AMUSINg : Outil d'Assemblage et d'Adaptation d'IHM

Anne-Marie Dery-Pinna

Rainbow project, I3S
930, route des Colles
BP 145
06903 Sophia-Antipolis
(+33/0)4 92 96 51 62
pinna@essi.fr

Jérémy Fierstone

Open-Plug & Rainbow, I3S
2600, route des crêtes
BP 176
06903 Sophia-Antipolis
(+33/0)4 97 24 50 68
fierston@essi.fr

RESUME

Dans ce papier nous présentons notre environnement de développement d'IHM basé sur des composants. L'IHM est considérée comme un service technique d'un composant métier à l'instar de la sécurité ou de la persistance. Le lien entre l'IHM et le composant métier est géré par un service d'interaction/coordination qui permet la reconfiguration des composants sans les modifier. Un service de fusion de composants d'IHM permet l'assemblage dynamique des composants d'IHM associés.

MOTS CLES : Assemblage, composition, fusion, composant, interaction, sémantique, vue abstraite, vue concrète.

ABSTRACT

In this paper we present our UI development environment based on components. The UI is considered as a technical service of a business component just like security or persistence. The dialog between UI and business components is managed by an interaction/coordination service that allows the reconfiguration of components without modifying them. A UI component merging service handles dynamic assembly of corresponding UI components.

CATEGORIES AND SUBJECT DESCRIPTORS: D.2.2 [SOFTWARE ENGINEERING]: Design Tools and Techniques – user interfaces.

GENERAL TERMS: Algorithms, Design, Languages.

KEYWORDS : Assembly, composition, merging, component, interaction, semantic, abstract view, concrete view.

INTRODUCTION

L'objectif de ce papier est de montrer comment les propriétés de la programmation par composants [6] (assemblage de composants, ajout et retrait dynamique de composants, interactions entre composants métiers et techniques) peuvent être exploitées pour faciliter le développement d'applications adaptables aux besoins des utilisateurs mobiles. L'idée est de faciliter, par exemple, la capacité de charger ou de décharger dynamiquement des fonctionnalités en fonction du contexte d'utilisation tout en conservant une IHM adaptée à la situation. L'originalité de notre solution est de considérer l'IHM comme un assemblage de composants d'IHM et un composant d'IHM comme un composant technique au service d'un composant métier de la même façon que la prise en charge de la sécurité et de la persistance [2]. Cette solution permet de changer dynamiquement le composant d'IHM afin de l'adapter à un nouveau support ou à un nouveau contexte d'utilisation. De plus, l'IHM peut évoluer en fonction du contexte d'exécution d'une application par ajout et retrait de composants d'IHM correspondants aux évolutions fonctionnelles. Cependant, pour conserver les propriétés d'assemblage de composants dans le cadre d'assemblage d'IHM, il est indispensable de mettre en place des règles de composition d'IHM très spécifiques. Dans cet article, nous ne détaillerons pas la façon de composer les composants métiers et techniques [4], nous nous focaliserons sur l'assemblage dynamique des composants d'IHM qui leur sont associés. Aussi la partie suivante illustre-t-elle le type d'application visée, puis des règles de compositions statiques et dynamiques de composants d'interface sont développées par la suite.

ILLUSTRATION DES BESOINS PAR L'EXEMPLE

Prenons comme illustration le scénario d'un fournisseur d'applications de gestion construites pour répondre aux besoins spécifiques de chaque client. Notons en particulier, le cas du commercial qui intervient souvent sur le site de l'acheteur et dont les tâches peuvent dynamiquement évoluer en fonction des besoins de ce dernier. Cet exemple est issu d'une étude de cas réelle menée dans le cadre d'un contrat RNTL avec des partenaires industriels. L'assemblage de composants métiers (imposé par les applications ASP de ce type) permet de rapidement construire l'application demandée à partir d'une bibliothèque de composants prédéfinis (tels que Customer, Salesperson, Bill, et Stock). A la conception, plusieurs composants sont assemblés statiquement dans un objectif fonctionnel précis. Par exemple, le commercial qui part en déplacement peut charger sur son assistant personnel les composants Catalog et Order. A l'exécution, un nouvel assemblage peut s'avérer utile pour ajouter dynamiquement un composant nécessaire pour répondre à un

