REFLEKSIJA I DINAMIČKO UČITAVANJE KLASA U PROGRAMSKOM JEZIKU JAVA

Samir Alagić[1], Amel Džanić[2]
[1]J.P Željeznice FbiH d.o.o. Sarajevo – Poslovno područje Bihać, Bihaćkih Branilaca bb, samir.alagic@zfbh.ba
[2]Tehnički fakultet Bihać, Ulica Irfana Ljubijankića bb 77000 Bihać, amel.dzanic@gmail.com

Ključne riječi: programski jezici, objektno orjentirano programiranje, refleksija, metapodatak, metaobjekt

SAŽETAK:

U ovom radu daćemo teoretski pregled o refleksiji te istražit ćemo mogućnosti objektno orijentiranih jezika u pogledu refleksije, na početku sa teoretskog aspekta, a zatim ćemo istražiti Javine mogućnosti u oblasti refleksije u polju dinamičkog učitavanja klasa te ćemo dati jedan jednostavan primjer kako se može refleksija i dinamičko učitavanje klasa iskoristiti za pravljenje fleksibilnog softvera.

1. UVOD

Etimološki, refleksija se u svom izvornom značenju odnosi na optički fenomen odražavanja slike objekta od ravne površine. Sljedeće značenje koju je riječ vremenom poprimila označavalo je primjenu (nečega) na samoga sebe. U računarstvu se pojam refleksije odnosi na domenu programa koji (stvarno ili potencijalno) opisuju i mijenjaju sebe ili srodne programe. Ideja upotrebe refleksije nije potpuno nova, korištena je od strane osnivača formalne logike i računarske znanosti u njihovim ranim teoremima o snazi i granicama rasuđivanja i računanja (*Cantor*, *Gödel*, *Turing*, itd.).
Pojam refleksije u moderno računarstvo uveo je *Brian Cantwell Smith* u svojoj doktorskoj disertaciji *„Proceduralna refleksija u programskim jezicima"* (*„Procedural Reflection in Programming Languages"*, Ph.D. Thesis, MIT) u kojoj tvrdi da „ako je moguće izgraditi proces računanja čija je namjena rasuđivanje o vanjskom svijetu, a čiji je sastojak proces (interpreter) koji rukuje s formalnim prikazom tog svijeta, onda je moguće izgraditi i takav proces računanja koji će rasuđivati o samome sebi, sadržavajući drugi proces (interpreter) koji rukuje s formalnim prikazom vlastitih operacija i struktura" [5]. Računarsku refleksiju (*computational reflection*) *Pattie Maes* opisuje kao „ponašanje predočeno od strane reflektivnog sistema, gdje je reflektivan sistem onaj računarski sistem koji je u posljedičnoj vezi sa samim sobom" [4]. Postoje dvije vrste reflektivnih arhitektura, razlikovane u ovisnosti o načinu na koji sistem prikazuje samoga sebe:

- proceduralne reflektivne arhitekture i
- deklarativne reflektivne arhitekture.

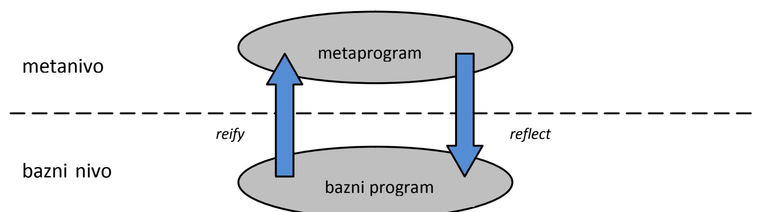
Proceduralna refleksija temelji se na otkrivanju implementacije sistema (programskog kôda), čija promjena kao posljedicu ima i promjenu izvršavanja sistema. Deklarativna arhitektura otkriva sistem (ili dio sistema) korištenjem tvrdnji o sistemu. Tvrdnje su najčešće izražene u obliku ograničenja na ponašanje sistema.

2. REFLEKSIJA

Refleksija omogućava programu da ispita sam sebe i da može napraviti promjene koje utječu na njegovo izvršavanje. Da bi izveo ovo samo-ispitivanje, program treba da ima prikaz samog sebe. *Smith* , u svojoj doktorskoj disertaciji, iznosi tri zahtjeva za sistem da bi on bio reflektivan i to [5]:

1. Sistem mora imati reprezentaciju (predstavljanje) samog sebe.
2. Mora postojati posljedična, uzročna veza (*causal connection*) između sistema i njegove reprezentacije.
3. Sistem mora imati *„odgovarajuću povoljnu tačku"* iz koje može obaviti posao refleksije.

