LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

Domēnspecifiskā valoda tīmekļa programmēšanai

Bakalaura darbs

Autors: Sandis Krutovs

Stud. apl. sk07110

Darba vadītājs: Dr.sc.comp. Agris Šostaks

RĪGA, 2012.

Anotācija

Bakalaura darba tēma ir izveidot savu domēnspecifisko valodu, lai laika ziņā atvieglotu programmēšanas koda rakstīšanu tīmekļa vietņu izstrādē. Darbs tiek sākts ar domēnspecifisko valodu aprakstu, nosacījumu izvirzīšanu izveidojamai valodai, papildinot to ar rīka izvēli un nobeidzot ar izveidotās valodas aprakstu un secinājumiem.

Risināmā problēma ir programmēšanai veltītā laika samazināšana, izmantojot jaunus rīkus un pieeju programmēšanai. Pieejamie domēnspecifisko valodu rīki ļauj izveidot jaunus valodu risinājumus koda ģenerēšanai. Darbā tiek pētīts kā noris domēnspecifisko valodu izstrāde un kā ieviešana var paātrināt projektu izstrādes tempu.

Darba rezultātā tika izstrādāta reāla domēnspecifiskā valoda, kas dod ieguldījumu gan praktiskā darbā, gan paplašina zinātnisko apvārsni par domēnspecifisko valodu pielietošanu. Tika iegūta arī pieredze domēnspecifiskas valodas ģenerācijai paplašinot zināšanu lauku par tīmekļu vides programmēšanu.

Abstract

Bachelor’s thesis is to create a domain-specific language, to faciliate the time to write programming code for web site development. Work begins with a description of domain-specific language, the conditions for setting up a language, further added with language tool selection process and ending with detailed description of language and conclusions.

The issue tackled in this work is the reduction of time devoted to programm using the new tools and approach to programming. Available domain-specific language tools allow to create new language code generation solutions. The paper explores the progress in domain-specific language design and paths to implement domain-specific language to accelerate development of projects.

This paper tries to get an insight into solving time issue on developing web projects using availible domain specificē language tools to create new domain specific language solution. Also experience of domain specific language creation was obtained broadening the knowledge of web project programming.

Atslēgvārdi

Domēnspecifiska valoda, PHP, tīmekļa vietne, koda ģenerēšana, Spoofax, Eclipse.

Darbā lietotie saīsinājumi

IMP – Rīka Eclipse iebūvētā Meta-Rīku platforma, www.eclipse.org/imp/

EMF – Rīka Eclipse modelēšanas ietvars

PHP – Valodas procesors tīmekļorientētiem risinājumiem, www.php.net

Satura rādītājs

1. IEVADS

Jebkuras programmas izstrādes procesa neatņemama sastāvdaļa ir laika termiņš, kādā programma ir jāizstrādā un tiek izstrādāta. Tiek uzskatīts, ka ir labi, ja pēdējais no minētajiem termiņiem ir ātrāk par pirmo, taču prakse rāda, ka biežāk notiek pretējais – nodošanas termiņš pārsniedz plānoto izstrādes termiņu. Risinot laika trūkuma problēmu projekta ietvaros, programmizstrādes process tiek pakļauts kompromisam ar programmprodukta kvalitātes nosacījumiem, piesaistītā darbaspēka kvalifikāciju un izmaksām.

Izstrādājot tīmekļa vietnes, kompromiss starp laiku un kvalitāti var kļūt par iemeslu neveiksmīgam produkta sākumam. Tieši tīmekļorientētam produktam slikts darbības sākums var nozīmēt arī produkta turpmāko neveiksmi. Kvalitāte lietojamības, drošības un slodzes aspektos nevar tikt atstāta neizstrādāta.

Domēnspecifiskās valodas piedāvā izstrādes procesu padarīt ātrāku un kvalitatīvāku. Ar koda ģenerācijas palīdzību tiek nodrošināta kļūdu izskaušana un process automatizēts, izslēdzot cilvēka kļūdas. Tāpat arī programmēšanas darbspējas netiek iztērētas koda pārrakstīšanai un identisku koda konstrukciju atkārtošanai.

Tīmekļu vietņu programmēšanai izmantot domēnspecifisku valodu nav ierasta prakse. Subjektīvi populārākās ir valodas orientētas programmēšanas valodas, kā, piemēram, Java, ar kuru var radīt ne tikai tīmeklim piemērotas lietotnes, vai arī PHP valoda, kura ir galvenokārt orientēta uz tīmekļa lapām, taču var tikt izmantota arī citiem mērķiem. Gan Java, gan PHP pieļauj iespēju izmantot dažādus programmēšanas ietvarus, kas atvieglo programmēšanas darbu, jo ietvarā jau ir definētas gatavas funkcijas, kuras atvieglo programmētājiem rakstāmā koda apjomu. Taču neskatoties uz to, lielu daļu no programmēšanas laika programmētājs veltī koda rakstīšanai, kas gan bieži atkārtojas, gan arī ir orientēts uz dažādu kontroles nosacījumu veidošanu.

Ne tikai tīmekļorientētam projektam ir svarīgi, cik ātri klientam varēs saražot gatavu programmas produktu vai pat tikai pirmo demonstrācijas variantu. Tāpēc autors pēta pieeju, kā veikt koda ģenerāciju ar domēnspecifiskas valodas palīdzību, lai veidotu programmas kodu konkrētam problēmas apgabalam. Vairāki praksē radīti produkti savā pamatstruktūrā izskatās vienādi, taču tos nav iespējams tieši pārkopēt un izmantot citu projektu vajadzībām. Ir nepieciešams veikt loģiskās izmaiņas, kas prasa laiku un rakstīšanu.

Šajā darbā apskatītā problēma ir programmēšanas ātrums un lietderīgums salīdzinot plaša pielietojuma valodas un domēnspecifiskās valodas.

Darba mērķis ir, balstoties uz autora pieredzi tīmekļu vietņu programmēšanas jomā, izveidot domēnspecifisko valodu. Tiešs praktisks mērķis būtu izveidot praktiski lietojamu risinājumu, ar kura palīdzību būtu iespējams ātri un kvalitatīvi radīt tīmekļorientētus projektus. Zinātniskais mērķis būtu dot ieskatu un novērtējumu domēnspecifiskas valodas radīšanai, lai ar šādu risinājumu palīdzētu programmēšanas vajadzībām.

Par darba hipotēzi tiek izvirzīts apgalvojums, ka ātrākais veids, salīdzinot ar plaša pielietojuma valodām, bibliotēkām un ievariem, kā radīt tīmekļorientētus projektus, ir izmantot domēnspecifisko valodu.

Lai pārbaudītu hipotēzi, šajā darbā tiek apskatīts, kas ir domēnspecifiska valoda, izvēlēts rīka risinājums tās radīšanai, ar rīku tiks radīta valoda, ar kuras palīdzību tiks ģenerēts tīmekļorientēts projekts. Iepriekš minētā procesa laikā tiks veikta analīze, kas tiks salīdzināta ar tīmekļa projekta radīšanu vienkāršā izstrādes platformā. Kā izejas valoda tīmekļa projekta izstrādei ir izvēlēta PHP. Apgalvojumi par tīmekļa vietņu izstrādi PHP valodā ir balstīti uz darba autora vairāk nekā astoņu gadu profesionālo pieredzi darbā ar PHP.

Darbā ir sekojošas daļas – ievads, analīze par domēnspecifiskajām valodām nodaļā „Domēnspecifiskās valodas", analīze par priekšnosacījumiem savas valodas radīšanai nodaļā „Savas domēnspecifiskās valodas radīšana", valodas radīšanas rīka apraksts nodaļās „Rīka izvēle valodas izstrādei" un „Izvēlētā rīka īpašības", valodas izstrāde nodaļā „Valodas izstrāde" un secinājumi. Analīze par domēnspecifiskajām valodām apraksta domēnspecifisko valodu iezīmes un sastāvdaļas. Analīze par priekšnosacījumiem savas valodas radīšanai ņem vērā teoriju un apskata citu projektu pieredzi izstrādājot domēnspecifisku valodu, nosaka nosacījumus, kādi ir vajadzīgi veidojot savu valodu, pamato, kādēļ šādai valodai būtu jāeksistē. Valodas radīšanas rīka aprakstā tiek analizēta rīka izvēle un tā sastāvdaļu īpašības. Valodas izstrādes daļā tiek detalizēti caurskatīta izstrādātā valoda.

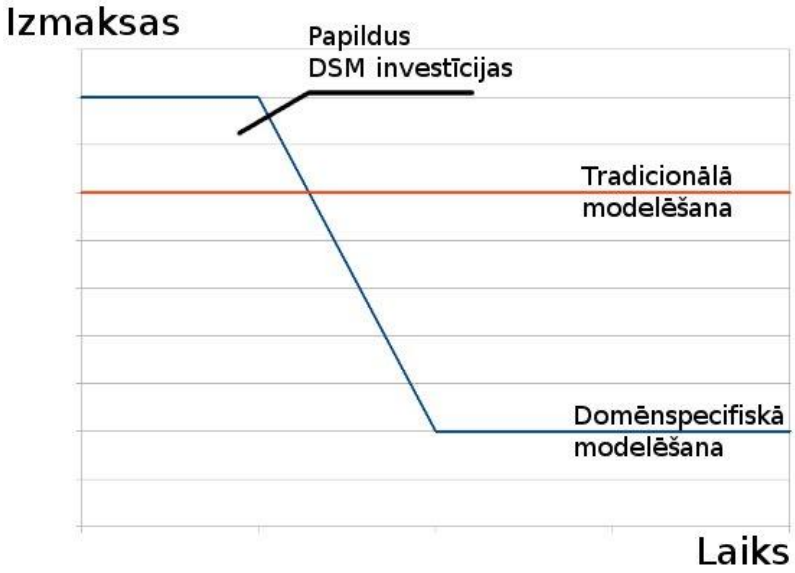
2. DOMĒNSPECIFISKĀS VALODAS

Viena no domēnspecifiskās valodas definīcijām saka, ka domēnspecifiskā valoda ir programmēšanas valoda vai izpildāma specifikāciju valoda, kas, caur noteiktiem apzīmējumiem abstrakcijas līmenī, ļauj konstruēt un izveidot izteiksmes sava domēna problēmu apgabalā[4]. Eksistē dažāda līmeņa domēnspecifiskās valodas. Tradicionālā un sākotnējā pieeja domēnspecifiskajām valodām ir bibliotēku izmantošana tradicionālajās programmēšanas valodās. Nākamais identificējamais un populārākais līmenis ir ietvaru izmantošana. Atsevišķas apraksta valodas lietošana abstrakcijai ir nākamā attīstības pakāpe.

Pretēji tradicionālajām vispārīgās nozīmes programmēšanas valodām, kurām ir plašs pielietojums, domēnspecifiska valoda orientējas uz kādu konkrētu "domēnu" jeb problēmapgabalu. Valoda tiek radīta specifiskas nozares problēmu risināšanai. Orientācija valodu padara par ļoti spēcīgu instrumentu šim vienam problēmgadījumam, taču pilnībā ignorē citus domēnus un to problēmas. Tiesa, viena domēnspecifiskā valoda var tikt izmantota citu domēnu problēmu risināšanā, lai gan nav tam paredzēta. Ar domēnu būtu jāsaprot kāda zināšanu bāze vai lietojuma sfēra, piemēram, personāla uzskaite, artilērijas vadības sistēma vai kāds sarežģīts algoritmu risinājums. Šajā domēnapgabalā eksistē noteiktas problēmas un noteikta risinājumu kārtība, ar kuru terminiem un definējumiem iespējams algoritmisku risinājumu. Iesaistot šos terminus valodā un mākot tos atblstoši transformēt vai interpretēt, iegūst valodu, kas spēj aprakstīt noteikta problēmu domēna risinājumus.

Domēnspecifiskas valodas pamatā tradicionāli ir deklaratīva mikro valoda, kas var izpausties gan kā grafisks zīmējums, gan kā tekstuāls kods. Ja tiek pielietota grafiskā modelēšana, tad mikro valoda tiek reprezentēta kā grafiski elementi caur speciālu rīku, kas paredzēts šādam nolūkam. Mikro valoda satur domēnam specifiskus līdzekļus, kas ļauj aprakstīt topošās sistēmas dažādus aspektus augstākā abstrakciju līmenī, nekā to var izdarīt ar tradicionālajām programmēšanas valodām. Apzīmējot domēna valodā iesaistītās komandas un to izmantojamību, ar valodas aprakstošajiem rīkiem tiek izveidoti modeļi un to elementi. Šīm abstrakcijām var tikt piemērotas transformācijas. Ja pielietoto transformāciju mērķis ir transformēt esošo mikrovalodas deklarāciju uz kodu, tad abstraktais modelis var tikt pārveidots uz kādu konkrētu kodu, ieskaitot plaši izmantojamās programmēšanas valodas. Mikro valodas sintakse var būt kāda no plaši izmantojamām programmēšanas valodām, jo domēnspecifiskās valodas abstrakcijām nav obligāti izmantot kādu īpašu sintaksi – var izmantot arī jau gatavo plašas nozīmes programmēšanas valodu, piemēram, Java vai C++.

