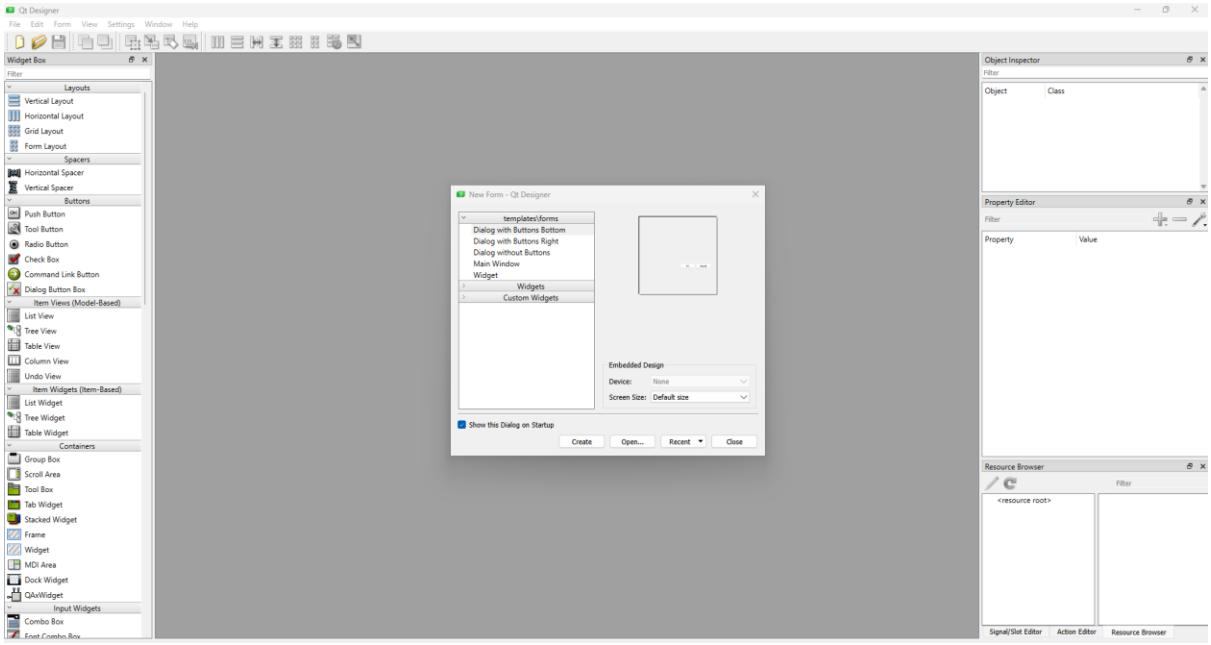
Criando Plugins QGIS com pyQGIS

Módulo 2 – QtDesigner, PluginBuilder e um plugin de verdade

1 - O QtDesiner e o PluginBuilder

Para facilitar a nossa vida existem duas ferramentas que auxiliam, e bastante, na criação de plugins. A primeira é o QtDesigner para construirmos as interfaces gráficas do usuário (GUI) e o PluginBuilder que cria praticamente todos os arquivos do plugin para nós.

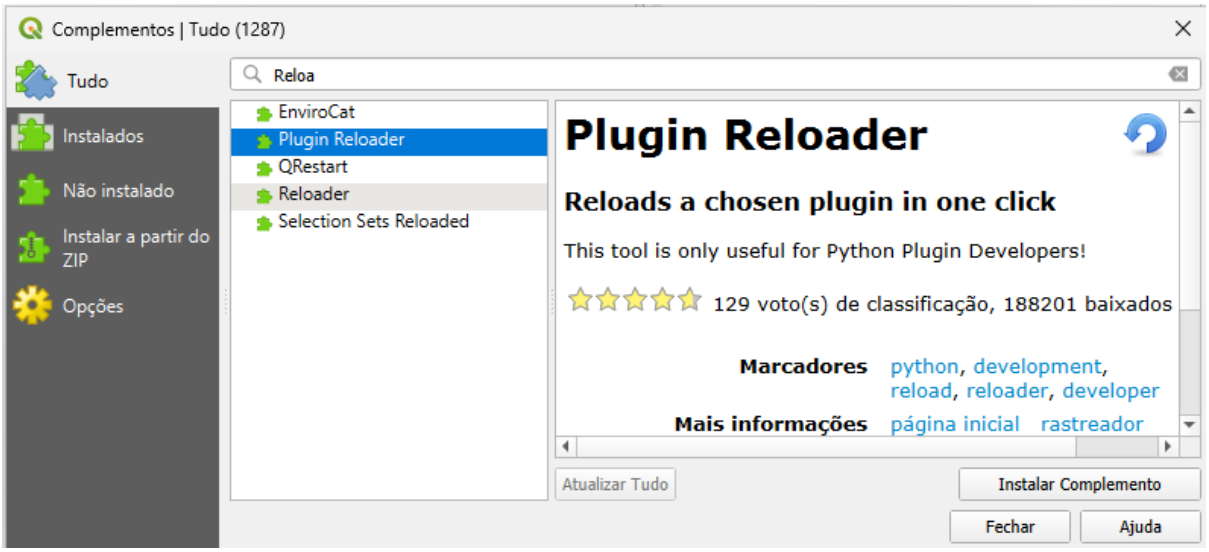
O QtDesigner já vem com o QGIS e a tela inicial dele é:



Usaremos ele para criar o nosso plugin de verdade deste módulo.

O PluginBuilder precisa ser instalado e fazemos isso abrindo o QGIS e indo menu **Complementos->Gerenciar e instalar Complementos**. Selecione o botão Tudo e digite builder na caixa de busca. O Plugin Builder 3 deve aparecer. Instale o complemento.

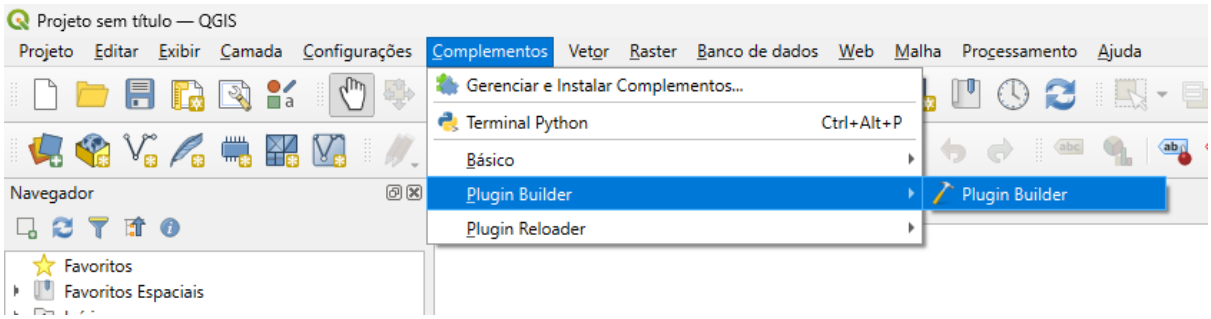
Outra ferramenta interessante de se ter é o Plugin Reloader. Instale ele também.



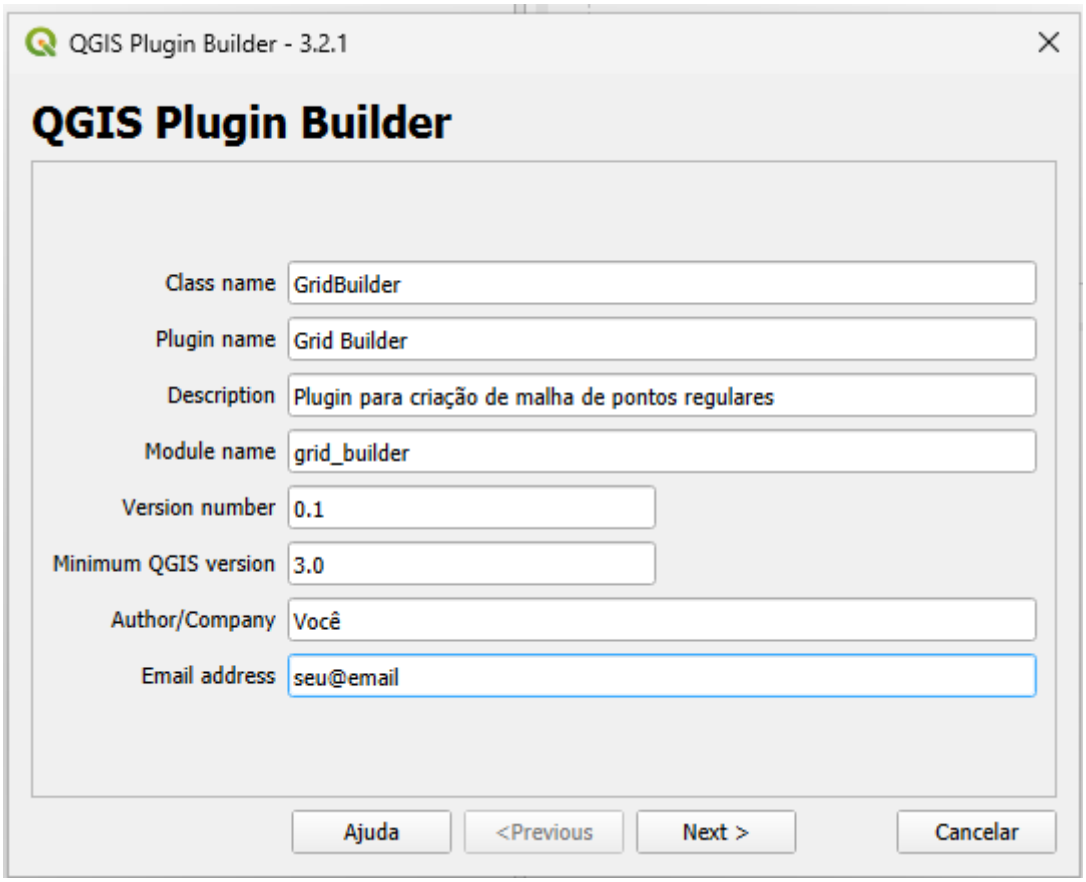
Vamos agora usar essas ferramentas na construção do nosso primeiro plugin funcional.

