Projet de Programmation Comparée : "Interfaces Utilisateurs"

5 mars 2013

1 Qu'est ce qu'une interface utilisateur et comment en programmer

1.1 Définition des concepts liés à la notion d'interface utilisateur

Qui sont les utilisateurs ?

Avant toute analyse ou réflexion, il est important de définir qui sont les utilisateurs du système considéré. Nous distinguerons deux catégories d'utilisateurs et nous utiliserons la terminologie suivante :

- **Programmeur :** il est notre cible directe, c'est lui qui utilisera les outils que nous allons développer afin de construire des interfaces utilisateurs (nous parlerons ici essentiellement d'interfaces graphiques) pour ses applications.

- **Utilisateur application :** il est l'utilisateur des produits développés par le programmeur, il est n'est pas notre cible directe mais il est important de ne pas l'oublier.

Une approche orientée utilisateur sera adoptée lors de la conception, les besoins étant différents pour les deux catégories d'utilisateurs considérées, nous ferons bien attention de prendre en compte les deux points de vue.

En particulier, celui de l'utilisateur application doit être pris en compte même s'il sera la cible du système développé par le programmeur et non directement celle du notre. Ses besoins doivent nous permettre de mieux déterminer les éléments nécessaires aux programmeurs pour satisfaire à ses exigences envers le produit fini.

Conception - génie logiciel

La conception d'une interface utilisateur doit être centrée autour des deux catégories d'acteurs définies précédemment, l'utilisation du modèle de conception centré utilisateur semble alors tout à fait correspondre aux besoins.

Ce dernier est itératif, chaque itération étant composée des trois phases suivantes :

- **Analyse :** on analyse les besoins des acteurs du système, un panel représentatif d'utilisateurs des deux catégories concernées doit être constitué afin d'établir les dits besoins.

- **Conception :** un prototype doit être conçu en fonction des besoins établis à l'étape précédente. Chaque prototype servira le plus souvent de base à celui à l'étape suivante

- **Évaluation :** sur la base du prototype réalisé, une évaluation est faite. Le procédé étant itératif, cette évaluation servira de base à la modification des besoins de la première étape de l'itération suivante.

Afin de mener à bien ces trois phases, il est nécessaire d'établir des critères d'évaluation qui serviront aussi de base à l'élaboration des besoins.

- **Pour l'utilisateur application :**
  - **Facilité d'utilisation :** la vitesse d'apprentissage, l'aide extérieure nécessaire ainsi que le nombre d'erreurs commises lors d'un test sont de bons indicateurs. Il faut de plus prendre en compte les différents niveaux de compétences parmi les utilisateurs.
  - **Correction des erreurs** : la possibilité d'annuler une action est indispensable (cf. partie 3.1)
  - **Rapidité :** le temps de réponse de l'interface doit être faible. Il faut définir par des tests quel est le délai admissible avant réponse et quel est le meilleur moyen de faire patienter l'utilisateur application.
  - **Aspect visuel** : il doit être agréable et clair. (nous pourrions parler d'aspect sensoriel pour être plus général, mais nous allons nous concentrer sur les interfaces graphiques)

- **Pour le programmeur :**
  - **Expressivité :** une bonne expressivité des langages / outils utilisés est nécessaire, un code clair et précis est plus agréable. De plus il est important d'essayer de rendre la programmation via le système intuitive.
  - **Adaptativité :** une bonne capacité d'adaptation aux différents support ainsi qu'une portabilité sont nécessaires.
  - **Sûreté :** un code qui compile doit être un code qui marche.
  - **Debugging :** des outils de debugging doivent être fournis, une analyse statique de pertinence serait un plus (détecter le maximum d'absurdités comme des éléments inutiles par exemple).
  - **IDE :** une intégration aux IDE populaires est nécessaire, la création d'un IDE spécifique peut être envisagé.
  - **Efficacité :** les critères en terme de consommation des ressources de la machine doivent être étudiés.

Tout au long de ce processus de conception, des outils issus du génie logiciel vont nous aider, en voici trois particulièrement adaptés :

- **Scénarios / diagrammes de cas d'utilisation :** leur utilisation, combinée aux critères mis en place ci-dessus, nous permettra à la fois d'établir les besoins et d'effectuer des tests choisis dans le cadre de l'évaluation.

- **Diagrammes relationnels - diagrammes de classes :** même si elle ne doit pas nous aveugler, la programmation orientée objet est adaptée au sujet. La réflexion à mener quant à l'organisation du système considéré se prête remarquablement bien à l'utilisation de ces outils. De plus, ils nous permettent, à un autre niveau, de nous mettre à la place du programmeur qui devra concevoir une interface en utilisant notre système.
- **Design pattern :** l'implémentation "native" de comportements génériques, typiquement pour les interactions entre l'utilisateur application et l'interface créée, permet à la fois de faciliter l'utilisation de notre système par le programmeur mais aussi d'établir un cadre sûr pour les les dites interactions. La partie 3.1 de ce rapport est un exemple de ce principe.

