Méthode d'implémentation efficace des modèles PAC et PAC-Amodeus

Thierry Duval
IRISA
Campus de Beaulieu
35042, Rennes, France
thierry.duval@irisa.fr

RESUME
Ce tutorial propose une méthode permettant de mettre en œuvre efficacement les modèles PAC et PAC-Amodeus en utilisant des patrons de conception bien connus. Cette méthode est présentée ici en deux étapes : tout d'abord pour visualiser graphiquement des concepts d'un noyau fonctionnel initial, en créant de nouveaux composants servant de proxys à certains composants applicatifs et associés à des composants de présentation, et ensuite pour rendre interactifs ces nouveaux composants, par héritage des composants illustrés. Pour ne pas compliquer inutilement les exemples nous nous focaliserons essentiellement sur cette première étape de visualisation, qui sera abordée de trois façons différentes, afin de montrer plusieurs mises en œuvres possibles des modèles PAC et PAC-Amodeus. La seconde étape, qui consiste à rendre interactifs les composants ainsi rendus visibles, sera ensuite la même pour les trois approches. L'exemple utilisé pour décrire les méthodes employées sera celui des tours de Hanoï, que l'on concevra tout d'abord d'un point de vue logique (c'est-à-dire non graphique et non interactif), puis que l'on fera évoluer en une version illustrée, puis finalement en une version interactive où l'utilisateur pourra déplacer les anneaux d'une tour à une autre en mode glisser - déposer (Drag'n Drop). La réalisation se fera en utilisant le langage java et son API graphique Swing.

Mots Clés
Implémentation de modèles d'architecture pour les IHM, Patrons de conception pour les IHM

ABSTRACT
This aim of this tutorial is to propose a methodology to implement efficiently the PAC and PAC-Amodeus software architectural models for Graphical User Interfaces. The idea is to be able to associate to these two models the use of well known sofware design patterns. First, we propose a methodology able to visualize the concepts embedded in an object oriented kernel : we present three ways to create new components that will serve as proxys to the initial objects we want to visualize. Then, we propose a way to extend these proxys to make interactive the initial software. Showing how to solve the HanoïTowers problem is the example taken to illustrate this methodology. This software is first designed to print to the screen how to solve the problem, then we propose a graphical visualisation to the solving of the problem, and last we propose the user to solve the problem by himself, by interactively dragging and dropping the rings from one tower to another, thanks to direct manipulation of the rings.

Keywords
Implementation of Software Architectural Models for GUI, Design Patterns for GUI

CATEGORIES AND SUBJECT DESCRIPTORS
D.1 [Software] : Programming Techniques; D.2 [Software] : Software Engeenering; H.5 [Information Systems] : Information Interfaces and Presentation

GENERAL TERMS
Design

INTRODUCTION
Ce tutorial propose une méthode permettant de mettre en œuvre efficacement les modèles PAC [1] et PAC-Amodeus [8] (l'extension du modèle PAC inspirée du modèle Arch [9]) en utilisant des patrons de conception bien connus détaillés dans le "GoF" [5]. Ces patrons de conception sont principalement le "Proxy" (GoF207), le "Template Method" (GoF325), "l'Abstract Factory" (GoF87), le "Singleton" (GoF127), et également "l'Observer" (GoF293). On pourra aussi comparer ceci à l'usage d'autres patrons de conception présentés dans [6] tels que l'Interface et le Delegation.

Cette méthode est présentée en deux temps. Tout d'abord pour visualiser graphiquement des concepts d'un noyau fonctionnel initial, en créant de nouveaux composants servant de proxys (patron "Proxy" (GoF207)) à certains composants applicatifs et associés à des composants de présentation. Ensuite nous montrons comment rendre interactifs ces nouveaux composants, par héritage des composants illustrés.

L'idée est ici de définir les services rendus par les éléments applicatifs à l'aide d'interfaces que ces éléments doivent

alors implémenter (patron “Template Method” (GoF325), ou tout simplement le patron Interface). Cela permet alors d’utiliser des fabriques de composants (patron “Abstract Factory” (GoF87)) pour créer ces composants de façon à ce que des composants visuels ou interactifs associés soient les plus indépendants possible de l’implémentation effective des composants applicatifs.

Pour chaque composant applicatif à rendre interactif, il est alors possible de fournir un proxy qui va être un composant chargé de gérer l’accès au composant applicatif et de mettre en place une infrastructure permettant de répercuter visuellement les évolutions de ce composant applicatif via un composant de présentation qui lui sera associé plus ou moins directement selon le modèle d’architecture choisi.

Si le modèle d’architecture retenu est PAC, le composant servant de proxy sera la facette contrôle d’un composant PAC, le composant applicatif étant alors la facette abstraction correspondante.

Si le modèle d’architecture choisi est PAC-Amodeus, le composant servant de proxy sera un composant Adaptateur de Noyau Fonctionnel, lequel sera ensuite rattaché à une facette abstraction d’un composant PAC. On a vu dans [3] et [4] qu’une telle décomposition pouvait ensuite être optimisée pour supprimer le composant ANF ainsi que la facette abstraction, on se retrouve alors dans un schéma équivalent à celui obtenu avec le modèle PAC.

Dans le patron de conception Proxy, le composant proxy a exactement la même interface que le composant dont il régule l’accès. Ici, dans nos différents cas d’utilisation, nos composants proxy auront une interface étendue pour permettre une utilisation dans le cadre des IHM, pour permettre notamment l’accès à leur composant de présentation associé.

On parlera également ici de l’usage du patron de conception Singleton (GoF127) pour que les composants qui en ont besoin puissent facilement accéder à la fabrique de composants, comme évoqué dans [2].

L’exemple utilisé pour décrire les méthodes employées sera celui des tours de Hanoï, que l’on concevra tout d’abord d’un point de vue logique (c’est-à-dire non graphique et non interactif), puis que l’on fera évoluer en une version illustrée, puis finalement en une version interactive où l’utilisateur pourra déplacer les anneaux d’une tour à une autre en mode glisser - déposer (Drag’n Drop). La réalisation se fera en utilisant le langage java et son API graphique Swing.

PREMIÈRE SOLUTION : PAC ET PROXY CONTRÔLE PAR DÉLÉGATION

Nous ferons ici l’usage du concept bien connu d’Interface, tel qu’il est décrit dans [6] (le patron du GoF qui s’en rapproche le plus est le Template Method), nous utiliserons également le concept de Proxy, en l’implémentant sous sa forme classique qui est proche du patron de conception Delegation. Enfin, les proxys se substitueront à leurs objets délégués grâce à l’utilisation du concept d’Abstract Factory.

Le principe

Les composants de contrôle associés à des objets applicatifs sont conçus sur la base du patron de conception Proxy : chaque composant contrôle sera donc chargé du filtrage des communications entre son composant applicatif associé et les autres composants, comme illustré figure 1. Dans cette utilisation on peut également considérer qu’on fait usage du patron de conception Delegation décrit dans [6].

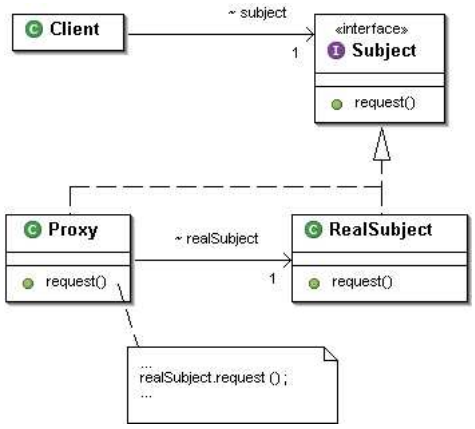

Figure 1. Patron de conception Proxy

De plus, pour éviter une trop grande dépendance du proxy vis à vis de son composant associé, on utilise le patron de conception Template Method (GoF325) comme illustré figure 2 (ou tout simplement l’Interface) et on force ainsi le concepteur à définir l’interface d’accès à chacun de ses composants applicatifs.

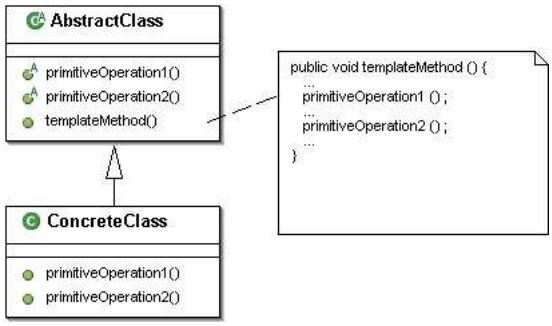

Figure 2. Patron de conception Template Method

Indirectement, cette approche permet alors un premier usage du patron de conception Abstract Factory pour permettre une évolution du comportement des composants applicatifs indépendamment des composants de contrôle associés. Cette première fabrique de composants sera ici essentiellement utilisée dans les classes de contrôle clientes des classes applicatives.

Par la suite, on demande également que chaque composant contrôle implémente l’interface de son composant applicatif associé, conformément au patron de conception Proxy. Une seconde utilisation du patron Abstract Factory, illustré figure 3 dans ce contexte, permet alors de substituer à chaque

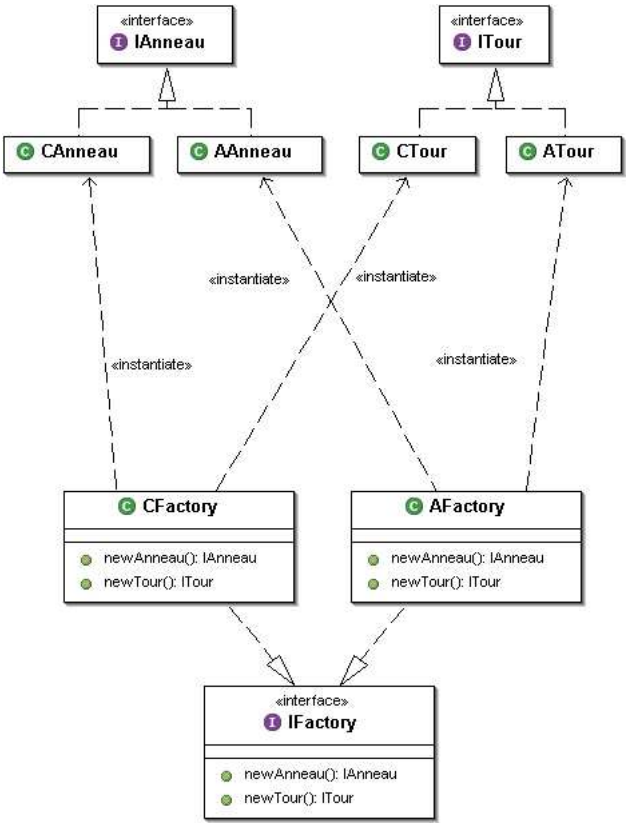

Figure 3. Patron de conception Abstract Factory

composant applicatif initial son composant proxy associé. Dans notre cas cela va donc permettre de passer facilement d'un composant applicatif à un composant de contrôle prévu pour une simple illustration, puis à un nouveau composant de contrôle prévu pour offrir une interaction à l'utilisateur (ici en mode Drag'n Drop). Ce seront toutes les autres classes clientes (autres que des contrôles) des classes d'un noyau initial, donc les classes qui feront également partie du noyau initial, qui doivent faire appel à cette seconde fabrique de composants, pour créer des composants de contrôle à la place des composants applicatifs et ainsi obtenir des applications graphiques interactives.

