UDC: 004.41 Info M str. 17-28

ИМПЛЕМЕНТАЦИЈА ИНТЕРНИХ DSL-ОВА НА JVM ПЛАТФОРМИ
INTERNAL DSLS IMPLEMENTATION ON JVM PLATFORM

Милош Шербић, проф. др Зоран Ђурић

РЕЗИМЕ: У раду је представљен појам доменско специфичних језика (енг. *domain specific language* – DSL). Затим су приказани програмерски принципи и шаблони за развој DSL-а, као што су: *fluent interface*, манипулација синтаксним стаблом, уграђивање типова, *reflective metaprogramming*, *runtime metaprogramming* и *compile-time metaprogramming*. Такође, урађен је преглед JVM језика (*Scala*, *Groovy*, *Java*) са њиховим значајним карактеристикама за концизну имплементацију DSL-а. Неке од карактеристика које су приказане, као што су концизна синтакса, преклапање оператора, метапрограмирање, анонимне функције и отворене класе су кључне за имплементацију DSL-а. Примјери кориштење појединих принципа и шаблона за развој DSL-а су приказани у исјечцима програмског кода из практичног рада имплементације DSL-а за обраду слике. Ове имплементације су урађене у *Scala*, *Groovy* и *Java* програмском језику.

КЉУЧНЕ РИЈЕЧИ: Java, Scala, Groovy, JVM, DSL

ABSTRACT: In this paper it was defined and explained term of domain specific language (DSL). Then, it was described principles and patterns for developing internal DSLs: fluent interface, AST manipulation, typed embedding, reflective metaprogramming, runtime metaprogramming and compile-time metaprogramming. Also, it was done overview of JVM languages (Scala, Groovy and Java) with accent on their features for concise DSL implementation. Some of the features which are shown, like concise syntax, operator overloading, metaprogramming, anonymous function and open classes are key features for implementing internal DSLs. Usage of some principles and patterns for developing internal DSLs, are shown on snippets of implemented examples of DSL for Image processing. And this implementations are implemented in Scala, Groovy and Java programming languages.

KEY WORDS: Java, Scala, Groovy, JVM, DSL

1. УВОД

Доменско-специфични језик (енг. *domain specific language* – DSL) представља рачунарски програмски језик ограничене експресивности, који је фокусиран на одређен домен [1]. DSL је програмски језик који има изражену „течност“ (енг. *fluency*) у комуникацији са његовима корисницима. „Течност“ језика као особина природних језика и DSL-а, представља брзину размјене информација у комуникацији између људи (ако се ради о природним језицима), док код DSL интерфејса она представља високи ниво читљивости и прилагођености синтаксе за његове кориснике. Код DSL-а је битно да се корисник чим прије може фокусирати на рјешење проблема, те да не троши вријеме на учење детаља ниског нивоа, који постоје код језика опште намјене као што су нпр. *Java*, *Scala* или *Groovy*. DSL пружа минимум могућности које су довољне да би се ријешио проблем из одређеног домена, али се обично не може користити за рјешавања проблема из других домена, док језик опште намјене се може користити за рјешавања проблема из више домена. DSL има јасну доменску фокусираност. Тако, на примјер, *HTML* представља DSL који служи за презентацију интернет садржаја, а *SQL* представља DSL за креирање упита према бази података. Посматрајући начине имплементације DSL-ова, могуће их је подијелити на интерне, екстерне и графичке (нетекстуалне).

Интерни или уграђени (енг. *embedded*) DSL-ови су DSL-ови који су имплементирани над постојећим језиком опште намјене кориштењем његове инфраструктуре с циљем реализације специфичне DSL семантике [1]. Користећи специфичне опције код појединих језика (метапрограмирање код *Groovy* језика или имплицитну конверзију код *Scala* језика), мијења се постојећа граматика језика над којим се реализује DSL у циљу добијања граматике и синтаксе која ће што више одговарати кориснику DSL-а .

Екстерни DSL-ови се развијају независно од било којег језика, а код оваквог типа DSL-а његов дизајнер мора креирати граматику датог DSL-а, без могућности ослањања на постојећу граматику као што је то случај код интерних DSL-ова. Такође, мора се имплементирати парсирање кода према одговарајућој синтакси, њено процесирање, као и њено мапирање у одговарајућу семантику. Другим ријечима, за реализацију екстерних DSL-ова, потребно је креирање одговарајућих интерпретера и/или компајлера [1].

Графички или нетекстуални DSL-ови представљају графичку презентацију домена [2]. Код текстуалних скрипти, доменску логику је тешко приказати на јасан и прегледан начин, зато што је често исувише комплексна. Многи проблеми из одређених домена се могу лакше и јасније представити графички. Доменски корисник тада може да манипулише графичком репрезентацијом, а по рјешењу проблема могуће је генерисати одговарајућу текстуалну репрезентацију.

У секцији 2 наведени су и описани опште познати шаблони за реализацију DSL-ова. Затим, у секцији 3 описани су *Java*, *Scala* и *Groovy* језици са својим особинама кључним за реализацију интерних DSL-ова. У секцији 4 дата је имплементација DSL-а за обраду слике, с циљем демонстрације примјене наведених шаблона и особина JVM језика. На крају рада дат је закључак.

2. ШАБЛОНИ ЗА РАЕЛИЗАЦИЈУ ИНТЕРНИХ DSL-ОВА

Основни циљ овог рада јесте демонстрација могућности реализације интерних DSL-ова помоћу језика који се извршавају на JVM платформи. Приликом реализације интерних DSL-ова, могу се примјењивати различити шаблони (енг. *pattern*) који користе могућности и особине појединих језика. У овој секцији описани су неки од познатијих шаблона за реализацију интерних DSL-ова.

Smart API или Fluent interface шаблон

За поједине интерне DSL-ове употребљава се и назив „течни“ (енг. *fluent*) интерфејси, како би се назначила API језичка природа код DSL-а. Помоћу *fluent interface* (*smart* API) шаблона, API DSL интерфејса се трансформише у синтаксу која је природнија (читљивија) доменском кориснику, тако да се називи метода (активности над доменом) формирају према доменским терминима. Реализацију *fluent interface* шаблона могуће је урадити на различите начине, а најпознатији је уланчавање метода. Овај начин подразумијева увезивање позива метода, тако да се сваки сљедећи позив методе врши над резултатом извршавања претходно позване методе.

Манипулација синтаксним стаблом

Манипулација синтаксним стаблом представља шаблон помоћу којег се може имплементирати DSL на основу манипулације AST-а (*Abstract Syntax Tree* – апстрактно синтаксно стабло). Апстрактно синтаксно стабло представља структуру апстрактне синтаксе програмског кода. Под термином „апстрактна синтакса“ се подразумијева да она одређује исправност језичких исказа са становишта њихове структуре без улажења у детаље конкретне синтаксе. Основна идеја оваквог концепта јесте да се умјесто евалуације израза из програмског језика, уради парсирање његовог AST стабла, а затим се уради одговарајућа трансформација над стаблом како би се добила синтакса неког другог језика, а која се даље може процесирати [1]. Примјер оваквог концепта је трансформације HQL (*Hibernate Query Language*) упита у SQL упите.

Уграђивање типова

Уграђивање типова (ент. *typed embedding*) је шаблон са којим се код статички типизираних језика, коришћењем типова апстрахује доменски проблем, у циљу концизније синтаксе DSL-а. Ово се постиже помоћу: креирања класа које апстрахују проблеме из домена, коришћењем функција вишег реда, коришћењем имплицитне конверзије и коришћењем експлицитних ограничења помоћу типова. Функције вишега реду су функције које могу да прихватају функције као аргументе, и које могу да враћају функције као повратну вриједност. Користећи типове статичког језика, дефинишу се специфични типови који ће испољавати жељено доменско понашање и садржавати пословна правила [2].

