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

About the author:

Hilaire Fernandes ist der Vizepräsident von OFSET, einer Organisation, die die Entwicklung 'freier' Lehrsoftware für den Gnome−Desktop fördert. Er hat auch Dr. Geo geschrieben, ein Grundlagenprogramm für dynamische Geometrie und er arbeitet gegenwärtig an Dr. Genius − ein anderes Lehrprogramm für Gnome.

Gnome Programmentwicklung mit Python (Teil 3)

Abstract:

Diese Artikelserie ist insbesondere Programmieranfängern gewidmet, die Gnome und GNU/Linux benutzen. Die für das Projekt gewählte Sprache Python vermeidet den üblichen Overhead von Compilersprachen wie C. Um diesen Artikel zu verstehen braucht man ein Grundverständnis von Python−Programmierung. Mehr Informationen über Python und Gnome kann man bei http://www.python.org und http://www.gnome.org finden.

Frühere Artikel dieser Serie:
1. Artikel
2. Artikel

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

Benötigte Werkzeuge

Wegen der Softwareabhängigkeiten, um das Programm auszuführen, das in diesem Artikel beschrieben wird, lesen Sie bitte die Liste aus dem ersten Artikel dieser Serie.

Sie werden ebenso folgendes benötigen:

- Die ursprüngliche .glade−Datei [drill.glade]. Diese Datei wurde leicht abgeändert seit dem letzten Mal, um Schieberegler einzubauen, um Übungen auf der Oberfläche auszuwählen.
- Dieses Mal wird der Quelltext des Pythonprogramms in vier Dateien verteilt:
    1. [drill1.py].
    2. [templateExercice.py].

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

Um Python−Gnome und LibGlade zu installieren und zu benutzen, lesen Sie bitte den ersten Artikel.

Entwicklungsmodell für die Übungen

Im vorigen, zweiten Artikel haben wir die Benutzerschnittstelle −− ***Drill*** −− erzeugt, die einen Rahmen für das Programmieren der im weiteren beschriebenen Übungen bildet. Nun werden wir die objektorientierte Entwicklung mit Python genauer betrachten, um ***Drill*** neue Funktionen hinzuzufügen. Bei dieser Untersuchung werden wir die Gnome−Aspekte von Python−Programmierung ausser Acht lassen.

Also lassen Sie uns da weitermachen, wo wir aufgehört haben: beim Einfügen eines Farbenspiels in ***Drill*** als Übung für den Leser. Wir werden dies benutzen, um unser gegenwärtiges Thema zu illustrieren und gleichzeitig eine Lösung für diese Übung anzubieten.

Objektorientierte Entwicklung

Kurz gesagt, ohne Anspruch auf eine gründliche Analyse zu erheben, versucht objektorientierte Entwicklung Dinge durch ***ist ein***−Beziehungen zu definieren und einzuordnen, ob sie nun in der physischen Welt existieren oder nicht. Dies kann man als Abstrahierung der Objekte sehen, die sich auf das Problem beziehen, welches uns interessiert. Wir können Vergleichbares in anderen Bereichen finden, so zum Beispiel die Kategorien des Aristoteles, die Klassifizierungslehren oder Seinslehren. In jedem Fall muß man eine komplexe Situation durch eine Abstrahierung verstehen. Diese Art der Entwicklung hätte sehr wohl auch kategorieorientierte Entwicklung genannt werden können.

In diesem Entwicklungsmodell werden die Objekte, die vom Programm verwendet werden oder das Programm darstellen ***Klassen***, genannt und die Darstellung dieser abstrakten Objekte sind ***Instanzen***. Klassen werden durch ***Attribute*** (enthalten Werte) und ***Methoden*** (Funktionen) definiert. Wir sprechen von einer Eltern−Kind−Beziehung bei einer gegebenen Klasse, bei der eine Kind−Klasse Eigenschaften von den Eltern erbt. Klassen sind in einer ***ist ein***−Beziehung organisiert, bei der das Kind noch ein ***ist ein***−Typ von Eltern ist, ebenso wie ein Kind−Typ. Klassen sind vielleicht nicht vollständig definiert, in welchem Fall sie abstrakte Klassen genannt werden. Wenn eine Methode deklariert wird, aber nicht definiert wird (der Funktionsblock ist leer), wird es auch virtuelle Methode genannt. Eine abstrakte Klasse hat eine oder mehrere dieser undefinierten Methoden und kann deshalb nicht instanziiert werden. Abstrakte Klassen erlauben Spezifizierungen in der Form der übernommenen Klassen − Kind−Klassen, in denen die reinen virtuellen Methoden definiert werden.

