Un langage objet pour la construction d'interfaces graphiques

Erick Gallesio
Université de Nice - Sophia Antipolis
I3S/CNRS - ESSI
Route des Colles - B.P. 145
06903 Sophia-Antipolis Cedex - FRANCE
Email: eg@unice.fr

Résumé: Ce papier présente un système graphique construit sur la boîte à outils graphique Tk et sur le langage Scheme. Tk est une *toolkit* graphique portable et très largement diffusée construite au dessus du langage Tcl. Tcl, quant à lui, est un langage de scripts, qui n'est pas adapté au développement de logiciels de taille conséquente. Afin d'améliorer le niveau de programmation de la toolkit Tk, nous avons défini le langage STklos, basé sur Scheme et offrant un système objet basé sur CLOS. Ce langage a permis de redéfinir proprement, sous la forme d'une hiérarchie de classes, tous les objets graphiques de Tk.

1 Motivations

Tk [11] est une boîte à outils graphique très largement diffusée et utilisée. Elle offre un ensemble de widgets assez complet: boutons, ascenseurs, menus, éditeurs de textes, ...Ces widgets permettent de construire facilement et rapidement des interfaces graphiques soignées sans avoir à connaître les arcanes de la programmation X11 (ou MS-Windows ou MacIntosh, puisque Tk a été aussi porté sur ces deux systèmes). Cette toolkit est construite au dessus d'un langage interprété appelé Tcl (Tool Command Language) [9].

Tcl est un petit langage de scripts ayant une syntaxe proche des shells que l'on peut trouver sous Unix. Ce langage a été développé dans le souci de pouvoir facilement embarquer son interprète dans une application existante. Par conséquent, son auteur a particulièrement veillé à ne pas le surcharger inutilement. Ceci explique, par exemple, que Tcl ne propose pas d'autres moyens pour structurer les données que les chaînes caractères et les tables associatives. En fait, Tcl a été conçu principalement pour l'écriture de petits scripts permettant d'assembler les gros composants d'une application, qui sont eux généralement écrits en C ou en C++. Toutefois, l'expérience montre que les gens rechignent à utiliser Tcl de cette façon, et qu'ils préfèrent utiliser un seul langage pour leur application, plutôt que d'en faire collaborer plusieurs.

Tk peut être vu comme une application qui embarque le langage Tcl [10]. La nature interpretée de Tcl offre à Tk une interface simple et attrayante pour développer des applications graphiques. Cette apparente facilité avec laquelle on peut écrire de petits scripts n'est malheureusement pas transposable pour de grosses applications, du fait de l'absence de moyens de structuration des données en Tcl. Notons qu'à ce jour, on dénombre plusieurs applications Tcl dépassant le millions de lignes, ce qui nous paraît particulièrement déraisonnable.

Afin d'améliorer l'utilisation de la toolkit Tk pour des applications plus importantes, nous avons décidé de remplacer le Tcl par un *vrai* langage de programmation. Le langage choisi pour cette substitution est le langage Scheme [1], car c'est un langage

- **de haut niveau** qui offre de nombreux types de données tels que tableaux, listes, chaînes de caractères, nombres, ...
- **petit** et donc facilement embarquable dans une application (comme Tcl)

Journées du GDR Programmation.
20, 21 et 22 novembre 1996.
Orléans.

- **efficace** qui permet par conséquent d'envisager l'écriture d'applications complètes sans avoir à recourir à la programmation en C ou en C++.

- **propre**, avec en particulier la notion de fermeture.

- facilement **extensible**.

STk [3] est donc un interprète Scheme, étendu pour accéder à la boîte à outils Tk. Les bases théoriques solides de Scheme permettent de penser que ce langage est tout-à-fait souhaitable pour l'écriture de programmes de taille raisonnable. Toutefois, la puissance d'expression de Scheme ne nous semble pas suffisante pour le développement de gros programmes. En particulier, l'absence de mécanisme objet nous semble être un frein important au développement de tels programmes. STklos, la couche objet de STk, a été définie pour pallier ce problème. Cette extension objets offre des mécanismes tels que méta-classes, héritage multiple ou fonctions génériques (proches de celle de CLOS [12] or Dylan [2]).

STklos a permis de redéfinir les widgets de la toolkit Tk comme des objets, et de les intégrer dans une hiérarchie de classe cohérente. L'utilisation de ces classes simplifie grandement la programmation de la bibliothèque Tk en fournissant un accès homogène aux options des widgets Tk, et en cachant les détails de bas niveau de cette toolkit.

