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

About the author:

O Hilaire Fernandes é o Vice–Presidente da OFSET, uma organização para promover o desenvolvimento de software educacional 'livre' para o ambiente de trabalho Gnome. Ele também escreveu o Dr. Geo, um programa pioneiro na geometria dinâmica e, está presentemente a trabalhar no Dr. Genius – outro programa educacional para o Gnome.

Desenvolvendo Aplicações para o Gnome com o Python (Parte 3)

Abstract:

Esta série de artigos é especialmente escrita para programadores novatos usando o Gnome e GNU/Linux. O Python, a linguagem escolhida para desenvolvimento, evita a habitual complicação das linguagens compiladas como o C. Para entender este artigo precisa de ter uma compreensão básica da programação em Python. Mais informação acerca do Python e do Gnome está disponível em http://www.python.org e http://www.gnome.org.

Artigos anteriores desta série :
– primeiro artigo
– segundo artigo

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

Utilitários necessários

Para a dependências de software necessárias para executar o programa descrito neste artigo, refira–se à lista referida na primeira parte da série destes artigos.

Também precisa de:

- O ficheiro original .glade [drill.glade] . Este ficheiro foi ligeiramente modificado desde a última vez para incorporar sliders para escolher os exercícios na interface.
- Desta vez o código fonte Python é distribuído em quatro ficheiros :
    1. [drill1.py].
    2. [templateExercice.py].

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

Para instalação e uso do Python−Gnome e LibGlade refira−se à Parte I.

Modele de desenvolvimento para os exercícios

No artigo anterior (parte 2), criámos a interface do utilizador −− ***Drill*** −− que é uma frame para o código dos exercícios descritos mais à frente. Agora damos uma vista de olhos ao desenvolvimento de objectos no Python, para adicionar funcionalidades ao ***Drill***. Neste estudo, deixamos de parte os aspectos do desenvolvimento do Python no Gnome.

Assim, comecemos onde deixámos as coisas, a inserção de um jogo de cores dentro do ***Drill*** como exercício para o leitor. Usaremos isto para ilustrar a nossa presente matéria e ao mesmo tempo para dar a solução ao exercício.

Desenvolvimento Orientado a Objectos

Brevemente, sem ter de fazer uma análise exaustiva, o desenvolvimento por objectos tenta definir e categorizar as coisas através de relações ***é um***, quer existam no mundo real ou não. Isto pode ser visto como abstraindo os objectos relacionados com o problema no qual estamos interessados. Podemos encontrar comparações em domínios diferentes, como as categorias de Aristóteles, taxonomias ou ontologias. Em cada caso, devemos entender uma situação complexa através da abstracção. Este tipo de desenvolvimento podia ser chamado desenvolvimento orientado às categorias.

Neste modelo de desenvolvimento, os objectos manipulado pelo programa, ou constituintes do programa são chamados ***classes*** e a representação destes objectos abstractos são as ***instâncias***. As classes são definidas por ***atributos*** (contendo valores) e ***métodos*** (funções). Falamos de uma relação pai−filho quando para uma dada classe uma classe filha herda propriedades do pai. As classes estão organizadas por uma relação ***é um***, onde o filho ***é ainda um*** tipo de pai bem como um tipo de filho. As classe podem não ser totalmente definidas, sendo neste caso chamadas de classes abstractas. Quando um método é declarado mas não definido (o corpo da função não é nada) é também chamado um método virtual. Uma classe abstracta tem um ou mais destes métodos indefinidos e devido a isto não pode ser instanciada. As classes abstractas permitem especificação na forma tomada pelas classes derivadas − classes filhas nas quais os métodos virtuais serão definidos.

As diferentes linguagens têm mais ou menos elegância em definir os objectos, mas o senso comum parece ser o seguinte:

1. Herança dos atributos e métodos da classe pai pelas classes filhas.
2. Possibilidade das classes filhas sobreporem os métodos herdados da classe pai.
3. Polimorfismo, onde uma classe pode ter várias classes pais.

Python e o Desenvolvimento Orientado a Objectos

No caso do Python foi escolhido o denominador menos comum. Este permite aprender o desenvolvimento orientado a objectos sem ter de entrar nos detalhes desta metodologia.

No Python todos os métodos dos objectos são sempre métodos virtuais. Isto quer dizer que podem sempre ser sobrepostos por uma classe filha −− o que geralmente é o que pretendemos quando usamos o desenvolvimento orientado por objectos −− e simplifica, ligeiramente, a sintaxe. Mas não é fácil distinguir entre métodos que podem e que não podem ser sobrepostos. Para além disto é impossível criar um objecto opaco por conseguinte negar o acesso a atributos e métodos do exterior do objecto. Em conclusão, os atributos de um objecto do Python podem ser lidos e escritos a partir do exterior do objecto.

