TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Kevin Raja 193536IAIB

# Koodibaasi refaktoreerimine ja üleviimine kihiliselt arhitektuurilt heksagonaalsele arhitektuurile riigiasutuse pärandisüsteemi näitel

Bakalaureusetöö

Juhendaja: Ants Torim
PhD

Tallinn 2023

# Autorideklaratsioon

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

Autor: Kevin Raja

22.05.2023

# Annotatsioon

Käesoleva töö eesmärk on anda ülevaade konkreetse näite põhjal pärandisüsteemi refaktoreerimisest ja arhitektuurilisest üleminekust. Olemasolev süsteem on vigane ning raskesti hallatav, millest on tingitud refaktoreerimise ja restruktureerimise vajadus tagamaks projekti edasiarendamise võimekust ja jätkusuutlikust.

Töö olulisemateks tulemusteks on pärandisüsteemi koodibaasi refaktoreerimine kasutamaks parimaid arendustavasid ning liikuda maksimaalses võimalikus mahus kihiliselt arhitektuurilt heksagonaalsele arhitektuurile. Töö moodustub teoreetilisest ja praktilisest osast, kus teoreetiline osa koosneb erialasest- ning teaduslikust kirjandusest.

Lõputöö on kirjutatud eesti keeles ning sisaldab 35 lehekülge teksti, 7 peatükki, 22 joonist.

# Abstract

Refactoring the Codebase and Transitioning from Layered Architecture to Hexagonal Architecture Based On an Example of a State Institution Legacy System

The aim of the thesis is to provide an overview of the refactoring and architectural transition of a legacy system based on a specific example. The existing system is faulty, which makes it difficult to maintain and manage - this has led to the need for refactoring and restructuring to ensure the project's capacity for further developments and sustainability.

The key outcomes of this thesis include the refactoring of the legacy system's codebase to use the best development practices and the transition from a layered architecture to hexagonal architecture to the maximum possible extent. The thesis comprises both theoretical and practical components, with the theoretical part encompassing specialized and scientific literature.

The thesis is in Estonian and contains 35 pages of text, 7 chapters, 22 figures.

# Lühendite ja mõistete sõnastik

| | |
|---|---|
| *As-Is* | Hetkeseis |
| *As-Was* | Eelnev, algne |
| DIP | Dependency Inversion Principle, tarkvarakomponendid peaksid sõltuma abstraktsioonidest, mitte konkreetsetest klassidest |
| *Gitlab pipeline* | Automatiseeritud protsess mille käigus kompileeritakse, ehitatakse, testitakse ja paigaldatakse rakendus. |
| *God class* | Klass, mis ei vasta SRP printsiibile, on massiivne ning teab ja teeb liiga palju |
| *Guard clause* | Tingimuslause, mis kontrollib teatud tingimuste olemasolu või puudumist ning tagastab kohe, kui need tingimused on täidetud. Tavaliselt kasutatakse meetodi alguses |
| Id | Unikaalne identifikaator |
| ISP | Interface Segregation Principle, liidese kasutaja ei pea sõltuma meetoditest, mida tal vaja ei lähe printsiip |
| LSP | Liskov Substitution Principle, Liskovi asenduse printsiip |
| *Merge* | Ühe koodiharu liitmine teisega |
| OCP | Open/Closed Principle, avatud laiendamiseks aga suletud muutmiseks printsiip |
| Pärandisüsteem | *Legacy system*, vananenud tarkvarasüsteem, mis on varem kasutusel olnud, ning mis on ka tänapäeval kasutuses |
| Refaktoreerimine | Olemasoleva koodi ümberstruktureerimine muutmata koodi käitumist |
| REST | *Representational State Transfer*, tarkvaraarhitektuur, mis seab veebirakendustele kindlad suhtlemise reeglid |
| SOAP | *Simple Object Access Protocol*, rakenduste suhtlemise protokoll |
| *Shortcut* | Otsetee, keerulisema korrektse lahenduse asemel lihtsama ja vigadega lahenduse kasutamine |
| SOLID | Viis objekti-orienteeritud programmeerimise printsiipi |
| SRP | Single-Responsibility Principle, klassil peaks olema üks põhjusmuutumiseks printsiip |
| *Timeout* | Erind, mis visatakse kui mingi protsess võtab rohkem aega kui talle on eraldatud |

| | |
|---|---|
| *To-Be* | Plaanitav seis tulevikuks |
| *Vs* | *Versus*, vastandama/võrdlema |
| *Wrapper class* | Klass, mis rakendab teist klassi andmata juurde lisaväärtust |
| X-tee | Turvaline ja tõestusväärtust tagav andmevahetuskiht |

# Sisukord

# Jooniste loetelu

# 1 Sissejuhatus

Eesti valitsus kiitis 2022. aastal heaks eelnõu, mille tulemusel tuleb ümber hinnata teatud maksu. Maksu alusandmete ja arvutuskäigu muutumise tagajärjel vajab uuendamist süsteem, mis haldab antud maksuga seotud tegevusi. Lisaks seaduslikele muudatustele on Kliendil soov luua süsteemile uusi featuure. Selle töö kontekstis on Klient riigiasutus, kelle haldusalas (valduses) on antud kirjatööl põhinev infosüsteem. "Teatud maks" jääb defineerimata ning hägusaks selleks, et vältida lõputöö raames õiguslikku bürokraatiat.

## 1.1 Taust ja probleem

Käesoleva töö raames ei arendata uut infosüsteemi, vaid täiendatakse olemasolevat süsteemi, s.t Teostaja pärib Kliendilt pärandsüsteemi. Antud kontekstis Teostaja on erasektorist pärit ettevõte, mis viib täide Kliendi soovitud infosüsteemi muudatused ja täiendused. Lisaks kirjeldab kirjatöö Teostaja refaktoreerimise ja restruktureerimise tegevusi päritud koodibaasi raames.

Tõdedes, et olemasoleva infosüsteemi ümberkirjutamine ei mahu nii ajalistesse kui ka finantsilistesse piiridesse, tuleb olemasoleva süsteemi peale edasi arendada. Pärandisüsteemi edasiarendamine toob endaga kaasa mitmeid probleeme, mida antud töös adresseeritakse. Täpsemalt käsitletakse töös põhiprobleemidena arusaamatut, keerukat ja parimatele tavadele mittevastavat koodi, madalat testikatvust ning arhitektuuri- ja struktuuriliste reeglite rikkumisest tulenevaid mõjusid. Koodibaas vajab refaktoreerimist ja restruktureerimist eelkõige infosüsteemi jätkusuutlikkuse- ja hallatavuse tagamiseks ning järgneva lisaarenduse hõlbustamiseks.

Projektis töötab kuus arendajat ning arendajaid, kes on töötanud kogu projekti vältel, on kaks, kellest üks on antud lõputöö autor. Umbkaudselt kolmandik projektis tehtud refaktoreerimisest ning testikatvuse tõstmisest on sooritatud töö autori poolt. Töö autor on ka peamine eesrakenduse arendaja.

Käesolev lõputöö omab autori arvates hariduslikku- ja (eelkõige) erialast väärtust lugejale, kuna võimaldab hinnata heksagonaalse arhitektuuri võimalikke eelised, keerukust ja probleeme, kõrvutades praktilise osa antud valdkonna teadusliku- ja erialase kirjandusega. Lõputööst saab päriselulise ülevaate pärandisüsteemi refaktoreerimisest.

## 1.2 Töö eesmärk

Käesolevas peatükis kirjeldatakse detailsemalt antud lõputöö eesmärke. Töö eesmärgiks on viia olemasolev tarkvarasüsteem üle kihiliselt arhitektuurilt maksimaalses võimalikus piiris heksagonaalsele arhitektuurile. Järgmine eesmärk on refaktoreerida koodibaasi vastavalt parimatele arendustavadele ja printsiipidele ning tõsta testikatvust vähemalt Kliendi poolt projektile ettenähtud tasemele.

### 1.2.1 Kihiliselt arhitektuurilt üleminek heksagonaalsele arhitektuurile

Viia kood üle kihiliselt arhitektuurilt maksimaalses võimalikus piiris heksagonaalsele arhitektuurile. Maksimaalses võimalikus piiris peamiselt põhjusel, et teatud osa koodist on üksteisega tihedas sõltuvuses ning nende loogika läbipõimunud – sellest tulenevalt on selliste osade ümberkirjutamine skoobist väljas, kuna antud muudatuste realiseerimine ei mahuks ajaraamidesse.

### 1.2.2 Koodibaasi refaktoreerimine

Refaktoreerida koodibaasi selliselt, et see vastaks parimatele arendustavadele ja printsiipidele, oleks arendajatele üheselt arusaadav ning ei sisaldaks projektis kasutatud koodianalüüsi tööriistade poolt tuvastatavaid vigu.

### 1.2.3 Testikatvuse tõstmine

Antud töö raames käsitletakse testidena üksusteste. Üksustestide kirjutamise peamiseks eesmärgiks on olemasoleva loogika toimimise valideerimine vastavalt nõuetele, ning anda arendajatele kindlust ja tagasisidet, et refaktoreerimise tulemusena jääb olemasolev loogika tööle. Kõrgem üksustestide katvus hõlbustab edasist arendust.

Testikatvust tõstetakse vähemalt Kliendi poolt projektile ettenähtud tasemele, milleks on 60%.

## 1.3 Metoodika

Antud jaotises kirjeldatakse töös kasutatud arhitektuurilise ülemineku, refaktoreerimise ja testimise metoodikaid.

### 1.3.1 Arhitektuuriline üleminek

Arhitektuurilise ülemineku protsessi alustatakse heksagonaalse arhitektuuri tavadele vastava struktuuri loomisega ning võimalike olemasolevate moodulite ümberliigutamisega vastloodud struktuuri. Pärast eelneva sammu realiseerimist toimub üleminek järk-järgult arendustükkide käigus.

Tulemusi valideerime SOLID [1] printsiipidele, kokkulepitud nimetustele, korrektsele nähtavuse tasemele ning üleüldiselt heksagonaalsest arhitektuurist tulenevate nõuetele ja soovitustele vastavusega.

### 1.3.2 Refaktoreerimine

Projekti algfaasis refaktoreeritakse kasutuskohad, mille identifitseerivad erinevad koodianalüüsimise tööriistad [2], [3], [4], [5], edaspidi jätkub refaktoreerimine jooksvalt arendustükkide põhiselt projekti lõpuni.

