TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Denni Karin 222431IAIB

ANDMEMUDELI TEISENDAMINE FHIR STRUCTUREDEFINITION JA UML FORMAATIDE VAHEL

Bakalaureusetöö

Juhendaja: Igor Bossenko
PhD

Tallinn 2025

Autorideklaratsioon

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

Autor: Denni Karin

04.06.2025

Annotatsioon

Käesoleva lõputöö eesmärk on luua üldotstarbeline lahendus, mis teisendab HL7 FHIR StructureDefinition formaadis andmemudelid automaatselt UML klassidiagrammideks, säilitades seejuures FHIR spetsiifilised mehhanismid ja andmete semantilise täpsuse. Lahendus koosneb käsureapõhisest konverterist ja Java Spring Boot REST teenusest, mis kasutavad HAPI FHIR teeki profiilide parsimiseks ning PlantUML diagrammide genereerimiseks.

Olulisemateks käsitletud probleemideks on FHIR *differential* ja *snapshot* vaadete koondamine, struktuurielementide vastendamine UML klassideks ning FHIR spetsiifiliste metaandmete visuaalne esitamine ilma diagramme üle koormamata. Töös arendati ka mehhanism vastupidise teisenduse – UML mudelist FHIR profiili – prototüüpseks toetuseks, analüüsiti selle keerukust ja pakuti suundi edasiseks uurimistööks.

Tulemuseks valmis skaleeritav, Docker konteinerina juurutatav tööriist, mis on integreeritud avatud lähtekoodiga TermX platvormi. Platvormi kasutaja saab FHIR profiili redigeerimise vaates ühe nupuvajutusega luua vastava UML diagrammi, kiirendades mudelite analüüsi ja tõstes nende mõistetavust. Lahendus lihtsustab FHIR standardile vastavate andmemudelite kasutuselevõttu Eestis ja mujal ning loob lähtekoha edasiseks arenduseks, et toetada ka keerukamaid FHIR konstruktsioone.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 40 leheküljel, 5 peatükki, 15 joonist ja 2 tabelit.

Abstract

Data model transformation between FHIR StructureDefinition and UML formats

The aim of this thesis is to create a general purpose solution that automatically converts data models expressed in the HL7 FHIR StructureDefinition format into UML class diagrams while preserving FHIR specific mechanisms and the semantic accuracy of the data. The solution comprises a command-line converter and a Java Spring Boot REST service that employ the HAPI FHIR library to parse profiles and PlantUML to generate diagrams.

The main problems addressed include consolidating FHIR differential and snapshot views, mapping structural elements to UML classes, and visualising FHIR specific metadata without overloading the diagrams. The thesis also develops a prototype mechanism for the reverse transformation – from a UML model to a FHIR profile – analyses its complexity and proposes directions for further research.

The outcome is a scalable tool that can be deployed as a Docker container and integrated into the open-source TermX platform. Within the platform's profile-editing view, a user can create the corresponding UML diagram with a single click, speeding up model analysis and improving comprehensibility. The solution facilitates the adoption of FHIR conformant data models in Estonia and beyond but also provides a foundation for further development to support more complex FHIR constructs.

The thesis is written in Estonian and comprises 40 pages of text, 5 chapters, 15 figures and 2 tables.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface*, rakendusliides
CI/CD	*Continuous Integration / Continuous Delivery*, pidev integratsioon ja pidev juurutamine
FHIR	*Fast Healthcare Interoperability Resources*, tervishoiu kiire andmevahetuse ressursid
HL7	*Health Level Seven*, Tervishoiu 7. taseme standardiorganisatsioon
HTTP	*Hypertext Transfer Protocol*, hüperteksti edastusprotokoll
JSON	*JavaScript Object Notation*, JavaScripti objektide märgistuskeel
MPI	*Master Patient Index*, patsiendi põhiandmete register
OMG	*Object Management Group*, objektorienteeritud tehnoloogiate ja standardite organisatsioon
Regex	*Regular expression*, regulaaravaldis
RESTful	*Representational State Transfer*, esinduslik olekuedastus
UI	*User Interface*, kasutajaliides
URI	*Uniform Resource Identifier*, ühtne ressursiidentifikaator
XML	*Extensible Markup Language*, laiendatav märgistuskeel

Sisukord

Jooniste loetelu

Tabelite loetelu

1. Sissejuhatus

Tervishoiusektoris on tõrgeteta andmevahetus ülioluline, sest mitmed infosüsteemid peavad jagama üksikasjalikku teavet patsientide, protseduuride ja ravi kohta. Selleks kasutatakse standardeid, mis võimaldavad andmeid ühtsel viisil kirjeldada ja edastada. Üks levinumaid ja tänapäevasemaid standardeid on FHIR (*Fast Healthcare Interoperability Resources*) [1].

FHIR kirjeldab terviseandmeid ressursipõhiselt ning täpsem spetsifikatsioon toimub profiilide kaudu. Profiilid säilitatakse masinloetavas StructureDefinition formaadis, tavaliselt XML (*Extensible Markup Language*) või JSON (*JavaScript Object Notation*) kujul. UML (*Unified Modeling Language*) klassidiagrammid on samal ajal laialt kasutatav vahend süsteemide struktuuri visuaalseks esitamiseks, aidates andmemudeleid paremini mõista ja kommunikeerida [2].

Käesoleva töö keskmes on andmemudeli teisendamine FHIR StructureDefinition formaadist UML klassidiagrammi kujule. Eesmärk on luua üldotstarbeline tööriist, mis suudab automaatselt tõlgendada olemasolevat FHIR profiili ning genereerida selle põhjal UML klassid koos atribuutide ja seostega, säilitades lähteprofiili olulise struktuuri ja sisu. Erilist tähelepanu pööratakse sellele, kuidas FHIR-i spetsiifilised mehhanismid ja reeglid kajastuvad korrektselt UML diagrammil. Selline tööriist aitab parandada FHIR profiilide mõistetavust ja toetab nende paremat haldust.

Kuigi töö keskendub peamiselt andmemudeli teisendamisele FHIR-ist UML-i, käsitletakse lisaks ka võimalust teostada vastupidine teisendus, mille käigus luuakse UML mudeli põhjal FHIR StructureDefinition profiil. Seda uuritakse, et hinnata kahe mudelikeele vahelise kaardistamise üldist rakendatavust ning selgitada välja kitsaskohad, mis võivad takistada automaatset ümberkujundamist.

Väljatöötatud lahendus on Java põhine ning loodud viisil, mis ei sõltu ühest kindlast platvormist, vaid sobib kasutamiseks igas kontekstis, kus on vajadus FHIR andmemudelite visuaalse esituse järele UML klassidiagrammidena. Käesolevas töös rakendatakse ja testitakse seda teisendust TermX platvormis – avatud lähtekoodiga teadmushaldus keskkonnas, mis toetab FHIR standardit ning pakub StructureDefinition-i redaktorit ja liideseid [3].

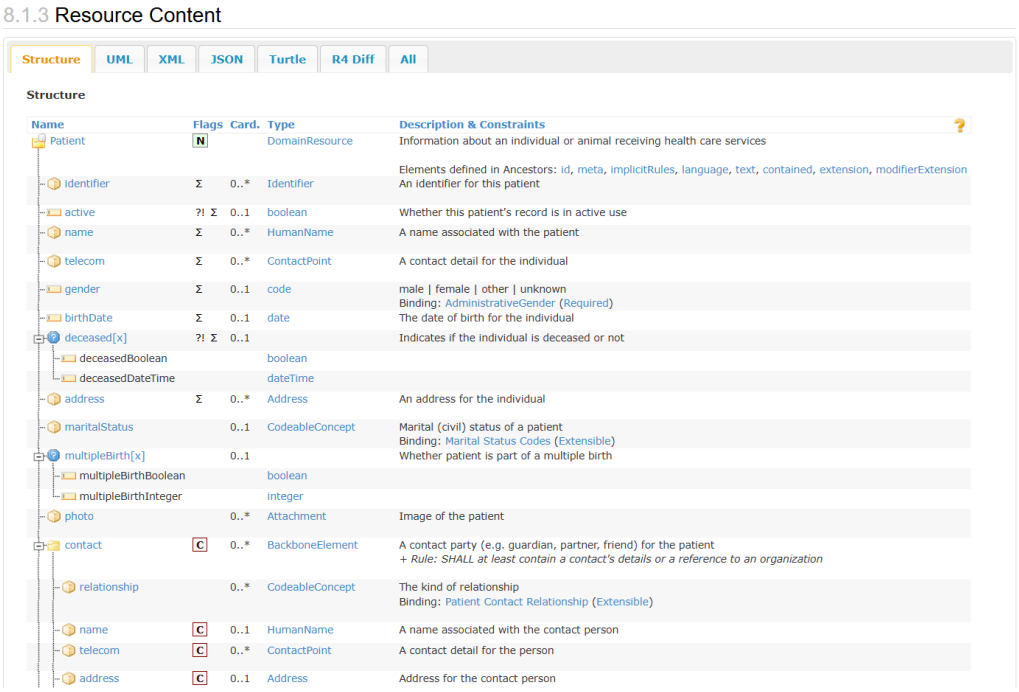
2. Taust

2.1 FHIR standard

FHIR – standard, mille on välja töötanud rahvusvaheline organisatsioon HL7 (*Health Level Seven*) [1]. See pakub ühtset ning struktureeritud viisi terviseandmete elektrooniliseks esitlemiseks ja vahetamiseks. Standard põhineb ressurssidel – modularsetel ja iseseisvatel andmeüksustel, mis modelleerivad konkreetseid tervishoiualaseid informatsioonielemente ning võimaldavad nende semantilist ja tehnilist integreeritavust erinevates süsteemides [4].

FHIR standardi Release 5 avaldati 2023. aasta märtsis ning see sisaldab kokku 157 ressurssi, mis toetavad standardiseeritud teabevahetust tervishoius. Ressursid on jaotatud erinevatesse kategooriatesse vastavalt nende kasutusotstarbele: kliinilised ressursid käsitlevad patsiendi ravi ja tervisega seotud andmeid; administratiivsed ressursid toetavad haldus- ja korraldusprotsesse; ning infrastruktuurilised ressursid on mõeldud dokumentide koostamiseks ja info edastamiseks.

Need kategooriad aitavad struktureerida ja kasutada FHIR ressursse erinevates kliinilistes ja tehnilistes kontekstides [5]. Joonis 1 illustreerib osa FHIR *Patient* resursi mudelist [6].



Joonis 1. FHIR patsiendi ressursi mudel.

Igal ressursil on standardiga määratud väljade komplekt, andmetüübid ja seosed, mis moodustavad tervikliku andmemudeli. See tagab, et kõiki ressursse esitatakse ühtsel viisil – elementide nimed, tüübid ja struktuur on standardis defineeritud [7].

2.1.1 Profiilid ja laiendused

Kuigi FHIR standard määratleb üldise andmemudeli, on tervishoiupraktikates sageli eripärasid tulenevalt riigi õigusruumist, asutuste vajadustest ja tööprotsessidest. FHIR lahendab seda paindlikkuse vajadust profiilide ja laienduste kaudu.

FHIR profiil on konkreetse kasutusjuhtumi jaoks kohandatud ressursi spetsifikatsioon – see kitsendab või laiendab mõne baasressursi definitsiooni, määrates täpsemalt, millised elemendid on nõutavad, millised väärtused lubatud ning kuidas tuleb andmeid esitada [8]. Näiteks võib profiil sätestada, et patsiendi ressursil on teatud väli kohustuslik või piirab mingi välja väärtuste hulka.

Profiilide loomise protsessi nimetatakse FHIR profiilimiseks, mis on oluline osa FHIR kasutuselevõtul kuna võimaldab standardset mudelit kohandada konkreetsetele nõuetele ilma ühilduvust murdmata [7]. Profiilid realiseeritakse FHIR standardis StructureDefinition ressursi abil [9].

2.2 StructureDefinition

StructureDefinition on HL7 FHIR standardis keskne mehhanism, millega kirjeldatakse FHIR andmemudelite struktuure. Tegemist on eraldiseisva FHIR ressursiga, mille abil on defineeritud kõik teised FHIR ressursid ja andmetüübid [10].

StructureDefinition sisaldab elementide definitsioone ning nende kasutusreegleid, võimaldades nii FHIR põhimudeleid kui ka konkreetsete kasutusjuhtude profiile vormistada ühtses masinloetavas vormis kas JSON või XML formaadis [11].

See tähendab, et nii FHIR standardi enda põhistruktuurid kui ka lokaalsetes rakendustes kohandatud profiilid on avaldatud StructureDefinition kogumikuna. Tänu sellele saab andmemudeli kirjeldusi hoida repositooriumides, võrrelda omavahel ja kasutada alusena koodi genereerimisel või näiteks kasutajaliidese loomisel.

2.2.1 Võtmeelemendid

StructureDefinition sisaldab mitmed võtmeväljad, mis kirjeldavad selle identiteeti ja omadusi [11]. Tabelis 1 on iga võtmeväli esitatud koos lühikirjeldusega, et selgitada nende tähendust ja rolli StructureDefinition ressursis.

Tabel 1. FHIR StructureDefinition ressursi võtmeväljad koos lühikirjeldusega.

Väli	**Kirjeldus**
url	Globaalselt unikaalne kanooniline URI (*Uniform Resource Identifier*), millega määratletakse konkreetne struktuur; püsib samana sõltumata serverist.
version	Autori hallatav versiooninumber, mis eristab konkreetset väljaannet või muudatust.
identifier	Täiendav formaalne identifikaator, mida saab kasutada definitsioonile viitamiseks teistes süsteemides või formaatides. Erineb ressursi lokaalsest *id*, toimides pigem välise viitena.
kind	Kirjeldab struktuuri määratluse liiki ehk millist tüüpi struktuuri antud ressursi kaudu defineeritakse (nt ressurss, komplekstüüp, primitiivtüüp või loogiline mudel).
type	Määrab konkreetse FHIR tüübi või ressursi nime, mida antud StructureDefinition defineerib või kitsendab; sisuliselt viitab see andmetüübile või ressursile, mille struktuur on definitsioonis kirjeldatud.
abstract	Tõeväärtusega väli, mis näitab, kas antud struktuur on abstraktne ehk mitte vahetult instantsieeritav. Neid ei kasutata otseselt andmevahetuses.
baseDefinition	Annab kanoonilise viite (URI) aluseks olevale struktuuri definitsioonile, millest käesolev definitsioon on tuletatud kas eristamise (*specialization*) või kitsendamise (*constraint*) teel.
title	Lühike kasutajasõbralik pealkiri, mis selgitab antud struktuuri olemust.
name	Masinloetav nimi, mida kasutatakse struktuurimääratluse identifitseerimiseks arvutisüsteemides. Tavaliselt lihtne ja ilma tühikuteta, sobib automaatseks töötlemiseks.
id	Ressursi lokaalne loogiline identifikaator, mida kasutatakse FHIR serveris selle ressursi URL viitamiseks ning mis pärast määramist enam ei muutu.

Lisas 2 on lühendatud näide *EEBasePatient* ressursi profiili StructureDefinition määratlusest, mis illustreerib võtmeväljade esinemist ja struktuuri FHIR masinloetavas JSON vormingus Eesti terviseandmete vahetuse kontekstis [12] kitsendades tavalist FHIR

Patient ressurssi [6]. Näites on esitatud ainult olulisemad väljad, et anda ülevaade profiili tuumstruktuurist.

2.2.2 Tüübid ja nende omadused

StructureDefinition ressursi abil saab FHIR-is kirjeldada erinevat liiki struktuure *kind* välja abil. Peamised kasutusliigid on järgmised:

1. **Andmetüüp (*complex-type* või *primitive-type*)**
   FHIR eristab kahte tüüpi andmetüüpe – primitiivsed ja komplekssed. Mõlemad on realiseeritud StructureDefinition ressursina, kus atribuudi *kind* väärtus on vastavalt [13]:
   - *primitive-type* – andmetüüp, mille väärtus on esitatav lihttekstina ja mis järgib määratud vormingut (näiteks *string*, *boolean*, *dateTime*).
   - *complex-type* – andmetüüp, mis koosneb alamväljadest, millede tüübid võivad olla kas primitiivsed või teised komplekssed tüübid (näiteks *Address.line*, *Address.city*).
2. **Ressurss (*resource*)**
   Kui StructureDefinition ressursi *kind* on *resource*, siis see tähistab kas FHIR standardset baasressurssi või selle alusel loodud kohandatud ressursiprofiili. Ressursside struktuur määratleb, milliseid andmetüüpe võib igas välja puhul kasutada, tagades andmemudeli ühtsuse ja valideeritavuse kogu FHIR ulatuses. Kui profiilis täpsustatakse või kitsendatakse mõne andmetüübi kasutust, siis tehakse seda eraldi StructureDefinition ressursi kaudu, mis baseerub vastaval standardtüübil [14].
3. **Loogiline mudel (*logical*)**
   Lisaks võimaldab FHIR määratleda loogilisi mudeleid, mis on abstraktsed struktuurid ning ei vasta otseselt ühelegi konkreetsele FHIR ressursile. Need on kavandatud eelkõige kontseptuaalsete või analüütiliste andmemudelite esitamiseks [15].

2.2.3 Mehhanismid andmemudelite kirjeldamiseks

FHIR StructureDefinition ressursi sees rakendatakse andmemudelite kirjeldamisel nelja olulist mehhanismi, mis koos võimaldavad väga paindlikult kehtestada nii elementide olemust, reegleid kui ka laiendusi:

1. **ElementDefinition**
   Iga StructureDefinition koosneb ElementDefinition üksustest, milles määratakse detailselt konkreetse ressursi või andmetüübi elementide omadused [16].
2. **Piirangud (*constraints*)**
   Constraints võimaldavad määratleda loogilisi tingimusi, mis kehtivad ühe või mitme välja kooskasutuse puhul. Sellised tingimused võimaldavad modelleerida ärireegleid, mis ei ole väljendatavad üksnes kardinaalsuse või tüübi tasandil.
3. **Viilud (*slicing*)**
   Slicing on mehhanism, millega saab korduvalt esinevaid elemente täpsustada nende väärtuste alusel [9].
4. **Laiendus (*extensions*)**
   Laienduste (*extensions*) kaudu on võimalik olemasolevatele ressurssidele või andmetüüpidele lisada täiendavaid andmevälju, mida standardne mudel algselt ei hõlma. Iga laiendus on realiseeritud StructureDefinition ressursina ning selle tüübiks on tavapäraselt *complex-type*, kuna tegemist on struktuuriliselt keeruka andmeelemendiga, mis baseerub üldisel *Extension* tüübimallil. Laienduse spetsifikatsioon määratleb *tag* (*uri*) / *value* paari, kirjeldades seejuures, millises kontekstis (näiteks konkreetse ressursi või andmetüübi raames) laiendust tohib kasutada ning millist tüüpi väärtusi see võib sisaldada [10].

