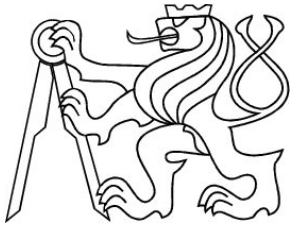
České vysoké učení technické v Praze
Fakulta stavební



DIPLOMOVÁ PRÁCE

Workflow builder pro Quantum GIS

Vypracoval: Zdeněk Růžička
Vedoucí práce: Ing. Martin Landa
Rok: Praha, 2012

Prohlášení

Prohlašuji, že jsem svou diplomovou práci na téma **Workflow Builder** vypracoval samostatně s pomocí svého vedoucího práce a za použití literatury a zdrojů uvedených v přiloženém seznamu v závěru práce.

V Praze dne _____ _____

podpis

Poděkování

Především děkuji vedoucímu mé diplomové práce Ing. Martinu Landovi, Ph.D. za odborné vedení, rychlé reakce na mé dotazy a ochotu hledat na ně odpovědi. Dále bych chtěl poděkovat Camilo Polimeris za napsání QGIS Processing Frameworku, jehož je tato práce součástí. V neposlední řadě bych chtěl poděkovat rodině a kamarádům za důvěru a podporu během studií.

Abstrakt

Diplomová práce si vytyčila za cíl vytvořit v prostředí Quantum GIS (dále jen QGIS) nástroj, který by umožňoval uživateli grafické propojování modulů z frameworku **QGIS Processing Framework**. V úvodní kapitole je představena knihovna Qt, resp. její verze PyQt pro jazyk Python, ve které byl celý **Workflow Builder** napsán. Dále je představeno prostředí QGIS a popsána práce s QGIS Processing Frameworkem.

V druhé kapitole diplomové práce je představena samotná aplikace Workflow Builder.

V poslední kapitole je zmínka o frameworku **SEXTANTE**, který se objevil v konci psaní této práce.

Klíčová slova

Quantum GIS, QGIS, workflow, open source, GIS, PyQt, QGIS Processing Framework

Abstract

Key words

Quantum QGIS, workflow, open source, GIS, PyQt, SAGA, QGIS Processing Framework

Obsah

Úvod

V dnešní době se můžeme setkat s geoinformačními (GIS) technologiemi doslova na každém kroku. V různých oblastech krajinného inženýrství, při plánování výstavby silnic, v územním plánování, při řešení krizových situacích či plánování záchranných akcí. Uživatel si může vybrat z nepřeberného množství již existujících GIS nástrojů. A s potěšením lze konstatovat, že svobodná řešení, nejen v oblasti geoinformačních technologií, drží krok s těmi proprietárními. Uživatel tedy nemusí sahat hluboko do kapsy. Co se týče nástrojů pro prohlížení, zpracování a analýzu geodat, můžeme jmenovat například GRASS GIS, gvSIG, Quantum GIS či SAGA GIS. Tato práce si ale nekladla za cíl srovnat GIS nástroje, ale implementaci nástroje do programu Quantum GIS, který by uživateli umožňoval vytvářet vlastní funkce s využitím již existujících funkcí.

Můžeme se také setkat s pojmy jako model builder či chaining. V této práci bude používán pojem workflow builder . Tento název byl převzat z projektu VisTrails, který byl inspirací pro grafickou část **Workflow Builder**u. Takzvané workflow buildery dávají uživateli možnost vytvářet si vlastní moduly za pomocí spojování výstupů a vstupů modulů již existujících. Uživatel tak nemusí spouštět každý modul zvlášť a starat se o výstupy, nová data, která se vytvoří jen dočasně a která uživatel v konečném výsledku nepotřebuje. Dále je pro uživatele také mnohem pohodlnější, může-li najít všechny funkce na jednom místě (tzv. toolbox).

V době psaní této diplomové práce existoval projekt **QGIS Processing Framework** studenta Camilo Polymeris z univerzity Universidad de Concepción. QGIS Processing Framework si kladl za cíl být frameworkem, který by sdružoval moduly z pluginů pro QGIS na jednom místě. Odtud by byly jednotlivé moduly volány, pomocí

workflow builderu spojovány, ukládány atp. V rámci tohoto projektu začala vznikat podpora pro použití modulů z jiného GIS nástroje - SAGA GIS. V době psaní této práce avizována podpora pro 170 modulů, ne všechny ale byly testovány a fungovaly správně. I přesto se mohlo začít s prací na workflow builderu.

Aktuální verzi workflow builderu můžeme najít zde:

`https://github.com/CzendaZdenda/qgis`

Ilustrační video pro práci s **Workflow Builder**em zde:

`http://youtu.be/4PxvWvTIyaU`

Kapitola 1

Teorie

V první části této kapitoly představím programovací jazyk Python, ve kterém byl **Workflow Builder** napsán. Jazyk Python je v dnešní době stále více oblíbený a můžeme ho najít snad téměř všude. V druhé části této kapitoly představím jeden ze svobodných systémů pro práci s geografickými daty - Quantum GIS a možnost rozšiřování jeho funkcionality pomocí zásuvných modulů, tzv. pluginů. To bylo původně možné jen v jazyce C++. Již nějakou dobu je ale také možné psát pluginy v jazyce Python, což přineslo výhody v podobě jednoduché šiřitelnosti (není nutná kompilace) a snazšího vývoje pluginů. V další části se budu věnovat knihovně Qt, respektive její verzi pro jazyk Python - PyQt4. Zde popíši nástroje, které jsem využil při psaní Workflow Builderu. Mezi tyto nástroje patří hlavně implementace architektury **MVC** v podobě model-view-delegate, **Graphics View Framework** pro vykreslování a správu dvojrozměrných grafických prvků, **signály** a **sloty** pro komunikaci mezi objekty knihovny Qt a mechanismus **Drag and Drop**. V předposlední části popíši projekt **QGIS Processing Framework**, co bylo jeho cílem, jeho koncem a také možnosti jeho rozšiřování. V poslední části představím modul **xml.dom**, resp. jeho odlehčenou verzi **xml.dom.minidom**, jazyka Python pro práci s objekty ve formátu XML.

1.1 Python

Python je objektově orientovaný a interpretovaný programovací jazyk s dynamickým a silným typováním. Python je charakteristický pro své vyjadřování struktury kódu pomocí odsazování, jehož dodržování je povinné. To vede k čitelnějšímu a přehlednějšímu kódu.

První verze jazyka byla uvolněna v roce 1991. Python navrhl Guido van Rossum, který byl inspirován jazyky jako C++ či Perl. Jedná se o **open source** projekt dostupný pod licencí Python Software Foundation License, která je kompatibilní s GPL licencí. Rozdíl je v tom, že u Python Software Foundation License můžeme měnit kód bez nutnosti zveřejnit změny jako open source [viz [1]]. Aktuální stabilní verze jsou 2.7.3 a 3.2.3 pro verzi Python 3.0, která byla uvolněna v roce 2008.

V dnešní době se s Pythonem můžeme setkat téměř všude. Pro jeho jednoduchost při psaní kódu je velmi populární a široce rozšířený. Jeho velká výhoda je tedy velmi čitelný kód a rychlost psaní. Jazyk je jednoduše přenositelný (není nutná kompilace), je multiplatformní a má kvalitní dokumentaci.

V mnoha projektech existuje skriptovací rozhraní pro Python. Jako příklad uveďme QGIS, GIMP, Inkscape, Scribus, LibreOffice, Blender nebo ArcGIS. Můžeme jej najít v projektech jako Maya, OpenShot Video Editor, Wammu, DopBox, MapServer či Gajim. Používá se pro psaní grafického rozhraní. Existují verze grafických knihoven Qt a GTK pro Python - PyQt, resp. PyGTK. Využívá se jako skriptovací jazyk pro psaní webových aplikací. V síti Internet se s ním můžeme také setkat v podobě nástrojů jako Django, Zope či Pylons. Byly napsány knihovny pro vědecké výpočty - NumPy, SciPy, Matplotlib.

[1] http://docs.python.org/license.html

PyQGIS

Quantum GIS nabízí podporu QGIS API pro jazyk Python. Jedná se o verzi PyQGIS. PyQGIS můžeme používat přímo v QGISu přes příkazovou řádku, psát zásuvné moduly či využít QGIS API pro náš vlastní program.

Existuje velmi dobře zpracovaná tzv. kuchařka jak psát v PyQGIS [5], tudíž nepovažuji za nutné se mnoho rozepisovat. Zmíním jen pár informací, které považuji za základní. Rastrové a vektorové vrstvy jsou reprezentovány třídami *QgsRasterLayer* a *QgsVectorLayer*. Třída *QgsMapLayerRegistry* slouží k načítání, správu a mazání geografických vrstev z QGISu. Třída se nachází v **qgis.core** knihovně. Pomocí příkazu **QgsMapLayer.instance().mapLayers()** získáme všechny vrstvy momentálně načtené v QGIS. Aktivní vrstvu získáme pomocí příkazu **qgis.utils.iface.activeLayer()**. Vrstvu přidáme do QGISu pomocí **QgsMapLayer.instance().addMapLayer(layer)** a odebereme **QgsMapLayer.instance().removeMapLayer(layer)**.

Vektorovou vrstvu vytvoříme takto **QgsVectorLayer**(cesta_k_souboru, jmeno_vrstvy, knihovna).

Například:

```
1 vector = QgsVectorLayer("~/geodata/contour.shp", "ContourLines", "ogr")
```

Rastrovou vrstvu vytvoříme takto **QgsRasterLayer**(cesta_k_souboru, jmeno_vrstvy).

Například:

```
1 raster = QgsRasterLayer("~/geodata/elevation.tiff", "Elevation")
```

Z mapových vrstev můžeme získat například souřadnicový systém pomocí metody *crs*(), zdrojový soubor metodou *source*(), jménou *name*() či rozsah *exent*().

Kompletní dokumentace QGIS API je dostupná zde [9].

1.2 Quantum GIS

Součástí Quantum GIS projektu jsou:

- QGIS Desktop - desktopová aplikace pro práci s geografickými daty (geodaty)
- QGIS Browser - rychlá a jednoduchá prohlížečka geodat, podporuje také prohlížení dat dostupných přes službu WMS
- QGIS Server - mapový server
- QGIS Client - webový klient založený na QGIS Server a knihovně GeoExt

Označením Quantum GIS (dále QGIS) se většinou myslí aplikace QGIS Desktop, v následujícím textu tomu nebude jinak.

QGIS je nejen prohlížečka geografických dat dostupná pro řadu platforem jako MS Windows, GNU/Linux, či Mac OS X, ale díky zásuvným modulům také velmi mocný nástroj pro práci s geografickými daty. QGIS podporuje díky knihovně OGR většinu vektorových formátů dat jako například ESRI Shapefile, GRASS, MapInfo či GML a díky knihovně GDAL mnoho rastrových formátů jako TIFF, ArcInfo, GRASS raster, ERDAS a další. Přes QGIS můžeme také přistupovat k datům uložených v geodatabázích PostGIS a SpatiaLite či k datům dostupných přes WMS a WFS služby.[2] QGIS je šířen pod licencí GNU Public Licence.

Program je psán v jazyce C++. Poslední stabilní verze nese označení 1.7.4. QGIS je jednoduše rozšiřitelná aplikace pomocí zásuvných modulů, tzv. pluginů. Pluginy mohou

[2] http://qgis.org/about-qgis/features.html

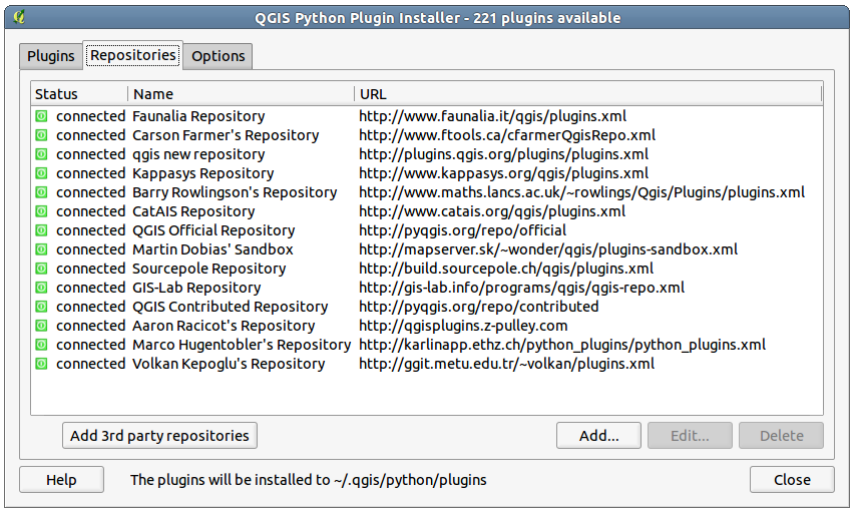
být psány v jazyce C++ nebo Python. QGIS má poměrně dobře zdokumentované API a nutno také podotknout, že komunita kolem QGIS je aktivní a podpora prostřednictvím mailing listů je na velmi vysoké úrovni.

Systém začal vyvíjet v roce 2002 Gary Sherman. Mělo jít o nenáročnou prohlížečku geodat pro operační systém GNU/Linux s širokou podporou datových formátů. Dlouhou dobu byl QGIS brán převážně jako grafická nadstavba pro jiný desktopový GIS - GRASS GIS. Přes GRASS Plugin je zpřístupněna řada modulů GRASS GIS.

Jak už bylo zmíněno, funkcionalitu QGIS rozšiřuje množství pluginů. Jako základní pluginy bych označil [3] **fTools** , který umožňuje pokročilé prostorové analýzy nad vektorovými daty, [4] **GdalTools** pro práci s rastrovými daty a již zmíněný [5] **GRASS Plugin** plugin, který zpřístupňuje funkce GRASSu uživatelům Quantum GIS.

V současnosti se na vývoji nejvíce podílí skupina vývojářů kolem organizace [6] Faunalia.