Refaktoreerimise edukust valideeritakse peamiselt *SOLID* printsiipide põhiselt, ***Clean Code***'le vastavusest, välise käitumise säilitamisega ja kohtades, kus võimalik, ka jõudluse järgi. Lisaks eelmainitule on kohustuslik iga sisse viidava muudatuse puhul saada heakskiit teis(t)elt arendaja(te)lt.

### 1.3.3 Testikatvuse tõstmine

Testikatvuse tõstmise esimeseks sammuks on ülevaate saamine projekti testikatvuse hetkeseisust. Katvuse analüüsimiseks kasutame selleks ettenähtud tööriistu. Testide kirjutamiseks valitakse eelisjärjekorras moodulid, mille relatiivne testikatvus on madalaim.

Tulemusi valideeritakse kasutades samu koodianalüüsimise tööriistu, mida kasutati algseisu hindamiseks. Tulemusi võib pidada edukaks, kui koodikatvuse tase rahuldab projektile ettenähtud piiri ja äriloogiliselt oluliste kohtade katvus ületab üheksakümne protsendi lävendi. Üks koht, mida antud metoodika ei kata, on erinevate äärejuhtumite valideerimine.

## 1.4 Ülevaade tööst

Töö teine peatükk keskendub olemasoleva- ja soovitava süsteemi arhitektuuri kirjeldamisele. Kolmanda peatüki eesmärk on anda ülevaade refaktoreerimise protsessist. Neljas peatükk kirjeldab testimist ja selle olulisust refaktoreerimise protsessis. Viies peatükk kirjeldab eelmainitud peatükkide rakendamist läbi arendusprotsesside ja nende tulemusi.

# 2 Arhitektuur

Süsteemi tarkvaraarhitektuur esindab/sümboliseerib üldise süsteemi struktuuri ja käitumisega seotud disainiotsuseid. Arhitektuur aitab sidusrühmadel mõista ja analüüsida, kuidas süsteem saavutab olulised omadused nagu muudetavus, kättesaadavus ja turvalisus [6]. Rakendused millel puudub formaalne arhitektuur on üldiselt tihedalt seotud, haprad, raskesti muudetavad ning neil puudub selge visioon või suund [7].

Tarkvaraarhitektuuri IEEE 42010 standardi definitsioon: "*Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*" [8].

## 2.1 SOLID printsiibid

SOLID printsiibid kirjeldavad, kuidas korraldada funktsioone ja andmestruktuure klassideks, ning kuidas need klassid peaksid omavahel seotud olema. Sõna "klass" kasutamine ei tähenda, et need printsiibid kehtivad ainult objekti-orienteeritud tarkvara kohta. Klass on lihtsalt funktsioonidest ja andmetest koosnev seotud grupp. Iga tarkvarasüsteem omab sellised gruppide koondisi, olenemata sellest, kas neid nimetatakse klassideks või mitte. SOLID printsiibid kehtivad nende/selliste gruppide kohta. Printsiipide eesmärk on aidata luua tarkvara, mis talub muudatusi ja on lihtsasti arusaadav [9].

Akronüüm *SOLID* koosneb viiest tähest, kus iga täht sümboliseerib ühte printsiipi:

- **S** The Single-Responsibility Principle (SRP) [1]
- **O** The Open/Closed Principle (OCP) [1]
- **L** The Liskov Substitution Principle (LSP) [1]
- **I** The Interface Segregation Principle (ISP) [1]
- **D** The Dependency Inversion Principle (DIP) [1]

**The Single-Responsibility Principle (SRP)**

''*A class should have only one reason to change*'' (Micah Martin, Robert C. Martin, 2006).

*SRP* kontekstis defineeritakse vastutust kui põhjust muutumiseks. Kui on võimalik mõelda rohkem kui üks põhjus klassi muutmiseks, siis on sellel klassil rohkem kui üks vastutus [1]. Kokkuvõtlikult – klassil peab peaks olema üks konkreetne ülesanne/vastutus/eesmärk.

**The Open/Closed Principle (OCP)**

*''Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification*'' (Micah Martin, Robert C. Martin, 2006).

Moodulid, mis vastavad *OCP*'le omavad kahte põhiatribuuti.

A. Avatud laiendamiseks tähendab, et mooduli käitumist saab laiendada. Kui rakenduse nõuded muutuvad, saab laiendada moodulit uute käitumiste lisamisega, mis rahuldavad neid muutusi [1].
B. Suletud muutmiseks tähendab, et mooduli käitumise laiendamine ei too kaasa muudatusi mooduli lähtekoodis ega binaarkoodis [1].

Kokkuvõtlikult – klassid tuleb struktureerida selliselt, et neid on võimalik uue funktsionaalsuse vajadusel laiendada, tehes seda muutmata olemasolevat koodi.

**The Liskov Substitution Principle (LSP)**

*''Subtypes must be substitutable for their base types*'' (Micah Martin, Robert C. Martin, 2006).

Alamklasse peab olema võimalik asendada nende ülemklassidega nii, et rakendus katki ei lähe. Eelnevast võib järeldada, et alamklassid peavad käituma samamoodi nagu nende ülemklassid [1]. Teisiti öeldes, alamtüüpide asendatavus võimaldab moodulit, mis on väljendatud põhitüübi terminites, laiendada ilma muudatusteta [1].

**The Interface Segregation Principle (ISP)**

*''Clients should not be forced to depend on methods they do not use*'' (Micah Martin, Robert C. Martin, 2006).

Liidese kasutajad peaksid sõltuma ainult nendest meetoditest, mida nad kasutavad (välja kutsuvad) [1]. Liidesed paljude meetoditega tuleks jagada mitmeteks väiksemateks Kliendi-spetsiifilisteks liidesteks [1].

**The Dependency-Inversion Principle (DIP)**

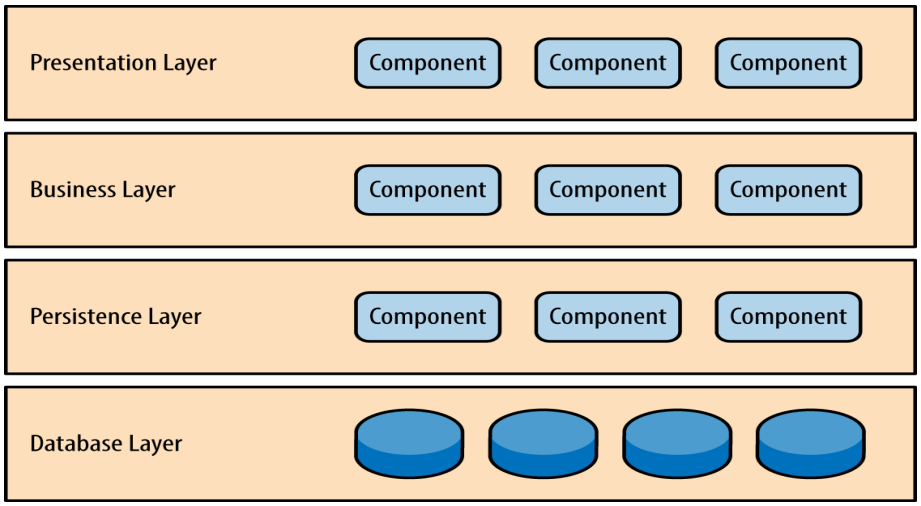
''*A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*

*B. Abstractions should not depend upon details. Details should depend upon abstractions*'' (Micah Martin, Robert C. Martin, 2006).

Lühidalt kirja panduna ütleb printsiip, et komponendid peavad sõltuma abstraktsioonidest, mitte konkreetsetest klassidest [1].

## 2.2 Kihiline arhitektuur

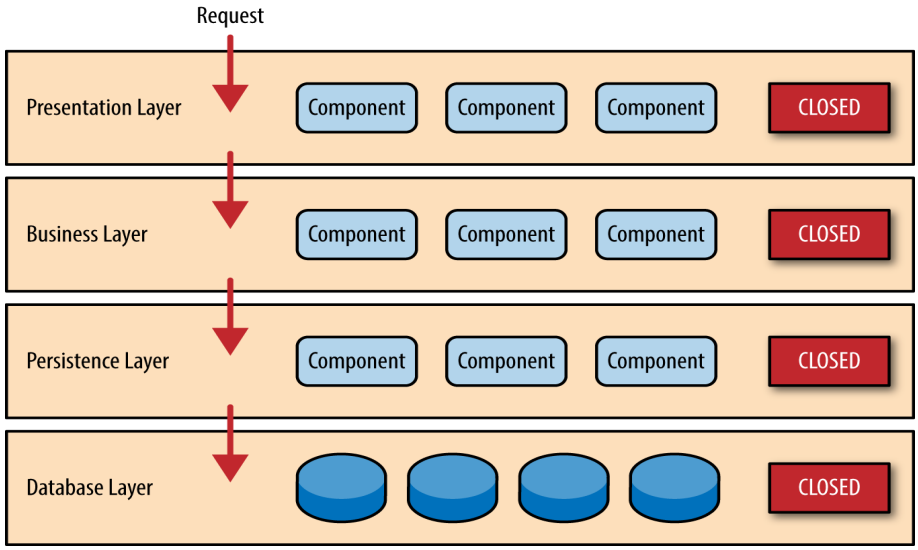
Kihilise arhitektuurimustri komponendid on jagatud horisontaalsetesse kihtidesse kus iga kiht täidab konkreetset rolli rakenduses. Kihilise arhitektuurimustri puhul ei määrata kindlaks kihtide arvu ega tüüpe, kuid enamus kihilisi arhitektuure koosneb neljast standardsest kihist: esitluskiht, äriloogika kiht, andmekiht ja andmebaasi kiht (Joonis 1) [7].




Joonis 1. Kihiline arhitektuur [7].

Igal kihil arhitektuuris on oma spetsiifiline roll ja vastutus rakenduses. Iga kiht arhitektuuris moodustab abstraktsiooni ülesannetest, mis tuleb täita konkreetse ärilise päringu täitmiseks. Komponendid kindlas kihis tegelevad ainult loogikaga, mis on seotud selle sama kihiga. Näiteks, komponendid esitluskihis tegelevad ainult esitlemise loogikaga, ning komponendid äriloogika kihis tegelevad ainult äriloogikaga [7].

Üks olulisi kontsepte kihilises arhitektuuris on suletud kiht. Suletud kiht tähendab seda, et päring peab liikuma kihtide vahel ülevalt alla lineaarselt, jätmata vahele ühtegi teist teekonnal olevat suletud kihti (Joonis 2) [7].