2.2.4 ElementDefinition ressursi võtmeväljad

ElementDefinition on HL7 FHIR standardi keskne mehhanism, millega täpsustatakse üksikute elementide omadusi nii FHIR ressursi, komplekstüübi kui ka primitiivtüübi tasandil. ElementDefinition on StructureDefinition ressursi üks põhilisi komponente, võimaldades hallata FHIR profiilide sisu masinloetaval viisil [16]. Tabel 2 kujutab endast kahe veeruga kokkuvõtet, kus on esitatud ElementDefinition ressursi võtmeväljad koos nende lühikirjeldusega.

Tabel 2. ElementDefinition ressursi võtmeväljad koos lühikirjeldusega.

Väli	**Kirjeldus**
path	Määrab elemendi hierarhilise asukoha FHIR struktuuris. Näitab, kuidas elemendid on pesastatud ja aitab määrata, millised reeglid või piirangud konkreetsele elemendile kehtivad.
id	Elemendi lokaalne tuvastaja, mida saab kasutada laienduste või muude metaandmete omavaheliseks viitamiseks.

Jätkub järgmisel lehel

Tabel 2 – *jätkub*

Väli	**Kirjeldus**
min, *max*	Kardinaalsust määravad väljad, mis näitavad, mitu korda element võib või peab esinema. Näiteks `min=0`, `max=1` tähendab, et element on valikuline ja ühekordne; `max="*"` lubab piiramatuid kordusi.
type	Määratleb, millist tüüpi andmeid element võib sisaldada – primitiivseid, komplekseid või teisi FHIR ressursse. Profiilides saab seda kitsendada konkreetsetele tüüpidele.
binding	Kasutatakse terminoloogiliste piirangute kehtestamiseks. Väli määrab, millist kindlat väärtuste loendit ehk *ValueSet* tohib selle välja puhul kasutada ja määrab sidumise tugevuse (näiteks *required*, *extensible*).
constraint	Sisaldab formaalseid reegleid (näiteks *FHIRPath* avaldised), mis peavad andmete kohta tõesed olema. Lubab väljendada keerukamaid tingimusi kui tüübid või kardinaalsus.
fixed[x], *pattern[x]*	Määravad kas fikseeritud väärtuse või mustri, millele elemendi väärtus peab vastama.
mustSupport	*Boolean* väli, mis tähistab, et antud elemendi käsitlemine on oluline FHIR-i põhiste rakenduste jaoks. Element ei pruugi olla kohustuslik, kuid süsteem peab suutma seda lugeda või edastada.
isModifier	Tähistab, et elemendi väärtus võib mõjutada teiste väljade tähendust viisil, mida ei tohi ignoreerida. Näiteks staatuseväli, mis muudab kogu andmeobjekti semantikat.
slicing	Mehhanism, mis lubab jagada korduvaid või mitut tüüpi elemente viiludeks, kus igale viilule kehtivad eraldi kitsendused. Kasulik keerukamate andmestruktuuride modelleerimisel.

Lisas 3 on lühendatud näide *RelatedPerson* ressursi profiili ühe elemendi ElementDefinition määratlusest, mis illustreerib võtmeväljade kasutust ja ülesehitust FHIR masinloetavas JSON vormingus [17].

2.2.5 Differential- ja snapshot vaated

StructureDefinition ressursis esitatakse ElementDefinition kogum kahes erinevas vormis: *differential* ja *snapshot*. *Differential* kirjeldab ainult erinevusi võrreldes aluseks oleva struktuuriga – see tähendab loetletakse need elemendid ja väljad, mida profiil muudab või lisab baasdefinitsioonile. *Snapshot* seevastu esitab täieliku, iseseisva loendi kõigist elementidest ja nende piirangutest, justkui rakendades *differential* muudatused baasstruktuurile ja tuues tulemuse välja terviklikul kujul [18].

Need kaks esitusviisi eksisteerivad erinevatel eesmärkidel: *differential* on kompaktne ja inimloetav profiili definitsioon (sobilik redigeerimiseks ja muutuste jälgimiseks), samas kui *snapshot* on operatiivseks kasutamiseks – sisaldades kogu informatsiooni ühes kohas, nii et valideerijad ja teised tööriistad ei pea baasprofiili eraldi läbi töötama. Praktikas genereeritakse *snapshot* automaatselt, lähtudes *differential*-st ja viidatud baasstruktuuridest, tagades et kõik pärandatud elemendid ja reeglid on lõppvaates olemas [9].

ElementDefinition üksikkirjed esinevad seega mõlemas vaates. *Differential*-is on tavaliselt iga muudetud elemendi kohta üks ElementDefinition, mis sisaldab ainult muudetud atribuute (näiteks uuendatud *min*, kitsendatud *type*), jättes määramata need väljad, mida ei muudeta (need päranduvad baasist). *Snapshot*-is on aga iga elemendi kohta ElementDefinition kõikide rakendunud väärtustega, nii baasist pärandatud kui profiilis uuendatud omadustega. Mõlemad mehhanismid on vajalikud – üks hõlbustab profiili defineerimist ja teine profiili rakendamist [9].

2.3 UML-i kasutamine andmemudelite visualiseerimisel

UML on objektorienteeritud süsteemide modelleerimise standard, mida haldab OMG (*Object Management Group*). Tegemist on laialt kasutatava keele ja notatsioonide kogumikuga, mis võimaldab süsteemi struktuuri ja käitumist visuaalselt kirjeldada [2].

Selle sobivus terviseandmete modelleerimisel tuleneb eelkõige selle ühtsest semantikast ja väljendusjõust – see pakub standardset viisi kirjeldada terviseinfosüsteemide olemeid ja nendevahelisi seoseid [19]. UML-i tugevuseks terviseandmete kontekstis on ka selle laiendatavus ja toetatud tööriistad.

Kaasaegses HL7 FHIR standardis andmetüübid ja ressursid on esitatavad UML klassidena ja seostena: iga FHIR tüüp on UML klass, millel on nimi, võimalik esiklassi seos ning atribuute koos nime, andmetüübi ja kardinaalsusega [20].

2.3.1 UML genereerimise tööriistad JSON põhised lahendused

Tänapäeval on saadaval mitmeid vabavaralisi tööriistu, mis toetavad UML diagrammide loomist ja mõnel juhul ka automaatset genereerimist JSON põhjal. Alljärgnevalt antakse ülevaade mõnest olulisemast vahendist – JSON2UML [21] ja jsonDiscoverer [22] – ning hinnatakse nende sobivust FHIR StructureDefinitioni andmemudelite visualiseerimiseks.

JSON2UML viitab tööriistade klassile, mis suudavad JSON formaadis andmekirjelduse

põhjal luua UML klassidiagrammi. Üldpõhimõte seisneb selles, et JSON elementidest tehakse UML klassid koos atribuutsuste ja alamklassi suhetega. JSON2UML tüüpi lahenduste eelis on automaatika – diagrammide koostamine toimub kiirelt, ilma et peaks käsitsi iga klassi looma [21].

Siiski on nendel ka piiranguid. Need vahendid on üldotstarbelised ning eeldavad üsna lihtstruktuurilist sisendit. Komplekssete standardite nagu FHIR puhul võib JSON2UML tulemus olla puudulik. Näiteks FHIR StructureDefinition on küll esitatav JSON formaadis, kuid sisaldab palju metaandmeid nagu *slicing*, laienduste deklaratsioonid, mida üldised JSON-i importijad arvesse ei võta. Seetõttu genereeritakse JSON2UML abil küll põhiklasside struktuur, kuid FHIR erikonstruktsioonid võivad jääda kajastamata.

jsonDiscoverer on tööriist, mis loob JSON andmete põhjal automaatselt UML klassidiagrammi. Põhimehhanism seisneb JSON objektide tuvastamises ning nende kaardistamises klassideks koos atribuutide ja võimalike seostega. Nagu ka teised *JSON to UML* lahendused, pakub jsonDiscoverer olulist ajavõitu, sest kasutaja ei pea igat klassi käsitsi defineerima – tööriist tuvastab struktuuri automaatselt [22].

Samal ajal pole see lahendus loodud keerukamate standardite, näiteks FHIR profiilide visuaalseks esitamiseks. Metaandmed jäävad jsonDiscoverer tavapärase kasutuse korral tähelepanuta. Tulemuseks on küll UML diagramm põhivormingu tasandil, kuid detailne FHIR spetsiifikaga mudel olulisel määral jääb puudulikuks.

2.3.2 UML genereerimise tööriistad XML põhised lahendused

Lisaks JSON variandile avaldab FHIR ka oma StructureDefinition ressursid XML formaadis, mis on paljudele modelleerimisvahenditele tuttav esitusviis. Turul leidub mitmeid tööriistu – näiteks Magicdrawc [23], EnterpriseArchitect [24], Modelio [25] laiendused, mis suudavad XSD või muust XML skeemist automaatselt UML diagrammi luua. Üldpõhimõte seisneb XML elementide ja nende tüüpide kaardistamises UML klassideks, kusjuures keerukamad elemendid muutuvad üldjuhul kompositsioonideks või alamklassideks.

Paraku on ka selliste lahenduste sobivus FHIR profiilide kirjeldamiseks piiratud, kuna terviseandmete standard kasutab mitmeid erimehhanisme, mida standardne *XML to UML* teisendusprogramm enamasti ei käsitle. Puudulik on tugi FHIR-ile omaste funktsioonide osas nagu näiteks *slicing*, laienduste semantiline tõlgendamine, terminoloogiate sidumine ning väliste reeglite ja piirangute rakendamine.

Eeltoodud põhjustel ei anna tavaliste vahendite otsene rakendamine FHIR profiilide UML diagrammide genereerimiseks rahuldavat tulemust JSON kui ka XML põhjal. Nende tööriistade piirangud tähendavad, et ilma kohandusteta jääks tulemus pealiskaudseks. Seega on tarvis lahendust, mis võimaldab paindlikult programmeerida FHIR StructureDefinitioni tõlgendust UML kujule.

2.3.3 Diagrammide koostamine

UML diagramme saab koostada mitmel erineval viisil, ulatudes täisfunktsionaalsetest visuaalsetest modelleerimistööriistadest (nagu Enterprise Architect [24], StarUML [26], Visual Paradigm [27]) kuni kergemate veebipõhiste lahendusteni (näiteks draw.io [28]) ja koodipõhiste vahenditeni (nagu PlantUML [29], Mermaid [30]). Graafilised rakendused on mugavad käsitsi skeemide joonistamiseks ja kiireks visuaalseks muutmiseks, aga need nõuavad sageli spetsiifilisi failivorminguid ning võivad jääda piiratud, kui on vaja teha ulatuslikke automatiseeritud teisendusi või integreerida diagrammide loomist erinevatesse protsessidesse.

Kergemate veebipõhiste lahenduste (näiteks draw.io) eelis on nende kättesaadavus – diagrammide loomine toimub brauseris, mistõttu pole vaja installida eraldi töölauarakendust [28]. Need sobivad kiireks meeskonnatööks, kus mitu inimest saavad samaaegselt skeeme redigeerida. Samas ei ole veebipõhised tööriistad alati mõeldud tihedaks automatiseerimiseks, sest nende põhirõhk on visuaalsel koostamisel.

Koodipõhised tööriistad, nagu PlantUML ja Mermaid, tööriistad lahendavad mitmeid eelmainitud kitsaskohti, sest diagrammid kirjeldatakse tekstina, mida on võimalik versioonihalduses käsitleda täpselt nagu koodi [31]. Selle tulemusena saab FHIR StructureDefinition ressursse automaatselt parsida ja vajaduse korral programmiliselt rikastada täiendavate detailidega, mis UML diagrammis olulised on.

2.4 TermX platvorm

TermX on avatud lähtekoodiga platvorm, mis on loodud tervishoiuvaldkonna teadmushalduseks ning semantiliseks integreerituseks. TermX hõlmab mitmeid omavahel integreeritud mooduleid, sealhulgas terminoloogia serverit, *wiki* süsteemi, andmemudelite disainerit ning andmemudelite teisenduste redaktorit [3].

Platvormi eesmärk on toetada terviseandmete sujuvat andmevahetust pakkudes kasutajale mugavaid graafilisi tööriistu nii terminoloogiate haldamiseks kui ka andmemudelite

defineerimiseks. TermX on loodud lähtudes tervishoiu andmevahetus standardist FHIR, et lihtsustada andmestandardite rakendamist ning parandada andmekvaliteeti [3].

2.4.1 Arhitektuur

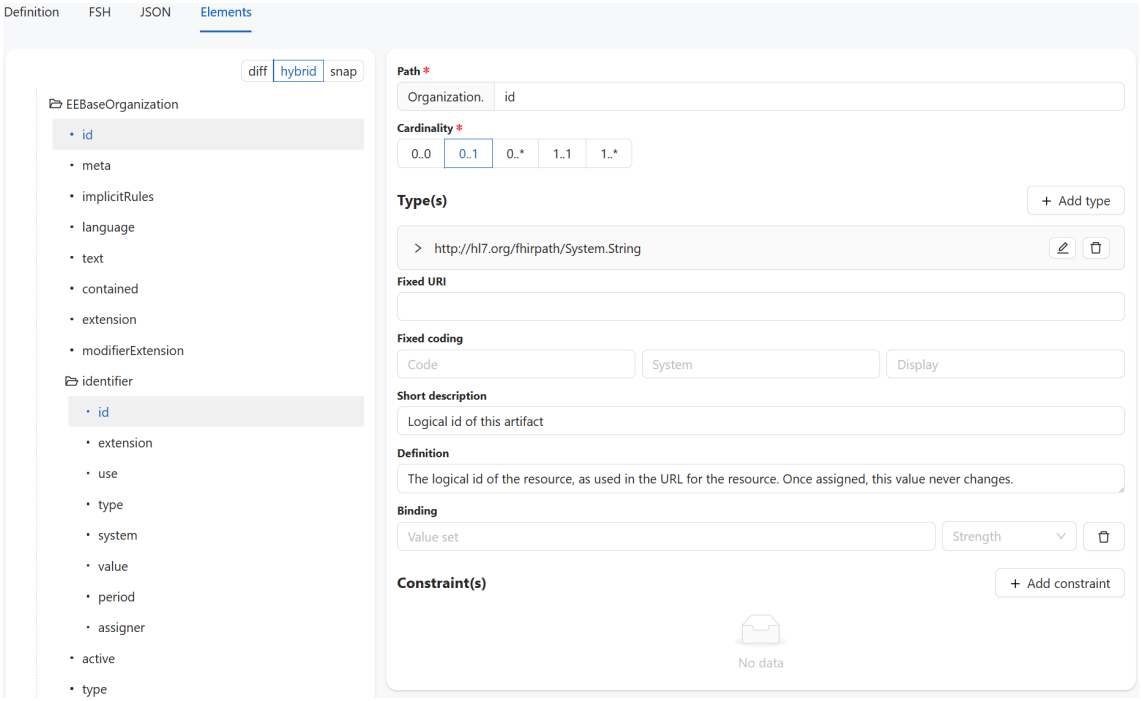
TermX arhitektuur on üles ehitatud moodulipõhiselt, koosnedes veebirakendusest ja taustaserverist. Veebirakendus on realiseeritud Angular raamistikus ning pakub kasutajale brauseris interaktiivset kasutajaliidest erinevate ülesannete täitmiseks [32]. Tagaplaanil töötab Java põhine serverirakendus, mis haldab andmeid ja äriloogikat ning suhtleb veebirakendusega RESTful (*Representational State Transfer*) API (*Application Programming Interface*) vahendusel. Andmete püsimälu jaoks kasutab TermX PostgreSQL andmebaasi. Platvorm toetab ühe sisselogimise lahendust OpenID Connect protokolli abil [33].

2.4.2 Praegune FHIR tugi

TermX on ehitatud nii, et ta peegeldab FHIR standardi mõisteid ja mehhanisme oma funktsionaalsuses. Platvormi sisemine andmemudel kasutab laialdaselt FHIR ressursse ja komponente, tagades, et TermX-st väljastatud artefaktid on standardipõhised. Eriti tugev on TermX tugi FHIR StructureDefinition ressurssidele, kuna just nende abil hallatakse kõiki TermX-is loodud andmemudeleid. TermX-i mudelidisainer on otseselt üles ehitatud StructureDefinition spetsifikatsioonile – see tähendab, et iga loomisel olev mudel järgib FHIR-i reegleid. Platvorm kontrollib sisemiselt, et mudeli redigeerimisel ei rikutaks kehtivaid reegleid, pakkudes seega valideeritud profiilide koostamist [33].

Mudelidisaineri abil saab kasutaja graafilise liidese kaudu luua ja hallata infomudeleid – olgu need uued loogilised andmemudelid, FHIR ressursipõhised profiilid või FHIR laiendused. TermX toetab mudelite importi ja eksporti nii FHIR JSON formaadis kui ka FHIR Shorthand vormingus. Kasutajamugavuse huvides esitab TermX mudelid visuaalselt puustruktuurina, mida saab interaktiivselt muuta, ilma et peaks käsitsi JSON koodi redigeerima [33].

Joonisel 2 on näidatud TermX-i mudelidisaineri kasutajaliides, kus kuvatakse *EEBaseOrganization* profiili [34] redigeerimine.



Joonis 2. *EEBaseOrganization* profiili visuaalne redigeerimine TermX-is.

2.4.3 FHIR StructureDefinition ja UML vahelise teisenduse vajadus

Praegu puudub võimalus genereerida UML klassidiagramme otse FHIR StructureDefinition kirjelduste põhjal, kuigi selline funktsionaalsus oleks väga kasulik. Visuaalne esitus muudaks FHIR profiilide struktuuri paremini mõistetavaks ning toetaks nende tõhusamat analüüsi, kasutamist ja dokumenteerimist. Lisaks aitaks automatiseeritud diagrammide loomine oluliselt säästa aega võrreldes käsitsi modelleerimisega.

TermX on FHIR-ist UML-i skripti rakendamiseks sobivaim keskkond, kuna tegemist on FHIR-i toetava avatud lähtekoodiga platvormiga, mille arhitektuur võimaldab arendajatel luua kohandatud töövooge ja laiendusi. Skripti saab integreerida TermX-i StructureDefinition redaktoriga, pakkudes kasutajale võimalust genereerida UML diagramm otse mudeliredaktori vaates.

