LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

JAVA KODA ATKALIZMANTOŠANA AR HUNTER

BAKALAURA DARBS

Autors: **Igors Gavrilovs**
Studenta apliecības Nr.: ig13047
Darba vadītājs: Dr.dat., profesors Uldis Straujums

RĪGA 2017

ANOTĀCIJA

Daudzos gadījumos programmatūras izstrādātājam ir vajadzīga funkcionalitāte, kuru jau sniedz kāda trešās puses bibliotēka. Šajā gadījumā ir vēlams atkārtoti izmantot pastāvošo kodu, nevis izstrādāt to no jauna. Hunter ir rīks, kas atvieglo koda atkalizmantošanu ar saistīto metožu meklēšanu datu bāzē un nepieciešamo koda apvalka automātisko sintēzi.

Darbā tiek izpētīts programmatūras atkalizmantošanas jēdziens un rīks Hunter.

Bakalaura darbs ir sadalīts trīs nodaļās. Pirmajā nodaļā notiek koda atkalizmantošanas jēdziena definīcija un atkalizmantošanas veidu klasifikācija. Otrajā nodaļā ir Hunter rīka augsta līmeņa apraksts. Trešajā nodaļā ir Hunter rīka eksperimentāls pētījums un rezultāti.

Atslēgvārdi: koda atkalizmantošana, koda sintēze, automātiskā programmēšana, Java.

ABSTRACT

Java code reuse with Hunter

In many cases, the software developer needs a functionality that already exists in some third party library. In this case, it is preferable to reuse existing code rather than develop it again. Hunters is a tool that facilitates code reuse by finding compatible methods in considerable code bases and automatically synthesizing any necessary wrapper code.

The purpose of this paper is to explore the concept of software reuse and tool Hunter.

Bachelor's thesis is divided into three sections. In the first section there is an explanation of the code reuse concept and classification of reuse types. In the second section there is a high-level description of Hunter. In the third section is posted Hunter experimental research and the results.

Keywords: code reuse, code synthesis, automatic programming, Java.

SATURS

APZĪMĒJUMU SARAKSTS

API - Lietojumprogrammas saskarne (Application Programming Interface)

GUI - Grafiskā lietotāja saskarne (Graphical user interface)

IDE - Integrētā izstrādes vide (Integrated development environment)

ILP - Veselo skaitļu lineārā programmēšana (integer linear programming)

Java - Firmas Sun Microsystems izstrādāta objektorientēta programmēšanas valoda

Java SE - Java Standarta Izdevums, plaši izmantota java skaitļošanas platforma izstrādāšanai un izvietošanai (Standard Edition)

JCL – Java klašu bibliotēka (Java Class Library)

JDK – Java izstrādāšanas komplekts (Java Development Kit)

IEVADS

Koda atkalizmantošana ir svarīga programmētāja darba daļa. Prasme atrast un efektīvi implementēt nepieciešamo kodu ir liela izstrādātāja priekšrocība. Autors izvēlējās šo tēmu, jo uzskata, ka tā ir interesanta un iegūtās zināšanas var noderēt turpmāk. Pirms iepazīšanās ar automātisko programmēšanu autoram bija izpratne tikai par manuālo koda atkalizmantošanu, kas faktiski ir vienkārša funkciju meklēšana internetā un kopēšana projektā.

Darbā autors pārbauda Java koda atkalizmantošanas iespējas, analizē atkalizmantošanas jēdzienu un veidus. Autors izvēlējās detalizēti aplūkot rīku Hunter un veikt eksperimentu sēriju, kas ietver sevī Hunter atrasto metožu veiktspējas pārbaudi, koda meklēšanas ātrumu salīdzināšanu un sarežģītas metodes automātisko sintēzi.

Izvirzīto hipotēžu pārbaudei autors izmantoja eksperimentu. Visas darbības tika veiktas virtuālā mašīnā ar Linux Ubuntu operētājsistēmu un Eclipse Mars integrētā izstrādes vidē. Laika mērījumiem tika izmantota java System.nanoTime() metode.

1. KODA ATKALIZMANTOŠANA

Koda atkalizmantošana ir programmatūras atkalizmantošanas īpašs gadījums. Programmatūras atkalizmantošana ir esošās programmatūras artefaktu vai zināšanu izmantošana jaunas programmatūras radīšanai [1].

Programmatūras atkalizmantošanas mērķis ir uzlabot programmatūras kvalitāti un produktivitāti. Atkalizmantošanas principu ievērošana ļauj saīsināt programmatūras izstrādes laiku un ražot vairāk standartizētu programmatūru. Atkalizmantojamība ir programmatūras vienības īpašība, kas norāda tās atkalizmantošanas iespējas [2].

Pastāv dažādi veidi, kā atkārtoti izmantot programmatūru. Tās svārstās no klašu un metožu atkalizmantošanai no bibliotēkām līdz lietojumprogrammatūras atkārtotai izmantošanai.

Koda atkalizmantošana tiek praktizēta jau no pašām pirmajām programmēšanas dienām. Programmētāji vienmēr atkārtoti izmantoja koda sadaļas, veidnes, funkcijas un procedūras. Pamatojoties uz to, ka ir pieejams liels atklātā pirmkoda daudzums un lielākā programmu daļa nav pilnīgi jauna, iespējams iedomāties, ka ievērojamā daļa koda, kas tiek rakstīts šodien, ir iepriekš sarakstīta un pieejama atklātā pirmkoda repozitorijā.

Kā atzīta studiju joma programmatūras inženierijā, koda atkalizmantošana sākas 1968. gadā, kad Douglas McIlroy [3, 86-2. lpp] piedāvāja tagad slavenā NATO konferencē par programmatūras inženieriju balstīt programmatūras nozari uz atkārtoti izmantojamiem komponentiem. Viņš apgalvoja, ka vēlētos redzēt, ka komponentes kļuvusi par cienīgu zaru programmatūras inženierijā [4, 85. lpp]. McIlroy uzskatīja, ka programmatūras komponentes (rutīnprogrammas) var būt plaši piemērojamas dažādām mašīnām un lietotājiem un tām jābūt sakārtotiem grupās atbilstoši precizitāti, noturību, vispārīgumu un veiktspēju.

Pastāv divas programmatūras atkalizmantošanas metodes: melnās kastes atkalizmantošana un baltās kastes atkalizmantošana. Melnās kastes atkalizmantošanas mērķis ir integrēt līdzekļus plānotā sistēmā bez sākotnējas līdzekļu modifikācijas. Baltās kastes atkalizmantošanā līdzekļi var tikt modificēti pirms integrācijās sistēmā [5].

Atkalizmantošanas iedalījums atkarīgi no objekta:

• Sistēmu atkalizmantošana.

Veselas sistēmas, kas var ietvert vairākas lietojumprogrammas, ko var izmantot atkārtoti.

• Lietotņu atkalizmantošana. Lietotni iespējams izmantot atkārtoti vai nu iekļaujot to bez izmaiņām citās sistēmās, vai izstrādājot lietotņu bloku.

• Komponenšu atkalizmantošana. Sastāvdaļas no apakšsistēmām un atsevišķus objektus var lietot atkārtoti.

• Objektu un funkciju atkalizmantošana. Mazapjoma programmatūras komponenti.

1. 1. Programmatūras bibliotēka

Ļoti izplatīts koda atkalizmantošanas piemērs ir programmatūras bibliotēkas. Daudzas kopīgas operācijas, piemēram, informācijas konvertēšana starp dažiem atšķirīgiem plaši pazīstamiem formātiem, ārējās atmiņas piekļuve, saskarne ar ārējām programmām vai manipulācijas ar datiem (skaitļiem, vārdiem, nosaukumiem, vietām, datumiem utt.) ir nepieciešāmas dažādām programmām.

Programmatūras bibliotēka ir datu un programmēšanas kodu kopa, kas tiek izmantota, lai izstrādātu programmatūru un lietojumprogrammas. Tā ir izstrādāta, lai palīdzētu gan programmētājam, gan programmēšanas valodas kompilatoram programmatūras veidošanā un izpildē [6].

Kaut arī šīm bibliotēkām ir lielas atkalizmantošanas iespējas, tās rada arī lielus izaicinājumus programmatūras izstrādātājam. Bieži vien šīs bibliotēkas ir lielas un sarežģītas un izraisa saskarsmes barjeras. Lai efektīvi izmantotu programmatūras bibliotēku, un tādējādi sasniegt augstas atkalizmantošanas likmes, ir nepieciešamas padziļinātas zināšanas par bibliotēkas saturu.

Standarta bibliotēka programmēšanā ir bibliotēka, pieejama visās programmēšanas valodas implementācijās [7]. Šīs bibliotēkas ir metodiski aprakstīti programmēšanas valodas specifikācijās. Iekšējais saturs ir paslēpts izstrādātājam un nekādas izmaiņas netiek lietotas atkārtotas lietošanas artefaktam. Šī atkalizmantošanas forma tiek uzskatīta par melnās kastes atkalizmantošanu.

