TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kristjan Stüff 213662 IABB

Tehisintellekti integreerimine tarkvaraarendusprotsessi olemasoleva mahuka koodibaasi põhjal

Bakalaureusetöö

Juhendaja: Tauno Treier
Magistrikraad

Tallinn 2024

Autorideklaratsioon

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

Autor: Kristjan Stüff

20.05.2024

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on integreerida tehisintellektil baseeruvat tööriista tarkvaraarendusprotsessi olemasoleva mahuka koodibaasi põhjal. Tööriista rakendamise tulemusel suudaksid ettevõttesse tulevad uued töötajad sisseelamisperioodil iseseisvamalt ning efektiivsemalt töötada.

Töö käigus antakse ülevaade läbiviidud uuringutest tehisintellekti rakendamisest arendusprotsessis ning eksisteerivatest tööriistadest. Seejärel püstitatakse nõuded, millele otsitav tööriist peab vastama. Peale tööriista valikut viiakse läbi praktiline eksperiment kasutades tööriista, kus lahendatakse tööriista abil ettevõttes esinenud ülesandeid.

Analüüsi osas hinnatakse põhjalikumalt tööriista vastavust nõuetele. Seejärel analüüsitakse abivahendi poolt genereeritud koodi ning võrreldakse seda inimarendajate lahendusega. Analüüs näitab, et tehisintellekti kasutamine tarkvaraarenduses on juba praegu otstarbekas ning aitab kogenematutel arendajatel olla iseseisvam. Küll aga on arenguruumi kogu koodibaasi sügavamal mõistmisel ning koodi taaskasutamisel.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 23 leheküljel, 6 peatükki, 6 joonist.

Abstract

Integrating artificial intelligence into the software development process using a large existing codebase

This bachelor's thesis explores the application of an AI-based tool in the software development process, focusing on understanding and reusing existing code within a large codebase. The main target group is developers who are new to the company and not yet familiar with the codebase. The thesis aims to evaluate the success of implementing such a tool by setting requirements to validate the results.

The author analyses previous application of AI in software development by examining its ability to solve problems, comparing its solutions to human-generated solutions, and assessing its capability to write clean and stylish code using existing literature. Then the author establishes requirements for the AI tool that will be used in the next part of the thesis.

To evaluate the capabilities of the selected tool, Bito, the author conducts a practical experiment using real-world software development tasks. After the experiment the author analyses and validates the code from Bito by comparing it to the code written by human developers. The results show that Bito can provide correct solutions and improve developer autonomy but needs further development in understanding and reusing the existing codebase.

In conclusion, this thesis provides an overview of the potential of AI in software development, emphasizing the need for further research to improve AI tools and integrate them into existing development processes.

The thesis is in Estonian and contains 23 pages of text, 6 chapters, 6 figures.

Lühendite ja mõistete sõnastik

AI	*Artificial intelligence* – tehisintellekt
API	*Application programming interface* – rakendusliides
IDE	*Integrated development environment* – integreeritud arenduskeskkond
Idioom	Üldtuntud ning kokkulepitud viisid kuidas probleemi programmeerimiskeeles lahendada. Näiteks kuidas *Python*'is kollektsiooni läbi itereeritakse
HTTPS	*Hypertext Transfer Protocol Secure* – hüperteksti edastamise protokoll lisatud turvalisusega. Kasutab krüpteerimist seega on turvalisem kui *HTTP*.

Sisukord

Jooniste loetelu

1 Sissejuhatus

Tehisintellekti areng on antud tööle eelnenud aastate jooksul olnud väga kiire. Muutused tehisaru võimekuses ning funktsionaalsuses on toimunud hüppeliselt ja ootamatult ka IT valdkonna spetsialistidele. Tarkvaraarenduses on hakanud paljud arendajad kasutama populaarseks saanud juturoboteid, millest tuntuim on *OpenAI* poolt tehtud *ChatGPT*. Kasutamise põhimõte on lihtne, tuleb kirjeldada juturobotile oma sõnadega lahendatavat probleemi ning robot suudab vastuseks koodi genereerida. Säärane tööviis ei pruugi olla efektiivne, kui seda kasutada suurettevõtetes ning mahuka koodibaasi põhjal.

1.1 Probleem

On loomulik protsess, et aja jooksul vahetuvad ettevõttes töötajad, sealhulgas ka tarkvaraarendajad. Kui tegemist on tarkvaraga, mida on juba pikemat aega arendatud, võib koodibaas olla väga mahukas ning uuele töötajale keeruline hoomata ja mõista. Täpsem probleem tekib kui uus arendaja hakkab tööülesandeid täitma. Uut funktsionaalsust lisades on väga suur tõenäosus, et midagi väga sarnast on juba varasemalt koodibaasis implementeeritud, kuid seda uus töötaja ei tea. Mõni kogenum arendaja teaks kohe, kust enam-vähem sarnast koodiblokki otsida, kuid uuel töötajal sellist teadmist koheselt ei ole. Kasutades interneti abi saab tõenäoliselt funktsionaalsed nõuded täidetud, kuid siit tekib teine probleem: kas säilis ka koodi stiil? Alguskuudel kulub uutel arendajatel kõige rohkem aega just olemasoleva koodibaasi ja stiiliga tutvumiseks ning mõistmiseks, millist tüüpi lahendusi tuleks kasutada. Kui eksisteeriva koodiga piisavalt tuttav ei olda, kirjutatakse uut koodi, mis ei ole samasuguse stiiliga nagu olemasolev ning võib tekkida olukord, kus ühte sama asja, näiteks tabeli kuvamist, on implementeeritud mitmes erinevas kohas täiesti eri moodi. Et koodi stiili säilitada ning teada saada, kus sarnaseid asju varem tehtud on, toetuvad uued töötajad kogenumatele kolleegidele, küsides nendelt abi. See aga tähendab, et kolleegidel on vähem aega tegeleda enda tööülesannetega ning kokkuvõttes meeskonna produktiivsus uue töötaja sisseelamisperioodil langeb.

1.2 Eesmärk

Antud bakalaureusetöö eesmärk on rakendada tehisintellektil põhinevat tööriista tarkvaraarendusprotsessis olemasoleva mahuka koodibaasi põhjal ning hinnata rakendamise edukust. Peamiseks sihtgrupiks on arendajad, kes on just ettevõttesse tulnud ning pole veel koodibaasiga tuttavad. Tööriista valikuks ning hindamiseks püstitab autor nõuded, mis aitavad tulemust valideerida.

Peamise eesmärgi saavutamiseks on töö autor püstitanud järgmised alameesmärgid:

1. Uurida ning kaardistada olemasolevad tööriistad, nende sarnasused ja erinevused.
2. Koostada nõuded, millele tööriist vastama peab.
3. Analüüsida olemasolevaid tööriistu põhjalikumalt ning valida nende seast sobivaim, mida põhjalikumalt kasutada.
4. Analüüsida valitud tööriista kasutusvõimalusi, hinnata kasutegurit ja vastavust nõuetele.
5. Viia läbi praktiline eksperiment, hindamaks tööriista võimekust ning sobivust.

1.3 Töö struktuur

Antud bakalaureusetöö koosneb 6 peatükist. Esimeses peatükis tuuakse välja probleem ning töö eesmärk. Teises peatükis antakse ülevaade autori poolt läbi töötatud kirjandusest, kus on uuritud *AI* ehk *Artificial Intelligence* rakendamist tarkvaraarendusprotsessis ning analüüsitakse selle sobivust/ebasobivust ja õnnestumist/ebaõnnestumist. Kolmandas peatükis koostatakse nõuded, millele valitav tööriist peab vastama ning seejärel selgitatakse täpsemalt tööriistade valikuprotsessi ja lõpliku otsuse tagamaid. Neljandas peatükis viiakse läbi praktiline eksperiment, et testida tööriista võimekust ettevõttes esinenud probleeme lahendada. Bakalaureusetöö viiendas peatükis analüüsitakse saadud tulemusi ja valitud tööriista. Lisaks sellele arutletakse otsuste üle, kus oleks saanud kasutada erinevaid alternatiive ning töö võimalikke edasiarendusi. Kuues peatükk on kokkuvõttev peatükk ning seejärel tuuakse välja kasutatud kirjandus ja lisad.

2 Tehisintellekti rakendamine tarkvaraarendusprotsessis

Järgnevas peatükis antakse ülevaade varasemalt läbiviidud uuringutest ja eksperimentidest, kus rakendati või uuriti tehisintellekti tarkvaraarendusprotsessis. Eesmärk on aru saada, mida on juba varasemalt katsetatud, kas see on õnnestunud ning kui on, siis millisel määral. Lisaks annab järgmine peatükk vastuse küsimusele, kas paljude arvamus, et *AI* asendab tarkvaraarendajaid, vastab tõele või mitte.

2.1 Võimekus lahendada probleeme

Nagu varasemalt mainitud on levinud arusaam, et tehisintellekt asendab tarkvaraarendajad. Arvamuse populaarsust kinnitavad mitmed uuringud sarnase pealkirjaga, millega autor antud bakalaeureusetöö raames on tutvunud. Arendajate asendamiseks peab aga *AI* suutma lahendada probleeme iseseisvalt ning genereerima toimivat ja korrektset koodi.