Lisaks peamisele fookusele võimaldab skripti arendusprotsess ka uurida vastupidist suunda – kuidas oleks võimalik UML mudelist luua FHIR profiil. See aitab paremini mõista, kuidas FHIR struktuurielemente saab kaardistada UML mõisteteks ning millised struktuursed erinevused ja tõlgendusprobleemid võivad selliste teisenduste käigus tekkida.

3. Teostus

3.1 Skripti lähtekoht

Arenduse algfaasis kaaluti, kuidas FHIR StructureDefinition andmeid tõhusalt töödelda. Kuna FHIR standard on mahukas ja kompleksne, pole otstarbekas kirjutada päris oma parserit nullist – see oleks aeganõudev, vigadele aldis ning dubleeriks suuresti juba olemasolevaid lahendusi [35]. Selle asemel otsustati tugineda valmis teekidele, mis on loodud FHIR andmemudelite töötlemiseks. Nii saab keskenduda äriloogikale ning olla kindel, et FHIR ressursi JSON/XML struktuuri lugemine ja valideerimine toimub korrektselt vastavalt standardile.

3.1.1 FHIR teekide olemasolu

Vaadeldi mitmeid FHIR töötlemiseks mõeldud lahendusi, kuid põhitähelepanu pöörati Pythonis ja Javas loodud teekidele, kuna autoril on varasem kogemus just nende programmeerimiskeeltega.

Python ökosüsteemis on saadaval teek nimega fhir.resources, mis pakub kergekaalulist FHIR standardi implementatsiooni Pythoni keeles [36]. Tegemist on teegiga, mis sisaldab klasse kõikidele FHIR ressursitüüpidele ja võimaldab luua, valideerida ning serialiseerida (JSON/XML) FHIR objekte Python-is [37]. fhir.resources on loodud silmas pidades lihtsalt kasutatavat skriptimist ning integreerub mugavalt Python-i veebiraamistikega. See sobib hästi automatiseeritud andmetöötluseks, näiteks tervishoiu töövoogude automatiseerimiseks. Samuti on fhir.resources aktiivselt arendatav ning vabalt kasutatav.

Java maailmas on standardiks kujunenud HAPI FHIR teek. HAPI FHIR on üks populaarsemaid avatud lähtekoodiga FHIR implementatsioone, mis on kirjutatud Java keeles [38]. See teek on väga terviklik – HAPI sisaldab nii kliendi- kui serveripoolseid komponente, pakkudes välja kõiki FHIR ressursitüüpe ning operatsioone. HAPI FHIR on kirjutatud puhtas Javas ja on Apache 2.0 litsentsiga vabavaraline projekt, mida arendab aktiivne avatud kogukond. Teegi funktsionaalsus on laiaulatuslik – lisaks ressursside parsimisele/seriaalsusele sisaldab see ka valideerimist, otsingut ning profiilide rakendamist [39].

3.1.2 Valiku põhjendus HAPI FHIR kasuks

Antud töö eesmärkide saavutamiseks osutus sobivaimaks lahenduseks Java põhine HAPI FHIR teek. Otsus langes sellele teegile mitme teguri koosmõjul, millest olulisemad olid selle ühilduvus olemasoleva süsteemiarhitektuuriga, ulatuslik funktsionaalsus, stabiilsus ning arenduskeskkonna toetus.

HAPI FHIR on väljakujunenud ja laialdase kasutajaskonnaga avatud lähtekoodiga projekt, mille arendust veab aktiivne rahvusvaheline kogukond. Teek järgib HL7 FHIR standardi ametlikke spetsifikatsioone ning selle pidev versiooniuuendus [40] tagab kooskõla ka uusimate FHIR versioonidega, sealhulgas R4 ja R5. Selle küpsus ja laialdane kasutus tervishoiusüsteemides annavad kindlustunde, et teek suudab töökindlalt käsitleda ka keerukamaid FHIR ressursse.

TermX platvormi tehnoloogiline alus – Java põhine serveriarhitektuur [32] – soodustab HAPI FHIR kasutuselevõttu, võimaldades selle integreerida otse olemasolevasse rakenduskeskkonda. Selline lähenemine aitab vältida täiendava mikroteenuse loomist, mis oleks Python põhise lahenduse puhul vajalik ning tooks kaasa lisakoormuse konteinerite, andmeside ja hooldusprotsesside haldamisel. Ühtlasi vähendab see süsteemi heterogeensust ja suurendab hooldusmugavust.

Teegi valikut toetas ka arendaja varasem kogemus Java-ga, mis võimaldas keskenduda arenduse sisulisele poolele ilma vajaduseta investeerida aega uute tööriistade või keeltega süvitsi tutvumiseks. See aitas kiirendada arendusprotsessi ning vähendas võimalike tehniliste probleemide riski.

Lisaks tehnilistele ja praktilistele kaalutlustele mängis rolli ka HAPI FHIR ümber kujunenud kogukond. Teek on põhjalikult dokumenteeritud ning saadaval on mitmeid tugikanaleid ja näiteid, mis hõlbustasid arendusprotsessi. Arvestades FHIR standardi keerukust ja pidevat arengut, on selline kogukonnatugi oluline tegur, mis tagab projekti pikaajalise jätkusuutlikkuse.

3.1.3 REST API loomise vajadus

Kuigi teisendusskript oleks võinud jääda ka ühekorra kasutatavaks käsurea tööriistaks, tekkis vajadus integreerida see dünaamilisemalt TermX platvormi. Et arendatud skript saaks olla selle platvormi osa, autor otsustas see pakkida lisaks RESTful veebiteenusena. REST API vorm võimaldab TermX platvormil kutsuda skripti funktsionaalsust standardse

HTTP (*Hypertext Transfer Protocol*) liidese kaudu, mis on tehnoloogianeutraalne ja laialt toetatud integratsioonimeetod.

TermX on veebipõhine platvorm, mille kasutajaliides suhtleb taustasüsteemidega HTTP API-de vahendusel [32]. Teisendusskripti kui REST API loomine tähendab, et näiteks TermX veebiliides saab päringu teel saata StructureDefinition ressursi skriptile ja vastu võtta genereeritud UML mudeli. See võimaldab reaalajas UML diagrammi koostamist, kui platvormi kasutaja seda vajab. Ilma REST API-ta peaks skripti käivitama manuaalselt või poolautomaatselt, mis pole platvormi kontekstis kasutajasõbralik ega skaleeru mitme kasutaja vajadustele.

3.1.4 Spring Boot kasutuselevõtt teenuse arendamisel

Arvestades eelnevat otsust kasutada HAPI FHIR Java teeki ja vajadust luua iseseisev REST teenus, oli loomulik valik ehitada see teenus Java baasil, kasutades Spring Boot raamistikku. Spring Boot on laialdaselt kasutatav raamistik mikroteenuste ja veebirakenduste kiireks loomiseks – see pakub sisseehitatud rakendusserverit, REST API loomise lihtsustust ja hulgaliselt valmis integraatoreid [41]. Kuna TermX serverikomponendid on Java põhised, tagab sama *stack*-i jätkamine parima ühilduvuse. Lisaks projekti autoril on varasem tugev kogemus Spring Booti-ga, mistõttu on teenuse arendamine selles keskkonnas kiirem ja kindlam.

3.1.5 Versioonihaldus, CI/CD ja integreerimine TermX töövoogu

Arendatud skripti ja veebiteenuse haldamiseks valiti versioonihaldusplatvormiks GitHub, mis võimaldab teha koostööd avatud lähtekoodiga projektina. Projekti hoidla on avalik, mis tähendab, et skript on avatud lähtekoodiga ning kättesaadav nii TermX tiimile kui ka laiemale huviliste ringile. See on kooskõlas TermX platvormi põhimõtetega – TermX komponentide kood on samuti avalikult saadaval GitHubis ja konteineritena juurutatav. Avatud lähtekoodi eelis on ka see, et tulevikus saab vajadusel teised arendajad skripti täiendada või kasutajad esitada probleemraporteid ning ettepanekuid, parandades seeläbi lahenduse laiendatavust ja kvaliteeti.

Et tagada stabiilne ja järjepidev arendusprotsess, autor otsustas integreerida projekti töövoogu ka CI/CD (*Continuous Integration / Continuous Delivery*) mehhanism. Selleks kasutatakse GitHub Actions töövooge, mis võimaldavad automaatselt testida ja ehitada projekti iga koodi muudatuse korral.

Docker on tarkvaraplatvorm, mis võimaldab rakendusi pakendada konteineritesse – kergetesse ja isoleeritud käituskeskkondadesse. Konteiner sisaldab kogu vajaliku tarkvarapaki, sealhulgas sõltuvused ja konfiguratsioonid, mistõttu saab rakendust täpselt samal kujul käivitada erinevates keskkondades (arendus, test, tootmine) [42].

TermX platvorm kasutab Docker konteineripõhist arhitektuuri, mis teeb Docker-i kasutamise teenuse juurutamiseks eriti sobivaks. Docker võimaldab ka lihtsat versioonihaldust ja testimist.

Kui uus Docker-i tõmmis on edukalt loodud, laetakse see üles konteineriregistrisse (GitHub Container Registry), kust see on valmis kasutamiseks TermX platvormil. Juurutusprotsess toimub vastavalt TermX-i töövoole, kus esmalt testitakse teenust arenduskeskkonnas, seejärel liidestatakse see UI-ga (*User interface*) ning lõpuks juurutatakse vajadusel produktsioonikeskkonda.

3.1.6 PlantUML kasutuselevõtt

PlantUML on koodipõhine UML diagrammide genereerimise lahendus, mis kasutab skeemide kirjeldamiseks tekstisüntaksit, võimaldades diagramme luua ja muuta täpselt nagu programmikoodi [43]. See lähenemine on eriti kasulik valdkondades, kus andmemudelid sageli muutuvad, näiteks FHIR StructureDefinition ressursse kasutades, sest operatiivne visuaalne ülevaade aitab kiiremini muudatustest aru saada. Lisaks on PlantUML küps avatud lähtekoodiga projekt, millel on tugev kasutajaskond, lai valik UML diagrammi tüüpe ning rohkelt kohandamisvõimalusi, mistõttu saab seda hõlpsasti integreerida näiteks HAPI FHIR-iga, et genereerida automaatselt terviklikke UML skeeme [44].

Vaadates ka Mermaid-i poole, võib mainida, et sellel on sarnane tekstipõhine lähenemine diagrammide loomisele [45]. Aga ometi kipub Mermaid UML diagrammide osas olema piiratum. See suudab luua peamised klassidiagrammid ning pakub ka mõningaid võimalusi stiilide ja kujunduse kohandamiseks, näiteks värvide, kujundite ja joonevormingute määramiseks. Kuid keerukamate notatsioonide või spetsiifiliste UML funktsioonide korral, mida võib nõuda näiteks FHIR-i andmemudelite täpne käsitlemine, jääb Mermaid tihtipeale vajaka.

Selle põhjal tegi autor otsuse kasutada PlantUML-i, lähtudes tööriista ulatuslikust UML diagrammide toest, avatud lähtekoodiga iseloomust, rakenduse kasutusmugavusest ning tugevast kogukonnast, samuti varasemast kogemusest, mis kinnitas PlantUML-i sobivust projekti eesmärkide täitmisel.

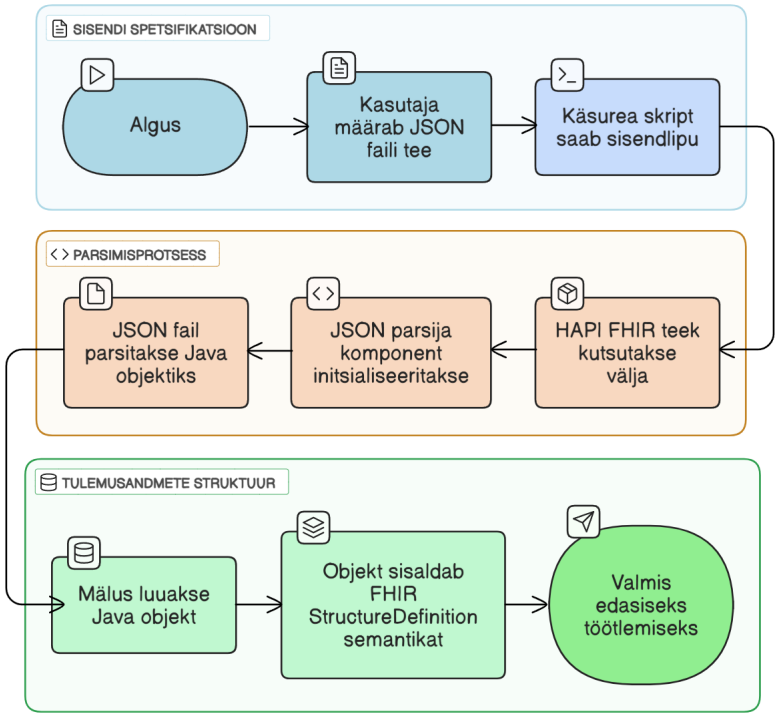
3.2 FHIR-i struktuuri teisendamine UML mudeliks

3.2.1 FHIR profiilide töötlemine HAPI FHIR teegi abil

Töövoo esimeses etapis toimub lähteandmete ehk FHIR-i StructureDefinition-i laadimine. See protsess algab vastava JSON vormingus faili määratlemisega, mille asukoht edastatakse käsureaskriptile spetsiaalse sisendlipu kaudu *-input*. Kasutaja saab sel viisil paindlikult määrata, millist sisendprofiili parasjagu töödelda, ilma et skript ise vajaks ümberkirjutamist.

Faili parsimiseks kasutatakse HAPI FHIR teeki. Täpsemalt rakendatakse selles etapis HAPI paketi komponenti *newJsonParser*, mille abil JSON struktuur tõlgendatakse otse Java objektiks [46]. Selle tulemusena luuakse mälus keerukate seoste ja sisemiste atribuutidega andmestruktuur, mis peegeldab kogu FHIR StructureDefinition-i semantilist sisu. See objekt toimib sisuliselt lähtepunktina kogu ülejäänud teisendustöötlusele — kõik järgnevad moodulid hakkavad töötlema just selle struktuuri kaudu kättesaadavaid klasse, elemente ja tüpoloogilisi seoseid.

Joonisel 3 on kujutatud töövoog alates sisendi määratlemisest kuni StructureDefinition objekti valmimiseni, mille baasil saab rakenduse järgmistes etappides edasi tegutseda.



Joonis 3. FHIR StructureDefinition parsimise töövoog.

3.2.2 Struktuurielementide vastendamine UML klassidele

Pärast FHIR andmete edukat parsimist kerkib järgmise olulise sammuna vajadus neid andmeid süstemaatiliselt ümber struktureerida selliselt, et neist oleks võimalik koostada selgesti mõistetav ja visuaalselt loetav UML klassidiagramm. Selle eesmärgi saavutamiseks kavandati sisemine objektimudel, mis jagab UML klassidiagrammi moodustavad komponendid loogilisteks ja selgelt eristatud klassideks.

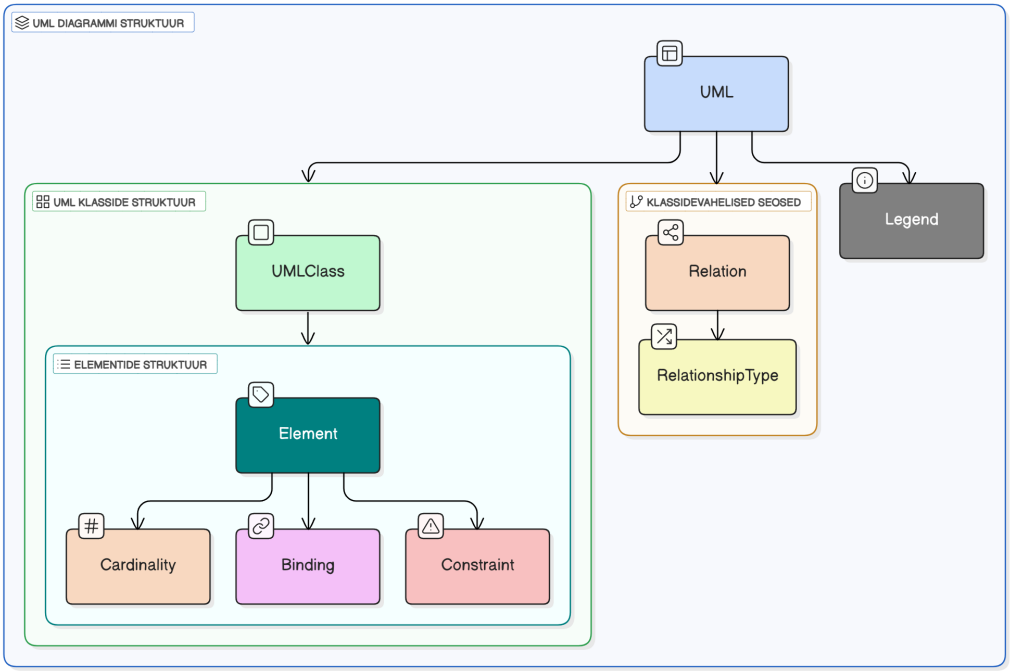
Kõige kesksem struktuurielement on UML klass, mis toimib konteinerina kogu diagrammi jaoks. See klass säilitab kõiki modelleeritavaid klasse, nendevahelisi suhteid ning ka täiendavaid visuaalseid elemente. Lisaks sisaldab UML objekt viidet peamisele, ehk juurklassile. Selline disain võimaldab diagrammi keskendada ning selgitada, milline osa mudelist on lähtepunkt ja millised on tugistruktuurid.

UML klasside esindamiseks loodi eraldi klass nimega *UMLClass*. Iga *UMLClass* esindab ühte FHIR elemendist tuletatud tüüpi ja sisaldab endas vastavaid atribuute, mis FHIR andmemudelis sellele tüübi tasemel on omistatud. Näiteks *Patient* ressurss tõlgendatakse vastavaks *UMLClass* objektiks, millel on kindel nimi ja atribuudid. Kui FHIR-element viitab teisele elemendile (see toimub läbi alamtee) luuakse vastav uus *UMLClass* ja nende kahe klassi vahele määratakse seos.

Selle seose esitamiseks defineeriti spetsiaalne *Relation* klass, mis modelleerib kahe klassi vahelise ühenduse. Iga *Relation* omab täpsustust selle kohta, mis tüüpi seos on tegemist, mida väljendab konstantide loend *RelationShipType*. See võimaldab eristada näiteks kompositsiooni, agregatsiooni või assotsiatiivseid seoseid. Lisaks loodi *Legend* klass, mis võimaldab lisada visuaalsesse diagrammi abistavaid märke või selgitusi.