Nous ferons donc la différence entre ces deux fabriques de composants, et il devra être possible d'obtenir l'une ou l'autre selon les besoins des composants :

- les composants de contrôle auront à leur disposition une méthode permettant d'obtenir explicitement une fabrique de composants applicatifs, et également une méthode permettant d'obtenir explicitement une fabrique de composants de contrôle,

- les composants applicatifs utiliseront quant à eux une troisième méthode qui leur permettra d'obtenir une seule fabrique de composants. Celle-ci sera une fabrique de composants de contrôle lorsque les composants seront utilisés dans un contexte interactif, et une fabrique de composants applicatifs lorsque les composants seront utilisés dans leur contexte initial non interactif.

Ces méthodes utiliseront le patron de conception Singleton, illustré figure 4, pour d'une part assurer de ne créer qu'une fabrique de chaque type, et d'autre part pour faciliter l'accès à ces fabriques sans avoir à paramétrer par une fabrique chaque composant susceptible d'en avoir l'usage.

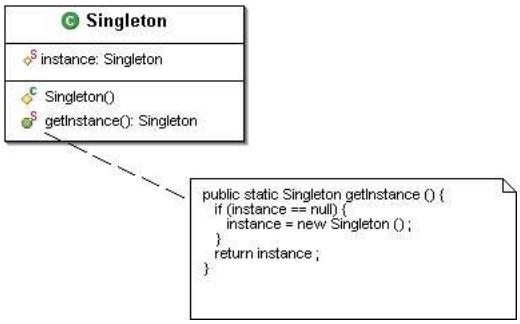

Figure 4. Patron de conception Singleton

Enfin, il en sera de même pour les composants de présentation : il sera ici encore fait usage d'interfaces et d'une fabrique de composants de présentation. Ceci se fera via une quatrième méthode permettant d'accéder cette fois à une fabrique de composants graphiques, destinée aux composants de contrôle pour créer leurs facettes présentation associées, mais sans être dépendants de l'API graphique effectivement utilisée.

On voit donc ici l'intérêt de l'usage des patrons de conceptionsuivants : Proxy (GoF207) (et Delegation), Template Method (GoF325) (ou plus simplement Interface) et Abstract Factory (GoF87) pour faciliter les évolutions d'une application à plusieurs niveaux :

- indépendance vis-à-vis des changements dans le noyau applicatif (évolutions du noyau applicatif),
- passage d'un mode non graphique à un mode graphique (passage au modèle PAC : remplacement des composants applicatifs par des composants de contrôle),
- passage d'un mode illustré à un mode interactif (évolutions du contrôle et de la présentation),
- indépendance vis-à-vis des API graphiques utilisées (évolutions de la présentation).

On voit également l'intérêt du patron de conception Singleton (GoF127) pour faciliter l'accès aux différentes fabriques de composants.

L'implémentation en java

Le package des interfaces des composants applicatifs

Ce package regroupe seulement quatre interfaces : IAnneau, ITour, IHanoi et IFactory, comme on peut le voir figure 5.

Les interfaces IAnneau et ITour vont décrire les fonctionnalités essentielles des anneaux et des tours que l'on utilis-

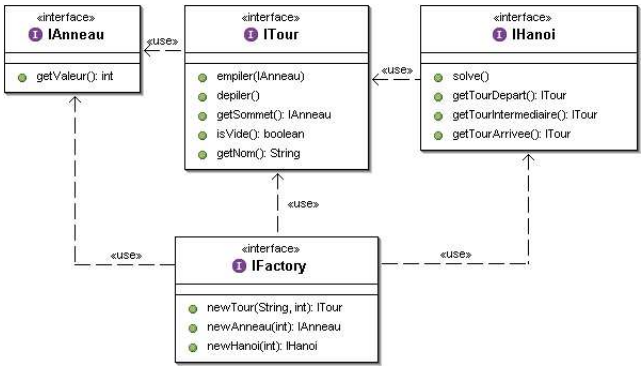

Figure 5. Modélisation UML du package interface de l'application

era pour réaliser l'application de résolution du problème des tours de Hanoï. L'interface IHanoi va décrire les fonctionnalités d'une classe de résolution du problème lui même. Enfin l'interface IFactory va décrire les méthodes à utiliser pour créer ces anneaux, tours et objet de résolution.

Le package applicatif

On y trouve simplement les quatre classes correspondant aux interfaces précédentes : AAnneau, ATour, AHanoi et AFactory, qui implémentent les interfaces décrites précédemment, comme on peut le voir figure 6. A noter qu'ici la classe AFactory est construite suivant le patron de conception Singleton de façon à ce que tous les objets applicatifs qui en ont besoin puissent facilement accéder à une instance unique de cette classe de création de composants.

L'utilisation des composants

La méthode de résolution "automatique" du problème des tours de Hanoï sera une méthode de la classe AHanoi :

```
protected void hanoi (ITour d,
                      ITour i,
                      ITour a,
                      int n) {
   if (n > 0) {
      hanoi (d, a, i, n - 1) ;
      IAnneau anneau = d.getSommet () ;
      d.depiler () ;
      a.empiler (anneau) ;
      hanoi (i, d, a, n - 1) ;
   }
}
```

Notons que cette procédure ne manipule que des interfaces et pourra donc être réutilisée par la suite pour manipuler toutes sortes de classes implémentant ces interfaces.

Une instance de la classe AHanoi pourra être construite ainsi :

```
public AHanoi (int n) {
   IFactory f = concreteFactory.
      ConcreteFactory.getFactory () ;
   this.n = n ;
   d = f.newTour ("d", n) ;
   i = f.newTour ("i", n) ;
```

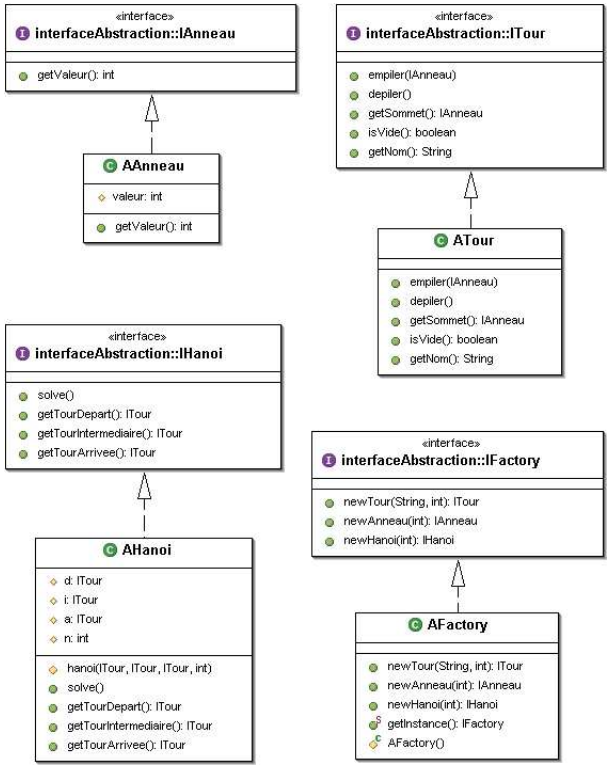

Figure 6. Modélisation UML du package applicatif

```
   a = f.newTour ("a", n) ;
   for (int ii = n ; ii > 0 ; ii --) {
      d.empiler (f.newAnneau (ii)) ;
   }
}
```

Notons ici l'appel à une fabrique de composants pour ne pas dépendre directement des composants utilisés effectivement. Il faudra que cette fabrique ait été initialisée correctement.

Moyennant cela, la méthode de résolution pourra être appelée de l'extérieur à l'aide de cette méthode qui utilise les tours déclarées dans la classe de résolution.

```
public void solve () {
   hanoi (d, i, a, n) ;
}
```

La classe MainHanoi qui fait la création effective de la fabrique de composants, qui la place en position d'être accessible, puis qui crée une application de résolution, se résume à cela :

```
public class MainHanoi {

   public static void main (String a []) {
      int n = 3 ;
      if (a.length > 0) {
          n = Integer.parseInt (a [0]) ;
      }
      concreteFactory.
         ConcreteFactory.setFactory (
            abstraction.
               AFactory.getInstance ()) ;
```

```
      IHanoi h = concreteFactory.
         ConcreteFactory.getFactory ().
            newHanoi (n) ;
      h.solve () ;
   }

}
```

On trouve bien ici l'obtention d'une fabrique de composants AFactory, placée dans une instance de ConcreteFactory afin de pouvoir être accessible par la suite. Il est aussitôt fait usage de cette AFactory via une demande explicite à l'instance de ConcreteFactory, afin de créer une instance d'une classe qui implémente l'interface IHanoi : ce sera ici une instance de AHanoi, à qui on pourra demander d'effectuer une résolution du problème.

Notons donc ici l'usage du patron de conception Singleton de façon à ne créer qu'une seule fabrique de composants ainsi que le stockage de cette fabrique dans un composant chargé de fournir la "bonne" fabrique de création des "bons" composants.

Le package des interfaces des composants présentation
Ce package regroupe quatre interfaces :

- les deux qui sont destinées à décrire les présentations de composants anneaux et tours qui permettront d'illustrer la résolution du problème des tours de Hanoï : IPAnneau et IPTour,
- une troisième, IPHanoi, destinée à fournir un réceptacle permettant d'accueillir ces présentations de tours et d'anneaux,
- une quatrième, IPFactory, qui décrit les méthodes permettant de créer ces présentations d'anneaux, de tours, et d'application de résolution.

