Une métaphore d'aide à la structuration d'une Application Interactive

Michelle Jacomi[1,2] *Stéphane Chatty*[1] *Philippe Palanque*[1,3]

[1]ÇENA
7 avenue Édouard Belin
BP 4005
31055 Toulouse cedex

[2]CISI
13, rue Villet
ZI du Palays
31029 Toulouse Cedex 4

[3]LIS-FROGIS
Université Toulouse 1
Place Anatole France
31042 Toulouse Cedex

{chatty,jacomi,palanque}@cena.dgac.fr

RÉSUMÉ

Structurer des applications interactives implique toujours des choix de conception complexes. A l'heure actuelle, les techniques disponibles ne couvrent pas les problèmes de structuration pour tous les types d'applications, quelle que soit leur taille. Les programmeurs sont donc obligés d'effectuer des choix sans maîtriser totalement leurs implications. Tout comme les métaphores ont été introduites au niveau des interfaces homme-machine, nous proposons l'utilisation de métaphores pour assister les programmeurs dans la structuration du code des applications interactives. Cet article décrit une métaphore cinématographique qui permet d'organiser l'interface homme-machine en objets, mais décrit aussi les liens entre ces objets et ceux du noyau fonctionnel. Nous montrons comment cette approche peut être combinée avec les techniques logicielles actuelles de manière à spécifier et construire des applications de taille significative. Ceci est illustré par la présentation d'un logiciel de gestion de l'espace aérien.

MOTS CLÉS : Architecture logicielle, conception d'interface utilisateur, métaphore

INTRODUCTION

La structuration d'applications interactives reste, par de nombreux aspects, un problème ouvert pour les concepteurs et les programmeurs. Les modèles d'architecture généraux tels que celui de Seeheim [17] décrivent comment doivent être organisés les grands blocs d'une application, mais ne donnent aucune indication sur leur construction pratique. A l'autre extrémité du spectre, les boîtes à outils à objets aident les programmeurs à construire les parties interactives de leurs applications, mais ne fournissent aucune aide pour organiser le noyau fonctionnel et structurer les différentes parties. Bien que de nouvelles approches comme les "design patterns" tentent de combler le fossé entre ces deux approches extrêmes, les programmeurs sont encore livrés à eux-mêmes pour la plupart des choix de conception. Ainsi, les programmeurs ne disposent pas d'une méthode complète pour organiser le développement d'applications de taille conséquente. Cela peut conduire à des situations inextricables pour les choix architecturaux dans le cas des applications interactives. En effet, les objets d'interaction sont nombreux et les liens entre ces objets et les objets du noyau fonctionnel sont multiples et complexes.

Cet article propose l'utilisation d'une métaphore pour guider les programmeurs. Les métaphores sont employées avec succès depuis de nombreuses années pour structurer les interfaces utilisateurs. Elles ont été introduites pour éviter que les utilisateurs ne soient perdus quand la complexité des systèmes augmente. Elles sont particulièrement utiles depuis l'apparition de l'interaction dirigée par l'utilisateur, où les utilisateurs sont maîtres de l'application, mais où pratiquement aucun "guide" ne leur est offert. Les métaphores comblent ce déficit en aidant les utilisateurs à comprendre la signification et le comportement des objets interactifs. En effet, elles permettent à l'utilisateur de deviner le comportement des objets représentés en fonction de celui de leur homologue dans la métaphore. Comme le soulignent [18], cette aide permet de "retrouver son chemin à travers la jungle des fonctionnalités".

Des métaphores ont également été introduites dans des environnements de programmation visuelle. On trouve par exemple le système Molière qui associe une métaphore théâtrale à un éditeur de programmes destiné à l'apprentissage de Smalltalk [3]. Ce système s'inspire du système Playground qui dispose d'une métaphore permettant de faire une analogie entre des animaux fictifs et les composants d'un programme, afin d'enseigner les langages de programmation aux enfants [9]. Néanmoins, de telles métaphores ne sont actuellement utilisées qu'à des fins éducatives, pour aider à appréhender la programmation par objets et faire concevoir par des non-spécialistes des petits programmes animés. En revanche, l'idée de se baser sur une métaphore pour assister des programmeurs dans la construction d'applications interactives n'a jamais à ce jour été explorée ni appliquée. Nous pensons qu'un parallèle peut être fait entre l'aide à fournir aux utilisateurs finaux et celle pour les concepteurs et les programmeurs. Ainsi, nous proposons d'associer des métaphores aux techniques classiques de construction logicielle.

Nous commençons cet article par la présentation des techniques de structuration existantes. Nous présentons ensuite la métaphore que nous avons définie et utilisée

pour construire notre application interactive, et nous montrons les liens entre cette métaphore et les architectures logicielles. Enfin, nous présentons une application de cette métaphore pour structurer un logiciel de gestion de l'espace aérien.

LES TECHNIQUES DE STRUCTURATION

La structuration des systèmes interactifs a été étudiée depuis de nombreuses années [17] mais constitue encore un sujet de recherche actif [7]. Le principal problème à résoudre consiste à définir et organiser les différents composants et à établir clairement leurs relations. Les solutions habituellement proposées sont des adaptations aux systèmes interactifs de techniques de génie logiciel existantes. Ces travaux de recherche peuvent être divisés en quatre catégories suivant le style d'approche qu'ils utilisent : abstraction d'abord, implémentation d'abord, réutilisation d'abord ou sémantique d'abord.