Klassi *UMLClass* sees leidub veel üks keskne komponent – *Element*, mis vastab ühele FHIR elementidele ja sisaldab ulatuslikku metaandmestikku. Seal on esitatud näiteks atribuudi nimi, tüüp, mitmesus, valikväärtused, piirangud ja teised omadused. Selle struktuuri rikastamiseks loodi täiendavad klassid *Cardinality*, *Binding* ja *Constraint*, mis vastavalt kirjeldavad atribuutide esinemissagedust, väärtuste sidumist kontrollitud sõnastikega ning muude reeglite kehtivust. See kihiline modelleerimine tagab, et teisendatav FHIR andmemudel säilitab oma semantilise täpsuse ka pärast konversiooni UML vormingusse ning võimaldab diagrammi hilisemat programmilist redigeerimist, laiendamist ja visualiseerimist.

Joonisel 4 on kujutatud selle sisemise objektimudeli üldstruktuur koos põhiklasside ja nende omavaheliste seostega.



Joonis 4. UML klassidiagrammi objektimudel.

3.2.3 Elementitaseme teisenduse käsitlus

Järgmine oluline etapp on FHIR StructureDefinition-i üksikelementide parsimine ning nende tõlgendamine UML komponentidena.

Protsess algab ElementDefinition loendi iteratsiooniga, mille käigus iga element töödeldakse eraldi. Igaüks neist esindab ühte loogilist komponenti FHIR andmemudelis. Et tagada elementide efektiivne haldamine ja nendevaheliste seoste jälgitavus, kasutatakse abistavat mälustruktuuri — *map*, kus iga ElementDefinition salvestatakse oma unikaalse identifikaatori alusel. See identifikaator on osa iga ElementDefinition objekti struktuurist ning vastab väljale *id*, mis FHIR spetsifikatsiooni kohaselt on kohustuslik ja unikaalne kogu konkreetse StructureDefinition ulatuses [47]. Selline *id* põhine kaardistus võimaldab luua täpseid viiteid ja seoseid elementide vahel teisenduse järgnevates etappides.

Erilist tähelepanu nõuab loendi esimene element, kuna see kujutab endast üldjuhul kogu StructureDefinition-i juurelementi. Just see element määratakse töötluse käigus kui peamine UML klass, mis tähistatakse vastava atribuudi kaudu ja toimib edaspidiste klasside loomise ning seoste konstrueerimise lähtepunktina.

Elementide analüüsi ja UML komponentide loomise ülesanne on delegeeritud klassile *ElementFactory*, mis on loodud vastavalt klassikalisele *Factory* disainimustrile. See muster võimaldab kapseldada objektide loomise loogika eraldiseisvasse komponendi, muutes kogu

süsteemi paindlikumaks, laiendatavamaks ja kergemini hooldatavaks [48]. *ElementFactory* vastutab iga FHIR ElementDefinition parsimise eest ja loob selle põhjal sobiva UML elemendi vastavalt eelnevalt määratletud reeglistikule.

Selleks, et teisendada FHIR andmemudel UML vormingusse, kasutatakse ainult neid ElementDefinition atribuute, mis kannavad semantilist või struktuurset tähendust UML diagrammi jaoks. Järgnevas loetelus on esitatud kõik kasutusele võetud väljad koos viitega FHIR-i sisemisele välja nimele:

- Identifikaator (*ElementDefinition.id*)
- Tee (*ElementDefinition.path*)
- Tüüp (*ElementDefinition.type*)
- Viitetüüp (*ElementDefinition.type.code*)
- Viiteprofiil (*ElementDefinition.type.targetProfile*)
- Kardinaalsus (*ElementDefinition.min, ElementDefinition.max*)
- Fikseeritud väärtus (*ElementDefinition.fixed[x]*)
- Sidumine väärtuskomplektiga (*ElementDefinition.binding*)
- Piirangud (*ElementDefinition.constraint*)
- *Slicing* eristustunnus (*ElementDefinition.slicing.discriminator*)

Edaspidi talletatakse eraldi ka elementide vahelised seosed. Selleks kasutatakse teist *map* struktuuri, mis kaardistab, milline element viitab teisele — see info on hiljem ülioluline, et õigesti konstrueerida UML klasside vahelised seosed.

Oluliseks aspektiks osutuvad ka FHIR-i spetsiifilised mehhanismid, nagu *slicing*, mida ei saa UML diagrammis kujutada otsejoones. Seetõttu arendati eraldi mehhanismid nende elementide tähistamiseks diagrammis visuaalselt erineval viisil. Samuti kogutakse ja salvestatakse kõik *binding* ja *constraint* tüüpi metaandmed, mida kasutatakse hiljem diagrammi legendi koostamiseks, et parandada diagrammi loetavust ja informatiivset väärtust.

3.2.4 Snapshot ja differential vaadete käsitlemise eripärad

Parsimisel on oluline mõista, et FHIR StructureDefinition esitab andmemudeli struktuuri kahes vaates: *snapshot* ja *differential*. Kuna igal *snapshot* elemendil on kogu vajalik teave juba kaasas, saab sellest kujundada iseseisvalt mõistetava ja visuaalselt täieliku klassidiagrammi, ilma et oleks vaja välist konteksti. Vastupidiselt sellele on *differential* vorm mõeldud mudelimuudatuste esindamiseks võrreldes mõne baasprofiiliga. See loob teisendamise seisukohalt olulise väljakutse: *differential* üksinda ei sisalda täielikku

informatsiooni UML klasside konstrueerimiseks.

Selle probleemi lahendamiseks rakendatakse kahefaasilist parsingu strateegiat. Esmalt töödeldakse täielikult *snapshot*, mille käigus genereeritakse kõik UML-i vastavad struktuurid ja salvestatakse need *map* struktuuri, kus iga element on identifitseeritav oma unikaalse *id* kaudu. Seejärel alustatakse *differential* elemendiloendi parsimist. Kui leitakse mõni element, mille *id* vastab varem salvestatud *snapshot* elemendile, toimub selle kahe vaate võrdlus: kõik *differential*-is esitatud muudatused kantakse üle vastavale *snapshot* elemendile, säilitades samas kogu esialgse konteksti ja metaandmed. Nii sünteesitakse kahe vaate põhjal täielik, kuid diferentseeritud UML.

3.2.5 UML diagrammi genereerimine PlantUML süntaksi alusel

Teisendusprotsessi lõppfaasis luuakse PlantUML süntaksile vastav tekstiline kirjeldus, mille põhjal genereeritakse visuaalne UML klassidiagramm. Selleks on rakendatud kihilist ja modulaarset objektimudelit, kus iga komponent teab, kuidas ennast graafiliseks kujundiks teisendada.

Diagrammi genereerimine toimub justkui dominoefekti põhimõttel: protsess algab juurklassist, mis esindab FHIR-profiili keskset ressurssi, ning seejärel käivitatakse järjestikku kõik seotud alamstruktuurid. Iga objekt kutsub välja järgmise, moodustades ahela, mille kaudu kogu mudel astmeliselt visualiseeritakse.

Kõik loodud objektimudeli klassid sisaldavad meetodit *toString()*, mis tagastab nende enda tekstilise esituse vastavalt PlantUML süntaksile. Iga komponent teab täpselt, kuidas väljendada oma sisemist struktuuri ja tähendust graafilises kujul, arvestades selle konteksti kogu mudelis. Tänu sellele saab iga objekt ise koostada oma vastava UML fragmendi, ilma et keskne generaator peaks tundma kogu mudeli detaile. See tagab paindliku, laiendatava ja hooldatava lahenduse, kus iga osa vastutab oma visualiseerimise eest.

Lisaks põhilisele struktureerimisele toetab skript ka visuaalset kohandamist. Selleks on loodud abifunktsioonid, mis võimaldavad lisada *bold* teksti, eristada visuaalselt näiteks *slicing* tüüpi klasse ning kujundada diagrammi kooskõlas PlantUML-i süntaktiliste reeglitega.

Kui kõik klassid on *toString()* meetodite abil genereerinud PlantUML süntaksile vastava koodi, koondatakse kogu väljund ühte tekstifaili. See fail sisaldab kogu UML diagrammi kirjelduse. Kuigi loodud skript ei genereeri visuaalset kujutist, saab kasutaja soovi korral selle tekstifaili käsitsi ette anda PlantUML-i tööriistale, et luua visuaalne väljund.

3.3 FHIR andmemudeli kujutamine UML klassidiagrammina

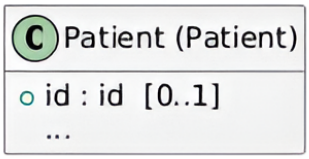
Pärast FHIR andmemudeli sisemise struktuuri analüüsi ja selle vastendamist UML objektimudelile liigub töövoog järgmisse etappi — visuaalse UML klassidiagrammi loomisesse. Käesolevas peatükis kirjeldatakse, kuidas eelnevalt parsetud ja modelleeritud andmestik teisendatakse PlantUML süntaksi abil visuaalseks esituseks. Eesmärk on luua struktureeritud ja üheselt mõistetav kujutusviis, mis säilitab FHIR profiili semantilise täpsuse, ent muudab selle hõlpsasti mõistetavaks nii arendajatele kui ka teistele valdkonnaspetsialistidele.

3.3.1 UML klasside struktuurne kujundus

UML klasside nimetamisel on oluline tagada, et iga nimi oleks selge, arusaadav ja kannaks edasi nii klassi struktuurset rolli kui ka semantilist tähendust. Käesolevas töövoos rakendatakse selget ja järjepidevat nimekonventsiooni, mis aitab hoida diagrammi loetavana ning võimaldab klasside tähendust kiiresti mõista.

Peamise klassi puhul, mis vastab FHIR profiili juurelementidele, moodustatakse klassi nimi kahest osast: esmalt tüüp ning seejärel sulgudes elemendi nimi. Mõlemad komponendid saadakse vastavast ElementDefinition objektist – tüüp väljast *type.code*, nimi aga elemendi identifikaatori alusel. Ülejäänud, tavapäraste klasside puhul kasutatakse ainult klassi nime ilma tüübitäpsustuseta sulgudes. Nende nimed esitatakse otse vastavalt elemendi tähendusele. Selline vahetegemine võimaldab rõhutada peamise klassi keskset rolli, hoides samal ajal ülejäänud diagrammi visuaalselt lihtsana ja fokusseerituna.

Joonisel 5 on toodud näide *EEBasePatient* profiili põhiklassist [12], mille nimi ja tüüp on visualiseeritud vastavalt kirjeldatud konventsioonile. Klassikonteineris kuvatakse ka mõned atribuudid, mille üksikasjalikum käsitlus ja visualiseerimispõhimõtted on esitatud järgmises peatükis.



Joonis 5. *EEBasePatient* klassi nimetus ja tüüp UML diagrammil.

3.3.2 Elemendiomaduste visualiseerimine

Iga FHIR StructureDefinition-i elemendi visualiseerimine UML diagrammis eeldab täpset ja süstemaatilist lähenemist, et säilitada selle semantiline tähendus ning tagada ühtne esitlusviis. Selleks töötati välja spetsiaalne süntaktiline ja visuaalne mall, mille järgi iga element kujutatakse UML klasside sees, järgides nii PlantUML-i kui ka FHIR-i formaadi loogikat.

Kõigepealt määratakse igale elemendile *visibility* ehk nähtavustasand, mis PlantUML-i süntaksis tähistatakse sümbolitega. Nendest kasutati kolme peamist tasandit, et edasi anda elemendi funktsionaalset tähendust:

- *public* (+) – kasutatakse enamiku tavapäraste elementide puhul. Graafiliselt kuvatakse seda PlantUML-is rohelise ringina, mis viitab sellele, et element on üldkasutatav ja avatud lugemiseks/muutmiseks.
- *protected* (#) – rakendub juhul, kui elemendil on määratud *Fixed Value*, mis tähendab, et väärtus peab olema fikseeritud ega tohi muutuda. See piirang nõuab visuaalset tähistust, et rõhutada piiratud kasutusala. Graafiliselt kuvatakse seda kollase rombina.
- *package-private* (~) – kasutatakse elementide puhul, mille tüübiks on *Canonical* või *Reference*. Sellised elemendid viitavad sageli teistele FHIR resurssidele ning nende kasutusala on piiratud kontekstuaalselt, mistõttu on nende tähistamine eraldi nähtavustasemega loogiline. Graafikas tähistatakse seda sinise kolmnurgana.

Pärast nähtavustaseme tähistamist kuvatakse elemendi nimi, millele järgneb koolon ja seejärel tüüp. Sellele järgneb kardinaalsus, mis märgitakse ruuduklambrite sees. See annab selgelt edasi, mitu korda element võib esineda ja kas see on kohustuslik. Kui elemendil on määratud fikseeritud väärtus, lisatakse see visualiseeringu lõppu, eelneva võrdlus märgiga.

Mõnede tüüpide puhul täpsustatakse UML diagrammis ka profiiliviited või lubatud väärtused, mis on seotud konkreetse elemendi kasutusega. See kehtib eelkõige tüüpide puhul nagu *Reference*, *Canonical*, aga ka *Extension*, *Address* ja muud sarnased struktuurid, millele võib olla määratud konkreetne *profile* või *targetProfile*. Sellisel juhul esitatakse tüüp laiendatud kujul, kus sulgudes on välja toodud lubatud profiilid või tüübid.

See standardiseeritud lähenemine võimaldab igat elementi esitada kompaktselt, kuid samas informatiivselt. Kasutaja saab ühelt pilgul aru mitte ainult elemendi nimest ja tüübist, vaid ka selle kasutusreeglitest, esinemissagedusest ning võimalikest piirangutest.

Joonisel 6 on toodud *EEBasePatient* profiili peamine UML klass [12]. Diagrammil on

visualiseeritud peamise klassi kõik struktuurielemendid koos tüüpide, kardinaalsuste ja nähtavustasemetega. Lisaks on näha, kuidas tüübid nagu *Reference*, *Extension* ja *Address* sisaldavad täiendavat teavet seotud profiilide kohta.

Joonis 6. *EEBasePatient* profiili peamine UML klass koos tüüpide, kardinaalsuste ja nähtavustasemetega.

3.3.3 Klassidevaheliste seoste esitamine

UML diagrammides on klassidevaheliste seoste korrektne visualiseerimine keskse tähtsusega, kuna see määrab ära andmemudeli sisemise loogika ja komponentide omavahelise hierarhia. FHIR andmemudeli kontekstis on suhete tüüpide selge eristamine eriti oluline, et edastada, kuidas üksikud elemendid ja alastruktuurid ressursside sees omavahel suhestuvad. Just sellele vajadusele vastates loodi läbimõeldud strateegia seostetüüpide visualiseerimiseks UML-is.

Kõigepealt määrati, et peaklassist lähtuvad kõik esmased seosed teistesse klassidesse kantakse UML diagrammile kompositsioonitüüpi ühendusena. See visualiseeritakse PlantUML-is suletud rombiga ühenduse alguses, mis tähistab tugevat elutsüklilist seost: kaasatud alamstruktuurid eksisteerivad ainult koos peaklassiga.

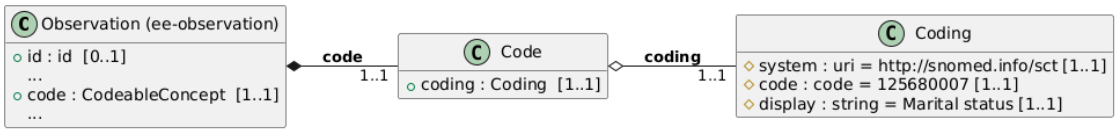
Iga seose juures on selgelt välja toodud ka element, mis seose loob — see esitatakse seose nimena otse ühendusjoone peal, tavaliselt vastates FHIR-i elemendi nimele. Lisaks on välja toodud ka kardinaalsus mis näitab, mitu korda see alamklass võib peaklassi kontekstis esineda.

Edasised klassid, mis ise lähtuvad juba nendest esmatasandi struktuuridest, ei visualiseerita

enam kompositsioonitüüpi seostena, vaid hoopis agregatsioonitüüpi ühendustena. UML-is tähistab seda tühja rombiga ühendus, mis osutab nõrgemale, osaliselt sõltumatule seosele. See eristus rõhutab, et kuigi andmestruktuurid on omavahel seotud, ei sõltu nende eksistents enam tingimata oma vanemklassist.

Selline kihiline ja seostüübipõhine visualiseerimine tagab, et lõplik UML diagramm ei esita lihtsalt struktuuri, vaid edastab ka olulist semantilist infot andmemudeli koostisosade tugevusastmete ja hierarhia kohta. See lähenemine aitab kasutajatel paremini mõista FHIR ressursside sisemist loogikat ning muudab diagrammi tõlgendamise intuitiivsemaks.

Joonisel 7 on näha näide klassidevaheliste seoste visualiseerimisest, mis on koostatud osaliselt *EEBaseMPISocialHistoryMaritalStatus* profiili põhjal [49]. Tegemist on Eesti kontekstis kasutatava FHIR profiiliga, mis kirjeldab inimese perekonnaseisuga seotud sotsiaalajaloo andmeid ning põhineb *EEBaseObservation* ressursil. Diagramm kujutab, kuidas peamisest *Observation* klassist lähtub *code* kompositsiooniline seos *Code* klassi, mis omakorda on seotud *Coding* klassiga nõrga agregatsiooniseosena.



Joonis 7. Seosed UML diagrammil profiili *EEBaseMPISocialHistoryMaritalStatus* näitel.

