Réification et réplication dans les interfaces graphiques: le toolkit Ubit

Eric Lecolinet

ENST INFRES / CNRS URA 820
46 rue Barrault
75013 Paris, France
elc@enst.fr - www.enst.fr/~elc

RESUME

Cet article présente une nouvelle boîte à outils graphique, basée sur un modèle de briques logicielles facilement combinables. Toutes les briques peuvent être partagées, y compris les gadgets graphiques afin de simplifier le contrôle de l'interaction et la synchronisation de vues multiples. Les interfaces peuvent indifféremment être spécifiées de manière déclarative ou procédurale en langage C++.

MOTS CLES : toolkits graphiques, langages déclaratifs de description, réification, ubiquité, vues multiples.

INTRODUCTION

Les interfaces graphiques (ou GUIs) sont non seulement difficiles à concevoir (du fait de la multiplicité des aspects humains et logiciels qu'il faut prendre en compte) mais aussi à implémenter [6]. Ce dernier point semble désormais être considéré comme une sorte d'écueil incontournable. Il est ainsi frappant de voir l'importance des travaux effectués ces dix dernières années pour concevoir des outils de création d'interfaces graphiques permettant d'éviter (ou de minimiser) la phase de codage au niveau de la boîte à outils graphique. A l'opposé, les travaux portant sur les principes de conception des boîtes à outils graphiques sont plutôt rares (la plupart des toolkits récents n'étant basés que sur des principes relativement classiques (par exemple, Java Swing.)

Les environnements de développement d'interfaces graphiques, ont certes largement fait leurs preuves (surtout en ce qui concerne les générateurs d'interface interactifs, les approches plus évoluées comme les environnements « model based » n'ayant pas encore dépassé le cadre des laboratoires [8]). Les outils interactifs présentent pourtant plusieurs inconvénients :

- Ils sont principalement conçus pour créer des interfaces « statiques » essentiellement composées de menus et boîtes de dialogue. Ils fournissent peu ou pas d'aide pour créer des objets graphiques spécifiques au domaine d'application, surtout si ces objets évoluent dynamiquement (i.e. sont crées, modifiés, détruits en fonction de l'interaction ou d'autres facteurs externes à l'application.)

- Ils fixent la disposition spatiale et la présentation de l'interface trop tôt dans le processus de conception. Toute modification ultérieure entraînera un important travail de réimplémentation. De plus, il est souvent difficile de spécifier des présentations graphiques qui s'adaptent au contexte de l'application ou aux préférences de l'utilisateur (par exemple la taille des polices selon la résolution de l'écran, le changement de taille interactif des fenêtres, etc.)

- Ils imposent une « fracture » entre la partie présentation (créée interactivement) et la partie interaction (généralement codée à l'aide d'un langage de programmation classique) [1]. Cette fracture rompt le processus de création itératif des interfaces complexes : les comportements doivent généralement être codés textuellement, ceci risquant d'interdire une utilisation ultérieure de l'outil interactif pour remodeler la présentation de l'interface (la plupart des outils n'étant pas isomorphes en ce sens que le code généré ne peut être librement modifié, sauf dans certains systèmes expérimentaux [3])

- Aucun outil de développement (ou toolkit graphique) courant ne propose de solution viable et générale pour implémenter de nouvelles techniques de visualisation et d'interaction (en particulier celles qui concernent la Visualisation de l'Information.) Ceci constitue un frein certain à la popularisation de ces techniques, et plus généralement, à la créativité des interfaces homme/machine.

Il nous semble que nous avons atteint une situation paradoxale, où il a fallu créer des outils de développement très complexes pour remédier aux défauts et aux lacunes des toolkits graphiques (dont l'utilisation directe via un langage de programmation est généralement excessivement compliquée.) Mais en même temps, ces outils de développement restreignent fortement notre liberté d'action et limitent les interfaces graphiques à des stéréotypes qui sont en deçà des possibilités des technologies actuelles. Il est d'autre part

raisonnable de penser que les langages de programmation textuels apportent des possibilités d'abstraction et de généralisation qui sont largement supérieures à ce que les outils de programmation visuelle peuvent généralement offrir (sauf cas particuliers où ces outils tendent d'ailleurs à devenir aussi complexes d'utilisation que leurs équivalents textuels.)