contexte d'exécution particulier. Par exemple, le commercial a besoin de connaître l'email d'un client particulier pour annuler un rendez-vous, il demande alors le chargement sur son terminal du composant Address (contenant les coordonnées d'un client) qui doit être dynamiquement assemblé avec le composant Customer afin d'enrichir l'application de base et ainsi associer à un client ses coordonnées. De la même façon, il doit pouvoir supprimer certaines fonctionnalités pour alléger la mémoire si nécessaire. L'IHM doit s'adapter à ce type d'évolutions fonctionnelles d'une application, aussi bien statiquement que dynamiquement. Si les modèles à composants classiques (EJB, CCM, .Net) permettent actuellement tous les assemblages statiques, voire le chargement dynamique de composants liés statiquement (c'est-à-dire au moment de l'utilisation des composants) tels que les travaux sur les DLL, seuls des travaux de recherche sur les composants tels que [1,2] permettent l'ajout et le retrait dynamique de composants non prévus à la conception. Les modèles à services tels que OSGi permettent la découverte de nouveaux services dynamiquement mais ne possèdent pas les propriétés d'assemblage des modèles à composants. Actuellement les travaux de recherche sur l'adaptabilité des IHMs concernent principalement la plasticité et la prise en compte du contexte [3]. Ils n'abordent pas le problème en terme de fusion de composants d'interfaces. En appliquant les technologies et propriétés inhérentes à l'assemblage de composants métiers à la construction d'IHM, nous devons obtenir une meilleure adaptation et réutilisation des IHM existantes.

Notre proposition se base sur un service d'interactions [2] qui prépare les composants pour interagir entre eux et un serveur d'interactions qui permet de définir des schémas d'interactions réutilisables pour permettre la liaison et la déliaison de composants métiers. L'originalité de cette proposition est de considérer l'IHM comme une propriété orthogonale (un composant technique associé au composant métier) et d'ainsi pouvoir éclater l'IHM globale en plusieurs composants d'IHM qui peuvent être fusionnés.

Dans la suite nous développons la spécificité de ces composants d'IHM et les algorithmes de fusion que nous avons mis en place afin de permettre l'ajout statique et dynamique de composants d'IHM à une IHM existante. Les aspects concernant la mise en œuvre d'interactions et du serveur d'interactions dans le cadre d'applications mobiles ne sont pas abordés dans cet article et font l'objet de travaux spécifiques du projet RAINBOW.

COMPOSANTS D'IHM FUSIONNABLES

Dans cette partie nous détaillons le cœur de l'environnement de développement d'IHM, Amusing [19]. La première section décrit les composants d'IHM et les trois sections suivantes les différentes mises en œuvre de la fusion de tels composants.

Qu'Appelle-t-on un Composant d'IHM ?

Dans notre modèle de composants, un composant métier peut être lié à un ou plusieurs composant(s) d'IHM à l'aide d'interactions logicielles. Une interaction représente le contrôleur de communication (comme dans le modèle Arch). L'environnement de développement réalisé comprend un serveur d'interactions permettant d'enregistrer et d'utiliser des schémas d'interaction / coordination pour assembler des composants métiers et pour les faire interagir avec leurs composants techniques. Les schémas d'interaction sont spécifiés en ISL [2].

