TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

# SALVESTATUD PROTSEDUURIDE JA KOLMEKIHILISE ARHITEKTUURI ANALÜÜS

MAGISTRITÖÖ

| | |
|---|---|
| Üliõpilane: | Reio Kokla |
| Üliõpilaskood: | 111531IABM |
| Juhendaja: | Ants Torim |

Tallinn
2015

# 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.

---------------------------------------------------- -------------------------------------------------

(*kuupäev*) (*allkiri*)

# Annotatsioon

Töö eesmärk on uurida ja võrrelda rakenduse äriloogika hoidmise erinevusi salvestatud protseduurides ja rakenduse koodis. Autor uurib, kas salvestatud protseduurid pakuvad arenduses samu võimalusi kui objektorienteeritud kood. Samuti uuritakse antud töös, milliseid aspekte peaks arvesse võtma, kui otsustatakse, kas äriloogika kirjutada salvestatud protseduuridesse ning mis on viimase puudujäägid võrreldes rakendusesisese äriloogikaga.

Antud töös võrdleb autor äriloogika paigutamist rakendusse ja salvestatud protseduuridesse järgmistes aspektide lõikes: jõudlus, testitavus, modifitseeritavus, duplitseerimine ning kapseldus.

Töö tulemusena on välja toodud aspektid, mille järgi teha äriloogika paigutamise otsust.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 64 leheküljel, 3 peatükki, 15 joonist, 3 tabelit.

# Abstract

The aim of the thesis is to investigate and compare two approaches where to hold application business logic: should it be in the application or stored procedures. The author examines whether stored procedures offer same opportunities as object oriented code. Also, author investigates which aspects should be taken into account when deciding whether to write business logic into stored procedures and what are the shortcomings compared to holding business logic in application code.

The author compares whether to put business logic into application of stored procedures within next aspects: performance, testability, modifiability, avoiding duplication and encapsulation.

Thesis results indicate aspects which should be taken into account while deciding where to hold business logic. The thesis also analyses advantages and disadvantages of both approaches.

The thesis is in Estonian and contains 64 pages of text, 3 chapters, 15 figures, 3 tables., etc.

## Lühendite ja mõistete sõnastik

**ISO** — ***International Organization for Standardization***
Rahvusvaheline Standardiorganisatsioon – valitsusväline rahvusvaheline organisatsioon, mis tegeleb rahvusvaheliste tööstuslike ja kommertsstandarditega

**TCP/IP** — ***Transmission Control Protocol / Internet Protocol***
Internetiprotokollistik, millega tagatakse võrkude toimimine (nt internet)

**CRC** — ***Class-responsibility-collaboration***
Rakenduse klassi vastutuse ja koostöö disainimisel kasutatav kaart

**ITU** — ***International Telecommunication Union***
Rahvusvaheline Telekommunikatsiooni Liit

**ITU-T** — ***ITU Telecommunication Standardization Sector***
ITU Telekommunikatsiooni Standardiseerimise sektor

**OSI** — ***Open Systems Interconnection***
Avatud süsteemide sidumise arhitektuur – ISO ja ITU-T koostöös 1977. aastal valminud andmesideprotokollide kontseptuaalne mudel (OSI-mudel)

**W3C** — ***World Wide Web Consortium***
Rahvusvaheline organisatsioon, mille missiooniks on viia veeb tema täieliku potentsiaalini, arendades protokolle ja juhtnööre, mis kindlustavad veebi pikaajalise kasvu

**XML** — ***Extensible Markup Language***
Laiendatav märgistuskeel – W3C poolt väljatöötatud ja soovitatud standardne üldotstarbeline märgistuskeel

**SQL** — ***Structured Query Language***

Struktuurpäringukeel ehk andmebaasi päringukeel

**DRY** — ***Don't Repeat Yourself***
Ära korda ennast disainipõhimõte

**API** — ***Application Programming Interface***
Rakendusliides ehk programmiliides – reeglistik olemasoleva valmisprogrammiga suhtlemiseks

**IDE** — ***Integrated Development Environment***
Integreeritud programmeerimiskeskkond

**FIFO** — ***First In, First Out***
Lihtjärjekorra meetod, mille puhul eeldatakse, et alati realiseeritakse esimesena soetatud varud esmajärjekorras

**LIFO** — ***Last In, First Out***
Vastupidine FIFO meetodile – eeldatakse, et alati realiseeritakse viimati soetatud varud esmajärjekorras

**CRUD** — ***Create, Read, Update and Delete***
Loo, loe, uuenda ja kustuta – 4 põhilist funktsiooni andmetöötluses

# Jooniste nimekiri

Joonis 1. OSI 7-kihiline mudel ........ 13
Joonis 2. Üksiku kihi CRC kaart ........ 14
Joonis 3. 3-kihiline arhitektuur ........ 18
Joonis 4. Turuväärtuse arvutus Javas ........ 35
Joonis 5. Ühiktestid Javas ........ 37
Joonis 6. Alamklasside muutujate deklareerimine ülemklassis ........ 39
Joonis 7. *FifoStrategy* klass peale refaktoreerimist ........ 39
Joonis 8. Funktsioon ostutehingute tuvastamiseks ........ 40
Joonis 9. Ostutehingute tegevuste protseduur ........ 41
Joonis 10. Väärtpaberi tootluse arvutuse protseduur peale refaktoreerimist ........ 41
Joonis 11. Uue tüübi deklareerimine ........ 42
Joonis 12. Protseduuri sisendid ja väljundid enne ........ 43
Joonis 13. Protseduur väljastab uue tüübi ........ 43
Joonis 14. Muutujate deklareerimine protseduuris ........ 43
Joonis 15. Muutujate deklareerimine vaate pealt ........ 44

# Tabelite nimekiri

Tabel 1. Tootluse arvutuse jõudlustesti tulemused.................................................................32
Tabel 2. Turuväärtuse arvutuse jõudlustesti tulemused..........................................................35
Tabel 3. Võrdluste kokkuvõte ......................................................................................45

## Sisukord

1. Sissejuhatus ..... 11
1.1 Taust ja probleem ..... 11
1.2 Ülesande püstitus ..... 11
1.3 Metoodika ..... 11
1.4 Ülevaade tööst ..... 12
2. Kihiline arhitektuur ..... 13
2.1 Kihilise arhitektuuri teke ..... 15
2.2 Kolmekihiline arhitektuur ..... 16
2.2.1 Esitluskiht ..... 18
2.2.2 Domeenikiht ..... 19
2.2.3 Andmeliidestuskiht ..... 20
2.3 Domeenikihi mustrid ..... 21
2.3.1 Transaktsiooni skript ..... 21
2.3.2 Domeenimudel ..... 22
2.3.3 Teenuste kiht ..... 22
2.3.4 Tabeli moodulid ..... 23
3. Äriloogika – rakenduses või protseduuris ..... 24
3.1 Salvestatud protseduurid ..... 24
3.2 ORM ..... 26
3.3 Äriloogika asukoha valimise aspektid ..... 28
3.3.1 Jõudlus ..... 28
3.3.2 Testitavus ..... 28
3.3.3 Modifitseeritavus ..... 29
3.3.4 Arusaadavus ..... 29
3.3.5 Duplitseerimise vältimine ..... 29
3.3.6 Kapseldamine ..... 29
3.3.7 Andmebaasi teisaldatavus ..... 30
4. Võrdlus ..... 31
4.1 Jõudlus ..... 32
4.1.1 Tootluse arvutuse test ..... 32

4.1.2 Turuväärtuse arvutuse test ........ 34
4.2 Testitavus ........ 36
4.3 Modifitseeritavus ........ 38
4.3.1 Java koodi refaktoreerimine ........ 38
4.3.2 Andmebaasiprotseduuri refaktoreerimine ........ 40
4.4 Duplitseerimise vältimine ........ 42
4.5 Kapseldamine ........ 44
4.6 Võrdluse kokkuvõte ........ 45
5. Kokkuvõte ........ 47
Summary ........ 49
Kasutatud kirjandus ........ 50
Lisa 1. Andmebaasitabelite struktuur ........ 52
Lisa 2. FIFO meetodi arvutuse klass ........ 53
Lisa 3. Keskmise meetodi arvutuse klass ........ 55
Lisa 4. FIFO meetodi arvutuse protseduur ........ 56
Lisa 5. Keskmise meetodi arvutuse protseduur ........ 58
Lisa 6. Turuväärtuse arvutuse protseduur ........ 59
Lisa 7. Turuväärtuse arvutuse protseduur tsükliga ........ 60
Lisa 8. Andmebaasi ühiktestid ........ 61
Lisa 9. Uued protseduurid peale FIFO meetodi arvutuse protseduuri refaktoreerimist ........ 63

# 1. Sissejuhatus

Iga rakendus hõlmab endas suuremal või vähemal määral äriloogikat. Äriloogika paigutamiseks on tänapäeval kaks põhilist lähenemist – hoida seda rakendusesiseselt või andmebaasiprotseduurides. Üldiselt valitakse selline lähenemine, millega ollakse rohkem kursis või mida ollakse harjunud kasutama, analüüsimata kummagi lähenemise eeliseid ja kitsaskohti.

## 1.1 Taust ja probleem

Mõlemal lähenemisel on oma eelised ja puudused, miks neid kasutada. Kui rakendus on tuvastatud jõudluskriitilisena, siis võidakse soovida kasutada andmebaasiprotseduure äriloogika jaoks. Probleem võib tekkida hiljem aga hoopis teistes aspektides, näiteks koodi hallatavuses, arusaadavuses jm aspektides ning tegelikult ei pruugi ka jõudlus olla alati andmebaasiprotseduurides kiirem.

Antud töö tulemus on vajalik rakenduste arendajatele ja arhitektidele. Esiteks selleks, et mõelda erinevatele aspektidele, mis mõjutavad äriloogika paigutamise valikut. Teiseks annab antud töö ka võrdlusmomendi, kuidas sarnaseid probleeme lahendada kummaski valikus.

## 1.2 Ülesande püstitus

Töö eesmärgiks on välja selgitada, milliseid aspekte tuleks analüüsida, kui otsustatakse, kuhu paigutada äriloogika. Samuti on töö eesmärgiks välja tuua erinevused leitud aspektide võrdluses mõlemas keskkonnas. Lisaks soovib autor töö käigus jõuda soovitusteni, millistel juhtudel tuleks äriloogika paigutada rakendusse ning millistel juhtudel salvestatud protseduuridesse.

## 1.3 Metoodika

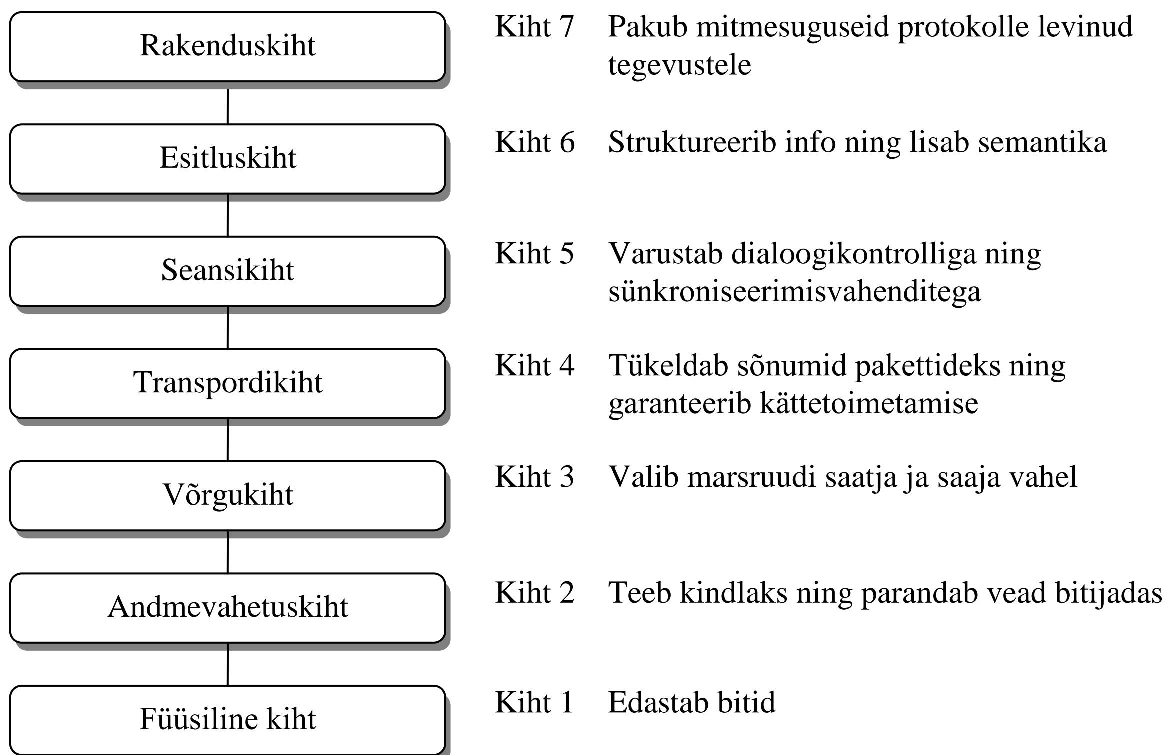
Töö eesmärkide saavutamiseks koostab autor rakenduse Java objektorienteeritud keeles ning arendab sama äriloogika ka andmebaasiprotseduurides. Seejärel rakendab autor mõlema lahenduse peal teste ning võrdleb nende tulemusi omavahel.

## 1.4 Ülevaade tööst

Antud töö on jagatud kolme peatükki. Esimeses teoreetilises peatükis annab autor ülevaate kihilisest arhitektuurist – põhjused, miks seda kasutatakse ning kus peaks kihilises arhitektuuris paiknema äriloogikakiht. Teises peatükis teeb autor ülevaate salvestatud protseduuridest ning aspektidest, mida tuleks jälgida, valides äriloogika asukohta. Kolmas peatükk on praktiline analüüs, kus autor võrdleb äriloogika rakendamist nii programmi koodis kui ka salvestatud protseduurides. Kolmanda peatüki lõpus analüüsib autor käsitletud lähenemiste eeliseid ja puuduseid ning annab omapoolsed soovitused.

# 2. Kihiline arhitektuur

Võrguprotokollid on ilmselt kõige tuntuim näide kihilisest struktuurist. Võrguprotokoll koosneb reeglite kogumist ning konventsioonidest, mis kirjeldavad, kuidas arvutiprogrammid suhtlevad kogu masina piires. Formaat, sisu ja tähendus on kindlaks määratletud kõikides sõnumites ning kõik stsenaariumid on tavaliselt üksikasjalikult jadadiagrammina kirjeldatud. Protokoll määratleb kokkulepped mitmetel abstraktsetel kihtidel, alates bit-de edastamise detailsusest kuni kõrgetasemelise rakendusloogikani. Seetõttu kasutavad disainerid mitmeid alamprotokolle ning seavad need kihtidesse. Iga kiht tegeleb konkreetse suhtlusaspektiga ning kasutab järgmise kihi teenuseid. Rahvusvaheline standardiorganisatsioon on määratlenud järgmise arhitektuurilise mudeli – OSI 7-kihiline mudel. (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996)




**Joonis 1. OSI 7-kihiline mudel**

Kihilise lähenemisviisi kasutamine on soovitatav praktika, vältida võiks protokolli täitmist monoliitplokina. Kihilise lähenemisviisi kasutamise kasuks räägib ka kontseptuaalselt erinevate probleemide rakendamise eelised – näiteks mitu meeskonda saab koos arendada,

mis toetab inkrementaalset arendust ja testimist. Kasutades pooleldi sõltumatuid osi, on hiljem võimalik üksikuid osi lihtsamini välja vahetada. Samuti on eeliseks võimalus uute tehnoloogiate (näiteks uute keelte või algoritmide) paremaks implementeerimiseks ümber kirjutada ainult väike osa koodist. (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996)

Üksikut kihti saab kirjeldada järgmise CRC kaardiga:

| Klass<br>Kiht J | Kaastööline:<br>• Kiht J - 1 |
|---|---|
| Vastutus<br>• Pakub teenuseid, mida kasutab kiht J + 1<br>• Delegeerib alamülesanded kihile J - 1 | |

**Joonis 2. Üksiku kihi CRC kaart**

Peamine struktuurne omadus kihilisel mustril on see, et kihi J teenuseid kasutab ainult kiht J + 1 – muid otseseid sõltuvusi kihtide vahel ei ole. Struktuuri võib võrrelda virna või isegi sibulaga – iga üksik kiht tõkestab alumiste kihtide otsese ligipääsu kõrgematesse kihtidesse. (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996).

