Utilisation du modèle PAC-Amodeus pour une réutilisation optimale de code dans le développement de plusieurs versions d'un logiciel commercial

Frantz Degrigny

e-Manation
14, rue du Maréchal Juin
44100, Nantes, France
frantz.degrigny@e-manation.com

Thierry Duval

IRISA - UMR 6074
Campus de Beaulieu
35042, Rennes, France
thierry.duval@irisa.fr

RESUME
Dans le cadre du développement d'une application de gestion de CVs : Zoé, application à forte composante IHM, nous avons utilisé le modèle d'architecture logicielle PAC-Amodeus pour structurer nos développements. Cette approche nous a ensuite permis de développer extrêmement rapidement une version simplifiée de notre logiciel : Léa. Après un bref rappel du modèle PAC-Amodeus et de la méthodologie proposée pour une implémentation efficace de ce modèle, nous présentons ici la façon dont nous l'avons effectivement mis en œuvre et les quelques écueils à éviter dans le cadre d'un développement logiciel de taille conséquente. Ceci nécessite d'adapter quelque peu la mise en oeuvre pour tenir compte des particularités du cadre d'implémentation que sont l'utilisation du langage Java (qui ne permet pas l'héritage multiple) ainsi que la présence de plus de 2000 classes dans notre application.

MOTS CLES : Modèle d'architecture logicielle pour les IHM, Méthodes de génie logiciel pour les IHM, Implémentation efficace d'un modèle d'architecture, Patrons de conception.

ABSTRACT
We have developed Zoé, a software solution dedicated to CVs' management. In order to ensure its strongly graphical and interactive aspect we have used the PAC-Amodeus Software Architectural Model for GUIs. This approach allowed us to very quickly and easily develop a simplified version of this software: Léa. After a brief reminder on the PAC-Amodeus model we give indications on how to implement it efficiently in the context of a large program (more than 2000 classes) written in Java, a programming language which allows only simple inheritance.

CATEGORIES AND SUBJECT DESCRIPTORS : D.2.2 [Design Tools and Techniques] : Object-Oriented Design methods, User Interfaces ; D.2.13 [Reusable Software] : Reusable libraries, Reuse models.

GENERAL TERMS : Design, Experimentation

KEYWORDS : Software Architectural Models for GUIs, Software Engineering Methods for GUIs, Efficient Implementation of Software Architectural Models, Design Patterns.

INTRODUCTION
Notre cadre de travail est celui des IHM pour de grosses applications, avec le souci de pouvoir s'adapter facilement et rapidement à différents types de modifications si nécessaire : changement d'API graphique ou différentes versions de logiciels à fournir (comme par exemple des restrictions de l'application initiale à certains cas particuliers). Nous avons donc choisi d'utiliser un modèle d'architecture logicielle pour structurer nos développements et leur garantir ainsi une plus grande souplesse d'évolution. Parmi les modèles connus de nos développeurs citons MVC [6], PAC [1], Arch [2] et PAC-Amodeus [7]. Comme les modèles de types PAC et PAC-Amodeus sont orientés agents et garantissent une séparation forte entre IHM et application proprement dite, au contraire de MVC, c'est vers ces modèles que nous nous sommes tournés. Nous avons alors naturellement choisi d'utiliser le modèle PAC-Amodeus, avec la méthodologie d'utilisation de ce modèle présentée dans [3] et [4]. Cela nous paraissait d'autant plus pertinent que nous utilisons pour nos développements le langage Java [8] et l'API graphique Swing [9], langage et API principalement utilisés comme exemples dans ces articles de méthodologie.

Nous allons donc ici rappeler brièvement la méthodologie de déploiement proposée pour le modèle d'architecture PAC-Amodeus, et montrer comment cela nous a permis de développer Zoé, notre première application de gestion de CVs, qui propose à l'utilisateur de nombreuses IHM de visualisation, de consultation et d'édition de CVs. Nous montrerons ensuite comment le fait d'avoir utilisé cette architecture nous a permis de

réaliser extrêmement rapidement une version réduite de Zoé : Léa. Puis nous mettrons l'accent sur des choix particuliers de déploiement qui nous semblent judicieux dans notre contexte (différents packages et grand nombre de classes applicatives, plus de 500), au niveau de la mise en œuvre du patron de conception « AbstractFactory », au niveau de l'utilisation des « model » de swing, ainsi que d'autres petits détails techniques. Nous énumérerons enfin quelques règles concrètes qui peuvent aider à respecter le modèle PAC-Amodeus lorsque l'on est en phase de codage.

