TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Merily Adams 206279IABB

Domeenikihi modelleerimine pärandsüsteemi järkjärgulise moderniseerimise käigus kinnistusraamatu süsteemi näitel

Bakalaureusetöö

Juhendaja: Tarvo Treier
Magistrikraad

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: Merily Adams

17.05.2023

Annotatsioon

Käesoleva bakalaureusetöö eesmärgiks on viia läbi kinnistusraamatu pärandsüsteemi analüüs ning disainida ja rakendada muu määruse koostamine vahemääruse alaliigi näitel, mille põhjal on võimalik teostada järkjärguline üleminek uuele rakendusele. Disainimisel keskendutakse domeenikihi korrektsele modelleerimisele andmemudelist ja ärinõuetest lähtuvalt.

Töö käigus antakse ülevaade erinevatest domeenikihi modelleerimise võimalustest ning analüüsitakse, millise olemuse ja mastaabiga rakendustes on antud mustrite kasutamine otstarbekas ja õigustatud. Kinnistusraamatu uue süsteemi ärilisi nõudeid, piiranguid ning olemasolevat andmemudelit arvesse võttes selgus, et loodavale kinnistusraamatule osutus lähteülesannet ja nõudeid arvesse võttes sobivaimaks valikuks domeenimudeli disainimuster.

Analüüsi ning tööle esitatud äriliste ja süsteeminõuete põhjal realiseeriti domeenimudeli mustri põhimõtteid järgides muu määruse koostamise funktsionaalsus ning määruse ja menetluse seose loomine vahemääruse liigi näitel. Realisatsiooni käigus saavutatud tulemusi analüüsitakse püstitatud kriteeriumite alusel ning hinnatakse tööle seatud eesmärkide saavutamist.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 32 leheküljel, 6 peatükki, 22 joonist.

Abstract

Modelling of the domain layer during the incremental modernisation of the legacy software using the example of the Land Registry management system

The aim of this thesis is to analyse the legacy system of the Land Registry management system and to design and implement drafting other rulings, using the example of the ruling with omissions subtype, based on which it is possible to carry out an incremental transition to the new application. The design focuses on correct modelling of the domain layer based on the data model and business requirements.

During the work, an overview is provided of different possibilities for modelling the domain layer, along with analysis of the usage of these patterns to determine their suitability and justification in applications of various nature and scale. Taking into account the business requirements, constraints, and existing data model of the Land Registry management system, it became clear that the most suitable choice for designing the domain model, considering the project's objectives and requirements, was the domain model design pattern.

Based on the analysis along with business and system requirements presented for the work, the functionality for drafting other rulings and creating a connection between the ruling and the proceeding, using the example of a ruling with omissions type were implemented following the principles of the domain model pattern. The results achieved during the implementation are analysed according to previously set criteria, and the achievement of the set goals for the work is also evaluated.

The thesis is in Estonian and contains 32 pages of text, 6 chapters, 22 figures.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface.*ehk rakendusliides
CQRS	*Command and Query Responsibility Segregation* ehk muster, mis eraldab andmebaasist lugemise selle andmete muutmisest
DDD	*Domain-Driven-Design* ehk domeenipõhine disain
GoF	*Gang of Four*. Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides, kes tutvustasid objektorienteeritud arenduskeeltes kasutatavaid erinevaid mustreid [1]
Kandemäärus	Määrus, mille alusel tehakse kanne kinnistusregistrisse [2]
Kinnistusraamat	Riiklik andmekogu, kus registreeritakse kinnisasjad ja kinnisasjadega seotud asjaõigused
KRIS	Kinnistusraamatu menetlustarkvara, mille abil koostavad kohtunikuabid kandemäärusi
KRIS4	Kinnistusraamatu vana versioon, pärandsüsteem
KRIS5	Kinnistusraamatu uus, loodav versioon
Vahemäärus	Määruse liik, millega määratakse tähtaeg kinnistamisavalduses või sellega kaasnenud dokumentides esinevate puuduste kõrvaldamiseks

Sisukord

Jooniste loetelu

1 Sissejuhatus

Infotehnoloogia valdkond on pidevas muutumises, mille käigus arenevad nii kasutatavad tehnoloogiad kui ka uute rakenduste ja infosüsteemide loomise praktikad. Seetõttu jäävad vanad süsteemid ajale üsna kiiresti jalgu. Tihti ei ole pärandsüsteemide kasutamine erinevatel põhjustel enam võimalik ega turvaline või nende haldamine on muutunud keeruliseks. Seepärast tuleb pärandsüsteeme moderniseerida.

1.1 Taust ja probleem

Pärandsüsteem on ärikriitiline tarkvarasüsteem, mida on keeruline muuta ja hallata ning mille rike võib ettevõttele tõsiselt mõjuda [3]. Pärandsüsteemid on loodud, kasutades tehnoloogiaid ning tööriistu, mis erinevad sellest, kuidas täna süsteemi arendatakse. Kuigi süsteemide loomise hetkel võisid need tehnoloogiad olla otstarbekad ning kaasaegsed, siis aja möödudes on neist olemas juba uuemad versioonid või on kasutusele tekkinud uusi efektiivsemaid raamistikke ja tehnoloogiaid, mida süsteemide loomisel kasutada [4].

Üks suurimaid pärandsüsteemidega seotud probleeme on selle haldamine. Ühest küljest tuleneb see probleem näiteks dokumentatsiooni puudumisest, mistõttu ei ole teada, kuidas süsteem on üles ehitatud, millised on vajalikud litsentsid või kes on lõppkasutajad. Teisalt aga ei ole teada süsteemi iseärasused, näiteks, kui koodis on programmivigade lahendamiseks funktsioonid vastavalt vigadele ümber tehtud, mis programmile peale vaadates kohe välja ei tule. Kuna inimesed, kes koodi haldavad, ühel hetkel vahetuvad, kaovad teadmised süsteemi nüanssidest. Ehkki pärandsüsteemid võivad nõuetekohaselt töötada, kasvab uute funktsionaalsuste lisamisel või koodimuudatuste tegemisel võimalus süsteemivigade või -rikete tekkeks [4].

Kuigi pärandsüsteemidega kaasneb palju tehnilisi probleeme, ei ole süsteemide asendamine ettevõtete jaoks kuigi lihtne, kuna nõuab suuri väljaminekuid. Üldiselt on pärandsüsteemid ettevõtete ärikriitilised süsteemid, mis sisaldavad vajalikke andmeid ja informatsiooni äritegevuse toimimiseks [5]. Ärikriitilistel süsteemidel on suurem tõenäosus aja jooksul saada pärandsüsteemiks, kuna selliste süsteemide olulisuse ja

stabiilsuse vajalikkuse tõttu võetakse süsteemi suurem täiendamine või asendamine ette vaid muude lahenduste puudusel.

Töö autor töötab bakalaureusetöö kirjutamise hetkel ettevõttes Finestmedia AS. Koostöös Registrite ja Infosüsteemide Keskusega luuakse uus kinnistusraamatu rakendus, mida peab Tartu Maakohtu kinnistusosakond [6]. Hetkel kasutusel olev kinnistusraamat KRIS4 on loodud toimima vaid Internet Exploreri toel, mida hakati 2022. aastast ametlikult kasutuselt kõrvaldama [7]. Töö kirjutamise hetkel töötab pärandsüsteem vaid Google Chrome'i Internet Exploreri laienduse abil.

Uue kinnistusraamatu kasutuselevõtt toimub järk-järgult. Sujuvaks üleminekuks on vaja uus rakendus luua selliselt, et see ei häiriks olemasoleva rakenduse tööd, lisaks peab süsteemi kasutajatel olema andmetele katkematu ligipääs nii uue kui ka vana rakenduse kaudu. Arenduse käigus luuakse uus ja kaasaegne rakendus, mis lisaks eelnevalt mainitud probleemile tegeleb ka kasutusmugavuse ja jõudlusega seotud probleemide lahendamisega.

1.2 Töö eesmärk

Bakalaureusetöö eesmärgiks on viia läbi olemasoleva rakenduse analüüs ning disainida ja rakendada muu määruse koostamine vahemääruse alaliigi näitel, mille põhjal on võimalik teostada järkjärguline üleminek uuele rakendusele. Disainimisel keskendutakse domeenikihi korrektsele modelleerimisele andmemudelist ja ärinõuetest lähtuvalt.

Bakalaureusetöö eesmärgi täitmiseks on autor püstitanud järgnevad alameesmärgid:

1. Uurida erinevaid domeenikihi modelleerimise mustreid, tuua välja nende eripärad ning anda ülevaade, milliste äriloogiliste nõuete ja probleemide lahendamiseks vastavad mustrid sobivad.
2. Analüüsida arendusjärgus oleva kinnistusraamatu piiranguid, nõudeid ning andmemudelit ja valida rakendusele sobiv domeenikihi modelleerimise viis.
3. Disainida ning realiseerida eelneva analüüsi põhjal kinnistusraamatus muu määruse koostamine vahemääruse liigi näitel.
4. Analüüsida ja valideerida valminud realisatsiooni ning käsitleda alternatiivseid lahendusi.

1.3 Ülevaade tööst

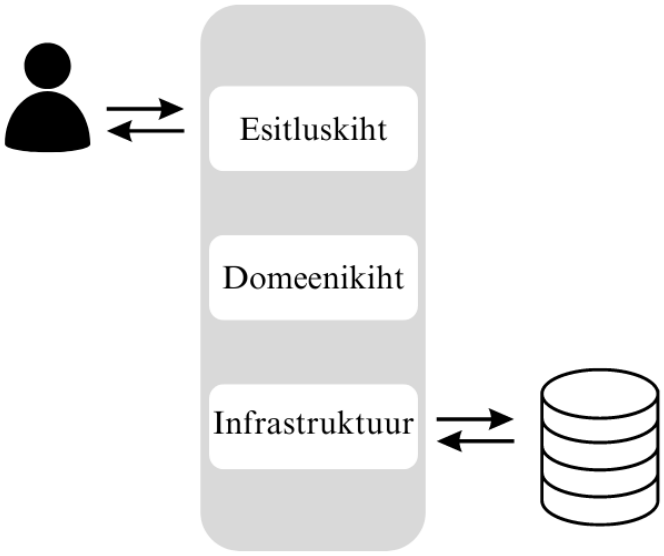
Töö teises peatükis tutvustatakse pärandsüsteemidega kaasnevaid murekohti ning antakse ülevaade erinevatest domeenikihi modelleerimise võimalustest, mis väljatoodud probleeme lahendada aitavad. Samuti analüüsitakse kirjeldatud domeenikihi disainimustrite juures selle omadusi ja ülesehitust ning tuuakse välja, milliste äriloogiliste nõuete ja probleemide lahendamiseks need kõige paremini sobivad. Kolmandas peatükis uuritakse arendusjärgus oleva kinnistusraamatu piiranguid, ärilisi nõudeid ning olemasoleva rakenduse andmemudelit, millest lähtuvalt tehakse saadud teadmiste põhjal uuele rakendusele sobivaima domeenikihi arhitektuuri valik. Neljandas peatükis keskendutakse äriloogikakihi modelleerimisele, mida illustreerivad kinnistusraamatul põhinevad näited, kujutades domeenikihi olemite ja meetodite loomist. Realisatsiooni käigus luuakse muu määruse koostamise funktsionaalsus. Töö viiendas peatükis kontrollitakse realisatsiooni ning sellele püstitatud süsteemi- ja ärinõuete täitumist. Seejärel valideeritakse saadud tulemusi ja arutletakse alternatiivsete lahenduste üle. Viimaks analüüsitakse tööle püstitatud eesmärkide saavutamist ja tuuakse välja võimalikud edasiarendused.