Läbiviidud uuringutest selgub, et lihtsate ning keskmise keerukusega ülesannetega suudavad *AI*'l põhinevad tööriistad juba üsnagi hästi hakkama saada, kohati ka ületada päris arendajate poolt loodud vastuseid. Näiteks Nikolaidis jt. [1] leidsid enda katsetustes, et nii *ChatGPT* kui ka *Github Copilot* suutsid lahendada 100 % lihtsatest ülesannetest korrektselt. Vahe tuli aga sisse keskmise tasemega ülesannete juures, kus *ChatGPT* lahendas 47,4 % ning *Copilot* 68,4 % etteantud ülesannetest. Sarnane vahe säilis ka raskete ülesannetega, kus *ChatGPT* õnnestumisprotsent oli 47,6% ning *Copilot*'il 71,4 %. Küll aga tuleb mainida, et oluline erinevus tuli mängu programmeerimiskeelte vahel: *Java* ning *Python* said tunduvalt rohkem korrektseid lahendusi võrreldes *C*'ga. Sarnase tulemuseni jõudsid ka Ekedahl ja Helander [2], kes märkisid, et madala ning keskmise keerukustasemega ülesandeid suutis *ChatGPT* lahendada üsnagi edukalt, kuid kõrge keerukustasemega ülesannete juures esines raskusi. Lisaks leidsid ka Kuhail jt. [3], et edukuse protsent langes, mida keerukamaks ülesanded muutusid ning samale järeldusele jõudsid ka Tian jt. [4]. Corso jt. [5] tulemused näitasid, et *AI*'l põhinevate tööriistade võime ülesandeid lahendada võiks olla palju parem. Parimate tulemustega *Github Coilot* suutis lahendada 47/100 probleemist, *ChatGPT* 34/100, *Google Bard* 23/100 ning *Tabnine* 20/100. Suurem puudus tekkis siis kui testiti välisest koodist sõltuvaid

probleeme, kus parima tulemuse pakkunud *Copilot* lahendas vaid 15/52 probleemist. Nguyen ja Nadi [6] uurisid samuti *Copilot*'i võimet programmeerimisülesandeid lahendada. Uuriti nelja erinevat programmeerimiskeelt ning igale keelele esitati 33 ülesannet. Kokku lahendas *Copilot* korrektselt 55 ülesannet 132-st ehk õnnestumisprotsent oli 41,7 %. Antud uuringus võrreldi järgmiseid programmeerimiskeeli: *Java*, *Python*, *C* ning *Javascript*, ja 33-st ülesandest oli korrektseid lahendusi vastavalt 19, 14, 13 ja 9. Kõige kõrgem korrektsuse määr oli *Java*'l, millele järgnesid *Python*, *C* ning *Javascript*.

Kõikides eelmainitud uuringutes kasutati programmeerimisprobleemide saamiseks platvormi *Leetcode*, mis on väga populaarne veebileht, kus on üle 2000 intervjuuküsimuse, mida on kasutatud ettevõtete poolt tarkvaraarendajate intervjuudel [7]. Keskkond on sääraseks uurimiseks sobilik just seetõttu, et sinna on palju laadinud enda lahendusi ka eri oskustasemel arendajad. Seega on hea võrrelda tehisintellekti pakutud vastuseid inimeste vastustega, et aru saada, kas *AI* suudab tõesti olla efektiivsem ning lahendada probleeme paremini kui inimesed.

Teistsuguseid andmeid kasutasid Yetistiren jt. [8], kes andsid tehisintellektile ette *HumanEval* andmekogumi, mis koosnes 164 probleemist. Iga probleem omas ülesannet, eeldatavat lahendust, mida võib võtta kui inimese poolt tehtud lahendust ning ühikteste lahenduse kontrollimiseks. Nende tulemustest selgus, et 79,1 % probleemidest suutis *Github Copilot* osaliselt ära lahendada, mis tähendab, et vähemalt üks etteantud testidest läks läbi. See 79,1 % sisaldas ka 28,7 % kogu ülesannetest, mille lahendused läbisid kõik testid.

2.2 Lahenduste võrdlus inimlahendustega

Eelnevale peatükile toetudes on võimalik väita, et *AI*'l baseeruvad tööriistad suudavad iseseisvalt ülesandeid lahendades toota korrektset ja toimivat koodi. Mida kergem on ülesanne, seda tõenäolisem on ka korrektne lahendus. Kui aga võrrelda saadud vastuseid tarkvaraarendajate poolt tehtud lahendustega, siis kas *AI* lahendused on efektiivsemad ja paremad?

Kuhail jt. [3] leidsid, et nende poolt läbiviidud testides suutis *ChatGPT* 58,5 % kordadest genereerida inimlahendustest ajaefektiivsemat koodi ning 69,6 % kordadest mäluefektiivsemat koodi. Ekedahl ja Helander [2] leidsid, et madala ning kesmise

keerukusega ülesannete puhul on tehisintellekt võimeline genereerima inimestest nii aja- kui ka mälukasutuse mõttes efektiivsemat koodi, kuid raskete ülesannete puhul ei olnud vahe inimprogrammeerijatega enam märkimisväärne. Raskete ülesannete puhul oli koodi efektiivsus sama, mis inimestel. Sarnase tulemuseni jõudsid ka Nascimento jt. [9], kelle uuringud näitasid, et kergete ja keskmise keerukusega ülesannete puhul suutis *ChatGPT* genereerida efektiivsema lahenduse, kui algtasemel programmeerijad. Küll aga ei suutnud tehisintellekt ületada kogenud arendajate lahendusi. Lisaks sellele ei suutnud *ChatGPT* lahendada rasket ülesannet, mida kogenud arendajad suutsid. Tian jt. [4] uuringutest selgus, et nii *ChatGPT* kui ka *Cordex* tööriistade poolt genereeritud kood oli keskmiselt efektiivsem kui inimeste poolt esitatud lahenduste kood. Samuti märkasid nad, et ülesannete keerukuse kasvades lahenduste efektiivsus langes. Corso jt. [5] leidsid, et *AI* poolt genereeritud kood oli kas samal tasemel inimarendajatega või natuke efektiivsem.

Nagu eelpool mainitud kasutasid Yetistiren jt. [8] teistsuguseid algandmeid ning koodi ajakeerukust uurides leidsid autorid, et korrektselt lahendatud probleemidest 87,2 % olid *Github Copilot'*i poolt pakutud lahenduste koodi efektiivsus samal tasemel inimlahendustega ning 12,8 % olid kõrgema efektiivsusega. Osaliselt korrektsete lahenduste hulgast olid 76,2 % sama efektiivsusega, mis inimlahendused ning 3,6 % olid madalama efektiivsusega.

2.3 Võimekus kirjutada puhast koodi ning järgida stiili

Varasemalt läbiviidud uuringud tõestavad, et tehisintellekt suudab efektiivselt lahendada talle esitatud ülesandeid, kuid see pole tarkvaraarenduses ainus oluline aspekt. Oluline on ka koodi puhtus ning stiil. Kas *AI*'l põhinevad tööriistad suudavad neile püstitatud ülesandeid lahendades säilitades koodi puhtust ning stiili?

Pudari ja Ernst [10] kasutasid oma uuringutes *Github Copilot* tööriista ning hindasid selle võimekust genereerida kvaliteetset ning puhast koodi. *Copilot*'i soovitusi võrreldi 25 *Python* programmeerimiskeele idioomi ning *AirBNB* 25 üldtuntud *JavaScript* praktikaga. *Python*'i idioomidega võrreldes soovitas *Copilot* esimesena korrektse variandi vaid kahel korral 25-st. Kaheksal korral allesjäänud 23-st oli oodatud vastus *Copilot*'i 10 soovituse seas, kuid 15-ne idioomi puhul ei olnud soovitud lahendusviisi pakutud vastuste seas. *JavaScript*'i tulemused olid samuti nõrgad. Esimesena pakkus *Copilot* korrektset lahendust vaid kolmel korral, viiel korral oli oodatav vastus 10 pakutava vastuse seas ning

17 korral ei pakkunud *Copilot* oodatud vastust üldse. Nagu Pudari ja Ernst [10] ka ise mainisid, on nende tulemuste põhjal näha, et sel ajahetkel (2023) ei olnud *Copilot* veel võimeline genereerima koodi „parimate praktikate“ järgi, mis võib kokkuvõttes koodi kvaliteedi taset alla viia.

Ka Corso jt. [5] jõudsid sarnasele järeldusele. Nemad võrdlesid *AI* poolt genereeritud koodi inimeste koodiga ning analüüsisid selle kvaliteeti. Selgus, et korrektsete lahenduste puhul suutis stiili kõige paremini säilitada *Tabnine*, millele järgnesid *Copilot*, *ChatGPT* ning viimasena *Bard*. Osaliselt korrektsete ning valede lahenduste puhul ei olnud tööriistade vahel statistiliselt olulisi erinevusi ja kõigi tulemused olid tunduvalt kehvemad kui korrektsete lahenduste puhul. Küll aga märkisid Corso jt. [5], et ka parima tulemusega *Tabnine*'i puhul pidi koodis tegema muudatusi, et see täielikult kattuks projekti stiiliga. Kokkuvõtlikult hindasid autorid, et *AI*'l põhinevatel tööriistadel on palju arenguruumi koodi stiili säilitamise osas.

