TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Holger Nõgu 206210IABB

Manuaalne refaktoreerimine võrreldes ChatGPT keelemudeliga ettevõtte X koodibaasi näitel

Bakalaureusetöö

Juhendaja: Tarvo Treier
MSc

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: Holger Nõgu

20.05.2024

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on anda objektiivne võrdlus manuaalse ja ChatGPT keelemudeliga teostatud refaktoreerimistulemuste vahel. Arenduses on üks suurimaid kuluallikaid koodi hooldamine, milleta muutub funktsionaalsuste lisamine paratamatult aeglaseks, kuna koodibaas läheb üle aja liiga keeruliseks. Teisest küljest aeglustub koos hooldamisega ka arendus, sest arendusaega kulutatakse hoopis vigade parandamisele.

Töö käigus viidi läbi manuaalne ja ChatGPT abil refaktoreerimine, mille tulemustest tuli välja, et mõlemal viisil parandati koodi hooldatavust. Käsitsi kulus küll palju rohkem aega ning tekkis juurde suuremas koguses koodi, ent keelemudel andis vigaseid lahendusi ning muudatused ei olnud nii sisukad.

Tulemustest saab järeldada, et manuaalne refaktoreerimine on võrreldes keelemudeliga kognitiivse keerukuse vähendamisel efektiivsem ja ChatGPT seevastu palju kiirem. Parima tulemuse võib saada mõlemat lähenemist kombineerides, kus käsitsi tehtud töö tagab töökindluse ja sisukuse ning keelemudel kiiruse.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 8 peatükki, 23 joonist, 7 tabelit.

Abstract

Manual Refactoring Compared to ChatGPT Language Model's Refactoring Capabilities in the Example of Company X's Codebase

The aim of this thesis is to provide an objective comparison between the results of manual and ChatGPT language model refactorings. One of the greatest costs in software development is maintenance. Without it, adding functionality becomes too difficult due to the increasing complexity of unhandled design flaws. On the other hand, maintenance also slows down development because time is spent on fixing mistakes instead of creating new features.

In order to make the distinctions, both manual and language model refactorings were performed, and the results show that readability improved in both cases, indicating that they were effective in making the code more maintainable. However, manual refactoring significantly increased the amount of code in the project and took more time than ChatGPT. The language model, on the other hand, provided solutions that were prone to errors and did not implement changes as complex and large as manual refactoring did.

Based on the analysis and findings of this thesis, it can be concluded that manual refactoring outperformed ChatGPT in reducing the complexity of the code, while the language model was much faster than the manual approach. The best results may be achieved by combining both methods of refactoring, where the manual approach ensures the correctness and scope of the changes, and ChatGPT ensures speed.

The thesis is in Estonian and contains 40 pages of text, 8 chapters, 23 figures, 7 tables.

Lühendite ja mõistete sõnastik

ACL	*Anti-corruption Layer* – kiht erinevate alamsüsteemide eraldamiseks
CQRS	*Command and Query Responsibility Segregation* – muster, mis eraldab andme salvestamis-, lugemis- ja värskendamistoiminguid
CRUD	Lühend sõnadest *Create*, *Read*, *Update* ja *Delete*
DTO	*Data transfer object*, andmete transpordi objekt
HTTP	*HyperText Transfer Protocol*, hüperteksti edastusprotokoll
LLM	*Large language model*, suur keelemudel
OOP	*Object-oriented programming*, objektorienteeritud programmeerimine
SRP	*Single-responsibility principle*, ühe vastutusala printsiip

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Infotehnoloogia üks hädavajalikke omadusi on muutliku keskkonnaga kohandumine, kuid üldjuhul langeb süsteemi korduval täiendamisel selle kvaliteet. Nõudlus komplekssemate tarkvarasüsteemide vastu kasvab üha kiiremini ning neid tuleb suuta hallata. See teeb tarkvarahoolduse üheks suurimaks kuluallikaks arenduses, mis võib moodustada üle 80% süsteemi eluaegsetest kuludest [1].

Refaktoreerimine on koodipuhastusviis, mille käigus parandatakse olemasolevat disaini tagantjärele väikeste visuaalset käitumist säilitavate võtetega. See aitab muuta tarkvara hooldamist lihtsamaks ja seeläbi vähem kulukamaks, ent sellega kaasnevad ka ohud. Hooldamisega aeglustub uute funktsionaalsuste lisamine ja suureneb tõenäosus teha vigu [2]. Seevastu muutub ilma hooldamiseta uuendamine aeglasemaks, kuna keerulist koodi on raske täiendada [3]. Appi tulevad igasugused tööriistad, mis pakuvad automatiseeritud lahendusi. Üks võimas valik on *Large language model* (LLM) ehk suur keelemudel, mis suudab oma tohutu andmete kogusega täita mitmeid levinud tarkvaraarendusülesandeid, aga ka sellel on omad nõrkused ja ohud [4].

Töö autor puutus ettevõttes X kokku projektiga, mida oldi juba mõnda aega arendatud. Koodi olid tekkinud disainivead, mille parandamist pidi tähtaegade tõttu edasi lükkama. Pärast ajasurve vähenemist sai autor ülesande tegeleda projekti hooldamisega. Selle raames tahtis ta teada saada, kas refaktoreerimine on tulemuslikum manuaalselt või mõnd keelemudelit kasutades.

Töö eesmärgiks on esitada objektiivne võrdlus, kuidas erinevad manuaalse ja ChatGPT keelemudeliga teostatud refaktoreerimiste tulemused. Selle täitmiseks võeti aluseks autori olemasolev rakendus ettevõttes X ning püstitati alameesmärgid:

- teostada allikatele tuginedes manuaalne refaktoreerimine;
- teostada ChatGPT keelemudeliga refaktoreerimine;
- hinnata ja võrrelda objektiivselt mõõdikute abil tulemusi.

Töö teises peatükis tuleb juttu refaktoreerimise alustest: mis see on, millal ja kuidas seda implementeerida, keelemudeli võimekus seda teostada ja tulemuste mõõtmine. Kolmandas peatükis antakse ülevaade töös kasutatavatest tarkvaradest, refaktoreerimise lähenemistest ning tööprotsessist. Järgnevas kahes peatükis, neljandas ja viiendas, on kirjas manuaalsest refaktoreerimisest, kus esimene räägib koodi analüüsist ning teine muudatuste realiseerimisest. Kuuendas peatükis on esitatud ChatGPT keelemudeli refaktoreerimist ja selle eripärasid. Viimaks võrreldakse seitsmendas peatükis mõlema meetodi tulemusi ning tehakse ettepanekuid ja järeldusi.

2 Refaktoreerimise alused

Refaktoreerimise definitsiooni kasutatakse sageli liiga üldiselt. Kui kood pole protsessi käigus mitu päeva töötanud, ei olnud tegemist refaktoreerimisega. Lisaks ei piisa ainult teadmistest, kuidas seda teostada, vaid peab oskama leida ka õiget kohta [2]. Tänaseks on abiks mitmed tööriistad, mis pakuvad automatiseeritud refaktoreerimist, ent nendelgi on omad puudused.

Selles peatükis saab ülevaate refaktoreerimise definitsioonist ning millal ja kuidas seda implementeerida. Tuuakse välja refaktoreerimise olulised printsiibid, seosed disainimustritega, keelemudeli võimalused ja ohud ning tulemuste hindamised valikud.

2.1 Refaktoreerimine

Refaktoreerimine on muudatuste tegemine tarkvaras, ilma et koodi visuaalne käitumine muutuks. Muudatused seisnevad väikestes sammudes: iga refaktoreerimine võib olla üks väike võte või väikeste võtete kombinatsioon. Selle eesmärk on teha sisemine struktuur lihtsamini loetavamaks ja muudetavamaks [2].

Muudatuste käigus on aga vigade tegemine paratamatu ning korrektseks refaktoreerimiseks on väga olulisel kohal testid, mis aitavad veenduda muudatuste õigsuses. Refaktoreerimise käigus ei lisandu teste (juhul, kui ei avastata uut testitavat kohta), kuna funktsionaalsust mitte ei teki juurde, vaid muudetakse olemasolevat [2].

Kapseldamine on üks objektorienteeritud programmeerimise (OOP) olulisi põhimõtteid. See seisneb selles, et andmed on peidetud objekti sisse ja nendega saab suhelda ainult läbi vastutava objekti [5]. OOP-i programmis on mooduli kontekstiks klass, kuhu saab peita seotud meetodeid. Refaktoreerimise käigus saab seda põhimõtet tugevdada, sest paljud muudatused seisnevad funktsionaalsuse ümberpaigutamisel õigesse konteksti, mis suurendab kapseldamist [2]. OOP-iga on tulnud kaasa suur kogus olulisi printsiipe: üks nendest on *Single-responsibility principle* (SRP), mis nõuab, et klassil ei tohi olla rohkem kui üks vastutus [3].

Refaktoreerimiseks on välja mõeldud mitmed käitumist säilitavad väikesed võtted. Kasutades just väikseid samme, saab minimeerida defektide tekkimist ning tagada tarkvara toimimine [2]. Tabelis 1 on osa nendest võtetest jaotatud kategooriatesse koos kirjelduste ja näidetega [2], [6], [7].

Tabel 1. Refaktoreerimisvõtete kategooriad koos selgituste ja näidetega.

Kategooria nimetus	**Kirjeldus**	**Võtted**
Composing methods	Suur osa refaktoreerimisest on seotud just selle kategooriaga. Siia kuuluvad võtted, mis hõlmavad meetodite lihtsustamist.	▪ *Extract method* – eraldada olemasolevast koodist mingi osa uueks meetodiks.
Moving features between objects	Võtted, mis liigutavad funktsionaalsust ringi.	▪ *Move function* – tõsta funktsioon ühest klassist teise. ▪ *Extract class* – eralda olemasolevast klassist osa funktsionaalsust uude klassi.
Organizing data	Andmetega töötamist lihtsustavad meetmed.	▪ *Replace primitive with object* – asendada primitiivne element (vt 2.2) objektiga, kus see on kapseldatud. ▪ *Replace type code with subclasses* – luua ülemklass ja kasutada tüüpi määravate elementide asemel selle alamklasse.
Dealing with generalization	Tegeleb enamasti meetodite paigutamisega pärimishierarhias.	▪ *Form template method* – liigutada alamklasside identsed meetodid ning järjekord ülemklassi ning lasta alamklassidel nende meetodite sisu määrata.
Simplifying conditional expressions	Tingimusliku loogika lihtsustamiseks.	▪ *Decompose conditional* – teha tingimuslausest eraldi meetod.
Simplifying method calls	Liideste lihtsustamiseks mõeldud võtted, mis teevad nendele viitamise selgemaks.	▪ *Introduce parameter object* – asendada parameetrid objektiga, kus need on omadustena kokku toodud.

Mitmed refaktoreerimise rajajaid olid seotud ka disainimustritega [2]. Joshua Kerievsky ühendas refaktoreerimise disainimustritega oma raamatus „*Refactoring to Patterns*“, kus ta tõi refaktoreerimisele juurde mitmeid võtteid. Ta leidis, et mustritel põhineval refaktoreerimisel on aluseks samad põhjused, mis tavalistel madalama astme võtetel [6].

2.2 Koodi haisud

Refaktoreerimise võtteid on palju ja enne nende implementeerimist tuleb kõigepealt leida koht, mis vajab parandamist. Neid vigaseid kohti nimetatakse *code smell*'ideks ehk koodi haisudeks [2]. Lihtsustamiseks on koostatud taksonoomia ehk klassifitseerimine, mis teeb selgemaks nende omavahelised seosed ja eripärad. Tabelis 2 on osad klassid välja toodud koos selgituste ja näidetega [8].

Tabel 2. Koodi haisude klassifikatsioonid koos selgituste ja näidetega.

Klassi nimetus	**Kirjeldus**	**Koodi haisud**
Bloaters	Disainivead, mis on seotud koodi paisumisega liiga suureks.	▪ *Data clumps* – hulk andmeid, mis kuuluvad kokku ja esinevad erinevates objektides koos. ▪ *Large class* – klass, mis teeb rohkem asju kui see peaks (SRP, vt 2.1) ja on liiga suureks paisund. ▪ *Primitive obsession* – primitiivsed andmed nagu *string* tüüpi muutujad.
Couplers	Vead, mis viitavad suurele sõltuvusele objektide vahel, mis läheb vastuollu OOP printsiipidega.	▪ *Feature envy* – meetod kasutab teise klassi andmeid rohkem kui enda klassi omi.
Functional abusers	Veaohtlikud funktsionaalsused.	▪ *Loops* – standardsed *for* ja *foreach* tsüklid, mida saab asendada moodsate alternatiividega.
Dispensables	Ebavajalikud koodiosad, mida võiks eemaldada.	▪ *Data class* – objektid, mis ei oma mingit funktsionaalsust peale omaduste. ▪ *Duplication* – koodi osa, mida on dubleeritud.

Koodi haisude jaoks pakutakse erinevaid refaktoreerimise meetmeid, mis aitavad neid kõrvaldada. Tabelis 3 on näha mõningaid koodi haise ja nende vastu kasutatavaid meetmeid, millest oli lähemalt kirjas Tabelis 1, lk 13, [2], [6].

Tabel 3. Koodi haisud koos vastavate refaktoreerimisvõtetega.

