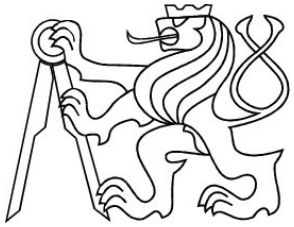
České vysoké učení technické v Praze
Fakulta stavební



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 na konci 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 (QGIS) nástroj, který by umožňoval uživateli grafické propojování modulů z frameworku **QGIS Processing Framework**. V úvodních dvou kapitolách je představeno prostředí QGIS a popsána práce s QGIS Processing Frameworkem jak z pozice uživatele GISu, tak z pozice vývojáře nových modulů. Dále je zmíněn SAGA Plugin, který byl napsán v rámci onoho frameworku.

V dalších částech diplomové práce představím z rychlíku knihovnu Qt, resp. její verzi PyQt pro jazyk Python, ve kterém byl celý Workflow Builder napsán.

V poslední kapitole diplomové práce je představena samotná aplikace umožňující grafické propojování modulů z QGIS Processing Framework Manageru do grafu, jejich postupné spuštění a případné uložení jako nový modul.

Klíčová slova

Quantum QGIS, workflow, open source, GIS, PyQt, SAGA, 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 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, řešení. A je pěkné, 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 implementací nástroje do programu Quantum GIS, který by uživateli umožňoval vytvářet vlastní funkce spojováním již existujících funkcí.

Můžeme se také setkat s pojmy jako model builder či chaining, v té práci budu používat pojem workflow builder. Tento název byl převzat z projektu VisTrails, který byl inspirací pro grafiku. 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 je mnohem pohodlnější, pakliže může najít všechny funkce na jednom místě (tzv. toolbox), nežli při hledání procházet všechny možné pluginy.

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 (tzv. toolbox). 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 - System for Automated Geoscientific Analysis (SAGA GIS). V době psaní existovala podpora pro 170 modulů, ne všechny ale byly testovány a fungovaly správně. Ale i přesto se mohlo začít s práci na workflow builderu.

Aktuální verzi workflow builderu můžete najít zde:

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

Kapitola 1

Teorie

V první části této kapitoly představím programovací jazyk Python, ve kterém byl napsán Workflow Builder. Jazyk Python je v dnešní době stále více oblíbený a můžeme ho najít snad všude, kam se podíváme. V druhé čáasti této kapitole představím jeden ze svobodných systémů pro práci s geografickými daty - Quantum GIS a možnost rozšiřování funkcionality pomocí zásuvných modulů, tzv. pluginů. To bylo původně možné jen v jazyce C++. Již nějakou dobu je také možné psát pluginy v jazyce Python, což přineslo výhody v podobě jednoduché šiřitelnosti (není nutná kompilace) a snažšího vývoje pluginů. V další části se budu věnovat knihovně Qt, respektive její verze 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 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 jazyka Python pro práci s objekty ve formátu XML.

1.1 Python

Python je interpretovaný, procedurální a objektově orientovaný jazyk, ve kterém se rychle programuje. Existuje pythoní verze Qt - PyQt.

1.2 Quantum GIS

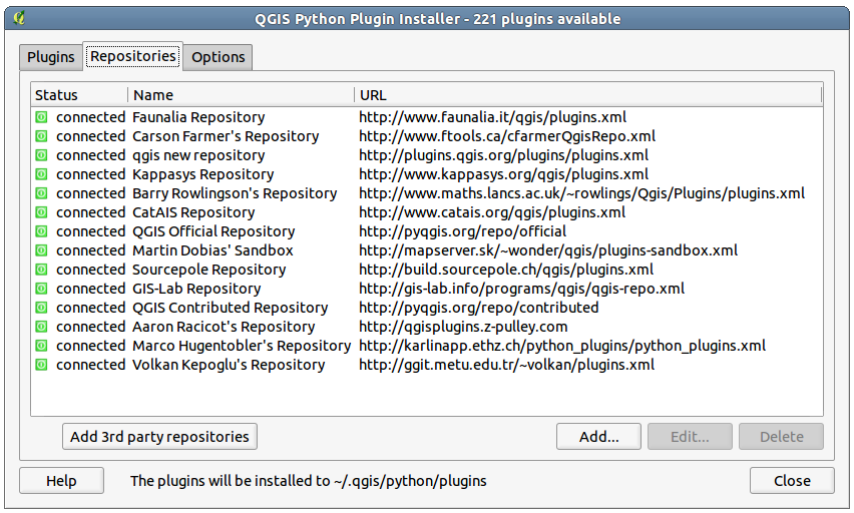
Quantum GIS je nejen prohlížečka geografických dat dostupná pro MS Windows, GNU/Linux, Unix, BSD a Mac OS X. Quantum GIS podporuje díky knihovně OGR většinu vektorových formátů dat jako například ESRI Shapefile, GRASS, MapInfo či GML a díky knihovně GDAL mnoho rastrových formátů jako TIFF, ArcInfo, GRASS raster, ERDAS a další. Přes Quantum GIS můžeme také přistupovat k datům uložených v geodatabázích PostGIS a SpatiaLite či k datům dostupných přes WMS a WFS služby. Quantum GIS je šířen pod licencí GNU Public Licence. [1]