L'interface IPAnneau décrit les services essentiels à la visualisation d'un anneau :

- setLocation : pour positionner précisément l'anneau,
- slipTo : pour faire glisser un anneau de sa position actuelle vers une position donnée,
- getHeight : pour obtenir la hauteur en pixels de l'anneau, servira à recalculer la position du sommet d'une tour.

L'interface IPTour décrit les services essentiels à la visualisation d'une tour :

- empiler : pour visualiser l'arrivée d'un nouvel anneau au sommet de la tour,
- depiler : pour visualiser le départ de l'anneau au sommet de la tour.

L'interface IPHanoi décrit le service essentiel à la visualisation de l'application de résolution du problème des tours de Hanoï :

- ajouter : pour accueillir la visualisation d'une tour.

Le package de présentation
Ce package regroupe quatre classes. Les deux premières, PAnneau et PTour, implémentent, à l'aide de l'API Swing, les interfaces IPAnneau et IPTour décrites dans le package interfacePresentation. La classe PHanoi sera ici un container permettant d'accueillir essentiellement des instances de PTour. Ces trois classes sont associées à la classe PFactory, qui implémente l'interface PFactory et qui permet de créer des instances de composants de présentation. Ceci est illustré figure 7.

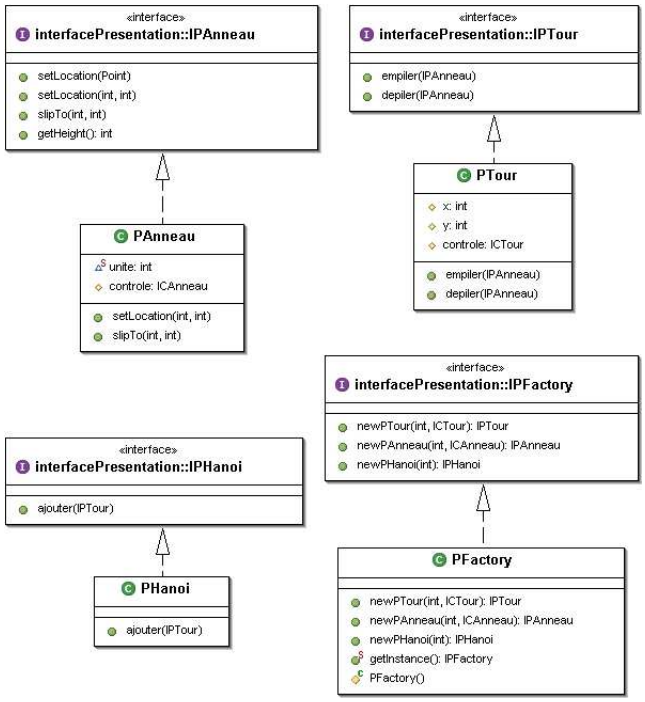

Figure 7. Diagramme de classes du package présentation

La classe PAnneau va implémenter l'interface IPAnneau et dériver de JPanel, c'est elle qui va décider :

- du nombre de pixels utilisés pour représenter la hauteur de l'anneau et l'unité de largeur de l'anneau,
- de la façon dont on positionne un anneau : setLocation agira ici au niveau du centre de l'anneau,
- de la façon dont on fait glisser un anneau d'une position à une autre : slipTo détermine le temps de glissement.

La classe PTour va implémenter l'interface IPTour et dériver de JPanel, c'est elle qui va décider :

- de la taille de la tour, relative au nombre maximum d'anneaux à accueillir,
- de la façon de visualiser l'arrivée d'un nouvel anneau au sommet de la tour : la méthode empiler fera glisser l'anneau jusqu'aux coordonnées du dessus de la tour exprimées dans le repère du container de la tour, elle accueillera ensuite (d'un point de vue Swing, via un add) la présentation de l'anneau, puis repositionnera cet anneau

relativement au système de coordonnées de la tour, et fera finalement glisser l'anneau à sa place au sommet de la tour, en coordonnées relatives à la tour.

- de la façon de visualiser le départ de l'anneau au sommet de la tour : la méthode depiler va faire glisser l'anneau au dessus de la tour en coordonnées relatives à la tour, elle va ensuite abandonner l'anneau (d'un point de vue Swing, via un remove), et le replacera finalement dans le container de la tour en le repositionnant au même endroit, avec cette fois de nouvelles coordonnées exprimées dans le système de coordonnées de ce container.

La classe PHanoi va implémenter l'interface IPHanoi et dériver de JFrame, c'est elle qui va décider de la façon d'ajouter une visualisation de tour, en s'assurant qu'elle soit placée, bien visible, à la suite des tours déjà présentes, en se redimensionnant de façon adéquate.

Le package des interfaces des composants contrôles

Dans un premier temps, pour simplement obtenir une visualisation graphique des anneaux et des tours, on a seulement besoin de définir des interface ICAnneau, ICTour et ICHanoi qui vont dériver respectivement des interfaces IAnneau, ITour et IHanoi, comme illustré figure 8. Ces interfaces ajouteront uniquement la déclaration d'une méthode permettant de récupérer le composant de présentation associé.

Il n'y a pas besoin ici de définir une interface pour une nouvelle fabrique de composants de contrôle : l'interface à utiliser sera impérativement celle du package interfaceAbstraction : IFactory.

Le package de contrôle

Il va être totalement indépendant de l'implémentation choisie pour les composants abstraction car il les connaîtra seulement par l'interface que chaque composant de contrôle va lui aussi implémenter. Chaque composant de contrôle devra créer son composant applicatif associé en utilisant la fabrique de composant applicatifs correspondante. Le choix de cette fabrique est effectué dans le programme principal.

De même, ce package va rester totalement indépendant de l'implémentation choisie pour les composants présentation, car ici encore il les connaîtra seulement par une interface, ces composants de présentation seront également créés en utilisant la fabrique de composant présentation correspondante. Encore une fois le choix de cette fabrique est effectué dans le programme principal.

Ces composants de contrôle pourront donc être associés à différentes implémentations de composants applicatifs et/ou de présentation.

Les constructeurs des composants contrôle seront du même type que le constructeur de CTour :

```
public CTour (String nom, int nbAnneauxMax) {
   abstraction = ConcreteFactory.
      getAFactory ().
```

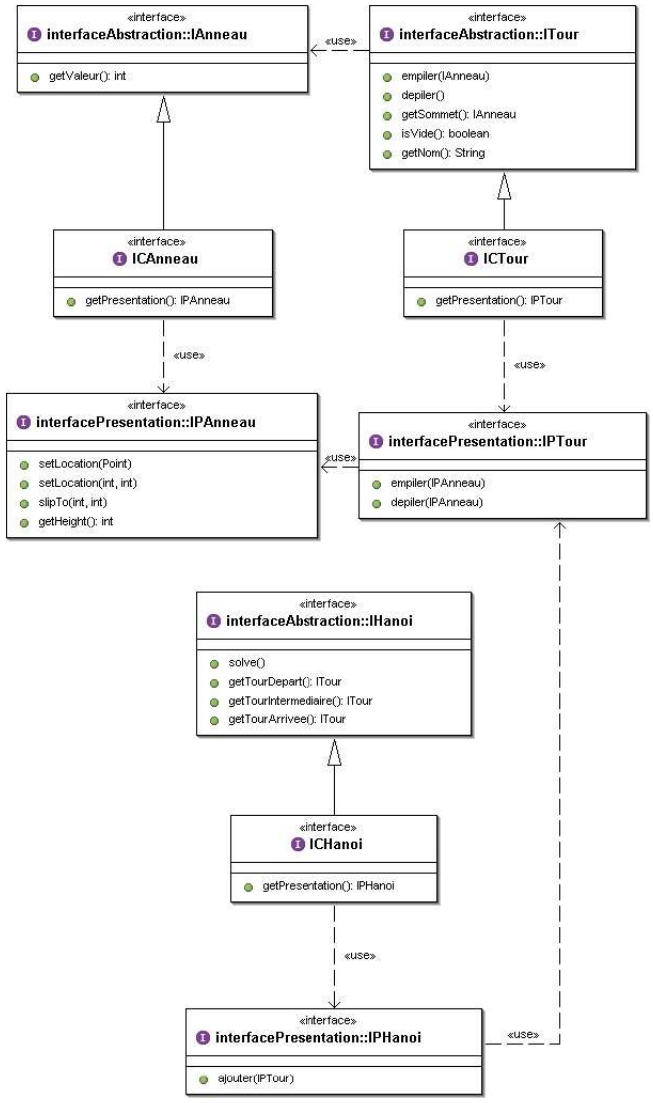

Figure 8. Modélisation UML du package interface contrôle

```
            newTour (nom, nbAnneauxMax) ;
      presentation = ConcreteFactory.
         getPFactory ().
            newPTour (nbAnneauxMax, this) ;
   }
```

On peut voir ici la façon dont les facettes abstraction et présentation sont créées, grâce à l'usage des fabriques de composants adéquates.

Les autres méthodes des composants contrôle seront le plus souvent semblables à cette méthode de la classe CTour :

```
public void empiler (IAnneau aa) {
   abstraction.empiler (aa) ;
   presentation.empiler (
      ((ICAnneau)aa).
         getPresentation ()) ;
}
```

On voit ici dans le cas de cette méthode empiler que le traitement effectif est délégué au composant applicatif, et qu'ensuite on maintient la cohérence entre partie applicative et partie présentation en empilant le composant de présentation de l'anneau sur le composant de présentation de la tour.

Ceci peut paraître être une entorse au modèle PAC car ce dernier stipule que les composants de présentation ne doivent pas communiquer, mais cet ajout est indispensable, et ici l'initiative de l'ajout est laissée au composant contrôle, ce qui est conforme à la logique du modèle PAC.

Notons au passage que pour pouvoir faire cela, il est nécessaire d'opérer un transtypage de l'anneau à empiler en un composant interface de contrôle d'anneau ICAnneau afin de pouvoir récupérer sa présentation associée.

Enfin, notons que le fait de faire glisser en douceur les anneaux d'une position à une autre (plutôt que de disparaître soudainement d'un endroit pour réapparaître tout aussi soudainement ailleurs) est laissé à la responsabilité des composants de présentation : les composants de contrôle se concentrent seulement sur l'essentiel, l'ordonnancement des actions.

Les diagrammes de classes de ces composants contrôles sont présentés figures 9, 10 et 11.