Verschiedene Sprachen sind mehr oder weniger geschickt beim Definieren von Objekten, aber der gemeinsame Nenner scheint folgendes zu sein:

1. Erben von Attributen und Methoden der Elternklasse durch das Kind.
2. Fähigkeit der Kind−Klasse, von der Elternklasse geerbte Methoden zu überschreiben und zu überladen.
3. Polimorphismus, bei der eine Klasse viele Elternklassen haben kann.

Python und objektorientierte Entwicklung

Bei Python wurde der kleinste gemeinsame Nenner gewählt. Dies erlaubt objektorientierte Entwicklung zu lernen, ohne sich in den Details dieser Methodik zu verlieren.

Bei Python sind die Objektmethoden immer virtuelle Methoden. Dies bedeutet, daß sie immer von einer Kind−Klasse überschrieben werden können −− was im Allgemeinen das ist, was wir wollen, wenn wir objektorientierte Entwicklung anwenden −− und was die Syntax leicht vereinfacht. Aber es ist nicht einfach zwischen Methoden zu unterscheiden, die überschrieben werden oder bei denen das nicht der Fall ist. Desweiteren ist es unmöglich, ein Objekt privat zu machen und damit Zugang zu Attributen und Methoden von außerhalb eines Objekts zu verweigern. Als Folge sind Python−Objekte von außerhalb des Objekts lesbar und beschreibbar.

Übung zur Elternklasse

In unserem Beispiel (siehe die Datei `templateExercice.py`) wollen wir viele Objekte des Typs `exercice` definieren. Wir definieren ein Objekt des Typs `exercice`, das uns als abstrakte Klasse dienen soll, um andere Übungen abzuleiten, die wir später erzeugen werden. Das Objekt `exemple` ist die Elternklasse aller anderen erzeugten Übungstypen. Diese abgeleiteten Übungstypen werden mindestens dieselben Attribute und Methoden wie die Klasse `exercice` haben, weil sie sie erben werden. Dies wird uns erlauben, alle verschiedenen Typen der exercice−Objekte auf die gleiche Weise zu beeinflussen, ohne Rücksicht auf das Objekt von denen sie instanziiert wurden.

Um zum Beispiel eine Instanz der Klasse `exercice` zu erzeugen, können wir schreiben:

```
from templateExercice import exercice

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

Eigentlich besteht keine Notwendigkeit, eine Instanz der Klasse `exercice` zu erzeugen, weil es nur eine Schablone ist, von der andere Klassen abgeleitet werden.

Attribute

- `exerciceWidget`: Das Widget enthält die Benutzerschnittstelle von exercice;
- `exerciceName`: der Name der Übung.

Wenn wir uns für andere Aspekte einer Übung interessieren, können wir Attribute hinzufügen, d.h. die erreichte Punktzahl oder wie oft sie ausgeführt wurde.

Methoden

- `__init__ (self)`: Diese Methode hat eine sehr konkrete Rolle in einem Python−Objekt. Sie wird automatisch während der Erzeugung einer Instanz dieses Objekts aufgerufen. Deswegen wird es auch Konstruktor genannt. Das Argument `self` ist ein Verweis auf die Instanz der Klasse

`exercice`, die die `__init__`–Methode aufgerufen hat. Es ist immer nötig, dieses Argument in Methoden zu spezifizieren, was bedeutet, daß eine Methode nicht null Argumente haben kann. Vorsicht, dieses Argument wird automatisch von Python hinzugefügt, so daß es nicht notwendig ist, es einzufügen, wenn man die Methode aufruft. Das Argument `self` erlaubt Zugriff auf die Attribute und andere Methoden einer Instanz. Ohne es ist ein solcher Zugriff unmöglich. Wir werden das später noch deutlicher sehen.

- `activate (self, area)`: aktiviert diese Instanz von exercice, indem es sein Widget in der exercice–Ebene von ***Drill*** plaziert. Das Argument `area` ist tatsächlich ein GTK+–Container, das die Plazierung des Widgets in ***Drill*** kontrolliert. Wenn man weiß, daß das Attribut `exerciceWidget` das Widget für die Übung enthält, muß man nur `area.add (self.exerciceWidget)` aufrufen, um exercice in ***Drill*** zu verwenden.
- `unactivate (self, area)`: entfernt das Widget vom Container ***Drill***. Von der Reihenfolge her ist dies die Gegenoperation, daher reicht es `area.remove (self.exerciceWidget)` aufzurufen.
- `reset (self)`: setzt die Übung auf Null zurück.

In Python sieht das wie folgt aus:

