PyQGIS developer cookbook

Release 2.6

QGIS Project

May 22, 2015

1 Introducere **1**
1.1 Consola Python . . . 1
1.2 Plugin-uri Python . . . 2
1.3 Aplicaii . . . 2

2 Încărcarea Straturilor **5**
2.1 Straturi Vectoriale . . . 5
2.2 Straturi raster . . . 6
2.3 Registrul straturilor de hartă . . . 7

3 Utilizarea straturilor raster **9**
3.1 Detaliile stratului . . . 9
3.2 Stilul desenării . . . 9
3.3 Recitirea straturilor . . . 11
3.4 Interogarea valorilor . . . 11

4 Utilizarea straturilor vectoriale **13**
4.1 Iteraii în straturile vectoriale . . . 13
4.2 Modificarea straturilor vectoriale . . . 14
4.3 Modificarea straturi vectoriale prin editarea unui tampon de memorie . . . 15
4.4 Crearea unui index spaial . . . 16
4.5 Scrierea straturilor vectoriale . . . 17
4.6 Furnizorul de memorie . . . 18
4.7 Aspectul (simbologia) straturilor vectoriale . . . 19
4.8 Lecturi suplimentare . . . 25

5 Manipularea geometriei **27**
5.1 Construirea geometriei . . . 27
5.2 Accesarea geometriei . . . 27
5.3 Predicate i operaiuni geometrice . . . 28

6 Proiecii suportate **31**
6.1 Sisteme de coordonate de referină . . . 31
6.2 Proiecii . . . 32

7 Folosirea suportului de hartă **33**
7.1 Încapsularea suportului de hartă . . . 33
7.2 Folosirea instrumentelor în suportul de hartă . . . 34
7.3 Benzile elastice i marcajele nodurilor . . . 35
7.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă . . . 36
7.5 Dezvoltarea elementelor personalizate pentru suportul de hartă . . . 37

8 Randarea hărilor i imprimarea **39**

8.1 Randarea simplă 39
8.2 Generarea folosind Compozitorul de hări 40

9 Expresii, filtrarea i calculul valorilor **43**
9.1 Parsarea expresiilor 44
9.2 Evaluarea expresiilor 44
9.3 Exemple 44

10 Citirea i stocarea setărilor **47**

11 Comunicarea cu utilizatorul **49**
11.1 Afiarea mesajelor. Clasa `QgsMessageBar` 49
11.2 Afiarea progresului 50
11.3 Jurnalizare 51

12 Dezvoltarea plugin-urilor Python **53**
12.1 Scrierea unui plugin 53
12.2 Coninutul Plugin-ului 54
12.3 Documentaie 58

13 Setările IDE pentru scrierea i depanarea de plugin-uri **59**
13.1 O notă privind configurarea IDE-ului în Windows 59
13.2 Depanare cu ajutorul Eclipse i PyDev 60
13.3 Depanarea cu ajutorul PDB 64

14 Utilizarea straturilor plugin-ului **65**
14.1 Subclasarea QgsPluginLayer 65

15 Compatibilitatea cu versiunile QGIS anterioare **67**
15.1 Meniul plugin-ului 67

16 Lansarea plugin-ului dvs. **69**
16.1 Depozitul oficial al plugin-urilor python 69

17 Fragmente de cod **71**
17.1 Cum să apelăm o metodă printr-o combinaie rapidă de taste 71
17.2 Inversarea Stării Straturilor 71
17.3 Cum să accesai tabelul de atribute al entităilor selectate 71

18 Biblioteca de analiză a reelelor **73**
18.1 Informaii generale 73
18.2 Construirea unui graf 73
18.3 Analiza grafului 75

Index **81**

CAPITOLUL 1

Introducere

Acest document are rolul de tutorial dar i de ghid de referină. Chiar dacă nu prezintă toate cazurile de utilizare posibile, ar trebui să ofere o bună imagine de ansamblu a funcionalitäii principale.

Începând cu versiunea 0.9, QGIS are suport de scriptare opional, cu ajutorul limbajului Python. Ne-am decis pentru Python deoarece este unul dintre limbajele preferate în scriptare. PyQGIS depinde de SIP i PyQt4. S-a preferat utilizarea SIP în loc de SWIG deoarece întregul cod QGIS depinde de bibliotecile Qt. Legarea Python cu Qt (PyQt) se face, de asemenea, cu ajutorul SIP, acest lucru permiând integrarea perfectă a PyQGIS cu PyQt.

DE EFECTUAT: Noiuni de bază PyQGIS (Compilare manuală, Depanare)

Există mai multe modalităi de a crea legături între QGIS i Python, acestea fiind detaliate în următoarele seciuni:

- scrierea comenzilor în consola Python din QGIS
- crearea în Python a plugin-urilor
- crearea aplicaiilor personalizate bazate pe QGIS API

Există o referină API QGIS completă care documentează clasele din bibliotecile QGIS. API-ul QGIS pentru python este aproape identic cu cel pentru C++.

Există unele resurse despre programarea în PyQGIS pe blog-ul QGIS. Parcurgei tutorialul QGIS portat în Python pentru câteva exemple de aplicaii simple produse de t3re păr i. O metodă bună de învăare, atunci când lucrai cu plugin-uri, este de a descărca câteva din depozitul de plugin-uri i să le examinai codul. De asemenea, folderul `python/plugins/` din instalarea QGIS conine unele plugin-uri cu ajutorul cărora putei învăa modul de dezvoltare al unor plugin-uri similare, care să efectueze unele dintre cele mai comune sarcini

1.1 Consola Python

Pentru scripting, se poate utiliza consola Python integrată. Aceasta poate fi deschisă din meniul: *Plugins* → *Consola Python*. Consola se deschide ca o fereastră de utilităi non-modală:

Captura de ecran de mai sus ilustrează cum să obinei accesul la stratul curent selectat în lista straturilor, să-i afiai ID-ul i, opional, în cazul în care acesta este un strat vectorial, să calculai numărul total de entităi spaiale. Pentru interaciunea cu mediul QGIS, există o variabilă de :date: *iface*, care este o instană a :clasei: *QgsInterface*. Această interfaă permite accesul la suprafaa hării, meniuri, barele de instrumente i la alte păr i ale aplicaiei QGIS.

Pentru confortul utilizatorului, următoarele instruciuni sunt executate atunci când consola este pornită (în viitor, va fi posibil să stabilii comenzi iniiale suplimentare)

```
from qgis.core import *
import qgis.utils
```

Pentru cei care folosesc des consola, ar putea fi util să stabilească o comandă rapidă pentru declanarea consolei (în meniul :menuselection: *Settings* -> *Configurare comenzi rapide* ...)

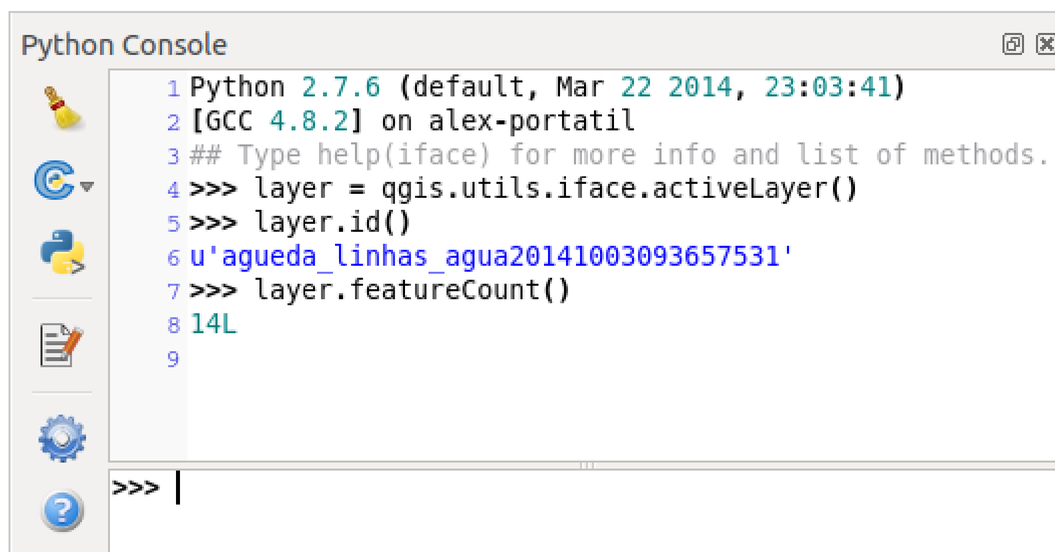

Figura 1.1: Consola Python din QGIS

1.2 Plugin-uri Python

QGIS permite îmbunătăirea funcionalităii sale, folosind plugin-uri. Acest lucru a fost iniial posibil numai cu ajutorul limbajului C. Odată cu adăugarea în QGIS a suportului pentru Python, este posibilă i folosirea de plugin-uri scrise în Python. Principalul avantaj faă de plugin-urile în C este simplitatea distribuiei (nu este necesară compilarea pentru fiecare platformă) iar dezvoltarea este mai uoară.

Multe plugin-uri care acoperă diverse funcionalităi au fost scrise de la introducerea suportului pentru Python. Instalatorul de plugin-uri permite utilizatorilor aducerea cu uurină, actualizarea i eliminarea plugin-uri Python. A se vedea pagina Depozitele de Plugin-uri Python pentru diferitele surse de plugin-uri.

Crearea de plugin-uri în Python este simplă, a se vedea :ref: *developing_plugins* pentru instruciuni detaliate.

1.3 Aplicaii

Adesea, atunci când are loc procesarea unor date GIS, este recomandabilă crearea unor script-uri pentru automatizarea procesului în loc de a face iarăi i iarăi aceeai sarcină. Cu PyQGIS, acest lucru este perfect posibil — importai :modul: *qgis.core*, îl iniializai i suntei gata pentru prelucrare.

Sau poate dorii să creai o aplicaie interactivă care utilizează unele funcionalităi GIS — măsurarea anumitor date, exportarea unei hări în format PDF sau orice alte funcionalităi. Modulul `qgis.gui` aduce în plus diverse componente GUI, mai ales widget-ul ariei hării, care poate fi foarte uor încorporat în aplicaiile cu suport pentru zoom, panning i/sau orice alte instrumente personalizabile.

1.3.1 Utilizarea PyQGIS în aplicaii personalizate

Notă: *nu* utilizai `qgis.py` ca nume pentru script-ul de test — datorită acestui nume, Python nu va fi capabil să importe legăturile.

Mai întâi de toate, trebuie să importai modul qgis, să setai calea către resurse — baza de date a proieciilor, furnizorii etc. Când setai prefixul căii având ca al doilea argument :const: *True*, QGIS va iniializa toate căile cu standardul dir în directorul prefixului. La apelarea funciei `initQgis()` este important să permitei aplicaiei QGIS să caute furnizorii disponibili.

```
from qgis.core import *
```

```
# supply path to where is your qgis installed
QgsApplication.setPrefixPath("/path/to/qgis/installation", True)

# load providers
QgsApplication.initQgis()
```

Acum putei lucra cu API QGIS — încărcai straturile i facei unele prelucrări sau startai un GUI cu o zonă pentru o hartă. Posibilităile sunt nelimitate :-)

Când ai terminat cu utilizarea bibliotecii QGIS, apelai funcia `exitQgis()` pentru a vă asigura că totul este curat (de exemplu, curăai registrul stratului hării i tergei straturile):

```
QgsApplication.exitQgis()
```

1.3.2 Rularea aplicaiilor personalizate

Trebuie să indicai sistemului dvs. unde să căute bibliotecile QGIS i modulele Python corespunzătoare, dacă acestea nu sunt într-o locaie standard — altfel Python va semnaliza:

```
>>> import qgis.core
ImportError: No module named qgis.core
```

Aceasta se poate repara prin setarea variabilei de mediu `PYTHONPATH`. În următoarele comenzi, `qgispath` ar trebui să fie înlocuit de către calea de instalare actuală a QGIS:

- în Linux: **export PYTHONPATH=/qgispath/share/qgis/python**
- în Windows: **set PYTHONPATH=c:\qgispath\python**

Calea către modulele PyQGIS este acum cunoscută, totui, acestea depind de bibliotecile `qgis_core` i `qgis_gui` (modulele Python servesc numai pentru ambalare). Calea către aceste biblioteci este de obicei necunoscută sistemului de operare, astfel încât vei obine iarăi o eroare de import (mesajul poate varia în funcie de sistem):

```
>>> import qgis.core
ImportError: libqgis_core.so.1.5.0: cannot open shared object file: No such file or directory
```

Remediai acest lucru prin adăugarea directoarelor în care rezidă bibliotecile QGIS la calea de căutare a linker-ului dinamic:

- în Linux: **export LD_LIBRARY_PATH=/qgispath/lib**
- în Windows: **set PATH=C:\qgispath;%PATH%**

Aceste comenzi pot fi puse într-un script bootstrap, care va avea grijă de pornire. Când livrai aplicaii personalizate folosind PyQGIS, există, de obicei, două posibilităi:

- să cerei utilizatorului să instaleze QGIS pe platforma sa înainte de a instala aplicaia. Instalatorul aplicaiei ar trebui să caute locaiile implicite ale bibliotecilor QGIS i să permită utilizatorului setarea căii, în cazul în care nu poate fi găsită. Această abordare are avantajul de a fi mai simplă, cu toate acestea este nevoie ca utilizatorul să parcurgă mai multe etape.
- să împacheteze QGIS împreună cu aplicaia dumneavoastră. Livrarea aplicaiei poate fi mai dificilă, iar pachetul va fi mai mare, dar utilizatorul va fi salvat de povara de a descărca i instala software suplimentar.

Cele două modele pot fi combinate - puei distribuii aplicaii independente pe Windows i Mac OS X, lăsând la îndemâna utilizatorului i a managerul de pachete instalarea QGIS pe Linux.

Încărcarea Straturilor

Haidei să deschidem mai multe straturi cu date. QGIS recunoate straturile vectoriale i de tip raster. În plus, sunt disponibile i tipuri de straturi personalizate, dar nu le vom discuta aici.

2.1 Straturi Vectoriale

Pentru a încărca un strat vectorial, specificai identificatorul sursei de date a stratului, numele stratului i numele furnizorului:

```
layer = QgsVectorLayer(data_source, layer_name, provider_name)
if not layer.isValid():
  print "Layer failed to load!"
```

Identificatorul sursei de date reprezintă un ir specific pentru fiecare furnizor de date vectoriale, în parte. Numele stratului se va afia în lista straturilor. Este important să se verifice dacă stratul a fost încărcat cu succes. În cazul neîncărcării cu succes, va fi returnată o instană de strat invalid.

Lista de mai jos arată modul de accesare a diverselor surse de date, cu ajutorul furnizorilor de date vectoriale:

- OGR library (shapefiles and many other file formats) — data source is the path to the file

  ```
  vlayer = QgsVectorLayer("/path/to/shapefile/file.shp", "layer_name_you_like", "cgr")
  ```

- PostGIS database — data source is a string with all information needed to create a connection to PostgreSQL database. `QgsDataSourceURI` class can generate this string for you. Note that QGIS has to be compiled with Postgres support, otherwise this provider isn't available.

  ```
  uri = QgsDataSourceURI()
  # set host name, port, database name, username and password
  uri.setConnection("localhost", "5432", "dbname", "johny", "xxx")
  # set database schema, table name, geometry column and optionally
  # subset (WHERE clause)
  uri.setDataSource("public", "roads", "the_geom", "cityid = 2643")

  vlayer = QgsVectorLayer(uri.uri(), "layer_name_you_like", "postgres")
  ```

- CSV or other delimited text files — to open a file with a semicolon as a delimiter, with field "x" for x-coordinate and field "y" with y-coordinate you would use something like this

  ```
  uri = "/some/path/file.csv?delimiter=%s&xField=%s&yField=%s" % (";", "x", "y")
  vlayer = QgsVectorLayer(uri, "layer_name_you_like", "delimitedtext")
  ```

  Note: from QGIS version 1.7 the provider string is structured as a URL, so the path must be prefixed with *file://*. Also it allows WKT (well known text) formatted geometries as an alternative to "x" and "y" fields, and allows the coordinate reference system to be specified. For example

```
uri = "file:///some/path/file.csv?delimiter=%s&crs=epsg:4723&wktField=%s" % (";", "shape")
```

- GPX files — the "gpx" data provider reads tracks, routes and waypoints from gpx files. To open a file, the type (track/route/waypoint) needs to be specified as part of the url

```
uri = "path/to/gpx/file.gpx?type=track"
vlayer = QgsVectorLayer(uri, "layer_name_you_like", "gpx")
```

- SpatiaLite database — supported from QGIS v1.1. Similarly to PostGIS databases, `QgsDataSourceURI` can be used for generation of data source identifier

```
uri = QgsDataSourceURI()
uri.setDatabase('/home/martin/test-2.3.sqlite')
schema = ''
table = 'Towns'
geom_column = 'Geometry'
uri.setDataSource(schema, table, geom_column)

display_name = 'Towns'
vlayer = QgsVectorLayer(uri.uri(), display_name, 'spatialite')
```

- MySQL WKB-based geometries, through OGR — data source is the connection string to the table

```
uri = "MySQL:dbname,host=localhost,port=3306,user=root,password=xxx|\
  layername=my_table"
vlayer = QgsVectorLayer( uri, "my_table", "ogr" )
```

- WFS connection:. the connection is defined with a URI and using the `WFS` provider

```
uri = "http://localhost:8080/geoserver/wfs?srsname=EPSG:23030&typename=union&version=1.0.0&re
vlayer = QgsVectorLayer("my_wfs_layer", "WFS")
```

URI poate fi creat folosind biblioteca standard `urllib`.

```
params = {
    'service': 'WFS',
    'version': '1.0.0',
    'request': 'GetFeature',
    'typename': 'union',
    'srsname': "EPSG:23030"
}
uri = 'http://localhost:8080/geoserver/wfs?' + urllib.unquote(urllib.urlencode(params))
```

2.2 Straturi raster

For accessing raster files, GDAL library is used. It supports a wide range of file formats. In case you have troubles with opening some files, check whether your GDAL has support for the particular format (not all formats are available by default). To load a raster from a file, specify its file name and base name

```
fileName = "/path/to/raster/file.tif"
fileInfo = QFileInfo(fileName)
baseName = fileInfo.baseName()
rlayer = QgsRasterLayer(fileName, baseName)
if not rlayer.isValid():
  print "Layer failed to load!"
```

Straturile raster pot fi, de asemenea, create dintr-un serviciu WCS.

```
layer_name = 'elevation'
uri = QgsDataSourceURI()
uri.setParam ('url', 'http://localhost:8080/geoserver/wcs')
```

```
uri.setParam ( "identifier", layer_name)
rlayer = QgsRasterLayer(uri, 'my_wcs_layer', 'wcs')
```

Alternatively you can load a raster layer from WMS server. However currently it's not possible to access GetCapabilities response from API — you have to know what layers you want

```
urlWithParams = 'url=http://wms.jpl.nasa.gov/wms.cgi&layers=global_mosaic&styles=pseudo&format=ima
rlayer = QgsRasterLayer(urlWithParams, 'some layer name', 'wms')
if not rlayer.isValid():
  print "Layer failed to load!"
```

2.3 Registrul straturilor de hartă

Dacă dorii să utilizai straturile deschise pentru randare, nu uitai să le adăugai la registrul straturilor de hartă. Acest registru înregistrează proprietatea asupra straturilor, acestea putând fi accesate ulterior din oricare parte a aplicaiei după ID-ul lor unic. Atunci când un strat este eliminat din registru, va fi i ters totodată.

Adding a layer to the registry:

```
QgsMapLayerRegistry.instance().addMapLayer(layer)
```

Layers are destroyed automatically on exit, however if you want to delete the layer explicitly, use:

```
QgsMapLayerRegistry.instance().removeMapLayer(layer_id)
```

DE EFECTUAT: Mai multe despre registrul straturilor de hartă?

CAPITOLUL 3

Utilizarea straturilor raster

Această seciune enumeră diverse operaiuni pe care le putei efectua cu straturile raster.

3.1 Detaliile stratului

A raster layer consists of one or more raster bands - it is referred to as either single band or multi band raster. One band represents a matrix of values. Usual color image (e.g. aerial photo) is a raster consisting of red, blue and green band. Single band layers typically represent either continuous variables (e.g. elevation) or discrete variables (e.g. land use). In some cases, a raster layer comes with a palette and raster values refer to colors stored in the palette.

```
>>> rlayer.width(), rlayer.height()
(812, 301)
>>> rlayer.extent()
u'12.095833,48.552777 : 18.863888,51.056944'
>>> rlayer.rasterType()
2  # 0 = GrayOrUndefined (single band), 1 = Palette (single band), 2 = Multiband
>>> rlayer.bandCount()
3
>>> rlayer.metadata()
u'<p class="glossy">Driver:</p>...'
>>> rlayer.hasPyramids()
False
```

3.2 Stilul desenării

Când un strat raster este încărcat, în funcie de tipul său, va moteni un stil de desenare implicit. Acesta poate fi modificat, fie prin modificarea manuală a proprietăilor rasterului, fie programatic. Există următoarele stiluri de desenare:

Index	Constant: QgsRasterLater.X	Comentariu
1	SingleBandGray	Imagine cu o singură bandă, afiată într-o gamă de nuane de gri
2	SingleBandPseudoColor	Imagine cu bandă unică, desenată de un algoritm cu pseudoculori
3	PalettedColor	"Palette" image drawn using color table
4	PalettedSingleBandGray	"Palette" layer drawn in gray scale
5	PalettedSingleBandPseudoColor	"Paleta" stratului desenată de un algoritm cu o pseudoculoare
7	MultiBandSingleBandGray	Strat care conine 2 sau mai multe benzi, din care o singură bandă desenată într-o gamă cu tonuri de gri
8	MultiBandSingleBandPseudoColor	Strat care conine 2 sau mai multe benzi, din care o singură bandă este desenată folosind un algoritm cu pseudoculori
9	MultiBandColor	Strat coninând 2 sau mai multe benzi, mapate în spaiul de culori RGB.