Предности које се добијају када се користе „уграђени типови“, јесте што се имплицитно преузима конзистентност типова од језика у којем су они креирани [2]. Валидност „уграђених типова“, који чине DSL-ов систем типова, је аутоматски провјерена од стране компајлера статичког језика у току фазе компајлирања. *Scala* језик има слабије могућности метапрограмирања, али зато користећи могућности његовог система типова (као што су *type inferencing* и функције вишег реда), могуће је креирати концизне DSL-ове. Код *Scala* језика је могуће дефинисати апстрактне типове помоћу кључне ријечи *type*, а затим се они могу користити у генеричким слојевима апликације.

Метапрограмирање

Метапрограмирање је техника програмирања која припада генеричкој парадигми, и која укључује писање програма чији су поједини дијелови написани да би генерисали друге дијелове програмског кода [3, 4, 5]. Уопштена дефиниција је да се метапрограмирањем „пишу“ програми који „пишу“ (генеришу) друге програме или врше њихову трансформацију. Метапрограми третирају друге програме као податке, то значи да они могу да их учитавају, генеришу, анализирају, трансформишу, па чак и да мијењају властити програмски код. Ова особина да могу да мијењају властити програмски код у току извршавања, омогућава метапрограмима да рукују новим ситуацијама без новог компајлирања. Језици којима су написани метапрограми називају се метајезици, а језици који су манипулисани од стране метапрограма се називају атрибутско-оријентисани програмски језици. Уколико језик посједује особину да буде сам себи метајезик, таква особина назива се „рефлексивност“ [6].

Како би се могле добити информације (метаподаци) о објектима, код којих нам је циљ да их метапрограмирањем модификујемо у току извршавања програма, користи се интроспекција типова (енг. *type introspection*). Интроспекција типова представља могућност програма да испитује типове и својства објеката у току извршавања програма, као и то да се добије одговора на питање о којем типу објекта се ради и да ли је објекат изведен из неке родитељске класе. Рефлексија (енг. *reflection*) користи принципе из интроспекције типова, и ова техника представља могућност да се испитују и модификују структуре и понашања (вриједности, метаподаци, својства, функције) програма у току њиховог извршавања. Овим се, између осталог, омогућава позивање метода над објектима, креирање нових објеката и мијењање атрибута објеката без познавања назива поља у току компајлирања.

Метапрограмирање је реализовано помоћу техника: *reflective metaprogramming, runtime metaprogramming* и *compile-time metaprogramming* [5].

Метаобјект протокол код метапрограмирања

Метакласе су класе, чије су инстанце класе, а не објекти. Понашање инстанци метакласа је дефинисано датом метакласом. Сваки језик има властити метаобјект протокол који представља скуп правила која дефинишу интеракцију између објеката, класа и метакласа [7].

Метаобјекти су објекти који могу креирати инстанце, вршити њихову иницијализацију, мијењати њихове особине и понашања. Метаобјект протокол дефинише начин на који се метаобјекти понашају и на који начин се мијењају.

Примјер кориштења метаобјекта јесте концепт рефлексије. Код рефлексије се користе метаобјекти, како би се приступило интерној структури објеката, те да би се дата структура могла мијењати. На тај начин је могуће да програм сам себе мијења у току властитог извршавања. Метаобјект протокол је протокол за приступ и манипулацију структуре и понашања објекта. Основне функционалности које метаобјект протокол подразумијева су [8]:

- Креирање или брисање класа
- Креирање новог поља објекта или нове методе
- Мијењање структуре класе тако да насљеђује другу класу
- Генерисање или промјена програмског кода методе класе

Reflective metaprogramming

Reflective metaprogramming представља шаблон за реализацију DSL-а гдје се користи извршно окружење језика да се динамички проналазе поља или методе објеката које нису познате прије покретања програма [2]. Овај шаблон се примјењује код DSL имплементација које раде са динамичким објектима, чије структуре и методе нису познате прије покретања програма, па се позиви метода одлажу све до тренутка када се добију комплетне информације о динамичким објектима. Садржај динамичких објеката може бити дефинисан у конфигурационим фајловима или у неким другим изворима.

Дефинисањем функције вишег реда код *Groovy* језика, која у себи садржи одредницу (код *Groovy* језика она се реализује помоћу *with* методе) са којим објектом ће се разријешити контекст анонимног блока кода (тј. тијела прослијеђене анонимне функције), представља *reflective metaprogramming* концепт. Разрјешење контекста представља проналажење објеката над којима ће се позивати методе из анонимног блока кода. Уколико методе из анонимног блока кода нису референциране на одређене објекте, тада ће се дате методе рефлексијом потражити унутар објекта над којим је позвана *with* метода.

Runtime metaprogramming

Runtime metaprogramming је шаблон за реализацију DSL-а код којег се у току извршавања програма динамички генеришу објекти, или мијењају методе и атрибути постојећих објеката [2]. Генерисање или мијењање структуре и понашања објеката се постиже тако да програмери манипулишу компонентама инфраструктуре извршног окружења језика, односно помоћу метакласа или метаобјеката. Овим се постиже да DSL-ови буду што читљивији и да буду што природнији за кориштење од стране доменских корисника. Примјер оваквог типа метапрограмирања код *Groovy* програмског језика, јесте да се помоћу метода *propertyMissing* и *methodMissing* неког објекта, може дефинисати динамички дио програма, који ће се извршити када се детектује недостатак методе или поља у датом објекту. Исто тако, помоћу објекта *ExpandoMetaClass*, који је метакласа *Groovy* објеката, можемо додавати или мијењати методе и поља објеката у току извршавања, без промјене у изворном програмском коду.

Compile-time metaprogramming

Код *Compile-time metaprogramming* шаблона, компајлер генерише обиман програмски код на основу метаподатака, а у циљу да се код изворног програмског кода постигне концизна синтакса, те да се програмер ослободи писања *boilerplate* програмског кода [2]. *Boilerplate* је програмски код који се мало или уопште не мијења, а појављује се на више мјеста у апликацији. У односу на *runtime metaprogramming*, гдје извршно окружење језика генерише фрагменте програмског кода, код оваквог приступа се избјегава оптерећивање и успоравање извршног окружења, јер се сав посао обавља у фази компајлирања. Програмер користи специфичне описе у програмском језику, које компајлер тумачи и на основу њих трансформише програмски код. Овакав начин метапрограмирања се базира на анотацијама код *Java* и *Groovy* програмских језика.

3. ПРЕГЛЕД ЈЕЗИКА ЗА ИМПЛЕМЕНТАЦИЈУ ИНТЕРНИХ DSL-ОВА НА JVM ПЛАТФОРМИ

Java програмски језик и ламбда изрази

Java програмски језик је статички типизиран и објектно-оријентисан програмски језик опште намјене, посебно погодан за развој клијент-сервер, дистрибуираних и конкурентних апликација. Са појавом *Java* 8 верзије, уведене су могућности функционалног програмирања. Једна од таквих могућности су ламбда изрази, односно анонимне функције у *Java* језику.

Како би се постигла продуктивност, програмски код треба да буде концизан односно без *boilerplate* програмског кода. Ово се постиже употребом ламбда израза односно анонимних функција, јер представља начин да се смањи број линија програмског кода, те да програмски код буде лакши за разумијевање. Ламбда изрази представљају само синтаксно помоћно средство за лакше и елегантније писање, те читање унутрашњих анонимних класа [9]. Првенствена намјена ламбда израза је да дефинишу имплементацију функционалних интерфејса, односно интерфејса који имају само једну методу, те тиме елиминишу потребу за писањем анонимних класа у таквим случајевима. Овим је дата могућност функционалног програмирања за *Java* програмски језик, јер је могуће прослиједити анонимну функцију као аргумент другој функцији, а може се извршити и њена додјела варијабли. Основна синтакса за ламбда изразе је „*parameter -> statements*“, гдје *parameters* представља листу параметара, *statements* представља један или више исказа, а знак „->“ представља ламбда оператор. Важне карактеристике ламбда израза су [10]:

- Декларисање типова је опционо – нема потребе за декларисањем типова параметара јер компајлер може да закључи тип на основу вриједности параметара.
- Опционе су заграде код листе параметара уколико је један параметар у изразу.
- Уколико је један исказ у тијелу израза онда нема потребе за витичастим заградама.
- Уколико постоји један израз у тијелу ламбда израза који враћа повратну вриједност, онда нема потребе за *return* кључном ријечи. Витичасте заграде за тијело израза су тада обавезне уколико се враћа нека вриједност.