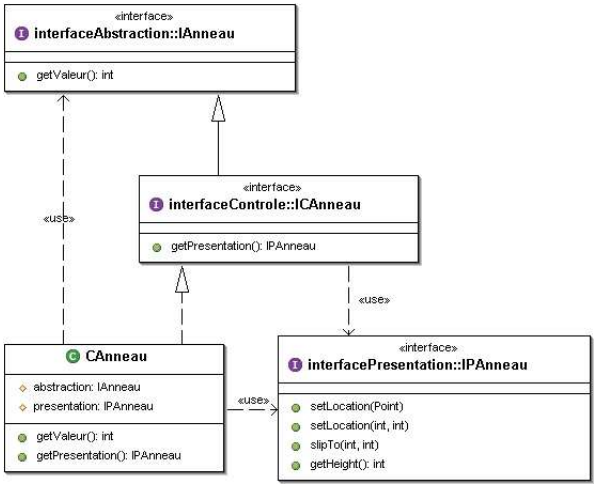

Figure 9. Modélisation UML du contrôle d'anneau

Par ailleurs, il serait possible de faire des contrôles complémentaires au niveau des méthodes des composants contrôle, pour prévoir une utilisation plus générale des ces composants. Dans le cas de la méthode empiler de la classe CTour, il faudrait soit vérifier qu'un anneau est bien empilable avant de l'empiler effectivement (c'est-à-dire tester une précondition), soit vérifier que l'anneau a bien été empilé avant de répercuter l'empilement au niveau de du composant présentation (c'est-à-dire tester une postcondition), soit encore faire les deux vérifications.

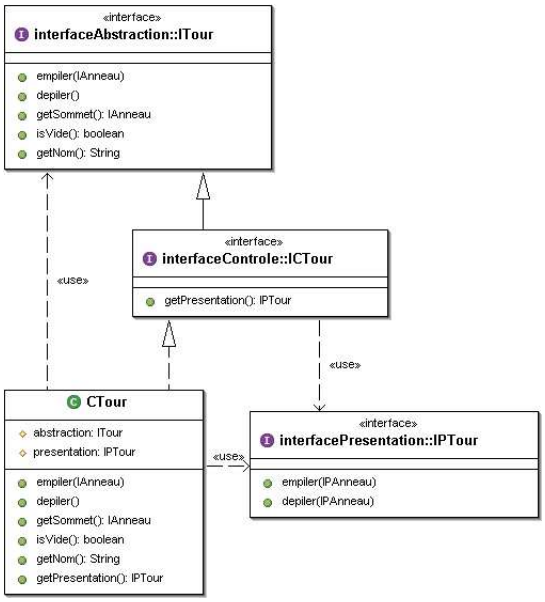

Figure 10. Modélisation UML du contrôle de tour

À condition qu'elle existe dans le composant abstraction associé, l'utilisation d'une précondition se ferait de la façon suivante :

```
public void empiler (IAnneau aa) {
   if (abstraction.isEmpilable (aa)) {
      abstraction.empiler (aa) ;
      presentation.empiler (
         ((ICAnneau)aa).
            getPresentation ()) ;
   }
}
```

Si une telle précondition n'existe pas, ce peut être au contrôle de la définir, jouant ainsi son rôle de régulation de l'accès à son composant abstraction associé.

L'autre solution à base de postcondition, toujours à condition qu'une telle postcondition existe dans le composant abstraction associé, aurait la forme suivante :

```
public void empiler (IAnneau aa) {
   abstraction.empiler (aa) ;
   if (abstraction.vientDEtreEmpile (aa)) {
      presentation.empiler (
         ((ICAnneau)aa).
            getPresentation ()) ;
   }
}
```

Dans ce dernier cas, il est fort probable que ce type de postcondition n'existe pas, c'est alors encore au contrôle de définir cette notion, en vérifiant que l'élément que l'on essaie d'empiler se trouve effectivement au sommet de la tour après avoir délégué l'empilement au composant abstraction associé. Cela peut se faire ici de cette façon :

```
public void empiler (IAnneau aa) {
   abstraction.empiler (aa) ;
   if (abstraction.getSommet () == aa) {
```

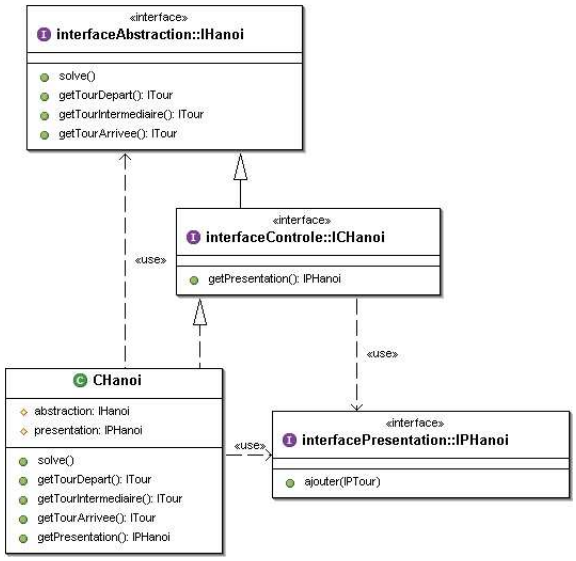

Figure 11. Modélisation UML du contrôle de résolution

```
        presentation.empiler (
           ((ICAnneau)aa).
              getPresentation ()) ;
     }
  }
```

Utilisation des composants contrôles

Le programme principal devient donc responsable du choix des différentes fabriques de composants qu'il doit mettre à la disposition des composants qui en auront besoin : les composants de contrôle et la classe Hanoi, puisqu'elle devra créer des tours et des anneaux :

- dans le cas de la classe Hanoi, c'est pour obtenir une fabrique qui va créer des composants de contrôle à la place des composants applicatifs initiaux : des ITours et des IAnneaux qui seront maintenant en pratique des instances de CTour et CAnneau,

- dans le cas des composants de contrôle c'est soit pour créer les bonnes instances de composants applicatifs et de présentation qui leur seront associés, soit pour créer d'autres composants de contrôle.

Pour une simple démonstration, on devra donc déterminer les fabriques de composants à utiliser : une fabrique de composants applicatifs (à l'usage des composants de contrôle pour créer leurs abstractions associés), une fabrique de composants de contrôle (à l'usage de la classe Hanoi car c'est elle qui crée tous les composants de contrôle, mais ce serait aussi à la disposition des composants de contrôle qui souhaiteraient créer d'autres composants contrôles) et une fabrique de composants de présentation (à l'usage des composants de contrôle pour créer leurs présentations associées).

La classe permettant d'accéder ensuite à ces différentes fabriques est ici la classe ConcreteFactory, elle propose quatre méthodes pour obtenir des fabriques de composants :

- getAFactory,
- getCFactory,
- getPFactory,
- getFactory.

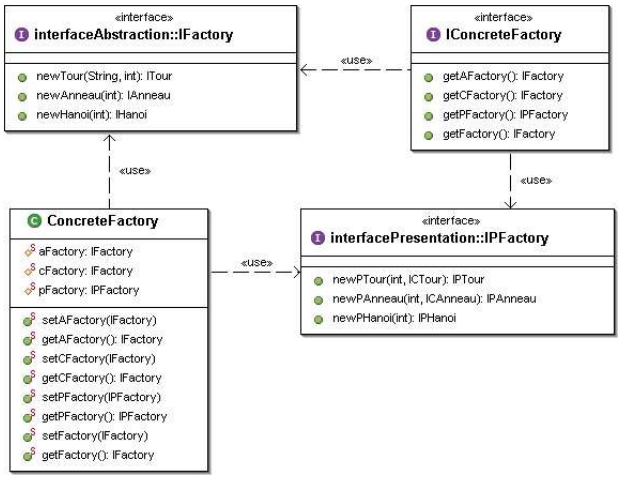

Figure 12. Modélisation UML de la classe ConcreteFactory

Les trois premières méthodes sont destinées à être utilisées par des composants de contrôle, elle vont permettre d'obtenir respectivement des instances de composants applicatifs, de composants de contrôle, et de composants de présentation. La quatrième, c'est-à-dire la méthode getFactory, est quant à elle destinée aux composants applicatifs initiaux : elle retourne en priorité la CFactory ou par défaut la AFactory si aucune CFactory n'a été désignée. C'est donc cette méthode qui doit être utilisée par les composants applicatifs (qui doivent pouvoir être utilisés en dehors d'un implémentation PAC) et par la classe Hanoi pour créer les différents composants nécessaires à la résolution du problème. 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.

Voici les choix à réaliser pour obtenir une illustration graphique de la résolution des tours de Hanoï :

```
import interfaceAbstraction.IHanoi;

public class MainHanoiPAC {

   public static void main (String args []) {
      int n = 5 ;
      if (args.length > 0) {
         n = Integer.parseInt (args [0]) ;
      }
      concreteFactory.ConcreteFactory.
         setAFactory (abstraction.AFactory.
                        getInstance ()) ;
      concreteFactory.ConcreteFactory.
         setCFactory (controle.CFactory.
                        getInstance ()) ;
      concreteFactory.ConcreteFactory.
```

```
        setPFactory (presentation.PFactory.
                        getInstance ()) ;
      IHanoi h = concreteFactory.
         ConcreteFactory.getFactory ().
            newHanoi (n) ;
      h.solve () ;
   }

}
```

Conclusions à propos de cette première méthode

Avec cette première méthode, il y a beaucoup de classes à créer pour réaliser une application visualisable à partir d'un noyau applicatif non graphique. Par contre, cette méthode induit des dépendances minimales entre les différents composants, en particulier grâce au mélange des patrons de conception Proxy (avec Delegation) et Template Method (ou Interface), ainsi bien sûr qu'à l'utilisation du patron Abstract Factory. Le patron de conception Singleton, qui garantit l'usage d'une seule fabrique par type de composants, facilite par ailleurs l'accès aux différentes fabriques à partir des composants qui en ont besoin.

SECONDE SOLUTION : PAC ET PROXY CONTRÔLE PAR HÉRITAGE

Le principe

L'idée est ici de voir s'il est possible d'être plus efficace au niveau de l'implémentation du patron de conception Proxy. Au lieu de déléguer certains traitements à un composant applicatif, chaque composant contrôle qui sert de proxy à un composant applicatif va directement hériter de ce composant. Nous utilisons donc tout simplement le mécanisme d'héritage au lieu du patron de conception Delegation, comme on peut le voir figure 13.

