TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Teadmussüsteemide õppetool

# Elioni toodete valdkonna veebiteenuste parendamine

Magistritöö

Üliõpilane: Peeter Karolin
Üliõpilaskood: 111708IABM
Juhendajad: Jaak Tepandi
Raino Kolk

Tallinn
2014

## Autorideklaratsioon

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

(*kuupäev*) (*allkiri*)

# Elioni toodete valdkonna veebiteenuste parendamine

Annotatsioon

Käesoleva magistritöö eesmärkideks on Elioni toodete müüki ja haldust toetava keskse süsteemi ja selle poolt teeninduskanalite rakendustele pakutavate teenuste tehniliste probleemide lahendamiseks sobiliku lahenduse strateegia valimine ning lähtudes valituks osutunud lahenduse strateegiast, strateegia esimese etapi realisatsiooni kavandamine.

Kavandamisel analüüsitakse ja võrreldakse lahenduse erinevate aspektidega seotud võimalikke valikuid, mille seast vastavalt kontekstile sobivaim valitakse ja põhjendatakse.

Antud tööl on kaks olulisemat tulemust. Esiteks on analüüsi ja võrdluse resultaadina leitud eelpool nimetatud keskse süsteemi probleemide lahendamiseks sobiv lahenduse strateegia. Teiseks on lähtudes valituks osutunud strateegiast kavandatud lahenduse esimene etapp. See sisaldab endas abstraktsioonikihi rakenduse arhitektuuri ning veebiteenuste liidese disaini. Kokkuvõttes tuuakse välja ka soovitused tulevikuks.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 64 leheküljel, 5 peatükki ja 5 joonist.

# Elion's product domain web services improvement

Abstract

Main goals of this thesis are to choose an improvement strategy for Elion’s product catalogue and customer management system and its services, which currently contain technical debt, and based on chosen strategy design the first stage of the implementation.

During the design process alternative choices concerning different aspects of the solution are analysed and compared.

This thesis has two substantial results. Firstly, as a result of the analysis process, a best suitable improvement strategy is chosen. Secondly, based on the chosen strategy, first stage of the implementation is designed. This includes the design of the abstraction layer application architecture and its web service interface. In the summary, recommendations for the future development are also proposed.

The thesis is written in Estonian and contains 64 pages of text, 5 chapters and 5 figures.

# Lühendite ja mõistete sõnastik

**CI** ***Continuous Integration***

Pidev integratsioon on tarkvaraarenduse praktika, kus tiimi liikmed oma tööd pidevalt integreerivad. Iga integratsiooni kontrollitakse automaatse redaktsiooniga (ingl. k. *build*), et tuvastada integratsiooni vigu võimalikult varakult. (Fowler, 2006)

**HTTP** ***Hypertext Transfer Protocol***

Serverite ja klientide vahel hüperdokumente edastada võimaldava rakendusprotokolli nimi (IT terministandardi sõnastik, 01.05.2014).

**JSON** ***JavaScript Object Notation***

JSON on kergekaaluline andmevahetusformaat, mis on programmeerimiskeelest sõltumatu (JSON, 01.05.2014).

**PL/SQL** ***Procedural Language/Structured Query Language***

Programmikeel firmalt Oracle, mida kasutatakse Oracle'i andmebaasihalduri poolt täidetavate trigerite ja salvestatud protseduuride kirjutamiseks. Kasutatakse ka andmebaasist SQL päringuga saadud andmetele täiendava töötluse lisamiseks. (Vallaste, 2014)

**REST** ***Representational State Transfer***

Tarkvara arhitektuuri stiil hajutatud hüpermeedia süsteemide jaoks, nagu näiteks veeb (Fielding, 2000). REST mõiste on täpsemini lahti seletatud peatükis 4.

**RPC** ***Remote Procedure Call***

Andmetöötluseks ressursside arvutivõrgu kaudu hankimise protsess (IT terministandardi sõnastik, 01.05.2014).

**SI** ***Soovitusindeks***

Soovitusindeks on indeks, mida arvutatakse ühe küsimuse põhjal: "Kui tõenäoliselt Te soovitaksite firma X teenuseid oma sõpradele-tuttavatele?"

Hinnanguskaalal 0-10 peetakse kindlateks soovitajateks 9-10 palli andnuid ja mittesoovitajateks 0-6 palli andnuid. Soovitusindeks arvutatakse lihtsa lahutustehtena, kus soovitajate osakaalust lahutatakse mittesoovitajate osakaal. (Kas üks küsimus…, 01.05.2014)

**SLA** ***Service Level Agreement***
Leping teenusepakkuja ja kasutaja vahel, kus on kirjas lepingu kehtivusaja vältel oodatav teenusekvaliteet (Vallaste, 2014).

**SOAP** ***Simple Object Access Protocol***
SOAP on kergekaaluline protokoll eesmärgiga vahetada struktuurset informatsiooni hajutatud keskkonnas (SOAP Version …, 2007).

**SQL** ***Structured Query Language***
Andmekäitluskeel, millega kasutajad saavad andmeid andmebaasist võtta ja neid ka modifitseerida (IT terministandardi sõnastik, 01.05.2014).

**UML** ***Unified Modeling Language***
Üldotstarbeline noteeringukeel keerulise tarkvara, peamiselt suurte objektorienteeritud projektide spetsifitseerimiseks ja visualiseerimiseks (Vallaste, 2014).

**URL** ***Uniform Resource Locator***
Universaalne nimetamisviis, mis näitab sobivat võtuprotokolli, dokumendi asukohasõlme nime Internetis, selle sõlme failikataloogi ja dokumendi identifikaatorit (IT terministandardi sõnastik, 01.05.2014).

**XML** ***Extensible Markup Language***
XML on märgistuskeel andmete struktuurseks esitamiseks.

**WSDL** ***Web Services Description Language***
XML keelel põhinev keel Veebiteenuste kirjeldamiseks (Web Services Description Language, 2007).

# Jooniste nimekiri

# Sisukord

# Sissejuhatus

Paljudes ettevõtetes eksisteerib n-ö suuri pärandsüsteeme, mida on pikka aega arendatud ja mis sisaldavad seetõttu suure tõenäosusega palju tehnilist võlga. Selliste omadustega süsteem on olemas ka Elionis. Elioni puhul on tegemist rätsepatööna loodud keskse süsteemiga, mis sisaldab tootekataloogi ning toetab kliendihaldust. Antud süsteem on aja jooksul kujunenud asjatult keerukaks, mistõttu on seda piisavalt efektiivselt ja kvaliteetselt raske edasi arendada ning liigne keerukus mõjub halvasti ka süsteemi üldisele jõudlusele.

Nimetatud keskse süsteemi üheks suureks funktsionaalsuse osaks, millele antud magistriöös keskendutakse, on ärilises mõttes toetada ettevõtte poolt pakutavate toodete müüki ja haldust erinevates teeninduskanalites nagu näiteks esindus ja veebi iseteenindus. Selleks pakub keskne süsteem tehniliselt erinevate teeninduskanalite rakendustele teenuseid. Elion soovib seejuures võimalikult paljude toodete müügi ja haldusega seotud tegevusi nendes rakendustes võimaldada. Uute toodete turule toomisel on konkurentsieelise saamiseks täiendavalt oluline ka aeg, mille jooksul ettevõte suudab uue toote müüki mitmes teeninduskanalis võimaldada.

Eelnevalt välja toodud soove ei ole keskses süsteemis ja selle poolt pakutavates teenustes einevate probleemide tõttu võimalik enam piisavalt hästi rahuldada. Ettevõttes on otsustanud probleemidele lahendus leida. Seejuures on täiendavalt otsustatud, et antud keskse süsteemi ja tema teenuste parendus viiakse läbi rätsepatööna, mitte ei asendata vastavat süsteemi osa valmis karbitootega.

Käesolev magistritöö on kirjutatud vastava parendusprojekti algusfaasi ajal ning selle projekti raames. Tervikuna on antud projekt väga mahukas, mida teostatakse koostöös Elioni IT, äri poole ning väliste IT partneritega. Selle tõttu piirdub antud magistritöö väiksema osaga koguprojektist. Töö autor on Elionis vastutaja antud töös käsitletava osa raames tehtavate otsuste eest. Kuna selles projektis osaleb palju inimesi, siis mõned otsused võeti vastu koostöös projektiga seotud arendajatega.

Käesoleva töö eesmärgid on.

- Valida ja põhjendada lahenduse strateegia, kuidas parendust üldiselt läbi viima hakata.
- Lähtudes valituks osutuvast lahenduse strateegiast, kavandada strateegia esimese etapi lahenduse realisatsioon.
    - Lähtudes muudatus läbi abstraktsiooni strateegiast, püstitada abstraktsioonikihti realiseeriva rakenduse arhitektuurile nõuded ja teha nõuetest lähtuvad valikud, millest moodustub selle rakenduse arhitektuur.
    - Seada veebiteenuste liidesele kui abstraktsioonikihile oodatavad omadused ning teha liidese disaini puhul valikuid, mis neid omadusi tagavad.

Püstitatud eesmärkideni jõudmiseks analüüsitakse ja võrreldakse erinevates kavandamise etappides võimalikke alternatiivseid valikuid, millest lähtudes kontekstist sobivaimad valitakse.

Töö oodatava tulemusena valitakse ja põhjendatakse strateegia, kuidas antud süsteemi osa parendust läbi viia. Täiendavalt luuakse lähtudes valituks osutunud muudatus läbi abstraktsiooni strateegiast nõuetele vastav abstraktsioonikihi rakenduse arhitektuur ning oodatud omadusi pakkuv veebiteenuste liides ehk abstraktsioonikiht.

Käesolevas töös on viis peatükki. Esimeses peatükis täpsustatakse probleemi ja tuuakse välja töö kontekst. Teises peatükis käsitletakse lahenduse strateegia valikut. Kolmas peatükk keskendub abstraktsioonikihi rakenduse arhitektuuri kavandamisele. Neljas peatükk käsitleb veebiteenuste liidese disaini. Viiendas peatükis antakse töös tehtud valikutele realisatsioonist lähtuv hinnang.

# 1. Probleemi täpsustus ja töö kontekst

Järgnevas peatükis käsitletakse lühidalt käesoleva töö konteksti ning tuuakse detailsemalt välja antud töö skoobiga seotud probleemid. Kõigepealt kirjeldatakse konteksti ärilisest ja seejärel tehnilisest aspektist.

## 1.1 Äriline kontekst ja probleemid

Elion on teenindusettevõte, mis müüb tooteid ja kaupu. Kaupade all on seejuures mõeldud füüsilisi esemeid ning toodete all immateriaalseid hüvesid. Tihti käsitletakse ka kaupu toodetena. Küll aga keskendub antud töö just toodete kui immateriaalsete hüvede müügile ja haldusele, mistõttu eristatakse selguse huvides kauba mõistet eraldi.

Käesolevas töös ei keskenduta äriliste probleemide puhul „mis“ küsimusele, vaid „kuidas“ küsimusele. Seega eeldatakse, et tehakse juba õiget asja ja seejuures käsitletakse vaid probleeme, mis on seotud äriliste vajaduste realiseerimisega. Kuna antud töö keskendub toodete müügile ja haldusele, siis välja toodavad ärilised probleemid ja lahendused on samuti seotud vaid selle valdkonnaga. Teised probleemid ei oma selles kontekstis tähendust. Selline kitsendamine on vajalik kompaktsuse säilitamiseks ning aitab paremini probleemide lahendamisel fookust hoida.

Ettevõttel on toodete müügi- ja haldustegevuste teostamiseks võimalik kasutada mitut teeninduskanalit: esindus, telefonimüük ja e-kanal, mis hõlmab endas veebi iseteenindust, televisiooni ning tulevikus potentsiaalselt ka mobiilirakendusi. Täna ettevõte juba müüb ja võimaldab hallata oma fookustooteid peaaegu kõigis teeninduskanalites. Sellegipoolest eksisteerib ka selliseid tooteid, mida saab praegu ainult esinduses või telefoni teel müüa ja hallata, kuid mille tellimine ja haldamine ka teistes teeninduskanalites võimaldaks ettevõtte müügi efektiivsust tõsta ning muudaks klientide elu mugavamaks ja kiiremaks.

Ettevõtte soov on võimaldada uusi tooteid lihtsalt ja laiaulatuslikult turule tuua, muutes nad müüdavaks ja hallatavaks võimalikult paljudes teeninduskanalites. Seejuures konkurentsieelise saamiseks tuleb seda teha küllalt lühikese aja jooksul, kuna ettevõte tegutseb valdkonnas, kus vajadused ja turg võivad ootamatult muutuda ning nendele muudatustele on vaja kiiresti reageerida.

Teisest küljest peab ettevõte oluliseks ka kvaliteeti. Siinkohal tuleb mainida, et toodete toimivuse kvaliteeti, mis on samuti oluline, antud töös ei käsitleta. Skoobis on toote müügi- ja haldustegevuse kvaliteet. Vastavate tegevuste kvaliteeti mõõdetakse ning selleks kasutab ettevõte klientide soovitusindeksit.

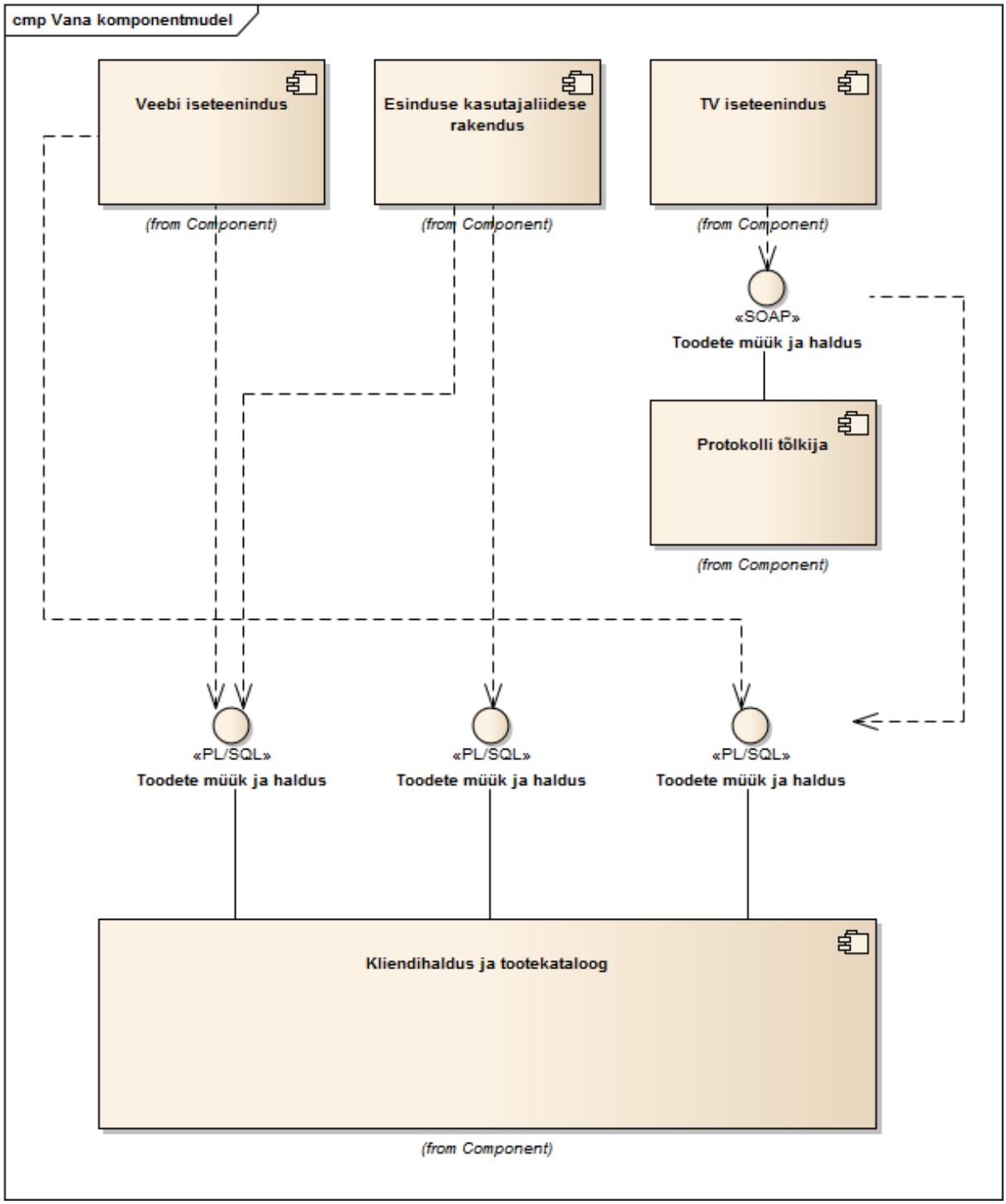
Eelpool kirjeldatud taustast ja äripoole soovidest lähtudes saab välja tuua tänased probleemid. Nimelt ei ole praegu võimalik kõiki tooteid, mida oleks efektiivne pakkuda ja hallata näiteks iseteeninduslikes kanalites, soovitud teeninduskanalites müüa. Samuti pole võimalik uusi tooteid äriliselt piisavalt kiiresti turule tuua. Need probleemid on omakorda tingitud sellest, et realisatsiooniks vajalik tarkvaraarendus võtab enamasti liiga palju aega. Lisaks tekkinud olukorrale, kus ettevõte võib kaotada konkurentsieelist, lisanduvad siia ka tarkvara arendusega seotud suured kulud.