Pour toutes ces raisons, nous pensons qu'il est maintenant souhaitable de repenser les principes qui gouvernent les couches basses et intermédiaires des interfaces graphiques, c'est-à-dire les toolkits graphiques. Ceci n'est nullement contradictoire avec les efforts de développements d'environnements de développements sophistiqués. Mais ces derniers devraient à notre avis s'attacher à résoudre des tâches de plus haut niveau (dans la logique des outils « model based ») au lieu de servir à compenser les lacunes des toolkits graphiques. Cette communication présente les principes d'un nouveau toolkit C++, nommé « Ubit » (pour Ubiquitous Brick Interaction Toolkit) qui tente de résoudre certains problèmes des toolkits classiques.

LE TOOLKIT « UBIT »

On peut tout d'abord se demander pourquoi les toolkits graphiques sont si difficiles à utiliser. Il est à ce propos étonnant de voir des néophytes créer des pages Web en « tapant du HTML » alors que les mêmes personnes s'estimeraient incapables d'écrire une petite interface avec un toolkit graphique classique (précisons que nous nous limitons ici aux aspects liés aux difficultés d'implémentation.) Il est de plus courant que les personnes qui créent ce type de pages préfèrent « coder » directement en HTML plutôt que d'utiliser des outils spécialisés ou qu'elles utilisent une stratégie mixte. Pourquoi n'en va-t'il pas de même avec les toolkits graphiques ?

Nous proposons les deux réponses suivantes. D'une part la gestion du contrôle est évidemment plus complexe dans les interfaces graphiques classiques. Ces aspects sont spécifiquement pris en compte dans le toolkit Ubit via des techniques de *réification*, de *partage* de composants et *d'ubiquité* décrites dans la section suivante. D'autre part, les « mark-up languages » tels HTML proposent un modèle simple et cohérent de disposition spatiale des composants textuels et iconiques. Ce modèle est d'autre part basé sur une spécification *déclarative* des données, contrairement aux toolkits graphiques, qui nécessitent presque toujours une spécification procédurale. Ce point est peut-être déterminant dans la mesure ou les spécifications déclaratives sont généralement plus compactes et plus proches du résultat graphique escompté (du moins en ce qui concerne des données telles que textes, images, etc.) C'est pourquoi le toolkit Ubit propose deux APIs C++, qui sont équivalentes et interchangeables. La première est une API objet classique qui ressemble un peu à celle du toolkit graphique Java AWT. La seconde API, également en C++, permet des descriptions déclaratives qui rappellent celles des mark-up languages.

Le toolkit Ubit permet enfin de créer facilement des objets graphiques spécifiques destinés à un domaine d'application particulier. Il est en effet essentiellement composé d'objets très génériques, appelés *briques de base*, qu'il est possible de combiner librement de manière à obtenir une grande variété de composants graphiques spécialisés.

