Heikki Ketoharju

Jaettua kieltä etsimässä – GraphQL-rajapinta tietomallin kohentelun välineenä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto– ja viestintätekniikka

Insinöörityö

1.12.2021

Tiivistelmä

Tekijä:	Heikki Ketoharju
Otsikko:	Jaettua kieltä etsimässä – GraphQL-rajapinta tietomallin kohentelun välineenä
Sivumäärä:	43 sivua
Aika:	1.12.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto– ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Vesa Ollikainen Development Lead Pasi Nissinen

---

Insinöörityössä pyrittiin kohentamaan ohjelmiston tietomallia kehittämällä GraphQL-rajapinta sovellusaluevetoisen suunnittelun keinoin. Tavoitteena oli parantaa ohjelmiston tietomallia ja luoda työprosessi tietomallin parantelemiseen.

Työn kuluessa arvioitiin GraphQL-rajapinnan soveltuvuutta tietomallin parantamisen välineeksi ja tutkittiin, kuinka hyvin teknologia soveltuu sovellusaluevetoisen suunnittelun välineeksi. Tavoitteena oli hahmottaa GraphQL:n rakenteen ja sovellusaluevetoisen suunnittelun yhtymäkohtia.

Insinöörityössä laadittiin pieni prototyyppisovellus, jonka kautta tietomallin parantamisen prosessia kehitettiin ja GraphQL-teknologiaa testattiin. Tämä prototyyppi toimi testialustana sovellusaluevetoisen suunnittelun käytäntöjen ja sovellusalan käsitteiden ymmärtämiseen.

GraphQL soveltuu hyvin sovellusaluevetoisen suunnittelun työkaluksi. Sen tapa esittää rajapinnan takana oleva järjestelmä olioiden verkkona ja sen ohjelmointikielistä riippumaton täsmäkielimuotoinen toteutus tekevät siitä hyvän teknologian vanhan järjestelmän tietomallin kohenteluun.

Insinöörityön tuloksena syntyi Notkean tietomallin paranteluksi nimetty työmalli, jonka avulla tietomallia voi selkeyttää. Pääperiaate on, että sanat, kaaviot ja koodi ovat kolme tapaa kommunikoida tietomallin sisältämiä ideoita kehittäjien ja sovellusalan asiantuntijoiden välillä.

Työn tuloksena syntyneen työmallin avulla on mahdollista rakentaa iäkkään ohjelmiston tietorakennetta parantava GraphQL-rajapinta.

Avainsanat: GraphQL, sovellusaluevetoinen suunnittelu, rajapinta, olioverkko, extreme programming

Abstract

Author:	Heikki Ketoharju
Title:	Seeking for a shared language - GraphQL as a tool for improving data model
Number of Pages:	43 pages
Date:	1 December 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Software Engineering
Supervisors:	Vesa Ollikainen, Senior Lecturer Pasi Nissinen, Development Lead

---

The aim of the study was to enhance the data model of software by developing a GraphQL interface with the means of Domain Driven Design. The goal was both to enhance the data model and to create a process for enhancing the data model.

During the study GraphQL was evaluated as a means of enhancement for the data model. It was researched how well this technology suits as a tool for domain driven development. The goal was to understand the confluences of Domain Driven Design and the structure of GraphQL API.

In the study a small prototype software was developed where the nature of GraphQL was tested and the process for enhancing the data model was developed. The prototype was an environment for understanding the practises of Domain Driven Design and the concepts of a domain.

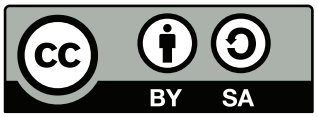
GraphQL suits well as a tool for Domain Driven Design. The way how it presents the system behind the API as an object graph and its programming language independent implementation in a form of domain specific language make it a good technology for enhancing the data model of a legacy system.

As a result, a working model for clarifying the data model was created, called *Notkea tietomallin parantelu*. The primary principle is that words, diagrams and code are three ways to communicate ideas included in the data model, between developers and domain experts.

With the resulting working model it is possible to build a GraphQL API that enhances the data model of a legacy system.

Keywords: GraphQL, Domain Driven Design, API, object graph, extreme programming

Licenses



Sisällys

Lyhenteet

LAMP: LAMP (Linux, Apache, MySQL, PHP).

Sanasto

Aggregaatti: Useiden olioiden kokoelma, jolla on yksi juuriolio. Käsitellään ohjelmalogiikassa yhtenä kokonaisuutena.

Assosiaatiotaulu: Engl. Map. suom. myös *hakurakenne* ja *sanakirja*. Tietorakenne, joka sisältää joukon avain-arvo -pareja.

GraphQL: Facebookin kirjoittama kyselykieli, joka esittää manipuloitavan datan olioverkkona.

HTTP: Hypertext Transfer Protocol. Webin perusprotokolla, jota käytetään hypertekstidokumenttien siirtoon.

Kaikenkattava kieli: Kokoelma käsitteitä ja sanastoa, joiden avulla ohjelmoijat ja sovellusalueen asiantuntijat keskustelevat kehitettävästä ohjelmistosta. Muotoutuu tiedon rouhintaprosessin tuloksena. Engl. Ubiquitous language.

Kulkusuunta: Olioita yhdistävän kytköksen suunta.

Liiketoimintalogiikan taso: Ohjelmiston sisäinen osa, joka pitää sisällään liiketoimintaan liittyvän ohjelmalogiikan. Engl. Domain layer.

ORM: Object-relational mapper, oliot tietokantatauluiksi muuntava teknologia.

Puhdas funktio: Funktio, jonka palauttama tulos riippuu vain sen saamien parametrien arvoista. Puhdas funktio ei aiheuta sivuvaikutuksia esimerkiksi tulostamalla ruudulle tai tallentamalla tiedostoon.

Repositorio: Rajapinta tiedon tallennusvälineeseen.

Sovellusaluemalli: Käsitteistä ja niiden välisistä suhteista koostuva, ohjelmakoodin muotoon kaapattava malli käsiteltävästä sovellusalueesta. Engl. Domain Model.

Sovellusaluevetoinen suunnittelu: Menetelmä, jossa ohjelmistoa kehitetään läheisessä yhteistyössä sovellusalueen asiantuntijoiden kanssa. (myös: Liiketoimintavetoinen suunnittelu.) Engl. Domain Driven Design.

Tiedon rouhinta: Prosessi, jossa tunnistetaan sovellusalan erityiskäsitteitä ohjelmoijan ja sovellusalueen asiantuntijan välisen kommunikaation avulla. Engl. Knowledge crunghing.

Tyyppi: Lausekkeeseen sidottu määritys lausekkeen palauttaman arvon tietotyypistä. Esimerkiksi kokonaisluku tai merkkijono.

Täsmäkieli: Erityisalaan liittyvä formaali kieli. Esimerkiksi ohjelmointi- tai kuvauskieli. Engl. Domain-Specific language.

Verkko: Tietorakenne, joka koostuu solmuista ja kaarista.

Yksilötyyppi: Ohjelmassa esiintyvä olio, jolla on identiteetti. Esimerkiksi yksittäinen asiakas tai lasku. Engl. Entity.

1 Johdanto

Pitkäikäinen, jatkuvasti kehitetty ohjelma mutkistuu herkästi. Jos kehitystiimi ei pidä varaansa, alun perin yksinkertaisesta ja selkeästä ohjelmakoodista ja suoraviivaisesta tietorakenteesta kasvaa hankalasti hallittava kokonaisuus.

Sotkuiseenkin ohjelmistoon voi kuitenkin saada selvyyttä taitavalla kohentelulla ja kehittämisellä. Kun toimintoja alkaa jakaa osiin ja rakentaa osien välille selkeitä nivelkohtia, avautuu mahdollisuus tehdä hedelmällisiä parannuksia.

Web-sovelluksessa eräs merkittävimpiä nivelkohtia on sovelluksen käyttöliittymän ja logiikkaosan välinen HTTP-rajapinta. Tässä insinöörityössä tarkastellaan, voiko tämän merkittävän nivelkohdan uudistamisen kanssa yhtä aikaa myös parantaa ohjelmiston sisäistä logiikkaa.

Työssä perehdytään GraphQL-rajapinnan toimintaan Domain Driven Designin näkökulmasta. Tuloksena syntyy työmalli, jonka avulla ohjelmiston tietomallia voidaan kohentaa.

Työ tehdään ohjelmistopalveluyritys Nordhealth Oy:n tilauksesta.

2 Lähtökohdat ja tavoitteet

2.1 Nordhealth Oy lyhyesti

Nordhealth Oy on vuonna 2001 perustettu ohjelmistopalveluyritys, joka tekee toiminnanohjausjärjestelmiä kahdelle eri toimialalle: Provet Cloud [1] -järjestelmää eläinlääkäriklinikoille ja Diarium-järjestelmää [2] terapeuteille. Molemmat järjestelmät ovat web-pohjaisia sovelluksia. Niitä siis käytetään web-selaimen kautta.

Nordhealthia voisi luonnehtia tyypilliseksi ohjelmistoalan yritykseksi: se tekee

erityisalalle suunnattua toiminnanohjausjärjestelmää. Tyypillinen on myös järjestelmien kehityskaari: alunperin ne ovat olleet työpöytäsovelluksia, ja siitä Nordhealth on muuntanut ne LAMP (Linux, Apache, MySQL, PHP)-alustalla toimiviksi web-sovelluksiksi. Kun Web on kehittynyt, on otettu suunnaksi sovellusten siirtäminen julkiseen pilveen ja käyttöliittymän rakentaminen erilliseksi yhden sivun JavaScript-sovellukseksi.

Diarium on Nordhealthin kahdesta järjestelmästä vanhempi. Se on suunnattu terapeuteille: fysio-, toiminta-, puhe- ja psykoterapeuteille. Järjestelmä on laajentunut yksinkertaisesta potilaskortistojärjestelmästä suurenkin terapia-alan yrityksen tarpeita vastaavaksi toiminnanohjausjärjestelmäksi, ja se on lisäksi myös Valviran tarkoittama A-luokan potilastietojärjestelmä. Diariumia käyttävä terapeutti voi siis lähettää käyntikirjaukset potilastiedon sähköiseen Kanta-rekisteriin.

2.2 Tarve työlle

Diariumin ikä näkyy tietomallin monimutkaisuutena. Vuosien saatossa tehty kehitystyö on tehnyt ohjelmiston joistain osista hankalia ymmärtää. Ohjelman kehittäminen on myös aloitettu aikana, jolloin sovelluksia ei automaattisesti tehty yhden sivun sovelluksiksi.

Nykyään tämä kuitenkin on normi, ja ohjelmaan on jo vuosia kehitetty REST-rajapintaan nojaavia toiminnallisuuksia. Samalla kehitystyön myötä on huomattu, että REST soveltuu huonosti joihinkin monimutkaisen toiminnanohjausjärjestelmän vaatimiin tehtäviin.

Olisiko REST-rajapinnalle sopivampi vaihtoehto? Entä voisiko rajapintaa rakentaessa myös parantaa ohjelmiston sisäistä tietomallia? Tämä säästäisi aikaa ja vaivaa, ja nopeuttaisi ohjelmiston jatkokehitystä.

2.3 Insinöörityön tavoite

Tämän insinöörityön tavoitteena on selvittää, onko ohjelmiston tietomallia mahdollista parannella rakentamalla GraphQL-rajapinta. Lisäksi työn myötä on tavoitteena löytää parempi tietomalli osaan Diarium-sovellusta.

Kolmas tärkeä tavoite on kehittää työmenetelmä, jonka avulla tietomallia on mahdollista korjata. Näin työssä tehtyjä havaintoja on mahdollista hyödyntää jatkossa.

Projektin edetessä rakennan pienen GraphQL-rajapinnan ja sitä hyödyntävän prototyyppisovelluksen. Tämä toimii kokeilukenttänä, jonka kautta työmenetelmää ja tietomallia etsitään.

3 Työn teoriataustaa

3.1 Sovellusaluevetoinen suunnittelu

Yleinen ongelma tietokoneohjelmistoja tehtäessä on, että ohjelmoijat tuntevat ohjelmiston erikoisalan heikosti. Esimerkiksi kiinteistötekniikkaa, kirjastokortistoa tai tämän työn tapauksessa terapiaklinikan toimintaa hoitavan ohjelmiston kehittäjä joutuu käsittelemään monimutkaisia, sovellusalaan sidottuja ongelmia. Näiden alojen asiantuntijat puolestaan tietävät, miten sovellusalan ongelmat ratkaistaan, mutta heiltä puuttuu taito suunnitella ohjelmistoja.

Tämän ongelman ylittäminen on aihe, jota Eric Evans käsittelee kirjassaan Domain Driven Design[3]. Evansin mielestä jokaisen monimutkaisemman ohjelmiston sisässä on sovellusaluemalli (Domain model), eli malli siitä, miten kyseinen ohjelmisto ratkaisee sovellusalan ongelmat. Malli voi kuitenkin olla piilossa ohjelmakoodin sisällä, ja saattaa olla, etteivät ohjelmiston kehittäjät edes tiedosta mallin olemassaoloa. Tämän mallin tuominen näkyväksi on sovellusaluevetoisen suunnittelun päätavoite.

Tiedon rouhinta (Knowledge crunching) on keskeinen väline sovellusaluemallin rakentamiseen. Evans kuvaa prosessin, jossa kehittäjät luonnostelevat yhdessä sovellusalueen asiantuntijoiden kanssa sovellusaluemallin. Tämä prosessi on tyypillisesti kehittäjien vetämä, ja se koostaa yhteen informaatiota monenlaisista lähteistä: sovellusalueen asiantuntijoilta, järjestelmän käyttäjiltä, ohjelmoijien aiemmasta kokemuksesta jne.

Malli kytketään tiiviisti yhteen ohjelmakoodin kanssa vuorottelemalla suunnittelun ja ohjelmistokehityksen välillä. Varhaiset prototyyppiversiot ohjelmistosta toimivat jatkosuunnittelun materiaalina. Kun prosessia jatketaan, malli muuttuu ja syvenee. Evansin mukaan tämä on seurausta siitä, että ohjelmoijien ymmärrys sovellusalueesta syvenee. [3.]

Tavoitteena on, että ohjelmistokehittäjien ja alan asiantuntijoiden välille rakentuu kaikenkattava kieli, jonka avulla kaikkien on mahdollista yhteisesti keskustella ohjelmiston toiminnasta ja kehitystarpeista. Kaikenkattava kieli yhdistää ja yhdenmukaistaa myös ohjelmiston eri osien parissa toimivien kehittäjien kielenkäyttöä.

Kaikenkattavan kielen käsitteitä ovat luokkien nimet ja ohjelmiston keskeiset operaatiot. Se myös sisältää käsitteet, joilla voidaan kuvata käsitteiden käyttöä rajoittavat ja hallinnoivat keskeiset säännöt. Kaikenkattavan kielen käsitteet elävät ohjelmakoodissa ja muodostavat koodin ytimessä sijaitsevan sovellusaluemallin.

Evans painottaa, että kaikenkattavaan kieleen tehtävä muutos muuttaa myös sovellusaluemallia ja ohjelmakoodia.