Sellises skeemis kõrgemal asuv kiht kasutab alumise kihi poolt sätestatud erinevaid teenuseid, kuid alumine kiht ei ole teadlik ülemisest kihist. Lisaks sellele peidab iga kiht tavaliselt kõik temast allpooljäävad kihid ülemistest kihtidest, seega kiht 4 kasutab kiht 3 teenuseid, mis kasutab omakorda 2. kihi teenuseid, kuid kiht 4 ei ole teadlik kihist 2. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Näitena võib tuua TCP/IP protokolli, mida peetakse „interneti protokolli komplektiks“ ning vaatleme, milliseid olulisi eeliseid on süsteemi kihtideks lagundamisel:

- Üksikut kihti saab mõista ühtse tervikuna, teadmata väga palju teistest kihtidest. On võimalik mõista, kuidas ehitada FTP teenust TCP peale, teadmata detaile, kuidas Etherneti kiht töötab.

- Kihte saab asendada alternatiivsete realisatsioonidega, millel on samad põhiteenused.

- Kihtidevaheline sõltuvus väheneb. Kui telekommunikatsioonifirma vahetab oma füüsilisi edastussüsteeme, ei pea FTP teenusega mingeid muudatusi tegema.
- Kihid on hea koht standardiseerimiseks. TCP ja IP on standardid, sest nad defineerivad, kuidas nende kihid peavad töötama.
- Valmisehitatud kihi peale saab kastutusele võtta mitu kõrgemataseme teenust. Niiviisi kasutab TCP/IP-d nii FTP, telnet, SSH kui ka HTTP. Vastasel korral peaksid kõik kõrgema taseme protokollid ise omama madalama taseme protokolle.

(Fowler, Patterns of Enterprise Application Architecture, 2002)

Kihistumine on tähtis tehnika, kuid sellel on ka omad puudused:

- Kihilise arhitektuuri puhul tuleb vahel teha mitmetasandilisi muudatusi. Klassikaline näide kihilises rakenduses on see, et lisades andmebaasi uue välja, mida tuleks kuvada kasutajaliideses, tuleb see lisada ka kõikidesse vahepealsetesse kihtidesse.
- Lisakihid võivad mõjuda halvasti jõudlusele – tabaliselt teisendatakse igas kihis andmed ühest esitusest teise.

(Fowler, Patterns of Enterprise Application Architecture, 2002)

## 2.1 Kihilise arhitektuuri teke

Tarkvarainseneride poolt on tarkvarasüsteemi tükeldamiseks kõige levinum tehnika tarkvarakihtide kasutamine. Kihtide mõiste hakkas kerkima 1990. aastatel klient-server süsteemide levikuga. Siis oli tegemist kahekihilise süsteemiga – klient oli vastutav kasutajaliidese ja tarkvara koodi eest ning serveriks oli üldiselt relatsiooniline andmebaas. Levinud kliendivahenditeks olid Visual Basic, Powerbuilder ja Delphi, mis tegid väga lihtsaks andmemahukate rakenduste ehitamise, kuna neil olid kasutajaliidese vidinad, mis olid teadlikud SQL-st. Seega võis kasutajaliideseid ehitada lohistades kontrollereid kujundusalale ning kasutades omaduslehti, et ühendada kontrollerid andmebaasiga. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Kui rakendus koosneb ainult kasutajaliidesest ning lihtsast andmete uuendamise loogikast andmebaasis, siis piisab klient-server süsteemidest täielikult. Probleem tuleb esile aga

domeeniloogikaga – ärireeglid, valideerimised, kalkulatsioonid jne. Üldiselt kirjutati nende sisu klienti, kuid see oli ebamugav ning tähendas, et rakenduse loogika oli otse kasutajaliideses. Domeeniloogika keerulisemaks muutumisel muutus koodiga töötamine väga raskeks. Lisaks oli kood duplitseeritud, mis tähendas ka kõige lihtsamate muudatuse tegemiseks mitme kasutajaliidese lehe läbikäimist ja igal pool sama loogika muutmist. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Alternatiiv oli panna domeeniloogika andmebaasi protseduuridesse. Salvestatud protseduurid andsid aga väga piiratud struktureerimise võimaluse, mis tõi taaskord kaasa ebamugava koodi. Paljud spetsialistid kasutasid relatsioonilisi andmebaase SQL standardi tõttu. SQL standardi kasutamine pakkus võimalust andmebaasiplatvormi lihtsalt vahetada. (Fowler, Patterns of Enterprise Application Architecture, 2002)

3-kihilisele süsteemile üleminek oli lahendus domeeniloogika hoidmise probleemile. 3-kihilises süsteemis on presentatsioonikiht kasutajaliidese jaoks, domeenikiht äriloogika jaoks ning andmeliidestuskint. Sellisel juhul saab keeruka äriloogika liigutada kasutajaliidesest välja ning panna see kihti, kus on võimalik seda korralikult struktureerida koos objektidega. (Fowler, Patterns of Enterprise Application Architecture, 2002)

## 2.2 Kolmekihiline arhitektuur

3-kihiline arhitektuur on klient-server arhitektuur, kus kasutajaliides, ärireeglid ning andmetele ligipääs on arendatud ja hoitud eraldi moodulitena, tihti ka eraldi platvormidel. Põhimõtteliselt on kolm kihti:

1. Esitlus-/kasutajaliidese kiht
2. Äriobjektid ja –loogika
3. Andmeliidestuskiht

Nimetatuid kihte saab arendada ning testida eraldi. (Mahmoodi, 2005)

Loogikate kolme erinevasse kihti jagamisel ning esitluskihi eraldamisel äriloogika- ning andmeliidestuskihist on mitu eelist, näiteks:

- Äriloogikakomponentide korduvkasutus annab tulemuseks kiire arenduse. Kui ühes rakenduses on arendatud ja testitud moodul, mis tegeleb klientide lisamise, uuendamise, kustutamise ja leidmisega süsteemist, on seda võimalik kasutada ka mõnes muus projektis, mis tegeleb klientidega.

- Süsteemi ümberkujundus on lihtne. Kuna äriloogika on eraldatud andmeliidestuskihist, siis andmeliidestuskihi muutmine ei mõjuta äriloogika moodulit olulisel määral. Kui liigutakse näiteks Microsoft SQL andmebaasilt Oracle peale, siis ei tohiks see kaasa tuua mingeid muudatusi äriloogika- ega esitluskihis.

- Muudatuste haldamine on lihtne. Kui äriloogikas tuleb teha väike muudatus, ei pea igasse arvutisse uut rakenduse versiooni paigaldama. Kui maksude protsent muutub näiteks 10%-lt 15%-ni, siis on vaja muudatus teha ainult äriloogikas kasutajaid häirimata.

- Paindlikum ressursside jaotus. Võrguliiklust on võimalik vähendada, kui funktsionaalserverid eemaldavad üleliigsed andmed ning saadavad kliendile vaid nõutud struktureeritud andmed.

- Kiirem arendus – erinevate kihtide arendamisega tegelevad erinevad meeskonnad. Presentatsioonikihiga töötab veebidisainer, äriloogikakihiga tarkvaraarendaja ning andmemudeliga andmebaasiarendaja (Papagelis).

(Mahmoodi, 2005)

Kuigi koodi kirjutamine esitluskihti võib tunduda lihtsam, ei ole see kõige parem lähenemine. Kihilise lähenemise kasud tulevad välja ajapikku. Kui koodi kirjutatakse üha enam ja enam juurde, juhtub järgmine:

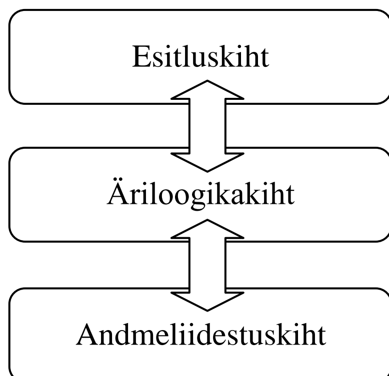
- Koodi kopeeritakse ja kleebitakse korduvalt või koodi kasutatakse klassides, mida võiks pigem liigutada äriloogikakihti.

- Kood, mis on väga sarnane, kopeeritakse ja kleebitakse väikeste erinevustega, mis teeb duplitseerimise raskemini jälgitavaks.

- Koodi on raskem hooldada – kuigi rakendused äriobjektidega on üldiselt suuremad rakendused, on need tihti paremini struktureeritud.

- Koodi on raske või võimatu ühiktestida.

Head arhitektuuri on alati raskem implementeerida, kuid kergem hallata, kuna hea arhitektuur vähendab tihti lõppkokkuvõttes koodi mahtu. (Mains, 2008)




**Joonis 3. 3-kihiline arhitektuur**

Joonis 3 kirjeldab äriloogikakihi ja andmeliidestuskihi eesmärki. Põhiline eelis 3-kihilisel arhitektuuril on andmeliidestus- ja esitluskihi eraldatus. Esitluskihis pole ühtegi funktsiooni, mis suhtleks otse andmebaasiga, vaid kõik vajalikud meetodid andmebaasiga suhtlemiseks on andmeliidestuskihis. See tähendab, et esitluskihis saab fokuseerida ainult informatsioonile, mis on vajalik kasutajale kuvamiseks. (Fakhar, 2010)

### 2.2.1 Esitluskiht

Esitluskiht hoolitseb kasutaja- ja tarkvaravahelise suhtluse eest. See võib olla kas käsurida, graafiline kasutajaliides vm. Esitluskihi peamised ülesanded on näidata informatsiooni kasutajale ning tõlgendada kasutajalt saadud käsud tegevusteks domeenikihile ja andmeallikale. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Järgmised komponendid on leitavad esitluskihist:

- Kasutajaliidese komponendid – need komponendid pakuvad mehhanisme, millega kasutaja saaks programmiga suhelda.

- Kasutajaliidese protsessi komponendid – aitavad sünkroniseerida ja juhtida kasutaja tegevusi. Takistavad protsessiloogikat ja olekuhaldusloogikat püsiprogrammeerimast kasutajaliidese elementidesse ning lubavad taaskasutada samu kasutaja käitumismustreid teistes kasutajaliidestes. (Meier, et al.)

### 2.2.2 Domeenikiht

Domeeni- ehk ärikihis on kirjeldatud kogu töö, mida rakendus peab tegema selle loomise valdkonnas. See hõlmab kõik arvutusi, mis tuleb teha sisendite ja salvestatud andmete pealt, samuti esitluskihist tulevate andmete valideerimist. Otsustada tuleb ka, millist andmeallika loogikat kasutada. See peaks tulenema esitluskihi käskudest. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Domeenikiht defineerib tööd, mida rakendus peab tegema ning juhib domeeniobjekte probleemide lahendusel. Ülesanded, millega see kiht tegeleb, on äri jaoks tähendusrikkad või vajalikud, et esitluskiht saaks suhelda andmeliidestuskihiga. Vastutav ärimõistete, -olukorra ning -reeglite esindamise eest. Seda kihti võib piltlikult vaadelda kui äritarkvara südant. (Evans, 2006)

Äriloogikakihi komponendid implementeerivad süsteemi põhifunktsionaalsused ning kapseldavad asjakohase äriloogika. Üldiselt on äriloogikakihis järgmised komponendid:

- Rakenduse fassaad – see on valikuline funktsionaalsus, mida saab kasutada mitme ärioperatsiooni kombineerimiseks ühtseks sõnumipõhiseks operatsiooniks. See funktsioon on kasulik, kui kasutajaliidese kihi komponendid on füüsiliselt eraldi masinas äriloogika kihi komponentidest. See annab võimaluse optimeerida kommunikatsioonimeetodit, mis neid ühendab.

- Ärikomponendid – need komponendid implementeerivad rakenduse äriloogika. Vaatamata sellele, kas äriloogika koosneb ühest sammust või on selle taga suur tööprotsess, on rakendusel vaja komponente, mis implementeerivad ärireegleid ja täidavad äriülesandeid.

- Äritöövoog – peale kasutaja käest andmete saamist kasutajaliidesest ning nende edastamist äriloogikakihile, saab rakendus neid andmeid kasutada äriprotsesside jaoks. Paljud äriprotsessid koosnevad mitmetest sammudest, mis tuleb täita kindla järjekorra alusel ning mis võivad omavahel suhelda vaid läbi kindla juhtimise. Äritöövoo komponendid defineerivad ja koordineerivad pikaajalisi ja mitmeetapilisi äriprotsesse ning neid saab implementeerida äriprotsesside haldamise vahenditega

- Äriolemuse komponendid – kasutatakse andmete edastamiseks komponentide vahel. Andmeteks on reaalse äritegevuse olemid nagu tooted või tellimused. Äriolemused,

mida rakendus kasutab sisemiselt on tavaliselt andmestruktuurid, näiteks XML, kuid neid saab implementeerida ka kasutades objekt-orienteeritud klasse, mis esindavad reaalmaailma olemeid, millega rakendus tegeleb.

(Meier, et al.)

### 2.2.3 Andmeliidestuskiht

Andmeliidestuskihi roll on suhelda erinevate infrastruktuuri osadega, mis on vajalikud rakendusele oma töö tegemiseks. Valdavalt on selleks suhtlemine andmebaasiga, milleks tihti on näiteks relatsiooniline andmebaas. Muster, mida kasutada selles kihis on kaugeleulatuv disainis ning teeb seega hiljem koodi refaktoreerimise raskeks, seega peab sellele valikule alguses väga suurt rõhku panema. See valik mõjutab tugevalt ka seda, kuidas hiljem domeenikihti disainida. Tark on eraldada andmebaasi ligipääs äriloogikakihist ja panna see eraldi klassidesse. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Hea viis, kuidas neid klassidesse organiseerida, on baseeruda andmebaasi tabelistruktuuridele, nii, et iga tabeli kohta on üks klass. Need klassid moodustavad värava tabelile. Ülejäänud rakendus ei pea teadma mitte midagi SQL'st ning kõik andmebaasiga suhtlevad SQL'd on lihtsalt ühest kohast leitavad. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Andmekihikomponendid võimaldavad ligipääsu andmetele nii antud rakenduse kui ka väliste süsteemide piires. Üldiselt on andmeliidestuskihis järgmised komponendid:

- Andmetele juurdepääsu komponendid – siia on kokku pandud kogu loogika, mis on vajalik andmekogumitega suhtlemiseks. Andmete ligipääsu tsentraliseerimisega on rakendus lihtsamini seadistatav ning hooldatavam.

- Andmete abistajad ja üldkasutatavad komponendid – enamik andmetega suhtlevad ülesanded kasutavad ühtset loogikat, mida saab eraldada ja rakendada eraldiseisvatesse korduvkasutatavatesse abikomponentidesse. See aitab andmete ligipääsukomponentide kompleksust vähendada ning tsentraliseerib loogika, mis omakorda lihtsustab hooldust. Ülejäänud ülesanded, mis on ühised kogu andmeliidestuskihi komponentidele ja mis pole ühegi komponendi spetsiifilised, saab implementeerida eraldi üldkasutatavate komponentidena. Mõlemad abi- ning üldkasutatavad komponente saab tihti korduvkasutada ka teistes rakendustes.

- Teenuseagendid – kui ärikomponent peab kasutama funktsionaalsust, mida pakub väline süsteem, tuleb implementeerida agent, mis haldab kogu suhtlussemantikat konkreetse välissüsteemiga. Teenuseagendid isoleerivad erinevate väliste teenuste eripära kogu rakendusest ning lisateenusena kaardistab objektid välissüsteemist tulnud formaadilt konkreetse rakenduse sees olevale formaadile.