```
class exercice:
    "A template exercice"
    exerciceWidget = None
    exerciceName = "No Name"
    def __init__ (self):
        "Create the exericice widget"
    def activate (self, area):
        "Set the exercice on the area container"
        area.add (self.exerciceWidget)
    def unactivate (self, area):
        "Remove the exercice fromt the container"
        area.remove (self.exerciceWidget)
    def reset (self):
        "Reset the exercice"
```

Dieser Code ist in der Datei `templateFichier.py`, der uns erlaubt, die besonderen Rollen aller Objekte aufzuzeigen. Die Methoden werden in der Klasse `exercice` deklariert und sind eigentlich Funktionen.

Wir werden sehen, daß das Argument `area` ein Verweis auf das GTK+–Widget ist, das von LibGlade konstruiert wird. Es ist ein Fenster mit Schiebereglern.

In diesem Objekt sind die Methoden `__init__` und `reset` leer und werden von den Kind–Klassen überschrieben, wenn es notwendig ist.

labelExercice, erstes Beispiel der Vererbung

Dies ist eine fast leere Übung. Es macht nur eines: es legt den Namen der Übung in die Übungsebene von ***Drill***. Es dient als Anfang für die Übungen, die den linken Baum von ***Drill*** bevölkern, den wir aber noch nicht erzeugt haben.

Genauso, wie das Objekt `exercice`, ist das Objekt `labelExercice` in einer eigenen Datei abgelegt, `labelExercice.py` . Als nächstes müssen wir sagen, wie die Eltern definiert sind, da dieses Objekt ein

Kind des Objekts `exercice` ist. Dies geschieht einfach über eine Importierung:

```
from templateExercice import exercice
```

Dies bedeutet wörtlich, daß die Definition der Klasse `exercice` in die Datei `templateExercice.py` in den gegenwärtigen Programmtext importiert wird.

Wir kommen nun zum wichtigsten Punkt, der Deklaration der Klasse `labelExercice` als Kindklasse von `exercice`.
`labelExercice` wird folgendermaßen deklariert:

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

Voilà, das reicht, damit `labelExercice` alle Attribute und Methoden von `exercice` erbt.

Natürlich gibt es noch einiges zu tun, insbesondere müssen wir das Widget der Übung initialisieren. Das erreichen wir, indem wir die Methode `__init__` überschreiben (d.h.. indem wir sie in der Klasse `labelExercice` neu definieren), diese letztere wird aufgerufen, wenn eine Instanz erzeugt wird. Das Widget muß auch im Attribut `exerciceWidget` referenziert werden, damit wir nicht die Methoden `activate` und `unactivate` der Elternklasse `exercice` überschreiben müssen.