Exercicio Classe Pai

No nosso exemplo, (veja o ficheiro `templateExercice.py`), gostaríamos de definir muitos objectos do tipo `exercice`. Definimos um objecto do tipo `exercice` para servir como uma classe base abstracta, para serem derivados outros exercícios que criaremos mais tarde. O objecto `exemple` é a classe pai de todos os outros tipo de exercícios criados. Pelo menos, os tipos de classes herdadas terão os mesmos atributos e métodos que a classe `exemple` visto que derivam dela. Isto permitir−nos−á manipular todos os tipos de objecto independentemente do objecto pelo qual foram instanciados.

Por exemplo, para criar uma instância da classe `exercice` podemos escrever:

```
from templateExercice import exercice

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

De facto, não existe necessidade de criar uma instância da classe `exercice` porque só é um modelo a partir do qual outras classes são derivadas.

Atributos

- `exerciceWidget` : a widget contendo a interface de utilizador do exercice;
- `exerciceName` : o nome do exercise.

Se estivermos interessados noutros aspectos do exercise podemos adicionar atributos, por exemplo, a pontuação obtida ou o número de vezes que correu.

Métodos

- `__init__ (self)`: este método tem uma regra muito precisa num objecto Python. É chamado automaticamente durante a criação da instância de um objecto. Por esta razão também é chamado construtor. O argumento `self` é uma referência para a instância da classe `exercise` que invocou o método `__init__ (self)`. É sempre necessário especificar este argumento nos métodos, ou seja um método não ficar sem argumentos. Cuidado, este argumento é adicionado automaticamente pelo Python, assim não é necessário incluí−lo sempre que se invoca um método. O argumento `self` permite acesso aos atributos e a outros métodos de uma instância. Sem ele, tal acesso é impossível.

Veremos isto depois mais detalhadamente.

- `activate (self, area)` : Activa esta instância do exercise ao pôr a sua widget na zona ***Drill*** do exercise. O argumento `area` é actualmente um contentor GTK+ que controla a posição da widget no ***Drill***. Ao saber este atributo o `exerciceWidget` contem a widget do exercício e só precisa de uma chamada à `area.add (self.exerciceWidget)` para desenhar o exercise no ***Drill***.
- `unactivate (self, area)` : remove a widget do contentor ***Drill***. Em termos de posicionamento esta é a operação inversa, assim basta chamar o `area.remove (self.exerciceWidget)`.
- `reset (self)` : reset do exercise para zero.

Em termos de código Python :

```
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"
```

Este código é incluído num ficheiro próprio `templateFichier.py`, o que nos permite clarificar as ideias específicas de cada objecto. Os métodos são declarados dentro da classe `exercice`, e no fundo são funções.

Veremos que o argumento `area` é uma referência para uma widget GTK+ construída pela LibGlade, é uma janela com sliders.

Neste Objecto, os métodos `__init__` e `reset` estão vazios e serão sobrepostos pelos das classes filhas se necessário.

labelExercice, Primeiro Exemplo de Herança

Isto é, praticamente um exercício vazio. Só faz uma coisa, põe o nome do exercício na zona ***Drill*** do exercise. Serve como princípio para os exercícios que preencheram a parte direita da árvore do ***Drill***, mas ainda não a criámos.

Do mesmo que o objecto `exercice`, o objecto `labelExercice` é posto num ficheiro próprio, `labelExercice.py` . De seguida, como este objecto é um filho do objecto `exercice`, precisamos de dizer como é que o pai é definido. Isto é feito através de um simples import:

```
from templateExercice import exercice
```

Isto literalmente, significa que a definição da classe `exercice` no ficheiro `templateExercice` é importada para o currente código.

Chegamos agora, ao aspecto mais importante, a declaração da classe `labelExercice` como classe filha da classe `exercice`.
A Classe`labelExercice` é declarada da seguinte forma :

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

Voilà, e é o suficiente para que a `labelExercice` herde os atributos e métodos da `exercice`.

Claro que ainda temos trabalho a fazer, em particular, precisamos de inicializar a widget do exercício. Fazemos isto sobrepondo o método `__init__` (isto é, redefinindo–o na classe `labelExercice`), este último é chamado quando uma instância é criada. Esta widget, também, deve ser referenciada no atributo `exerciceWidget` assim não precisaremos de sobrepor os métodos `activate` e `unactivate` da classe pai `exercice`.

```
  def __init__ (self, name):
      self.exerciceName = "Un exercice vide" (Trans. note: an empty exercise)
      self.exerciceWidget = GtkLabel (name)
      self.exerciceWidget.show ()
```

Este é o único método que sobrepomos. Para criar uma instância do `labelExercice`, só precisamos de chamar :

```
monExercice = labelExercice ("Um exercício que nada faz")
```

Para aceder aos seus atributos ou métodos :

```
# O nome do exercício
print monExercice.exerciceName

# Colocar o widget do exercice no contentor "area"
monExerice.activate (area)
```

colorExercice, Segundo Exemplo de Herança

Aqui começamos a transformação da cor do jogo, visto no primeiro artigo deste série, numa classe do tipo `exercice` que se chamará `colorExercice`. Colocá–la–emos no seu próprio ficheiro, `colorExercice.py` , que será concatenado a este artigo com o código fonte completo.

As alterações requeridas ao código fonte inicial consistem, basicamente, na redistribuição das funções e variáveis em métodos e atributos na classe `colorExercice`.

As variáveis globais são transformadas em atributos declarados no princípio da classe :

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

Como para a classe `labelExercice`, o método `__init__` é sobreposto para acomodar a construção das widgets do exercice :

