A PYTHON PROGRAMMING LIBRARY FOR QUICK DEVELOPMENT OF SELF-REGISTERING PLUGIN SYSTEMS

P. Marić[1], Ž. Živanov[1], M. Hajduković[1], D. D. Milašinović[2]

[1] University of Novi Sad, Faculty of Technical Sciences, Novi Sad, Serbia,
{petarmaric, zzarko, hajduk}@uns.ac.rs
[2] University of Novi Sad, Faculty of Civil Engineering, Subotica, Serbia,
ddmil@gf.uns.ac.rs

Abstract (English): *Software solutions which require dynamic adding and changing of functionality while software is running have sometimes been known to use the plugin system concept for implementation. This approach suggests that the existing functionality of the software should be decomposed into modules that are implemented as independent plugins. This paper presents the meta-programming based Python software library named "simple_plugins" that offers easy and rapid development of expandable systems for the automatic registration of plugins. This library also offers automatic deactivation of individual plugins, depending on the software's run-time environment. Software library and its source code is publicly available under the BSD Open Source license.*

Abstract (Serbian): *Prilikom razvoja softverskih rešenja, kod kojih postoji potreba za dinamičkim dodavanjem i izmenom funkcionalnosti u toku rada, kao način implementacije može se primeniti koncept sistema programskih dodataka (eng. "plugin system"). Upotrebom takvog pristupa, postojeće funkcionalnosti softverskog rešenja se mogu modularizovati i implementirati kao nezavisni programski dodaci. U ovom radu je predstavljena "Python" programska biblioteka "simple_plugins" koja na bazi metaprogramiranja nudi jednostavan i brz razvoj proširivih sistema za automatsko registrovanje programskih dodataka. Ova biblioteka ostavlja i mogućnost automatskog deaktiviranja pojedinih programskih dodataka, u zavisnosti od okruženja u kojem se softversko rešenje izvršava. Programska biblioteka i njen izvorni kod su javno dostupni pod "BSD Open Source" licencom.*

Keywords: *metaprogramming, plugins, Python, Open Source*

1. Uvod

Po kojoj god metodologiji da se radi, sastavni deo životnog ciklusa softvera predstavlja neki oblik njegovog održavanja, koje ponekad uključuje i dodavanje novih funkcionalnosti ili izmenu postojećih. U zavisnosti od interne organizacije softvera i raspoloživosti izvornog koda krajnjim korisnicima, ovakve izmene može odraditi samo strana koja je inicijalno i razvijala program, ili to mogu odraditi i krajnji korisnici. U oba slučaja, izmene mogu zahtevati promenu postojećeg izvornog koda, i/ili dodavanje novih datoteka.

Pojedine kategorije softvera zahtevaju mogućnost izmene funkcionalnosti i u toku njegovog izvršavanja, odnosno mogućnost da se, bez gašenja programa, izmeni način rada pojedinih njihovih komponenti. I ovakve dinamičke izmene rada softvera mogu biti ograničene na mogućnosti koje su unapred predviđene (recimo izbor između nekoliko predefinisanih načina prikaza podataka), ili mogu biti dodavane nakon što je softver isporučen korisniku.

U različitim programskim jezicima se mogu koristiti različiti mehanizmi koje postoje u njima da se realizuje dinamička izmena funkcionalnosti, te se može reći i da je način realizacije dinamičke izmene funkcionalnosti zavisan i od konkretnog programskog jezika. U ovom radu će biti predstavljen jedan sistem dinamičkog proširenja rada programa za programski jezik *Python* [1], zasnovan na ideji programskih dodataka (engl. *plugin*). Programski dodaci, pored toga što omogućavaju dinamičku izmenu funkcionalnosti, za pojedine klase problema predstavljaju i zgodan način modularizacije programa, čime se olakšava nezavisan razvoj i testiranje njegovih komponenti. Programske dodatke može razvijati bilo ko, ako se upozna sa načinom rada sistema programskih dodataka za konkretan softver, u toku razvoja samog softvera, ili nakon što je on ušao u upotrebu.

Tipična kategorija softvera koja koristi programske dodatke su softver za reprodukciju multimedijalnih datoteka, gde se korišćenjem programskih dodataka od strane trećih lica razvija podrška za nove ulazne formate datoteka, novi načini vizuelizacije sadržaja multimedijalnih datoteka, ili kompletne izmene korisničkog interfejsa. Naravno, sistem programskih dodataka se može uspešno primeniti i na druge kategorije softvera kod kojih je potrebna izmena funkcionalnosti u toku rada.