2 Domeenikihi modelleerimise võimalused ja analüüs

Selles peatükis antakse esmalt ülevaade pärandsüsteemide põhiprobleemidest. Seejärel uuritakse erinevaid domeenikihi modelleerimise võimalusi, tuues iga mustri kohta välja, milliseid probleeme nende omadused aitavad lahendada. Samuti analüüsitakse iga mustri juures, milliste äriloogiliste nõuete täitmiseks need enda olemuse juures kõige paremini sobivad, et uut süsteemi planeerides oleks võimalik mõistlik valik teha.

2.1 Domeenikiht

Rakendused jaotuvad suures plaanis kolmeks kihiks: esitluskihiks, domeenikihiks ja infrastruktuuri kihiks (Joonis 1). Esitluskiht on interaktiivne rakenduskiht lõppkasutajale, mille kaudu on võimalik edastada kasutaja sisendit domeenikihile ning kuvada domeenikihilt saadud andmeid kasutajale. Domeenikiht sisaldab valideerimisi ja arvutusi ning infrastruktuur ehk andmetele juurdepääsukiht määrab, kuidas hallata püsivaid andmeid andmebaasis või kaugteenustes [8].



Joonis 1. Rakenduse kihid.

Domeenikiht asub rakenduse keskmes. See kiht vastutab ärikontseptsioonide, äriolukorra teabe ja ärireeglite esitamise eest [9]. Domeenikiht hõlmab arvutusi, mis põhinevad

sisenditel ja salvestatud andmetel, kõikide esitluskihist tulevate andmete valideerimist ja sealt saadud käskudest lähtudes välja selgitamist, millist andmeallika loogikat edasi saata [10].

2.2 Ülevaade varasemast kirjandusest

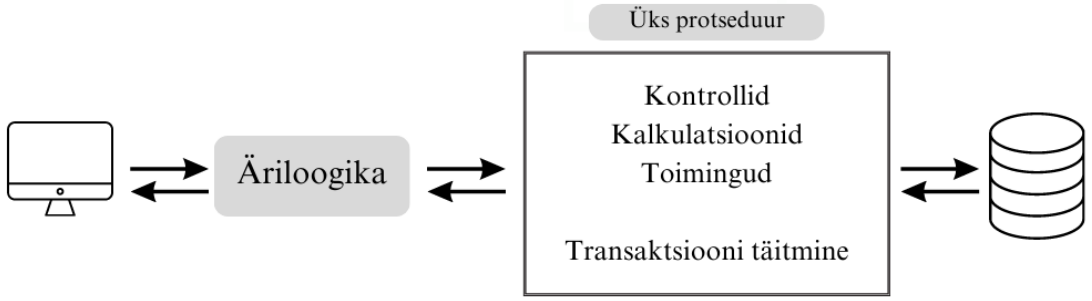
2014. aastal avaldatud uurimuses [3] viidi läbi intervjuud kaardistamaks, kuidas tajutakse pärandsüsteeme ja nende moderniseerimist tööstuses. Uurimuse käigus selgus, et kuigi pärandsüsteeme nähakse usaldusväärsete rakendustena, kaasnevad nendega mitmed murekohad. Esiteks peavad tänapäeva dünaamilises ärikeskkonnas organisatsioonid kiiresti kohanema erinevate muutustega ja seetõttu on oluline, et ärinõuete muudatused kajastuksid ka ettevõtte süsteemides. Paraku ei ole pärandsüsteemid muutuvate ärinõuete toetamiseks kuigi paindlikud. Ühtlasi toodi põhjustena, miks pärandsüsteeme moderniseeritakse, välja nende suur ülalpidamiskulu, teadmiste ja dokumentatsiooni puudumine ning süsteemide vastuvõtlikkus ebaõnnestumistele.

Pärandsüsteemide keeruka süsteemiarhitektuuriga tegelemine on üks olulisi väljakutseid, millega süsteemide moderniseerimisel silmitsi seistakse [3]. Efektiivse ning hea rakenduse üks eeldustest on korrektselt disainitud arhitektuur, mis võtab arvesse süsteemi vajadusi ja eripärasid. 2018. aastal avaldatud uurimuses [11] katsetati erinevaid äriloogikakihi disainimustreid akadeemiliste infosüsteemide peal. Uurimusest järeldus, et domeeniloogika mustrid mõjutavad tarkvara kvaliteeti. Üks mõõdik, mida mustri valik mõjutab, on programmi modulaarsus, mis näitab, kui palju on programmis eraldiseisvaid sõltumatuid osi. Samuti sõltub valikust rakenduse korduvkasutatavuse näit, mis eelkõige väljendub selles, kas klassid on disainitud taaskasutatavaks ja kui palju ühte klassi välja kutsutakse. Mida rohkem saab ühte klassi kasutada, seda kvaliteetsem on kood. Disainimustri valikust oleneb ka rakenduse hallatavus ehk programmi arusaadavus ning võimalus koodi täiendada, muuta või parandada. See, kui analüüsitav või testitav süsteem on, disainimustri valikust ei sõltu.

2.3 Transaktsiooniskript

Transaktsiooniskripti disainimustrit peetakse üheks lihtsaimaks disainimustriks [11]. Mustri lihtsus peitub selles, et transaktsiooniskript jaotab äriloogika protseduurideks, kus iga protseduur käsitleb ühte esitluskihi päringut või kasutusjuhtu. Igasugune

interaktsioon, mis kliendi- ja serverisüsteemi vahel on, sisaldab endas teatud koguses loogikat. Lihtsama protseduuri puhul võib selle sisu olla vaid info kuvamine andmebaasis, kuid teistel juhtudel võib üks protseduur sisaldada kasutaja sisendit esitluskihist, selle valideerimist ning seejärel andmete andmebaasi salvestamist või nende edastamist välistele teenustele [10], [12] (Joonis 2).



Joonis 2. Transaktsiooniskript.

Transaktsiooniskript suhtleb andmebaasiga kas otse või läbi õhukese andmebaasi *wrapperi* [10], mille abil andmebaas suhtleb andmeallikatega [13]. Iga transaktsioon sisaldab endas veidi lihtsat äriloogikat, näiteks valikut, millist informatsiooni näidata, sisendi valideerimist või arvutuste tegemist [14]. Igal transaktsioonil on oma transaktsiooniskript. Skriptid peaksid asuma klassides, mis on eraldi nendest, kus tegeletakse rakenduse andmete esitlusega ning andmeallikatega [10].

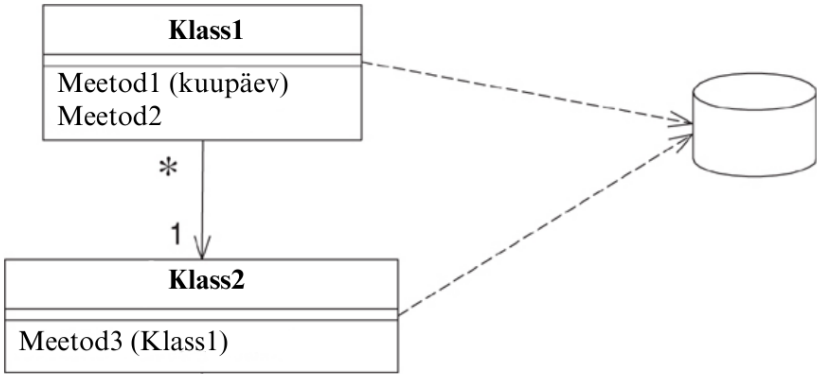
Mustri põhiline tugevus on tema lihtsus, mille tõttu on see arendajatele kergesti kasutatav ja mõistetav ning sellel disainimustril põhineva rakenduse loomine ja arendamine on kiire, kuna vajab realiseerimiseks vähem tööd kui teised domeeniloogika mustrid [15], [16]. Teine transaktsiooniskripti tugevus on eraldatus teistest transaktsioonidest: seda mustrit kasutades ei pea jälgima, mida teevad teised transaktsioonid [10]. Uute nõuete ilmnemisel on klassi väga lihtne lisada uusi meetodeid, kartmata olemasolevaid funktsioone mõjutada või rikkuda, mis on aluseks sellele, et mainitud omadus annab mustri kasutamisel rakendusele väga hea modulaarsuse [11], [17].

Transaktsiooniskripti kasutamine sobib väikeste rakenduste puhul, mis sisaldavad vaid vähesel määral loogikat ning ei vaja toimimiseks suurt arhitektuuri. Sellisel juhul tasub mustri kasutamine rakenduse lihtsust ja jõudlust silmas pidades ära, kuna transaktsiooniskripti puhul on süsteemi ressursside kasutus madal [10], [16]. Transaktsiooniskripti võib kasutada näiteks lihtsate broneerimissüsteemide puhul, kus

ühe transaktsiooni käigus toimuvad lihtsad kontrollid, vajadusel hinna arvutamine ja broneerimine [18]. Mustri kasutamine muutub keeruliseks juhul, kui rakenduse äriloogika keerukus kasvab. Kuna transaktsiooniskripti kogu idee on käsitleda vaid ühte transaktsiooni, on võimalus, et keeruka äriloogika puhul satub erinevaid transaktsioone kasutades rakendusse palju koodi- ja funktsioonide duplikatsioone, mille tõttu on aina raskem puhast koodi kirjutada ja mustrit hästi disainitud olekus hoida [10], [16]. Kui on näha, et rakenduse äriloogika keerukus kasvab, tasub arhitektuur üle viia mõnele teisele mustrile, kuid üldiselt ei ole transaktsiooniskripti refaktoreerimine mõneks teiseks mustriks kuivõrd lihtne, sest teised domeeniloogika mustrid sisaldavad suuremal või vähemal määral klasse, mis esindavad mõnda äriüksust, mis transaktsiooniskriptile omane ei ole [16].

2.4 Domeenimudel

Domeenimudel on domeeni objektmudel, mis sisaldab nii käitumist kui ka andmeid. Mudel loob omavahel seotud objektide võrgu, kus iga objekt esindab mõnda tähendusrikast rakenduse osa, mille suurus võib üsna palju varieeruda [10]. Domeenimudeli abil on võimalik luua keerulise loogikaga rakendusi. Selle asemel, et kasutada ühte protseduuri, mis käsitleb kogu kasutaja toiminguga seonduvat äriloogikat, on domeenimudelis mitu erinevat objekti, millest igaüks käsitleb vaid osa domeeniloogikast, olles vastutav kindla ülesande eest [17], [19] (Joonis 3).



Joonis 3. Domeenimudel.

Domeenimudeli lisamine rakendusse hõlmab terve kihi objektide sisestamist, mis modelleerivad seda ärivaldkonda, mille jaoks rakendus luuakse. See tähendab, et kiht sisaldab objekte, millest ühed jäljendavad ettevõtte andmeid ning teised kajastavad

organisatsioonis kasutatavaid reegleid. Kuna ettevõtte käitumine võib palju muutuda, on oluline, et seda kihti oleks võimalik lihtsalt muuta, ehitada ning testida. Seetõttu on vajalik, et domeenimudel ning süsteemi teised kihid oleksid omavahel minimaalselt seotud. Objektorienteeritud domeenimudel näeb sageli sarnane välja kasutatava andmebaasimudeliga, kuid sellel on siiski palju erinevusi: domeenimudel segab omavahel andmeid ja protsesse, omab mitme väärtusega atribuute ja keerukat seoste võrku ning kasutab pärimist [10].