To query the current drawing style:

```
>>> rlayer.drawingStyle()
9
```

Straturile cu o singură bandă raster pot fi desenate fie în nuane de gri (valori mici = negru, valori ridicate = alb), sau cu un algoritm cu pseudoculori, care atribuie culori valorilor din banda singulară. Rasterele cu o singură bandă pot fi desenate folosindu-se propria paletă. Straturile multibandă sunt, de obicei, desenate prin maparea benzilor la culori RGB. Altă posibilitate este de a utiliza doar o singură bandă pentru desenarea în tonuri de gri sau cu pseudoculori.

Următoarele seciuni explică modul în care se poate interoga i modifica stilul de desenare al stratului. După efectuarea schimbărilor, ai putea fora actualizarea suprafeei hării, a se vedea *Recitirea straturilor*.

DE EFECTUAT: îmbunătăiri de contrast, de transparenă (date nule), min/max definit de utilizator, statistici bandă

3.2.1 Rastere cu o singură bandă

They are rendered in gray colors by default. To change the drawing style to pseudocolor:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandPseudoColor)
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
```

The `PseudoColorShader` is a basic shader that highlights low values in blue and high values in red. Another, `FreakOutShader` uses more fancy colors and according to the documentation, it will frighten your granny and make your dogs howl.

There is also `ColorRampShader` which maps the colors as specified by its color map. It has three modes of interpolation of values:

- liniar (`INTERPOLAT`): culoarea rezultată fiind interpolată liniar, de la intrările hării de culori, în sus sau în jos faă de valoarea înscrisă în harta de culori
- discret (`DISCRET`): culorile folosite fiind cele cu o valoare egală sau mai mare faă de cele din harta de culori
- exact (`EXACT`): culoarea nu este interpolată, desenându-se doar pixelii cu o valoare egală cu cea introdusă în harta de culori

Pentru a seta o gamă de culori pentru umbrire interpolate, variind de la verde la galben (pentru valori ale pixelilor între 0-255):

```
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.ColorRampShader)
>>> lst = [ QgsColorRampShader.ColorRampItem(0, QColor(0,255,0)), \
    QgsColorRampShader.ColorRampItem(255, QColor(255,255,0)) ]
>>> fcn = rlayer.rasterShader().rasterShaderFunction()
```

```
>>> fcn.setColorRampType(QgsColorRampShader.INTERPOLATED)
>>> fcn.setColorRampItemList(lst)
```

To return back to default gray levels, use:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.SingleBandGray)
```

3.2.2 Rastere multibandă

By default, QGIS maps the first three bands to red, green and blue values to create a color image (this is the `MultiBandColor` drawing style. In some cases you might want to override these setting. The following code interchanges red band (1) and green band (2):

```
>>> rlayer.setGreenBandName(rlayer.bandName(1))
>>> rlayer.setRedBandName(rlayer.bandName(2))
```

In case only one band is necessary for visualization of the raster, single band drawing can be chosen — either gray levels or pseudocolor, see previous section:

```
>>> rlayer.setDrawingStyle(QgsRasterLayer.MultiBandSingleBandPseudoColor)
>>> rlayer.setGrayBandName(rlayer.bandName(1))
>>> rlayer.setColorShadingAlgorithm(QgsRasterLayer.PseudoColorShader)
>>> # now set the shader
```

3.3 Recitirea straturilor

If you do change layer symbology and would like ensure that the changes are immediately visible to the user, call these methods:

```
if hasattr(layer, "setCacheImage"):
  layer.setCacheImage(None)
layer.triggerRepaint()
```

Primul apel garantează că imaginea din cache a stratului este tearsă în cazul în care cache-ul este activat. Această funcionalitate este disponibilă începând de la QGIS 1.4, în versiunile anterioare această funcie neexistând — pentru a fi siguri de cod, că funcionează cu toate versiunile de QGIS, vom verifica în primul rând dacă metoda există.

Al doilea apel emite semnalul care va fora orice suport de hartă, care conine stratul, să emită o reîmprospătare.

With WMS raster layers, these commands do not work. In this case, you have to do it explicitly:

```
layer.dataProvider().reloadData()
layer.triggerRepaint()
```

In case you have changed layer symbology (see sections about raster and vector layers on how to do that), you might want to force QGIS to update the layer symbology in the layer list (legend) widget. This can be done as follows (`iface` is an instance of QgisInterface):

```
iface.legendInterface().refreshLayerSymbology(layer)
```

3.4 Interogarea valorilor

To do a query on value of bands of raster layer at some specified point:

```
ident = rlayer.dataProvider().identify(QgsPoint(15.30,40.98), \
  QgsRaster.IdentifyFormatValue)
if ident.isValid():
  print ident.results()
```

În acest caz, metoda `results` returnează un dicionar, cu indicii benzii ca i chei, i valorile benzii ca valori.

```
{1: 17, 2: 220}
```

CAPITOLUL 4

Utilizarea straturilor vectoriale

Această secţiune rezumă diferitele acţiuni care pot fi efectuate asupra straturilor vectoriale.

4.1 Iteraţii în straturile vectoriale

Parcurgerea elementelor dintr-un strat vectorial este una dintre cele mai obişnuite activităţi. Mai jos este prezentat un exemplu de cod de bază, simplu, pentru a efectua această sarcină şi care arată unele informaţii despre fiecare entitate spaţială. Variabila `layer` se consideră a conţine un obiect `QgsVectorLayer`

```
iter = layer.getFeatures()
for feature in iter:
    # retrieve every feature with its geometry and attributes
    # fetch geometry
    geom = feature.geometry()
    print "Feature ID %d: " % feature.id()

    # show some information about the feature
    if geom.type() == QGis.Point:
      x = geom.asPoint()
      print "Point: " + str(x)
    elif geom.type() == QGis.Line:
      x = geom.asPolyline()
      print "Line: %d points" % len(x)
    elif geom.type() == QGis.Polygon:
      x = geom.asPolygon()
      numPts = 0
      for ring in x:
        numPts += len(ring)
      print "Polygon: %d rings with %d points" % (len(x), numPts)
    else:
      print "Unknown"

    # fetch attributes
    attrs = feature.attributes()

    # attrs is a list. It contains all the attribute values of this feature
    print attrs
```

Attributes can be referred by index.

```
idx = layer.fieldNameIndex('name')
print feature.attributes()[idx]
```

4.1.1 Parcurgerea entităilor selectate

Convenience methods.

For the above cases, and in case you need to consider selection in a vector layer in case it exist, you can use the `features()` method from the built-in Processing plugin, as follows:

```
import processing
features = processing.features(layer)
for feature in features:
    # do whatever you need with the feature
```

This will iterate over all the features in the layer, in case there is no selection, or over the selected features otherwise.

if you only need selected features, you can use the :func: *selectedFeatures* method from vector layer:

```
selection = layer.selectedFeatures()
print len(selection)
for feature in selection:
    # do whatever you need with the feature
```

4.1.2 Parcurgerea unui subset de entităi

Dacă dorii să parcurgei un anumit subset de entităi dintr-un strat, cum ar fi cele dintr-o anumită zonă, trebuie să adăugai un obiect *QgsFeatureRequest* la apelul funciei *getFeatures()*. Iată un exemplu

```
request=QgsFeatureRequest()
request.setFilterRect(areaOfInterest)
for f in layer.getFeatures(request):
    ...
```

The request can be used to define the data retrieved for each feature, so the iterator returns all features, but return partial data for each of them.

```
# Only return selected fields
request.setSubsetOfAttributes([0,2])
# More user friendly version
request.setSubsetOfAttributes(['name','id'],layer.pendingFields())
# Don't return geometry objects
request.setFlags(QgsFeatureRequest.NoGeometry)
```

4.2 Modificarea straturilor vectoriale

Cei mai muli dintre furnizorii de date vectoriale suportă editarea datelor stratului. Uneori, acetia acceptă doar un subset restrâns de aciuni de editare. Utilizai funcia `capabilities()` pentru a afla care set de funcii este disponibil

```
caps = layer.dataProvider().capabilities()
```

By using any of following methods for vector layer editing, the changes are directly committed to the underlying data store (a file, database etc). In case you would like to do only temporary changes, skip to the next section that explains how to do *modifications with editing buffer*.

4.2.1 Adăugarea entităilor

Creai câteva instane ale clasei `QgsFeature` i transmitei o listă a acestora metodei furnizorului `addFeatures()`. Acesta va returna două valori: rezultatul (true/false) i lista entităilor adăugate (ID-ul lor fiind stabilit de către depozitul de date)

```
if caps & QgsVectorDataProvider.AddFeatures:
  feat = QgsFeature()
  feat.addAttribute(0, 'hello')
  feat.setGeometry(QgsGeometry.fromPoint(QgsPoint(123, 456)))
  (res, outFeats) = layer.dataProvider().addFeatures([feat])
```

4.2.2 tergerea entităilor

Pentru a terge unele entităi, e suficientă furnizarea unei liste cu ID-uri

```
if caps & QgsVectorDataProvider.DeleteFeatures:
  res = layer.dataProvider().deleteFeatures([5, 10])
```

4.2.3 Modificarea entităilor

Este posibilă, fie schimbarea geometriei unei entităi, fie schimbarea unor atribute. În următorul exemplu are loc mai întâi schimbarea valorilor atributelor cu indexul 0 sau 1, iar mai apoi se schimbă geometria entităii

```
fid = 100   # ID of the feature we will modify

if caps & QgsVectorDataProvider.ChangeAttributeValues:
  attrs = { 0 : "hello", 1 : 123 }
  layer.dataProvider().changeAttributeValues({ fid : attrs })

if caps & QgsVectorDataProvider.ChangeGeometries:
  geom = QgsGeometry.fromPoint(QgsPoint(111,222))
  layer.dataProvider().changeGeometryValues({ fid : geom })
```

4.2.4 Adăugarea i eliminarea câmpurilor

Pentru a adăuga câmpuri (atribute), trebuie să specificai o listă de definiii pentru acestea. Pentru tergerea de câmpuri e suficientă furnizarea unei liste de indeci pentru câmpuri.

```
if caps & QgsVectorDataProvider.AddAttributes:
  res = layer.dataProvider().addAttributes([QgsField("mytext", QVariant.String), QgsField("myint"

if caps & QgsVectorDataProvider.DeleteAttributes:
  res = layer.dataProvider().deleteAttributes([0])
```

After adding or removing fields in the data provider the layer's fields need to be updated because the changes are not automatically propagated.

```
layer.updateFields()
```

4.3 Modificarea straturi vectoriale prin editarea unui tampon de memorie

Când editai vectori în aplicaia QGIS, în primul rând, trebuie să comutai în modul de editare pentru stratul în care lucrai, apoi să efectuai modificări pe care, în cele din urmă, să le salvai (sau să le anulai). Modificările nu vor fi scrise până când nu sunt salvate — ele rezidând în memorie, în tamponul de editare al stratului. De asemenea, este posibilă utilizarea programatică a acestei funcionalităi — aceasta fiind doar o altă metodă pentru editarea straturilor vectoriale, care completează utilizarea directă a furnizorilor de date. Utilizai această opiune atunci când furnizai unele instrumente GUI pentru editarea straturilor vectoriale, permiând utilizatorului să decidă dacă să salveze/anuleze, i punându-i la dispoziie facilităile de undo/redo. Atunci când salvai modificările, acestea vor fi transferate din memoria tampon de editare în furnizorul de date.

Pentru a afla dacă un strat se află în modul de editare, utilizai `isEditing()` — funciile de editare funcionînd numai atunci când modul de editare este activat. Utilizarea funciilor de editare

```
# add two features (QgsFeature instances)
layer.addFeatures([feat1,feat2])
# delete a feature with specified ID
layer.deleteFeature(fid)

# set new geometry (QgsGeometry instance) for a feature
layer.changeGeometry(fid, geometry)
# update an attribute with given field index (int) to given value (QVariant)
layer.changeAttributeValue(fid, fieldIndex, value)

# add new field
layer.addAttribute(QgsField("mytext", QVariant.String))
# remove a field
layer.deleteAttribute(fieldIndex)
```

Pentru ca undo/redo să funcioneze în mod corespunzător, apelurile de mai sus trebuie să fie înglobate în comenzi undo. (Dacă nu vă pasă de undo/redo i dorii să stocai imediat modificările, atunci vei avea o sarcină mai uoară prin :ref: *folosirea <editorului> furnizorului de date*.) Cum să utilizai funcionalitatea undo

```
layer.beginEditCommand("Feature triangulation")

# ... call layer's editing methods ...

if problem_occurred:
  layer.destroyEditCommand()
  return

# ... more editing ...

layer.endEditCommand()
```

`beginEndCommand()` va crea o comandă internă "activă" i va înregistra modificările ulterioare din stratul vectorial. Cu apelul către `endEditCommand()` comanda este împinsă pe stiva undo, iar utilizatorul va putea efectua undo/redo prin GUI. În cazul în care ceva nu a mers bine pe timpul efectuării schimbărilor, metoda `destroyEditCommand()` va elimina comanda i va da înapoi toate modificările făcute pe perioada când această comandă a fost activă.

Pentru a activa modul de editare, este disponibilă metoda `startEditing()`, pentru a opri editarea există :func:*commitChanges* i `rollback ()` — totui, în mod normal, ar trebui să nu nevoie de aceste metode i să permitei utilizatorului declanarea acestor funcionalităi.

4.4 Crearea unui index spaial

Indecii spaiali pot îmbunătăi dramatic performana codului dvs, în cazul în care este nevoie să interogai frecvent un strat vectorial. Imaginai-vă, de exemplu, că scriei un algoritm de interpolare, i că, pentru o anumită locaie, trebuie să aflai cele mai apropiate 10 puncte dintr-un strat, în scopul utilizării acelor puncte în calculul valorii interpolate. Fără un index spaial, singura modalitate pentru QGIS de a găsi cele 10 puncte, este de a calcula distana tuturor punctelor faă de locaia specificată i apoi de a compara aceste distane. Această sarcină poate fi mare consumatoare de timp, mai ales în cazul în care trebuie să fie repetată pentru mai multe locaii. Dacă pentru stratul respectiv există un index spaial, operaiunea va fi mult mai eficientă.

Gândii-vă la un strat fără index spaial ca la o carte de telefon în care numerele de telefon nu sunt ordonate sau indexate. Singura modalitate de a afla numărul de telefon al unei anumite persoane este de a citi toate numerele, începând cu primul, până când îl găsii.

Indicii spaiali nu sunt creai în mod implicit pentru un strat vectorial QGIS, dar îi putei realiza cu uurină. Iată ce trebuie să facei.

1. creare index spaial — următorul cod creează un index vid

```
index = QgsSpatialIndex()
```

2. adăugare entităi la index — indexul ia obiectul `QgsFeature` i-l adaugă la structura internă de date. Putei crea obiectul manual sau putei folosi unul dintre apelurile anterioare către funcia `nextFeature()` a furnizorului

```
index.insertFeature(feat)
```

3. o dată ce ai introdus valori în indexul spaial, putei efectua unele interogări

```
# returns array of feature IDs of five nearest features
nearest = index.nearestNeighbor(QgsPoint(25.4, 12.7), 5)

# returns array of IDs of features which intersect the rectangle
intersect = index.intersects(QgsRectangle(22.5, 15.3, 23.1, 17.2))
```

4.5 Scrierea straturilor vectoriale

Putei scrie în fiierele coninând straturi vectoriale folosind clasa `QgsVectorFileWriter`. Aceasta acceptă orice alt tip de fiier vector care suportă OGR (fiiere shape, GeoJSON, KML i altele).

Există două posibilităi de a exporta un strat vectorial:

- dintr-o instană a `QgsVectorLayer`

```
error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_shapes.shp", "CP1250", None, "ESRI

if error == QgsVectorFileWriter.NoError:
  print "success!"

error = QgsVectorFileWriter.writeAsVectorFormat(layer, "my_json.json", "utf-8", None, "GeoJSO
if error == QgsVectorFileWriter.NoError:
  print "success again!"
```

Al treilea parametru se referă la codificarea textului de ieire. Doar unele formate au nevoie de acest lucru pentru o funcionare corectă - fiierul shape fiind printre ele — totui, în cazul în care nu utilizai caractere internaionale nu trebuie să vă îngrijoreze codificarea. În al patrulea parametru, care acum are valoarea `None`, se poate specifica destinaia CRS — dacă este trecută o instană validă a `QgsCoordinateReferenceSystem`, stratul este transformat pentru acel CRS.

Pentru denumirile valide ale driver-elor, vă rugăm să consultai formatele suportate de OGR — ar trebui să trecei valoarea în coloana "Code", ca i nume de driver. Opional, putei stabili dacă se exportă numai entităile selectate, transmiând opiunile specifice driver-ului pentru creare sau indicând generatorului să nu creeze atribute — analizai documentaia pentru sintaxa completă.

- direct din entităi

```
# define fields for feature attributes. A list of QgsField objects is needed
fields = [QgsField("first", QVariant.Int),
          QgsField("second", QVariant.String)]

# create an instance of vector file writer, which will create the vector file.
# Arguments:
# 1. path to new file (will fail if exists already)
# 2. encoding of the attributes
# 3. field map
# 4. geometry type - from WKBTYPE enum
# 5. layer's spatial reference (instance of
#    QgsCoordinateReferenceSystem) - optional
# 6. driver name for the output file
writer = QgsVectorFileWriter("my_shapes.shp", "CP1250", fields, QGis.WKBPoint, None, "ESRI Sh
```

```
if writer.hasError() != QgsVectorFileWriter.NoError:
  print "Error when creating shapefile: ", writer.hasError()

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes([1, "text"])
writer.addFeature(fet)

# delete the writer to flush features to disk (optional)
del writer
```

4.6 Furnizorul de memorie

Furnizorul de memorie este destinat, în principal, dezvoltatorilor de plugin-uri sau de aplicaii ter3. El nu stochează date pe disc, permiând dezvoltatorilor să-l folosească ca pe un depozit rapid pentru straturi temporare.

Furnizorul suportă câmpuri de tip string, int sau double.

Furnizorul de memorie suportă, de asemenea, indexarea spaială, care este activată prin apelarea furnizorului funciei `createSpatialIndex()`. O dată ce indexul spaial este creat, vei fi capabili de a parcurge mai rapid entităile, în interiorul unor regiuni mai mici (din moment ce nu este necesar să traversai toate entităile, ci doar pe cele din dreptunghiul specificat).

Un furnizor de memorie este creat prin transmiterea `"memoriei"` ca ir furnizor către constructorul `QgsVectorLayer`.

Constructorul are, de asemenea, un URI care definete unul din următoarele tipuri de geometrie a stratului: `"Point"`, `"LineString"`, `"Polygon"`, `"MultiPoint"`, `"MultiLineString"` sau `"MultiPolygon"`.

URI poate specifica, de asemenea, sistemul de coordonate de referină, câmpurile, precum i indexarea furnizorului de memorie. Sintaxa este:

crs=definiie Specificai sistemul de referină de coordonate, unde definiia poate fi oricare din formele acceptate de: `QgsCoordinateReferenceSystem.createFromString()`

index=yes Specificai dacă furnizorul va utiliza un index spaial.

field=nume:tip(lungime,precizie) Specificai un atribut al stratului. Atributul are un nume i, opional, un tip (integer, double sau string), lungime i precizie. Pot exista mai multe definiii de câmp.

Următorul exemplu de URI încorporează toate aceste opiuni

```
"Point?crs=epsg:4326&field=id:integer&field=name:string(20)&index=yes"
```

Următorul exemplu de cod ilustrează crearea i popularea unui furnizor de memorie

```
# create layer
vl = QgsVectorLayer("Point", "temporary_points", "memory")
pr = vl.dataProvider()

# add fields
pr.addAttributes([QgsField("name", QVariant.String),
                    QgsField("age",  QVariant.Int),
                    QgsField("size", QVariant.Double)])

# add a feature
fet = QgsFeature()
fet.setGeometry(QgsGeometry.fromPoint(QgsPoint(10,10)))
fet.setAttributes(["Johny", 2, 0.3])
pr.addFeatures([fet])
```

```
# update layer's extent when new features have been added
# because change of extent in provider is not propagated to the layer
vl.updateExtents()
```

În cele din urmă, să verificăm dacă totul a mers bine

```
# show some stats
print "fields:", len(pr.fields())
print "features:", pr.featureCount()
e = layer.extent()
print "extent:", e.xMin(),e.yMin(),e.xMax(),e.yMax()

# iterate over features
f = QgsFeature()
features = vl.getFeatures()
for f in features:
  print "F:",f.id(), f.attributes(), f.geometry().asPoint()
```

4.7 Aspectul (simbologia) straturilor vectoriale

Când un strat vector este randat, aspectul datelor este dat de **render** i de **simbolurile** asociate stratului. Simbolurile sunt clase care au grijă de reprezentarea vizuală a tuturor entităilor, în timp ce un render determină ce simbol va fi folosit doar pentru anumite entităi.

