TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Sten Vaher

Õppijate lahendusprotsesside analüüsimine Thonny logifailide põhjal

Bakalaureusetöö (9 EAP)

Juhendaja: Heidi Meier, MSc

Tartu 2020

Õppijate lahendusprotsesside analüüsimine Thonny logifailide põhjal

Lühikokkuvõte:

Programmeerimise õppimine on protsess, mille puhul õpetaja ei näe, kuidas õppija oma teadmisi omandab. Ainuke senine vahend õpetajate jaoks on õppijatele ülesannete lahendada andmine, mille käigus saab hinnata lahenduse korrektsust, kuid puudu jääb tihtilugu lahenduskäik. Thonny on programmeerimiskeskkond, mis salvestab ülesande lahenduskäigu logifaili ning MOOC-i „Programmeerimise alused“ käigus koguti õppijate arvestusülesannete lahenduskäigu logifailid. Logifailidest on võimalik näha iga õppija lahendusprotsessi, mille uurimine võib anda õpetajatele täpsemaid juhtnööre õppijate edaspidiseks õpetamiseks. Bakalaureusetöö käigus uuriti MOOC-i arvestustöö logifaile, märkides erinevaid parameetreid. Eesmärk oli kirjeldada erinevate õppijate lahendusprotsesse ning uurida erinevusi õppijate vahel. Peamiseks uuritavaks tingimuseks oli õppijate, kes alustavad testimisega varakult, võrdlus õppijatega, kes teevad suurema osa koodist enne testimist valmis. Saadud tulemuste põhjal tuli välja, et pideva testimise õpetamine ei pruugi olla ainuõige lähenemine programmeerimise õpetamisele. Töö käigus selgus, et enim on õppijatel probleeme andmetüüpide ja süntaksiga. Selle tulemusena pakuti välja, et iga õppija võiks korra mõnele targemale inimesele lahti seletada enda kirjutatud koodi.

Võtmesõnad:

Thonny, logifailid, Python, õppeprotsess, analüüs, MOOC

CERCS: P175 – Informaatika, süsteemiteooria

Analysis of Student Learning Process Using Thonny Log Files

Abstract:

The process of studying programming is a process at which the teachers are unable to know how the student acquires their knowledge. The only tool for teachers to assess the student this far is to give the students individual tasks to solve. This only assesses the correctness of the solution but will not give an overview of the process which leads to the solution itself. Thonny is a programming environment that stores the solution process in log files and during the MOOC “Introduction to Programming”, the log files of the final test were collected. It is possible to see the process which lead to each student’s solution from these files and this information could be helpful for the teachers for upcoming semesters. During this study log files of the MOOC’s finals are analysed. The main objective of the study is to describe the process of solving programming problems for students that are starting programming

and to compare students that are testing their code continuously against students that are not. The most important feature that is being looked at is whether the students starts with testing at the start of their code writing or not. As it turns out, teaching students to constantly test after every move, might not be the only way to learn programming. It turned out that most errors students did were object and syntax errors. This lead to a possible solution of making each student explain their code to another person that knows more about programming.

Keywords:
Thonny, log files, Python, process of studying, analysis, MOOC
CERCS: P175 – Informatics, systems theory

Sisukord

Sissejuhatus

Iga aastaga tuleb maailma juurde uusi tehnoloogilisi lahendusi ning selle tulemusena soovib aina enam eri vanuses inimesi õppida programmeerima. Tartu Ülikooli laekus aastal 2019 ainuüksi informaatika bakalaureuseõppe erialale 406 avaldust, samas kui alles 5 aastat tagasi oli avaldusi 364 [1]. Lisaks õpetatakse programmeerimist ka MOOC-idel. Näiteks on Tartu Ülikooli korraldatud kursustel „Programmeerimise alused“ ja „Programmeerimisest maalähedaselt“ osalejaid kokku olnud vastavalt 5127 ja 8656 [2]. Aina suureneva nõudluse tõttu on tarvis muuta õpetamise protsess aina efektiivsemaks.

Tartu Ülikoolis õpetatakse algkursuseid üliõpilastele programmeerimise keeles Python ning soovitatakse neil kasutada Tartu Ülikooli endise õppejõu Aivar Annamaa loodud programmeerimiskeskkonda Thonny. Selle keskkonna loomisel oli üheks oluliseks eesmärgiks parandada õppimise ning õpetamise protsessi. Thonny puhul on väga oluliseks õpetamist parandavaks funktsionaalsuseks see, et Thonny koostab programmi kirjutamise jooksul kasutaja tehtud tegevustest logifaili. Seega on logifailis informatsiooni kogu ülesande lahenduse käigu kohta ning nende hilisem uurimine annab ülevaate õppijate lahenduskäikude erinevustest.

Varem on Thonny logifaile uuritud erinevates bakalaureuse- ja magistritöödes, mille käigus on loodud edaspidist uurimist lihtsustavaid programme ja analüüsitud õppijate lahendusprotsesse. Magistritöödes on uuritud õppijate käitumismustreid [3] ja bakalaureusetööde raames on loodud programme, mis analüüsivad logifailide sisu [4] [5] ning ka programm, mis korrastab logifaile, ehk mis pakib lahti tuntumaid arhiiviformaate ning teeb eraldi logifailid nendest logidest, milles sisalduvad mitmete ülesannete lahendused [6]. Välisriikides on uuritud õppijate lahendusprotsesse läbi lahenduskäigu jooksul kogutud hetkevõtete ja nupuvajutuste [7].

Bakalaureusetöö eesmärk on tutvustada Thonny logifailide põhjal õppijate lahendusprotsesse ja selgitada, kuidas erinevad nende õppijate lahendusprotsessid, kes testisid programmi jooksvalt ja kes seda ei teinud. Lisaks pakutakse tulemuste põhjal ideid õppimisprotsessi edendamiseks.

Uurimistöös analüüsitakse 2018. aasta MOOC-i „Programmeerimise alused“ õppijate arvestusülesande lahendusprotsesse, mille korrastamiseks ja analüüsimiseks kasutatakse varem loodud programme [5] [6]. Õppijate lahendusprotsesside grupeerimiseks kasutatakse

varasema magistritöö käigus loodud grupeerimisaluseid [3]. Selles töös lähenetakse õppijate õppeprotsessi käigu analüüsimisele varasemate töödega võrreldes suurema valimiga.

Esimeses peatükis käsitletakse uurimistöö raamistust, käsitletakse uurimistöö läbiviimiseks olulisi aspekte. Tutvustatakse ka programmeerimise algõppega seonduvat informatsiooni ning MOOC-i „Programmeerimise alused“. Samuti tutvustatakse programmeerimiskeskkonda Thonny ja varasemaid uuringuid. Teises peatükis käsitletakse uurimuse metoodikat. Kolmandas peatükis kirjeldatakse Thonny logifailides sisalduvat informatsiooni ja analüüsitakse õppijate lahendusprotsesse Thonny logifailide põhjal.

1. Programmeerimise algõpe ja Thonny

Selles peatükis kirjeldatakse uurimistööga seotud olulisi aspekte. Esimeses kahes alapeatükis kirjutatakse TÜ programmeerimise algkursustest ja täpsemalt ka MOOCist „Programmeerimise alused“, mille arvestusülesande logid on selle bakalaureusetöö analüüsi aluseks. Kolmandas alapeatükis antakse ülevaade programmeerimiskeskkonnast Thonny. Viimases alapeatükis käsitletakse varasemaid uurimistöid, mis on seotud õppijate lahendusprotsesside uurimisega.

1.1 Programmeerimise algkursused

Programmeerimisega alustamine muutub iga aastaga lihtsamaks. Kui arvutite tuleku algaegadel oli selle õppimine võimalik vaid ülikoolides, siis nüüd tuleb iga aastaga juurde erinevate tehnoloogiate õpetamisele tuginevaid koolitusi ning suure arengu on ka läbi teinud internetipõhised kursused ja materjalid. Praegusel hetkel on programmeerimise õpetamise üks probleem see, et õpetajal on võimalik näha ainult õppijate lõpplahendust. Selle olukorra lahendamiseks kogutaksegi lahenduste kohta logifaile, milles kajastub kogu õppija lahendusprotsess.

Bakalaureusetöö põhineb 2018. aasta MOOC-i „Programmeerimise alused“ arvestustööde logifailide uurimisel. Seega tutvustatakse kõigepealt MOOC-ide päritolu ning seejärel vaadeldakse Tartu Ülikooli korraldatud programmeerimise MOOC-e.

Yuan ja Powell kirjutavad oma artiklis järgnevat. Sõna MOOC on akronüüm ingliskeelsetest sõnadest *massive*, *open*, *online*, *course*. Termini MOOC võttis esimest korda kasutusele Dave Cormier, kes kasutas seda, et kirjeldada Siemensi ja Downesi „Connectivism and Connective Knowledge“ kursust. Kursus oli loodud 25 õppija jaoks, kes said selle eest ainepunkte ning samal ajal avati see tasuta internetis kõigile soovijatele. Selle avamise tulemusena registreeris tasuta kursusele 2 300 inimest üle maailma [8].

Tartu Ülikooli arvutiteaduste instituut on pakkunud peamiselt kolme programmeerimise teemalist MOOC-i: „Programmeerimisest maalähedaselt”, „Programmeerimise alused” ja „Programmeerimise alused II”. Nendest esimese kahe sihtgrupiks on kõik inimesed, kellel varasem kokkupuude programmeerimisega puudub ja viimane neist arendab edasi varasemalt kogutud teadmisi.

Tartu Ülikoolis alustati MOOC-ide loomisega 2014. aastal ning esimene eestikeelne MOOC oli „Programmeerimine maalähedaselt“. Järgmise aasta kevadel võttis sellest osa 650 õppijat ning sügisel juba 1534 õppijat. Tartu Ülikoolis toimus 2015. aastal 7 MOOC-i, mille lõpetas 54% alustajatest, samal ajal kui mujal maailmas lõpetas 5-15% MOOC-i alustajatest. [9] [10].

1.2 MOOC „Programmeerimise alused“

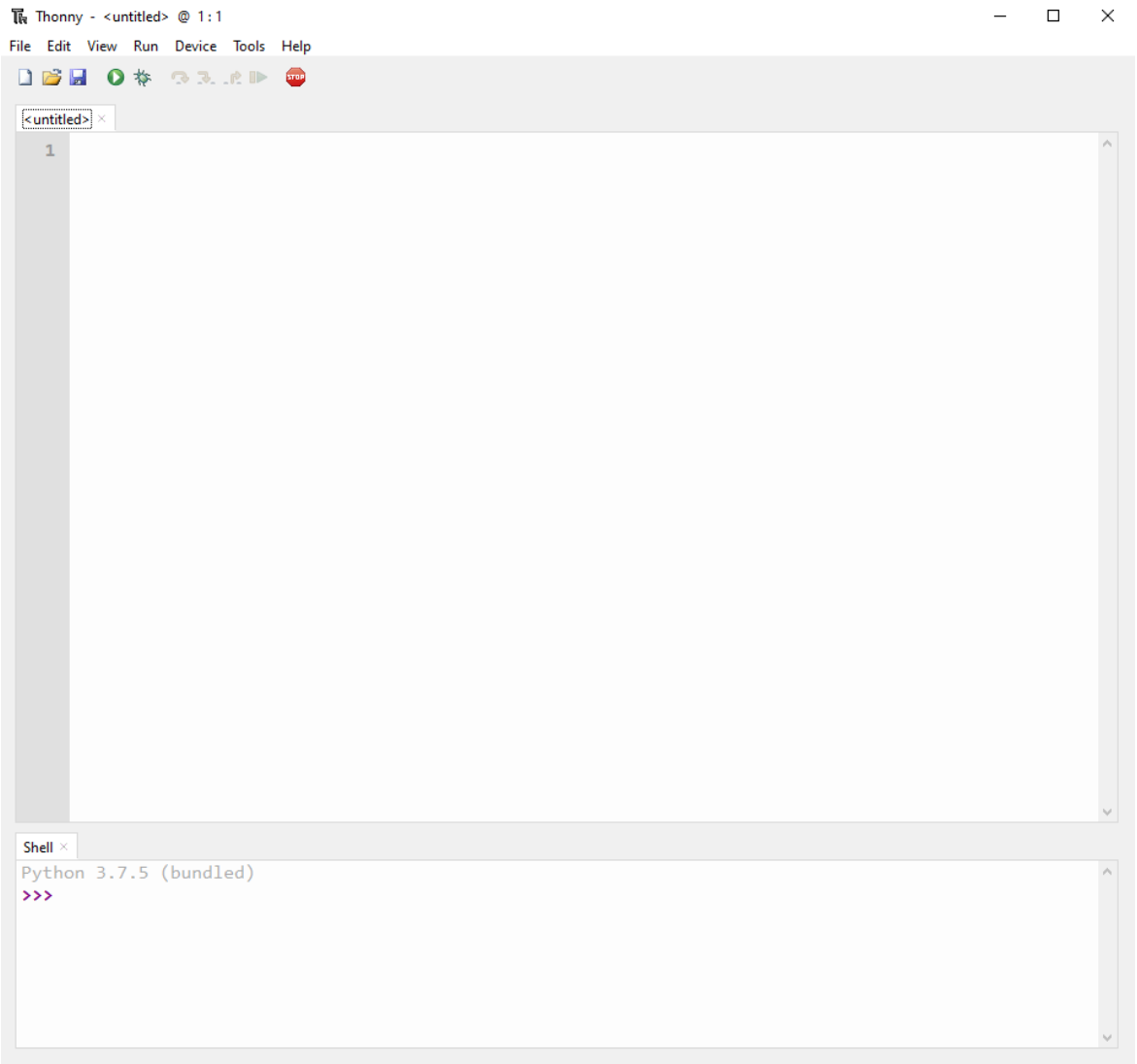
Bakalaureusetöö raames käsitletakse põhiliselt MOOC-i „Programmeerimise alused“, millele registreerimisel oli soovitatav läbida varasemalt „Programmeerimisest maalähedaselt“ kursus, kuid oli ka võimalik hakkama saada ilma. MOOC-i eesmärk on tutvustada algoritmilist mõtteviisi ja programmeerimist neile, kes sellega varasemat kokkupuudet ei oma [11].