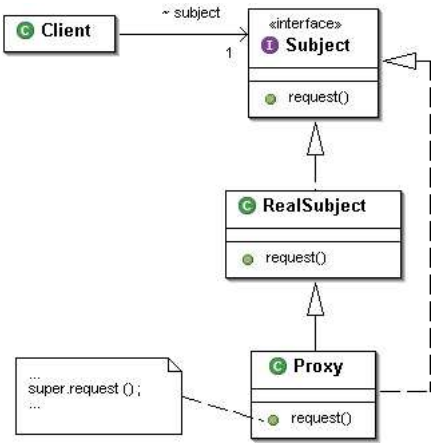

Figure 13. Patron de conception Proxy avec héritage

Comme les composants de contrôle vont maintenant hériter des classes de l'abstraction, on va avoir une plus grande dépendance du contrôle vis à vis de l'application.

Par contre, il y aura moins de code à écrire car toutes les méthodes des composants de contrôle qui n'étaient que de simples relais vers des méthodes des composants applicatifs pourront être réutilisées telles-quelles, sans être redéfinies au niveau des composants de contrôle.

L'implémentation en java

On utilisera les trois mêmes packages d'interfaces de composants, ainsi que le package applicatif et le package de présentation, seul le package d'implémentation des composants de contrôle va changer.

Le nouveau package de contrôle

Il va maintenant dépendre directement de l'implémentation choisie pour les composants abstraction car les composants de contrôle liés à des objets applicatifs à visualiser vont dériver de ces objets, comme illustré figure 14.

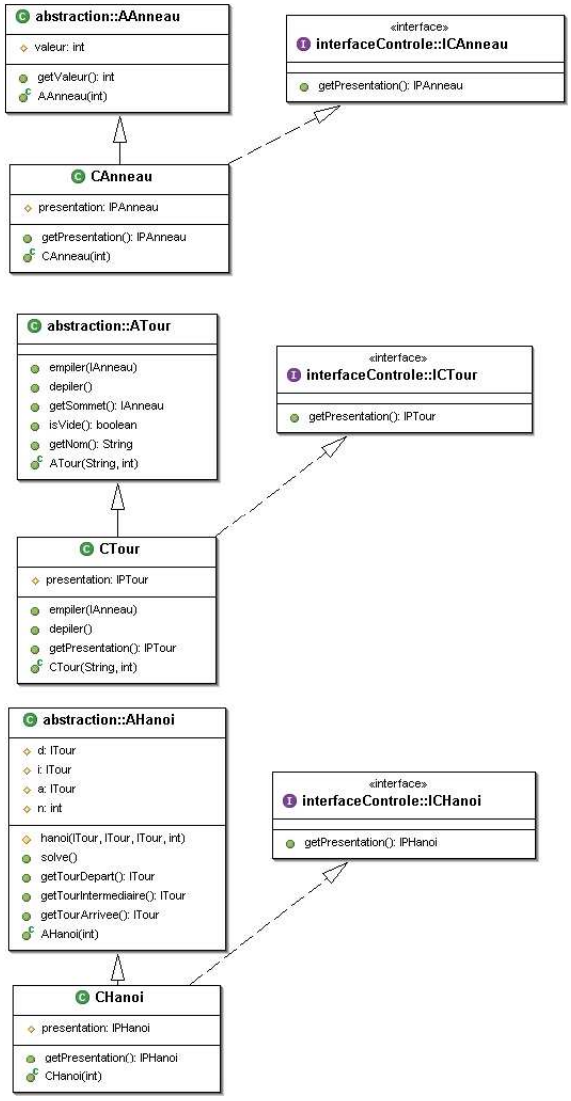

Figure 14. Modélisation UML du package controle-Heritage

Chaque composant de contrôle va donc devoir importer sa classe ancêtre (par exemple ATour pour le contrôle CTour), et son constructeur devra invoquer le constructeur de son

ancêtre puis créer un composant de présentation. Le constructeur de CTour est tout à fait représentatif d'un tel constructeur :

```
public CTour (String nom, int nbAnneauxMax) {
   super (nom, nbAnneauxMax) ;
   presentation = ConcreteFactory.
      getPFactory ().
         newPTour (nbAnneauxMax, this) ;
}
```

L'usage d'une fabrique de composants adéquate ne permet maintenant plus que d'être indépendant vis à vis des classes d'implémentation de la présentation.

Les autres méthodes des composants contrôle feront typiquement l'appel à la méthode correspondante du composant ancêtre, puis maintiendront la cohérence avec le composant de présentation. Ces méthodes seront le plus souvent semblables à cette méthode de la classe CTour :

```
public void empiler (IAnneau aa) {
   super.empiler (aa) ;
   presentation.empiler (
      ((ICAnneau)aa).getPresentation ()) ;
}
```

Comme c'était le cas avec la première méthode d'implémentation présentée, il serait ici encore possible de faire des vérifications complémentaires au niveau des méthodes des composants contrôle, pour prévoir une utilisation plus générale de ces composants.

Conclusions à propos de cette seconde méthode

Il y a toujours autant de classes à créer, mais elles sont un peu plus simples que dans le cas précédent, car pour beaucoup de méthodes des composants contrôle on peut réutiliser telles-quelles les méthodes du composant applicatif dont on hérite. Par contre on induit des dépendances fortes entre les composants contrôle et les composants applicatifs dont ils dérivent : chaque package de contrôle ainsi défini dépend directement de l'implémentation de l'abstraction à contrôler : il faudra donc un nouveau package de contrôle pour chaque nouvelle implémentation du noyau applicatif.

Cette méthode peut également être utilisée en cas d'absence de définition d'interfaces pour les composants applicatifs et de contrôle : le fait que les composants de contrôle héritent des composants applicatifs permet de substituer ces composants applicatifs par leurs composants contrôles héritiers. Ce peut donc être une solution permettant un déploiement du modèle PAC moins pérenne qu'avec la première méthode, mais sensiblement plus rapide.

TROISIÈME SOLUTION : PAC-AMODEUS, PROXY ANF PAR HÉRITAGE ET PROXY CONTRÔLE PAR DÉLÉGATION

Le principe

Enfin, l'idée est ici de voir ce qu'une implémentation différente, en faisant usage du patron de conception Observer (GoF293) pour l'adaptation des composants applicatifs sous forme de composants, peut apporter au niveau de l'indépendance entre noyau fonctionnel et IHM.

Lors d'un changement d'état susceptible d'être visualisé graphiquement, un composant applicatif devrait prévenir automatiquement les composants de contrôle qui peuvent être intéressés par ce changement d'état. Il faut donc mettre en place un mécanisme de souscription d'abonnement à des changements et de diffusion des changements auprès des abonnés, en suivant le patron de conception Observer illustré figure 15.

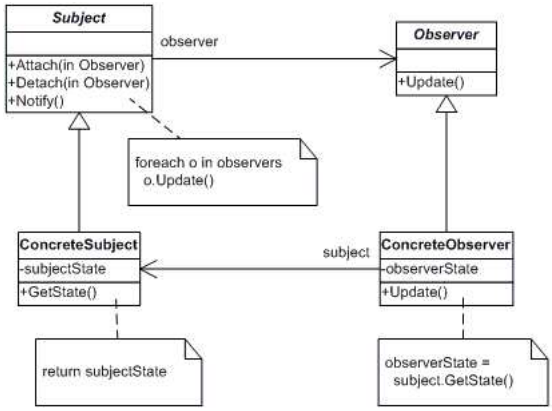

Figure 15. Patron de conception Observer

Comme les composants applicatifs ne savent a priori pas faire ceci, conformément au modèle PAC-Amodeus cela va être à des composants d'un Adaptateur de Noyau Fonctionnel (ANF) d'adapter les composants applicatifs pour leur faire remplir ces nouvelles fonctionnalités : permettre de souscrire un abonnement à des changements et diffuser effectivement ces changements aux abonnés. Ces composants ANF vont donc en quelque sorte être des proxys (au sens de notre seconde méthode) des composants applicatifs.

Les composants de contrôle devront alors être capables de souscrire un abonnement auprès du composant de l'ANF qui leur sera associé, et implémenter les méthodes leur permettant de recevoir les changements d'états auxquels ils auront souscrit. Ici encore ces composants de contrôle seront des proxys (au sens de notre première méthode) des composants ANF.

L'implémentation en java

On utilisera ici encore les trois mêmes packages d'interfaces de composants, ainsi que le package applicatif et le package de présentation. Le package d'implémentation des composants de contrôle va être différent, et on va créer un nouveau package ANF pour adapter les composants applicatifs qui en ont besoin. On créera également un nouveau package interfaceANF pour y décrire les nouvelles fonctionnalités qui devront être proposées par les composants du package ANF.

Les packages d'adaptation du noyau fonctionnel

Une seule classe du noyau applicatif devra être adaptée pour propager ses changements d'états : la classe tour. Cela va se faire à l'aide de l'interface IANFTour (illustrée figure 16) et de la classe ANFTour (illustrée figure 17). En effet, afin

de pouvoir diffuser les arrivées et les départs d'anneaux vers d'éventuels observateurs ayant souscrit un abonnement à ces changements, il va falloir effectuer deux types d'ajouts :

- la possibilité de souscrire à la diffusion de ces événements,
- la diffusion effective des événements à chaque changement d'état suite à l'empilement ou au dépilement d'un anneau.

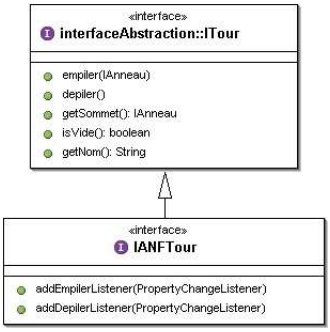

Figure 16. Modélisation UML du package interfaceANF

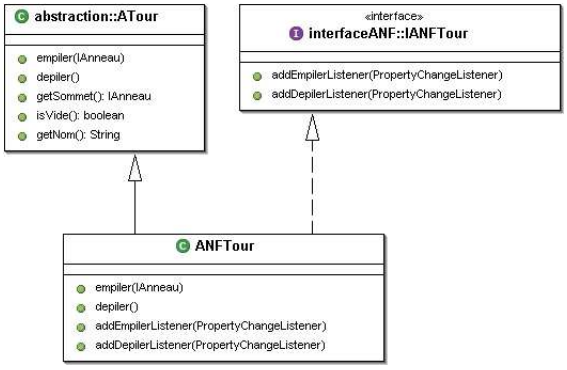

Figure 17. Modélisation UML du package ANF

Il faut bien entendu également fournir ici une fabrique de composants, qui créera une instance d'ANFTour lorsqu'on lui demandera de créer une tour, et qui s'appuiera sur le package applicatif initial pour créer des instances de AAnneau et de AHanoi (classes inchangées du noyau fonctionnel) lorsqu'on lui demandera de créer des anneaux ou bien des objets de résolution du problème des tours de Hanoï.