Tipul de render pentru un strat oarecare poate fi obinut astfel:

```
renderer = layer.rendererV2()
```

i cu acea referină, să explorăm un pic

```
print "Type:", rendererV2.type()
```

Există mai multe tipuri de rendere disponibile în biblioteca de bază a QGIS:

Tipul	Clasa	Descrierea
singleSymbol	QgsSingleSymbolRendererV2	Asociază tuturor entităilor acelai simbol
categorizedSymbol	QgsCategorizedSymbolRendererV2	Asociază entităilor un simbol diferit, în funcie de categorie
graduatedSymbol	QgsGraduatedSymbolRendererV2	Asociază fiecărei entităi un simbol diferit pentru fiecare gamă de valori

There might be also some custom renderer types, so never make an assumption there are just these types. You can query `QgsRendererV2Registry` singleton to find out currently available renderers.

Este posibilă obinerea coninutului renderului sub formă de text — lucru util pentru depanare

```
print rendererV2.dump()
```

4.7.1 Render cu Simbol Unic

Putei obine simbolul folosit pentru randare apelând metoda `simbol()`, i-l putei schimba cu ajutorul metodei `setSymbol()` (notă pentru dezvoltatorii C++: renderul devine proprietarul simbolului.)

4.7.2 Render cu Simboluri Categorisite

Putei interoga i seta numele atributului care este folosit pentru clasificare: folosii metodele `classAttribute()` i `setClassAttribute()`.

Pentru a obine o listă de categorii

```
for cat in rendererV2.categories():
  print "%s: %s :: %s" % (cat.value().toString(), cat.label(), str(cat.symbol()))
```

În cazul în care `value()` reprezintă valoarea utilizată pentru discriminare între categorii, `label()` este un text utilizat pentru descrierea categorie iar metoda `symbol()` returnează simbolul asignat.

Renderul, de obicei, stochează atât simbolul original cât i gamele de culoare care au fost utilizate pentru clasificare: metodele `sourceColorRamp()` i `sourceSymbol()`.

4.7.3 Render cu Simboluri Graduale

Acest render este foarte similar cu renderul cu simbol clasificat, descris mai sus, dar în loc de o singură valoare de atribut per clasă el lucrează cu intervale de valori, putând fi, astfel, utilizat doar cu atribute numerice.

Pentru a afla mai multe despre gamele utilizate în render

```
for ran in rendererV2.ranges():
  print "%f - %f: %s %s" % (
      ran.lowerValue(),
      ran.upperValue(),
      ran.label(),
      str(ran.symbol())
     )
```

putei folosi din nou `classAttribute()` pentru a afla numele atributului de clasificare, metodele `sourceSymbol()` i `sourceColorRamp()`. În plus, există metoda `mode()` care determină modul în care au fost create gamele: folosind intervale egale, cuantile sau o altă metodă.

Dacă dorii să creai propriul render cu simbol gradual, putei face acest lucru aa cum este ilustrat în fragmentul de mai jos (care creează un simplu aranjament cu două clase)

```
from qgis.core import *

myVectorLayer = QgsVectorLayer(myVectorPath, myName, 'ogr')
myTargetField = 'target_field'
myRangeList = []
myOpacity = 1
# Make our first symbol and range...
myMin = 0.0
myMax = 50.0
myLabel = 'Group 1'
myColour = QtGui.QColor('#ffee00')
mySymbol1 = QgsSymbolV2.defaultSymbol(myVectorLayer.geometryType())
mySymbol1.setColor(myColour)
mySymbol1.setAlpha(myOpacity)
myRange1 = QgsRendererRangeV2(myMin, myMax, mySymbol1, myLabel)
myRangeList.append(myRange1)
#now make another symbol and range...
myMin = 50.1
myMax = 100
myLabel = 'Group 2'
myColour = QtGui.QColor('#00eeff')
mySymbol2 = QgsSymbolV2.defaultSymbol(
     myVectorLayer.geometryType())
mySymbol2.setColor(myColour)
mySymbol2.setAlpha(myOpacity)
myRange2 = QgsRendererRangeV2(myMin, myMax, mySymbol2 myLabel)
myRangeList.append(myRange2)
myRenderer = QgsGraduatedSymbolRendererV2('', myRangeList)
myRenderer.setMode(QgsGraduatedSymbolRendererV2.EqualInterval)
myRenderer.setClassAttribute(myTargetField)
```

```
myVectorLayer.setRendererV2(myRenderer)
QgsMapLayerRegistry.instance().addMapLayer(myVectorLayer)
```

4.7.4 Lucrul cu Simboluri

Pentru reprezentarea simbolurilor există clasa de bază `QgsSymbolV2`, având trei clase derivate:

- `QgsMarkerSymbolV2` — pentru entităi de tip punct
- `QgsLineSymbolV2` — pentru entităi de tip linie
- `QgsFillSymbolV2` — pentru entităi de tip poligon

Fiecare simbol este format din unul sau mai multe straturi (clase derivate din `QgsSymbolLayerV2`). Straturile simbolului realizează în mod curent randarea, clasa simbolului servind doar ca un container pentru acestea.

Având o instană a unui simbol (de exemplu, de la un render), este posibil să o explorăm: metoda `type()` spunându-ne dacă acesta este un marker, o linie sau un simbol de umplere. Există i metoda `dump()` care returnează o scurtă descriere a simbolului. Pentru a obine o listă a straturilor simbolului

```
for i in xrange(symbol.symbolLayerCount()):
  lyr = symbol.symbolLayer(i)
  print "%d: %s" % (i, lyr.layerType())
```

Pentru a afla culoarea simbolului folosii metoda `color()`, iar pentru a schimba culoarea `setColor()`. În cazul simbolurilor marker, în plus, putei interoga pentru dimensiunea simbolului i unghiul de rotaie cu metodele `size()` i `angle()`, iar pentru simbolurile linie există metoda `width()` care returnează lăimea liniei.

Dimensiunea i lăimea sunt în milimetri, în mod implicit, iar unghiurile sunt în grade.

Lucrul cu Straturile Simbolului

Aa cum s-a arătat mai înainte, straturile simbolului (subclase ale `QgsSymbolLayerV2`), determină aspectul entităilor. Există mai multe clase de strat simbol de bază, pentru uzul general. Este posibilă implementarea unor noi tipuri de strat simbol i, astfel, personalizarea în mod arbitrar a modului în care vor fi randate entităile. Metoda `layerType()` identifică în mod unic clasa stratului simbol — tipurile de straturi simbol de bază i implicite sunt SimpleMarker, SimpleLine i SimpleFill.

Putei obine, în modul următor, o listă completă a tipurilor de straturi pe care le putei crea pentru o anumită clasă de simboluri

```
from qgis.core import QgsSymbolLayerV2Registry
myRegistry = QgsSymbolLayerV2Registry.instance()
myMetadata = myRegistry.symbolLayerMetadata("SimpleFill")
for item in myRegistry.symbolLayersForType(QgsSymbolV2.Marker):
  print item
```

Rezultat

```
EllipseMarker
FontMarker
SimpleMarker
SvgMarker
VectorField
```

clasa `QgsSymbolLayerV2Registry` gestionează o bază de date a tuturor tipurilor de straturi simbol disponibile.

Pentru a accesa datele stratului simbol, folosii metoda `properties()` care returnează un dicionar cu valori-cheie ale proprietăilor care îi determină aparena. Fiecare tip de strat simbol are un set specific de proprietăi pe care le utilizează. În plus, există metodele generice `color()`, `size()`, `angle()`, `width()` împreună cu cu omologii lor de setare. Desigur, mărimea i unghiul sunt disponibile doar pentru straturi simbol de tip marcer iar lăimea pentru straturi simbol de tip linie.

Crearea unor Tipuri Personalizate de Straturi pentru Simboluri

Imaginai-vă că ai dori să personalizai modul în care se randează datele. Vă putei crea propria dvs. clasă de strat de simbol, care va desena entităile exact aa cum dorii. Iată un exemplu de marker care desenează cercuri roii cu o rază specificată

```
class FooSymbolLayer(QgsMarkerSymbolLayerV2):

  def __init__(self, radius=4.0):
    QgsMarkerSymbolLayerV2.__init__(self)
    self.radius = radius
    self.color = QColor(255,0,0)

  def layerType(self):
    return "FooMarker"

  def properties(self):
    return { "radius" : str(self.radius) }

  def startRender(self, context):
    pass

  def stopRender(self, context):
    pass

  def renderPoint(self, point, context):
    # Rendering depends on whether the symbol is selected (Qgis >= 1.5)
    color = context.selectionColor() if context.selected() else self.color
    p = context.renderContext().painter()
    p.setPen(color)
    p.drawEllipse(point, self.radius, self.radius)

  def clone(self):
    return FooSymbolLayer(self.radius)
```

Metoda `layerType()` determină numele stratului simbol, acesta trebuind să fie unic printre toate straturile simbol. Proprietăile sunt utilizate pentru persistena atributelor. Metoda `clone()` trebuie să returneze o copie a stratului simbol, având toate atributele exact la fel. În cele din urmă, mai există metodele de randare: `startRender()` care este apelată înainte de randarea primei entităi, i `stopRender()` care oprete randarea. Efectiv, randarea are loc cu ajutorul metodei `renderPoint()`. Coordonatele punctului(punctelor) sunt deja transformate la coordonatele de ieire.

Pentru polilinii i poligoane singura diferenă constă în metoda de randare: ar trebui să utilizai `renderPolyline()` care primete o listă de linii, respectiv `renderPolygon()` care primete lista de puncte de pe inelul exterior ca prim parametru i o listă de inele interioare (sau nici unul), ca al doilea parametru.

De obicei, este convenabilă adăugarea unui GUI pentru setarea atributelor tipului de strat pentru simboluri, pentru a permite utilizatorilor să personalizeze aspectul: în exemplul de mai sus, putem lăsa utilizatorul să seteze raza cercului. Codul de mai jos implementează un astfel de widget

```
class FooSymbolLayerWidget(QgsSymbolLayerV2Widget):
  def __init__(self, parent=None):
    QgsSymbolLayerV2Widget.__init__(self, parent)

    self.layer = None

    # setup a simple UI
    self.label = QLabel("Radius:")
    self.spinRadius = QDoubleSpinBox()
    self.hbox = QHBoxLayout()
    self.hbox.addWidget(self.label)
    self.hbox.addWidget(self.spinRadius)
    self.setLayout(self.hbox)
```

```
    self.connect(self.spinRadius, SIGNAL("valueChanged(double)"), \
      self.radiusChanged)

  def setSymbolLayer(self, layer):
    if layer.layerType() != "FooMarker":
      return
    self.layer = layer
    self.spinRadius.setValue(layer.radius)

  def symbolLayer(self):
    return self.layer

  def radiusChanged(self, value):
    self.layer.radius = value
    self.emit(SIGNAL("changed()"))
```

Acest widget poate fi integrat în fereastra de proprietăi a simbolului. În cazul în care tipul de strat simbol este selectat în fereastra de proprietăi a simbolului, se creează o instană a stratului simbol i o instană a widget-ului stratului simbol. Apoi, se apelează metoda `setSymbolLayer()` pentru a aloca stratul simbol widget-ului. În acea metodă, widget-ul ar trebui să actualizeze UI pentru a reflecta atributele stratului simbol. Funcia `symbolLayer()` este utilizată la preluarea stratului simbol din fereastra de proprietăi, în scopul folosirii sale pentru simbol.

La fiecare schimbare de atribute, widget-ul ar trebui să emită semnalul `changed()` pentru a permite ferestrei de proprietăi să-i actualizeze previzualizarea simbolului.

Acum mai lipsete doar liantul final: pentru a face QGIS contient de aceste noi clase. Acest lucru se face prin adăugarea stratului simbol la registru. Este posibilă utilizarea stratului simbol, de asemenea, fără a-l adăuga la registru, dar unele funcionalităi nu vor fi disponibile: de exemplu, încărcarea de fiiere de proiect cu straturi simbol personalizate sau incapacitatea de a edita atributele stratului în GUI.

Va trebui să creăm metadate pentru stratul simbolului

```
class FooSymbolLayerMetadata(QgsSymbolLayerV2AbstractMetadata):

  def __init__(self):
    QgsSymbolLayerV2AbstractMetadata.__init__(self, "FooMarker", QgsSymbolV2.Marker)

  def createSymbolLayer(self, props):
    radius = float(props[QString("radius")]) if QString("radius") in props else 4.0
    return FooSymbolLayer(radius)

  def createSymbolLayerWidget(self):
    return FooSymbolLayerWidget()

QgsSymbolLayerV2Registry.instance().addSymbolLayerType(FooSymbolLayerMetadata())
```

Ar trebui să transmitei tipul stratului (cel returnat de către strat) i tipul de simbol (marker/linie/umplere) către constructorul clasei părinte. `createSymbolLayer()` are grijă de a crea o instană de strat simbol cu atributele specificate în dicionarul *props*. (Atenie, tastele reprezintă instane QString, nu obiecte "str"). Există, de asemenea, metoda `createSymbolLayerWidget()` care returnează setările widget-ului pentru acest tip de strat simbol.

Ultimul pas este de a adăuga acest strat simbol la registru — i am încheiat.

4.7.5 Crearea renderelor Personalizate

Ar putea fi utilă crearea unei noi implementări de render, dacă dorii să personalizai regulile de selectare a simbolurilor pentru randarea entităilor. Unele cazuri de utilizare: simbolul să fie determinat de o combinaie de câmpuri, dimensiunea simbolurilor să depindă în funcie de scara curentă, etc

Urmatorul cod prezintă o simplă randare personalizată, care creează două simboluri de tip marker i apoi alege aleatoriu unul dintre ele pentru fiecare entitate

```
import random

class RandomRenderer(QgsFeatureRendererV2):
  def __init__(self, syms=None):
    QgsFeatureRendererV2.__init__(self, "RandomRenderer")
    self.syms = syms if syms else [QgsSymbolV2.defaultSymbol(QGis.Point), QgsSymbolV2.defaultSymb

  def symbolForFeature(self, feature):
    return random.choice(self.syms)

  def startRender(self, context, vlayer):
    for s in self.syms:
      s.startRender(context)

  def stopRender(self, context):
    for s in self.syms:
      s.stopRender(context)

  def usedAttributes(self):
    return []

  def clone(self):
    return RandomRenderer(self.syms)
```

Constructorul clasei părinte `QgsFeatureRendererV2` are nevoie de numele renderului (trebuie să fie unic printre rendere). Metoda `symbolForFeature()` este cea care decide ce simbol va fi folosit pentru o anumită entitate. `startRender()` i `stopRender()` vor avea grijă de iniializarea/finalizarea randării simbolului. Metoda `usedAttributes()` poate returna o listă de nume de câmpuri a căror prezenă o ateaptă renderul. În cele din urmă `clone()` ar trebui să returneze o copie a renderului.

Ca i în cazul straturilor simbol, este posibilă ataarea unui GUI pentru configurarea renderului. Acesta trebuie să fie derivat din `QgsRendererV2Widget`. Următorul exemplu de cod creează un buton care permite utilizatorului setarea primului simbol

```
class RandomRendererWidget(QgsRendererV2Widget):
  def __init__(self, layer, style, renderer):
    QgsRendererV2Widget.__init__(self, layer, style)
    if renderer is None or renderer.type() != "RandomRenderer":
      self.r = RandomRenderer()
    else:
      self.r = renderer
    # setup UI
    self.btn1 = QgsColorButtonV2("Color 1")
    self.btn1.setColor(self.r.syms[0].color())
    self.vbox = QVBoxLayout()
    self.vbox.addWidget(self.btn1)
    self.setLayout(self.vbox)
    self.connect(self.btn1, SIGNAL("clicked()"), self.setColor1)

  def setColor1(self):
    color = QColorDialog.getColor(self.r.syms[0].color(), self)
    if not color.isValid(): return
    self.r.syms[0].setColor(color);
    self.btn1.setColor(self.r.syms[0].color())

  def renderer(self):
    return self.r
```

Constructorul primete instane ale stratului activ (`QgsVectorLayer`), stilul global (`QgsStyleV2`) i renderul curent. Dacă nu există un render sau renderul are alt tip, acesta va fi înlocuit cu noul nostru render, în caz contrar vom folosi renderul curent (care are deja tipul de care avem nevoie). Coninutul widget-ului ar trebui să fie actualizat pentru a arăta starea actuală a renderului. Când dialogul renderului este acceptat, metoda `renderer()` a widgetului este apelată pentru a obine renderul curent — acesta fiind atribuit stratului.

Ultimul bit lipsă este cel al metadatelor renderului i înregistrarea în registru, altfel încărcarea straturilor cu renderul nu va funciona, iar utilizatorul nu va fi capabil să-l selecteze din lista de rendere. Să finalizăm exemplul nostru de RandomRenderer

```
class RandomRendererMetadata(QgsRendererV2AbstractMetadata):
  def __init__(self):
    QgsRendererV2AbstractMetadata.__init__(self, "RandomRenderer", "Random renderer")

  def createRenderer(self, element):
    return RandomRenderer()
  def createRendererWidget(self, layer, style, renderer):
    return RandomRendererWidget(layer, style, renderer)

QgsRendererV2Registry.instance().addRenderer(RandomRendererMetadata())
```

În mod similar cu straturile simbol, constructorul de metadate abstracte ateaptă numele renderului, nume vizibil pentru utilizatori i numele opional al pictogramei renderului. Metoda `createRenderer()` transmite instana `QDomElement` care poate fi folosită pentru a restabili starea renderului din arborele DOM. Metoda `createRendererWidget()` creează widget-ul de configurare. Aceasta nu trebuie să fie prezent sau ar putea returna *None*, dacă renderul nu vine cu GUI-ul.

Pentru a asocia o pictogramă renderului ai putea să o asignai în constructorul `QgsRendererV2AbstractMetadata` ca un al treilea argument (opional) — constructorul clasei de bază din funcia `__init__()` a RandomRendererMetadata devine

```
QgsRendererV2AbstractMetadata.__init__(self,
    "RandomRenderer",
    "Random renderer",
    QIcon(QPixmap("RandomRendererIcon.png", "png")))
```

Pictograma poate fi asociată ulterior, de asemenea, în orice moment, folosind metoda `setIcon()` a clasei de metadate. Pictograma poate fi încărcată dintr-un fiier (aa cum s-a arătat mai sus), sau dintr-o resursă Qt (PyQt4 include compilatorul .qrc pentru Python).

4.8 Lecturi suplimentare

DE EFECTUAT: crearea/modificarea simbolurilor, modificarea stilului (`QgsStyleV2`), modificarea gamelor de culori (`QgsVectorColorRampV2`), rendere bazate pe reguli (citii această postare pe blog), explorarea straturilor unui simbol i a regitrilor renderelor

Manipularea geometriei

Punctele, liniile i poligoanele, care reprezintă entităi spaiale sunt frecvent menionate ca geometrii. În QGIS acestea sunt reprezentate de clasa `QgsGeometry`. Toate tipurile de geometrie posibile sunt frumos prezentate în pagina de discuii JTS.

Uneori, o geometrie poate fi de fapt o colecie de simple geometrii (simple-păr). O astfel de geometrie poartă denumirea de geometrie multi-parte. În cazul în care conine doar un singur tip de geometrie simplă, o denumim multi-punct, multi-linie sau multi-poligon. De exemplu, o ară formată din mai multe insule poate fi reprezentată ca un multi-poligon.

Coordonatele geometriilor pot fi în orice sistem de coordonate de referină (CRS). Când extragem entităile dintr-un strat, geometriile asociate vor avea coordonatele în CRS-ul stratului.

5.1 Construirea geometriei

Există mai multe opiuni pentru a crea o geometrie:

- from coordinates

  ```
  gPnt = QgsGeometry.fromPoint(QgsPoint(1,1))
  gLine = QgsGeometry.fromPolyline( [ QgsPoint(1,1), QgsPoint(2,2) ] )
  gPolygon = QgsGeometry.fromPolygon( [ [ QgsPoint(1,1), QgsPoint(2,2), QgsPoint(2,1) ] ] )
  ```

  Coordonatele sunt obinute folosind clasa `QgsPoint`.

  O polilinie (linie) este reprezentată de o listă de puncte. Poligonul este reprezentat de o listă de inele liniare (de exemplu, linii închise). Primul inel este cel exterior (limita), inele ulterioare opionale reprezentând găurile din poligon.