Koodina kuvattu sovellusaluemalli tulee myös eriyttää ohjelmassa omaksi arkkitehtuuriseksi tasokseen. Onnistuneesti erotettu sovellusalakohtainen logiikka muodostaa sovelluksen liiketoimintalogiikan tason (Domain layer).

3.1.1 Sovellusaluevetoisen suunnittelun rakennuspalikat

Koska Evansin lähtökohta on, että sovellusaluemalli ilmaistaan nimenomaan ohjelmakoodin kautta, tarvitaan joukko käytännön työkaluja, joiden avulla sovellusaluemalli on mahdollista toteuttaa teknisesti. Käyn seuraavaksi läpi muutamia keskeisimpiä.

Yksilötyyppi edustaa käytännössä kaikkea, jolla on identiteetti. Esimerkiksi kahdella ihmisellä voi olla sama nimi, mutta he ovat silti identiteetiltään eri henkilöitä. Yksilötyyppi voi myös muuttaa muotoaan. Esimerkiksi ihminen kasvaa aikuiseksi ja vaikkapa vaihtaa sukunimensä. Identiteetti säilyy silti. Todella suuri osa sovellusaluemallista koostuu juuri yksilötyypeistä, sillä niitä on suurin osa kaikenkattavan kielen käsitteistä.

Esimerkiksi terapeutin kahdesta eri hoitokäynnistä luomilla laskuilla on identiteetti. Kahdella laskulla voi olla sisällytettynä samat hoitotoimenpiteet ja maksajakin voi olla sama, mutta silti laskut ovat omia erillisiä kokonaisuuksiaan.

Käsitteiden lisäksi tärkeä osa mallia ovat käsitteiden väliset suhteet. Tosielämässä kaikki liittyy kaikkeen, mutta sovellusalueen sisäinen logiikka tarvitsee vain osajoukon kaikista suhteista toimiakseen. Huolellisesti valitut suhteet käsitteiden välillä paljastavat hyödyllistä tietoa mallin perimmäisestä tarkoituksesta.

Myös kulkusuunta yksilötyyppien välillä on tärkeä. Se vaikuttaa paitsi ohjelmiston tekniseen monimutkaisuuteen, myös siihen, minkälaisia asioita mallilla on mahdollista ilmaista.

Esimerkiksi käynnin ja laskun välinen suhde voi olla yksi- tai kaksisuuntainen, ja tämä vaikuttaa huomattavasti ohjelmiston toimintaan.

- Tapaus A: käynniltä löytyy tieto, mille laskulle se on lisätty, mutta lasku ei osaa kertoa, mikä käynti sillä on laskutettuna.
- Tapaus B: lasku tietää, mikä käynti sille on lisätty, mutta käynniltä ei voi kysyä, millä laskulla se on.

Kolmas vaihtoehto on mahdollistaa kulkeminen molempiin suuntiin näiden kah-

den käsitteen välillä. Tällöin malli on monipuolisempi, mutta myös ohjelman tekninen monimutkaisuus kasvaa. Mikäli käsitteiden väliset suhteet on toteutettu teknisesti viittauksina olioiden välillä, viittauksia täytyy päivittää mahdollisesti kahteen eri paikkaan, ja lisäksi varoa, ettei viittauksista muodostu kehämäisiä. Onkin tärkeää pyrkiä rajoittamaan olioiden väliset suhteet ja kulkusuunnat mahdollisimman vähäisiksi, jotta ohjelma pysyy teknisesti mahdollisimman yksinkertaisena.

Toisinaan sovellusalueeseen kuuluvat oliot muodostavat ryhmiä, joissa yksi olio on juuriolio, ja muut riippuvat tästä. Tällaisia ryhmiä Evans kutsuu nimellä aggregaatti. Esimerkiksi laskurivi on hyödytön ilman laskua, jolle se kuuluu. Lasku ja laskurivi muodostavatkin aggregaatin - kokonaisuuden, jota käsitellään aina ryhmänä. Esimerkiksi tietokannasta ei kysytä yksittäisiä laskurivejä, vaan aina ensiksi lasku, ja sitten siihen kuuluvat rivit. Samoin ohjelmassa ei anneta suoraa viittausta laskuriviin laskun ulkpuolelle, vaan laskurivin kanssa toimiminen tapahtuu laskun tarjoamien toimintojen kautta.

Kun monimutkaisempia sovellusalueen käsitteitä koostetaan, on toisinaan hyödyllistä siirtää koostamistyö kokonaan erilliseksi oliokseen. Näitä olioita Evans kutsuu tehdas-luokiksi (factory), samaan tapaan kuin olio-ohjelmoinnin perinteessä muutenkin. Tehdas-luokan tehtävänä on koostaa useasta alikäsitteestä koostuva kokonaisuus.

Esimerkiksi laskulla voi olla maksaja, joukko laskutettavia asioita, ja niitä vastaavat laskurivit ja juokseva numerointi. Tällaisen kokonaisuuden rakentaminen ei enää onnistu yksinkertaisella laskun luontikomennolla, vaan tarvitaan erillinen tehdas-luokka, jolta voi pyytää eri tavoin rakennettuja laskuja. Tämä luokka voi varmistaa, että jokainen lasku saa uniikin numeronsa, että kaikki laskuun kuuluvat laskurivit ovat paikallaan ja että maksajan perustiedot ovat myös laskulla mukana ja käytettävissä.

Mielenkiintoinen käsite Evansilla on repositorio. Se edustaa projektissa tiedon tallennuspaikkaa, kuten vaikka tietokantaa. Ajatus on, että sovellusaluemallin ta-

solla ei esitetä konkreettista tiedontallennustoteutusta, vaan ainoastaan abstrakti tietovarasto, *repositorio*. Tähän repositorioon on mahdollista kohdistaa erilaisia hakuja sekä muokkausoperaatioita, ja repositorio palauttaa käytettäväksi kokoelman olioita, jotka edustavat sovellusaluemallin käsitteitä.

Esimerkiksi laskut on mahdollista hakea laskurepositoriosta, johon on toteutettu tarvittavat hakumetodit esimerkiksi laskunumeron tai laskun päiväyksen perusteella. Näin sovellusaluemallia kehitettäessä ei tarvitse pohtia, mihin tieto on tallennettu ja millaisella kyselyllä se haetaan. Esitän listauksessa 1 hyvin yksinkertaisen repositorioluokan laskujen hakemiseksi.

```
1   Class InvoiceRepository:
2
3     def get_all():
4       return self.invoices
5
6     def find_by_date(date)
7       return self.invoices.filter(date)
```

Koodiesimerkki 1: Yksinkertainen esimerkki abstraktista repositoriosta.

Esitetty repositorio tallentaa tiedon vain olion sisään, mutta metodit voisivat yhtä hyvin sisältää esimerkiksi SQL-kutsuja tai jonkin ORM-järjestelmän käyttökutsuja.

3.1.2 Mallin hiominen refaktoroimalla

Refaktoroinnilla tarkoitetaan koodin rakenteen muuttamista niin, että koodin tuottama tulos ei muutu. Refaktoroimalla voi siistiä koodia ja tehdä siitä luettavampaa.

Evansin mukaan hyvä sovellusaluemalli syntyy hyvin harvoin ilman refaktorointia. Hän ei tarkoita refaktoroinnilla pelkästään koodin teknistä siistimistä, vaan refaktoroinnin motivaationa on tuoda sovellusaluemalli paremmin näkyviin koodin tasolla. Tämä vaatii joustavasti kirjoitettua koodia, jota on helppo muunnella.[3, osa 3.]

Keskeinen keino joustavan koodin tuottamiseksi ovat edeltäkäsin kirjoitettavat yksikkötestit. Kun testi laaditaan ennen koodin kirjoittamista, koodin keskinäiset kytkökset muodostuvat löyhiksi. Testien suoritustiheys on myös oleellista. Parhaisiin tuloksiin päästään jatkuvalla testaamisella. Se on menetelmä, jossa testityökalu tarkkailee lähdekooditiedostojen muutoksia, ja suorittaa testipatteriston uudelleen joka kerta, kun koodiin tulee muutoksia. [4, luku 6.]

Kattavasti yksikkötestattua koodia on myös helppo muunnella. Sen rakenteeseen ja toimintaan voi tehdä suuriakin muutoksia ilman, että tarvitsee pelätä takaiskuja. Tämä edellyttää, että testit ovat siistejä ja vähäeleisiä ja kirjoitettu samalla tarkkuudella kuin tuotantokoodi. Paras metodi tähän tulokseen pääsemiseksi on kirjoittaa testejä ja tuotantokoodia rinnakkain rivi riviltä. Ohjenuorana voi käyttää ns. testipohjaisen kehittämisen kolmea sääntöä. Ne ovat: 1. Tuotantokoodia ei saa kirjoittaa, ennen kuin on kirjoittanut epäonnistuvan yksikkötestin 2. Yksikkötestiä saa kirjoittaa vain sen verran kuin vaaditaan sen epäonnistumiseen. Kääntäjän tai tulkin raportoima virhe lasketaan epäonnistumiseksi. 3. Tuotantokoodia saa kirjoittaa vain sen verran, että yksikkötesti menee läpi. [5, luku 9.]

Joustavan järjestelmän suunnittelussa on myös hyvä seurata nk. YAGNI-periaatetta.(“You Aren’t Gonna Need It.” suom. “Et sinä sitä tarvitse.”) Kyseinen periaate edellyttää, että vain sellainen koodi kirjoitetaan, jolle on tarvetta. Luokkiin tai moduuleihin ei lisätä metodeita eikä toimintoja varmuuden varalta. [6.]

Evans toteaa, että joustava rakenne on monesti seurausta siitä, että sovellusaluemalli kehittyy paremmaksi. Joustava ohjelmisto ei ole monimutkainen, eikä se piilota toiminnallisuuksiaan outojen rajapintojen taakse. Joustava ohjelmisto koostuu mahdollisimman pienestä määrästä löyhästi toisiinsa kytkeytyviä käsitteitä. [3, luku 10.]

3.2 GraphQL-teknologian kuvaus

Useimmat sovellukset ovat nykyään web-sovelluksia, ja useimmat web-sovellukset puolestaan perustuvat REST (Representational State Transfer) -rajapintoihin. REST on arkkitehtuurityyli, jossa palvelin esittää asiakasohjelmalle joukon resursseja, joita asiakasohjelma voi pyytää ja muunnella tilattomia pyyntöjä käyttäen. [7.]

REST-tyyli on vakiintunut web-sovellusten toteuttamisteknologiaksi, mutta siihen liittyy myös eräitä ongelmia. Mikäli REST-tyyppisestä rajapinnasta halutaan hakea usean eri resurssin verkko, joudutaan tekemään erillinen pyyntö jokaista resurssia kohti. Toisissa tapauksissa taas halutaan vain osa resurssin esittämästä tiedosta, mutta joudutaan silti hakemaan koko resurssi. [8; 9]

GraphQL on Facebookin kehittämä kyselykieli, jolla edellämainittuja ongelmia pyritään ratkaisemaan. Sen alkuperäinen suunnitteluperiaate oli tarjota web-asiakasohjelmien kehittäjille aiempaa laajempi vapaus rajapintapyyntöjen laatimiseen. GraphQL esittää datan olioiden keskinäisenä verkkona. [10.]

GraphQL on vielä suhteellisen uusi teknologia, eikä siitä löydy kovin paljoa materiaalia. Keskeinen tiedonlähde ovat Facebookin [10] ja GraphQL-säätiön [11] julkaisemat verkkomateriaalit sekä Apollo Graph -yrityksen [12] GraphQL-teknologiaa käsittelevä materiaali.

GraphQL koostuu kahdesta osasta: kyselykielestä sekä tyyppijärjestelmästä. Kyselykielellä muotoillaan pyyntö, johon GraphQL-palvelun tulee vastata. Tyyppijärjestelmä taas tarkistaa pyynnön oikeellisuuden.

GraphQL ei ole varsinainen rajapinta, sillä palvelun toteuttamisteknologia on määrittelyn ulkopuolella. Useimmiten GraphQL-palvelut on toteutettu HTTP-teknologian päälle, mutta muitakin, kuten WebSocketia, voi käytttää. GraphQL ei myöskään määrittele, miten kyselyn vastaus tulee muodostaa tai millä ohjelmointikielellä järjestelmä tulee toteuttaa.

Aivan täysin kielineutraali GraphQL ei silti ole. Osa konventioista on JavaScript-konventioita. Esimerkiksi kenttien ja muuttujien nimet on tapana kirjoittaa camelCase- ja PascalCase -muodoissa, eli vaihdellen suuria ja pieniä kirjaimia sanan sisällä samoin kuin kyseiset sanat tässä virkkeessä on tehty. [13.]

3.2.1 Verkoista

GraphQL esittää datan olioiden välisenä verkkona. Verkko eli graafi on tietorakenne, joka koostuu N:stä solmusta ja niitä yhdistävistä kaarista. [14] Verkkojen avulla on mahdollista esittää monenlaisia asioiden välisiä suhteita, kuten esimerkiksi reittikartta usean kaupungin välisistä teistä.

Olio-ohjelmoinnin tyyli esittää ratkaistavan ongelmakentän olioiden välisinä verkkoina. Ohjelmassa ei ole juuri lainkaan jaettua tilan käsittelyä, vaan kaikki tai lähes kaikki ohjelman sisältämä tieto on olioissa. Näin käsiteltävät ongelmakokonaisuudet saadaan jaettua pienempiin, hallittavan kokoisiin osiin. [15.]

GraphQL:n avulla sovellusala on mahdollista esittää verkon muodossa määrittelemällä GraphQL-tyyppien verkko. Tämän avulla rajapinta tarjoaa asiakasohjelmalle rakenteen, joka muistuttaa olio-ohjelmoinnin olioverkkoja. [16.]

Juuri verkkorakenne tekee olio-ohjelmoinnista tehokkaan välineen tosielämän ongelmien ratkaisemiseen. Oliot voivat edustaa ohjelman sovellusaluemallin käsitteitä, ja verkko muodostaa niiden välisen käsitekartan.

Käsitteiden verkkoa kuvaa myös Eric Evans Domain Driven Design -kirjassa. Keino rakentaa kaikenkattava kieli kehittäjien ja alan asiantuntijoiden välille on antaa tärkeille käsitteille nimet, sijoittaa ne mallissa oikeisiin suhteisiin keskenään ja myös toteuttaa ne koodissa.

Jotta tietokone saa käsitteistä täyden hyödyn irti, on käsitteet jaettava muodollisesti määritettyihin tyyppeihin.

3.2.2 Tyyppijärjestelmä

Tietokoneet käsittelevät dataa ottamatta sen enempää kantaa sen tyyppiin. Pohjimmiltaan data on vain bittijonoja muistissa, tai elementtejä joukossa. Kun tällaista järjestämätöntä ja tyypittämätöntä joukkoa ryhdytään käsittelemään, on välttämätöntä järjestää se erilaisiin kategorioihin. Tämä tiedon luokittelu synnyttää tyyppijärjestelmän, mutta järjestelmä ei ole formaalisti määritelty, eikä tietokone voi niin ollen tehdä tyyppitarkistusta.