Tehisintellekti poolt genereeritud koodi kvaliteeti uurisid ka Catir ja Claesson [11], kes kasutasid *Copilot* ja *Codepal* tööriistasid. Koodi kvaliteeti hinnates leidsid nad, et pidevalt kasutati häid ja selgeid muutujate ning meetodite nimetusi. Teine aspekt, mida autorid uurisid oli kommentaaride arv. Kommentaaride kogus varieerus palju nii programmeerimiskeelte kui ka ülesannete tüüpide vahel. *Python*'i ja *Java* keelses koodis oli kommentaare rohkem kui *C*'s ning lühemate ja lihtsamate ülesannete vastustes oli samuti vähem kommentaare. Kokkuvõttes järeldasid autorid, et *AI* poolt genereeritud kood oli väga sarnane inimese poolt kirjutatud koodile ning seetõttu on seda sama lihtne lugeda, mõista ning muuta kui inimeste poolt kirjutatud koodi.

Tehisintellektil põhinevate abivahendite koodi kvaliteeti analüüsis ka Kantek [12], kes kasutas kvaliteedi hindamiseks *SonarCloud* tarkvara. Autor võrdles omavahel *Github Copilot*, *Tabnine*, *ChatGPT* ja *CodeGeex* tööriistu. Keerukuse hinnang oli kõige madalam *Copilot*'il, teisel kohal oli *Tabnine*, seejärel *CodeGeex* ning kõige keerukamat koodi genereeris *ChatGPT*. Tarkvaravigade ehk *bug*'ide arvu võrreldes oli neid kõige rohkem *ChatGPT* koodis (78), järgmine oli *Copilot* (51) ning kõige vähem vigu oli *CodeGeex* (40) ja *Tabnine* (26) genereeritud koodides. Viimasena analüüsis Kantek [12] *code smell*'ide olemasolu. *Code smell* on üldjuhul lihtsasti märgatav osa koodist, mis võib viidata sügavamale probleemile [13]. Hea näide oleks pikk meetod, millele peale vaadates tekib kohe mõte, et see võiks olla lühem ning ehk saaks kuskil asju ümber teha, kuid tegeliku vastuseni jõudmiseks tuleb koodi süveneda. Kantek [12] leidis, et kõige

lõhnavamat koodi pakkus *Tabnine*, seejärel *ChatGPT*, *CodeGeex* ning kõige vähem *code smell*'e sisaldas *Copilot*'i kood. Kokkuvõttes olid kõik genereeritud programmid *SonarCloud*'i hindamiskategooriate kehvemas otsas. Tarkvaravigade parandamiseks eeldas *SonarCloud* ajakulu 2 päeva ja 4 tundi ning *code smell*'ide eemaldamiseks 9 päeva ning 3 tundi. Kokkuvõttes saab antud tulemustest järeldada, et AI poolt genereeritud kood ei ole koheselt kasutatav ning seda tuleb refaktoreerida, et saavutada puhas koodibaas.

Autori teada ei ole töö kirjutamise hetkeks veel *AI* kasutamisest olemasolevas mahukas koodibaasis ning selle võimest genereerida samasuguse stiiliga koodi uuringuid tehtud. Seetõttu on tegemist aktuaalse teemaga, mida antud bakalaureusetöö raames käsitletakse.

3 Tööriista valik

Selles peatükis koostab autor nõuded valitavale tööriistale toetudes tutvutud kirjandusele, juhendajaga läbiviidud intervjuule ning enda ekspertiisile. Peale seda antakse lühiülevaade kaalukausil olnud tööriistadest ning lähtudes püstitatud nõuetest põhjendatakse, miks osutus lõplik valik just selliseks.

3.1 Nõuded

Selleks, et mõista tarkvaraarendajate üldiseid ootuseid AI tööriistadele uuris autor antud teemal olemasolevat kirjandust. Liang jt. [14] viisid tarkvaraarendajate seas läbi küsitluse, uurides nendelt, kuidas nad on *AI* programmeerimisabilisi kasutanud ning millest nad neid kasutades puudust tundnud on. Tulemustest selgus, et kõige populaarsem kasutus oli korduva või väga lihtsa koodi genereerimine. Selle alla liigituvad näiteks andmete määramine objektidele, *API* ehk *application programming interface* lõpp-punktide *CRUD* (*Create, Read, Update, Delete*) operatsioonide sisu genereerimine, sorteerimisalgoritmid ja muu sarnane. Abiliste parendusena toodi välja kogu koodi konteksti parem mõistmine, loomulikus keeles suhtlus, tagasiside andmine genereeritud vastustele ning mittefunktsionaalsete nõuete, nagu näiteks koodi loetavus ning arusaadavus, paremat täitmist.

Sarnase uuringu viisid läbi ka Wang jt. [15], kes küsitlesid 599 arendajat 18 erinevast IT firmast. Küsimustikus uuriti, kuidas on vastajad juba kasutanud *AI* tööriistu ning mis on nende ootused sellistele abilistele. Tulemustest selgus, et peamised kasutusviisid on *API* argumentide soovitus, kasutatavate muutujate nimede ennustus ning meetodi skeleti soovitus. Ootused abiliste funktsionaalsusele kattusid suures osas juba eelmainitud kasutusviisidega. Kõige enam oodati, et abilised suudaksid poolelioleva rea lõpuni genereerida, *API* argumente soovitada ning meetodite kondikava ennustada. Ajalise ootamise suhtes olid 85 % vastanutest rahul, kui rea lõpuni genereerimine võtab aega kuni 0,2 sekundit. Lisaks nendele avaldati soovi ka abiliste õppimisvõime vastu, et pakutud vastused adapteeruksid ja kohaneksid kasutaja koodi stiiliga.

Autori poolt koostatud nõuded valitavale tööriistale on järgmised:

- **Koodi privaatsus –** Lähtekoodi, mille kohta küsimusi küsitakse ning mida täiendada soovitakse ei tohi tööriist salvestada ega kasutada treeninguks.
- **Kasutusmugavus** – Tööriist peab olema integreeritud *IDE*'sse ning peab eksisteerima vestlusaken.
- **Olemasoleva koodi taaskasutamine –** Vastuseid genreerides peab tööriist kogu koodibaasist teadlik olema ning olemasolevaid meetodeid võimalusel taaskasutama.
- **Stiili säilitamine** – Vastuseid pakkudes peab tööriist genereerima sarnase stiiliga koodi nagu eksisteerivas koodibaasis.
- **Vastuse genereerimise aeg alla 10 sekundi**.
- **Kättesaadavus -** Peaks saama kohe kasutusele võtta ning eksisteerima prooviperiood.

Koodi privaatsus on üks olulisemaid nõudeid, sest ettevõtte lähtekood sisaldab ka unikaalseid võtteid probleemide lahendamiseks ning need ei tohiks lekkida konkurentidele kasutuseks. Oluline on, et valitav tööriist ei salvestaks küsimustes refereeritavat koodi teenusepakkuja serveritesse ega kasutaks seda teistele kasutajatele vastuste genereerimiseks. Aktsepteeritav on koodi lokaalselt säilitamine ning selle põhjal „õppimine", et tulevasi vastuseid relevantsemaks teha. Samuti on aktsepteeritav ajutine koodilõigu edastamine teenusepakkuja serverisse, et vastust saada, kuid ainult krüpteeringu olemasolul.

Kasutusmugavus on teine oluline nõue. Kui tööriista kasutamine oleks aeganõudev ja ebamugav kaoks selle rakendamise põhimõte täielikult. Peamine eesmärk on arendusprotsessi kiirendada ning efektiivsemaks muuta. Seetõttu on oluline, et valitav tööriist oleks lihtsasti kasutatav. Selleks on selge eelistus, et abivahendit oleks võimalik otse ettevõttes kasutusel olevas *IDE*'s ehk *integrated development environment*'is, *Visual Studio Code*'is kasutada. Teine kasutusmugavuse nõue on vestlusakna olemasolu. Oluline on, et tööriist suudaks tavakeelest koodi genereerida. Ei ole vaja mitmeid nupuvajutusi, vaid saab lihtsalt inimkeeles selgitada, mida on vaja muuta ning milline võiks olla oodatav tulemus.

Antud töö peamine eesmärk on tööriista võime olemasoleva koodi stiili rakendada ka genereeritavatele vastustele. Nõutud on eksisteeriva koodi stiili mõistmine ning selle rakendamine genereeritud vastustes. Lisaks sellele peab tööriist suutma taaskasutada olemasolevat koodi. Mõlemat tehes peab tööriist suutma leida relevantset koodi kogu *IDE*'s lahtioleva tööruumi ulatuses, mitte ainult lahtiolevatest failidest. Vastasel juhul saaks arendaja ise tulemuseni jõuda otsingufunktsiooni kasutades. See aga on just peamine situatsioon, mida autor vältida ning parendada soovib. Eesmärk on mitte koodi duplikeerida ja koodi puhtust maksimaalsena hoida. Niimoodi säilib koodibaasi ühtsus ning seda on lihtsam hallata.

