SVEUČILIŠTE U ZAGREBU – GEODETSKI FAKULTET
UNIVERSITY OF ZAGREB – FACULTY OF GEODESY
Zavod za geomatiku; Katedra za geoinformatiku
Institute of Geomatics; Chair of Geoinformatics
Kačićeva 26; HR-10000 Zagreb, CROATIA
Web: http://www.geof.hr; Tel.: (+385 1) 46 39 222; Fax.: (+385 1) 48 28 081

Usmjerenje: Geoinformatika, Diplomski studij geodezije i geoinformatike

DIPLOMSKI RAD

Razvoj proširenja za QGIS platformu u Pythonu

Izradio:

Siniša Slovenec
Jalšje 22
49214 Veliko Trgovišće
sslovenec@geof.hr

mentor: prof. dr. sc. Damir Medak

Zagreb, rujan 2012.

I. Autor	
Ime i prezime:	Siniša Slovenec
Datum i mjesto rođenja:	18. 10. 1987., Zabok

II. Diplomski rad	
Naslov:	Razvoj proširenja za QGIS platformu u Pythonu
Mentor:	prof. dr. sc. Damir Medak
Voditelj:	Dražen Odobašić, dipl. ing.

III. Ocjena i obrana	
Datum zadavanja zadatka:	27. 01. 2012.
Datum obrane:	14. 09. 2012.
Sastav povjerenstva pred kojim je branjen diplomski rad:	prof. dr. sc. Damir Medak prof. dr. sc. Drago Špoljarić dr. sc. Ivan Medved

Izrada proširenja za QGIS platformu u Pythonu

Sažetak: *Ovaj rad opisuje postupak izrade proširenja za Quantum GIS platformu. Proširenje je razvijeno u programskom jeziku Python. Za izradu je korištena PyQt biblioteka. Kao rezultat, razvijeno je proširenje koje omogućuje brzu izmjenu atributnih vrijednosti odabranih vektorskih geografskih objekata. Također omogućuje efikasnije uređivanje skupa atributa jednog vektorskog sloja. Dokumentiranjem procesa izrade utvrđene su tehnologije i paradigme neophodne za razvoj proširenja te mogućnosti QGIS platforme.*

Ključne riječi: *QGIS, Python, Qt, proširenje*

Development of QGIS plug-in with Python

Abstract: *This paper describes the process of developing an extension for Quantum GIS using Python programming language and modules from Qt and QGIS libraries. As a result, a plug-in was developed and it can then be used to edit attribute values of the selected feature or to edit attribute fields of a particular vector layer. Analysis of the plugin development process pinpointed essential resources and paradigms used in the process and it also demonstrated capabilites of QGIS platform.*

Keywords: *QGIS, Python, Qt, plug-in*

SADRŽAJ

1. UVOD

Odabrati ovu temu nije bio težak posao. Još kao učenik gimnazije volio sam programiranje pa mi je ovo bila idealna prilika da produbim znanje i steknem iskustvo koje bi mi moglo zatrebati u profesionalnoj karijeri. Također, prateći svjetske trendove razvoja tržišta informacija shvatio sam kako je zahtjev za proizvodnjom kvalitetnih softverskih rješenja sve veći, što nije ni čudno pošto takvi proizvodi uvelike ubrzavaju svakodnevni rad s podatcima, a to se pogotovo odnosi na prostorne podatke.

Praktičan zadatak ovog rada je razviti proširenje (eng. *plug-in*) za Quantum GIS aplikaciju koristeći njezinu platformu te dokumentirati razvojni proces. QGIS platforma je skup modula i biblioteka koje sadrže gotove algoritme alata svojstvenih GIS aplikacijama, ali ne sadržavaju sve resurse potrebne za izradu aplikacija ili proširenja. Zbog toga je važno upoznati se s ulogama ostalih resursa neophodnih za razvoj, kao što su programski jezik Python i Qt platforma. Kako bi ovaj rad pomogao početnicima obrađene su i programske paradigme kao što su objektno-orijentirano programiranje i programiranje zasnovano na događajima.

Nakon toga će svi ti elementi biti iskorišteni za izradu proširenja te će biti objašnjen način korištenja pomoću konkretnih primjera. Drugim riječima, proučavanje procesa razvoja ovog proširenja nudi mogućnost upoznavanja s metodama kod općenitog programiranja aplikacija, ali i specifičnostima programiranja aplikacija svojstvenih području geoinformatike. Na posljetku je potrebno razmotriti ulogu i iskoristivost QGIS platforme u razvoju proširenja.

2. POSTAVLJANJE PROBLEMA

2.1. Implementacija

Iako izrada samostalne aplikacije pruža veću slobodu u pristupu i metodi programiranja, izrada proširenja ima svoje prednosti. Zapravo, najveća mana proširenja u odnosu na samostalnu aplikaciju jest da se to proširenje može koristiti isključivo u sklopu već gotove aplikacije.

Međutim, ova mana može se u mnogim slučajevima pokazati kao prednost jer proširenje ne treba zasebni pokretački dio već koristi onaj od aplikacije. Isto tako programer koji radi na proširenju već unaprijed ima dizajnirano okružje glavne aplikacije u kojem razvija nove funkcije koristeći postojeće funkcije aplikacije. Dakle ušteda vremena je još jedna pozitivna posljedica ovakvog pristupa.

Za izradu proširenja Quantum GIS-a moguće je koristiti i C++ programski jezik. Međutim, iako je C++ popularniji i češće korišteni programski jezik[1], Python nudi jednostavniju sintaksu i veću priklonost objektno-orijentiranoj metodi programiranja koja me osobno najviše zanima. Također, Python mi je svježiji jezik i početno znanje je opširnije pa je zbog toga odabran za rješavanje problema ovog diplomskog rada.

Da bi uspješno došli do rješenja i razvili proširenje, potrebno je prije svega oformiti ideju oko toga kakvu će proširenje imati svrhu i što njime želimo postići.

Naravno, taj problem ne mora biti isključivo vezan uz GIS domenu, ali s obzirom na to da je Quantum GIS alat za obradu prostornih podataka, bilo bi šteta ne iskoristiti

[1] http://langpop.com/

ogroman potencijal njegovih biblioteka u toj domeni. Također se moramo zapitati kakva je priroda problema koji želimo riješiti i da li već postoji slično rješenje.

Prigodno mjesto za pronalazak primjera je zajednica korisnika sličnih proširenja. Na taj način lako dolazimo do podataka kakva rješenja mogu obogatiti i pomoći zajednici, odnosno koja rješenja već postoje i koja prema tome nema smisla rekreirati.

Kako bi što bolje dobili osjećaj o načinu primjene našeg proširenja, valja pronaći aplikacije koje se bave rješavanjem problema istog tipa te proučiti njihovu strukturu i funkcionalnost u odnosu na namjenu. Takva aplikacija je sam Quantum GIS.

Nakon pregleda par internetskih foruma na kojima se vodi diskusija o Quantum GIS aplikaciji i njezinoj primjeni, odlučio sam napraviti proširenje kojem bi svrha bila izmjena i uređivanje vrijednosti atributa vektorskih podataka učitanih u Quantum GIS. Proširenje će nositi ime *Attribute Changer*, a osim ovog alata sadržavat će i alat za uređivanje skupa atributa pojedinog vektorskog sloja, nazvan *Attribute Settings*.

Važno je napomenuti, a posebno onima koji su početnici u programiranju, kako je već i u ovoj fazi poželjno imati osnovno znanje programiranja aplikacija kako bi bili sigurni da nam želje ne prelaze mogućnosti. Drugim riječima, moramo biti sigurni da kada zamislimo operacije koje će naša aplikacija izvoditi, da odmah možemo zaključiti da li je to moguće napraviti i na koji način ako jest. S takvim pristupom odmah možemo okvirno odrediti vrijeme potrebno za izradu, što je vrlo korisno kod ovakvog posla. Naravno da to osnovno znanje nije od kritične važnosti i moguće je učiti u hodu, ali onda treba računati na mogućnost zastoja u kodiranju, a time i prolongiranja vremena izrade.

2.2. Funkcije proširenja

Unutar Quantum GIS aplikacije već je moguće mijenjati vrijednosti atributa, ali prvo je potrebno ući u prikaz atributne tablice (*Open Attribute Table*), nakon toga uključiti mod za izmjenu vektorskog sloja, pretražiti tablicu da bi našli željeni geografski objekt (eng. *feature*) i na posljetku izmijeniti vrijednosti atributa te pohraniti promjene.

Proširenje će nuditi ubrzanje tog procesa i drugačiji pristup izboru geografskog objekta. Umjesto da vrijednosti atributa geografskog objekta tražimo u atributnoj tablici prepunoj podataka, označit ćemo geografski objekt na grafičkom prikazu te tako pozvati atributne vrijednosti svojstvene samo tom objektu gdje ih onda možemo mijenjati i spremati bez suvišnih međukoraka.

Osnovne funkcije ovog proširenja bile bi mogućnost pokretanja unutar Quantum GIS-a kao alata i registriranja svaki put kada klikom miša označimo geografski objekt vektorskog sloja (točka, linija ili poligon) u prikazu. Nakon označavanja trebao bi se pojaviti dijalog s popisom atributa tog objekta zajedno s vrijednostima atributa.

Vrijednosti atributa trebalo bi biti moguće zamijeniti upisom vlastitih vrijednosti, s time da program provjerava upis i upozorava ako nije u skladu s pravilima. Pritom bi dijalog imao dva gumba; gumb za spremanje nastalih promjena (*Save*) i gumb za odbacivanje promjena i zatvaranje dijaloga (*Cancel*).

Osim alata za promjenu atributa potreban je i alat za uređivanje atributa vektorskog sloja. *Attribute Settings* alat također nudi praktičnije sučelje od sučelja QGIS-a. Kućice za ispis svojstava odabranog atributa služit će i za definiranje svojstava novih atributa. Alat bi stoga imao standardne funkcije kao što su dodavanje i brisanje atributa te spremanje promjena.

3. KORIŠTENE TEHNOLOGIJE

3.1. Python

Guido van Rossum kasnih 80-ih godina prošlog stoljeća započinje razvoj jezika koji je imao mogućnost prepoznavanja i rješavanja grešaka nastalih prilikom izvođenja koda. Ovaj programski jezik nazvan je Python po britanskoj televizijskoj seriji *Monty Python's Flying Circus*. Prvi se puta počinje koristiti 1989. godine na CWI-u[2] u Nizozemskoj, a nastao je kao zamjena za do tada korišteni programski jezik ABC.

U listopadu 2000. godine izlazi Python 2.0 inačica jezika. U to vrijeme Python postaje programski jezik čijem razvoju doprinosi zajednica korisnika, za razliku od nekih drugih jezika koji su razvijani od strane profesionalnih razvojnih programera. Uz ovaj važan iskorak, Python 2.0 je također predstavio pregršt novih mogućnosti kao što je automatska memorijska alokacija podataka i podrška za *Unicode*[3] znakove.

Posljednja generacija Pythona, nazvana Python 3.0, izdana je 2008. godine nakon duljeg perioda testiranja. Isprva, inačica nije bila reverzibilno kompatibilna s 2.x inačicom koda, međutim, do danas su mnoge važnije mogućnosti prilagođene za inačice 2.6 i 2.7.

Glavne odlike ovog programskog jezika su čitljivost koda, iskoristivost, jednostavnost sintakse, opsežnost osnovne biblioteke i visoki potencijal proširivanja. U prilog tome, važno je napomenuti kako danas ne samo da postoji ogroman izvor dodataka za Python koji ga dodatno obogaćuju, nego je razvijen niz platformi

---

[2] *Centrum Wiskunde & Informatica*, nizozemski državni institut za matematiku i računalne tehnologije.

[3] Standard za razmjenu podataka usmjeren na prikaz slova na način neovisan o jeziku, računalnom programu ili računalnoj platformi. (http://hr.wikipedia.org/wiki/Unicode)

zasnovanih na modulima ovog programskog jezika. Jedna od tih platformi (PyQGIS) je tema ovog rada. Osim PyQGIS-a spomenuo bih još Pygame, platformu dizajniranu za razvoj računalnih igara. Sastoji se od biblioteka računalne grafike i zvuka, prilagođenih za primjenu pomoću Python programskog jezika.

Važno je napomenuti da je Python interpreterski jezik (eng. *interpreted language*). Da bi shvatili kako interpretatorski jezik funkcionira moramo se upoznati s terminom jezik prevodilac (eng. *compiled language*).

Prevoditeljski jezici (C, Delphi, Java, Visual Basic...) su svi jezici koji imaju sposobnost vlastiti kod prevesti u kod koji procesor razumije i kojeg je u stanju izvršiti (tzv. računalni kod). Interpreterski jezik nema tu sposobnost što znači da mora koristi upravo prevoditeljski jezik u tu svrhu. Dakle, interpreterski jezik samo tumači naredbe prevoditeljskom jeziku koji potom te naredbe prevodi u računalni kod. Krajnja posljedica ovoga je da su svi interpretatorski jezici (BASIC, JavaScript, Python, Ruby...) pisani u nekom od prevoditeljskih jezika. Python je tako napisan u C-u.

Python je napredan, potpuno dinamičan programski jezik koji podržava višestruke metode programiranja. Tako uz objektno-orijentirano programiranje (eng. *Object-oriented programming*) koje se smatra najizvornije ovom jeziku, potpuno podržava još proceduralno (strukturirano) programiranje, a djelomično se može iskoristiti i za funkcionalno programiranje te aspektno programiranje (eng. *Aspect-oriented programming*). Python također ima svojstvo automatskog brisanja nekorištenih zapisa (eng. *garbage collector*) iz memorije što ima za posljedicu smanjivanje potencijalnih grešaka tijekom izvođenja programa.

Jezik prepoznaje riječi koje smo definirali kao imena klasa, varijabli, metoda itd. i to odmah po upisu. Stoga nije potrebno prvo prevesti kod (eng. *compile*) nego program odmah tijekom pisanja koda raspoznaje da li je korisnik definirao vlastitu funkciju, klasu ili varijablu, i ako jest, koja su njezina svojstva i naziv. Ako se dotična funkcija, klasa ili varijabla pozove u nastavku pisanja koda, program automatski

prikaže podsjetnik o sposobnostima iste. Ovo je vrlo korisno, pogotovo u opsežnom kodu gdje programer ne može pamtiti imena svih naredbi napamet.

Autori Pythona su se vodili idejom da se pri rješavanju problema koncentracija posveti samom problemu, a ne finesama programskog koda (Vasić i Marković, 2004.). To se najbolje vidi u sljedećim razlikama u sintaksi u odnosu na ostale programske jezike:

- tipovi podataka imaju visoki nivo apstrakcije[4], što omogućava zapisivanje složenih operacija koristeći samo jednu naredbu
- blok naredbi se definira uvlačenjem koda umjesto posebnim znakovima (`BEGIN...END` ili `{...}`)
- nije potrebno eksplicitno deklarirati tipove podataka za varijable i argumente, već ih jezik automatski deklarira prema upisanoj vrijednosti

Zbog toga, kod napisan u Pythonu može biti i do pet puta kraći nego isti kod napisan u Javi ili C-u[5]. U nastavku je radi usporedbe prikazan program koji ispisuje kvadrate prvih 10 brojeva, napisan u C++ i Pythonu.

[4] U računalnoj znanosti, apstrakcija je proces definiranja podataka i programa kao reprezentacija vlastitih semantičkih značenja pri čemu nivo apstrakcije određuje u kojoj su mjeri detalji implementacije sakriveni (http://en.wikipedia.org/wiki/Abstraction_(computer_science)).
[5] http://www.python.org/doc/essays/comparisons.html

C++ kod:

```
#include <iostream>
using namespace std;

void main()
{
  int i, s;
  for(i= 1; i < 11; i++)
  {
    s = i*i;
    cout << s << endl;
  }
}
```

Python kod:

```
for i in range (1, 11):
      s = i*i
      print s
```

Postoji mišljenje da je Python jedan od najlakših programskih jezika za korištenje (lakši i od QBasica). Samim tim i vrijeme potrebno za svladavanje osnova Pythona je dosta kratko (Vasić i Marković, 2004.).

Kao i ostali interpreterski jezici, Python je nešto sporiji u izvršavanju uspoređujući ga s prevoditeljskim jezicima. Međutim, postoji niz alata kao što su Cython ili Jython koji automatski transformiraju Python kod u C kod, odnosno u Java kod. Kada je kod jednom transformiran, prilikom izvršavanja više ne prolazi kroz proces interpretacije pa stoga ne gubi na brzini.

Python je odličan izbor za pisanje skripti, ali se također upotrebljava u širokoj lepezi samostalnih programa i to u komercijalne svrhe. Python interpreteri su dostupni za mnoge operacijske sisteme, uključujući Windows, Mac OS X i GNU/Linux.

U nastavku su po tablicama grupirani znakovi i termini koji su rezervirani i prepoznati od Pythona. Ovi su prikazi namijenjeni za upoznavanje gramatike Pythona, a mogu poslužiti i kao brzi podsjetnik. Svaki znak ili termin istaknut je

plavom bojom i nadopunjen primjerom kako se pravilno upotrebljava u kodu. Narančastom bojom su istaknuti svi ostali rezervirani termini.

U Tablica 1. imamo zbir razdjelnika. Razdjelnik je niz od jednog ili više znakova koji se koristi za definiranje granica između nezavisnih cjelina u sintaksi koda. Tipični razdjelnici su zarez, točka, dvotočka, zagrade, a tu se još nalaze znakovi poput ' += ' koji se ponašaju kao operatori (npr. `x+=y` znači isto što i `x=x+y`).

Tablica 2. sadrži znakove korištene za matematičke operacije. Njihova zadaća je iz poznatih argumenata procesuirati nove podatke. Znakovi '+' i '−' također imaju i unarnu zadaću što znači da mogu izvršiti operaciju nad jednim argument umjesto dva (npr. mijenjanje predznaka).

U Tablica 3. su izlistani relacijski operateri. Njihova zadaća je provjeravati odnos između argumenata, a vraćaju jedino boolean vrijednosti (`True` ili `False`).

U Tablica 4. navedeni su najčešći tipovi podataka koji se mogu upotrijebiti u ovom programskom jeziku. Alati za baratanje različitim tipovima podataka su neophodan segment svakog modernog računalnog jezika. Tako i Python ima prostran asortiman korisnih vrsta podataka koji su jasni i jednostavni za korištenje. Valja napomenuti da u Pythonu nije potrebno prethodno definirati tip podatka za neku varijablu, već joj je dovoljno dodijeliti vrijednost da Python prema toj vrijednosti zaključi o kojem se tipu podatka radi.

Tablica 1. Razdjelnici u Pythonu

RAZDJELNICI			
(	)	[	]
{	}	,	:
.	`	=	;
+=	-=	*=	/=
//=	%=	<=	|=
^=	>>=	<<=	**=
'	"	\	@

Tablica 2. Operatori u Pythonu

SIMBOL	ULOGA	PRIMJER
`=`	Dodavanje vrijednosti varijabli.	`var1 = 'Python'`
`+`	Zbrajanje.	`x = x + b`
`-`	Oduzimanje i mijenjanje predznaka.	`x = x - 1` `y = -c`
`*`	Množenje.	`x = x * 2`
`**`	Potenciranje.	`x = 2**2`
`/`	Dijeljenje. Ako su djeljenik i djelitelj cijeli brojevi, količnik će iznositi cjelobrojnu razliku. Da bi količnik bio realan broj, barem jedan od brojeva mora biti realnog tipa.	`x = 7 / 3` ----> x je 2 `x = 7 / 3.0` ----> x je 2.33
`//`	Vraća isključivo razliku dijeljenja.	`x = 7 // 3.0` ----> x je 2.0
`%`	Vraća ostatak dijeljenja.	`x = 7 % 3` ----> x je 1

Tablica 3. Relacijski operatori u Pythonu

SIMBOL	ZNAČENJE	PRIMJER
`>`	Strogo veće	`if x > y: ...`
`>=`	Veće ili jednako	`if z >= 3: ...`
`<`	Strogo manje	`if 14 < 'a': ...`
`<=`	Manje ili jednako	`while a <= 7: ...`
`==`	Jednako	`if lista[i] == len(rijec): ...`
`!=` ili `<>`	Nije jednako	`while randrange(0,10) != x: ...`

Tablica 4. Vrste podataka u Pythonu

TIP	OPIS	PRIMJER
`int` (integer)	Nepromjenjiv cijeli broj	`1337`
`float`	Nepromjenjiv realni broj	`3.14`

`str` (string)	Nepromjenjiv niz Unicode znakova	`'Monty'`
		`"Python"`
		`"""Viselinijski tekst"""`

`bool` (boolean)	Nepromjenjiva vrijednost za točno i netočno	`True` `False`
`list`	Promjenjiv skup različitih tipova podataka	`[7.1, 'rijec', True]`
`tuple`	Nepromjenjiv skup različitih tipova podataka	`(7.1, 'rijec', True)`
`dict` (dictionary)	Promjenjiv skup parova (ključ: vrijednost)	`{'kljuc1':5, 1.1:False, 'a':'vrijednost2'}`
`complex`	Nepromjenjiv kompleksni broj	`5+1.2j`
`bytes`	Nepromjenjiv niz byteova	`b'Niz znakova'`
`bytearray`	Promjenjiv niz byteova	`bytearray(b'Niz znakova')`
`set`	Niz vrijednosti zapisanih bez poretka i bez ponavljanja	`set([7.1, 'rijec', True])`

Posljednje dvije tablice sadrže najčešće ključne riječi (eng. *expression*). Svaka od ovih ključnih riječi ima funkciju koja je ukratko opisana u tablici. Kod imenovanja varijabli potrebno je izbjegavati te termine jer su rezervirani. U protivnome se javlja greška kod izvršavanja.