Abstraction d'abord : Les Modèles

Ce type d'approche propose des modèles généraux qui sont plus adaptés à la compréhension des systèmes interactifs qu'à leur implémentation. Parmi ces approches, on retrouve le modèle de Seeheim [17] et le modèle Arch/Slinky [1]. Le principal apport de ces modèles est de simplifier les applications interactives en les décomposant en entités abstraites.

Ces modèles sont très utiles pour la gestion de haut niveau des applications, mais ils sont difficilement utilisables pour leur implémentation. Par exemple, le modèle de Seeheim ne peut pas être appliqué à la lettre lors d'un développement par objets. En effet, l'organisation en couches n'est pas compatible avec des modèles à objets qui prônent le couplage faible et la forte cohérence entre les objets. A cause de cette contradiction, des modèles comme celui présenté dans [12] raffinent le modèle de Seeheim de manière à être plus proche de l'implémentation.

Le modèle par agents est un autre style d'approche prônant l'abstraction d'abord, en organisant l'application en un ensemble d'agents coopérants. Ces modèles peuvent être utilisés récursivement pour définir l'application à différents niveaux d'abstraction. L'aspect communicatif des systèmes interactifs fournit une base pour structurer ensuite le système comme un réseau d'interacteurs, chacun traitant un sous-ensemble particulier du dialogue homme-machine. Les modèles les plus utilisés sont MVC et PAC [6], maintenant étendu au modèle PAC-Amodeus [7] qui est un mélange entre le modèle Arch/Slinky et le modèle à agents PAC.

Les principaux désavantages de ces approches sont le manque de support méthodologique pour aider à la décomposition descendante, et le fait que le nombre d'éléments du modèle est trop réduit pour fournir une classification significative des composants dans le cas d'une application de taille réelle.

Implémentation d'abord : Les Boîtes à outils

Les boîtes à outils apportent des fonctions et des objets pour construire des systèmes interactifs. Par comparaison aux modèles, elles répondent au problème de structuration des systèmes interactifs de façon diamétralement opposée. Les boîtes à outils sont habituellement basées sur des modèles conceptuels qui imposent une manière prédéfinie pour organiser le code des applications. Par exemple, les environnements basés sur les événements comme Sassafras [11] ou plus récemment Visual Basic^{TM} organisent le code suivant des gestionnaires d'événements. Certains environnements de développement sont associés à d'autres types de squelettes d'applications comme dans MacApp. Il faut un long temps d'apprentissage aux programmeurs pour savoir les utiliser, et ils sont extrêmement liés à la boîte à outils sous-jacente. En conséquence, ces environnements ne fournissent pas des règles de composition ou des composants réutilisables. Un autre problème est qu'ils ne sont pas associés à des modèles abstraits, et qu'il est donc très difficile pour les programmeurs d'avoir une vue globale de l'application.

Réutilisation d'abord : Design Patterns

Une nouvelle approche consiste à proposer les "design patterns" comme modèle pour construire des applications [4, 10]. Ils procurent des informations utiles à la fois pour structurer un système et pour aider à son implémentation. Ces "patterns" peuvent être vus comme la "colle" entre les modèles abstraits et les implémentations, mais ils s'adressent davantage aux aspects de bas niveau du cycle de vie d'une application, et sont donc plus appropriés au niveau de l'implémentation qu'au niveau conceptuel. Dans [4] deux "patterns" (MVC and PAC) répondent à la question des systèmes interactifs, mais ils sont seulement des extensions de modèles utilisés depuis longtemps dans le domaine des systèmes interactifs.

Sémantique d'abord : Métaphores

Une métaphore apporte un support grâce auquel les utilisateurs peuvent rapidement apprendre comment utiliser un système. Grâce aux métaphores, les utilisateurs déduisent la signification, le comportement et la manipulation des objets en faisant la correspondance entre les aspects du système et leur connaissance de la métaphore. Par exemple, grâce à la métaphore du bureau, les utilisateurs peuvent déduire qu'un objet ressemblant à une poubelle servira aux objets qui doivent être supprimés, et qu'il faudra pour cela déposer ces objets dans la poubelle. On ne donne pas d'instructions explicites sur la manière de faire. Ainsi, la connaissance concernant la suppression des objets fait partie de la connaissance de l'utilisateur sur l'utilisation des poubelles dans le monde réel [15].

Les métaphores n'ont pas seulement été utilisées pour organiser la présentation des systèmes interactifs. Plusieurs boîtes à outils ou environnements de développement ont été conçus suivant des métaphores. Par exemple X_{TV}[2] est une librairie graphique basée sur une métaphore télévisuelle, et Whizz [5] qui est une extension de X_{TV} permettant de construire des applications interactives animées, utilise quant à elle une métaphore musicale pour traduire l'animation. Un environnement de programmation basé sur une métaphore aide les concepteurs à organiser les différents composants logiciels. Le besoin d'une métaphore est d'autant plus important pour les applications incluant des outils d'interaction avancés car elles contiennent un grand nombre d'objets d'interface spécia-

lisés et interagissant ensemble.