2. Postojeći pristupi u razvoju sistema programskih dodataka za *Python* programski jezik

Analizom popularnih *Open Source* projekata, baziranih na *Python* programskom jeziku, uočena su dva osnovna pristupa, koje sistemi programskih dodataka koriste za formiranje repozitorijuma lokacija *Python* modula sa programskim dodacima:

- aktivni, gde sistem samostalno pokušava da otkrije lokacije modula sa programskim dodacima; npr. pretraga fajl sistema na predefinisanim putanjama za datotekama određenog naziva
- pasivni, gde je sistemu potrebno na neki način dostaviti lokacije modula sa programskim dodacima:
  - ručno (ili polu-automatizovano) navođenje lokacija modula u specijalnoj konfiguracionoj datoteci
  - uspostavljanje konvencije da su sami programski dodaci odgovorni da, prilikom svoje instalacije, kreiraju specijalne datoteke u predefinisanom direktorijumu, čime sadržaj takvog direktorijuma efektivno predstavlja repozitorijum lokacija modula sa programskim dodacima.

Nakon što se ustanovi koji *Python* moduli potencijalno poseduju programske dodatke, postoje različiti mehanizmi registrovanja dodataka:

- automatizovani, gde sistem sam registruje sve *Python* klase/funkcije u modulu koje:
  - implementiraju određeni interfejs
  - imaju jedan od predefinisanih naziva
- manuelni, gde autor programskog dodatka, prilikom inicijalizacije svog modula:
  - ručno poziva pomoćnu funkciju putem koje se registruju njegovi programski dodaci
  - navodi spisak programskih dodataka koje treba registrovati.

Python podržava koncept metaklasa [2-6], jer u programskom jeziku *Python* svaka klasa je instanca odgovarajuće metaklase [2], kao što je i svaki objekat instanca odgovarajuće klase. Samim tim metaklasa može da kontroliše atribute i ponašanje klase, ili pak da utiče na sam način kreiranja klase [3]. Stoga metaklase značajno olakšavaju napredno metaprogramiranje i pružaju nove interesantne mogućnosti u razvoju složenih softverskih rešenja (npr. videti *Python* paket `django.db.models` iz popularnog *Django* [7] *Open Source* projekta).

Marty Alchin je predstavio interesantno alternativno rešenje [8] za automatsko registrovanje programskih dodataka, koje radi na bazi metaprogramiranja, metaklasa i drugih dinamičkih osobenosti *Python* programskog jezika. On je razvio *Python* metaklasu koja će već činom navođenja u okviru neke klase, dinamički redefinisati način kreiranja, atribute i ponašanje kako te klase, tako i svih njenih trenutnih i budućih klasa naslednica. Ovim je postigao da se sve klase, koje koriste njegovu metaklasu (bilo to direktno ili indirektno kroz mehanizam nasleđivanja), automatski registruju kao programski dodaci odgovarajuće bazne klase, i to već u fazi učitavanja sadržaja *Python* programskog modula.

3. Programska biblioteka `simple_plugins`

U okviru projekta Ministarstva prosvete, nauke i tehnološkog razvoja „Računarska mehanika u teoriji konstrukcija" (OI 174027) se, između ostalog, radi na razvoju softvera za analizu materijalnih konstrukcija, koji kao rezultat rada daju podatke u stanju konstrukcije nakon različitih spoljašnjih i unutrašnjih uticaja. Jedan deo tih proračuna se oslanja na rešavanje složenih diferencijalnih jednačina za različite vrste greda (engl. *beam*) i realizovan je u okviru programske biblioteke `beam_integrals` [9, 10, 11]. Prilikom razvoja ove biblioteke ukazala se potreba da se, u zavisnosti od zadatog graničnog uslova, izmeni način izračunavanja integrala i rešavanja karakterističnih jednačina, kao i da se novi granični uslovi i načini računanja mogu brzo i lako dodavati. Analizom postojećih rešenja za *Python* programski jezik se uočilo da nijedan u potpunosti ne odgovara konkretnom problemu, što je i dovelo do realizacije programske biblioteke `simple_plugins` [12], nastale kao proširenje postojećeg sistema programskih dodataka *M. Alchin*-a. Biblioteka `simple_plugins` je svoj život započela kao integralni programski modul `beam_integrals` biblioteke, ali je relativno brzo uočeno da ima znatan potencijal za upotrebu i u drugim softverskim rešenjima, te je stoga izdvojena kao posebna celina. Do momenta pisanja ovog rada, programski paket biblioteke je preuzet preko 4600 puta [13], što samo govori u prilog njenoj korisnosti.