  Geometriile multi-parte merg cu un nivel mai departe: multi-punctele sunt o listă de puncte, multi-liniile o listă de linii iar multi-poligoanele sunt o listă de poligoane.

- from well-known text (WKT)

  ```
  gem = QgsGeometry.fromWkt("POINT (3 4)")
  ```

- from well-known binary (WKB)

  ```
  g = QgsGeometry()
  g.setWkbAndOwnership(wkb, len(wkb))
  ```

5.2 Accesarea geometriei

First, you should find out geometry type, `wkbType()` method is the one to use — it returns a value from QGis.WkbType enumeration

```
>>> gPnt.wkbType() == QGis.WKBPoint
True
>>> gLine.wkbType() == QGis.WKBLineString
True
>>> gPolygon.wkbType() == QGis.WKBPolygon
True
>>> gPolygon.wkbType() == QGis.WKBMultiPolygon
False
```

As an alternative, one can use `type()` method which returns a value from QGis.GeometryType enumeration. There is also a helper function `isMultipart()` to find out whether a geometry is multipart or not.

Pentru a extrage informaii din geometrie, există funciile accessor pentru fiecare tip de vector. Iată cum le putei utiliza

```
>>> gPnt.asPoint()
(1,1)
>>> gLine.asPolyline()
[(1,1), (2,2)]
>>> gPolygon.asPolygon()
[[(1,1), (2,2), (2,1), (1,1)]]
```

Notă: tuplurile (x, y) nu reprezintă tupluri reale, ele sunt obiecte :class:*QgsPoint*, valorile fiind accesibile cu ajutorul metodelor `x()` i `y()` .

Pentru geometriile multiparte există funcii accessor similare: `asMultiPoint()`, `asMultiPolyline()`, `asMultiPolygon ()`.

5.3 Predicate i operaiuni geometrice

QGIS folosete biblioteca GEOS pentru operaiuni geometrice avansate, cum ar fi predicatele geometrice (`contains()`, `intersects()`, ...) i operaiunile de setare (`union()`, `difference()`, ...). Se pot calcula, de asemenea, proprietăile geometrice, cum ar fi suprafaa (în cazul poligoanelor) sau lungimea (pentru poligoane i linii)

Iată un mic exemplu care combină iterarea entităilor dintr-un strat dat i efectuarea unor calcule bazate pe geometriile lor.

```
# we assume that 'layer' is a polygon layer
features = layer.getFeatures()
for f in features:
  geom = f.geometry()
  print "Area:", geom.area()
  print "Perimeter:", geom.length()
```

Ariile i perimetrele nu iau în considerare CRS-ul atunci când sunt calculate folosind metodele clasei `QgsGeometry`. Pentru un calcul mult mai puternic al ariei i al distanei se poate utiliza clasa `QgsDistanceArea`. În cazul în care proieciile sunt dezactivate, calculele vor fi planare, în caz contrar acestea vor fi efectuate pe un elipsoid. Când elipsoidul nu este setat în mod explicit, parametrii WGS84 vor fi utilizai pentru calcule.

```
d = QgsDistanceArea()
d.setProjectionsEnabled(True)

print "distance in meters: ", d.measureLine(QgsPoint(10,10),QgsPoint(11,11))
```

Putei căuta mai multe exemple de algoritmi care sunt inclui în QGIS i să folosii aceste metode pentru a analiza i a transforma datele vectoriale. Mai jos sunt prezente câteva trimiteri spre codul unora dintre ele.

Additional information can be found in followinf sources:

- Geometry transformation: Reproject algorithm

- Distance and area using the `QgsDistanceArea` class: Distance matrix algorithm
- Multi-part to single-part algorithm

CAPITOLUL 6

Proiecii suportate

6.1 Sisteme de coordonate de referină

Sisteme de coordonate de referină (SIR) sunt încapsulate de către clasa `QgsCoordinateReferenceSystem`. Instanele acestei clase pot fi create prin mai multe moduri diferite:

- specify CRS by its ID

  ```
  # PostGIS SRID 4326 is allocated for WGS84
  crs = QgsCoordinateReferenceSystem(4326, QgsCoordinateReferenceSystem.PostgisCrsId)
  ```

  QGIS folosete trei ID-uri diferite pentru fiecare sistem de referină:

  - `PostgisCrsId` — ID-uri folosite în interiorul bazei de date PostGIS.
  - `InternalCrsId` — ID-uri folosite în baza de date QGIS.
  - `EpsgCrsId` — ID-uri asignate de către organizaia EPSG

  În cazul în care nu se specifică altfel în al doilea parametru, PostGIS SRID este utilizat în mod implicit.

- specify CRS by its well-known text (WKT)

  ```
  wkt = 'GEOGCS["WGS84", DATUM["WGS84", SPHEROID["WGS84", 6378137.0, 298.257223563]],'
         PRIMEM["Greenwich", 0.0], UNIT["degree",0.017453292519943295],'
         AXIS["Longitude",EAST], AXIS["Latitude",NORTH]]'
  crs = QgsCoordinateReferenceSystem(wkt)
  ```

- create invalid CRS and then use one of the `create*()` functions to initialize it. In following example we use Proj4 string to initialize the projection

  ```
  crs = QgsCoordinateReferenceSystem()
  crs.createFromProj4("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs")
  ```

Este înelept să verificăm dacă a avut loc crearea cu succes a CRS-ului (de exemplu, efectuând o căutare în baza de date): `isValid()` trebuie să întoarcă `True`.

Reinei că pentru iniializarea sistemelor de referină spaiale, QGIS trebuie să caute valorile corespunzătoare în baza de date internă `srs.db`. Astfel, în cazul în care creai o aplicaie independentă va trebui să stabilii corect căile, cu ajutorul `QgsApplication.setPrefixPath()`, în caz contrar baza de date nu va fi găsită. Dacă executai comenzile din consola QGIS python sau dezvoltai vreun plugin, atunci totul este în regulă: totul este deja configurat pentru dvs.

Accesarea informaiilor sistemului de referină spaial

```
print "QGIS CRS ID:", crs.srsid()
print "PostGIS SRID:", crs.srid()
print "EPSG ID:", crs.epsg()
print "Description:", crs.description()
print "Projection Acronym:", crs.projectionAcronym()
```

```
print "Ellipsoid Acronym:", crs.ellipsoidAcronym()
print "Proj4 String:", crs.proj4String()
# check whether it's geographic or projected coordinate system
print "Is geographic:", crs.geographicFlag()
# check type of map units in this CRS (values defined in QGis::units enum)
print "Map units:", crs.mapUnits()
```

6.2 Proiecii

Putei face transformarea între diferitele sisteme de referină spaiale, cu ajutorul clasei `QgsCoordinateTransform`. Cel mai simplu mod de a o folosi este de a crea CRS-urile sursă i destinaie i să construii cu ele o instană `QgsCoordinateTransform`. Apoi, doar repetai apelul funciei `transform()` pentru a realiza transformarea. În mod implicit, aceasta face transformarea în ordinea deja precizată, dar este capabilă de a face i transformarea inversă

```
crsSrc = QgsCoordinateReferenceSystem(4326)    # WGS 84
crsDest = QgsCoordinateReferenceSystem(32633)  # WGS 84 / UTM zone 33N
xform = QgsCoordinateTransform(crsSrc, crsDest)

# forward transformation: src -> dest
pt1 = xform.transform(QgsPoint(18,5))
print "Transformed point:", pt1

# inverse transformation: dest -> src
pt2 = xform.transform(pt1, QgsCoordinateTransform.ReverseTransform)
print "Transformed back:", pt2
```

Folosirea suportului de hartă

Widget-ul suportului de hartă este, probabil, cel mai important în QGIS, deoarece prezintă o hartă compusă din straturi suprapuse i permite atât interaciunea cu harta cât i cu straturile. Suportul arată întotdeauna o parte a hării definită de caseta de încadrare curentă. Interaciunea se realizează prin utilizarea unor **instrumente pentru hartă**: există instrumente de panoramare, de mărire, de identificare a straturilor, de măsurare, de editare vectorială i altele. Similar altor programe de grafică, există întotdeauna un instrument activ, iar utilizatorul poate comuta între instrumentele disponibile.

Map canvas is implemented as `QgsMapCanvas` class in `qgis.gui` module. The implementation is based on the Qt Graphics View framework. This framework generally provides a surface and a view where custom graphics items are placed and user can interact with them. We will assume that you are familiar enough with Qt to understand the concepts of the graphics scene, view and items. If not, please make sure to read the overview of the framework.

Ori de câte ori harta a fost deplasată, mărită/micorată (sau alte aciuni care declanează o recitire), harta este randată iarăi în interiorul granielor curente. Straturile sunt transformate într-o imagine (folosind clasa `QgsMapRenderer`) iar acea imagine este afiată pe suport. Elementul grafic (în termeni ai cadrului de lucru Qt Graphics View) responsabil pentru a afiarea hării este `QgsMapCanvasMap`. Această clasă controlează, de asemenea, recitirea hării randate. În afară de acest element, care acionează ca fundal, pot exista mai multe **elemente ale suportului hării**. Elementele tipice suportului de hartă sunt benzile elastice (utilizate pentru măsurare, editare vectorială etc) sau marcajele nodurilor. Elementele suportului sunt de obicei utilizate pentru a oferi un răspuns vizual pentru instrumentele hării, de exemplu, atunci când se creează un nou poligon, instrumentul corespunzător creează o bandă elastică de forma actuală a poligonului. Toate elementele suportului de hartă reprezintă subclase ale `QgsMapCanvasItem` care adaugă mai multe funcionalităi obiectelor de bază `QGraphicsItem`.

Pentru a rezuma, arhitectura suportului pentru hartă constă în trei concepte:

- suportul de hartă — pentru vizualizarea hării
- elementele — elemente suplimentare care pot fi afiate în suportul hării
- instrumentele hării — pentru interaciunea cu suportul hării

7.1 Încapsularea suportului de hartă

Canevasul hării este un widget ca orice alt widget Qt, aa că utilizarea este la fel de simplă ca i crearea i afiarea lui

```
canvas = QgsMapCanvas()
canvas.show()
```

Acest cod va produce o fereastră de sine stătătoare cu suport pentru hartă. Ea poate fi, de asemenea, încorporată într-un widget sau într-o fereastră deja existente. Atunci când se utilizează fiiere .ui i Qt Designer, punei un `QWidget` pe formă pe care, ulterior, o vei promova la o nouă clasă: setai `QgsMapCanvas` ca nume de clasă i stabilii `qgis.gui` ca fiier antet. Utilitarul ‘‘pyuic4 ‘‘ va avea grijă de ea. Acesta este un mod foarte convenabil de încapsulare a suportului. Cealaltă posibilitate este de a scrie manual codul pentru a construi suportul hării i alte widget-uri (în calitate de copii ai ferestrei principale sau de dialog), apoi creai o aezare în pagină.

În mod implicit, canevasul hării are un fundal negru i nu utilizează anti-zimare. Pentru a seta fundalul alb i pentru a permite anti-zimare pentru o redare mai bună

```
canvas.setCanvasColor(Qt.white)
canvas.enableAntiAliasing(True)
```

(În cazul în care vă întrebai, `Qt` vine de la modulul `PyQt4.QtCore` iar `Qt.white` este una dintre instanele `QColor` predefinite.)

Now it is time to add some map layers. We will first open a layer and add it to the map layer registry. Then we will set the canvas extent and set the list of layers for canvas

```
layer = QgsVectorLayer(path, name, provider)
if not layer.isValid():
  raise IOError, "Failed to open the layer"

# add layer to the registry
QgsMapLayerRegistry.instance().addMapLayer(layer)

# set extent to the extent of our layer
canvas.setExtent(layer.extent())

# set the map canvas layer set
canvas.setLayerSet( [ QgsMapCanvasLayer(layer) ] )
```

După executarea acestor comenzi, suportul ar trebui să arate stratul pe care le-ai încărcat.

7.2 Folosirea instrumentelor în suportul de hartă

The following example constructs a window that contains a map canvas and basic map tools for map panning and zooming. Actions are created for activation of each tool: panning is done with `QgsMapToolPan`, zooming in/out with a pair of `QgsMapToolZoom` instances. The actions are set as checkable and later assigned to the tools to allow automatic handling of checked/unchecked state of the actions – when a map tool gets activated, its action is marked as selected and the action of the previous map tool is deselected. The map tools are activated using `setMapTool()` method.

```
from qgis.gui import *
from PyQt4.QtGui import QAction, QMainWindow
from PyQt4.QtCore import SIGNAL, Qt, QString

class MyWnd(QMainWindow):
  def __init__(self, layer):
    QMainWindow.__init__(self)

    self.canvas = QgsMapCanvas()
    self.canvas.setCanvasColor(Qt.white)

    self.canvas.setExtent(layer.extent())
    self.canvas.setLayerSet( [ QgsMapCanvasLayer(layer) ] )

    self.setCentralWidget(self.canvas)

    actionZoomIn = QAction(QString("Zoom in"), self)
    actionZoomOut = QAction(QString("Zoom out"), self)
    actionPan = QAction(QString("Pan"), self)

    actionZoomIn.setCheckable(True)
    actionZoomOut.setCheckable(True)
    actionPan.setCheckable(True)

    self.connect(actionZoomIn, SIGNAL("triggered()"), self.zoomIn)
```

```
    self.connect(actionZoomOut, SIGNAL("triggered()"), self.zoomOut)
    self.connect(actionPan, SIGNAL("triggered()"), self.pan)

    self.toolbar = self.addToolBar("Canvas actions")
    self.toolbar.addAction(actionZoomIn)
    self.toolbar.addAction(actionZoomOut)
    self.toolbar.addAction(actionPan)

    # create the map tools
    self.toolPan = QgsMapToolPan(self.canvas)
    self.toolPan.setAction(actionPan)
    self.toolZoomIn = QgsMapToolZoom(self.canvas, False) # false = in
    self.toolZoomIn.setAction(actionZoomIn)
    self.toolZoomOut = QgsMapToolZoom(self.canvas, True) # true = out
    self.toolZoomOut.setAction(actionZoomOut)

    self.pan()

  def zoomIn(self):
    self.canvas.setMapTool(self.toolZoomIn)

  def zoomOut(self):
    self.canvas.setMapTool(self.toolZoomOut)

  def pan(self):
    self.canvas.setMapTool(self.toolPan)
```

You can put the above code to a file, e.g. `mywnd.py` and try it out in Python console within QGIS. This code will put the currently selected layer into newly created canvas

```
import mywnd
w = mywnd.MyWnd(qgis.utils.iface.activeLayer())
w.show()
```

Just make sure that the `mywnd.py` file is located within Python search path (`sys.path`). If it isn't, you can simply add it: `sys.path.insert(0, '/my/path')` — otherwise the import statement will fail, not finding the module.

7.3 Benzile elastice i marcajele nodurilor

To show some additional data on top of the map in canvas, use map canvas items. It is possible to create custom canvas item classes (covered below), however there are two useful canvas item classes for convenience: `QgsRubberBand` for drawing polylines or polygons, and `QgsVertexMarker` for drawing points. They both work with map coordinates, so the shape is moved/scaled automatically when the canvas is being panned or zoomed.

Pentru a afia o polilinie

```
r = QgsRubberBand(canvas, False)  # False = not a polygon
points = [ QgsPoint(-1,-1), QgsPoint(0,1), QgsPoint(1,-1) ]
r.setToGeometry(QgsGeometry.fromPolyline(points), None)
```

Pentru a afia un poligon

```
r = QgsRubberBand(canvas, True)  # True = a polygon
points = [ [ QgsPoint(-1,-1), QgsPoint(0,1), QgsPoint(1,-1) ] ]
r.setToGeometry(QgsGeometry.fromPolygon(points), None)
```

Reinei că punctele pentru poligon nu reprezintă o simplă listă: în fapt, aceasta este o listă de inele coninând inele liniare ale poligonului: primul inel reprezintă grania exterioară, în plus (opional) inelele corespund găurilor din poligon.

Benzile elastice acceptă unele personalizări, i anume schimbarea culorii i a lăimii liniei

```
r.setColor(QColor(0,0,255))
r.setWidth(3)
```

Elementele suportului sunt legate de suportul hării. Pentru a le ascunde temporar (i a le arăta din nou, folosii combinaia `hide()` i `show()`. Pentru a elimina complet elementul, trebuie să-l eliminăm de pe scena canevasului

```
canvas.scene().removeItem(r)
```

(În C + + este posibilă tergerea doar a elementului, însă în Python `del r` ar terge doar referina iar obiectul va exista în continuare, acesta fiind deinut de suport)

Banda elastică poate fi de asemenea utilizată pentru desenarea de puncte, însă, clasa `QgsVertexMarker` este mai potrivită pentru aceasta (`QgsRubberBand` ar trasa doar un dreptunghi în jurul punctului dorit). Cum să utilizai simbolul nodului

```
m = QgsVertexMarker(canvas)
m.setCenter(QgsPoint(0,0))
```

În acest mod se va desena o cruciuliă roie pe poziia [0,0]. Este posibilă personalizarea tipului pictogramei, dimensiunea, culoarea i lăimea instrumentului de desenare

```
m.setColor(QColor(0,255,0))
m.setIconSize(5)
m.setIconType(QgsVertexMarker.ICON_BOX) # or ICON_CROSS, ICON_X
m.setPenWidth(3)
```

Pentru ascunderea temporară a markerilor vertex i pentru eliminarea lor de pe suport, acelai lucru este valabil i pentru benzile elastice.

7.4 Dezvoltarea instrumentelor personalizate pentru suportul de hartă

Putei crea propriile instrumente, pentru a implementa un comportament personalizat pentru aciunile executate de către utilizatori pe canevas.

Instrumentele de hartă ar trebui să motenească clasa `QgsMapTool` sau orice altă clasă derivată, i să fie selectate ca instrumente active pe suport, folosindu-se metoda `setMapTool()`, aa cum am văzut deja.

Iată un exemplu de instrument pentru hartă, care permite definirea unei limite dreptunghiulare, făcând clic i trăgând cursorul mouse-ului pe canevas. După ce este definit dreptunghiul, coordonatele sale sunt afiate în consolă. Se utilizează elementele benzii elastice descrise mai înainte, pentru a arăta dreptunghiul selectat, aa cum a fost definit.

```
class RectangleMapTool(QgsMapToolEmitPoint):
  def __init__(self, canvas):
      self.canvas = canvas
      QgsMapToolEmitPoint.__init__(self, self.canvas)
      self.rubberBand = QgsRubberBand(self.canvas, QGis.Polygon)
      self.rubberBand.setColor(Qt.red)
      self.rubberBand.setWidth(1)
      self.reset()

  def reset(self):
      self.startPoint = self.endPoint = None
      self.isEmittingPoint = False
      self.rubberBand.reset(QGis.Polygon)

  def canvasPressEvent(self, e):
      self.startPoint = self.toMapCoordinates(e.pos())
      self.endPoint = self.startPoint
      self.isEmittingPoint = True
```

```
        self.showRect(self.startPoint, self.endPoint)

    def canvasReleaseEvent(self, e):
        self.isEmittingPoint = False
        r = self.rectangle()
        if r is not None:
          print "Rectangle:", r.xMin(), r.yMin(), r.xMax(), r.yMax()

    def canvasMoveEvent(self, e):
        if not self.isEmittingPoint:
          return

        self.endPoint = self.toMapCoordinates( e.pos() )
        self.showRect(self.startPoint, self.endPoint)

    def showRect(self, startPoint, endPoint):
        self.rubberBand.reset(QGis.Polygon)
        if startPoint.x() == endPoint.x() or startPoint.y() == endPoint.y():
          return

        point1 = QgsPoint(startPoint.x(), startPoint.y())
        point2 = QgsPoint(startPoint.x(), endPoint.y())
        point3 = QgsPoint(endPoint.x(), endPoint.y())
        point4 = QgsPoint(endPoint.x(), startPoint.y())

        self.rubberBand.addPoint(point1, False)
        self.rubberBand.addPoint(point2, False)
        self.rubberBand.addPoint(point3, False)
        self.rubberBand.addPoint(point4, True)    # true to update canvas
        self.rubberBand.show()

    def rectangle(self):
        if self.startPoint is None or self.endPoint is None:
          return None
        elif self.startPoint.x() == self.endPoint.x() or self.startPoint.y() == self.endPoint.y():
          return None

        return QgsRectangle(self.startPoint, self.endPoint)

    def deactivate(self):
        QgsMapTool.deactivate(self)
        self.emit(SIGNAL("deactivated()"))
```

7.5 Dezvoltarea elementelor personalizate pentru suportul de hartă

DE EFECTUAT: how to create a map canvas item

Randarea hărilor i imprimarea

Există, în general, două abordări atunci când datele de intrare ar trebui să fie randate într-o hartă: fie o modalitate rapidă, folosind `QgsMapRenderer`, fie producerea unei ieiri mai rafinate, prin compunerea hării cu ajutorul clasei `QgsComposition`.