Scala програмски језик

Scala је модеран програмски језик који користи могућности функционалне и објектно-оријентисане парадигме [11]. Као програмски језик, *Scala* је дизајнирана како би се изразили стандардни програмски шаблони на концизан и елегантан начин, чиме би се преобиман *Java* програмски код и *boilerplate* програмски код, могао написати у *Scala* језику са знатно мањим бројем линија. *Scala* је статички типизиран и компајлерски језик. Компајлер *Scala* језика је заслужан за генерисање *boilerplate* програмског кода, на основу концизног програмског кода, у фази компајлирања. Назив *Scala* има значење „*Scalable Language*“ (срп. скалабилни језик), јер је дизајниран да се може проширивати са захтјевима корисника, као и да програми који су у *Scala* језику написани могу бити једноставни, али и веома комплексни [12]. Управо је спој функционалне и објектно-оријентисане парадигме главни разлог зашто је *Scala* скалабилна, јер дати спој омогућава креирања нових концепта који омогућавају изградњу великих и проширивих система.

Класа код Scala програмског језика

Као и код *Java* програмског језика, и у *Scala* језику класе су основа за инстанцирање објеката, које могу садржавати методе, варијабле, типове, објекте, *trait*s (програмска структура која је слична интерфејсима *Java* програмског језика – са подразумијеваним методама) и класе. Класа код *Scala* језика има доста синтаксних сличности, али и разлика у односу на имплементацију код *Java* језика. Наведене разлике најбоље се могу анализирати када се упореде имплементације у *Scala* и у *Java* језику на неком примјеру. Слиједе примјери имплементација апстракција књиге и аутора, које су приказане на сл. 1 (*Scala* језик) и сл. 2 (*Java* језик).

```
1  class Author(val name: String)
2  class Book (private var _title: String = "ScalaBook", val
   author: Author) {
3    def title = _title
4    def title_= (newTitle: String): Unit = _title = newTitle
5
6    def printAuthor(prefix: String = "Prefix") = {
7      var concat = prefix + author.name
8      println(concat)
9    }
10   override def toString: String = s"BookTitle: $title"
11 }
12
13 object MainApp {
14   def main(args: Array[String]) {
15     var book = new Book(author = new Author("ScalaAuthor"))
16     book printAuthor "Name of Author: "
17     println("Title of Book: " + book.title)
18     book.title = "NewTitle"
19     println(book)
20   }
21 }
```

Слика 1

Посматрајући имплементације у *Scala* и *Java* језику, уочава се мањи број линија програмског кода у верзији имплементираној *Scala* језиком. Такође, код имплементације класе у *Scala* језику, види се да је доста синтаксно чистија и концизнија, јер нема потребе за мноштво витичастих заграда за означавање блокова кода са једним изразом. Нема потребе за навођењем типова варијабли и повратних типова метода, те нема потребе за означавањем краја исказа навођењем симбола „;“. Аутоматско закључивање типова, односно *type inference* је омогућено захваљујући компајлеру *Scala* језика, који закључује повратни тип метода на основу задњег исказа код метода, и који може закључити тип варијабле на основу њене вриједности. *Type inference* није омогућен код закључивања типова параметара метода (осим код параметара анонимних функција), и није омогућен код закључивања типова параметара рекурзивних функција. Код дефинисања варијабли или поља, довољно је навести *val* или *var* испред идентификатора варијабле или поља. Са *val* се означавају варијабле које не мијењају вриједност приликом иницијализације, односно ради се о *immutable* варијаблама. Док се са *var* означавају варијабле које могу мијењати вриједности.

Наведене особине синтаксе *Scala* језика омогућавају концизно писање програма, што је кључно приликом реализација DSL језика. Још неке важније особине *Scala* језика су:

- Дефиниција класе обједињује дефинисање атрибута и конструктура. Уколико се не наведе експлицитно, класе су подразумијевано *public*, као и њени атрибути који су означени са *val* или *var*. Код дефиниције класе *Author* на сл. 1 потребна је само једна линија програмског кода, и представља концизан начин дефиниције класе. Уколико се не наведе у конструктору ни *val* ни *var* за поља, онда су поља дефинисана као приватна, и видљива су само унутар класе.

```
1  public class Author {
2      public final String name;
3      public Author(String name) {
4          this.name = name;
5      }
6  }
7  public class Book {
8      private String title;
9      public final Author author;
10
11     public Book(String title, Author author) {
12         this.title = title;
13         this.author = author;
14     }
15     public void printAuthor(String prefix) {
16         String concat = prefix + this.author.name;
17         System.out.println(concat);
18     }
19     public String toString() {
20         return "BookTitle: " + this.title;
21     }
22     public String getTitle() {
23         return title;
24     }
25     public void setTitle(String title) {
26         this.title = title;
27     }
28     public static void main(String[] args) {
29         Book book = new Book("JavaBook", new
   Author("JavaAuthor"));
30         book.printAuthor("Name of Author: ");
31         System.out.println("Title of Book: " + book.
   getTitle());
32         book.setTitle("NewTitle");
33         System.out.println(book);
34     }
35 }
```

Слика 2

- Декларисање типа варијабле се наводи послије идентификатора варијабле, а повратни тип методе се наводи послије листе параметара.
- Конструктори и методе могу имати именоване аргументе, а који могу имати подразумијеване вриједност, овакве могућности изостају код *Java* језика.
- Дефинисање метода или функција се реализује са *def* кључном ријечи, и није потребно користити *return* кључну ријеч уколико се узима вриједност задњег израз као повратна вриједност. Оно што се може специфично рећи за методе у *Scala* језику, јесте да нема *static* кључне ријечи. Да би метода била статичка она мора бити дефинисана у *Scala singleton* објекту, а који се дефинише са кључном ријечи *object*. Именовање метода је флексибилно, и за њихове називе могу се користити карактери који се користе као оператори, те се на овај начин може симулирати преклапање оператора. У *Scala* језику је могуће дефинисати локалне функције, које су видљиве само унутар неке функције.
- Подразумијевани увози (енг. *import*) у класама су реализовани користећи *Predef* објекат, који садржи помоћне методе. *Predef* објекат се подразумијевано увози код сваке јединице за компајлирање. Тиме се методе унутар *Predef* објекта могу користити у свакој класи без њиховог експлицитног квалификовања, као што је нпр. *println* метода.
- У *Scala* језику сви типови су објекти, нема примитивних типова. Свака инстанца класе у *Scala* језику подразумијевано је подтип типа *AnyRef*, овај тип је еквивалент *Object* типу у *Java* језику. *AnyRef* је подтип *Any* типа, а разлог за овакву хијерархију је да би *Any* обухватао све типове, односно, обухвата и оне типове које представљају примитивне типове у *Java* језику, а у *Scala* језику су заправо објекти (*Double*, *Int*).

Case класа код Scala програмског језика

Case класа је попут обичне *Scala* класе која је *immutable*, међутим има неколико кључних разлика. *Case* класа се може посматрати као објектно-оријентисано проширење *enum* концепта [11]. Како се *case* класа може инстанцирати без кључне ријечи *new*, она се може посматрати као параметризовани *enum* или нека форма вриједносних типова. Када се *case* класа дефинише, приликом њеног компајлирања генерише се мноштво метода које нису дефинисане у класи. Тако је генерисана и метода *applay()*, која има улогу *factory* методе за инстаницарање објекта класе – она се имплицитно позива када се не наведе кључна ријеч *new.* Неке од генерисаних метода може користити програмер, а неке су генерисане како би биле од користи окружењу у којем се извршава програм. Аутоматски генерисане методе које програмер може користити су: *get* методе, *toString()*, *hashCode()* и *equals()* методе. Са овим генерисањем се смањује *boilerplate* код, као и могућност прављења грешака приликом развоја, јер наведене методе нису дефинисане у *source* коду, а њихово генерисање се подразумијева код *case* класе. Резултат наведеног је, да се само са једном линијом дефиниције *case* класе (сл. 3) може „уштедити“ писање неколико десетина линија програмског кода које је потребно написати у *Java* језику или генерисати помоћу развојног окружења.

```
1  case class AuthorCase(val name: String)
```

Слика 3

Имплицитна конверзија

Scala језик има опцију проширивања постојећих класа помоћу имплицитне конверзије. Како су у *Scala* језику сви типови објекти, могуће је и за типове који су у *Java* језику примитивни, као што су нпр. *int* или *double*, додавати методе или поља која ће бити од користи приликом реализације концизног DSL-а. Основна предност имплицитне конверзије код *Scala* језика јесте што је лексички ограничена, односно додана понашања ће бити доступна само ако се експлицитно наведе у одређеном лексичком опсегу. Овим се контролише доступност датих промјена, и спрјечава да дође до проблема некомпатибилности дате измјењене класе у другим дјеловима апликације. Имплицитном конверзијом се реализује концепт отворених класа.