Discussion

Bien que les différents modèles présentés ci-dessus aient prouvé leur utilité pour aider à la conception des systèmes interactifs, ils n'apportent pas suffisamment d'indications pour structurer les composants logiciels nécessaires dans une application interactive complexe. Décomposer un tel système en agents implique la création de centaines de composants d'interface. Ces composants, leurs rôles et leurs relations sont difficiles à identifier pour les concepteurs, et ceci se révèle encore plus compliqué lors de la phase de maintenance du système. Le principal problème de ces approches est qu'elles ne proposent des indications qu'à un niveau générique, c'est-à-dire pour tous types d'applications interactives et non pour l'application qui doit spécifiquement être construite. Ce point est développé dans le paragraphe suivant.

LA METAPHORE CINEMATOGRAPHIQUE

Nous devions concevoir une application interactive intégrant de nouvelles techniques d'interaction. Pour intégrer de tels mécanismes nous avons été confrontés à des problèmes de choix d'architecture, notamment pour assurer la souplesse et la modularité de la structure globale de l'application. Pour répondre à ces problèmes, nous avons utilisé un schéma d'architecture métaphorique basé sur le modèle PAC pour la structure interne des objets interactifs. Cette section présente la métaphore qui est basée sur l'emploi de plusieurs caractéristiques du monde cinématographique. Cependant, comme l'a constaté [14] "les métaphores n'imposent pas une correspondance exacte de chaque détail concret d'un objet ou d'une situation vers un autre". C'est pourquoi certains aspects de la métaphore seront développés, tandis que d'autres ne seront pas repris.

Contexte

Nous avons utilisé la boîte à outils X_{TV}-Whizz comme point de départ. La métaphore télévisuelle proposée dans X_{TV}permet aux concepteurs de structurer la plupart des composants de l'application. Toutefois, cette métaphore ne fournit pas une base de description pour le comportement et les relations entre les objets de l'interface et ceux du noyau fonctionnel. De plus, cette métaphore se place à un niveau d'abstraction unique, qui est très proche de l'implémentation. Nous avons donc étendu cette métaphore pour répondre à ces problèmes. La métaphore cinématographique qui en résulte n'est pas seulement dédiée à modéliser les composants de la librairie, mais permet également de décrire les composants d'une application, ainsi que leurs relations, et de les appréhender suivant différents niveaux de détails.

La métaphore

Tous les objets sont articulés autour des actions de l'utilisateur et coordonnés par un objet global qui est le réalisateur. Son rôle est d'assurer la cohérence globale et la bonne exécution de l'application en fonction des ses spécifications, de son état courant et des actions de l'utilisateur.

Le réalisateur est aidé dans son rôle par des assistants, qui sont responsables de parties spécifiques de l'application. Le découpage hiérarchique entre le réalisateur et les assistants correspond ainsi au découpage fonctionnel de l'application. Les assistants sont classés suivant trois catégories :

1. *Les assistants associés au noyau fonctionnel*, qui assurent la communication et la cohérence entre la partie Interface et le Noyau Fonctionnel. Ces assistants correspondent d'une manière pratique à la partie Adaptateur du Noyau Fonctionnel du modèle Arch. Ainsi, le gestionnaire de fichiers du Macintosh est un assistant associé au noyau fonctionnel qui contient le gestionnaire réel des fichiers.

2. *Les assistants associés à l'interface*, et dédiés aux interactions avec les utilisateurs. Ces assistants sont situés dans les parties Dialogue et Présentation du modèle Arch. Le gestionnaire de l'aide du Macintosh peut être vu comme un assistant à l'Interface.

3. *Les assistants hybrides*, qui englobent le modèle Arch de la partie Adaptateur du noyau fonctionnel jusqu'à la partie Présentation. Ces assistants encapsulent chacun une fonctionnalité interactive complète de l'application.

Cette distinction entre les assistants vient des modèles abstraits présentés en introduction et est importante car elle préserve l'indépendance entre la partie fonctionnelle et la partie interface de l'application.

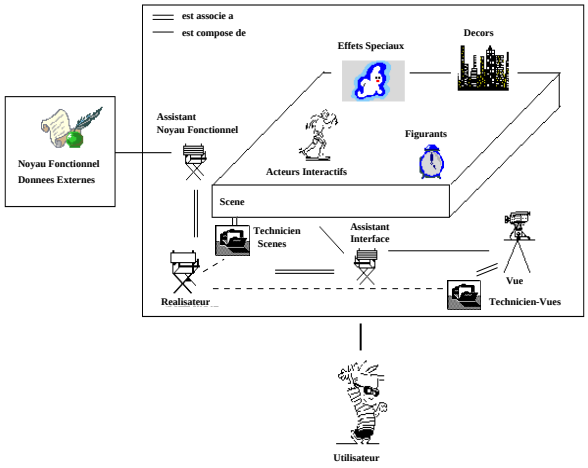

Figure 1: La métaphore cinématographique

Les différents objets de la métaphore cinématographique sont présentés en Figure 1. Les objets de l'interface, baptisés *Acteurs*, ont des caractéristiques ("costumes" graphiques et "capacités" sonores) et des comportements prédéfinis ("scénarios"). Ils peuvent également réagir aux actions utilisateurs ainsi qu'aux événements émis par les autres acteurs. Les Acteurs sont localisés sur des surfaces virtuelles baptisées *Scènes*, et sont visualisés par les utilisateurs grâce à des objets baptisés des *Vues* (i.e. des fenêtres au sens X).