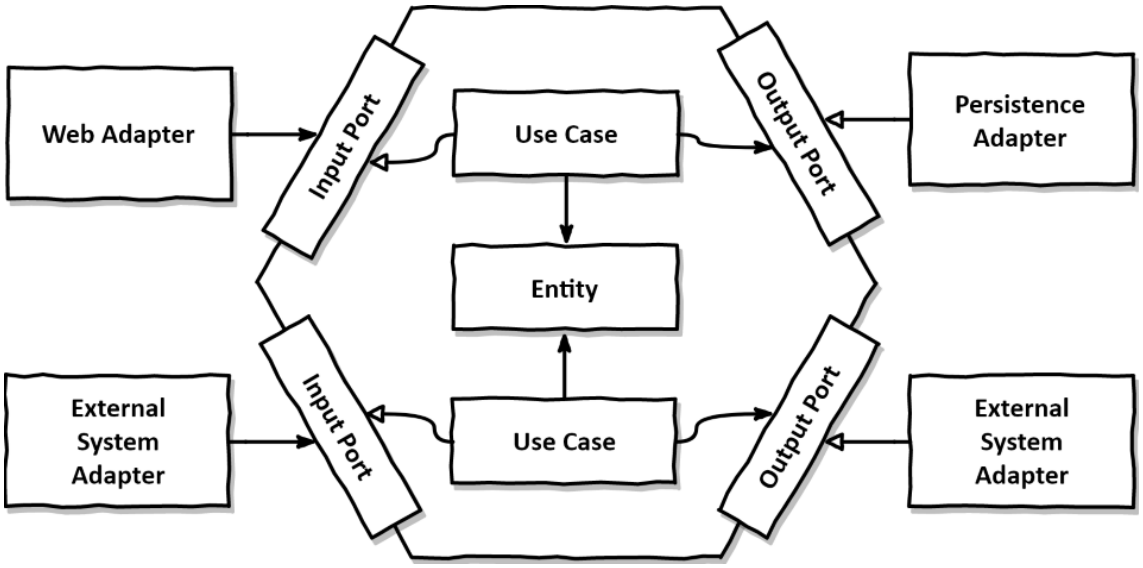



Joonis 2. Kihiline arhitektuur suletud kihtidega [7].

Avatud kiht tähendab seda, et kiht mis on märgitud avatuks, sellest võib mööduda ning minna sellest järgmise (alumise) kihini. Avatud kihi üheks kasutuskohaks on näiteks jagatud üldkasutatav teenus [7].

## 2.3 Heksagonaalne arhitektuur

Mõiste "Heksagonaalne arhitektuur" pärineb Alistair Cockburnilt ja on olnud kasutuses juba pikemat aega. See rakendab samu põhimõtteid, mida Robert C. Martin hiljem kirjeldas oma raamatus ***Clean Architecture***. Joonis 3 näitab, milline heksagonaalne arhitektuur võib välja näha. Rakenduse tuuma kujutatakse kuusnurgana, andes sellele arhitektuuristiilile tema nime. Kuusnurkne kuju samas ei oma arhitektuuriliselt mingit mõju ega tähendust [10].




Joonis 3. Heksagonaalse arhitektuuri illustreeriv kuusnurk [10].

Kuusnurga tsentraalne osa koosneb domeeniüksustest (*domain entity*) ja kasutusjuhtudest (*use case*), mis töötavad koos domeeniüksustega. Tuleb tähele panna, et kuusnurgas puuduvad väljaminevad sõltuvused, seega sõltuvuse reegel Robert Martini ***Clean Architecture***'st pädeb. Vastupidiselt, sõltuvused kuusnurgas on sissepoole [10].

Väljaspool kuusnurka resideeruvad erinevad adapterid, mis suhtlevad rakendusega. Vasakul pool kuusnurka asuvaid adaptereid nimetatakse juhtivateks adapteriteks, ning paremal pool asuvaid nimetatakse juhitavateks adapteriteks. Adaptereid nimetatakse

juhtivatakse seetõttu, et need adapterid "kutsuvad" rakenduse tuuma, erinevalt juhitavatest adapteritest, mida kutsub rakendus tuum ise [10].

Rakenduse tuuma ja adapterite vahelise suhtluse võimaldamiseks pakub rakenduse tuum konkreetseid porte. Juhtivate adapterite jaoks võib selline port olla liides, mis on rakenduse tuumas ühe kasutusjuhu klassi poolt rakendatud ja kutsutud adapteri poolt. Juhitava adapteri jaoks võib see olla liides, mis on rakenduse tuumaga seotud adapteri poolt rakendatud, mida rakenduse tuum omakorda kutsub [10].

*SOLID* printsiibid on osa olulistest alustaladest rakendamaks heksagonaalset arhitektuuri. Peamisteks *SOLID* printsiipideks heksagonaalses arhitektuuris on *Single Responsibility Principle* ja *Dependency Inversion Principle* [10].

Pöörates ümber sõltuvuste suunda saavutatakse olukord, kus domeeni koodil puuduvad välised sõltuvused, mis tähendab, et domeeni saab modelleerida vastavalt ärilistele vajadustele sõltumata välistest agentidest [10].

Heksagonaalse arhitektuuri omadusteks on testitavus ning sõltumatus raamistikest, kasutajaliidesest, andmebaasist ja muudest välistest agentidest [10]. Sõltumatus raamistikest tähendab, et näiteks soovi korral vahetada olemasolev *Postgres*'i andmebaas *Oracle*'i baasi vastu piisaks olemasoleva andmebaasi adapteri kustutamisest ja uue implementeerimisest - domeeni loogika ja muud kohad koodis jääksid sellest muutusest puutumatuks.

## 2.4 Heksagonaalne arhitektuur vs kihiline arhitektuur

Definitsiooni järgi on klassikalise kihilise arhitektuuri aluseks andmebaas. Esitluskiht sõltub äriloogika kihist, mis omakorda sõltub andmekihist ja seega andmebaasist. Eelnevast saab järeldada, et kõik ehitatakse andmekihi peale (*database-driven design*) [7] [10].

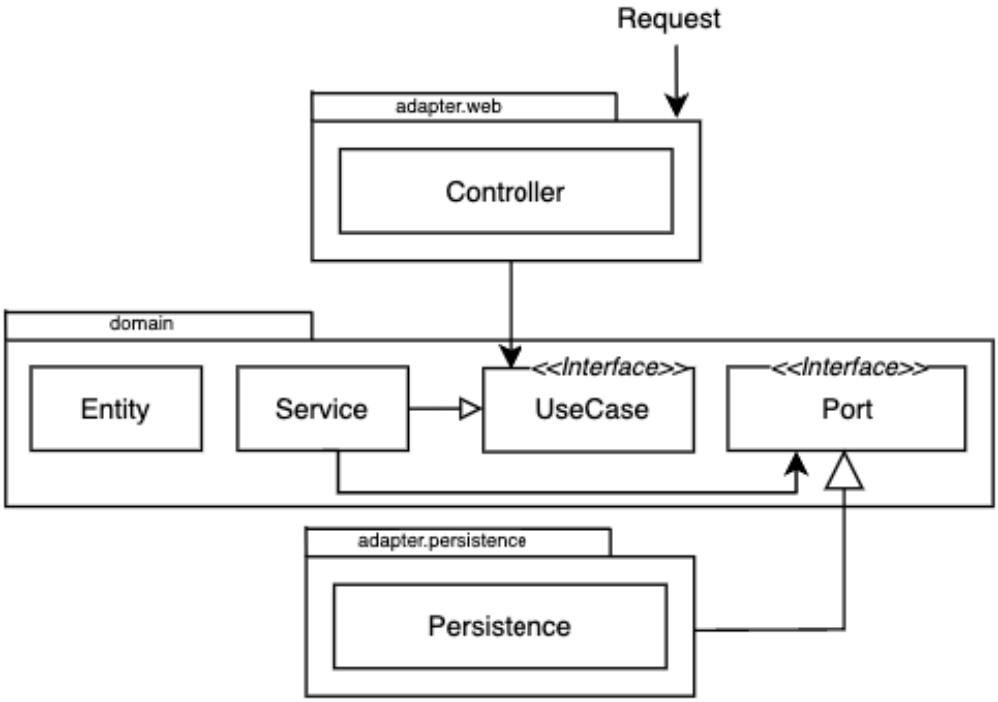
Andmebaasikeskne disain kaasab mitmeid probleeme, millest peamine on äriloogikast distantseerumine. Tavaliselt üritatakse luua mudeleid mis on juhitud ärivajadustest, lihtsustades seega kasutajate suhtlust nendega. Peamiselt püütakse modelleerida käitumist, mitte olekut. Kuigi olek on igas rakenduses oluline osa, on käitumine see, mis muudab olekut ja seega ajendab äri [10].

Heksagonaalne arhitektuur, erinevalt kihilisest arhitektuurist, ei sõltu andmebaasist ning põhineb domeenil ehk äriloogikal (*domain driven design*), mis tagab paindlikkuse ja avatuse muudatusteks. Siit saab tuua ühe suurima ning olulisema erinevuse – heksagonaalne arhitektuur on juhitud domeenist, samas kihiline arhitektuur on juhitud andmebaasi poolt [10].

Järgnevalt toob autor kahe arhitektuurimustri võrdluses välja erinevused andmebaasi aspektist. Märgin ära, et võrdluses kasutan varasemalt mainitud neljakihilist arhitektuuri. Näite aluseks on olukord, kus kasutaja läbi kasutajaliidese soovib näha oma konto andmeid.

Kihilises arhitektuuris kasutuskoha voolu (*flow*) visualiseerib hästi eelnevalt väljatoodud joonis (Joonis 2), kus kasutaja tehtud päring saab sisendi esitluskihist, kust see omakorda liigub edasi äriloogika kihti ning lõpuks andmebaasi kihti. Kuna kihilises arhitektuuris sõltub äriloogika kiht andmekihist siis järelikult äriloogika kiht on teadlik andmekihist ning selle mudelitest, läbi mille saab äriloogika kiht kasutada andmekihis defineeritud objekte.

Heksagonaalses arhitektuuris kasutuskoha voolu (*flow*) illustreerib Joonis 4. Jooniselt on näha, et sõltuvused on ümber pööratud (*DIP*), mille tagajärjel domeen ei sõltu andmebaasist. Sõltuvuste ümberpööramise tulemusel ei ole domeen teadlik andmebaasist ega saa seega kasutada seal defineerituid objekte. See tähendab, et mõlemas moodulis eksisteerivad neile vastavad mudelid ning moodulite vahelises suhtluses tuleb tõlgendada mudeleid ühelt kujult teisele.