Kursust on korraldatud 7 korda. Iga kursus kestis 8 nädalat, kus igal nädalal tutvustati uut programmeerimisega seotud teemat. Iganädalase teema ajal tuli lahendada erinevaid ülesandeid. MOOC-i lõpus tuli kõikidel õppijatel läbida arvestustöö, kus võeti kogu õpitud informatsioon kokku. Õppijad, kes kirjutasid Thonnys enda töö, pidid lisaks lahendusele esitama ka logifailid. Samas kui õppija ei lahendanud Thonnys, pidi ta tegema kokkuvõtte, kus tuli kirjeldada lahendamise raskusi ja kergusi [12].

1.3 Programmeerimiskeskkond Thonny

Thonny on algajasõbralik Pythoni integreeritud programmeerimiskeskkond, mille põhirõhk on algaja töö visuaalsemaks muutmisel. Thonny autor on Tartu Ülikooli endine õppejõud Aivar Annamaa, kes tegi selle tarkvara, et aidata ülikooli programmeerimise algkursustel olevaid õppijaid. Tegu on vabavaralise tarkvaraga, mida on võimalik paigaldada Windowsis, Macis ja Linuxis [13].

Thonny autori sõnul on tegu programmeerimiskeskkonnaga, milles on kokku koondatud kõik algajale vajaminevad funktsionaalsused. Terve arendusprotsessi käigus oli põhirõhk algajatel programmeerijatel ja mitte üldisel kasutajakogemusel nagu teiste sarnaste keskkondade puhul (joonis 1). Siiski kuna õpetajaid on erinevaid ja kõik inimesed ei õpi samamoodi, tehti Thonny lähtekood avalikuks ning nad pooldavad igati keskkonna endale sobivaks muutmist [14].



Joonis 1. Thonny kasutajaliides

MOOC-i „Programmeerimise alused“ raames soovitati õppijatel kasutada õppetöö sooritamiseks Thonnyt. Bakalaureusetöö raames on kõige olulisemal kohal Thonny funktsionaalsus, mille abil keskkond logib õppija iga kasutussessiooni. Logis kajastub iga lahendaja tehtud liigutus, mis on seotud Thonnyga, näiteks märgitakse klahvivajutused, salvestamised jpm.

1.4 Õppijate lahendusprotsesse käsitlevad uurimused

Õppimine on protsess, mille tulemused varieeruvad ning mida igaüks kogeb erinevalt. Sellest tulenevalt on ka õppimise protsessi uurimisele lähenetud mitme külje alt. Viimastel aastatel on Tartu Ülikoolis tehtud mitmed tööd, kus uuritakse programmeerimisülesannete lahendusprotsesse, kuid kõik nendest on läbi viidud väikesel õppijate valimil.

Õppijate käitumismustritest on kirjutanud H. Meier, kes liigitas õppijad protsessi järgi kaheks: „Müüriladujad“ ja „Kiviraiujad“. Nende peamine erinevus oli esimene hetk, mil õppija hakkas testima oma kirjutatud koodi. „Müüriladujad“ alustasid koodi testimist kohe algusest ja kirjutasid järk-järgult, samal ajal kui „Kiviraiujad“ kirjutasid suurema osa oma koodist valmis ja alles siis hakkasid testima. Lisaks koostati grupp „Meistrid“ õppijatest, kelle lahendusprotsess oli liiga lühike, et neid üheselt grupeerida [3].

Samal ajal K. Pedel uuris enda töös õppijate erinevaid koodikirjutamise aspekte, näiteks ülesande lahendamise ajal käivitamiste arv ja vahe salvestamiste arvuga. Töö põhiosas tegi autor programmi, mis muutis lihtsamaks õppijate logifailidest informatsiooni kättesaamise [4].

Pythoni algõppega tegelevate MOOC-ide „Programmeerimise alused“ ja „Programmeerimisest maalähedaselt“ veateadetest on kirjutanud magistritöö R. Kodasmaa. Autor leidis, et kõige sagedamini esinevad vead on süntaksivead, millele järgnevad tüübivead ja nimevead [15].

Lisaks on logifailidest informatsiooni kättesaamiseks teinud programme ka A. Roosi ja V. Tõnisson, kes oma töödes võtsid sisendiks õppijalt saadud logifaili ja üritasid sealt informatsiooni kätte saada. Tõnissoni koostatud programm abistab uurijaid, kellel on vaja saada pikast logifailist kätte ainult ühe ülesande lahendus, eraldades logifaili mitmeks eraldi failiks vastavalt kasutaja soovile ning lisaks pakib see ka lahti erinevaid arhiivfaile [6]. Samuti pakib programm lahti arhiivifaile. Samas Roosi programm koostas visuaalse kokkuvõtte erinevatest logifailis olevatest väärtustest, näiteks käivitamiste arv, vigade arv ja kopeerimiste arv [5].

Rahvusvaheliselt on tegelenud Soome teadlased õppijate hetkevõtete (ingl *snapshots*) ja nupuvajutuste (ingl *keystrokes*) uurimisega. Kuna ainult lõpptulemuse järgi on rakse hinnet panna, kogutakse õppijatelt oma töö protsessi kohta hetkevõtteid. Soome ülikoolis uuriti peamiselt, kui tihti õppijad teevad hetkevõtteid ja mitu nupuvajutust jääb hetkevõtete vahele [7]. Õppijate tüüpidesse jagamisega on tegeletud Java ülesannete lahendamise kontekstis ning saadud kolm tüüpi: „stoppers“, „movers“, „tinkerers“. Esimesed jäid probleemi lahendamise mitteoskamise ajal seisma, teised liiguvad vaikselt probleemi lahendusele lähemale ning kolmandad proovivad igati programmi lihtsalt tööle saada [16]. Veateadete uurimisega tegelesid Marceau, Fisler ja Krishnamurti, kes leidsid, et veateadete õppijatele näitamine aitas neil ülesande lahendamisega edasi liikuda. Nad salvestasid õppijate tehtud muudatused

iga klahvivajutuseni ning selle põhjal üritasid leida, millised veateated on õppijatele kõige raskemad [17].

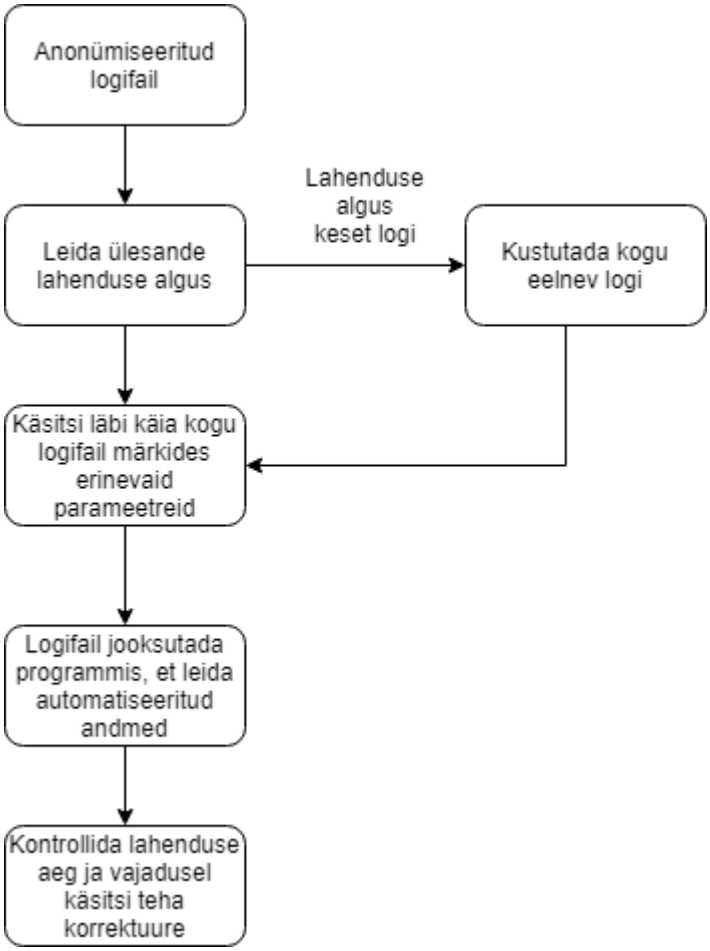
2. Materjal ja metoodika

Selles peatükis kirjeldatakse, mis vahendite abil analüüsiti õppijate töödest saadavaid andmeid. Samuti käsitletakse, millised valikud tehti töödeldavate logifailide leidmiseks. Kirjeldatakse kogu tehtud töö järjekorda ning analüüsi metoodikat.

Bakalaureusetöös kasutati Thonny logifaile, mis pärinesid Tartu Ülikooli „Programmeerimise alused“ MOOC-i õppijatelt. Kursus toimus 2018. aasta sügisel ning kestis 8 nädala vältel. Uuritavad logid on MOOC-i arvestusülesande 8.1 lahendused. Kokku esitasid enda logid 789 õppijat. Kõik, kes kasutasid selle ülesande lahendamisel Thonnyt, pidid esitama logifailid ning ülejäänutel tuli esitada kirjalik selgitus ülesande raskuse kohta. Selgitusi bakalaureusetöös ei uuritud.

Logifailid saadi anonümiseeritud kujul, kus õppijate nimed olid asendatud järjekorranumbriga. Logifaile oli kokku 789. Kuna ühe logi läbitöötamisele kulus umbes 10 minutit aega, siis töötati läbi logifailid üle ühe ehk kõik paarituarvulise järjekorranumbriga failid. Samas 395 läbitöötatud tööst 86 olid tühjad või mõnel muul moel mittekasutatavad. Seega lõpuks jäid valimisse 309 lahenduse logifailid.

Selleks, et välja töötada protsessi logifailide läbitöötamiseks, oli tarvis enne tutvuda logifailide sisu ning varem tehtud sarnaste töödega. Algse plaani järgi taheti kasutada varasema aasta jooksul Thonny logifailide läbitöötamise lihtsustamiseks valmistatud programme. Esimene nendest oleks võinud olla V. Tõnissoni bakalaureusetöö raames koostatud programm [6], mis peaks hõlbustama logifailide läbitöötamist, kus ühes logis on mitme erineva ülesande lahendused. Kahjuks sellest programmist käesoleva töö raames abi ei olnud, kuna ta muutis protsessi aeglasemaks, ilma, et oleks andnud piisavalt head tulemust. Prooviti mitme lahendusega failist eraldada iga erineva ülesande lahendus, aga saadud tulemus oli logi, mille lõpp oli korrektne, aga algus siiski jäi muutmata. Seetõttu oli kiirem teha selle programmi plaanitavat protsessi käsitsi. Teiseks oli plaanis kasutada A. Roosi bakalaureusetöö raames loodud analüüsi automatiseerimise programmi ning seda ka tehti. Programmi tööks oli välja uurida programmi kirjutamise käigus tehtud vigade arv ning nende sisu, käivitamiste arv nii veateatega kui ilma ja kleepimiste arv koos kleebitud teksti sisuga.