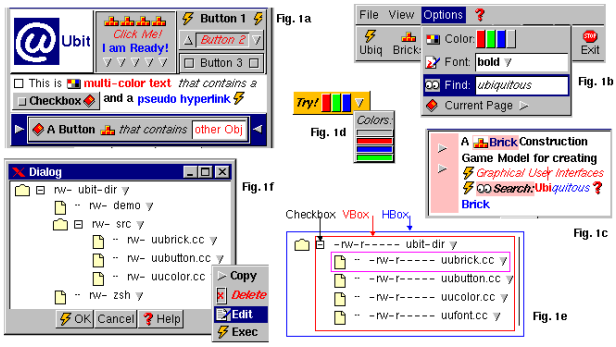

Figure 1: Composition de briques graphiques

ARCHITECTURE ET PROPRIETES

Réification, composition, généricité

L'architecture du toolkit est fondée sur la notion de briques dynamiquement combinables. Une *brique de base* est un *petit* objet qui implémente et qui contrôle une fonctionnalité bien précise (par opposition avec la plupart des toolkits classiques où les composants graphiques gèrent souvent un nombre important de propriétés très diversifiées.) Les briques Ubit ne se limitent pas aux composants graphiques visibles (les traditionnels *widgets* ou *controls*) mais permettent aussi de représenter n'importe quelle propriété matérielle (une décoration, des ombrages, du texte, des images, des symboles…) ou immatérielle (une couleur, un type de police, un comportement, une condition, une fonction de callback.) Tous les composants d'une interface graphiques sont ainsi représentés d'une manière similaire, héritent d'une même classe de base, et se comportent de la même manière vis à vis de la composition.

Les *gadgets* (qui correspondent aux objets graphiques traditionnels tels que boutons, menus, boîtes de dialogue…) sont tous créés à partir d'une brique de base particulière appelée **Ubox**. Cet objet possède les deux fonctionnalités suivantes :

- Il sert de container générique (au sens des « Design Patterns ») pouvant contenir n'importe quelle combinaison de briques de n'importe quel type,

- Il implémente un large ensemble de comportements standards qui peuvent activés ou désactivés dynamiquement suivant l'effet recherché.

Ceci implique plusieurs propriétés intéressantes. D'abord, la plupart des gadgets ne sont en fait que des variantes d'une classe unique configurée de manière ad hoc. Ceci simplifie considérablement l'implémentation du toolkit et l'API que doit utiliser le programmeur (quasi identique pour tous les gadgets standards.) D'autre part, un gadget peut contenir n'importe quelle combinaison d'objets « matériels » ou « immatériels », y compris d'autres gadgets. Cette propriété permet de créer une large variété de composants graphiques par simple composition (voir Figure 1) sans avoir à définir de nouvelles classes. Ce dernier aspect est important dans la mesure où la plupart des objets spécifiques sont rarement réutilisables (ce qui peut conduire à avoir quasiment autant d'instances que de classes, si les objets ne sont pas suffisamment flexibles.) Ce point rejoint un des aspects des systèmes objets de type « prototype-instance » (par exemple Amulet [7].)

Spécification pseudo-déclarative

Outre le fait qu'elle s'opère dynamiquement, la composition d'objets peut s'exprimer de manière procédurale (via une classique méthode *add*) ou déclarative. L'API déclarative repose sur un mécanisme d'addition obtenu par surcharge de l'opérateur + :

```
utext(ucallback(foo, UState::action)
      + upix(« ubit.xpm »)
      + UColor::red
      + « Click Me »
      + Ufont::bold + UColor::blue
      + « I am Ready ! »
      + ubutton(USymbol::down
                + umenu(ubutton(…)
                        + utext(…)
                        + …
      )         )       )
```

L'exemple précédent montre ainsi comment créer une zone de texte contenant un pixmap, un texte de couleur rouge, un texte en gras de couleur bleue et un bouton contenant une flèche vers le bas qui ouvre un menu contextuel. Cette zone de texte est également active : une fonction de callback (qui pourrait également être une méthode d'un objet quelconque) sera appelée chaque fois qu'une certaine condition sera vérifiée.

Il est intéressant de remarquer que l'ordre des enfants importe : d'une part, les composants matériels sont affichés dans un ordre cohérent (qui dépend de l'algorithme de disposition spatiale utilisé par le container), d'autre part, les composants immatériels (par exemple la couleur) ne s'appliquent qu'à partir de l'endroit où ils sont définis dans la liste et jusqu'à l'endroit où ils sont éventuellement redéfinis. Bien que la syntaxe soit différente, ce type de spécification est en fait assez proche de langages tels que *troff* ou *html*.