Koodi hais	**Refaktoreerimise võtted**
Duplicate code	▪ Kui esineb sarnane kood kahe meetodi vahel, kasutada võtet *Extract method*, mis toob sarnase koodi ühte meetodi. ▪ Kui alamklassid teevad samu asju samas järjekorras, saab kasutada võtet *Form template method*. ▪ Kui alamklassides on sarnased meetodid, välja arvatud objekti loomise meetod, siis saab kasutada võtet *Introduce polymorphic creation with factory method*, mis loob ülemklassi, kus on sarnased meetodid kokku toodud, ja kutsub objekti loomisel *Factory method*'it, mille alamklassid üle kirjutavad.
Data Class	▪ Kui leidub funktsionaalsust, mis sobiks paremini *Data Class*'i, siis saab kasutada võtet *Extract method* või *Move function*, mis aitab selle sinna tõsta.
Long function	▪ Pikka meetodit aitab lühemaks teha *Extract method*, mis lõikab selle väiksemateks osadeks. ▪ Tingimuslausete jaoks saab kasutada võtet *Decompose conditional*, mis tõstab selle eraldi meetodiks.
Primitive obsession	▪ Kui primitiivsed andmed määravad tüüpi, kasutada võtet *Replace type code with subclasses*, mis loob nende jaoks eraldi klassid. ▪ Primitiivsete andmete korrektseks käsitlemiseks saab kasutada *Replace primitive with object*, mis kapseldab need ning pakub valideerimist. ▪ Kui primitiivseid andmeid saadetakse ringi parameetritena, saab kasutada võtet *Introduce parameter object*, mis tõstab need ühte objekti kokku.
Data Clumps	▪ Kui parameetrite hulk on liiga suur, saab kasutada võtet *Introduce parameter object*, mis tõstab need ühte objekti kokku.
Large Class	▪ Pikka klassi saab lõigata väiksemaks *Extract class*'iga (Tabel 2).
Feature Envy	▪ Kui mingi meetod töötab liiga palju mingi teise klassi andmetega, siis kasutada võtet *Move function*, et viia see sinna.
Loops	▪ Asendada tavalised *foreach* ja *for* tsüklid võttega *Replace loop with pipeline*, mis asendab need moodsamate lahendustega.

2.3 Tehisintellektiga refaktoreerimine

Tänaseks on enamik moodsaid programmeerimiskeskkondi toonud sisse automatiseeritud refaktoreerimise funktsioone, mis teevad selle protsessi kiiremaks ja töökindlamaks, nagu ReSharper ja IntelliJ IDEA [2]. Nende funktsionaalsus ei kata ära kõiki refaktoreerimise võimalusi, aga tänu suurtele keelemudelitele on tarkvaratehnika võimalused palju suuremad [9].

Üks nendest keelemudelitest on ChatGPT, mis pakub küll laiemat refaktoreerimise funktsionaalsust, kuid sellel on mõningaid nõrkusi. Kuna sisendi teksti pikkus on limiteeritud, ei saa kogu koodi kontekstiks ette anda ning seega on tal piiratud arusaam laiemast koodibaasist, mis põhjustab valesid eeldusi ja viimaks vigaseid tulemusi [4]. ChatGPT-le saab küll pärast vastust uue sisendi esitada, kuid järgnevaid vastuseid võidakse ehitada eelmiste põhjal. See tähendab, kui paluda koodifragmente refaktoreerida, siis erinevaid koodi osi võivad mõjutada varasemad vastused. Selle vastu aitab uue vestlusakna avamine, kus pole eelnevaid vastuseid, kuid sellega kaasneb uus keelemudeli nõrkus – ChatGPT vastused on väga varieeruvad. Küsides ühte sama küsimust uutes vestlusakendes võib saada iga kord erineva vastuse [9].

Refaktoreerimise efektiivsus ChatGPT-ga sõltub suuresti sisendist, kuna see on väga tundlik märksõnadele. Täpsustades kvaliteedi märksõnu on võimalik saada suuremaid muudatusi kui ilma. Lisaks saab anda juurde ka muid täpsustusi, nagu „*with no explanation*“, mis võimaldab vastuseks saada ainult koodi, ning kasutatud programmeerimiskeel, mis aitab vältida võimalikku valetuvastust keelemudeli poolt [9].

Kuna keelemudelil on suur oht teha vigu, siis riskide maandamiseks on väga olulisel kohal testid. Nagu on kirjas peatükis 2.1, lk 12, siis refaktoreerimise jaoks tulebki kasutada teste, et veenduda muudatuste õigsuses, seda nii manuaalsete muudatuste kui ka automaatsete tööriistade puhul.

Nõrkustest ja tundlikkusest sõltumata suudavad keelemudelid teha refaktoreerimist, muutes olemasolevat koodi paremaks, kuid see pole piisavalt kaugel, et seda saaks usaldada järelvalveta [9], [10]. Nagu Martin Fowler ja Kent Beck väitsid, siis ükski mõõdikute kogum ei saa inimese intuitsiooni vastu [2].

2.4 Refaktoreerimise tulemuste mõõtmine

Koodi kvaliteedi mõõtmiseks on olemas mitmeid mõõdikuid, mille abil on võimalik teha objektiivseid järeldusi koodi kvaliteedi kohta. Ühed lihtsamad on koodi suuruse mõõdikud, näiteks read, funktsioonid, klassid ja failid [11], [12].

Nagu varasemalt kirjas, siis koodi refaktoreerimise üks eesmärkidest on koodi loetavuse parandamine (vt 2.1) ning see tuleneb koodi keerukusest, mida on võimalik mõõta. Varasemalt oli kasutusel *cyclomatic complexity* ehk tsüklomaatiline keerukus, mis oli originaalselt mõeldud tuvastama, kui raske on koodi testida. Ent see ei anna aimu, kui keeruline on haldajal tegeleda koodifragmentidega, millel on täpselt sama *cyclomatic complexity*, aga erinevad süntaksi keerukused. Selleks tuleb kasutusele *cognitive complexity* ehk kognitiivne keerukus, mis annab hinnangu koodivoo kohta: kui suurt kognitiivset pingutust on programmeerijal vaja nende voogude mõistmiseks ehk kuidas arendajad tajuvad hooldatavust. Mõlemad annavad hinnangu koodi keerukusele, ent enamik arendajaid nõustub, et *cyclomatic complexity* ei suuda tuvastada koodi arusaadavust [13].

3 Metoodika

Töö refaktoreerimise protsess oli mahukas ning paremaks ülevaateks on see jaotatud erinevateks etappideks. Järgnevas peatükis antakse ülevaade, milliseid metoodikaid ja töövahendeid kasutati töö läbiviimiseks ning millistest osadest refaktoreerimise protsess koosnes.

3.1 Kasutatavad refaktoreerimise meetodid ja tööriistad

Ettevõtte X projekt, mille peal autor refaktoreerimist teostas, on ehitatud .NET 7 raamistikul C# keeles, kus on äriloogikaga seotud 1039 koodi rida, 67 klassi, 309 funktsiooni ja 48 faili. Manuaalne refaktoreerimine toimub Visual Studio Community 2022 tarkvaras ja lähtutakse Martin Fowleri raamatu „*Refactoring: Improving the Design of Existing Code*“ ning Joshua Kerievsky raamatu „*Refactoring to Patterns*“ metoodikatest [2], [6]. Autor valis Fowleri meetmed, kuna ta on refaktoreerimise üks nimekamaid eksperte ja populariseerijaid, ning Joshua Kerievsky, keda Fowler pidas üheks refaktoreerimise teerajajateks, kuna ta tõi täiendavat väärtust disainimustrite kaasamisega [2], [6], [14]. Töös kasutatakse eesmärgi täitmiseks ChatGPT keelemudelit, millele leidis autor teaduslikke allikaid sisendite koostamise osas [9]. Teisi keelemudeli alternatiive selle töö raames ei katsetata, kuna analüüs on mahukas ja eesmärgi täitmiseks piisab ühest. Täpsemalt kasutatakse ChatGPT versiooni 3.5, mis pärast teksti sisendeid annab refaktoreeritud tulemusi ning paneb need viimaks kokku. Tulemuste hindamiseks valiti SonarQube tarkvara, kuna see on üks populaarsemaid valikuid, annab järelduste tegemiseks vajalikke objektiivseid mõõdikuid ning oli ettevõttes X juba kasutusel [11], [15]. Muudatuste kontrollimiseks on projektil 6 integratsiooni- ja 56 ühiktesti, mis katavad 95,9% koodibaasist.

3.2 Tööprotsessi kirjeldus

Manuaalne refaktoreerimine koosneb kahest osast: analüüs ning realiseerimine. Analüüsi käigus otsiti lähtekoodist koodi haise ning realiseerimises leiti ja võeti kasutusele nende likvideerimiseks sobivad refaktoreerimisvõtted.

Tehisintellektiga refaktoreerimisel lähtuti artiklist, kus esitati sisendi mall, kuhu sai lisada programmeerimiskeele, kvaliteediatribuudi märksõna ja koodifragmendi. Kvaliteediatribuute oli 8 ja iga märksõna ning fragmendi jaoks avati uus vestlusaken [9]. Koodifragmendi valimisel võeti klass ja lisati kontekstiks juurde sõltuvad objektid või võimalusel nende liidesed. Nii tekkis iga klassi kohta 8 refaktoreerimise tulemust, mis pärast tehisintellekti abil kokku pandi. Vigased tulemused filtreeriti välja [10].

Mõlema refaktoreerimise käigus kontrolliti muudatuste õigsust olemasolevate testidega. Vajadusel tehti testidele täiendusi, et need sobituksid refaktoreerimise käigus muutunud liidestega.

Refaktoreerimistulemuste võrdlemiseks võeti projekti koodibaasi mõõdikud enne ja pärast manuaalset ja keelemudeliga teostatud refaktoreerimist ehk kokku 3 korda. Viimaks võrreldi mõõdetud tulemusi omavahel.

4 Manuaalne koodi analüüs

Järgmine peatükk on manuaalse refaktoreerimise esimene osa, kus analüüsitakse lähtekoodi. Siit saab ülevaate olemasolevast ettevõtte X projektist, selle struktuurist ning ilmnenud disainivigadest. Kõiki vigu detailselt ei käsitleta, vaid antakse üks näide koodi haisu kohta.

4.1 Ülevaade lähtekoodist

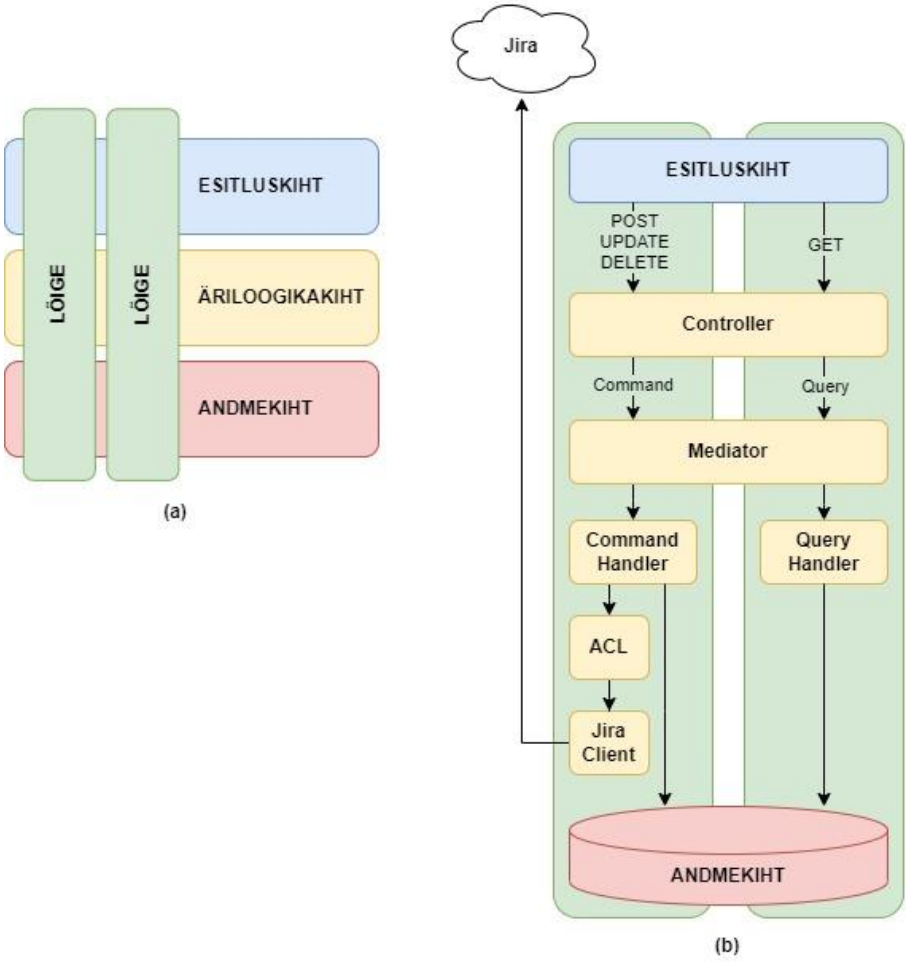
Ettevõtte X projekt loodi arveldamise protsessi automatiseerimiseks, kus on võimalik hallata hankeid, partnereid ja nendega seotud arendajaid. Võttes arvesse arendajate tunnihinnad ja logitud töötunnid projekti kohta, saab iga kuu ülevaate hangete seisust ning projektidele kulunud ressurssidest. Koodibaasi suurusest on ülevaade peatükis 3.1, lk 18.