(Meier, et al.)

## 2.3 Domeenikihi mustrid

### 2.3.1 Transaktsiooni skript

Enamik rakendusi töötab põhimõttel, et üks transaktsioon järgneb teisele. Ühe transaktsiooniga võidakse vaadata mingeid andmeid, teise transaktsiooniga neid muuta. Iga transaktsioon kliendi ja serveri vahel sisaldab mingit sorti loogikat. Mõnel juhul võib see olla lihtne informatsiooni näitamine andmebaasist, teisel juhul aga võib see sisaldada mitmeid valideerimis- ning arvutussamme. Transaktsiooni skripti muster organiseerib kogu selle loogika peamiselt ühte protseduuri, tehes ühendused andmebaasi otse või läbi „õhukese“ andmeliidestuskihi. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Transaktsiooni skriptiga on domeenikihi loogika organiseeritud põhiliselt transaktsioonide järgi, mida süsteem täidab. Kui näiteks üheks transaktsiooniks on hotellitoa broneerimine ja selleks tuleks kontrollida toa saadavust, arvutada toa hind ning uuendada andmebaasi uue broneeringuga, siis kõik need tegevused on leitavad ühes protseduuris. Taolise mustri üheks eeliseks on see, et ei ole vaja teada, mida teised transaktsioonid teevad. Selle mustri järgi saab transaktsioon oma sisendi, suhtleb andmebaasiga, muudab andmeid ning salvestab tegevuse. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Transaktsiooni skripti eelis peitub tema lihtsuses. Sellisel viisil domeenikihi ülesehitamine on väga loomulik rakendustele, kus on väga vähe äriloogikat – seda on lihtne mõista ning see ei tekita jõudlusprobleeme. Kui aga äriloogikat tuleb juurde, on seda väga raske hoida heas disainiolekus. Esimene probleem on duplitseerimine erinevates transaktsioonides. Kuna idee on hoida kõik transaktsioonid omaette protseduurides, siis ühine kood kipub duplitseerima. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Keerukamate äriloogikatega rakenduste puhul tuleks kasutada domeenimudeli mustrit, mis annab parema võimaluse koodi struktureerida, suurendades arusaadavust ning vähendades duplikatsiooni. (Fowler, Patterns of Enterprise Application Architecture, 2002)

### 2.3.2 Domeenimudel

Äriloogika võib olla rakendustes väga keeruline – reeglid ja loogikad, mis kirjeldavad palju erinevaid juhtumeid ja käitumisvõimalusi. Domeenimudel loob omavahel ühenduses olevate objektide võrgu, kus iga objekt esindab mõnd mõtestatud osa. Objektid presenteerivad äriandmeid või ärireeglite kogumikke, mida rakendada äriandmetele. Domeenimudeli mustri järgi näeb rakenduse domeenikiht välja väga sarnane andmebaasimudeliga. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Domeenimudelit tuleks kasutada siis, kui äriloogika on keeruline ning ärireeglid on pidevalt muutuvad. Kui ärireeglid ei muutu pidevalt, on mõistlikum kasutada transaktsiooni skripti mustrit. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Üks suurimaid erinevusi transaktsiooniskriptil ja domeenimudelil on päringu struktuuri muutmise mõju. Transaktsiooniskriptil tuleks selleks muuta kogu skripti. Kui on rohkem domeeniloogikaid, mis kasutavad samu andmeid, tuleks neid kõiki eraldi muuta. Domeenimudeliga tuleb aga muuta vaid üht eraldatud koodisektsiooni ning domeeniloogika ise jääb muutumatuks. See on transaktsiooniskripti ja domeenimudeli suurim vahe – esialgne keerukus domeeniloogika juures tasub ära, kui koodis on palju äriloogikat. (Fowler, Domain Logic and SQL, 2003)

### 2.3.3 Teenuste kiht

Tarkvararakendustel on üldiselt mitmeid erinevaid liideseid aplikatsiooni andmetele ja loogikale ligipääsuks: andmetelaadurid, kasutajaliidesed jne. Vaatamata nende erinevatele eesmärkidele vajavad nad rakenduse poolt tihti sarnast käitumist, et ligi pääseda andmetele ning et rakendada äriloogikat. Need tegevused võivad olla väga keerulised, hõlmates transaktsioone üle mitme allika ning reegleid saadud vastustega edasi tegelemiseks. Selliste loogikate eraldi hoidmine iga liidese jaoks tekitab koodi duplitseerimist. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Teenuste kiht defineerib rakenduse piirid ning selle saadaolevad operatsioonid. See kapseldab aplikatsiooni äriloogika, kontrollib transaktsioone ning koordineerib allikatelt saadud vastuste edasise käitumise. (Fowler, Patterns of Enterprise Application Architecture, 2002)

Teenuste kihti saab rakendada kahel moel. Esiteks operatsioonikihi lähenemisega rakendatakse teenuste kihti kui „paksu“ kihti, mis ise implementeerib rakenduse loogika, kuid delegeerib äriloogika kapseldatud äriloogika objektidele. Teine lähenemisviis on kui domeenifassaad, millega rakendatakse teenuste kihti kui õhukest fassaadi, mille taga on domeenimudel. Sellise fassaadi klassid ei implementeeri ise äriloogikat, vaid seab toimingute piirid, läbi mille liidesed suhtlevad rakendusega. (Fowler, Patterns of Enterprise Application Architecture, 2002)

### 2.3.4 Tabeli moodulid

Traditsioonilises objekt-orienteeritud lähenemises tekitatakse iga andmebaasirea kohta omaette objekt (domeenimudeli muster). Tabel mooduli puhul on sarnaselt domeenimudelile iga andmebaasi tabeli kohta rakenduses defineeritud klass, kuid kui domeenimudeli puhul on iga tabeli rea kohta eraldi objekt, siis tabeli mooduli puhul on ühes objektis konkreetse elemendi andmekogum, olles sarnane andmebaasipäringu tulemusega. Seega ei ole objektil ka kindlat identiteeti tabeli reaga, vaid konkreetset rida tuleb objektilt üldiselt pärida kasutades andmebaasitabeli primaarvõtme veergu. (Fowler, Patterns of Enterprise Application Architecture, 2002)

# 3. Äriloogika – rakenduses või protseduuris

Paljud autorid rõhutavad rakenduse loogika lahtilõhkumist kihtideks. Kuigi erinevad autorid kasutavad erinevaid kihte, on üheks ühiseks teemaks äriloogika ja andmeliidestuse eraldamine erinevatesse kihtidesse. Kuna paljud rakendused hoiavad andmeid relatsioonilistes baasides, siis kihistumisega üritatakse eraldada äriloogikat relatsioonilisest andmebaasist. (Fowler, Domain Logic and SQL, 2003)

Paljud arendajad kalduvad relatsioonilisi andmebaase käsitlema kui salvestusmehhanisme, mis peaks olema täielikult peidetud. On olemas palju raamistikke, mille eeliseks on keeruka SQL-i varjamine rakenduse arendaja eest. Siiski on SQL enamat, kui andmete uuendamise ja kättesaamise mehhanism. SQL-i päringutöötlemine suudab töödelda palju ülesandeid ning varjates SQL-i, on arendajate eest peidetud võimas vahend. (Fowler, Domain Logic and SQL, 2003)

Relatsioonilised andmebaasid toetavad kõik standardset päringu keelt – SQL'i. Arvatavasti on see ka põhiline põhjus, miks relatsioonilised andmebaasid on nii populaarsed. Standardne viis andmebaasidega suhtlemiseks on andnud tugeva võimaluse olla kindlast andmebaasipakkujast sõltumatu. See on omakorda aidanud turule tulla mitmel erineval relatsioonilisel andmebaasil ning lisaks aidanud vähendada objektorienteeritud väljakutseid. (Fowler, Domain Logic and SQL, 2003)

SQL'l on mitmeid tugevusi, kuid eriti tõuseb esile väga võimas võimekus teha andmebaasi päringuid, lastes klientidel filtreerida ning summeerida suuri andmehulki väga väheste SQL'i koodiridadega. Kasutades võimsaid SQL päringuid, minnakse vastuollu kihilise rakenduse arhitektuuri aluspõhimõtetega, mis omakorda pärsib äriloogikakihti. (Fowler, Domain Logic and SQL, 2003)

## 3.1 Salvestatud protseduurid

Salvestatud protseduurid on populaarsed ning väga laialdaselt kasutatavad, sest lubavad arendajatel andmetöötlust teha otse andmebaasiserveris, mitte tekstisiseste SQL'dega. Järgnevalt on välja toodud põhjused, miks kasutada salvestatud protseduure:

- Hooldatavus – kuna skriptid on kõik ühes kohas on uuenduste ja sõltuvuste jälgimine andmebaasiskeemide muutmisega seoses muutunud lihtsamaks.

- Testimine – kuna salvestatud protseduurid on ühes kohas, ei teki segadust, et ärireeglid on laiali erinevates rakenduse klassides.

- Kiirus/optimeerimine – salvestatud protseduurid on serveris puhverdatud. Protseduuride täitmise kavad on lihtsalt ülevaadatavad ilma rakendust käivitamata.

- Andmekogumikepõhine andmete töötlemine – SQL'i tugevuseks on võimekus töödelda kiirelt ja efektiivselt suurt hulka andmekogumikku korraga. Tarkvarakoodis on samaväärne iteratiivne tsükkel ehk reapõhine andmete töötlemine.

- Turvalisus – ligipääsu tabelitele saab piirata läbi andmebaasirollide. Andmetele saab ehitada liidesed andmete lugemiseks, millega andmestruktuur ning andmed ise on muutmise osas kaitstud. Andmete ja sellele ligipääsetavate protseduurikoodide turvalisust on lihtsam rakendada, kui teha seda üle rakenduse koodi.

(Hambrick, 2013)

Salvestatud protseduuridel on ka puudusi, mis takistavad neid võtmast kasutusele kui ühtset lahendust rakenduse andmebaasiga suhtlemiseks. Järgmisena on toodud nimekiri põhjustest, miks salvestatud protseduurid ei pruugi olla parim lahendus rakenduse jaoks:

- Piiratud koodifunktsionaalsus – salvestatud protseduuride kood ei ole nii jõuline nagu rakenduse kood, eriti tsüklites (näiteks kursorid on väga aeglased ning kasutavad intensiivselt protsessorit).

- Teisaldatavus – keerulist protseduuri koodi ei ole võimalik üle tõsta ühest andmebaasi platvormilt teisele.

- Ärireeglite asukoht – kuna salvestatud protseduurid ei ole nii lihtsasti grupeeritavad/kapseldatavad, tähendab see seda, et ärireeglid on laiali erinevates salvestatud protseduurides. Rakenduse koodiarhitektuur aitab aga tagada seda, et ärireeglid on kapseldatud ühte objekti.

- Andmekogumikepõhise töötluse kasutamine – liiga palju lisakulu tekib sellega, kui ka mittekeerulised andmepäringud tehakse läbi salvestatud protseduuride. Selle tulemusena ei tohiks iga päringu kohta teha oma protseduuri, vaid hoida lihtsad andmeteküsimise päringud rakendusesiseste SQL'dena.

- Kulu – olenevalt firma olemusest võib juhtuda, et salvestatud protseduuride kirjutamiseks oleks vaja palgata eraldi andmebaasiarendaja. Osad ärid ei luba arendajatele ligipääsu andmebaasidele, vaid nõuavad, et seda rolli täidaksid andmebaasiadministraatorid, mis tõstab taas arenduse hinda.

Mõned põhjused kattuvad eelpool toodud nimekirjaga, miks kasutada salvestatud protseduure, kuna antud põhjused võivad erisuguste arendajate lõikes olla kas puuduseks või eeliseks.:(Hambrick, 2013)

Arendajate kõige levinumad argumendid salvestatud protseduuride kasutamiseks:

- Jõudlus – päringu plaan on andmebaasiserveris kompileeritud, nii, et päringud jooksevad kiiremini. Samuti suudab üks protseduur täita mitu SQL käsku, mis vähendab rakenduse ja andmebaasiserverivahelist andmete liiklust.

- Turvalisus – protseduuridega saab hästi kaitsta andmetele otse ligipääsu, lisaks kaitsevad parameetrid SQL rünnakute eest.

- Koodi taaskasutus – andmebaasipäringud saab korra kirjeldada ning taaskasutada mitmeid kordi.

- Äriloogika kapseldamine – äriloogika hoitakse kõik ühes kohas ehk andmebaasiprotseduurides.

(Lawry, 2012)

## 3.2 ORM

ORM on koodil põhinev rakendus, mis aitab arendajatel töötada andmebaasidega. ORM-i eesmärk on luua andmemudeli koodiline esitlus, kui see on paigas, ei pea andmetele ligipääsuks kirjutama ridagi SQL'i. ORM-id on väga efektiivsed CRUD operatsioonidega. Tavaliselt on 90% rakenduses kasutatavatest SQL käskudest just CRUD päringud. ORM

kirjutab selle tarbeks parametriseeritud päringuid, nii, et arendaja ei peaks selle peale aega kulutama. Seega muutub ka salvestatud protseduuride kirjutamine lihtsateks select või update päringuks ajaraiskamiseks. (Lawry, 2012)

Põhjused, miks kasutada ORM'i:

- Vähem käsitsi kirjutatud koodi – ORM'i kasutades kulub kõvasti vähem aega, kui sellele, et valmis kirjutada salvestatud protseduurid ning need rakendustest välja kutsuda.

- Kiirem arendus – arendajad peavad tihti arendama rohkem funktsionaalsust vähema ajaga. ORM'i vahendid võimaldavad koodigenereerimist lihtsa regenereerimisega. Kui andmemudel all muutub, aitab ORM need tuvastada ja parandada automaatselt, salvestatud protseduurde tuleb aga käsitsi muuta igal pool, kuhu see muudatus mõjub.

- Hooldatavus ja rektooring – agiilse arenduse põhiliseks eelduseks on see, et rakenduse loogika muutub tulevikus. Andmemudel ja äriloogika muutub. ORM'i kasutades saab need muudatused kiirelt ja lihtsalt ära tehtud. Üldiselt piisab koodi kompileerimisest, et aru saada, kas kõik muudatused said tehtud. Salvestatud protseduure kasutades on oht üht kohta parandades teha katki loogikad muudes kohtades.

- Jõudlus – ORM'd muudavad ja kustutavad andmeid andmebaasis mitte iga muudatuse peale koodis, vaid siis, kui selleks antakse käsk koodist. Paljud ORM'd pakuvad lisaks ka täiustusi jõudluse parendamiseks, kasutades näiteks vahemälu.

- Korrektne kihiline arhitektuur – andmebaas peaks olema lihtsalt „püsivuse kiht". See ei tohiks sisaldada rakenduse äriloogikat. ORM'i kasutades paikneb äriloogika hästi hallatavas koodis, mitte ei ole hajutatud erinevate salvestatud protseduuride ja trigerite vahele. Nii on tagatud ka korrektne kihiline arhitektuur.

- Äriloogika kapseldus – andmeüksused, mis on ORM'i poolt genereeritud, on äriobjektid. Äriobjektidele saab koodis kirjutada juurde ärireeglid, mida kasutavad ära ka esitluskiht ja andmeliidestuskiht.