Outre la structure de l'interface et la présentation qui en découle, ce type de spécification permet également de d'imposer des *comportements* ou des *dépendances implicites*. Le toolkit implémente tout d'abord un certain nombre de comportements contextuels qui dépendent des relations de parenté entre gadgets. Ainsi, un menu fils d'un bouton est-il automatiquement ouvert quand ce bouton est activé (il en irait de même avec un bouton et une boîte de dialogue ou d'autres combinaisons légitimes.) De plus, ce comportement varie suivant que le bouton est isolé ou fait partie d'une barre de menu (l'apparence du bouton change également dans ce cas.) Ces comportements contextuels permettent de simplifier la spécification de l'interaction et de réduire le nombre de classes d'objets à manipuler (il n'y a qu'un seul type de bouton ou de menu dans Ubit, qui apparaîtront et se comporteront de manière différente selon leur contexte.)

Les briques Ubit peuvent aussi être « conditionnées » par des briques spécifiques qui expriment des conditions d'état du gadget courant ou d'un autre gadget ou encore une valeur booléenne résultant d'un calcul quelconque. Les briques conditionnées sont actives (ou apparaissent) uniquement lorsque cette condition est vérifiée. On peut ainsi par exemple spécifier de manière déclarative le fait qu'un objet change de couleur quand la souris passe dessus, qu'un gadget apparaisse ou disparaisse suivant la valeur d'une *checkbox*, etc.

Styles, disposition spatiale et « fantômes »

Les Styles sont des objets particuliers qui définissent l'apparence des gadgets dans tous les états standards. Les Styles sont en fait des collections de spécifications graphiques qui s'appliquent par défaut en l'absence de spécifications explicites. L'utilisation des styles permet de partager l'essentiel des ressources graphiques (fontes, couleurs…) entre les objets et simplifie le paramétrage des spécificités graphiques des applications (on pourrait ainsi redéfinir le « look » standard du toolkit)

La disposition spatiale s'effectue au moyen d'objets spécialisés qui fonctionnent suivant une logique similaire aux opérateurs du langage HTML en ce qui concerne les alignements (gauche, droite, centré, justifié, etc.). La version courante du toolkit intègre la disposition en mode bloc vertical ou horizontal, en mode page (n'importe quel type d'objet pouvant être incorporé dans le flot naturel de la page) ou sous forme de tableaux. Il est également possible de positionner des gadgets à un endroit arbitraire, ceux-ci apparaissant alors au-dessus des autres objets. De plus, ces gadgets « flottants » peuvent éventuellement être transparents.

Enfin, les gadgets fantômes sont des objets standard dont la plupart des propriétés graphiques de présentation ont été désactivées (en leur rajoutant dynamiquement une brique particulière qui provoque cet effet). Ainsi, ces objets ne disposent pas d'encadrement ni de décoration spécifique ce qui les rend indiscernables. Cependant, ces objets continuent à interagir normalement. Ceci permet de créer facilement des interacteurs qui semblent avoir des formes quelconques.

Partage et ubiquité

Une conséquence importante de la réification de toutes les composantes ou propriétés d'une interface graphique est que ces objets peuvent être partagés par plusieurs parents et contrôler leur apparence et leur comportement. Un objet couleur pourrait ainsi être partagé par de nombreux gadgets parents. Un changement de valeur effectué sur cet objet aurait pour conséquence une mise à jour immédiate et implicite de tous ses parents. Ce dispositif permet donc faciliter le paramétrage de multiples objets graphiques ou leur synchronisation.

Cette possibilité reste vraie pour les gadgets « partagés » (ayant plusieurs parents.) Deux cas doivent alors être distingués. Lorsque les menus ou les boîtes de dialogue ont plusieurs parents, des comportements implicites sont activés lorsque cela a un sens. Ces comportements implicites sont déterminés dynamiquement et peuvent donc dépendre du parent activé (par exemple le même menu s'ouvrira différemment suivant que son bouton parent est incorporé ou non dans une barre de menu.)