Tegemist on monoliitsüsteemiga, kus kogu funktsionaalsus on toodud ühe rakenduse sisse. Kasutatud on kolmekihilist ja vertikaalset arhitektuuri: esitlus, äriloogika ja andme horisontaalsed kihid on omakorda vertikaalseteks lõikudeks jaotatud (Joonis 1 a). Joonisel 1 tähistab sinine esitlus-, kollane äriloogika- ja punane andmekihi elemente ning roheline vertikaalse arhitektuuri lõiget. Esitluskiht vastutab kasutajaliidese eest, äriloogikakiht süsteemi tööloogika eest ja andmekiht andmete haldamise eest. Iga vertikaalne lõige on kogu funktsionaalsus kasutajaliidesest kuni andmebaasini, mille eesmärk on maksimeerida kihtide ja minimaliseerida lõikude vahelist sõltuvust. Seda implementeeriti CQRS (*Command and Query Responsibility Segregation*) ja *Mediator* mustritega, kus CQRS jaotas CRUD meetodid kaheks: *Create*, *Update* ning *Delete* kuuluvad gruppi *Command* ja *Read* gruppi *Query* [16], [17]. Kummagi grupi jaoks tekkisid *Mediator* mustriga vastavalt *Command* ja *Query Handler*'id, mis haldasid kindla vertikaalse lõike loogikat (Joonis 1 b).

Arendajate logitud töötunde pidi hankima Jira keskkonnast. Selleks võeti kasutusele .NET-i HttpClient, mis päris neid HTTP (*HyperText Transfer Protocol*) päringutega. Tulemuste tõlkimiseks domeenimudeliteks oli *Handler*'i ja *Client*'i vahele loodud ACL

(*Anti-corruption Layer*), mis aitas hoida domeenist eraldi välissüsteemide loogikat ja teha rakendus seeläbi vähem sõltuvaks välistest teguritest (Joonis 1 b) [18].

Esitluskiht suhtleb äriloogikakihi *Controller*'itega üle HTTP päringute ning need omakorda läbi *Mediator*'i vastavate *Handler*'itega. Refaktoreerimise käigus ei muudeta esitluskihti, et koodi visuaalne käitumine säiliks (vt 2.1).



Joonis 1. Projekti arhitektuuri diagrammid: (a) üldisem diagramm, (b) detailsem diagramm, kus ACL on *Anti-corruption Layer*.

4.2 Analüüsimine

Töö autor suutis olemasolevast koodist leida 19 disainiviga ehk koodi haisu, mille loetelu on Tabelis 4. Problemaatilised elemendid olid seotud Jira kommunikatsiooni vertikaalse lõikega (vt 4.1). Ülejäänud lõigud olid suhteliselt lühikesed tänu kasutatud arhitektuurile, mistõttu ei leidnud töö autor neis disainivigu. Kõikidest haisudest olid 9 *Bloater*, 7 *Dispensable* ja 3 *Functional abuser* (vt 2.2).

Tabel 4. Koodi haisude projektis esinemiste ülevaade.

Kategooria	**Koodi hais**	**Esinemiste arv**
Bloater	*Long function*	5
	Primitive obsession	3
	Data clumps	1
Dispensable	*Duplicate code*	5
	Data class	2
Functional abusers	*Loops*	3

4.2.1 Paisutajad

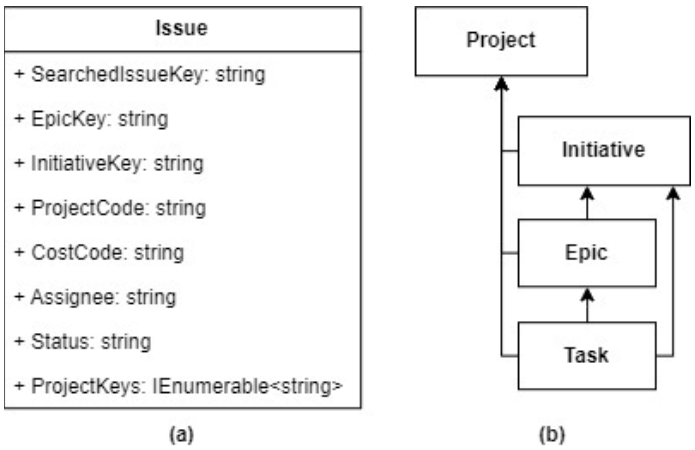
Paisutajate ehk *Bloater* haisude seast esines kõige rohkem *Long function*'eid. Paljud meetodid tegid rohkem kui ühte asja, mis läheb vastuollu SRP-ga (vt 2.1). Joonisel 2 on näha *Command Handler*'it, mis haldab domeenimudeleid ehk arendajate raporteid ja nendega seotud projekte. Sisu on asendatud kommentaaridega, mis kirjeldavad selle sisu. See meetod on 49 rida pikk ja teeb 9 erinevat asja. Selliseid disainivigu esines viies meetodis.

```
public async Task<Unit> Handle(Command command, CancellationToken
       cancellationToken)
{
       // Get Developer models from database
       // Select Jira user keys from Developer models
       // Ask ReportsService class to fetch Reports from Jira by user keys
       // Add new Reports to database
       // Ask ProjectService class to fetch Projects from Jira for Reports
       // Get existing Projects from database
       // Merge existing Projects with new ones
       // Add new Projects to database
       // Save changes
       // Return Unit.Value
}
```

Joonis 2. *Long function* koodi haisu näide.

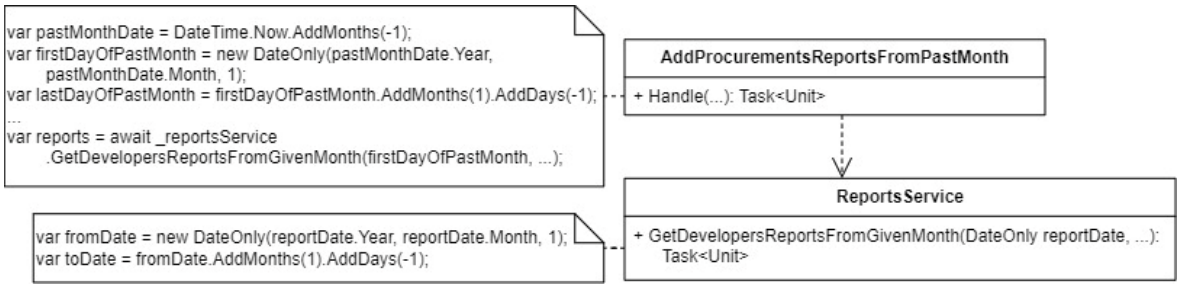
Jirast tuleb mitmeid *string* tüüpi väärtuseid, mistõttu tekkis ACL-i (vt 4.1) väga palju primitiivseid elemente. Joonisel 3 a on näha *Issue* DTO-d (*Data Transfer Object*). *Issue* on Jira keskkonna objekt, mille alla on arendaja tööd loginud. See on üldine klass, mis võib Jira kontekstis tähendada mitut erinevat tüüpi: *Task*, *Epic*, *Initiative* või *Project*. Variandid moodustavad hierarhia, kus kõige all olev *Task* saab olla seotud *Project*, *Epic*

ja *Initiative* tüüpidega, *Epic* seevastu *Project*'i ja *Initative*'iga ning *Initative* vaid *Project*'iga (Joonis 3 b). *Project* tüübil pole kõrgemaid tüüpe, aga sellel on *ProjectCode, CostCode, Status* ja *Assignee* väärtused, mis on teiste jaoks tühjad. Kõik variandid kaardistatakse *Issue* objektile, kus puuduolevad *string* väljad jäetakse tühjaks. Kõik see loogika jääb objekti omaduste taha peitu, mis teeb selle primitiivseks.



Joonis 3. (a) *Primitive obsession* koodi haisu näide, (b) peidetud objekti hierarhia.

Primitiivseid objekte leidus veel teisteski klassides, kus olid kasutusel domeeniobjektid, nagu kulukohakood ja kuupäevade vahemik, kus viimane oli seotud ka teise koodi haisuga – *Data clump*. Joonisel 4 on näha kahte klassi, kus mõlemal on kasutusel samad elemendid: möödunud kuu algus- (*firstDayOfPastMonth* ja *fromDate*) ja lõppkuupäev (*lastDayOfPastMonth* ja *toDate*). Need elemendid on omavahel seotud, kuna annavad kokku ajavahemiku ja esinevad erinevates klassides koos. Seega peab neid omavahel tugevemalt ja selgemalt siduma [2].



Joonis 4. *Data clump* koodi haisu näide.

On näha, et alguskuupäev antakse *ReportService* klassi meetodile argumendina kaasa, kus seda omakorda samade kuupäevade loomiseks kasutatakse. Seega on siin lisaks teine koodi hais: *Duplication*, millest on lähemalt kirjas järgmises peatükis.

4.2.2 Asendatavad

Teine kõige sagedasem koodi hais oli *Duplication*, mis esines viies erinevas meetodis. Joonisel 5 on näha koodi osa, mida on dubleeritud: nii *outwards* kui ka *inwards* projektid otsitakse *response* objektist ühtemoodi. Abiks on võetud küll abimeetod *CheckIfIssueIsActive*, kuid see ei eemalda kogu duplikatsiooni.

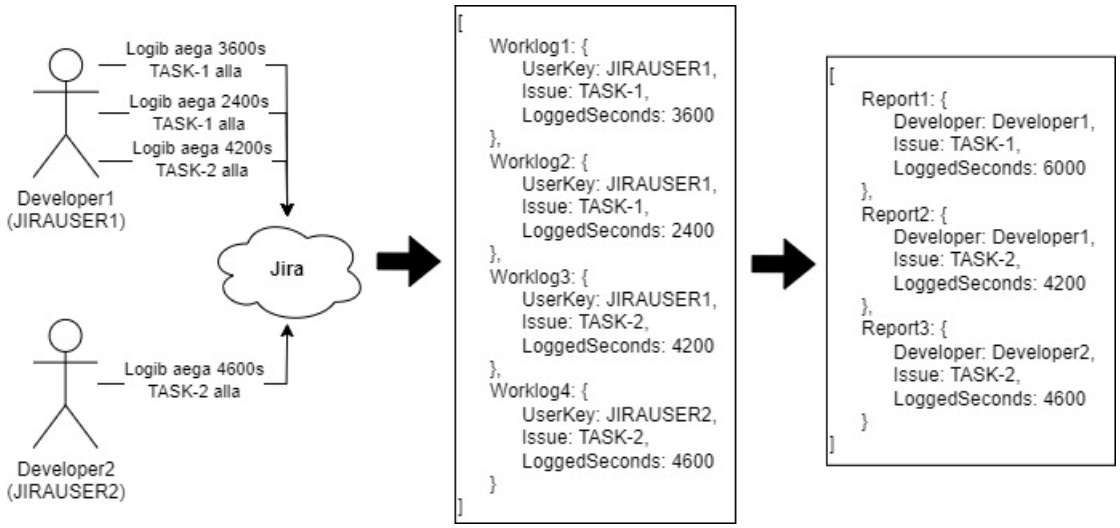
```
public async Task<Response> GetIssuesDetails(Query query,
       CancellationToken cancellationToken)
{
       // Get response from Jira

       var outwardsProjects = response.Fields.RelatedProjects
              .Where(Project => Project.OutwardsRelatedProject != null &&
                     CheckIfIssueIsActive(Project.OutwardsRelatedProject))
              .Select(Project => Project.OutwardsRelatedProject!.Key);
       var inwardsProjects = response.Fields.RelatedProjects
              .Where(Project => Project.InwardsRelatedProject != null &&
                     CheckIfIssueIsActive(Project.InwardsRelated Project))
              .Select(Project => Project.InwardsRelatedProject!.Key);

       // Create Issue
}
```

Joonis 5. *Duplication* koodi haisu näide.

Joonisel 3, lk 23, oli välja toodud *Issue* DTO, mille sees polnud midagi muud kui primitiivsed omadused. Tegemist on *Data class* haisuga, kus objekt üksi ei oska midagi teha. DTO põhimõte ongi vaid andmete manustamine ja transportimine, aga *Issue* objekt ei pea olema DTO ja saab olla palju sisukam.

4.2.3 Funktsionaalsuste kuritarvitajad

Worklog on arendaja Jira keskkonna ajalogiobjekt, mis tekib iga logimise tagant (Joonis 6). Iga objekti küljes on arendaja Jira kasutajavõti, seotud *Issue* (vt Joonis 3 a, lk 23) ning logitud aeg. Logide kollektsioonis võib esineda ühte *Issue* objekti mitu korda erineva arendaja ja ajaga. *Report*, mida *Worklog* objektist tahetakse saada, peab tekkima iga arendaja unikaalse *Issue* kohta, kus sama *Issue* logide tunnid on kokku liidetud. Kogu seda protsessi on näha Joonisel 6, kus 4 ajalogimisest tekib 3 *Report* objekti.



Joonis 6. Arendaja ajalogimisest *Report* objekti tekkimine.