1.2 Division du travail

La conception d'une interface graphique par dessus un moteur d'application peut se diviser en quatre étapes ; une bibliothèque graphique devrait fournir des outils adéquats pour chacune, et minimiser les interdépendances entre elles.

Communication entre le moteur et l'interface Une bibliothèque graphique doit fournir des outils adaptées à la « communication » entre l'interface et le moteur logique de l'application

- affichage : ensemble des données du moteur affichées (sous diverses formes) par l'interface ;
- actions : ensemble des actions de l'utilisateur modifiant l'état des données.

Cette partie du code pose les bases de ce qu'utilisera l'interface, ce à quoi elle est destinée, mais doit demeurer indépendante de son implémentation réelle, des éléments graphiques concrets utilisés pour la fabriquer, et de la plateforme à laquelle elle est destinée.

Éléments de l'interface Il s'agit ensuite de définir les éléments de l'interface graphique qui serviront de support aux données et possibilités définis précédemment ; ceux-ci devront donc interagir avec le système logique de communication, et non directement avec le moteur de l'application.

Ils sont évidemment dépendants de la plateforme à laquelle l'interface est destinée : les raccourcis clavier d'un ordinateur seront remplacés par des gestes tactiles sur une tablette, par exemple.

La bibliothèque se doit de définir un panel aussi vaste que possible d'éléments graphique à disposition du développeur.

Notons que ces éléments ne sont pas forcément tous graphiques (lecture d'un texte ou commande vocale).

Placement des éléments Les éléments doivent ensuite être assemblés pour former une interface cohérente et fonctionnelle. À cette fin, de nombreux outils doivent être présents pour le développeur : divers « layout » permettant diverses dispositions des éléments au sein d'une même fenêtre, gestion intelligente des redimensionnements, système d'onglets, éléments permettant le zoom et défilement, fenêtres pop-up, panneaux d'options, etc.

Une telle configuration s'effectue évidemment sur un ensemble d'éléments de l'interface déjà défini, dont la taille et l'aspect graphique doivent également pouvoir être éventuellement personnalisés au cas par cas.

Par ailleurs, le développeur peut laisser une partie de cette configuration de l'interface à disposition de l'utilisateur final : lui permettre de choisir les éléments présents d'une barre d'outils, ou le

placement de celle-ci sur certains bords de la fenêtre, tout en excluant certaines autres modifications, le menu restant toujours identique.

Enfin, la forme finale d'une interface doit pouvoir être aisément enregistrée, et il doit être possible pour l'utilisateur final de passer d'une configuration à une autre s'il en existe plusieurs possibles pour une application donnée.

Aspect général Enfin, la bibliothèque peut permettre au développeur de modifier l'aspect général de son interface, en personnalisant l'aspect d'un type d'éléments graphiques. Cette dernière partie est optionnel, mais si la possibilité existe, elle doit être indépendante des trois premières : modifier l'aspect général des boutons ne doit pas impacter le code de l'interface déjà écrite.

2 Analyse de l'existant

Difficultés de la programmation d'interfaces graphiques

Programmer une interface graphique entièrement à la force du poignet n'est guère agréable, pour différentes raisons :

- Apprendre à programmer une interface graphique en utilisant l'une ou l'autre API existante peut se révéler fastidieux et complexe. En particulier, nous remarquerons que GTK+ tend à être plus laid et moins intuitif que des bibliothèques reposant sur le modèle objet, comme Qt ou Swing.
- Dans tous les cas, la programmation est longue, répétitive et, à de rares exceptions (TkInter ?), extrêmement verbeuse.
- Ceci est, a priori, dû à ce que Swing, GTK+, Qt et les autres bibliothèques reposent entièrement sur un modèle de widgets : des composants imbriqués les uns dans les autres de façon quasi-infinie, qu'on doit tous définir, paramétrer et placer manuellement. On parle d'ailleurs de widget toolkit pour définir les bibliothèques permettant de construire des interfaces graphiques.

Environnements de développement

Différents environnements de développement intégré (IDE) plus ou moins sophistiqués, tels que QtCreator/QtDesigner ou Glade, peuvent faciliter la création d'interfaces graphiques, mais il est toujours nécessaire de savoir coder et de comprendre l'API utilisé avant de pouvoir vraiment en tirer parti.

MacOS propose, avec XCode et différents outils comme Cocoa, une vision intéressante -> à développer.

Portabilité

Qt est disponible en natif sur la plupart des systèmes d'exploitation existants, et notamment les systèmes embarqués, incluant Android, VxWorks ou encore BlackBerry OS. GTK+ lui n'est disponible que sur les systèmes de bureaux majeurs : Windows, MacOS, Linux/Unix.