Täiendavalt ei ole toote müügi- ja haldustegevuste kvaliteet jõudnud veel soovitud tasemini ja vajab seega parendamist. Tihti on selle põhjus vastavaid protsesse toetavate rakenduste tehnilises pooles. Näiteks esineb veebi iseteeninduses liiga palju tehnilisi tõrkeid ning jõudlus on samuti seejuures madal. Sellekohane info pärineb tagasiside küsitlustest, mida klientidelt küsitakse. Veebi iseteeninduse tagasisides on välja toodud ka muid probleeme, kuid tehnilised probleemid on seal praegu suure osakaaluga ning seetõttu on see probleem prioriteetne. Negatiivne tagasiside on tulnud ka ettevõtte teeninduse poole pealt. Teenindajad tajuvad samuti tehnilisi probleeme nende jaoks kasutada olevates rakendustes. Viimased ei pruugi küll otseselt kliendile välja paista, kuid kaudne mõju klienditele siiski eksisteerib. Näiteks kui süsteemi jõudlus pole piisavalt hea, siis tuleb vastavad ooteajad kliendiga suheldes kuidagi ära siluda.

Nagu öeldud, on välja toodud ärilised probleemid oma olemuselt tänaste tehniliste probleemide tagajärjeks. Seega juurpõhjused on tegelikult hoopis tehnilised. Selle peatüki järgnevates jaotistes tuuakse täpsemalt välja just need tehnilised probleemid.

## 1.2 Tehniline kontekst

Enne tehniliste probleemide käsitlemist on hea välja tuua rakenduste omavahelisi seoseid väljendav komponentdiagramm (vt. Joonis 1), mis annab esmase kiire ülevaate tänasest tehnilisest kontekstist.




**Joonis 1 Vana komponentmudel**

Antud joonisel on süsteemid nimetatud nende funktsioonide järgi. Samuti on lihtsustuse huvides süsteemide liidesed nimetatud üldise nimega. Jooniselt 1 on näha, et hetkel on kasutusel üks keskne tootekataloogi ja kliendihaldusega seotud funktsionaalust pakkuv komponent, mis pakub erinevaid liideseid erinevatele tarbijatele. Lisaks on näha, et tehnoloogiliselt on veebi iseteeninduse ja esinduse kasutajaliidese rakendus liidestatud keskse süsteemiga kasutades andmebaasi funktsioone ja protseduure ning vastavat

andmebaasispetsiifilist protokolli. TV iseteenindus on liidestatud läbi tarkvara, mis tõlgib andmebaasispetsiifilise protokolli SOAP protokolliks. Kõik kolm keskse süsteemi poolt pakutavat põhiliidest on sama nimega, kuna sellega soovib autor rõhutada ebaühtsust sarnaste tegevuste vaates. Täpsemalt on seda ebaühtsust analüüsitud järgmises alampeatükis 1.3.

## 1.3 Tehnilised probleemid

Jooniselt 1 on näha, et kõik teeninduskanaleid toetavad rakendused kasutavad täna lõppkokkuvõttes ikkagi ainult üht keskset süsteemi. Üldarhitektuuri vaates on olukord tegelikult hea. See võiks olla veelgi hullem, kui kliendihaldussüsteeme või tootekatalooge oleks mitu. Küll aga tulevad probleemid välja detailsemal tasemel: koodi struktuur, andmemudel ja kasutatavad tehnoloogiad. Järgnevalt tuuakse välja nende alamosadega seotud suuremad põhilised probleemid, mille lahendamiseks antud magistritöö lahenduse strateegia pakub ning selle strateegia kontseptsiooni toimivust tõestab (ingl. k. *proof of concept*). Lisaks tuleb ära mainida, et kirjeldatavad probleemid puudutavad just keskset süsteemi. See süsteem on neist kõige vanem ning pika arendusaja tõttu on selles tekkinud tehnilist võlga.

### 1.3.1 Koodi struktuur

Enne koodi probleemide käsitlemist tasub lühidalt ära mainida, miks on üldiselt oluline vältida halva koodi kirjutamist, millest näiteks antud jaotises kirjeldatud probleemid tulenevad. Halva koodi haldamine ja selles muudatuste tegemine võtab palju aega, selles tekib lihtsamini vigu ning neid tekkinud vigu on raske parandada. Lõpuks viivad kõik nimetatud probleemid arendusmeeskonna produktiivsuse alla. (Martin, 2009, 4-7)

Antud töös käsitletud keskses süsteemis on üheks probleemiks koodi dubleerimine. Sisuliselt samade tegevuste tegemiseks või andmete pärimiseks on kohati kirjutatud mitu erinevat üksteisest sõltumatut protseduuri. Nendes on olemas küll ühisosa, kuid see on kopeeritud kõikdesse vastavatesse protseduuridesse. Seetõttu tekib tihti toodangukeskkonnas vigu, kui mingi arenduse käigus oli vaja muuta seda protseduuride ühisosa. Mõni kopeeritud osaga protseduur jääb suure tõenäosusega kahe silma vahele.

Teisest küljest on probleeme ka korduvkasutatava koodiga, mis on ühekordselt kirjutatud. Nendes kohtades on kokku koondatud liiga palju erandeid, mida reaalselt on väga harva vaja, kuid nende eranditega arvestatakse siiski ka siis, kui nad on mittevajalikud. Selle tagajärjel muutub vastava koodi käivitamine enamus juhtudel ebamõistlikult aeglaseks.

Täiendavalt võib välja tuua probleemi, et tihti on rakenduse tuumloogika üles ehitatud ühe konkreetse kasutajaliidese spetsiifikast lähtudes. Seega on rakenduse loogika liiga seotud kasutajaliidesega. Tavaliselt on selleks kasutajaliideseks esindustes kasutatava rakenduse kasutajaliides. Tihti ei sobi aga sama lähenemine e-kanalitele, mistõttu tuleb hakata kuhugi taas erandeid realiseerima.

Viimaseks keskse süsteemi koodiga seotud oluliseks probleemiks on automaattestide puudumine. Selle tõttu ei saa muudatuste tegemisel olla piisavalt kindel, et kõik eelnev funktsionaalsus jääb sama moodi tööle. Kuna süsteem on suur, siis iga muudatuse puhul terve süsteemi käsitsi üle testimine on mõeldamatu, mistõttu tekivad arenduste tagajärjel vead. Automaattestide puudumine on ilmselt omakorda üks põhjustest, miks pole varem koodi arenduste käigus refaktoreeritud.

### 1.3.2 Andmemudel ja süsteemi liides

Antud süsteemi praegune andmemudel on pika aja jooksul välja kujunenud. Seejuures seisneb probleem selles, et erinevate täienduste sisse viimisel pole tihti üle vaadatud olemasolevat andmemudelit ning selle tõttu pole vanu osi muudetud uutele vajadustele sobivaks. Tulemuseks on keeruline ja kohati sarnaseid asju erinevat moodi nimetav ja dubleeriv andmemudel. See omakorda muudab keskse süsteemi arenduse taas üha keerukamaks.

Eelnevaga on seotud veel täiendav probleem, et see keeruline andmemudel paistab välja ka süsteemi poolt pakutavates teenustes ning seeõttu peavad selle keerukusega tegelema kõik välised tarbijad, kes selle süsteemiga liidestuvad. Seepärast on iga tarbija enda rakenduses loonud keskse süsteemiga suhtlemiseks teisenduskihi, mis antud keerukust võimalikult palju ära peidab. Neid teisenduskihte pole aga tehtud üksteisega kooskõlastatult ning iga väline tarbija on selle enda jaoks erinevalt lahendanud. See omakorda muudab arendused, mis erinevaid tarbijaid mõjutavad, keerukamaks ja ajamahukamaks, kuna muudatusi tuleb teha paljudes kohtades. Nende teisenduskihtide rohkuse probleem on ilmselt tingitud kokkulepete ja tervikvastutuse puudumisest, kuid juurpõhjus on siiski selles, et keskne süsteem pakub juba algselt liidest, mis keerukat andmemudelit välja peegeldab.

Keerukas andmemudel avaldab mõju ka uute arendajate sisseelamisele. Autor on ettevõttes töötamise aja jooksul täheldanud, et tihti tuleb uutele inimestele seletada samu erandeid ja muid ebakõlasid, mis võiksid oluliselt lihtsamalt lahendatud olla. Aeg-ajalt korraldatakse ka vastavaid koolitusi, kus seletatakse olemasoleva süsteemi andmemudelit.

Andmemudeli vaates puudub ka üks kokkulepitud domeeni terminoloogia erinevate süsteemide vahel. Näiteks koodi keelusest lähtuvalt on see keskses süsteemis eestikeelne ning erinevates teeninduskanalite rakendustes inglisekeelne. Eelistatud on seejuures inglise keel, mis tagab suurema arendajate valiku – saab kasutada ka välisriikide arendajaid. Lisaks on inglise keeles kirjutatud tarkvara potentsiaalselt müüdav näiteks TeliaSonera grupi sees. Küll aga pole taas erinevates teeninduskanalite rakendustes see terminoloogia ühte moodi inglise keelde tõlgitud. Keelelise erinevuse tõttu tekib ka erinevate rakenduste arendajatel omavahel suhtlemisel raskusi, kuna kogu aeg on vaja teada, kuidas üht ja sama mõistet erinevates süsteemides nimetatakse. Keelsuse probleem pikendab samuti uute arendajate sisseelamisaega.

### 1.3.3 Kasutatud tehnoloogiad

Antud keskne süsteem kasutab Oracle andmebaasisüsteemi ning äriloogika on realiseeritud PL/SQL programmeerimiskeeles. Tehnoloogilises vaates pole selles rakenduses PL/SQL laialdane kasutamine enam kõige parem valik. Täpsemalt on selle probleemi põhjendus lahti kirjutatud jaotises 3.2.2, kus valitakse ja põhjendatakse PL/SQLle sobivam alternatiiv.

# 2. Lahenduse strateegia

Antud peatükis käsitletakse lahenduse strateegiate üldiseid põhimõtteid ning valitakse nende seast välja üks, millega proovitakse läheneda eelmises peatükis kirjeldatud ärilistele probleemidele, mis on suuresti tingitud tehnilistest probleemidest.

## 2.1 Lähtepunkt

See jaotis kirjeldab antud töö kontekstis probleemide lahendamise lähtepunkti. Kõigepealt tuuakse välja kirjeldatud probleemide lahendamisega seotud varasemad otsused. Need otsused on tehtud enne käesolevat tööd ning nende põhjendused ei kuulu antud töö skoopi. Lisaks tuuakse välja piirangud, mis on praegusel hetkel teada ja millega probleemide lahendamisel arvestama peab.

### 2.1.1 Eelnevad otsused

Selle töö kontekstis on oluline ära märkida, et enne antud töö alustamist on tehtud eelanalüüs ja vastu võetud otsus, et kirjeldatud probleemid on piisavalt olulised ning olukord vajab tõepoolest parandamist. Seda on tõdenud nii ettevõtte äri kui ka IT pool. Seega on muudatuse läbiviimiseks olemas vastavate osapoolte toetus.

Kuna süsteemi tervikuna on vaja palju parendada, siis üheks variandiks oleks olnud juurutada tänase süsteemi asemel mingisugune uus valmis karbitoode. Küll aga on IT osakonna poolt eelnevalt otsustatud, et olemasolevat keskset süsteemi hakatakse hoopis ise parendama. Praegusel hetkel pole selge, kas selle tulemusel luuakse täiesti uus süsteem või viiakse olemasolevas süsteemis läbi põhimõttelised muudatused. Mõlema variandi puhul on tegu väga suure muudatusega. Karbitootest loobumise otsuse täpsemaid põhjuseid käesoleva töös põhjalikult ei käsitleta, kuna need ei kuulu antud töö skoopi. Lühidalt võib mainida, et siiski kaaluti ja hinnati mitut karbitoodet, kuid ükski neist ei osutunud erinevatel põhjusel piisavalt sobivaks. Probleemid seisnesid näiteks integratsiooni keerukuses, kvaliteedis ja tarkvara paindlikkuses.

### 2.1.2 Piirangud

Üheks antud töö piiranguks on standardkomponendid, mis on ettevõttes tehnilise arhitektuuri vaatest kokku lepitud. Need standardkomponendid hõlmavad süsteemide erinevaid osi ja kihte

ning need kujutavad endast valikuid, mille seast on lubatud süsteemide arendamisel vastavaid komponente valida. Seega peavad probleemide lahendamisega seotud tehnilised valikud sellest nimekirjast lähtuma.

Antud muudatuste tegemise perioodil on siiski vaja ka uusi ärivajadusi jätkuvalt realiseerida. Seega ei saa uusi arendusi tegemata jätta. See tähendab omakorda seda, et uue terviklahenduse elluviimist peab kindlasti olema võimalik jaotada väiksematesse etappidesse, mida järk-järgult toodangukeskkonda kanda saab.

## 2.2 Lahenduse strateegia valik

Antud alampeatükis käsitletakse võimalikke lahenduse strateegiaid ning valitakse nende seast sobivaim. Lisaks probleemidele on strateegia valiku puhul arvesse võetud eelmises alampeatükis kirjeldatud varasemalt tehtud otsused ning välja toodud piirangud.

Sarnaste probleemide lahendusi uurides on autor tuvastanud kolm potentsiaalset strateegia kandidaati:

1. Muudatus läbi abstraktsiooni (ingl. k. *Branch By Abstraction*).
2. Muudatus läbi versioonihalduse (ingl. k. *Feature Branch*).
3. Funktsionaalsuse lüliti (ingl. k. *Feature Toggle*).

Need kandidaadid rahuldavad kõik üldiseid nõudeid, mis vastavatele strateegiatele on püstitatud. Kõik need mustrid sobivad otsusega ise süsteemi parendada. Lisaks ei ole need mustrid sellise taseme mustrid, mis oleksid seotud konkreetsete tehnoloogiatega ning seetõttu on need ühilduvad ka ettevõttes kehtestatud standardkomponentide nimekirjaga. Lõpuks võimaldavad kõik need lähenemised ka järk-järgult muudatust toodangukeskkonda kanda.

Antud töö kontekstis sobib kõige paremini Hammanti (2007) ja Fowleri (2014) poolt kirjeldatud muudatus läbi abstraktsiooni strateegia. See muster pakub sobivat lahendust olukorras, kus soovitakse parendada või asendada üht pakkujat, millel on mitu tarbijat ning tarbijate ja pakkuja vahel on selgelt välja toodav liides. Antud töö kontekstis on samuti sellise olukorraga tegemist. Täpsemalt on see muster defineeritud läbi nelja sammu.

1. Loo tarbijate ja pakkuja vahele abstraktsioonikiht. Antud töö kontekstis asendaks abstraktsioonikiht keskse süsteemi ja teeninduskanalite vahel olevat teenuste kihti.

2. Vii kõikide tarbijate suhtlus vastava pakkujaga üle abstraktsioonikihi peale.

3. Realiseeri uus parendatud pakkuja, millega saab suhelda läbi abstraktsioonikihi, ning liidesta kõik tarbijad kasutama abstraktsiooni kihti, mida realiseerib siis juba uus pakkuja. Seejuures on väga oluline, et abstraktsioonikiht oleks kaetud automaattestidega, kuna see annab kindluse, et uus pakkuja töötab samamoodi nagu vana.

4. Eemalda vana pakkuja ja vajadusel ka abstraktsioonkiht uue pakkuja ja tarbijate vahel.

Fowler (2014), Hammant (2007), Humble (2011) ja Rehn (2012) rõhutavad eriti antud mustri tugevust, et selle puhul on tehniliselt muudatusi lihtne järk-järgult toodangukeskkonda kanda, mis on selle mustri eelis muudatus läbi versioonihalduse mustri ees. Versioonihaldussüsteemide vaates ei pea muudatus läbi abstraktsiooni mustri kasutamisel looma eraldi peaharust sõltumatut haru, millel muudatus läbi versioonihalduse põhimõte just põhinebki. Need autorid toovad välja ja ka töö autor on tööl kogenud viimase lähenemise puuduseks just asjaolu, et peaharust sõltumatute harudega tekivad versioonihaldussüsteemides probleemid muudatuste tagasi kandmisel peaharru. Head versioonihaldussüsteemid suudavad seda siiski suhteliselt hästi teha, kuid programmikoodi semantilisel tasemel jäävad need endiselt hätta. Mida suurem ja pikaajalisem on muudatus, seda suuremaks antud probleem muutub. Seega ei sobi muudatus läbi versioonihalduse muster väga suurte muudatuste jaoks. Antud töö kontekstis tehtav muudatus on aga tõepoolest suur ja seega on hea mõte vältida muudatus läbi versioonihalduse mustrit. Täiendavalt ei ole muudatus läbi versioonihalduse muster sobilik ka selle tõttu, et töös kirjeldatav parendus hõlmab mitut erinevat rakendust ning selle tõttu muutub selle põhimõtte kasutamine veelgi keerulisemaks – kõikide rakenduste versioonihaldussüsteemides tuleb luua eraldi harud. Seetõttu on hiljem kõikide rakenduste muudatuste peaharudesse kandmine veelgi riskantsem ja ajamahukam.

Üheks alternatiiviks on kasutada funktsionaalsuse lüliti mustrit, mille põhimõtet Fowler (2010) oma artiklis kirjeldab. Töö autori arvates ei ole see aga antud töö kontekstis üldpõhimõttena siiski piisavalt sobiv, kuna suurte muudatuste puhul muutub lüliti loomine keerukaks. Lisaks mainib Humble (2011), et funktsionaalsuse lüliti muster sobib paremini uute funktsionaalsuste loomiseks ilma versioonihaldussüsteemis lisaharu tekitamata, hoides

neid funktsionaalsusi peidetuna seni, kuni nad lõplikult valmis saavad. Seejuures muudatus läbi abstraktsiooni on just pigem mõeldud süsteemis suuremahuliste parenduste tegemiseks, mis ei ole pruugi olla uue funktsionaalsuse loomine.