Eksistē ārējās un iekšējās domēnspecifiskās valodas. Iekšējās domēnspecifiskās valodas balstās uz kādu jau esošu valodu, vai tā būtu cita domēnspecifiskā valoda vai plaši izmantojamā valoda, tomēr izstrāde uz citas domēnspecifiskās valodas pamata, piemēram, LISP, ir vieglāka, turklāt izstrādājamā domēnspecifiskā valoda manto visas iespējas un ierobežojumus, kas ir pamata valodā. Ārējā domēnspecifiskā valoda tiek izstrādāta atsevišķi un kā šīs valodas lielākā priekšrocība ir tās iespēju brīvība realizēt visu jebkādā vēlamajā formā. Taču lai pilnībā izmantot jaunizveidoto valodu, ir nepieciešams būvēt tulkotāja rīku saderībai ar citām valodām, kas ir lielākais trūkums šim modelim.



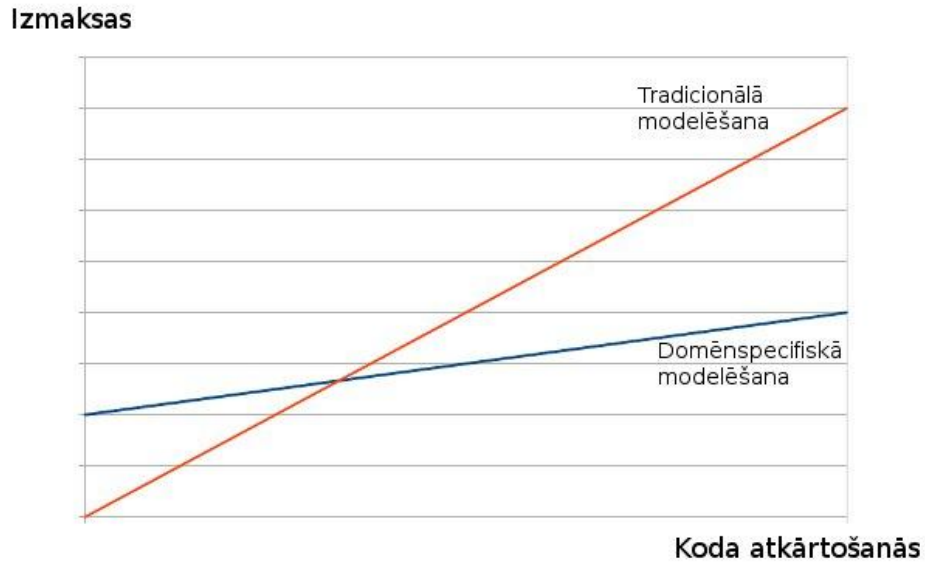
2.1. Attēls. **Izmaksu un laika grafiks tradicionālajām un domēnspecifiskajām valodām**

Šāda deklaratīvas valoda, savienota ar transformācijām uz plaša pielietojuma nozīmes programmēšanas valodām, rada iespēju izvairīties no nevajadzīga programmēšanas koda rakstīšanas. Plaši izmantojamās programmēšanas valodās programmētājam nākas aprakstīt diezgan zema līmeņa nianses, no kurām daudzas atkārtojas. Šādai izstrādei nākas veltīt daudz laika un risināt grūtības, kas rodas rakstot algoritmus, salāgojot vairākus programmas moduļus. Tiek uzskatīts, ka domēnspecifiskās valodas prasa mazāk rakstīšanas no programmētāja puses[11], turklāt samazina kļūdu apjomu. No otras puses, plaši izmantojamās valodas neprasa tādu analīzi un laika ieguldījumu veicot definēšanu un analīzi pirms pašas programmēšanas.

Attēls 2.1. attēlo izmaksas abiem programmēšanas modeļiem atkarībā no laika, kas pagājis kopš izstrādes sākuma[5], bet attēls nr. 2.2. attēlo izmaksas pret koda atkārtošanos[5].

Kā redzams, izmaksas abos gadījumos domēnspecifiskajām valodām ir augstākas sākumā, taču projekta tālākajos attīstības posmos, domēnspecifiskās valodas iegūst ļoti lielu pārsvaru.

Domēnspecifiskās mikro valodas apraksta augsta līmeņa sakarības starp programmas elementiem. Līdz ar to izstrāde no vienas puses ir sarežģītāka, jo nākas pievērsties plānošanai un nianšu formulēšanai, taču, no otras puses, korekti izveidota abstrakcija un veikta transformācija atļauj izvairīties no zema līmeņa koda rakstīšanas. Zema līmeņa koda izveide transformācijas gaitā ir standartizēta, tātad, ļoti niecīgas iespējas, ka šajā kodā tiks pieļautas kādas cilvēciskās kļūdas. Šādu kodu ir iespējams izmantot atkārtoti.



2.2. *Attēls.* Izmaksu un koda atkārtošanās grafiks tradicionālajām un domēnspecifiskajām valodām

Turklāt šāda augsta līmeņa domēna abstrakcijas modelis ļoti labi pakļaujas izmaiņām. Ja nepieciešams veikt kādus uzlabojumus esošajā kodā, tad izmaiņas var tikt veiktas esošajā modelī mainot nosacījumus un pārģenerējot transformācijas. Šāda kārtība nodrošina kļūdu nepieļaušanu, jo uzģenerēto kodu būs radījusi programma, nevis cilvēks. Ja kļūdas eksistēs, tad tās būs transformācijā vai augstākā līmeņa kodā.

Domēnspecifiskās valodas īpašības ir sekojošas[8]:

- Valodas konstrukcijas atbilst domēnvalodas principiem: valoda ir saprotama savas sfēras profesionāļiem, ir efektīva lietošanā, piedāvā modulācijas un integrācijas iespējas.

- Katrs valodas konstrukcijas elements tiek izmantots, lai apzīmētu tieši vienu un atšķirīgu konceptu no domēna apgabala.

- Valodai ir atbalsta rīki, kas nodrošina modeļa un programmu vadību ar izveidošanas, dzēšanas, labošanas, atkļūdošanas un transformāciju iespējām.

- Domēnspecifisko valodu un tās rīkus ir iespējams izmantot vienkopus ar citām valodām ar minimālu piepūli, jo ir būtiski, lai valoda būtu integrēta kopā ar citām konstrukcijām. Tas arī paredz, ka pašai valodai ir iespējas paplašināties un izmantot jaunas konstrukcijas un konceptus.

- Lai realizētu savu potenciālu un būtu iespējams izveidot pietiekami daudz rīkus, gan pašam domēnam gan domēnspecifiskajai valodai jāeksistē samērā ilgā laika posmā, kas mērāms vismaz vairākos gados.

- Valoda ir pietiekami vienkārša, lai domēna (darījuma) eksperti jeb cilvēki, kas nav IT speciālisti, varētu ar to konstruēt savā ikdienas darbā, nemainot ierasto praksi.

- Valoda ir kvalitatīva, jo tā piedāvā vispārējus mehānismus kā uzbūvēt kvalitatīvas sistēmas.

Domēnspecifisko valodu piemēri šobrīd ir atrodami arī daudzos populāros rīkos. Viens no vispopulārākajiem risinājumiem varētu būt Microsoft Excel formulas, kas arī ir maza domēnspecifiskā valoda, kas orientēta uz abstraktu aprēķinu veikšanu un nedarbojas ārpus savas vides. Citas populārās domēnspecifiskās valodas ir SQL – datubāzu pieprasījumu valoda, HTML – tīmekļa vietņu reprezentācijas valoda.

3. SAVAS DOMĒNSPECIFISKĀS VALODAS RADĪŠANA

3.1. Priekšnosacījumi valodas veidošanai

Vajadzība pēc savas domēnspecifiskās valodas var būt pamatota no vairākiem aspektiem. Piemēram, domēnspecifiskās valodas ir vairāk piemērotas lielu un sarežģītu projektu izveidošanai, kura rakstīšana plaša pielietojuma valodā ir sarežģīta – piemēram, paralelizācijas, optimizācijas un transformācijas darbības. Tāpat arī daudzas līdzīgas vai vienādas algoritmiskas darbības var tikt aprakstītas augstākā līmenī un izveidotas automatizēti. Datu struktūru izveide un saglabāšana atšķiras starp plaša pielietojuma valodu sintaksēm, kamēr tās apraksts domēnspecifiskajā valodā saglabā savu integritāti[1] un to ir iespējams transformēt uz individuālām plašas pielietojamības valodām. Katrai šai darbībai ir nepieciešama sava domēnspecifiskā valoda, taču šādas valodas plānošana un izveide arī nav vienkārša. Darbam ar domēnspecifisko valodu ir nepieciešami domēna jomā izglītoti un domēna problēmā pieredzējuši cilvēki, taču šādu cilvēku resursi parasti ir ļoti ierobežoti, it sevišķi ja tas ietver sevī šauru problēmu specifiskā apgabalā.

Lai sāktu izstrādāt savu domēnspecifisko valodu, ir nepieciešamas zināšanas par modelēšanu un valodas izstrādi. Var izšķirt sešus attīstības posmus, kādos eksistē DSL valoda: lēmumu, analīzes, projektēšanas, īstenošanas, uzstādīšanas[12] un uzturēšanas[11]:

- Lēmumu pieņemšana ir definēšanas posms, kurā tiek noteikts kāda veida domēnspecifiskā valoda tiks izstrādāta vai arī tiks izmantota kāda plaši pielietojamā valoda.

- Analīze identificē problēmapgabalu, domēna modeļa elementus, terminus, konceptus un konceptu saistības.

- Projektēšana izveido saites no veidojamās domēnspecifiskās valodas uz plaši izmantojamām valodām, definējot gramatiskos kontekstus. Posms ir atkarīgs no tā, kāda veida valoda tiek veidota un vai valoda ir izstrādāta pilnīgi no nulles.

- Īstenošanā tiek realizēts labākais veids, kādā domēnspecifiskā valoda tiks konstruēta. Šis posms ir īpaši svarīgs, ja valodas rezultātam ir jābūt izpildāmai programmai. Ir nepieciešams izveidot vai apkopot rīku komplektu, kas nodarbosies ar valodas tulkošanu, aplikāciju ģenerēšanu, pirmsapstrādi un valodas iekļaušanu citā kodā.

- Uzstādīšanas posmā ar izveidoto domēnspecifisko valodu kā arī tās rīkiem tiek izveidots domēna modelis un no šī modeļa radīts no modeļa izveidots modeļa attēls. Ar modeļa attēlu tiek domāts starprezultāts, kāds rodas pēc ģenerēšanas. Attēli var būt vairāki, jo ģenerējamo kodu var papildināt un pārģenerējot radīsies cits domēna modeļa attēls. Posma rezultāts ir izpildāma binära programma, ja vien domēnspecifiskā valoda nav domāta tikai aprakstoša.

- Uzturēšanas posms, kurš tiek realizēts, ja ir nepieciešams veikt izmaiņas esošajā domēspecifiskajā valodā, kas ierobežo domēna ekspertu rīcību modeļu līmenī. Nevar noliegt, ka arī pats problēmapgabals attīstās un evolucionē līdz ar laiku, tāpēc ir kritiski turpināt arī tā attīstību, lai modeļu būvēšanas līmenī būtu iespējams realizēt jaunākās izmaiņas.

Būtiskāks aspekts domēnspecifiskās valodas pabeigšanai ir zināšanas[4], kas rodas no praktiska darba ar problēmapgabalu un iemaņām to izmantojot. Tāpēc valodas attīstībā ir jāpiedalās kompetentiem cilvēkiem, vajadzības gadījumā pieaicinot pat kādu pārstāvi no gaidāmās lietotāju-profesionāļu auditorijas – domēna ekspertu. Eksperta zināšanas nodrošinās valodas atbilstību lietošanas prasībām un palīdzēs izveidot valodu par praktiski pielietojamu eksperta domēna lietošanas vajadzībām.

Visgrūtākais darbs ir izstrādāt domēnspecifisko valodu no nulles jeb pilnīgi neeksistējošām iepriekšējām iestrādēm. Lai varētu veikt šādu izstrādi gan domēnspecifiskās valodas izstrādātājiem, gan ļoti vēlams arī domēna ekspertiem ir jābūt ar priekšzināšanām citu domēnspecifisko valodu izstrādē un jābūt ļoti kompetentiem pašā domēnā. Lielākoties gan eksistē dažādi dokumenti vai ierastās prakses norādes, kuras var izmantot domēnspecifiskās valodas izstrādē. Izmantot gatavas iestrādes, formālos dokumentus vai eksistējošu programmēšanas valodu atmaksājas, jo izgudrot no jauna ir ļoti grūti. Turklāt pārtaisīšanas iespēja, tiklīdz kāda domēnspecifiskā valoda ir sasniegusi noteiktu attīstības posmu, kļūst dārgāka un krietni ilgāka, nekā palikšana pie esošās valodas risinājuma.

Lai gan domēnspecifiskās valodas ir bijušas salīdzinoši nepopulāras un to popularitāte ir krasi augusi tikai pēdējos gados, ir veikti pētījumi, kas analizē domēnspecifisko valodu veidošanas nosacījumus. Izanalizējot trīs dažādu domēnspecifisko valodu izstrādes procesus[7], tiek izvirzīti sekojoši ieteikumi veidojot savu valodu:

- Valodas izstrādē izmantot visus pieejamos jau gatavos formālos aprakstus, kas ir pieejami domēna ekspertiem un neizgudrot šos aprakstus no jauna.