La section suivante présente le langage STklos. La section 3 montre comment les widgets Tk peuvent être intégrés harmonieusement dans des objets STklos. Enfin, la section 4 donnera un bref aperçu de l'état actuel du projet STklos.

2 Présentation de STklos

Avec STk, la programmation de la toolkit Tk peut se faire à deux niveaux. Le premier niveau correspond à l'utilisation de primitives spéciales qui étendent l'interprète Scheme standard. Le second niveau, quant à lui, consiste à utiliser les widgets Tk comme des objets STklos. Bien-sûr, les deux niveaux peuvent être utilisés indifféremment dans un programme si nécessaire. Toutefois, en général, on préferera écrire ses programmes avec la couche objet, et ne recourir à la couche de base que lorsque c'est strictement nécessaire.

2.1 STk: la couche de base

Au démarrage d'une session de l'interprète STk, l'utilisateur a d'emblée l'accès à la couche de base qui permet d'interagir avec la toolkit Tk. L'utilisation des widgets de la toolkit Tk est alors assez similaire à celle que l'on en fait en Tcl (à quelques règles de réécriture près).

La création de nouveaux objets graphiques (boutons, menus, éditeurs ...) se fait à l'aide de primitives spéciales STk. Par exemple, la création d'un nouveau bouton peut se faire de la façon suivante:

```
(button '.b)
```

Pour Tk, le nom d'une widget est assez similaire à un nom de fichier absolu Unix; il reflète la position de cet objet dans la hiérarchie graphique des widgets. Ainsi, dans l'exemple précédent, le nom de notre nouveau bouton est "`.b`". Ce nom indique que "`b`" est un descendant de "`.`", la fenêtre mère. Ici, le nom de la widget est *quoté* pour prendre en compte les règles d'évaluation de Scheme.

L'appel d'une primitive de création de widget, telle que `button`, construit un nouvel objet Scheme de type *Tk-command.* Cet objet est automatiquement associé au symbole utilisé pour la création de la widget (`.b` dans le cas présent). Pour l'interprète STk, une *Tk-command* est un type de fonction spéciale, qui est utilisée pour paramétrer la widget qui lui est associée. Par exemple, l'expression

```
(.b 'configure :text "Hello, world" :border 3)
```

permet de mettre à jour les options `text` et `border` du bouton `.b`. Bien-sûr, comme en Tcl/Tk, ces paramètres auraient pu être passés au moment de la création de la widget, et l'on aurait pu écrire:

```
(button '.b :text "Hello, world" :border 3)
```

Tk offre un mécanisme très général pour associer un script à un événement externe (*e.g.* un appui sur une touche ou une action de la souris). Ce script, que l'on appelle généralement un *handler*, est automatiquement déclenché dès que l'événement attendu se produit. En Tcl, le script exécuté est en fait stocké dans une chaîne de caractères qui est évaluée au niveau global, alors qu'en STK, ce script est constitué d'une fermeture Scheme. L'expression suivante montre comment ajouter un *handler* prenant en compte l'appui du bouton droit de la souris sur l'objet graphique `.b`:

```
(bind .b "<ButtonPress-3>"
      (let ((compteur 0))
        (lambda ()
          (set! compteur (+ compteur 1))
          (format #t "Appui du bouton: ~A~%" compteur))))
```

Cet exemple simple illustre assez bien le fait que les *handlers* que l'on peut écrire en STK sont beaucoup plus *propres* que ceux de Tcl. En effet, en STK, on profite des règles de portée propres à Scheme qui permettent à un handler d'avoir ses propres variables globales (comme `compteur` ici), alors qu'en Tcl un handler est une simple chaîne de caractères sans notion d'environnement.

Même si les fermetures Scheme constituent un outil puissant pour l'expression de handlers associés à des événements externes,, programmer une interface graphique en utilisant seulement les constructions de la couche de base devient vite rédhibitoire. En fait, la couche de base de STK doit être considérée comme une sorte de langage d'assemblage pour la programmation d'interfaces graphiques. Nous verrons en particulier dans la section 3 comment cette couche peut être utilisée pour *réifier* les widgets Tk dans des classes STKLOS.

2.2 STK: la couche objet

STKLOS, l'extension objet de STK, est proche de CLOS [12]; elle est brièvement introduite dans cette section. Notons que nous ne considérons ici que l'aspect *langage* STKLOS, et que nous mettons de côté, pour l'instant, tout ce qui est relatif à la programmation de la toolkit Tk.