Swing offre une définition de la portabilité différente de celle de Qt et GTK+ : en effet, la bibliothèque n'est utilisable qu'en Java, mais celui-ci étant multiplateforme, Swing devient fonctionnel

sur tout système sur lequel Java est disponible. En revanche il ne peut être utilisé qu'en java, alors que Qt et GTK+ disposent de « bindings » vers de nombreux autres langages. Nativement, Qt est une bibliothèque C++, et dispose de bindings Java, Perl, Python, PHP ou encore Ruby. GTK+ est une bibliothèque C, disposant de bindings C++, Java, Perl, Python, PHP, et Ruby mais aussi C# et Javascript ; GTK+ dispose même d'un binding Vala, un langage spécifiquement créé pour les développeurs de Gnome, basé sur C# et qui compile vers C !

Les éditeurs d'interfaces de GTK+ et Qt, respectivement Glade et QtDesigner, permettent d'enregistrer les interfaces créées sous forme de fichier XML. Le concept est intéressant puisqu'il permet de « porter » une interface dans différents langages, sans avoir à modifier le code. Cela aide donc à séparer le fond et la forme du logiciel. Il existe par ailleurs des langages de descriptions d'interface graphique basés sur XML (XAML et XUL par exemple) -> à approfondir.

Adaptation des interfaces à leur environnement

Les différents outils dédiés à la programmation d'interfaces graphiques, en particulier Swing, nous promettent souvent de s'adapter au *look and feel* de l'environnement dans lequel l'application est exécutée. Si l'idée est bonne puisqu'elle permettrait aux programmeurs d'obtenir des applications parfaitement intégrées dans les différents OS sans avoir à faire quoique ce soit de particulier, le résultat est en pratique souvent aléatoire et rarement esthétique. [Screenshots]

L'exemple particulier du HTML / CSS

Bien que n'étant pas fait pour la création d'interfaces graphiques stricto sensu, le couple HTML CSS propose une approche intéressante de la conception d'une interface web : séparer le style de la description des éléments.

Le but est ici de pouvoir afficher un même contenu sous différentes formes, par choix de l'utilisateur sur les sites qui le proposent, ou selon le support utilisé : un document HTML peut utiliser différentes feuilles de styles et définir laquelle sera utilisée selon le média sur lequel il sera affiché (grâce à l'attribut média de la balise style), et un document `css` peut lui-même choisir d'afficher un élément différemment selon le support, par exemple pour optimiser le rendu d'une page web à l'impression (*media screen* lorsque la page est affichée par un navigateur, *print* lorsqu'on veut l'imprimer).

Néanmoins on a un problème similaire à celui des interfaces graphiques qui n'ont pas le même rendu sous différents environnements : tous les navigateurs n'interprètent pas le CSS de la même manière alors qu'ils le devraient.

De plus, le concept de « boites dans des boites » est toujours présent (les balises imbriquées du HTML).

Performances

Si autrefois GTK+ était célèbre pour être plus rapide que Qt (ce qui apparemment était dû au vieux compilateur utilisé par Qt ?), les deux ont aujourd'hui des performances similaires que ce soit au niveau de la mémoire ou du CPU, ce n'est donc plus un critère pour les départager.

3 Outils à développer

3.1 Actions

Une librairie graphique se doit de fournir une représentation des *actions* que l'utilisateur peut accomplir. Fondamentalement, il s'agit d'une fonction qui a accès au moteur de l'application, contrairement à l'interface proprement dite, mais d'autres mécanismes internes s'y greffent.

Ces *actions* sont indépendantes des éléments graphiques concrets qui l'implémentent, et donc en particulier de la plateforme sur laquelle tourne l'interface utilisateur.

Toute intervention de l'utilisateur final sur le système de l'application doit passer par une *action* telle que définie par la librairie.

Deux principes nous guident :
- La façon dont l'utilisateur accomplit cette action n'a aucune importance ; l'action n'a pas besoin de savoir qu'elle a été déclenchée par un bouton, une entrée de menu, un raccourci clavier, une commande vocale, ou même comme conséquence automatique d'une autre action.
- Tous les paramètres des éléments graphiques liés à une action doivent être « transférés » à l'action si possible, tels que :
  - les raccourcis claviers, noms, descriptions, textes d'aide ou icônes associés à une action ;
  - la possibilité d'accomplir l'action (qui déterminera si le bouton ou l'entrée de menu sont actifs ou non, par exemple).

Ce mécanisme d'action fourni par la librairie graphique doit être doté d'un pattern « observer », permettant à d'autres éléments d'être notifié du déclenchement de l'action.