Jaunās programmatūras izveidei ir pieejams liels atkalizmantošanas bibliotēku skaits. Piemēram, Java bibliotēku meklētājprogramma findJAR.com indeksē vairāk nekā 140,000 Java klašu bibliotēkas [8] un Java SE 8 ar JDK 1.8.0 satur 4240 klases [9, 104 lpp.]. Java Class Library (JCL) ir dinamiski ielādējamo bibliotēku kopums, kas Java lietotnes var izsaukt izpildes laikā. Piemēram, tas satur GUI bibliotēkas java.awt un java.swing, matemātisko bibliotēku java.math, java.applet u.c.

Bibliotēku implementācijās bieži vien ir labi pārbaudītas un notestētas. Trūkumi ir nespēja rediģēt informāciju, kas var ietekmēt veiktspēju vai vēlamo rezultātu, kā arī mācību un konfigurācijas izmaksas.

1. 2. Lietojumprogrammas saskarne

Lietojumprogrammas saskarne (API) ir apakšprogrammu, definīciju, protokolu un rīku kopums programmatūras izstrādei.

API parasti ir saistītas ar programmatūras bibliotēku. Lietojumprogrammas saskarne apraksta un nosaka paredzamo uzvedību (specifikācija), kamēr bibliotēka ir šo noteikumu kopuma īstenošana [10]. Vienai API var būt vairākas realizācijas, kas var izpausties kā dažādu bibliotēku kopa, kas kuras koplieto tādu pašu programmēšanas saskarni. Lietojumprogrammas saskarnes atdalīšana no tās implementācijas ļauj programmām vienā programmēšanas valodā izmantot bibliotēku, kas ir rakstīta citā. Piemēram, jo Scala un Java kompilējās saderīgā baitkodā, Scala izstrādātāji var izmantot jebkuru Java API [11].

Lietojumprogrammas saskarnes piemērs: Open Graphics Library (OpenGL) – neatkarīga vairākplatformu atklātā grafiskā bibliotēka, kas izmanto divdimensiju un trīsdimensiju grafiku.

1. 3. Satvars

Programmatūras satvars ir platforma lietojumprogrammu izstrādāšanai. Tā ir abstrakcija, kurā programmatūru kas nodrošina vispārēju funkcionalitāti var selektīvi mainīt ar papildu lietotāja uzrakstīto kodu, tādējādi nodrošinot efektivitāti.

Satvars ir līdzīgs lietojumprogrammas saskarnei (API), lai gan tehniski satvars ietver API. Kā liecina nosaukums, satvars kalpo par pamatu programmēšanai, savukārt API nodrošina piekļuvi elementiem, ko atbalsta sistēma. Satvars var ietvert arī bibliotēkas, kompilatoru un citas programmas nepieciešamas programmatūras izstrādei [12].

Satvari ir vidēji lielas vienības, kuras var tikt izmantotas atkārtoti. Tie ir kaut kur starp sistēmu atkalizmantošanu un komponentu atkalizmantošanu. Tas ir apakšsistēmas, kas sastāv no abstraktu un konkrēto klašu kolekcijam un saskarnem starp tām. Apakšsistēma tiek īstenota, pievienojot komponentes, lai aizpildītu projektējuma daļas un instanciējot abstraktās klases satvarā. Satvari veicina veselu arhitektūras atkalizmantošanu šauri definētā lietotnes apgabalā un ļauj atkārtoti izmantot ne tikai atsevišķas komponentes, bet visu sistēmu, veidoto no komponentēm, kopā ar to savstarpējām attiecībām.

Programmatūras satvari ļauj veikt atkalizmantošanu trīs līmeņos. Pie projektējuma līmeņa ir tikai abstraktie konstruēšanas risinājumi, ietverti projektēšanas šablonos, kas tiek atkalizmantoti. Nākamā, zemākā līmeņa, atkalizmantošanas elementi ir saskarnes definīcijas

un to starpsavienojumi. Visbeidzot, zemākajā atkalizmantošanas līmenī ir koda atkalizmantošana, kas sastāv no konkrētam komponentēm un konkrētam klasēm [13].

1. 4. Automātiskā programmēšana

Automātiskā programmēšana ir datorprogrammēšanas veids, kurā kāds mehānisms rada datorprogrammas, kas ļauj programmētājam rakstīt kodu augstākā abstrakcijas līmenī. Automātisko programmēšanu varētu rezumēt kā "Mākslīgais intelekts satik kompilatoru".

Automātiskās programmēšanas sistēmas tika iedalītas trijās dažādās formās:

1. Ārējais mašīnkods – instrukcijas parādās kā mnemonisks operācijas simbols un decimālā adrese.
2. Asemblervalodas – operāciju simboli, algebrisko simbolu izmantošana operanda adreses reprezentēšanai un pārejas galamērķis.
3. Algoritmiskas valodas – standarts matemātisks apzīmējums aritmētisko darbību aprakstam un dinamiski elementi skaitļošanas plūsmas aprakstam.

Mašīnkoda rakstīšanā iesaistīti vairāki soļi — procesa salaužana diskrētās instrukcijās, īpašas atmiņas vietas piešķiršana visām komandās un I/O buferu pārvalde. Pēc tam, kad ir paveikti visi soļi matemātisko rutīnprogrammu implementācijai, subrutīnu bibliotēkas un šķirošanas programmas, uzdevums ir apskatīt lielāku programmēšanas procesu. Nepieciešams saprast, kā var atkārtoti izmantot notestētu kodu un palīdzēt mašīnai programmēšanā. Kā ir ieprogrammēts, izstrādātāji pārbauda procesu un cenšās domāt par veidiem, kā abstrahēt šos soļus un iekļaut tos augstākā līmeņa valodā. Tas noveda pie interpretatoru, assembleru, kompilatoru un generatoru attīstības — programmam, kas ir projektētas, lai darbināt vai radīt citas programmas, kas ir automātiskā programmēšana [14, 56 lpp.].

1. 4. 1. Programmu sintēze

Programmu sintēze ir automātiskās programmēšanas īpaša forma, kas ir visbiežāk pārī ar formālās verifikācijas tehniku. Programmu sintēzes mērķis ir izmantot plašus informācijas repozitorijus ar pieejamu pirmkodu, lai ģenerētu specifiskas funkcijas vai klases, kas atbilst lietotāja specifikācijām [15].

Var atšķirt konstruktīvo sintēzi no deduktīvas sintēzes. Konstruktīvā sintēzē hipotēze balstās uz specifikāciju, kas ir konstruktīvi pierādīta, un no šī pierādījuma tiek iegūta programma. Deduktīvajā pieejā programma tiek secināta tieši no spesifikācijas ar piemēroto to pārveidošanu.

Programmu sintēzes sākumpunkts ir parasti formāla specifikācija, kas ir izteiksme kādā formālajā valodā (valoda, kurai ir sintakse, semantika un parasti pierādījumu teorija). Programmu sintēzei, tādējādi, ir daudz attiecību ar formālo specifikāciju. Turklāt, programmu sintezē arī ir saistīta ar formālām metodem datorsistēmu attīstībā un ar automātisko programminženieriju. Visām šīm disciplīnām ir kopīgs mērķis — uzlabot programmatūras kvalitāti.

1. 4. 2. Semantiski balstīta kodu meklēšana

Semantika ir valodas jēdziens, kas ir atšķirīgs no jēdziena sintakse, kas bieži ir saistīts ar datorprogrammēšanas valodu atribūtiem [16]. Kopumā semantika ir saistīta ar īpašiem vārdiem un apzīmējumiem. Piemēram, semantiskais tīkls izmanto vārdus, kas reprezentē tīkla elementus.

Galvenais sapratīgā koda meklēšanā ir ļaut lietotājam precizēt, ko viņš meklē, tik precīzi, cik vien ir iespējams. Tas ietver atslēgvārdus kā neformālu aprakstu, signatūru un testpiemērus.

1. 4. 3. Pirmkoda repozitorijs

Pirmkoda repozitorijs ir failu arhīvs un tīmekļa hostinga iekārta, kur tiek turēts liels pirmkodu daudzums programmatūrai vai tīmekļa lapām, vai nu publiski vai privāti. Tos bieži izmanto open-source projektos un citos vaikāku izstrādātāju projektos lai apstādātu dazādas versijas.

Piemērs: GitHub, tīmeklī balstīts Git repozitoriju mitināšanas pakalpojums.

S6Search koda meklētājprogrammai ir repozitorija iespējas izvēle: SearchCode, GitHub, Hunter, CodeExchange un GitZip [17]

1. 5. Atkalizmantošanas priekšrocības

Zemākas izstrādes izmaksas. Koda atkalizmantošana palielina programmētāja produktivitāti. Izstrādāšanas izmaksas ir proporcionālas izstrādātas programmatūras lielumam. Atkalizmantošana nozīmē mazāku kodu rindiņu skaitu, kas ir jāraksta.