Domeenimudeleid on kahte tüüpi. Lihtsad mudelid on disainitud andmebaasi järgi, kus iga andmebaasi tabeli jaoks on üks domeeniobjekt. Selle mudeli puhul saab kasutada otsest *Active Record* andmebaasi kaardistamise ja sidumise mustrit, mis viib andmetele ligipääsu loogika domeeniobjekti. Teine domeenimudel on „rikas“ domeenimudel, mis võib andmebaasi disaini poolest väga palju erineda pärimiste, strateegiate ning omavahel ühendatud väikeste objektide keeruliste võrkude tõttu. „Rikas“ domeenimudel on parem keerulise loogika kirjeldamiseks, kuid seda on andmebaasiga raskem siduda. Seetõttu vajab see mudel andmekaardistajat (ing.k *Data Mapper*), mis liigutab andmeid objektide ja baasi vahel, kuid hoiab need üksteisest lahus ning iseseisvatena. Seetõttu on see parim viis käsitleda neid juhtumeid, kus domeenimudel ja andmebaasiskeem üksteisest erinevad [10].

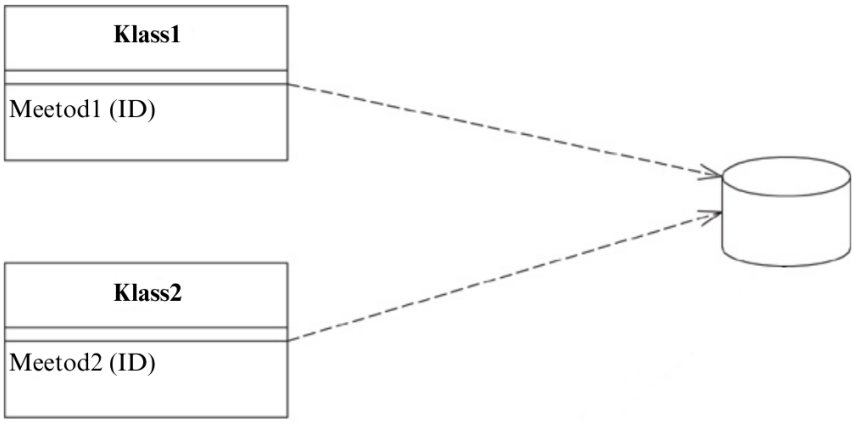
Mustri puhul on eeliseks loogika organiseeritus, kus iga objektitüüp vastutab vaid sellele määratud ülesannete eest. Samuti on tugevuseks see, et rakenduse loogika ja andmeallikate keerukuse suurenedes saab domeenimudelit vastavalt skaleerida [14]. Samas, kui aja jooksul domeeni skoop suureneb, võib rakenduse ülesehitus muutuda liiga kompleksseks [20]. Suurim probleem domeeniloogika puhul on laialivalguvad domeeniobjektid, mis tähendab seda, et kui klass sisaldab vastutusi, mis on vajalikud vaid ühe kasutusjuhu puhul, võivad klassid liiga suureks minna [10]. Lisaks võib mustri puhul ühe toimingu käitumise jälgimine olla keeruline, sest toimingu osad on erinevate komponentide vahel laiali jaotatud [14].

Domeenimudeli disainimuster sobib süsteemidele, millel on keeruline töövoog ja äriloogika ning mis eeldab tegeliku ärivaldkonna mudeli modelleerimist, kui ärireeglid on komplekssed, pidevalt muutuvad ning hõlmavad endas valideerimist, arvutusi ja tuletusi [10], [17]. Seega sobib domeenimudelit kasutada näiteks e-kaubanduse veebirakendustes, kus on määratud erinevat tüüpi olemid, mis omavahel seotud on [19].

Kui aga rakenduses on vaid lihtsad kontrollid ja kerged arvutused, on transaktsiooniskript parem valik [10]. Domeenimudeli mustri rakendamine võib olla äärmiselt kulukas. See on tehniliselt kõige keerulisem ning nõuab rakendamiseks arendajaid, kellel on hea arusaam objektorienteeritud programmeerimisest [17].

2.5 Tabelimoodul

Tabelimoodul on üks eksemplar, mis haldab andmebaasitabeli või vaate kõigi ridade äriloogikat [10] (Joonis 4). See tähendab, et muster liigitab äriprotsessid vastavalt eelnevalt loodud tabelitele, kus ühe andmebaasitabeli kohta on domeenikihis üks klass. Iga tabelimoodul sisaldab protsesse, mida on kontrolleril tarvis selleks, et andmeid andmebaasitabelist saada [10], [11].



Joonis 4. Tabelimoodul.

Tabelimoodulid sarnanevad klassidele domeenimudelis, kuid on andmebaasi struktuuriga tihedamalt seotud. Selle mustri puhul on rakenduses üks-ühele kaardistus (ing.k *mapping*) andmebaasitabelite ja domeenikihis asuvate klasside vahel ning kliendid kasutavad ainult ühte klassieksemplari iga tabeli jaoks [14].

Tabelimooduli eelisena võib tuua välja asjaolu, et see võimaldab rakenduse andmeid ja käitumist kokku panna ning samal ajal kasutada ka relatsioonilise andmebaasi tugevusi [10]. Käitumise andmetabeliga kokku viimine annab palju ka kapseldamise eeliseid, mille abil piiratakse otsest juurdepääsu objekti sisule, varjatakse andmefunktsioone, meetodeid või olekut [10], [21]. Kapseldamise muudab tabelimoodulite puhul lihtsaks asjaolu, et rakenduse käitumine on viidud andmete juurde, millega see töötab [10]. Võrdluseks teiste domeenikihi disainimustritega võib välja tuua, et tabelimoodulid on struktureeritumad

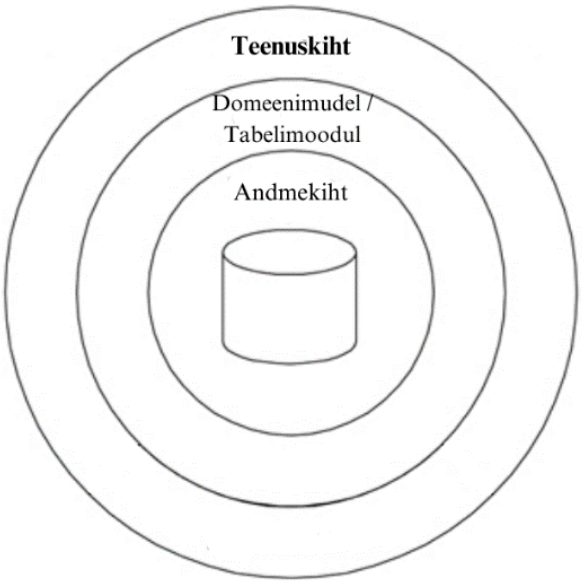
kui transaktsiooniskript ning arusaadavamad ja lihtsama andmeallikate kaardistamisega kui domeenimudel [14]. Lisaks on uurimus näidanud, et tabelimoodulite mustriga rakendusel on kõigist töös uuritavatest disainimustritest parim hallatavuse väärtus [11]. Kuid sarnaselt transaktsiooniskriptile ei ole tabelimooduli mustrit võimaik kasutada keerulise äriloogikaga rakendustes, kuna selle puhul ei saa olla eksemplaridevahelisi seoseid ning ka polümorfism ei tööta kuigi hästi, seega ei ole keeruliste süsteemide puhul tabelimoodulid niivõrd paindlikud, nagu seda on domeenimudeli muster [10], [14].

Nagu ka varasemalt öeldud, toetub tabelimoodul tabelipõhistele andmetele. Tabeliandmed on üldiselt SQL-i päringu tulemus ning neid hoitakse kirjekomplektis (ing.k *Record Set*), mis jäljendab SQL-i tabelit. Seega on tabelimoodulite mustrit mõistlik kasutada rakendustes, kus andmetele on juurdepääs kirjekomplekti abil. Konkreetse rakenduse näitena võib tuua kasutajate haldamise süsteemi, mis hõlmab endas kasutajate registreerimist ja sisselogimist, käsitledes kasutajate tabeli kõikide ridade äriloogikat [22]. Lisaks asetab mustri kasutamine mainitud andmestruktuuri suurel määral koodi keskmesse, seega kui rakenduses kasutatakse kirjekomplekti andmestruktuuri, peaks sellele juurdepääs olema lihtne ja sirgjooneline, mida on võimalik tabelimoodulite mustriga saavutada [10].

2.6 Teenuskiht

Tihti vajavad rakendused salvestatavate andmete ja rakendatava loogika jaoks erinevat tüüpi liideseid. Vaatamata nende erinevatele eesmärkidele vajavad liidesed sageli ühist suhtlust rakendusega, et pääseda juurde selle andmetele, nendega manipuleerida ning käivitada süsteemi äriloogika. Suhtlemise loogika kodeerimine igas moodulis eraldi põhjustab palju duplikatsioone, seega on selliste probleemide vältimiseks parem koondada äriloogika ülesehitamine ühte teenuskihti [23].

Teenuskiht on domeeniloogika abstraktsioon [23]. See määratleb rakenduse piirid, loob saadaolevate toimingute komplekti ja koordineerib rakenduse vastust igas toimingus [10]. Teenuskihi abil on võimalik esitlus domeeniloogikast lahti siduda, muutes mitme liidese toetamise lihtsaks [14]. Levinud viis domeeniloogika käsitlemiseks on jagada see kaheks osaks, kus teenuskiht asetatakse aluseks oleva domeenimudeli või tabelmooduli kohale [24] (Joonis 5).



Joonis 5. Teenuskiht.

Kuna ärirakendustel on tihti sama funktsionaalsuse kohta mitu liidest, tasub sellistel juhtudel jaotada äriloogika domeeniloogikaks ning rakenduse loogikaks [10]. Domeeniloogika alla kuuluvad kõik reeglid ja protseduurid, mis on äri juhtimiseks vajalikud. Rakenduse loogika on justkui sild domeeniloogika ja kasutajaliidese vahel, mis rakendab neid ärireegleid spetsiifilise rakenduse raames. See sisaldab kõiki reegleid ja protsesse, mis kontrollivad, kuidas kasutaja andmetega suhtleb [25]. Mõlema loogika kombineerimine samades klassides võib kaasa tuua probleeme klasside korduvkasutatavusega, mistõttu koondab teenuskiht igat tüüpi äriloogika eraldi kihti, muutes puhtad domeeniobjektiklassid erinevate rakenduste või liideste vahel korduvkasutatavaks [10].

Teenuskihi rakendamiseks on kaks varianti. Domeeni fassaadi lähenemisviisi puhul rakendatakse teenuskiht õhukeste fassaadide kogumina domeenimudelile. Fassaade realiseerivad klassid äriloogikat ei realiseeri, seda teeb domeenimudel. Õhukesed fassaadid loovad aga piirid ja toimingute kogumi, mille kaudu kliendikihid rakendusega suhtlevad. Toiminguskripti lähenemisviisis rakendatakse teenuskiht hoopis paksemate klasside kogumina, mis realiseerivad otseselt rakenduse loogikat, kuid delegeerivad domeeniloogika domeeniobjekti klassidele [10].

Teenuskihti tuleks kasutada vaid juhul, kui arendataval rakendusel on mitu klienti ja kasutajaliidest ning selle kasutusjuhud hõlmavad endas mitut keerukat tehinguressurssi [10]. Näiteks sobib teenuskiht keerulisematele tervishoiusüsteemidele [26]. Kui

rakendusel on vaid üks klient, on rakenduse ehitamiseks viise, kus teenuskiht ei ole vajalik ning tehingute juhtimine ja vastuste koordineerimine käib kontrollerite abil [10].

