TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Veebi- ja andmebaasipõhise metamodelleerimise vahendi üleviimine andmebaasi trigeritel põhinevaks süsteemiks

Magistritöö

Üliõpilane: Andro Põlluste
Üliõpilaskood: 111688 IAPM
Juhendaja: Erki Eessaar

Tallinn
2016

Autorideklaratsioon

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

(*kuupäev*)	(*allkiri*)

Annotatsioon

Veebi- ja andmebaasipõhise metamodelleerimise vahendi üleviimine andmebaasi trigeritel põhinevaks süsteemiks

Käesoleva magistritöö eesmärk on realiseerida veebi- ja andmebaasipõhise metamodelleerimisvahendi WebMeta üleviimine SQL-andmebaasi trigeritel põhinevaks süsteemiks ning kirjeldada seda protsessi. Kõigepealt tutvustatakse tarkvara evolutsiooni mõistet, olemasolevat metamodelleerimisvahendit ning tuuakse välja põhjused, miks on vaja ette võtta tarkvara uuendamine. Seejärel kirjeldatakse vahendi evolutsioneerimist ning tehtud valikuid. Magistritöö tulemusena luuakse metamodelleerimisvahendi WebMeta uus versioon koos uue arhitektuuri, kujunduse ning funktsionaalsusega administraatorile. See võimaldab neil defineerida modelleerimiskeeli ja nende elemente ning sellega algatada andmebaasiobjektide loomine, mis on vajalikud, et lõppkasutajad saaksid loodud modelleerimiskeeli kasutada.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 74 leheküljel.

Abstract

Transforming a Web-Based and Database-Based Metamodelling Tool into a Database Triggers Based System

The purpose of this thesis is to transform web- and database based metamodelling tool WebMeta into system that is based on SQL database triggers and describe the process. In the beginning, software evolution concept and the existing metamodelling tool are introduced together with reasoing, why there was a need for upgrade. After this, there is description about evolution process and choices made. As a result of this thesis a new version of the metamodeling tool WebMeta is created with new architecture, user interface and functionality for modeler. One can define modelling languages and their elements and thus initiate the creation of database objects that are needed to support the use of modeling languages by end users.

The thesis is in Estonian and contains 74 pages of text.

Põhimõisted ja lühendid

CASE system	*Computer Aided Software Engineering system*, tarkvara, mis on mõeldud süsteemide kavandamiseks kasutades ühte või mitut modelleerimiskeelt ja pakub lisaks funktsionaalsusi nagu mudelite kontroll, mudelite teisendamine sh teisendamine teisteks mudeliteks, koodiks, dokumentideks, testideks.
Meta-CASE system	*Meta Computer Aided Software Engineering system*, metamodelleerimisvahend. Vahend, mis on mõeldud CASE vahendite loomiseks.
UML	*Unified Modelling Language*, ühtne mudelikeel
ORM	*Object Relational Mapping*, objektrelatsiooniline vastavusse seadmine
MVC	*Model View Controller*, Mudel Vaade Kontroller

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Tõenäoliselt on kõik, kes on tarkvaraarendusega kokku puutunud, kogenud, et aja jooksul tarkvara „vananeb“ ning vajab uute funktsionaalsuste lisamiseks järjest rohkem tööd. Samuti amortiseerub tarkvara nii visuaalselt kui ka sisuliselt ning samade funktsionaalsuste loomiseks tekib lihtsamaid ning elegantsemaid mooduseid. Seda nähtust nimetatakse ka tarkvara evolutsiooniks. Mõnikord osutub kasulikumaks kogu tarkvara uuesti kirjutamine, et viia sisse kaasaegsemaid tehnoloogiaid ja lähenemisi.

Käesoleva magistritöö raames viiakse Rünno Sgirka poolt loodud metamodelleerimisvahend WebMeta veebi- ja andmebaasipõhiselt süsteemilt SQL-andmebaasi trigeritel põhinevaks süsteemiks. Kuna tegemist on väga mahuka muutusega, siis osutus mõistlikumaks luua kogu rakendus algusest peale, võttes arvesse vahepeal kogunenud teadmisi ning uusi vajadusi. Nimetatud metamodelleerimisvahend pakub võimalust defineerida uusi modelleerimiskeeli ja neis sisalduvaid mudelielemente, mille põhjal luuakse vahendi poolt automaatselt andmebaasiskeemid, -tabelid, -veerud ja -rollid. Need on vajalikud, et lõppkasutaja saaks valitud modelleerimiskeeles mudeleid koostada. Valitud lähenemine pakub alternatiivi sisse kodeeritud funktsionaalsusega piiratud modelleerimisvahenditele.

Keskkond jaguneb kahe kasutajagrupi vahel: administraatorite ning lõppkasutajate vahel. Administraatorid defineerivad uue keele, mille põhjal genereeritakse lõppkasutajale pakutav andmebaasiskeem. Lõppkasutajatel on seejärel võimalus näha loodud vahendi üldist ülesehitust ning sisestada ja hallata andmeid mudelites. Töös käsitletakse administraatori vaate parendamist ning viimist üle trigeritel põhinevaks süsteemiks.

Käesoleva magistritöö ülesehitus on järgmine. Kõigepealt kirjeldatakse tarkvara evolutsiooni mõistet ning sellega seotud seaduspärasusi. Seejärel vaadeldakse WebMeta vahendit enne evolutsioneerimist, tuuakse välja põhjused, miks on vaja uuendamine ette võtta ning kirjeldatakse metamodelleerimisvahendit pärast evolutsioneerimist. Järgnevas peatükis tuuakse välja uuendamiseks valitud tehnilised lahendused, nende eelised ja puudused. Sellele järgneb uue versiooni arendamise kirjeldus. Lõpetuseks on välja toodud ideed edaspidiseks arendustegevuseks antud rakenduse juures.

2. Tarkvara evolutsioon

Käesolevas töös toimub veebi- ja andmebaasipõhise metamodelleerimise vahendi (Sgirka 2008; Eessaar ja Sgirka 2010a; Eessaar ja Sgirka 2010b) üleviimine andmebaasi trigeritel põhinevaks süsteemiks. See protsess on näide tarkvara evolutsioonist. Metamodelleerimise vahendid üldiseslt ja see vahend konkreetselt võimaldavad kasutusele võtta uusi modelleerimiskeeli või parandada olemasolevaid (luues või muutes nende keelte kasutamiseks mõeldud keskkondi) ning sellega soodustada erinevate tarkvarasüsteemide evolutsiooni. Seega tundub mõistlik kõigepealt kirjutada veidi tarkvara evolutsiooni kohta.

Tarkvara evolutsioon on mõiste, mida kasutatakse arvutiteaduses kirjeldamaks tarkvara loomise ning hilisema uuendamise protsessi. Seda protsessi kirjeldatakse Lehmani seadustega (Lehman 1996). Need kehtivad reaalse maailma probleemide lahendamiseks loodud tarkvara kohta, mis sõltub selle keskkonnast ning peab olema kohandatav muutuvate nõudmiste ja olukordadega. Sellist tarkvara nimetatakse E-tüüpi tarkvaraks. Kuigi neid nimetatakse seadusteks kirjeldavad need tegelikult seaduspärasusi, mis on üldjuhul tarkvara loojate kontrolli alt väljas.

Lehmani seadused (Lehman 1996):

1) Pidev muutumine – kasutuses olevat tarkvara peab pidevalt muutuvate nõudmistega kohandama, muidu muutub aja jooksul selle kasutamine üha vähem rahuldust pakkuvaks.

2) Kasvav keerukus – tarkvara arenedes selle keerukus kasvab, kui ei tehta tööd keerukuse kontrolli all hoidmiseks või vähendamiseks. Muudatusi tehes kasvab süsteemi seoste keerukus ning selle läbi kogu süsteemi entroopia. Kui süsteemi keerukuse kasvu ei piirata, siis kasvab oluliselt süsteemi rahuldavas töökorras hoidmise keerukus. Kui aga võetakse ette meetmeid piiramaks keerukust, siis jääb vähem aega süsteemi arenguks. Kuna ressurssi on alati piiratud kogus, siis süsteemi vananedes aeglustub süsteemi areng hoolimata kasutatavast strateegiast. Praktikas sõltub nende kahe meetme tasakaal süsteemile saadavast tagasisidest. Kui keerukus kasvab liiga suureks, siis on parim lahendus tarkvara ümberkirjutamine ja „puhtalt lehelt alustamine“.

3) Enese regulatsioon – tarkvara evolutsiooni protsess on isereguleeruv.

4) Organisatsioonilise stabiilsuse säilitamine (sama töö juures) – tarkvara evolutsiooni saavutamiseks tehtava töö hulk on tarkvara eluea jooksul stabiilse suurusega.

5) Tuttavlikkuse hoidmine – areneva tarkvara erinevad väljalasked pakuvad suhteliselt ühesuguse hulga muudatusi. Liiga suur muudatuste hulk ühes väljalaskes muudab muudatuste tegemise ning muudetud tarkvara tundmaõppimise liiga keeruliseks.

6) Pidev kasv – tarkvara pakutavate funktsionaalsuste hulka tuleb pidevalt suurendada, et tagada kasutajate jätkuv rahulolu tarkvaraga. Kasv tuleb lisanduvate nõudmiste ja kasutaja vajaduste ning tagasiside tulemusena. Teisalt mõjutavad arendamist piirangud, näiteks eelarve ning seatud tähtajad, mis tingivad teatud funktsionaalsuste edasi lükkamise järgmistesse arenduse etappidesse.

7) Vähenev kvaliteet – kui tarkvara pidevalt ei hooldata ning ei kohandata muutuva keskkonnaga, siis kasutajadd tunnevad tarkvara kvaliteedi vähenemist.

8) Tagasiside süsteemid – tarkvara loomine on mitmetest tsüklitest koosnev, mitmeid osapooli hõlmav, mitmeastmelise tagasisidega protsess. Sellest tuleb aru saada ja seda aktsepteerida, et oleks võimalik seda protsessi edukalt muuta või parandada. Koos tarkvara vanuse kasvuga muutub tavaliselt ka selles muudatuste tegemine raksemaks.

3. Metamodelleerimisvahendi WebMeta evolutsioneerimine

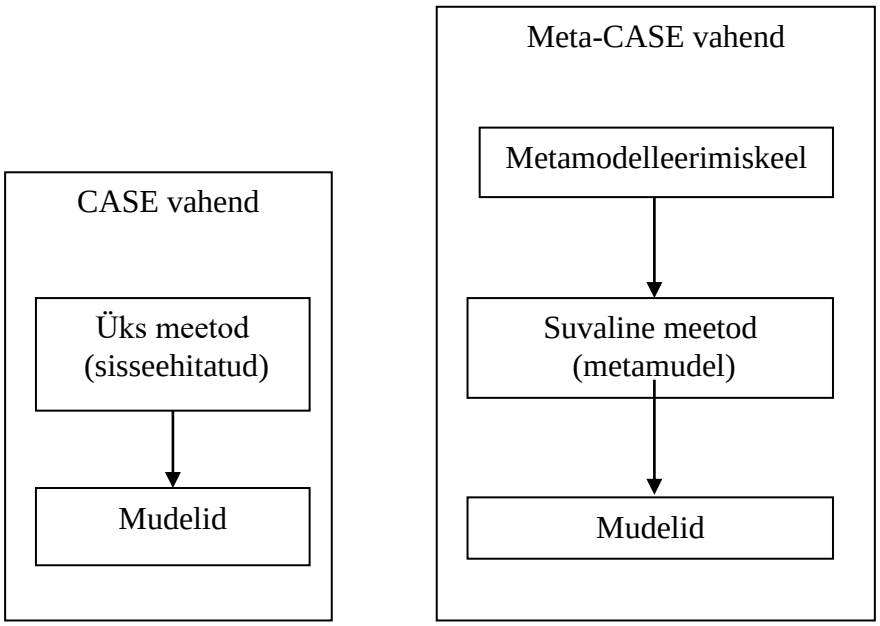
Tarkvaraarendusele aitab kaasa süsteemide arendamiseks mõeldud tarkvaraliste arendusvahendite ehk CASE (ingl Computer Aided Software Engineering) vahendite kasutamine. Need aitavad automatiseerida, hallata ja lihtsustada tarkvara arendusprotsessi. CASE vahendite pakutavate funktsionaalsuste hulka võivad kuuluda vahendid nõuete analüüsiks, arendusprotsessi kirjeldamiseks ja arendustöö ajakava koostamiseks (E-teatmik, 2015). Antud töös käsitlen CASE vahenditest modelleerimisvahendeid. Need pakuvad ühte või mitut modelleerimiskeelt ning lisaks funktsionaalsuseid, nagu mudelite kontroll ja mudelite teisendamine. Mudelit võib teisendada teiseks mudeliks, koodiks, dokumentatsiooniks, testideks. Modelleerimiseks mõeldud CASE vahendite näideteks on Rational Rose ja Enterprise Architect, mida töö kirjutamise ajal (2015) kasutatakse TTÜ õppetöös.

Kuigi CASE vahendid pakuvad palju kasulikku, kipuvad need olema suured ning halvasti kohandatavad programmid. See pole probleemiks seni, kuni ei teki spetsiifilisi vajadusi kohanduste järgi, mis soodustaksid konkreetse ettevõtte või arendusmeeskonna paremat toimimist, kuid mille sisse viimine olemasolevatesse CASE vahenditesse oleks keerukas ja kulukas, kui mitte võimatu.

Selle probleemi lahendamiseks on välja töötatud Meta-CASE ehk metamodelleerimise vahendid. „Meta“ tähendab tase kõrgemal olemist. Metamodelleerimine tähendab tarkvaraarenduse tulemite (nt mudelite) loomiseks mõeldud keelte kirjeldamist. Metamodelleerimise vahend on mõeldud uute CASE vahendite loomiseks. Selles on võimalik ära kirjeldada uues CASE vahendis kasutatavad tarkvarakeeled, selles keeles loodavate tulemitega tehtavad tegevused ning lõpuks sellest kirjelduses mingil viisil töötav CASE vahend genereerida. Metamodelleerimise vahendite kasutajad jagunevad seega laias plaanis kaheks: administraatorid (metamodelleerijad), kes defineerivad uusi CASE vahendeid ning lõppkasutajad (modelleerijad), kes kasutavad loodud vahendeid arendustöö tegemiseks. Klassikalise modelleerimisvahendi ning metamodelleerimisvahendi erinevus on välja toodud allpool (vt Joonis 1). Klassikalises modelleerimisvahendis saab kasutada ühte või mitut sisseehitatud modelleerimiskeelt (nt UMLi), mida pole võimalik üldse muuta või saab muuta ainult piiratud ulatuses (nt UML puhul sisseehitatud laiendusmehhanismi abil uusi modelleerimise profiile kirjeldades. Metamodelleerimise vahendi abil saab luua uusi modelleerimisvahendeid, mis toetavad uusi

modelleerimiskeeli. Need keeled võivad olla üldised keeled või mingi valdkonna jaoks spetsialiseeritud keeled (ingl Domain Specific Languages).

Sellise täiendava abstraktsiooni kihiga modelleerimisvahendite hulk on väga piiratud. Otsingu tulemustena jäi suurima tegutsejana silma MetaEdit+, mis pakub kaasaegset modelleerimiskeelte loomise ning kasutamise keskkonda (Metacase, 2015).



Joonis 1: CASE vahend vs Meta-CASE vahend (The Full Wiki, 2015).

Allpool on veel välja toodud metamodelleerimisvahendeid, kuid kodulehtede järgi nende arendamisega väga aktiivselt ei tegeleta:

- Atom3 (http://atom3.cs.mcgill.ca/distrib.dtml)
- GME (http://www.isis.vanderbilt.edu/projects/gme)

Kõik mainitud vahendid on saadaval ainult kasutaja arvutisse installeeritavate rakendustena, veebipõhiseid metamodelleerimise vahendeid mul leida ei õnnestunud. Küll on aga olemas hulgaliselt veebipõhiseid modelleerimisvahendeid (Cacoo, Creately, MxGraph jne), millest annab hea ülevaate Lauk(2013).

Ühe veebipõhise metamodelleerimisvahendi prototüübi lõi Rünno Sgirka oma magistritöö raames 2008 aastal (Sgirka 2008) ning on seda ka hilisemalt täiendanud (Eessaar ja Sgirka

2010a; Eessaar ja Sgirka 2010b). Järgnevalt vaatlen seda vahendit evolutsioneerimise seisukohast, toon välja rakenduse hetkeseisu, kitsaskohad ning parendamist vajavad kohad ning seisu pärast evolutsioneerimist. Järgnevates peatükkides toon välja tegevused, mida tegin WebMeta vahendi parendamiseks. Vahe tegemiseks olemasoleva ja uue süsteemi vahel kasutan vastavalt nimetusi WebMeta3 ja WebMeta4.

3.1 WebMeta vahend enne evolutsioneerimist

Kasutan nime WebMeta3, kuna enne käesolevat arendustsüklit oli vahend juba läbinud mitu varasemat arendustsüklit.

WebMeta3 on veebipõhise liidesega metamodelleerimisvahendi prototüüp, mis pakub võimalust defineerida erinevaid modelleerimiskeeli (sh nendes sisalduvaid mudelielemente) ning lõppkasutajal neid kasutades koostada ja hallata uusi mudeleid. Rakendus on jagatud kahe keskkonna vahel: administreerimine ehk keelte loomise ning lõppkasutaja keskkond. WebMeta3 on andmebaasipõhine, st et nii modelleerimiskeelte kirjeldused kui ka nendes loodavad mudelid salvestatakse andmebaasis (täpsemalt PostgreSQL andmebaasis).

Rakendus on kirjutatud PHP keeles ja kasutab Apache veebiserverit ning andmebaasisüsteemina PostgreSQL serverit. Tutvuda saab sellega aadressil http://viktor.ld.ttu.ee/metacase3. Järgnevalt tutvustan praeguse WebMeta rakenduse tööpõhimõtteid.

