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.

- Deux catégories : nous utiliserons la terminologie suivante afin de distinguer les deux catégories d'utilisateurs : utilisateur application, et programmeur.
- Dans les deux cas, il faut avoir une approche orientée utilisateur, les besoins seront différents, mais l'approche à adopter la même.
- Utilisateur application : il faut prendre en compte à la fois l'aspect sensoriel (visuel si écran il y a) et l'aspect "contrôle" (gestion des périphériques d'entrée).
- Interaction utilisateur application - programme : plusieurs "niveaux" de compétences parmi les utilisateurs avec des besoins différents.

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. L'utilisateur application doit être pris en compte même s'il sera la cible du système développé par le programmeur et non directement la cible du notre.

Ses besoins doivent nous permettre de mieux déterminer ce dont ont besoin les programmeurs pour satisfaire aux exigences des utilisateurs application.

- Pour l'utilisateur application :
  - Vitesse d'apprentissage - aide nécessaire / intégrée - nombre d'erreurs commises lors d'un test.
  - Correction des erreurs.
  - Temps de réponse.
  - Efficacité - navigation rapide ?
- Pour le programmeur :
  - Programmation intuitive.
  - Bonne expressivité - ne pas avoir un code verbeux à produire (capacité d'adaptation au support etc etc ?).
  - Typage fort - sûreté.
  - Debugging aisé - analyse de pertinence (détecter le maximum d'absurdités)
  - Intégration aux IDE populaires ? Création d'un IDE ?
- **Les outils :** tout au long de ce processus de conception, des outils du génie logiciel vont nous aider.
  - 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 parties qui devraient demeurer indépendantes le plus possible.

- Communication entre le moteur et l'interface :
  - Affichage : ensemble des données du moteur affichées (sous diverse forme) par l'interface ;
  - Actions : ensemble des actions de l'utilisateur modifiant l'état du moteur. Tout élément de l'interface permettant d'agir sur le moteur devrait être lié à une "Action". Une même Action doit pouvoir être effectuée par différents éléments d'une interface, et par différentes interfaces.

  Cette section doit être indépendante de l'interface finale (éléments, aspect, etc) et de la plateforme.
- Éléments de l'interface ; dépendant de la plateforme, lié à la partie précédente.
- Personnalisation des éléments : positionnement, taille, aspect...
  Une telle configuration est évidemment dépendante de la partie précédente.
  Certains paramètres de cette configuration doivent pouvoir être modifiés par l'utilisateur final.

Ces configurations doivent pouvoir être enregistrées et s'échanger facilement.
- Aspect général.

2 Analyse de l'existant

Analyse de Swing / GTK (qui ont évidement des problèmes insurmontables) ainsi que du couple HTML CSS et le concept intéressant de la séparation du style de la déclaration des objets.

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).