Tablica 5. Izjave u Pythonu

IZJAVA	OPIS	PRIMJER

`print`	Ispisuje tekst.
	Postoji više vrsta formatiranja teksta ukoliko je ispis dinamičan kao u primjeru.
	`%` označuje mjesto upisa varijable a oznaka nakon toga tip konverzije (`s` se ponaša isto ko i `str()`).
	Isto tako {} označava mjesto upisa varijable, a upisani indeks odgovara poretku varijabli u zagradi funkcije `format()`
		`print 'Ispisan je broj ' + str(broj)`
		
		`print 'Ispisan je broj %s' % broj`
		
		`print 'Ispisan je broj {0}'.format(broj)`

`if`	Izjava koja uvjetno izvršava blokove koda.	`if broj == 0:`
		`  print 'Broj je nula'`
		`elif broj < 0:`
		`  print 'Broj je negativan'`
		`else:`
		`  print 'Broj je pozitivan'`

`in`	Provjerava da li je neki objekt sadržan u nekom drugom objektu.	`lista = [7.1, 'rijec', True]`
		`if 7.1 in lista:`
		`  print '7.1 je dio liste'`

`for`	Iterator koji prolazi kroz objekt koji sadrži više elemenata i vraća svaki element u blok koda.	`for element in lista:` `  print element.index()`

`while`	Petlja koja ponavlja blok koda dok je uvjet zadovoljen.	`while broj > 0:`
		`  print 'Broj je veci od nule'`
		`  broj -= 1`

`try`	Prepoznaje greške koje se javljaju tijekom izvođenja te pokreće blok koda koji korisnik namjeni za rješavanje tih grešaka.	`try:`
		`  kolicnik = 3.0 / i`
		`except ZeroDivisionError:`
		`  print "Varijabla 'i' ne smije imati vrijednost 0"`

`class`	Pretvara blok koda u definiciju klase.	`class kalkulator(self):`
		`  self.gumb1 = Button()`
		`  self.gumb1.setText( '+' )`
		`  .`
		`  .`
		`  .`

`def`	Pretvara blok koda u definiciju funkcije.	`def kub(x):` `  print "Kub upisanog broja je " + str(x*x*x)`

`break`	Poziva se u petlji s namjerom da zaustavi njeno daljnje izvođenje.	`for broj in lista:`
		`  if broj == 5:`
		`  print 'Broj je pronadjen'`
		`  break`

Tablica 6. Izjave u Pythonu (nastavak)

IZJAVA	OPIS	PRIMJER

`return`	Može se koristiti samo unutar funkcije. Vraća neku vrijednost i ako je pozvana u petlji zaustavlja njeno daljnje izvođenje.	`def maksimum (x,y):`
		`  if x > y:`
		`  return x`
		`  else:`
		`  return y`

`continue`	Poziva se u petlji s namjerom da preskoči izvođenje ostatka koda u bloku i prebaci se na sljedeći element	`while broj < 100:`
		`  broj += 1`
		`  if broj % 2 == 0:`
		`  continue`
		`  print broj`

`pass`	Ne radi ništa. Koristi se kod funkcija koje trenutno ne želimo definirati, a ne smijemo ih ostaviti praznima.	`def funkcija():` `  pass`

`lambda`	Kreira funkciju unutar linije koda.
	Koristi se kod jednostavnih funkcija koje se ne koriste u više navrata. Nije potrebno imenovati funkciju ni koristiti `def` proceduru.
		`for i in (1, 2, 3, 4, 5):`
		`  a = lambda x: x*x`
		`  print a(i)`

`import`	Služi za uvoz modula u Python skriptu.	`import math` `import xlrd`
`from`	Isto se koristi kod uvoza modula. Točno specificira koje će se varijable uvesti.	`from math import sqrt` `from qgis import core, gui`

`not`	Izraz Booleove algebre.
	Negacija vrijednosti.
		`lista = [7.1, 'rijec', True]`
		`if 'stvar' not in lista:`
		`  print 'stvar nije pronadjena'`

`and`	Izraz Booleove algebre Provjerava da li su svi uvjeti zadovoljeni.	`if 0<broj<10 and broj % 2 == 0:` `  print 'To je jednoznamenkasti parni broj'`
`or`	Izraz Booleove algebre. Provjerava da li je barem jedan uvjet zadovoljen.	`if 0<broj<10 or broj % 2 == 0:` `  print 'Broj zadovoljava uvjet'`

Pošto Python nije glavna tema ovog rada, u njemu nećemo temeljito razdijeliti strukturu jezika niti ćemo se dublje pozabaviti njegovom filozofijom. Ipak moram napomenuti kako je poznavanje jezika od ključne važnosti ako želimo razumjeti na koji način djeluje QGIS platforma, odnosno proširenje koje ćemo razviti. Na Internetu postoji pregršt besplatnih vodiča, kako za početnike tako i napredne korisnike, uz koje je učenje Pythona lako i brzo.

3.1.1. Alati za programiranje

Python interpreter je slobodan softver otvorenog koda čiji je razvoj veoma intenzivan i može se preuzeti s Interneta (URL-1). Trenutno postoje dvije provjerene inačice za skinuti. To su 2.7.3. i 3.2.3. Razlika između 2. i 3. generacije je da 3. generacija nije unatrag kompatibilna s 2. generacijom jer se radilo na pročišćenju i reimplementaciji jezika. Najveće poboljšanje doživjela je podrška za Unicode znakove, tj. svi stringovi su sad Unicode znakovi, što olakšava manipuliranje njima.

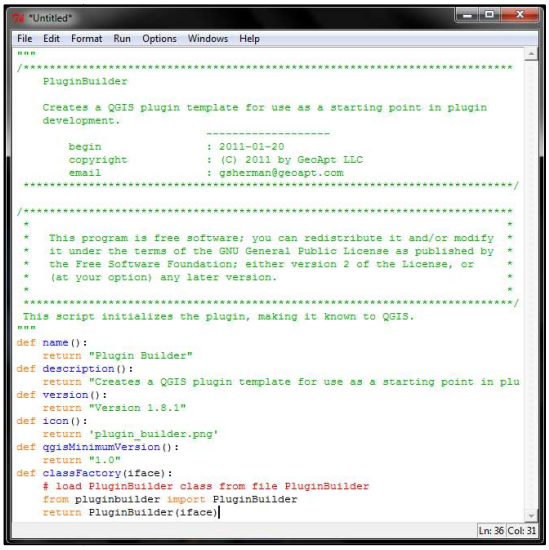
Instalacijski paket Quantum GIS-a sadrži vlastiti Python interpreter (2.7 inačica) koji se instalira s QGIS-om. Razlog tome je što QGIS standardno omogućava uvoz proširenja razvijenih u Pythonu, što znači da mora imati integriran i posebno prilagođen Python interpreter da bi proširenja radila.

Želimo li pisati kod u bilo kojem sučelju, integrirani Python interpreter neće biti dovoljan. Iako integrirani interpreter može izvršiti program, ipak nije u stanju pružiti podršku tijekom pisanja koda. Radi toga je potrebno instalirati samostalnu instalaciju Pythona koja se povezuje s IDE-om.

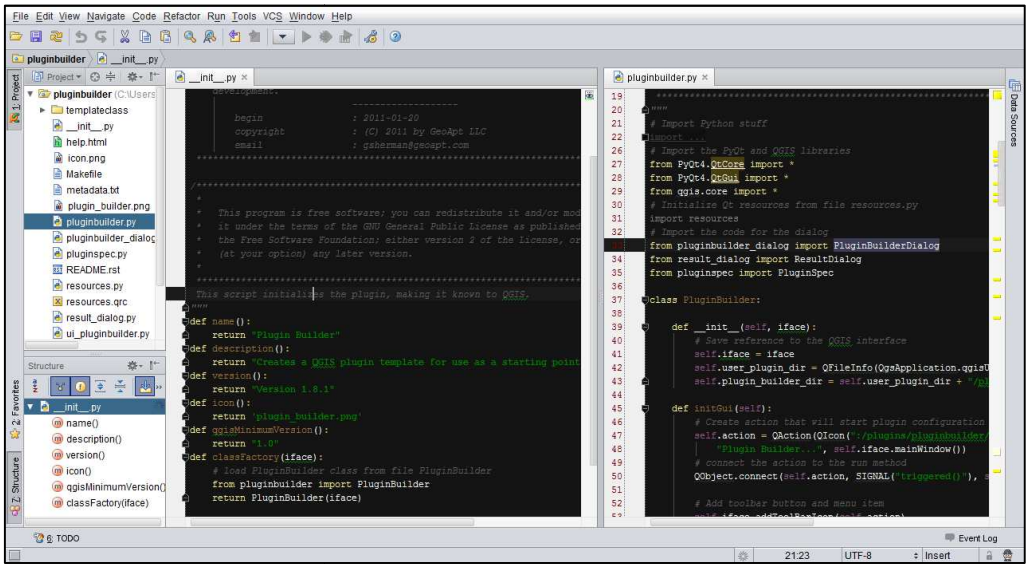
Integrirano razvojno okruženje ili IDE (eng. *Integrated Development Environment*) je aplikacija koja programerima pruža sve potrebne elemente za pisanje i uređivanje koda. Sa samostalnom instalacijom Python-a 2.7.3. dolazi osnovni Pythonov IDE nazvan IDLE.

IDE se obično sastoji od:

- urednika izvornog koda (eng. *editor*)
- alata za automatizaciju izgradnje programa
- programa za pronalaženje pogrešaka (eng. *debugger*)



Slika 1. Sučelje IDE-a koje dolazi s instalacijom Pythona (IDLE)



Slika 2. Sučelje komercijalnog IDE-a (PyCharm)

IDLE nudi sučelje koje će zadovoljiti većinu korisnika, međutim postoje IDE sučelja koja nude naprednije alate za automatizaciju rada i bolju preglednost koda. Neki od njih su besplatni kao što je PyScripter, a neki su komercijalne izvedbe kao što je to PyCharm tvrtke JetBrains.

Usporedbom Slika 1. sa Slika 2. možemo uočiti veliku razliku između standardnog i profesionalnog IDE-a. Osim što profesionalni IDE nudi bolju preglednost i pregršt alata za uređivanje, također provjerava sintaksu koda u realnom vremenu. Sukladno tome ističe greške, nudi automatsko dopunjavanje itd. Efikasan IDE vrijedan je dodatak u svakodnevnom procesu razvoja programa.

3.2. Programske paradigme

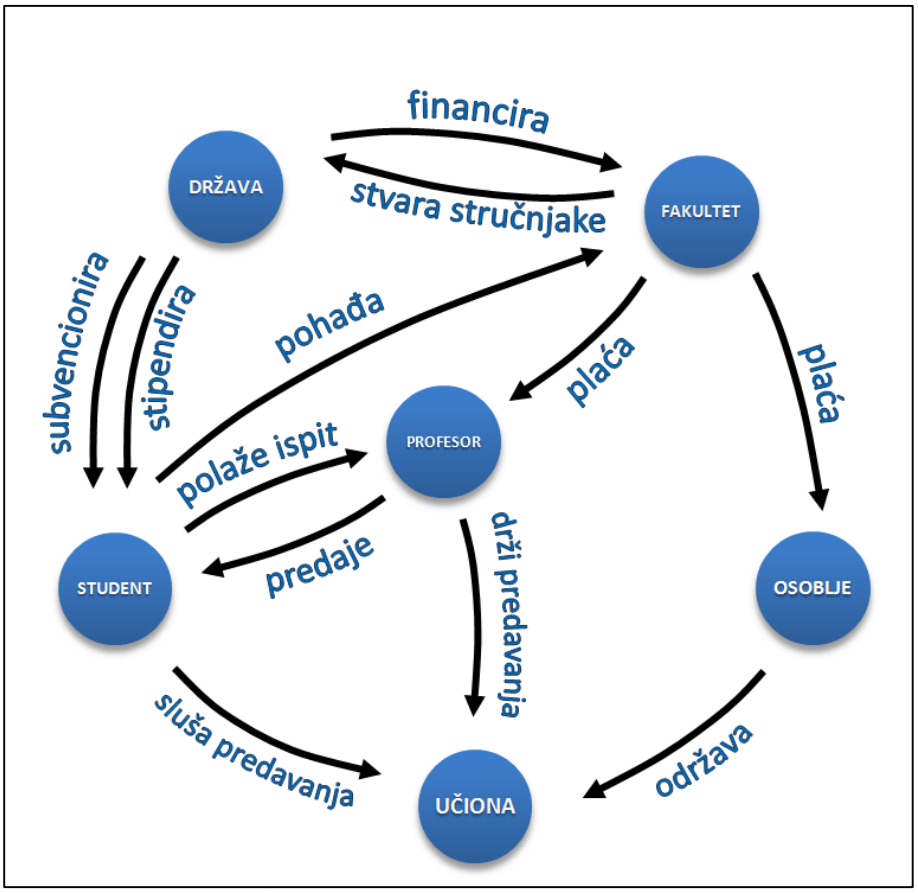
3.2.1. Objektno-orijentirano programiranje

Objektno-orijentirano programiranje je predložak programiranja (paradigma) koji koristi objekte kao glavne elemente koji pokreću mehanizam programa. Sada se nameće pitanje što su zapravo objekti?

Na istome se principu temelji objektno-orijentirano programiranje, jer program nije ništa drugo nego sustav, i to sustav namijenjen rješavanju nekog problema ili zadaće. Važno nam je samo da elementi budu entiteti svojstveni namjeni programa i da imaju veze jedni s drugima. Dakle, prvo se upoznajemo s problemom kojeg treba riješiti, nakon toga modeliramo sustav elemenata i relacija potrebnih za rješavanje i tek onda taj model pretočimo u skup naredbi, odnosno kod.

Upravo te elemente zovemo objektima u svijetu OOP-a (objektno-orijentiranog programiranja). Odmah je potrebno napomenuti da, iako su objekti osnovne komponente OOP-a, njih ne kreiramo direktno u kodu. U kodu definiramo klase, a

objekti nastaju kao realizacije (instance) klasa i to tek tijekom izvršavanja programa. Prema tome, tijekom pisanja koda mi ne manipuliramo s gotovim objektima, već definiramo kako objekt izgleda, koje su mu osobine i funkcije, kada će se stvoriti (instancirati) i kako će se ponašati.



Slika 3. Apstraktni model djelovanja sustava fakulteta

Jednostavno rečeno, klasa je "nacrt" za "izradu" objekta. U praktičnome smislu klasa je niz naredbi koje eksplicitno definiraju kako će se ponašati objekt nastao iz te klase.

Zamislimo bilo koji sustav u prirodi. Da bi lakše shvatili kako funkcionira obično ga nastojimo klasificirati na osnovne elemente i prikazati na apstraktan način. Jedan takav primjer nalazi se na Slika 3. gdje je apstrahiran sustav fakulteta. Klasificirali smo ga s pet osnovnih elemenata (plavi krugovi) i svakom elementu dodali relaciju u odnosu na ostale elemente. Na primjer, profesor predaje kolegij studentu nakon čega student polaže ispit kod profesora. Objekti profesor i student su relacijski povezani te utječu jedan na drugoga.

Struktura objekta sastoji se od atributa (neki ih zovu poljima vrijednosti ili samo poljima) i metoda. Atributi su sve osobine, odnosno informacije koje neki objekt sadrži. Metode su funkcionalnosti koje je objekt u stanju izvršiti. Drugim riječima, pozivanjem metode mijenjamo informacije koje objekt nosi. Ako usporedimo prethodnu sliku apstraktnog sustava s metodama tada bismo metode mogli najbliže opisati kao funkcije koje tvore relacije između objekata.

Dakle, objekt možemo zamisliti kao auto, a klasu kao nacrt tog auta. Polja vrijednosti bi prema tome bile njegove osobine (kao što su boja, broj vrata, marka itd.), a metode najbolje možemo zamisliti kao naredbe koje auto izvršava. Recimo: upali motor; vozi; koči itd.

Ti objekti će se razlikovati samo po vrijednostima koje nose, dok će metode svima biti zajedničke.

Osim metoda definiranih od korisnika postoje i privatne metode. To su već ugrađene metode koje olakšavaju definiranje klasa. Privatne se nazivaju iz razloga jer se ne mogu pozvati izvan klase, u čemu je i sama bit tih metoda. Njih se ne treba pozivati nego se same izvrše kada se objekt koji ih posjeduje nađe u određenoj okolnosti.

Najbolji primjer je `__add__()` metoda koja se aktivira samo kada objekt pokušamo zbrojiti (dodavanje matematičkog operatora '+'). Na primjer, definiramo novu klasu i nazovemo ju Vektor. Ako kreiramo dva objekta iz te klase

```
vektor1 = Vektor(1,2)
vektor2 = Vektor(3,4)
```

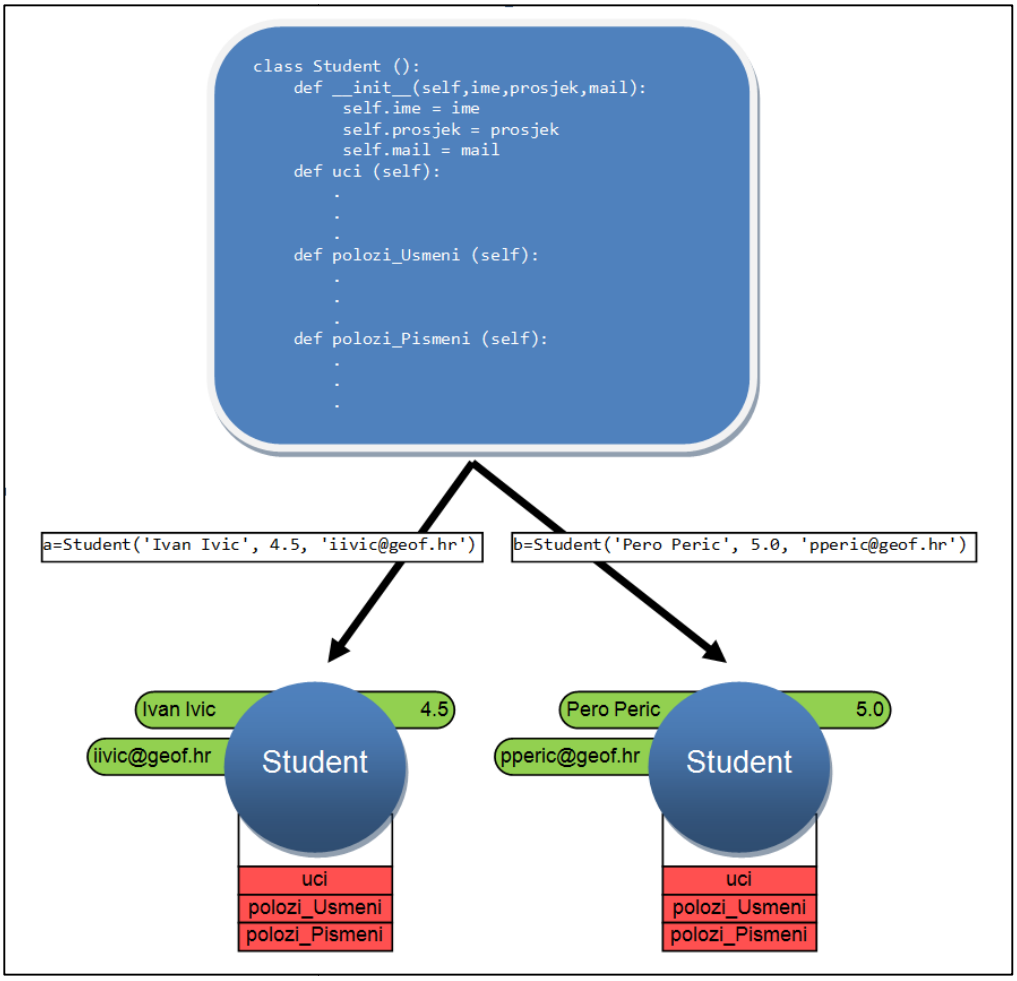
i zbrojimo ih, javlja se greška jer Python nije dobio upute kako zbrajati objekte klase Vektor. Zbog toga u klasu uvrstimo metodu `__add__()` u kojoj definiramo da se rezultat zbrajanja dva objekta iz klase Vektor sastoji od dva broja. Prvi broj dobivamo kao zbroj prvih argumenata jednog i drugog objekta, a drugi broj dobivamo kao zbroj drugih argumenata (`(a,b)+(x,y) = (a+x,b+y)`). Iz toga vidimo da `__add__()` metoda nije ništa drugo nego funkcija koja se izvršava samo ako zbrajamo dva objekta nastala iz neke klase.

Od privatnih metoda najčešće se koristi `__init__()` metoda čija je funkcija automatski se izvršiti čim se klasa instancira. Zbog te sposobnosti često se u njoj deklariraju atributi objekta.