Program je napsán v jazyce C++. Poslední stabilní verze nese označení 1.7.4. Quantum GIS je lehce rozšiřitelný program pomocí pluginů, které mohou být psány na jazyce Python nebo C++. Quantum GIS má poměrně dobře zdokumentované API a nutno také podotknout, že komunita kolem Quantum GISu je aktivní a podpora skrze mailinglisty je na vysoké úrovni.

Systém začal vyvíjet v roce 2002 Gary Sherman. Mělo jít o nenáročnou prohlížečku geodat pro Linux s širokou podporou datových formátů. Dlouhou dobu byl Quantum GIS brán převážně jako grafická nadstavba pro jiný desktopový GIS - GRASS GIS. Přes GRASS Plugin QGIS zpřístupňuje řadu modulů GRASS GIS. V současnosti se na vývoji nejvíce podílí skupina vývojářu kolem organizace [2] Faunalia.

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

[2] http://www.faunalia.co.uk/quantumgis



Obr. 1.1: QGIS Python Plugin Installer - správa repozitářů

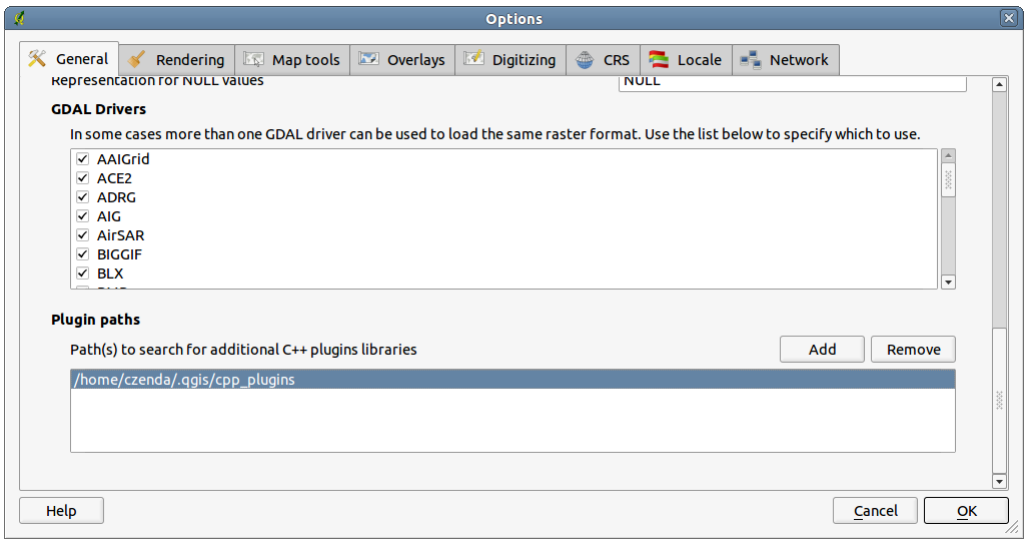
Funkcionalitu Quantum GIS rozšiřuje množství pluginů. Jako základní pluginy bych označil [3] **fTools** , který umožňuje pokročilé prostorové analýzy nad vektorvovými daty, [4] **GdalTools** pro práci s rastrovými daty a již zmíněný [5] **GRASS Plugin** plugin, který zpřístupňuje funkce GRASSu uživatelům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

1.2.1 Správa pluginů

Jak už bylo zmíněno Quantum GIS umožňuje uživatelům rozšiřovat funkce programu dle jejich potřeb v podobě 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 můžete 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 stáhnou do adresáře:

- *$HOME\.qgis\python\plugins* - v případě OS GNU/Linux

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



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

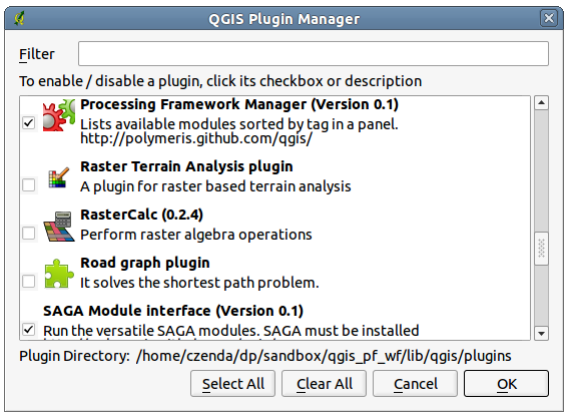
- *C:\Documents and Settings\USER\.qgis\python\plugins* - v případě OS Windows bývá cesta podobná této