Joonis 2. Logi läbitöötamise protsess

Logifailidest informatsiooni hankimise protsess nägi välja järgnev (joonis 2). Kõigepealt pakiti lahti anonüümseks tehtud zip-fail, mille sees oli õppija esitatud logifailide ülesande lahendus. Nendel õppijatel, kes arvestusülesande kirjutasid Thonnys, olid selleks lahenduseks logifailid ning kõikidel teistel tekstifail, milles kirjeldati oma töö tegemise protsessi. Viimaseid tekstifaile selle töö raames läbi ei uuritud. Logifailide esitajate lahendused jagunesid peamiselt kolmeks:

- Inimesed, kes esitasid kõik kursusel tehtud logifailid kokku pakitult
- Inimesed, kes esitasid oma viimase päeva logifailid kokku pakitult
- Inimesed, kes esitasid õiged logifailid kokku pakkimata

Sellise ebamäärase esitusstiili tõttu läks kõige kauem aega esimese grupi inimeste peale, kuna nende töödest oli kõigepealt vaja üles leida õige sektsioon, kus nad arvestusülesannet lahendasid ning seejärel õige logifail, kus ülesannet alustati. Kõige kiirem oli kolmanda grupi esitatud tööde läbitöötamine, samas nende seas oli ka kõige rohkem selliseid inimesi, kes esitasid ainult viimase poole oma tehtud ülesandest.

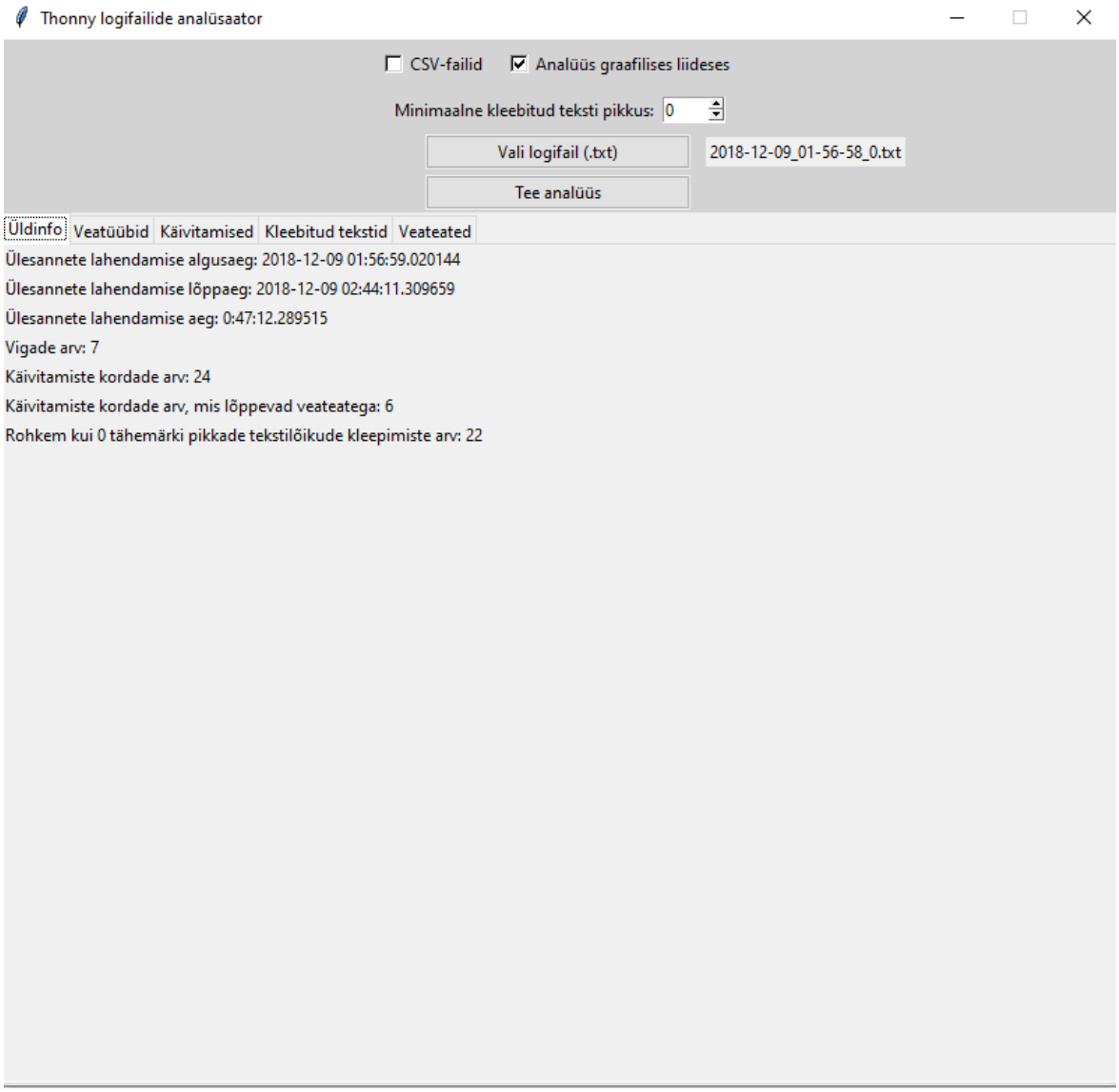
Pärast õigete logifailide väljaotsimist tuli leida nendest arvestusülesande lahendusprotsessi algus. Selle leidmiseks tuli käsitsi läbi töötada mitu logi enne, kui oli võimalik kindlaks teha, kust lahendusprotsess alguse sai. Pärast algpunkti leidmist tuli kustutada kogu eelnev, et ajavõtt oleks võimalikult ühtlane.

Järgmiseks tuli hakata logifaili käsitsi läbi töötama, samal ajal jälgides mitmeid uuritavaid parameetreid. Põhiline asi, mida töö tegemise juures jälgiti, oli see moment, mil esimest korda testiti oma programmi tööd. Kui õppija testis oma programmi enne tsükli kirjutamist ehk muutujate või funktsiooni loomise ajal, muutus „Testis jooksvalt“ väärtus tõeseks. Kui aga õppija kirjutas valmis suure osa oma lahendusest enne, kui seda testima asus, siis märgiti „Kirjutas lahenduse suures ulatuses valmis“ tõeseks. Enamikel juhtudel need olid üksteist välistavad, kuid oli paar üksikut tööd, kus esimene test tehti kohe alguses funktsiooni juures ja seejärel järgmine test pärast kogu ülesande valmis tegemist, sarnastel juhtudel võisid mõlemad väljad samaaegselt märgitud saada.

Teine väljade kombinatsioon, mida ülesande lahenduse käsitsi läbitöötamisel jälgima pidi, oli siluri ja teiste ülesannete abiks kasutamine. Kui ükskõik millisel hetkel õppija kasutas siluri nuppu, muutus selle töö puhul tõeseks „Kasutas silurit“ väli, seda isegi juhtudel, kui silurit ei kasutatud otstarbekal eesmärgil või mindi kogemata vale nupu pihta. Samas välja „Kasutas abiks eelnevat ülesannet“ tõeseks muutmiseks pidi õppija avama mõne muu faili, milles oli varasem informatsioon mõne teise ülesande lahendusega (kirja ei läinud need juhul, kui arvestusülesannet lahendati mitmes failis).

Viimane käsitsi jälgitavatest väljadest „Sai ülesande lahendatud“ tuli märkida pärast logifaili täieliku läbitöötamist, juhul kui lahenduse lõpuks õppija sai veateadetest lahti, oli see väli positiivne, vastasel juhul aga negatiivne.

Pärast käsitsi analüüsi lõpetamist pidi läbitöötatud logifaili sisestama logifailide analüsaatorisse (joonis 3). Sealt saadud tulemuste põhjal sai täita järgmised väljad: „Lahendamise aeg“, „Vigade arv“, „Käivitamise kordade arv“, „Veateatega lõppevaid käivitusi“, „Veateated“, „Kleepimiste arv“. Kuna paljud õppijad lahendasid ülesannet pikema perioodi vältel, sai vajadusel käsitsi kustutatud rohkem kui 3 tunnised pausid, et need ei viiks lahendusaega ebamääraselt kõrgeks.



Joonis 3. Analüsaatori kasutajaliides

Läbivaatuse andmete kogumisel toetuti varasematele uurimustele ning valiti võimalikult palju selliseid väljasid, mida andis automatiseeritult koguda. Varasemates uurimustes on olnud olulisel kohal testimisega alustamise aeg [3], vigade arv ja sagedus [15].

Logifailide läbitöötamisel oli olulisel kohal õppijate käitumismustrite jälgimine. Põhiliseks edasise analüüsi aluseks sai väli „Testis jooksvalt“, kuna see andis kõige enam informatsiooni õppija lahendusprotsessi kohta [3]. Selle töö raames proovitigi välja selgitada, kas pideval testimisel on mingisugune eelis.

Sarnaselt varasema magistritööga [3] sai liigitatud lahendusstiilid vastavalt esimese testi sooritamise järgi kaheks, need, kes alustasid testimist kohe algusest, ja vastupidi. Kuna siiski liigitusprotsessis olid teatavad erinevused, ei saa neid võrrelda üks-ühele.

Samuti jälgiti läbitöötamisel erinevate veateadete esinemist ning need järjestati tüüpide esinemissageduse järgi. Lisaks vaadeldi kleepimiste sisu, siluri kasutamist ja toodi välja lisamärkused selle kohta, kui palju õppija tegelikult ka aru sai sellest, mida ta kirjutas.

Andmete analüüsimisel toodi välja vastavalt varasema liigituse järgi lahendajate hea ja halb näidislahendus. Lisaks liigitati varakult testimisega alustajate ning vastupidise metoodikaga grupid aja järgi kolmeks: lahenduse aeg vähem kui üks tund, aeg ühe ja kolme tunni vahel ning üle kolme tunnine aeg. Aja järgi lahenduste liigitamine aitas uurida äärejuhtumeid, kuna leidus nii õppijaid, kes tegid koodi valmis ning jõudsid lahenduseni ühe testimisega kui ka neid, kelle ülesandele kulunud aeg oli mõõdetav päevades.

3. Õppijate lahendusprotsesside analüüs Thonny logifailide põhjal

Selles peatükis käsitletakse Thonny logifailidest saadud informatsiooni analüüsimisega seonduvat. Alguses tutvustatakse, millist informatsiooni on võimalik logifailidest kätte saada ning seejärel antakse ülevaade selle uuringu käigus saadud informatsioonist. Pärast analüüsi pakutakse välja võimalusi, kuidas edaspidi uurimisega edasi liikuda.

3.1 Thonny logifailide analüüs

Thonny puhul on tegemist programmeerimiskeskkonnaga, mis kirjutab automaatselt programmeerija ühest kasutussessioonist txt-vormingus logifaili. See tähendab, et iga kord, kui kasutaja oma keskkonna ülesande lahendamise jooksul sulgeb, tekib sellest eraldi fail. Logidest leiab informatsiooni peaaegu iga lahendaja tehtud nupuvajutuse kohta ning need on üles märgitud, kasutades programmeerimiskeelest sõltumatut andmevahetusvormingut JSON (inglise keeles JavaScript Object Notation). Selle vormingu lugemiseks tuleb tähelepanu pöörata loogeliste sulgude sees olevatele võtme ja väärtuse paaridele.

Thonny logifailides sisalduvat informatsiooni on varem käsitletud mitmetes Tartu Ülikooli lõputöödes. Esmalt on sellest kirjutanud Pedel, kes jagas Thonny logifailidest saadava informatsiooni viieks: Thonny programmiaknaga seotud kirjed, programmikoodiga seotud kirjed, failidega seotud kirjed, programme käivitamisega seotud kirjed, käskudega seotud kirjed [4]. Pedeli töö käigus tehtud klassifikatsiooni kasutas ka kaks aastat hiljem Meier, kes oma töös jälgis põhiliselt programmikoodi ja käivitamisega seotud kirjeid [3].

Selles bakalaureusetöös süvenetakse Pedeli klassifikatsiooni järgi samuti programmikoodiga seotud kirjetele ja programmi käivitamisega seotud kirjetele, kuid lisaks kasutatakse ka väikesel määral failidega seotud kirjeid, kuna järgitakse, kas õppija kasutab varem tehtud lahendusi.

3.1.1 Üldiselt kirjete lugemisest

Kuna Thonny kasutab logifailide kirjutamisel JSON andmevahetusvormingut, koosneb logifaili sisu loogelistes sulgudes olevatest võti-väärtus paaridest, mis on omavahel eraldatud komadega. Iga loogelise sulu sees on kirjeldatud ühte programmeerija tehtud tegevust.

Kõikides loogelistes sulgudes on tegevuse tüüpi kirjeldav võti „sequence“, mille sisu näitab kasutaja tehtud tegevust (nt Open, <FocusIn>, TextInsert, TextDelete, <FocusOut> jne) ja

aega näitav võti „time“, millele vastab ISO 8601 standardil põhinev ajaväärtus, kus on toodud aeg välja kuni kuue sekundi murdosa näitava numbrini kujul „yyyy-MM-dd’T’HH:mm:ss.SSSSSS“.

Tänu iga kirje puhul aja märkimisele niivõrd sügavalt, on võimalik logi kirjed panna kronoloogilisse järjekorda, millega omakorda saab kogu kasutussessiooni taasesitada.