3 Kinnistusraamatu analüüs

Selle peatüki eesmärk on anda ülevaade kinnistusraamatu olemusest ning kasutusel oleva pärandsüsteemi põhiprobleemidest, mida hilisema realisatsiooni käigus lahendama hakatakse. Seejärel analüüsitakse arendusjärgus oleva rakenduse piiranguid, ärilisi nõudeid ning olemasolevat andmemudelit, mille abil valitakse disainimiseks ja realiseerimiseks sobiv domeenikihi modelleerimise muster.

3.1 Pärandsüsteemi probleem kinnistusraamatu näitel

Eestis kasutusel olev kinnistusraamat on Tartu Maakohtu kinnistusosakonna poolt elektrooniliselt peetav riiklik register, milles talletatakse kinnisasju ja kinnisasjadega seotud õiguseid. Lisaks andmete talletamisele on kinnistusraamatu infosüsteemi eesmärgiks anda täielikku õiguslikku informatsiooni kinnisasjade, nende omanike, õiguste ja koormatiste kohta. Kuna nendele andmetele tuginetakse igapäevases majandustegevuses ja strateegiliste otsuste tegemisel, on kinnistusraamatu andmete tervikluse tagamine väga oluline.

Kinnistusraamatu hetkel kasutusel olev versioon KRIS4 võeti kasutusele 2006. aastal, mis tähendab, et see on enda tehnoloogilise eluea lõpus. Paraku ei ole võimalik vananenud tehnoloogiat kaasaegseks muuta. Tulenevalt vastuvõetavatest seaduse- ja ärinõuete sagedastest muudatustest peab kinnistusraamatu süsteem olema muutuste suhtes paindlik. Pärandsüsteemi uuenduste sisseviimine on aga kulukam kui uue süsteemi arendamine, mistõttu ei ole selle haldamine pikas perspektiivis kuigi jätkusuutlik. Oluline aspekt on ka see, et on võrdlemisi keeruline leida arendajaid, kes oskaksid pärandsüsteemi arendada.

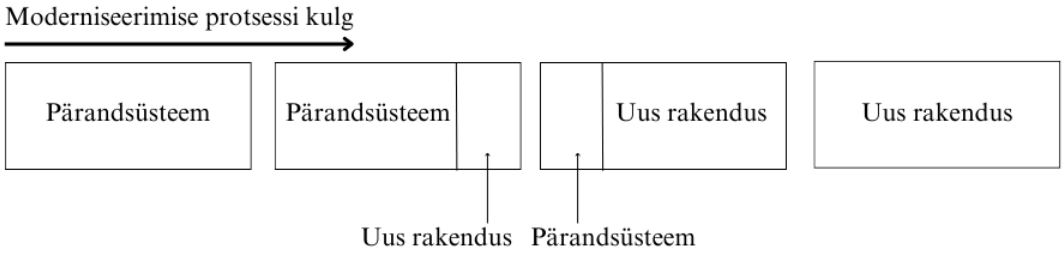
Uue süsteemi ehitamise üheks suuremaks ajendiks oli kasutajamugavuse uuendamise vajalikkus. Aja jooksul on kinnisvaratehingud muutunud üha keerulisemaks. Ühe menetluse raames tuleb sageli tegeleda sadade või tuhandete kinnisasjade koormamisega. Tänane süsteem aga ei suuda oma jõudluse juures tihtipeale ilma IT-spetsialisti manuaalse sekkumiseta toime tulla suurte tehingute menetlemisega, mille tõttu pikeneb oluliselt nii menetlusaeg kui väheneb kasutusmugavus. Kasutusmugavust ning ka turvalisust silmas pidades on üks kriitilisemaid probleeme asjaolu, et KRIS4 on loodud töötama vaid Internet Exploreril, mille tugi on tänaseks lõppenud. Käesoleva töö

kirjutamise hetkel toimib rakendus vaid Google Chrome'i Internet Exploreri laienduse abil. Samuti tasub ära märkida, et rakenduse kasutajaliidese disain ja funktsionaalsus ei ole igapäevast tööd soodustav ning kasutajale käepärane.

Kõik väljatoodud probleemid on ärikriitilised ning vajavad lahendusi äri katkematuks toimimiseks. Kuna probleemide mastaap takistab lõppkasutajate tööd, muutes pärandsüsteemis tegutsemise äärmiselt ebamugavaks, kasutatakse rakenduse järkjärgulist migreerimist kaasaegsesse süsteemi, et kasutajad saaksid harjuda loodava süsteemiga ning teha jõudluse seisukohast kriitilisemad toimingud juba uues rakenduses, mis aitab neil enda tööd kiirendada.

3.2 Piirangud uue rakenduse loomisel

Nagu varasemalt mainitud, on kinnistusraamatu andmed igapäevase majandustegevuse ja otsuste vajalik sisend, mistõttu kogu uue infosüsteemi arenduse aja jooksul peab kasutajatel olema katkematu ligipääs kinnistusraamatu andmetele. Seetõttu toimub pärandsüsteemi moderniseerimine järkjärgulise arendusena, mille käigus püütakse hoida süsteemi kogu aeg täielikult töökorras, vähendades samas moderniseerimise ajal ette tulevaid tehnilisi töid ja riske [27]. Järkjärgulise arenduse puhul arendatakse uut rakendust väiksemate osade kaupa, mis aja jooksul pärandkoodi funktsionaalsuse üle võtavad (Joonis 6).



Joonis 6. Pärandsüsteemi järkjärguline moderniseerimine.

Kuna kliendi poolt on seatud nõue, et KRIS5 toimiks andmete töötlemiseks ja menetlemiseks paralleelselt ning liidestatult vana süsteemiga, kasutab ja täiendab uus rakendus KRIS4 põhiandmebaasi, tänu millele jäävad andmed kättesaadavaks nii uue kui vana rakenduse kaudu. Moderniseerimise protsessi juures on aga tähtis silmas pidada, et kõik muudatused, mis andmebaasis tehakse, ei tohi olemasoleva rakenduse tööd mingil moel häirida ega takistada. Seatud piirang muudab aga andmebaasitabelite muutmise

keerulisemaks. Ühest süsteemist teise viidav tehing või toiming tõenäoliselt kas loeb, loob või värskendab andmebaasi ühte või mitut tabelit. Seega ei saa eeldada, et tehing, mida uude süsteemi üle viiakse, on ainus, mis konkreetset andmebaasitabelit muudab [27]. Järelikult ei ole andmetele juurdepääsu seisukohast võimalik kogu andmebaas korraga välja vahetada, mistõttu on vaja domeeniloogika modelleerimisel lähtuda olemasoleva andmebaasi skeemist ning selle ülesehitusest.

3.3 Ärilised nõuded

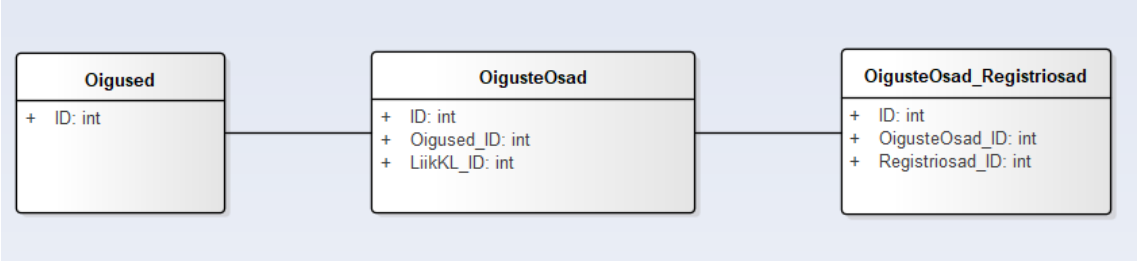
Kinnistusraamat on olemuselt äärmiselt keeruline ja suuremahuline süsteem. Andmekogu pidamist reguleerib üle kümne erineva õigusakti. Kuna seaduseid ning muid õigusakte pidevalt arendatakse ja modifitseeritakse, peavad muudatused kajastuma ka kinnistusraamatus. Seetõttu peab ühe nõudena olema uus rakendus modulaarse ülesehitusega. Kuna modulaarsus väljendab süsteemis sõltumatuid osi, teeb see omadus rakenduse hästi hallatavaks, testitavaks ning vastuvõtlikuks uutele muudatustele ja täiendustele, mida tasub domeenimustri valikul silmas pidada. Ärireeglite lisandumisel on tähtis, et rakendust oleks võimalik vastavalt vajadusele skaleerida selliselt, et süsteemi arhitektuur säiliks puhtana, mille tõttu on ka edaspidi võimalik süsteemi täiustada ja hallata. Kliendi püstitatud nõuetest lähtuvalt peab loodav kinnistusraamat olema jätkusuutlikul tehnilisel platvormil ja arhitektuuril, olles seeläbi toetatav professionaalsete arenduspartnerite poolt.

Tulenevalt komplekssetest ärireeglitest, nõuetest ja mastaabist on kinnistusraamat äriloogiliselt üsna keerulise ülesehitusega. Domeenikihi modelleerimisel on tarvis rakenduse arhitektuur disainida selliselt, et see oleks reaaleluline ja äriprotsesse ning -reegleid kirjeldav. Vajadus tuleb ühest küljest asjaolust, et organisatsiooni reeglite muutumisel on täiustusi lihtne ka koodis väljendada ja testida. Teisalt aga peaks äriloogika domeenimudelist selgelt välja paistma, kuna sel viisil on koodis mustvalgelt kajastatud kliendi vajadused ja nõuded ning tuleviku vaates on süsteemi arendajatel parem arusaam toimingute töövoogudest, kui need on üles ehitatud äriloogiliselt korrektselt ning reaalelulisi toiminguid silmas pidades.

3.4 Andmemudeli analüüs

Kinnistusraamatu andmebaas kasutab relatsioonilist mudelit, millega väljendatakse andmebaasis olevaid seoseid tabelite vahel. Relatsioonilises andmemudelis esindab tabeli iga rida seotud andmeväärtuste kogumit, millest igaüks tähistab reaalelulist olemit või suhet [28].

Olemasoleva rakenduse andmemudelit uurides selgub, et mudel ei ole modelleeritud täielikult korrektselt ning esineb tabeleid ja nendevahelisi seoseid, kus andmemudel ei ole üks-ühele vastavuses äriloogikaga. Võttes aluseks tabeli *OigusteOsad*, mis hoiab infot õiguse N arvu osade kohta, selgub, et samasse tabelisse on lisatud ka seos õiguse ja valitsevate registriosade vahel, kus tabel *OigusteOsad* käitub vahetabelina (Joonis 7). Tabelis *OigusteOsad* on andmeväli *LiikKL_ID*, mis väljendab õiguse osa liiki. Juhul, kui õigusel on valitsev registriosa, on välja väärtuseks määratud *null*. Kuigi andmebaasiskeemi põhjal oleks õiguste osad ja valitsev registriosa justkui üks olem, tuleb domeenikihis teha olemite vahel eristus, et äriloogika korrektselt modelleeritud oleks.



Joonis 7. Seos õiguste ja valitsevate registriosade vahel.

Uue rakenduse ehitamisel on piirang jätta vana süsteem paralleelselt toimima, mistõttu tuleb domeenikiht modelleerida olemasoleva andmebaasiskeemi põhjal. Kuna äriloogika ja andmemudel omavahel üheses seoses ei ole, tuleb domeenikiht modelleerida selliselt, et domeeniobjektideks ei ole mitte üks-ühele andmebaasitabelites olevad olemid, vaid objekt esindab äriloogikast tulenevat olemit.

3.5 Domeenikihi arhitektuuri valik ja selle põhjendus

Teises peatükis analüüsiti põhjalikult nelja erinevat domeenikihi modelleerimise võimalust. Väljatoodud mustrid olid transaktsiooniskript, domeenimudel, tabelimoodul ning teenuskiht. Selleks, et rakendus oleks ka tulevikus hallatav ning uuendatav, tuleb

vastavalt süsteemi omadustele, piirangutele ja nõuetele valida sobiv domeeni disainimuster.