Joonis 4. Heksagonaalse arhitektuuri andmevahetuse voo näide.

Eelnevast on näha kihilises arhitektuuris selgelt komponentide tihedat sidusust, tehes selle vähem agiilseks ning skaleeritavaks. Kihtide sõltuvus ja loogika ning mudelite kasutamine teistes kihtides päädib sellega, et tehes muudatuse ühes kihis tuleb teha vastavaid muudatusi teistes kihtides. Kuna heksagonaalses arhitektuuris on sõltuvused ümber pööratud ning puuduvad välised sõltuvused, siis muudatused ühes moodulis ei mõjuta teisi, ehk muudatusi tehes piisab ainult vajaliku spetsiifilise koodi muutmisest. Lisaks sõltumatus välistest teguritest ja tehnoloogiatest võimaldab heksagonaalses arhitektuuris vahetada välja andmebaasi, muutmata selleks ümbritsevat koodi.

# 3 Refaktoreerimine

Refaktoreerimine on protsess, mille käigus parandatakse olemasoleva koodi disaini muutes selle sisemist struktuuri, samal ajal jättes mõjutamata selle välimist käitumist [11].

Tarkvara evolutsiooni kontekstis kasutatakse restruktureerimist ja refaktoreerimist tarkvara kvaliteedi parandamiseks (näiteks laiendatavus, modulaarsus, taaskasutatavus, keerukus, hooldatavus, tõhusus) [12]. Refaktoreerimist ja restruktureerimist kasutatakse ka *reengineering* kontekstis, mis on uuritava süsteemi läbivaatamine ja muutmine uue vormi loomiseks ning selle uue vormi järgne rakendamine. Selle konteksti puhul on restruktureerimine vajalik pärand- või halvenenud koodi teisendamiseks rohkem modulaarsele või struktureeritumale kujule või isegi koodi migreerimiseks teisele programmeerimiskeelele või keeletüübile [12].

## 3.1 Refaktoreerimise protsess

Refaktoreerimise protsess koosneb mitmest erinevast tegevusest [12]:

1. Tuvasta, kus tarkvara peaks refaktoreerima [12].
2. Määratle, milliseid refaktoreerimise praktikaid peaks rakendama tuvastatud kohtadele [12].
3. Taga rakenduse käitumise säilimine [12].
4. Rakenda refaktoreerimist [12].
5. Hinda refaktoreerimise mõju tarkvara kvaliteedile (näiteks keerukus, arusaadavus, hooldatavus) või protsessile (näiteks produktiivsus, kulu, pingutus) [12].
6. Säilita järjepidevus refaktoreeritud koodi ja muude tarkvara artifaktide vahel (nagu näiteks dokumentatsioon, disain, dokumendid, nõuded, tehniline kirjeldus, testid, jms) [12].

## 3.2 Refaktoreerimise praktikad

Käesolevas punktis toob autor välja projektis enim kasutatud refaktoreerimise praktikad. Siinkohal on oluline täheldada, et refaktoreerimise praktikad ei piirdu selles alampunktis mainituga.

- Extract function [13]
- Extract variable [13]
- Inline function [13]
- Inline variable [13]
- Rename Variable [13]
- Change Function Declaration [13]
- Decompose Conditional [13]
- Replace Nested Conditional with Guard Clause [13]
- Consolidate Conditional Expression [13]
- Separate Query from Modifier [13]

## 3.3 Tehniline võlg

Tehnilise võla definitsioon Steve McConnelli järgi (Steve McConnell, 2013, Managing technical debt (slides), in Workshop on Managing Technical Debt): *''A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).*''

Tehnilise võla peamiseks põhjuseks peetakse tähtaegade survet [14]. Tehnilise võla alla võib lugeda järgmist [15], [14]:

- Arhitektuuriline võlg
- Struktuuriline võlg
- Testide võlg
- Dokumentatsiooni võlg
- Madal sisemine kvaliteet
- Koodi keerukus
- Koodi stiili rikkumised
- Koodi "haisud" (*code smells*)

# 4 Testimine

Valideerimine on protsess, mille eesmärk on suurendada kindlustunnet, et programm töötab nii, nagu on ette nähtud. Üldiselt hõlmab valideerimine testimist. Testimise soovitud tulemus on tagada, et programm vastab seatud spetsifikatsioonidele, kuid nagu Edsger Dijkstra märkis, võib testimine tõestada vigade olemasolu, kuid mitte nende puudumist. Siiski suurendab hoolikas testimine oluliselt kindlustunnet, et programm töötab ootuspäraselt [16].

Antud kirjatöö raames käsitletakse testidena üksusteste (*unit test*).

## 4.1 Üksustest

Üksustest on automatiseeritud test, mis:

- Kontrollib väikest osa koodist (*unit*) [17],
- Teeb seda kiiresti [17],
- Teed seda isolatsioonis (selle punkti kohta on erinevaid arvamusi) [17]

Javas on üksus tavaliselt klass ning isolatsioonis jooksutatavat testi valideeritakse võrreldes oodatud tulemust ja tegelikku tulemust. Üksustestid kutsuvad klassi meetodeid, et luua vaadeldavaid tulemusi mida kontrollitakse automaatselt [16].

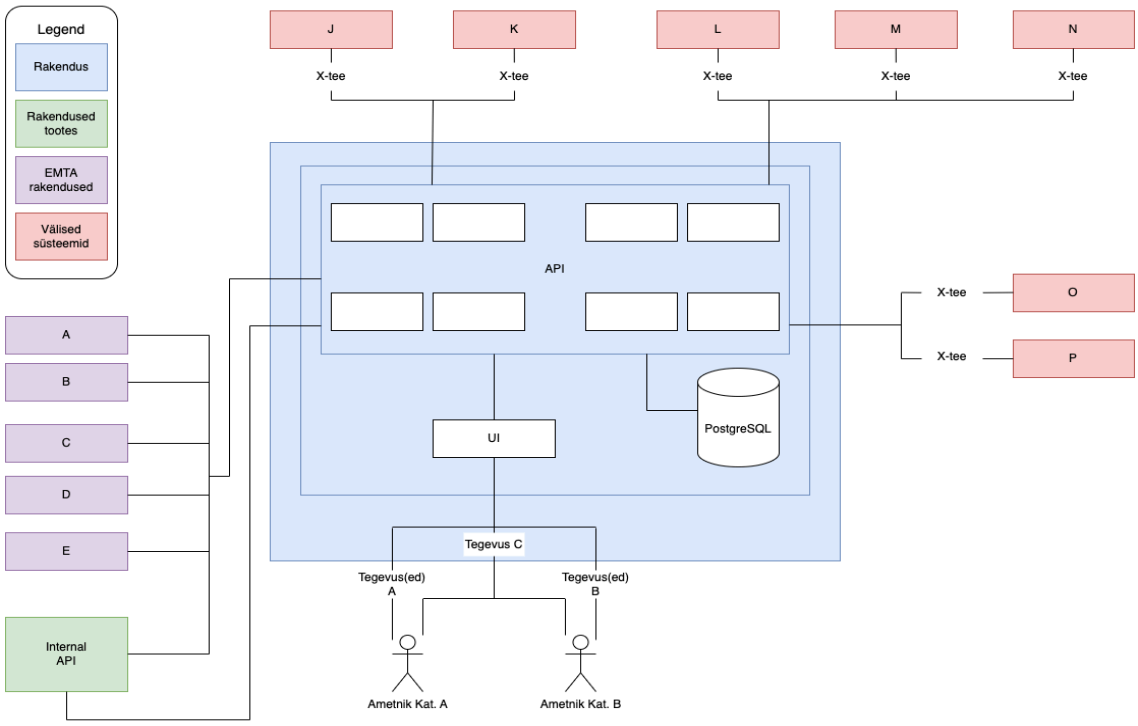
Käesolevas projektis kasutatakse üksustestimiseks *JUnit* [18] testimise raamistikku.

# 5 Arendus

Selles peatükis käsitletakse tehtud tööd nii arhitektuuri, refaktoreerimise kui testimise raames. Peatükid on sisu järgi loogiliselt järjestatud, tegelik arendusprotsess algas refaktoreerimisega ja testikatvuse tõstmisega.

## 5.1 Arhitektuur

Projekti arhitektuuriskeemi visualiseerib Joonis 5.




Joonis 5. Käesoleva töö koodibaasi arhitektuuriskeem.

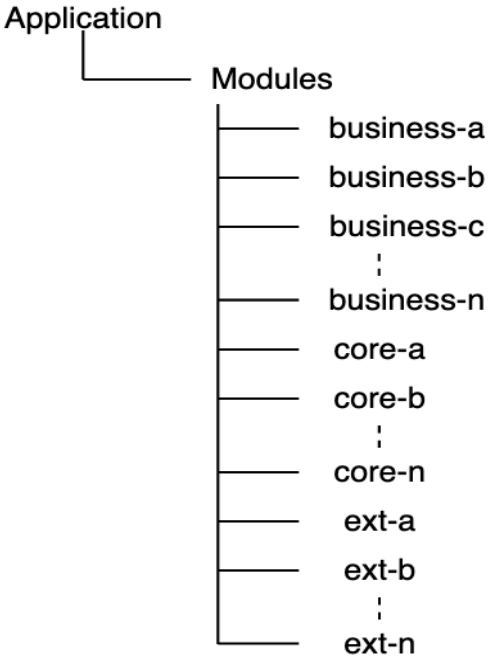
Sinine tsentraalne osa representeerib rakendust mille kohta antud lõputöö käib. Rakendusel on *Postgres* andmebaas ning kasutajaliides. Rakenduse peamisteks kasutajateks on kahte eritüüpi- ning erinevate õigustega ametnikke. Suhtlus väliste süsteemidega toimub üle X-tee ning EMTA- ja projektisiseselt kasutades REST ja SOAP protokolle. Siinkohal projektisiseselt tähendab seda, et projekt koosneb kahest erinevast koodibaasist, millel mõlemal on oma andmebaas ning ees- ja tagarakendus.

### 5.1.1 As-Was

Antud alampunktis käsitletakse esialgset päritud koodi.

Päritud kood oli struktureeritud moodulite põhiselt. Moodulid vastavad nimetamise skeemile kujul eesliide-tegevusvaldkond. Lisaks võib mooduleid veel kategoriseerida kolme suuremasse gruppi vastavalt nende eesliidetele, kus *business* eesliitega moodulid tegelevad äriloogikaga, *core* eesliitega tegelevad erinevate konfigureerimistega, ning *ext* eesliitega esindavad väliseid teenuseid.