2 - Construindo o esqueleto do Grid_Builder no Plugin Builder 3

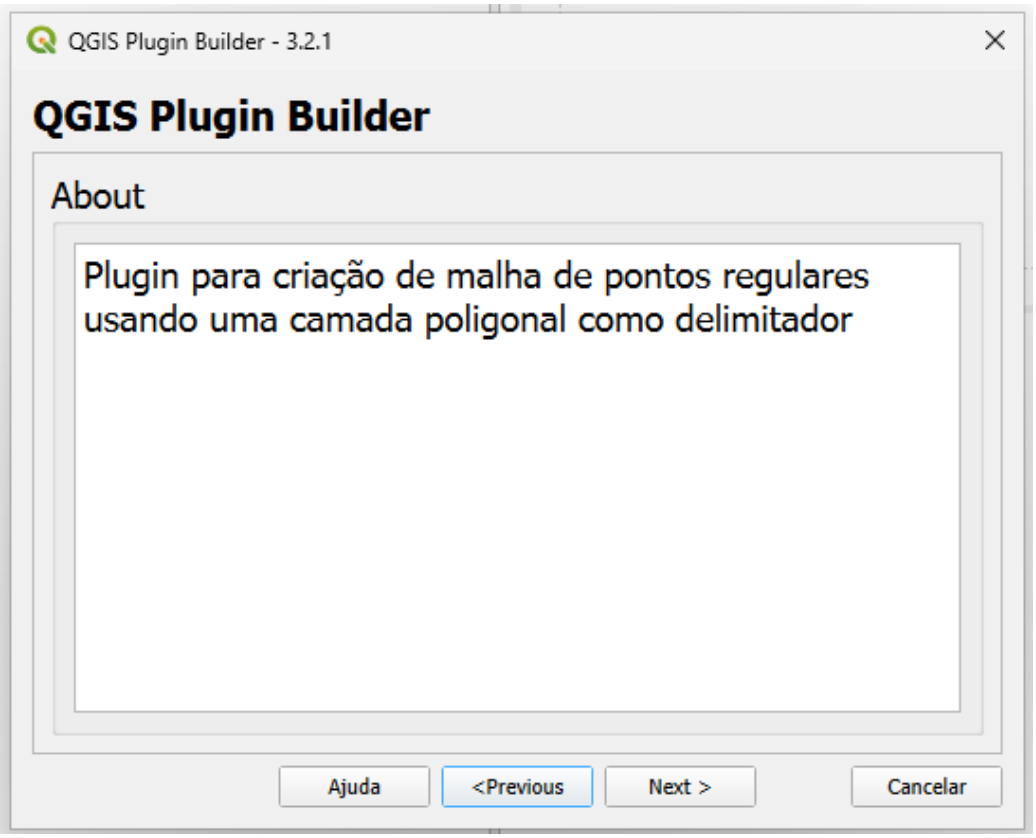
Vamos iniciar pelo Plugin Builder:



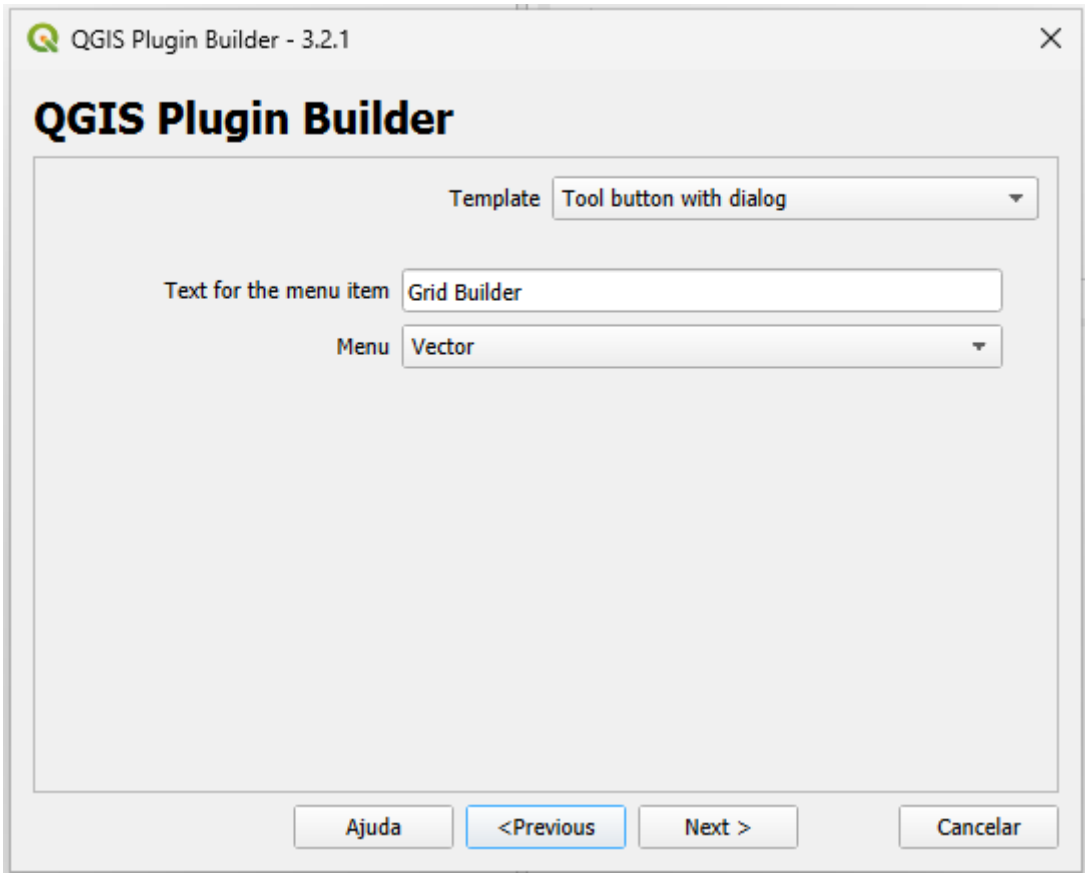
Inicie o plugin e preencha os campos dos formulários conforme as imagens a seguir:



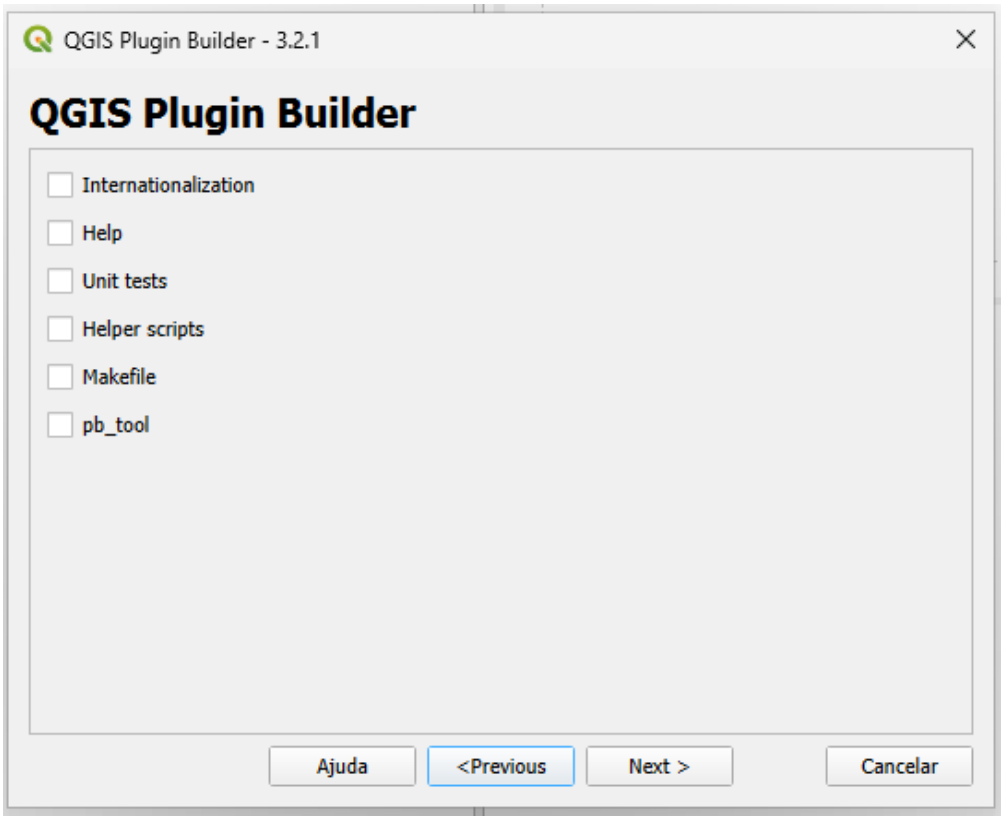
Esse primeiro formulário será usado na criação do arquivio metadata.txt e na definição do nome das classes do plugin.