Muudatus läbi abstraktsioonikihi põhimõtte puhul on Humble (2011) välja toonud ka reaalsed edulood, kus vahetati välja andmebaasiga suhtlemise raamistik ja kasutajaliidese loomise raamistik. Nende näidete põhjal on veelgi rohkem alust arvata, et antud muster sobib, kuna see on juba reaalselt ennast hästi töötavana tõestanud.

## 2.3 Skoobi täpsustus lähtuvalt valitud strateegiast

Eelmises peatükis valitud muudatus läbi abstraktsiooni põhimõtte tervikuna realiseerimine on suuremahuline projekt, mis ei mahuks piisava detailsusega antud töö skoopi. Seetõttu keskendutakse käesolevas magistritöös edaspidi mustris kirjeldatud esimesele sammule ja esimesele osale teisest sammust. Seega disainitakse antud töö raames abstraktsioonikiht keskse süsteemi ja teeninduskanaleid toetatavate rakenduste vahele ning pannakse idee toimivuse tõestamiseks üks teeninduskanali rakendus keskse süsteemiga suhtlema läbi abstraktsioonikihi. Selle osa realiseerimise tulemusena saab hinnata, kas uus abstraktsioonikiht on tõesti piisavalt hea ja seega valituks osutunud üldine strateegia on õige valik antud töö probleemide lahendamiseks.

Veebi iseteeninduse rakendus on esimene rakendus, mis hakkab keskse süsteemiga suhtlema läbi abstraktsioonikihi. Veebi iseteenindus valiti selle tõttu, et selle vajalik funktsionaalsus seoses toodete müügi ja haldusega on väiksem teenindajate rakenduse ja suurem televisiooni iseteeninduse rakenduse funktsinoaalsusest. Seega on veebi iseteenindus parajalt suur, et vastava töö tulemuse põhjal saab strateegia sobivust piisavalt täpselt hinnata. Veebi iseteeninduse valimise juures on positiivne veel asjaolu, et sellisel juhul saab abstraktsioonikihti luua kliendist lähtudes. Näiteks teenindajatele mõeldud rakenduse valimisel võib see hakata teenindajast lähtuma. Sellegipoolest tuleb siiski vältida konkreetse kasutajaliidese spetsiifilist abstraktsioonikihti, kuna ühest kasutajaliidesest sõltuvad teenused on üks praegustest probleemidest, mida kõrvaldada püütakse.

Täpsustamist vajab veel abstraktsioonikihi laius ja paksus. Keskse süsteemi funktsionaalsus tervikuna on küll suur, kuid seejuures keskendutakse sellele osale, mis tagab toodete müügi ja nende haldamiseks vajaliku funktsionaalsuse. Kuna esimese etapi eesmärk on pigem hinnata

uue abstraheeritud liidese headust, siis antud töö kontekstis on abstraktsioonikiht võimalikult õhuke, et saaks võimalikult kiiresti ja ilma palju jõudu kulutamata selle idee toimivust tõestada.

Edasi käsitletakse abstraktsioonikihti realiseeriva rakenduse arhitektuuri ning seejärel lisaks ka selle poolt pakutavaid abstraheeritud veebiteenuseid, sest muudatus läbi abstraktsiooni muster on oma olemuselt liiga üldine ja ei ütle midagi selle kohta, kuidas detailsemalt abstraktsioonikihti luua. Kuna see asjaolu on samuti idee toimuvuse tõestamiseks oluline, siis selle tõttu käsitletakse käeolevas töös ka antud detailsemat osa. Täiendavalt tuleb ära mainida, et kirjeldatud skoobi kitsenduse tõttu lahendab antud magistritöö keskse süsteemi teenustega seotud probleeme. Keskse süsteemi sisemise ülesehitusega seotud probleemid kuuluvad strateegia realiseerimise järgmistesse etappidesse.

# 3. Abstraktsioonikihi arhitektuur

Antud peatükis käsitletakse eelmises peatükis kitsendatud töö skoobi osa, mis puudutab abstraktsioonkihi loomist üldisel rakenduse tasemel. Täpsemalt püstitatakse ja põhjendatakse abstraktsioonikihi arhitektuuri puudutavad nõuded ning nendest nõuetest lähtuvalt valitakse ja põhjendatakse kasutatav tehnoloogia ja infrastruktuur, millel abstraktsioonikihti realiseeriv rakendus põhinema hakkab.

Arhitektuuri mõiste on subjektiivne ning sõltuvalt kontekstist on sellel palju definitsioone. Selguse huvides olgu öeldud, et antud peatükk keskendub erinevate arhitektuuri vaadetest lähtuvalt rakenduse arhitektuuri vaatele ning defineerimisel on lähtutud Fowlerist (2002, 1), kelle kohaselt on arhitektuur midagi, mida on raske muuta ja seega on hea võimalikult varakult vastavad otsused õigesti teha.

## 3.1 Nõuded abstraktsioonikihi arhitektuurile

Rakenduse arhitektuuri nõuete kaardistamisel on lähtutud töö alguses välja toodud tänastest ärilistest ja tehnilistest probleemidest ning juba täna Elionis toimivatest headest põhimõtetest. Kuna nõuded on püstitatud arhitektuurile, siis ei keskendu need detailidele. Kaardistamisel on püütud arvestada ka võimalike tuleviku vajadustega. Kuna tuleviku vajadusi ei ole kunagi võimalik täpselt ette teada, siis nende puhul on nõuded vähem täpsemad ning nende osakaal on ka väiksem. Pigem on püütud järgida põhimõtet, et rakendus oleks erinevates aspektides võimalikult paindlik muudatuste suhtes. See tähendab seda, et mitte nii väga keskenduda konkreetsetele võimalikele tuleviku vajadustele, vaid püüda säilitada paindlikkust suvaliste muudatuste suhtes, hoides erinevate osade vahel sõltuvused minimaalsetena. Seda sama nõuete kaardistamise paindlikkuse põhimõtet on soovitanud ka Jason Bloomberg (2013a, 4-8).

Antud alampeatükis esitatud nõuded tulenevad põhiliselt rakenduse tasemest, kuid siin on esitatud ka nõudeid, mille tegelik vajadus tuleneb teenuste disainist, kuid mis omavad mõju ka rakenduse arhitektuurile. Teenuste disaini on täpsemalt käsitletud peatükis 4. Nende nõuete, mille tegelik vajadus selgub töö hilisemas osas, põhjendused on vastavalt viidatud.

Nõuete kaardistamisel on aluseks võetud ISO/IEC 25010 standardi tootekvaliteedi mudeli karakteristikud ja alamkarakteristikud (Systems and software engineering …, 2011). Seda kasutatakse kui erinevate nõuete aspektide kontrollnimekirjana, et piisavalt veenduda, et midagi olulist pole märkamata jäänud. Seega on arhitektuuri nõuded just selle mudeli alusel kategooriatesse jaotatud. Igat kategooriat on käsitletud eraldi. Igas kategoorias on esitatud nõuded ja põhjendused, miks sellised nõuded on vajalikud. ISO tootekvaliteedi mudelist on välja jäetud kasutatavuse ja turvalisusega seotud nõuded. Kasutatavusest on loobutud, kuna antud rakendus on tehnilise iseloomuga, millel puudub kasutajaliides, ning seega puudub vajadus kasutatavuse nõuete järele. Turvalisusega seotud nõuded on välja jäetud seepärast, et need klassifitseeruvad ettevõtte jaoks sensitiivseks infoks neid ei soovitud antud töös avaldada.

Edasi esitatakse nõuded, mis on alampeatükkidesse jaotatud ISO standardi alusel. Need nõuded esitatakse järgmise struktuuri alusel.

| |
|---|
| **ID** – vastava nõude unikaalne identifikaator. Selle identifikaatori alusel viidatakse valikute alampeatükis vastavale nõudele, mida konkreetse valikuga rahuldatakse. |
| **Nõude sõnaline kirjeldus** |
| **Põhjendus** – siin tuuakse välja põhjendus, miks antud nõue oluline on. |

### 3.1.1 Rakenduse baasfunktsionaalsus

Siin jaotises on välja toodud ainult need rakenduse funktsionaalsusega seotud olulised nõuded, mis on sõltumatud rakenduse ärilisest funktsionaalsusest. Selles jaotises on nad välja toodud selle tõttu, et need on rakenduse arhitektuuri vaatest siiski olulised. Seega ei kajasta järgnevad nõuded rakenduse täielikku funktsionaalsust.

| | |
|---|---|
| **ID** | F1 |
| **Kirjeldus** | Rakenduse logid peavad järgima syslogi protokollist tulenevat formaati. |
| **Põhjendus** | Syslog on üks jaotises 2.1.2 kirjeldatud standardkomponent, mida ettevõttes kasutatakse rakenduste logide saatmiseks rakendusserveritest kesksesse logidehoidlasse, kus neid vastavalt juriidilistele nõuetele kindel aeg hoitakse. |

| | |
|---|---|
| | Antud juriidilised nõuded on väljaspool käesoleva töö skoopi. |

| | |
|---|---|
| **ID** | F2 |
| **Kirjeldus** | Rakendus peab logima kõiki sisse tulevaid päringuid, enda poolt välja tehtud päringuid ja nende mõlema jooksul tekkinud vigu. |
| **Põhjendus** | Tegu on Elionis heaks tavaks kujunenud minimaalse tehnilise logimise nõudega, mida iga rakendus peab rahuldama. Sellised andmed on siiani olnud piisavad, et analüüsida võimalikke vigu ja nende põhjuseid. |

| | |
|---|---|
| **ID** | F3 |
| **Kirjeldus** | Rakendus peab pakkuma automatiseeritud käivitatavat spetsifikatsiooni (ingl. k. *executable specification*) enda poolt pakutavatele veebiteenustele. |
| **Põhjendus** | Käivitatav spetsifikatsioon on nagu dokumentatsioon, mis on kogu aeg tegeliku funktsionaalsusega kooskõlas, kuna käivitatav spetsifikatsioon on ühtlasi ka automaattest. See on seega vajalik testitavuse tagamiseks ning selleks, et käivitatava spetsifikatsiooni tulemuse põhjalt saaks uus arendaja, kes peab vastava rakenduse veebiteenuseid kasutama hakkama, kiire ülevaate kogu funktsionaalsusest ning testi tulemustest saab ta infot, kas mingi konkreetne funktsionaalsus on praegu toimiv või on seal vigu. |

### 3.1.2 Soorituse tõhusus

| | |
|---|---|
| **ID** | MF1 |
| **Kirjeldus** | Rakendus peab olema võimeline töötlema suvalisel ajahetkel vähemalt 30 paralleelset pöördumist, ilma et rakendus muutuks vähemalt ühele pöördumisele kättesaamatuks. |
| **Põhjendus** | See nõue lähtub veebi iseteeninduse praegusest koormusest, millele on kindluse mõttes lisatud varu. Seega on see nõue oluline toodangukeskkonnas |

| | |
|---|---|
| | toimimiseks. Varu on lisatud selleks, et oleks kaetud ka lähiaja tuleviku vajadus, kuna päringute arv on kasvavas trendis. |

| | |
|---|---|
| **ID** | MF2 |
| **Kirjeldus** | 80% enimkasutatud pöördumistest ei tohi võtta üle 0,5 sekundi aega, kui rakendusel puudub koormus. |
| **Põhjendus** | Kuna tegu on e-kanalite kasutajaliidest toetava rakendusega, siis nende rakenduste kasutatavus kannatab, kui antud rakenduse pöördumised hakkavad liiga palju aega võtma. Näiteks väidab kasutatavuse ekspert Jakob Nielsen (2010), et 10 sekundiliste viiteaegade juures hakkavad kasutajad suure tõenäosusega lehelt lahkuma. 80% on nõudes sees seetõttu, et vähe kasutatud päringuid pole alati mõttekas nii heale tasemele optimeerida. |

| | |
|---|---|
| **ID** | MF3 |
| **Kirjeldus** | 80% enimkasutatud pöördumistest ei tohi võtta üle 1 sekundi aega, kui rakendus töötleb paralleelselt 30 pöördumist. |
| **Põhjendus** | See nõue on vajalik selleks, et rakendus ei muutuks aeglaseks, kui see on koormuse all. Tegu on koormustundlikkuse nõudega (ingl. k. *load sensitivity*), mille olulisust on jõudlusest rääkides rõhutanud ka Fowler (2002, 7-8). |

| | |
|---|---|
| **ID** | MF4 |
| **Kirjeldus** | Ükski pöördumine ei tohi suvalisel ajahetkel võtta aega üle 5 sekundi |
| **Põhjendus** | Eelmistest kahest nõudest jäid välja vähekasutatud pöördumised. See nõue katab ka need pöördumised ära. Selle nõude vajadus seisneb selles, et pikalt kestvad päringud oleksid ka kontrolli all. Kuna antud rakenduse poolt pakutavate veebiteenuste iseloom pole analüütilist tüüpi, siis ei tohikski need päringud nii |

| | |
|---|---|
| | palju aega võtta. Kui mingisugune pöördumine võtab juba üle 5 sekundi aega, siis tuleb hakata vastavat pöördumist optimeerima. |

| | |
|---|---|
| **ID** | MF5 |
| **Kirjeldus** | Rakendust peab olema võimalik võimalikult lihtsalt ja ilma koodi muutmata skaleerida. |
| **Põhjendus** | See nõue tuleneb sellest, et tulevikus peab antud rakendus pakkuma teenuseid ka teistele teeninduskanalite rakendustele. Sellisel juhul on oodata koormuse kasvu ning selle tõttu peab rakendust olema võimalik lihtsalt ja kiiresti skaleerida. Kiiruse tõttu ei tohi see tähendada koodi muudatust. |

| | |
|---|---|
| **ID** | MF6 |
| **Kirjeldus** | Rakenduse redaktsioon (ingl. k. *build*) peab olema võimalik valmis ehitada vähemalt 5 minutiga. |
| **Põhjendus** | Elioni arendusprotsessis kasutatakse CId. Selle tõttu on oluline, et redaktsiooni valmimine, mis hõlmab ka automaattestide käivitamist, oleks piisavalt kiire. Arvulises vaates pole välja toodud 5 minutit isegi nii oluline. Pigem on oluline nõue, et kui redaktsiooni valmimine läheb teatud ajalisest piirist üle, tuleb hakata paremat lahendust otsima. |

### 3.1.3 Ühilduvus

| | |
|---|---|
| **ID** | MF7 |
| **Kirjeldus** | Rakendus peab ilma konfliktideta töötama koos teiste samas serveris paiknevate rakendustega. |
| **Põhjendus** | Antud nõue on oluline selle tõttu, et ühel konkreetsel tehnilisel platvormil ei tööta rakendused kunagi täielikus isolatsioonis. Seetõttu peab loodav rakendus |

| | |
|---|---|
| | sellega arvestama. See nõue on tingitud nõudest MF16. |

| | |
|---|---|
| **ID** | MF8 |
| **Kirjeldus** | Rakenduse poolt pakutavad veebiteenused peavad kasutama protokolli, mis järgib REST arhitektuuri stiili ühtse liidese piiranguid. |
| **Põhjendus** | REST arhitektuuri stiil seab lisaks muudele piirangutele ka teatud piirangud kasutatavale protokollile. Protokolli valik on aga otseselt seotud rakenduse arhitektuuri tasemega. Selle tõttu on antud nõue siin alampeatükis. REST arhitektuuri stiili enda kasutamise vajadus tuleneb aga veebiteenuste disainist, mida käsitletakse peatükis 4. Seal selgub, miks on autor valinud probleemide lahendamiseks REST arhitektuuri stiili. |

| | |
|---|---|
| **ID** | MF9 |
| **Kirjeldus** | Rakenduse poolt pakutavad veebiteenused peavad kasutama protokolli, mida on võimeline kasutama nii veebi iseteeninduse, televisiooni iseteeninduse, mobiili kui ka teenindajatele mõeldud rakendus. |
| **Põhjendus** | See nõue tuleneb antud abstraktsioonikihi rakenduse eesmärgist, et see peab olema lõpuks kõikide tarbijate ja pakkuja rakenduse vahel. Lisaks on nõudesse tuleviku paindlikkuse aspekti huvides lisatud ka mobiili rakendusega ühilduvuse vajadus. |

### 3.1.4 Töökindlus

| | |
|---|---|
| **ID** | MF10 |
| **Kirjeldus** | Rakendus peab olema kättesaadav 99,72% ajast. |
| **Põhjendus** | Vastava nõude protsent tuleneb sellest, et teenus, mille toimimiseks antud rakendus on hädavajalik, on äriliselt oluline ning selle SLA määrab nii kõrge protsendi. Kuna antud rakendus toetab vähemalt üht e-kanalit, siis eraldi tööaega |

| | |
|---|---|
| | pole määratud ning seda mõõdetakse 24/7 lõikes. |

| | |
|---|---|
| **ID** | MF11 |
| **Kirjeldus** | Rakendusel tohib olla maksimaalselt 2 mitteplaneeritud katkestust kuus. |
| **Põhjendus** | Sarnaselt eelmisele nõudele on see nõue samuti tingitud toetatava teenuse SLAst. |

| | |
|---|---|
| **ID** | MF12 |
| **Kirjeldus** | Maksimaalne lubatud katkestuse kestvus on 1 tund katkestuse kohta. |
| **Põhjendus** | See nõue on samuti SLAst tingitud ning otseselt seotud kättesaadavuse ja lubatud katkestuste arvuga. |

| | |
|---|---|
| **ID** | MF13 |
| **Kirjeldus** | Rakendus peab töötama ja tagastama vastavaid veateateid, kui vähemalt üks süsteem, millest antud rakendus sõltub, on kättesaamatu. |
| **Põhjendus** | See nõue on oluline rakenduse tarbijate jaoks, kes saavad sellisel juhul õigemat ja läbipaistvamat informatsiooni vigade kohta ja seega saavad need vastavalt vajadusele omakorda paremini nendele vigadele reageerida. |

