LinuxFocus article number 266
http://linuxfocus.org

Développer des Applications Gnome avec Python (Partie 3)

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

L´auteur:

Hilaire Fernandes est le vice-président d'OFSET, une organisation pour promouvoir le développement de logiciels éducatifs libres pour le bureau Gnome. Il a aussi écrit Dr.Geo, un logiciel primé de géométrie dynamique, et il est actuellement occupé avec Dr.Genius un autre logiciel éducatif de mathématiques pour le bureau Gnome.

Résumé:

Cette série d'articles est spécialement écrite pour des débutants en programmation sous Gnome et GNU/Linux. Le langage de développement choisi, Python, évite la surcharge habituelle avec des langages compilés comme le C. Avant d'étudier cet article quelques notions de programmation sous Python sont nécessaires. Plus d'informations sur Python et Gnome sont disponbibles aux adresses http://www.python.org et http://www.gnome.org.

Articles précédents dans la série :
premier article
second article

Needed tools

Pour les besoins logiciels à l'exécution du programme décrit dans cet article, vous pouvez vous référer à la liste de la même rubrique de la partie I de cette série d'articles.

Aussi vous aurez besoin :

- Du fichier .glade original [drill.glade] . Ce fichier a été légèrement modifié depuis la dernière fois pour incorporer des ascenseurs dans la zone exercice de l'interface.
- Du code source en Python. Il est cette fois-ci éclaté en trois fichiers :
  1. [drill1.py].
  2. [templateExercice.py].
  3. [colorExercice.py].

4. [labelExercice.py].

Pour l'installation et l'utilisation de Python-Gnome et LibGlade vous pouvez aussi vous référer à la partie I.

Modèle de développement des exercices

Lors de la dernière partie nous avions mis en place l'interface utilisateur -- **Drill** -- devant servir de cadre pour le déploiement d'exercices. Cette fois-ci, nous allons nous intéresser de plus près au modèle de développement interne des exercices devant s'insérer dans **Drill**. Ce sera l'occasion d'explorer plus en détail les possibilités de développement orienté objet du langage Python. Pour l'essentiel, cette partie traitera donc plus de développement Python pur que de développement Gnome en Python.

La dernière fois, j'avais laissé un exercice pratique en suspens. À savoir la transformation du petit jeu de couleur, réalisé lors de la première partie de cette série d'articles, en un exercice à insérer dans **Drill**. Nous nous servirons de cet exemple pour illustrer notre exposé et, par la même occasion, nous donnerons une solution à cette exercice.

Le développement orienté objet

En quelques mots et sans prétention d'être exhaustif, le développement objet s'attache à caractériser et catégoriser -- en général -- par des relations du type **est-une-sorte-de** des objets du monde réel ou non. Cela peut être perçu comme une conceptualisation de ces objets par rapport à une problématique à laquelle nous nous intéressons. Nous pouvons les comparer dans d'autres domaines, aux catégories d'Aristote, aux taxinomies ou ontologies. Dans tout ces cas il s'agit bien d'appréhender, par une réduction conceptuelle, une situation complexe. Ce modèle de développement aurait aussi bien pu s'appeler développement orienté catégorie.

Dans ce modèle de développement, les objets manipulés par le programme, ou constituant le programme, sont appelés des **classes** et des représentants de ces objets conceptuels des **instances**. Les objets sont caractérisés par des **attributs** (des valeurs en général) et des **méthodes**. Les objets peuvent ne pas être totalement caractérisés, dans ce cas nous parlons de classes abstraites, c'est par exemple le cas lorsqu'une méthode est déclarée mais non définie (nous parlons de méthode virtuelle pure, le corps de la méthode est vide). Pour créer une instance d'une classe, celle-ci ne doit pas être abstraite. Les classes abstraites permettent de spécifier la forme prise par les classes héritières. Classes dans lesquelles les méthodes virtuelles seront définies. Les classes sont rangées entre elles par une relation du type **est-une-sorte-de**, dite relation d'héritage, nous parlons dans ce cas de classe(s) parente(s) d'une classe donnée.