8.1 Randarea simplă

Randai mai multe straturi, folosind `QgsMapRenderer` — creai destinaia dispozitivului de colorare (`QImage`, `QPainter` etc), setai stratul, limitele sale, dimensiunea de ieire i efectuai randarea

```
# create image
img = QImage(QSize(800,600), QImage.Format_ARGB32_Premultiplied)

# set image's background color
color = QColor(255,255,255)
img.fill(color.rgb())

# create painter
p = QPainter()
p.begin(img)
p.setRenderHint(QPainter.Antialiasing)

render = QgsMapRenderer()

# set layer set
lst = [ layer.getLayerID() ]  # add ID of every layer
render.setLayerSet(lst)

# set extent
rect = QgsRect(render.fullExtent())
rect.scale(1.1)
render.setExtent(rect)

# set output size
render.setOutputSize(img.size(), img.logicalDpiX())

# do the rendering
render.render(p)

p.end()

# save image
img.save("render.png","png")
```

8.2 Generarea folosind Compozitorul de hări

Compozitorul de hări reprezintă un instrument foarte util în cazul în care dorii să elaborai ceva mai sofisticat decât simpla randare de mai sus. Utilizănd Constructorului este posibilă crearea unor machete complexe de hări, coninând extrase de hartă, etichete, legendă, tabele i alte elemente care sunt de obicei prezente pe hările tipărite. Machetele pot fi apoi exportate în format PDF, ca imagini raster sau pot fi transmise direct la o imprimantă.

Compozitorul constă într-o serie de clase. Toate acestea fac parte din biblioteca de bază. Aplicaia QGIS are un GUI convenabil pentru plasarea elementelor, dei nu face parte din biblioteca GUI. Dacă nu suntei familiarizai cu cadrul de lucru Qt Graphics View, atunci vă încurajăm să verificai documentaia acum, deoarece compozitorul este bazat pe el.

Clasa centrală a Compozitorului este `QgsComposition`, care este derivată din `QGraphicsScene`. Să creăm una

```
mapRenderer = iface.mapCanvas().mapRenderer()
c = QgsComposition(mapRenderer)
c.setPlotStyle(QgsComposition.Print)
```

Reinei: compoziia este o instană a `QgsMapRenderer`. În cod, ne ateptăm să rulăm în interiorul aplicaiei QGIS i, astfel, să folosim render-ul suportului de hartă. Compoziia utilizează diveri parametri ai render-ului, cei mai importani fiind setul implicit de straturi de hartă i graniele curente. Atunci când utilizai compozitorul într-o aplicaie independentă, vă putei crea propria dvs. instană de render de hări, în acelai mod cum s-a arătat în seciunea de mai sus, i să-l transmitei compoziiei.

Este posibilă adăugarea diferitelor elemente (hartă, etichete, ...) în compoziie — aceste elemente trebuie să fie descendeni ai clasei `QgsComposerItem`. Elementele suportate în prezent sunt:

- map — this item tells the libraries where to put the map itself. Here we create a map and stretch it over the whole paper size

  ```
  x, y = 0, 0
  w, h = c.paperWidth(), c.paperHeight()
  composerMap = QgsComposerMap(c, x,y,w,h)
  c.addItem(composerMap)
  ```

- label — allows displaying labels. It is possible to modify its font, color, alignment and margin

  ```
  composerLabel = QgsComposerLabel(c)
  composerLabel.setText("Hello world")
  composerLabel.adjustSizeToText()
  c.addItem(composerLabel)
  ```

- legend

  ```
  legend = QgsComposerLegend(c)
  legend.model().setLayerSet(mapRenderer.layerSet())
  c.addItem(legend)
  ```

- scale bar

  ```
  item = QgsComposerScaleBar(c)
  item.setStyle('Numeric') # optionally modify the style
  item.setComposerMap(composerMap)
  item.applyDefaultSize()
  c.addItem(item)
  ```

- săgeată
- imagine
- formă
- tabelă

În mod implicit, elementele compozitorului nou creat au poziia zero (colul din stânga sus a paginii) i dimensiunea zero. Poziia i dimensiunea sunt măsurate întotdeauna în milimetri

```
# set label 1cm from the top and 2cm from the left of the page
composerLabel.setItemPosition(20,10)
# set both label's position and size (width 10cm, height 3cm)
composerLabel.setItemPosition(20,10, 100, 30)
```

În jurul fiecărui element este desenat, în mod implicit, un cadru. Astfel se elimină cadrul

```
composerLabel.setFrame(False)
```

Pe lângă crearea manuală a elementele compozitorului, QGIS are suport pentru abloane, care sunt, în esenă, compoziii cu toate elementele lor salvate într-un fiier .qpt (cu sintaxă XML). Din păcate, această funcionalitate nu este încă disponibilă în API.

Odată ce compoziia este gata (elementele compozitorului au fost create i adăugate la compoziie), putem trece la producerea unui raster i/sau a unei ieiri vectoriale.

Setările de ieire implicite pentru compoziie sunt pentru o pagină A4 i o rezoluie de 300 DPI. Le putei modifica, atunci când este necesar. Dimensiunea hârtiei este specificată în milimetri

```
c.setPaperSize(width, height)
c.setPrintResolution(dpi)
```

8.2.1 Ieire ca imagine raster

Următorul fragment de cod arată cum se randează o compoziie într-o imagine raster

```
dpi = c.printResolution()
dpmm = dpi / 25.4
width = int(dpmm * c.paperWidth())
height = int(dpmm * c.paperHeight())

# create output image and initialize it
image = QImage(QSize(width, height), QImage.Format_ARGB32)
image.setDotsPerMeterX(dpmm * 1000)
image.setDotsPerMeterY(dpmm * 1000)
image.fill(0)

# render the composition
imagePainter = QPainter(image)
sourceArea = QRectF(0, 0, c.paperWidth(), c.paperHeight())
targetArea = QRectF(0, 0, width, height)
c.render(imagePainter, targetArea, sourceArea)
imagePainter.end()

image.save("out.png", "png")
```

8.2.2 Ieire în format PDF

Următorul fragment de cod randează o compoziie într-un fiier PDF

```
printer = QPrinter()
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName("out.pdf")
printer.setPaperSize(QSizeF(c.paperWidth(), c.paperHeight()), QPrinter.Millimeter)
printer.setFullPage(True)
printer.setColorMode(QPrinter.Color)
printer.setResolution(c.printResolution())

pdfPainter = QPainter(printer)
```

```
paperRectMM = printer.pageRect(QPrinter.Millimeter)
paperRectPixel = printer.pageRect(QPrinter.DevicePixel)
c.render(pdfPainter, paperRectPixel, paperRectMM)
pdfPainter.end()
```

CAPITOLUL 9

Expresii, filtrarea i calculul valorilor

QGIS are un oarecare suport pentru parsarea expresiilor, cum ar fi SQL. Doar un mic subset al sintaxei SQL este suportat. Expresiile pot fi evaluate fie ca predicate booleene (returnând Adevărat sau Fals) sau ca funcii (care întorc o valoare scalară).

Trei tipuri de bază sunt acceptate:

- — număr atât numere întregi cât i numere zecimale, de exemplu, `123`, `3.14`
- ir — acesta trebuie să fie cuprins între ghilimele simple: `'hello world'`
- referină către coloană — atunci când se evaluează, referina este substituită cu valoarea reală a câmpului. Numele nu sunt protejate.

Următoarele operaiuni sunt disponibile:

- operatori aritmetici: `+`, `-`, `*`, `/`, `^`
- paranteze: pentru forarea priorităii operatorului: `(1 + 1) * 3`
- plus i minus unari: `-12`, `+5`
- funcii matematice: `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- funcii geometrice: `$area`, `$length`
- funcii de conversie: `to int`,`to real`,`to string`

i următoarele predicate sunt suportate:

- comparaie: =, !=, >, >=, <, <=
- potrivirea paternurilor: `LIKE` (folosind % i _), ~ (expresii regulate)
- predicate logice: `AND`, `OR`, `NOT`
- verificarea valorii NULL: `IS NULL`, `IS NOT NULL`

Exemple de predicate:

- `1 + 2 = 3`
- `sin(angle) > 0`
- `'Hello' LIKE 'He%'`
- `(x > 10 AND y > 10) OR z = 0`

Exemple de expresii scalare:

- `2 ^ 10`
- `sqrt(val)`
- `$length + 1`

9.1 Parsarea expresiilor

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> exp.hasParserError()
False
>>> exp = QgsExpression('1 + 1 = ')
>>> exp.hasParserError()
True
>>> exp.parserErrorString()
PyQt4.QtCore.QString(u'syntax error, unexpected $end')
```

9.2 Evaluarea expresiilor

9.2.1 Expresii de bază

```
>>> exp = QgsExpression('1 + 1 = 2')
>>> value = exp.evaluate()
>>> value
1
```

9.2.2 Expresii cu entităi

Următorul exemplu va evalua expresia dată faă de o entitate. “Column” este numele câmpului din strat.

```
>>> exp = QgsExpression('Column = 99')
>>> value = exp.evaluate(feature, layer.pendingFields())
>>> bool(value)
True
```

De asemenea, putei folosi `QgsExpression.prepare()`, dacă trebuie să verificai mai mult de o entitate. Utilizarea `QgsExpression.prepare()` va spori viteza evaluării.

```
>>> exp = QgsExpression('Column = 99')
>>> exp.prepare(layer.pendingFields())
>>> value = exp.evaluate(feature)
>>> bool(value)
True
```

9.2.3 Tratarea erorilor

```
exp = QgsExpression("1 + 1 = 2 ")
if exp.hasParserError():
  raise Expection(exp.parserErrorString())

value = exp.evaluate()
if exp.hasEvalError():
  raise ValueError(exp.evalErrorString())

print value
```

9.3 Exemple

Următorul exemplu poate fi folosit pentru a filtra un strat i pentru a întoarce orice entitate care se potrivete unui predicat.

```
def where(layer, exp):
  print "Where"
  exp = QgsExpression(exp)
  if exp.hasParserError():
    raise Expection(exp.parserErrorString())
  exp.prepare(layer.pendingFields())
  for feature in layer.getFeatures():
    value = exp.evaluate(feature)
    if exp.hasEvalError():
      raise ValueError(exp.evalErrorString())
    if bool(value):
      yield feature

layer = qgis.utils.iface.activeLayer()
for f in where(layer, 'Test > 1.0'):
  print f + " Matches expression"
```

CAPITOLUL 10

Citirea i stocarea setărilor

De multe ori, pentru un plugin, este utilă salvarea unor variabile, astfel încât utilizatorul să nu trebuiască să le reintroducă sau să le reselecteze, la fiecare rulare a plugin-ului.

These variables can be saved a retrieved with help of Qt and QGIS API. For each variable, you should pick a key that will be used to access the variable — for user's favourite color you could use key "favourite_color" or any other meaningful string. It is recommended to give some structure to naming of keys.

Putem face diferena între mai multe tipuri de setări:

- **global settings** — they are bound to the user at particular machine. QGIS itself stores a lot of global settings, for example, main window size or default snapping tolerance. This functionality is provided directly by Qt framework by the means of QSettings class. By default, this class stores settings in system's "native" way of storing settings, that is — registry (on Windows), .plist file (on Mac OS X) or .ini file (on Unix). The QSettings documentation is comprehensive, so we will provide just a simple example

```
def store():
  s = QSettings()
  s.setValue("myplugin/mytext", "hello world")
  s.setValue("myplugin/myint",  10)
  s.setValue("myplugin/myreal", 3.14)

def read():
  s = QSettings()
  mytext = s.value("myplugin/mytext", "default text")
  myint  = s.value("myplugin/myint", 123)
  myreal = s.value("myplugin/myreal", 2.71)
```

  Al doilea parametru al metodei `value()` este opional i specifică valoarea implicită, dacă nu există nici o valoare anterioară stabilită pentru setare.

- **setările proiectului** — variază între diferite proiecte i, prin urmare, ele sunt conectate cu un fiier de proiect. Culoarea de fundal a suportului hării sau sistemul de coordonate de referină (CRS), de exemplu — fundal alb i WGS84 ar putea fi potrivite pentru un anumit proiect, în timp ce fondul galben i proiecia UTM ar putea fi mai bune pentru altul. În continuare este dat un exemplu de utilizare

```
proj = QgsProject.instance()

# store values
proj.writeEntry("myplugin", "mytext", "hello world")
proj.writeEntry("myplugin", "myint", 10)
proj.writeEntry("myplugin", "mydouble", 0.01)
proj.writeEntry("myplugin", "mybool", True)

# read values
mytext = proj.readEntry("myplugin", "mytext", "default text")[0]
myint = proj.readNumEntry("myplugin", "myint", 123)[0]
```

  După cum putei vedea, metoda `writeEntry()` este folosită pentru toate tipurile de date, dar există mai

multe metode pentru a seta înapoi setarea, iar cea corespunzătoare trebuie să fie selectată pentru fiecare tip de date.

- **setările stratului hării** — aceste setări sunt legate de o anumită instană a unui strat de hartă cu un proiect. Acestea *nu* sunt conectate cu sursa de date a stratului, aa că dacă vei crea două instane ale unui strat de hartă dintr-un fiier shape, ele nu vor partaja setările. Setările sunt stocate în fiierul proiectului, astfel încât, în cazul în care utilizatorul deschide iarăi proiectul, setările legate de strat vor fi din nou acolo. Această funcionalitate a fost adăugată în QGIS v1.4. API-ul este similar cu QSettings — luând i returnând instane QVariant

```
# save a value
layer.setCustomProperty("mytext", "hello world")

# read the value again
mytext = layer.customProperty("mytext", "default text")
```

Comunicarea cu utilizatorul

Această seciune prezintă câteva metode i elemente care ar trebui să fie utilizate pentru a comunica cu utilizatorul, în scopul meninerii coerenei interfaei cu utilizatorul.

11.1 Afiarea mesajelor. Clasa `QgsMessageBar`

Folosirea casetelor de mesaje poate fi o idee rea, din punctul de vedere al experienei utilizatorului. Pentru a arăta o mică linie de informaii sau un mesaj de avertizare/eroare, bara QGIS de mesaje este, de obicei, o opiune mai bună.

Folosind referina către obiectul interfeei QGIS, putei afia un text în bara de mesaje, cu ajutorul următorului cod

```
iface.messageBar().pushMessage("Error", "I'm sorry Dave, I'm afraid I can't do that", level=QgsMe
```

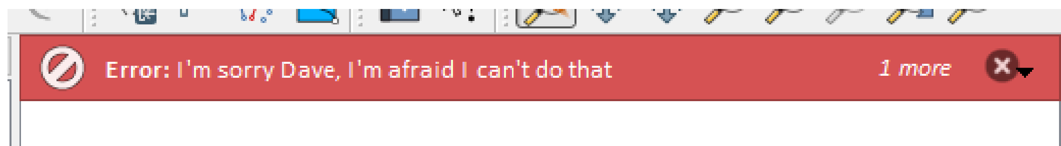

Figura 11.1: Bara de mesaje a QGIS

Putei seta o durată, pentru afiarea pentru o perioadă limitată de timp

```
iface.messageBar().pushMessage("Error", ""Ooops, the plugin is not working as it should", level=Q
```

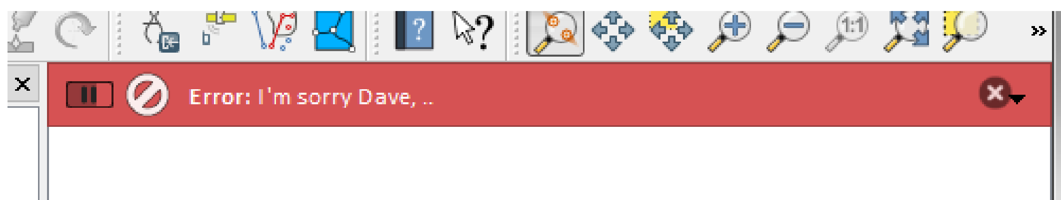

Figura 11.2: Bara de mesaje a QGIS, cu cronometru

Exemplele de mai sus arată o bară de eroare, dar parametrul `level` poate fi utilizat pentru a crea mesaje de avertizare sau informative, folosind constantele `QgsMessageBar.WARNING` i respectiv `QgsMessageBar.INFO`.

Widget-urile pot fi adăugate la bara de mesaje, cum ar fi, de exemplu, un buton pentru afiarea mai multor informaii

```
def showError():
    pass

widget = iface.messageBar().createMessage("Missing Layers", "Show Me")
```

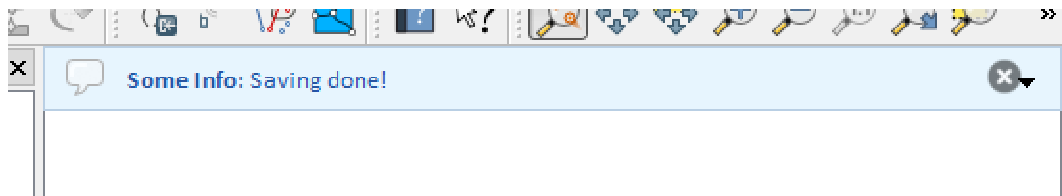

Figura 11.3: Bara de mesaje a QGIS (info)

```
button = QPushButton(widget)
button.setText("Show Me")
button.pressed.connect(showError)
widget.layout().addWidget(button)
iface.messageBar().pushWidget(widget, QgsMessageBar.WARNING)
```

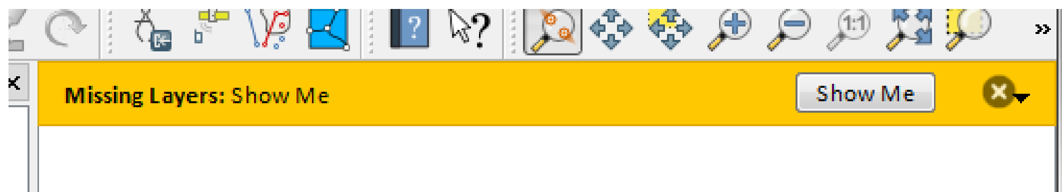

Figura 11.4: Bara de mesaje a QGIS, cu un buton

Putei utiliza o bară de mesaje chiar i în propria fereastră de dialog, în loc să apelai la o casetă de text, sau să arătai mesajul în fereastra principală a QGIS

```
class MyDialog(QDialog):
    def __init__(self):
        QDialog.__init__(self)
        self.bar = QgsMessageBar()
        self.bar.setSizePolicy( QSizePolicy.Minimum, QSizePolicy.Fixed )
        self.setLayout(QGridLayout())
        self.layout().setContentsMargins(0,0,0,0)
        self.buttonbox = QDialogButtonBox(QDialogButtonBox.Ok)
        self.buttonbox.accepted.connect(self.run)
        self.layout().addWidget(self.buttonbox , 0,0,2,1)
        self.layout().addWidget(self.bar, 0,0,1,1)

    def run(self):
        self.bar.pushMessage("Hello", "World", level=QgsMessageBar.INFO)
```

11.2 Afiarea progresului

Barele de progres pot fi, de asemenea, incluse în bara de mesaje QGIS, din moment ce, aa cum am văzut, aceasta acceptă widget-uri. Iată un exemplu pe care îl putei încerca în consolă.

```
import time
from PyQt4.QtGui import QProgressBar
from PyQt4.QtCore import *
progressMessageBar = iface.messageBar().createMessage("Doing something boring...")
progress = QProgressBar()
progress.setMaximum(10)
progress.setAlignment(Qt.AlignLeft|Qt.AlignVCenter)
progressMessageBar.layout().addWidget(progress)
iface.messageBar().pushWidget(progressMessageBar, iface.messageBar().INFO)
for i in range(10):
    time.sleep(1)
```

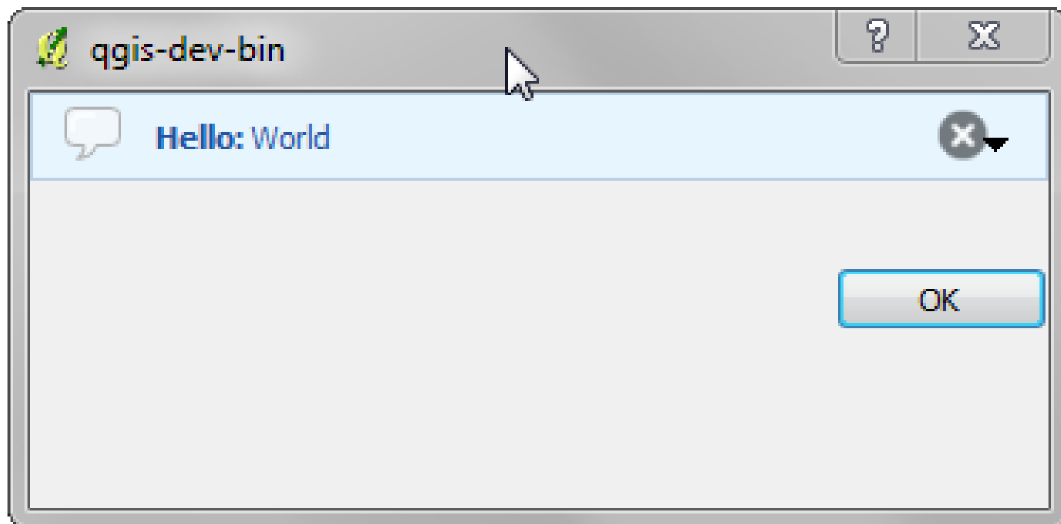

Figura 11.5: Bara de mesaje a QGIS, într-o fereastră de dialog

```
    progress.setValue(i + 1)
iface.messageBar().clearWidgets()
```

De asemenea, avei posibilitatea să utilizai bara de stare internă pentru a raporta progresul, ca în exemplul următor

```
count = layers.featureCount()
for i, feature in enumerate(features):
    #do something time-consuming here
    ...
    percent = i / float(count) * 100
    iface.mainWindow().statusBar().showMessage("Processed {} %".format(int(percent)))
iface.mainWindow().statusBar().clearMessage()
```

11.3 Jurnalizare

Putei utiliza sistemul de jurnalizare al QGIS, pentru a salva toate informaiile pe care dorii să le înregistrai, cu privire la execuia codului dvs.