### 3.1.5 Hooldatavus

| | |
|---|---|
| **ID** | MF14 |
| **Kirjeldus** | Rakenduse kood peab võimalikult palju olema jagatud loogiliselt üksteisest sõltumatutesse moodulitesse. |
| **Põhjendus** | Vana süsteem, mida parendada üritatakse, on juba täna koodi vaates segaselt |

| | |
|---|---|
| | kirjutatud, mille tõttu ei ole seda enam efektiivne hallata – raske on erinevaid mõjusid ja seoseid hinnata. See avaldab otsest mõju kvaliteedile. Kuna see on ka üheks lahendatavaks probleemiks, siis juba abstraktsioonikihi rakenduse puhul soovitakse sama viga vältida. |

| | |
|---|---|
| **ID** | MF15 |
| **Kirjeldus** | Mitme mooduli poolt kasutatud jagatud loogika peab olema selgelt eristatav. |
| **Põhjendus** | Kahtlemata ei saa tervet rakendust täiesti sõltumatuteks mooduliteks jaotada. Selle tõttu eksisteerib jagatud loogikat ning see tuleb ka selgelt teisest loogikast eristada. See on oluline selleks, et suurendada korduvkasutatavust ning anda vihjeid, et eristatud jagatud loogika puhul tuleb mõjude ja seoste hindamisel olla tähelepanelikum. |

### 3.1.6 Porditavus

| | |
|---|---|
| **ID** | MF16 |
| **Kirjeldus** | Rakendus peab töötama ettevõtte infrastruktuuri poolt pakutaval sobival platvormil. |
| **Põhjendus** | See nõue on tingitud jaotises 2.1.2 kirjeldatud piirangust, et ettevõttes on kehtestatud tarkvara erinevate osade vaates standardkomponentide nimekiri. Antud rakendus on oma olemuselt piisavalt lihtne, mistõttu puudub vajadus hakata ettevõttesse valima ja põhjendama uut platvormi. |

| | |
|---|---|
| **ID** | MF17 |
| **Kirjeldus** | Rakenduse kood peab olema kirjutatud nii, et hiljem oleks seda võimalikult lihtne liidestada uue parendatud keskse süsteemiga. |
| **Põhjendus** | See nõue on tingitud valitud muudatus läbi abstraktsiooni mustrist. Selle mustri sammust 3 alates on vaja loodav abstraktsioonikihi rakendus liidestada uue |

| | |
|---|---|
| | pakkujaga. Antud hetkel on veel ebaselge, kas tulevikus luuakse täiesti uus pakkuja või parendatakse olemasolevat, aga paindlikkuse huvides peaks lahti jätma mõlemad valikud. |

| | |
|---|---|
| **ID** | MF18 |
| **Kirjeldus** | Rakenduse poolt pakutavad veebiteenused peavad pärast olulisi uuendusi olema vähemalt 2 kuud tagasiühilduvad eelmise versiooniga. |
| **Põhjendus** | Kuna antud rakendusel hakkab tulevikus olema mitu tarbijat, siis tekib kahtlemata olukordi, kus kõik tarbijad ei suuda samaaegselt uut versiooni kasutusele võtta. Selle tõttu tuleb muudatuste sisseviimise lihtsustamiseks nende kasutuselevõtt tarbijatest lahti siduda. Seega on vajalik, et rakendus toetaks samaaegselt korraga mitut versiooni. Nii tagatakse sujuvam ja probleemivabam üleminek uutele versioonidele. |

## 3.2 Nõuetest lähtuvad valikud

Antud jaotises kirjeldatakse ja põhjendatakse eelmises jaotises välja toodud kõikidest nõuetest lähtuvalt tehtud erinevaid arhitektuuri aspekte puudutavaid valikuid. Lähtudes antud peatükis kasutatavast arhitektuuri vaatest ja definitsioonist ei ole välja toodud valikuid, mis on seotud rakenduse kontekstis väga detailse ja spetsiifilise tasemega.

### 3.2.1 Liidese tehnoloogiad

Lähtudes nõudest MF8 peab rakendus olema võimeline kasutama teenuste vaates protokolli, mis järgib REST arhitektuuri stiilist tulenevaid piiranguid. Seetõttu osutus valituks HTTP protokoll, kuna HTTP protokoll on REST stiili järgivate arhitektuuride puhul de facto protokoll (Bloomberg, 2013b). Samuti on HTTP päringute tarbimise tugi olemas paljudes programmeerimiskeeltes (Richardson, Ruby, 2007, 107). Seega rahuldab HTTP protokolli valimine ka nõudest MF9 tulenevad vajadused, et liidestus abstraktsioonikihiga peab kasutama protokolli, millega on võimeline liidestuma nii veebi iseteeninduse, mobiili, teenindaja liidese kui ka televisiooni iseteeninduse rakendus. Lisaks tänastele vajadustele

säilitab ka HTTP kasutamine oma laia toetatavusega tuleviku vaates paindlikkuse liidestuda süsteemidega, mida praegu veel ei eksisteeri.

Lisaks HTTP protokollile on vaja valida andmete edastamise formaat, kuna HTTP seda ei piira. Antud töö vaates on edastatavate andmete puhul tegu tekstilist, mitte binaarset, tüüpi andmetega, mis peavad olema masinloetavad. Seega peab valituks osutuv andmeformaat andmeid esitama piisavalt struktuurselt. Parimateks ja enamlevinud kandidaatideks on XML ja JSON. Need kaks on valikus, kuna need on taas kujunenud veebiteenuste puhul de facto standardiks. Küll aga peaks valima ühtsuse ja lihtsuse huvides ainult ühe. Antud juhul on paremaks valikuks JSON. See andmeformaat on edastatava andmemahu mõttes XMLst oluliselt kergekaalulisem. Samuti on JSON formaadi puhul paljudes enamlevinud programmeerimiskeeltes olemas vastavad sisseehitatud vahendid või teegid, millega antud formaadis andmeid töödelda (JSON, 01.05.2014). Lisaks eelistavad veebilehitseja põhised veebiteenuste tarbijad JavaScripti tõttu tavaliselt JSON formaati teistele andmeformaatidele. Sama põhjenduse toob välja ka Allamaraju (2010, 58).

### 3.2.2 Platvormi valik

Jaotises 1.2 selgus, et tarbijad on praegu keskse süsteemiga liidestatud läbi andmebaasi protseduuride ja funktsioonide kasutades selleks otse andmebaasiga suhtlemiseks mõeldud protokolli või SOAP ja WSDL kombinatsiooni. Eelmises alampeatükis selgus, et uueks liidese protokolliks on valitud HTTP ja andmeformaadiks kasutatakse JSONit. SOAP kasutab samuti HTTPd, kuid ainult transpordiks ja seega ei saa seda lugeda sobivaks valikuks. Samuti seisneb tänane SOAP lahendus hetkel üks ühele andmebaasi protseduuride ja funktsioonide pakkumises kasutades SOAP protokolli, mis ei sobi REST arhitektuuri stiili põhimõtetega. Täpsemalt on taaskord sellest juttu teenuste disaini peatükis 4. Selle tõttu ei ole antud abstraktsioonikihi rakenduse ülesandeks ainult protokolli tõlkimine. Seega on vaja leida sobiv platvorm, mis vastavat teisendust teostab.

Üheks kandidaadiks on kasutada sama vahetarkvara, mis TV iseteeninduse jaoks hetkel ainult protokolli teisendab. Nimetatud vahendiks on Oracle Fusion Middleware. See vahend aga ei sobi antud probleemi lahendamiseks, kuna antud tarkvara on liiga kohmakas ning see ei paku arendamise vaates vajalikku paindlikkust ja keskkonna lihtsust, et selles saaks piisavalt efektiivselt nii rahalises kui ajalises mõttes soovitud teisendusi teostada. Lisaks on selle tarkvara funktsionaalsus tegelikult oluliselt laiem, mis muudab tema nõuded riistvarale kõrgeks. Selle tõttu vajab see ka lihtsamate tegevuste jaoks oluliselt rohkem ressurssi, kui

alternatiivsed kergekaalulisemad erilahendused. Lisaks on selle tarkvara jõudlusega toodangukeskkonnas varasemalt probleeme esinenud. Seega ei ole antud tarkvara puhul tegu sobiva vahendiga, mis oleks ka tulevikus jätkusuutlik.

Teiseks analüüsitavaks kandidaadiks on Oracle Application Express tarkvara vastav moodul, mis on mõeldud just REST veebiteenuste loomiseks. Kuna teenindajatele mõeldud müügiliidese rakendus on loodud vahendiga Application Express, siis see tarkvara on Elionis juba täna kasutusel. Taaskord on antud vahendi puhul probleemiks keerukama loogika realiseerimise ebaefektiivsus ja piisava paindlikkuse puudumine. Lihtsamate REST teenuste loomiseks sobib see vahend väga hästi, kuid Elioni kontekstis on andmebaasis olev loogika oluliselt keerukam, mille realiseerimine nõuaks antud vahendi puhul palju käsitööd ning suur osa loogikast tuleks realiseerida PL/SQL keele protseduuride ja funktsioonidega. Seega oleks abstraktsioonikihi rakenduse loogika realiseeritud PL/SQL keeles. Esmalt vähendab see tuleviku vaates muudatuste suhtes paindlikkust, kuna võib-olla kasutatakse uue süsteemi realiseerimiseks mingit muud platvormi, mille tagajärjel peaks vastav abstraktsioonikiht läbi PL/SQL keele suhtlema uue süsteemiga, mis vastavalt valitavale platvormile võib olla ebamõistlik. Lisaks pole loodava abstraktsioonikihi rakenduse spetsiifikast ning Elioni kontekstist lähtudes PL/SQL keele nii laialdane kasutamine järgnevatel põhjustel otstarbekas.

1. PL/SQL pole täielikult objekt-orienteeritud programmeerimiskeel. Objekt-orienteeritud programmeerimise eesmärk on muuhulgas võimaldada luua paremini hallatavat ja lihtsamini arusaadavat koodi läbi parema seostatuse reaalses maailmas oleva ärilise domeenimudeliga (Object Oriented Software …, 01.05.2014). Teisest küljest tähendab teise keele valimine baasist eemal oleva rakenduse kasutamist, mis võib jõudlusele halvasti mõjuda. Teatud olukordades, näiteks spetsiifilise andmetöötluse puhul, võib PL/SQL kasutamine olla põhjendatud, kuna PL/SQL on andmebaasi lähedal ning seetõttu saab sellest lähtuvalt jõudluse võitu – puudub vajadus andmeid üle võrgu kanda ning puudub vajadus eraldi spetsiaalse välise tarkvara järgi, kuna niikuinii opereeritakse ainult andmebaasis olevate andmetega. Küll aga on antud kontekstis pigem tegu keerulise arvutusliku äriloogikaga, mis ei hõlma suurte andmemahtude peal opereerimist. Seejuures pooldab autor pigem Fowleri (2003) arvamust, et pigem tuleks keerulise tarkvara loomisel lähtuda hallatava koodi kirjutamisest ning jõudlusega tegeleda lähtuvalt vajadusest alles siis, kui see probleeme hakkab tekitama. Seega sobiks objektorienteeritud keel antud ülesande lahendamiseks paremini.

2. Lisaks tekib PL/SQL kasutamisel suur sõltuvus andmebaasist. Selle tagajärjel on näiteks lähtekoodihaldus keerulisem. Programmikood on andmebaasiga seotud ning selleks, et arenduste mõju näha ja katsetada, tuleb vastavad koodi muudatused andmebaasi kanda. Selle tõttu on igale arendajale sõltumatu arenduskeskkonna loomine samuti raskendatud - iga arendaja arvutisse tuleb installeerida andmebaas, mis näiteks võrreldes Java põhiste rakendustega on oluliselt keerulisem üles seada ning samuti vajab andmebaasi lahendus rohkem arvuti ressurssi.

3. PL/SQL keel on seotud ühe toote ja tootjaga. See on halb selle tõttu, et sellisel juhul on kogu rakendus seotud Oracle tarkvaraga. Oracle andmebaasisüsteem on aga suletud lähtekoodiga kommertstarkvara ning Elion on võtnud suuna võimalusel sellisest tarkvarast loobuda.

4. Tuginedes ettevõtte tänasele kogemusele on tööjõuturult raske leida PL/SQL arendajaid.

Eelnevate põhjenduste tõttu ei sobi ka vahend Oracle Application Express. Seega on vaja abstraktsioonikihi rakenduse realiseerimiseks valida muu paindlikum platvorm. Platvormide ja programmeerimiskeelte koha pealt pakub Elioni standardkomponentide nimekiri täiendavalt välja Javat ja Pythonit. Soovitatud keelte nimekirja puhul on tegemist valiku piiramisega. Antud töö skoopi ei kuulu detailne põhjendus, miks just need keeled on valikus. Valikud ise on piiratud selle tõttu, et vähendada erinevate tehnoloogiate kasutust, mis omakorda lihtsustab oluliselt süsteemide haldamist. Samuti on arenduskompetentsi vajadus ühtsem, mille tagajärjel saavad samad arendajad lihtsamini ettevõtte erinevate valdkondade projekte teha. Nendest kahest keelest osutus Java Pythoni ees valituks järgmiste põhjuste tõttu.

1. Java puhul on tegu maailmas levinud keelega. Lisaks keskendutakse näiteks ka Tallinna Tehnikaülikoolis palju Java keelele, mistõttu on suurem tõenäosus Java arendajaid tööjõuturult leida.

2. Java on Elionis ennast tõestanud keel – paljud ärikriitilised rakendused on Javas realiseeritud.

3. Elionil endal on rohkem hea kompetentsiga Java kui Pythoni arendajaid.

4. Ettevõtte haldusüksuse poolelt on Javal põhinevatele rakendustele hea tugi.

5. Ärikriitilise olemusega rakenduse jaoks eksisteerib juba praegu vajalik infrastruktuur.

Java platvormi valimine võimaldab vähese vaevaga rahuldada nõuded F1 (rakendus peab logima syslog protokolliga) ja F3 (rakendus peab oma veebiteenustele pakkuma käivitatavat spetsifikatsiooni). Nende nõuete rahuldamiseks on Javas juba vastavad levinud teegid realiseeritud. Logimise kontekstis saab esile tõsta Apache log4j, mis on ettevõttes täna juba laialdaselt kasutusel. Käivitatava spetsifikatsiooni näidetest saab välja tuua vahendi Concordion, mida samuti on ettevõttes varem katsetatud.

Java platvormi valimisest lähtuvalt täpsustuvad infrastruktuurile püstitatud nõuded MF7 ja MF16. Vastavalt olemasolevale Java rakenduste infrastruktuurile on seega nõuded järgmised.

1. Rakendust peab saama kompileerida ja see peab töötama Java versioon 7 virtuaalmasinas.

2. Rakendus peab töötama JBoss rakendusserveris.

Nende nõuete täitmiseks ei pea rakenduse vaates midagi erilist tegema, kuid sellegipoolest tuleb nende nõuetega arvestada juba alguses, kuna hiljem võib tegu olla kalli muudatused. Java versiooniga ühilduvuse vaates tuleb kasutatavate väliste teekide puhul valida versioon, mis sobib nõutud Java versiooniga. JBoss rakendusserveris töötamise jaoks peab rakendus kasutama Java Servlet standardit. Kuna ettevõttes töötab iga Java rakendus eraldi JBossi instantsis ja seega ka eraldi Java virtuaalmasina protsessis, siis ei teki väliste teekide kasutamise konflikte erinevate rakenduste vahel.

### 3.2.3 Infrastruktuur

Infrastruktuuri otsuseid mõjutavad kõige enam toetatava teenuse SLA ning jaotises 3.1.2 käsitletud rakenduse soorituse tõhususega seotud nõuded. Kõrge SLA tõttu on vaja rakendus kindlasti paigaldada mitme rakendusserveri vahel kobarasse (ingl. k. *cluster*). Minimaalselt peab kobaras olema vähemalt 3 aktiivset ja võrdset rakendusserveri instantsi. Kolme on vaja selle tõttu, et igal ajahetkel oleks liiasus tagatud. Kolme läheb vaja siis, kui toimub rakenduse uue versiooni paigaldus toodangukeskkonda või rakendusserver on mingil muul põhjusel hooldusrežiimis. Sellisel juhul on ainult üks instants hooldusrežiimis ja ülejäänud kaks on üksteisele tagavaraks. Samuti tagab kobarlahendus ka parema ettenägematute tõrgete taluvuse. Kuna lubatud katkestuste arv on väike, siis kobar on selle juhul ainuvõimalik lahendus.

Tõhususe koha pealt saab väita, et infrastruktuuri vaatest on eelnevalt kirjeldatud infrastruktuuriga rakenduse tervikvaatest võimalik täita soorituse tõhususega seotud nõudeid. Selleks annab alust teadmine, et täna on juba toodangu keskkonnas samade tõhususe nõuetega ning samal infrastruktuuri lahendusel töötav veebi iseteeninduse rakendus.