(Lawry, 2012)

## 3.3 Äriloogika asukoha valimise aspektid

Hoides äriloogikat kas või osaliselt salvestatud protseduurides, tuleb arvestada mitme erisuguse aspektiga:

- Jõudlus
- Testitavus
- Modifitseeritavus
- Arusaadavus
- Duplitseerimise vältimine
- Kapseldamine
- Andmebaasi teisaldatavus

(Fowler, Domain Logic and SQL, 2003)

### 3.3.1 Jõudlus

Kuigi spetsialistid kipuvad esimese asjana äriloogika paigutamisel mõtlema jõudluse peale, siis ei peaks see olema mitte üldse esimene küsimus. Alustuseks peaks hoopiski mõtlema, kuidas kirjutada hallatavat koodi. Alles peale seda tuleks identifitseerida kitsaskohad ning asendada need kiirema, kuid mitte nii selge koodiga. Põhiline põhjus on selles, et enamikes süsteemides on ainult väike osa koodist jõudlus-kriitiline ning jõudlust on palju lihtsam parendada hästi struktureeritud ning hallatav koodis. (Fowler, Domain Logic and SQL, 2003)

Jõudlus ei ole ainus faktor, millele tugineda äriloogika paigutamise otsustamisel, kuid tihti on see otsustav. Kui rakenduses on mõni kitsaskoht, mida kindlasti tuleb jõudluse mõttes parandada, siis teised faktorid jäävad tahaplaanile. (Fowler, Domain Logic and SQL, 2003)

### 3.3.2 Testitavus

Disainiteemadest rääkides ei ole testitavus piisavalt esindatud aspekt. Ometigi räägib testipõhise arenduse kasulikkusest taaselustatud arusaam, et testitavus on oluline osa disainist. Levinud praktika kohaselt SQL'i koodi ei testita, kuid on mitmeid vahendeid, mida saab kasutada andmebaasikeskkondade testimiseks. (Fowler, Domain Logic and SQL, 2003)

### 3.3.3 Modifitseeritavus

Iga pikaealise rakendusega saab kindel olla ühes asjas – see muutub palju. Selle tulemusena tuleb tagada, et süsteem oleks üles ehitatud nii, et seda oleks kerge muuta. Modifitseeritavus on arvatavasti põhiline põhjus, miks tahetakse äriloogikat hoida rakenduses. (Fowler, Domain Logic and SQL, 2003)

### 3.3.4 Arusaadavus

SQL'i nähakse tihti kui erilist keelt – miski, millega tarkvaraarendajad ei peaks tegelema. Paljud raamistikud reklaamivad end seeläbi, et neid kasutades on võimalik vältida SQL'ga tegelemist. Paljud arendajad leiavad, et SQL'ga on raskem tegeleda, kui traditsiooniliste arenduskeeltega ning et paljusid SQL idioome mõistavad ainult SQL'i eksperdid. (Fowler, Domain Logic and SQL, 2003)

### 3.3.5 Duplitseerimise vältimine

Üks lihtsamaid, kuid samas tugevamaid disainipõhimõtteid seisneb duplitseerimise vältimises, mis on formuleeritud ka kui *DRY* (eesti k. ära korda ennast) põhimõte. (Fowler, Domain Logic and SQL, 2003)

### 3.3.6 Kapseldamine

Kapseldamine on üks tuntud objektorienteeritud disaini põhimõte ning see kehtib üldiselt rakenduste disainile. Sisuliselt ütleb see, et programm tuleb jagada mooduliteks, mis peidavad andmestruktuurid liideste ja protseduuride taha. Selle põhimõtte eesmärgiks on võimaldada muuta alusandmete struktuuri, põhjustamata suurt „lainetuse" efekti kogu süsteemis. (Fowler, Domain Logic and SQL, 2003)

Rakendustes saab kapseldamist teha kihtidega, kus püütakse eraldada domeenikihi loogikat andmeallika loogikast. Sellisel juhul ei mõjuta andmebaasi disaini muudatused koodi, mis töötab äriloogikaga. (Fowler, Domain Logic and SQL, 2003)

Rakendustes saavutatakse kapseldus läbi *API*-de – SQL's vastaks sellele vaadete tegemine. Tabelit muutes saab selle peale teha vaate, mis toetab vana tabeli struktuuri. Suurim probleem on andmete uuendamisega, mida tihti ei ole võimalik teha läbi vaadete. Sellepärast kasutatakse tihti andmemanipulatsiooni mähkimist salvestatud protseduuridesse. (Fowler, Domain Logic and SQL, 2003)

Andmebaasivaadete või salvestatud protseduuride kasutamine pakub kapseldust vaid teatud piirini. Paljudes rakendustes tulevad andmed mitmest allikast, seejuures mitte ainult relatsioonilistest andmebaasidest, vaid ka vanadest süsteemidest, teistest rakendustest ning failidest. Sellisel juhul saab täieliku kapselduse tagada vaid kiht rakenduse koodis, mis omakorda tähendab, et domeeniloogika peaks olema mälus. (Fowler, Domain Logic and SQL, 2003)

### 3.3.7 Andmebaasi teisaldatavus

Üks põhjus, miks arendajad ei söanda kasutada keerulisemat SQL'i on andmebaasi teisaldatavuse probleem. Vaatamata sellele, et SQL peaks olema standardne keel, mida saab kasutada erinevatel andmebaasiplatvormidel, ei ole see reaalsuses päris nii. Praktikas on SQL enamasti standardne, kuid paljude väikeste mööndustega. SQL'i, mida saaks kergesti kasutada mitmel erineval andmebaasiplatvormil, tuleks ehitada suure ettevaatlikkusega. Seda tehes kaotatakse aga samal ajal paljudes SQL'i võimalustes. (Fowler, Domain Logic and SQL, 2003)

Otsus andmebaasi teisaldatavuse vajalikkuse kohta on üldiselt projektipõhine. Tänapäeval ei ole see enam nii suur probleem kui vanasti. Suured korporatsioonid kasutavad tihti kindlaid andmebaasiplatvorme ning nende vahetamine on üsna kulukas tegevus ja seega väga ebatõenäoline. Sellisel juhul tasuks ära kasutada kõiki eeliseid, mida projektis kasutusel olev andmebaasiplatvorm pakub. (Fowler, Domain Logic and SQL, 2003)

# 4. Võrdlus

Antud peatükis võrdleb autor eelmises peatükis välja toodud aspekte, mille järgi otsustada, kas äriloogikat paigutada andmebaasiprotseduuridesse või mitte. Võrdluse jaoks viib autor läbi testid näiterakenduse peal. Võrreldakse jõudluse, testitavuse, modifitseeritavuse, duplitseerimise vältimise ja kapseldamise aspekti.

Autor ei võrdle arusaadavuse aspekti, kuna leiab, et see on väga personaalne, olenedes palju kogemusest ja igapäevasest kokkupuutest antud valdkonnaga. Lisaks ei testi autor andmebaasi teisaldatavuse aspekti, kuna protseduuriline keel on igal andmebaasiplatvormil küllalt erinev ning seega tuleks protseduurid kindlasti ümber kirjutada, kui äriloogika on hoitud protseduurides.

Võrdlus on autori töö ning peatüki lõpus teeb autor üldistatud analüüsi, mis hõlmab kõiki testitud aspekte. Lisaks annab autor soovitusi, millistel juhtudel tuleks äriloogika paigutada andmebaasiprotseduuridesse ning millistel juhtudel mitte.

Testide läbiviimiseks on tehtud veebirakendus, kasutades Javat ja *Spring* raamistikku, rakendust käivitatakse *Spring Boot*-i abil *Jetty* konteineris. Näiterakenduses on kasutusele võetud PostgreSQL 9.3 andmebaas ning andmebaasiga suhtlemiseks kasutatakse *Hibernate ORM*-i ja *JPA*-d.

PostgreSQL on võimekas avatud lähtekoodiga objekt-relatsiooniline andmebaasisüsteem, mida on arendatud rohkem kui 15 aastat ning millel on tõestatud arhitektuur.. See tagab usaldusväärsuse, andmete terviklikkuse ning õigsuse. PostgreSQL töötab kõikidel peamistel operatsioonisüsteemidel, sealhulgas Linux-l, UNIX-l ja Windows-l. See vastab täielikult ACID nõuetele ning toetab välisvõtmeid, sidumisi, vaateid, trigereid ja salvestatud protseduure (mitmes keeles). See sisaldab enamusi SQL:2008 andmetüüpe ning toetab suurte binaarsete objektide hoidmist, nagu näiteks pildid, helid või videod. (About PostgreSQL)

Näidisrakendus on 10-st andmebaasitabelist koosnev portfellihaldusrakendus, mis võimaldab portfelli väärtust arvutada erinevatel meetoditel (keskmine, FIFO ja LIFO). Antud arvutusmeetodid on rakenduse äriloogika ning need arvutused on realiseeritud nii Javas kui ka andmebaasiprotseduurides. Andmebaasitabelite struktuur on välja toodud lisas 1.

Rakenduse loogika on järgmine: kasutajal on investeerimisportfellid, mis koosnevad väärtpaberikontost ja sellega seotud arveldusarve kontost. Väärtpaberikontoga on seotud väärtpaberid, millega on seotud väärtpaberitehingud. Väärtpaberitehingutega on seotud ka rahatehingud, mis on omakorda seotud arveldusarve kontoga. Kõik rahatehingud ei ole ilmtingimata seotud väärtpaberitehingutega (näiteks dividendid, kontohaldustasud jne). Rakendusse on esialgu implementeeritud keskmise ja FIFO meetodil portfelli/väärtpaberitootluse arvutamine.

## 4.1 Jõudlus

Jõudlustestide läbiviimiseks on kasutatud kahte meetodit. Esimeses meetodis mõõdetakse Java koodis lihtsustatult, palju kulub aega meetodi väljakutsumiseks – katset korratakse 10 korda, et saada keskmine aeg. Teises meetodis kasutatakse Apache JMeter-t jõudlustestide läbiviimiseks. Testimiseks on kasutusele võetud kahte sorti arvutused – esiteks portfelli(de) tootlus(t)e arvutus FIFO ja keskmisel meetodil, mille tulemus jääb rakendusse, ning teiseks portfelli(de) turuväärtuse arvutus ühe aasta lõikes, mille tulemused salvestatakse andmebaasi tabelisse. Samu teste viiakse läbi nii andmebaasis kui rakenduses.

### 4.1.1 Tootluse arvutuse test

Tootluse arvutuse testiks on autor teinud rakendusse klassid FifoStrategy ja AverageStrategy ning andmebaasi vastavad protseduuri. Java kood ja protseduurid tootluse arvutuseks asuvad lisades 2 – 5. Esimeses katses arvutati ühe portfelli tootlus, millega on seotud 7 väärtpaberit, 60 väärtpaberi- ja 6 rahatehingut.

**Tabel 1. Tootluse arvutuse jõudlustesti tulemused**

| Meetod | Portfelle | Tehinguid | Java FIFO | Andmebaas FIFO | Java Keskmine | Andmebaas Keskmine |
|---|---|---|---|---|---|---|
| Rakendus | 1 | 66 | 217ms | 136,7ms | 204,1ms | 32,9ms |
| Rakendus | 101 | 1166 | 1 208,2ms | 4 544,2ms | 1 206,4ms | 860,9ms |
| JMeter | 1 | 66 | 262ms | 184ms | 259ms | 98ms |
| JMeter | 101 | 1166 | 2 157ms | 11 749ms | 2 167ms | 1 911ms |

Testi tulemusena võttis Java koodis FIFO meetodil portfelli väärtuse arvutamine keskmiselt 217ms, samal meetodil andmebaasis aga 136,7ms. Andmebaasi arvutus oli keskmiselt 37% kiirem. Keskmise meetodi tulemuste vahe oli aga veel suurem – Javas võttis arvutus keskmiselt aega 204,1ms, andmebaasiprotseduuri kasutades aga hoopis 32,9ms. Andmebaasiprotseduuriga arvutus oli keskmiselt 83,88% kiirem.

Keskmise arvutuse loogika on FIFO meetodist lihtsam ja seega võtab see ka vähem aega. Keskmise meetodit kasutades käiakse väärtpaberitega seotud tehingud ükshaaval tehingu tegemise järjekorras läbi ning arvutatakse tulemus. FIFO meetodit kasutades käiakse tsüklit kasutades läbi kõik väärtpaberitega seotud tehingud nagu ka keskmise meetodil, kuid lisaks iga müügitehingu puhul käiakse läbi ka kõik varasemad ostutehingud, mis tähendab, et tsükleid tuleb ühe korraga palju enam. Antud näite 60st väärtpaberitehingust müügitehinguid oli 24.

Teises katses kasutati sama, aja mõõtmise, meetodit, kuid tootlust arvutati 101-le portfellile, millega oli seotud 411 väärtpaberit ning 1100 väärtpaberi- ja 66 rahatehingut. Kuna arvutus käib väärtpaberipõhiselt, siis võib öelda, et arvutusmaht on ca 58 korda suurem.

Testi tulemusena võtab rakenduses FIFO meetodil kõikide portfellide tootluse arvutus keskmiselt 1208,2ms ja keskmise meetodil 1206,4ms ehk üsna võrdselt. Andmebaasiprotseduuriga arvutades võttis aga FIFO meetodil arvutus keskmiselt 4544,2ms, mis on ca 376% aeglasem, kui rakenduses arvutades. Keskmise meetodiga arvutusel oli protseduuri kasutamine ikka kiirem, võttes aega keskmiselt 860,90ms ja olles seega ca 28% kiirem.

Kui ühe portfelli ja 7 väärtpaberiga tootluse arvutus oli andmebaasiprotseduuridega märgatavalt kiirem, siis väärtpaberikoguse kasvuga kahanes ka kiirus. Mahu ca 60-kordselt kasvul suurenes arvutusaeg rakendusesiseselt umbes 6 korda, andmebaasiprotseduuride puhul aga 26 (keskmise meetodi puhul) ja 33 (FIFO meetodi puhul) korda. Üheks faktoriks on kindlasti see, et iga väärtpaberi kohta tehakse uus ühendus andmebaasiga ning kutsutakse seal välja protseduur. Samas jäi keskmise arvutus ka mahu suurenedes rakendusesisese arvutusega võrreldes kiiremaks, kuid keerulisema loogikaga FIFO arvutus muutus aga hoopis aeglasemaks.

Apache JMeter on avatud lähtekoodiga tarkvara, mis võimaldab läbi viia jõudlusteste. Sellega saab simuleerida suurt koormust serverile, et analüüsida tulemuslikkust erinevat tüüpi koormuste all. (Apache JMeter)

Apache JMeter-ga sai tekitatud koormus, kus 100 samaaegset kasutajat teevad veebiserveri pihta 2 päringut 2 sekundilise vahega. Päringu tulemusena käivitab rakendus tootluse käivituse sarnaselt esimesel meetodis tehtud katsetustele.

Teise katse esimene meetod näitab taas, et ühe portfelli arvutuses on siiski andmebaasiga arvutus kiirem. FIFO meetodi puhul käis arvutus rakenduses keskmiselt 262ms, andmebaasis 184ms, keskmise meetodi puhul vastavalt 259ms ja 98ms. Mitme portfelli tootluse arvutus on ootuspäraselt aeglasem: FIFO arvutus käis keskmiselt 11749ms, keskmise arvutus 1911ms.

Koormuse all muutusid rakenduse arvutused 1 portfelli puhul ca 1,2 – 1,3 korda aeglasemaks, 101 portfelli puhul ca 1,8 korda aeglasemaks. Andmebaasiprotseduuridega 1 portfelli arvutus läks FIFO puhul 1,35 korda aeglasemaks, ja keskmise arvutus ligi 3 korda aeglasemaks. Kuna see arvutus oli nii 1 kui ka 101 portfelliga kõige kiirem, siis on alust arvata, et suurem osa ajakaotusest tuleb arvata rakenduse muude osade arvele (lehe kuvamine, serveripoolne töö jm). 101. portfelliga läks andmebaasiprotseduuridega arvutus aeglasemaks ca 2,2 – 2,5 korda.

### 4.1.2 Turuväärtuse arvutuse test

Tootluse arvutuse testis küsiti peale andmebaasis andmete väljaarvutamist need tagasi rakendusse, turuväärtuse arvutuse testis aga arvutatakse suurel hulgal andmeid välja ning salvestatakse need andmebaasitabelisse. See tähendab, et Java koodi puhul loetakse esialgu kõik portfellid, kontod, väärtpaberid ja nendega seotud tehingud rakendusse, tehakse arvutus ning seejärel salvestatakse tulemus andmebaasi tagasi. Andmebaasiprotseduuride puhul küsitakse andmebaasist ainult portfellide nimekiri ning seejärel kutsutakse iga portfelli kohta välja andmebaasiprotseduur, massilise arvutuse puhul käivitatakse vaid andmebaasiprotseduur, mis arvutab kõikidele portfellidele korraga tootlust..