Joonisel 7 on näha, et *Report* objekti loomise loogikat lahendatakse *foreach* tsükliga, kuid kahe kollektsiooni omavaheliseks grupeerimiseks, mitme elemendi koostamiseks tsüklis ja nende kogumiseks on .NET raamistikus olemas eraldi meetodid [19]. Tegemist on *Loop* koodi haisuga ja tsükli asendamine olemasolevate lahendustega teeks selle koodifragmendi lühemaks.

```
var worklogDeveloperGroups = worklogDTOs.GroupBy(worklog =>
       worklog.DevelopersUserKey);
var reports = new List<Report>();

foreach (var worklogDeveloperGroup in worklogDeveloperGroups)
{
    // Select current groups Developer
    // Select current groups distinct Worklogs by Issue

    foreach (var distinctWorklog in distinctWorklogDTOs)
    {
        // Select Worklogs that are related to Developer and Issue
        // Sum up all related Worklogs logged hours
        // Create Report
        // Add Report to list
    }
}
```

Joonis 7. *Loop* koodi haisu näide.

4.2.4 Muud disainivead

Kõiki disainivigu ei saa üheselt kategoriseerida mingi kindla koodi haisu alla. Kategooriate definitsioonide ja haisude järgi on siiski võimalik saada selleks suund [2].

Joonisel 7, lk 25, oli näha koodifragmenti klassist *ReportsService*, kus tsüklis käsitleti *Worklog* DTO kollektsiooni, mis päritakse *WorklogManager* klassist. Nagu oli kirjas, siis tagastatud kollektsioonis võib esineda objekte, mis on seotud sama *Issue*'ga. Dubleeritud logisid hakatakse ühendama suures ja pesastatud ehk kahekordses tsüklis, mis hakkab oma olemuselt muutuma *Bloater* koodi haisuks, kus kood paisub liiga suureks. Lisaks peaks olema *Worklog* DTO-de haldamine, nagu nimigi ütleb, *WorklogManager* klassi vastutusala ehk see viitab *Coupler* haisule. Kindlat haisu ei suutnud töö autor määrata, kuid on olemas refaktoreerimisvõtted, mis aitavad seda olukorda lahendada. Praegust olukorda ning eelnevates peatükkides käsitletud koodi haise hakataksegi järgmises peatükis lahendama.

5 Manuaalne refaktoreerimine

Järgnev peatükk on manuaalse refaktoreerimise teine osa, kus keskendutakse esimeses osas (vt 4) leitud disainivigade likvideerimisele. Peatükis loetletud refaktoreerimisvõtted ja seoseid vastavate koodi haisudega on toodud välja Tabelis 3, lk 15. Iga refaktoreerimisvõtte kohta on esitatud üks näide.

5.1 Ülevaade realisatsioonist

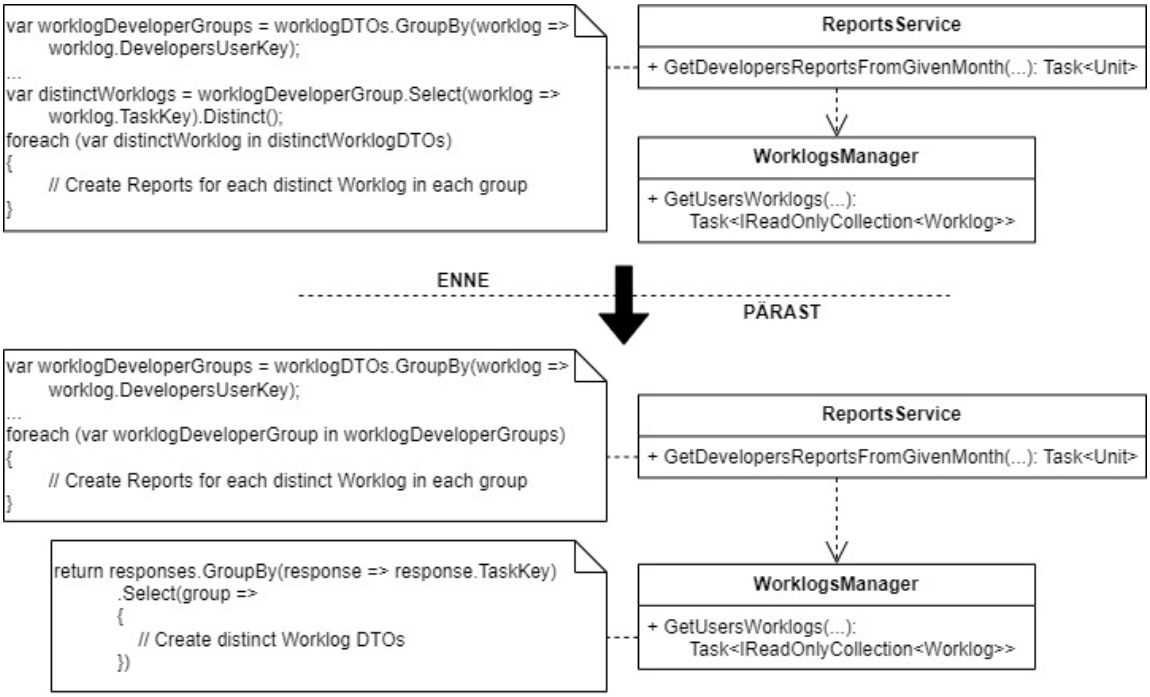
Töö autor kasutas projekti refaktoreerimisel kokku 25 võtet. Neist 8 olid *Composing methods*, 4 *Organizing data*, *4 Moving features between object*, 4 *Dealing with generalisation*, 3 *Simplifying conditional expressions* ja 2 *Simplifying method calls* (vt 2.1). Tabelis 5 on näha kokkuvõtet kasutatud võtetest. Autoril kulus disainivigade analüüsimisele (vt 4) ja tabelis loetletud refaktoreerimisvõtete realiseerimisele mitu täispikka tööpäeva.

Tabel 5. Projektis kasutatud refaktoreerimisvõtete ülevaade.

Kategooria	**Võte**	**Kasutuste arv**
Composing methods	*Extract method*	5
	Replace loop with pipeline	3
Organizing data	*Replace type code with subclasses*	2
	Replace primitive with object	2
Moving features between object	*Move function*	2
	Extract class	2
Dealing with generalisation	*Form template method*	2
	Introduce polymorphic creation with factory method	2
Simplifying conditional expressions	*Decompose conditional*	3
Simplifying method calls	*Introduce parameter object*	2

5.1.1 Funktsionaalsuste liigutamine objektide vahel

Peatükis 4.2.4, lk 26, oli kirjeldatud olukorda, kus esines *Bloater* ja *Coupler* laadi disainiviga. Töö autor lahendas selle *Move function* võttega, mis on sobilik vastutuse jaotamiseks ja kapseldamiseks [2]. Joonisel 8 on näha, kuidas objektide unikaalsuse loogika liigub *ReportService* klassist *WorklogManager* klassi. Sellisel moel vähenes juba suureks paisunud *foreach* tsükkel (Joonis 7, lk 25) ning suurenes kapseldamine, kuna vastutus liigutati *WorklogManager* klassi juurde, mis tegeles kollektsiooni tagastamisega.

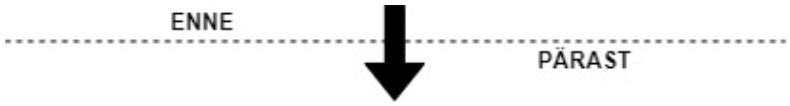


Joonis 8. *Move function* refaktoreerimisvõtte näide.

5.1.2 Meetodite koostamine

Kõige rohkem esines lähtekoodis *Bloater* koodi haise (vt 4.2). Joonisel 2, lk 22, oli näha väga pikka meetodit, mis tegeles üheksa erineva ülesandega. *Long function* koodi haisu vastu aitab *Extract function*, mis aitab üle paisunud meetodi väiksemateks tükkideks teha. Joonisel 9 on näha, kuidas töö autor võttis olemasolevast meetodist erinevad ülesanded ja jaotas need nelja erineva abimeetodi vahel laiali: *GetAllRelatedDevelopers*, *AddNewReportsToPastMonth*, *AddNewReportsProjects* ja *MergeNewProjectsExistingOnes*. Tulemusena vähenes *Handler* meetod 37 rea võrra.

```
public async Task<Unit> Handle(...)
{
	// Get Developer models from database
	// Select Jira user keys from Developer models
	// Ask ReportsService class to fetch Reports from Jira by user keys
	// Add new Reports to database
	// Ask ProjectService class to fetch Projects from Jira for Reports
	// Get existing Projects from database
	// Merge existing Projects with new ones
	// Add new Projects to database
	// Save changes
	// Return Unit.Value
}
```

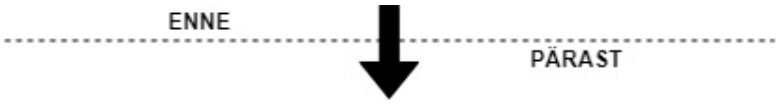



```
public async Task<Unit> Handle(...)
	// Call GetAllRelatedDevelopers to get related Developer Jira user keys
	// Call AddNewReportsToPastMonth with Jira user keys to get Report models
	// Call AddNewReportsProjects with Report models
	// Save changes
	// Return Unit.Value

private async Task<string[]> GetAllRelatedDevelopers(...)
	// Get Developer models from database
	// Select Developers Jira user keys from Developer models

private async Task<ICollection<Report>> AddNewReportsToPastMonth(...)
	// Ask ReportsService to fetch Reports from Jira
	// Add new Reports to database

private async Task AddNewReportsProjects(...)
	// Ask ProjectService to fetch Projects for all new Reports from Jira
	// Call MergeNewProjectsExistingOnes
	// Add new Projects to database

private async Task<ICollection<Project>> MergeNewProjectsExistingOnes(...)
	// Get existing Projects from database
	// Merge existing Projects with new ones
```

Joonis 9. *Extract function* refaktoreerimisvõtte näide.

Peatükis 5.1.1, lk 28, kasutas autor *Move function* võtet, mis tegi *ReportService* klassi *foreach* tsüklit lühemaks, mille tulemust on näha Joonisel 10 esialgses versioonis. Tsükli asendamiseks kasutati *Replace loop with pipeline* võtet, mis tõi sisse .NET raamistiku *GroupJoin* ja *Select* meetodid [19]. Uus lahendus kasutab raamistiku poolt pakutud *query expression*'eid ja on kompaktsem.

```
var worklogDeveloperGroups = worklogDTOs.GroupBy(worklog =>
      worklog.DevelopersUserKey);
var reports = new List<Report>();

foreach (var worklogDeveloperGroup in worklogDeveloperGroups)
{
    // Select current groups Developer
    // Select current groups Worklog DTOs
    foreach (var worklog in WorklogDTOs)
    {
        // Sum up all related Worklogs logged hours
        // Create Report
        // Add Report to list
    }
}
return reports;
```

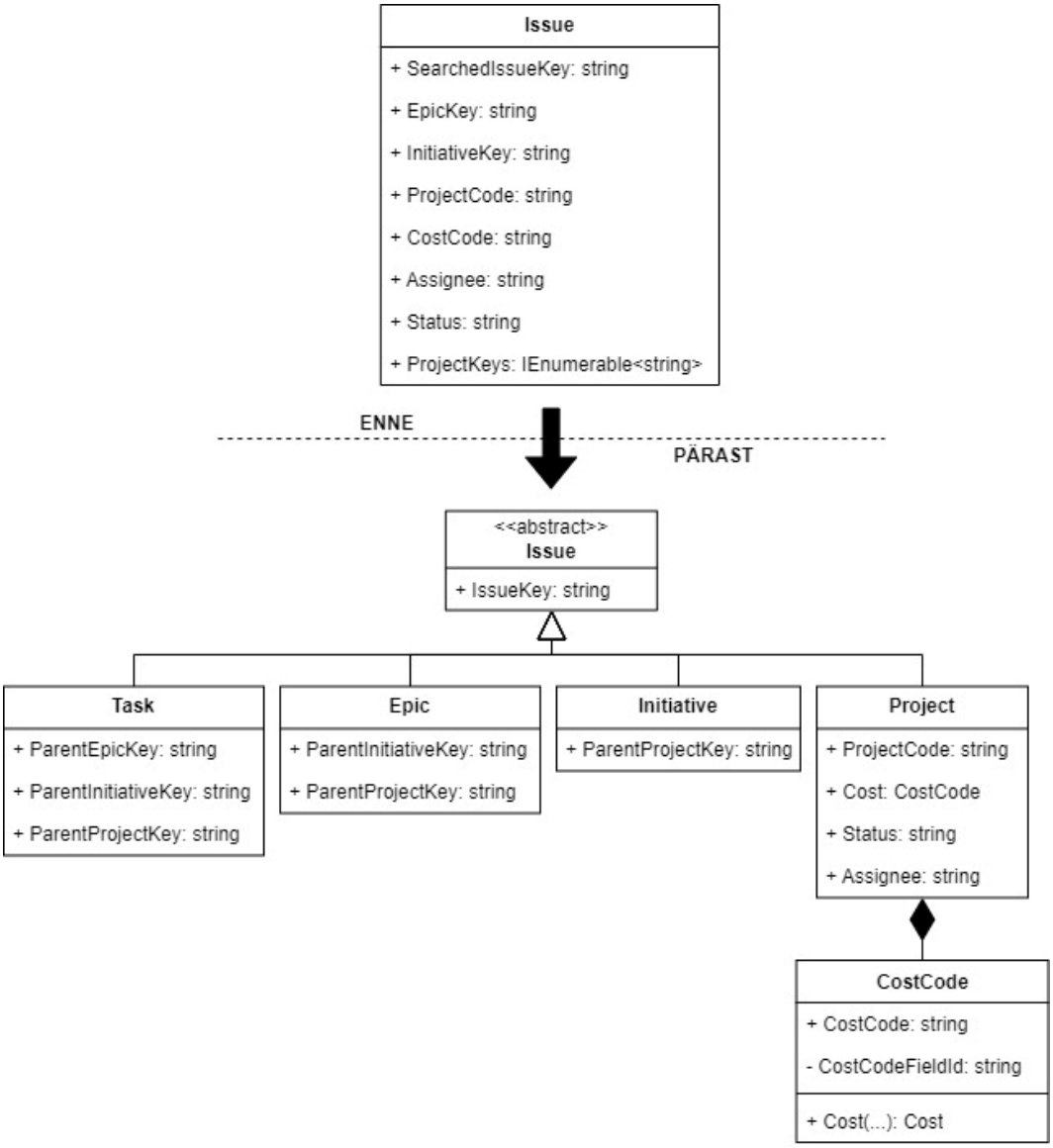



```
return developers.GroupJoin(worklogs,
             developer => developer.UserKey,
             worklog => worklog.DevelopersUserKey,
             (developer, developerWorklogs) => GenerateDevelopersReports(...))
      .SelectMany(reports => reports)
      .ToList();

private static IEnumerable<Report> GenerateDevelopersReports(...) =>
      worklogs.Select(worklog =>
      {
             // Create report
      );
```

Joonis 10. *Replace loop with pipeline* refaktoreerimisvõtte näide.