S obzirom na to da se `simple_plugins` oslanja na *Python* programski jezik, u njegovoj realizaciji su iskorištene karakteristike tog programskog jezika, kao i njegova postojeća programska podrška [14]. Pošto je sam programski jezik platformski neutralan, to je bila dobra osnova da i `simple_plugins` bude platformski neutralno rešenje [14-16]. Treba naglasiti da je od samog početka razvoja celokupno rešenje javno dostupno [12], pod *BSD Open Source* licencom [17].

Osnovna ideja se bazira na rešenju koje je predstavio *M. Alchin* [8], dok je implementacija `AttrDict` klase preuzeta sa [18]. Implementacija konfiguracije podsistema programskih dodataka, kroz unutrašnju `Meta` klasu, inspirisana je *Python* paketom `django.db.models` iz *Django Open Source* projekta.
Biblioteka `simple_plugins` proširuje rešenje koje je predstavio *M. Alchin* i uvodi dodatna poboljšanja, npr:

1. Svaka klasa, koja nasledi baznu klasu podsistema programskih dodataka (eng. *plugin mount point*) [8], biće registrovana kao programski dodatak, osim u slučaju da naziv te klase počinje sa stringom `'Base'`. Time se programeru pruža mogućnost da izbegne registrovanje apstraktnih ili parcijalno definisanih klasa.
2. Svaki kreirani podsistem programskih dodataka se može konfigurisati kroz dodavanje unutrašnje `Meta` klase u baznu klasu podsistema programskih dodataka, pri čemu su dostupna sledeća podešavanja:
   a) `id_field`, koji predstavlja naziv atributa klase koji će se koristiti za jednoznačnu identifikaciju konkretne klase unutar podsistema programskih dodataka. Ako se ovo podešavanje ne navede, koristiće se atribut `id`.
   b) `id_field_coerce`, koji predstavlja funkciju koja će se koristiti prilikom poziva metode `coerce` za konverziju prosleđene joj vrednosti u tip podatka koji odgovara atributu klase na koji ukazuje podešavanje `id_field`, a zarad identifikacije konkretne klase unutar podsistema programskih dodataka. Ako se ovo podešavanje ne navede, koristiće se *Python* funkcija `int`.
3. Mogućnost deregistracije postojećeg programskog dodatka korišćenjem skrivene metode `_unregister_plugin`, koja je automatski ubačena u baznu klasu podsistema programskih dodataka.
4. Keširano dinamičko svojstvo `plugins` koje je automatski ubačeno u baznu klasu podsistema programskih dodataka i sadrži bogate informacije o trenutno registrovanim programskim dodacima:
   a) `classes`, koji predstavlja spisak klasa registrovanih programskih dodataka
   b) `instances`, koji predstavlja spisak objekata koji su instance klasa registrovanih programskih dodataka. Koristi se za optimizaciju utroška radne memorije i kreiranje isključivo jednog objekta (tzv. *singleton pattern* [19]) za svaki registrovani programski dodatak
   c) `id_to_instance`, koji predstavlja mapiranje sa *ključa* na *vrednost* (Python tip *dict* [20]), gde je *ključ* vrednost atributa na koji ukazuje podešavanje `id_field`, a *vrednost* je *singleton* objekat registrovanog programskog dodatka
   d) `id_to_class`, koji predstavlja mapiranje sa *ključa* na *vrednost*, gde je *ključ* vrednost atributa na koji ukazuje podešavanje `id_field`, a *vrednost* je klasa registrovanog programskog dodatka
   e) `class_to_id`, koji predstavlja mapiranje sa *ključa* na *vrednost*, gde je *ključ* klasa registrovanog programskog dodatka, a *vrednost* je vrednost atributa na koji ukazuje podešavanje `id_field`
   f) `instances_sorted_by_id`, koji predstavlja spisak objekata koji su instance klasa registrovanih programskih dodataka, gde je spisak sortiran po vrednosti atributa na koji ukazuje podešavanje `id_field`
   g) `valid_ids`, koji predstavlja spisak vrednosti atributa na koji ukazuje podešavanje `id_field`, za sve klase registrovanih programskih dodataka.
5. Metoda `coerce`, koja je automatski ubačena u baznu klasu podsistema programskih dodataka, pruža mogućnost konverzije prosleđene joj vrednosti u odgovarajući *singleton* objekat registrovanog programskog dodatka, primenom funkcije na koji ukazuje podešavanje `id_field_coerce`, a na osnovu dinamičkog svojstva `plugins.id_to_instance`.

