TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Siim Erik Potsepp 164256 IABB

JETBRAINS META PROGRAMMING SYSTEM KASUTAMINE VALDKONNAPÕHISTE KEELTE DISAINIMISEL

bakalaureusetöö

Juhendaja: Gunnar Piho
Doktorikraad

Tallinn 2019

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Siim Erik Potsepp

07.05.2019

Annotatsioon

Valdkonnapõhised keeled on väikesemahulised programmeerimiskeeled, mis on suunatud valdkonna ekspertidele. Vaatamata pikale ajaloole pole valdkonnapõhiseid keeli valdkonna ekspertide poolt kasutusele võetud. Peamiseks põhjuseks on piisavalt mugava töövahendi puudumine. See kirjatöö käsitleb valdkonnapõhiste keelte loomiseks mugava töövahendi puudumise probleemi. Töö eesmärgiks on välja selgitada, kas ettevõtte JetBrains poolt loodud arenduskeskkond MPS (*Meta Programming System*) on sobiv töövahend valdkonnapõhiste keelte loomiseks.

Töö käigus luuakse kaks valdkonnapõhist keelt ning üks näiteprojekt objektorienteeritud meetodil. Põhilisteks arendusvahenditeks on MPS ja Intellij Idea ning töös kasutatakse peamiselt Java programmeerimiskeelt. Kahe valdkonnapõhise keele ning näiteprojekti loomise põhjal ei ole autori hinnangul MPS sobiv vahend valdkonnapõhiste keelte loomiseks. Peamisteks põhjusteks on liiga suur õppimiskõver ja vajalikud algteadmised MPS-i kasutamiseks. Autori järeldust kinnitavad ka teised uurimused MPS-i kohta.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 22 leheküljel, 3 peatükki, 20 joonist, 0 tabelit.

Abstract

Designing domain specific languages with JetBrains Meta Programming System

Domain specific languages (DSL) are small programming languages, with simple syntax. DSL's are meant to handle the needs of a specific domain. Domain can be anything from biology, insurance, banking or aviation. Because of the simple syntax DSL's can be used by domain experts, who have deep knowledge about the domain and are thus most competent to describe software requirements. However, domain specific languages are not widely used by domain experts and the main reason is the lack of appropriate tool for using DSL's.

This paper covers the problem of not having appropriate tools for designing and using DSL's. The goal of this paper is to assess whether JetBrains MPS (Meta Programming System) is appropriate tool for designing and using DSL's. MPS is a language workbench, that is meant to design, test and use DSL's. MPS uses projectional editor, which means logic is represented in text–like manner, but any change in code is changing AST (abstract syntax tree) in the background. AST is a tree–like, model structure of the code, that represents only the most important parts of the code.

The main tools used in this paper are MPS and Intellij Idea and all the code examples are in Java programming language. MPS is used to design two DSL's, while Intellij Idea is used to create a sample project in a traditional object–oriented way. Based on the experiences with MPS author finds that MPS is not appropriate tool for designing and using DSL's. Main points that author found are steep learning curve, amount of knowledge needed before using MPS and MPS not solving the communication problem between software developers and domain experts. Authors finds are also supported by other researches about MPS.

The thesis is in estonian and contains 22 pages of text, 3 chapters, 20 figures, 0 tables.

Lühendite ja mõistete sõnastik

MPS	Meta Programming System
AST	*Abstract Syntax Tree*, abstraktne süntaksipuu
DSL	*Domain Specific Language*, valdkonnapõhine keel

Sisukord

1 Sissejuhatus 8
2 Metoodika 9
2.1 JetBrains MPS 10
2.1.1 Arendusprotsess MPS-is 11
2.2 Kasutatud arenduskeskkonnad ja mustrid 12
2.3 Tööprotsess 13
3 Valdkonnapõhised keeled MPS-is ja näiteprojekt 14
3.1 Esimene valdkonnapõhine keel 14
3.1.1 MPS-i spetsiifika „*ClassGeneratoris*“ 17
3.2 Teine valdkonnapõhine keel 17
3.3 Näiteprojekt objektorienteeritud meetodil 21
4 Tulemuste analüüs 24
4.1 Martin Fowler keele tööpinkidest ja valdkonnpõhistest keeltest 24
4.2 Teised MPS-i uurimused 25
4.3 Autori järeldused 26
5 Kokkuvõte 29

Jooniste loetelu

Joonis 1. Abstraktse süntaksipuu näide matemaatilise tehte näol. 10
Joonis 2. Sõiduki kategooria valdkonna esitus abstraktse süntaksipuuna. 11
Joonis 3. „*ClassGenerator*“ valdkonnapõhise keele tühi mall. 14
Joonis 4. Näide „*ClassGenerator*“ valdkonnapõhise keele kasutaja kirjutatud koodist. 14
Joonis 5. „*ClassGenerator*“ valdkonnapõhise keele poolt genereeritud koodi näide, mis esitab „on“ tüüpi seost. 15
Joonis 6. „*ClassGenerator*“ valdkonnapõhise keele poolt genereeritud koodi näide, mis esitab „omab“ ja „on“ tüüpi seoseid. 16
Joonis 7. Näide generaatori aspekti realisatsioonist, mis genereerib klassid ilma „on“ tüüpi seoseta. 16
Joonis 8. Näide generaatori aspekti realisatsioonist, mis genereerib klassid „on“ tüüpi seosega. 16
Joonis 9. Baas mõõtme defineerimine keele lõppkasutaja poolt. 18
Joonis 10. Tuletatud mõõtme defineerimine keele lõppkasutaja poolt. 18
Joonis 11. Ühikute defineerimine keele lõppkasutaja poolt. 19
Joonis 12. Koguste defineerimine keele lõppkasutaja poolt. 19
Joonis 13. Näide keele lõppkasutaja poolt kirjutatud „*Quantity*“ valdkonnapõhisest keelest. 19
Joonis 14. Täielik näide „*Quantity*“ valdkonnapõhise keele poolt genereeritud Java koodist. 20
Joonis 15. „*Quantity*“ valdkonnapõhise keele generaatori realisatsioon. 20
Joonis 16. Klassi *DerivedMeasure* realisatsioon näiteprojektis. 21
Joonis 17. Klassi *MeasureTerms* realisatsioon näiteprojektis. 22
Joonis 18. Klassi *MeasureTerm* realisatsioon näiteprojektis. 22
Joonis 19. Koguse mustri kasutamine näiteprojektis. 23
Joonis 20. Kiiruse ja pindala valemi defineerimine koguse mustri kasutamise jaoks. 23

1 Sissejuhatus

Tarkavaraarenduse üks suurimaid probleeme on selle valdkonna tekkimise algusest olnud kommunikatsioonibarjäär tarkvaraarendajate ja valdkonna ekspertide vahel. Valdkonna ekspertide all mõeldakse kindlustuse, meditsiini, bioloogia, ehituse ja teiste valdkondade asjatundjaid, kes defineerivad nõuded loodavale tarkvarale. Probleemi on välja toonud tuntud arvutiteadlased nagu Martin Fowler ja Kent Beck [1] ning antud probleemiga on isiklikult kokku puutunud ka töö autor. Üks pakutavaid lahendusi sellele probleemile on valdkonnapõhiste keelte (*domain specific language*) kasutamine.

Valdkonnapõhised keeled on oma olemuselt väikesemahulised programmeerimiskeeled, mis on mõeldud kitsa valdkonna probleemide lahendamiseks. Valdkonnapõhistes keeltes kasutatavad süntaksi elemendid valitakse keelt disainides selliselt, et need oleksid mõistetavad nii tarkvaraarendajale kui valdkonna eksperdile. Esimesi programmeerimiskeeli nagu Cobol, Fortran ja Lisp võib samuti lugeda valdkonnapõhisteks keelteks [2]. Valdkonnapõhiste keeltega on põhjalikult tegelenud Martin Fowler, Dines Bjørner ja Eric Evans.

Hoolimata pikast ajaloost ei ole valdkonnapõhised keeled siiani valdkonna ekspertide poolt kasutusele võetud ning tarkvaraarendajad kasutavad oma igapäeva töös peamiselt üldotstarbelisi keeli nagu Java, Pyton või C#. Põhjus on selles, et puudub töövahend, mille tööprotsess oleks piisavalt lihtne valdkonna eksperdi jaoks ning piisavalt tulemuslik tarkvaraarendajatele. See kirjatöö kätkeb endas sobiva töövahendi puudumise probleemi.