3.3.4 Tüübivalikute ja määrangute esitamine

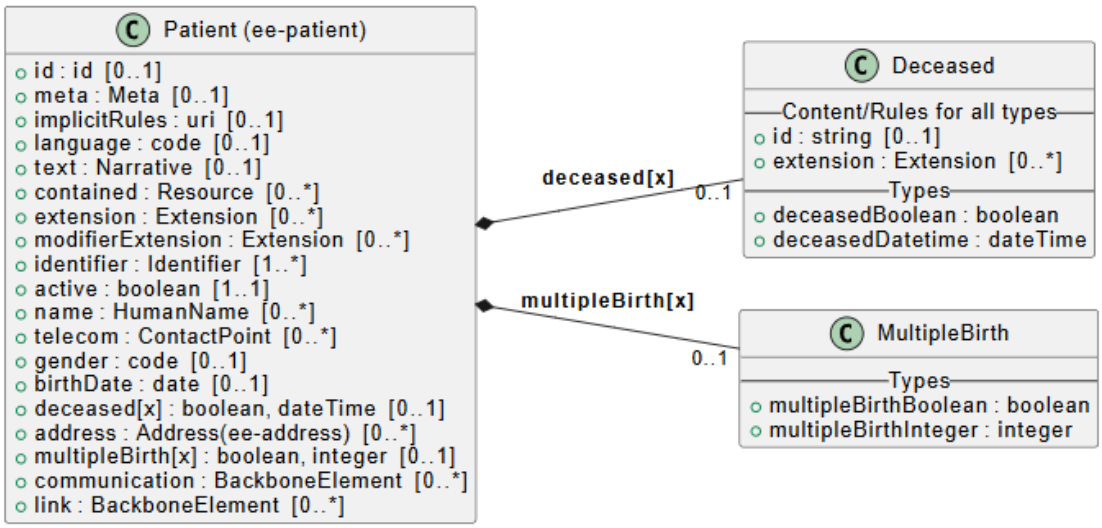
FHIR spetsifikatsioonis on defineeritud mehhanism nimega *Choice of Types*, mis võimaldab määrata, et üks konkreetne element võib esineda mitmes erinevas andmetüübi vormis. Selline paindlikkus on FHIR modelleerimises sisuliselt oluline, võimaldades ühtset struktuuri erinevates kontekstides, kuid UML diagrammide kontekstis võib see kaasa tuua tõlgenduslikku ebamäärasust, kui tüübivalikuid ei visualiseerita üheselt arusaadaval viisil.

Probleemi lahendamiseks on välja töötatud spetsiaalne käsitlusviis, mille keskmes on eraldiseisva UML klassi loomine iga sellise tüübiühiku jaoks, millel esineb rohkem kui üks lubatud tüüp. Kui skript tuvastab *Choice of Types* olukorra, konstrueeritakse sellele vastav *Choices of Types* UML klass. See klass koondab kõik antud elemendile määratud võimalikud tüübivariandid — kas loeteluna või struktureeritud rühmana nimega *Types*, mis teeb mudeli loogika visuaalselt üheselt mõistetavaks.

Lisaks tüüpide esitlusele võimaldab antud klassi struktuur siduda ka muid metaandmeid,

mis kehtivad kõigi tüüpvariantide kohta. Selline kontsentreeritud esitusviis vähendab vajadust andmete dubleerimise järele ning toetab FHIR struktuuri säilimist ka UML visualiseeringu tasandil.

Joonisel 8 on toodud näide *EEMPIPatient* profiili kahest *Choice of Types* elemendist — *deceased[x]* ja *multipleBirth[x]*. Tegemist on Eesti rahvastikuregistri põhjal koostatud FHIR profiiliga, mis kirjeldab patsiendi põhiandmeid MPI (*Master Patient Index*) süsteemi kontekstis ning põhineb *EEBasePatient* ressursil. Mõlema elemendi puhul on loodud eraldi UML klassid, mis koondavad nende tüübivalikud, säilitades samas seose lähteelementidega.



Joonis 8. *Choice of Types* elementide visualiseerimine UML diagrammis *EEMPIPatient* näitel.

3.3.5 Slice mustrite esitamine UML diagrammil

Üks keerukamaid ning samas olulisi väljakutseid FHIR StructureDefinition-i teisendamisel UML diagrammiks on seotud *slice* elementide korrektse käsitlemise ja visualiseerimisega.

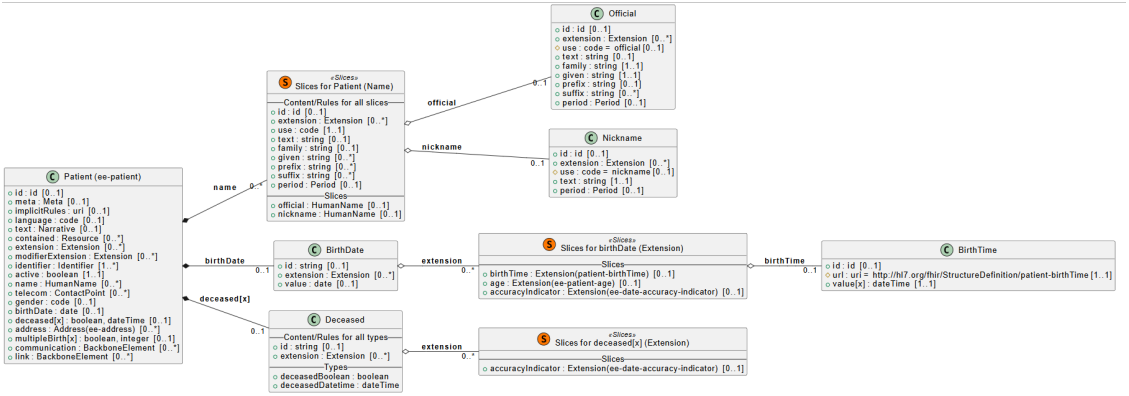
Selleks, et säilitada *slice*-ide tähendust ja vältida nende käsitlemist tavapäraste elementidena, kavandati spetsiaalne mudelikäsitlus. Iga *slice* grupp visualiseeritakse esmalt eraldi metaobjektina, mille tähistuseks on vormistus *Slices for [element]*. Selle objekti alla koondatakse kõik viilud, määratledes nende nimed, tüübid ning vajadusel ka grupile kehtivad ühised omadused.

Käsitletud struktuur põhineb FHIR ametlikus dokumentatsioonis toodud visualiseerimispõhimõtetel, kus viilud esitatakse hierarhilises puuvaates. Viilutatava elemendi tuvastamine toimub automaatselt, lähtudes *discriminator* välja olemasolust, mis määratleb millise omaduse või tunnuse alusel viilud eristatakse. *Discriminator*-i

tuvastamine on skriptis võtmekohaks, mille alusel alustatakse vastava *slice* grupi loomist.

Lisas 4 on toodud näide sellest, kuidas FHIR-i ametlik dokumentatsioon visualiseerib viilutatud elemente puustruktuuris *EEMPIPatient* profiili põhjal [50]. Elemendi *Slices for name* all kuvatakse viilud *official* ja *nickname* eraldi ridadena koos neile omaste atribuutidega.

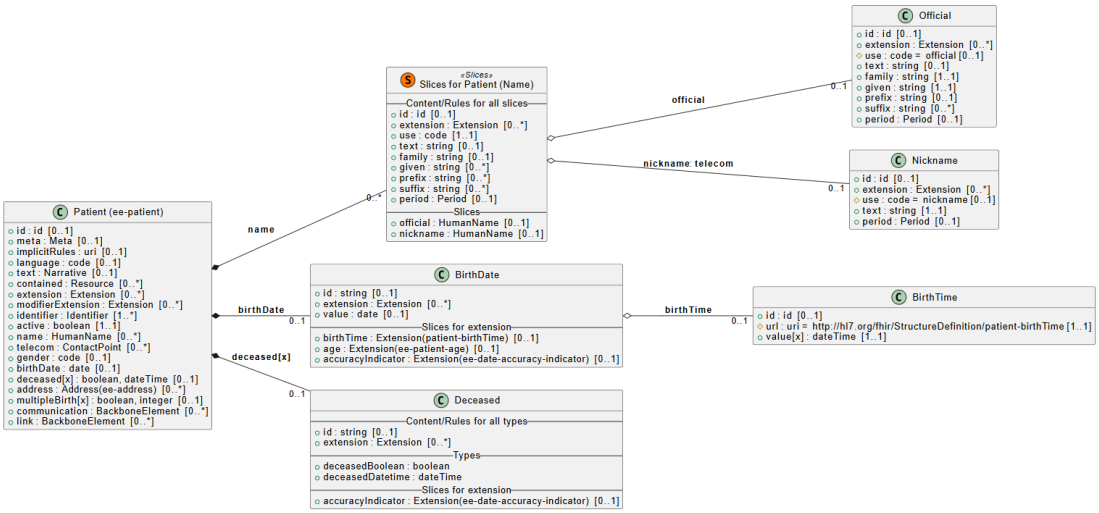
Joonisel 9 on kujutatud automaatselt genereeritud UML diagramm, mis põhineb samal lühendatud *EEMPIPatient* profiilil [50] ja kus iga *slice* on visualiseeritud eraldi UML klassina. Diagrammil on selgelt näha, et iga *slice* on eraldatud ja seotud läbi metaobjekti *Slices for Patient (Name)*, mis sisaldab ka *slice*-de ühiseid atribuute.



Joonis 9. Täielik UML diagramm slice klasside kaupa *EEMPIPatient* põhjal.

Suuremate FHIR profiilide puhul võib selline *slice*-de käsitlus UML diagrammi olulise mahukuseni lisati skriptile paindlikkust võimaldav valikuvõimalus. Selleks töötati välja spetsiaalne konfiguratsioonilipp *reduceSliceClasses*, mille aktiveerimisel muudetakse visualiseerimisloogikat: iga *slice*-i jaoks eraldi klassi loomise asemel koondatakse *slice*-id otse põhiklassi sisse ning tähistatakse seal vastava visuaalse eristusega.

Joonisel 10 on esitatud sama *EEMPIPatient* profiili [50] UML visualiseerimine juhul, kui on aktiveeritud *reduceSliceClasses* režiim. Selles lähenemises ei looda igale viilule eraldi klassi — selle asemel on viilud koondatud põhiklassi sisse ja tähistatud visuaalselt alamvariantidena.



Joonis 10. *EEMPIPatient* visualiseerimine *reduceSliceClasses* režiimis.

Siiski toimub selline lihtsustatud koondamine ainult juhul, kui viiludel puuduvad eraldiseisvad ühiselemendid, mis on spetsiaalselt määratletud ja kuuluvad eraldi grupistruktuuri. Kui viiludel on ühised elemendid, mis vajavad iseseisvat kirjeldamist, säilitatakse nende eraldi klassid ka *reduceSliceClasses* režiimis. Selline tingimus tagab, et mudeli semantiline täpsus ei kaoks ka juhul kui kasutatakse lihtsustatud visualiseerimist.

3.3.6 Piirangute modelleerimine UML skeemides

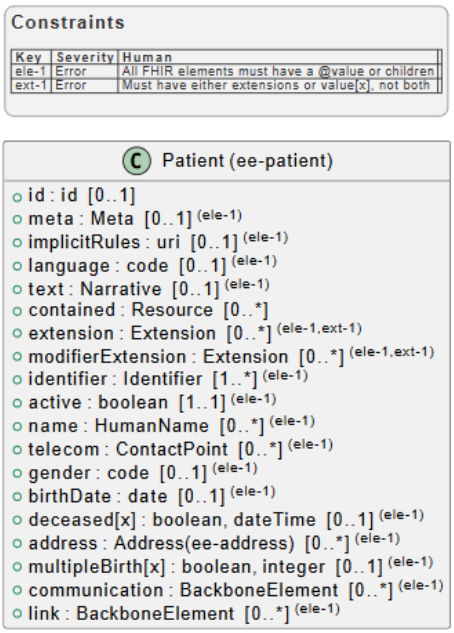
FHIR andmemudelis mängivad *constraints* ehk piirangud olulist rolli, määratledes lisareegleid või tingimusi, mis peavad kehtima konkreetsete elementide puhul. Need võivad hõlmata keerulisemaid loogilisi tingimusi, mida ei saa väljendada lihtsalt tüübi või kardinaalsus kaudu. UML diagrammi kontekstis on oluline leida viis, kuidas neid piiranguid visualiseerida selgelt ja samas visuaalset ülekoormust vältides.

Valitud visualiseerimisviis lähtub ideest, et iga piirang seostatakse vahetult selle elemendiga, millele ta rakendub, kuid tehakse seda diskreetselt. Konkreetse *Element* objekti klassi lõppu lisatakse märge kujul ülaindeks, kus piirang on tähistatud selle *constraint key* alusel. See on lühike unikaalne tunnus, mis võimaldab elemente mitte visuaalselt koormata pikkade seletustega, kuid annab kasutajale viite edasiseks uurimiseks.

Kõik kasutatud *constraint*-id koondatakse diagrammi legendisse — spetsiaalsesse selgitavasse alajaotusesse, kus iga *constraint key* juurde lisatakse selle *severity* ning *human-readable text*, mis kirjeldab piirangu sisulist tähendust. See lähenemine tagab, et piirangud on dokumenteeritud ja arusaadavad, kuid ei koorma diagrammi liigselt.

Lisaks on skripti lisatud paindlikkus nende visualiseerimise osas. Kasutaja saab käsureal aktiveerida või välja lülitada piirangute kuvamise, kasutades selleks lippu *showConstraints*. See võimaldab kasutajal kohandada väljundi detailsust vastavalt sihtgrupi või kasutusjuhtumi vajadustele.

Joonisel 11 on toodud näide *EEMPIPatient* profiili [50] põhjal sellest, kuidas piirangud visualiseeritakse UML diagrammis. Iga piirangu tähis on paigutatud vastava atribuudi lõppu ülaindeksina ning piirangute täpsemad selgitused koos tõsiduse ja kirjeldusega on lisatud diagrammi legendi.



Joonis 11. Piirangute visualiseerimine UML diagrammis *EEMPIPatient* näitel.

3.3.7 Väärtuste sidumine ja väärtusepiirangud

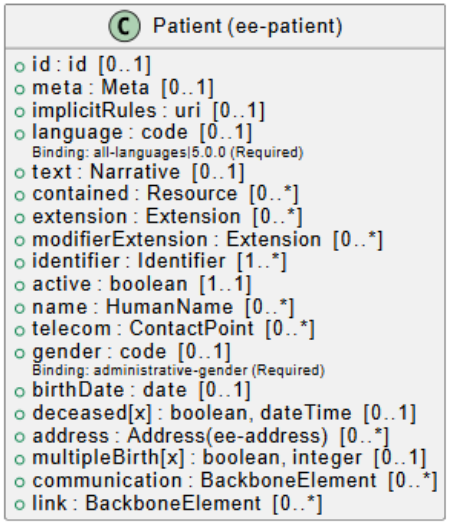
Binding ehk väärtussidumine on FHIR mehhanism, mille eesmärk on määratleda, milliseid väärtusi konkreetne element tohib sisaldada. Sidumine toimub üldjuhul viitega välishallatavale väärtuskomplektile (*ValueSet*), koodisüsteemile või terminoloogilisele sõnastikule. UML diagrammi kontekstis on oluline leida tasakaalustatud viis *binding*-u esitamiseks: see peab olema informatiivne, kuid mitte visuaalselt ülekoormav.

Valitud visualiseerimisviis lähtub diskreetsuse ja loetavuse põhimõtetest. Kui FHIR element on seotud mõne väärtuskomplektiga, lisatakse vastav märge otse selle UML elemendi alla. Esitusvormis kasutatakse selgelt eristatavat tekstilist viidet, mille alguses on *Binding:*, millele järgneb lühikirjeldus sidumise sisust. Täiendavalt kajastatakse ka sidumise tugevus (*binding strength*), mille väärtuseks võib olla näiteks *required*, *preferred*, *example* või *extensible*. Selline kujutusviis võimaldab kiiresti mõista, kas konkreetse

atribuudi väärtus peab vastama mingile välishallatud reeglistikule või mitte.

Süsteemi paindlikkuse tagamiseks on *binding* elementide visualiseerimine tehtud valikuliseks. Skript toetab käsureapõhist seadistust läbi lipu *showBindings*, mille abil saab kasutaja otsustada, kas *binding* teave lisatakse diagrammile või mitte.

Joonisel 12 on näidatud, kuidas *binding* teave visualiseeritakse UML diagrammis *EEMPIPatient* [50] profiili põhjal. Näiteks on atribuudi *language all* esitatud viide väärtuskomplektile *all-languages* ning atribuudi *gender all administrative-gender*, mõlemad koos määratud tugevusastmega — sel juhul *Required*.



Joonis 12. *Binding* elementide visualiseerimine UML diagrammis *EEMPIPatient* põhjal.

3.3.8 Diagrammide lugemisjuhendi kasutuselevõtt

UML diagrammide koostamisel on kriitilise tähtsusega leida tasakaal diagrammi informatiivse sisutiheduse ja visuaalse loetavuse vahel. FHIR StructureDefinition-i puhul ei koosne andmemudel üksnes elementide ja nendevaheliste seoste struktuurist, vaid sisaldab ka mitmeid metaandmeid, mis määratlevad profiili päritolu, identiteedi ning kasutuskonteksti. Selleks, et selline teave oleks UML diagrammil kättesaadav, kuid ei häiriks selle peamist struktuurilist ülesehitust, otsustati vastavad metaandmed koondada eraldi visuaalsesse komponenti — legendisse. Legend kujutab endast diskreetset, kuid sisuliselt tähenduslikku alajaotust, mis paikneb diagrammi äärealal ja ei sekku klassistruktuuri kujutamisse.

Legendis esitatakse olulisemad StructureDefinition-i metaandmed, mis võimaldavad profiili üheselt identifitseerida ning mõista selle tähendust FHIR ökosüsteemis. Legend sisaldab järgmisi välju:

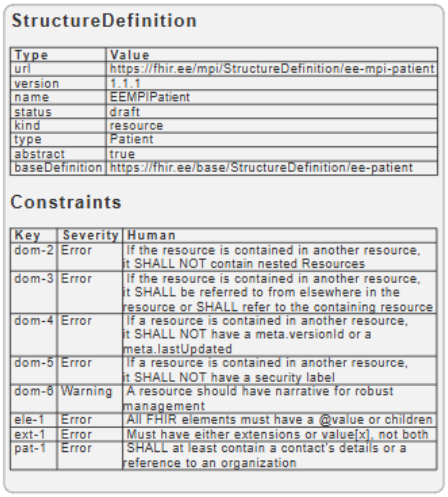
- *URL* – profiili unikaalne identifikaator.
- *Version* – versiooninumber.
- *Name* – ametlik nimetus.
- *Status* – olek.
- *Kind* – profiili kategooria.
- *Type* – FHIR tüüp, millele profiil kehtib.
- *Abstract* – märge, kas profiil on abstraktne.
- *Base Definition* – viide baasprofiilile, millel konkreetne profiil põhineb.

Lisaks üldmetaandmetele sisaldab legend ka koondatud loetelu kõikidest piirangutest, mis profiilis määratletud on.

Et tagada UML diagrammi kohandatavus erinevates kasutusstsenaariumites — võimaldab skript kontrollida legendi nähtavust käsurealipu *showLegend* kaudu. Selle lippu aktiveerides lisatakse legend automaatselt diagrammi osaks; välja lülitades jäetakse see teadlikult kõrvale.