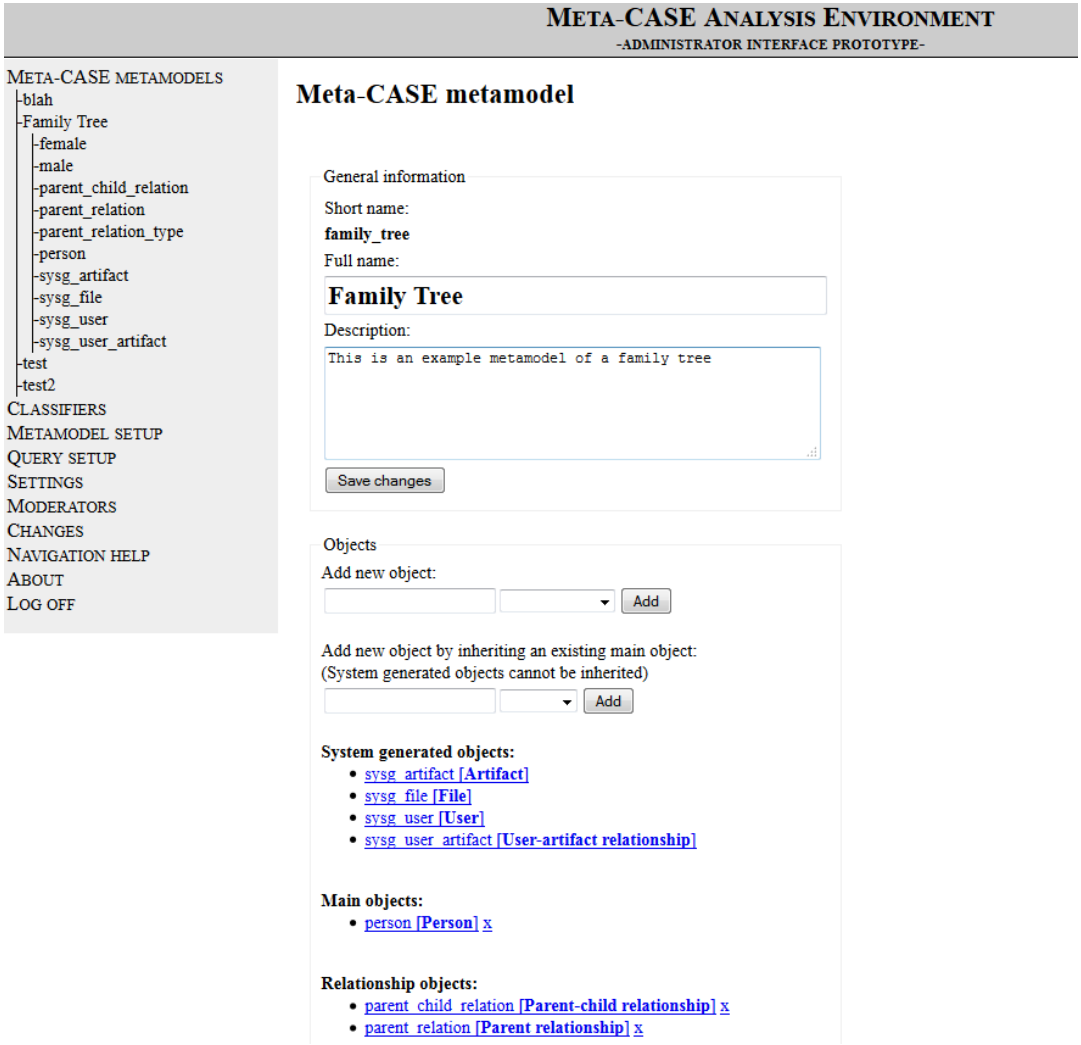
Administreerimisliidese abil saab luua ja hallata modelleerimiskeelte kirjeldusi ning neid kasutada võimaldavate CASE vahendite kirjeldusi. Modelleerimiskeelte/CASE vahendite kirjeldused on skeemis *public* olevates baastabelites (edaspidi nimetan baastabeleid lihtsalt tabeliteks). Iga defineeritud modelleerimiskeele kohta luuakse andmebaasis eraldi skeem. Näiteks kui luuakse modelleerimiskeel *keel*, siis keele nimi, kirjeldus ja muud andmed (neid võib nimetada keele metaandmeteks), talletatakse skeemis *public* olevates tabelites ning lisaks luuakse andmebaasis skeem *keel*. Nii kirjelduste salvestamine kui vastavate andmebaasiobjektide loomine peab olema koondatud ühte transaktsiooni. PostgreSQLis on see võimalik, sest andmekäitluskeele ja andmekirjelduskeele lauseid saab ühte transaktsiooni koondada. Andmebaasiskeem *public* luuakse PostgreSQL andmebaasis automaatselt. Nagu kõigil teistel skeemidel on ka sellel unikaalne nimi, mis tähendab, et süsteemis pole võimalik defineerida modelleerimiskeelt *public* (sest andmebaasis ei saa olla kaks ühesuguse nimega skeemi).

Iga modelleerimiskeel kirjeldab hulga mudelielemente, mida saab kasutada selle keele abil loodud mudelites. Antud süsteemis kutsutakse mudelielemente **objektideks**. Kui kirjeldatakse uus objekt, siis objekti andmed (nimi, kirjeldus jne) talletatakse jällegi skeemis *public* olevates tabelites ning paralleelselt luuakse metamudelile (keelele) vastavasse skeemi objektile vastav tabel. Seega kui keelde *keel* lisatakse objekt *mudeli_element*, siis skeemi *keel* lisatakse tabel *mudeli_element*.

Igal mudelielemendil (objektil) võivad olla omadused ehk atribuudid. Antud süsteemis nimetatakse neid **objekti alamobjektideks**. Kui objektile lisatakse alamobjekt, siis alamobjekti andmed (nimi, kirjeldus) talletatakse skeemis *public* olevates tabelites ning paralleelselt luuakse metamudelile vastavasse skeemi objektile vastavasse tabelisse alamobjektile vastav veerg. Näiteks, kui metamodelleerimise skeemis *keel* lisatakse objektile *mudeli_element* alamobject *alamelement*, siis andmebaasi skeemis *keel* lisatakse tabelisse *mudeli_element* veerg *alamelement*. (Sgirka, 2008)

Kasutajad on jagatud kolme gruppi: administraatorid, moderaatorid ning lõppkasutajad. Administraatorid defineerivad uusi modelleerimiskeeli ning nende keelte kasutamiseks mõeldud kasutajakeskkondi. Moderaatoritel on õigus teha keeltes muudatusi, kuid neid mitte kustutada. Lõppkasutajad näevad administraatori defineeritud menüüsid ning saavad keeles luua mudeleid. *Public* andmebaasiskeemis olevates metaandmete tabelites on defineeritud lõppkasutajale nähtav menüüde ülesehitus. Lõppkasutaja hallatavad andmed (mudelid) on metamudelile vastavas skeemis olevates tabelites. Allpool on välja toodud WebMeta3 administreerimise keskkonna vaade enne evolutsioneerimist (vt Joonis 2).

Administraator saab defineerida lõppkasutajatele kontrollpäringuid, et kasutajal oleks võimalik kontrollida mudelite terviklikkust ja kooskõlalisust. Mudelite terviklikkuse ja kooskõlalisuse kontroll ei toimu automaatselt vaid lõppkasutaja saab alustada mudelite koostamist ning sobival hetkel käivitada kontrolle, et leida mudelitest parandamist vajavad kohad (või kui neid ei ole, siis veenduda mudeli headuses).



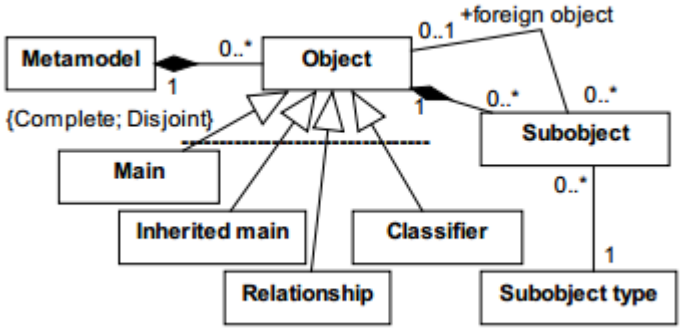
Joonis 2: WebMeta administraatori vaade enne evolutsioneerimist

3.2 Evolutsioneerimise põhjused

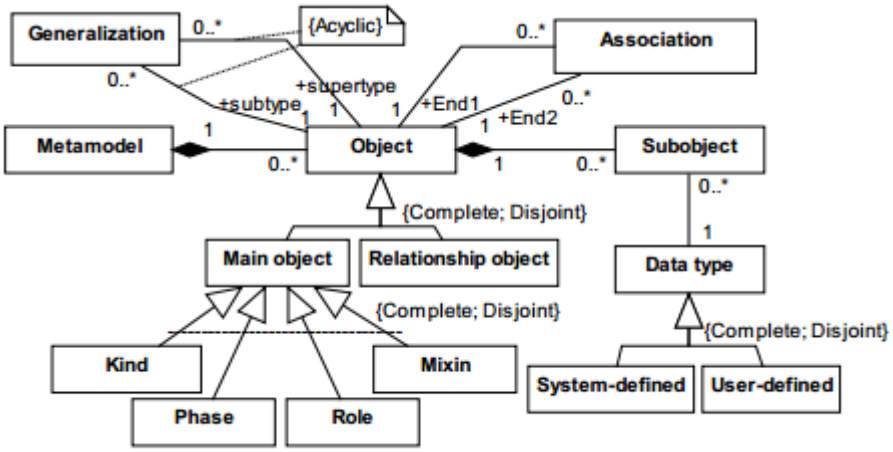
Kuigi olemasolev prototüüp töötas, siis sai uuendamine ette võetud. Seda nii eesmärgiga parandada koodi kvaliteeti, tarkvara arhitektuuri kui ka viia sisse funktsionaalseid täiendusi. Need põhjused ühtivad Lehmani tarkvaraevolutsiooni seadustest tulenevate vajadustega muudatuste järgi toimivas tarkvaras. Järgnevalt ära toodud olulisemad põhjused muutusteks.

Kõige olulisemaks ajendiks oli vajadus muuta keelte kirjeldamiseks kasutatava keele (mida kirjeldab meta-metamudel) struktuuri vastavalt Eessaar ja Sgirka (2013) artiklis toodud nõudmistele. See tähendab sisulisi muudatusi kogu vahendi ülesehituses. Näiteks peab olema võimalik senisest palju paremini kirjeldada üldistusseoseid objektide vahel ning süsteem peab nende üldistuste puhul kontrollima teatud reeglitest kinnipidamist. Graafiliselt on erinevused

välja toodud järgnevatel joonistel (Vt Joonis 3 ja Joonis 4), kus on näha WebMeta4 täiustatud metamodelleerimise keel. (Eessaar ja Sgirka, 2013)



Joonis 3: Metamodelleerimissüsteemi meta-metamudel WebMeta3's (Eessaar ja Sgirka, 2013)



Joonis 4: Metamodelleerimissüsteemi meta-metamudel WebMeta4's (Eessaar ja Sgirka, 2013)

Teiseks oluliseks ajendiks oli soov delegeerida andmebaasiobjektide loomise ülesanne rakenduselt andmebaasi trigeritele. See mis seni genereeriti PHP's dünaamilise SQL abil, tuli viia PostgreSQL andmebaasis olevate trigerite kohustuseks. Eesmärgiks on teha kasutajaliideste loomine võimalikult lihtsaks, et rakenduste ümberkirjutamine uusi kasutajaliideste loomise tehnoloogiaid arvestades oleks võimalikult kerge ning et seda saaksid teha ka inimesed (nt üliõpilased), kellel pole süvendatud teadmisi metamodelleerimisest, metamudelitest ning PostgreSQList. Kasutajaliidese poolt tulevad muudatused (metaandmete) tabelitesse ning nende tabelitega seotud trigerite abil realiseeritud loogika hoolitseb selle eest, et õiged anbmebaasiobjektid saaksid õiges järjekorras loodud/muudetud/kustutatud. Kui tulevikus on

plaanis muuta loogikat, mille alusel luuakse metaandmete põhjal andmebaasiobjektid (nt võtta kasutusele kuuenda normaalkuju tabelid, mille korral igale kontseptuaalsele atribuudile vastab eraldi andmebaasi tabel), siis pole vaja muuta administraatori rakendust, vaid kõik muudatused tuleb teha trigeri koodis. See vähendab kasutajaliidese sidusust ülejäänud süsteemiga ning teeb võimalikuks nende üksteisest sõltumatu arendamise.

Sealt tuleneb ka uue versiooni kolmas põhjus, mis samuti mainitud Lehmani seaduste all, et senise rakenduse koodi ümberkirjutamine oleks ebamõistlik võrreldes rakenduse nullist üles ehitamisega. Selle otsuse kasuks rääkis ka see, et olemasolev kood oli küllalt eakas ning viimaks kvaliteeti uuele tasemele oli mõistlik värskemaid teadmiseid kasutades otsast peale alata.

3.3 WebMeta vahend pärast evolutsioneerimist

Nagu ka Lehmani seadused viitavad, on tarkvara arenduse puhul tegemist piiratud ressursside jagamisega. Seega ka antud töö puhul otsustati uuendada ainult metamodelleerimise vahendi administratiivset poolt. Lõppkasutaja vaates on endiselt üsna palju puudujääke ning selle edasiarendamine toimub järgmistes arendustsüklites. Järgnevalt esitan ülevaate sellest, mis evolutsioneerimise käigus juurde tehtud ja mida muudetud. WebMeta4ga on võimalik tutvuda aadressil http://apex.ttu.ee/ WebMeta/.

Oluline punkt on andmebaasisüsteemi pakutavate vahendite abil realiseeritava süsteemi alamosa suurenemises võrreldes vana programmiga. Kasutajate halduseks võeti kasutusele Kõikehõlmav Andmebaasi Turvalisuse Mudel (*A Comprehensive Database Security Model*). (Downs, 2009). Selle puhul jäetakse kasutajate tuvastamine ja neile piirangute rakendamine andmebaasisüsteemi ülesandeks, seades igale rakenduse kasutajale vastavusse andmebaasi kasutaja. Valitud lähenemine võimaldab andmebaasi tasemel väga detailselt defineerida kasutajate juurdepääsu andmebaasi objektidele (mis selle süsteemi puhul vastavad erinevatele keeltele ja nende elementidele). See vähendab rakenduses teostatavate kontrollide vajalikkust. Tänu sellele muutub võimalikuks ja praktiliseks andmebaasis õiguste jagamine läbi andmebaasi rollide. See lihtsustab tulevikus süsteemi täiendamist ja uute rollide lisamist. Samuti lihtsustab see lähenemine andmetega (selle süsteemi puhul mudelite ja modelleerimiskteelte) tehtavate tegevuste logimist. Info selle kohta, kes andmeid muutis, registreeritakse trigerite abil ning seda infot ei ole vaja salvestada ja koguda rakendusest. Kahjuks on sellel lahendusel ka miinuseid. Iga kasutaja jaoks peab olema aktiivne andmebaasi sessioon ning ei saa kasutada ühenduste jagamist

(ing *connection pooling*). Võimalik juurdepääsupiirangute detailsusaste sõltub kasutatavast andmebaasisüsteemist. PostgreSQLis (9.4) on see õnneks piisavalt detailne.

Teine oluline eelis andmebaasi osa suurenemisel on andmebaasiobjektide loomise vastutuse üleminek rakenduselt andmebaasis loodud trigeritele. See muudab süsteemi komponentide rollide jaotust loogilisemaks: andmebaasisüsteem ja selle abil realiseeritud andmebaasiobjektid tegelevad andmetega ning andmestruktuuride genereerimisega ning rakenduse ülesandeks on kasutajatele funktsionaalsuse pakkumine. See vastab huvide eristamise (*Separation of Concerns*) disainiprintsiibile (Trosin, 2009). Kahjuks ei pääsenud ka sellise lahenduse puhul dünaamilisest SQL lausete loomisest, kuid selle osa vähenes võrreldes olemasoleva rakendusega. Samas pole see lahendus kooskõlas kontsentreeritud vastutuse printsiibiga (*Single Responsibility Principle*) (Trosin, 2009), sest andmebaasi osas tegeletakse nüüd nii andmete talletamisega kui ka andmemuudatustest tulenevalt süsteemi struktuuri ja käitumise muutmisega. Samas rõhutab ka Trosin (2009), et erinevate disainiprintsiipide rakendamisega ei maksa minna ekstreemseks ning tuleb leida konkreetse olukorra jaoks sobivaim lahendus.

WebMeta4 süsteemis paigutatakse modelleerimiskeelte metaandmed skeemi *public* asemel eraldi skeemi nimega *mmm* (nagu meta-metamudel). Trigerid reageerivad *mmm* skeemis olevates tabelites toimuvatele andmemuudatustele. Nende käivitamise tulemusena luuakse keeltele vastavad andmebaasiskeemid, keelte mudelielementidele vastavad tabelid ning mudelielementide omadustele vastavad veerud. Eraldi skeemi kasutamine tagab keelte metaandmete parema eraldatuse muudest andmebaasis olevatest andmetest ja kasutajatest. Varasemalt kasutuses olnud *public* skeem on vaikimisi kättesaadav kõigile andmebaasile juurdepääsu omavatele kasutajatele. Uue lähenemise täiendavaks plussiks on andmebaasiobjektide dünaamiline loomine (meta)andmete muudatuste tulemusena, mis jätab rakenduse tasemele üsna vabad käed ning võimaldab kasutada ORM (Object-Relational Mapping – objekt-relatsiooniline vahendamine) lahendusi. ORM võimaldab automaagiliselt teisendada andmeobjektid rakenduse objektideks ja vastupidi. (Wikipedia, 2014)

Tänu ORM toele oli võimalik võtta kasutusele ka raamistik, mis toetaks kiiremat arendamist ning vähendaks ülemäärase koodi kirjutamist. Raamistikuks sai valitud PHP raamistik Yii. See võimaldas lisaks eespoolmainitud boonusele kasutada rakenduse poolt pakutavat kasutajate õiguste haldust, kus lisaks andmebaasipõhisele õiguste süsteemile oli toetatud rakenduste sisesed õiguste ja rollide jagamised. Lisaks pakub see MVC (*model-view-controller* ehk mudel-vaade-

kontroller) mustri kasutamise võimalust, mis viib paremini struktureeritud koodini ning lahutab andmed ning kasutajaliidese visuaalse poole. (Tutorialspoint, 2014) Samuti võimaldas raamistik lihtsa vaevaga kasutusele võtta ORM toe ning vähendas sellega oluliselt arendaja poolt kirjutatava koodi hulka. Samuti pakub Yii raamistik võimalust andmebaasi objektide pealt genereerida valmis vormid ning objektid, tänu millele jääb arendaja ülesandeks genereeritud objektide ja vaadete kohandamine. Negatiivse poolena võib raamistiku puhul välja tuua täiendava õppimise vajaduse ning võimaliku kogenematusest tulevate mitte kõige optimaalsemate lahenduste kasutamise, kuid selle kaalub üles muu raamistikust saadav kasu. Samuti kord raamistiku endale selgeks teinuna, on uute funktsionaalsuste lisamine kiirem.

Evolutsioneerimise käigus lisati süsteemi võimalus kirjeldada metamudelite loomisel objektide vahelisi üldistusseoseid ning defineerida loendtüüpe, mida saab hakata kasutama vastavas keeles mudelite loomisel. Samuti on võimalik registreerida iga metamudeli korral selle hetkeseisund.

4. Kasutatavad mudelid, mustrid ja tehnoloogiad

WebMeta4 rakenduse loomiseks valisin PHP (PHP.net, 2014) programmeerimiskeele versiooniga 5.4 ning PostgreSQL (PostgreSQL, 2014) andmebaasisüsteem versiooniga 9.4. Ka vana süsteemi versioon kasutas PostgreSQLi.

Järgnevalt on välja toodud WebMeta4 loomise juures kasutatavad tehnoloogiad ning nende valiku põhjused koos valitud lähenemise plusside ja miinustega.

4.1 Raamistik

Tarkvaraarenduse raamistik ehk rakenduskarkass on universaalne taaskasutatav platvorm rakenduste loomiseks, pakkudes parimate praktikate kogumina tuge arendajale. Raamistik peidab endasse suure hulga funktsionaalsust, mida antud programmeerimiskeeles on enamuste rakenduste loomiseks vaja, vähendades koodi hulka, mida arendajad peavad soovitud tulemuse saavutamiseks looma.

Raamistikuks sai valitud PHP raamistik Yii versiooniga 1.1, mis on lühend sõnadest Yes It Is (tõlkes „Jah see on“). Põhilisteks valiku kriteeriumiteks said pikk nimekiri toetatud funktsionaalsust ning väike õppimiskurv, et raamistikku kasutusele võtta. (Yiiframework, 2014)

Kuna käsitletav rakendus on loodud PHPs, siis käsitlen PHP raamistiku omaseid jooni. Raamistiku kasutamisel on mitmeid eeliseid, mis hetkel välja toodud valitud Yii raamistiku näol, kuid mis on väikeste variatsioonidega ka teiste levinumate PHP raamistike arsenalis:

1) Arendajat suunatakse kasutama objektorienteeritud lähenemist, mis soodustab rakenduse paremat disaini. Selle tulemusena luuakse kindlaid funktsionaalsusi koondavad klassid, mis lihtsustab tulevikus rakendusest aru saamist ning täiendamist.

2) Sisse ehitatud MVC tarkvara-arenduse arhitektuurimustri tugi, millega eraldatakse andmete objektid, andmete pealt loodav visuaalne pool (kasutajaliides) ning neid kahte siduvad kontrollerid.