1.2.1 Správa pluginů



Obr. 1.1: QGIS Python Plugin Installer - správa repositářů

QGIS umožňuje uživatelům rozšiřovat funkce programu dle jejich potřeb v podobě

[3] http://www.ftools.ca/
[4] http://www.faunalia.co.uk/gdaltools
[5] http://grass.osgeo.org/wiki/GRASS_and_QGIS
[6] http://www.faunalia.co.uk/quantumgis

zásuvných modulů. Díky dobře zdokumentovanému API může uživatel pohodlně psát pluginy v jazyce C++ nebo Python. Pluginy píší jak vývojáři Quantum GISu, tak i obyčejní uživatelé. Pluginy si můžeme stáhnout z oficiálních či neoficiálních repositářů. Pro instalování pluginů napsaných v jazyce Python a správu repositářů slouží nástroj **QGIS Python Plugin Installer**, dostupný přes *Plugins* → *Fetch Python Plugins...*. Jak je vidět z [Obr.1.1], takto nainstalované pluginy se uloží do adresáře:

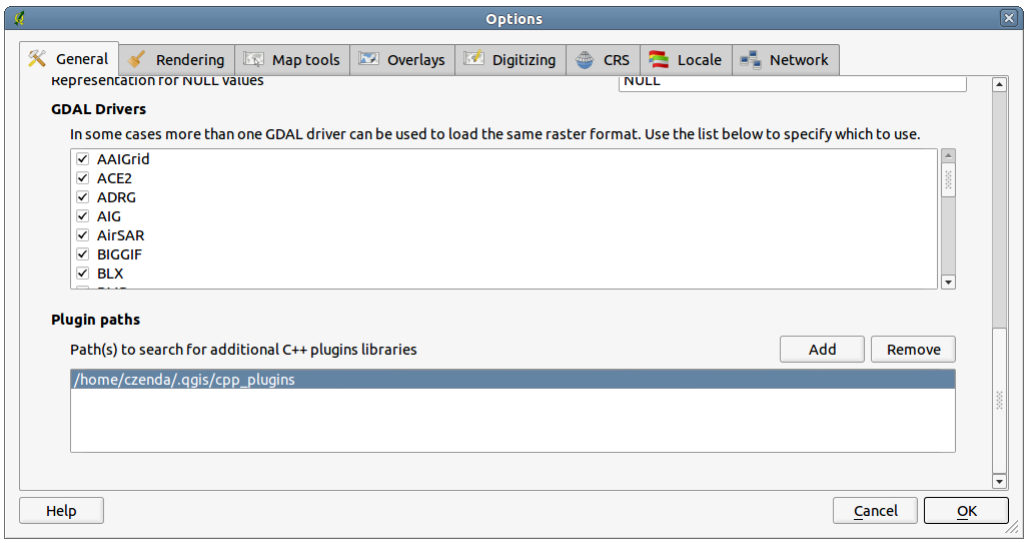
- *$HOME\.qgis\python\plugins* - v případě OS GNU/Linux
- *C:\Documents and Settings\USER\.qgis\python\plugins* - v případě OS Windows bývá cesta podobná této.

V případě, že uživatel napíše plugin v jazyce Python, doporučuje se ho uložit do výše uvedeného adresáře. Je zde také možnost uložit plugin do adresáře *$QGIS_INSTALL_DIR \share\ qgis\python\plugins*, ale při případné opětovné kompilaci by byly změny pravděpodobně ztraceny.
Pluginy psané v jazyce C++ se po přeložení ukládají standardně v *$QGIS_INSTALL_DIR \lib\qgis\plugins*. Uživatel má také možnost nastavit nová úložiště pro svoje pluginy pomocí *Settings* → *Options* a v záložce *Generals* zadat cestu [Obr.1.2].

Všechny nainstalované pluginy, ať psané v jazyce C++ či Python, může uživatel spravovat přes **QGIS Plugin Manager** - *Plugins*→*Manage Plugins...* [Obr.1.3].

1.2.2 Psaní vlastního pluginu

Pluginy mohou být psány v jazyce C++ a Python. Již z charakteristiky daných jazyků vyplývá, že pro jednoduché, nenáročné či na začátku vývoje pluginu, se bude hodit spíše jazyk Python, který se nemusí kompilovat a píše se v něm rychleji než v jazyce C++. Pro rozsáhlejší projekty je lepší použít jazyk C++, protože obecně jsou programy psané v kompilovaných jazycích mnohem rychlejší než programy psané v jazycích interpretovaných.



Obr. 1.2: *Settings→Options→Generals* - přidání nové cesty k pluginům psaných v jazyce C++

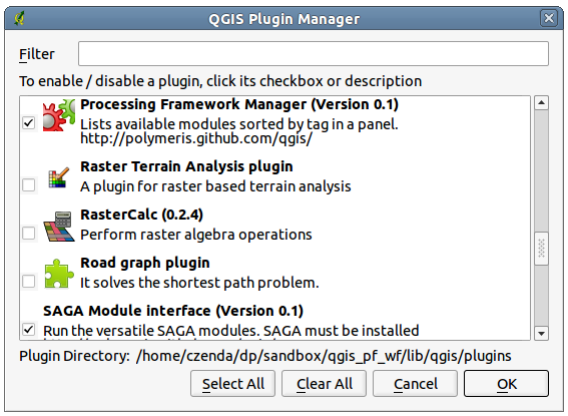
1.2.3 Python plugin

Chceme-li psát plugin v Pythonu, budeme k tomu potřebovat mít nainstalovaný **QGIS**, **Python** minimálně verze 2.5, **Qt** a její pythoní verzi **PyQt**. Při psaní pluginu v jazyce Python využíváme nástroje PyQGIS. Kromě dokumentace k QGIS API [9] také doporučuji kuchařku, kde nalezneme, jak psát pomocí PyQGIS [5]. Nejjednodušší možnost jak začít psát svůj plugin se jeví využít nástroj **Plugin Builder**. **Plugin Builder** je plugin, který vygeneruje základní soubory s kódem. Ty potom můžeme začít upravovat.

Základní soubory jsou:

- *__init__.py* - inicializační soubor
- *plugin.py* - hlavní soubor pluginu

Tyto dva výše zmíněné soubory jsou dostačující k tomu, aby se náš plugin objevil v **Plugin Manager**u. Dále budeme potřebovat nějaké grafické rozhraní pro náš plugin. Daný soubor (.ui soubor) si můžeme vytvořit pomocí **Qt Designer**u a přeložit jej pomocí nástroje **pyuic4** do souboru Pythonem dobře čitelného. Bude-li náš plugin ob-



Obr. 1.3: *Plugins →Manage Plugins...* - správa pluginů

sahovat další soubory jako ikony, obrázky či zvuky, vytvoříme si soubor s příponou *.qrc*. Soubor je .xml dokumentem a obsahuje relativní cesty k našim souborům. *.qrc* soubor můžeme vytvořit ručně nebo pomocí **Qt Designer**u. Soubor poté opět přeložíme do souboru čitelného Pythonem pomocí **pyrcc4**.

Pakliže napíšeme plugin, o kterém si myslíme, že by mohl být užitečný, že by ho mohl používat také někdo další, můžeme se pokusit jej nahrát do repositáře. Více informací [http://plugins.qgis.org/plugins/] a [http://plugins.qgis.org/]

__init__.py

Inicializační soubor, který slouží k získání informací o zásuvném modulu a jeho načtení. Soubor by měl obsahovat funkce $name()$, $description()$, $version()$, $qgisMinimumVersion()$ a $authorName()$, které vrací textové řetězce udávající jméno, popis, verzi pluginu, minimální požadovanou verzi QGIS a jméno autora pluginu, a $classFactory(iface)$.

Funkce $classFactory(iface)$ vrací instanci třídy reprezentující náš plugin. *iface* odkaz na instanci třídy **QgisInterface**, umožňující pluginu přistupovat k funkcím QGIS. Tato funkce je volaná **QGIS Plugin Manager**em.

Od verze QGIS 2.0 pravděpodobně nebudou akceptována metadata z inicializačního souboru *__init__.py*, ale pouze ze souboru *metadata.txt*.

Ve verzi QGIS 1.9.90 mohou být pluginy zobrazovány nejen v menu *Plugins*, ale mohou být rozděleny pomocí kategorií *Raster*, *Vector*, *Database* a *Web*. V inicializačním

souboru tedy přibude funkce *category*(), která tuto informací vrací.

Inicializační soubor může poté vypadat takto [Ukázka kódu 1.1].

```
1  def name():
2    return "Nazev zasuvneho pluginu"
3
4  def description():
5    return "Popis pluginu."
6
7  def version():
8    return "Version 0.1"
9
10 def qgisMinimumVersion():
11   return "1.0"
12
13 def authorName():
14   return "Tonda"
15
16 def category():
17   return "Raster"
18
19 def classFactory(iface):
20   from plugin import Plugin
21   return Plugin(iface)
```

Ukázka kódu 1.1: __init__.py - inicializační soubor

plugin.py

V souboru *plugin.py* se nachází hlavní třída reprezentující náš plugin. Ta musí obsahovat metody *__init__*, initGUI a unload. Metoda *__init__* slouží mimo jiné k uchovávání odkazu na **QgisInterface**. Metoda *initGUI* inicializuje plugin. Inicializace, i když název k tomu svádí, nemusí být spojena s GUI. Metoda *unload*, jak název napovídá, se stará o akce po deaktivaci pluginu.

Soubor se zásuvným modulem korespondující s předchozím inicializačním souborem [Ukázka kódu 1.1] by mohl vypadat takto [Ukázka kódu 1.2].

```
1 class Plugin:
2     def __init__(self, iface):
3         self.iface = iface
4     def unload(self):
5         print "Plugin Plugin by deaktivovan."
6     def initGui(self):
7       print "Plugin Plugin byl nacten"
```

Ukázka kódu 1.2: plugin.py - plugin

1.2.4 C++ plugin

QGIS Processing Framework je plugin psaný v jazyce Python, proto se zde nebudu mnoho zmiňovat o pluginech psaných v jazyce C++. Více informací o tvorbě pluginů v C++ můžete najít v [7]QGIS Coding and Compilation Guide.

[6]http://download.osgeo.org/qgis/doc/manual/qgis-1.5.0_coding-compilation_guide_en.pdf

1.3 Qt, PyQt

V současné době se vývojem Qt zabývá firma Nokia, která Qt koupila v roce 2008 od norské společnosti Trolltech. Společnost Trolltech započala s vývojem Qt v roce 1999. Qt je poměrně mocný soubor nástrojů pro psaní grafických aplikací v jazyce C++. Není to ale pouze knihovna pro psaní GUI. Qt nabízí také řadu programů, které usnadňují vývojáři práci. Například velmi kvalitní IDE v podobě Qt Creator či Qt Designer pro pohodlnou tvorbu grafického rozhraní pouhým přetahováním widgetů myší. Qt Designer umožňuje pohodlně rozvrhnout a umístit jednotlivé widgety, seskupovat je do layoutů či nastavovat parametry.

Existuje také mimo jiné verze pro Python - v současnosti verze PyQt4. PyQt je vyvíjena firmou Riverbank Computing. Z rodiny Qt, resp. PyQt, byla v této práci využita samotná knihovna pro psaní kódu, obzvláště pak její Graphics View Framework, a program QtDesigner.

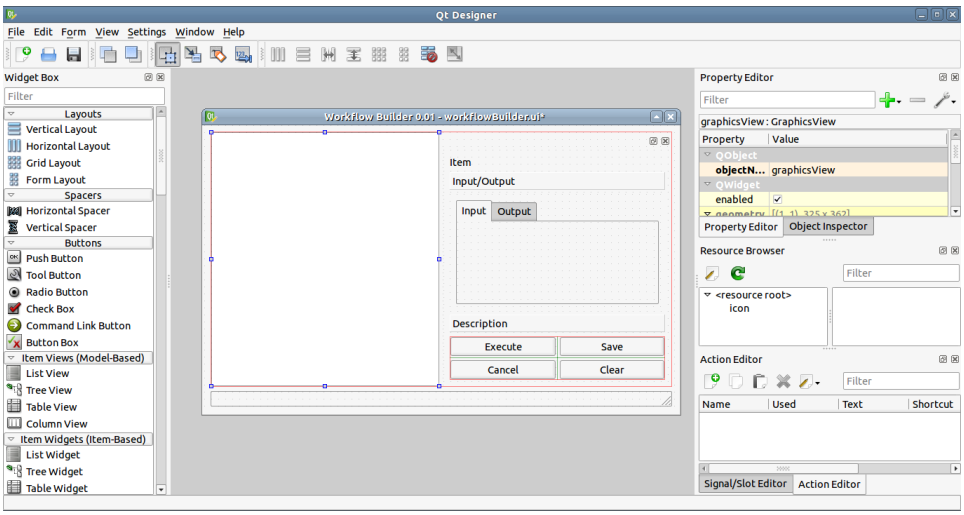
V této podkapitole se také zmíním o architektuře Model View Controller (MVC) a její implementaci v Qt, kterou jsem použil u Processing Manageru (toolbox pro QGIS Processing Framework), a Graphics View Framework, který jsem využil při práci na vizuální stránce práce. V závěru zmíním projekt VisTrails, který byl inspirací při návrhu grafického znázornění Workflow Builderu.

Pro bližší informace o Qt (i o PyQt4) a jejich možnostech doporučuji oficiální dokumentaci Qt, která je dostupná online a je na velmi vysoké úrovni [3].

Soubor vytvořený v Qt Designer (s příponou **.ui**) lze jednoduše přeložit programem **pyuic4** do kódu pro Python srozumitelného.

```
1  pyuic4 soubor_vytvoreny_v_Qt_Designer.ui -o py_soubor.py
```

Ukázka kódu 1.3: pyuic4 - přeložení .ui souboru do pythoního kódu



Obr. 1.4: Qt Designer - nástroj pro tvorbu grafického rozhraní

1.3.1 Signály a sloty

Každý objekt knihovny Qt, který je potomkem třídy QObject, má své signály a sloty. Signál je to, co objekt vysílá (emituje), má svůj název. Využívá se při různých změnách objektu. Například když klikneme na objekt **QPushButton**, vyšle se signál *clicked()*. Tento signál poté můžeme zachytit pomocí metody *connect()*, kterou dědí každý takový objekt knihovny Qt od třídy **QObject**. Takové propojení signálu a slotu může vypadat například takto $connect(kdoVyslal, SIGNAL(\text{"}clicked()\text{"}), SLOT())$. Slot je v podstatě metoda či funkce, která se zavolá na základě nějakého podnětu, signálu.

Signály a sloty v Qt se hojně využívají při tvorbě grafického rozhraní. Jakékoliv kliknutí, změna textu v **QLineEdit**, změna prvku v **QComboBox**, změna pozice grafického objektu (**QGraphicsObject**) či zavření okna vysílají signály. Tyto signály se vysílají bez ohledu, zdali jim naslouchámе či nikoliv. Každá třída má signály a sloty, které dědí po předku, a navíc může obsahovat další, které jsou pro ni typické a mohou se programátorovi hodit. Pakliže nás nějaká změna zajímá, můžeme daný signál zachytit. Kromě toho si můžeme sami napsat svoje vlastní signály či sloty a nemusí se jednat jen o grafické rozhraní. Musíme mít ale na paměti, že objekt, který vysílá signál, musí být potomek objektu **QObject**. Vyslat signál můžeme pomocí me-

tody *emit(SIGNAL(),...)*. V metodě $SIGNAL()$ uvedeme název signálu a dále za ním parametry, které se s ním vyšlou. Signály poté spojíme se slotem pomocí metody $QObject.connect(QObject, SIGNAL, SLOT)$. Tato metoda je statická, tudíž ji můžeme volat přímo z **QObject**. Slot je metoda, která se spouští na základě signálu.

