by Hilaire Fernandes
<hilaire(at)ofset.org>

About the author:

Hilaire Fernandes é il vicepresidente della OFSET, un'organizzazione che promuove lo sviluppo del Software 'Libero' per l'istruzione su desktop Gnome. Ha scritto anche Dr. Geo, un programma per principianti di geometria dinamica, e sta attualmente lavorando su Dr. Genius, un altro programma educativo per Gnome.

Translated to English by:
Lorne Bailey
<sherm_pbody(at)yahoo.com>

Sviluppare Applicazioni per Gnome con Python (Parte 3)

Abstract:

Questa serie di articoli é scritta principalmente per chi si avvicina per la prima volta alla programmazione Gnome con Linux. Python, il linguaggio scelto per lo sviluppo, evita le complicazioni dei linguaggi compilati come il C. Le informazioni in questo articolo assumono che abbiate un'infarinatura della programmazione Python. Maggiori informazioni su Python e Gnome sono disponibili a http://www.python.org e http://www.gnome.org.

Articoli precedenti di questa serie :
– primo articolo
– secondo articolo

Tool richiesti

Per conoscere i tool necessari all'esecuzione dei programmi contenuti in questo articolo, fate riferimento alla lista contenuta nel primo articolo di questa serie.

Vi servirà inoltre:

- Il file .glade originale[drill.glade] . Questo file é stato leggermente modificato dall'ultima volta per aggiungere delle barre scorrevoli per la selezione degli esercizi nell'interfaccia.
- Questa volta i sorgenti Python vengono distribuiti in quattro file :
    1. [drill1.py].
    2. [templateExercice.py].

3. [colorExercice.py].
4. [labelExercice.py].

Per l'installazione e l'uso di Python−Gnome e di LibGlade fate riferimento alla parte 1.

Modello di Sviluppo per gli esercizi

Nel precedente articolo (parte 2) abbiamo creato l'interfaccia utente −− ***Drill*** −− che é una gabbia per la programmazione degli esercizi che descriveremo in seguito. Ora daremo un'occhiata più da vicino alla programmazione orientata agli oggetti di Python, per aggiungere funzionalità a ***Drill***. In questo studio, lasceremo da parte gli aspetti della programmazione Gnome con Python.

Riprendiamo da dove avevamo interrotto, l'inserimento di un gioco dei colori dentro ***Drill*** come esercizio per il lettore. Lo useremo per illustrare il nostro obiettivo attuale e allo stesso tempo offrire una soluzione all'eercizio.

Sviluppo Orientato agli Oggetti

Brevemente, senza voler analizzare esaustivamente il problema, la programmazione orientata agli oggetti tenta di definire e dividere in categorie le cose in base a relazioni ***é un*** ("is a"), che esistano o meno nel mondo reale. Può essere visto come una astrazione degli oggetti relativi al problema del quale siamo interessati. Possiamo trovare comparazioni in diversi campi come le categorie di Aristotele, le tassonomie o le ontologie. In ogni caso si deve capire una situazione complessa attraverso un'astrazione. Questo tipo di sviluppo avrebbe potuto benissimo chiamarsi sviluppo orientato alla categoria.

In questo modello di sviluppo gli oggetti manipolati dal programma, o che costituiscono il programma, vengono chiamati ***classi*** e rappresentazioni di questi oggetti astratti sono le ***istanze***. Le classi contengono ***attributi*** (che contengono valori) e ***metodi*** (funzioni). Si parla di una relazione padre−figlio per una certa classe quando una classe figlia eredita delle proprietà da un genitore. Le classi sono organizzate in relazioni ***é un***, dove un'istanza del figlio ***é un***'istanza dei metodi e attributi della classe padre, oltre che dei metodi e attributi della classe figlio. Le classi possono essere non completamente definite, nel qual caso vengono chiamate classi astratte. Quando un metodo viene dichiarato ma non definito (il corpo della funzione é vuoto) viene chiamata metodo virtuale. Una classe virtuale ha uno o più metodi come questo e quindi non può essere istanziata. Le classi astratte permettono di definire la forma che le classi derivate dovranno avere − classi figlie in cui i metodi puramente virtuali saranno definiti.