MODELE D'IMPLEMENTATION DE PAC-AMODEUS

C'est globalement celui décrit dans [3] et [4], qui s'appuie sur le modèle PAC-Amodeus ainsi que sur quelques patrons de conception décrits dans le livre du « Gang of Four » (usuellement référencé sous l'appellation « GoF ») [5] :

- Le noyau applicatif est structuré comme un ensemble de classes (Java pour notre application) qui peuvent communiquer entre elles et n'ont aucun lien avec quelque aspect graphique que ce soit. Ce noyau est totalement indépendant de toute bibliothèque ou API graphique donnée.
- Chaque classe du noyau initial qui doit être visualisée va être mise en relation, via un composant « adaptateur de domaine » avec un agent PAC qui va se charger d'établir le lien avec une couche graphique de présentation. Pour des raisons d'optimisation, nous supprimons ici la facette abstraction du composant PAC associé, et nous fusionnons ce composant adaptateur de domaine avec le composant contrôle de cet agent PAC. Ceci est possible parce que nous implémentons notre IHM avec le même langage que celui utilisé par le noyau fonctionnel, et nous notons bien qu'au passage nous acceptons d'introduire ainsi une dépendance forte des composants contrôle de l'IHM vis à vis du noyau fonctionnel, en introduisant ici une petite perte de portabilité du contrôle de notre IHM. Enfin, la relation entre la classe initiale et ce composant contrôle est un héritage. Le résultat est illustré Figure 1. Ce composant contrôle est en quelque sorte un Proxy (GoF207) du composant initial.

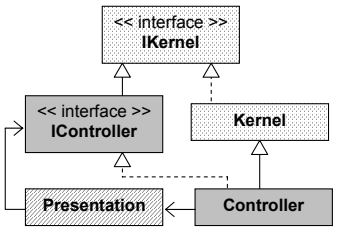


Figure 1 : Principe de déploiement de PAC-Amodeus.

- Notre application devra mettre en présence lorsque c'est nécessaire des composants de contrôle et non pas des composants applicatifs initiaux, cela se fera via l'utilisation du patron de conception « Abstract Factory » (GoF87).
- La facette présentation de chaque agent PAC est modélisée par une interface Java, qui a pour rôle d'être une présentation « logique » du composant, et qui sera implémentée par une instance concrète d'un composant « vue » lié à une boîte à outil graphique (AWT, Swing, SWT, ...) comme illustré Figure 2.

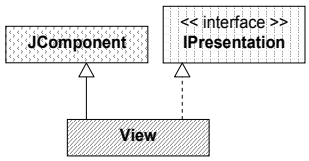


Figure 2 : Déploiement des composants présentation.

ZOE : PLUS DE 2000 CLASSES A DEVELOPPER ...

Zoé est un logiciel destiné aux services de Ressources Humaines pour accélérer la saisie des CVs en base de données. Zoé utilise des techniques d'intelligence artificielle (IA), de traitement automatique de la langue naturelle (TALN) et d'analyse sémantique pour trouver les informations contenues dans les documents numériques (ou papier après traitement par reconnaissance optique de caractères). Les informations détectées sont ensuite utilisées pour remplir automatiquement un formulaire. La structure et l'HM de ce formulaire sont paramétrées par un fichier de configuration, ceci pour adapter parfaitement le formulaire à la base de données qu'il doit renseigner. L'aspect de l'IHM de Zoé est présenté Figure 3.

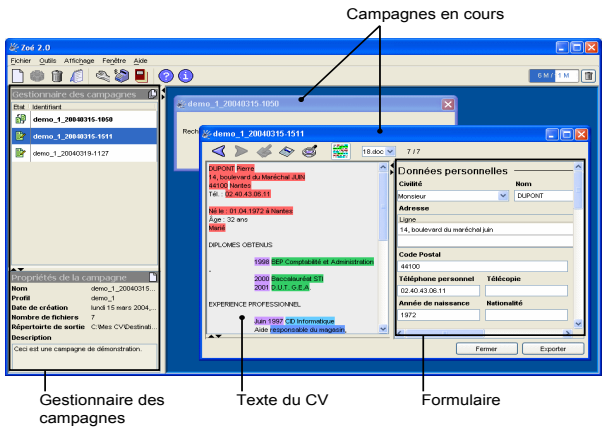


Figure 3 : IHM du logiciel Zoé.

La performance[1] de Zoé pour la détection sémantique est la suivante : plus de 90% pour les données d'état civil et 70% pour les autres informations.

La méthodologie de déploiement utilisée pour implémenter efficacement le modèle PAC-Amodeus, qui fusionne le composant adaptateur de domaine avec la

[1] La Performance est mesurée comme suit : Performance = 2 x nombre d'infos correctement détectées / (nombre d'infos à détecter + nombre d'infos détectées).

facette contrôle du composant PAC tout en supprimant sa facette abstraction, permet de ne pas avoir à créer un trop grand nombre de classes supplémentaires. Ce déploiement nous fait néanmoins passer d'un peu plus de 500 classes pour le noyau fonctionnel à plus de 2000 classes pour l'application ainsi rendue interactive, ce qui représente déjà un très grand nombre de classes.