Mazāka kļūdu iespējamība. Implementācijas, ko piedāva lietojumprogrammas saskarnes un bibliotēkas, ir parasti labi notestētas; tātad, koda atkalizmantošana samazina iespējamību, ka implementācijas ir kļūdainas.

Zemāki riski projekta izmaksu aprēķinā. Esošās programmatūras izmaksas jau ir zināmas, savukārt izstrādes izmaksas vienmēr ir strīdīgs jautājums. Tas ir nozīmīgs faktors projekta vadībai, jo tas samazina kļūdas robežu projekta izmaksu aprēķinā. Tas ir īpaši svarīgi tad, ja tiek atkalizmantotas salīdzinoši lielas programmatūras komponentes, piemēram, apakšsistēmas.

Atbilstība standartiem. Daži standarti, piemēram, lietotāja interfeisa standarti, var tikt īstenoti ka atkalizmantojamo komponenšu kopa. Piemēram, ja izvēlnes lietotāja interfeiss tiek implementēts, izmantojot atkārtoti lietojamas komponentes, visas lietotnes sniegs lietotājiem tādus pašus formātus. Standarta lietotāju saskarnes izmantošana uzlabo uzticamību, jo lietotāji pieļauj mazāk kļūdas, kad sastopās ar pazīstamu interfeisu.

1. 6. Atkalizmantošanas trūkumi

Komponentu bibliotēkas radīšana, uzturēšana un izmantošana. Atkalizmantošanas bibliotēkas vai kodu bāzes aizpilde un nodrošināšana var būt dārga. Ir jāpielago attīstības process, lai pārliecinātos, ka bibliotēka tiek izmantota.

Atkārtoti izmantojamu komponentu meklēšana, saprašana un pielāgošana. Programmatūras komponentes ir nepieciešams atrast, saprast un, dažkārt, pielāgot darbam jaunajā vidē. Inženieram ir jābūt pietiekami pārliecinātam par atrast komponentu bibliotēkā pirms komponenšu meklēšana kļūst par daļu no izstādes procesa.

Palielinātas uzturēšanas izmaksas. Ja atkalizmantotās sistēmas pirmkods vai komponente nav pieejamas, tad uzturēšanas izmaksas var būt augstākas, jo atkalizmantošanas sistēmas elementi var kļūt nesaderīgi ar sistēmas izmaiņām.

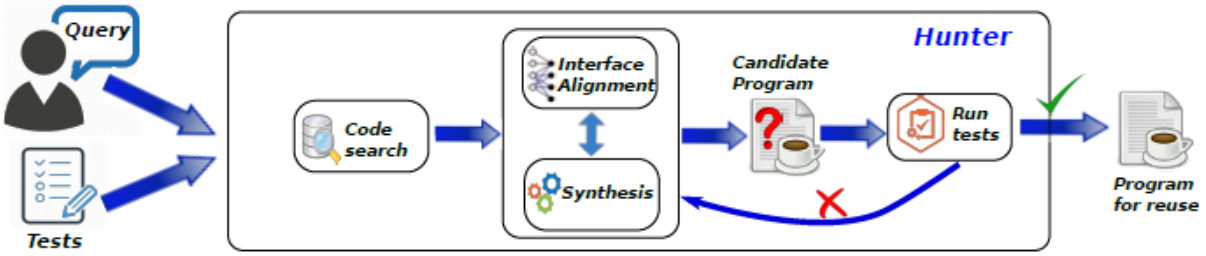
Rīku atbalsta trūkums. Daži programmatūras rīki neatbalsta izstrādi ar atkalizmantošanu. Var būt grūti vai neiespējami integrēt šos rīkus ar komponentu bibliotēku sistēmās. Īpaši tas attiecināms uz rīkiem, kas nodrošina iegulto sistēmu inženieriju.

2. HUNTER

Hunter ir rīks, kas atvieglo Java koda atkalizmantošanu ar saistīto metožu meklēšanu datu bāzē un nepieciešamo koda apvalka automātisko sintēzi. Programma tika izlaista 2016. gadā pētnieku grupai no Teksasas universitātes Ostinā, ASV [18]. Šobrīd Hunter atrodas Beta izstrādāšanas posmā.

Tehniskās prasības: Linux operētājsistēma, Eclipse IDE Neon (4.6), Mars (4.5) vai Oxygen (4.7); Java un JDK 1.7 vai augstāks. Hunter tiešsaistē: https://marketplace.eclipse.org/content/hunter

Programmai ir trīs galvenie moduļi (2.1. att.): koda meklēšana, interfeisa izlīzināšana kopā ar koda sintēzi un koda testēšana. Process atkārtojas līdz brīdim, kad visi testi būs izieti.



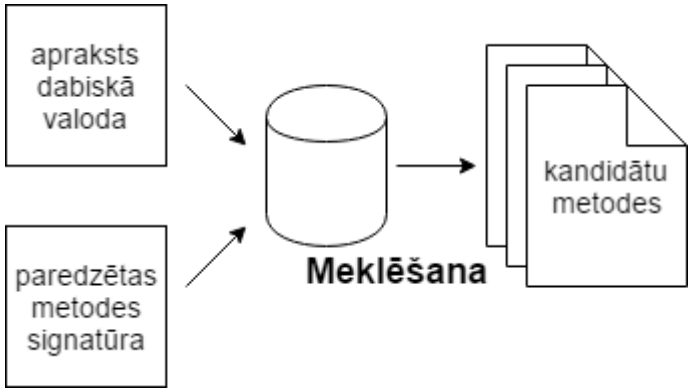
2.1.att. **Hunter rīka darbplūsma** [18]

Testēšana nav obligāta, bet var ievērojami palīdzēt programmai izvēlēties lietotājam vajadzīgo metodi. Gadījumā, ja programma ir pārpratusi funkcijas apraksta būtību, tiks izvēlēts algoritms, kas atgriež pareizo rezultātu.

2.1. Koda meklēšana

Hunter var izmantot jebkuru koda meklētājprogrammu, kas atgriež rezultātus ar metožu detalizāciju. Pašreizējā implementācijā Hunter in integrēts ar Pliny koda meklēšanas programmu, kas lieto datu bāzi ar vairāk nekā 12 miljoniem [18] Java metodēm, kas ir iegūti no atvērtā pirmkoda krātuvēm, piemēram, GitHub vai Bitbucket.

Pirmais darbplūsmas solis ir meklēšanas modulis (sk. 2.2. att).



2.2. att. **Koda meklēšanas process**

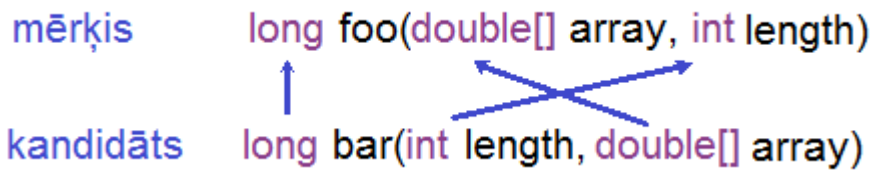
Hunter ņem vēlamās metodes parakstu (signature) kopā ar dabiskās valodas aprakstu (vienas līnijas komentārs angļu valodā) un formulē vaicājumu. Šis vaicājums tiek izmantots, lai atrastu kodu bāzē galveno kandidātu metodi, kas ir vislīdzīgākais vēlamajai metodei. Programmai ir svarīgs argumentu tips un skaits. Metodes nosaukums neietekmē vaicājuma rezultātus.

Pēc koda meklēšanas Hunter veido līdzības matricu no argumentu tipiem, ko izmanto turpmāk.

2.2. Interfeisa izlīdzināšana

Ieejas parametri starp mērķi un kandidātu ne vienmēr sakrīt ar to veidiem vai secību. Šajā gadījumā tiek izmantota interfeisa izlīdzināšana (interface alignment).

Pēc koda meklēšanas un līdzības matricas izveides, Hunter cenšas uzģenerēt vēlamas metodes realizāciju, izmantojot kandidāta funkciju (sk. 2.3. att). Programma izmanto tipu līdzības matricu un mēģina atrast optimālo izlīdzināšanu starp vēlamo signatūru (signature) un adaptēšanas kandidātu (candidate adaptee) [18].