- Iedziļināties un izprast organizacionālās lomas cilvēkiem, kas lietos šo valodu, kā arī jāizprot esošā risinājumu procesa projektu.

- Gandrīz vienmēr domēnspecifiskās valodas projektēšanā netiek projektēta programmēšanas valoda. Jāprojektē ir tikai nepieciešamākās lietas un ir jāizvairās no pārlieku sīkas projektēšanas.

- Jācenšas sasniegt 80% gatavu risinājumu. Ja lielākā daļa no objektiem var tikt apkopota, tad ekspertiem ir pietiekami daudz laika, lai izdomātu kā atrisināt sarežģītākās detaļas, piemēram, izmantojot jau gatavo risinājumu.

- Jāsamazina valodas sarežģītība līdzsvarojot vides detaļas. Jo vienkāršāka valoda, jo mazāka iespēja pieļaut kļūdas.

- Jācenšas panākt un jāgaida lielāka produktivitāte no ekspertiem. Tā kā valoda, kurā rakstīs eksperti, būs vienkāršota un no tās tiks ģenerēts rezultāts, tad ekspertu ieguldījums, rakstot domēnspecifiskajā valodā, jāuztver kā ļoti produktīvs. Papildus, arī eksperti mācīsies izstrādes un, iespējams, iegūs jaunas zināšanas par savu domēnapgabalu.

- Jācenšas pielāgot jaunā tehnoloģija jau ieviestajām tehnoloģijām vai praksēm. Šajā jomā nozīmīga loma ir domēna ekspertam, kas var dot neatsveramus padomus, lai jauno tehnoloģiju neaizmirstu, jo ir grūti mainīt tradicionālās darba metodes un instrumentus.

- Nepieņemt par pašsaprotamu, ka domēna eksperti zinās, ko domēnvaloda mācēs izdarīt viņu vietā jeb saīsinās automatizējamos darbus un ko domēnvaloda nemācēs izdarīt viņu vietā. Ir jāsaprot, ka gala lietotāji – domēna eksperti, būs jāapmāca, ja izveidotā valoda nebūs intuitīva.

Rīki, ar ko var veidot domēnspecifiskās valodas, iedalās divos tipos: grafiskajos un tekstuālajos. Atšķirība starp tiem ir modeļu veidošanas pieeja. Grafiskie rīki ļauj veidot domēnspecifisko valodu izmantojot vizuālus lietotāja interfeisa līdzekļus. Tekstuālie rīki ļauj veidot domēnspecifisko valodu rakstot izteiksmes. Sīkāki piemēri izstrādes rīkiem tiks pieminēti nākamajās nodaļās.

Izvēloties rīku, izšķiršanās starp grafisko un tekstuālo tipu nozīmē izšķirties arī par noteiktiem mīnusiem un plusiem katrai valodai. Grafisku modeļa valodu ir vienkāršāk uztvert, vienkāršāk iemācīties un tajā var viegli orientēties, bet to ir grūtāk uzturēt izstrādātājiem un, ja ir nepieciešams atkļūdot kādu transformāciju, ir daudz grūtāk izsekot no grafiskās vides

līdz tekstuālajam modelim[1]. Attiecīgi, tekstuālā rīka valodā ir pretēji grafiskajai - grūtāk orientēties un iemācīties, taču vieglāk uzturēt.

Piemēri grafiskajiem izstrādes rīkiem: MetaEdit[13], MS DSL Tools[14], GMF[15]. Piemēri tekstuālajiem izstrādes rīkiem: Spoofax[16], StrategoXT[17], oaW[18], MontiCore[19].

3.2. Pamatojums valodas veidošanai

Darba autora praksē bieži nākas saskarties ne tikai ar laika termiņā steidzamiem projektiem, bet arī nākas secināt, ka rakstītais kods bieži atkārtojas un daudz laika no projekta izstrādes aiziet atkārtojot iepriekšējo projektu jau rakstīto kodu. Starp izvēli rakstīt no jauna un nokopēt no cita projekta, šķiet, lielāks produktivitātes ieguvums ir otrajam variantam, taču ļoti bieži projektu prasības atšķiras un šāds apstāklis liek veltīt vairāk laika un pūļu, lai iepriekšējo kodu pielāgotu cita projekta prasībām. Līdz ar to jāsecina, ka abi no minētajiem variantiem nav ideāli.

Programmējot tīmekļorientētu projektu un izmantojot jau gatavus ietvarus un bibliotēkas, daļa no koda rakstīšanas ietaupās, tiesa, paplašinot koda apjoma bāzi un uzticoties citu koda kvalitātei un drošībai. Taču parasti gatavu risinājumu lielākais klupšanas akmens ir kādas specifiskas projekta prasību nianses, kuru risinājums nav tieši atbalstīts izvēlētajā ietvarā vai bibliotēkā. Ja reiz gatava risinājuma nav, tad programmētajam atliek iziet uz kompromisu – realizēt daļu no projekta ar ietvarā eksistējošajām komandām un daļu no projekta rakstīt pašrocīgi, iespējams, arī ietverot jau kādu gatavu algoritma daļu no ietvara pēc iespējas precīzāk nodrošinot prasību izpildi. Jāpiezīmē, ka visbiežāk speciālgadījumi tiek risināti ar jau izmantotām un pārbaudītām metodēm, tātad, atkārtojot iepriekš rakstītu algoritmu.

Autoram uzzinot par domēnspecifisko valodu eksistenci un jau gatavajiem piemēriem – WebDSL[11], radās priekšstats, ka autora problēmsituācijā koda ģenerēšana ir vēl viens no veidiem kā atrisināt neproduktīvo situāciju. Tiesa, ar vienu piegājienu izveidot pilnībā strādājošu risinājumu, kas no apraksta ģenerētu gatavu un lietojamu kodu, nebūtu iespējams, taču, apkopojot pieredzi un domēnspecifisko valodu īpašības, būtu iespējams radīt risinājumu, kas veiktu standarta bloku ģenerāciju, lai topošajam projektam būtu ielikts strādājošs pamats.

No izveidotā pamata būtu iespējams turpināt attīstīt projektu izmainot detaļas un pievienojot trūkstošās funkcijas.

Pieeja radīt domēnspecifisko valodu no jauna pamudināja piemērota rīka eksistence autora darba videi. Eksistējošā WebDSL valoda nodrošināja koda ģenerāciju Java programmēšanas valodā, lai gan autors strādā ar PHP valodu, bet šai valodai gatava domēnspecifiskā valoda nav iztaisīta. Turklāt arī bez ģenerētās valodas, autors vēlējās iztaisīt risinājumu, kuram būtu iespējas adaptēt dažādas papildfunkcijas – piemēram, atšķirt tīmekļa projekta stila līmeni no koda līmeņa, lai varētu pēc izvēles un pēc iespējas vienkāršāk darboties ar tīmekļa vietnes vizuālo atveidojumu.

Radītā domēnspecifiskā valoda būtu kā piemērs, kā domēnspecifiskas valodas veidošana var atvieglot programmētāja darbu, veidojot pēc struktūras līdzīgus projektus un ietaupot resursus, rakstot vienu un to pašu kodu. Veidojamā valoda, iespējams, nepretendētu uz valodu, kas būtu plaši lietojama, toties tā atspoguļotu paša programmētāja radītās pieredzes akumulāciju un produkta radīšanu ar šīs zināšanu bāzes palīdzību, lietojot programmētājam saprotamus valodas un arhitektūras principus.

4. RĪKA IZVĒLE VALODAS IZSTRĀDEI

Lai sāktu izstrādāt valodu, ir nepieciešams rīks ar ko sākt izstrādes procesu. Izvēloties rīku, jāņem vērā, ka izvēlētā varianta nianses ļoti spēcīgi ietekmēs izstrādājamo domēnspecifisko valodu. Ja rīks būs liela un spēcīga programmatūras ražotāja produkts, tad var sagaidīt, ka tam būs liels palīdzības atbalsts, taču būs arī ierobežojumi no ražotāja. Savukārt, ja rīku izstrādās maza cilvēku grupiņa, var gadīties, ka tas ir atvērtā koda risinājums bez ierobežojumiem, taču var trūkt lietošanas dokumentācija un pats rīks var izzust pēc pāris nedēļām.

4.1.Tabula.

Izstrādes rīku salīdzinājuma tabula.

Rīks	**Vide**	**Licence**	**Problēmas**
MPS	MPS + Java	Open source + Apache 2.0	Grūti apgūt, nav standarta redaktora
OOMEGA	Java + Eclipse	Open Source + Dual	Tikai modeļu interpretācija
Rascal	Java + Eclipse IMP	Open Source	Daļējs IDE atbalsts
Spoofax	Java + Eclipse	Open Source	Nav integrēts EMF
Essential	Net 4.0	Evaluation	Tikai modeļu interpretācija, licence
Xtext	Java + Eclipse EMF	Open Source (EPL)	Nepieciešams daudz koda, lai radītu lietojamus valodas rīkus
EMFText	Java + Eclipse EMF	Open Source (EPL)	Radās iespaids, ka ir sarežģīta programmēšana

Tāpēc par rīku salīdzināšanas kritēriju var izmantot dalību „Valodu rīku izaicinājumā”[23], kas notiek kopš 2011. gada. Minētais valodu rīku sacensības pasākums ir domāts, lai salīdzinātu vairākus domēnspecifisko valodu izstrādes rīkus un noskaidrotu, kurš rīks ir labākais. Par labāko rīku tiek saukts tāds rīks, kas spēj tikt galā ar uzdevumu un dalībniekiem nav bijušas grūtības izstrādājot šo risinājumu. Protams, ka pats labākais rīks netiek nosaukts, taču sacensība dod ieskatu vairāku rīku salīdzinājumam pēc to platformām, ierobežojumiem, trūkumiem, kā arī to izstrādātājiem parāda virzienu, kurā būtu vēlams

attīstīties. Valodas izstrādes rīka dalība sacensībā arī parāda, ka rīks ir pietiekami populārs, lai kāds arī viņu censtos reāli izmantot vai reklamēt.

Izmantojot „Valodu rīku izaicinājuma" tīmekļa vietnes informāciju, var iegūt salīdzināmo materiālu par vairākiem no rīkiem. Tabulā 4.1. parādīti piedāvātie salīdzināmie domēnspecifisko valodu izstrādes rīki[23], taču izaicinājuma vietnē var aplūkot plašāku sarakstu. Autors savas valodas izstrādes projektu vēlas veidot ar tekstuālu, nevis grafisku rīku, tāpēc daļu no salīdzināmajiem rīkiem varēja jau atmest sākumā. Priekšroka tekstuālam DSL tika dota, jo autoram šķiet svarīgāk rakstīt kodu, nevis iegūt iespēju grafiski redzēt koda sakarības, kas ir grafiskas domēnspecifiskās valodas priekšrocība.

Komerciālā licence būtu otrs faktors, kuru autors atmet, jo projekta budžeta ietvari neļauj izmantot maksas programmatūru. Lai gan komerciālie produkti piedāvā lielāku kvalitāti, autoram šķiet izaicinošāk izmēģināt produktu, kuru ir sagatavojušas personas savas intereses dēļ. Komercālās licences faktora dēļ no salīdzinājuma saraksta bija jāizņem OOMEGA un Essential.

Izpētot atlikušo negrafisko domēnspecifisko valodu izstrādes rīku piedāvājumu, par piemērotāko darba uzdevuma veikšanai tika izvēlēts Spoofax. Pārējie tuvākie kandidāti bija Xtext un EMFText. Izvēlēties Xtext atturēja piezīme, ka trūkst vairāki rīki, kuri jāprogrammē pašam, lai iedzīvinātu šī rīka izstrādātās valodas lietošanu. EMFText tika izmēģināts praktiski, taču pirmais iespaids par šo rīku nebija pozitīvs. Spoofax valodas sintakse subjektīvu apsvērumu dēļ šķita uztveramāka nekā EMFText. Pārējiem tabulā 4.1. minētajiem kandidātiem aplūkotie trūkumi tika novērtēti kā krietni būtiskāki mīnusi.

Vienīgais Spoofax minētais vērā ņemamais mīnuss bija integrācijas trūkums ar EMF jeb Eclipse rīka modeļu ietvaru. Darba autors uzskata, ka integrācijas trūkums nav būtisks, jo Spoofax ir atšķirīgs risinājums un integrēšanās ar EMF nav jābūt obligātam nosacījumam.

Spoofax ir 2007. gadā sākts projekts, kas apvieno sevī SDF[20] gramatikas un Stratego[17] transformācijas iekš vienota Eclipse izstrādes rīka[24]. Eclipse ir gan domēnspecifiskās valodas izstrādes vide, gan arī turpmākā radītās valodas izstrādes vide. Ja neskaita konkrētu piesaisti vienai videi, valodas izstrādātājam ir ērti neuztraukties par šīs valodas darba vides turpmāku izstrādi. Gan koda redaktors, gan nepieciešamie kompilatori var atrasties vienuviet, pat tā, ka valodas rakstītājs neko par tiem nezinās, bet strādās tikai ar koda rakstīšanu un rezultāta saņemšanu.