Le package de contrôle

Comme pour le package ANF associé, on n'a ici à redéfinir qu'une seule classe : la classe de contrôle de tour, CTour, dont le modèle UML est présenté figure 18. Bien entendu on doit aussi fournir une fabrique de composants qui va s'appuyer sur les classes du package de contrôle initial pour créer les objets qui n'ont pas besoin d'être en relation avec les objets de l'adaptateur de noyau fonctionnel : les instances d'anneau et d'objet de résolution.

Le constructeur de la nouvelle classe CTour va être un peu plus complexe que les précédents car il va devoir s'abonner aux nouveaux services décrits par l'interface IANFTour :

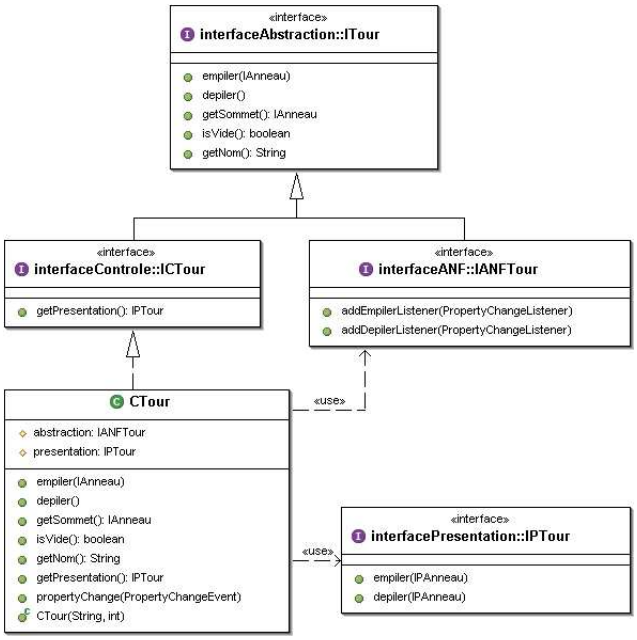

Figure 18. Modélisation UML du package controleANF

```
public CTour (String nom, int nbAnneauxMax) {
   abstraction = (IANFTour)ConcreteFactory.
      getAFactory ().
         newTour (nom, nbAnneauxMax) ;
   presentation = ConcreteFactory.
      getPFactory ().
         newPTour (nbAnneauxMax, this) ;
   // abonnement aux evenements propages
   // par l'abstraction associee
   abstraction.addDepilerListener (this) ;
   abstraction.addEmpilerListener (this) ;
}
```

Cela se fait à l'aide des nouvelles méthodes décrites dans l'interface IANFTour : addDepilerListener et addEmpilerListener. Cela n'est possible que parce que cette classe CTour implémente également une nouvelle interface (ici l'interface PropertyChangeListener) qui lui permettra d'être prévenue lors des changements d'états de la tour associée via un appel à la méthode propertyChange.

L'avantage de ce principe est que les méthodes de type empiler et dépiler sont maintenant de simples relais vers le composant applicatif associé (en pratique, une instance de ANFTour) :

```
public void empiler (IAnneau aa) {
   abstraction.empiler (aa) ;
}

public void depiler () {
   abstraction.depiler () ;
}
```

Enfin, il faut implémenter la méthode propertyChange, appelée lors de la diffusion des changements d'états de l'abstraction. Cette diffusion est réalisée sur le modèle des

java beans, en s'appuyant donc sur les interfaces PropertyChangeEvent et PropertyChangeListener :

```
public void propertyChange (
   PropertyChangeEvent evt) {
   if (evt.getPropertyName ().
      equals ("empiler")) {
      ICAnneau a = (ICAnneau)evt.
         getNewValue () ;
      presentation.empiler (
         a.getPresentation ()) ;
   } else if (evt.getPropertyName ().
      equals ("depiler")) {
      ICAnneau a = (ICAnneau)evt.
         getNewValue () ;
      presentation.depiler (
         a.getPresentation ()) ;
   }
}
```

L'implémentation de la diffusion des changements d'états peut se faire de différents façons, en se basant comme ici sur des services existants (les PropertyChangeSupport des java beans) ou bien en créant soi-même de nouveaux protocoles de communication entre objets applicatifs qui diffusent leurs changements et objets de contrôle qui reçoivent ces changements.

Conclusions à propos de cette troisième méthode

Bien que l'implémentation de cette méthode soit assez différente des deux précédentes pour la seule classe concernée, CTour, cette méthode semble quand même équivalente à la première en termes de fonctionnalités : on garde la même indépendance du contrôle vis à vis du noyau applicatif.

Par contre, cette méthode alourdit quelque peu le codage à cause des protocoles à mettre en place entre composants applicatifs et composants de contrôle.

Les ajouts de code restent ici limités car d'une part une seule classe est concernée et d'autre part on peut se reposer sur une implémentation du patron Observer qui est celle des java beans.

Enfin, par rapport aux deux premières méthodes présentées, cette méthode ajoute quelques traitements supplémentaires à chaque modification visuelle à apporter à un composant interactif. En effet, de part son aspect très générique, qui permet de mettre à jour de la même façon n'importe quelle évolution grahique, il y a des opérations ou vérifications supplémentaires à réaliser à chaque fois. Ici ce sont des tests à effectuer afin de savoir quelle action réaliser ; et dans le cas "canonique" de ce patron de conception, ce sont des invocations de méthodes supplémentaires de la part des composants observateurs vers les composants observés. Ceci n'est pas pénalisant dans le cas classique d'une IHM pilotée principalement par l'utilisateur, par contre cela peut amener des pertes d'efficacité dans le cas d'applications pilotées plutôt par un grand nombre d'éléments qui évoluent et qu'il faut visualiser en temps réel, comme c'est le cas dans des visualisations d'applications de simulation ou encore des applications de réalité virtuelle.

PASSAGE À LA MANIPULATION DIRECTE

Le principe

Pour les trois cas vus précédemment, on peut passer d'un mode démonstration à un mode interaction par manipulation directe de la même façon, en créant de nouvelles classes de contrôle qui dérivent des classes qui ont été présentées, et en en faisant de même pour les classes de présentation. Nous présenterons donc ici comment réaliser cette extension dans le cas où l'on utilise la première méthode présentée : utilisation du modèle PAC avec composants de contrôle qui sont des proxys des composants applicatifs et qui utilisent effectivement une instance de composant applicatif selon le patron Delegation.

Conformément aux recommandations du modèle PAC, nous allons essayer de rendre les composants de contrôle les plus indépendants possibles des couches de présentation utilisées pour obtenir la manipulation directe (ici les packages dnd et datatransfer de java). Ceci va nous conduire à définir un protocole précis entre les composants de contrôle et les composants de présentation pour gérer finement cette manipulation directe et confiner dans les composants de présentation tous les détails relatifs au mécanisme de drag'n drop proposé par java.

Nouveaux besoins au niveau des composants de contrôle

On va devoir décrire les nouveaux services nécessaires à une manipulation directe des anneaux et des tours dans le cadre de la résolution manuelle du problème des tours de Hanoï. En pratique, rien de nouveau pour l'objet de résolution du problème, mais de nouveaux besoins au niveau :

- des anneaux : il faut pouvoir les rendre sélectionnables ou pas, et pouvoir les rendre sélectionnés ou pas selon qu'un utilisateur les manipule ou pas,
- des tours : il faut pouvoir savoir si un anneau est empilable ou pas sur une tour, et pouvoir savoir si un anneau est dépilable ou pas d'une tour.

Ceci est nécessaire pour des raisons fonctionnelles mais aussi afin de pouvoir offrir une rétroaction maximale (sémantique) à un utilisateur lors des déplacements des anneaux d'une tour à une autre.

Il faudra aussi être capable de prendre en compte le début et la fin d'une manipulation d'anneau de type Drag'n Drop d'une tour sur une autre.

Nouveaux besoins au niveau des composants de présentation

On va retrouver ici des besoins de services similaires à ceux des composants de contrôle :

- pour les anneaux : il faut pouvoir visualiser qu'un anneau est sélectionnable (ou non), et pouvoir le visualiser sélectionné ou non selon qu'un utilisateur le manipule ou pas,
- pour les tours : il faut pouvoir visualiser qu'un anneau est empilable ou pas sur une tour, et pouvoir visualiser qu'un anneau est dépilable ou pas d'une tour.

On va avoir également besoin de services qui permettront d'obtenir des composants sur lesquels il sera possible d'agir directement, et dont on pourra changer l'aspect en fonction de l'état du composant applicatif associé. Dans le cas des anneaux, il s'agit simplement de pouvoir obtenir le composant de contrôle associé, pour lui indiquer qu'il a un début de drag'n drop à traiter.

Dans le cas des tours, il s'agit de pouvoir visualiser :

- qu'un anneau sort de la zone correspondant à la tour associée,
- que le début de drag'n drop reconnu sur la tour associée est valide ou non,
- que le drag'n drop qui s'est terminé sur la tour associée est valide ou non.

L'implémentation en java

Nouvelles interfaces de contrôle

Comme indiqué précédemment et illustré figure 19, nous devrons proposer deux nouvelles interfaces dans le package interfaceControle, conformément aux besoins exprimés au niveau des composants de contrôle : ICAnneauInter et ICTourInter.

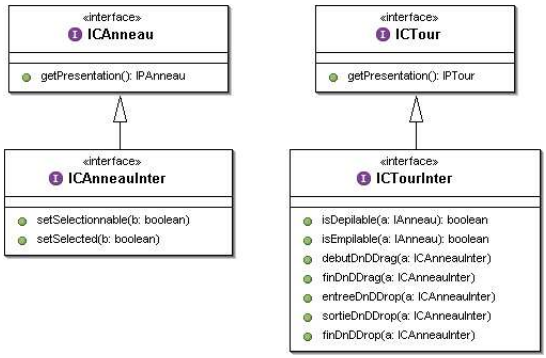

Figure 19. Modélisation UML du package interfaceControle incluant les classes d'interaction

L'interface ICAnneauInter décrit des services qui permettent :

- de rendre sélectionnable ou non un anneau selon que l'anneau se trouve ou non au sommet d'une tour,
- de rendre sélectionné ou non un anneau selon qu'un utilisateur le manipule ou pas.