Příklad:

Předpokládejme, že *tonda* je objekt třídy **Tonda**, která je potomkem třídy **QObject**. Když jde *tonda* domů, vyšle signál $SIGNAL("jdu")$ s parametrem "*domu*":

```
1    tonda = Tonda()
2    tonda.emit(SIGNAL("jdu"),"domu")
```

Ukázka kódu 1.4: vyslání slotu pod názvem "*jdu*" s atributem "*domu*"

Marie je také potomek třídy **QObject** a nachází se kdekoliv. Marie má v sobě zabudovaný slot a jakmile zachytí signál od *tondy*, že odešel, začne jednat:

```
1  class Marie(Object):
2      def __init__(self):
3          QObject.__init__(self)
4          self.connect(tonda, SIGNAL("jdu"), self.jednat)
5          # mohou nasledovat dalsi spojeni
6
7      def jednat(self, parametr):
8          # tady se Marie muze rozhodnout na zaklade parametru,
9          # jak bude jednat
10         # napriklad:
11         if parametr is "domu":
12             self.vecere()
13
14 marie = Marie()
```

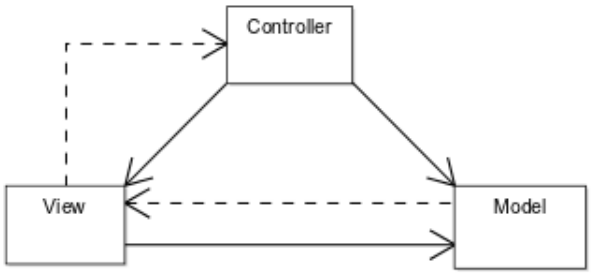
Ukázka kódu 1.5: zachycení signálu "*odesel*" od tondy

Tonda může emitovat kolik signálů chce a záleží pouze na tom, kolika signálům bude

marie naslouchat.

1.3.2 Model-View architektura

Při seznamování s projektem QGIS Processing Framework a po komunikaci s Camilo Polymeris (student, který začal psát QGIS Processing Framework), jsem začal s přepsáním Processing Manageru (panelu s moduly) z QTreeWidget do MVC architektury. Standardní MVC architektura dělí aplikaci do tří částí, které jsou na sobě co nejméně závislé. Jsou to Model, View a Controller. Oddělení dat od aplikační a prezentační logiky dělá kód přehlednější a lépe udržitelným. Velkou výhodou také je, že jeden model se může zobrazit v několika různých pohledech vždy jiným způsobem.

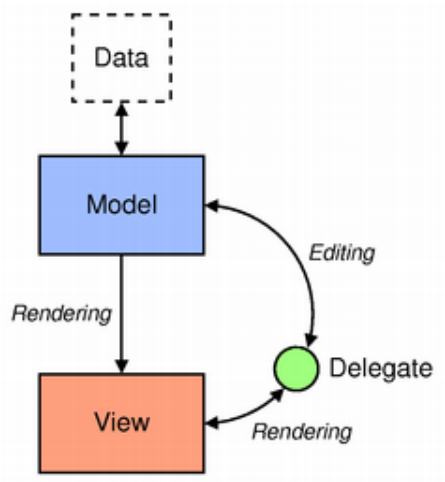


Obr. 1.5: Propojení jednotlivých částí architektury MVC

- **Model** se stará o data - není to pouze místo, kde jsou uložená data, ale jsou zde také definována pravidla, kterýma se jednotlivá data řídí
- **View** se stará o zobrazení dat v Modelu a o uživatelské rozhraní
- **Controller** spravuje reakce na uživatelovy podněty

V Qt se implementace MVC architektury objevila s verzí Qt4 v podobě model-view-delegate, kde funkci *controller*u částečně přebírá *view* (pohled) a částečně *delegate* (delegát). Delegát určuje, jak budou data editována, případně zobrazena, a komunikuje přímo s pohledem a s modelem. V některých případech může pohled zastávat funkci delegáta. Jedná se o případy, kdy se data editují pomocí jednoduchých editačních nástrojů jako je například editace pomocí **QLineEdit** u řetězců. Mluvíme tedy o

model/view architektuře. Jednoduchý příklad, kde je možné vidět použití hierarchicky uložených dat do modelu a zobrazených ve stromovém pohledu lze najít v **Příloze A - ukázka použití model/view architektury v PyQt4**. V příkladu uvedeném v téže příloze je také vidět použití vlastního delegáta a proxy modelu pro vyhledávání mezi daty.



Obr. 1.6: model-view architektura v Qt4

- **Model** - tady se nic nemění oproti standardní MVC architektuře; komunikuje se zdrojem dat a poskytuje API pro ostatní komponenty architektury (*view* a *delegate*)
- **View** - zobrazuje data a navíc nabízí základní nástroje pro jejich editaci
- **Delegate** - delegát můžeme definovat vlastní widgety sloužící k editaci dat z Modelu; může také definovat, jak se budou jaká data zobrazovat

Komunikace mezi jednotlivými komponentami probíhá pomocí signálů a slotů. Qt pro každý prvek architektury (*model*, *view* a *delegate*) poskytuje základní čistě abstraktní třídy plus několik dalších tříd již přímo použitelných implementací. Například pro data z tabulky můžeme přímo využít **QTableModel** a **QTableView**. Pakliže nám žádná z tříd nevyhovuje, můžeme samozřejmě reimplementovat třídy již existující.

Model

Každý model je založený na abstraktní třídě **QAbstractItemModel**. Pakliže budeme chtít zobrazovat data jako seznam či v tabulce, můžeme se poohlédnout po dalších abstraktních třídách **QAbstractListModel**, resp. **QAbstractTableModel** implementující další prvky, které jsou pro daný model typické. Už z názvu je zřejmé, že žádná z těchto tříd nemůže být použita přímo. Mohou nám posloužit k napsání svých vlastních modelů. Také se můžeme pokusit vybrat si z několika základních modelů, které jsou připraveny k přímému použití. Mezi tyto modely patří například **QStringListModel** pro seznamy řetězců a **QStandardModel** pro složitější data. Dále existují typy určené pro přístup do databází **QSqlQueryModel**, **QSqlTableModel** a **QSqlRelationalTableModel** či **QFileSystemModel**, který poskytuje informace o souborech a složkách na vašem lokálním souborovém systému. Máme tedy několik předpřipravených modelů. Nebudou-li nám plně vyhovovat, můžeme si kteroukoliv třídu vybrat a reimplementovat ji.

Dále existují tzv. proxy modely, které stojí mezi view a "standardním"modelem a poskytují podporu pro zpracování dat. Například **QSortFilterProxyModel**, který umožňuje uživateli vytvořit pravidla pro řazení a filtraci dat.

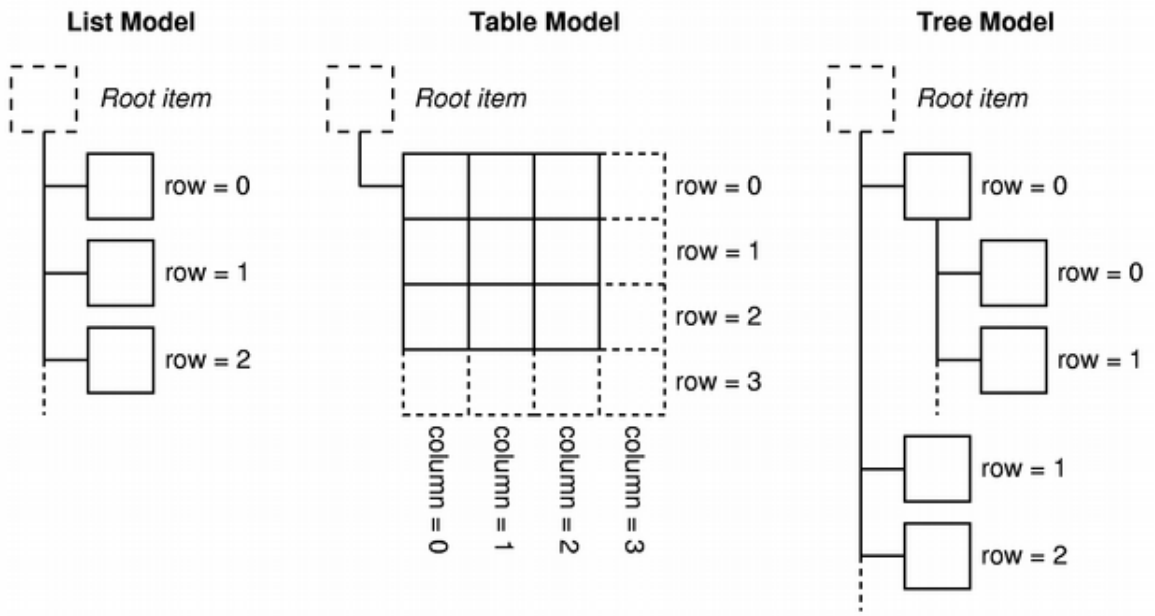
View a Delegate získávají a manipulují s daty uloženými v modelech pomocí indexů třídy **QModuleIndex** a rolí. Index nám udává pozici v modelu pomocí rodiče, řádku a sloupce. V indexu mohou být uložena různá data pomocí různých rolí.

metoda	popis
child(int row, int column)	vrací potomka na dané pozici QModelIndex
data(int role)	vrací data v podobě QVariant
model()	vrací model, ve kterém se nachází daný index
parent()	vrací rodiče v podobě QModelIndex
row(), column()	vrací číslo sloupce/řádku daného indexu vůči jeho rodiči

Tabulka 1.1: některé metody třídy QModelIndex

Role je celé číslo, na základě kterého můžeme zjistit "roli"dat. Existuje několik

základních rolí jako $DisplayRole(0)$, $ToolTipRole(3)$ či $UserRole(32)$. Samozřejmě si můžeme v podstatě zvolit jakékoliv celé číslo. Zde se s oblibou využívá $UserRole$, která reprezentuje nejvyšší číslo z předdefinovaných rolí (například $UserRole$ + celé číslo). Z QModelIndex dostaneme data pomocí metody $data(role)$.



Obr. 1.7: modely v model-view architektuře a jejich pozicování

Obecně se data do modelu ukládají pomocí metody $setData$(QModelIndex index, QVariant value, int role). Z objektu QVariant můžeme dostat naše data voláním metod jako $toInt()$, $toString()$, $toRect()$... případně $toPyObject()$ [Ukázka kódu 1.6].

metoda	popis
data(QModelIndex index, int role)	vrací data v podobě QVariant
setData(QModelIndex index, QVariant value, int role)	uloží data do modelu
insertRow(int row, QModelIndex parent)	vloží data na danou pozici
removeRow(int row, QModelIndex parent)	maže data na dané pozice
index(int row, int column, QModelIndex parent)	vrací index na dané pozici

Tabulka 1.2: některé metody třídy QAbstractItemModel

U modelu **QStandardItemModel** se může k položkám v modelu přistupovat také jako **QStandardItem**. Data se do modelu ukládají jako **QStandardItem()** objekt.

Pro uložení informací do objektu **QStandardItem** se používá metoda *setData*(QVariant *data*, int *role*). Pakliže nastavíme data pouze pomocí *setData*(*data*), role se nastaví na $UserRole + 1$. Data potom dostaneme z **QStandardItem** pomocí metody *data*(*role*). Model s QStandardItemModel s QStandardItem se hodí pro hierarchicky uspořádaná data. Ukázka použití QStandardItem viz [Ukázka kódu 1.6].

```
1  from PyQt4.QtCore import Qt
2  from PyQt4.QtGui import QStandardItem
3
4  student = QStandardItem("Tonda")
5  student.setData(24)
6  student.setData("Geoinformatika", Qt.UserRole + 2)
7
8  # vytiskne (24, True) - True znamena, ze se jedna o cislo
9  print student.data().toInt()
10 # vytiskne (24, True)
11 print student.data(Qt.UserRole + 1).toInt()
12 # vytiskne "Tonda"
13 print student.data(Qt.DisplayRole).toString()
14 # vytiskne "Geoinformatika"
15 print student.data(Qt.UserRole + 2).toString()
```

Ukázka kódu 1.6: QStandardItem - vytvoření a získání dat

Obecně tedy data ukládáme pomocí metody *setData*(data) a získáváme pomocí *data*(), kde na stejnou pozici můžeme uložit více dat s různými rolemi.

View

Pomocí pohledů Qt umožňuje zobrazovat data uložená v modelu. Jeden model můžeme zobrazovat v několika různých pohledech. Všechny pohledy jsou potomky abstraktní třídy **QAbstractItemView**, ta je potomek třídy QAbstractScrollArea a přes QFrame se dostaneme ke třídě **QWidget**. S pohledy tedy můžeme zacházet jako s ostatními widgety. Model se do view nastaví pomocí metody *setModel*(QAbstractItemModel model). Jednotlivé prvky z modelu jsou pohledu dostupné opět

pomocí indexů (QModelIndex). Pohled dokáže prvky zobrazit, řekněme, obyčejně. Pakliže si chceme se zobrazením prvků pohrát více (například měnit font či barvu podle nějakých vlastností dat), použijeme k tomu delegáta. Ten se nastaví pomocí metody *setItemDelegate*(QAbstractItemDelegate delagate). Ukázka nastavení modelu viz [Ukázka kódu 1.7].

```
1  model = QStandardItemModel()
2  delegate = QItemDelegate()
3
4  view = QTreeView()
5  view.setModel(model)
6  view.setItemDelegate(delegate)
```

Ukázka kódu 1.7: View - vytvoření pohledu a nastavení modelu a delegáta

Pro data, která budou zobrazována jako seznam, můžeme využít pohled **QListView**, pro tabulková data **QTableView**, pro stromová data pak **QTreeView**.

Delegate

Všichni delegáti musí být potomky abstraktní třídy **QAbstractItemDelegate**. Qt nám nabízí k přímému použití třídy QItemDelegate a QStyledItemDelegate. *View* má defaultně nastaveného delegáta QStyledItemDelegate. Pomocí delegáta můžeme určit, jak se budou dané položky z modelu zobrazovat a jak se budou editovat.

Pakliže máme v modelu například jako data uloženy studenty a chceme, aby byli vypisováni studenti modře a studentky červeně, můžeme si vytvořit vlastního delegáta, který nám to umožní. V tomto případě [Ukázka kódu1.8] využijeme QItemDelegate a přepíšeme metodu *paint*.

```
class Delegate(QItemDelegate):
    def __init__(self, parent=None, *args):
        QItemDelegate.__init__(self, parent, *args)

    def paint(self, painter, option, index):
        painter.save()

        # nastaveni fontu
        painter.setPen(QPen(Qt.black))
        painter.setFont(QFont("Times", 10, QFont.Bold))

      # nastaveni barvy podle pohlavi
        if index.data(Qt.UserRole + 3).toString() == "female":
            painter.setPen(QPen(Qt.red))
        elif index.data(Qt.UserRole + 3).toString() == "male":
            painter.setPen(QPen(Qt.blue))

        value = index.data(Qt.DisplayRole)
        if value.isValid():
            text = value.toString()
            painter.drawText(option.rect, Qt.AlignLeft, text)

        painter.restore()
```

Ukázka kódu 1.8: Delegate - přepsání metody *paint*

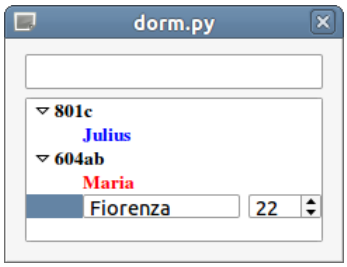
V tomto příkladě 1.8 předpokládáme, že data (v podobě indexů) obsahují informaci o pohlaví uloženou pod rolí $Qt.UserRole + 3$.