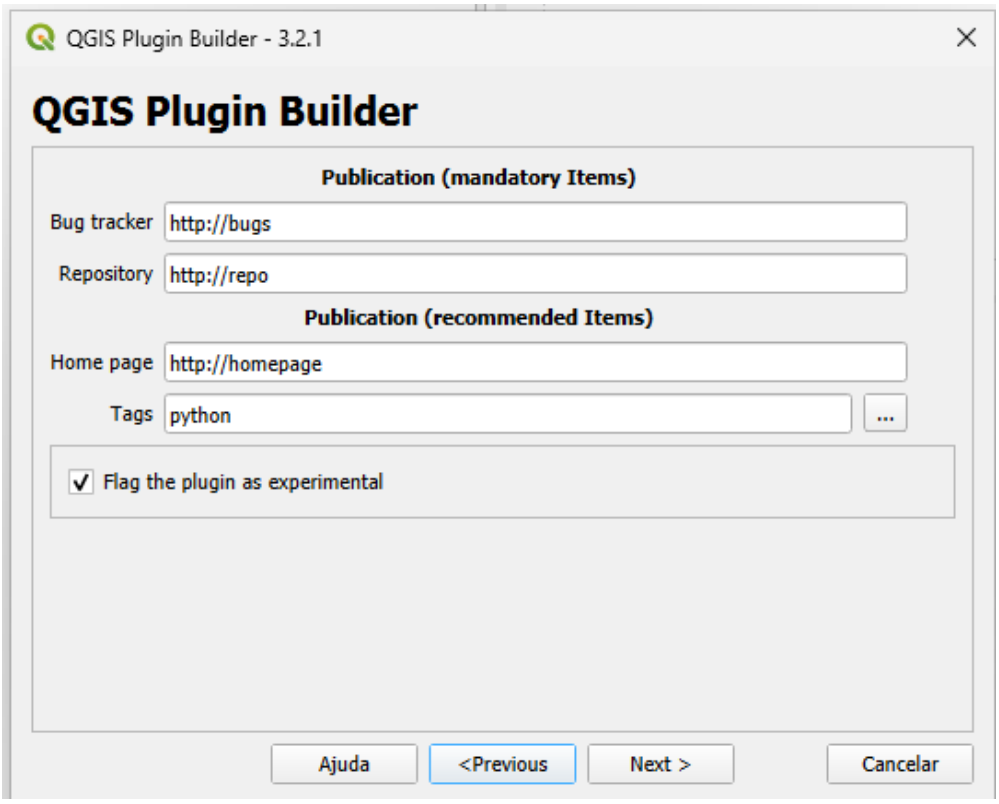
Descrição mais detalhada sobre o plugin que também será colocado no arquivo metadata.txt.



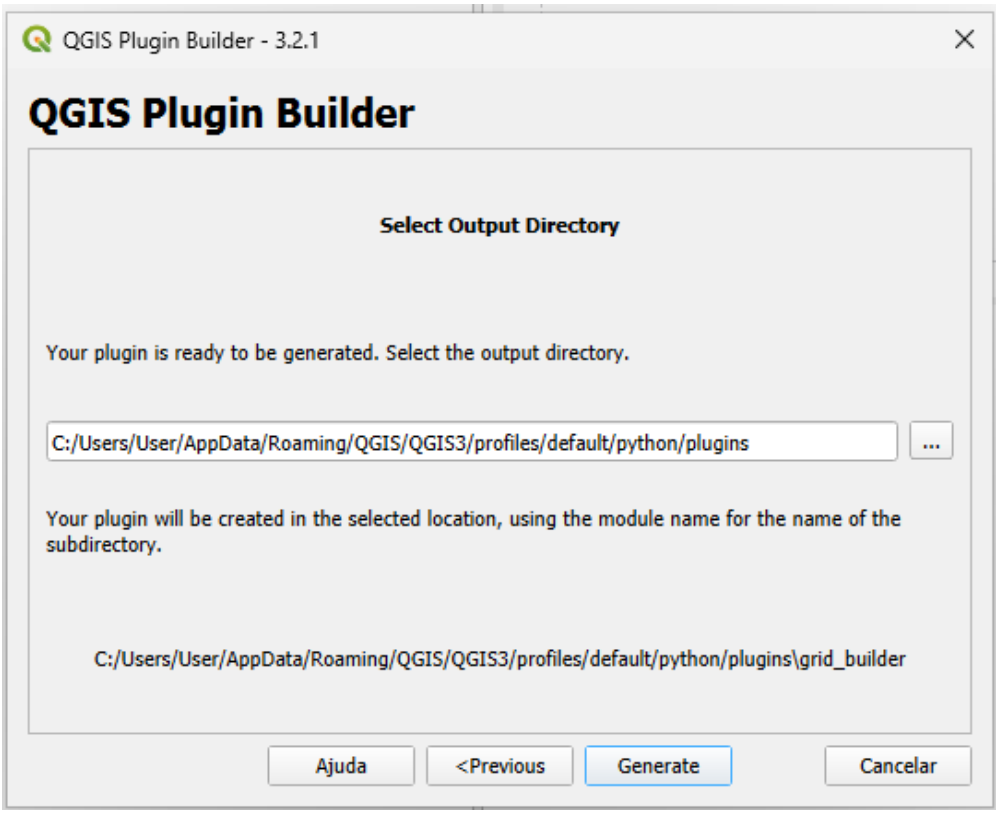
Template (tipo) do plugin, texto que vai aparecer no menu e em qual menu será listado o plugin.



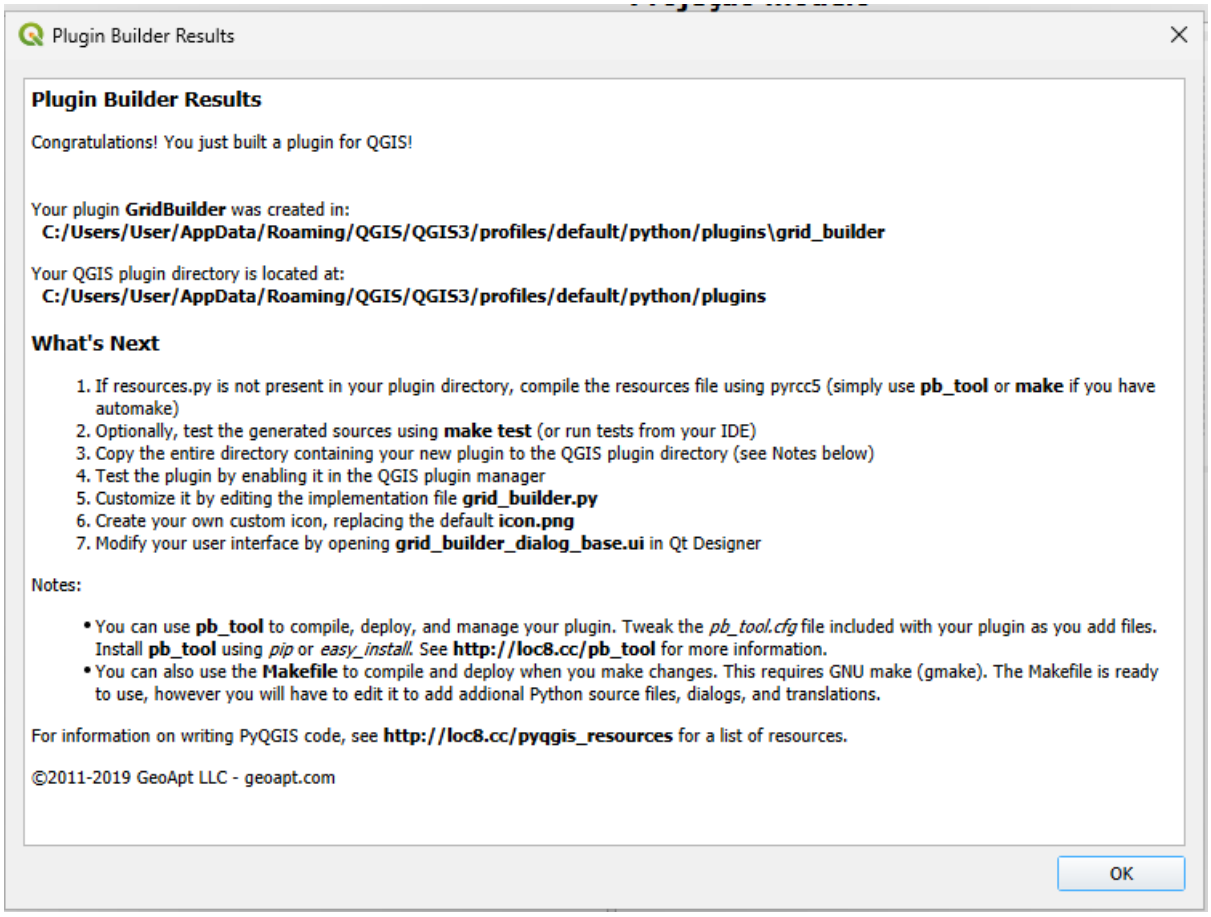
Desmarque todos para esse plugin,



Cheque a caixa de plugin experimental pois não iremos distribuir esse plugin no momento.