Анонимне функције код Scala програмског језика

Како је *Scala* функционалан језик, могуће је функције третирати као податке, односно смјештати их у варијабле, прослијеђивати их као аргументе других функција, и да буду повратна вриједност других функција. Ово је могуће у *Scala* језику коришћењем анонимних функција (зову се још и функцијски литерали). Код анонимних функција симбол „=>“ одваја листу параметара функције од њене имплементације

Groovy програмски језик

Groovy је, опционо типизиран и припада динамичким језицима, са могућностима статичког типизирања и компајлирања [13]. Намијењен је за *Java* платформу да би се повећала продуктивности *Java* програмера. Ово је могуће због тога што је *Groovy* концизан, синтаксно познат *Java* програмерима и веома експресиван језик. Захваљујући добро развијеном *runtime* и *compile-time* метапрограмирању, *Groovy* је прикладан за креирање DSL-а. У *Groovy* језику анонимне функције се називају *closures*, и оне представљају основни дио функционалне парадигме код *Groovy* језика. *Java* програмски код се може користити унутар *Groovy* програмског кода, као да се ради о легалном *Groovy* програмском коду. Ово је посљедица компатибилности синтаксе *Java* језика са *Groovy* синтаксом, тако да су *while*/*for* петља, *if*/*else* конструкција и *switch* наредба из *Java* језика потпуно легалне у *Groovy* језику. Коришћење *Groovy* програмског кода унутар *Java* језика је могуће захваљујући библиотекама и класама које омогућавају овакву комуникацију. У *Groovy* језику олакшан је рад са колекцијама, регуларним изразима, као и са процесирањима за *XML*, *JSON* и *HTML* структуре, јер нису потребне посебне библиотеке за манипулацију овим форматима, већ је подршка уграђена у сам језик.

Класа код Groovy програмског језика

Синтакса *Groovy* класе је веома слична *Java* класи, уз неке олакшице као што су подразумијевано генерисања конструктора и *property* метода. Подразумијевани модификатор за приступ класама, методама и варијаблама је *public*. Слично као и код *Scala* језика, у циљу постизања концизне синтаксе, симбол „;“ за крај исказа је опционан. Ово не важи ако се више исказа налази у једној линији. Опционе су заграде за листу аргумената код позива метода, али само код метода које садрже бар један параметар и када не постоји могућности за двосмисленост параметара. Кључна ријеч *return* која враћа објекат или вриједност из методе је опциона, *Groovy* ће аутоматски вратити резултат задњег евалуираног израза. Све наведене особине се могу искористити за реализацију концизног DSL језика. Карактеристике *Groovy* класе, али и самог језика, показане су, такође, на примјеру апстракције књиге и аутора. Из примјера датог на сл. 4 може се уочити сљедеће:

- Могуће је дефинисати, као и код *Java* језика, нивое приступа као што су: *public*, *private* и *protected*. Уколико се не наведе модификатор приступа за поља класе, она су тада приватна и аутоматски се генеришу припадајуће *get* и *set* методе за дата поља. Овим се уклања *boilerplate* код за писање појединачних *get* и *set* метода.
- Аутоматски се генеришу све комбинације конструктора за поља дате класе, чиме се доста редукује *boilerplate* код за писање појединачних конструктора.
- Како је *Groovy* динамички типизиран језик, довољно је варијаблу или методу означити са *def*, те на тај начин није потребно наводити тип варијабле или повратни тип метода. Овим је дозвољено да се тип мијења у току извршавања програма. *Groovy* је и статички типизиран језик, па је могуће декларисати тип варијабле или повратни тип методе, а који остаје фиксан након компајлирања.
- Методе и конструктори могу имати именоване аргументе, који могу имати подразумијеване вриједности.
- *Groovy* има велики број анотација којима је могуће редуковати *boilerplate* код, и искористити *compile-time* метапрограмирање. На примјер, ако се класа означи анотацијом *@Immutable*, аутоматски ће сва поља класе бити означена *final* модификатором и неће се моћи мијењати након додјеле вриједности. Аутоматско генерисање *equals* и *hashCode* метода је омогућено помоћу *@EqualsAndHashCode* анотације. Генерисање *toString* методе је омогућено помоћу *@ToString* анотације.
- Уколико је потребна провјера да ли је објекат правилно референциран, те како би се избјегао *NullPointerException* изузетак приликом приступа таквом објекту, користи се *safe-dereference* оператор са нотацијом „?.“. Овакав оператор је уграђен и код појединих функција вишег реда. То важи нпр. за *max()* функцију код колекција, гдје се не треба провјеравати да ли је колекција правилно референцирана прије позива *max()* функције.
- Елвис оператор са нотацијом „?:“ омогућава да се концизно напише *if\else* конструкција, приликом чега се дефинише подразумијевана вриједност за *else* блок. Када је потребно провјерити да неки објекат нема *null* вриједност, користићемо овакву концизну нотацију. Уколико објекат нема *null* вриједност, резултат израза ће бити дати објекат који се испитује. У другом случају, када објекат има *null* вриједност, вратиће се дата подразумијевана вриједност коју смо дефинисали код оваквог израза.

Анонимне функције код Groovy програмског језика

Анонимне функције (познате и као *closures*), представљају отворени, анонимни блок кода који може имати аргументе, повратну вриједност и који се може додијелити варијабли [13]. *Closure* може референцирати варијабле које су декларисане у њеном окружењу, али може садржавати и слободне варијабле које су дефинисане изван њеног опсега. Основна синтакса за *closure* јесте: „{ [*closureParameters* ->] *statements* }“, гдје је „[*closureParameters*->]“ опциона листа параметара који су раздвојени запетама. Под *statements* се подразумијева нула или више *Groovy* исказа. C*losure* има важну улогу како би пружио један дио функционалне парадигме у *Groovy* програмском језику, јер се функције третирају као подаци и могу се прослијеђивати као аргумент другим функцијама вишег реда.

```
1  class Author {
2      String name
3  }
4  class Book {
5      private String title
6      Author author
7
8      def printAuthor(authorPrefix = 'Author Info: ') {
9          System.out.println(authorPrefix + author.name)
10     }
11     void printTitle(params) {
12         System.out.println(params.titlePrefix + " " +
   title)
13     }
14     static void main(String[] args) {
15         def book = "BookDynamic"
16         System.out.println("Dynamic typing for Book: " +
   book)
17         book = new Book()
18         book.title = "Book1"
19         book.author = new Author(name: "Author1")
20         System.out.println("Book title: " + book.title +
21 ", author: " +  book.author.name)
           book = new Book(author: new Author(name:
22 "Author4"))
23         System.out.println("Book title: ${book.
24 title?:"NoTitle"}, author: ${book.author?.name}")
25         book.printAuthor()
26         book.printAuthor "Info about Author: "
27         book.title = "Title2"
28         book.printTitle titlePrefix:"Book Title: "
29     }
30 }
```