5.1.3 Andmete organiseerimine

Joonisel 3 a, lk 23, oli näha *Issue* objekti, mis koosnes vaid *string* tüüpi omadustest, mis varjasid Jira tüübi loogikat (Joonis 3 b, lk 23). Tegemist on primitiivse objektiga, mille vastu kasutas töö autor *Replace primitive with object* ja *Replace type code with subclasses* võtteid. Joonisel 11 on näha, kuidas ühest objektist tekib viis erinevat tüübiklassi, mis näitavad selgemalt erinevate *Issue*'de eripärasid. *Task* tüübil saab olla seos *Epic*, *Initiative* ja *Project* tüüpidega, aga *Epic*'ul *Initiative* ja *Project* tüüpidega. See loogika oli varasemalt peidus. Lisaks tekkis eraldi objekt *Project* omadusele *CostCode*. Varasemalt vastutas selle valideerimise eest *IssuesManager*, milles objekt asus, kuid nüüd on tõstetud see vastutus omadusega kokku, mis suurendab kapseldamist ja vähendab primitiivsust.

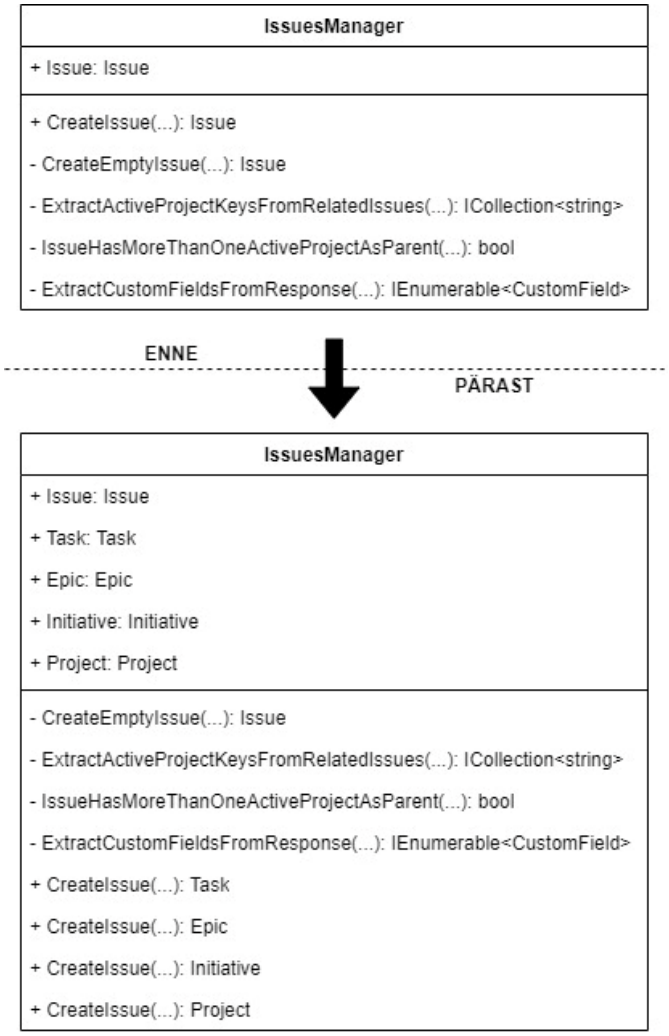


Joonis 11. *Replace primitive with object* ja *Replace type code with subclasses* refaktoreerimisvõtete näide.

Koodi terviklikkuse säilitamisel tekkis eelneva võttega uus disainiviga. Algne *IssuesManager* klass, mis tegeles *Issue* loomisega, pidi sellisel juhul mitut erinevat tüüpi looma, mille tulemusena tekkis duplikatsioon ja kood paisus suureks. Selle lahendamiseks võttis autor ette teise võtte, millest on lähemalt kirjas järgnevas peatükis.

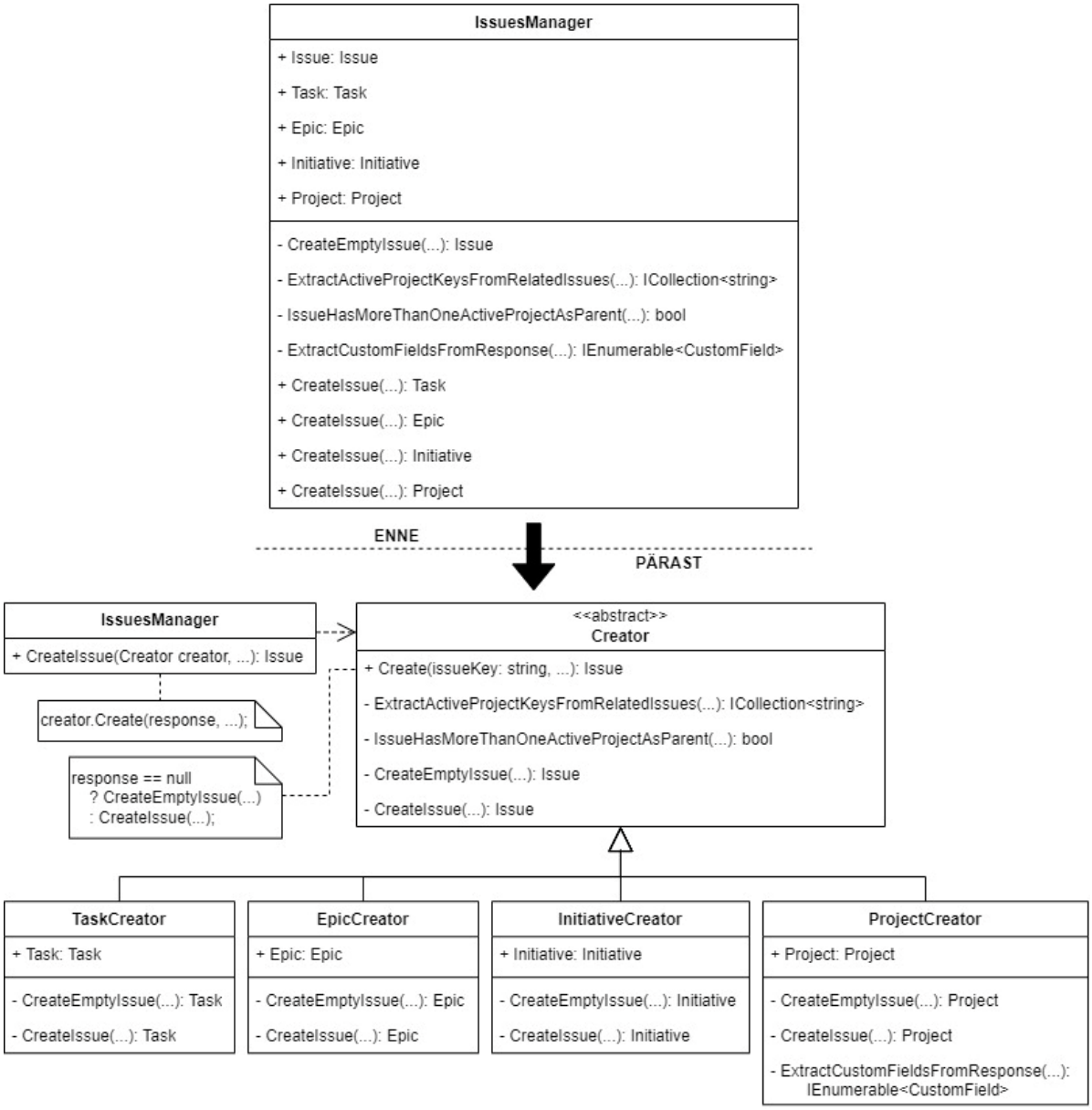
5.1.4 Üldistuste haldamine

Eelnevas peatükis oli kirjas primitiivsest *Issue* klassist, mille taga peitus Jira loogika. Primitiivsuse vähendamiseks kasutati võtet, mis tõi sisse alamklassid (Joonis 11, lk 31), mistõttu tekkisid sarnased klasside loomise meetodid, mida on näha Joonisel 12.



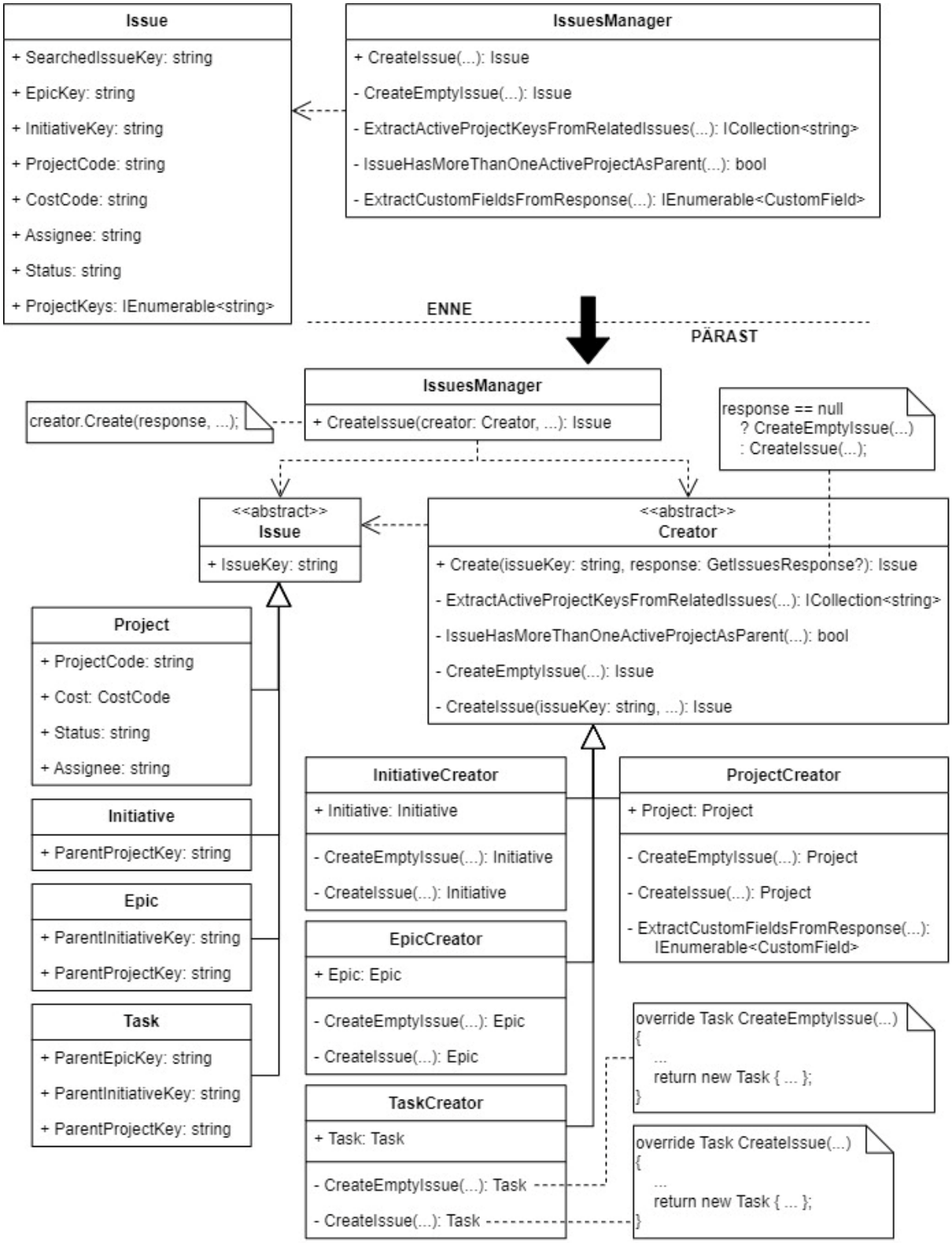
Joonis 12. Refaktoreerimise käigus paisunud klass.