Nous avons ainsi été confrontés à un besoin fort d'organisation de nos composants, ce qui nous a par exemple amenés à organiser le logiciel en différents packages – organisation orthogonale à la division classique en trois packages Abstraction (Kernel), Contrôle (Control) et Présentation (Presentation) – et à fournir une fabrique de composants par package au lieu d'une seule. Nous avons également dû tenir compte des particularités du langage choisi pour l'implémentation de notre application : Java, qui ne propose pas d'héritage multiple. Ceci impose quelques contraintes d'implémentation au niveau des composants de contrôle PAC, qu'on ne peut faire hériter à la fois de composants du noyau et de composants de présentation, ce qui pourtant pourrait bien souvent accélérer le développement de l'IHM et limiter le nombre de classes à développer. Ces deux aspects d'implémentation, ainsi qu'un certain nombre de pièges à éviter, seront discutés dans la section abordant les points techniques.

Par ailleurs, le fait d'utiliser une telle architecture a eu des effets non prévus mais qui s'avèrent tout à fait souhaitables au niveau de l'organisation de l'équipe. En effet, cette architecture est structurante car elle permet d'homogénéiser en partie les styles de codage des différentes personnes : il devient plus facile de naviguer dans du code que l'on n'a pas écrit soi-même. De plus, il est bien plus facile de se répartir les tâches, par exemple en attribuant un développeur au codage de l'IHM, un autre au codage du noyau, et les deux tour à tour au codage du contrôleur.

CONVENTIONS UTILISEES DANS LES EXEMPLES

Dans la suite de l'article, lorsque nous présenterons des exemples utilisant des diagrammes de classes, nous utiliserons les conventions suivantes :

Nommage des classes :

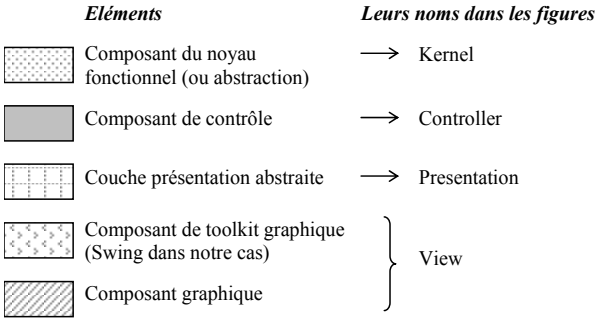
- Une classe abstraite commence par un « A ».
- Une interface commence par un « I ».
- Tous les objets du package « Presentation » commencent par « IP » (**I**nterface **P**resentation).
- Tous les contrôleurs finissent par « Ctrl ».
- Tous les composants graphiques se terminent par « View ».

Considérons par exemple un objet « Campagne » appartenant au noyau fonctionnel :

- son contrôleur s'appelle CampagneCtrl,
- son interface de présentation, IPCampagne,
- et son composant graphique, CampagneView.

Couleurs des schémas

Pour repérer encore plus facilement les différents types de composants, nous leur associerons également différents codes de couleurs :



DE ZOE A LEA : 7 SEMAINES SEULEMENT !

Léa est une version simplifiée de Zoé, qui offre moins d'options à l'utilisateur, ces options étant prédéterminées dans Léa alors qu'elles sont modifiables dans Zoé.

Le noyau applicatif est le même, certaines parties d'IHM sont simplifiées, d'autres sont augmentées (pour permettre d'éviter certaines étapes qui deviennent inutiles dans Léa à cause des choix imposés, pour simplifier la tâche de l'utilisateur, ...).

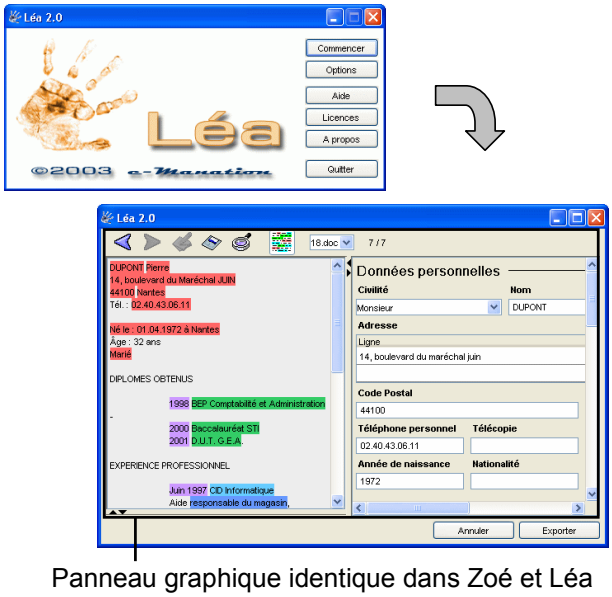


Figure 4 : IHM du logiciel Léa

Au contraire de Zoé qui accepte plusieurs types de formulaires, permet la gestion de plusieurs campagnes simultanées (une campagne est le traitement d'un lot de CV) et permet l'édition des lexiques, l'application Léa quant à elle n'accepte qu'un seul formulaire et une seule campagne à la fois. Ses lexiques ne sont pas éditables. L'IHM de Léa se présente comme un assistant et, hormis la première fenêtre, sa vue est la même que celle d'une

campagne dans Zoé. L'aspect de l'IHM de Léa est présenté Figure 4.

Zoé est plutôt destiné aux cabinets de recrutement qui ont plusieurs clients et donc plusieurs profils de base de données alors que Léa est plutôt destiné aux entreprises qui organisent leur recrutement en interne.