2.3. att. **Interfeisa izlīdzināšanas piemērs**

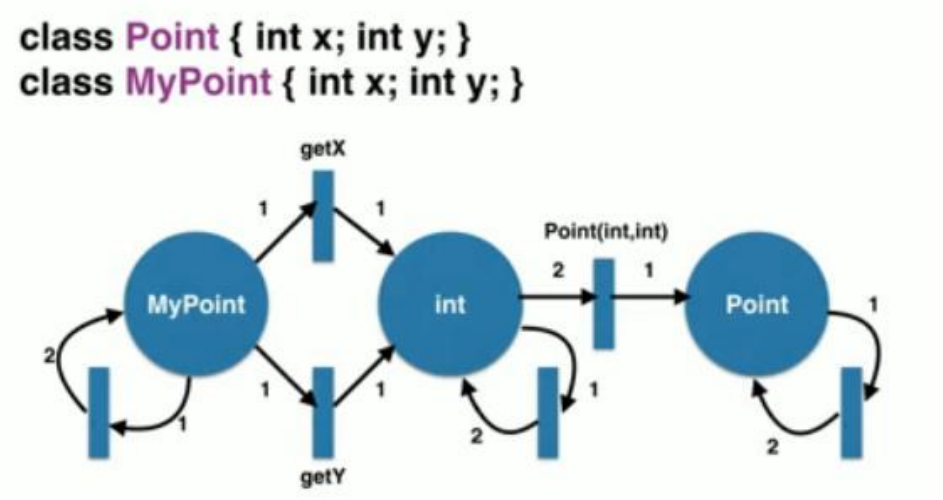
Automātiskā argumentu izlīdzināšana ir iespējama arī tad, ja arguments ir atsauce uz objektu.

2.3. Koda sintēze

Hunter algoritms izsauc adaptera ģenerācijas metodi AdapterGen un automātiski ģenerē apvalka kodu, balstoties uz šo izlīdzinājumu. Šāda veida pārveidošanas paplašinā koda meklēšanas un sintēzes iespējas un ļauj pāriet no viena argumentu tipa uz otru. Piemēram, ja funkcijā figurē atsauce uz objektu MyPoint – punkts, kura koordinātes tiek noteiktas ar diviem int tipa atribūtiem, Hunter izveidos funkciju, kuras argumenti būs divi int tipa argumenti.

Pašreizējā Hunter versijā AdapterGen procedūra izmanto SyPet rīku [18], kurš izmanto jaunu tipu virzītas sintēzes algoritmu, kura pamatā ir Petri tīklu sasniedzamības analīze.

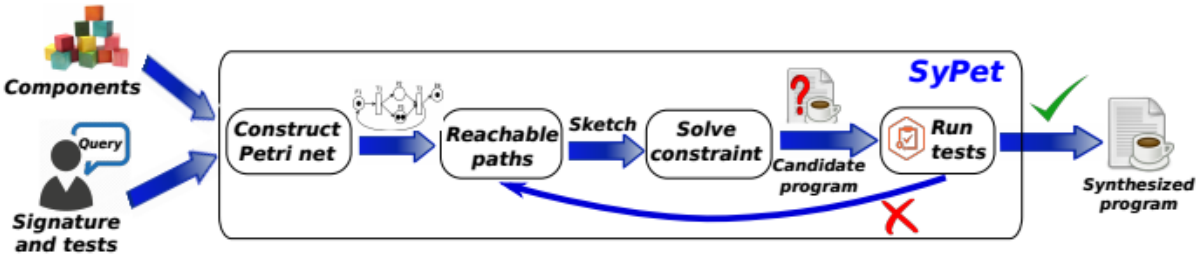
Tieša divdaļu grafa (2.4. att.) mezgli atbilst tipiem, pārejas reprezentē metodes un marķierierīces (tokens) apzīmē mainīgo skaitu, kas ir pieejams katram noteiktam tipam [19].



2.4. att. **Apvalka koda sintēze, izmantojot Petri tīklus**

2.3.1. SyPet rīks

Sasniedzamības problēma ir viena no galvenajām Petri tīklu problēmām. Petri tīklu redukcija pārveido oriģinālo PT uz PT', kur PT' īpašības var vieglāk analizēt.



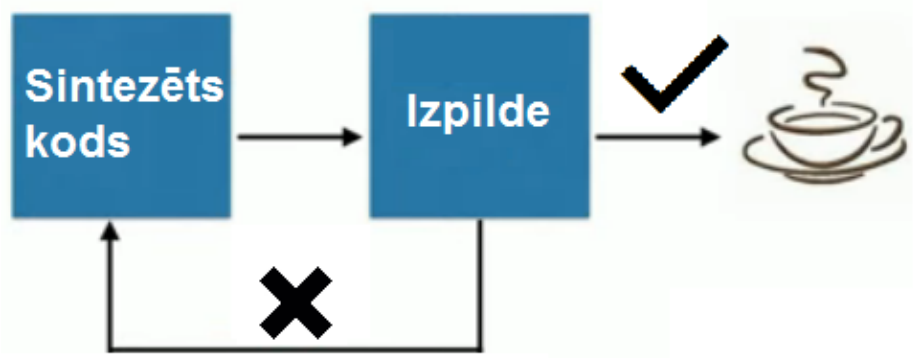
2.5. att. **SyPet rīka algoritms [15]**

SyPet rīks var veiksmīgi sintezēt netriviālus plānošanas uzdevumus, kas tiek apkopoti no tiešsaistes forumiem un GitHub projektiem.

2.4. Koda testēšana

Pēc apvalka koda sintēzes dotai kandidāta funkcijai, Hunter automātiski atrisina visas atkarības un uzsāk testēšanu (2.6. att). Ja kāda testpiemēra rezultāts ir negatīvs, Hunter atkāpās (backtracks), pievienojot papildu ierobežojumus attiecīgai ILP problēmai un jautā ILP risinātāju par citu risinājumu.

Ja šāds risinājums nepastāv, tas nozīmē, ka Hunter ir izmantojusi visas derīgas kanditāta izlīdzināšanas.



2.6. att. **Sintezētas programmas pārbaudes shēma**

Algoritms beidzas vai nu kad Hunter atrod implementāciju, kas iziet visus testpiemērus, vai nu programmai beidzas visas iespējamas adaptējamas funkcijas.

2.4.1. JUnit testi

JUnit ir vienībtestēšanas satvars Java programmēšanas valodai. Tas ir svarīgs rīks testu virzīta izstrāde. JUnit nodrošina statiskās metodes, lai testētu noteiktus nosacījumus, izmantojot Assert klasi. Šādi apgalvojumi parasti sākas ar assertFalse vai assertTrue atslēgvārdu.

Eclipse integrētā izstrādes vide atbalsta JUnit testēšanas satvaru un ļauj veikt interaktīvus testus. JUnit testpiemērus jāizveido atsevišķā nodaļā; pirms katra testpiemēra jābūt anotācijai “@Test”.

2.5. Vienkāršs piemērs

Mūsu programmai ir nepieciešama funkcija, kas pārbauda, vai dotais skaitlis ir pirmskaitlis un atgriež boolean tipa vērtību. Mēģināsim atrast vai sintezēt šo funkciju ar rīku Hunter. Eclipse izstrādes vidē izveidosim klasi PrimeNumber.java. Lai izveidotu vaicājumu, ir nepieciešama funckijas signatūra jeb prototips un vienā rindā komentārs angļu valodā – funkcijas apraksts dabiskā valoda. Kā argumentu funkcija checkPrime (2.7. att.) saņems vienu int tipa mainīgo un atgriezīs "true" vai "false".

```
public class PrimeNumber {

    //@check prime number
        public static boolean checkPrime(int number) {

        }
}
```

2.7. att. **Funkcijas apraksts un signatūra**

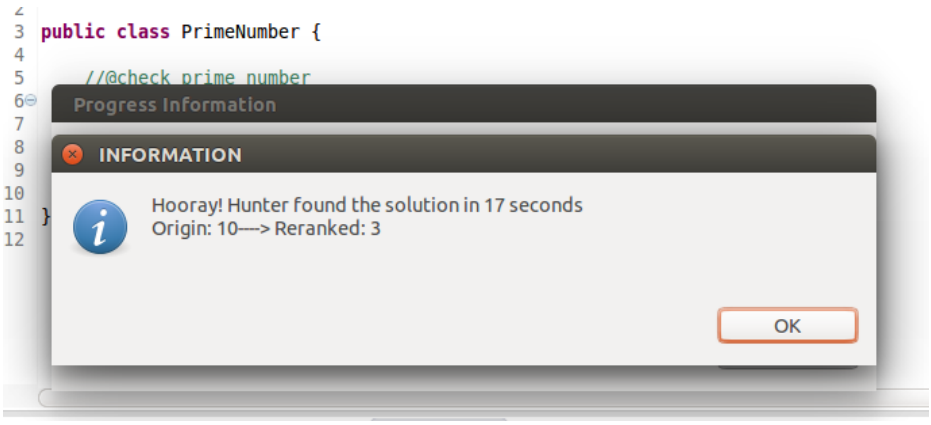
Lai pārbaudītu atrasta risinājuma pareizību, pievienosim JUnit testpiemēru. Atsevišķā klasē TestPrimeNumber.java izveidosim metodi ar „@Test" anotāciju un pievienosim pārbaudi. Testu uzskata par izturētu, ja metode checkPrime atgriezīs "true" ar argumentu int number = 59. Būtu jāatceras, ka jābūt importētai galvenai klasei: import hunter.PrimeNumber;

```
4  import org.junit.Test;
5  import hunter.PrimeNumber;
6
7
8  public class TestPrimeNumber {
9
10     @Test
11     public void test1() {
12         assertTrue(PrimeNumber.checkPrime(59) == true);
13     }
14 }
15
```

2.8. att. **JUnit testpiemērs**

Tālāk jānovieto kursors uz metodi checkPrime un jānospiež taustiņu kombināciju ctrl+F5. Sākas testēšanas un meklēšanas process.

