TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

IAG40LT

Janar Salumaa 112649

PROGRAMMIKOODI REFAKTORISEERIMINE JA OPTIMEERIMINE C# NÄITEL

Bakalaureusetöö

Tarmo Robal (PhD)
Teadur

Tallinn 2015

Autorideklaratsioon

Olen koostanud antud töö iseseisvalt. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud. Käesolevat tööd ei ole varem esitatud kaitsmisele kusagil mujal.

Autor: Janar Salumaa

09.06.2014

Annotatsioon

Antud bakalaureusetöö on kirjutatud teemal „Programmikoodi refaktoriseerimine ja optimeerimine C# näitel“.

Lõputöö koosneb kolmest sisulisest peatükist. Töö peatükis 2 annab autor ülevaate terminist Refaktoriseerimine ja mida see täpselt tähendab. Kolmandas peatükis annab autor ülevaate terminist Optimeerimine ja seletab selle lahti. Neljandas peatükis uurib autor erinevaid programmikoode, millest osad on refaktoriseerimata, teised on refaktoriseeritud ja kolmandad on seotud programmikoodi algoritmidega. Uurimise eesmärgiks on leida, millised refaktoriseerimise või optimeerimise võtted on kõige kasulikumad.

Lõputöö koostamisel püstitas autor eesmärgi täitmiseks järgmised uurimisülesanded: selgitada välja refaktoriseerimise eelised ja puudused, millal oleks mõistlik olemasolevat programmikoodi refaktoriseerida, mida kujutab endast programmikoodi optimeerimine ja kuidas seda rakendada.

Töö käigus jõuab autor järeldusele, et programmikoodi loomisel tuleks lähtuda võimalikult palju refaktoriseerimise tavadest, mis on põhimõtteliselt ka programmikoodi loomise tavad, et vältida juba eos ebakvaliteetse programmikoodi loomist. Optimeerimise kohapealt leiab autor, et alati tasub ennem probleem täielikult läbi mõelda, et saaks kasutusele võtta võimalikult optimaalse algoritmi lahendatava probleemi jaoks, kuna sageli erinevad algoritmid ei oma olulisi eeliseid teiste ees, küll aga puudusi võrreldes üksteisega.

Lõputöö võtmesõnadeks on: refaktoriseerimine, C#, optimeerimine, programmikood, programmeerija.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 4 peatükki, 2 tabelit ja 11 joonist.

Abstract

Code refactoring and optimization on the Example of C#

The underlying aim of this thesis is to explore and find out which optimization and refactoring techniques are the most beneficial to use in programming with C# language. This thesis consists of three main chapters. In Chapter 2 the author gives an overview of the term “refactoring” and what it means. In Chapter 3 the author gives an overview and explains the term “optimization”. In Chapter 4 the author explores a variety of programming codes, some of which are not refactored, some are refactored and the others are related to the program code algorithms.

When drafting the thesis, the author set the following research tasks to meet the goal: to identify the advantages and disadvantages of refactoring, when it would be reasonable to refactor an existing program code, what is program code optimization, and how to implement it.

The author concluded from the thesis investigation that when creating a program code it would be wise to follow as much as possible refactoring practice. This is basically the practice of creating program code in order to avoid poor-quality code generation. When it comes to optimization, the author finds that it is necessary to always think through a problem beforehand in order to use the most optimal algorithm for this problem, because some algorithms do not have any advantage over others, but they may have a lot of drawbacks.

The main part consists of 40 pages and is written in Estonian.

Thesis keywords are: refactoring, C#, optimization, code, software programming.

Lühendite ja mõistete sõnastik

OOP Object-Oriented Programming, objekt-orienteeritud programmeeerimine

DLL Dynamic-Link-Library, DLL-teek (Käivitatav käsklus, mis on salvestatud eraldiseisvana ning mida erinevad programmid saavad kasutada)

SQL Structured Query Language, struktuurpäringukeel

Sisukord

Jooniste nimekiri

Tabelite nimekiri

1. Sissejuhatus

Bakalaureusetöö läbivateks teemadeks on programmikoodi refaktoriseerimine ja optimeerimine, millest esimene kujutab endast programmikoodi loetavuse ja süstemaatilise arhitektuuri parandamist ja teine programmi töötsükli võimekuse suurendamist. Selline teemade valik on tingitud selle valdkonnaga pideva kokkupuute tagajärjel. Töö eesmärgiks oli uurida võimalikult universaalseid ja ka teistele teemast huvitatud isikutele arusaadavaid programmikoodi esitamise viise. Uurimise põhieesmärk oli leida erinevaid programmikoodi refaktoriseerimise tehnikaid ehk teisisõnu programmi erinevaid struktuurilisi ülesehitusi. Programmikoodi refaktoriseerimisel peab jääma programmi enda taustaloogika muutumatuks ehk samade sisendite puhul on ka väljundid samad. Programmikoodi sisu muutmist, mille puhul samuti programmi väline eesmärk säilib, nimetatakse optimeerimiseks. Need kaks protseduuri paistavad peale vaadates samad, kuid rakendamise eesmärk on siiski erinev. Erinevus seisneb peamiselt selles, et refaktoriseerimist rakendatakse olemasoleva programmikoodi loetavuse parandamiseks, kuid optimeerimist kasutatakse programmi töö aspektide nagu näiteks jõudlus, ressursi nõudlikkus parandamiseks.

Bakalaurusetöös seatud eesmärkide saavutamine oli jaotatud 4 erinevasse etappi:

- Töötada läbi erialane kirjandus refaktoriseerimise ja optimeerimise kohta, et luua teoreetiline raamistik, mille põhjal praktilist osa rakendada.
- Koostada näitlikke erinevate algoritmide ja struktuuridega programmikoode, mille rakendamise eesmärgid jäävad samaks.
- Analüüsida neid erinevaid programmikoode erinevate monitooringu programmidega ehk profileerijatega.
- Töö tulemuste analüüs, järeldused ja kokkuvõtte koostamine.

Töö lõpus esitatakse erinevate refaktoriseerimise tehnikatega ülesandeid nagu duplikaatide eemaldamine, loetavuse parandamise ja teised. Seda tehakse selleks, et saaks ka visuaalse ülevaate töö eesmärkidest. Võrdlusmomentideks on erinevad programmikoodi osad, kus

muudetakse algselt esitatud koodi vastavalt uurimise käigus selguvatele tehnikatele. Uurimustöös analüüsib autor, kui palju võib kommenteerimata või puuduliku struktuuriga programmikood aeglustada selle programmikoodi hilisemat edasiarengut.

Programmikoodi näited on esitatud kõik C# keeles. C# on üldotstarbeline porgrammeerimiskeel, mille arendajaks ja loojaks on Microsoft Corporation. C# on C-keelest tulenev objekt-orienteeritud programmeerimiskeel, mille süntaks on samalaadne C-keelega.

Programmikoodi optimeerimise näidetes tuuakse võrdlusena välja erinevad algoritmid, mille töö tulemusena lõppvastused jäävad samaks, kuid vastuseni jõudmise protsess erineb. Eksperimentides kasutatakse suuri andmehulkasid ja korduvaid katseid, et tulemuses paistaksid välja erinevused.

Optimeerimise analüüsi tegemiseks kasutas autor erinevaid monitooringuprogramme, milleks olid Visual Studio lisad ja Scitech Net Memory Profiler, millega saab analüüsida programmikoodi mälukasutust. Neid programme nimetatakse profileerijateks (programmikoodi analüsaatorid) ja nende kasutamise eesmärk on leida erinevusi programmide töötsüklis.

Käesolev lõputöö koosneb kolmest sisulisest peatükist. Töö esimeses peatükis annab autor teoreetilise ülevaate terminist Refaktoriseerimine toetudes teaduslikele lähtekohtadele ning seletab lahti selle tööpõhimõtte. Teises peatükis teeb autor sedasama termini Optimiseermine kohta. Kolmandas ehk viimases peatükis uurib ja võrdleb autor erinevaid programmikoode, millest osad on refaktoriseerimata, teised refaktoriseeritud ja kolmandad on seotud programmikoodi optimeerimisega kasutades sama ülesande lahendamiseks erinevaid algoritme.

2. Refaktoriseerimine

2.1. Mis on refaktoriseerimine?

Programmikoodi refaktoriseerimine on protsess, kus olemasoleva programmikoodi struktuuri ja sisu muudetakse sellisel viisil, et säilib rakenduse algoritm, kuid loetavus ja üleüldine arusaadavus paraneb. Refaktoriseerimine parandab põhiliselt tarkvara mittefunktsionaalseid omadusi. Põhilisteks refaktoriseerimise eelisteks on programmikoodi parem loetavus ja kergem struktuur, mis toob endaga kaasa tulevikus lihtsama hoolduse ja edasiarenduse. Refaktoriseerimise puuduseks on see, et temaga ei kaasne uut funktsionaalsust, kuid selle läbiviimine võtab aega. Korraliku refaktoriseerimise tagajärjel kaasneb selle protseduuriga väljendusrikkam sisemine arhitektuur, mis omakorda võimaldab programmikoodi paindlikkust [1]. Refaktoriseerimine on olemasoleva programmikoodi disaini kvaliteedi tõstmine [2].

„*Järjepidevalt arendades programmikoodi disaini, me teeme selle järjest kergemaks ja kergemaks, millega tööd teha. Kui sa saavutad järjepideva programmikoodi refaktoriseerimise harjumuse, siis sa avastad, et on palju kergem laiendada ja hooldada olemasolevat programmikoodi*“ – Joshua Kerievsky [3].

2.2. Refaktoriseerimise ajalugu

Kuigi programmikoodi refaktoriseerimist on tehtud mitteametlikult juba ajast, kui selle kohta veel viited puudusid, siis William Grisworld-i 1991 PhD väitekiri on üks esimesi suuremaid akadeemilisi uuringuid programmikoodi refaktoriseerimisest funktsionaalsel ja protseduurilisel tasemel. Sellele järgnes William Opdyke-i 1992 väitekiri refaktoriseerimisest objekt-orienteeritud (OOP) programmikoodil. Eelnevalt on refaktoriseerimise teooria ja mehhanism olnud käsitletud kui programmi transformeerumise süsteemid [1].

Esimene teadaolev refaktoriseerimise sõna kasutus avalikus kirjanduses oli William F. Opdyke-i ja Ralph E. Johnson-i artiklis 1990 a. septembris [4]. Grisworld-i Ph.D väitekiri ja William F. Opdyke väitekiri avaldatud 1992 a. kasutasid samuti seda terminit [5].

Terminit *faktoriseerimine* on kasutatud Forth-i kogukonnas (Forth on programmeerimiskeel) vähemalt 1980-ndate aastate algusest. Kuues peatükk Leo Brodie raamatust „Thinking Forth“ (1984) on samuti pühendatud just sellele teemale [5].

Ekstreemprogrammeerimise (agiilne programmeerimise stiil) funktsiooni tükeldamise meetod omab sama tähendust nagu faktoriseerimine Forth-is. Funktsiooni tükeldamise meetod on see, kus olemasolev funktsioon või meetod tükeldatakse väiksemateks meetoditeks, kus neid oleks võimalik kergem hallata [6].

Refaktoriseerimine on kaudselt olemasoleva programmikoodi ilustamine, siis kindlasti selle mitteametlik ajalugu jääb kaugemasse minevikku, võrreldes selle termini esmakordse mainimisega. Lõppkokkuvõttes tahab iga programmeerija kirjutada võimalikult loetavat ja arusaadavat programmikoodi.

2.3. Põhilised vajadused refaktoriseerimiseks

Refaktoriseerimine on protsess, mille käigus programmikoodile uut funktsionaalsust ei lisandu, vaid parandatakse olemasoleva koodi mittefunktsionaalseid omadusi. Refaktoriseerimine, kui arendustsükkel sellisel juhul väikeste projektide puhul finantsiliselt kasulik ei ole. Enamasti tehaksegi enne refaktoriseerimist analüüs, kas seda protsessi on üldse mõttekas läbi viia.

Sellegipoolest paljud projektid arenevad ajaliselt edasi olenevalt vajadusele. Näiteks on vaja parandada turvaauk, mis ajapikku on ilmnenud või siis lisada uusi elemente või täiendata programmikoodi olemasolevat funktsionaalsust. Selles etapis tavaliselt ilmnebki see, et programmikood oleks vajanud eelnevat refaktoriseerimist, kui see juhuslikult puudub.

Programmikoodi edasi arendaja ei pruugi olla selle autor, kuna ettevõtetes töötajad vahetuvad ja võõrast programmikoodi on raske edasi arendada, kui sellel puudub korralik

struktuur, kommentaarid või funktsioonide nimetused on ebaselged. Samuti võib samadel nüanssidel tekkida probleeme ka esialgsel autoril, sest ajapikku inimesed unustavad eelnevalt tehtu detaile. Sellisel juhul võib edasiarendus võtta palju aega ja kuna aeg on raha võib selline ettevõtmine lõppkokkuvõttes osutuda kallimaks, kui oleks olnud esialgne koodi refaktoriseerimine.