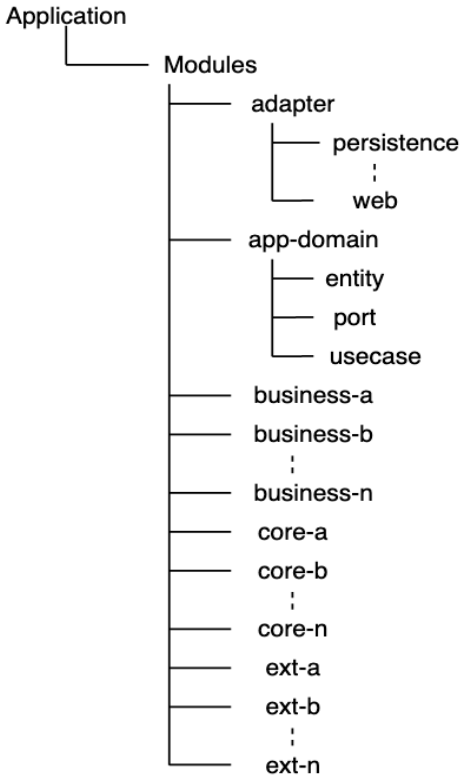
Moodulid sisaldavad kõike, mida neil läheb vaja – andmebaasi suhtlus, kontrollerid, andmeedastusobjektid jms. Ehk rakendus koosneb moodulitest, mille sisemine arhitektuur põhineb kihilisel arhitektuuril.




Joonis 6. As-Was koodibaasi organiseerimise struktuur.

### 5.1.2 As-Is

Kõrgema taseme ülevaatel võib tunduda, et võrdluses algse arhitektuuriga ei ole väga midagi muutunud. Põhjus, miks moodulite ja pakkide struktuur ei ole suuremas vastavuses heksagonaalse arhitektuuriga on pigem lihtne – olemasolevas pärandisüsteemis on suur osa koodist niivõrd tihedalt seotud, et nende täielikku restruktureerimist antud projekti faasis ei olnud võimalik rakendada tulenevalt projekti tähtaegadest.

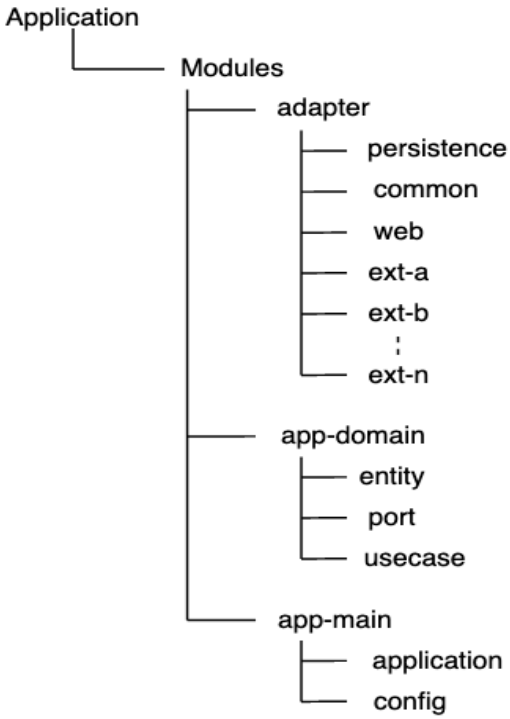



Joonis 7. As-Is koodibaasi (ehk hetkeseis) organiseerimise struktuur.

Antud struktuuris esindab üks *core* moodul rakenduse *main* moodulit. Peamiseks modifikatsiooniks koodi organiseerimise struktuuri poolelt võrreldes klassikalise heksagonaalse arhitektuuriga on juhtivate- ja juhitavate adapterite ning portide mitte eristamine. Põhjus sellise struktuuri eelistamiseks on fakt, et projekt ei ole piisavalt suur, et anda lisaväärtust mainitud pakkide loomiseks ning pigem süvendab põhjendamatult pakettide harusid.

### 5.1.3 To-Be

Koodi lõplik organiseerimise struktuur peaks vastama suuresti heksagonaalse arhitektuuri tavale, ning antud töö raames koosneksid kolmest peamisest moodulist: adapter, app-domain ja app-main.




Joonis 8. To-Be (ehk tuleviku) koodibaasi organiseerimise struktuur.

Võrreldes *as-is* seisuga refaktoreeritakse ning liigutatakse heksagonaalse struktuurivälised moodulid vastavatesse heksagonaalse organisatsiooni moodulitesse ning luuakse uus moodul *app-main*, mis hõlmab endas rakenduse *main* klassi ja muid vajalikke konfigureerimise faile, mis omakorda on jagatud *app-main* alla *config* paketti.

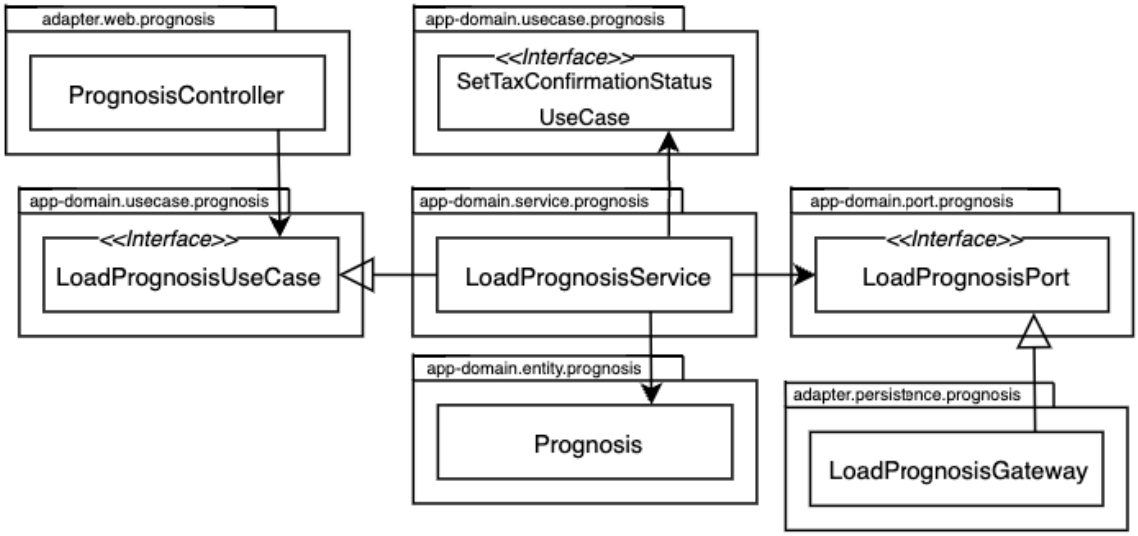
### 5.1.4 Implementatsioon

Peatükis tuuakse näiteks implementatsioon kasutusjuhtumist, mille eesmärk on leida isikule vastav prognoositav maks.

Domeeni mudelite poolelt langes eelistuseks õhuke (*anemic*) domeeni mudel [10], et hoida domeeni mudelid võimalikult kompaktse ning puhtana. Õhukese domeeni mudeli eelistamine tähendab seda, et äriloogika implementeeritakse *use case* sees.

*PrognosisController* kutsub *LoadPrognosisUseCase* vastavat funktsionaalsust, kui antud otspunkti vastu tehakse päring. *Service* implementeerib *LoadPrognosisUseCase* liidest ja kutsub *SetTaxConfirmationUseCase* liidest ning sellele lisaks väljamineva pordi liidest *LoadPrognosisPort*, mida implementeerib andmebaasi adapteris *LoadPrognosisGateway*, et saada andmebaasist vajalikud isiku kohta käivad maksu andmed ning neid töödelda vastavalt äriloogikale. Üleüldiselt on projektis kasutusel

standard, mille järgi pannakse kasutusjuhtude meetodi nimeks "*execute*". Joonis 9 annab graafilisema ülevaate kirjeldatud olulisematest komponentides.




Joonis 9. Prognoositava maksu leidmise skeem.

Representatsioon koodina on kujutatud järgnevatel joonistel Joonis 10-15.

Prognoosi kontrolleris kutsutakse *LoadPrognosisUseCase* liidese *execute* meetodit, mis konverteeritakse *page* objektiks. Kontrollerit illustreerib Joonis 10Joonis 10.

```
package ee.aa.bb.app.web.prognosis;

import ...

@InternalApi
@RequiredArgsConstructor
class PrognosisController {

  private final LoadPrognosisUseCase loadPrognosisUseCase;

  @GetMapping("/v1/prognosis")
  Page<PrognosisDto> getPrognosis(PrognosisFilterDto filter,
                                  Pageable pageable) {
    var request = PageableUtil.toRequest(pageable,
                                         filter.toRequest());
    var response = loadPrognosisUseCase.execute(request);

    return PageableUtil.convert(response, PrognosisDto::new);
  }
}
```

Joonis 10. Prognoositava maksu leidmise kontroller.

*Use case* on siinkohal (ning seda enamasti) väga lihtne. Antud näite puhul sisaldab see ühte meetodit, mille sisendiks on *port*-liidese spetsiifiline klass, mis hoiab endas päringust tulnud parameetreid. *Use case*'i illustreerib Joonis 11.

```
package ee.aa.bb.app.app.domain.usecase.prognosis;

import ...

public interface LoadPrognosisUseCase {
  PagedResponse<Prognosis> execute(LoadPrognosisPort.Request request);
}
```

Joonis 11. Prognoositava maksu leidmise use case.

*Service* implementeerib varem defineeritud *use case* liidest - see on koht, kus rakendatakse äriloogikat. Antud näites ning ka teistes lihtsamates kasutuskohtades (näiteks kasutuskohad, mille ainsaks eesmärgiks on baasist andmete pärimine) on ka *service* suhteliselt hõre. *Service'it* illustreerib Joonis 12.

```
package ee.aa.bb.app.domain.usecase.prognosis;

import ...

@Service
@RequiredArgsConstructor
class LoadPrognosisService implements LoadPrognosisUseCase {

  private final LoadPrognosisPort loadPrognosisPort;
  private final SetTaxConfirmationStatusUseCase
      setTaxConfirmationStatusUseCase;

  @Override
  public PagedResponse<Prognosis> execute(LoadPrognosisPort.Request
                                          request) {
    var pagedPrognosis = loadPrognosisPort.fetchForClient(request);

    setTaxConfirmationStatusUseCase
        .execute(pagedPrognosis.getEntities());

    return pagedPrognosis;
  }
}
```

Joonis 12. Prognoositava maksu leidmise service.