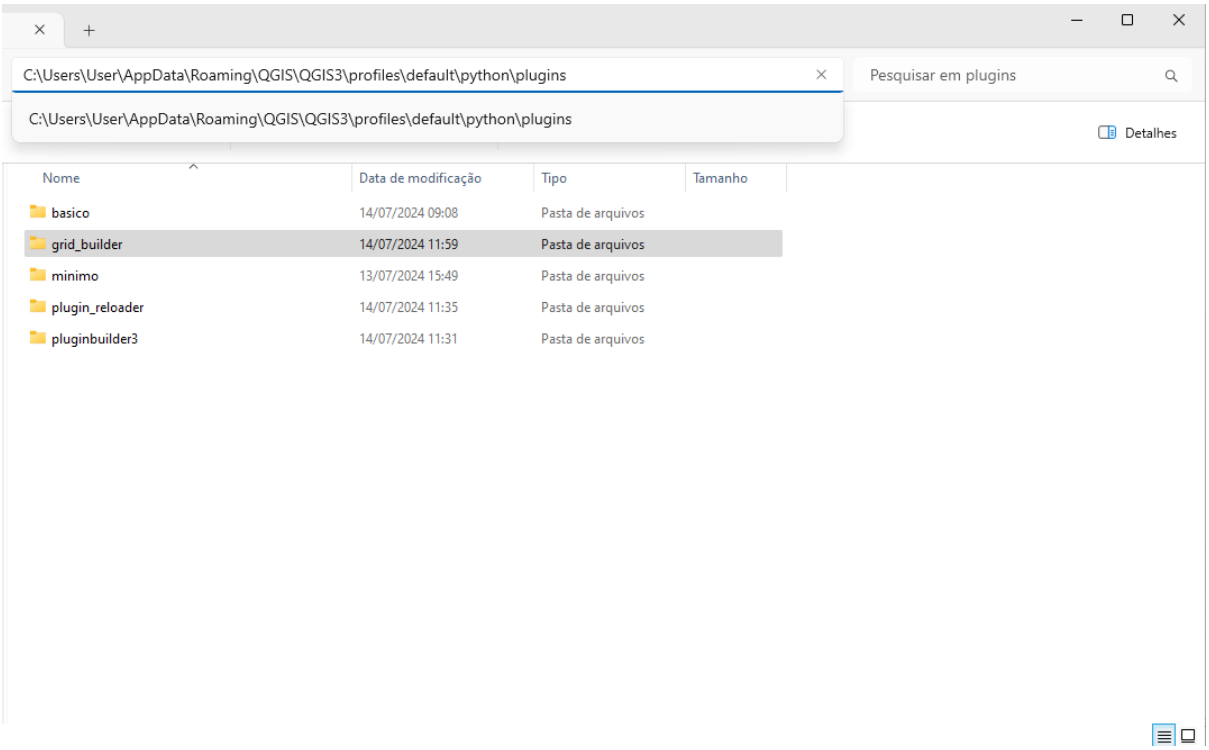


A pasta de plugin do sistema (nesse caso em sistema Windows). Clique **Generate** após selecionar o diretório de plugins.

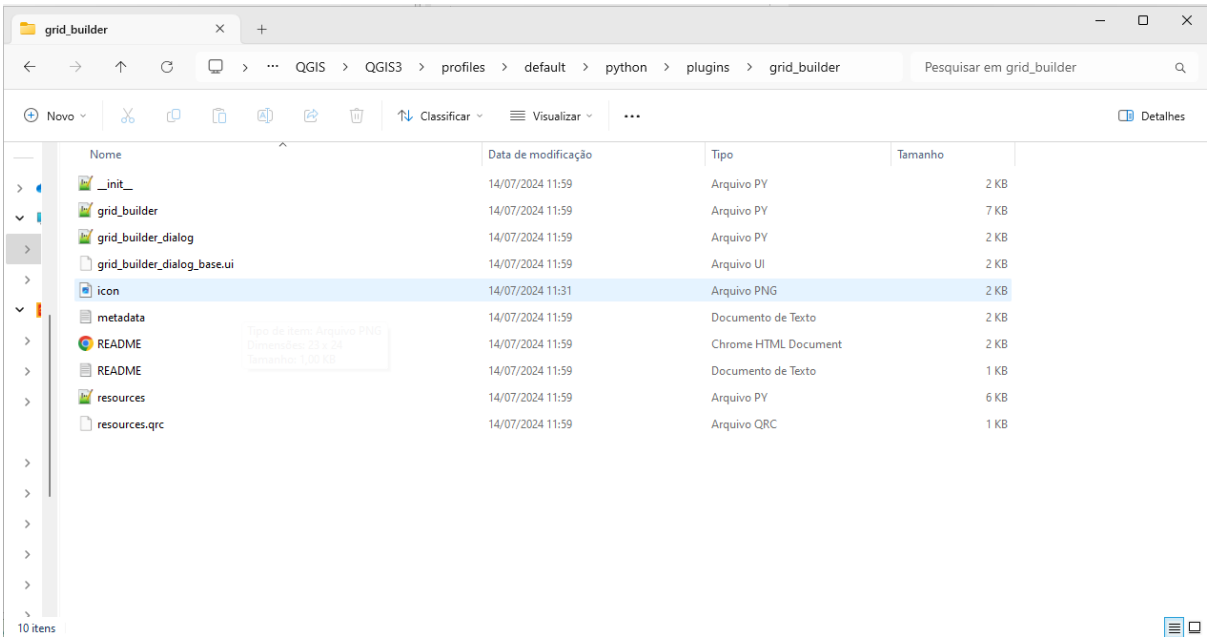


Pronto, os arquivos base de seu plugin foram criados na pasta:

C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\grid_builder



Os arquivos gerados:



Os oito arquivos necessários mais dois arquivos README com instruções do PluginBuilder foram criados automaticamente.

Com base no README gerado abaixo vamos aos próximos passos.

Plugin Builder Results

Congratulations! You just built a plugin for QGIS!

Your plugin **GridBuilder** was created in:
C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\grid_builder

Your QGIS plugin directory is located at:
C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins

What's Next

1. If resources.py is not present in your plugin directory, compile the resources file using pyrcc5 (simply use **pb_tool** or **make** if you have automake)
2. Optionally, test the generated sources using **make test** (or run tests from your IDE)
3. Copy the entire directory containing your new plugin to the QGIS plugin directory (see Notes below)
4. Test the plugin by enabling it in the QGIS plugin manager
5. Customize it by editing the implementation file **grid_builder.py**
6. Create your own custom icon, replacing the default **icon.png**
7. Modify your user interface by opening **grid_builder_dialog_base.ui** in Qt Designer

Notes:

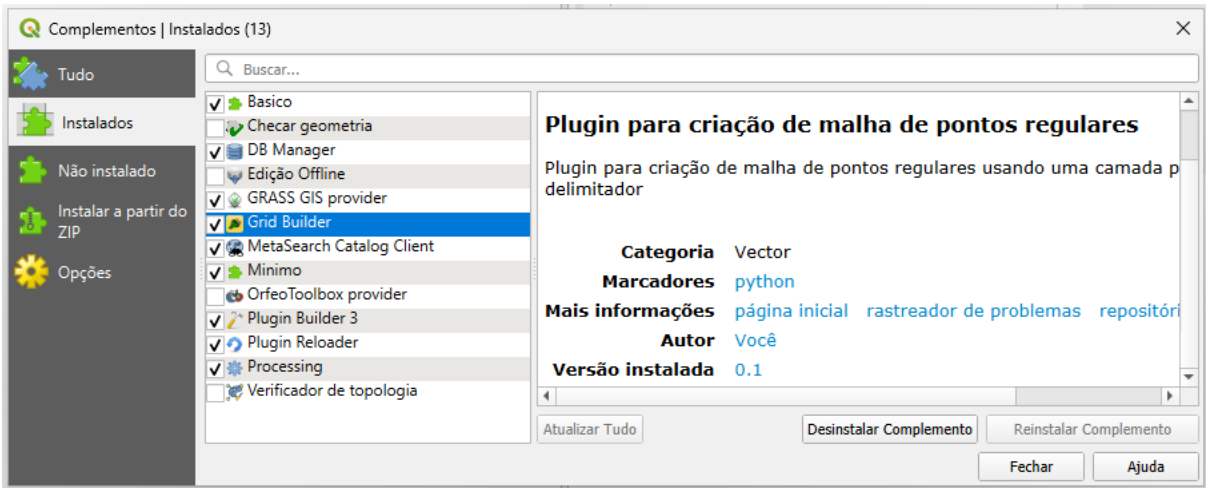
- You can use **pb_tool** to compile, deploy, and manage your plugin. Tweak the *pb_tool.cfg* file included with your plugin as you add files. Install **pb_tool** using *pip* or *easy_install*. See **http://loc8.cc/pb_tool** for more information.
- You can also use the **Makefile** to compile and deploy when you make changes. This requires GNU make (gmake). The Makefile is ready to use, however you will have to edit it to add addional Python source files, dialogs, and translations.

For information on writing PyQGIS code, see **http://loc8.cc/pyqgis_resources** for a list of resources.

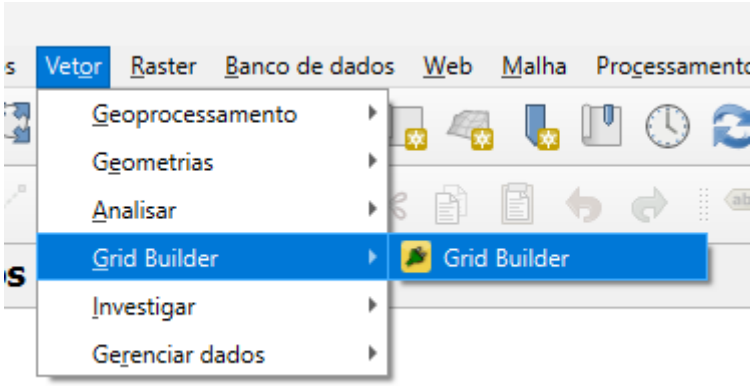
O arquivo resouces.py foi gerado automaticamente, desta forma não precisaremos de executar o pyrcc5, somente se quisermos mudar o icone padrão do PluginBuilder.

Os arquivos já estão no diretório correto de plugins.

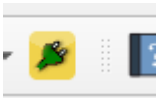
Vamos testar ele iniciando o QGIS e abrindo o **Complementos->Gerenciar e instalar Complementos**. Em Instalados vemos que ele não foi instalado ainda. Marque ele e instale para testarmos.