3.2 Olemasolevad tööriistad

Bakalaureusetöö kirjutamise ajaks on tehisintellektil põhinevaid tööriistu saadaval üsnagi mitmeid. Pakutavate vahendite võrdlust ning *AI* tööriistade hetkeolukorra uuringuid on ka varem läbi viidud. Et saada ülevaadet, mis variante ning funktsionaalsuseid on saadaval, on autor tutvunud eelnevalt läbi viidud uuringutega ning toestanud ka iseseisvat internetiotsingut. Hliš jt. [16] leidsid enda uuringus, et kättesaadavad intelligentsed tööriistad olid *Github Copilot*, *Tabnine*, *Kite* ning *IntelliCode*. Lisaks neile neljale leidsid autorid ka *CACHEA*, *Pythia* ning *Galois* nimelised tööriistad kuid need ei olnud edasiseks uurimiseks kättesaadavad. Funktsionaalsuse poolest oli koodi lõikude genereerimiseks uuringu läbiviimise hetkeks võimeline vaid *Github Copilot*, kuid sõnade ning ridade genereerimist suutsid kõik kolm peale *IntelliCode'*i. Inimkeelsete käskude põhjal suutsid koodi genereerida vaid *Copilot* ja *Tabnine*.

Liang jt. [14] küsitluse tulemustest selgus, et populaarseim abivahend oli *Github Copilot* mida kasutas 306 vastajat, sellele järgnesid *Tabnine* 118 vastajaga, *Amazon Codewhisperer* 50 kasutajaga ning kõige vähem kasutajaid oli *ChatGPT*'l, vaid 25. 54 vastajat kasutasid ka organisastioonispetsiifilist tööriista, kuid need ei ole avalikult kättesaadavad. Kasulikkuse poolest selgus, et kõige suurema osa kirjutatavast koodist suutis kasutajatele genereerida organisatsioonipõhine tööriist, mis genereeris 37% kirjutatavast koodist. Järgmine oli *Copilot* 30,5 %-ga, seejärel *Tabnine* ning *ChatGPT* 20 %-ga ja kõige kehvemat tulemust pakkus *Codewhisperer* 5 %-ga. Wang jt. [15] said teada, et nende küsitletavate seast suurem osa kasutas *IDE*'sse sisse ehitatud tööriistu, lausa 96 %. Populaarsuselt järgmised on kompilaatoripõhised tööriistad ning kolmanda osapoole pistikprogrammina pakutavad tööriistad ei ole väga populaarsed. Neid oli kasutanud vaid 13 % vastanutest. Kasutatud oli *IntelliCode*, *Copilot*, *Tabnine*, *Kite* ning

AiXcoder. Tööriistad, mida teised oma töödes maininud ei ole on Corso [5] poolt välja toodud *Google Bard* ning Catir ja Claessoni [11] katsetatud *CodePal*.

3.3 Tööriista valiku protsess

Kuna paljudes uuringutes oli *Github Copilot* tulemustelt parim esineja ning väga populaarne kandidaat *AI* tööriistaks, alustas autor esimesena selle lähemat uurimist. See toetab integratsiooni populaarseimate *IDE*'dega nagu *Visual Studio Code, Visual Studio, Vim, Jetbrains* ja muud. *Copilot* on töö kirjutamise hetkel kolme erineva tasemega tellimuspõhine tööriist. Saadaval on *Individual, Business* ning *Enterprise* taseme tellimused. *Individual* ja *Business* plaanid on funktsionaalsuselt sisuliselt samad, erinevus on litsentside haldamises ning andmete säilitamise osas. Nimelt vaikimisi salvestatakse *Individual* tellimuse korral tööriistale tehtud päringud, vastused ning nendes sisalduv kood. Koodi põhjal treenitakse ka keelemudeli vastuseid. Küll aga on võimalik ka *Individual* tellimuse korral seadetest see variant maha võtta, et koodilõike ei säilitataks ega nende põhjal ei treenitaks keelemudelit. *Business* ja *Enterprise* tellimuste korral koodilõike ei salvestada ega kasutata mudelite treenimiseks. *Enterprise* pakub lisafunktsionaalsust kasutaja repositooriumidel põhineva vestluse näol nii *IDE*'s kuid ka *Github*'i kodulehel. Lisaks sellele pakub ka *Pull request*'ide analüüsi. Antud töö eesmärki silmas pidades tundus algselt *Enterprise* versioon kõige sobivam, kuna mainitakse kasutaja repositooriumi põhist vestlust ning koodi ei salvestata ega kasutata keelemudeli treenimiseks. Kahjuks ei ole aga *Enterprise* versioon lihtsasti saadaval, selleks peab suhtlema müügitiimiga ning organisatsioon peab olema juba *Github Enterprise Cloud* kasutaja. Ettevõte, kus autor töötab, ei ole aga *Github Enterprise Cloud* kasutaja, seega *Enterprise* versiooni katsetada ei saanud. Kuna üks oluline nõue oli koodi turvalisus otsustas autor ka *Business* ning *Individual* teenuseid mitte kasutada. Põhjuseks on *Copilot*'i toimimispõhimõte. Kuigi *Github* lubab, et koodi ei salvestada ega kasutata mudeli treenimiseks saadetakse küsimustes kontekstiks esitatud kood ikkagi *Github*'i serveritesse, kus vastuse saamiseks seda töödeldakse. Kuigi *Github* kasutab mitmeid ennetavaid praktikaid nagu krüpteerimine info lekkimise ennetamiseks, eksisteerib ikka risk andmelekkeks. Kahjuks ei elimineeri need praktikad aga koodi lekkimist täielikult. Niu jt. [17] ning Finkman jt. [18] tõestasid enda töödes, et *Copilot*'i kasutatav keelemudel võib kasutada ühe kasutaja koodi teiste vastuste genereerimisel. Turvalisuse huvides soovis autor koodi kasutamist teistele kasutajatele vastuste genereerimisel vältida, seega *Copilot* kasutusse ei läinud [19].

Teine populaarne variant, mis uuringutest välja tuli, oli *Tabnine*. *Tabnine* on samuti tellimuspõhine toode kolme erineva tasemega: *Basic*, *Pro* ning *Enterprise*. Kõiki tasemeid saab kasutada sarnaselt *Copilot*'iga populaarseimates *IDE*'des. Erinevus tuleb funktsionaalsusest, *Basic* pakub üldist tehisintellekti poolt koodi generatsiooni ning vestlusfunktsiooni piiranguga. *Pro* poolt pakutavad *AI* mudelid on võimekamad ning personaliseeritud *IDE*'s olevale koodile. *Enterprise* versioon lubab veelgi kasulikumat funktsionaalsust, näiteks lokaalset jooksutamist ning koodibaasipõhist *AI* mudelit ning vestlust, mis on ka antud töö peamine eesmärk. Kahjuks aga on *Enterprise* versiooni puhul vajalik minimaalselt üheaastane tellimus ning seetõttu jäi antud versioon kasutamata. Kuna *Tabnine'i* iga versiooni puhul koodi lokaalsest masinast välja ei saadeta on turvanõue täidetud, seega autor liikus edasi, et hinnata vastavust ülejäänud nõuetele. Kahjuks ei vastanud *Pro* versioon koodi stiili ning taaskasutuse nõuetele. *Tabnine* oli võimeline kontekstiks võtma *IDE*'s lahtiolevat faili ning sealt üles leidma sarnase funktsionaalsusega koodi, mida küsiti, kuid see pole piisav. Et olla kasulik peab abivahend suutma kogu *IDE* tööruumi ulatuses koodi läbi otsida. Kui abivahend suutis leida sarnast koodi, mida implementeerida, siis kasutas peamiselt kopeerimist ehk ei arvestanud päringus mainitud muudatusi ning kopeeris täpselt samasuguse eksisteeriva koodi. See ei täida koodi duplikatsiooni vähendamise nõuet. Eelmainitud nõuetele mitte vastamise tõttu *Tabnine* tööriista edasi ei uuritud [20].

Kolmandaks variandiks osutus *Bito*. Selle tööriistani jõudis autor iseseisva veebiotsingu käigus, sest üheski teaduskirjanduse allikas seda mainitud ei ole. Puudulik kajastus on tõenäoliselt seetõttu, et esimene avalik *Bito* versioon tuli alles 2023. aasta juunis. *Bito*, nagu ka kaks eelnevalt käsitletud abivahendit, on tellimuspõhise mudeliga. Saadaval on nii tasuta variant kui ka tasuline. Tasuta versioon kasutab *GPT 3.5* ja teisi sarnase võimekusega keelemudeleid ning on piiratud päringute arvuga. Tasuline plaan kasutab aga *GPT 4* mudelit ja muid kõrgeima taseme mudeleid. Kasutuslimiit enamjaolt puudub, kuid *GPT 4* mudelile saab päringuid teha 400 ning peale seda on iga täiendav päring lisatasu eest 0,10 dollarit. Lisaks sellele pakub tasuline variant ka koodibaasi mõistmise funktsionaalsust. Turvalisuse poolelt vastab *Bito* püstitatud nõuetele, see tähendab, et koodi ei salvestata ega kasutata keelemudeli treenimiseks. Kuna funktsionaalsus tundub paljulubav ning turvanõuded on täidetud asus autor tööriista katsetama [21].