Diversi linguaggi hanno sintassi più o meno eleganti per definire gli oggetti, ma il denominatore comune sembra essere il seguente:

1. Ereditarietà degli attributi e metodi del genitore da parte del figlio.
2. Possibilità della classe figlio di sovrascrivere e sovraccaricare i metodi ereditati dal genitore.
3. Polimorfismo, dove una classe può avere più di una classe genitrice.

Python e la Programmazione Orientata agli Oggetti

Nel caso di Python questo é il minimo comun denominatore scelto. Permette di imparare la programmazione orientata agli oggetti senza perdersi nel dettagli della metodologia.

In Python i metodi degli oggetti sono sempre metodi virtuali. Questo significa che possono sempre essere sovrascritti da una classe figlia –– che normalmente é quello che vogliamo quando programmiamo a oggetti –– e che semplifica la sintassi. Ma non è facile distinguere tra metodi sovrascritti e non. Per di più è impossibile rendere opaco un oggetto e quindi impedire l'accesso ai suoi attributi o metodi da parte di un oggetto esterno.. In comclusione, gli attributi di un oggetto in Python sono sia leggibili che scrivibili da un oggetto esterno.

Esercizio della Classe Genitrice

Nel nostro esempio (vedi file `templateExercice.py`), vorremmo definire molti oggetti di tipo `exercice`. Definiamo un oggetto di tipo `exercice` allo scopo di avere una classe base astratta per derivarne altri esercizi che creeremo più avanti. L'oggetto `example` è la classe genitrice di tutti gli altri tipi di esercizi creati. Questi tipi di esercizi derivati avranno almeno gli stessi attributi e metodi della classe `exercice` perché li erediteranno. Questo ci permetterà di manipolare tutti i diversi tipi di oggetti–esercizi in modo uguale, senza badare a quale classe appartengono.

Per esempio, per creare un'istanza della classe `exercice` possiamo scrivere :

```
from templateExercice import exercice

monExercice = exercice ()
monExercice.activate (ceWidget)
```

In effetti non c'è bisogno di creare un'istanza della classe `exercice` perché è solo un modello da cui altre classi derivano.

Attributi

- `exerciceWidget` : il widget che contiene l'interfaccia utente dell'esercizio;
- `exerciceName` : il nome dell'esercizio.

Se siamo interessati negli altri aspetti dell'esercizio possiamo aggiungere attributi, per esempio il punteggio o il numero di volte che un esercizio è stato lanciato.

Metodi

- `__init__ (self)` : questo metodo ha un ruolo preciso negli oggetti in Python. Viene chiamato automaticamente durante la creazione di una istanza di questo oggetto. Per questa ragione viene chiamato anche costruttore. L'argomento `self` è un riferimento all'istanza della classe `exercice` che ha chiamato il metodo `__init__`. È sempre necessario specificare questo argomento nei metodi, quindi un metodo deve sempre avere almeno un argomento. Fate attenzione, poiché questo argomento viene aggiunto automaticamente da Python, quindi non è necessario includerlo quando richiamate il metodo. L'argomento `self` consente l'accesso agli attributi e agli altri metodi di una istanza. Senza di

esso, tale accesso sarebbe impossibile. Scendereme nei dettagli più avanti.

- `activate (self, area)` : attiva questa istanza di esercizio posizionando il suo widget nella zona preposta dell'interfaccia di ***Drill***. L'argomento `area` è un container (contenitore) GTK+ che controlla il posizionamento del widget all'interno di ***Drill***. Sapendo che l'attributo `exerciceWidget` contiene il widget dell'esercizio, è sufficiente chiamare `area.add(self.exerciceWidget)` per incorporare l'esercizio in ***Drill***.
- `unactivate (self, area)` : rimuove il widget dal contenitore di ***Drill***. In termini di posizionamento, è l'operazione inversa, quindi sarà sufficiente chiamare `area.remove (self.exerciceWidget).`
- `reset (self)` : reimposta l'esercizio annullando tutto.