3.1. Primer upotrebe

Jedan primer dela mogućnosti programske biblioteke `simple_plugins` dat je na narednim listinzima, koji skupa čine jednostavan *Python* program za učitavanje slika sa proširivom podrškom za rad sa više različitih formata.

Na početku potrebno je učitati potrebne biblioteke i definisati baznu klasu podsistema programskih dodataka:

```
import argparse
import os
from friendly_name_mixin import FriendlyNameFromClassMixin
from simple_plugins import CoercionError, PluginMount


class BaseImageFormat(FriendlyNameFromClassMixin):
    __metaclass__ = PluginMount

    class Meta:
        id_field = 'file_type'
        id_field_coerce = str
```

```
    @property
    def file_type(self):
        return self.name.lower().split()[0]

    def load(self, filename):
        print "Loading '%s' via %s" % (filename, type(self))
```

gde je kroz unutrašnju `Meta` klasu navedeno da će se dinamički određeno svojstvo `file_type` koristiti za jednoznačnu identifikaciju programskog dodatka unutar podsistema.

Nakon toga potrebno je definisati željene klase programskih dodataka (zarad kratkoće primera data je minimalna implementacija), gde će sve klase koje naslede baznu klasu `BaseImageFormat` biti automatski registrovane kao njeni programski dodaci:

```
class JPEGImageFormat(BaseImageFormat): pass
class PNGImageFormat(BaseImageFormat): pass
class BMPImageFormat(BaseImageFormat): pass
```

Na kraju, potrebno je definisati glavnu funkciju programa:

```
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('filename')
    args = parser.parse_args()

    try:
        file_type = os.path.splitext(args.filename.lower())[1][1:]
        image_loader = BaseImageFormat.coerce(file_type)
        image_loader.load(args.filename)
    except CoercionError:
        print "ERROR: Unknown image format for '%s'!" % args.filename,
        print "Supported file types: %s" % ', '.join(BaseImageFormat.plugins.valid_ids)

if __name__ == '__main__':
    main()
```

gde je prvo inicijalizovana *Python* programska podrška za parsiranje argumenata komandne linije, putem koje će se od krajnjeg korisnika preuzimati naziv željene datoteke. Upotrebom `BaseImageFormat.coerce` metode, koju je biblioteka `simple_plugins` automatski ubacila u baznu klasu, vrši se konverzija prosleđene joj vrednosti (tip datoteke) u odgovarajući *singleton* objekat programskog dodatka. Ukoliko postoji odgovarajući programski dodatak za traženi tip datoteke pozvaće se njegova `load` metoda:

```
$ python example.py foo.png
Loading 'foo.png' via <class '__main__.PNGImageFormat'>
```

Ukoliko u sistemu ne postoji odgovarajući programski dodatak ispisaće se poruka o grešci uz spisak podržanih tipova datoteka, na osnovu dinamički ubačenog svojstva `BaseImageFormat.plugins.valid_ids`:

```
$ python example.py bad.format
ERROR: Unknown image format for 'bad.format'! Supported file types: bmp, jpeg, png
```

Za primere naprednijih mogućnosti pogledati izvorni kod programske biblioteke `beam_integrals` [9, 10].

4. Podsistem za automatsku verifikaciju

Jenkins je poznat kao vodeći *Open Source Continuous Integration server* [21] i koristi se za automatsko prevođenje, testiranje i praćenje toka razvoja biblioteke `simple_plugins`, čiji je javan *Jenkins* projekat dostupan na [22].

Podsistem za automatsku verifikaciju `simple_plugins` biblioteke razvijan je sa ciljem da pokrije svaki aspekt projekta uz 100% pokrivenosti koda testovima i sadrži 16 međusobno nezavisnih testova [22], koje naš *Jenkins* integracioni server automatski izvršava svaki put kada se registruje promena u javno dostupnom repozitorijumu izvornog koda za biblioteku `simple_plugins` [12].