L'interface ICTourInter décrit des services qui permettent :

- de dire si l'empilement ou le dépilement d'un anneau est possible ou non,
- de traiter l'arrivée ou le départ d'un composant anneau suite à une manipulation directe :
  - debutDnDDrag : quand on a reconnu, côté départ (forcément !), le début d'une manipulation d'anneau,
  - finDnDDrag : quand on a reconnu, côté départ, la fin d'une manipulation d'anneau,
  - entreeDnDDrop : quand on a reconnu, côté arrivée, qu'un anneau vient d'entrer dans la tour,
  - sortieDnDDrop : quand on a reconnu, côté arrivée, qu'un anneau vient de sortir de la tour,
  - finDnDDrop : quand on a reconnu, côté arrivée, la fin d'une manipulation d'anneau.

  En effet, une tour est à la fois un réceptacle de départ et un réceptacle d'arrivée pour une manipulation directe.

Nouvelles classes de contrôle

On n'aura donc que deux nouvelles classes à implémenter dans le package controle : CAnneauInter et CTourInter, comme illustré figure 20 et 21, en plus de la fabrique de composants permettant de créer des composants de contrôle destinés à des interactions en manipulation directe (qui créera par ailleurs des instance de CHanoi).

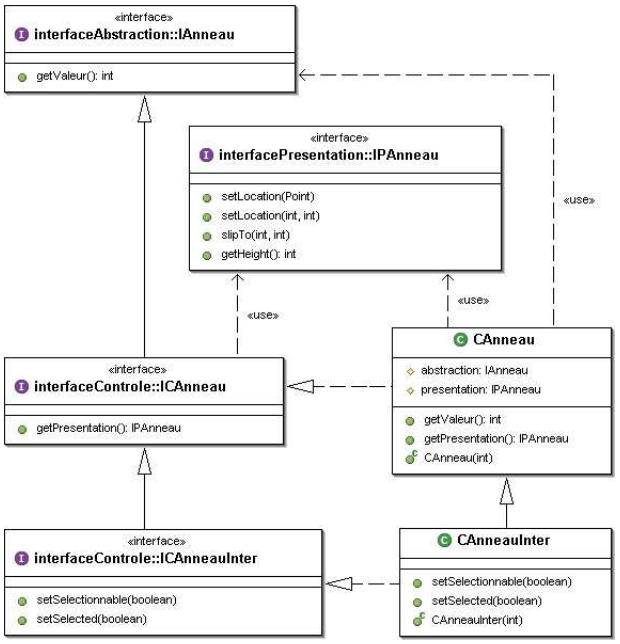

Figure 20. Modélisation UML de la classe de contrôle d'anneau interactif

Conformément à l'interface ICAnneauInter qu'elle implémente, la classe CAnneauInter permet de rendre sélectionnable ou non un anneau et de rendre sélectionné ou non un anneau, et ce en visualisant à chaque fois de façon adéquate la présentation d'anneau associée, et en lui demandant d'être capable le cas échéant de traiter des demandes de manipulation directe en provenance de l'utilisateur.

De même, la classe CTourInter va permettre de dire si l'empilement ou le dépilement d'un anneau est possible, elle pourra aussi traiter l'arrivée ou le départ d'un composant anneau suite à une manipulation directe, le tout en appelant les méthodes adéquates déclarées dans l'interface ICAnneauInter selon qu'un anneau est empilé ou dépilé, et entre ou sort de la tour lors d'une manipulation directe.

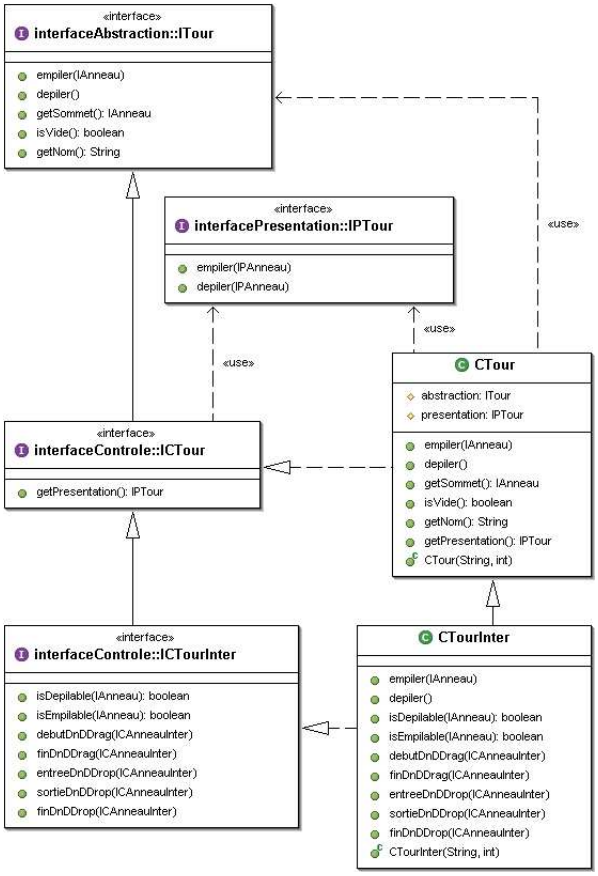

Figure 21. Modélisation UML de la classe de contrôle de tour interactive

Nouvelles interfaces de présentation

Le package des interfaces de présentation regroupe maintenant deux interfaces supplémentaires, IPAnneauInter et IPTourInter, dérivées des interfaces IPAnneau et IPTour, comme illustré figure 22. Ces nouvelles interfaces décrivent des services qui permettront d'obtenir des composants sur lesquels il sera possible d'agir directement, et dont on pourra changer l'aspect en fonction de l'état du composant applicatif associé.

L'interface IPAnneauInter décrit des services qui permettent :

- de visualiser sélectionnable ou non un anneau, et de le rendre désignable ou pas par l'utilisateur,
- de visualiser sélectionné ou non un anneau,
- d'obtenir le composant de contrôle associé.

L'interface IPTourInter décrit des services qui permettent :

- de visualiser si l'empilement ou le dépilement d'un anneau est possible ou non,
- de visualiser de façon adéquate l'arrivée ou le départ d'un composant anneau suite à une manipulation directe.

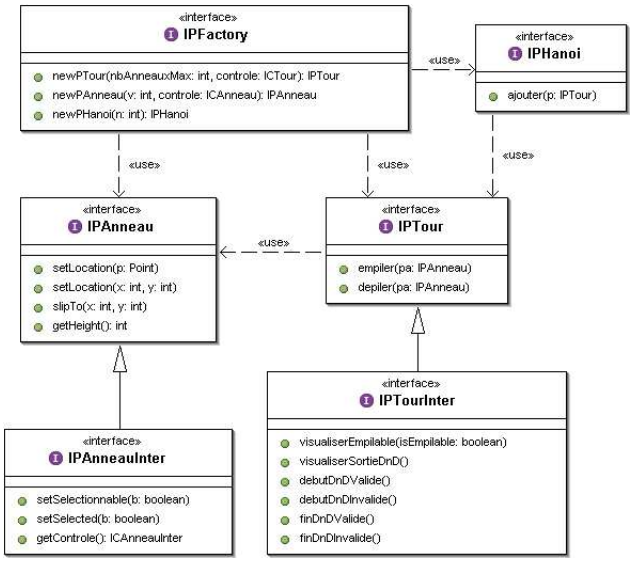

Figure 22. Modélisation UML du package interfacePresentation incluant les classes d'interaction

Nouvelles classes de présentation

On y trouve donc les classes PAnneauInter et PTourInter qui implémentent respectivement les interfaces IPAnneauInter et IPTourInter, et qui dérivent respectivement des classes PAnneau et PTour. Ces deux classes sont associées à la classe PFactoryInter (qui ici encore implémente l'interface PFactory), qui permet de même d'en créer des instances. Cette fabrique de composants se contente par ailleurs de créer des instances de PHanoi car il n'y a pas de nouvelle classe PHanoiInter.

La classe PAnneauInter, illustrée figure 23, hérite des services fournis par la classe PAnneau et implémente en plus les services décrits par l'interface IPAnneauInter qui permettent de visualiser de façons différentes un anneau selon son état sélectionnable ou non, sélectionné ou non, en changeant le curseur de la souris et en changeant la couleur de l'anneau. Elle implémente également d'autres méthodes décrites par l'interface Transferable de façon à pouvoir être manipulée par drag'n drop, qui ont essentiellement pour but d'obtenir la donnée à transférer lors de la manipulation directe : ici ce sera l'instance de présentation elle-même.

De même, La classe PTourInter, illustrée figure 24, hérite des services fournis par la classe PTour et implémente en plus les services décrits par l'interface IPTourInter. Elle va tout d'abord devoir redéfinir les services d'empilement et de dépilement : lors d'un empilement ou d'un dépilement, un anneau ne devra plus glisser jusqu'au sommet de la tour, mais devra aller directement à son nouvel emplacement. Cette classe va également devoir créer des classes (et les instancier) permettant de gérer le drag'n drop à l'aide de Swing, c'est-à-dire :