Nõude MF5 kohaselt peab rakendust olema võimalik skaleerida ilma koodi muutmata. Selle tõttu on infrastruktuuri vaates otsustatud hoida rakendus serveri poolel olekuta. See lähenemine võimaldab antud rakenduse läbilaskevõimet suurendada ainult kobarasse uute rakendusserverite lisamisega, ilma et peaks midagi täiendavat tegema. Seega on selle lähenemise üheks eeliseks koodi muutmise vajaduse puudumine. Täiendavalt ei pea olekuta rakenduse puhul erinevate kobarasse kuuluvate rakenduste vahel seansse sünkroniseerima, mis võib suuremahuliste seansside puhul jõudlusele halvasti mõjuda.

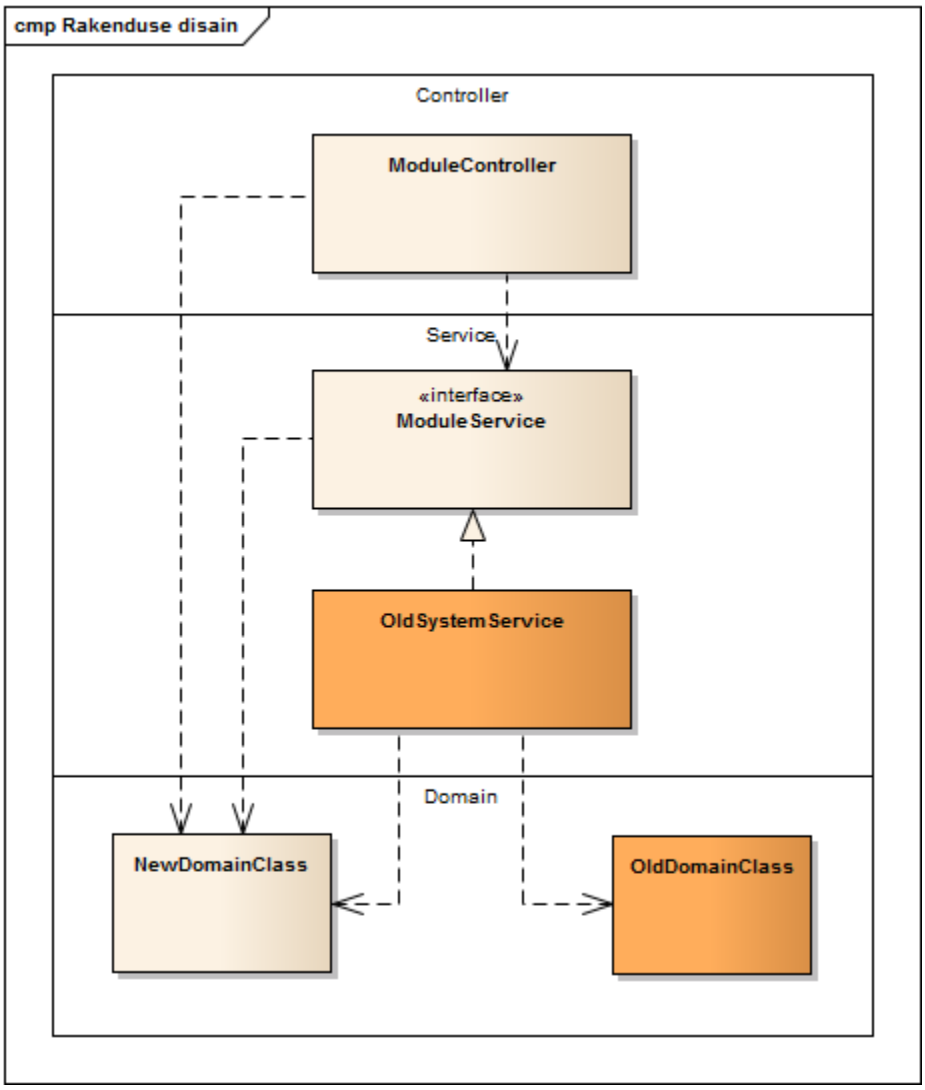
### 3.2.4 Rakenduse disain

Rakenduse disaini mõjutavad eelkõige moduleerituse ja liidestatusega seotud nõuded MF14, MF15 ja MF17. Kuna antud rakenduse eesmärk on luua veebiteenuste abstraktsioonikiht, siis sellel rakendusel antud töö skoobi vaates domeeniloogikat ei ole, kuna tegelik domeeniloogika jääb endiselt kesksesse süsteemi. Selle rakenduse ülesanneteks on ainult liidese abstraheerimine ja seega ka selleks vajalik andmete teisendamine. Tegeliku domeeniloogika puudumine muudab selle rakenduse disaini lihtsamaks.

Horisontaalsel suunal jaotatakse rakenduse funktsionaalsus eraldiseisvateks mooduliteks. Konkreetseteks mooduliteks jaotamise aluseks on Elioni domeenimudel, mille põhjal loogiliselt kokku kuuluvad domeeni objektid on ühes moodulis. Seega konkreetne rakenduse moodulite nimekiri on rakenduse disaini kontekstis ebaoluline, kuna mooduliteks jaotamise põhimõte on sõltumatu konkreetsest domeenimudelist. Konkreetsest domeenimudelist on täpsemalt juttu peatükis 4. Sellegipoolest ei ole võimalik antud rakendust jaotada üksteisest täielikult sõltumatuteks mooduliteks ning seega kasutatakse ka erilist jagatud moodulit, kuhu koondatakse erinevate vertikaalsete kihtide klasse, mis on kasutusel rohkem kui ühes moodulis. Heaks näiteks on domeeniklass klient, mis on seotud paljude moodulitega oma keskse olemuse tõttu.

Iga moodul sisaldab järgnevalt kirjeldatud vertikaalse jaotuse komponente. Vertikaalsel suunal on rakendus üles ehitatud nii, et seda oleks tulevikus võimalikult lihtne liidestada uue

keskse süsteemiga või olemasoleva keskse süsteemi parendatud realisatsiooniga, ilma et see mõjutaks väliseid tarbijaid (vt Joonis 2).




**Joonis 2 Rakenduse disain**

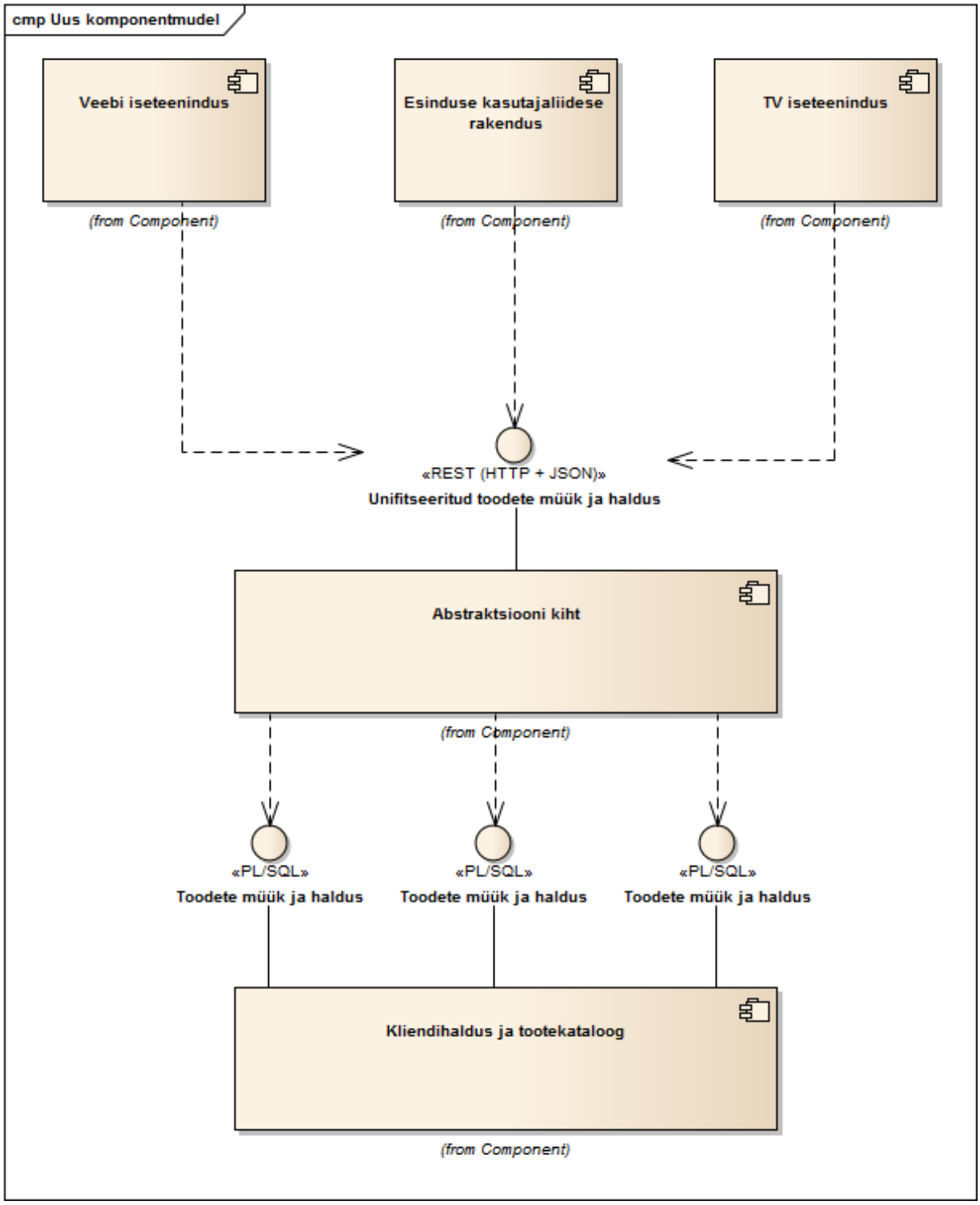
See on lihtsustatud ja üldisatud joonis ühe suvalise mooduli kihilisest läbilõikest, kust on näha, et rakendus vahendab vana süsteemi teenuseid ja teisendab neid uude andmeformaati. Domeenikiht on seejuures väga õhuke, sisaldades vaid andmeid hoidvaid klasse, kuna tegelik realisatsioon jääb endiselt tänasesse kesksesse süsteemi. Vana süsteemiga seosed on joonisel märgitud oranži värviga. Siit järeldub, et vana süsteemi teenuste liides on tõlgitud uue süsteemi liideseks läbi OldSystemService klassi, mis realiseerib ModuleService liidest. Viimane omakorda sõltub vaid uutest domeeniklassidest. Seega puudub sellise lahenduse puhul teenuste kihis sõltuvus vana süsteemi teenustest ja domeeniklassidest ning vana süsteemi uuega asendamiseks on vaja vaid luua ModuleService liidesele uus realisatsioon.

Joonisel välja toodud kontrolleri ülesanne on rakenduse poole tehtud HTTP pöördumised vastu võtta ja suunata need õigete teenusklasside meetodite vastu. Siin on võimalik rahuldada nõude F2 alamosa, mille puhul on vajalik, et rakendus logiks kõiki sissetulevaid pöördumisi.

Kuna taustsüsteemiga suhtlus on koondatud ainult ühte kihti, siis sellisel juhul on võimalik ka vähese vaevaga rahuldada nõue MF13, et rakendus peab tagastama veateateid, kui taustsüsteem pole kättesaadav. Selleks on vaja tagada antud kihis üldine veatöötlus, mis kõikidele taustsüsteemi pöördumistele peab rakenduma. Täiendavalt võimaldab see koht rahuldada ka logimise nõude F2 teise poole, mille puhul on vajalik, et rakendus logiks kõik väljuvad pöördumised.

## 3.3 Uus tehniline kontekst

Eelmisest alampeatükist lähtuvalt luuakse abstraktsioonikihi rakendus eraldiseisva Java platvormil baseeruva rakendusena. Visuaalse ülevaate saamiseks esitatakse antud alampeatükis uue tehnilise konteksti komponentdiagramm (vt. Joonis 3).




**Joonis 3 Uus komponentmudel**

# 4. Veebiteenuste liidese disain

Antud peatükk käsitleb abstraktsioonikihi rakenduse poolt pakutavate veebiteenuste liidese disaini. Esmalt tuuakse välja loodava liidese soovitud omadused. Seejärel analüüsitakse veebiteenuste liidese disainimisel tehtavaid erinevaid valikuid, mis liidesele need soovitud omadused tagavad.

## 4.1 Veebiteenuste liidese oodatavad omadused

Selles jaotises välja toodud veebiteenuste liidesele oodatavad omadused on tingitud peatükis 2 kirjeldatud probleemidest.

1. Veebiteenuste liides peab olema inglisekeelne. See omadus on tingitud probleemist, et tänane keskse süsteemi liides on eestikeelne. Selle probleemi täpsustus on välja toodud jaotises 1.3.2.

2. Veebiteenuste liides peab kasutama ühtset domeeniterminoloogia. Lisaks kasutatava keele ühtlustamisele on oluline inglise keele puhul kasutada oma olemuselt samade asjade puhul vaid üht mõistet.

3. Veebiteenuste liides peab olema võimalikult lihtne ja intuitiivne. Konkreetsemalt peab selleks veebiteenuste liides teisendama tänast keskse süsteemi liidest kõigile tarbijatele ühiselt sobivale kujule ja olema võimalikult lihtsalt arusaadav. See on vajalik selleks, et vältida tulevikus erinevate tarbijate poolt loodavaid dubleeritud äriloogika teisendamise realisatsioone ning muuta liides arendajatele arusaadavamaks, mis lühendab tarbija rakenduste uute arendajate sisseelamisaega.

4. Veebiteenuse pakkuja ja tarbija peavad olema üksteisega võimalikult vähe sidestatud (ingl. k. *loose coupling*). See omadus on oluline selleks, et tulevikus oleks võimalik rohkem ja lihtsamalt teostada selliseid muudatusi, mis ei tingiks kõikide tarbijate muudatust või uut veebiteenuste liidese versiooni. Vähene sidestatud muudab erinevate muudatuste tagasiühilduvuse toetamise lihtsamaks ning soosib mittedubleeriva koodi loomist. See omadus aitab leevendada tänast probleemi, et

muudatuste erinevate tarbijatega kooskõlastamine on aeganõudev, mistõttu on tekkinud erinevatele tarbijatele samast teenusest mitu erinevat realisatsiooni.

## 4.2 Veebiteenuste liidese disaini valikud

Selles jaotises käsitletakse veebiteenuste liidese disainiga seoses tehtud valikuid, mis lähtuvad põhiselt eelmises jaotises kirjeldatud soovitud omadustest. Lisaks peavad need valikud arvestama ka abstraktsioonikihi rakenduse arhitektuuri puudutavate tehtud valikutega.

### 4.2.1 Domeeniterminoloogia

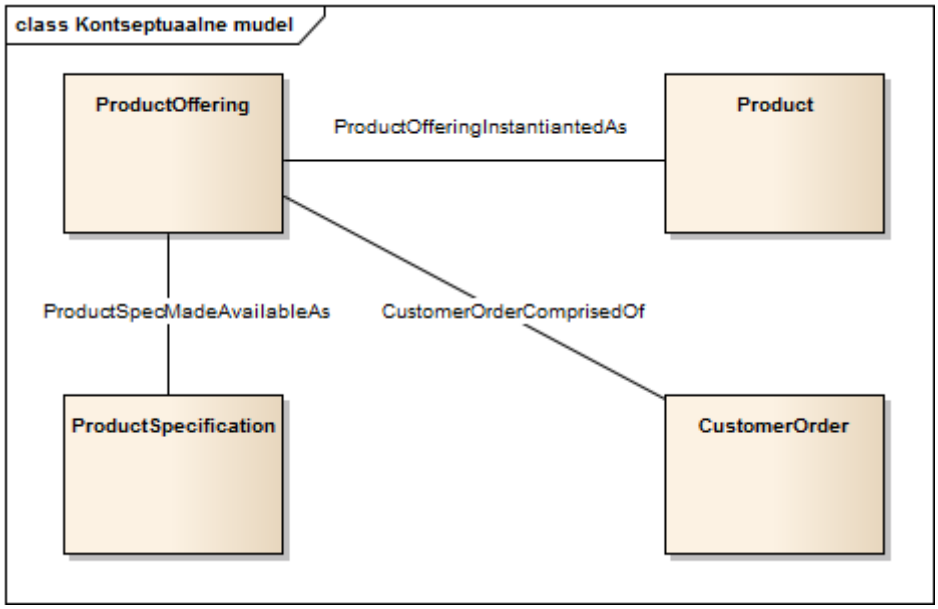
Domeeniterminoloogiaga seotud valik peab eelkõige lähtuma veebiteenuste liidesele soovitud omadususte st 1 ja 2, mille kohaselt peab domeeniterminoloogia olema inglisekeelne ja ühtne.

Domeeniterminoloogia puhul pole seda mõtet Elionil endal välja töötada, kuna see on suur töö ning tõenäoliselt ei anna oma domeeniterminoloogia loomine mingisugust erilist konkurentsieelist. Seetõttu on mõistlik uurida maailmas levinud raamistikke. Seejuures sobib kõige paremini ühenduse TMForum poolt loodud standardikogumiku Frameworx (TM Forum Frameworx, 01.05.2014) andmemudelit käsitlev osa Information Framework (Information Framework …, 01.05.2014). TMForum on ühendus, kuhu kuulub üle 900 ettevõtte üle maailma ning nende ettevõtete puhul on tegu Elioniga sarnaste teenust pakkuvate ettevõtetega. Seetõttu on TMForum poolt hallatud standardikogumik just selle tööstusharu spetsiifiline. Lisaks on selle standardikogumiku andmemudelit käsitleva osa eesmärk pakkuda just ühtset domeeniterminoloogiat ja kontseptuaalset andmemudelit. Võttes kasutusele üle maailma levinud terminoloogia, suudetakse terminoloogiat mitte ainult Elioni sees ühtlustada, vaid ka näiteks teiste telekommunikatsiooni ettevõtetega, kes sama raamistikku kasutavad.