Ja Hunter atrada vajadzīgo metodi, parādīsies veiksmes ziņojums (2.9. attēls). Hunter atrada risinājumu 17 sekunžu laikā.



2.9. att. **Veiksmes ziņojums**

Attēlā 2.10. ir redzams rezultāts. Hunter piedāvātais risinājums lieto trešās puses bibliotēku.

```
1  package hunter;
2
3  public class PrimeNumber {
4
5      //@check prime number
6          public static boolean checkPrime(int number) {
7              boolean ret = new model.RSA().isPrime(number);
8              return ret;
9
0
1          }
2  }
```

2.10. att. **Hunter sintezēts rezultāts**

Bieži Hunter automātiski importē trešās puses bibliotēkas. Šādā gadījumā vajadzīgie faili tiek pievienoti projektam automātiski. Vienas funkcijas darbībai var būt nepieciešamas vairākas Java bibliotēkas.

3. HUNTER RĪKA PĒTĪJUMS

Veicot Hunter spraudņa pētījumu, darba autoram radās daži jautājumi par programmas darbību. Atklātās problēmas izraisīja četru hipotēžu izvirzīšanu. Hipotēžu pārbaudei autors veica virkni eksperimentu, kuru rezultāti varētu palīdzēt izdarīt secinājumus par Hunter efektivitāti un nepieciešamību Java koda atkalizmantošanā.

Izmantotā datora parametri:

- Intel Core i3-4000M Procesors 2.40GHz
- RAM 4,00 GB (3.88GB derīgas atmiņas)
- 64-bit Operētājsistēma (Windows 10)

Izmantotās virtuālās mašīnas parametri:

- Linux Ubuntu 14.04 64-bit
- 2280MB atmiņa
- Java EE Eclipse (Mars.2) v 4.5.2

Autora plāns rīka pētīšanai: īss situācijas apraksts, hipotēzes izvirzīšana, eksperiments un rezultāti.

3.1. Ātrumu salīdzinājums

Pirmā eksperimenta mērķis bija pārbaudīt, vai rīks Hunter tiešām ir tik ātrs un efektīvs salīdzinājumā ar koda manuālo meklēšanu Internetā, ka to apgalvo programmas izstrādātāji savā publikācijā ar rīka augsta līmeņa pārskatu [18]. Tika izvirzīta **1. Hipotēze**: koda automātiskā sintēze ar rīku Hunter ir daudz atrāks process, nekā koda manuāla meklēšana.

Eksperimenta gaitā tika mērīts laiks, nepieciešams spraudnim Hunter funkcijas automātiskai sintēzei un laiks, kas nepieciešams, lai atrastu risinājumu tam pašam uzdevumam patstāvīgi. Tā kā Hunter koda sintēzes procesā izmanto interneta pieslēgumu un koda repozitoriju, tika nolemts atļaut izmantot internetu arī koda manuālai meklēšanai.

Eksperimenta rezultātā tika konstatēts, ka koda automātiskā sintēze ar Hunter tiešām vairākas reizes ātrāk par manuālo meklēšanu. Tabulā 3.1. var redzēt laika atšķirību starp abām koda atkalizmantošanas pieejām. Ir redzams, ka salīdzinoši ātri autoram izdevās uzrakstīt funkciju (sk. 1. pielikumu), kas izdruka divdimensiju masīvu – 113 sekundes, savukārt trešās funkcijas veidošana aizņēma gandrīz 10 minūtes.

3.1. tabula

Laika salīdzinājums: manuālā koda meklēšana un automātiska koda sintēze ar Hunter

Java funkcijas apraksts Hunter vaicājumam	Atgriešanās tips	Argumenti	Manuāla meklēšana, laiks sekundēs	Meklēšana ar Hunter, laiks sekundēs	Starpība
fill matrix with random numbers	void	int[][]	133	-	-
print array	void	int[][]	113	23	490%
create matrix with random numbers	int[][]	int a, int b	569	15	3793%
remove all primes	void	int[][]	168	-	-
invert the given matrix	int[][]	int[][]	278	17	1635%

Hipotēze tika apstiprināta. Izmantojot Hunter, iespējams atrast populāro funckiju daudz atrāk, nekā meklējot manuāli.

Eksperimenta laikā tika konstatēta problēma: Hunter neatrada atbilstošu kodu diviem vaicājumiem. Šī problēma ir kļuvusi par iemeslu nākamas hipotēzes izvirzīšanai.

3.2. Komplekso metožu sintēzes pārbaude

2. Hipotēze: Hunter ir ļoti neefektīvs kombinētās metodes sintēzē.

Kombinētā jeb salikta metode ir metode, kuru var attēlot kā divu atsevišķu funkciju savienojumu. Eksperimenta veikšanai tika ņemtas triviālas funkcijas, kuras Hunter viegli atrod kodu bāzē.

Otrā eksperimenta mērķis bija attiecīgas hipotēzes pārbaude. Katrs vaicājums saturēja loģisko elementu vai nu “and”, vai nu “if” loģisko elementu. Autoram bija interesanti saprast, cik veiksmīgi Hunter apstrādās vaicājumus ar šāda veida aprakstiem.

Zemāk esošā tabula parāda eksperimenta rezultātus.

3.2. tabula

Salikto funkciju automātiska sintēze ar rīku Hunter

Java funkcijas apraksts Hunter vaicājumam	Atgriešanās tips	Argumenti	Koda sintēzes rezultāts
remove duplicates and calculate arithmetic mean	double	int[]	pozitīvs
square number if number is prime	int	int	negatīvs
multiply numbers and calculate square root	double	int, int	negatīvs
reverse sign if number is prime	int	int	negatīvs
reverse and toUpperCase	String	String	negatīvs

Gandrīz visos gadījumos meklēšanas rezultāts izrādījās negatīvs. 2. Hipotēze tika apstiprināta. Programma ir neproduktīva sarežģīto funkciju meklēšanā.

Piemēram, izrādījās, ka Hunter veiksmīgi atrod metodi, kas reversē virkni un metodi, kas visus virknes burtus pārvērš par lielajiem (3.1.att.), bet nevar korekti atrast metodi, kas vienlaicīgi veic abas šīs vienkāršās darbības. Pirmkods 2. pielikumā.

```
//@reverse
    public static String reverse(String str) {
        java.lang.String ret = new c0090555.LongestPalindromicSubstring().reverseString
        return ret;
    }

//@toUpperCase
public static String upp(String str) {
    java.lang.String ret = org.apache.lucene.analysis.kr.utils.StringUtil.upperCase(str
    return ret;

}

// reverse and toUpperCase

//@reverse then toUpperCase
    public static String revupp(String str) {
        java.lang.String ret = new c0090555.LongestPalindromicSubstring().reverseString
        return ret;


    }
```

3.1.att. **Hunter sintezētas metodes**

No šiem rezultātiem var secināt, ka Hunter rīka iespējas ir ļoti ierobežotas. Vispārīgas funkcijas samērā viegli atrast un savienot patstāvīgi, neizmantojot trešās puses programmas.

Nākama eksperimenta mērķis bija veiktspējas pārbaude.

Tika nolemts pārbaudīt, cik ātri strādā Hunter sintezētie algoritmi. Vienu un to pašu rezultātu var sasniegt ar dažādiem līdzekļiem. Tika nolemts salīdzināt divu algoritmu ātrumu: pirmo uzdevumu risināja Hunter sintezēts algoritms, otro manuāli rakstīts.

3.3. Veiktspējas testēšana

3. Hipotēze: Hunter vienmēr atrod metodi ar optimālāko izpildes laiku.

Pirmajā gadījumā tika pārbaudīts matricu raizināšanas ātrums (3.3. tabula).

3.3. tabula

Matricu reizinājuma izpildes laika salīdzinājums

Matricas izmērs	Algoritma iterāciju skaits	Hunter algoritma izpildes laiks, ms			Vidējais	Manuāli atrasta algoritma izpildes laiks, ms			Vidējais	Vidēja starpība
10x10	500	48	44	106	66	27	36	33	32	206%
50x50	100	172	206	320	233	128	132	129	130	179%

Autors pamanīja, ka Hunter atrasta funkcija matricu reizināšanai izmanto metodi no trešās puses bibliotēkas org.misc.aux, kas nav nepieciešams, jo funkcija ir triviāla (pirmkods 3. pielikumā). Tas palēnināja algoritmu divreiz.

