Spécification de développement, contraintes liées aux projets du groupe le_terrier

André Connes, andre.connes@toulouse.iufm.fr

Historique;

- v0.1 6 mai 2002
- Révision v0.3 14 juillet 2002
- v8.11 22 novembre 2008

Ce document essaye de préciser les contraintes que nous souhaitons respecter lors de l'écriture (et/ou modification) de fichiers Tcl/Tk afin d'assurer une meilleure maintenance et lisibilité du code produit. Dans une seconde partie, nous précisons les contraintes liées au courriel échangé sur la liste béta relative au groupe “le_terrier”. Enfin la dernière partie décrit le “à faire” tel qu'évoqué lors des RMLL (Rencontres Mondiales du Logiciel Libre) 2002 à Bordeaux.

1. Introduction

Il ne s'agit pas de contraintes dans le but d'ennuyer le développeur bénévole.

Il s'agit de respecter quelques règles simples qui, dans le passé, ont permis une relecture aisée du code produit.

Car une application naît, vit par la volonté de différents développeurs.

Mais elle meurt si elle est illisible.

Enfin, la lisibilité est source d'apprentissage.

Développer un programme c'est écrire du code avec l'intention de communiquer avec des personnes plutôt qu'avec des machines !

2. Documentation du code à l'aide de commentaires

Les commentaires donnent des indications au lecteur mais sont ignorées du calculateur (sauf si ce sont des directives pour l'interpréteur).

Bien documenter le code par des commentaires est difficile mais c'est une nécessité.

Commentons sans excès, mais commentons, les parties difficiles à comprendre.

Evitons les commentaires

- faux

```
  set i [expr $i - 1] ;# passer à l'élément suivant
```

- sans intérêt

```
  set i [expr $i + 1] ;# ajouter 1 à la variable i
```

Préférons donc

```
  set i [expr $i + 1] ;# passer à l'individu suivant
```

encore mieux

```
  incr $i ;# passer à l'individu suivant
```

Documentons les procédures en précisant :

- L'action et les effets réalisés par la procédure, en particulier le return
- Les variables en entrée
- Les variables globales utilisées

Exemple

```
# PutManChapter --
#
#      Put a single manual chapter, with title, properly tagged, into
#      both the side panel (titles only) and the main panel (titles + text).
#
# Arguments:
#      tagno     Tag label, only used internally to connect panels "side"
#                and "main".
#      text      The name of the chapter = link title appearing in "side".
#      chapter   The plain text of the chapter (or part of it).
#
proc PutManChapter {tagno text chapter} {
global frq colours fonts
    # creates a button in the "side" panel, carrying a keyword
    # which is searched for (command!) in the "main" panel if pressed.
```

3. Identifieur

Tout programme contient **deux sortes de symboles** : les premiers appartiennent au **langage**, les seconds sont les identifieurs.

Les premiers peuvent être un caractère spécial ou une paire de caractères comme

```
  + - ! == != > < { } [ ] # ;# etc.
```

ou un mot réservé comme

```
  if else elsif while proc etc.
```

Les identifieurs peuvent être des identifieurs standard comme

```
  puts gets set pack frame etc.
```

ou des identifieurs définis par l'utilisateur.

Le choix des identifieurs définis par l'utilisateur est fondamental : un bon choix rend le programme plus facile à lire, à comprendre, à modifier, à corriger. Un identifieur long n'est pas nécessairement le meilleur. Si un identifieur n'est utilisé que peu de fois, dans une partie réduite du programme, une lettre peut être un bon identifieur, mais une lettre ne sera pas un bon choix pour un identifieur utlisé fréquemment dans différentes parties du programme.

```
  noPage numPage
```

sont des identifieurs plus compréhensibles que NP ou XXX pour désigner un numéro de page.

```
  pi
```

est sûrement mieux que A pour désigner le nombre 3,141592…

4. Littéral et constante

60, 3.141592 et “fin” sont des littéraux. Nous pouvons nommer les littéraux. **Un littéral nommé est une constante**.

Après les instructions :

```
set maxLignes 60
set pi 3.141592
set consigneFin fin
```

nous pouvons écrire :

```
for { i = 0; $i <= $MaxLignes; incr $i } {
  faire un traitement
  sur la ligne numéro i
}
set circonference [expr 2 * $pi * $r]
while { $consigne != $consigneFin } {
  quelque
  chose
  à faire
}
```

L'utilisation des constantes rend le programme plus facile à lire, à comprendre, à modifier, à corriger.

5. Variable

Les indications concernant les constantes s'appliquent aux variables d'autant plus que TclTk ne fait pas de différence entre les deux ! Il n'existe pas de constante en TclTk mais l'utilisateur doit faire comme si afin d'améliorer le code écrit.