In codice Python, il risultato è il seguente :

```
class exercice:
    "Una base per gli esercizi"
    exerciceWidget = None
    exerciceName = "No Name"
    def __init__ (self):
        "Crea il widget dell'esercizio"
    def activate (self, area):
        "Visualizza l'esercizio nell'area-contenitore"
        area.add (self.exerciceWidget)
    def unactivate (self, area):
        "Rimuove l'esercizio dal contenitore"
        area.remove (self.exerciceWidget)
    def reset (self):
        "Annulla l'esercizio"
```

Questo codice è incluso in un file a parte, `templateFichier.py`, che ci permette di di chiarire lo specifico ruolo di ogni oggetto. I metodi vengono dichiarati all'interno della classe `exercice`, e sono a tutti gli effetti delle funzioni.

Vedremo che l'argomento `area` è un riferimento al widget GTK+ costruito da LibGlade, è una finestra con barre di scorrimento.

In questo oggetto, i metodi `__init__` e `reset` sono vuoti e verranno sovrascritti dalle classe figlie se necessario.

labelExercice, Primo Esempio di Ereditarietà

Quello che andiamo ad analizzare è sostanzialmente un esercizio vuoto. Fa solo una cosa, mette il nome dell'esercizio nella zona per gli esercizi di ***Drill***. Serve come punto di partenza per gli esercizi che popolano l'albero nella parte sinistra di ***Drill***, ma che non abbiamo ancora creato.

Allo stesso modo dell'oggetto `exercice`, l'oggetto `labelExercice` è contenuto in un suo file, `labelExercice.py`. Fatto questo, poiché questo oggetto è figlio dell'oggetto `exercice` dovremo specificare come è definito il genitore. Lo facciamo con una import :

```
from templateExercice import exercice
```

Questo significa, letteralmente, che la definizione della classe `exercice` nel file

`templateExercice.py` viene importata nel codice corrente.

Veniamo ora all'aspetto più importante, la dichiarazione della classe `labelExercice` come figlio della classe `exercice`.
`labelExercice` viene dichiarata in questo modo :

```
class labelExercice(exercice):
```

Voilà, è abbastanza per fare in modo che `labelExercice` erediti tutti gli attributi e metodi di `exercice`.

Naturalmente abbiamo ancora del lavoro da fare, in particolare dobbiamo inizializzare il widget dell'esercizio. Lo possiamo fare sovrascrivendo il metodo `__init__` (per esempio ridefinendolo nella classe `labelExercice`); quest'ultimo viene chiamato quando viene creata un'istanza. Inoltre questo widget deve essere passato nell'attributo `exerciceWidget` in modo da non dover sovrascrivere i metodi `activate` e `unactivate` della classe genitrice `exercice`.

```
    def __init__ (self, name):
        self.exerciceName = "Un exercice vide" (un esercizio vuoto)
        self.exerciceWidget = GtkLabel (name)
        self.exerciceWidget.show ()
```

Questo è l'unico metodo che sovrascriviamo. Per creare un'istanza di `labelExercice` dobbiamo solo richiamare :

```
monExercice = labelExercice ("Un exercice qui ne fait rien")
(NdT: "Un exercice qui ne fait rien" significa "un esercizio che non fa nulla")
```

Per accedere ai suoi metodi e attributi :

```
# Nome dell'esercizio
print monExercice.exerciceName

# Mettiamo il widget dell'esercizio nel contenitore "area"
monExerice.activate (area)
```

colorExercice, Secondo esempio di Ereditarietà

Qui inizialo la trasformazione del gioco dei colori visto nel primo articolo della serie in una classe di tipo `exercice` che chiameremo `colorExercice`. La metteremo in un suo file, `colorExercice.py`, che verrà allegato a questo articolo con il codice sorgente completo.

Le modifiche richieste al codice sorgente iniziale consistono principalmente nella redistribuzione delle funzioni e delle variabili come metodi e attributi della classe `colorExercice`.