3.1.2 Programmikoodiga seotud kirjed

Peamised programmikoodiga seotud tegevused jagunevad kaheks: sisestamine ja kustutamine. Redaktorisse koodi sisestamiseks on omakorda kaks võimalust: programmikoodi saab klaviatuurilt sisestada või saab kleepida juba varasemalt kirjutatud koodi.


```
{
    "index": "1.0",
    "text": "0",
    "tags": "None",
    "text_widget_id": 69927472,
    "text_widget_class": "CodeViewText",
    "sequence": "TextInsert",
    "time": "2018-12-09T12:35:39.889689"
},
```


Joonis 4. TextInsert tüüpi logifaili kirje programmi

Programmikoodi käsitsi sisestamise puhul tekib JSON kirje, mille “sequence” väärtus on TextInsert. See kirje koosneb 7 võti-väärtus paarist, milles võti “index” näitab teksti sisestamise asukohta kujul reanumber ja elemendi asukoht punktiga eraldatuna. Näiteks joonisel on „index“ väärtuseks 1.0, mis tähendab, et tegevus toimub esimesel real asukohaga 0 (joonis 4). Võti “text” näitab, mis väärtus “index” positsioonile kirjutatakse.

Samas “text_widget_” sõnega algavad võtmed näitavad, kuhu antud “text” kirjutatakse. Iga uue programmi puhul on “text_widget_id” erinev, kuid saadud numbriga käib kaasas peamiselt kaks erinevat “text_widget_class” väärtust: “CodeViewText” ja “ShellText”, mis vastavalt kirjeldavad, kas tegu on programmiaknasse või käsureale kirjutatud tekstiga.


```
{
    "index": "6.21",
    "text": "e",
    "tags": "('io', 'stdin')",
    "text_widget_id": 63869616,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2018-12-09T12:39:07.466799"
},
```


Joonis 5. TextInsert tüüpi logifaili kirje käsureale

Võtme „tags“ väärtus on oluline, kui väärtus sisestatakse käsureale. Nimelt jagunevad käsureale sisestatud andmed kaheks: need, mille sisestab kasutaja „stdin“ (joonis 5) ja need, mis tekivad programmi töö käigus „stdout“.


```
{
    "index": "4.36",
    "text": "float(\"4.0\")",
    "tags": "None",
    "text_widget_id": 81596912,
    "text_widget_class": "CodeViewText",
    "sequence": "TextInsert",
    "time": "2018-10-31T17:44:27.309576"
},
{
    "widget_id": 81596912,
    "widget_class": "CodeViewText",
    "text_widget_id": 81596912,
    "text_widget_class": "CodeViewText",
    "sequence": "<<Paste>>",
    "time": "2018-10-31T17:44:27.309576"
},
```


Joonis 6. Teksti kleepimine

Teksti kleepimisel tekib logisse kaks erinevat JSON tüüpi kirjet (joonis 6). Esimene nendest on TextInsert, mille sisu on varasemalt lahti seletatud ja teise kirje võtmele „sequence“ vastab „<<Paste>>“, selle kirje olemasolu puhul saadaksegi teada, et tegu on kleepimisega. Mõlemad loodud kirjed on omavahel seotud sellega, et nende ajatemplid on täpselt samad.


```
{
    "index1": "4.16",
    "index2": "None",
    "text_widget_id": 102411632,
    "text_widget_class": "CodeViewText",
    "sequence": "TextDelete",
    "time": "2018-11-01T19:35:21.355291"
},
```


Joonis 7. Teksti kustutamine programmist

Teine suurem klass kirjeid, mis on seotud programmikoodiga, hõlmab teksti kustutamise protsessi (joonis 7). Kustutamise kirjed saab samuti eristada asukoha järgi, ehk siis kirjed, mis sisestatakse programmi tekstialasse, ja need, mis sisestatakse käsureale. Vastupidiselt „TextInsert“ kirjele, ei ole tarvis kustutamisel „tags“ välja, kuna programmi töö tulemusena kunagi ühtegi rida ei kustutata.

```
{
    "index1": "38.0",
    "index2": "38.0",
    "text_widget_id": 75371696,
    "text_widget_class": "ShellText",
    "sequence": "TextDelete",
    "text_widget_context": "shell",
    "time": "2018-10-31T17:44:28.378354"
},
```


Joonis 8. Teksti kustutamine vahemikuna

Samas on kustutamise tüüpe kaks, mis eristuvad kustutatud andmete koguse kaupa (joonis 7,8). Esimene võimalus on kustutada üks märk, sel juhul võtme „index1“ väärtus täidetakse sarnaselt varasemale teksti asukoha märkimisele, kuid „index2“ väärtus jääb tühjaks. Seega kustutatakse element, mis asub kohal „index1“. Näiteks joonisel 7 on kustutatud element, mis asub 4. real ja 16. positsioonil. Teine kustutamisvõimalus on lahti saada mitmest märgist, mis kuuluvad teatud vahemikku. Selle ala alguspunkti märgib väli „index1“ ja lõppu „index2“ nagu näha jooniselt 8. Ülejäänud väljad väärtustatakse sarnaselt „TextInsert“ kirjetele.

3.1.3 Programmi käivitamisega seotud kirjed

Programmi käivitamisega tekivad mitmed erinevad kirjed, mis põhiliselt jagunevad kolmeks: käivitamise kirje, veateade, käsureale kuvatav vastus. Käesolevas alapeatükis käsitletakse kõiki kolme.


```
{
    "index": "3.4",
    "text": "%cd 'C:\\Users\\adaka\\Desktop\\DESKTOP 2\\python'\n%Run '2.1 j\u00e4\u00e4tumine.py'\n",
    "tags": "('automagic', 'toplevel', 'command')",
    "text_widget_id": 75371696,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2018-10-31T17:32:20.035026"
},
```


Joonis 9. Käivitamisel tekkiv esimene kirje

Kõige esimesena tekib programmi jooksutamisel uus TextInsert kirje (joonis 9), mille puhul võtmele “text” vastab käivitatava faili asukoht. Selle kirjega käib kaasas umbes 6 erinevate programmide jooksutamiseks vajalikku JSON kirjet, millest kõige olulisem on viimane programmijupi käivitamise kirje (joonis 10). Selles kajastub võtme “command_text” väärtusena fail, mida jooksutatakse ning samuti märgitakse ära aeg.

```
{
    "command_text": "%Run '2.1 j\u00e4\u00e4tumine.py'\n",
    "sequence": "ShellCommand",
    "time": "2018-10-31T17:32:20.463999"
},
```


Joonis 10. Käivitamise viimane kirje

Pärast käivitamisprotsessi jaguneb käivitamine kaheks: õnnestunud ja veateatega lõppev. Õnnestunud käivitamise juures veateadet ei teki ning käsureal jooksutatakse programm vastavalt kirjas olevatele juhistele. Õnnestunud käivitamise lõpetab käsureale teksti “>>>” sisestamine, kuna tegu on tekstiga, mis on Thonny käsurea alguses juhul, kui programm on ootel.

Samas ebaõnnestunud käivitamise korral, tekib programme jooksutamisel veateade. Töö käigus tuli ette 12 erinevat tüüpi veateadet, kuid logides on nende üldine struktuur väga saranane.


```
{
    "index": "18.0",
    "text": "TypeError: '>' not supported between instances of 'float' and 'str'\n",
    "tags": "('io', 'stderr')",
    "text_widget_id": 75371696,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2018-10-31T17:36:04.063277"
},
```


Joonis 11. Tüübiveaga lõppev käivitus

Programmi jooksutamisel saame teada, et on esinenud veateade, kui käsureale tekib „TextInsert“ tüüpi kirje, mille võti „tags“ sisaldab „stderr“ sõnet. Täpsemalt tekib selliseid kirjeid tervelt 6. Nende kuue kirjega väljastatakse koht, kus viga on esinenud ja veatüüp. Veateate sisu on toodud välja viimases JSON kirjes võtmele „text“ vastavas väärtuses. Joonisel 11 on näha tüübiviga, mis on tekkinud, kui kasutaja on üritanud võrrelda arvulist väärtust sõnega, kuid programmil ei ole võimalik aru saada, kas arv on suurem kui tekst. Selles veateates on kaasas oluliste elementidena „text_widget_class“, mis ütleb, et tekst tuleb lisada käsureale ja „index“, mis ütleb, kuhu käsureal see minema peab.

```
{
    "index": "458.0",
    "text": "SyntaxError: invalid syntax\n",
    "tags": "('toplevel', 'error')",
    "text_widget_id": 75371696,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2018-11-02T00:06:37.011797"
},
```


Joonis 12. Süntaksivea viimane kirje käivitamisel

Teine moodus veateate tuvastamiseks on see, kui tekib kirje, mille „tags“ võtmele vastab „error“ sõne sisaldav väärtus. Sellisel puhul võib olla tegemist näiteks süntaksiveaga „SyntaxError“. Selle teatega käib üldiselt kaasas 7 kirjet, millest olulisemad on sarnaselt teiste vigadega, viimane, kus on välja toodud veatüüp (joonis 12), kuid ka abikirjena esimene, kus on välja toodud asukoht (joonis 13). Veateate keskmiste kirjete puhul on välja toodud täpsemalt vea asukohta täpsustav informatsioon.


```
{
    "index": "455.2",
    "text": "File \"C:\\Users\\adaka\\Desktop\\DESKTOP 2\\python\\sport.py\", line 5",
    "tags": "('toplevel', 'error', 'hyperlink')",
    "text_widget_id": 75371696,
    "text_widget_class": "ShellText",
    "sequence": "TextInsert",
    "text_widget_context": "shell",
    "time": "2018-11-02T00:06:37.008800"
},
```


Joonis 13. Süntaksivea esimene kirje

Käivitamisega on seotud ka programmist saadud vastus. Thonny, nagu ka enamik teisi redaktoreid, kirjutab saadud vastuse käsureale. Sellise kirjutamise puhul on tegu alapeatükis 3.1.2 käsitletud „TextInsert“ käsuga, mille puhul „text_widget_class“ on „ShellText“ ehk kirjutatakse käsureale ja võtmele „tags“ vastab „stdout“, ehk kirjutatu tuleb programmist endast. Täpsem programmi töö logides ei kajastu.

3.1.4 Failidega seotud kirjed

Thonnys on võimalik failidega teha mitmeid erinevaid tegevusi: avada, salvestada, sulgeda. Samas selle töö raames ollakse enim huvitatud just failide avamisega seotud kirjetest, kuna selle põhjal saadakse teada, millisel ajahetkel õppija alustas ülesande lahendamist ja kas ta on kasutanud abiks teisi faile.

```
{
    "editor_id": 57425488,
    "editor_class": "Editor",
    "sequence": "NewFile",
    "text_widget_id": 57496624,
    "text_widget_class": "CodeViewText",
    "time": "2018-11-02T00:08:28.969127"
},
```


Joonis 14. Uue faili loomine

Faili loomisega luuakse logifaili uus „NewFile“ tüüpi kirje (joonis 14), mille tulemusena luuakse uus fail. Kirjega käivad kaasas mitmed varasemalt tuntud väljad. Seda kirjet kasutatakse, et leida üles koht, mil õppija alustas arvestusülesande lahendamist.


```
{
    "editor_id": 14597744,
    "editor_class": "Editor",
    "filename": "C:/Users/adaka/Desktop/DESKTOP 2/python/2.1 j\u00e4\u00e4tumine.py",
    "sequence": "Open",
    "text_widget_id": 81596912,
    "text_widget_class": "CodeViewText",
    "time": "2018-10-31T17:31:46.958394"
},
```


Joonis 15. Olemasoleva faili avamine

Teine failidega seotud kirje on „Open“ (joonis 15), mida kasutatakse, et leida vastet küsimusele, kas õppija kasutab abiks eraldi faile. Kui õppija avab faili, mis sisaldab mõne varasema ülesande lahendust, saamegi välja tõeseks märkida. Kirjega „Open“ kaasas käiv kõige olulisem võti on „filename“, mis näitab avatava faili asukohta. Pärast faili avamist tekib logisse „TextDelete“ ja „TextInsert“ kirjete kombinatsioon, mille tulemusena sisestatakse programmi faili sisu.

3.2 Õppijate lahendusprotsesside analüüs

Järgnevates alapeatükkides käsitletakse selle bakalaureusetöö raames saadud praktilise osa tulemusi. Esimene alapeatükk käsitleb õppijatele esitatud ülesande püstitust, teises tuuakse välja õppijate peamiste lahendusprotsesside kirjeldused ning kolmandas analüüsitakse kogu saadud informatsiooni.