Pidevalt esitletakse uusi töövahendeid, mis lubavad lahendada valdkonnapõhiste keeltega kaasas käivad probleemid. Selle kirjatöö eesmärgiks on uurida, kas ettevõtte JetBrains poolt välja töötaud arendustarkvara MPS (*Meta Programming System*) on sobiv vahend valdkonnapõhiste keelte kasutamiseks ning loomiseks. Töö käigus antakse ülevaade MPS-i võimalustest, hinnatakse valdkonnapõhiste keelte loomise tööprotsessi, võrreldakse MPS-iga saadud koodi klassikaliste tehnoloogiate abil kirjutatud koodiga ning analüüsitakse tulemusi. Autor võrdleb saadud tulemusi ka teiste MPS-i puudutavate uurimustööde tulemustega.

2 Metoodika

MPS (*Meta Programming System*) on valdkonnapõhiste keelte loomiseks ning kasutamiseks mõeldud keele tööpink (*language workbench*). Mõistet keele tööpink kasutas esimest korda arvutiteadlane Martin Fowler. Lihtsustatult koondab mõiste keele tööpink enda alla kõiki arendusvahendeid, millega saab luua ning kasutada valdkonnapõhiseid keeli [3].

Keele tööpingid võimaldavad rakendada keeltele suunatud programmeerimise (*language oriented programming*) arendusmeetodit. Keeltele suunatud programmeerimine keskendub tarkvara loomisele valdkonnapõhiste keelte abil ning mõiste võttis kasutusele Martin Fowler [3]. Selline arendusmeetod, kus kasutatakse erinevaid programmeerimiskeeli korraga, osutub väga keeruliseks tavaliste arendusvahenditega nagu Visual Studio ja JetBrains Intellij Idea. Põhjus on parserites, mis seovad keele süntaksi, ehk visuaalse esituse, keele poolt esitatud semantika, ehk koodi tähendusega.

Sarnase süntaksiga keelte kasutamine ühes dokumendis muudab parseri töö keeruliseks, sest parser ei suuda sama süntaksiga keele elemente eristada. Nimetatud probleem viib koodi mitte–tekstilise esituse ideeni, mis eemaldaks vajaduse kasutada parsereid [4]. Mitte–tekstilise koodi esituse puhul toimub tarkvara loogika kirjeldamine ja salvestamine mudeli kujul. See on üks peamisi erinevusi traditsiooniliste arendusvahendite ja keele tööpinkide vahel. Traditsioonilised arendusvahendid esitavad ja salvestavad loogikat tavatekstina. Keele tööpingid keskenduvad aga valdkonna põhiobjektide, näiteks arve, klient või leping, kirjeldamisele ja salvestamisele mudelite kujul.

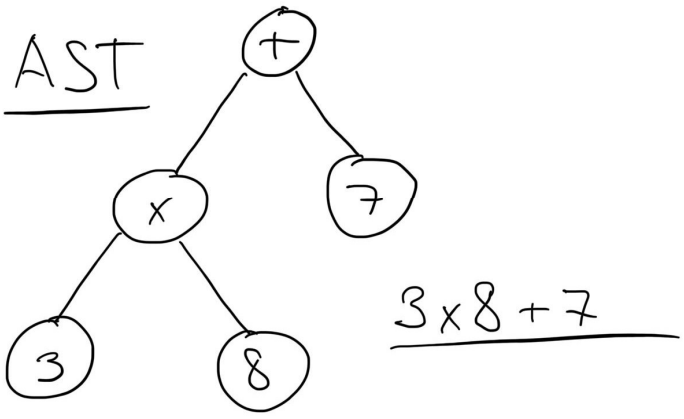
Lisaks valdkonna põhiobjektide kirjeldamisele võimaldavad keele tööpingid luua abstraktse kihi nende põhiobjektide kasutamiseks. Abstraktse kihi all mõeldakse valdkonnapõhise keele loomist. Valdkonnapõhine keel on lihtsasti arusaav ning võimaldab kirjeldada tarkvara loogikat ilma üldkasutatavat keelt kasutamata. Valdkonnapõhises keeles võivad põhiobjektid olla esitatud graafiliselt või tekstilist. Antud kirjatöös käsitletav keele tööpink MPS esitab valdkonna põhiobjektid tekstiliselt.

2.1 JetBrains MPS

MPS on üks paljudest keele tööpinkidest, mis lubab lahendada valdkonnapõhiste keeltega ning üldiselt tarkvaraarenduses esinevad probleemid. Peamiselt rõhutakse valdkonna ekspertide kaasamisele arendusprotsessi ning suhtlusbarjääri vähendamisele tarkvaraarendajate ja valdkonna ekspertide vahel. MPS-i töötas välja Tšehhi ettevõte JetBrains. JetBrainsi tuntumad tooted on Intellij Idea, ReSharper ning RubyMine. MPS sai alguse majasisesest projektist ning tänaseks on see tasuta kättesaadav arendustarkvara, mis on litsentseeritud Apache 2.0 litsentsiga [5].

JetBrains kirjeldab MPS-i kui keele tööpinki, mis on mõeldud valdkonnapõhiste keelte loomiseks. JetBrainsi hinnangul on valdkonnapõhiste keelte kasutamise abil võimalik automatiseerida ettevõtte tööprotsesse, vähendada kommunikatsiooni barjääri ning muuta tarkvara äriloogikat ilma tarkvaraarendaja poolse abita [6].

MPS-is toimub mudelite muutmine tekstiliselt, jäljendades tavameetodil koodi kirjutamist. Seega toimub igasuguse koodi muutmise korral mitte teksti vaid mudeli muutus. Mudel koosneb valdkonna põhiobjektidest ning MPS-is salvestatakse põhiobjektid ning nende vahelised seosed AST-na (abstraktne süntaksipuu) [4]. AST on lihtsustatud versioon parsimis puust [7], mis esitab kogu koodi loogika ning semantika puukujuliselt. Joonisel 1 on toodud näide AST-st.



Joonis 1. Abstraktse süntaksipuu näide matemaatilise tehte näol.

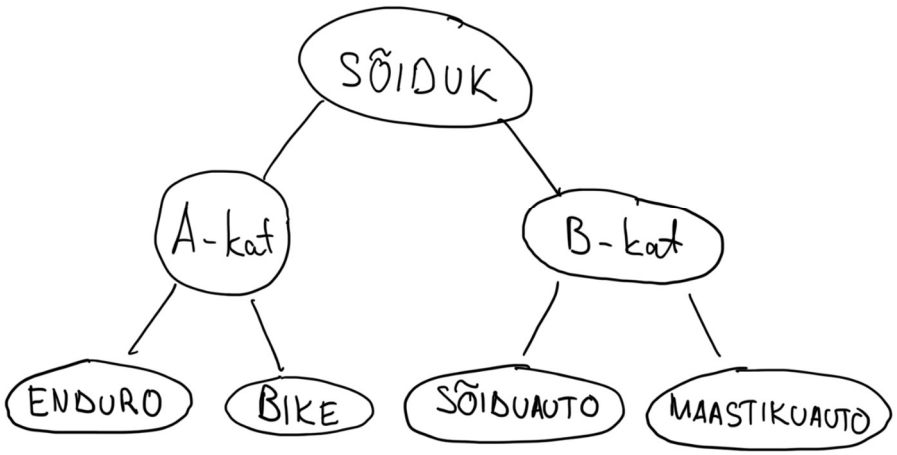
Joonisel on esitatud lihtne matemaatiline tehe. Puu abil on võimalik teha kindaks tehete järjekord, märgid ning tehtes kasutatavad arvud. Erinevalt parsimis puust on AST-s

esitatud vaid need elemendid, mis annavad koodile tähenduse. MPS-i eesmärk ongi AST esitamine võimalikult kasutajasõbralikult.

2.1.1 Arendusprotsess MPS-is

MPS-i tööprotsess jaguneb kaheks. Valdkonnpõhiste keelte disainimine ning loodud keelte kasutamine. Valdkonnapõhise keele disainimisel on vaja minimaalselt kirjeldada kolm põhilist keele osa: valdkonna põhiobjektid, keele süntaks ning valdkonnapõhisest keelest genereeritav üldkasutatav keel. MPS-is nimetatakse keele disaini osasid aspektideks.