4 Eksperiment

Selleks, et hinnata *AI* tööriista võimekust ning kasutegurit tarkvaraarendusprotsessis viib autor läbi eksperimendi. Eksperimendi eesmärk on hinnata tööriista võimet koodi stiili säilitada ning efektiivselt taaskasutada olemasolevat koodi. Käesolev peatükk selgitab eksperimendi sisu ning seejärel tulemusi.

4.1 Metoodika

Antud eksperimendi eesmärk on hinnata abivahendi poolt pakutud tulemust. Kui tulemuseks on töötav ning eelpool püstitatud nõuetele vastav kood, saab eksperimendi lugeda edukaks. Kuna eesmärk on muuta arendusprotsess efektiivsemaks, tuleks arvestada ka ajakuluga, see tähendab kui kaua aega kulub adekvaatse tulemuseni jõudmiseks. Antud töö ei keskendu aga ajalise kulu mõõtmisele, vaid pigem võimekuse kaardistamisele. Ajakulu võetakse arvesse analüüsi osas ning hinnatakse, kas abivahendi kasutamine tuli kasuks või mitte.

Eksperimendi läbiviimiseks kasutatakse autori ettevõttes esinenud reaalseid tarkvaraarendusülesandeid, mis on juba tarkvaraarendajate poolt lahendatud. Enne *Bito*'le ülesande andmist kasutatakse ära *Git* versioonihalduse võimalust taastada koodi mineviku olukord. Koodibaas taastatakse samasse seisundisse nagu enne ülesande täitmist. Seejärel alustab autor probleemi lahendamist *Bito* abiga. Niimoodi samme tagasi võttes on hiljem väga hea võrdlusmomenti tekitada. Nii inimarendajal, kes ülesannet esialgu lahendas, kui ka *AI* tööriistal on täpselt sama alguspunkt ning sama probleem lahendamiseks. Kuna sisendid on mõlemale üsnagi sarnased, on lõpptulemus võrreldav ning võimalik hinnata, kas ja kuidas lahendusviisid erinevad.

4.2 Tulemused

Eksperimendi läbiviimiseks valib autor kolm erinevat ülesannet eri keerukustasemetega. Ülesannete valikul keskendub autor sellele, et tööriistal oleks olemasolevas koodis kuskilt kasutatavat koodi otsida. Ehk siis esimeses peatükis mainitud olukorda, kus on vaja arendada funktsionaalsus, mis kuskil koodibaasis juba rakendatud on.

Esimeseks ülesandeks valib autor hästi lihtsa ülesande, mis on aga väga korduv. Nimelt on ülesandeks ühe äpi piires kasutatavad dialoogid muuta lohistatavaks. Dialoogi

lohistatavaks muutmiseks tuleb elemendile lisada *draggable="true"* atribuut. Antud kontekstis mõeldakse äpi all suurema põhirakenduse sees olevat alamrakendust. Muudatus ise on väga lihtne, kuid selle ülesandega oli autori eesmärk testida, kas tööriist leiab üles kõik dialoogid, kus on vaja muudatus sisse viia. Niimoodi saab arusaama tööriista võimekusest koodibaasi mõista ning analüüsida. Antud ülesanne ei olnud täiesti edukas. Peale mitut erineva sõnastusega päringukatset suutis abivahend tuvastada 12-st muudatust vajavast dialoogist seitse. Muudatus, mida tööriist soovitas, oli korrektne. Küll aga pakuti muutmiseks ka teisi dialooge, mis ei olnud antud ülesande skoobis. *AI*'le juhiseid andes täpsustas autor konkreetset kausta projektis, kust dialooge leida, mida lohistatavaks teha. Täpsustusest olenemata pakkus *Bito* muutmiseks ka antud kaustast väljaspool asuvaid dialooge. Lisaks sellele sattus tööriist segadusse kirjeldades koodi hetkeseisu. Tuues koodilõike eksisteerivatest dialoogidest, selgitas abivahend, millised dialoogid on juba lohistatavad ning millistele peaks lohistamist lubava atribuudi lisama. Nendes selgitustes esines vigu. Mõned dialoogid, millel tegelikult ei olnud vastavat atribuuti veel olemas, nimetas *Bito* juba lohistatavateks, kuigi ka näidatud koodilõigus ei olnud korrektset atribuuti olemas. Kokkuvõttes ei saavutanud autor selle ülesandega tulemust, mis soovis. Eeldatav käitumine oleks olnud kõikide dialoogide üles leidmine, kinnitamaks, et midagi ei jäänud kahe silma vahele.

Teine ülesanne on keerukustasemelt veidi keerulisem, keskmise keerukusega. Nimelt on vaja kahe veeru numbrilised väärtused korrektselt vormindada. Muudatus tuli sisse viia kahes vaates. Antud ülesande eesmärk oli testida, kas tööriist leiab varasemalt implementeeritud vormindamise meetodi üles ning oskab korrektselt soovitada, kuidas seda antud olukorras rakendada. Ka selle ülesande täitmisega esines tööriistal raskuseid. Jällegi proovis autor erinevaid päringuid ning täpsustas, kus failis on vaja muudatus sisse viia ning mida täpsemalt on vaja muuta. Kohati sai *Bito* ülesandest valesti aru ning pakkus vastusteks sisu valedest kohtadest. Mõnel korral pakkus tööriist ka implementeeritavaid vormindusfunktsioone, kuid need ei olnud olemasoleva koodiga kooskõlas, mis tähendab, et varasemalt ei olnud koodibaasis sääraseid meetodeid kasutatud ning lõpptulemus ei oleks olnud täielikult korrektne. Ühe vastuse puhul pakkus tööriist välja esmapilgul sobiva funktsiooni, kuid lähemal uurimisel oli funktsioon vigane ning tegelikult koodibaasis seda ei eksisteerinud. Autor ei oska kahjuks arvata, miks selline vastus genereeruda sai ning klienditoega suheldes ei osatud samuti autorile vastust anda. Korrektse vastuseni jõudis autor samm-haaval, mitte ainult ühe päringuga. Muutes päringu sõnastust natuke täpsemaks ei pakkunud *Bito* täiesti korrektset vastust, kuid

mainis pealtnäha korrektse meetodi olemasolu ning selle asukohta. Vastusest saadud info põhjal sai autor täiendavaid küsimusi küsida ning mitme vastuse põhjal koostada korrektse lahenduse. Saadud infokildude alusel oli korrektset lahendust üsnagi lihtne implementeerida.

Kolmandaks ülesandeks on vaatele vahekaartide lisamine. Olemasoleva vaate sisu pidid mõlemad lisatavad vahekaardid kasutama, kuid andmed, mida kuvada, tuli filtreerida erinevate vahekaartide vahel. Antud ülesande puhul ei olnud ootust, et andmete filtreerimist oskaks abivahend teha, kuna see sõltus äriloogikast. Autor soovib saada näiteid, kuidas vahekaartide lisamist varasemalt on tehtud ning näidet, kuidas vahekaartide vahetamisel sisu muuta. Ülesande eesmärk on taaskord hinnata tööriista võimet leida eksisteerivat koodi, mis täidaks püstitatud ülesannet ning soovitada antud olukorrale vastavat lahendust. Niimoodi säilib koodibaasis ühtlane stiil ning hallatavus paraneb. Järjekordselt katsetas autor eri päringuid ning *Bito* soovitas mitut eri lahendust. Antud ülesandega sai abivahend üsna hästi hakkama, soovitatud lahendused toimisid hästi ning kõrvalist ega ebavajalikku infot vastustes ei kajastunud.

5 Analüüs

Selles peatükis kontrollitakse valituks osutunud tööriista vastavust püstitatud nõuetele, analüüsitakse eelnevas peatükis saadud tulemusi, võrreldakse neid inimarendajate poolt tehtud lahendustega ning hinnatakse, kas integreerimine arendusprotsessi oleks mõistlik või mitte. Lisaks sellele arutletakse edasiarendustest ning alternatiivsetest lahendustest.

5.1 Tööriista analüüs

Koodi privaatsuse nõuetele vastab *Bito* peaaegu täielikult. Tööriist indekseerib koodi lokaalselt ning säilitab indeksid vaid kasutaja arvutis. Kontekstiks vajalikke koodilõike võidakse saata koos päringuga teenusepakkuja serverisse, kuid koodi sinna ei salvestata ning kustutatakse peale vastuse saamist. Päringute saatmine on krüpteeritud ning toimub kasutades *HTTPS* ehk *Hypertext Transport Protocol Secure* protokolle. Tehtud päringuid ning saadud vastuseid ei salvestata kuhugi mujale peale kasutaja enda masina ning nende põhjal ei treenita ühtegi *AI* mudelit [21].