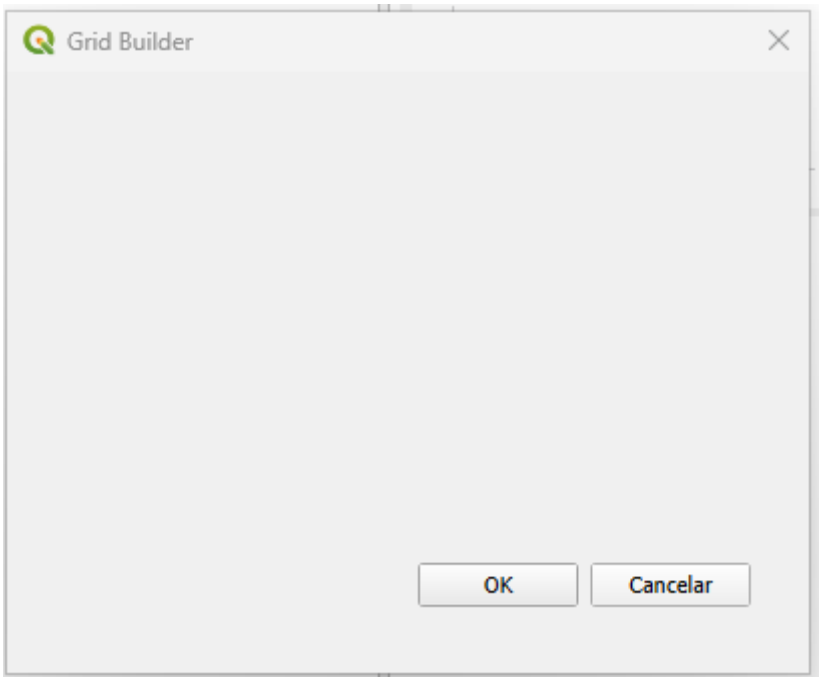
Inicie ele pelo Menu Vetor.



Ou pelo ícone na barra de ferramentas.



Funcionando, mas sem funcionalidade ainda.

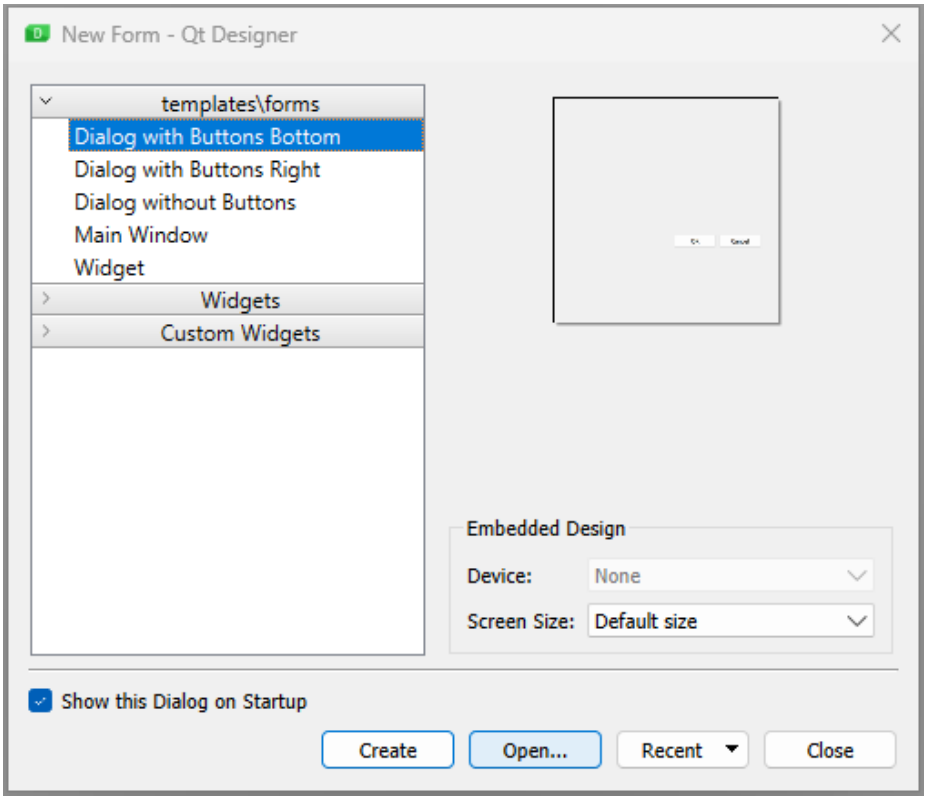


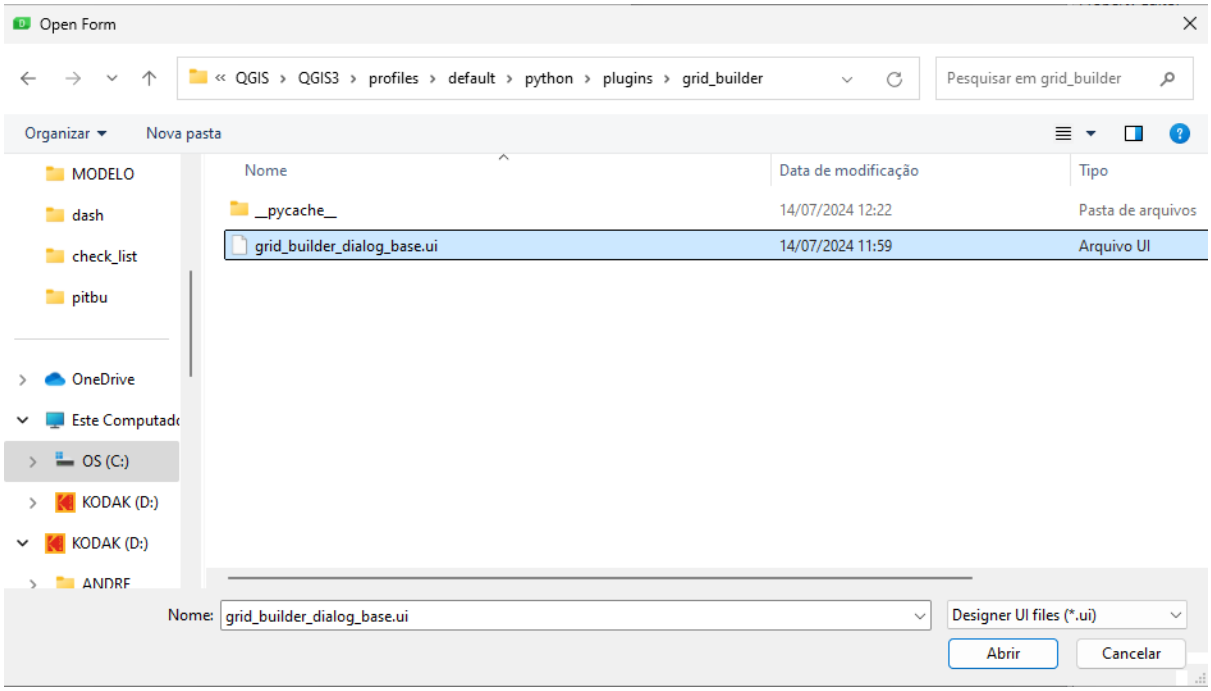
Vamos construir a interface gráfica do usuário (GUI) e adicionar a funcionalidade agora na próxima seção.

3 - O plugin Grid_Builder

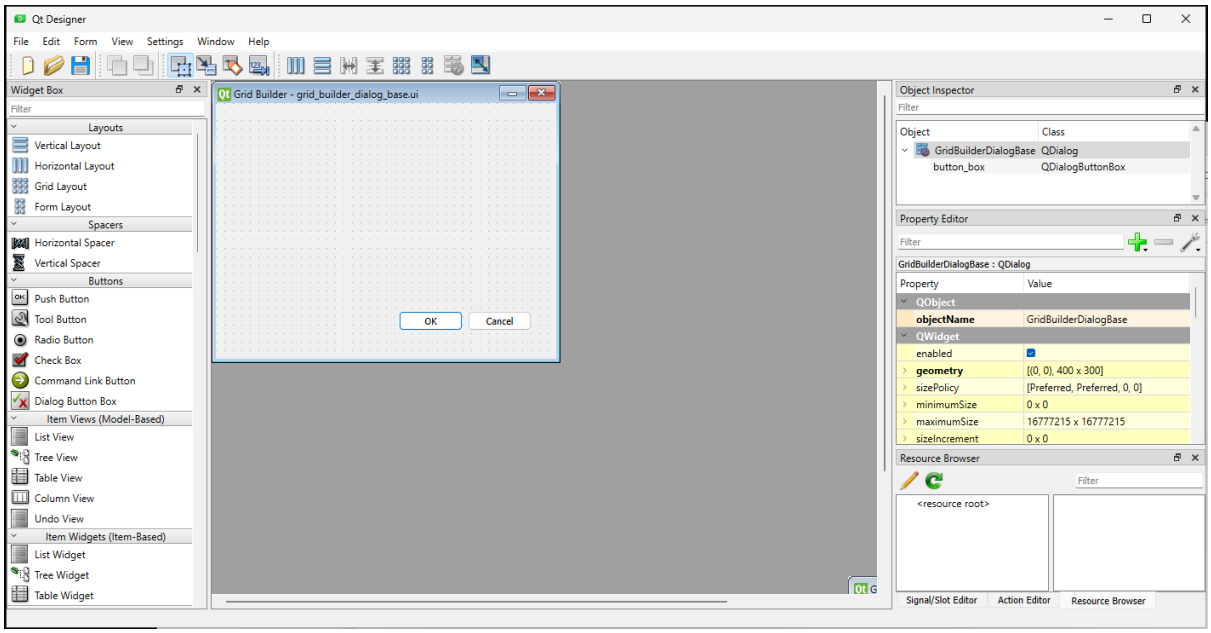
O plugin Grid Builder, como o próprio nome já fala constrói malha de pontos regulares usando uma camada espacial do tipo polígono para delimitar onde a malha será criada. Essa ferramenta, embora simples, é muito útil para criar malhas de amostragem.