Testid on üles ehitatud koormuselt sarnaselt tootluse arvutuse testile nii, et esimesel korral arvutatakse andmeid ühele portfellile ning teisel korral 101-le portfellile, seda nii rakenduses kui protseduuris. Turuväärtust arvutatakse testides ühe aasta kohta, mis tähendab iga päeva kohta iga portfelli iga väärtpaberikoguse väljaarvutust. Java koodis on ühe portfelli jaoks turuväärtuse arvutamiseks vaja kolme tsüklit – iga päeva kohta aastas, iga konto kohta

portfellis ja iga väärtpaberi kohta kontol. Lisaks on vaja iga päeva kohta läbi käia kõik väärtpaberitehingud, et saada teada väärtpaberi kogus antud päeva kohta. Kuna hind on iga päeva kohta eraldi andmebaasis välja arvutatud, ei hakata seda eraldi otsima tsükliga, vaid konkreetse päeva väärtpaberi hind päritakse andmebaasist.

```
public void calculatePortfolioMarketValuesForPeriod(Portfolio portfolio, LocalDate start, LocalDate end) {
    double price;
    for (LocalDate currentDate = start; currentDate.isBefore(end.plusDays(1)); currentDate = currentDate.plusDays(1)) {
        double value = 0;
        for (Account account : portfolio.getSecurityAccounts()) {
            for (Asset asset : account.getAssets()) {
                price = equityPriceService.getAssetPriceOnDate(asset.getEquity(), currentDate);
                if (price == 0) {
                    try {
                        price = assetPriceMap.get(asset);
                    } catch (Exception e) {
                        price = 0;
                    }
                } else {
                    assetPriceMap.put(asset, price);
                }
                value += asset.getQuantityOnDate(currentDate) * price;
            }
        }
        portfolio.addMarketValue(new MarketValue(portfolio, currentDate, value, "EUR"));
    }
    marketValueRepository.save(portfolio.getMarketValues());
}
```

**Joonis 4. Turuväärtuse arvutus Javas**

Andmebaasiprotseduure tegi autor kokku neli – nii ühe kui kõikide portfellide arvutamiseks kaks protseduuri sarnaselt Java koodile tsüklitega, ülejäänud kaks ilma tsükliteta, kasutades ära SQL-i võimekust töötada mitte reakaupa, vaid andmekogumite kaupa. Tootluse arvutuse protseduurid kõikide portfellide jaoks on välja toodud lisades 6 ja 7.

**Tabel 2. Turuväärtuse arvutuse jõudlustesti tulemused**

| Portfelle | Java | Andmebaas | Anembaas Tsükkel |
|---|---|---|---|
| 1 | 907ms | 2 337ms | 1 323ms |
| 101 | 42 159ms | 3 169ms | 208 904ms |

Teises testis tuleb väga hästi välja andmebaasi võimekus suurte andmemahtude juures. Ühe portfelli turuväärtuse arvutus aasta kohta oli kõige kiirem Javas, võttes 907ms, veidi aeglasem oli andmebaasiprotseduur, mis jäljendas Java loogikat ning kõige aeglasem, Java arvutusest

2,5 korda aeglasem, oli protseduur, kus tsükleid ei kasutatud, vaid arvutati turuväärtused iga päeva kohta korraga välja.

101 portfelliga oli tulemus aga hoopis teistsugune. Kui Java arvutus läks andmemahu kasvades aeglasemaks ca 47 korda, siis andmebaasiprotseduur, mis arvutas sarnase loogika alusel, läks aeglasemaks ca 158 korda. Mõlemale tuli juurde üks tsükkel, mis käis läbi ka kõik portfellid ükshaaval. Väga head kiiruse saavutas protseduuriga, mis tsükleid ei kasutanud. Antud protseduur võttis võrreldes 1 portfelli turuväärtuse arvutusega aega vaid 1,3 korda kauem.

Testi tulemusena on näha, et tsüklitega andmetöötluses sõltub aeg tugevalt andmemahust. Kuna Java töötleb andmeid ükshaaval, mistõttu tuleb kasutada tsükleid, siis SQL-s on palju võimalusi, millega tsükleid vältida. Seetõttu on suurte andmemahtude arvutuste puhul kindlasti soovitatav võimalusel eelistada arvutusi andmebaasis. Väiksemate andmemahtude ning lihtsama loogikaga ei tule andmebaasi jõudluse eelis esile – paljudes näidetes oli rakendusesisene arvutus isegi kiirem.

## 4.2 Testitavus

Antud rakendusele tegi autor äriloogika jaoks ka ühiktestid, mis on tarkvaraarenduses arvutiprogrammi väikseimate osade testid. Ühiktestid aitavad probleeme varem avastada, näiteks peale muudatusi koodis on võimalik teada saada, kas loogika jäi terveks või on rakendusse tekkinud vead. Testitavateks ühikuteks olid FIFO ja keskmisel meetodil väärtpaberi tootluse arvutamise loogika.

Java rakenduses sai mugavalt kasutada JUnit raamistikku, mis on väga kasutajasõbralik. JUnit teste saab hõlpsasti käivitada kasutatavas arendusrakenduses või rakenduse ehitamisel.

```java
@Test
public void fifoStrategyTest() {
    Asset asset = getAsset();
    asset.setCalculationStrategy(new FifoStrategy());
    asset.calculatePerformance();

    assertEquals(1000.0, asset.getQuantity());
    assertEquals(1100.0, asset.getBookCost());
    assertEquals(1.1, asset.getBuyPrice());
    assertEquals(750.0, asset.getRealizedProfitLoss());
    assertEquals(900.0, asset.getUnRealizedProfitLoss());
    assertEquals(2000.0, asset.getMarketValue());
    assertEquals(1650.0, asset.getProfitLoss());
}
```

**Joonis 5. Ühiktestid Javas**

Andmebaaside puhul ei ole ühiktestimine väga levinud praktika. PostgreSQL-le leiab ühiktestimiseks kaks enim soovitatavat raamistikku: pgTAP ja Plpgunit. Mõlemad raamistikud tekitavad andmebaasi funktsioonid, millega saab testides saadud tulemusi võrrelda vastavalt oodatud tulemusele. Mõlemas raamistikus tuleb sarnaselt iga loodava ühiktesti jaoks teha uus protseduur. Antud rakenduse juures on autor kasutanud Plpgunit raamistikku.

PostgreSQL-i sai tehtud ühiktestid sama põhimõtte järgi kui Java rakenduses JUnit-t kasutades. Autor testib FIFO ja keskmise meetodi arvutusloogikat ühiktestidega. Nii JUnit-s kui ka raamistikus on testimise idee sarnane – genereeritakse näiteandmed (Javas näiteobjektid, andmebaasis näiteandmeread), käivitatakse arvutus ning võrreldakse tulemusi. Javas on mugav kirjutada mitme erineva tulemuse võrdlemiseks järjest võrdlemisfunktsioone. Kui üks võrdlus ebaõnnestub, on kogu test ebaõnnestunud. Andmebaasitestimisel tuleb selleks teha rohkem tööd. Peale iga tulemuse võrdlemist, kasutades raamistikku funktsiooni ASSERT.IS_EQUAL() ei muutu automaatselt testi tulemus õnnestunuks või ebaõnnestunuks. Eraldi tuleb kontrollida funktsiooni väljundiks olevat *boolean* väärtust, kas tulemus oli võrdne oodatuga või ei, ning kui ei olnud, tuleb tagastada funktsioonist saadud tulemus. Selle tulemusena raamistik teab, et antud test ebaõnnestus. Kui tulemus oli võrdne võib edasi minna järgmise võrdlusega. Kui kõik võrdlused on edukalt läbitud, tuleb eraldi välja kutsuda funktsioon ASSERT.OK(), mis annab raamistikule teada, et antud test läbiti edukalt. Andmebaasi ühiktesti kood on välja toodud lisas 8.

Nii Javas kui ka andmebaasis tuleb testide jaoks tekitada testandmed – kuna rakendus on olekuta, siis testi jaoks loodud objektid ei mõjuta kuidagi andmeid, mida kasutatakse toodangus. Andmebaas on aga olekuga, mis tähendab, et testandmete tegemine on väga riskantne ning toodanguandmebaasis ei oleks kindlasti soovitatav neid hoida. Võimalus on need tekitada testide käivitamise ajal ning seda lähenemist on kasutatud ka antud rakenduse juures. Autor on teinud eraldi protseduuri testandmete tekitamiseks ning ka kustutamiseks. Sellega jääb aga siiski üles risk andmete sattumiseks live baasi, mistõttu neid teste ei tohiks toodangubaasi üldse paigaldada. Javas on testid aga rakenduse loomulik osa ja on sellega kaasas ka toodangukeskkonnas.

## 4.3 Modifitseeritavus

Antud rakendusel on lisaks FIFO ja keskmise arvutusmeetodile lisaks ka LIFO meetodil arvutamine. Kuna FIFO ja LIFO meetodi on oma olemuselt väga sarnased, on nende loogikate eraldi meetodites/protseduurides hoidmine koodi duplitseerimine. Koodi modifitseeritavuse testimiseks refaktoreerime antud meetodeid nii Javas kui andmebaasis. Koodi refaktoreerimise tulemused on näidatud FIFO arvutuse näite pealt.

Refaktoreerimise juures on väga tähtis, et eelnevalt oleks valmis kirjutatud testid, mis aitavad refaktoreerimisel tehtud vigu avastada. Hetkel on olemas testid nii Javas kui andmebaasis, seega võib alustada refaktoreerimisega.

### 4.3.1 Java koodi refaktoreerimine

Kuna FIFO ja LIFO arvutusmeetodid on sarnased (FIFO arvutuse klassi kood, mida autor refaktoreerima hakkab, on välja toodud lisas 2), võiksid nad kuuluda ühe ülemklassi alla. Selleks kasutab autor superklassi eraldamise metoodikat (Fowler, Refactoring. Improving the Design of Existing Code, 2003), mille käigus teeb autor esiteks uue abstraktse klassi *FirstOutCalculations* ning praegused arvutusmeetodiklassid *FifoStrategy* ja *LifoStrategy* muudab selle alamklassideks. Superklassi eraldamise käigus tuleb kasutada väljade üles tõmbamise ja meetodite üles tõmbamise metoodikaid (Fowler, Refactoring. Improving the Design of Existing Code, 2003). Väljade üles tõmbamise põhimõte seisneb selles, et kui kaks alamklassi omavad sama sisuga välju, tuleks need deklareerida ülemklassis. Liigutame *FirstOutCalculations* klassi alamklasside ühised globaalsed muutujad.

```
15 public abstract class FirstOutCalculations {
16
17     protected List<SecurityTransactionInCalculation> securityTransactionsInCalculation = new ArrayList<>();
18     protected List<SecurityTransactionInCalculation> securityTransactionsToBeRemoved = new ArrayList<>();
19
20     protected double quantity = 0;
21     protected double bookCost = 0;
22     protected double buyPrice = 0;
23     protected double realizedProfitLoss = 0;
```

**Joonis 6. Alamklasside muutujate deklareerimine ülemklassis**

Nii *FifoStrategy* kui ka *LifoStrategy* klassis on massiivne meetod *calculateAssetPerformance*, mis hetkel hoiab kogu tootluse arvutuse loogikat. Meetod on pikk ja lohisev ning selle peal saab rakendada meetodi väljatõmbamise metoodikat (Fowler, Refactoring. Improving the Design of Existing Code, 2003). Selle käigus tõstab autor välja ostu- ja müügitehingute loogika ning tekitab uued meetodid *calculateBuyTransaction* ja *calculateSellTransaction*. Lisaks tekitab autor uue meetodi dividendide lisamiseks realiseeritud tulule/kulule nimega *addDividendToRealizedProfitLoss* ning väärtpaberiandmete seadmiseks *setAssetData*. Peale refaktoreerimist on kood palju lühem ning kergemini mõistetavam ja hallatavam. Kõik uued meetodid on paigutatud *FirstOutCalculations* ülemklassi, et neid saaks kasutada nii *FifoStrategy* kui ka *LifoStrategy*. Refaktoreerimise tulemusel *FifoStrategy* klassi sisu:

```
public class FifoStrategy extends FirstOutCalculations implements CalculationStrategy {

    @Override
    public void calculateAssetPerformance(Asset asset) {
        List<SecurityTransaction> securityTransactionList = asset.getSecurityTransactionsThatAreIncludedInCalculation();

        if (securityTransactionList != null) {

            Collections.sort(securityTransactionList);

            for (SecurityTransaction securityTransaction : securityTransactionList) {
                double transactionQuantity = securityTransaction.getQuantity();
                if (transactionQuantity >= 0) {
                    calculateBuyTransaction(securityTransaction);
                } else {
                    Collections.sort(securityTransactionsInCalculation);
                    calculateSellTransaction(securityTransaction);
                }
            }
        }

        addDividendToRealizedProfitLoss(asset);

        setAssetData(asset);

    }
}
```

**Joonis 7. *FifoStrategy* klass peale refaktoreerimist**

### 4.3.2 Andmebaasiprotseduuri refaktoreerimine

Andmebaasi refaktoreerimisel ei saa kasutada superklassi eraldamise metoodikat, kuna andmebaasis puuduvad protseduuride vahel ülem-alam seosed, seoses sellega ei ole võimalik ka muutujaid paigutada ainult ühte kohta. Andmebaasides saab siiski kasutada meetodite väljatõmbamise metoodikat (Ambler & Sadalage, 2006), mille tulemusena tuleks igale taaskasutatavale koodilõigule teha eraldi protseduur andmebaasi. FIFO tootluse arvutuse protseduur, mida autor refaktoreerima hakkab, on välja toodud lisas 4.