3.2.1 Ülesande püstitus

Analüüsimisel kasutati 2018. aastal toimunud MOOC-i “Programmeerimise alused” arvestusülesande lahendusi. Kursuse raames lahendasid õppijad igal nädal ülesandeid ning lõpus testimaks nende teadmisi, tuli igaühel läbida arvestus. Töö koosnes kahest osast: esimese osa raames oli ülesanne, mille pidi igaüks lahendama ning teise osa raames tuli lahen-

duskäigu logifailid esitada. Kui õppija ei lahendanud ülesannet Thonnys, tuli ise juurde kirjutada lahendusprotsessi ülevaade. Ülesande kirjeldus kodulehel oli järgnev (MOOC Programmeerimise alused, 2018):

8.2 Arvestusülesanne

Kohustuslikult tuleb lahendada (8.1a ja 8.2a) või (8.1b ja 8.2b).

KONTROLLÜLESANNE 8.1a ARVESTUSÜLESANNE

Saja Aakri metsa elaniku karupoeg Puhhi fännidest metsaomanikud on oma metsatükkide pindalad kirjutanud aakrites (1 aaker = 0,4047 hektarit).

Ühel omanikul on ainult ühe puuliigi metsad. Konkreetsete puuliikide puhul on teada aastane metsa juurdekasv hektari kohta tihumeetrites (tm/ha). Näiteks kase puhul võib see olla 4,8 tm/ha, kuuse puhul 6,6 tm/ha, männi puhul 3,7 tm/ha.

Omanik tahab teada, mitu tihumeetrit metsa aastas teatud suurusest suuremates metsatükkides juurde kasvab.

Koostada funktsioon `juurdekasv`, mis

- võtab argumentideks metsatüki pindala (ujukomaarv aakrites) ja metsa aastase juurdekasvu hektari kohta (ujukomaarv),
- tagastab selle pindalaga metsatüki aastase juurdekasvu ümardatuna sajandikeni.

Vihje

Arvutamiseks võib kasutada valemit: (metsatüki juurdekasv) = (metsatüki pindala aakrites) * 0,4047 * (aastane juurdekasv)

Koostada programm, mis

- küsib kasutajalt
  - failinime (failis on eraldi ridadel metsatükkide pindalad aakrites);
  - vastava puuliigi aastast juurdekasvu hektari kohta tihumeetrites (ujukomaarv);
  - piiri, mitmest aakrist suuremad metsatükid arvesse võtta (ujukomaarv);
- loeb failist metsatükkide pindalad;
- arvutab (funktsiooni `juurdekasv` abil) ja väljastab metsatüki aastase juurdekasvu, kui selle metsatüki pindala on sisestatud piirist suurem;
- väljastab teate “Metsatükki ei võeta arvesse”, kui metsatüki pindala ei ole sisestatud piirist suurem;
- väljastab lõpuks ekraanile, mitme metsatüki juurdekasv arvutati.

Näide funktsiooni `juurdekasv` rakendamisest

```
>>> juurdekasv(3.78,6.6)
10.1
>>>
```

Näide programmi tööst

Faili andmed.txt sisu:

0.9
3.78
2.05
1.58

```
>>> %Run yl8.1a.py
  Sisestage failinimi: andmed.txt
  Sisestage aastane juurdekasv hektari kohta tihumeetrites: 6.6
  Sisestage piir, mitmest aakrist suuremad metsatükid arvesse võtta: 2
  Metsatükki ei võeta arvesse
  Metsatüki aastane juurdekasv on 10.1
  Metsatüki aastane juurdekasv on 5.48
  Metsatükki ei võeta arvesse
  Arvutati 2 metsatüki juurdekasv.
>>>
```

Arvestusülesandele sarnane ülesanne koos ühe võimaliku lahendusega võib olla ka abiks mõtete kogumisel. Lahendamise video

KONTROLLÜLESANNE 8.2a ARVESTUSÜLESANDE LAHENDAMISE PROTSESS

Esitamine *Moodle*'is.

Tekstina esitatakse arvamus arvestusülesande raskusastme ja sobivuse kohta. Thonny kasutajad peavad esitama logifaili. Kes teeb mõne teise vahendiga, see peab põhjalikumalt kirjeldama lahendamise raskusi ja kergusi ning samuti peab andma võimalikult täpse ajalise ülevaate, kui palju ülesande lahendamisele aega kulus.

Logifaili saamise video

JUHEND 1 (Thonnyst user_logs'ini)

1. Kui Thonny on avatud, siis pange see kinni (siis tekib viimane logi).
2. Avage Thonny.
3. Valige menüüst Tools.
4. Tools'i alt leiate Open Thonny data folder, mis avab kasutaja Thonny kausta.

5. Avage sellest kaustast user_logs.
6. Sealt valige logid, mille failinimi on seotud antud ülesandega. (Failinimi on kuupäev, mil antud ülesandeid sooritasite.)
7. Tõstke need ühte kausta ning pakkige kokku kas .rar või .zip failiks.
8. Esitage kokkupakitud fail Moodle'i kaudu.

JUHEND 2 (Thonnyst user_logs'ini)

1. Kui Thonny on avatud, siis pange see kinni (siis tekib viimane logi).
2. Avage Thonny.
3. Valige menüüst Tools.
4. Tools'i alt leiate Export usage logs, mis pakib logid zip-failiks kokku.
5. Esitage kokkupakitud fail.

JUHEND 3 (Thonnyt kasutamata user_logs'ini)

1. Kui Thonny on avatud, siis pange see kinni (siis tekib viimane logi).
2. Avage Minu Arvuti/My Computer.
3. Valige sealt kaust, kuhu on installeeritud Windows.
4. Kettalt valige Users kaust, kust leidke kasutaja kaust, kes on kasutanud Thonnyt.
5. Kasutaja kaustas peaks asuma .thonny kaust.
6. .thonny kaust sisaldab kausta User_logs, kuhu on salvestatud Thonny logid..
7. Sealt valige logid, mille failinimi on seotud antud ülesandega. (Failinimi on kuupäev, mil antud ülesandeid sooritasite.)
8. Tõstke need logid ühte kausta ning pakkige kokku kas .rar või .zip failiks.
9. Esitage kokkupakitud fail Moodle' kaudu.

Üks potentsiaalne lahendus antud ülesandele võiks olla selline (joonis 16):

```
def juurdekasv(pindala, aastaneJuurdekasv):
    return round(pindala * 0.4047 * aastaneJuurdekasv, 2)

failinimi = input("Sisestage failinimi: ")
aastaneJuurdekasv = float(input("Sisestage aastane juurdekasv hektari kohta tihumeetrites: "))
piir = float(input("Sisestage piir, mitmest aakrist suuremad metsatükid arvesse võtta: "))

lugeja = 0

with open(failinimi, "r", encoding="UTF-8") as f:
    read = [i.rstrip("\n") for i in f.readlines()]

for rida in read:
    if float(rida) > piir:
        print("Metsatüki aastane juurdekasv on " + str(juurdekasv(float(rida), aastaneJuurdekasv)))
        lugeja += 1
    else:
        print("Metsatükki ei võeta arvesse")

print("Arvutati " + str(lugeja) + " metsatüki juurdekasv.")
```

Joonis 16. Potentsiaalne lahendus

3.2.2 Põhilised lahendusprotsessid

Inimesed on erinevad ja sellest tulenevalt ka nende õppimise stiilid on erinevad. Kuigi ülesande lahendamisega sai hakkama 98.4% õppijatest, kes sellega alustasid, olid nende lahendusprotsessid seinast seina. Järgnevas alapeatükis töötatakse süvitsi läbi 4 põhilist lahendusmeetodit ning üks äärejuhtum, et saada ülevaade põhilistest lahendamise stiilidest. Läbi töötatakse üks hea ja üks halvem näide mõlemast suuremast testijate lahendusstiilist.

Üks selgesti eraldatav õppijate tüüp olid õppijad, kes alustasid lahenduse kirjutamist funktsioonist, kirjutasid selle valmis ja kohe peale seda testisid äsja kirjutatut. Selle tüübi testimismetoodika jagunes kaheks: õppijad, kes kirjutasid lisaks funktsioonile ka faili avamiseks vajalikud käsud ja testisid koodis ning õppijad, kes tegid valmis funktsiooni ja testisid seda käsurealt.

Selle lahendusviisiga käis kaasas ka teiste failide avamine ning siluri kasutamine. Samas peamine miinus oli õppijatel andmetüüpide mittetundmine ning tihtilugu pärast vea saamist läks selle parandamisega kaua aega, kuna ei osatud veateadetega probleemi asukohta määratleda.

Positiivne näide selle grupi esindajatest on järgnev õppija, kelle lahendusaeg oli 32 minutit ning kes tegi kokku 4 viga. Ta alustas koodi kirjutamist faili avamisega ning testis seda rida kohe peale kirjutamist, leides üles oma esimese vea, kus ta oli *open* funktsiooni asemel kasutanud *read* funktsiooni. Vea parandamine käis kähku ning õppija asus enda funktsiooni kirjeldamise juurde. Enne selle juurde asumist avas õppija 3 varem lahendatud ülesannet. Nüüd kirjutas lahendaja funktsiooni valmis ja käivitas programmi. Testimise käigus selgus, et *math* mooduli *round* funktsioon oli valesti imporditud. Lahendaja seejärel muutis veidi koodi, mille käigus test läks läbi, aga viga otseselt ei paranenud.

Järgmise sammuna kirjutati valmis kasutaja sisendi küsimine ning for-tsükli algne loogika. Pärast seda käivitas lahendaja oma programmi ning tegi järeldused, et sisendi küsimine oli korrektne, kuid esialgsesse funktsiooni on tulnud viga taaskord *math* mooduli importimisega. Nüüd sai õppija oma veast aru ning parandas selle. Taaskord testides toimis programm soovitult. Üle jäi ainult viimistleda sisendi küsimise ja väljundi sõnastust ning ülesanne oligi lahendatud.

Samas veidi negatiivsema näitena on sama grupi lahendajatest õppija, kelle lahendus võttis 2 tundi ja 42 minutit ning tegi 34 viga. Ta avas enda lahenduse programmi ja kohe lisas sellele näidisülesande lahenduse. Seejärel kirjutas valmis esimese funktsiooni ja kasutaja

sisendi küsimise. Pärast seda kirjutas rea, mis testis funktsiooni töötamist vastavalt sisendile.

Esimesel testimisel sai faili puudumise tõttu vea, seega tegi endale vajaliku faili ja pärast seda test õnnestus. Järgnevalt kirjutas ta valmis failist andmete lugemise loogika ja alguse osa for-tsüklist, mille tagajärjel esines veateade „AttributeError: 'list' object has no attribute 'split'", kuna õppija üritas kasutada *split* meetodit listi ja mitte sõne peal (joonis 17).

```
list = [f.read()]
list.split(' ')
```

Joonis 17. AttributeError: 'list' object has no attribute 'split'

Probleemi lahendamiseks üritas õppija teha *split* meetodist tuleva vaste sõneks, aga sisemine probleem jäi alles (joonis 18). Seejärel üritas ta faili sisse lugemisest saadud andmetüüpi *split* meetodiga paika saada ning lõpuks lahendusena tegi ta sisse loetud failist saadud informatsiooni sõneks ning seejärel *splitis* sõne õigesti, et saada vajalikud arvud failist kätte.

```
list = [f.read()]
str(list.split(' '))
```

Joonis 18. Probleemi lahenduse katse 1

Pärast selle vea lahendamist tuli veel for-tsüklis viga sõne ja ujukomaarvu võrdlemisest, mille kallal lahendaja pikka aega pead murdis. Nimelt üritas õppija pidevalt *float* funktsiooni sisendiks anda listi. Pärast 15. testikatset avas õppija uue akna, kus ta rahulikult proovis läbi failist andmete sisselugemist. Saadud uute teadmiste põhjal jõudis lahendaja pärast tund aega pusimist probleemi lahenduseni. Seejärel oli õppijal vaja viia saadud tulemus ülesandes soovitud kujule ning ülesanne ära esitada.

Teine eristatav grupp lahendajaid oli vastupidine eelnevale. Õppija sai aru, mida ülesanne temalt nõudis ja seejärel kirjutas valmis suure osa koodist ning hakkas testima alles siis, kui ülesanne peaaegu lahendatud oli. Seejärel aga selgus, et kirjutatud koodis on vead ning hakkas neid veateadete põhjal ära parandama. Tihtilugu võis sellise lahendusega kaasas käia rohkem vigu ja käivitamisi kui pideva testimise metoodikaga.

Nende õppijate peamine probleem oli samuti andmetüüpide vead, kuid lisaks tehti ka palju vigu süntaksiga, kuna jooksvalt ei testitud kõiki kirjutatud koodijuppe. Samas jõudis sellist tüüpi lahendaja kiiremini õige lahenduseni, kuna koodi täielikuks valmis kirjutamiseks oli vaja ülesandest aru saada ja üle keskmise koodikirjutamise oskust.