Kõige olulisem aspekt on valdkonna põhiobjektide kirjeldamine, sest põhiobjekte kasutades luuakse kõik ülejäänud keele aspektid. Sarnaselt objektorienteeritud programmeerimise [8] klassidele defineeritakse MPS-is põhiobjekti nimi, omadused (*property*), laiendid ning muud põhiobjekti iseloomustavad omadused. Valdkonna põhiobjektid asetsevad ka keelt esindava AST sõlmpunktides [9]. Joonisel 2 on esitatud näide valdkonnast, mis kirjeldab sõidukite kategooriaid.



Joonis 2. Sõiduki kategooria valdkonna esitus abstraktse süntaksipuuna.

Selleks, et valdkonnapõhist keelt kasutada on vaja iga valdkonna põhiobjekti jaoks defineerida süntaksi aspekt, ehk kirjapilt. Süntaks määrab valdkonna põhiobjekti visuaalse esituse [10]. Oluline on muuta valdkonnapõhine keel lihtsatsi kasutatavaks ning arusaadavaks. Peamine erinevus üldkasutatavate keelte ja valdkonnpõhiste keelte vahel ongi lihtsustaud kirjapilt, mis on mõistetav ka valdkonnaeksperdile.

Selleks, et arvuti operatsioonisüsteem suudaks valdkonnapõhises keeles kirjeldatud käsklusi täita, tuleb valdkonnapõhine keel tõlkida üldkasutatavasse keelde. MPS-is defineeritakse tõlkimisreeglid generaatori aspektis. Tõlkimiseks kasutab MPS generatiivset meetodit ning parserit selles protsessis ei ole. See tähendab, et lõppkasutaja poolt kirjutatud valdkonnapõhise keele AST muudetakse samm sammu haaval üldkasutatava keele AST-ks. Oluline on siinkohal, et mudeli kuju küll muutub, kuid semantika, ehk koodi tähendus jääb samaks. Uue üldkasutatava keele mudeli suudab arvuti käivitada ning valdkonnapõhise keelega antud käsklused täita.

MPS võimaldab lisaks kolmele põhilisele aspektile kirjeldada veel üle kümne erineva aspekti. Näiteks kasutajale soovituste andmine ja veateated. Aspektide keerukuse ning ajapiirangute tõttu käsitletakse selles kirjatöös ainult eelnevalt kirjeldatud kolme põhilist aspekti.

2.2 Kasutatud arenduskeskkonnad ja mustrid

Antud töö põhiliseks arendusvahendiks on JetBrainsi MPS. Nagu eelnevas peatükis mainitud tuleb valdkonnapõhine keel tõlkida üldkasutatavasse keelde. Antud töös on üldkasutatavaks keeleks Java programmeerimiskeel. Töö käigus luuakse ka valdkonnapõhiste keeltega sama funktsionaalsust täitev näiteprojekt, mis on realiseeritud täielikult Javas. Näiteprojektis kasutatakse teist JetBrainsi arendusvahendit Intellij Idea Community.

Intellij Idea on üks levinumaid Java kirjutamiseks mõeldud arenduskeskkondasid. Esimene versioon Intellij Ideast avaldati aastal 2001 ning selle eelisteks konkurentide ees on põhjalik koodi analüüs, intelligentne koodi soovituste andmine ja refaktoreerimise võimalused [11]. Java on üldkasutatav, objektorienteeritud programmeerimiskeel. Keele töötas 1991. aastal välja Sun Microsystems ning Java süntaks on disainitud C++ programmeerimiskeele järgi [12]. Vaatamata oma pikale ajaloole ja kiiresti muutuvale infotehnoloogia sektorile on Java siiani üks kõige laialdasemalt kasutatavaid programmeerimiskeeli äritarkvara loomiseks [13].

Loodud valdkonnapõhistes keeltes ning näiteprojektis realiseeritakse väike osa Jim Arlowi ja Ila Neustadti poolt kirjeldatud koguse (*Quantity*) arhetüüp mustrist. Arhetüüp mustrid kirjeldavad kõige sagedamini äritarkvaras esinevaid osasid. Näiteks kirjeldavad

arhetüüp mustrid toodet, inventuuri, raha ja tellimust [14]. Nende mustrite kasutamine äriloogikas võimaldab koodi taaskasutamist erinevate projektide vahel, kiirendab tarkvaraloome protsessi ning parandab koodi loetavust. Töös realiseeritud koguse arhetüüp musteri abil saab tarkvaras väljendada numbrilist väärtust soovitud mõõtühikus, näiteks 15 m/s (meetrit sekundis) või 10 min (minutit). Koguse arhetüüp mustri koodinäidete aluseks on võetud Dr Gunnar Piho doktoritöö [15].

Lisaks koguse arhetüüp mustrile lähtutakse töö käigus puhta koodi (*Clean code*) parimatest tavadest. Seda nii valdkonnapõhiste keelte kui näiteprojekti puhul. Puhas kood on väga laialdane mõiste ning oleneb suuresti koodi kirjutajast. Robert C. Martin toob oma raamatus „Clean Code“ välja puhta koodi kõige olulisemad osad, milleks on objektide üheselt mõistetav nimetamine, koodi lugemise lihtsus, funktsioonide taaskasutatavus ning põhimõte, et iga meetod peaks täitma rangelt ühte funktsionaalsust [16].

2.3 Tööprotsess

Tööprotsessi käigus disainitakse MPS-is kaks valdkonnapõhist keelt ning luuakse üks näiteprojekt objektorienteeritud meetodil. Näiteprojekt realiseerib valdkonnapõhiste keeltega sama funktsionaalsust ning näiteprojekti abil võrreldakse tarkvara loomise efektiivsust MPS-i ja objektorienteeritud meetodi vahel. Peale valdkonnapõhiste keelte disainimist kasutatakse loodud valdkonnapõhiseid keeli nii, nagu seda teeksid keele lõppkasutajad. Selle tulemusena genereerib MPS üldkasutatava Java koodi. MPS-i poolt genereeritud koodi võrreldakse objektorienteeritud meetodil kirjutatud näiteprojekti koodiga. Võrreldakse koodi puhtust, koodi loome protsessi lihtsust ja koodi kirjutamiseks kulunud aega.

3 Valdkonnapõhised keeled MPS-is ja näiteprojekt

Koguse arhetüüp muster realiseeritakse algelisel kujul nii valdkonnapõhistes keeltes kui näiteprojektis. Esimese loodud valdkonnapõhine keel võimaldab genereerida kolm põhilist tarkvara osa: klassid, klasside omadused (*property*) ning klasside vahelised seosed. Teine keel võimaldab kasutajal defineerida spetsiifiliselt koguse arhetüüp mustrile omaseid elemente, tavaliste klassi väljade näol. Näiteprojektis on realiseeritud mõlema keele funktsionaalsused.

3.1 Esimene valdkonnapõhine keel

Esimese MPS-is loodud valdkonnapõhise keele „*ClassGenerator*“ eesmärk on kõikide tarkvaras esinevate klasside, klassi omaduste ning klasside vaheliste seoste kirjeldamine mugavalt ning kergesti hoomatavalt. Keelega „*ClassGenerator*“ käsitletav valdkond on seega Java klasside genereerimine. Loodud keele süntaks võimaldab Java klasse defineerida ilma Java kogemuseta. Joonisel 3 on välja toodud keele tühi mall, mida keele lõppkasutaja täitama hakkab ning joonisel 4 on toodud näide keele lõppkasutaja poolt kirjutatud valdkonnapõhisest keelest.

```
Nimi: <no name>

Klassid:
<< ... >>

Klasside vahelised seosed:
<< ... >>
```

Joonis 3. „*ClassGenerator*“ valdkonnapõhise keele tühi mall.

```
Nimi: Sõidukid

Klassid:
auto, buss, uks, sõiduk

Klasside vahelised seosed:
auto on sõiduk
auto omab uks
buss on sõiduk
```

Joonis 4. Näide „*ClassGenerator*“ valdkonnapõhise keele kasutaja kirjutatud koodist.

Joonisel 4 toodud keele näite põhjal genereerib MPS viis *java* laiendusega faili. Viiest failist neli esitavad joonisel 4 kirjeldatud klasse ning viimane tuleneb MPS-i spetsiifikast. Klasse kirjeldavad failid esitavad „auto“, „buss“, „uks“ ja „sõiduk“ klasse koos omaduste ja klasside vaheliste seostega. Klasse kirjeldavate failide hulk oleneb kirjeldatud klasside arvust. Viienda failina genereeritakse üks MPS-i spetsiifikast tulenev *java* laiendusega fail nimega „sõidukid“, mis esitab keele juur elementi. Antud juurelement genereeritakse olenemata kirjeldatud klasside arvust.