Kinnistusraamat on olemuselt pidevalt muutuv süsteem. Rakendus peab tulenevalt õigusaktide ja seaduste sagedastest muudatustest olema kergesti uuendatav ning vastuvõtlik uute täiustuste suhtes. Kõige lihtsam muster, nagu ka teises peatükis on välja toodud, on transaktsiooniskript. Äritegevusse uute nõuete lisandumisel on selle mustri kasutamisel üsna kerge koodi funktsionaalsust juurde lisada. Kuna transaktsiooniskript annab teatavasti rakendusele väga hea modulaarsuse, oleks sellega võimalik teostada kliendi püstitatud ärinõue rakenduse ülesehituse kohta. Mustri kasutamise teeb kinnistusraamatu puhul keeruliseks fakt, et kinnistusraamat on äriloogiliselt väga kompleksse ülesehitusega, mis transaktsiooniskripti kasutamise nõuetega aga kokku ei sobi. Muster on mõeldud pigem lihtsate ja väikeste rakenduste jaoks. Ka tabelimooduli muster ei ole loodud väga keerulise äriloogikaga rakenduste jaoks, kuigi selle puhul on uurimus näidanud parimat hallatavuse astet, mis kinnistusraamatu süsteemi juures tähtsat rolli mängib. Kõige paremini saab kompleksse rakendusega hakkama domeenimudel, mis organisatsiooni käitumist jäljendab ning mida vastavalt süsteemi loogika keerukuse kasvule skaleerida saab. Teenuskihi mustri puhul jäi kõlama peamine mõte, et teenuskihti tasub kasutada vaid juhul, kui lisaks mitmele keerulisele tehinguressursile on rakenduses mitu klienti ja kasutajaliidest. Kuigi kinnistusraamatu süsteem on kompleksne, ei ole rakendusel teenuskihi mustri kasutamise tingimused täidetud.

Kinnistusraamatu arendus toimub järk-järgult, kus arenduse jooksul hoitakse mõlemad süsteemid paralleelselt töös, kusjuures nad toimetavad samal andmebaasil. Kinnistusraamatu rakendus kasutab relatsioonilist andmebaasi, millel kõige efektiivsemalt toimetab tabelimooduli muster, kuna see on andmebaasi struktuuriga väga tihedalt seotud. Kinnistusraamatu täpsema analüüsi käigus aga selgus, et andmemudel ei ole korrektselt modelleeritud ning ei vasta kõikidele ärinõuetele. Kuna tabelimooduli puhul tuuakse klassid andmetabelitest üks-ühele domeenikihti, ei oleks sel viisil domeenikiht äriloogiliselt korrektse ehitusega. Tulenevalt piirangust jätta projekti arendades mõlemad süsteemid tööle, ei saa baasis kogu mudelit ümber teha ärilisi vajadusi kirjeldavaks, mistõttu seda mustrit kinnistusraamatu puhul kasutada ei saa. Teine muster, mis nii käitumist kui andmeid sisaldab ning on tabelimoodulile sarnane, on domeenimudel, kuid erinevalt tabelimoodulist ei ole see andmebaasiga nii tihedalt seotud.

„Rikas“ domeenimudel võib andmebaasiskeemist erineda, võimaldades domeenikihis luua olemeid, mis esindavad äriloogikat, mitte ei kajasta üks-ühele andmetabeleid.

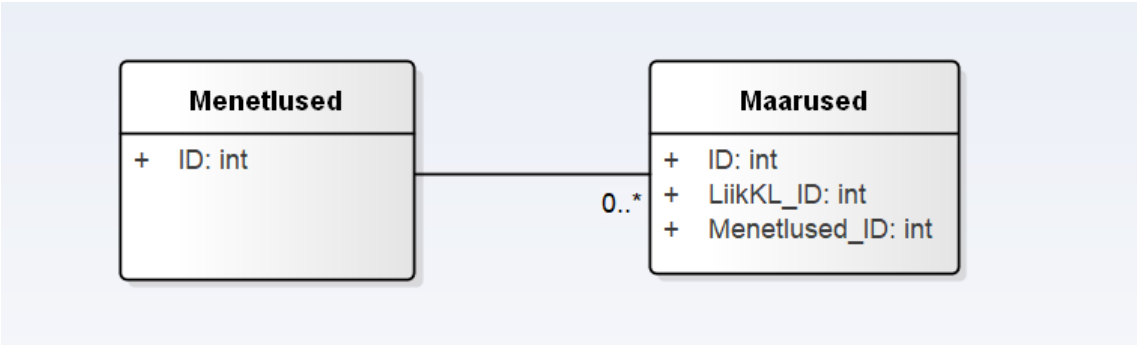
Piiranguid, ärinõudeid ja andmemudeli analüüsi arvestades osutus kõige sobivamaks domeenikihi mustriks „rikas“ domeenimudel, mis võimaldab ehitada keerulise ülesehituse ja äriloogikaga rakendusi, mida oleks võimalik nõuete lisandumise ja reeglite muutumise korral hõlpsalt täiendada. Samuti on mustrit kasutades võimalik domeenikiht modelleerida äri kirjeldavaks, ilma, et peaks modelleerimisel andmebaasist juhinduma.

4 Realisatsioon

Selles peatükis keskendutakse äriloogika kihi modelleerimisele eelneva analüüsi põhjal ning tuuakse kinnistusraamatu põhjal näide, kuidas modelleerida rakenduse domeenikihi olemid ja meetodid. Esmalt antakse ülevaade rakenduses menetluste ja määruste seosest ning tuuakse välja süsteeminõuetes kirjeldatud kriteeriumid, mida realisatsiooni käigus teostada tuleb. Seejärel realiseeritakse kinnistusraamatu alammoodul „Muu määruse koostamine“, mille käigus rakendatakse muu määruse liik vahemäärus ning näidatakse määruse liigi loomise ja menetlusega sidumise meetodeid.

4.1 Ülevaade realisatsioonist ning sellele püstitatud kriteeriumitest

Kõik kinnistusraamatus tehtavad toimingud on hõlmatud menetlustega. Menetluse käigus saab menetleja teha otsuseid, mida nimetatakse määrusteks ning lisada neid menetlusse. Menetluste ja määruste seos on kujutatud joonisel 8. Menetlusel saab olla mitu määrust, seega on määruste tabel menetluste külge seotud andmevälja *Menetlused_ID* kaudu, mis väljendab selle menetluse unikaalset identifikaatorit, milles määrus on koostatud. Määruseid on kinnistusraamatu projektis mitmeid erinevaid liike, millel on kõigil oma olemus ning kriteeriumid, millal ning millele need kohalduvad. Määruse liik asub klassifikaatorite tabelis ning on määruste tabelis väljendatud *LiikKL_ID* andmevälja abil.



Joonis 8. Menetluste ja määruste seos.

Käesoleva töö raames luuakse muude määruste koostamise alamplokk, kus realiseeritakse vahemääruse loomine. Vahemäärus on määrus, millega määratakse tähtaeg kinnistamisavalduses või sellega kaasnenud dokumentides esinevate puuduste kõrvaldamiseks. Järgnevalt on välja toodud mõned kliendi poolt koostatud käesoleva töö

raames relevantsed süsteeminõuded, mis hõlmavad muude määruste genereerimise äriloogikat ning mida tuleb domeenikihi realiseerimise käigus silmas pidada:

- Muu määruse loomiseks peab kasutaja olema määratud menetluse menetlejaks.
- Muud määrust saab luua järgmistes menetluse staatustes:
    - 1: „Registreeritud“
    - 2: „Sisse kantud“
    - 4: „Suunatud“
    - 5: „Projekt“
    - 131: „Puuduste kõrvaldamisel“
    - 132: „Vahemääruse tähtaeg ületatud“
    - 1258: „Kinnistuskohtuniku menetluses“.
- Määrusele määratakse unikaalne number.
- Määruse liik määratakse vastavalt kasutaja poolt tehtud valikule.
- Sõltuvalt määruse liigist määratakse puuduste kõrvaldamise seisund. Kui määruse liik on „Vahemäärus“, lisatakse seisund „287: Esinesid puudused“.
- Määrus seotakse menetlusega.
- Määrus seotakse kinnistusjaoskonnaga. Kinnistusjaoskonnaks määratakse määrusega seotud menetlust menetlev kinnistusjaoskond.
- Määruse andmetesse salvestatakse määruse loomise aeg.

Domeenikihi realiseerimiseks kasutatakse C# programmeerimiskeelt ja .NET raamistikku.

4.2 Domeenikihi realisatsioon

4.2.1 Vahemääruse olemi realiseerimine

Olemid esindavad domeeniobjekte, olles domeenimudeli väga tähtis osa, kuna just need on kogu mudeli aluseks. Domeenipõhise disaini (DDD) arendusmustrit järgides peab domeeniolem rakendama domeeni loogikat või käitumist, mis on seotud olemi andmetega [29].

Selleks, et kasutaja saaks luua vahemäärust, tuleb esmalt implementeerida vahemääruse olem, mis asub klassis *RulingOnOmission* (Joonis 9). Kuna vahemäärus on määruse alaliik, siis pärib vahemäärus klassist *Ruling*. See tähendab, et *Ruling* on määruste

baasklass, millest pärivad kõik tema alamklassid, saades baasklassi omadused ja käitumise.

```
public class RulingOnOmission : Ruling
{
    [UsedImplicitly]
           protected RulingOnOmission() : base(RulingType.RulingOnOmission){}

    internal RulingOnOmission(RulingNr rulingNumber,
             DepartmentId departmentId, DateTime currentTime)
       : base(RulingType.RulingOnOmission, rulingNumber, departmentId,
       RulingOmissionState.OmissionsExisted, currentTime) {}
}
```

Joonis 9. Vahemääruse olem.

Määruse loomisel antakse antud näites baasklassi atribuutidena kaasa määruse liik *RulingType.RulingOnOmission*, mis kujutab vahemäärust, eelnevalt genereeritud määruse number, menetlusega seotud kinnistusjaoskond ning aeg, mis on väärtustatud praeguse ehk määruse loomise ajahetkega. Lisaks teostatakse vahemääruse loomisel süsteeminõue puuduste kõrvaldamise seisundi kohta, kus *RulingOmissionState.OmissionsExisted* väljendab seisundit „Esinesid puudused“.

4.2.2 Määruse loomine

Enne määruse loomist tuleb kontrollida, kas menetlus on ühes süsteeminõuetes välja toodud staatuses. Kuigi kontroll ise ei toimu domeenikihis, tuleb ärinõuetes kirjeldatud piirang kirjutada domeenikihti *ProceedingStatus* klassifikaatorite klassi. *ProceedingStatus* kasutab *SmartEnum* teeki. *Enumid* on väärtustüübid, mida saab kasutada, et täpsustada mõnda tüüpi või kategooriat, antud näites on selleks menetluse staatused. *SmartEnumi* puhul viiakse selle klassi *enumi* tüübid, nende omadused, käitumine ja loogika. *SmartEnumi* kasutamine võimaldab seega koodi paremini hallata ja testida ning domeeniloogikat paremini laiendada [30]. Antud näite puhul tuleb vastavalt vajadusele luua uus menetluse staatus ja sellega seotud ärireeglid *enum* tüübina *ProceedingStatus* klassi. Sel juhul on kogu staatuse loogika ja reeglid ühes kohas ning neid kasutavad klassid ei pea muretsema ärireeglite pärast. Joonisel 10 on kujutatud menetluse staatuste defineerimist, kus näitena luuakse staatused „Puuduste kõrvaldamisel“ ning „Vahemääruse tähtaeg ületatud“.

```
public static readonly ProceedingStatus EliminationOfOmissions =
new(nameof(EliminationOfOmissions), 131);

public static readonly ProceedingStatus WithOmissions =
new(nameof(WithOmissions), 132);
```

Joonis 10. Menetluse staatuste defineerimine.