Tri *Smith*-ova zahtjeva za reflektivni sistem mogu se zadovoljiti na način kako su to uradili *Friedman* i *Wand*, što je prikazano na slici 1.



Slika 1. Friedman – Wand-ov model za izvršenje reflektivnog sistema [3]

Ovaj model uvodi operacije reifikacije, otkrivanja (*reification*) i refleksije (*reflection*), bazni program mora se moći renderirati u svoju reprezentaciju, prije nego što metaprogram može početi sa svojim radom. Bazni nivo, program u izvršavanju, ima dostupnu *reify* operaciju koja pretvara pokrenuti program (uključujući *stack frame*, upravljačke registre itd.) u strukturu podataka koju predaje metaprogramu. Metaprogram ispituje bazni program za informacije i pravi promjene koristeći dobivene strukture podataka. Ove promjene se reflektiraju u ponašanju baznog programa, kad on nastavi sa svojim izvršavanjem. Metaprogram poziva *reflect* operaciju koja omogućava nastavljanje izvršenja baznog programa. Metaprogrami rade na sličan način kao što rade i programeri. Programeri pišu programski kôd, tada ispituju vrijednosti polja i pozivaju metode. Kreiramo klase transformacije, dodajemo svojstva i aspekte i tako dalje. Metaprogrami imaju sposobnost mnogih od ovih operacija i oni su algoritamski izražaj ovih aktivnosti. Ukoliko se takve aktivnosti trebaju ponavljati, algoritamski opis može da poveća produktivnost, smanji teškoće, te ukloni ljudske greške [1].

2.1. Reflektivno objektno orijentirano programiranje

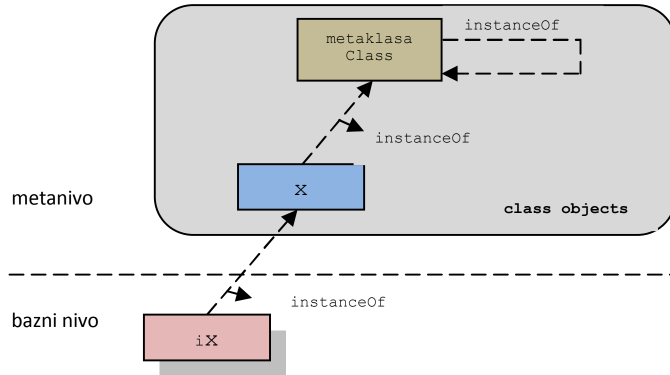
Daljnje opisivanje refleksije biti će zasnovano na klasno baziranom programiranju . Reflektivni klasno bazirani programski jezici trebali bi da se podvrgnu sljedećim postulatima [1]:

1. Postoji ne prazan konačan skup objekata, gdje je svaki objekat identificiran preko vrijednosti koju nazivamo referenca objekta (*object reference*).
2. Svaki objekat ima jedinstveno asociran entitet koji se naziva **klasa**.
3. Svaka klasa je reprezentirana (*reified*) sa objektom.

Na slici 2 ilustrirana je objektno orijentirana organizacija za refleksiju. Metaobjekti (*class object*) se nalaze na metanivou zato što su reifikacija programa. Reflektivna preračunavanja se dešavaju kada se pozivi metoda izvode na objektu klase. Razlika između ove situacije i slike 1 jeste ta da reifikacija postoji kontinuirano kroz izvršavanje programa [2].

Jednostavnost i uniformnost refleksije u objektno orijentiranim jezicima proizilazi iz tretmana klasa i objekata. Pa ipak, kombinacija drugog i trećeg postulata vodi nas do cirkularnosti koja govori da objekti koji predstavljaju klase moraju i sami imati klase. Ove klase moraju također imati objekte koji imaju klase i tako dalje. Ova cirkularnost je jedna od glavnih problema za organiziranje modela koji će zadovoljiti sva tri postulata. Slika 2 ilustrira da relaciju između objekta i njegove reprezentacije

klase možemo posmatrati kao graf. Vrhovi grafa su objekti a njegovi rubovi su *instanceOf* relacija. Ako uzmemo ovako postavljenu sliku kao graf za analizu, shodno sa teoremom iz teorije grafova, može nas uspješno dovesti do organizacije koja će ispuniti sva tri postulata



Slika 2. Objektno orijentirana organizacija za refleksiju sa cikličnom strukturom[1]

2.2. Metaobjekt protokoli

Refleksija „ovlašćuje“ neki program da ispita i modificira samog sebe, gdje su modifikacije povremeno povezane sa karakteristikama programa. Metaobjekti sadrže prikaz programa, a metaobjekt protokol je sučelje, interfejs sa metaobjektima [6]. Razmotrićemo koje vrste operacija bi željeli izvesti sa metaobjekt protokolom.