En STKLOS, la définition d'une nouvelle classe se fait avec la macro `define-class`. Par exemple,

```
(define-class Point ()
   ((x :init-keyword :x :accessor x-of)
    (y :init-keyword :y :accessor y-of)))
```

permet de définir les caractéristiques d'un point (deux champs sont déclarés ici: `x` et `y`). La création d'une instance de la classe `Point` se fait avec le constructeur `make`:

```
(define p (make Point :x 10 :y 20))
```

L'évaluation de la forme précédente définit un nouveau point et initialise ses slots `x` et `y` avec les valeurs 10 et 20. Le contenu d'un slot peut toujours être accédé en lecture et en écriture par les deux primitives `slot-ref` et `slot-set!`. Ces primitives permettent l'accès à un champ au plus bas niveau et l'on préfère en général utiliser des fonctions d'accès (*accessor*) puisqu'elles conduisent en général à un code plus lisible. Par exemple, la lecture du champ `y` du point `p` défini précedemment peut se faire avec

```
(y-of p)                 ; ou (slot-ref p 'y)
```

puisque la fonction d'accès `y-of` a été définie pour le champ `y`. La mise à jour de ce champ peut se faire avec la forme syntaxique étendue `set!` de STKLOS:

```
(set! (y-of p) 1)        ; ou (slot-set! p 'y 1)
```

Définissons maintenant la classe `Rectangle` qui hérite de la classe `Point`:

```
(define-class Rectangle (Point)
  ((width  :init-keyword :width  :accessor width-of)
   (height :init-keyword :height :accessor height-of)))
```

Les instances de cette classe comportent quatre champs: **x**, **y**, **width** et **height**. Les méthodes[1] définies pour les instances de la classe **Point** peuvent aussi être utilisées pour les instances de la classe **Rectangle**. Par exemple, le champ **x** d'un **Rectangle** peut être accédé avec la fonction d'accés **x-of** définie plus haut.

La définition précédente permet de représenter un rectangle avec un point de référence, une hauteur et une largeur. Cette représentation est suffisante la plupart du temps, mais il n'est pas rare d'avoir besoin d'une representation utilisant les coordonnées de deux points opposés du rectangle. Les *champs virtuels* de STk permettent de gérer facilement ce type de situation. Un champ virtuel est défini comme un champ normal, si ce n'est son allocation qui est qualifiée avec le mot-clé **:vitual**. Un tel champ n'occupe pas de place en mémoire et sa lecture (resp. sa mise à jour) provoque l'exécution d'une fonction de lecture (resp. écriture). Les fonctions d'accès au champ virtuel doivent être fournies par l'utilisateur au moment de la déclaration du champ grâce aux options **slot-ref** et **slot-set!**. L'exemple ci-dessous est une autre définition de la classe **Rectangle** utilisant des champs virtuels.

```
(define-class Rectangle (Point)
  ((width  :init-keyword :width  :accessor width-of)
   (height :init-keyword :height :accessor height-of)

   (x2     :init-keyword :x2     :accessor x2-of
           :allocation   :virtual
           :slot-ref  (lambda (obj) (+ (x-of obj) (width-of obj)))
           :slot-set! (lambda (obj val)
                        (set! (width-of obj) (- val (x-of obj)))))

   (y2     :init-keyword :y2 :accessor y2-of
           :allocation   :virtual
           :slot-ref  (lambda (obj) (+ (y-of obj) (height-of obj)))
           :slot-set! (lambda (obj val)
                        (set! (height-of obj) (- val (y-of obj)))))))
```

Ici, **x2** et **y2** sont déclarés comme des slots virtuels. Les fonctions de lecture et d'écriture sont ici des fonctions anonymes qui lisent ou affectent les valeurs du champ correspondant en fonction des autres champs de l'objet. Notons que ces fonctions d'accès peuvent être amenées à changer la valeur de champs standard pour assurer la cohérence interne de l'objet.

Comme les champs virtuels ne consomment pas de mémoire, ils pourraient être facilement simulés avec des méthodes en STklos. Toutefois, la déclaration d'un champ virtuel permet aux fonctions d'introspection de *voir* ce champ comme s'il était un champ standard. A l'inverse, l'utilisation d'un couple de méthodes pour simuler un champ virtuel permet de cacher celui-ci à ces fonctions.