Information Framework jaotab olemid kaheksasse suuremasse valdkonda. See raamistik katab ära kogu ettevõttega seotud võimalikud mõisted. Seetõttu tuleb ära mainida, et kuna käesoleva töö skoop piirdub toodete müügi ja haldusega, siis kasutatakse antud töös tervest raamistikust ära ainult kahe valdkonna mõisteid. Nendeks valdkondadeks on toode (ingl. k. *Product*) ning klient (ingl. k. *Customer*). Kliendi valdkonnast kasutatakse ainult kliendi tellimuse mõistet, kuna ülejäänud on väljaspool antud töö skoopi.

Täiendavalt tuleb siiski ära mainida, et antud raamistiku puhul on tegu kontseptuaalse ja teenusepakkujate tööstusharu jaoks üldistatud mudeliga. Selle tõttu on vaja seda praktikas

kasutamiseks vastavalt vajadusele kohandada ja konkreetsemaks muuta. Selleks on vaja Elioni vaates kasutatavad olemid panna spetsiifiliselt telekommunikatsiooni valdkonna konteksti. Seda saab kõige paremini teha illustreerides vastavaid üldistatud olemeid konkreetsete näidetega telekommunikatsiooni valdkonnast. Seega on järgnevalt esitatud UML klassidiagrammina (vt. Joonis 4) Information Frameworki vastavad mõisted klassidena, et saada esmane ülevaade kasutatavast klassidest ja nende omavahelistest seostest. Seejärel on need mõisted täpsemini näidete abil lahti seletatud.




**Joonis 4 Kontseptuaalne mudel**

**Toote spetsifikatsioon (ingl. k *Product Specification*)**

Joonisel kujutatud neliku vaates on põhimõisteks toote spetsifikatsioon. See on midagi, mida saab kliendile müüa. Seejuures ei sisalda toote spetsifikatsioon informatsiooni tingimuste kohta, mille alusel seda kliendile müüakse. Elioni kontekstis on toote spetsifikatsioonideks näiteks internet, telefon ja televisioon. Seega ütleb spetsifikatsioon seda, mida müüakse. Seejuures võivad toote spetsifikatsioonid moodustada ka hierarhiaid. Elioni näitel võib televisiooni alla kuuluda veel hulk teemapakette, salvestamine ja noppekanalid.

**Toote pakkumine (ingl. k *Product Offering*)**

Selleks, et toote spetsifikatsioone müüa saaks, kasutatakse mõistet toote pakkumine. See on midagi, mis defineerib ära tingimused, mille alusel pakkumisse kuuluvaid toote spetsifikatsioone müüakse. Seega on toote spetsifikatsioon nagu mall, mille alusel saab pakkumisi koostada. Näiteks võivad Elioni puhul eksisteerida eraldi era ja äri interneti

pakkumised. Mõlemad on seotud ühe interneti toote spetsifikatsiooniga, aga nad defineerivad ära erinevad müügitingimused. Müügitingimuseks võib olla näiteks hind või tähtajaline leping.

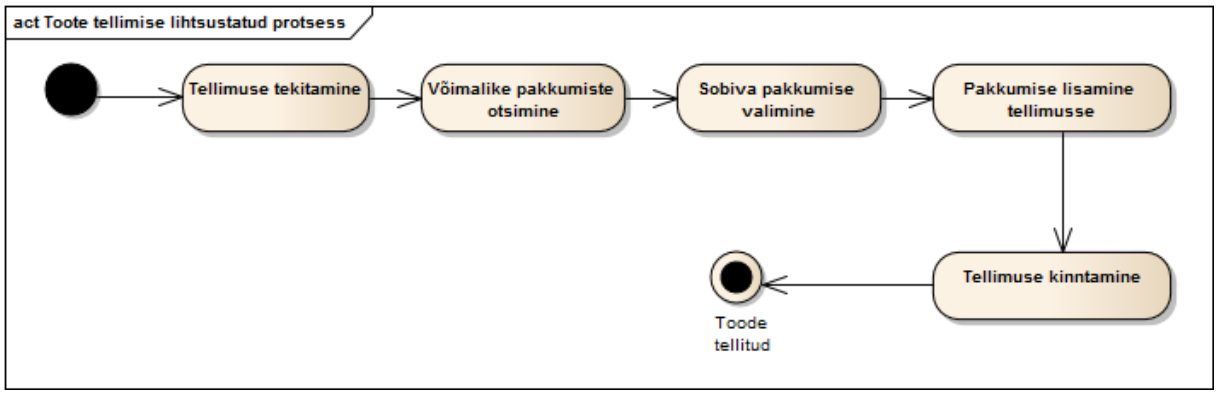
**Tellimus (ingl. k *Customer Order*)**

Selleks, et klient saaks tellida toote pakkumisi, tuleb neid tellida läbi tellimuste. Antud raamistiku puhul ei käsitleta tellimust, kui mingit kliendi ja ettevõtte omavahelist siduvat objekti vaid pigem on see justkui ostukorv, kuhu koondatakse kliendi soovid. Tellimusse saab lisada soovitud toote pakkumisi. Seejärel saab tellimuse kas kinnitada või tühistada.

**Toode (ingl. k *Product*)**

Toode on midagi, mis on juba konkreetse kliendi oma ja tekib pärast tellimuse kinnitamist. Pärast tellimuse kinnitamist moodustatakse tellimuses olnud toote pakkumistega seotud kliendile kuuluvad toote eksemplarid. Sisuliselt on toodete puhul tegu konkreetse siduvusega, mille alusel hiljem kliendile arveid koostatakse. Arvelduse osa antud töö skoopi ei kuulu.

Eelpool kirjeldati ära klassid ja nende seosed. See väljendas aga ainult mudeli struktuurset osa. Käitumuslikust aspektist (vt. Joonis 5) on seega lihtsustatud tellimise protsess järgmine: pakutavate toote pakkumiste seast valitakse välja sobivad, mis lisatakse tellimusse ning pärast vastav tellimus kinnitatakse.




**Joonis 5 Toote tellimise lihtsustatud protsess**

### 4.2.2 Ühise äriloogika koondamine

Liidese lihtsuse ja intuitiivsuse omadus on konkreetselt tingitud hetkeolukorrast, kus keskne süsteem peegeldab välja liiga palju süsteemi sisest keeruat realisatsiooni, mistõttu peavad

kõik tarbijad sellega ise hakkama saama. Seega on vaja lisaks ühtlustatud domeeniterminoloogiale ka abstraktsioonikihis teostada teisendust, mis muudaks veebiteenuste liidese teiste süsteemide jaoks lihtsamaks.

Lahenduse strateegiaks valitud muudatus läbi abstraktsiooni muster on oma olemuselt üldine, mistõttu ei puuduta see detailse taseme küsimust, kuidas täpsemalt abstraktsioonikihti realiseerida. Seega on antud töö kontekstis vaja realisatsiooni jaoks uurida ka selle tasemega seonduvaid mustreid, mis sobiksid vastavaks lihtsustuseks. Kuna selle taseme puhul on tegu üsna detailse probleemiga, siis lahenduse kandidaatideks sobivad Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides poolt kirjeldatud sarnased struktuursed mustrid adapter (Design Patterns …, 2000, 139 - 151) ja fassaad (Design Patterns …, 2000, 185 - 195).

Need mustrid on üsna sarnased, kuid siiski veidi erineva eesmärgi ja sisuga. Antud probleemi lahendamiseks sobib seejuures siiski fassaad muster, kuna selle mustri rakendamine tagab veebiteenuste liidesele paremini soovitud lihtsuse omaduse. Fassaadi puhul on eesmärgiks just võimalikult palju realisatsiooni detailsust fassaadi kasutaja eest ära peita. Näiteks hetkeolukorras on vaja tellimuse kinnitamiseks välja kutsuda kaks kindlat teenust kindlas järjekorras. Fassaadiga saaks luua ühe kinnitamise veebiteenuse, mis kutsuks neid mõlemaid ise välja ning tarbijad ei pea sellest spetsiifikast midagi teadma. Adapter mustri eesmärk on aga seevastu pigem ainult teisendada mitte kokkusobivaid liideseid nii, et neid saaks koos kasutada. Seega pole adapteri eesmärk loogika kapseldamine ja seeläbi selle peitmine, mis antud töö kontekstis lisaväärtust loob.

Hetkeolukorras on erinevad tarbijad juba vastavaid teisenduskihte endale juba loonud, mis samuti antud mustreid kasutavad. Mõned neist järgivad rohkem adapter mustrit ja mõned pigem fassaad mustrit. Abstraktsioonikihi veebiteenuste liidese puhul on seega eesmärk luua üks ühtne fassaad, mida kõik teeninduskanalite rakendused saavad kasutada. See aitab vähendada dubleerimise vajadust ja muudab keskse süsteemi liidese kõikide tarbijate jaoks ühtseks ning võimalikult lihtsalt kasutatavaks.

### 4.2.3 Veebiteenuste liidese disaini stiil

Veebiteenuste liidese disaini stiili all on mõeldud põhimõtteid, millest lähtuvalt liidest disainitakse. Selle definitsiooni järgi on need põhimõtted omased kas RPC või REST stiilile. Antud probleemide kontekstis on REST stiili kuuluvate põhimõtete kasutamine parem valik.

REST stiili valimise põhjendamiseks ja konkreetse töö kontekstis selle rakendamise täpsustamiseks on vajalik kõigepealt välja tuua REST stiili puudutavad põhipunktid.

REST on tarkvara arhitektuuri stiil hajutatud hüpermeedia süsteemide jaoks nagu näiteks veeb (Fielding, 2000). Selle definitsiooni kohaselt on REST tervikuna RPCst laiem mõiste. Sellegipoolest on REST ja RPC veebiteenuste liidese disaini kontekstis võrreldavad, kuna REST arhitektuuri stiil puudutab ka seda vaadet. REST üldisemalt aga defineerib ära hulga piiranguid, mille järgimine tagab süsteemile teatud omadused. Fieldingu (2000) kohaselt on RESTi poolt kohustuslikud piirangud ja nendest tulenevad omadused järgmised:

- Kliendi ja serveri vastutused peavad olema eraldatud. Selle põhimõtte järgmine vähendab kliendi ja serveri sidestatust (ingl. k. *tight coupling*). Seda saab illustreerida näitega, kus klient ei tea, kuidas serveris andmeid salvestatakse ja server ei tea, mis olekus klient parasjagu on.

- Kliendi ja serveri vaheline suhtlus peab olema ilma olekuta. Iga kliendi pöördumine ei tohi olla seotud mingi eelmise kliendi pöördumisega. Server ei pea hoidma kliendi olekut. Selle tulemusena on klient ja server taas vähem sidestatud ning selle põhimõtte järgimine võimaldab lihtsamat skaleeritavust.

- Efektiivsuse tagamiseks peab päringute vastustes olema võimalik lisada infot, kas ja kui kaua seda vastust vahemälus hoida tohib.

- Ühtne liides (ingl. k. *uniform interface*) – see tähendab seda, et ressursside infot päritakse ja muudetakse läbi ühtse liidese. Ühtse liidese kasutamine võimaldab hoida liidest lihtsana, kuna see on üldistatud ja seega sõltumatu konkreetsest realisatsioonist. Lisaks aitab ühtne liides vähendada ka sidestatust. Ühtne liides on omakorda defineeritud läbi nelja omaduse, mida liides omama peab, et seda ühtseks nimetada saaks:

    - Ressursse peab olema võimalik unikaalselt tuvastada.

    - Ressursside muutmine peab käima läbi nende representatsioonide.

    - Kõik vastussõnumi tõlgendamiseks vajalik info peab sisalduma selles sõnumis.

    - Rakenduse oleku muutmiseks peab kasutama hüpermeediat.

- Kihtidest koosneva süsteemi puhul peab üks kiht olema teadlik ainult vahetult tema kõrval olevast kihist. Selline piirang hoiab kihtide sõltumatust paremini kontrolli all. Näiteks ei tohi kliendi poolel olla eristatav, kas suheldakse vahetult lõppserveriga või muu vaheserveriga. Selle näite puhul võimaldab antud piirang paremat jõudlust, kuna vaheserveritega võib lõppserverite koormust hajutada.

Nendest põhipunktidest selgub, et REST on tervikuna ning veebiteenuste liidese disaini kontekstis seejuures siiski üsna üldine, kuna tegu on pigem vaid põhimõtetega ning seega ei täpsusta see ka näiteks tegelikuks realisatsiooniks kasutatavaid tehnoloogiaid. Richardson ja Ruby (2007, 80) kohaselt jätavad RESTi algne definitsioon ja piirangud liiga palju ruumi tõlgendamiseks. Selle tõttu on RESTist väga palju erinevaid tõlgendusi ja arusaami just konkreetse programse realisatsiooni vaates (Richardson, Ruby, 2007, xvi – xvii). Samale probleemile viitab ka Allamaraju (2010, ix - x), mis oli tema väitel ka raamatu RESTful Web Services Cookbook kirjutamise põhjuseks. Antud töö vaates kasutatakse seega veebi tehnoloogiatel põhinevat Richardsoni ja Ruby (2007, Chapter 4) poolt väljapakutud ressurssidel põhinevat arhitektuuri (ingl. k. *Resource Oriented Architecture*), mis on samuti REST stiili järgiv, kuid paneb RESTi konkreetselt veebitehnoloogiate konteksti. Lisaks võetakse aluseks ka Allamarju (2010) raamatus välja pakutud veebi tehnoloogiatest lähtuvad mustrid RESTi rakendamiseks. Täpsemini on antud töö kontekstis detailsel tasemel RESTi kasutust veebiteenuste disaini stiilis käsitletud jaotises 4.3.

REST stiil osutus veebiteenuste liidese disaini kontekstis valituks põhjusel, et RESTi järgides on võimalik paremini soovitud omadusi pakkuda. Omadus 4, mille kohaselt peavad veebiteenuste tarbija ja pakkuja olema üksteisega võimalikult vähe sidestatud, vaates tagab REST tarbija ja pakkuja vahel suurema sõltumatuse kui RPC stiili järgimine (Bloomberg, 2013b). Seda võimaldavad ühtne liides ja piirang, mille kohaselt rakenduse oleku muutmiseks peab kasutama hüpermeediat. Omadus 3, mille kohaselt peab veebiteenuste liides olema võimalikult lihtne ja intuitiivne, on samuti RESTi puhul paremini tagatud. Täpsemalt tagab seda ühtse liidesega seotud alampiirang, mille kohaselt peavad teenuse väljakutsed sisaldama kõike vajalikku infot, et seda saaks töödelda. RPC stiili puhul on tihti teenustest paremaks arusaamiseks suurem roll teenustest lahus oleval dokumentatsioonil (Richardson ja Ruby, 2007, 20).

Eelnevast selgub, et REST veebiteenuste liidese disaini stiili kontekstis tulevad selle tugevused osaliselt välja tänu ühtse liidese piirangule, mis on otseselt seotud liideses

kasutatava protokolliga. Kuna see puudutab rohkem rakenduse arhitektuuri taset, mida käsitleti peatükis 3, siis selle tõttu ongi jaotises 3.1 välja toodud ka REST stiilist tingitud nõuded. Seejuures rakenduse arhitektuuri tasemel valitud HTTP protokoll avaldab ka head mõju teenuste disaini tasemele. Kui RESTi veebiteenuseid realiseerida kasutades HTTP protokolli, siis nõuab see protokolli algselt kavandatud viisil kasutamist, mis tagab parema süsteemide omavahelise vaikimisi protokolli tasemel ühilduvuse. See omakorda vähendab veebiteenuste vaates sidestatust (Richardson ja Ruby, 2007, xvi). HTTP ette nähtud viisil kasutamise puhul saab ära kasutada ka olemasolevat veebi infrastruktuuri ja standardkomponente nagu näiteks vahemälu realiseerivat tarkvara, ilma et peaks seda käsitsi ise arendama (Allamaraju, 2010, 3). Lisaks muudab HTTP kasutus protokolli tasemel veebiteenused kergekaalulisemaks, kuna veebiteenuste vaates pakub protokoll vastavat funktsionaalsust läbipaistavalt ilma veebiteenust ennast mõjutamata. RPC stiili puhul kasutatakse HTTPd pigem transpordi protokollina ning paljuski sõltutakse tegelikult selle stiili puhul muust tehnoloogiast (Richardson ja Ruby, 2007, xvi). Selle tulemusena on süsteemis rohkem kohti, kus võivad täiendavad vead tekkida. Lisaks mõjub täiendava tehnoloogia kasutamine halvasti jõudlusele.

RESTi puhul pole siiski tegu hõbekuuliga. REST on küll RPC ees parem valik, kuid sellegipoolest ei kaota RESTi järgmine täielikult sidestatust tarbija ja pakkuja vahel, kuna sõnumite sisust ja sõnumis sisalduvate linkide tähendustest aru saamist ei ole võimalik praegu automatiseerida. See tuleb endiselt tarbijasse käsitsi sisse programmeerida ja see tekitab vähemalt mingisugusel minimaalsel tasemel sidestatuse. Sama probleemi toob välja Jason Bloomberg (2013b) oma koolituse materjalides ning ka Richardson ja Ruby (2007, xv) juhivad tähelepanu asjaolule, et arvutid pole piisavalt head dokumentide tõlgendajad.