Introspekcija (*introspection*) je postupak ispitivanja strukture i stanja programa. Ispis imena varijabli instance u nekom objektu je primjer introspektivne upotrebe metaobjekt protokola. Ovo su neki od primjera introspektivnih operacija:

- ispitivanje vrijednosti varijable instance,
- ispis metoda klase.

Intercesija (posredovanje, *intercession*) se odnosi na sposobnost metanivoa da posreduje u operaciji nekog programa i promjeni njegove karakteristike. Tipično, intercesija se pojavljuje pri promjeni strukture programa metanivoa. Ove strukturalne promjene se potom manifestiraju kao promjene ponašanja. Promjena povezivanja (u tabeli metoda) neke metode sa kôdom koji implementira metodu je jedan primjer intercesije. Jednako tako, preskakanje poziva metode i preusmjeravanje istog je također jedan primjer intercesije. Evo još nekih primjera operacije intercesije:

- dodavanje novih metoda klasi,
- dodavanje novih varijabli instanci klasi,
- promjena postavki roditeljskih (*parent*) klasa klase,
- redefiniranje metoda,
- kontroliranje metode odašiljanja.

Svi metaobjekt protokoli imaju ograničenja. Unutrašnje svojstvo (metaobjekt protokola) je neko (izračunljivo) svojstvo koje se može programirati sa metaobjekt protokolom. Vanjsko svojstvo je neko (izračunljivo) svojstvo koje nije unutrašnje. U Javi, na primjer, izračunavanje klase objekta od nekog objekta je unutrašnje svojstvo; jednostavno korištenje metode `getClass`. Sa druge strane, izračunavanje instanci nekog objekta klase je vanjsko svojstvo Java metaobjekt protokola. Java metaobjekt protokol je skoro u potpunosti introspektivan. Prema tome, mnogo je korisnih reflektivnih zadataka koji se ne mogu postići direktno sa Java metaobjekt protokolom. Jedan od ciljeva ovog rada je da pokaže kako prepoznati takve situacije, te da pokaže šta učiniti kada se one pojave.

3. DINAMIČKO UČITAVANJE KLASA U PROGRAMSKOM JEZIKU JAVA

Refleksija dopušta Java programima da učitaju nove klase individualno, ostvarujući veći stepen fleksibilnosti. Budući da se ova operacija dešava u potpunosti unutar *Jave*, ona također održava prenosivost. Dinamičko učitavanje omogućava Java aplikaciji da pronađe i koristi klase koje nisu bile dostupne kada je aplikacija pisana. Kada se kombinira sa dobrim objektno-orijentiranim dizajnom, to povećava fleksibilnost Java aplikacija, povećavajući vjerovatnost održavanja koraka sa promjenama u zahtjevima. Pogodnost metode `Class.forName` vraća objekat klase kojoj je dato u potpunosti kvalificirano ime klase. Zapamtimo funkcionalnost ove metode misleći na *get class for name* (*dobiti klasu za ime*). Mi pozivamo `forName` pogodnu (praktičnu) metodu jer je to statička metoda za programiranje pogodnosti, u ovom slučaju, racionalizacija upotrebe *Class loader*-a [1]. *Class loader* apstrakuje proces učitavanja klase prije njenog prvog instanciranja čime je čini raspoloživom za upotrebu. Svaki put kada se zahtjeva instanciranje klase ili se klasa statički poziva, *Class loader* u pozadini, transparentno za sistem, pronalazi i učitava zahtijevanu klasu u memoriju JVM. Razlozi za postojanje *Class loader*-a se nalaze u samoj osnovi zahtjeva Java jezika: da bude nezavisan od platforme i da podržava distribuirane sisteme. Jezik koji je nezavisan od platforme ne smije da se oslanja na specifični fajl sistem za učitavanje biblioteka. Dalje, pošto se očekuje da Java učitava klase koje se nalaze distribuirane po mreži, korištenje fajl sistema sasvim prestaje da ima smisla. Zato, kada se u programu zahtjeva instanciranje neke klase, JVM zahtjeva od *Class loader*-a da je učita. *Class loader* traži klasu na način na koji je već dizajniran: na fajl sistemu, na mreži, itd. Velika razlika između korištenja `forName` i obične klase je ta što `forName` ne zahtjeva naziv klase za vrijeme prevođenja. `Class.forName` osigurava učitavanje klase i vraća referencu na objekt klase. Ovo je ostvareno korištenjem *Class loader*-a, u pravilu povezanog sa klasom objekta koja poziva metodu `forName`. *Class loader* može imati već učitanu klasu. Ako je tako, sam *Class loader* vraća objekt klase koja je ranije učitana. Ako klasa nije ranije učitana, sistem *Class loader*-a uobičajeno pretražuje putanju klase za odgovarajućim `.class` podatkom. Ako je podatak pronađen, *bytecodes* (kôdovi bajtova) su pročitani i *loader* konstruira klasu objekta. Ako podatak nije pronađen, *loader* baca izuzetak `ClassNotFoundException`. Ponašanje opisano ovdje je ponašanje za sistem *Class loader*. Dinamičko učitavanje proizvodi objekte klase. Iz toga slijedi, da instanciranje ovih objekata klase postaje bitno. Java pruža dvije opcije za kreiranje instanci iz objekata klase: korištenjem samog objekta klase i drugo pomoću metaobjekata koji predstavljaju konstruktore klase. Kada se želi niz, pogodnost klase Array može se koristiti za reflektivno kreiranje niza [1].
Na sljedećem primjeru ćemo pokazati veoma efikasno korištenje dinamičkog učitavanja klase. Budući da je riječ ovdje o jednostavnom primjeru naša namjera je samo da se prikaže kako refleksivan kod može veoma da olakša primjenu novih zahtjeva. Naš primjer se sastoji od programa koji izračunava obim jednakostraničnih geometrijskih tijela. Pretpostvka je da je u prvoj verziji softvera, naručilac, tražio da se računa samo obimi za trokut i četverokut. Na slici je prikazana klasa za Trokut mada sve ostale klase za geometrijska tijela su slične te ih nećemo prikazivati. Također zbog polimorfizma smo implementirali interfejs MetodaObim koji je prikazan na slici 3.

```
package refleksija;
public class Trokut implements MetodaObim{
    private double stranica;
    public void setStranica(double str) {
        this.stranica = stranica;
    }
    public Trokut() {}
        public double obim() {
        return stranica*3;
    }
}
```