Pakliže uživatel napíše plugin v jazyce Python, doporučuji ho uložit do tohoto adresáře. Je také sice možnost uložit plugin do adresáře *$QGIS_INSTALL_DIR \share\qgis\python\plugi* ale při případné opětovné kompilaci by byly změny ztraceny.
Pluginy psané v C++ se po přeložení ukládají standardně v *$QGIS_INSTALL_DIR \lib\qgis\plugins* případně uživatel může přidat nová úložiště 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...* [Obrázek 1.3.

1.2.2 Psaní vlastního pluginu

Pluginy mohou být psány na jazyce C++ a Python. Již z charakteristiky daných jazyků vyplývá, že pro jednoduché, nenáročné či na začátku vývoje pluginy, 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 sáhnout po jazyce C++. Obecně jsou



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

programy psané v kompilovaných jazycích mnohem rychlejší než programy psané v jazycích interpretovaných.

1.2.3 Python plugin

Při psaní pluginu v jazyce Python využíváme nástroje PyQGIS. Kromě dokumentace k Quantum GIS API také doporučuji [4]. Dále můžeme využít nástroj Plugin Builder, což je v podstatě také plugin, který vygeneruje základní soubory, kód, který potom začneme upravovat podle tak, aby náš plugin dělal to, co chceme.

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 [6]QGIS Coding and Compilation Guide.

[5]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.

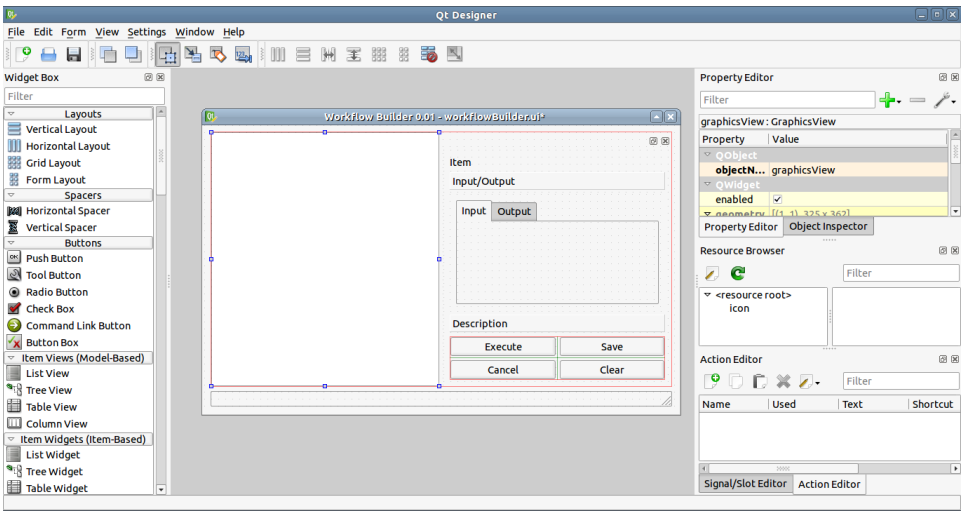
Existuje také mimo jiné verze pro Python - v současnosti verze PyQt4. PyQt je vyvíjena firmou Riverbank Computing. Z rodiny Qt, resp. PyQt, se v této práci využila samotná knihovna pro psaní kódu, obzvláště pak její Graphics View Framework, a program QtDesigner.

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 grafiky Workflow Builderu.

Soubor vytvořený v Qt Designer (s příponou **.ui**) lze jednoduše přeložit programem **pyuic4** do pythoního kódu, který lze poté dále použít.

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

Ukázka kódu 1.1: 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 $connect(kdoVyslal, SIGNAL("clicked()"), SLOT())$. Slot je v podstatě metoda či funkce, která se zavolá na základě nějakého podmětu, 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 metody

emit(SIGNAL(),...), kde v $SIGNAL()$ uvedeme název signálu a dále za ním parametry, které se s ním vyšlou. Poté je spojíme se slotem, metodou, která se na jeho základě spustí, pomocí metody $QObject.connect(QObject, SIGNAL, SLOT)$. Tato metoda je statická, tudíž ji můžeme volat přímo z **QObject**.