Tyypitys tarkoittaa määrättyjä rajoituksia, joiden avulla voidaan varmistaa muuttujan oikeellisuus. Staattinen tyypitys on menetelmä, jossa lausekkeiden tyyppi voidaan määrittää staattisen analyysin avulla, siis jo käännösaikaisesti. Vahva tyypitys taas mahdollistaa tyypin tarkistamisen luotettavasti ajon aikana. [17.]

GraphQL-rajapinta koostuu tyypeistä, joita rajapinnalle lähetettävä kysely käyttää. Kyselyssä määritetään pyydettävät tyypit ja niiden kentät. Rajapinta palauttaa takaisin oliota edustavan joukon kenttiä assosiaatiotaulu{assosiaatiotaulun} muodossa, eli tietorakenteena, joka koostuu avain-arvo-pareista. [10.]

GraphQL:ää käyttävät sovellukset on kuitenkin useimmiten kirjoitettu dynaamisesti tyypitetyillä kielillä, jotka eivät sisällä tyyppitarkistuksia, tai tarkistukset ovat ajonaikaisia ja vapaaehtoisia. Esimerkiksi alkuperäinen GraphQL-referenssi-implementaatio on kirjoitettu JavaScriptillä [18.].

Samoin Python-kieli on dynaamisesti tyypitetty, ja olioiden tunnistamisessa se käyttää duck-tyypitykseksi kutsuttua menetelmää. Duck-tyypityksessä olion tyyppiä ei tarkasteta välttämättä edes ajonaikaisesti, vaan oletetaan sen sisältämien jäsenten perusteella [19]. Jos esimerkiksi oliosta löytyvät kentät *summa* ja *lasku-numero*, oletetaan, että olio on lasku.

Tässä mielessä voidaan ajatella, että GraphQL on keino tuoda ajonaikaisia tyyppitarkistuksia myös dynaamisesti tyypitetyillä kielillä kirjoitettuun sovellukseen. Rajapinta erottaa toisistaan ohjelmiston taustaosan ja käyttöliittymän, joten tällä rajalla tehtävän tyyppitarkistuksen voi ajatella ehkäisevän virheitä ja parantavan

ohjelmiston luotettavuutta.

Oheisessa koodiesimerkissä 2 kuvaan rajapinnan edustamien tyyppien, ja sitä myötä sen palauttamien olioiden väliset suhteet. ConsolidatedInvoice-tyyppisessä oliossa on sisällä invoices-kenttä, joka on lista Invoice-tyyppisiä olioita.

Laskutuksessa ConsolidatedInvoicella eli koontilaskulla tarkoitetaan yhdistelmälaskua, joka kokoaa yksittäisiä laskuja (Invoice).

Tältä rajapinnalta voi pyytää listaa koontilaskuista. GraphQL:n tyyppijärjestelmä takaa, että koontilaskun sisällä on invoices-jäsen, joka sisältää listan Invoice-tyyppisiä olioita, eli siis laskuja.

```
type Query {
  consolidatedInvoices [ConsolidatedInvoice]
}

Type Invoice {
  number: Int
  sum: Float
  date: Date
}

type ConsolidatedInvoice {
  number: Int
  invoices: [Invoice]
}
```

Koodiesimerkki 2: Esimerkki GraphQL-skeemasta, joka määrittelee yksinkertaisen rajapinnan.

3.2.3 Skeema

GraphQL:n olioverkko kuvataan täsmäkielellä määritellyn skeeman avulla.

Täsmäkieli on ohjelmointikieltä korkeamman tason kieli, joka on suunniteltu jollekin kapealle sovellusalueelle [20]. Esimerkkejä täsmäkielistä ovat esimerkiksi UNIX-tyyppisistä järjestelmistä tutut *sed*- ja *awk*-kielet. Tällaisen kielen avulla on mahdollista määritellä monimutkaisiakin asioita nopeasti. [21] Kieli tarjoaa tavanomaista ohjelmointikieltä ilmaisuvoimaisemman ja luonnollista kieltä täsmällisem-

män tavan määritellä asioita.

Eräs täsmäkielen kanssa hyödynnettävä suunnittelumalli on kielen käyttäminen tietorakenteiden abstraktioon[22]. GraphQL-rajapinnassa on kysymys juuri tästä.

GraphQL-rajapinnan tyypit, niille tehtävät kyselyt ja mutaatiot kuvataan skeemassa GraphQL-kielen avulla. Edellä esitetty ConsolidatedInvoice- ja Invoice-olioista koostuva esimerkki on validi GraphQL-skeema. Tämä skeemamäärittelyihin käytettävä kieli on riippumaton ohjelmointikielestä. GraphQL-kirjastot eri kielissä lukevat skeeman, tarkistavat sen, ja sen jälkeen suorittavat Skeeman avulla ajonaikaisen tyyppitarkistuksen.

GraphQL-kehityksen tyylejä on useita, ja yksi suosittu tapa on kirjoittaa skeema ensiksi. Se tarjoaa suuntaviivat sekä rajapinnan tekniselle toteutukselle, että myös graafisen asiakasohjelman laatimiselle.[23; 24]

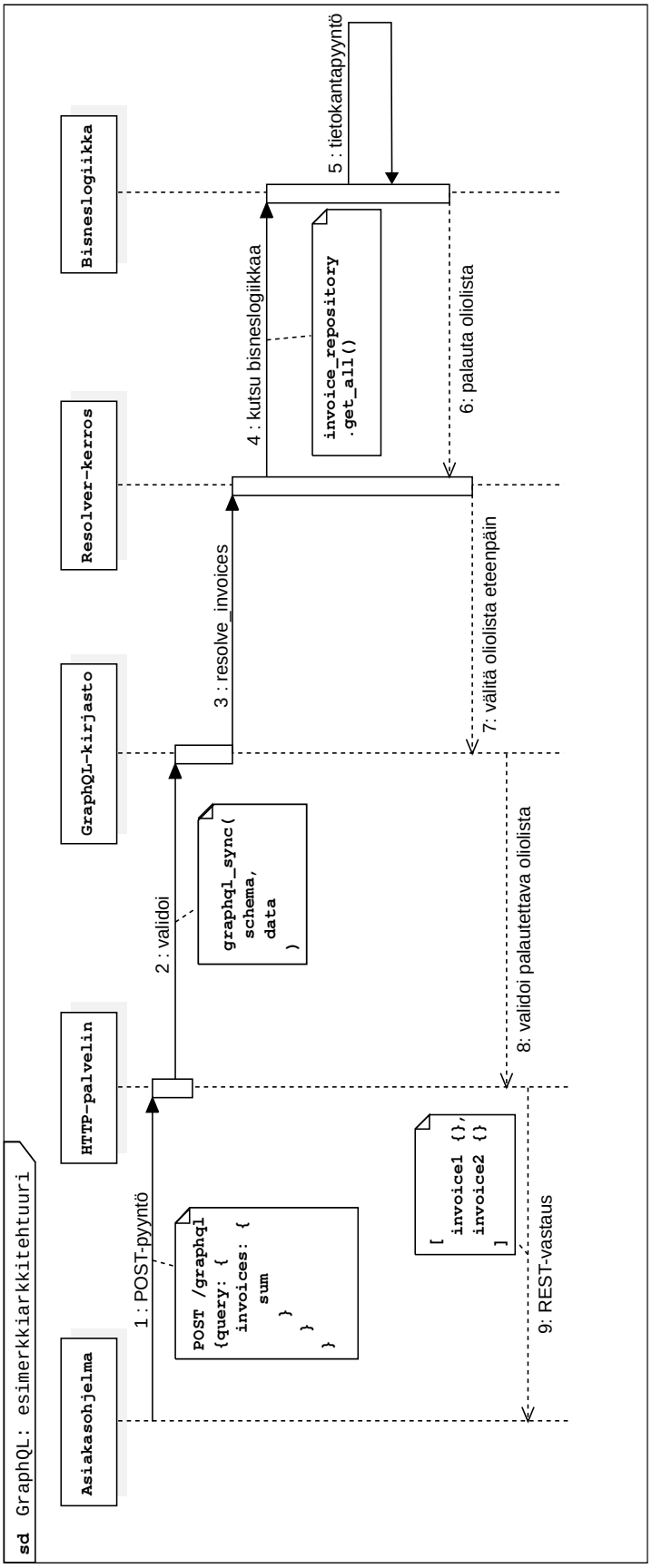
GraphQL-skeemaa voi siis verrata Eric Evansin esittämään ajatukseen kaikenkattavasta kielestä. Esimerkiksi GraphQL Foundationin materiaaleissa esitetään, että GraphQL-skeemaa tulisi ajatella jaettuna kielenä oman ohjelmointitiimin kesken, ja myös käyttäjien kanssa kommunikoimiseen .[16.]

3.2.4 Miten GraphQL-sovellus toimii

Kuvassa 1 esitän yksinkertaisen GraphQL-sovelluksen arkkitehtuurin. Arkkitehtuuri on kerroksittainen, ja rajapinnalle esitettävä pyyntö liikkuu siinä ylhäältä alaspäin. Koska GraphQL on teknologianeutraali, tämä on vain yksi, toki suhteellisen tyypillinen esimerkki mahdollisesta GraphQL-sovelluksesta.

Ensiksi pyyntö saapuu HTTP-palvelimeen. Tämä palvelin on käytännössä ohjelmointikielestä riippuen kirjasto, joka vastaanottaa HTTP-pyyntöjä ja osaa käsitellä niitä. Tyypillisesti GraphQL-rajapinnassa on vain yksi `graphql`-niminen resurssi, jolle pyynnöt esitetään. Pyynnöt ovat aina POST-tyyppisiä.

HTTP-kerros ottaa pyynnön vastaan ja lukee sen body-osassa olevan merkkijo-



Kuva 1: Esimerkkiarkkitehtuuri GraphQL-sovellukselle

nomuotoisen GraphQL-kyselyn. Tämä kysely on tehty GraphQL-kyselykielellä. Rajapinta antaa kyselyn GraphQL-kirjastolle.

Kirjasto ottaa kyselyn vastaan ja tarkistaa sen oikeellisuuden GraphQL-skeemaa vasten. Mikäli kysely on muodoltaan oikeanlainen, kirjasto ohjaa sen eteenpäin resolvereiksi kutsutuille funktioille. Resolver-funktiot eivät ole osa kirjastoa, vaan sovelluksen kehittäjä kirjoittaa ne. Käytännössä resolver-funktio on nk. puhdas funktio, jonka tehtävänä on tuottaa vastaus yksittäiseen GraphQL-kyselyn kenttään.

Laskutusta käsittelevässä esimerkissä rajapinnan invoices-kenttää voisi vastata `invoices_resolver`-niminen funktio.

Resolver-funktioiden kerros on alin kerros, josta GraphQL-palvelu tietää. Sen alla oleva ohjelmistologiikka on riippumaton rajapinnan olemassaolosta. Tyypillisesti siellä voi sijaita sovelluksen liiketoimintalogiikka ja infrastruktuuri, kuten esimerkiksi tiedon tallennusteknologia.

3.2.5 Query ja Mutation -juurityypit

GraphQL-palveluissa on muutama oletuksena määritelty tyyppi, joita kutsutaan juurityypeiksi. Tässä käsittelen niistä kahta keskeisintä, Query- ja Mutation-tyyppiä.

Rajapintaan voi tehdä kyselyjä Query-tyyppisen juuriolion kautta. Tämän olion kentät määrittävät, mitä dataa rajapinnalta voidaan kysellä. Kentät ovat ikäänkuin sisäänmenoaukkoja, joiden kautta oliorakenteita voi pyytää.

Kun aiemman koodiesimerkin 2 mukaisesti määritellystä GraphQL-rajapinnasta halutaan hakea tietoja, tehdään Query-tyypin consolidatedInvoices-kenttään kysely, joka kuvaa halutun oliopuun rakenteen tyyppien avulla. Tätä kuvaa koodiesimerkki 3.

```
1 {
2   consolidatedInvoices {
3     number
4     invoices {
```

```
5         number
6         sum
7       }
8     }
9   }
```

Koodiesimerkki 3: Kysely, joka pyytää consolidatedInvoices-olioita

Kyselyssä määritellään kentät, jotka palautuvassa datassa halutaan nähdä. Näin myös palautuvan oliopuun syvyyttä voidaan kontrolloida. Oheisessa esimerkissä haetaan paitsi lista koontilaskuista, myös jokaisen koontilaskun alle invoices-kenttään lista siihen kuuluvista laskuista. Niistä pyydetään jokaisesta laskunumero ja laskun summa. Näin edetään verkkoa pitkin tarvittavan datan luo.

Mutation-juurityyppiä puolestaan käytetään datan muunnoksiin. Mutation-tyypin sisältämiin kenttiin lähetetään kysely, jossa mukana olevat parametrit kertovat, miten dataa muokataan. Parametrit ovat yhtä lailla tyypitettyjä kuin rajapinnan kentät, ja GraphLQ-kirjasto tarkistaa niiden tyypin oikeellisuuden. Mutation-komennot voivat myös palauttaa oliorakenteita.

Oheisessa koodiesimerkissä 4 määritellään invoice-niminen operaatio uuden laskun luomiseen. Operaatiolle on merkitty paluuarvo, joka on Invoice-tyyppinen olio. Käytännössä operaatio siis palauttaa juuri luodun laskun.

```
1 Mutation {
2   invoice(customerId: Int, appointmentDate: String): Invoice
3 }
```

Koodiesimerkki 4: Esimerkki GraphQL-mutaatiosta

Kun tällaista mutaatiota käytetään, kutsuun liitetään myös kyselyosa, jossa listataan ne kentät, jotka palautuvasta oliosta halutaan. Koodiesimerkissä 5 pyydetään laskunumero ja laskun summa.

```
1 {
2 invoice(customerId: 1, appoitnmentDate: "2021-09-21") {
3     number
4     sum
5   }
6 }
```

Koodiesimerkki 5: Esimerkki parametreja käyttävästä GraphQL-kyselystä

3.3 GraphQL ja sovellusaluevetoinen suunnittelu

Eric Evansin kirjassa sovellusaluemalli rakennetaan olio-ohjelmoinnin tekniikoita käyttäen. Vaikka se ei olekaan ainoa tapa rakentaa sovellusaluemalli, on se kuitenkin tyylinä hyvin suosittu.

Kuten edellä olen esittänyt, GraphQL puolestaan on olioverkkoon perustuva rajapinta, jossa rajapinnan tarjoama tieto on jäsennetty olioiksi ja niiden välisiksi suhteiksi.

Voidaan siis esittää kysymys, kuinka hyvin GraphQL-rajapinta soveltuu sovellusaluevetoisen suunnittelun tarpeisiin. On helppo kuvitella, että olioverkolla on mahdollista heijastaa sovellusaluemalli, ja jopa mallintaa se lähes yksi yhteen. Toisaalta GraphQL-rajapinta palauttaa dataa, kun taas olio-ohjelmoinnin keinon luotu olioverkko voi esittää dynaamisen ja muuttuvan mallin. Onko GraphQL toimiva väline ohjelmiston tietomallin parantamiseen? Tätä kysymystä lähden työni käytännön osassa selvittämään.