Plusieurs scènes peuvent être affichées dans une vue. Une scène regroupe tous les acteurs intervenants pour

un même groupe fonctionnel, et qui sont donc sous la direction d'un assistant-réalisateur. Le contenu de chaque scène est organisé de manière à assurer la cohérence avec le noyau fonctionnel.

Nous avons distingué deux types de vues :

1. les vues *globales*, qui permettent de visualiser des scènes contrôlées par des assistants différents, ce qui correspond dans notre métaphore cinématographique à des plans d'ensemble, dirigés directement par le réalisateur.

2. les vues *locales*, propres à un assistant, l'aidant dans son travail pour régler des plans de détail (interactions locales), mais n'étant pas "retransmis" au réalisateur.

Pour assurer la cohérence générale de l'application pour la gestion des vues globales et des scènes dirigées par les assistants, nous avons identifié deux types d'assistants de production directement associés au réalisateur : le gestionnaire des caméras ayant connaissance des vues globales, et le gestionnaire des scènes ayant connaissance de l'ensemble des scènes.

Pour permettre aux concepteurs une classification plus précise des acteurs figurants sur les scènes, la métaphore distingue quatre catégories d'acteurs :

1. *Les Acteurs Interactifs* sont des entités qui peuvent être manipulées par l'utilisateur. Ils correspondent en général aux concepts du domaine qui sont visualisés par l'interface.

2. *Les Figurants* sont des acteurs dont l'apparence et/ou le comportement évoluent en fonction de l'état courant de l'application, mais ne peuvent pas être manipulés par les utilisateurs. Les Figurants ne dépendent que de l'application, et ne sont pas directement liés aux interactions.

3. *Les Effets Spéciaux* sont des acteurs temporaires créés dynamiquement et utilisés pour fournir du feed-back pendant les interactions. Le feed-back peut être visible ou sonore. Par exemple, dans la métaphore du bureau, un acteur temporaire est créé lorsque l'utilisateur déplace un fichier à l'écran.

4. *Les Décors*, sont des objets graphiques non interactifs définissant l'apparence statique de l'interface. Leur apparence ne change jamais lors d'une session. Ils constituent l'arrière-plan de l'affichage comme par exemple le fond d'écran du bureau Macintosh.

Un scénario décrit le comportement de chaque acteur en fonction de son état interne et de l'état courant de l'application qui est géré par le réalisateur.

Cependant, ni la structure interne des objets, ni le mécanisme de communication entre les objets ne sont décrit par la métaphore cinématographique.

Décomposition des Acteurs

Pour décrire la structure interne des objets, nous avons appliqué et raffiné le modèle PAC [6]. La Figure 2 présente la représentation graphique de ce raffinement du modèle.

- Le Contrôle décrit la communication entre les objets. Deux mécanismes ont été identifiés. Le premier, appelé distribution est utilisé lorsque l'émetteur ne connaît pas le destinataire (une sorte de broadcast). Le deuxième est appelé client-serveur, et est utilisé lorsque l'émetteur conserve une référence sur le destinataire et invoque directement ses services (fonctions publiques).

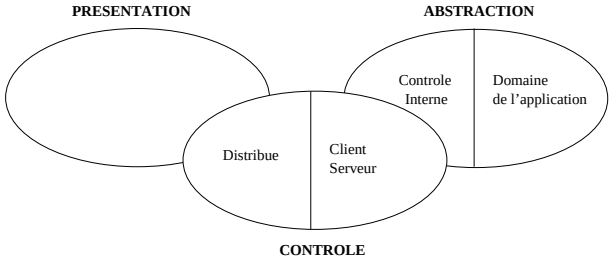

Figure 2: Raffinement du modèle PAC

- L'Abstraction est composée de deux parties : la partie Application et la partie Interne. La première partie contient les attributs et les fonctions relatifs au domaine de l'application. La partie Application contient donc les concepts sémantiques spécifiques à l'objet. La deuxième partie gère l'état interne de l'objet, et encapsule les attributs et les fonctions assurant la cohérence interne de l'objet.
- La Présentation contient les attributs graphiques ou sonores des acteurs. Ces attributs ne peuvent être manipulés par des fonctions qu'à partir des mécanismes de communication décrits plus haut.

UNE APPLICATION COMPLETE

Osmose est une plate-forme interactive développée au CENA, destinée à optimiser le découpage de l'espace aérien en secteurs ainsi que la répartition des flux de trafic. La Figure 3 montre l'utilisation d'Osmose pour l'édition de la sectorisation et des routes aériennes.

Figure 3: L'interface d'Osmose

A ce jour, Osmose représente plus de 100000 lignes de code, dont près de 90% sont dédiées à l'interface. Pour développer un logiciel d'une telle taille, il est nécessaire d'appliquer rigoureusement une méthode de structuration. Cette méthode doit répondre à trois objectifs : identifier les objets de l'interface, établir les liens entre les objets et organiser l'ensemble de ces objets. Les objets définis pour l'interface doivent être identifiés de façon univoque pour assurer la pérennité du projet, la cohérence globale de l'application, et la compréhension de l'ensemble du logiciel par tout programmeur intervenant dans son développement.

La métaphore cinématographique présentée dans cet article a été utilisée pour structurer les classes d'objets définies dans Osmose et concevoir l'architecture globale de l'application. Tous les composants logiciels construits correspondent à une entité de la métaphore.