Töö autor kasutas selle vastu *Introduce polymorphic creation with factory method* võtet, mis tõi sisse *Factory method* disainimustri, mille puhul tekitati abstraktne *Creator* liides *Issue* tüüpide loomiste jaoks ning lasti alamklassidel muuta loodavate objektide tüüpi. Lisaks kasutati *Form template method* võtet, mis viis tüübi koostamise sammud ja nende järjekorra ülemklassi ning lasi alamklassidel sammud üle kirjutada [17]. Joonisel 13 on näha tulemust.



Joonis 13. *Introduce polymorphic creation with factory method* ja *Form template method* refaktoreerimisvõtete näide.

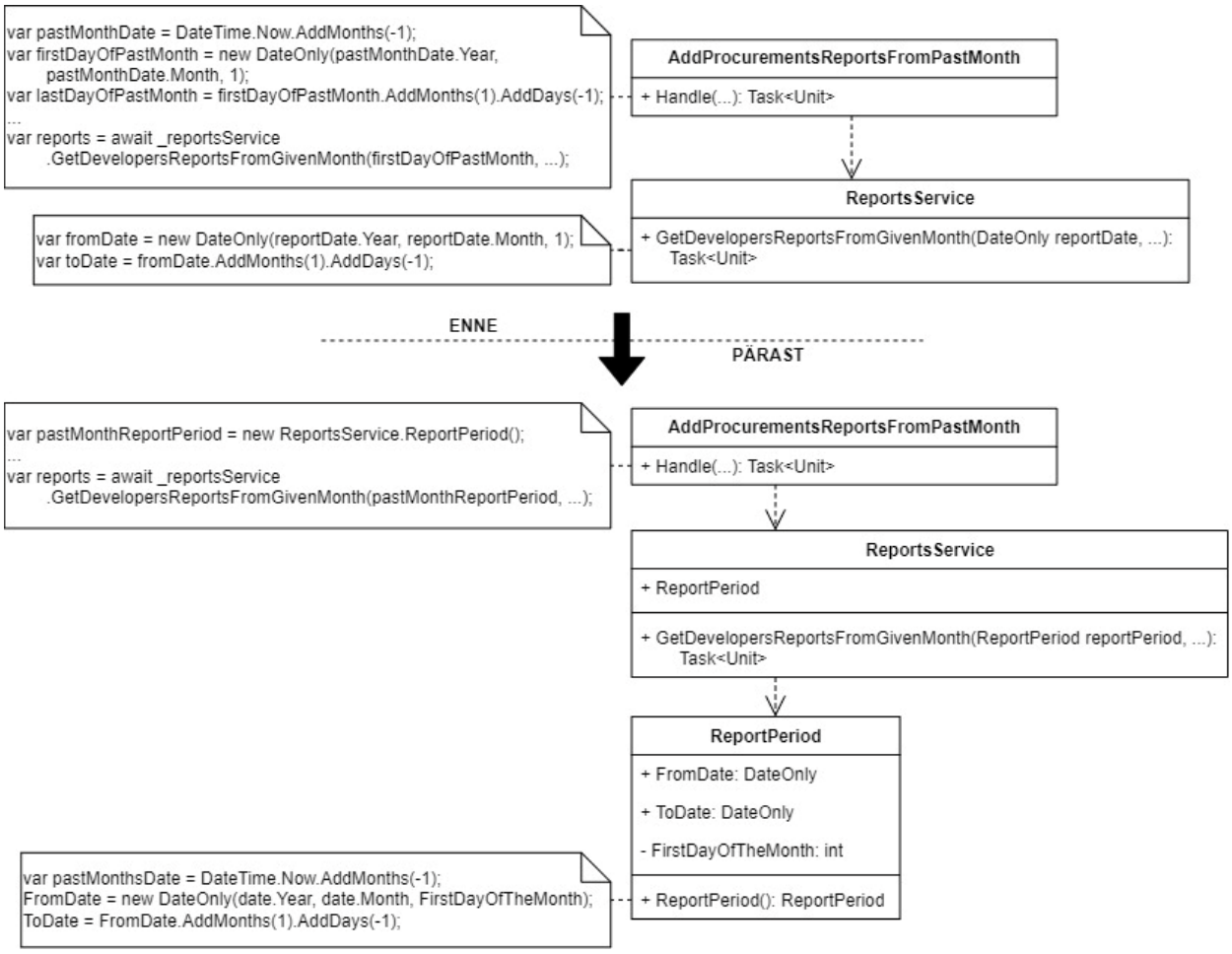
Peatükis 5.1.3, lk 30, ja praeguses käsitleti kokku nelja erinevat refaktoreerimisvõtet: *Replace primitive with object*, *Replace type code with subclasses*, *Introduce polymorphic creation with factory method* ja *Form template method*. Nende tulemusel tehti suur muudatus, mida refaktoreerimine endast kujutabki (vt 2.1). Pärast igat võtet säilis koodi toimimine ning kokkuvõttes kaotati mitu haisu. Kõigi nende võtete tulemust on näha Joonisel 14.



Joonis 14. *Replace primitive with object*, *Replace type code with subclasses*, *Introduce polymorphic creation with factory method* ja *Form template method* refaktoreerimisvõtete näide.

5.1.5 Funktsionaalsuste lihtsustamine

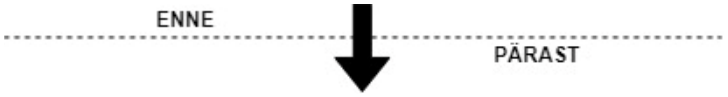
Analüüsi käigus leiti peatükis 4.2.1, lk 22, *Data clump*, *Primitive obsession* ja *Duplication*. Need tulenesid kahest omavahel seotud elemendist: kuu algus- ja lõppkuupäev. Autor kasutas nende vastu *Introduce parameter object* ja *Replace primitive with object* võtteid. Tulemusena tekkis *ReportPeriod* objekt (Joonis 15), mis tõi kokku perioodi omadused ning nendega seotud valideerimisloogika, mistõttu suurenes kapseldamine ja vähenes primitiivsus. Lisaks määrati see *ReportService* klassi meetodi parameetriks ja tulemusena kadus dubleeritud perioodiloomise koodifragment, kuna periood oli juba olemas.



Joonis 15. *Introduce parameter object* ja *Replace primitive with object* refaktoreerimisvõtete näide.

Joonisel 16 on näha ülemises koodifragmendi variandis tingimusloogikat. Meetod on pikk: see koosneb 25 reast ja omab mitut erinevat ülesannet. Selle lihtsustamiseks kasutas autor *Decompose conditional* võtet, mis viis tingimuslause eraldi meetodisse. Algne meetodi sisu läks lühemaks ning tingimusloogikale on pandud selgitav nimetus.

```
public async Task<List<Project>> GetCorrespondingProjectsForReports(...)
{
	...
	issueHierarchies.RemoveAll(hierarchy =>
		hierarchy.ProjectKeys.Distinct().Count() != 1);
	...
}
```




```
public async Task<List<Project>> GetCorrespondingProjectsForReports(...)
{
	...
	issueHierarchies.RemoveAll(hierarchy =>
		!HierarchyHasExactlyOneActiveProject(hierarchy));
	...
}

private static bool HierarchyHasExactlyOneActiveProject(Hierarchy hierarchy) =>
	hierarchy.ProjectKeys.Distinct().Count() == 1;
```

Joonis 16. *Decompose conditional* refaktoreerimisvõtte näide.

6 Tehisintellektiga refaktoreerimine

Järgmine peatükk on ChatGPT keelemudeliga refaktoreerimisest. Esitatakse ülevaade kasutatud sisendimallidest, genereeritud lahenduste võtetest ja mõningatest probleemidest.

6.1 Ülevaade realisatsioonist

Töö autor võttis keelemudeliga refaktoreerimise aluseks teadusartikli, mis pakkus keelemudeli jaoks sisendimalli: „*With no explanation refactor the* [programmeerimiskeel] *code to improve its quality and* [kvaliteedi märksõna]: [kood]“. Kvaliteedi märksõnu oli 8: *performance*, *complexity*, *coupling*, *cohesion*, *design size*, *readability*, *reusability* ja *understandability*. Iga kvaliteedimärksõna kohta koostati 8 sisendit, millesse jäi veel kood lisada. Iga sisend sisestati uude vestlusaknasse, et varasemad lahendused ei mõjutaks järgmisi (vt 2.3). Nii tekkis iga fragmendi kohta 8 tulemust [9]. Nagu peatükis 3.2, lk 18, kirjas, koosnes koodifragment klassist ning kontekstiks lisati sõltuvad objektid või võimalusel nende liidesed.

Tulemuste ühendamiseks katsetas autor erinevaid allika järgi koostatud malle [9]. Nende seast säilitas kõige rohkem muudatusi järgmine sisendimall: „*I have 8 variants of the same* [programmeerimiskeel] *class*. *Merge them into one*: [kood]“. Prooviti ka detailsemaid malle, milles oli lisaks kvaliteedimärksõnu, kuid nendega läksid osad muudatused kaduma.

6.2 Realiseerimine

Keelemudel suutis sisendite järgi teha muudatusi vaid etteantud klassi raames. See tähendab, et klassidevahelisi võtteid ei pakutud. ChatGPT kasutas kokku 55 võtet, millest moodustub 12 unikaalset. Autor jaotas need võtted nelja erinevasse kategooriasse, mida on Tabelis 6 näha koos kirjelduste ja kasutuste arvuga. Kui manuaalsele refaktoreerimisele (vt peatükke 4 ja 5) kulus autoril mitu täispikka tööpäeva, siis keelemudeliga võttis see aega ühe päeva.

Tabel 6. ChatGPT poolt pakutud refaktoreerimisvõtted koos kirjeldustega.

Kategooria	**Võtted**	**Kasutuste arv**
Struktuuri optimeerimine	Muutis süntaksit – tõi juurde või eemaldas *lamda expression*'eid ehk „=>“ või loogelisi sulge ehk „{}“	16
	Tegi juurde abimeetodi – liigutas mingi osa funktsionaalsusest eraldi meetodisse	7
	Eemaldab duplikatsiooni – eemaldas dubleeritud koodifragmendi	2
Tõhususe parandamine	Muutis tühja väärtuse loogikat – lisas vaikimisi väärtusi või muutis liideste parameetreid, et need lubaksid või ei lubaks tühjasid väärtusi	5
	Täpsustas tagastustüüpe – lisas *return*'i juurde tagastatava tüübi, kui see oli puudu	3
	Muutis tingimusloogikat – täpsustas tingimuslauset või vähendas pesastamist ehk koodi sisaldumist teise sees	3
	Muutis tsüklit – tegi tsükli efektiivsemaks	1
Loetavuse parandamine	Muutis elemendi nime – andis elemendile selgema nimetuse	4
	Tõi juurde muutujaid – lõi funktsioonide jada lahti ja tõi asemele uued muutujad pärast igat funktsiooni, mida kasutas järgmise väljakutsumiseks	2
Programmeerimiskeele spetsiifilised muudatused	Eemaldas *required* objekti omadustelt – kaotas *required* modifikaatori objekti omaduse tagant	10
	Vahetas *async* meetodi tavalise vastu – vahetas *AddAsync* meetodi *Add*'iga	1
	Vahetas tüübimääramise viisi – vahetas arvutuses tüübi *double* täpsustaja *foalting-point* „f“-iga	1

Tulemuste seas tekkis kuus erinevat viga, millest neli olid seotud liidese sobimatusega. Näiteks tahtis ChatGPT muuta *Collection* tüüpi *List* tüübiks, mis ei sobinud meetodi liidesega. Ülejäänud kaks viga tulenesid C# programmeerimiskeele ebakorrektsest kasutamisest. Ühtegi tõsist äriloogilist viga ei tekkinud ning autor filtreeris ilmnenud vead välja.

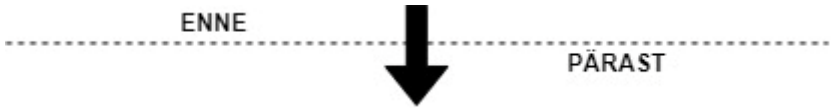
Kaheksa erineva tulemuse seas esines kohati ühe väiksema fragmendi kohta mitu muudatuse varianti, kus mõlemat korraga ei saanud kasutada. Autor tahtis vältida manuaalset valimist nende seast ja seepärast lasi seda teha tehisintellektil.

ChatGPT suutis leida mõned muudatused, mille peale töö autor ise ei tulnud. Nimelt peatükis 4.2.3, lk 24, oli juttu *foreach* tsüklist, mis lõi *Report* objekte. Autoril oli selle disainivea lahendamiseks teine lähenemine (vt 5.1.2) kui keelemudelil, millest tuleb lähemalt juttu peatükis 6.2.2, lk 40. ChatGPT tõi sisse ka keelespetsiifilise muudatuse asünkroonse meetodiga, millest tuleb juttu peatükis 6.2.4, lk 43, mille peale autor ei osanud tulla. See näitab, et keelemudel võib leida lähenemisi, mille peale arendaja ei oska koheselt tulla.