2.4. Ebaselgelt defineeritud ja koostatud kood

Põhilised näited ebaselgest programmikoodist on see, kui ta ei ole loetav. See tähendab, et programmikoodi meetodid ja parameetrid on defineeritud viisil, kus nad ei ole isekommenteerivad ja nende eesmärk on raskesti arusaadav. Teiseks tunnuseks on programmikoodi struktuuri puudus, ehk meetodid ei ole kindlalt ära jaotatud vastavalt oma funktsionaalsusele klassidesse või ei ole kindla süsteemi järgi järjestatud klassis.

Järgnevalt on vaadeldud erinevaid probleeme, mis võivad tekkida programmikoodi kirjutamisel või edasisel arendusel, kui ei ole lähtutud refaktoriseerimise põhimõtetest.

2.4.1 Surnud kood

Surnud koodiks programmeerimises kutsutakse koodi, mille poole kunagi ei pöörduta ja mida ei käivitata. Selle põhjuseks võib olla palju erinevaid asjaolusid. Enamasti on selleks see, et vastavat meetodit enam ei vajata, kuid see on jäetud programmikoodist mingil põhjusel eemaldamata. Teistel juhtudel lihtsalt meetodi loogika on ülesehitatud selliselt, et programmikoodi tsükkel ei jõuagi sinna faasi, kus ta peaks selle surnud koodi meetodi välja kutsuma. Samuti välja kommenteeritud kood võib arendajale tekitada palju peavalu ja küsimusi.

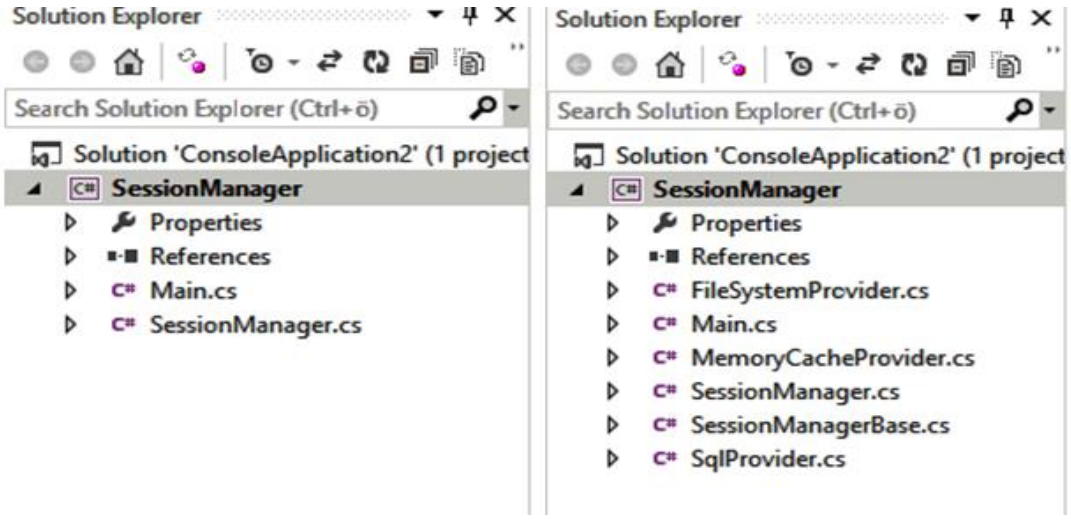
Surnud kood on väga suureks segajaks, kui tuleb hiljem otsida mingit probleemi või viga programmitöös, kus selle olemasolu võib seda protsessi pikendada märgatavalt ja suunata probleemi otsijat väga valedesse suundadesse.

2.4.2 Programmikoodi loogiline struktuuri ülesehitus

Kindla struktuuri olemasolul on programmikood hästi loetav võrreldes programmikoodiga, mille ülesehitusel ei ole struktuurile rõhku pandud. Kuna tal on nii-öelda loogiline juhend,

mis tähendab ka seda, et ilma olemasoleva dokumentatsioonita on kohe aru saada, kus võivad asuda teatud funktsioonid. Kindla struktuuri olemasolu vähendab võimalust, et tekib koodi duplikaate, mis on peamised vajadused refaktoriseerimiseks. Duplikaatide eemaldamine on seotud ka programmikoodi optimeerimisega. Selle protsessi käigus programmi enda virtuaalne suurus juba väheneb ja samuti väheneb ka mälukasutus. Programmikoodi kindel struktuur soosib ka tulevikus edasiarendamist, isegi kui arendaja ei ole esialgse koodi autor ja enamasti ühel programmil võib olla üle kümne või saja erineva autori.

Järgnevas näites (Joonis 1) on näha programmikoodi failisüsteemi struktuurilised erinevused. Näites on tegemist sessioonihalduriga, milles saab sessiooni salvestada vahemällu, faili või SQL-andmebaasi ja sealt lugeda.



(a) (b)

Joonis 1. Programmikoodi failisüsteemi struktuurilised erinevused. Ebaselge struktuur on esitatud vasakul (a) ja selge struktuur on paremal (b).

Joonisel 1a on esitatud näitena sessioonihalduri struktuur, kus puudub info või viited sessioonihalduri sisu kohta. Tegemist on ebaselge sessioonihalduri struktuuriga, mis tähendab, et struktuurist ei ole võimalik välja lugeda, mis funktsionaalsusi sessioonihaldur omada võib. Seevastu Joonisel 1b on tegemist selge struktuuriga ehk see struktuur annab ülevaate programmikoodist enne programmikoodi ennast nägemata. Joonis 1b toob märksõnadena välja *SQL, FileSystem, MemoryCache, SessionManager*, millest võib

järeldada, et antud sessioonihaldur salvestab sessiooni failisüsteemi, *SQL* andmebaasi või vahemällu, kuid see ei pruugi olla sedasi ja selles tasuks veenduda, kui tegemist ei ole programmeerija enda kirjutatud programmikoodiga.

2.5. Refaktoriseerimise tehnikad

Programmikoodi refaktoriseerimises kasutatakse palju erinevaid tehnikaid. Kõikide tehnikate üldine eesmärk on sama, ehk saavutada võimalikult loetav ja struktuurne programmikood. Järgnevalt on esitatud ülevaade peamistest kasutatavatest tehnikatest.

2.5.1 Koodi abstraheerimise tehnikad

Abstraktsioon on enamasti võõras mõiste, mis enamasti viitab millelegi, millel puudub füüsiline olemus. See ei eksisteeri ei ajas ega kohas, kuid sellest räägitakse kui millegi tüübist, nagu näiteks idee [7]. Ilma abstraktsioonita me programmeeriksime siiamaani masinakoodis ja võimalik, et ilma selleta arvutisüsteeme ei olekski, kuna kõrgtaseme keeled interpreteeritakse kompilaatori poolt masinkoodiks ja seda interpreteerimist võimaldabki abstraktsus [8].

Programmikoodi abstraktsus on see, kus kasutatakse üldiselt abstraktset baasklassi (OOP keeled), mida otseselt välja kutsuda ei saa, kuid mis määrab oma alamklasside peamised atribuudid ja meetodid. Sellisel juhul ei teki ühtede ja samade koodide korduvkasutust.

Erinevaid refaktoriseerimise tehnikaid on palju ja kindlasti on paljud selle valdkonnaga tegelevad isikud nendest tehnikatest erinevatel arvamustel. Levinumad refaktoriseerimise tehnikad on:

Kapseldamine – tekkis 1960. alguses kui ilmusid esimesed OOP keeled. Kuna tollal oli arvutimälu väga piiratud ja pidevalt samasuguse koodi kordamine ei saanud tulla kõne allagi. Ka inimese mälu on piiratud siis peagi avastati, et lihtsalt funktsioonide väljakutsumine, sealjuures funktsiooni enda tausta mitte nägemine, tegi programmeerimise programmeerijatele palju hõlpsamaks [9].

Seega kapseldamine on protsess, kus eraldatakse klassi funktsionaalne sisu ja liides. Liides on klass, kus ei ole funktsiooni realisatsiooni. Kapseldamise käigus antakse omadustele väärtused läbi *Set* funktsioonide ja omaduste väärtus küsitakse läbi *Get* funktsioonide. Programmeerija ei pea teadma, mis tegelikult toimub funktsiooni väljakutsumisel. Temale piisab vaid sellest, kui teatud tüüpi ülesande lahendamiseks saab funktsiooni esile kutsuda kindlatel andmetüüpidel ja parameetritel. Tänapäeva objekt-orienteeritud kõrgkeeled enamasti baseeruvadki olemasolevatel liidestel, kus põhifunktsionaalsus on juba olemas.

Üldiste- ja geneeriliste tüüpide kasutamine – Kuna C#-is nagu ka C++-is ja JAVA-s on kõik andmetüübid peale *string*-i objekt tüüpi (string on reference), siis saab kergelt teha tüübivahetusi. Seoses sellega on võimalik kirjutada universaalsemaid koode, mille käigus jääb vähem ruumi duplikaatidele. Duplikaadid on programmeerimises suur viga, millest tuleks hoiduda, kuna nad suurendavad programmi virtuaalset mahtu ja põhjustavad ka ebaselget koodi. Samuti võib duplikaatide esinemine olla programmis esinevate vigade peamiseks põhjustajaks [10].

```
public void Set<T>(T obj)
{
    // to code
}
0 references
public void Test()
{
    Set<string>("hello");
    Set<DateTime>(DateTime.Now);
}
```

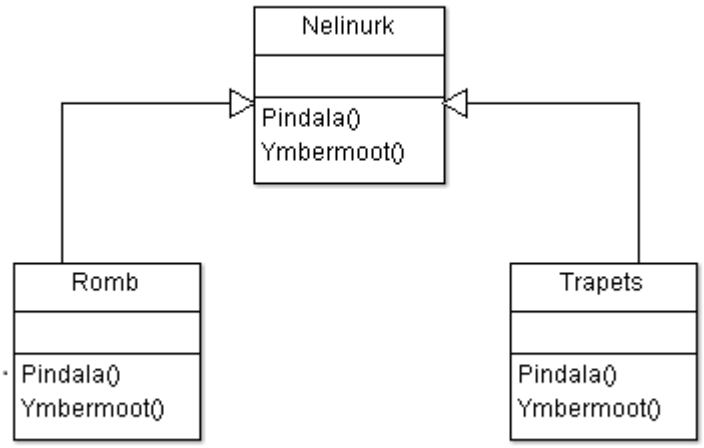
Joonis 2. Geneerilise andmetüübi kasutamine C#-is.

Joonisel 2 on näha geneerilise tüübi kasutamist, kus on funktsioon nimega *Set* ja peale funktsiooni on kohe nurksulud, milles saab ära märkida funktsiooni poolt kasutatava andmetüübi. Funktsioon *Set* on universaalne funktsioon kõigi andmetüüpide jaoks.

Polümorfism – See termin programmeerimises tähendab seda, et samasuguse nimega meetodil võib olla mitu erinevat käitumist. Polümorfism esineb enamasti baasklassi alamklassides. Kui alamklass omab sama nimega meetodit kui baasklass, siis alamklassi meetod kirjutab üle baasklassi funktsiooni. Näiteks, kui baasklass on nelinurk, mis omab

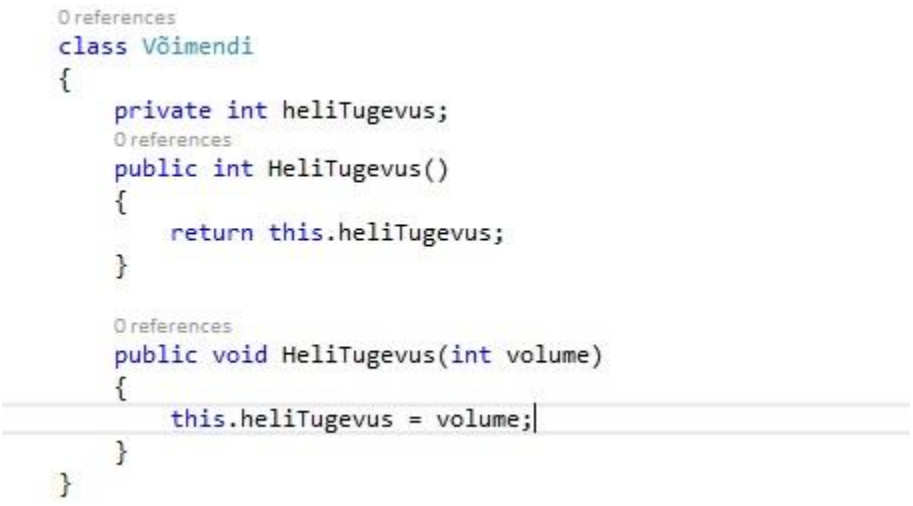
meetodeid pindala ja ümbermõõt, siis alamklass võib olla juba spetsiifilisem, ehk nelinurga erijuht (näiteks romb) ja kuna erinevad nelinurgad omavad erinevaid valemeid pindala ja ümbermõõdu arvutamiseks, siis oleks ebaloogiline kirjutada igale nelinurga erijuhule oma funktsioon erineva nimega (näiteks: rombi_ümbermõõt), kui saaks lihtsalt üle kirjutada baasklassi funktsiooni (ümbermõõt), mis väljastaks tulemusena vastava nelinurga ümbermõõdu.