```
def __init__ (self):
    self.exerciceName = "Le jeu de couleur" # Translator Note: the color game
    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 ()
```

Nada de novo comparativamente ao código inicial, somente o `GnomeCanvas` referenciado no atributo `exerciceWidget`.

O outro método sobreposto é o `reset`. Visto que faz um reset do jogo para zero, deve ser alterado para o jogo colorido :

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

Os outros métodos são cópias directas das funções originais, com a adição da variável `self` para permitir acesso aos atributos e métodos da instância. Existe uma excepção nos métodos `buildStar` e `buildShape` onde um parâmetro decimal `k` é substituído por um número inteiro. Eu notei um comportamento estranho no documento `colorExercice.py` onde os números decimais declarados no código são truncados. O problema parece estar no módulo `gnome.ui` e no French locale (onde os números decimais usam a vírgula como separador em vez do ponto). Trabalharei para encontrar o código do problema antes do próximo artigo.

Ajustamentos Finais no Drill

Definimos agora dois tipos de exercícios `labelExercice` e `colorExercice`. Criamos instâncias deles com as funções `addXXXXExercice` no código `drill1.py`. As instâncias são referenciadas num dicionário `exerciceList` no qual as chaves também são argumentos para as páginas de cada exercício na árvore da direita :

```
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 ()
```

A função `addGameExercice` cria uma folha na árvore com o atributo `id="Games/Color"` ao chamar a função `addExercice`. Este atributo é usado como chave para a instância do jogo colorido, criado pelo comando `colorExercice()` no dicionário `exerciceList`.

De seguida, devido à elegância do polimorfismo num desenvolvimento orientado a objectos, podemos correr os exercícios ao usar as mesmas funções que agem diferentemente para cada objecto, sem termos de nos preocupar com a sua implementação interna. Só chamamos métodos definidos na classe base abstracta `exercice` e elas fazem coisas diferentes quer na classe colorExercice ou quer na classe labelExercice. O programador "fala" para todos os exercícios do mesmo modo, mesmo que a "resposta" de cada exercício seja um pouco diferente. Para fazer isto combinamos o uso de cada atributo `id` das páginas da árvore com o dicionário `exerciceList` ou a variável `exoSelected` que se refere ao exercício em uso. Dado isto todos os exercícios são filhos da classe `exercice`, usamos os seus da mesma maneira para controlar os exercícios

em toda a sua variedade.

```
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)
```

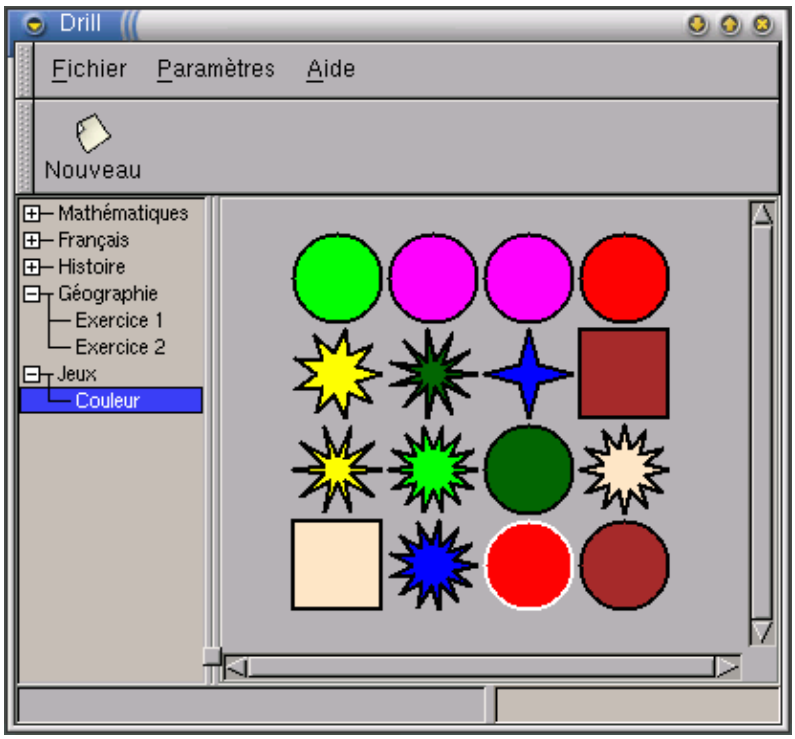

Fig. 1 – Janela Principal do Drill, com o exercício colorido

Assim termina o nosso artigo. Descobrimos as atracções do desenvolvimento orientado a objectos no Python ainda com o aditivo de uma interface de utilizador gráfica. No próximo artigo continuaremos a descobrir as widgets do Gnome através da codificação de novos exercícios que vamos inserir no ***Drill***.

Apendice: Código Fonte 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 *

# 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
```


2005-01-10, generated by lfparser_pdf version 2.51