Evitons

```
   a, i, n, a20, n7, toto, zorro, etc.
```

Peu de temps après l'écriture d'un code utilisant de telles variables, nous ne saurons plus à quoi elles peuvent bien servir.

6. Indentation du code

La règle générale est la suivante :

- Toute indentation supplémentaire est obtenue en ajoutant 2 espaces de plus à l'indentation précédente.

Indentation dans les boucles, les conditions. Remarquons l'usage des parenthèses (notion de bloc) :

- ouvrante en fin de ligne,
- fermante sous l'identifieur qui l'a ouverte.

Voir les exemples ci-dessus dans le paragraphe : littéral et constante.

7. En-tête GPL pour le code

Par exemple :

```
#**************************************************************************
#  Copyright (C) 2008 David Lucardi <davidlucardi@aol.com>
#
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 2 of the License, or
#  any later version.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
```

```
#  along with this program; if not, write to the Free Software
#  Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307,
USA.
#
#*****************************************************************************
#  File  : $$$
#  Author  : davidlucardi@aol.com
#  Modifier: andre.connes@toulouse.iufm.fr
#  Date    : 24/04/2002
#  Licence : GNU/GPL Version 2 ou plus
#
#  Description:
#  ------------
#
#  @version
#  @author     David Lucardi
#  @modifier   Andre Connes
#  @project    Le terrier
#  @copyright  Eric Seigne 24/04/2002
#
#  *****************************************************************************
```

8. Internationalisation

Il est souhaitable de penser à l'internationalisation des logiciels pour l'utilisateur (peu importe le programmeur).

Deux fichiers paraissent essentiels :

- un fichier de configuration .conf
- un (des) fichier(s) de messages traduits préfixés (fr_ uk_ etc.)

8.1 Le fichier de configuration

Ce fichier précise la langue utilisée par l'utilisateur, par l'élève, plus éventuellement d'autres paramètres tels que dimension de l'écran, etc.

Exemple du fichier de configuration lapins.conf :

```
    # lapins.tcl configuration file
    set LG fr ;# ... es fr it uk us ...

    set WIDTH_SIZE 640 ;#800
    set HORIZONTAL_SIZE 450 ;#600
```

ou bien, c'est bien mieux !

```
    set LG [lindex [split $env(LANG) "_"] 0]
```

8.2 Le fichier des messages

Ce fichier n'est défini que pour les langues autres que celle du développement.

Ce fichier contient les messages traduits. Nous utilisons pour cela le module “msgcat” ; pour cela il est nécessaire d'insérer au préalable dans le code le fichier msg.tcl :

```
    source msg.tcl.
```

Exemple :

```
    .menu add cascade -label [mc "À propos"] -menu .menu.fichier
```

Exception : si le message doit contenir une variable, par exemple $var, le message doit être composé d'un seul mot suivi ou précédé de la variable.

Exemple :

```
    .menu add cascade -label "[mc Aide] $var" -menu .menu.fichier
```

Autre exception : le message contient plusieurs variables séparées par des mots à traduire. Dans ce cas, l'exemple suivant paraît suffisamment clair et se passe de tout commentaire.

```
append titre "ALLER - " [mc "Groupe"] " : " $groupe " / " [mc "Dossier"] " :
" $rep " / " [mc "Texte"] " : " $demarre
wm title . $titre
```

et précédemment nous aurions pu écrire :

```
append message [mc "Aide]" $var
.menu add cascade -label $message -menu .menu.fichier
```

8.3 Pour traduire, comment ça marche ?

Voir la doc sur le site abuledu :

```
http://docs.abuledu.org/abuledu/developpeur/les_logiciels_du_terrier_faciles
_a_traduire
```

9. Le courriel sur la liste beta du groupe "le_terrier"

L'objet de cette partie est de permettre de trier les messages au moyen de filtres.

Divers projets ont été développés, sont en cours de développement, ou en phase de test ; certains écrits en Tcl/Tk, d'autres en python, java... Pour mémoire :

- mulot
- associations
- aller
- calculs
- bonjour poussins

Dans ce contexte, pour chacun des projets, **précisons ce projet dans le champ 'objet' du message en écrivant en abrégé (lt- mis pour “le_terrier”) entre chevrons []** :

- lt-mulot
- lt-associssions
- lt-aller
- lt-calculs
- lt-poussins

Et précisons l'objet. Enfin, nous pouvons commencer le corps du message par un **bonjour** et le terminer par un **au revoir**.

Exemple :

```
   [lt-mulot] saisir désignation cellule dans mulot.tcl
```

10. A faire

- répertoire de sauvegarde des traces-élèves (utilisation d'une variable d'environnement)
- format de la trace-élève (a priori en XML) ; les champs sont à définir
- language de développement futur : python+Qt ou java ?

Si nous choisissons python+Qt, un bon outil de développement serait Qt-designer

Utiliser la forge : gforge.ryxeo.com