Kui menetluse staatused on defineeritud, on võimalik luua kollektsioon staatustega, milles on muu määruse koostamine lubatud. Selleks luuakse samasse klassi *OtherRulingCreationEnabledStatuses* (Joonis 11), mida on võimalik enne muu määruse looma hakkamist kontrolliks kasutada.

```
public static IEnumerable<ProceedingStatus>
OtherRulingCreationEnabledStatuses =>
    new[] { Registered, Entered, Directed, Pending,
         EliminationOfOmissions, WithOmissions,InProceedingOfAJudge };
```

Joonis 11. Muude määruste koostamiseks menetluse lubatud staatused.

Määruste erinevad liigid on defineeritud klassifikaatori klassis *RulingType*, mis kasutab samuti *SmartEnum* teeki ning kus on defineeritud nii liigid, mis on rakenduses juba kasutusel kui ka need, mis edasise arenduse käigus realiseeritakse. Joonis 12 kujutab vahemääruse liigi defineerimist, kus lisaks liigile määratakse ära selle liigi ID väärtus, milleks antud juhul on 43 ning tõeväärtuse atribuut, mis kirjeldab, kas muu määruse loomine on lubatud või mitte.

```
public static readonly RulingType RulingOnOmission =
new(nameof(RulingOnOmission), 43, true);
```

Joonis 12. Vahemääruse liigi defineerimine.

Kuna muudest määrustest on töö kirjutamise hetkel realiseeritud vaid vahemäärus, on lisatud klassi tõeväärtuse atribuut *IsOtherRulingAddingEnabled* (Joonis 13), mis tõese väärtuse korral võimaldab kasutajaliideses valida vaid tõeseks määratud muude määruste liike. Vaikimisi on atribuudi väärtus väär. Tõene väärtus väärtustatakse vaid nendele muudele määrustele, mille funktsioonid ning toimiv andmeedastus on realiseeritud.

```
public bool IsOtherRulingAddingEnabled { get; }

private RulingType(string name, int value, bool
isOtherRulingEnabled = false) : base(name, value)
{
       IsOtherRulingAddingEnabled = isOtherRulingEnabled;
}
```

Joonis 13. Määruse liigi tõeväärtuse atribuut.

Rakenduskihis asuvad CQRS põhimõtteid järgides käskude töötlejad (ing.k *command handler*). Määruste käskude töötlejad sisaldavad endas loogikat, millega esitatakse domeenikihile käsk luua uus määrus. Kuigi käsk tuleb rakenduskihist, peavad DDD printsiipe järgides kõik olemi andmetega seotud toimingud käima olemiklassi meetodite kaudu. Sel viisil ei muutu objektorienteeritud koodist transaktsiooniskript ning rakendus jääb hallatavaks ja kontrollitavaks [31]. Vastavalt määruse liigile toimub loomine määruse liikide *RulingType* klassis, mille eest vastutab meetod *CreateRuling* (Joonis 14).

```
public static Ruling? CreateRuling(RulingType type, RulingNr rulingNumber,
DepartmentId departmentId, DateTime currentTime)
    {
        if (type == RulingOnOmission)
            return new RulingOnOmission(rulingNumber, departmentId,
currentTime);

        else if (type == RulingOnEntry)
            return new RulingOnEntry(rulingNumber, departmentId,
RulingOmissionState.NoOmissions, currentTime);

        else throw NotImplementedException;
    }
```

Joonis 14. Määruse loomine vastavalt liigile.

Meetod *CreateRuling* kasutab GoF tehase meetodi mustrit (ing.k *factory method pattern*). Tehasemeetodi puhul defineeritakse liides objekti loomiseks, kuid lastakse alamklassidel otsustada, millist objekti luuakse [1]. *CreateRuling* võtab teiste hulgas sisendiks määruse liigi, mille abil on võimalik luua uus määruse olem ning tagastada õige liigiga määrus, mis teeb selle meetodi kõikide määruste loomisel korduvkasutatavaks, kuna sel moel on objektide loomine paindlikum, ning iseseisvalt testitavaks. Kui meetod õiget määruse liiki ei leia, tagastatakse käskude töötlejale *NotImplementedException*.

4.2.3 Määruse lisamine menetlusse

Kui määrus on loodud, tuleb see lisada menetluse külge. Kuna domeenimudel kajastab reaalelu, kus menetluse käigus luuakse uus määrus ning võttes aluseks andmetabelite seosed, kus menetluse ja määruse vahel on vastavalt üks-mitmele seos, kajastatakse määruse lisamine menetluse klassis *Proceeding*. Klassi on loodud kaks *AddRuling* meetodit, mis võtavad sisenditeks erineva arvu parameetreid (Joonis 15). Erinev parameetrite arv tuleneb sellest, et kandemääruse puhul võib määruse loomine kohalduda mitmele toimingule korraga, mistõttu on kahe parameetriga *AddRuling* meetodi üheks sisendiks toimingute list *List<Operation> operations*. Määruse ning menetluse seose

loomise loogika on aga vaid ühes *AddRuling* meetodis, mida teine meetod samuti enda töös kasutab.

```
public T AddRuling<T>(T ruling) where T : Ruling
    {
        Rulings.AreUpdatedWith(() => _rulings.Add(ruling));
        AddChange(() => new ProceedingEvents.V1.RulingAdded(Id, ruling.Id));

        return ruling;
    }


public T AddRuling<T>(T ruling, List<Operation> operations) where T : Ruling
    {
        foreach (var operation in operations)
            operation.AddRuling(ruling);

        return AddRuling(ruling);
    }
```

Joonis 15. Määruse lisamine menetlusse.

Joonisel kujutatud esimeses meetodis luuakse esmalt menetluse ja määruse seos menetluse juurde. Seejärel lisatakse muudatused sündmusesse (ing.k *event)*. Sündmused võimaldavad klassil või objektil teavitada teisi klasse või objekte, kui midagi huvipakkuvat toimub [32]. Seega, kui domeenisündmus käivitub, tehakse selle juures veel kõrvalsündmus, mis käitub domeenisündmuselt tulnud sõnumi peale. Sellega on määruse ja menetluse seos loodud ning *AddRuling* meetodi töö lõpeb määruse tagastamisega.

4.2.4 Vahemääruse alaliigi käskude töötleja

Domeenikihile käskude andmiseks kasutatakse käskude töötlejaid, mis asuvad rakenduskihis. CQRS põhimõtteid kasutades on loodud *AddOtherRulingCommandHandler,* mis sisaldab endas kahte meetodit: *Handle* ning *AddNewOtherRulingToProceeding* (Lisa 2). Muude määruste käskude töötlejas pannakse kasutaja jaoks kokku eelnevates alapeatükkides kirjeldatud osadest ühtne tervik. Seal toimuvad esmased valideerimised, mille läbimisel antakse domeenikihile korraldused luua muu määruse objekt ja lisada loodud olem menetlusse.

Selleks, et kontrollida, kas menetlus on staatuses, milles muud määrust saab luua, tuleb kõigepealt pärida andmebaasist vastavalt menetluse numbrile õige menetlus ning kontrollida selle staatust. Kui menetlus ei ole ühes eelnevalt defineeritud lubatud

staatuses, ei ole võimalik muud määrust luua (Joonis 16). Kasutajale tagastatakse vea korral erind ning veatekst.

```
var proceeding = await _proceedingRepository.GetByProceedingNrAsync(
   command.ProceedingNr, cancellationToken)
       .GetValueOrFailNotFound(command.ProceedingNr);

if (!ProceedingStatus.OtherRulingCreationEnabledStatuses.Contains(
   proceeding.Status))
     throw BusinessRuleViolationException.Translated("proceeding-
     view.draft-other-rulings.error.unenabledProceedingStatus");
```

Joonis 16. Menetluse staatuse kontrollimine.

Ühtlasi on süsteeminõuetes kirjeldatud, et muu määruse koostamiseks peab kasutaja olema määratud menetluse menetlejaks. Ka selle kontrolli käsk antakse edasi käskude töötlejas, mis läbi erinevate meetodite ja andmebaasipäringu kaudu selgitab välja, kas kasutaja on antud menetluse menetleja (Joonis 17). Sellega välistatakse võimalus, kus muu määrus luuakse kasutaja poolt, kellel selleks õigust ei ole.

```
if (!(await
_proceedingAuthorizationService.CurrentUserIsRegistrarOfProceeding(
command.ProceedingNr, cancellationToken)).IsAuthorized)
    throw BusinessRuleViolationException
             .Translated("ruling.error.compilerNotRegistrar");
```

Joonis 17. Menetluse menetleja kontrollimine.

Määrus seotakse kindla kinnistusjaoskonnaga. Kinnistusjaoskond saab olla vaid see üksus, kus menetletakse menetlust, milles määrust koostama hakatakse. Süsteeminõue määruse ja kinnistusjaoskonna seose kohta täidetakse, kui tuvastatakse autenditud kasutaja ning päritakse andmebaasist tema jaoskonna identifikaatori *departmentID* järgi kinnistusjaoskond (Joonis 18).

```
var authenticatedUser = _currentUserService.AuthenticatedUser;

var activeDepartment = await _departmentRepository.GetByIdAsync(
authenticatedUser.DepartmentId, cancellationToken)
      .GetValueOrFailNotFound(authenticatedUser.DepartmentId);
```

Joonis 18. Kinnistusjaoskonna määramine.

Muu määruse liigi klassifikaatori ID, menetlus ning kinnistusjaoskond, milles määrust looma hakatakse, antakse edasi meetodile *AddNewOtherRulingToProceeding* (Joonis 19). Lisaks sisaldab meetod ühe atribuudina tühistamise märki (ing.k *cancellation token),* mida kasutatakse toimingute ühiseks tühistamiseks.

```
await AddNewOtherRulingToProceeding(command.OtherRulingTypeId,
proceeding, activeDepartment, cancellationToken);
```

Joonis 19. Muu määruse lisamise meetodi väljakutsumine.

Kuna muu määruse liike on mitmeid, määratakse *AddNewOtherRulingToProceeding* meetodis esmalt määruse liik, mille tuvastamine käib liigi ID alusel klassifikaatorite hoidlas (ing.k *repository*). Lisaks täidetakse süsteeminõue, kus määrusele määratakse unikaalne number määruse numeraatoris kirjeldatud loogika abil (Joonis 20).

```
var otherRulingType = await
_classifierRepository.GetClassifierValueAsync(
otherRulingTypeId, RulingType.Descriptor, cancellationToken);

var otherRulingNr = await _rulingNumerator.GenerateNumberAsync
(cancellationToken);
```

Joonis 20. Määruse liigi ja unikaalse numbri määramine.

Kuna kõik muude määruste liigid ei ole bakalaureusetöö kirjutamise hetkel veel realiseeritud, loodi eelnevalt domeenikihti määruse liikide klassi tõeväärtusatribuut, mis väljendab, kas vastavat liiki saab eesrakenduses rippmenüüst valida ja muude määruste loomisel kasutada. Käskude töötlejas tehakse vastav kontroll, mis valitud liigi koostamise lubamatuse korral tagastab erindi, kus tagastatakse kasutajale vea korral ka tekst, milles väljendatakse määruse liigi toetamatust (Joonis 21).

```
if (!otherRulingType.IsOtherRulingAddingEnabled)
    throw BusinessRuleViolationException.Invariant(
    $"Unsupported ruling type '{otherRulingType.Name}'");
```

Joonis 21. Määruse alaliigi lubatuse kontroll.