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 1.2: vyslání slotu pod názvem "*jdu*" s atributem "*domu*"

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

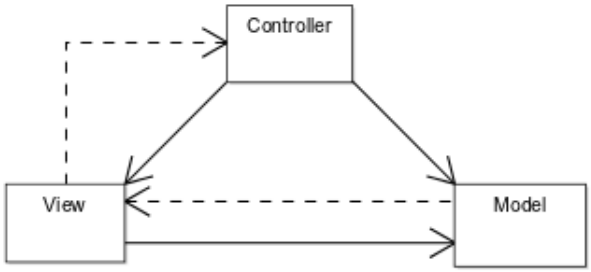
```
1    class Marie(Object):
2        def __init__(self):
3            QObject.__init__(self)
4            self.connect(tonda, SIGNAL("jdu"), self.jednat)
5            # mohou nasledovat dalsi spojeni
6
7        def jednat(self, parametr):
8            # tady se Marie muze rozhodnout na zaklade parametru,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

Ukázka kódu 1.3: 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. Velikou výhodou také je, že jeden model se může zobrazit v několika různých pohledech vždy jiným způsobem.

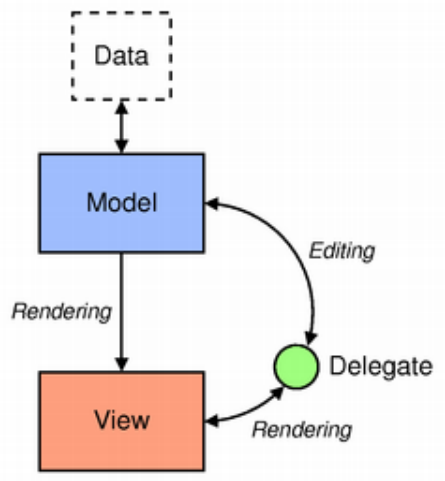


Obr. 1.5: Propojení jednotlivých částí architektury MVC

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

V Qt se implementace MVC architektury objevila s verzí Qt4 v podobě model-view-delegate. Kde funkci *controller*u přebírá částečně *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 **Příloha 1**. V příkladu 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 dat
- **Delegate** pomocí delegáta můžeme definovat vlastní widgety pro slouží editaci dat a také může 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 poohlídnout po dalších abstraktních třídách **QAbstractListModel**, resp. **QAbstractTableModel** implementující další prvky, které jsou pro daný model typické. Jak je již z názvu vidno, 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 ale plně vyhovovat, můžeme si kterýkoliv vybrat a reimplementovat jej.

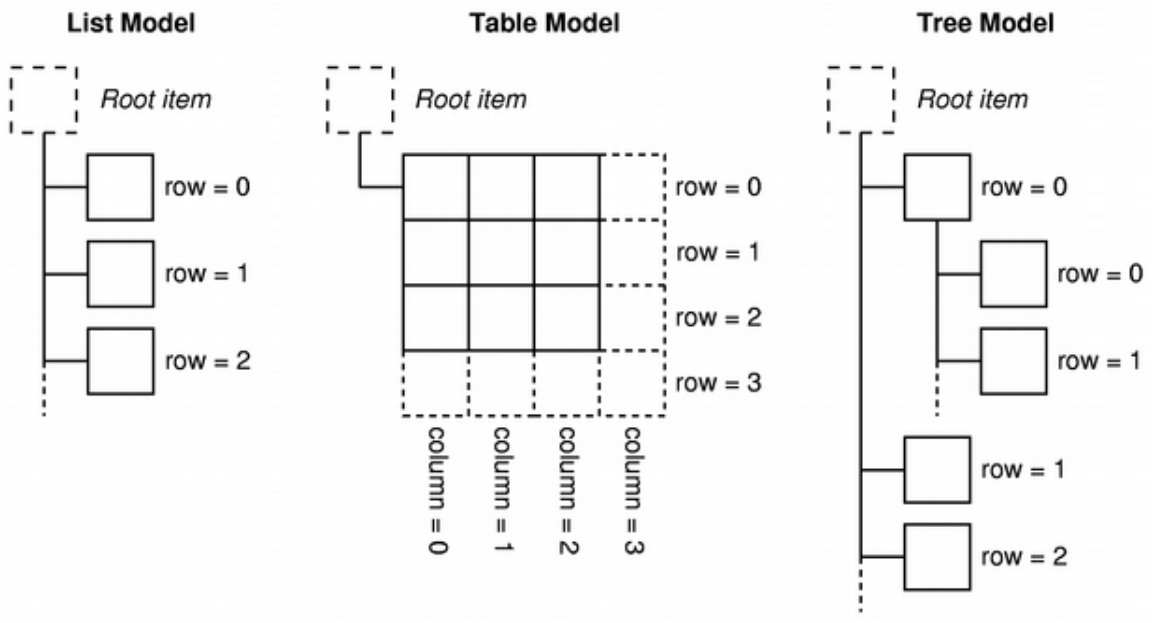
Dále existují tzv. proxy modely, které stojí mezi view a "standardním"modelem a poskytují podporu pro zpracování dat. Například **QSortFilterProxyModel**, který umožňuje uživateli vytvořit pravidla pro řazení a filtraci dat.

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



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

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

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 $data(role)$.

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 $toPyObject()$ [v ukázce 1.4].

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 s **QStandardItem** pomocí metody *data*(*role*). Model s QStandardItemModel s QStandardItem se hodí pro hierarchicky uspořádaná data. [ukázka použití QStandardItem 1.4