4 Työn kulku

Yrityksessä haluttiin valita laskutuksen sisältä tapaus, jossa hoitokäynti tulee voida jakaa usealle eri maksajalle osoitetuille laskuille, ja nämä laskut tulee voida hyvittää itsenäisesti. Mallintamisen aiheeksi valittu laskutus oli aihealueena minulle tuntematon, ja yksi prosessin haasteita olikin, pystynkö muutamassa viikossa omaksumaan riittävästi laskutuksen käsitteitä toimivan mallin aikaansaamiseksi.

4.1 Laskutuksen taustaa

Ohjelmistoa käyttävä terapeutti kirjaa järjestelmään hoitokäyntejä ja laskuttaa niitä. Monesti samalle maksajalle – etenkin, jos tämä ei ole yksityishenkilö vaan

instituutio – kertyy monta eri laskua, jotka lähetetään yhtenä joukkona esimerkiksi kerran kuukaudessa. Tätä kutsutaan koontilaskuksi.

Toisinaan jo luodussa laskussa huomataan virhe. Kirjanpidon periaatteiden mukaan laskuja ei kuitenkaan saa tuhota, vaan virheellinen lasku on oikaistava tai kumottava. Tässä ohjelmistoprototyypissä oletetaan, että kyse on aina virheellisesti laskutetusta käynnistä, jota maksaja kieltäytyy maksamasta ja joka pitää kumota. Tämä tapahtuu luomalla hyvityslasku.

Hyvityslasku on voitava luoda siten, että yksittäisellä laskulla oleva yksittäinen rivi voidaan kumota, muun laskun (ja sen sisältävän koontilaskun) säilyessä avoimena.

4.2 Työskentelytavat

Päätin tehdä työn lyhyissä iteraatioissa ketterän kehityksen periaatteita seuraten. Tämä tyyli soveltuu hyvin yhteen tietomallin kehittämisen kanssa, sillä Evansin kirjassa kuvattu työtapa on samankaltainen. Lisäksi lyhyet iteraatiot ovat nykyään tyypillinen tapa tehdä ohjelmistoa.[25; 26] En asettanut iteraatioille mitään ennalta määrättyä kestoa.

Sovellusaluevetoisessa suunnittelussa oleellista on ohjelmoijan ja sovellusalueen asiantuntijan välinen kommunikaatio. Asetin siis tiimimme tuoteomistaja Lauran sovellusalueen asiantuntijan rooliin, ja käytin häntä kuvitteellisen asiakkaan edustajana. Tämä rooli sopi Lauralle erinomaisesti johtuen hänen työkokemuksestaan fysioterapeuttina ja yritäjänä.

Malli laadittiin englanninkielisillä käsitteillä, koska Nordhealth on viime vuosina laajentanut toimintaansa ulkomaille, ja yrityksen viralliseksi kieleksi on vaihdettu englanti.

Päätin, että jokainen iteraatio aloitetaan minun ja tuoteomistaja Lauran välisellä suunnittelukokouksella, jonka pääasiallisena tavoitteena oli keskustellen ja piirtäen etsiä toimivaa ohjelmiston tietomallia. Prosessin kuluessa tavaksi vakiintui,

että kokouksen aluksi käytiin nopeasti läpi siihen asti aikaansaadun ohjelmisto-prototyypin toiminnallisuus.

Pyrin noudattamaan työskentelyssä Eric Evansin esittämää tiedon rouhimisen periaatetta, jossa suunnittelu ja ohjelmistokehitys limittyvät keskenään. Iteraation aikana ohjelmoin uuden version ohjelmistoprototyypistä, kokouksessa syntyneiden ajatusten pohjalta. Kehitin ohjelmistoa testivetoisesti: kirjoitin ensin epäonnistuvan yksikkötestin ja sen jälkeen tuotantokoodia sen verran, että yksikkötestin suorittaminen onnistui. Käytin valmistunutta prototyyppiä jatkokeskustelujen pohjana.

4.3 Teknologiavalinnat

Kirjoitin esimerkkiohjelmiston tyypilliseksi web-sovellukseksi, jossa palvelinohjelmisto ja selaimessa toimiva asiakasohjelma kommunikoivat keskenään HTTP-pyyntöjen avulla. Valitsin palvelinohjelmiston kehityskieleksi Python-kielen, koska sitä käytetään Nordhealthilla muutenkin. Python on myös syntaksiltaan suoraviivainen ja tässä mielessä helppokäyttöinen, prototyyppien rakentamiseen soveltuva kieli.

Pythonin kanssa käytettäväksi HTTP-kirjastoksi valitsin Falcon-kirjaston [27] puhtaasti sen yksinkertaisuuden vuoksi. Muita vaihtoehtoja olivat Django ja Flask [28], mutta molemmat niistä sisälsivät paljon toimintoja, joita ei tässä projektissa tarvittu. Niissä on mukana esimerkiksi tuki sivupohjille, jota rajapintaa kehitettäessä ei tarvita. Ohjelmaan tarvittiin tuki vain yhdelle HTTP-resurssille, joka vastaa pyyntöihin JSON-muotoisella dokumentilla. GraphQL-kirjastoista harkitsin Ariadne [29]- ja Graphene [30] -kirjastojen välillä. Valitsin Ariadnen, koska se on tarkoitettu skeema edellä tapahtuvaan kehitystyöhön.

Yksikkötestijärjestelmänä käytin Pytest-kirjastoa. Tietokantaa sovellukselle ei tarvittu, vaan rakenteet voidaan tallentaa muistiin ajonaikaisesti. Tämä helpottaa myös ohjelmiston tietorakenteen refaktorointeja, sillä tietokantaa ei ole tarve muokata tai luoda uudelleen ohjelmiston mallin muuttuessa.

Asiakassovelluksen kirjoitin Vue.js-JavaScript-kirjastoa käyttäen, koska se on Nordhealthilla käytössä jo ennestään. GraphQL-rajapinnan kanssa kommunikoimiseen käytin Apollo-kirjastoa, ja sen Vueen integroivaa Vue Apollo -kirjastoa.

4.4 Kuvaus prosessin etenemisestä iteraatio iteraatiolta

Esitän seuraavaksi prosessin etenemisen iteraatio kerrallaan. Olen valinnut tähän osuuteen päiväkirjamaisen ja epämuodollisen tyylin, jonka kautta pyrin kuvaamaan prosessin luonnetta. Ohjelmiston kehittäminen koostui vapaamuotoisista keskusteluista, joihin kuului paljon kokeiluja ja hapuiluja eri suuntiin. Tämä tyyli oli oma pyrkimykseni synnyttää tiedon rouhimisen prosessi.

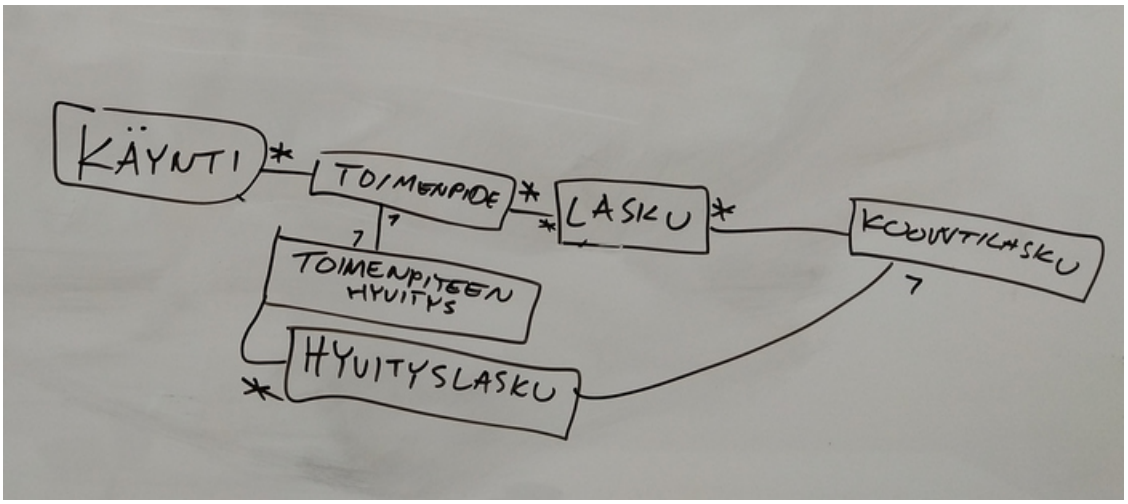
Malleja esittävissä kuvissa käyttämäni notaatio on hyvin alkeellinen, eikä millään tapaa muodollinen. Se muistuttaa etäisesti UML-kielen luokkakaavioita, ja sen keskeisin tarkoitus on luoda jaettu ymmärrys eri käsitteiden välisistä suhteista.

4.4.1 Iteraatio 1: projekti käynnistyy, kirjanpidon alkeita

Iteraation aloittavassa kokouksessa kävimme läpi laskutuksen perusperiaatteita. Käsittelimme laajasti koko ongelmakenttää, ja pohdimme mahdollisia ratkaisuja laskutuksen liepeillä oleviin kysymyksiin. Näistä monet jäivät toteutetun prototyypin ulkopuolelle, mutta ne selkiyttivät käsitystä ongelmakentän luonteesta.

Laura selitti kärsivällisesti kirjanpidon alkeita, jotka olivat minulle enimmäkseen uutta tietoa. Keskeinen oivallus oli, että kirjanpidossa kirjanpitoaineistoa ei saa luomisen jälkeen enää muuttaa. Jos laskua halutaan myöhemmin korjata, on luotava erillinen kirjanpitotosite, jolla oikaisu tehdään. Tätä kutsutaan tässä yhteydessä hyvityslaskuksi.

Suunnittelimme yksinkertaisen mallin, jossa hoitokäynnit liittyvät laskuihin ja laskut kootaan koontilaskuille. Yksittäiset käynnit voidaan lisätä myös hyvityslaskulle. Tämän mallin tarkoituksena oli luoda yksinkertainen esimerkkisovellus, joka kykenee laskuttamaan käyntejä, ja sen jälkeen lisäämään niitä hyvityslaskulle,



Kuva 2: Ensimmäinen malli

sekä näyttämään hyvityslaskun kokonaissumman. Malli on esitetty kuvassa 2.

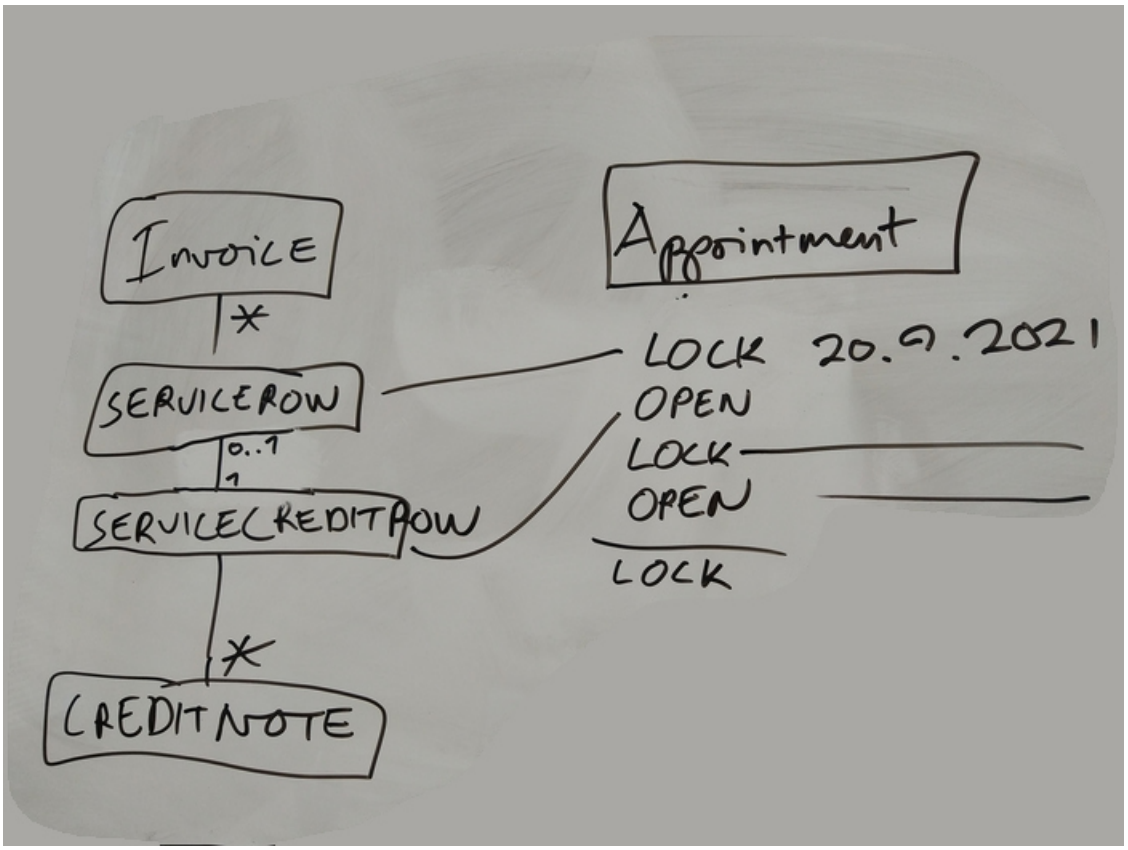
Tämän mallin sisältävän ohjelmistoprototyypin toteuttamiseen kului kaksi viikkoa, ja näin iteraatio oli prosessin pisin. Myöhemmät iteraatiot kestivät noin viikon. Kulunutta aikaa selittää, että rakensin prototyypin puhtaalta pöydältä, jolloin aikaa kului myös sovelluksen pohjan pystyttämiseen.

Ensimmäisessä iteraatiossa syntynyt ohjelmisto ei malliltaan täysin vastannut tätä ensimmäisen kokouksen piirrosta. Yksinkertaisuuden vuoksi oletin, että käynnillä on aina vakiohintainen toimenpide, jolloin Toimenpide-olio jäi mallista pois. Tehdessä huomasin myös, että koontilaskun ja hyvityslaskun välille ei tarvita mitään suoranaista yhteyttä. Riittää, että käynti on yhdistetty näihin molempiin.

4.4.2 Iteraatio 2: malli syvenee

Iteraation aloittavassa kokouksessa kävimme läpi syntyneen ohjelmistoprototyypin. Läpikäynnin jälkeen olin hieman epävarma, miten tapaamista tulisi jatkaa. Päätin ottaa puheeksi minua koko viikon ajan vaivanneen käyntien ja laskujen läheisen kytköksen. Ohjelmoidessa tuntui väärältä ja kömpelöltä, että käynti lisätään suoraan laskulle ja vielä hyvityslaskullekin.