Editor (widget pro editaci dat) nastavíme reimplementací metody *createEditor*, která vrací QWidget. Aby se po editaci změnila data v modelu, musíme také reimplementovat metodu *setModelData*. V [Ukázka kódu1.9] předpokládáme, že třída *editStud* je widget, který je složen ze dvou dalších widgetů - QLineEdit pro editaci jména a QSpinBox pro nastavení věku.

```
class Delegate(QItemDelegate):
    def __init__(self, parent=None):
        QItemDelegate.__init__(self, parent)
    def createEditor(self, parent, option, index):
      # editStud je widget slozeny z QLineEdit a QSpinBox
        editor = editStud(index, parent)
        return editor
    def setModelData(self, editor, model, index):
        model.setData(index,QVariant(editor.name()))
        model.setData(index,QVariant(editor.age()), Qt.UserRole+4)
```

Ukázka kódu 1.9: Delegate - přepsání metod *createEditor* a *setModelData*

Použití QStandardItemModel s QTreeView za použití proxy modelu a delegáta z ukázek 1.8 a 1.9 můžeme dostat výsledek podobný tomuto:



Obr. 1.8: Ukázka použití QStandardItemModel, QTreeView, delegáta a proxy modelu.

V horní části vidíme widget QLineEdit, který je propojený s proxy modelem a slouží s vyhledávání v datech. Barevně odlišené pohlaví je způsobené delegátem, stejně jako řádek, který se edituje (QLineEdit + QSpinBox).

1.3.3 Drag and Drop

Zjednodušeně řečeno Drag and Drop je mechanismus, který nám umožňuje vzít jeden objekt z jednoho místa a přesunout ho na místo druhé. A to nejen v rámci jedné aplikace, ale také mezi různými aplikacemi. Na základě 'dopadu' (drop) objektu můžeme vyvolávat různé akce. Můžeme definovat, který objekt může být přetahován nebo které objekty mohou dopadnout na daný objekt (které objekty budou akcepto-

vány). Data se přenáší pomocí objektu **QDrag**, do kterého se uloží data v podobě **QMimeData**.

Pro umožnění chytnutí objektu (widgetu) myší přepíšeme metodu *mouseMoveEvent*, která je děděna z **QWidget**. Zde můžeme nastavit, které tlačítko myši budeme akceptovat a další pravidla na základě kterých se vytvoří či nevytvoří objekt třídy **QDrag**.

Akceptování dopadnutých objektů nastavíme metodou *acceptDrops* s parametrem True. Dále musíme přepsat metodu *dragEnterEvent*(event), *dragMoveEvent*(event) a *dropEvent*(event), kde akceptujeme event (událost) pomocí metody její *accept*(). Jednotlivé události jsou objekty tříd **QDragEnterEvent**, **QDragMoveEvent** a **QDropEvent**. Třída **QDropEvent** obsahuje metodu *source*(), která nám vrací zdrojový widget **QDrag** objektu. Pomocí toho se také můžeme rozhodnout, zda danou událost příjmeme či nikoliv.

Tohoto mechanismu jsem využil při přetahování modulů z Processing Manageru do Workflow Builderu. Dále toho také využívá Graphics View Framework při pohybu grafických prvků.

1.3.4 Graphics View Framework

Graphics View Framework nabízí prostředí pro práci s velkým počtem dvojrozměrných prvků. Nabízí také widget (**QGraphicsView**), ve kterém se dané prvky zobrazují. Podporuje funkce jako zoom, změna měřítka os nebo rotaci. Prostředí umožňuje spravovat klasické události jako je kliknutí myší, její pohyb či stisknutí klávesy.

Prostředí staví, podobně jako model-view architektura, na principu, kdy jsou samotná data oddělena od způsobu jejich zobrazení. V Graphics View Framework je model v podobě scény (**QGraphicsScene**) a pohled zastupuje třída **QGraphicsView**. Základní třídou dvojrozměrných prvků je **QGraphicsItem**. Z této třídy se dědí několik dalších jejich reimplementací jako QGraphicsRectItem, QGraphicsPathItem či QGraphicsSimpleTextItem.

QGraphicsItem

QGraphicsItem je základní třída, ze které vychází ostatní 2D objekty. Pomocí metody *setPos* se nastaví pozice vůči rodiči. Pakliže žádný rodič není, bere se pozice ve scéně (QGraphicsScene). V Graphics View Framework také funguje hierarchie. Prvkům můžeme nastavit rodiče buď při jejich vytváření, či pomocí metody *setParentItem*(QGraphicsItem parent). Prvkům můžeme nastavit různé vlastnosti jako například jak budou graficky vypadat či zdali mohou být přesouvány. Mezi standardní grafické prvky, které reprezentují klasické tvary, patří:

- QGraphicsRectItem
- QGraphicsPathItem
- QGraphicsLineItem
- QGraphicsPolygonItem
- ...

Prvkům můžeme dále nastavovat tzv. ToolTip pomocí metody *setToolTip*(QString toolTip) či tzv. flagy pomocí *setFlag*(GraphicsItemFlag flag, bool enabled = true) a *setFlags*(GraphicsItemFlags flags). Flagy slouží k nastavení chování prvku. Například chceme-li s prvkem pohybovat, nastavíme flagy QGraphicsItem.ItemIsMovable, QGraphicsItem.ItemIsSelectable a QGraphicsItem.ItemIsFocusable na hodnotu true [viz Ukázka kódu 1.10].

```
1  rect = QGraphicsRectItem()
2  rect.setFlags(QGraphicsItem.ItemIsMovable | \
3                QGraphicsItem.ItemIsSelectable | \
4                QGraphicsItem.ItemIsFocusable)
```

Ukázka kódu 1.10: Nastavení flagů u QGraphicsRectItem

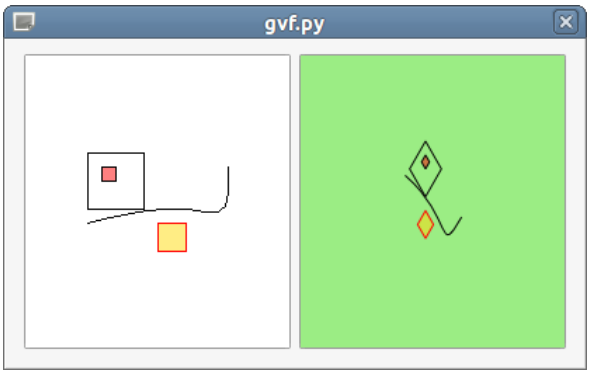
QGraphicsScene

Obecně se do scény prvky přidávají pomocí metody *addItem*(QGraphicsItem item). U klasických tvarů jako čtyřúhelník, elipsa či linie můžeme použít rovnou metody k tomu určené jako *addRect*(QRectF rect), *addEllipse*(QRectF rect) či *addLine*(QLine line). Prvky se mažou ze scény pomocí *removeItem*(QGraphicsItem item). Pro smazání všech prvků ve scéně slouží metoda *clear*(). Další užitečné metody jsou *items*(), která vrací všechny prvky scény, *itemAt*(QPointF point) vracící prvek na vybrané pozici ve scéně, či *selectedItems*(), která nám vrací seznam prvků, které jsou vybrány.

QGraphicsView

U QGraphicsView nastavíme scénu pomocí *setScene*(). Dále můžeme nastavit možnost přibližování a oddalování, měřítko, barvu pozadí, vyhlazování hran u prvků, můžeme rotovat scénu atp.

Na obrázku Obr. 1.9 je vidět jedna scéna zobrazena ve dvou rozdílných pohledech. Změna scény provedená v jednom pohledu se projeví také v druhém pohledu. U pohledu vpravo byla nastavena barva pozadí pomocí metody *setBackgroundBrush*(QBrush brush), změněno měřítko os pomocí *scale*(int x, int y) a scéna byla otočena o 45° pomocí *rotate*(int angle).



Obr. 1.9: Zobrazení jedné scény ve dvou různých pohledech.

Při tvorbě vlastního pohledu můžeme reimplementovat metody pro správu Drag and Drop prostředí (*mousePressEvent*, *dragEnterEvent*, *dragMoveEvent*, *dropEvent*...),

spravovat vstup z klávesnice (*keyPressEvent*, *keyReleaseEvent*), události vyvolané myší (*mousePressEvent*, *mouseDoubleClickEvent*, *mouseMoveEvent*...) či metodu *wheelEvent* pro definování chování widgetu při použití prostředního kolečka myši (QWheelEvent je defaultně ignorován).

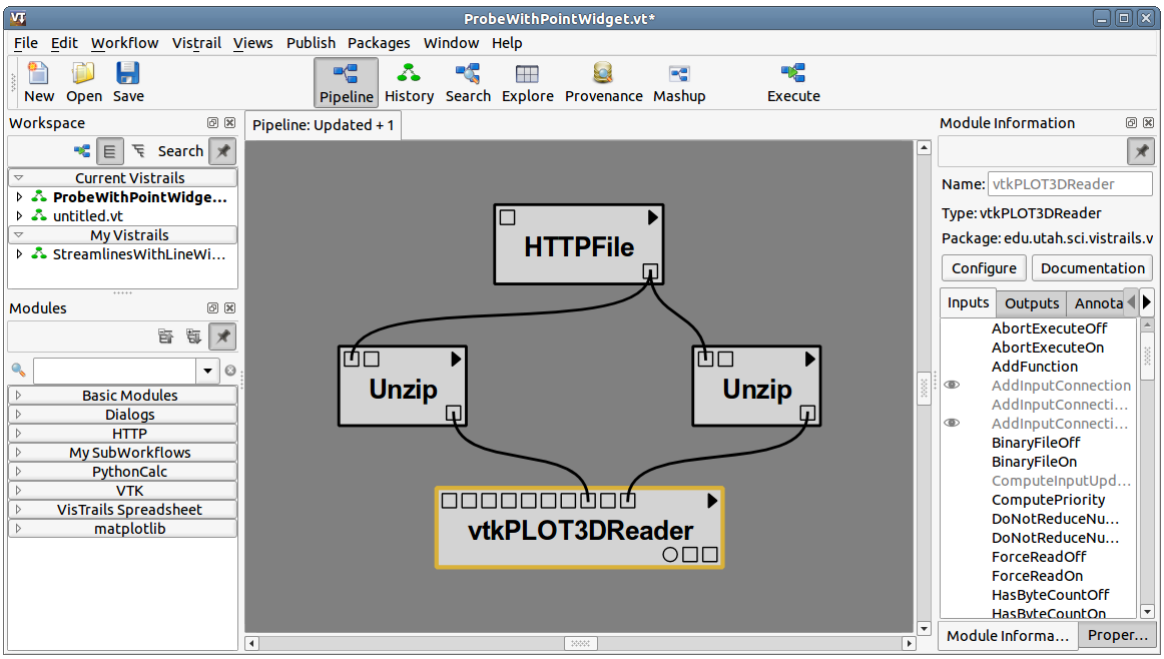
1.3.5 VisTrails

VisTrails je systém pro správu workflow diagramů vyvíjený na University of Utah. Je napsán v Pythonu za pomoci knihovny PyQt. Systém je open source a uvolněný pod licencí GPL v2.

Na začátku práce na Workflow Builderu se nabízela možnost využít některých svobodných projektů pro modelování workflow diagramů. Vzhledem k tomu, že QGIS využívá knihovnu Qt a QGIS Processing framework samotný je psaný v Pythonu, naskýtali se jako možnosti inspirace projekty Orange či [8] VisTrails.

Nejvíce mě oslovilo grafické zpracování VisTrails [viz Obr. 1.10]. Uživatel na první pohled vidí, jaký parametr je s kterým spojen. Ne pouze který modul je s kterým spojen jak to často bývá u podobných programů. Studoval jsem kód a využil jsem prvky scény **QGraphicsModule** reprezentující modul, **QGraphicsPort** reprezentující vstupní a výstupní parametry modulu a **QGraphicsConnection**, reimplementace třídy QGraphicsPathItem a reprezentující spojení mezi parametry. Dále jsem se inspiroval postranním panelem, který zobrazuje informace o právě vybraném modulu a umožňuje také nastavování parametrů.

[8] http://www.vistrails.org/index.php/Main_Page



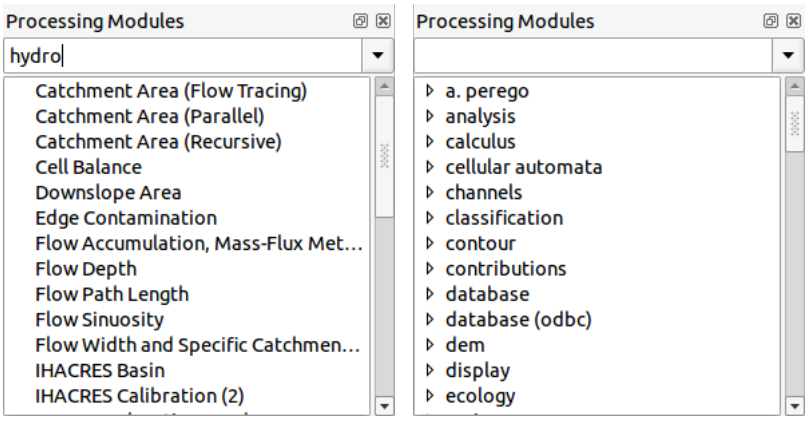
Obr. 1.10: Ukázka spojení prvků v systému VisTrails.

1.4 QGIS Processing Framework

QGIS Processing Framework vznikl v rámci projektu [9]GSoC 2011 . Student Camilo Polymeris z univerzity Universidad de Concepción si kladl za cíl napsat obecný framework, do kterého budou zapadat všechny moduly všech pluginů QGISu a každý modul bude možné použít buď samostatně nebo spojovat s jinými.

V době psaní této práce byla na světě první verze Processing Frameworku a vše nasvědčovalo tomu, že práce na frameworku budou pokračovat a nástroje v Processing Frameworku budou přibývat. Existovala totiž pouze částečná podpora pro funkce z SAGA GIS a plugin zpřístupňující funkce Orfeo Toolboxu (OTB). Orfeo Toolbox je svobodný software poskytující nástroje pro zpracování snímku z dálkového průzkumu Země.

V době mého připojení k QGIS Processing Frameworku byl projekt na začátku. Pro seznámení s projektem jsem přepsal Processing Manager (toolbox) z QTreeWidget do MVC architektury.



Obr. 1.11: QGIS Processing Framework - Processing Manager

Processing Manager je část QGIS Processing Frameworku, která zpřístupňuje všechny

[9]Google Summer of Code. Projekt společnosti Google na podporu studentu. více na http://code.google.com/soc/

moduly dostupné skrze QGIS Processing Framework z jednoho místa. Jedná se o panel se seznamem modulů, které jsou rozděleny podle tagů do různých skupin (například 'raster', 'hydrology'). Každý modul obsahuje seznam tagů, které napovídají, k čemu daný modul slouží. Uživatel může najít hledaný modul prohledáváním samotného stromu, či využít vyhledávací okénko v horní části panelu. Processing Manager prohledává tagy daného modulu a jeho název. Modul obsahuje dále popis, ale protože se tagy generují z tohoto popisu, není nutné popis procházet Obr.1.11.