Positiivne lahendus nende õppijate puhul piirdus tihtilugu ülesande täieliku valmis kirjutamisega ning seejärel ühe korra testimisega või mõne esinenud kirjavea parandamisega. Ühe näitena oleks õppija, kelle lahendus võttis 33 minutit ning tegi 4 viga. Ta alustas kasutajalt sisendi küsimise kirjutamisega. Saades selle valmis kirjutas lahendaja valmis for-tsükli esmase loogika ja kuna seal oli tarvis kasutada funktsiooni, liikus ta poole pealt funktsiooni kirjutamise juurde. Pärast oma idee valmis saamist sai ta aru, et tal oli põhimõtteliselt kogu lahendus kirjas, seega ta viis kogu teksti nõutud kujule ning tõi tsüklisse vajaliku positiivsete arvutuste meelespidamise muutuja.

Esimene kord, kui lahendaja testis oma tehtud koodi, oli pärast kogu koodi valmis kirjutamist. Käivitamisel selgus, et üks muutuja oli jäänud väärtustamata, pärast näpuvea parandamist liikus ta testimisega edasi. Järgmisena selgus, et for-tsüklis oli õppija võrrelnud omavahel sõne ja ujukomaarvu tüüpi muutujaid. Lahendusena viis õppija sõne tüüpi muutuja ujukomaarvu tüübile üle. Kuna ta tegi seda valesti, tuli tal veel 2 vigast käivitust selle vea parandamisel. Pärast selle vea parandamist oligi ülesanne lahendatud.

Samas selle grupi negatiivse poole pealt tuli ette õppijaid, kes kirjutasid suure osa koodist valmis, kuid nüüd tekkinud vigu ei osatud koheselt lahendada. Sellise grupi näiteks on õppija, kelle lahendus võttis 1 tund ja 42 minutit ning ta tegi 30 viga. Lahendusprotsess algas üksteise alla kogu koodi lahti kirjutamisega, aga süntaksit seda tehes ei jälgitud. Koodi kirjutamisel pani õppija oma töö 2 korda kinni ning alles kolmandal korral, kui kogu kood valmis oli, proovis ta seda testida. Kuid nagu varasemalt mainitust võis eeldada, sai lahendaja kohe süntaksivea. Probleemi lahendamiseks kommenteeris õppija vigased kohad välja, kuni programm tööle hakkas. Seejärel proovis ta tööle saada failist lugemist. Töö teeb eriliseks fakt, et õppija ei alustanud funktsiooni tööle saamisest vaid hoopis faili sisse lugemisest.

Pärast failist informatsiooni kättesaamist asus õppija funktsiooni kirjutamise juurde, mille ta ühe hooga valmis kirjutas ning asus kohe for-tsüklit kirjutama, ilma funktsiooni testimata. Pärast peaaegu kogu lahenduse nüüdseks valmis kirjutamist, proovis õppija taaskord testida, kuid seekord sai ta uuesti süntaksivea, mis tuli funktsioonist, kus olid *round* funktsiooni sulud valesti (joonis 19, rida 2). Süntaksivea mõttest õppija hästi aru ei saanud, seega edasiliikumiseks ta kommenteeris lõigu välja.

```
def juurdekasv(metsatüki_pindala, aastane_juurdekasv):
     metsatüki_juurdekasv = round(metsatüki_pindala * 0,4047 * aastane_juurdekasv),2)
     return metsatüki_juurdekasv
```

Joonis 19. Funktsiooni esialgne versioon ilma testimata

Nüüd taaskordsel käivitamisel tuli sama andmetüüpidest mittearusaamisega seotud viga nagu eelnevas töös, nimelt üritati võrrelda listi ja ujukomaarvu tüüpi andmeid (joonis 20, rida 5). Seda viga lahendaja parandama ei hakanud ja selle asemel kopeeris teisest failist uue for-tsükli, mille tulemusena üritas ülesande viimast osa täita. Uuesti avamisel tekkis sama võrdluse viga nagu varem mainitud.

```
fail = open(failinimi, encoding="UTF-8")
for rida in fail:
    metsatüki_pindala += [float(rida)]

if metsatüki_pindala > piir:
    print("Metsatüki aastane juurdekasv on" + str(metsatüki_juurdekasv))
else:
    print("Metsatükki ei võeta arvesse")
```

Joonis 20. Katkine koodijupp

Probleemi lahendamisel tõi õppija esimese proovina if-else lause tsükli sisse, aga viga selle tulemusena ei muutnud. Seejärel kommenteeris õppija vea välja ja liikus edasi. Nüüd programmi jooksutamine õnnestus, ning seejärel lahendaja sulges programmi.

Järgmisel päeval alustas õppija uue tsükli koostamisega, mille tulemusena ta läbis listi metsatüki_pindala element haaval, see liigutas vea edasi funktsiooni väljakutsumiseni. Nagu esimese print-lause lõpust on näha, üritab õppija välja kutsuda funktsioonisisest muutujat ja see ei õnnestu (joonis 19, rida 2; joonis 20, rida 6). Pärast funktsiooni ümbert kommentaarimärkide eemaldamist jõuab ta tagasi eelneva veani, mis oli seotud *round* funktsioni sulgudega. Selle lahendamine võtab paar minutit ja 3 testikatset.

Edasi liigub õppija siluri kasutamise juurde, mille abil ta saab aru, et funktsioon on valesti esile kutsutud, kuid parandamiseks jätab ta siiski funktsiooni parameetriks muutuja metsatüki_juurdekasv. Nüüd ta teeb palju muudatusi, mis teevad uusi vigu juurde ja seejärel õppija sulgeb programmi.

Järgmisel avamisel alustab õppija kõikide muutujate nimede ümbernimetamisega. Kuni pärast paari katset jõuab ta olukorda, kus ainuke viga on taaskord vale funktsiooni väljakutsu-

mise süntaks. Nüüd avab õppija varasemalt tehtud ülesandeid, et oma veast aru saada. Seejärel kasutab ta silurit ja teeb mitmeid muudatusi, kuid funktsiooni väljakutsumist ta esialgu ei muuda. Õppija avab veel rohkem eelnevaid lahendusi ja teeb muudatusi, mis teda edasi ei aita. Pärast tükk aega pusimist jõuab õppija olukorda, kus funktsioonisisese muutuja nimi on sama, mis globaalsel muutujal. Selle tulemusena Thonny enam välja kutsutut veaks ei loe ja näitab õppijale *tuplet*, mis koosneb listist ja ujukomaarvust. *Tuple* sisu on tõenäoliselt seotud mälus oleva informatsiooni asukohaga. Pärast tundi aega siluriga programmi läbitöötamist õppija muudab funktsioonis oleva muutuja nime ning programm hakkab korrektselt tööle. Seejärel korrastab õppija saadud tulemuse ja esitab selle.

Järgnevalt käsitletakse tööd, mis võttis ajaliselt kõige kauem aega. Üritatakse aru saada, kuidas tulevikus võiks õppijate tööle kuluvat aega vähendada ning muuta materjalid kergemini omandatavaks. Kõige pikema ajaga töö autoril kulus lahenduseni jõudmiseks 13 tundi ja 45 minutit, kust on välja arvestatud suuremad pausid. Õppija käivitas programmi 239 korda, millest 140 lõppesid veateatega.

Õppija alustas andemete faili koostamise ja abiülesande avamisega. Koodi kirjutamise osas tegi õppija kõigepealt valmis sisendi küsimisega seonduva osa ning seejärel funktsiooni kirjutamise juurde asudes avas õppija 11 varasemalt lahendatud ülesannet ning kopeerist ühest neist endale funktsiooni sisse for-tsükli, mida ta kasutas failist andmete lugemiseks. Pärast korrastamist sai õppija süntaksivea, mille ta kiirelt lahendas.

Seejärel faili jooksutamine küll õnnestus, aga õppija ei saanud soovitud print käsu vastet. Nüüd kirjutas õppija uue for-tsükli eelnevalt failist listi loetud andmete lugemiseks (joonis 21).

```
def juurdekasv(fail):
    järjend = []
    summa = 0
    for rida in fail:
        järjend += [float(rida)]

     for el in järjend:
        if int(el) < 6:
            summa += int(el)


    return summa
```

Joonis 21. Funktsiooni kirjeldus kahe for-tsükliga

Õppija sai mitu süntaksiviga, mille parandamine õnnestus, kuid pärast vigadeta jooksutamist taaskord soovitud tulemuseni ta ei jõudnud. Nimelt lahendab õppija kogu ülesannet funktsioonis, mida ta kordagi välja ei kutsu. Lahendaja käivitas programmi mitukümmend korda, muutes funktsiooni sisu, selle tulemuseni tuli paar korda süntaksivigu, aga üldiselt midagi ei muutunud. Pärast tükk aega pusimist liikus õppija teise faili, kus ta üritas väikest osa koodist tööle saada.

Esimeses lisafailis üritas õppija kogu ülesande töötsüklit tööle saada, mis tal ei õnnestunud, pärast mitmeid tunde ja mitmekümneid käivitamisi liikus ta teise lisafaili, kus ta üritas tööle saada failist andmete lugemist. Kolmandas lisafailis hakkas ta samuti andmete lugemist tegema, kuni pärast mitmekümneid vigu sellega hakkama sai. Töötava koodi pani ta otse esialgsesse lahendusse, aga midagi ei muutunud, kuna funktsiooni ei ole siiani välja kutsutud.

Viimaks lisas õppija funktsiooni väljakutsumise print lausesse, mille abil kogu eelnev kood tööle hakkas. Nüüd töötas kogu arvutusloogika, aga kokku ei suutnud programm lugeda, mitut suurust töödeldi, kuna funktsioon ei tagastanud ühtegi väärtust.

Sellele probleemile lahenduse leidmiseks tegi õppija kogu arvutusprotsessi katki, kuna ta ei tundnud andmetüüpide erinevusi. Seega ta sai mitukümmend tüübiviga. Nende lahendamiseks üritas ta kõiksugu erinevad muutujad *float* funktsiooni abil arvutüübiks muuta.

Selle tulemusena tekkis vigu peaaegu igalt enda tehtud funktsiooni realt, seega õppija läks tagasi viimasesse töötavasse punkti. Nüüd alustas ta sama protsessi, aga rida haaval, et aru saada, mis töötab ja mis mitte. Selles punktis oli õppija mitu tundi ning kasutas teistest lahendustest saadud koodijuppe, et edasi liikuda.

Pärast tundide viisi erinevaid koodijuppide muudatusi ja liigutamisi, jõudis õppija jälle töötavasse olukorda. Nüüd liikus ta summa kokku arvutamiseni, selleks üritas ta luua uue listi, kuhu sisse lisada kõik töödeldud tulemused. Selle asemel aga üritas ta uuesti arvutuste osa korda saada, korrutades omavahel failist tulnud sõne kujul arvu ja enda lisatud ujukomaarvu tüüpi arve.

Järjekordselt kulus lugematu arv teste, enne kui lahendaja jõudis töötava versioonini, aga seekordne versioon sai hakkama vajalike arvutuste tegemisega ja luges ka kokku vajalike metsatükkide koguse. Seejärel õppija silus oma koodi, lisades funktsiooni uued sisestatavad parameetrid ja testides igal sammul. Selle tulemusena tekkis iga natukese aja tagant andmetüüpidest ja funktsiooni kasutamisest mitte arusaamise tõttu erinevaid vigu.

Lõpuks pärast mitu tundi võtnud silumist jõudis õppija olukorda, kus ta oli valmis töö esitama.

3.2.3 Lahendusprotsesside analüüs logifailide põhjal

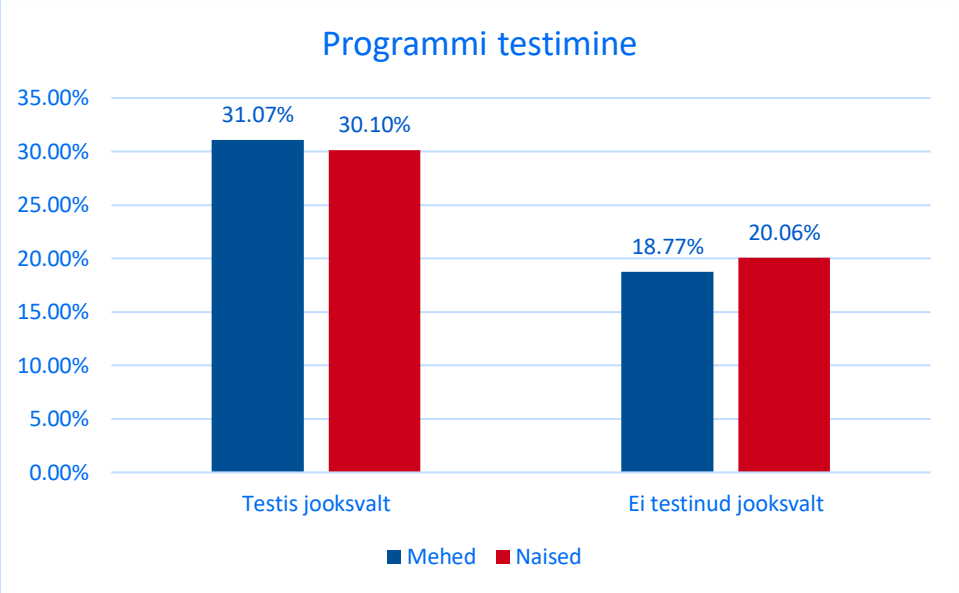
Iga arvestustöö esitanud tudengi lahendusprotsess oli erinev, kuid siiski üldjoontes läheneti ülesande lahenduseni jõudmiseks sarnaselt. Varasemates töödes on peetud üheks oluliseks edasijõudmise tuvastamise faktoriks õppija saadud veateateid [3] [18].