3 Intégration des objets graphiques Tk en STklos

3.1 La hiérarchie de classes

Cette section montre comment les widgets de Tk ont pu être *integrées* dans des classes STklos. Chaque objet graphique défini dans la toolkit Tk est representé par une classe STklos. Ces classes sont organisées suivant une hiérarchie qui est brièvement décrite ici. Tout d'abord, toutes les classes se partagent un ancêtre commun: la classe **<Tk-object>**[2]. Cette classe définit en particulier deux champs qui sont nécessaires pour la communication avec la toolkit Tk[3].

- Le champ **parent** est une référence sur l'objet qui contient (graphiquement) la widget courante.

[1] En STklos[5] l'exécution d'une méthode repose sur un sous-ensemble du mécanisme d'appel de *fonctions génériques* de CLOS (seules les méthodes primaires sont supportées)

[2] **<Tk-object>** est une classe virtuelle qui ne doit pas être instanciée (les classes dont le nom commence avec le préfixe "**Tk-**" sont toutes des classes virtuelles. Elle correspondent aux classes *spécifiques de l'implémentation* de [7]).

[3] L'implémentation actuelle est en réalité un petit peu différente mais, pour ne pas compliquer, nous nous contenterons ici de cette hiérarchie simplifiée.

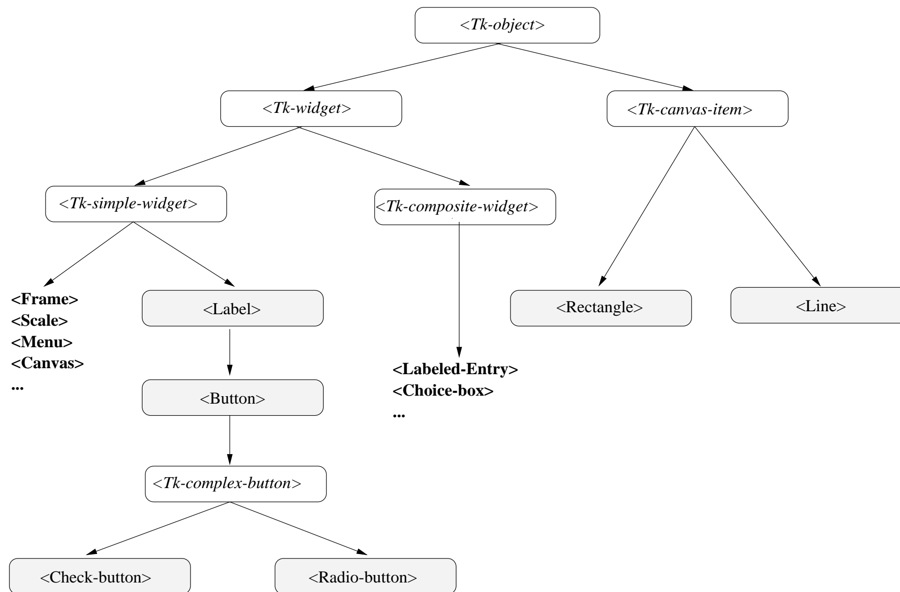

Figure 1: *Une vue partielle de la hiérarchie* STKLOS

- Le champ `Id` contient une référence sur l'objet de la couche de base (une *Tk-command*) qui implémente la widget STKLOS. C'est en fait ce champ qui établit la correspondance entre la couche de base et la couche objet. De plus, celui-ci garantit que l'objet de base ne sera pas libéré par le GC.

Le niveau suivant de la hiérarchie de classes se subdivise en deux branches: les classes `<Tk-widget>` et `<Tk-canvas-item>`. Les instances de la première classe correspondent aux objets graphiques classiques de réalisation d'interfaces (boutons, menus, glissières, ...) alors que les instances de `<Tk-canvas-item>` correspondent aux objets contenus dans un canvas[4] (lignes, rectangles, ellipses, ...). Une vue partielle de la hiérarchie de classes est présentée à la figure 1. Quelques points importants doivent être notés sur cette figure:

- En Tk, les objets d'interface (*e.g.* boutons) sont des objets de première classe, alors que les composants de canvas (*e.g.* rectangles) ne peuvent être lus ou écrits qu'au travers de commandes spécifiques au canvas les contenant. Par conséquent, l'accès aux options de widgets ou de composants de canvas se font de façon différente. En effet, l'accès à un composant de canvas nécessite l'utilisation de deux références: une au canvas, et une autre (un entier) permettant de repérer de façon unique le composant. Afin de pouvoir manipuler les composants de canvas comme des objets de première classe, `<Tk-canvas-item>` définit un slot supplémentaire contenant le numéro d'identification du composant dans son canvas.