```
  def __init__ (self, name):
      self.exerciceName = "Un exercice vide" (Übersetzer: eine leere Übung)
      self.exerciceWidget = GtkLabel (name)
      self.exerciceWidget.show ()
```

Dies ist die einzige Methode, die wir überschreiben. Um eine Instanz von `labelExercice` zu erzeugen, muß man nur folgendes aufrufen:

```
monExercice = labelExercice ("Un exercice qui ne fait rien")
(Anm.d.Übers.: "Un exercice qui ne fait rien" bedeutet "eine Übung, die nichts macht")
```

Um auf ihre Attribute oder Methoden zuzugreifen:

```
# Le nom de l'exercice (Anm.d.Übers.: Der Name der Übung)
print monExercice.exerciceName

# Placer le widget de l'exercice dans le container "area"
# (Anm.d.Übers.: Das Widget der Übung in den Container
# "area" einfügen.)
monExerice.activate (area)
```

colorExercice, zweites Beispiel der Vererbung

Hier beginnen wir die Umwandlung der Spielfarbe, die wir im ersten Artikel dieser Serie gesehen haben, in eine Klasse des Typs `exercice`, diese werden wir `colorExercice` nennen. Wir speichern sie in einer eigenen Datei `colorExercice.py`, die diesem Artikel mit dem vollständigen Quelltext angefügt ist.

Die benötigten Änderungen im urspünglichen Quelltext bestehen nur aus einer Umverteilung von Funktionen und Variablen in Methoden und Attribute in der Klasse `colorExercice`.

Die globalen Variablen werden in Attribute umgewandelt, die am Beginn der Klasse deklariert werden:

```
class colorExercice(exercice):
    width, itemToSelect = 200, 8
    selectedItem = rootGroup = None
```

```
    # to keep trace of the canvas item
    colorShape = []
```

Wie in die Klasse `labelExercice` wird die Methode `__init__` überschrieben, um die Konstruktion der Übungswidgets zu anzupassen:

```
def __init__ (self):
    self.exerciceName = "Le jeu de couleur" # Anm.d.Übers.: die Farbspiel
    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 ()
```

Nichts Neues verglichen mit dem ursprünglichen Code, wenn es nur der `GnomeCanvas` ist, auf den im Attribut `exerciceWidget` verwiesen wird.

Die andere überschriebene Methode ist `reset`. Da es das Spiel auf Null zurücksetzt, muß es für das Farbspiel angepaßt werden:

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

Die anderen Methoden sind direkte Kopien der originalen Funktionen, mit der hinzugefügten Nutzung der Variablen `self`, um den Zugriff auf die Attribute und Methoden der Instanz zu erlauben. Es gibt eine Ausnahme in den Methoden `buildStar` und `buildShape`, wo der dezimale Parameter `k` durch eine ganze Zahl ersetzt wird. Ich bemerkte ein merkwürdiges Verhalten im Dokument `colorExercice.py`, wo die Dezimalzahlen, die vom Quelltext gelesen werden, abgeschnitten werden. Das Problem scheint im Modul `gnome.ui` zu sein und bei der französischen Lokalisierung (wo Dezimalzahlen ein Komma als Trenner benutzen, anstatt eines Punkts). Ich werde daran arbeiten, die Quelle des Problems vor dem nächsten Artikel zu finden.

Letzte Einstellungen in Drill

Wir haben nun zwei Typen von Übungen –– `labelExercice` und `colorExercice`. Wir erzeugen von ihnen Instanzen mit den Funktionen `addXXXXExercice` im Programmtext `drill1.py`. Auf die Instanzen wird in einer Liste `exerciceList` verwiesen, in denen die Schlüssel auch Argumente der Seiten jeder Übung sind, auf dem linken Baum:

```
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 ()
```

Die Funktion `addGameExercice` erzeugt ein Blatt im Baum mit dem Attribut `id="Games/Color"`, indem es die Funktion `addExercice` aufruft. Dieses Attribut wird als Schlüssel für die Instanz das Farbspiel benutzt, das vom Befehl `colorExercice()` in der Liste `exerciceList` erzeugt wird.

Als nächstes können wir dank der Eleganz des Polymorphismus bei objektorientierter Entwicklung die Übungen ausführen, indem wir dieselben Funktionen benutzen, die für jedes Objekt anders agieren, ohne sich um ihre interne Implementierung zu kümmern. Wir rufen nur Methoden auf, die in der abstrakten Basisklasse `exercice` definiert sind und sie machen unterschiedliche Dinge in den Klassen colorExercice oder labelExercice. Der Programmierer "spricht" mit allen Übungen auf die gleiche Weise, sogar, wenn die "Antwort" jeder Übung ein bißchen anders ist. Um das zu tun, kombinieren wir die Benutzung des Attributs `id` der Seiten des Baums und die Liste `exerciceList` oder die Variable `exoSelected` , die sich auf die benutzte Übung bezieht. Wenn alle diese Übungen Kinder der Klasse `exercice` sind, dann benutzen wir ihre Methoden auf die gleiche Weise, um die Übungen in all ihrer Unterschiedlichkeit zu kontrollieren.