Vamos iniciar pelo QtDesigner para criarmos a nossa interface gráfica de usuário (GUI). Abra o arquivo **grid_builder_dialog_base.ui** localizado em C:\Users\User\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\grid_builder no primeiro diálogo do programa em **Open.**

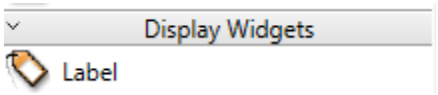


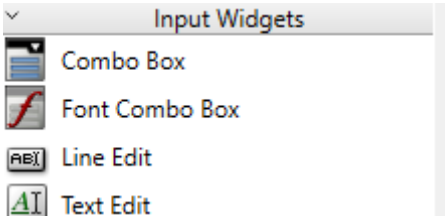


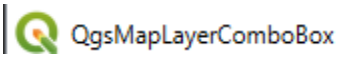
A seguinte tela aparecerá:



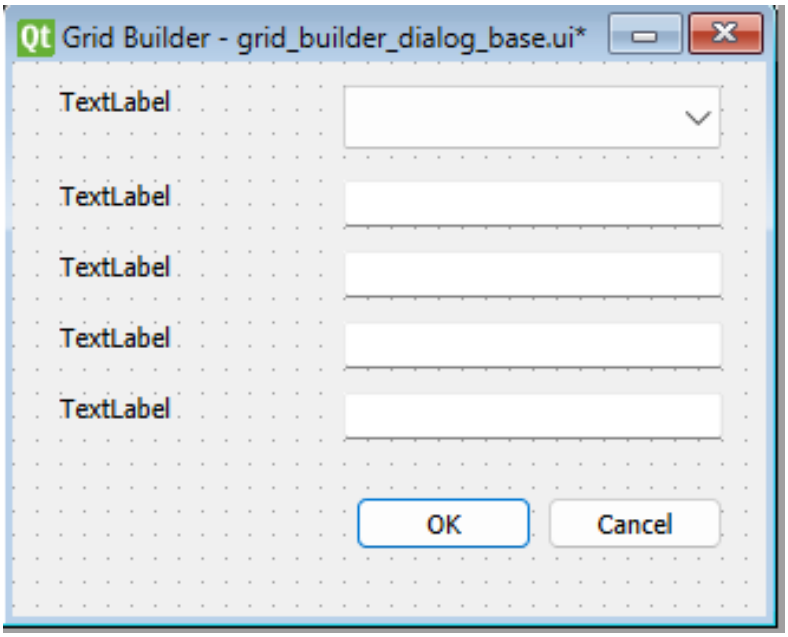
Vamos adicionar 5 widgets do tipo Label, 4 widgets do tipo Line Edit e um widget do tipo MapLayerComboBox. Basta clicar no Widget e arrastar até a janela do diálogo.



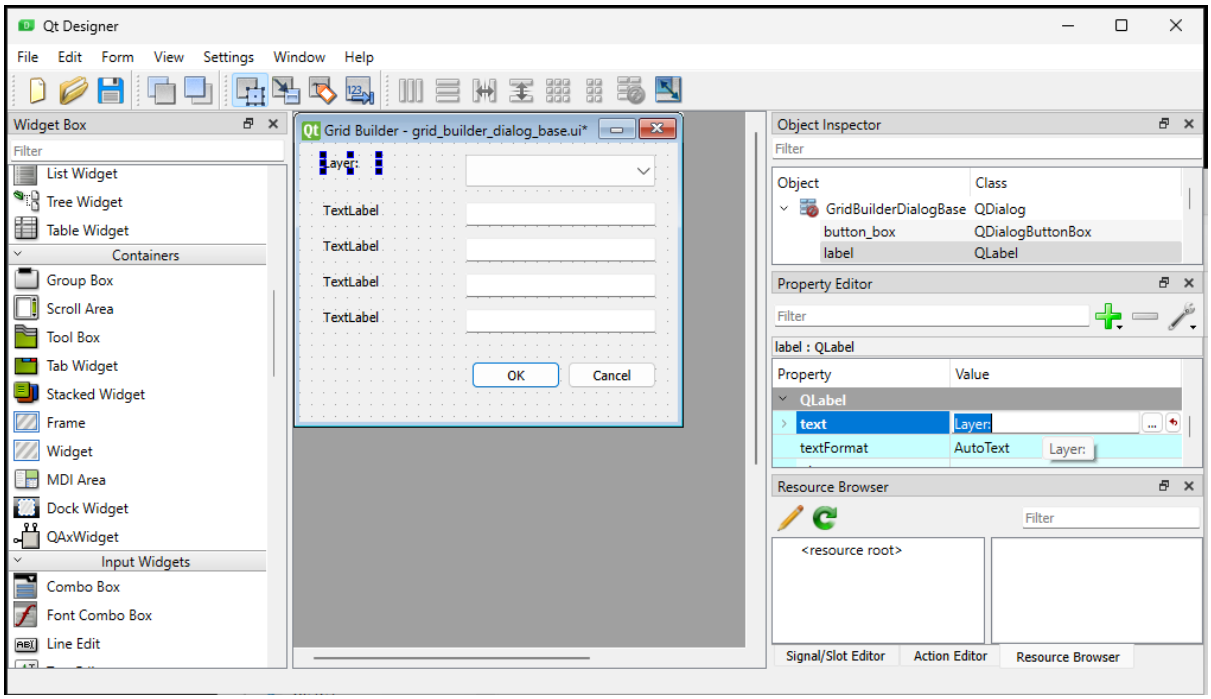




Teremos o diálogo mais ou menos com o seguinte formato:



Clique no Primeiro TextLabel e no property Editor altere o campo text de TextLabel para **Layer:** conforme abaixo:



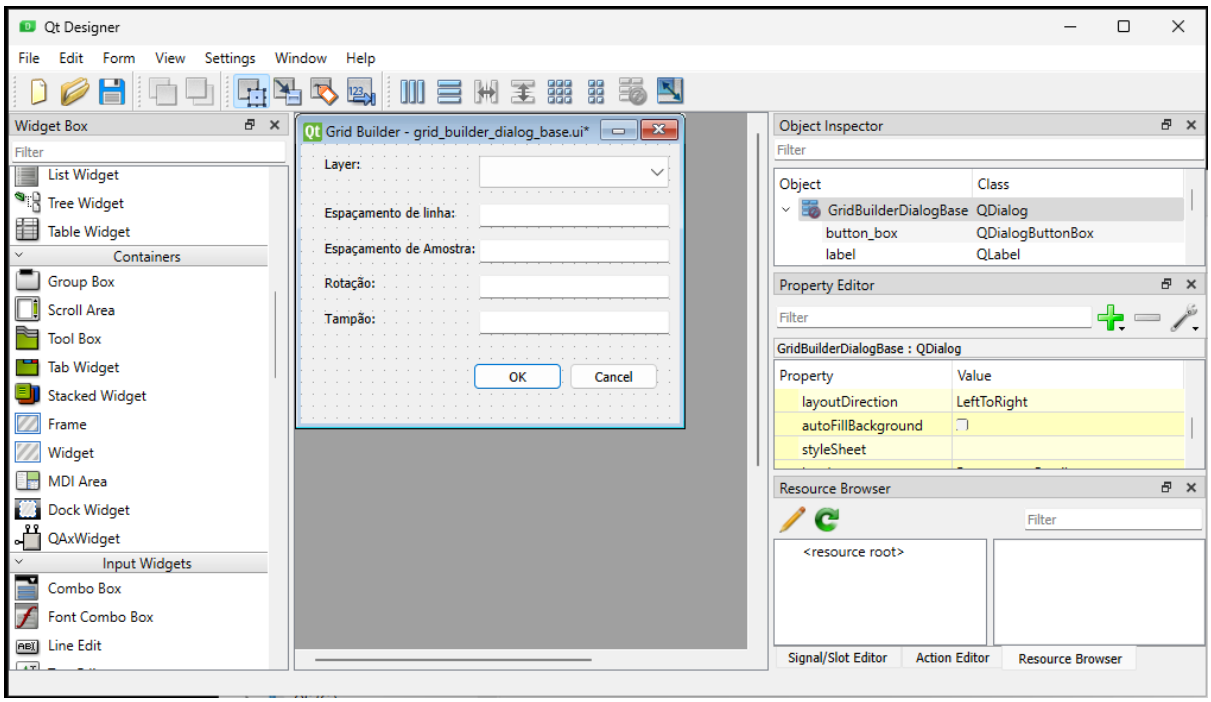
Repita o procedimento para os demais TextLabel renomeando eles para

Espaçamento de linha:

Espaçamento de Amostra:

Rotação:

Tampão:



O widget mMapLayerComboBox ficará inalterado, faremos a configuração pelo código python.