- La hiérarchie de classes présentée ici permet d'avoir une meilleure appréhension de la toolkit Tk, même si la notion d'héritage est complètement absente de Tk. D'après la figure 1, un *bouton* (classe `<Button>`) peut être vu comme une *étiquette* (classe `<Label>`) *réactive*. Par conséquent, les champs ou méthodes chargés de la mise en forme (fonte, couleur, ...) d'une étiquette ou d'un bouton peuvent être partagés dans la classe `<Label>`. Ainsi, seuls les champs ou les méthodes spécifiques à un texte réactif seront définis dans la classe `<Button>` (typiquement le champ contenant la fermeture à appliquer lorsque l'on cliquera sur le bouton).

- Les widgets simples et composites partagent un ancêtre commun: `<Tk-widget>`. Par conséquent, les widgets composites, entièrement écrites en Scheme, sont traitées de la même façon que les widgets prédéfinies de la toolkit, écrites, quant à elles, généralement en C. Pour plus d'information sur les widgets composites, voir [4].

[4] La widget *canvas* livrée avec Tk permet l'édition struturée 2D.

3.2 Accès aux options des objets graphiques Tk

Chaque widget Tk connait un certain nombre d'options qui permettent de modifier son comportement ou sa mise en forme. Les options sont spécifiées en Tk soit au moment de la création, soit, après coup, à l'aide de l'opération **configure**, disponible sur toutes les widgets Tk. En STKLOS, une option de la widget Tk est considérée comme un champ de l'objet représentant cette widget. Ainsi, la lecture de l'état d'une option, ou sa reconfiguration, correspondent respectivement en STKLOS à la lecture ou à l'écriture d'un champ d'objet. L'exemple suivant montre une définition possible en STKLOS de la classe **<Button>**:

```
(define-class <Button> (<Label>)
   ((command :accessor command :init-keyword :command
             :allocation :tk-virtual))
   :metaclass <Tk-metaclass>)
```

Cette nouvelle classe hérite de la classe **<Label>** et contient un champ propre appelé **command**. L'allocation de ce slot est qualifiée avec le mot-clé **:tk-virtual**. Les champs *Tk-virtual* sont traités spécialement: ceux-ci peuvent être utilisés de la même manière que les champs d'instance, mais ne sont pas alloués dans le monde Scheme (*i.e.* leur valeur est sauvegardée dans les structures internes de la toolkit Tk, plutôt que dans un objet Scheme). La lecture ou l'écriture de tels champs passe par l'utilisation de la fonction **configure** de Tk vue en 2.1. Un champ *Tk-virtual* est en fait un cas particulier de champ virtuel géré par la méta-classe **<Tk-metaclass>**. La définition d'une classe utilisant cette méta-classe permet de capturer les accès au widgets Tk au plus bas niveau. Par conséquent, la valeur d'un champ virtuel d'une widget contient toujours la valeur actuelle de l'option Tk correspondante, ce qui évite tout problème de synchronisation entre la couche base et la couche objet. La spécification de la méta-classe de la classe **<Button>** est précisée ici avec l'option **:metaclass**[5]. Il est important de noter ici que la construction des fonctions d'accès est faite au moment de la création de la classe, et que, par conséquent, l'accès à un champ d'objet Tk n'est pas plus coûteux que l'accès à un champ virtuel. Par conséquent, la modification d'une widget est **aussi efficace** que l'on utilise la couche STKLOS ou le mécanisme de configuration standard de Tk.

La définition précédente de la classe **<Button>** n'est pas suffisante pour une intégration complète de la widget bouton de Tk dans une classe STKLOS. En fait, le MOP de STKLOS nécessite qu'une méthode de la fonction générique **Tk-constructor** soit définie pour chaque widget Tk. Cette méthode est appelée au moment de la création d'une nouvelle instance de la classe **<Tk-widget>**. Cette méthode doit rendre le nom de la commande Tk (une *Tk-command*) devant être appelée pour créer effectivement la widget. Pour la classe **<Button>**, la méthode suivante doit être définie:

```
(define-method Tk-constructor ((b <Button>))
   button)
```

Ces définitions de classe et de méthode suffisent pour définir une nouvelle widget STKLOS. Ce point est particulièrement important puisque, comme on peut le voir, l'intégration d'une nouvelle widget se fait à un coût quasi-nul. Ceci permet d'adapter facilement la distribution de STKLOS aux nouvelles versions de Tk (qui sont assez fréquentes).