Kokku laekus arvestustöid 798, millest läbi uuriti pooled. Pärast läbi uuritud töödest poolikute tööde ja tühjade tööde eemaldamist jäi järgi 309 logifaili. Nendest 154 olid meeste ja 155 naiste logid. Meeste koostatud tööde lahendusaeg oli keskmiselt 2h 08min samal ajal, kui naiste tööde keskmine aeg oli 2h 20min. Keskmiselt oli veateateid vastavalt 23.9 ja 25 (tabel 1).

Tabel 1. Meeste ja naiste andmed

	Mehi	**Naisi**	**Koos**
Töid	154	155	309
Keskmine vigade arv	23.90	25.00	24.45
Keskmine lahendusaeg	2:07:31	2:20:14	2:13:54
Testis jooksvalt(%)	62.34%	60.00%	61.17%
Kasutas silurit (%)	29.87%	21.94%	25.89%
Sai ülesande lahendatud (%)	99.35%	97.42%	98.38%
Kasutas teist ülesannet (%)	37.66%	38.71%	38.19%
Testimiste vahed olid pikad (%)	44.81%	44.52%	44.66%
Keskmine käivitamiskordade arv	42.60	41.43	42.01
Keskmine veateatega lõppevate käivitamiste arv	22.85	23.40	23.12
Keskmine kleepimiste arv	10.87	11.31	11.09

Bakalaureusetöös valiti peamiseks uurimisfaktoriks hetk, mil õppijad alustasid oma programmi testimisega. Õppijad, kes testisid jooksvalt, alustasid testimist enne for-tsükli kirjutamist, ehk siis funktsiooni või faili sisse lugemise ajal. Kõikidest töödest 31.07% olid jooksvalt testivad mehed ja 30.10% jooksvalt testivad naised. Meeste ja naiste puhul testimise alustamise jagunemine on näha allolevalt jooniselt (joonis 22).



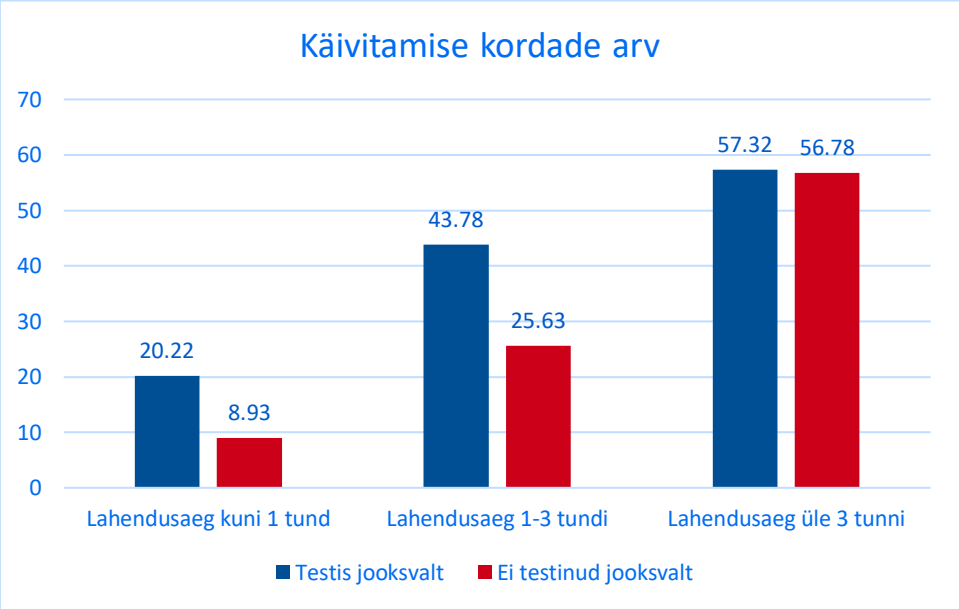
Joonis 22. Programmi testimine

Üldiselt oli keskmine testijate lahendusaeg 2h 51min, mille jooksul tehti keskmiselt 33.36 viga ja käivitati programmi 57.38 korda. Suurema osa valmis kirjutajate keskmine lahendusaeg oli 1h 15min, mille jooksul tehti keskmiselt 10.42 viga ja käivitati programmi 17.81 korral. Varasemalt lahendatud ülesannet kasutas abiks testimisega varem alustanud grupist 51.32% õppijatest, samas õppijate puhul, kes tegid suurema osa koodist valmis, oli abi kasutajaid 17.50%. Siluri kasutamise sagedus oli vastavalt 32.28% ja 15.83% (tabel 2).

Tabel 2. Andmed esimese testimise järgi

	Testis jooksvalt	**Ei testinud jooksvalt**	**Koos**
Töid	189	120	309
Keskmine vigade arv	33.36	10.42	24.45
Keskmine lahendusaeg	2:51:28	1:14:44	2:13:54
Testis jooksvalt(%)	-	-	-
Kasutas silurit (%)	32.28%	15.83%	25.89%
Sai ülesande lahendatud (%)	97.35%	100.00%	98.38%
Kasutas teist ülesannet (%)	51.32%	17.50%	38.19%
Kirjutas suurte juppide kaupa (%)	10.05%	99.17%	44.66%
Keskmine käivitamiskordade arv	57.38	17.81	42.01
Keskmine veateatega lõppevate käivitamiste arv	31.48	9.95	23.12
Keskmine kleepimiste arv	14.56	5.63	11.09

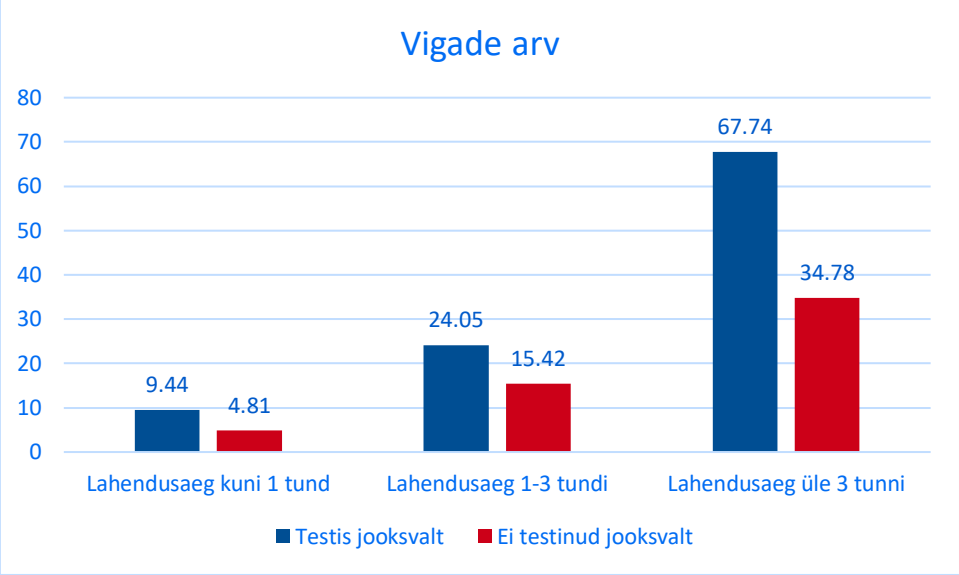
Nende õppijate puhul, kes omandasid MOOC-i jooksul teadmised väga heal tasemel, oli raske eristada lahendusstiile, kuna tihtilugu kulges lahendusprotsess ilma vigadeta ja otseselt vahepealne käivitamine ja testimine ei olnud vajalik. Nende õppijate puhul tehti keskmiselt vähe vigu ja käivitamisi. Leidus õppijaid, kes kirjutasid koodi täies ulatuses valmis ja testisid ainult lõpus, et kas programm töötab või mitte. Seetõttu on ka mitte jooksvalt testivate õppijate, kelle lahendusaeg on vähem kui 1 tund, keskmine käivitamiste arv ainult 8.9 käivitamist ja keskmine vigade arv 4.8. Samas kui testijate grupi keskmine, kes lahendasid ülesande vähem kui tunniga, käivitamiste arv oli 20.2 ja keskmine vigade arv 9.4 (joonis 23, 24, tabel 3). Testijate kõrgemad näitajad on tihtilugu seotud sellega, et tehti algusepoole mõni kirjaviga ja muututi ettevaatlikumaks.



Joonis 23. Keskmine käivitamise kordade arv

Vaadates õppijaid, kes pidid veidi rohkem lahendamisega vaeva nägema, annab saadud tulemuste pilt kahemõttelisi ideid. Tundub, et kui õppija oskas valmis kirjutada suure osa koodist ja lahendamine ei võtnud üleliia kaua aega, pidi tal olema piisavalt hea arusaam koodikirjutamisest, et veateadete abiga kõik tööle saada. Lahendajad, kellel kulus 1 kuni 3 tundi, tegid jooksvalt testides keskmiselt töös 24.05 viga ja käivitasid keskmiselt oma programmi 43.78 korda. Samal ajal kui suurema koodi valmis kirjutajad tegid keskmiselt 15.42 viga ja käivitasid programmi 25.63 korda (joonis 23, 24, tabel 3). Siinse grupi peamised erinevused

võivad tuleneda sellest, et kui õppijal on oskused koodi valmis kirjutamiseks ilma jupphaaval testimata, on tal piisav teoreetiliste teadmiste pagas, mis aitab tal veateadete abil kiiremini lahenduseni jõuda.



Joonis 24. Keskmine vigade arv

Äärejuhtumite korral, kui õppijatel kulus üle 3 tunni ülesande lahendamiseks, muutus andmete kogumine raskemaks. Nende tööde seas oli 57 õppijat, kes testisid kohe algusest peale ja ainult 9 õppijat, kes kirjutasid suurema osa koodist valmis. Samas saadud tulemustest tuli seekord välja, et jooksvalt testijad tegid keskmiselt 67.7 viga, samal ajal kui suurema osa valmis kirjutajad tegid keskmiselt 34.8 viga. Keskmine käivitamiste arv oli testijatel 57.3, samal ajal kui teisel grupil oli 57.8 (joonis 23, 24, tabel 3).

Tabel 3. Andmed testimise ja aja järgi

	Testis jooksvalt <1h	Ei testinud jooksvalt <1h	Testis jooksvalt <3h	Ei testinud jooksvalt <3h	Testis jooksvalt >3h	Ei testi jooksvalt >3
Töid	50	73	82	38	57	9
Keskmine vigade arv	9.44	4.81	24.05	15.42	67.74	34.78
Keskmine lahendusaeg	0:39:54	0:32:57	1:53:27	1:38:33	6:10:19	5:13:00
Kasutas silurit (%)	20.00%	4.11%	32.93%	14.63%	42.11%	5.48%
Sai ülesande lahendatud (%)	100.00%	100.00%	97.56%	46.34%	94.74%	12.33%
Kasutas teist ülesannet (%)	26.00%	6.85%	48.78%	13.41%	77.19%	6.85%

Testimiste vahed olid pikad (%)	22.00%	100.00%	2.44%	45.12%	10.53%	12.33%
Keskmine käivitamiskordade arv	20.22	8.93	43.78	25.63	33.34	56.78
Keskmine veateatega lõppevate käivitamiste arv	8.58	4.60	22.34	14.13	33.34	35.67
Keskmine kleepimiste arv	5.20	3.23	11.55	8.87	34.48	11.33

Uuritud töödes tegid õppijad kokku 7555 viga, mis jagunesid 12 erinvaks tüübiks. Enim puutusid õppijad kokku andmetüübi mittetundmisest tekkinud vigadega, mida esines 89.6% õppijate töödest. Teisel kohal olid süntaksivead, mis esinesid 76.7% õppijate töödest ning kolmandal kohal nimetüübi mittetundmisest tekkinud vead 71.8% õppijate töödest (tabel 4).