Bito vastab ka kättesaadavuse ning kasutusmugavuse nõuetele. Tööriista kasutamiseks ei pea läbima valideerimisprotsesse ega nõustuma pikema minimaalse kasutusajaga. Nii tasuta kui ka tasulist versiooni saab hakata kasutama vaid mõne minutiga ning kättesaadavuspiiranguid ei ole. *Bito* kasutamine on üsnagi mugav, sest on olemas laiendus, mida saab *IDE*'le lisada ning nautida koodiaknas ilmuvaid soovitusi või suhelda läbi suhtlusakna. Juturoboti funktsioon on eriti mugav, kuna lihtsas tekstis on end kergem väljendada ning ei pea teadma täpseid tehnilisi väljendeid.

Kõige olulisemad nõuded ning käesoleva bakalaureusetöö peamine rõhk oli tööriista võimel olemasolevat koodibaasi mõista ning ära kasutada. Selles vallas on tööriistal veel arenguruumi. On selge, et algne ning põhiline arusaam olemasolevast koodist eksisteeris, kuid tihti esines ka koodi valesti mõistmist või üldse niiöelda hallutsinatsioone, kus tööriist presenteeris kasutajale koodi, mida tegelikult koodibaasis ei eksisteerinud. Kui aga selgemalt täpsustada, millist konkreetset elementi või funktsionaalsust koodibaasist leida, suutis tööriist enamjaolt pakkuda vähemalt vastust, mille põhjal edasi uurida ning täpsustavaid küsimusi küsida.

5.2 Lahenduste analüüs

Dialoogide lohistatavaks muutmise lahenduse kood oli korrektne. Dialoogid, mis tööriist üles leidis, muutusid peale lahenduse implementeerimist lohistatavaks ja ülesanne sai täidetud. Küll aga ei suutnud tööriist kõiki dialooge tuvastada ning tekitas veidi segadust mainides dialooge, mis ei olnud antud ülesande skoobis. Autori soov oli, et *AI* abivahend suudaks antud ülesande puhul pakkuda kindlustunnet ning kiiremat lahendusaega, leides üles kõik vajalikud dialoogid ning näidates, kus on vaja muudatus sisse viia. Kuna nii mõnedki dialoogid jäid abivahendil märkamata, tuli ikkagi autoril ise skoobis olev kood üle käia ning kontrollida. Seetõttu soovitud ajavõitu ega kindlust selle ülesande lahendamisel abivahendist ei olnud. Küll aga oli mugav otse *IDE*'s küsida, kuidas üldse dialooge lohistatavaks saab teha ning kiirelt vastus saada. Ei pea aknaid vahetama ega internetibrauserist ise otsima hakkama ning see oli vägagi mugav. Sarnast kasu *AI* tööriistade kasutamisel märkasid ka Vaithilingam jt. [22] ning Liang jt. [14] enda läbiviidud küsitluste tulemustes.

Teise ülesande puhul oli muudatuste skoobiks konkreetne äpp, kuid abivahendilt ootas autor lahenduste otsimist kogu koodibaasist. Numbriliste väärtuste vormindamisel on juba rohkem analüüsitavat koodi ning saab tekitada parema võrdlusmomendi arendaja poolt implementeeritud lahendusega. Esialgsed lahendused, mida tööriist pakkus, olid küll sisult toimivad, kuid ei oleks vastanud ühtlase koodi nõuetele. Vormindamise nõudeid täpsustades suutis abivahend genereerida funktsioone, mis toimisid ning muutsid numbrid selliseks nagu kirjeldatud. Näiteks ühes päringus täpsustas autor, et kuvatavad väärtused ei tohiks omada enam kui kolme komakohta ning sisaldada üleliigseid nulle. Seejärel suutis tööriist genereerida vormindamisfunktsiooni, mis tegi just seda ning selgitas ka, kuidas vaates antud funktsiooni kasutada. Antud funktsioon on välja toodud joonisel 1.

```
formatNumber: function (value) {
    if (value !== null && value !== undefined) {
        return parseFloat(value).toFixed(3).replace(/\.?0+$/, "");
    }
    return value;
}
```

Joonis 1. Genereeritud vormindamisfunktsioon

Võrreldes pakutud vormindamisfunktsiooni varasemalt implementeeritud meetodiga, mille tööriist lõpuks suutis ka välja pakkuda, on näha, et oodatav funktsioon on üsnagi teistsuguse sisuga, kui *AI* poolt pakutud lahendus. Tehisintellekti pakutud vastuses pole

kasutatud eksisteerivat vormindamisfunktsiooni *getFormattedValueNoTrailingZeros*, mis tegelikult numbri vormindamise eest hoolitseb. Lisaks sellele ei ole algselt pakutud funktsioonis ühikuid numbrile järgi pandud ega käsitletud olukord, mil peaks tagastama sidekriipsu väärtuse asemel. Korrektne funktsioon on esitatud joonisel 2.

```
formatWorkText: function (oActivity) {
  if (oActivity) {
    const oInputHelpModel = this.getView().getModel('inputHelp');
    const oInputHelpModelData = oInputHelpModel.oData;
    const iFormattedWork =
Global.getFormattedValueNoTrailingZeros(oActivity.Work, 11701,
oInputHelpModelData);
    if (oActivity.Work > 0) {
      return iFormattedWork + ' ' + oActivity.DurationUnitSymbol + '/' +
oActivity.DefaultUnitSymbol;
    } else {
      return '-';
    }
  }
}
```

Joonis 2. Korrektne vaatepõhine funktsioon

Vaatepõhises funktsioonis kasutatud *getFormattedValueNoTrailingZeros* meetod tuli samuti uuesti implementeerida, kuna antud ülesande äpis seda veel tehtud ei olnud. *AI* üritas ka seda ise soovitada, kuid mingil põhjusel tekitas korrektsest meetodist vigase meetodi. Tööriista poolt pakutud meetod (Joonis 3) sisaldab süntaksiviga, kus üritati uuesti deklareerida muutujat, mis meetodi parameetrites juba olemas oli (*iPrecision*, *bNumber*). Lisaks sellele olid süntaksiviga tekitanud meetodi parameetrid valed. Kui aga abivahend õiget meetodit vastuses kasutas, oli lõpliku lahenduseni jõudmine üsnagi lihtne ning kiire. Õige meetodi kasutamiseni jõudis abivahend mitme küsimusega. Esimeses vastuses viidati korrektsele vormingufunktsioonile ning selle põhjal sai autor tööriistalt täpsustavaid küsimusi küsida ning jõuda korrektsemate vastusteni. Tööriist oskas öelda korrektselt asukoha, kus varasemalt sarnast vormindamist on kasutatud ning selle abil oli lihtne koodi ka antud ülesande jaoks kohandada. Ajalises mõttes võrreldes arendajaga, kes algselt probleemi lahendas, suurt võitu ei olnud. Ajakulu tuli kokkuvõttes enam-vähem sama, mis inimarendajal. Küll aga tuleb siin arvestada, et peamine ajakulu oli õige vastuse saamiseks abivahendilt. Kui esimeses vastuses oleks kohe õige info olnud, oleks ülesandele kulunud ligi tund aega vähem.

```
number: function(value, iPrecision, bTrailingZeros, bNumber) {
  const oGlobalModel = this.getModel('global');
  const oGlobalData = oGlobalModel.getData();
  const oCustomerFormatSettings = oGlobalData.CustomerFormatSettingsByTypeId;
  const iPrecision = oCustomerFormatSettings[typeId] &&
oCustomerFormatSettings[typeId].NumberPrecision;
  const bTrailingZeros = oCustomerFormatSettings[typeId] &&
oCustomerFormatSettings[typeId].TrailingZeros;
  const sFormattedValue = formatter.number(value, iPrecision, bTrailingZeros,
bNumber);
  return sFormattedValue;
}
```

Joonis 3. Abivahendi poolt pakutud vigane vormindusfunktsioon

Vahekaartide lisamise ülesannet suutis analüüsitav abivahend üsna hõlpsalt lahendada. Küll aga ei olnud esialgsed lahendused puhta koodi standarditega täielikus kooskõlas [23]. Esialgsetes vastustes soovitas tööriist koodi duplikatsiooni. Kuna mõlemad vahekaardid pidid kasutama sama vaadet, oleks võinud vaate kood säilida ning olla taaskasutatav mõlema vahekaardi poolt ning vahe oleks tulnud sisse andmete filtreerimisel. Säärast lahendust oskas autor otsida, kuna komponentide kasutamise juhendis oli teatud stiilinäiteid mainitud, kuid täis lahendust sealse materjali põhjal koostada ei olnud võimalik. Kui autor päringus täpsustas, et soovib taaskasutada olemasolevat vaadet mõlema vahekaardi jaoks, oli lahenduseks vaate koodi duplikeerimine mõlema vahekaardi alla (Joonis 4).

```
<f:DynamicPage fitContent="true" class="sapUiNoContentPadding">
  <f:content>
    <IconTabBar id="classificationTabBar" select="onTabSelect">
      <items>
        <IconTabFilter text="Tab 1">
          <i:TreeTableExt id="classificationUsagesTable" ...>
            <!-- Existing table structure here -->
          </i:TreeTableExt>
        </IconTabFilter>
        <IconTabFilter text="Tab 2">
          <i:TreeTableExt id="classificationUsagesTable" ...>
            <!-- Same table structure, different filter -->
          </i:TreeTableExt>
        </IconTabFilter>
      </items>
    </IconTabBar>
  </f:content>
</f:DynamicPage>
```