Spoofax ir izstrādes videi ir arī vairākas problēmas. Viena no tām saistās ar atkļūdošanu. Piemēram, pieļaujot kļūdu transformācijas loģikā vai neuzrakstot kāda koka elementa transformāciju pievienotajā līmenī, lietotājs izvadā nesaņem viegli interpretējamu problēmas aprakstu, kā vien java klases kļūdas, kas nesniedz lietderīgu informāciju par kļūdas vietu un saturu. Līdz ar to veicot izstrādi, bija lietderīgi doties uz priekšu lēniem soļiem, pēc katriem sīkākajiem labojumiem cītīgi pārkompilējot visu valodas rīku. Taču kopumā Spoofax ir iebūvēta palīdzība ar koda atkļūdošanu. Valodas gramatiskā sintakse pakļaujas vizuālai kļūdu labošanai, jo rīks iekrāso nepareizi konstruētu gramatisko sintaksi. Arī transformācijas deklarācija pakļaujas šādiem vizuāliem labojumiem.

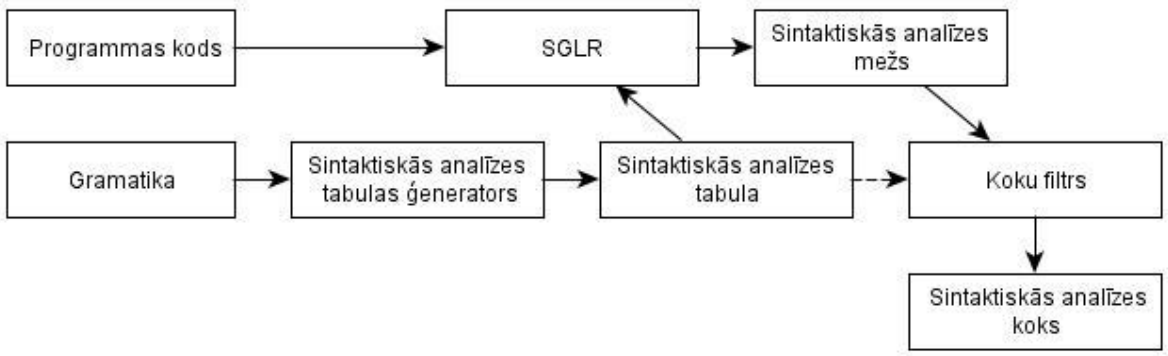
5. IZVĒLĒTĀ RĪKA ĪPAŠĪBAS

Šajā nodaļā aprakstīts kā ar Spoofax rīku tiek izveidota domēnspecifiska valoda, demonstrējot Spoofax komponentu īpašības. Spoofax sastāv no trīs svarīgiem komponentiem – SDF gramatikas, Stratego/XT transformāciju valodas un Eclipse izstrādes vides. Ar izvēlētā rīka palīdzību, apvienojot visas tā komponentus galā tiek radīta gatava valodas vide ar redaktoru, kas atbalsta koda automātisku pabeigšanu un sintakses izcelšanu, valodu, kas atspoguļo autora problēmdomēnu – tīmekļa vietņu izstrādi, un ģenerācijas procesu, kas no uzrakstītās valodas rada kodu PHP valodā.

5.1. Gramatikas definēšana

Izvēlētajā izstrādes rīkā Spoofax valodas gramatika tiek definēta ar Sintakses Definēšanas Formālismu jeb SDF. SDF tiek uzskatīta par labāku gramatiku nekā BNF un Yacc, ar ko tā tiek salīdzināta kā sintaktiski bagātāka, modulārāka, deklaratīva valoda[20]. Programmētājam ir iespēja uzrakstīt jebkuru gramatiku ar SDF palīdzību augstā līmenī modelējot gan eksistējošas valodas, gan izgudrotas, gan arī apvienot tās vienā valodā.

Eksistē vienots, iepriekš sagatavots rīku komplekts, kas nolasa un tulko definēto SDF gramatiku. Tāpēc SDF ir priekšrocības pār tām gramatikām, kurām nepieciešami vairāki atsevišķi rīki un, sekojoši, arī vairāki pārmaiņu stāvokļi, lai ģenerētu gramatiskās struktūras. Definēšana notiek gan leksiskā, gan bezkonteksta gramatikā, ieskaitot arī neskaidru definīciju ģenerēšanu. Ar neskaidrām definīcijām tiek domāti stāvokļi, kad gramatikā ar nolūku vai kļūdas pēc var rasties vairāki iespējami gramatikas nolasīšanas varianti.



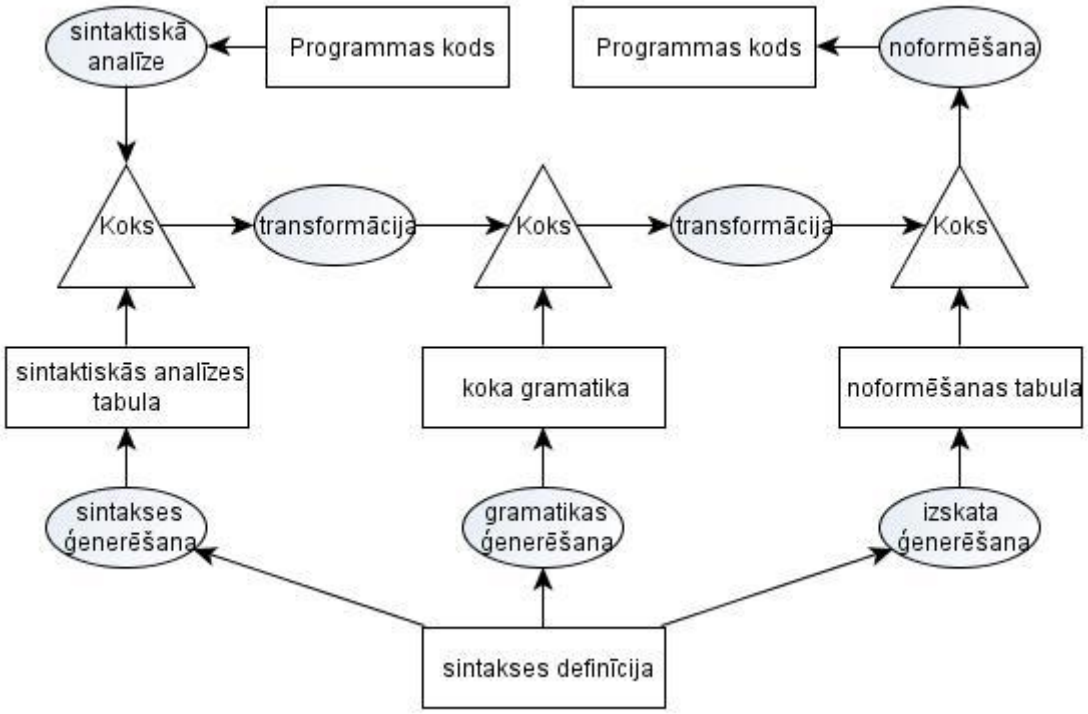
5.1.1. Attēls. **SDF darba plūsma**

SDF gramatika ļoti elastīgi atļauj definēt gan konkrētus komandu nosaukumus, gan arī izmantot jebkādus nosaukumus, iegūstot no tiem izmantojamus parametrus. Valodas parametri var pārklāties, tādējādi ietaupot koka apraksta veidošanai paredzētās koda rindiņas. Iespējamas arī dažādu nosacījumu izmantošanu gramatikas veidošanā.

Attēlā 5.1.1. redzams SDF gramatikas dzīves cikls[21]. Tas sākas ar sintaktiskās analīzes tabulas ģenerēšanu, ņemot lietotāja definēto gramatiku un apvienojot to vienotā definīcijā – visas definētās komandas apvienotas vienā vietā, kas tiek turpmāk izmantota izvadkoka ģenerēšanā. Sintakšu definīcijas tiek pārbaudītas, meklējot triviālas definīcijas kļūdas, normalizācijas procesā no gramatikas tiek izvāktas visas sīkkomandas un papildnosacījumi. Rezultāts ir sintaktiskās analīzes tabula SLR (Simple LR) formātā. Ar SLR tabulu sintaktiskās analīzes tabulu rīks SGLR[25] var sintaktiski analizēt lietotāja doto tekstu, radot sintaktiskās analīzes mežu, kas caur filtru rada sintaktiskās analīzes koku. Galarezultātā iegūtā koka datus tālāk var izmantot citi rīki.

5.2. Koda transformācija

Spoofax koda transformācijai izmanto Stratego/XT ietvaru. Komplektu veido divas daļas – Stratego un XT. Stratego ir valoda, lai aprakstītu transformācijas loģiku un XT ir rīku komplekts, kas palīdz analīzē un izvada noformēšanā. Stratego/XT ir paredzēts dažāda veida koda transformācijām, par ievadu ņemot izejas koda avotu un, pielietojot transformācijas noteikumus, iegūst, piemēram, koda arhitektūru vai pārveido kodu atbilstoši dotajiem nosacījumiem – veic izmaiņas kādās koda darbībās vai pat pilnībā pārraksta kodu kādā citā programmēšanas valodā[22].



5.2.1. Attēls. **Stratego/XT darba plūsma**

Stratego izmanto kokus un SDF gramatiku, lai darbotos ar kodu. Kā tas ir parādīts 5.2.1. attēlā, no ievadītā programmas koda pēc analīzes tiek veiktas divas transformācijas, pēc kurām noformēto kodu izvada kā rezultātu. Lai tas sekmīgi funkcionētu, ir jābūt nodefinētai sintakses definīcijai par to kā šo ievadu transformēt uz izvadu. No viena ievada – SDF gramatikas un transformāciju apraksta – tiek iegūti trīs datu objekti: sintaksei, gramatikai un noformējumam, lai veiktu koda izmaiņas.

5.3. Automātiska izstrādes vide

SDF un Stratego/XT ir apvienoti zem viena nosaukuma – Spoofax, bet Spoofax izstrādes vide ir Eclipse. Izstrādājot savu valodu ar Spoofax, par valodas izstrādes vidi automātiski var iegūt Eclipse. Tas ir liels ieguvums, jo ne visi domēnspecifiskie valodu būves rīki savam rezultātam automātiski nodrošina arī grafisku risinājumu, kas atbalsta tādas funkcijas kā koda automātiska turpināšana, kļūdu izcelšana un koda iekrāsošana atbilstoši definētajiem valodas uzstādījumiem.

Protams, Eclipse var būt arī nevēlams risinājums izstrādes videi, taču tādā gadījumā nākas pašam veidot izstrādes vidi, kas iekļauj visus Spoofax rīkus. Spoofax ļauj arī turpināt darbu ar ģenerēto kodu Eclipse redaktorā, pielāgojot to jaunās domēnspecifiskās valodas lietošanai. Izmantojot Eclipse kā izveidotās domēnspecifiskās valodas redaktoru tiek iegūtas tās pašas papildus iespējas, kas pie domēnspecifiskās valodas izstrādes – koda automātiska turpināšana, kļūdu izcelšana un koda iekrāsošana. Izstrādājot domēnspecifisko valodu ir iespējams uztādīt nākamā koda vizualizācijas parametrus, ko izmantos Eclipse.

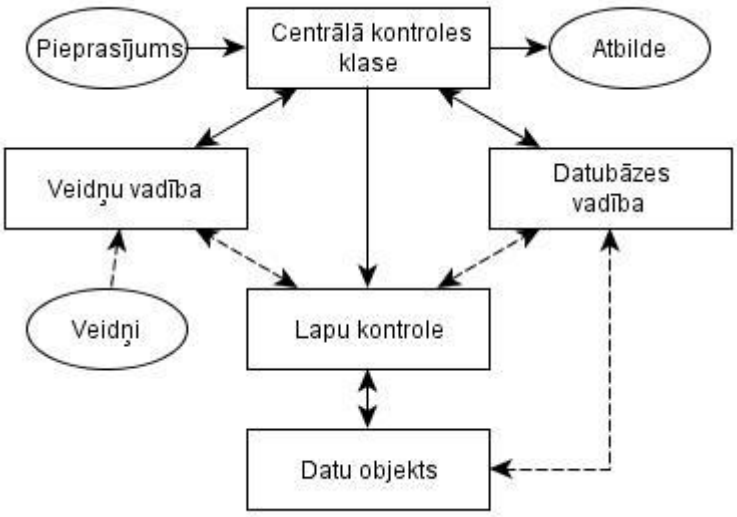
Eclipse iekļaujas arī vairākas citas izstrādes platformas – Java, C++, PHP. Eksistē arī izveidoti modeļu veidošanas ietvari – IMP, EMF, taču Spoofax nesadarbojas ar tiem.

6. VALODAS IZSTRĀDE

6.1. Valodas arhitektūra

Valodas veidošana tika sākta ar pēdējo taisīto projektu izvērtēšanas un līdzīgo algoritma iezīmju meklēšanu. Vairāki no pieredzes laikā realizētajiem projektiem ir saturējuši līdzīgas strukturālās iezīmes, kas ir ļāvis identificēt kopsaucējus. Šiem kopsaucējiem ir jākļūst par nākamās veidojamās valodas sintakses parametriem, lai domēnspecifiskā valoda spētu tos automātiski uzģenerēt.