Tabel 4. Vigade esinemine töödes

Veatüüp	Esinemissagedus
TypeError	89.64%
SyntaxError	76.70%
NameError	71.84%
ValueError	56.63%
FileNotFoundError	37.86%
AttributeError	15.86%
IndentationError	9.39%
LookupError	2.91%
UnboundLocalError	2.59%
UnicodeDecodeError	1.94%
ZeroDivisionError	1.29%
OSError	0.97%

Kuigi varasemates uurimustes on pigem olnud lühema lahendusajaga õppijad, kes alustasid testimisega varem [3], võivad erinevused olla seotud mitme erineva suurema hulga testimise eripäraga. Esiteks olid testimisega varem alustajaid peaaegu kaks korda rohkem kui pikema osa valmiskirjutajaid. Teiseks viis pikema osa valmiskirjutajate keskmist tulemust alla grupp õppijaid, kes kirjutasid kogu koodi valmis ühe hooga ja seejärel tegid 1–3 testi, et

kirjavead parandada ning esitasid töö valmiskujul. Samuti viis mitte pidevalt testijate tulemust alla see, et õppijad, kes oskasid suurema osa koodist valmis kirjutada, omasid rohkem teadmisi koodi süntaksi kohta ja üldiselt oskasid ette mõelda.

Samas vigade esinemise poolest läksid enim esinenud vigade tüübid kokku varasema uuringuga [15]. Mõlema puhul on enim esinenud vead seotud andmetüüpide mittetundmisega ning selle järgnevad süntaksi ja nimevead.

3.3 Ettepanekud

Järgnevates alapeatükkides tuuakse välja ettepanekud, mis võiksid edasist uurimist ja õpetamist edasi viia. Esimeses alapeatükis tuuakse välja potentsiaalselt tulevikus automatiseeritavad parameetrid. Teises alapeatükis antakse analüüsi tulemuste põhjal soovitusi õppejõududele, et edasi arendada algajate programmeerijate õpetamise protsessi.

3.3.1 Soovitused automatiseerimiseks

Töö käigus on kasutatud varem loodud programme, et automatiseerida võimalikult suur osa tehtavast tööst, samas siiski on vaja iga logi algusest lõpuni läbi käia. Et vähendada uurimise käigus tehtavat töö hulka, tuleks automatiseerida enamate parameetrite leidmine.

Üks tulevikus potentsiaalselt automatiseeritav parameeter oleks esimese testimise aeg võrdluses ülesande alustamise ajaga. Selle jaoks oleks vaja leida esimese kasutaja käivitamise aeg ning sellest lahutada õppija ülesandega alustamise aeg. Saadud vahe põhjal saaks teada, millal toimus esimene testimine. Lisaks oleks võimalik automatiseerida siluriga seonduva informatsiooni otsing. Nimelt võiks programm aru saada igast korrast, mil õppija silurit kasutas, ja need eraldi üles märkida. Parameeter, mida praeguses töös ei uuritud, aga tulevikus oleks võrdlemisi kerge uurida ja ka automatiseerida oleks avatud programmide hulk, kuna leidus töid, kus avati kümneid erinevaid faile.

3.3.2 Soovitused õppejõududele

Bakalaureusetöö tulemusena selgus, et enim esinevad vead tulenevad andmetüüpide ja süntaksi mittetundmisest. Kui nende vigade esinemist vähendada, oleks võimalik ka keskmist algaja programmeerija tööle kuluvat aega vähendada.

Üks meetod juhul, kui on kursus praktikumidega, mis võiks aidata algajatel õppijatel oma vigadest paremini aru saada, oleks see, et iga tudeng peab klassi ees ühe ülesande ekraani peal lahendama. Protsessi käigus peaks õppija seletama lahenduskäiku ning jooksvalt võivad kõik klassis viibijad esitada küsimusi. Nii oleks võimalik igaühel aru saada, kus on

nende teadmistes lüngad. Selle protsessi tulemusena võiks areneda õppijate arusaam andmetüüpidest ja süntaksist.

Veidi parem versioon eelmisest oleks, kui iga õppija peaks lühiülesande lahendama õppejõu või mõne programmeerija ees, kus samamoodi tuleks iga koodijupp lahti seletada, nii oleks tagasiside personaalsem.

Logide poole pealt tuleks õppejõududel kontrollida juba jooksvalt iga õppija lahenduskäiku ja üritada lahti saada vigadest, mida õppija teeb. Seda saaks teha näiteks tunnikontrollide puhul. Logidest on võimalik juba varakult kätte saada vajalik informatsioon, et läheneda õppijatele individuaalselt. Eraldi võiks teha konsultatsiooni õppijatele, kelle logidest väljendub, et nende lahendusprotsess kvalifitseerub äärejuhtumiks, näiteks kulub liiga palju aega või teeb liiga palju vigu võrreldes teistega.

Kokkuvõte

Programmeerimise õppimisele on igal inimesel erinev lähenemine ning samal ajal igal õpetajal erinev õpetamise metoodika. Mida aeg edasi, seda populaarsemaks on muutunud erinevad internetipõhised MOOC-i stiilis kursused programmeerimise algteadmiste õpetamiseks. Siiski on õppejõul õppimise protsessi ainult esitatud tööde põhjal raske jälgida.

Bakalaureusetöö eesmärk oli 2018. aasta MOOC-i „Programmeerimise alused“ logifailide põhjal selgitada õppijate peamisi lahendusprotsesse ning võrrelda nende õppijate lahendusprotsesse, kes testisid programmi jooksvalt ning kes seda ei teinud.

Töö esimeses osas tutvustati uurimistöö raamistust. Lisaks tutvustati varasemaid uurimusi, mis käsitlevad õppijate lahendusprotsesse. Teises osas kirjutati detailselt Thonny logide uurimise protsessist. Logidest saadi suur osa informatsioonist automaatselt, kasutades varem tehtud programmi, kuid oli ka mitmeid parameetreid, mis tuli leida käsitsi. Läbi töötati 490 tööd, järeldusi tehti 309 logi põhjal. Kolmandas osas analüüsiti informatsiooni, mis nendes logides leidus. Esmalt selgitati, mis kujul Thonny logides andmeid salvestab ning seejärel tutvustati õppijate lahendusprotsesse. Kuna suur osa õppijatest lähenes lahendamisele üldjoontes sarnaselt, siis tutvustati mitut enim nähtud lahendusstiili. Lõpuks analüüsiti saadud tulemusi.

Bakalaureusetöö käigus oli põhirõhk lahenduskäigu testimise alustamise ajal, ehk kas õppija alustas testimist kohe algusest või mitte. Testimisega alustas enne tsüklit 189 õppijat ning hiljem 120, samas tööle kuluv aeg oli keskmiselt vastavalt 2 tundi 51 minutit ja 1 tund 15 minutit. Sellist vahet võib selgitada mitmete asjaoludega: äärejuhtumid, kus õppija teeb töö valmis ning testib 1 korra; õppijad, kes testivad pidevalt, aga ei oska veateadetega midagi peale hakata; õppijad, kes oskavad koodist suure osa valmis kirjutada, oskavad ka oma vigu kiiremini parandada.

Töö käigus tutvustati süvitsi ka nelja õppija lahendusi. Näidistöödeks valiti tööd, kus oli üks hea ja üks halvem näide mõlema suurema testijate grupi kohta. Näidetes olid näha õppijates enim probleeme tekitanud arvestustöö osad.

Arvestustööde lahendamise käigus esines 12 erinevat veatüüpi, millest kõige populaarsemateks osutusid andmetüüpide mittetundmisest tulenevad tüübivead, millega puutus kokku 89.64% lahendajatest. Sellele järgnesid esinemissageduselt süntaksivead ja nimevead. Samade vigadega puututi enim kokku ka näidislahendustes.

Bakalaureusetöö tulemusi võib kasutada tulevates uurimistes, et edasi arendada logifailidest saadava informatsiooni automatiseerimist. Kaks peamist parameetrit, mida võiks veel automatiseerida, on esimese testimise aeg ning siluri kasutamise kordade arv. Lisaks on võimalik bakalaureusetöö tulemuste põhjal õppejõududel teha muutusi õpetamise protsessis. Kuna enamik vigu on seotud andmetüüpide või süntaksi mittetundmisega, oleks abiks, kui iga õppija peaks vähemalt korra enda loodud programmi töö lahti seletama.

Viidatud kirjandus

[1] Tartu Ülikooli sisseastumise statistika: https://www.ut.ee/et/sisseastumine/statistika-0. (Kasutatud 07.05.2020)

[2] Tartu Ülikooli arvutiteaduste instituudi MOOC-ide statistika: https://progmooc.cs.ut.ee/moocid/statistika/ (Kasutatud 07.05.2020)

[3] H. Meier, „Õppijate käitumismustrid programmeerimisülesande lahendamisel: logifailide analüüs," Tartu Ülikool, Tartu, 2018. https://dspace.ut.ee/handle/10062/66149 (Kasutatud 07.05.2020)

[4] K. Pedel, „E-kursuse "Programmeerimise alused" logifailide analüüs," Tartu Ülikool, Tartu, 2016. https://dspace.ut.ee/handle/10062/56240 (Kasutatud 07.05.2020)

[5] A. Roosi, „Thonny logifailide analüüsi automatiseerimine," Tartu Ülikool, Tartu, 2019. https://dspace.ut.ee/handle/10062/66233 (Kasutatud 07.05.2020)

[6] V. Tõnisson, „Programmeerimisülesannete lahendamisel tekkivate Thonny logifailide korrastamine," Tartu Ülikool, Tartu, 2019. https://dspace.ut.ee/handle/10062/66244 (Kasutatud 07.05.2020)

[7] A. Vihavainen, M. Luukkainen ja P. Ihantola, „Analysis of Source Code Snapshot Granularity Levels," SIGITE '14: Proceedings of the 15th Annual Conference on Information technology education, Atlanta, 2014. https://dl.acm.org/doi/10.1145/2656450.2656473 (Kasutatud 07.05.2020)

[8] S. Powell ja L. Yuan, „MOOCs and open education: Implications for higher education," 2013. https://www.researchgate.net/publication/265297666_MOOCs_and_Open_Education_Implications_for_Higher_Education (Kasutatud 07.05.2020)

[9] T. Ristolainen, L. Pilt ja M. Lukas, „MOOCid – hääbuv haip või jätkusuutlik initsiatiiv?," Koolielu, 2016. https://koolielu.ee/info/readnews/505742/moocid-%E2%80%93-haabuv-haip-voi-jatkusuutlik-initsiatiiv (Kasutatud 07.05.2020)

[10] M. Lepp, P. Luik, T. Palts, K. Papli, R. Suviste, M. Säde ja E. Tõnisson, „MOOC in Programming: A Success Story," Proceedings of the International Conference on e-Learning, pp. 138-147, 2017.

[11] Programmeerimise alused 2018/19 registreerimine: https://www.ut.ee/et/mooc/programmeerimise-alused?language=et (Kasutatud 07.05.2020)

[12] MOOC Programmeerimise alused 2016/17: https://courses.cs.ut.ee/2017/eprogalused/ (Kasutatud 07.05.2020).

[13] A. Annamaa, „Thonny, a Python IDE for Learning Programming," Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, pp 343, 2015. https://dl.acm.org/doi/10.1145/2729094.2754849 (Kasutatud 07.05.2020)

[14] A. Annamaa, „Introducing Thonny, a Python IDE for learning programming," Proceedings of the 15th Koli Calling Conference on Computing Education Research, pp 117-121, 2015. https://dl.acm.org/doi/10.1145/2828959.2828969 (Kasutatud 07.05.2020)

[15] R. Kodasmaa, „Programmeerimiskeele Python veateated programmeerimise,“ Tartu Ülikool, Tartu, 2017. https://dspace.ut.ee/handle/10062/65819 (Kasutatud 07.05.2020)

[16] D. Perkins, C. Hancock, R. Hobbs, F. Martin ja R. Simmons, „Conditions of Learning in Novice Programmers,“ Journal of Education Computing Research, pp. 37-55, 1986. https://journals.sagepub.com/doi/abs/10.2190/GUJT-JCBJ-Q6QU-Q9PL?journalCode=jeca (Kasutatud 07.05.2020)

[17] G. Marceau, K. Fisler ja S. Krishnamurthi, „Measuring the effectiveness of error messages designed for novice programmers,“ Proceedings of the 42nd ACM technical symposium on Computer science education, pp. 499-504, 2011. https://dl.acm.org/doi/10.1145/1953163.1953308 (Kasutatud 07.05.2020)

[18] M. C. Jadud, „Methods and tools for exploring novice compilation behaviour in BlueJ,“ University of Kent, Kent, 2006. https://dl.acm.org/doi/10.1145/1151588.1151600 (Kasutatud 07.05.2020)

Lisad

I. Andmete tabel

Kõik läbitöötatud andmed on leitavad järgnevalt aadressilt:

https://docs.google.com/spreadsheets/d/1dfTbE88Csg1YdJOhia1apgKu5JhwPyhdpwpcK7ZXCHs/edit?usp=sharing

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Sten Vaher**

annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose
Õppijate lahendusprotsesside analüüsimine Thonny logifailide põhjal,
mille juhendaja on Heidi Meier,
reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

1. 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.

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

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

Sten Vaher

08.05.2020