Слика 4

Runtime metaprogramming код Groovy програмског језика

За сваки позив методе, *Groovy* извршно окружење провјерава о којем се типу објекта ради (да ли је изворно писан у *Groovy* или *Java* језику) и на основу тога се покреће одређени механизам делегирања позива методе. Сваки објекат који је писан у *Groovy* језику, имплементира *groovy.lang.GroovyObject* интерфејс, и то подразумијевано са *groovy.lang.GroovyObjectSupport* класом, која омогућава пренос позива метода према *groovy.lang.MetaClass* објекту. Методе које *GroovyObject* садржи (сл. 5) су:

- *invokeMethod* – ова метода је првенствено намијењена да се користи у спрези са *GroovyInterceptable* интерфејсом или са *MetaClass*-ом објекта, када је потребно извршити пресретање свих позива метода. Она се позива и када није присутна метода која се позива у датом објекту, међутим не препоручује се да се за то користи, и за такву ситуацију треба користити *methodMissing* методу.
- *getProperty* – свако читање стања *property* поља може се редефинисати имплементацијом ове метода.
- *setProperty* – свако мијењање стања *property* поља може се редефинисати имплементацијом ове метода.
- *getMetaClass* – повратна вриједност ове методе је метакласа припадајућег објекта.
- *setMetaClass* – помоћу ове методе може се поставити властита имплементација *MetaClass* интерфејса, чиме се могу додавати или мијењати поља и методе објекта коме припада *MetaClass*-а.

```
1  class BookMetaClass {
2      String title
3      Integer numberOfPages
4
5      def invokeMethod(String name, Object args) {
6          return "Method invocaton with name: $name and
   parametar: $args"
7      }
8      def getProperty(String name) {
9          if (name != 'naslov')
10             return metaClass.getProperty(this, name)
11         else
12             return title
12     }
13     void setProperty(String name, Object value) {
14         this.@"$name" = "Value $value is overridden"
15     }
16 }
17
18 def bookMetaObject = new BookMetaClass(title:
   "Book1", numberOfPages: 100)
19 println(bookMetaObject.someMethod("Test", 5))
20 println("Getter of title on Serbian: $bookMetaObject.
   naslov")
21 println("Regular getter: $bookMetaObject.
   numberOfPages")
23 bookMetaObject.title = "NewBook"
24 println(bookMetaObject.title)
25 println("Getting title without getters: "
26         + bookMetaObject.metaClass.
   getAttribute(bookMetaObject, 'title'))
27 bookMetaObject.metaClass.setAttribute(bookMetaObject,
   'title', 'Book2')
28 println("Setting title without setters, new title is:
   $bookMetaObject.title")
```

Слика 5

4. ПРИМЈЕРИ РЕАЛИЗАЦИЈЕ DSL-А ЗА ОБРАДУ СЛИКЕ

Наведене особине JVM језика и поједини шаблони за реализацију DSL-а су примјењени у процесу развоја DSL-а за обраду слике[1]. DSL за обраду слике реализован је помоћу *Scala*, *Groovy* и *Java* језика, а користи *Scrimag*e [14] библиотеку која садржи имплементиране операције за обраду слике. У наставку ове секције издвојени су поједини дијелови програмског кода DSL-а за обраду слике у сврху демонстрирања принципа развоја DSL-а, те концизних особина појединих језика које су описане у претходним секцијама. Реализовани DSL-ови су првенствено намијењени само како би се демонстрирали принципи за развоја DSL-а, као и поједине особине језика на практичном примјеру, те није био циљ да се детаљно улази у рјешавање проблема обраде слике. Поред наведених циљева, сврха приказаних DSL-ова је и да покаже њихове међусобне синтаксне разлике које су посљедица реализације у различитим језицима (*Scala*, *Groovy* и *Java*). На основу датих реализација може се извући закључак у којим језицима је лакше и „природније“ за кориснике имплементирати интерни DSL.

Имплементирани DSL обухвата основне операције над сликама: учитавање слике у одређеним форматима, експортовање слике у одређеним форматима, скалирање слике, процесирање блока (низа) операција над сликом, ротирање слике, обртање слике око оса, транслацију слике, аутоматско одсијецање „вишка“ слике по специфицираној боји, одсијецање дијела слике у зависности од фактора скалирања, одсијецање дијела слике за одређени број пиксела, те употреба одређених филтера над сликом.

Реализација DSL-а помоћу Scala језика

Приликом реализација DSL-а помоћу *Scala* језика кориштене су функције вишег реда. Тијело анонимне функције, које се прослеђује функцији вишег реда (сл. 6, линија 24), користи се за реализацију блока операција над сликом (сл. 8 (а), линија 7).

Затим, кориштена је имплицитна конверзија (сл. 7, линија 21) која омогућава проширење *Int* типа методама које DSL-у дају природнији изглед (*cm* и *px* методе). Метода *cm* конвертује прослијеђени параметар (децимални број) у број пиксела, како би метода *scaleTo* (сл. 8 (а), линија 3) примила валидне параметре за скалирање слике. Метода *px* враћа прослијеђени параметар без промјене. Исти принцип проширивања са методама код већ постојећег типа кориштен је и код *Double* типа.

Такође, на сл. 6 (линија 29) приказана је симулација преклапања оператора „+“, захваљујући могућности именовања метода у *Scala* језику помоћу специјалних карактера.

1 Комплетан пројекат се може погледати *online* на *https://github.com/milos-serbic/ImageProcessing*

```
1  class ImageDSLScala(width: Int = 200, height: Int = 200){
2  ...
3  def apply (format: Format = jpg, compression: Int = 0) = {
4  format label match {
5    case png.label => compression match {
6        case 100 => writer = PngWriter MaxCompression
7        case 0 => writer = PngWriter NoCompression
8        case _ =>
9        {
10         val comTmp = math.ceil(compression / 10.toDouble).
   toInt
11         if(comTmp > 9)
12           writer = PngWriter(9)
13         else
14           writer = PngWriter(comTmp)
15       }
16     }
17   case jpg.label => compression match {...}
18   ...
19   case _ => writer = JpegWriter NoCompression
20 }
21 this
22 }
23 ...
24 def processing(processing:() => ImageDSLScala) = {
25   processing()
26   this
27  }
28 ...
29 def +(image: ImageDSLScala) = {
30   imageObj = imageObj composite(new AlphaComposite(0.5f),
   image getImageObj)
31   this
32  }
33 ...
34 }
35 object ImageDSLScala {
36 ...
37   def apply (format: Format = jpg) = {
38     new ImageDSLScala apply(format, 0)
39   }
40 ...
41 }
```

Слика 6

Приликом реализације вишеструке примјене филтера (сл. 8 (а), линија 23), кориштена је локална функција *times* (сл. 7, линија 13) у циљу рекрузивног позивања операција за обраду слике које се просљеђују у тијелу анонимне функције.

Уколико се дефинише *apply* метода (сл. 6, линија 37) унутар *singleton* објекта, са циљем како би DSL корисник могао да конструише објекат, тада ће DSL корисник моћи користити конструктор (имплицитно се позива *apply* метода) датог DSL објекта (*ImageDSLScala*) без кључне ријечи *new*. Овим се постиже концизна синтакса приликом конструкције DSL објекта (сл. 8 (а), линија 3), који омогућава интерфејс за рад са сликом.

Примијењен је *smart* API шаблон на начин да свака метода враћа властити објекат из којег се позива, како би се увезали позиви метода. Кориштењем апстрактног типа помоћу кључне ријечи *type* (сл. 7, линија 28), примијењен је шаблон уграђивања типова са којим се постиже параметризација типа код *NumberSyntax* конструкције. Шаблони *smart* API и уграђивање типова су описани у секцији 2. Поједини дијелови програмског кода су изостављени ради поједностављеног приказа.

```
1  object UtilitiesScala {
2  ...
3  type SyntaxInt = Int
4  class IntegerSyntax (syntax: SyntaxInt) extends
   NumberCm(syntax: SyntaxInt) with  NumberSyntax {
5      type T = SyntaxInt
6      def pct = syntax
7      def px = syntax
8      def scaleFactor = syntax
9      def inverseScaleFactor = 1/syntax.toDouble
10
11     def times(operation: => Unit): Unit = {
12      def times(syntax: Int, operation: => Unit):
   Unit = {
13       operation
14       if (syntax > 1)
15        times(syntax-1, operation)
16      }
17     times(syntax, operation)
18     }
19 }
20 implicit def IntSyntax(syntax: SyntaxInt) = new
   IntegerSyntax(syntax)
21 ...
22 abstract class NumberCm(syntax: Double) {
23   def cm: Int = new BigDecimal(syntax *
   37.795275590551).setScale(1, BigDecimal.ROUND_HALF_
   UP).intValue()
24 }
25 trait NumberSyntax {
26   type T
27   def scaleFactor: T
28   ...
29 }
30 ...
31 }
```

Слика 7

На слици 8 (б) могу се погледати појединачне операције DSL-а које су реализоване према претходно наведеним принципима за *Scala* језик.