La définition suivante montre comment utiliser notre nouvelle classe **<Button>**:

```
(define b (make <Button> :font "fixed"
                         :command (lambda () (display "Hello\n"))))
```

Cette expression affecte à la variable **b** une nouvelle instance de la classe **<Button>**. Changer la police de caractère associée à cet objet peut se faire en utilisant simplement la primitive **slot-set!** ou la forme genéralisée **set!** que l'on a vues en 2.2.

3.3 Comparaison de STKLOS et de Tk standard

Certains des avantages de STKLOS sur Tk standard ont déjà été cités précedemment; nous poursuivons ici cette discussion.

[5] La citation de la méta-classe est en fait inutile ici,et aurait pu être omise, puisque la classe **<Label>**, ou plus probablement un de ses ancêtres, l'a déjà spécifiée. Dans ce cas, le système choisira automatiquement la méta-classe la plus spécifique.

3.3.1 Masquage de détails de bas niveau

Lorsque l'on utilise les widgets Tk au travers STkLOS, le principal avantage que l'on en tire est très certainement que la plupart des *bizarreries* de Tk peuvent être cachées, ce qui améliore considérablement le niveau avec lequel on programme une interface graphique. En particulier, l'utilisation de STkLOS permet d'oublier les conventions de nommage de Tk. Ces conventions, nous l'avons vu, imposent que le nom d'une widget reflète sa position dans la hiérarchie de widget. De plus, Tk n'offre aucun moyen pour définir des noms de façon relative. Ces contraintes rendent difficile la construction d'interfaces graphiques, mais surtout la définition de composants d'interface standard réutilisables. De plus, cette convention impose des changements lourds dans le code lorsque l'on décide de modifier la hiérarchie des widgets (ce qui est assez fréquent lorsque l'on conçoit une interface graphique complexe, puisque on est souvent amener à un développement par essai/erreur).

En STkLOS, les conventions de nommage de Tk sont complètement cachées et la seule chose que le programmeur doit connaître sur un objet (que ce soit une widget, un composant de canvas, ou un attribut de texte) est la widget qui le contient (cet objet est appelé son *parent*). L'exemple suivant montre comment créer une petite hiérarchie d'objets:

```
(define f  (make <Frame>))
(define b1 (make <Button> :text "B1" :parent f))
(define b2 (make <Button> :text "B2" :parent f))
```

Les boutons `b1` et `b2` créés ici spécifient que leur parent est la *boîte* `f`. Comme cette boîte ne spécifie pas de parent particulier, celle-ci est considérée comme étant un descendant direct de la fenêtre principale de l'application. Notons que seule la définition de `f` devrait être changée si l'on décidait que cette boîte ne devait plus être un descendant direct de la fenêtre principale. Une modification de la hiérarchie de cette widget est automatiquement propagée à toutes ses fenêtres filles. STkLOS étend aussi la notion de parent pour prendre en compte les composants de canvas (rectangles, lignes, bitmaps, ...): un composant est considéré comme un descendant du canvas auquel il appartient. Cette vision des composants de canvas permet de les manipuler comme des objets de première classe. Par exemple,

```
(define c (make <Canvas>))
(define r (make <Rectangle> :parent c :coords '(0 0 50 50)))
```

permet de définir un rectangle `r` dans le canvas `c`. Comme nous l'avons dit précedemment, l'accès à ce rectangle nécessite l'utilisation de deux références en Tk standard: le canvas `c` et son numéro d'identification dans ce canvas. En STkLOS, ces deux informations sont dans l'objet `r`. Par exemple, après l'exécution de l'expression suivante

```
(bind r "<Enter>" (lambda (x y)
                    (format #t "La souris entre en  ~A ~A~%" x y)))
```

un message est affiché à chaque fois que la souris entre dans le rectangle `r`. Il est important de noter ici que nous utiliserions **exactement** la même expression pour associer un tel comportement à une simple widget (comme un bouton ou un menu par exemple), alors qu'il faut deux formes syntaxiques différentes en Tcl/Tk, puisque dans ce cas les procédures d'accès à un composant de canvas ou à une widget sont différentes.