Olles läbinud erinevaid kontrolle ning kogunud kokku kõik määruse loomiseks vajamineva informatsiooni, antakse käskude töötleja poolt domeenikihile korraldus määruse loomiseks. Määruse liigi klassi *CreateRuling* meetodisse antakse parameetritena kaasa muu määruse liik, määrusele genereeritud unikaalne number, kinnistusosakonna, kus määrus luuakse, identifikaator ning määruse loomise ajahetk. Kui valitud liigiga määrus on loodud, annab käskude töötleja domeenikihis asuvale menetluse klassile *Proceeding* käsu lisada loodud määrus menetlusse (Joonis 22).

```
var otherRuling = RulingType.CreateRuling(otherRulingType,
otherRulingNr, activeDepartment.Id, _dateTime.LocalNow);

   if (otherRuling == null)
      return;

proceeding.AddRuling(otherRuling);
```

Joonis 22. Määruse loomise ja menetlusse lisamise käsud.

5 Tulemused ja järeldused

Selles peatükis kontrollitakse, kas süsteemi- ja ärinõuded said realisatsiooni käigus täidetud. Seejärel valideeritakse saadud tulemusi ja arutletakse alternatiivsete lahenduste üle. Viimaks analüüsitakse tööle püstitatud eesmärkide täitumist ja tuuakse välja võimalikud edasiarendused.

5.1 Realisatsiooni nõuete kontrollimine

Eelmises peatükis kirjeldatud realisatsiooni käigus toodi näide domeenikihi modelleerimisest domeenimudeli mustri abil. Implementatsiooni käigus loodi vahemääruse olem, realiseeriti muude määruste koostamine ning loodi seos määruste ja menetluste vahel. Käesolevas alapeatükis analüüsitakse realisatsiooni ning selle vastavust äri- ja süsteeminõuetele.

Vaadates tagasi kolmandas peatükis püstitatud piirangutele ja ärinõuetele, kus on välja toodud, et rakendus peab paralleelselt töötama olemasoleva süsteemiga sama andmebaasi peal ning uus rakendus peab olema modulaarse ülesehitusega jätkusuutlikul arhitektuuril, võib öelda, et mõlemad nõuded said täidetud. Loodud funktsionaalsus asendab pärandsüsteemis oleva muude määruste koostamise ploki. Kaasaegseid tehnoloogiaid ja parimaid praktikaid kasutades muutub lisatud osa kasutajate jaoks intuitiivsemaks, kasutajasõbralikumaks ning jõudluselt kiiremaks. Kuigi domeenimudelil ei ole teises peatükis välja toodud uurimuse näitel neljast analüüsitud domeenikihi mustrist kõige parem modulaarsus, on see siiski loodud viisil, mis võimaldab kergesti rakendust muudatuste esinedes kohandada, tuues näiteks realisatsiooni käigus vajalikuks osutunud domeenikihti lisatud menetluse staatused või realiseeritud määruste liigid. Samuti muudab rakenduse modulaarsemaks käskude töötlejate kasutamine, mis võimaldab olemasoleva käskude töötleja vajadusel välja vahetada ning ehitada domeenikihi peale uue kasutusloo, asendades olemasoleva töötleja funktsionaalsuse. Jätkusuutlikku arhitektuuri silmas pidades võib välja tuua ka asjaolu, et realiseeritud alamplokk on implementeeritud ärireeglitele ja -nõuetele tuginedes. Seega on tulevikus kinnistusraamatu arendajatel võimalik lihtsalt järgida toimingute töövoogusid ning seetõttu paremini mõista rakenduse ülesehitust ja andmeedastust erinevate rakenduse kihtide vahel.

Neljandas peatükis on kirjeldatud kliendi poolt püstitatud süsteeminõuded spetsiifiliselt muude määruste koostamist silmas pidades, mida käesoleva töö realisatsiooni raames lahendati. Süsteeminõue menetluste staatuste kohta realiseeriti, kui domeenikihti loodi *ProceedingStatus* klassi menetluse staatuste tüübid, mida rakenduskihis kontrollina kasutati.

Vahemääruse liigi olemis realiseeriti süsteeminõue, mis kirjeldas määruste puuduste kõrvaldamise seisundit. See nõue realiseeriti viisil, kus vahemääruse loomisel *RulingOnOmission* klassis määratleti määruse defineerimisel puuduste tüübina *RulingOmissionState.OmissionsExisted*, mis tähistab seisundit "Esinesid puudused".

Määruse liigi valiku jaoks loodi *RulingType* klassifikaatorite klassi tehasemeetodi mustrit kasutades funktsioon *CreateRuling,* mis vastavalt kasutaja poolt valitud määruse tüübile loob õige liigiga ja selle omadustega määruse. *CreateRuling* on loodud viisil, et seda oleks võimalik tulevikus kasutada kõikide määruse liikide loomise puhul.

Määruse sidumine menetlusega toimub *Proceeding* klassis, tulenevalt andmetabelite seosest ja äriloogikast. Menetlusse on loodud kaks *AddRuling* meetodit, millest üks on mõeldud kandemäärustele ja teine muude määruste liikidele. *AddRuling* meetodis luuakse seos menetluse ja määruse vahel, lisades määruse menetlusse ning käivitades kõrvalsündmuse edasisteks toiminguteks.

Rakenduskihis ühendati *AddOtherRulingCommandHandler* käskude töötlejas kõik eelnevalt domeenikihti loodud osad ühtseks tervikuks. Selle meetodites viidi läbi kontroll menetluse menetleja kohta, edastati käsk määruse unikaalse numbri genereerimiseks, seoti loodav määrus kinnistusjaoskonnaga ning salvestati määruse andmetesse määruse loomise aeg. Kõik määruse loomiseks vajaminevad väärtused anti sisendina domeenikihti vahemääruse loomise meetodile kaasa, kus loodi määrus ning seejärel andis käskude töötleja domeenikihis asuvale menetluse klassile korralduse määruse ja menetluse sidumiseks.

5.2 Tulemuste valideerimine

Domeenikihi modelleerimisel järgis töö autor puhta koodi ning domeenimudeli ja tehasemeetodi mustrite kasutamise põhimõtteid. Kõik loodud funktsionaalsused on kaetud autori poolt kirjutatud toimivate integratsioonitestidega. Esmasel

koodiülevaatusel sai töö autor kolleegidelt tagasisidet mõne ettepanekuga, mis realisatsiooni käigus teostati. Kuigi lisatud alamplokki ei ole töö kirjutamise hetkel veel manuaalselt testitud ega *live* pandud, tegi töö autor toimivast funktsionaalsusest demonstratsiooni nii kolleegidele, kliendile kui lõppkasutajatele, mis kõigilt osapooltelt heakskiidu sai.

5.3 Alternatiivid

Pärandsüsteemi moderniseerimiseks on mitmeid erinevaid viise. See, millist lähenemist kasutatakse, sõltub pärandsüsteemi olemusest, moderniseerimise kriitilisusest ning paljudest muudest teguritest.

Kui KRIS4 moderniseerimise aluseks oleks olnud ainult kasutusmugavus, Internet Exploreri toe lõppemine ning ajale jalgu jäänud kasutajaliides, oleks kõige kergem olnud uue eesrakenduse loomine, mis kasutaks olemasoleva rakenduse API-d. Seda tüüpi moderniseerimine on tavaliselt madalate kulude ja riskiga, kuid tõstab äriväärtust veebisaitide parema kasutatavuse, hooldatavuse ja vastupidavuse näol [33].

Eeldusel, et oleks olnud võimalik teha kogu rakendus algusest peale ümber, võiks esmalt muuta andmebaasi ärilisi vajadusi kirjeldavaks: vaadata üle, mis peab andmebaasis kajastuma ning milliseid tabeleid enam vaja ei ole, lisaks korrigeeriks tabelitevahelisi seoseid. Hetkel on rakenduses selliseid kohti, kus on andmebaasi vale disaini tõttu ebaloogilisust. Kui andmemudel oleks modelleeritud korrektselt, oleks uues süsteemis vähem keerukust kui see hetkel endas sisaldab.

Vaadates tulevikku, on olemas reaalne võimalus, et arendusjärgus olev rakendus tuleb omakorda ümber teha. Hetkel ollakse siiski suurel määral sõltuv vanast rakendusest, kuna arendust piirab vajadus hoida ka see süsteem paralleelselt töös nii, et igal ajahetkel oleks võimalik kinnistusraamatu andmetele ligi pääseda ja kõiki tööks vajalikke toiminguid teha. Kui aga ühel hetkel pärandsüsteemi töö lõpetatakse, tekib suurem vabadus muuta mõningaid funktsionaalsusi, muuta andmetabeleid või rakenduse arhitektuuri, et süsteem oleks veelgi loogilisemalt ja puhtamalt üles ehitatud.

5.4 Töö eesmärgi täitumine

Käesoleva töö teoreetilises osas anti põhjalik ülevaade erinevatest domeenikihi modelleerimise võimalustest ning toodi välja, millise arhitektuuriga rakendustele need kõige paremini sobivad. Ülevaate andmise käigus selgus, et iga mustri puhul on kriteeriumid, millal on nende kasutamine otstarbekas ja millised on mustrite kasutamise piirangud.

Olemasoleva süsteemi andmebaasiskeemi ning uue rakenduse äriliste nõuete ja piirangute analüüsi põhjal selgus, et teoreetilises osas välja toodud mustritest sobib realisatsioonis kasutamiseks kõige paremini domeenimudeli muster. Valik tehti selle põhjal, et domeenimudel sobib kõige paremini suurtele ja äriloogiliselt keerulistele rakendustele, mis on vastuvõtlikud sagedaste muudatuste suhtes ning ei sõltu liigselt andmebaasi ülesehitusest, mis samuti valiku tegemisel suurt rolli mängis.

Eelnevate analüüside põhjal disainiti ja realiseeriti muu määruse koostamise funktsionaalsus vahemääruse alaliigi näitel, võttes töö põhifookuseks domeenikihi korrektse modelleerimise domeenimudeli disainimustrit järgides. Loodud alammoodul võtab üle olemasoleva pärandsüsteemi funktsionaalsuse, mistõttu on muid määrusi edaspidi võimalik luua juba uues rakenduses. Käesoleva töö realisatsiooni osas toodud näide on alus, mille põhjal modelleerida domeenikihti ning teostada järkjärguline üleminek uuele rakendusele.

5.5 Võimalikud edasiarendused

Enne rakenduse loomist tuleb põhjalikult süsteemi arendamist planeerida: mõelda arhitektuurile ja selle ülesehitusele, uurida ärinõudeid, panna kirja vajaminevad funktsionaalsused ning pidada silmas ka piiranguid, mis rakenduse arendamist võivad kujundada. Eric Evans on öelnud, et tarkvara süda on selle võime lahendada kasutaja jaoks domeeniga seotud probleeme [34]. Seega on tähtis, et domeenikihi ja kogu äriloogika modelleerimine oleks tehtud läbimõeldult. Vastavalt rakenduse olemusele ja eesmärgile tuleb valida sobiv modelleerimise disainimuster ja sellest kinni pidada. Sellisel juhul on rakendus kõige puhtam, hallatavam ning esineb väiksem tõenäosus, et tulevikus tekivad süsteemi hooldamisega suured probleemid ja nendega kaasnevad kulud.