Zoé et Léa utilisent cependant tous les deux le même moteur d'analyse sémantique. La partie du noyau qui est différente pour Léa est le gestionnaire de campagnes : Léa implémente un gestionnaire trivial qui ne contient qu'une seule campagne.

Les différences entre les deux produits sont plus nombreuses au niveau des contrôleurs. En général, nous avons utilisé une classe abstraite qui factorise le code commun, comme présenté Figure 5.

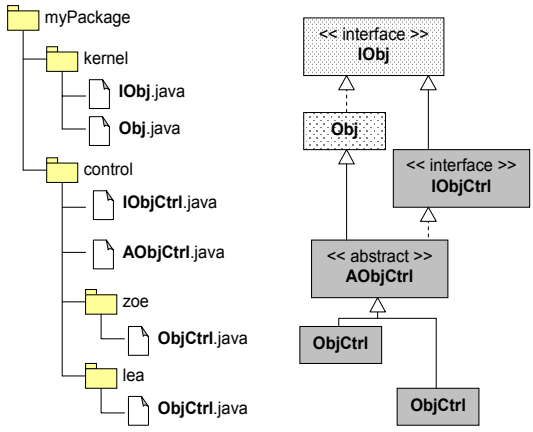


Figure 5 : Pour un même objet « Obj » du noyau, deux contrôleurs différents étendent un contrôleur abstrait.

On peut estimer que le temps de développement de Léa sans utiliser le modèle PAC-Amodeus n'aurait pas été beaucoup plus important : fournir de nouveaux éléments d'IHM et quelques fonctionnalités supplémentaires dans le noyau était de toutes façons incontournable, et le reste aurait pu être obtenu par copier-coller, plus quelques petites adaptations. Cependant, nous nous sommes rendus compte que l'effet structurant de l'utilisation du modèle PAC-Amodeus, en particulier sur la séparation qu'il impose entre le noyau fonctionnel et l'IHM proprement dite, facilitait forcément le développement de notre seconde application, qui aurait été plus longue à développer si la première avait été moins bien structurée.

Mais en fait, l'avantage essentiel du modèle et de sa méthode de déploiement est pour la maintenance : sans PAC-Amodeus, il aurait fallu fournir l'effort de deux maintenances en parallèle, sur deux applications qui risquaient de plus d'être moins bien conçues. Les gains de l'utilisation du modèle PAC-Amodeus sont donc encore plus importants à moyen et long termes qu'à court terme.

POINTS TECHNIQUES

Hiérarchie des packages

La question qui vient à l'esprit en premier lieu est la suivante : doit-on organiser les packages en largeur ou en profondeur ? Ces deux types d'organisation sont présentés Figure 6.

Notre réponse à cette question est que la seconde solution, à savoir le découpage des packages en profondeur, est plus adaptée à un découpage par composants, c'est celle que nous avons choisie.

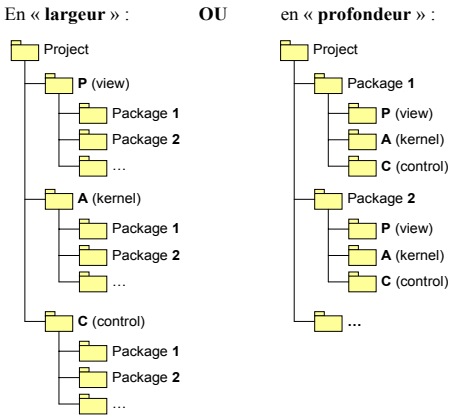


Figure 6 : Organisations possibles des packages.

Interface minimale pour les composants

Nous avons utilisé un package « **framework** » qui donne à tous les contrôleurs et toutes les présentations une même interface minimale commune, comme illustré Figure 7. Ce package n'est pas nécessaire mais il est très pratique pour partager des composants entre plusieurs applications.

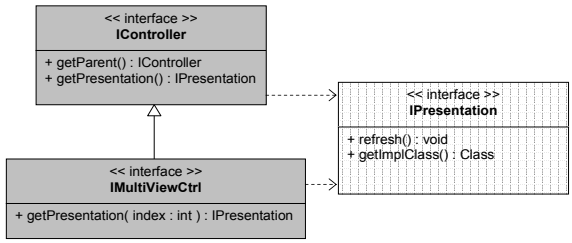


Figure 7 : Diagramme UML du package « framework ».

Sur cette figure, l'interface `IController` rappelle que les contrôleurs doivent être organisés en arbre et qu'ils sont susceptibles d'agir sur une présentation. L'interface `IMultiViewCtrl` est utile pour les contrôleurs qui ont plusieurs vues.

L'interface `IPresentation` doit quant à elle être étendue par toutes les présentations abstraites. Elle impose seulement une méthode qui permet de « rafraîchir » (mettre à jour) chaque présentation. En Java, on peut ajouter à cette interface une méthode `getImplClass()`, qui retourne le type de l'objet

graphique qui implémente concrètement la présentation abstraite. Cette méthode peut être utile au moment du trans-typage.