Na Slika 4. može se vidjeti odnos klase i objekta. Uzeli smo klasu "`Student`" iz prethodnog primjera i kreirali dva objekta pomoću nje. Klasa sadrži upute koje atribute jedan student mora imati (ime, prosjek ocjena, e-mail...) i koje sve funkcije mora izvršavati (polaganje ispita, učenje...). Na strelicama je prikazan dio koda čija je zadaća instancirati objekt. U zagradi su argumenti koji se preslikavaju u vrijednosti atributa.

Tako su kreirana dva objekta. Jedan nazvan "`a`" koji predstavlja studenta Ivana Ivića, odnosno objekt nazvan "`b`" koji predstavlja studenta Peru Perića.

Tijekom pisanja koda, atributi i metode nekog objekta pozivaju se na isti način. Prvo se navede ime (oznaka) objekta, a potom slijedi ime atributa, odnosno metode. Jedina razlika je da na kraju imena metode uvijek stoje zagrade zbog mogućeg unosa argumenata. Između imena objekta i imena atributa/metode stoji točka koja odjeljuje dva pojma.



Slika 4. Instanciranja objekata iz klase

Recimo da želimo doznati prosjek studenta Ivića i zatim ga poslati da uči, naredbe bi izgledale ovako:

```
a.prosjek
a.uci()
```

Vrlo jednostavno. Pozvani atribut će uvijek vraćati svoju vrijednost (4.5 u ovom slučaju), a pozvana metoda će naprosto izvršiti svoj blok koda.

Klase definiramo samo jedanput u programu, ali je zato iz svake klase moguće instancirati koliko god objekata želimo. Zato uvodimo opći termin koji implicira na instancirani objekt, tj. omogućuje da se unutar objekta pristupi atributima i metodama. U Pythonu je to termin "`self`". Taj termin inače može biti bilo koja riječ, ali je "`self`" najčešće korišten i smatra se konvencijom među Python programerima.

Termin "`self`" se uvijek piše kao prvi argument bilo koje metode definirane unutar klase. To znači da metoda kao prvi argument prima objekt.

Sada kada je malo jasnije kako se ponašaju objekti možemo bolje razumjeti na koji način djeluje Python. Naime, u Pythonu se svaki njegov dio ponaša kao objekt. Od tipova podataka do izjava, svaki element je definiran kao klasa i kada ga upotrijebimo, zapravo ga instanciramo.

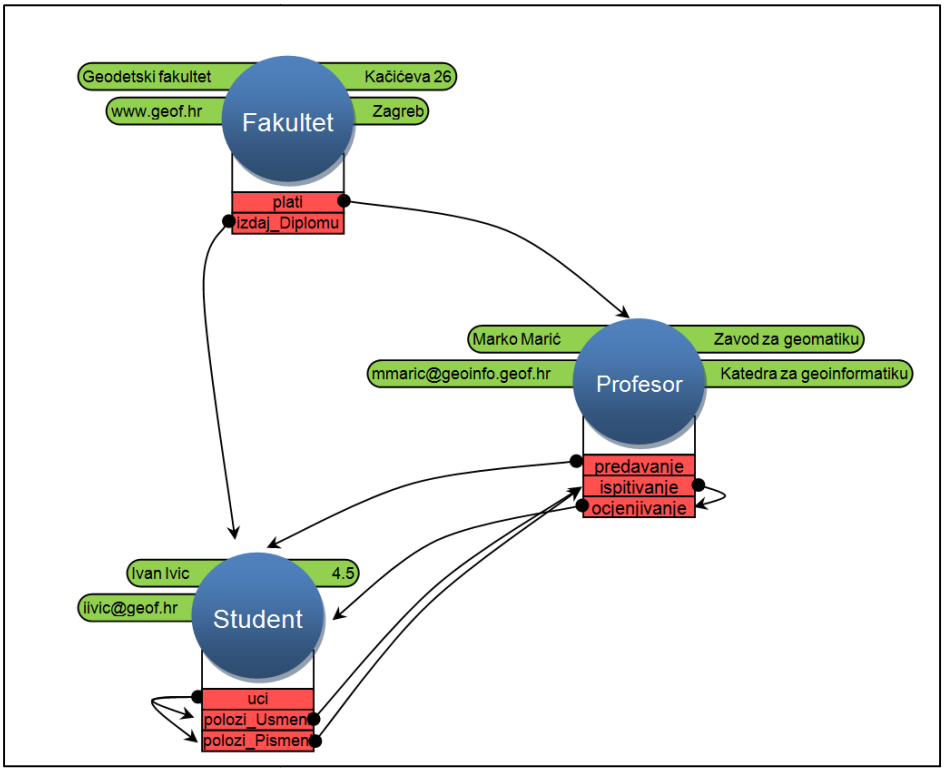
Uzmimo na primjer jednostavnu naredbu kao što je `a = 1` . Varijabla "`a`" je zapravo objekt imena "`a`" realiziran iz klase "`int`" (integer) koja predstavlja vrstu podatka. Jedini atribut mu je vrijednost koju nosi (1 u ovome slučaju), a pomoću metoda je definirano njegovo ponašanje u slučaju zbrajanja, množenja itd. (`__add__()`, `__mul__() ...`).

Zbog svega toga jasno je kako možemo povezati različite objekte na različite načine što nam daje visok stupanj slobode kod modeliranja programa. Međutim ta sloboda ima i negativnu posljedicu koja se često javlja kod kompleksnijih programa, prilikom čega teško pratimo "kretanje" objekata i njihov međusobni utjecaj.

Ako si želimo predočiti ideju objektno-orijentiranog programiranja, najlakše ćemo to učiniti tako da zamislimo skup različitih objekata koji se isprepleteno kreću u prostoru, razmjenjuju informacije i manipuliraju jedni drugima. Radi vizualizacije, uzeli smo prethodni primjer sustava fakulteta, izbacili nepotrebne elemente i na Slika 5. prikazali kao shemu programa. Možemo vidjeti da postoje višestruke kombinacije relacija.

Da sumiramo, objektno-orijentirano programiranje je način programiranja koji koristi objekte zajedno s njihovom interakcijom u svrhu dizajniranja aplikacija. Svojstva objektno-orijentiranog programiranja mogu se u grubo podijeliti na četiri pojma:

- Apstrakcija - postupak pojednostavljivanja složene stvarnosti modeliranjem sustava odgovarajućih klasa
- Višeobličje - mogućnost korištenja operatora ili funkcije na različite načine ovisno o situaciji
- Enkapsulacija - mogućnost skrivanja dijelova klase od ostalih objekata
- Nasljeđivanje - svojstvo koje omogućuje da nove klase nasljeđuju svojstva već definiranih klasa



Slika 5. Odnos objekata unutar programa

3.2.2. Event-based programiranje

Programiranje zasnovano na događajima (eng. *Event-based programming*) je paradigma koja se upotrebljava kod programa s grafičkim sučeljem. Poznato je da programi s grafičkim sučeljem imaju visok stupanj interakcije s korisnikom. Prema tome, program očekuje od korisnika da izvrši neku akciju (npr. pritisak gumba mišem) i ta akcija se prepoznaje kao događaj. Događaj se povezuje s nekom naredbom, tako da kada se događaj ostvari naredba dobiva znak da se izvrši.

Događaji se stvaraju tijekom izvođenja aplikacije, a osim ljudskom interakcijom mogu još biti izazvani funkcijama različitih dijelova programa (npr. brojač, internetska veza, provjera uvjeta...).

Sustav događaja sastoji se od tri elementa:

- Izvor događaja
- Nositelj događaja
- Odredište događaja

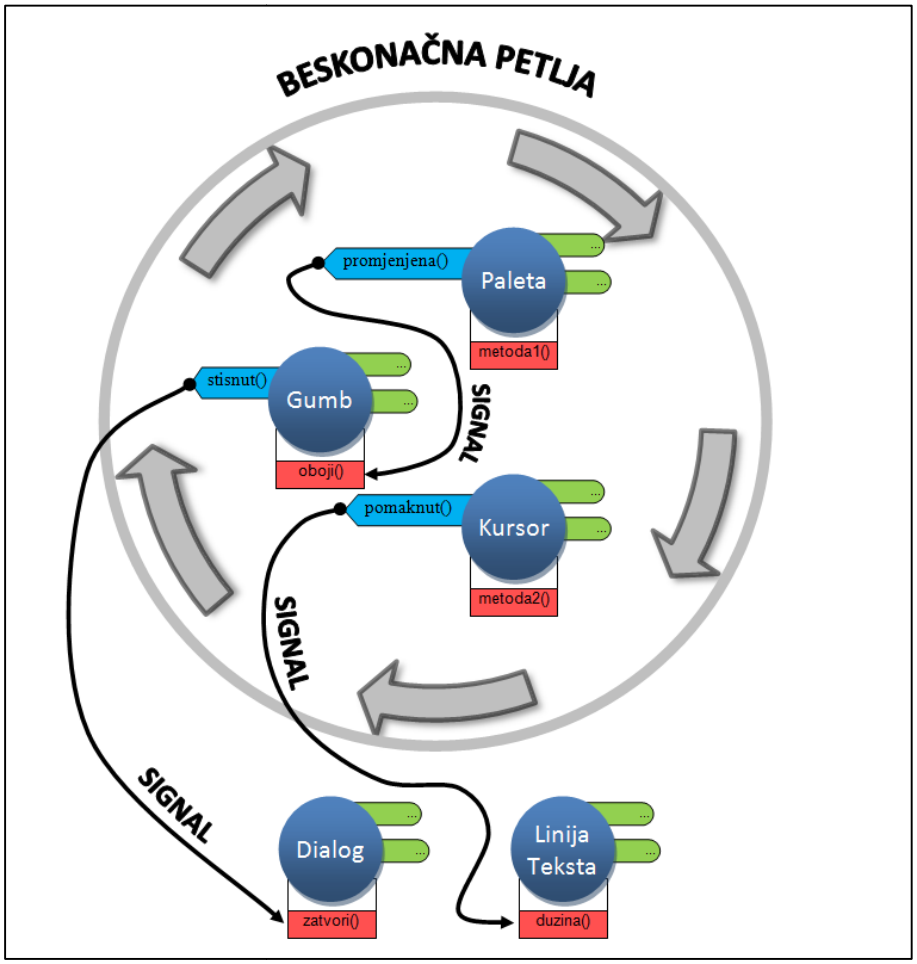
Izvor događaja je objekt čije se stanje promijenilo. On generira događaj. Nositelj događaja je objekt koji preuzima i nosi informacije o nastalim promjenama. Kada kažemo događaj (eng. *event*) zapravo se misli na nositelja. Svrha događaja ili odredište događaja je objekt koji želi biti obaviješten o događaju.

Cijela paradigma zasniva se na beskonačnoj petlji koja se cijelo vrijeme izvodi u pozadini programa, a zadužena je za tri stvari. Prva je detekcija promjene stanja nad objektima, druga je prepoznavanje o kakvom se događaju radi i na kraju slanje događaja prema objektima.

U PyQt biblioteci sustav događaja je predstavljen pomoću signala i priključaka (eng. *slot*). Iz objekta pošiljatelja emitira se signal kada se događaj pojavi. Signal putuje do objekta primatelja te aktivira njegov priključak. Priključak je zapravo bilo koja

pozivna funkcija (metoda, atribut...), a nazvan je priključak da bi mu se istaknula namjena u sustavu događaja.

Veliki broj PyQt klasa već ima definirane signale i priključke, samo ih je potrebno spojiti. Primjerice klasa koja predstavlja gumb (`QPushButton`) standardno ima definiran signal koji se aktivira prilikom pritiska gumba (`clicked()`) te priključak koji, ako se aktivira, prikazuje padajući izbornik gumba (`showMenu()`).



Slika 6. Sustav događaja

Ostale signale koji nisu prethodno definirani, možemo sami definirati. Na primjer, možemo vrlo lako definirati nekom objektu da pošalje signal ako mu određeni atribut poprimi vrijednost "5".

Gledajući kod nekih programa koji koriste PyQt biblioteku za izradu grafičkog sučelja, može se naići na dvije različite vrste u sintaksi definiranja signala. Obje su točne, samo jedna je starija verzija, a jedna novija. Novija verzija više odgovara pythonovskom izražavanju.

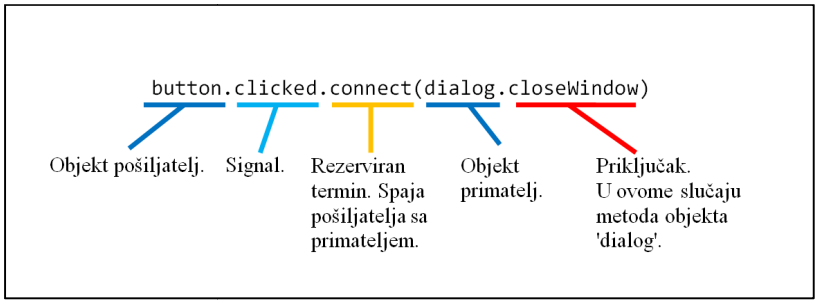
Primjer:

```
# stari način:
QtCore.QObject.connect(button, QtCore.SIGNAL('clicked()'),
dialog.closeWindow)

# novi način:
button.clicked.connect(dialog.closeWindow)
```

Možete primijetiti kako se ime metode kada se nalaze na mjestu priključka pišu bez zagrada.

Na istome primjeru možemo odmah raščlaniti naredbu na elemente. Sintaksa je toliko jednostavna da čitajući je praktički možemo složiti rečenicu koja bi glasila ovako: "Pritiskanje gumba poveži s metodom "closeWindow" objekta "dialog"."



Slika 7. Elementi naredbe za slanje signala i njihova zadaća

Ova metoda programiranja je ključna za bilo kakav program složenijeg tipa, odnosno program koji ne izvršava naredbe po redu kojim su napisane. Razumjeti programiranje temeljeno na događajima znači biti sposoban obaviti veliki broj zadataka koje pred nas stavlja izrada softvera, a funkcionalno grafičko sučelje je samo jedan od njih.

3.2.3. GUI programiranje

Grafičko korisničko sučelje ili GUI (eng. *graphical user interface*) je vrsta sučelja koja omogućuje korisnicima da interakciju s elektroničkim uređajima vrše koristeći grafičke elemente (slike, ikone...) umjesto komandne linije.

Danas grafičko sučelje nije samo svojstveno računalima, već je sastavni dio raznih uređaja kao što su dlanovnici, media reproduktori, uredska pomagala, konzole pa čak i kućanski aparati. Kao što vidimo, okruženi smo grafičkim sučeljima, iz jednostavnog razloga što su od velike pomoći prilikom rukovanja uređajima te pojednostavljuju rad kako običnim korisnicima tako i onim naprednijim. Prema tome sasvim je jasno kako će grafičko sučelje igrati veliku ulogu prilikom izrade naše aplikacije.

Poznavanje programskog jezika, programiranja metodom događaja i biblioteke za izradu sučelja daje nam sve potrebne elemente za izradu grafičkog sučelja. Međutim, sama izrada je sve samo ne jednostavna.

Danas postoje različite pomoćne aplikacije za "crtanje" grafičkog sučelja, kao što je Qt Designer, koje nam omogućuju jednostavnu i brzu izradu grafičkih elemenata sučelja. No izrada grafičkog sučelja ne staje ovdje, već je to samo mali dio cjelokupnog procesa.

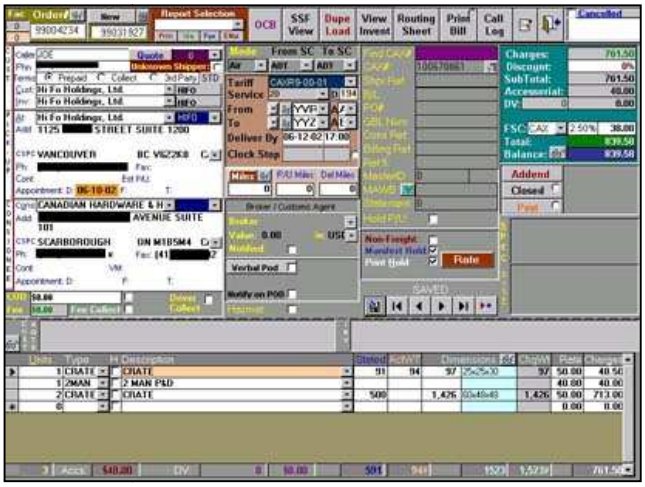
Najveći dio posla otpada na uspostavljanje funkcionalnosti sučelja. Ovaj korak zahtjeva veliko posvećivanje pozornosti detaljima. Recimo da imamo neku listu s imenima osoba. Ako izbrišemo jedno ime i spremimo promjene, možda bi bilo korisno da program automatski označi sljedeće ime u listi. Sve ovisi o namjeni aplikacije.

Gotovo grafičko sučelje ne garantira da se radi o dobrom sučelju. Loše grafičko sučelje poništava svoju svrhu, jer može otežati razumijevanje aplikacije. Idealno grafičko sučelje je jednostavno, efikasno i brzo. Veliki porast u jednostavnosti može pridodati dizajn. Zbog toga se razvojni tim često sastoji i od skupine dizajnera koji ne mare za implementaciju, što samo otežava posao programerima.

Još jedan važan segment tijekom dizajniranja sučelja jest anuliranje mogućnosti da korisnik nehotice izazove zastoj programa. Iz tog je razloga potrebno osigurati da elementi koji međusobno nisu kompatibilni nikad ne dođu u doticaj.

Primjer možemo naći i u proširenju ovog rada. Imamo polje koje prima samo brojeve. Ako upišemo string znakove i pohranimo promjene, program ne javlja nikakvu grešku, ali prebriše vrijednosti u nulu. Ovakvo rješenje je vrlo loše jer može izazvati velike štete na račun produktivnosti. Zbog toga se aplikaciju često daje na testiranje neutralnim subjektima prije izdavanja, budući da tijekom izrade nije moguće predvidjeti sve greške koje se mogu javiti.

Kao što vidimo, izrada grafičkog sučelja zahtjeva pomno planiranje prije same implementacije. Osim funkcionalnosti ono mora biti sposobno komunicirati s korisnikom, a postizanje tog cilja ovisi o više faktora, a ne samo o znanju programiranja.



Slika 8. Primjer lošeg grafičkog sučelja

Slika 9. Primjeri dobro dizajniranih grafičkih sučelja

U praktičnome smislu programiranje grafičkog sučelja nije ništa drugačije od programiranja ostalih dijelova aplikacije. Bilo da kreiramo dijalog ili obični gumb, kao i svi elementi oni imaju svojstva objekta. Dakle, imaju svoje osobine i svoje funkcije.

Međutim, na sve te objekte grafičkog sučelja gledamo kao na poseban dio koda koji je zadužen za prikaz. Znači ako u programu imamo definiranu metodu koja računa korijen iz zadanog broja, ona ima svoju funkciju koja se izvodi i daje rezultat, ali mi

to izvođenje ne vidimo jer nije grafički prikazano. Kada se objekt grafičkog sučelja izvodi, on je uvijek prikazan, što nam govori da svi objekti grafičkog sučelja moraju imati definiran izgled i položaj u virtualnome 2D prostoru.

Kada bi sustav grafičkih elemenata išli razvijati od samoga početka, naišli bi na ogroman posao. Možemo samo zamisliti koje bi sve značajke morali dodati običnome gumbu da ispuni svoju svrhu. Osim definiranja izgleda (dizajn okvira i ispuna) potrebno je definirati i ponašanje prilikom raznih utjecaja na gumb (rastezanje dijaloga u kojem se gumb nalazi, pritisak mišem na gumb, prelazak mišem preko gumba...).

Na svu sreću takav sustav ne moramo sami izrađivati jer već postoje mnogi gotovi i spremni za uporabu. Jedan takav je PyQt biblioteka koja se može preuzeti s interneta (URL-2) i jednostavno uvesti u Python kod. PyQt je skup od dvadesetak modula nastalih iz okvira Qt biblioteke, a prilagođeni su za korištenje s Python programskim jezikom.

Osim PyQt-a postoji pregršt drugih platforma sposobnih za izradu grafičkog sučelja. Najpoznatiji paketi su PyGtk, PyKDE, TkInter i WxPython. Slobodno možemo reći da PyQt nije najjednostavniji paket, ali ima na jednom mjestu sve potrebno za izradu sučelja, stoga neće biti potrebe za instalacijom dodatnih modula. Također, aplikacija Quantum GIS-a razvijena je koristeći Qt platformu što je velika prednost na strani PyQt-a.

Potrebno je još spomenuti PySide platformu čija je biblioteka identična onoj od PyQt-a. Razlika je jedino u licenciranju. PySide se nalazi pod LGPL licencom (*Lesser General Public License*) koja omogućuje fleksibilniju distribuciju. Ako PyQt biblioteku nekog programa zamijenimo s onom PySidea, program ne mijenja funkcionalnost.

3.3. PyQt

PyQt je skup modula, kompatibilnih s Python programskim jezikom, specijaliziranih za izradu svih vrsta aplikacija. PyQt zapravo omogućava pristup Qt biblioteci pomoću Pythona.

Qt je paket za razvoj aplikacija napisan u C++ programskom jeziku. Dizajniran je od strane Haavarda Norda i Eirika Chambe-Enga koji su osnovali tvrtku za razvoj softvera Quasar Technologies. Tvrtka danas nosi ime Qt Development Frameworks i vlasništvo je Nokie. Od 2008. godine razvoj Qt paketa usredotočen je na upotrebu pri izradi platforme za Nokijine uređaje (*Symbian*). Qt paket je tijekom godina zbog svoje popularnosti, usklađen s platformama Windowsa, Mac OS X-a i Linuxa.