3.4. tabula

Longest common subsequence izpildes laika salīdzinājums

Virknes garums	Algoritma iterāciju skaits	Hunter algoritma izpildes laiks, ms			Vidējais	Manuāli atrasta algoritma izpildes laiks, ms			Vidējais	Vidēja starpība
30	100	58	102	86	82	111	73	65	83	99%
150	100	346	369	290	335	424	428	511	454	74%

Otrā algoritma veiktspēja bija gandrīz identiska: izpildes laika atšķirība svārstījās no 99 līdz 74 procentiem (3.4. tabula).

Rezultāts: 3. Hipotēze noraidīta. Ne vienmēr risinājums, ko piedāvā Hunter, ir efektīvs no veiktspējas viedokļa.

Ceturtā eksperimenta mērķis ir notestēt Hunter rīka spēju sintezēt metodes ar dažāda tipa argumentiem. Kā norādīts iepriekš, Hunter izmanto automātisko argumentu izlīdzināšanu. Programmas izstrādātāji apgalvoja, ka pāreja uz vienotu formu ir spraudņa stiprā puse.

3.4. Interfeisa izlīdzināšanas testēšana

4. Hipotēze: Hunter efektīvi sintezē metodes ar dažāda tipa argumentiem.

Eksperimentālie rezultāti liecina, ka tā ir taisnība. Divos no trim gadījumiem Hunter veiksmīgi atrisināja dažādu argumentu problēmu (3.5 tabula).

3.5. tabula

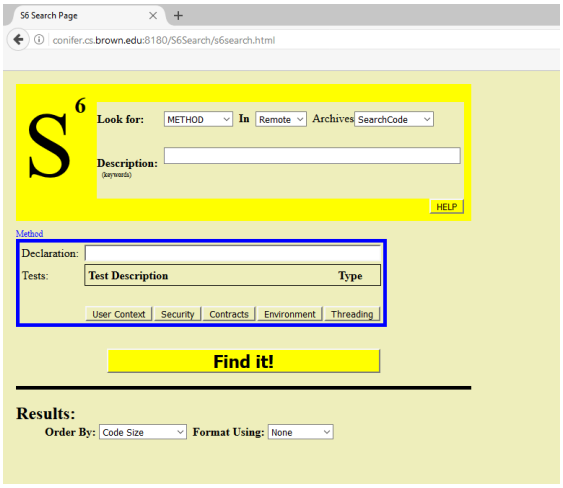
Hunter vaicājumi funkcijām ar dažāda tipa argumentiem

Java funkcijas apraksts Hunter vaicājumam	Atgriešanās tips	Argumenti	Koda sintēzes rezultāts
matrix addition	int[][]	int[][], Vector<Vector<Integer>>	pozitīvs
return array with removed common elements	int[]	Vector<Integer> , int[]	pozitīvs
return arithmetic average	double	int a, double b	negatīvs

Rezultāts: 4. hipotēze ir dāļēji apstiprināta (pirmkods 4. pielikumā). Interfeisa izlīdzināšana ir visinteresantāka programmas daļa un galvenā Hunter priekšrocība. Argumentu pārveidošana ir tas, kas padodas izcili automātiskā programmēšanā.

3.5. Līdzīga programma S6Search

Hunter nav vienīgā semantiski balstīta koda meklēšanas programma. Brauna Universitātes pētnieki ir izstrādājuši savu programmu koda automātiskai sintēzei. Attēlā 3.2. ir redzema tīmekļa saskarne. Programma ir pieejama tiešsaistē: http://conifer.cs.brown.edu:8180/S6Search/s6search.html



3.2.att. **S6Search sistēmas tīmekļa saskarne**

Darba autors salīdzinaja S6Search un Hunter. Tabulā 1.1. var redzēt līdzības un atšķirības starp abām sistēmām.

2.1. tabula

Automātiska koda sintēzes programmu salīdzinājums

	S6	Hunter
automātiska argumentu nolīdzināšana	nav	ir
testi	obligāti	pēc izvēles
meklēšanas objekts	metode vai klase	metode
papildu meklēšanas opcijas	ir	nav
saskarne	web	Eclipse IDE
rezultāts	vairāki varianti	viens variants
izpildes laiks	vidējais	ātrs
programmēšanas valoda	java	
meklēšanas metode	semantiski bazēta koda meklēšana	
ieejas dati	apraksts + deklarācija	

Starp vairākiem koda variantiem S6Search lietotājam risinājums jāatlasa patstāvīgi.

REZULTĀTI

Autors literatūras avotu studiju rezultātā izveidojis koda atkalizmantošanas jēdziena aprakstu, apskatījis atkalizmantošanas priekšrocības un trūkumus.

Autors aprakstījis un izpētījis koda atkalizmantošanas rīku Hunter. Darbā autors izveidoja, izpildīja un dokumentēja eksperimentus Hunter darbības četru aspektu pārbaudei:

• rīka Hunter metožu sintēzes laika salīdzinājums ar manuālo metožu meklēšanas laiku,

• Hunter spēja sintezēt sarežģītus algoritmus,

• Hunter algoritmu veiktspējas testēšana. Sintezētā algoritma veiktspēja tika salīdzināta ar līdzīga algoritma veiktspēju,

• Hunter interfeisa izlīdzināšanas algoritma darbība.

Autors iepazinās ar JUnit testēšanas ietvaru un veiksmīgi izmantoja to Hunter sintezēto metožu pārbaudei.

Darba autors izveidoja salīdzinājuma tabulu divām līdzīgam semantiski balstītām koda meklētājprogrammām – Hunter un S6Search.

SECINĀJUMI

Izrādījās, ka koda atkalizmantošanas jēdziens ir tikai programmatūras atkalizmantošanas īpašs gadījums. Autors uzskata, ka šis jēdziens ir ļoti plašs, jo tā vai citādi visi programmētāji savā praksē saskaras ar programmatūras atkalizmantošanu. Vispopulārākā platforma šāda veida risinājumiem ir StackOverflow.

Iepazīstoties ar rīka Hunter iespējam, autors konstatē, ka programma patiešām ātri sintezē metodes. Ja algoritma nosaukums ir plaši pazīstams un Java metode ir salīdzinoši vienkārša, Hunter būs noderīgs Java koda atkalizmantošanā. Turklāt, Hunter efektīvi sintezē metodes ar dažāda tipa argumentiem, kas ļauj programmētājam ietaupīt laiku datu tipu konvertēšanā.

Autors secina: ja algoritms loģiski sastāv no vairākiem vienkāršiem algoritmiem, Hunter visticamāk to neatradīs un kodu būs jāraksta manuāli. Pēc autora domām tas nopietni ierobežo automātisko koda atkalizmantošanu. Daudz vieglāk būtu atrast algoritmus un savienot pašam.

Ne vienmēr atrastam risinājumam būs laba veiktspēja. Var gadīties, ka metode izmantos trešās puses bibliotēku tur, kur pietiktu ar desmit koda rindiņām. Jāatzīmē, ka praksē ir jāpārbauda katrās bibliotēkas licenci un lietošanas noteikumus. Vairumā uzņēmumu mazpazīstamo amatieru bibliotēku izmantošana nebūs pieļaujama.

Autors konstatē, ka Hunter algoritmos mainīgo nosaukumi nav semantiski, kas ir slikto, jo vienmēr ir vieglāk saprast mainīgo "int result", nevis "int sypet_var55".

Lielākais automātiskās koda atkalizmantošanas trūkums no autora viedokļa ir tas, ka šajā gadījumā programmētājs par algoritmu zina tikai "kas", bet nezina "kā". Tas ir bīstami no tehniskā viedokļa un liek apšaubīt ekonomisko programmētāja vajadzību.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. W. Frakes, C. Terry, "*Software Reuse: Metrics and Models*," ACM Computing Surveys, vol. 28, No. 2, June 1996, pp. 1–21. Pieejams: https://pdfs.semanticscholar.org/90d4/403a28839c3f292b0033cdd58e6e9ca5e0bf.pdf

2. W. Frakes, K. Kang, "*Software Reuse Research: Status and Future*," IEEE Transactions on Software Engineering, vol. 31, no. 7, 2005, pp. 1–8. Pieejams: https://pdfs.semanticscholar.org/660a/9612cdc5baf70c82fa1db291ded7bef39546.pdf

3. T. Gonzalez, J. Diaz-Herrera, A. Tucker, *Computing Handbook: Computer Science and Software Engineering (3rd ed.)*, Boca Raton, USA. CRC Press, 2014.