Le variabili globali vengono trasformate in attributi dichiarati all'inizio della classe :

```
class colorExercice(exercice):
    width, itemToSelect = 200, 8
    selectedItem = rootGroup = None
    # to keep trace of the canvas item
    colorShape = []
```

Come per la classe `labelExercice`, il metodo `__init__` viene sovrascritto per consentire la costruzione del widget dell'esercizio :

```
def __init__ (self):
```

```
    self.exerciceName = "Le jeu de couleur" # NdT: Il gioco dei colori
    self.exerciceWidget = GnomeCanvas ()
    self.rootGroup = self.exerciceWidget.root ()
    self.buildGameArea ()
    self.exerciceWidget.set_usize (self.width,self.width)
    self.exerciceWidget.set_scroll_region (0, 0, self.width, self.width)
    self.exerciceWidget.show ()
```

Niente di nuovo in confronto al codice iniziale, se non per lo `GnomeCanvas` che viene memorizzato e visto come attributo `exerciceWidget`.

L'altro metodo sovrascritto è `reset`. Poiché reimposta il gioco allo stato iniziale, deve essere personalizzato per il gioco dei colori :

```
    def reset (self):
        for item in self.colorShape:
            item.destroy ()
        del self.colorShape[0:]
        self.buildGameArea ()
```

Gli altri metodi sono copie dirette delle funzioni originali, con l'aggiunta dell'uso delle variabili `self` per consentire l'accesso agli attributi e ai metodi dell'istanza. C'è un'eccezione rappresentata dai metodi `buildStar` e `buildShape` dove il parametro decimale `k` viene rimpiazzato da un numero intero. Ho notato strani comportamenti nel documento `colorExercice.py` dove il numero decimale preso dal mio codice sorgente veniva troncato. Il problema sembra essere legato al modulo `gnome.ui` e alla localizzazione francese (dove i numeri decimali usano la virgola come separatore dei decimali al posto del punto) (NdT: anche in Italia si usa la stessa convenzione). Cercherò la soluzione al problema prima del prossimo articolo.

Affinamenti finali in Drill

Abbiamo quindi due tipi di esercizi –– `labelexercice` e `colorExercice`. Creeremo delle istanze a queste classi con le funzioni `addXXXXXExercice` nel codice in `drill1.py`. Le istanze vengono memorizzate in un dizionario `exerciceList` nel quale le chiavi sono anche argomenti per le pagine di ogni esercizio nell'albero nella parte sinistra:

```
def addExercice (category, title, id):
    item = GtkTreeItem (title)
    item.set_data ("id", id)
    category.append (item)
    item.show ()
    item.connect ("select", selectTreeItem)
    item.connect ("deselect", deselectTreeItem)
[...]
def addGameExercice ():
    global exerciceList
    subtree = addSubtree ("Jeux")
    addExercice (subtree, "Couleur", "Games/Color")
    exerciceList ["Games/Color"] = colorExercice ()
```

La funzione `addGameExercice` crea una foglia nell'albero con l'attributo `id="Games/Color"` chiamando la funzione `addExercice`. Questo attributo viene usato come chiave per l'istanza del gioco dei colori creato dal comando `colorExercice()` nel dizionario `exerciceList`.

Ora, grazie all'eleganza del polimorfismo nella programmazione a oggetti, possiamo lanciare gli esercizi usando le stesse funzioni che si comportano diversamente per ogni oggetto senza doverci preoccupare della loro implementazione interna. Chiamiamo solo metodi definiti nella classe astratta `exercice` e agiscono diversamente nella classe colorExercice rispetto a labelExercice. Il programmatore "parla" agli esercizi allo

stesso modo, anche se la "risposta" di ogni esercizio è leggermente diversa. Per ottenere questo combiniamo l'uso dell'attributo `id` nelle pagine dell'albero e del dizionario `exerciceList` o della variabile `exoSelected` che si riferisce all'esercizio in uso. Dato che tutti gli esercizi sono figli della classe `exercice` usiamo i suoi metodi allo stesso modo per controllare tutti gli esercizi.

```
def on_new_activate (obj):
    global exoSelected
    if exoSelected != None:
        exoSelected.reset ()

def selectTreeItem (item):
    global exoArea, exoSelected, exerciceList
    exoSelected = exerciceList [item.get_data ("id")]
    exoSelected.activate (exoArea)

def deselectTreeItem (item):
    global exoArea, exerciceList
    exerciceList [item.get_data ("id")].unactivate (exoArea)
```