Slika 10. Popis klasa GUI modula (lijevo) Qt biblioteke i detaljan opis pojedine klase (desno)

Instalacijom Quantum GIS-a dobivamo i PyQt4 biblioteku spremnu za implementaciju, no ako želimo i dodatne alate morat ćemo instalirati zasebno izdanje. Sa stranice Riverbank Computinga (URL-2) možemo skinuti PyQt4

instalacijski paket. Tijekom instalacije potrebno je odabrati direktorij u koji smo instalirali samostalni Python paket tako da se platforma poveže za Python interpreterom.

PyQt4 paket sadrži sljedeće biblioteke i alate:

- PyQt
- Qt
- Qt Designer
- Qt Linguist
- Qt Assistant
- pyuic4
- pylupdate4
- pyrcc4
- Irelease
- QScintilla

PyQt ne sadrži sve pomoćne alate koji se nalaze u Qt paketu, ali nastoji u njihovom ostvarenju. Pored toga platforme su im identične što znači da je PyQt kompletno rješenje za izradu aplikacija. PyQt klase u kodu možemo prepoznati po prefiksu "Q" u imenu (npr. *QLabel*).

PyQt biblioteka sadrži sljedeće Python module: QtCore, QtGui, QtHelp, QtNetwork, QtOpenGL, QtDBus, QtDeclarative, QtScript, QtSql, QtSvg, QtTest, QtWebKit, QtXml, QtXmlPatterns, phonon, QtMultimedia, QtAssistant, QtDesigner, QAxContainer, Qt, QtDBus i uic. Ovi moduli sadrže preko 300 klasa i preko 6,000 funkcija i metoda. Osim modula, PyQt sadrži i alate kao što su pyuic4, pyrcc4 i pylupdate4.

Na stranicama već spomenutog Riverbank Computinga nalazi se sva potrebna dokumentacija o Qt klasama (URL-3).

Za naše proširenje potrebni su sljedeći moduli i alati:

- **QtCore** - modul koji sadrži klase za interno upravljanje aplikacijom (postavke aplikacije, petlja za događaje, signal-priključak mehanizam...)
- **QtGui** - modul koji sadrži klase grafičkih elemenata
- **QtDesigner** - modul koji omogućuje PyQt-u iskorištavanje Qt Designer aplikacije
- **pyuic4** - alat za konverziju grafičkog sučelja (.ui) u Python kod (.py)
- **pyrcc4** - alat za uklapanje XML kolekcije resursa u Python modul

3.4. Qt Designer

Iako je Qt Designer alat koji dolazi s instalacijom PyQt paketa, sigurno zaslužuje zasebno poglavlje. Qt Designer je samostalna aplikacija koju možete pokrenuti iz startnog izbornika Windowsa, a nalazi se u datoteci pod imenom "*PyQt GPL*". Aplikacija služi za dizajniranje i izgradnju grafičkog sučelja iz Qt komponenti.

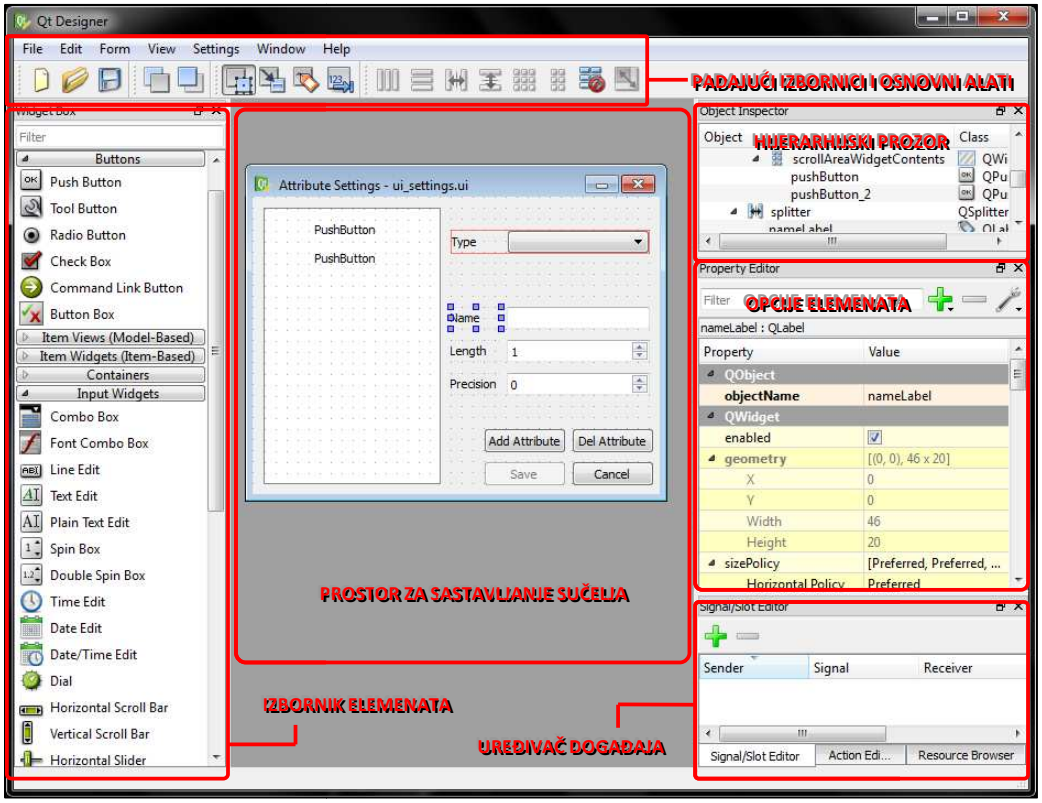
U osnovi, Qt Designer je aplikacija čije grafičko sučelje (Slika 11.) služi za izradu grafičkog sučelja. U njemu možemo komponirati i prilagođavati grafičke elemente u maniri WYSIWYG (eng. *what-you-see-is-what-you-get*). Dakle, ono što vidimo, tako će i na kraju izgledati.

Glavni dio sučelja čine izbornik grafičkih elemenata i prostor za sastavljanje sučelja. Kao što mu i ime govori, izbornik grafičkih elemenata sadrži sve grafičke elemente (eng. *widgets*) koji su definirani u Qt biblioteci. Pomoću miša ih možemo jednostavno povući u prostor za sastavljanje te smjestiti na željeno mjesto unutar prozora.

S desne su strane tri prozora. Gornji prozor prikazuje hijerarhiju grafičkih elemenata. Hijerarhija je vrlo važna jer osigurava funkcionalnost grafičkog sučelja. Dakle, ako želimo da nam se u novome sučelju gumbi nalaze unutar prostora s kliznom trakom,

moramo biti sigurni da je taj prostor nadređeni element svim gumbima. U suprotnome gumbi ne bi bili ograničeni prostorom i pojavili bi se izvan njega.

U sredini se nalazi prozor za uređivanje svojstva grafičkih elemenata. Dovoljno je označiti objekt (unutar hijerarhijskog prozora ili u glavnom prikazu) i u prozoru će se izlistati njegova svojstva i vrijednosti tih svojstva (ime, dimenzije, ispisani tekst...) koje možemo mijenjati.



Slika 11. Sučelje Qt Designera

U podnožju se nalazi treći prozor koji služi kao uređivač signala i priključaka. On ne ovisi o označenom objektu nego prikazuje sve signale koje kreiramo za trenutno sučelje. Dovoljno je dodati novu konekciju i odabrati objekte (pošiljatelj i primatelj),

a program će nam, prema njihovim svojstvima, automatski ponuditi vrstu signala i priključka kojima ih možemo spojiti.

Prema tome, moguće je, koristeći samo Qt Designer, napraviti gotovu aplikaciju, bez pisanja koda. Jedini uvjet je da za pravilan rad aplikacije nisu potrebni drugi objekti osim grafičkih elemenata koji sačinjavaju grafičko sučelje.

Kada smo zadovoljni dizajnom grafičkog sučelja spremamo ga u .ui format. Da bi ga QGIS prepoznao prvo ga je potrebno transformirati u Python kod. Za to nam je potreban alat pyuic4.

Kako bismo pokrenuli pyuic4 moramo koristiti komandnu liniju Windowsa. U *Command Promptu* uđemo u direktorij gdje je spremljena .ui datoteka te nakon toga upisujemo naredbu za transformaciju:

```
pyuic4 -x sucelje.ui -o sucelje.py
```

Izjava `-x` generira dodatni kod koji omogućava pokretanje sučelja kao skripte. Ako je sučelje u sklopu proširenja onda je ova izjava nepotrebna. Nakon toga dajemo ime .ui datoteke koju treba transformirati (u ovom slučaju "sucelje.ui"). Izjava `-o` označava da sučelje treba transformirati u novu datoteku i nakon te izjave potrebno je navesti ime i ekstenziju nove datoteke.

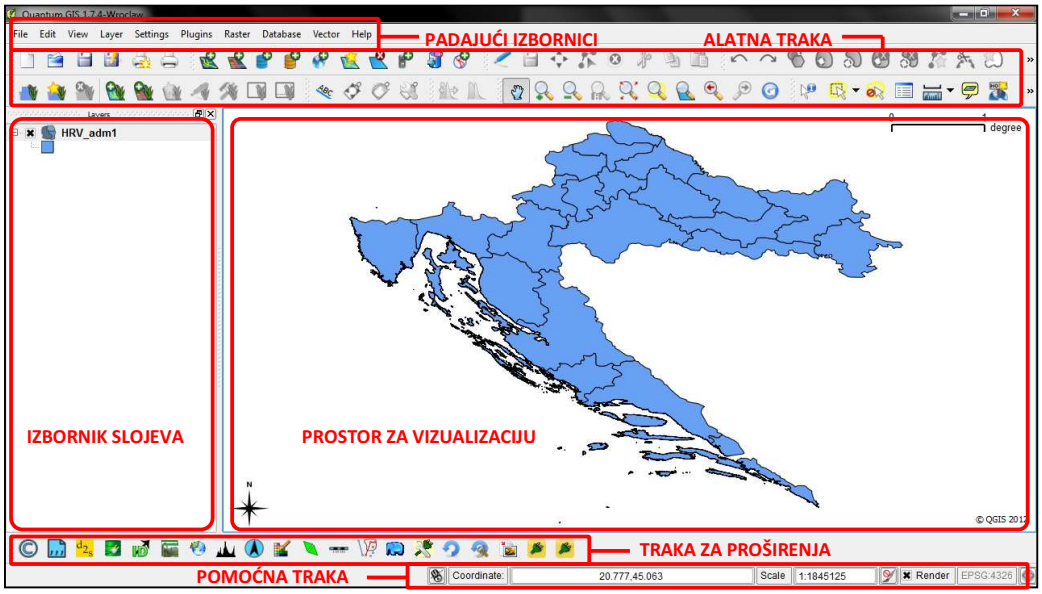
Ova aplikacija ne samo da nam može uštedjeti vrijeme i odmah prikazati naš rad, nego je kud i kamo jednostavnija od pisanja linija koda.

3.5. Quantum GIS

3.5.1. O programu

Quantum GIS ili QGIS je slobodan i jednostavan za korištenje geografski informacijski sustav (GIS), licenciran pod GNU općom javnom licencom (*General Public License*). QGIS je službeni projekt Open Source Geospatial Foundation-a (OSGeo). Aplikacija je kompatibilna s Linux, Unix, Mac OSX, Windows i Android operativnim sustavima i podržava brojne vektorske i rasterske formate te formate baza podataka.

Razvoj QGIS-a započinje Gary Sherman 2002. godine. Tek 2007. godine postaje projekt OSGeo-a, a dvije godine kasnije izlazi inačica 1.0. Trenutno najnovija, a ujedno i inačica korištena u sklopu ovog rada, je QGIS 1.7.4, kodnog imena *Wrocław*.



Slika 12. Sučelje Quantum GIS-a

Quantum GIS nudi kontinuirani rast broja alata u vidu osnovnih funkcija i dodataka. Nudi mogućnost vizualizacije, uređivanja, upravljanja i analize podataka te komponiranja vlastitih karata i njihov ispis.

Glavni razlog zašto je ovaj softverski paket prikladan za temu je taj što je to volonterski projekt i kao takav otvoren za bilo kakav doprinos u vidu poboljšanja od strane korisnika i svih zainteresiranih. Zbog te otvorenosti i dostupnosti vodiča vrlo lako možemo naučiti kako prilagoditi aplikaciju vlastitim potrebama.

Program je napisan u C++-u, a najvećim dijelom koristi klase Qt platforme. *Osim Qt*-a QGIS usko surađuje s bibliotekama poput GEOS i SQLite. GDAL, GRASS GIS, PostGIS i PostgreSQL također se preporučuju jer omogućuju pristup dodatnim formatima podataka.

QGIS aplikacija zauzima relativno malo diskovnog prostora uspoređujući je s onima komercijalnih rješenja, a obrada QGIS datoteka zahtjeva manje radne memorije i procesorskih resursa, radi čega se može koristiti i na starijim računalnim sustavima.

Iako instalacija Quantum GIS-a ima integriranu podršku za Python i PyQt module, ona sama nije dovoljna za razvoj proširenja. Python konzola QGIS-a nema ni osnovne funkcije editiranja teksta te uglavnom služi za brzu komunikaciju s unutarnjim procesima QGIS-a. Zato instalacija vanjskih softverskih paketa nudi dodatne funkcionalnosti koje će biti od velike pomoći prilikom razvoja aplikacije.

3.5.2. QGIS platforma

Aplikacijsko programsko sučelje ili skraćeno API (eng. *Application Programming Interface*) je skup specifikacija koje služe kao uputa za rad softverskim komponentama. Specifikacije opisuju rutine, vrste podataka, klase i varijable nekog programskog sustava.

Quantum GIS je jedan takav sustav s razvijenom platformom koja je slobodna za korištenje. Kako je Python jedan od najboljih jezika za skriptiranje[6], platforma je ubrzo prilagođena jeziku i nastao je PyQGIS. PyQGIS biblioteka oslanja se na PyQt4 jer se cijela QGIS platforma temelji na Qt bibliotekama, a to dozvoljava savršenu integraciju PyQGIS-a s PyQt-om. QGIS klase uglavnom možemo prepoznati po "Qgs" prefiksu u imenu (npr. *QgsField*).

Postoje tri načina na koja možemo koristiti platformu:

- izdavanje naredbi unutar QGIS-a pomoću integrirane Python konzole
- razvoj samostalnih aplikacija
- razvoj proširenja za Quantum GIS

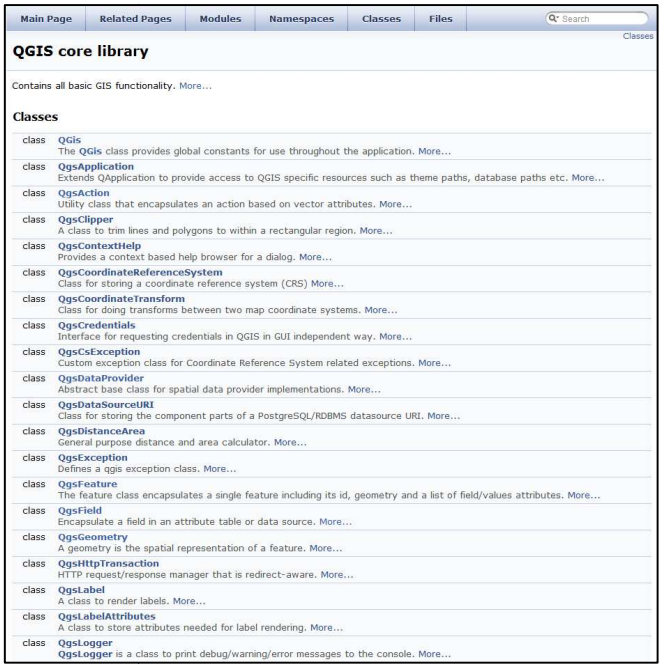
Python konzola svoju primjenu nalazi prilikom upotrebe QGIS-a kada želimo doći do detaljnijih podataka koji nam nisu dostupni uporabom standardnih alata. Upisom Python naredbi možemo u svakom trenutku pristupiti elementima platforme. Pomoću elemenata kontroliramo procese i varijable "iza sučelja" te tako prilagođavamo rad aplikacije vlastitim potrebama. Za neke složenije primjene konzola nije prikladna.

Platforma sadrži komponente za razvoj proširenja koje se mogu iskoristiti i kod razvoja samostalne aplikacije. Naravno, klase su svojstvene GIS aplikaciji, ali to ne znači da se klase u novonastalom proširenju ili aplikaciji ne bi mogle iskoristiti u druge svrhe. Također je važno napomenuti da svaku klasu vrlo lako možemo izmijeniti i prilagoditi. Prema tome mogućnosti nove aplikacije ovise samo o ideji i stupnju dorade.

U tehničkom smislu, izrada proširenja je identična izradi samostalne aplikacije. Jedina razlika je u implementaciji. Proširenje definiramo po pravilima za integraciju s Quantum GIS-om, tako da se može pokretati jedino unutar njega. Prednost

[6] http://www.daniweb.com/community-center/geeks-lounge/threads/53442/best-scripting-language-

proširenja u odnosu na samostalnu aplikaciju je što se vrlo lako može povezati na već gotove alate QGIS-a te time izbjeći mukotrpnu izradu osnovnih elemenata (izbornici, prostor za vizualizaciju, pomoćne trake...).



Slika 13. Dokumentacija klasa QGIS platforme

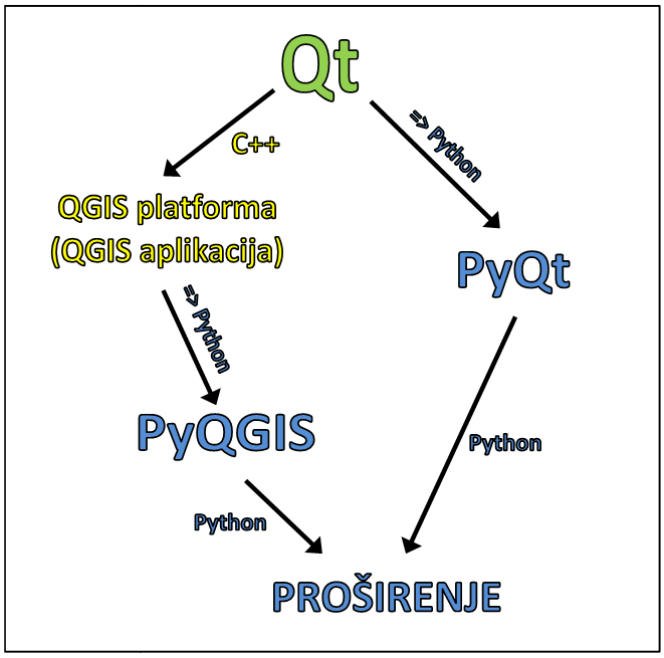
Platforma je podijeljena na pet modula:

- **QGIS core library** - biblioteka osnovnih GIS funkcionalnosti
- **QGIS gui library** - biblioteka grafičkih elemenata
- **QGIS analysis library** - biblioteka naprednih alata za prostornu analizu vektorskih i rasterskih podataka
- **MapComposer** - biblioteka alata potrebnih za izradu karata
  - usko surađuje s bibliotekom grafičkih elemenata
- **QGIS network analysis library** - biblioteka naprednih alata za izradu i analizu topologije

Popis klasa kao i kratka objašnjenja mogu se naći na službenoj stranici QGIS-a (URL-4). Svakoj klasi je moguće pregledati atribute i metode zajedno s vrstom podataka i potrebnim argumentima. Neke klase čak imaju nacrtan dijagram povezanosti s drugim klasama radi boljeg razumijevanja njihovog međudjelovanja.

Upute za PyQGIS koje ujedno sadrže primjere za vježbu mogu se naći u *on-line* vodiču (URL-5). Ovaj vodič se konstantno nadograđuje i trenutno je jedini valjani izvor za učenje PyQGIS-a.

Na Slika 14. Odnos platformi kod izrade QGIS proširenjanalazi se shema koja prikazuje koje su nam sve platforme i biblioteke potrebne za izradu proširenja i u kakvom su međusobnom odnosu. Kao što je već prije spomenuto, osnovnu strukturu QGIS platforme, odnosno QGIS aplikacije, čine klase i rutine iz Qt paketa. Pošto je QGIS platforma napisana u C++ jeziku, za korištenje s Pythonom potrebna nam je njezina Python interpretacija, PyQGIS.



Slika 14. Odnos platformi kod izrade QGIS proširenja

S QGIS platformom dobili smo klase sposobne izvršavati zadatke tipične za jednu GIS aplikaciju. Međutim, QGIS platforma ne posjeduje sve elemente koji bi prethodne klase upakirali u funkcionalnu aplikaciju. Zato nam je potreban Qt paket koji je prilagođen Pythonu. PyQt biblioteka sadrži klase za grafičko i funkcionalno modeliranje aplikacije.

U narednim tablicama navedene su klase i metode PyQGIS biblioteka korištenih u sklopu razvoja našeg proširenja. U desnom stupcu su objašnjene njihove uloge. Možemo vidjeti kako se platforma primjenjuje u svim segmentima proširenja, od spajanja proširenja s QGIS sučeljem do izmjene učitanih vektorskih podataka.