Joonis 3 illustreerib eelnevat näidet nelinurgast ja nelinurga alamklassidest klassidiagrammina.



Joonis 3. Baasklass ja tema alamklassid nelinurga näitel.

Teine polümorfismi vorm on see, kus ühes klassis kasutatakse sama nimega funktsioone, kuid sisendparameetrid ja/või tagastatavad parameetrid on erinevad. Näiteks funktsioon *HeliTugevus()* tagastaks, kui kõrge parajasti helitugevuse nivoo on, kuid *Helitugevus (int volume)*, määraks uue helitugevuse. Joonisel 4 on illustreeriv pilt programmi ’Võimendi’ *HeliTugevus* funktsioonidest.



Joonis 4. Polümorfismi avaldumine samanimeliste meetodite korral.

2.5.2 Programmikoodi dekomponeerimise tehnikad

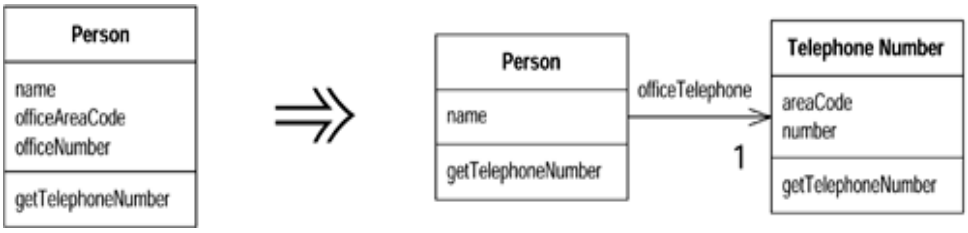
Dekomponeerimine tähendab olemasoleva programmikoodi tükeldamist läbi mille on võimalik programmikoodi lihtsustada. Programmikoodi dekomponeerimise tehnikad on programmeerimisel ja hilisemal refaktoriseerimisel väga vajalikud, kuna iga programm peaks olema ülesehituselt kindla ja igati arusaadava struktuuriga. Seda seetõttu, kuna kunagi ei või ette teada, millal võib tekkida vajadus seda programmi edasi arendada või teha vajalikke muudatusi. Seoses sellega, kui programm on oma ülesehitusel väga ebaselge, võib selle programmi refaktoriseerimine finantsiliselt väga kalliks minna.

Peamised tehnikad mida programmikoodi dekomponeerimisel kasutatakse on:

Komponentide loomine – tuleneb samuti mingil määral kapseldamisest, kuid on sügavama põhjaga. Eesmärk on luua väikesed, täiuslikud ja eraldi testitavad tarkvarakomponendid, mille eesmärgiks on võimalikult palju kaotada koodi kordamist. Samuti teeb selline lähenemine koodi kasutamise palju kergemaks ja arusaadavamaks. Põhilised komponendid on nn teenused *(service).* Näiteks kui loodav programm on seotud palju piltide haldamisega, siis oleks mõistlik luua teenus *ImageService*, mis tegeleb näiteks piltide üleslaadimisega, andmebaasist nime võtmisega, pildi kärpimisega või moonutamisega, jne. Seega täiuslik komponent igasuguste piltidega seotud toimingute tegemisteks, mis oleks korduvkasutatav nii antud programmi kui ka teiste programmide raames. Komponentidega suhtlemisel kasutatakse liideseid (*interface*), mis üldjuhul sisaldavad endas kõiki avalikke

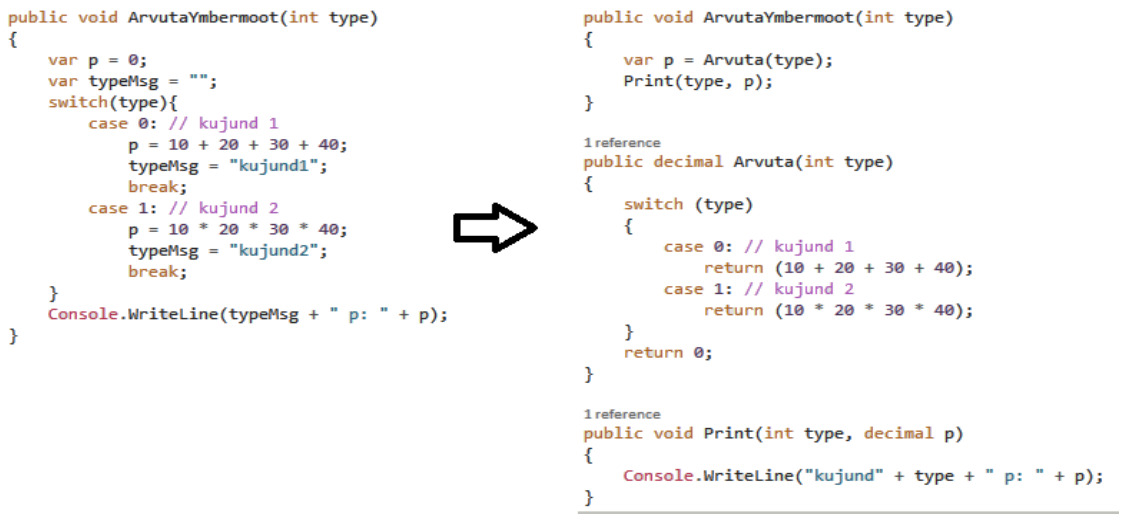
funktsioone selle komponendi kohta. Kui funktsioonide nimed on korralikult defineeritud ja kommenteeritud, siis lihtsam on uurida kas vastav komponent sisaldab endas vajaminevat funktsiooni võrreldes selle funktsionaalsuse otsimisega tervest programmikoodist. Komponentide olemasoluga on tunduvalt kergem ja kiirem ka olemasolevat programmikoodi edasi arendada. Komponendid asuvad põhiprogrammist eraldi. Windowsi platvormil arendatud *win32 või win64* rakenduste puhul asuvad programmi komponendi *DLL* laiendusega failides.

Klassist meetodite välja võtmine *(Extract class)* – Sellist refaktoriseerimist kasutatakse, kui olemasolev klass muutub liiga suureks ja selle üleüldine mõiste muutub ebaselgeks. Üldiselt võiks iga klass olla mingi kindla mõistega, et tekiks kindel ja tugev sisemine struktuur. Juhul kui klassi on tekkinud või hakkavad tekkima meetodid, mis enam otseselt ei kuulu sinna, siis tulekski need sealt nii-öelda välja kiskuda ja paigutada teise klassi, kuhu ta struktuurselt kuulub või vajadusel tekitada talle uus klass. Seda tasuks teha isegi siis, kui klassi võib jääda ainult üks meetod, kuna sellegipoolest aitab see kaasa kindla struktuuri loomisele. Joonisel 5 on näha skeem, kus ühest suurest klassist moodustatakse kaks väiksemat klassi, mis on refaktoriseerimise kohapealt palju loogilisem.



Joonis 5. Klassist meetodite väljavõtmine ja viimine teise klassi [11].

Meetodite lõhkumine *(Method extract)* – Sellist refaktoriseerimise tehnikat kasutatakse juhul kui meetod oma funktsionaalsuselt ja suuruselt läheb liiga suureks ja raskelt loetavaks. Suured meetodid on suurema tõenäosusega vigased. Et lihtsustada seda probleemi, siis olekski otstarbekam suured meetodid teha mitmeks väiksemaks meetodiks, kus üks peameetod, mis on vastavalt kasutaja poolt välja kutsutud, kutsub omakorda välja ühe või mitu teist meetodit, et sooritada ettenähtud funktsionaalsus. Joonisel 6 on esitatud meetod *ArvutaYmbermoot* (Joonis 6a) ja selle baasil loodud uus tükeldatud ümbermõõdu arvutamise meetod (Joonis 6b).



```
public void ArvutaYmbermoot(int type)
{
    var p = 0;
    var typeMsg = "";
    switch(type){
        case 0: // kujund 1
            p = 10 + 20 + 30 + 40;
            typeMsg = "kujund1";
            break;
        case 1: // kujund 2
            p = 10 * 20 * 30 * 40;
            typeMsg = "kujund2";
            break;
    }
    Console.WriteLine(typeMsg + " p: " + p);
}
```

```
public void ArvutaYmbermoot(int type)
{
    var p = Arvuta(type);
    Print(type, p);
}

1 reference
public decimal Arvuta(int type)
{
    switch (type)
    {
        case 0: // kujund 1
            return (10 + 20 + 30 + 40);
        case 1: // kujund 2
            return (10 * 20 * 30 * 40);
    }
    return 0;
}

1 reference
public void Print(int type, decimal p)
{
    Console.WriteLine("kujund" + type + " p: " + p);
}
```

(a) (b)

Joonis 6. Näide: suurema meetodi lõhkumine mitmeks väikeseks, (a) algne meetod, (b) tükeldamisel saadud meetod.

2.5.3 Tehnikad koodi loetavuse parandamiseks.

Koodi võimalikult hea loetavus ja tuleviku edasiarendus oleneb täielikult koodi struktuurist ja selle ülesehituses lähtutud põhimõtetest. Et programmikood oleks võimalikult loetav, peab see olema võimalikult informatiivne ehk meetodite nimed peaksid olema iseennast kommenteerivad ning programmikoodi struktuur oleks selge ja arusaadav. Koodi informatiivseks muutmiseks on mitmeid erinevaid tehnikaid ja nendeks on peamiselt:

Parameetri või meetodi asukoha muutmine – Üleüldine eesmärk on see, et kõik meetodid ja parameetrid asuksid võimalikult loogiliselt terves struktuuris. Loogika seisneb selles, et ühte kindlat suurt funktsionaalsust esitavad meetodid ja parameetrid asuksid kindlas klassis või selle baasklassis. Kui meetod või parameeter asub temale mitte sobilikus kohas, tuleks rakendada asukoha muutmist. Näiteks kui pindala arvutuse meetod asub ümbermõõdu arvutamise klassis, siis see meetod või parameeter tuleks nihutada temale loogilisse klassi, milleks on pindala arvutamise klass.

Nimede muutmine – Programmikood peaks olema võimalikult isekommenteeriv. Kui programmi meetodi nimi on arusaamatu või üldse ilma igasuguse tähenduseta, siis see tuleks nimetada ümber vastavalt, mida antud meetod teeb, või mis parameetri enda üleüldine mõte koodis on. Väga raske oleks aru saada meetodi olemusest, kui meetodi nimi

on näiteks *Meetod1()* aga tema funktsionaalsus selle eest on e-posti saatmine. Siis võikski olla meetodi nimi *SaadaEpost()*. Või siis parameetri nimi on lihtsalt *A*, kuid temas hoitakse näiteks kolmnurga pindala väärtust. Sellisel juhul oleks sobiv nimi talle *kolmnurgaPindala*, kuigi pindala tähistatakse ka tähega S, kuid selle mõiste pikas programmikoodis jääb kindlasti esialgu arusaamatuks.

Objekt-orienteeritud programmeerimises kasutatakse refaktoriseerimisena veel meetodite liigutamist alamklassist baasklassi ja vastupidi. Objekt-orienteeritud programmeerimises iga alamklass omandab ka baasklassi sisu, mis programmeerimise redaktoris välja ei paista. Seetõttu oleks mõistlik uurida, kui baasklassis esinevate funktsioonide seast kasutab mingit funktsiooni ainult üks tema alamklassidest, siis tuleks nihutada see funktsioon ka sinna alamklassi. Toimida tuleks ka vastupidiselt, et kui mitu alamklassi kasutab sarnast funktsiooni, mille saaks realiseerida ühe funktsioonina baasklassis.

3. Programmikoodi optimeerimine

Arvutiteadustes programmi optimeerimine või tarkvara optimeerimine on protsess, kus muudetakse programmi teatud aspekte või sisemist struktuuri, et programm töötaks efektiivsemalt või kasutaks vähem ressursse [12].

Kokkuvõtvalt võib programmikoodi optimeerida viisil, et ta käivituks veel kiiremini, oleks võimeline töötama väiksema mäluressursiga või teiste ressurssidega või tarbida vähem voolu (programmikoodi optimeerimine riistvara tasemel).