Analizējot praksē realizētos projektus, tika secināts, ka, pamatā kopsaucēju veidojošie projekti, sākas ar "uzdevumu pārvaldnieku", kas organizē visa projekta darbu un sadala izsauktos uzdevumus pa konkrētu uzdevumu klasēm. Uzdevumu klases ražo attēlojamo saturu un izsauc darbību klases, kas iegūst konkrētus datus. Attēlojamais saturs tiek veidots ar veidņu palīdzību, ko vada veidņu pārvaldnieks. Lapu kontroles jeb darbību klases atbild par datu objekta veidošanu un pārvaldīšanu, cik tas attiecas uz izveidošanu, saglabāšanu un izmainīšanu. Projektos bieži šīs te uzdevumu klases tiek apvienotas vienā - tīmekļa sadaļa izsauc vienotu funkciju, kas ir datu vadības un grafiskā izvada apvienojums. Ja vienkāršiem tīmekļa projektiem, kas sastāv no dažām saskarnēm, šāda pieeja noder, tad lielākos projektos ir grūti turpināt uzturēt projektu un nodrošināt datu integritāti vairākās saskarnēs. Līdz ar to potenciāli apvienojamās daļas ir nepieciešams atdalīt, lai varētu tās gan vienkāršāk uzturēt, gan arī vienkāršāk atkļūdot.



6.1.1. Attēls. **Tīmekļa vietnes struktūra**

Aprakstītie elementi attēloti 6.1. attēlā, kur tiek ilustrēta pieprasījuma saņemšana no klienta, pēc kuras tīmekļa vietne parasti izmanto vienu centrālo kontroles klasi jeb „uzdevumu pārvaldnieku", lai izsauktu atbilstošo definēto darbību, kas arī atbildēs klientam ar noformētu atbildi. Lai kontroles klase varētu izsaukt darbību un noformēt atbildi, tiek izmantotas papildklases – veidņu, datubāzes un konkrētās darbības vai „lapas" vadības moduļi. Šie papildmoduļi tiek veidoti kā citas klases, kuras tiek izsauktas obligāti vai vajadzības gadījumā.

Veidņu vadības modulis nodarbojas ar veidņa ielādi un datu ievietošanu veidnī. Ar veidni tiek saprasts noformējums HTML aprakstā, kuru saņem vietnes pieprasītājs atkarībā no sava pieprasījuma. Šajā veidņu sistēmā bez HTML sintakses apraksta, tiek paredzētas arī atsevišķas ievadvietas, kurās var ievietot dinamiskus datus – tekstu, citu veidni vai arī tekstu kopu. Veidojamā domēnspecifiskā valoda gan neveidos HTML lapu aprakstus, bet tikai tīmekļa vietnes algoritmus valodā PHP. Šāds solis ir apzināts, jo vietņu vizuālais noformējums tiek izveidots atsevišķi un vietnes loģiskajam kodam ir tikai jāsastrādājas ar šo vizuālo noformējumu. Nodalot vizuālo no loģiskā, tiek iegūta brīvība izdarīt izmaiņas un noformēt vietni.

Datubāzes vadības modulis nodrošina sadarbību starp vietnes datiem un ārēju datubāzes dzini. Elementārākās tīmekļa vietnes visbiežāk izmanto kādu konkrētu datubāzes dzini, taču tam nevajadzētu būt nozīmīgam faktoram. Vietnes veiktos datubāzes pieprasījumus var noformēt atbilstoši, lai nebūtu nozīmes dziņa veidam. Pārējie vadības moduļi pieprasa informāciju no datubāzes, izmantojot vienotu pieprasījumu izsaukšanu, kurā tiek noformētas datubāzes pieslēgšanās tehniskās nianses.

Darbības vai lapas vadības modulis atbilst katram konkrētam pieprasījumam kādai sadaļai. Tīmekļa vietnei mēdz būt dažādi pieprasījumi – vienkārša saskarnes atlase vai saskarnes atlase ar datu pieprasījumu. Datu pieprasījums nozīmē, ka iepriekšējā redzamā saskarnē lietotājs ir ievadījis kādu informāciju, kuru nosūta apstrādei uz tīmekļa vietni. Ja vien šāda informācijas nosūtīšana nenotiek slēpti, kā tas ir gadījumos ar kādu izveidotu kodu automātiskai periodiskai informācijas apmaiņai, pēc datu pieprasījuma lietotājam tiek parādīta cita satura saskarne. Tiek pieņemts, ka ar datu ievades, datu organizēšanas un datu izvades funkcijām nodarbojas darbības vadības modulis.

Lai darbības vadības modulis spētu apstrādāt datus, tiek ieviests datu organizēšanas modulis ar ko sadarboties. Izdalīšanos no vadības moduļa nosaka fakts, ka ar vienu un to pašu datu kopu var nodarboties vairāki darbības moduļi, piemēram, ievades, labošanas, meklēšanas

darbības. Datu organizēšanas modulis nodarbojas ar datu organizēšanu, iegūstot informāciju no datubāzes, nodrošinot piekļuvi no darbības vadības moduļiem un saglabājot informāciju atpakaļ datubāzē, ja tā tiek izmainīta. Ar šādu centralizētu datu saglabāšanu, tiek arī vienkāršota pārbaužu eksistence un izmaināmās vērtības iespējams pārbaudīt, piemēram, pret injekciju esamību.

No uzskaitītajām valodas detaļām gala iznākumā dažas no komponentēm būs statiskas jeb atsevišķu klašu ģenerācija tiks aizstāta ar vienkārša teksta iekļaušanu. Minētās komponentes ir veidņu vadība un daļēji arī datubāzes vadība. Komponentes nav jāģenerē, jo tās ir kā bibliotēkas, lai pārējās komponentes darbotos - to saturs ir statisks. Satura ģenerācija notiek elementiem, kuru saturs ir mainīgs – centrālās vadības kontroles, lapu kontroles un datu objektiem. To saturs sastāv no datiem, kurus raksta domēnspecifiskās valodas lietotājs – mainīgie, objekti, to mijiedarbības. Datubāzes vadības klases vienīgie ģenerējamie parametri ir pieejas dati kā lietotāja vārds, parole un datubāzes dziņa tips.

6.2. Valodas izteiksmes

Balstoties uz apkopoto informāciju par tipisko tīmekļa vietnes uzbūvi un elementu attiecībām, tika izlemts, ka valodas struktūra būs vairāku kategoriju objekti, katrs ar saviem parametriem. Aprakstot attiecības starp šiem objektiem tiks iegūta daudzšķautņains tīmekļa vietnes apraksts ar dažādiem ģenerējamiem parametriem. Valodas struktūra atgādinās iepriekš raksturoto vietnes uzbūvi, jo, nodefinējot vajadzīgos vadības moduļus, iespējams arī uzģenerēt atlikušo vietnes kodu.

6.2.1. Kontroles klase

Galvenais un nozīmīgākais objekts ir visa projekta aprakstošais objekts, kas tiek nosaukts par "project". Šajā objektā tiek aprakstīti nozīmīgākie parametri: datubāzes tips un pieslēgšanās, ielādējamās viednes sadaļas, veidņu sistēmas iestatījumi. Balstoties uz šo informāciju būs iespējams noģenerēt datubāzes pieslēgšanās klasi, veidņu ielādes klasi un centrālo klasi, kas atbildēs par vietnes sadaļu ielādes kontroli.

```
1    "project"   ID   "{"   {WebProperties   ","}*   "}"   Blocks*   ->   Start   {
     cons("WebProject") }
2    "title" ":" STRING -> WebProperties{ cons("CommonTitle") }
3    "pages" "{" {PageElements ","}* "}" -> WebProperties{ cons("WPages") }
4    "template" STRING "{" {TemplateProperties ","}* "}" -> WebProperties{ cons
     ("BasicTemplate") }
5    "database" ID "{" {DBElems ","}* "}" -> WebProperties{cons("Database")}
6    ID -> PageElements{cons("WPage")}
7    ID ":" "default" -> PageElements{cons("PageDefault")}
```

Minētajā piemērā SDF sintaksē aprakstīts valodas sākuma sintakse. Tiek sagaidīts, ka pašā minimālākajā projekta definīcijā būs uzrakstīts programmas bloks „project" ar nosaukumu. Par vēlamajiem sintakses datiem tiek uzdoti projekta parametri un citu klašu bloki, dotajā sintaksē apzīmēti, attiecīgi, ar „WebProperties" un „Blocks".

„WebProperties" blokā iespējamie definējamie parametri ir četri – „title", „pages", „database" un „template". „Title" parametrs deklarē tekstu, kas noklusēti rādīsies tīmekļa vietnes nosaukumā. „Pages" parametrs definē objektus un vienlaicīgi arī darbības, uz ko reaģēs kontroles klase, ienākot klienta pieprasījumam. „Template" ir sākums veidņa deklarācijai, kas tiks ielādēts attēlojot vietni. Vietnes definēšana iepriekšminētajā vietā nozīmē definēt pamatveidni, kas attēlosies katru reizi pieprasot attēlot vietni un tiks izmantots par pamatveidni visām darbībām. „Database" ir definējams datubāzes parametru bloks, kas ļaus uzģenerēt datubāzes piekļuves objektu.

„Pages" iespējamie apakšelementi ir divi – vienkāršs identifikators un identifikators ar noklusējuma atzīmi. Atšķirība ir tā, ka ar noklusējuma atzīmi tiek ielādēts, ja nav kāds konkrēts sadaļas pieprasījums jeb standarta lapa, ko lietotāji redz atveroties sākot darbu ar tīmekļa vietni.

```
pages {
        Flowers,
        Bushes,
        Intro : default,
        Trees
}
```

Minētajā koda piemērā parādīts, ka eksistē četri lapu objekti, no kuriem „Intro” tiks ielādēts pēc noklusējuma.

6.2.2. Veidņa parametru bloks

Lai gan veidņu ielāde ir atsevišķa klase un tiek uztverta kā atsevišķa papildus darbība, valodā tā tiek definēta tikai vadības klases un lapu klašu ietvaros kā apakšobjekti. Veidņa parametru apraksts tiek pārģenerēts inicializācijas funkcijā, kas izsauc veidņa klasi tā objekta ietvaros, kurā veidnis ir definēts, izmantojot definētos parametrus.

```
1    "template" STRING "{" {TemplateProperties ","}* "}" -> Virsobjekts
2    ID "->" ID -> TemplateProperties{cons("TmplID")}
3    ID "->" "self" ID ->TemplateProperties{cons("TmplIDS")}
4    ID "->" STRING -> TemplateProperties{cons("TmplStr")}
5    ID "->" -> TemplateProperties{cons("TmplEmpty")}
6    ID "->" "list" "{" {TemplateListProp ","}* "}" -> TemplateProperties{
     cons("TmplList")}
7    "(" {TemplateProperties ","}* ")" -> TemplateListProp{}
```

Veidņa komandu bloks sastāv no iespējamām ievietošanas variācijām. Veidņa nosaukums, definēts teksta formātā, apzīmē veidņa datni, kurā glabājas manipulējamais izvada apraksts. Izvada aprakstā tiek ievietoti variāciju parametri, kur identifikators apzīmē nomaināmo tekstu veidņa datnē. Eksistē divu veidu nomaināmie parametri – vienkāršie un paplašinātie. Vienkāršie nomaiņas parametri nomainās 1:1, jeb ierakstītais teksts tiek ievietots veidnī. Paplašinātie nomaiņas parametri nozīmē vairākrindu ievadi, var definēt, piemēram, vairākas tabulas rindiņas. Tas tiek apzīmēts ar „TemplateListProp” saikni, kas ved atpakaļ pie parametru koka sākuma, tādējādi padarot parametru aprakstu arī par savu apakšparametru aprakstu. Vienkāršie definētie parametri arī iedalās, kā redzams ir ID->STRING un ID -> self ID. Šādi tiek norādīts, ka ievadāmais parametrs nav vienkārši teksts, bet gan mainīgais, kas jāielasa veidnī no konkrētā datu objekta mainīgajiem.

```
template „example.html” {
        titleelement -> self title,
        textelement -> „Example text”
}
```

Minētajā piemēra aprakstītas divi veidņa parametri, no kuriem „titleelement” tiks ielādēts no vērtības, kas ir uzstādīta konkrētajā klasē, savukārt „textelement” būs konstants teksts „Example text”.

6.2.3. Datubāzes parametru bloks

Apakšparametru bloks kontroles klasei, kas atbild par pieslēguma izveidošanu datubāzei. Definētā informācija tiek izmantota, lai uzģenerētu pieslēgšanos ar dotajiem parametriem. Datubāzes parametru bloks tiek norādīts tikai viens – kontroles bloka sākumā. Pastāv iespēja arī atļaut izmantot šo bloku citās vietās, piemēram, datu objektos, kas atļautu ģenerēt datu atlasi arī no citām datubāzēm. Taču šāds projekts jau ir uzskatāms par sarežģītu un tipiskajā variantā datu glabāšana notiek vienā datubāzē.

```
"database" ID "{" {DBElems ","}* "}" -> WebProperties{cons("Database")}
"user" ":" STRING -> DBElems{cons("DBUser")}
"pass" ":" STRING -> DBElems{cons("DBPass")}
"db" ":" STRING -> DBElems{cons("DBName")}
"host" ":" STRING -> DBElems{cons("DBHost")}
"port" ":" STRING -> DBElems{cons("DBPort")}
```

Kā redzams SDF gramatikas piemērā, tiek pieņemti vairāki konstanti parametri – lietotājvārds, parole, datubāzes vārds, pieslēguma serveris un pieslēguma ports. Datubāzes veids tiek definēts kā identifikators inicializējot „database” objektu. Atkarībā no šī identifikatora tiek veidota datubāzes klase, kas tiek uzģenerēta atbilstoši norādītajam datubāzes dzinim.

```
database mysql {
        user : „admin”,
        pass : „@dm1n”,
        db : „example_db”,
        port : „3012”,
        host : „127.0.0.1”
}
```

Redzamajā piemērā ir datubāzes objekta definīcija, kas tiek ierakstīta kontroles klases ietvaros. Tiek nodefinēti visi nepieciešamie ievadinformācijas lauki.