Afin de garantir une indépendance du support lors de l'assemblage, le composant d'IHM doit être décrit de manière abstraite. Nous proposons dans notre modèle un mini langage de spécification d'IHM indépendant des plates-formes cibles, SUNML (pour *Simple Unified Natural Markup Language*) détaillé en 3.3, dans la lignée XML. Un composant d'IHM est décrit dans ce langage puis instancié sous forme d'arbre abstrait représentant la structure abstraite de l'IHM [7]. Cette vue abstraite peut être ensuite projetée vers une ou plusieurs vues concrètes à l'aide d'un renderer (actuellement Swing ou Vocal). La vue abstraite et la ou les vues concrètes sont liées entre-elles. La vue abstraite est utilisée, dans notre architecture, comme un méta objet représentant à la fois la structure abstraite et conservant les données des IHM concrètes (valeurs saisies par l'utilisateur) qui doivent être synchronisées en permanence avec cette vue abstraite. Cette indirection supplémentaire apporte la flexibilité utile à la composition des IHM ainsi qu'à leur mobilité. La vue abstraite est liée au composant métier qui ne suppose rien de la manière dont est projeté un widget abstrait (WA[1]). Il est également possible de lier directement le composant métier à la vue concrète. Un tel lien direct peut être utilisé pour optimiser l'exécution lorsqu'un assemblage est figé lors de la conception. Mais dans ce cas, adaptation et assemblage dynamiques ne sont plus possibles. Le schéma suivant présente les éléments de base d'un composant d'IHM obtenu par réification de la description SUNML et projection de l'arbre abstrait :

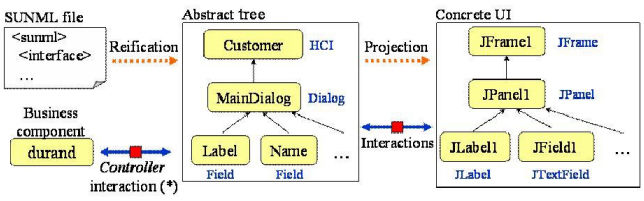

Figure 1. Architecture visée

[1] Dans notre terminologie, le terme *widget abstrait* correspond au terme *objet d'interaction abstrait* dans le domaine des IHM.

Les Algorithmes de Fusion

La fusion de composants d'interface est réalisée à partir des vues abstraites et permet d'obtenir une nouvelle vue abstraite. Comme toute vue abstraite, celle-ci peut être utilisée dans une autre fusion ou pour une projection. L'opérateur *add* inclut un composant d'IHM dans un autre composant d'IHM par ajout d'un arbre de WA dans un autre. L'opérateur *select* permet de n'extraire d'un composant d'IHM qu'une sous partie en récupérant un sous arbre à partir de son chemin (comme dans une arborescence de fichiers). L'opérateur *union* plus précis que l'ajout permet une fusion de deux composants d'IHM en supprimant les redondances d'information. L'union fusionne des arbres en unifiant les abstraits sémantiquement identiques. Dans la Figure 2 le nom du client a été identifié comme partie commune pouvant être représentée par un seul WA. A l'inverse, l'opérateur *intersect* permet de récupérer dans plusieurs composants d'IHM la partie commune. Il retourne l'ensemble des WA sémantiquement identiques. Dans la figure 3, les WA correspondant respectivement au widget concret (WC) *nom* et *name* sont identifiés comme devant être un WA commun.

Bien que ces algorithmes s'appliquent à des arbres, ceux-ci ne se simplifient pas en simples algorithmes de fusion d'arbres ordonnés. En effet, pour déterminer l'identité entre deux WA (éléments de l'arbre), nous avons introduit la notion de sémantique d'un WA, spécifiée sous forme d'un attribut *semantic* décorant l'arbre abstrait. Actuellement cet attribut est spécifié par le développeur et éventuellement modifiable dynamiquement avant fusion afin de spécifier une équivalence sémantique entre des WA hétérogènes (issus de développeurs différents, dans des langues différentes, etc.). L'automatisation de ces algorithmes pour des applications à grande échelle nécessite un travail en amont pour pouvoir en partie générer les attributs « semantic » correctement. Les pistes envisagées sont de partir d'ontologies associées à des domaines d'applications spécifiques et plus simplement, , de la cohérence avec les composants métiers. Sur ce dernier point, cet attribut pourrait être généré par défaut à partir de la description fonctionnelle du composant métier (représentant les mêmes informations au niveau du composant métier)...