Sõna optimeerimine tuleneb enamasti sõnatüvest optimaalne, kuid programmeerimises on väga haruldane see, kui optimeerimine tagab täielikult optimaalse programmikoodi. Programmikood peale optimeerimist on enamasti optimaalsuse lähedal ainult teatud küljest, milleks optimeerimist üldse rakendati.

3.1. Optimeerimise vajadus

Programmikoodi optimeerimine on protsess, kus muudetakse programmi algoritme, mille töö tulemusel väljund ei muutu. Kuna algoritmid on erinevad, siis läbi algoritmide on võimalik parandada programmikoodi ressursside kasutamist.

Algoritm on kindla järjekorraga samm sammuhaaval käskude jada ettenähtud kalkulatsioonide tegemiseks. Algoritme kasutatakse kalkulatsioonideks, andmete töötluseks ja automatiseeritud analüüsiks. Algoritm on lõpliku suurusega loetelu hästi defineeritud tegevustest tegemaks mingit kalkulatsiooni [13]. Programmeerimises võib funktsioon nimetada ülekantud tähenduses samuti algoritmiks, kuna funktsiooni tulem on kindel tegevuste jada.

Kuna tulemuseni võib jõuda mitut erinevat moodi, siis sellest tulenevad ka erinevad algoritmid. Seoses sellega jagatakse algoritme veel eraldi klassidesse erinevate aspektide poolest. Iga samasuguse probleemi erinev algoritm toetub enamasti erinevatele

printsiipidele ja seoses sellega ka programmeerimises erinevad algoritmid kasutavad ressursse erinevalt.

Optimeerimiseks kasutatakse selleks vastavaid programme mida nimetatakse profileerijateks ehk programmid, mis kujutavad endast programmikoodi töötsükli dünaamilise analüüsi läbiviijaid. Neid programme on erinevaid ja esineb peaaegu igas valdkonnas. Nende programmidega saab ülevaate programmi mälukasutusest (näiteks: Memory Profiler) teatud programmi osadest või siis ülevaate andmebaasi päringutest (näiteks: Entity Profiler). Paljude programmide, mis kasutavad suuri andmebaase, probleemid hakkavadki andmete töötlusest või päringutest ja need vead enamasti tulevad läbimõtlemata algoritmist. Profileerijad on manuaalseks optimeerimiseks ja nende eesmärk ongi leida nii-öelda ’pudelikaelad’ programmikoodist.

’Pudelikaelad’ on koodiosad, mis on põhilised jõudluse tarbijad. Tihtipeale nimetatakse neid ka kuumadeks kohtadeks. Nad on peamised optimeerimise vajalikkuse põhilised tekitajad, tulenedes valesti valitud algoritmist või siis programmeerimiskeele enda piirangutest.

Nagu eelnevalt öeldud, siis programmi täielikult või universaalselt optimeerida ei ole võimalik. Sellepärast optimeeritaksegi programme vastavalt tarbija vajadusele. Programmi ühe aspekti optimeerimine tuleneb üldiselt teise aspekti arvelt ja siis tulebki kindlaks teha, milline neist kõige vajalikum vastavas projektis on. Kiiremini töötav kood tähendab, et see programmikood kasutab rohkem mälu, mis omakorda võib tekitada vajadust suurema mälukiibi kasutamiseks, kui olemasolev mälukiip sellist ressurssi ei oma. Kui mälu on väga limiteeritud, siis tuleb kasutada teistpidi optimeerimist, mille käigus programmi algoritm saab olema aeglasem tulemuseni jõudmisel, kuid mahub ettenähtud mälu võimaluste piiridesse [14].

Programmikoodi teiseks optimeerimise võimaluseks peale erinevate algoritmide kasutamist on vahemälu kasutamine. Vahemälu on suure juurdepääsukiirusega mälu, mille põhiliseks eesmärgiks on saavutada võimalikult kiire ligipääs andmetele, mis parajasti asuvad põhimälus. Vahemälu on nii protsessoril kui ka kõvakettal. Rakendused kasutavad siiski põhiliselt kõvakettal olevat vahemälu komponenti.

Vahemälu kasutamine on kasulik ja tihtipeale ka kohustuslik kõigi erinevate programmikoodide ja algoritmide puhul. Vahemälu kasutamine annab võimaluse korduvate andmepäringute korral need salvestada ja hiljem lugeda, ilma et peaks algoritmi nende andmete töötluseks kasutama. Vahemälude põhiline kasutamisvaldkond on reaalajasüsteemid, kuna need põhinevad enamasti suurte arvutuste tegemisel.

Kokkuvõtvalt puudub optimeerimise võimalus, mis muudab programmi kõik aspektid optimaalseks ja lähtudes sellest optimeeritakse programme vastavalt viisile, mis on tarbijale kõige kasulikum ja rahuldab projekti algfaasis püstitatud tingimused. Arvutitehnika kiire arenguga on võimalused laienenud ja riistvara osade maksumus on langenud, kui võrrelda seda kümnendite taguse ajaga. Kümnendite tagusel ajal keskenduti peamiselt programmikoodi suurusele ja mälu kasutusele, mitte operatsioonide kiirusele.

3.2. Optimeerimise faasid

Optimeerima ei pea alati programmikoodi lõppfaasis. Sageli teostatakse esimene optimeerimine programmikoodi loomise disaini faasis, kus tehakse paberil valmis struktuur, mida programmeerimisel jälgitakse ja milline ta lõpuks olla võiks. Üleüldiselt oleneb programmi jõudlus tema arhitektuurist.

Optimeerimine disainifaasis - Disainifaasis optimeerimine tähendab seda, et programmikoodi disaini loomisel valitakse algoritmid, mida oleks otstarbekas kasutada antud probleemi lahendamiseks. Algoritm valitakse üldiselt sellejärgi, mida süsteemi riistvaraline pool võimaldab ja millised on kliendi ootused programmile. Kui süsteemi riistvara seab piirangud mäluressursile, tuleb valida aeglasem algoritm. Kui aga on näiteks tegemist reaalajasüsteemiga, mis vajab kiiret keskkonna analüüsimise võimet, siis muidugi tuleb selleks kasutada võimsamat riistvaralist poolt, mis seda lubaks, aga ka kiiremat algoritmi, mis omakorda vajab võimsamat riistvaralist poolt.

Optimeerimine lähtekoodi loomisel - Teiseks optimeerimise faasiks saab nimetada olukorda, kus optimeerimine toimub lähtekoodi kirjutamise ajal. Alati ei saa igat probleemi näha ette disainifaasis ja siis tuleb teha optimeerimised lähtekoodi loomise ajal. Teisisõnu tuleb valida teine algoritm antud probleemi lahendamiseks. Selles faasis optimeerimist

esineb harva ja teatud süsteemi loomise praktikate juures nagu näiteks testidel põhinev arendus (*Test-driven developement*). Kõige rohkem esineb seda agiilsete süsteemide loomise protsessis, kuna agiilsete põhimõtetega arenduse juures testitakse igat programmikoodi lõiku kohe peale selle loomist.

Optimeerimine lõppfaasis - Peamiseks optimeerimise faasiks on lõppfaasis optimeerimine, kus toimub ka enamasti terve programmi testimine ja vead ilmnevad kõige tõenäolisemalt. Selles faasis võivad ka tihtipeale ilmneda vead, kus näiteks teatud osad programmist ei sobigi omavahel kokku. Selliseid vigu on disaini ja vahepealses faasis väga raske leida või tähele panna.

Samuti lõppfaasis näeb programmi üleüldist töötamist ja saab teha analüüsi, kas valitud algoritmid täidavad ettenähtud vajadusi.

Tihtipeale võib lõppfaasis selguda, et antud programmeerimiskeel üldse ei sobigi probleemi lahenduseks. Erinevad keeled on erinevate võimalustega ja vahest võib saada ainult ühe programmeerimiskeele kasutamine nii-öelda pudelikaelaks. Näiteks programmeerimiskeel Phyton kasutab väga suures osas C-keeles kirjutatud teeke ja C-keel omakorda kasutab jälle assembleris kirjutatud teeke [15].

3.3. Millal optimeerida?

Programmikoodi optimeerimine võib vähendada loetavust ja lisada juurde koodi ainult selleks, et parandada programmi jõudlust. See võib teha süsteeme keerulisemaks, mis omakorda toob kaasa süsteemi raskema halduse ja hilisema vigade otsimise. Selle tulemusena viiakse programmikoodi optimeerimist enamasti läbi programmikoodi loomise lõppfaasis.

„Ennatlik optimeerimine“ on fraas, mida kasutatakse kirjeldamaks olukorda kus programmeerija laseb ennast segada jõudluse mõtetest programmikoodi osa või osade loomise ajal. See võib tuua kaasa koodi, mis ei ole nii puhas, kui ta oleks võinud olla või kood on ebakorrektne. See tuleneb sellest, et programmikood on optimeerimisega tehtud

keeruliseks ja programmikoodi autorile võib tekitada see segadust edasisel programmeerimisel, kuna ta peab mõtlema pidevalt läbi, mida ta eelnevalt on loonud.

Parem võimalus seoses sellega on kõigepealt disainida programmile terviklik struktuur, mille alusel luua programmikood ja hakata alles siis uurima, millised programmi osad on aeglased või ebakorrektsed ja vajavad edasist optimeerimist. Kui optimeerimist teha viimases faasis, siis võivad kergemalt tulla esile ka jõudluse probleemid, mida ei oleks olnud võimalik ennatliku optimeerimisega leida või vältida [16].

3.4. Optimeerimise probleemid

Optimeerimisel võib tihtipeale tulla ette palju erinevaid probleeme. Alati ei saa olla kindel, et uus valitud algoritm viib meid vajadusi täitva tulemuseni, kuid selle algoritmi testimine võib võtta aega, mis omakorda toob kaasa suuremad kulud.

Kuna optimeerimine ei too endaga kaasa uut funktsionaalsust, siis selle protseduuri rakendamine võib endaga kaasa tuua uusi vigu süsteemis või olemasoleva töötava koodi muuta mitte töötavaks, kuna optimeerimisel muudetakse olemasolevat algoritmi. Samuti käsitsi programmeerija poolt optimeeritud kood võib muutuda loetamatumaks, kui mitte optimeeritud kood ja seoses sellega võib see kaasa tuua hilisema hoolduse või tööshoidmisega probleeme.

Sõltuvalt sellest on optimeerimisel alati mingi hind ja enne selle protseduuri tegemist tuleks teha kindlaks, kas süsteem üldse vajab optimeerimist ja kas see on seda investeeringut väärt.

3.5. Automaatne optimeerimine

Isegi kui programmeerijad on programmi täielikult enda nägemuse järgi ära optimeerinud, toimub hilisem optimeerimine kompileerimisel, kus lähtekood muudetakse assembler keeleks ja hiljem interpreteeritakse masinkoodiks. Põhiliseks automaatseks optimeerijaks ongi kompilaator. Kompilaatoreid uuendataksegi pidevalt sellepärast, et leitakse uusi ja tõhusamaid optimeerimise algoritme. On leitud, et globaalsel optimeerimisel, ehk siis

automaatsel optimeerimisel on võit optimeerimisest kõige suurem. Kuigi üldiselt on kõige parem optimeerimise viis leida võimalikult hea algoritm [17].

Automaatne optimeerimine on samuti äriliselt tunduvalt odavam, kuna selle protseduuri läbiviimiseks ei ole vaja kasutada inimtööjõudu. Sellegipoolest, kui programmeerija on esialgse koodi loonud ebakvaliteetsel tasemel, kus lihtsamadki protsessid võivad kasutada suurt hulka arvuti jõudlusest, siis ei ole märgatavat abi ka automaatsest optimeerimisest, kuna programmikoodi tuum ise on ebakvaliteetne ja automaatne optimeerimine ei muuda olemasoleva koodi algoritmi.

Automaatsel optimeerimisel ei teki ka loetamatut koodi. Programmi, mis optimeerimist teeb nimetatakse optimeerijaks. Kuna kompilaator ei ole üks terviklik programm vaid samamoodi teatud algoritmil töötav erinevate programmide jaga, mille kõik etapid üldiselt kompileerimisel lähtekood läbib. Muidugi on ka eraldiseisvaid optimeerimise tarkvarasid (näiteks: CPLEX, GUROBI). Optimeerijate eelis on see, et nad optimeerivad programmikoodi vastavalt süsteemis kasutatavale riistvaralisele arhitektuurile.

4. Programmikoodi optimeerimise ja refaktoriseerimise näited

Bakalaureusetöö praktilise osa eesmärgiks oli näidata erinevaid optimeerimise ja refaktoriseerimise võimalusi erinevate struktuuride programmikoodidega ning võrrelda neid omavahel erinevate aspektide põhjal. Põhilisteks võrdluse aspektideks olid programmikoodi pikkus, programmi füüsiline suurus, mälu kasutus ja töötsükli kestvus. Teiseks eesmärgis oli proovida optimeerida teatud probleeme ja leida neist nii-öelda ’pudelikaelad’.