Joonis 4. Pakutud lahendus vahekaartide implementatsiooniks

Alternatiivse lahendusena pakkus tööriist välja ka taaskasutatava vaate eraldi fragmendiks tegemise ja omakorda fragmendi kasutamist vahekaartidel (Joonis 5).

```
<IconTabBar>
    <items>
        <IconTabFilter text="Global" key="global">
            <content>
              <!--Include the same view/fragment used for the 'global' tab-->
              <core:Fragment fragmentName="your.fragment.name" type="XML"/>
            </content>
        </IconTabFilter>
        <IconTabFilter text="Project Default" key="projectDefault">
            <content>
              <!--Include the same view/fragment used for the 'global' tab-->
              <core:Fragment fragmentName="your.fragment.name" type="XML"/>
            </content>
        </IconTabFilter>
    </items>
</IconTabBar>
```

Joonis 5. Abivahendi poolt pakutud fragmendi kasutamine

Autori arvates ei oleks see aga mõistlik lahendus, kuna konkreetset vaate struktuuri kasutatakse vaid neil kahel vahekaardil, seega ei oleks mõistlik vaate struktuuri antud failist välja tõsta. Korrektsem ning ka ülesande täitmisel inimarendaja poolt implementeeritud lahendus säilitab vaate struktuuri ning muudab vahekaartidele vajutades kuvatavaid andmeid (Joonis 6). Andmete muutmine toimub *IconTabBar* elemendi *select* sündmuse toimumisel, kui kutsutakse välja meetod *onTabSelect*. Niimoodi ei ole *IconTabFilter* elementidesse vaja vaate koodi üldse kirjutada ning olemasolevale vaate koodile on vaja üles lisada vaid *IconTabBar* elemendid.

```
<f:DynamicPage fitContent="true" class="sapUiNoContentPadding">
    <f:content>
      <VBox fitContainer="true">
        <IconTabBar id="idIconTabBar" select="onTabSelect"
backgroundDesign="Transparent" headerBackgroundDesign="Transparent"
expandable="false" class="fixContentContainerShowing" visible="false">
          <items>
            <IconTabFilter icon="sap-icon://globe"
text="{i18n>MasterGlobalIconTabBarText}"
tooltip="{i18n>MasterGlobalClassificationTooltip}" key="global" />
            <IconTabFilter icon="sap-icon://factory"
text="{i18n>MasterProjectDefaultIconTabBarText}"
tooltip="{i18n>MasterProjectDefaultClassificationTooltip}"
key="projectDefault" />
          </items>
        </IconTabBar>
         <i:TreeTableExt id="classificationUsagesTable" ...>
           ...
         </i:TreeTableExt>
      </VBox>
    </f:content>
  </f:DynamicPage>
```

Joonis 6. Vaate struktuuri taaskasutav lahendus

Eksperimendi huvides proovis autor saada tehisintellektilt vastust, kus oleks koodi duplikatsiooni välditud, kuid sellise vastuseni autor ei jõudnud. Küll aga sai juba esimese paari vastuse abil ülesannet lahendada, seega tööriistalt täiesti valmis implementatsiooni saamine ei olnud oluline. Võrreldes ülesandele logitud ajakuluga oli *AI*'d kasutades ajakulu tunduvalt väiksem.

5.3 Tööriista rakendamise edukus

Tööriista rakenduse edukuse hindamiseks saab esialgu vaadelda ülesannete ajakulu: kas tööriista kasutades suutis autor ülesandeid kiiremini lahendada, kui arendajad, kes tööriista ei kasutanud? Nagu eelnevas alapeatükis mainitud, esimest ülesannet lahendades ei tekitanud tööriista kasutamine suurt ajavõitu ning lahendamiseks kulunud aeg oli üsna võrdne nii tööriistaga kui ilma tööriistata. Vormindamise ülesande puhul samuti suurt ajavõitu ei esinenud. Kuna aga ajakulu tekkis peamiselt päringute muutmisest kuniks jõuti sobiva vastuseni usub autor, et tööriista pikemaajalisel kasutusel tekib parem arusaam päringute koostamisest ning on võimalik ajakulu vähendada. Kuigi antud eksperimendi korral ei tekkinud suurt ajavõitu, võtaks tulevikus korrektse lahenduseni jõudmine vähem aega ning sellisel juhul oleks ajavõit juba märkimisväärne. Vahekaartide ülesanne

demonstreeris hästi tööriista kasulikkust. Ajakulu ülesande lahendamiseks oli tunduvalt väiksem, mitme tunni võrra, sest sobiva lahenduseni jõuti üsna väheste päringutega.

Teine aspekt, mis autori arvates tõestab tööriista kasulikkust on iseseisvuse suurendamine. Uute arendajate esimesed töönädalad mööduvad tihti kolleegide abiga ning see on märkimisväärne ajakulu ka nendele. Seetõttu on hea kui eksisteerib säärane tööriist, mis suudab olla koodibaasiga tuttava kolleegi asendaja. Vabastades niimoodi kolleegi aega ja suurendades uue töötaja iseseisvust. Antud tööriista võimekust olla eelmainitud asendaja tõestab fakt, et kõik ülesanded said täidetud ning aega kulus kas sama palju või vähem, kui ilma tööriistata. Sellele aitas kaasa kindlasti juturoboti funktsiooni olemasolu, sest see võimaldas tööriistaga suhelda samamoodi nagu toimuks suhtlus kolleegiga.

Nagu eelpool mainitud oleks tööriista kasutamise alternatiiv kolleegilt abi küsimine. Ka seda tehes tuleks enne läbi viia iseseisev internetiotsing ning koodi uurimine, et arendaja endale ülesandest parema arusaama saaks. Edukuse saavutamiseks tulekski tööriista kasutada kui kolleegi. Eksperimendi käigus selgus, et lahenduseni jõudmiseks oli tihti vaja küsida infot jupp-haaval ning kasutada vastusest saadud infot järgmise küsimuse küsimiseks. Võrreldes tööriista kasutamist internetiotsinguga usub autor, et tööriista kasutamine on palju efektiivsem. Esiteks on tööriist juba *IDE*'s olemas ja selle kasutamiseks ei pea aknaid vahetama. Teiseks suudab abivahend jagada sama infot, mida internetiotsing, kuid lisaväärtusena on tööriist võimeline vastuseid otsima ka olemasolevast koodibaasist. Seetõttu on vastused personaalsemad ning üldjuhul ka kasulikumad. Eksperimenti läbi viies tundis autor sama. Tööriista kasutamine oli väga mugav ning kasulik. Tulemus on üldjuhul parem, kui internetiotsingu puhul ning vastuste saamiseks ei pea *IDE*'st lahkuma. Küll aga genereerib tööriist vaid ühe vastuse korraga ning võrdlust pakutud vastuste vahel peab kasutaja ise suutma läbi viia. Internetiotsingul leidub tihti erinevaid lahendusi ning inimesi, kes on võrdluse juba läbi viinud ning selgitavad plusse ja miinuseid. Säärase võrdluse puudujääki täheldasid ka Vaithilingam jt. [22].

5.4 Alternatiivid ning edasiarendused

Nagu eelpool olevast kirjanduse ülevaatest on näha eksisteerib funktsionaalselt lubavaid tööriistu üsna mitmeid. Alternatiivina oleks võinud kasutada näiteks *Copilot Enterprise* versiooni, *Tabnine Enterprise* versiooni või mitut tööriista korraga, et nende tulemusi

omavahel võrrelda. Sarnane põhjalikum tööriistade võrdlus võiks ka olla üks töö edasiarendustest. Küll aga tuleb mainida, et säärased funktsionaalsuste võrdlemised on küll konkreetsel eksperimendi läbiviimise hetkel vägagi huvitavad ning kasulikud, kuid lühikese aja möödudes võivad antud tulemused kehtetuks muutuda. Säärast fenomeni tundis ka autor olemasolevat kirjandust läbi töötades, uuringutes mainitud funktsionaalsused olid töö kirjutamise hetkeks juba kardinaalselt muutunud ning kasutuskõlblikku materjali leidus vähe.

Teine alternatiiv väljendub tulemuse hindamises. *AI* poolt genereeritud ning selle abiga kirjutatud koodi oleks saanud hinnata näiteks *SonarCloud* tööriistaga, nagu tegi Kantek [12]. Samuti oleks saanud inimeste kirjutatud koodi *SonarCloud*'iga hinnata ning võrrelda tulemusi. See aga eeldaks *SonarCloud* olemasolu ettevõtte protsessides. Kuna töö kirjutamise hetkel autori töökohas *SonarCloud* kasutusel ei olnud, ei saanud autor seda ka kasutada.