Spajanjem PyQGIS-a i PyQt-a okupili smo sve resurse koji su nam potrebni za olakšan razvoj GIS aplikacije ili, u ovom slučaju, njezinog proširenja.

Tablica 7. Korištene klase iz QGIS platforme

KLASA	OPIS
`QgisInterface`	Definira sučelje izloženo od strane glavne aplikacije.
`QgsMapCanvas`	Grafički prikazuje sve vrste GIS podataka unutar prostora za vizualizaciju.
`QgsMapToolEmitPoint`	Vraća mišem označne koordinate točke unutar područja za vizualizaciju.
`QgsGeometry`	Definira prostorne odlike nekog GIS elementa.
`QgsFeature`	Predstavlja GIS element zajedno s njegovim svojstvima.
`QgsField`	Predstavlja atributno polje zajedno s njegovim svojstvima.
`QgsVectorLayer`	Predstavlja vektorski sloj.
`QgsVectorDataProvider`	Predstavlja pružatelja vektorskih podataka.

Tablica 8. Korištene metode iz QgisInterface klase

METODA	OPIS
`addToolBarIcon()`	Dodaje ikonu na traku za proširenja.
`addPluginToMenu()`	Dodaje prečac proširenja u izbornik proširenja.
`removeToolBarIcon()`	Uklanja ikonu iz trake za proširenja.
`removePluginMenu()`	Uklanja prečac proširenja iz izbornika proširenja.

Tablica 9. Korištene metode iz QgsMapCanvas klase

METODA	OPIS
`setMapTool()`	Aktivira određeni alat nad prostorom za vizualizaciju.
`unsetMapTool()`	Deaktivira trenutno korišteni alat nad prostorom za vizualizaciju.
`currentLayer()`	Vraća trenutno označeni sloj.

Tablica 10. Korištene metode iz QgsGeometry klase

METODA	OPIS
`fromPoint()`	Konstruira geometriju iz točke.
`buffer()`	Stvara tampon zonu iz dane geometrije.
`boundingBox()`	Vraća pravokutnik koji opasuje dani GIS element.

Tablica 11. Korištene metode iz QgsFeature klase

METODA	OPIS
`attributeMap()`	Vraća riječnik s popisom atributa danog sloja.
`geometry()`	Vraća geometriju GIS elementa.

Tablica 12. Korištene metode iz QgsVectorLayer klase

METODA	OPIS
`dataProvider()`	Vraća pružatelja podataka.
`pendingFields()`	Vraća riječnik polja u kojima su zapisana svojstva atributa.
`setSelectedFeatures()`	Označava dane GIS elemente u grafičkom prikazu.
`startEditing()`	Omogućuje uređivanje danog sloja.
`changeAttributeValue()`	Omogućuje izmjenu atributnih vrijednosti.
`commitChanges()`	Pohranjuje izmjene nastale nad slojem.
`rollBack()`	Onemogućuje daljnje uređivanje danog sloja i odbacuje bilo kakve nastale promjene.

Tablica 13. Korištene metode iz QgsVectorDataProvider klase

METODA	OPIS
`attributeIndexes()`	Vraća listu s ispisanim indeksima atributa danog vektorskog sloja.
`select()`	Označava GIS elemente koji dodiruju pravokutnik definiran s `boundingBox()` metodom.
`nextFeature()`	Dohvaća sljedeći GIS element u danom nizu.
`capabilities()`	Vraća mogućnosti vektorskog sloja.

Tablica 14. Korištene metode iz QgsField klase

METODA	OPIS
`name()`	Vraća ime atributnog polja.
`type()`	Vraća vrstu podatka atributnog polja.
`typeName()`	Vraća ime vrste podatka atributnog polja.
`length()`	Vraća dužinu atributnog polja.
`precision()`	Vraća broj decimalnih mjesta atributnog polja.

4. IZRADA PROŠIRENJA

4.1. Uvod

U praktičnome smislu, proširenje za Quantum GIS nije ništa drugo nego skup datoteka sadržanih u mapi koja nosi naziv proširenja. Da bi Quantum GIS prepoznao i koristio proširenje, ono mora odgovarati unaprijed određenim pravilima:

1. Mapa mora biti smještena u direktorij za QGIS proširenja
2. Mapa mora sadržavati točno određene datoteke

Nakon instalacije QGIS automatski stvori mapu za pohranu proširenja. Direktorij te mape u Windowsima je "`C:\Users\ImeKorisnika\.qgis\python\plugins`". Oznaka particije ovisi o tome na kojoj je particiji instaliran operativni sustav (u ovome slučaju C:\). "`ImeKorisnika`" je također varijabilno od sustava do sustava.

Sadržaj mape općenito se sastoji od sljedećih datoteka:

- __init__.py
- plugin.py
- metadata.txt
- resources.qrc
- resources.py
- form.ui
- form.py

Datoteka `__init__.py` je konstruktor modula od kojega sve počinje. Python pomoću nje označava lokalni direktorij kao direktorij koji sadrži Python module. Kada bismo izbrisali `__init__.py` datoteku iz lokalnog direktorija, Python više nebi mogao pretražiti taj direktorij kod uvoza modula. Datoteka nema ni jednu klasu, ali sadrži funkcije koje vraćaju sve potrebne informacije o proširenju (ime, opis, verzija...).

Posljednja funkcija `classFactory()` je najvažnija funkcija. Ona se automatski izvršava kad u QGIS-u uvezemo proširenje, a njezina zadaća je da pozove metodu iz `plugin.py` modula u kojoj je definirana inicijalizacija proširenja (npr. dodavanje i odstranjivanje proširenja iz QGIS sučelja).

Modul `plugin.py` je centralni modul proširenja jer ovdje po prvi put započinjemo s definiranjem funkcionalnosti našeg proširenja. Tako se osim inicijalizacije ovdje mogu naći glavne funkcije koje naše proširenje izvodi. Važno je napomenuti da za razliku od `__init__.py`, modul `plugin.py` ne mora nositi to ime. Štoviše, uobičajeno je da nosi ime proširenja. Osim toga, `plugin.py` može biti razdijeljen u bezbroj modula ako nam je nezgodno sve funkcije, odnosno klase utrpati u jedan modul. Samo je važno da su moduli međusobno povezani.

Tekstualna datoteka `metadata.txt` namijenjena je da autor proširenja u nju zabilježi sve što smatra važnim za bolje razumijevanje rada proširenja. Za sada ova datoteka nije nužna, ali od verzije QGIS-a 1.8 postat će neizostavni dio.

Datoteka `resources.qrc` je XML dokument stvoren od strane Qt Designera, a služi za popis putanja svih iskorištenih resursa tijekom modeliranja grafičkog sučelja (ikone, slike...).

Modul `resources.py` je prijevod prethodne XML datoteke u Python jezik. Da bismo preveli dobivenu `.qrc` datoteku moramo koristiti `pyrcc4` program. Na Windows platformi pokreće se unutar *Command Prompta*. Prvo uđemo u direktorij gdje se `.qrc` datoteka nalazi, a zatim pozivamo prevoditelja sljedećom naredbom:

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

Izjava "`-o`" označava da je izlazni rezultat datoteka, a izraz nakon toga (`resources.py`) uzima kao ime te datoteke. Svaka izmjena `.qrc` datoteke zahtijeva novo prevođenje.

Datoteka `form.ui` sadrži grafičko sučelje izrađeno u Qt Designeru. Ovisno o proširenju, može postojati bezbroj ovakvih datoteka i obično su nazvane proizvoljnim imenom.

Modul `form.py` prijevod je prethodne datoteke u Python kod. Da bismo preveli `.ui` datoteku moramo koristiti pyuic4 program. Unutar Windowsa pokreće se na isti način kao i `pyrcc4` program. U ovome slučaju naredba glasi:

```
pyuic4 form.ui -o form.py
```

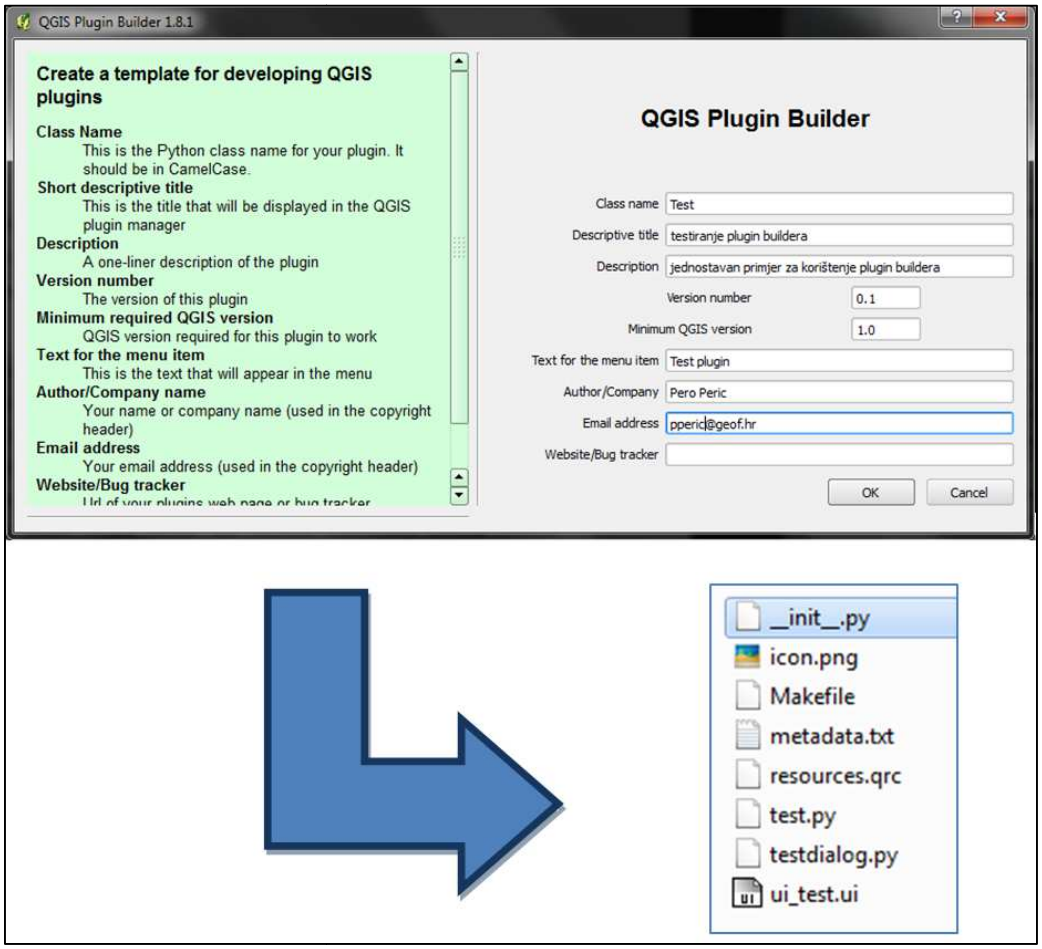
Izjava "`-o`" označava da je izlazni rezultat datoteka, a izraz nakon toga (`form.py`) uzima kao ime te datoteke. Svaka izmjena `.ui` datoteke zahtijeva novo prevođenje.

Važno je napomenuti kako grafički dio aplikacije ne mora biti strogo odijeljen od ostatka aplikacije. Sadržaj `form.py` modula bez problema možemo integrirati u `plugin.py` modul jer je kod i jedno i drugo. Dva su razloga zašto su ovdje moduli odijeljeni. Prvi je razlog taj što grafički modul nije ručno kodiran nego je dobivena zasebna `.ui` datoteka iz Qt Designera. Drugi razlog je bolja preglednost modula, odnosno koda.

Osim navedenih datoteka preporučeno je još ubaciti sve resurse koje proširenje koristi. Tako se prilikom prebacivanja proširenja na drugo računalo prebacuju i resursi što znači da će uvijek biti dostupni. U datoteku ovog proširenja dodane su dvije ikone (*icon1.png* i *icon2.png*), izrađene pomoću *3ds Max* i *Photoshop* softverskih paketa, koje će biti dodijeljene prečacima alata.

4.2. Plugin Builder

Izrada proširenja za QGIS može se pojednostavniti i ubrzati korištenjem *Plugin Buildera*. *Plugin Builder* () je proširenje koje se može skinuti i instalirati unutar QGIS-a odabirom naredbe *Fetch Python Plugins...* koja se nalazi unutar izbornika *Plugins*.



Slika 15. Unos varijabli proširenja u Plugin Builder i struktura kreirane mape proširenja

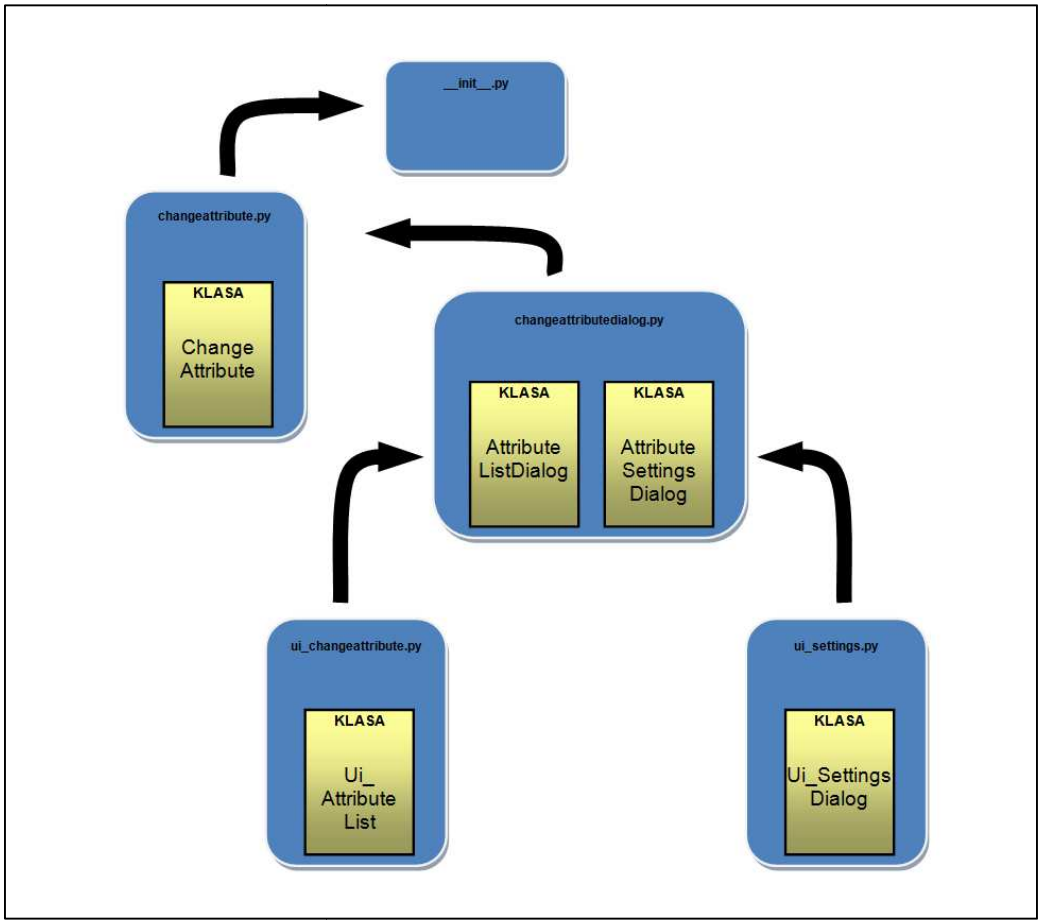
U osnovi, ovo proširenje služi za izgradnju kostura novog proširenja. Kao što vidimo na Slika 15., potrebno je samo zadati općenite podatke o novome proširenju kao što su ime, autor, verzija proširenja, itd., a *Plugin Builder* će u odnosu na te podatke stvoriti mapu s osnovnim datotekama novog proširenja u kojima su definirane generičke funkcije (npr. povezivanje proširenja s QGIS-om, dodavanje prečaca proširenja u izbornike...). Dakle, *Plugin Builder* nas oslobađa elementarnih procesa prilikom izrade proširenja.

Prije pokretanja proširenja unutar QGIS-a, potrebno je još prevesti .ui i .qrc datoteke u Python kod i nakon toga cijelu mapu kopirati u direktorij za QGIS proširenja. Novonastalo proširenje nema nikakvih funkcija. tj. isprogramirano je da prilikom pokretanja kreira prazan dijaloški prozor kojega jedino možemo zatvoriti. Ovim putem isprogramirani su temelji proširenja kojega je sada potrebno nadograditi s vlastitim funkcijama i mogućnostima.

Za izradu proširenja Attribute Changer također je korišten *Plugin Builder*. Na Slika 16. prikazana je struktura proširenja. Gotovo svi moduli su generirani od strane *Plugin Buildera*. Jedini naknadno dodan modul je modul koji sadrži klasu za definiranje grafičkog sučelja Attribute Settings alata (`ui_settings.py`).

4.3. Struktura proširenja - Attribute Changer

Prije nego što krenemo s objašnjavanjem koda, razmotrimo apstraktni model ovog proširenja (Slika 16.). Program je podijeljen u pet modula. Početni modul je *__init__.py* koji veže proširenje s Quantum GIS-om. Funkcija `classFactory()` uvozi *changeattribute.py* modul i instancira klasu `ChangeAttribute`.



Slika 16. Apstraktni prikaz međuodnosa modula i klasa u Attribute Changer proširenju

`ChangeAttribute` klasa sadrži metode za učitavanje i odstranjivanje proširenja iz sučelja. Osim ovih uobičajenih funkcija klasa je još zadužena za izravnu interakciju s atributima odabranog sloja. S obzirom na povratne informacije objekata iz ostalih modula, metode ove klasa mijenjaju atributne vrijednosti vektorskog sloja.

Proširenje je zamišljeno da se sastoji od dva alata. Prvi alat služi za odabir elementa vektorskog sloja i unos atributnih vrijednosti, dok drugi alat služi za podešavanje atributnih polja sloja. Ova klasa definira metode koje prate koji smo alat odabrali i

sukladno tome kreiraju odgovarajući dijaloški prozor instanciranjem klasa iz *changeattributedialog.py* modula.

Modul `changeattributedialog.py` sadrži dvije klase. Obje klase definiraju dijalog. Jedan dijalog odnosi se na GUI prvog alata, a drugi dijalog na GUI drugog alata.

Dijalozima je još jedino potrebno definirati grafički izgled. Zato svaki dijalog uvozi svoj modul (*ui_changeattribute.py*, odnosno *ui_settings.py*) u kojem je definirana klasa s grafičkim sučeljem.

U ovim klasama (`Ui_AttributeList` i `Ui_Settings`) , osim grafičkog izgleda, definirane su i funkcionalnosti koje sučelje mora obavljati. Sučelje mora pamtiti promjene nad atributima koje korisnik zada. Te promjene program prenosi u početnu klasu (`ChangeAttribute`), a metode iz početne klase na temelju toga pohranjuju promjene u atributnu tablicu odabranog sloja.

4.4. Pojašnjenje koda

4.4.1. Modul changeattribute.py

Modul se sastoji od klase `ChangeAttribute` u kojoj su definirane sljedeće metode: `__init__()`, `initGui()`, `unload()`, `checkIfEnabled()`, `selectFeature()` i `attributeSettings()`.

Metoda `__init__()` se automatski pokreće instanciranjem `ChangeAttribute` klase te služi za spajanje proširenja sa sučeljem QGIS-a i kreiranje `QgsMapToolEmitPoint` objekta koji prati interakciju miša s prozorom za vizualizaciju.

```
self.clickTool = QgsMapToolEmitPoint(self.canvas)
```

Metoda `initGui()` dodaje proširenje u izbornik, tj. postavlja dva prečaca za pokretanje Attribute Changer i Attribute Settings alata. Svakom prečacu dodan je signal koji se odašilje nakon odabira alata mišem. Signal prvog alata pokreće metodu `checkIfEnabled()`, dok drugi signal pokreće metodu `attributeSettings()`.

```
self.action.changed.connect(self.checkIfEnabled)
self.action2.triggered.connect(self.attributeSettings)
```

Metoda `unload()` radi upravo suprotno u odnosu na `initGui()` metodu. Ona uklanja prečace iz QGIS sučelja. Sučelje je definirano pomoću objekta `iface` (skraćenica za *interface*).