- un DragGestureListener : pour reconnaître un début de drag'n drop,

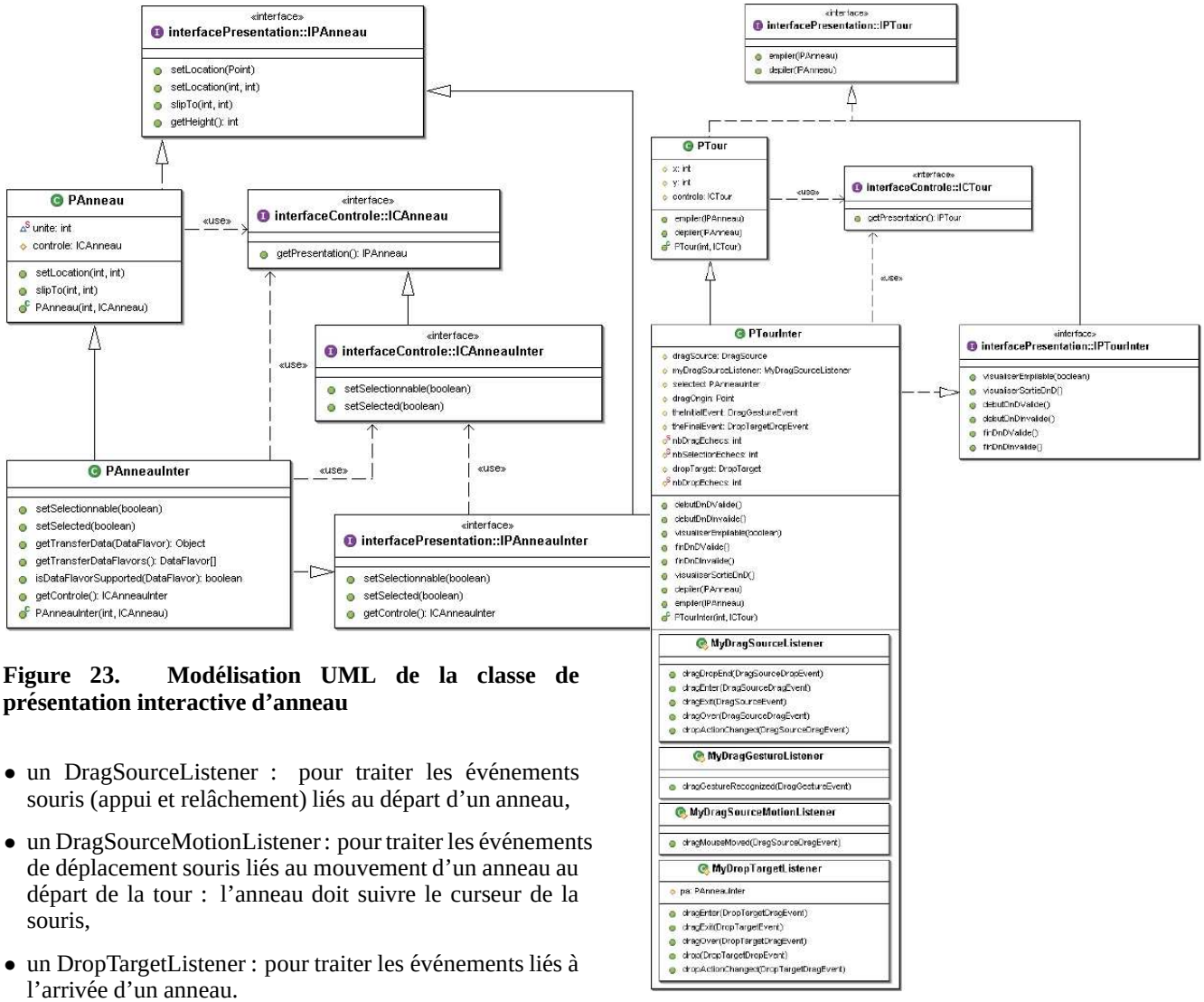

Figure 23. Modélisation UML de la classe de présentation interactive d'anneau

- un DragSourceListener : pour traiter les événements souris (appui et relâchement) liés au départ d'un anneau,
- un DragSourceMotionListener : pour traiter les événements de déplacement souris liés au mouvement d'un anneau au départ de la tour : l'anneau doit suivre le curseur de la souris,
- un DropTargetListener : pour traiter les événements liés à l'arrivée d'un anneau.

Les méthodes de ces classes auront la charge d'appeler aux endroits adéquats les méthodes suivantes du composant contrôle associé (après avoir transtypé ce composant contrôle en ICTourInter) : debutDnDDrag, finDnDDrag, entreeDnD-Drop, sortieDnDDrop et finDnDDrop. Il est à noter qu'à chaque fois le composant anneau qu'on est en train de manipuler est une instance de Transferable ou de JComponent, qu'il faut donc d'abord transtyper en IPAnneauInter, à qui on peut ensuite demander son composant contrôle (déclaré ICAnneau donc à transtyper également en ICAnneauInter) à transmettre au composant ICTour associé, lui aussi transtypé en ICTourInter. On aura donc du code ressemblant à ceci :

```
public void dragGestureRecognized (
   DragGestureEvent event) {
   selected = null ;
   try {
      selected =
         (PAnneauInter)getComponentAt (
            event.getDragOrigin ()) ;
   } catch (Exception e) {
   }
   if (selected != null) {
      ...
```

Figure 24. Modélisation UML de la classe de présentation interactive de tour

```
      ((ICTourInter)controle).
          debutDnDDrag (
             (ICAnneauInter)selected.
                getControle()) ;
      ...
   }
}
```

En retour, ces différentes méthodes du composant contrôle associé invoqueront une des méthodes de PTourInter décrites dans l'interface IPTourInter :

- visualiserEmpilable : pour visualiser si un anneau est empilable ou non sur la tour,
- visualiserSortieDnD : pour visualiser qu'un anneau sort de la zone correspondant à la tour,
- debutDnDValide, debutDnDInvalide : pour visualiser que le début de DnD reconnu sur la tour est valide ou pas,

- finDnDValide, finDnDInvalide : pour visualiser que le DnD qui s'est terminé sur la tour associée est valide ou pas.

Utilisation des packages interactifs

On a ensuite très peu de changements à effectuer pour obtenir une application interactive permettant à un utilisateur d'essayer de résoudre le problème des tours de Hanoï par manipulation directe des anneaux en les dépilant d'une tour et en les empilant sur une autre : il suffit ici encore de déterminer les bonnes fabriques de composants à utiliser..

```
public static void main (String a []) {
   int n = 3 ;
   if (a.length > 0) {
      n = Integer.parseInt (a [0]) ;
   }
   concreteFactory.ConcreteFactory.
      setAFactory (
         abstraction.AFactory.
            getInstance ()) ;
   concreteFactory.ConcreteFactory.
      setCFactory (
         controle.CFactoryInter.
            getInstance ()) ;
   concreteFactory.ConcreteFactory.
      setPFactory (
         presentation.PFactoryInter.
            getInstance ()) ;
   IHanoi h = concreteFactory.
      ConcreteFactory.getCFactory ().
         newHanoi (n) ;
   //h.solve () ;
}
```

Ici l'appel à la méthode solve disparaît pour laisser le choix de résolution à l'utilisateur.

Cette procédure est presque identique à la précédente, seules les fabriques de composants contrôles et présentations sont différentes car elles permettent de créer des composants destinés à supporter une manipulation directe de l'utilisateur.

Conclusion sur le passage au mode interactif

Nous avons choisi ici de créer de nouveaux composants de contrôle et de présentation par héritage des composants utilisés pour obtenir une visualisation à partir d'un package applicatif. Cette méthode est compatible avec les trois façons d'obtenir une visualisation à partir d'un package applicatif que nous avons présentées dans ce document.

Le but était ici de pouvoir gérer la manipulation directe au niveau des composants de contrôle en restant le plus indépendant possible de l'implémentation java du drag'n drop. Pour cela il a fallu définir de nouvelles interfaces pour les composants de contrôle interactifs et pour les nouveaux composants de présentation interactifs, afin de pouvoir mettre en place un dialogue entre composants de contrôle et leurs composants de présentation associés. Ceci a permis de restreindre la connaissance des classes java de gestion du drag'n drop aux seuls composants de présentation. Les nouveaux composants de contrôle interactifs ainsi définis devraient pouvoir être réutilisés en cas de changement d'implémentation des couches "basses" de présentation.

CONCLUSION

Les trois méthodes présentées pour déployer une application interactive selon le modèle PAC ou le modèle PAC-Amodeus sont viables. Elles offrent différents degrés d'indépendance du contrôle de l'IHM par rapport à un noyau applicatif à rendre interactif. Dans un premier temps ces méthodes ont été comparées sur un exemple d'application à illustrer graphiquement.

La première méthode offre la meilleure indépendance du contrôle, mais demande de créer beaucoup de classes et d'interfaces. Cela conduit à des dépendances minimales entre les différents composants, en particulier grâce au mélange des patrons de conception Proxy (et Delegation) et Template Method (ou Interface), ainsi bien sûr qu'à l'utilisation du patron Abstract Factory.

La seconde méthode peut permettre de limiter le nombre de classes à créer, qui de plus sont un peu plus simples que dans le cas précédent, car pour beaucoup de méthodes des composants contrôle on peut réutiliser telles-quelles les méthodes du composant applicatif dont on hérite. Par contre on induit des dépendances fortes entre les composants contrôle et les composants applicatifs dont ils dérivent. Cette solution conduit donc à un déploiement du modèle PAC moins pérenne qu'avec la première méthode, mais sensiblement plus rapide.

L'implémentation proposée dans la troisième méthode est assez différente des deux premières, mais elle semble fonctionnellement équivalente à la première méthode, même si elle alourdit quelque peu le codage à cause des protocoles à mettre en place entre composants applicatifs et composants de contrôle. Notons enfin que cette troisième méthode risque d'être moins efficace pour visualiser l'évolution d'un grand nombre d'objets interactifs autonomes.

Pour le passage de l'illustration graphique à l'interaction par manipulation directe des présentations associées aux composants applicatifs, ces trois méthodes sont en fait équivalentes, et on n'a donc présenté ce passage que dans le cadre de la première méthode. Dans cette partie nous avons essayé d'assurer la meilleure indépendance possible entre les composants de contrôle et les composants de présentation, pour rendre la partie de contrôle indépendante de la couche graphique utilisée.

Nous espérons que ce document permettra aux développeurs d'IHM de mettre en oeuvre plus facilement un modèle d'architecture de type PAC ou PAC-Amodeus pour rendre interactif un noyau applicatif à l'aide d'un langage objet comme Java ou C++, en s'appuyant sur les patrons de conception adequats.

REFERENCES

1. Coutaz J. *Interfaces homme-ordinateur, conception et réalisation*, Dunod informatique, 1990.

2. 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, 149–156.

3. 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, 86–93.

4. 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, 407–418.

5. Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns : Elements of reusable Object-Oriented Software*, Addison-Wesley, 1995.

6. Grand M. *Patterns in Java Volume 1 : A Catalog of Reusable Design Patterns Illustrated with UML*, Wiley Publishing Inc., 2002.

7. Jézéquel J.M., Train M., Mingins, C. *Design Patterns and Contracts*, Addison-Wesley, octobre 1999.

8. Nigay L. Conception et modélisation logicielle des systèmes interactifs : application aux interfaces multimodales

   Thèse de Doctorat de l'Université de Grenoble 1, IMAG, 1994.

9. The UIMS Workshop Tool Developers. ARCH : A Metamodel for the Runtime Architecture of an Interactive System In *SIGCHI Bulletin, 24, 1, pp. 32-37*, 1992.