Valdkonnapõhise keele *„ClassGenerator“* poolt käsitletav valdkond on Java klasside genereerimine. Antud valdkonna eksperdiks on näiteks tarkvaraarendaja või infosüsteemi analüütik ning valdkonna põhiobjektideks on „klass“, „seos“ ja „seose tüüp“. Neid põhiobjekte kasutab valdkonnapõhise keele lõppkasutaja Java klasside, klasside omaduste ja klasside vaheliste seoste kirjeldamiseks. Objekt „klass“ esitab antud valdkonnas Java klassi. Objekt „seos“ esitab klasside vaheliste seoste olemasolu ning objekt „seose tüüp“ esitab klasside vaheliste seoste tüüpi.

Vastavalt seose tüübile, mille väärtus „*ClassGenerator*“ valdkonnapõhises keeles on kas „on“ või „omab“ seatakse eelnevalt kirjeldatud klassid omavahelistesse seosetesse. Seose tüübile „on“ vastab Javas klassi laiendamine võtmesõnaga *extends*. Vastavalt joonisel 4 olevale näitele, on defineeritud klassid „buss“ ja „sõiduk“ ning need klassid on seatud omavahel „on“ tüüpi seosesse. Selliselt kirjeldatud seose põhjal genereeritakse kood, mis on esitatud joonisel 5.

```
package myComp.mps.beginner.classGenerator.sandbox;

/*Generated by MPS */


public class buss extends sõiduk {
}
```

Joonis 5. „*ClassGenerator*“ valdkonnapõhise keele poolt genereeritud koodi näide, mis esitab „on“ tüüpi seost.

Seose tüübile „omab“ vastab Javas klassi omaduste ehk *Bean property* kirjeldamine [17]. Javas koosneb klassi omadus privaatsest väljast ning avalikust *getter* ja *setter* paarist. Joonisel 4 oleva näite põhjal genereeritakse klasside „auto“ ja „uks“ ning nende vahelise seose põhjal kood, mis on esitatud joonisel 6.

```
public class auto extends sõiduk {
  private uks myUks;
  public uks getUks() {
    return this.myUks;
  }
  private void _setUks(uks value) {
    this.myUks = value;
  }
  private uks setUks(uks value) {
    _setUks(value);
    return value;
  }
  private void refToUks() {
    new Reference<uks>() {
      public uks get() {
        return getUks();
      }
      public void set(uks value) {
        _setUks(value);
      }
    };
  }
}
```

Joonis 6. „*ClassGenerator*“ valdkonnapõhise keele poolt genereeritud koodi näide, mis esitab „omab“ ja „on“ tüüpi seoseid.

Klasside „auto“ ja „uks“ vahelise seose põhjal genereerib MPS Javas privaatse välja nimega „myUks“, avaliku *getter* ja *setter* paari ning viite „uks“ tüüpi muutujale. Genereeritud Java koodi ülesehitus defineeritakse MPS-is valdkonnapõhise keele disaineri poolt. Joonisetel 7 ja 8 on esitatud väike osa „*ClassGeneratoris*“ defineeritud generaatori aspektist.

```
root template
input Class

public class $[map_Class] {
  $LOOP$[public ->$[Class] $[propName] {get; private set;}]
}
```

Joonis 7. Näide generaatori aspekti realisatsioonist, mis genereerib klassid ilma „on“ tüüpi seoseta.

```
root template
input Relation

public class $[map_Relation] extends ->$[Object] {
  $LOOP$[public ->$[Class] $[propName] {get; private set;}]
}
```

Joonis 8. Näide generaatori aspekti realisatsioonist, mis genereerib klassid „on“ tüüpi seosega.

Generaatori realiseerimiseks kasutatakse Javat koos MPS-i poolt pakutavate lisa funktsionaalsustega. „*ClassGenerator*“ keele puhul analüüsib MPS kirjeldatud klasse ja

nende vahelisi seoseid. Vastavalt seose tüübile valib MPS, kas Java koodi genereerimiseks kasutatakse joonisel 7 või joonisel 8 esitatud malli. Kui analüüsitav klass on mõne teise kirjeldatud klassiga „on“ tüüpi seoses, siis kasutatakse joonisel 8 esitatud malli, mis lubab Java koodis *extend* tüüpi seoseid. Mõlema malli puhul itereeritakse läbi kõik analüüsitava klassi „omab“ tüüpi seosed, kui neid eksisteerib. Valdkonnapõhises keeles lõppkasutaja poolt kirjeldatud klassi nimede põhjal võimaldab MPS määrata Javas genereeritavate klasside nimed ja omaduste tüübid.

3.1.1 MPS-i spetsiifika „*ClassGeneratoris*“

„*ClassGenerator*“ keele disainimiseks on lisaks eelpool mainutud põhiobjektidele kasutatud veel kolme objekti, mis tagavad keele soovitud käitumise. Keele käitumine näitab, kuidas MPS reageerib lõppkasutaja sisestustele ning keele käitumisreeglid defineerib keele disainer valdkonnapõhist keelt luues. Näitena keele käitumise kohta lubab MPS joonisel 4 toodud valdkonnapõhises keeles seose tüübi kohale kirjutada ainult „on“ või „omab“ ja teiste sisestuste puhul annab MPS veateate. Teise näitena lubab MPS klasside vaheliste seoste kirjeldamisel kasutada vaid neid klasside nimesid, mis on eelnevalt samas dokumendis kirjeldatud.

Keele käitumiseks vajalikud objektid on „*ClassGenerator*“, „klassi viide“ ja „seose tüübi viide“. Nagu peatükis 2.1.1 mainitud, on valdkonnpõhiste keelte kasutamine MPS-is otse abstraktse süntaksipuu muutmine. Sarnaselt kõigile puu–tüüpi struktuuridele, vajab ka abstraktne süntaksipuu juurelementi. Objekt „*ClassGenerator*“ on samanimelise valdkonnapõhise keele „C*lassGenerator*“ jaoks vajalik juurelement. Joonisel 4 tood näites kajastub objekt „*ClassGenerator*“ esimesel real, kus defineeritakse juur elemendi nimi. Objektid „klassi viide“ ja „seose tüübi viide“ on vajalikud selleks, et keelt kirjutades saaks lõppkasutaja sisestada ainult eelnevalt defineeritud klasse ning seose tüübina saaks sisestada vaid tüüpe „on“ ja „omab“.

3.2 Teine valdkonnapõhine keel

Teine MPS-is disainitud valdkonnapõhine keel on „*Quantity*“. „*Quantity*“ keele poolt käsitletavaks valdkonnaks on koguse arhetüüp mustri realiseerimine tarkvaras. „*Quantity*“ eesmärgiks on valdkonnpõhises keeles kirjeldada koguse arhetüüp mustri põhielemente ning kirjelduse põhjal genereerida põhielemendid Java koodis klassi väljade (*field*) näol. Koguse mustri põhielemendid, mida „*Quantity*“ keelega kirjeldatakse

on baas mõõde, tuletatud mõõde, ühik ning kogus. Koguse mustris üks põhielemente on ka arv, kuid see on MPS-i poolt *integer* muutuja tüübina eelnevalt kirjeldatud.

Quantity keelt kasutades peab keele lõppkasutaja esmalt defineerima baas mõõtmed. Baas mõõtmed defineeritakse läbi nime ning sümboli. Joonisel 9 on toodud näide „*Quantity*“ keeles baas mõõtmete defineerimisest.

```
Baas mõõtmed:
Nimi: aeg  Sümbol: t
Nimi: pikkus  Sümbol: s
```

Joonis 9. Baas mõõtme defineerimine keele lõppkasutaja poolt.

Tuletatud mõõtmete kirjeldamisel peab lõppkasutaja defineerima tuletatud mõõtme nime, sümboli ning valemi. Valem näitab loogikat, kuidas baas mõõtmete kombineerimisel saada tuletatud mõõde. Valemi kirjeldamiseks esitab lõppkasutaja kõigi valemis esinevate baas mõõtmete nime ning astme, eraldades erinevad baas mõõtmed komadega. Tuletatud mõõtmete kirjeldamisel saab kasutada vaid eelnevalt samas dokumendis kirjeldatud baas mõõtmeid. Joonisel 10 on esitatud näide tuletatud mõõtmete kirjeldamisest „*Quantity*“ valdkonnapõhises keeles. Joonisel 10 esitatud näitega on kirjeldatud kiiruse ja pindala mõõtmeid.

```
Tuletatud mõõtmed:
Nimi: kiirus Sümbol: v
  Valem: pikkus 1, aeg -1
Nimi: pindala Sümbol: S
  Valem: pikkus 2
```

Joonis 10. Tuletatud mõõtme defineerimine keele lõppkasutaja poolt.