4. M.D.Mcilroy, "Mass produces software components," in *NATO Software Engineering conference*, 1968, pp. 79–82.

5. Managing CBSE and Reuse [tiešsaiste] [skatīts 03.05.2017]. Pieejams: http://www.idt.mdh.se/kurser/cdt501/2007/lectures/Managing%20CBSE%20and%20Reuse.pdf

6. Technopedia [tiešsaiste] [skatīts 12.05.2017]. Pieejams: https://www.techopedia.com/definition/3828/software-library

7. N. Dale, C. Weems, *Programming And Problem Solving With C++*, Sudbury, USA. Jones and Bartlett, 2005. Pieejams: https://books.google.lv/books?id=EkwyAgAAQBAJ

8. JAR search engine [tiešsaiste] [skatīts 02.05.2017]. Pieejams: http://findjar.com/index.x

9. R. Liguori, P. Liguori, *Java 8 Pocket Guide*, Sebastopol, USA. O'Reilly, 2014.

10. Application programming interface [tiešsaiste] [skatīts 03.05.2017]. Pieejams: http://www.immagic.com/eLibrary/ARCHIVES/GENERAL/WIKIPEDI/W120623A.pdf

11. Combining Scala and Java [tiešsaiste] [skatīts 03.05.2017]. Pieejams: http://www.artima.com/pins1ed/combining-scala-and-java.html

12. Techterms [tiešsaiste] [skatīts 07.05.2017]. Pieejams: https://techterms.com/definition/framework

13. Application of software framework technology [tiešsaiste] [skatīts 15.05.2017]. Pieejams: https://www.pnp-software.com/Publications/MontaltoPasettiSalernoDasia02-V1.0.pdf

14. A. M. Koss, "*Programming on the Univac 1: A Woman's Account*", IEEE Annals of the History of Computing 25, no. 1, 2003, 48–59.

15. Program Synthesis by Sketching [tiešsaiste] [skatīts 24.05.2017]. Pieejams: https://people.csail.mit.edu/asolar/papers/thesis.pdf

16. Techopedia Definition of Demantics [tiešsaiste] [skatīts 24.05.2017]. Pieejams: https://www.techopedia.com/definition/687/semantics-computing

17. S6 Search Engine Web [tiešsaiste] [skatīts 15.05.2017]. Pieejams: http://conifer.cs.brown.edu:8180/S6Search/s6search.html

18. Y. Wang, Y. Feng, R. Martins, A. Kaushik, I. Dillig, S. P. Reiss, "Hunter: Next-Generation Code Reuse for Java," in *Foundations of Software Engineering International Symposium,* 2016. pp. 1–5. Pieejams: https://www.cs.utexas.edu/~isil/hunter.pdf

19. Type-directed Component-based Synthesis using Petri Nets [tiešsaiste] [skatīts 23.05.2017]. Pieejams: http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2215.pdf

PIELIKUMI

1. pielikums. Metodes vaicājuma ātruma salīdzināšanai

```
//@fill matrix with random numbers
	public static void fillnum(final int[][] arr) {
		new gaohannk.RotateImage().rotate(arr); //23s wrong
	}

//manual fill matrix with random numbers
	public static void fillnum(final int[][] arr){
		for (int i = 0; i < arr.length; i++) {
		   for (int j = 0; j < arr[i].length; j++) {
			arr[i][j] = (int)(Math.random()*10);
		   }
		}
	}

//@print array
	public static void printarr(final int[][] arr) {
		com.rick.algorithms.util.Util.printMatrix(arr); //23s
	}

//manual print array
	public static void prntarr(final int[][] array) {
		System.out.println(Arrays.deepToString(array));
	}

//@create matrix with random numbers
	public static int[][] fillnum(int a, int b) {
		int[][] ret = fb.matrix.SpiralOrderPrint.getMatrix(a, b);
		return ret; //15s
	}

//manual create matrix with random numbers
	public static int[][] fillnum2(int a, int b) {

		int[][] arr = new int[a][b];

		for(int i = 0; i<a;i++){arr[i] = new int[b];}

		for (int i = 0; i < arr.length; i++) {
		   for (int j = 0; j < arr[i].length; j++) {
			arr[i][j] = (int)(Math.random()*10);
		   }
		}
		return arr;
	}


// delete all prime numbers from matrix
```

```
//  remove all primes -- 18/ wrong (printing)
// public static void removePrimes(int[][] matrix) {
//
// }

//manual make 0 all primes
	public static void removePrimes(int[][] matrix) {

		for (int i = 0; i < matrix.length; i++) {
		   for (int j = 0; j < matrix[i].length; j++) {
			if(isPrime(matrix[i][j])==true) matrix[i][j]=0;
		   }
		}
	}

//manual make 0 all primes
		public static boolean isPrime(int n) {
			if (n < 2) return false;
			 if (n == 2) return true;
			 if (n % 2 == 0) return false;
			for (int i = 3; i * i <= n; i += 2)
				if (n % i == 0) return false;
				return true;
			}

//@invert the given matrix
	public static int[][] inv(int[][] matrix) {
		int[][] ret = horizon13th.Matrix.RotateImage.rotate(matrix);
		return ret;
	}

//manual invert matrix
	public static int[][] inv2(int[][] matrix) {
		int n = matrix.length;
	    int half = n / 2;

	    for (int layer = 0; layer < half; layer++) {
	       int first = layer;
	       int last = n - 1 - layer;

	       for (int i = first; i < last; i++) {
	          int offset = i - first;
	          int j = last - offset;
	          int top = matrix[first][i]; // save top

	          // left -> top
	          matrix[first][i] = matrix[j][first];

	          // bottom -> left
	          matrix[j][first] = matrix[last][j];

	          // right -> bottom
```

```
                matrix[last][j] = matrix[i][last];

                // top -> right
                matrix[i][last] = top; // right <- saved top
            }
        }

        return matrix;
    }
```

2. pielikums. Paaugstinātas sarežģītības metodes

```
//@remove duplicates and calculate arithmetic mean  13s
    public static double funct(int[] arr) {
            int ret = new FreeTymeKiyan.Candy().candy(arr);
            return ret;
    }

//@ square number if number is prime
    public static int checkPrime(int number) {
            //wrong: no solution
    }

//@multiply numbers and calculate square root
    public static double funct(int num, int num2) {
            //wrong: no solution
    }

//@reverse sign if number is prime
    public static int checkPrime(int number) {
            //wrong: no solution
    }

// reverse and toUpperCase
//@reverse then toUpperCase
            public static String revupp(String str) {
                    java.lang.String ret = new
                    c0090555.LongestPalindromicSubstring().reverseString(str); //wrong
                    return ret;
            }
```