4.1. Testimiseks kasutatud tarkvara ja selle kirjeldus

Kuna eesmärk on testida võimalikult palju erinevaid aspekte, siis on vajadus kasutada ka võimalikult erinevate eesmärkidega profileerijaid, mis on nagu eelnevat mainitud - programmikoodi töötsükli analüsaatoreid.

Kõik programmikoodi näited on kirjutatud kasutades arenduskeskkonda Visual Studio 2013. Visual Studio on Microsofti tarkvaraarendusplatvorm sisaldades ka Microsoft C# keelt ja vahendeid selle baasil arendamiseks. Microsoft Visual Studio pakub erinevaid võimalusi programmikoodi analüüsimiseks (erinevad teegid: STOPWATCH, profiler).

Kõikide analüüsitud programmikoodi lõikude juures mõõdeti töötsükli aega kasutades Visual Studio *STOPWATCH* lisapaketti. Sellega saab töötsükli ajalise kestvuse ülevaate programmikoodi osast vastavalt arendaja soovidele. Selleks tuleb soovitud kohas käivitada *STOPWATCH* ja peale soovitud kohta tuleb see seisata. See lisapakett väljastab tulemuse mitmel erineval viisil. Põhilisteks on *takt* ja millisekund. Üks takt on 100ns ehk 1/10 000 ms.

Mälukasutust uuritud näidetes on analüüsitud programmiga .NET Memory Profiler, mis sai valitud, kuna on spetsiaalselt C#-keele profileerimiseks. Tegemist on spetsiaalse tarkvaraga, millega saab analüüsida programmide mälukasutust ja mäluliikumist. Töötab ta

ainult programmidega, mis on tehtud *.net frameworkiga*, mis omakorda on samuti ainult operatsioonisüsteemile Windows ja toodetud on ta Microsofti poolt nagu Visual Studio ise. Net Memory Profiler on Scitech Software AB toodang [18].

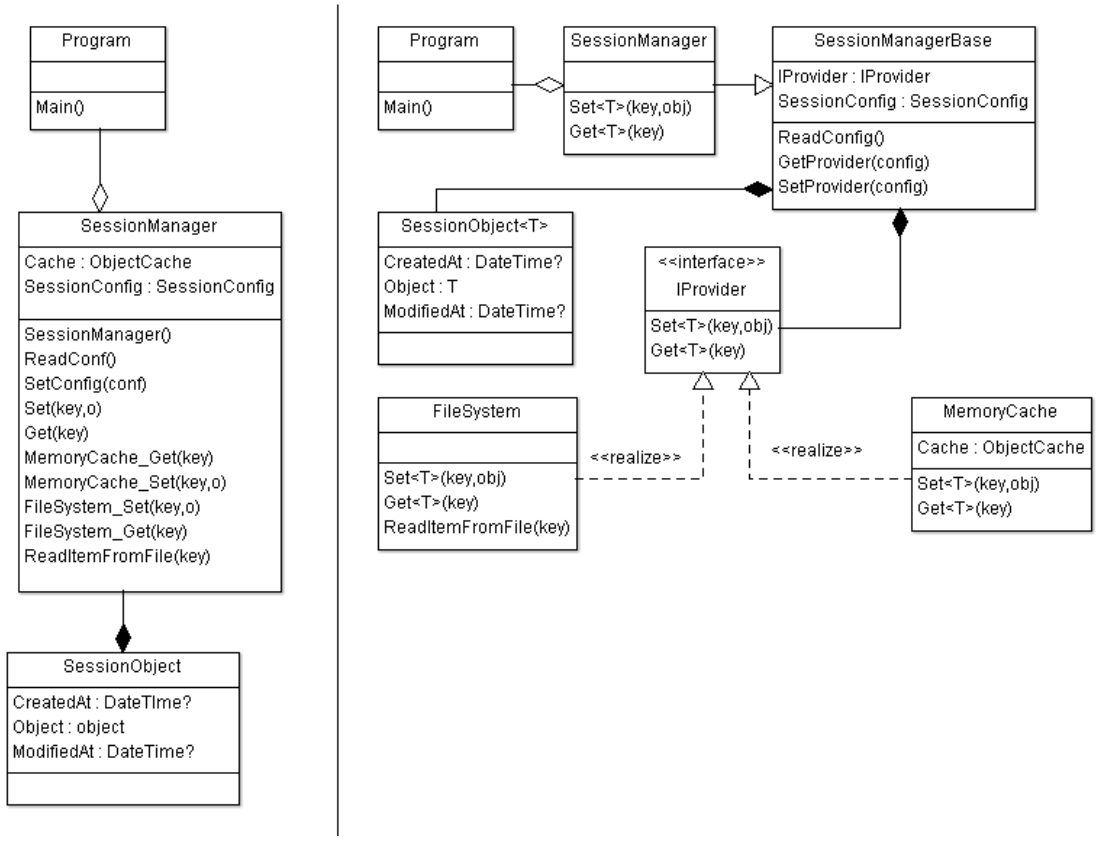
4.2. Analüüsitava programmikoodi refaktoriseerimine

Sessioonihalduri lähtekood esialgsel kujul on esitatud Lisas 1, kus teda ei ole veel refaktoriseeritud ega optimeeritud. Sessioonihalduri kood sai valitud sellepärast, et see on töö autori enda poolt kirjutatud programmikood enne lõputöö koostamist. Selle sessioonihalduri tööpõhimõte seisneb selles, et objekte või parameetreid saab lisada sessiooni, mis salvestatakse maha vastavalt kasutaja poolt valitud varustaja keskkonda. Keskkondadeks on vahemälu, failisüsteem ja *MSSQL* andmebaas. Programmil on kaks osa. Üks neist on kasutajaliides, mis suhtleb sessioonihalduriga ja teine on sessiooni haldur ise, mis omab kogu funktsionaalsust. Kasutaja pääseb ligi ainult kolmene funktsioonile, milleks on siis *Get* (võta), *Set* (pane), *setConfig* (varustaja). Funktsiooniga *Get* saab kasutaja küsida andmeid sessioonist ja funktsiooniga *Set* saab kasutaja andmeid sessiooni panna. Funktsioon *setConfig* annab kasutajale võimaluse valida, millisesse varustajasse sessioon maha salvestatakse.

Lisas 1 esitatud programmikoodis on näha palju kordusi. *Get* ja *Set* funktsiooni väljakutsumisel iga käivituse juures uurib sessioonihalduri antud funktsioon millise varustajaga tegemist on (Lisas 1 SessionManager.cs read 63, 66, 70, 78, 82, 86). Teiseks on *SessionObject* objekt-tüüpi, millel küll kindel suurus puudub, aga mida ei ole otstarbekas kasutada, kuna eraldab endale vajatust rohkem mälu. Nagu näha, siis *SQL* varustaja kood puudub, kuid ta töötab samal põhimõttel, mis Failisüsteem, et salvestab andmebaasi maha pika stringi. Failisüsteemi ja *SQL*-i puhul ongi miinuseks see, et andmete lugemisel tuleb neid alati õigesse tüüpi teisendada. Vahemälul seda probleemi aga ei ole. Eelisteks on see, et sessioon salvestatakse maha kõvakettale, mitte arvuti mällu, mis suurte objektide hulkade puhul vahemälu varustajale ei sobikski. Selle eest on vahemälu andmete töötlemisel kiirem.

Lisas 2 on esitatud sama sessioonihalduri kood, kuid refaktoriseeritud ja optimeeritud kujul. Refaktoriseerimise ja optimeerimise tulemusena on tekkinud ühe klassi asemel neli klassi ja koodiridu on samuti rohkem. Vähemalu ja failisüsteemi varustajad on viidud eraldi

klassidesse, millega on saavutatud sessiooni halduri klassi puhtus. Samuti on kogu varustajate *config* ja sellega tehtavad toimetused viidud uude klassi, mis on sessiooni halduri baasklassiks. Nüüd on sessioonihalduri klassis ainult funktsioonid, mis on vastavad tema struktuurile ja nimetusele. Kasutusele on võetud liides *IProvider*, mille abil saab failisüsteemi ja vahemälu klassid omavahel ühtse loogika järgi viia ja selle abil saab lahti korduvatest päringutest *Get* ja *Set* funktsioonides, millist varustajat kasutada. Varustaja valimine toimub ainult korra programmi esimesel laadimisel. Optimeerimise seisukohast on kasutusele võetud *generic* tüüp, et saaks objektid salvestada maha vastavalt nende tüübile ja ei peaks neid teisendama *object* tüüpi ja hiljem tagasi tema enda tüüpi. Joonisel 7 on esitatud klassidiagrammid sessioonihalduri refaktoriseerimata ja refaktoriseeritud programmikoodist, kus Lisa 1 esitatud koodi klassidiagrammi on kujutatud Joonisel 7a ja Lisa 2 esitatud koodi klassidiagrammi Joonisel 7b.



(a) (b)

Joonis 7. Analüüsitava programmikoodi klassidiagramm: (a) klassidiagramm Lisas 1 toodud programmikoodile, (b) klassiskeem Lisas 2 esitatud programmikoodile.

Kui vaadelda Lisa 1 ja Lisa 2 esitatud programmikoodi, siis saame neid võrrelda mitme erineva aspekti alusel. Need aspektid ja nende väärtused on toodud Tabelis 1. Väärtused on saadud 20 kordselt testimisel ja tabelis on esitatud enim esinenud väärtuste keskmine, kuna programmi töötsükkel võib olla häiritud teiste töötavate protsesside poolt.

Tabel 1. Sessioonihalduri võrdlusaspektid

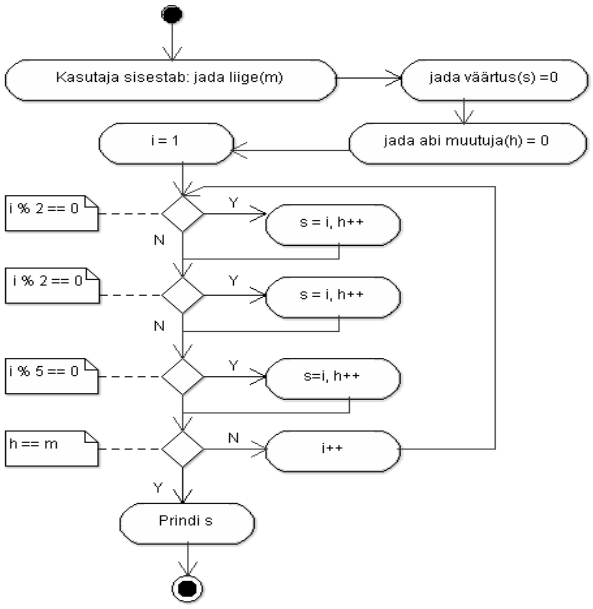
Omadus	**Esialgne kood (Lisa 1)**	**Refaktoriseeritud ja optimeeritud kood (Lisa 2)**
Programmikoodi pikkus (rida)	180	253
Programmikoodi suurus (bytes)	8 704	10 240
Programmikoodi töötsükli aeg (takt)	52080 = ~5,2ms	36270 = ~3,6ms
Programmikoodi mälukasutus (kb)	4361,6	4092,1

Võrdlusest selgub, et programmikoodi pikkus on variantidel märgatav, kuid siin tuleks märkida, et iga uue klassiga defineeritakse uuesti liidespaketid, mille pikkus sellest 253 reast on juba 40. Seega tegelik koodi pikkuse suurenemine on 253-40-180 = 33 rida ehk pikkuse kasv ca 15% Programmikoodi suurus baitides paistab olevat ka vastavalt paigas tema ridade arvuga (erinevus samuti ca 15%), kuid erinevust Lisa 2 kasuks on näha töötsükli ajas – võit ca 30%. Mõlemad variandid salvestasid mällu sama objekti (ainsuses) ja pärisid seda. Refaktoriseerimata programmikoodil, mis teisendab andmed *object* tüüpi on taktide arv pea 145% suurem kui refaktoriseeritud ja optimeeritud programmikoodil. Kui arvestada, et tulemus on saadud ühe objekti puhul, siis ligikaudselt 100000 objekti puhul on see refaktoriseerimata süsteemi puhul 5,2*100 000 = 520ms sekundit ja refaktoriseeritud süsteemi puhul 3,6 * 100 000 = 360ms ehk pea 1,5 korda kiirem. Muidugi oleneb see ka parajasti kasutatavatest objekti tüüpidest, mida sessiooni paigutatakse, seega saab väita, et üleval toodud 145% vahe on suhteline. Mälukasutuse testimisel kasutatakse viite erineva andmetüübi objekti (*string, int, decimal, datetime, bool*). Igat andmetüüpi lisatakse sessioonihaldurisse tuhat korda ja seejärel päritakse kõik need 5000 objekti. Tabelis olevad mälukasutuse analüüsi tulemused baseeruvad 5000 objekti rakendamisel. Tulemuste vahe on märgatav, kus Lisa 2 on pea 300kb väiksema mälukasutusega, mis tuleneb sellest, et Lisas 2 esitatud programmikood kasutab andmetüüpide töötlemisel geneerilisi andmetüüpe. Sellest võib järeldada, et geneerilised andmetüübid on palju kompaktsemad, kui seda on