Kolmanda osana peab keele lõppkasutaja kirjeldama ühikuid. Ühik koosneb mõõtmest, nimest, sümbolist ning tegurist. Ühikus kasutatav mõõde võib olla baas mõõde või tuletatud mõõde, kuid peab olema samas dokumendis eelnevalt kirjeldatud. Ühiku tegur näitab mitu korda erineb antud ühik SI–süsteemi baasühikust. Näiteks on kilomeeter pikkuse baasühikust meeter 1000 korda suurem, seega on tegur 1000 ning millimeeter on baasühikust 1000 korda väiksem, seega on tegur 0,001. Joonisel 11 on esitatud ühikute kirjeldamise näide valdkonnapõhises keeles „*Quantity*“.

```
Ühikud:
Mõõde: kiirus Nimi: meetritSekundis Sümbol: m/s Tegur: 1
Mõõde: kiirus Nimi: kilomeetritSekundis Sümbol: km/s Tegur: 3.6
Mõõde: pikkus Nimi: meeter Sümbol: m Tegur: 1
Mõõde: pikkus Nimi: kilomeeter Sümbol: km Tegur: 1000
```

Joonis 11. Ühikute defineerimine keele lõppkasutaja poolt.

Peale mõõtmete ja ühikute kirjeldamist on lõppkasutajal võimalik kirjeldada koguseid. Selleks on vaja määrata koguse arvuline väärtus ning ühik, milles kogust mõõdetakse. Koguse kirjeldamisel kasutatakse ainult ühiku sümbolit ning kasutatav ühik peab olema eelnevalt samas dokumendis kirjeldatud. Joonisel 12 on esitatud näide koguste kirjeldamisest valdkonnapõhises keeles „*Quantity*“. Joonisel 13 on terviklik näide keele lõppkasutaja poolt kirjutatud „*Quantity*“ valdkonnapõhisest keelest.

```
Kogused:
v = 1 m/s
v = 4 m/s
s = 3 m
```

Joonis 12. Koguste defineerimine keele lõppkasutaja poolt.

```
Nimi: Kogused

Baas mõõtmed:
Nimi: aeg  Sümbol: t
Nimi: pikkus  Sümbol: s

Tuletatud mõõtmed:
Nimi: kiirus Sümbol: v
  Valem: pikkus 1, aeg -1
Nimi: pindala Sümbol: S
  Valem: pikkus 2

Ühikud:
Mõõde: kiirus Nimi: meetritSekundis Sümbol: m/s Tegur: 1
Mõõde: kiirus Nimi: kilomeetritSekundis Sümbol: km/s Tegur: 3.6
Mõõde: pikkus Nimi: meeter Sümbol: m Tegur: 1
Mõõde: pikkus Nimi: kilomeeter Sümbol: km Tegur: 1000

Kogused:
v = 1 m/s
v = 4 m/s
s = 3 m
```

Joonis 13. Näide keele lõppkasutaja poolt kirjutatud „*Quantity*“ valdkonnapõhisest keelest.

Joonisel 13 esitatud näite põhjal genereerib MPS koguse arhetüüp mustri põhielemendid Java koodis, privaatsete väljade näol. „*Quantity*“ valdkonnapõhises keeles on vaja kirjeldada vähemalt baas mõõtmed, ühikud ning kogused, et MPS oleks võimeline Java

koodi genereerima. Joonisel 14 on toodud „*Quantity*“ poolt genereeritud Java koodi näide, mis põhineb joonisel 13 esitatud kirjeldustel.

```
package Quantity.sandbox;

/*Generated by MPS */


public class Kogused {

  private Measure aeg = new Measure("aeg", "t");
  private Measure pikkus = new Measure("pikkus", "s");

  private Measure kiirus = new Measure("kiirus", "v", pikkus, aeg, 1, -1);
  private Measure pindala = new Measure("pindala", "S", pikkus, 2);

  private Unit meetritSekundis = new Unit(kiirus, "meetritSekundis", "m/s", "1");
  private Unit kilomeetritSekundis = new Unit(kiirus, "kilomeetritSekundis", "km/s", "3.6");
  private Unit meeter = new Unit(pikkus, "meeter", "m", "1");
  private Unit kilomeeter = new Unit(pikkus, "kilomeeter", "km", "1000");

  private Quantity quantity_a = new Quantity(1, "m/s");
  private Quantity quantity_b = new Quantity(4, "m/s");
  private Quantity quantity_c = new Quantity(3, "m");
}
```

Joonis 14. Täielik näide „*Quantity*“ valdkonnapõhise keele poolt genereeritud Java koodist.

Genereeritud Java koodis algväärtustatakse väljad kasutades *new* võtmesõna ja vastava klassi konstruktorit. Mõõtmete, ühikute ja koguste omadused esitatakse genereeritud Javas konstruktori parameetritena. Baas mõõtme omadused on näiteks nimi ja sümbol. Joonisel 14 oleva koodi genereerib MPS generaatori aspekti põhjal ning genereeritud kood asub *java* laiendusega failis. Joonisel 15 on toodud väike osa „*Quantity*“ keele generaatorist.

```
$LOOP$[private Measure $[baseMeasure] = new Measure("$[Name]", "$[Symbol]"); ]
```

Joonis 15. „*Quantity*“ valdkonnapõhise keele generaatori realisatsioon.

Joonisel 15 esitatud koodi abil genereeritakse Java koodis baas mõõtmed. MPS kasutab Java koodi genereerimiseks keele lõppkasutaja kirjeldusi. Joonisel 15 esitatud näites itereerib MPS läbi kõik lõppkasutaja poolt kirjeldatud baas mõõtmed, loob *Measure* tüüpi privaatse välja ning algväärtustab välja kasutades *Measure* klassi konstruktorit. Välja nime määrab MPS baas mõõtme nime järgi. Sarnast loogikat kasutades genereerib MPS ka kõik ülejäänud „*Quantity*“ keeles kirjeldatavad ja joonisel 14 esitatud osad.

„*Quantity*“ valdkonnapõhise keele eesmärk on tarkvaras realiseerida koguse arhetüüp muster. Muster realiseeritakse väga algeliselt ning selleks peab keele lõppkasutaja

kirjeldama koguse mustri põhielemendid: baas mõõtmed, tuletatud mõõtmed, ühikud ning kogused. Keele lõppkasutaja kirjelduste põhjal genereeritakse mustri põhielemendid Java koodis privaatsete väljade näol.

3.3 Näiteprojekt objektorienteeritud meetodil

MPS-i tööprotsessi võrdlemiseks üldlevinud meetoditega luuakse Intellij Ideas näiteprojekt. Näiteprojekti kood realiseerib valdkonnapõhiste keeltega „*ClassGenerator*“ ja „*Quantity*“ sama funktsionaalsust. Näiteprojekt on tavaliste Java klasside kogum, milles on kirjeldatud kaheksa klassi. Klassides *Measure*, *Unit* ning *Qunatity* on kirjeldatud ainult klassi omadusi ning konstruktoreid. Täpsemalt on näiteprojektis kirjeldatud *BaseMeasure*, *DerivedMeasure*, *MeasureTerm*, *MeasureTerms* ja *Kogused* klasse.

Näiteprojektis on baas mõõde ning tuletatud mõõde kirjeldatud eraldi, vastavate klassidega *BaseMeasure* ja *DerivedMeasure*. Nii *BaseMeasure* kui *DerivedMeasure* pärinevad klassist *Measure*. Klassis *Measure* on defineeritud omadused *name* ja *symbol* ning neid omadusi kasutavad klassid *DerivedMeasure* ja *BaseMeasure* oma konstruktorites. Joonisel 16 on esitatud klassi *DerivedMeasure* realiseerimine näiteprojektis.

```
package OOPKogusedHelpers;

public class DerivedMeasure extends Measure{
    private MeasureTerms terms;

    public MeasureTerms getTerms(){ return this.terms; }

    public void setTerms(MeasureTerms value){ this.terms = value; }

    public DerivedMeasure(String name, String symbol, MeasureTerms terms){
        this.terms = terms;
        setName(name);
        setSymbol(symbol);
    }
}
```

Joonis 16. Klassi *DerivedMeasure* realisatsioon näiteprojektis.