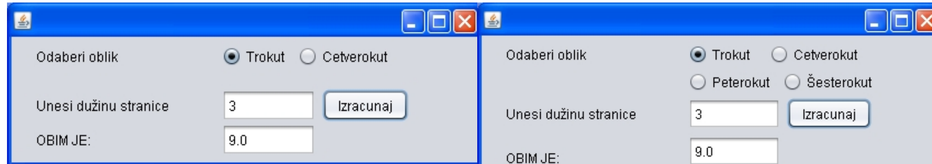
```
package refleksija;

public interface MetodaObim {
    public double obim();
    public void setStranica(double str);
}
```

Slika 3. Listing koda za klasu Trokut i interfejs MetodaObim

Ukoliko se u glavnoj klasi implementira refleksivni kod preko dinamičkog učitavanja klase za razliku od klasičnog koda koji bi zahtijevao *if – then* logiku, za određivanje koju instancu objekta treba

napraviti da bi se izvršila operacija obima za odabrano geometrijsko tijelo, tada dodavanje novih zahtijeva softveru voma elegantno uraditi što ćemo i prikazati . Pretpostavimo da imamo formu kao na slici 4 , te da imamo novi zahtijev od klijenta koji traži uključivanje još peterokuta i šesterokuta.



Slika 4. Prikaz izgleda aplikacije prije i poslije promjene specifikacije

Bez primjene refleksije kod koji bi se izvršavao pri pritisku na dugme izračunaj je prikazan na slici 5.

```
if (jRBTrokut.isSelected()){
            Trokut tr = new Trokut();
            tr.setStranica(Double.parseDouble(jTStranica.getText()));
            jTRezultat.setText(new Double(tr.obim()).toString());
        }else{
            Cetverokut ce = new Cetverokut();
            ce.setStranica(Double.parseDouble(jTStranica.getText()));
            jTRezultat.setText(new Double(ce.obim()).toString());
}
```

Slika 5. Prikaz prvobitnog dijela koda koji vrši odabir tražene metode

Nakon implementacije novih zahtjeva kod se usložnjava i prikazan je na slici 6.

```
if (jRBTrokut.isSelected()){
            Trokut tr = new Trokut();
            tr.setStranica(Double.parseDouble(jTStranica.getText()));
            jTRezultat.setText(new Double(tr.obim()).toString());
        }else if (jRBCetvorokut.isSelected()){
            Cetverokut ce = new Cetverokut();
            ce.setStranica(Double.parseDouble(jTStranica.getText()));
            jTRezultat.setText(new Double(ce.obim()).toString());
         }else if (jRBPeterokut.isSelected())
             Peterokut  p= new Peterokut();
            p.setStranica(Double.parseDouble(jTStranica.getText()));
            jTRezultat.setText(new Double(p.obim()).toString());
        else{
            Sesterokut s = new Sesterokut();
            s.setStranica(Double.parseDouble(jTStranica.getText()));
            jTRezultat.setText(new Double(s.obim()).toString());
}
```