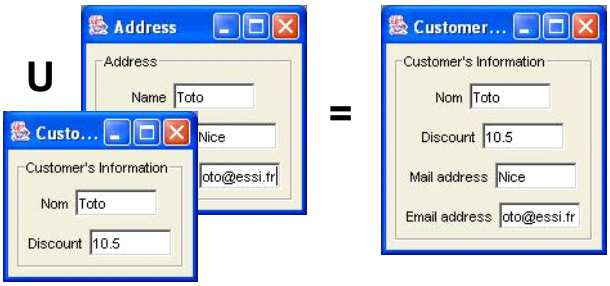

Figure 2. Fusion par l'opérateur d'union

Les opérations de fusion ne respectent pas les propriétés de commutativité et d'associativité. Deux WA déclarés comme étant sémantiquement identiques ne contiennent pas forcément les mêmes données (modifiables par l'utilisateur) ni les mêmes caractéristiques (un élément d'un menu peut être sémantiquement identique à un bouton dans une autre IHM). Un choix entre ces deux instances de WA étant nécessaire, dans les algorithmes, l'instance correspondant au premier arbre en argument est retournée. Par exemple, les instances de WA correspondant aux WC *nom* et *name* (cf. Figures 2 et 3) ont la même sémantique, l'union et l'intersection privilégiant le premier argument donnent l'instance *nom* comme résultat. Les résultats des fusions dépendant fortement de l'ordre des composants à fusionner, la propriété ensembliste $A + B = (A \cup B) + (A \cap B)$ se réécrit par la propriété suivante que respectent les opérateurs de fusion :

$$A + B = (A \cup B) + (B \bullet A)$$

La fusion dynamique des IHM est possible en utilisant directement l'API d'Amusing ou en spécifiant la fusion à l'aide d'un langage de fusion, WML (pour Widget Merging Language) basé sur XML. Ce langage permet de spécifier la manière de fusionner des composants d'IHM, donnés en argument d'entrée, et de retourner le résultat (comme une méthode).

Les Interactions au Service des Fusions Dynamiques des Vues Abstraites

Lorsqu'un nouveau composant métier est ajouté dynamiquement dans l'application, les composants d'IHM correspondants doivent être également ajoutés dynamiquement à l'IHM en cours d'exécution. Il est alors capital de gérer l'assemblage structurel dynamiquement en utilisant la représentation abstraite de l'IHM. La fusion (cf. 3.2) peut gérer l'assemblage dynamique par manipulation des vues abstraites. Ce service de fusion peut être invoqué par l'utilisateur à l'exécution. De plus, afin d'étendre l'utilisabilité de ce service, on peut utiliser les interactions logicielles afin d'automatiser l'ajout d'une nouvelle vue abstraite lors de l'ajout d'un composant. Ainsi, l'interaction invoque dynamiquement le service de fusion avec les règles de fusion prédéfinies. La règle ISL suivante illustre cette fusion automatique avec le cas typique d'un composant, représentant statiquement une liste d'éléments quelconques, adaptée dynamiquement par cette règle pouvoir recevoir dynamiquement un composant Customer :

```
interaction CustomerListMerging(AUI cus-
tomerList) implements MergingService {
  customerList.add(AUI customer) ->
cus-
tomerList.add(customer.select("*/name",
"*/discount"))
}
```

Dans cet exemple, tout composant *Customer* ajouté dans la liste est fusionné avec la structure abstraite de la liste en utilisant l'opérateur *select*. Le comportement du com-

posant doit être également fusionné avec les comportements des autres composants grâce aux interactions logicielles. En effet, alors que les algorithmes de fusion ne proposent qu'une fusion structurelle des IHM, leurs comportements dépendent de cette fusion. Dans cet exemple, on voudrait également fusionner la sauvegarde de tous les clients lors de la sauvegarde de la liste (bouton *save*). Cette fusion comportementale n'est possible qu'après la fusion structurelle et peut être réalisée à l'aide du schéma d'interaction suivant :

```
interaction CustomerListMerging(AUI
list,
       sequence<AbstractUI> customers) {
  list.manageEvent(AbstractEvent e) ->
   if(e.getDesc() == "save") {
     foreach(customers, customer) {
       customer.manageEvent(e)
     }
   }
}
```

Chaque composant C*ustomer* fusionné avec le composant *List* doit être lié par cette interaction qui stipule que chaque réception du message *manageEvent* inhérent à la réception d'un événement concret résultant du click sur le bouton *save*, doit correspondre au nouveau comportement spécifié en partie droite de la règle, en l'occurrence l'envoi de ce même message sur chaque composant client (dans notre exemple, l'IHM du composant Customer appelle la méthode de sauvegarde lors de sa fermeture).