Selon les langages, il existe une plus où moins grande finesse dans la caractérisation des objets. Cependant le plus petit dénominateur commun semble être celui-ci :

1. Héritage des attributs et des méthodes de la classe parente par la classe héritière.
2. Dans une classe héritière, possibilité de surcharger les méthodes héritées de la classe parente (i.e. redéfinir une méthode héritée).
3. Polymorphisme, une classe donnée peut avoir plusieurs classes parentes.

Python et le développement orienté objet

En ce qui concerne Python, c'est ce plus petit dénominateur commun qui a été choisi. Cela permet de s'initier au développement objet sans se perdre dans les détails de ce type de développement.

En Python, les méthodes d'un objet sont toujours virtuelles. Cela signifie qu'elles peuvent toujours être surchargées par une classe héritière -- ce que nous souhaitons faire en général en développement objet -- cela simplifie légèrement la syntaxe mais ne permet pas de distinguer rapidement ce qui est effectivement surchargé de ce qui ne l'est pas. Ensuite il n'est pas possible de rendre obscur un objet, c'est à dire rendre impossible l'accès à des attributs ou méthodes depuis l'extérieur de l'objet. Les attributs d'un objet Python sont accessibles aussi bien en lecture qu'en écriture depuis l'extérieur de l'objet.

La classe parente `exercice`

Dans notre exemple (voir le fichier `templateExercice.py` , nous souhaitons caractériser des objets de type exercice. Nous définissons donc naturellement un objet de type `exercice`. Cet objet sert de base conceptuel aux autres types d'exercices que nous créerons par la suite. L'objet `exemple` est la classe parente de tous les autres types d'exercices créés. Ces types d'exercices auront ainsi au minimum les mêmes attributs et méthodes que la classe `exercice`. Ce minimum commun nous permettra de manipuler identiquement toutes les instances d'exercices, même dans leur plus grande diversité, quelque soit l'objet dont ils sont une instance.

Par exemple, pour créer une instance de la classe `exercice` nous pourrions écrire :

```
from templateExercice import exercice

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

En fait il n'y a pas d'intérêt à créer des instance de la classe `exercice` car elle n'est qu'un modèle à partir duquel d'autre classes sont dérivées.

Les attributs

- `exerciceWidget` : le widget contenant l'interface utilisateur de l'exercice ;
- `exerciceName` : le nom de l'exercice.

Si nous devions nous intéresser à d'autres aspects d'un exercice nous pourrions lui ajouter des attributs.

Je pense par exemple au score sur un exercice, au nombre de fois qu'il a été fait, etc.

Les méthodes

- `__init__ (self)`: cette méthode a un rôle très précis dans un objet Python. Celle-ci est automatiquement appelée lors de la création d'une instance de cet objet. Pour cette raison elle est aussi nommée constructeur. L'argument `self` est une référence de l'instance de la classe `exercice` d'où est appelée la méthode `__init__`. Il est toujours nécessaire dans des méthodes de spécifier cet argument, cela veut donc dire qu'une méthode ne peut avoir zéro argument. Attention cet argument est alimenté automatiquement par Python, il n'est donc pas nécessaire de le placer lors de l'appel d'une méthode. L'argument `self` permet d'accéder aux attributs et autres méthodes d'une instance. Sans lui il est impossible d'y avoir accès. Nous verrons cela plus en détail par la suite.
- `activate (self, area)` : active l'exercice -- représenté par l'instance d'où est appelée cette méthode -- en plaçant son widget dans la zone exercice de **Drill**. L'argument `area` est en fait cet emplacement dans **Drill**, c'est un container GTK+. Sachant que l'attribut `exerciceWidget` contient le widget de l'exercice, il suffit de l'appel `area.add (self.exerciceWidget)` pour empaqueter l'exercice dans **Drill**.
- `unactivate (self, area)` : enlève le widget de l'exercice du container de **Drill**. Ici en terme de mise en paquet, c'est l'opération contraire, donc un `area.remove (self.exerciceWidget)` suffit.
- `reset (self)` : remet à zéro l'exercice.

En terme de code Python cela donne la chose suivante :