Slika 6. Listing dijela koda koji upravlja sa odabirom tražene metode nakon promjene specifikacije

Kao što možemo vidjeti iz ovih slika da nam se kod sve više posložnjava i ukoliko bi imali drastično povećanje zahtijeva bio bi jako nepregledan. Ukoliko se primjeni refleksivni kod koji omogućuje dinamičko učitavanje klasa tada uopšte nema potreba ga mijenjati za povećanje u zahtijevima, naime kod ostaje isti i prikazan je na slici 7.

```
try {
      try {MetodaObim Geom = (MetodaObim)
                      Class.forName("refleksija." + Klasa).newInstance();
           Geom.setStranica(Double.parseDouble(jTStranica.getText()));
           jTRezultat.setText(new Double(Geom.obim()).toString());
            } catch ( InstantiationException | IllegalAccessException ex) {}
        } catch (ClassNotFoundException ex) {}
```

Slika 7. Listing dijela koda koji je reflektivan i koji vrši odabir tražene metode

Jedini problem kod ovakve realizacije jeste kako prosljediti naziv klase, ali to se može na više načina elegantno uraditi.

4. ZAKLJUČAK

Uopšteno, postoje tri tehnike koje API refleksija može koristiti da bi omogućila promjenu karakteristika: direktna modifikacija metaobjekta, operacije za korištenje metapodataka i intercesija, u kojoj je kôdu dozvoljeno da posreduje u raznim fazama izvršavanja programa. Ove osobine daju refleksiji sposobnost da naš softver učini fleksibilnim. Aplikacije programirane refleksijom se lakše prilagođavaju promjenama u zahtjevima. Reflektivne komponente se vrlo često ponovo besprijekorno koriste i u drugim aplikacijama. Ove prednosti su nam dostupne u našem trenutnom Java razvojnom alatu. Refleksija je moćna, ali nije magična. Moramo ovladati subjektom da bi naš softver učinili fleksibilnim. Nije dovoljno samo naučiti koncepte i upotrebu API. Također moramo biti u stanju da razlikujemo situacije kada je refleksija apsolutno neophodna od onih kada se refleksija može koristiti kao prednost, do onih kada je se treba kloniti. Tri su problema, tj. pitanja koja su uveliko spriječili široku upotrebu refleksije:

- sigurnost,
- složenost koda i
- vrijeme izvršenja operacije.

Briga o sigurnosti je pogrešno usmjerena. Java je toliko dobro napravljena i njena API refleksija tako pažljivo ograničena da se sigurnost lako kontroliše. Učenjem kada koristiti refleksiju a kada ne, možemo izbjeći nepotrebni složeni kôd koji često može biti posljedica amaterske upotrebe refleksije. Na dalje, treba vrijednovati dostignuće (učinak) našeg dizajna, i prema tome osigurati da rezultirajući kôd ispunjava zahtjeve. Troškovi održavanja softvera tri do četiri ili više puta premašuju troškove razvoja. Tržište softvera povećava svoje zahtjeve za fleksibilnosti. Znanje kako proizvesti fleksibilan kôd povećava našu vrijednost na tržištu. Refleksija – introspekcija nakon kojih slijedi promjena ponašanja (karakteristika) – je put prema fleksibilnom softveru. Obećanje refleksije je veliko i njeno vrijeme je došlo.

5. LITERATURA

[1] Ira R. Forman, Nate Forman: „Java Reflection in Action“, Manning, Publications Co., 2005.
[2] P.Wegner: „Dimensions of Object-Based Language Design“, OOPSLA '87 Conference Proceedings, Oktobar 1987.
[3] D.P. Friedman, M. Wand. „Reification: Reflection without Metaphysics“ Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, 1984.
[4] Pattie Maes, „*Concepts and Experiments in Computational Reflection*“, Proceedings of OOPSLA'87, ACM SIGPLAN Notices, 22, 147-155, ACM Press
[5] Brian C. Smith: „*Procedural Reflection in a Procedural Language*“ Ph.D. Thesis, Massachusetts Institute of Technology, 1982.
[6] G. Kiczales, J. des Rivieres, D. G. Bobrow. „The Art of the Metaobject Protocol“ Boston, MA: The MIT Press, 1991.