out (isEmpilable (a)) ;
}
```

Cette visualisation d'empilabilité se fait ici au niveau du composant Tour.

Pour terminer cette étape, on ajoute le code de la fabrique chargée de créer les contrôles, c'est-à-dire C_Factory. Avec l'exemple de Hanoï, C_Factory contient par défaut la méthode newA_Anneau(int v) car A_Anneau a un constructeur A_Anneau(int v), la méthode newA_Tour (String nom, int n) car le constructeur de A_Tour est A_Tour (String nom, int n), etc. Le code par défaut de ces deux méthodes (et auquel il n'y a rien à ajouter) est :

```
public IA_Anneau newA_Anneau (int v) {
    return (new C_Anneau (v)) ;
}
public IA_Tour newA_Tour (String nom, int nbAnneauxMax) {
    return (new C_Tour (nom, nbAnneauxMax)) ;
}
```

De même que toutes les fabriques précédemment énoncées, C_Factory implémente le patron de conception Singleton qui se traduit dans ce cas précis par :

```
protected C_Factory () {}
private static IA_Factory instance = new C_Factory () ;
public static IA_Factory getInstance () { return instance ; }
```

Etape 5 : Finalisation du code. Après avoir réalisé ces quatre étapes, le code produit est complété par l'écriture du paquetage correspondant à la concreteFactory qui aura la charge d'instancier les classes définies lors de ces quatre étapes[4]. Le code de cette « concrete factory » est :

```
package concreteFactory;
 (...import ...)
public class ConcreteFactory {
  protected static IA_Factory aFactory ;
  public static void setAFactory (IA_Factory f) { aFactory = f ; }
  public static IA_Factory getAFactory () { return (aFactory) ; }
  (... idem pour C_Factory, et P_Factory...)

  public static void setFactory (IA_Factory f) { aFactory = f ;}
  public static IA_Factory getFactory (){
      IA_Factory factory = aFactory ;
      if (cFactory != null) {
         factory = cFactory ;
      }
      return (factory) ;
  }
}
```

Les trois méthodes getA_Factory, getC_Factory et getP_Factory sont destinées à être utilisées par des composants de contrôle, et vont permettre d'obtenir respectivement des instances de composants applicatifs, de contrôle, et de présentation. La méthode getFactory est quant à elle destinée aux composants applicatifs initiaux : elle retourne en priorité la C_Factory si elle existe, ou la A_Factory si aucune C_Factory n'a été proposée. C'est donc cette méthode qui doit être utilisée par les composants applicatifs (pouvant ainsi être utilisés en dehors d'une implémentation PAC) pour créer les différents composants qu'une abstraction pourrait avoir besoin de créer. Ceci a pour résultat de substituer aux composants applicatifs initiaux des composants de contrôle quand c'est possible, afin d'obtenir des applications graphiques interactives à la place des applications non graphiques initiales.

Il reste alors à écrire le code du main. Celui-ci aura en charge la création de l'Abstraction (et donc de toutes les abstractions qui la composent), de la Présentation (idem) et du Contrôle (idem), ainsi que la création des liens entre toutes ces instances. Par défaut, le code sera :