Refaktoreerimist alustab autor ostutehingu kontrolli funktsiooni koostamisega. Ka SQL-s saab protseduurile sisendiks anda tabeli rida, mis objektorienteeritud keele mõistes oleks kui objekti sisendiks andmine. Seda lähenemist autor protseduuris ei kasuta, kuna SQL-s ei ole vaja tehingutüübi teadasaamiseks tervet andmerida. Lisaks on võimalik anda protseduurile sisendiks ka rohkem kui ainult 1 tabelirida, mis tähendaks, et protseduur väljastab ka vastuse kõikide tabeliridade kohta, mis ei ole aga soovitud tulemus.

```
CREATE OR REPLACE FUNCTION is_buy_transaction(IN transaction_id int, OUT is_buy_transaction boolean) AS $$
DECLARE
BEGIN
        select exists(
        select
                *
        from
                security_transaction st
        join
                transaction_type tt
                on st.transaction_type_id = tt.id
                and code = 'BUY'
        where
                st.id = transaction_id
        ) into is_buy_transaction;
END;
$$ LANGUAGE PLPGSQL;
```

**Joonis 8. Funktsioon ostutehingute tuvastamiseks**

Järgmisena eraldab autor suurest protseduurist ostutehingute tegevused eraldi protseduuri. Kahjuks kõiki koodiridu protseduuri üks ühele ümber tõsta ei anna. Ostutehingute puhul lisatakse ostutehingu andmed ajutisse tabelisse müügitehingute protsessimise tarvis ning andmed võetakse record andmetüübist. PostgreSQL aga protseduuri sisendina record tüüpi deklareerida ei luba. Seega on tarvilik protseduuri sisenditeks määrata kõik väärtused eraldi.

```
CREATE OR REPLACE FUNCTION first_out_calculate_buy_transaction(IN quantity numeric, IN book_cost numeric
        , IN row_quantity numeric, IN row_price numeric, IN row_transaction_date date
        , OUT out_quantity numeric, OUT out_book_cost numeric) AS $$
DECLARE

BEGIN
        out_quantity = quantity + row_quantity;
        out_book_cost = book_cost + (row_quantity * row_price);
        INSERT INTO tmp_transactions_in_calculation (price, quantity, transaction_date, deleted)
                VALUES (row_price, row_quantity, row_transaction_date, false);
END;
$$ LANGUAGE PLPGSQL;
```

**Joonis 9. Ostutehingute tegevuste protseduur**

Sama loogikat rakendab autor ka müügitehingutega, dividendide arvutamisega ning väärtpaberi andmete seadmistega. Saadud protseduurid on toodud välja lisas 9. Protseduuride eraldamisega sai tootluse arvutuse protseduurist välja tõsta ka mitmeid muutujaid, mis tähendab vähem koodi duplitseerimist FIFO ja LIFO tootluse arvutuse protseduurides.

```
CREATE OR REPLACE FUNCTION calculate_asset_performance_in_fifo (IN assetId int
        , OUT market_value numeric, OUT book_cost numeric, OUT quantity numeric, OUT buy_price numeric, OUT realized_profit_loss numeric
        , OUT unrealized_profit_loss numeric, OUT profit_loss numeric) AS $$
DECLARE
        security_transactions CURSOR FOR
                SELECT
                        ST.*
                FROM
                        SECURITY_TRANSACTION ST
                JOIN
                        TRANSACTION_TYPE TT
                        ON ST.TRANSACTION_TYPE_ID = TT.ID
                        AND TT.INCLUDED_IN_CALC
                WHERE
                        ASSET_ID = assetId
                ORDER BY
                        TRANSACTION_DATE;
        row record;
BEGIN
        book_cost = 0;
        quantity = 0;
        realized_profit_loss = 0;

        PERFORM first_out_create_tmp_transactions_table();

        FOR row in security_transactions
        LOOP
                IF
                        row.quantity > 0
                THEN
                        select out_quantity, out_book_cost into quantity, book_cost
                                from first_out_calculate_buy_transaction(quantity, book_cost, row.quantity, row.price, row.transaction_date);
                ELSE
                        select out_quantity, out_book_cost, out_realized_profit_loss into quantity, book_cost, realized_profit_loss
                                from first_out_calculate_sell_transaction (quantity, realized_profit_loss, row.quantity, row.price);
                END IF;
        END LOOP;

        select out_realized_profit_loss INTO realized_profit_loss from first_out_add_dividend_to_realized_profit_loss(assetId, realized_profit_loss);

        select out_buy_price, out_market_value, out_unrealized_profit_loss, out_profit_loss INTO buy_price, market_value, unrealized_profit_loss, profit_loss
                from first_out_calculate_asset_data(assetId, quantity, book_cost, realized_profit_loss);
END;
$$ LANGUAGE PLPGSQL;
```

**Joonis 10. Väärtpaberi tootluse arvutuse protseduur peale refaktoreerimist**

FIFO ja LIFO koode võrreldes selgub, et erinevus on ainult kahes koodireas ning erinevus seisneb selles, kas müügitehingu puhul eelnevaid ostutehinguid tuleks vaadata kõige vanemad eespool või uuemad eespool. Seega selle asemel, et meetodeid välja tõmmata protseduuridest, tundub lihtsam ja parem valik kasutada meetodi parametriseerimise (Ambler & Sadalage, 2006) metoodikat. Selle tulemusena jääks kahest protseduurist alles üks, protseduurile tekib

juurde uus parameeter, kas arvutame FIFO või LIFO järgi ning üks tingimuslause, kumba pidi ostutehinguid järjestada.

Loomulikult saab antud olukorras teha ka Java koodis kahest klassist ühe ning juhtida parameetriga, kas ostutehinguid käiakse läbi ühes või teises järjekorras. Igas olukorras aga ei ole võimalik niimoodi konsolideerida ja sellepärast soovis autor näidata, kuidas objektorienteeritud keeles ülemklassi tekitamisega saab duplitseerivat koodi kerge vaevaga vähendada, pannes kõik ühised muutujad ja meetodid ülemklassi ning jättes alamklassidesse vaid nendele klassidele omased meetodid ja muutujad.

## 4.4 Duplitseerimise vältimine

Modifitseeritavuse punktis oli näha, kuidas refaktoreerimisega oli Java koodis võimalik duplitseeritavat koodi vähendada, tõstes näiteks ühised muutujad ülemklassi. Ka SQL-s oli võimalik ühiseid muutujaid vähendada, liigutades nad eraldi vastava loogika protseduuridesse.

Peale andmebaasiprotseduuri refaktoreerimist jäi veel sinna alles mõned duplitseerivad koodiread. Näiteks on protseduuris kirjeldatud 7 erinevat väljundit, mis peavad FIFO ja LIFO protseduuridel alati kattuma, kuna neilt oodatakse samu väljundeid. Kui tekib uus väli, mida mõlemad protseduurid peaksid väljastama, siis tuleks teha parandus mõlemasse protseduuri. Üheks lahenduseks sellele on deklareerida andmebaasi uus tüüp, mille sees on kirjeldatud kõik väljad, mida protseduur väljastama peab. Ning protseduuri väljastab alati seda tüüpi.

```
CREATE TYPE first_out_parameters as (
        market_value numeric,
        book_cost numeric,
        quantity numeric,
        buy_price numeric,
        realized_profit_loss numeric,
        unrealized_profit_loss numeric,
        profit_loss numeric
);
```

**Joonis 11. Uue tüübi deklareerimine**

Autor deklareeris uue tüübi *first_out_parameters*, kus on kirjeldatud need 7 andmevälja, mida protseduur peab väljastama. Protseduuri tegemise algus on muutunud:

```
CREATE OR REPLACE FUNCTION calculate_asset_performance_in_fifo (assetId int
    , OUT market_value numeric, OUT book_cost numeric, OUT quantity numeric, OUT buy_price numeric
    , OUT realized_profit_loss numeric, OUT unrealized_profit_loss numeric, OUT profit_loss numeric) AS $$
```

**Joonis 12. Protseduuri sisendid ja väljundid enne**

```
CREATE OR REPLACE FUNCTION calculate_asset_performance_in_fifo (assetId int
    , OUT out_data first_out_parameters) AS $$
```

**Joonis 13. Protseduur väljastab uue tüübi**

Protseduuri kood muutus lihtsamaks, vähendati vigade tekitamise võimalust ning kui protseduur peaks väljastama mõne uue välja, tuleb muuta ainult tüübi definitsiooni.

Duplitseerivast koodist on alles jäänud veel kahe muutuja deklareerimine:

```
DECLARE
    security_transactions CURSOR FOR
        SELECT
            ST.*
        FROM
            SECURITY_TRANSACTION ST
        JOIN
            TRANSACTION_TYPE TT
            ON ST.TRANSACTION_TYPE_ID = TT.ID
            AND TT.INCLUDED_IN_CALC
        WHERE
            ASSET_ID = assetId
        ORDER BY
            TRANSACTION_DATE;
    row record;
```

**Joonis 14. Muutujate deklareerimine protseduuris**

Kuna ka kursori sisu on nii LIFO kui FIFO meetodi puhul sama, tuleks duplitseerimise vältimiseks teha sellest päringust vaade – see tagab selle, et mõlemad protseduurid saavad alati sama sisu kursorisse ning ka uued protseduurid ei pea koodi duplitseerima.

```
DECLARE
    security_transactions CURSOR FOR
        SELECT
            *
        FROM
            SECURITY_TRANSACTIONS_INCLUDED_IN_CALC
        WHERE
            ASSET_ID = assetId
        ORDER BY
            TRANSACTION_DATE;
    row record;
```

**Joonis 15. Muutujate deklareerimine vaate pealt**

Kursori loogika on viidud protseduurist välja ja on seega ühest kohast hallatav. SQL's saab päris palju duplitseerimist vähendada, kuid näiteks *cursor* ja *record* tüüpi muutujate deklareerimisest siiski lahti ei saa ning need tuleb eraldi deklareerida nii FIFO kui ka LIFO protseduuris.

## 4.5 Kapseldamine

Javas on kapseldust lihtne implementeerida, muutes klassi muutuja ainult klassisiseselt kättesaadavaks ning välismaailm saab välja väärtust lugeda või väärtust seada läbi avalike meetodite. Sellega on tagatud see, et väljadele ligipääs on ainult läbi kindlate meetodite, mistõttu saab allolevaid andmeid muuta nii, et välismaailm sellest midagi ei tea. Samuti on võimalik avalikke meetoditesse panna näiteks reegleid, millele antud välja väärtus vastama peab. Samuti on lihtne võimaldada välismaailmal ainult välja väärtust lugeda või ainult välja väärtust kirjutada.

Andmebaasis saab samuti kapseldust implementeerida, näiteks et tabelist andmeid lugeda ja kirjutada, tuleks kasutada salvestatud protseduure. Autori hinnangul on selline lähenemine aga keerulisem, kui Java koodi puhul. Üheks põhjuseks on see, et igale väljale või tabelile, vastavalt soovile, tuleks teha eraldi protseduur andmebaasi, mille tulemusena on andmebaasis väga palju erinevaid protseduure ning nende tegemiseks kulub märgatavalt palju aega. Teiseks põhjuseks on see, et andmebaasis tuleks tegeleda õiguste juhtimisega – kasutajal ei tohi olla otsest ligipääsu tabelitele, aga kasutajal peab olema õigus salvestatud protseduuridele.

## 4.6 Võrdluse kokkuvõte

Järgnevalt on autor välja toonud võrdluste kokkuvõtte iga teema lõikes, kuidas iga aspekt mõjutab otsust, kas valida andmebaasiprotseduur äriloogikakihiks või mitte.

**Tabel 3. Võrdluste kokkuvõte**

| | |
|---|---|
| Jõudlus | Autori hinnangul on jõudlus ainus põhjus, miks võiks kaaluda äriloogika tõstmist salvestatud protseduuridesse. Testide tulemusel selgus, et rakendusesisesed arvutused on tihti kiiremad, kui sarnased arvutused salvestatud protseduurides. Andmebaasieelised tulevad esile suurte andmemahtude ning keerukate andmepäringute puhul, kuna SQL-ga on võimalik vältida tsüklite kasutamist. Seega võiks autori hinnangul jõudluse mõttes kirjutada loogika siiski rakendusse, kuid jõudluskriitilisemate loogikate jaoks teha erand ning kirjutada nende jaoks andmebaasiprotseduur. |
| Testitavus | Autor leiab, et rakendusesiseseid meetodeid on ühiktestida mugavam kui andmebaasiprotseduure. Samas on sama tulemust võimalik saavutada ka andmebaasiprotseduuride testimise puhul, kuigi see nõuab rohkem tööd. Seega leiab autor, et kui äriloogika on paigutatud andmebaasiprotseduuridesse, ei ole testitavuse aspekt probleem ning kindlasti tuleks kogu protseduurides olev äriloogika katta ka ühiktestidega. |
| Modifitseeritavus | Autor leiab, et nii java koodi kui andmebaasiprotseduure on võimalik refaktoorida, tehes ühiskasutatavaid meetodeid Javas ning protseduure andmebaasis. Andmebaasiprotseduurid on muutunud mõnes mõttes sarnaseks objektorienteeritud koodiga, lubades vastu võtta ka andmekogumeid, mitte ainult ühekaupa väärtuseid. See kindlasti suurendab refaktoreerimise võimalusi andmebaasiprotseduurides. |

| | |
|---|---|
| Duplitseerimise vältimine | Autor leiab, et duplitseerimist on SQL’s võimalik vältida sarnaselt Javale. Kuna PostgreSQL’s ei ole võimalik deklareerida globaalseid muutujaid, siis jääb igasse protseduuri siiski alles lokaalsete muutujate deklareerimine, kuid SQL-i loogikaid on võimalik asendada näiteks vaatega ning veerud, mida protseduur väljastab, saab asendada uue deklareeritud tüübiga, mis objektorienteeritud koodiga võrreldes tähendaks justkui uue deklareeritud klassi väljastamist. |
| Kapseldamine | Kapseldust on võimalik saavutada nii Javas kui SQL’s, kuid autor leiab, et SQL’s täieliku kapseldamise saavutamine nõuab rohkem tööd – andmebaasi tuleb tekitada väga palju erinevaid protseduure või vaateid andmete lugemiseks ja kirjutamiseks. Kapseldus andmebaasides nõuab ka andmebaasiadministraatori tuge, mida lihtsamate rakenduste arendustel tavaliselt ei pruugi vaja minna. |

# 5. Kokkuvõte

Igas rakenduses on vähemal või suuremal määral äriloogikat, mis paigutatakse tänapäeval kas rakendusse või protseduuridesse. Antud töö üheks eesmärgiks oli analüüsida aspekte, mida tuleks arvesse võtta äriloogika paigutamise valiku puhul. Töö eesmärgiks oli ka välja tuua antud aspektide võrdlus kahe erineva lähenemise puhul ning lisaks oli töö eesmärgiks jõuda soovitusteni, millistel juhtudel tuleks äriloogika paigutada rakendusse ning millistel juhtudel salvestatud protseduuridesse.

Autor kirjeldas esimeses peatükis kihilise arhitektuuri põhimõtet, kus 3-kihilises arhitektuuris esitluskiht, äriloogikakiht ning andmeliidestuskiht. Iga kiht suhtleb ainult temast järgmise kihiga ega ole teadlik järgnevatest kihtidest. Kihilise arhitektuuri eelisteks on võimalus arendada kihte eraldiseisvalt, välja vahetada ühe kihi lahendus teisi kihte segamata jne. Autor tõi välja ka äriloogikakihi ehitamiseks kasutatavad erinevad mustrid.

Teises peatükis kirjeldas autor salvestatud protseduuride eeliseid ja puuduseid ning tõi välja võimalused, kuidas arendaja peaks võimalikult vähe kirjutama SQL koodi. Lisaks tõi autor välja aspektid, mida tuleks arvestada äriloogika paigutamisega salvestatud protseduuridesse: nendeks on jõudlus, testitavus, modifitseeritavus, arusaadavus, duplitseerimise vältimine, kapseldamine ning andmebaasi teisaldatavus.

Kolmas peatükk oli autoripoolne praktiline analüüs, mille käigus võrdles autor äriloogikat nii rakenduses kui salvestatud protseduurides järgmistes aspektides: jõudlus, testitavus, modifitseeritavus, duplitseerimise vältimine ja kapseldus.

Analüüsi käigus leidis autor, et äriloogikat tasuks hoida rakenduse koodis ning võiks andmebaasiprotseduuri tõsta vaid juhul, kui see annab tugeva jõudluse võidu. Andmebaasi tugevus jõudluse koha pealt tuleb välja keerukate ja andmemahukate päringute puhul. Tihti on mitte nii andmemahuka loogika arvutus kiirem aga just rakenduses, mistõttu tuleks loogika andmebaasi paigutamist kindlasti eelnevalt analüüsida. Testitavuse aspektist leidis autor, et palju kiirem ja mugavam on seda teha Javas. Kuigi sama lõpptulemuse saab saavutada ka andmebaasiprotseduuridega, nõuab see rohkem tööd. Modifitseeritavuse, duplitseerimise vältimise ja kapselduse puhul leidis autor, et nii Java kui andmebaasiprotseduurid pakuvad võimalusi kõikide teemade tegelemiseks. Autor leidis, et duplitseerimist on võimalik vältida

paremini Java koodis, kuna näiteks ei saanud muutujate deklareerimist teha mitmele protseduurile globaalseks.

# Summary

Analysis of Stored Procedures and Three-Layer Architecture

The aim of the thesis was to investigate and compare two approaches where to hold application business logic – should it be in the application business layer or in stored procedure. The aim was to examine whether stored procedures offer same opportunities as object oriented code. Also, the author identifeid aspects that should be taken into account when deciding whether to write business logic into stored procedures or what would be the shortcomings compared to holding business logic in application code.

The author made practical analysis to compare next aspects between application code and stored procedures: performance, testability, modifiability, avoiding duplication and encapsulation.

The author found out that business logic should usually be held in application code and should be placed in stored procedures only, if it guarantees better performance.