Klass *DerivedMeasure* kasutab tuletatud mõõdu valemi kirjeldamiseks klassi *MeasureTerms*. Klassis *MeasureTerms* on defineeritud list, mis sisaldab endas kõiki tuletatud mõõdu valemis esinevaid mõõte. Klassis *MeasureTerms* kirjeldatud funktsiooni *Add* abil on võimalik mõõte valemisse lisada. Listi tüüp on *MeasureTerm* ning klassis

MeasureTerm on kirjeldatud mõõt ja mõõdu aste. Joonisel 17 on esitatud klassi *MeasureTerms* realisatsioon ja joonisel 18 klassi *MeasureTerm* realisatsioon.

```
package OOPKogusedHelpers;

import java.util.List;

public class MeasureTerms {
    private List<MeasureTerm> measureTerms;

    public List<MeasureTerm> getMeasureTerms() {
        return measureTerms;
    }

    public void setMeasureTerms(List<MeasureTerm> measureTerms) {
        this.measureTerms = measureTerms;
    }

    public void Add(MeasureTerm term){
        this.measureTerms.add(term);
    }
}
```

Joonis 17. Klassi *MeasureTerms* realisatsioon näiteprojektis.

```
package OOPKogusedHelpers;

public class MeasureTerm {
    private Measure measure;
    private int power;

    public Measure getMeasure() { return measure; }

    public void setMeasure(Measure measure) { this.measure = measure; }

    public int getPower() { return power; }

    public void setPower(int power) { this.power = power; }

    public MeasureTerm(Measure measure, int power){
        this.measure = measure;
        this.power = power;
    }
}
```

Joonis 18. Klassi *MeasureTerm* realisatsioon näiteprojektis.

Klasside *MeasureTerm* ja *MeasureTerms* kasutamist tarkvaras illustreerivad joonis 19 ja joonis 20. Joonisel 20 on esitatud kiiruse ja pindala valemi defineerimine, kus muutujad „kiiruseValem“ ja „pindalaValem“ on eelnevalt defineeritud klassi väljad.

```
public class Kogused {
    private BaseMeasure aeg = new BaseMeasure( name: "aeg",  symbol: "t");
    private BaseMeasure pikkus = new BaseMeasure( name: "pikkus",  symbol: "s");

    private MeasureTerms kiiruseValem = new MeasureTerms();
    private MeasureTerms pindalaValem = new MeasureTerms();

    private DerivedMeasure kiirus =
            new DerivedMeasure( name: "kiirus",  symbol: "v", kiiruseValem);
    private DerivedMeasure pindala =
            new DerivedMeasure( name: "pindala",  symbol: "S", pindalaValem);

    private Unit meetritSekundis =
            new Unit(kiirus,  name: "meetritSekundis",  symbol: "m/s",  factor: 1);
    private Unit kilomeetritSekundis =
            new Unit(kiirus,  name: "kilomeetritSekundis",  symbol: "km/s",  factor: 3.6);
    private Unit meeter =
            new Unit(pikkus,  name: "meeter",  symbol: "m",  factor: 1);
    private Unit kilomeeter =
            new Unit(pikkus,  name: "kilomeeter",  symbol: "km",  factor: 1000);

    private Quantity quantity_a = new Quantity( amount: 1, meetritSekundis);
    private Quantity quantity_b = new Quantity( amount: 4, meetritSekundis);
    private Quantity quantity_c = new Quantity( amount: 3, meeter);

    public void Initailize(){...}
}
```

Joonis 19. Koguse mustri kasutamine näiteprojektis.

```
public void Initailize(){
    kiiruseValem.Add(new MeasureTerm(pikkus, power: 1));
    kiiruseValem.Add(new MeasureTerm(aeg, power: -1));

    pindalaValem.Add(new MeasureTerm(aeg,  power: 2));
}
```

Joonis 20. Kiiruse ja pindala valemi defineerimine koguse mustri kasutamise jaoks.

Joonisel 19 esitatud klass *Kogused* illustreerib koguse mustri kasutamist tarkvaras, defineerides klassi väljade näol baas ja tuletatud mõõtmed, tuletatud mõõtmete valemid, ühikud ning kogused. Valemid defineeritakse klassis *Kogused* funktsiooniga *Initialize*. Joonisel 19 toodud näitega on kirjeldatud samad klassi väljad, mis „*Quantity*“ valdkonnapõhise keele näitega joonisel 13.

Näiteprojekti eesmärk on tekitada võrdlusmoment erinevate tehnoloogiate vahel. Selleks defineeritakse tarkvaraliselt koguse arhetüüp mustri põhielemendid. Näiteprojekt koosneb kaheksast Java klassist, millest seitse on koguse arhetüüp mustri kirjeldamiseks ja üks on mustri kasutamiseks. Mustri kasutamiseks kirjeldatakse mustri põhielemendid privaatsete klassi väljade näol.

4 Tulemuste analüüs

Autori hinnangul on MPS hästi töötav integreeritud arenduskeskkond, kuid pole sobiv töövahend valdkonnapõhiste keelte loomiseks ning kasutamiseks. See hinnang põhineb varasema tarkvaraarendus kogemuse, MPS-is kahe valdkonnapõhise keele loomise ning objektorienteeritud meetodiga sama funktsionaalsuse realiseerimise põhjal. Peamised argumendid antud hinnangu kujunemisel olid MPS-i tööprotsessi keerukus, õppimiseks ja arendamiseks kulunud aeg ning MPS-i poolt seatud piirangud genereeritud koodile. Lisaks kujundasid autori hinnangut ka teiste autorite uurimused MPS-i kohta.

4.1 Martin Fowler keele tööpinkidest ja valdkonnpõhistest keeltest

Martin Fowleri hinnangul on valdkonnapõhiste keelte abil võimalik tõsta tarkvaraarendaja efektiivsust ning parandada suhtlust valdkonna ekspertide ja tarkvaraarendajate vahel. Eemaldades vajaduse luua tõlke ja lihtsustades valdkonnapõhistee keelte disainimist, näeb Fowler keele tööpinke võimalusena suurendada valdkonnapõhiste keelte kasutamist. Samas toob Fowler keele tööpinkide ja valdkonnapõhiste keelte kohta välja mitmeid puudujääke. Peamine nendest on seotud küsimusega, kas valdkonnapõhisest keelest saadav kasu on suurem kui uue valdkonnapõhise keele loomiseks ja õppimiseks kulunud ajaline ressurss [3]. Viimasele küsimusele Fowler oma otsest hinnangut ei anna, kuid ta on välja toonud, et tehnoloogiate hulk, mida tarkavaraarendajad peavad omandama on niigi suur [18].

Fowler rõhutab, et valdkonnapõhise keele kasutusele võtmiseks peab see olema võimalikult lihtne ning selgelt piiritletud funktsionaalsusega. Sellise abstraktsiooni loomine vajab kogemusi tarkvaraarenduses ning teadmisi konkreetses valdkonnas. Martin Fowleri hinnangul parandavad keele tööpingid nagu MPS valdkonnapõhiste keelte loomise protsessi. Probleemiks on aga ühtse standardi puudumine erinevate tööpinkide vahel. Standardi puudumine tähendab seda, et loodud valdkonnapõhist keelt saab kasutada vaid ühes tööpingis ning keele üleviimine teiste tööpinkide peale otse on võimatu [3].

Martin Fowler on välja toonud, et tema hinnangul valdkonna eksperdid ise tarkvara ärinõudeid ei kirjuta, kuigi keele tööpingid ja valdkonnpõhised keeled peaksid seda võimaldama. Põhjus on piisavalt mugava ja veakindla vahendi puudumine. Fowleri hinnangul oleks juba piisavalt hea, kui valdkonnapõhised keeled võimaldaksid ekspertidel lugeda tarkvaraarendajate poolt kirjeldatud tarkvara nõudeid. Tarkvara nõuete kirjeldamine valdkonna ekspertide poolt otse on Fowleri hinnangul liiga suur eesmärk [19].