3.3.2 Accès uniforme à la toolkit Tk

L'utilisation de fonctions génériques améliore aussi sensiblement le niveau de programmation de la toolkit Tk, puisqu'elle permet un accès uniforme aux commandes de Tk. Supposons que l'on veuille accéder à la valeur courante d'une widget glissière (scale widget) ou à un éditeur de texte (entry widget) avec la fonction générique `value`. Ceci peut être exprimé très facilement avec la définition de méthode suivante:

```
(define-method value ((obj <Tk-simple-widget>))
   ((Id obj) 'get))
```

Dans ce cas, une méthode suffit pour implémenter la fonction de lecture de la valeur puisque l'accès en lecture se fait avec la même sous-option pour les widgets `scale` et `entry` en Tk. La définition de la méthode d'écriture est un petit peu plus complexe car les façons de changer la valeur d'une `entry` ou d'une `scale` sont différentes en Tk.

```
(define-method (setter value) ((obj <Scale>) v)
  ((Id obj) 'set v))

(define-method (setter value) ((obj <Entry>) v)
  ((Id obj) 'delete 0 'end)
  ((Id obj) 'insert 0 v))
```

L'utilisation de la même fonction générique (avec deux méthodes différentes) permet de cacher ces détails de bas niveau au programmeur, et nous donne la possibilité de lui présenter une interface d'accès cohérente à la toolkit. Dans l'appel suivant

```
(set! (value x) 100)
```

le système choisit la méthode à appliquer en fonction de la classe de `x`. Bien-sûr, une erreur[6] sera déclenchée si `x` n'est pas un objet de type `<Scale>` ou `<Entry>`. Notons que cette notion de valeur peut aussi être facilement implémentée avec un slot virtuel (voir 2.2) même si `:value` n'existe pas comme une option de Tk en tant que telle. Cette approche, qui est celle qui a été choisie dans la distribution actuelle, permet aux fonctions d'introspection de voir la valeur d'une widget comme un champ standard. En particulier, la distribution de STkLOS offre un petit constructeur d'interfaces qui utilise pleinement les possibilités d'introspection offertes par le langage pour construire graphiquement des interfaces. La définition de `value` comme un slot virtuel pour la plupart des widgets Tk permet d'ajuster cette valeur au même titre que leur couleur de fond ou leur police de caractères, qui sont elles des options standard. La définition d'un constructeur d'interface utilisant simplement les constructions de bases de Tcl/Tk aurait été beaucoup plus difficile.

4 Etat actuel

Implémentation Dans la section précédente nous avons vu, au travers différents exemples, les apports d'un langage objet pour l'utilisation de la toolkit Tk. La simplicité de ces exemples pourrait faire croire que la définition d'un système objet *ad-hoc* pour les widgets Tk pourrait suffire pour avoir une vision objet de la toolkit. Cette approche a déjà été utilisée plusieurs fois pour Tcl, et il existe à ce jour plusieurs extensions objets (toutes incompatibles) pour le langage Tcl. Cette approche ne nous semble pas souhaitable. Nous pensons au contraire qu'il faut tout d'abord définir un système objet cohérent et facilement extensible, et que si cette tâche est atteinte, l'intégration des widgets Tk doit se faire facilement. En fait, les programmeurs d'interfaces graphiques ont souvent besoin d'utiliser de l'introspection sur les objets qu'ils manipulent. De même, ils ont aussi besoin de définir des objets composites ainsi que des fonctions d'accès aux champs de ces widget composites. Ces raisons nous ont amené à construire l'extension objet de STk sur un protocole méta-objet, pour la souplesse que ce type d'implémentation procure. Une version simplifiée du protocole méta-objet de STkLOS est présenté dans [6]; la définition de widgets composites en STkLOS, quant à elle, est décrite dans [4].

Performances Pour comparer les performances de STk et de Tcl, il faut tout d'abord comparer les performances de la couche de base de STk avec Tcl/Tk.

Tcl est par nature un langage interprété, et de nombreux aspects de ce langages le rendent difficile à compiler[7]. De plus, le fait que Tcl soit un langage de chaînes de caractères implique que les valeurs manipulées par un programme Tcl doivent toujours être converties en chaînes de caractère, ce qui est très inefficace.