5. Zaključak

U dinamičkim programskim jezicima, kao što je *Python*, metaprogramiranje pruža mogućnosti koje često nisu direktno dostupne u statičkim programskim jezicima. Biblioteka `simple_plugins` koristi upravo takve mehanizme da obezbedi sistem automatskog registrovanja programskih dodataka, koji se lako integriše u *Python* softver. Glavne osobine biblioteke, kao što su njena proširivost, jednostavnost upotrebe i brzina usvajanja u postojećim i novim softverskim rešenjima, bez potrebe za značajnim izmenama postojećeg koda, je izdvajaju od sličnih rešenja. Takođe, biblioteka se može tako konfigurisati da, u zavisnosti od okruženja u kojem se izvršava (različiti operativni sistemi, na primer), automatski deaktivira pojedine programske dodatke, čime se može postići bolja prenosivost krajnjeg softverskog rešenja. Pošto je dostupna pod *BSD Open Source* licencom, biblioteka se može besplatno i slobodno koristiti, kao i menjati ukoliko se ukaže potreba.

Zahvalnica

Ovo istraživanje je deo projekta Ministarstva prosvete, nauke i tehnološkog razvoja „Računarska mehanika u teoriji konstrukcija" (OI 174027). Autori se zahvaljuju na pruženoj podršci.

Literatura

[1] G. Van Rossum, „Python programming language", 1994
[2] C. Măries, „Understanding python metaclasses", http://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/
[3] E. Bendersky, „Python metaclasses by example", http://eli.thegreenplace.net/2011/08/14/python-metaclasses-by-example/
[4] StackOverflow, „What is a metaclass in python?", http://stackoverflow.com/questions/100003/what-is-a-metaclass-in-python
[5] Python Software Foundation, „Python 2.7.9 documentation: Customizing class creation", https://docs.python.org/2/reference/datamodel.html#customizing-class-creation
[6] J. Vanderpla, „A primer on python metaclasses", https://jakevdp.github.io/blog/2012/12/01/a-primer-on-python-metaclasses/
[7] Django Software Foundation, „Django: The Web framework for perfectionists with deadlines", https://www.djangoproject.com/
[8] M. Alchin, „A simple plugin framework", http://martyalchin.com/2008/jan/10/simple-plugin-framework/
[9] P. Marić, D.D. Milašinović, „beam_integrals public code repository", https://bitbucket.org/petar/beam_integrals/src/
[10] P. Marić, „A hybrid software architecture for supporting the harmonic coupled finite strip method", PhD thesis, University of Novi Sad, Faculty of technical sciences, Trg Dositeja Obradovića 6, Novi Sad, Serbia, 2016, http://www.cris.uns.ac.rs/DownloadFileServlet/Disertacija145528213566439.pdf?controlNumber=(BISIS)99999&fileName=145528213566439.pdf&id=4938
[11] P. Maric, D.D. Milašinovic, D. Goleš, Ž. Živanov, M. Hajdukovic (2017), "A Hybrid Software Solution for the Harmonic Coupled Finite Strip Method Characteristic Equations", in P. Iványi, B.H.V. Topping, G. Várady, (Editors), Proceedings of the Fifth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, Civil-Comp Press, Stirlingshire, UK, Paper 31, 2017. doi:10.4203/ccp.111.31
[12] P. Marić, „simple_plugins public code repository", https://bitbucket.org/petar/simple_plugins/src/
[13] Python Developers, „PyPI Statistics | simple_plugins", http://www.pypi-stats.com/package/?q=simple_plugins
[14] Python Software Foundation, „Python 2.7.9 documentation: The Python Standard Library", https://docs.python.org/2/library/
[15] Cunningham & Cunningham, Inc, „c2 wiki: Platform independence", http://c2.com/cgi/wiki?PlatformIndependence
[16] Python Software Foundation, „Python 2.7.9 documentation: Distributing Python Modules", https://docs.python.org/2/distutils/index.html
[17] Open Source Initiative, „The BSD 3-Clause License", http://opensource.org/licenses/BSD-3-Clause
[18] StackOverflow, „Javascript style dot notation for dictionary keys unpythonic?", http://stackoverflow.com/q/224026
[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, „Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, 1994
[20] Python Software Foundation, „Python 2.7.9 documentation: The Python Tutorial – Data Structures", https://docs.python.org/2/tutorial/datastructures.html
[21] Jenkins CI community, „Jenkins CI - An extendable open source continuous integration server", http://jenkins-ci.org/
[22] P. Marić, „simple_plugins Jenkins project page", http://ci.petarmaric.com/job/simple_plugins/