## 4.3 RESTi järgivate veebiteenuste disaini detailsed põhimõtted

Antud töö raames valmis veebiteenuste liidese disaini tulemusena ka toodete valdkonnaga seotud veebiteenuste detailne spetsifikatsioon. Täieliku konkreetse spetsifikatsiooni asemel tuuakse antud jaotises välja selle spetsifikatsiooni loomise käigus analüüsitud REST veebiteenuste disaini detailsed veebitehnoloogiast tulenevad põhimõtted. Need põhimõtted on spetsifikatsioonist olulisemad, kuna need selgitavad, kuidas teenused on disainitud ja kuidas neid edaspidi disainida. Ainult spetsifikatsioonist ei pruugi kõik kasutatud põhimõtted välja tulla.

Detailse realisatsiooni kontekstis tõlgendatakse tihti REST stiili nõudeid vääralt (Fielding, 2008). Kuna Elionis puuduvad hetkel REST veebiteenuste disaini põhimõtted ja RESTi stiili nõudeid tõlgendatakse tihti vääralt, siis võetakse antud töös kasutatud põhimõtted Elioni REST veebiteenuste disainipõhimõtete kogumiku esimese versiooni loomise aluseks. Paljud nendest põhimõtetest on ka Elioni kontektsist sõltumatud, mistõttu saab neid kasutada ka väljaspool Elioni.

Kasutatud põhimõtted on jaotatud erinevatesse kategooriatesse ning need põhimõtted on vajadusel parema selguse huvides illustreeritud konkreetsete spetsifikatsiooni näidetega, mis pärinevad töö käigus loodud spetsifikatsioonist. Enamasti põhinevad need põhimõtted Allamaraju (2010) raamatul RESTful Web Services Cookbook ning Richardsoni ja Ruby (2007) raamatul RESTful Web Services.

### 4.3.1 Üldised põhimõtted

1. REST veebiteenused peavad olema inglisekeelsed. See on Elioni kontekstiga seotud põhimõte, mille põhjendus on välja toodud jaotises 1.3.2.

2. REST veebiteenuste loomiseks kasutatakse HTTP protokolli, kuna see on RESTi puhul *de facto* protokoll. Täpsemalt on protokolli valik põhjendatud jaotises 3.2.1

### 4.3.2 HTTP protokolli järgimine

Selles kategoorias on mõistlik viidata HTTP standardile (Hypertext Transfer Protocol ..., 1999), kuid kuna see standard on põhimõtte jaoks liiga suur, siis selles alampeatükis tuuakse välja põhilised punktid, mis tundusid töö autorile HTTP protokolli uurides olulised.

1. Meetodid GET, HEAD, OPTIONS on ohutud meetodid. See tähendab, et nende kasutamine ei tohi tekitada andmetes muudatusi. Elioni kontekstis on siin lubatud aga logikirjete tekkimine.

2. Meetodid GET, HEAD, OPTIONS, PUT, DELETE peavad tagastama korduval sama sisuga väljakutsel sama tulemuse. POST meetod võib korduval sama sisuga väljakutsel tagastada erinevaid tulemusi.

3. PUT meetodiga asendatakse terve ressurss, mis vastava URLi peal on. Suurte ressursside puhul on mõnikord vajadus teha osalisi uuendusi, selleks et ei peaks üle võrgu saatma liiga palju mittevajalikku informatsiooni. Selleks kasutatakse suurte

ressursside puhul alamressursse. Näiteks ressurss */customers/{customer-id}* tagastab lisaks kliendi baasandmetele ka kliendi juriidilise aadressi. Selleks, et ainult juriidilist aadressi muuta, on loodud alamressurss */customers/{customer-id}/legal-address*, kus PUT meetodi kasutamine on lubatud. Üheks alternatiiviks pakub Allamaraju (2010, 201 - 203) välja PATCH meetodi kasutamist, kuid see pole hea, kuna PATCH meetod suurendab süsteemide sidestatust sellega, et teenuse tarbija peab omama teadmist, kuidas täpselt osaline ressursi muutmine läbi PATCH meetodi käib. Seda HTTP protokoll ette ei kirjuta ja seega on see iga ressursi spetsiifiline. Veel üheks alternatiiviks on kasutada POST meetodit URLil */customers/{customer-id}*, kuid see pole samuti hea, kuna sarnaselt PATCH meetodi kasutamisele peab tarbija teadma, millisel kujul POST meetodi sisu peab olema. Richardson ja Ruby (2007, 101- 102) toovad lisaks välja, et sellisel viisil POST meetodi kasutamine viitab RPC stiilile. Nende põhjuste tõttu on töö autor otsustnud alternatiivide seast osalise andmete muutmiseks kasutada PUT meetodit alamressursil.

4. Vastussõnumite puhul tuleb kasutada HTTP standardiga seotud staatuskoode vastavalt nende kasutuse kirjeldusele. Selle punkti vastu on lihtne eksida, kuid selle vastu eksimise puhul ei pruugi veebi standardkomponendid nagu näiteks vahemälu tarkvara enam oodatavalt käituda. Näiteks kasutatakse konkreetses spetsifikatsioonis 5xx seeria vastuskoode 200 asemel, kui vastuse töötlemiseks vajalik sõltuv süsteem ei ole hetkel kättesaadav.

### 4.3.3 URL disain

1. REST on ressursile orienteeritud stiil. Selle tõttu kasutatakse kasutatakse sarnaselt objekt orienteeritud põhimõttele ressursside puhul nimisõnu, kuna ressurss on midagi, mitte ressurss teeb midagi. Seda soovitavad ka Richardson ja Ruby (2007, 108). Näiteks on toodete URL */products/{product-id}*, mitte */getProduct?id={product-id}*. Samuti saab ka protseduurseid ressursse nimisõnadega nimetada. Näiteks on vaja teeninduskanali rakendustes valideerida sõbranumbrite vastavust ärireeglitega seatud piirangutele. Kuna on soov see täpne äriloogika rakenduste eest ära peita, siis selleks on loodud ressurss */friend-number-validator*. Sinna saab teha GET päringu koos valideeritava numbriga ning saada tagasi valiidsusega seotud info.

2. Kogumikressursid kasutavad nimisõna mitmuse vormi. Kogumikressurss on nimekiri tüüpi ressurss, mis tagastab mitu üht tüüpi ressurssi. Näide: tooted */products*, mis

tagastab toodete objektidest koosneva massiivi, milles omakorda iga toode viitab lingiga ressursile */products/{product-id}*.

3. Päringu parameetreid (ingl. k. *query string*) kasutatakse objekt tüüpi ressursside puhul ainult filtreerimiseks, sorteerimiseks või projektsiooniks ning need parameetrid peavad olema valikulised (Allamaraju, 2010, 138 – 139). Seejuures ei tohi päringu parameetrites kasutada näiteks SQL stiilis päringuid, kuna need võivad tekitada jõudluse probleeme ja tekitavad suuremat sidestatust läbi täiendava teadmise, kuidas päringuid koostada (Allamaraju, 2010, 139). Näiteks ressursi */product-offerings* kaudu saab GET meetodiga küsida kõiki hetkel kehtivaid toote pakkumisi. Küll aga on tihti vajadus küsida just spetsiaalselt ainult konkreetse toote spetsifikatsiooni pakkumisi. Seega saab kasutada ka järgnevat URLi */product-offerings?product-spec-id=12345*.

   Protseduursete ressursside puhul võib päringu parameetreid kasutada sisendparameetrite jaoks. Protseduurne ressurss on jällegi näiteks sõbranumbrite validaator. Seega oleks vastav URL järgmine: */friend-number-validator?number=6400000*.

4. URLides ei kasutata mittehierarhilise struktuuri puhul koma ega semikoolonit, kuna see on vähe levinud praktika. Seda enam, et nii mõnigi programmeerimiskeele teek ei toeta sellist loogikat (Allamaraju, 2010, 77). Selle asemel kasutatakse URLi raja osas kaldkriipsu ning päringu osas kasutatakse eraldi parameetreid.

5. Versioneerimiseks kasutatakse URLis prefiksit raja osas. Seda ainult sellisel vajadusel, et korraga oleks toodangukeskkonnas mitu versiooni ühest ressursist. Näiteks: */v1/customers/{customer-id}* ja */v2/customers/{customer-id}*. Alternatiivina võib kasutada ka versioonitähist päringu parameetris või serveri enda nimes (Allamaraju, 2010, 248). Päringu parameetri kasutamine läheb kirjeldatud päringu parameetri kasutamise põhimõttega vastuollu ning see oleks ka läbipaistmatum, kui versiooni tähistamine URLi raja osas. Elionis pole võimalik kasutada versiooni tähistamist serveri nimes, kuna kõik Elioni veebiteenused on kättesaadavad ühe serveri pealt ning seega mõjutaks versioon kõiki veebiteenuseid. See pole aga piisavalt granulaarne.

6. URLis kasutatakse ressursi nimede loetavuse lihtsustamiseks miinusmärki, mitte *CamelCase* stiili või alakriipsu. Näitena saab välja tuua mitmest sõnast koosnevad

ressursid. Näiteks toote parameetrite ressurss */product-characteristics* või toote pakkumised */product-offerings*.

7. URLide raja osast tulenev hierarhia tuleks hoida võimalikult lame. Põhjus on selles, et nii on URLi struktuur lihtsam ning puudub vajadus kontrollida vastuolusid. Näiteks äriloogika kohaselt võib igal kliendil olla tooteid ja iga toode on alati konkreetse kliendiga seotud. Seega on üheks variandiks kirjeldada ühele konkreetsele tootele vastava ressursi URL kujul */customers/{customer-id}/products/{product-id}*, kuna konkreetne toode kuulub alati konkreetse kliendi juurde. Küll aga tekib sellisest struktuurist vajadus kontrollida täiendavaid vastuolusid. Sellist tüüpi päringu puhul tuleb minimaalselt alati vaadata, kas rakenduses autentinud isikul on õigus antud toodet näha. Küll aga on just sellise URLi puhul vaja ka kontrollida,kas see toode kuulub tõesti URLis sisalduvale kliendile. Vastasel juhul võib koostada vastuoluliste andmetega URLe. Sellepärast on parem URList eemaldada hierarhia ning võimaldada küsida toote andmeid URLilt */products/{product-id}*. Nii peab ainult kontrollima, kas sisse loginud kasutajal on õigus antud toodet näha. Seega saab selle näite põhjal formuleerida hierarhia eemaldamise reegli: kui hierarhia vaates eksisteerib seotud alamobjektil oma unikaalne identifikaator, siis moodusta sellest uus juur URL.

8. Disaini teenuse avalehe URL nii, et see sisaldaks kõiki neid linke või linkide malle, mida tarbijad võivad soovida jõudluse huvides endale n-ö järjehoidjaks salvestada. See aitab vähendada pakkuja ja tarbija vahel sidestatust, kuna veebiteenuse tarbijad ei pea rohkem kui üht URLi teadma. Näiteks on antud töös loodud veebiteenusel ressurss */entry-point*. See ressurss tagastab kõik lingid ja iga lingi kohta *rel* parameetri nime, mille alusel saab vastavat linki tuvastada. See on justkui jooksvalt päringuga laetav konfiguratsioon.

### 4.3.4 Ressursid ja representatsioonid

1. Limiteeri ressursside representatsioonideks kasutatavate erinevate meediatüüpide arvu. Põhjus on selles, et Elionis on enamasti veebiteenused ettevõttesisesed ning puudub vajadus HTTP standardist tuleneva sisu läbi rääkimiseks (ingl. k. *content negotiation*). Loomulikult peab seejuures arvestama HTTP standardiga, kuid võib eeldada, et teenuste kliendid on meie kontrolli all ning näiteks tekstiliste andmestruktuuride puhul kasutame ainult JSON meedia tüüpi. Mida rohkem on samale ressursile erinevaid toetatud meediatüüpe, seda rohkem kulub aega selle realiseerimiseks, testimiseks ning

hilisemaks haldamiseks (Allamaraju, 2010, 135). Võimalusel on seega soovitav hoida selle põhimõtte vaates valikud lahti ning võimaldada lihtsalt uusi meediatüüpe lisada või neid vahetada, kuid ainult siis, kui selleks tõesti on vajadus.

2. Tekstilise ressursside puhul eelista JSON meedia tüüpi. Selle põhjendus on toodud välja antud töö alampeatükis 3.2.1.

3. Ära kasutata ise loodud spetsiifilisi meediatüüpe. Ise loodud meediatüübid on kasulikud vaid siis, kui vastavatel andmetel on palju tarbijaid ja tegu on pigem üldise standardiga (Allamaraju, 2010, 52 – 56). Need aitavad kaasa sõnumi semantikast arusaamise automatiseerimisele, kuid Elioni puhul on nende kasutamine pigem üleliigne ja ebaefektiivne, kuna niikuinii ei hakata looma teenuste tarbijate poole loogikat, mis hakkaks sõnumi sisust aru saama lähtuvalt meediatüübist. Seetõttu kasutatakse näiteks antud töös sõnumite puhul meediatüüpi application/json.

4. Püüa vältida kontroller tüüpi ressursse. Kontroller tüüpi ressursid on sellised, mis muudavad mitut ressurssi korraga ja seega teevad nad tegevusi, mis otseselt ei seostu otseselt ühegi HTTP ühtse liidese meetodiga. Kontroller tüüpi ressursside puhul kasutatakse tavaliselt HTTP POST meetodit. Richardson ja Ruby (2007, 101 – 102) nimetavad selliste ressursside kasutamist POST meetodi ülelaadimiseks, mis tähendab seda, et HTTP ühtne liidese meetod ei sisalda täielikku infot, mida andmetega tegelikult tehakse. Samuti nimetavad nad ka sellist lähenemist pigem liidese halvaks disainiks, kuid teisest küljest väidavad ka, et mõnikord on see vältimatu. Seetõttu tuleks pigem proovida lahendada sama ülesannet kasutades vajadusel mitut ressurssi ja HTTP meetodeid korrektsemalt. Allamaraju (2010, 39 – 43) soovitab kasutada kontroller tüüpi ressursse juhul, kui on soov klientide eest abstraheerida keerukat äriloogikat. Sellist piirangut järgitakse ka Elioni teenuste puhul. Kui tõesti tekib kontroller tüüpi ressursi järgi vajadus, siis tuleks vähemalt igale spetsiaalsele operatsioonile tekitada eraldi ressurss, et ei juhtuks seda, et üks ressurss võimaldab teha mitut erinevat tegevust. See muudaks antud ressursi läbipaistmatuks.

   Näiteks antud töö kontektis oli kasu ressursist */orders/{order-id}/order-state*, mis on ka kontroller tüüpi ressurss, kuna sinna tehes PUT päringu sisuga

```
{
  order-state: "confirmed"
}
```

luuakse tegelikult taustaks tellimuses olnud pakkumistest kliendi toote ressursid. Sellisel juhul on aga tegu õigustatud otsusega, kuna sellega peidetakse tarbija eest konkreetse realisatsiooni detailid.

### 4.3.5 Lingid

1. Iga ressurss peab olema viidatud vähemalt ühest teisest ressursist. Ainuke erand on veebiteenuse sisenemiseks mõeldud URL, mida tarbijad peavad eelnevalt teadma. Vastasel juhul peavad tarbijad vastavad URLid endale kindlasti meelde jätma ning seega on rikutud RESTi põhimõtteid sellega, et teenusest väljaspool olev informatsioon juhib käitumist. See on üks REST teenuste realisatsiooni tüüpilisi probleeme, mille toob Fielding (2008) ka oma artiklis välja.

2. Linkide puhul kasuta alati absoluutseid URLe. See on tarbijate jaoks lihtsam, kuna nad ei pea teadma näiteks baas URLi, millest lingid lähtuvad, ega hoidma meeles, milline oli viimase pöördumise URL. Sellisel juhul ei pea tarbijad ka URLe ise genereerima.

3. Allamaraju (2010, 90 - 91) soovitab JSON representatsioonides kasutada Atom protokolli link elemendile sarnast struktuuri, kus linki iseloomustavateks parameetriteks on viide (ingl. k. *href*) ning lingi seose tüüp (ingl. k. *rel*). Antud töös on kõikidel ressurssidel JSON parameeter nimega „links“, mille sees on massiiv kõikidest selle ressursi juurde kuuluvatest linkidest. Näiteks kliendi toote representatsioonis on järgmised lingid:

```
"links" :
[
  {
    "href" : "https://path.to.api/v1/products/123/children-products"
    "rel" : ""
  },
  {
    "href" : "https://path.to.api/v1/products/123/state"
    "rel" : ""
  },
  {
```

```
    "href" : "https://path.to.api/v1/product-
characteristics/products/123"
    "rel" : ""
  },
  {
    "href" : "https://path.to.api/v1/product-offerings/products/123"
    "rel" : ""
  }
]
```

4. Lingi seose tüüp peab andma informatsiooni selle lingi tähenduse kohta. Allamaraju (2010, 89 – 90) soovitab, et seose tüübi abil peaks olema võimalik vastata järgmistele küsimustele.

   - Mis ressursile see link viitab?
   - Mis on selle lingi olulisus?
   - Mis tegevusi saab klient selle ressursi peal teha?
   - Mis on pöördumise ja vastuse toetatud representatsiooni formaadid?