Décomposition fonctionnelle

L'analyse fonctionnelle d'Osmose a permis de déterminer différents groupes :

- Les fonctions liées au noyau fonctionnel, qui permettent de gérer les relations avec :
  - l'optimisation de la sectorisation,
  - l'optimisation de l'affectation des flux de trafic,
  - l'évaluation de la charge des secteurs.
- Les fonctions liées au caractère interactif de l'application :
  - l'historique des interactions,
  - la grille d'édition,
  - les filtres paramétrables,
  - les sorties imprimées,
  - l'aide en ligne,
  - l'état courant de l'application.
- Les fonctions mixtes ou hybrides :
  - l'édition des secteurs,
  - la visualisation des informations sur les aéroports, les balises, les routes aériennes et les plans de vol.
  - la visualisation des cartes géographiques.

Identification des assistants

En appliquant notre métaphore, nous avons tout d'abord identifié les assistants responsables de chacun de ces groupes fonctionnels. Ces assistants secondent le réalisateur *OsmoseDirector*, qui assure quant à lui la cohérence globale de l'application. Nous avons identifié les assistants suivant les trois catégories distinguées dans notre métaphore :

- Les assistants entièrement dépendants du noyau fonctionnel, comme le *Genetic Algorithms Assistant*, interfaçant les résultats fournis par les calculs d'optimisation, et le *SectorEvaluation Assistant*, chargé du calcul de la charge des secteurs en fonction des flux de trafic.
- Les assistants purement interactifs, dépendant uniquement des interactions de l'utilisateur. C'est le cas du *Undo/Redo Assistant* qui gère les listes mémorisant les manipulations directes des utilisateurs sur les objets présentés dans l'éditeur. C'est également le cas du *MagicLenses Assistant* gérant l'affichage de filtres mobiles de visualisation à la demande de l'utilisateur.
- Les assistants "hybrides" ayant un rôle mixte de communication avec le noyau fonctionnel d'une part et l'utilisateur d'autre part. L'*ATC Assistant* et le *Sector Assistant* sont des assistants hybrides, car ils gèrent des objets interactifs liés au noyau fonctionnel (les données de l'espace aérien). Le *Map Assistant* est lui aussi un assistant hybride assurant la représentation des contours géographiques de différents pays.

Structure des assistants

Nous avons ensuite appliqué la métaphore pour identifier les rôles au sein d'un assistant. Suivant la métaphore, un *Assistant* gère des vues locales pour assurer différents types d'affichage, des scènes pour organiser les informations visualisées, et des acteurs qui représentent les entités interactives ou non.

Cette décomposition se retrouve dans tous les assistants, et nous la détaillons dans la figure 4 pour l'*AirportAssistant*, qui seconde l'*ATCAssistant* pour la gestion des aéroports. L'*AirportAssistant* gère une vue permettant de visualiser des objets interactifs représentants des aéroports, et organisés sur une scène particulière. Cette décomposition étant commune à tous les assistants gérant les données ATC, nous avons défini une classe abstraite *ATCHandler*, dont dérivent les assistants *AirportAssistant*, *BeaconAssistant* et *WayAssistant*. A ce niveau d'implémentation, nous nous sommes alors penchés sur les design patterns existants [10] pour vérifier si l'un d'entre eux étaient en adéquation avec les assistants. Parmi eux, nous avons choisi le design pattern *Singleton* comme base d'implémentation des Assistants. En effet, ce design pattern a la propriété d'assurer qu'une classe ne possède qu'une seule instance, et permet d'avoir un point d'accès global à celle-ci. Cela est tout à fait conforme au caractère unique de chacun de nos assistants dans notre métaphore. Cela est décrit par le code C++ suivant :

```
template <class ATCData> class ATCHandler: public DirectorAssistant {
protected:
    ATCHandler ();
public:
    ~ATCHandler ();
/* Objets Interactifs */
    DictionnaryOf<ATCData>* ATCDataTable;
    ListHandler* TheDialogBox;
/* Scènes */
    XtvStage DataStage;                                          10
    XtvStage DataLabelStage;
/* Fonctions liées au noyau fonctionnel */
    virtual void ReadATCData (const char* filename);
/* Fonctions liées à la présentation */
    virtual void ResetData ();
    void ManageDisplay ();
};

class AirportAssistant : public ATCHandler <Airport> {
public:                                                          20
    static AirportAssistant* GetInstance();
    void ReadATCData (const char* filename);
    void ResetData ();
protected:
    static AirportAssistant* AirportAssistantSingleton;
};
```