6.2.4. Datu objekts

Datu objekts, nosaukts par "dataobject", apraksta mazu ielādējamo datu domēnu. Klase, izmantojot iepriekš ģenerēto datubāzes moduli, nodrošinās datu lauku ielādi un izmainīto datu saglabāšanu datubāzē.

```
"dataobject" ID "{" {DataProperties ","}* "}" -> Blocks{cons("DataObject")}
"fields" "{" {FieldObjects ","}* "}" -> DataProperties{cons("DataFields")}
"table" ":" STRING -> DataProperties{cons("DataTable")}
ID ":" ID -> FieldObjects{cons("DataField")}
```

Būtiskākie parametri ir ierakstīti datu objekta „fields” apakšparametros. Parametru koks „DataProperties” ir apvienots gan datu objektam, gan arī darbību objektam, tāpēc tam ir vairāk apakškopas elementu, nekā vajadzīgs. Datu objekts izmanto tikai apakškoka parametru attiecību ID:ID. Ar to tiek apzīmēti izmantojamo datu lauki, kuri tiks apstrādāti un pieejami darbību objektiem.

```
dataobject FElement{
```

```
        table : „tb_flowers”,
        fields {
                id : integer,
                name : string,
                color: string
        }
}
```

Minētajā piemērā ir redzams datu objekta apraksts, kur tiek pateikta visa nepieciešamā informācija, lai varētu uzģenerēt klasi, kas būtu spējīga izveidot funkcijas datu nolasīšanai, apstrādāšanai un saglabāšanai.

No šiem dotajiem datiem tiek ģenerēti vaicājumi uz datubāzi SQL pieprasījumu veidā. Taču apzinoties ģenerēšanu un iespējamo datubāzes struktūru, ģenerācija ar nolūku notiek nepilnīgi. Nepilnīgi tādā ziņā, ka ar nolūku tiek plānots, ka datu objektā esošie dati var būt sadalīti, piemēram, vairākās datubāzes tabulās. Veidotā domēnspecifiskā valoda neģenerē datubāzes struktūru, tādējādi atļaujot datubāzes struktūru veidot programmētājam vai kādai citai domēnspecifiskajai valodai. Tas arī nozīmē, ka ir diezgan grūti, autora prāt, nevajadzīgi, censties precīzi deklarēt atlasāmo lauku atrašanās vietu. Tiek paredzēts, ka SQL pieprasījumi tiks veidoti un laboti pēc ģenerācijas ar roku. Attīstot valodu nākotnē, protams, tiks izvērtēta iespēja domēnspecifiskajā valodā precīzi deklarēt tabulu attiecības un tādējādi pilnīgi ģenerēt SQL pieprasījumus.

Lielu nozīmi arī dod pareiza datu tipa deklarācija, kas atļauj lielākos datu objektos ietaupīt daudzu pārbaužu rakstīšanu, jo tās tiek uzģenerētas. Piemēram, pārbaude vai ievadītais mainīgais ir numurs.

6.2.5. Lapas vai darbību objekts

Atspoguļo saskarni ar konkrētu tās darbību. Izmanto principu, ka vienas saskarnes ietveros noris viena darbība, kura tad arī tiek apstrādāta. Par izvaddatiem rūpējas veidņu objekti, par datiem rūpējas datu objekti – darbību objekts nodrošina sadarbību starp tiem.

```
1     "page" ID "{" {PageProperties ","}* "}"   -> Blocks{cons("Page")}
2     ID -> PageProperties{cons("Prop")}
3     "template" STRING "{" {TemplateProperties ","}* "}" -> PageProperties{
      cons("Template")}
4     "form" ID "(" {Formproperties ","}* ")" -> PageProperties{cons("PageForm")}
5     "template" STRING "{" {TemplateProperties ","}* "}" -> Formproperties{
      cons("Template")}
6     ID ":" STRING -> Formproperties{cons("FormProp")}
7     ID ":" ID -> Formproperties{cons("FormStatus")}
8     "fields" "{" {FieldObjects ","}* "}" -> Formproperties{cons("FFields")}
9     "success" ":" "load" ID -> Formproperties{cons("SuccessLoad")}
10    "fail"   ":"   "string"   STRING   ->   Formproperties{cons("FailMessage")}
11    STRING "->" ID "::" ID -> FieldObjects{cons("FormLinks")}
```

Sastāv no vairākiem komponentiem. Veidni var nodefinēt gan pašai darbībai, izmantojot šo objektu tikai kā statistu saskarni, gan arī formas elementam, kas nodrošina pieprasījuma veidošanu un apstrādi. Formas elementam var definēt apakšparametrus „fields”, kas norāda uz sasaisti ar datu objektu. Formas apstrādes darbības veiksmīgas izpildes rezultātā tiek ielādēts cits darbības objekts vai arī kļūdas gadījumā tiek izvadīts kļūdas paziņojums.

```
page Flower {
        form addflower {
                template „addflower.html” {
                        fields -> list {
                        (rowtitle->„Name”,rowname->”fieldFName”,rowval->),
                        (rowtitle->„Color”,rowname->”fieldFColor”,rowval->)
                        },

                        Formname->addflower,
                        Formmethod->post,
                        Formaction->
                },
                button-> „Add”,
                fields {
                „FName”->FElement:name,
                „FColor”->FElement:color
```

```
                },
                success : load otherPage,
                fail : string „Failed to add”
        }
}
```

Redzamajā piemērā aprakstīta lapas kontrole, kurā eksistē ievadforma ar diviem datu laukiem, kas ir savienoti ar datu objektu un tā datu laukiem. Ievadot informāciju, dati tiek nosūtīti uz datu objektu, kur tiek veikta saglabāšana datubāzē.

6.3. Valodas transformācijas

Ar Stratego valodas palīdzību tiek nodefinētas transformācijas no koka uz kodu. Spoofax atļauj veidot daudzslāņainas transformācijas, kas ļauj valodas ģenerācijā iesaistīt vairākas pseidovalodas, kura katra var tikt pārģenerēta uz jaunu slāni. Konkrētā darba ietvaros minētā funkcionalitāte netika pielietota, jo pēc autora domām, pilnīgi pietika arī ar vienu transformācijas līmeni. Tomēr, valodai attīstoties, pastāv iespēja papildināt ģenerējamos līmeņus vai izmainīt kodu.

Stratego atļauj ģenerēšanas procesā veikt arī papildus darbības, kā, piemēram - pārbaudīt sintakses korektumu un kļūdas gadījumā pārtraukt ģenerēšanu. Tādējādi ir iespējams uzstādīt, piemēram, obligātos un neobligātos parametrus.

Transformāciju ģenerēšanas galvenā stratēģija bija no ienākošās sakārtotās informācijas uzģenerēt atbilstošu kodu. Ieejas dati ir A-Term koks, kas tiek iegūts no gramatikas ievades analīzes. Balstoties uz šo koku, Stratego valodā tiek nodefinētas pārejas, kādas ir jāizpilda katram koka elementam.

Koda ģenerācija, atbilstoši arhitektūras plānam, katram elementam ir ar dažādu grūtības pakāpi. Nepieciešams noģenerēt galveno klasi, tās izsaukšanu, atbilstošās papildklases kā veidņu un datubāzes klasi, kā arī lapu klases un datu objektu klases. Veidņu un datubāzes klašu ģenerācija ir elementāra, jo nepieciešams nodefinēt minimālu koka datu pārtulkošanu. Šajā gadījumā veidņu pamata klases apraksts sastāv no iepriekš uzrakstīta koda, kas tiek

vienkārši pievienots klāt pārējam kodam, tādēļ nekāda īpašā transformācija šeit nav nepieciešama. Veidņu pamata klase gan nav saistīta ar veidņu inicializāciju kodā, kas gan tiek ģenerēta katrā atsevišķā izsaukuma vietā.

SDF gramatikas koks sākas ar sākuma elementu, šajā gadījumā pēc gramatikas definīcijas tas saucas „WebProject", kā tas ir aprakstīts 6.2.1. nodaļā. Sākuma elements kā koka virsotne satur visas pārējās koka virsotnes, tāpēc transformācija uzģenerē kontroles klasi un kontroles klases izsaukšanu. Kontroles klases sastāvdaļas un citas klases tālāk tiek ģenerētas atkarībā no definētajiem nosacījumiem.

```
to-php:
WebProject(x, main*, other*) -> $[<?php
class main {
...
[pageslist]
}
[other]
[database]
$db = new database();
$x = new main($db);
?>] with databaseE := <filter(?Database(_,_))> main*;
database := <map(create-database)> databaseE;
pagesE := <filter(?WPages(_))> main*;
pageslist := <to-php> pagesE;
other := <to-php> other*
```

Koda piemērā parādīts kontroles klases transformācijas kods. Elementam WebProject ir 3 apakšelementi, kā tas jau ir definēts SDF valodā. Pirmais ir nosaukuma identifikators, otrais ir visi argumenti, kas definēti kontroles klasei un trešais ir visu pārējo klašu definīcijas. Stratego pierakstā šos te elementus pieraksta kā argumentus un tālāk ar viņiem ir iespējams operēt un nolasīt konstrukciju veidošanai. Ir iespējams nodefinēt dažādus nosaukumus definīciju kopai, tā lai vienam un tam pašam koka elementam būtu dažādas transformācijas atkarībā no izsaukšanas nosacījumiem.

Argumenti var būt dažādi un atkarībā no tā ir vai nu jāuzraksta katram iespējamam argumentam transformācija, vai arī izsaucot kādu transformāciju vispirms ir jāizfiltrē

izpildāmais koks. Pastāv iespēja transformācijas laikā izveidot pavisam jaunu koku, ko izpildīt. Iepriekš redzamajā transformācijas koda piemērā redzams, ka, piemēram, datubāze tiek izsaukta ar speciālu transformāciju „create-database", bet padodamais arguments ir iepriekš izveidots, izsaucot filtrēšanas funkciju uz koka elementa argumentu. Šajā gadījumā tiek izfiltrēti visi tie koka elementi, kuru nosaukums ir „Database" un kas sevī satur divus argumentus.

Transformācijas var notikt ne tikai koka elementiem, kuriem ir dots nosaukum. Ar Stratego pastāv iespēja ģenerēt saturu no koka elementiem bez nosaukuma, balstoties tikai uz koka elementa apakšelementu skaitu. Komanda „map" sameklē atbilstošo koda transformāciju dotajā transformācijas kategorijā. Iepriekš dotajā piemērā atbilstošā koda transformācija tiek meklēta ietvarā „create-database". Ja šajā ietvarā nebūs transformācija padotajiem koka elementiem, izraisīsies problēmas, kas tika aprakstītas 4. nodaļā.

Funkciju izsaukumu rezultātā tiek izveidots mainīgais, kas tiek ierakstīts ģenerētā teksta izvadā. Pareizi strukturējot un izvirzot prioritātes koka transformācijas ģenerēšanai, iespējams izveidot programmatūras tekstu.

6.4. Ģenerēšanas piemērs

Sākot veidot specializētu valodu, kas veiktu koda ģenerāciju programmētāja vietā, bija jāpadomā par robežām cik tālu ģenerācija notiks. Ideāls variants būtu iespēja no manas radītās valodas izveidot pilnīgi gatavu kodu, kas aizstātu līdzšinējo koda rakstīšanu. Taču konkrētās valodas izstrādei ir jānotiek pakāpeniski, atkarībā no autora nepieciešamībām koda izstrādē. Tādēļ valodas ierobežojums ir veikt koda ģenerāciju tīmekļa vietnes koda pamatnei, uz kura bāzes būtu iespējams turpināt koda izstrādi.

Pūlēm, kas tiek patērētās, sastādot šo pamatni bez domēnspecifiskās valodas iesaistīšanas, vajadzētu būt lielākām, nekā ja šāda valoda tiek iesaistīta. Turklāt, attīstot projektus ar šādas valodas palīdzību, tiek noskaidrotas valodas vājās vietas un iespēju ierobežojumi, kurus iespējams novērst vai paplašināt valodas iespējas.

Pabeidzot valodas pamatnes izstrādi, autoram pavērās iespēja izmēģināt izveidoto valodu praksē. Tika iegūts pasūtījuma pieteikums ar klienta prasībām par mazas informācijas sistēmas izveidi, kas būtu bāzēta kā tīmekļa vietne. Sistēmas mērķis būtu reģistrēt tehniskas

ierīces labošanas centrā un prasības būtu iespējas rediģēt iekārtu parametrus, klientu datus un veikt atskaišu izdrukas.