The author found out that performance is usually better in application code, but the strength of SQL comes out with big data and complex queries. Especially because database is set-based and thus can avoid cycles (cursors) that cannot be avoided usually in object oriented code.

The author used JUnit to make unit tests in Java and Plpgunit to make unit tests in PostgreSQL database. Same result was achieved in Java and database, but with JUnit it took much less code. Plpgunit framework needed very much extra handwork so that framework would be aware of the result of test.

From modifiability, avoiding duplication and encapsulation perspective author discovered that nearly same situation could be achieved with stored procedures as in java. For example the author refactored stored procedure and as the result duplicate code and duplicate variables was reduced. The problem still existed with declaring variables in every stored procedure that can be avoided in Java by putting commonly used variables in parent class.

# Kasutatud kirjandus

About PostgreSQL. (kuupäev puudub). Kasutamise kuupäev: 9. mai 2015. a., allikas PostgreSQL: http://www.postgresql.org/about/

Ambler, S. W., & Sadalage, P. J. (2006). Refactoring Databases: Evolutionary Database Design. Addison-Wesley.

Apache JMeter. (kuupäev puudub). Kasutamise kuupäev: 9. mai 2015. a., allikas Apache JMeter: http://jmeter.apache.org/

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-Oriented Software Architecture Volume 1: A System of Patterns. London: Wiley.

Evans, E. (2006). Domain-Driven Design: Tackling Complexity in the Heart of Software. Westford: Addison-Wesley.

Fakhar, A. (Oktoober 2010. a.). 3-Tier Architecture in asp.net using c#. Kasutamise kuupäev: 15. aprill 2015. a., allikas Programming Tutorials: http://nice-tutorials.blogspot.in/2010/10/3-tier-architecture-in-aspnet-using-c.html

Fowler, M. (2002). Patterns of Enterprise Application Architecture. Boston: Addison-Wesley.

Fowler, M. (Veebruar 2003. a.). Domain Logic and SQL. Kasutamise kuupäev: 19. aprill 2015. a., allikas http://martinfowler.com/articles/dblogic.html

Fowler, M. (2003). Refactoring. Improving the Design of Existing Code. Addison-Wesley.

Hambrick, P. (4. juuni 2013. a.). Advantages and Drawbacks of Using Stored Procedures for Processing Data. Kasutamise kuupäev: 22. aprill 2015. a., allikas Segue Technologies: http://www.seguetech.com/blog/06/04/Advantage-drawbacks-stored-procedures-processing-data

Lawry, K. (7. august 2012. a.). Why I Avoid Stored Procedures (And You Should Too). Kasutamise kuupäev: 22. aprill 2015. a., allikas Technimusings: https://kevinlawry.wordpress.com/2012/08/07/why-i-avoid-stored-procedures-and-you-should-too/

Mahmoodi, R. (27. juuli 2005. a.). 3-tier architecture in C#. Kasutamise kuupäev: 16. aprill 2015. a., allikas Code Project: http://www.codeproject.com/Articles/11128/tier-architecture-in-C

Mains, B. (28. aprill 2008. a.). Introduction to 3-Tier Architecture. Kasutamise kuupäev: 17. aprill 2015. a., allikas dotnetslackers.com: http://dotnetslackers.com/articles/net/IntroductionTo3TierArchitecture.aspx

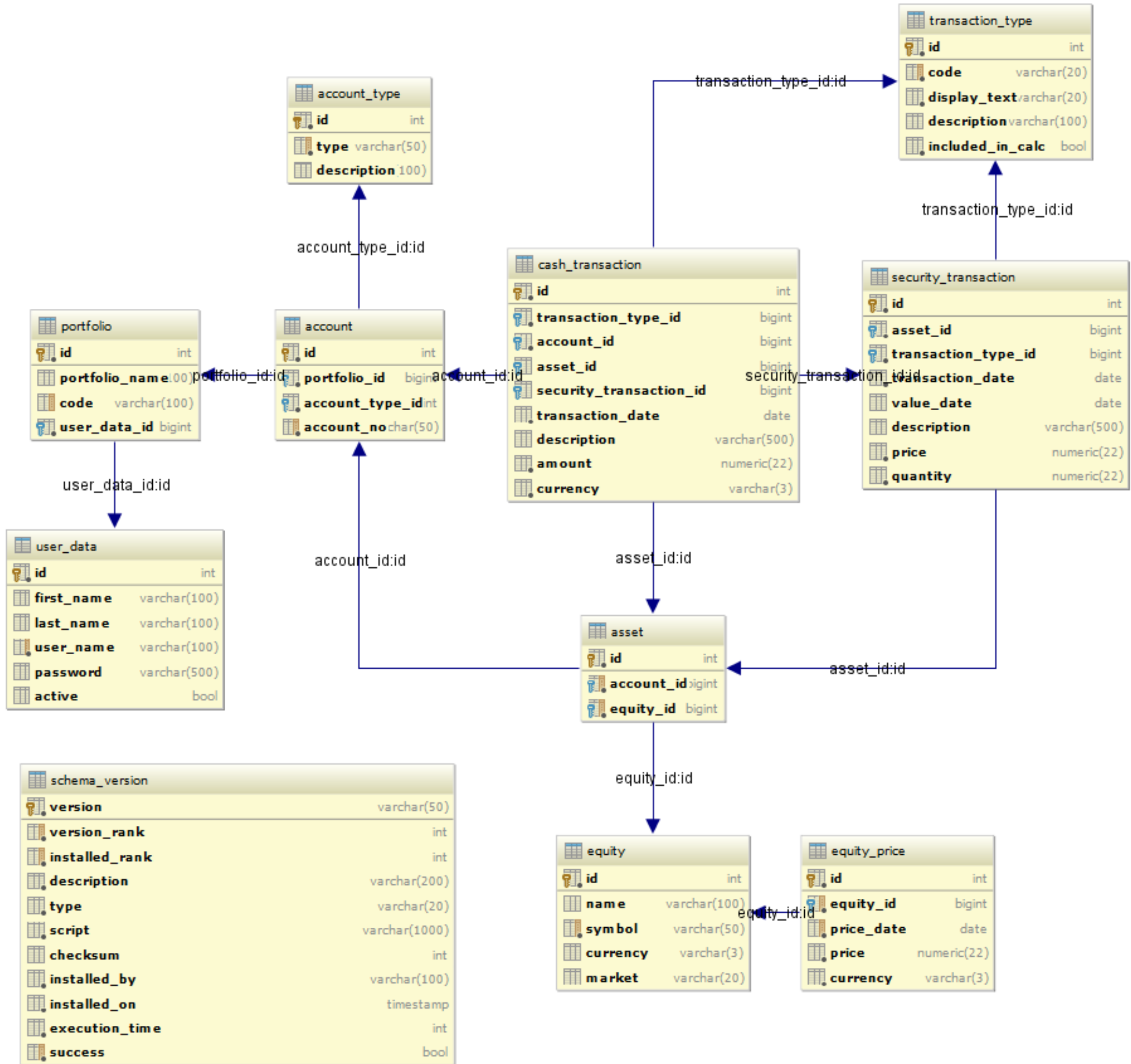
Meier, J. D., Homer, A., Hill, D., Taylor, J., Bansode, P., Wall, L., . . . Bogawat, A. (kuupäev puudub). Application Architecture Guide - Chapter 9 - Layers and Tiers. Kasutamise kuupäev: 15. aprill 2015. a., allikas Developer Guidance Share: http://www.guidanceshare.com/wiki/Application_Architecture_Guide_-_Chapter_9_-_Layers_and_Tiers

Papagelis, M. (kuupäev puudub). Kasutamise kuupäev: 16. aprill 2015. a., allikas http://queens.db.toronto.edu/~papaggel/courses/csc309/docs/lectures/web-architectures.pdf

pgTAP. (kuupäev puudub). Kasutamise kuupäev: 9. mai 2015. a., allikas pgTAP: http://pgtap.org/

PostgreSQL Unit Testing Framework (plpgunit). (kuupäev puudub). Kasutamise kuupäev: 6. mai 2015. a., allikas Github: https://github.com/mixerp/plpgunit

# Lisa 1. Andmebaasitabelite struktuur



# Lisa 2. FIFO meetodi arvutuse klass

```
public class FifoStrategy implements CalculationStrategy {

    private List<SecurityTransactionInCalculation> securityTransactionsInCalculation = new ArrayList<>();
    private List<SecurityTransactionInCalculation> securityTransactionsToBeRemoved = new ArrayList<>();

    private double quantity = 0;
    private double bookCost = 0;
    private double buyPrice = 0;
    private double realizedProfitLoss = 0;

    @Override
    public void calculateAssetPerformance(Asset asset) {
        List<SecurityTransaction> securityTransactionList = asset.getSecurityTransactionsThatAreIncludedInCalculation();
        Collections.sort(securityTransactionList);

        for (SecurityTransaction securityTransaction : securityTransactionList) {
            double transactionQuantity = securityTransaction.getQuantity();
            if (isBuyTransaction(securityTransaction)) {
                quantity += transactionQuantity;
                bookCost += transactionQuantity * securityTransaction.getPrice();
                securityTransactionsInCalculation.add(new SecurityTransactionInCalculation(
                                securityTransaction.getPrice(), securityTransaction.getQuantity())
                );
            } else {
                double sellQuantity = -transactionQuantity;
                for (SecurityTransactionInCalculation transactionInCalculation : securityTransactionsInCalculation) {
                    double transactionInCalculationQuantity = transactionInCalculation.getQuantity();
                    if (sellQuantity <= transactionInCalculationQuantity) {
                        double remainingQuantity = transactionInCalculationQuantity - sellQuantity;
                        if (remainingQuantity == 0) {
                            securityTransactionsToBeRemoved.add(transactionInCalculation);
                        } else {
                            transactionInCalculation.setQuantity(remainingQuantity);
                        }
                        quantity -= sellQuantity;
                        realizedProfitLoss += (sellQuantity * securityTransaction.getPrice())
                                - (sellQuantity * transactionInCalculation.getPrice());
                        break;
                    } else {
                        quantity -= transactionInCalculationQuantity;
                        sellQuantity -= transactionInCalculationQuantity;
                        realizedProfitLoss += (transactionInCalculationQuantity * securityTransaction.getPrice())
                                - (transactionInCalculationQuantity * transactionInCalculation.getPrice());
                        securityTransactionsToBeRemoved.add(transactionInCalculation);
                    }
                }
                securityTransactionsInCalculation.removeAll(securityTransactionsToBeRemoved);
                securityTransactionsToBeRemoved.clear();
                bookCost = getBookCost();
            }
        }
```

```java
        buyPrice = getBuyPrice();

        List<CashTransaction> cashTransactionList = asset.getCashTransactionsThatAreIncludedInCalculation();

        for (CashTransaction cashTransaction : cashTransactionList) {
            realizedProfitLoss += cashTransaction.getAmount();
        }

        double marketValue = quantity * asset.getEquity().getLastEquityPrice();
        double unRealizedProfitLoss = marketValue - bookCost;

        asset.setQuantity(quantity);
        asset.setBookCost(bookCost);
        asset.setBuyPrice(buyPrice);
        asset.setRealizedProfitLoss(realizedProfitLoss);
        asset.setUnRealizedProfitLoss(unRealizedProfitLoss);
        asset.setMarketValue(marketValue);
        asset.setProfitLoss(realizedProfitLoss + unRealizedProfitLoss);
    }

    public double getBuyPrice() {
        if (quantity == 0) {
            return 0;
        } else {
            return bookCost / quantity;
        }
    }

    public double getBookCost() {
        double bookCost = 0;
        for (SecurityTransactionInCalculation transactionInCalculation : securityTransactionsInCalculation) {
            bookCost += transactionInCalculation.getPrice() * transactionInCalculation.getQuantity();
        }
        return bookCost;
    }

    protected boolean isBuyTransaction(SecurityTransaction securityTransaction) {
        return securityTransaction.getTransactionType().getCode().equals("BUY");
    }
}
```

# Lisa 3. Keskmise meetodi arvutuse klass

```
public class AverageStrategy implements CalculationStrategy {

    @Override
    public void calculateAssetPerformance(Asset asset) {
        List<SecurityTransaction> securityTransactionList = asset.getSecurityTransactionsThatAreIncludedInCalculation();

        double quantity = 0;
        double bookCost = 0;
        double buyPrice = 0;
        double realizedProfitLoss = 0;

        if (securityTransactionList != null) {
            Collections.sort(securityTransactionList);

            for (SecurityTransaction securityTransaction : securityTransactionList) {
                double transactionQuantity = securityTransaction.getQuantity();
                quantity += transactionQuantity;
                if (transactionQuantity >= 0) {
                    bookCost += transactionQuantity * securityTransaction.getPrice();
                    buyPrice = bookCost / quantity;
                } else {
                    bookCost = quantity * buyPrice;
                    realizedProfitLoss += (transactionQuantity * buyPrice) - (transactionQuantity * securityTransaction.getPrice());
                    if (quantity == 0) {
                        buyPrice = 0;
                    }
                }
            }
        }

        List<CashTransaction> cashTransactionList = asset.getCashTransactionsThatAreIncludedInCalculation();

        if (cashTransactionList != null) {
            for (CashTransaction cashTransaction : cashTransactionList) {
                realizedProfitLoss += cashTransaction.getAmount();
            }
        }

        double marketValue = quantity * asset.getEquity().getLastEquityPrice();
        double unRealizedProfitLoss = marketValue - bookCost;

        asset.setQuantity(quantity);
        asset.setBookCost(bookCost);
        asset.setBuyPrice(buyPrice);
        asset.setRealizedProfitLoss(realizedProfitLoss);
        asset.setUnRealizedProfitLoss(unRealizedProfitLoss);
        asset.setMarketValue(marketValue);
        asset.setProfitLoss(realizedProfitLoss + unRealizedProfitLoss);
    }
}
```

# Lisa 4. FIFO meetodi arvutuse protseduur

```
CREATE OR REPLACE FUNCTION calculate_asset_performance_in_fifo (assetId int
    , OUT market_value numeric, OUT book_cost numeric, OUT quantity numeric, OUT buy_price numeric
    , OUT realized_profit_loss numeric, OUT unrealized_profit_loss numeric, OUT profit_loss numeric) AS $$
DECLARE
    security_transactions CURSOR FOR
        SELECT ST.*
        FROM SECURITY_TRANSACTION ST
        JOIN TRANSACTION_TYPE TT
            ON ST.TRANSACTION_TYPE_ID = TT.ID
            AND TT.INCLUDED_IN_CALC
        WHERE ASSET_ID = assetId
        ORDER BY TRANSACTION_DATE;

    cash_transactions CURSOR FOR
        SELECT CT.*
        FROM ASSET ASS
        JOIN CASH_TRANSACTION CT
            ON ASS.ID = CT.ASSET_ID
        JOIN TRANSACTION_TYPE TT
            ON CT.TRANSACTION_TYPE_ID = TT.ID
            AND TT.INCLUDED_IN_CALC
        WHERE ASS.ID = assetId
        ORDER BY TRANSACTION_DATE;

    last_price numeric;
    sell_quantity numeric;
    row record;
    row_tmp record;
    remaining_quantity numeric;
BEGIN
    book_cost = 0;
    quantity = 0;
    realized_profit_loss = 0;

    DROP TABLE IF EXISTS tmp_transactions_in_calculation;
    CREATE LOCAL TEMP TABLE tmp_transactions_in_calculation (
        row_num SERIAL PRIMARY KEY,
        price numeric(22, 6),
        quantity numeric(22, 6),
        transaction_date date,
        deleted boolean
    ) ON COMMIT DROP;

    last_price = get_asset_last_price(assetId);
```

```
FOR row in security_transactions
LOOP
    IF
        row.quantity > 0
    THEN
        quantity = quantity + row.quantity;
        book_cost = book_cost + (row.quantity * row.price);
        INSERT INTO tmp_transactions_in_calculation (price, quantity, transaction_date, deleted) VALUES (row.price, row.quantity, row.transaction_date, false);
    ELSE
        sell_quantity = -1 * row.quantity;
        FOR row_tmp IN SELECT * FROM tmp_transactions_in_calculation WHERE NOT deleted ORDER BY transaction_date
        LOOP
            IF
                sell_quantity <= row_tmp.quantity
            THEN
                remaining_quantity = row_tmp.quantity - sell_quantity;
                IF
                    remaining_quantity = 0
                THEN
                    UPDATE tmp_transactions_in_calculation SET deleted = TRUE WHERE row_num = row_tmp.row_num;
                ELSE
                    UPDATE tmp_transactions_in_calculation SET quantity = remaining_quantity WHERE row_num = row_tmp.row_num;
                END IF;

                quantity = quantity - sell_quantity;
                realized_profit_loss = realized_profit_loss + ((sell_quantity * row.price) - (sell_quantity * row_tmp.price));
                EXIT;
            ELSE
                quantity = quantity - row_tmp.quantity;
                sell_quantity = sell_quantity - row_tmp.quantity;
                realized_profit_loss = realized_profit_loss + ((row_tmp.quantity * row.price) - (row_tmp.quantity * row_tmp.price));
                UPDATE tmp_transactions_in_calculation SET deleted = TRUE WHERE row_num = row_tmp.row_num;
            END IF;
        END LOOP;

        book_cost = 0;
        FOR row_tmp IN SELECT * FROM tmp_transactions_in_calculation WHERE NOT deleted ORDER BY transaction_date
        LOOP
            book_cost = book_cost + (row_tmp.price * row_tmp.quantity);
        END LOOP;
    END IF;
END LOOP;

IF
    quantity = 0
THEN
    buy_price = 0;
ELSE
    buy_price = book_cost / quantity;
END IF;

FOR row in cash_transactions
LOOP
    realized_profit_loss = realized_profit_loss + row.amount;
END LOOP;

    market_value = quantity * last_price;
    unrealized_profit_loss = market_value - book_cost;
    profit_loss = realized_profit_loss + unrealized_profit_loss;
END;
$$ LANGUAGE PLPGSQL;
```