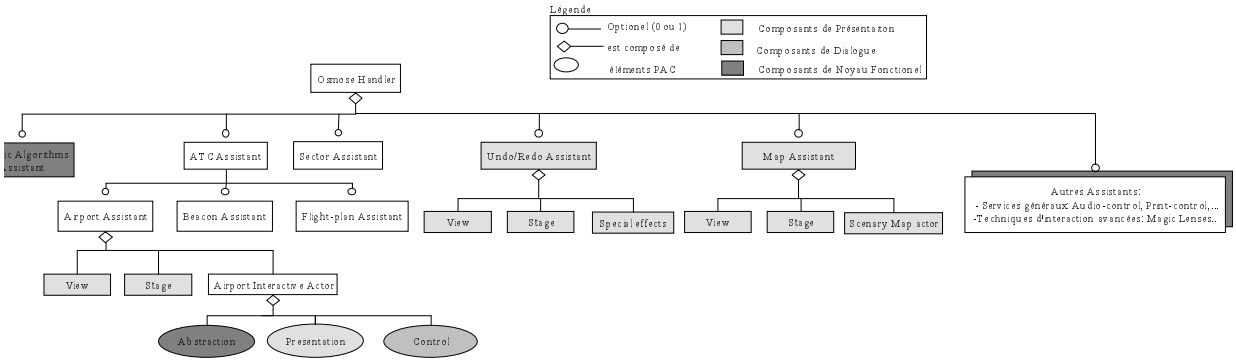

Figure 4: Un sous-ensemble de la hiérarchie des classes dans Osmose

Notons qu'un assistant ne gère pas uniquement des objets interactifs. Le *MapAssistant* par exemple ne gère quant à lui que des acteurs décors, représentant les contours géographiques de différents pays, sur lesquels l'utilisateur n'a aucune interaction. Quant à l'*Undo/Redo Assistant*, il gère des acteurs *Effets Spéciaux* qui matérialisent et retracent les interactions de l'utilisateur sur les objets interactifs édités.

Les assistants de production

Une fois décrits tous les assistants de notre application, nous avons défini les assistants de production secondant le réalisateur pour centraliser l'ensemble des vues globales et des scènes dont se servent les assistants. Ayant chacun un caractère unique, ces assistants ont également pour base le design pattern *Singleton*. Nous donnons ci-dessous le code décrivant la structure du gestionnaire des scènes :

```
class MemoStage : public ProducerAssistant {
public:
    enum KindOfStages {MapBorders, SectorsShape,Ways...};
    enum {NbKindOfStages=20};
    static MemoStage* GetInstance();
    ArrayOf <XtvStage> ArrayOfStages;
    bool DefinedStages[NbKindOfStages};
private:
    MemoStage();
    ~MemoStage();                                          10
    static MemoStage* MemoStageSingleton;
```

Cette centralisation des listes des scènes et des vues est utile dans de nombreux cas. Dans le cas des scènes, elle nous permet de les considérer comme des feuilles de calque que l'on peut superposer ou enlever à volonté, contrôlant ainsi la visualisation des informations contenues sur chaque feuille. Ainsi, pour gérer les filtres mobiles et paramétrables de visualisation, le *MagicLenses Assistant* a recours à l'objet *MemoStage* pour connaître les scènes disponibles chez les autres assistants. Le *MagicLenses Assistant* utilise ensuite une vue particulière pour visualiser ou non le contenu des scènes des autres assistants suivant la requête de l'utilisateur. De même, pour gérer l'impression des données affichées à l'écran, le *Print Assistant*, parcourra la liste des scènes fournies par l'objet *MemoStage*, et imprimera ou non (en fonction de la demande de l'utilisateur) les acteurs présents sur ces scènes.

L'assistant chargé de la gestion des vues permet de visualiser le contenu de scènes contrôlées par des assistants différents à travers une vue unique dirigée directement par le réalisateur. Cet assistant correspond donc à un objet décrivant l'ensemble des vues globales de l'application, le réalisateur répartissant ensuite certaines scènes de ses assistants sur ces vues globales. Nous donnons ci-dessous le code de la structure du gestionnaire des vues, suivi ensuite d'un exemple d'affectation de scènes à une vue.

```
class OsmoseScenary : public ProducerAssistant {
public:
/* Definition des vues globales de l'application */
    static XtvView* OsmoseView;
    static DraftView* EditorView;
    static XtvBltView* StatusView;
    static XtvBltView* BlackboardView;
    static XtvBltView* ServicesView;
    static XtvBltView* SectorsToolsView;
    static XtvBltView* ATCToolsView;                       10
    static XtvBltView* OptimizationToolsView;
    static LimitedInterStage* OsmoseMainStage;
/* Fonction d'initialisation */
    static void InitOsmoseScenary ();
private:
        virtual dummy() = 0;
};

/* Exemple d'affectation des scenes par OsmoseDirector lors de l'initialisation */
                                                           20
OsmoseScenary::EditorView->AddStage(TheSectorHandler.TheSectorStage);
OsmoseScenary::EditorView->AddStage(TheMapHandler.MapStage);
OsmoseScenary::EditorView->AddStage
                    (TheATCHandler.TheWayHandler.DataStage);
...
```

Description des acteurs

Après avoir défini les assistants composant Osmose, nous avons ensuite appliqué notre métaphore pour décrire les acteurs. Chaque acteur interactif est construit suivant le modèle PAC, comme cela est représenté par un lien de décomposition au bas de la figure 4 pour l'objet *Airport*.