```
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 1.4: QStandardItem - vytvoření a získání dat

Obecně tedy data ukládáme pomocí metody *setData* 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 se nastaví pomocí metody *setItemDelegate*(QAbstractItemDelegate delagate). [ukázka viz 1.5]

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

Ukázka kódu 1.5: 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 defalutně 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 kódu **??**] využijeme QStyledItemDelegate a přepíšeme metodu *paint* například takto:

```
1  class Delegate(QItemDelegate):
2      def __init__(self, parent=None, *args):
3          QItemDelegate.__init__(self, parent, *args)
4
5      def paint(self, painter, option, index):
6          painter.save()
```

```
7
8           # nastaveni fontu
9           painter.setPen(QPen(Qt.black))
10          painter.setFont(QFont("Times", 10, QFont.Bold))
11
12        # nastaveni barvy podle pohlavi
13          if index.data(Qt.UserRole + 3).toString() == "female":
14              painter.setPen(QPen(Qt.red))
15          elif index.data(Qt.UserRole + 3).toString() == "male":
16              painter.setPen(QPen(Qt.blue))
17
18          value = index.data(Qt.DisplayRole)
19          if value.isValid():
20              text = value.toString()
21              painter.drawText(option.rect, Qt.AlignLeft, text)
22
23          painter.restore()
```

Ukázka kódu 1.6: Delegate - přepsání metody *paint*

V tomto příkladě 1.6 předpokládáme, že data (v podobě indexů) obsahují informaci o pohlaví uloženou pod rolí $Qt.UserRole + 3$.

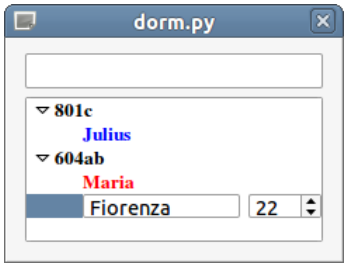
Editor (widget pro editaci dat) nastavíme reimplementací metody *createEditor*, která vrací QWidget. Aby se po editaci změnila data v modelu, musíme také reimplementovat metodu *setModelData*. V ukázce kódu 1.7 předpokládáme, že třída *editStud* je widget, který je složen z dvou dalších widgetů - QLineEdit pro editaci jména a QSpinBox pro nastavení věku.

```
1  class Delegate(QItemDelegate):
2      def __init__(self, parent=None):
3          QItemDelegate.__init__(self, parent)
4      def createEditor(self, parent, option, index):
5        # editStud je widget slozeny z QLineEdit a QSpinBox
6          editor = editStud(index, parent)
7          return editor
```

```
8     def setModelData(self, editor, model, index):
9         model.setData(index,QVariant(editor.name()))
10        model.setData(index,QVariant(editor.age()), Qt.UserRole+4)
```

Ukázka kódu 1.7: Delegate - přepsání metod *createEditor* a *setModelData*

Použití QStandardItemModel s QTreeView za použití proxy modelu a delegáta z ukázek 1.6 a 1.7 můžeme dostat výsledek podobný tomuto:



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

V horní části vidíme widget QLineEdit, která 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). Celý příklad najdete v příloze.

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 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 akceptovány). 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 příjmem 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 grafických dvojrozměrných prvků. Nabízí také widget, ve kterém se dané prvky zobrazují. Podporuje funkce jako zoom nebo rotaci. Prostředí umožňuje spravovat klasické události jako kliknutí myši, její pohyb či stisknutí klávesy.

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

QGraphicsItem

QGraphicsItem je základní třída, ze které vychází ostatní grafické 2D prvky. Pomocí metody setPos se nastaví pozice v 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 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 1.8].