Le second cas concerne les autres gadgets (ceux qui ne possèdent pas de fenêtre propre.) Ces objets sont alors visuellement répliqués dans leurs parents (de telle sorte qu'ils pourront apparaître autant de fois qu'ils ont de parents, d'où le terme *d'ubiquité*.) Cette réplication n'implique pas de duplication mémoire mais un partage des données de l'objet concerné. Cette dernière propriété peut être utilisée pour synchroniser des objets à états (par exemple la même *checkbox* apparaissant à des endroits différents de l'application) ou des vues entières (par exemple du texte composite, incluant des liens, des images, etc.), les enfants d'objets répliqués étant eux-mêmes répliqués. Cette technique est d'autant plus utile qu'elle assure une cohérence logique de l'application mais n'impose pas que toutes les vues soient strictement identiques. Certaines caractéristiques graphiques telles que les fontes, les couleurs, les facteurs d'échelle, la disposition spatiale peuvent éventuellement avoir des valeurs différentes pour chacune des vues.

CONCLUSION

Bien que l'objectif final du projet soit plutôt de fournir de nouvelles techniques d'interaction et de visualisation (en particulier des techniques de visualisation de l'information), nous nous sommes pour l'instant essentiellement limité à l'implémentation des objets d'une boîte à outils classique. Cette démarche visait d'une part à démontrer la viabilité du modèle par rapport à une base connue, et d'autre part à fournir un toolkit généraliste, extensible mais « autosuffisant.» Ce dernier aspect nous paraît important dans la mesure où il conditionne l'utilisation – et donc l'évaluation – du toolkit par des tiers pour réaliser des applications en « vraie grandeur.»

La version actuelle du toolkit, implémentée en C++ et fonctionnant sous X-Window, est librement accessible à l'URL : www.enst .fr/~elc. Une version MS-Windows sera réalisée ultérieurement. D'autre part, cette brève communication ne pouvant détailler tous les aspects du toolkit Ubit ni offrir une comparaison substantielle à l'état de l'art, des informations complémentaires pourront être obtenues à la référence [4] ou à l'URL déjà citée.

Un des aspects intéressants du modèle est qu'il permet de spécifier une part importante des comportements et des interactions entre objets de manière déclarative (soit explicitement, soit implicitement dans le cas des dépendances structurelles du DAG d'instanciation.) Cette propriété simplifie la programmation et il est donc vraisemblable qu'elle réduise d'autant les risques d'erreur lors de l'implémentation d'interfaces. D'autre part, cet aspect pourrait également être mis à profit pour effectuer des contrôles automatiques de validité et de cohérence des interfaces réalisées ou pour faciliter l'implémentation d'environnement de développement de type « model-based ». Nous comptons d'ailleurs adapter l'environnement expérimental de développement XXL [3] au toolkit Ubit afin de tester certaines de ces idées.

BIBLIOGRAPHIE

1. Coutaz J. *Interfaces homme-ordinateur*. Dunod 1990.
2. Esteban O., Chatty S., Palanque P. Whizz'ed, a visual programming enviroment for building highly interactive software. In proc. *Interact* 1995.
3. Lecolinet E., XXL : A Dual Approach for Building User Interfaces. In Proc. *UIST*, 99-108, 1996.
4. Lecolinet E., A Brick Construction Game Model for Creating GUIs: The Ubit Toolkit. In proc. *Interact*,. 510-518, 1999.
5. Linton M., Tang S., Churchill S. Redisplay in Fresco. In *The* X *Ressource*, 9, 63-69, 1994.
6. Myers B.A. User Software Tools. In *TOCHI* 2(1) 1995.
7. Myers B.A. et al. The Amulet Environment. In *IEEE Transactions on Software Engineering*, 23 (6), 1997.

8. Vanderdonckt J. (ed.) *Computer-aided design of user interfaces (CADUI)* Presses Univ. De Namur. 1996.