```
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"
```

Ce code est inclus dans son propre fichier `templateFichier.py`, cela nous permet de clarifier les rôles spécifiques de chaque objet. Les méthodes sont déclarées à l'intérieur de la classe `exercice`, ce sont en fait des fonctions.

À propos de l'argument `area`, nous verrons par la suite que c'est une référence d'un widget GTK+ construit par LibGlade, c'est une fenêtre avec ascenseurs.

Dans cet objet, les méthodes `__init__` et `reset` sont vides, elle seront surchargées par des classes héritières si nécessaire.

`labelExercice`, premier exemple d'héritage

Cet exercice est presque un exercice vide. Il ne fait qu'une chose : afficher le nom de l'exercice dans la zone exercice de **Drill**. Il nous sert de pis-aller pour les exercices qui peuplent l'arbre de gauche de **Drill** mais qui ne sont pas encore créés.

Comme pour l'objet `exercice`, l'objet `labelExercice` est placé dans son propre fichier, `labelExercice.py`. Ensuite, étant donné que cet objet est un héritier de l'objet `exercice`, nous avons besoin de lui indiquer les définitions de ce dernier. Cela ce fait simplement par une importation :

```
from templateExercice import exercice
```

Cela signifie littéralement que la définition de la classe `exercice` qui est dans le fichier `templateExercice.py` est importée dans le code courant.

Nous arrivons maintenant à l'aspect le plus important, la déclaration de la classe `labelExercice` en tant que classe héritière de `exercice`. Lors de la déclaration de `labelExercice`, cela se fait de la façon suivante :

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

Voilà, cela suffit pour que `labelExercice` hérite de tous les attributs et toutes les méthodes de `exercice`.

Bien sûr il nous reste du travail à faire, en particulier initialiser le widget de l'exercice. Nous le faisons en surchargeant la méthode `__init__` (i.e. en la redéfinissant dans la classe `labelExercice`), celle-ci est appelée lorsqu'une instance est créée. Aussi ce widget devra être référencé dans l'attribut `exerciceWidget`, de cette façon nous n'aurons pas besoin de surcharger les méthodes `activate` et `unactivate` de la classe `exercice`.

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

C'est la seule méthode que nous surchargeons. Pour créer une instance de `labelExercice` il suffit de faire l'appel :

```
monExercice = labelExercice ("Un exercice qui ne fait rien")
```

Pour accéder à ses attributs ou ses méthodes :

```
# Le nom de l'exercice
print monExercice.exerciceName
```

```
# Placer le widget de l'exercice dans le container "area"
monExerice.activate (area)
```

`colorExercice`, deuxième exemple d'héritage

Ici nous abordons la transformation du jeu de couleur, vu dans le premier article de cette série, en une classe de type `exercice`, plus précisément nous nommons cette classe `colorExercice`, il est placé dans son propre fichier `colorExercice.py` dont le code source complet est en annexe de cet article.

Par rapport au code source initial, il s'agit essentiellement d'une redistribution des fonctions et variables en méthodes et attributs dans la classe `colorExercice`.

Les variables globales sont transformées en attributs déclarés au début de la classe :

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

Comme pour la classe `labelExercice`, la méthode `__init__` est surchargée pour contenir la construction des widgets de l'exercice :

```
    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 ()
```

Rien de nouveau par rapport au code initial, si ce n'est que le `GnomeCanvas` est référencé dans l'attribut `exerciceWidget`.

L'autre méthode surchargée est `reset`, elle remet à zéro le jeu, elle doit donc être spécialisée au jeu de couleur :

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

Les autres méthodes sont la transcription directe des fonctions, avec en plus l'utilisation de la variable `self` pour accéder aux attributs et méthodes de l'instance. Il existe juste une exception dans les méthodes `buildStar` et `buildShape` où le paramètre décimal `k` a été remplacé par un paramètre entier.

J’ai noté un comportement étrange dans le document `colorExercice.py` où les nombres décimaux saisis dans le code source sont tronqués. Ce problème semble être lié au module `gnome.ui` et au locale français (où les nombres décimaux ont leur partie entière et leur partie décimale délimitées par une virgule et non un point). Je tâcherai de trouver la source du problème d’ici le prochain article.

Derniers ajustements dans Drill

Nous avons deux types d’exercice -- `labelExercice` et `colorExercice`. Nous en créons des instances depuis les fonctions `addXXXXExercice` dans le code `drill1.py`. Les instances sont référencées dans un dictionnaire `exerciceList` dont les clés sont également stockées comme arguments des feuilles de chaque exercice dans l’arbre de gauche :