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

Ukázka kódu 1.8: Nastavení flagů u QGraphicsRectItem

QGraphicsScene

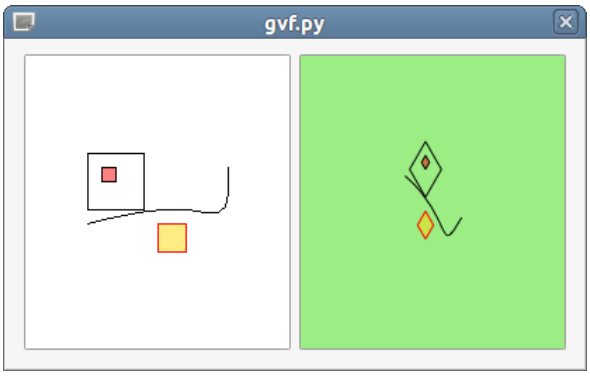
Do scény se prvky přidávají pomocí metody *addItem*(QGraphicsItem item) a mažou se ze scény pomocí *removeItem*(QGraphicsItem item). Pro smazání všech prvků ve scéně použijeme metodu *clear*(). Další užitečné metody jsou například *items*(), která nám vrací všechny prvky scény, *itemAt*(QPointF point), která nám vrací prvek na vybrané pozici, či *selectedItems*(), která nám vrací seznam vybraných prvků.

QGraphicsView

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

rotovat scénu atp.

Na obrázku je vidět jedna scéna zobrazena ve dvou rozdílných pohledech. Změna scény provedená v jednom pohledu, se projeví také v druhém pohledu.



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

1.3.5 VisTrails, Orange

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 VisTrails.

Nakonec mě nejvíce oslovilo grafické zpracování VisTrails. Uživatel na první pohled vidí, který parametr je s kterým spojen. Ne pouze který modul je s kterým spojen jak to často bývá u podobných programů. Jal jsem se tedy studovat kód a využil jsem prvky scény QGraphicsModule, QGraphicsPort a QGraphicsConnection.

Obrazek

1.4 QGIS Processing Framework

QGIS Processing Framework vznikl v rámci projektu [7]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.

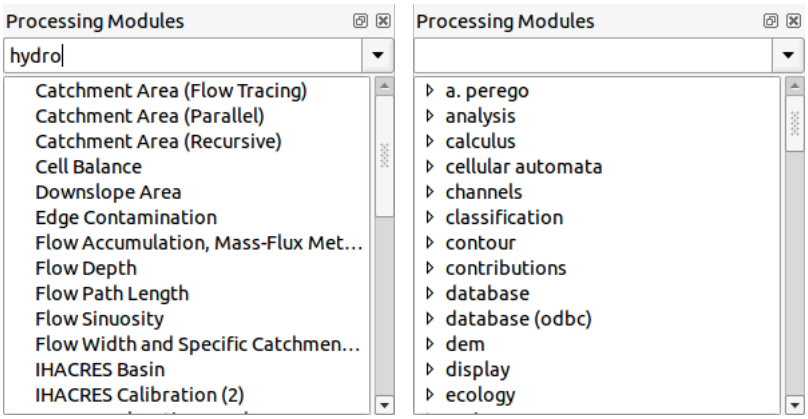
V době psaní této práce byla na světě první verze Processing Frameworku a vše nasvědčovalo tomu, že práce na frameworku budou pokračovat a nástroje v Processing Frameworku budou přibývat. Existovala totiž pouze částečná podpora pro funkce z SAGA GIS a plugin zpřístupňující funkce Orfeo Toolboxu (OTB).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

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

Napsat o tom alespoň odstavecek Processing Manager zpřístupňuje z jednoho místa všechny moduly dostupné skrze QGIS Processing Framework. Moduly jsou rozděleny podle tagů do různých skupin, větví. 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 ještě popis, ale protože se tagy generují z tohoto popisu, nemá cenu popis procházet Obr.1.10.

Dostupnost skrze pythoní konzoli v QGISU. Nacitani modulu, nastavování parametrů, spuštění. takže asi zase malé tabulky - metody a atributy Modulu, Parametru.

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



Obr. 1.10: QGIS Processing Framework - Processing Manager

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 [[1]] 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 1.3. 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.

1.4.2 Psaní pluginu pro PF

QGIS Processing Framework funguje tak, že si každý může při psaní pluginu pro QGIS může při dodržení pár pravidel *"zaregistrovat"* zaregistrovat svůj modul do frameworku.

Každý plugin může mít několik vstupních a výstupních parametrů. V současné době dovoluje framework uživateli použít parametry 1.4.