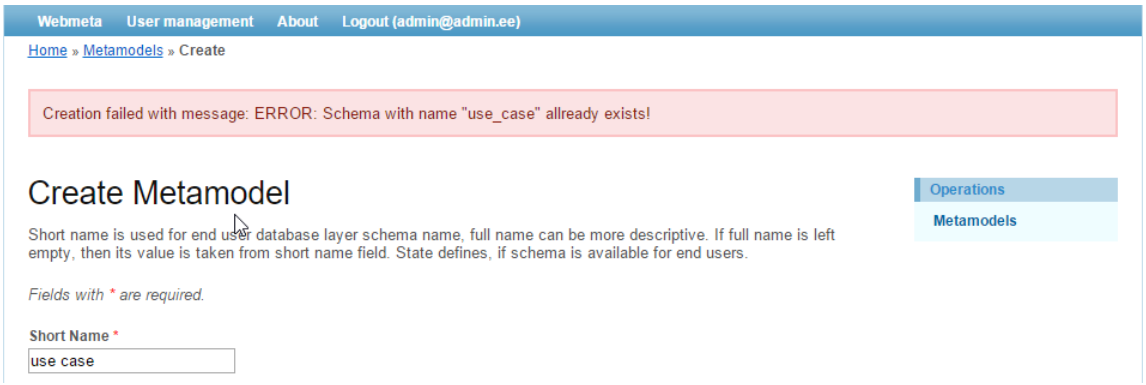
3) ORM ehk Object Relational Mappingu tugi, mis tähendab, et seotakse ära SQL-andmebaasi tabelid-veerud ja PHP objektid ning luuakse CRUD tegevuste tugi.

Põhitegevusteks on andmete loomine (Create), lugemine (Read), uuendamine (Update) ja kustutamine (Delete).

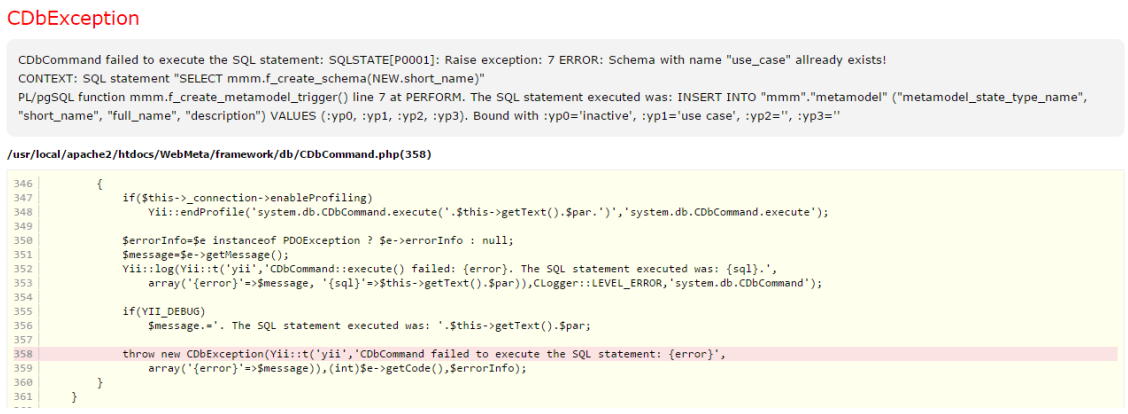
4) Sisse ehitatud kasutajate ja õiguste haldus lehtede ning meetodite kaupa. Näiteks saab defineerida, et ühtede rollidega kasutajatel on õigus lehti vaadata, teiste rollidega kasutajatel on õigusi ka lehel muudatusi teha.

5) Sisse ehitatud enamkasutatavad validaatorid, et kontrollida kasutajate sisestusi ning lihtsalt defineerida objekti andmete reegleid.

6) Parem logimine ning vigade haldus võrreldes tavalise rakendusserverist tuleva infoga. Samuti on võimalus vigade kuvamist lõppkasutajale piirata ühe parameetri muutmisega, et vältida liigse info näitamist lõppkasutajale vea korral, mis võiks kujutada endast turvalisuse riski (Vt Joonis 5). Samas arendamise käigus kuvatakse põhjalik vea kirjeldus koos eelnevate välja kutsutud meetodite ning kasutatavate sisenditega (Vt Joonis 6).

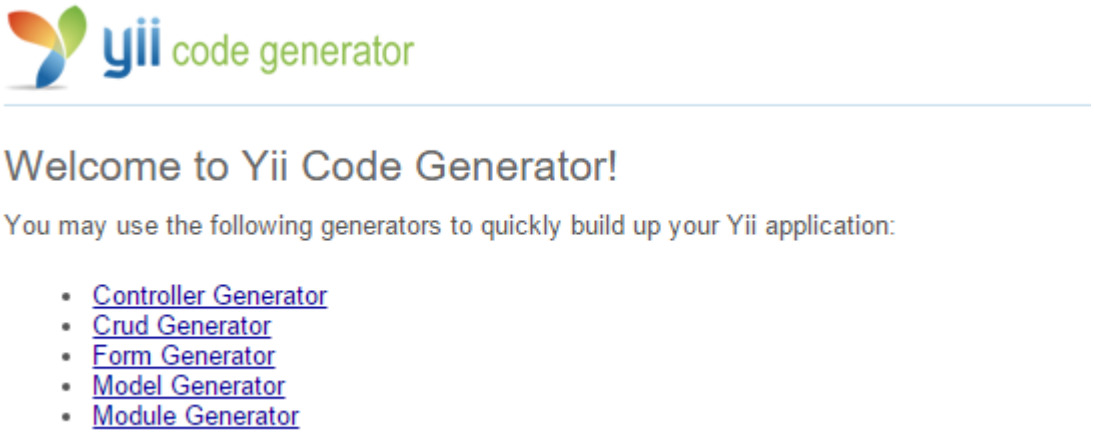


Joonis 5: Veateade mitte-arendusvaates



Joonis 6: Veateade arendusvaates

7) Korduva koodi genereerimise tugi raamistiku poolt. Ette antud andmebaasi pealt genereeritakse vajalikud objektid, vaated ning objektidevahelised seosed ning funktsionaalsus CRUD tegevuste jaoks. (Vt Joonis 7)



Joonis 7: Yii koodigeneraator

8) Lai funktsioonide tugi deklaratiivselt kirjeldada enamkasutatavad veebilehe komponendid, näiteks vormid või menüüd. Genereeritud objektide kujundus ühtib juba olemasoleva rakenduse disainiga. Samas lihtne on sisse viia ka kohandusi või vajadusel ikkagi vajalikke komponente ise teha.

Vaatamata paljudele kasulikele funktsionaalsustele on raamistiku puhul ka paar kitsaskohta. Neist esimesena tooks välja suure täiendava koodi mahu, mis tuleb rakendusega alati kaasa panna, hoolimata sellest, et tõenäoliselt ei kasutata kogu pakutud funktsionaalsust. Kuna pakutud funktsionaalsuse hulk on suur, siis tekitab see ka täiendavat ressursikulu rakenduse käitamisel. Yii raamistiku puhul vähendatakse viimati mainitud nähtust kasutades hilist laadimist (*lazy loading*), mille puhul luuakse-päritakse objektid alles siis, kuid neid vaja läheb.

Kuna ka raamistike puhul toimub tarkvara evolutsioon, siis tähendab raamistiku kasutamine selle ajakohastamise ning uuendamise vajalikkust, et kasutada rakendusserverite pakutavaid uusi funktsionaalsusi. Samuti on uuendused olulised, et parandada leitud turvalisuse vigu ning vahepeal kasutusele võetud optimaalsemaid lahendusi.

Alternatiiv raamistiku kasutamisele oleks eespoolmainitud funktsionaalsuse omapoolne loomine, mis annaks küll parema ülevaate ning kontrolli tarkvaras toimuvate protsesside üle, kuid

tähendaks märkimisväärset ajakulu ning eeldaks häid teadmisi erinevate oluliste komponentide parimate praktikate ning tehnikate kohta.

4.2 Trigeripõhine lähenemine

WebMeta jaoks vajaliku funktsionaalsuse loomiseks on võimalik kasutada erinevaid lähenemisi. Üks võimalus on kogu äriloogika hoidmine rakenduses ning anda rakendusserveri tasemel korraldused luua lõppkasutajatele suunatud keeltele vastavad skeemid ja tabelid. Seda varianti kasutas ka eelmine WebMeta versioon. Selle lähenemise eeliseks on õhuke andmebaasi kiht, kuid sellega ka positiivne pool piirdub. Negatiivse külje pealt võib välja tuua olulise koodi keerukuse tõusu, seal hulgas rakenduses sisalduvate SQL lausete ning transaktsioonide haldamine, et ei tekiks poolikuid muudatusi.

Teine võimalus oleks luua keelte kirjelduste halduseks andmebaasi vaated ja andmebaasiserveris talletatud rutiinid (funktsioonid ja protseduurid), mille poole rakenduse koodis õiges kohas pöörduda. Vaated ja rutiinid moodustaksid andmebaasi avaliku liidese, mis võimaldaks andmebaasi kasutajate (rakenduste) eest kapseldada. Selle lähenemise miinusena võib välja tuua asjaolu, et rutiinide ja vaadete muutmise puhul on vaja üle vaadata nende kõik väljakutsed ning seoses sellega on juures ka üks täiendav võimalik vea tekkimise koht. Selle lahenduse positiivseteks külgedeks on näiteks võimalus peita teatavaid andmebaasi struktuuri muudatusi avaliku liidese taha (nii, et andmebaasi struktuuri muutudes ei pea hakkama liidese kasutajaid ümberprogrammeerima) ning võimalus realiseerida liidese abil üks kiht mitmetasemelisest turvalisuse tagamise süsteemist (kasutajatel on võimalus kasutada liidese elemente, aga mitte pöörduda otse tabelite poole).

Kahe eelneva variandi puhul peavad kas rakendus või funktsioonid tegelema baastabelite poole pöördumise ja nendes info haldamisega.

WebMeta uue versiooni jaoks sai valitud andmebaasi trigerite kasutamine. Triger on andmebaasis salvestatud programm, mis käivitub mingi sündmuse (enamasti andmemuudatus) peale juhul kui on täidetud käivitamiseks vajalikud eeltingimused (eeltingimuste määrane trigeri spetsifikatsioonis pole kohustuslik). Deklaratiivsete kitsenduste kõrval on trigerid teine põhiline viis aktiivse andmebaasi realiseerimiseks. Aktiivses andmebaasis reageerib andmebaasisüsteem andmebaasis toimuvatele sündmustele, mitte ei vaata neid vaikides ja leplikult pealt.

Trigerid hoolitsevad mudelite hoidmiseks mõeldud andmestruktuuride loomise ja muutmise eest, käivitudes administraatori poolt keele metaandmetes tehtud muudatuste tulemusena. Kõik loodavad trigerid on reataseme trigerid. See tähendab, et need trigerid käivituvad üks kord iga metaandmete tabelis lisatud/muudetud/kustutatud rea kohta.

Trigerite kasutamine ei välista andmebaasi avaliku liidese loomist, kuid antud töös seda ei tehta. Järgnevalt on välja toodud trigerite lähenemise positiivsed ja negatiivsed küljed.

4.2.1 Trigerite plussid

Antud ülesandepüstituse puhul pean kõige olulisemaks trigerite kasutamise plussiks metamodelleerimise meta-taseme andmete (metaandmete) ning andmetest uute andmebaasiobjektide loomise loogika lähedust. See lähenemine tagab meta-andmete vastavuse genereeritud andmebaasi objektidele ka siis, kui keegi peaks andmebaasi tabelites käsitsi muudatusi tegema.

Sellest tuleneb ka trigerite teine oluline pluss, nimelt andmete ning terviklikkuse kontroll on andmebaasi andmete lähedal ning andmete muutmisel kontrollitakse, kas sellest tulenev uus olukord on reeglitega kooskõlas või mitte. Trigerite puhul on tagatud ka muudatuste atomaarsus. See tähendab, et muudatus meta-andmetes jõuab andmebaasi skeemi täielikult või keeratakse see muudatus tagasi ning tekitatakse erand. Kui andmebaasiobjektide genereerimise loogika eest hoolitseks rakendus, siis peavad arendajad ka programmeerima tehtud muudatuste tagasivõtmise, mis keerulisemate protsesside puhul on väga veaaldis.

Trigeripõhine kasutamine parandab süsteemi turvalisust, sest muudatusi ning skeemide ja tabelite genereerimist saab esile kutsuda ainult kasutaja, kellel on õigus teha muudatusi metaandmetes (*mmm* skeemis olevates tabelites). Praegu saab kasutaja tegevust piirata skeemide ja tabelite tasemel (kasutades piirangute jõustamise hõlbustamiseks rolle), siis alates PostgreSQL versioonist 9.5 on oodata ka reapõhiste õiguste haldamise lisandumist (PostgreSQL, 2015). See tähendaks WebMeta rakenduse kontekstis näiteks andmebaasi õiguste tasemel piirangut, teha muudatusi ainult kasutaja enda loodud metamudeli objektide kirjeldustes (näiteks ainult teatud ridades, mis on tabelis *mmm.metamodel_object*).

Kõikehõlmav Andmebaasi Turvalisuse Mudel (Downs, 2009) koos trigerite abil tegevuste logimisega annab hea ülevaade, milline lõppkasutaja on milliseid muudatusi esile kutsunud.

PostgreSQL trigeri funktsioonis saan kasutada funktsiooni SESSION_USER, mille väärtuseks on antud juhul sessiooni algataja kasutajanimi, mis vastab lõppkasutajale.

Kui andmebaasiobjektide loomise loogika oleks realiseeritud rakenduses, siis oleks muudatuste logimiseks vaja rakendusest saata täiendavaid väljakutseid andmebaasi poolele koos tegevuse algatanud osapoolte infoga. See tähendaks täiendavat andmete liikumist rakenduse ja andmebaasiserveri vahel, millest antud kasutuse puhul saab loobuda. Kasutatud lahendus tagab ka andmete parema kvaliteedi, sest väheneb oht, et ühes või teises kohas unustatakse tegevuse tegija andmed kaasa anda.

Heaks näiteks trigeripõhise ärireeglite ning andmete terviklikkuse reeglite jõustamise kohta on Oracle RuleGen andmebaasi raamistik. Need reeglid jõustavad andmebaasi tasemel piiranguid registreeritavatele andmetele. RuleGen võimaldab kasutajal defineerida vajalikud jõustamist vajavad piirangud ning raamistik genereerib piirangute jõustamiseks mõeldud protseduurse koodi. (RuleGen, 2015)

Trigerite kasutamine võimaldab teha äriloogikas muudatusi inimestel, kellel pole juurdepääsu rakenduse lähtekoodile või kellel pole piisavaid oskuseid rakendust muuta.

4.2.2 Trigerite miinused

Kahjuks pole trigerite kasutamine ilma negatiivse pooleta. Tänu sellele on palju arutatud, millal, kas ja kuidas on trigerite kasutamine mõistlik. Järgnevalt on välja toodud argumendid, mis räägivad trigerite kahjuks.

Trigerite kasutamine seob rakenduse tugevalt valitud andmebaasisüsteemiga, mis oluliselt lisab keerukust, kui on soov vahetada andmebaasisüsteemi, sest siis tuleb nende protseduurne kood ümber kirjutada. Kuigi paljud andmebaasisüsteemid toetavad trigereid ja nende kirjutamiseks mõeldud protseduurseid keeli, on paraku erinevates süsteemides kasutuses erinevad keeled.

Olulisi põhjuseid on välja toonud Rob van Wijk oma blogi artiklis „Database triggers are evil“ (Wijk 2007), kus ta toob välja järgmised murekohad. Esiteks on probleemiks trigerite käivitumisel toimuva automaagiline tegevus, mille toimumisest kasutajal ei pruugi aimugi olla. Samuti vähendab see rakenduse arendamisel intuitiivsust ning raskendab hilisemat veaotsingut, kui midagi on valesti läinud.

Automaagiat saab vähendada korraliku dokumentatsiooniga ning korraliku logimisega. Näiteks PostgreSQL *auto_explain* moodul pakub alates PostgreSQL 9.4 võimaluse koos lausete täitmisplaanide logimisega logida infot ka nende lausete käivitanud trigerite kohta, ilma rakenduse arendajate poolt täiendava koodi kirjutamiseta. (Itagaki, 2014). Vea situatsioonides pakub PostgreSQL väga head vea konteksti infot, kus on näha, milliste meetodite väljakutse tulemusena ning milliste andmetega jõuti veasistuatsioonini.

Automaagilisuse probleem võib tõenäoliselt rohkem välja lüüa suuremate ja keerulisemate süsteemide puhul, kus on paljude tabelite koosmõjul võib andmebaasi poole peal ootamatusi juhtuda. WebMeta projekti selle koha pealt liiga keeruliseks ei liigitaks, et poleks võimalik jägida andmetega toimuvaid sündmuseid. See oleneb ka kuidas tarkvara parasjagu arendatakse. Kui on eraldi andmebaasi ning kliendi süsteemi arendajad, siis tuleks võtta trigereid kui liidest, kus on kokku lepitud sündmused, mis võivad toimuda pärast andmete saatmist andmebaasi poolele. Kui kogu rakenduse arendusprotsessiga tegeleb sama meeskond, siis on oluline hea dokumentatsioon, ning oluline teadmine, et vaja arvestada ka andmebaasi poolel toimuva loogikaga. See võib rohkem olla probleemiks uute arendajate lisandumisel meeskonda. Käesolevas töös viidi kogu WebMeta uue versiooni programmeerimine nii rakenduse kui andmebaasi poole pealt läbi ühe ja sama inimese poolt.

Teine oluline aspekt on trigerite kirjutamise oskus, sest enamjaolt kiputakse unustama, et on vaja arvestada mitmete kasutajate sessioonidega ning see vajab trigerite koodis korrektset andmeelementide lukustamist. Teisalt täpselt samade asjadega tuleb arvestada ka rakenduse koodi kirjutades ning probleemiks on pigem see, et rakenduse arendajad ei tea andmebaasi tehnoloogia kõiki nüansse. Sellest vaatepunktist lähtudes olekski parem, kui andmetega tehtavate operatsioonide kirjutamine jääks nende õlule, kes seda paremini oskavad.

Lisaks võib siin välja tuua võlts-kindlustunde, mida halvasti kirjutatud trigerid annavad. Arendajad mõtlevad, et trigerid reageerivad korrektselt kõigile sündmustele, kuid tegelikkuses leidub andmebaasi kasutamise stsenaariume, mille tulemusena jõuavad trigeritest hoolimata andmebaasi ebakorrektsed andmed. Samas tuleb ka öelda, et alates Postgres 9.1 on SERIALIZABLE isolatsioonitase (kõige kõvem võimalik transaktsioonide isolatsioonitase) realiseeritud *Serializable Snapshot Isolation (*SSI, SerializableSI) algoritmi põhjal. Kui kõik andmebaasis toimuvad transaktsioonid kasutaksid (SSI-põhist) SERIALIZABLE isolatsioonitaset, siis ei oleks vaja ühegi ärireegli jõustamiseks kasutada tabeli või selle üksikute

ridade ilmutatud kujul lukustamist. See muudab lihtsamaks nii trigerite kui ka andmebaasi kasutatavate rakenduste kirjutamise. Käesolevas töös eeldatakse, et nii kõrget isolatsioonitaset ei kasutata ning transaktsioonide vahel on READ COMMITTED isolatsioonitase, mis on ühtlasi PostgreSQL vaikimisi isolatsioonitase.

Kolmanda punktina toob van Wijk välja andmemuudatuste aeglustumise võrreldes ilma trigeriteta andmebaasiga. (Wijk, 2007) Samas WebMeta4 lahenduse puhul ei mängi see olulist rolli, sest administreerimise poole pealt pole väga suurt andmebaasi kasutamise koormust ette näha ning ka andmemahud pole suured (suuremate tabelite korral räägime maksimaalselt tuhandetest ridadest).

Dünaamiliste käskude jaoks on trigerite poolt välja kutsutud funktsioonides vaja kasutada EXECUTE lauseid, mis kujutavad teatavat turvariski ning vajavad kirjutamisel täiendavat hoolt. Funktsioonide ette antud argumente tuleb töödelda *quote_ident* funktsiooniga, mis hoolitseb, et ei tekiks SQL süstimise (*SQL injection*) ründe ohtu.