3. pielikums. Metodes veiktspējas testēšanai

```
// ------------------
// matrix multiplication
// ------------------

package hunter3;

import java.util.Vector;

public class MatrixM {


	//Hunter
	//@  matrix multiplication
	public static int[][] multiply(final int[][] first, final int[][] second) {
		double[][] sypet_var54 = new double[first.length][];
		int sypet_var58 = 0;
		for (int[] sypet_var56 : first) {
			int sypet_var59 = 0;
			double[] sypet_var55 = new double[sypet_var56.length];
			for (int sypet_var57 : sypet_var56) {
				sypet_var55[sypet_var59] = sypet_var57;
				sypet_var59++;
			}
			sypet_var54[sypet_var58] = sypet_var55;
			sypet_var58++;
		}
		double[][] sypet_var60 = new double[second.length][];
		int sypet_var64 = 0;
		for (int[] sypet_var62 : second) {
			int sypet_var65 = 0;
			double[] sypet_var61 = new double[sypet_var62.length];
			for (int sypet_var63 : sypet_var62) {
				sypet_var61[sypet_var65] = sypet_var63;
				sypet_var65++;
			}
			sypet_var60[sypet_var64] = sypet_var61;
			sypet_var64++;
		}
		double[][] ret = org.misc.aux.LinearAlgebra.multiply(sypet_var54,
sypet_var60);
		int[][] sypet_var66 = new int[ret.length][];
		int sypet_var70 = 0;
		for (double[] sypet_var68 : ret) {
			int sypet_var71 = 0;
			int[] sypet_var67 = new int[sypet_var68.length];
			for (double sypet_var69 : sypet_var68) {
				int sypet_var73 = (int) sypet_var69;
				sypet_var67[sypet_var71] = sypet_var73;
				sypet_var71++;
			}
```

```
                    sypet_var66[sypet_var70] = sypet_var67;
                    sypet_var70++;
                }
                return sypet_var66;
        }

//------- alg1

          public static int[][] multiply2(int[][] a, int[][] b) {
              int m1 = a.length;
              int n1 = a[0].length;
              int m2 = b.length;
              int n2 = b[0].length;
              if (n1 != m2) throw new RuntimeException("Illegal matrix dimensions.");
              int[][] c = new int[m1][n2];
              for (int i = 0; i < m1; i++)
                 for (int j = 0; j < n2; j++)
                    for (int k = 0; k < n1; k++)
                       c[i][j] += a[i][k] * b[k][j];
              return c;
           }

//------- alg1

        public static void main(String[] args) {
                int[][] a = {{1,2,2,6,7,7,7,7,5,8},
                             {1,2,2,6,7,9,7,7,5,8},
                             {1,2,2,4,7,7,7,7,5,8},
                             {1,3,2,6,7,7,7,7,5,8},
                             {1,2,2,6,7,7,1,7,5,8},
                             {1,2,2,6,2,7,7,7,5,8},
                             {1,2,2,6,7,7,7,7,2,8},
                             {1,3,3,6,7,1,7,7,5,8},
                             {1,2,9,6,7,7,9,7,5,8},
                             {1,2,9,6,7,7,4,7,5,8}};

                int[][] b = {{5,2,2,6,7,7,1,7,5,8},
                             {6,2,2,6,7,9,2,7,5,8},
                             {7,2,2,4,7,7,3,7,5,8},
                             {2,3,2,6,7,7,4,7,5,8},
                             {3,2,2,6,7,7,5,7,5,8},
                             {4,2,2,6,2,7,7,6,5,8},
                             {5,2,2,6,7,7,7,7,2,8},
                             {6,3,3,6,7,1,7,8,5,8},
                             {7,2,9,6,7,7,9,9,5,8},
                             {8,2,9,6,7,7,4,2,5,8}};

                // int[][] b2       (50x50 matrica)
                // int[][] a2       (50x50 matrica)

                 long startTime = System.nanoTime();
```

```
                for (int i = 0; i < 100; i++) {
                        int[][] c = multiply2(a,b);
                }

                int[][] c = multiply2(a,b);

            long elapsedTime = System.nanoTime() - startTime;

              System.out.println("time in millis: "
                    + elapsedTime/1000000);
        }
}


// ------------------
// longest common subsequence
// ------------------
package hunter3;

import java.util.Arrays;
import java.util.Vector;

public class Testclass {

        //hunter
        //@ lcs string
        public static String lcss(String s1, String s2) {
                java.lang.String ret = th.c.Leetcode.DP.LCS.lcs(s1, s2);
                return ret;
        }

        //alg1
        public static String lcss1(String a, String b) {
           int[][] lengths = new int[a.length()+1][b.length()+1];

           // row 0 and column 0 are initialized to 0 already

           for (int i = 0; i < a.length(); i++)
              for (int j = 0; j < b.length(); j++)
                 if (a.charAt(i) == b.charAt(j))
                    lengths[i+1][j+1] = lengths[i][j] + 1;
                 else
                    lengths[i+1][j+1] =
                       Math.max(lengths[i+1][j], lengths[i][j+1]);

           // read the substring out from the matrix
           StringBuffer sb = new StringBuffer();
           for (int x = a.length(), y = b.length();
               x != 0 && y != 0; ) {
              if (lengths[x][y] == lengths[x-1][y])
                 x--;
              else if (lengths[x][y] == lengths[x][y-1])
```

```
                y--;
            else {
                assert a.charAt(x-1) == b.charAt(y-1);
                sb.append(a.charAt(x-1));
                x--;
                y--;
            }
        }

        return sb.reverse().toString();
    }

    public static void main(String[] args) {

            String s1="fHWtErBZ6DuXpRnn0LmiJleLz1kfEU";
            String s2="JqaAmaNVcVbfF40W0Lr48JWD47p5W2";

            String
s3="PLYsSLchFgpqJqaCKZbkuW3tc3jraZuEvQLsKEjkwCP5kSce2Y2TUGf0pBhthV3bHI0
qlxTBwUkVD9nsXycITBDsf7g5QmPxqjigy1XZJv4jsDqo4qmN2kUBs2I87L96fY2k1HLptb
Dowout4QNpFV";
            String
s4="pKVuMRyuyNKy67JUDkUCp8YUgICKbTmJn2oGIq6tIK6lnp74jToZFhKlxFli8x9I27
CrjsAADbd9ThKJ1Hwhw2Mfi8PMWHv2qInZMdjEmhgI3zjJYSp6ycGfK1W8ryqzzGMpYd
Q3mofu5KScbxT9ZR";

        long startTime = System.nanoTime();

            for (int i = 0; i < 100; i++) {
                lcss1(s3,s4);
            }

        long elapsedTime = System.nanoTime() - startTime;

        System.out.println("time in millis: "
            + elapsedTime/1000000);
    }
}
```

4. pielikums. Metodes ar dažādu veidu argumentiem

```
// 1 addition
//@matrix addition
		public static int[][] addm(int[][] a, Vector<Vector<Integer>> b) {
			double[][] sypet_var1 = new double[a.length][];
			int sypet_var5 = 0;
			for (int[] sypet_var3 : a) {
				int sypet_var6 = 0;
				double[] sypet_var2 = new double[sypet_var3.length];
				for (int sypet_var4 : sypet_var3) {
					sypet_var2[sypet_var6] = sypet_var4;
					sypet_var6++;
				}
				sypet_var1[sypet_var5] = sypet_var2;
				sypet_var5++;
			}
			double[][] sypet_var7 = new double[b.size()][];
			int sypet_var11 = 0;
			for (java.util.Vector<Integer> sypet_var9 : b) {
				int sypet_var12 = 0;
				double[] sypet_var8 = new double[sypet_var9.size()];
				for (int sypet_var10 : sypet_var9) {
					sypet_var8[sypet_var12] = sypet_var10;
					sypet_var12++;
				}
				sypet_var7[sypet_var11] = sypet_var8;
				sypet_var11++;
			}
			double[][] ret = org.misc.aux.Matrix.add(sypet_var1, sypet_var7);
			int[][] sypet_var13 = new int[ret.length][];
			int sypet_var17 = 0;
			for (double[] sypet_var15 : ret) {
				int sypet_var18 = 0;
				int[] sypet_var14 = new int[sypet_var15.length];
				for (double sypet_var16 : sypet_var15) {
					int sypet_var20 = (int) sypet_var16;
					sypet_var14[sypet_var18] = sypet_var20;
					sypet_var18++;
				}
				sypet_var13[sypet_var17] = sypet_var14;
				sypet_var17++;
			}
			return sypet_var13;
		}


		//@print matrix
		public static void prnt(int[][] m){
			com.rick.algorithms.util.Util.printMatrix(m);
		}
```

```
	public static void main(String[] args) {

		Vector<Vector<Integer>> ve = new Vector<Vector<Integer>>();
		Vector<Integer> ved1 = new Vector<Integer>();
		ved1.add(1);
		ved1.add(2);

		Vector<Integer> ved2 = new Vector<Integer>();
		ved2.add(3);
		ved2.add(4);

		ve.add(ved1);
		ve.add(ved2);

		int[][] tgt = {{2,4},{6,8}};

		prnt(addm(tgt,ve));
	}


//---
// remove common elements

	//@return array with removed common elements
		public static int[] rmd(Vector<Integer> b, int[] a) {
			java.lang.String[] sypet_var16 = new String[b.size()];
			int sypet_var18 = 0;
			for (int sypet_var17 : b) {
				String sypet_var20 = String.valueOf(sypet_var17);
				sypet_var16[sypet_var18] = sypet_var20;
				sypet_var18++;
			}
			java.lang.String[] sypet_var21 = new String[a.length];
			int sypet_var23 = 0;
			for (int sypet_var22 : a) {
				String sypet_var25 = String.valueOf(sypet_var22);
				sypet_var21[sypet_var23] = sypet_var25;
				sypet_var23++;
			}
			java.lang.String[] ret = new
org.pf.text.StringUtil().remove(sypet_var16, sypet_var21);
			int[] sypet_var26 = new int[ret.length];
			int sypet_var28 = 0;
			for (String sypet_var27 : ret) {
				int sypet_var30 = Integer.parseInt(sypet_var27);
				sypet_var26[sypet_var28] = sypet_var30;
				sypet_var28++;
			}
			return sypet_var26;
		}
```

```
	public static void main(String[] args) {

		Vector<Integer> ve = new Vector<Integer>();
		ve.add(1);
		ve.add(2);
		ve.add(3);
		ve.add(4);
		ve.add(5);

		int[] tgt = {1,4,5,6};

		System.out.println(Arrays.toString(rmd(ve,tgt))); //[2,3]
	}

//----

//@return arithmetic average
	public static double avg(int a, double b) {
		//wrong: no solution
	}
```

Bakalaura darbs „Java koda atkalizmantošana ar Hunter” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Igors Gavrilovs

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: profesors, Dr. dat. Uldis Straujums 29.05.2017.

Recenzents: asociētais profesors, Dr. dat. Edgars Celms

Darbs iesniegts Datorikas fakultātē 29.05.2017.

Dekāna pilnvarotā persona:

vecākā metodiķe Ārija Sproģe_____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē:

Komisijas sekretārs: