TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Adeline Talvik

Aine “Automaadid, keeled ja translaatorid” koodibaasi moderniseerimine Java 21-le

Bakalaureusetöö (9 EAP)

Juhendaja:
Simmo Saan, MSc

Tartu 2025

Aine “Automaadid, keeled ja translaatorid” koodibaasi moderniseerimine Java 21-le

Lühikokkuvõte:

Aastatega on programmeerimiskeelde Java lisatud palju programmeerimist mugavdavaid ja trafarettkoodi vähendavaid konstruktsioone, näiteks kirjemustrid ja lülitiga mustrisobitus. Bakalaureusetöö eesmärk oli moderniseerida Javat kasutava aine “Automaadid, keeled ja translaatorid” koodibaas. Moderniseerimine hõlmas koodibaasis prevalentse verboosse *Visitor*-disainimustri alternatiividega asendamist, klasside kirjeteks teisendamist, abstraktsete klasside sulgliidestega välja vahetamist ning meetodite uuendamist. Töö tulemusel oli lisatud failidest ja dokumentatsioonist hoolimata koodihoidlas umbes 4000 rida vähem sisu.

Võtmesõnad: Java, moderniseerimine, *Visitor*-i disainimuster, lüliti

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Migrating the “Automata, Languages and Compilers” Codebase to Java 21

Abstract:

Over the years, the Java programming language has been updated with many code constructs which make programming easier and reduce boilerplate code, such as record patterns and pattern matching for switches. The goal of this Bachelor’s thesis was to modernise the codebase for the Java-based course “Automata, Languages and Compilers”. This included replacing the Visitor design pattern with alternatives, converting classes to records, converting abstract classes to sealed interfaces and updating methods. By the end of the modernisation process, the line count of the code repository was reduced by approximately 4000, despite the added files and documentation.

Keywords: Java, modernisation, Visitor pattern, switch

CERCS: P170 Computer science, numerical analysis, systems, control

Sisukord

Sissejuhatus

Tartu Ülikooli aine "Automaadid, keeled ja translaatorid" (edaspidi AKT) eesmärk on süvendada arusaamist arvutiprogrammide ülesehitusest ja täitmisest Java programmeerimiskeele abil[1]. Kursus ainekoodiga LTAT.03.006 on kuulunud alates 2017/18. õppeaastast informaatika bakalaureuseõppe õppekavasse.

Aastatega on Javasse lisatud palju programmeerimist mugavdavaid ja trafarettkoodi (ingl *boilerplate*) vähendavaid konstruktsioone. AKT koodihoidlasse on aastate jooksul tehtud erinevaid märkmeid Java uuenduste kohta, kuid ulatuslikku moderniseerimist pole ette võetud.

Bakalaureusetöö eesmärk on uuendada AKT koodibaasi: asendada kasutuses olev verboosne *Visitor*-i disainimuster moodsamate Java 21 võimalustega ning muuta koodi loetavamaks, lühemaks ja kergemini modifitseeritavaks.

Esimeses peatükis avatakse refaktoreerimise mõistet, kirjeldatakse *Visitor*-i disainimustrit ja avatakse tööga seotud Java uuenduste sisu. Teises peatükis iseloomustatakse AKT koodibaasi ja selle moderniseerimise metoodikat. Kolmandas peatükis kirjeldatakse moderniseerimise käigus kerkinud probleeme ja nende lahendusi. Neljandas peatükis vaadeldakse muudatuste kvantitatiivseid näitajaid ja praktikumijuhendajate kommentaare uuendatud koodile.

[1] https://ois2.ut.ee/#/courses/LTAT.03.006/details

1. Tehniline taust

Selles peatükis selgitatakse refaktoreerimise mõistet ja protsessi, tutvustatakse AKT koodibaasis prevalentset *Visitor*-i disainimustrit ning avatakse aastatega Javasse lisatud mugavdavaid konstruktsioone, millega on võimalik *Visitor*-i disainimustrit asendada.

1.1 Refaktoreerimine

Refaktoreerimiseks (ingl *refactoring*) nimetatakse programmikoodi struktuuri lihtsustamist või täiustamist[2]. Refaktoreerimise mõistet hakati kasutama 1990ndatel aastatel, varaste termini populariseerijate seas olid näiteks Opdyke ja Johnson (1992) ning Fowler (1999). Fowler (2018) toob levinumate refaktoriseerimismeetoditena välja järgnevad tegevused:

1. Muutujate ja funktsioonide eraldamine ja kombineerimine;
2. Funktsioonide päiste muutmine (sh funktsiooni ümber nimetamine, argumentide ümber nimetamine, lisamine või eemaldamine);
3. Uue parameetrina edasi antava objekti loomine (nt funktsiooni päises alg- ja lõppkuupäeva argumentide kombineerimine uueks objektiks `dateRange`);
4. Kapseldamine ehk muutujate või omaduste teise olemisse tõstmise.

Refaktoreerimise mõju hindamiseks on erinevaid lähenemisi. Sehgal jt (2022) kasutasid mõõtmisel "lõhnava koodi" (ingl *code smell*; tunnus programmikoodis, mis viitab sügavamale probleemile[3]) tuvastuste arvu ning näitasid, et see on energiakuluga tugevalt positiivselt korreleeritud. Objektorienteeritud keelte, nagu Java ja C++, puhul kasutatakse koodibaasi kvaliteedi ning seega ka refaktoreerimise mõju hindamiseks näiteks Chidamber-Kemerer mõõdustikku, mis hõlmab kuut arvutuslikku näitajat (Chidamber ja Kemerer, 1994).

Mens ja Tourwé (2004) võtavad refaktoreerimise protsessi kokku järgnevate sammudega:

1. Struktuurseid muudatusi vajavate programmilõikude tuvastamine;
2. Rakendatavate faktoreerimisvõtete valimine;
3. Veendumine, et struktuurimuudatused ei mõjuta programmi töökäiku;

[2] https://akit.cyber.ee/term/1617
[3] https://martinfowler.com/bliki/CodeSmell.html

4. Programmikoodi muutmine;
5. Muudatuste mõju (loetavus, arusaadavus jms) ning kulu (aeg, jõudlus jms) hindamine;
6. Ühtluse tagamine muudetud koodi ja teiste programmiga seotud materjalide (dokumentatsioon, nõuded, testid jms) vahel.

Refaktoreerimine pole alati kasulik – seda tasub teha süsteemselt ja mõõdukuse piires. Suurte programmide puhul on koodijupid tihti jaotatud erinevatesse failidesse ning failid omavahel seotud. Seetõttu mõjutab refaktoreerimine tõenäoliselt rohkemat, kui ainult töödeldavat koodijuppi. Refaktoreerimine on ajamahukas, nõuab keskendumist ja võib koodi mõne mõõdiku alusel halvemaks muuta. Näiteks Agnihotri ja Chug (2024) uurimuses sooritatud refaktoreerimisega langesid mõned Chidamber-Kemerer mõõdustiku näitajad.

1.2 *Visitor*-i disainimuster

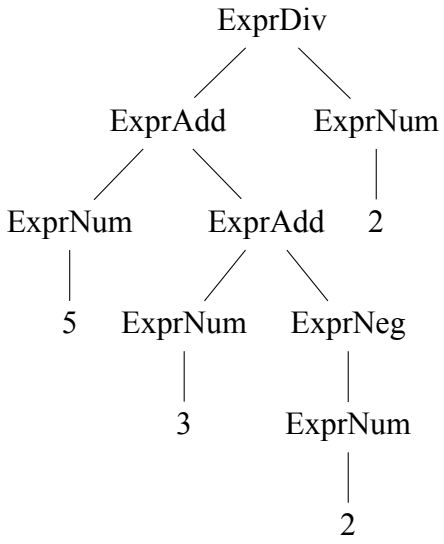
Visitor-i disainimustris (edaspidi autori eestindatud "külastaja") eraldatakse rangelt ligipääsetavad isendid ning isenditel tehtavad operatsioonid, implementeerides toimingud vastavalt ligipääsetava isendi tüübile (Palsberg ja Jay, 1998).

Külastaja disainimustrit demonstreeritakse AKT neljandal nädalal tutvustatava keelega Expr. Keel Expr sisaldab lihtsamaid aritmeetilisi avaldisi, mis koosnevad arvudest, sulgudest ning summa (+), jagatise (/) ning vastandarvu võtmise (−) operatsioonidest. Keelde kuuluvad näiteks järgnevad avaldised:

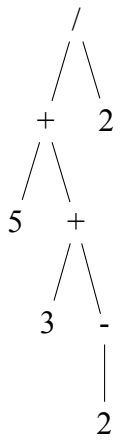
- $-(2 + -6)$, (väärtus 4);
- $-(8/2)$, (väärtus -4);
- $(5 + (3 + -2))/2$, (väärtus 3).

Keeles Expr on kõik avaldised klassi `ExprNode` isendid ning iga avaldise alamavaldis on samuti `ExprNode` isend. Viimane näiteavaldis koosneb ühest jagatisest (alamklass `ExprDiv`), ühest vastandarvu võtmise operatsioonist (alamklass `ExprNeg`), kahest liitmisest (alamklass `ExprAdd`) ning neljast arvust (alamklass `ExprNum`). Joonisel 1a on viimane näiteavaldis esitatud keele Expr klassipuuna ning joonisel 1b avaldispuuna.

Traditsioonilises külastaja disainimustris on igas alamklassis vaja defineerida aktsepteerimismeetod `accept`, et kompilaator saaks tuvastada, mis tüüpi objektiga on



(a) Näiteavaldis klassipuuna.



(b) Näiteavaldis avaldispuuna.

Joonis 1. Puukujul esitused keele Expr avaldisest $(5 + (3 + -2))/2$.

tegemist, ning loodavas külastajas tuleb defineerida iga tüübi jaoks külastamise meetod `visit` (Palsberg ja Jay, 1998).

Keeles Expr on külastaja implementeeritud järgnevalt. Keele Expr jaoks on loodud külastaja klass `ExprVisitor`, milles on külastamise meetod `ExprNode` jaoks ning abstraktsed külastamise meetodid kõikide `ExprNode` alamklasside jaoks (Javas tähistab `T` tüübiparameetrit):

```
public abstract class ExprVisitor<T> {
  public T visit(ExprNode node) {
    return node.accept(this);
  }
  protected abstract T visit(ExprNum num);
  protected abstract T visit(ExprNeg neg);
  protected abstract T visit(ExprDiv div);
  protected abstract T visit(ExprAdd add);
}
```

Abstraktses ülemklassis `ExprNode` on defineeritud abstraktne aktsepteerimismeetod:

```
public abstract class ExprNode {
  public abstract <T> T accept(ExprVisitor<T> visitor);
}
```

Klassides, mis laiendavad (märksõna `extend`) klassi `ExprNode`, on implementeeritud määramata tüübiga aktsepteerimismeetod `accept(ExprVisitor)`, milles kutsutakse välja argumendina antud külastajal meetod `visit`. Näide klassist `ExprDiv`:

```
public class ExprDiv extends ExprNode {
  // ...
  @Override
  public <T> T accept(ExprVisitor<T> visitor) {
    return visitor.visit(this);
  }
}
```

Kui soovitakse külastaja abil mõnd meetodit implementeerida, siis tuleb meetodis luua soovitud tagastustüübiga `ExprVisitor` isend ning iga vastava alamklassi jaoks meetod `visit` üle kirjutada.

Näitena kasutatakse keele Expr väärtustajat – objekti või meetodit, mille abil saab veenduda soovitud lekseemijada tegelikus väärtuses soovitud keskkonnas. Joonisel 2 esitatud külastajaga väärtustamise meetodis `evalWithVisitor(ExprNode)` implementeeritud külastaja `ExprVisitor` käivitab kindla funktsiooni sõltuvalt argumendina saadud tipu tüübist:

- `ExprNum` korral tagastatakse tipu väärtus (tüüpi `Integer`);
- `ExprNeg` korral tehakse rekursiivne kutse tipus sisalduval `ExprNode` isendil (alamtipul) ja tagastatakse selle vastandväärtus;
- `ExprAdd` korral tehakse rekursiivne kutse tipu “vasakul” ja “paremal” alamtipul ning tagastatakse kutsete väärtuse summa;
- `ExprDiv` korral tehakse rekursiivne kutse tipu “lugeja” ja “nimetaja” alamtipul ning tagastatakse kutsete väärtuse jagatis;

Visitor-i mõistet hakati kasutama 1990ndate alguses (Lynch ja Rose, 1994) ning Gamma jt (1995: 331-349) panid kirja külastaja disainimustri põhitõed. Külastaja kasutamine objektorienteeritud keelte puhul on mugav, kuna objektide tüübid meetodites on rangelt määratletud ja muudatusi saab alamklasside piires hõlpsasti sisse kanda (Palsberg ja Jay, 1998).

```java
public static int evalWithVisitor(ExprNode expr) {
  ExprVisitor<Integer> visitor = new ExprVisitor<>() {
    @Override
    public Integer visit(ExprNum num) {
      return num.getValue();
    }
    @Override
    public Integer visit(ExprNeg neg) {
      return -visit(neg.getExpr());
    }
    @Override
    public Integer visit(ExprAdd add) {
      return visit(add.getLeft()) + visit(add.getRight());
    }
    @Override
    public Integer visit(ExprDiv div) {
      return visit(div.getNumerator()) / visit(div.getDenominator());
    }
  };
  return expr.accept(visitor);
}
```

Joonis 2. Keele Expr väärtustamise meetod külastajaga.

Külastaja kasutamisel on mõned kitsaskohad. Iga alamklass nõuab omaette `accept(Visitor)` meetodit, mis tagab tüübikorrektsuse, kuid lisab igasse alamklassi korduvad koodiread; meetodit `accept` pole võimalik ühe korra ülemklassis implementeerida, kuna `accept` peab tagastama külastajale info tipu tüübi kohta. Aine AKT puhul on töö juhendaja sõnul ebamugav, et `visit`-meetodite argumente pole võimalik muuta – kui on soov anda edasi infot näiteks väärtustamiskeskkonna kohta, tuleb seda teha kõrvalistel viisidel, näiteks külastajale isendivälju lisades. Tüüpide range määratlemise tõttu on suuremates programmides vaja luua mitu erineva päisega sarnast meetodit, nagu Connolly Bree ja Ó Cinnéide (2023) näites koera haukumise funktsiooni kohta `barkAt(Animal)`, mis nõuab iga abstraktse klassi `Animal` alamklassi (nt `Cat`, `Sheep`) jaoks eraldi meetodit, isegi kui tehtav heli on loomade puhul identne. Lisaks näitavad Connolly Bree ja Ó Cinnéide samas uurimuses, et olenevalt programmeerimiskeelest, programmi argumentidest ja kompilaatorist võib külastaja kasutamine

võrdlemisi energiamahukas olla: nende Java parser-rakenduses õnnestus traditsioonilise külastaja mustri modifitseerimisel või järjenditöötlusega asendamisel energiakulu kuni 12 protsenti vähendada. C++ parseris oli järjenditöötlusega programmi energiakulu kuni 67 protsenti väiksem, kuid C++ parseri algseid külastaja väljakutseid modifitseerides tõusis energiakulu märkimisväärselt – kõige kehvemal juhul kasvas energiakulu 2013 protsenti.

1.3 Java uuendused

Järgnev avatud lähtekoodiga Java arenduskomplekti (Java Development Kit, lühend JDK) OpenJDK uuenduste protsess on avalik ning täismahus kirjeldatud OpenJDK JDK täienduspakkumise (JDK Enhancement Proposal, edaspidi JEP) nr 1 lehel[4].

Suuremate Java muudatuste pärimisel tuleb esmalt koostada ülesehitusele vastav JEP mustand, mis OpenJDK valitud kogukonna liikmetele läbivaatuseks esitatakse. Sobivad uuendused kantakse JEP arhiivi ning avalikustatakse laiale kasutajaskonnale uurimiseks. Toetust kogunud muudatused lisatakse OpenJDK arendusplaani. Aktiivne arendus algab, kui täiustuse sisseviimiseks on piisavalt ressursse. Arendatud moodulid võivad olla mitu aastat saadaval eelvaatena (ingl *preview*), enne kui need lõplikult Javasse lisatakse.

Alates 2019. aasta märtsis avalikustatud JDK 10 versioonist on Javat uuendatud kaks korda aastas: märtsis ja septembris[5]. 2025. aasta mai seisuga on uusim version JDK 24, viimane pika kestustoega (ingl *long-term support*, lühend LTS) versioon on JDK 21. AKT koodibaasi poliitika on toetada viimast LTS versiooni ja mitte kasutada eelvaate faasis olevaid mooduleid.

Järgnevates alapeatükkides kirjeldatakse peamisi JEP uuendusi, mida AKT koodibaasi moderniseerimisel kasutatakse.

1.3.1 *Switch*-avaldised

Switch-laused (edaspidi "lüliti" või autori eestindatud "lülituslause") võtavad argumendiks mingit tüüpi väärtuse ning käivitavad vastavalt argumendi väärtusele kindla programmijupi (Zhang jt, 2021). Käitumiselt saab lülituslauset võrrelda tingimuslausete (ingl *if statement*) jadaga. Lülitid on Javas olnud alates vähemalt Java 2 versioonist (Gosling jt, 2000).

[4] https://openjdk.org/jeps/1
[5] https://openjdk.org/projects/jdk/

JEP 361[6] raames lisati Javasse *switch*-avaldised (edaspidi autori eestindatud "lülitusavaldis"), mis erinevad lülituslausetest noolesüntaksi (ingl *arrow labels*) ja väärtuste tagastamise funktsionaalsuse poolest. JEP 361 lehel illustreeritakse uuendust nädalapäevade näitega (nädalapäevade nimed lühendatud):

```
// Lülituslausega:
int numLetters;
switch (day) {
  case MON:
  case FRI:
  case SUN:
    numLetters = 6;
    break;
  case TUE:
    numLetters = 7;
    break;
  case THU:
  case SAT:
    numLetters = 8;
    break;
  case WED:
    numLetters = 9;
    break;
  default:
    throw new
      IllegalStateException(
        "Wat: " + day);
}
```

```
// Lülitusavaldisega:
int numLetters = switch (day) {
  case MON, FRI, SUN -> 6;
  case TUE -> 7;
  case THU, SAT -> 8;
  case WED -> 9;
};
```

Tänu väärtuse tagastamisele saab lülitit rakendada otse muutuja `numLetters` algatamisele (ingl *initialization*), lülitusavaldises saab juhtusid komadega kombineerida ning pole vaja kirjutada trafaretseid muutujate väärtustamise ridu ja `break` käske. Lisaks pole vaja defineerida vaikejuhu `default` käitumist, kuna lülitusavaldise kasutamisel lisab Java kompilaator ise vaikimisi erindi viskamise.

[6] https://openjdk.org/jeps/361

1.3.2 Mustrisobituse mugavdamine

Mustrisobituseks (ingl *pattern matching*) nimetatakse konstruktsiooni, mis käivitab kindla programmijupi vastavalt argumentide kujule, sh argumentide tüüpidele ja arvule (Hong jt, 2022). Mustrisobitust kasutades võib näiteks rekursiivne järjendi summa leidmise meetod välja näha järgmine (näide programmeerimiskeelest Idris):

```
summa : List Integer -> Integer
summa [] = 0
summa [x] = x
summa (x :: xs) = x + summa xs
```

Ülaltoodud näites tähistab `(x :: xs)` vähemalt üheelemendilist järjendit, mis koosneb esimesest elemendist ehk peast (ingl *head*) `x` ja alamjärjendist `xs`. Meetod `summa` vaatleb rekursiivselt iga argumendina antud järjendi elementi ning tagastab järjendi elementide summa. Mustrisobitus võimaldab juhtumid mugavalt defineerida sõltuvalt järjendi elementide arvust:

1. Kui järjend on tühi, siis tagastatakse arv 0;
2. Kui järjendis on üks element, siis tagastatakse elemendi `x` väärtus;
3. Kui järjendis on vähemalt kaks elementi, siis liidetakse järjendi pea `x` väärtusele rekursiivse väljakutse `summa xs` väärtus.

Mustrisobituse käigus määratakse muutujatele `x` ja `xs` automaatselt väärtused. Tasub täheldada, et ülaltoodud meetodis `summa` on üheelemendilise järjendi juht liigne, kuid üheelemendilise järjendi juhu sisse jätmine aitab paremini illustreerida mustrisobituse võimekust.

JEP 394-ga[7] lisati Java `instanceof` operaatorile mustrisobitusega muutujanime määramise funktsionaalsus. JEP 394 lehel esitatakse järgmine näide:

```
// Enne JEP 394
if (obj instanceof String) {
  String s = (String) obj;
  ...
}
```

```
// Pärast JEP 394
if (obj instanceof String s) {
  ...
}
```

[7] https://openjdk.org/jeps/394

Täiustuse abil saab eemaldada dubleeriva tüübiteisenduse ja määrata muutujale `s` väärtuse otse tingimuslause tingimuses.

Uuendust rakendades saab keele Expr väärtustamise meetodi esitada joonisel 3 toodud kujul.

```
public static int eval(ExprNode expr) {
  if (expr instanceof ExprNum num) {
    return num.getValue();
  } else if (expr instanceof ExprAdd add) {
    return eval(add.getLeft()) + eval(add.getRight());
  } else if (expr instanceof ExprDiv div) {
    return eval(div.getNumerator()) / eval(div.getDenominator());
  } else if (expr instanceof ExprNeg neg) {
    return -eval(neg.getExpr());
  } else {
    // Siia ei peaks kunagi jõudma (v.a. null).
    throw new IllegalArgumentException();
  }
}
```

Joonis 3. Keele Expr väärtustamise meetod tingimuslausetega.

Võrdluseks tasub täheldada, et joonisel 2 kujutatud külastajaga väärtustamise meetodis ei olnud vaikejuhu käitumist defineerida vaja.

1.3.3 *Switch*-mustrisobitus

Pakkumises JEP 441[8] esitatud *switch*-mustrisobitus (eestindatult "lülitiga mustrisobitus") ühildab lülitusavaldised ja mustrisobituse – see võimaldab käivitada kindlat programmijuppi vastavalt objekti tüübile ning vastavalt tüübile määrata sobivaima muutujanime.

Näiteks asendades joonisel 3 kujutatud tingimuslausete jada mustrisobitusega lülitusavaldisega, saab keele Expr väärtustamise funktsiooni esitada joonisel 4 kujutatud koodiga.

[8] https://openjdk.org/jeps/441

```
public static int eval(ExprNode expr) {
  return switch (expr) {
    case ExprNum num -> num.getValue();
    case ExprAdd add -> eval(add.getLeft()) + eval(add.getRight());
    case ExprDiv div ->
      eval(div.getNumerator()) / eval(div.getDenominator());
    case ExprNeg neg -> -eval(neg.getExpr());
    // Siia ei peaks kunagi jõudma (v.a. null).
    default -> throw new IllegalArgumentException();
  };
}
```

Joonis 4. Keele Expr väärtustamise meetod lülitiga mustrisobitusega.

Moderniseeritud meetodis pole vaja kirjutada mitut `instanceof` operaatoriga tingimuslauset ja tagastamine saab toimuda otse lülitil, vähendades `return`-käskude arvu.

1.3.4 Kirjed

Järgnev kokkuvõte pärineb JEP 395 lehelt[9]. Kirjed (ingl *records*) on range ülesehitusega muutmatud andmestruktuurid. Kirje loomisel väärtustatakse argumentidena antud väärtused privaatsetes muutmatutes (`private final`) väljades, tekitatakse ligipääsuks argumentidega samanimelised avalikud meetodid ning genereeritakse objektide võrdlemiseks ja väljastamiseks nõutud `equals(Object)`, `hashCode()` ja `toString()` meetodid.

Dokumentatsioonis esitatakse näide klassist `Point`, mille eesmärk on talletada mingisuguse kahedimensioonilise punkti `x` ja `y` koordinaate:

```
class Point {
  private final int x;
  private final int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
```

[9] https://openjdk.org/jeps/395

```
  }

  int x() { return x; }
  int y() { return y; }

  public boolean equals(Object o) {
    if (!(o instanceof Point)) return false;
    Point other = (Point) o;
    return other.x == x && other.y == y;
  }

  public int hashCode() {
    return Objects.hash(x, y);
  }

  public String toString() {
    return String.format("Point[x=%d, y=%d]", x, y);
  }
}
```

Olgu loodud kaks klassi `Point` isendit `A` ja `B`, mille mõlema koordinaadid on $(1, 2)$, st punktid `A` ja `B` on matemaatilises mõttes võrdsed. Java dokumentatsiooni[10] kohaselt:

1. Kui meetodit `equals(Object)` pole defineeritud, siis isendite `A` ja `B` võrdlemisel on tulemus alati `false`, sest vaikimisi kontrollib Java isendi võrdsust rangelt iseendaga, st isendi ekvivalentsiklass koosneb vaid ühest elemendist.

2. Kui meetodit `hashCode()` pole defineeritud, siis iga `Point` isendi räsi on suvaliselt genereeritud. Teoreetiliselt on see lubatud, kuid kui klassis `Point` on meetod `equals()` defineeritud, siis rikutakse Java eeskirja, et kaks võrdset objekti tagastavad sama räsi.

3. Kui meetodit `toString()` pole defineeritud, siis esitatakse isendi `A` printimisel kuju, mis koosneb klassi `Point` nimest, märgist `@` ja kuueteistkümnendsüsteemis räsist, näiteks `Point@3e2`.

[10] https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html

Eelneva klassi kirjeks teisendades saab `Point` välja näha järgmine:

```
record Point(int x, int y) { }
```

Koordinaatidega $(1, 2)$ kirjete `A` ja `B` korral tagastatakse võrdlemisel `true`, isendid omavad sama räsi ja printimisel väljastatakse `Point[x=1, y=2]`, mh pääseb `A` koordinaatidele ligi käskudega `A.x()` ja `A.y()`. Soovi korral saab meetodite käitumist üle kirjutada.

1.3.5 Kirjemustrid

JEP 440-s[11] soovitatud kirjemustrid (ingl *record patterns*) ühildavad range struktuuriga kirjed ja mustrisobituse: kirjete sobitamisel saab isendiväljade väärtustele ligi pääseda ja otse muutujanimed määrata. Näide dokumentatsioonist:

```
// Enne JEP 440
record Point(int x, int y) {}
static void printSum(Object obj) {
  if (obj instanceof Point p) {
    int x = p.x();
    int y = p.y();
    System.out.println(x+y);
  }
}

// Pärast JEP 440
static void printSum(Object obj) {
  if (obj instanceof Point(int x, int y)) {
    System.out.println(x+y);
  }
}
```

JEP 440 kirjemustrite uuendusega saab kirje `Point` isendi tuvastades koordinaatide `x` ja `y` muutujad tingimuslause tingimuseplokis väärtustada, eemaldades tingimuslause kehas triviaalsed väärtustamise koodiread.

[11] https://openjdk.org/jeps/440

Kombineerides JEP 440 kirjemustrid JEP 441 lülitiga mustrisobitusega, saab trafarettkoodi märkimisväärselt vähendada. Olgu keele Expr alamklassid teisendatud kirjeteks. Joonisel 4 esitatud funktsioon `eval(ExprNode)` võiks selle eelduse korral teisenduda joonisel 5 näidatud kujule.

```
public static int eval(ExprNode expr) {
  return switch (expr) {
    case ExprNum(int value) -> value;
    case ExprAdd(var left, var right) -> eval(left) + eval(right);
    case ExprDiv(var num, var denom) -> eval(num) / eval(denom);
    case ExprNeg(var expr) -> -eval(expr);
    // Siia ei peaks kunagi jõudma (v.a. null).
    default -> throw new IllegalArgumentException();
  };
}
```

Joonis 5. Keele Expr väärtustamise meetod lülitiga mustrisobituse ja kirjemustritega.

Uuendatud meetodis saab tänu lülitiga mustrisobitusele lüliti argumendi tüübi tuvastades otse isendiväljadele muutujanimed määrata, asendades isendiväljade väärtuste väljakutsed.

1.3.6 Sulgklassid ja -liidesed

Liides (ingl *interface*) on abstraktne andmestruktuur, mis koosneb objektiga tehtavate operatsioonide kirjeldustest ja nende kitsendustest[12].

Järgnev kokkuvõte on koostatud JEP 409 lehe[13] põhjal. Sulgklassid (ingl *sealed classes*[14]) ja sulgliidesed (ingl *sealed interfaces*) on vastavalt klassid või liidesed, mida saavad implementeerida ja laiendada vaid vastavas sulgklassis või -liideses loetletud alamklassid. Ligipääsuks peavad alamklassid sulgobjekti laiendamist (`extend`) klassi päises deklareerima. Lisaks peavad alamklassid määrama kas klassi muutumatuse (`final`), alamtüüpide rangeks määramiseks suletuse (`sealed`) või avatuse (`non-sealed`), kui on soov luua abstraktsele

[12] https://akit.cyber.ee/term/3740
[13] https://openjdk.org/jeps/409
[14] https://akit.cyber.ee/term/14509

struktuurile lahtise ülesehitusega alamklasse. Uuenduse dokumentatsioonis on toodud näide geomeetriliste kujundite defineerimisest erinevate muudetavuse ja suletuse tasemetega:

```
// Abstraktne suletud struktuur kujundite defineerimiseks
public abstract sealed class Shape
  // Lubame vaid nelja kujundiklassi
  permits Circle, Rectangle, Square, WeirdShape { ... }

// Muutmatu alamklass
public final class Circle extends Shape { ... }

// Suletud alamklass
public sealed class Rectangle extends Shape
  permits TransparentRectangle, FilledRectangle { ... }

// Teist järku muutmatud alamklassid
public final class TransparentRectangle extends Rectangle { ... }
public final class FilledRectangle extends Rectangle { ... }

// Muutmatu alamklass
public final class Square extends Shape { ... }

// Avatud alamklass
public non-sealed class WeirdShape extends Shape { ... }
```

Liideste puhul on struktuur sarnane eranditega, et liidesed saavad olla ainult abstraktsed – muuhulgas on abstraktsuse deklareerimine liigne – ning muutmatust ei saa abstraktse struktuuri puhul määrata, kuna abstraktsele struktuurile ei saa luua isendit.

Sulgklasse ja -liideseid kasutades saab lülituslausetes ning lülitiga mustrisobitusel tagada programmikoodi tervikluse ilma vaikeklauslita (`default`), kuna kirjeldamata juhtumit ei saa suletuse tõttu esineda.

Sulgliideste kasutuse iseloomustamiseks arendatakse edasi joonisel 5 esitatud keele Expr külastajata väärtustamismeetodit, sh eeldatakse, et keele Expr klassid on kirjeteks teisendatud. Olgu keele Expr ülemklass `ExprNode` teisendatud sulgliideseks, mille puhul lubatakse klassi implementeerida vaid klassidel (täpsemalt kirjetel) `ExprNum`, `ExprAdd`, `ExprDiv` ja `ExprNeg`, st `ExprNode` päis on järgneval kujul:

```
public sealed interface ExprNode
  permits ExprNum, ExprAdd, ExprDiv, ExprNeg
```

Olgu alamklasside päised muudetud liidese implementeerimisele vastavaks. Kõikide nende eelduste kehtimisel implementeeritav lõplik `eval(ExprNode)` meetod on esitatud joonisel 6.

```
public static int eval(ExprNode expr) {
  return switch (expr) {
    case ExprNum(int value) -> value;
    case ExprAdd(var left, var right) -> eval(left) + eval(right);
    case ExprDiv(var num, var denom) -> eval(num) / eval(denom);
    case ExprNeg(var expr) -> -eval(expr);
  };
}
```

Joonis 6. Keele Expr väärtustamise meetod pärast kõikide uuenduste rakendamist.

Sulgliideseid kasutades ei pea lülitis vaikeklauslit defineerima, kuna argumendi `expr` tüüp saab olla vaid üks neljast `ExprNode` päiseses loetletud tüübist.

2. Koodibaasi kirjeldus ja moderniseerimise metoodika

Selles peatükis iseloomustatakse AKT koodibaasi ülesehitust, kirjeldatakse moderniseerimise metoodikat ja teisendatavaid faile. Peatüki lõpus käsitletakse plaani vastavust peatükis 1.1 kirjeldatud refaktoreerimise skeemile.

2.1 AKT koodihoidlate kirjeldus

Aine "Automaadid, keeled ja translaatorid" õppejõududel ja praktikumijuhendajatel on oma privaatne koodivaramu. Privaatse varamu põhjal luuakse igal aastal tudengitele avalik hoidla, tavaliselt GitHubi keskkonnas, ja hoidlas tehakse iga nädal tudengitele uusi materjale nähtavaks. Pärast praktikumi avalikustatakse tudengitele praktikumiülesannete korrektsed lahendused.

Töö autorile anti ligipääs õppejõudude ja praktikumijuhendajate privaatsele koodihoidlale ning järgnev kirjeldus põhineb privaatsel koodihoidlal, kuna see sisaldab kogu kursuse koodimaterjale. Muuhulgas kirjeldatakse vaid töö raames asjakohaseid katalooge. Keelte puhul jäetakse suurte eripärasuste puudumisel süntaksi ja reeglistiku kirjeldamine vahele.

2.1.1 Erinevate nädalate ülesanded

Tudengid puutuvad esimest korda kokku külastajate ja väärtustamisega neljandal nädalal, kui uuritakse keeli Bool, Expr ja Rnd. Kaustas on lisaks näidiskeel Rec, mis on sama reeglistikuga nagu Expr, aga kasutab Java 21 täiustusi.

Viiendal nädalal tutvustatakse järgnevatel nädalatel käsitletavaid keeli Kala ja Aktk: Kala keelt kasutatakse kuuendal, seitsmendal ja üheksandal nädalal teemade tutvustamiseks, Aktk keele tükihaaval implementeerimine jäetakse tudengitele alates viiendast nädalast kodutööks. Keele Kala konstruktsioonid defineeritakse kuuendal nädalal, Aktk konstruktsioonid seitsmendal nädalal. Seitsmendal nädalal kirjutatakse teegi ANTLR[15] abil genereeritud lekserile ja parserile tuginedes keele abstraktse süntaksipuu (ingl *abstract syntax tree*, lühend AST; puukujuline abstraktne esitus programmist) loomise meetod. Keelele Aktk luuakse kaheksandal nädalal interpretaator, üheksandal nädalal tüübikontrollija ja kümnendal nädalal kompilaator.

Kõiki kodutöid saavad tudengid soovi korral esitada ka programmeerimiskeeles Kotlin, seega on koodihoidlas eraldi kataloog Kotlini kodutööde lahenduste jaoks.

[15] https://www.antlr.org/

2.1.2 Kausta toylangs keelte kirjeldus

Neljandal nädalal avatakse tudengitele eksami näidete kaust `toylangs`, mis sisaldab privaatses hoidlas palju erinevaid keeli, millest mõned avalikustades välja jäetakse, kuna neid keeli saab kasutada eksamiülesannetes. Kausta `toylangs` keeled tuginevad ühisele abstraktsele struktuurile `AbstractNode`, mis võimaldab klassi `AbstractNodeVisualizer` ja teegi GRAPHVIZ[16] abil luua pilte abstraktsetest süntaksipuudest.

Edasistes lõikudes kasutatakse `toylangs` keelte kirjeldamisel selgemate näidete huvides fiktiivset keelt nimega Keel, mida saab asendada mis tahes AKT-s käsitletava keele nimega.

Iga keele Keel kaustas on pakett `ast`, mis sisaldab keele süntaksikomponentide ja külastajate `KeelAstVisitor` klasse. Ülesehitus sarnaneb peatükis 1.2 tutvustatud keelele Expr: kaust `expr/ast` sisaldab külastajat `ExprVisitor`, abstraktset ülemklassi `ExprNode` ning tipuklasse `ExprNum`, `ExprAdd`, `ExprNeg` ja `ExprDiv`. Mõne keele puhul võib kaust `ast` sisestada teisi alamkaustu süntaksikomponentide selgemaks eraldamiseks.

Peaaegu kõikidel keeltel on fail `KeelAst`, mille abil saab teisendada lekseemijada keele Keel AST-ks, et seda oleks arvutil mugavam töödelda.

Peaaegu kõikidel keeltel on olemas fail `KeelEvaluator` ehk keele Keel väärtustaja. Tavapärane keele Keel väärtustamise meetod sarnaneb joonisel 2 kujutatud keele Expr külastajaga meetodile.

Enamik keeltel on olemas fail `KeelCompiler` ehk kompilaator, mis tõlgendab soovitud keele arvutile arusaadavamaks keeles[17]. Aine AKT ülesannetes kompileeritakse kood programmeerimiskeelel C põhinevasse assemblerkeelde CMA (Wilhelm ja Seidl, 2010). Koodibaasi testides kontrollitakse CMA programmikirjutaja pinu (ingl *stack*[18]) kattumist oodatud pinuga.

Tavapäraselt antakse tudengitele ette keele kirjeldus ning nende ülesanne on luua ANTLR tööriistale arusaadavad lekseemide reeglid failis `Keel.g4`, genereerida reeglite põhjal ja ANTLR abiga keelele lekser ja parser ning seejärel implementeerida AST loomise meetod, väärtustaja

[16] https://graphviz.org/
[17] https://akit.cyber.ee/term/12272
[18] https://akit.cyber.ee/term/10638

ning kompilaator. Töö korrektsust saavad tudengid kontrollida koodihoidlas asuvate testidega, millest mõned avalikustatakse alles pärast esitamise tähtaja möödumist.

2.2 Moderniseerimise protsess

Moderniseerimise protsess tavapärase AKT keele puhul on üldiselt järgmine:

1. Muuda keele Keel kausta `ast` abstraktne ülemklass sulgliideseks, täpsustades sealjuures failid, mis liidest implementeerida võivad;
2. Teisenda keele tipuklassid kirjeteks;
3. Implementeeri failis `KeelEvaluator` modernne väärtustamismeetod `evalNew`;
4. Implementeeri failis `KeelCompiler` modernne kompileerimismeetod `compileNew`;
5. Eemalda vana meetod `eval` ja nimeta `evalNew` ümber meetodiks `eval`;
6. Eemalda vana meetod `compile` ja nimeta `compileNew` ümber meetodiks `compile`;
7. Eemalda keele `ast` kaustast fail `KeelAstVisitor`;
8. Käivita testid ja veendu programmitöö korrektsuses, küsides vajadusel nõu juhendajalt.

Tasub täheldada, et tööprotsess ei sisalda keelele vastava faili `KeelAst` modifitseerimist, kuna `KeelAst` sisu tugineb suuresti ANTLR-teegile. Failist `KeelAst` külastaja eemaldamine likvideeriks 2025. aasta alguse seisuga tööriista ANTLR täielikkuse kontrolli. Vastava funktsionaalsuse soov on ANTLR loojale edastatud[19], kuid 2024. aasta märtsi GitHub postituse[20] põhjal on ebatõenäoline, et ANTLR selle võimekuse saab.

2.2.1 Kausta toylangs moderniseerimise metoodika

Kausta `toylangs` moderniseerimiseks pandi paika järgnev plaan:

1. Loo liides `AbstractNode2`, mis sisaldab algul samu meetodeid, mis olemasolev klass `AbstractNode`;
2. Loo soovitud `toString()` kuju jaoks klass `NodeUtils` ja ülekirjutatav meetod `nodeToString(AbstractNode2)`;

[19] https://github.com/antlr/antlr4/issues/3282, https://github.com/antlr/antlr4/issues/4310
[20] https://github.com/antlr/antlr4/pull/4419

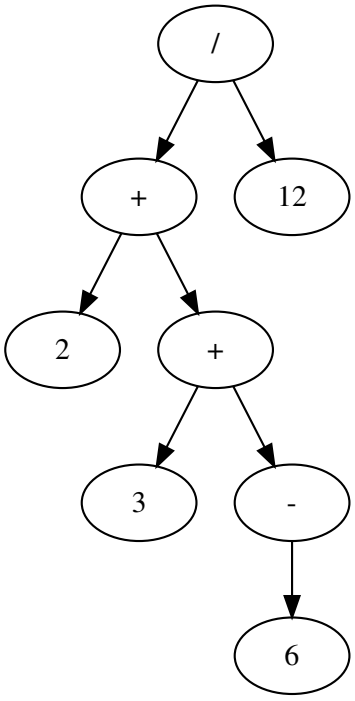
3. Loo `AbstractNode2` meetodid klassi `AbstractNodeVisualizer`, et oleks võimalik luua illustreerivaid pilte liidest `AbstractNode2` implementeerivatest keeltest;
4. Implementeeri ükshaaval kõik `toylangs` keeled vastavalt peatükis 2.2 kirjeldatud skeemile, hoides sealjuures juhendajat toimuvaga kursis;
5. Viimistle klasse ja kui võimalik, siis eemalda liigseid meetodeid;
6. Eemalda vana `AbstractNode` klassi kasutavad meetodid kõikidest failidest;
7. Eemalda vana `AbstractNode` ja nimeta klass `AbstractNode2` ümber `AbstractNode`-iks.

Kausta `toylangs` keeled jagatakse mõtteliselt kaheks sõltuvalt sellest, millal neid viimati õppetöös kasutati – esmalt moderniseeritakse enimkasutatud keeled, seejärel keeled, mida näidetes või eksamitel pole pikemat aega kasutatud. Mõtteline kaheks jaotamine võimaldab moderniseerimise vältel tööprotsessi paremini hinnata.

Kuna `toylangs` keeli kasutatakse aktiivselt ka õppetöös, siis ei mainita enamik keelte nimesid, kirjeldusi või olemust.

2.2.2 AbstractNodeVisualizer teisendamine

Kausta `toylangs` klass `AbstractNodeVisualizer` loob teegi GRAPHVIZ abil hõlpsasti väljastatava graafi keele süntaksikomponentidest ja seostest nende vahel. Visualiseerija soovitud väljundgraaf on sarnane joonisel 7 kujutatud keele Rec avaldise $(2 + (3 + (-6)))/12$ ja visualiseerija `RecVisualizer` väljundi näitele. Klassis `AbstractNodeVisualizer` tugineb graafi loomine meetodile `visit(AbstractNode)`, mis omakorda tugineb abstraktse klassi `AbstractNode` meetodile `getAbstractNodeList()`. Meetod `getAbstractNodeList()` tagastab väljakutsutud `AbstractNode` isendi (keele tipu) alamtipud. Iga keele kausta `ast` iga klass peab seda meetodit implementeerima, et väljund sisaldaks soovitud tippe. Lisaks on abstraktses klassis `AbstractNode` loodud meetodid `getNodeLabel()`,



Joonis 7. RecVisualizer näiteväljund.

`getNodeInfo()`, `listNode(List, String)`, `concat(List, List)`, `cons(AbstractNode, List)` ja `snoc(List, AbstractNode)`, et tippude teksti ja tagastatavaid liste saaks üle kirjutada. Lahendus on verboosne ning enamik `ast` klassidest peavad meetodid sobival kujul üle kirjutama.

Keerulisuse vähendamiseks seati ülesandeks klassi `AbstractNodeVisualizer` külastajalt teisendamine ja `AbstractNode` liigsete meetodite eemaldamine. Selleks pandi paika soovitud lõpptulemid:

1. Liidesesse `AbstractNode` jäävad alles vaid meetodid `renderPngFile(Path)` graafiväljundi mugavaks genereerimiseks ja `getNodeInfo()` isendi mugavdatud nime väljastamiseks;
2. Liidesesse `AbstractNode` luuakse üldistatud tippude läbimise meetod `getChildNodes()`, mida ei pea ümber kirjutama igas implementeerivas klassis, vaid ainult erandjuhtudel.

Klassi `AbstractNodeVisualizer` teisendamine on planeeritud tööprotsessi kõige keerulisem osa.

2.3 Moderniseerimine Mens ja Tourwé skeemi järgi

Moderniseerimisel järgitakse suuresti Mens ja Tourwé (2004) refaktoreerimise skeemi:

1. **Struktuurseid muudatusi vajavate programmilõikude tuvastamine**.

   Suuremad programmilõigud, mis struktuurseid muudatusi vajavad, on neljanda nädala keeled, kausta `toylangs` keeled koos abstraktse ülemklassi ja visualiseerijaga ning hilisematele õppenädalatele laiali jaotatud keeled Kala ja Aktk.

2. **Rakendatavate faktoreerimisvõtete valimine**.

   Sobivates kohtades asendatakse klassid kirjetega, abstraktsed klassid sulgklasside ja sulgliidestega ning tehakse lülitites vastavaid muudatusi. Kogu koodibaasis soovitakse eelnevate sammude ja meetodite struktuurimuudatustega vabaneda külastaja disainimustri kasutamisest.

   Fowler (2018) kirjeldatud faktoreerimisvõtetest kasutatakse peamiselt järgnevaid:

   (a) Funktsiooni päise muutmine;

   (b) Meetodite eraldamine;

(c) Muutujate eraldamine;

(d) “Surnud koodi” eemaldamine.

3. **Veendumine, et struktuurimuudatused ei mõjuta programmi töökäiku**.

   Veendumiseks kasutatakse arvukalt olemasolevaid teste ja keerulisemate murekohtade lahendamisel palutakse juhendajal varakult veenduda töö korrektsuses.

4. **Programmikoodi muutmine**.

   Programmikoodi muutmisel üritatakse uuendusi sisse kanda ühe keele haaval, et ülejäänud projekt ootuspäraselt toimiks. Enne muudatuste koodihoidlasse sisse kandmist veendutakse, et testid lähevad läbi ja programmid käituvad korrektselt.

5. **Muudatuste mõju (loetavus, arusaadavus jms) ning kulu (aeg, jõudlus jms) hindamine**.

   Muudatuste subjektiivset mõju hindavad AKT praktikumijuhendajad, kuna muudatusi ei jõuta sisse kanda enne kevadsemestri algust. Muudatuste ajakulu, mõju jõudlusele vms selles töös ei arvestata. Muudatuste edukust saab selles töös kvantitatiivselt hinnata vaid eemaldatud, muudetud ja lisatud failide ja ridade arvuga.

6. **Ühtluse tagamine muudetud koodi ja teiste programmiga seotud materjalide (dokumentatsioon, nõuded, testid jms) vahel**.

   Dokumentatsiooni ja õppematerjalidega ühtlust selle töö raames ei tagata.

3. Koodibaasi moderniseerimine

Selles peatükis kirjeldatakse koodibaasi moderniseerimise protsessi, moderniseerimise käigus tekkinud probleeme ja erisusi ning kuidas neid lahendati.

3.1 Neljanda nädala näitekeeled

Neljanda nädala näitekeeled Bool, Expr ja Rnd on võrdlemisi lihtsa struktuuriga ning ei sõltu eriti teistest projektis asuvatest failidest. Lisaks on neljanda nädala keelte seas keel Rec, mis on identne keelega Expr, kuid mis on juba tudengitele näite loomise huvides moderniseeritud. Keel Rec ei tugine üheski meetodis külastajale, vaid operatsioonid tehakse otse süntaksipuu tippudel.

3.1.1 Keeled Expr ja Rec

Keelt Expr tutvustatakse süvitsi peatükis 1.2. Tudengite praktikumiülesanne on luua sellele keelele väärtustaja klassis `VisitorIntro`. Tärnülesannetena saavad tudengid harjutada teiste puustruktuuri ülesannete lahendamist failis `ExprMaster` meetodeid implementeerides.

Keele Expr klassi `VisitorIntro` väärtustamise meetod ilma külastajata on kujutatud joonisel 3. Keele Expr moderniseeritud teisiku Rec väärtustamise meetod on identne joonisel 6 kujutatud meetodiga, kui joonisel iga suure algustähega sõne `Expr` asendada sõnega `Rec`. Olulisemad erinevused nende väärtustajate vahel on järgmised:

1. Kõik juhud saab tänu lülitusavaldisele tuua ühe `return` käsu alla;
2. Lülitusavaldise abil saab `if`-ahela asendada `case`-idega vastavalt tipu tüübile, mis nõuab vähem tähemärke, kui `instanceof` kasutamine;
3. Tänu kirjete kasutamisele saab mustrisobitusega otse juhu tuvastades muutujad väärtustada, ilma et peaks välja kutsuma näiteks isendil `num` meetodit `value()` või `getValue()`;
4. Liides `RecNode` on suletud, seega pole vaja lisada `else`- või vaikejuhtu.

Keel Rec oli juba moderniseeritud, kuid polnud loodud vastavat tärnülesannete faili `RecMaster` ja visualiseerimise faili `RecVisualizer`, mille abil saab süntaksipuud näiteks pildi- või PDF-failina väljastada. Klassi `RecMaster` ning vastavate testide implementeerimine oli võrdlemisi triviaalne tänu IntelliJ IDEA koodiredaktori võimekusele, kuid `ExprVisualizer` põhjal koostatud `RecVisualizer` nõudis kogenematuse ja `Visitor`-klassi kaotamise tõttu rohkem tööd. Lõpptulemuses kasutatakse tippude läbimiseks rekursiivset meetodit `goThroughNodes(RecNode)`, milles luuakse iga tipu juures teegi GRAPHVIZ isend ning

seejärel ühendatakse isend rekursiivse väljakutse tulemusega. Lehttippu jõudes ühendamist ei toimu ning rekursioon sügavamale ei lähe.

Näiteks avaldis $(2 + (3 + (-6)))/12$ on `RecVisualizer` abil esitatav joonisel 7 kujutatud viiekihilise graafina, kus tippudes on kas aritmeetilised operatsioonid või arvulised väärtused.

Keel Expr on üks vähestest keeltest, kus töö tulemusel külastaja disainimuster alles jääb, et tudengid saaksid võrrelda erinevate disainimustrite kasutust sama ülesande lahendamiseks.

3.1.2 Keeled Bool ja Rnd

Keel Bool on loogikaavaldiste keel, mis koosneb operatsioonidest $\wedge$, $\vee$, $\Rightarrow$, $\neg$ ja lehtedes asuvatest muutujatest, mille väärtus võib olla kas "tõene" või "väär". Tõeste muutujate hulk antakse faili `BoolEvaluator` meetodile `eval(BoolNode, Set)` argumendina ning muutujatipuni jõudes kontrollitakse, kas muutujanimi sisaldub selles hulgas. Tudengite ülesanne on implementeerida väärtustaja `BoolEvaluator` ning soovi korral lahendada lisaülesanded failis `BoolMaster`.

Keel Bool oli esimene keel, milles oli vaja kõik failid teisendada. Keele AST abstraktne klass `BoolNode` muudeti sulgliideseks, klass `BoolVisitor` eemaldati ning klassid `BoolImp`, `BoolNot`, `BoolNot`, `BoolVar` teisendati kirjeteks. Ainus kirje, mis vajas lisatööd, oli `BoolVar`, milles oli vaja vaikimisi `toString()` meetod üle kirjutada.

Keele Bool väärtustaja teisendamiseks tuli eemaldada `BoolVisitor` isendi loomine, luua mustrisobitusega lüliti ning rekursiivsed `visit(BoolNode)` väljakutsed asendada `eval(BoolNode, Set)` väljakutsetega.

Klassis `BoolMaster` asuvad raskemad puu struktuuri harjutused, näiteks avaldises implikatsiooni teisendamine teistele operaatoritele ja abstraktsest süntaksipuust otsustuspuu (ingl *decision tree*) tegemine. Meetodid teisendati edukalt.

Keelt Bool moderniseerides oli vaja teisendada ka keeles Kotlin kirjutatud fail `BoolEvaluatorKt`, kuna olemasolevad Kotlini lahendused tuginevad keele Java kausta `ast` failidele ning `BoolVisitor` eemaldamine mõjutas seetõttu ka Kotlini programmide korrektsust.

Keele Rnd moderniseerimine vastas peatükis 2.2 kirjeldatud keele teisendamise protsessile ja oli suuresti identne keele Bool moderniseerimisele.

3.2 Kausta toylangs keeled

Kausta `toylangs` keelte struktuur ja ka abstraktsete klasside jaotus varieerub ulatuslikult: mõnel keelel on üks suur abstraktne ülemklass, mõni jaotub kolmeks, sealjuures on ülemklassidel omakorda abstraktsed alamklassid, mida keeled laiendavad. Üks tagajärgedest on see, et mõne keele puhul pole sulgliidese kasutamine võimalik, kuna sulgliideseid ei saa implementeerida klassid väljaspool seda kausta, kus sulgliides ise defineeritud on[21] (veateade ingl *Class is not allowed to extend sealed class from another package*). Selguse huvides on aga kasulik eraldada mõne keele puhul failid alamkaustadesse. Seepärast jäeti mõne keele ülemliides mittesuletuks ning seetõttu on mustrisobituses vaja lisada ka vaikejuht `default`, kuhu programm ei tohiks tavapärase töö käigus jõuda.

3.2.1 Väärtustajate teisendamine

Väärtustajate teisendamine vastas suuresti planeeritud tööprotsessile – kõik väärtustajad teisendati edukalt lülititele.

Mitme alamliidese puhul loodi mitu eraldi väärtustamismeetodit ning sõltuvalt tippu implementeeritavast liidesest kutsutakse välja kas üks või teine väärtustamismeetod. See tuleb ette näiteks programmides, kus on vaheldumisi avaldised (ingl *expression*) ja laused (ingl *statement*). Väärtustajaid tuleb vastavalt argumentidele mõnikord vaheldumisi välja kutsuda, mis võib tunduda küll esmapilgul segane, kuid eraldab koodi lihtsasti hallatavateks osadeks. Lisaks võimaldab mitme väärtustamismeetodi kasutamine defineerida erinevaid tagastustüüpe avaldiste ja lausete jaoks, mida pole monoliitse külastaja puhul võimalik teha.

Eelistatud lahendus mittesuletud ülemklassi korral on eelnevas lõigus kirjeldatud mitme meetodi loomine ja vastavalt alamtipu tüübile välja kutsumine, kuid vähemalt ühe keele puhul polnud see võimalik, kuna selle keele spetsifikatsiooni kohaselt võivad alamtipud olla erinevat tüüpi – näiteks liidest `KeelLause` implementeeriv programm `KeelProgramm` võib koosneda ainult ühest väärtusest, mis kuulub liidese `KeelVäärtus` alla, või mitmest definitsioonist ja tingimuslausest, mis kuuluvad liidese `KeelLause` alla ja sisaldavad omakorda vaheldumisi alamtippudena `KeelLause` ja `KeelVäärtus` isendeid. Selle keele puhul jäeti alles vaid üks üldine väärtustamismeetod, mille lülitis oli mittesuletuse tõttu vaja lisada vaikejuht `default`.

[21] https://docs.oracle.com/en/java/javase/21/language/sealed-classes-and-interfaces.html

3.2.2 Kompilaatorite teisendamine

Kompilaatoriklasse teisendades kasutati algul peamiselt staatilist abimeetodit `compileHelper`, millele lisati argumentidega kaasa kõik isendid, mida programmi toimimiseks ja tulemuse tagastamiseks vaja oli. See aga polnud meetodite ja muutujate staatilisuse suhtes alati korrektne, kuna staatilised meetodid ei ole seotud ühtegi kindla isendiga[22]. Teisisõnu ei modifitseeritud staatilise meetodi rekursiivsetel väljakutsetel argumendina antud isendeid, nagu oli soovitud.

Töö mõttelise esimese poole lõpus otsustati kompilaatoriklasside sisu standardiseerida järgnevalt:

1. Iga kompilaatorifaili `KeelCompiler` staatilises meetodis `compile` luuakse isend `keelCompiler`, millel on väljad assemblerkeele programmikirjutaja `pw` ning muude mittestaatiliste isendite, näiteks muutujate järjendi jaoks;
2. Mittestaatiline abimeetod `compileNode` või teistsuguse päisega `compile` kutsutakse seejärel välja sellel isendil, näiteks `keelCompiler.compileNode(KeelNode);`
3. Programmikirjutaja sisu loetakse pärast tippude läbimist isendiväljast, näiteks käsuga `keelCompiler.pw.toProgram()`.

Näiteks `toylangs` keele Imp korral näeb meetod `compile` välja järgmine:

```
public static CMaProgram compile(ImpProg prog) {
  CMaProgramWriter pw = new CMaProgramWriter();
  ImpCompiler impCompiler = new ImpCompiler(
    new CMaProgramWriter(), new ArrayList<>()
  );
  impCompiler.compileNode(prog);
  return impCompiler.pw.toProgram();
}
```

Uuendatud kujul on keele Imp puhul tudengite ülesanne implementeerida meetod `compileNode(ImpProg)`.

Hiljem otsustati sarnast protsessi rakendada `toylangs` keelte väärtustajatele, st peameetod `eval` failis `KeelEvaluator` loob soovitud väljadega isendi `keelEvaluator`, millel

[22] https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.3.2

kutsutakse välja mittestaatiline meetod `evalNode(node)` ja peameetod tagastab `keelEvaluator.evalNode(KeelNode)` väärtuse.

3.2.3 AbstractNodeVisualizer teisendamine

Visualiseerija `AbstractNodeVisualizer` teisendamiseks uuriti koos juhendajaga Java klasside `Class` ja `Record` dokumentatsiooni ning tuvastati Java programmide peegeldamise (ingl *reflection*) funktsionaalsus, mis võimaldab isendil ligi pääseda isendi enda meetoditele, konstruktoritele ja isendiväljadele[23]. Loodud `getChildNodes()` meetodis, millele visualiseerija tugineb, kasutati peegeldamist, et üldistada meetodi käitumine erinevat tüüpi isendiväljade puhul.

Visualiseerija meetodile `getNodeInfo()` lisati vaikimisi töökäik, mis enamik klasside puhul võimaldab visualiseerija tipule soovitud info väljastada. Kui vaikimisi `getNodeInfo()` tagastatav väärtus tipule lisatava sõne kohta ei sobi või on ebakorrektne, saab meetodi tipuklassis hõlpsasti üle kirjutada. Tööprotsessi lõpuks oli tipuinfo meetod vaja üle kirjutada 19 failis.

Valminud meetodi `getChildNodes()` teadaolevad kitsaskohad on järgmised:

1. `getChildNodes()` oskab vaikimisi läbida vaid kirjeid. Teoreetiliselt on klasside vaikimisi käsitlemise võimekus implementeeritav, kuid kuna kõik kausta `toylangs` keelte `ast` failid teisendusid kirjeteks, otsustati klasside töötlemist mitte lisada.
2. `getChildNodes()` ei suuda tõenäoliselt läbida mõnd `Collections` isendit ja võib-olla `Object[]` massiive. Soovitud kujus on ulatuslikult veendutud vaid `List` ja `Set` klassidega.
3. `getChildNodes()` ei oska vaikimisi käsitleda kirjet, mis kasutab mõnd alamobjekti, näiteks `enum`. Selliseid juhte esineb harva ning siis on lihtsaim lahendus meetodi `getChildNodes()` käsitsi üle kirjutamine korrektsete tippude läbimiseks ning `getNodeInfo()` üle kirjutamine tipule soovitud teksti lisamiseks.
4. `getChildNodes()` tagastatav järjend sõltub kirje isendiväljade järjekorrast: kui väljundi elementide järjekord ei vasta soovitud kujule, saab järjekorra käsitsi üle kirjutada.

Loodud meetod `getChildNodes()` on täismahus loetav töö lisas A.

[23] https://www.oracle.com/technical-resources/articles/java/javareflection.html

3.2.4 Meetodile toString() Velocity malli loomine

Kausta `toylangs` keelte puhul kasutati algselt keelte sisu sõneliseks väljastamiseks (printimiseks) erinevaid `AbstractNode` meetodeid, mis polnud kuigi hästi dokumenteeritud: `getAbstractNodeList()`, `getNodeInfo()`, `getNodeLabel()`, `buildString(StringBuilder)`, `canHaveEmptyChildList()`, erinevate argumentidega meetod `dataNode()` ning ülekirjutatud `toString()`. Kuna visualiseerija teisendamisega vabaneti meetodist `getAbstractNodeList()`, siis töö täielikkuse huvides otsustati leida ka alternatiiv keerulisele ja verboossele `toString()` koostamisele. Juhendajaga koostöös otsustati asendusena kasutada IntelliJ IDEA võimekust luua Apache Velocity mallikeeles käsitsi mall soovitud `toString()` meetodi genereerimiseks.

Järgnev kokkuvõte on koostatud Apache Velocity kasutusjuhendi põhjal (*Apache*, 2025). Apache Velocity Template Language (VTL) on Java-põhine mallikeel, mis koosneb peamiselt direktiivide (Apache käsud) ja viitade (muutujad, isendiväljad, meetodid) kombineerimisest väljastatavate tekstilõikudega. Direktiividel kasutatakse prefiksit `#`, viitadel prefiksit `$`. Näiteks VTL keeles HTML-mall, mis sisaldab nimemuutujat ja väljastatavat teksti, võib välja näha järgmine:

```
<html>
  <body>
  #set( $foo = "Velocity" )
  Hello $foo World!
  </body>
</html>
```

Selles näites määratakse direktiiviga `#set` viite `$foo` väärtuseks sõne `Velocity`, mis seejärel lisatakse fraasi `Hello World!` keskele. Malli põhjal genereeritud kood on seega järgmine:

```
<html>
  <body>
  Hello Velocity World!
  </body>
</html>
```

Tasub täheldada, et mallis pole vaja eraldi välja kirjutada reavahetusi või tühemikke – need võetakse vaikimisi arvesse. Keeles on defineeritud tingimuslaused, kommentaarid, tsüklid, failide

lugemine jpt funktsioonid. Meetodite, viidete alamliigi abil saab malli lisada ka Java koodi, mida ootuspäraselt kompileeritakse.

Koodiredaktor IntelliJ IDEA võimaldab kasutajal VTL malli põhjal mugavalt genereerida erinevaid koodijuppe, sealhulgas ka `toString()` meetodit[24]. Malli loomisel saab kasutada erinevaid IntelliJ loodud viitasid, nagu `$classname` täispika klassinime saamiseks, `$fields` isendiväljade väärtuste järjendina tagastamiseks ja `$member.primitive` kontrollimaks, kas tegemist on primitiivse Java muutujaga.

Koodihoidlale VTL-is `toString()` malli koostamiseks oli esmalt vaja kaardistada kõik käsitletavad failitüübid, analüüsida VTL meetodeid ning luua skeem tüüpide kontrollimiseks, kuna malliga sooviti katta võimalikult palju erinevaid juhte ning piiratud viitade arvu tõttu oli mall vaja koostada korrektses järjekorras, et üks tingimuslause ekslikult ei kehtiks ekslikult teise, soovimatu isendivälja tüübi puhul.

Täispikkuses `toString()` mall ja ühe eemaldatud lõiguga `toString()` malli dokumentatsioon on loetavad vastavalt töö lisades B ja C.

3.3 Keeled Kala ja Aktk

Keele Kala teisendamise protsess ei vastanud peatükis 2.2 kirjeldatud keele teisendamise protsessile, kuid kogu keele Kala `ast` klassid on paigutatud ühte faili ning sisse kantavad muudatused vastasid üsna otseselt algselt planeeritule: `ast` klassid teisendati kirjeteks ja liidesteks, eemaldati liigsed `equals(KalaNode)` meetodid ning muudeti isendiväljade väljakutsed, et need vastaksid kirjete isendiväljade ligipääsule. Kompilaator teisendus hõlpsasti külastajalt lülitile.

Keel Aktk on ainus selle töö raames käsitletud keel, milles on võimalik muutujate ja kausta `ast` klasside sisu *setter*-meetoditega üle kirjutada. Koostöös juhendajaga otsustati *setter*-meetoditega klassid jätta klassideks ning kõik teised klassid teisendada kirjeteks, et keele alustalasid ei peaks drastiliselt ümber tegema.

Keele Aktk klass `AstNode` oli võimalik teisendada sulgliideseks, kuna kõik Aktk keelekomponendid asuvad ühes kaustas. Tänu sellele saab implementeerivates klassides eemaldada vaikejuhu kirjeldamise.

[24] https://www.jetbrains.com/help/idea/generate-tostring-settings-dialog.html

Faili `AstNode` abstraktset alamklassi `Literal` kasutati vaid kahes kirjeks teisenduvas failis `IntegerLiteral` ja `StringLiteral`. Pärast juhendajaga konsulteerimist otsustati abstraktne klass `Literal` eemaldada ning tuua kirjeks teisenduvad klassid alamliidese `Expression` alla. Protsessi lõpuks jäi `ast` kausta ülemliides `AstNode`, alamliidesed `Expression`, `Statement` ja `VariableBinding`, viis klassi ning kaheksa kirjet.

Klasside `AktkInterpreter`, `AktkBinding`, `AktkTypeChecker` ja `AktkCompiler` teisendamine vastas üsna otseselt tavapärasele tööprotsessile: lõplikes meetodites loob iga klass isendi, mille peal kutsutakse välja lülitiga töötlemismeetod. Nende klasside puhul oli vaja teisendada ka Kotlini lahendustefailid, kuna Kotlinis kirjutatud lahendustes kasutatakse Java `ast` klasse. Ümber kirjutamine hõlmas enamasti `AstVisitor` laiendamise eemaldamist, isendiväljadega klassi loomist ning `visit(AstNode)` meetodite lülititega asendamist.

Keele Aktk visualiseerija `AstVisualizer` meetodid teisendusid ilusti lülititele – vastavad meetodid nimetati `visit(Object)` ja `visit(AstNode)` pealt ümber meetoditeks `goThroughNodes(Object)` ja `goThoughAstNode(AstNode)`.

Keele Aktk `ast` kausta jäeti alles `AstVisitor`, kuna mõned varasemad lahendusefailid tuginevad sellele ning nende teisendamine hõlmaks olemasoleva töö dubleerimist. Nende lahendusefailide olemasolu, korrektsus ja vajalikkus on vaja kooskõlastada praktikumijuhendajate ja õppejõududega ning selle eest vastutab töö juhendaja.

3.4 Ühtlustamine ja dokumenteerimine

Moderniseeritavate klasside kaardistamise käigus leiti näiteks kolmanda ja seitsmenda nädala kaustades klasse, mida sai mugavalt trafarettkoodi vähendamiseks kirjeks teisendada.

Võimalikult palju väärtustaja- ja kompilaatoriklassidest teisendati kirjeteks, kuna enamik nende klasside isendiväljadest püsivad programmi töö vältel Java mõistes lõplikud, st isendivälja ei saa programmi käigus üle kirjutada.

Meetodite nimed standardiseeriti ja seati paremini vastavusse tehtava ülesandega: klassi `KeelEvaluator` peamiste meetodite nimedeks valiti `eval(Node)`, vajadusel kasutati `evalNode(Node)`. Erinevate tagastustüüpide puhul nimetati meetodid näiteks `evalExpr(Expr)`, `evalStmt(Stmt)`, erinevates kaustades defineeritud liideste puhul näiteks `evalExpr(Expr)`, `evalStmt(Stmt)`, `evalBinaryOperator(BinaryOperator)`. Sama protsessi viidi läbi ka klassides `KeelCompiler`, `KeelInterpreter`, `KeelBinding`, `KeelTypeChecker` jpt.

Kompilaatorite `KeelCompiler` konstruktorite päistes standardiseeriti programmikirjutaja `pw` asetus: `pw` on alati esimene argument.

Kausta `toylangs` ülemliides `AbstractNode` nimetati ümber `NodeInterface`-iks ning varasem `AbstractNodeVisualizer` nimetati ümber `NodeVisualizer`-iks.

Töö käigus valminud uued meetodid ja klassid, nagu `StringNode` ja `NodeInterface.getChildNodes()`, dokumenteeriti detailselt, et kood oleks mõistetav ning vajadusel mugavalt modifitseeritav.

Meetodite ligipääsu vähendati vähima võimaliku tasemeni: enamasti jäeti klass ise ning mõni põhimeetod testimise huvides avalikuks, kuid alam-meetodite ligipääsutase vähendati `private` või mõnel haruldasel juhul `protected` peale.

Juhendaja loal ja soovitusel eemaldati vanu kommentaare ning modifitseeriti olemasolevaid kommentaare, et need oleksid vastavuses muudetud koodiga.

4. Töö tulemused

Selles peatükis kirjeldatakse tehtud töö mõõdetavaid näitajaid ning aine AKT praktikumijuhendajatelt kogutud tagasisidet modifitseeritud koodi kohta.

4.1 Kvantitatiivsed näitajad

Andmed tabelites 1a ja 1b on esitatud 6. mai 2025 seisuga AKT privaatse koodihoidla tõmbekutse (ingl *pull request*[25]) põhjal. Tasub täheldada, et muudetud failide ja koodiridade arv üksi ei iseloomusta mitte midagi koodi kvaliteedi, sisu, olulisuse või töö keerukuse kohta.

Tabel 1. Koodihoidla tõmbekutse põhjal lisatud, eemaldatud ja muudetud failide ja ridade arv.

(a) Tõmbekutse kohaselt muudetud failide arv.

FAILIDE ARV	
Kokku	353
“Lisatud”	18
“Eemaldatud”	33
“Muudetud”	302
...millest ümber nimetatud	1

(b) Tõmbekutse kohaselt muudetud ridade arv.

RIDADE ARV	
Lisatud	5373
...millest “lisatud” failides	830
...millest “muudetud” failides	4543
Eemaldatud	9466
...millest “eemaldatud” failides	984
...millest “muudetud” failides	8482

Tõmbekutse kohaselt on kokku modifitseeritud (lisatud, eemaldatud, sisuliselt muudetud, ümber nimetatud) 353 faili. Failide liigutamisel koodihoidlas loetakse faile “eemaldatuks” ja “lisatuks”: teisaldamisi sisse arvestades on “loodud” 18 uut faili ning “kustutatud” 33 faili, vahe tuleneb peamiselt kustutatud `KeelAstVisitor` failidest. Ümber nimetati vaid üks fail, kuid kuna ümber nimetamisega muudeti failis kaks rida, siis loetakse seda siinkohal faili “muutmiseks”. Eelmainitud märkust arvestades “muudeti” 302 faili.

Repositooriumis tehtud muudatustega on lisatud 5373 rida ning eemaldatud 9466 rida. See arv sisaldab muuhulgas ka `toString()` malli ja selle dokumentatsiooni, ulatuslikku `NodeVisualizer` ja `NodeInterface` dokumentatsiooni, töö käigus loodud failide

[25] https://akit.cyber.ee/term/16710

`StringNode`, `RecMaster`, `RecVisualizer` ja `RecMasterTest` koodiridu. Tasub täheldada, et failide koodihoidlas ümber tõstmisel loetakse ridu samuti "eemaldatuks" ja "lisatuks" – mitme faili puhul seda ka tehti.

Töö raames "lisatud" failides oli kokku 830 rida, töö raames "eemaldatud" failides kokku 984 rida. See tähendab, et "muudetud" failides lisati 4543 ning eemaldati 8482 rida.

Näitajate ja eelmainitu põhjal saab väita järgmist:

1. Moderniseerimise tulemusel oli koodihoidlas $33 - 18 = 15$ faili vähem.
2. Tööprotsessi järgselt sisaldas koodihoidla $9466 - 5373 = 4093$ rida vähem sisu, hoolimata lisatud failidest ja dokumentatsioonist.
3. Tööprotsessi alguses koodihoidlas asuvates failides, mida ei tõstetud töö käigus ringi, vähendati sisu $8482 - 4543 = 3939$ rea võrra.

Eeldusel, et töö alguses koodihoidlas asuv sisu ning töö käigus kirjutatud kood ja dokumentatsioon järgivad suuresti vormistamise häid tavasid, ning eeldusel, et koodihoidla sisaldas tööd alustades minimaalselt liigset sisu, saab nendele väidetele tuginedes järeldada, et töö eesmärk trafarettkoodi vähendada ja koodi lühemaks muuta saavutati edukalt.

4.2 Tagasiside

Tagasiside on kogutud privaatse koodihoidla tõmbekutsele lisatud kommentaaride kaudu, kuna ajanappuse tõttu polnud võimalik struktuursemalt tagasisidet koguda. Muuhulgas jõudis muudatusi kommenteerida vaid üks praktikumijuhendaja.

Üldpildis on praktikumijuhendaja tagasiside positiivse meelestatusega, kuid konstruktiivne.

Peatükis 3.2 mainiti, et mõne keele puhul jäeti ülemliides mittesuletuks. Tagasisidet andnud praktikumijuhendaja ei nõustunud mõttega, et kausta `ast` klassid võiksid olla jaotatud omakorda alamkaustadesse, kui seetõttu liidese suletust defineerida ei saa – lüliti kasutamise mõte võib tema arvates ilma selle muutuseta ära kaduda. Samuti oskab IntelliJ IDEA sulgklassi puhul ise implementeerivad klassid tuvastada, mis oleks programmeerides kasulik.

Praktikumijuhendaja soovitas, et mõnd muutuja- ja klassinime võiks kohendada, kuna koodiread on pikkade muutujanimede tõttu üle saja sümboli pikad. Mõnel juhul võib tema sõnul aidata ka IntelliJ IDEA automaatse vormistamise funktsionaalsus.

Mõne keeles Kotlin muudetud faili kohta kommenteeris praktikumijuhendaja, et ".. muudatused on tehtud eeldatavasti üsna põlve otsast, sest Kotlin pole piisavalt käpas", ning lisas tõmbekutsele ulatuslikult parandusi ja vormingusoovitusi. Kotlini väärtustamise meetodite `eval` puhul mainis praktikumijuhendaja ideed meetodid hoopis `extension`-iteks[26] muuta.

Peatükis 3.4 kirjeldatud väärtustaja- ja kompilaatoriklasside kirjeteks teisendamisega praktikumijuhendaja ei nõustunud, kuna tema silmis tasub kirje andmestruktuuri kasutada eelkõige andmeid talletavate klasside puhul. Töö autori jaoks illustreerib see kommentaar ilmekalt, et üht ülesannet võib lahendada mitmel moel ja muudatuste sisse kandmisel tuleb leida kompromiss mõjutatavate osapoolte vahel.

Praktikumijuhendaja esitas kommentaarides alternatiivse liidese `NodeInterface` meetodi `getChildNodes()` kuju, millega saaks vähendada tingimuslausete pesastamist.

Tagasisides soovitatud koodimuudatusi ei kantud selle töö raames sisse.

[26] https://kotlinlang.org/docs/extensions.html

Kokkuvõte

Bakalaureusetöö eesmärk oli uuendada Tartu Ülikooli aine "Automaadid, keeled ja translaatorid" Java koodibaas, sealjuures asendada trafaretne *Visitor*-i disainimuster modernsemate lahendustega ja muuta koodi selgemaks.

Töös anti ülevaade refaktoreerimisest, *Visitor*-i disainimustrist ja Java programmeerimist mugavdavatest uuendustest. Töö tulemusel moderniseeriti eelmainitud koodibaas, täiendati dokumentatsiooni ning loodi uusi, lihtsamini hallatavaid viise erinevate probleemide lahendamiseks. Uuendamisega vähendati märkimisväärselt sisuridade arvu koodibaasis, mõjutamata seejuures funktsionaalsust.

Töö raames ei muudetud tekstilisi õppematerjale ning tõenäoliselt tekib järgnevatel aastatel AKT ainet lugedes väikeseid ebakõlasid koodibaasi ja materjalide vahel.

Autor on tehtud tööga suuresti rahul – tema arvates on kood selgem ja moderniseerimise eesmärk edukalt saavutatud. Lisaks rõõmustab autorit, et tööl on praktiline väärtus ning järgnevatel aastatel ainet "Automaadid, keeled ja translaatorid" võtvaid tudengeid õpetatakse vähesema trafarettkoodiga koodihoidlas.

Viited

Agnihotri M. ja Chug A. (2024). Understanding the Effect of Batch Refactoring on Software Quality. *International Journal of System Assurance Engineering and Management* 15.6, lk 2328–2336. DOI: 10.1007/s13198-023-02247-x.

Apache (2025). Apache Velocity Engine - User Guide. https://velocity.apache.org/engine/1.7/user-guide.html (22.04.2025).

Chidamber S. R. ja Kemerer C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20.6, lk 476–493. DOI: 10.1109/32.295895.

Connolly Bree D. ja Ó Cinnéide M. (2023). Energy Efficiency of the Visitor Pattern: Contrasting Java and C++ Implementations. *Empirical Software Engineering* 28.6, lk 145. DOI: 10.1007/s10664-023-10387-8.

Fowler M. (1999). Refactoring: Improving the Design of Existing Code. 1st Edition. Addison-Wesley.

– (2018). Refactoring: Improving the Design of Existing Code. 2nd Edition. Addison-Wesley.

Gamma E., Helm R., Johnson R. ja Vlissides J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Gosling J., Joy B., Steele G. ja Bracha G. (2000). The Java Language Specification. 2nd Edition. Addison-Wesley.

Hong S., Zhang Y., Li C. ja Bai Y. (2022). ReInstancer: Automatically Refactoring for Instanceof Pattern Matching. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. Pittsburgh, Pennsylvania: ACM, lk 183–187. DOI: 10.1145/3510454.3516868.

Lynch D. C. ja Rose M. T. (1994). Internet System Handbook. Addison-Wesley.

Mens T. ja Tourwé T. (2004). A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30.2, lk 126–139. DOI: 10.1109/TSE.2004.1265817.

Opdyke W. F. ja Johnson R. E. (1992). Refactoring Object-Oriented Frameworks. *University of Illinois at Urbana-Champaign*.

Palsberg J. ja Jay C. B. (1998). The Essence of the Visitor Pattern. *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac '98) (Cat. No.98CB 36241)*. Vienna, Austria: IEEE Comput. Soc, lk 9–15. DOI: 10.1109/CMPSAC.1998.716629.

Sehgal R., Mehrotra D., Nagpal R. ja Sharma R. (2022). Green Software: Refactoring Approach. *Journal of King Saud University - Computer and Information Sciences* 34.7, lk 4635–4643. DOI: 10.1016/j.jksuci.2020.10.022.

Zhang Y., Li C. ja Shao S. (2021). ReSwitcher: Automatically Refactoring Java Programs for Switch Expression. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Wuhan, China: IEEE, lk 399–400. DOI: 10.1109/ISSREW53611.2021.00108.

Wilhelm R. ja Seidl H. (2010). Compiler Design: Virtual Machines. Springer. DOI: 10.1007/978-3-642-14909-2.