Avantages : L'utilisation de ce package permet un code plus homogène et l'utilisation de certains packages comme des composants. En effet, ce package regroupe les dépendances.

Fabrique de Fabriques

Plutôt que d'utiliser une seule fabrique de composants, qui aurait à créer dans notre cas plus de 1000 types d'objets de types différents (et de natures différentes : objets du noyau, objets de contrôle, objets de présentation), il nous a semblé nécessaire de regrouper les créations d'objets par familles de composants. Nous avons donc choisi d'utiliser une Fabrique abstraite par package pour une question de taille des classes et d'organisation en composants. Il nous a également semblé intéressant que la Fabrique soit un Singleton (GoF127) pour un confort d'utilisation.

Pour ces deux raisons nous avons été amenés à utiliser une Fabrique (appelée ici Builder) de Fabriques. Le patron de conception « Abstract Factory » (GoF87) est ici appliqué deux fois en cascade, comme illustré Figure 8.

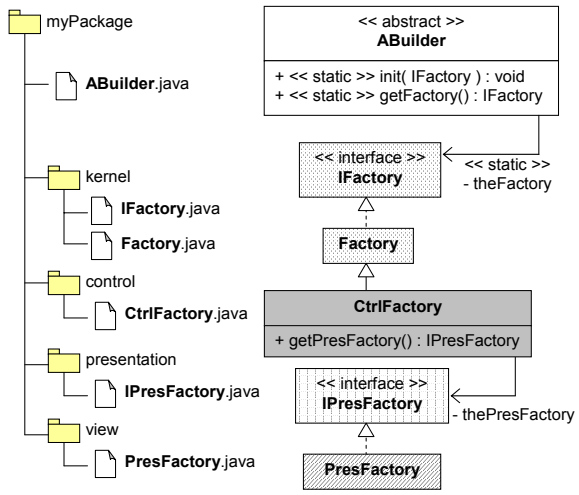


Figure 8 : Organisation des fabriques abstraites par package.

Il y a un Builder de Fabriques par package et un Builder général dans le package « **application** ». Ce dernier est chargé de l'initialisation des autres Builders.

Dans le code des classes du package chaque appel à **`new`** est remplacé par `Builder.getFactory().createMyObj()`.

Le package de l'application a une structure un peu particulière, présentée Figure 9. Il contient, en plus des classes décrites précédemment, les Builders de Fabriques nécessaires à l'initialisation du programme ainsi que le point d'entrée dans le programme (la méthode `main`).

Autre avantage, les différents sous packages peuvent ainsi être utilisés en composants, chaque composant étant réalisé par un package contenant les sous packages Présentation, Abstraction, Contrôle, les fabriques abstraites correspondantes et un Builder. L'utilisation des interfaces du package « **framework** » assure que les différents packages sont compatibles entre eux.

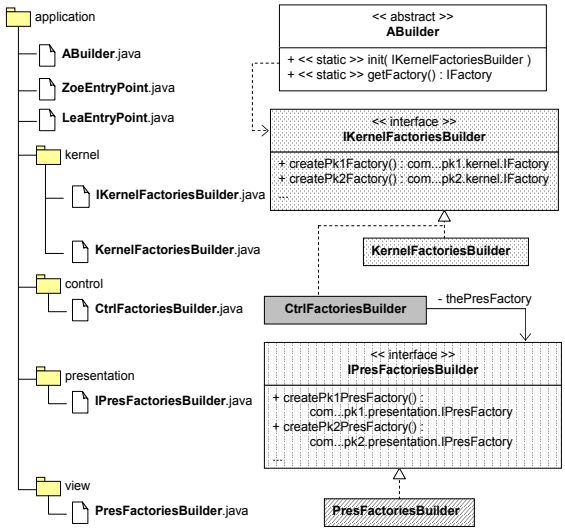


Figure 9 : Organisation des fabriques de fabriques au niveau du package « application ».

Finalement le point d'entrée dans le programme a l'allure suivante :

```
public class ZoeEntryPoint {

  public static void main(final String[] args) {

    final IPresFactoriesBuilder
              presFactBuilder = new ZoeViewFactoriesBuilder();

    final ZoeControlFactoriesBuilder ctrlFactBuilder = new
              ZoeControlFactoriesBuilder(presFactBuilder);

    ABuilder.init(ctrlFactBuilder);

    final IApplication application =
              ABuilder.getFactory().createApplication();

    application.start();

  }

} //end of class ZoeEntryPoint
```

On voit bien ici, que le démarrage de l'application est initié par l'abstraction et non pas par la vue. Ce qui est bien sûr préférable pour des raisons de modularité du code, mais qui est rarement le cas dans la plupart des applications.