Legend toimib seega kui semantiline ankur kogu UML diagrammi kontekstis. See loob vaatajale kiire ülevaate profiili olemusest ja ulatusest, määrates ära millise mudeli, versiooni ja päritoluga on tegemist ning millised reeglid on selle struktuurile kehtestatud — ilma et see hägustaks või koormaks diagrammi põhistruktuuri.

Joonisel 13 on esitatud legend EEMPIPatient profiili [50] põhjal UML diagrammi kontekstis. Ülemises osas kuvatakse kõik olulised *StructureDefinition*-i metaandmed, allpool aga koondtabel kõigi diagrammis kasutatavate piirangutega.



StructureDefinition

Type	Value
url	https://fhir.ee/mpi/StructureDefinition/ee-mpi-patient
version	1.1.1
name	EEMPIPatient
status	draft
kind	resource
type	Patient
abstract	true
baseDefinition	https://fhir.ee/base/StructureDefinition/ee-patient

Constraints

Key	Severity	Human
dom-2	Error	If the resource is contained in another resource, it SHALL NOT contain nested Resources
dom-3	Error	If the resource is contained in another resource, it SHALL be referred to from elsewhere in the resource or SHALL refer to the containing resource
dom-4	Error	If a resource is contained in another resource, it SHALL NOT have a meta.versionId or a meta.lastUpdated
dom-5	Error	If a resource is contained in another resource, it SHALL NOT have a security label
dom-6	Warning	A resource should have narrative for robust management
ele-1	Error	All FHIR elements must have a @value or children
ext-1	Error	Must have either extensions or value[x], not both
pat-1	Error	SHALL at least contain a contact's details or a reference to an organization

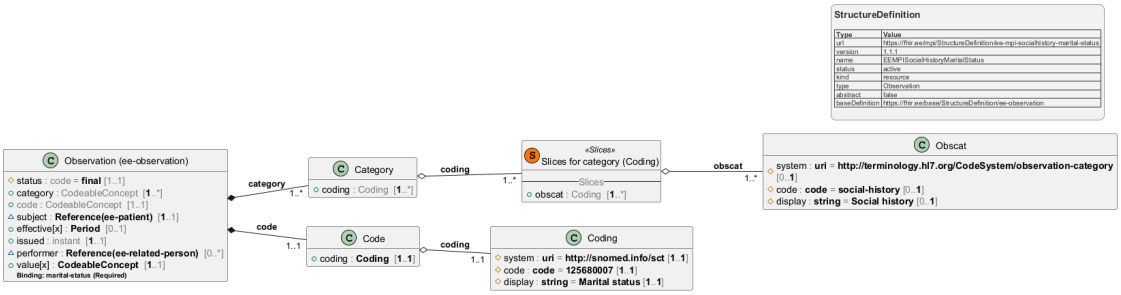
Joonis 13. *EEMPIPatient* profiili legend metaandmete ja piirangute visuaalne koondstruktuur.

3.3.9 Differential vaade märgistus UML diagrammil

Kahetasemeline parsinguviis avaldub selgelt ka UML diagrammi visuaalses esituses. Kuna *differential* tasemel kirjeldatud muudatused kantakse üle olemasolevale *snapshot* struktuurile, võimaldab loodud visualiseerimisloogika diagrammil üheselt eristada, millised klassielemendid on pärit baasprofiilist ning millised on kasutaja poolt modifitseeritud.

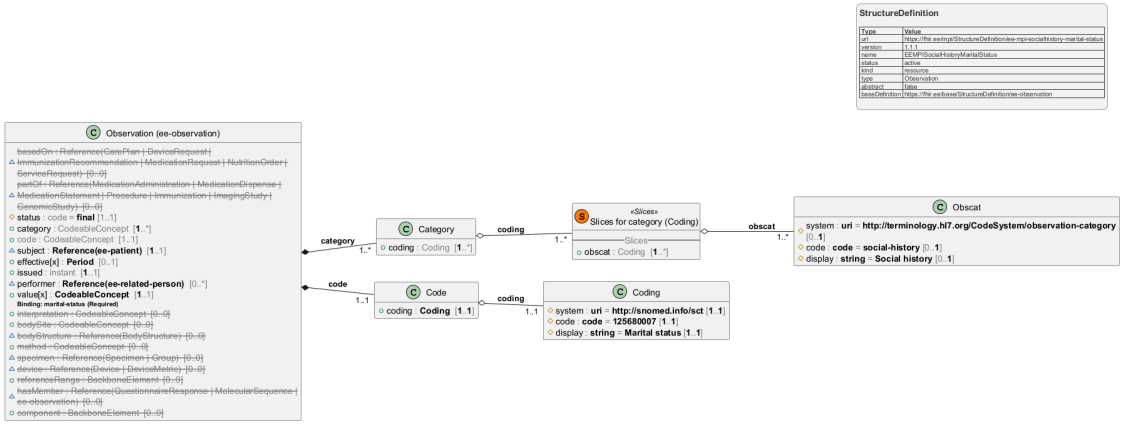
Selle saavutamiseks kasutatakse PlantUML-i süntaksi võimalusi, et tähistada struktuurimuudatusi visuaalselt eristuva kujundusega. Näiteks muudetud või spetsiaalselt kohandatud elemendid esitatakse diagrammil paksus kirjas (*bold*), samas kui päritud elemendid kuvatakse standardses, neutraalses stiilis. Selline tähistusviis tagab, et muudetud struktuurid paistavad kohe silma, võimaldades kiiret orientatsiooni keerukates mudelites.

Joonisel 14 on näidatud *EEMPISocialHistoryMaritalStatus* profiilil põhinev [49] UML, kus modifitseeritud elemendid on UML diagrammis esile tõstetud paksus kirjas. Ülejäänud struktuurielemendid pärinevad baasprofiilist ning on kujutatud vaikestiilis.



Joonis 14. Modifitseeritud elementide visuaalne esiletõstmine UML diagrammil *EEMPISocialHistoryMaritalStatus* profiilis.

Joonisel 15 on illustreeritud sama *EEMPISocialHistoryMaritalStatus* profiili põhjal [49] kuidas *differential* vaates käsitletakse baasprofiilis defineeritud, kuid eriprofiilis teadlikult eemaldatud elemente. Sellised elemendid tähistatakse kardinaalsusega *0..0*, viidates nende eemaldamisele semantilisel tasandil. UML diagrammil visualiseeritakse need läbikriipsutatud kujul (*crossout style*), mis võimaldab kasutajal selgelt eristada profiilist välja jäetud struktuuriosi. Eemaldatud elementide visualiseerimine on samuti valikuline funktsioon, mille kuvamist saab juhtida spetsiaalse käsurealipuga *hideRemovedObjects*.



Joonis 15. Eemaldatud elementide visualiseerimine UML diagrammil *EEMPISocialHistoryMaritalStatus* profiilis.

3.4 UML mudeli teisendamine FHIR-i struktuuriks

3.4.1 Keerukus vastupidises teisenduses

Kuigi FHIR profiilide teisendamine UML klassidiagrammideks osutus tehniliselt hästi hallatavaks tänu FHIR-i rangetele reeglitele ja selgelt defineeritud struktuurile, on vastupidine protsess – UML diagrammist FHIR StructureDefinition ressursi genereerimine – tunduvalt keerulisem. Peamine põhjus seisneb erinevates modelleerimisparadigmades ning UML keele üldises paindlikkuses, mis ei taga kindlat semantilist tähendust igale klassile või atribuudile.

FHIR standard esindab tugevalt formaliseeritud andmestruktuure. UML klassidiagrammid seevastu on eelkõige visuaalne ja üldotstarbeline modelleerimiskeel, mille abil saab kujutada väga erinevaid süsteeme. Selle tulemusena ei ole UML klasside, omaduste ja seoste tähendus alati üheselt tõlgendatav, mis teeb automaatse teisendamise FHIR-i keeruliseks ja sageli ka mitmetimõistetavaks.

Töö käigus, eeskätt FHIR-st UML-i teisendusskripti loomisel, ilmnes, et mitmed FHIR spetsiifilised atribuudid, nagu *isSummary*, *mapping* ja *slicing.rules* ei ole UML-is otseselt esitatavad või on nende väljendamine äärmiselt keerukas. Nende omaduste korrektne ja semantiliselt täpne käsitlemine eeldaks UML mudeli laiendamist täiendava semantikaga, mis omakorda suurendaks mudeli keerukust, vähendaks loetavust ning kasvataks selle mahtu. Tulenevalt nendest piirangutest osutus FHIR profiili täielik rekonstrueerimine UML esitusest osaliseks ja piiratud ulatusega ülesandeks, mis vajab eraldi käsitlemist ning täiendavat arendustööd.

Kuna teisendusprotsess vajab väga täpset ja täielikku sisendit, tuleb UML mudel esmalt rohkem standardiseerida. Veelgi enam – kuna UML süntaksit kirjutatakse tihti käsitsi tekstifailidena (näiteks PlantUML kujul), võivad lihtsad kirjavead, puuduvad märgid või vale nimetus põhjustada ebaõnnestunud teisendust. Seega tuleb teisendusprotsessis arvestada ka sisendi valideerimise vajadusega: isegi väike viga, nagu vale kirjapilt või vale süntaktiline vorming, võib tekitada tõrkeid, mille põhjus on ilma tööriistadeta raskesti tuvastatav.

3.4.2 Prototüüp: mida on tehtud ja millised on järgmised sammud

Vaatamata keerukusele, autor proovis teha töö raames esimesed katsetused UML mudeli põhjal FHIR-i rekonstrueerimiseks. Selleks loodi algeline parser, mis põhineb regex (*regular expression*) põhistel mustritel ning suudab tuvastada ja töödelda PlantUML süntaksiga UML klasside kirjelduse. Parseri eesmärk on analüüsida UML diagrammi tekstilist kuju ja kaardistada selle sisu võimalikult täpselt tagasi FHIR StructureDefinition elementideks.

Kasutatud regulaaravaldised – vaata lisa 5 – võimaldasid tuvastada järgmisi struktuurseid komponente:

- *CLASS_PATTERN* – tuvastab UML klasside ja struktuuride (*class*, *struct*) deklaratsioonid koos nende kehasse kuuluvate väljadega.
- *FIELD_PATTERN* – tuvastab klasside sees olevad atribuudid, võimaldades eraldada tüübi, nähtavustaseme, vaikimisi väärtused ja kardinaalsuse kasutuse.
- *BINDING_PATTERN* – leiab väärtuste sidumise (*ValueSet*) deklaratsioonid koos vastava kommentaariga.
- *RELATION_PATTERN* – tuvastab klassidevahelised seosed ning nende tüübi ja semantilise tähenduse.
- *CLASS_NAME_PATTERN* – tuvastab klasside nimetused koos tüübi või kontekstiga, kui need on esitatud kujul *Type (Name)*.
- *CLASS_GROUP_PATTERN* – eraldab klassigrupid ja struktuurilised jaotused diagrammis, võimaldades täiendavat konteksti hilisemaks analüüsiks.

Sellise aluse abil suudeti koostada esmased ElementDefinition kirjeldused ning genereerida nende põhjal osaliselt struktureeritud StructureDefinition JSON objektid. Eriti huvitavaks lähenemiseks osutus tekstipõhiste markerite kasutamine – näiteks FHIR-ile iseloomulike *snapshot* ja *differential* vaadete esindamine *bold*-itud märksõnade abil UML diagrammis. Kuigi tegemist on mittestandardse konventsiooniga, pakub see paindliku viisi, kuidas UML tekstis lisada FHIR-i spetsiifilisi tähendusi ilma diagrammi visuaalset loetavust rikkumata.

Samas tuleb rõhutada, et kogu lahendus on väga varajases staadiumis ning toetab ainult piiratud hulka FHIR-i mehhanisme. Enamus keerukamatest reeglitest, vajavad veel eraldi käsitlemist. Samuti eeldab kogu süsteem, et UML on loodud väga kindlate reeglite ja süntaktiliste mustrite järgi – väiksem kõrvalekalle või süntaksiviga võib põhjustada töötlusvigu või katkist väljundit.

Praktikas tähendab see, et lisaks parserile oleks vaja arendada ka valideerimiskiht, mis tuvastaks süntaksivead UML kirjelduses ning annaks kasutajale tagasisidet vea olemuse ja asukoha kohta. Selline lähenemine võiks näiteks hoiatada puuduvate tüüpide, valede kardinaalsuste, tundmatute *binding*-ute või formaadivigade eest, võimaldades seejärel UML-i sisend korrigeerida enne kui seda kasutatakse FHIR profiili genereerimiseks.

3.5 TermX platvormiga integreerimine

3.5.1 REST API loomine

REST API loomise eesmärk oli algusest peale võimaldada käsurea skripti teisendusteenust kasutada veebirakenduste kaudu, pakkudes lihtsat ja fokusseeritud liidest, mis sobiks erinevatesse süsteemidesse integreerimiseks. Kuigi TermX platvorm oli esimene konkreetne integratsioonikeskkond, ei olnud loodud API olemuselt piiratud vaid selle platvormi kasutusega. Teenus kujundati teadlikult võimalikult kergekaaluliseks ja üldotstarbeliseks, et seda saaks kasutada vajadusel ka teistes projektides või süsteemides.

Veebiteenuse arendamiseks valiti Java Spring Boot raamistik. Rakenduse arhitektuur tugines üheleainsale POST tüüpi *endpoint*-ile, kuhu kliendirakendus sai edastada JSON vormingus FHIR StructureDefinition ressursi. Eraldi tähelepanu pöörati HTTP päiste (*headers*) kasutamisele, et võimaldada paindlikumat juhtimist ilma sisendformaati muutmata. Näiteks sai kliendi poolt määrata päisega *Content-Type* soovitud väljundformaadi, valides kas tekstilise PlantUML skripti (*text/plain*), rastergraafika pildina (*image/png*) või vektorvormingus diagrammi (*image/svg+xml*).

Vaate valik toimus *Accept* päise kaudu, kus määrati vastav parameeter – näiteks *Accept: application/json; view=snapshot* või *Accept: application/json; view=differential*. See võimaldas serveril teada, millist StructureDefinition-i esitlusviisi kasutaja soovis teisendada UML diagrammiks.

Lisaks toetati spetsiaalsete HTTP päiste kaudu skripti täiendavate parameetrite seadistamist, mis võimaldasid muuta UML diagrammi genereerimise käitumist vastavalt kasutaja

soovidele. Kasutatavad päised olid järgmised:

- *X-Hide-Removed-Objects* – määrab, kas UML diagrammilt tuleb eemaldada FHIR profiili modifitseerimise käigus kustutatud elemendid.
- *X-Show-Constraints* – määrab, kas UML diagrammil kuvatakse ka FHIR piirangud ja *constraints*.
- *X-Show-Bindings* – võimaldab näidata *bindings* UML diagrammil.
- *X-Reduce-Slice-Classes* – võimaldab vähendada *slice* klasside arvu, koondades neid vajadusel kompaktsemalt ja lihtsustades seeläbi diagrammi struktuuri.
- *X-Hide-Legend* – määrab, kas diagrammiga kaasatakse legend.

Serveri sisemine töövoog oli üles ehitatud nii, et iga saabunud päringu korral käivitati taustaprotsess, mis kasutas käsurea skripti. See protsess edastas skriptile vajalikud sisendid ning käivitas vastava teisenduse, saades tulemuseks soovitud väljundfailid. Pärast eduka vastuse edastamist kliendile tegi server automaatse puhastuse, eemaldades ajutised failid, mis protsessi käigus loodi. Vigade tekkimise korral tagastati kliendile standardiseeritud veateade, mis sisaldas probleemide olemuse selgitust.

3.5.2 Docker

REST API arenduse valmimisel suunati tähelepanu teenuse konteineriseerimisele, et integreerida loodud lahendus sujuvalt TermX platvormi olemasolevasse arhitektuuri.

Loodud teisendusteenuse konteineriseerimine viidi ellu spetsiaalselt välja töötatud CI/CD töövoo abil GitHub Actionsi platvormil. Konteineriks muutmise protsess hõlmas esmalt serverirakenduse ja käsurea-skripti kompileerimist ning nende failide koondamist ühtsesse töökausta, mille alusel moodustati terviklik Docker tõmmis. Töötatud töövoog järgib järgmisi peamisi samme:

- Spring Boot baasil loodud REST serveri kompileerimine.
- Käsurea konverteri kompileerimine
- Mõlema komponendi koondamine ühtsesse konteinerisse.
- Docker tõmmise loomine ja üleslaadimine GitHub Container Registry-sse.

Lisa 6 on toodud täielik GitHub Actions töövoo kirjeldus.

3.5.3 TermX veebirakendusega sidumine

TermX platvormiga lõpliku integreerimise oluline osa hõlmas ka kasutajaliidese täiendamist, et võimaldada FHIR StructureDefinition-i põhjal UML diagrammide genereerimist otse veebirakenduse kaudu. Selleks juurutati spetsiaalne kasutajaliidese komponent, mis võimaldas kasutajal mugavalt saata FHIR-i kirjeldusserverile ning saada vastu soovitud kujul visuaalne või tekstiline väljund.

TermX veebirakendus, mis on üles ehitatud Angulari raamistikule ja kasutab TypeScripti keelt, kohandati vastavalt uuele funktsionaalsusele. Täiendused viidi läbi *Structure Definitions* mooduli lehel, kus loodi uus alamvaade UML diagrammide genereerimiseks. Sellel vaatel said kasutajad määrata teisenduse parameetrid, mis hiljem edastati serverisse HTTP päiste kaudu.

Uuendatud kasutajaliides võimaldas sisestada ka väljundi failinime, mille all genereeritud fail oleks hiljem kergesti allalaaditav. Kasutajaliidese kaudu oli võimalik initsieerida POST päring serverile, mille peale server vastas kas pildina või tekstilise UML kirjeldusega, sõltuvalt kasutaja tehtud valikutest. Kuvamise loogika käsitles automaatselt nii pildifailide kui ka tekstiformaadi tulemusi, tagades kasutajale sujuva ja intuitiivse kogemuse. Lisas 7 on esitatud kasutajaliidese vaated.

Kasutajaliidese täiendused viidi läbi eraldi arendusharus. Pärast arendustöö lõpetamist ja esmaseid lokaalseid teste anti uuendatud kood üle TermX platvormi põhitiimile integreerimiseks ja lõplikuks testimiseks. Testimise käigus kinnitati, et loodud lahendus töötas ootuspäraselt: FHIR profiilidest sai genereerida korrektseid UML diagramme vastavalt kasutaja seatud parameetritele ning tulemused olid stabiilselt kättesaadavad nii visuaal- kui tekstivormis.