Metoda `checkIfEnabled()` provjerava da li je prečac Attribute Changer aktiviran. Ako je aktiviran, postavlja kao trenutni alat `QgsMapToolEmitPoint`, a ako nije, onda ga uklanja. Kada je alat aktiviran, metoda prati da li će se izvršiti pritisak miša nad prozorom za vizualizaciju. Ako dođe do pritiska tipke miša, odašilje se signal `canvasClicked` koji poziva metodu `selectFeature()`.

```
if state:
    self.canvas.setMapTool(self.clickTool)
    self.clickTool.canvasClicked.connect(self.selectFeature)
else:
    self.canvas.unsetMapTool(self.clickTool)
    self.clickTool.canvasClicked.disconnect(self.selectFeature)
```

U metodu `selectFeature()` tada se prenose argumenti generirani od strane `QgsMapToolEmitPoint` objekta (koordinate točke koja označava poziciju pokazivača miša). Metoda pretvara točku u geometriju, točnije u kvadratni poligon (eng. *bounding box*).

```
# setup the provider select to filter results based on a rectangle
pntGeom = QgsGeometry.fromPoint(point)
# scale-dependent buffer of 2 pixels-worth of map units
pntBuff = pntGeom.buffer( (self.canvas.mapUnitsPerPixel() * 2),0)
rect = pntBuff.boundingBox()
```

Nakon toga provjerava da li je označen sloj i ako jest prolazi kroz sve elemente tog sloja i provjerava da li se geometrija pojedinog elementa presijeca s geometrijom kvadratnog poligona. Kada pronađe element za kojega vrijedi uvjet, označi ga i pošalje signal za kreiranje dijaloga iz modula *changeattributedialog.py*. Signalom se

prenose podatci o tom elementu (imena i vrijednosti atributa). Kada jednom pokrenemo instancirani dijalog, on čeka povratnu informaciju.

```
result = dlg.exec_()
```

Kao rezultat izvršavanja dijaloga možemo dobiti dvije vreijdnosti. Vrijednost 0 ako je odbačen (`reject()`) ili vrijednost 1 ako je prihvaćen(`accept()`). Ako dijalog bude prihvaćen onda metoda prebriše vrijednosti atributa s vrijednostima dobivenih iz grafičkog sučelja.

```
if result == 1:
    cLayer.startEditing()
    for i in range(len(fields)):
        cLayer.changeAttributeValue(feat.id(), i, dlg.ui.fieldList["valueField" +
                        str(i)].text())
        cLayer.commitChanges()
```

Metoda `attributeSettings()` slična je prethodnoj metodi. Također instancira klasu za kreiranje dijaloga iz modula *changeattributedialog.py*, ali nakon aktiviranja Attribute Settings alata.

```
dlg2 = AttributeSettingsDialog(fields, cLayer, provider, caps)
```

Ova metoda također očekuje povratnu informaciju od dijaloga. Ako je dijalog prihvaćen pohranjuju se promjene nastale nad slojem (`commitChanges()`), a ako nije onda se promjene poništavaju (`rollBack()`).

```
if result2 == 0:
    cLayer.rollBack()
elif result2 == 1:
    cLayer.commitChanges()
```

4.4.2. Modul changeattributedialog.py

Ovaj modul je prilično kratak iako sadrži dvije klase. Obje klase instanciraju dijalog, a pošto su opisane kao podklase `QDialog` klase već je definirano sve što nam je potrebno za dijalog.

Kao što je već bilo spomenuto u prošlom poglavlju, oba alata proširenja pokreću vlastiti dijalog. Instanciranjem klase `AttributeListDialog` pokrećemo dijalog za Attribute Changer, a instanciranjem klase `AttributeSettingsDialog` pokrećemo dijalog za Attribute Settings alat. Jedina dodatna naredba u klasama služi za instanciranje `Ui_AttributeList` klase iz *ui_changeattribute.py* modula, odnosno instanciranje `Ui_SettingsDialog` klase iz *ui_settings.py* modula.

```
self.ui = Ui_AttributeList(self, fields, attrMap)
```

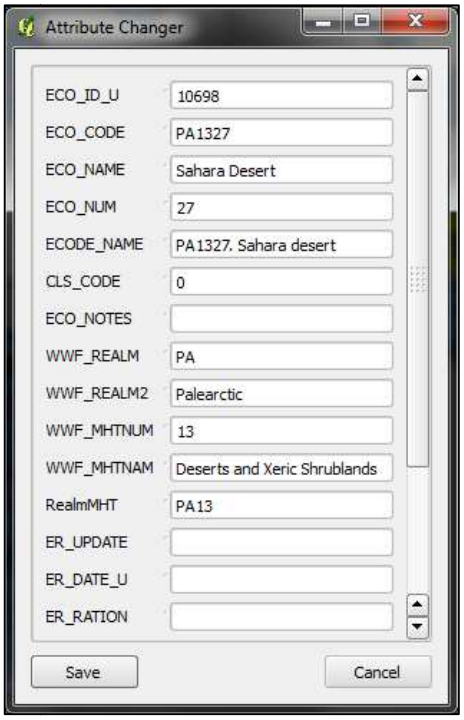
```
self.ui = Ui_SettingsDialog()
```

4.4.3. Modul ui_changeattribute.py

Prvi modul odnosi se na grafičko sučelje alata Attribute Changer. Većina grafičkog sučelja `Ui_SettingsDialog` klase definirana je pomoću Qt Designera. Neke smo dijelove prvotnog koda izmijenili da bi što bolje odgovarao našim potrebama.

Osim definiranog grafičkog sučelja tu su i metode koje osiguravaju međusobnu komunikaciju između grafičkih elemenata. Tako osim `__init__()` metode imamo još `retranslateUi()` i `checkState()` metode.

Metoda `__init__()` služi za iscrtavanje grafičkog sučelja. Preko dijaloga dobiva informacije o označenom geometrijskom elementu (broj atributa i njihove vrijednosti). Finalni izgled grafičkog sučelja prikazan je na Slika 17.



Slika 17. Sučelje alata Attribute Changer

Sučelje se sastoji od gumba za spremanje i odbacivanje izmjena te prostora s pomičnom trakom. U taj prostor smještena su imena atributa (lijevi stupac) zajedno s odgovarajućim vrijednostima (desni stupac).

Pošto broj atributa može varirati od sloja do sloja, grafičko sučelje nije bilo moguće kompletno izraditi u Qt Designeru, već ga je trebalo izmijeniti na način da se broj redaka u prozoru dinamički generira s obzirom na broj atributa sloja. Taj problem je riješen korištenjem `for` iteratora.

```
for (i, value) in attrMap.iteritems():
    self.splitter = QSplitter(self.scrollAreaWidgetContents)
    self.splitter.setOrientation(Qt.Horizontal)
    self.splitter.setObjectName(_fromUtf8("splitter_{0}".format(i)))
    self.attributeNameField = QLabel(self.splitter)
    self.attributeList.append(self.attributeNameField)
    self.attributeList[i].setObjectName(_fromUtf8("attributeNameField_{0}".format(i)))
    self.attributeList[i].setMinimumSize(QSize(labelWidth, 0))
    self.attributeList[i].setMaximumSize(QSize(labelWidth, 16777215))
    self.attributeList[i].setText(_fromUtf8("{0}".format(fields[i].name())))
```

```
    self.fieldList["valueField" + str(i)] = QLineEdit(self.splitter)
    self.fieldList["valueField" +
     str(i)].setObjectName(_fromUtf8("attributeValueField_{0}".format(i)))

    try:
        self.fieldList["valueField" +
        str(i)].setText(_fromUtf8("{0}".format(value.toString())))
    except UnicodeEncodeError:
        codedValue = value.toString().toUtf8()
        self.fieldList["valueField" + str(i)].setText(_fromUtf8("{0}".format(codedValue)))

    maxLength = fields[i].length()
    typeName = fields[i].typeName()
    attType = fields[i].type()
    prec = fields[i].precision()
    if attType == 6:
        self.fieldList["valueField" + str(i)].setToolTip('Name: {0} \nType: {1} \nMax length:
          {2} \nPrecision: {3}'.format(fields[i].name(),typeName,maxLength, prec))
    else:
        self.fieldList["valueField" + str(i)].setToolTip('Name: {0} \nType: {1} \nMax length:
          {2}'.format(fields[i].name(),typeName,maxLength))
        self.fieldList["valueField" + str(i)].cursorPositionChanged.connect(lambda old, new,
          i=i: (self.checkState(i, fields, palette1, palette2, palette3, palette4)))
```

U iteratoru se generiraju PyQt objekti. `QLabel` objekti se koriste za ispis imena atributa, a `QLineEdit` objekti su nam pogodniji za ispis vrijednosti atributa pošto alat mora omogućiti upis i pohranu vlastitih vrijednosti.

Širinu `QLabel` objekata trebalo je također dinamički generirati jer je poželjno da se prilagođava imenu atributa s najviše slova. S takvim pristupom osigurali smo da lijevi stupac nije preširok, a opet da se vidi cijelo ime atributa s najviše znakova. Zato program prolazi kroz sve atribute i traži ime atributa s najviše znakova. Nakon toga najviši broj pomnoži s konstantom 8 i dobiveni rezultat postavlja kao širinu `QLabela`.

```
maxSize=0
for (i,value) in fields.iteritems():
    if len(fields[i].name()) > maxSize:
        maxSize=len(fields[i].name())

labelWidth = maxSize*8

self.attributeList[i].setMinimumSize(QSize(labelWidth, 0))
self.attributeList[i].setMaximumSize(QSize(labelWidth, 16777215))
```

Svaki novi objekt `QLabela` zapisujemo u listu, a `QLineEdita` u riječnik, tako da u bilo koje vrijeme možemo manipulirati njima. Svaki novi objekt dobiva svojstvene osobine kao što su na primjer ime objekta, ispisani tekst, signal ili opis (eng. *tooltip*).

Kada se `QLineEdit` objekt kreira prvo je potrebno provjeriti tekst koji dobiva iz atributne tablice. Naime, može se dogoditi da ne prepozna znakove kao što su hrvatska slova č, ž, š, prilikom čega se javlja greška. Zato program svaki neregularni zapis pretvara u UTF-8 format.

```
try:
    self.fieldList["valueField" + str(i)].setText(_fromUtf8("{0}".format(value.toString())))
except UnicodeEncodeError:
    codedValue = value.toString().toUtf8()
    self.fieldList["valueField" + str(i)].setText(_fromUtf8("{0}".format(codedValue)))
```

Osim polja vezanih za svojstva atributa tu su još gumbi za pohranu izmjena (*Save*) i odbacivanje izmjena (*Cancel*). Pritiskom gumba za pohranu aktivira se signal koji prihvaća dijalog, što daje metodi `selectFeature()`, iz *changeattribute.py* modula, znak da trenutne vrijednosti atributa prebriše s onima koje su upisane u Attribute Changer sučelju. Pritiskom gumba *Cancel* zatvara se dijalog i promjene su odbačene.

```
self.gridLayout.addWidget(self.cancelButton, 1, 2, 1, 1)
QObject.connect(self.cancelButton, SIGNAL(_fromUtf8("clicked()")), Dialog.reject)
```

Svako atributno polje ima strogo definiran tip podataka (`integer`, `real` ili `string`), dužinu zapisa i preciznost. Prilikom upisa vlastitih vrijednosti u Attribute Changer prozor može se dogoditi da zapis ne zadovoljava pravila (npr. upis slova u polje koje je namijenjeno samo za integer brojeve). U tom slučaju program ne javlja grešku, a zapis pretvara u 0.

Da bi izbjegli ovo neugodno iznenađenje, koristimo `checkState()` metodu. Ova metoda se poziva svaki put kada unutar `QLineEdit` objekta pomaknemo kursor.

```
self.fieldList["valueField" + str(i)].cursorPositionChanged.connect(lambda old, new, i=i:
                (self.checkState(i, fields, palette1, palette2, palette3, palette4)))
```

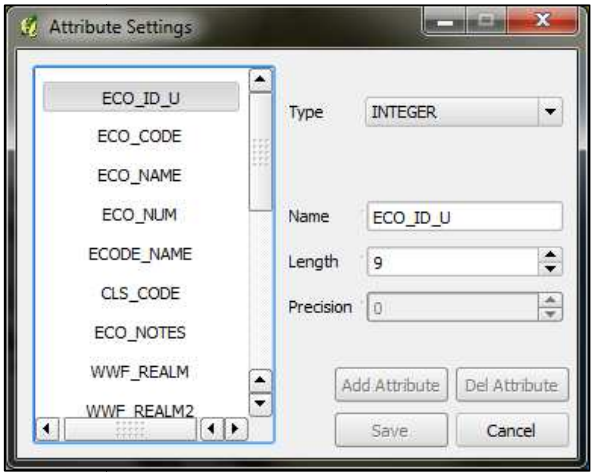
`QLineEdit` objekt u metodu šalje argumente koji nose informacije o objektu, a metoda provjerava dužinu zapisa, vrstu podataka i da li postoji neki znak koji ne pripada toj vrsti podataka. Ukoliko je greška pronađena program zacrveni dotično `QLineEdit` polje i ispiše o kakvoj se grešci radi.

Posljednja metoda `retranslateUi()` je u cijelosti kreirana od strane Qt Designera te služi za ispravni prikaz svih znakova natpisa koji se pojavljuju u prozoru.

4.4.4. Modul ui_settings.py

Posljednji modul *ui_settings.py* sastoji se od klase `Ui_SettingsDialog` koja sadrži metode `setupUi()`, `retranslateUi()`, `attributeSelected()`, `makeField()`, `deleteAttribute()` i `addAttribute()`.

Kao i prethodni modul, ovaj modul sadrži klasu koja definira grafičko sučelje dijaloga, ali ovog puta Attribute Settings alata. Na lijevoj strani sučelja nalazi se popis, odnosno izbornik atributa trenutno označenog sloja, dok desna strana nudi pregled svojstva odabranog atributa. U donjem desnom uglu nalaze se gumbi za manipuliranje atributnom tablicom (dodavanje novih atributa i brisanje postojećih), kao i gumbi za pohranu promjena, odnosno odbacivanje promjena (Slika 18.).



Slika 18. Sučelje alata Attribute Settings

Metoda `setupUi()` ima istu ulogu kao `__init__()` metoda u prethodnom modulu, a to je da prikaže grafičko sučelje. Svi elementi u sučelju su kreirani statično osim popisa imena atributa. Popis atributa nije ništa drugo nego skup `QPushButton`

objekata kojima smo izmijenili izgled i ponašanje. Na taj način omogućili smo `QPushButtonu` da se koristi za označavanje željenog atributa.

```
for (i, field) in fields.iteritems():
    self.attributeList["attributeField" + str(i)] =
            QPushButton(self.scrollAreaWidgetContents)
    self.attributeList["attributeField" + str(i)].setSizePolicy(sizePolicy)
    self.attributeList["attributeField" + str(i)].setMinimumSize(QSize(130, 20))
    self.attributeList["attributeField" + str(i)].setMaximumSize(QSize(130, 20))
    self.attributeList["attributeField" + str(i)].setCheckable(True)
    self.attributeList["attributeField" + str(i)].setAutoExclusive(True)
    self.attributeList["attributeField" + str(i)].setFlat(True)
    self.attributeList["attributeField" + str(i)].setObjectName(_fromUtf8("pushField" +
            str(i)))
    self.attributeList["attributeField" + str(i)].setText(field.name())
    self.attributeList["attributeField" + str(i)].toggled.connect(lambda new, i=i:
            (self.attributeSelected(i, fields, provider, caps)))
```

Metoda `attributeSelected()` aktivira se svaki put kada označimo ime atributa. Povezuje ime atributa s atributnom tablicom pa iz nje uzima podatke o atributu. Ti podatci se tada prikazuju u kućicama na desnoj strani prozora (*Type*, *Name*, *Length* i *Precision*).

Međutim to nije njihova jedina svrha. Kako se sadržaj u kućicama može mijenjati od strane korisnika, program je definiran da prati bilo kakve promjene. Ukoliko dođe do promjene, aktivira se signal koji prenosi nastale promjene kao argumente u `makeField()` metodu.

Metoda `makeField()` preuzima sve promjene nad poljima i pomoću njih stvara novi `QgsField` objekt, tj. novi atribut .

```
#naredba za kreiranje novog atributa tipa "real"
self.newField = QgsField(self.a, 6, "Real", self.d, self.e)

# naredba za kreiranje novog atributa tipa "string"
self.newField = QgsField(self.a, 10, "String", self.d)
```

Tijekom stvaranja novog atributnog polja QGIS ne dozvoljava da mu dužina bude veća od 20 ili preciznost veća od 5 ukoliko se radi o `real` vrsti podataka. Kada ove dvije vrijednosti zadovoljavaju uvjete moguće je pritisnuti *Add Attribute* gumb koji aktivira metodu `addAttribute()`.

Metoda `addAttribute()` radi upravo ono što joj i ime govori. Ona uzima prethodno kreirani `QgsField` objekt i dodaje ga kao novo atributno polje vektorskog sloja. Međutim, ne zapisuje ga direktno u atributnu tablicu nego ga stavlja u međuspremnik[7].

```
cLayer.startEditing()
cLayer.beginEditCommand('Add Attribute')
```

Međuspremnik je ovdje vrlo koristan jer nam omogućava dodavanje novog atributa u popis i nastavak rada nad atributima, ali i odbacivanje promjena ako tako odlučimo. Tek kada pritisnemo gumb za spremanje (*Save*) dajemo do znanja programu da nastale promjene spremi.

Metoda `deleteAttribute()` označeni atribut odstranjuje s popisa, također u međuspremnik, pa se obrisani atribut lako može povratiti. Kao i kod dodavanja atributa, promjene se prenose u glavni modul gdje se mogu ili pohraniti ili odbaciti.

```
cLayer.startEditing()
cLayer.beginEditCommand('Delete Attribute')
cLayer.deleteAttribute(i)
cLayer.endEditCommand()
```

[7] Odnosi se na stog koji pamti stanje programa prije i poslije izvršene naredbe (eng. *Undo/Redo stack*)

5. UPUTE ZA RUKOVANJE

Prvo je potrebno unutar *Quantum GIS* aplikacije učitati proširenje. U glavnom izborniku odaberemo *Plugins → Manage Plugins...* naredbu. Otvara se prozor s izbornikom proširenja. Ako je proširenje izrađeno prema prethodno navedenim pravilima i smješteno u direktorij za proširenja trebalo bi se nalaziti na tome popisu.

Označimo *Attribute Changer* i pritiskom na gumb *OK* zatvorimo prozor. Na donjoj traci *QGIS* sučelja pojavljuju se dva nova prečaca. Jedan služi za pokretanje alata *Attribute Changer* ( ), dok drugi služi za pokretanje *Attribute Settings* alata ( ). Kako naše proširenje radi s vektorskim podatcima, neophodno je prije upotrebe u *QGIS* učitati vektorski sloj te ga označiti.

Aktiviranjem alata *Attribute Changer* ( ) kursor miša mijenja izgled u križ, a program očekuje da njime označimo željeni vektorski element (točka, linija ili poligon) u prikazu. Nakon označavanja elementa, na ekranu se otvori *Attribute Changer* prozor. Prozor sadrži popis atributnih vrijednosti označenog elementa. Svaku vrijednosti možemo promijeniti i spremiti pritiskom tipke *Save*, odnosno odbaciti promjene pritiskom tipke *Cancel* (Slika 19.).

Prelaskom pokazivačem miša preko kućica gdje su ispisane vrijednosti atributa, pojavljuje se obavijest o svojstvima atributa (vrsta podataka, maksimalna dužina zapisa...). Tijekom izmjene atributnih vrijednosti program prati unos i upozorava ukoliko unešena vrijednost ne odgovara osobini atributa. U tom slučaju kućica za upis mijenja boju u crvenu, a lijevo od kućice javlja se objašnjenje upozorenja.

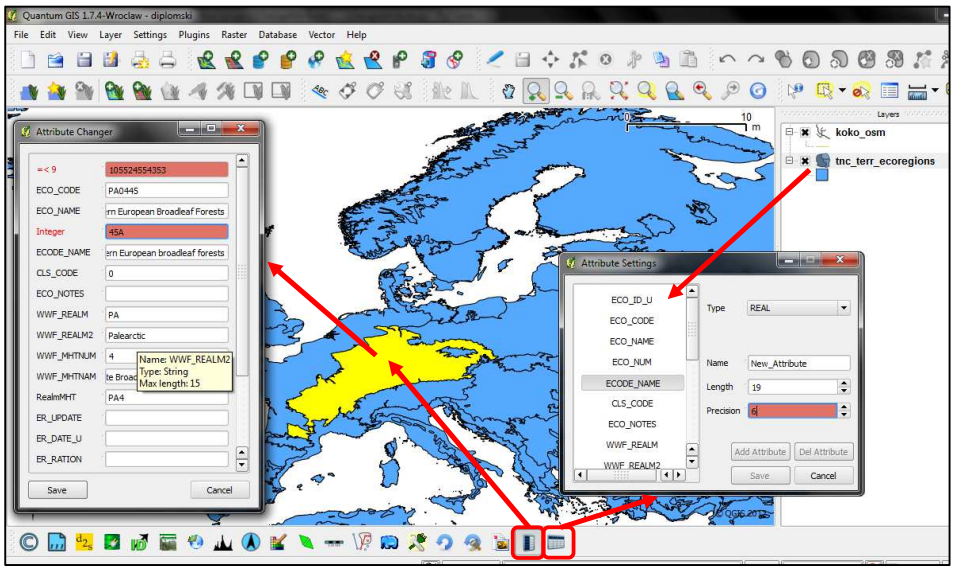
Nakon spremanja ili odbacivanja promjena možemo odabrati sljedeći vektorski element za izmjenu ili deaktivirati alat jednostrukim pritiskom miša na isti prečac.

Ako želimo urediti skup atributa označenog sloja, umjesto njihovih vrijednosti, tada biramo *Attribute Settings* alat ( ). Aktivacijom ovog alata otvara se prozor koji s

lijeve strane sadrži popis svih atributa u trenutnom sloju. Odabirom atributa s popisa, s desne strane se ispisuju svojstva tog atributa (*Type*, *Name*, *Length* i *Precision*). Ispod kućica s vrijednostima nalaze se tipke za dodavanja i brisanje atributa, odnosno spremanje i odbacivanje promjena.