L'appel à la méthode `init()` du Builder du package de l'application provoque l'initialisation, en cascade, des Builders des autres packages.

```
public abstract class ABuilder {

private static IFactory theFactory = null;

  public static void init( IKernelFactoriesBuilder
                           aFactoriesBuilder) {

  //inits all sub-packages builders
```

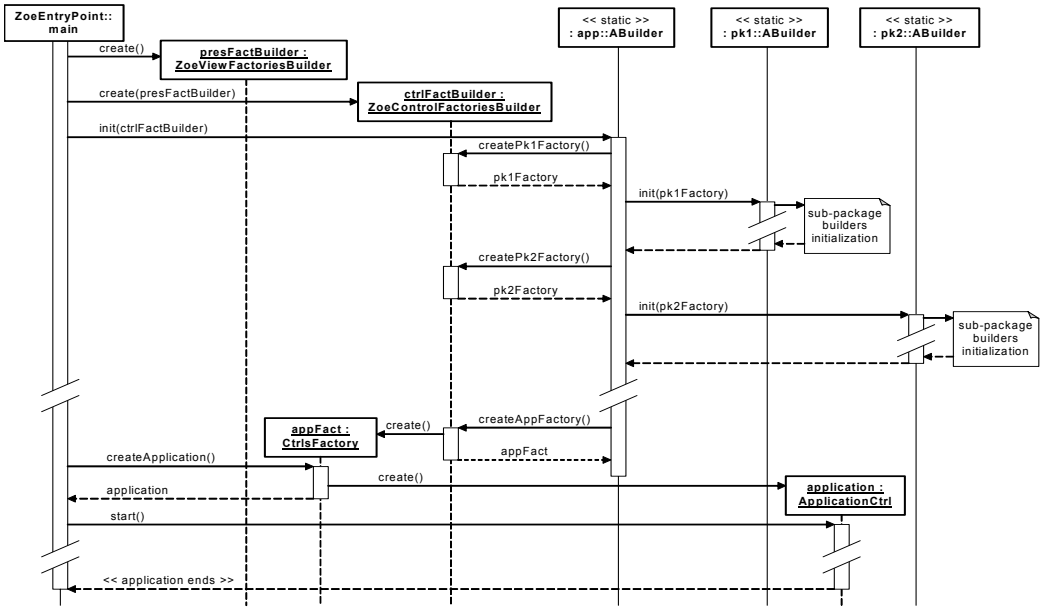


Figure 10 : Diagramme de séquence UML montrant l'initialisation du programme via les Builders.

```
   com...pk1.Builder.init(aFactoriesBuilder.createPk1Factory());
   com...pk2.Builder.init(aFactoriesBuilder.createPk2Factory());
   //...

   //inits its own factory
   theFactory = aFactoriesBuilder.createAppFactory();

   }

//...

} //end of class ABuilder
```

C'est au niveau du point d'entrée de chaque programme (ici Zoé ou Léa) que les composants à utiliser concrètement sont choisis par l'instanciation des Builders de fabriques correspondants. Il y a une famille de composants regroupant les fonctionnalités du programme (kernel et control) et une autre famille de composants pour les éléments de l'IHM. C'est en particulier là que peut se faire le choix d'utiliser une API graphique particulière pour l'instance d'application en cours, en choisissant les bonnes familles de composants de présentation.

Intégrer les « Model » de Swing dans les contrôleurs

Dans notre contexte de développement, nous avons souhaité profiter du découpage MVC des composants Swing, en présentant nos composants de contrôle comme des modèles pour les « JComponents », ceci pour profiter des mécanismes de notification prévus dans Swing. Il semble évident que bien que situées dans le package Swing, les interfaces des modèles sont plutôt du domaine du contrôle.

Mais pour des raisons d'absence d'héritage multiple en Java, utiliser les modèles de Swing pour les contrôleurs en même temps qu'implémenter le modèle PAC-Amodeus peut poser des problèmes :

- un contrôleur doit-il hériter de l'implémentation d'un modèle Swing et implémenter une interface du noyau applicatif (en encapsulant un objet du noyau applicatif) ?

- ou bien doit-il, comme prévu, hériter d'un objet du noyau et implémenter une interface de modèle Swing ?

La solution vient, encore une, fois de l'utilisation du patron de conception Proxy (GoF207).

Une première solution est donc de faire hériter le contrôleur du modèle Swing et de le faire implémenter une interface du noyau, ce qui est présenté Figure 11.

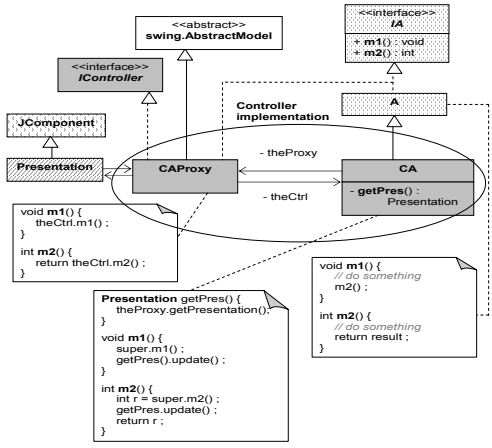


Figure 11 : Le contrôleur hérite du modèle Swing et implémente une interface du noyau.