```
public static void main (String args []) {
   //création de l'Abstraction
   concreteFactory.ConcreteFactory.
       setAFactory (abstraction.AFactory.getInstance ()) ;
   //création du Contrôle
   concreteFactory.ConcreteFactory.
       setCFactory (controle.CFactory.getInstance ()) ;
   //création de la Présentation
   concreteFactory.ConcreteFactory.
       setPFactory (presentation.PFactory.getInstance ()) ;
}
```

C'est ici qu'on choisit les paquetages effectivement utilisés pour créer l'application interactive, en particulier le paquetage de présentation qui s'appuie sur une API

[4] Nous rappelons que ce paquetage est le même quelle que soit l'application modélisée.

graphique particulière[5] (pour notre exemple, on peut proposer un choix parmi trois paquetages s'appuyant respectivement sur les API AWT, Swing et SWT).

Il reste alors à ajouter les lignes propres à l'application, et à lancer éventuellement un traitement au niveau de l'Abstraction ou de la Présentation.

CONCLUSION

La méthodologie présentée ici possède deux avantages majeurs. Tout d'abord, elle impose un découpage précis des paquetages, et structure de façon rigoureuse les classes ainsi que le code des méthodes. Enfin, elle permet d'étendre facilement le code existant, en surchargeant les interfaces définies (IA_X, IP_X et IC_X), ainsi que les classes qui en dépendent, de façon à étendre par exemple les fonctionnalités de l'application.

Cette méthodologie implique l'écriture d'un nombre important de classes et de lignes de code dont une grande partie peut être générée automatiquement grâce à l'outil ModX. Cela concerne par exemple les import dans les IP, IC, P et C, ou bien encore l'application des règles d'écriture des méthodes de l'étape 3 et de l'étape 4. Rappelons qu'avec ce type d'outil, un concepteur se trouve déchargé de l'écriture de tous les enchaînements basiques entre composants et qu'il peut alors se focaliser sur l'expression de la dynamique du dialogue, à l'intérieur des composants de contrôle.

Récemment, nous avons montré que les capacités d'extension du code produit par notre méthodologie permettent l'intégration automatique de traces [2]. Ces traces sont générées par le noyau fonctionnel et par l'IHM, sous forme de fichiers XML. Grâce à ces traces, il nous est possible de savoir comment réagissent le noyau fonctionnel et l'IHM (débuts et fins datés d'appels de méthodes, avec gestion des appels en cascade). Ceci nous permet par exemple de comprendre comment une personne utilise l'application, mais aussi de voir si son comportement évolue au fur et à mesure de l'utilisation de l'application. Ces traces nous permettent de voir aussi si l'IHM répond aux attentes du concepteur, si l'utilisateur change son comportement au fur et à mesure des sessions, ou bien encore si des personnes différentes utilisent la même IHM de façons différentes. Nous travaillons actuellement sur des mécanismes de trace plus évolués (politiques de traces, formatages divers, etc.) basés sur notre méthodologie.

REMERCIEMENTS

Les auteurs remercient pour leurs supports financiers partiels le programme MIAOU du contrat de plan Etat Région Nord Pas de Calais et le FEDER.

[5] Ceci dans le cas où on implémente plusieurs présentations au choix.

BIBLIOGRAPHIE

1. Calvary, G., Coutaz, J. & Thevenin, D., "A Unifying Reference Framework for the Development of Plastic User Interfaces", in R. Little & L. Nigay (eds.), *Proceedings of the 2001 Engineering of Human--Computer Interaction Conference (EHCI'2001)*, Lecture Notes in Computer Science, Springer-Verlag.
2. Caulier S., Deschodt C. "Projet d'implémentation de PAC et PAC Amodeus avec des Design Patterns", *rapport de projet de l'UE Génie Logiciel des IHM (GLIHM) du Master Professionnel E-Services 2005-2006*, Université Lille 1, France.
3. Coutaz J. *Interfaces homme-ordinateur, conception et réalisation*, Dunod informatique, 1990.
4. Degrigny F., Duval T. "Utilisation du modèle PAC-Amodeus pour une réutilisation optimale de code dans le développement de plusieurs versions d'un logiciel commercial", In *Actes de la 16ème Conférence Francophone sur l'Interaction Homme-Machine (IHM'04)*, Namur, Belgique, septembre 2004, pp. 149–156.
5. Duval T, Nigay L. "Implémentation d'une application de simulation selon le modèle PAC-Amodeus", In *Actes de la 11ème Conférence Francophone sur l'Interaction Homme-Machine (IHM'99)*, Montpellier, France, novembre 1999, pp. 86–93.
6. Duval T., Pennaneac'h F. "Using the PAC-Amodeus Model and Design Patterns to Make Interactive an Existing Object-Oriented Kernel", In *Proceedings of the TOOLS EUROPE 2000 Conference*, Mont Saint-Michel, France, IEEE, juin 2000, pp. 407–418.
7. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns : Elements of reusable Object-Oriented Software*, Addison-Wesley, 1995.
8. Grand M. *Patterns in Java Volume 1 : A Catalog of Reusable Design Patterns Illustrated with UML*, Wiley Publishing Inc., 2002.
9. Kent, S. "Model Driven Engineering", In *Proceedings of the 3rd International Conference on Integrated Formal Method IFM 2002* (May 15-17, 2002, Turku, Finland), Butler, M., Petre, L., Sere, K., (eds.), LNCS vol. 2335, Springer-Verlag, May 2002, pp. 286-298.
10. Le Pallec, X. "The ModX Home Page", http://noce.univ-lille1.fr/projets/ModX, 2005.
11. Nigay L. "Conception et modélisation logicielle des systèmes interactifs : application aux interfaces multimodales", *Thèse de Doctorat de l'Université de Grenoble 1, IMAG*, 1994.
12. Tarby J-C., Marvie R., Le Pallec X., Nebut M. "Processus de modélisation incrémentale pour le développement d'applications interactives basées sur PAC", *Rapport de recherche LIFL 2006-02*, http://www.lifl.fr, Lille, France, 2006.