Les actions doivent pouvoir être aisément composées, afin de permettre au développeur de n'implémenter que les interactions minimales avec son système, tout en proposant à l'utilisateur final des fonctionnalités simples autant qu'avancés, résultant éventuellement de combinaisons complexes de ces briques de base.

Enfin, les actions effectuées doivent pouvoir être enregistrées, afin d'en conserver un historique. Idéalement, si chaque action dispose également d'une fonction « inverse » permettant d'annuler ses effets, la bibliothèque graphique peut fournir elle-même la fonctionnalité « undo / redo », aujourd'hui devenue indispensable à toute interface moderne.

3.2 Bindings

Très souvent, l'intervention de l'utilisateur modifie des valeurs internes au moteur de l'application ; parfois néanmoins, l'inverse peut être également utile : la modification d'une valeur du moteur de l'application modifie l'état d'un élément de l'interface. Il s'agit alors de modéliser efficacement la liaison d'une propriété d'un élément graphique à une valeur de la logique interne de l'application.

Voici quelques exemples qui pourraient se révéler utiles au développement d'une interface :
- progression d'une opération affichée par une barre de progression ;
- possibilité d'effectuer une action liée à un booléen, impliquant l'apparence des éléments graphiques qui lui sont liés (actifs ou non) ;
- champs d'un label liée à une chaîne de caractère, position d'un curseur liée à une valeur numérique ;

– contenu d'un panneau lié à une image dynamiquement déterminée par le moteur applicatif.

Si l'élément graphique peut être édité par l'utilisateur, la liaison doit être effective dans les deux sens : le changement du champs du label par l'utilisateur doit modifier la valeur de la chaîne de caractère en interne, tout autre changement de la valeur doit être immédiatement répercuté dans l'affichage, comme dans le cas de la barre d'url des navigateurs internet (qui est évidemment actualisé en cas de redirection, ou si l'utilisateur utilise un autre moyen pour parvenir sur un page web).

À cette fin, la librairie graphique peut proposer des représentations des types simples « pertinents » comprenant un pattern observer à destination des éléments de l'interface ; la liaison d'une propriété graphique à la valeur deviendrait alors immédiate et transparente pour le développeur.

L'inconvénient est que le moteur de l'application doit alors utiliser la librairie graphique pour implémenter de telles valeurs.

3.3 Le modèle relationnel

Les différents élément d'une interface utilisateur doivent être mis en relation les uns avec les autres afin de former un tout cohérent. Le principe le plus basique, qui est celui du modèle à widget évoqué dans la partie 2, est de ne considérer qu'une seule relation : la relation de parenté entre le contenant et le contenu.

Il serait cependant intéressant de pouvoir définir plus finement les relations entre ces différents éléments, plus précisément, l'idée serait de ne pas avoir à décrire l'emplacement d'un élément par rapport à un autre, mais plutôt les relations entre ces éléments, un peu à la manière du couple HTML / CSS.

Un premier exemple de relation autre qu'une relation de parenté est la relation « menu » qui est, à un niveau basique, présente dans Swing : pour certains éléments il est possible de définir un menu sans avoir à spécifier que ce dernier est contenu dans l'élément demandeur. Il y a une relation de « menu » entre ces deux éléments et non une relation directe de positionnement.

Ce modèle a pour but de permettre à l'interface produite de s'adapter à son environnement d'utilisation. De plus, une analyse statique de la cohérence d'une interface pourrait certainement se faire à partir d'un tel modèle (réflexion en cours).

3.4 Langages intermédiaires

Le système a vocation à être porté sur différents langages de programmation ; l'utilisation de langages intermédiaires, propres au dit système, permet entre autre une portabilité intéressante.

Deux langages intermédiaires sont introduit à cet effet :

– **Style de l'interface :** notre solution à base de modèle relationnel doit être complétée, afin d'obtenir l'adaptativité recherchée, d'une déclaration de style dont le rôle est de traduire les relations entre les composants en une présentation graphique adaptée au support. Ceci nécessite d'une part l'élaboration d'un langage intermédiaire servant à décrire cette traduction et d'autre part la réalisation d'un interpréteur du dit langage. L'utilisation d'une telle architecture permet à la fois d'avoir un comportement par défaut propre à chaque support, mais peut

aussi permettre au programmeur d'établir une charte graphique pour toutes ses applications, et ce sans avoir à s'en préoccuper à chaque nouveau projet.

– **Structure de l'interface :** une autre volonté de notre part est de permettre la sauvegarde de la structure d'une interface. Avoir une description de ladite structure non liée à langage source permet une réutilisation intéressante sur différents projets. La difficulté d'une telle représentation se situe au niveau des interactions entre le moteur de l'application et l'interface utilisateur, cependant le modèle d'actions est adapté à cette démarche (cette partie est encore floue, nous sommes encore en phase de réflexion quant à la faisabilité et la pertinence de la chose).