```
QgsMessageLog.logMessage("Your plugin code has been executed correctly", QgsMessageLog.INFO)
QgsMessageLog.logMessage("Your plugin code might have some problems", QgsMessageLog.WARNING)
QgsMessageLog.logMessage("Your plugin code has crashed!", QgsMessageLog.CRITICAL)
```

CAPITOLUL 12

Dezvoltarea plugin-urilor Python

Este posibil să se creeze plugin-uri în limbajul de programare Python. În comparaie cu plugin-urile clasice scrise în C++ acestea ar trebui să fie mai uor de scris, de îneles, de meninut i de distribuit, din cauza naturii dinamice a limbajului Python.

Plugin-urile Python sunt listate, împreună cu plugin-urile C++, în managerul de plugin-uri QGIS. Ele sunt căutate în aceste căi:

- UNIX/Mac: `~/.qgis/python/plugins` and `(qgis_prefix)/share/qgis/python/plugins`
- Windows: `~/.qgis/python/plugins` and `(qgis_prefix)/python/plugins`

Home directory (denoted by above `~`) on Windows is usually something like `C:\Documents and Settings\(user)` (on Windows XP or earlier) or `C:\Users\(user)`. Since Quantum GIS is using Python 2.7, subdirectories of these paths have to contain an __init__.py file to be considered Python packages that can be imported as plugins.

Pai:

1. *Ideea*: Conturai-vă o idee despre ceea ce vrei să facei cu noul plugin QGIS. De ce-l facei? Ce problemă dorii să rezolve? Există deja un alt plugin pentru această problemă?
2. *Creare fiiere*: Se creează fiierele descrise în continuare. Un punct de plecare (__`init.py`__). Completai |*Metadatele plugin-ului* (`metadata.txt`). Un corp python principal al plugin-ului (`mainplugin.py`). O formă în QT-Designer (`form.ui`), cu al său `resources.qrc`.
3. *Scriei codul*: Scriei codul în interiorul `mainplugin.py`
4. *Testul*: Închidei i re-deschidei QGIS, apoi importai-l din nou. Verificai dacă totul este în regulă.
5. *Publicare*: Se publică plugin-ul în depozitul QGIS sau vă facei propriul depozit ca un "arsenal" de "arme GIS" personale

12.1 Scrierea unui plugin

De la introducerea plugin-urilor Python în QGIS, a apărut un număr de plugin-uri - pe pagina wiki a Depozitelor de Plugin-uri putei găsi unele dintre ele, le putei utiliza sursa pentru a afla mai multe despre programarea în PyQGIS sau să aflai dacă nu cumva duplicai efortul de dezvoltare. Echipa QGIS menine, de asemenea, un *Depozitul oficial al plugin-urilor python*. Suntei gata de a crea un plugin, dar nu avei nici o idee despre cum ai putea începe? În pagina wiki cu idei de plugin-uri Python sunt listate dorinele comunităii!

12.1.1 Fiierele Plugin-ului

Iată structura de directoare a exemplului nostru de plugin

```
PYTHON_PLUGINS_PATH/
  MyPlugin/
    __init__.py    --> *required*
    mainPlugin.py  --> *required*
    metadata.txt   --> *required*
    resources.qrc  --> *likely useful*
    resources.py   --> *compiled version, likely useful*
    form.ui        --> *likely useful*
    form.py        --> *compiled version, likely useful*
```

Care este semnificaia fiierelor:

- `__init__.py` = Punctul de plecare al plugin-ului. Acesta trebuie să aibă metoda `classFactory()` i poate avea orice alt cod de iniializare.
- `mainPlugin.py` = Principalul codul lucrativ al plugin-ului. Conine toate informaiile cu privire la aciunile plugin-ului i ale codului principal.
- `resources.qrc` = Documentul .xml creat de Qt Designer. Conine căi relative la resursele formelor.
- `resources.py` = Traducerea fiierului .qrc descris mai sus.
- `form.ui` = GUI-ul creat de Qt Designer.
- `form.py` = Traducerea form.ui descris mai sus.
- `metadata.txt` = Necesar pentru QGIS >= 1.8.0. Conine informaii generale, versiunea, numele i alte metadate utilizate de către site-ul de plugin-uri i de către infrastructura plugin-ului. Începând cu QGIS 2.0 metadatele din `__init__.py` nu mai sunt acceptate, iar `metadata.txt` este necesar.

Aici este o modalitate on-line, automată, de creare a fiierelor de bază (carcase) pentru un plugin tipic QGIS Python.

De asemenea, există un plugin QGIS numit Plugin Builder care creează un ablon de plugin QGIS i nu are nevoie de conexiune la internet. Aceasta este opiunea recomandată, atât timp cât produce surse compatibile 2.0.

Warning: Dacă avei de gând să încărcai plugin-ul la *Depozitul oficial al plugin-urilor python* trebuie să verificai că plugin-ul urmează anumite reguli suplimentare, necesare pentru *Validare* plugin

12.2 Coninutul Plugin-ului

Aici putei găsi informaii i exemple despre ceea ce să adăugai în fiecare dintre fiierele din structura de fiiere descrisă mai sus.

12.2.1 |Metadatele plugin-ului

În primul rând, managerul de plugin-uri are nevoie de preluarea câtorva informaii de bază despre plugin, cum ar fi numele, descrierea etc. Fiierul `metadata.txt` este locul potrivit pentru a reine această informaie.

Important: Toate metadatele trebuie să fie în codificarea UTF-8.

Numele metadatei	Obligatoriu	Note
nume	True	un ir scurt coninând numele pluginului
qgisMinimumVersion	True	notaie cu punct a versiunii minime QGIS
qgisMaximumVersion	False	notaie cu punct a versiunii maxime QGIS
descriere	True	scurt text care descrie plugin-ul, HTML nefiind permis
despre	False	text mai lung care descrie plugin-ul în detalii, HTML nefiind permis
versiune	True	scurt ir cu versiunea notată cu punct
autor	True	nume autor
email	True	e-mail-ul autorului, *nu* va fi afiat pe site-ul web
jurnalul schimbărilor	False	ir, poate fi pe mai multe linii, HTML nefiind permis
experimental	False	semnalizator boolean, *True* sau *False*
învechit	False	semnalizator boolean, *True* sau *False*, se aplică întregului plugin i nu doar la versiunea încărcată
etichete	False	o listă de valori separate prin virgulă, spaiile fiind permise în interiorul etichetelor individuale
pagina de casă	False	o adresă URL validă indicând spre pagina plugin-ului dvs.
depozit	False	o adresă URL validă pentru depozitul de cod sursă
tracker	False	o adresă validă pentru bilete i rapoartare de erori
pictogramă	False	un nume de fiier sau o cale relativă (relativă la directorul de bază al pachetului comprimat al plugin-ului)
categorie	False	una din valorile *Raster*, *Vector*, *Bază de date* i *Web*

În mod implicit, plugin-urile sunt plasate în meniul *Plugin-uri* (vom vedea în seciunea următoare cum se poate adăuga o intrare de meniu pentru plugin-ul dvs.), dar ele pot fi, de asemenea, plasate i în meniurile *Raster*, *Vector*, *Database* i *Web*.

Există o "categorie" de intrare de metadate corespunzătoare pentru a preciza că, astfel, plugin-ul poate fi clasificat în consecină. Această intrare este folosită ca indiciu pentru utilizatori i le spune unde (în care meniu), poate fi găsit plugin-ul. Valorile permise pentru "categorie" sunt: Vector, Raster, Baza de date sau Web. De exemplu, dacă plugin-ul va fi disponibil din meniul *Raster*, adăugai următoarele în `metadata.txt`

```
category=Raster
```

Note: În cazul în care valoarea *qgisMaximumVersion* este vidă, ea va fi setată automat la versiunea majoră plus *0.99* încărcată în depozitul *Depozitul oficial al plugin-urilor python*.

Un exemplu pentru acest metadata.txt

```
; the next section is mandatory

[general]
name=HelloWorld
email=me@example.com
author=Just Me
qgisMinimumVersion=2.0
description=This is an example plugin for greeting the world.
    Multiline is allowed:
    lines starting with spaces belong to the same
    field, in this case to the "description" field.
    HTML formatting is not allowed.
about=This paragraph can contain a detailed description
    of the plugin. Multiline is allowed, HTML is not.
version=version 1.2
; end of mandatory metadata

; start of optional metadata
category=Raster
```

```
changelog=The changelog lists the plugin versions
    and their changes as in the example below:
    1.0 - First stable release
    0.9 - All features implemented
    0.8 - First testing release

; Tags are in comma separated value format, spaces are allowed within the
; tag name.
; Tags should be in English language. Please also check for existing tags and
; synonyms before creating a new one.
tags=wkt,raster,hello world

; these metadata can be empty, they will eventually become mandatory.
homepage=http://www.itopen.it
tracker=http://bugs.itopen.it
repository=http://www.itopen.it/repo
icon=icon.png

; experimental flag (applies to the single version)
experimental=True

; deprecated flag (applies to the whole plugin and not only to the uploaded version)
deprecated=False

; if empty, it will be automatically set to major version + .99
qgisMaximumVersion=2.0
```

12.2.2 __init__.py

This file is required by Python's import system. Also, Quantum GIS requires that this file contains a `classFactory()` function, which is called when the plugin gets loaded to QGIS. It receives reference to instance of `QgisInterface` and must return instance of your plugin's class from the `mainplugin.py` — in our case it's called `TestPlugin` (see below). This is how `__init__.py` should look like

```
def classFactory(iface):
  from mainPlugin import TestPlugin
  return TestPlugin(iface)

## any other initialisation needed
```

12.2.3 mainPlugin.py

Aici este locul în care se întâmplă magia, i iată rezultatul acesteia: (de exemplu `mainPlugin.py`)

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
from qgis.core import *

# initialize Qt resources from file resources.py
import resources

class TestPlugin:

  def __init__(self, iface):
    # save reference to the QGIS interface
    self.iface = iface

  def initGui(self):
    # create action that will start plugin configuration
    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWin
```

```
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)

    # add toolbar button and menu item
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

    # connect to signal renderComplete which is emitted when canvas
    # rendering is done
    QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest

  def unload(self):
    # remove the plugin menu item and icon
    self.iface.removePluginMenu("&Test plugins",self.action)
    self.iface.removeToolBarIcon(self.action)

    # disconnect form signal of the canvas
    QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderT

  def run(self):
    # create and show a configuration dialog or something similar
    print "TestPlugin: run called!"

  def renderTest(self, painter):
    # use painter for drawing to map canvas
    print "TestPlugin: renderTest called!"
```

Singurele funcii care trebuie să existe în fiierul sursă al plugin-ului principal (de exemplu `mainPlugin.py`) sunt:

- `__init__` –> which gives access to Quantum GIS' interface
- `initGui()` –> apelat atunci când plugin-ul este încărcat
- `unload()` –> apelat atunci când plugin-ul este descărcat

Putei vedea că în exemplul de mai sus se folosete `addPluginToMenu()`. Aceasta va adăuga aciunea meniului corespunzător la meniul *Plugins*. Există metode alternative pentru a adăuga aciunea la un alt meniu. Iată o listă a acestor metode:

- `addPluginToRasterMenu()`
- `addPluginToVectorMenu()`
- `addPluginToDatabaseMenu()`
- `addPluginToWebMenu()`

Toate acestea au aceeai sintaxă ca metoda `addPluginToMenu()`.

Adăugarea unui meniu la plugin-ul dvs. printr-una din metoldele predefinite este recomandată pentru a păstra coerena în stilul de organizare a plugin-urilor. Cu toate acestea, putei adăuga grupul dvs. de meniuri personalizate direct în bara de meniu, aa cum demonstrează următorul exemplu :

```
def initGui(self):
    self.menu = QMenu(self.iface.mainWindow())
    self.menu.setObjectName("testMenu")
    self.menu.setTitle("MyMenu")

    self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWin
    self.action.setObjectName("testAction")
    self.action.setWhatsThis("Configuration for test plugin")
    self.action.setStatusTip("This is status tip")
    QObject.connect(self.action, SIGNAL("triggered()"), self.run)
```

```
    self.menu.addAction(self.action)

    menuBar = self.iface.mainWindow().menuBar()
    menuBar.insertMenu(self.iface.firstRightStandardMenu().menuAction(), self.menu)

def unload(self):
    self.menu.deleteLater()
```

Don't forget to set QAction and QMenu objectName to a name specific to your plugin so that it can be customized.

12.2.4 Fiier de resurse

Putei vedea că în `initGui()` am folosit o pictogramă din fiierul de resurse (denumit `resources.qrc`, în cazul nostru)

```
<RCC>
  <qresource prefix="/plugins/testplug" >
     <file>icon.png</file>
  </qresource>
</RCC>
```

Este bine să folosii un prefix pentru a evita coliziunile cu alte plugin-uri sau cu oricare alte păr i ale QGIS, în caz contrar, s-ar putea obine rezultate nedorite. Trebuie doar să generai un fiier Python care va conine resursele. Acest lucru se face cu comanda **pyrcc4**

```
pyrcc4 -o resources.py resources.qrc
```

i asta e tot ... nimic complicat :)

Dacă ai făcut totul corect, ar trebui să găsii i să încărcai plugin-ul în managerul de plugin-uri i să vedei un mesaj în consolă, atunci când este selectat meniul adecvat sau pictograma din bara de instrumente.

Când lucrai la un plug-in real, este înelept să stocai plugin-ul într-un alt director (de lucru), i să creai un fiier make care va genera UI + fiierele de resurse i să instalai plugin-ul în instalarea QGIS.

12.3 Documentaie

Documentaia pentru plugin poate fi scrisă ca fiiere HTML. Modulul `qgis.utils` oferă o funcie, `showPluginHelp()`, care se va deschide navigatorul de fiiere, în acelai mod ca i altă fereastră de ajutor QGIS.

Funcia `showPluginHelp`()` caută fiierele de ajutor în acelai director ca i modulul care îl apelează. Acesta va căuta, la rândul său, în `index-ll_cc.html`, `index-ll.html`, `index-en.html`, `index-en_us.html` i `index.html`, afiând ceea ce găsete mai întâi. Aici `ll_cc` reprezintă limba în care se afiează QGIS. Acest lucru permite multiplelor traduceri ale documentelor să fie incluse în plugin.

Funcia `showPluginHelp()` poate lua, de asemenea, parametrii packageName, care identifică plugin-ul specific pentru care va fi afiat ajutorul, numele de fiier, care poate înlocui 'index' în numele fiierelor în care se caută, i seciunea, care este numele unei ancore HTML în documentul în care se va poziiona browser-ul.

Setările IDE pentru scrierea i depanarea de plugin-uri

Dei fiecare programator preferă un anumit editor IDE/Text, iată câteva recomandări de setare a unor IDE-uri populare, pentru scrierea i depanarea plugin-urilor Python specifice QGIS.

13.1 O notă privind configurarea IDE-ului în Windows

În Linux nu este necesară nici o configurare suplimentară pentru dezvoltarea plug-in-urilor. Dar în Windows trebuie să vă asigurai că avei aceleai setări de mediu i folosii aceleai bibliotecile i interpretor ca i QGIS. Cel mai rapid mod de a face acest lucru, este de a modifica fiierul batch de pornire de a QGIS.

Dacă ai folosit programul de instalare OSGeo4W, îl putei găsi în folderul bin al propriei instalări OSGeoW. Căutai ceva de genul `C:\OSGeo4W\bin\qgis-unstable.bat`.

Pentru utilizarea IDE-ului Pyscripter, iată ce avei de făcut:

- Facei o copie a qgis-unstable.bat i redenumii-o pyscripter.bat.
- Deschidei-o într-un editor. Apoi eliminai ultima linie, cea care startează QGIS.
- Adăugai o linie care să indice calea către executabilul Pyscripter i adăugai i argumentul care stabilete versiunea de Python ce urmează a fi utilizată (2.7 în cazul QGIS 2.0)
- De asemenea, adăugai i un argument care să indice folderul unde poate găsi Pyscripter dll-ul Python folosit de către QGIS, acesta aflându-se în folderul bin al instalării OSGeoW

```
@echo off
SET OSGEO4W_ROOT=C:\OSGeo4W
call "%OSGEO4W_ROOT%"\bin\o4w_env.bat
call "%OSGEO4W_ROOT%"\bin\gdal16.bat
@echo off
path %PATH%;%GISBASE%\bin
Start C:\pyscripter\pyscripter.exe --python25 --pythondllpath=C:\OSGeo4W\bin
```

Acum, când vei efectua dublu clic pe acest fiier batch, el va starta Pyscripter, având calea corectă.

Mai popular decât Pyscripter, Eclipse este o alegere comună în rândul dezvoltatorilor. În următoarele seciuni, vom explica cum să-l configurai pentru dezvoltarea i testarea plugin-urilor. Când îl pregătii pentru utilizarea în Windows, ar trebui să creai, de asemenea, un fiier batch pe care să-l utilizai la pornirea Eclipse.

Pentru a crea fiierul batch, urmai aceti pai.

- Localizai folderul în care rezidă file:*qgis_core.dll*. În mod normal, el se găsete în `C:OSGeo4Wappsqgisbin`, dar dacă ai compilat propria aplicaie QGIS, atunci el va fi în folderul `output/bin/RelWithDebInfo`
- Localizai executabilul `eclipse.exe`.
- Creai următorul script i folosii-l pentru a starta Eclipse, atunci când dezvoltai plugin-uri QGIS.

```
call "C:\OSGeo4W\bin\o4w_env.bat"
set PATH=%PATH%;C:\path\to\your\qgis_core.dll\parent\folder
C:\path\to\your\eclipse.exe
```

13.2 Depanare cu ajutorul Eclipse i PyDev

13.2.1 Instalare

Pentru a utiliza Eclipse, asigurai-vă că ai instalat următoarele

- Eclipse
- Aptana Eclipse Plugin sau PyDev
- QGIS 2.0

13.2.2 Pregătirea QGIS

E necesară efectuarea anumitor aciuni pregătitoare pentru însui QGIS. Două plugin-uri sunt de interes: *Remote Debug* i *Plugin Reloader*.

- Mergei la *Plugins* → *Fetch python plugins*
- Căutai *Remote Debug* (la această dată este încă experimental, deci, în cazul în care nu-l observai, va trebui să activai plugin-urile experimentale în fila Opiuni). Instalai-l.
- De asemenea, căutai *Plugin Reloader* i instalai-l. Acest lucru vă va permite să reîncărcai un plug-in, fără a fi necesare închiderea i repornirea QGIS.

13.2.3 Configurarea Eclipse

În Eclipse, creai un nou proiect. Putei să selectai *General Project* i să legai ulterior sursele dvs. reale, aa că nu prea contează unde plasai acest proiect.

Acum, facei clic dreapta pe noul proiect i alegei *New* → *Folder*.

Facei clic pe **[Advanced]** i alegei *Link to alternate location (Linked Folder)*. În cazul în care deja avei sursele pe care dorii să le depanai, le putei alege, în caz contror, creai un folder aa cum s-a explicat anterior

Acum, în fereastra *Project Explorer*, va apărea arborele sursă i vei putea începe să lucrai la cod. Avei disponibile deja evidenierea sintaxei i toate celelalte instrumente puternice din IDE.

13.2.4 Configurarea depanatorului

Pentru a vedea depanatorul la lucru, comutai în perspectiva Depanare din Eclipse (*Window* → *Open Perspective* → *Other* → *Debug*).

Acum, pornii serverul de depanare PyDev, alegând *PyDev* → *Start Debug Server*.

În acest moment Eclipse ateaptă o conexiune de la QGIS către serverul de depanare, iar când QGIS se va conecta la serverul de depanare va fi permis controlul scripturilor Python. Exact pentru acest lucru am instalat plugin-ul *Remote Debug*. Deci, startai QGIS, în cazul în care nu ai făcut-o deja i efectuai clic pe simbolul insectei.

Acum putei seta un punct de întrerupere i de îndată ce codul îl va atinge, execuia se va opri, după care vei putea inspecta starea actuală a plug-in-ului. (Punctul de întrerupere este punctul verde din imaginea de mai jos, i se poate seta printr-un dublu clic în spaiul alb din stânga liniei în care dorii un punct de întrerupere)

Un aspect foarte interesant este faptul că putei utiliza consola de depanare. Asigurai-vă că execuia este, în mod curent, stopată, înainte de a continua.