Реализација DSL-а помоћу Groovy језика

Код реализације DSL-а помоћи *Groovy* језика, кориштен је *category* концепт из датог језика. Ово је заправо концепт отворених класа, са којим се проширује постојећа *Integer* класа (сл. 9, линија 3), тако да се на приказаном примјеру (сл. 11 (а), линија 7) могу видјети позиви методе *cm* над цјелобројним вриједностима. Такође, *category* концепт је примијењен и над *Double* класом (сл. 9, линија 12).

```
1  import utilitiesScala.UtilitiesScala._
2  //primjer poziva operacija u jednoj liniji
3  var palm = ImageDSLScala(png) importFrom "palm.jpg"
4  scaleTo(7.9375 cm, 7.9375 cm)
5  //primjer poziva operacija u bloku
6  var night = ImageDSLScala(png)
7  night.processing(() => {
8    night importFrom "night.jpg"
9    night scaleToHeight 800
10   night filterWith FilterEnum.GlowFilter
11   night rotate right
12   night exportTo "nightLambdaBlock.png"
13 })
14 //primjer kompozicije dvije slike pomocu preklopljenog
   operatora +
15 var see = ImageDSLScala(png) importFrom "see.jpg"
16 var rock = ImageDSLScala(png) importFrom "rock.jpg"
17 see + rock exportTo "seeRock.png"
18 //primjer visestruke primjene filtera nad slikom
19 run = ImageDSLScala(png) importFrom "run.jpg"
20 var amount = 0.2f
21 var exposure = 0.3f
22 var counter = 0
23 5 times {
24   counter+=1
25   run filterWith(() => new ChromeFilter(amount + 0.5f,
26 exposure + 1.0f)) exportTo s"runFilterAdjust${counter}.png"
27 }
```

Слика 8 (а)

```
1  var run = ImageDSLScala(png) importFrom "run.jpg"
2  var rock = ImageDSLScala(png) importFrom "rock.jpg"
3  //rotacija na lijevo
4  run rotate left exportTo "runRotateLeft.png"
5  //obrtanje oko x ose
6  run flip horizontally exportTo "runFlipHorizontally.
   png"
7  //transliranje
8  run translateBy(x=50 px, y=50 px) background Color.
   White exportTo "runTranslate.png"
9  //automatsko odsjecanje prema zadanoj boji
10 run autocrop(background=white) exportTo "runAutocrop.
   png"
11 //kropovanje slike
12 run cropBy(2 scaleFactor) exportTo "runCrop.png"
13 run cropBy(4 inverseScaleFactor) exportTo "runCropInv.
   png"
14 //kompozicija slike sa drugom slikom za odredjeni
15 postotak
16 run add (rock, compositType=alpha, compositPct=40 pct)
   exportTo "runRock.png"
17 //primjena filtera
18 run filterWith FilterEnum.PixelateFilter exportTo "run-
   FilterPixelate.png"
```

Слика 8 (б)

```
1  trait NumberCm {
2  ...
3  static Integer getCm(Integer self){
4          new BigDecimal(self * 37.795275590551).
5  setScale(1, BigDecimal.ROUND_HALF_UP).intValue()
6   }
7  }
8  class IntegerCategory implements NumberCm {
9      static Integer getPx(Integer self){ self }
10     ...
11 }
12 class DoubleCategory extends IntegerCategory
   implements NumberCm, PrintValue{
13     static Double getScaleFactor(Double self){ 1/self.
   toDouble()}
14 }
```

Слика 9

Код *DoubleCategory* класе се може видјети како је код *Groovy* језика могуће реализовати неки вид вишеструког насљеђивања, јер се имплементира више *traits* конструкција. *Traits* су структуралне конструкције које се могу посматрати као интерфејси, а који могу да садрже подразумијеване имплементације (методе) и стања (поља).

Приликом реализације DSL-а помоћу *Groovy* језика кориштен је *runtime metaprogramming* шаблон (детаљније описан у секцији 2). Помоћу овог принципа проширивана је *ImageDSLGroovy* објекат у току извршавања програма (сл. 11 (а), линија 7) пољима *x* и *y* која нису дефинисана у изворном програмском коду. Додавање нових поља је реализовано редефинисањем *propertyMissing* методе (сл. 10, линија 11).

Код сваког позива методе из *Groovy* програмског кода, *Groovy* извршно окружење ће пронаћи *MetaClass*-у датог објекта и њој ће се делегирати позив методе. *MetaClass*-а је интерфејс који се састоји из два дијела. Први дио је клијентски API и он је дефинисан *MetaObjectProtocol* интерфејсом, а који *MetaClass*-а насљеђује. Други дио представљају методе из *MetaClass* интерфејса које користи *Groovy* компајлер и извршно окружење. Подразумијевана имплементација *MetaClass* интерфејса код сваког објекта је *MetaClassImpl*, која имплементира подразумијевано проналажење метода. Овакво проналажење метода подразумијева претрагу међу обичним методама класе објекта, а уколико се не пронађе ниједна метода на овај начин, позваће се редефинисана *methodMissing* метода. Уколико није редефинисана *methodMissing* метода, позваће се имплементирана метода *invokeMethod* из *GroovyObject* интерфејса. Поред подразумијеване имплементације *MetaClass* интерфејса, *Groovy* језик посједује и посебну *MetaClass*-у која се зове *ExpandoMetaClass*. Она је посебна зато што омогућава да се њеним посредовањем, динамички додају или мијењају методе, конструктори, поља и статичке методе класе којој припада *ExpandoMetaClass* инстанца. *Groovy* сваку инстанцу *java.lang.Class* класе, обезбјеђује са посебним пољем *metaClass* које ће омогућити приступ референци *ExpandoMetaClass* инстанце. Користећи ову *ExpandoMetaClass* инстанцу, могу се додавати или мијењати метода и поља дате класе. *ExpandoMetaClass* код реализације DSL-а (сл. 10, линија 12) користи се за динамичко додавање нових поља *ImageDSLGroovy* инстанци. Како би *ImageDSLGroovy* могао користити *ExpandoMetaClass* инстанцу, потребно је да се она иницијализује унутар *ImageDSLGroovy* конструктора (сл. 10, линија 5).

Када је потребно приступити пољима која се динамички додају у току извршавања, користи се *getProperty* метода (припада *GroovyObject* интерфејсу) којој се прослеђује назив динамички доданог поља (сл. 10, линија 33).

Код реализације DSL-а, урађено је редефинисање методе *methodMissing* (сл. 10, линија 15), те је и на овај начин кориштен *runtime metaprogramming* шаблона. Овим је одређено понашање програма када се позива метода која није дефинисана унутар *ImageDSLGroovy* објекта, а чији је назив „rotiraj“ (сл. 11 (а), линија 16).

Такође, код реализације DSL-а приказано је кориштење *closure* (анонимне функције) код *translate* методе (сл. 11 (а), линија 6), а имплементација је дата на сл. 10 (линија 28). Кориштењем *with* методе унутар тијела *translate* методе, примијењио се *reflective metaprogramming* концепт који је описан у секцији 2. У суштини, овдје *with* метода клонира *closure* објекат који јој је прослијеђен. Затим, *delegate* пољу (клонираног *closure* објекта) додјељује се референца одређеног објекта, који је у овом случају тренутни *ImageDSLGroovy* објекат из којег се *translate* метода позива (ово је дефинисано са *this.with*). Уколико *closure* не може разријешити позив методе (метода која се позива унутар *closure* је без референце на објекат), тада ће се делегирати позив методе на тренутни *ImageDSLGroovy* објекат.