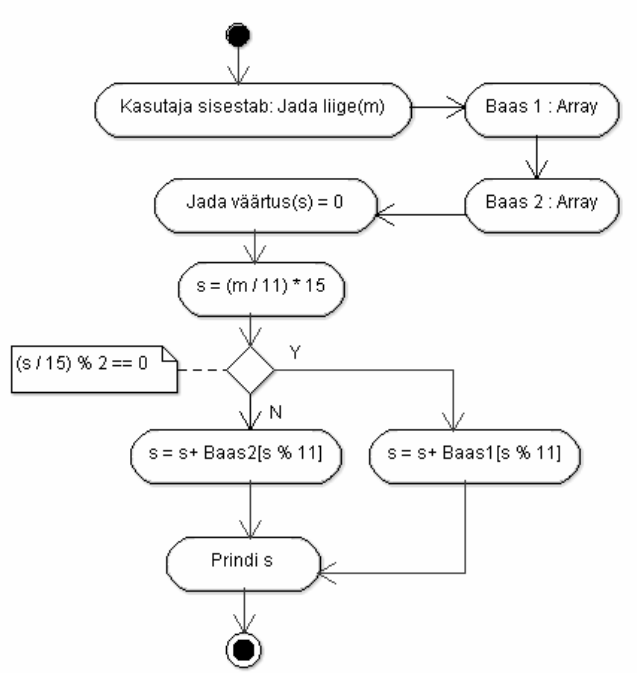
object andmetüübid. Ühesõnaga, teadmata objekti andmetüüpide puhul tasub alati kasutada geneerilisi andmetüüpe, kuna üks objekt võib koosneda mitmest teisest objektist. Nagu näiteks objekt Isik võib koosneda nimest, telefoninumbrist, vanusest, sünnikuupäevast jne, siis geneeriliste andmetüüpide mitte kasutamise puhul võib kasutatud mäluruum osutuda ebaotstarbekalt suureks.

4.3. Analüüsitava programmikoodi optimeerimine

Programmikoodi optimeerimise analüüsimiseks on programmikoodiks valitud arvude jadal põhinev ülesanne. Näites on tegemist matemaatilise ülesandega sellepärast, et neile leidub rohkem algoritme lahendamiseks. Lisas 3 on esitatud programmikood, mille töö põhimõtteks on leida arvude jada teatud liige. Jada on koostatud põhimõttel, et kõik arvud, mis jaguvad kahe, kolme või viiega on seatud ritta kasvavas järjekorras. Esitatud programmikoodi algoritm põhineb sellel, et ta hakkab alates ühest järjest läbi käima numbreid ja kui number vastab eelnevalt toodud kirjeldusele, siis ta teeb märke sellest. Iga järgmise kirjeldusele vastava numbriga ta suurendab nende arvude kogust ühe võrra, kuni jõuab liikmeni, mida kasutaja soovis. Arvude mällu salvestamist Lisa 3 esitatud programmiga ei toimu, vaid ta hoiab mälus ainult kahte arvu korraga ehk liiget ja mitmes liige ta on. Sellega hoidutakse ebavajalikust mälu kasutamisest. Kui selline protseduur oleks pidevalt korduv, siis oleks targem juba salvestada need liikmed mällu, et ei peaks seda jada koguaeg läbi käima. Programmikoodi (Lisa 3) algoritm on esitatud Joonisel 8 ja järgnevalt on tekstis esitatud kui Algoritm 1.



Joonis 8. Arvujada programmi esialgse algoritmi tegevuste diagramm (Algoritm 1).



Joonis 9. Arvujada programmi algoritmi optimeeritud tegevuste diagramm (Algoritm 2).

Algoritm 1 optimeeritud programmikood on esitatud Lisas 4 ja selle tegevuste diagramm Joonisel 9 (tekstis kui Algoritm 2). Algoritm 2 on sarnaseid tingimusi järgiv

programmikood, kuid programmikoodi algoritm võrreldes Algoritm 1-ga on erinev. Algoritm 2 programmikood ei hakka järjestikuliselt jada läbi käima, vaid arvutab selle teatud valemi järgi. Valem seisneb selles, et suurim erinev liige nende puhul saab olla kolm korda viis, mis on viisteist ja iga viieteist järjestikuse arvu kohta on jadas tingimustele vastavaid liikmeid üksteist. Huvitav on asja juures see, et kui võtta arvudest 1-15 moodul 15 saame ühe numbrite baasi ja 16-30 moodul 15 saame teise baasi ja peale seda hakkavad need baasid üle ühe korduma. Joonisel 10 on selle teooria tõestus, kus on näha, et arvude baasid hakkavad üle ühe korduma. Samuti paistab sellelt jooniselt välja, et iga 15 arvu kohta on kriteeriumitele vastavaid arve 11.

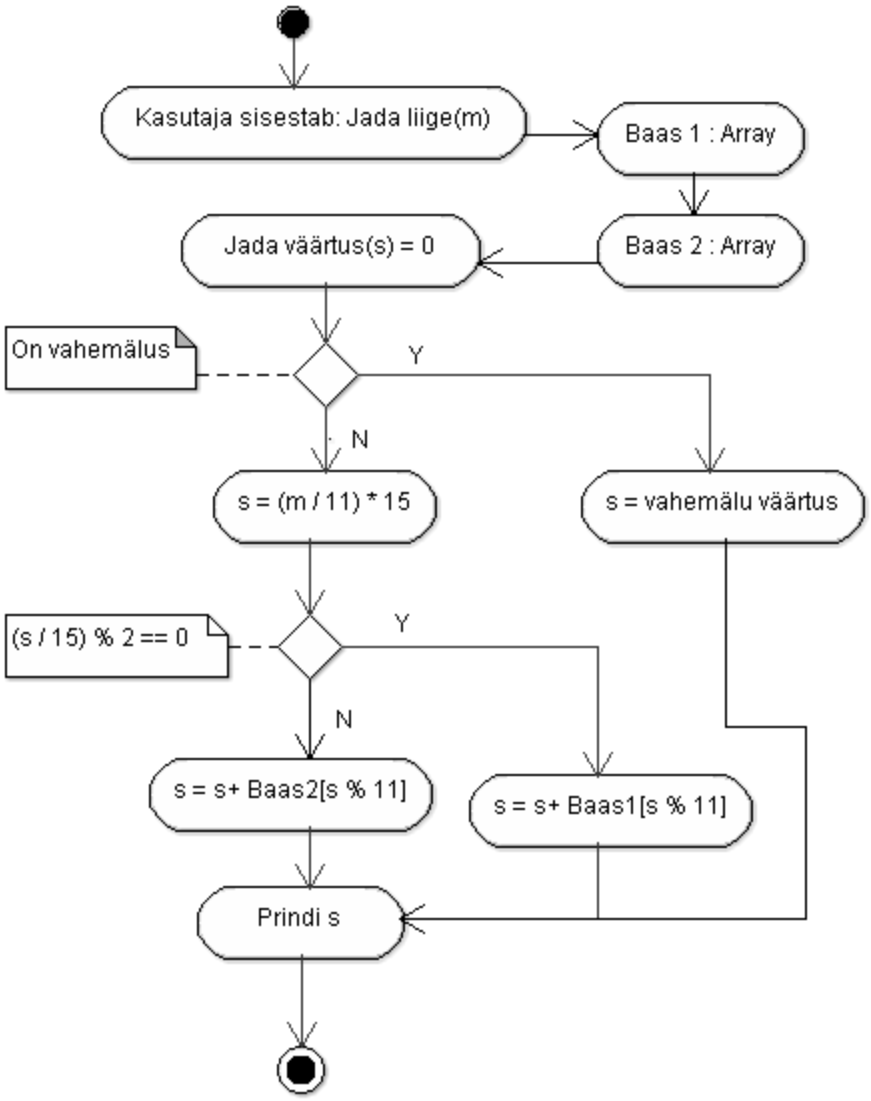
```
Max index 1500. Elementindex > 1500 = 1500
Enter wanted element index:1500
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -15: 11 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -30: 22 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -45: 33 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -60: 44 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -75: 55 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -90: 66 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -105: 77 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -120: 88 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -135: 99 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -150: 110 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -165: 121 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -180: 132 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -195: 143 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -210: 154 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -225: 165 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -240: 176 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -255: 187 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -270: 198 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -285: 209 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -300: 220 11
Base(2,3,4,5,6,8,9,10,12,14,15,) Elements Count -315: 231 11
Base(1,3,5,6,7,9,10,11,12,13,15) Elements Count -330: 242 11
```

Joonis 10. Algoritmi 2 baaside tõestus.

Jada liikme x leidmise valem Algoritm 2 korral:

$$element = (x \; mod \; 11) * 15 + baseIndex \; (x \; mod \; 11)$$

Elementide baasi saame valida vastavalt [(x mod 11) * 15] mod 2 –le. Kui see moodul on null, siis baasiks on paaritu baas ehk hetkel [2,3,4,5,6,8,9,10,12,14,15] ja kui moodul on üks, siis baasiks on paaris baas ehk hetkel [1,3,5,6,7,8,9,10,12,13,15]. Kusjuures taoline loogika sobib ka teiste tingimuste puhul. Teoreetiliselt peaks Lisas 4 esitatud algoritm arvutama välja jada liime kohal 50 000 ja 100 000 sama ajatsükli jooksul. Tabelist 2 on näha, et see teoreertiline väide peab ka praktikas paika.



Joonis 11. Vahemälu kasutava algoritmi tegevusdiagramm (Algoritm 3)

Algoritm 3 on sama lähteülesannet järgiv lahendus nagu Algoritm 1 ja Algoritm 3. Algoritm 3 programmikood on esitatud Lisas 5 ja selle tegevuste diagramm Joonisel 11. Algoritm 3 põhineb oma funktsionaalsuselt Algoritm 2-l. Algoritm 3 kasutab vahemälu ehk programmikoodi tasemel *MemoryCache*-i. *MemoryCache* on oma funktsionaalsuselt ideaalne sellise programmikoodi tööpõhimõtte jaoks, kuna igale vahemällu lisatud elemendile saab külge panna eraldi tema aegumise aja, mis välistab juhu, et vahemälu saab täis. Lõppkokkuvõttes jäävad mällu ainult sagedalt esinevad arvud. Oma töötsükli juures küsib Algoritm 3 kõigepealt vahemälust, kas selline element on olemas seal. Kui on siis

väljastab selle, kui ei siis arvutab selle kasutades Algoritm 2-te, lisab selle vahemällu ja väljastab. Algoritm 3 puhul on vahemälu elemendi aegumise ajaks pandud 10 sekundit.

Neid kolme algoritmi saab võrrelda omavahel peamiselt nende töötsükli kiiruse järgi. Kuna kõik kolm programmikoodid on üpriski lühikesed, millest tuleneb ka nende failisüsteemi sarnane suurus. Arvestades algoritmide töötsükleid ja kuidas hoitakse parameetreid mälus, siis mälukasutuselt esimesed kaks algoritmi väga erinevad ei ole, kuid kolmanda algoritmi mälu kasutus oleneb parajasti vahemälus olevate elementide hulgast.

Järgnevalt esitatud tabelis on väärtused erinevate algoritmi omaduste kohta, kus on omavahel võrreldud Algoritm 1, Algoritm 2 ja Algoritm 3. Väärtused iga tulba tabelis on saadud 20 kordselt algoritmi testmisel ja sealt saadud enim esinenud väärtus. Tabeli päises olevad arvud 500 000, 1 000 000 ja 1 000 001 on vastava jada elemendi koht. Algoritm 3 puhul on erinevus see, et 1 000 000 on vahemälus olev väärtus ja 1 000 001 mitte.