Prognoosi leidmise *port* sisaldab kliendile makstava prognoositava maksu leidmise meetodit *fetchForClient* ning ühte sisemist klassi. Sisemised klassid (*Request* ja *Response*) on *use case* ja *port*-liidestele vastavad andmemudelid. *Request* klassi kasutatakse peamiselt siis kui päringuga tuleb mitmeid parameetreid. *Response* on sisuliselt spetsiifiline, kitsa kasutuskohaga tagastatav mudel, ning kuna see ei ole taaskasutatav siis deklareeritakse liidese sees. *Port*'i illustreerib Joonis 13.

```
package ee.aa.bb.app.domain.port.prognosis;

import ...

public interface LoadPrognosisPort {

  PagedResponse<Prognosis> fetchForClient(Request request);

  @Getter
  @SuperBuilder
  class Request extends PagedRequest {
    private final Long taxPayerPersonId;
    private final List<Long> specialIds;
    private final List<Long> unitIds;
    private final Integer taxYear;
    private final String joinedPersonCode;
  }
}
```

Joonis 13. Prognoosi leidmise port.

Mudel, nagu varasemalt mainitud, on hõre (*anemic*), ehk koosneb ainult vajalikest väljadest (Joonis 14).

```
package ee.aa.bb.app.domain.entity.prognosis;

@Builder
@Getter
@Setter
public class Prognosis {
  private final Long id;
  private final Long taxUseId;
  private final String unitName;
  private final String unitCode;
  private final Long unitId;
  private final String unitDataSource;
  private final String specialName;
  private final Long specialId;
  private final String taxPayerPersonCode;
  private final String taxPayerPersonName;
  private final Long taxPayerPersonId;
  private final BigDecimal predictableTaxAmount;
  private final Integer year;
  private boolean taxStatus;
}
```

Joonis 14. Prognoositava maksu domeeni mudel.

Andmebaasi adapteris implementeerib *LoadPrognosisGateway LoadPrognosisPort* liidest, mille käigus päritakse isikule vastavad prognoositavad maksu summad (Joonis 15).

```
package ee.aa.bb.app.persistence.prognosis;

import ...
import org.jooq.DSLContext;

@Component
@RequiredArgsConstructor
class LoadPrognosisGateway implements LoadPrognosisPort {
  // constants omitted for brevity

  private final DSLContext dsl;

  @Override
  public PagedResponse<Prognosis> fetchForClient(
                          LoadPrognosisPort.Request request) {
    PageRequest pageable = FilteredQueryUtil.toPageRequest(request);

    var order = getSortOrder(pageable);
    var orderProperty = getOrderPropertyAsValidOrderString(order);

    if (!CLIENT_SORT_FILTER_FIELDS.containsKey(orderProperty)) {
      throw new ApiException(400);
    }

    var condition = whereCondition(request);

    return getPrognosisPagedResponse(pageable, order,
                                     orderProperty, condition);
  }
  // helper methods omitted
}
```

Joonis 15. Prognoositava maksu andmebaasi adapteri implementatsioon.

Vastavus *SOLID* printsiipidele on tagatud. *DIP* on rahuldatud, kuna kõrgtaseme moodulid ja madalataseme moodulid mõlemad sõltuvad abstraktsioonidest, mitte konkreetsetest klassidest. *OCP* on rahuldatud, eelistades kompositsiooni pärimise asemel. *LSP* siinkohal ei rakendu, kuna ühelgi klassil ei ole alamklasse. *SRP* puhul võib vaielda, kas antud koodi raames on seda rahuldatud, kuna *fetchForClient* meetodis tehakse väiksemat sorti valideerimine. Samuti on tagatud *ISP* rahuldamine, kuna liideseid on mitmeid ning need on sisaldanud ainult ühte meetodit.

Ümber kirjutamise protsess kihiliselt arhitektuurilt heksagonaalsele arhitektuurile seisnes olemasolevast koodist loogiliste komponentide eraldamisega ning nende

restruktureerimine vastavalt käimasoleva implementatsiooni peatükile eraldiseisvateks komponentideks, mis asetsevad projekti struktuuris samuti vastavates pakettides.

### 5.1.5 Modifikatsioonid

Projektis ei ole lähenetud arhitektuurile nii, et kood peaks olema alati 1:1 vastavuses heksagonaalse arhitektuuri põhiprintsiipidega. Peamiseks modifikatsiooniks või *shortcut*'iks on *wrapper* klasside eemaldamine. Tähendab: kohtades, kus äriloogikat ei rakendata, jäetakse ära vastavad *use case* ja *service* ning kutsutakse otse *port*'i. Olukorda, kus tekivad *wrapper* klassid, illustreerivad Joonis 16 ja Joonis 17.

```
package ee.aa.bb.app.domain.usecase.prognosis;

public interface LoadPrognosisTotalTaxUseCase {

  BigDecimal execute(Long taxPayerPersonId,
                     @Nullable String specialCode, Integer taxYear);
}
```

Joonis 16. Use case mis on sisuliselt wrapper klass.

```
package ee.aa.bb.app.domain.usecase.prognosis;

@Service
@RequiredArgsConstructor
class LoadPrognosisTotalTaxService implements
LoadPrognosisTotalTaxUseCase {

  private final LoadPrognosisTotalTaxPort port;

  @Override
  public BigDecimal execute(Long taxPayerPersonId, String specialCode,
                            Integer taxYear) {
    return port.fetchTotalTax(taxPayerPersonId, specialCode, taxYear);
  }
}
```

Joonis 17. Service mis on sisuliselt wrapper klass.

Selliste *wrapper* klasside vältimiseks kutsume välja otse vajalikku sihtmärki, milleks antud näite puhul on *LoadPrognosisTotalTaxPort* liides meetodiga *fetchTotalTax*.

## 5.2 Refaktoreerimine

Refaktoreerimise protsess algas projektis koodianalüüsi tööriistade kasutamisega, saamaks ülevaate esmaselt tähelepanu nõudvatest kohtadest. Analüüsimiseks kasutame järgnevaid tööriistu: SonarQube [4], PMD [2], SpotBugs [3] ja CheckStyle [5]. Tööriistade poolt genereeritud raportite pealt jagati arendajate vahel töö, et kiirendada

protsessi. Lisaks eelmainitule põhineb refaktoreerimine Robert C. Martini ***Clean Code*** raamatust pärinevatele põhimõtetele ning SOLID printsiipidele.

### 5.2.1 As-Was

Projekti alguses tulenenud vajadus refaktoreerimise järele kirjeldab algse koodibaasi seisu ehk parimalt – halb. Algne koodibaas sisaldas suures hulgas koodianalüüsi tööriistade poolt tuvastatavaid vigu, ei vastanud nii *clean code* põhimõtetele kui ka *SOLID* printsiipidele.

*PMD* vigu tuvastati 27, *checkStyle* vigu 948 ja *spotBugs* vigu 23.

### 5.2.2 As-Is

Hetkeseisuga on likvideeritud kõik olulised vead peale *god class*'ide, mida on rakendus seadistatud ignoreerima. Lisaks ignoreeritakse üksikuid *checkStyle* poolt tuvastatavaid vähem olulisi stiilivigu. Tehtud töö tulemusena ei leidu enam projektis koodianalüüsi tööriistade poolt leitavaid vigu, koodibaas vastab *clean code* põhimõtetele ning heksagonaalsel arhitektuuril olev kood vastab *SOLID* printsiipidele.

Lisaks, *Gitlab* keskkonda on loodud *pipeline*, mis automaatselt käivitab testid ning *PMD, checkStyle* ja *SonarQube* analüüsid. Vigade leidmisel on koodi *merge* takistatud ning arendajale saadetakse vastav email. *Pipeline* läbimiseks peavad kõik protsessid lõpetama edukalt, mis on ka *merge* eelduseks.

### 5.2.3 To-Be

Oodatava lõpptulemusena ei tohiks koodibaas sisaldada ühtegi koodianalüüsi tööriistade poolt leitavat viga. Lisaks automaatselt leitavatele vigadele peab kood vastama ***Clean Code*** põhimõtetele ning *SOLID* printsiipidele. Antud projekti raames sellise tulemuseni ei jõuta peamiselt põhjusel, et koodibaasis leidub mitmeid *god class*'e mille refaktoreerimine ei mahu projekti ajapiirangutesse. Et analüüsi tööriistad ei annaks eelmainitud viga on tööriistad seadistatud selliselt, et neid vigu ignoreeritaks. Peamine põhjus selliste vigade ignoreerimiseks, mida ei parandata, on *pipeline* seadistus, mis automaatselt kontrollib mainitud vigu ja vigade olemasolu korral on koodi *merge* protsess blokeeritud.

### 5.2.4 Implementatsioon

Antud peatükis tuuakse näited projektis tehtud erinevatest refaktoreerimistest ja nende kasust.

Esimese näitena toob autor välja kasutuskoha, kus arendaja eelnevalt arvas, et kõik töötab korrektselt ja kiirelt, aga kuna arendaja katsetas loogikat oma lokaalse andmebaasiga, milles olid ainult üksikud kirjed, siis tegelik mõju jõudlusele jäi peitu. Jõudluse probleem sai ilmsiks kui rakendus ühendada testandmebaasiga, milles on miljoneid kirjeid, ning sealt tulenes tõdemus, et arendatud funktsionaalsus sai reaalse andmevooga *timeout*'i.

```
@Override
public Page<PrognosisDto> getPrognosis(PrognosisFilterDto filter,
                                       Pageable pageable) {
  Page<PrognosisDto> page =
       internalService.getPrognosis(filter, pageable);
  page.stream().forEach(this::setPreviousYearTaxAmount);
  return page;
}
```

Joonis 18. Algne meetod mille sisemine funktsionaalsus tekitab *timeout*'i.

Põhjus, miks antud funktsionaalsus viskab *timeout* erindit, tuleneb sellest, et itereerides kutsutakse *setPreviousYearTaxAmount()* meetodit, mille sees tehakse iga kord uus baasipäring. Lahendus probleemile oli lihtne: kuna kõik vajalikud andmed masspäringuks on olemas ja kirjete arv lehel on kõigest 10-100, siis tehakse kogu eelnev itereerimine ühe masspäringuga, mille järel itereeritakse üle esialgsete andmete, lisades neile masspäringust leitud vastav kirje.

```
@Override
public Page<PrognosisDto> getPrognosis(PrognosisFilterDto filter,
                                       Pageable pageable) {
  Page<PrognosisDto> page =
       internalService.getPrognosis(filter, pageable);

  if (!page.isEmpty()) {
     processPrognosisService
         .linkEachWithCorrespondingUnit(page.getContent());
  }

  return page;
}
```

Joonis 19. *Timeout* erindi saanud ülemine meetod pärast refaktoreerimist.