Ühe täiendava olulise mõttekohana tooksin välja teatava keerukuse andmebaaside versioneerimises, sest objektide hulk, mille versioone jälgida suureneb. Tekib vajadus täiendavate protsesside ning vahendite järgi, et ei kirjutataks üle teineteise muudatusi ja andmebaasi uuendamise protsess annaks korrektse tulemuse. See võib osutuda probleemiks suuremate arendusmeeskondade puhul. Praeguses arendusfaasis ei pidanud ma täiendavat andmebaasi loogika versioneerimist vajalikuks.

Teine tähelepanek WebMeta arendamisest on, et trigeripõhise süsteemi puhul on kord vigaselt andmebaasi sattunud andmeid trigerite tõttu keeruline eemaldada. Selle puhul oli abiks trigerite välja lülitamine enne kustutamisi. See probleemi lahendamine võib suuremate meeskondade puhul olla probleemiks, kuid sellisel juhul võib abi olla täiendavate andmebaasi funktsioonide loomisest, mis võimaldavad probleemseid andmeid eemaldada.

4.2.3 Trigerite modelleerimine

Andmebaasi trigerite modelleerimisel kasutatakse lihtsamatel juhtudel sageli jada ning voo diagramme, kuid WebMeta4 andmebaasitrigerite kirjeldamisel tundus kõige mõistlikum kasutada trigerite-sündmuste vastavustabelit. Seda põhjusel, et keerukamate trigerite tingimuste ning tegevuste puhul ei mahuks kirjeldused piisavalt hästi diagrammidele ning tabel on ülevaatlikum.

4.2.4 Trigerite testimine

Sarnaselt tarkvara rakendustele vajavad testimist ka trigerid, et tagada nende ootuspõhine toimimine ja hilisemate muudatuste puhul tööle jäämine. Kõige lihtsamalt vaadates tuleb testida trigeritega kaetud andmebaasitabelite põhjal lisamise, muutmise ja kustutamise lauseid ning kontrollida, kas saavutatakse oodatud tulemus. Kindlasti vajavad testimist ka olukorrad, kus on vajalik vea tekkimine. Vajalik on kontrollida, kas trigerid annavad õige tulemuse samaaegsete andmemuudatuste korral trigeritega kaetud tabelites.

Selliseid teste võib kirjutada eraldi paketti ning vajadusel läbi jooksutada. Nagu php rakenduste puhul leidub arendamist ja testimist toetavaid raamistike, nii on loodud ka andmebaasi rutiinide testimiseks oma raamistikke. PostgreSQL puhul jäi kõige põhjalikuma funktsionaalsusega silma PgTap. PgTAP on PL/SQLis ja PL/pgSQLis kirjutatud PostgreSQL andmebaasi laiendus, mis pakub välja valiku lihtsustavaid meetodeid, et kontrollida koodi korrektset toimimist ning oodatud tulemuste kehtimist. Lisaks testimisele toetab pgTAP tulemuste väljastamist TAP (Test Anything Protocol) formaadis, mis lihtsustab selle tööriista sidumist muude testimisvahenditega. (Wheeler, 2014)

WebMeta4 arendamise käigus sai testimise funktsionaalsus üles seatud ning ka mõned testid kirjutatud. Kahjuks terve rakenduse testidega katmine töö valmimise raames polnud võimalik, sest PgTap tööle saamine Windows operatsioonisüsteemiga keskkonnas oli üsna vaevarikas ning ajakohaseid juhendeid nappis. Samas pärast tööle saamist oli tegemist hea funktsionaalsuse ning intuitiivse vahendiga, et kontrollida andmebaasi töö korrektsust. PgTap testimise näide on toodud näite peatükis.

4.3 Andmebaasiobjektide nimetamine

Esmapilgul ei pruugi rakendusi arendades tunduda oluline mõelda nimetuste peale, olgu selleks andmebaasiobjektid, klassid, paketid, rollid, vms. Ometi annab see olulise panuse hilisemas rakenduse koodi mõistmises. Fowler (2009) on öelnud, et arvutiteaduses on kaks rasket asja, millest üks on asjade nimetamine.

Järgnevalt on välja toodud, mida tasuks tarkvaraarenduses andmebaasiobjektide nimetamisel arvesse võtta ning mida sai ka WebMeta4 arendamise käigus arvestatud ja kasutatud. Esiteks nimed peaksid olema hästi kirjeldavad ning soovitatavalt mitte lühendid. Nime näiteks on

keele_nimi. Tühiku rolli täidab alakriips. Kindlasti ei saa kasutada andmebaasisüsteemides kasutatava andmebaasikeele võtmesõnu (nt SELECT, TABLE). Keele nimi ei saa olla *public*, *information_schema*, *pg_catalog* ja *pg_toast_temp_1*, sest sellise nimega skeemid luuakse igas PostgreSQL andmebaasis automaatselt.

Andmebaasiobjektide nimetamisel on soovitatav kasutada nimisõna ainsuse vormi. Sellisel juhul ei pea muretsema käänamise pärast ning nimi ei seostu tabelist oodatavate ridade arvuga. (Sarkuni, 2014)

Primaarvõtmete veergude puhul on kaks koolkonda. Ühed soovitavad ühest veerust koosneva primaarvõtme korral kasutada ainult nime *id*, sest see on lühem võrreldes teise eelistusega. Teisel puhul soovitatakse kasutada kuju *<tabeli_nimi>_id*. Selle lähenemise eeliseks on võimalus kirjutada andmebaasi *join* päringud lühemalt kasutades tingimuseks *USING (t2_id)*, selle asemel, et kirjutada *ON (t1.t2_id = t2.id)*. Rakenduse SQL lausetes kasutasin kuju *<tabeli_nimi>_id*. Välisvõtme veeru puhul on standardiks kasutada nimetust *<viidatava_tabeli_nimi>_id*, nagu ka eelnevas näites sai kasutatud.

Piirangute (*constraint*) nimetamisel on tava jätta nimesse ka piirangut tähistav lühend (vt Tabel 1). (Hazlewood, 2014)

Tabel 1: Nimetustes kasutavad piirangute prefiksid

Piirangu tüüp	**prefiks**
primary key	pk_
foreign key	fk_
check	ck_
not null	nn_
unique	uq_
index	idx_

Rutiinide puhul on hea tava tähistada sisendparameetrid prefiksiga *in_* ning väljundparameetrid prefiksiga *out_* ning kohalikke defineeritud muutujaid alakriipsuga prefiksiks, et koodis oleks lihtne aru saada, millise muutujaga parasjagu tegu.

Kuna rakendust luues tuli palju kokku puutuda dünaamiliste nimede kokkupanemisega, siis lõin selle jaoks eraldi andmebaasi funktsioonid, et kõikjal oleks samamoodi genereeritud välisvõtmete või piirangute nimed. See lihtsustab tulevast nimede loogika muutmist ning annab tulevastele arendajatele ühtse koha selle loogika järgi vaatamiseks ning vajadusel muutmiseks. Ühtlasi on see kooskõlas huvide eristamise ning kontsentreeritud vastutuse printsiipidega.

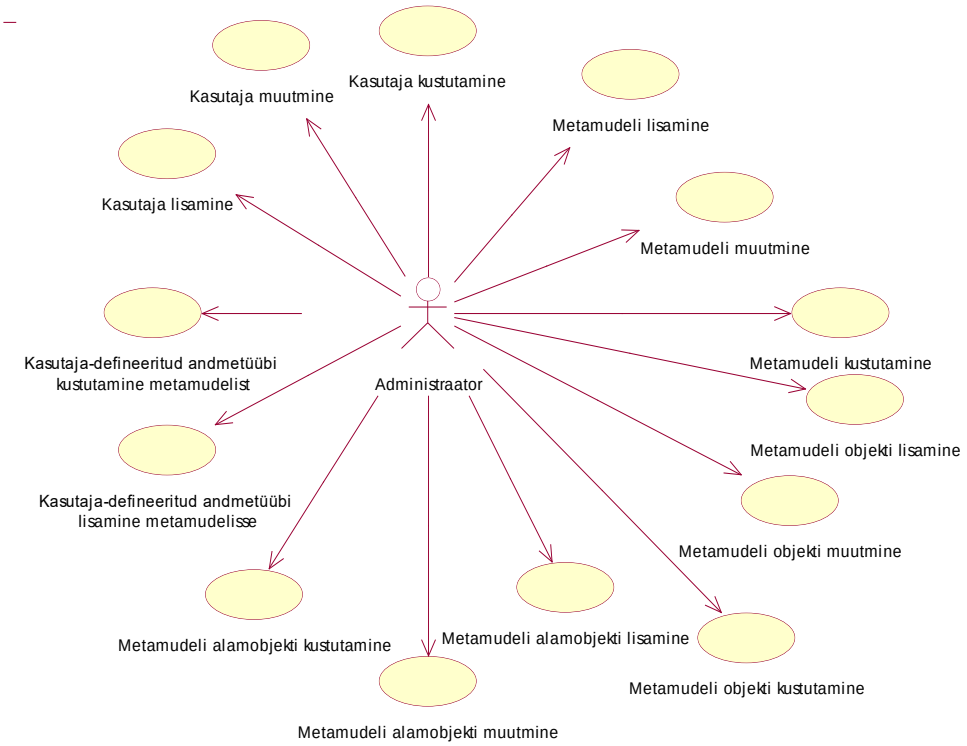
Dünaamiliste andmebaasiobjektide nimede puhul võib üsna kiirelt vastu tulla PostgreSQL poolt vaikimisi piiranguks seatud 63 baidine nime pikkuse piirang. Selle töö raames eraldi sellele probleemile lahendust ei otsita ning jääb WebMeta järgmiste iteratsioonide parandada.

5. WebMeta uue versiooni arendamine

Webmeta uue versiooni puhul on valitud trigeritel põhinev lähenemine, kus keelte metaandmeid sisaldavate tabelite pealt genereeritakse andmebaasi trigerite abil modelleerimisvahendi kasutajatele mudelite hoidmiseks vajalikud tabelid ning õiguste jagamiseks vajalikud rollid ning õigused. Iga modelleerimisvahendi metaandmete hulka kuuluvad ka andmed selle kohta, millised kasutajad saavad seda vahendit kasutada. Administraatori poolt uue kasutaja loomine tähendab andmebaasis trigerite abil automaatselt uue kasutaja loomist ning sellele sobivate rollide määramise kaudu õiguste jagamist. Trigeripõhine lähenemine on detailsemalt kirjeldatud eelmises peatükis.

5.1 Kasutusjuhtude kirjeldused

Joonis 8 kujutab kasutusjuhtude diagrammi, kust võib näha administraatori kohustusi süsteemis.



Joonis 8: Adminstraatori kasutusjuhtude diagramm

Järgnevalt on need kasutusjuhud kirjutatud lahti kõrgtaseme formaadis keskendudes administraatori eesmärkidele selle süsteemi kasutamise kontekstis. Administraatori poolt käivitatud sündmustele vastavad trigerite sündmused on kirjeldatud trigerite-sündmuste vastavustabelis.

1) **Kasutaja lisamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator sisestab kasutaja andmed, valib tüübi, staatuse ning soovi korral ka seob saadaval olevate metamudelitega.

2) **Kasutaja muutmine**

Tüüp: Sekundaarne
Tegutsejad: Administraator
Kirjeldus: Administraator muudab kasutaja parooli, staatust või seoseid metamudelitega.

3) **Kasutaja kustutamine**

Tüüp: Sekundaarne
Tegutsejad: Administraator
Kirjeldus: Administraator kustutab kasutaja.

4) **Metamudeli lisamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator sisestab vajalikud metamudeli andmed ning salvestab.

5) **Metamudeli muutmine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator uuendab metamudeli andmeid ning salvestab.

6) **Metamudeli kustutamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator kustutab metamudeli.

7) **Metamudeli objekti lisamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator sisestab metamudeli objekti loomiseks vajalikud andmed.

8) **Metamudeli objekti muutmine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator muudab objekti andmeid või objekti seoseid teiste objektidega.

9) **Metamudeli objekti kustutamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator kustutab metamudeli objekti.

10) **Metamudeli objekti alamobjekti lisamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator lisab metamudeli objekti alamobjekti.

11) **Metamudeli objekti alamobjekti muutmine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator uuendab metamudeli objekti alamobjekti.

12) **Metamudeli objekti alamobjekti kustutamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator kustutab metamudeli objekti alamobjekti.

13) **Metamudeli objekti seose loomine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator valib metamudeli objektid ning nendevahelised seose ning salvestab

14) **Metamudeli objekti seose kustutamine**

Tüüp: Primaarne
Tegutsejad: Administraator
Kirjeldus: Administraator kustutab metamudeli objektide vahelise seose.

15) **Kasutaja defineeritud andmetüübi lisamine metamudelisse**

Tüüp: Sekundaarne
Tegutsejad: Administraator
Kirjeldus: Administraator defineerib metamudelile kuuluva andmetüübi, mida saab määrata metamudeli objekti alamobjekti tüübiks.

16) **Kasutaja defineeritud andmetüübi kustutamine metamudelist**

Tüüp: Primaarne
Tegutsejad: Administraator

Kirjeldus: Administraator kustutab metamudeli juurest andmetüübi.

17) **Kasutaja defineeritud andmetüübi väärtuse lisamine kasutaja defineeritud andmetüübile**

Tüüp: Sekundaarne
Tegutsejad: Administraator
Kirjeldus: Administraator sisestab kasutaja defineeritud andmetüübile väärtuse.

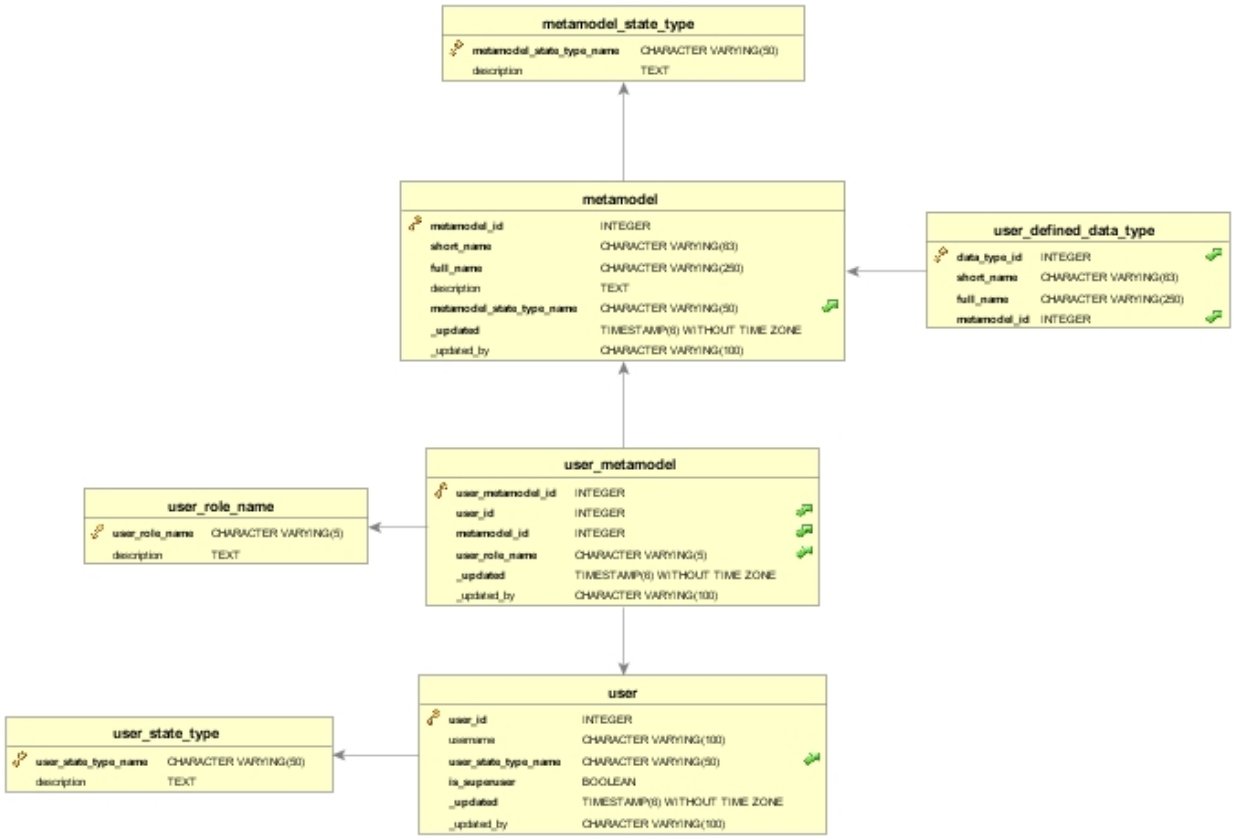
1) **Kasutaja defineeritud andmetüübi väärtuse kustutamine kasutaja defineeritud andmetüübist**

Tüüp: Sekundaarne
Tegutsejad: Administraator
Kirjeldus: Administraator kustutab kasutaja defineeritud andmetüübi väärtuse.

5.2 Andmebaas

Rakenduse andmebaasisüsteemina on kasutusele võetud PostgreSQL versiooniga 9.4 ehk kõige uuem rakenduse loomise ajal saada olev versioon. Kuna andmebaasisüsteemidesse lisandub uute versioonidega uusi erisusi, kuid väga harva mõni erisus eemaldatakse, siis see annab kindlust, et süsteem funktsioneerib ka uuemate versioonide korral. Valik langes PostgreSQL peale, kuna varasem versioon kasutas samuti seda. Tegemist on tasuta ja avatud lähtekoodiga andmebaasisüsteemiga, mis pakutavate võimaluste hulgalt ei jää alla parimatele kommertssüsteemidele, toetab suhteliselt suures mahus SQL standardit ning pakub arendajatele erinevaid võimalusi süsteemi laiendada. Andmebaasisüsteemide populaarsuse indeksis on see 2015. aasta detsembri seisuga viiendal kohal (populaarsuselt neljas SQL andmebaasisüsteem) (DB-Engines Ranking 2015).

Joonis 9 kujutab kasutajate ning metamudelite andmete seoseid andmebaasis. Siin on näha, et kasutaja kohta hoitakse hetkel ainult kasutajanime ja staatust (tabel *user*), kuid mitte soolatud ja räsitud parooli. See on võimalik tänu parooli hoidmisele PostgreSQL andmebaasi sisemisel tasemel, sest igale sisselogimiseks mõeldud kasutajakontole vastab eraldi andmebaasi kasutaja. Kasutaja võimalikud olekud on defineeritud tabelis *user_state_type.* Hetkel on kaks võimalikku staatust: aktiivne ja mitteaktiivne. Iga kasutaja võib olla seotud ühe või mitme metamudeliga ühes või mitmes rollis. Hetkel on rollide valikus ainult administraatori ja lõppkasutaja roll. Rolli nime ja kirjeldust hoitakse eraldi tabelis *user_role_name,* et oleks hiljem mugav lisada rolle ning muuta kasutajate rollide loogikat, lubades ühele kasutajale mitme rolli omistamist. Metamudeli nähtavus lõppkasutajatele sõltub metamudeli olekust (tabel *metamodel_state_type*).



Joonis 9: WebMeta4 administraatori andmebaasiskeem – metamudeli ja kasutaja seosed

Joonis 10 kujutab tabeleid, kus on andmed metamudelite objektide, alamobjektide ning nendevaheliste seoste kohta. Igal metamudelil on null või rohkem objekti (tabel *metamodel_object*), millel omakorda on null või rohkem alamobjekti (tabel *metamodel_subobject*). Igal alamobjektil peab olema üks kindel andmetüüp (tabel *data_type*). Selleks võib olla süsteemi defineeritud andmetüüp (tabel *system_defined_data_type*), milleks on PostgreSQL poolt toetatud andmetüüpide valik. Teise variandina võib alamobjekti tüübiks olla kasutaja poolt metamudelile defineeritud tüüp, mis on nähtav ainult kindla metamudeli raames (tabel *user_defined_data_type*). Kasutaja-defineeritud tüüp on selles kontekstis loendtüüp, st peale metamudeli loomist tuleb tüübi alusel loodud tabelis registreerida sellesse tüüpi kuuluvad väärtused.