```
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)
```

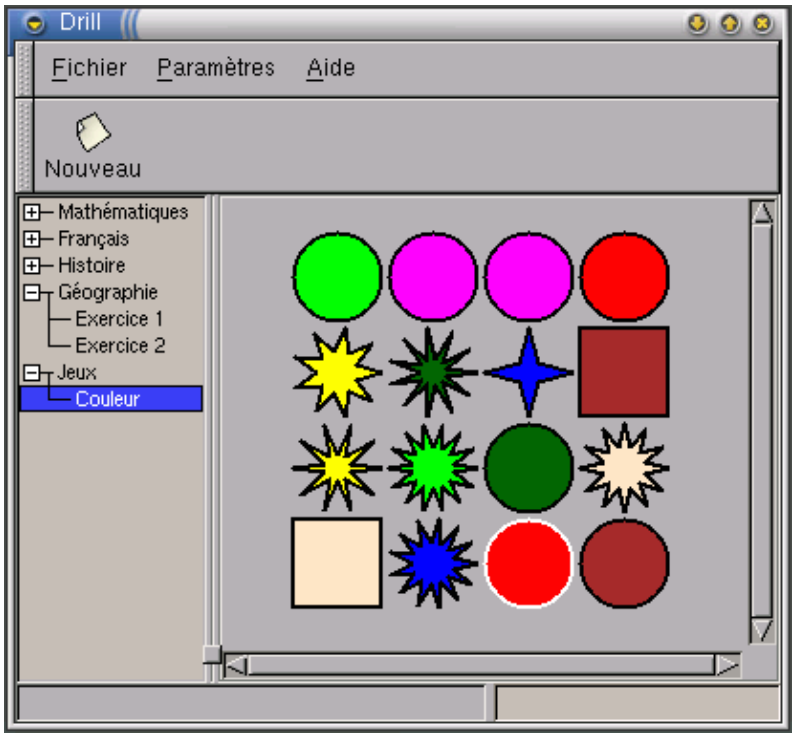

Bild 1 – Hauptfenster von Drill mit der Farbübung

Damit endet der Artikel. Wir haben die Vorzüge von objektorientierter Entwicklung mit Python im Bereich einer graphischen Schnittstelle entdeckt. In den folgenden Artikeln werden wir fortfahren, die Gnomewidgets zu entdecken, indem wir neue Übungen programmieren, die wir in ***Drill*** implementieren werden.

Anhang: Vollständiger Programmtext

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 *

# Import the exercice class
from colorExercice import *
from labelExercice import *

exerciceTree = currentExercice = None
# The exercice holder
exoArea = None
exoSelected = None
exerciceList = {}

def on_about_activate(obj):
    "display the about dialog"
    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

```
# Exercice pure virtual class
# exercice class methods should be override
# when exercice class is derived
class exercice:
    "A template exercice"
    exerciceWidget = None
    exerciceName = "No Name"
    def __init__ (self):
        "Create the exericice widget"
    def activate (self, area):
        "Set the exercice on the area container"
        area.add (self.exerciceWidget)
    def unactivate (self, area):
        "Remove the exercice fromt the container"
        area.remove (self.exerciceWidget)
    def reset (self):
        "Reset the exercice"
```

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):
    "A dummy exercie, it just prints a label in the exercice area"
    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
```

```
# Exercice 1 : color game

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):
        "build a shape of 'type' and 'color'"
        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: factor to get the internal radius"
        "n: number of branch"
        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):
        "get the n-th non null element of 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
```


2005-01-11, generated by lfparser_pdf version 2.51