Pyysin Lauraa kertomaan enemmän siitä, mitä käynnin laskuttaminen oikeastaan tarkoittaa, ja hän kuvasi, millaisissa tilanteissa laskuja voidaan luoda. Huomioni



Kuva 3: Toinen malli

kiinnittyi puheessa esiintyneeseen termiin **Laskutusperuste**. Tämä tuntui valtavan kiinnostavalta, ja Laura avasi asiaa tarkemmin. Kun laskulle lisätään laskutettavia asioita, täytyy siihen olla jokin peruste.

Käytimme tapaamisen loppuosan tämän idean kehittelemiseen. Päädyimme ajatukseen, jossa laskulle lisätään käynnin sijasta palvelurivi, joka viittaa käyntiin. Tapaamisen jälkeisen viikon kehitystyötä ohjasi nyt uusi ajattelutapa: käyntiä sinänsä ei liitetä laskuun, vaan käynti laskutetaan, mikäli laskutusperuste täyttyy.

Yksi toisen tapaamisen aikana syntyneistä malleista on esitetty kuvassa 3. Siinä laskulle liitetään palvelurivi, joka vastaa yksittäistä hoitokäyntiä. Mikäli käynti hyvitetään, palveluriviä vastaa hyvityslaskuun kiinnitetty hyvitysrivi.

Kuvan laidassa näkyy myös hahmotelma siitä, miten käynnin voisi hyvittämisen jälkeen laskuttaa uudelleen. Sarja monimutkaisia lukitusoperaatioita tuntui Laurasta jo piirtämisen yhteydessä hankalalta ymmärtää. On myös kuvaavaa, että

pidimme tapaamisen aikana tätä syntynyttä mallia puutteellisena, ja kehitimme siitä mielestämme paremman version.

Mallin toteuttamisessa kesti vajaat neljä päivää. Jotta uusi ajatus palvelurivistä oli mahdollista lisätä koodiin, täytyi olemassaolevaa ohjelmaa aluksi refaktoroida voimakkaasti. Aiemmin suoraan laskulle kytkettyyn käyntiin piti sijoittaa sisälle palveluriviä kuvaava olio, ja lasku muuttaa koostumaan näistä. Käynnin ja hyvityslaskun välinen kytkös piti katkaista, ja tilalle rakentaa kytkös laskurivin ja hyvitysrivin välille.

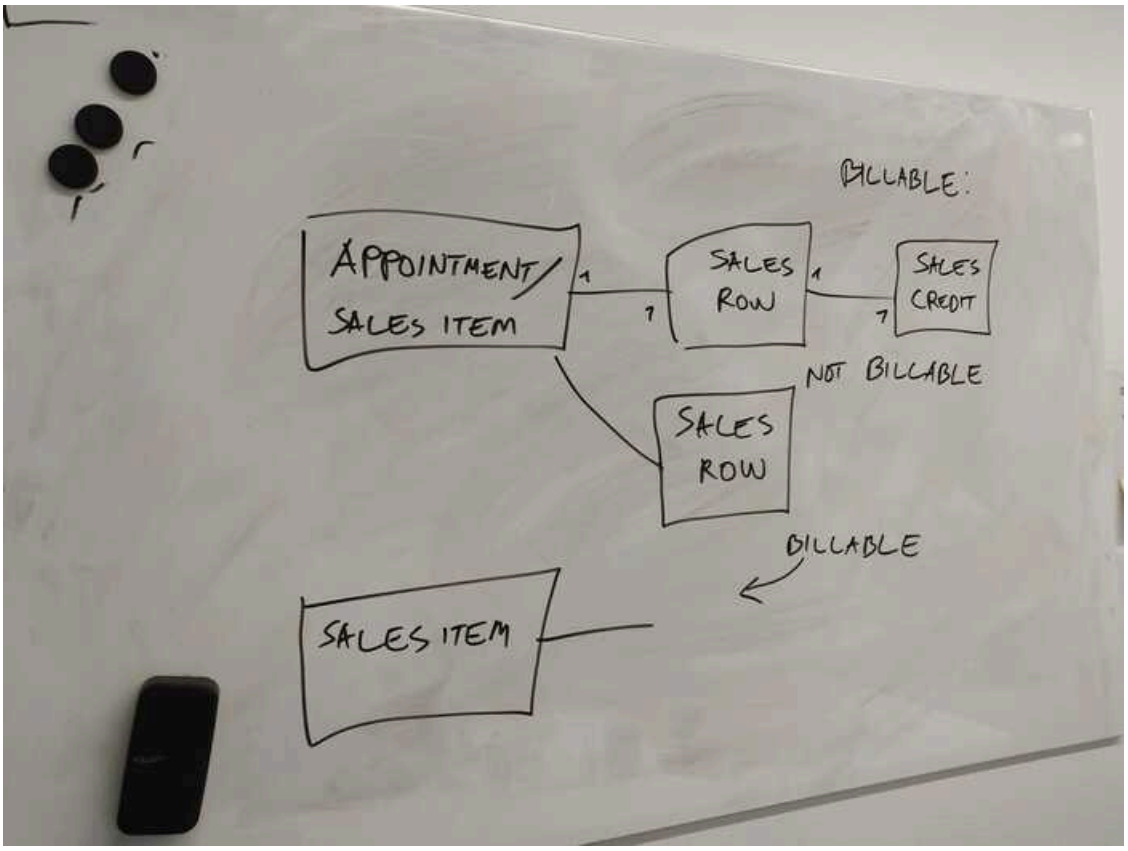
Koodin refaktoroimisessa kului aluksi päivä, ja sen jälkeen uuden, palvelurivejä ja hyvitysrivejä käsittelevän koodin luomiseen kaksi päivää.

Perjantaihin tultaessa olin refaktoroinut prototyyppiohjelmaa ja sen jälkeen käyttänyt kolme päivää ensimmäisen käyttäjätarinan parissa. Vaikutti, että aikataulu pettää, eikä mitään tule valmiiksi maanantaille sovittuun seuraavaan tapaamiseen.

Yllättäen perjantaina puolen päivän jälkeen kaikki yksikkötestit menivät läpi, käyttäjätarina valmistui, ja ohjelmistoprototyypin toiminnassa tuntui tapahtuvan laadullinen hyppäys. Vaikutti kuin prototyyppiohjelma olisi oppinut itsekseen jotain laskutuksesta. Ohjelman logiikka toimi paremmin kuin mitä itse ymmärsin laskutuksesta.

Loppujen kahden käyttäjätarinan toteuttaminen onnistui kahdessa tunnissa, ja vaati vain joitain rivejä koodia. Sovellusaluemalli oli syventynyt.

Ohjelmoidessa syntynyt tietomalli sisälsi samat asiat, joista kokouksessa oli puhuttu, mutta niiden suhteet olivat toisenlaiset. Olin tuottanut ominaisuudet yksikkötesti yksikkötestiltä, ja tämä malli oli yksinkertaisin, jolla kaikki testit menivät läpi. Malli on esitetty kuvassa 4



Kuva 4: Kuva, jossa käyntiin kytkeytyy palvelurivi ja palveluriviin hyvitysrivi

4.4.3 Iteraatio 3: malli osoittaa joustavuutensa

Kolmannen iteraation aluksi pidimme jälleen suunnittelukokouksen. Tämän tapaamisen keskeisimpänä ongelmana oli, miten jo kertaalleen laskutettu ja hyvitetty käynti voidaan laskuttaa uudelleen.

Yritimme piirtää monenlaisia erilaisia malleja ja diagrammeja, mutta mikään niistä ei tuntunut osuvalta. Tämä tapaaminen oli tunnelmaltaan kaikista tapaamisista jähmein. Kun kommunikaatio ei sujunut, myös malli kehittyi kehnosti.

Laadimme tapaamisen lopuksi kuitenkin joukon käyttäjätarinoita, jotka tähtäsivät tavoitteeseemme, hyvitetyn käynnin uudelleenlaskuttamiseen.

Aloitin myös tämän viikon ohjelmointityön samantapaisella mallin refaktoroinnilla kuin edellisen. Tällä kertaa se oli suppeampi, koska uusia ajatuksia oli vähemmän.

Ohjelmoidessa yllätyin: sain rakennettua yhdessä päivässä ominaisuuden, jossa kertaalleen laskutettu ja hyvitetty käynti voidaan laskuttaa uudelleen. Käytännössä riitti, että muutin käynnillä olevan viittauksen palveluriviin listamuotoiseksi. Nyt käyntiin oli mahdollista kytkeä yhden sijasta monta eri palveluriviä, jolloin uudelleenlaskutus onnistui. Palvelurivien, ja niihin kytkeytyvien hyvitysrivien suhteista oli helppo rakentaa säännöt sille, onko käynti laskutettavissa vai ei.

Tämä listamuoto on esitetty kuvassa 4. Siinä tarkastellaan palvelurivien listan viimeisintä jäsentä, ja tällä perusteella arvioidaan, onko käynti laskutettavissa vai ei.

Kolmas iteraatio oli iteraatioista lyhin, ja se kesti vain noin neljä päivää.

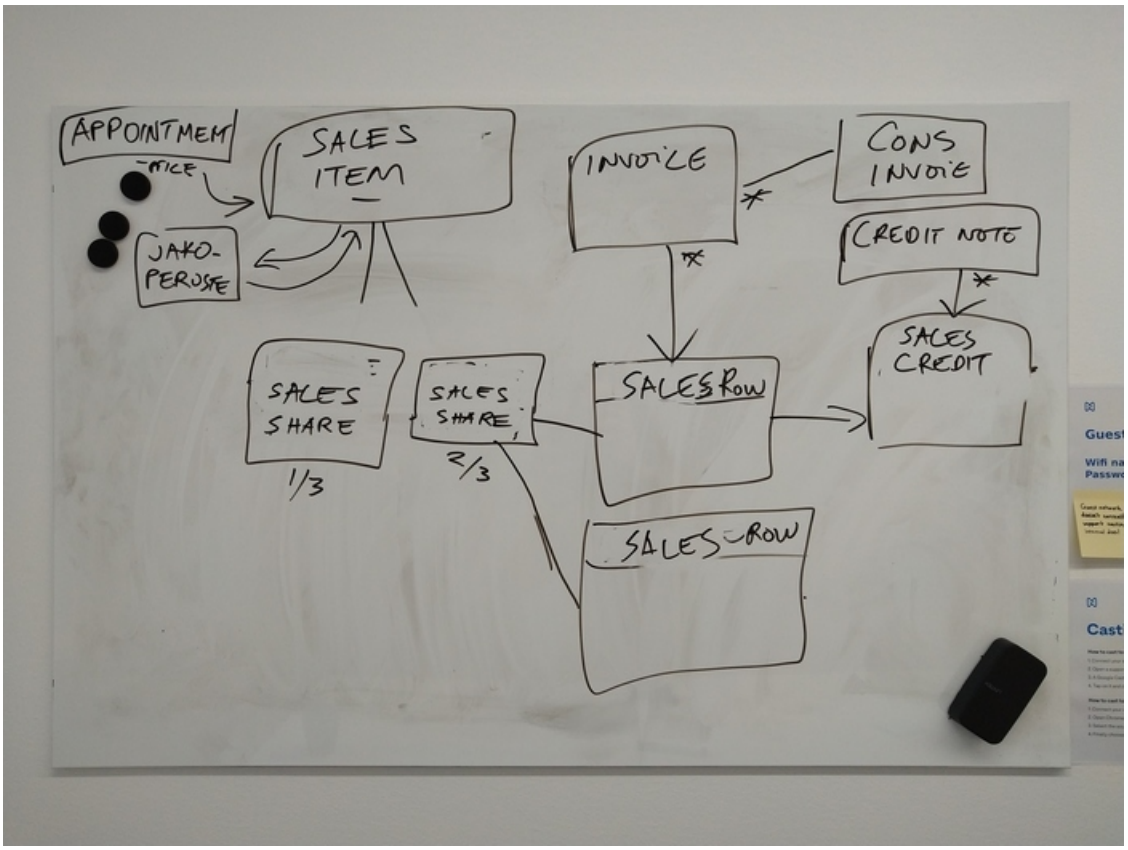
4.4.4 Iteraatio 4: piilossa ollut käsite löytyy

Neljännen tapaamisen keskeinen ongelma oli, että käynti tuntui olevan edelleen liian vahvasti kytketty laskuun. Yksittäistä käyntiä vastasi yksi palvelurivi. Tämä käy ongelmalliseksi, mikäli käynti halutaan jakaa kahdelle maksajalle. Jo ensimmäisessä tapaamisessa oli käynyt selväksi, että ainut siisti tapa jakaa käynti kahdelle maksajalle on tehdä kaksi erillistä laskua. Käyntiin kytkettyä palveluriviä ei kuitenkaan voi lisätä molemmille laskuille.

Vaikutti siltä, että käynnin ja laskun välistä puuttui edelleen jokin käsite. Olin keskustellut aiemmin viikolla laskutuksen kanssa työskennelleen tiimikaverini kanssa, ja hän kiinnitti huomiota siihen, että laskuille laitettiin “palvelurivejä”. Hän oli itse käyttänyt omissa malleissaan “myyntiä”.

Nyt muistin tämän keskustelun ja ehdotin sen pohjalta, että laskutuksessa ei käsiteltäisikään suoraan käyntejä vaan palvelumyyntiä. Laura totesi, että kaikki laskuille laitettava on lopulta myyntiä – ikäänkuin se olisi ollut itsestäänselvyys! Ohjelmoijalle tämä oivallus oli kuitenkin uusi tieto, ja se avasi täysin uuden näkökulman laskutukseen.

Loimme kokouksessa mallin, jossa käynti muunnetaan myynniksi eli SalesItem-



Kuva 5: Kolmas malli

olioksi. Nyt käynneistä ei tarvitse välittää lainkaan laskuja käsiteltäessä. SalesItem puolestaan voidaan jakaa maksajille suunnatuiksi osuuksiksi, SalesShareiksi, ja yksittäisellä laskulla on maksajan osuuteen kytketty myyntirivi, SalesRow. Malli on esitetty kuvassa 5

Kokouksen jälkeen minua odotti jälleen refaktorointityö, joka oli projektin suurin. Käynnin perinpohjainen irrottaminen koko laskutuslogiikasta ja kahden uuden käsitteen laittaminen näiden väliin vaativat laajan remontin koko ohjelman toimintalogiikkaan.

Uusien ominaisuuksien toteuttaminen refaktoroinnin jälkeen oli huomattavan suoraviivaista, ja tuloksena syntyi ohjelma, joka pääsi alkuperäiseen tavoitteeseensa, jaetun käynnin hyvittämiseen ja uudelleen laskuttamiseen. Ohjelma oli muutamassa viikossa laajentunut yllättävän monipuoliseksi, vaikka sitä ei pinnalle päin heti huomannutkaan.

4.5 Huomioita prosessista

Pienen prototyyppiohjelman kehittäminen oli valtavan hyödyllistä, ja se tuotti tukun tärkeitä oivalluksia siitä, miten sovellusaluevetoista suunnittelua tehdään ja mikä on GraphQL:n merkitys prosessissa.

Mallia kehittäessä vaadittujen voimakkaiden refaktorointijaksojen määrä yllätti. Ennalta kirjallisuudesta luettuna ei refaktoroinnin määrää ollut helppo hahmottaa. Omakohtainen tekeminen paljasti, miten oleellinen osa Sovellusaluevetoista suunnittelua refaktorointien tekeminen on.