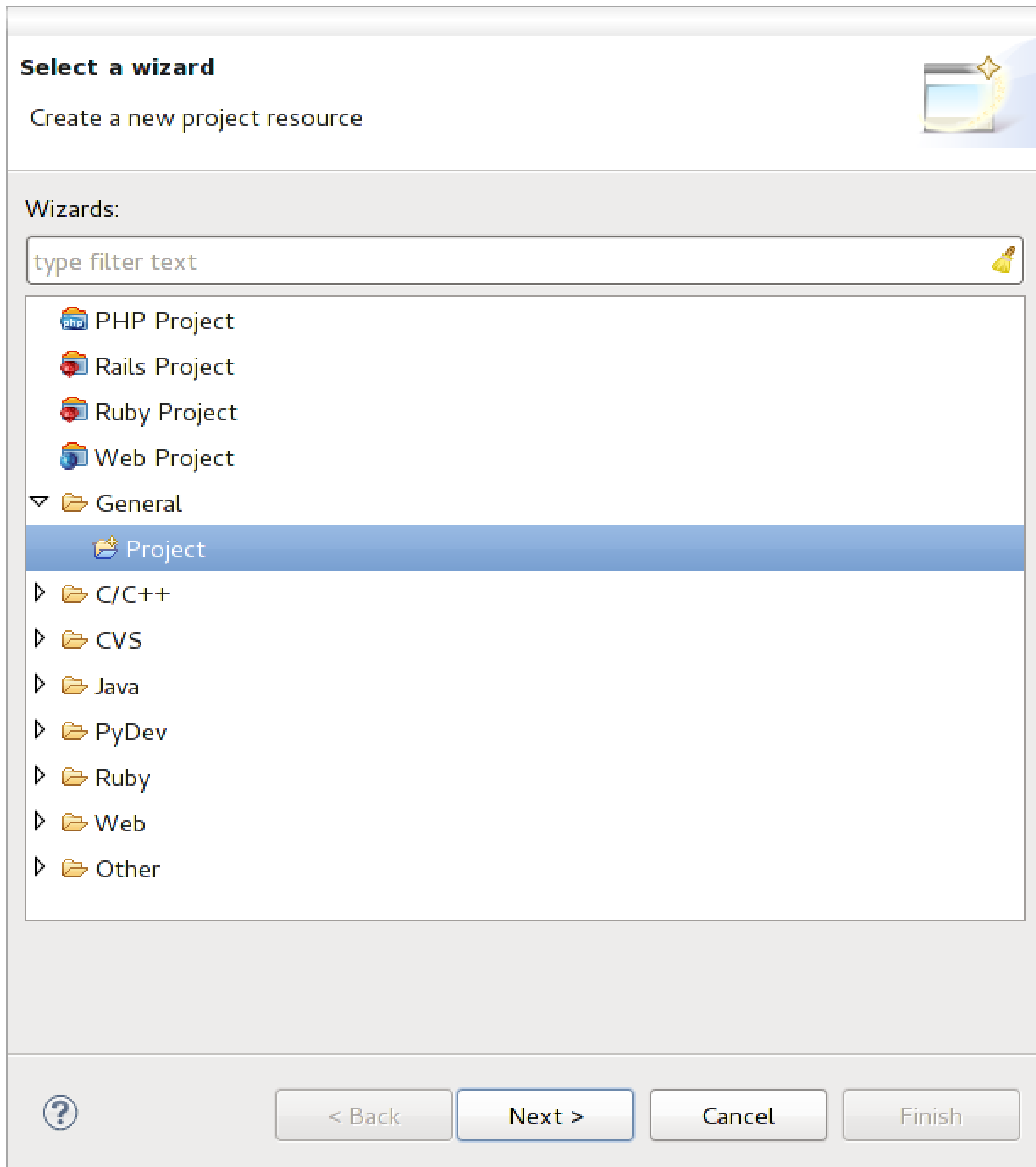

Figura 13.1: Proiectul Eclipse

```
88
89      def printProfile(self):
90          printer = QPrinter( QPrinter.HighResolution )
91          printer.setOutputFormat( QPrinter.PdfFormat )
92          printer.setPaperSize( QPrinter.A4 )
93          printer.setOrientation( QPrinter.Landscape )
94
95          printPreviewDlg = QPrintPreviewDialog( )
96          printPreviewDlg.paintRequested.connect( self.printRequested )
97
98          printPreviewDlg.exec_()
99
100     @pyqtSlot( QPrinter )
101     def printRequested( self, printer ):
102         self.webView.print_( printer )
```

Figura 13.2: Punct de întrerupere

Deschidei fereastra consolei (*Window* → *Show view*). Se va afia consola *Debug Server*, ceea ce nu este prea interesant. În schimb, butonul **[Open Console]** permite trecerea la mult mai interesanta consolă de depanare PyDev. Facei clic pe săgeata de lângă butonul **[Open Console]** i alegei *PyDev Console*. Se deschide o fereastră care vă va întreba ce consolă dorii să deschidei. Alegei *PyDev Debug Console*. În cazul când aceasta este gri, vă cere să pornii depanatorul i să selectai un cadru valid, asigurai-vă că ai ataat depanatorul la distană, iar în prezent suntei pe un punct de întrerupere.

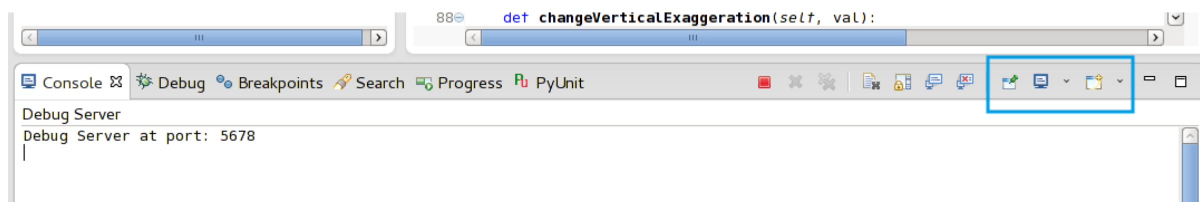

Figura 13.3: Consola de depanare PyDev

Acum avei o consolă interactivă care vă permite să testai orice comenzi din interior, în contextul actual. Putei manipula variabile, să efectuai apeluri API sau orice altceva.

Un pic enervant este faptul că, de fiecare dată când introducei o comandă, consola comută înapoi la serverul de depanare. Pentru a opri acest comportament, avei posibilitatea să facei clic pe butonul *Pin Console* din pagina serverului de depanare, pentru reinerea acestei decizii, cel puin pentru sesiunea de depanare curentă.

13.2.5 Configurai Eclipse pentru a înelege API-ul

O caracteristică facilă este de a pregăti Eclipse pentru API-ul QGIS. Aceasta va permite verificarea eventualelor greeli de ortografie din cadrul codului. Dar nu doar atât, va permite ca Eclipse să autocompleteze din importurile către apelurile API.

Pentru a face acest lucru, Eclipse analizează fiierele bibliotecii QGIS i primete toate informaiile de acolo. Singurul lucru pe care trebuie să-l facei este de a-i indica lui Eclipse unde să găsească bibliotecile.

Facei clic pe *Window* → *Preferences* → *PyDev* → *Interpreter* → *Python*.

Vei vedea interpretorul de python (pe moment versiunea 2.7) configurat, în partea de sus a ferestrei i unele file în partea de jos. Filele interesante pentru noi sunt *Libraries* i *Forced Builtins*.

În primul rând deschidei fila Libraries. Adăugai un folder nou i selectai folderul python al aplicaiei QGIS instalate. Dacă nu tii unde se află acest director (acesta nu este folderul plugin-urilor) deschidei QGIS, startai o consolă python i pur i simplu introducei `qgis`, apoi apăsai Enter. Acest lucru vă va arăta care modul QGIS este folosit, precum i calea sa. tergei `/qgis/__init__.pyc` i vei obine calea pe care o căutai.

Ar trebui să adăugai, de asemenea, propriul director de plugin-uri aici (în Linux este `~/.qgis/python/plugins`).

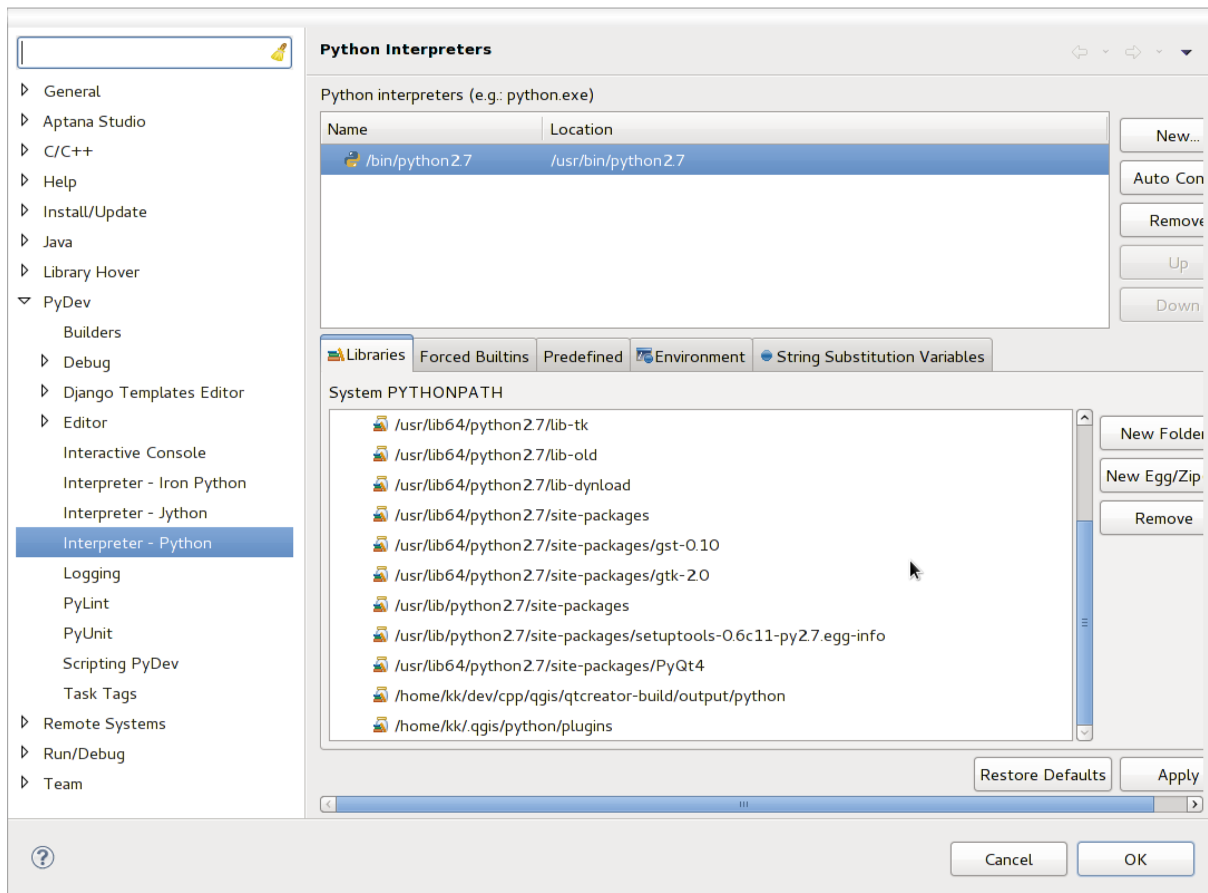

Figura 13.4: Consola de depanare PyDev

Apoi deschidei tab-ul *Forced Builtins*, facei clic pe *New...* i introducei `qgis`. Acest lucru îl va face pe Eclipse să analizeze API-ul QGIS. Probabil dorii, de asemenea, ca Eclipse să tie i despre API-ul PyQt4. Prin urmare, adăugai i PyQt4 ca builtin forat. Probabil că ar trebui să fie deja prezent în fila bibliotecilor.

Facei clic pe *OK* i ai terminat.

Notă: la orice modificare a API-ului QGIS (de exemplu, dacă compilai QGIS master i s-a schimbat fiierul SIP), ar trebui să mergei înapoi la această pagină i pur i simplu facei clic pe *Apply*. Acest lucru va permite Eclipse să analizeze toate bibliotecile din nou.

Pentru o altă setare posibilă de Eclipse, pentru a lucra cu plugin-urile Python QGIS, verificai acest link

13.3 Depanarea cu ajutorul PDB

Dacă nu folosii un IDE, cum ar fi Eclipse, putei depana folosind PDB, urmând aceti pai.

Mai întâi, adăugai acest cod în locul în care dorii depanarea

```
# Use pdb for debugging
import pdb
# These lines allow you to set a breakpoint in the app
pyqtRemoveInputHook()
pdb.set_trace()
```

Apoi executai QGIS din linia de comandă.

În Linux rulai:

$ ./Qgis

În Mac OS X rulai:

$ /Applications/Qgis.app/Contents/MacOS/Qgis

Iar când aplicaia atinge punctul de întrerupere avei posibilitatea de a tasta în consolă!

DE EFECTUAT: Adăugai informaii pentru testare

CAPITOLUL 14

Utilizarea straturilor plugin-ului

Dacă plugin-ul dvs. folosete propriile metode de a randa un strat de hartă, scrierea propriului tip de strat, bazat pe QgsPluginLayer, ar putea fi cel mai bun mod de a implementa acest lucru.

DE EFECTUAT: Verificai corectitudinea i elaborai cazuri de corectă utilizare pentru QgsPluginLayer, ...

14.1 Subclasarea QgsPluginLayer

Below is an example of a minimal QgsPluginLayer implementation. It is an excerpt of the Watermark example plugin:

```
class WatermarkPluginLayer(QgsPluginLayer):

  LAYER_TYPE="watermark"

  def __init__(self):
    QgsPluginLayer.__init__(self, WatermarkPluginLayer.LAYER_TYPE, "Watermark plugin layer")
    self.setValid(True)

  def draw(self, rendererContext):
    image = QImage("myimage.png")
    painter = rendererContext.painter()
    painter.save()
    painter.drawImage(10, 10, image)
    painter.restore()
    return True
```

De asemenea, pot fi adăugate metode pentru citirea i scrierea de informaii specifice în fiierul proiectului

```
def readXml(self, node):
  pass

def writeXml(self, node, doc):
  pass
```

Când încărcai un proiect care conine un astfel de strat, este nevoie de o fabrică de clase

```
class WatermarkPluginLayerType(QgsPluginLayerType):

  def __init__(self):
    QgsPluginLayerType.__init__(self, WatermarkPluginLayer.LAYER_TYPE)

  def createLayer(self):
    return WatermarkPluginLayer()
```

Putei adăuga, de asemenea, codul pentru afiarea de informaii personalizate în proprietăile stratului

```
def showLayerProperties(self, layer):
  pass
```

CAPITOLUL 15

Compatibilitatea cu versiunile QGIS anterioare

15.1 Meniul plugin-ului

If you place your plugin menu entries into one of the new menus (*Raster*, *Vector*, *Database* or *Web*), you should modify the code of the `initGui()` and `unload()` functions. Since these new menus are available only in QGIS 2.0, the first step is to check that the running QGIS version has all necessary functions. If the new menus are available, we will place our plugin under this menu, otherwise we will use the old *Plugins* menu. Here is an example for *Raster* menu

```
def initGui(self):
  # create action that will start plugin configuration
  self.action = QAction(QIcon(":/plugins/testplug/icon.png"), "Test plugin", self.iface.mainWindo
  self.action.setWhatsThis("Configuration for test plugin")
  self.action.setStatusTip("This is status tip")
  QObject.connect(self.action, SIGNAL("triggered()"), self.run)

  # check if Raster menu available
  if hasattr(self.iface, "addPluginToRasterMenu"):
    # Raster menu and toolbar available
    self.iface.addRasterToolBarIcon(self.action)
    self.iface.addPluginToRasterMenu("&Test plugins", self.action)
  else:
    # there is no Raster menu, place plugin under Plugins menu as usual
    self.iface.addToolBarIcon(self.action)
    self.iface.addPluginToMenu("&Test plugins", self.action)

  # connect to signal renderComplete which is emitted when canvas rendering is done
  QObject.connect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest)

def unload(self):
  # check if Raster menu available and remove our buttons from appropriate
  # menu and toolbar
  if hasattr(self.iface, "addPluginToRasterMenu"):
    self.iface.removePluginRasterMenu("&Test plugins",self.action)
    self.iface.removeRasterToolBarIcon(self.action)
  else:
    self.iface.removePluginMenu("&Test plugins",self.action)
    self.iface.removeToolBarIcon(self.action)

  # disconnect from signal of the canvas
  QObject.disconnect(self.iface.mapCanvas(), SIGNAL("renderComplete(QPainter *)"), self.renderTest
```

CAPITOLUL 16

Lansarea plugin-ului dvs.

O dată ce plugin-ul este gata i credei că el ar putea fi de ajutor pentru unii utilizatori, nu ezitai să-l încărcai la *Depozitul oficial al plugin-urilor python*. Pe acea pagină putei găsi instruciuni de împachetare i de pregătire a plugin-ului, pentru a lucra bine cu programul de instalare. Sau, în cazul în care ai dori să înfiinai un depozit propriu pentru plugin-uri, creai un simplu fiier XML, care va lista plugin-urile i metadatele lor, exemplu pe care îl putei vedea în depozite pentru plugin-uri.

16.1 Depozitul oficial al plugin-urilor python

Putei găsi depozitul *oficial* al plugin-urilor python la http://plugins.qgis.org/.

Pentru a folosi depozitul oficial, trebuie să obinei un ID OSGEO din portalul web OSGEO.

O dată ce ai încărcat plugin-ul, acesta va fi aprobat de către un membru al personalului i vei primi o notificare.

DE EFECTUAT: Introducei un link către documentul guvernanei

16.1.1 Permisiuni

Aceste reguli au fost implementate în depozitul oficial al plugin-urilor:

- fiecare utilizator inregistrat poate adăuga un nou plugin
- membrii *staff-ului* pot aproba sau dezaproba toate versiunile plugin-ului
- utilizatorii care au permisiunea specială *plugins.can_approve* au versiunile pe care le încarcă aprobate în mod automat
- utilizatorii care au permisiunea specială *plugins.can_approve* pot aproba versiunile încărcate de către alii, atât timp cât acetia sunt prezeni în lista *proprietarilor* de plugin-uri
- un anumit plug-in pot fi ters i editat doar de utilizatorii *staff-ului* i de către *proprietarii* plugin-uri
- în cazul în care un utilizator fără permisiunea *plugins.can_approve* încarcă o nouă versiune, versiunea plug-inului nu va fi aprobată, din start.

16.1.2 Managementul încrederii

Membrii personalului pot acorda *încredere* creatorilor de plugin-uri, bifând permisiunea *plugins.can_approve* în cadrul front-end-ului.

Detaliile despre plugin oferă legături directe pentru a crete încrederea în creatorul sau *proprietarul*.plugin-ului.

16.1.3 Validare

Metadatele plugin-ului sunt importate automat din pachetul arhivat i sunt validate, la încărcarea plugin-ului.

Iată câteva reguli de validare pe care ar trebui să le cunoatei atunci când dorii să încărcai un plugin în depozitul oficial:

1. the name of the main folder containing your plugin must contain only contains ASCII characters (A-Z and a-z), digits and the characters underscore (_) and minus (-), also it cannot start with a digit
2. `metadata.txt` este necesar
3. toate metadatele necesare, menionate în *tabela de metadate* trebuie să fie prezente
4. the *version* metadata field must be unique

16.1.4 Structura plugin-ului

Conform regulilor de validare, pachetul compresat (.zip) al plugin-ului trebuie să aibă o structură specifică, pentru a fi validat ca plugin funcional. Deorece plugin-ul va fi dezarhivat în interiorul directorului de plugin-uri ale utilizatorului, el trebuie să aibă propriul director în interiorul fiierului zip, pentru a nu interfera cu alte plugin-uri. Fiierele obligatorii sunt: `metadata.txt` i `__init__.py`. Totui, ar fi frumos să existe un `README` i, desigur, o pictogramă care să reprezinte pluginul (`resources.qrc`). Iată un exemplu despre modul în care ar trebui să arate un plugin.zip.

```
plugin.zip
  pluginfolder/
  |-- i18n
  |    |-- translation_file_de.ts
  |-- img
  |    |-- icon.png
  |    `-- iconsource.svg
  |-- __init__.py
  |-- Makefile
  |-- metadata.txt
  |-- more_code.py
  |-- main_code.py
  |-- README
  |-- resources.qrc
  |-- resources_rc.py
  `-- ui_Qt_user_interface_file.ui
```

CAPITOLUL 17

Fragmente de cod

Această seciune conine fragmente de cod, menite să faciliteze dezvoltarea plugin-urilor.