Modul je reprezentován třídou **Module** a jeho instance třídou **ModuleInstace**. Z třídy **Module** získáváme informace o modulu. Pomocí metody *name*() získáme jméno modulu, metoda *description*() vrací popis, metoda *tags*() a metoda *instance*() vrací instanci třídy **ModuleInstace** daného modulu. Zavoláním metody *parameters*() získáme seznam parametrů daného modulu. Parametry jsou třídy **Parameter**.

U **ModuleInstace** můžeme pomocí metody *setValue*(Parameter, hodnota) přiřazovat parametrům konkrétní hodnoty. Metodou *value*(Paramter) získáme hodnotu parametru. Pomocí metody *setState*() s parametrem "2"spouštíme daný modul. Metoda *module*() vrací module (Module).

Parametry jsou třídy **Parameter**. Uchovávají v sobě informaci o názvu a popisu parametru, zdali je parametr povinný, jakého je typu a role (např. vstupní, výstupní) a jeho defaultní hodnotu. Přehled metod třídy **Parameter** je zobrazen v [Tabulka1.3].

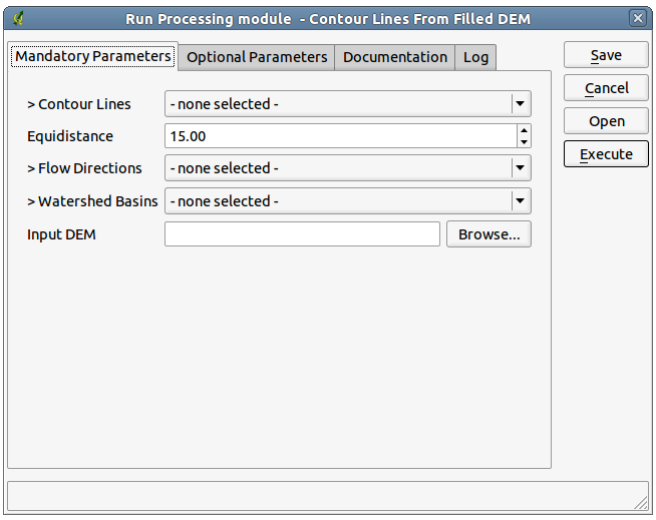
Modulů z **QGIS Processing Framework** jsou přístupné buď přes **Processing Manager** nebo přes pythoní konzoli [Ukázka kódu 1.11]. Konzoli lze spustit například klávesovou zkratkou **Ctrl+Alt+ p**.

metoda	popis
name()	vrací jméno
description()	vrací popis
type()	vrací typ [viz Tabulka 1.5
setRole(role)	nastaví roli
role()	vrací roli
setMandatory(bool)	nastaví zdali je parametr povinný
isMandatory()	vrátí hodnotu, zdali je parametr povinný
setDefaultValue()	nastaví defaultní hodnotu parametru
defaultValue()	vrátí defaultní hodnotu parametru

Tabulka 1.3: Metody třídy Parameter.

```
1  import processing
2
3  # seznam vsech registrovanych modulu
4  processing.framework.modules()
5
6  # vrati dany modulu
7  mod = processing.framework['nazev_modulu']
8  # vraci seznam parametru daneho modulu
9  mod.parameters()
10
11 # vytvori instanci modulu
12 instance = mod.instance()
13 # nastavi paramter
14 instance.setValue(parametr, hodnota)
15 # spusti modul
16 instance.setState(2)
17
18 # ziskani hodnoty parametru
19 instance.value(parametr2)
```

Ukázka kódu 1.11: Přístup k modulům přes konzoli.



Obr. 1.12: QGIS Processing Framework - okno pro nastavení a spuštění modulu

Spouští-li se modul přes **Processing Manager**, objeví se dialogové okno pro nastavení parametrů modulu a následné spuštění [viz Obr.1.12].

1.4.1 SAGA Plugin

SAGA Plugin vznikl v rámci stejného projektu Camila Polymeris pro GSoC 2011. Měl zpřístupňovat funkce programu SAGA GIS pomocí jeho API uživatelům Quantum GIS. Na stránkách projektu [2] se deklaruje, že by mělo být podporováno 170 modulů z celkových 425. Toto číslo vychází z předpokladu, že moduly, u kterých jsou všechny vstupní i výstupní parametry podporovány, pracují správně. Podporované parametry SAGA GIS a jejich reprezentace v Processing Frameworku 1.4. Parametry SAGA GIS, které nejsou podporované Processing Frameworkem: *Table field, Data Object, Grid list, Table, Node, Shape list, Parameters, Point Cloud, TIN, Static table, Table list, Color, TIN list a Colors*. Dále nejsou podporované interaktivní moduly. Bohužel ale nebyl plugin plně dokončen a skutečný počet správně pracujících pluginů není roven 170.

SAGA parametr	**PF Parameter**
Int Double Degree	NumericParameter
Range	RangeParameter
Bool	BooleanParameter
String Text	StringParameter
Chioce	ChoiceParameter
FilePath	PathParameter
Shapes	VectorLayerParameter
Grid	RasterLayerParameter

Tabulka 1.4: parametry SAGA GIS podporované Processing Frameworkem

1.4.2 Psaní pluginu pro PF

Plugin pro QGIS Processing Framework se v mnohém neliší od normálních pluginů psaných pro QGIS. Inicializační soubor se prakticky vůbec neliší. Třída reprezentující samotný plugin obsahuje navíc metodu *modules*(), která vrací seznam modulů třídy processing.**Module** (dále jen Module), který poskytuje daný plugin. Plugin samotný se musí registrovat v frameworku. To se provede příkazem processing.framework.registerModuleProvider(self), který se vloží do metody *initGUI*().

Plugin tedy může obsahovat několik modulů, které vrací pomocí metody *modules*(). Každý modul se skládá ze sebe a ze svojí "instance". Tedy z podtříd tříd Module a processing.ModuleInstance (dále jen ModuleInstance). Module je pro daný modul základní třída, která definuje typy parametry modulu, jeho název, popis a tagy. A vrací jeho instanci v podobě ModuleInstance, která slouží ke spouštění modulu s nastavenými parametry a spravuje, co se děje po provedení modulu. To znamená, že pakliže chceme spustit modul, musíme nastavit parametry a ty se nastavují v instanci, ne v modulu samotném. Modul poté spustíme metodou instance setStatus(2). Instance by mela obsahovat kód, který vstupní parametry zpracuje a poté také nastaví výstupní hodnoty.

Modul může mít několik vstupních a výstupních parametrů. V současné době dovoluje framework uživateli použít parametry 1.5.

Jako příklad uvedeme plugin Plugin, který bude mít na vstupu dva parametry. Jeden parametr pro načtení cesty k rastrovému souboru a druhý pro zadání jeho názvu, pod kterým se objeví v QGISu. Výstup bude jeden - rastr třídy QgsRasterMapLayer. Příklad je pouze ilustrativní.

Inicializační soubor nebudu uvádět, protože se nijak neliší od toho, když píšeme normální plugin pro QGIS. Soubor se samotným pluginem bude obsahovat třídu Plugin, která reprezentuje náš plugin [Ukázka kódu 1.12]. Dále třídu RasterToQgis(processing.Module) [Ukázka kódu 1.13] a třídu RasterToQgisInstance(processing.ModuleInstance) [Ukázka kódu 1.14] .

parametr	popis	grafická reprezentace
NumericParameter	číslo	QSpinBox
RangeParameter	dvojice číselných hodnot	pár QSpinBox
BooleanParameter	boolean	QCheckBox
ChoiceParameter	seznam možností např. vrstev, metod	QComboBox
StringParameter	textový řetězec	QLineEdit
PathParameter	cesta k souboru	QLineEdit + QPushButton
VectorLayerParameter	QgsVectorLayer	QComboBox s registrovanými vektorovými vrstvami
RasterLayerParameter	QgsRasterLayer	QComboBox s registrovanými rastrovými vrstvami

Tabulka 1.5: parametry podporované Processing Frameworkem

```
1  class Plugin:
2      def __init__(self, iface):
3          self.iface = iface
4      def unload(self):
5          pass
6      def modules(self):
7          return [self.rasterInputLayer]
8      def initGui(self):
9          self.rasterInputLayer = RasterToQgis(self.iface)
10         processing.framework.registerModuleProvider(self)
```

Ukázka kódu 1.12: Třída Plugin pro QGIS Processing Framework

```
class RasterToQgis(processing.Module):
    def __init__(self, iface = None):
        self.iface = iface
        self.inParamPath = PathParameter("Path to input raster",
         role=Parameter.Role.input)
        self.inParamName = StringParameter("Name of layerr",
         role=Parameter.Role.input)
        self.outParam = RasterLayerParameter("Output raster",
         role = Parameter.Role.output)
        self.outParam.setMandatory(False)
        processing.Module.__init__(self, "Input raster by path",
            description = "Description",
            parameters = [self.inParamPath,self.inParamName,self.outParam],
            tags = ["raster", "input"])

    def instance(self):
        return RasterToQgisInstance(self, self.inParamPath,
                                    self.inParamName,  self.outParam)
```

Ukázka kódu 1.13: Třída RasterToQgis reprezentující modul pro QGIS Processing Framework

V příkladu [Ukázka kódu 1.13] jsme u parametrů jsem nastavili pouze nejnutnější atributy jako název a roli. Role nám říká, zdali je parametr povinný či volitelný. U parametrů můžeme nastavit také popis či počáteční hodnotu při jejich tvorbě pomocí parametrů v konstruktoru *description* a *defaultValue*. Nastavit defaultní hodnotu můžeme také metodou *setDefaultValue*(value).

```
class RasterToQgisInstance(processing.ModuleInstance):
    def __init__(self, module, inParamPath, inParamName,  outParam):
        self.inParamPath = inParamPath
        self.inParamName = inParamName
        self.outParam = outParam
        processing.ModuleInstance.__init__(self, module)
        QObject.connect(self,
```

```
8               self.valueChangedSignal(self.stateParameter),
9               self.onStateParameterChanged)
10      def onStateParameterChanged(self, state):
11          if state == StateParameter.State.running:
12              path = self[self.inParamPath]
13              name = self[self.inParamName]
14              raster = QgsRasterLayer(path, name)
15              self.setValue(self.outParam, raster)
16              self.setState(StateParameter.State.stopped)
```

Ukázka kódu 1.14: Třída RasterToQgisInstance reprezentující instanci modulu pro QGIS Processing Framework

Instance kontroluje stav (state) modulu. Je-li nastaven hodnotu 2 (StateParameter.State.running) vezme si vstupní parametry a na jejich základě vytvoří novou rastrovou vrstvu **QgsRasterLayer**. A tu nastaví do odpovídajícího výstupního parametru.

1.4.3 Závěr

Bylo by dobré vyřešit vstupní vrstvy aby například rastrová vrstva zadaná jako PathParameter byla kompatibilní s parametrem RasterLayerParameter. Dát vývojáři pluginu možnost, aby mohl uživatel zadat vrstvu buď pomocí cesty nebo výběrem z již načtených vrstev. Dát tedy uživateli obě možnosti.

Do této chvíle je napsán OTB Plugin pro zpracování družicových snímků a rozepsán SAGA Plugin s podporou několika pluginů z SAGA GIS.

Camilo Polymeris měl v plánu pokračovat na projektu v rámci GSoC 2012, ale po objevení frameworku SEXTANTE svoji žádost stáhl a zapojil se do prací na SEXTANTE. QGIS Processing Framework se tedy zdá být mrtvým projektem.

Dále bych chtěl upozornit, že během vývoje se změnila struktura dat. Na začátku bylo jádro frameworku uloženo v python/processing a Processing Manager, GUI a dialog pro spouštění modulů v python/plugins/processingplugin. V současné době je vše uloženo v python/processingmanager, resp. jádro v python/processingmanager/processing. To může na začátku psaní vlastního pluginu způsobit menší problém v chybně zadané cestě k frameworku.

1.5 xml.dom.monidom

Nově vzniklé workflow se ukládají do souboru využívajíce jazyk XML. Výhoda XML dokumentu je, že se s ním snadno pracuje a jde v podstatě pouze o textový dokument.

XML nabízí jednoduché uložení hierarchicky strukturovaných dat. O prvcích XML dokumentu hovoříme jako o elementech. Elementy jsou ohraničeny počátečními a koncovými znaky, tzv. tagy. XML dokument obsahuje vždy právě jeden kořenový element, který se může skládat z dalších a dalších elementů. V příkladu XML dokumentu (Ukázka kódu 1.15]) je kořenový element *Graph*. Elementy mohou obsahovat atributy (dvojice jméno="hodnota"). Jména atributů se v rámci jednoho elementu nesmí opakovat. Elementy také mohou obsahovat text, který se uvádí mezi počátečním a koncovým znakem.

Příklad XML dokumentu:

```
<Graph name="Addition two rasters">
    Description...
    <SubGraph id="17">
          <Module id="769" name="Input raster by path">
           You can register raster layer to QGIS by giving the path.
           <tag> workflow builder </tag>
          </Module>
        <Module id="998" name="Operations with two rasters">
            Pixel by pixel operations with two rasters.
          </Module>
     </SubGraph>
</Graph>
```

Ukázka kódu 1.15: Příklad XML dokumentu

Pro práci s XML dokumenty se v prostředí jazyka Python nabízí několik modulů. Při psaní této práce byl vybrán **xml.dom**, resp. jeho odlehčená verze **xml.dom.minidom**, který k XML dokumentu přistupuje přes rozhraní DOM (Document Object Model). **DOM** je rozhraní pro přístup a práci s XML dokumenty. Je to zároveň standard

organizace World Wide Web Consortium (W3C). W3C je organizace, která se zabývá standardy v prostředí Word Wide Web.

Na začátku je potřeba si vytvořit objekt (*Document*), který bude reprezentovat celý XML dokument. *Document* je, stejně jako všechny ostatní elementy (objekty třídy Element) XML dokumentu, podtřídou třídy *Node*. Do takto vytvořeného objektu se poté mohou přidávat další elementy. Třída **Document** obsahuje statickou metodu *createElement*(*tagName*). Metodu můžeme tedy volat z třídy **Document** nebo z již existujícího objektu. To samé platí i v případě metody createTextNode(), která slouží k vytvoření textu, který se poté může vložit do elementu.

Pro přidání elementu do jiného elementu použijeme metodu *appendChild*(*newChild*) nebo *insertBefore*(*newChild*, *refChild*). Chceme-li nahradit jeden element druhým, použijeme metodu *replaceChild*(*newChild*, *oldChild*). Pro mazání elementu se používá metoda *removeChild*(*oldChild*). K získání informace, zdali element obsahuje atributy, zavoláme metody *hasAttributes*(). Všechny tyto metody jsou metody třídy **Node**. Třídy jako **Document** či **Element**, stejně jako všechny ostatní prvky XML dokumentu, jsou podtřídami třídy **Node**, tudíž i ony dědí tyto metody.