L'objet *Airport* est un acteur interactif, dont l'abstraction correspond à la définition d'un aéroport en tant que donnée de l'espace aérien, la présentation correspond à un objet graphique, et la partie interactive contient une réaction au click de l'utilisateur sur la représentation graphique. L'implémentation des objets interactifs s'inspire librement du design pattern PAC, décrit dans [4]. Nous donnons ci-dessous un extrait du code décrivant la classe *Airport*, qui dérive de la classe *ATCNode* :

```
class ATCNode : public WhzPolymorph {
public:
    ATCNode ();
    ~ATCNode ();
/* Attributs liés à l'Abstraction-Partie Application */
    ATCTrafficInfos ATCInfos;
/* Attributs liés à l'Abstraction — Partie Interne */
    bool OwnsInfo;
/* Attributs liés à la Présentation */
   GraphicInfosATCNode GraphicInfos;
/* Attributs liés au Contrôle */
    static XtvReflex* SelectNode;
    WhzReflexPlug FlowReflex;
/* Fonctions liées à la Présentation */
    void Appear (XtvStage& s);
    virtual void DisplayTimeSchedule ();
    virtual void DisplayFlowSchedule ();
/* Fonctions liées à l'Abstraction — Partie Application */
    void AddDepartingPln (PLN& pln);
/* Fonctions liées au Contrôle type Client-Serveur */
    const char* GetIndic();
    void ManageInfoSensors ();
/* Fonctions liées au Contrôle type distribué */
    virtual void SelectAction (DnnEvent&);
    virtual void FlowAction (const WhzNote&);
};

class Airport : public ATCNode {
public:
     Airport ();
     ~Airport ();
/* Fonctions liées au Contrôle type Client-Serveur */
    const char* GetFullName();
/* Fonctions liées à la Présentation */
    void DisplayTimeSchedule ();
protected:
/* Attributs liés à la présentation */
static XtvIcon* AirportIcon;
```

Relations entre les éléments de la métaphore

La métaphore modélise de façon implicite les interactions entre les objets, grâce aux relations définies entre assistants, vues, scènes et acteurs, le tout sous le contrôle global du réalisateur. Nous distinguons différentes dimensions pour les communications au sein de notre application :

- les communications *horizontales* de haut niveau, nécessitant de coordonner des assistants entre eux. Ces communications sont gérées par l'intermédiaire du réalisateur *OsmoseDirector*, et de ses deux assistants de production gérant de façon globale les scènes et les vues définies pour l'ensemble de l'application. Elles s'effectuent par des transmissions de données via l'invocation de fonctions clients-serveurs. Nous avons donné un exemple de ce type de communication lorsque nous avons décrit le fonctionnement du *MagicLenses Assistant* et *Print Assistant*, qui ont besoin de connaître les scènes des autres assistants. Un autre exemple de ce type de communication concerne le calcul de l'évaluation de la charge des secteurs. Ce calcul, effectué sous le contrôle *SectorEvaluation Assistant*, fait entrer en jeu les données courantes de sectorisation qui sont gérées par le *Sector Assistant*, et les données de trafic gérées par l'ATC Assistant. Le réalisateur *OsmoseDirector* doit donc intervenir pour transmettre ces différentes données au *SectorEvaluation Assistant*. L'évaluation de la charge est donc un exemple de fonctionnalité "globale" de l'application, directement sous le contrôle d'*OsmoseDirector*, car elle fait intervenir des données gérées par plusieurs assistants différents. Nous donnons ci-dessous la ligne de code correspondant à cet appel de fonction :

```
TheSectorEval3D.EvalSectors
    (TheSectorHandler.ThePresentationData.ListSectors,
    (TheATCHandler.TheWayHandler.WayParts);
```

- les communications *horizontales* de bas-niveau, concernant les acteurs dirigés par un même assistant. Dans ce cas, on utilise pleinement les fonctionnalités de contrôle distribué disponibles dans la librairie X_{TV}-Whizz, les acteurs n'ayant pas une connaissance explicite du nombre de leurs congénères. L'édition d'un secteur est un exemple de communication horizontale. Un secteur est représenté graphiquement par un polygone, composé par une succession de sommets et de frontières. Chacune de ces entités correspond à un acteur interactif, réagissant aux actions de l'utilisateur pour les déplacer. Ainsi, lorsqu'un utilisateur clique sur un sommet, celui-ci se déplacera, mais les frontières et les secteurs adjacents seront également modifiés. La transmission de la position courante du sommet se fait par distribution, en utilisant le mécanisme de propagation de notes définis dans Whizz. Les connexions entre les différents objets composants un secteur sont définis par le *Sector Assistant*, lors de l'initialisation des secteurs. La figure 5 illustre cette distribution des données dans le cas du déplacement du sommet d'un secteur.

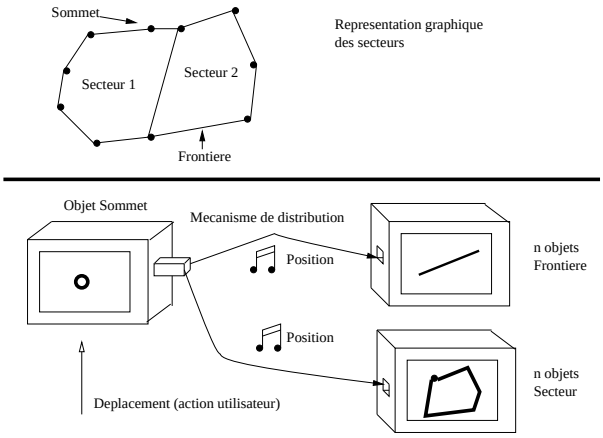

Figure 5: Représentation graphique des secteurs et liens établis dans le cas du déplacement d'un sommet

- les communications *verticales* (ou *hiérarchiques*) de haut et de bas niveau, gérées en général par l'appel de fonctions clients-serveurs dans le sens haut-bas, et assu-

rées par distribution dans le sens bas-haut. Un exemple de communication du haut vers le bas est la requête d'un assistant pour qu'un acteur apparaisse sur une scène. Un exemple de communication du bas vers le haut est l'envoi d'un message annonçant la suppression d'un acteur, lorsque celle-ci est effectuée par un utilisateur par désignation directe.