Tabel 2. Algoritmide programmikoodi omaduste võrdlus

Omadus	**Algoritm 1, 500 000**	**Algoritm 2, 500 000**	**Algoritm 1, 1 000 000**	**Algoritm 2, 1 000 000**	**Algoritm 3, 1 000 000**	**Algoritm 3, 1 000 001**
Programmikoodi pikkus (rida)	51	33	51	33	53	53
Programmikoodi suurus (byte)	5 120	5 632	5 120	5 632	6 144	6 144
Programmikoodi töötsükli aeg (takt)	24531	8	46444	8	66	23
Programmikoodi mälukasutus (kB)	56,6	56,8	56,6	56,8	279,1	**279,2**

Tabelist 2 paistab, et Algoritm 1 ja Algoritm 2 koodirea pikkus ja failisuurused ei ole omavahel vastavuses, kuid see võib tulla sellest, et Algoritm 2 puhul on baasid programmikoodi sisse kirjutatud ja nende puhul ei ole tegemist arenduskeskkonna poolsete funktsioonidega vaid konkreetsete parameetritega. Sama kehtib ka Algoritm 3 kohta, kuna tegemist on siiski Algoritm 2 edasiarendusega. Programmi töötsükli ajas aga on näha, et Algoritm 1 kohta võib väita, et töötsükli aeg suureneb lineaarselt vastavalt jadaliikmele, mis on igati ka loogiline. Arvestades, et Algoritm 1 käib terve tsükli läbi, kuni jõuab

tulemuseni. Algoritm 2 puhul sai kinnitust eelnevalt teoreetiline väide, et programmikoodi töötsükli aeg jääb sõltumatuks vastavalt jada liikme arvule, kuna liikme leidmiseks kasutatakse sama valemit. Algoritm 3 puhul on 1 000 000 vahemälus olev suurus ja 1 000 001 mitte. Sellest ka selline erinevus töötsükli ajas. Imelik on asja juures see, et vahemälust võetava väärtuse saamiseks kulub pea kolm korda rohkem aega, mis peaks olema tegelt vastupidi. Järeldada saab sellest seda, et sellise algoritmi mõistes ei ole kasulik vahemälu kasutada, kuna valemi põhjal arvutus on tunduvalt kiirem. Samuti arvutusega saadud väärtue jaoks kulub samuti kolm korda rohkem aega kui Algoritm 2 puhul. Arvatavasti suurendab seda aega lõpus vahemälusse lisamise funktsioon. Siit nähtub, et valemile toetuv programmikood on tsüklilisest koodist tunduvalt kiirem. Täpsemalt 24531 / 8 = ~3066 korda kiirem ja vahemälu kasutavast koodist 66 / 8 = ~8 korda kiirem. Mälukasutuselt on Algoritm 1 ja Algoritm 2 praktiliselt võrdsed. Mõlemad kasutavad oma töötsükli ajal ~60kb mälu. Algoritm 3 kasutab aga pea kolm korda rohkem kui Algoritm 1 ja Algoritm 2. See tuleneb sellest, et väärtusi hoitakse vahemälus. Kuna aruandest selgub, et optimeeritud algoritm ehk valemile toetuv algoritm on pea igas võrdlusaspektis parem kui esialgne ehk optimeerimata programmikood, siis võib teha järelduse, et esialgne optimeerimata algoritm on kasutu. Samuti saab ka väita seda, et sellise eesmärgiga algoritmide puhul ei ole mõistlik kasutada vahemälu. Algoritm 1 võiks teoorias olla kasulik juhul, kui programmi käivitamisel kõik liikmed salvestada vahemällu, mis küll võtab kauem aega ja kasutab liigselt mälu kuid hiljem arvujada liikme päringu puhul saaks selle liikme otse mälust küsida ja ei peaks selleks tegema uusi arvutusi, kuid Algoritm 3 kahjuks lükkab selle teoreetilise väite ümber sellise ülesande püstituse juures. Suurte objektide puhul ja meeletute arvutuste tegemise juures kindlasti oleks see õigustatud lähenemine.

5. Kokkuvõte

Käesoleva bakalaureusetöö teemaks oli „Programmikoodi refaktoriseerimine ja optimeerimine C# näitel“.

Uurimise eesmärgiks oli leida, millised programmikoodi refaktoriseerimise või optimeerimise võtted on kõige kasulikumad C# keskkonnas. Selle uuringu tulemusena tulid välja paljud asjaolud, mis ei vastanud esialgsetele ootustele. Nii saab lükata ümber osad teoreetilised väited, mis autor püstitas teema arenduses nagu näiteks väide, et vahemälu kasutamise kasulikkus on alati olemas.

Lõputöö koostamisel püstitas autor eesmärgi täitmiseks järgmised uurimisülesanded: selgitada välja refaktoriseerimise eelised ja puudused; millal oleks mõistlik olemasolevat programmikoodi refaktoriseerida; mida kujutab endast programmikoodi optimeerimine ja kuidas seda rakendada.

Programmikoodi refaktoriseerimine on programmikoodi kvaliteedi tõstmine nii struktuurselt kui loetavuselt. Samuti likvideerib refaktoriseerimine programmikoodist ära kordused ehk duplikaadid, läbi mille toimub ka kaudne programmikoodi optimeerimine. Refaktoriseerimiseks on C# keskkonnas mitmeid erinevaid tehnikaid. C# erilisus võrreldes teiste keeltega on see, et ta omab nii-öelda *generic* tüüpi objekte, millel otseselt puudub sisu, aga kui *generic* tüüpi objektile määrata mingi objekt, siis *generic* tüüp võtab enda väärtuseks selle objekti väärtuse. Teised tehnikad on üldiselt läbivad kõikides teistes programmeerimiskeeltes. Programmikoodi refaktoriseerimisel tuleb selgitada, kas selle protseduuri läbiviimine on antud projektis üldse kasulik, kuna refaktoriseerimine uut funktsionaalsust programmile ei lisa, küll toob endaga aga kaasa kulusid.

Programmikoodi optimeerimine seisneb programmi töötsükli algoritmi valikus või selle välja vahetamises. Erinevatel algoritmidel on erinevad eelised. Samuti saab erinevaid algoritme kombineerida vahemälu kasutamisega, kui tegemist on suurte objektidega. Programmeerimise maailmas täielik optimaalsus puudub. Teatud aspekti kvaliteedi

tõstmine tähendab teise aspekti kvaliteedi langetamist nagu näiteks programmi töötsükli kiirus tuleneb üldiselt vahemälu kasutusest. Optimeerimiseks on olemas palju erinevaid automaatseid optimeerijaid, millest kõige tavalisem on programmikoodi kompilaator. Kompileerimisel programmikood optimeeritakse, kuna kõrgtaseme programmeerimise keelest saab edasi assembler keel ja selle muutmise käigus teatud algoritmide järgi kompilaator seda teebki. Programmikoodi optimeerimiseks on olemas eraldi programmi analüüsimise tarkvarad mida nimetatakse profileerijateks. Profileerijatega saab leida programmikoodi mälu lekkeid ja ’pudelikaelu’, mis neid põhjustavad. Samuti saab nendega vaadata näiteks andmebaasi päringuid.

Optimeerimine ja refaktoriseerimine on kaudselt sarnased tegevused, kuid optimeerimine võib olla vastupidine refaktoriseerimisele, kus väga rasked ja keerukad algoritmid võivad programmikoodi loetavuse praktiliselt võimatuks teha.

Kasutatud kirjandus

[1] Fowler Martin : „Refactoring: Improving the design of existing code“. Addison-Wesley Longman Publishing Co., Inc (1999) .

[2] Mõiste [WWW] http://www.webopedia.com/TERM/R/refactoring.html [Online].

[3] Kerievsky Joshua: „Refactoring to Patterns“. Addison-Wesley Longman Publishing Co., Inc (2004).

[4] Griswold, William G: „*Program Restructuring as an Aid to Software Maintenance“*. (July 1991) Ph.D. thesis. University of Washington. Retrieved 2011-12-24.

[5] Martin Fowler „*Martin Fowler Bliki : Etymology of Refactoring*“(10.09.2003) (online artikkel).

[6] Martin Fowler, Jay Fields, Shane Harvie : „Refactoring Ruby Edition“. (2009).

[7] Abrams, Meyer Howard; Harpham, Geoffrey Galt : „A Glossary of Literary Terms“ (2011).

[8] Luciano Floridi, J. W. Sanders „*Levellism and the Method of Abstraction*“ Research Report (22.11.04).

[9] Danijel Arsenovski „Professional refactoring in C# and asp.net“. Wiley Publishing, Inc. (2009).

[10] D. Advani, Y. Hassoun, and S. Counsell. „*Understanding the complexity of refactoring in software systems: a tool-based approach“*. Intl. J. Gen. Sys, 35(3): 329-346, (2006) (e-artikkel).

[11] Joonis [WWW] http://sourcemaking.com/files/sm/images/07fig03.gif (pilt veebist).

[12] Robert Sedgewick , Kevin Wayne : „Algorithms“. Addison-Wesley Longman Publishing Co., Inc. (1984).

[13] Moschovakis, Yiannis N. (2001). "*What is an algorithm?"*. In Engquist, B.; Schmid, W. Mathematics Unlimited — 2001 and beyond. Springer. pp. 919–936 (Part II). (e-artikkel).

[14] Donald Knuth: „The art of computer programming“. Addison-Wesley Longman Publishing Co., Inc (1968).

[15] Donald Knuth: „The Errors of Tex“, (July 1989), (e-artikkel).

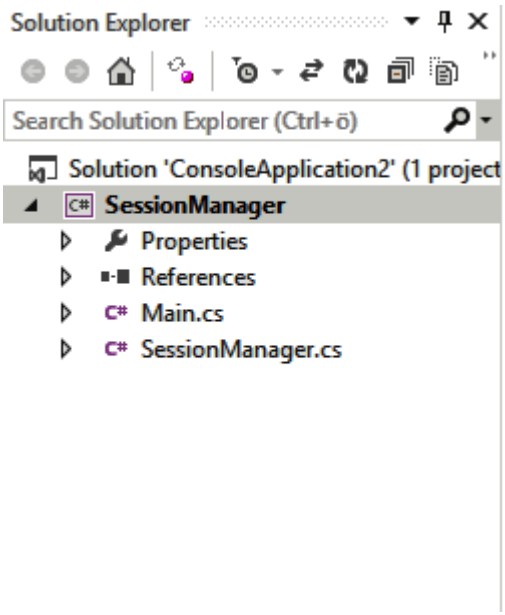
[16] Knuth Donald: "Structured Programming with go to Statements". (December 1974).(e-artikkel).

[17] Cooper, Keith D., and Torczon, Linda: „Engineering a Compiler“, Morgan Kaufmann, (2004).

[18] Net Memory Profiler koduleht [WWW] http://memprofiler.com/ [Online].

LISA 1. Refaktoriseerimata sessioonihalduri kood

Sessioonihalduri esialgne kood, mida ei ole refaktoriseeritud ega optimeeritud.



Ülemisel joonisel on kujutatud sessioonihalduri struktuur. Sessioonihalduri kood on järgmine:

Main.cs-i sisu:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SessionManagerNotRefactored
{
    class Main
    {
        static void Main(string[] args)
        {
            // initialize
            SessionManager manager = new SessionManager();
```

```
            manager.SetConfig(SessionConfig.Filesystem);
            // set
            manager.Set("string", "hello world");
            // get
            var obj = manager.Get("string");
            Console.WriteLine(obj.CreatedAt + " " + obj.Object);

        }
    }
}
```

SessionManager.cs-i sisu:

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq;
4.  using System.Text;
5.  using System.Configuration;
6.  using System.Runtime.Caching;
7.  using System.IO;
8.
9.  namespace SessionManagerNotRefactored
10. {
11.     public enum SessionConfig
12.     {
13.         MemoryCahce = 1,
14.         Filesystem = 2,
15.         Sql = 3
16.
17.     }
18.
19.     public class SessionObject
20.     {
21.        public DateTime? CreatedAt { get; set; }
22.         public object Object { get; set; }
23.         public DateTime? ModifiedAt { get; set; }
24.     }
25.

26.     public class SessionManager
27.     {
```

```
28.         private SessionConfig SessionConfig;
29.         private ObjectCache Cache { get; set; }
30.         private readonly CacheItemPolicy _policy = null;
31.         public SessionManager()
32.         {
33.             ReadConf();
34.             Cache = MemoryCache.Default;
35.             _policy = new CacheItemPolicy
36.             {
37.                 AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(2)
38.             };
39.
40.         }
41.
42.         private void ReadConf()
43.         {
44.
45.             var conf = ConfigurationManager.AppSettings["Config"];
46.             if (conf == "cache") SessionConfig = SessionConfig.MemoryCahce;
47.             if (conf == "file") SessionConfig = SessionConfig.Filesystem;
48.             if (conf == "sql") SessionConfig = SessionConfig.Sql;
49.         }
50.
51.         public void SetConfig(SessionConfig conf)
52.         {
53.             SessionConfig = conf;
54.         }
55.
56.         public void Set(string key, object o)
57.         {
58.             var sessionObject = new SessionObject
59.             {
60.                 CreatedAt = DateTime.Now,
61.                 Object = o
62.             };
63.             if(SessionConfig == SessionConfig.MemoryCahce){
64.                 MemoryCache_Set(key, sessionObject);
65.             }
66.             if (SessionConfig == SessionConfig.Filesystem)
67.             {
68.                 FileSystem_Set(key, sessionObject);
```