   Viimast võib seejuures pigem eirata, kuna ühe põhimõttena on soovitatav kasutada võimalikult vähe meediatüüpe ning tekstiliste ressursside puhul on niikuinii soovitatud kasutada ainult JSON formaati. Seega võib seda võtta juba vaikimisi teadmisena, et formaat on sama nagu kõigil teistel ressurssidel. Teistele küsimustele vastamiseks tuleb seega dokumenteerida linkide seose tüübid ning viidata sellele dokumentatsioonile läbi seose tüübi atribuudi. Näiteks antud töös on linkide seosetüübid jaotatud laias laastus kaheks: struktuursed ja käitumuslikud. Struktuursed on sellised lingid, mille eesmärk on pakkuda lisainformatsiooni struktuurselt konkreetse ressursiga seotud teistest ressurssidest. Käitumuslikud on sellised lingid, mille olulisus seisneb selles, et need pakuvad viiteid ressurssidele, mille abil on võimalik mingisuguseid tegevusi teha. Selline jaotamine annab esialgse vihje, millist HTTP meetodit on mõeldud selle lingi kasutamiseks. Struktuursete puhul kasutatakse enamasti GET meetodit, kuid käitumuslike ressursside puhul muutvaid PUT, POST ja DELETE meetodeid. Selline jaotus on analoogne UML diagrammide kategoriseerimisele struktuurseteks ja käitumuslikeks. Näiteks hierarhilistel klienditoodetel on struktuurne link tema alamtoodetele.

```
{
  "href" : "https://path.to.api/v1/products/123/children-products"
  "rel" : "https://path.to.api.doc/link-
relations/structural#children-products"
}
```

Käitumuslik näide on näiteks toote pakkumise lisamine ostukorvi.

```
{
  "href": "https://path.to.api/v1/orders/{order-id}"
  "rel" : "https://path.to.api.doc/link-relations/behavioural#add-to-
order"
}
```

Iga lingi tüübi URL viitab seega selle lingi tüübi dokumentatsioonile.

5. Linkide edastamiseks tekstilist tüüpi ressursside puhul ära kasuta HTTP Link päist. Kuna tekstilist tüüpi andmete puhul eelistatakse JSON formaati ja linke esitatakse selles Atom protokolliga sarnasel põhimõttel, siis puudub vajadus lubada kasutada linke ka HTTP protokolli Link päises. Link päiste puudumine hoiab linkide edastamise ühtsena.

6. Parametriseeritud linkide edastamiseks kasuta URLi malle (ingl. k. *URL template*). Seda soovitab Allamaraju (2010, 101-103), kui server ei suuda täpseid linke genereerida. URL mallid eristuvad tavalistest linkidest sellega, et nad sisaldavad parameetreid. Parameetrid tunneb ära selle järgi, et need on sümbolite „{„ ja „}“ vahel.

   Näiteks antud töös on iga pakkumise juures link, kuidas seda tellimusse lisada. Kuna kasutajal võib aga olla mitu aktiivset tellimust, siis need lingid on esitatud URL mallidena. Seega sisaldab pakkumine järgmist link objekti:

```
{
  "href": "https://path.to.api/v1/orders/{order-id}"
  "rel" : "https://path.to.api.doc/link-relations/behavioural#add-to-
order"
}
```

# 5. Realisatsioonist lähtuv hinnang

Antud magistritöös valitud lahenduse strateegia ellu viimisega on Elionis algust tehtud. Töö valmimise hetkeks on loodud töös käsitletud abstraktsioonikihi rakendus ning pool veebi iseteeninduse toodete müügi ja haldusega seotud teenustest kasutavad uusi abstraheeritud veebiteenuseid. Sellest lähtuvalt saab antud peatükis välja tuua realisatsioonil põhinevad olulisemad hinnangud käesoleva töö käigus tehtud erinevatele valikutele.

Praeguseks valmis oleva realisatsiooni põhjal saab väita, et lahenduse strateegiaks valitud muudatus läbi abstraktsiooni on sobilik lahenduse strateegia. Antud parenduse puhul on tõepoolest tegu suuremahulise muudatusega ning teostuse käigus oli mitmel korral kasu selle suuremast sõltumatusest teistest arendustest, kuna muudatusest mõjutatud süsteeme arendati paralleelselt abstraktsioonikihi arendamise ja juurutamise ajal edasi. Realiseerimise ajal tuli teiste arendustega seoses ette olukordi, kus konkreetsesse versiooni arendatud muudatused tuli versioonist välja võtta. Selle tõttu tekkisid probleemid nende arendustega, mis olid tehtud versioonihaldussüsteemi vaates väljavõetavate muudatustega samas harus. Täpsemalt oli probleeme kattuva koodiga. Selliste probleemide lahendamine võttis üsna palju aega. Sõltumatuse tõttu ei puudutanud need probleemid aga arendatavat abstraktsioonikihti.

Täiendavalt oli kasu sellest, et abstraktsioonikiht asus eraldi rakenduses. See võimaldas rakendusega seotud funktsionaalsust veelgi sõltumatult arendada ja testida. Täpsemini võimaldas see paigaldada rakenduse toodangukeskkonda, ilma et lõppkasutajad seda veel näeksid. Selle tulemusena sai esmaste testidega täiendavalt veenduda, et rakendus ka toodangukeskkonnas tõepoolest töötab. Näiteks sai üle kontrollida võrguligipääsude konfiguratsiooni korrektsuse.

Platvormist lähtudes oli palju kasu sellest, et abstraktsioonikihi rakenduse arhitektuuri vaates otsustati Java mitte PL/SQL kasuks. Täpsemalt seisnes kasu selles, et antud töö käigus tuli tihti ette olukordi, kus lühikese aja jooksul teostati arenduskeskkonnas palju muudatusi ning selle tulemusena toimus pidev koodi uuendamine. Seda oleks andmebaasi lahenduse juures olnud keeruline teostada, kuna mõne muudatuse puhul oleks pidanud andmebaasi palju varasemasse seisu taastama, mis oleks mõjutanud ka teisi arendusi, mis paralleelselt sama

arenduskeskkonna andmebaasi kasutavad. Seega realiseerus varasemalt töös välja toodud PL/SQL keele puudusest tulenev risk.

Veebiteenuste liidese disaini vaates toimis hästi fassaadi mustri rakendamine ühisosa abstraheerimiseks ja äriloogika koondamiseks. Selle tulemusena on nüüd veebi iseteeninduse rakenduse osa, kus kasutatakse abstraktsioonikihti, äriloogika vaates muutunud oluliselt lihtsamaks, keskenduses peamiselt nüüd vaid veebi kasutajaliidese spetsiifika realiseerimisele. Lisaks oli fassaadist ka kasu oma olemuselt sarnaste, kuid keskses süsteemis andmemudeli vaates erinevalt kirjeldatud olemite koondamisel ühe veebiteenuse alla. Näiteks kõigepealt realiseeriti abstraktsioonikihis keskse süsteemi üht tüüpi tootepakkumiste kasutamise võimalus ning hiljem lisandus ka teist tüüpi tootepakkumiste kasutamise võimalus. Kuna abstraheeritud veebiteenuse spetsifikatsioon jäi samaks, siis ei pidanud pärast teise tüübi lisandumist enam veebi iseteeninduse rakendust enam muutma.

Täiendavalt oli kasu veebiteenuste disaini detailsete põhimõtete olemasolust. Selle abil oli lihtne arendajatele neid põhimõtteid selgitada. See omakorda tagas ka lihtsamini teenuste spetsifikatsiooni vaates ühtuse, kuna teenused disainiti juba esimesest korrast korralikult ning mõnikord polnud vaja isegi väga täpset spetsifikatsiooni ette kirjutada. Kuna projekti algusfaasis ei olnud vastavad disaini põhimõtted kokku lepitud, siis oli võimalik täpsemalt näha põhimõtete olemasolust tulenevat kasu. Just algusfaasis esines erinevate veebiteenuste vaates ebakõlasid. Näiteks ei olnud kõik ressursid viidatud, mistõttu olid paljud URLid veebi iseteeninduse rakendusse sisse kirjutatud.

Veebiteenuste liidese disaini stiili vaates ei ole RESTi valik võrreldes RPC stiiliga veel märkimisväärset kasu toonud. Eelkõige on põhjus ilmselt selles, et veebiteenuste liides pole veel piisavalt valmis ning samuti pole veebiteenuste liidesel praegu lisaks veebi iseteenindusele ühtegi teist tarbijat. Näiteks ei olnud palju kasu RESTile omasest tarbija ja pakkuja vähesest sidestatusest. Täiendavalt tuli välja, et RESTi poolt nõutav ühtne liides ei ole alati antud kontekstis kõige optimaalsem lahendus. Selle tõttu tehakse võrreldes RPC stiiliga mõnikord rohkem päringuid abstraktsioonikihi poole.

Üldiselt oli siiski ka teiste tarbijate seas abstraktsioonikihi vastu huvi suur. Näiteks tundsid selle vastu huvi ja soovisid seda kiiremini kasutusele võtta TV iseteeninduse arendajad. See lisas täiendavat kindlust, et antud projekt on tõesti vajalik.

# Kokkuvõte

Antud magistritöö eesmärgiks oli Elioni toodete müüki ja haldust toetava keskse süsteemi ja selle poolt teeninduskanalite rakendustele pakutavate teenuste tehniliste probleemide lahendamiseks valida sobilik lahenduse strateegia ning lähtudes valituks osutunud lahenduse strateegiast, kavandada strateegia esimese etapi lahenduse realisatsioon.

Lahenduse strateegia valikul ja selle esimese etapi realisatsiooni kavandamisel analüüsis ja võrdles autor töö probleemi lahenduse erinevate aspektidega seotud alternatiivseid ja potentsiaalselt sobivaid valikuid. Valikute seast sobivaimate valimisel lähtuti konkreetse aspekti kontekstist ning Elioni kontekstist tulenevatest piirangutest.

Töö tulemusena leiti, et kolme kandidaadi seast on muudatus läbi abstraktsiooni muster sobivaim lahenduse strateegia. Sellest lähtuvalt sisaldas strateegia esimene etapp abstraktsioonikihi kavandamist. Seega on töö täiendavateks tulemusteks abstraktsioonikihti realiseeriva rakenduse arhitektuurile püstitatud nõuded ning neid nõudeid rahuldav arhitektuur. Lisaks rakenduse arhitektuurile püstitati töö tulemusena ka veebiteenuste liidesele, mis antud töö kontekstis moodustabki abstraktsioonikihi, oodatavad omadused ning disainiti nendest omadustest lähtuv veebiteenuste liides.

Realisatsiooni põhjal tehtud hinnangutest lähtudes saab töö kohta välja tuua järgmised olulisemad järeldused.

- Valitud lahenduse strateegia esimesest etapist lähtudes võib järeldada, et tegu on valikutest sobivaimaga ja see annab tulevikuks kindlust, et strateegia tervikuna on samuti sobiv.

- Abstraktsioonikihi rakenduse arhitektuuri puhul Java platvormi valimine võimaldas efektiivsemalt abstraktsioonikihti luua.

- Veebiteenuste liidese disaini vaates on ühtse piisavalt äriloogikat peitva fassaadi loomisest palju kasu, kuna see vähendab oluliselt veebiteenuste tarbija rakenduses olevat äriloogikat.

Antud töö puhul eesmärgid täideti, kuid töö käigus lahendati vaid osa välja toodud keskse süsteemiga seotud probleeme, kuna keskenduti ainult strateegia esimesele etapile. Seetõttu on antud töö puhul edasisteks sammudeks valitud strateegia järgmiste etappide kavandamine ning realiseerimine. Seejuures tuleks kindlasti lähtuda antud töö raames lahendamata jäänud probleemidest. Konkreetsemalt tuli realisatsioonist lähtuvast hinnangust välja, et REST stiili valimine pole jõudnud ennast täielikult õigustada ning seetõttu peaks järgnevates etappides uuesti hindama REST stiili tegelikku eelist käesolevas töös alternatiivina käsitletud RPC stiili ees.

# Summary

Main goals of this thesis were to choose an improvement strategy for Elion's product catalogue and customer management system and its services, which currently contain technical debt, and based on chosen strategy design the first stage of the implementation.

In order to choose the best suitable improvement strategy and design the first stage of the implementation, the author of this thesis compared and analysed potentially suitable choices concerning multiple aspects of the solution. The most suitable choices were made based on Elion's context and the specifics of the aspect in question.

As a result of this thesis, branch by abstraction was found to be the best solution among three possible alternatives. Based on that choice, the first stage of the implementation was about the design of the abstraction layer. Therefore, requirements for the abstraction layer application architecture and an application architecture meeting these requirements are additional results of this thesis. In addition to the application architecture expected properties for the new web service interface were also worked out and web service interface with these properties was designed. In this context the web service interface forms the abstraction layer.

Based on the actual implementation, important conclusions are listed below.

- Based on the implementation of the first stage, the chosen strategy appears to be the best choice, which gives reason to believe that other future stages of the implementation are successful as well.

- The choice of Java platform for the abstraction layer application architecture enabled efficient development of the abstraction layer application.

- Using the façade pattern to hide business logic specifics in the web service interface was very useful, because it minimizes the needed amount of business logic in the consumer applications.

Goals of this thesis were achieved. However, due to the focusing on the first stage of the implementation, only part of the described problems was resolved.

Therefore, next steps for future development include the design and implementation of the remaining stages of the chosen strategy. The future development should definitely take into account these unresolved problems of this thesis. Furthermore, based on the actual implementation, the advantages of using REST architecture style over the compared RPC style in the web services interface design were not evident in this thesis. Therefore, it is essential to re-evaluate the choice of using REST architecture style instead of RPC style.

# Kasutatud kirjandus


1. Allamaraju, S. (2010). RESTful Web Services Cookbook. First Edition. Sebastopol : O'Reilly Media, Inc.

2. Bloomberg, J. (2013a). The Agile Architecture Revolution. New Jersey : John Wiley & Sons, Inc.

3. Bloomberg, J. (2013b). LZA SOA training & certification: koolitus. 24 – 27 september 2013. Tallinn

4. Design Patterns: Elements of Reusable Object-Oriented Software. (2000). / E. Gamma, R. Helm, R. Johnson, J. Vlissides. 21th Printing. USA : Addison-Wesley

5. Fielding, R. (2000) Architectural Styles and the Design of Network-based Software Architectures : doktoritöö. University of California, Irvine

6. Fielding, R. (2008). REST APIs must be hypertext-driven [WWW] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (01.05.2014)

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

8. Fowler, M. (2003). Domain Logic and SQL [WWW] http://martinfowler.com/articles/dblogic.html#LookingAtPerformance (01.05.2014)

9. Fowler, M. (2006). Continuous Integration [WWW] http://martinfowler.com/articles/continuousIntegration.html (01.05.2014)

10. Fowler, M. (2007). FeatureToggle [WWW] http://martinfowler.com/bliki/FeatureToggle.html (01.05.2014)

11. Fowler, M. (2009). FeatureBranch [WWW] http://martinfowler.com/bliki/FeatureBranch.html (01.05.2014)

12. Fowler, M. (2014). BranchByAbstraction [WWW] http://martinfowler.com/bliki/BranchByAbstraction.html (01.05.2014)

13. Hammant, P. (2007). Introducing Branch By Abstraction [WWW] http://paulhammant.com/blog/branch_by_abstraction.html (01.05.2014)

14. Hammant, P. (2011). Avoiding 'Big Bang' for Branch By Abstraction [WWW] http://paulhammant.com/2011/05/13/avoid-big-bang-for-branch-by-abstraction/ (01.05.2014)

15. Humble, J. (2011). Make Large Scale Changes Incrementally with Branch By Abstraction [WWW] http://continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by-abstraction/ (01.05.2014)

16. Hypertext Transfer Protocol -- HTTP/1.1 [WWW] http://tools.ietf.org/html/rfc2616 (01.05.2014)

17. Information Framework (SID) Overview [WWW] http://www.tmforum.org/Overview/14018/home.html (01.05.2014)

18. IT terministandardi sõnastik [WWW] http://eki.ee/dict/its (01.05.2014)

19. JSON [WWW] http://www.json.org/ (01.05.2014)

20. Kas üks küsimus kliendisuhte tugevuse mõõtmiseks on piisav? [WWW] http://www.emor.ee/public/documents/uuringusuunad/NPS_ja_TRIM.pdf (01.05.2014)

21. Martin, R. C. (2009). Clean Code: A Handbook of Agile Software Craftsmanship. USA : Prentice Hall.

22. Nielsen, J. (2010). Website Response Times [WWW] http://www.nngroup.com/articles/website-response-times/ (01.05.2014)

23. Object Oriented Software Development: Object-Oriented Programming [WWW] http://cs.smu.ca/~porter/csc/465/notes/oop.html (01.05.2014)

24. Rehn, C. (2012). Continuous Integration: Aspects in Automation and Configuration Management [WWW] http://www.christian-rehn.de/wp-content/uploads/downloads/2012/04/seminar_ci.pdf (01.05.2014)

25. Richardson, L., Ruby, S. (2007). RESTful Web Services. First Edition. Sebastopol : O'Reilly Media, Inc.

26. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) (2007) [WWW] http://www.w3.org/TR/soap12-part1/ (01.05.2014)

27. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models. (2011). 4.2 Product quality model : ISO/IEC 25010:2011

28. TM Forum Frameworx [WWW] http://www.tmforum.org/TMForumFrameworx/1911/Home.html (01.05.2014)

29. Vallaste, H. (2014). e-Teatmik: IT ja sidetehnika seletav sõnaraamat [WWW] http://www.vallaste.ee (01.05.2014)

30. Web Services Description Language (WSDL) (2007) [WWW] http://www.w3.org/TR/wsdl20/ (01.05.2014)