- les communications entre l'utilisateur et l'application. Elles peuvent être "globales", et s'adresser directement au réalisateur (via une boîte de dialogue), ou "spécifiques", et agir directement sur un objet interactif (mécanismes des réflexes de la boîte à outils X_{TV}).

CONCLUSION

A travers le développement d'Osmose, nous avons pu expérimenter l'utilité d'une métaphore pour aider à identifier les différentes classes composant une application interactive. Utiliser une métaphore guide les programmeurs à au moins trois niveaux :

- La métaphore étant par définition à objets, tous les avantages de cette approche, comme la réutilisabilité, la fiabilité et l'encapsulation sont entièrement supportés.
- La métaphore apporte une aide pour identifier les objets de présentation, les objets du noyau fonctionnel et leurs relations. Ceci est utile à la fois lors de l'étape de conception et lors de la phase de maintenance, car cela permet aux programmeurs de mieux appréhender l'organisation de l'application.
- La métaphore aide à répartir les tâches de programmation, en fonction de l'emploi du temps ou entre les programmeurs. L'utilisation de la métaphore facilite la phase d'intégration.

De plus, l'architecture basée sur la métaphore assure une grande flexibilité de l'application, puisque l'on peut entièrement isoler un assistant (c'est-à-dire un groupe fonctionnel), et configurer ainsi l'application avec un nombre variable d'assistants. Par conséquent, on peut adapter les fonctionnalités activables d'Osmose suivant les différents types d'utilisateurs potentiels : experts en sectorisation, experts en affectation de flux ou gestionnaires de la sectorisation.

Osmose constitue un banc d'essai grandeur réelle des techniques de structuration de logiciels interactifs existantes à ce jour, puisque les objets sont structurés de façon interne suivant le modèle PAC, organisés entre eux suivant la métaphore cinématographique, et implémentés lorsque cela été possible en s'inspirant de design patterns existants.

Néanmoins, bien qu'elles aient démontré leur utilité à travers nos développements, nous sommes conscients que les métaphores ne constituent qu'un guide pour définir l'architecture d'une application. Ainsi, les métaphores apparaissent comme un complément aux méthodes de décomposition existantes qui restent quant à elles un support indispensable à la structuration logicielle.

BIBLIOGRAPHIE

1. A metamodel for the runtime architecture of an interactive system. The UIMS Tools Developers Workshop, SIGCHI Bulletin vol. 24, n1, pp 32-37.
2. Beaudouin-Lafon M. Berteaud Y. Chatty S. Creating direct manipulation interfaces with XTV. EX'90, European conference on X Window.
3. Borne I. A Visual Programming Environment for Smalltalk. Proceedings of the 1993 IEEE Symposium on Visual Languages. IEEE Computer Society Press, pp.214-218
4. Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. Pattern-Oriented Software Architecture. A system of patterns. Wiley Publ.
5. Chatty S. Defining the behaviour of animated interfaces. Engineering for Human Computer Interaction Conference pp. 95-109, North Holland.
6. Coutaz J. PAC an implementation model for dialogue design. Proceedings of the Interact'87 conference. North Holland. pp. 431-437.
7. G.Calvary, Coutaz J., Nigay L. From a Single-User Architectural Design to PAC: a Generic Software Architecture Model for CSCW Proceedings of ACM CHI'97 pp. 242-249.
8. Fekete J.D. Un modèle multicouche pour la construction d'applications graphiques interactives, PhD thesis, Université Paris Sud.
9. Fenton J. & Beck K. Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages. OOPSLA'89 Conference Proceedings, pp.123-137.
10. Gamma E., Helm R., Johnson R. Vlissides J. Design patterns. Elements of Reusable Object-Oriented Software. Addison Wesley.
11. Hill R. Supporting concurrency, communication and synchronisation in Human-Computer Interaction. The Sassafras UIMS. Proceedings of ACM CHI'87 pp. 241-248.
12. Hudson S.E. UIMS support for direct manipulation interfaces. Computer Graphics vol.21 n2. pp120-124.
13. Hussey A. & Carrington D. Comparing two user-interface architectures: MVC and PAC. FAHCI'96, Springer Verlag.
14. Lakoff G. & Johnson M. Metaphors we live by. The University of Chicago Press.
15. Lundell J. & Anderson S. Designing a "Front Panel" for Unix: The Evolution of a Metaphor CHI95 ACM p.573-579
16. Nigay L. & Coutaz J. A design space for multimodal systems: Concurrent processing and Data fusion. Proceedings of INTERCHI'93, ACM Press, pp. 172-178.
17. Pfaff G.E. et al. User Interface Management Systems, G.E. Pfaff Ed., Eurographics Seminars, Springer Verlag.
18. Tscheligi M. & Väänänen-Vainio-Mattila. Metaphors in User Interface Development: Methods and Requirements for Effective Support. Critical Issues in User Interface System Engineering, pp. 249-263, Benyon & Palanque (Eds.), Springer Verlag.