Atributy nastavíme u objektů třídy **Element** pomocí metody *setAttribute*(*name*, *value*). Metodou *hasAttribute*(*name*) se dotazujeme, zdali daný element obsahuje atribut *name*. Pomocí metody *getAttribute*(*name*) získáme hodnotu atributu *name*. A pomocí metody *removeAttribute*(name) smažeme atribut *name*.

Metoda *getElementsByTagName*(*tagName*) vrací seznam elementů korespondující s *tagName* a potomky daného elementu nacházející se v daném elementu.

XML dokument uložíme do souboru zavoláním metody writexml(*file*, *indent*, *addindent*, *encoding*) dané instance třídy Document. *File* je soubor připravený pro zápis, *indent* udává odsazení na začátku nového elementu (například začátek nového řádku), *addindent* udává přírůstkové odsazení pro potomky daného elementu (například tabulátor) a *encoding* je kódování. Pouze *file* je povinný parametr.

Příklad z [Ukázka kódu 1.15] bychom tedy mohli vytvořit kódem [Ukázka kódu 1.16] a zapsat do souboru [Ukázka kódu 1.17]. upravit

```
from xml.dom.minidom import Document

doc = Document()

graph = Document().createElement("Graph")
graph.setAttribute("name","Addition two rasters")
doc.addChild(graph)

graphDesc = Document().createTextNode("Description...")
graph.addChild(graphDesc)

subGraph = doc.createElement("SubGraph")
subGraph.setAttribute("id","17")
graph.addChild(subGraph)

module01 = doc.createElement("Module")
module01.setAttribute("id","769")
module01.setAttribute("name","Input raster by path")
module01Desc = doc.createTextNode("You can register ... the path.")
module01.addChild(module01Desc)
tag = doc.createElement("Tag")
tagDesc = doc.createTextNode("workflow builder")
module01.addChild(tag)
subGraph.addChild(module01)

module02 = doc.createElement("Module")
module02.setAttribute("id","998")
module02.setAttribute("name","Operations with two rasters")
module02Desc = doc.createTextNode("Pixel by pixel ... rasters.")
module02.addChild(module02Desc)
subGraph.addChild(module02)
```

Ukázka kódu 1.16:]ukázka tvorby XML dokumentu z [Ukázka kódu 1.15]

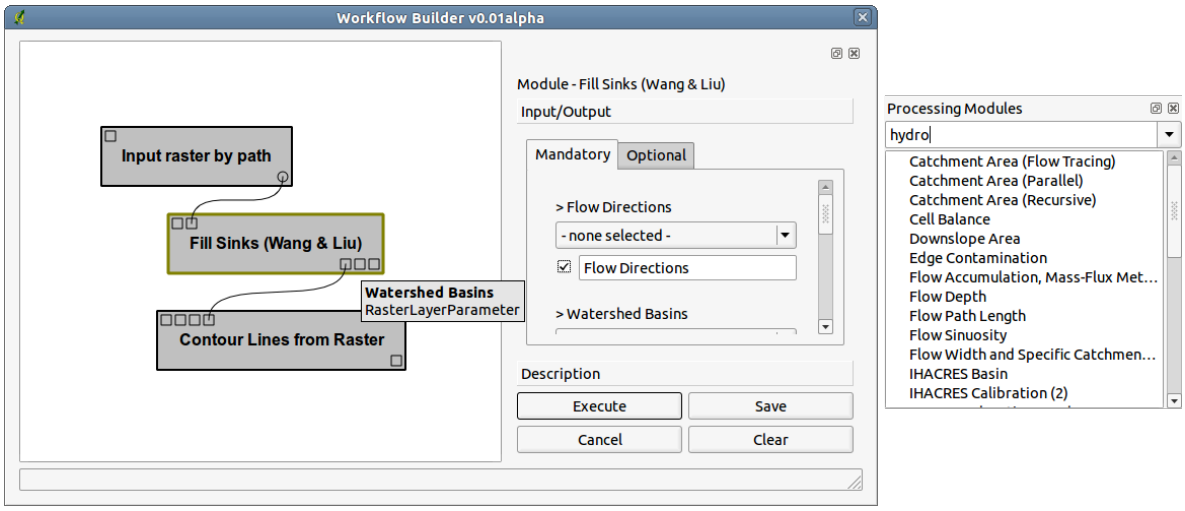
```
1 from xml.dom import Document
2
3 file = open("xml.xml", "w")
4
5 doc = Document()
6 doc.writexml(file, indent="\n",  addindent="\t", encoding="UTF-8")
7
8 file.close()
```

Ukázka kódu 1.17: uložení XML dokumentu do souboru

Pro otevření souboru použijeme metodu *xml.minidom.parse*(*path*), kde *path* je cesta k xml souboru.

Kapitola 2

Workflow Builder



Obr. 2.1: Workflow Builder

Workflow Builder umožňuje uživateli propojovat moduly dostupné z QGIS Processing Framework skrze Processing Manager. Výsledný graf (proces, workflow) lze poté jednoduše uložit jako nový modul QGIS Processing Frameworku. Dialogové okno Workflow Builder se skládá ze scény v levé části, která slouží k manipulaci s moduly, propojování jejich vstupních a výstupních parametrů pomocí myši. V panelu v pravé části lze nastavovat hodnoty jednotlivým parametrům, které nejsou propojeny. V pravé spodní části se nachází tlačítka pro spuštění procesu (*Execute*), k otevření dialogového okna pro uložení procesu, pro smazání celé scény a pro schování dialogového okna Workflow Builderu. Jakmile uživatel klikne na tlačítko pro uložení (*Save*), otevře se

nové dialogové okno, kde bude vyzván k nastavení nového modulu. Uživatel zadá jméno modulu, tagy, které napovídají o jeho využití, popis a hlavně parametry. U parametrů může uživatel nastavit, zdali chce, aby se parametr musel zadávat pokaždé i v novém modulu, či hodnota bude pokaždé stejná a tudíž se nemusí ani zobrazovat. Dále může uživatel nastavit alternativní název parametru. Pravá spodní část dialogového okna dále obsahuje dvě tlačítka - *Cancel* a *Clear*. *Cancel* slouží k schování okna a *Clear* k smazání workflow, tedy všech objektů scény.

Pro práci s Workflow Builder je dobré mít také alespoň jeden plugin, který registruje své moduly v QGIS Processing Frameworku, dále plugin **Workflow for Processing Framework Manager**, který byl napsán pro načítání modulů vytvořených pomocí Workflow Builderu a uložených do souboru ve formátu XML. Dále můžeme použít plugin **Input parameters for WB**, který přidává do QGIS Processing Frameworku moduly sloužící pro načítání vektorových a rastrových dat ze souboru, které jsou následně dostupny přes výstupní parametr daného modulu. Uživatel tedy není vázán jen na vrstvy, které jsou načteny v QGIS.

Pro začátek je vhodné shlédnout instruktážní video:

`http://youtu.be/4PxvWvTIyaU`

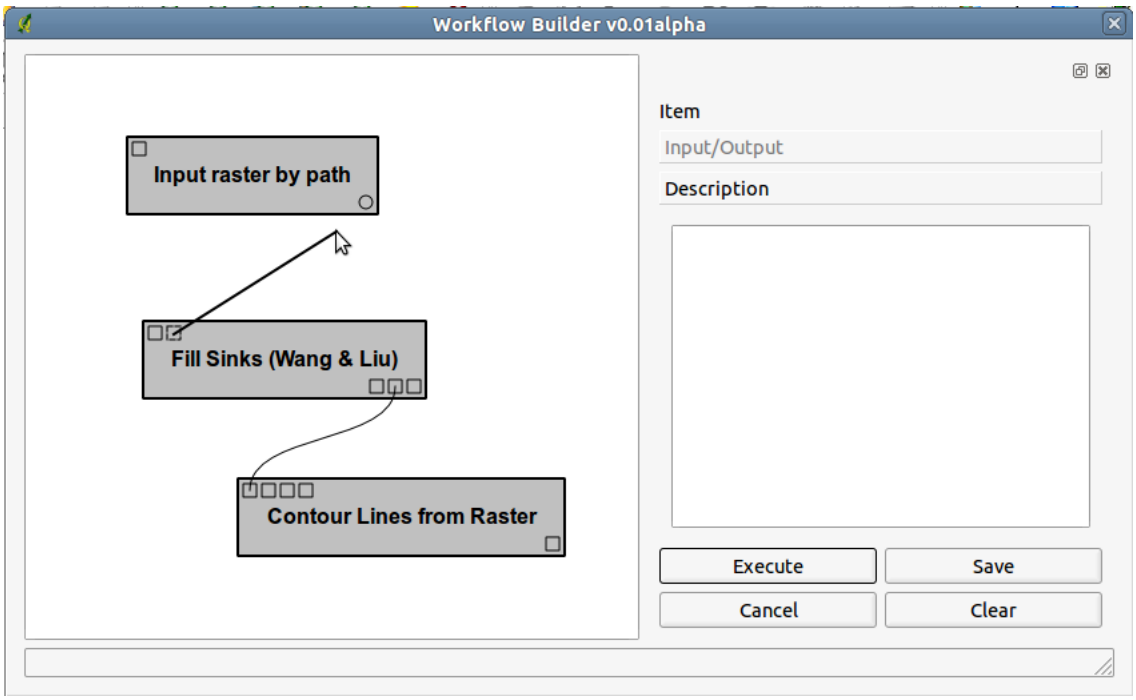
2.1 Tvorba workflow

K celkovému workflow se přistupuje jako k orientovanému grafu. V tomto smyslu je graf objekt třídy **Graph** a vytvoří se při spuštění Workflow Builderu. Vrcholy představují moduly, které jsou třídy Module, a hrany představují spojení, která jsou třídy **Connection**. Do grafu se postupně přidávají moduly podle toho, jak uživatel pomocí myši přetahuje moduly z Processing Manageru. Objekt třídy Module se vytvoří na základě instance přetaženého modulu (název, popis, tagy a parametry). Modul z Workflow Builderu obsahuje parametry třídy **Port**. Instance třídy Port se také vytváří automaticky a přiřazují se danému modulu. Grafická reprezentace Module je **QGraphicsModuleItem**, který také podle **Port**ů v Module vytvoří **QGraphicsPortItem**y. Při spojování portů mezi sebou se kontroluje, zdali koresponduje typ parametru (RasterLayerParameter, NumericPrameter, ...), spojuje-li se vstupní parametr s výstupním, zdali nejsou oba parametry parametry stejného modulu a zdali není vstupní parametr prázdný (to znamená, že není spojený s jiným parametrem). Typ a název parametru můžeme zjistit posunutím myši nad parametr (čtvereček - povinný parametr, kolečko - volitelný parametr). Pakliže jsou splněny všechny podmínky, vytvoří se spojení třídy Connection a jeho grafická reprezentace **QGraphicsConnectionItem**. Connection se poté přidá do Graphu. QGraphicsConnectionItem se přidá do scény třídy **DiagramScene**, která je reimplementací třídy QGraphicsScene z knihovny Qt. Pro tvorbu spojení stačí kliknout na požadovaný vstup/výstup a táhnout myší na druhý parametr Obr.2.2.

Zrušit modul či spojení můžeme tím, že si jej myší označíme a stiskneme klávesu *Delete*. K mazání prvků se může také použít tlačítko *Clear* v pravé spodní části dialogového okna, které smaže všechny prvky ve scéně včetně modulů a spojení.

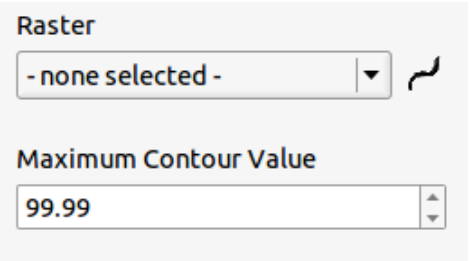
V Graphu máme tedy uloženy moduly a spojení mezi nimi (Module a Connetion). Jsou uloženy jako slovníky, kde klíč je identifikační číslo modulu, resp. spojení a hodnota je instance třídy Module, resp. Connection. Ty se během tvorby workflow mění podle toho, jak uživatel přidává a odebírá moduly, spojuje je a ruší spojení.

Po kliknutí na konkrétní modul se zobrazí jeho parametry v pravém postranním panelu. Ty jsou děleny na povinné a volitelné. Toto dělení, podobně jako označení vý-



Obr. 2.2: Workflow Builder - tvorba spojení

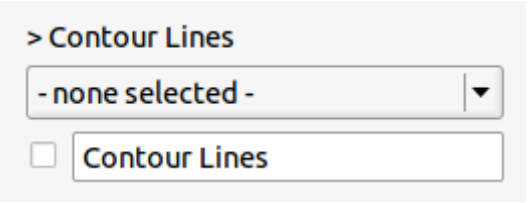
stupního parametru symbolem ”⟩ ”před jeho název, bylo převzato z klasického dialogu pro spuštění modulu v QGIS Processing Frameworku.



Obr. 2.3: Workflow Builder - vstupní parametr

Na Obr.2.3 je vidět, že widget pro nastavení vstupního parametru se skládá z jeho názvu parametru (QLabel), dále z widgetu, který se generuje na základě jeho typu (podobně jako u QGIS Processing Frameworku) a pakliže je parametr spojen s jiným, objeví se vpravo ikona signalizující spojení.

Widget pro nastavení výstupního parametru obsahuje řádek navíc se zaškrtávacím



Obr. 2.4: Workflow Builder - výstupní parametr

polem (QCheckBox) a textovým polem pro zadání názvu výstupní vrstvy (QLineEdit). Řádek slouží k načtení výstupní vrstvy do QGIS pod uživatelem zadaným názvem, pakliže zaškrtne zaškrtávací pole. Toto mělo být pouze provizorní řešení. Workflow Builder byl testován s SAGA Pluginem a ten je v současné době napsán tak, že nerespektuje zadaný výstupní parametr a jedná-li se o vrstvu (rastrovou nebo vektorovou), vytvoří novou a tu vždy načte pod náhodně vygenerovaným názvem do QGIS.

Dialogové okno Workflow Builderu spouští a ukládá workflow přes instanci třídy **Graph**.

2.2 Spuštění workflow

Workflow se spouští tlačítkem *Execute*. Jakmile se uživatel rozhodne spustit celý proces (workflow), v grafu se vytvoří podgrafy. Ty jsou v podstatě souvislými komponentami grafu. Tvoří se rekurzivně tak, že se vytvoří podgraf třídy **SubGraph** a z prvotního seznamu všech modulů v grafu se vyjme jeden a vloží se do něj. Poté se hledají další moduly, které patří do stejné komponenty, do stejného podgrafu. Z původního seznamu všech modulů v grafu se vyjmou všechny moduly spojené s prvním modulem a uloží se do podgrafu, poté se z původního seznamu vyjmou moduly, které jsou spojené s předchozími moduly a tak dále dokud existují spojení. Zároveň se ukládají do podgrafu i spojení (instance třídy Connection). Pakliže již neexistuje další propojený modul a v původním seznamu ještě zůstali nějaké moduly, vytvoří se nový podgraf a postupuje se stejným způsobem jako u předchozího podgrafu. Jeli původní seznam modulů prázdný, znamená to, že všechny moduly z grafu jsou rozděleny do podgrafů.