Izmantojot izveidoto domēnspecifisko valodu, izdevās uztaisīt pamatni datu apstrādei un attēlošanai. Lietas, pie kurām nācās piestrādāt, iesaistoties ar pašrocīgu programmas rakstīšanu, bija atskaišu izdrukas, kurām vajadzēja atsevišķu bibliotēku PDF ģenerēšanai, un meklēšana, kuras eksistence nav iekļauta valodā. Meklēšanas funkcija pieprasa, lai tīmekļa vietne spētu atmiņā saglabāt iepriekš ievadītos meklēšanas rezultātus, kā arī atvērt datus no rezultāta.

Pētījuma 2. pielikumā var aplūkot ģenerācijas piemēru vienam no tīmekļa vietnes lapas kontroles objektiem. Lai gan, iespējams, domēnspecifiskās valodas sintaksi var vēl vairāk saīsināt, jau šeit no 30 koda rindiņām domēnspecifiskās valodas koda tiek uzģenerēta klase 100 PHP koda rindiņu apjomā. Lai šī klase pilnvērtīgi strādātu, tiek uzģenerēts arī datu objekts no 14 koda rindiņām uz 145 PHP koda rindiņām.

6.5. Domēnspecifiskās valodas ieguvumi

Ieguvumu no konkrētās domēnspecifiskās valodas vislabāk var novērtēt, ja vērtē tādus parametrus, kā uzrakstītā koda rindiņu skaitu un patērēto laiku. Šos vērtēšanas parametrus gan arī nav viegli salīdzināt, jo laiku, kas tiek patērēts rakstot vienu un to pašu kodu ir grūti izmērīt. Arī kodu var uzrakstīt īsāk un garāk, tāpat eksistē arī laiks, kas aiziet, lai izdomātu kādu algoritmu jāraksta.

Rēķinot koda rindiņu apmēru, tad skatot praktisko piemēru, no apmēram 50 rindiņām domēnspecifiskās valodas koda varēja uzģenerēt rezultātu apmēram 500 koda rindiņu apmērā. Protams, ka šādu rezultātu nevar viennozīmīgi vērtēt kā konstantu ģenerācijas apmēru – ģenerācijas rezultātu apmēri ir atkarīgi no tā kādus datus nepieciešams ģenerēt.

Ģenerējot sarežģītības ziņā elementāras vietnes, ieguvums no domēnspecifiskās valodas ir mazs, jo elementārām vietnēm parasti ir nepieciešama vizuālā satura attēlošana. Tiklīdz tiek iesaistītas datu iesūtīšanas formas ar datu apstrādi, tā domēnspecifiskas valodas ģenerētais rezultāts sāk gūt priekšrocības.

Ja pieņem, ka uz viena datu lauka apstrādes uzrakstīšanu valodas rediģēšanas rīkā rakstot PHP kodu tīmekļa vietnei paiet līdz 10 minūtēm uzrakstot nolasīšanas, ierakstīšanas

un satura validācijas komandas datu objekta mainīgajam, kā arī notestējot mainīgā darbību, tad domēnspecifiskajā valodā atliek deklarēt mainīgā eksistenci un datu tipu, kas aizņem teksta uzrakstīšanas laiku līdz 30 sekundēm. Ja salīdzina maksimālo laika patēriņu un praksē pārbaudīto piemēru, tad datu objekts ar 8 mainīgajiem prasīja uzrakstīšanas laiku līdz 4 minūtēm domēnspecifiskajā valodā, kamēr rakstot funkcijas ar roku paietu līdz stundai un divdesmit minūtēm.

Vēl viena svarīga detaļa ir pats izstrādes posms. Sākot izstrādi, nedrīkst aizmirst, ka var būt un parasti ir vairāki varianti, kurus pasūtītājs izskatīs, lai pieņemtu tikai pēdējo no variantiem. Ir būtiska atšķirība, ja klients maina daļu no nosacījumiem vai pat tikai kādu sīku detaļu, kas izjauc esošās programmas loģiku. Veikt izmaiņas kodā nozīmē izsekot un izmainīt algoritmus, kā arī pārbaudīt vai galarezultātā tīmekļa vietne vēl strādā. Izmantojot koda ģenerāciju, iespējams mainīgos vai vietnes datu apstrādes loģiku izmainīt, netērējot laiku koda pārskatīšanai. Protams, konkrētajā gadījumā balstoties uz neatkarīgu veidņu saturu un datubāzi, dažas no programmēšanas lietām ir jāizdara papildus ģenerācijai.

Šobrīd izveidotajā valodā nav atrisināts jautājums par koda pārmaiņām un ar roku pieliktajām izmaiņām. Pārģenerējot kodu, izmaiņas izzudīs. Piemēram, veidņu sistēma ir balstīta uz datnēm, kas atrodas ārpus ģenerējamās informācijas un nemainās pārģenerācijas laikā. Var rasties koda nesaderība, taču kods nepazudīs. Sadarbība ar datubāzi, kur var nākties pielabot SQL datnes, lai atlasītu informāciju no vairākām tabulām, var tikt piemēram, apieta izmantojot datubāzes skatus, kas jau atlasa visas tabulu sakarības. Atliek vienīgi algoritma kods, kas tiek pievienots tieši ģenerētajam saturam. Šajā gadījumā brīdī, kad tiek ieviests papildus kods, ir jāizvērtē, vai algoritma sintaksi nav iespējams pievienot domēnspecifiskās valodas komandām. Paļaujoties uz iespējamību, ka jauno pielikumu būs iespējams pievienot, ir jāpapildina domēnspecifiskā valoda un problēma ar koda zaudēšanu pārģenerācijas gadījumā kļūst neaktuāla.

Viena no PHP valodas problēmām ir atkļūdošana – no izstrādātāja viedokļa tā ir pietiekami sarežģīta un nepārskatāma. Kad PHP valodas kods tiek pārbaudīts testēšanas vidē, iespējams uzstādīt dažādus kļūdu ziņošanas līmeņus, no kuriem zemākais ir „piezīmes", kas apraksta vietas kodā, kur izpildes laikā, iespējams, notiek kāda kļūda, bet tā neietekmē izpildes kvalitāti. Parasti pašrocīgi programmējot šāda tipa kļūdas rodas ļoti daudz, piemēram, tiek algoritmā tiek izmantota definēta globālā konstante, kas nav deklarēta – PHP valoda nedeklarēto konstanti iztulko kā „Null" vērtību un ziņo par to „piezīmju" kļūdā. Ģenerētajā valodā šādi kļūdu paziņojumi tika izskausti līdz stāvoklim, kad netika ziņots vispār par kādām

piezīmēm koda izpildē. Protams, valodas izstrādes sākumā dažas kļūdas bija, taču tās bija viegli novērst un, galvenais, kods bija jāpielabo tikai vienā vietā, bet kļūdas labojums pielietojās visās izmantošanas vietās ģenerējot.

6. SECINĀJUMI

Par darba mērķi tika izvirzīts izveidot domēnspecifisko valodu un salīdzināt projekta izveidošanas izmaksas laika un kvalitātes ziņā ar tīmekļa vietņu izstrādi, kas tiek veikta ar plaša pielietojuma valodu. Pētījuma laikā tika izveidota strādājoša domēnspecifiskā valoda, kas gan pilnībā neaizstāj programmēšanas darbu, bet atvieglo programmēšanas uzsākšanu ar ģenerētu kodu, aizstājot vajadzību programmēt pamatelementus tīmekļa vietnes projektiem. Tika noskaidrots, ka kvalitātei ir tieša saistība ar vēlmi ieguldīt laiku un resursus domēnspecifiskās valodas izstrādē - jo vairāk valoda tiks pielietota un attīstīta, jo mazāk laika un darba nāksies pielietot dažādu pasūtījuma nianšu programmēšanai.

Darba hipotēzē tika izvirzīts apgalvojums, ka ātrākais veids, salīdzinot ar plaša pielietojuma valodām, bibliotēkām un ievariem, kā radīt tīmekļorientētus projektus, ir izmantot domēnspecifisko valodu. Izvērtējot darba rezultātus, šādu apgalvojumu nevar pamatot, jo darba rezultāts nav devis neapgāžamu, pat vienā līmenī ar bibliotēkām un ietvariem salīdzināmu iznākumu. Rezultātu var uzskatīt par pārāku par vienkāršu plaša pielietojuma valodas lietošanu, jo no aprakstoša koda tiek ģenerēts kods lielākā apjomā un dziļumā, turklāt laika ziņā tiek iegūts ietaupījums. Taču salīdzinot ar bibliotēkām un ietvariem laika ietaupījuma nav. Līdz ar to hipotēze nav pierādīta, lai gan ilgākā laika posmā attīstot un pilnveidojot domēnspecifisko valodu, ir iespēja, ka kādā brīdī ģenerācijas laika un kvalitātes ieguvums pārsniedz gatavu rakstīta bibliotēku un ietvaru kodu izmantošana. Jāņem vērā, ka domēnspecifiskās valodas ģenerētais produkts ir īpaši pielāgots konkrētajam projektam, lai gan ietvari un bibliotēkas paļaujas, ka katrs projekts tiks pielāgots viņu prasībām.

Ja salīdzina koda ģenerēšanu pret koda rakstīšanu, tad var diskutēt par laika ieguvumu pret radītā koda daudzumu. Radot domēnspecifisko valodu vienreizējai lietošanai vai tīmekļa vietnēm ar statisku saturu jeb elementāru sarežģītību nav iespējams iegūt kādu labumu, šādā situācijā labāk ir rakstīt kodu neizmantojot domēnspecifiskas valodas. Ja salīdzina sarežģītāku tīmekļa vietņu ar vairākiem datu laukiem ģenerācijas procesu pret koda rakstīšanu, vēl aizvien var argumentēt atrodot noteiktus apstākļus, ka ieguvums nav pietiekami būtisks un ieguldījumu vērts. Taču, ja izstrādes procesā būs nepieciešams veikt labojumus, tad izmaiņu izdarīšana domēnspecifiskajā valodas kodā un rezultāta ģenerēšana ir daudzkārt ātrāka un kvalitatīvāka par koda labošanu ar redaktoru.

Bakalaura darba rezultātā autors ir ieguvis paplašinātu pieredzi ar domēnspecifiskajām valodām, iemaņas domēnspecifiskas valodas veidošanā un valodu, kura gan nespēj veikt visas

vajadzīgās prasības sarežģīta tīmekļa projekta veidošanai. Taču, turpinot ģenerēt projektus un uzlabojot valodas iespējas, ir domēnspecifisko valodu ir iespējams attīstīt līdz līmenim, kad arī sarežģīta līmeņa projekti pilnībā var tik ģenerēti ar minimālu vai nekādu koda labošanu.

Izmantotā literatūra un avoti

1. Comparison of Textual and Visual Notations of DOMMLite Domain-Specific Language / Aut.kol. I.Dejanović, M.Tumbas, G.Milosavljević, B.Perišić. In CEUR Workshop Proceedings Volume 639, CEUR-WS.org, 2010.
2. Composing Domain Specific Design Environments/ Aut.kol. A.Ledeczi, A.Bakay, M. Maroti. In Computer Volume 34, Issue 11, IEEE Computer Society Press, 2001. ISSN: 0018-9162. - 1.lpp.
3. Declarative Access Control for WebDSL: Combining Language Integration and Seperation of Concerns/ Aut.kol. D.Groenewegen, E.Visser. In International Conference on Web Engineering, IEEE Computer Society, Washington D.C., 2008.
4. Domain-Specific Languages: An Annotated Bibliography/ Aut. kol. A. van Deursen, P. Klint, J. Visser, New York, USA, ACM, 2000. - 1.-3.lpp
5. Domain Specific modelling: enabling full code generation / Aut.kol. S. Kelly, J.-P. Tolvanen, Wiley-IEEE Computer Society Pr, 2008. ISBN-10: 0470036664. - 36.-37.lpp.
6. Language Workbenches: The Killer-App for Domain Specific Languages?, M. Fowler [tiešsaiste]. - 2007. [atsauce 12.12.2007.] Pieejams internetā: http://martinfowler.com/articles/languageWorkbench.html
7. Lessons Learned from Real DSL Experiments, D. Wile. In Science of Computer Programming, Volume 51, Issue 3, Elsevier North-Holland, Inc., 2004. ISSN: 0167-6423. - 2.-4.lpp
8. Requirements for Domain-Specific languages/ Aut.kol. D. S. Kolovos, R. F. Paige, T. Kelly, F. A. C. Pollack. In Proceeding of the First ECOOP Workshop on Domain-Specific Program Development (ECOOP'06), Nantes, France, 2006. - 2.lpp.
9. Stratego/XT A Language and Toolset for Program Transformation/ Aut.kol. M. Braveboer, K. T. Kalleborg, R. Vermaas, E. Visser. In Science of Computer Progrmming, Volume 72, Issues 1-2, 2008. - 2.lpp.
10. WebDSL [Tiešsaiste], Z. Hemel. Atsauce[26.05.2010.], pieejams http://www.slideshare.net/zefhemel/webdsl-presentation.
11. WebDSL: A Case Study in Domain-Specific Language Engineering, E. Visser. In Lecture Notes in Computer Science, Volume 5235, Springer Berlin / Heidelberg, 2008. ISSN: 0302-9743. - 6.,9.lpp.