L'autre solution, présentée Figure 12, où le contrôle hérite d'un objet du noyau et implémente une interface de modèle Swing, présente l'avantage de rester dans la logique du déploiement habituel du modèle PAC-Amodeus. Elle nous parait ainsi plus intéressante car elle

permet l'utilisation de la couche Présentation abstraite (ce qui n'est pas le cas pour la première solution).

Dans les deux cas, il suffirait de réécrire les quelques interfaces de modèles utilisées pour pourvoir réutiliser le code dans un contexte où l'on n'utiliserait plus Swing.

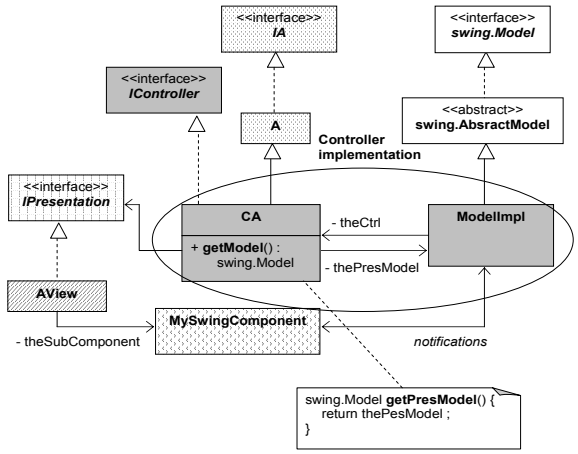


Figure 12 : Le contrôleur hérite du noyau et utilise une interface de modèle Swing.

C'est cette seconde solution qui a été mise en oeuvre dans Zoé et Léa, alors que la première solution a été utilisée dans une autre application développée antérieurement à e-Manation.

Pièges à éviter

Ordre de construction des objets. Le premier piège auquel il faut faire attention est l'ordre de construction des objets. En effet, à cause des héritages, l'ordre de construction des objets peut être surprenant au premier abord. La règle qu'il vaut mieux respecter est de ne pas commencer de traitement dans les constructeurs, ceci afin d'éviter d'appeler une opération sur un objet encore inexistant. Le traitement est lancé explicitement après la fin de l'initialisation par l'appel d'une méthode `start()`.

Hiérarchie des contrôleurs. Les contrôleurs sont organisés en arbre, dans lequel chaque nœud connaît son père, ceci pour permettre à chaque message d'atteindre son destinataire par un chemin unique. Mais le noyau n'étant pas nécessairement organisé en arbre, il peut être nécessaire d'indiquer à un contrôleur, après sa construction, qui est son père. Cette construction, nous le rappelons, est déclenchée par l'appel de la méthode correspondante de la fabrique.

Entorses à la règle. Normalement, les vues concrètes ne doivent pas communiquer entre elles (ce rôle est réservé au contrôleurs). Mais l'utilisation de boîtes à outils graphiques nous oblige à quelques entorses, notamment pour les opérations concernant :

- la mise à jour de la hiérarchie des composants,
- l'affichage ou le masquage des composants et
- la mise à jour de la taille des fenêtres et de la géométrie des composants.

La seule possibilité est de limiter le plus possible ces interactions et la connaissance que les vues ont de la structure les contrôleurs.

Exemple : examinons le cas d'une application dont un module doit avoir ses opérations accessibles via le menu de la fenêtre principale.

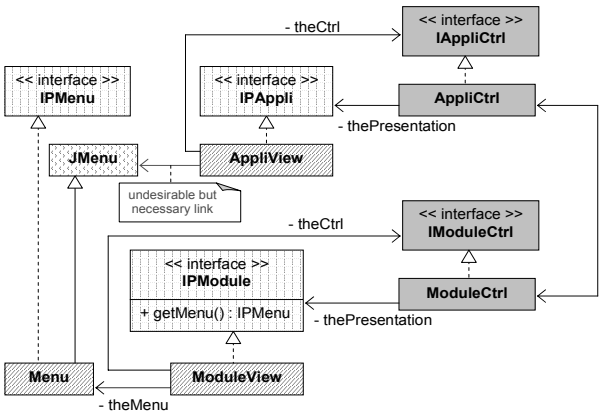


Figure 13 : Diagramme UML montrant l'implémentation des menus dans Zoé.

Le plus simple est que le menu soit un membre de la vue du module et que la vue de l'application en possède une référence. Nous avons ainsi implémenté les menus des modules de Zoé comme dans la Figure 13.

L'interface `IPMenu` propose une présentation abstraite pour le menu. Le lien inévitable entre la vue de l'application et le menu du module est dessiné en gris.