Jakmile máme vytvořeny podgrafy, zjistíme, zdali je pro nás graf, resp. všechny jeho podgrafy, validní. To znamená, že se prochází každý podgraf a u každého modulu se kontroluje, zdali jsou u jeho modulů nastaveny všechny povinné vstupní parametry, případně jestli u nich existuje spojení. Pakliže se narazí na modul, u kterého není nějaký povinný vstupní parametr nastaven, uloží se do seznamu nevalidních modulů. Projdou-li se všechny moduly v podgrafu a alespoň jeden není v pořádku, není validní, vypíše se hlášení na lištu ve spodní části Workflow Builderu s informací, že některé moduly nejsou nastaveny a nemůže se pokračovat ve spouštění workflow. Zároveň se také označí moduly, o které se jedná.

Jsou-li všechny povinné vstupní parametry u všech modulů nastaveny nebo spojeny s jiným, zkontroluje se každý podgraf, zdali neobsahuje cyklus. To se provádí pomocí prohledávání grafu do hloubky [viz Ukázka kódu2.1]. Neprojde-li kontrola, vypíše se hláška, že graf obsahuje cyklus. Projde-li kontrola a graf neobsahuje cykly, začnou se spouštět postupně všechny podgrafy. Tím zjistíme, že jsou validní a neměli by se objevit žádné známé problémy.

```
        def find(v):
         # oznacim si vrchol
            v.mark = True
            eV = [seznam hran vychazejicich z vrcholu v]
            for e in eV:
                w = e[1] # koncovy modul spojeni
                # jedna se o puvodni vrchol
                if w.id is vv.id:
                    vv.loop =  True
                    break
                # prozkoumame ho
                if not w.mark:
                    find(w)

        V = [seznam vrcholu v podobe Module]
        E = [seznam dvojic pocatecni a koncovy modul spojeni]

        # prochazim vrchol podgrafu
        for vv in V:
            find(vv)
            for v in V:
                v.mark = False
            if vv.loop:
               # najdu-li cyklus
                return vv.loop

        return False
```

Ukázka kódu 2.1: Hledání cyklu v podgrafu

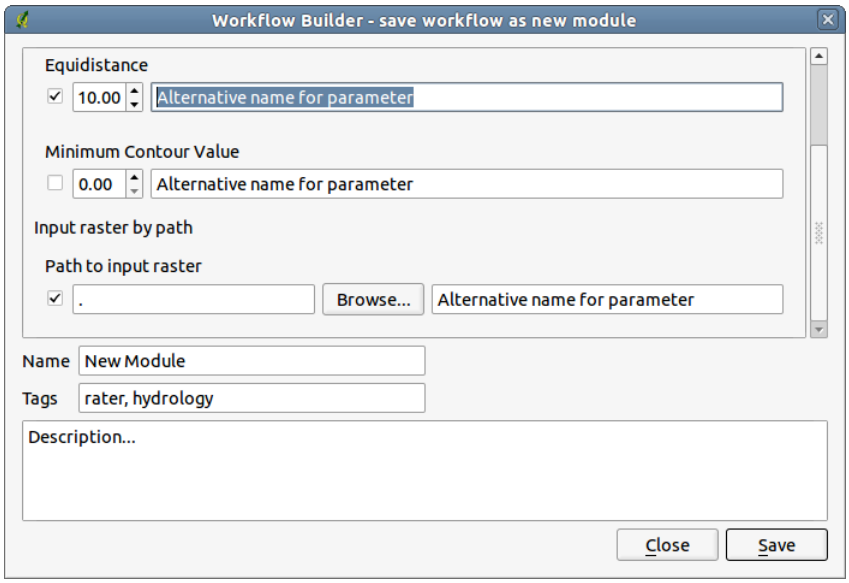
Samotné spouštění podgrafu začne tak, že se vezme libovolný modul z podgrafu a zkontroluje se, zdali jsou nastaveny všechny vstupy. Pakliže jsou všechny nastaveny, vytvoří se instance PF Modulu, nastaví se parametry a spustí se. Jsou-li výstupní parametry spojeny s jinými moduly, nastaví se hodnota parametru na druhém konci spojení právě získanou hodnotou. Pakliže nejsou některé vstupní parametry nastaveny,

sleduje se jejich spojení a pokusí se spustit předchozí modul. Pakliže i u něho jsou nějaké parametry nenastaveny, opět se sleduje jejich spojení. Vše se opakuje do té doby, dokud se nespustí nějaký modul a ten po úspěšném provedení nastaví vstupní hodnoty v grafu následujících modulů na základě svých výstupních hodnot. Postupně se nakonec spustí všechny moduly a výstupní data se uloží. Tento proces se také řeší rekurzí.

Pozn. kontrolují se pouze vstupní parametry, protože SAGA Plugin momentálně ignoruje, zdali nastavíme výstupní parametr či ne - vytvoří si vždy nový.

2.3 Uložení workflow

K uložení nového modulu slouží tlačítko *Save* ve spodní části postranního panelu. Nejdříve zkontroluje, zdali graf (workflow) neobsahuje cyklus. Je-li graf v pořádku, otevře se dialogové okno pro nastavení informací o novém modulu. Z obrázku [Obr.2.5] je vidět, že uživatel zadává jméno modulu, tagy, popis a hlavně parametry. U nich se uživatel rozhodne, zdali je chce v novém modulu zadávat anebo se nebudou měnit a tudíž si nastaví jejich hodnotu při tvorbě modulu a nebude je zaškrtávat. U parametrů, které se budou měnit, uživatel zaškrtne zaškrtávací pole. Případně může zadat alternativní název parametru, který se mu bude zobrazovat místo stávajícího.



Obr. 2.5: Workflow Builder - dialog pro uložení nového modulu

Po nastavení se klikne na tlačítko *Save*. Kontroluje se, zdali je zadán název nového modulu. Nový modul se uloží jako soubor ve formátu xml do $\$HOME/.qgis/python/workflows$ s názvem stejným jako název modulu.

2.3.1 Výstupní xml souboru

XML nabízí jednoduché uložení hierarchicky strukturovaných dat. O prvcích XML dokumentu hovoříme jako o elementech. Elementy jsou ohraničeny počátečními a koncovými značkami, tzv. tagy. XML dokument obsahuje vždy právě jeden kořenový ele-

ment, který se může skládat z dalších a dalších elementů. V našem případě je kořenový element Graph. Ten se skládá z minimálně jednoho podgrafu (SubGraph) a ten poté minimálně z jednoho modulu (Module). Podgraf dále může obsahovat spojení mezi moduly (Connection). Modul kromě toho obsahuje elementy parametr (Port) a tag (tag) a také popis. Tagy a popis jsou také obsaženy v Grafu.

atribut	příklad
name	Addition two rasters
tags	['raster', 'hydrology']

Tabulka 2.1: Atributy elementu Graph

Atributy elementu více méně korespondují s atributy objektů z Workflow Builderu. DOM reprezentace konkrétního parametru vypadá takto [Ukázka kódu2.2].

```
<Port connected="True" default_value="[]" id="944" moduleID="897"
      name="Raster" optional="False" porttype="1" should_be_set="False"
      type="processing.parameters.RasterLayerParameter" value="[]">
```

Ukázka kódu 2.2: příklad DOM reprezentace parametru

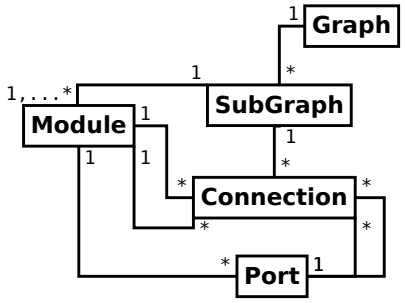
2.4 Načtení workflow do PF Manageru

Pro načtení byl napsán nový QGIS plugin **Workflow for Processing Framework Manager**, který načítá xml soubory z $\$HOME/.qgis/python/workflows$ adresáře. Na jejich základě vytvoří nové moduly, potomky processing.Module, a ty registruje v QGIS Processing Frametorku. Pro jednoduchou práci s daty uloženými v xml formátu se opět používá modul pythoní xml.dom.minidom.

Plugin načte a registruje nové moduly, když je on sám načten do QGISu. Podobně to platí i u Processing Manageru, který načte registrované moduly, když je poprvé spuštěn. Z toho plyne, že když uložíme nový modul, soubor se sice uloží, ale plugin ho hned automaticky nenačte. Aby se nový modul automaticky objevil v Processing Manageru, deaktivují se pluginy **Workflow for Processing Framework Manager** a **QGIS Processing Framework** a znovu se načtou. Poté opět otevřeme Processing Manager. Toto řešení je však kostrbaté.

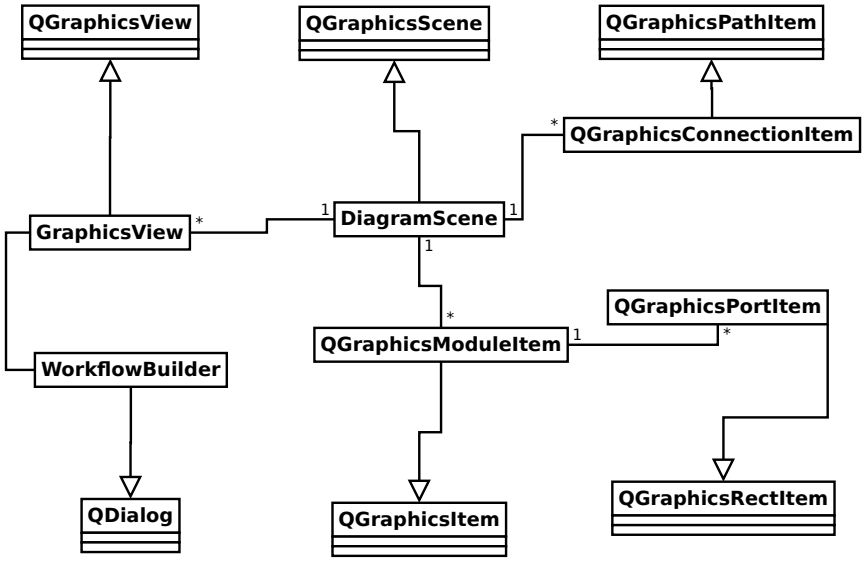
2.5 Třídy

O logickou část Workflow Builderu se starají třídy Graph reprezentující graf, SubGraph reprezentující podgraf, Module reprezentující modul, Connection reprezentující spojení a Port reprezentující parametr modulu. Na diagramu Obr.2.6 je znázorněný vztah po vytvoření podgrafů (souvislých komponent) v grafu. Existuje vždy právě jedna instance třídy **Graph**. Tato instance může obsahovat libovolné množství instancí třídy **SubGraph**. Každá tato instance obsahuje minimálně jednu instanci třídy **Module**, dále může obsahovat instance třídy **Connection**. **Module** může obsahovat instance tříd **Port**, **Connection** a právě jednu instanci třídy **SubGraph**. Spojení v sobě drží informaci o počátečním a koncovém parametru a modulu.



Obr. 2.6: Diagram znázorňující vztahy mezi třídami Graph, SubGraph, Connection, Module a Port

Dialogové okno je instance třídy **WorkflowBuilder**, která je potomkem třídy **QDialog** z knihovny Qt. WorkflowBuilder se skládá z **GraphicsView** (reimplementace třídy QGraphicsView z Qt). GraphicsView zobrazuje prvky skrze scénu (**DiagramScene** - potomek QGraphicsScene z Qt). Třídu Module reprezentuje ve scéně třída **QGraphicsModuleItem**, třídu Connection třída **QGrahicsConnectionItem** a parametry jsou reprezentovány **QGraphcisPort**.



Obr. 2.7: Diagram znázorňující vztahy mezi třídami GraphcisView, GraphicsScene, QGraphicsModuleItem, QGraphicsConncetionItem a QGraphicsPortItem v třídě WorkflowBuilder

Třída Graph

Třída Graph je v podstatě samotné workflow. Obsahuje všechny moduly a spojení, které se ve workflow vyskytují. Hlavní metody jsou *executeGraph*() a *save*(). Metoda *executeGraph*() postupně prochází všechny podgrafy a jsou-li validní a neobsahují cyklus, spouští jejich moduly. Validní podgraf je ten, u jehož každého modulu jsou všechny vstupní parametry buď nastaveny nebo spojeny s jiným. Metoda *save*() vytvoří xml soubor reprezentující nový modul a obsahující všechny podgrafy, moduly a spojení. Metoda *addConnection*() přidá do grafu spojení, *addModule*() přidá do grafu modul, *addSubGraph*() přidá do grafu podgraf, *findLoop*() prochází graf a vrací True, najde-li v grafu cyklus, *xml*() vytvoří DOM reprezentaci grafu.

Třída SubGraph

V řeči teorie grafů instance třídy **SubGraph** reprezentují souvislé komponenty grafu. V našem případě se jedná o instanci třídy **Graph**, která reprezentuje workflow.

Hlavní metody jsou *executeSGraph*() a *xml*(). Metoda *executeSGraph*() spouští všechny moduly v podgrafu. Metoda *xml*() je důležitá při ukládání nového modulu do souboru, vytvoří DOM reprezentaci podgrafu. Metody *prepareToExecute*() a *findLoop*() se spouští před samotným spuštěním podgrafu. Metoda *prepareToExecute*() prochází všechny moduly a zjišťuje, zdali jsou u každého modulu nastaveny vstupní parametry či jsou spojeny s jiným parametrem. Pakliže jsou, označí podgraf jako validní pomocí metody *setValid*(). Metoda *findLoop*() slouží k nalezení cyklu v daném podgrafu. Pomocí metod *addModule*() a *setConnections*() přidáme do podgrafu modul, resp. nastavíme spojení.

Třída Module

Třída Module reprezentuje PF Module v prostředí Workflow Builderu. Instance třídy v sobě uchovávají jméno, popis, tagy a parametry PF Modulu. Parametry se uchovávají v podobě instance třídy Port.

Důležité jsou metody *getInstancePF*(), *execute*() a *xml*(). Metoda *getInstancePF*() vrací již nastavenou instanci třídy PF Module, která koresponduje s modulem z Workflow Builderu. Pakliže u modulu ještě nebyla vytvořena instance PF Modulu, vytvoří ji pomocí *processing.framework*[*nazev_modulu*].*instance*(). **Port**ům modulu nastaví odkazy na parametry právě vytvořené instance třídy PF Module.

Metoda *execute*() nastaví instanci PF Modulu parametru podle aktuálních hodnot **Port**ů modulu a spustí instanci PF Modulu. Potom nastaví hodnoty výstupů z PF Modulu do **Port**ů modulu a dále nastaví hodnoty i u **Port**ů, které jsou s daným výstupem (Portem, parametrem) spojené.