```
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 fonction `addGameExercice` crée, par l’appel à la fonction `addExercice` une feuille dans l’arbre avec comme attribut `id="Games/Color"`, ce même attribut est utilisé comme clé de l’instance de l’exercice couleur -- créée par la commande `colorExercice()` -- dans le dictionnaire `exerciceList`.

Ensuite, et c’est là toute l’élégance du polymorphisme dans le développement orienté objet, nous pouvons manipuler, depuis les fonctions de traitement qui utilisent les différents objets exercices, les exercices quelque soit leur architecture interne. Seules les méthodes définies dans la classe virtuelle de base `exercice` sont utilisées, et elles font, par exemple, des choses différentes dans chaque classe colorExercice ou labelExercice. Le programmeur "parle" à tous les exercices de la même façon, même si ces exercices sont un peu différents. Pour ce faire nous combinons à la fois l’utilisation de l’attribut `id` des feuilles de l’arbre et le dictionnaire `exerciceList` ou la variable `exoSelected` qui référence l’exercice en cours d’utilisation. Étant donné que tous les exercices sont des héritiers de la classe `exercice`, nous utilisons ses méthodes comme autant de point de contrôle des exercices, dans toutes leurs variétés.

```
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)
```

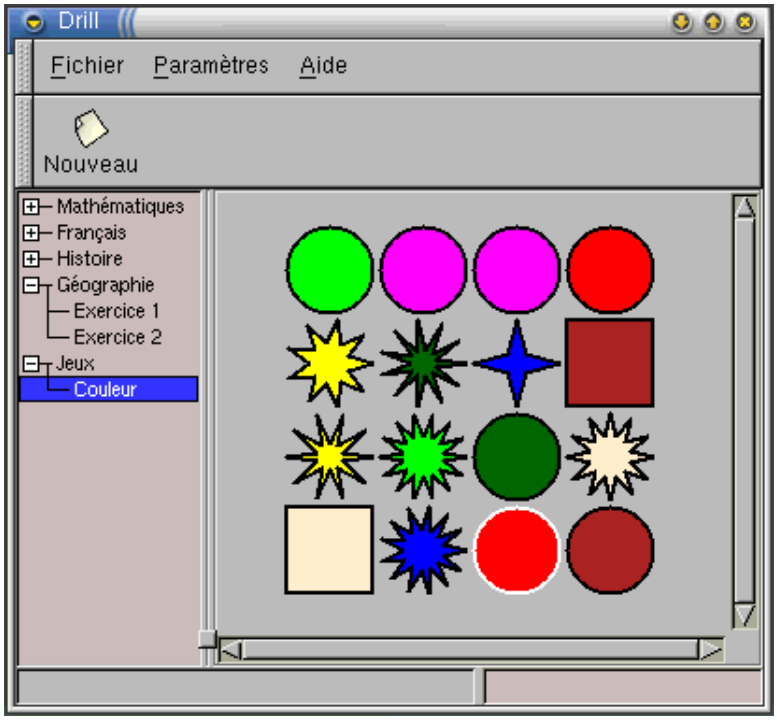

Fig. 1 - Fenêtre principale de Drill, avec l'exercice couleur

Cela clôt ici notre article. Nous avons donc découvert les attraits du développement orienté objet en Python dans le cadre d'application avec interface graphique. Dans les prochains articles nous continuerons la découverte des widgets Gnome à travers la réalisation de nouveaux exercices que nous insérerons dans **Drill**.

Appendice: Le source complet

drill1.py

```
#!/usr/bin/python
# Drill - 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 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
```

2005-01-14, generated by lfparser_pdf version 2.51