Refaktoroiminen ei ole tässä tyylissä pelkästään tekninen keino pitää koodia siistinä, vaan strateginen väline, jonka avulla mallia parannetaan ja kaikenkattavaa kieltä kehitetään.

Toinen keskeinen keino kaikenkattavan kielen kehittämiseen tässä prosessissa oli suunnittelutapaamistemme kielen tarkka seuraaminen. Pyrin nappaamaan Lauran kanssa käydyistä keskusteluista termejä, joita käytimme, ja etenkin termejä, joita Laura käytti.

Eric Evans mainitsee, että kaikenkattavan kielen rakentamisessa oleellista on löytää sanat, joita alan asiantuntijat käyttävät. Etenkin puuttuvien käsitteiden tunnistaminen puheen seasta auttaa paljon mallin parantamisessa. [3.]

4.6 Sovellusaluevetoisen suunnittelun käsitteiden hyödyntäminen

Käytin tietomallin koodia rakentaessani apuna Evansin esittelemiä käsitteitä. Useat käsitteet, kuten käynnit ja laskut, kuvasin yksilötyyppeinä. Käsitteistä koostuvat kokonaisuudet ovat aggregaatti-rakenteissa. Esimerkiksi laskun sisältämät laskurivit tai myynnin sisältämät myyntiosuudet.

Koska yksinkertaisessa prototyyppisovelluksessani ei ole ollenkaan tietokantaa, toteutin käsitteille Evansin mallin mukaisesti repositoriot. Käytännössä ne ovat vain yksinkertaisia luokkia, jotka pitävät sisällään linkitetyn listan olioita.

Laskujen luominen puolestaan oli operaationa niin monimutkainen, että siihen tarvitsin Eric Evansin esimerkin mukaisesti erillisen tehdasluokan. Senkin jälkeen operaatio oli melko monimutkainen. Vielä prototyyppiohjelman kehityksen päättyessä minulla oli vahva tunne, että tehdasluokkaan jääneet sotkuiset koodin osat olivat seurausta jostakin puuttuvasta käsitteestä.

4.7 GraphQL-rajapinnan ja sovellusaluemallin yhteys

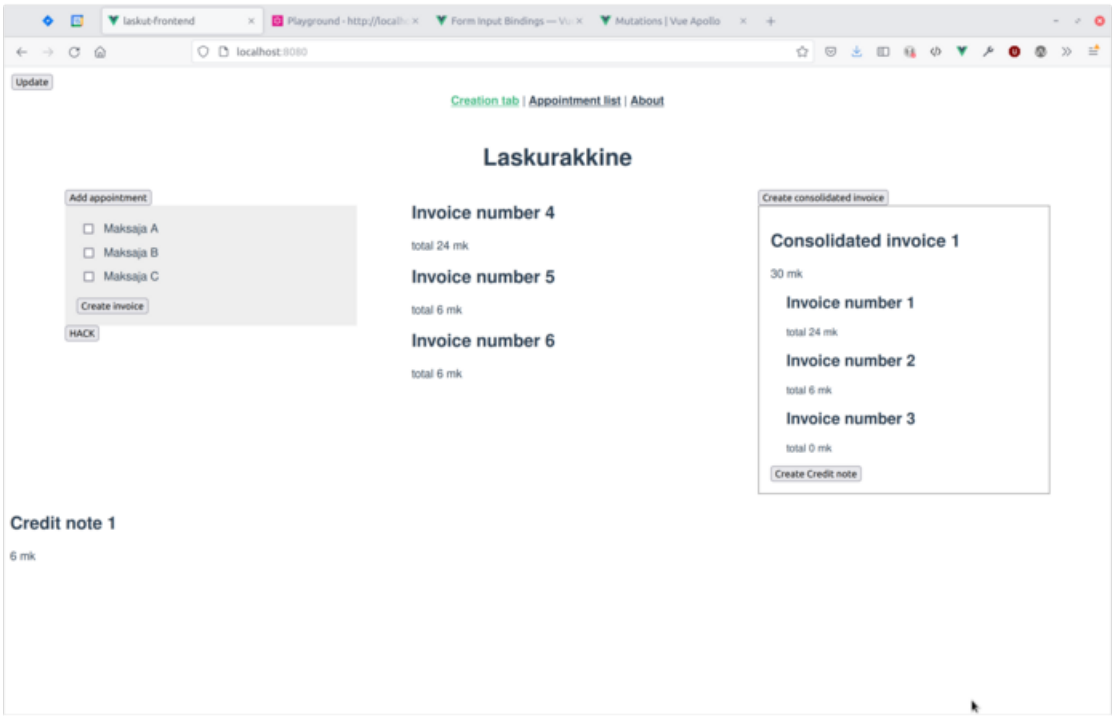
GraphQL-skeemat tuntuivat heijastavan todella osuvasti rakentamiamme käsitekarttoja. Useimmiten oli mahdollista siirtää rakentamamme käsitteiden verkko sellaisenaan GraphQL-skeeman sisälle. Tein tämän monesti ensimmäisenä osana ohjelmointijaksoa. Tämä myös tarkoitti, että palvelinohjelman ja asiakasohjelman välinen rajapinta sai ensimmäisenä kyvyt uuden käsitekartan version käsittelemiseen.

Koska GraphQL on riippumaton käytetystä ohjelmointikielestä, myös malli irtautui niistä välittömistä teknologioista, joilla prototyyppisovelluksen rakensin. Voidaan siis sanoa, että GraphQL-rajapintakehityksessä sovellusalueen käsitteet saavat merkittävämmän roolin kuin alla käytettävät täsmälliset teknologiavalinnat.

Huonommin GraphQL-rajapinta kuvasi mallin dynaamisia muutoksia. Mutaatioiden avulla voidaan ilmaista, minkälaisia toimintoja malliin voidaan kohdistaa, ja toiminnon onnistuminen voidaan toki havaita muuttuneena mallina. Rajapinnasta ei kuitenkaan suoraan päällepäin näe, missä ovat mallin dynaamiset nivelkohdat.

Esimerkiksi laskutusta käsittelevässä mallissa tällainen nivelkohta on käynnin ja myynnin välissä. Kun käynti muuttuu myynniksi laskulle lisättäessä, tapahtuu käsitteellinen muutos, joka antaa prototyyppisovellukselle sen sisältämän voiman ja joustavuuden.

GraphQL-rajapinta myös lisäsi yhden ylimääräisen tason monimutkaisuutta: skeemaa piti päivittää erikseen, mutta pelkillä skeeman muutoksilla ei vielä ol-



Kuva 6: Laskujen lisäysnäkymä

lut mahdollista selvittää mallin toimivuutta. Pelkästään ohjelmakoodissa toteutettu sovellusaluemalli olisi ollut yksinkertaisempi muokata. Toisaalta myös REST-rajapinta olisi todennäköisesti tuonut vastaavaa monimutkaisuutta, kun sovellusaluemalli olisi pitänyt esittää joukkona resurseja.

5 Tulokset

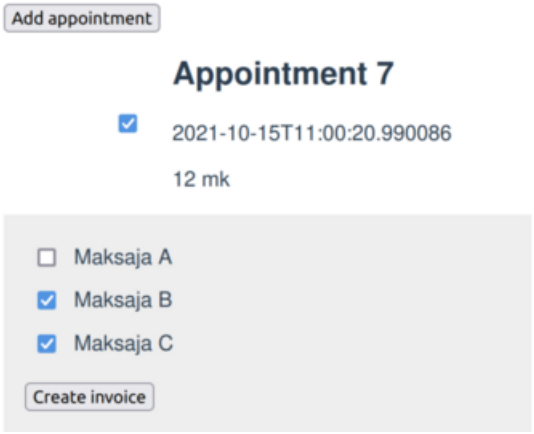
5.1 Prototyyppiohjelman esittely

Prototyyppiohjelma on yksinkertainen yhden sivun selainsovellus. Sen avulla voi luoda käyntejä, laskuttaa niitä, koota laskuja koontilaskuiksi ja hyvittää yksittäisiä laskurivejä.

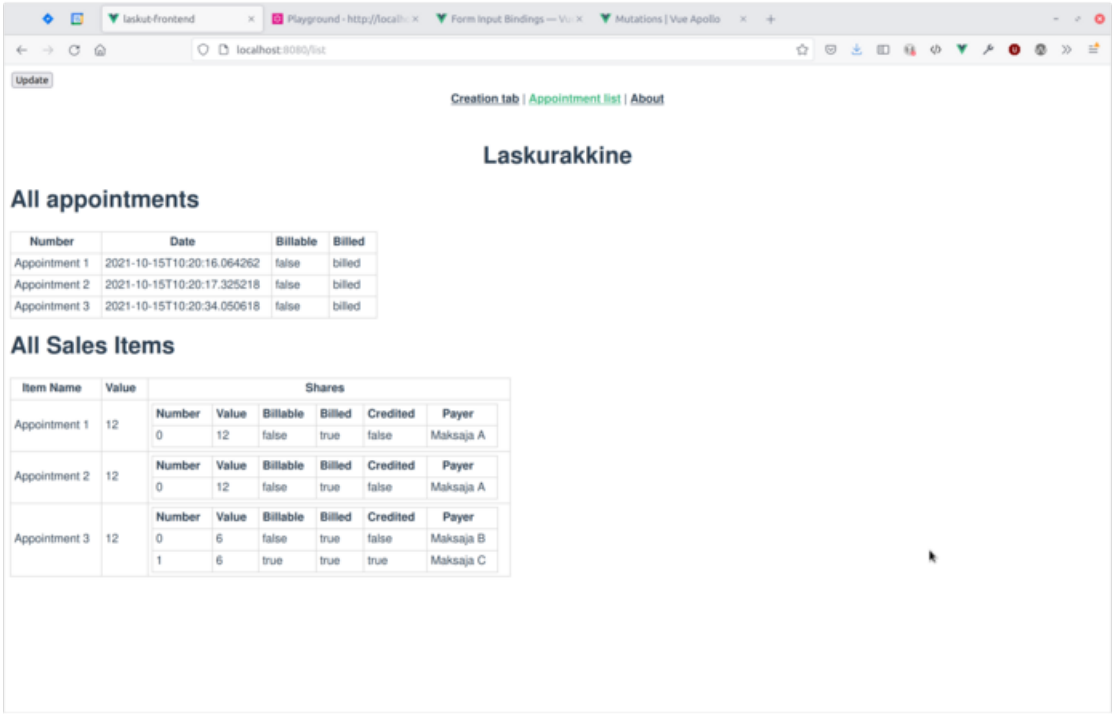
Ohjelman pääkäyttöliittymä on esitetty kuvassa 6

Laskujen sisältöjä voi tarkastella, ja ohjelma laskee laskujen avoimet summat.

Lisäksi käyttöliittymästä voi valita maksajan luotavalle laskulle. Mikäli maksajia



Kuva 7: Esimerkki käynnin jakamisesta usealle maksajalle

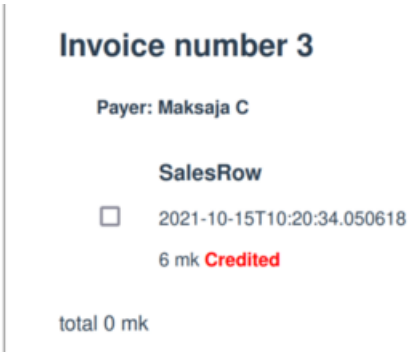


Kuva 8: Ohjelman listanäkymä

valitaan useampia, ohjelma jakaa käynnit kahdelle eri maksajalle. Tämä on esitetty kuvassa 7.

Erillisessä listanäkymässä (kuva 8) voi tarkastella luotujen käyntien tilaa sekä laskutettavan myynnin tilaa. Ohjelma näyttää, onko käynti laskutettu vai laskuttamaton. Myynnin osalta ohjelma näyttää, miten myynti jakautuu eri maksajille, ja onko summa avoin, laskutettu vai hyvitetty.

Kuvassa 9 on esitetty miten ohjelma näyttää hyvitetyn laskurivin.



Kuva 9: Ohjelma näyttää, että yksittäinen laskurivi on hyvitetty

Käytin asiakasohjelman tekemiseen niin vähän aikaa kuin mahdollista. Se näkyy tyylin hiomattomuutena. Luonnosmaisen näköinen ulkoasu myös kommunikoi muille prosessiin osallistuville, että ohjelmisto tai etenkään sen käyttöliittymä ei ole tarkoitettu tuotantokäyttöön, vaan apuvälineeksi erilaisten tietomallin piirteiden kartoittamiseen.

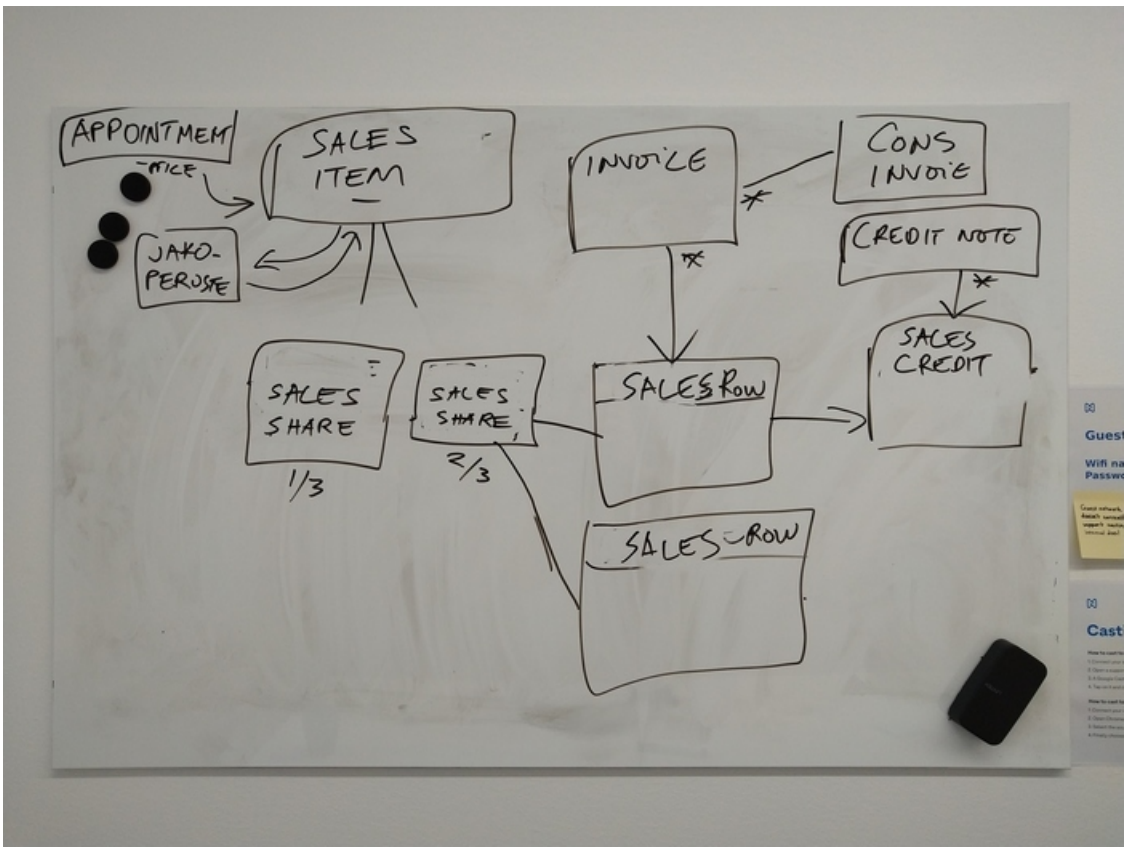
5.2 Parannuksia tietomalliin

Viiden viikon aikana syntyi pieni malli laskutukseen tietorakenteen parantamiseksi. Lisäksi löytyi kaksi pientä ideaa, joita voi hyödyntää, kun ohjelmistoa kehitetään.