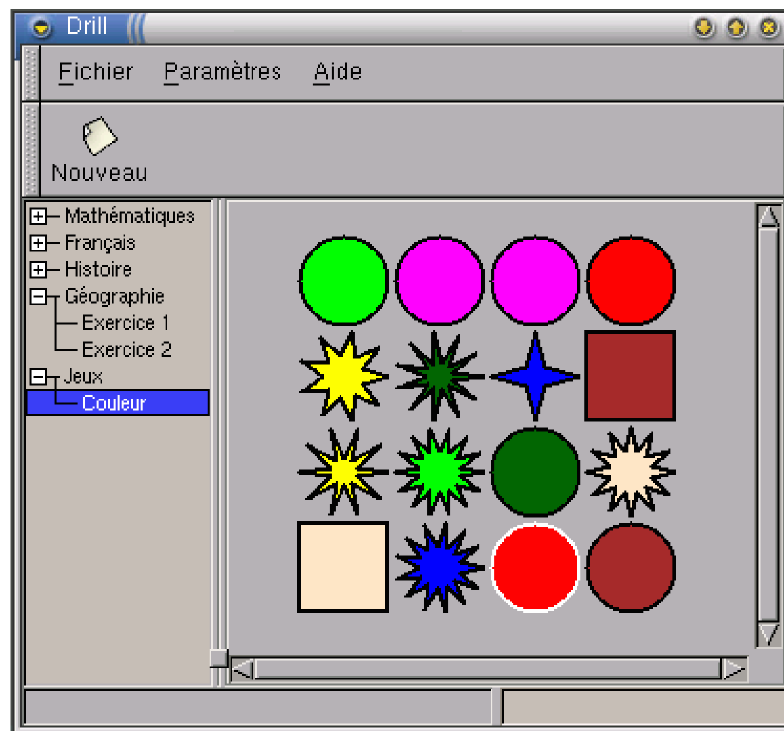

Fig. 1 – Finestra principale di Drill, con l'esercizio dei colori

Finisce qui il nostro articolo. Abbiamo scoperto le attrattive della programmazione orientata agli oggetti con Python nell'ambito delle interfacce grafiche. Nel prossimo articolo continueremo a scoprire i widget di Gnome programmando nuovi esercizi da inserire in ***Drill***.

Appendice: Codice Sorgente Completo

drill1.py

```
#!/usr/bin/python
# Drill - Teo Serie
# Copyright Hilaire Fernandes 2002
# Release under the terms of the GPL licence
# You can get a copy of the license at http://www.gnu.org


from gnome.ui import *
from libglade import *

# Importa la classe exercice
from colorExercice import *
from labelExercice import *

exerciceTree = currentExercice = None
# Il contenitore degli esercizi
exoArea = None
exoSelected = None
exerciceList = {}

def on_about_activate(obj):
    "mostra la finestra di informazioni (about)"
    about = GladeXML ("drill.glade", "about").get_widget ("about")
    about.show ()

def on_new_activate (obj):
    global exoSelected
    if exoSelected != None:
        exoSelected.reset ()

def selectTreeItem (item):
    global exoArea, exoSelected, exerciceList
    exoSelected = exerciceList [item.get_data ("id")]
    exoSelected.activate (exoArea)

def deselectTreeItem (item):
    global exoArea, exerciceList
    exerciceList [item.get_data ("id")].unactivate (exoArea)

def addSubtree (name):
    global exerciceTree
    subTree = GtkTree ()
    item = GtkTreeItem (name)
    exerciceTree.append (item)
    item.set_subtree (subTree)
    item.show ()
    return subTree

def addExercice (category, title, id):
    item = GtkTreeItem (title)
    item.set_data ("id", id)
    category.append (item)
    item.show ()
    item.connect ("select", selectTreeItem)
    item.connect ("deselect", deselectTreeItem)


def addMathExercice ():
    global exerciceList
    subtree = addSubtree ("Mathématiques")
    addExercice (subtree, "Exercice 1", "Math/Ex1")
    exerciceList ["Math/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "Math. Ex2")
    exerciceList ["Math/Ex2"] = labelExercice ("Exercice 2")
```

```
def addFrenchExercice ():
    global exerciceList
    subtree = addSubtree ("Français")
    addExercice (subtree, "Exercice 1", "French/Ex1")
    exerciceList ["French/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "French/Ex2")
    exerciceList ["French/Ex2"] = labelExercice ("Exercice 2")

def addHistoryExercice ():
    global exerciceList
    subtree = addSubtree ("Histoire")
    addExercice (subtree, "Exercice 1", "Histoiry/Ex1")
    exerciceList ["History/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "Histoiry/Ex2")
    exerciceList ["History/Ex2"] = labelExercice ("Exercice 2")

def addGeographyExercice ():
    global exerciceList
    subtree = addSubtree ("Géographie")
    addExercice (subtree, "Exercice 1", "Geography/Ex1")
    exerciceList ["Geography/Ex1"] = labelExercice ("Exercice 1")
    addExercice (subtree, "Exercice 2", "Geography/Ex2")
    exerciceList ["Geography/Ex2"] = labelExercice ("Exercice 2")

def addGameExercice ():
    global exerciceList
    subtree = addSubtree ("Jeux")
    addExercice (subtree, "Couleur", "Games/Color")
    exerciceList ["Games/Color"] = colorExercice ()


def initDrill ():
    global exerciceTree, label, exoArea
    wTree = GladeXML ("drill.glade", "drillApp")
    dic = {"on_about_activate": on_about_activate,
           "on_exit_activate": mainquit,
           "on_new_activate": on_new_activate}
    wTree.signal_autoconnect (dic)
    exerciceTree = wTree.get_widget ("exerciceTree")
    # Temporary until we implement real exercice
    exoArea = wTree.get_widget ("exoArea")
    # Free the GladeXML tree
    wTree.destroy ()
    # Add the exercice
    addMathExercice ()
    addFrenchExercice ()
    addHistoryExercice ()
    addGeographyExercice ()
    addGameExercice ()

initDrill ()
mainloop ()
```