4. Tulemused

4.1 UML diagrammide kuvamine TermX platvormil

Üheks oluliseks kasutusstsenaariumiks, kus loodud UML diagrammi genereerimise lahendus TermX platvormil rakendust leiab, on FHIR StructureDefinition tüüpi andmemudelite haldamine ja visualiseerimine.

Genereeritud UML diagrammide funktsionaalsus integreeriti FHIR StructureDefinition-i tasemele, lisades iga mudeli haldusvaatesse spetsiaalse nupu, mille abil kasutaja saab vajadusel käivitada automaatse UML diagrammi loomise protsessi. Selline lähenemine võimaldab kasutajal visuaalselt kiiresti mõista ja analüüsida konkreetse mudeli struktuuri, klassidevahelisi seoseid ning andmeelementide omadusi ilma vajaduseta käsitsi dokumentatsiooni või JSON struktuure läbi töötada.

Praktilisest vaatenurgast tähendab see, et iga FHIR StructureDefinition mudel, mille kasutaja TermX platvormil avab, saab nüüd ühe nupuvajutusega teisendada graafiliseks UML skeemiks, kasutades taustal loodud REST API ja käsurea skripti lahendust.

Sellise lahenduse juurutamine suurendab oluliselt TermX platvormi kasutusmugavust ja andmemudelite haldamise tõhusust, pakkudes kiiret ja visuaalset ülevaadet mudelite struktuurist ning toetades seeläbi nii modelleerimis- kui valideerimisprotsesse.

4.2 Autonoomne command-line skript

Iseseisev käsurea skript töötati välja eesmärgiga pakkuda kasutajatele võimalust FHIR profiilidest UML diagramme genereerida, sõltumata konkreetsest platvormist või keskkonnast. Skript on loodud avatud lähtekoodiga tööriistana GitHub-is[1] ning mis on suunatud kõigile, kes töötavad FHIR andmemudelitega ja vajavad nende struktureeritud visuaalset esitust UML diagrammidena.

Skript on ette nähtud kasutamiseks kahel erineval viisil, pakkudes kasutajatele paindlikkust vastavalt nende tehnilistele eelistustele. Esimene kasutusviis on otsene käsurea-käivitus, kus kasutaja annab skriptile vajaliku sisendfaili FHIR StructureDefinition JSON formaadis ning soovitud parameetrid käsurea argumentidena. Selline lähenemine sobib hästi kiirete

[1] `https://github.com/SpectreH/fhir-uml-converter`

teisenduste jaoks, kus pole vaja täiendavat serveri infrastruktuuri.

Teine võimalus on skripti käivitamine läbi Docker konteineri, mis võimaldab skripti serverina üles seada ja pakkuda REST API liidest. Selle meetodi korral tõuseb üles server, mis kuulab sissetulevaid päringuid ja käitleb neid automaatselt. See variant sobib keskkondadesse, kus on vaja sagedasi ja struktureeritud päringuid või kus soovitakse integreerida teisendusteenus mõne muu rakenduse või süsteemi töövoogu.

Skripti lihtsaks kasutamiseks on projektiga kaasas põhjalik *README.md*, mis kirjeldab samm-sammult kuidas tööriista alla laadida, seadistada ja kasutada nii käsurea kui ka Docker põhise tööviisi korral. Dokumentatsioon sisaldab ka üksikasjalikke juhiseid erinevate konfiguratsioonivõimaluste ja parameetrite kohta, võimaldades kasutajal kohandada väljundit oma vajadustele vastavaks.

4.3 Valideerimine

Arendustöö lõppfaasis viidi läbi loodud teisendusskripti valideerimine, mille eesmärgiks oli veenduda selle korrektses ja ootuspärases toimimises erinevate sisendprofiilide puhul. Valideerimine toimus praktilisel viisil: võeti mitmed reaalsed FHIR StructureDefinition ressursid, sealhulgas Eesti terviseandmete kontekstis kasutatavad FHIR profiilid, mis on avaldatud Eesti riiklikus HL7 FHIR artefaktide repositooriumis [51] kui ka MPI profiilide kogumikus [52] (*Resource Profiles* alajaotuses). Nende põhjal genereeriti UML diagrammid ning võrreldi tulemusi visuaalselt, kontrollides, kas kõik olulised elemendid ja struktuurid olid korrektse kaardistusega diagrammidesse üle kantud. Mõned UML diagrammid võib leida lisas 8.

Valideerimisprotsess kinnitas, et skript suudab usaldusväärselt töödelda erinevaid profiile ja genereerida täpseid UML kujutisi. Kõik kasutatud sisendelemendid said korrektse visuaalse esindatuse ning teisenduse loogika käitus stabiilselt. Süvatasemel automaatset valideerimist (näiteks üksustestide või integreerimistestide näol) ei peetud vajalikuks, kuna skript tugineb otse FHIR JSON StructureDefinition formaadile, mille puhul kehtivad juba väga ranged sisseehitatud standardipõhised valideerimisreeglid. Kui sisendfail ei vasta FHIR-i formaadinõuetele, annab skript sellest koheselt veateate ega jätka töötlemist, vältides seeläbi vigade levikut.

Tulevikus võiks valideerimisprotsessi edasi arendada ka koodipõhiseks automaatseks kontrolliks. Näiteks võrrelda genereeritud UML klasside arvu ja struktuuri originaalse StructureDefinition-iga või kontrollida, kas kõik kohustuslikud väljad on vastavalt spetsifikatsioonile korrektselt visualiseeritud. See suurendaks testimise ulatust ning looks

aluse põhjalikumaks regressioonitestimiseks edasiste arenduste käigus.

4.4 Tulevikulahendused

Tulevikus avaneb teisendusskripti ja sellega seotud lahenduste edasiarendamiseks mitmeid võimalusi. Üheks oluliseks suunaks oleks olemasoleva skripti laiendamine, et see suudaks töödelda veelgi laiemat hulka FHIR ressursitüüpe ja struktuurielemente. Kuigi praegune lahendus katab valdava osa kasutatavatest FHIR StructureDefinition-ist, jäävad mõned keerulisemad või harvem kasutatavad konstruktsioonid väljapoole esialgset haaret. Tulevikus võiks skripti täiendada, et see hõlmaks rohkem erikonstruktsioone ja toetaks veelgi keerukamaid FHIR modelleerimismustreid.

Lisaks sisulisele laiendamisele oleks võimalik arendada edasi ka genereeritavate UML diagrammide visuaalset esitust. Näiteks saaks kasutajale pakkuda võimalust muuta klasside paigutust diagrammil, valida erinevaid esitlusstiile või kohandada diagrammide üldist disaini, et parandada loetavust ja vastavust konkreetsele kasutuskontekstile.

Eraldi ja keerukamaks suunaks tulevikutöös on vastupidine teisendus ehk UML diagrammist FHIR StructureDefinition-i genereerimine. See ülesanne on oluliselt keerulisem, sest nõuab spetsiaalse ja üheselt mõistetava UML süntaksi määratlemist, mis võimaldaks automaatselt ja täpselt taastada FHIR standardi struktuurid. Sellise süntaksi väljatöötamine eeldab põhjalikku analüüsi UML ja FHIR modelleerimispõhimõtete vaheliste seoste osas ning võib osutuda eraldiseisvaks uurimisprojektiks. Lisaks tuleb hinnata, kas sellise vastupidise teisenduse arendamine on praktiliselt vajalik ja tehnoloogiliselt realistlik, arvestades FHIR mudelite keerukust ja nende täpsuse nõudeid.

5. Kokkuvõte

Käesoleva bakalaureusetöö eesmärgiks oli arendada lahendus, mis võimaldab automaatselt teisendada FHIR StructureDefinition formaadis andmemudeleid UML klassidiagrammideks. Töö raames loodi käsureapõhine skript ja REST API teenus, mis tõlgendab FHIR profiile, kaardistades nende struktuuri ja semantilisi omadusi vastavateks UML klassideks, seosteks ja atribuutideks, kasutades PlantUML süntaksit diagrammide genereerimiseks.

Teisenduse teostamiseks kasutati Java põhist HAPI FHIR teeki, Spring Boot raamistikku ja CI/CD töövoogu GitHub Actionsi ja Docker-iga. Loodud lahendus integreeriti TermX platvormi, kus see võimaldab visualiseerida FHIR-põhiseid andmemudeleid graafiliselt, parandades mudelite mõistetavust ja lihtsustades andmevahetuse standardite rakendamist. Töö käsitles ka vastupidist suunda – UML diagrammide põhjal FHIR profiilide loomist – mille jaoks koostati prototüüp ning hinnati võimalikke edasisi arengusuundi.

Töö tulemusel valmis skaleeritav ja laiendatav lahendus, mis toetab terviseandmete struktureeritud käsitlemist ja visualiseerimist erinevates rakendustes, aidates kaasa FHIR standardi rakendamise tõhususele Eestis ja laiemalt.

Kasutatud kirjandus

[1] HL7 International. *FHIR Standard (Fast Healthcare Interoperability Resources)*. [Vaadatud: 31-03-2025]. URL: https://hl7.org/fhir.

[2] Object Management Group. *Unified Modeling Language (UML), v2.5.1. OMG Standard*. [Vaadatud: 31-03-2025]. URL: https://www.omg.org/spec/UML/2.5.1/About-UML.

[3] TermX Platform. *Knowledge Management and Sharing Platform for Healthcare*. [Vaadatud: 31-03-2025]. URL: https://termx.org.

[4] FHI 360. *Fast Healthcare Interoperability Resources (FHIR)*. [Vaadatud: 08-04-2025]. URL: https://www.fhi360.org/wp-content/uploads/2023/12/resource-bhp-fhir.pdf.

[5] HL7 International. *Resources defined as part of FHIR*. [Vaadatud: 01-04-2025]. URL: https://hl7.org/fhir/resourcelist.html.

[6] HL7 International. *Patient - FHIR v5.0.0*. [Vaadatud: 02-04-2025]. URL: https://hl7.org/fhir/R5/patient.html.

[7] Kodjin. *What Are FHIR Profiles: Best Practices & Examples*. [Vaadatud: 01-04-2025]. URL: https://kodjin.com/blog/what-are-fhir-profiles/.

[8] Office of the National Coordinator for Health Information Technology (ONC). *Introduction to FHIR Resources Fact Sheet*. [Vaadatud: 01-04-2025]. URL: https://www.healthit.gov/sites/default/files/page/2021-04/Intro%20to%20FHIR%20Resources%20Fact%20Sheet.pdf.

[9] HL7 International. *Profiling FHIR*. [Vaadatud: 01-04-2025]. URL: https://build.fhir.org/profiling.html.

[10] Harold R. Solbrig et al. "Modeling and Validating HL7 FHIR Profiles Using Semantic Web Shape Expressions (ShEx)". In: *Journal of Biomedical Informatics* 67 (2017). [Vaadatud: 01-04-2025], pp. 90–100. DOI: 10.1016/j.jbi.2017.02.009. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5502481/.

[11] HL7 International. *StructureDefinition - FHIR v6.0.0-ballot2*. [Vaadatud: 01-04-2025]. URL: https://build.fhir.org/structuredefinition.html.

[12] HL7 Estonia. *EEBase Patient - Estonian Base Implementation Guide v1.1.1*. [Vaadatud: 08-04-2025]. URL: https://fhir.ee/packages/ee-base/1.1.1/site/StructureDefinition-ee-patient.html.

[13] HL7 International. *FHIR Data Types*. [Vaadatud: 01-04-2025]. URL: https://hl7.org/FHIR/types.html.

[14] HL7 International. *Resource - FHIR v6.0.0-ballot2*. [Vaadatud: 01-04-2025]. URL: https://www.hl7.org/fhir/resource.html.

[15] HL7 International. *Logical Models - FHIR v6.0.0-ballot2*. [Vaadatud: 01-04-2025]. URL: https://build.fhir.org/logical.html.

[16] HL7 International. *ElementDefinition - FHIR v6.0.0-ballot2*. [Vaadatud: 01-04-2025]. URL: https://build.fhir.org/elementdefinition.html.

[17] HL7 International. *RelatedPerson - FHIR v5.0.0*. [Vaadatud: 02-04-2025]. URL: http://hl7.org/fhir/relatedperson.html.

[18] Data Standards Wales. *Profile Descriptions - FHIR Standards Wales Implementation Guide*. [Vaadatud: 02-04-2025]. URL: https://simplifier.net/guide/fhir-standards-wales-implementation-guide/Home/Introduction/Profile-Descriptions?version=1.0.0.

[19] Miguel Olivero et al. "Facilitating the design of HL7 domain models through a model-driven solution". In: *BMC Medical Informatics and Decision Making* 20 (May 2020), p. 104. DOI: 10.1186/s12911-020-1093-4.

[20] HL7 International. *UML Definition - FHIR v6.0.0-ballot2*. [Vaadatud: 04-04-2025]. URL: https://build.fhir.org/uml.html.

[21] Jens Fudickar. *json2puml: Generate PlantUML files based on JSON files*. [Vaadatud: 04-04-2025]. URL: https://github.com/jfudickar/json2puml.

[22] SOM Research. *JSON Discoverer: Tool for Discovering Implicit Schemas in JSON Documents*. [Vaadatud: 04-04-2025]. URL: https://github.com/SOM-Research/jsonDiscoverer.

[23] Dassault Systèmes. *No Magic - Standard-based Modeling Solutions*. [Vaadatud: 04-04-2025]. URL: https://www.3ds.com/products/catia/no-magic.

[24] Sparx Systems. *Enterprise Architect: UML Modeling Tool*. [Vaadatud: 04-04-2025]. URL: https://sparxsystems.com.

[25] Modeliosoft. *Modelio: Open Source Modeling Environment*. [Vaadatud: 04-04-2025]. URL: https://www.modelio.org/.

[26] Ltd. MKLabs Co. *StarUML – A sophisticated software modeler*. [Vaadatud: 08-04-2025]. URL: https://staruml.io.

[27] Visual Paradigm International Ltd. *Visual Paradigm – The #1 Development Tool Suite*. [Vaadatud: 08-04-2025]. URL: https://www.visual-paradigm.com.

[28] JGraph Ltd. *diagrams.net – Free Online Diagram Software*. [Vaadatud: 08-04-2025]. URL: https://www.diagrams.net.

[29] Arnaud Roques. *PlantUML – Open-Source UML Diagram Generator*. [Vaadatud: 08-04-2025]. URL: https://plantuml.com.

[30] Knut Sveidqvist and contributors. *Mermaid – Generation of diagrams like flowcharts or sequence diagrams from text in a similar manner as markdown*. [Vaadatud: 08-04-2025]. URL: https://mermaid.js.org.

[31] Daniel Mackay. *Software Diagrams - Plant UML vs Mermaid*. [Vaadatud: 08-04-2025]. URL: https://www.dandoescode.com/blog/plantuml-vs-mermaid.

[32] TermX Contributors. *Architecture - TermX tutorial*. [Vaadatud: 10-04-2025]. URL: https://tutorial.termx.org/en/architecture.

[33] Igor Bossenko et al. "TermX: The Semantic Interoperability, Knowledge Management and Sharing Platform". In: *ResearchGate* (Sept. 2024). [Vaadatud: 10-04-2025]. URL: https://www.researchgate.net/publication/383635890_TermX_The_semantic_interoperability_knowledge_management_and_sharing_platform.

[34] HL7 Estonia. *EEBase Organization - Estonian Base Implementation Guide v1.1.1*. [Vaadatud: 10-04-2025]. URL: https://build.fhir.org/ig/HL7EE/ig-ee-base/StructureDefinition-ee-organization.html.

[35] HL7 International. *Open Source Implementations*. [Vaadatud: 10-04-2025]. URL: https://confluence.hl7.org/display/FHIR/Open+Source+Implementations.

[36] Gregor Lichtner. *fhir.resources: FHIR Resources Release R5*. [Vaadatud: 10-04-2025]. URL: https://github.com/glichtner/fhir.resources.

[37] Md Nazrul Islam. *fhir.resources: FHIR Resources as Model Classes*. [Vaadatud: 10-04-2025]. URL: https://pypi.org/project/fhir.resources/.

[38] G. S. Brauer, S. M. Winden, and T. A. Wetter. “The HL7 Fast Healthcare Interoperability Resources (FHIR) standard: Strengths, weaknesses, and future directions”. In: *Journal of Biomedical Informatics* 116 (2021). [Vaadatud: 10-04-2025], p. 103734. DOI: 10.1016/j.jbi.2021.103734. URL: https://www.sciencedirect.com/science/article/pii/S1532046421000848.

[39] Smile Digital Health. *HAPI FHIR – The Open Source FHIR API for Java*. [Vaadatud: 10-04-2025]. URL: https://hapifhir.io.

[40] HAPI FHIR Project Contributors. *Changelog: 2025 - HAPI FHIR Documentation*. [Vaadatud: 13-04-2025]. URL: https://hapifhir.io/hapi-fhir/docs/introduction/changelog.html.

[41] Spring Team. *Building an Application with Spring Boot*. [Vaadatud: 13-04-2025]. URL: https://spring.io/guides/gs/spring-boot.

[42] Amazon Web Services. *What is Docker?* [Vaadatud: 13-04-2025]. URL: https://aws.amazon.com/docker/.

[43] Kapila Asanga Dias. *Transforming Simplified Requirement into a UML Use Case Diagram Using an Open Source Tool*. [Vaadatud: 08-04-2025]. URL: https://www.researchgate.net/publication/315809182_Transforming_Simplified_Requirement_in_to_a_UML_Use_Case_Diagram_Using_an_Open_Source_Tool.

[44] PlantUML. *Use Case Diagram Syntax and Features*. [Vaadatud: 08-04-2025]. URL: https://plantuml.com/use-case-diagram.

[45] Mermaid Contributors. *About Mermaid*. [Vaadatud: 08-04-2025]. URL: https://mermaid.js.org/intro/.

[46] HAPI FHIR Project Contributors. *Parsing and Serializing - HAPI FHIR Documentation*. [Vaadatud: 22-04-2025]. URL: https://hapifhir.io/hapi-fhir/docs/model/parsers.html.

[47] HL7 International. *StructureDefinition - FHIR v6.0.0-ballot2*. [Vaadatud: 22-04-2025]. URL: https://build.fhir.org/structuredefinition-definitions.html.

[48] GeeksforGeeks. *Factory Method Design Pattern in Java*. [Vaadatud: 22-04-2025]. URL: https://www.geeksforgeeks.org/factory-method-design-pattern-in-java/.

[49] HL7 Estonia. *EEMPISocialHistoryMaritalStatus - Resource Profile*. [Vaadatud: 22-04-2025]. URL: https://fhir.ee/ig/mpi/1.1.1/site/StructureDefinition-ee-mpi-socialhistory-marital-status.profile.json.html.