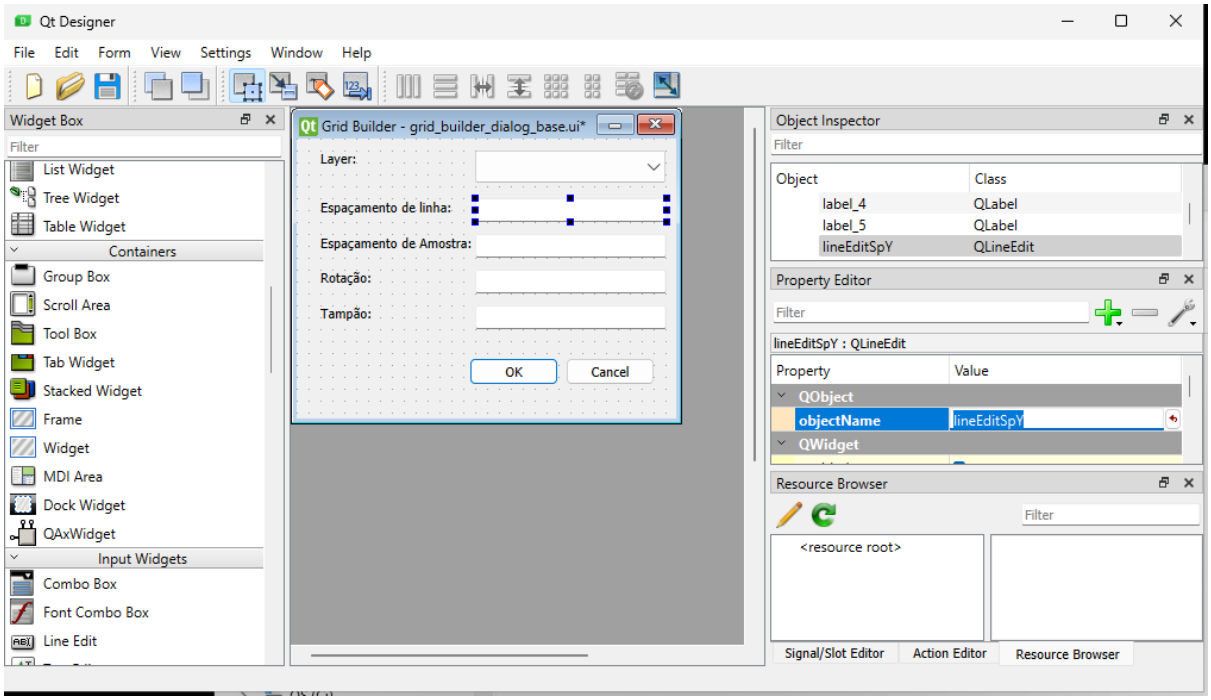
Agora altere a propriedade objectName dos campos LineEdit para:

lineEditSpY

lineEditSpX

lineEditRota

lineEditBuf



Pronto, salve o diálogo e feche o QtDesigner.

Vamos agora editar o arquivo **grid_builder.py** para realizar a tarefa. Vamos ter de adicionar algumas bibliotecas de suporte via **import** e adicionar o código na função **run** que vai fazer a validação inicial dos campos e a criação da camada de pontos da malha.

As bibliotecas serão (adicionar as faltantes):

```
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication
from qgis.PyQt.QtGui import QIcon, QIntValidator
from qgis.PyQt.QtWidgets import QAction,QMessageBox

# Initialize Qt resources from file resources.py
from .resources import *
# Import the code for the dialog
from .grid_builder_dialog import GridBuilderDialog
import os.path
from qgis.core import QgsProject, QgsPointXY, QgsGeometry, QgsVectorLayer,QgsFeature
from qgis.core import QgsPoint, QgsField, QgsMapLayerType, QgsWkbTypes
```

A função **run** ficará assim:

```
def run(self):
        if self.first_start == True:
            self.first_start = False
            self.dlg = GridBuilderDialog()
            self.dlg.mMapLayerComboBox.setShowCrs(True)
```

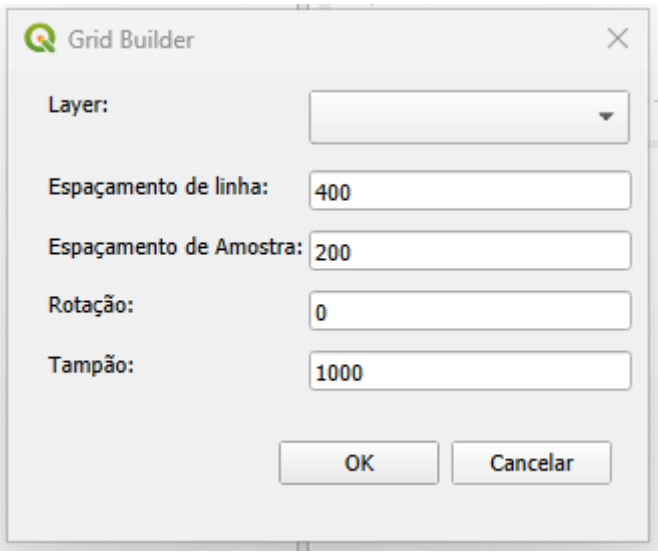
```
            self.map_layers = QgsProject.instance().mapLayers().values()
            self.allow_list = [
                                lyr.id() for lyr in self.map_layers if lyr.type() ==
QgsMapLayerType.VectorLayer
                                and lyr.geometryType()== QgsWkbTypes.PolygonGeometry
                            ]
            self.except_list = [l for l in self.map_layers if l.id() not in self.allow_list]
            self.dlg.mMapLayerComboBox.setExceptedLayerList(self.except_list)
            onlyInt = QIntValidator()# mudar para QDoubleValidator se for usar lat long também
            self.dlg.lineEditSpY.setText('400')
            self.dlg.lineEditSpY.setValidator(onlyInt)
            self.dlg.lineEditSpX.setText('200')
            self.dlg.lineEditSpX.setValidator(onlyInt)
            self.dlg.lineEditRota.setText('0')
            self.dlg.lineEditRota.setValidator(onlyInt)
            self.dlg.lineEditBuf.setText('1000')
            self.dlg.lineEditBuf.setValidator(onlyInt)

        self.dlg.show()
        result = self.dlg.exec_()
        if result:
            if self.dlg.lineEditSpY.text()=='' or self.dlg.lineEditSpX.text()=='' or
self.dlg.lineEditRota.text()=='' or self.dlg.lineEditBuf.text()=='':
                QMessageBox.warning(self.iface.mainWindow(),
                         'Erro',
                         "Entre todos os campos por favor\nSaindo...")
                return
            layer = self.dlg.mMapLayerComboBox.currentLayer()
            feats = [ feat for feat in layer.getFeatures()]
            points = []
            spacing_y = int(self.dlg.lineEditSpY.text())
            spacing_x = int(self.dlg.lineEditSpX.text())
            rotacao= int(self.dlg.lineEditRota.text())
            extensao= int(self.dlg.lineEditBuf.text())
            #executar o código
            #------------------
            #
            for feat in feats:
                centroid = feat.geometry().centroid().asPoint()
                extent = feat.geometry().boundingBox()
                xmin=int(round(extent.xMinimum()-extensao, -2))
                ymin=int(round(extent.yMinimum()-extensao, -2))
                xmax=int(round(extent.xMaximum()+extensao, -2))
                ymax=int(round(extent.yMaximum()+extensao, -2))
                rows = int(((ymax) - (ymin))/spacing_y)
                cols = int(((xmax) - (xmin))/spacing_x)
                x = xmin
                y = ymax
                geom_feat = feat.geometry()
                for i in range(rows+1):
                    for j in range(cols+1):
                        pt = QgsPointXY(x,y)
                        tmp_pt = QgsGeometry.fromPointXY(pt)
                        tmp_pt.rotate(rotacao, centroid)
                        if tmp_pt.within(geom_feat):
                            points.append(tmp_pt.asPoint())
                        x += spacing_x
                    x = xmin
                    y -= spacing_y

            epsg = layer.crs().postgisSrid()
            #gerando pontos
            uri = "PointZ?crs=epsg:" + str(epsg) + "&field=id:integer""&index=yes"
            mem_layer = QgsVectorLayer(uri,'gridpoints','memory')
            prov = mem_layer.dataProvider()
            feats = [ QgsFeature() for i in range(len(points)) ]
            for i, feat in enumerate(feats):
                feat.setAttributes([i])
                feat.setGeometry(QgsPoint(points[i].x(),points[i].y(),0.0))

            prov.addFeatures(feats)
            QgsProject.instance().addMapLayer(mem_layer)
            QMessageBox.information(self.iface.mainWindow(),
                         'Pronto',
                         "Pontos de amostragem criados!")
            Return
```

Antes de comentarmos o código adicionado vamos testar o plugin. Abra o QGIS e o plugin será carregado já com as alterações feitas. Ao iniciarmos o plugin teremos:

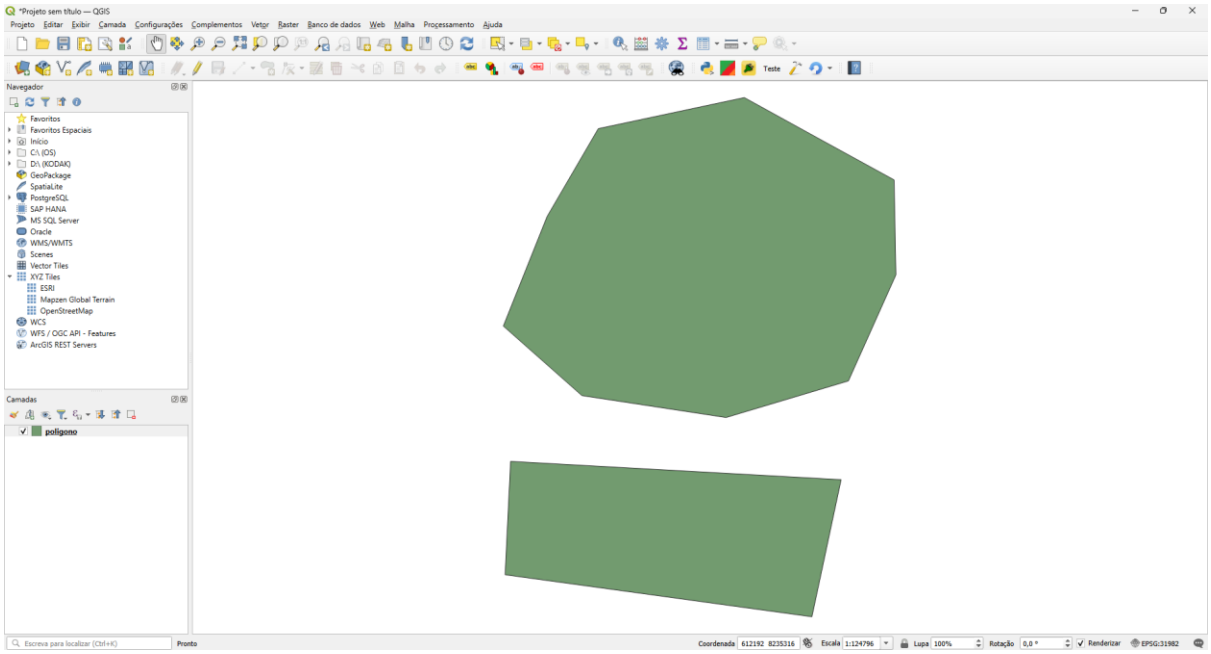


Primeiro crie ou abra uma camada do tipo polígono para executar o plugin Grid Builder.