# Lisa 5. Keskmise meetodi arvutuse protseduur

```
CREATE OR REPLACE FUNCTION calculate_asset_performance_in_average (IN assetId int
    , OUT market_value numeric, OUT book_cost numeric, OUT quantity numeric, OUT buy_price numeric
    , OUT realized_profit_loss numeric, OUT unrealized_profit_loss numeric, OUT profit_loss numeric) AS $$
DECLARE
    security_transactions CURSOR FOR SELECT * FROM SECURITY_TRANSACTION WHERE ASSET_ID = assetId ORDER BY TRANSACTION_DATE;
    cash_transactions CURSOR FOR SELECT CT.*
    FROM ASSET ASS
        JOIN CASH_TRANSACTION CT
        ON ASS.ID = CT.ASSET_ID
    WHERE
        ASS.ID = assetId
    ORDER BY
        TRANSACTION_DATE;
    last_price numeric;
    row record;
BEGIN
    market_value = 0;
    book_cost = 0;
    quantity = 0;
    buy_price = 0;
    realized_profit_loss = 0;
    unrealized_profit_loss = 0;
    profit_loss = 0;

    last_price = get_asset_last_price(assetId);

    FOR row in security_transactions
    LOOP
        quantity = quantity + row.quantity;
        IF row.quantity > 0
        THEN
            book_cost = book_cost + (row.quantity * row.price);
            buy_price = book_cost / quantity;
        ELSE
            book_cost = quantity * buy_price;
            realized_profit_loss = realized_profit_loss + ((row.quantity * buy_price) - (row.quantity * row.price));
            IF quantity = 0
            THEN
                buy_price = 0;
            END IF;
        END IF;
    END LOOP;

    FOR row in cash_transactions
    LOOP
        realized_profit_loss = realized_profit_loss + row.amount;
    END LOOP;

    market_value = quantity * last_price;
    unrealized_profit_loss = market_value - book_cost;
    profit_loss = realized_profit_loss + unrealized_profit_loss;
END;
$$ LANGUAGE PLPGSQL;
```

# Lisa 6. Turuväärtuse arvutuse protseduur

```
CREATE OR REPLACE FUNCTION calculate_portfolio_market_value_all_portfolios(IN start_date DATE, IN end_date DATE)
    RETURNS VOID AS $$
DECLARE
BEGIN
    DELETE FROM MARKET_VALUE WHERE VALUE_DATE BETWEEN start_date AND end_date;

    INSERT INTO MARKET_VALUE (PORTFOLIO_ID, VALUE_DATE, VALUE, CURRENCY)
    SELECT
          PORTFOLIO_ID
        , VALUE_DATE
        , SUM(VALUE) AS VALUE
        , 'EUR' :: TEXT AS CURRENCY
    FROM (
        SELECT
              AC.PORTFOLIO_ID
            , ASS.EQUITY_ID
            , SUM(ST.QUANTITY) * PRICE.PRICE AS VALUE
            , PRICE.VALUE_DATE
        FROM PORTFOLIO PO
        JOIN ACCOUNT AC
            ON PO.ID = AC.PORTFOLIO_ID
        JOIN ACCOUNT_TYPE AT
            ON AC.ACCOUNT_TYPE_ID = AT.ID
            AND AT.TYPE = 'SECURITY'
        JOIN ASSET ASS
            ON AC.ID = ASS.ACCOUNT_ID
        JOIN SECURITY_TRANSACTION ST
            ON ASS.ID = ST.ASSET_ID
        JOIN
            (SELECT
                  A.VALUE_DATE
                , A.PRICE
                , A.EQUITY_ID
            FROM (
                SELECT DT.VALUE_DATE, PRICE, EQUITY_ID
                    , ROW_NUMBER() OVER (PARTITION BY EQUITY_ID, DT.VALUE_DATE ORDER BY PRICE_DATE DESC) AS ROW_NUM
                FROM
                    (SELECT GENERATE_SERIES(START_DATE, END_DATE, '1 day') AS VALUE_DATE) AS DT
                JOIN
                    EQUITY_PRICE EP ON EP.PRICE_DATE <= DT.VALUE_DATE
                ) A
            WHERE
                A.ROW_NUM = 1
            ) PRICE
            ON ST.TRANSACTION_DATE <= PRICE.VALUE_DATE
            AND ASS.EQUITY_ID = PRICE.EQUITY_ID
        GROUP BY AC.PORTFOLIO_ID, ASS.EQUITY_ID, PRICE.VALUE_DATE, PRICE.PRICE
        ) A
    GROUP BY
          PORTFOLIO_ID
        , VALUE_DATE;
END;
$$ LANGUAGE PLPGSQL;
```

# Lisa 7. Turuväärtuse arvutuse protseduur tsükliga

```
CREATE OR REPLACE FUNCTION calculate_portfolio_market_value_loop_all_portfolios(IN start_date DATE, IN end_date DATE)
    RETURNS VOID AS $$
DECLARE
    dt record;
    prtfl record;
    accnt record;
    asst record;
    price_var numeric;
    quantity_var numeric;
    value_var numeric;
BEGIN
    FOR dt IN
        SELECT GENERATE_SERIES AS DT
        FROM GENERATE_SERIES(START_DATE, END_DATE, '1 day')
    LOOP
        FOR prtfl IN
            SELECT * FROM PORTFOLIO
        LOOP
            value_var = 0;
            FOR accnt IN
                SELECT A.*
                FROM ACCOUNT A
                JOIN ACCOUNT_TYPE AT ON A.ACCOUNT_TYPE_ID = AT.ID AND AT.TYPE = 'SECURITY'
                WHERE PORTFOLIO_ID = prtfl.ID
            LOOP
                FOR asst IN
                    SELECT A.*
                    FROM ASSET A
                    WHERE ACCOUNT_ID = accnt.ID
                LOOP
                    SELECT PRICE into price_var
                    FROM (
                        SELECT PRICE, ROW_NUMBER() OVER (PARTITION BY EQUITY_ID ORDER BY PRICE_DATE DESC) AS ROW_NUM
                        FROM EQUITY_PRICE
                        WHERE EQUITY_ID = asst.EQUITY_ID
                            AND PRICE_DATE <= dt.dt
                    ) A WHERE A.ROW_NUM = 1;
                    SELECT SUM(QUANTITY) INTO quantity_var
                    FROM SECURITY_TRANSACTION
                    WHERE ASSET_ID = asst.id AND TRANSACTION_DATE <= dt.dt;
                    value_var = value_var + (COALESCE(quantity_var, 0) * price_var);
                END LOOP;
            END LOOP;

            IF NOT EXISTS (SELECT * FROM MARKET_VALUE WHERE PORTFOLIO_ID = prtfl.ID AND VALUE_DATE = dt.dt)
            THEN
                INSERT INTO MARKET_VALUE (PORTFOLIO_ID, VALUE_DATE, VALUE, CURRENCY)
                    VALUES (prtfl.ID, dt.dt, value_var, 'EUR');
            ELSE
                UPDATE MARKET_VALUE SET VALUE = value_var WHERE PORTFOLIO_ID = prtfl.ID AND VALUE_DATE = dt.dt;
            END IF;
        END LOOP;
    END LOOP;
END;
$$ LANGUAGE PLPGSQL;
```

# Lisa 8. Andmebaasi ühiktestid

```
CREATE OR REPLACE FUNCTION unit_tests.asset_performance_in_fifo_test()
    RETURNS test_result AS $$
DECLARE
    message test_result;
    result boolean;

    asset_id int;
    mv numeric;
    bc numeric;
    q numeric;
    bp numeric;
    rpl numeric;
    upl numeric;
    pl numeric;
BEGIN
    PERFORM tests_setup();

    SELECT ASS.ID INTO ASSET_ID
    FROM ASSET ASS
    JOIN ACCOUNT ACC
    ON ASS.ACCOUNT_ID = ACC.ID
    WHERE ACC.ACCOUNT_NO = 'TEST_SECURITY';

    SELECT * FROM calculate_asset_performance_in_fifo(asset_id) INTO mv, bc, q, bp, rpl, upl, pl;

    PERFORM remove_test_data();

    SELECT * FROM ASSERT.IS_EQUAL(370.0, mv) INTO message, result;
    IF
        NOT RESULT
    THEN
        RETURN MESSAGE;
    END IF;

    SELECT * FROM ASSERT.IS_EQUAL(362.5,  bc) INTO message, result;
    IF
        NOT RESULT
    THEN
        RETURN MESSAGE;
    END IF;

    SELECT * FROM ASSERT.IS_EQUAL(25.0, q) INTO message, result;
    IF
        NOT RESULT
    THEN
        RETURN MESSAGE;
    END IF;

    SELECT * FROM ASSERT.IS_EQUAL(14.5, bp) INTO message, result;
    IF
        NOT RESULT
    THEN
        RETURN MESSAGE;
    END IF;
```

```
    SELECT * FROM ASSERT.IS_EQUAL(256.25, rpl) INTO message, result;
    IF
        NOT RESULT
    THEN
        RETURN MESSAGE;
    END IF;

    SELECT * FROM ASSERT.IS_EQUAL(7.5, upl) INTO message, result;
    IF
        NOT RESULT
    THEN
        RETURN MESSAGE;
    END IF;

    SELECT * FROM ASSERT.IS_EQUAL(263.75, pl) INTO message, result;
    IF
        result
    THEN
        SELECT ASSERT.OK('VERY GOOD') INTO MESSAGE;
    END IF;

    RETURN message;
END;
$$ LANGUAGE PLPGSQL;
```

# Lisa 9. Uued protseduurid peale FIFO meetodi arvutuse protseduuri refaktoreerimist

```
CREATE OR REPLACE FUNCTION first_out_calculate_sell_transaction(IN quantity numeric, IN realized_profit_loss numeric
        , IN row_quantity numeric, IN row_price numeric
        , OUT out_quantity numeric, OUT out_book_cost numeric, OUT out_realized_profit_loss numeric) AS $$
DECLARE
        sell_quantity numeric;
        remaining_quantity numeric;
        row_tmp record;
BEGIN
        out_quantity = quantity;
        out_realized_profit_loss = realized_profit_loss;
        sell_quantity = -1 * row_quantity;
        FOR row_tmp IN SELECT * FROM tmp_transactions_in_calculation WHERE NOT deleted ORDER BY transaction_date
        LOOP
                IF
                        sell_quantity <= row_tmp.quantity
                THEN
                        remaining_quantity = row_tmp.quantity - sell_quantity;
                        IF
                                remaining_quantity = 0
                        THEN
                                UPDATE tmp_transactions_in_calculation SET deleted = TRUE WHERE row_num = row_tmp.row_num;
                        ELSE
                                UPDATE tmp_transactions_in_calculation SET quantity = remaining_quantity WHERE row_num = row_tmp.row_num;
                        END IF;

                        out_quantity = out_quantity - sell_quantity;
                        out_realized_profit_loss = out_realized_profit_loss + ((sell_quantity * row_price) - (sell_quantity * row_tmp.price));
                        EXIT;
                ELSE
                        out_quantity = out_quantity - row_tmp.quantity;
                        sell_quantity = sell_quantity - row_tmp.quantity;
                        out_realized_profit_loss = out_realized_profit_loss + ((row_tmp.quantity * row_price) - (row_tmp.quantity * row_tmp.price));
                        UPDATE tmp_transactions_in_calculation SET deleted = TRUE WHERE row_num = row_tmp.row_num;
                END IF;
        END LOOP;

        out_book_cost = 0;
        FOR row_tmp IN SELECT * FROM tmp_transactions_in_calculation WHERE NOT deleted ORDER BY transaction_date
        LOOP
                out_book_cost = out_book_cost + (row_tmp.price * row_tmp.quantity);
        END LOOP;
END;
$$ LANGUAGE PLPGSQL;
```

```
CREATE OR REPLACE FUNCTION first_out_add_dividend_to_realized_profit_loss(IN assetId int, IN realized_profit_loss numeric
        , OUT out_realized_profit_loss numeric) AS $$
DECLARE
        cash_transactions CURSOR FOR
                SELECT
                        CT.*
                FROM
                        ASSET ASS
                JOIN
                        CASH_TRANSACTION CT
                        ON ASS.ID = CT.ASSET_ID
                JOIN
                        TRANSACTION_TYPE TT
                        ON CT.TRANSACTION_TYPE_ID = TT.ID
                        AND TT.INCLUDED_IN_CALC
                WHERE
                        ASS.ID = assetId
                ORDER BY
                        TRANSACTION_DATE;
BEGIN
        out_realized_profit_loss = realized_profit_loss;
        FOR row in cash_transactions
        LOOP
                out_realized_profit_loss = out_realized_profit_loss + row.amount;
        END LOOP;

END;
$$ LANGUAGE PLPGSQL;
```

```
CREATE OR REPLACE FUNCTION first_out_create_tmp_transactions_table() RETURNS VOID AS $$
DECLARE
BEGIN
        CREATE LOCAL TEMP TABLE tmp_transactions_in_calculation (
                row_num SERIAL PRIMARY KEY,
                price numeric(22, 6),
                quantity numeric(22, 6),
                transaction_date date,
                deleted boolean
        ) ON COMMIT DROP;
END;
$$ LANGUAGE PLPGSQL;
```

```
CREATE OR REPLACE FUNCTION first_out_calculate_asset_data(IN asset_id int, IN quantity numeric, IN book_cost numeric, IN realized_profit_loss numeric
        , OUT out_buy_price numeric, OUT out_market_value numeric, OUT out_unrealized_profit_loss numeric, OUT out_profit_loss numeric) AS $$
DECLARE
        last_price numeric;
BEGIN

        last_price = get_asset_last_price(asset_id);

        IF
                quantity = 0
        THEN
                out_buy_price = 0;
        ELSE
                out_buy_price = book_cost / quantity;
        END IF;

        out_market_value = quantity * last_price;
        out_unrealized_profit_loss = out_market_value - book_cost;
        out_profit_loss = realized_profit_loss + out_unrealized_profit_loss;
END;
$$ LANGUAGE PLPGSQL;
```