6.2.1 Struktuuri optimeerimine

Kõige rohkem muutis keelemudel koodifragmentide süntaksit. Joonisel 17 on näha *WorklogManager* ja *IssueManager* klasside konstruktoreid. *Worklog* klassil asendati *lamda expression* „=>“ loogeliste sulgudega, aga *Issue* klassil viidi kõik ühele reale. See näitab keelemudeli nõrkust, et see ei anna koguaeg ühtset vastust (vt 2.3).

```
public WorklogsManager(IJiraClient jiraClient) =>
       _jiraClient = jiraClient;

public IssuesManager(IJiraClient jiraClient) =>
       _jiraClient = jiraClient;
```

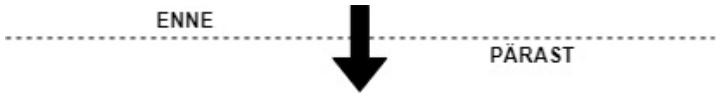



```
public WorklogsManager(IJiraClient jiraClient)
{
       _jiraClient = jiraClient;
}

public IssuesManager(IJiraClient jiraClient) => _jiraClient = jiraClient;
```

Joonis 17. Keelemudeli süntaksi muutmise refaktoreerimisvõtte näide.

Leidus ka variante, kus murti ridu või eemaldati loogelised sulud, millest viimast on näha Joonisel 18 pärast tingimusloogikat. Lisaks on seal veel näha abimeetodite *GetActiveRelatedProjects* ja *ExtractCostCode* loomist ning duplikatsiooni eemaldamist. Selle sama duplikatsiooni tuvastas ka töö autor peatükis 4.2.2, lk 24.

```
if (response == null)
{
	return new()
	{
		SearchedIssueKey = query.IssuesKey
	};
}

var outwardsProjects = response.Fields.RelatedProjects
	.Where(Project => Project.OutwardsRelatedProject != null &&
		CheckIfIssueIsActive(Project.OutwardsRelatedProject))
	.Select(Project => Project.OutwardsRelatedProject!.Key);
var inwardsProjects = response.Fields.RelatedProjects
	.Where(Project => Project.InwardsRelatedProject != null &&
		CheckIfIssueIsActive(Project.InwardsRelated Project))
	.Select(Project => Project.InwardsRelatedProject!.Key);

var costCode = response.Fields.CostCode != null
	? JsonSerializer.Deserialize<CostCodeResponse>(...)?.Group.Fields
		.FirstOrDefault(...)?.Value.Trim() ?? string.Empty
	: string.Empty;
```




```
if (response == null)
	return new Response { SearchedIssueKey = query.IssuesKey };

var activeOutwardsProjects = GetActiveRelatedProjects(...);
var activeInwardsProjects = GetActiveRelatedProjects(...);

var costCode = response.Fields.CostCode != null
	? ExtractCostCode(response.Fields.CostCode)?.Trim() ?? string.Empty
	: string.Empty;
```

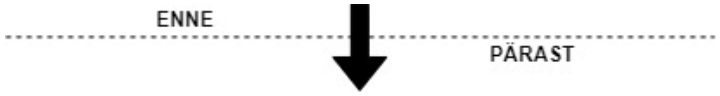
Joonis 18. Keelemudeli ridade ja abimeetodiga duplikaadi vähendamise näide.

Lisaks oli Joonisel 18 näha, et täpsustati *return* tüüpi. Sellest ja muudest tõhusust parandavatest võtetest tuleb juttu järgmises peatükis.

6.2.2 Tõhususe parandamine

Joonisel 19 on näha, kuidas täpsustatakse *return* tüüpi ehk *new* asemel on nüüd *new PaginatedResult<Result>*. Lisaks on näha, et meetodi *Search* väljakutsumisest eemaldati *null-forgiving* märk „!“ ja selle liidest muudeti: *SearchString* parameeter võib olla nüüd tühi väärtus.

```
public async Task<PaginatedResult<Result>> Handle(...)
{
	var filteredQuery = Search(request.SearchString!, ...);
	...
	return new(...) { ... };
}

private static IQueryable<Procurement> Search(string searchString, ...);
```




```
public async Task<PaginatedResult<Result>> Handle(...)
{
	var filteredQuery = Search(request.SearchString, ...);
	...
	return new PaginatedResult<Result>(...) { ... };
}

private static IQueryable<Procurement> Search(string? searchString, ...);
```

Joonis 19. Keelemudeli tagastustüübi täpsustamise ja parameetrite muutmise näide.

Keelemudel tegi koodi tõhususe parandamiseks muudatusi ka tsüklis ja tingimuslauses. Joonisel 20 on näha fragmenti, kus tingimuslause muutub pikemaks ning *foreach* tsükkel tehti ümber: välimine tsükkel itereerib arendajate Jira kasutajavõtmeid ning sisemine arendajaga seotud *Worklog* objekte, mis on omakorda *TaskKey* järgi grupeeritud. Tingimuslause puhul on lisandunud *null* kontroll asjatu, kuna meetodi liides, kust *worklogDTOs* tulevad, ei võimalda tühja tagastatavat väärtust. Tsükli muudatus on seevastu asjakohane. Joonisel 6, lk 25, on näha kuidas *Worklog* ehk ajalogi objektidest peavad tekkima *Report* objektid. Objekti loomiseks on vaja kõikide samade arendajate ja *Issue*'de (vt 4.2.1) ajalogide tunnid kokku liita. Joonisel 20 varasemal lahendusel on näha, et sisemise tsükli sees on vaja hankida kõik itereeritava *Worklog* objektiga seotud ajalogid, mille tunnid liidetakse kokku. Uuel lahendusel ei ole vaja seda enam teha, kuna selle tsükli itereeritav *taskGroup* ongi juba hulk logisid, mille tunnid peab kokku liitma. Nagu peatükis 6.2, lk 37, oli kirjas, siis töö autor kasutas teist lähenemist (vt 5.1.2).

```
if (!worklogDTOs.Any())
...
var worklogDeveloperGroups = worklogDTOs.GroupBy(worklog =>
      worklog.DevelopersUserKey);
...
foreach (var worklogDeveloperGroup in worklogDeveloperGroups)
{
      ...
      var distinctWorklogDTOs = worklogDeveloperGroup
            .DistinctBy(worklog => worklog.TaskKey);
      foreach (var distinctWorklog in distinctWorklogDTOs)
      {
            var relatedWorkLogDTOs = worklogDeveloperGroup.Where(...);
            var hours = relatedWorkLogDTOs.Select(...).Sum();
            ...
      }
}
```

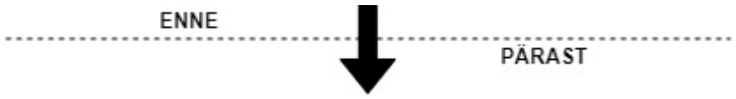

```
if (worklogDTOs == null || !worklogDTOs.Any())
...
var distinctDeveloperKeys = worklogDTOs
      .Select(worklog => worklog.DevelopersUserKey)
      .Distinct();
foreach (var developerKey in distinctDeveloperKeys)
{
      ...
      var developerWorklogs = worklogDTOs.Where(w =>
            w.DevelopersUserKey == developerKey);
      foreach (var taskGroup in developerWorklogs.GroupBy(w => w.TaskKey))
      {
            var hours = taskGroup.Sum(w => w.LoggedHours);
            ...
      }
}
```

Joonis 20. Keelemudeli tsükli ja tingimuslause muutmise näide.

6.2.3 Loetavuse parandamine

Loetavuse parandamiseks muutis keelemudel mõningate elementide nimesid ning tõi juurde muutujaid. Joonisel 21 on näha kahte *Handler* meetodit (vt 4.1). Esimesel eemaldas ChatGPT *lamda expression*'i „=>“ ja lisas muutuja, mis kirjeldab funktsioonide tulemust. Teisel muutis see muutujate *reports* ja *filteredQuery* nimesid hoopis *paginatedReports* ja *filteredReports* nimedeks, mis kirjeldavad rohkem nende sisu.

```
public async Task<IReadOnlyCollection<Result>> Handle(...) =>
	await _context.Partners.Select(p =>
			new Result { ... })
		.ToListAsync(cancellationToken);

public async Task<PaginatedResult<Result>> Handle(...)
{
	var reports = await filteredQuery
		.GetPaginated(...)
		.ToListAsync(...);
}
```



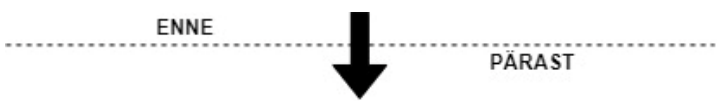

```
public async Task<IReadOnlyCollection<Result>> Handle(...)
{
	var partners = await _context.Partners
		.Select(p => new Result { ... })
		.ToListAsync(cancellationToken);
	return partners;
}

public async Task<PaginatedResult<Result>> Handle(...)
{
	var paginatedReports = await filteredReports
		.GetPaginated(...)
		.ToListAsync(...);
}
```

Joonis 21. Keelemudeli muutujate lisamise ja elementide nime muutmise näide.

6.2.4 Programmeerimiskeele spetsiifilised muudatused

Keelemudel leidis Joonisel 22 näidatud tehtele lühema viisi, kuidas vormistada arvu C# programmeerimiskeeles. Kuna *BillableSeconds* on *int* tüüpi, siis peab selle ümber tegema komakohaga tüübiks, et saada õige tulemus. Selle jaoks oli algses lahenduses „(*double*)“, aga ChatGPT vahetas selle *floating-point* täpsustaja „f“-iga.

```
var loggedHours = (float) Math.Round(
	(double) response.BillableSeconds / 3600, 2);
```

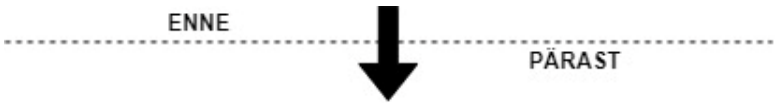



```
var loggedHours = (float) Math.Round(response.BillableSeconds / 3600f, 2);
```

Joonis 22. Keelemudeli tüübimääraja muutmise näide.

Keelemudel kaotas kõigi objektide omadustelt ära *required* märked. Joonisel 23 on näha *Command* objekti, kust need kadusid. Sellel juhul ei olegi need olulised, kuna validatsiooni atribuut *Required* juba määrab need omadused kohustuslikuks. Osade tagastatavate objektide puhul olid *required* märked siiski olulised, kuna need ei tohtinudki ilma nende omadusteta eksisteerida, aga keelemudel eemaldas need sellegipoolest.

```
public record Command : IRequest<Unit>
{
	[Required] public required string ProcurementName { get; init; }
	[Required] public required string ReferenceNumber { get; init; }
	...
}
public async Task<Unit> Handle(...)
{
	...
	await _context.Procurements.AddAsync(...);
	await _context.SaveChangesAsync(...);
	...
}
```




```
public record Command : IRequest<Unit>
{
	[Required] public string ProcurementName { get; init; }
	[Required] public string ReferenceNumber { get; init; }
	...
}
public async Task<Unit> Handle(...)
{
	...
	_context.Procurements.Add(...);
	await _context.SaveChangesAsync(...);
	...
}
```

Joonis 23. Keelemudeli *required* eemaldamise ja *async* meetodi vahetamise näide.

Lisaks oli Joonisel 23 näha ka *Handler* meetodit, kus asendati *AddAsync* hoopis tavalise *Add*'iga. Selle asünkroonse meetodi puhul on muudatus turvaline, kuna andmebaasist ei pärita pärast *Add*'i enam andmeid. Seega on muudatus põhjendatud ja seda soovitab ka dokumentatsioon [20].

7 Tulemused ja järeldused

Selles peatükis võrreldakse manuaalse ja ChatGPT keelemudeli refaktoreerimise tulemusi. Tulemuste põhjal valideeritakse tööd, tehakse järeldusi ning esitatakse võimalusi töö edasiarendusteks.

7.1 Mõõdikute tulemused

Tabelis 7 on näha SonarQube rakendusest võetud mõõdikute tulemusi. Suuruste ja keerukuste mõõdikute kohta on lähemalt kirjas peatükis 2.4, lk 17. SonarQube pakub automaatset disainivigade tuvastamist, kus järgitakse C# jaoks 478 reeglit. Võlgnevus näitab, kui palju aega võib hinnanguliselt minna nende vigade parandamiseks [11].

Tabel 7. Projekti mõõdikute tulemused SonarQube rakendusest enne ja pärast refaktoreerimisi.

	Enne muudatusi	**Pärast manuaalset refaktoreerimist**	**Pärast keelemudeli refaktoreerimist**
Suurus			
Ridade arv	1039	1241	1047
Klasside arv	67	81	67
Funktsioonide arv	309	352	316
Failide arv	48	55	48
Duplikatsioon			
Dubleeritud ridu	0	0	0
Hooldatavus			
Probleemid	4	1	35
Võlgnevus	15 min	5 min	45 min
Keerukus			
Cyclomatic complexity	372	414	379
Cognitive complexity	43	26	34

On näha, et manuaalne refaktoreerimine suurendas lähtekoodi palju rohkem kui keelemudel: ridade arv suurenes vastavalt 19,44% ja 0,77%. Numbritest tuleb välja ka ChatGPT sisendi mõjusus, millest oli ka kirjas peatükis 6.2, lk 37: võis puududa piisav kontekst uute klasside või failide loomiseks, mistõttu neid ei tekkinud juurde.

Manuaalne refaktoreerimine vähendas ja keelemudel suurendas disainivigade kogust. Versioon enne muutusi väidab, et seal oli vaid 4 viga, kuigi manuaalse analüüsi käigus leiti koguni 19 (vt 4.2). Need neli viga on järgmised: kahes kohas peaks olema *FirstOrDefault* meetodi asemel *First*, tühja liidese peaks eemaldama ning ühe TODO kommentaari peab lõpuni viima. Nagu peatükis 4, lk 20, oli näha, siis autori poolt leitud vead vastasid Martin Fowleri raamatus kirjeldatud disainivigadele, kuid SonarQube ei suutnud neid või nende laadseid vigu tuvastada [2]. Sellegipoolest leidis tarkvara probleeme ja keelemudel tegi neid suures koguses juurde.