Da bi izbrisali atribut dovoljno ga je označiti u popisu te pritisnuti tipku *Del Attribute*. Važno je napomenuti da u nekim situacijama nije moguće izbrisati atribut iz sloja. Ovo je izazvano zbog ograničenja upravljačkog programa OGR biblioteke koja je zadužena za čitanje i pisanje vektorskih podataka. Problem je moguće izbjeći instalacijom najnovije verzije GDAL biblioteke na računalo.

Dodavanje atributa zavisi o vrijednostima u kućicama iznad. Naime, osim ispisivanja svojstva označenog atributa, one služe i za definiranje novog atributa. Promjenom bilo koje stavke program automatski uzima trenutne vrijednosti kao vrijednosti svojstva novog atributa. Program ovdje također prati upisane vrijednosti i upozorava na nevaljani unos (Slika 19.). Kada definiramo sve stavke (*Type*, *Name*, *Length* i *Precision*), pritiskom tipke *Add Attribute* dodajemo novi atribut na začelje popisa, odnosno u trenutni sloj.



Slika 19. Odnos alata proširenja sa sučeljem QGIS-a

6. ZAKLJUČAK

Tema ovog rada zahtijevala je proučavanje i implementaciju Quantum GIS platforme u izradi modernih softverskih rješenja. U svrhu temeljitijeg proučavanja platforme razvijeno je proširenje za QGIS nazvano *Attribute Changer*, čija je zadaća pojednostavniti i ubrzati izmjenu atributnih vrijednosti vektorskih slojeva učitanih u aplikaciju.

Izrada i dokumentiranje procesa izrade proširenja predstavlja efikasno rješenje problema. Iako razvoj samostalne aplikacije u pravilu zahtijeva korištenje većeg broja QGIS klasa, izrada ovog proširenja pruža sve informacije potrebne za razumijevanje uloge QGIS platforme. Osim toga, rad opisuje i preostale komponente (Python i PyQt) koje u sintezi s QGIS platformom čine kompletan bazis za izradu aplikacije.

Razvoj aplikacije kao i razvoj proširenja nije banalan posao te zahtijeva poznavanje naprednog programiranja. Iako ovaj rad ne može poslužiti kao cjelovit izvor znanja o programiranju, zasigurno može poslužiti kao vodič kroz razvoj aplikacije koji će nam razvrstati proces na osnovne elemente, pojasniti način djelovanja i na kraju ukazati na što treba usmjeriti posebnu pozornost. Osim objašnjenja koncepta programiranja, ovaj rad pojašnjava i praktični dio izrade proširenja, a to je implementacija koda u odnosu na funkcije koje program obavlja.

Na posljetku, uzevši u obzir iskustvo stečeno tijekom istraživanja teme ovog diplomskog rada, QGIS platforma se dokazala kao još jedan vrijedan resurs koji pojednostavljuje i skraćuje vrijeme potrebno za izradu proširenja. Ovo posebno vrijedi za programe koje se bave rješavanjem problema iz područja geoinformatike jer su u stanju iskoristiti sav potencijal koji platforma nudi. Nažalost, trenutno na Internetu postoji samo jedan vodič za primjenu QGIS platforme pomoću Pythona,

koji je još k tome siromašan sadržajem. Zbog toga je neophodno kombinirati vodič s QGIS-ovom C++ API dokumentacijom.

QGIS platforma je odličan pomoćni alat za izradu aplikacija jer zbog svoje jednostavnosti omogućuje relativno laku prilagodbu aplikacije vlastitim potrebama. Osim toga moguće ju je iskoristiti za rješavanje više različitih problema. Tako uz izradu manjih i usko specijaliziranih proširenja omogućuje i izradu potpuno samostalnih softverskih paketa. Ovo je vrlo važno jer osigurava ulogu QGIS-a u budućnosti. Kako profesionalni softverski paketi drže visoke cijene, sve više komercijalnih kompanija i tvrtki okreće se slobodnim rješenjima radi rezanja troškova. QGIS platforma je zbog visokog stupnja prilagođavanja i GPL licence odlična alternativa.

LITERATURA

Ferg, S. (2006.): Event-Driven Programming: Introduction, Tutorial, History.

Philips, D. (2010.): Python 3 Object Oriented Programming.

Shaw, Z. A. (2011.): Learn Python The Hard Way, Release 2.0.

Summerfield, M. (2008.): Rapid GUI Programming with Python and Qt.

Vasić, V., Marković, J. (2004.): Python programski jezik, Univerzitet u Novom Sadu, Tehnološki fakultet

Quantum GIS Coding and Compilation Guide, Quantum GIS Development Team.

Popis URL-ova

URL-1: Python Download, http://www.python.org/download/, 5.6.2012.

URL-2: PyQt4 Package Download, http://www.riverbankcomputing.co.uk/software/pyqt/download/, 10.6.2012.

URL-3: PyQt's Modules, http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/modules.html, 10.6.2012.

URL-4: Quantum GIS Modules, http://qgis.org/api/modules.html, 10.6.2012.

URL-5: PyQGIS Developer Cookbook, http://qgis.org/pyqgis-cookbook/#indices-and-tables, 15.6.2012.

URL-6: Qt Designer/Python for Windows in 30 Mins, http://talk.maemo.org/showthread.php?t=43663, 15.7.2012.

URL-7: PyQt4 tutorial, http://zetcode.com/tutorials/pyqt4/, 16.6.2012.

URL-8: Python tutorial, http://zetcode.com/tutorials/pythontutorial/, 5.6.2012.

URL-9: Qt Concepts, http://www.commandprompt.com/community/pyqt/c1036#AEN1040, 18.7.2012.

URL-10: Python GUI Development, https://www.youtube.com/playlist?list=PLA955A8F9A95378CE&feature=mh_lolz, 23.7.2012.

URL-11: Using Qt Designer, http://www.riverbankcomputing.co.uk/static/Docs/PyQt4/html/designer.html, 17.7.2012.

URL-12: Creating GUI Applications in Python with Qt, http://www.cs.usfca.edu/~afedosov/qttut/, 16.7.2012.

URL-13: Building a simple Plug-in, http://www.qgisworkshop.org/html/workshop/plugins_tutorial.html, 5.6.2012.

URL-14: Python Course: Object-oriented programming, http://www.python-course.eu/object_oriented_programming.php, 5.6.2012.

URL-15: PyQt – Signals, Slots and Layouts Tutorial, http://www.harshj.com/2009/05/14/pyqt-signals-slots-and-layouts-tutorial/, 5.6.2012.

URL-16: Qt Designer Manual, http://doc.qt.nokia.com/4.7-snapshot/designer-manual.html, 15.7.2012.

URL-17: Lambda Tutorial, http://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial 23.7.2012.

POPIS SLIKA

POPIS TABLICA

PRILOZI

PRILOG A - Izvorni kod proširenja

MODUL changeattribute.py

```
 1  # Uvoz PyQt i QGIS biblioteka
 2  from PyQt4.QtCore import *
 3  from PyQt4.QtGui import *
 4  from qgis.core import *
 5  from qgis.gui import *
 6  # Inicijalizacija Qt resursa iz resources.py modula
 7  import resources
 8  # Uvoz klasa koje definiraju dijaloge
 9  from changeattributedialog import AttributeListDialog, AttributeSettingsDialog
10
11
12  class ChangeAttribute:
13
14      def __init__(self, iface):
15          # Referiranje na sučelje QGIS-a
16          self.iface = iface
17          # Referiranje na prostor za prikaz
18          self.canvas = self.iface.mapCanvas()
19          # Inicijalizacija alata koji stvara QgsPoint objekt nakon svakog pritiska
20            tipke miša
21          self.clickTool = QgsMapToolEmitPoint(self.canvas)
22
23
24      def initGui(self):
25          # Kreiranje postupka koji pokreće proširenje
26          self.action = QAction(QIcon(":/plugins/changeattribute/icon1.png"), \
27              "Attribute Changer", self.iface.mainWindow())
28          self.action2 = QAction(QIcon(":/plugins/changeattribute/icon2.png"),\
29              "Attribute Settings", self.iface.mainWindow())
30          # Spajanje postupka sa metodama
31          self.action.setCheckable(True)
32          self.action.changed.connect(self.checkIfEnabled)
33          self.action2.triggered.connect(self.attributeSettings)
34
35          # Dodavanje prečaca u traku za proširenja i glavni izbornik
36          self.iface.addToolBarIcon(self.action)
37          self.iface.addPluginToMenu("&Attribute Changer", self.action)
38          self.iface.addToolBarIcon(self.action2)
39          self.iface.addPluginToMenu("&Attribute Settings", self.action2)
40
41
42
43      def checkIfEnabled (self):
44          state = self.action.isChecked()
45          if state:
46              self.canvas.setMapTool(self.clickTool)
47              self.clickTool.canvasClicked.connect(self.selectFeature)
48          else:
49              self.canvas.unsetMapTool(self.clickTool)
```

```
            self.clickTool.canvasClicked.disconnect(self.selectFeature)
    def unload(self):
        # Uklanjanje proširenja iz sučelja QGIS-a
        self.iface.removePluginMenu("&Attribute Changer",self.action)
        self.iface.removeToolBarIcon(self.action)
        self.iface.removePluginMenu("&Attribute Settings",self.action2)
        self.iface.removeToolBarIcon(self.action2)


    # Namještanje poslužitelja vektorskih podataka da filtrira elemente pomoću
       geometrije pravokutnika
    def selectFeature(self, point, button):

        pntGeom = QgsGeometry.fromPoint(point)
        # Transformacija točke u površinu
        pntBuff = pntGeom.buffer( (self.canvas.mapUnitsPerPixel() * 2),0)
        rect = pntBuff.boundingBox()
        # Dohvaćanje trenutno označenog sloja i poslužitelja podataka
        cLayer = self.canvas.currentLayer()
        selectList = []
        if cLayer:
            provider = cLayer.dataProvider()
            feat = QgsFeature()
            # Kreiranje naredbe za grafičko označavanje elementa u prikazu
            allAttrs = provider.attributeIndexes()
            provider.select(allAttrs, rect) # Značenje argumenata: vrati geometriju
             svih elemenata koji nose atribute iz allAttrs liste i koji se
             podudaraju s geometrijom rect objekta
            while provider.nextFeature(feat):
                # Filtrira prethodnu geometriju tako da odabire samo one elemente
                  koji se dotiču površine nastale iz točke
                if feat.geometry().intersects(pntBuff):
                    selectList.append(feat.id())
                    attrMap = feat.attributeMap()
                    fields = cLayer.pendingFields()
                    break


            if selectList:
                # Označava element na grafičkom prikazu
                cLayer.setSelectedFeatures(selectList)
                self.canvas.unsetMapTool(self.clickTool)
                dlg = AttributeListDialog(fields, attrMap)
                dlg.show()
                result = dlg.exec_()
                if result == 1:
                    cLayer.startEditing()
                    for i in range(len(fields)):
                        cLayer.changeAttributeValue(feat.id(), i,
                         dlg.ui.fieldList["valueField" + str(i)].text())
                    cLayer.commitChanges()
                self.canvas.setMapTool(self.clickTool)


        else:
            QMessageBox.information( self.iface.mainWindow(),"Info", "No layer
             currently selected" )
```

```
115
116     def attributeSettings(self):
117         cLayer = self.canvas.currentLayer()
118         if cLayer:
119             provider = cLayer.dataProvider()
120             fields = cLayer.pendingFields()
121             caps = provider.capabilities()
122             dlg2 = AttributeSettingsDialog(fields, cLayer, provider, caps)
123             dlg2.show()
124             result2 = dlg2.exec_()
125             if result2 == 0:
126                 cLayer.rollBack()
127             elif result2 == 1:
128                 cLayer.commitChanges()
129         else:
130             QMessageBox.information( self.iface.mainWindow(),"Info", "No layer
131              currently selected" )
```

MODUL changeattributedialog.py

```
1   from PyQt4 import QtCore, QtGui
2   from ui_changeattribute import Ui_AttributeList
3   from ui_settings import Ui_SettingsDialog
4
5   class AttributeListDialog(QtGui.QDialog):
6       def __init__(self, fields, attrMap):
7           QtGui.QDialog.__init__(self)
8           self.ui = Ui_AttributeList(self, fields, attrMap)
9
10  class AttributeSettingsDialog(QtGui.QDialog):
11      def __init__(self, fields, cLayer, provider, caps):
12          QtGui.QDialog.__init__(self)
13          self.ui = Ui_SettingsDialog()
14          self.ui.setupUi(self, fields, cLayer, provider, caps)
```

MODUL ui_changeattribute.py

```
1  from PyQt4.QtCore import *
2  from PyQt4.QtGui import *
3
4  try:
5      _fromUtf8 = QString.fromUtf8
6  except AttributeError:
7      _fromUtf8 = lambda s: s
8
9
10 class Ui_AttributeList(object):
11     def __init__(self, Dialog, fields, attrMap):
12
13         maxSize=0
14         for (i,value) in fields.iteritems():
15             if len(fields[i].name()) > maxSize:
16                 maxSize=len(fields[i].name())
17
18         labelWidth = maxSize*8
19
20         Dialog.setObjectName(_fromUtf8("Dialog"))
21         Dialog.setWindowModality(Qt.ApplicationModal)
22         Dialog.setEnabled(True)
23         Dialog.resize(302, 457)
24         Dialog.setSizeGripEnabled(False)
25         flags = Qt.WindowStaysOnTopHint
26         Dialog.setWindowFlags(flags)
27         self.gridLayout = QGridLayout(Dialog)
28         self.gridLayout.setObjectName(_fromUtf8("gridLayout"))
29         self.saveButton = QPushButton(Dialog)
30         self.saveButton.setObjectName(_fromUtf8("saveButton"))
31         self.gridLayout.addWidget(self.saveButton, 1, 0, 1, 1)
32         spacerItem = QSpacerItem(40, 20, QSizePolicy.Expanding, QSizePolicy.Minimum)
33         self.gridLayout.addItem(spacerItem, 1, 1, 1, 1)
34         self.scrollArea = QScrollArea(Dialog)
35         self.scrollArea.setWidgetResizable(True)
36         self.scrollArea.setObjectName(_fromUtf8("scrollArea"))
37         self.scrollAreaWidgetContents = QWidget()
38         self.scrollAreaWidgetContents.setGeometry(QRect(0, 0, 282, 408))
39
40         self.scrollAreaWidgetContents.setObjectName(_fromUtf8("scrollAreaWidgetCont
41         ents"))
42         self.formLayout = QFormLayout(self.scrollAreaWidgetContents)
43         self.formLayout.setFieldGrowthPolicy(QFormLayout.AllNonFixedFieldsGrow)
44         self.formLayout.setObjectName(_fromUtf8("formLayout"))
45         self.fieldList = {}
46         self.attributeList = []
47
48         palette1 = QPalette()
49         brush = QBrush(QColor(226, 115, 105))
50         brush.setStyle(Qt.SolidPattern)
51         palette1.setBrush(QPalette.Active, QPalette.Base, brush)
52         brush = QBrush(QColor(226, 115, 105))
53         brush.setStyle(Qt.SolidPattern)
54         palette1.setBrush(QPalette.Inactive, QPalette.Base, brush)
55         brush = QBrush(QColor(240, 240, 240))
56         brush.setStyle(Qt.SolidPattern)
57         palette1.setBrush(QPalette.Disabled, QPalette.Base, brush)
58
59         palette2 = QPalette()
60         brush2 = QBrush(QColor(255, 255, 255))
61         brush2.setStyle(Qt.SolidPattern)
62         palette2.setBrush(QPalette.Active, QPalette.Base, brush2)
```

```
63          brush2 = QBrush(QColor(255, 255, 255))
64          brush2.setStyle(Qt.SolidPattern)
65          palette2.setBrush(QPalette.Inactive, QPalette.Base, brush2)
66          brush2 = QBrush(QColor(240, 240, 240))
67          brush2.setStyle(Qt.SolidPattern)
68          palette2.setBrush(QPalette.Disabled, QPalette.Base, brush2)
69
70          palette3 = QPalette()
71          brush3 = QBrush(QColor(255, 0, 4))
72          brush3.setStyle(Qt.SolidPattern)
73          palette3.setBrush(QPalette.Active, QPalette.WindowText, brush3)
74          brush3 = QBrush(QColor(255, 0, 4))
75          brush3.setStyle(Qt.SolidPattern)
76          palette3.setBrush(QPalette.Inactive, QPalette.WindowText, brush3)
77          brush3 = QBrush(QColor(120, 120, 120))
78          brush3.setStyle(Qt.SolidPattern)
79          palette3.setBrush(QPalette.Disabled, QPalette.WindowText, brush3)
80
81          palette4 = QPalette()
82          brush4 = QBrush(QColor(0, 0, 0))
83          brush4.setStyle(Qt.SolidPattern)
84          palette4.setBrush(QPalette.Active, QPalette.WindowText, brush4)
85          brush4 = QBrush(QColor(0, 0, 0))
86          brush4.setStyle(Qt.SolidPattern)
87          palette4.setBrush(QPalette.Inactive, QPalette.WindowText, brush4)
88          brush4 = QBrush(QColor(120, 120, 120))
89          brush4.setStyle(Qt.SolidPattern)
90          palette4.setBrush(QPalette.Disabled, QPalette.WindowText, brush4)
91
92          for (i, value) in attrMap.iteritems():
93              self.splitter = QSplitter(self.scrollAreaWidgetContents)
94              self.splitter.setOrientation(Qt.Horizontal)
95              self.splitter.setObjectName(_fromUtf8("splitter_{0}".format(i)))
96              self.attributeNameField = QLabel(self.splitter)
97              self.attributeList.append(self.attributeNameField)
98
99              self.attributeList[i].setObjectName(_fromUtf8("attributeNameField_{0}".f
100             ormat(i)))
101             self.attributeList[i].setMinimumSize(QSize(labelWidth, 0))
102             self.attributeList[i].setMaximumSize(QSize(labelWidth, 16777215))
103             self.attributeList[i].setText(_fromUtf8("{0}".format(fields[i].name())))
104             self.fieldList["valueField" + str(i)] = QLineEdit(self.splitter)
105             self.fieldList["valueField" +
106             str(i)].setObjectName(_fromUtf8("attributeValueField_{0}".format(i)))
107
108             try:
109                 self.fieldList["valueField" +
110             str(i)].setText(_fromUtf8("{0}".format(value.toString())))
111             except UnicodeEncodeError:
112                 codedValue = value.toString().toUtf8()
113                 self.fieldList["valueField" +
114                  str(i)].setText(_fromUtf8("{0}".format(codedValue)))
115
116             maxLength = fields[i].length()
117             typeName = fields[i].typeName()
118             attType = fields[i].type()
119             prec = fields[i].precision()
120             if attType == 6:
121                 self.fieldList["valueField" + str(i)].setToolTip('Name: {0} \nType:
122                  {1} \nMax length: {2} \nPrecision:
123                  {3}'.format(fields[i].name(),typeName,maxLength, prec))
124             else:
125                 self.fieldList["valueField" + str(i)].setToolTip('Name: {0} \nType:
126                  {1} \nMax length: {2}'.format(fields[i].name(),typeName,maxLength))
127
```

```
                self.fieldList["valueField" +
                str(i)].cursorPositionChanged.connect(lambda old, new, i=i:
                (self.checkState(i, fields, palette1, palette2, palette3,
                palette4)))
            self.formLayout.setWidget(i, QFormLayout.SpanningRole, self.splitter)


        self.scrollArea.setWidget(self.scrollAreaWidgetContents)
        self.gridLayout.addWidget(self.scrollArea, 0, 0, 1, 3)
        self.cancelButton = QPushButton(Dialog)
        self.cancelButton.setObjectName(_fromUtf8("cancelButton"))
        self.gridLayout.addWidget(self.cancelButton, 1, 2, 1, 1)

        self.retranslateUi(Dialog)
        QObject.connect(self.cancelButton, SIGNAL(_fromUtf8("clicked()")),
        Dialog.reject)
        QObject.connect(self.saveButton, SIGNAL(_fromUtf8("clicked()")),
        Dialog.accept)
        QMetaObject.connectSlotsByName(Dialog)

    def retranslateUi(self, Dialog):
        Dialog.setWindowTitle(QApplication.translate("Dialog", "Attribute Changer",
        None, QApplication.UnicodeUTF8))
        self.saveButton.setText(QApplication.translate("Dialog", "Save", None,
        QApplication.UnicodeUTF8))
        self.cancelButton.setText(QApplication.translate("Dialog", "Cancel", None,
        QApplication.UnicodeUTF8))

    def checkState(self, i, fields, palette1, palette2, palette3, palette4):
        allowedChars = ['0','1','2','3','4','5','6','7','8','9','.',',']
        # Provjera da li je zapis duži od maksimalnog
        if len(self.fieldList["valueField" + str(i)].text()) > fields[i].length():
            self.fieldList["valueField" + str(i)].setPalette(palette1)
            self.attributeList[i].setText(_fromUtf8("=<
            {0}".format(fields[i].length())))
            self.attributeList[i].setPalette(palette3)
        # Ako nije, provjera da li je zapis nešto drugo osim stringa
        elif fields[i].type() != 10:
            a = 0
            if fields[i].type() == 2:
                allowedChars = allowedChars[:-2]
            # Provjera da li se u zapisu pojavljuje NE-brojčani znak
            for item in self.fieldList["valueField" + str(i)].text():
                if item not in allowedChars:
                    self.fieldList["valueField" + str(i)].setPalette(palette1)

                     self.attributeList[i].setText(_fromUtf8("{0}".format(fields[i]
                     .typeName())))
                    self.attributeList[i].setPalette(palette3)
                    a = 1
                    break
            # Ako je prethodno pronađen NE-brojčani znak, a u novom upisu je SVE
             ispravljeno, tada maknemo crvenu paletu
            if a == 0:
                self.fieldList["valueField" + str(i)].setPalette(palette2)

                 self.attributeList[i].setText(_fromUtf8("{0}".format(fields[i].name
                 ())))
                self.attributeList[i].setPalette(palette4)
        # Ovdje se podrazumijeva da je zapis u granicama dužine i da su svi znakovi
          dopušteni jer je vrsta podataka string
        else:
            self.fieldList["valueField" + str(i)].setPalette(palette2)
            self.attributeList[i].setText(_fromUtf8("{0}".format(fields[i].name())))
            self.attributeList[i].setPalette(palette4)
```