Код реализација DSL-а кориштено је преклапања оператора „+“ (сл. 10, линија 20), у циљу реализације оператора за композицију слика (сл. 11 (а), линија 23).

Приликом реализација DSL-а кориштен је и *smart* API шаблон, гдје све методе које се користе за операције над сликом враћају властити објекат, како би се увезивали позиви метода.

На слици 11 (б) дате су појединачне операције DSL-а које су реализоване према претходно наведеним принципима за *Groovy* језик.

```
class ImageDSLGroovy {
...
ImageDSLGroovy (String format = "jpg", int compression = 0, String fullPath = "") {
    def mc = new ExpandoMetaClass(ImageDSLGroovy, false, true)
    mc.initialize()
    this.metaClass = mc
    ...
 }
...
def propertyMissing(String name, value) {
    this.metaClass."$name" = value
 }
...
def methodMissing(String name, args) {
    if(name == "rotiraj")
        this.rotate args[0]
}
...
ImageDSLGroovy plus(ImageDSLGroovy image) {
    imageObj = imageObj.composite new AlphaComposite(0.5), image.imageObj
    this
 }
...
def processing(closure) {
    this.with closure
}
def translate(closure) {
    this.with closure
 }
def background(params) {
    if(params.color == "black")
        imageObj = imageObj.translate this.getProperty("x"), this.getProperty("y"), Color$.MODULE$.Black()
    else if(params.color == "white")
        imageObj = imageObj.translate this.getProperty("x"), this.getProperty("y"), Color$.MODULE$.White()
    this
 }
...
}
```

Слика 10

```
//primjer poziva operacija u jednoj liniji
def flower = imageDSLGroovy format:png importing from:"flower.jpg"
def night = imageDSLGroovy format:png
//primjer poziva operacija u bloku
use(IntegerCategory) {
 flower.translate {
    x = 50.cm
    y = 50.cm
    background color:white
    exporting to:"flowerTranslate.jpg"
 }
 night.processing {
    importing from:"night.jpg"
    scale height:800.px
    filterWith FilterEnum.GlowFilter
    rotiraj desno
    exporting to:"nightClosure.png"
 }
}
//primjer kompozicije dvije slike pomocu preklopljenog operatora +
def rock = imageDSLGroovy format:png importing from:"rock.jpg"
def see = imageDSLGroovy format:png importing from:"see.jpg"
(see + rock).exporting to:"seeRock.png"
```

Слика 11 (а)

```
def run = imageDSLGroovy format:png importing from:"run.jpg"
def rock = imageDSLGroovy format:png importing from:"rock.jpg"
//rotacija na desno
run.rotate right exporting to:"runRotateRight.png"
//obrtanje oko y ose
run.flip vertically exporting to:"runFlipVertically.png"
//automatsko odsjecanje prema zadanoj boji
run.autocrop background:white exporting to:"runAutocrop.png"
//kropovanje slike
run.cropBy 2.scaleFactor exporting to:"runCrop.png"
run.cropBy 4.3d.inverseScaleFactor exporting to:"runCropInv.png"
//kompozicija slike sa drugom slikom za odredjeni postotak
run.add {
    imageForCompostion = rock
    compositType = alpha
    compositPct = 40.pct
    adding
    exporting to:"runRock.png"
}
//primjena filtera
run.filterWith FilterEnum.NoiseFilter exporting to:"runFilterNoise.png"
```

Слика 11 (б)

Реализација DSL-а помоћу Java језика

Код реализације DSL-а помоћу *Java* језика, приказано је коришћење ламбда израза у циљу концизнијег записа операција за обраду слике. Како би се реализовао блок наредби (сл. 12, линија 15) у *Java* језику, потребно је користити одговарајући функционални интерфејс. У овом случају то ће бити *Consumer* функционални интерфејс. Дати интерфејс садржи методу *accept* која може да прима један параметар, а који одређује тип објекта над којим ће се извршавати операције из блока наредби. У овом случају параметар је типа *ImageDSLJava* (сл. 12, линија 3). Код реализације DSL-а помоћу *Java* језика, примијењен је и *smart* API шаблон, како би се увезали позиви метода (сл. 12, линија 11). Имплементација методе *processing*, као и коришћење датог DSL-а је приказано на сл. 12.

```
public class ImageDSLJava {
...
public ImageDSLJava processing(Consumer<ImageDSLJava> consumer){
    consumer.accept(this);
    return this;
 }
...
}
//primjer poziva operacija u jednoj liniji
ImageDSLJava flower = imageDSLJava(png).
importFrom("flower.jpg").scaleToHeight(4).
exportTo("flower.png");
//primjer poziva operacija u bloku
ImageDSLJava night = imageDSLJava(png);
night.processing(img -> {
    img.importFrom("night.jpg");
    img.scaleToHeight(800);
    img.filterWith(FilterEnum.GlowFilter);
    img.rotate(right);
    img.exportTo("nightLambdaBlock.png");
});
```

Слика 12

На сл. 13 могу се погледати појединачне операције DSL-а које су реализоване према претходно наведеним принципима за *Java* језик. Када би користили само „чисту“ *Scrimag*e библиотеку, без коришћења наведених принципа, позиви појединачних операција (у *Scala* језику) би изгледале као што су приказане на сл. 14. Уколико би се упоредиле приказане операције на сл. 13, са операцијама на сл. 14, може се примјетити да је коришћење *Scrimag*e библиотеке у програмском језику *Scala* веома слично са реализованим DSL-ом у *Java* језику. Ово је резултат изостанка концизне синтаксе код *Java* језика, као и изостанка примјене многих концепта и особина које омогућавају концизну синтаксу у *Scala* језику. Једина већа разлика употребе DSL-а код *Java* језика у односу на „чисту“ *Scrimag*e библиотеку, је у томе што се могу користити позиви операција у блоку програмског кода (сл. 12, линија 15), помоћу ламбда израза.

```
1  ImageDSLJava run = imageDSLJava(png).importFrom("run.
   jpg");
2  ImageDSLJava rock = imageDSLJava(png).
   importFrom("rock.jpg");
3  //rotacija na lijevo
4  run.rotate(left).exportTo("runRotateLeft.png");
5  //obrtanje oko x ose
6  run.flip(horizontally).exportTo("runFlipHorizontally.
   png");
7  //transliranje
8  run.translate(50, 50, white).exportTo("runTranslate.
   png");
9  //automatsko odsjecanje prema zadanoj boji
10 run.autocrop(white).exportTo("runAutocrop.png");
11 //kropovanje slike
12 run.cropBy(2.0).exportTo("runCrop.png");
13 run.cropBy(0.25).exportTo("runCropInv.png");
14 //kompozicija slike sa drugom slikom za odredjeni
   postotak
15 run.composing(rock, alpha, 40).exportTo("runRock.
   png");
16 //primjena filtera
17 run.filterWith(FilterEnum.GlowFilter).
   exportTo("runFilterGlow.png");
```

Слика 13

```
1  implicit val writer = PngWriter.NoCompression
2  var run = Image.fromFile(new File("input/run.jpg"))
3  var rock = Image.fromFile(new File("input/rock.jpg"))
4  //skaliranje
5  run.scaleTo(200, 200).output("output/runScaled.png")
6  (writer)
7  //rotacija na lijevo
8  run.rotateLeft.output("output/runRotateLeft.png")
9  (writer)
10 //obrtanje oko x ose
11 run.flipX.output("output/runFlipHorizontally.png")
12 (writer)
13 //transliranje
14 run.translate(50, 50, Color.White).output("output/
15 runTranslate.png")(writer)
16 //automatsko odsjecanje prema zadanoj boji
17 run.autocrop(Color.White).output("output/runAuto-
18 crop.png")(writer)
19 //kropovanje slike
20 run.resize(2.0).output("output/runCrop.png")(writer)
21 run.resize(0.25).output("output/runCropInv.png")
   (writer)
   //kompozicija slike sa drugom slikom za odredjeni
   postotak
   run.composite(new AlphaComposite(0.5f), rock).
   output("output/runAlphaComposite.png")(writer)
   //primjena filtera
   run.filter(new GlowFilter(0.5f)).output("output/run-
   Glow.png")(writer)
```

Слика 14

Анализа реализованих DSL-ова

На основу прегледа особина језика (кључних за концизнију синтаксу) које су наведене у Табели 1, као и претходно приказаних DSL-ова, извршена је анализа имплементираних DSL-ова.