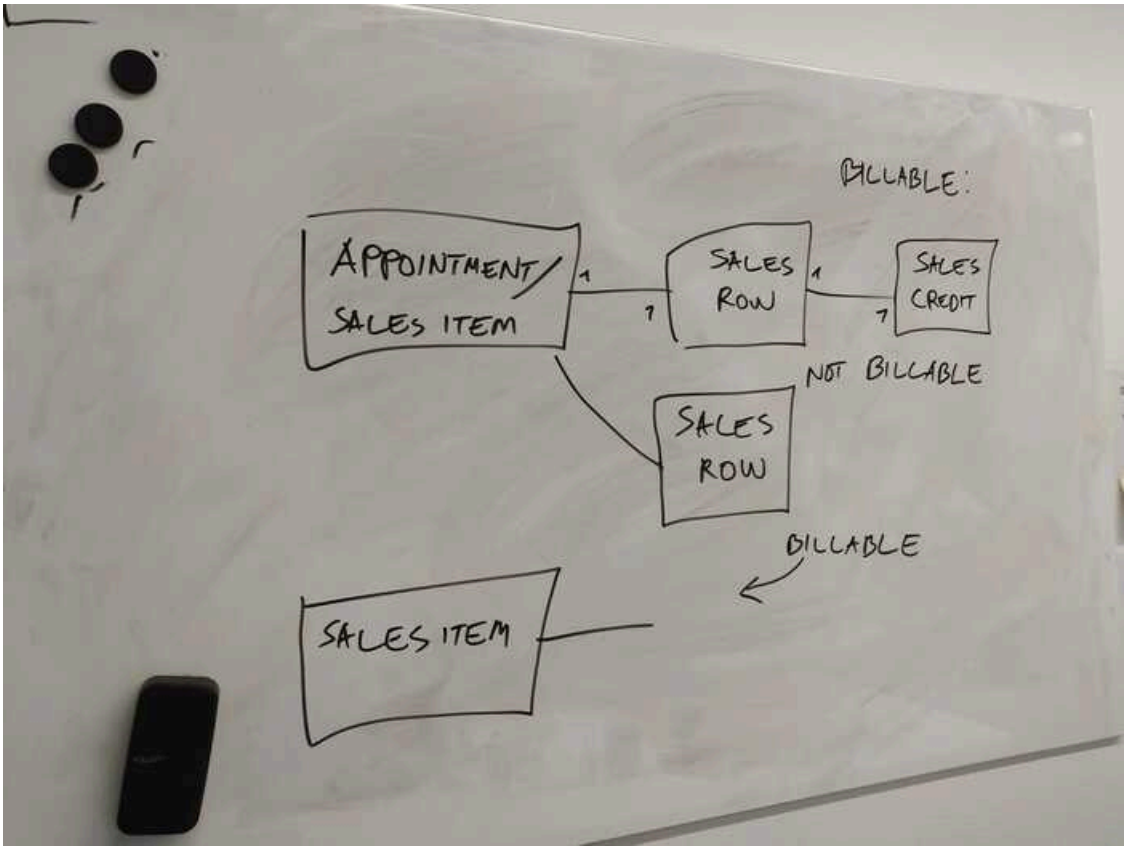
Pieni malli laskutuksen parantamiseksi on esitetty kuvassa 10. Se pitää sisällään ajatuksen käynnin muuttumisesta myynniksi, kun se saapuu laskutuksen piiriin. Oma erikoisuutensa on myös käynnin jakamiseen liittyvä jakoperuste.

Prototyypissä emme ottaneet kantaa, millä perusteella käynnin hinnan osittaminen eri maksajille tapahtuu. Käytännössä ohjelmassa on eri maksajatahojen kanssa tehtyjä sopimuksia, jotka määrittelevät ehtoja maksuosuuden suuruudesta. Jakoperuste on siis dynaamisesti muuttuva käsite, joka riippuu haluttuihin maksajiin ja käyntiin kytkettyyn hoitojaksoon yhdistyvistä sopimuksista.

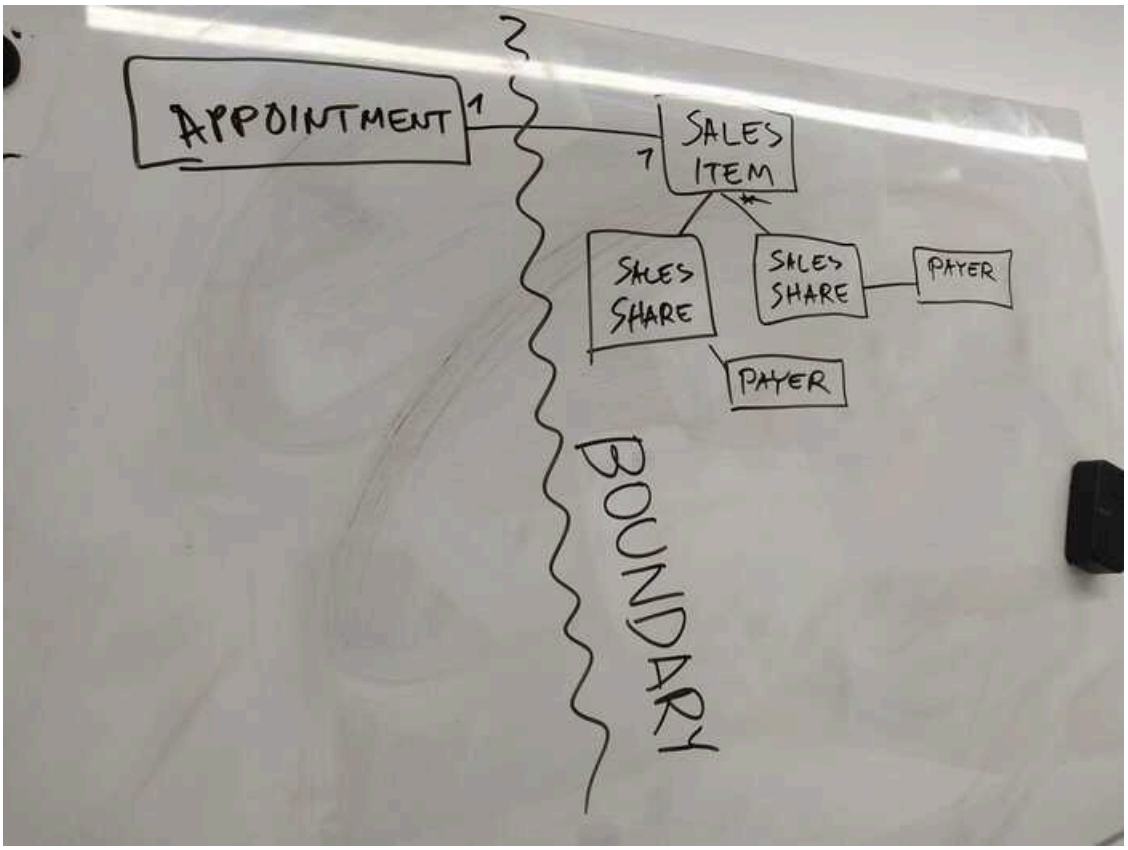
Kaksi pientä ideaa ovat molemmat käyttökelpoisia erillään mallista. Ensimmäinen niistä on myynnin, myyntirivin ja hyvitysrivin välinen tiivis yhteysketju. Tämä idea (kuva 11) mahdollistaa hyvin yksinkertaisen ja joustavan myynnin laskutus-



Kuva 10: Lopullinen malli



Kuva 11: Idea 1



Kuva 12: Idea 2

ja hyvityslogiikan.

Toinen pieni idea on, että käynti kannattaisi erottaa selkeästi laskulle tulevasta myynnistä. Tällöin on mahdollista myös esimerkiksi vaihtaa myöhemmin maksajaa, jolta käynti laskutetaan ilman, että jo muodostettuihin laskuihin tarvitsee kajota. Tämä ajatus on esitetty kuvassa 12.

6 Kuvaus työmallista

Tämän työn kuluessa syntyi työmalli, jota kutsun *notkeaksi tietomallin parantaluksi*. Työmalli on parhaimmillaan tilanteissa, joissa ohjelmiston sovellusaluemalli sisältää monimutkaisia ja vain erityisalan asiantuntijalle aukeavia käsitteitä ja käsitesuhteita. Eduksi on myös, mikäli sovelluksen tekninen vaativuus ei ole kovin poikkeuksellinen. Tyypillinen bisnessovellus, jossa painopiste on sovellusalueessa, on hyvä kohde tälle työmallille.

Tämän työmallin pääperiaate on, että **sanat, kaaviot ja koodi** ovat kolme kommunikaation muotoa, jotka täydentävät toisiaan. Kuvaan seuraavassa tämän työmallin keskeiset ominaispiirteet.

Lyhyet iteraatiot

Lyhyet iteraatiot, joiden välissä pidetään suunnittelutapaaminen, ovat ehdoton edellytys tietomallin ripeälle kehittämiselle. On tärkeää, että iteraation kuluessa syntyy käyttökelpoinen ohjelmistoversio, jonka avulla mallin toimivuutta voidaan testata ja todentaa.

Keskusteleva suhde tuoteomistajan ja kehittäjän välillä

Koska tavoitteena on luoda kieli, jota voivat käyttää niin ohjelmoijat kuin liiketoimintaväkikin, sen kehittämiseen luontevin ja todennäköisesti ainoa tapa on rakentaa mallia keskustelevalla otteella.

Käyttäjätarinoihin pohjautuva työlista

Ketterän kehityksen työkalupakista peräisin oleva ajatus yksinkertaisista käyttäjätarinoista soveltuu hyvin tietomallin kehittämisen lähtökohdaksi. Kun huomion keskipisteenä ovat ne asiat, joita käyttäjä voi ohjelmistolla tehdä, on myös syntyvä malli lähempänä alan realismia.

Keskittyminen rajapinnan rakenteeseen ohjelmiston rakenteen sijasta

Keskeinen kysymys työtä tehtäessä on, ilmaiseeko rajapinta sovellusalueen ja ongelmakentän riittävän monipuolisesti. Teknisiin yksityiskohtiin, ohjelmointikieliin ja kirjastoihin keskittymisen sijasta huomio kannattaa pitää juuri rajapinnan ilmaisemassa käsiteverkossa.

Rajapintaskeeman rakentaminen käsitteiden pohjalta

GraphQL-rajapintaskeema kannattaa rakentaa ennenkuin kirjoittaa skeeman toteuttavaa koodia. Näin kielen käsitteet ja niiden väliset suhteet tulevat formaalisti ilmaistuksi. Ohjelmakoodin kirjoittamisen aikana skeema saattaa myös tarkentua, ja silloin kannattaa muutokset tehdä välittömästi.

Koodin ja mallin pitäminen lähekkäin

Välttämätön osa tätä työtyyliä on koodin ja mallin vastaavuus. Mallissa käytettävät käsitteet on löydyttävä koodista, ja koodissa tulisi olla lähinnä vain nämä käsitteet ja niiden väliset suhteet sellaisina, kuin ne kaikenkattavassa kielessä ilmenevät.

Voimakkaat refaktoroinnit

Voimakkaat refaktoroinnit ovat keino muokata koodin esittämästä mallista joustava ja ilmaisuvoimainen. Nämä refaktoroinnit vaativat ehdottomasti tuekseen jämerän yksikkötestisetin. Sen rakentaminen onnistuu käytännössä vain testit edellä tekemällä.

Ohjelmoinnissa vastaan tulleet ongelmat jatkosuunnittelun lähtökohtina

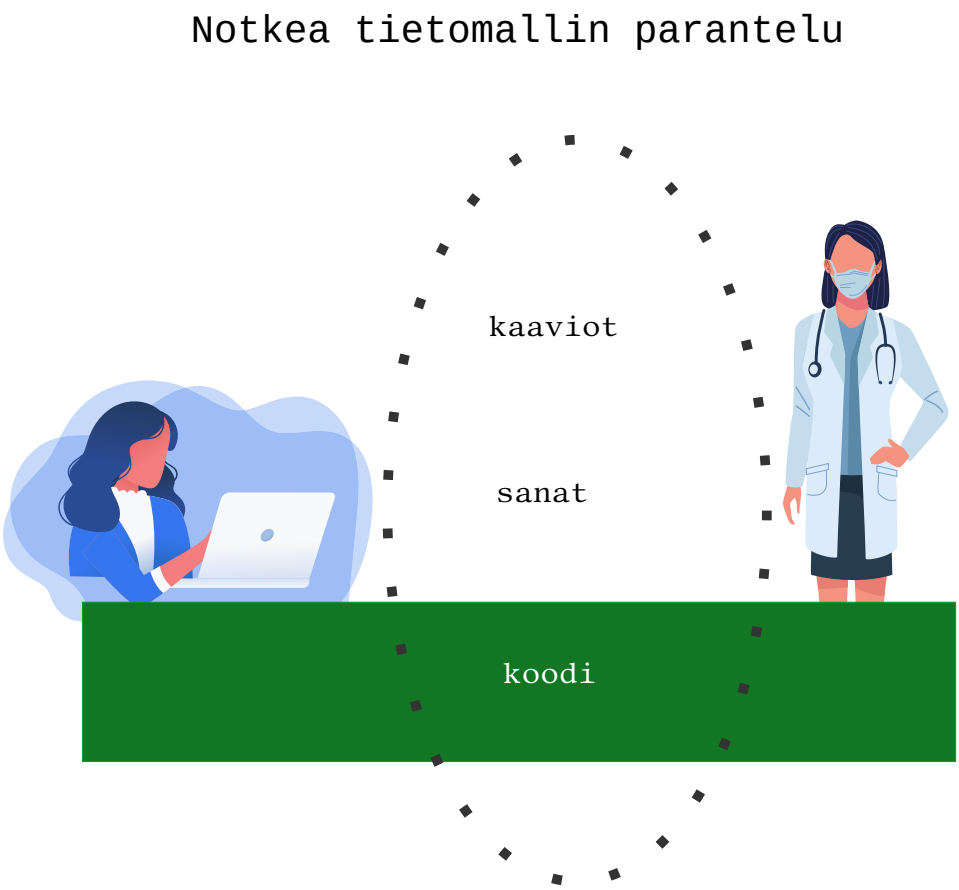
Suunnittelutapaamisten ja ohjelmointiprosessin välinen yhteys ei saa olla vain yksisuuntainen. Mikäli suunnittelutapaaminen nähdään ainoana osana prosessia, jossa suunnittelua tapahtuu ja ohjelmointi pelkästään suunnitelmien mekaanisena toteuttamisena, hyödyt tästä työskentelytyylistä jäävät hyvin vähäisiksi. Ohjelmointiprosessi on suunnittelutyön toisenlainen vaihe, ja siinä ilmenevät ongelmat ovat oivallinen maaperä seuraavan suunnittelutapaamisen aiheiksi.