Iga metamudeli objekt võib olla seotud teiste sama metamudeli objektidega läbi üldistuse, mis on defineeritud tabelis *generalization*, või objektide vahelise seose, mis on defineeritud tabelis *association*. Viimase variandi puhul on võimalik defineerida ka seoste kordsus (tabel *d_multiplicity*), mida arvestatakse mudelite hoidmiseks mõeldud tabelite piirangute loomisel.

Joonis 10: WebMeta4 administraatori andmebaasiskeem – metamudeli ja objektide seosed

Tabel 2 annab ülevaate andmebaasis tehtud arendustöö mahust, tuues välja erinevat tüüpi andmebaasiobjektide arvu metaandmete hoidmiseks mõeldud skeemis *mmm*.

Tabel 2: Andmebaasiobjektide arv skeemis mmm

Andmebaasiobjekti tüüp	Arv
Baastabelid	16
Vaated	2

Andmebaasiobjekti tüüp	Arv
Trigerid	24
Funktsioonid, mis pole otse seotud trigeritega, kuid mida kutsutakse välja trigeri funktsioonidest	60

5.3 Kõikehõlmav Andmebaasi Turvalisuse Mudel

Kuna WebMeta3 järgmist versiooni planeerides sai palju rõhku pandud andmebaasi rolli suurendamisele, siis sai ka juhendaja soovitusel kasutusele võetud Kõikehõlmav Andmebaasi Turvalisuse Mudel. Kui klassikaliste veebirakenduste puhul jäetakse kogu kasutajate õiguste ning ka turvalisuse pool rakenduse kanda, hoides andmebaasis ainult kasutaja kohast infot, siis valitud lähenemise puhul kasutatakse maksimaalselt ära andmebaasi kasutajate õiguste süsteemi. (Downs, 2009)

Selle saavutamiseks luuakse iga süsteemi lõppkasutaja jaoks ka andmebaasi kasutaja ning andmeid vajava süsteemi puhul võetakse kasutusele juba sisse loginud kasutaja andmebaasisessioon. Kasutajapõhised sessioonid toovad kaasa selle, et väga detailselt saab piirata, milliste andmetega on kasutajal õigus tegeleda, mis omakorda parandab kaitset SQL-süstimise tüüpi rünnete vastu. Kaitse tuleneb sellest, et kasutaja pääseb tänu talle määratud rollidele ligi ainult talle mõeldud andmetele konkreetse skeemi piirides ning pole näiteks võimalik pääseda ligi teiste kasutajate skeemidele või kasutajate andmetele. Andmebaasi kasutajate puhul on võimalus hallata juurdepääse läbi erinevate andmebaasi rollide, mis vabastab täiendavast kohustusest, et kontrollida, millistele skeemidele ja tabelitele on tal juurdepääs. Kui rakendus suhtleks andmebaasiga kui üks kasutaja, siis tuleks sellele rakendusele anda kõik õigused, mida sellel rakendusel iganes vaja läheb. Kui rakendust õnnestub rünnata SQL-süstimise meetodil, siis oleks andmebaasis õiguste poolest vähe takistusi pahanduse tegemiseks. Samuti poleks võimalik andmebaasi tasemel piisavalt täpselt jälgida, kes on muudatuse/pahanduse tegija, sest andmebaasisüsteem logib selle kui rakendusele vastava kasutaja, mitte kui lõppkasutaja.

Kasutatava lahenduse puhul tuleb arvestada, et kasutajanimed ja rolli nimed peavad olema serveril asuva PostgreSQL andmebaaside klastri piires unikaalsed. Seega kui loodava kasutaja kasutajanimi või metamudeli alusel loodava rolli nimi on serveris juba kasutusel, siis on tulemuseks erand.

Kogu selle positiivse juures on negatiivseks küljeks see, et ei saa andmebaasiühenduste jaoks kasutada laialtlevinud ühenduste haldust *connection pooling*, mille puhul vähendatakse vajalike andmebaasiühenduste hulka sellega, et jagatakse kasutajate vahel mitteaktiivseid ühendusi. Valitud andmebaasi turvalisuse mudeli puhul tähendab aga iga kasutaja ühte andmebaasi ühendust.

5.4 Sündmuste-tegevuste vastavustabelid

Sündmuste tegevuste vastavustabelid on aluseks andmebaasi trigerite loomisele.

Vastavalt administraatori poolt tulevale sisendile luuakse/muudetakse/kustutatakse trigerite abil andmebaasiobjekte. Järgnevalt on kirjeldatud andmebaasi puudutavad sündmused ning nende sündmuste tagajärjel andmebaasis toimuvad tegevused. Tegevused on jaotatud kaheks: meta-andmete skeemis *mmm* olevate andmete põhjal toimuvad muudatused/kontrollid ja metamudelile (CASE vahendile) vastava skeemiga seotud muudatused. *mmm* skeemis tehtavate tegevuste juures ei mainita sündmuste juures kirja pandud andmemuudatuste läbiviimist.

Enamustel kirjeldatavatel objektide metaandmete hulka kuuluvad atribuudid lühike nimi (*short name*) ja täisnimi (*full name*). Lühike nimi muudetakse trigerite poolt sobivaks andmebaasiobjekti nimeks (identifikaatoriks). See nimi peab olema kooskõlas andmebaasisüsteemis nimedele kehtestatud piirangutega. Samuti tuleb nimede puhul jälgida jaotises 4.3 nimetatud piiranguid. Nimes lubatakse ainult ladina tähestiku tähti, numbreid ning alakriipsu ning nimi ei tohi olla pikem kui PostgreSQL identifikaatorite vaikimisi maksimaalne pikkus – 63 baiti. Selle saavutamiseks eemaldatakse lühikesest nimest sümbolid +":$%;()=[]{}#&.!?/,' ning asendatakse tühikud alakriipsuga. Pikema nime puhul nimi lühendatakse 63 baidi pikkuseni. Täisnimi on mõeldud CASE vahendis kasutajatele näitamiseks.

5.4.1 Kasutaja

Kasutajate haldamiseks kasutatakse andmebaasi kasutajaid ning õiguste andmiseks rolle. Rollid võimaldavad ühtselt defineerida õigusi andmebaasi skeemide ja tabelite kaupa (vt Tabel 3).

Loodavas metamudelis tähendab ADMIN roll seda, et selle rolli kandja võib metamudeli alusel mudeleid luua, vaadata, muuta ja kustutada. Loodavas metamudelis tähendab USER roll seda, et selle rolli kandja võib metamudeli alusel mudeleid ainult vaadata.

Tabel 3: Kasutajate haldusega seotud sündmuste-tegevuste vastavustabel

Sündmus	**Tegevus**
Luuakse uus kasutaja (lisatakse uus rida tabelisse *mmm.user*)	**andmebaasi tasemel:** Kontrollitakse, kas parool on piisavalt turvaline. Kui ei ole, siis katkestatakse. Luuakse (CREATE USER lausega) uus andmebaasi kasutaja (ei ole ülikasutaja), mille nimi on kasutaja e-meili aadress ja parool on loodud kasutaja parool. Loodud kasutaja nimi on piiritletud identifikaator (jutumärkides), et nimes saaks kasutada e-meilis lubatud märke nagu punkt.
Kasutaja seotakse metamudeliga *USER* õigustes. (Lisatakse uus rida tabelisse *mmm.user_metamodel*).	**andmebaasi tasemel:** Andmebaasi kasutajale antakse (GRANT lausega) vastava metamudeli *USER* roll (ilma õigusteta seda rolli edasi jagada) nimega r_<skeemi nimi>_USER (roll tekib koos metamudeli loomisega).
Kasutajalt võetakse metamudeli suhtes *USER* roll (kustutatakse rida tabelist *mmm.user_metamodel*)	**andmebaasi tasemel:** Andmebaasi kasutajale võetakse (REVOKE lausega) vastava metamudeli *USER* roll nimega r_<skeemi nimi>_USER.
Kasutaja seotakse metamudeliga *ADMIN* õigustes (lisatakse uus rida tabelisse *mmm.user_metamodel*)	**andmebaasi tasemel:** Andmebaasi kasutajale antakse (GRANT lausega) vastava metamudeli *ADMIN* roll (ilma õigusteta seda rolli edasi jagada) nimega r_<skeemi nimi>_ADMIN

Sündmus	**Tegevus**
	(roll tekib koos metamudeli loomisega).
Kasutajalt võetakse metamudeli suhtes *ADMIN* roll (kustutatakse rida tabelist *mmm.user_ metamodel*)	**andmebaasi tasemel:** Andmebaasi kasutajalt võetakse (REVOKE lausega) vastava metamudeli *ADMIN* roll nimega r_<skeemi nimi>_ADMIN.
Muudetakse kasutaja parooli	**andmebaasi tasemel:** Kontrollitakse, kas parool on piisavalt turvaline. Kui ei ole, siis katkestatakse. Muudetakse (ALTER USER lausega) andmebaasi kasutaja parooli.
Kustutatakse kasutaja (kustutatakse rida tabelist *mmm.user*)	***mmm* skeem:** kustutatakse kasutajaga seosed metamudelitega tabelist *mmm.user_metamodel*, sellega käivitatakse tegevused „Kasutajalt võetakse metamudeli suhtes USER roll“, „Kasutajalt võetakse metamudeli suhtes ADMIN roll“. **andmebaasi tasemel:** eemaldatakse kasutaja andmebaasist „DROP USER“-lausega.

5.4.2 Metamudel

Igale metamudelile vastab andmebaasi skeem, milles olevates tabelites registreeritakse sellele metamudelile vastava keele ja CASE vahendi abil loodud mudeleid. Metamudel võib olla nii aktiivne kui ka passiivne. Sõltuvalt sellest on vastav CASE vahend lõppkasutajale nähtav/mittenähtav. Metamudeliga saab siduda kasutajaid, kes saavad vastavat CASE vahendit kasutada. Tabel 4 kirjeldab metamudeliga seotud tegevusi.

Tabel 4: Metamudelite haldusega seotud sündmuste-tegevuste vastavustabel

Sündmus	**Tegevus**
Lisatakse uus metamudel, mis kirjeldab mingit modelleerimiskeelt (lisatakse uus rida tabelisse *mmm.metamodel*).	***mmm* skeem:** Vajadusel muudetakse metamudeli lühikest nime, et seda oleks võimalik kasutada andmebaasi skeemi nimena. Kui juba leidub sellise lühikese nimega metamudel, siis katkestatakse. Kui lühike nimi on mõni reserveeritud nimedest (*mmm, public, information_schema, pg_catalog* või *pg_toast_test_1*), siis katkestatakse. Kui kasutaja pole sisestanud pikka nime, siis enne lühikese nime muutmist kirjutatakse lühike nimi ka pika nime välja. **Metamudelile vastav skeem:** Luuakse skeem (CREATE SCHEMA lausega), mille nimeks saab metamudeli lühike nimi <lühike_nimi> Luuakse rollid (CREATE ROLE lausega) r_<lühike_nimi>_USER ja r_<lühike_nimi>_ADMIN, millele antakse *USAGE* õigus skeemi <lühike nimi> suhtes. Rollidele antakse andmebaasiga ühenduse loomise (CONNECT) õigus ja loodud skeemi kasutamise (USAGE) õigus (GRANT lausega) (ilma õigusteta seda õigust edasi jagada). Skeemi lisatakse süsteemne abitabel *sysg_file* (*sysg* nagu „system generated“), et võimaldada lõppkasutajal registreerida mudelifaile. Seda ei registreerita metamudeli objektina. Selles tabelis on veerg nii faili

Sündmus	**Tegevus**
	nime jaoks kui ka faili enda jaoks (*bytea* tüüpi). Plaanis oleks hoida faile andmebaasis, et neile rakenduksid samad turvakontrolli, varundamis/taastamise ja transaktsioonide mehhanismid nagu ülejäänud andmetele. Faili nimi peab olema unikaalne.
Muudetakse metamudeli lühikest nime (muudetakse rida tabelis *mmm.metamodel*)	***mmm skeemis*:** Kui juba leidub sellise lühikese nimega metamudel, siis katkestatakse. Kui lühike nimi on mõni reserveeritud nimedest (*mmm, public, information_schema, pg_catalog* või *pg_toast_test_1*), siis katkestatakse. **Metamudelile vastav skeem:** Muudetakse andmebaasi skeemi nime (ALTER SCHEMA lausega). Muudetakse skeemiga seotud rollide nimesid (ALTER ROLE lausega).
Kustutatakse metamudel (kustutatakse rida tabelist *mmm.metamodel*).	***mmm skeemis*:** Kustutatakse metamudeli objektide andmed. See käivitab sündmused: „Kustutatakse metamudeli objekt“. Kustutatakse metamudeli klassifikaatorid. See käivitab sündmused: „Andmetüübi kustutamine“. **Metamudelile vastav skeem:** Rollidelt r_<lühike_nimi>_USER ja r_<lühike_nimi>_ADMIN võetakse kõik skeemi kasutamise (USAGE) õigus ja andmebaasiga ühenduse loomise (CONNECT) õigus.

Sündmus	Tegevus
	Kustutatakse skeem nimega metamudeli <lühike_nimi> ja kustutatakse kõik sellesse skeemi kuuluvad objektid (DROP SCHEMA … CASCADE lausega). Kustutakase andmebaasi roll nimega r_<lühike_nimi>_USER (DROP ROLE lausega). Kustutakase andmebaasi roll nimega r_<lühike_nimi>_ADMIN (DROP ROLE lausega).

5.4.3 Metamudeli objekt

Guizzardi et al. (2004) kirjeldab ontoloogiliselt hästi-defineeritud UML kontseptuaalse modelleerimise profiili. Profiil kirjeldab hulga kontseptuaalse mudeli klasside iseloomustamiseks mõeldud kategooriaid (nt *kind*, *subkind*, *phase*) ning esitab põhjalikul kaalutlusel põhineva reeglistiku, millised on lubatud üldistusseosed erinevate nendesse kategooriatesse kuuluvate klasside vahel. WebMeta nõuab metamudeli objekti defineerimisel objekti tüübi määramist. Võimalikeks tüüpideks on profiili poolt määratud kategooriad (KIND, SUBKIND, PHASE. ROLE, CATEGORY, ROLE MIXIN, MIXIN) ning lisaks RELATIONSHIP, mis on mõeldud seose objektide iseloomustamiseks. Metamudeli objekti alusel luuakse andmebaasis tabel, et võimaldada hoida mudelite elemente. RELATIONSHIP tüüpi objekt on mõeldud n-aarsete (n>=2) seoste defineerimiseks.

Seda tüüpi metamudeli objekti puhul tekitatakse siduv vahetabel koos vastavate välisvõtmetega. Samuti on võimalik lisada tabelite vaheline seos läbi üldistusseose. Üldistusseost lubab süsteem defineerida vaid siis, kui see on kooskõlas Guizzardi et al. (2004) profiiliga. Kuna metamudeli objektidele on tüübi määramine kohustuslik ning süsteem teostab automaatselt sellist kontrolli, siis aitab see parandada loodavate metamudelite (ja selle kaudu ka keelte) kvaliteeti. Hetkel ei toetata mitmest pärimist, ehk ühel metamudeli objektil saab olla ainult üks ülaobjekt. Alamobjektide hulk pole piiratud. Tabel 5 kirjeldab metamudeli objektidega seotud trigerite tegevusi.

Tabel 5: Metamudeli objektiga seotud trigerite sündmuste vastavustabel

Sündmus	**Tegevus**
Lisatakse metamudeli objekt (lisatakse uus rida tabelisse *mmm.metamodel_object*)	***mmm skeemis*:** Vajadusel muudetakse objekti lühikest nime, et seda oleks võimalik kasutada andmebaasi tabeli nimena. Kontrollitakse, et uus nimi ei kattuks ühegi teise selle metamudeli objekti lühikese nimega ja kasutaja-defineeritud tüübi lühikese nimega, vastasel juhul katkestatakse. Kui kasutaja pole sisestanud pikka nime, siis kirjutatakse lühike nimi ka pika nime välja enne lühikese nime muutmist.w **Metamudelile vastav skeem:** Luuakse objektile vastav tabel (CREATE TABLE lausega), milles on primaarvõtme veerg nimega <lühike_nimi>_id (tüüp SERIAL), _updated (tüüp TIMESTAMP, kohustuslik veerg, vaikimisi väärtus LOCALTIMESTAMP(0)), _updated_by (tüüp VARCHAR(128), kohustuslik veerg, vaikimis väärtus SESSION_USER). Need on süsteemsed veerud, mida ei registreerita metamudeli objekti alamobjektidena. Rollile r_<skeemi_nimi>_USER antakse tabeli <lühike_nimi> suhtes SELECT õigus (GRANT lausega) (ilma õigusteta seda õigust edasi jagada). Rollile r_<skeemi_nimi>_ADMIN antakse SELECT, INSERT, UPDATE ja DELETE õigused tabeli <lühike_nimi> suhtes ning samuti kõik õigused selle tabeliga seoses tekkinud arvujada generaatori suhtes (GRANT lausega) (ilma õigusteta seda õigust edasi jagada).