Pour éviter de donner aux vues trop de connaissance sur la structure des contrôleurs, il suffit de limiter le dialogue de chaque vue à son propre contrôleur. Dans ce cadre, on peut donner l'initiative à la vue :

```
public class AppliView implements IPAppli extends JFrame {

   public AppliView(IAppliCtrl aCtrl) {
   //...
   //Do not write this : JMenu moduleMenu = (JMenu)
   //          theCtrl.getModule().getPresentation().getMenu();
   JMenu moduleMenu = (JMenu) theCtrl.getModulePres().getMenu();
   theMenuBar.addMenu( moduleMenu );
   //...
   }

//...

} //end of class AppliView

public class AppliCtrl implements IAppliCtrl extends Appli {

   public IPModule getModulePres() {
     return (IPModule)theModuleCtrl.getPresentation();
   }

//...

} //end of class AppliCtrl
```

ou au contrôleur :

```
public class AppliView implements IPAppli extends JFrame {

   public void addModuleMenu(IPMenu aModuleMenu) {
     theMenuBar.addMenu( (JMenu)aModuleMenu );
   }

//...

} //end of class AppliView

public class AppliCtrl implements IAppliCtrl extends Appli {

   private IPAppli thePresentation;

   public AppliCtrl() {
     ICtrlFactory ctrlFactory =
                 (ICtrlFactory) Builder.getFactory();
     thePresentation =
                 ctrlFactory.getPresFactory().createPAppli(this);
     //...
     IPMenu moduleMenu =
          ((IPModule)theModuleCtrl.getPresentation()).getMenu();
     thePresentation.addModuleMenu(moduleMenu);
     //...
   }

   //...

} //end of class AppliCtrl
```

Eviter les boucles infinies. Comme les Vues notifient les Contrôleurs à chaque mise à jour et inversement, il faut ajouter un mécanisme de contrôle pour éviter les boucles infinies. Ce mécanisme peut consister simplement à réaliser un test de façon à ne propager des changements valeurs que si ces valeurs ont effectivement été modifiées.

Règles guides

Notre expérience nous a permis d'énoncer quelques règles que chacun doit respecter pour s'assurer que le code qu'il produit quotidiennement respecte le modèle théorique.

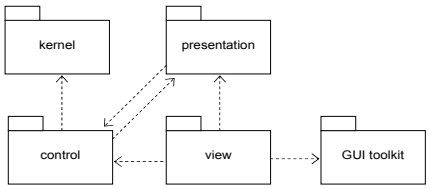


Figure 14 : Dépendances entre packages.

Ces règles sont les suivantes :

- Les classes du noyau et de la couche présentation ne doivent pas avoir de dépendances (`import` en Java) vers les packages « Contrôle » et « Vue ».
- Les contrôleurs ne doivent pas avoir de dépendances vers les classes de la boîte à outils graphique.
- Chaque vue ne doit connaître que son propre contrôleur.
- Les vues et les contrôleurs ne doivent pas contenir de traitement. Tout le code des fonctionnalités du programme doit se retrouver localisé dans les classes du noyau.
- Les contrôleurs doivent contenir uniquement du code de transmission de messages (appel de procédures) et de conversion de types entre la vue et le noyau.
- Les contrôleurs ne doivent connaître les vues que par l'intermédiaire des interfaces de présentation.

Le schéma de la Figure 14 permet de garder à l'esprit quelles sont les dépendances autorisées entre les sous packages PAC-Amodeus.

CONCLUSION

Les développements effectués valident largement le modèle utilisé ainsi que la méthode d'implémentation du modèle. Nous allons même au-delà de la simple validation en mettant en avant quelques règles supplémentaires permettant une meilleure prise en compte de la complexité d'une application commerciale, complexité principalement due à la taille de l'application ainsi qu'au besoin de pouvoir décliner le produit en plusieurs versions. Enfin, nous identifions quelques problèmes supplémentaires dus aux outils concrets d'implémentation Java et son API graphique Swing. Certains de ces problèmes sont suffisamment généraux pour être rencontrés avec de nombreux autres outils de développement, alors qu'ils ne sont pourtant quasiment jamais évoqués dans les articles traitant des modèles d'architecture logicielle. C'est donc ici aussi l'occasion de pointer ces problèmes concrets et de présenter nos solutions pour y remédier.

BIBLIOGRAPHIE

1. Coutaz J. Interfaces homme-ordinateur, conception et réalisation. Dunod informatique, 1990.
2. The UIMS Workshop Tool Developers. Arch : A metamodel for the runtime architecture of an interactive system. SIGCHI Bulletin, 24, 1, pages 32-37, 1992.
3. Duval T. and Nigay L. Implémentation d'une application de simulation selon le modèle PAC-Amodeus. In IHM, Toulouse, France, pages 86-93, 1999.
4. Duval T. and Pennaneac'h F. Using the PAC-Amodeus Model and Design Patterns to Make Interactive an Existing Object-Oriented Kernel. In Tools Europe, IEEE, Mont Saint-Michel, France, 2000.
5. Gamma E., HelmR., Johnson R., and Vlissides J. Design Patterns: Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.
6. Goldberg A. Information models, views and controllers. Dr Dobb's Journal, july 1990.
7. Nigay L. Conception et modélisation logicielle des systèmes interactifs : application aux interfaces multimodales. Thèse de Doctorat de l'Université de Grenoble 1, IMAG, 1994.
8. Campione M., Walrath C., Huml A. The Java™ Tutorial, Sun. http://java.sun.com/docs/books/tutorial/
9. Walrath C., Campione M., Huml A., Zakhour S. The JFC Swing Tutorial, Sun. http://java.sun.com/docs/books/tutorial/uiswing/