Käesoleva töö raames sai põhjalikumalt analüüsitud ja realiseeritud domeenimudeli muster. Tuleviku vaates oleks sarnaselt domeenimudelile võimalik luua informatiivsed ja reaalelulised näited ka teistele töös kirjeldatud mustritele ning koostada detailne juhend erinevate mustrite näidetega, mis annaks ülevaate, kuidas domeenikihti erinevalt modelleerida. Antud kokkuvõtet oleks võimalik rakenduse arhitektuuri planeerimisel juhendmaterjalina kasutada ning seeläbi paremini süsteemi kavandada ja visualiseerida.

6 Kokkuvõte

Käesoleva bakalaureusetöö põhieesmärgiks oli viia läbi kinnistusraamatu pärandsüsteemi analüüs ning disainida ja rakendada muu määruse koostamine vahemääruse alaliigi näitel, mille põhjal on võimalik teostada järkjärguline üleminek uuele rakendusele.

Erinevaid domeenikihi modelleerimise võimalusi uurides sai autor paremad teadmised disainimustritest, nende ülesehitusest, piirangutest ja sobivusest erineva mastaabi ja keerukusega rakendustele. Töö teoreetiline osa oli aluseks uue kinnistusraamatu rakenduse disainimustri valimisele.

Uue kinnistusraamatu loomist piirab vajadus jätta olemasolev süsteem paralleelselt tööle. Kuna mõlemad süsteemid toimivad samal andmebaasil, ei tohi uue rakenduse arendamine takistada ega häirida pärandsüsteemi tööd, seega arendatakse uut süsteemi järk-järgult. Analüüsides olemasoleva rakenduse andmemudelit ning uue rakenduse ärilisi nõudeid ja piiranguid, selgus, et parim disainimuster kinnistusraamatu puhul on domeenimudel, mida on võimalik kasutada keerulise äriloogikaga suurte ning pidevalt muutuvate süsteemide puhul.

Bakalaureusetööle püstitatud põhi- ja alameesmärgid said täidetud. Töö realisatsiooni käigus loodi domeenimudeli mustrit kasutades muu määruse koostamise funktsionaalsus, määruse menetlusse lisamise meetodid ning defineeriti vahemääruse liik. Disainimisel keskenduti domeenikihi korrektsele modelleerimisele andmemudelist ja ärinõuetest lähtuvalt.

Lõputöö andis autorile uusi teadmisi nii domeenikihi erinevatest mustritest kui ka oskusi, kuidas rakenduse eripärasid arvestades sellele sobiv arhitektuuriline muster valida. Kuna loodava rakenduse arhitektuuri kavandamine on iga uue projekti raames vajalik ning aktuaalne, annavad käesolevas töös käsitletud teemad hea ülevaate erinevatest domeenikihi modelleerimise võimalustest, olles võimalik juhendmaterjal uute süsteemide loogikakihi planeerimisel.

Kasutatud kirjandus

[1] E. Gamma, R. Helm, R. Johnson ja J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994.

[2] Riigi Teataja, „Kinnistusraamatuseadus," 15 09 1993. [Võrgumaterjal]. Available: https://www.riigiteataja.ee/akt/117032023061. [Kasutatud 06 05 2023].

[3] R. Khadka, B. V. Batlajery, A. M.Saeidi, S. Jansen ja J. Hage, „How do professionals perceive legacy systems and software modernization?," Association for Computing Machinery, New York, 2014.

[4] R. Annett, Working with Legacy Systems, Packt Publishing, 2019.

[5] H. K. A. Bakar, R. Razali ja D. I. Jambari, „Implementation Phases in Modernisation of Legacy Systems,"*6th International Conference on Research and Innovation in Information Systems (ICRIIS)*, Johor Bahru, Malaysia, 2019.

[6] Justiitsministeerium, „Kinnistusraamat," [Võrgumaterjal]. Available: https://www.just.ee/kohtud-ja-oigusteenused/registrid/kinnistusraamat. [Kasutatud 14 04 2023].

[7] S. Lyndersay, „The future of Internet Explorer on Windows 10 is in Microsoft Edge," Microsoft Windows Blogs, [Võrgumaterjal]. Available: https://blogs.windows.com/windowsexperience/2021/05/19/the-future-of-internet-explorer-on-windows-10-is-in-microsoft-edge/. [Kasutatud 14 04 2023].

[8] M. Fowler, „Presentation-Domain-Data Layering," 26 08 2015. [Võrgumaterjal]. Available: https://martinfowler.com/bliki/PresentationDomainDataLayering.html. [Kasutatud 10 05 2023].

[9] Microsoft, „Design a DDD-oriented microservice," 13 04 2022. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ddd-oriented-microservice. [Kasutatud 10 05 2023].

[10] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[11] S. Rochimah, P. G. Nuswantara ja R. J. Akbar, „Analyzing the Effect of Design Patterns on Software Maintainability: a Case Study," IEEE, Batu, Indonesia, 2018.

[12] R. Raszczynski, „Inviqa," Inviqa UK Ltd., 31 10 2011. [Võrgumaterjal]. Available: https://inviqa.com/blog/architecture-patterns-domain-model-and-friends. [Kasutatud 21 04 2023].

[13] IBM, „Wrappers and wrapper modules," IBM Documentation, 20 01 2023. [Võrgumaterjal]. Available: https://www.ibm.com/docs/en/db2/11.5?topic=systems-wrappers-wrapper-modules. [Kasutatud 21 04 2023].

[14] M. Dailey, „Software Architecture Design. Architectural Patterns," Computer Science and Information Management Asian Institute of Technology.

[15] M. Turis, „Domain-driven design with architectural patterns,“ Masaryk University Faculty of Informatics, Brno, 2019.

[16] G. Peipman, „Transaction Script Pattern,“ Gunnar Peipman Programming Blog, 24 10 2013. [Võrgumaterjal]. Available: https://gunnarpeipman.com/transaction-script-pattern/. [Kasutatud 21 04 2023].

[17] S. Millett ja N. Tune, Patterns, Principles, and Practices of Domain-Driven Design, Wrox, 2015.

[18] Java Design Patterns, „Transaction Script,“ [Võrgumaterjal]. Available: https://java-design-patterns.com/patterns/transaction-script/. [Kasutatud 12 05 2023].

[19] Java Design Patterns, „Domain Model,“ [Võrgumaterjal]. Available: https://java-design-patterns.com/patterns/domain-model/. [Kasutatud 23 04 2023].

[20] N. Khaitan, „Design Patterns for the database layer,“ Medium, 11 09 2022. [Võrgumaterjal]. Available: https://medium.com/towards-polyglot-architecture/design-patterns-for-the-database-layer-7b741b126036. [Kasutatud 23 04 2023].

[21] „Encapsulation - definition & overview,“ Sumo Logic, [Võrgumaterjal]. Available: https://www.sumologic.com/glossary/encapsulation/. [Kasutatud 21 04 2023].

[22] Java Design Patterns, „Table Module,“ [Võrgumaterjal]. Available: https://java-design-patterns.com/patterns/table-module/. [Kasutatud 12 05 2023].

[23] „Service Layer,“ Java Design Patterns, [Võrgumaterjal]. Available: https://java-design-patterns.com/patterns/service-layer/. [Kasutatud 23 04 2023].

[24] H. Hussen, „Patterns of Enterprise Application Architecture — Organizing Domain Logic,“ Medium, 09 03 2022. [Võrgumaterjal]. Available: https://medium.com/javarevisited/patterns-of-enterprise-application-architecture-organizing-domain-logic-50efd9ea3f39. [Kasutatud 23 04 2023].

[25] W. Meier, „Business Logic vs. Application Logic: The Key Differences You Need to Know,“ APIsec, 18 05 2022. [Võrgumaterjal]. Available: https://www.apisec.ai/blog/business-logic-vs-application-logic. [Kasutatud 23 04 2023].

[26] H. YANG, K. LIU ja W. LI, „Adaptive Requirement-Driven Architecture for Integrated Healthcare Systems,“ Academy Publisher, 2010.

[27] R. C. Seacord, S. Comella-Dorda, G. Lewis, P. Place ja D. Plakosh, „Legacy System Modernization Strategies,“ Carnegie Mellon University, Pittsburgh, PA, 2001.

[28] R. Peterson, „Relational Data Model in DBMS | Database Concepts & Example,“ Guru99, 04 03 2023. [Võrgumaterjal]. Available: https://www.guru99.com/relational-data-model-dbms.html. [Kasutatud 30 04 2023].

[29] J. Montemagno, T. Jain, G. Warren, Y. Victor, M. Veloso, J. Parente ja M. Wenzel, „Design a microservice domain model,“ Microsoft, 13 04 2022. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model. [Kasutatud 03 05 2023].

[30] V. Sharma, „Making enums smarter in C#,“ Medium, 23 08 2020. [Võrgumaterjal]. Available: https://medium.com/null-exception/making-enums-smarter-in-c-518108cdaa73. [Kasutatud 07 05 2023].

[31] Microsoft, „Implement a microservice domain model with .NET,“ 01 03 2023. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/net-core-microservice-domain-model. [Kasutatud 07 05 2023].

[32] Microsoft, „Events (C# Programming Guide),“ 12 04 2023. [Võrgumaterjal]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/. [Kasutatud 06 05 2023].

[33] T. Tritchew, „A Multi-Step Incremental Approach to Modernization (Step 2: Create Options — “How will we Modernize”),“ Medium, 02 06 2020. [Võrgumaterjal]. Available: https://medium.com/@tedt_39153/a-multi-step-incremental-approach-to-modernization-step-2-create-options-how-will-we-d4a938739253. [Kasutatud 09 05 2023].

[34] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2003.

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

Mina, Merily Adams

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

17.05.2023

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.

Lisa 2 – Vahemääruse alaliigi käskude töötleja meetodid

```
public async Task Handle(AddOtherRulingsCommand command, CancellationToken
cancellationToken)
    {
      var proceeding = await _proceedingRepository.GetByProceedingNrAsync(
         command.ProceedingNr, cancellationToken)
             .GetValueOrFailNotFound(command.ProceedingNr);

      if (!ProceedingStatus.OtherRulingCreationEnabledStatuses.Contains(
        proceeding.Status))
            throw BusinessRuleViolationException.
              Translated("proceeding-view.draft-other-rulings.
              error.unenabledProceedingStatus");

      if (!(await
     _proceedingAuthorizationService.CurrentUserIsRegistrarOfProceeding(
     command.ProceedingNr, cancellationToken)).IsAuthorized)
           throw BusinessRuleViolationException
             .Translated("ruling.error.compilerNotRegistrar");

      var authenticatedUser = _currentUserService.AuthenticatedUser;

      var activeDepartment = await _departmentRepository.GetByIdAsync(
        authenticatedUser.DepartmentId, cancellationToken)
            .GetValueOrFailNotFound(authenticatedUser.DepartmentId);

      await AddNewOtherRulingToProceeding(command.OtherRulingTypeId,
      proceeding, activeDepartment, cancellationToken);
    }


private async Task AddNewOtherRulingToProceeding(int otherRulingTypeId,
Proceeding proceeding, Department activeDepartment, CancellationToken
cancellationToken)
    {
      var otherRulingType = await _classifierRepository
          .GetClassifierValueAsync(otherRulingTypeId, RulingType.Descriptor,
       cancellationToken);

       var otherRulingNr = await
       _rulingNumerator.GenerateNumberAsync(cancellationToken);

       if (!otherRulingType.IsOtherRulingAddingEnabled)
            throw BusinessRuleViolationException
             .Invariant($"Unsupported ruling type '{otherRulingType.Name}'");

       var otherRuling = RulingType.CreateRuling(otherRulingType,
       otherRulingNr, activeDepartment.Id, _dateTime.LocalNow);

       if (otherRuling == null)
```

```
            return;

        proceeding.AddRuling(otherRuling);
    }
```