[50] HL7 Estonia. *EEMPIPatient - Resource Profile*. [Vaadatud: 02-05-2025]. URL: https://fhir.ee/ig/mpi/1.1.1/site/StructureDefinition-ee-mpi-patient.html.

[51] HL7 Estonia. *Estonian Base Implementation Guide - Artifacts Summary*. [Vaadatud: 02-05-2025]. URL: https://build.fhir.org/ig/HL7EE/ig-ee-base/branches/master/artifacts.html.

[52] HL7 Estonia. *Master Patient Index - Artifacts Summary*. [Vaadatud: 02-05-2025]. URL: https://fhir.ee/ig/mpi/1.1.1/site/artifacts.html.

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

Mina, Denni Karin

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose "ANDMEMUDELI TEISENDAMINE FHIR STRUCTUREDEFINITION JA UML FORMAATIDE VAHEL", mille juhendaja on Igor Bossenko
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

04.06.2025

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

Lisa 2 – Näide EEBasePatient profiili StructureDefinition määratlusest

```
{
    "resourceType": "StructureDefinition",
    "id" : "ee-patient",
    ...
    "url" :
    "https://fhir.ee/base/StructureDefinition/ee-patient",
    "version" : "1.1.1",
    "name" : "EEBasePatient",
    "title" : "EEBase Patient",
    ...
    "kind": "resource",
    "abstract": false,
    "type": "Patient",
    "baseDefinition":
    "http://hl7.org/fhir/StructureDefinition/Patient",
    ...
}
```

Lisa 3 – Näide RelatedPerson profiili ühe ElementDefinition määratlusest

```
{
    "resourceType" : "StructureDefinition",
    "url" :
    "http://hl7.org/fhir/StructureDefinition/RelatedPerson",
    "version" : "5.0.0",
    ...
    "id" : "Patient",
    "snapshot" : {
        "element" : [
        ...
        {
            "id" : "RelatedPerson.patient",
            "path" : "RelatedPerson.patient",
            "short" : "The patient this person is related to",
            "definition" : "The patient this person is...",
            "min" : 1,
            "max" : "1",
            "base" : {
                "path" : "RelatedPerson.patient",
                "min" : 1,
                "max" : "1"
            },
            "type" : [{
            "code" : "Reference",
            "targetProfile" : [
                "http://hl7.org/fhir/StructureDefinition/Patient"
            ]
        }],
    ...
}
```

Lisa 4 – FHIR ametlik slice puuvaade EEMPIPatient profiili põhjal

Joonis 16. *EEMPIPatient* profiili viilude puuvaade

Lisa 5 – UML-ist FHIR-i teisenduseks kasutatud regulaaravaldised

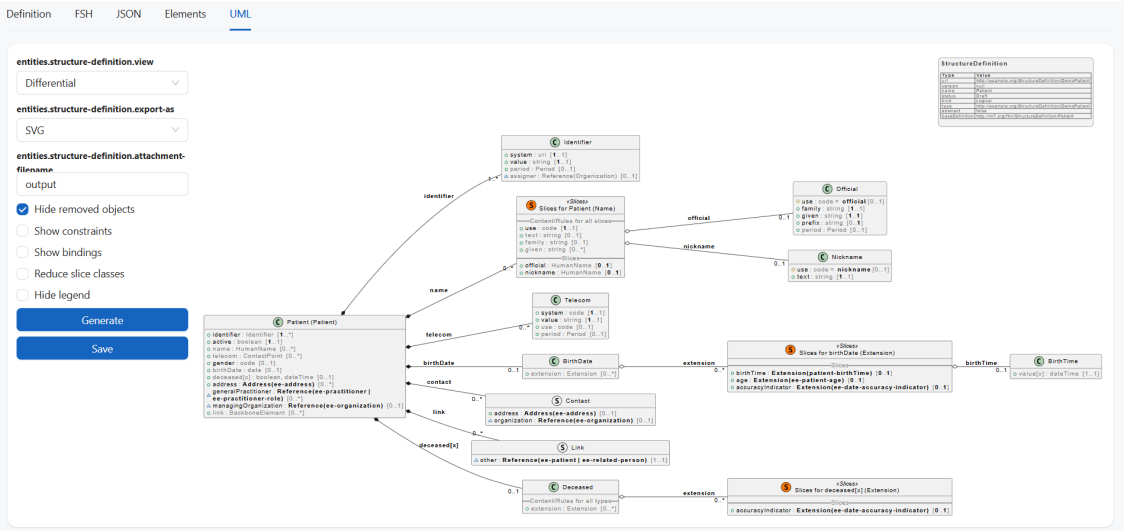
```
private static final Pattern CLASS_PATTERN = Pattern.compile(
        "(?sm)(class|struct)\\s+\"([^\"]+)" +
            "\"\\s*(<<.*?>>)?\\s*\\{([\\s\\S]*?)(?=^[}]\\s*$)",
        Pattern.MULTILINE
);

private static final Pattern FIELD_PATTERN = Pattern.compile(
        "(?m)^\\s*\\{([^}]*)}\\s+([+\\-~#])" +
            "\\s+([^:\\s]+)\\s*:\\s*([^\\[=\\n]+)" +
            "(?:\\s*=\\s*\\*\\*([^*]+)\\*\\*)?" +
            "(?:\\s*\\[([^]]*)])?" +
            "(?:\\s*<<([^>]+)>>)?\\s*$"
);

private static final Pattern BINDING_PATTERN = Pattern.compile(
        "(?m)^\\s*\\*\\* Binding \\*\\*:" +
            "\\s*([^{]+)\\{([^}]+)}:\\s*//(.*?)//\\s*$"
);

private static final Pattern RELATION_PATTERN = Pattern.compile(
        "(?m)^\"([^\"]+)\"\\s*(--\\S+)" +
            "\\s*\"([^\"]+)\"\\s*\"([^\"]+)\"\\s*" +
            ":\\s*\\*\\*([^*]+)\\*\\*\\s*$"
);

private static final Pattern CLASS_NAME_PATTERN = Pattern.compile(
        "^([^\"]+)\\s*\\(([^)]+)\\)$"
);

private static final Pattern CLASS_GROUP_PATTERN = \\
Pattern.compile("(?m)^\\s*--(.*?)--\\s*$");
```

Lisa 6 – CI/CD töövoog ja konteineriseerimine GitHub Actions platvormil

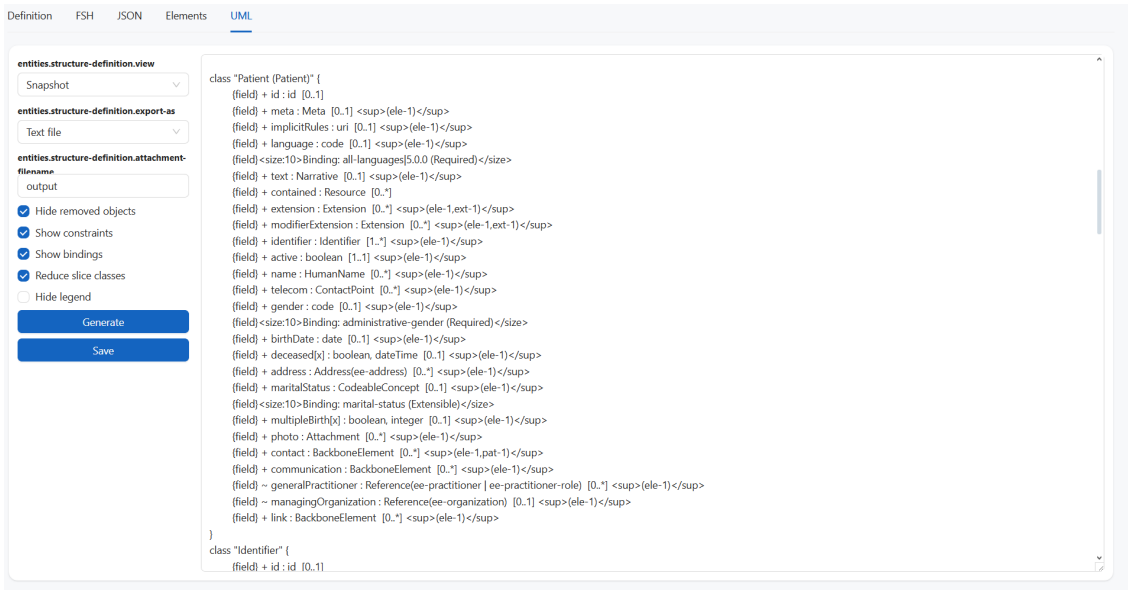
```
name: Build ARM + AMD
on:
  push:
    branches: [ "main" ]
    tags: [ '*.*.*' ]
  workflow_dispatch:
env:
  REGISTRY: ghcr.io
  IMAGE_NAME: fhir-uml-server
jobs:
  build:
    runs-on: ubuntu-latest
    permissions:
      packages: write
      contents: read
    steps:
      - name: Checkout sources
        uses: actions/checkout@v4

      - name: Set up Temurin 21 and Gradle cache
        uses: actions/setup-java@v4
        with:
          distribution: temurin
          java-version: 21
          cache: gradle

      - name: Build server JAR
        working-directory: ./server
        run: |
          chmod +x ./gradlew
          ./gradlew build --no-daemon

      - name: Build converter JAR
        working-directory: ./converter
        run: |
          chmod +x ./gradlew
```

```
      ./gradlew build --no-daemon

    - name: Set up QEMU
      uses: docker/setup-qemu-action@v3

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v3

    - name: Log in to GHCR
      uses: docker/login-action@v3
      with:
        registry: ${{ env.REGISTRY }}
        username: ${{ github.actor }}
        password: ${{ secrets.GITHUB_TOKEN }}

    - name: Set latest tag
      if: github.ref == 'refs/heads/main'
      run: echo "LATEST_TAG=latest" >> $GITHUB_ENV

    - name: Docker meta
      id: meta
      uses: docker/metadata-action@v5
      with:
        images: ${{ env.REGISTRY }}/$
         // {{ github.repository_owner }}/${{ env.IMAGE_NAME }}
        tags: |
          type=semver,pattern={{version}}
          type=semver,pattern={{major}}.{{minor}}
          type=raw,value=${{ env.LATEST_TAG }}
        flavor: |
          latest=false

    - name: Build & push multi-arch image
      uses: docker/build-push-action@v6
      with:
        context: .
        file: ./Dockerfile
        push: true
        tags: ${{ steps.meta.outputs.tags }}
        labels: ${{ steps.meta.outputs.labels }}
        platforms: linux/amd64,linux/arm64
```

Lisa 7 – Valminud kasutajaliidese vaated

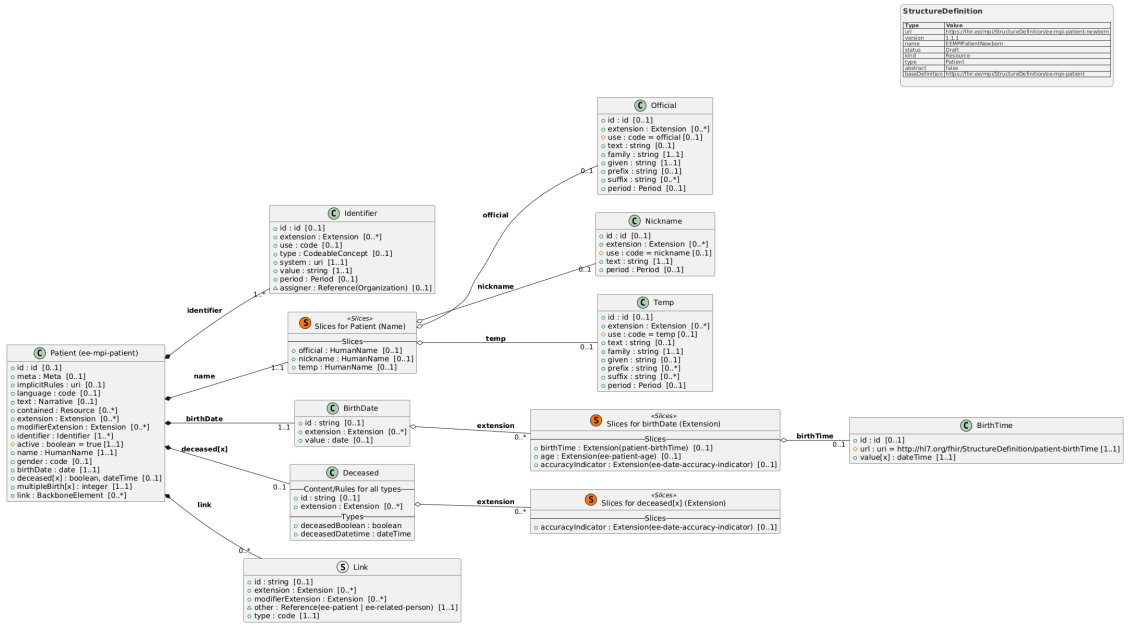


Joonis 17. TermX UML generaatori kasutajaliides – *differential* vaade.

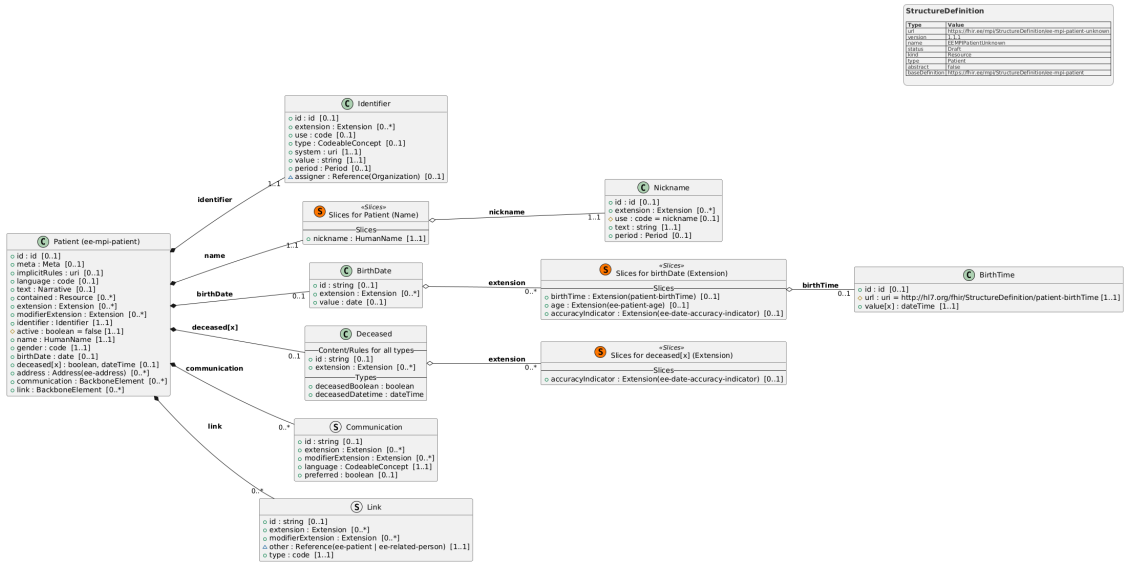


Joonis 18. TermX UML generaatori kasutajaliides – *snapshot* vaade (tekstifail).

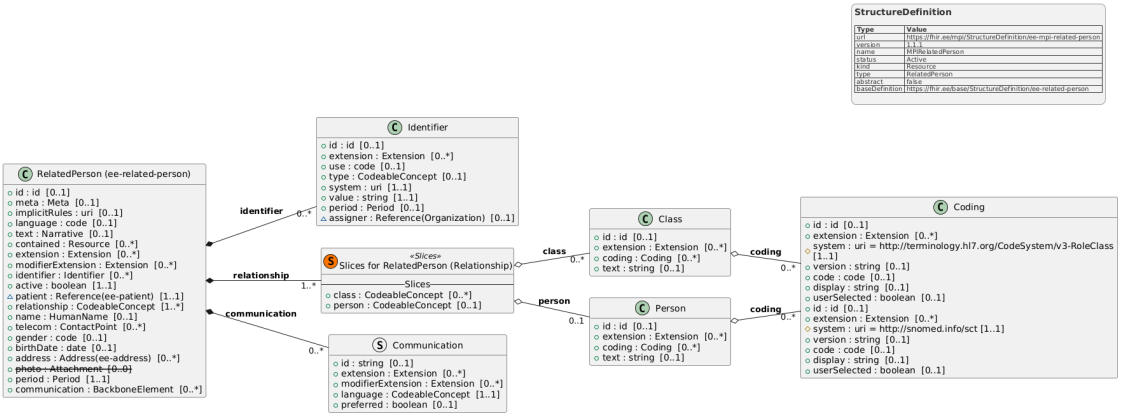
Lisa 8 – Genereeritud UML diagrammid



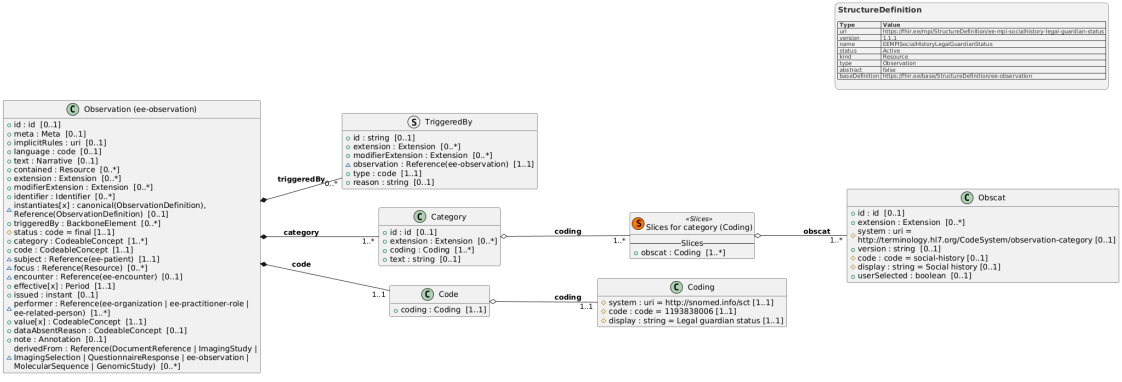
Joonis 19. UML klassidiagramm – *EEMPIPatientNewborn* profiil.



Joonis 20. UML klassidiagramm – *EEMPIPatientUnknown* profiil.



Joonis 21. UML klassidiagramm – *MPIRelatedPerson* profiil.



Joonis 22. UML klassidiagramm – *EEMPISocialHistoryLegalGuardianStatus* profiil.