Sündmus	**Tegevus**
Lisatakse metamudeli objektile üldistusseos (lisatakse uus rida tabelisse *mmm.generalization*)	***mmm skeemis*:** Kontrollitakse, kas loodav ülatüüp on lubatud sellele metamudeli objekti tüübile. kui ei ole lubatud, siis katkestatakse. Kontrollitakse et objekt ei oleks enda otsene ega kaudne vanem või laps. Kui on, siis katkestatakse. **Metamudelile vastav skeem:** Lisatakse objektile vastavasse tabelisse uus kohustuslik veerg, milles on viited üldisemale objektile nimega <ülemobjekti_lühike_nimi>_id (ALTER TABLE … ADD COLUMN lausega). Objektile vastavas tabelis luuakse välisvõtme kitsendus (<ülemobjekti_lühike_nimi>_id) REFERENCES <ülemobjekti_lühike_nimi> (<ülemobjekti_lühike_nimi>_id ON DELETE CASCADE. Kitsenduse nimi on fk_<alamobjekti_lühike_nimi>_<ülemobjekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Objektile vastavas tabelis luuakse kitsendus UNIQUE (<ülemobjektid_lühike_nimi>_id) nimega uk_<alamobjekti_lühike_nimi>_<ülemobjekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega).
Kustutatakse metamudeli üldistusseos (kustutatakse rida tabelist *mmm.generalization*)	***Metamudelile vastav skeem:*** Eemaldatakse välivõtme veerg alamtabelist <ülemobjekti_lühike_nimi>_id (ALTER TABLE … DROP COLUMN lausega).
Muudetakse metamudeli objekti lühikest nime (muudetakse rida tabelis *mmm.metamodel_object*)	***mmm skeemis:*** Vajadusel muudetakse objekti lühikest nime, et seda oleks võimalik kasutada andmebaasi tabeli nimena. Kontrollitakse,

Sündmus	**Tegevus**
	et uus nimi ei kattuks ühegi teise selle metamudeli objektide lühikese nimega ja kasutaja-defineeritud tüübi lühikese nimega, vastasel juhul katkestatakse. ***Metamudelile vastav skeem:*** Muudetakse tabeli nime (ALTER TABLE … RENAME lausega).
Muudetakse metamudeli objekti tüüpi (muudetakse rida tabelis *mmm.metamodel_object*)	***mmm skeemis*:** Lubatud ainult KIND, SUBKIND, PHASE. ROLE, CATEGORY, ROLE MIXIN, MIXIN tüüpidele. Kontrollitakse uue objekti tüübi koosõla üldistuste abil seotud objektide tüüpidega. Vastuolu korral katkestatakse.
Luuakse metamudeli objektide vaheline seos (lisatakse rida tabelisse *mmm.association*)	***mmm skeemis*:** Vajadusel muudetakse seose lühikest nime, et seda oleks võimalik kasutada tabeli veeru nimena. Kontrollitakse, et uus nimi ei kattuks ühegi teise selle metamudeli objekti lühikese nimega ja kasutaja-defineeritud tüübi lühikese nimega, vastasel juhul katkestatakse. Kui kasutaja pole sisestanud pikka nime, siis kirjutatakse enne lühikese nime muutmist lühike nimi ka pika nime välja. **Metamudelile vastav skeem:** Kui lisatud seos on kujul et kordused on vastavalt „täpselt üks“ – „null või rohkem“, siis lisatakse „null või rohkem“ poolse kordsusega metamudeli objekti tabelisse kohustuslik välisvõtme veerg nimega <seose_lühike_nimi>_id (ALTER TABLE … ADD COLUMN lausega). Loodud veerule lisatakse välisvõtme kitsendus „täpselt üks“ poolse kordsusega

Sündmus	**Tegevus**
	metamudeli objekti tabeli primaarvõtme veerule nimega fk_<seose_lühike_nimi>_id_<viidatava_objekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Kui lisatud seos on kujul et kordused on vastavalt „täpselt üks“ – „null või üks“ , siis lisatakse „null või üks“ poolse kordsusega metamudeli objekti tabelisse kohustuslik välisvõtme veerg nimega <seose_lühike_nimi>_id (ALTER TABLE … ADD COLUMN lausega). Loodud veerule lisatakse välisvõtme kitsendus „täpselt üks“ poolse kordsusega metamudeli objekti tabeli primaarvõtme veerule nimega fk_<seose_lühike_nimi>_id_<viidatava_objekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Lisaks lisatakse loodud veerule <seose_lühike_nimi>_id UNIQUE kitsendus nimega uk_<seose_lühike_nimi>_<viidatava_objekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Kui lisatud seos on kujul et kordused on vastavalt „null või üks“ – „null või rohkem“ , siis lisatakse „null või rohkem“ poolse kordsusega metamudeli objekti tabelisse mittekohustuslik välisvõtme veerg nimega <seose_lühike_nimi>_id (ALTER TABLE … ADD COLUMN lausega). Loodud veerule lisatakse välisvõtme kitsendus „null või üks“ poolse kordsusega metamudeli objekti tabeli primaarvõtme veerule nimega fk_<seose_lühike_nimi>_id_<viidatava_objekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Kui lisatud seos on kujul et kordused on vastavalt „null või üks“ – „null või üks“ , siis lisatakse ühe (sisendis end1 objektiks oleva) metamudeli objekti tabelisse kohustuslik välisvõtme veerg nimega <seose_lühike_nimi>_id (ALTER TABLE … ADD COLUMN lausega). Loodud veerule

Sündmus	**Tegevus**
	lisatakse välisvõtme kitsendus teise metamudeli objekti tabeli primaarvõtme veerule nimega fk_<seose_lühike_nimi>_id_<viidatava_objekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Lisaks lisatakse loodud veerule <seose_lühike_nimi>_id UNIQUE kitsendus nimega uk_<seose_lühike_nimi>_<viidatava_objekti_lühike_nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Seose lühikese nime kasutamine välisvõtme veergude ning kitsenduste nimedes võimaldab lisada seoseid ka objektile endale või mitmekordseid seoseid erinevate metamudeli objektide vahel.
Kustutatakse metamudeli objektide vaheline seos (kustutatakse rida tabelist *mmm.association*)	**Metamudelile vastav skeem:** Eemaldatakse välivõtme veerg <seose_lühike_nimi>_id (ALTER TABLE … DROP COLUMN lausega).
Kustutatakse metamudeli objekt (kustutatakse rida tabelist *mmm.metamodel_object*)	***mmm skeemis*:** Kustutatakse metamudeli objektiga seotud seosed teiste metamudeli objektidega: käivitatakse sündmused „Kustutatakse metamudeli objektide vaheline seos“ ning „Kustutatakse metamudeli üldistusseos“. Kustutatakse metamudeli objektiga seotud alamobjektid. Sellega käivitatakse sündmus „Kustutatakse metamudeli objekti alamobjekt“. **Metamudelile vastav skeem:** Kontrollitakse, et tabelis ei eksisteeriks andmeid, vastasel juhul katkestatakse. Rollilt r_<skeemi_nimi>_USER võetakse tabeli

Sündmus	Tegevus
	<lühike_nimi> suhtes SELECT õigus (REVOKE lausega). Rollilt r_<skeemi_nimi>_ADMIN võetakse SELECT, INSERT, UPDATE ja DELETE õigused tabeli <lühike_nimi> suhtes ning samuti kõik õigused selle tabeliga seoses tekkinud arvujada generaatori suhtes (REVOKE lausega) Kustutakse tabel nimega <lühike_nimi> (DROP TABLE lausega).

5.4.4 Metamudeli objekti alamobjekt

Metamudeli objekti alamobjektile vastab metamudeli skeemis oleva tabeli üks veerg. Alamobjekti puhul tuleb ka määrata tüüp, kohustuslikkus ning unikaalsus.

Eristan kahte tüüpide kategooriat – andmebaasis defineeritud tüübid ning metamudeli kirjeldaja poolt loodud kasutaja-defineeritud tüübid ehk klassifikaatorid. Võimalused asendada alamobjekti tüüp teise tüübiga on hetkel väga piiratud. Hetkel on toetatud ainult üleminek INTEGER tüübilt TEXT tüübile. Tabel 6 kirjeldab metamudeli objekti alamobjektiga seotud trigerite tegevusi.

Tabel 6: Metamudeli objektide alamobjektide haldamisega seotud sündmuste-tegevuste vastavustabel

Sündmus	Tegevus
Lisatakse metamudeli objekti alamobjekt (lisatakse rida tabelisse *mmm.metamodel_subobject*)	***mmm skeemis*:** Vajadusel muudetakse objekti lühikest nime, et seda oleks võimalik kasutada andmebaasi tabeli veeru nimena. Kui objektil juba on sellise nimega alamobjekt, siis katkestatakse. Kui kasutaja pole sisestanud pikka nime, siis kirjutatakse enne lühikese nime muutmist lühike nimi ka pika nime välja.

Sündmus	**Tegevus**
	Metamudelile vastav skeem: Kui on valitud unikaalsuse nõue, siis käitutakse nagu unikaalsuse nõude muutmisel (vt. sündmus „Muudetakse metamudeli objekti alamobjekti unikaalsuse nõuet“). Kui on valitud kohustuslikkuse nõue, siis käitutakse nagu kohustuslikkuse nõude muutmisel. (vt. sündmus „Muudetakse metamudeli objekti alamobjekti kohustuslikkuse nõuet“). Kui on valitud andmebaasi poolt defineeritud andmetüüp, siis luuakse veerg vastavalt valitud andmetüübiga (ALTER TABLE .. ADD COLUMN lausega). Kui on valitud kasutaja poolt loodud andmetüüp lühikese nimega <kt_lühike_nimi>, siis luuakse veerg nimega <lühike_nimi>_id (ALTER TABLE … ADD COLUMN lausega) ning antud veerule välisvõti kasutaja defineeritud tüübi tabelile nimega fk_<lühike nimi>_<kt_lühike nimi>_id (ALTER TABLE … ADD CONSTRAINT lausega)
Muudetakse metamudeli objekti alamobjekti lühikest nime (muudetakse rida tabelis *mmm.metamodel_subobject*)	***mmm skeemis*:** Vajadusel muudetakse väärtust <lühike_nimi>, et see oleks kasutatav veeru nimena. Kui objektil juba on sellise nimega alamobjekt, siis katkestatakse. **Metamudelile vastav skeem:** Muudetakse tabelis veeru nime (ALTER TABLE …

Sündmus	**Tegevus**
	RENAME lausega).
Muudetakse metamudeli objekti alamobjekti andmetüüpi (muudetakse rida tabelis *mmm.metamodel_subobject*)	***Mmm skeemis*:** Muuta saab ainult andmebaasis defineeritud tüüpe, milledest omakorda on toetatud ainult üleminek INTEGER tüübilt TEXT tüübile. Kui valitud tüüp ei vasta tingimustele, siis katkestatakse. **Metamudelile vastav skeem:** Muudetakse veeru tüüpi (ALTER TABLE … ALTER COLUMN lausega).
Muudetakse metamudeli objekti alamobjekti kohustuslikkuse nõuet (muudetakse rida tabelis *mmm.metamodel_subobject*)	**Metamudelile vastav skeem:** Kontrollitakse, kas tabelis olevad andmed võimaldavad muutust (st kui määratakse alamobjekt kohustuslikus, siis igas reas on olemas alamobjektile vastav väärtus). Kui ei, siis katkestatakse. Kui lisati kohustuslikkuse nõue, siis lisatakse *NOT NULL* tabeli veeru deklaratsioonile (ALTER TABLE … ALTER COLUMN lausega). Kui eemaldati kohustuslikkuse nõue, siis eemaldatakse *NOT NULL* tabeli veeru deklaratsioonilt (ALTER TABLE … ALTER COLUMN lausega).
Muudetakse metamudeli objekti alamobjekti unikaalsuse nõuet (muudetakse rida tabelis *mmm.metamodel_subobject*)	**Metamudelile vastav skeem:** Kontrollitakse, kas tabelis olevad andmed võimaldavad muutust (st kui määratakse alamobjekt unikaalseks, siis igas reas on unikaalne väärtus). Kui ei, siis katkestatakse.

Sündmus	**Tegevus**
	Kui lisati unikaalsuse nõue, siis lisatakse tabeli veerule UNIQUE kitsendus uk_<tabeli nimi>_<veeru nimi> (ALTER TABLE … ADD CONSTRAINT lausega). Kui eemaldati unikaalsuse nõue, siis kustutatakse veerult UNIQUE kitsendus uk_<tabeli nimi>_<veeru nimi> (ALTER TABLE … DROP CONSTRAINT lausega).
Kustutatakse metamudeli objekti alamobjekt (kustutatakse rida tabelist *mmm.metamodel_subobject*)	**Metamudelile vastav skeem:** Kontrollitakse, et veerus ei oleks andmeid. Kui on, siis katkestatakse. Kustutatakse alamobjektile vastav veerg (ALTER TABLE … DROP COLUMN lausega).

5.4.5 Kasutaja defineeritud andmetüüp

Kasutaja saab defineerida uusi andmetüüpe. Andmetüüp on nime omav lõplik väärtuste hulk. Iga sellise andmetüübi kohta luuakse sellesse kuuluvate väärtuste hoidmiseks eraldi tabel, mis on metamudelile vastavas skeemi. Seega saab iga taolist andmetüüpi kasutada ainult ühe metamudeli piires. Tabel 7 kirjeldab kasutaja defineeritud andmetüübiga seotud trigerite tegevusi.

Tabel 7: Kasutaja defineeritud andmetüüpide haldamisega seotud sündmuste-tegevuste vastavustabel

Sündmus	**Tegevus**
Andmetüübi loomine (lisatakse rida tabelisse *mmm.user_defined_data_type*)	***Mmm skeemis*:** Vajadusel muudetakse lühikest nime, et oleks kasutatav andmebaasi tabeli nimena. Kontrollitakse, et metamudelis poleks kasutaja-defineeritud tüüpi ega metamudeli objekti nimega <lühike_nimi>. Kui on,

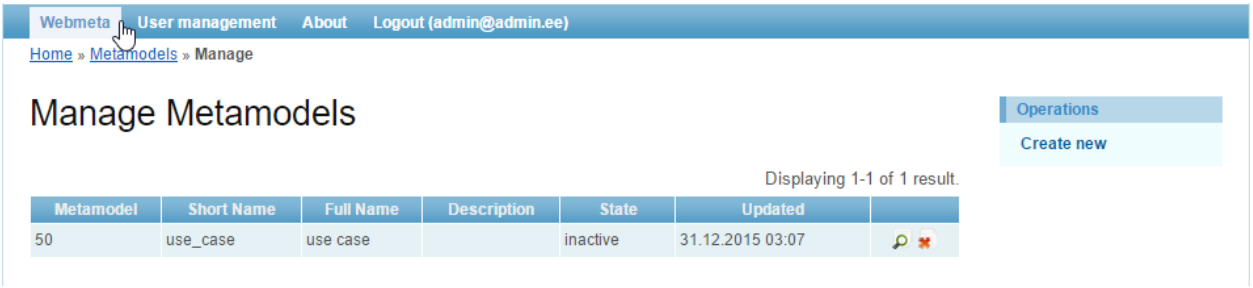
Sündmus	**Tegevus**
	siis katkestatakse. **Metamudelile vastav skeem:** Luuakse tabel <lühike_nimi>_type (CREATE TABLE lausega) koos veergudega <lühike_nimi>_type_id (tüüp SERIAL), *name* (tüüp VARHCAR(255)) ja *is_active* (tüüp BOOLEAN). Veerg <lühike_nimi>_type_id on primaarvõti ja *name* on UNIQUE kitsendusega. Veerg *is_active* on kohustuslik. Rollile r_<skeemi_nimi>_USER ja r_<skeemi_nimi>_ADMIN antakse SELECT õigus selle tabeli suhtes (GRANT lausega) (ilma õiguseta seda õigust edasi jagada).
Andmetüübi kustutamine (kustutatakse rida tabelist *mmm.user_defined_data_type*)	**Metamudelile vastav skeem:** Kontrollitakse, kas kasutaja-defineeritud andmetüüpi kasutatakse mõne metamudeli objekti alamobjekti tüübina. Kui jah, siis katkestatakse. Rollildelt r_<skeemi_nimi>_USER ja r_<skeemi_nimi>_ADMIN võetakse SELECT õigus selle tabeli suhtes (REVOKE lausega). Kustutakse andmetüübile vastav tabel (DROP TABLE lausega).

5.5 Kasutajaliides

Kasutajaliidese väljanägemise juures on kasutatud raamistiku poolt vaikimis kaasa tulevat kujundust. Allpool on välja toodud administreerimisliidese kujundus põhiliste kasutusjuhtude korral.

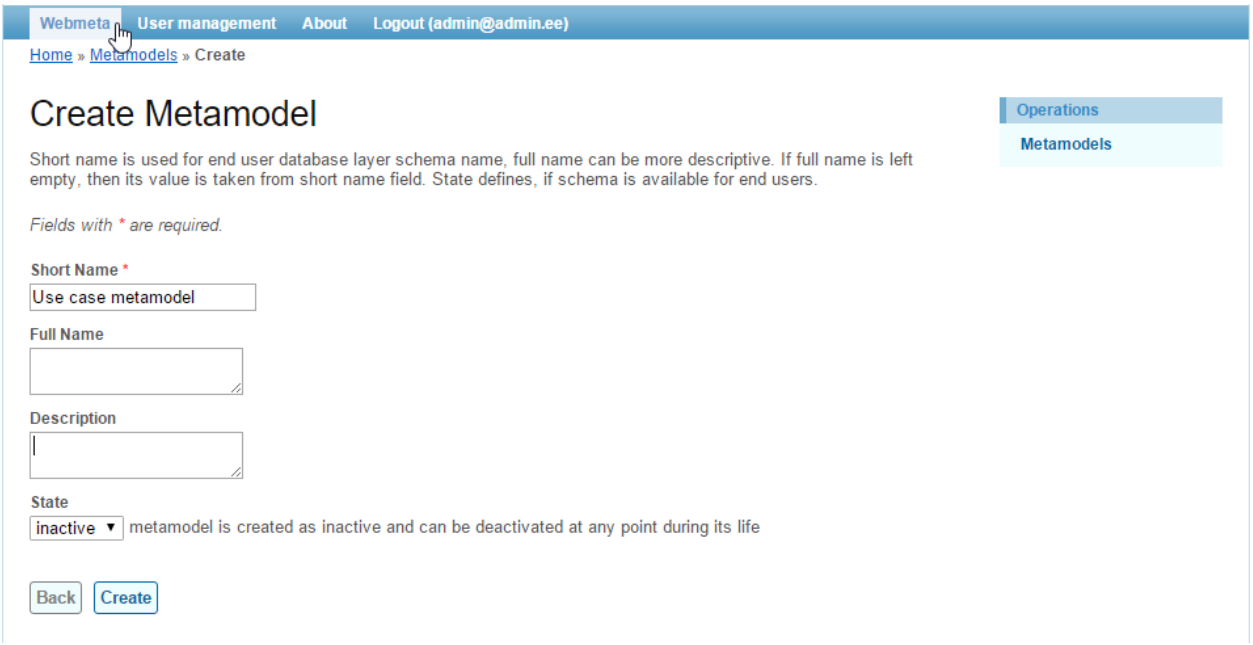
5.5.1 Metamudeli haldus

Metamudelite avalehel on nimekiri defineeritud metamudelitest, ning lingid uue metamudeli loomiseks ning olemasolevate detailandmete vaatamiseks ning kustutamiseks (vt Joonis 11).



Joonis 11: Metamudelite loetelu

Uue metamudeli lisamisel on vaja sisestada vähemasti lühike nimi. Täiendavalt saab lisada täisnime ja kirjelduse ning muuta metamudeli ka kohe aktiivseks. Vaikimisi luuakse metamudel mitteaktiivses olekus (vt Joonis 12).



Joonis 12: Uue metamudeli lisamine

Metamudeli detailandmete lehel on võimalik näha infot metamudeli kohta: id, lühike nimi, täisnimi, kirjeldus, olek ning viimase uuendamise kuupäev ja kellaaeg. Lisaks on võimalik vaadata metamudeliga seotud olevaid metamudeli objekte koos linkidega nende lisamiseks, muutmiseks ning kustutamiseks. Lisaks on lingid klassifikaatorite ning metamudeli objektide

vaheliste seoste haldamiseks (vt Joonis 13). Klassifikaatorite haldamise vaadet kirjeldatakse eraldi jaotises „Klassifikaatorite haldus“. Metamudeli objektide vaheliste seoste haldamis kirjeldatakse peatükis „Metamudeli objektide vaheliste seoste haldus“.

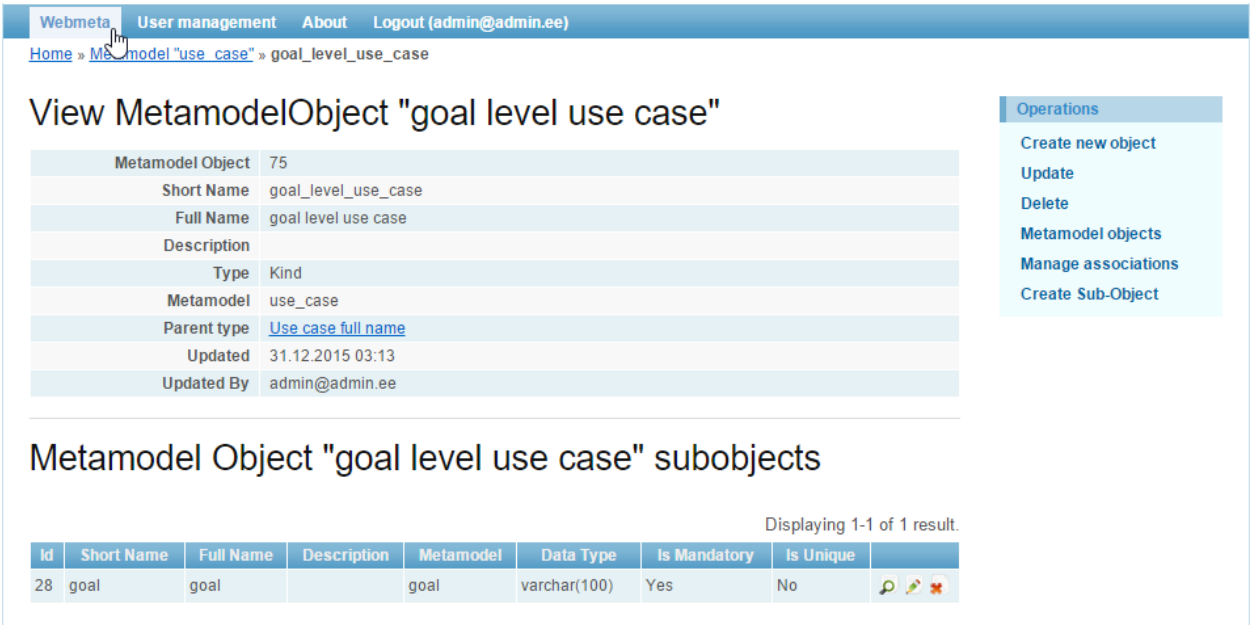
Joonis 13: Metamudeli detailvaade

5.5.2 Metamudeli objekti haldus

Metamudeli objekti sisestamise ja muutmise puhul on võimalik sisestada/muuta nimesid, tüüpi, kirjeldust ning lisada – muuta ülemtüüpi ehk lisada või eemaldada üldistusseost (vt Joonis 14).