templateExercice.py

```
# Classe virtuale pure Exercice
# i metodi di exercice dovrebbero essere sovrascritti
# nelle classi derivate
class exercice:
    "Uno schema di esercizio"
```

```
    exerciceWidget = None
    exerciceName = "No Name"
    def __init__ (self):
        "Crea il widget dell'esercizio"
    def activate (self, area):
        "Imposta l'esercizio nell'area contenitrice"
        area.add (self.exerciceWidget)
    def unactivate (self, area):
        "Toglie l'esercizio dal contenitore"
        area.remove (self.exerciceWidget)
    def reset (self):
        "Reimposta l'esercizio"
```

labelExercice.py

```
# Dummy Exercice - Teo Serie
# Copyright Hilaire Fernandes 2001
# Release under the terms of the GPL licence
# You can get a copy of the license at http://www.gnu.org

from gtk import *
from templateExercice import exercice

class labelExercice(exercice):
    "Un esercizio stupido, semplicemente scrive una frase nell'area degli eserc
 +++izi"
    def __init__ (self, name):
        self.exerciceName = "Un exercice vide"
        self.exerciceWidget = GtkLabel (name)
        self.exerciceWidget.show ()
```

colorExercice.py

```
# Color Exercice - Teo Serie
# Copyright Hilaire Fernandes 2001
# Release under the terms of the GPL licence
# You can get a copy of the license at http://www.gnu.org

from math import cos, sin, pi
from whrandom import randint
from GDK import *
from gnome.ui import *

from templateExercice import exercice


# Exercizio 1 : gioco dei colori

class colorExercice(exercice):
    width, itemToSelect = 200, 8
    selectedItem = rootGroup = None
    # to keep trace of the canvas item
    colorShape = []
    def __init__ (self):
        self.exerciceName = "Le jeu de couleur"
        self.exerciceWidget = GnomeCanvas ()
        self.rootGroup = self.exerciceWidget.root ()
```

```
        self.buildGameArea ()
        self.exerciceWidget.set_usize (self.width,self.width)
        self.exerciceWidget.set_scroll_region (0, 0, self.width, self.width)
        self.exerciceWidget.show ()
    def reset (self):
        for item in self.colorShape:
            item.destroy ()
        del self.colorShape[0:]
        self.buildGameArea ()
    def shapeEvent (self, item, event):
        if event.type == ENTER_NOTIFY and self.selectedItem != item:
            item.set(outline_color = 'white') #highligh outline
        elif event.type == LEAVE_NOTIFY and self.selectedItem != item:
            item.set(outline_color = 'black') #unlight outline
        elif event.type == BUTTON_PRESS:
            if not self.selectedItem:
                item.set (outline_color = 'white')
                self.selectedItem = item
            elif item['fill_color_gdk'] == self.selectedItem['fill_color_gdk'] \
                 and item != self.selectedItem:
                item.destroy ()
                self.selectedItem.destroy ()
                self.colorShape.remove (item)
                self.colorShape.remove (self.selectedItem)
                self.selectedItem, self.itemToSelect = None, \
                 self.itemToSelect - 1
                if self.itemToSelect == 0:
                    self.buildGameArea ()
        return 1

    def buildShape (self,group, number, type, color):
        "costruisce una forma in base al 'type' (tipo) e 'color' (colore)"