Obr. Dialogového okna

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

Tabulka 1.3: parametry SAGA GIS podporované Processing Frameworkem

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 1.4: parametry podporované Processing Frameworkem

1.4.3 Závěr

Chtělo by to vyřešit vstupní vrstvy aby například rastrová vrstva zadaná jako PathParameter byla kompatibilní s parametrem RasterLayerParameter. Dát vývojáři pluginu možnost dát uživateli možnost zadat vrstvu buď pomocí cesty nebo výběrem z jich načtených vrstev. Dát uživateli obě možnosti.

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 nad SEXTANTE. QGIS Processing Framework se tedy zdá být mrtvým projektem. Alespoň prozatím...

1.5 .xml.dom.monidom

Nově vzniklé workflow se ukládají ve formátu XML. Výhoda tohoto formátu je, že se s ním snadno pracuje a jde v podstatě pouze o textový dokument.

Python nabízí několik možností pro zpracování XML dokumentů.

XML nabízí jednoduché uložení hierarchicky strukturovaný dat. O prvcích XML dokumentu hovoříme jako elementech. Elementy jsou ohraničeny počátečními a koncovými znaky, tzv. tagy. XML dokument obsahuje vždy právě jeden kořenový element. Ten se může skládat z dalších a dalších elementů. Elementy mohou obsahovat atributy (dvojice jméno="hodnota"). Atributy se nesmějí v opakovat. Elementy kromě toho mohou obsahovat text. Text se uvádí mezi počáteční a koncový znak.

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 1.9: Příklad XML dokumentu

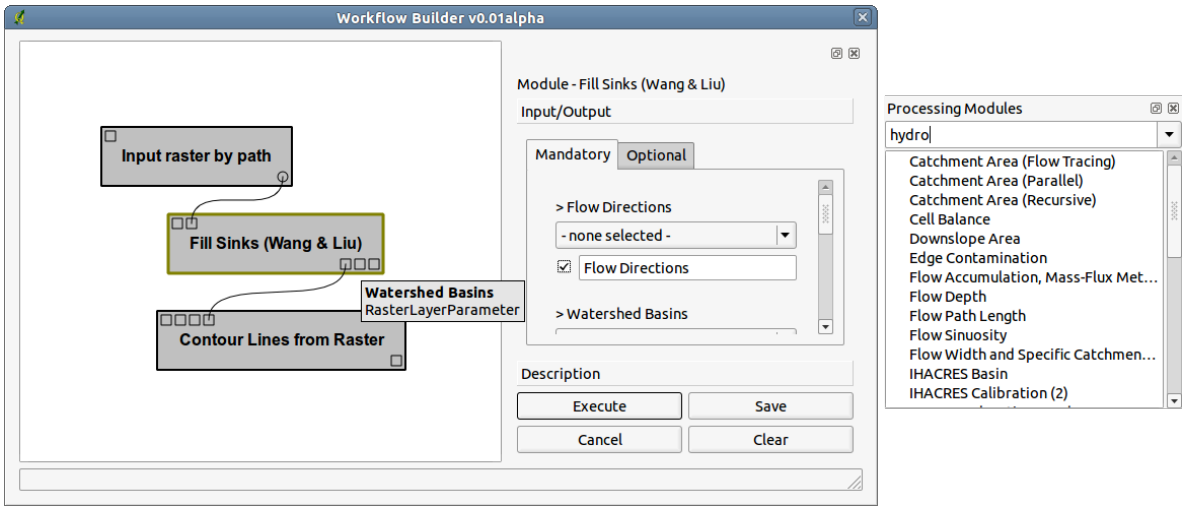
Pro práci s XML dokumenty existuje v prostředí jazyka Python několik modulů. Mě se osvědčil modul xml.dom.minidom. Na začátku je potřeba si vytvořit objekt (*Document*), který bude reprezentovat celý XML dokument. *Document* je, stejně jako všechny ostatní elementy XML dokumentu, je podtřídou třídy *Node*. Do takto vytvořeného objektu se poté mohou

načtení xml

———-

Kapitola 2

Workflow Builder



Obr. 2.1: Workflow Builder

Workflow Builder umožňuje uživateli propojovat moduly z QGIS Processing Framework. Výsledný graf 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, jejich propojování pomocí myši. V panelu v pravá části lze zadávat jednotlivé parametry modulů, které nejsou propojeny. V pravé spodní části se nachází tlačítka pro spuštění a uložení procesu, pro smazání scény a pro vypnutí dialogového okna. Jakmile uživatel klikne na tlačítko pro uložení, otevře se nové dialogové okno, kde bude vyzván k nastavení nového modulu. Jméno modulu, tagy, 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.