Esittelen työmallin kuvassa 13.

6.1 Työmallin vahvuuksia

Tämän työmallin tuloksena syntyvä ohjelmisto tai ohjelmiston osa on notkea ja helposti muokattavissa. Lisäksi sen rakenne mukailee hyvin läheisesti sovellusalan sisäistä logiikkaa. Tämä läheinen yhteys tekee helpoksi tietomallin jatkokehittämisen, kuten uusien ominaisuuksien lisäämisen tai olemassaolevien muuttamisen. Kun sovellusalan toimintatavat muuttuvat, vastaa ohjelmistoon tehtävän muutoksen suuruus sovellusalalla tapahtuvan muutoksen suuruutta.

Tiheän kehityssyklinsä vuoksi tämä työmalli on myös suhteellisen hallittava ja ennustettava: ohjelmiston toimintojen tila on liiketoimintaeksperttien tutustuttavissa viikoittain. Tätä tietoa on mahdollista käyttää jäljellä olevan työmäärän arvioinnissa. Tiheä kehityssykli ja käyttäjätarinoihin pohjaava työlista myös suojaavat tarpeettoman työn tekemiseltä. Prosessissa syntyy lähinnä sellaista koodia,



Kuva 13: Kuva työmallista

joka tarvitaan käyttäjälle näkyvien ominaisuuksien toteuttamiseen.

Koska työtyyli on keskusteleva ja ohjelmiston rakenne pyritään pitämään avoimena muutoksille, on muutoksien tekeminen mahdollista koko kehitysprosessin ajan, ja tarvittaessa vielä hyvinkin loppuvaiheessa prosessia. GraphQL-rajapintaskeeman kehittäminen on hyvä tapa dokumentoida tätä jaettua kieltä ja erottaa kielen käsitteet muusta koodista.

6.2 Työmallin haasteita

Tämä työmalli vaatii ohjelmoijalta paljon. Se edellyttää jatkuvaa kiinnostusta sovellusalueen piirteistä ja laajaa tarkkaavuutta suunnittelutapaamisten keskusteluissa. Lisäksi se edellyttää kykyä sietää epävarmuutta ja muuttaa suunnitelmia usein ja isosti. Teknisellä tasolla työmalli edellyttää kykyä joustavan ohjelmiston

suunnittelemiseen laajojen refaktorointien tekemiseen kireässä aikataulussa pysyen ja tiukkaa keskittymistä niihin päämääriin, jotka mallin kehittämisessä on kulloinkin asetettu.

Jotta tällainen työmalli voi olla hedelmällinen tuotantotasoisen ohjelmiston tekemisessä, se edellyttää myös, että työtyyli ohjelmoijan ympärillä vastaa tällaista tekemisen tapaa. Liiketoimintavetoisella suunnittelulla on vaikutuksia paitsi ohjelmistotuotannon prosessiin, myös sitä ympäröiviin prosesseihin, kuten asiakastarpeiden kokoamiseen ja tulevan kehitystyön suunnittelemiseen.

6.3 Työmallin vaihtoehtojen punnitsemista

Monimutkaisten sovellusalueiden kanssa työskenteleminen on haaste joka tapauksessa. Olisi houkuttelevaa ajatella, että tällaisen monimutkaisuuden hallitseminen, asiakkaan tarpeiden ymmärtäminen ja tietomallin hahmotteleminen kuuluisi jollekulle muulle kuin ohjelmoijalle. Tällöin liiketoiminta-alasta ymmärtävä henkilö voisi kirjoittaa määrittelydokumentin, ja ohjelmoijan tehtäväksi jäisi määrittelyn mukaisen ohjelmiston toteuttaminen. Tällaisen työtavan haasteena kuitenkin on, että koodi ja määrittely voivat erkaantua nopeastikin toisistaan. Käytettävään ohjelmointiympäristöön liittyvät rajoitteet saattavat estää määrittelyn mukaisen mallin toteuttamisen. Tällöin malli lohkeaa kahdeksi: määrittelyn mukainen, tarkkaan suunniteltu malli jää paperille, ja koodiin päätyy kehitystyön ohessa nopeasti improvisoitu vaihtoehtoinen versio. [3]

Toinen vaihtoehto on ketterä, iteratiivinen kehitystyyli ilman kaikenkattavaa kieltä. Tässä prosessissa käyttäjävaatimuksia toteutetaan viikko viikolta, ja kehittäjät saavat iteraation päätyttyä palautetta siihen mennessä valmiiksi saadun ohjelmiston pohjalta. Yhteistä kieltä ei kuitenkaan pyritä rakentamaan, eikä kehittäjien edellytetä kirjoittavan sovellusalueen logiikkaa mukailevaa koodia. Riskinä on, että kehitystyön kuluessa ilmaantuu sellaisia käyttäjävaatimuksia, joita on hankala sovittaa olemassaolevaan ohjelmistoon. Koska koodi ei noudata yhteistä jaettua kieltä, voi kehittäjien koodiin kirjoittama malli poiketa suurestikin siitä, miten sovellusalueen ekspertit asiat ymmärtävät. [3]

Molemmissa edellä esitetyissä vaihtoehtoisissa tyyleissä kehitystahti hidastuu ja työ vaikeutuu prosessin edetessä, mikäli esitetyt riskit toteutuvat. Myös riski koodin laadun heikentymisestä on olemassa, mikäli ohjelmistoon joudutaan nopeasti tekemään muutoksia uusien käyttäjävaatimusten pohjalta.

7 Yhteenveto

Tässä insinöörityössä pyrittiin kohentamaan ohjelmiston tietomallia kehittämällä GraphQL-rajapinta sovellusaluevetoisen suunnittelun keinoin. Tavoitteena oli parantaa nykyistä tietomallia ja luoda työprosessi tietomallin parantelemiseen. Samalla etsittiin vastausta kysymykseen, miten hyvin GraphQL-rajapinta teknologiana sopii tällaiseen prosessiin.

Saadakseni vastauksia kysymyksiin laadin pienen prototyyppisovelluksen, jonka tehtäväksi asetimme yhdessä tilaajan edustajien kanssa laskutukseen liittyvän konkreettisen ongelman ratkaisemisen. Keskeiset vastaukset työn nostamiin kysymyksiin tulivat juuri tämän sovelluksen kehitysprosessin kautta. Samalla kehitystyö muovasi lopullista työprosessia.

Tehdyn kokeilun perusteella GraphQL-rajapinta soveltuu hyvin sovellusaluevetoisen suunnittelun tarpeisiin. Tämä johtuu sen verkkomaisesta luonteesta, jolla on helppo mallintaa sovellusalueen käsitteiden keskinäisiä suhteita. Se, että GraphQL-verkko on nimenomaan rajapinta, helpottaa käsitteellisen mallin erottamista omaksi kokonaisuudekseen sovelluksen sisällä, ja irrottaa mallin konkreettisesta teknologiasta.

Koska GraphQL-rajapinta määritellään täsmäkielen avulla, mallia on helppo muokata osana iteratiivista kehitysprosessia. Tämä mahdollistaa tutkimusmatkat ja kokeilut erilaisilla malleilla, kun käsitteiden välisiä suhteita voidaan muuttaa ensiksi skeemassa, ja vasta sitten taustalla olevassa koodissa.

Projektin aikana löysin kohteeksi valitun laskutuksen ongelman ratkaisevan tietomallin, ja luonnostelin *notkean tietomallin parantelun* periaatteet. Pääperiaate

on, että **sanat, kaaviot ja koodi** ovat kolme tapaa kommunikoida tietomallista kehittäjien ja liiketoimintaihmisten välillä.

Tietomallin toteuttaminen olemassaolevassa ohjelmistossa oli rajattu jo alunperinkin tämän projektin ulkopuolelle, ja se vaatisi vielä lisätyötä ja suunnittelua. Syntynyttä GraphQL-rajapintaskeemaa olisi mahdollista käyttää jatkotyön pohjana, ja näin parantaa myös vanhan ohjelmiston sisäistä logiikkaa.

GraphQL-kieli on pohjimmiltaan melko yksinkertainen, mutta joitain sen ominaisuuksia jäi tässä kartoittamatta. Esimerkiksi kielen tarjoamat Input Typet, jotka mahdollistavat tallennettavan datan esittämisen olioverkkona rajapintakyselyssä, sekä Union Typet, jotka tarjoavat tuen polymorfismille, jäivät tässä vaiheessa kartoittamatta. Jatkokysymykseksi siis jää, miten nämä monimutkaisemmat ominaisuudet niveltyvät yhteen sovellusaluevetoisen suunnittelun kanssa.

Ehkä keskeisin johtopäätös insinöörityöstä on kuitenkin alalla laajasti ja eri muodoissa toistettu näkemys. Olipa esittäjänä Osmo A. Wiio (“viestintä epäonnistuu, paitsi sattumalta”), Gerald Weinberg (“No matter how it looks at first, it’s always a people problem.”) tai Melvin Conway (“Organizations, who design systems, are constrained to produce designs which are copies of the communication structures of these organizations.”), lopulta ajatus kiertyy samaan johtopäätökseen: ohjelmistossa esiintyvät ongelmat eivät useinkaan johdu teknologiasta vaan ongelmista ihmisten välisessä kommunikaatiossa.

Lähteet

1 Veterinary Software for animal clinics and hospitals | Provet Cloud lokakuu 2021. [Online; accessed 31. Oct. 2021]. Verkkoaineisto. <https://www.provet.cloud>.

2 Diarium – potilastietojärjestelmä ja laskutusohjelma terapeuteille lokakuu 2021. [Online; accessed 31. Oct. 2021]. Verkkoaineisto. <https://www.diarium.fi>.

3 Evans, Eric. 2004. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

4 Beck, K. & Andres, C. 2004. Extreme Programming Explained: Embrace Change. Addison-Wesley.

5 Martin, Robert C. 2008. Clean code: a handbook of agile software craftsmanship. Prentice Hall.

6 Jeffries, Ron. Huhtikuu 1998. You're NOT gonna need it! [Online; accessed 21. Oct. 2021]. Verkkoaineisto. <https://ronjeffries.com/xprog/articles/practices/pracnotneed>.

7 Fielding, Roy Thomas. 2000. Architectural Styles and the Design of Network-based Software Architectures. University of California, Irvine. <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

8 Prisma. Ei julkaisupäivää. GraphQL vs REST - A comparison. [Online; accessed 19. Oct. 2021]. Verkkoaineisto. <https://www.howtographql.com/basics/1-graphql-is-the-better-rest>.

9 Why use GraphQL? Marraskuu 2020. [Online; accessed 19. Oct. 2021]. Verkkoaineisto. <https://www.apollographql.com/blog/graphql/basics/why-use-graphql>.

10 Facebook. 2018. GraphQL Spec. Verkkoaineisto. <spec.graphql.org>. Luettu 06. 10. 2021.

11 GraphQL | A query language for your API ei julkaisupäivää. [Online; accessed 30. Oct. 2021]. Verkkoaineisto. <https://graphql.org>.

12 Apollo GraphQL ei julkaisupäivää. [Online; accessed 30. Oct. 2021]. Verkkoaineisto. <https://www.apollographql.com>.

13 GraphQL schema basics ei julkaisupäivää. [Online; accessed 19. Oct. 2021]. Verkkoaineisto. <https://www.apollographql.com/docs/apollo-server/schema/schema/#naming-conventions>.

14 Pozrikidis, Constantine. 2014. An Introduction to Grids, Graphs, and Networks. Oxford University Press.

15 Booch, Grady; Maksimchuk, Robert A; Engel, Michael W; Young, Bobbi J; Conallen, Jim & Houston, Kelli A. 2008. Object-oriented analysis and design with applications. Vol. 3. Addison-Wesley.

16 Foundation, GraphQL. Ei julkaisupäivää. Thinking in Graphs | GraphQL. [Online; accessed 16. Oct. 2021]. Verkkoaineisto. <https://graphql.org/learn/thinking-in-graphs>.

17 Cardelli, Luca & Wegner, Peter. 1985. ”On Understanding Types, Data Abstraction, and Polymorphism”. ACM Computing Surveys 17.4, s. 471–522.

18 graphql. Lokakuu 2021. graphql-js. [Online; accessed 16. Oct. 2021]. Verkkoaineisto. <https://github.com/graphql/graphql-js>.

19 Glossary — Python 3.10.0 documentation lokakuu 2021. [Online; accessed 16. Oct. 2021]. Verkkoaineisto. <https://docs.python.org/3/glossary.html#term-duck-typing>.

20 Landin, P.J. 1966. ”The next 700 programming languages”. Communications of the ACM 9.3, s. 157–166.

21 Eric S. Raymond. 2003. The Art of UNIX Programming. Addison-Wesley. <http://catb.org/~esr/writings/taoup/html/>.

22 Spinellis, Diomidis. Helmikuu 2001. ”Notable Design Patterns for Domain Specific Languages”. Journal of Systems and Software 56.1, s. 91–99. <http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html>.

23 Using A Schema-First Design As Your Single Source of Truth | Nordic APIs | marraskuu 2017. [Online; accessed 16. Oct. 2021]. Verkkoaineisto. <https://nordicapis.com/using-a-schema-first-design-as-your-single-source-of-truth>.

24 99designs heinäkuu 2021. [Online; accessed 16. Oct. 2021]. Verkkoaineisto. <https://99designs.com/blog/engineering/schema-driven-development>.

25 eu, Consultancy. Toukokuu 2020. ”Half of companies applying Agile methodologies & practices”. Consultancy.eu. <https://www.consultancy.eu/news/4153/half-of-companies-applying-agile-methodologies-practices>.

26 What is an Iteration? Ei julkaisupäivää. [Online; accessed 31. Oct. 2021]. Verkkoaineisto. <https://www.agilealliance.org/glossary/iteration/>.

27 The Falcon Web Framework — Falcon 3.0.1 documentation ei julkaisupäivää. [Online; accessed 30. Oct. 2021]. Verkkoaineisto. <https://falcon.readthedocs.io/en/stable>.

28 Welcome to Flask — Flask Documentation (2.0.x) ei julkaisupäivää. [Online; accessed 30. Oct. 2021]. Verkkoaineisto. <https://flask.palletsprojects.com/en/2.0.x>.

29 Ariadne · Python GraphQL Schema-first ei julkaisupäivää. [Online; accessed 30. Oct. 2021]. Verkkoaineisto. <https://ariadnegraphql.org>.

30 Graphene-Python ei julkaisupäivää. [Online; accessed 30. Oct. 2021]. Verkkoaineisto. <https://graphene-python.org>.