12. When and How to Develop Domain-Specific Languages/ Aut.kol. M. Mernik, J. Heering, A.M.Sloane. In ACM Computing Surveys, Volume 37, Issue 4, ACM, 2005. ISSN: 0360-0300. - 2.,3. lpp.
13. Metacase [Atsauce: 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://www.metacase.com/mep/
14. MS Modeling Tool [Atsauce 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://www.microsoft.com/downloads/details.aspx?FamilyId=57A14CC6-C084-48DD-B401-1845013BF834&displaylang=en
15. Eclipse Graphical Modelling Platform [Atsauce 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://www.eclipse.org/modeling/gmp/
16. Spoofax [Atsauce 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://strategoxt.org/Spoofax
17. Stratego XT [Atsauce 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://strategoxt.org/
18. Eclipse Modelling Platform [Atsauce 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://www.eclipse.org/modeling/
19. MontiCore [Atsauce 10.06.2010.] [Tiešsaiste] Pieejams internetā: http://www.monticore.org
20. A Quick Introduction To SPF, J.Visser, J.Scheerder. [Atsauce 19.05.2012.] [Tiešsaiste] Pieejams internetā: ftp://ftp.stratego-language.org/pub/stratego/docs/sdfintro.pdf
21. SDF sintakse, [Atsauce 16.04.2012.] [Tiešsaiste] http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html
22. http://hydra.nixos.org/build/2230584/download/1/manual/chunk-chapter/tutorial-software-transformation-systems.html [Atsauce 05.04.2012.]
23. http://www.languageworkbenches.net/index.php?title=LWC2011_Comparison_Matrix [Atsauce 05.04.2012.]
24. Eclipse tool [Atsauce 10.05.2012.][Tiešsaiste] Pieejams internetā: http://www.eclipse.org
25. SGLR rīks, [Atsauce 12.05.2012.] [Tiešsaiste] http://www.program-transformation.org/Sdf/SGLR

Pielikumi

1. SDF gramatikas pilns piemērs

```
1   module PhpValoda
2   imports Common
3   exports
4     context-free start-symbols
5       Start
6     context-free syntax
7   "project"  ID  "{"  {WebProperties  ","}*  "}"  Blocks*  ->  Start
    {cons("WebProject")}
8   "title" ":" STRING -> WebProperties{cons("CommonTitle")}
9   "pages" "{" {PageElements ","}* "}" -> WebProperties{cons("WPages")}
10  "template"  STRING  "{"  {TemplateProperties  ","}*  "}"  ->
    WebProperties{cons("BasicTemplate")}
11  ID -> PageElements{cons("WPage")}
12  ID ":" "default" -> PageElements{cons("PageDefault")}
13  "database" ID "{" {DBElems ","}* "}" -> WebProperties{cons("Database")}
14  "user" ":" STRING -> DBElems{cons("DBUser")}
15  "pass" ":" STRING -> DBElems{cons("DBPass")}
16  "db" ":" STRING -> DBElems{cons("DBName")}
17  "host" ":" STRING -> DBElems{cons("DBHost")}
18  "port" ":" STRING -> DBElems{cons("DBPort")}
19  "page" ID "{" {PageProperties ","}* "}"  -> Blocks{cons("Page")}
20  "dataobject" ID "{" {DataProperties ","}* "}" -> Blocks{cons("DataObject")}
21  ID -> PageProperties{cons("Prop")}
22  "template"  STRING  "{"  {TemplateProperties  ","}*  "}"  ->
    PageProperties{cons("Template")}
23  "fields" "{" {FieldObjects ","}* "}" -> DataProperties{cons("DataFields")}
24  "table" ":" STRING -> DataProperties{cons("DataTable")}
25  STRING "->" ID "::" ID -> FieldObjects{cons("FormLinks")}
26  ID ":" ID -> FieldObjects{cons("DataField")}
27  "form" ID "(" {Formproperties ","}* ")" -> PageProperties{cons("PageForm")}
28  "template"  STRING  "{"  {TemplateProperties  ","}*  "}"  ->
    Formproperties{cons("Template")}
29  ID ":" STRING -> Formproperties{cons("FormProp")}
30  ID ":" ID -> Formproperties{cons("FormStatus")}
31  "fields" "{" {FieldObjects ","}* "}" -> Formproperties{cons("FFields")}
32  "success" ":" "load" ID -> Formproperties{cons("SuccessLoad")}
33  "fail" ":" "string" STRING -> Formproperties{cons("FailMessage")}
```

```
34    ID "->" ID -> TemplateProperties{cons("TmplID")}
35    ID "->" "self" ID ->TemplateProperties{cons("TmplIDS")}
36    ID "->" STRING -> TemplateProperties{cons("TmplStr")}
37    ID "->" -> TemplateProperties{cons("TmplEmpty")}
38    ID     "->"     "list"     "{"     {TemplateListProp     ","}*     "}"     ->
      TemplateProperties{cons("TmplList")}
39    "(" {TemplateProperties ","}* ")" -> TemplateListProp{}
```

2. Koda transformācijas piemērs

Domēnspecifiskās valodas koda piemērs lapas kontroles objektam.

```
page inventory {
	form inventorycontrol(
		fields {
			"Cat"->Stock::category,
			"Man"->Stock::manufacturer,
			"Itm"->Stock::item,
			"Sup"->Stock::supplier,
			"Pcp"->Stock::costPrice,
			"Psp"->Stock::salePrice,
			"Pis"->Stock::quantityStock,
			"Sld"->Stock::quantitySold
		},
		template "inventoryForm" {
			fields->list{
				(title->"Category",name->"fieldCat",value->),
				(title->"Manufacturer",name->"fieldMan",value->),
				(title->"Item",name->"fieldItm",value->),
				(title->"Supplier",name->"fieldSup",value->),
				(title->"Product cost price",name-
>"fieldPcp",value->),
				(title->"Product sale price",name-
>"fieldPsp",value->),
				(title->"Product in stock",name->"fieldPis",value-
>),
				(title->"Sold products",name->"fieldSld",value->)
			},
			button -> "Add",
			fmeth->"post",
			fname->"formAdd",
			fact->
		},
		button : "Add",
		method : "post",
		success : load viewInventory,
		fail : string "Cannot add item to inventory"
	)
}
```

Ģenerēšanas rezultāts lapas kontroles objektam

```
class classPageinventory {
	private $db;
	private $body;
	public function __construct(&$db){
		$this->db = $db;
		$this->body = "";
	}

	private function getForminventorycontrol(){

		$tmpl = new template("form.html");
		$vars = array("fields"=>array(array("title"=>"Category",
					"name"=>"fieldCat",
					"value"=>"",
					),
				array("title"=>"Manufacturer",
					"name"=>"fieldMan",
					"value"=>"",
					),
				array("title"=>"Item",
```

```
                                    "name"=>"fieldItm",
                                    "value"=>"",
                                    ),
                              array("title"=>"Supplier",
                                    "name"=>"fieldSup",
                                    "value"=>"",
                                    ),
                              array("title"=>"Product cost price",
                                    "name"=>"fieldPcp",
                                    "value"=>"",
                                    ),
                              array("title"=>"Product sale price",
                                    "name"=>"fieldPsp",
                                    "value"=>"",
                                    ),
                              array("title"=>"Product in stock",
                                    "name"=>"fieldPis",
                                    "value"=>"",
                                    ),
                              array("title"=>"Sold products",
                                    "name"=>"fieldSld",
                                    "value"=>"",
                                    ),
                              ),
                        );
                        $tmpl->addVars(
                        $vars
                        );
                        //return $tmpl->output();
                        $this->body .= $tmpl->output();

                  }
                  private function workForminventorycontrol(){
                        $well=(bool)true;

                        $this->dobject["Stock"] = new Stock($this->db);

                        if (isset($_REQUEST["fieldCat"])){
                        $this->dobject["Stock"]->valcategory($_REQUEST["fieldCat"]);
                        }
                        if (isset($_REQUEST["fieldMan"])){
                        $this->dobject["Stock"]->valmanufacturer($_REQUEST["fieldMan"]);
                        }
                        if (isset($_REQUEST["fieldItm"])){
                        $this->dobject["Stock"]->valitem($_REQUEST["fieldItm"]);
                        }
                        if (isset($_REQUEST["fieldSup"])){
                        $this->dobject["Stock"]->valsupplier($_REQUEST["fieldSup"]);
                        }
                        if (isset($_REQUEST["fieldPcp"])){
                        $this->dobject["Stock"]->valcostPrice($_REQUEST["fieldPcp"]);
                        }
                        if (isset($_REQUEST["fieldPsp"])){
                        $this->dobject["Stock"]->valsalePrice($_REQUEST["fieldPsp"]);
                        }
                        if (isset($_REQUEST["fieldPis"])){
                        $this->dobject["Stock"]-
>valquantityStock($_REQUEST["fieldPis"]);
                        }
                        if (isset($_REQUEST["fieldSld"])){
                        $this->dobject["Stock"]->valquantitySold($_REQUEST["fieldSld"]);
                        }
```

```
            if(!$this->dobject["Stock"]->save()) $well=(bool)false;

            if ($well){

                  $obj = new classPageviewInventory($this->db);
                  $this->body .= $obj->body();
                  //return $this->body;
                  return (bool)true;

            }else{

                  $this->body.='<p class="Error">Cannot add item to
inventory</p>';

            }
            return (bool)false;
      }


      public function head(){}
      public function body(){

            if (isset($_REQUEST["hfieldForm"]) &&
$_REQUEST["hfieldForm"]=='inventorycontrol'){
                  if ($this->workForminventorycontrol()){
                        return $this->body;
                  }
            }
            $this->body.=$this->getForminventorycontrol();

            return $this->body;
      }
}
```

Saskarnes veidņa „form.html” kods

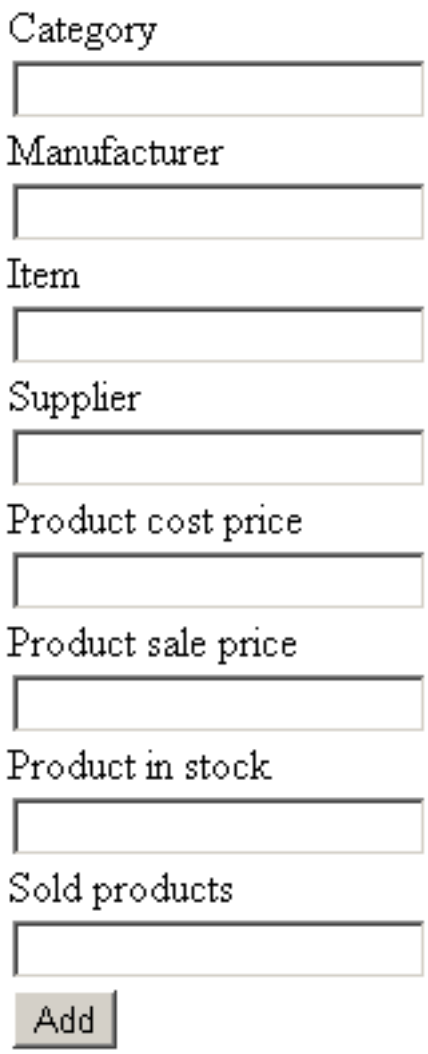
```
<form name="%fname%" method="%fmeth%" action="%fact%">
      (fields%
      <div><span class="title">%title%</span><input type="text" name="%name%"
value="%value%" /></div>
      %fields)
      <input type="submit" name="hfieldsubmit" value="%button%" />
</form>
```

Ģenerētā rezultāta attēlotais HTML kods, ko redz tīmekļa vietnes apmeklētāji pārlūkprogrammā

```
<html><head><meta http-equiv="Content-Type" content="text/html; charset=windows-1257">
<style>
.title {width:200px;display:block;}
</style>
</head>
<body>
<form name="formAdd" method="post" action="">

      <div><span class="title">Category</span><input type="text" name="fieldCat"
value=""></div>

      <div><span class="title">Manufacturer</span><input type="text" name="fieldMan"
value=""></div>

      <div><span class="title">Item</span><input type="text" name="fieldItm" value=""></div>

      <div><span class="title">Supplier</span><input type="text" name="fieldSup" value=""></div>
```

```
        <div><span class="title">Product cost price</span><input type="text" name="fieldPcp"
value=""></div>

        <div><span class="title">Product sale price</span><input type="text" name="fieldPsp"
value=""></div>

        <div><span class="title">Product in stock</span><input type="text" name="fieldPis"
value=""></div>

        <div><span class="title">Sold products</span><input type="text" name="fieldSld"
value=""></div>

        <input type="submit" name="hfieldsubmit" value="Add">
</form>

</body></html>
```

Redzamais attēls pārlūkprogrammā:



Dokumentāra lapa

Bakalaura darbs „Domēnspecifiskā valoda tīmekļa programmēšanai” izstrādāts LU Datorzinātņu fakultātē.

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

Autors: Sandis Krutovs

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: Dr.sc.comp. Agris Šostaks

Recenzents: Dr.dat.doc. Lelde Lāce

Darbs iesniegts Datorikas fakultātē

Metodiķe: Ārija Sproģe