Sarnaselt disainivigadele ei suutnud SonarQube leida duplikatsiooni, mida autor peatükis 4.2.2, lk 24, leidis. Fowleri raamatu järgi vastab Joonisel 5, lk 24, olev koodifragment *duplication* koodi haisule (vt 2.2) [2]. Kokku leidis autor seda disainiviga 5 erinevas kohas, kuid SonarQube seda ei tuvastanud. Edaspidi tuleks leida juurde tarkvara või laiendus, mis tuvastab disainivigu paremini.

Cyclomatic complexity ehk tsüklomaatiline keerukus näitas, kui raske on koodi testida, ning *cognitive complexity* ehk kognitiivne keerukus, kui raske on inimesel mõista koodivoogu (vt 2.4). Tulemuste järgi on näha, et manuaalne refaktoreerimine küll suurendas testimise keerukust 11,29%, kuid vähendas mõistmise keerukust koguni 39,5%. Keelemudel suurendas samuti testimise keerukust ning vähendas kognitiivset keerukust, kuid mõlemat manuaalsest lähenemisest palju vähem: vastavalt 1,88% ja 20,93%.

7.2 Refaktoreerimise valideerimine

Nagu peatükis 2.1, lk 12, oli kirjas, koosneb refaktoreerimine visuaalset käitumist säilitavatest võtetest. See tähendab, et lõppkasutaja ei peaks aru saama, et midagi muutus. Nii pärast manuaalset kui ka keelemudeliga teostatud refaktoreerimisi ei muutunud kasutaja jaoks midagi. Nagu peatükis 4.1, lk 20, oli kirjas, siis esitluskihti ei muudetud. Kõik varasemad funktsionaalsused säilisid ja töötavad lõppkasutaja jaoks

samamoodi. Selle õigsust kontrolliti olemasoleva 6 integratsiooni ja 56 ühiktestiga, mis katsid 95,9% koodibaasist. Kõik testid õnnestusid pärast igat manuaalset võtet ning pärast igat tehisintellekti poolt pakutud koodifragmendi lisamist.

Tulemuste põhjal (vt 7.1) on võimalik öelda, et muudatused parandasid mõlemal juhul koodi hooldatavust, kuna *cognitive complexity* vähenes. See tähendab, et nii manuaalselt kui ka tehisintellektiga parendati mingil määral koodi.

7.3 Järeldused

Peatükis 7.1, lk 45, võrreldi SonarQube tulemuste põhjal refaktoreerimiste tulemusi. Kuna tarkvara poolt pakutud disainivead ei vastand manuaalse analüüsi käigus leitud vigade kogusele (vt 4), siis neid ei võetud järeldustes arvesse. Töö käigu ning võrdluste põhjal saab teha järgnevaid järeldusi:

- Manuaalne refaktoreerimine on võrreldes keelemudeliga efektiivsem kognitiivse keerukuse vähendamisel, kuid see eeldab suuremat ajakulu.
- Manuaalselt on võimalik viia sisse suuremaid ja keerulisemaid muudatusi kui keelemudeliga, kuid see toob kaasa suurema koodibaasi. Tuleb olla ettevaatlik, et mitte tekitada koodi asjatult keerukust.
- ChatGPT-ga on võimalik saada kiiremini märgatavaid tulemusi kui manuaalse refaktoreerimisega ning lahenduse variante, mille peale ei pruugi arendaja tulla. Pakutud lahendusi ei saa iseseisvalt potentsiaalsete vigade tõttu usaldada ja parima tulemuse võib saada mõlemat meetodit kombineerides, mis kiirendaks manuaalset protsessi, aga säilitaks töökindlust ja põhjalikkust.
- ChatGPT keelemudelile ei piisa klassi ja selle sõltuvate objektide liideste koodifragmendi sisendist, et pakkuda muudatusi, mis ulatuvad kaugemale kui etteantud klass. Tuleb leida sisukam ja laiema skoobiga sisendivalimise viis.
- SonarQube tarkvara ei tuvasta tõhusalt Martin Fowleri poolt pakutud või selle laadseid disainivigu [2]. Võimalusel tuleks leida mõni alternatiivne tarkvara või laiendus vigade otsimiseks. Sellegipoolest suudab see leida mõõdikud, mille põhjal saab järeldusi teha.

7.4 Alternatiivid

Manuaalse ja keelemudeli refaktoreerimise võrdlust oleks saanud teha ka teise tehisintellektiga. Mõned võimsad alternatiivid ChatGPT-le on näiteks GitHub Copilot ja Gemini [10]. Keelemudelite puhul on võimalik kasutada tulemuste saamiseks ka mõnda muud sisendimalli, mis võib anda laiema koodibaasi ülevaate või parema arusaama soovitud tulemustest.

Võrdluseks oleks saanud kasutada ka teise programmeerimiskeelega projekti. SonarQube võimaldab lisada laiendusi ning töö autor suutis leida Java programmeerimiskeele jaoks laienduse, mis pakub paremat Martin Fowleri raamatus loetletud koodi haisude leidmist [2]. Java programmeerimiskeelele on ka süsteemis rohkem kontrollreegleid, millega disainivead välja tuleksid: C# keelele on 478 ja Javale 698 [21].

Koodi analüüsimiseks on ka teisi tarkvarasid. Java projekti puhul saab kasutada PMD tarkvara, mis leiab koodis levinud programmeerimisvigu, kus on üle 400 sisseehitatud reegli. C# jaoks on olemas CPD, mis aitab leida koodist duplikatsiooni [22].

7.5 Edasiarendamise võimalused

Tulemustest oli näha, et ChatGPT ei saanud olemasoleva sisendiga teha muudatusi, mis ulatuvad kaugemale kui etteantud klass. Vertikaalse arhitektuuri puhul võiks proovida anda ChatGPT keelemudelile sisendiks terve üks funktsionaalsuse lõik (vt 4.1). Nii on suurem tõenäosus saada piiridesse ulatuva suurema kontekstiga sisendi ja on parem tõenäosus saada sisukamaid muudatusi.

Analüüsi käigus tuli välja, et ChatGPT keelemudeliga üksi ei suudetud tuua sisse nii suuri muudatusi kui manuaalselt. Tulevikus võiks proovida kombineerida manuaalset ning keelemudeliga refaktoreerimist ja vaadata, kas see annab kiirema ja sisukama lahenduse kui ainult manuaalse refaktoreerimisega.

Lisaks peaks tulevikus kasutama rohkem kui lihtsalt SonarQube rakendust koodi analüüsimiseks, et saaks teha laiema võrdluse. Näiteks on võimalik võtta lisaks teisi tarkvarasid, nagu PMD ja CPD või mõni tehisintellekt.

8 Kokkuvõte

Refaktoreerimine on koodipuhastusviis, mis koosneb mitmetest käitumist säilitavatest võtetest. Ilma koodi hooldamiseta muutub uute funktsioonide lisamine keerulisemaks ja aeglasemaks, ent ka hooldamisega aeglustub arendus, kuna arendusaega kulutatakse hoopis varasemate lahenduste parandamisele. Selle puhul on abiks refaktoreerimist automatiseerivad tööriistad, kuid nendelgi on omad nõrkused.

Lõputöö eesmärk oli anda objektiivne võrdlus manuaalse ja tehisintellekti refaktoreerimise tulemuste vahel. Selleks viidi allikate põhjal läbi manuaalne ja ChatGPT keelemudeliga automaatne refaktoreerimine. Tulemusi võrreldi tarkvara SonarQube poolt pakutud mõõdikute abil, et saada objektiivseid järeldusi.

Tulemustest tuli välja, et manuaalse refaktoreerimisega on võimalik saada sisukamaid lahendusi ja parandada koodi kognitiivset keerukust paremini kui ChatGPT-ga. Seevastu on manuaalne töö palju aeglasem ja toob juurde rohkem koodi kui keelemudel. Samas ei saa tehisintellekti üksi usaldada, kuna see tekitab suure tõenäosusega vigu.

Tööst saab järeldada, et mõlemad meetodid parendavad mingil määral koodi. Parima tulemuse võib saada mõlemat varianti kombineerides, kus manuaalne refaktoreerimine tagab põhjalikkuse ja töökindluse ning keelemudel kiiremad tulemused. Tulevikus võiks kasutada SonarQube rakendust koos laienduste või muu tarkvaraga, et saada parem ülevaade tulemuste efektiivsusest ja erinevustest.

Kasutatud kirjandus

[1] A. Almogahed and M. Omar, "Refactoring Techniques for Improving Software Quality: Practitioners' Perspectives," in *Journal of Information and Communication*, vol. 20, no. 4, pp. 511-539, 2021. [Online]. Loetud aadressil: doi.org/10.32890/jict2021.20.4.3.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Boston, MA, USA: Addison-Wesley, 2018.

[3] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, London, United Kingdom: Pearson, 2008.

[4] E. A. AlOmar, A. Venkatakrishnan, M. W. Mkaouer, C. D. Newman and A. Ouni, "How to Refactor this Code? An Exploratory Study on Developer-ChatGPT Refactoring Conversations," in *In Proceedings of MSR '24: Proceedings of the 21st International Conference on Mining Software Repositories (MSR 2024),* 2024. [Online]. Loetud aadressil: doi.org/10.48550/arXiv.2402.06013.

[5] Microsoft, "Overview of object oriented techniques in C#," 2024. [Online]. Loetud aadressil: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/. Kasutatud: 25.04.2024.

[6] J. Kerievsky, *Refactoring to Patterns*, Boston, MA, USA: Addison-Wesley, 2004.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Boston, MA, USA: Addison-Wesley, 1999.

[8] M. Jerzy and M. Lech, "Code Smells: A Comprehensive Online Catalog and Taxonomy," in *Developments in Information and Knowledge Management Systems for Business Applications*, vol. 462, pp. 543–576, 2023. [Online]. Loetud aadressil: doi.org/10.1007/978-3-031-25695-0_24.

[9] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss and E. A. AlOmar, "Exploring ChatGPT's code refactoring capabilities: An empirical study," in *Expert Systems with Applications*, vol. 249, 2024. [Online]. Loetud aadressil: doi.org/10.1016/j.eswa.2024.123602.

[10] A. Tornhill, M. Borg and E. Mones, "Refactoring vs Refuctoring: Advancing the state of AIautomated code improvements," CodeScene, Sweden, 9 January 2024. [Online]. Loetud aadressil: https://codescene.com/hubfs/whitepapers/Refactoring-vs-Refuctoring-Advancing-the-state-of-AI-automated-code-improvements.pdf. Kasutatud: 28.04.2024.

[11] SonarSource, "Metric definitions," 2024. [Online]. Loetud aadressil: https://docs.sonarsource.com/sonarqube/latest/user-guide/metric-definitions/. Kasutatud: 12.05.2024.

[12] S. D. Palma, D. D. Nucci, F. Palomba and D. A. Tamburri, "Toward a catalog of software quality metrics for infrastructure code," in *Journal of Systems and Software*, vol. 170, p. 110726, 2020. [Online]. Loetud aadressil: doi.org/10.1016/j.jss.2020.110726.

[13] G. A. Campbell, "Cognitive Complexity - a new way of measuring understandability," Switzerland, SonarSource, White Paper, 2023. [Online]. Loetud aadressil: https://www.sonarsource.com/docs/CognitiveComplexity.pdf. Kasutatud: 13.05.2024.

[14] E. A. AlOmar, A. Peruma, M. W. Mkaouer, C. Newman, A. Ouni and M. Kessentini, "How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation," in *Expert Systems with Applications*, vol. 167, 2021. [Online]. Loetud aadressil: doi.org/10.1016/j.eswa.2020.114176.

[15] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," in *Journal of Systems and Software*, vol. 198, p. 111575, 2023. [Online]. Loetud aadressil: doi.org/10.1016/j.jss.2022.111575.

[16] Microsoft, "CQRS pattern," 2024. [Online]. Loetud aadressil: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs. Kasutatud: 26.04.2024.

[17] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Boston, MA, USA: Addison-Wesley, 1994.

[18] Microsoft, "Anti-corruption Layer pattern," 2024. [Online]. Loetud aadressil: https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer. Kasutatud: 01.05.2024.

[19] Microsoft, "Query keywords (C# Reference)," 5 August 2023. [Online]. Loetud aadressil: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/query-keywords. Kasutatud: 20.04.2024.

[20] Microsoft, "Additional Change Tracking Features," 12 January 2023. [Online]. Loetud aadressil: https://learn.microsoft.com/en-us/ef/core/change-tracking/miscellaneous#add-versus-addasync. Kasutatud: 10.05.2024.

[21] SonarSource, "SonarSource static code analysis," 2024. [Online]. Loetud aadressil: https://rules.sonarsource.com/. Kasutatud: 13.05.2024.

[22] PMD Open Source Project, "Documentation Index," PMD Open Source Project, 12 5 2024. [Online]. Loetud aadressil: https://pmd.github.io/pmd/index.html. Kasutatud: 12.05.2024.

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

Mina, Holger Nõgu

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Manuaalne refaktoreerimine võrreldes ChatGPT keelemudeliga ettevõtte X koodibaasi näitel“, mille juhendaja on Tarvo 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.