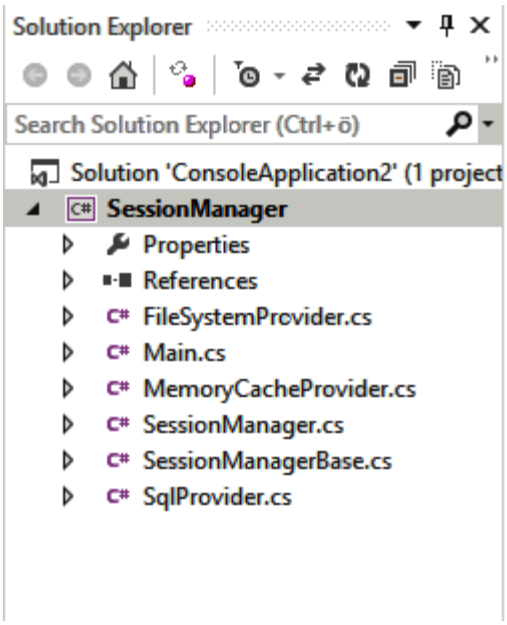
```
69.             }
70.             if (SessionConfig == SessionConfig.Sql)
71.             {
72.                 // not implemented
73.             }
74.
75.         }
76.         public SessionObject Get(string key)
77.         {
78.             if (SessionConfig == SessionConfig.MemoryCahce)
79.             {
80.                 return MemoryCache_Get(key);
81.             }
82.             if (SessionConfig == SessionConfig.Filesystem)
83.             {
84.                 return FileSystem_Get(key);
85.             }
86.             if (SessionConfig == SessionConfig.Sql)
87.             {
88.                 // not implemented
89.             }
90.             // something went baddd
91.             return null;
92.
93.         }
94.
95.         private void MemoryCache_Set(string key, SessionObject obj)
96.         {
97.             var size = Cache.Count();
98.             Cache.Add(key, obj, _policy);
99.
100.         }
101.
102.         private SessionObject MemoryCache_Get(string key)
103.         {
104.             var obj = Cache.Get(key) as SessionObject;
105.             return obj;
106.         }
107.
108.         private void FileSystem_Set(string key, SessionObject o)
109.         {
```

```
110.            var objectToWrite = o;
111.            if (objectToWrite == null) return;
112.            var writer = new StreamWriter("C:/test/data.txt", true);
113.            var line = key + "|" + objectToWrite.CreatedAt + "|"
114.         objectToWrite.Object + "|" + objectToWrite.ModifiedAt;
115.            writer.WriteLine(line);
116.            writer.Close();
117.
118.        }
119.
120.        private SessionObject FileSystem_Get(string key)
121.        {
122.            var result = ReadItemFromFile(key);
123.            var result2 = new SessionObject
124.            {
125.                CreatedAt = result.CreatedAt,
126.                Object = result.Object,
127.                ModifiedAt = result.ModifiedAt
128.            };
129.            return result2;
130.
131.        }
132.
133.        private SessionObject ReadItemFromFile(string key)
134.        {
135.            var reader = new StreamReader("C:/test/data.txt");
136.            while (true)
137.            {
138.                var line = reader.ReadLine();
139.                if (line == null) break;
140.                var values = line.Split('|');
141.                if (values[0] == key)
142.                {
143.
144.                    return new SessionObject
145.                    {
146.                        CreatedAt = values[1] != "" ?
147.         Convert.ToDateTime(values[1]) : new DateTime(),
148.                        Object = values[2],
149.                        ModifiedAt = values[3] != "" ?
150.                Convert.ToDateTime(values[3]) : new DateTime()
```

```
151.
152.                     };
153.                 }
154.
155.             }
156.             return null;
157.         }
158.     }
159.
160. }
```

LISA 2. Refaktoriseeritud sessioonihalduri kood

Refaktoriseeritud sessioonihalduri struktuur



Järgnevalt on välja toodud programmikoodi osad.

Main.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Configuration;
using System.Dynamic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class MainProgram
    {
```

```
        static void Main(string[] args)
        {
            var sessionManager = new SessionManager();
            // set value
            sessionManager.Set<DateTime>("aeg", DateTime.Now);
            // value
            var obj = sessionManager.Get<DateTime>("aeg");
            // print
            Console.WriteLine(obj.CreatedAt + " " + obj.Object);


        }

    }


}
```

SessionManagerBase:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    public class SessionManagerBase
    {
        protected SessionConfig SessionConfig { get; set; }
        // provider
        protected IProvider Provider { get; set; }


        protected void ReadConfig()
        {
            var conf = ConfigurationManager.AppSettings["Config"];
            if (conf == "cache") SessionConfig = SessionConfig.MemoryCahce;
            if (conf == "file") SessionConfig = SessionConfig.Filesystem;
```

```
            if (conf == "sql") SessionConfig = SessionConfig.Sql;
            GetProvider(SessionConfig);
        }

        protected void GetProvider(SessionConfig config)
        {
            switch (SessionConfig)
            {
                case SessionConfig.MemoryCahce:
                    Provider = new MemoryCaching();
                    break;
                case SessionConfig.Filesystem:
                    Provider = new Filesystem();
                    break;
                case SessionConfig.Sql:
                    Provider = new MemoryCaching();
                    break;
            }
        }

        protected void SetProvider(SessionConfig config)
        {
            SessionConfig = config;
            GetProvider(SessionConfig);
        }
    }
    public class SessionObject<T>
    {
        public DateTime? CreatedAt { get; set; }
        public T Object { get; set; }
        public DateTime? ModifiedAt { get; set; }
    }

    public enum SessionConfig
    {
        MemoryCahce = 1,
        Filesystem = 2,
        Sql = 3

    }
}
```

SessionManager.cs

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    public class SessionManager : SessionManagerBase
    {

        // constructor
        public SessionManager()
        {
            ReadConfig();
        }

        public void SetConfig(SessionConfig config)
        {
            SetProvider(config);
        }

        public void Set<T>(string key, T o)
        {
            var sessionObject = new SessionObject<T>
            {
                CreatedAt = DateTime.Now,
                Object = o
            };

            Provider.Set(key, sessionObject);
        }

        public SessionObject<T> Get<T>(string key)
        {
             return Provider.Get<T>(key);
        }
```

```
    }
}
```

MemoryCacheProvier.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Caching;
using System.Runtime.Remoting.Messaging;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{


    public class MemoryCaching : IProvider
    {
        private ObjectCache Cache { get; set; }
        private readonly CacheItemPolicy _policy = null;


        public MemoryCaching()
        {
            Cache = MemoryCache.Default;
            _policy = new CacheItemPolicy
            {
                AbsoluteExpiration = DateTimeOffset.Now.AddMinutes(2)
            };
        }
        public void Set<T>(string key, SessionObject<T> obj)
        {
            var size = Cache.Count();
            Cache.Add(key, obj, _policy);

        }
        public SessionObject<T> Get<T>(string key)
        {
            var obj = Cache.Get(key) as SessionObject<T>;
            return obj;
```

```
        }
    }
}
```

FileSystemProvider

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    public class Filesystem : IProvider
    {

        public Filesystem()
        {
            if (File.Exists("C:/test/data.txt"))
            {
                File.Delete("C:/test/data.txt");
            }
        }

        public void Set<T>(string key, SessionObject<T> o)
        {
            var objectToWrite = o;
            if (objectToWrite == null) return;
            var writer = new StreamWriter("C:/test/data.txt",true);
            var line = key + "|" + objectToWrite.CreatedAt + "|" +
objectToWrite.Object + "|" + objectToWrite.ModifiedAt;
            writer.WriteLine(line);
            writer.Close();

        }
        public SessionObject<object> ReadItemFromFile(string key)
        {
            var reader = new StreamReader("C:/test/data.txt");
            while(true){
```

```
            var line = reader.ReadLine();
                if (line == null) break;
            var values = line.Split('|');
                if(values[0] == key){

                    return new SessionObject<object>
            {
                CreatedAt = values[1] != "" ? Convert.ToDateTime(values[1]) :
new DateTime(),
                Object = values[2],
                ModifiedAt = values[3] != "" ? Convert.ToDateTime(values[3]) :
new DateTime()

            };
                }

            }
            return null;
        }
        public SessionObject<T> Get<T>(string key)
        {
            var result = ReadItemFromFile(key);
            var result2 = new SessionObject<T>{
                CreatedAt = result.CreatedAt,
                Object = (T)result.Object,
                ModifiedAt = result.ModifiedAt
            };
            return result2;

        }
    }
}
```

IProvider

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography.X509Certificates;
using System.Text;
using System.Threading.Tasks;
```

```
namespace ConsoleApplication1
{
    public interface IProvider
    {
        void Set<T>(string key, SessionObject<T> obj);
        SessionObject<T> Get<T>(string key);
    }
}
```

LISA 3. Tsükliline algoritm

Arvujada programmikood, kus jadaliikme leidmine toimub tsükliliste sammude alusel.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace Algo1_tsükkel
{
    class Program
    {
        static void Main(string[] args)
        {
            // INITILIZE
            int Divider_1 = 2;
            int Divider_2 = 3;
            int Divider_3 = 5;
            // current element
            int Element = 1;
            // elements count
            int ElementsCount = 0;
            // element index which is asked from user
            int ElementIndex = 0;
            Console.WriteLine("Enter wanted element index:");
            ElementIndex = Convert.ToInt32(Console.ReadLine());
            while (true)
            {
                if (Element % Divider_1 == 0)
                {
                    ElementsCount += 1;
                }

                else if (Element % Divider_2 == 0)
                {
                    ElementsCount += 1;
```

```
                }

                else if (Element % Divider_3 == 0)
                {
                    ElementsCount += 1;
                }

                if (ElementsCount == ElementIndex)
                {

                 Console.WriteLine("Element at index {0} is {1}", ElementIndex,
Element);
                    break;
                }

                Element++;

            }
        }
    }
}
```

LISA 4. Valemile toetuv algoritm

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace algo_valem
{
    class Program
    {
        static void Main(string[] args)
        {
            int Sequence = 15;
            int ElementsInSequence = 11;
            // base for even
            int[] Base_Even = {2,3,4,5,6,8,9,10,12,14,15};
            // base for odd
            int[] Base_Odd = {1,3,5,6,7,9,10,11,12,13,15};
            // element index which is asked from user
            int ElementIndex = 0;
            Console.WriteLine("Enter wanted element index:");
            ElementIndex = Convert.ToInt32(Console.ReadLine());
            int Element = 0;
            Element = (ElementIndex / ElementsInSequence) * Sequence;
            if((Element / Sequence) % 2 == 0) {
                Element += Base_Even[(ElementIndex % ElementsInSequence) -1];
            }
            else{
                Element += Base_Odd[(ElementIndex % ElementsInSequence) -1];
            }
            Console.WriteLine("Element at index {0} is {1}", ElementIndex,
Element);

        }
    }
}
```

Lisa 5. Valemile toetuv algoritm, mis kasutab ka vahemälu

```
using System;
using System.Runtime;
using System.Diagnostics;
using System.Runtime.Caching;


namespace algo_valem
{
	class Program
	{
	// Cache
		public static ObjectCache Cache = MemoryCache.Default;
		static void Main(string[] args)
		{
			Stopwatch watch = new Stopwatch();
			int Sequence = 15;
			int ElementsInSequence = 11;
			// base for even
			int[] Base_Even = { 2, 3, 4, 5, 6, 8, 9, 10, 12, 14, 15 };
			// base for odd
			int[] Base_Odd = { 1, 3, 5, 6, 7, 9, 10, 11, 12, 13, 15 };
			// element index which is asked from user
			int ElementIndex = 0;
			string strElementIndex = "";
			Console.WriteLine("Enter wanted element index:");
			strElementIndex = Console.ReadLine();
			ElementIndex = Convert.ToInt32(strElementIndex);
			Cache.Add("1000000", (int) 1000000, new CacheItemPolicy());
			Cache.Add("500000", "500000", new CacheItemPolicy());
			watch.Start();
			int Element = 0;
			if (strElementIndex != null &&
			Cache.Contains(strElementIndex))
			{
				Element = Convert.ToInt32(Cache[strElementIndex]);
			}
```

```
                else
                {
                        Element = (ElementIndex / ElementsInSequence) *
                Sequence;
                        if ((Element / Sequence) % 2 == 0)
                        {
                                Element += Base_Even[(ElementIndex %
                        ElementsInSequence) - 1];
                        }
                        else
                        {
                                Element += Base_Odd[(ElementIndex %
                        ElementsInSequence) - 1];
                        }
                        // add to cache
                        Cache.Add(Element.ToString(), Element.ToString(), new
                CacheItemPolicy());
                }
                watch.Stop();
                Console.WriteLine("Element at index {0} is {1}",
        ElementIndex, Element);
                Console.WriteLine(watch.ElapsedTicks);
        }
    }
}
```