Joonis 14: Metamudeli objekti lisamine

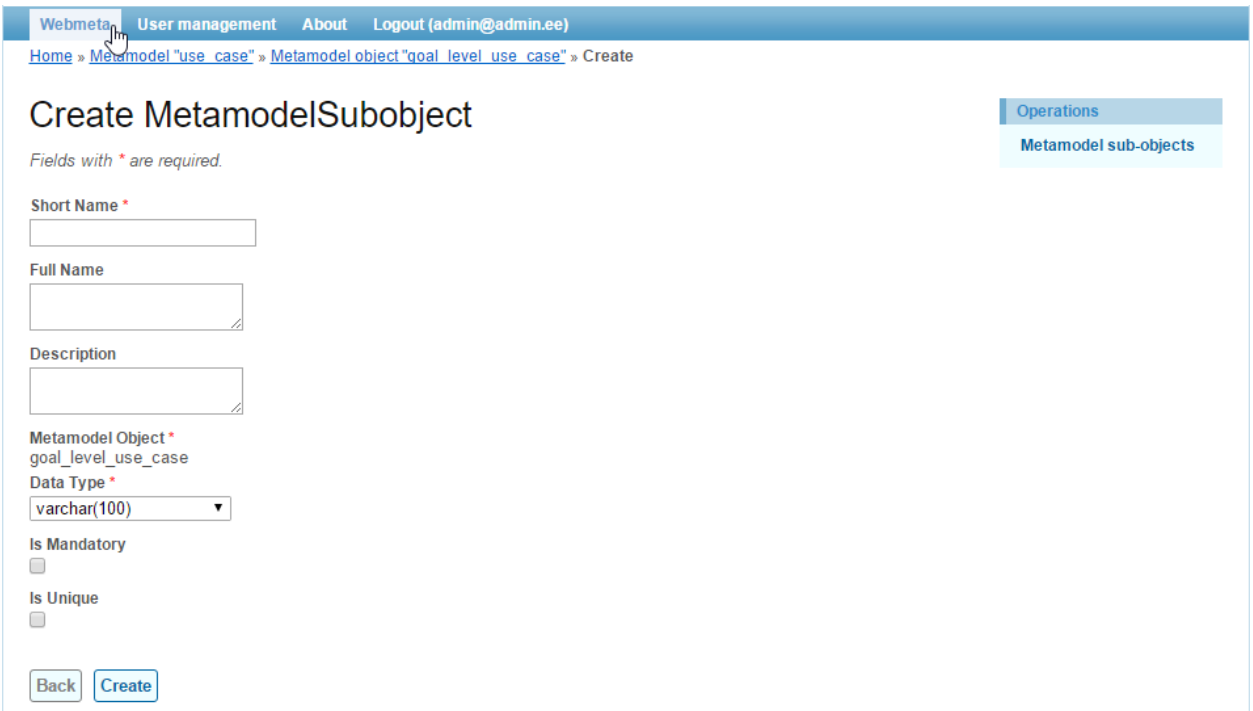
Metamudeli objekti detailvaatest on võimalik näha objekti kohta: id, lühike nimi, täisnimi, kirjeldus, tüüp, metamudel, mille alla ta kuulub, ülemtüüp, seotud objektid (kui seoseid on) ning viimatise muutmise kuupäev ja muutja. Lisaks on vaates näha metamudeli objektiga seotud alamobjekte ning lingid nende loomiseks, detailandmete vaatamiseks, muutmiseks ja kustutamiseks (vt Joonis 15).



Joonis 15: Metamudel objekti detailvaade

5.5.3 Metamudeli objekti alamobjekti haldus

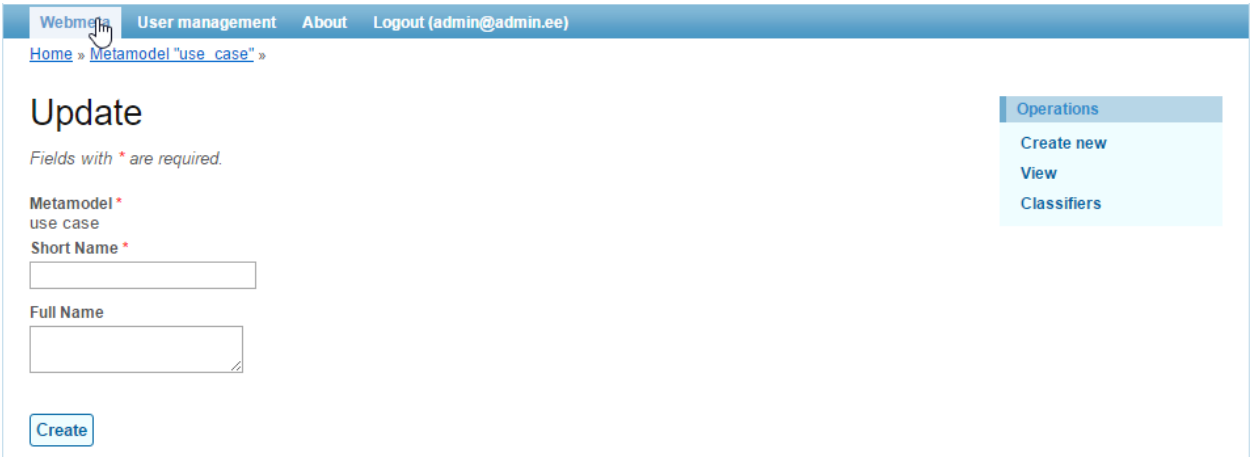
Metamudeli objekti alamobjekti lisamise ja muutmise vaates on võimalik lisada ja muuta lühikest nime, täisnime, kirjeldust, andmetüüpi ning kohuslikkuse ja unikaalsuse piiranguid (vt Joonis 16). Andmetüüpi on võimalik muuta ainult siis, kui olemasolev tüüp on muudetav millekski muuks, vastasel juhul kuvatakse lihtsalt olemasoleva andmetüübi nime.



Joonis 16: Metamudeli objekti alamobjekti lisamine

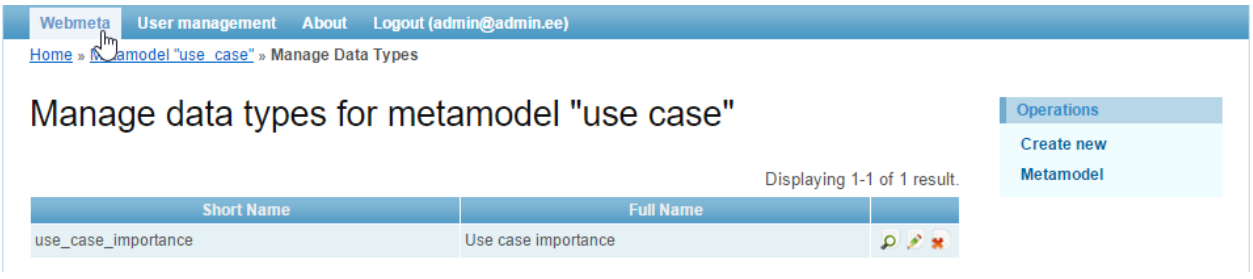
5.5.4 Klassifikaatorite haldus

Klassifikaatorite ehk kasutaja defineeritud andmetüüpide lisamise puhul on võimalik sisestada lühikest nime, täisnime ja kirjeldust. Samuti kuvatakse metamudeli nimi, mille alla antud klassifikaator kuulub (vt Joonis 17).



Joonis 17: Uue andmetüübi lisamine

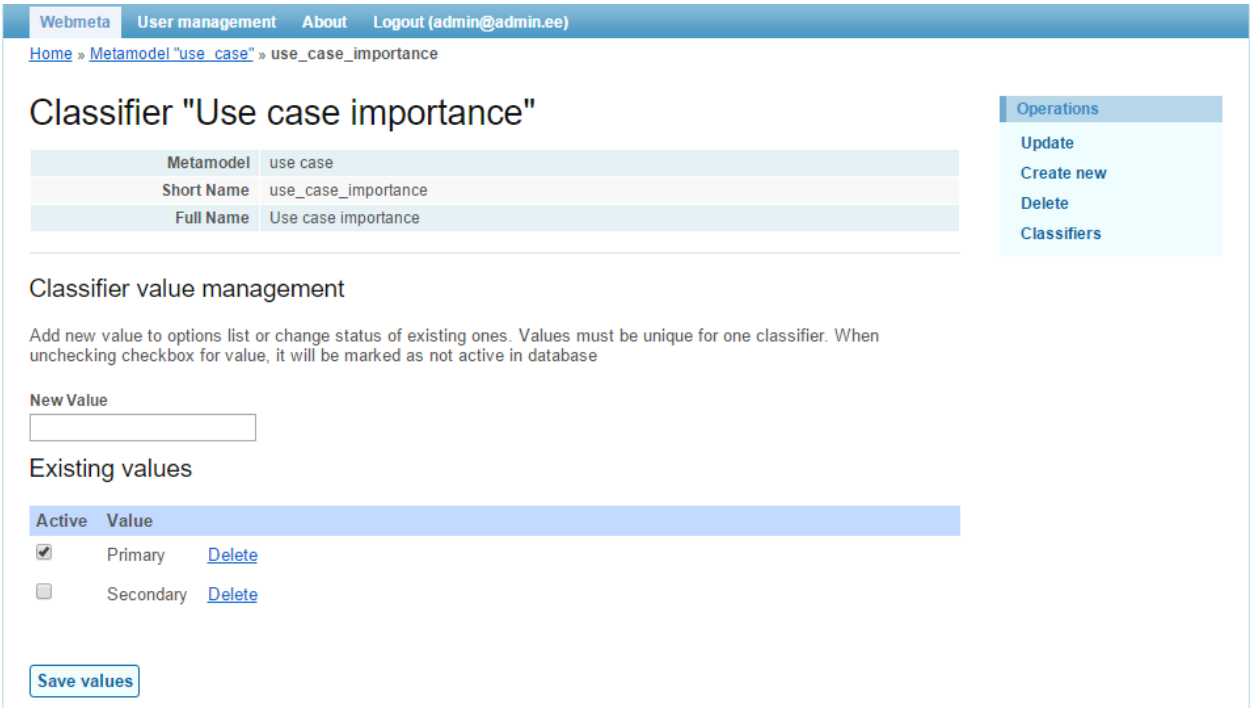
Metamudeli klassifikaatorite loetelus on nimekiri metamudelile loodud klassifikaatoritest koos linkidega uue klassifikaatori loomiseks, olemasolevate detailandmete vaatamiseks ning kustutamiseks. (Vt Joonis 18)



Joonis 18: Andmetüüpide loetelu

Klassifikaatori vaate all on võimalik lisada talle võimalikke väärtusi, mida lõppkasutajale näidata. Samuti saab määrate, kas väärtus on aktiivne või mitte. (Vt Joonis 19) Kui väärtus on aktiivne, siis mudelite haldaja (vaade, mida käesolevas töös ei realiseeritud) näeb seda klassifikaatoriga kirjeldatud andmete lisamisel või muutmisel valitavate väärtuste nimekirjas.

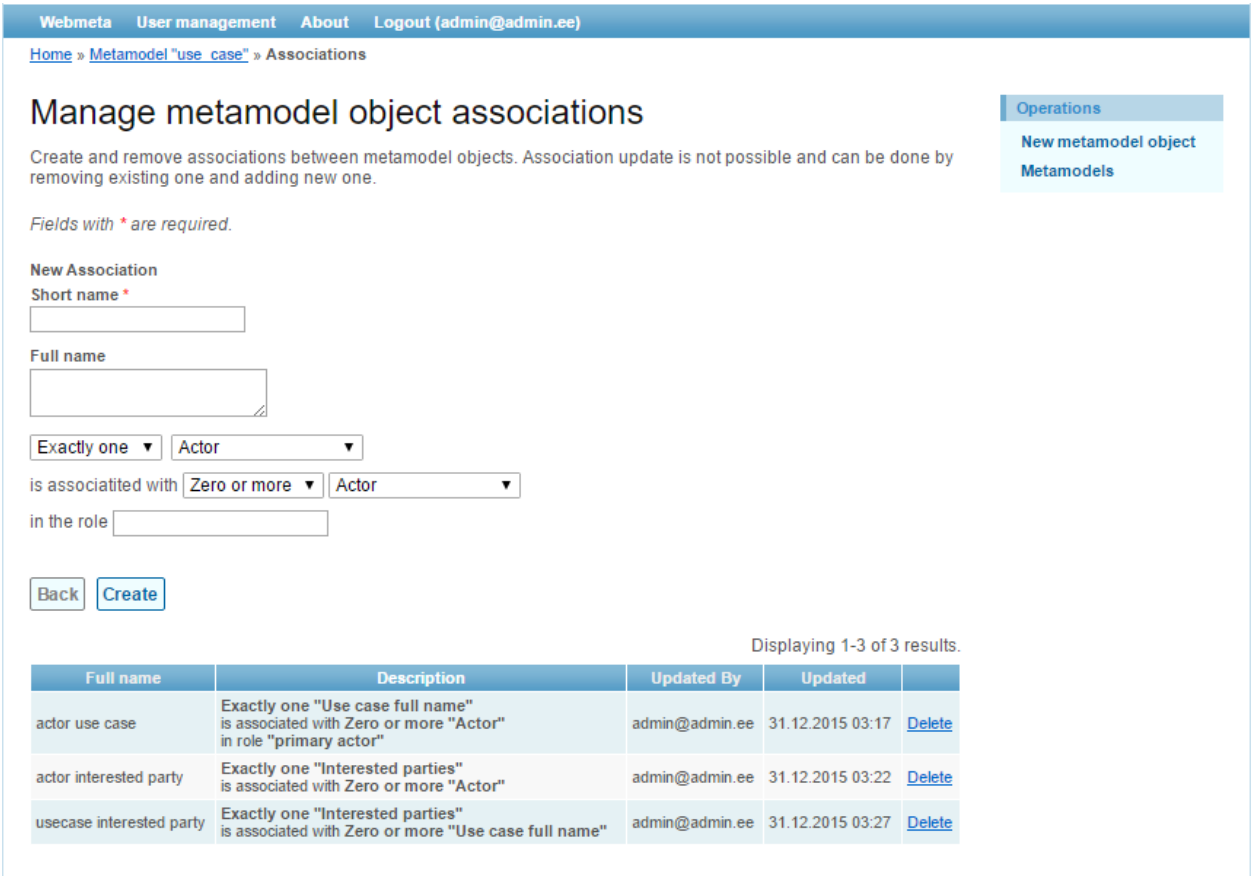
Kui väärtus on mitteaktiivne, siis mudelite haldaja näeb seda väärtust juhul kui seda on mudelis kasutatud, kuid väärtuse muutmisel ei saa seda enam uuesti valida.



Joonis 19: Klassifikaatori detailvaade

5.5.5 Metamudeli objektide vaheliste seoste haldus

Metamudeli objektide vaheliste seoste halduse vaates saab defineerida uusi seoseid ning eemaldada vanu (Vt Joonis 20). Väli „Short name“ on kohustuslik, kuna selle põhjal genereeritakse seoste välisvõtme veerud. Seejärel tuleb lausendi kujul defineerida seos võimalike objektide ning kordsustega, mille põhjal trigerid genereerivad õiged veerud, kitsendused ning välisvõtmed. Seoste võimsustike defineerimiseks tuleb moodustada lausendid, mis on mugavam ja seose kirjeldajale arusaadavam, kui tavaliselt CASE vahendites pakutav võimalus võimsustikke liitboksist valida



Joonis 20: Metamudeli objektidevaheliste seoste haldus

5.6 Trigeri näide

Järgnevas peatükis teen läbi WebMeta4 rakenduse ühe sammu, et näidata trigerite toimimist. Selle jaoks valin ühe lihtsaima protseduuridest – metamudeli lisamise.

Kasutaja poolt lisatud metamudeli lisamise andmete sisestamisel *mmm.metamodel* andmebaasi käivitub triger *mmm.f_create_metamodel_trigger* (Vt Joonis 21). Järgnevalt on lahti seletatud, mida trigeris tehakse ning toodud ka kasutatud funktsioonide kirjeldused.

```
1  BEGIN
2      PERFORM mmm.f_log_trigger('t_create_metamodel');
3      NEW.full_name := mmm.f_get_full_name( NEW.full_name, NEW.short_name);
4      NEW.short_name:=mmm.f_improve_name(NEW.short_name);
5      NEW._updated_by:=mmm.f_get_active_username();
6      PERFORM mmm.f_create_schema(NEW.short_name);
7  RETURN NEW;
8  END;
```

Joonis 21: Triger *mmm.f_create_metamodel_trigger*

Joonis 21 rida 2 juures logitakse andmebaasis trigeri väljakutse. Selle jaoks on kasutuses lihtne PostgreSQL funktsioon, mis võimaldab ühest kohast trigerite käivitumise logimist sisse-välja lülitada. Samuti registreeritakse maha trigerit välja kutsunud kasutaja nimi, mille saab tänu kasutajapõhistele andmebaasiühendustele sessioonist küsida. (Vt Joonis 22) Trigerite käivitumist on vaja logida, et oleks ülevaatlikult võimalik jälgida erinevate trigerite käivitumise järjekorda.

```
1   CREATE OR REPLACE FUNCTION mmm.f_log_trigger(
2       IN in_trigger_name TEXT
3   ) RETURNS void AS  $$
4   DECLARE
5       _logging_enabled BOOLEAN := true;
6       _created_by TEXT;
7   BEGIN
8       _created_by := mmm.f_get_active_username();
9       IF _logging_enabled THEN
10          INSERT INTO mmm.trigger_log(trigger_name, created_by)
11  VALUES(in_trigger_name, _created_by);
12      END IF;
13  END;
14  $$ LANGUAGE plpgsql SECURITY DEFINER STRICT
```

Joonis 22: Andmebaasifunktsioon *mmm.log_trigger_name*

Joonis 21 rida 3 juures kutsutakse välja funktsioon *mmm.f_get_full_name*. Selle funktsiooni eesmärgiks on tagada *full_name* väärtuse olemasolu. Kui kasutaja poolt pole sisestatud *full_name* väärtust, siis võetakse väärtus väljalt *short_name*. (Vt Joonis 23)

```
1   CREATE OR REPLACE FUNCTION mmm.f_get_full_name (
2     IN in_full_name   TEXT,
3     IN in_short_name  mmm.d_short_name,
4     OUT out_full_name mmm.d_short_name
5    ) AS $$
6   BEGIN
7       IF in_full_name = NULL OR in_full_name = '' THEN
8           out_full_name := in_short_name;
9       ELSE
10          out_full_name := in_full_name;
11      END IF;
12  END;
13  $$ LANGUAGE plpgsql SECURITY DEFINER STRICT
```