Algset meetodit palju ei pidanud muutma. Lisati andmete eksisteerimise kontroll ning *setPreviousYearTax* meetod asendati uue loodud *linkEachWithCorrespondingUnit* meetodiga.

```
@Override
public void setPreviousYearTaxAmount(List<PrognosisDto>
                                     prognosis) {
  var personIds = prognosis.stream
      .map(PrognosisDto::getPersonId)
      .distinct().toList();
  var codes = prognosis.stream()
      .map(PrognosisDto::getUnitCode).toList();
  Var taxYear = prognoosis.stream.findFirst()
      .map(PrognosisDto::getTaxYear)
      .orElseThrow(TaxYearMissingException::new);
  var units = unitService.getTaxAmount(personIds, codes, taxYear - 1);

  for (PrognosisDto dto : prognosis) {
    units.stream().filter(tu ->
        tu.getCode().equals(dto.getUnitCode()))
        .findFirst()
        .ifPresent(result -> {
          dto.setPreviousYearTaxAmount(result.getTaxAmount());
          dto.setUnitId(result.getId());
        });
  }
}
```

Joonis 20. Meetod mis teeb masspäringu ja seab vastavusse korrektsed andmed.

Refaktoreerimisega sai täidetud eesmärk parandada *timeout* tekkimine. Ümber tehtud lahenduse jõudlus on selgelt parem lõpetades protsessi alla 500ms. Üks viga uues loodud meetodis siiski leidub; nimelt meetodi nimi ei kirjelda täpselt, mida tahetakse saavutada, sest lisaks maksule seatakse ka üks Id. Korrektsem meetodi nimi oleks *setUnitIdAndPreviousYearTaxAmount*. Antud refaktoreerimise eesmärk oli puhtalt jõudluse parandamine kui mitte öelda korda tegemine.

Järgnev näide (Joonis 21) demonstreerib *Replace Nested Conditional with Guard Clause, Extract method* ja *Decompose conditional* refaktoreerimise võtete rakendamist *updateUnits* meetodi näitel.

```
protected void updateUnits(Share1 section1, SpecialEntity
                                            special) {
  // variables omitted for brevity
  updateSharesBySection1(section1, immovable);

  List<Unit> units = section1.getUnits()
      .getValue().getSelectedUnit();
  for (Unit cu : units) {
    Optional<RegisteredUnitEntity> regOpt =
        special.getRegisteredUnits().stream().filter(reg ->
                isSameUnit(cu.getAttribute().getValue(), reg))
            .findAny();
    var taxYear = DateUtil.getTaxYear(closingDate);

    if (ricuOpt.isPresent()) {
      if (Objects.nonNull(closingDate)) {
        if (taxYear < special.getYear()) {
          special.getSpecialSubEntities().remove(ricuOpt.get());
        } else {
          ricuOpt.get().setActiveTo(closingDate);
        }
      }
    } else {
      if (closingDate == null || taxYear >= immovable.getYear()) {
        addUnit(cu, immovable, changeDate, closingDate);
      }
    }
  }
}
```

Joonis 21. Algne *updateUnits* meetod.

Eelnevas koodis hakkab esimesena silma mitmetasemeline *if/else* plokk. Lisaks eelmainitule on kogu *if/else* plokk veel omakorda *for*-tsükli sees. Refaktoreerimise eemärk siinkohal on tingimuslausete ploki suuremas mahus eemaldamine ning lihtsustamine, vähendades meetodi keerukust ja tehes seda arendajale paremini loetavaks.

Joonis 22 demonstreerib refaktoreeritud *updateUnits* meetodit.

```
protected void updateUnits(Share1 section1, SpecialEntity
                                                special) {
  // variables omitted for brevity
  updateSharesBySection1(section1, immovable);

  List<Unit> units = section1.getUnits()
                    .getValue().getSelectedUnit();
  for (Unit cu : units) {
    Optional<RegisteredUnitEntity> regOpt =
      special.getRegisteredUnits().stream().filter(reg ->
         isSameUnit(cu.getAttribute().getValue(), reg))
         .findAny();
    var taxYear = DateUtil.getTaxYear(closingDate);

    if (regOpt.isPresent()) {
      modifyRegisteredEntity (immovable, closingDate, ricuOpt.get());
    } else if (isSpecialActive(special, closingDate)) {
      addUnit(cu, special, changeDate, closingDate);
    }
  }
}

private void modifyRegisteredEntity(SpecialEntity special,
                                    LocalDateTime closingDate,
                                    SpecialSubEntity ricu) {
  if (!Objects.nonNull(closingDate)) {
    ricu.setActiveTo(null);
    return;
  }

  if (DateUtil.getTaxYear(closingDate) < special.getYear()) {
    var ricuList = new ArrayList<>(special .getSpecialSubEntities());
    ricuList.remove(ricu);
    special.setSpecialSubEntities(ricuList);
    entityManager.flush();
  } else {
    ricu.setActiveTo(closingDate);
  }
}

private static boolean isSpecialActive(SpecialEntity special,
                                       LocalDateTime closingDate) {
  return closingDate == null ||
      DateUtil.getTaxYear(closingDate) >= special.getTaxYear();
}
```

Joonis 22. Refaktoreeritud *updateUnits* meetod.

Originaalsest meetodist eraldati kogu sisemine tingimuslausete plokk uueks meetodiks (*Extract method*). Uue meetodi alguses on rakendatud *Replace Nested Conditional with Guard Clause* ehk on kasutusele võetud meetodi alguses *guard clause*, ning esialgsest tingimuslausete plokist on alles jäänud *guard clause* ja üks *if/else* tingimuslause. Teine

loodud abistav meetod *isSpecialActive* rakendab *Decompose conditional* refaktoreerimise võtet, mille eesmärk on tingimuslause keerukamad tingimused tõsta eraldi meetodisse.

Rakendatud refaktoreerimised selgelt vähendasid meetodi keerukust ning oluliselt parandasid koodi loetavust. Lisaks vastavad refaktoreerimise järgsed meetodid SRP printsiibile.

### 5.2.5 Ajakulu ja pingutus

Kogu projekti vältel (*as-is* seisuga) on refaktoreerimise ja tehnilise võla alla logitud 867h, millest 92h on panustatud töö autori poolt. Tuleb mainida, et kogu mainitud aja alla kuulub ka projekti algusfaas, kus oli omajagu tegemist projekti ülesseadmisega, samuti on selle aja sees arvestatud serverite hooldamine jmt infrastruktuuri korras hoidmine, parandamine. Refaktoreerimise käigus sai uuendatud süsteemi sõltuvusi ning tehnoloogiaid, kasutamaks kõige uuemaid versioone.

Refaktoreerimise protsessi raskendasid asjaolud, et tegevusega alustati kohe projekti algul ning seda teostavatele arendajatele oli see uueks projektiks, mis tähendab, et sellel hetkel olid arendajate domeeni teadmised puudulikud. Peamine raskendav aspekt seisnes siiski kehval ja keerukal koodil, tundmatutes lühendites, madalas testikatvuses ja koodis eesti ning inglise keele segamini kasutamises. Suurte refaktoreeritavate tükkide puhul tekkis veel üks negatiivne kõrvalmõju: nimelt suured *merge request*'id, millele koodiülevaatuse tegemine võtab palju aega ning pingutust. Autoripoolne suurim *merge request* sisaldas 2861 lisatud rida ja 2653 kustutatud rida. Antud numbrid on võetud *Gitlab'*ist *Changes* sektsioonist.

## 5.3 Testimine

Testikatvuse tõstmise protsess algas projekti algul ning toimus paralleelselt refaktoreerimisega. Edasises arenduses toimus testimine arendustükkide põhiselt. Testikatvuse all on mõeldud üksustestide katvust, kuna hetkel ei ole suudetud seadistada tööriistu selliselt, et need võtaks arvesse integratsiooniteste.

### 5.3.1 As-Was

Algse koodibaasi testikatvus oli väga madal, kõigest 20.9% ning üksusteste oli kokku 615. Peale madala testikatvuse olid mitmed integratsioonitestid katki ning nende jooksutamiseks kasutatav tehnoloogia valesti seadistatud.

### 5.3.2 As-Is

Projekti alguses testikatvuse tõstmise protsess lõpetati kui saavutati 60% piir, täpsemalt oli sel hetkel testikatvuseks 61.3%. Hetkeseisuga on projekti testikatvus 67.3% ning üksusteste on projektis 2260. Testikatvuse protsent siinkohal ei anna väga täpset ülevaadet, kuna katvuse sisse ei arvestata integratsiooniteste, mida projektis realiseeritakse igas arendustükis, kus on suhtlus andmebaasiga. Integratsiooniteste on projektis 90. Üksustestide arv on seega kasvanud 615-lt 2260-le.

### 5.3.3 To-Be

Tuleviku üheks eesmärkideks on seadistada tööriistad arvestama integratsiooniteste, mis annaks parema ülevaate tegelikust koodikatvuse tasemest. Projektile seatud testikatvuse nõuet peab jätkuvalt tagama ning kõrgem testikatvus on tavaliselt parem, aga mitte alati.

### 5.3.4 Heksagonaalne vs kihiline

Kahe arhitektuurimusti vahel on näha erinevusi testitavuses. Heksagonaalse arhitektuuri komponendid on olnud selgelt paremini testitavad, üheks selle põhjuseks on antud arhitektuuri *SOLID* alustalad nagu *SRP* ja *DIP*. Peamiseks toetavaks printsiibiks loeks siinkohal *SRP*'d, sest testida meetodeid, mis teevad ainult ühte asja, on pigem lihtne, ning samal ajal motiveerib see arendajaid rohkem teste kirjutama. Lisaks nõuavad kihilise arhitektuuri komponentide testid rohkem andmete ja teenuste simuleerimist (testi duubleid), selle väite tõestamiseks statistika puudub ning ei saa kindlalt öelda, et vahe tuleneb puhtalt arhitektuurilistest erinevustest, kuna rakenduses ei ole korrektselt rakendatud kihilist arhitektuuri.

### 5.3.5 Ajakulu ja pingutus

Selles sektsioonis käsitletakse testimise protsessi all perioodi, mis oli projekti alguses pühendatud testikatvuse tõstmiseks 60 protsendini. Aega kulus selleks tiimil 437h millest töö autori panus oli 93h.