4.2 Teised MPS-i uurimused

Daniel Ratiu, Vaclav Pechi ja Kolja Dummanni poolt välja antud aruanne kirjeldab nende mitme aasta kogemusi MPS-i õpetamisel. Aruanne toob välja, et MPS-iga valdkonnapõhiste keelte loomine nõuab laialdasi teadmisi nii valdkonnapõhistest keeltest, kui üldlevinud arendus metoodikatest nagu objektorienteeritud programmeerimine. Kõik autorite koolitustel osalejad on mitmeaastase tarkvaraarenduse kogemusega ning seda koolitajad ka eeldavad [20]. Raamat „*The MPS Language Workbench: Volume I*“ toob samuti välja, et MPS on suunatud kogemustega tarkvaraarendajatele, mitte valdkonna ekspertidele [21].

Ratiu, Pechi ja Dummanni aruandes on välja toodud, et nende osalejate joaks, kes pole varasemalt valdkonnapõhiste keeltega kokku puutunud on MPS-i kasutamine suur katusumus. Põhjuseks on MPS-i enda keerukus ning valdkonnapõhiste keelte disaini põhimõtted nagu õigete valdkonna põhiobjektide valimine, parima mudeli koostamine ja sobiva süntaksi defineerimine [20].

Aruande autorite peamised tähelepanekud MPS-i kohta on järsk õppimiskõver, õpilaste vajadus abi järele ka peale koolitust ning hästi töötava keele loomise keerukus. Järsu õppimiskõvera peamiseks põhjuseks toovad koolitajad väga laialdased teadmised MPS-i kohta, mida õpilased peavad omandama enne selle kasutamist. MPS-i erinev terminoloogia teistest arendusvahenditest ning teksti kirjutamise piirangud tänu abstraktsele süntaksipuule suurendavad veelgi õppimiskõverat. Koolitajate hinnangul vajab lihtsa ning hästi töötava valdkonnapõhise keele loomine pikaldasi kogemusi ja oskusi. Oskuste vajalikkus muudab MPS-iga keelte disainimise keeruliseks just algajatele tarkvaraarendajatele [20].

Kirjatöö, mis käsitleb MPS-i lahendust SPLASH 2016 *Language Workbench Challenge* [22] ülesannetele, kirjeldab teksti töötlust MPS-is. Töös rõhutatakse, et teksti muutmisel MPS-is muudetakse otse abstraktset süntaksipuud ning see tekitab lisa töö keele disainerile. Näiteks tavalise liitmistehte realiseerimisel peab disainer kindlaks tegema, et kasutaja ei sisesta numbrite asemel midagi muud, numbrite vahel oleks liitmisemärk ning et kõik elemendid oleksid õiges järjekorras [23].

SPLASH 2016 ülesandeid kirjeldavas kirjatöös tuuakse veel välja, et tekstitöötlus on raskendatud ka valdkonnapõhise keel lõppkasutaja jaoks. MPS võimaldab lõppkasutajal muuta ainult puu elemente, kuid mitte kogu dokumendis olevat teksti. Dokumendi osad, mis pole puu elemendid on ainult keele presentatsiooni osad ning nende muutmist MPS ei võimalda. Neid osasid saab muuta ainult keele disainer, keelt luues või hiljem muutes. Lisaks toob kirjatöö välja, et MPS-is lisab valdkonnapõhise keele kasutaja kogemusele keerukust teksti kopeerimise ja kleepimise funktsioon. Kopeerimine ja kleepimine on võimalik, kui kopeeritakse tervet abstraktse süntaksipuu osa, kuid üksikute sõnade kopeerimist MPS ei luba [23].

Juhtumiuuringus(*Case study*), mis käsitleb kogemusi MPS-iga 10-ne aasta jooksul jõutakse järeldusele, et MPS on valmis reaalse maailma projektide jaoks. Peamisteks argumentideks on erinevate keelte haldamise ja koos kasutamise lihtsus, projektsiooniline tekstitöötlus [4], keelte disainimise metoodika ning MPS-i vastupidavus suuremahulisetele projektidele ja töökoormusele. Samas tuuakse selgelt välja, et iga argumendipuhul esineb ka kitsaskohti. Peamiste kitsaskohtadena toovad autorid välja koodi silumise (*debug*) ning teksti muutmise keerukuse. MPS-i koodi silumise võimalused on nende hinnangul algelised ning vähe informatiivsed. Teksti muutmise teeb keerukaks abstraktne süntaksipuu, mille tõttu takistab MPS vaba teksti muutmist [24].

4.3 Autori järeldused

Kahe valdkonnapõhise keele ning võrdleva projekti realiseerimine tulemusena jõudis autor järeldusele, et MPS-i suurim puudujääk on eelnevate oskuste olemasolu vajalikkus. Seda kinnitab ka Ratiu, Pechi ja Dummanni poolt välja antud aruanne. Lisaks MPS-i spetsiifilistele oskustele peab valdkonnapõhise keele disaineril olema varasem kogemus ka üldkasutatavate keeltega. Seda illustreerib autori poolt loodud mõlema valdkonnapõhise keele generaatori realisatsioon, milles on suures mahus Java koodi.

Lisaks peab keele disainer olema kursis abstraktse süntaksipuu, puhta koodi põhimõtete, konkreetse valdkonna ning objektorienteeritud programmeerimise kontseptsiooniga. Viimane on oluline, et kirjeldada valdkonnapõhises keeles põhiobjekte võimalikult efektiivselt ning seda rõhutavad ka Martin Fowler, Ratiu, Pech ja Dummann.

Kahe valdkonnapõhise keele loomise põhjal järeldas autor, teiseks suureks probleemiks MPS-is on tekstitöötlus. MPS lubab oma kodulehel, et tekstitöötlus on sama mugav, kui vaba teksti muutmine [6], kuid autori hinnangul see nii ei ole. Autori hinnangut kinnitavad kõik eelnevalt käsitletud uurimused MPS-i kohta. Teksti muutmine on võimalik vaid selleks ettenähtud kohtades ning sellisel käitumisel on autori hinnangul kaks külge. Valdkonnapõhise keele lõppkasutaja vaatenurgast, on teksti muutmise kitsendus hea, vähendades vigade tõenäosust. Valdkonnapõhise keele disainerile seab see aga ette mitmeid takistusi, näiteks „*Quantity*“ keele generaatoris, mis loob ühiku kirjelduse põhjal *Unit* tüüpi Java välja. Autori eesmärk oli ühiku konstruktoris panna kõrvuti mõõdu nimi ning aste, kuid MPS-i piirangute tõttu ei olnud see võimalik. Sellise piiranguga harjumine nõuab arendajalt aega ning võib autori hinnangul tekitada meelehärmi kohtades, mis oleksid tavameetodiga lihtsasti lahendatavad.

Autoril kulus MPS-i õppimiseks ning kahe valdkonnapõhise keele *Quantity* ja *ClassGenerator* kirjutamiseks ligikaudu poolteist kuud, tegeledes iganädalaselt MPS-iga umbes kaheksa tundi. Arvestades aja kulu ning tõsiasja, et keeled realiseerivad väga algelisi funktsionaalsusi, on autori hinnangul MPS-i õppimiskõver liiga pikk ning ei tasu ennast ära. Kaks loodud keelt on autori hinnangul küll kasutatavad, kuid oma olemuselt väga lihtsad ning vigade altid. Keeled on defineeritud kasutades ainult kolme põhilist keele kirjeldamise elementi ning realiseerimata on jäänud suurem osa MPS-i võimalustest, mis reaalse projekti puhul oleksid vajalikud. Õppimiskõvera pikkuse ja keerukuse toovad välja ka Ratiu, Pech ja Dummann oma aruandes.

Mudelipõhise tarkvaraarenduse üks suurimaid probleeme on äriloogikas olevate meetodite sisu genereerimine. Autori hinnangul ei lahenda MPS seda probleemi, sest see eeldaks väga põhjaliku valdkonnapõhise keele defineerimist ning valdkonnapõhise keele maht kasvaks kiiresti. See läheb vastuollu Martin Fowleri põhimõttega, et valdkonnpõhine keel peaks olema lihtne ja lühike. Illustreeriv näide on valdkonnapõhine keel *Quantity*, mis realiseerib üsna detailse koodi genereerimist. *Quantity* keele loomiseks kulunud aja, keele vähese funktsionaalsuse ning piisava kasutamis keerukuse põhjal

järeldab autor, et MPS ei paku head lahendust äriloogikas olevate meetodite sisu genereerimiseks. Lisaks järeldab autor *Quantity* keele loomise tulemusena, et laialdasema funktsionaalsuse tagamisega kasvaks loodava valdkonnapõhise keele keerukus liiga suureks.