Посматрајући реализације DSL-а помоћу *Scala* и *Groovy* језика у односу на *Java* језик, може се видјети да је синтакса концизнија, захваљујући концизнијој синтакси *Scala* и *Groovy* језика. Нема непотребних „шумова“, односно синтаксних литерала попут: заграда код листе параметара методе, симбола „;“ за крај исказа, оператор приступа („.“) за позивање метода, литерала попут двоструких наводника. Двоструки наводници се могу избјећи коришћењем статичких промјењивих које садрже дате *String* вриједности, или енкапсулацијом *String* вриједности помоћу *case* класа (код *Scala* језика). Значење наведених литерала је познато само програмерима, и немају потребе бити присутни код реализованог DSL-а, јер корисницима DSL-а они нису јасни, а нису им ни битни. Тиме је корисницима DSL-а, синтакса DSL језика ближа њиховом домену коришћења, односно њиховим терминима које користе код обраде слике. Код реализације DSL-а помоћу *Java* језика ово није било могуће јер синтакса језика то не дозвољава. Осим тога, прилагођавање синтаксе DSL-а према доменским корисницима код *Scala* и *Groovy* језика помоћу именованих и подразумијеваних аргумената код метода и конструктора, концепта отворених класа који се примјењује над бројевима из DSL-а, даје природнију и читљивију синтаксу у односу на имплементацију код *Java* језика. Приликом реализације DSL-а помоћу *Scala* и *Groovy* језика коришћено је преклапање оператора, како би се реализовала концизна операција композиције двије слике. Таква реализација није била могућа код *Java* језика, јер *Java* језик нема могућност преклапања оператора. Код реализације помоћу *Java* језика коришћене су анонимне функције за реализацију блока наредби, као и код *Scala* и *Groovy* језика.

Особине језика	*Java*	*Scala*	*Groovy*
Заграде код листе аргумената су опционе	-	± (могуће је само када метода нема или има један аргумент)	± (једино није могуће када метода нема аргумената)
Симбол „;“ за крај исказа је опцион	-	+	+
Оператор приступа „.“ је опцион	-	± (могуће је само када метода нема или има један аргумент)	+
Именовани аргументи код метода и конструктора	-	+	+
Подразумијевани аргументи код метода и конструктора	-	+	+
Концепт отворених класа	-	+	+
Преклапање оператора	-	+	+
Анонимне функције	+	+	+

Табела 1

Приликом реализације DSL-а помоћу *Java* језика, реализован је само шаблон *smart* API, који је увезивао позивање метода. Овдје су називи метода прилагођени терминима из домена обраде слике, како би се постигла што већа читљивост („течност“) програмског кода DSL-а. Код *Scala* језика, кориштени су *smart* API и *typed embedding* шаблони. Приликом реализације DSL-а помоћу *Groovy* језик кориштени су *smart* API, *reflective metaprogramming* и *runtime metaprogramming* шаблони.

Оно што корисник добија кориштењем реализованих DSL-ова, јесте чистија синтаксa и интерфејс прилагођен операцијама за обраду слике, у односу на синтаксу када би користио класичне библиотеке (за обраду слике) писане у појединим језицима, а за чије би кориштење морао познавати и сам програмски језик.

Извршавање DSL скрипте, корисник може позивати типичним командама из командне линије за покретање *Scala*, *Groovy* или *Java* програма. Овај начин је прикладан уколико је потребно извршити велики број операција над сликом, или над више слика (попут „*batch*“ обраде), а које су наведене у скриптиној датотеци. У односу на скрипту, ручно извршавање великог броја операција уз помоћ класичних апликација за обраду слике, представљао би временски трајан и мукотрпан посао. Осим начина да се DSL извршава као скрипта, сви наведени језици за реализацију DSL-а имају своје интерактивне конзоле REPL (енг. *Read–Eval–Print Loop*), помоћу којих се могу извршавати појединачне операције DSL-а, али и погледати добијени резултати.

ЗАКЉУЧАК

Посматрајући развој програмских језика још од времена C и C++ језика, па до појаве *Java*, *Scala* и *Groovy* језика, може се уочити тренд повећавање концизности у синтакси датих језика. Синтакса језика тежи да буде што сличнија природним језицима, и тежи да се програмер ослободи писања *boilerplate* програмског кода. Увођењем DSL језика, који се још називају и „мини језицима“, покушава се омогућити концизна и природна синтакса корисницима одређеног домена, који нису програмери, а који би помоћу датог DSL-а могли потврдити пословна правила која се програмирају у одређеној апликацији. Дата пословна правила се могу написати помоћу DSL језика, а чија ће синтакса бити позната корисницима одређеног домена. Идеалан је случај када корисници DSL-а могу потпуно слободно да програмирају и рјешавају проблеме из свог домена помоћу DSL језика. Оно што је већ речено у претходној анализи, *Scala* и *Groovy* језик захваљујући својој концизној синтакси, концепту отворених класа, преклапању оператора и једноставнијим концептом метапрограмирања, успјешно реализује концизан DSL језик. Помоћу *Java* језика, код реализације DSL-а се не добија до тог нивоа природнија и концизнија синтакса, управо због недостатка наведених могућности.

ЛИТЕРАТУРА

[1] *Martin Fowler, Domain - Specific Languages, 2010.* година
[2] *Debasish Ghosh, DSLs in action, 2011.* година
[3] *Krzysztof Czarnecki*, *Generative Programming*, докторска дисертација, 1998. година
[4] *Jonathan Bartlett*, *Introduction to metaprogramming*, https://www.ibm.com/developerworks/library/l-metaprog1, 2005. година
[5] *Joshi Prateek, What Is Metaprogramming?*, https://prateekvjoshi.com/2014/04/05/what-is-metaprogramming-part-22, 2014. година
[6] *Krauss Aaron*, *Programming Concepts: Type Introspection and Reflection*, https://thesocietea.org/2016/02/programming-concepts-type-introspection-and-reflection, 2016. година
[7] *Gregor Kiczales*, *Jim des Rivieres*, *Daniel G. Bobrow*, *A Review of The Art of the Metaobject Protocol*, 2010. година
[8] *Brian Foote*, *Ralph E. Johnson*, *Reflective Facilities in Smalltalk-80*, 1989. година
[9] *Fu Cheng*, *A Practical Guide for Java 8 Lambdas and Streams*, 2018. година
[10] *Java8 Tutorial, https://www.tutorialspoint.com/java8*
[11] *Tour of scala*, *https://docs.scala-lang.org/tour/tour-of-scala.html*
[12] *Loverdo Christos*, *Steps in Scala: An Introduction to Object-Functional Programming*, 2010. година
[13] *Groovy* документација, *http://groovy-lang.org/documentation.html*
[14] *Stephen Samuel, Scrimage* библиотека, *https://github.com/sksamuel/scrimage*

Милош Шербић, дипл. инж. ел. Unicredit Bank Banja Luka a.d., РС, БиХ
Контакт: milosserb@gmail.com
Област интерресовања: објектно-оријентисано пројектовање и програмирање, развој web апликација, Java EE технологије

проф. др Зоран Ђурић, Електротехнички факултет, Универзитет у Бањој Луци, РС, БиХ
Контакт: zoran.djuric@etf.unibl.org
Област интерресовања: сигурност, криптографија, PKI, платни системи и протоколи, формална верификација, машинско учење, data science, објектно-оријентисано програмирање и моделовање, Интернет програмирање, развој мобилних апликација, XML-базирана међуоперативност, Web сервиси, рачунарске мреже, penetration testing, систем интеграција