Käesoleva bakalaureusetöö jätkuna võiks uurida sarnaste tööriistade kasutamist ka testide kirjutamisel. Juba antud töös läbiviidud eksperimendi käigus oli näha *AI* võimekust olemasolevat koodibaasi mõista, mis võiks võimaldada lisaks ühiktestide genereerimisele ka näiteks integratsioonitestide genereerimist. Säärane abivahend testide genereerimiseks lahendaks peamise probleemi, miks teste tarkvaraarenduses ei rakendata: ajakulu. Ajavõit sõltub küll tööriista võimekusest, kuid potentsiaalne kasu on kindlasti olemas. Peaaegu korrektsete genereeritud testmeetodite ülevaatamine ning väikeste muudatuste sisse viimine on tunduvalt vähem ajakulukas kui kõigi meetodite nullist kirjutamine.

Lisaks testide kirjutamisele tasuks uurida, kuidas tehisintellekt saaks tõhustada ja lihtsustada arendusprotsessi aspekte peale algse koodi genereerimise. Näiteks *pull request*'ide kontrollimisel võiks *AI* tööriistast abi olla, kes teostaks esialgse analüüsi ning oskaks tuvastada *bug*'e ning ka *code smell*'e. Säärane abivahend muudaks koodi läbivaatuse protsessi kiiremaks ning lubaks arendajatel rohkem keskenduda funktsionaalsuste loomisele.

6 Kokkuvõte

Antud bakalaureusetöö eesmärgiks oli kasutada *AI* tööriista tarkvaraarendusprotsessis mahuka koodibaasi põhjal ning hinnata selle edukust. Töö motivatsiooniks oli autori isiklik kogemus just ettevõttesse tööle asununa muuta enda tööd iseseisvamaks ning kvaliteetsemaks.

Eesmärgi saavutamiseks töötas autor läbi olemasolevat kirjandust, et saada parem ülevaade varasemalt läbiviidud uuringutest ning *AI* rakendustest tarkvaraarenduses. Seejärel koostas autor nõuded, millele valitav tööriist pidi vastama. Nõuded koostati toetudes nii olemasolevale kirjandusele, juhendajaga peetud intervjuule ning autori enda ekspertiisile.

Peale tööriista valikut viis autor läbi praktilise eksperimendi, milles lahendas ettevõttes esinenenud reaalseid programmeerimisülesandeid. Tulemuse valideerimiseks võrreldi tööriista poolt genereeritud lahendusi inimarendajate lahendustega ning analüüsiti lahendusi.

Analüüsi tulemusena selgus, et tehisintellekti kasutamine on kogenematule tarkvaraarendajale üsnagi kasulik ning tõstab iseseisvust märgatavalt. Seega töö peamine eesmärk sai täidetud. Küll aga tuleb teostada abivahendi poolt genereeritud vastustele põhjalik kontroll, sest eksperimendi käigus sai autor tööriistalt ka vigaseid ning valesid vastuseid. Tulevikus on oluline jätkata tehisintellekti uurimist ja laiendada selle rakendamist tarkvaraarenduses, et maksimeerida tehnoloogia pakutavaid eeliseid ja soodustada valdkonna arengut.

Kasutatud kirjandus

[1] N. Nikolaidis, K. Flamos, D. Feitosa, A. Chatzigeorgiou and A. Ampatzoglou, "The End of an Era: Can AI Subsume Software Developers? Evaluating ChatGPT and COpilot Capabilities Using LeetCode Problems", *SSRN*, 2023, doi: http://dx.doi.org/10.2139/ssrn.4422122. [Kasutatud: 05.04.2024].

[2] H. Ekendahl and V. Helander, "Can Artificial Intelligence Replace Humans in Programming?", *Digitala Vetenskapliga Arkivet*, 2023. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1776700&dswid=-6153 [Kasutatud: 07.04.2024].

[3] M. A. Kuhail, S. S. Mathew, A. Khalil, J. Berengueres and S. J. H. Shah, "“Will I Be Replaced?” Assessing ChatGPT's Effect on Software Development and Programmer Perceptions of AI Tools", *Science of Computer Programming*, vol. 235, no. 103111, 2024, doi: https://doi.org/10.1016/j.scico.2024.103111. [Kasutatud: 07.04.2024].

[4] H. Tian, W. Lu, T.-O. Li, X. Tang, S.-C. Cheung, J. Klein and T. F. Bissyandé, "Is ChatGPT the Ultimate Programming Assistant - How Far Is It?", *arXiv*, vol. 1, no. 1, 2023, doi: https://doi.org/10.48550/arXiv.2304.11938. [Kasutatud: 08.04.2024].

[5] V. Corso, L. Mariani, D. Micucci and O. Riganelli, "Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants", *arXiv*, 2024, doi: https://doi.org/10.48550/arXiv.2402.08431. [Kasutatud: 10.04.2024].

[6] N. Nguyen and S. Nadi, "An Empirical Evaluation of GitHub Copilot’s Code Suggestions", in *The 2022 Mining Software Repositories Conference*, 2022, pp. 1-5, doi: https://doi.org/10.1145/3524842.3528470. [Kasutatud: 10.04.2024].

[7] LeetCode, "LeetCode". [Online]. Available: https://leetcode.com [Kasutatud: 15.04.2024].

[8] B. Yetiştiren, E. Tüzün and I. Özsoy, "Assessing the Quality of GitHub Copilot’s Code Generation", in *18th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2022, pp. 62-71, doi: https://doi.org/10.1145/3558489.3559072. [Kasutatud: 16.04.2024].

[9] N. Nascimento, P. Alencar and D. D. Cowan, "Artificial Intelligence versus Software Engineers: An Evidence-Based Assessment Focusing on Non-Functional Requirements", *Research* Square, 2023, doi: https://doi.org/10.21203/rs.3.rs-3126005/v1. [Kasutatud 20.04.2024].

[10] R. Pudari and N. A. Ernst, "From Copilot to Pilot: Towards AI Supported Software Development", a*rXiv*, 2023, doi: https://doi.org/10.48550/arXiv.2303.04142. [Kasutatud: 20.04.2024].

[11] E. Catir and R. Claesson, "Problem Solving Using Automatically Genereated Code", *Digitala Vetenskapliga Arkivet*, 2023. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1770592/FULLTEXT01.pdf [Kasutatud: 22.04.2024].

[12] B. P. Kantek, "AI-driven Software Development Source Code Quality", [Magistritöö], Faculty of Informatics, Masaryk University, Brno, Tšehhi, 2023. [Online]. Available: https://is.muni.cz/th/mdt17/kantek_dp.pdf [Kasutatud: 24.04.2024].

[13] M. Fowler, "Code smell", 2006. [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html [Kasutatud 25.04.2024].

[14] J. T. Liang, C. Yang and B. A. Myers, "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges", a*rXiv*, 2023, doi: https://doi.org/10.48550/arXiv.2303.17125. [Kasutatud 26.04.2024].

[15] C. Wang, J. Hu, C. Gao, Y. Jin, T. Xie, H. Huan, Z. Lei and Y. Deng, "How Practitioners Expect Code Completion?", in *ESEC/FSE 2023: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1294-1306, doi: https://doi.org/10.1145/3611643.3616280. [Kasutatud 27.04.2024].

[16] T. Hliš, L. Četina, T. Beranič and L. Pavlič, "Evaluating the Usability and Functionality of Intelligent Source Code Completion Assistants: A Comprehensive Review", *Applied Sciences*, vol. 13, no. 24, 2023, doi: https://doi.org/10.3390/app132413061. [Kasutatud 28.04.2024].

[17] L. Niu, S. Mirza, Z. Maradni and C. Pöpper, "CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot", in *32nd USENIX Security Symposium*, 2023, pp. 2133-2150. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23 [Kasutatud 30.04.2024].

[18] A. Finkman, E. Bar-Kochva, A. Shapira, D. Mimran, Y. Elovici and A. Shabtai, "CodeCloak: A Method for Evaluating and Mitigating Code Leakage by LLM Code Assistants", a*rXiv*, 2024, doi: https://doi.org/10.48550/arXiv.2404.09066. [Kasutatud 01.05.2024].

[19] Github, "Github Copilot", 2024. [Online]. Available: https://github.com/features/copilot [Kasutatud 05.05.2024].

[20] Tabnine, "Tabnine", 2024. [Online]. Available: https://www.tabnine.com/pricing/ [Kasutatud 05.05.2024].

[21] Bito, "Bito", 2024. [Online]. Available: https://bito.ai [Kasutatud 07.05.2024].

[22] P. Vaithilingam, T. Zhang and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models", in *CHI Conference on Human Factors in Computing Systems*, 2022, no. 332, pp. 1-7, doi: https://doi.org/10.1145/3491101.3519665. [Kasutatud 10.05.2024].

[23] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. United States of America, Pearson, 2008. [E-book]. Available: https://learning.oreilly.com/library/view/clean-code-a/9780136083238/ [Kasutatud 13.05.2024].

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Kristjan Stüff

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Tehisintellekti integreerimine tarkvaraarendusprotsessi olemasoleva mahuka koodibaasi põhjal“, mille juhendaja on Tauno Treier
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

20.05.2024

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.