17.1 Cum să apelăm o metodă printr-o combinaie rapidă de taste

În plug-in adăugai în `initGui()`

```
self.keyAction = QAction("Test Plugin", self.iface.mainWindow())
self.iface.registerMainWindowAction(self.keyAction, "F7") # action1 triggered by F7 key
self.iface.addPluginToMenu("&Test plugins", self.keyAction)
QObject.connect(self.keyAction, SIGNAL("triggered()"),self.keyActionF7)
```

În `unload()` add

```
self.iface.unregisterMainWindowAction(self.keyAction)
```

Metoda care este chemată atunci când se apasă tasta F7

```
def keyActionF7(self):
  QMessageBox.information(self.iface.mainWindow(),"Ok", "You pressed F7")
```

17.2 Inversarea Stării Straturilor

Începând de la QGIS 2.4, un nou API permite accesul direct la arborele straturilor din legendă. Exemplul următor prezintă modul în care se poate inversa vizibilitatea stratului activ

```
root = QgsProject.instance().layerTreeRoot()
node = root.findLayer(iface.activeLayer().id())
new_state = Qt.Checked if node.isVisible()==Qt.Unchecked else Qt.Unchecked
node.setVisible(new_state)
```

17.3 Cum să accesai tabelul de atribute al entităilor selectate

```
def changeValue(self, value):
  layer = self.iface.activeLayer()
  if(layer):
    nF = layer.selectedFeatureCount()
    if (nF > 0):
    layer.startEditing()
    ob = layer.selectedFeaturesIds()
    b = QVariant(value)
    if (nF > 1):
```

```
      for i in ob:
      layer.changeAttributeValue(int(i),1,b) # 1 being the second column
    else:
      layer.changeAttributeValue(int(ob[0]),1,b) # 1 being the second column
    layer.commitChanges()
    else:
      QMessageBox.critical(self.iface.mainWindow(),"Error", "Please select at least one feature fi
  else:
    QMessageBox.critical(self.iface.mainWindow(),"Error","Please select a layer")
```

Metoda necesită un parametru (noua valoare pentru câmpul atribut al entităi(lor) selectate) i poate fi apelat de către

```
self.changeValue(50)
```

CAPITOLUL 18

Biblioteca de analiză a reelelor

Starting from revision ee19294562 (QGIS >= 1.8) the new network analysis library was added to the QGIS core analysis library. The library:

- creează graful matematic din datele geografice (straturi vectoriale de tip polilinie)
- implementează metode de bază din teoria grafurilor (în prezent, doar algoritmul lui Dijkstra)

Biblioteca analizelor de reea a fost creată prin exportarea funciilor de bază ale plugin-ului RoadGraph, iar acum avei posibilitatea să-i utilizai metodele în plugin-uri sau direct în consola Python.

18.1 Informaii generale

Pe scurt, un caz tipic de utilizare poate fi descris astfel:

1. crearea grafului din geodate (de obicei un strat vectorial de tip polilinie)
2. rularea analizei grafului
3. folosirea rezultatelor analizei (de exemplu, vizualizarea lor)

18.2 Construirea unui graf

Primul lucru pe care trebuie să-l facei — este de a pregăti datele de intrare, ceea ce înseamnă conversia stratului vectorial într-un graf. Toate aciunile viitoare vor folosi acest graf, i nu stratul.

Ca i sursă putem folosi orice strat vectorial de tip polilinie. Nodurile poliliniilor devin noduri ale grafului, segmentele poliliniilor reprezentând marginile grafului. În cazul în care mai multe noduri au aceleai coordonate, atunci ele sunt în acelai nod al grafului. Astfel, două linii care au un nod comun devin conectate între ele.

În plus, în timpul creării grafului este posibilă "fixarea" ('legarea") de stratul vectorial de intrare a oricărui număr de puncte suplimentare. Pentru fiecare punct suplimentar va fi găsită o potrivire — cel mai apropiat nod sau cea mai apropiată muchie a grafului. În ultimul caz muchia va fi divizată iar noul nod va fi adăugat.

Atributele stratului vectorial i lungimea unei muchii pot fi folosite ca proprietăi ale marginii.

Convertorul din strat vectorial în graf este dezvoltat folosind modelul de programare al Constructorului. De construirea grafului răspunde aa-numitul Director. Există doar un singur Director pentru moment: QgsLineVectorLayerDirector. Directorul stabilete setările de bază ale Constructorului, care vor fi folosite pentru a construi un graf dintr-un strat vectorial de tip linie. În prezent, ca i în cazul Directorului, există doar un singur Constructor: QgsGraphBuilder, care creează obiecte <http://qgis.org/api/classQgsGraph.html>'_. Este posibil să dorii implementarea propriilor constructori care să construiască grafuri compatibile cu bibliotecile, cum ar fi BGL sau NetworkX.

Pentru a calcula proprietăile marginii este utilizată strategia modelului de programare. Pentru moment doar strategia QgsDistanceArcProperter este disponibilă, care ia în considerare lungimea traseului. Putei implementa propria strategie, care va folosi toi parametrii necesari. De exemplu, plugin-ul RoadGraph folosete strategia care calculează timpul de călătorie, folosind lungimea muchiei i valoarea vitezei din atribute.

Este timpul de a aprofunda acest proces.

Înainte de toate, pentru a utiliza această bibliotecă ar trebui să importăm modulul networkanalysis

```
from qgis.networkanalysis import *
```

Apoi, câteva exemple pentru crearea unui director

```
# don't use information about road direction from layer attributes,
# all roads are treated as two-way
director = QgsLineVectorLayerDirector(vLayer, -1, '', '', '', 3)

# use field with index 5 as source of information about road direction.
# one-way roads with direct direction have attribute value "yes",
# one-way roads with reverse direction have the value "1", and accordingly
# bidirectional roads have "no". By default roads are treated as two-way.
# This scheme can be used with OpenStreetMap data
director = QgsLineVectorLayerDirector(vLayer, 5, 'yes', '1', 'no', 3)
```

Pentru a construi un director ar trebui să transmitem stratul vectorial, care va fi folosit ca sursă pentru structura grafului i informaiile despre micările permise pe fiecare segment de drum (circulaie unilaterală sau bilaterală, sens direct sau invers). Acest apel arată în felul următor

```
director = QgsLineVectorLayerDirector(vl, directionFieldId,
                                      directDirectionValue,
                                      reverseDirectionValue,
                                      bothDirectionValue,
                                      defaultDirection)
```

Iată lista completă a ceea ce înseamnă aceti parametri:

- `vl` — stratul vectorial utilizat pentru a construi graf
- `directionFieldId` — indexul câmpului din tabelul de atribute, în care sunt stocate informaii despre direciile drumurilor. Dacă este `-1`, atunci aceste informaii nu se folosesc deloc. Număr întreg.
- `directDirectionValue` — valoarea câmpului pentru drumurile cu sens direct (trecere de la primul punct de linie la ultimul). ir de caractere.
- `reverseDirectionValue` — valoarea câmpului pentru drumurile cu sens invers (în micare de la ultimul punct al liniei până la primul). ir de caractere.
- `bothDirectionValue` — valoarea câmpului pentru drumurile bilaterale (pentru astfel de drumuri putem trece de la primul la ultimul punct i de la ultimul la primul). ir de caractere.
- `defaultDirection` — direcia implicită a drumului. Această valoare va fi folosită pentru acele drumuri în care câmpul `directionFieldId` nu este setat sau are o valoare diferită de oricare din cele trei valori specificate mai sus. Număr întreg. `1` indică sensul direct, `2` indică sensul inversă, iar `3` indică ambele sensuri.

Este necesară, apoi, crearea unei strategii pentru calcularea proprietăilor marginii

```
properter = QgsDistanceArcProperter()
```

Apoi spunei directorului despre această strategie

```
director.addProperter(properter)
```

Acum putem crea constructorul, care va crea graful. Constructorul clasei QgsGraphBuilder ia mai multe argumente:

- crs — sistemul de coordonate de referină de utilizat. Argument obligatoriu.

- otfEnabled — utilizai sau nu reproiectarea “din zbor”. În mod implicit const:*True* (folosii OTF).
- topologyTolerance — tolerana topologică. Valoarea implicită este 0.
- ellipsoidID — elipsoidul de utilizat. În mod implicit “WGS84”.

```
# only CRS is set, all other values are defaults
builder = QgsGraphBuilder(myCRS)
```

De asemenea, putem defini mai multe puncte, care vor fi utilizate în analiză. De exemplu

```
startPoint = QgsPoint(82.7112, 55.1672)
endPoint = QgsPoint(83.1879, 54.7079)
```

Acum că totul este la locul lui, putem să construim graful i să “legăm” aceste puncte la el

```
tiedPoints = director.makeGraph(builder, [startPoint, endPoint])
```

Construirea unui graf poate dura ceva timp (depinzând de numărul de entităi dintr-un strat i de dimensiunea stratului). `tiedPoints` reprezintă o listă cu coordonatele punctelor “asociate”. Când s-a terminat operaiunea de construire putem obine graful i să-l utilizăm pentru analiză

```
graph = builder.graph()
```

Cu următorul cod putem obine indecii punctelor noastre

```
startId = graph.findVertex(tiedPoints[0])
endId = graph.findVertex(tiedPoints[1])
```

18.3 Analiza grafului

Analiza de reea este utilizată pentru a găsi răspunsuri la două întrebări: care noduri sunt conectate i identificarea celei mai scurte căi. Pentru a rezolva această problemă, biblioteca de analiză de reea oferă algoritmul lui Dijkstra.

Algoritmul lui Dijkstra găsete cea mai bună cale între unul dintre vârfurile grafului i toate celelalte, precum i valorile parametrilor de optimizare. Rezultatele pot fi reprezentate ca cel mai scurt arbore.

Arborele drumurilor cele mai scurte reprezintă un graf (sau mai precis — arbore) orientat, ponderat, cu următoarele proprietăi:

- doar un singur nod nu are muchii de intrare — rădăcina arborelui
- toate celelalte noduri au numai o margine de intrare
- dacă nodul B este accesibil din nodul A, apoi calea de la A la B este singura disponibilă i este optimă (cea mai scurtă) în acest graf

Pentru a obine cel mai scurt arbore folosii metodele `shortestTree()` i `dijkstra()` ale clasei QgsGraph-Analyzer. Se recomandă utilizarea metodei `dijkstra()`, deoarece lucrează mai rapid i utilizează memoria mult mai eficient.

Metoda `shortestTree()` este utilă atunci când dorii să vă plimbai de-a lungul celei mai scurte căi. Aceasta creează mereu un nou obiect de tip graf (QgsGraph) care acceptă trei variabile:

- source — graf de intrare
- startVertexIdx — Indexul punctului de pe arbore (rădăcina arborelui)
- criterionNum — numărul de proprietăii marginii de folosit (începând de la 0).

```
tree = QgsGraphAnalyzer.shortestTree(graph, startId, 0)
```

Metoda `dijkstra()` are aceleai argumente, dar întoarce două tablouri. În prima matrice, elementul i conine indexul marginii de intrare, sau -1 în cazul în care nu există margini de intrare. În a doua matrice, elementul i conine distanta de la rădăcina arborelui la nodul i, sau DOUBLE_MAX dacă rădăcina nodului este imposibil de căutat.

```
(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, startId, 0)
```

Iată un cod foarte simplu pentru a afia arborele celei mai scurte căi, folosind graful creat cu metoda `shortestTree()` (selectai stratul linie în TOC i înlocuii coordonatele cu ale dvs). **Atenie**: folosii acest cod doar ca exemplu, deoarece el va crea o mulime de obiecte QgsRubberBand , putând fi lent pe seturi de date de mari dimensiuni.

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.743804, 0.22954)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)

tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)

i = 0;
while (i < tree.arcCount()):
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (tree.vertex(tree.arc(i).inVertex()).point())
  rb.addPoint (tree.vertex(tree.arc(i).outVertex()).point())
  i = i + 1
```

Same thing but using `dijkstra()` method:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-1.37144, 0.543836)
tiedPoint = director.makeGraph(builder, [pStart])
pStart = tiedPoint[0]

graph = builder.graph()

idStart = graph.findVertex(pStart)
```

```
(tree, costs) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

for edgeId in tree:
  if edgeId == -1:
    continue
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor (Qt.red)
  rb.addPoint (graph.vertex(graph.arc(edgeId).inVertex()).point())
  rb.addPoint (graph.vertex(graph.arc(edgeId).outVertex()).point())
```

18.3.1 Găsirea celor mai scurte căi

To find the optimal path between two points the following approach is used. Both points (start A and end B) are "tied" to the graph when it is built. Then using the methods `shortestTree()` or `dijkstra()` we build the shortest path tree with root in the start point A. In the same tree we also find the end point B and start to walk through the tree from point B to point A. The Whole algorithm can be written as

```
assign  = B
while  != A
    add point  to path
    get incoming edge for point
    look for point , that is start point of this edge
    assign  =
add point  to path
```

În acest moment avem calea, sub formă de listă inversată de noduri (nodurile sunt listate în ordine inversă, de la punctul de final către cel de start), ele fiind vizitate în timpul parcurgerii căii.

Aici este codul de test pentru consola Python a QGIS (va trebui să selectai stratul linie în TOC i să înlocuii coordonate din cod cu ale dvs.), care folosete metoda `shortestTree()`

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.835953, 0.15679)
pStop = QgsPoint(-1.1027, 0.699986)

tiedPoints = director.makeGraph(builder, [pStart, pStop])
graph = builder.graph()

tStart = tiedPoints[0]
tStop = tiedPoints[1]

idStart = graph.findVertex(tStart)
tree = QgsGraphAnalyzer.shortestTree(graph, idStart, 0)

idStart = tree.findVertex(tStart)
idStop = tree.findVertex(tStop)

if idStop == -1:
  print "Path not found"
```

```
else:
  p = []
  while (idStart != idStop):
    l = tree.vertex(idStop).inArc()
    if len(l) == 0:
      break
    e = tree.arc(l[0])
    p.insert(0, tree.vertex(e.inVertex()).point())
    idStop = e.outVertex()

  p.insert(0, tStart)
  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor(Qt.red)

  for pnt in p:
    rb.addPoint(pnt)
```

And here is the same sample but using `dikstra()` method

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(-0.835953, 0.15679)
pStop = QgsPoint(-1.1027, 0.699986)

tiedPoints = director.makeGraph(builder, [pStart, pStop])
graph = builder.graph()

tStart = tiedPoints[0]
tStop = tiedPoints[1]

idStart = graph.findVertex(tStart)
idStop = graph.findVertex(tStop)

(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

if tree[idStop] == -1:
  print "Path not found"
else:
  p = []
  curPos = idStop
  while curPos != idStart:
    p.append(graph.vertex(graph.arc(tree[curPos]).inVertex()).point())
    curPos = graph.arc(tree[curPos]).outVertex();

  p.append(tStart)

  rb = QgsRubberBand(qgis.utils.iface.mapCanvas())
  rb.setColor(Qt.red)

  for pnt in p:
    rb.addPoint(pnt)
```

18.3.2 Ariile de disponibilitate

Aria de disponibilitate a nodului A este un subset de noduri ale graf-ului, care sunt accesibile din nodul A iar costurile căii de la A la aceste noduri nu sunt mai mari decât o anumită valoare.

Mai clar, acest lucru poate fi dovedit cu următorul exemplu: "Există o echipă de intervenie în caz de incendiu. Ce zone ale oraului acoperă această echipă în 5 minute? Dar în 10 minute? Dar în 15 minute?". Răspunsul la aceste întrebări îl reprezintă zonele de disponibilitate ale echipei de intervenie.

Pentru a găsi zonele de disponibilitate putem folosi metoda `dijksta()` a clasei `QgsGraphAnalyzer`. Este suficientă compararea elementelor matricei de costuri cu valoarea predefinită. În cazul în care costul[i] este mai mic sau egal decât o valoare predefinită, atunci nodul i se află în zona de disponibilitate, în caz contrar este în afară.

Mai dificilă este obinerea granielor zonei de disponibilitate. Marginea de jos reprezintă un set de noduri care încă sunt accesibile, iar marginea de sus un set de noduri inaccesibile. De fapt, acest lucru este simplu: marginea disponibilă a atins aceste margini parcurgând arborele cel mai scurt, pentru care nodul de start este accesibil, spre deosebire de celelalt capăt, care nu este accesibil.

Iată un exemplu

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

from qgis.core import *
from qgis.gui import *
from qgis.networkanalysis import *

vl = qgis.utils.iface.mapCanvas().currentLayer()
director = QgsLineVectorLayerDirector(vl, -1, '', '', '', 3)
properter = QgsDistanceArcProperter()
director.addProperter(properter)
crs = qgis.utils.iface.mapCanvas().mapRenderer().destinationCrs()
builder = QgsGraphBuilder(crs)

pStart = QgsPoint(65.5462, 57.1509)
delta = qgis.utils.iface.mapCanvas().getCoordinateTransform().mapUnitsPerPixel() * 1

rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
rb.setColor(Qt.green)
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() - delta))
rb.addPoint(QgsPoint(pStart.x() + delta, pStart.y() + delta))
rb.addPoint(QgsPoint(pStart.x() - delta, pStart.y() + delta))

tiedPoints = director.makeGraph(builder, [pStart])
graph = builder.graph()
tStart = tiedPoints[0]

idStart = graph.findVertex(tStart)

(tree, cost) = QgsGraphAnalyzer.dijkstra(graph, idStart, 0)

upperBound = []
r = 2000.0
i = 0
while i < len(cost):
  if cost[i] > r and tree[i] != -1:
    outVertexId = graph.arc(tree [i]).outVertex()
    if cost[outVertexId] < r:
      upperBound.append(i)
  i = i + 1

for i in upperBound:
```

```
centerPoint = graph.vertex(i).point()
rb = QgsRubberBand(qgis.utils.iface.mapCanvas(), True)
rb.setColor(Qt.red)
rb.addPoint(QgsPoint(centerPoint.x() - delta, centerPoint.y() - delta))
rb.addPoint(QgsPoint(centerPoint.x() + delta, centerPoint.y() - delta))
rb.addPoint(QgsPoint(centerPoint.x() + delta, centerPoint.y() + delta))
rb.addPoint(QgsPoint(centerPoint.x() - delta, centerPoint.y() + delta))
```

în execuie
aplicaii personalizate, 3
încărcare
Fiiere GPX, 6
Geometrii MySQL, 6
straturi cu text delimitat, 5
Straturi OGR, 5
Straturi PostGIS, 5
straturi raster, 6
Straturi SpatiaLite, 6
straturi vectoriale, 5
Straturi WMS, 6

aisteme de coordonate de referină, 31
API, 1
aplicaii personalizate
în execuie, 3
Python, 2

calcularea valorilor, 42
consolă
Python, 1

entităi
straturi vectoriale iterarea, 13
expresii, 42
evaluare, 44
parsare, 43

Fiiere GPX
încărcare, 6
filtrare, 42
furnizor de memorie, 18

geometrie
accesare, 27
construire, 27
manipulare, 25
predicate i operaiuni, 28
Geometrii MySQL
încărcare, 6

ieire
folosirea Compozitorului de Hări, 39
imagine raster, 41
PDF, 41

index spatial
folosind, 16
interogare
straturi raster, 11
iterarea
entităi, straturi vectoriale, 13

metadata, 56
metadata.txt, 56

personalizat
rendere, 23
plugin-uri, 69
apelarea unei metode printr-o combinaie rapidă de taste, 71
atributele de acces ale entităilor selectate, 71
comutarea straturilor, 71
depozitul oficial al plugin-urilor python, 69
dezvoltare, 51
documentaie, 58
fiier de resurse, 58
fragmente de cod, 58
implementare help, 58
lansarea, 64
metadata.txt, 54, 56
scriere, 53
scriere cod, 54
testare , 64
proiecii, 32
Python
aplicaii personalizate, 2
consolă, 1
dezvoltarea plugin-urilor, 51
plugin-uri, 1

randare hartă, 37
simplu, 39
rastere
multibandă, 11
simplă bandă, 10
recitire
straturi raster, 11
registrul straturilor de hartă, 7
adăugarea unui strat, 7
render cu simbol gradual, 20
render cu simbologie clasificată, 19

render cu un singur simbol , 19
rendere
    personalizat, 23
resources.qrc, 58

setări
    citire, 45
    global, 47
    proiect, 47
    stocare, 45
    strat de hartă, 48
simbologie
    render cu simbol clasificat, 19
    render cu simbol gradual, 20
    render cu un singur simbol , 19
    vechi, 25
simboluri
    lucrul cu, 21
straturi cu text delimitat
    încărcare, 5
Straturi OGR
    încărcare, 5
Straturi PostGIS
    încărcare, 5
straturi raster
    încărcare, 6
    detalii, 9
    folosind, 7
    interogare, 11
    recitire, 11
    stil de desenare, 9
Straturi SpatiaLite
    încărcare, 6
straturi vectoriale
    încărcare, 5
    editare, 14
    iterarea entităi, 13
    scris, 17
    simbologie, 19
Straturi WMS
    încărcare, 6
straturile plugin-ului, 64
    subclasarea QgsPluginLayer, 65
straturile simbolului
    crearea tipurilor personalizate, 21
    lucrul cu, 21
suportul hării, 32
    încapsulare, 33
    arhitectură, 33
    benzi de cauciuc, 35
    dezvoltarea elementelor personalizate pentru suportul de hartă , 37
    dezvoltarea instrumentelor de hartă personalizate, 36
    instrumente pentru hartă, 34
    marcaje vertex, 35

tipărire hartă, 37