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 ve formátu XML. Dále se může hodit **Input parameters for WB**, který přidává do Processing Frameworku moduly, které slouží pro načítání vektorových a rastrových dat se souboru.
Pro začátek můžete shlédnout instruktážní video:

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

2.1 Tvorba workflow

Při spuštění Workflow Builderu se vytvoří objekt Graph. Do něho se postupně přidávají či z něj mažou moduly (třídy Module) a spojení (třídy Connection). Při přetažení PF Modulu do scény se vytvoří modul Module, který se uloží do Graphu. Modul obsahuje parametry PF Moduly, které jsou reprezentovány třídou Port. Grafická reprezentace Module je QGraphicsModule, který také podle Portů v Module vytvoří QGraphicsPort. Při spojování portů mezi sebou se kontroluje, zdali koresponduje typ (RasterLayerParameter, NumericPrameter, ...), spojuje-li se vstupní parametr s výstupním, zdali nejsou oba parametry parametry stejného modulu a pokud je vstupní parametr prázdný (to znamená, že není spojený s jiným parametrem). Pakliže jsou splněny všechny podmínky, vytvoří se spojení třídy Connection a jeho grafická reprezentace QGraphicsConnection. Connection se poté přidá do Graphu. Pro tvorbu spojení stačí kliknout na požadovaný vstup/výstup a táhnout myší na druhý parametr.

V Graphu máme tedy uloženy moduly a spojení mezi nimi (Module a Connetion). Jsou uloženy jako slovníky - id:Module, resp. id:Connection. Ty se během tvorby mění jak uživatel přidává a odebírá moduly, spojuje je a maže spojení. Zrušit modulu či spojení můžeme tím, že si jej myší označíme a stiskneme klávesu *Delete*.

2.2 Spuštění procesu

Jakmile se uživatel rozhodne spustit celý proces (workflow) začíná se tvořit graf.

Tvoří se rekurzivně tak, že ze se vytvoří podgraf třídy SubGraph, z prvotního seznamu všech modulů v grafu se vyjme jeden a vloží se de něj. Poté se vyjmou z původního seznamu všechny moduly spojené s prvním modulem a vkládají se 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. Pakliže již neexistuje další propojený modul a v původním seznamu ještě zůstali nějaké moduly, vytvoří se nový podgraf a postupuje se stejně jako u předchozího. Jakmile nezůstali žádné moduly v seznamu, začneme postupně spouštět workflow.

Postupně se spouští každý podgraf. První 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 modul nastaven, uloží se do seznamu nevalidních modulů. Jakmile se projdou všechny moduly v podgrafu všechny jsou v pořádku, začne se s kontrolou, zdali podgraf neobsahuje smyčku. Jeli stále vše v pořádku začneme s rekurzivním spouštění modulů. Pakliže nejsou všechnu moduly v pořádku, zastaví se provádění procesu spuštění a označí se moduly, které je třeba ještě nastavit, a ve spodní liště Module Builderu se vypíše poznámka, že je třeba označené moduly nastavit. Pakliže podgraf obsahuje smyčku, proces se také zastaví a také vypíše poznámku na spodní liště.

Obrázek se smyčkou

Obrázek se s nenastavenými moduly

Zde bych chtěl podotknou

pseudokód rekurzivní ho spouštění modulů

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

XML soubor se uloží do $\$HOME/.qgis/python/workflows$.

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

Tabulka 2.1: atributy elementu Graph

2.3.1 Popsání výstupního xml souboru

XML nabízí jednoduché uložení hierarchicky strukturovaný dat. O prvcích XML dokumentu hovoříme jako elementech. Elementy jsou ohraničeny počátečními a koncovými značkami, tzv. tagy. XML dokument obsahuje vždy právě jeden kořenový element. Ten 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 popis. Graf také obsahuje tagy a popis.

2.4 Načtení workflow do PF Manageru

Kapitola 3

SEXTANTE

3.1 Srovnání QGIS Processing Framework v SEXTANTE

3.2 Srovnání Workflow Builder v SEXTANTE Modeler

Závěr

Během práce na Workflow Builderu pro QGIS Processing Framework jsem se více seznámil s knihovnou Qt a jejím Graphics View Frameworkem.

Dále musím zmínit existenci druhého frameworku, který se objevil v konci psaní této práce. SEXTANT pro Quantum GIS. Daný framework má v podstatě podobné cíle a v současné době se zdá být on tou pravou cestou pro qgis, ikdyž ne všechny moduly jsou momentálně plně funkční.

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

Bibliography

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

[2] *Official homepage of Quantum GIS*. URL: http://www.qgis.org.

[3] Mark Pilgrim. *Ponořme se do Python(u) 3. Dive Into Python 3.* CZ.NIC, z. s. p. o., 2010. URL: http://qgis.org/api/.

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

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

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