CONCLUSION

Le travail présenté dans cet article est intégré dans un environnement de développement d'IHM ouvert Amusing [28]. Avec cet environnement, nous avons déjà développé plusieurs applications à base de composants : un jeu de bataille navale adaptable aux utilisateurs (voyants et non voyants) et une application de gestion pour commerciaux mobiles. Ces expérimentations nous ont permis de valider les algorithmes de fusion. De premières expérimentations prometteuses ont été faites pour automatiser en partie des adaptations à partir de graphes d'états de contextes [9], cependant cette approche expérimentale doit être poursuivie plus avant. Nous travaillons également à l'intégration d'un WYSIWYG pour assembler des composants de manière visuelle afin d'aider la personne en charge de l'assemblage des composants ou services de fusionner les composants en lui permettant de spécifier les WA qu'il considère communs par des relations d'équivalence sémantique. Une des principales difficultés de ce travail est d'avoir une représentation pertinente de l'arbre abstrait.

Si l'adaptation aux composants métiers paraît actuellement facilement atteignable, un autre de nos objectifs étant l'adaptation aux supports est en cours d'étude. Même si l'environnement intègre des renderers, il n'intègre actuellement aucun moteur d'adaptation et ne prend pas en compte la plasticité des interfaces. Une première approche envisagée est de se connecter à un moteur d'adaptation existant en traduisant l'arbre abstrait résultant de la fusion sous forme d'arbres abstraits compris par le moteur cible (RIML, par exemple). SUNML étant un sous langage à balise, cette traduction doit être simple. Des travaux de recherche plus complets doivent être envisagés pour déterminer comment les règles de fusion proposées peuvent conserver ou non la plasticité des interfaces et sous quelles limites (gamme de supports…).

REMERCIEMENTS

Ce travail nécessite une expertise importante en terme de besoins applicatifs, nous tenons à remercier les partenaires du RNTL Aspect pour leur apport sur ce point.

REFERENCES

1. L. Berger. Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés: le modèle MICADO. PhD thesis, Université de Nice-Sophia Antipolis, octobre 2001.
2. M. Blay-Fornarino, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Un service d'interactions : principes et implémentation. Revue TSI, 2004.
3. Calvary, J. Coutaz, D. Thevenin. A Unifying Reference Framework for the Development of Plastic User Interfaces. IFIP WG2.7 (13.2) Working Conference, EHCI01,Toronto, May 2001, Springer Verlag Publ., LNCS 2254, M. Reed Little, L. Nigay Eds, pp.173-192.
4. Nano, M. Blay-Fornarino, A-M. Dery, and M. Riveill. An abstract model for integrating and composing services in component platforms. In ECOOP'2002, Malaga, June 10, 2002.
5. A-M. Pinna-Dery, J. Fierstone, M. Riveill, E. Picard. User Interface: a Technical Component in Component-Based Models in Response to Human Computer Interaction Adaptation. Rapport de recherche, février 2004.
6. M. Riveill and P. Merle. La programmation par composants. In Techniques de l'Ingénieur - Informatique, H2759, décembre 2000.
7. D. Thevenin. La plasticité des Interfaces Homme Machine : une approche, un outil. Computer Science Ph.D Thesis, Joseph Fourier University - Grenoble I, 2001, 212 p.
8. D. Thevenin. « Adaptation en Interaction Homme-Machine: Le cas de la Plasticité ». Ph.D. thesis, Université Joseph Fourier, Grenoble, 21 December 2001.
9. Dey, A.K., Salber, D. Abowd, G.D. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications in Human-Computer Interaction (HCI) Journal, Vol. 16(2-4), 2001, pp. 97-166.