        w = self.width / 4
        x, y, r = (number % 4) * w + w / 2, (number / 4) * w + w / 2, w / 2 - 2
        if type == 'circle':
            item = self.buildCircle (group, x, y, r, color)
        elif type == 'squarre':
            item = self.buildSquare (group, x, y, r, color)
        elif type == 'star':
            item = self.buildStar (group, x, y, r, 2, randint (3, 15), color)
        elif type == 'star2':
            item = self.buildStar (group, x, y, r, 3, randint (3, 15), color)
        item.connect ('event', self.shapeEvent)
        self.colorShape.append (item)

    def buildCircle (self,group, x, y, r, color):
        item = group.add ("ellipse", x1 = x - r, y1 = y - r,
                          x2 = x + r, y2 = y + r, fill_color = color,
                          outline_color = "black", width_units = 2.5)
        return item

    def buildSquare (self,group, x, y, a, color):
        item = group.add ("rect", x1 = x - a, y1 = y - a,
                          x2 = x + a, y2 = y + a, fill_color = color,
                          outline_color = "black", width_units = 2.5)
        return item

    def buildStar (self,group, x, y, r, k, n, color):
        "k: fattore da cui ricavare il raggio interno"
        "n: numero di rami"
        angleCenter = 2 * pi / n
        pts = []
        for i in range (n):
            pts.append (x + r * cos (i * angleCenter))
```

```
            pts.append (y + r * sin (i * angleCenter))
            pts.append (x + r / k * cos (i * angleCenter + angleCenter / 2))
            pts.append (y + r / k * sin (i * angleCenter + angleCenter / 2))
        pts.append (pts[0])
        pts.append (pts[1])
        item = group.add ("polygon", points = pts, fill_color = color,
                          outline_color = "black", width_units = 2.5)
        return item

    def getEmptyCell (self, l, n):
        "prende l'n-esimo elemento non nullo di l"
        length, i = len (l), 0
        while i < length:
            if l[i] == 0:
                n = n - 1
            if n < 0:
                return i
            i = i + 1
        return i

    def buildGameArea (self):
        itemColor = ['red', 'yellow', 'green', 'brown', 'blue', 'magenta',
                     'darkgreen', 'bisque1']
        itemShape = ['circle', 'squarre', 'star', 'star2']
        emptyCell = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
        self.itemToSelect, i, self.selectedItem = 8, 15, None
        for color in itemColor:
            # two items of same color
            n = 2
            while n > 0:
                cellRandom = randint (0, i)
                cellNumber = self.getEmptyCell (emptyCell, cellRandom)
                emptyCell[cellNumber] = 1
                self.buildShape (self.rootGroup, cellNumber, \
                 itemShape[randint (0, 3)], color)
                i, n = i - 1, n - 1
```


Webpages maintained by the LinuxFocus Editor team © Hilaire Fernandes "some rights reserved" see linuxfocus.org/license/ http://www.LinuxFocus.org	Translation information: fr --> -- : Hilaire Fernandes <hilaire(at)ofset.org> fr --> en: Lorne Bailey <sherm_pbody(at)yahoo.com> en --> it: Alessandro Pellizzari <alex(at)neko.it>

2005-01-10, generated by lfparser_pdf version 2.51