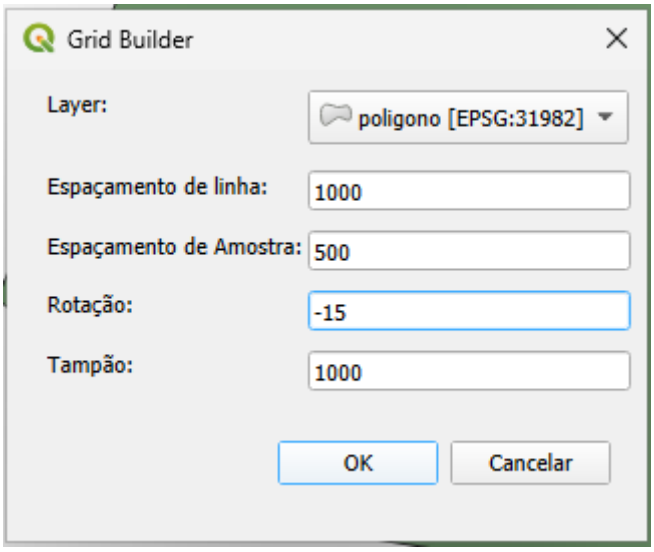
Clique em cancelar e crie um polígono (em coordenadas UTM, lat long funciona, mas os espaçamentos e o tampão devem ser entrados como decimal de grau e os campos de entrada validam somente inteiros, para aceitar decimais é preciso mudar o “validator” para QDoubleValidator no código).

Caso prefira carregue a camada polígono do site em :
https://gdatasystems.com/pyqgis/index.php

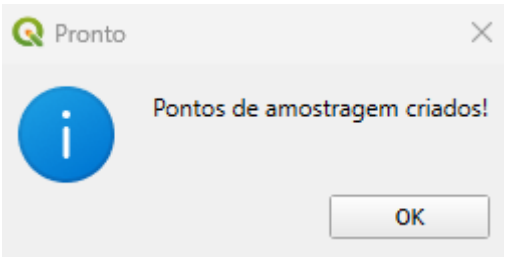
Camada polígono carregada.

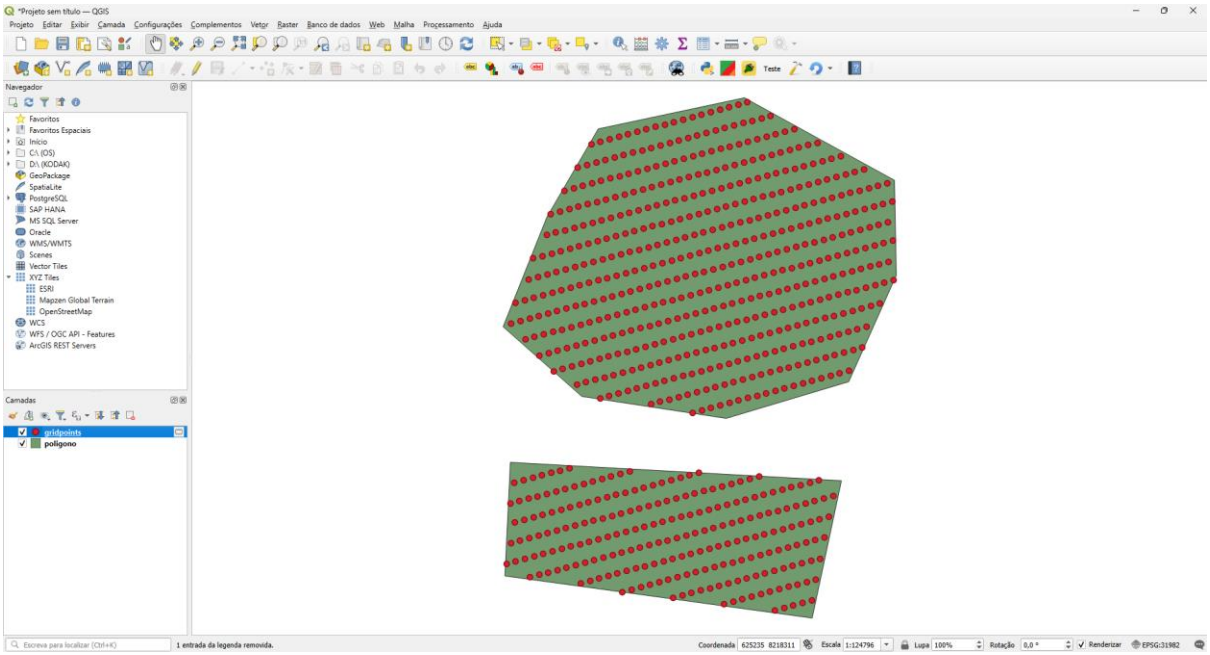


Execute o plugin com os seguintes parâmetros:



A seguinte mensagem aparece e a camada temporária gridpoint é criada.





Agora é só salvar a camada criada.

Este bloco do código inicializa os widgets do plugin carregando os polígonos abertos no combo box e adicionando os valores iniciais nas linhas editáveis.

```
def run(self):
        if self.first_start == True:
            self.first_start = False
            self.dlg = GridBuilderDialog()
            self.dlg.mMapLayerComboBox.setShowCrs(True)
            self.map_layers = QgsProject.instance().mapLayers().values()
            self.allow_list = [
                                lyr.id() for lyr in self.map_layers if lyr.type() ==
QgsMapLayerType.VectorLayer
                                and lyr.geometryType()== QgsWkbTypes.PolygonGeometry
                            ]
            self.except_list = [l for l in self.map_layers if l.id() not in self.allow_list]
            self.dlg.mMapLayerComboBox.setExceptedLayerList(self.except_list)
            onlyInt = QIntValidator()mudar para QDoubleValidator se for usar lat long também
            self.dlg.lineEditSpY.setText('400')
            self.dlg.lineEditSpY.setValidator(onlyInt)
            self.dlg.lineEditSpX.setText('200')
            self.dlg.lineEditSpX.setValidator(onlyInt)
            self.dlg.lineEditRota.setText('0')
            self.dlg.lineEditRota.setValidator(onlyInt)
            self.dlg.lineEditBuf.setText('1000')
            self.dlg.lineEditBuf.setValidator(onlyInt)
```

O bloco seguinte checa se os campos estão todos preenchidos e assinala eles às variáveis do programa.

```
        if result:
            if self.dlg.lineEditSpY.text()=='' or self.dlg.lineEditSpX.text()=='' or
self.dlg.lineEditRota.text()=='' or self.dlg.lineEditBuf.text()=='':
                QMessageBox.warning(self.iface.mainWindow(),
                         'Erro',
                         "Entre todos os campos por favor\nSaindo...")
                return
            layer = self.dlg.mMapLayerComboBox.currentLayer()
            feats = [ feat for feat in layer.getFeatures()]
            points = []
            spacing_y = int(self.dlg.lineEditSpY.text())
            spacing_x = int(self.dlg.lineEditSpX.text())
            rotacao= int(self.dlg.lineEditRota.text())
            extensao= int(self.dlg.lineEditBuf.text())
```

Aqui lemos o polígono e assinalamos sua extensão, calculamos o buffer e checamos qual sistema de coordenada devemos usar para gerar os pontos.

```
            #executar o código
            #------------------
            #
            for feat in feats:
                centroid = feat.geometry().centroid().asPoint()
                extent = feat.geometry().boundingBox()
                xmin=int(round(extent.xMinimum()-extensao, -2))
                ymin=int(round(extent.yMinimum()-extensao, -2))
                xmax=int(round(extent.xMaximum()+extensao, -2))
                ymax=int(round(extent.yMaximum()+extensao, -2))
                rows = int(((ymax) - (ymin))/spacing_y)
                cols = int(((xmax) - (xmin))/spacing_x)
                x = xmin
                y = ymax
```

Finalmente criamos os pontos e carregamos eles em um arquivo na memória.

```
geom_feat = feat.geometry()
    for i in range(rows+1):
        for j in range(cols+1):
            pt = QgsPointXY(x,y)
            tmp_pt = QgsGeometry.fromPointXY(pt)
            tmp_pt.rotate(rotacao, centroid)
            if tmp_pt.within(geom_feat):
                points.append(tmp_pt.asPoint())
            x += spacing_x
        x = xmin
        y -= spacing_y

epsg = layer.crs().postgisSrid()
uri = "PointZ?crs=epsg:" + str(epsg) + "&field=id:integer""&index=yes"
mem_layer = QgsVectorLayer(uri,'gridpoints','memory')
prov = mem_layer.dataProvider()
feats = [ QgsFeature() for i in range(len(points)) ]
for i, feat in enumerate(feats):
    feat.setAttributes([i])
    feat.setGeometry(QgsPoint(points[i].x(),points[i].y(),0.0))

prov.addFeatures(feats)
QgsProject.instance().addMapLayer(mem_layer)
QMessageBox.information(self.iface.mainWindow(),
             'Pronto',
             "Pontos de amostragem criados!")
Return
```

No próximo módulo avançaremos um pouco mais com um novo exemplo de plugin que usa uma biblioteca externa, não padrão do QGIS. Até lá!