MODUL ui_settings.py

```
1   from PyQt4.QtCore import *
2   from PyQt4.QtGui import *
3   from qgis.core import *
4
5   try:
6       _fromUtf8 = QString.fromUtf8
7   except AttributeError:
8       _fromUtf8 = lambda s: s
9
10  class Ui_SettingsDialog(QObject):
11      def setupUi(self, Dialog, fields, cLayer, provider, caps):
12
13          self.attributeList = {}
14
15          typeList = ['INTEGER','REAL','STRING']
16          Dialog.setObjectName(_fromUtf8("Dialog2"))
17          Dialog.setEnabled(True)
18          Dialog.resize(370, 270)
19          sizePolicy = QSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
20          sizePolicy.setHorizontalStretch(0)
21          sizePolicy.setVerticalStretch(0)
22          sizePolicy.setHeightForWidth(Dialog.sizePolicy().hasHeightForWidth())
23          Dialog.setSizePolicy(sizePolicy)
24          Dialog.setMinimumSize(QSize(370, 270))
25          Dialog.setMaximumSize(QSize(370, 270))
26          Dialog.setSizeGripEnabled(False)
27          Dialog.setModal(False)
28          flags = Qt.WindowStaysOnTopHint
29          Dialog.setWindowFlags(flags)
30          self.scrollArea = QScrollArea(Dialog)
31          self.scrollArea.setGeometry(QRect(10, 10, 161, 251))
32          sizePolicy = QSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
33          sizePolicy.setHorizontalStretch(0)
34          sizePolicy.setVerticalStretch(0)
35
36           sizePolicy.setHeightForWidth(self.scrollArea.sizePolicy().hasHeightForWidth())
37          self.scrollArea.setSizePolicy(sizePolicy)
38          self.scrollArea.setMinimumSize(QSize(161, 251))
39          self.scrollArea.setMaximumSize(QSize(161, 251))
40          self.scrollArea.setWidgetResizable(True)
41          self.scrollArea.setObjectName(_fromUtf8("scrollArea"))
42          self.scrollAreaWidgetContents = QWidget()
43          self.scrollAreaWidgetContents.setGeometry(QRect(0, 0, 159, 249))
44
45          palette = QPalette()
46          brush = QBrush(QColor(255, 255, 255))
47          brush.setStyle(Qt.SolidPattern)
48          palette.setBrush(QPalette.Active, QPalette.Base, brush)
49          brush = QBrush(QColor(255, 255, 255))
50          brush.setStyle(Qt.SolidPattern)
51          palette.setBrush(QPalette.Active, QPalette.Window, brush)
52          brush = QBrush(QColor(255, 255, 255))
53          brush.setStyle(Qt.SolidPattern)
54          palette.setBrush(QPalette.Inactive, QPalette.Base, brush)
55          brush = QBrush(QColor(255, 255, 255))
56          brush.setStyle(Qt.SolidPattern)
57          palette.setBrush(QPalette.Inactive, QPalette.Window, brush)
58          brush = QBrush(QColor(255, 255, 255))
59          brush.setStyle(Qt.SolidPattern)
60          palette.setBrush(QPalette.Disabled, QPalette.Base, brush)
61          brush = QBrush(QColor(255, 255, 255))
62          brush.setStyle(Qt.SolidPattern)
```

```
63          palette.setBrush(QPalette.Disabled, QPalette.Window, brush)
64
65          self.palette1 = QPalette()
66          brush = QBrush(QColor(226, 115, 105))
67          brush.setStyle(Qt.SolidPattern)
68          self.palette1.setBrush(QPalette.Active, QPalette.Base, brush)
69          brush = QBrush(QColor(226, 115, 105))
70          brush.setStyle(Qt.SolidPattern)
71          self.palette1.setBrush(QPalette.Inactive, QPalette.Base, brush)
72          brush = QBrush(QColor(240, 240, 240))
73          brush.setStyle(Qt.SolidPattern)
74          self.palette1.setBrush(QPalette.Disabled, QPalette.Base, brush)
75
76          self.palette2 = QPalette()
77          brush2 = QBrush(QColor(255, 255, 255))
78          brush2.setStyle(Qt.SolidPattern)
79          self.palette2.setBrush(QPalette.Active, QPalette.Base, brush2)
80          brush2 = QBrush(QColor(255, 255, 255))
81          brush2.setStyle(Qt.SolidPattern)
82          self.palette2.setBrush(QPalette.Inactive, QPalette.Base, brush2)
83          brush2 = QBrush(QColor(240, 240, 240))
84          brush2.setStyle(Qt.SolidPattern)
85          self.palette2.setBrush(QPalette.Disabled, QPalette.Base, brush2)
86
87          self.scrollAreaWidgetContents.setPalette(palette)
88          self.scrollAreaWidgetContents.setAutoFillBackground(True)
89
90         self.scrollAreaWidgetContents.setObjectName(_fromUtf8("scrollAreaWidgetContent
91         s"))
92          self.formLayout = QFormLayout(self.scrollAreaWidgetContents)
93          self.formLayout.setFieldGrowthPolicy(QFormLayout.AllNonFixedFieldsGrow)
94          self.formLayout.setObjectName(_fromUtf8("formLayout"))
95
96          sizePolicy = QSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
97          sizePolicy.setHorizontalStretch(0)
98          sizePolicy.setVerticalStretch(0)
99
100         self.scrollArea.setWidget(self.scrollAreaWidgetContents)
101         self.splitter_2 = QSplitter(Dialog)
102         self.splitter_2.setGeometry(QRect(180, 130, 181, 20))
103         self.splitter_2.setOrientation(Qt.Horizontal)
104         self.splitter_2.setObjectName(_fromUtf8("splitter_2"))
105         self.lengthLabel = QLabel(self.splitter_2)
106         self.lengthLabel.setObjectName(_fromUtf8("lengthLabel"))
107         self.lengthSpinBox = QSpinBox(self.splitter_2)
108         self.lengthSpinBox.setEnabled(True)
109         self.lengthSpinBox.setMinimumSize(QSize(130, 0))
110         self.lengthSpinBox.setMaximumSize(QSize(130, 16777215))
111         self.lengthSpinBox.setMinimum(1)
112         self.lengthSpinBox.setMaximum(1000)
113         self.lengthSpinBox.setObjectName(_fromUtf8("lengthSpinBox"))
114         self.splitter_3 = QSplitter(Dialog)
115         self.splitter_3.setGeometry(QRect(180, 160, 181, 20))
116         self.splitter_3.setOrientation(Qt.Horizontal)
117         self.splitter_3.setObjectName(_fromUtf8("splitter_3"))
118         self.precisionLabel = QLabel(self.splitter_3)
119         self.precisionLabel.setObjectName(_fromUtf8("precisionLabel"))
120         self.precisionSpinBox = QSpinBox(self.splitter_3)
121         self.precisionSpinBox.setMinimumSize(QSize(130, 0))
122         self.precisionSpinBox.setMaximumSize(QSize(130, 16777215))
123         self.precisionSpinBox.setObjectName(_fromUtf8("precisionSpinBox"))
124         self.splitter = QSplitter(Dialog)
125         self.splitter.setGeometry(QRect(180, 100, 181, 20))
126         self.splitter.setOrientation(Qt.Horizontal)
127         self.splitter.setObjectName(_fromUtf8("splitter"))
```

```
128         self.nameLabel = QLabel(self.splitter)
129         self.nameLabel.setObjectName(_fromUtf8("nameLabel"))
130         self.nameLineEdit = QLineEdit(self.splitter)
131         self.nameLineEdit.setMinimumSize(QSize(90, 0))
132         self.nameLineEdit.setMaximumSize(QSize(130, 16777215))
133         self.nameLineEdit.setObjectName(_fromUtf8("nameLineEdit"))
134         self.saveBtn = QPushButton(Dialog)
135         self.saveBtn.setEnabled(False)
136         self.saveBtn.setGeometry(QRect(210, 240, 75, 23))
137         self.saveBtn.setObjectName(_fromUtf8("saveBtn"))
138         self.saveBtn.clicked.connect(Dialog.accept)
139         self.cancelBtn = QPushButton(Dialog)
140         self.cancelBtn.setGeometry(QRect(290, 240, 75, 23))
141         self.cancelBtn.setObjectName(_fromUtf8("cancelBtn"))
142         self.cancelBtn.clicked.connect(Dialog.reject)
143         self.addBtn = QPushButton(Dialog)
144         self.addBtn.setGeometry(QRect(210, 210, 75, 23))
145         self.addBtn.setObjectName(_fromUtf8("addBtn"))
146         self.addBtn.setEnabled(False)
147         self.addBtn.clicked.connect(lambda: self.addAttribute(provider, cLayer,
148          fields, Dialog, sizePolicy, caps))
149         self.deleteBtn = QPushButton(Dialog)
150         self.deleteBtn.setGeometry(QRect(290, 210, 75, 23))
151         self.deleteBtn.setObjectName(_fromUtf8("deleteBtn"))
152         self.deleteBtn.setEnabled(False)
153         self.widget = QWidget(Dialog)
154         self.widget.setGeometry(QRect(180, 30, 181, 22))
155         self.widget.setObjectName(_fromUtf8("widget"))
156         self.horizontalLayout = QHBoxLayout(self.widget)
157         self.horizontalLayout.setMargin(0)
158         self.horizontalLayout.setObjectName(_fromUtf8("horizontalLayout"))
159         self.typeLabel = QLabel(self.widget)
160         self.typeLabel.setMinimumSize(QSize(45, 0))
161         self.typeLabel.setMaximumSize(QSize(42, 16777215))
162         self.typeLabel.setObjectName(_fromUtf8("typeLabel"))
163         self.horizontalLayout.addWidget(self.typeLabel)
164         self.typeComboBox = QComboBox(self.widget)
165         self.typeComboBox.setObjectName(_fromUtf8("typeComboBox"))
166         self.typeComboBox.addItems(typeList)
167         self.horizontalLayout.addWidget(self.typeComboBox)
168
169         self.counter = 0
170
171         for (i, field) in fields.iteritems():
172             self.attributeList["attributeField" + str(i)] =
173             QPushButton(self.scrollAreaWidgetContents)
174             self.attributeList["attributeField" + str(i)].setSizePolicy(sizePolicy)
175             self.attributeList["attributeField" + str(i)].setMinimumSize(QSize(130,
176             20))
177             self.attributeList["attributeField" + str(i)].setMaximumSize(QSize(130,
178             20))
179             self.attributeList["attributeField" + str(i)].setCheckable(True)
180             self.attributeList["attributeField" + str(i)].setAutoExclusive(True)
181             self.attributeList["attributeField" + str(i)].setFlat(True)
182             self.attributeList["attributeField" +
183             str(i)].setObjectName(_fromUtf8("pushField" + str(i)))
184             self.attributeList["attributeField" + str(i)].setText(field.name())
185             self.attributeList["attributeField" + str(i)].toggled.connect(lambda new,
186             i=i: (self.attributeSelected(i, fields, provider, caps)))
187             self.formLayout.setWidget(i, QFormLayout.LabelRole,
188              self.attributeList["attributeField" + str(i)])
189
190         self.attributeList["attributeField0"].setChecked(True)
191         self.retranslateUi(Dialog)
192         QMetaObject.connectSlotsByName(Dialog)
```

```
193     def retranslateUi(self, Dialog):
194         Dialog.setWindowTitle(QApplication.translate("Dialog", "Attribute Settings",
195          None, QApplication.UnicodeUTF8))
196         self.lengthLabel.setText(QApplication.translate("Dialog", "Length", None,
197          QApplication.UnicodeUTF8))
198         self.precisionLabel.setText(QApplication.translate("Dialog", "Precision",
199          None, QApplication.UnicodeUTF8))
200         self.nameLabel.setText(QApplication.translate("Dialog", "Name", None,
201          QApplication.UnicodeUTF8))
202         self.saveBtn.setText(QApplication.translate("Dialog", "Save", None,
203          QApplication.UnicodeUTF8))
204         self.cancelBtn.setText(QApplication.translate("Dialog", "Cancel", None,
205          QApplication.UnicodeUTF8))
206         self.addBtn.setText(QApplication.translate("Dialog", "Add Attribute", None,
207          QApplication.UnicodeUTF8))
208         self.deleteBtn.setText(QApplication.translate("Dialog", "Del Attribute", None,
209          QApplication.UnicodeUTF8))
210         self.typeLabel.setText(QApplication.translate("Dialog", "Type", None,
211          QApplication.UnicodeUTF8))
212 
213     def attributeSelected(self, i, fields, provider, caps):
214         if caps & provider.DeleteAttributes:
215             self.deleteBtn.setEnabled(True)
216             self.deleteBtn.clicked.connect(lambda: self.deleteAttribute(i, cLayer))
217 
218         if caps & provider.AddAttributes:
219             self.addBtn.setEnabled(False)
220             self.counter += 1
221             if self.counter > 1:
222                 self.typeComboBox.currentIndexChanged.disconnect(self.makeField)
223                 self.nameLineEdit.textChanged.disconnect(self.makeField)
224                 self.lengthSpinBox.valueChanged.disconnect(self.makeField)
225                 self.precisionSpinBox.valueChanged.disconnect(self.makeField)
226 
227         self.nameLineEdit.setText(fields[i].name())
228         if fields[i].type() == 10:
229             self.typeComboBox.setCurrentIndex(2)
230             self.lengthSpinBox.setValue(fields[i].length())
231             self.precisionSpinBox.setValue(0)
232             self.precisionSpinBox.setEnabled(False)
233         elif fields[i].type() == 2:
234             self.typeComboBox.setCurrentIndex(0)
235             self.lengthSpinBox.setValue(fields[i].length())
236             self.precisionSpinBox.setValue(0)
237             self.precisionSpinBox.setEnabled(False)
238         else:
239             self.typeComboBox.setCurrentIndex(1)
240             self.lengthSpinBox.setValue(fields[i].length())
241             self.precisionSpinBox.setEnabled(True)
242             self.precisionSpinBox.setValue(fields[i].precision())
243 
244 
245 
246         self.a = self.nameLineEdit.text()
247         self.b = self.typeComboBox.currentIndex()
248         self.c = self.typeComboBox.currentText()
249         self.d = self.lengthSpinBox.value()
250         self.e = self.precisionSpinBox.value()
251 
252         if caps & provider.AddAttributes:
253 
254             self.typeComboBox.currentIndexChanged.connect(self.makeField)
255             self.nameLineEdit.textChanged.connect(self.makeField)
256             self.lengthSpinBox.valueChanged.connect(self.makeField)
257             self.precisionSpinBox.valueChanged.connect(self.makeField)
```

```
258     def makeField(self, argument):
259 
260         sender = self.sender()
261 
262         if sender == self.nameLineEdit:
263             self.a = argument
264         elif sender == self.typeComboBox:
265             self.b = argument
266         elif sender == self.lengthSpinBox:
267             self.d = argument
268         else:
269             self.e = argument
270 
271         if self.lengthSpinBox.value() > 20 and self.typeComboBox.currentIndex() != 2:
272             self.lengthSpinBox.setPalette(self.palette1)
273             a = 0
274         else:
275             self.lengthSpinBox.setPalette(self.palette2)
276             a = 1
277 
278         if self.precisionSpinBox.value() > 5:
279             self.precisionSpinBox.setPalette(self.palette1)
280             b = 0
281         else:
282             self.precisionSpinBox.setPalette(self.palette2)
283             b = 1
284         if a + b == 2:
285             self.addBtn.setEnabled(True)
286         else:
287             self.addBtn.setEnabled(False)
288 
289         if self.b == 0:
290             self.newField = QgsField(self.a, 2, "Integer", self.d)
291             self.precisionSpinBox.setEnabled(False)
292             self.precisionSpinBox.setValue(0)
293 
294         elif self.b == 1:
295             self.precisionSpinBox.setEnabled(True)
296             self.newField = QgsField(self.a, 6, "Real", self.d, self.e)
297         else:
298             self.newField = QgsField(self.a, 10, "String", self.d)
299             self.precisionSpinBox.setEnabled(False)
300             self.precisionSpinBox.setValue(0)
301 
302 
303     def deleteAttribute(self, i, cLayer):
304         self.saveBtn.setEnabled(True)
305         cLayer.startEditing()
306         cLayer.beginEditCommand('Delete Attribute')
307         cLayer.deleteAttribute(i)
308         cLayer.endEditCommand()
309 
310     def addAttribute(self, provider, cLayer, fields, Dialog, sizePolicy, caps):
311         self.nameExist = False
312         for (i, field) in fields.iteritems():
313             if field.name() == self.newField.name():
314                 self.nameExist = True
315                 break
316 
317         if self.nameExist:
318             QMessageBox.information(Dialog, "Info", "Name {0} is already
319              taken!".format(self.newField.name()))
320         elif self.newField.name() == '':
321             QMessageBox.information(Dialog, "Info", "Name not specified!")
322 
```

```
323         else:
324             self.addBtn.setEnabled(False)
325             self.saveBtn.setEnabled(True)
326             cLayer.startEditing()
327             cLayer.beginEditCommand('Add Attribute')
328             cLayer.addAttribute(self.newField)
329             i = len(self.attributeList)
330             fields[i] = self.newField
331             self.attributeList["attributeField" + str(i)] =
332             QPushButton(self.scrollAreaWidgetContents)
333             self.attributeList["attributeField" + str(i)].setSizePolicy(sizePolicy)
334             self.attributeList["attributeField" + str(i)].setMinimumSize(QSize(130,
335             20))
336             self.attributeList["attributeField" + str(i)].setMaximumSize(QSize(130,
337             20))
338             self.attributeList["attributeField" + str(i)].setCheckable(True)
339             self.attributeList["attributeField" + str(i)].setAutoExclusive(True)
340             self.attributeList["attributeField" + str(i)].setFlat(True)
341             self.attributeList["attributeField" +
342             str(i)].setObjectName(_fromUtf8("pushField" + str(i)))
343             self.attributeList["attributeField" +
344             str(i)].setText(self.newField.name())
345             self.attributeList["attributeField" + str(i)].toggled.connect(lambda new,
346             i=i: (self.attributeSelected(i, fields, provider, caps)))
347             self.attributeList["attributeField" + str(i)].setChecked(True)
348             self.formLayout.setWidget(i, QFormLayout.LabelRole,
349             self.attributeList["attributeField" + str(i)])
350             cLayer.endEditCommand()
```

PRILOG B - Sadržaj priloženog medija

Br.	Ime datoteke	Opis
1.	Diplomski rad.pdf	Tekst diplomskog rada
2.	ChangeAttribute	Mapa koja sadrži izvorni kod proširenja

ŽIVOTOPIS

CURRICULUM	VITAE
Osobni podatci	
Ime i prezime	Siniša Slovenec
Adresa	Jalšje 22, 49214 Veliko Trgovišće
Telefon	+385 91 333 18 10
E-mail	sslovenec@gmail.com
Godina rođenja	1987.
Radno iskustvo	
ljeto 08.; ljeto 09.	**GEO-BT d.o.o.**, Zabok Zadaci: - terenska izmjera - izrada elaborata
jesen 11.	**Ocean Media d.o.o.**, Zaprešić Zadaci: - izrada 3D modela - 3D vizualizacija
Školovanje	
rujan 06. - rujan 12.	**Geodetski fakultet**, Zagreb
Fakultet	Smjer: Geoinformatika
listopad 02. - lipanj 06.	**Gimnazija A.G. Matoš**, Zabok
Srednja škola	Smjer: Prirodoslovno - matematički

Strani jezici	
	Engleski jezik
Čitanje	Napredno
Pisanje	Napredno
Govor	Napredno
	Njemački jezik
Čitanje	Osnovno
Pisanje	Osnovno
Govor	Osnovno
Znanja i vještine	
Tehničke sposobnosti	**Geodetske sposobnosti** Kratak opis: Teorijsko znanje stečeno tijekom petogodišnjeg studija. Praktično iskustvo stečeno tijekom honorarnih poslova na terenu i u uredu uglavnom katastarske domene. **Računalne sposobnosti** Kratak opis: Vrlo dobro poznavanje AutoCAD softverskog paketa. Odlično snalaženje u MS Office paketu, točnije Wordu, Excelu i PowerPointu. Napredno znanje vizualizacije 3D modela u 3ds Maxu zajedno s V-Rayem. Osnovno znanje korištenja Adobe Photoshopa. **Programski jezici** Kratak opis: Vrlo dobro poznavanje programskog jezika Python zajedno s Qt i QGIS platformom. Osnovno znanje izrade aplikacije. Osnovno znanje C++ i Pascal jezika.