Metoda *xml*() vytvoří DOM reprezentaci Modulu, která slouží pro uložení celého workflow do souboru formátu xml.

Instance třídy Module je jednoznačně identifikovatelná pomocí jejího identifikačního čísla, které je v rámci grafu (Graph) jedinečné.

Instance třídy **Module** jsou ve scéně reprezentována instancemi třídy **QGraphicsModelItem**.

Třída Port

Instance třídy **Port** reprezentují parametry PF Modulu. Jsou jednoznačně identifikovatelné pomocí identifikačního čísla, které je v rámci modulu jedinečné a pomocí identifikačního čísla modelu.

Uchovává v sobě informace jako název parametru, typ, zdali je parametr volitelný či povinný, zdali je parametr výstupní či vstupní, popis nebo výchozí hodnotu. Po úspěšném spuštění modulu a v případě, že je **Port** výstup, uloží se také nová hodnota.

Pomocí metody $getValue()$ získáme aktuální hodnotu, metoda $outputData()$ vrací výstupní data, $destinationPorts()$ vrací porty, které jsou s daným portem spojené a ve spojení jsou vedeny jako cílové, $getToolTip()$ vrací textový řetězec sloužící jako nápověda pro daný port, $isConnected()$ vrací zdali je daný port spojen s jiným a $xml()$ vrací DOM reprezentaci portu. Je-li **Port** výstupní, pomocí metody $addItToCanvas()$ zjistíme, zdali si uživatel přál načíst vrstvu po spuštění modulu do QGIS, a metoda $outputName()$ nám vrátí jméno, pod kterým se má vrstva načíst.

Instance třídy **Port** jsou ve scéně reprezentovány instancemi třídy **QGraphicsPortItem**.

Třída Connection

Třída **Connection** v terminologii teorie grafů reprezentuje hrany. Uchovává v sobě informaci o počátečním a koncovém modulu (Module), resp. parametru (Port). A obsahuje jedinou metodu xml(), která vrací DOM reprezentaci spojení.

Instance třídy **Connection** jsou ve scéně reprezentovány instancemi třídy **QGraphicsConnectionItem**.

Třída GraphicsView

Třída **GraphicsView** je reimplementací třídy **QGraphicsView** z knihovny Qt. Byla reimplementována metoda $wheelEvent()$, která umožňuje funkci zoom, a metody $dragEnterEvent()$, $dragMoveEvent()$ a $dropEvent()$ pro spravování událostí týkajících se prostředí Drag and Drop. **GraphicsView** přijímá pouze objekty z Processing

Manageru. Metoda *keyPressEvent*() je reimplementována tak, aby se po stisknutí klávesy *Delete* smazaly všechny vybrané prvky.

Třída DiagramScene

Třída **DiagramScene** je reimplementací třídy **QGraphicsScene** z knihovny Qt. Byly reimplementovány metody *mousePressEvent*(), *mouseMoveEvent*() a *mouseReleaseEvent*(). Tyto metody řeší, zdali uživatel pouze kliknul na modul a chce, aby se mu zobrazili informace o parametrech, či kliknul na parametr a chce jej spojit s jiným. Také se zde řeší, zdali mohou být parametry spojeny. Pakliže ano, vytvoří se spojení (instance třídy Connection) a na jeho základě také instance třídy **QGraphicsConnection**.

Pomocí metod *addModule*(processing.Module) se vytvoří nejdříve Objekt třídy Module a na jeho základě objekt **QGraphicsModuleItem**. Metoda *delModule*() maže modul ze scény (DiagramScene) i z grafu (Graph) a zároveň i jejich spojení s druhými moduly. Metoda *delConnection*() maže spojení ze scény (DiagramScene) i z grafu (Graph).

Metoda *clearDockPanel*() smaže informace z pravého postranního panelu.

Kapitola 3

SEXTANTE

3.1 Srovnání QGIS Processing Framework v SEXTANTE

3.2 Srovnání Workflow Builder v SEXTANTE Modeler

Závěr

Byl vytvořen nástroj Workflow Builder pro QGIS Processing Framework. Je to nástroj, který nabízí uživateli grafickou cestou spojovat již existující moduly a vytvářet tímto způsobem moduly nové. Workflow Builder pracuje s moduly dostupných skrz rozhraní QGIS Processing Framework. Vývoj tohoto rozhraní se ale bohužel zastavil s uvolněním jiného frameworku SEXTANTE.

Verze SEXTANTE pro Quantum GIS se objevila na konci psaní této práce. Daný framework má v podobné cíle a v současné době se zdá být on tou lepší cestou pro QGIS, i když ne všechny moduly jsou momentálně plně funkční.

Během práce na Workflow Builderu pro QGIS Processing Framework jsem se více seznámil s knihovnou Qt, architekturou MVC a jejím Graphics View Frameworkem. Někde jsem četl, že s architekturou MVC se dá seznámit za pár minut, ale naučit se ji správně využívat může trvat měsíce, i roky. Musím přiznat, že v mém případě to platí stoprocentně. Tudíž pakliže by projekt QGIS Processing Framework pokračoval, pokusil bych se stávající kód přepsat do podoby, která by splňovala všechny zásady a pravidla architektury MVC, tak jak nám umožňuje Qt.

Ukázky kódu

Rejstřík

Použitá literatura

[1] Jiří Demel. *Grafy a jejich aplikace*. ACADEMIA, 2002. ISBN: 80-200-0990-6.

[2] *Module Support for QGIS Processing Framework*. URL: https://github.com/polymeris/qgis/wiki/Module-Support.

[3] *Online Reference Documentation*. 2011. URL: http://doc.qt.nokia.com/.

[4] Mark Pilgrim. *Ponořme se do Python(u) 3. Dive Into Python 3.* CZ.NIC, z. s. p. o., 2010.

[5] *PyQGIS Developer Cookbook*. 2012. URL: http://www.qgis.org/pyqgis-cookbook/index.html.

[6] *PyQt Class Reference*. URL: http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/classes.html.

[7] *Python v2.7.3 documentation*. URL: http://docs.python.org/.

[8] Quantum GIS Development Team. *Quantum GIS Geographic Information System*. Open Source Geospatial Foundation. 2009. URL: http://qgis.osgeo.org.

[9] Quantum GIS Development Team. *Quantum GIS Geographic Information System API Documentation*. Open Source Geospatial Foundation. URL: http://qgis.org/api/.

Příloha na CD

Seznam souborů na přiloženém CD:

- **inputparameters** - adresář s pluginem "Inputs parameters for WB"
- **processingmanager** - adresář se samotným QGIS Processing Frameworkem
- **saga** - adresář s pluginem "SAGA Plugin"
- **workflow_builder** - adresář s pluginem "Workflow for Processing Framework Manager"
- *readme.txt*

Příloha A - ukázka použití model/view architektury v PyQt4

```
1  import sys
2
3  from PyQt4.QtCore import *
4  from PyQt4.QtGui import *
5
6  class editStud(QWidget):
7      """
8          Widget, ktery se objevi pri editaci dat.
9      """
10     def __init__(self, index, parent=None):
11         super(editStud, self).__init__(parent)
12         self.index = index
13         self.setFocusPolicy(Qt.StrongFocus)
14         self.setAutoFillBackground(True)
15         self.layout = QHBoxLayout()
16         self.layout.setMargin(0)
17         self._initGui()
18
19     def _initGui(self):
20         name = QLineEdit(self.index.data().toString())
21         age = QSpinBox()
22         age.setValue(self.index.data(Qt.UserRole+4).toInt()[0])
23         self.layout.addWidget(name)
24         self.layout.addWidget(age)
25         self.setLayout(self.layout)
```

```
26
27      def age(self):
28          for child in self.children():
29              if isinstance(child, QSpinBox):
30                  return child.value()
31          return False
32
33      def name(self):
34          for child in self.children():
35              if isinstance(child, QLineEdit):
36                  return child.text()
37          return False
38
39  class Delegate(QItemDelegate):
40      """
41          Pomoci tohoto delegata nastavime, aby se student vypisoval modre
42          a studentka cervene. Dale nastavime, aby se pri editaci objevil
43          radek pro editaci jmena studenta(ky) a QSpinBox pro editaci veku.
44      """
45      def __init__(self, parent=None):
46          QItemDelegate.__init__(self, parent)
47
48      def createEditor(self, parent, option, index):
49          # nastaveni widgetu pro editaci dat
50          editor = editStud(index, parent)
51          return editor
52
53      def setModelData(self, editor, model, index):
54          # aktualizace dat v modelu podle editace
55          model.setData(index,QVariant(editor.name()))
56          model.setData(index,QVariant(editor.age()), Qt.UserRole+4)
57          sIndex = model.mapToSource(index)
58          toolTip = "<b>vek</b>: {0} <b>pohlavi</b>: {1}". \
59              format(index.data(Qt.UserRole+4).toInt()[0], index.data(Qt.UserRole+3).t
60          model.setData(index,QVariant(toolTip), Qt.ToolTipRole)
61
```

```
62      def paint(self, painter, option, index):
63          # pomoci teto funkce nastavime font jmena studentu a dale barvu jmena podle
64          painter.save()
65
66          # nastaveni fontu a barvy
67          painter.setPen(QPen(Qt.black))
68          painter.setFont(QFont("Times", 10, QFont.Bold))
69
70          if index.data(Qt.UserRole + 3).toString() == "female":
71              painter.setPen(QPen(Qt.red))
72          elif index.data(Qt.UserRole + 3).toString() == "male":
73              painter.setPen(QPen(Qt.blue))
74
75          value = index.data(Qt.DisplayRole)
76          if value.isValid():
77              text = value.toString()
78              painter.drawText(option.rect, Qt.AlignLeft, text)
79
80          painter.restore()
81
82  class ProxyModel(QSortFilterProxyModel):
83      '''
84      Proxy model bude vyhledavat podle jmena, pohlavi, veku a pokoje.
85      '''
86      def filterAcceptsRow( self, source_row, source_parent ):
87          result = False
88
89          useIndex = self.sourceModel().index(source_row, 0, source_parent)
90
91          name = self.sourceModel().data(useIndex, Qt.DisplayRole).toString()
92          room = self.sourceModel().data(useIndex, Qt.UserRole+5).toString()
93          age = self.sourceModel().data(useIndex, Qt.UserRole+4).toString()
94          sex = self.sourceModel().data(useIndex, Qt.UserRole+3).toString()
95          floor = self.sourceModel().data(useIndex, Qt.UserRole+2).toString()
96
97          if ( floor ):
```

```
98              result = True
99          elif ( name.contains(self.filterRegExp()) ):
100             result = True
101         elif (room.contains(self.filterRegExp())):
102             result = True
103         elif (sex.contains(self.filterRegExp())):
104             result = True
105         elif (age.contains(self.filterRegExp())):
106             result = True
107
108         return result
109
110
111 def main():
112     """
113         Hlavni funkce, kde se vytvori a naplni model, vytvori a nastavi proxy server
114     """
115     app = QApplication(sys.argv)
116
117     # vytvoreni modelu
118     #
119     kolej = QStandardItemModel()
120
121     # naplneni modelu daty
122     #
123     # vyvoreni moveho pokoje
124     item = QStandardItem("801c")
125     # Qt.UserRole+2 bude slouzit jako cislo patra
126     item.setData(8, Qt.UserRole+2)
127     # nastaveni ToolTipu napovedy, ktera se zobrazi kdyz prejedeme pres polozku
128     item.setToolTip("room no. {0} on {1}. floor".format(item.data(Qt.DisplayRole).to
129     # nastavime, aby se nemohla kolej editovat ani vybrat
130     item.setEditable(False)
131     item.setSelectable(False)
132     # pridame pokoj do koleje/modelu
133     kolej.appendRow(item)
```

```
134     # vytvoreni noveho prvku, studenta, ktereho vlozime do pokoje, bude jeho potomek
135     itemS = QStandardItem("Julius")
136     itemS.setData("male", Qt.UserRole+3)
137     itemS.setData(27, Qt.UserRole+4)
138     itemS.setData(item.data(Qt.DisplayRole).toString(), Qt.UserRole+5)
139     itemS.setToolTip("<b>age</b>: {0} <b>sex</b>: {1}".format(itemS.data(Qt.UserRole
140     item.appendRow(itemS)
141
142     # vytvorime novy pokoj a naplnime ho studenty
143     item = QStandardItem("604ab")
144     item.setData(6, Qt.UserRole+2)
145     item.setToolTip("room no. {0} on {1}. floor".format(item.data(Qt.DisplayRole).to
146     item.setEditable(False)
147     item.setSelectable(False)
148     kolej.appendRow(item)
149     itemS = QStandardItem("Maria")
150     itemS.setData("female", Qt.UserRole+3)
151     itemS.setData(22, Qt.UserRole+4)
152     itemS.setData(item.data(Qt.DisplayRole).toString(), Qt.UserRole+5)
153     itemS.setToolTip("<b>age</b>: {0} <b>sex</b>: {1}".format(itemS.data(Qt.UserRole
154     item.appendRow(itemS)
155
156     itemS = QStandardItem("Fiorenza")
157     itemS.setData("female", Qt.UserRole+3)
158     itemS.setData(22, Qt.UserRole+4)
159     itemS.setData(item.data(Qt.DisplayRole).toString(), Qt.UserRole+5)
160     itemS.setToolTip("<b>age</b>: {0} <b>sex</b>: {1}".format(itemS.data(Qt.UserRole
161     item.appendRow(itemS)
162
163     # proxy model
164     #
165     proxyModel = ProxyModel()
166     # nastaveni zdrojoveho modelu
167     proxyModel.setSourceModel(kolej)
168     # nebudeme rozlisovat mala/velka pismena
169     proxyModel.setFilterCaseSensitivity(0)
```

```
170     #proxyModel.setDynamicSortFilter(True)
171
172     # vytvoreni pohledu(view)
173     #
174     kolejView = QTreeView()
175     # nastavime, aby se nam nezobrazovala hlavicka
176     kolejView.header().setVisible(False)
177     # nastavime, aby se nam strom pekne rozbaloval - samozrejmne to neni nutne
178     kolejView.setAnimated(True)
179     # nastavime model do pohledu
180     kolejView.setModel(proxyModel)
181
182     # vytvorime delegata
183     #
184     delegate = Delegate()
185     # nastavime model do pohledu
186     kolejView.setItemDelegate(delegate)
187
188     # vytvorime dialog, ve kterem se vse zobrazi
189     dorm = QDialog()
190     layout = QVBoxLayout()
191     dorm.setLayout(layout)
192     # radek pro filtrovani
193     filterBox = QLineEdit()
194     layout.addWidget(filterBox)
195     layout.addWidget(kolejView)
196
197     # propojeni radku pro filtrovani s proxy modelem
198     QObject.connect(filterBox, SIGNAL("textChanged(QString)"), proxyModel.setFilterRe
199
200     dorm.show()
201     app.exec_()
202
203 main()
```

Ukázka kódu 3.1: dorm.py - ukázka použití model/view architektury v PyQt4