JetBrains toob oma kodulehel välja, et MPS-i üks eesmärk on suhtluse parandamine valdkonna ekspertide ja tarkvaraarendajate vahel, kasutades valdkonnapõhiseid keeli. Loodud valdkonnapõhiste keelte põhjal järeldab autor, et seda eesmärki MPS ei täida. Autori hinnangul parandab suhtlust korralikult realiseeritud valdkonnapõhine keel, kuid keele loomine toimub siiski valdkonna eksperdi ja tarkvaraarendaja koostöös. Sellise koostöö käigus toimub kahe osapoole vahel suhtlus sama moodi nagu tavameetodil tarkvaraarenduse puhul ning esinevad ka samad probleemid, milleks on info edastamise puudulikkus ning valesti mõistmine. Seega tõstab MPS autori hinnangul probleemi lihtsalt teise kohta, mitte ei lahenda seda.

Teiseks MPS-i eesmärgiks on võimaldada valdkonna eksperdil muuta tarkvara äriloogikat ilma tarkvaraarendaja abita. Selle eesmärgi täidab MPS autori hinnangul teatud piirini. Valdkonnapõhiste keelte loomise käigus jõudis autor järeldusele, et suurem osa arendusest ning keele disainimisest toimub Java koodis. Autori hinnangul tähendab see, et kui uueneb tehnoloogia, näiteks andmebaasi süsteem või soovib keele kasutaja lisa funktsionaalsusi on muudatuste tegemiseks siiski vaja tarkvaraarendajat. Seega on autori hinnangul MPS-i kasutamine ilma tarkvaraarendajata võimalik tingimustel, et süsteem ei vaja pidevaid uuendusi ning kasutajal ei teki lisa funktsionaalsuse vajadusi.

Kahe loodud valdkonnapõhise keele ja näiteprojekti põhjal järeldab autor, et MPS ei lahenda valdkonnapõhiste keeltega ning üldiselt tarkvaraarenduses esinevaid probleeme. Tarkvara loomine üldkasutatavate meetoditega on efektiivsem ning tarkvara muutmine ilma tarkvaraarendaja abita ei ole MPS-i abiga saavutatav. Autor leiab, et MPS on hästi töötav arenduskeskkond, kuid pole mõeldud reaalse tarkvara loomiseks. Autori hinnang MPS-i kohta ei ole lõplik tõde, kuid hinnang põhineb autori varasematel kogemustel tarkvaraarenduses ja valdkonnapõhiste keelte loomisel MPS-is.

5 Kokkuvõte

Valdkonnapõhised keeled on lihtsa süntaksiga programmeerimiskeeled, mille eesmärk on kindla valdkonna probleemide lahendamine. Valdkonnapõhised keeled on suunatud valdkonna ekspertidele, kuid vaatamata valdkonnapõhiste keelte pikale ajaloole, pole neid siiani valdkonna ekspertide poolt kasutusele võetud. Peamiseks põhjuseks on piisavalt mugava töövahendi puudumine. Selle töö eesmärk on välja selgitada, kas ettevõtte JetBrains poolt loodud arenduskeskkond MPS (*Meta Programming System*) on sobiv vahend valdkonnapõhiste keelte loomiseks ning kasutamiseks.

Töö käigus luuakse MPS-iga kaks valdkonnapõhist keelt ning üks näiteprojekt üldlevinud objektorienteeritud meetodil. Põhilisteks töövahenditeks on MPS ja Intellij Idea ning kõik koodi näited on Java programmeerimiskeeles. Valdkonnapõhiste keelte ning näiteprojektiga realiseeritakse koguse arhetüüp muster, mis võimaldab tarkvaraliselt kirjeldata arvulist väärtust kindlas mõõtühikus.

Kahe valdkonnapõhise keele ning näiteprojekti loomise põhjal järeldab autor, et MPS ei ole sobiv vahend valdkonnapõhiste keelte loomiseks. Hinnangu kujunemise peamisteks argumentideks on MPS-i suur õppimiskõver, eelnevate algteadmiste vajadus ning suhtlusbarjääri püsimine tarkvaraarendajate ja valdkonna ekspertide vahel. Autori hinnangut kinnitavad ka teised uurimused MPS-i kohta.

Kasutatud kirjandus

[1] „Manifesto for Agile Software Development,“ 2001. [Võrgumaterjal]. Available: https://agilemanifesto.org/.

[2] A. v. Deursen, P. Klint ja J. Visser, „Domain-Specific Languages: An Annotated Bibliography,“ ACM SIGPLAN Notices, Amsterdam, 2000.

[3] M. Fowler, „Language Workbenches: The Killer-App for Domain Specific Languages?,“ 12 June 2005. [Võrgumaterjal]. Available: https://www.martinfowler.com/articles/languageWorkbench.html.

[4] „How Does MPS Work,“ JetBrains, [Võrgumaterjal]. Available: https://www.jetbrains.com/mps/concepts/.

[5] „MPS FAQ,“ JetBrains, 7 May 2019. [Võrgumaterjal]. Available: https://www.jetbrains.com/help/mps/mps-faq.html.

[6] „MPS Jetbrains,“ JetBrains, [Võrgumaterjal]. Available: https://www.jetbrains.com/mps/.

[7] C. E. F. Pfenning, „Higher-order Abstract Syntax,“ *ACM SIGPLAN Notices - Proceedings of the SIGPLAN '88 conference on Programming language design and implementation,* kd. 23, nr 7, pp. 199-208 , 1988.

[8] B. Meyer, Touch of Class: Learning to Program Well with Objects and Contracts, Berlin: Springer, 2009.

[9] A. Makarkin ja V. Pech, „MPS Structure,“ 27 Detsember 2018. [Võrgumaterjal]. Available: https://confluence.jetbrains.com/display/MPSD20183/Structure.

[10] A. Makarkin ja V. Pech, „MPS Editor,“ 14 Jaanuar 2019. [Võrgumaterjal]. Available: https://confluence.jetbrains.com/display/MPSD20183/Editor.

[11] „Intellij Idea features,“ JetBrains, [Võrgumaterjal]. Available: https://www.jetbrains.com/idea/features/.

[12] L. Lemay ja C. L. Perkins, Teach Yourself Java in 21 Days, Indianapolis: Sams Publishing, 1996.

[13] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang ja L. Réveillère, „Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects,“ *IEEE 37th Annual Computer Software and Applications Conference*, Kyoto, 2013.

[14] J. Arlow ja I. Neustadt, Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML, Indianapolis: Sams Publishing, 2002.

[15] G. Piho, Archetypes Based Techniques for Development of Domains, Requirements and Software : Towards LIMS Software Factory, Tallinn: TUT Press , 2011.

[16] R. C. Martin, Clean Code A Handbook of Agile Software Craftsmanship, Stoughton: Prentice Hall, 2008.

[17] „Java Properties,“ Oracle, [Võrgumaterjal]. Available: https://docs.oracle.com/javase/tutorial/javabeans/writing/properties.html.

[18] M. Fowler, „DslQandA,“ 9 September 2008. [Võrgumaterjal]. Available: https://martinfowler.com/bliki/DslQandA.html.

[19] M. Fowler, „BusinessReadableDSL,“ 15 Detsember 2008. [Võrgumaterjal]. Available: https://martinfowler.com/bliki/BusinessReadableDSL.html.

[20] D. Ratiu, V. Pech ja K. Dummann, „Experiences with Teaching MPS in Industry: Towards Bringing Domain Specific Languages Closer to Practitioners,“ 17-22 September 2017. [Võrgumaterjal]. Available: https://ieeexplore.ieee.org/abstract/document/8101252.

[21] F. Campagne, The MPS Language Workbench: Volume I, CreateSpace Independent Publishing Platform, 2016.

[22] „Challenge, LWC@SLE 2016 Language Workbench,“ [Võrgumaterjal]. Available: https://2016.splashcon.org/track/lwc2016.

[23] E. Schindler, K. Schindler, F. Tomassetti ja A. M. Sutii, „Language Workbench Challenge 2016: the JetBrains Meta Programming System,“ *LWC@SLE 2016 Language Workbench Challenge, Splash2016*, Amsterdam, 2016.

[24] M. Voelter, B. Kolb, T. Szabó, D. Ratiu ja A. v. Deursen, „Lessons learned from developing mbeddr: a case study in language engineering with MPS,“ *Software & Systems Modeling,* kd. 18, nr 1, p. 585–630, February 2019.