Joonis 23: Andmebaasifunktsioon *mmm.f_get_full_name*

Joonis 21 rida 4 juures kutsutakse välja funktsioon *mmm.f_improve_name*. Selle funktsiooni eesmärgiks on kohendada sisendiks tulevat nime nii palju, et see sobiks kasutamiseks andmebaasi objekti nimena. Selle juures asendatakse sisendis mitte sobivad sümbolid alakriipsudega, täpitähed asendatakse tavaliste tähtedega ning piiratakse nime pikkust 63 baidini, mis on vaikimisi PostgreSQL andmebaasi objekti nime maksimaalne pikkus. (Vt Joonis 24) Sobilikku nime pikkust saab küll muuta andmebaasi konfiguratsiooni muutes, kuid antud

rakendus näeb ette töö vaikimisi seadetega andmebaasi konfiguratsiooniga, et see saaks põhimõtteliselt edukalt töötada võimalikult suurel hulgal serveritel.

```
1  CREATE OR REPLACE FUNCTION mmm.f_improve_name(
2      IN in_name            mmm.d_full_name,
3      OUT out_improved_name mmm.d_short_name
4  ) AS $$
5  BEGIN
6    out_improved_name := substring(translate(lower(in_name),
7              'õÕäÄöÖüÜ+":$%;()=[]{}#&.!?/,''','_oOaOoOuU'),1,63);
8  END; $$ LANGUAGE plpgsql;
```

Joonis 24: Andmebaasifunktsioon *mmm.f_improve_name*

Joonis 21 rida 5 väärtustatakse *_updated_by* väli sisse loginud kasutaja nimega. Nime leidmiseks on andmebaasifunktsioon *mmm.f_get_active_username*. (Vt Joonis 25).

```
1  CREATE OR REPLACE FUNCTION mmm.f_get_active_username(
2    OUT out_username varchar(100)
3  ) AS $$
4  DECLARE
5  BEGIN
6    SELECT usename INTO out_username
7    FROM pg_stat_activity
8    WHERE state = 'active';
   END; $$ LANGUAGE plpgsql;
```

Joonis 25: Andmebaasifunktsioon *mmm.f_get_active_username*

Joonis 21 rida 6 hoolitseb lõppkasutaja andmebaasiskeemi loomise eest. Kõigepealt kontrollitakse, et ega funktsiooni parameetrina kaasa antud skeemi nimi pole keelatud nimede hulgas. Kui on, siis tekitatakse erand. Kui kõik on korrektne, siis luuakse andmebaasi skeem ning kutsutakse välja funktsioonid, mis hoolitsevad skeemi rollide loomise ning kasutaja vaikimisi tabelite loomise eest. (Vt Joonis 26)

```
1   CREATE OR REPLACE FUNCTION mmm.f_create_schema(
2     in_schema mmm.d_short_name
3   ) RETURNS void AS $$
4   DECLARE
5     _sql_stmt       TEXT;
6     _forbidden_names CONSTANT TEXT ARRAY := ARRAY ['public', 'mmm', 'mm',
7   'information_schema', 'pg_catalog', 'pg_toast_temp_1'];
8   BEGIN
9     IF (lower(in_schema) = ANY (_forbidden_names))
10    THEN
11      RAISE EXCEPTION 'Name (%) is forbidden!', in_schema;
12    END IF;
13    _sql_stmt := 'CREATE SCHEMA ' || quote_ident(in_schema);
14    EXECUTE _sql_stmt;
15    PERFORM mmm.f_create_schema_role(in_schema);
16    PERFORM mmm.f_create_user_tables(in_schema);
17
18    EXCEPTION
19    WHEN duplicate_schema
20      THEN
```

```
21          RAISE EXCEPTION 'Schema with name "%" allready exists!', in_schema;
22  END;
23
```

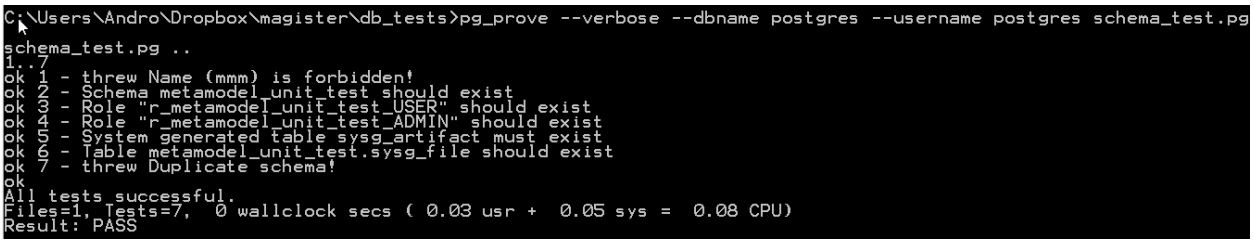
Joonis 26: Andmebaasifunktsioon *mmm.f_create_schema*

Joonis 27 kirjeldab antud trigeri testi PgTap testimisvahendis. Real 2 defineeritakse, mitu testi on plaanitud. Real 5 testitakse keelatud nimega metamudeli lisamist ning oodatavat eranditeadet, püüdes lisada metamudelit nimega ’mmm’ . Real 9 lisatakse kirje tabelisse *mmm.metamodel* ning edasi kontrollitakse ridatel 10 – 18, kas reaalselt loodi ka vastav skeem, rollid, automaatselt genereeritavad tabelid. Real 21 testitakse erandi viskamist koos vajaliku teatega, kui proovitakse lisada metamudelit, mis juba tabelis eksisteerib. Lõpuks tehakse *ROLLBACK* ehk taastatakse andmebaasis testi eele olukord.

```
1   BEGIN;
2       SELECT plan(7);
3
4       -- Test invalid metamodel name
5       SELECT throws_ok('INSERT INTO mmm.metamodel(short_name, full_name) VALUES
    (''mmm'', ''mmm'');', 'Name (mmm) is forbidden!');
6
8       -- Test creating metamodel with proper name
9       INSERT INTO mmm.metamodel(short_name, full_name) VALUES ('metamodel unit
    test', 'test');
10      SELECT has_schema( 'metamodel_unit_test' );
11
12      -- Test metamodel roles
13      SELECT has_role('r_metamodel_unit_test_USER');
14      SELECT has_role('r_metamodel_unit_test_ADMIN');
15
16      -- Test metamodel mandatory tables
17      SELECT has_table('metamodel_unit_test'::name, 'sysg_artifact' ::name, 'System
    generated table sysg_artifact must exist');
18      SELECT has_table('metamodel_unit_test'::name, 'sysg_file'::name);
19
20      -- Test creating duplicate metamodel
21      SELECT throws_ok('INSERT INTO mmm.metamodel(short_name, full_name) VALUES
    (''metamodel_unit_test'', ''metamodel_unit_test'');', 'Duplicate schema!');
22
23      -- Finish the tests and clean up.
24      SELECT * FROM finish();
    ROLLBACK;
```

Joonis 27: Metamudeli lisamise trigeri test PgTap vahendil

Joonis 28 näitab kirjeldatud testi jooksutamise ekraanitõmmist, kus on näha testide arv, iga testi kirjeldus ning lõpuks testide jooksutamiseks kulunud aeg.



Joonis 28: PgTap metamudeli trigeri testi tulemus.

6. Arendusvaade

WebMeta4 arenduse järgmises etapis on kõige olulisem kindlasti realiseerida lõppkasutajatele (modelleerijatele ning mudelite valdajatele) suunatud rakenduse osa. Kuna suur keerukus kandus andmebaasi trigerite peale, siis oleks vaja lisada testid, mis vähendaksid edasistes arendustes olemasoleva loogika katki minemise võimalust. Samuti tuleks mitme arendajaga andmebaasi arendades pöörata tähelepanu andmebaasi versioonihaldusele.

Teine oluline arendussuund oleks muuta lihtsamaks juba olemasolevate metamudelite ning objektide kustutamist ja muutmist. Hetkel on piiratud muudatuste tegemine tabelitesse, kus on juba andmeid. Edaspidi oleks oluline töötada välja toimiv süsteem, mis laseks teha võimalikult palju muudatusi ning pakuks lõppkasutajale muudetud tabelites oleva info migreerimist uude struktuuri.

Mudelite erinevate versioonide hoidmiseks tasub kaaluda ankurmodelleerimise tehnikat(Anchor Modeling, 2014). Selle kasutamise korral tekiks iga *metamudeli objekti alamobjekti* kohta metamudelile vastavasse skeemi eraldi tabel. Praegu tekib eraldi tabel iga *metamudeli objekti* kohta. Sellisel puhul kogutaks lõppkasutajale esitatav andmete tervikpilt kokku andmebaasi vaadete (*views*) kaudu. Sellisel viisil loodud tabelid pakuvad häid võimalusi skeemi evolutsioneerimiseks, andmemuudatuste ajaloo säilitamiseks ning puuduvate andmetega toimetulekuks. Saal (2015) on realiseerinud ankurmodelleerimise veebipõhisele vahendile PostgreSQL jaoks mõeldud andmebaasikeele lausete generaatori. See näitab, et PostgreSQL andmebaasis on ankurmodelleerimise alusel andmebaasi ülesehitamine täiesti mõeldav. Samas tähendaks selle võimaluse realiseerimine käesolevas töös kirjutatud trigerite ümberkirjutamist. Lisaks baastabelitele (tabelitele) peaksid need hakkama looma ka vaateid.

Pikemas perspektiivis tasuks kindlasti kaaluda ka dünaamilise graafilise kasutajaliidese kasutuselevõttu, mis võimaldaks praeguste modelleerimisrakenduste sarnast kasutajakogemust. Selle koha peal võib kasuks tulla PostgreSQL JSON andmetüüp, sest JavaScriptiga suhtlemisel liigub tänapäeval enamus andmetest just JSON kujul. Ühe sellise tööriistana jäi silma JavaScripti teek JointJS, mis võimaldab luua JavaScripti ja HTML5 abil graafilisi modelleerimislahendusi. (JointJS, 2016)

Kuigi esialgselt polnud seda plaanis, kujunes loodud süsteemist välja kiiret prototüüpimist võimaldav keskkond, kus kasutaja saab kirjeldada kontseptuaalse andmemudeli kui metamudeli ning sellest tekivad automaatselt kontseptuaalsele andmemudelile vastavad tabelid ja kitsendused. Kui oleks olemas ka lõppkasutaja vaade, siis saaks uue süsteemi kohta nõudmiste esitajad kohe süsteemi sisse logida, näeksid andmete sisestamise ning muutmise vorme ja saaksid neid proovida. Kokkuvõttes võiks see muuta nõudmiste kogumise tulemuslikumaks ja kiiremaks. Põhimõtteliselt saaks seda sama lahendust kasutada ka reaalsete andmebaaside loomiseks, mille vastu võib hakata rakendusi looma. WebMeta süsteem muutuks sellisel juhul osaks ettevõtte infosüsteemist ning seda võiks iseloomustada kui arendamise allsüsteemi.

7. Kokkuvõte

Käesoleva magistritöö eesmärgiks oli evolutsioneerida metamodelleerimisvahendi WebMeta veebi- ja andmebaasipõhisest vahendist üle SQL-andmebaasi trigeritel põhinevaks veebi- ja andmebaasipõhiseks süsteemiks.

Töö tulemusena valmis uus versioon WebMeta rakendusest koos uue arhitektuuri, kasutajaliidese kujunduse ning andmebaasisüsteemiga. Uuendamise käigus sai valmis administraatori vaade, kes haldab kasutajaid ja metamudeleid. Nende alusel genereerivad andmebaasi trigerid lõppkasutajate poolt süsteemi kasutamiseks vajalikud andmebaasiobjektid ning nende kasutamise õigused. Õiguste haldamine toimub läbi andmebaasis loodud rollide. Kasutajate tuvastamiseks kasutatakse lahendust, kus igale lõppkasutajale vastab eraldi andmebaasi kasutaja.

Töö käigus muudeti võrreldes eelmise versiooniga olulisel määral metamudelite kirjeldamiseks kasutatava keele struktuuri, võttes arvesse vahepeal sellel teemal tehtud uurimistööd. Tehniliselt kasutab loodud tarkvarasüsteem PHP programmeerimiskeelt koos Yii raamistikuga ning PostgreSQL andmebaasisüsteemi. WebMeta uue versiooniga on võimalik tutvuda aadressil http://apex.ttu.ee/WebMeta/.

Loodud rakenduse puhul sai palju rõhku pööratud andmebaasitrigerite tööle. Kindlasti vajaks need tulevikus ka ühikteste, et vähendada vigade riski edasisel arendamisel. Samuti vajab uurimist, kuidas andmebaasi trigereid oleks kõige parem modelleerida. Edasist arendust vajab esmajärjekorras lõppkasutaja (modelleerija ja mudelite vaataja) vaade.

8. Summary

The main goal of this thesis was to evolve the existing metamodelling tool WebMeta from web-based and database-based system to web-based and database-based system that extensively uses SQL database triggers.

As the result of this work, a new version of WebMeta application was created with new architecture, user interface design and database. Current result only provides environment for administrator, who manage users and metamodels. Triggers generate database schemas and other database objects for the end users based on these these as well as grant privileges for using the objects. Management of privileges takes place by using database roles. For user identification, the system uses a solution where each end user has a corresponding database user.

The language that is used for the management of metamodels is quite different in the new version and it takes into account the research in this field.

Technically the created software system utilizes PHP programming language with Yii framework and PostgreSQL Database Management System. It is possible to see the new version of WebMeta in http://apex.ttu.ee/WebMeta/.

New application uses extensively database triggers. Thus, it would be reasonable to create unit tests for these to minimize errors during development cycles. One should also investigate how to best model detaabase triggers. Next step in development would be to create a new version of end-user (modeller and model observer) application.

9. Kasutatud kirjandus

1. **Anchor Modeling.** [Võrgumaterjal] 2014. a. [Tsiteeritud: 11.12.2014] http://www.anchormodeling.com/?page_id=2.

2. **DB-Engines Ranking.** [Võrgumaterjal]. 2015. [Tsiteeritud 08.01.2015] http://db-engines.com/en/ranking

3. **Downs, K. 2009.** The Database Programmer. *A Comprehensive Database Security Model* . [Võrgumaterjal] 19.02.2009. [Tsiteeritud: 10.102014] http://database-programmer.blogspot.com/2009/02/comprehensive-database-security-model.html.

4. **Eessaar, E., Sgirka, R. 2010a.** *A Database-Based and Web-Based Meta-CASE System*. In: Advanced Techniques in Computing Sciences and Software Engineering: International Conference on Systems, Computing Sciences and Software Engineering (SCSS 08). Ed. K. Elleithy. Springer Netherlands. pp. 379-384.

5. **Eessaar, E., Sgirka, R. 2010b.** *A SQL Database Based Meta-CASE System and its Query Subsystem*. In: Innovations in Computing Sciences and Software Engineering: International Conference on Systems, Computing Sciences and Software Engineering (SCSS 09). Eds. T. Sobh, K. Elleithy. Springer Netherlands. pp. 57-62.

6. **Eessaar, E., Sgirka, R. 2013.** An Ontological Analysis of Metamodeling Languages. In: Information Systems Development Reflections, Challenges and New Directions: 20th International Conference on Information Systems Development (ISD2011) Edinburgh, Scotland, August 24-26, 2011. Eds. R, Pooley, J. Coady, C. Schneider, H. Linger, C. Barry, M. Lang. Springer New York. pp. 381-392.

7. **E-teatmink. 2015.** E-teatmik. [Võrgumaterjal] 2015. [Tsiteeritud: 05. 01 2015] http://vallaste.ee/index.htm?Type=UserId&otsing=73

8. **Fowler, M. 2009.** TwoHardThings. [Võrgumaterjal] 2009. [Tsiteeritud: 03.01.2016] http://martinfowler.com/bliki/TwoHardThings.html

9. **Guizzardi, G., Wagner, G., Guarino, N., van Sinderen, M. 2004.** An Ontologically Well-Founded Profile for UML Conceptual Models. In: Advanced Information Systems Engineering. Eds. A. Persson, J. Stirna. Springer Berlin Heidelberg. LNCS Vol. 3084. pp. 112-126.

10. **Hazlewood, L. 2014.** SQL Naming Conventions and Style Guide. *Les Hazlewood.* [Võrgumaterjal] 2014. [Tsiteeritud: 03.01.2015] http://leshazlewood.com/software-engineering/sql-style-guide/.

11. **Itagaki, T. 2014.** PostgreSQL Documentation: 9.4: auto-explain. *PostgreSQL*. [Võrgumaterjal] 2014. [Tsiteeritud: 02.01.2015] http://www.postgresql.org/docs/9.4/static/auto-explain.html.

12. **Lauk, K. 2013.** *Veebipõhiste UML CASE-vahendite võrdlus*. Bakalaureusetöö. TTÜ Informaatikainstituut.

13. **Lehman, M.M. 1996.** *Laws of Software Evolution Revisited*. Proceedings of 5th European Workshop, EWSPT '96 Nancy, France, October 9-11, 1996. LNCS Vol. 1149, Springer, Berlin, pp. 108–124.ˇ

14. **Metacase. 2015.** Metacase. *MetaEdit+ Modeler DSM Tool.* [Võrgumaterjal] 2015. [Tsiteeritud: 05.01.2015] http://www.metacase.com/mep/.

15. **PHP.net. 2014.** PHP. *PHP*. [Võrgumaterjal] 2014. [Tsiteeritud: 01.09.2014] http://php.net/.

16. **PostgreSQL. 2014.** PostgreSQL. *PostgreSQL.* [Võrgumaterjal] 2014. [Tsiteeritud: 06.05.2014] http://www.postgresql.org/.

17. **PostgreSQL. 2015.** PostgreSQL 9.5rc1 Documentation, *PostgreSQL* [Võrgumaterjal] 2015. [Tsiteeritud: 31.12.2015] http://www.postgresql.org/docs/9.5/static/ddl-rowsecurity.html

18. **RuleGen. 2015.** Architecture of RuleGen. *RuleGen.* [Võrgumaterjal] 2015. [Tsiteeritud: 01.01.2015] http://www.rulegen.com/architecture-of-rulegen.

19. **Saal, E., 2015.** Ankurmodelleerimise mudelite realisatsiooni generaator PostgreSQL jaoks. Magistritöö. TTÜ Informaatikainstituut. [Võrgumaterjal] [Tsiteeritud: 03.01.2016] http://digi.lib.ttu.ee/i/?3677 (10.12.2015)

20. **Sarkuni, S. 2014.** How I Write SQL, Part 1: Naming Conventions. [Võrgumaterjal] 16.022014. [Tsiteeritud: 10.11.2014] https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/.

21. **Sgirka, R. 2008.** Meta-CASE analüüsikeskkonna projekteerimine ja prototüübi koostamine: magistritöö. TTÜInformaatikainstituut.

22. **The Full Wiki. 2015.** MetaCASE tool. *The Full Wiki.* [Võrgumaterjal] 2015. [Tsiteeritud: 05.01.2015. a.] http://www.thefullwiki.org/MetaCASE_tool.

23. **Trosin, A. 2009**. Separation of Concern vs Single Responsibility Principle (SoC vs SRP). Artur Trosin's Blog. [Võrgumaterjal] January 26, 2009. [Tsiteeritud: 09.05.2015. a.] http://weblogs.asp.net/arturtrosin/separation-of-concern-vs-single-responsibility-principle-soc-vs-srp.

24. **Tutorialspoint. 2014.** MVC Pattern. *Tutorialspoint.* [Võrgumaterjal] 2014. [Tsiteeritud: 10.10.2014] http://www.tutorialspoint.com/design_pattern/mvc_pattern.htm.

25. **Wheeler, David E. 2014.** pgTAP. [Võrgumaterjal] 2014. [Tsiteeritud: 02.01.2015] http://pgtap.org/documentation.html.

26. **Wijk, R. 2007.** Database triggers are evil. *rwijk.blogspot.com.* [Võrgumaterjal] 13.09.2007. [Tsiteeritud: 10.102014] http://rwijk.blogspot.com/2007/09/database-triggers-are-evil.html.

27. **Wikipedia. 2014.** Object-relational mapping. *Wikipedia.* [Võrgumaterjal] 04.12. 2014. [Tsiteeritud: 12.12.2014] http://en.wikipedia.org/wiki/Object-relational_mapping.

28. **Yiiframework. 2014.** Yiiframework. *Yiiframework.* [Võrgumaterjal] 2014 [Tsiteeritud: 01.05.2014] http://www.yiiframework.com/.

29. **JointJS. 2016.** Joint JS, *Joint JS*. [Võrgumaterjal] 2016 [Tsiteeritud 01.01.2016] http://www.jointjs.com/