L'implémentation de STk est elle aussi basée sur un interprète. Touefois, il est facile de construire un interprète efficace pour le langage Scheme. Le passage de paramètres aux widgets Tk est une opération coûteuse puisqu'elle nécessite des conversions en chaînes de caractères pour *converser* avec Tk. Ce surcoût, heureusement, est généralement largement compensé par la très grande lenteur de l'interprète Tcl. En moyenne, une interface écrite en STk est donc plus rapide que le programme équivalent écrit en Tcl.

Considèrons maintenant les performances de STkLOS par rapport a STk. En STkLOS, comme en CLOS, le mécanisme de fonction générique est l'apect le plus coûteux. Pour résoudre ce problème, l'appel de fonctions

[6]plus exactement, le système appelle la fonction générique `no-applicable-method` qui, par défaut, déclenche un erreur, comme en CLOS. L'utilisateur peut spécialiser cette fonction générique pour certaines classes si nécéssaire.

[7]plusieurs compilateurs ont déjà été annoncé pour Tcl mais, à notre connaissance, il n'existe à ce jour rien d'exploitable

génériques est entièrement traité en C (alors que le reste de STKLOS est implémenté en Scheme). Cela permet de minimiser le rapport entre les temps d'appel d'une fonction et d'une fonction générique. Toutefois la technique d'implémentation actuelle des fonctions génériques est encore assez directe, et l'utilisation de techniques de mémorisation comme celles présentés en[8] devrait permettre d'améliorer ce rapport (qui est acuellement de 3 environ). Ceci explique qu'une interface écrite en STKLOS est généralement plus lente que l'interface équivalente écrite en STK (et peut même, dans certains cas, être plus lente que la même interface écrite en Tcl/Tk).

5 Conclusion

Nous avons présenté dans ce papier le langage objet STKLOS et comment celui-ci peut être adapté pour une utilisation simplifiée de la toolkit graphique Tk. L'intérêt de ce système est qu'il permet une *réification* propre des widgets graphiques d'une toolkit non objet, et par là donc d'améliorer le style de programmation que l'on peut avoir avec cette boîte à outils.

STKLOS fournit aussi un *vrai* langage de programmation pour la toolkit Tk. De plus,le protocole méta-objet de STK fournit un moyen agréable d'accéder aux différentes options des widgets Tk. Il permet de simplifier le développement d'interfaces graphiques avec Tk, et par là même d'étendre l'intérêt de cette toolkit.

Distribution

STKLOS est distribué avec STK. Il tourne sur un grand nombre de machines Unix ainsi que sur Windows (95 et NT). La dernière version de ce *package* est disponible à l'URL suivante: `http://kaolin.unice.fr/`

References

[1] William Clinger and Jonathan Rees (editors). *Revised*^{4} Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), 1991.

[2] Apple Computer. *Dylan: an Object Oriented Dynamic Language.* April 1992.

[3] Erick Gallesio. Embedding a scheme interpreter in the Tk toolkit. In Lawrence A. Rowe, editor, *First Tcl/Tk Workshop, Berkeley*, pages 103–109, June 1993.

[4] Erick Gallesio. STKLOS: A scheme object oriented system dealing with the tk toolkit. In ICS, editor, *Xhibition 94, San Jose, CA*, pages 63–71, June 1994.

[5] Erick Gallesio. STK reference manual. Technical Report RT 95-31a, I3S CNRS / Université de Nice - Sophia Antipolis, juillet 1995.

[6] Erick Gallesio. Designing a Meta Object Protocol to wrap a Standard Graphical Toolkit. In *ISOTAS'96*, pages 137–156. LNCS 1049, Springer-Verlag, march 1996.

[7] Gregor Kickzales and John Lamping. Issues in the design and specification of class libraries. In *Proceedings of OOPSLA*, 1992.

[8] Gregor J. Kiczales and Luis H. Rodriguez Jr. *Object-Oriented Programming: The CLOS Perspective*, chapter Efficient Method Dispatch in PCL, pages 335–348. The MIT Press, Cambridge, MA, 1993.

[9] John K. Ousterhout. Tcl: an embeddable command language. In *USENIX Winter Conference*, pages 183–192, January 1990.

[10] John K. Ousterhout. An X11 toolkit based on the Tcl Language. In *USENIX Winter Conference*, pages 105–115, January 1991.

[11] John K. Ousterhout. *Tcl and the Tk toolkit.* Addison-Wesley, 1994.

[12] Guy L. Steele Jr. *Common Lisp: the Language, 2nd Edition.* Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1990.