Testide kirjutamist takistasid samad asjaolud mis on kirjeldatud peatükis 5.2.5. Testide kirjutamise protsessis hakkasid ilmuma arhitektuurilised vead, mis omakorda raskendasid testide kirjutamist ja äriloogikast arusaamist. Peamiselt ilmnesid SRP printsiibi rikkumised ja erinevad kihtide vahelisest läbipõimumisest ja sõltumisest tulenevad aspektid. Eelnevatest probleemidest tulenevalt tuli paljude testide tegeliku äriloogikani jõudmiseks simuleerida ebavajalikult suur osa rakendusest, mis kaotab üksustestide eesmärgi. Ja veel - kuna meetodid, mida testitakse, teevad mitmeid asju, tuli luua mitmeid teste antud meetodi valideerimiseks, ning igas testis tuli rakendada simuleerimisi, mis puhtalt visuaalselt teevad testid palju pikemaks, raskemini loetavaks, keerukamaks ning vähem arusaadavaks.

# 6 Ülevaade tehtud tööst ning plaanid tulevikuks

Refaktoreerimise ja testikatvuse tõstmise protsess projekti alguses oli raskendatud ning ajakulukas, kuid tagasi vaadates kindlasti vajalik. Refaktoreerimise tulemusena paranes arendajate produktiivsus (subjektiivne), rahulolu ning koodist arusaamise tase. Pärandisüsteemide puhul teab tulevikus arvestada, et refaktoreerimine võib võtta oodatust rohkem aega ja pikemad perioodid, kus arendajad peavad ainult refaktoreerima ja teste kirjutama, on demotiveerivad.

Pärandisüsteemis kihiliselt arhitektuurilt heksagonaalsele üleminek võiks teoorias olla jõukohane enamikele arendustiimidele. Antud projekti raames oli see selgelt raskendatud tulenevalt ebakorrektselt struktureeritud koodist, mis tegi loogiliselt kokku käivate koodijuppide ümberkirjutamise heksagonaalsele arhitektuurile sisuliselt võimatuks. Põhjus eelnevale väitele on mainitud loogika läbipõimumine teiste kihtidega ja tihedad sõltuvused, mis tähendab, et väiksemategi koodiosade eraldamine nõuab suuresti loogika ümber kirjutamist olemasolevas moodulis. Kasutuskohtades, kus kihilist arhitektuuri oli rakendatud korrektselt, oli üleminek heksagonaalsele arhitektuurile muretu ning pigem otsejooneline. Samuti seab heksagonaalne arhitektuurimuster rangemad piirid koodi struktureerimisele, mis teeb reeglite rikkumise raskemaks ning lihtsamini avastatavaks, mida enamus arendajaid, kes selle arhitektuurimustriga on kokku puutunud, suudavad kergemini tuvastada.

Nii kihilise kui ka heksagonaalse arhitektuuri eesmärk on luua tarkvara, mis on loogiline, struktureeritud ning kergesti hooldatav. Üleminek kihiliselt arhitektuurilt heksagonaalsele arhitektuurile nõuab sageli radikaalseid muudatusi koodibaasis, eriti kui olemasolev kood on halvasti struktureeritud ja kirjutatud või tihedalt seotud erinevate kihtidega. Heksagonaalse arhitektuuri kasutusele võtmise peamine eelis seisneb selles, et see võimaldab paremini eraldada tarkvara domeeni tehnoloogiatest, loogikast ja sellest, kuidas seda domeeni välismaailmale esitletakse.

Kui olemasolev kood on juba hästi struktureeritud, kirjutatud vastavalt parimatele arendustavadele ja kihid arhitektuuris on selgesti eristatavad, võib olla võimalik osalise

koodi ümberkirjutamata jätmine. Siiski sageli on parem viia läbi põhjalik refaktoreerimine, et tagada uues arhitektuuris parem hooldatavus ning laiendamise võimekus.

On oluline meeles pidada, et refaktoreerimine võib olla ajamahukas protsess, mis kipub nõudma märkimisväärset investeeringut aja ja ressursside osas. Seetõttu on oluline otsust enne refaktoreerimist põhjalikult kaaluda erinevaid võimalusi, et tagada uude arhitektuuri ülemineku kulu- ja üleüldine tõhusus.

Heksagonaalne arhitektuur on hea valik suuremate ja keerukamate tarkvarasüsteemide jaoks, eriti nende jaoks, kus on palju väliseid sõltuvusi või kus on vaja kõrgemat modulaarsust ja testitavust. See arhitektuurimuster soodustab selget vastutuste eraldamist rakenduse äriloogika ja väliste sõltuvuste vahel, asetades põhilise rõhu projekti äriliste vajadustele. Heksagonaalse arhitektuuri modulaarne struktuur võimaldab arendajatel lisada uusi funktsioone või adaptereid ilma olemasolevat koodibaasi mõjutamata, mis omakorda muudab süsteemi lihtsamalt laiendatavaks ning edasi arendatavaks. Juhul, kui projekti nõuded vastavad eelmainitule, oleks soovituslik eelistada heksagonaalset arhitektuuri kihilisele arhitektuurile.

Tuleb siiski rõhutada, et heksagonaalne arhitektuurimuster ei pruugi alati olla parim valik igale tarkvaraprojektile, seda eriti väiksemate projektide puhul, millel on vähem väliseid sõltuvusi või lihtsamad ja vähem domeenikesksed nõuded. Eelmainitu puhul võib heksagonaalne arhitektuur kaasa tuua liiase keerukuse ning nõuda suuremat eeltööd. Väiksemate ning vähem keerukate tarkvaraprojektide puhul võiks selgelt eelistada kihilist arhitektuuri heksagonaalsele arhitektuurile.

Üheks tulevikuplaaniks on kahe eraldiseisva rakenduse ühendamine üheks. Ühtne rakendus lihtsustaks rakenduste vahelist andmete sünkroonis hoidmist ja andmevahetust ennast. Põhinedes eelnevale kogemusele seaks hüpoteesi, et kahe rakenduse üheks liitmine võiks tänu heksagonaalse arhitektuuri kasutamisele olla palju lihtsam kui kihilise arhitektuuriga.

# 7 Kokkuvõte

Käesoleva lõputöö eesmärgiks oli pärandisüsteemi refaktoreerimine ning arhitektuuriline üleminek kihiliselt arhitektuurilt heksagonaalsele arhitektuurile. Edasiarendatud süsteem on kaasaegsem, hallatavam ning jätkusuutlikum kui selle eelnev versioon.

Töö täitis kõik algselt seatud eesmärgid. Töö eesmärgid olid seatud väga teadlikult ning piiritletult kuna töö alguseks oli teada, et tegelikku soovitud tulemust ei ole võimalik antud ajaraamis teostada. Rakendus on tulemusena heksagonaalse ning kihilise arhitektuuri hübriid.

Heksagonaalsele arhitektuurile viidud kood on kooskõlas antud arhitektuuri tavadega, SOLID printsiipidega ning vastab *clean code* printsiipidele. Kihilisele arhitektuurile jäänud kood vastab suuremas osas *clean code* printsiipidele ning osaliselt SOLID printsiipidele. Vanale struktuurile jäetud kood ei vasta parimatele arendustavadele ning jäi refaktoreerimata ja restruktureerimata antud töö raames liiga suure mahu ja keerukuse tõttu. Lisaks vähendati refaktoreerimise käigus tehnilist võlga maksimaalsele võimalikule tasemele, kus alles jäänud võlg on seotud vana süsteemi *god class*'idega ning koodianalüüsi tööriistade poolt tuvastatavad vead on likvideeritud.

Testikatvuse tõstmise tagamail tõusis rakenduse testikatvus rohkem kui kolmekordselt ning rahuldab projektile sätestatud minimaalset testikatvuse taset. Kõrgem testikatvus tõestab paremini rakenduse toimimist ning teeb arendajatele süsteemimuudatuste sisseviimise palju lihtsamaks ja kindlamaks.

# Kasutatud kirjandus


[1] M. Martin ja R. C. Martin, Agile Principles, Patterns, and Practices in C#, Pearson, 2006.

[2] „PMD," [Võrgumaterjal]. Available: https://pmd.github.io/. [Kasutatud 18 4 2023].

[3] „SpotBugs," [Võrgumaterjal]. Available: https://spotbugs.github.io/. [Kasutatud 18 04 2023].

[4] SonarSource, „SonarQube," [Võrgumaterjal]. Available: https://www.sonarsource.com/products/sonarqube/. [Kasutatud 18 4 2023].

[5] „CheckStyle," [Võrgumaterjal]. Available: https://checkstyle.sourceforge.io/. [Kasutatud 18 4 2023].

[6] „Carnegie Mellon University," [Võrgumaterjal]. Available: https://www.sei.cmu.edu/our-work/software-architecture/. [Kasutatud 16 4 2023].

[7] M. Richards, Software Architecture Patterns, O'Reilly Media, Inc., 2015.

[8] J. Ingeno, „ISO/IEC/IEEE 42010 standard definition," %1 *Software Architect's Handbook*, Packt Publishing, 2018, p. 594.

[9] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design, Pearson, 2017.

[10] T. Hombergs, Get Your Hands Dirty on Clean Architecture, Packt Publishing, 2019.

[11] M. Alshayeb, „Empirical investigation of refactoring effect on software quality," *Information and Software Technology,* 2009.

[12] T. Mens ja T. Tourwé, „A survey of software refactoring," *IEEE Transactions on Software Engineering,* kd. 30, nr 2, pp. 126 - 139, 2004.

[13] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 2018.

[14] P. Kruchten, R. L. Nord ja I. Ozkaya, „Technical Debt: From Metaphor to Theory and Practice," *IEEE Software,* kd. 29, nr 6, pp. 18-21, 2012.

[15] P. Kruchten, R. L. Nord, I. Ozkaya ja D. Falessi, „Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes,* kd. 38, nr 5, pp. 51-54, 2013.

[16] M. Olan, „Unit testing: Test early, test often," *Journal of Computing Sciences in Colleges - JCSC,* kd. 19, 2003.

[17] V. Khorikov, Unit Testing Principles, Practices, and Patterns, Manning Publications, 2020.

[18] „JUnit website," [Võrgumaterjal]. Available: https://junit.org/. [Kasutatud 16 4 2023].

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

Mina, Kevin Raja

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "Koodibaasi refaktoreerimine ja üleviimine kihiliselt arhitektuurilt heksagonaalsele arhitektuurile riigiasutuse pärandisüsteemi näitel", mille juhendaja on Ants Torim
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

22.05.2023

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