Lisad

Lisa A. Liidese NodeInterface meetod getChildNodes

```
default List<NodeInterface> getChildNodes() {
  // tagastatav järjend
  List<NodeInterface> nodes = new ArrayList<>();
  // oskame tegeleda vaid kirjetega
  Class<?> clazz = this.getClass();
  if (clazz.isRecord()) {
    // käime läbi kõik Record väljad
    for (RecordComponent recordComponent : clazz.getRecordComponents()) {
      try {
        // rekursiivne kutse
        Object field = recordComponent.getAccessor().invoke(this);
        // kui väljasid leidub
        if (field != null) {
          if (field instanceof Collection<?> collection) {
            // kui väli on mingit tüüpi collection (tavaliselt prog())
            // siis lisame iga NodeInterface elemendi
            // või kui pole NodeInterface, siis lisame StringNode ümber
            for (Object obj : collection) {
              if (obj instanceof NodeInterface) {
                nodes.add((NodeInterface) obj);
              } else {
                nodes.add(new StringNode(String.valueOf(obj)));
              }
            }
          } else if (field.getClass().isArray()) {
            // kui array, siis lisame iga elemendi StringNode sees
            for (Object obj : (Object[]) field) {
              if (obj instanceof NodeInterface) {
                nodes.add((NodeInterface) obj);
              } else {
                nodes.add(new StringNode(String.valueOf(obj)));
              }
            }
          } else if (field instanceof NodeInterface node) {
            // kui üksik NodeInterface tipp, siis lisame
            nodes.add(node);
          } else {
            // kui leht (arvuline-tekstiline väärtus), siis loome StringNode tüüpi objekti
            nodes.add(new StringNode(String.valueOf(field)));
          }
        } else {
          // lisame graafikule ka null-väärtused
          nodes.add(new StringNode("null"));
        }
      } catch (IllegalAccessException | InvocationTargetException e) {
        throw new RuntimeException(e);
```

```
        }
      }
    } else {
      throw new IllegalArgumentException(
        "getChildNodes() oskab käsitleda vaid kirjeid (Record)."
      );
    }
    // tagastame järjendi
    return nodes;
  }
```

Lisa B. Meetodi toString malli dokumentatsioon

Kasutamise juhend

Mall on kirjutatud Java-põhises mallikeeles Apache Velocity, mida IntelliJ kasutab erinevate mallide jaoks[27].

1. Ava soovitud kirje (`record`) või klass.
2. Kasuta paremklikki või klahvikombinatsiooni (Windowsis vaikimisi `Alt+Insert`), et avada meetodite genereerimise menüü.
3. Vali `toString()` genereerimine.
4. Vali ülemisel ribal `Settings`.
5. Vaheta end vaheaknale `Templates`.
6. Loo üleval vasakul nuppu + või Windowsi arvutis klahvi `Insert` vajutades uus mall.
7. Määra uuele mallile nimi.
8. Kopeeri `toString_mall.txt` sisu ja kleebi see malliaknasse.
9. Eemalda tabulatsioonid ja muud tühemikud (`whitespace`). Kui tühemikke mitte eemaldada, lisatakse kõik tühemikud ka genereeritavasse koodi. Lihtsaim viis tühemikud eemaldada on teha kogu mallikood aktiivseks (Windowsis `Ctrl+A`) ning vajutada `Shift+Tab`, kuni kogu kood on vasakule joondunud.
10. Vajuta `Ok`.
11. `toString()` genereerimise aknas ava üleval rippmenüü ning vali enda loodud mall.
12. Määra soovitud parameetrid ja vajuta `Ok`, et `toString()` genereerida. Kui `toString()` on juba olemas, annab IntelliJ sellest teada – käitu vastavalt soovile.

Kitsaskohad

1. Kui liidese või abstraktse klassi `KeelNode` meetodid ei vasta väiketähtedega klassinimedele, siis on genereeritav `toString()` ebakorrektne ning vajab manuaalset korrigeerimist.

[27] https://www.jetbrains.com/help/idea/generate-tostring-settings-dialog.html

2. Kui kirjes või klassis kasutatakse alamobjekti, näiteks `enum`, siis on algselt genereeritav `toString()` tõenäoliselt ebasobiva kujuga ning vajab manuaalset korrigeerimist.
3. Mallikoodi puhtuse huvides leidub enamikes genereeritud `toString()` meetodites konkateneerimine tühisõnega.
4. Primitiivide massiivide (`primitive array`) puhul tugineb mall `Arrays.stream()` meetodile, mis nõuab mõne keele puhul eraldi importimist.
5. Primitiivide massiivide (`primitive array`) või klassi `Collections` alamklasside puhul on tühja järjendi korral väljund `""` või `''`, mitte näiteks tühi massiiv või `null`.
6. Rakenduvad samad piirangud, mis kõikide AKT keelte puhul: keele AST klassid peavad olema formaadis `KeelKlass` ning loomismeetodid väiketähtedega, et kõik sujuvalt koos töötaks.
7. Potentsiaalse alamtüübi määramine on tehtud regulaaravaldise abil ja mitte täielikult ohukindlalt, seega kui klassinimi peaks sisaldama sama alamsõne, mis mõni Java klass, näiteks `Character` või `String`, siis võib mall ekslikult arvata, et vaadeldav klass või selle alamklass vastab nendele Java klassidele.
8. Kui klassi `Collections` alamklassi puhul pole alamtüüp `Character` või `String`, siis rakendatakse väga üldine regulaaravaldis, mis asendab look- ja kantsulud tühisõnedega. Seetõttu võib mõne genereerimise puhul käitumine olla ootamatu.

Malli inimloetav kuju

Inimloetavast kujust on välja jäetud reavahetused: kui reavahetusi pole mainitud, siis on iga kirjutamise `kirjuta` lõpus reavahetus.

Kirjutatav tekst on `` ` `` vahel.

Mugavuse ja loetavuse huvides paljud jutumärgid ja plussid ära jäetud.

```
Kirjuta `public java.lang.String toString() {`

Kui isendivälju > 0:
  Määra i=0
  Kirjuta `[klassinimi-keel väiketähtedes](`
  Iga isendivälja väli puhul:
    Kui i=0:
      Kirjuta `"` ilma reavahetuseta
    Muidu:
      Kirjuta `", ` ilma reavahetuseta
```

```
Määra alamtüüp=""
Kui väli on primitiivide massiiv:
  Määra alamtüüp=[välja tüüp, millest eemaldatud kantsulud]
Muidu kui väli on objektimassiiv:
  Määra alamtüüp=[välja tüüp, millest eemaldatud kantsulud]
Muidu kui väli on Collections alamklass:
  Määra alamtüüp=[välja tüüp, millest eemaldatud (?<=<)[^>]+(?=>)]

Kui väli on Boole'i muutuja:
  Kirjuta `[välja sisu ilma muutusteta]`
Muidu kui väli on objektimassiiv:
  Kui alamtüüp sisaldab alamsõne "Character":
    Kirjuta `Arrays.stream([välja sisu]).
      {pane ümber ühekordsed jutumärgid,
        ühenda komadega, viimasele ära lisa koma}`
  Muidu kui alamtüüp sisaldab alamsõne "String":
    Kirjuta `Arrays.stream([välja sisu]).
      {pane ümber kahekordsed jutumärgid,
        ühenda komadega, viimasele ära lisa koma}`
  Muidu:
    Kirjuta `Arrays.toString([välja sisu])`

Muidu kui välja tüüp või alamtüüp on Character:
  Kirjuta `'[välja sisu ilma muutusteta]'`
Muidu kui välja tüüp või alamtüüp on String:
  Kirjuta `"[välja sisu ilma muutusteta]"`
Muidu kui väli on arvuline:
  Kirjuta `[välja sisu ilma muutusteta]`
Muidu kui väli on primitiiv: (siin saab olla vaid char)
  Kirjuta `'[välja sisu ilma muutusteta]'`
Muidu kui väli on primitiivide massiv:
  Kui alamtüüp on boolean:
    Kirjuta `Arrays.toString([välja sisu]).{eemalda kantsulud} +`
  Muidu kui alamtüüp on char:
    Kirjuta `Arrays.toString([välja sisu]).
      {eemalda kantsulud,
        ümbritse elemendid ühekordsete jutumärkidega} +`
  Muidu:
    Kirjuta `Arrays.toString([välja sisu]).
      {eemalda kant- ja looksulud} +`

Muidu kui väli on Collections alamklass:
  Kui alamtüüp sisaldab alamsõne "Character":
    Kirjuta `[väli].stream().
      {pane ümber ühekordsed jutumärgid,
        ühenda komadega, viimasele ära lisa koma} +`
  Muidu kui alamtüüp sisaldab alamsõne "String":
    Kirjuta `[väli].stream().
      {pane ümber kahekordsed jutumärgid,
```

```
          ühenda komadega, viimasele ära lisa koma} +`
      Muidu:
        Kirjuta `Arrays.toString([välja sisu]).
          {eemalda kant- ja looksulud}`

    Muidu:
      Kirjuta `[välja sisu ilma muutusteta]`
    Määra i=i+1
  Kirjuta `);`
Muidu
  Kirjuta `[klassinimi-keel väiketähtedes]();`
Kirjuta `}`
```

Lisa C. Meetodi toString mall

Loetavuse huvides tühimikke lisatud; tegelik mall on tühimike ja taaneteta.

```
public java.lang.String toString() {
#if ( $members.size() > 0 )
  #set ( $i = 0 )
return "${classname.replaceFirst("[A-Z][a-z]*", "").toLowerCase()}(" +
  #foreach( $member in $members )
    #if ( $i == 0 )
    "##
    #else
    ", ##
    #end
    #set ( $childType = "" )
    #if ( $member.primitiveArray )
      #set ( $childType = $member.typeName.replaceAll("\\[\\]", "") )
    #elseif ( $member.objectArray )
      #set ( $childType = $member.typeName.replaceAll("\\[\\]", "").trim() )
    #elseif ( $member.collection )
      #set ( $childType = $member.typeName.replaceAll("(?<=<)[^>]+(?=>)").trim() )
    #end
    #if ( $member.boolean )
    " + $member.accessor +
    #elseif ( $member.objectArray )
      #if ( $childType.contains("Character") )
      " + java.util.Arrays.stream( $member.accessor )
        .map(s -> "'" + s + "'")
        .reduce((s1, s2) -> s1 + ", " + s2).orElse("") +
      #elseif ( $childType.contains("String") )
      " + java.util.Arrays.stream( $member.accessor )
        .map(s -> "\"" + s + "\"")
        .reduce((s1, s2) -> s1 + ", " + s2).orElse("") +
      #else
      " + java.util.Arrays.toString( $member.accessor ) +
      #end
    #elseif ( $childType == "Character" || $member.typeName == "Character" )
    '" + $member.accessor + "'" +
    #elseif ( $childType == "String" || $member.string )
    \"" + $member.accessor + "\"" +
    #elseif ( $member.numeric )
    " + $member.accessor +
    #elseif ( $member.primitive )
    '" + $member.accessor + "'" +
    #elseif ( $member.primitiveArray )
      #if ( $childType == "boolean" )
      " + java.util.Arrays.toString( $member.accessor )
        .replaceAll("[\\[\\]]", "") +
      #elseif ( $childType == "char" )
      " + "'" + Arrays.toString( $member.accessor )
```

```
        .replaceAll("[\\[\\]]", "")
        .replaceAll(", ", "', '") + "'" +
      #else
      " + java.util.Arrays.toString( $member.accessor )
        .replaceAll("[\\[\\]{}]", "") +
      #end
    #elseif ( $member.collection )
      #if ( $childType.contains("Character") )
      " + $member.accessor .stream()
        .map(s -> "'" + s + "'")
        .reduce((s1, s2) -> s1 + ", " + s2).orElse("") +
      #elseif ( $childType.contains("String") )
      " + $member.accessor .stream()
        .map(s -> "\"" + s + "\"")
        .reduce((s1, s2) -> s1 + ", " + s2).orElse("") +
      #else
      " + $member.accessor .toString().replaceAll("[\\[\\]{}]", "") +
      #end
    #else
    " + $member.accessor +
    #end
    #set ( $i = $i + 1 )
  #end
")";
#else
return "${classname.replaceFirst("[A-Z][a-z]*", "").toLowerCase()}()";
#end
}
```

Litsents

Mina, **Adeline Talvik**,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

   Aine "Automaadid, keeled ja translaatorid" koodibaasi moderniseerimine Java 21-le,

   mille juhendaja(d) on Simmo Saan,

   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Adeline Talvik
13.05.2025